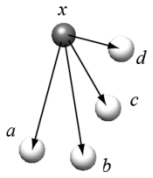


CS483 Assignment # 5

Due dates: Thurs. Apr. 2, or Tues. Mar. 31 for 5 bonus marks.

Marks: **Ex. 1:** [7], **Ex. 2**: [8], **Ex. 3**: [7], **Ex. 4**: [8], **Ex. 5**: [10]. **Ex. 6**: [7], **Ex. 7**: [8], **Ex. 8**: [10], **Ex. 9**: [10].
The assignment will be marked out of 75.

Part 1: Using distance geometry to build molecular models

Preamble

Run the following script to build a model of cyanide:

```
import chimera, numpy
from chimera import runCommand, Element, Point
from BuildStructure import _newModel, _newResidue
from chimera.molEdit import addAtom, addBond

runCommand("close session; set bg_color white")
cnMol = _newModel("Cyanide")
res1 = _newResidue(cnMol, "Res1")
carb = addAtom('C1', Element('C'), res1, Point(0.0, 0.0, 0.0))
nitr = addAtom('N1', Element('N'), res1, Point(0.0, 1.5, 0.0))
addBond(carb,nitr)

chimera.viewer.viewAll()
```

Your display should show a simple two atom molecule. Note that it can be found in the Model Panel and you can select the "residue" by using the **Select Residue** menu invocation. If you change the atom representation to sphere you should get a visualization similar to that seen at:
http://en.wikipedia.org/wiki/Cyanide

You should examine the script to understand how the molecule was built. Even though it is a simple example, it should be clear to you that a more extensive script could be used to generate any arbitrary structure if you knew the bonding patterns of the atoms and their locations.

Biochemists who study protein folding processes often start with simple helix structures typically involving alanine and some other amino acid[1]. In this assignment we will be building helices that are arbitrary sequences of alanine and serine amino acids. We will go through a sequence of short exercises that develop the routines that you will need for the final exercise in Part 1 of the assignment.

Exercise 1: Building a distance matrix

Distance geometry algorithms require a collection of typical distances that will be used to define bond lengths. We will use the protein with PDB ID = 2X5O. It is a high resolution analysis and the protein has all possible dipeptide examples: AA, AS, SA, and SS.
Start by writing a function that will generate a distance squared matrix for a residue. Here is the header:

[1] S. Marqusee, V.H. Robbins, and R.L. Baldwin. Unusually stable helix formation in short alanine-based peptides. *Proc. Natl. Acad. Sci. USA.* **86** (1989) pp. 5286-5290.

```
def dist2_A(r, aNames_L)
```

Inputs: `r` is a residue object and `aNames_L` is a list of strings representing an ordered list of the names of the atoms in the residue.
Output: a shape (`n`, `n`) array holding the squares of the distances between the atoms in the residue. Note: `n = len(aNames_L)`.

It would be possible, within the function script, to get the list of atom names from the residue object, but we want the array to have a row and column ordering that is determined by an order that we specify. Test the function by giving it the ALA residue at position 84 and the SER residue at position 82 (both from 2X5O). Print out the atom name list and distance squared matrix for each residue.

The data should be made available for scripts in the exercises to follow. This can be done by writing out a text file, but this requires formatting for the write operations and then using carefully constructed code when the data is read by the other script. An easier approach is to serialize the data. This can be done by using the Python `pickle` class. To write out the data use statements such as:

```
import pickle
pickleDat = (aNames_D, distSq_D)
output = open('DistSq Data', 'wb')
pickle.dump(pickleDat, output)
output.close()
```

Later (as in Exercise 2), in the script requiring the data, you can simply reload the data using:

```
import pickle
pkl_file = open('DistSq Data', 'rb')
(aNames_D, distSq_D) = pickle.load(pkl_file)
pkl_file.close()
```

After this, the contents of the tuple `(aNames_D, distSq_D)` can be used as if they were defined in this script.

Exercise 2: Using distance geometry to get atom coordinates

Write a function that will return a dictionary mapping atom names to atom coordinates when given an atom name list and the corresponding distance squared matrix. Here is the header:

```
def buildAtomCoordsDict(dSQ_A, aNames_L)
```

Inputs: `dSQ_A` is an *n* by *n* array holding the distance squared matrix, `aNames_L` is a list of strings representing an ordered list of the names specifying the significance of the rows and columns in the `dSQ_A` matrix, and *n* has the same meaning as in the previous exercise.
Output: a dictionary object with the atom names as keys and corresponding values being `Point` objects holding the atom coordinates.

