Supporting Information 2

Elucidation of Catalytic Strategies of Small Nucleolytic Ribozymes from Comparative Analysis of Active Sites

Daniel D. Seith,[1,‖] Jamie L. Bingaman,[1,‖] Andrew J. Veenis,[1] Aileen C. Button,[1,2] Philip C. Bevilacqua[1,3,*]

[1]Department of Chemistry and Center for RNA Molecular Biology, The Pennsylvania State University, University Park, Pennsylvania 16802

[2]Department of Biochemistry, The University of Vermont, Burlington, Vermont 05405

[3]Department of Biochemistry and Molecular Biology, The Pennsylvania State University, University Park, Pennsylvania 16802

[‖]These two authors contributed equally to this work.

*Corresponding author: pcb5@psu.edu

Contents

Supporting Methods (Scripts)

Supporting Methods (Scripts)

<u>I. γ, β, and δ Scissile Phosphate Plugin</u>

```
from Tkinter import *
from pymol import cmd, stored
import math
import numpy as np


def __init__(self):
    self.menuBar.addmenuitem('Plugin', 'command',
                             'Scissile Phosphate Contacts',
                             label='Scissile Phosphate Contacts',
                             command=lambda s=self: getInfo(s))


def getInfo(app):
    root = Tk()

    def get_data():
        resn_name = e2.get()
        resn_name = resn_name.split()
        parent_mol = e1.get()
        atom_name = e3.get()

        # Close GUI window
        root.destroy()
        remove_additional_conformations()
        getDist(atom_name, resn_name[-1], parent_mol)

    l1 = Label(root, text="Selection Name:")
    l1.grid(row=1, column=0)

    e1 = Entry(root)
    e1.grid(row=1, column=1)
    e1.delete(0, END)
    e1.insert(0, "1p1")

    l2 = Label(root, text="Resin Name:")
    l2.grid(row=2, column=0)

    e2 = Entry(root)
    e2.grid(row=2, column=1)
    e2.delete(0, END)
    e2.insert(0, "2AD")
    l3 = Label(root, text="Atom Name:")
    l3.grid(row=3, column=0)

    e3 = Entry(root)
    e3.grid(row=3, column=1)
    e3.delete(0, END)
    e3.insert(0, "N")

    B1 = Button(root, text="Finish", command=get_data)
    B1.grid(columnspan=2)

    mainloop()


cmd.extend('getInfo', getInfo)


def remove_additional_conformations():

    stored.alt_b = []
    cmd.iterate("alt B", "stored.alt_b.append(index)")
    alt_b = stored.alt_b
    if len(alt_b) > 0:
        cmd.remove("alt B")
        print("Atoms included in alternative conformation B were removed.")

    stored.alt_all = []
    cmd.iterate("all", "stored.alt_all.append(alt)")
    alt_all = stored.alt_all

    alt_all_filtered = []
    for a in range(len(alt_all)):
        if len(alt_all[a]) != 0:
            alt_all_filtered.append(alt_all[a])

    additional_conformations = False
    if len(alt_all_filtered) > 0:
        for a in range(len(alt_all_filtered)):
            if alt_all_filtered[a] != 'A':
                additional_conformations = True
    if additional_conformations:
        print("ERROR: Additional conformations are present.")


def getCAtwo(searchAtom, search_resn):
    adenine_atoms = ["N1", "C2", "N3", "C4", "C5", "C6", "N6", "N7", "C8", "N9"]
    adenine_na_one = ["C2", "N3", "C4", "C5", "C6", "C5", "C6", "C8", "N9", "C8"]
    adenine_na_two = ["C6", "N1", "C2", "N3", "C4", "N1", "N1", "C5", "N7", "C4"]

    # purine is referred to in PyMOL as P5P
    purine_atoms = ["N1", "C2", "N3", "C4", "C5", "C6", "N7", "C8", "N9"]
    purine_na_one = ["C2", "N3", "C4", "C5", "C6", "C5", "C8", "N9", "C8"]
    purine_na_two = ["C6", "N1", "C2", "N3", "C4", "N1", "C5", "N7", "C4"]

    # one_deaza_adenine is referred to in PyMOL as 1DP
    one_deaza_adenine_atoms = ["C1", "C2", "N3", "C4", "C5", "C6", "N6", "N7", "C8", "N9"]
    one_deaza_adenine_na_one = ["C2", "N3", "C4", "C5", "C6", "C5", "C6", "C8", "N9", "C8"]
    one_deaza_adenine_na_two = ["C6", "C1", "C2", "N3", "C4", "C1", "C1", "C5", "N7", "C4"]

    cytosine_atoms = ["N1", "C2", "O2", "N3", "C4", "N4", "C5", "C6"]
    cytosine_na_one = ["C2", "N1", "C2", "C4", "C5", "C4", "C6", "N1"]
    cytosine_na_two = ["C6", "N3", "N1", "C2", "N3", "C5", "C4", "C5"]

    inosine_atoms = ["N1", "C2", "N3", "C4", "C5", "C6", "O6", "N7", "C8", "N9"]
    inosine_na_one = ["C2", "N3", "C4", "C5", "C6", "C5", "C6", "C8", "N9", "C8"]
    inosine_na_two = ["C6", "N1", "C2", "N3", "C4", "N1", "N1", "C5", "N7", "C4"]

    guanine_atoms = ["N1", "C2", "N2", "N3", "C4", "C5", "C6", "O6", "N7", "C8", "N9"]
    guanine_na_one = ["C2", "N3", "C2", "C4", "C5", "C6", "C5", "C6", "C8", "N9", "C8"]
    guanine_na_two = ["C6", "N1", "N1", "C2", "N3", "C4", "N1", "N1", "C5", "N7", "C4"]

    # six_deoxy_guanine is referred to in PyMOL as MTU
    six_deoxy_guanine_atoms = ["N1", "C2", "N2", "N3", "C4", "C5", "C6", "N7", "C8", "N9"]
    six_deoxy_guanine_na_one = ["C2", "N3", "C2", "C4", "C5", "C6", "C5", "C8", "N9", "C8"]
    six_deoxy_guanine_na_two = ["C6", "N1", "N1", "C2", "N3", "C4", "N1", "C5", "N7", "C4"]

    six_amino_guanine_atoms = ["N1", "C2", "N2", "N3", "C4", "C5", "C6", "N6", "N7", "C8",
"N9"]
    six_amino_guanine_na_one = ["C2", "N3", "C2", "C4", "C5", "C6", "C5", "C6", "C8", "N9",
"C8"]
    six_amino_guanine_na_two = ["C6", "N1", "N1", "C2", "N3", "C4", "N1", "N1", "C5", "N7",
"C4"]

    uracil_atoms = ["N1", "C2", "O2", "N3", "C4", "O4", "C5", "C6"]
    uracil_na_one = ["C2", "N1", "C2", "C4", "C5", "C4", "C6", "C5"]
    uracil_na_two = ["C6", "N3", "N1", "C2", "N3", "C5", "C4", "C5"]

    na_one_name = "N/A"
    na_two_name = "N/A"

    if "C" in searchAtom:
```

```
    return na_one_name, na_two_name

if search_resn == "A":
    for n in range(0, len(adenine_atoms)):
        if searchAtom == adenine_atoms[n]:
            na_one_name = adenine_na_one[n]
            na_two_name = adenine_na_two[n]
            break

if search_resn == "P5P":
    for n in range(0, len(purine_atoms)):
        if searchAtom == purine_atoms[n]:
            na_one_name = purine_na_one[n]
            na_two_name = purine_na_two[n]
            break

if search_resn == "1DP":
    for n in range(0, len(one_deaza_adenine_atoms)):
        if searchAtom == one_deaza_adenine_atoms[n]:
            na_one_name = one_deaza_adenine_na_one[n]
            na_two_name = one_deaza_adenine_na_two[n]
            break

if search_resn == "I":
    for n in range(0, len(inosine_atoms)):
        if searchAtom == inosine_atoms[n]:
            na_one_name = inosine_na_one[n]
            na_two_name = inosine_na_two[n]
            break

if search_resn == "G":
    for n in range(0, len(guanine_atoms)):
        if searchAtom == guanine_atoms[n]:
            na_one_name = guanine_na_one[n]
            na_two_name = guanine_na_two[n]
            break

if search_resn == "MTU":
    for n in range(0, len(six_deoxy_guanine_atoms)):
        if searchAtom == six_deoxy_guanine_atoms[n]:
            na_one_name = six_deoxy_guanine_na_one[n]
            na_two_name = six_deoxy_guanine_na_two[n]
            break

if search_resn == "N6G":
    for n in range(0, len(six_amino_guanine_atoms)):
            if searchAtom == six_amino_guanine_atoms[n]:
                na_one_name = six_amino_guanine_na_one[n]
                na_two_name = six_amino_guanine_na_two[n]
                break

    if search_resn == "C":
        for n in range(0, len(cytosine_atoms)):
            if searchAtom == cytosine_atoms[n]:
                na_one_name = cytosine_na_one[n]
                na_two_name = cytosine_na_two[n]
                break

    if search_resn == "U":
        for n in range(0, len(uracil_atoms)):
            if searchAtom == uracil_atoms[n]:
                na_one_name = uracil_na_one[n]
                na_two_name = uracil_na_two[n]
                break

    return na_one_name, na_two_name


def getCAtwo_CVC(searchAtom):
    cytosine_atoms = ["N9", "C4", "O01", "N01", "C02", "N02", "C01", "C8"]
    cytosine_na_one = ["C4", "N01", "C4", "C02", "C01", "C02", "C8", "N9"]
    cytosine_na_two = ["C8", "N9", "N9", "C4", "N01", "C01", "C02", "C01"]

    na_one_name = "N/A"
    na_two_name = "N/A"

    for n in range(0, len(cytosine_atoms)):
        if searchAtom == cytosine_atoms[n]:
            na_one_name = cytosine_na_one[n]
            na_two_name = cytosine_na_two[n]
            break

    return na_one_name, na_two_name


def distance(point_one, point_two):
    dist = math.sqrt((point_two[0] - point_one[0]) ** 2 + (point_two[1] - point_one[1]) ** 2 + (
        point_two[2] - point_one[2]) ** 2)
    return dist


def cross_product(point_one, point_two):
    one_cross_two = [point_one[1] * point_two[2] - point_one[2] * point_two[1],
                     point_one[2] * point_two[0] - point_one[0] * point_two[2],
                     point_one[0] * point_two[1] - point_one[1] * point_two[0]]

    return one_cross_two


def vector(point_one, point_two):
    output_vector = [point_two[0] - point_one[0],
                     point_two[1] - point_one[1],
                     point_two[2] - point_one[2]]
    return output_vector


def plane(P1, P2, P3):
    v1 = vector(P1, P2)
    v2 = vector(P2, P3)
    v1_v2 = cross_product(v1, v2)

    rhs = 0
    for n in range(0, len(P)):
        rhs += v1[n] * v1_v2[n]

    plane_equation = [v1_v2[0], v1_v2[1], v1_v2[2], rhs]

    return plane_equation


def dot_product(v1, v2):
    product = v1[0] * v2[0] + v1[1] * v2[1] + v1[2] * v2[2]
    return product


def get_angle_from_coords(P1, P2, P3):
    v1 = vector(P1, P2)
    v2 = vector(P2, P3)

    # Prevent dividing by zero
    if distance(P1, P2) * distance(P2, P3) == 0:
        return 0

    angle_out = math.acos((dot_product(v1, v2)) / (distance(P1, P2) * distance(P2, P3)))
    angle_out = 180 - angle_out * 180.0 / math.pi

    return angle_out
```

```
def get_angle_from_vectors(v1, v2):
    # Prevent dividing by zero
    if vector_length(v1) * vector_length(v2) == 0:
        return 0
    var = math.fabs((dot_product(v1, v2)) / (vector_length(v1) * vector_length(v2)))

    # Prevent inverse cosine domain errors
    if var > 1.0:
        var = 2.0 - var
    angle_out = math.acos(var)

    angle_out = 180 - angle_out * 180.0 / math.pi
    return angle_out


def vector_length(v1):
    # The purpose of this function is to extract the length of a 3-coordinate vector
    vector_len = [x ** 2 for x in v1]
    vector_len = math.sqrt(sum(vector_len))
    return vector_len


def get_angle_between_normal_vectors(h1, p1, p2, p3):
