Assignment 2 –Molecule Webserver Application

Version 1.01 (last update: Feb. 2, 7:02)
Changes highlighted in yellow
Due date: Tue, Feb 28, 9:00 AM

Summary and Purpose

For this assignment, you will be writing a webserver in python that uses your C library from assignment 1 to render a simple view of a molecule based on an uploaded SDF file.

Deliverables

You will be submitting:

1) A file called **`mol.h`** that contains your typedefs, structure definitions and function prototypes (see below).
2) A file called **`mol.c`** that contains your c function code.
3) A python library call MolDisplay.py that generates SVG images of molecules.
4) A python program called server.py that acts as a webserver that allows uploading of SDF files and serves SVG files.
5) A **`makefile`** that contains the instructions for the following targets:
    a. **`mol.o`** – a position independent (**`-fpic`**) object code file created from **`mol.c`**.
    b. **`libmol.so`** – a shared library (-shared) created from **`mol.o`**.
    c. **`molecule_wrap.c`** and **`molecule.py`** a pair of files that provide a Python interface to your C code (these are generated using the **`swig`** program base on the instructor supplied **`molecule.i`** file.
    d. **`molecule_wrap.o`** – an object file that is an object library to interface with your C code.
    e. **`_molecule.so`** – a shared object library used by molecule.py to interface between C and Python.
    f. **`clean`** – this target should delete all **`.o`**, **`.so`** and executable files.

    All compilation must be done with the **`-std=c99 -Wall -pedantic`** options and produce no warnings or errors. The **`makefile`** must correctly identify all relevant dependencies and include them as prerequisites in its rules.

You will submit all of your work via git to the School's **`gitlab`** server. (As per instructions in the labs.) Under no condition will an assignment be accepted by any other means.

All files should be managed in a single directory (don't organize your files into subdirectories based on types; sorry).

This is an individual assignment. Any evidence of code sharing will be investigated and, if appropriate, adjudicated using the University's Academic Integrity rules.

Setting up your work environment and using git:

Decide on your development environment. You can work on your own machine, the SoCS VM, no machine, or the SoCS ssh servers. However, you must test your code on the SoCS servers or SoCS VM as this is where it will be evaluated. If your code does not work properly on the SoCS systems, then it does not work.

```
git clone https://gitlab.socs.uoguelph.ca/2750W23/skremer/A2
```

But, use your own login ID instead of `skremer`.

Then, `cd A2`, to move to the assignment directory.

Work in the A2 directory. Use the command:

```
git add filename
```

For each file that you create as part of your code (see deliverables).

Every so often (especially if you get something working), use the command:

```
git commit -a
```

to commit your changes.

Use the command:

```
git tag -a Feb10a -m 'implemented the first two functions'
```

You will need to use a tag like "`Feb10a`" to request help with your code via the course help page.

And then make sure to use:

```
git push --all --follow-tags
```

to push everything to the server with the tag included.

If you want to make sure you know exactly what's on the server you can rename the A2 directory to A2.day1 and then use

```
git clone https://gitlab.socs.uoguelph.ca/2750W23/skremer/A2
```

to get a copy of exactly what is currently in the repo.

You can use the same command to retrieve your code on a different machine or architecture. E.g. to move your code from your laptop development to the SoCS server for testing.

PART I: Expanding and improving A1

Be sure to fix any errors in your A1 that were detected during grading as you will be penalized a second time for things that continue not to work correctly.

New data structures

Replace the bond structure with the following:

```
typedef struct bond
{
  unsigned short a1, a2;
  unsigned char epairs;
  atom *atoms;
  double x1, x2, y1, y2, z, len, dx, dy;
} bond;
```

bond defines a structure that represents a co-valent bond between two atoms. **a1** and **a2** are indices of the two atoms in the co-valent bond within an array with address **atoms**. **epairs** is the number of electron pairs in the bond (i.e. **epairs**=2 represents a double bond). **x1**, **y1**, **x2**, **y2** will store the **x** and **y** coordinates of atoms **a1** and **a2** respectively. **z** will store the average **z** value of **a1** and **a2**. **len** will store the distance from **a1** to **a2**. **dx** and **dy** will store the differences between the **x** and **y** values of **a2** and **a1** divided by the length of the bond.