Write a small mainline program to test your function. To get the distance matrix and atom name list use the `pickle` statements discussed in Exercise 1. Print the dictionaries for both ALA and SER.

Exercise 3: Building a residue model

Write a function that will build a model of a residue when given the dictionary produced by Exercise 2 along with a bond list that specifies the atom pairs (each specified by name) to be bonded. You will likely use the script implemented in Exercise 2 modified to include the function and any other needed statements. Here is the header for the function:

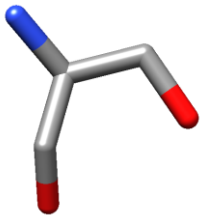
```
def buildRes(coordDict, bonds_L, newModel, resType)
```

Inputs: `coordDict` is a dictionary mapping atom names to atom coordinates held in `Point` representation, `bonds_L` is a list of pairs of atom names, and *n* has the same meaning as in the previous exercise. `newModel` is a model object defined in the mainline and `resType` is a character string specifying the type of the residue.

Output: the residue object with atoms and bonds specified by the input parameters. The residue will become part of the `newModel` and, as such, will be seen in the display.

Write a small mainline program to test your function. You will have to generate your own bond list. The function will be using the output of `buildAtomCoordsDict` to get the required dictionary.

The display for SER should look like the following:



Exercise 4: Generating data for geometric buildup

Write a function that generates a dictionary that stores distances between atoms that are in adjacent residues. Function header:

```
def atmAtmDist2(r1, r2)
```

Inputs: `r1, r2` are residue objects representing two consecutive residues in the protein being processed.

Output: a dictionary holding the squares of the distances between the atoms in the residues. Dictionary key: atom names separated by a colon, for example, "CA:O". The first name is for an atom in `r1` and the second name is for an atom in `r2`. The value corresponding to the key will be the square of the distance between the two named atoms.

Relevant Notes:

- The eventual use of the dictionary will be to calculate coordinates of atoms in a residue using the known coordinates of atoms in the previous residue.
- The first atom name of the dictionary key should be in the tuple: `("CB", "O", "N", "C", "CA")`. In other words, you should avoid distances to the `r1 OG` atom when the `r1` residue is serine. We want the *known* coordinates in the geometric buildup algorithm to be consistent with a well formed helix. The position of `OG` in serine depends on the rotameric setting. Using it in `r1` will simply introduce errors in the geometric buildup algorithm (if you have some time to

experiment you might try to use it in `r1` to see what happens). Of course, we will want `OG` to be in the dictionary keys (but only after the ":").

To get the dictionary values, you should fetch the protein with PDB ID = 2X5O (as in Exercise 1). It is a high resolution analysis and the protein has all possible dipeptide examples: AA, AS, SA, and SS. The following table specifies the chain position of the first residue in each pair:

`r1,r2`	`r1` position
`AA`	83
`AS`	287
`SA`	82
`SS`	288

As in Exercise 1, your mainline script should pickle the results so that they can be used in Exercise 5. Since there are four dictionaries you should consider the possibility of combining them into a single data structure, for example, a dictionary of dictionaries and then this top level dictionary gets pickled.

Exercise 5: Building a helix model

Write a function that uses the results provided in the previous exercise to extend the model by adding on a new residue. Function definition line:

```
def extendStructure(prevCoords_D, atomAtomDistSq_D)
```

Inputs: `prevCoords_D` is a dictionary mapping atom names to atom coordinates held in `Point` form, `atomAtomDistSq_D` is a dictionary produced by the function scripted in Exercise 4.

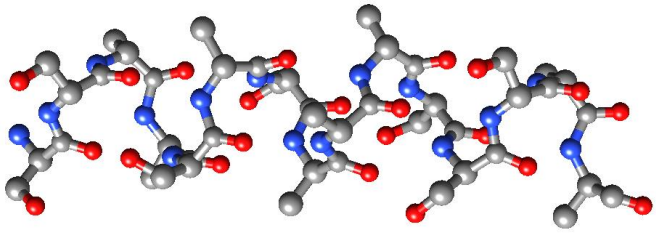
Output: a dictionary object with the atom names as keys and corresponding values being `Point` objects holding the atom coordinates for the new residue. The function should use the geometric buildup algorithm to derive the coordinates of all atoms in the new residue based on known coordinates of atoms in the previous residue.

