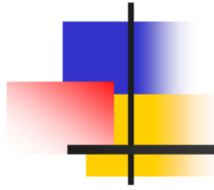

CS483 Assignment #2: Linear Algebra & Protein Geometry

Marks: (The assignment will be marked out of 80).

Ex. 1: Q**1**: [4], Q**2**:[3], Q**3:**[3] , Q**4**:[1+2+3 = 6], Q**5:**[3], Q**6**:[4], Q**7:**[6]

Ex. 2: [5+5+8+8 = 26], **Ex. 3**: [4+6+5+10 = 25]

Due date: Thursday Feb. 2. **"Early Bird" Due date: Tues. Jan. 31 to get 8 bonus marks.**

Exercise 1: Linear Algebra

1. Let $S = \{v^{(1)}, v^{(2)}, ..., v^{(n)}\}$ be an orthogonal set of vectors. This means that $v^{(i)\mathrm{T}} v^{(j)} = 0$ if $i \neq j$ and $v^{(i)\mathrm{T}} v^{(j)} \neq 0$ if $i = j$. Prove that S is a linear independent set.

2. Suppose A and B are $n \times n$ orthogonal matrices. Show that $C = AB$ is an orthogonal matrix.

3. Show that if A is an orthogonal matrix, then det(A) is either 1 or –1.

4. Suppose A is 3 by 4 matrix with the following Singular Value Decomposition:

$$A = USV^{\mathrm{T}} = \begin{bmatrix} u^{(1)} & u^{(2)} & u^{(3)} \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v^{(1)} & v^{(2)} & v^{(3)} & v^{(4)} \end{bmatrix}^{\mathrm{T}}$$

   where U and V are both real orthogonal matrices. Note that $u^{(i)}$ represents the ith column of U and $v^{(i)}$ represents the ith column of V.

   a) What is the rank of A?

   b) What are the eigenvalues and eigenvectors of $A^{\mathrm{T}}A$?

   c) What is $Av^{(1)}$?

5. Prove that if A is a symmetric invertible matrix then A^{-1} is also symmetric.

6. Avoiding the expansion of the following determinant, give an argument showing that this determinant is zero for all nonzero values of a, b, c and d:

$$\det\left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ a & b & c & d \\ 1/a & 1/b & 1/c & 1/d \\ b+c+d & a+c+d & a+b+d & a+b+c \end{bmatrix}\right).$$

7. Suppose you are given an $m \times n$ matrix X, and it is necessary to compute the inverse of XX^{T}. It is possible that XX^{T} does not have an inverse because $\det\left(XX^{\mathrm{T}}\right) = 0$. However, the matrix $XX^{\mathrm{T}} + \lambda I$ is always invertible if $\lambda > 0$. Provide a short proof of this statement.

Exercise 2: Problems Dealing with Inter-atomic Distances

There are several applications in which it is necessary to find all atoms that are near neighbours of some query atom. For example, we might ask: "Which atoms are within 6 Angstroms of the alpha carbon atom in the 9th residue of chain "A"? Here is a list of the requirements for our problem:

i. The "within" distance is called a *distance threshold* and it will be a pre-specified value that is held constant for the entire execution of the script. Let us designate this distance threshold as `dThreshold`. We assume "within" means the inter-atomic distance is *less than or equal* to this threshold distance.
ii. This exercise requires that you implement a script that will read a protein file from the PDB and then build a data structure that will help to answer a near neighbour query in **constant time**.
iii. The construction of this data structure should be done in O(*n*) time where *n* is the number of atoms in the PDB file.

In practise `dThreshold` will be some small floating point value such as 5.0. Consider a rectangular 3D "bounding box" that can contain all atoms of the given protein. To be more specific, we want the smallest bounding box with sides having lengths that are *exact multiples* of `dThreshold`. We will consider the inside of the box to be a 3D array of cubes each cube having a length, width, and depth equal to `dThreshold`. Each cube in the box will have 3D **integer** "grid" coordinates (c_x, c_y, c_z) that identify the cube's position within the bounding box. Figure 1 shows the front of the bounding box as a red grid, the bottom of the bounding box as a blue grid and the left side of the box as a green grid[1].

a) Write a Python function called `convCoordsToGridCoords` specified by the following:
Input: The 3D coordinates of a single atom along with any other information that you might need.
Return: The grid coordinates of the cube that would contain this atom.
Hints: You should pass the value of `dThreshold` to this function and (depending on how you implement the function) it could also use *xMin*, *yMin*, and *zMin*. Here *xMin* represents the minimum value of the *x* component of the atom coordinates for all atoms in the PDB file. The other values are defined in a similar fashion. To get the grid coordinates corresponding to atom coordinates (x, y, z) consider the integer part of the calculation $[(x, y, z) – (xMin, yMin, zMin)]$/`dThreshold` .

