Software Design: Final Report

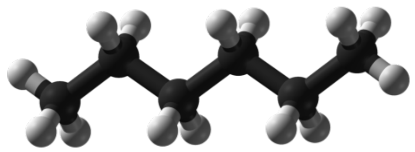
Samantha Kumarasena, Emily Tumang, Claire Barnes, Anne Wilkinson

Github:
Our entire project so far can be found at: https://github.com/skumarasena/MolecularModeling.git

Project Proposal: 3D molecular modeling
Description: Translate structural formulas into a 3D visualization of a molecule. Given a standard structural formula (*simple example below*):

Hexane: CH3CH2CH2CH2CH2CH3

we would be able to produce a diagram similar to:



Maximum Deliverable: Be able to display a 3D model of a complex molecule given a structural formula, allowing the user to rotate and interact with the molecular model. Handle molecules with double and triple bonds. Handle molecules with nonzero formal charges. Handle molecules with expanded octets/abnormal valence (which is very difficult, as these atoms do not have a tetrahedral geometry as normal atoms would!)

Minimum Deliverable: Be able to display a 3D representation of a simple (i.e. inorganic molecules and basic carbon chains), single-bonded molecule given a structural formula.

First Step:
The first step in this project will probably be building the basic structure of a single central atom. Bonding atoms and lone pairs are arranged around nuclei in a tetrahedral pattern with a bond angle of 109.5 degrees. Implementing this basic structure will probably be the first part of this project. We would also write the functions and classes required for this basic structure.

Design Proposal:
Visualization Methods:
In order to begin work on our project, we needed to find a way of visualizing our code in 3-D. We narrowed our options down to three different visualization methods: VPython, Unity, and Blender. This choice was important to make early on, as this decision affects how we work, and how far we can take our project. After talking with Julian and doing some research, we were able to compare the pros and cons of each option. VPython is a Python module, making it simple and easy to use. Conversely, it does not produce very attractive results, it is difficult to build complex structures, and it is slow. Since we want to build upon this project as much as we can, these constraints caused us to rule out vpython.

Another option was Unity. Unity is a program commonly used in game development in order to make 3-D graphics. Unity is aesthetic and has lots of documentation, but unfortunately can only be edited in Windows and OSX, which means we may have to switch between Ubuntu and Windows fairly often during development. Unity's scripting is in a language similar to Python called Boo. However, after doing some research on Boo, we discovered that there were more differences from Python than we initially thought.

We decided to use Blender because not only does it run and edit in all platforms, but it also scripts in Python. The main advantage Unity has over Blender is that there is more documentation, but one of our team members already has a small amount of experience in Blender, so we felt that this would not be an issue.

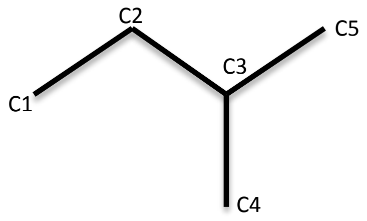
Data Structures:
The user will input a structural formula in the form of a string. This string will be parsed, and the hydrogen atoms will be removed (hydrogen atoms only serve to ensure that an atom's formal charge is zero, so we automatically know how many hydrogen atoms are bonded to any given atom. They can be removed to simplify the data structure). For example, for an input 'CH3CH2NH2', the resulting string would be 'CCN'.

From this string, we will create 'Atom' objects that represent each atom in the chemical formula. (See the attached class hierarchy diagrams and sample usages.)The 'Atom' class will have sub-classes for each element, describing how many electron pairs and bonds each atom must have (as all atoms in our simulation must have zero formal charge). For example, a carbon atom must have four bonds. A nitrogen atom must have three bonds and one lone electron pair. An oxygen atom must have two bonds and two lone electron pairs. We plan to have classes for carbon, oxygen, and nitrogen (and possibly phosphorus and sulfur), as the vast majority of organic compounds can be constructed with these atoms.

