# 1 Calculating backbone dihedral angles given the coordinates

## Calculation of Phi and Psi dihedral angles

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Variables | | | | | | | |
| Backbone | N | … | Ca | … | C | … | N |
| Labels | A |  | B |  | C |  | D |
| ω Angle | B – C – D – B | | | | | | |
| ϕ Angle | C – D – B – C | | | | | | |
| ψ Angle | A – B – C – D | | | | | | |

Table 1: Overview of a schematic backbone with corresponding labels. Angels are shown by label.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Vectors | Normal Vector | Unit Vector | Angle between vectors | Atan2 |
| X1 = B – A | N1 = X1 x X2 | û= |  |  |
| X2 = C – B | N2 = X2 x X3 | û |
| X3 = D – C |

Table 2: Table containing the formulas for the vectors to eventually calculate the torsion angles. Formulas are based on labels from table 1.

The phi and psi angles are calculated by looking at the atoms in the backbone. Using the coordinates of these atoms, we can draw a plane in a ‘space’. The angle between these planes is called the torsion angle. The first plane is between A – B – C, and the second plane is between B – C – D. The psi and phi angles are thus calculated using the formulas in Table 2.

## Strategy for assigning secondary structure

*Describe a strategy for assigning secondary structure based on phi and psi angles. You should assign the secondary structure type for each residue, given the backbone coordinates of a PDB file. The classification should be based on the phi and psi angles of the residue from the equations above; you may use the idea of a ramachandran plot to define your assignment criteria. Here a classification in three groups [alpha, beta, other] would be sufficient. Please do not exceed 75 words in your explanation.*

Ramachandran plot -> angles -> groups [alpha, beta, other]

## Implement you strategy for assigning secondary structure

#!/usr/bin/python

# The line above tells the system which python command it should use

### RUNNING THE SCRIPT ####

# your can run the script by typing:

# > ./readPDB.py pdb\_file\_name

# or by typing

# > python readPDB.py pdb\_file\_name

# Note: you first need to make your script executable by the user with:

# > chmod u+rx readBlastTable.py

### PACKAGES ####

# First we import some packages that we may need

**import** sys # system

**import** os # operating system

**import** re # regular expression

**import** math # math modules

### FILE NAMES ###

# Here we set the filenames that the program uses.

fn\_out = "phi\_psi.txt"

### GLOBAL VARIABLES ###

# Here we put the "Global Variables" - variables that maybe accessed by all functions in this script

# keeps the information of the pdb coordinates and

# in the form: pdbcoord[chain][resnum][atomtype] = coordinates

# where chain is a character, resnum a number, and atomtype a string

# coordinates is a list of the x,y and z coordinates

pdbcoord = **dict**()

# keeps information of the sequence, stored in the format

# pdbseq[chain][resnum]

pdbseq = **dict**()

############## FUNCTION DEFINITIONS ####################

# It is important to keep your code modular,

# i.e. split the code into different funtions

# so that it is easy to interpret, read and debug

#########################################

### readPDB, reads the PDB file given as a command line argument

### and stores the information in pdbcoord and pdbseq

**def** readPDB():

filename = None

# check if a single filename is given

**if**(**len**(sys.argv) != 2):

**print** "Usage: script\_name file\_name"

exit(1)

# get the file name

**else**:

filename = sys.argv[1]

#open file for reading

**print** "opening file ", filename

# "r" indicates you open the file to read

#try opening the file, and give warning if not possible

**try**:

infile = **open**(filename, 'r')

**except** IOError: # In case of IOError return empty collection

**print** "Error: Cannot open PDB file " + filename + "."

exit(1)

# Loop over all the lines in the file:

# "readlines()" will return a list with all the lines

**for** line **in** infile.readlines():

# rstrip remove the "\n" from the line

line = line.rstrip()

# take the first 4 characters of the line

first4 = line[0:4]