Function prototypes and descriptions to change for this assignment

```
void bondset( bond *bond, unsigned short *a1, unsigned short *a2, atom
**atoms, unsigned char *epairs );
```

This function should copy the values pointed to by **a1**, **a2**, **atoms**, and **epairs** into the corresponding structure attributes in **bond**. In addition, you should call the **compute_coords** function (see below) on the bond.

```
void bondget( bond *bond, unsigned short *a1, unsigned short *a2, atom
**atoms, unsigned char *epairs );
```

This function should copy the structure attributes in **bond** to their corresponding arguments: **a1**, **a2**, **atoms**, and **epairs**.

```
void compute_coords( bond *bond );
```

This function should compute the **z**, **x1**, **y1**, **x2**, **y2**, **len**, **dx**, and **dy** values of the bond and set them in the appropriate structure member variables.

```
int bond_comp( const void *a, const void *b );
```

You will have written a bond comparison function in A1. This will need to be modified based on the new structure used for **bond**. (It will actually be much simpler now, because you can assume that each bond's **z** value has been correctly set.)

```
void mol_xform( molecule *molecule, xform_matrix matrix );
```

This function should be modified to apply the **compute_coords** function (above) to each bond in the molecule.

PART II: Creating a Python library from your C-code.

swig is a toolkit that is used to interface compiled languages with scripting languages. I uses an interface file with a **.i** extension. Your instructor has provided a file **molecule.i** that interfaces with your C-code. You can run swig with the command:

```
swig -python molecule.i
```

The program will produce two files as output. **molecule_wrap.c** and **molecule.py**. The molecule_wrap.c file should be compiled with the **-fPIC** option and the **-I** option with the path to the python include file to create a **molecule_wrap.o** file. You will need to find the correct include file path which depends on the version of python you are using. On UNIX the python header file can be found in: **/usr/include/python3.7m**; while on OS-X I found it in: **/Library/Frameworks/Python.framework/Versions/3.11/include/python3.11**.

The **molecule_wrap.o** file can then be compiled to create a **_molecule.so** file. You will need to the **-shared** option to create the **.so** file, **-l** option for the Python language library, another **-l** option for your **mol** library, the **-dynamiclib** option and two **-L** directives for the location of your **mol** library and the location of the Python language library. On UNIX the Python language library is called **python3.7m** and is found in the directory: **/usr/lib/python3.7/config-3.7m-x86_64-linux-gnu**. On OS-X I found them as: **python3.11** in **/Library/Frameworks/Python.framework/Versions/3.11/lib.**

Once you have everything built, you can try the following program:

```
import molecule;

mol = molecule.molecule();  # create a new molecule object

# create 3 atoms
```

```
mol.append_atom( "O", 2.5369, -0.1550, 0.0000 );
mol.append_atom( "H", 3.0739, 0.1550, 0.0000 );
mol.append_atom( "H", 2.0000, 0.1550, 0.0000 );

# caution atom references in append_bond start at 1 NOT 0
mol.append_bond( 1, 2, 1 );
mol.append_bond( 1, 3, 1 );

for i in range(3):
    atom = mol.get_atom(i);
    print( atom.element, atom.x, atom.y, atom.z );

for i in range(2):
    bond = mol.get_bond(i);
    print( bond.a1, bond.a2, bond.epairs, bond.x1, bond.y1, bond.x2, bond.y2,
bond.len, bond.dx, bond.dy );
```

The output of this program should look like this:

```
O 2.5369 -0.155 0.0
H 3.0739 0.155 0.0
H 2.0 0.155 0.0
0 1 1 2.5369 -0.155 3.0739 0.155 0.6200556426644305 0.8660513074156804 0.4999551309103556
0 2 1 2.5369 -0.155 2.0 0.155 0.6199690395495571 -0.8660109872423447 0.5000249693520707
```

PART III: Creating a Python library to read SDF files, represent the molecules, and write SVG files.

Create a Python file called MolDisplay.py. Import the “molecule” Python module to access all the features from your C library.

Define the following constants:

```
radius = { 'H': 25,
           'C': 40,
           'O': 40,
           'N': 40,
         };

element_name = { 'H': 'grey',
                 'C': 'black',
                 'O': 'red',
                 'N': 'blue',
               };

header = """<svg version="1.1" width="1000" height="1000"
                  xmlns="http://www.w3.org/2000/svg">""";

footer = """</svg>""";

offsetx = 500;
offsety = 500;
```

Create an **Atom** class. This class should be a wrapper class for your **atom** class/struct in your C code. Objects of the **Atom** class should be initialized by calling an **Atom(c_atom)** constructor method with an **atom** class/struct as its argument. The constructor should store the **atom** class/struct as a member variable. It should also initialize a member variable, **z**, to be the value in the wrapped class/struct.

Add an **__str__** method to the **Atom** class. This method takes no arguments. You will need this for debugging. Make this return a string that displays the **element**, **x**, **y**, and **z** values of the wrapped **atom**.