    # The purpose of this function is to take in four points as arguments and determine the angle
between the
    # normal vectors of the planes created by points h1, p1, p2 and p1, p2, p3.

    # Transform each pair of points to 3 vectors
    vector_h1_p1 = vector(h1, p1)
    vector_p1_p2 = vector(p1, p2)
    vector_p1_p3 = vector(p1, p3)

    # Get the cross-product of the two pairs of vectors
    cross_p1 = cross_product(vector_h1_p1, vector_p1_p2)
    cross_p2 = cross_product(vector_p1_p2, vector_p1_p3)

    # Use the two cross products to find the resulting angle
    angle_ab = get_angle_from_vectors(cross_p1, cross_p2)

    # Treat angles near 180 and near 0 as the same (177 would be the same as 3)
    if angle_ab > 90:
        angle_ab = 180 - angle_ab
    return angle_ab
```

```
def direction(P1, P2):
    eqn = [P2 - P1 for P1, P2 in zip(P1, P2)]
    return eqn


def find_length(fictional_proton_direction, over_all_distance):
    length = math.sqrt(
        over_all_distance ** 2 / (fictional_proton_direction[0] ** 2 + fictional_proton_direction[1]
** 2 +
                                  fictional_proton_direction[2] ** 2))
    return length


def extend(point, line_direction, length):
    end_point = np.array([point + length * line_direction for point, line_direction in zip(point,
line_direction)])
    return end_point


def replace_mod_nb(nb_name):
    # Create a list of modified NB names, where the left reflects the old and the right reflects the
new
    modlist = [["2AD", "OMC", "OMU", "A2M", "DA", "DC", "CVC", "DU", "AP3", "GX1",
"DG", "AVC", "3DA", "3AD"],
               ["A", "C", "U", "A", "A", "C", "C", "U", "A", "G", "G", "A", "A", "A"]]
    for n in range(0, len(modlist[0])):
        if nb_name == modlist[0][n]:
            nb_name = modlist[1][n]

    return nb_name


def getAngle(parent_string_array, parent_index_array, contact_string_array,
contact_index_array,
             aoi_atom_name, aoi_resn_name, aoi_index, ca_atom_name, ca_resn_name,
ca_resi_number,
             ca_chain_letter, nucleotide_list):
    # Assign default values for variables
    angle = 0
    ca_prot_atom = "H0"
    ca_prot_coords = [0, 0, 0]
    ca_prot_resn = ["Z"]

    # Replace modified nucleobase name with un-modified name
    ca_resn_name_modded = replace_mod_nb(ca_resn_name)
    # Returns an angle of 0 for atoms that belong to residues that are not listed in the list of
nucleotides
    if ca_resn_name_modded not in nucleotide_list:
        print("NOTE. An angle of 0 was returned for the following atom: CA " + ca_atom_name +
" resn " + ca_resn_name
              + " resi " + ca_resi_number
              + " chain " + ca_chain_letter)
        return 0

    # Get the index and x, y, z coordinates for each atom
    ca_index = getIndex(contact_string_array, contact_index_array, ca_atom_name,
ca_resn_name, ca_resi_number,
                        ca_chain_letter)
    ca_coords = np.array(cmd.get_coords("index " + str(ca_index)))
    ca_coords = [x for x in ca_coords[0]]

    aoi_coords = np.array(cmd.get_coords("index " + str(aoi_index)))
    aoi_coords = [x for x in aoi_coords[0]]

    # Return 0 if the CA is the AOI
    if ca_index == aoi_index:
        return 0

    # Handle primary amines separately because they have two protons
    primary_amine_list = [["N2", "N6", "N4"],
                        ["G", "A", "C"]]

    for x in range(0, len(primary_amine_list[1])):
        if ca_resn_name_modded == primary_amine_list[1][x]:
            primary_amine_list[1][x] = ca_resn_name
            break

    # Assume the contact atom does not already have a proton
    proton_available = False

    # Find all atoms within 8 A of search_name
    stored.adjacent_atom_string = []
    stored.adjacent_atom_index = []
    cmd.iterate("all within 8 of index " + str(aoi_index),
               "stored.adjacent_atom_string.append((resi, name, resn, b, chain))")
    cmd.iterate("all within 8 of index " + str(aoi_index),
"stored.adjacent_atom_index.append((index))")

    adjacent_string_array = np.array(stored.adjacent_atom_string)
    adjacent_index_array = np.array(stored.adjacent_atom_index, dtype=np.int32)
    # Define the adjacent contact atom names
    na_1_atom_name, na_2_atom_name = getCAtwo(ca_atom_name, ca_resn_name_modded)

    # Define the adjacent contact atom names of CVC
    if ca_resn_name == "CVC":
        na_1_atom_name, na_2_atom_name = getCAtwo_CVC(ca_atom_name)

    # Notifies the user if either of the atoms adjacent to the CA could not be identified
    if na_1_atom_name == "N/A" or na_2_atom_name == "N/A":
        print("ERROR. One or both of the atoms adjacent to the CA could not be identified.")

    # Get indices and coordinates for the two adjacent contact atoms
    na_1_index = getIndex(adjacent_string_array, adjacent_index_array, na_1_atom_name,
ca_resn_name, ca_resi_number,
                          ca_chain_letter)
    na_1_coords = np.array(cmd.get_coords("index " + str(na_1_index)))
    na_1_coords = [x for x in na_1_coords[0]]

    na_2_index = getIndex(adjacent_string_array, adjacent_index_array, na_2_atom_name,
ca_resn_name, ca_resi_number,
                          ca_chain_letter)
    na_2_coords = np.array(cmd.get_coords("index " + str(na_2_index)))
    na_2_coords = [x for x in na_2_coords[0]]

    # If getIndex can't find the atom, return an angle of -1
    if ca_index < 0 or na_1_index < 0 or na_2_index < 0:
        print("ERROR. A contact atom or one of its adjacent atoms was assigned an angle of -1.")
        print(ca_atom_name + '\t' + ca_resn_name)
        return -1

    # finds the indices of the amine protons (if any)
    stored.amine_proton_index = []
    cmd.iterate(
        "elem H and resi " + ca_resi_number + " and chain " + ca_chain_letter + " within 1.1 of
index " +
        str(ca_index), "stored.amine_proton_index.append((index))")
    ca_proton_index_list = stored.amine_proton_index

    # Get the list of protons for the contact atom (Ex: the N2G has 2 protons)
    for primary_amine_name, primary_amine_resn in zip(primary_amine_list[0],
primary_amine_list[1]):
        if ca_atom_name == primary_amine_name and ca_resn_name == primary_amine_resn:

            proton_available = True

            # ensures that the plugin identifies only 2 protons
            if len(ca_proton_index_list) != 2:
                print("ERROR. The number of protons found for a primary amine is not 2.")
                return -1

            # Declare a list to hold all potential angles
            ca_angle_list = []

            # Iterate through each proton name and append the contact angle to a list
            for index in ca_proton_index_list:
                ca_proton_coords = np.array(cmd.get_coords("index " + str(index)))
                ca_proton_coords = [x for x in ca_proton_coords[0]]
                ca_angle_list.append(get_angle_from_coords(aoi_coords, ca_proton_coords,
ca_coords))

    if proton_available:
        # Calculate the angle formed between the normal vectors of the prot, ca1, ca2 and ca1, ca2,
ca3
        normal_angle = get_angle_between_normal_vectors(ca_proton_coords, ca_coords,
na_1_coords, na_2_coords)

        # If the geometry is without error, calculate the angle
        if normal_angle < 5:
            # If the atom has more existing protons, use the proton with the most ideal geometry
            ca_angle_list = [math.fabs(180 - x) for x in ca_angle_list]
            angle = 180 - min(ca_angle_list)
            angle = round(angle, 2)
            return angle
        else:
            print("Residue number " + ca_resi_number + " on chain " + ca_chain_letter + " is non-
planar.")
            return -1

    if not proton_available:
        midpoint_direction = direction(na_2_coords, na_1_coords)
        midpoint_coords = extend(na_2_coords, midpoint_direction, 0.5)

        fictional_proton_direction = direction(midpoint_coords, ca_coords)
        midpoint_ca_distance = distance(midpoint_coords, ca_coords)
        over_all_distance = midpoint_ca_distance + 1
        midpoint_to_proton_factor = find_length(fictional_proton_direction, over_all_distance)
        fictional_proton_coords = list(extend(midpoint_coords, fictional_proton_direction,
midpoint_to_proton_factor))
        fictional_proton_coords = [round(x, 4) for x in fictional_proton_coords]

        angle = get_angle_from_coords(aoi_coords, fictional_proton_coords, ca_coords)
    angle = round(angle, 2)
    return angle


"""
The purpose of getIndex is to receive two arrays and two strings. One array will
contain a list of indices while the other will contain a list of atom names and residue names.
This can be used to find the index of atoms in the -1 and +1 bases as well as the index of
atoms identified by the "select all within 8" statement.
"""


def getIndex(targetStringArray, targetIndexArray, searchName, searchResn,
search_residue_number, search_chain_letter):
    # Give target_index a default value
    target_index = -1
    if "aoi" not in search_residue_number and search_chain_letter != "Z":
        # Find the index of the target atom
        for n in range(len(targetIndexArray)):
            if targetStringArray[n][1] == searchName and targetStringArray[n][2] == searchResn
and \
                            targetStringArray[n][0] == search_residue_number and targetStringArray[n][
                4] == search_chain_letter:
                target_index = targetIndexArray[n]
                break
    elif search_chain_letter == "Z":
        # Find the index of the target atom
        for n in range(len(targetIndexArray)):
            if targetStringArray[n][1] == searchName and targetStringArray[n][2] == searchResn
and \
                            targetStringArray[n][0] == search_residue_number:
                target_index = targetIndexArray[n]
                break
    else:
        # Find the index of the aoi
        for n in range(len(targetIndexArray)):
            if targetStringArray[n][1] == searchName and targetStringArray[n][2] == searchResn:
                target_index = targetIndexArray[n]
                break

    return target_index


def getDist(search_name, searchResn, parentMol):
    # Give AOI_Index a default value
    aoi_index = -1
    # Add protons to all possible atoms so their orientation can be determined by get_angle
    cmd.h_add("all")

    # Find the PDBID of the structure
    cmd.select("PDBID", "index 1")
    stored.pdb = ""
    cmd.iterate("PDBID", "stored.pdb = model")
    pdb = str(stored.pdb)

    # Initialize the array to hold the index, name, resi and b-factor of the target molecules
    stored.target = []
    stored.targetIndex = []
    cmd.iterate(parentMol, "stored.target.append((resi, name, resn, chain, b))")
    cmd.iterate(parentMol, "stored.targetIndex.append((index))")
    parent_string_array = np.array(stored.target)
    parent_index_array = np.array(stored.targetIndex, dtype=np.int32)

    aoi_index = getIndex(parent_string_array, parent_index_array, search_name, searchResn,
"aoi", "a")

    if aoi_index == -1:
        print("ERROR. AOI index could not be found.")
        return -1

    # Checks for potential modified nucleotides that don't match those listed in this plugin
    # accepts water (HOH) as an "accepted nucleotide"
    cmd.select("Target_Contacts", "all within 8 of index " + str(aoi_index))

    stored.nearby_residues = []
    cmd.iterate("Target_Contacts", "stored.nearby_residues.append(resn)")

    nearby_resn_list = np.array(stored.nearby_residues)
    nucleotide_list = ["A", "G", "C", "U", "2AD", "OMC", "OMU", "A2M", "DA", "DC", "CVC",
"HOH", "DU", "MG", "MN", "TB",
                       "NA", "ZN", "AP3", "GX1", "DG", "IRI", "IR", "NCO", "CO", "TL", "BA", "SR",
"AVC", "1DP", "I",
                       "MTU", "N6G", "3DA", "3AD", "P5P"]
    unlisted_nucleotide = []

    for n in range(len(nearby_resn_list)):
        if nearby_resn_list[n] not in nucleotide_list:
            if nearby_resn_list[n] not in unlisted_nucleotide:
                print("NOTE. " + nearby_resn_list[n] + " is not listed as a modified nucleotide in this
plugin.")