#test if atom or hetatom

**if**(first4 == "ATOM"):

# get all the info you need, see

# http://www.wwpdb.org/documentation/format32/sect9.html#ATOM

# get atom type

atom\_type = line[12:16].strip()

# get amino acid type

aa\_type = line[17:20].strip()

# get residue number

res\_num = **int**(line[22:26])

# chain in PDB file

chain = line[21]

# get the coordinates

xcoord = **float**(line[30:38])

ycoord = **float**(line[38:46])

zcoord = **float**(line[46:54])

#store all information

# if chain does not exists create new entry

**if** chain **not** **in** pdbcoord:

pdbcoord[chain] = **dict**()

pdbseq[chain] = **dict**()

# if resnum does not exists create new entry

**if** res\_num **not** **in** pdbcoord[chain]:

pdbcoord[chain][res\_num] = **dict**()

# if atom\_type does not exists, create new entry

**if** atom\_type **not** **in** pdbcoord[chain][res\_num]:

pdbcoord[chain][res\_num][atom\_type]=**dict**()

# store coordinates as a vector

pdbcoord[chain][res\_num][atom\_type] = [xcoord,ycoord,zcoord]

# store sequence

pdbseq[chain][res\_num] = aa\_type

#end if ATOM

# end loop readlines()

# close the infile

infile.close()

# end function readPDB

########################################################

### calculates dihedral angle:

### a1, a2, a4 & a4 give coordinates of atoms.

### the atoms ai are given as lists (vectors) in the format a1 = [x,y,z]

**def** calculateDihedral(a1,a2,a3,a4):

dihedral = 0

# START CODING HERE

# calculate normal vectors to planes defined by a1,a2,a3 and a2,a3,a4

# you may use the functions "cross\_product","dot\_product" and "magnitude" defined below

# you can also use the python math function "math.atan2" and "math.degrees"

# get 2 vectors per plane:

d1 = vector(a1, a2)

d2 = vector(a3, a2)

d3 = vector(a3, a4)

# get normal vectors

v1 = cross\_product(d1, d2)

v2 = cross\_product(d2, d3)

# calculte sin and cos

sin = dot\_product(cross\_product(unitVector(v1), unitVector(v2)), unitVector(d2))

cos = dot\_product(unitVector(v1), unitVector(v2))

dihedral = math.degrees(math.atan2(sin, cos))

# END CODING HERE

**return** dihedral

# end function calculateDiheral

###############################

**def** unitVector(v1):

**return** [item/magnitude(v1) **for** item **in** v1]

###############################

**def** vector(b1, b2):

**return** [b1[0]-b2[0], b1[1]-b2[1], b1[2]-b2[2]]

###############################

### takes cross product of vectore v1, v2

### returns a vector

**def** cross\_product(a,b):

i= a[1]\*b[2] - a[2]\*b[1]

j= - (a[0]\*b[2] - a[2]\*b[0])

k= a[0]\*b[1] - a[1]\*b[0]

**return** [i,j,k]

# end function cross\_product

##############################

### takes dot product of vectors v1 & v2

### returns a float

**def** dot\_product(v1,v2):

ans = v1[0]\*v2[0]+v1[1]\*v2[1]+v1[2]\*v2[2]

**return** ans

# end function dot\_product

####################################

### computes the magnitude for a 3D vector

### returns a float

**def** magnitude(v1):

ans = math.sqrt(v1[0]\*v1[0]+v1[1]\*v1[1]+v1[2]\*v1[2])

**return** ans

# end function magnitude

#####################################

### assigns secondary structure

### given the phi and psi angle or a residue

### feel free to change function definition

### to make your method more advanced, or try alternative methods

**def** assignSecondaryStructure(phi,psi):

# START CODING

# Use a typical Ramachandran plot to assign secondary structure

# choose from "loop","alpha","beta" or "other"

secondary\_structure = "other"

**if** psi > 90 **and** phi < -20 **and** phi > -170:

secondary\_structure = 'beta'