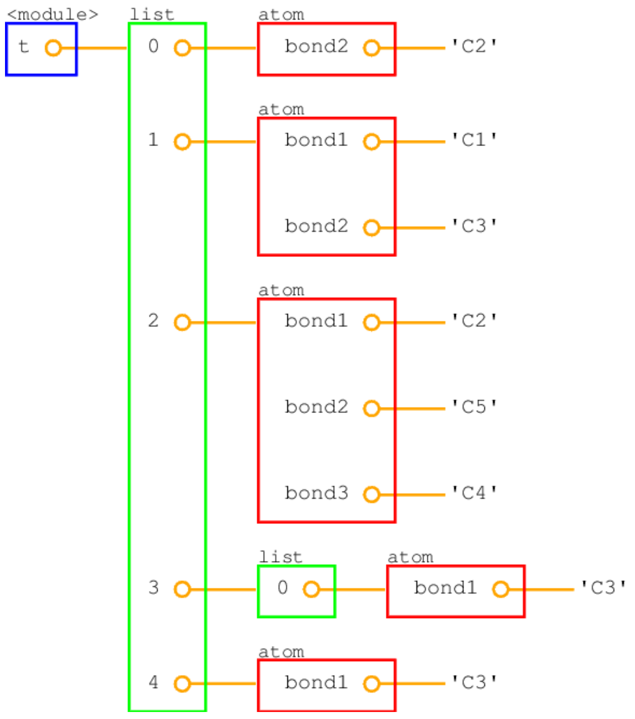
The 'Atom' superclass will have a 'name' field, a string that represents the name of the atom in element-number format. For example, the compound CH3CH2NH2 would be made up of three atoms with the names C1, C2, and N3. The 'Atom' superclass will also have four bond variables. (Additionally, the 'Atom' superclass could have an 'angle' variable, which would change if we decided to implement double and triple bonds.)

The element subclasses (C, N, O, and possibly P and S) assign lone pairs/hydrogen atoms to the bond variables inherited from the superclass 'Atom'.

In order to display the molecule properly, we needed to figure out how to organize the 'Atom' objects in an appropriate data structure. However, we found that it is difficult to represent branched chemical structures in terms of Python data structures. We decided that there were two main options we could follow. Diagrams represent the molecule CH3CH2CH(CH3)CH3, with no hydrogen.

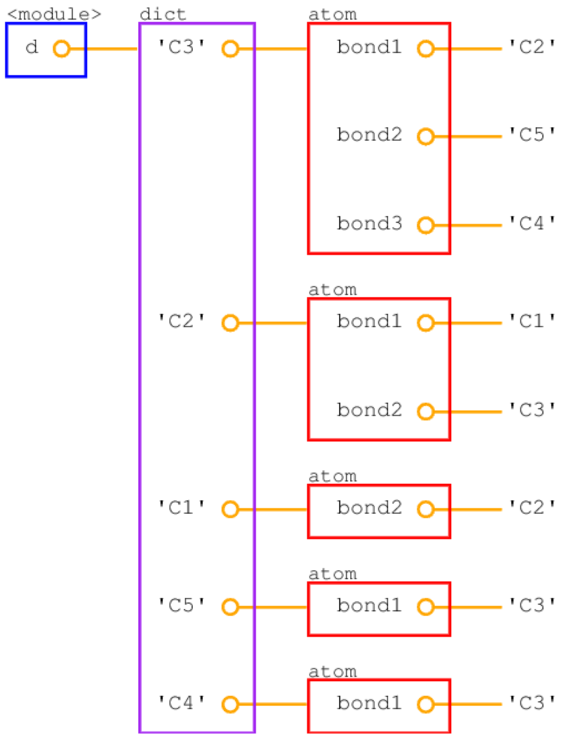


1. Nested List
Each chain of the molecule will be represented by a list of atoms, with each subchain- any chain that branches off the main one- as a sublist within this list. Subchains bonded to the same atom will be represented as lists within a larger list directly following the main-chain atom to which it is bonded (ergo, a subchain bonded to C3 would be included in the list as [C1,C2,C3,[C4,C5],C6], while if C3 had 2 subchains they would be [C1,C2, C3[[C4, C5],[C7]],C6] etc. This preserves the structure of the molecule, since each branch is located at its branch point in the data structure. This will allow us to display chains easily; however, it may be slower to return to a branch point than a dictionary.