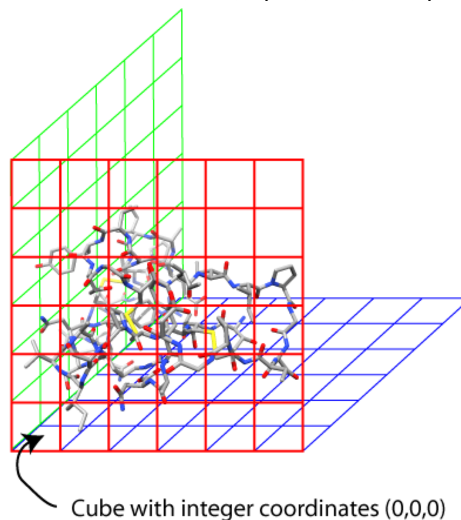

Figure 1: Three sides of the bounding box.

[1] Figure 1 is only for purposes of explanation. This exercise does not require that you generate such a scene.

b) How are we going to use the grid coordinates of the atoms? A possibility is have them act as indices for a huge three dimensional array, each entry of the array containing a list of indices that designate those atoms in the atom list of the protein that are in the cube with these grid coordinates. There is a big drawback to this scheme: It would require a lot of space and many of the entries would be empty because the protein typically does not intersect all cubes in the bounding box. Instead, you should write another function called `buildGridDictionary` that does the following:
Input: A list of the atoms in the protein along with any other information that you might need.
Return: A dictionary (call it `gridDictionary`) that maps grid coordinates to lists of atom indices. More specifically: a dictionary key will be a 3-tuple of integers representing the grid coordinates of a cube inside the bounding box. The dictionary contents associated with a key will be a list of atom indices for those atoms that can be found in the cube with grid coordinates specified by that key. An atom index, call it `ix`, can be used to get an atom object by using a reference such as: `yourProt.atoms[ix]`. Once you have the atom object you can get any of its object attributes, for example, the position of the residue that contains it, the chain ID, its coordinates, etc.

c) Write a function called `getListOfIndicesOfNbrAtoms` that does the following:
Input: The 3D coordinates of a single "query" atom along with any other information that you might need.
Returns: A *near neighbour list* specifying the indices of all atoms that are located within distance `dThreshold` from the query atom.
Hints: First convert the 3D coordinates of the query atom to its grid coordinates. This will designate a "central cube" that is surrounded by 26 other cubes having grid coordinates that are easily computed. Use the `gridDictionary` to get indices of all atoms in the 3 by 3 by 3 set of cubes. Some of these atoms will be within `dThreshold` of the query atom and others will be beyond this threshold, so you will have to do a final check to verify that an atom is within threshold. If the distance between an atom and the query atom is less than or equal to `dThreshold` then put its index into the return list. The return list should *not* include the index of the query atom. These hints will have to be somewhat modified when the query atom is in a cube that is on a face of the bounding box. Note that the amount of computation that is done to handle a query is not related to the total number of atoms because (realistically) there is a small upper bound on the number of atoms that can be found in a set of 27 cubes. Consequently, a query runs in constant time.

d) To show that your functions are properly working, write a mainline program that performs the following animation:
   i) Use the `raw_input()` function to get a PDB identifier from the user.
   ii) Fetch the PDB file from the Protein Data Bank (or from your local disk).
   iii) Assume `dThreshold` equal to 4.0.
   iv) Build the grid dictionary.
   v) Display the protein in a wire representation by using the following code in your script:

```
runCommand('~ribbon')
runCommand('sel #0')
runCommand('display sel')
runCommand('represent wire sel')
runCommand('~sel')
```

vi) Run a loop that goes over all alpha carbon atoms in the protein and for each such atom print its atom identification string and residue type followed by the list of near neighbour atoms each designated with its atom identification string and residue type. For example, the first alpha carbon of 1HIV with `dThreshold` = 4.0 would produce:

```
1.A@CA PRO
Near neighbours:
1.A@N PRO
1.A@CD PRO
1.A@CG PRO
1.A@CB PRO
99.B@O PHE
1.A@C PRO
2.A@N GLN
2.A@CA GLN
1.A@O PRO
353.A@O HOH
```

As shown above, an atom identification string will be built as follows:
residue position string + "." + chainId string + "@" + atom name string.