**elif** psi > -70 **and** psi < -20 **and** phi > -170 **and** phi < 10:

secondary\_structure = 'alpha'

**elif** psi > 0 **and** psi < 90 **and** phi > 45 **and** phi < 90:

secondary\_structure = 'alpha'

# END CODING

**return** secondary\_structure

# end function assignSecondaryStructure

### FUNCTION printHits ####

# This function prints all the keys and values from globale variable "BLAST\_HITS"

**def** printPhiPsi(fn\_out):

# open outfile, to write

outfile = **open**(fn\_out, 'w')

# obtain chains from the dictonary "pdbcoord", and sort the list

list\_chains = **sorted**(pdbcoord.keys())

# loop over all chains

**for** chain **in** list\_chains:

# obtain residue numbers from the dictonary "pdbcoord", and sort the list

list\_residue\_numbers = **sorted**(pdbcoord[chain].keys())

# loop over residue numbers

**for** residue **in** list\_residue\_numbers:

# catch "KeyError" exceptions from dictonary"

# makes sure you program does not crash when a certain

# atom type does not exist, it does give you a warning

**try**:

# START CODING

# here you need to decide which atoms you should use

# for calculating your dihedral angles

# you should use the function "calculateDihedral(a1,a2,a3,a4)"

# variable that will hold final phi value

phi = None

# variable that will hold finals psi value

psi = None

# you can use and change the following example,

# to test if a residue exists in pdbcoord

previous\_res = residue - 1

next\_res = residue + 1

## check if previous residue exists

# determine phi

**if** (previous\_res **not** **in** pdbcoord[chain]):

phi = None

**else**:

a1 = pdbcoord[chain][previous\_res]['C']

a2 = pdbcoord[chain][residue]['N']

a3 = pdbcoord[chain][residue]['CA']

a4 = pdbcoord[chain][residue]['C']

phi = calculateDihedral(a1, a2, a3, a4)

# determine psi

**if** (next\_res **not** **in** pdbcoord[chain]):

psi = None

**else**:

a1 = pdbcoord[chain][residue]['N']

a2 = pdbcoord[chain][residue]['CA']

a3 = pdbcoord[chain][residue]['C']

a4 = pdbcoord[chain][next\_res]['N']

psi = calculateDihedral(a1, a2, a3, a4)

# END CODING

# handle any key errors

**except** KeyError **as** error:

**print**

"WARNING KeyError: ", error, " in residue ", chain, residue

# get the amino acid

aa = pdbseq[chain][residue]

sse = assignSecondaryStructure(phi, psi)

# print to file

**print** >> outfile, chain, residue, aa, phi, psi, sse

# end loop list\_residue\_numbers

# end loop list\_chains

# close outfile

outfile.close()

**print** "written file", fn\_out

# end function printPhiPsi

############## PROGRAM ##################

# first read in the PDB file

# call the function "readPDB"

readPDB()

# Print out the uniprot identifiers together with their evalues

# call the function "printPhiPsi"

printPhiPsi(fn\_out)