2: Linked atoms:
While parsing the formula we will create a dictionary of atom objects, with the keys being the names of each atom (as stored by the object.) Each atom object will initially store only the name of the atom before it, but will be updated as parsing continues to contain the names of all the atoms bonded to it. This allows us to search the structure quickly, which will make representing branching structures

easier as we can quickly return to the branch point. It also depends directly on the data stored in each atom object, which should ensure that atoms are bonded to each other correctly.



Development Plan:
So far, we have decided on a module to use and come up with a couple of different options for data structures. In order to better visualize each option, we wrote some simple pseudocode and drew some state diagrams for each data structure option. Our next step will be to discuss the pros and cons of each data structure and choose accordingly. After this, we can implement some basic code that will work for simple chemical structures before moving onto more complex chains. Our team decided to have an emphasis on teamwork, so the plan is to have more frequent team meetings and less individual work. At the end of each meeting we will likely assign small tasks to each member, but the goal is to have everyone understand the entire scope of the project. We can do this by having each member present her findings, difficulties encountered, etc in her individual work at the beginning of each meeting before moving onto the next topic.

Design Refinement
Significant changes:

One significant change in our code is the implementation of subclasses. We initially wrote a version of our project that had a general Atom class, with no subclasses for each of the common elements. (This version of the code is in our Github repository, inside the folder labeled “version1”.) However, we found that this code contained a lot of conditional statements based on the element we were working with (for example, assigning size, color, and electron pairs to Atom objects all required a conditional statement depending on the element of the Atom object.) When we realized this, we decided to work on implementing subclasses for each of the common elements: hydrogen, carbon, nitrogen, oxygen, phosphorus, and sulfur. (This version of our project can be viewed in our Github repository, inside the folder labeled “version2”.) These subclasses assign information about the color and size of the Atom object, as well as its number of electron pairs.

Additionally, we finalized our data structure. In our design proposal, we mentioned that we were attempting to decide between different data structures -- we were considering trees, graphs, and other data structures. In the end, we decided not to implement trees or graphs, and instead decided to use a system of linked Atom objects (in which each Atom object contains a list of references to other Atom objects to which it is directly bonded). We felt this would be just as effective as the other data structures we were considering, and would be easier to implement. We believe this data structure will make it relatively easy to iterate through a list of Atom objects and build a molecule.

Significant progress:

We have successfully managed to translate user input (in the form of a string representation of a standardized structural formula) into an arbitrarily nested list of Atom objects. To do so, we stripped the input string of hydrogen atoms (to identify the base chain of Atoms). An example is shown below:

Input string: ‘CH3CH2CH(CH2NH2)CH2CH3’
Hydrogen removal: ‘CCC(CN)CC’

We then converted the string into a nested list of strings. An example for the same input string is shown below:

Input string: ‘CCC(CN)CC’
Nested list: [‘C’, ‘C’, ‘C’, [‘C’, ‘N’], ‘C’, ‘C’]

Then, we used that nested list to create a similarly nested list of Atom objects. An example for the same input string is shown below:

Nested list: [‘C’, ‘C’, ‘C’, [‘C’, ‘N’], ‘C’, ‘C’]
List of Atom objects: [C, C, C, [C, N], C, C]

Now that we have successfully translated user input into Atom objects, we can begin processing the Atom objects to determine which atoms are bonded to each other, and display the results in Blender.

We have successfully implemented subclasses of the Atom class for each of the relevant elements. Each subclass calls the parent class ‘Atom’: instances of a given element subclass contain variables representing the Atom’s name, display color, size, position, and a list of bonds. During the implementation process, we were receiving syntax errors when initializing subclasses without bonds assigned. This was because when Atom was called within the subclass, it was passed a None type as an argument. To solve this, we implemented a try catch statement that calls Atom without passing a bond assignment when a syntax error arises. Within the atom class, if it is called without bonds then it is automatically assigned to None. Later in the element subclass, if the atom bonds attribute is None, it is assigned either an empty list or a pair of electrons depending on the type of element.