                unlisted_nucleotide.append(nearby_resn_list[n])
    # Find contacts within 5 A of aoi_index
    cmd.select("Target_Contacts", "all within 5 of index " + str(aoi_index))

    stored.atom_string = []
    stored.atom_index = []
    cmd.iterate("Target_Contacts", "stored.atom_string.append((resi, name, resn, b, chain))")
    cmd.iterate("Target_Contacts", "stored.atom_index.append((index))")

    contact_string_array = np.array(stored.atom_string)
    contact_index_array = np.array(stored.atom_index, dtype=np.int32)

    # Give the chain a default value if one is not already assigned
    # Deletes metal ions from the list
    metal_remove = False
    metal_index_remove = []
    for n in range(len(contact_string_array)):
        if len(contact_string_array[n][4]) < 1:
            contact_string_array[n][4] = 'Z'
        if contact_string_array[n][1] == "MG" or contact_string_array[n][1] == "MN" or
contact_string_array[n][1] == \
                "TB" or contact_string_array[n][1] == "NA" or contact_string_array[n][1] == "ZN" or
\
                contact_string_array[n][1] == "IR" or contact_string_array[n][1] == "CO" or
contact_string_array[n][1] \
                == "TL" or contact_string_array[n][1] == "BA" or contact_string_array[n][1] == "SR":
            metal_remove = True
            metal_index_remove.append(n)

    if metal_remove:
        contact_string_array = np.delete(contact_string_array, metal_index_remove, 0)
        contact_index_array = np.delete(contact_index_array, metal_index_remove, 0)

    # Find metals within 10 A of aoi_index
    stored.metal_string = []
    stored.metal_index = []
    cmd.iterate("name MG within 10 of index " + str(aoi_index),
"stored.metal_string.append((resi, name, resn, b, "
                                                "chain))")
    cmd.iterate("name MG within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")
    cmd.iterate("name MN within 10 of index " + str(aoi_index),
"stored.metal_string.append((resi, name, resn, b, "
                                                "chain))")
    cmd.iterate("name MN within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")
```

```
    cmd.iterate("name TB within 10 of index " + str(aoi_index), "stored.metal_string.append((resi,
name, resn, b, "
                                                                   "chain))")
    cmd.iterate("name TB within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")
    cmd.iterate("name NA within 10 of index " + str(aoi_index),
"stored.metal_string.append((resi, name, resn, b, "
                                                                   "chain))")
    cmd.iterate("name NA within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")
    cmd.iterate("name ZN within 10 of index " + str(aoi_index), "stored.metal_string.append((resi,
name, resn, b, "
                                                                   "chain))")
    cmd.iterate("name ZN within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")
    cmd.iterate("name IR within 10 of index " + str(aoi_index), "stored.metal_string.append((resi,
name, resn, b, "
                                                                   "chain))")
    cmd.iterate("name IR within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")
    cmd.iterate("name CO within 10 of index " + str(aoi_index),
"stored.metal_string.append((resi, name, resn, b, "
                                                                   "chain))")
    cmd.iterate("name CO within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")
    cmd.iterate("name TL within 10 of index " + str(aoi_index), "stored.metal_string.append((resi,
name, resn, b, "
                                                                   "chain))")
    cmd.iterate("name TL within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")
    cmd.iterate("name BA within 10 of index " + str(aoi_index),
"stored.metal_string.append((resi, name, resn, b, "
                                                                   "chain))")
    cmd.iterate("name BA within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")
    cmd.iterate("name SR within 10 of index " + str(aoi_index), "stored.metal_string.append((resi,
name, resn, b, "
                                                                   "chain))")
    cmd.iterate("name SR within 10 of index " + str(aoi_index),
"stored.metal_index.append((index))")

    metal_string_array = np.array(stored.metal_string)
    metal_index_array = np.array(stored.metal_index, dtype=np.int32)

    # Give the chain a default value if one is not already assigned
    for n in range(len(metal_string_array)):
        if len(metal_string_array[n][4]) < 1:
            metal_string_array[n][4] = 'Z'

    # Open a text file to hold the data
    # File will hold results of calculations
    f = open('C:\\Users\\Drew\\Box Sync\\Drew\\Python\\Plugin Output\\%s.txt'
             % (pdb + '_' + search_name + '_' + searchResn), 'w')

    # Create header
    f.write("***%s Contacts***\n" % (search_name + searchResn))
    f.write("Index" + '\t' + "Resi" + '\t' + "Name" + '\t' + "Distance" + '\t' + "Angle" + '\t'
            + "B-Factor" + '\t' + "Chain" + '\n')

    # Determine the distance between each atom and target
    # Write the distance to the file
    for n in range(0, len(contact_index_array)):
        write = True

        dist = cmd.distance(("index " + str(aoi_index)), ("index " + str(contact_index_array[n])))
        try:
            if "MN" in contact_string_array[n][1]:
                ca_angle = 0
            elif "NCO" in contact_string_array[n][2]:
                ca_angle = 0
            elif "N" in contact_string_array[n][1]:
                ca_angle = getAngle(parent_string_array, parent_index_array, contact_string_array,
contact_index_array,
                                    search_name, searchResn, aoi_index, contact_string_array[n][1],
                                    contact_string_array[n][2], contact_string_array[n][0],
contact_string_array[n][4],
                                    nucleotide_list)
            else:
                ca_angle = 0
        except ValueError:
            print("VALUE ERROR")
            ca_angle = 0

        # Add a unique identifier for the AOI, since the contact_string_array has a fixed length,
        # the atom name is saved to a different variable before the atom name is replaced by "AOI"
        if searchResn == contact_string_array[n][2] and search_name == contact_string_array[n][
            1] and dist < 1:
            AOI = contact_string_array[n][1]
            contact_string_array[n][1] = "AOI_"

            f.write(
                str(contact_index_array[n]) + '\t' + contact_string_array[n][2] +
contact_string_array[n][0] + '\t' +
                contact_string_array[n][1] + AOI + '\t' + "0.00000000000" + '\t' + str(ca_angle) + '\t' +
                contact_string_array[n][
                    3] + '\t' + contact_string_array[n][4] + '\n')
            write = False

        if write:
            f.write(
                str(contact_index_array[n]) + '\t' + contact_string_array[n][2] +
contact_string_array[n][0] + '\t' +
                contact_string_array[n][1] + '\t' + str(dist) + '\t' + str(ca_angle) + '\t' +
contact_string_array[n][
                    3] + '\t' + contact_string_array[n][4] + '\n')

    # Determine the distance between each metal and target
    # Write the distance to the file
    for n in range(0, len(metal_index_array)):

        dist = cmd.distance(("index " + str(aoi_index)), ("index " + str(metal_index_array[n])))
        ca_angle = 0

        f.write(
            str(metal_index_array[n]) + '\t' + metal_string_array[n][2] + metal_string_array[n][0] +
'\t' +
            metal_string_array[n][1] + '\t' + str(dist) + '\t' + str(ca_angle) + '\t' +
metal_string_array[n][3] + '\t'
            + metal_string_array[n][4] + '\n')

    f.close()
```

II. α Plugin

```
from Tkinter import *
from pymol import cmd
from pymol import stored


def __init__(self):
    self.menuBar.addmenuitem("Plugin", "command",
                             "Alpha Data Collection",
                             label="Alpha Data Collection",
                             command=lambda s=self: number_of_chains(s))

# This is where all the text files will be dumped
directory = 'C:\\some directory'


# removes atoms of conformation B, if present
def remove_additional_conformations():

    stored.alt_b = []
    cmd.iterate("alt B", "stored.alt_b.append(index)")
    alt_b = stored.alt_b
    if len(alt_b) > 0:
        cmd.remove("alt B")
        print("Atoms included in alternative conformation B were removed.")

    stored.alt_all = []
    cmd.iterate("all", "stored.alt_all.append(alt)")
    alt_all = stored.alt_all

    alt_all_filtered = []
    for a in range(len(alt_all)):
        if len(alt_all[a]) != 0:
            alt_all_filtered.append(alt_all[a])

    additional_conformations = False
    if len(alt_all_filtered) > 0:
        for a in range(len(alt_all_filtered)):
            if alt_all_filtered[a] != 'A':
                additional_conformations = True
    if additional_conformations:
        print("ERROR: Additional conformations are present.")


def unique_residue_numbers(cofactor, minus_one_nucleotide):
```

```
# Retrieve the pdb of the crystal structure
cmd.select("index_1", "index 1")
stored.pdb = ""
cmd.iterate("index_1", "stored.pdb = model")
pdb = stored.pdb

if cofactor != "none":
    cmd.remove("resn " + cofactor)

# Create an array that contains the index and residue number of all 2' oxygens
cmd.select("two_prime_oxygen", "name O2'")
stored.residue_identifier_O2 = []
stored.two_prime_oxygen_chain = []
stored.two_prime_oxygen_index = []
stored.two_prime_oxygen_b = []
cmd.iterate("two_prime_oxygen", "stored.residue_identifier_O2.append(resi)")
cmd.iterate("two_prime_oxygen", "stored.two_prime_oxygen_chain.append(chain)")
cmd.iterate("two_prime_oxygen", "stored.two_prime_oxygen_index.append(index)")
cmd.iterate("two_prime_oxygen", "stored.two_prime_oxygen_b.append(b)")
residue_identifier_O2 = stored.residue_identifier_O2
two_prime_oxygen_chain = stored.two_prime_oxygen_chain
two_prime_oxygen_index = stored.two_prime_oxygen_index
two_prime_oxygen_b = stored.two_prime_oxygen_b


# Create an array that contains the index and residue number of all phosphorus
cmd.select("phosphorus", "name P")
stored.residue_identifier_P = []
stored.phosphorus_chain = []
stored.phosphorus_index = []
stored.phosphorus_b = []
cmd.iterate("phosphorus", "stored.residue_identifier_P.append(resi)")
cmd.iterate("phosphorus", "stored.phosphorus_chain.append(chain)")
cmd.iterate("phosphorus", "stored.phosphorus_index.append(index)")
cmd.iterate("phosphorus", "stored.phosphorus_b.append(b)")
residue_identifier_P = stored.residue_identifier_P
phosphorus_chain = stored.phosphorus_chain
phosphorus_index = stored.phosphorus_index
phosphorus_b = stored.phosphorus_b

# Create an array that contains the index and residue number of all 5' oxygens
cmd.select("five_prime_oxygen", "name O5'")
stored.residue_identifier_O5 = []
stored.five_prime_oxygen_index = []
stored.five_prime_oxygen_b = []
```

```
    cmd.iterate("five_prime_oxygen", "stored.residue_identifier_O5.append(resi)")
    cmd.iterate("five_prime_oxygen", "stored.five_prime_oxygen_index.append(index)")
    cmd.iterate("five_prime_oxygen", "stored.five_prime_oxygen_b.append(b)")
    residue_identifier_O5 = stored.residue_identifier_O5
    five_prime_oxygen_index = stored.five_prime_oxygen_index
    five_prime_oxygen_b = stored.five_prime_oxygen_b

    # Adjusts the arrays if the 2' oxygen is not on the residue in front of the residue containing the
phosphorus at the
    # beginning of the arrays
    count = 0
    while str(residue_identifier_O2[0]) >= str(residue_identifier_P[0]):
        residue_identifier_P.remove(residue_identifier_P[0])
        phosphorus_chain.remove(phosphorus_chain[0])
        phosphorus_index.remove(phosphorus_index[0])
        phosphorus_b.remove(phosphorus_b[0])
        count += 1
        if count >= 11:
            print ("Something went wrong. Line 107")
            break

    # Adjusts the arrays if the 2' oxygen is not on the residue in front of the residue containing the
5' oxygen at the
    # beginning of the arrays
    count = 0
    while str(residue_identifier_O2[0]) >= str(residue_identifier_O5[0]):
        residue_identifier_O5.remove(residue_identifier_O5[0])
        five_prime_oxygen_index.remove(five_prime_oxygen_index[0])
        five_prime_oxygen_b.remove(five_prime_oxygen_b[0])
        count += 1
        if count >= 11:
            print ("Something went wrong. Line 119")
            break

    # Adjusts the arrays such that the last O2' resides one residue before the last phosphorus
    count = 0
    while residue_identifier_O2[-1] >= residue_identifier_P[-1]:
        residue_identifier_O2.remove(residue_identifier_O2[-1])
        two_prime_oxygen_index.remove(two_prime_oxygen_index[-1])
        two_prime_oxygen_chain.remove(two_prime_oxygen_chain[-1])
        two_prime_oxygen_b.remove(two_prime_oxygen_b[-1])
        count += 1
        if count >= 11:
            print ("Something went wrong. Line 131")
            break
    # Adjust the arrays when there is a break in the RNA (for 2 piece RNAs) or when the O2' has
been mutated
    # to something else
    range_of_minimum_length = range(
        min([len(residue_identifier_O2), len(residue_identifier_P), len(residue_identifier_O5)]))
    for d in range_of_minimum_length:
        count = 0
        while ((int(residue_identifier_O2[d]) + 1) != int(residue_identifier_P[d])) or \
                ((int(residue_identifier_O2[d]) + 1) != int(residue_identifier_O5[d])):