## Discussion of secondary structure assignment strategy

There is a Helix-turn-Helix in the PDB file (1TIM.pdb), residues 131 t/m 153, my program misclassified a few residues, see Table 3. Although the overall result is quite accurate, some residues are misclassified. To improve the classification the angles for classification can be made more strictly or be more loosely defined. There are no set values for dihedral angles.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Classified by program** | | | | | | **DSSP** |
| **Chain** | **Residue** | **AA** | **PSI** | **PHI** | **Group** | |
| A | 131 | LYS | -119.660761282 | 126.664576478 | Beta | alpha |
| A | 132 | LEU | -4.63769487301 | -74.4829834106 | other | alpha |
| A | 133 | ASP | -36.9299325009 | -48.9887037886 | alpha | alpha |
| A | 134 | GLU | -77.3276703808 | -45.4370875286 | alpha | alpha |
| A | 135 | ARG | -43.2712788048 | -51.8429428277 | alpha | alpha |
| A | 136 | GLU | -48.2961088807 | -67.9868912668 | alpha | alpha |
| A | 137 | ALA | -50.9878899232 | -16.796792709 | other | other |
| A | 138 | GLY | 98.8684825578 | 20.3356888547 | other | other |
| A | 139 | ILE | -83.3039832975 | 24.6927121196 | other | alpha |
| A | 140 | THR | -50.8693031272 | -55.396672181 | alpha | alpha |
| A | 141 | GLU | -40.2221850298 | -67.8530877116 | alpha | alpha |
| A | 142 | LYS | -53.2919685577 | -57.0784940807 | alpha | alpha |
| A | 143 | VAL | -44.8524734273 | -76.808537603 | other | alpha |
| A | 144 | VAL | -50.9536036741 | -53.4518093923 | alpha | alpha |
| A | 145 | PHE | -53.2679988274 | -51.6362428278 | alpha | alpha |
| A | 146 | GLN | -51.9843343988 | -64.3905148269 | alpha | alpha |
| A | 147 | GLU | -45.2384821291 | -52.7362111044 | alpha | alpha |
| A | 148 | THR | -53.5234698107 | -38.5135607168 | alpha | alpha |
| A | 149 | LYS | -55.6457519463 | -62.5753633509 | alpha | alpha |
| A | 150 | ALA | -39.0833035121 | -63.826687557 | alpha | alpha |
| A | 151 | ILE | -43.7762031131 | -72.6315341871 | other | alpha |
| A | 152 | ALA | -45.5900564728 | -54.4972837434 | alpha | alpha |
| A | 153 | ASP | -43.0504120286 | -62.30372823 | alpha | alpha |

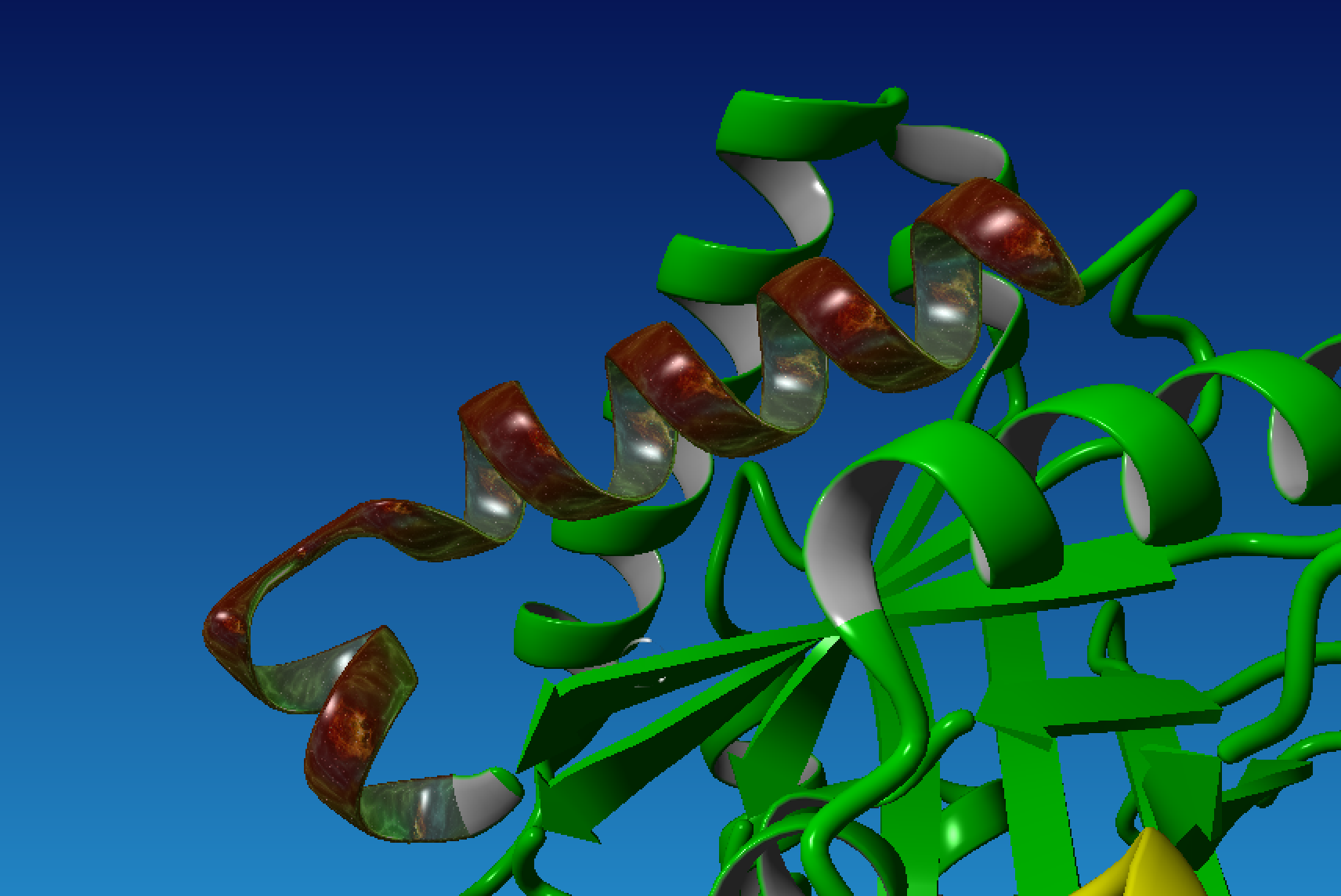
Table 3: Table showing a helix-turn-helix event that is classified by a python program and in the dssp file. There are a few misclassifications in the group column for residues 131, 132, 139, 143 and 151.