Add an **svg** method to the **Atom** class. This method takes no arguments. This class must return the following string:

```
'  <circle cx="%.2f" cy="%.2f" r="%d" fill="%s"/>\n'
```

Exactly as shown with two leading spaces and all spaces and punctuation exactly as presented, but use the "**%**" string substitution operator to fill in 4 values:

The x coordinate of the centre of the circle representing the atom. This should be computed by multiplying the **x** coordinate of the **atom** by **100.0**, and adding the **offsetx** constant, described above.

The y coordinate of the centre of the circle representing the atom. This should be computed by multiplying the **y** coordinate of the **atom** by **100.0**, and adding the **offsety** constant, described above.

The radius of the circle representing the atom. This should be computed by looking up the **atom**'s **element** name in the **radius** dictionary described above.

The colour of the circle representing the atom. This should be computed by looking up the **atom**'s **element** name in the **element_name** dictionary described above.

Create a **Bond** class. This class should be a wrapper class for your **bond** class/struct in your C code. Objects of the **Bond** class should be initialized by calling a **Bond(c_bond)** constructor function with an **bond** class/struct as its argument. The constructor should store the **bond** class/struct as a member variable. It should also initialize a member variable, **z**, to be the value in the wrapped class/struct.

Add an **__str__** method to the **Bond** class. This method takes no arguments. You will need this for debugging. Make this return a string that displays the relevant information of the wrapped bond.

Add an **svg** method to the **Bond** class. This method takes no arguments. This class must return the following string:

```
'  <polygon points="%.2f,%.2f %.2f,%.2f %.2f,%.2f %.2f,%.2f" fill="green"/>\n'
```

Exactly as shown with two leading spaces and all spaces and punctuation exactly as presented, but use the "`%`" string substitution operator to fill in 8 values. We will use this to draw a 20-pixel-wide line from the centre of one atom, to the centre of another atom. The 8 values will represent the **`x`**, and **`y`** coordinates or the 4 corners of a rectangle that looks like a thick line between the atoms at either side of the bond. The end points of the thick line can be calculated just like the positions of the centres of the circles used to depict the atoms. Note that you can use the **`x1`**, **`y1`**, **`x2`** and **`y2`** values in the bond structure to access these. Those values are at the centre of the short edges of the rectangle. To determine the corners or the rectangle, we need to move perpendicularly to the direction of the bond 10 pixels from the centre.

Example:

> Suppose we have two atoms located at: `0.5, 0.0, 0.0` and `-0.5, 0.0, 0.0`. They are horizontally side-by-side with a distance of 1 between them. The centres of the circles will be located at pixel coordinates of `450.0, 500.0` and `550.0, 500.0`. In this example, **`bond.dx=1.0`** and **`bond.dy=0.0`** because the bonds are located horizontally to each other. If we want to move perpendicularly to the bond from the position `450.0,500.0` we should move to `450.0,500.0-bond.dx*10.0` and `450.0,500.0+bond.dx*10.0` (which will be 10 pixels above and below the atom centres). Notice how the **`dx`** value gets multiplied by the **`y`** value. At the other end of the bond we should move to `550.0,500.0+bond.dx*10.0` and `550.0,500.0-bond.dx*10.0`. Make sure you give the coordinates in a consistent direction around corners of the rectangle (not in an hour-glass patten), or the rectangle won't show up.

Create a Molecule class that is a subclass of the molecule class.

Add a **`__str__`** method that prints out the bonds and the atoms in the molecule for debugging purposes. This method takes no arguments.

Add a **`svg`** method. This method takes no arguments. It return the header string above, followed by the return values of the **`svg`** methods of the **`Atom`** and **`Bond`** classes, and then the footer string. These strings should be appended to each other ***exactly*** as produced with no extra spaces, punctuation, etc. Note that the **`svg`** string for the **`atom`**s and **`bond`**s must be appended in order of increasing **`z`** value. You may assume that the **`mol_sort`** function has been applied, so that **`atom`**s and **`bond`**s are sorted individually, but you will need to interleave the two arrays to produce the correct ordering of **`svg`** entries. You can do this by applying the final pass of a merge sort function. The algorithm might be something like this:

```
a1 <- pop( atoms )
b1 <- pop( bonds )
if a1.z < b1.z:
      append( a1.svg() )
      a1 <- pop( atoms )
else: # b1.z < a1.z
      append( b1.svg() )
      b1 <- pop( bonds )
```

You may assume there are no identical **z** values.