We also began prototyping a function that creates our new data structure (a list of Atom objects that reference the Atom objects to which they are directly bonded.) In this function, we determine which Atom objects are directly bonded, and add those bonds to each of the Atom objects' bond lists. The function we have currently implemented can handle molecular chains of arbitrary length. However, we have yet to properly implement a function that can handle nested chains. Since this function is meant to handle an arbitrarily nested list of bonded Atom objects, we tried to write the function recursively. However, it currently appears that the recursive call is not being made, and so sub-chains are not being evaluated. Though this functionality is not required for our minimum deliverable, implementing an improved version of this function will be the next step in our project; we would like to implement nested chains in our final project.

We are currently attempting to define molecule placement based on the angles between atoms, since we know the angles at which atoms in molecules are located. To do this, the positions of the current atom and the atom it is bonded to are translated to a vector with an angle. This is then used to determine the position of other atoms bonded to the current atom.

After implementing subclasses, we were able to write a make_atom function that takes an atom and then generates a three dimensional sphere in Blender. Each atom has different attributes such as color and size that define how it is represented in Blender. We are also working on having location be contained within each atom, so that Blender can then take the location and draw the atom at the correct distance. This program is located in our repo in the script blendersphere.py.blend.

Abstraction and Language:

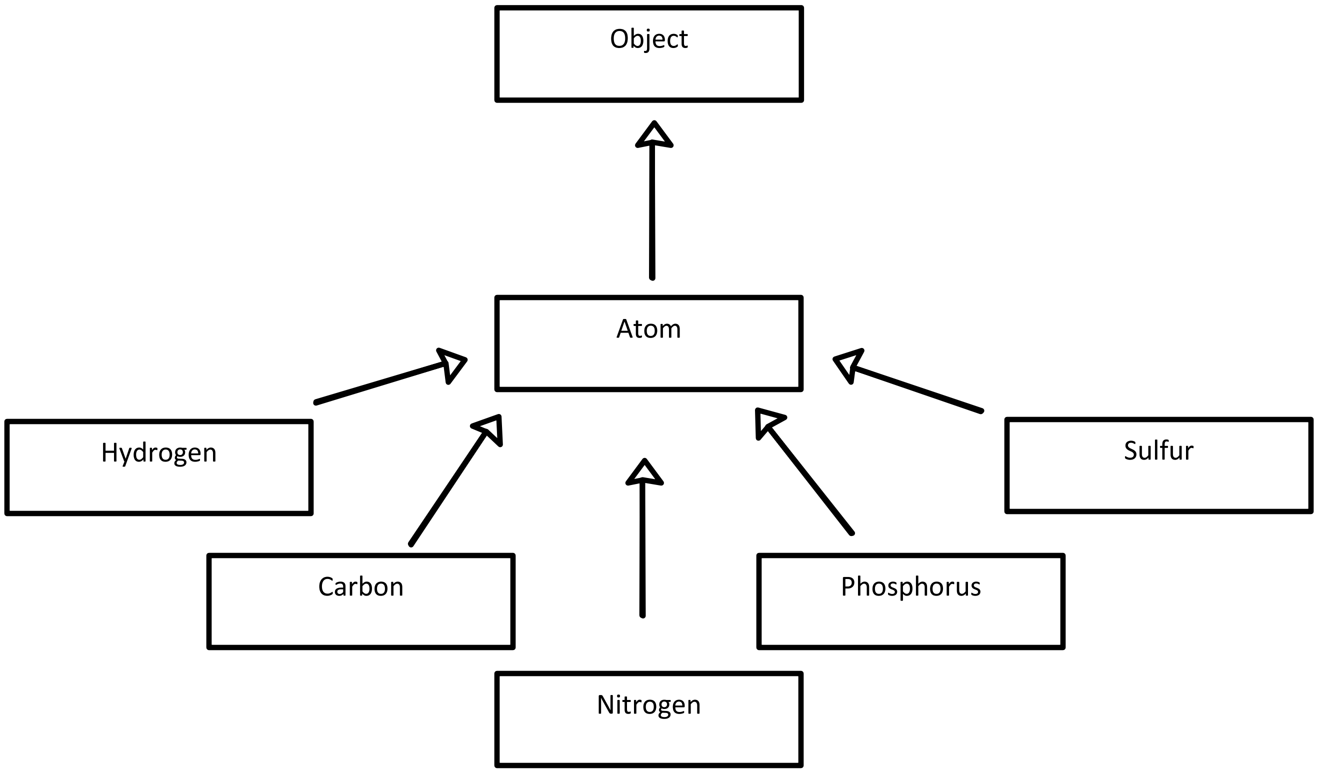
To write a molecular modeling tool, it made a lot of sense to use some very basic terms in the chemistry world. For example, we have a class of objects called Atom which contains an element attribute (Atom.elem) to contain information about what type of element a particular instance is (i.e. Carbon, Hydrogen, Nitrogen, etc.). Atoms also have a name attribute so that we can keep track of instances of Atom that have are the same element, this is vital because pretty much 100% of the time, we will have many carbons and hydrogens in the same molecule. In our newest version of the project, we also have atom subclasses, so we have classes called Hydrogen, Carbon, Nitrogen, etc. that all inherit from Atom, making it easier for us to instantiate new atoms.