Figure 1: YASARA view of residue 131 t/m 153 showing a helix turn helix structure (red).

## Dihedrals versus hydrogen bonds

*No currently available secondary structure assignment program (e.g. Stride or DSSP), rely solely on dihedral angles for their assignment. Describe the pros and cons of using hydrogen bonds versus dihedral angles for secondary structure assignments. Please do not exceed 75 words in your explanation.*

Hydrogen atoms have high affinity for forming hydrogen bonds. Ideally, all hydrogen atoms in secondary structures want to be completely satisfied. By looking at the amount of unsatisfied hydrogen atoms it is possible to deduce if a particular structure is likely to be the correctly folded structure. A down side of this approach would be the computational power that is necessary to check this for every hydrogen atom. Using dihedral angles is easier since we only need the coordinates for each atom in the backbone. However, the downside of this approach is that this shows only which angles could be related to a secondary structure.

# Propensities for amino acids to be buried

## Implement propensities to be buried

|  |
| --- |
| #!/usr/bin/python  # The line above tells the system which python command it should use  ### RUNNING THE SCRIPT ####  # your can run the script by typing:  # > ./readDSSP.py pdb\_file\_name  # or by typing  # > python readDSSP.py directory\_name  # Note: you first need to make your script executable by the user with:  # > chmod u+rx readDSSP.py  ### PACKAGES ####  # First we import some packages that we may need  **import** sys # system  **import** os # operating system  **import** re # regular expression  **import** math # math modules  ### FILE NAMES ###  # Here we set the filenames that the program uses.  #file with info about unolded surface accessibility  fn\_unfolded = "/Users/Erik/Documents/Master/Structural Bioinformatics/Assignment 1/AccUnfold.data"  # output file  fn\_out = "propensity\_buried.txt"  ### GLOBAL VARIABLES ###  # Here we put the "Global Variables" - variables that maybe accessed by all functions in this script  # stores unfolded accessibility scores  # indexed by amino acid type: unfolded\_acc[aa]= accessibility\_value  unfolded\_acc = **dict**()  # stores the counts for amino acids  # indexed by amino acid type: all\_aa\_count[aa]=count  all\_aa\_count = **dict**()  # stores the counts for buried amino acids  # indexed by amino acid type: buried\_aa\_count[aa]=count  buried\_aa\_count = **dict**()  ############## FUNCTION DEFINITIONS ####################  # It is important to keep your code modular,  # i.e. split the code into different funtions  # so that it is easy to interpret, read and debug  ########################################################  ### reads file with surface accessibility values for  ### different amino acid types, stores info in unfolded\_acc[aa]  **def** readUnfoldedAcc(fn\_unfolded):  # open file for reading  **print** "opening file ", fn\_unfolded  # "r" indicates you open the file to read  #try opening the file, and give warning if not possible  **try**:  infile = **open**(fn\_unfolded, 'r')  **except** IOError: # In case of IOError return empty collection  **print** "Error: Cannot open PDB file " + fn\_unfolded + "."  exit(1)  **for** line **in** infile.readlines():  line.rstrip()  # split into columns  fields = line.split()  aa = fields[0]  acc = **float**(fields[2])  # store unfolded accessibility value  unfolded\_acc[aa]=acc  # end for line  # close file  infile.close()  # end function readUnfoldedAcc  #########################################  ### lists files in directory given as command line argument  ### gets all files ending in "./dssp"  ### calls readDSSP for each dssp file  **def** readDir():  # check if a single filename is given  **if**(**len**(sys.argv) != 2):  **print** "Usage: script\_name directory\_name"  exit(1)  # get the file name  **else**:  fn\_dir = sys.argv[1]  #check if directory exists  **if**( **not** os.path.isdir(fn\_dir)):  **print** fn\_dir, "is not an exisiting directory"  exit(1)  # obtain file names from directory  list\_fn = **sorted**(os.listdir(fn\_dir))  # call readDSSP