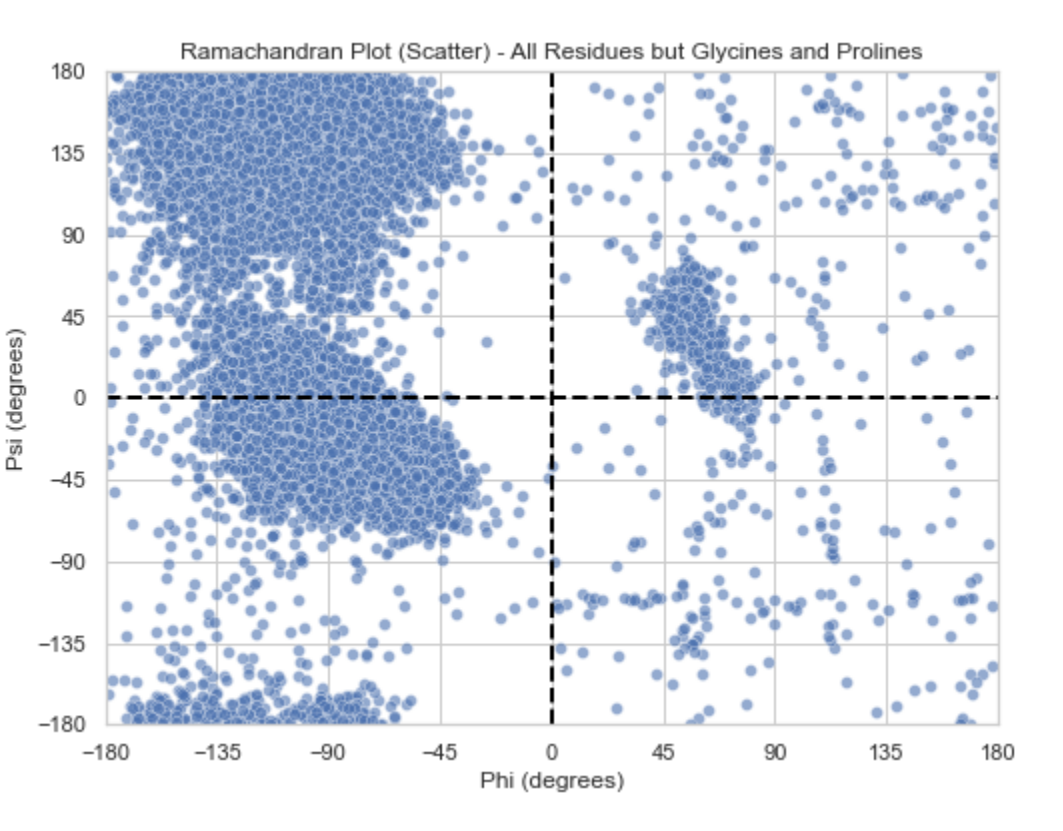
Ramachandran Report

Author: Yu-Sheng Chen

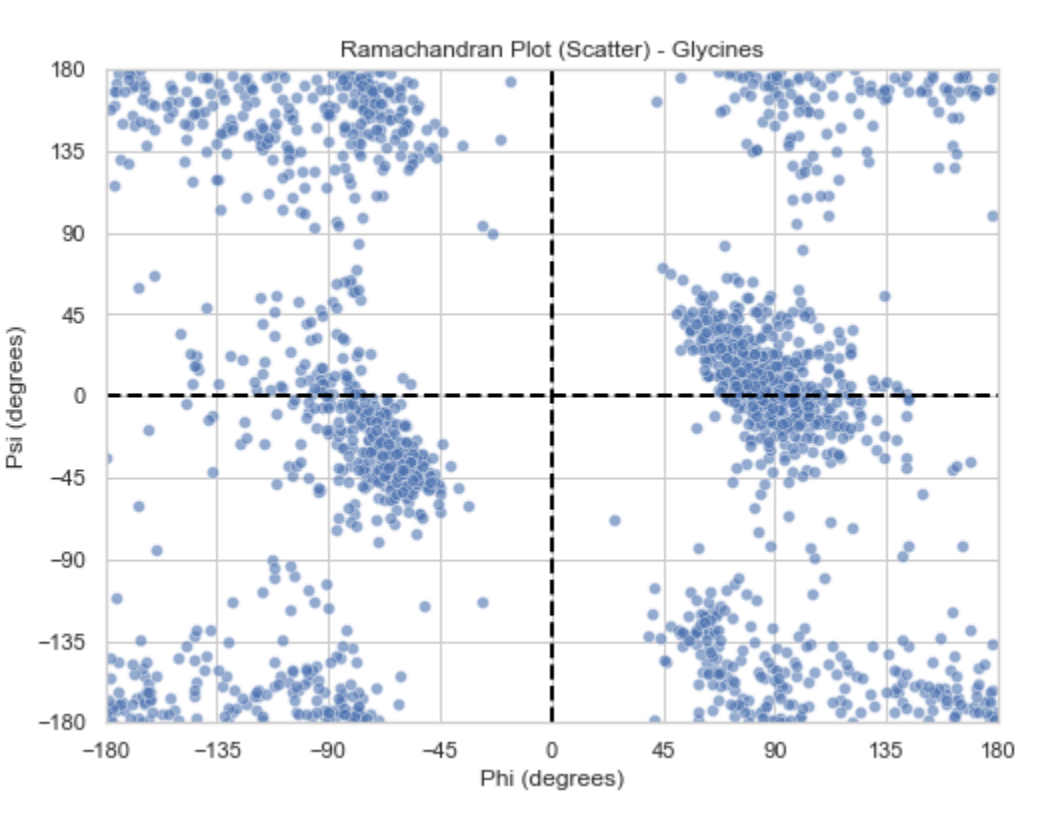
The code can be found at the end of this document. Or please