Implementation issues:

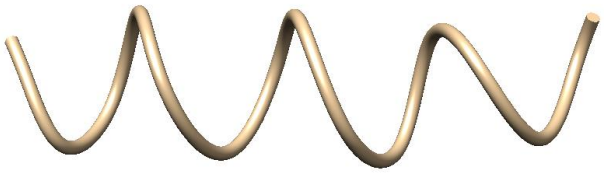
Note that there are going to be, not four, but five atoms in `r1` with known coordinates and you will get more accuracy if all five are used in the linear system that is to be solved in geometric buildup. To get a solution of the system you can use the `linalg.lstsq` function that is provided by `numpy`. There are several Internet pages describing the use of this function.

Testing the function:

The mainline program should use the `raw_input` function to get a string from the user that is an arbitrary sequence containing upper case `A`'s and `S`'s representing the positions of alanine and serine residues in a helix, for example: `SSAASASAAASSSAA`. The mainline script should use the functions developed for Exercises 2 and 3 to put the first residue into the display (in the example, this would be serine). This is immediately following by iterations of a loop that handles the remaining residues (in the example: serine, alanine, alanine, serine, ….., alanine) each pass through the loop executing invocations of the functions developed in Exercises 4, 5, and 3. This should produce the display of all residues. It will be necessary to have a final loop that puts in the bonds that connect the residues, specifically: the bonds between the final "`C`" of a residue and the initial "`N`" of the next residue.
If the user submits the string given in the above example, then the display should show:



You can put this model into ribbon form but since it was not created by a PDB file that would explicitly specify secondary structure, you will get a rounded ribbon or "noodle" representation:



Part 2: Separation surfaces between molecular models

Motivation

Use Chimera to fetch the protein file 1YCQ and use the **Tools Surface/Binding Analysis Intersurf** menu to put a surface between the two proteins in the protein complex. When the Compute Interface Surface dialog appears select both chains after clicking on the Chains tab. Then click on the OK button. You get a smooth surface that separates the two molecules. Surface constructions, such as this, are often done in studies of protein-protein interactions. To fully appreciate the significance of the surface you should hide the ribbons and display the atoms using different colours for each chain.

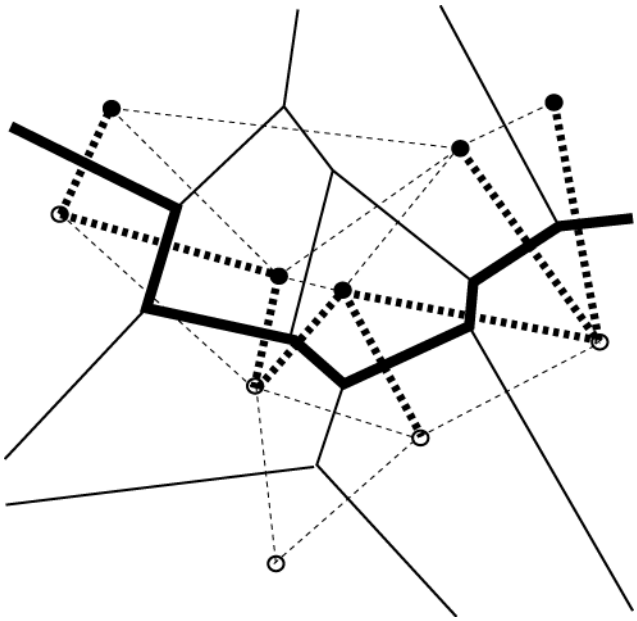
The object of this part of the assignment is to generate a similar surface separating a ligand from its binding site. The surface will be piecewise linear (a triangulated surface) and so it will not look as smooth as the surface provided by Intersurf. There are extra steps that could be taken to smoothen the piecewise linear surface but that is beyond the scope of the course.

Note that Intersurf works with two chains and a ligand is usually part of chain (not a chain by itself). Another benefit to be gained by doing this with your own code, relates to the control that you have for the coloring of the surface. You have the option of giving the triangles in the surface a colour that is dependent on some physical parameter of your choice (for example, the distance between atoms on either side of the surface).

So, how does one compute such as surface? Before any additional steps involving surface smoothing, we would construct a 3D Voronoi surface that is derived from a 3D Delaunay triangulation. Voronoi surfaces are found in many graphics applications including cartoon animations (do a Google search for images related to Voronoi surfaces). A similar search for Delaunay triangulation will help you to see the relationship between a Delaunay triangulation (also called a tessellation) and the Voronoi surface.