on all .dssp files  **for** fn **in** list\_fn:  # check if dssp file  **if**(os.path.splitext(fn)[1] == ".dssp"):  readDSSP(os.path.join(fn\_dir,fn))  # end for list\_fn  # end function readDir  #########################################  ### readPDB, reads the PDB file given as a command line argument  ### and stores the information in pdbcoord and pdbseq  **def** readDSSP(filename):  **print**(filename)  #open file for reading  **print** "opening file ", filename  # "r" indicates you open the file to read  #try opening the file, and give warning if not possible  **try**:  infile = **open**(filename, 'r')  **except** IOError: # In case of IOError return empty collection  **print** "Error: Cannot open PDB file " + filename + "."  exit(1)  # keep track of the first line starting with " #"  start\_reading = False  # Loop over all the lines in the file:  # "readlines()" will return a list with all the lines  **for** line **in** infile.readlines():  # rstrip remove the "\n" from the line  line = line.rstrip()  # take the first 4 characters of the line  first4 = line[0:4]  **if**(first4 == " # "):  start\_reading = True  # only start processing file after " #" line  **elif**(start\_reading):  # get information from the DSSP file, see  # http://swift.cmbi.ru.nl/gv/dssp/  # get amino acid type  aa\_type = line[13]  # skip amino acids marked '!'  **if**(aa\_type **in** '!abcd'):  **continue**  # get residue number  # print(type(line[5:10].strip())  res\_num = **int**(line[5:10].strip())  # get chain  chain = line[11]  # get surface accessible area  acc = **int**(line[34:38].strip())  # check if amino acid type has been seen before, if not create entry  **if**(aa\_type **not** **in** all\_aa\_count):  all\_aa\_count[aa\_type] = 0  buried\_aa\_count[aa\_type] = 0  # count all amino acids  all\_aa\_count[aa\_type] = all\_aa\_count[aa\_type] +1.0  # decide if amino acid is buried  buried = decideIfBuried(aa\_type,acc)  **if**(buried):  # count buried amino acids  buried\_aa\_count[aa\_type] = buried\_aa\_count[aa\_type] +1.0  #end if ATOM  # end loop readlines()  # close the infile  infile.close()  # end function readPDB  #################################  ### function takes aa, the amino acid type  ### acc the solvent accessibility  ### you should return True, only if the residue is buried  **def** decideIfBuried(aa,acc):  buried = False  # START CODING HERE  # you can use unfolded\_acc[aa], to decide if a residue is buried  **return** acc / unfolded\_acc[aa] < 0.07  # STOP CODING HERE  **return** buried  # end function decideIfBuried  ###########################################  # This function prints the propensities, based on the counts  **def** printPropensities(fn\_out):  # open outfile, to write  outfile = **open**(fn\_out,'w')  # obtain all amino acid types  list\_aa = **sorted**(all\_aa\_count.keys())  total\_buried = 0  total\_aa\_count = 0  # START CODING HERE  # you should calculate the propensity for each amino acid type to be buried  # you can use all\_aa\_count and buried\_aa\_count[aa]  # you can use the following loop structure over all amino acids:  # for aa in list\_aa:  # ... all\_aa\_count[aa] ...  # ... buried\_aa\_count[aa] ...  #  # to print to the output file you can use:  # print >> outfile, aa, propensity\_buried  final = []  **for** aa **in** list\_aa:  # count total buried and aa  total\_buried += buried\_aa\_count[aa]  total\_aa\_count += all\_aa\_count[aa]  # calculate fraction all  fraction\_all = total\_buried / total\_aa\_count  **for** aa **in** list\_aa:  # calculate fraction buried  fraction\_buried = buried\_aa\_count[aa] / all\_aa\_count[aa]  # calculate propensity buried  propensity\_buried = fraction\_buried / fraction\_all  **print** >> outfile, aa, **round**(propensity\_buried, 3)  # END ANSWER  # END CODING HERE  # close outfile  outfile.close()  **print** "written file", fn\_out  # end function printPropensities  ############## PROGRAM ##################  # read accessibilities in unfolded form  readUnfoldedAcc(fn\_unfolded)  # go through the directory given as a command line argument  readDir()  # Print out the propensities to be buried  printPropensities(fn\_out) |