Add a **parse** method. The parse method should take a file object (technically a TextIOWrapper object) like that returned by the "**open**" function as its single argument. You may assume that all input files will be valid "**.sdf**" files, like the three examples provided. You should call the **append_atom** method and **append_bond** method of the molecule as you read in all the atoms and bonds from the file to create a molecule object representing the molecule in the "**.sdf**" file.

Debug and test your code with the provided **.sdf** files. You can save the **svg** function output to a file and open it with your browser to see the circles and rectangles.

PART IV: Creating a webserver to upload SDF files and return SVG files.

Create a file called **server.py**. Import the **HTTPServer** and **BaseHTTPRequestHandler** classes from the **http.server** Python module.

Create your own subclass of the **BaseHTTPRequestHandler** that provides implementation of the **do_GET** and **do_POST** methods.

Use a single command line argument, to specify the listening port or your webserver.

IMPORTANT: To avoid using the same port as your classmates use port 5####, where #### are the last 4 digits of your student ID. Failure to follow this instruction, or attempts to connect to your classmates webservers maybe considered violations of the ***Schools AUP*** or ***Academic Misconduct***.

Implement your **do_GET** method so that it presents a web-form when the path, "**/**" is requested and generates a **404** error otherwise. The web-form should look something like this:

```
<html>
  <head>
    <title> File Upload </title>
  </head>
  <body>
    <h1> File Upload </h1>
```

```
    <form action="molecule" enctype="multipart/form-data" method="post">
       <p>
         <input type="file" id="sdf_file" name="filename"/>
       </p>
       <p>
         <input type="submit" value="Upload"/>
       </p>
    </form>
  </body>
</html>
```

Implement a do_POST method that sends an svg file to the client when the path "/molecule" is requested and generates a **`404`** error otherwise. HINTS:

(1) use **`self.rfile`** to access the uploaded file,
(2) use the **`MolDisplay.Molecule`** class that you created in Part III.
(3) Skip 4 lines of header information in **`self.rfile.`**
(4) Use "**`Content-type: image/svg+xml`**" to inform the browser of the data being sent.
(5) Call the **`sort()`** method on the molecule between the **`parse()`** and **`svg()`** methods.

You can write additional helper functions as necessary to make sure your code is modular, readable, and easy to modify.

Testing

You are responsible for testing your code to make sure that it works as required. The CourseLink web-site will be updated with some test programs to get you started. However, we will use a different set of test programs to grade your code, so you need to make sure that your code performs according to the instructions above by writing more test code.

Your assignment will be tested on the SoCS servers. If you are developing in a different environment, you will need to allow yourself enough time to test and debug your code on the target machine. If your code works on one machine/environment but not another, you code is incorrect. Correct code will work consistently across all machines/environments. We will NOT test your code on YOUR machine/environment.

Ask Questions

The instructions above are intended to be as complete and clear as possible. However, it is YOUR responsibility to resolve any ambiguities or confusion about the instructions by asking questions in class, via the discussion forums, or by e-mailing the course e-mail.

Nightmare mode

Add 3 input text fields to the web-form. Assign them the same "id" and "name". Use the following values "roll", "pitch", "yaw". Apply a rotation to the molecule in degrees equal

around the x, y, and z axes for “pitch”, “yaw” and “roll”, respectively, before displaying the molecule. Add a **`rotate`** method to the **`Molecule`** object in **`molecule.i`** that takes 3 unsigned short integers as arguments (roll, pitch, yaw), creates the appropriate rotation matrix and applies it to the molecule.

Git

You must submit your **`.c`**, **`.h`**, **`.py`** and **`makefile`** using **`git`** to the School’s git server. Only code submitted to the server will be graded. Do ***not*** e-mail your assignment to the instructor. We will only grade one submission; we will only grade the last submission that you make to the server and apply any late penalty based on the last submission. So once your code is complete and you have tested it and you are ready to have it graded make sure to commit and push all of your changes to the server, and then do not make any more changes to the A1 files on the server.

Academic Integrity

Throughout the entire time that you are working on this assignment. You must not look at another student’s code, nor allow your code to be accessible to any other student. You can share additional test cases (beyond those supplied by the instructor) or discuss what the correct outputs of the test programs should be, but do not share ANY code with your classmates.

Also, do your own work, do not hire someone to do the work for you.

Grading Rubric

bondset	1
bondget	1
compute_coords	3
bond_comp	1
mol_xform	1
Atom.__init__	1
Atom.svg	2
Bond.__init__	1
Bond.svg	4
Molecule.svg	7
Molecule.parse	4
do_GET	2
do_SET	4
style	4
makefile	4
Total	40