visit the Github repo: https://github.com/YSChen0609/PDB_Protein_Analysis

(1) Ramachandran plots

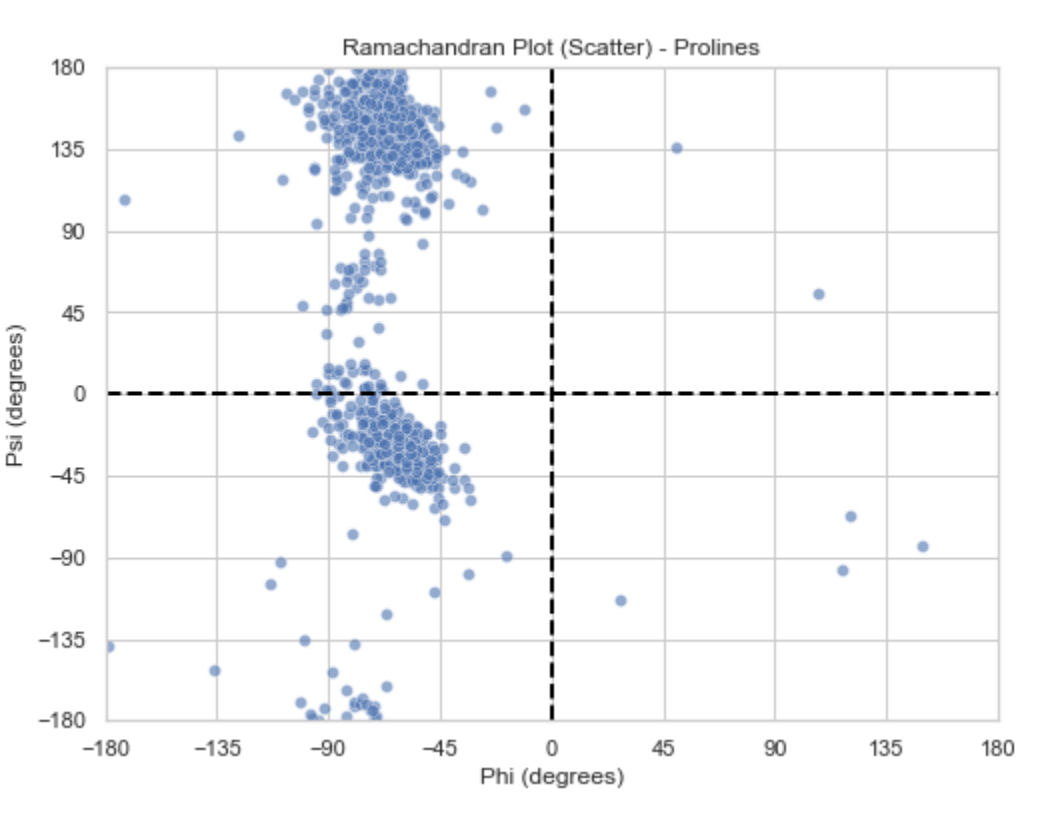
(a) All residues but glycines and prolines.