## Table with propensities to be buried

The following table was created using the DSSP library containing 3051 dssp files. Amino acids that are not present in ‘AccUnfold.data’ were skipped.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Amino Acid** | **Triple Letter Code** | **Single Letter Code** | **Property** | **Propensities** |
| Alanine | Ala | A | Non-Polar Aliphatic | 1.378 |
| Cysteine | Cys | C | Hydrophobic/Slightly Polar | 1.947 |
| Aspartic Acid | Asp | D | Polar (-) | 0.402 |
| Glutamic Acid | Glu | E | Polar (-) | 0.274 |
| Phenylalanine | Phe | F | Non-Polar Aromatic | 1.758 |
| Glycine | Gly | G | Hydrophillic&Hydrophobic | 0.976 |
| Histidine | His | H | Polar (+) | 0.723 |
| Isoleucine | Ile | I | Non-Polar Aliphatic | 1.832 |
| Lysine | Lys | K | Polar (+) | 0.14 |
| Leucine | Leu | L | Non-Polar Aliphatic | 1.702 |
| Methionine | Met | M | Non-Polar Aliphatic | 1.481 |
| Asparagine | Asn | N | Polar Neutral | 0.521 |
| Proline | Pro | P | Non-Polar Aliphatic | 0.658 |
| Glutamine | Gln | Q | Polar Neutral | 0.403 |
| Arginine | Arg | R | Polar (+) | 0.31 |
| Serine | Ser | S | Polar Neutral | 0.804 |
| Threonine | Thr | T | Polar Neutral | 0.866 |
| Valine | Val | V | Non-Polar Aliphatic | 1.765 |
| Tryptophan | Trp | W | Non-Polar Aromatic | 1.442 |
| Tyrosine | Tyr | Y | Non-Polar Aromatic | 1.158 |

Table 4: Table with propensities for the amino acids calculated by the readDSSP.py script.