Bonds are a critical part of any molecule, and we decided to make bonds an attribute of Atom and each instance of Atom contains a list of the atoms and electrons it is bonded to. This means that bonds are not separate objects and are really just defined by which atoms reference each other and the locations of those atoms.

The inherent vocabulary in the chemistry world made it easy to come up with this naming system. Talking about our code is very intuitive because we structured it such that each individual atom is a subclass of its element's name which it just a type of Atom. It also makes sense that atoms are objects but bonds aren't, considering bonds aren't actually physical things but rather the result of forces between atoms interacting with each other. So if you know even the most basic of all basic chemistry concepts, the way this code is structured and the naming conventions we gave everything should be fairly intuitive and easy to follow in conversation.

Diagrams:

In our class structure, the Atom class inherits from Object, and there are a number of specific element classes that inherit from Atom (Hydrogen, Carbon, Nitrogen, Oxygen, Phosphorus, and Sulfur). Atom objects also contain a list of references to other Atom objects. These relationships are included in the UML diagram of our class structure (*below)*:



Final Project Update
Major Changes:
Since the Design Refinement, we finished implementing our parsing functions, namely make_bonds(), which had been problematic for us. (make_bonds() is a function that takes a nested list of empty Atom objects and assigns bonds to them based on their positions in the list). The function was capable of handling lists representing single chains (i.e. non-nested lists) of arbitrary length, but it could not handle nested/branched chains at all. We fixed this error, and now the function will work for nested chains as well. Currently, all our parsing functions can handle nested chains, meaning that we could work on displaying nested chains in the future.

Additionally, we finished implementing functions to display atoms and chain them together into molecules. In order to determine atom position, we took advantage of the fact that most molecules consist of chained tetrahedra with various rotations and translations applied. We created a set of vectors representing the vertices of a tetrahedron with vertex-to-center distance b, b being a constant representing bond length. We then used a rotation matrix to orient the vectors along the vector between a previously placed central atom and the atom to which it was bonded, essentially setting this atom to the fourth vertex of the tetrahedron.Finally, we translated the center of the vector structure to the location of this central atom. This method is capable of creating chains of tetrahedra with and without electron pairs, allowing us to implement a variety of atoms in simple chain structures.