(b) Only glycines

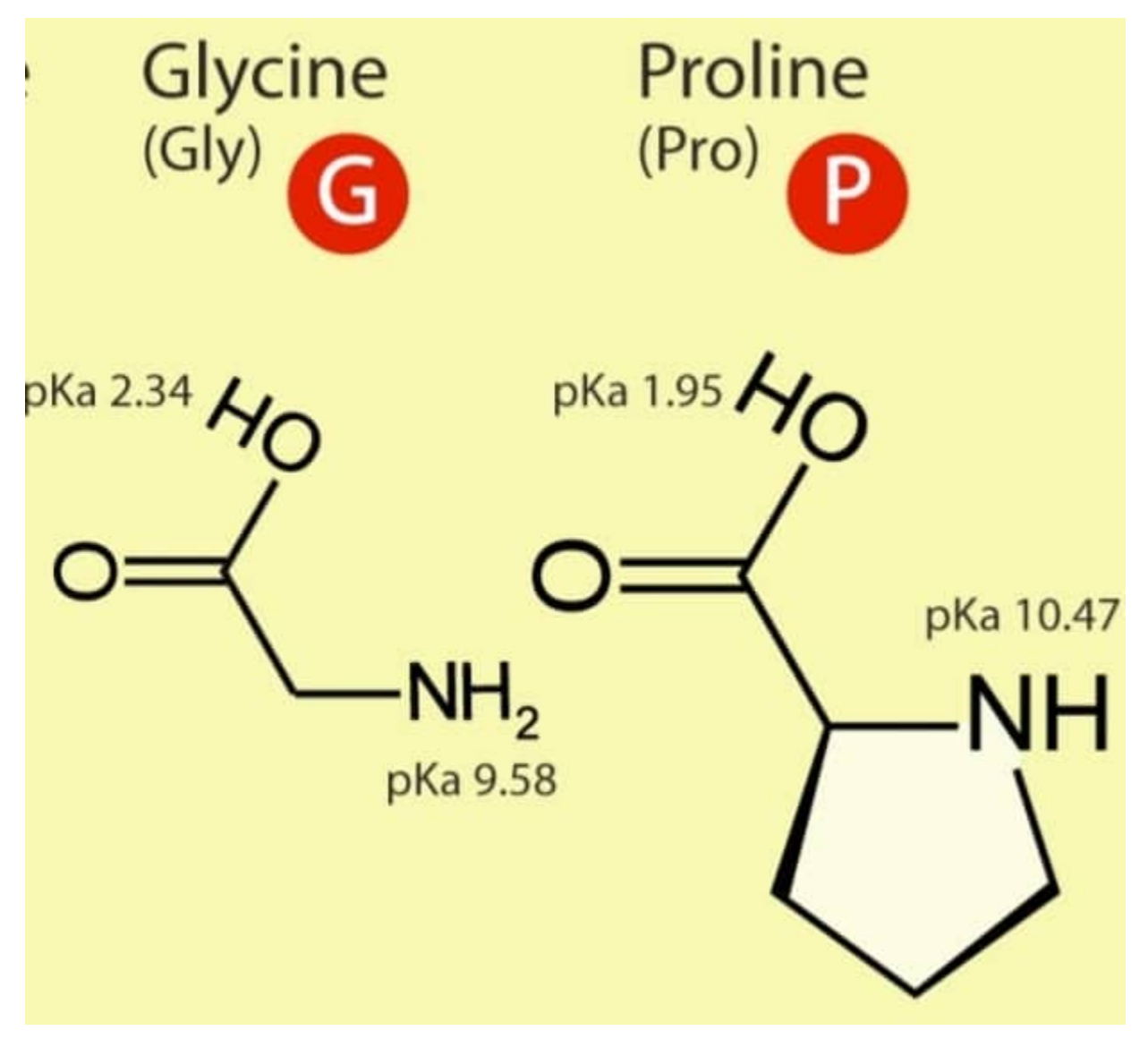


(c) Only prolines



(2) Compare plot (b) to (a), describe the difference in their secondary structure patterns and explain such a difference for glycines.

We can easily observe from the graphs that glycines have a wider range of (phi, psi) angle pairs when compared to all residues but glycines and prolines. This is because glycines has the smallest structure among all 21 amino acids (see below), which allows it to have a more diverse angle in the secondary structures, including the relatively rare left-handed helix.