## Discussion on propensities

*Information regarding amino acids was found on* [*http://www.russelllab.org/aas/*](http://www.russelllab.org/aas/) *.*

Alanine; The side chain is very non-reactive and is thus rarely directly involved in protein function and present in just about all non-critical protein contexts. The propensity seems normal for an amino acid that can almost fit in anywhere.

Cysteine; Plays an important role for sulfur bridges and is often found in the protein binding sites, therefore it is logical that the propensity is quite high.

Aspartic Acid; Prefers to be on the surface of protein structures, is hydrophilic and generally forms hydrogen bonds with Polar(+) amino acids. It is not surprising that the propensity is very low for this amino acid.

Glutamic Acid; Being charged and polar, Glutamates, prefer generally to be on the surface of protein, thus the low propensity is quite accurate.

Phenylalanine; Being hydrophobic, Phenylalanine prefers to be buried in protein hydrophobic cores, thus the propensity is accurate.

Glycine; there is much conformational flexibility in glycine, it can reside in parts of protein structures that are forbidden to all other amino acids. So, the propensity of around 1 makes sense.

Histidine; Histidine’s is rather ambiguous about whether it prefers to be buried in the protein core or exposed to solvent. This makes it an ideal residue for protein functional centers. This does not really show in the propensity.

Isoleucine; Being hydrophobic, Isoleucine prefers to be buried in protein hydrophobic cores which shows in the propensity.

Lysine; one can find Lysine’s where part of the side-chain is buried, and only the charged portion is on the outside of the protein. However, this is by no means always the case, and generally Lysine’s prefer to be on the outside of proteins. I would expect the propensity to be slightly higher.

Leucine; Being hydrophobic, Leucine prefers to be buried in protein hydrophobic cores. This strives with the calculated propensity.

Methionine; Being hydrophobic, Methionine prefers to be buried in protein hydrophobic cores. This strives with the calculated propensity.

Asparagine; Being polar, Asparagine prefers generally to be on the surface of proteins. I would expect the calculated propensity to be slightly lower.

Proline; Being hydrophilic and the preference for turn structures means that Prolines are generally on the protein surface, which is also what we calculated in the propensity.

Glutamine; Due to its polarity, glutamine is generally found on the surface of a protein. This is not what we observed in our calculations.

Arginine; Is generally found on the outside of proteins, this is what we observed.

Serine; Is a fairly indifferent amino acid, and often found in tight turns. Therefore, serine is almost found as often within the protein as on the surface. This is close to what we observed.

Threonine; As a fairly indifferent amino acid, it can be found on the surface as well as in the core of a protein. This is close to what we observed.

Valine; Due to its hydrophobic nature, valine is often found within a protein. This is also what we observed.

Tryptophan; Due to its hydrophobic nature, tryptophan is often found within a protein. This is also what we observed.

Tyrosine; Being partially hydrophobic, tyrosine is often found on the inside and outside of a protein. This is in accord with our observations.

## 2.4

Calculating the propensities on a small database (few proteins) will most likely lead to biased results. As the representation of each amino acid in a smaller database is not guaranteed. Therefore, using a larger database will have a better indication of the location of amino acids within different protein structures. A small database of very simple proteins can also lead to biased propensities of amino acids.

# Ramachandran Plots

## Contour lines

A Ramachandran plot shows the distribution of phi and psi angles derived from a pdb file. Where each cluster shows a conformational energy. The clusters are made for angles with similar energetically allowed regions. Contour lines are drawn around clusters with similar conformational energy. The contour lines indicate a core region and an allowed region. These show a clear difference for outlying angles. Being derived from pdb files means that this is knowledge based and not ab invito.

In the case of a reference set with higher quality strctrures, we could expect the contours line to be more closely shaped around the dots. ASince the contours line mainly shows a region/cluster from a percentage in this contour. Out liners currently are ffecting the range of thece contour. I think its safe to expect where will be less outlines in high quality data. Which would result in smaller, but more accurate regions.