            if ((int(residue_identifier_O2[d]) + 1) > (int(residue_identifier_P[d]))) and \
                    ((int(residue_identifier_O2[d]) + 1) > int(residue_identifier_O5[d])):
                residue_identifier_P.remove(residue_identifier_P[d])
                phosphorus_chain.remove(phosphorus_chain[d])
                phosphorus_index.remove(phosphorus_index[d])
                phosphorus_b.remove(phosphorus_b[d])
                residue_identifier_O5.remove(residue_identifier_O5[d])
                five_prime_oxygen_index.remove(five_prime_oxygen_index[d])
                five_prime_oxygen_b.remove(five_prime_oxygen_b[d])
                if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < len(
                        range_of_minimum_length):
                    range_of_minimum_length.remove(range_of_minimum_length[-1])
                count += 1
                if count >= 100:
                    print("Something went wrong. Line 156")
                    break
            elif ((int(residue_identifier_O2[d]) + 1) > (int(residue_identifier_P[d]))) and \
                    ((int(residue_identifier_O2[d]) + 1) == int(residue_identifier_O5[d])):
                residue_identifier_P.remove(residue_identifier_P[d])
                phosphorus_chain.remove(phosphorus_chain[d])
                phosphorus_index.remove(phosphorus_index[d])
                phosphorus_b.remove(phosphorus_b[d])
                if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < len(
                        range_of_minimum_length):
                    range_of_minimum_length.remove(range_of_minimum_length[-1])
                count += 1
                if count >= 100:
                    print("Something went wrong. Line 169")
                    break
            elif ((int(residue_identifier_O2[d]) + 1) == (int(residue_identifier_P[d]))) and \
                    ((int(residue_identifier_O2[d]) + 1) > int(residue_identifier_O5[d])):
                residue_identifier_O5.remove(residue_identifier_O5[d])
                five_prime_oxygen_index.remove(five_prime_oxygen_index[d])
                five_prime_oxygen_b.remove(five_prime_oxygen_b[d])
                if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < len(
                        range_of_minimum_length):
                    range_of_minimum_length.remove(range_of_minimum_length[-1])
                count += 1
                if count >= 100:
                    print("Something went wrong. Line 181")
                    break
            elif ((int(residue_identifier_O2[d]) + 1) < (int(residue_identifier_P[d]))) and \
                    ((int(residue_identifier_O2[d]) + 1) < int(residue_identifier_O5[d])):
                residue_identifier_O2.remove(residue_identifier_O2[d])
                two_prime_oxygen_index.remove(two_prime_oxygen_index[d])
                two_prime_oxygen_chain.remove(two_prime_oxygen_chain[d])
                if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < len(
                        range_of_minimum_length):
                    range_of_minimum_length.remove(range_of_minimum_length[-1])
                count += 1
                if count >= 100:
                    print("Something went wrong. Line 193")
                    break
            elif ((int(residue_identifier_O2[d]) + 1) < int(residue_identifier_P[d])) and \
                    ((int(residue_identifier_O2[d]) + 1) == int(residue_identifier_O5[d])):
                residue_identifier_O2.remove(residue_identifier_O2[d])
                two_prime_oxygen_index.remove(two_prime_oxygen_index[d])
                two_prime_oxygen_chain.remove(two_prime_oxygen_chain[d])
                residue_identifier_O5.remove(residue_identifier_O5[d])
                five_prime_oxygen_index.remove(five_prime_oxygen_index[d])
                five_prime_oxygen_b.remove(five_prime_oxygen_b[d])
                if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < len(
                        range_of_minimum_length):
                    range_of_minimum_length.remove(range_of_minimum_length[-1])
                count += 1
                if count >= 100:
                    print("Something went wrong. Line 208")
                    break
            elif ((int(residue_identifier_O2[d]) + 1) == int(residue_identifier_P[d])) and \
                    ((int(residue_identifier_O2[d]) + 1) < int(residue_identifier_O5[d])):
                residue_identifier_O2.remove(residue_identifier_O2[d])
                two_prime_oxygen_index.remove(two_prime_oxygen_index[d])
                two_prime_oxygen_chain.remove(two_prime_oxygen_chain[d])
                residue_identifier_P.remove(residue_identifier_P[d])
                phosphorus_chain.remove(phosphorus_chain[d])
                phosphorus_index.remove(phosphorus_index[d])
                phosphorus_b.remove(phosphorus_b[d])
                if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < len(
                        range_of_minimum_length):
                    range_of_minimum_length.remove(range_of_minimum_length[-1])
                count += 1
                if count >= 100:
                    print("Something went wrong. Line 224")
                    break
            else:
                print("Something went wrong. Line 227")
                break

    # Ensures that each residue stored in the program that contains a phosphorus also contains a
O5'
    if residue_identifier_P != residue_identifier_O5:
        print ("The array containing residues with phosphorus do not match the array containing
residues with O5'")
        return

    # compare letters of chains for P and O2' to check if they are in the same chain.
    # If they are not, the angle won't be calculated
    for d in range_of_minimum_length:
        count = 0
        while (str(two_prime_oxygen_chain[d])) != str(phosphorus_chain[d]):
            print("An angle was not determined between resi " + str(
                residue_identifier_O2[d]) + " and resi " +
                  str(residue_identifier_P[d]))
            phosphorus_chain.remove(phosphorus_chain[d])
            residue_identifier_P.remove(residue_identifier_P[d])
            phosphorus_index.remove(phosphorus_index[d])
            phosphorus_b.remove(phosphorus_b[d])
            residue_identifier_O5.remove(residue_identifier_O5[d])
            five_prime_oxygen_index.remove(five_prime_oxygen_index[d])
            five_prime_oxygen_b.remove(five_prime_oxygen_b[d])
            residue_identifier_O2.remove(residue_identifier_O2[d])
            two_prime_oxygen_index.remove(two_prime_oxygen_index[d])
            two_prime_oxygen_chain.remove(two_prime_oxygen_chain[d])
            range_of_minimum_length.remove(range_of_minimum_length[-1])
            count += 1
            if count >= 100:
                print("Something went wrong. Line 256")
                break

    # Finds the angles and writes them to a text file
    with open(directory + "\\%s Non-Scissile Alpha.txt" % pdb, "w") as f_non_scissile:
        f_non_scissile.write("*** %s Non-Scissile Alpha ***" % pdb + "\n" + "chain" + '\t' + "resi"
+ "\t" + "angle" +
                             "\n")
        for i in range(len(residue_identifier_O2)):
            angle = cmd.angle("Residue: " + residue_identifier_O2[i],
                              "index " + str(two_prime_oxygen_index[i]),
                              "index " + str(phosphorus_index[i]),
                              "index " + str(five_prime_oxygen_index[i]))

            # Give a default chain letter of Z if no chain letter was previously assigned
            if len(two_prime_oxygen_chain[i]) < 1:
                two_prime_oxygen_chain[i] = 'Z'

            # Writes the angle about the scissile phosphate to a separate text file
            if residue_identifier_O2[i] == minus_one_nucleotide:
                with open(directory + "\\%s Scissile Alpha.txt" % pdb,
                          "w") as f_scissile:
                    f_scissile.write(
                        "*** %s Scissile Alpha ***" % pdb + "\n" + "chain" + '\t' + "resi" + "\t" + "angle"
+ "\n")
                    f_scissile.write(
                        two_prime_oxygen_chain[i] + "\t" + residue_identifier_O2[i] + "\t" + str(angle) +
"\n")
            else:
                f_non_scissile.write(two_prime_oxygen_chain[i] + "\t" + residue_identifier_O2[i] +
"\t" + str(angle) +
                                     "\n")


def repetitive_residue_numbers(cofactor, minus_one_nucleotide, chain_letters):

    # Retrieve the pdb of the crystal structure
    cmd.select("index_1", "index 1")
    stored.pdb = ""
    cmd.iterate("index_1", "stored.pdb = model")
    pdb = stored.pdb

    # Removes all atoms within the object specified by the user
    if cofactor != "none":
        cmd.remove("resn " + cofactor)

    chain_letters = chain_letters.upper().split(' ')

    f_non_scissile = open(directory + "\\%s Non-Scissile Alpha.txt" % pdb, "w")
    f_non_scissile.write("*** %s Non-Scissile Alpha ***" % pdb + "\n" + "chain" + "\t" + "resi"
+ "\t" + "angle" + "\n")
```

```
    # Iterates through the chains specified by the user
    for z in range(len(chain_letters)):

        # Continues to the next iteration if the current chain contains no atoms
        if (cmd.count_atoms("chain %s" % chain_letters[z])) <= 0:
            continue

        # Any chains that contain amino acids are excluded
        stored.amino_acid_test = []
        cmd.iterate("chain " + chain_letters[z], "stored.amino_acid_test.append(resn)")
        amino_acid_test = stored.amino_acid_test
        if ("ARG" or "HIS" or "LYS" or "ASP" or "GlU" or "SER" or "THR" or "ASN" or "GLN"
or "CYS" or "SEC" or "GLY"
                or "PRO" or "ALA" or "VAL" or "ILE" or "LEU" or "MET" or "PHE" or "TYR" or
"TRP") in \
                amino_acid_test:
            continue

        else:
            # Create an array that contains the index and residue number of all 2' oxygens
            cmd.select("two_prime_oxygen", "name O2' and chain " + chain_letters[z])
            stored.residue_identifier_O2 = []
            stored.two_prime_oxygen_index = []
            stored.two_prime_oxygen_chain = []
            stored.two_prime_oxygen_b = []
            cmd.iterate("two_prime_oxygen", "stored.residue_identifier_O2.append(resi)")
            cmd.iterate("two_prime_oxygen", "stored.two_prime_oxygen_index.append(index)")
            cmd.iterate("two_prime_oxygen", "stored.two_prime_oxygen_chain.append(chain)")
            cmd.iterate("two_prime_oxygen", "stored.two_prime_oxygen_b.append(b)")
            residue_identifier_O2 = stored.residue_identifier_O2
            two_prime_oxygen_chain = stored.two_prime_oxygen_chain
            two_prime_oxygen_index = stored.two_prime_oxygen_index
            two_prime_oxygen_b = stored.two_prime_oxygen_b

            # Create an array that contains the index and residue number of all phosphorus
            cmd.select("phosphorus", "name P and chain " + chain_letters[z])
            stored.residue_identifier_P = []
            stored.phosphorus_index = []
            stored.phosphorus_chain = []
            stored.phosphorus_b = []
            cmd.iterate("phosphorus", "stored.residue_identifier_P.append(resi)")
            cmd.iterate("phosphorus", "stored.phosphorus_index.append(index)")
            cmd.iterate("phosphorus", "stored.phosphorus_chain.append(chain)")
            cmd.iterate("phosphorus", "stored.phosphorus_b.append(b)")
            residue_identifier_P = stored.residue_identifier_P
            phosphorus_chain = stored.phosphorus_chain
            phosphorus_index = stored.phosphorus_index
            phosphorus_b = stored.phosphorus_b

            # Create an array that contains the index and residue number of all 5' oxygens
            cmd.select("five_prime_oxygen", "name O5' and chain " + chain_letters[z])
            stored.residue_identifier_O5 = []
            stored.five_prime_oxygen_index = []
            stored.five_prime_oxygen_b = []
            cmd.iterate("five_prime_oxygen", "stored.residue_identifier_O5.append(resi)")
            cmd.iterate("five_prime_oxygen", "stored.five_prime_oxygen_index.append(index)")
            cmd.iterate("five_prime_oxygen", "stored.five_prime_oxygen_b.append(b)")
            residue_identifier_O5 = stored.residue_identifier_O5
            five_prime_oxygen_index = stored.five_prime_oxygen_index
            five_prime_oxygen_b = stored.five_prime_oxygen_b

            # Adjusts the arrays if the 2' oxygen is not on the residue in front of the residue
containing the
            # phosphorus at the beginning of the arrays
            count = 0
            while str(residue_identifier_O2[0]) >= str(residue_identifier_P[0]):
                residue_identifier_P.remove(residue_identifier_P[0])
                phosphorus_chain.remove(phosphorus_chain[0])
                phosphorus_index.remove(phosphorus_index[0])
                phosphorus_b.remove(phosphorus_b[0])
                count += 1
                if count >= 11:
                    print ("Something went wrong. Line 372")
                    break

            # Adjusts the arrays if the 2' oxygen is not on the residue in front of the residue
containing the 5' oxygen
            # at the beginning of the arrays
            count = 0
            while str(residue_identifier_O2[0]) >= str(residue_identifier_O5[0]):
                residue_identifier_O5.remove(residue_identifier_O5[0])
                five_prime_oxygen_index.remove(five_prime_oxygen_index[0])
                five_prime_oxygen_b.remove(five_prime_oxygen_b[0])
                count += 1