The next figure shows the situation in a 2D setting. The dashed lines represent the lines for the Delaunay triangulation. The important feature of the triangulation is that the dashed lines join points that can be considered to be near neighbours of each other (a very important concept if the points are representing atoms). Note that there are two "families" of points: seen in the diagram as filled in circles along with

others that are not filled. The heavy dashed lines are part of the Delaunay triangulation but they are special because these dashed lines connect two points each one in a different family (or perhaps we should say each in a different *chain*).
The solid lines are called Voronoi lines and they can be constructed once the Delaunay triangulation is defined. Note that each point is surrounded by and separated from its neighbours by a set of these solid Voronoi lines. The special heavy solid line in the figure essentially defines a piecewise linear separation between the two different families of points.



As you might have guessed by now, we want to do this type of thing in 3D space. In this case, the Delaunay tessellation is made up of tetrahedrons instead of triangles and the Voronoi piecewise linear line will become a triangulated surface in 3D space.

You should download the Delaunay Starter Script that is available on the course website. The script contains a function called `computeTetrahedrals` that you will be using. It is not necessary for you to understand the script within this function definition. You can regard it as a "black box" that can be described by knowing its input and output:

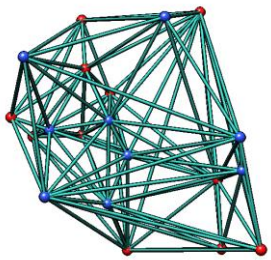
```
def computeTetrahedrals(data)
```

Input: `data` is a list of Point objects designating the coordinates of atoms in the 3D space.
Output: a list of lists. You may consider the output to be a list of tetrahedrons where each tetrahedron is a list of four integers. Each integer can be used as an index into the `data` list given to the function. Consequently, by using the indexes in conjunction with the `data` input, you can derive the coordinates of each tetrahedron so that it can be used to generate display structures in the Chimera window.

Issues:
Unfortunately, `computeTetrahedrals` does not accept atom objects so you will need to retain some extra data structure that allows you to map the index of a tetrahedron list back to the atom that generated the coordinates related to that index.

Continue your reading of the starter script, by studying the mainline program. It uses the `BuildStructure` class to set up two families of atoms (oxygen atoms and nitrogen atoms). No bonds are put in place, we just need two set of atoms to illustrate the use of the Delaunay tetrahedrons. Eventually, you will replace this code with a script that fetches a protein and creates two atom families (atoms in a binding site and atoms in the ligand within the site).

You should study the script that is after the "Processing of coordinates" comment. It extracts coordinates from the atoms in the model and passes them to the `computeTetrahedrals` function. The remainder of the mainline script works with the tetrahedrons to get all the edges that make up the tetrahedrons. These edges are represented in the display using spindles. You should see something like this:



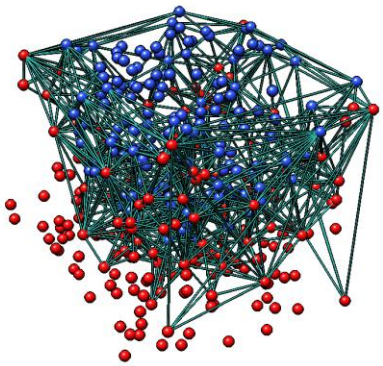
If you carefully inspect the Chimera version of the figure you should be able to see how the tetrahedrons are filling the space defined by the randomly placed atoms. While the atoms have randomly determined positions, the family designation is not random. The blue atoms are in a paraboloid bowl that is nestled in a larger collection of red atoms. If you change the number of atoms to 200 you will have a more realistic number of atoms but the tessellation will be a dense incomprehensible mess of spindles. How do we get a separating surface from this? To find out do the following exercises:

Exercise 6: Discarding useless tetrahedrons

In the Delaunay figure we saw that the heavy dashed lines were used to link the points in different families. In other words, a triangle in the tessellation was only useful if it contained at least one edge that linked two points in different families.

In this exercise you should modify the starter script so that it extracts useful tetrahedrons from the list of tetrahedrons returned by the `computeTetrahedrals` function. If a tetrahedron has all four points in the same family then it is useless and can be ignored.

Modify the script to show spindles that are representing edges in tetrahedrons that are useful. Your structure will have fewer tetrahedrons and will yield a display that is something like this:



Many of the tetrahedrons are gone. You will be left with a collection of spindles that resembles a bird's nest.