(3) Check plot (c), describe and explain the pattern again for prolines.

In contrast with glycines, prolines have a more narrow (phi, psi) pattern when compared to all residues but glycines and prolines. We can see that prolines rarely show up in the 1st and 4th quadrant. Also, the phi angles largely range only from -45 to -90 degrees. This can also be explained by the structure of prolines. It is the fact that proline has a much larger structure (the N group) than glycines, thus its diversity of possible (phi, psi) angle pairs is limited.

Appendix: Code

In [ ]:
```
import requests
import random
import re
from string import Template
import pandas as pd
import time
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

In [ ]:
```
# tools
def log_method(func):
    def wrapper(*args, **kwargs):
        print(f"Starting {func.__name__}...")
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Finished {func.__name__}. Time taken: {end_time - start_time} seconds.")
        return result
    return wrapper

def sampleWithConstraints(samplePool, pattern):
    """
    randomly return ONE sample from the samplePool using the constraint string (regex pattern string)
    """
    random.shuffle(samplePool)

    # case: match string (attempt)
    for item in samplePool:
        if pattern.match(item):
            return item

    # case: no match string
    return False


def normVecCross(x,y,z):
    """
    Get the unit norm of a plane defined by vectors (v x w), where
        - v = y-x
        - w = z-y
        - x, y, z are (1*3)d pandas series
    """
    x = x.to_numpy(dtype=float)
    y = y.to_numpy(dtype=float)
    z = z.to_numpy(dtype=float)

    cross_product = np.cross(y-x, z-y)

    return cross_product/np.linalg.norm(cross_product)
```

In [ ]:
```
class PDB_Parser():
    def __init__(self, ATOM_DATA_FILE_PATH = f'./AtomData_{int(time.time())}.csv'):
        # Sequence clusters at <identity> % sequence identity clustering
        self.CLUSTER_FILE_URL = "https://cdn.rcsb.org/resources/sequence/clusters/clusters-by-entity-30.txt"
        self.PDB_URL_TEMPLATE = Template("https://files.rcsb.org/download/${pdb_id}.pdb")
        self.FASTA_URL_TEMPLATE = Template("https://www.rcsb.org/fasta/entry/${pdb_id}")
        self.structQueue = []
        self.structQueueSIZE = 125
        self.STRUCT_FORMAT = re.compile('^.{4}_[0-9]$')
        self.CHAIN_FORMAT = re.compile('\|Chains? (.*?)\|')
        self.AUTH_CHAIN_FORMAT = re.compile('\[auth (.+)\]')
        self.BACKBONE_ATOMS = (' N  ',' CA ',' C  ')
        self.atomData = []
        self.ATOM_DATA_FILE_PATH = ATOM_DATA_FILE_PATH


    @log_method
    def getStruct(self):
        """
        loop over the largest 100 clusters (the first 100 lines),
        and extract one random structure for each cluster.
        """
        r = requests.get(self.CLUSTER_FILE_URL, stream=True)
        self.structQueue.clear()

        for line in r.iter_lines():
            line = line.decode("utf-8")
            if len(self.structQueue) == self.structQueueSIZE:
                break
            if line:
                # randomly sample a structure
                structs = line.split()
                struct_id = sampleWithConstraints(structs, self.STRUCT_FORMAT)

                if not struct_id:
                    continue

                self.structQueue.append(struct_id)


    def getFASTA(self, struct_id):
        """
        A subroutine that fetches the FASTA File, and return the chain_id
            Note: struct_id (instances in structQueue) = {pdb_id}_{pe_id}
        """
        fasta_url = self.FASTA_URL_TEMPLATE.substitute(pdb_id=struct_id[:4])
        r = requests.get(fasta_url, stream=True)

        for line in r.iter_lines():
            line = line.decode("utf-8")
            # match the line with the struct_id
            if line[0] != ">":
                continue

            if line[1:7] == struct_id:
                chain_ids = self.CHAIN_FORMAT.findall(line)[0].split(", ")