                if count >= 11:
                    print ("Something went wrong. Line 384")
                    break

            # Adjusts the arrays such that the last O2' resides one residue before the last phosphorus
            count = 0
            while residue_identifier_O2[-1] >= residue_identifier_P[-1]:
                residue_identifier_O2.remove(residue_identifier_O2[-1])
                two_prime_oxygen_index.remove(two_prime_oxygen_index[-1])
                two_prime_oxygen_chain.remove(two_prime_oxygen_chain[-1])
                two_prime_oxygen_b.remove(two_prime_oxygen_b[-1])
                count += 1
                if count >= 11:
                    print ("Something went wrong. Line 396")
                    break

            # Adjust the arrays when there is a break in the RNA (for 2 piece RNAs) or when the O2'
has been mutated
            # to something else
            range_of_minimum_length = range(min([len(residue_identifier_O2),
len(residue_identifier_P),
                                             len(residue_identifier_O5)]))
            for d in range_of_minimum_length:
                count = 0
                while ((int(residue_identifier_O2[d]) + 1) != int(residue_identifier_P[d])) or \
                        ((int(residue_identifier_O2[d]) + 1) != int(residue_identifier_O5[d])):
                    if ((int(residue_identifier_O2[d]) + 1) > (int(residue_identifier_P[d]))) and \
                            ((int(residue_identifier_O2[d]) + 1) > int(residue_identifier_O5[d])):
                        residue_identifier_P.remove(residue_identifier_P[d])
                        phosphorus_chain.remove(phosphorus_chain[d])
                        phosphorus_index.remove(phosphorus_index[d])
                        phosphorus_b.remove(phosphorus_b[d])
                        residue_identifier_O5.remove(residue_identifier_O5[d])
                        five_prime_oxygen_index.remove(five_prime_oxygen_index[d])
                        five_prime_oxygen_b.remove(five_prime_oxygen_b[d])
                        if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < \
                            len(range_of_minimum_length):
                            range_of_minimum_length.remove(range_of_minimum_length[-1])
                        count += 1
                        if count >= 100:
                            print("Something went wrong. Line 421")
                            break
                    elif ((int(residue_identifier_O2[d]) + 1) > (int(residue_identifier_P[d]))) and \
                            ((int(residue_identifier_O2[d]) + 1) == int(residue_identifier_O5[d])):
                        residue_identifier_P.remove(residue_identifier_P[d])
                        phosphorus_chain.remove(phosphorus_chain[d])
                        phosphorus_index.remove(phosphorus_index[d])
                        phosphorus_b.remove(phosphorus_b[d])
                        if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < \
                            len(range_of_minimum_length):
                            range_of_minimum_length.remove(range_of_minimum_length[-1])
                        count += 1
                        if count >= 100:
                            print("Something went wrong. Line 434")
                            break
                    elif ((int(residue_identifier_O2[d]) + 1) == (int(residue_identifier_P[d]))) and \
                            ((int(residue_identifier_O2[d]) + 1) > int(residue_identifier_O5[d])):
                        residue_identifier_O5.remove(residue_identifier_O5[d])
                        five_prime_oxygen_index.remove(five_prime_oxygen_index[d])
                        five_prime_oxygen_b.remove(five_prime_oxygen_b[d])
                        if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < \
                            len(range_of_minimum_length):
                            range_of_minimum_length.remove(range_of_minimum_length[-1])
                        count += 1
                        if count >= 100:
                            print("Something went wrong. Line 446")
                            break
                    elif ((int(residue_identifier_O2[d]) + 1) < (int(residue_identifier_P[d]))) and \
                            ((int(residue_identifier_O2[d]) + 1) < int(residue_identifier_O5[d])):
                        residue_identifier_O2.remove(residue_identifier_O2[d])
                        two_prime_oxygen_index.remove(two_prime_oxygen_index[d])
                        two_prime_oxygen_chain.remove(two_prime_oxygen_chain[d])
                        if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < \
                            len(range_of_minimum_length):
                            range_of_minimum_length.remove(range_of_minimum_length[-1])
                        count += 1
                        if count >= 100:
                            print("Something went wrong. Line 458")
                            break
                    elif ((int(residue_identifier_O2[d]) + 1) < int(residue_identifier_P[d])) and \
                            ((int(residue_identifier_O2[d]) + 1) == int(residue_identifier_O5[d])):
                        residue_identifier_O2.remove(residue_identifier_O2[d])
                        two_prime_oxygen_index.remove(two_prime_oxygen_index[d])
                        two_prime_oxygen_chain.remove(two_prime_oxygen_chain[d])
                        residue_identifier_O5.remove(residue_identifier_O5[d])
                        five_prime_oxygen_index.remove(five_prime_oxygen_index[d])
                        five_prime_oxygen_b.remove(five_prime_oxygen_b[d])
                        if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < \
                            len(range_of_minimum_length):
                            range_of_minimum_length.remove(range_of_minimum_length[-1])
                        count += 1
                        if count >= 100:
                            print("Something went wrong. Line 473")
                            break
```

```
          elif ((int(residue_identifier_O2[d]) + 1) == int(residue_identifier_P[d])) and \
                  ((int(residue_identifier_O2[d]) + 1) < int(residue_identifier_O5[d])):
              residue_identifier_O2.remove(residue_identifier_O2[d])
              two_prime_oxygen_index.remove(two_prime_oxygen_index[d])
              two_prime_oxygen_chain.remove(two_prime_oxygen_chain[d])
              residue_identifier_P.remove(residue_identifier_P[d])
              phosphorus_chain.remove(phosphorus_chain[d])
              phosphorus_index.remove(phosphorus_index[d])
              phosphorus_b.remove(phosphorus_b[d])
              if min([len(residue_identifier_O2), len(residue_identifier_P),
len(residue_identifier_O5)]) < \
                      len(range_of_minimum_length):
                  range_of_minimum_length.remove(range_of_minimum_length[-1])
              count += 1
              if count >= 100:
                  print("Something went wrong. Line 489")
                  break
          else:
              print("Something went wrong. Line 492")
              break

      # Ensures that each residue stored in the program that contains a phosphorus also
contains a O5'
      if residue_identifier_P != residue_identifier_O5:
          print ("The array containing residues with phosphorus do not match the array
containing residues with "
                 "O5'")
          break

      # compare letters of chains for P and O2' to check if they are in the same chain.
      # If they are not, the angle won't be calculated
      for d in range_of_minimum_length:
          count = 0
          while (str(two_prime_oxygen_chain[d])) != str(phosphorus_chain[d]):
              print("An angle was not determined between resi " + str(
                  residue_identifier_O2[d]) + " and resi " +
                    str(residue_identifier_P[d]))
              phosphorus_chain.remove(phosphorus_chain[d])
              residue_identifier_P.remove(residue_identifier_P[d])
              phosphorus_index.remove(phosphorus_index[d])
              phosphorus_b.remove(phosphorus_b[d])
              residue_identifier_O5.remove(residue_identifier_O5[d])
              five_prime_oxygen_index.remove(five_prime_oxygen_index[d])
              five_prime_oxygen_b.remove(five_prime_oxygen_b[d])
              residue_identifier_O2.remove(residue_identifier_O2[d])
              two_prime_oxygen_index.remove(two_prime_oxygen_index[d])
```

```
                two_prime_oxygen_chain.remove(two_prime_oxygen_chain[d])
                range_of_minimum_length.remove(range_of_minimum_length[-1])
                count += 1
                if count >= 100:
                    print("Something went wrong. Line 522")
                    break

        # Finds the angles and writes them to a text file
        for i in range(len(residue_identifier_O2)):
            angle = cmd.angle("Residue: " + residue_identifier_O2[i],
                              "index " + str(two_prime_oxygen_index[i]),
                              "index " + str(phosphorus_index[i]),
                              "index " + str(five_prime_oxygen_index[i]))

            # Give a default chain letter of Z if no chain letter was previously assigned
            if len(two_prime_oxygen_chain[i]) < 1:
                two_prime_oxygen_chain[i] = 'Z'

            # Writes the angle about the scissile phosphate to a separate text file
            if residue_identifier_O2[i] == minus_one_nucleotide and two_prime_oxygen_chain[i] ==
chain_letters[0]:
                with open(directory + "\\%s Scissile Alpha.txt" % pdb,
                          "w") as f_scissile:
                    f_scissile.write(
                        "*** %s Scissile Alpha ***" % pdb + "\n" + "chain" + '\t' + "resi" + "\t" + "angle"
+ "\n")
                    f_scissile.write(
                        two_prime_oxygen_chain[i] + "\t" + residue_identifier_O2[i] + "\t" + str(angle) +
"\n")
            else:
                f_non_scissile.write(two_prime_oxygen_chain[i] + "\t" + residue_identifier_O2[i] +
"\t" + str(angle) +
                                     "\n")

    f_non_scissile.close()


def get_chain_letters(cofactor, minus_one_nucleotide):

    # stores a list of chain letters entered by the user to collect data on
    # the letters should be separated by a space and the first letter is assumed to contain the scissile
phosphate
    def run_repetitive_residue_numbers():

        chain_letters = e1.get()
        root.destroy()
        repetitive_residue_numbers(cofactor, minus_one_nucleotide, chain_letters)

    root = Tk()

    l1 = Label(root, text="Chains: ")
    l1.grid(row=1, column=0)

    e1 = Entry(root)
    e1.grid(row=1, column=1)
    e1.delete(0, END)
    e1.insert(0, "A")

    B1 = Button(root, text="Enter", command=run_repetitive_residue_numbers)
    B1.grid(columnspan=2)

    def key(event):
        run_repetitive_residue_numbers()

    root.bind("<Return>", key)
    root.focus_set()

    root.mainloop()

    cmd.extend('number_of_chains', number_of_chains)


def number_of_chains(app):

    # stores -1 residue number of scissile phosphate and cofactor name (if present) entered by the 
user
    def get_angle():
        cofactor = e1.get()
        minus_one_nucleotide = e2.get()
        root.destroy()
        remove_additional_conformations()

        # determines whether there are repeating residue numbers in the structure
        stored.residue_identifier_O2 = []
        cmd.iterate("name O2'", "stored.residue_identifier_O2.append(resi)")
        residue_identifier_O2 = stored.residue_identifier_O2
        number = []
        for a in range(len(residue_identifier_O2)):
            number.append(residue_identifier_O2.count(str(residue_identifier_O2[a])))
        repetitive = False
        for a in range(len(number)):
            if number[a] > 1:
```

```
            repetitive = True

    # two separate routes are taken depending on whether the nucleotide numbering is repetitive
    if repetitive:
        get_chain_letters(cofactor, minus_one_nucleotide)
    else:
        unique_residue_numbers(cofactor, minus_one_nucleotide)

root = Tk()

l1 = Label(root, text="Co-factor name: ")
l1.grid(row=1, column=0)

e1 = Entry(root)
e1.grid(row=1, column=1)
e1.delete(0, END)
e1.insert(0, "none")

l2 = Label(root, text="-1 Nucleotide: ")
l2.grid(row=2, column=0)

e2 = Entry(root)
e2.grid(row=2, column=1)
e2.delete(0, END)
e2.insert(0, "0")

B1 = Button(root, text="Enter", command=get_angle)
B1.grid(columnspan=2)

def key(event):
    get_angle()

root.bind("<Return>", key)
root.focus_set()

root.mainloop()

cmd.extend('number_of_chains', number_of_chains)
```

III. γ, β, and δ All-Phosphates Plugin

```
from Tkinter import *
import numpy as np
from pymol import cmd, stored



def __init__(self):
    self.menuBar.addmenuitem('Plugin', 'command',
                             'All Phosphate Contacts',
                             label='All Phosphate Contacts',
                             command=lambda s=self: get_info(s))

# This is where all the text files will be dumped
directory = 'C:\\some directory'



# Pops up a window which accepts input from the user
def get_info(app):
    root = Tk()

    remove_additional_conformations()

    # Runs when the O2' button is pressed
    def get_O2_data():
        minus_one_nucleotide = E1.get()
        chain_array = E3.get().upper().split(' ')
        for a in range(len(chain_array)):
            if chain_array[a] == 'NONE':
                continue
            elif len(chain_array[a]) != 1:
                print("ERROR: Incorrect entry.")