Exercise 7: Discarding "skinny" tetrahedrons

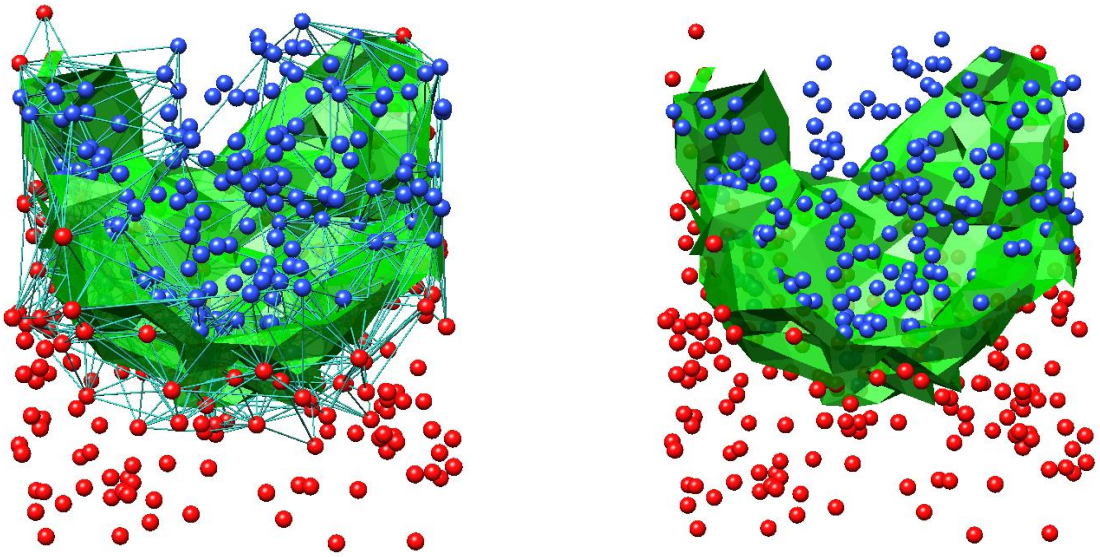
You will have tetrahedrons in the tessellation that involve atoms from different families but the atoms are so far apart that they would not be significantly interacting in a real molecule. Do more modifications to the previous script so that the tetrahedrons are filtered again, this time with respect to edge length of the tetrahedrons. Your modification will scan the useful tetrahedrons and it should generate a *final* set of tetrahedrons such that the longest side in a tetrahedron is less than a threshold value (call it `edgeLengthThreshold`). Try a threshold value of 4.0. The spindle display should look something like a more carefully constructed bird's nest. You will see that the display is much like the previous figure, but without the longer tetrahedrons sticking out beyond the nest.

Exercise 8: Generating the Voronoi surface

You now have the tetrahedrons needed to define the separating surface. The surface will be constructed from a set of surface "patches" each patch defined by a final useful tetrahedron. A useful tetrahedron can be characterized as belonging to one of the following three cases:

Case 1: a single blue atom and three red atoms (this has three "crossover" edges of the tetrahedron gong from the blue family to the red family)
Case 2: a single red atom and three blue atoms (three crossover edges of the tetrahedron going from the blue family to the red family)
Case 3: two blue atoms and two red atoms (four crossover edges of the tetrahedron go from the blue family to the red family)