                # Select ONE of the chain_id(s)
                while len(chain_ids) != 0:
                    chain_id = chain_ids.pop(-1)
                    auth_chain_id = self.AUTH_CHAIN_FORMAT.search(chain_id)
                    if auth_chain_id:
                        return auth_chain_id.group(1)
                    else:
                        return chain_id

        return False


    def getPDB(self, struct_id, chain_id):
        """
        Get the PDB File, and parse it to a clean format for further usage.
         Return a list of dicts containing atomData (segment of the whole)
            Note:
                - struct_id (instances in structQueue) = {pdb_id}_{pe_id}
                - chain_id is from FASTA (use method getFASTA())
                - for the line(str) slicing indices, please referr to the PDB file format
        """
        pdb_url = self.PDB_URL_TEMPLATE.substitute(pdb_id=struct_id[:4])
        r = requests.get(pdb_url, stream=True)
        segAtomData = []

        for line in r.iter_lines():
            line = line.decode("utf-8")
            if line.startswith('ATOM'):
                if line[21] == chain_id and line[12:16] in self.BACKBONE_ATOMS:
                    segAtomData.append({
                            'atom_name' : line[12:16],
                            'residue_name' : line[17:20],
                            'x' : line[30:38],
                            'y' : line[38:46],
                            'z' : line[46:54]
                        })

            elif len(segAtomData) != 0 and line[21] != chain_id:
                return segAtomData

        return False

    @log_method
    def getAtomData(self, save_to_csv=False):
        """
        Get the FULL Atom Data for further analysis, go through the subroutine:
            - getFASTA
            - getPDB
         and return a dataframe with columns: atom_name, residue_name, x, y, z.
        """
        cnt = 0
        for struct_id in self.structQueue:
            chain_id = self.getFASTA(struct_id)
            seg_atomData = self.getPDB(struct_id, chain_id)

            if seg_atomData:
                # skip the cif format ones
                # TODO: To improve, modify to handle the cif ones
                self.atomData += seg_atomData
                cnt += 1

        atom_data = pd.DataFrame.from_dict(self.atomData)

        if save_to_csv:
            atom_data.to_csv(self.ATOM_DATA_FILE_PATH)
            print(f'The Atom Data is saved at {self.ATOM_DATA_FILE_PATH}.')

        print(f'{cnt} Structures Included.')

        return atom_data


    def main(self, save_to_csv=False):
        self.getStruct()

        return self.getAtomData(save_to_csv=save_to_csv)
```

In [ ]:
```
class Ramachandran_Analysis():
    def __init__(self, AtomData=None):
        """
        AtomData is a pandas dataframe having the format (columns): atom_name, residue_name, x, y, z
        """
        self.AtomData = AtomData
        self.AngleData = []
        self.ANGLE_CSV_PATH = f'./Angles_{int(time.time())}.csv'

    @log_method
    def getAngles(self, save_to_csv=False): # TODO: split all chains and calculate
        """
        Use a sliding window to calculate the (phi, psi) angles,
        and return (update) the self.AngleData (list of dict).

        - Phi Angle = arccos(n(C_i-1, N_i, CA_i)*n(N_i, CA_i, C_i))
        - Psi Angle = arccos(n(N_i, CA_i, C_i)*n(CA_i, C_i, N_i+1))

        Note that the sign of the phi angle is the sign of the inner product of
            n(C_i-1, N_i, CA_i) and (C_i-CA_i), sign of psi in a similar fashion.
        """
        if self.AtomData is None:
            print("Atom Data is not imported! Please new a instance and initialize with one!")
            return

        # Find indices of 'CA' (discard the first and last CA since they have unexist angles)
        ca_indices = self.AtomData[self.AtomData['atom_name'] == ' CA '].index[1:-1]
        self.AngleData.clear()

        for ca_idx in ca_indices:

            # get the plane norms and some vectors
            planeNorm_C_N_CA = normVecCross(
                                    self.AtomData.iloc[ca_idx-2, 2:5],    # C_i-1
                                    self.AtomData.iloc[ca_idx-1, 2:5],    # N_i
                                    self.AtomData.iloc[ca_idx, 2:5]       # CA_i
                                )

            planeNorm_N_CA_C = normVecCross(
                                    self.AtomData.iloc[ca_idx-1, 2:5],    # N_i
                                    self.AtomData.iloc[ca_idx, 2:5],      # CA_i
                                    self.AtomData.iloc[ca_idx+1, 2:5]     # C_i
                                )

            planeNorm_CA_C_N = normVecCross(
                                    self.AtomData.iloc[ca_idx, 2:5],      # CA_i
                                    self.AtomData.iloc[ca_idx+1, 2:5],    # C_i
                                    self.AtomData.iloc[ca_idx+2, 2:5]     # N_i+1
                                )

            vec_CA_C         =  (self.AtomData.iloc[ca_idx+1, 2:5].astype(float)       # C_i
                                     - self.AtomData.iloc[ca_idx, 2:5].astype(float)   # CA_i
                                ).to_numpy(dtype=float)

            vec_C_N          = (self.AtomData.iloc[ca_idx+2, 2:5].astype(float)        # N_i+1
                                    - self.AtomData.iloc[ca_idx+1, 2:5].astype(float)  # C_i
                                ).to_numpy(dtype=float)