                return

        # Checks that the user didn't enter more than 5 chain letters
        if len(chain_array) > 5:
            print("ERROR: This plugin cannot handle more than 5 chains.")
            return

        # Checks whether or not an amino is substituted in for the 2'-OH according to the user input
        amino_substitution = False
        if (E4.get() == 'y') or (E4.get() == 'Y'):
            amino_substitution = True
        elif (E4.get() == 'n') or (E4.get() == 'N'):
            amino_substitution = False
        else:
```

```
        print("ERROR: Incorrect entry.")
        return
    root.destroy()
    find_O2_CA(minus_one_nucleotide, chain_array, amino_substitution)

# Runs when the NBO button is pressed
def get_NBO_data():
    plus_one_nucleotide = E2.get()
    chain_array = E3.get().upper().split(' ')
    for a in range(len(chain_array)):
        if chain_array[a] == 'NONE':
            continue
        elif len(chain_array[a]) != 1:
            print("ERROR: Incorrect entry.")
            return

    # Checks that the user didn't enter more than 5 chain letters
    if len(chain_array) > 5:
        print("ERROR: This plugin cannot handle more than 5 chains.")
        return
    root.destroy()
    find_NBO_CA(plus_one_nucleotide, chain_array)

# Runs when the O5' button is pressed
def get_O5_data():
    plus_one_nucleotide = E2.get()
    chain_array = E3.get().upper().split(' ')
    for a in range(len(chain_array)):
        if chain_array[a] == 'NONE':
            continue
        elif len(chain_array[a]) != 1:
            print("ERROR: Incorrect entry.")
            return

    # Checks that the user didn't enter more than 5 chain letters
    if len(chain_array) > 5:
        print("ERROR: This plugin cannot handle more than 5 chains.")
        return
    root.destroy()
    find_O5_CA(plus_one_nucleotide, chain_array)

# Runs when the All button is pressed
def get_all_data():
    minus_one_nucleotide = E1.get()
    plus_one_nucleotide = E2.get()
    chain_array = E3.get().upper().split(' ')
    for a in range(len(chain_array)):
        if chain_array[a] == 'NONE':
            continue
        elif len(chain_array[a]) != 1:
            print("ERROR: Incorrect entry.")
            return

    # Checks that the user didn't enter more than 5 chain letters
    if len(chain_array) > 5:
        print("ERROR: This plugin cannot handle more than 5 chains.")
        return

    # Checks whether or not an amino is substituted in for the 2'-OH according to the user input
    amino_substitution = False
    if (E4.get() == 'y') or (E4.get() == 'Y'):
        amino_substitution = True
    elif (E4.get() == 'n') or (E4.get() == 'N'):
        amino_substitution = False
    else:
        print("ERROR: Incorrect entry.")
        return
    root.destroy()
    find_O2_CA(minus_one_nucleotide, chain_array, amino_substitution)
    find_NBO_CA(plus_one_nucleotide, chain_array)
    find_O5_CA(plus_one_nucleotide, chain_array)

L1 = Label(root, text="Scissile Phosphate -1 Nucleotide:")
L1.grid(row=0, column=0, columnspan=3)

E1 = Entry(root, width=6, justify=CENTER)
E1.grid(row=0, column=3)
E1.delete(0, END)
E1.insert(0, '4')

L2 = Label(root, text="Scissile Phosphate +1 Nucleotide:")
L2.grid(row=1, column=0, columnspan=3)

E2 = Entry(root, width=6, justify=CENTER)
E2.grid(row=1, column=3)
E2.delete(0, END)
E2.insert(0, '5')

L3 = Label(root, text="Chains of Interest:")
L3.grid(row=2, column=0, columnspan=3)

E3 = Entry(root, width=6, justify=CENTER)
```

```
    E3.grid(row=2, column=3)
    E3.delete(0, END)
    E3.insert(0, 'A')

    L4 = Label(root, text="N2' at -1 Nucleotide? (y/n):")
    L4.grid(row=3, column=0, columnspan=3)

    E4= Entry(root, width=6, justify=CENTER)
    E4.grid(row=3, column=3)
    E4.delete(0, END)
    E4.insert(0, "n")

    B1 = Button(root, text="O2'", width=8, command=get_O2_data)
    B1.grid(row=4, column=0)

    B2 = Button(root, text="NBO", width=8, command=get_NBO_data)
    B2.grid(row=4, column=1)

    B3 = Button(root, text="O5'", width=8, command=get_O5_data)
    B3.grid(row=4, column=2)

    B4 = Button(root, text="All", width=8, command=get_all_data)
    B4.grid(row=4, column=3)

    def key(event):
        get_all_data()

    root.bind("<Return>", key)
    root.focus_set()

    mainloop()

cmd.extend('get_info', get_info)


# removes atoms of conformation B, if present
def remove_additional_conformations():

    stored.alt_b = []
    cmd.iterate("alt B", "stored.alt_b.append(index)")
    alt_b = stored.alt_b
    if len(alt_b) > 0:
        cmd.remove("alt B")
        print("Atoms included in alternative conformation B were removed.")

    stored.alt_all = []
    cmd.iterate("all", "stored.alt_all.append(alt)")
    alt_all = stored.alt_all

    alt_all_filtered = []
    for a in range(len(alt_all)):
        if len(alt_all[a]) != 0:
            alt_all_filtered.append(alt_all[a])

    additional_conformations = False
    if len(alt_all_filtered) > 0:
        for a in range(len(alt_all_filtered)):
            if alt_all_filtered[a] != 'A':
                additional_conformations = True
    if additional_conformations:
        print("ERROR: Additional conformations are present.")


def find_O2_CA(minus_one_nucleotide, chain_array, amino_substitution):

    stored.pdb = ""
    cmd.iterate("index 1", "stored.pdb = model")
    pdb = stored.pdb

    # Finds and stores information on the N2', if present and specified by the user
    amino_info = []
    if amino_substitution:
        stored.amino_info = []
        cmd.iterate("elem N within 1.8 of (chain " + chain_array[0] + " and resi " +
minus_one_nucleotide +
                    " and name C2')", "stored.amino_info.append((index, name, elem, resn, resi, chain,
b))")
        amino_info = stored.amino_info

    stored.O2_info_1 = []
    stored.O2_info_2 = []
    stored.O2_info_3 = []
    stored.O2_info_4 = []
    stored.O2_info_5 = []
    stored.C2_info_1 = []
    stored.C2_info_2 = []
    stored.C2_info_3 = []
    stored.C2_info_4 = []
    stored.C2_info_5 = []

    # Generates list(s) of all O2's within each nucleotide linkage for the chains specified by the
user
```

```
if chain_array[0] == 'NONE':
   cmd.iterate("name O2'",
              "stored.O2_info_1.append((index, name, elem, resn, resi, chain, b))")
   cmd.iterate("name C2'",
              "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
   O2_info_1 = np.array(stored.O2_info_1)
   C2_info_1 = np.array(stored.C2_info_1)

   # Removes the last O2' if it resides on the last residue
   if C2_info_1[-1][4] == O2_info_1[-1][4] and C2_info_1[-1][5] == O2_info_1[-1][5]:
      O2_info_1 = np.delete(O2_info_1, -1, 0)
   O2_info = O2_info_1

elif len(chain_array) == 1:
   cmd.iterate("name O2' and (chain " + chain_array[0] + ")",
              "stored.O2_info_1.append((index, name, elem, resn, resi, chain, b))")
   cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
              "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
   O2_info_1 = np.array(stored.O2_info_1)
   C2_info_1 = np.array(stored.C2_info_1)

   # Removes the last O2' if it resides on the last residue
   if C2_info_1[-1][4] == O2_info_1[-1][4] and C2_info_1[-1][5] == O2_info_1[-1][5]:
      O2_info_1 = np.delete(O2_info_1, -1, 0)
   O2_info = O2_info_1

elif len(chain_array) == 2:
   cmd.iterate("name O2' and (chain " + chain_array[0] + ")",
              "stored.O2_info_1.append((index, name, elem, resn, resi, chain, b))")
   cmd.iterate("name O2' and (chain " + chain_array[1] + ")",
              "stored.O2_info_2.append((index, name, elem, resn, resi, chain, b))")
   cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
              "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
   cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
              "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
   O2_info_1 = np.array(stored.O2_info_1)
   O2_info_2 = np.array(stored.O2_info_2)
   C2_info_1 = np.array(stored.C2_info_1)
   C2_info_2 = np.array(stored.C2_info_2)

   # Removes the last O2' if it resides on the last residue
   if C2_info_1[-1][4] == O2_info_1[-1][4] and C2_info_1[-1][5] == O2_info_1[-1][5]:
      O2_info_1 = np.delete(O2_info_1, -1, 0)
   if C2_info_2[-1][4] == O2_info_2[-1][4] and C2_info_2[-1][5] == O2_info_2[-1][5]:
      O2_info_2 = np.delete(O2_info_2, -1, 0)
   O2_info = np.concatenate((O2_info_1, O2_info_2))
```

```
elif len(chain_array) == 3:
    cmd.iterate("name O2' and (chain " + chain_array[0] + ")",
                "stored.O2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O2' and (chain " + chain_array[1] + ")",
                "stored.O2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O2' and (chain " + chain_array[2] + ")",
                "stored.O2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
                "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
    O2_info_1 = np.array(stored.O2_info_1)
    O2_info_2 = np.array(stored.O2_info_2)
    O2_info_3 = np.array(stored.O2_info_3)
    C2_info_1 = np.array(stored.C2_info_1)
    C2_info_2 = np.array(stored.C2_info_2)
    C2_info_3 = np.array(stored.C2_info_3)

    # Removes the last O2' if it resides on the last residue
    if C2_info_1[-1][4] == O2_info_1[-1][4] and C2_info_1[-1][5] == O2_info_1[-1][5]:
        O2_info_1 = np.delete(O2_info_1, -1, 0)
    if C2_info_2[-1][4] == O2_info_2[-1][4] and C2_info_2[-1][5] == O2_info_2[-1][5]:
        O2_info_2 = np.delete(O2_info_2, -1, 0)
    if C2_info_3[-1][4] == O2_info_3[-1][4] and C2_info_3[-1][5] == O2_info_3[-1][5]:
        O2_info_3 = np.delete(O2_info_3, -1, 0)
    O2_info = np.concatenate((O2_info_1, O2_info_2, O2_info_3))

elif len(chain_array) == 4:
    cmd.iterate("name O2' and (chain " + chain_array[0] + ")",
                "stored.O2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O2' and (chain " + chain_array[1] + ")",
                "stored.O2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O2' and (chain " + chain_array[2] + ")",
                "stored.O2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O2' and (chain " + chain_array[3] + ")",
                "stored.O2_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
                "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[3] + ")",
            "stored.C2_info_4.append((index, name, elem, resn, resi, chain, b))")
    O2_info_1 = np.array(stored.O2_info_1)
    O2_info_2 = np.array(stored.O2_info_2)
    O2_info_3 = np.array(stored.O2_info_3)
    O2_info_4 = np.array(stored.O2_info_4)
    C2_info_1 = np.array(stored.C2_info_1)
    C2_info_2 = np.array(stored.C2_info_2)
    C2_info_3 = np.array(stored.C2_info_3)
    C2_info_4 = np.array(stored.C2_info_4)

    # Removes the last O2' if it resides on the last residue
    if C2_info_1[-1][4] == O2_info_1[-1][4] and C2_info_1[-1][5] == O2_info_1[-1][5]:
        O2_info_1 = np.delete(O2_info_1, -1, 0)
    if C2_info_2[-1][4] == O2_info_2[-1][4] and C2_info_2[-1][5] == O2_info_2[-1][5]:
        O2_info_2 = np.delete(O2_info_2, -1, 0)
    if C2_info_3[-1][4] == O2_info_3[-1][4] and C2_info_3[-1][5] == O2_info_3[-1][5]:
        O2_info_3 = np.delete(O2_info_3, -1, 0)
    if C2_info_4[-1][4] == O2_info_4[-1][4] and C2_info_4[-1][5] == O2_info_4[-1][5]:
        O2_info_4 = np.delete(O2_info_4, -1, 0)
    O2_info = np.concatenate((O2_info_1, O2_info_2, O2_info_3, O2_info_4))

elif len(chain_array) == 5:
    cmd.iterate("name O2' and (chain " + chain_array[0] + ")",
            "stored.O2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O2' and (chain " + chain_array[1] + ")",
            "stored.O2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O2' and (chain " + chain_array[2] + ")",
            "stored.O2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O2' and (chain " + chain_array[3] + ")",
            "stored.O2_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O2' and (chain " + chain_array[4] + ")",
            "stored.O2_info_5.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
            "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
            "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
            "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[3] + ")",
            "stored.C2_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[4] + ")",
            "stored.C2_info_5.append((index, name, elem, resn, resi, chain, b))")
    O2_info_1 = np.array(stored.O2_info_1)
    O2_info_2 = np.array(stored.O2_info_2)
    O2_info_3 = np.array(stored.O2_info_3)
    O2_info_4 = np.array(stored.O2_info_4)
```

```
        O2_info_5 = np.array(stored.O2_info_5)
        C2_info_1 = np.array(stored.C2_info_1)
        C2_info_2 = np.array(stored.C2_info_2)
        C2_info_3 = np.array(stored.C2_info_3)
        C2_info_4 = np.array(stored.C2_info_4)
        C2_info_5 = np.array(stored.C2_info_5)

        # Removes the last O2' if it resides on the last residue
        if C2_info_1[-1][4] == O2_info_1[-1][4] and C2_info_1[-1][5] == O2_info_1[-1][5]:
            O2_info_1 = np.delete(O2_info_1, -1, 0)
        if C2_info_2[-1][4] == O2_info_2[-1][4] and C2_info_2[-1][5] == O2_info_2[-1][5]:
            O2_info_2 = np.delete(O2_info_2, -1, 0)
        if C2_info_3[-1][4] == O2_info_3[-1][4] and C2_info_3[-1][5] == O2_info_3[-1][5]:
            O2_info_3 = np.delete(O2_info_3, -1, 0)
        if C2_info_4[-1][4] == O2_info_4[-1][4] and C2_info_4[-1][5] == O2_info_4[-1][5]:
            O2_info_4 = np.delete(O2_info_4, -1, 0)
        if C2_info_5[-1][4] == O2_info_5[-1][4] and C2_info_5[-1][5] == O2_info_5[-1][5]:
            O2_info_5 = np.delete(O2_info_5, -1, 0)
        O2_info = np.concatenate((O2_info_1, O2_info_2, O2_info_3, O2_info_4, O2_info_5))

    else:
        print("ERROR: This plugin cannot handle more than 5 chains.")