The display algorithm as initially implemented created a chain of tetrahedra that looped inward on itself, a result of repeatedly applying the same rotation to the structure. In order to avoid this and better simulate an actual atom, we applied a rotation in the y-axis to every second atom, producing a chain. This rotation is similar to that which would occur in an actual molecule, where inter-atomic forces push atoms apart. Finally, we implemented bond length scaling relative to the radius of the bonded atoms, in order to make the model appear more realistic.

To draw linked tetrahedra of atoms, we used rotation matrices to rotate our axes for each atom we plotted. Each time we plot an atom, the reference frame for the next atom changes (see **Fig. 1** of the attached documents). The easiest way to place new atoms is to rotate the reference frame with respect to the old one, and we accomplished this with the rotation matrix shown below:

$$\begin{matrix} \cos(\mathrm{atan}(x/z)+\pi) & 0 & -\sin(\mathrm{atan}(x/z)+\pi) \\ 0 & 1 & 0 \\ \sin(\mathrm{atan}(x/z)+\pi) & 0 & \cos(\mathrm{atan}(x/z)+\pi) \end{matrix}$$

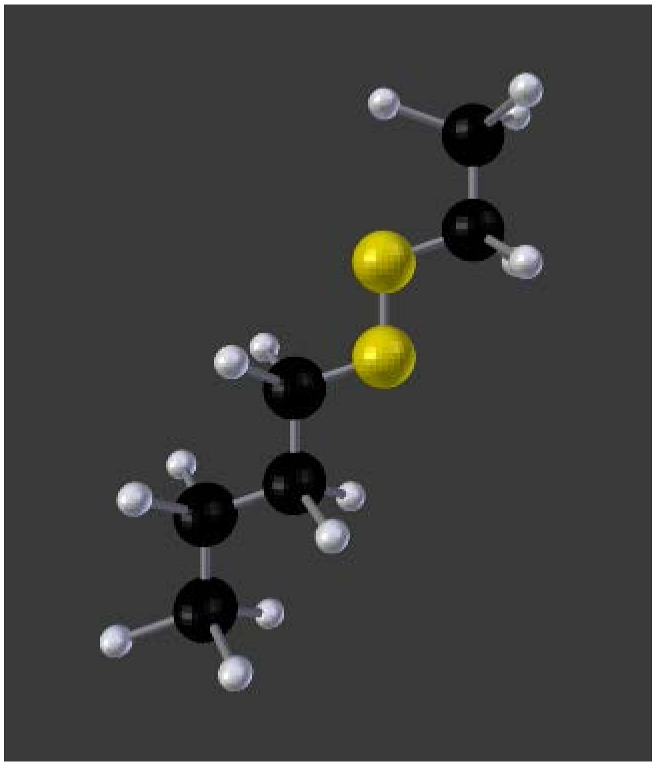
This rotation matrix represents a rotation of the base atom around the *y*-axis to create the new central atom of the next tetrahedron (see **Fig. 2** of the attached documents). After the reference frame has been shifted, the remaining two atoms are attached accordingly (**Fig. 3**). In this manner, the base atoms of tetrahedra can be attached to form an atom “chain” (**Fig. 4**).

Currently, this code works very well for single chains of arbitrary length. However, it does not yet work for “nested” (or “branching”) chains. This is because after rotation of the axes occurs, the remaining two atoms are attached in predetermined rotations with respect to the coordinate systems. If we want to implement branched chains, we will need to position these remaining two atoms via a 120-degree rotation about the z-axis (see **Fig. 5** of the attached documents) because we need to know how those atoms are defined relative to the previous coordinate system. This rotation can be represented by the following rotation matrix:

$$\begin{matrix} \cos(120) & -\sin(120) & 0 \\ \sin(120) & \cos(120) & 0 \\ 0 & 0 & 1 \end{matrix}$$

We tried to implement this rotation, but ran into difficulties with the angles we should use -- we are unsure of our calculations.

Analysis of the Outcome:



The output of our program in Blender when a $CH_3CH_2SSCH_2CH_2CH_2CH_3$ molecule is given as input. The black molecules represent Carbons, yellow represent Sulfur and gray represent Hydrogen.