            # get the sign and cos value of phi angle
            phi_sign = np.sign(np.dot(planeNorm_C_N_CA, vec_CA_C))
            phi_cos = np.dot(planeNorm_C_N_CA, planeNorm_N_CA_C)

            # get the sign and cos value of psi angle
            psi_sign = np.sign(np.dot(planeNorm_N_CA_C, vec_C_N))
            psi_cos = np.dot(planeNorm_N_CA_C, planeNorm_CA_C_N)

            # get the signed (phi, psi) angles
            phi = np.degrees(phi_sign * np.arccos(phi_cos))
            psi = np.degrees(psi_sign * np.arccos(psi_cos))

            self.AngleData.append({
                            'residue_name' : self.AtomData.loc[ca_idx, 'residue_name'],
                            'phi'          : phi,
                            'psi'          : psi
                        })


        self.angle_df = pd.DataFrame.from_dict(self.AngleData)

        if save_to_csv:
            self.angle_df.to_csv(self.ANGLE_CSV_PATH)
            print(f'Angle Data is saved at: {self.ANGLE_CSV_PATH}.')

        return

    def subplot(self, angles_df, df_idx):
        # Create scatter plot
        sns.set(style="whitegrid")
        plt.figure(figsize=(8, 6))
        sns.scatterplot(x=angles_df['phi'],
                        y=angles_df['psi'],
                        alpha=0.6)

        # Set labels and title
        title = ['All Residues but Glycines and Prolines', 'Glycines', 'Prolines']

        plt.xlabel('Phi (degrees)')
        plt.ylabel('Psi (degrees)')
        plt.title(f'Ramachandran Plot (Scatter) - {title[df_idx]}')

        # Set x and y axis limits
        plt.xlim(-180, 180)
        plt.ylim(-180, 180)

        # Set x and y axis ticks
        plt.xticks(np.arange(-180, 181, 45))
        plt.yticks(np.arange(-180, 181, 45))

        # Draw x and y axes as dotted lines
        plt.axhline(0, color='black', linestyle='--', lw=2)  # Horizontal line for x axis
        plt.axvline(0, color='black', linestyle='--', lw=2)  # Vertical line for y axis

        # Show plot
        plt.show()


    def plot(self, angle_filepath=None):
        """
        Generate Ramachandran plots for:
         (a) all residues but glycines and prolines
         (b) all glycines
         (c) all prolines

        Note that the Angle DataFrame should have columns: residue_name, phi, psi
        """

        if angle_filepath is not None:
            angle_data = pd.read_csv(angle_filepath, index_col=0)
        else:
            try:
                angle_data = self.angle_df
                # will raise error if not executed self.getAngles()
            except:
                print('There is no Angle Data to be plot. \
                        It seems you have not executed self.getAngles()!')

        angle_rest = angle_data.loc[~angle_data['residue_name'].isin(('GLY', 'PRO'))]
        angle_gly  = angle_data.loc[angle_data['residue_name'] == 'GLY']
        angle_pro  = angle_data.loc[angle_data['residue_name'] == 'PRO']

        for df_idx, angles_df in enumerate([angle_rest, angle_gly, angle_pro], start=0):
            self.subplot(angles_df, df_idx)

    def main(self):
        self.getAngles(save_to_csv=True)
        self.plot()
```

In [ ]:
```
if __name__ == '__main__':
    # new a PDB_Parser and get the atom data from PDB/FASTA DB
    p = PDB_Parser()
    atom_data = p.main()

    # new a Ramachandran_Analysis and plot the Ramachandran Plot for three subsets
    r = Ramachandran_Analysis(atom_data)
    r.main()
```

```
Starting getStruct...
Finished getStruct. Time taken: 0.6352319717407227 seconds.
Starting getAtomData...
94 Structures Included.
Finished getAtomData. Time taken: 78.37231206893921 seconds.
Starting getAngles...
Angle Data is saved at: ./Angles_1707777955.csv.
Finished getAngles. Time taken: 85.58501076698303 seconds.
```