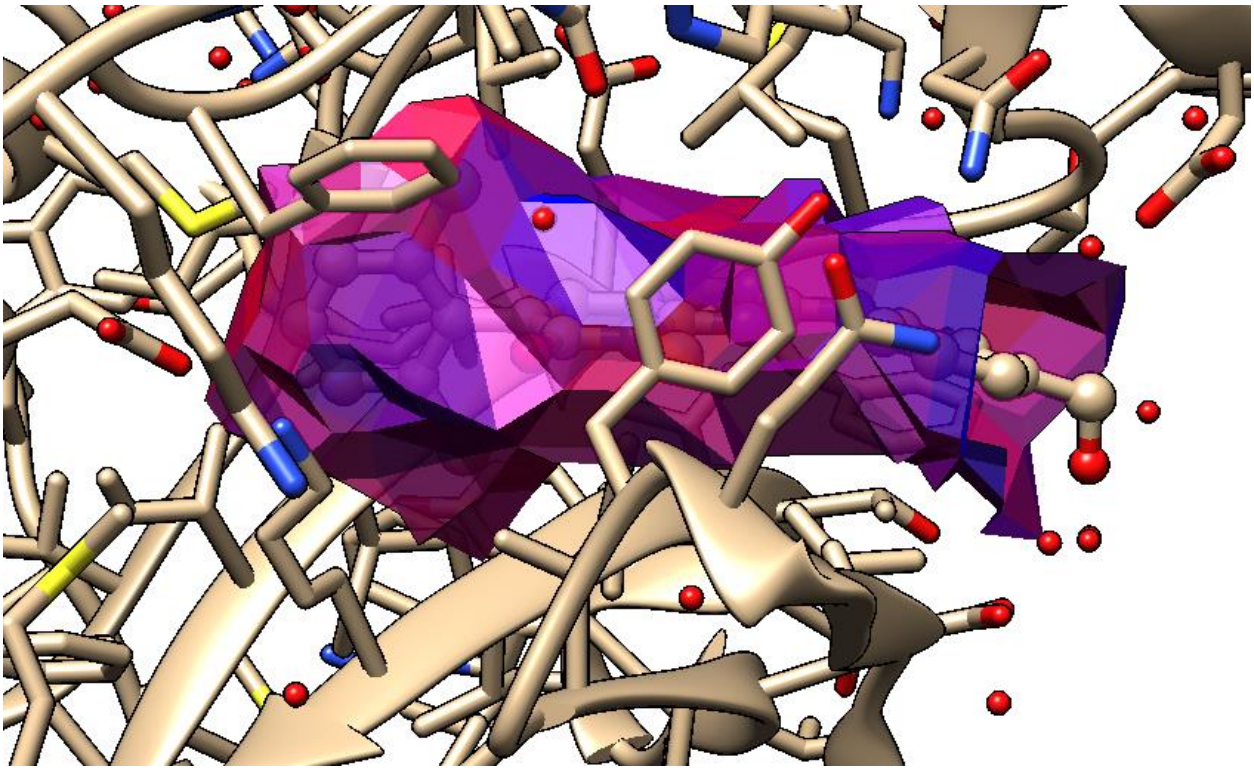
To get a patch from a tetrahedron you will **compute the center of each crossover edge** and these points will define the vertices of the surface patch. For Case 1 and Case 2 the surface patch is a triangle and for Case 3 the surface patch is a quadrilateral. Note that patches for tetrahedrons that share a face will join in a continuous fashion because both patches share two vertices. You should use the `Surfaces` class in the `StructBio` package to draw the surface. A `Surfaces` object will have an `addPolygon` function that allows you to add a patch with a specified colour and transparency. The first display of the next figure shows what to expect with 400 atoms and a green separating surface. Note that Chimera's lighting model will put a glaring highlight on certain planes and not on others. Consequently, the surface has a "crystalline" appearance that is not as compelling as the smooth surface of Intersurf. Observe that the spindles are missing in the second display of the figure. You should be able to arrange this by going to the Model Panel and removing the check mark in the S box corresponding to the spindles model.



Exercise 9: Generating a separating surface between ligand and binding site

For this **last** (☺) exercise you should replace the code that generates the random atoms with a script that fetches a PDB file for a protein that has a ligand (for example, 1OPK which is a tyrosine kinase with an inhibitor called P16). As before, you will have two families of atoms. One family will be the atoms in the ligand and the other family will be the atoms in the binding site that are close to the ligand. Recall 2(b) of Assignment 4 where we have the definition of "close to the ligand": We will say that a residue is close to the ligand if any one of its atoms is within 4 Angstroms of some atom in the ligand. That is, the shortest inter atomic distance is less than or equal to 4. Your script should use a function that returns a list of standard residues that are close to a specified ligand. This can be easily and quickly done in O(n) time if you use the Shell class. With the atom families defined, your script can proceed with the generation of a surface around the ligand.

You should experiment with different settings of `edgeLengthThreshold`. Smaller values of this threshold will lead to a smoother "less choppy" surface but at the risk of losing some needed tetrahedrons resulting in a surface with holes. The next figure presents the display when 1OPK was used:



To give the surface color some meaning you should give it a color that represents the distance between atoms on either side of the patch. In the preceding figure, the patch is coloured red if the average length of the crossover edges is 3. Angstroms and coloured blue if the average length of the crossover edges is equal to `edgeLengthThreshold`. For average lengths between these two extremes a linear interpolation should be done to determine the values in the colour tuple[2]. Transparency was set to 0.8 so that the ligand atoms could be recognized. Note that ligand atoms are in ball & stick representation and the binding site atoms are in a stick representation.

[2] If you were trying to improve the affinity of the drug candidate, the blue areas of the surface would indicate places where more structure could be added to the ligand (taking care to ensure that the molecule would still be capable of docking into the site).