        return

    # Create a file to write non scissile phosphate data to
    with open(directory + '\\%s.txt' % (pdb + " Non-Scissile Phosphate O2' Contacts"), "w") as
f_non_scissile:

        # Creates a header
        f_non_scissile.write("***%s Contacts***\n" % (pdb + " Non-Scissile Phosphate O2'
Contacts"))
        f_non_scissile.write("aoiResi" + '\t' + "aoiAtom" + '\t' + "caIndex" + '\t' + "caAtom" + '\t' +
"caResi" + '\t'
                             + "caChain" + '\t' + "Dist" + '\n')

        # If an amino was found, determines the distances of CAs to the amino and writes them to a
separate text file
        for a in range(len(amino_info)):

            # Give a default chain letter of Z if no chain letter was previously assigned
            if len(amino_info[a][5]) < 1:
                amino_info[a][5] = 'Z'

            # Find and stores all contacts within 5 angstroms of the amino
            stored.CA_info = []
            cmd.iterate(("all within 5 of index " + str(amino_info[a][0])),
"stored.CA_info.append((index, name, elem, "
                                                                    "resn, resi, chain, b))")
            CA_info = np.array(stored.CA_info)

            for b in range(len(CA_info)):
                if len(CA_info[b][5]) < 1:
                    CA_info[b][5] = 'Z'

            with open(directory + '\\%s.txt' % (pdb + " Scissile Phosphate O2' Contacts"), "w") as
f_scissile:

                f_scissile.write("***%s Contacts***\n" % (pdb + " Scissile Phosphate O2' Contacts"))
                f_scissile.write("aoiResi" + '\t' + "aoiAtom" + '\t' + "caIndex" + '\t' + "caAtom" + '\t' +
"caResi" +
                              '\t' + "caChain" + '\t' + "Dist" + '\n')

                for b in range(len(CA_info)):
                    dist = cmd.distance(("index %s" % amino_info[a][0]), ("index %s" %
CA_info[b][0]))

                    f_scissile.write(amino_info[a][3] + amino_info[a][4] + '\t' + amino_info[a][1] + '\t' +
                                  str(CA_info[b][0]) + '\t' + CA_info[b][1] + '\t' + CA_info[b][3] +
CA_info[b][4] +
                                  '\t' + CA_info[b][5] + '\t' + str(dist) + '\n')

        # Iterates through all the stored O2's
        for a in range(len(O2_info)):

            if len(O2_info[a][5]) < 1:
                O2_info[a][5] = 'Z'

            # Find and stores all contacts within 5 angstroms of each O2'
            stored.CA_info = []
            cmd.iterate(("all within 5 of index " + str(O2_info[a][0])),
"stored.CA_info.append((index, name, elem, "
                                                                  "resn, resi, chain, b))")
            CA_info = np.array(stored.CA_info)

            for b in range(len(CA_info)):
                if len(CA_info[b][5]) < 1:
                    CA_info[b][5] = 'Z'

            # Determines the CA distances to the O2' at the scissile phosphate and writes them to a
separate text file
```

```
            if O2_info[a][4] == minus_one_nucleotide and (O2_info[a][5] == chain_array[0] or
chain_array[0] == 'NONE'):
                with open(directory + '\\%s.txt' % (pdb + " Scissile Phosphate O2' Contacts"), "w") as
f_scissile:

                    f_scissile.write("***%s Contacts***\n" % (pdb + " Scissile Phosphate O2'
Contacts"))
                    f_scissile.write("aoiResi" + '\t' + "aoiAtom" + '\t' + "caIndex" + '\t' + "caAtom" + '\t'
+ "caResi"
                                     + '\t' + "caChain" + '\t' + "Dist" + '\n')

                    for b in range(len(CA_info)):
                        dist = cmd.distance(("index %s" % O2_info[a][0]), ("index %s" %
CA_info[b][0]))

                        f_scissile.write(O2_info[a][3] + O2_info[a][4] + '\t' + O2_info[a][1] + '\t' +
                                         str(CA_info[b][0]) + '\t' + CA_info[b][1] + '\t' + CA_info[b][3] +
                                         CA_info[b][4] + '\t' + CA_info[b][5] + '\t' + str(dist) + '\n')

                continue

            # Determines the CA distances to the O2' and writes them to the non-scissile text file
            for b in range(len(CA_info)):
                dist = cmd.distance(("index %s" % O2_info[a][0]), ("index %s" % CA_info[b][0]))

                f_non_scissile.write(O2_info[a][3] + O2_info[a][4] + '\t' + O2_info[a][1] + '\t' +
str(CA_info[b][0]) +
                                     '\t' + CA_info[b][1] + '\t' + CA_info[b][3] + CA_info[b][4] + '\t' +
CA_info[b][5]
                                     + '\t' + str(dist) + '\n')

    # Deletes all distance objects in PyMOL to prevent PyMOL from crashing by storing too
many objects
    cmd.delete("dist*")

    print("O2' Data Collection Complete")

def find_NBO_CA(plus_one_nucleotide, chain_array):

    stored.pdb = ""
    cmd.iterate("index 1", "stored.pdb = model")
    pdb = stored.pdb

    stored.OP1_info_1 = []
    stored.OP1_info_2 = []
    stored.OP1_info_3 = []
    stored.OP1_info_4 = []
    stored.OP1_info_5 = []
    stored.C2_info_1 = []
    stored.C2_info_2 = []
    stored.C2_info_3 = []
    stored.C2_info_4 = []
    stored.C2_info_5 = []

    # Generates list(s) of all OP1s within each nucleotide linkage for the chains specified by the
user
    if chain_array[0] == 'NONE':
        cmd.iterate("name OP1",
                    "stored.OP1_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2'",
                    "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        OP1_info_1 = np.array(stored.OP1_info_1)
        C2_info_1 = np.array(stored.C2_info_1)

        # Removes the first OP1 if it resides on the first residue
        while C2_info_1[0][4] == OP1_info_1[0][4] and C2_info_1[0][5] == OP1_info_1[0][5]:
            OP1_info_1 = np.delete(OP1_info_1, 0, 0)
        OP1_info = OP1_info_1

    elif len(chain_array) == 1:
        cmd.iterate("name OP1 and (chain " + chain_array[0] + ")",
                    "stored.OP1_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                    "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        OP1_info_1 = np.array(stored.OP1_info_1)
        C2_info_1 = np.array(stored.C2_info_1)

        # Removes the first OP1 if it resides on the first residue
        while C2_info_1[0][4] == OP1_info_1[0][4] and C2_info_1[0][5] == OP1_info_1[0][5]:
            OP1_info_1 = np.delete(OP1_info_1, 0, 0)
        OP1_info = OP1_info_1

    elif len(chain_array) == 2:
        cmd.iterate("name OP1 and (chain " + chain_array[0] + ")",
                    "stored.OP1_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name OP1 and (chain " + chain_array[1] + ")",
                    "stored.OP1_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                    "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                    "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
        OP1_info_1 = np.array(stored.OP1_info_1)
        OP1_info_2 = np.array(stored.OP1_info_2)
        C2_info_1 = np.array(stored.C2_info_1)
        C2_info_2 = np.array(stored.C2_info_2)

        # Removes the first OP1 if it resides on the first residue
        while C2_info_1[0][4] == OP1_info_1[0][4] and C2_info_1[0][5] == OP1_info_1[0][5]:
            OP1_info_1 = np.delete(OP1_info_1, 0, 0)
        while C2_info_2[0][4] == OP1_info_2[0][4] and C2_info_2[0][5] == OP1_info_2[0][5]:
            OP1_info_2 = np.delete(OP1_info_2, 0, 0)
        OP1_info = np.concatenate((OP1_info_1, OP1_info_2))

    elif len(chain_array) == 3:
        cmd.iterate("name OP1 and (chain " + chain_array[0] + ")",
                    "stored.OP1_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name OP1 and (chain " + chain_array[1] + ")",
                    "stored.OP1_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name OP1 and (chain " + chain_array[2] + ")",
                    "stored.OP1_info_3.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                    "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                    "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
                    "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
        OP1_info_1 = np.array(stored.OP1_info_1)
        OP1_info_2 = np.array(stored.OP1_info_2)
        OP1_info_3 = np.array(stored.OP1_info_3)
        C2_info_1 = np.array(stored.C2_info_1)
        C2_info_2 = np.array(stored.C2_info_2)
        C2_info_3 = np.array(stored.C2_info_3)

        # Removes the first OP1 if it resides on the first residue
        while C2_info_1[0][4] == OP1_info_1[0][4] and C2_info_1[0][5] == OP1_info_1[0][5]:
            OP1_info_1 = np.delete(OP1_info_1, 0, 0)
        while C2_info_2[0][4] == OP1_info_2[0][4] and C2_info_2[0][5] == OP1_info_2[0][5]:
            OP1_info_2 = np.delete(OP1_info_2, 0, 0)
        while C2_info_3[0][4] == OP1_info_3[0][4] and C2_info_3[0][5] == OP1_info_3[0][5]:
            OP1_info_3 = np.delete(OP1_info_3, 0, 0)
        OP1_info = np.concatenate((OP1_info_1, OP1_info_2, OP1_info_3))

    elif len(chain_array) == 4:
        cmd.iterate("name OP1 and (chain " + chain_array[0] + ")",
                    "stored.OP1_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name OP1 and (chain " + chain_array[1] + ")",
                    "stored.OP1_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name OP1 and (chain " + chain_array[2] + ")",
                "stored.OP1_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP1 and (chain " + chain_array[3] + ")",
                "stored.OP1_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
                "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[3] + ")",
                "stored.C2_info_4.append((index, name, elem, resn, resi, chain, b))")
    OP1_info_1 = np.array(stored.OP1_info_1)
    OP1_info_2 = np.array(stored.OP1_info_2)
    OP1_info_3 = np.array(stored.OP1_info_3)
    OP1_info_4 = np.array(stored.OP1_info_4)
    C2_info_1 = np.array(stored.C2_info_1)
    C2_info_2 = np.array(stored.C2_info_2)
    C2_info_3 = np.array(stored.C2_info_3)
    C2_info_4 = np.array(stored.C2_info_4)

    # Removes the first OP1 if it resides on the first residue
    while C2_info_1[0][4] == OP1_info_1[0][4] and C2_info_1[0][5] == OP1_info_1[0][5]:
        OP1_info_1 = np.delete(OP1_info_1, 0, 0)
    while C2_info_2[0][4] == OP1_info_2[0][4] and C2_info_2[0][5] == OP1_info_2[0][5]:
        OP1_info_2 = np.delete(OP1_info_2, 0, 0)
    while C2_info_3[0][4] == OP1_info_3[0][4] and C2_info_3[0][5] == OP1_info_3[0][5]:
        OP1_info_3 = np.delete(OP1_info_3, 0, 0)