While this list is being printed change the display representation of the alpha carbon to "ball and stick" using a statement such as:

```
runCommand("represent bs #0:" + atomIdentificationString)
```

and change the representation of the near neighbour atoms to "stick" using a statement such as:

```
runCommand("represent stick #0:" + atomIdentificationString).
```

When going to the next alpha carbon, you should recover the wire representation by using:

```
runCommand("represent wire #0:" + atomIdentificationString).
```

Note: the string "`#0:`" is designating model[0] in the list of open models.
To pace this "animation" you can have the user type any key or you can do `time.sleep(0.1)` before the next new alpha carbon is handled. To make runCommand and time.sleep operational you should start your code with:

```
from chimera import runCommand
import time
```

What to hand in: A well-documented Python script (both the print version for TA comments and the .py file so that the TA can run your program).

Exercise 3:
Computing the Interaction Energy between a Pair of Rotamers

In a simple study of protein flexibility we might choose to keep the protein backbone fixed in space while altering the positions of atoms in the side-chains. As a further simplifying assumption, we can assume a side-chain conformation is restricted to be that of a *rotamer*. To see how Chimera can give you the rotamers for a particular side-chain consider the following code snippet:

```
import chimera
from chimera import runCommand
import Rotamers
from Rotamers import getRotamers, useRotamer, NoResidueRotamersError

targetPath = "1CRN"
prot = chimera.openModels.open(targetPath, type="PDB")[0]

runCommand("set bg_color white")
runCommand("~ribbon")
runCommand("repr stick")
runCommand("show")


exampleRes = prot.findResidue(28)
try:
    rot_L = getRotamers(exampleRes)[1]
except NoResidueRotamersError:
    print ix, residue.type, "no rotamers"

numRots = len(rot_L)
permission = ""
rotIx = 0
while permission == "":
    useRotamer(exampleRes, [rot_L[rotIx]])
    print "Rotamer number ", rotIx, " of ", numRots
    permission = raw_input("")
    rotIx += 1
    rotIx = rotIx%numRotS
```

If you run the script, Chimera will fetch the 1CRN protein and will replace the residue at index 28 with one of its rotamers. If you press your Enter key the loop in the program continues and the next rotamer is displayed. As you keep pressing the Enter key, the script will cycle through all 6 rotamers and then start again with the first rotamer, etc. Study this script so that you understand how to use the two functions: `getRotamers` and `useRotamer`.

In an application such as *side-chain packing*[2], it is usually required that we compute the interaction energy between two rotamers corresponding to two residues that are nearby to each other (close enough to interact or even collide). To do this you will need to write scripts for the following functions:

a) `EA(a,b)`:
   Input: Atom objects `a` and `b`.
   Output: Interaction energy $EA(a,b)$ between atoms `a` and `b`. We will use the formula from the paper referenced in the footnote at the bottom of this page:

$$EA(a,b) = \begin{cases} 0 & r \geq R_{a,b} \\ 10 & r \leq 0.8254 R_{a,b} \\ 57.273\left(1 - \frac{r}{R_{a,b}}\right) & \text{otherwise} \end{cases}$$

[2] http://ttic.uchicago.edu/~jinbo/ps/SideChain-RECOMB.pdf

where r is the distance between atoms a and b and $R_{a,b}$ is the sum of their radii. Note that atom `radius` can be derived as an attribute of an atom object in Chimera.

b) `ER(m,i,k,j)`:

We use the notation r_m^i to designate the rotamer with index i being used for the side-chain site having residue index m.

Input: The integers m, i, k, and j are used to specify the two rotamers r_m^i and r_k^j.

Output: Interaction energy between the two rotamers r_m^i and r_k^j specified by the arguments.