We have achieved our minimum deliverable, which is to display a string of Carbon and Hydrogen atoms. We also exceeded our expectations by configuring our code to display other atoms such as Sulfur, Hydrogen and Phosphorus. The main difference between these atoms and the Hydrogens and Carbons is that they are different sizes, and sometimes have electron pairs instead of bonds. We are still having trouble implementing our maximum deliverable, which involves molecular sub-chains that break the coplanar string of central atoms. This is a problem mostly in the mathematics side of our project as opposed to the coding. We hope to clear up the remaining issues with our rotational matrices with one of the Linearity professors and have a fully functional representation of sub-chains before our presentation at Expo. The project ended up being about the right level of difficulty, with the exception of the mathematics being a lot harder than anticipated.

Bug Report:

As we mentioned before, our rotational matrices for tetrahedra do not work for sub-chains. We've experimented with many different matrices and angle calculation. Often times the code will work for one or two cases, but then break on a third case and display deformed tetrahedra. This has something to do with how we are rotating the axes, and the angles at which they are rotating.

Reflection on the Design:

Looking back, it would have been nice to have spent more time discussing the display functions prior to implementation. We did not realize how difficult displaying atoms would be, and so we did not spend much time thinking about those functions at the beginning of the process. We found that displaying tetrahedra is more complex than we thought, and involves some relatively interesting math. We would have liked to have spent more time thinking about it, as the logistics of the math can be difficult to visualize spatially. Additionally, this problem was made more difficult by the fact that Blender lacks support for spherical coordinate systems.

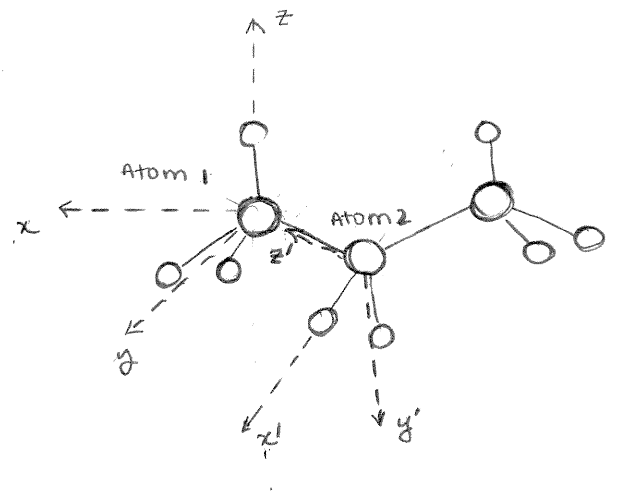
We also found that we really liked our class design and structure. Creating an Atom class made a lot of sense within the context of the structure of our project. Our project was very conducive to object-oriented programming because atoms actually are objects with their own properties. Creating an Atom class resulted in code that is intuitive and straight-forward from a physical perspective. We did consider making "Bond" objects representing individual bonds between atoms, but decided against it (as bonds are more of a property of atoms, rather than a concrete object in and of themselves). We believe this was a good decision. Additionally, the use of an Atom class easily lent itself to subclasses representing different elements.

Division of labor:

During this project, we planned and wrote most of our code during group meetings. Rather than dividing labor, we coded as a group. We definitely could have improved upon our division of labor. Part of the problem was that we found it difficult to divide our project into distinct parts (aside from parsing and display) because our code is very linear. We should have planned out the structure of our code better. If we had planned out the interfaces between our functions better, we could have assigned different functions to different people. However, we did not spend much time on determining the flow of our code, so much of the coding for this project was done during group meetings. Planning and designing interfaces between functions would definitely have helped in parallelizing the workload.

Drawings and Figures:

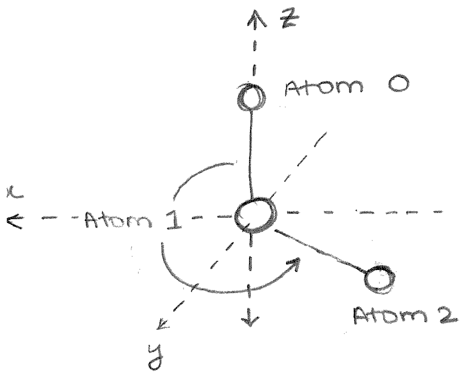
Fig. 1: Changing reference frame



Atom 1 is the first central atom with axes x, y, and z.

Atom 2 is the second central atom, branching off from Atom 1. It has axes x', y', and z'. These axes are translated and rotated versions of x, y, and z.

Fig. 2: Rotation around the y-axis.



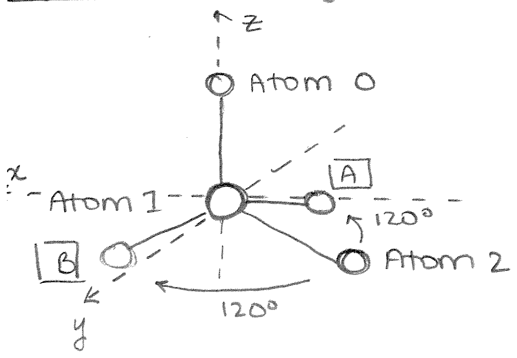
Atom 1 is the central atom, analogous to Atom 1 in Fig. 1.

Atom 0 is an atom attached to Atom 1, and is rotated around the y-axis to obtain the position for Atom 2.

Atom 2 is the next central atom.

* y is coming out of the page in this diagram.

Fig. 3: Attaching the remaining atoms.

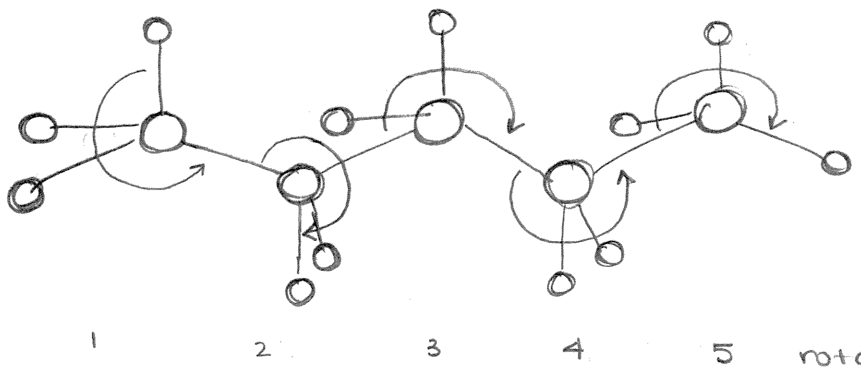


After Atoms 0, 1, and 2 have been placed, Atoms A and B are positioned 120° away from Atom 2 in both directions.

Note that this could be performed by a 120° rotation about the z-axis in both directions. See Fig. 5.

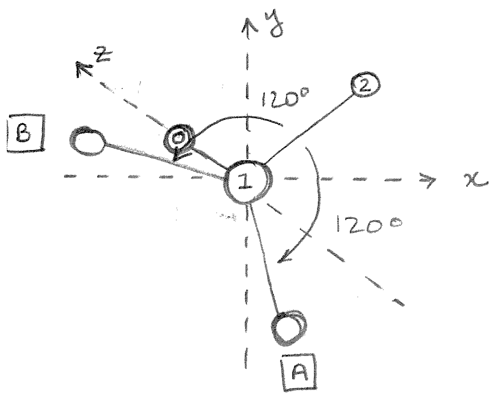
Drawings and [illegible]

Fig. 4: Forming a "chain" out of many base tetrahedra.



rotations about the y-axis.

Fig. 5: Attaching the remaining atoms — now with rotation.



After Atoms 0, 1, and 2 are in place, Atom 2's position is rotated 120° around the z-axis to yield the positions of atoms A and B.

(Atoms 0, 1, 2, A, and B are in analogous positions as in Fig. 3. Atom 0 is along the z-axis, pointed away from the viewer.)

* z is going into the page, in this case.