The value of ER is to be computed as the sum of all possible $EA(a,b)$ values:

$$\sum_{a \in r_m^i} \sum_{b \in r_k^j} EA(a,b)$$

Note that you will need the two functions: `getRotamers` and `useRotamer`. The first function is used to get a list of rotamers and the second function is used to replace a particular side-chain with a rotamer. Once the rotamer is in place, we can use Chimera to get the coordinates of all its atoms. Doing the same for a nearby residue will allow you to derive all possible (a,b) pairings needed as inputs for the EA formula. Be careful: Not all residues have rotamers (glycine, for example). The notation $a \in r_m^i$ means that a is an atom "in" the rotamer and not in the backbone. The next function will deal with the backbone atoms.

c) `ES(r,i)`:

Input: Integers m and i used to specify the rotamer r_m^i.

Output: The *intrinsic* energy (sometimes called the "self" energy) of rotamer r_m^i.

The value of ES is to be computed as the sum of all possible $EA(a,b)$ values:

$$\sum_{a \in r_m^i} \sum_{b \in B} EA(a,b)$$

In this formula B is the set of atoms in the backbone that are within 10 Angstroms of the alpha carbon associated with residue r. Use functions from the previous exercise to find these atoms.

d) **Rotamer Dead-End Elimination (DEE)**:

The *side-chain packing problem* can be stated as follows: Assuming the protein backbone is held in a fixed conformation, how can we determine the conformation of all the side-chains so that the total energy is minimized? The problem is NP hard but researchers have many algorithms that employ heuristic approaches to get a reasonable "solution" of the problem. Side-chain packing can be employed in protein folding algorithms and in the analysis of binding pockets during drug design studies. To make the problem more tractable various simplifications can be imposed:

i. As stated, we assume the positions of the backbone atoms are fixed.
ii. We assume that the conformation of a residue will be chosen from its rotamer set.
iii. We use a simple energy function.

Before any heuristic strategy is applied, a researcher will strive to eliminate any rotamer that cannot be part of the solution. Suppose we compare two rotamers for the *same* residue. We will refer to rotamers at the same residue site as *sibling* rotamers. The energy contribution made by a rotamer will be its intrinsic energy plus the sum of all its interaction energies with nearby rotamers situated at other nearby side-chain sites. If the "best" energy contribution of r_m^i is **larger** than the "worst" energy

contribution of a sibling r_m^j then rotamer r_m^i can be eliminated because its sibling will always be better at reducing the total energy independent of the conformations of nearby side chains.
Here is a mathematical expression for the previous statement. We can remove r_m^i from further consideration if we can find a sibling r_m^j such that the following inequality is true:

$$ES\left(r_m^i\right)+\sum_{k\in N(m)}\min_p\left\{ER\left(r_m^i,r_k^p\right)\right\}>ES\left(r_m^j\right)+\sum_{k\in N(m)}\max_p\left\{ER\left(r_m^j,r_k^p\right)\right\}$$

where $\min_p\left\{ER\left(r_m^i,r_k^p\right)\right\}$ is the minimum (best) energy possible between rotamer r_m^i of the side-chain site at residue[m] and **any** rotamer r_k^p of the side-chain site at residue[k]. In other words, we extract the minimum interaction energy as p ranges across all the siblings for residue[k]. A similar definition holds for the max function term. We use the notation $k\in N(m)$ to mean that k is an integer in the set of integers designated as $N(m)$. Here $N(m)$ is the set of indices of residues that are *nearby* residue m. For the purposes of this exercise, a nearby residue will have its alpha carbon within 10 Angstroms of the alpha carbon of residue[m]. (Again, you need the routines of the previous exercise).

Write a script that will do Dead-End Elimination for all the side-chain sites of a given protein.
Input: The PDB ID of a protein.
Output: Print a list of rotamers that have been eliminated by your DEE algorithm. Rotamers should be identified by residue type, residue index and rotamer index.

As a test, use the first model in the PDB file with ID = 1E0G.

Be careful about the organization of the computations. Try to avoid duplication of effort in the computations producing the min and max values. The script is likely to have a long execution time (especially for large proteins) even if you do manage to avoid duplication of computations.

What to hand in: A well-documented Python script (both the print version for TA comments and the .py file so that the TA can run your program).