    while C2_info_4[0][4] == OP1_info_4[0][4] and C2_info_4[0][5] == OP1_info_4[0][5]:
        OP1_info_4 = np.delete(OP1_info_4, 0, 0)
    OP1_info = np.concatenate((OP1_info_1, OP1_info_2, OP1_info_3, OP1_info_4))

elif len(chain_array) == 5:
    cmd.iterate("name OP1 and (chain " + chain_array[0] + ")",
                "stored.OP1_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP1 and (chain " + chain_array[1] + ")",
                "stored.OP1_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP1 and (chain " + chain_array[2] + ")",
                "stored.OP1_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP1 and (chain " + chain_array[3] + ")",
                "stored.OP1_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP1 and (chain " + chain_array[4] + ")",
                "stored.OP1_info_5.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
                    "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[3] + ")",
                    "stored.C2_info_4.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[4] + ")",
                    "stored.C2_info_5.append((index, name, elem, resn, resi, chain, b))")
        OP1_info_1 = np.array(stored.OP1_info_1)
        OP1_info_2 = np.array(stored.OP1_info_2)
        OP1_info_3 = np.array(stored.OP1_info_3)
        OP1_info_4 = np.array(stored.OP1_info_4)
        OP1_info_5 = np.array(stored.OP1_info_5)
        C2_info_1 = np.array(stored.C2_info_1)
        C2_info_2 = np.array(stored.C2_info_2)
        C2_info_3 = np.array(stored.C2_info_3)
        C2_info_4 = np.array(stored.C2_info_4)
        C2_info_5 = np.array(stored.C2_info_5)

        # Removes the first OP1 if it resides on the first residue
        if C2_info_1[0][4] == OP1_info_1[0][4] and C2_info_1[0][5] == OP1_info_1[0][5]:
            OP1_info_1 = np.delete(OP1_info_1, 0, 0)
        if C2_info_2[0][4] == OP1_info_2[0][4] and C2_info_2[0][5] == OP1_info_2[0][5]:
            OP1_info_2 = np.delete(OP1_info_2, 0, 0)
        if C2_info_3[0][4] == OP1_info_3[0][4] and C2_info_3[0][5] == OP1_info_3[0][5]:
            OP1_info_3 = np.delete(OP1_info_3, 0, 0)
        if C2_info_4[0][4] == OP1_info_4[0][4] and C2_info_4[0][5] == OP1_info_4[0][5]:
            OP1_info_4 = np.delete(OP1_info_4, 0, 0)
        if C2_info_5[0][4] == OP1_info_5[0][4] and C2_info_5[0][5] == OP1_info_5[0][5]:
            OP1_info_5 = np.delete(OP1_info_5, 0, 0)
        OP1_info = np.concatenate((OP1_info_1, OP1_info_2, OP1_info_3, OP1_info_4,
OP1_info_5))

    else:
        print("ERROR: This plugin cannot handle more than 5 chains.")
        return

    stored.OP2_info_1 = []
    stored.OP2_info_2 = []
    stored.OP2_info_3 = []
    stored.OP2_info_4 = []
    stored.OP2_info_5 = []
    stored.C2_info_1 = []
    stored.C2_info_2 = []
    stored.C2_info_3 = []
    stored.C2_info_4 = []
    stored.C2_info_5 = []
    # Generates list(s) of all OP2s within each nucleotide linkage for the chains specified by the
user
    if chain_array[0] == 'NONE':
        cmd.iterate("name OP2",
                    "stored.OP2_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2'",
                    "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        OP2_info_1 = np.array(stored.OP2_info_1)
        C2_info_1 = np.array(stored.C2_info_1)

        # Removes the first OP2 if it resides on the first residue
        while C2_info_1[0][4] == OP2_info_1[0][4] and C2_info_1[0][5] == OP2_info_1[0][5]:
            OP2_info_1 = np.delete(OP2_info_1, 0, 0)
        OP2_info = OP2_info_1

    elif len(chain_array) == 1:
        cmd.iterate("name OP2 and (chain " + chain_array[0] + ")",
                    "stored.OP2_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                    "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        OP2_info_1 = np.array(stored.OP2_info_1)
        C2_info_1 = np.array(stored.C2_info_1)

        # Removes the first OP2 if it resides on the first residue
        while C2_info_1[0][4] == OP2_info_1[0][4] and C2_info_1[0][5] == OP2_info_1[0][5]:
            OP2_info_1 = np.delete(OP2_info_1, 0, 0)
        OP2_info = OP2_info_1

    elif len(chain_array) == 2:
        cmd.iterate("name OP2 and (chain " + chain_array[0] + ")",
                    "stored.OP2_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name OP2 and (chain " + chain_array[1] + ")",
                    "stored.OP2_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                    "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                    "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
        OP2_info_1 = np.array(stored.OP2_info_1)
        OP2_info_2 = np.array(stored.OP2_info_2)
        C2_info_1 = np.array(stored.C2_info_1)
        C2_info_2 = np.array(stored.C2_info_2)

        # Removes the first OP2 if it resides on the first residue
        while C2_info_1[0][4] == OP2_info_1[0][4] and C2_info_1[0][5] == OP2_info_1[0][5]:
            OP2_info_1 = np.delete(OP2_info_1, 0, 0)
        while C2_info_2[0][4] == OP2_info_2[0][4] and C2_info_2[0][5] == OP2_info_2[0][5]:
```

```
        OP2_info_2 = np.delete(OP2_info_2, 0, 0)
    OP2_info = np.concatenate((OP2_info_1, OP2_info_2))

elif len(chain_array) == 3:
    cmd.iterate("name OP2 and (chain " + chain_array[0] + ")",
                "stored.OP2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP2 and (chain " + chain_array[1] + ")",
                "stored.OP2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP2 and (chain " + chain_array[2] + ")",
                "stored.OP2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
                "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
    OP2_info_1 = np.array(stored.OP2_info_1)
    OP2_info_2 = np.array(stored.OP2_info_2)
    OP2_info_3 = np.array(stored.OP2_info_3)
    C2_info_1 = np.array(stored.C2_info_1)
    C2_info_2 = np.array(stored.C2_info_2)
    C2_info_3 = np.array(stored.C2_info_3)

    # Removes the first OP2 if it resides on the first residue
    while C2_info_1[0][4] == OP2_info_1[0][4] and C2_info_1[0][5] == OP2_info_1[0][5]:
        OP2_info_1 = np.delete(OP2_info_1, 0, 0)
    while C2_info_2[0][4] == OP2_info_2[0][4] and C2_info_2[0][5] == OP2_info_2[0][5]:
        OP2_info_2 = np.delete(OP2_info_2, 0, 0)
    while C2_info_3[0][4] == OP2_info_3[0][4] and C2_info_3[0][5] == OP2_info_3[0][5]:
        OP2_info_3 = np.delete(OP2_info_3, 0, 0)
    OP2_info = np.concatenate((OP2_info_1, OP2_info_2, OP2_info_3))

elif len(chain_array) == 4:
    cmd.iterate("name OP2 and (chain " + chain_array[0] + ")",
                "stored.OP2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP2 and (chain " + chain_array[1] + ")",
                "stored.OP2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP2 and (chain " + chain_array[2] + ")",
                "stored.OP2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP2 and (chain " + chain_array[3] + ")",
                "stored.OP2_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
            "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[3] + ")",
            "stored.C2_info_4.append((index, name, elem, resn, resi, chain, b))")
    OP2_info_1 = np.array(stored.OP2_info_1)
    OP2_info_2 = np.array(stored.OP2_info_2)
    OP2_info_3 = np.array(stored.OP2_info_3)
    OP2_info_4 = np.array(stored.OP2_info_4)
    C2_info_1 = np.array(stored.C2_info_1)
    C2_info_2 = np.array(stored.C2_info_2)
    C2_info_3 = np.array(stored.C2_info_3)
    C2_info_4 = np.array(stored.C2_info_4)

    # Removes the first OP2 if it resides on the first residue
    while C2_info_1[0][4] == OP2_info_1[0][4] and C2_info_1[0][5] == OP2_info_1[0][5]:
        OP2_info_1 = np.delete(OP2_info_1, 0, 0)
    while C2_info_2[0][4] == OP2_info_2[0][4] and C2_info_2[0][5] == OP2_info_2[0][5]:
        OP2_info_2 = np.delete(OP2_info_2, 0, 0)
    while C2_info_3[0][4] == OP2_info_3[0][4] and C2_info_3[0][5] == OP2_info_3[0][5]:
        OP2_info_3 = np.delete(OP2_info_3, 0, 0)
    while C2_info_4[0][4] == OP2_info_4[0][4] and C2_info_4[0][5] == OP2_info_4[0][5]:
        OP2_info_4 = np.delete(OP2_info_4, 0, 0)
    OP2_info = np.concatenate((OP2_info_1, OP2_info_2, OP2_info_3, OP2_info_4))

elif len(chain_array) == 5:
    cmd.iterate("name OP2 and (chain " + chain_array[0] + ")",
            "stored.OP2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP2 and (chain " + chain_array[1] + ")",
            "stored.OP2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP2 and (chain " + chain_array[2] + ")",
            "stored.OP2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP2 and (chain " + chain_array[3] + ")",
            "stored.OP2_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name OP2 and (chain " + chain_array[4] + ")",
            "stored.OP2_info_5.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
            "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
            "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
            "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[3] + ")",
            "stored.C2_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[4] + ")",
            "stored.C2_info_5.append((index, name, elem, resn, resi, chain, b))")
    OP2_info_1 = np.array(stored.OP2_info_1)
    OP2_info_2 = np.array(stored.OP2_info_2)
```

```
        OP2_info_3 = np.array(stored.OP2_info_3)
        OP2_info_4 = np.array(stored.OP2_info_4)
        OP2_info_5 = np.array(stored.OP2_info_5)
        C2_info_1 = np.array(stored.C2_info_1)
        C2_info_2 = np.array(stored.C2_info_2)
        C2_info_3 = np.array(stored.C2_info_3)
        C2_info_4 = np.array(stored.C2_info_4)
        C2_info_5 = np.array(stored.C2_info_5)

        # Removes the first OP2 if it resides on the first residue
        if C2_info_1[0][4] == OP2_info_1[0][4] and C2_info_1[0][5] == OP2_info_1[0][5]:
            OP2_info_1 = np.delete(OP2_info_1, 0, 0)
        if C2_info_2[0][4] == OP2_info_2[0][4] and C2_info_2[0][5] == OP2_info_2[0][5]:
            OP2_info_2 = np.delete(OP2_info_2, 0, 0)
        if C2_info_3[0][4] == OP2_info_3[0][4] and C2_info_3[0][5] == OP2_info_3[0][5]:
            OP2_info_3 = np.delete(OP2_info_3, 0, 0)
        if C2_info_4[0][4] == OP2_info_4[0][4] and C2_info_4[0][5] == OP2_info_4[0][5]:
            OP2_info_4 = np.delete(OP2_info_4, 0, 0)
        if C2_info_5[0][4] == OP2_info_5[0][4] and C2_info_5[0][5] == OP2_info_5[0][5]:
            OP2_info_5 = np.delete(OP2_info_5, 0, 0)
        OP2_info = np.concatenate((OP2_info_1, OP2_info_2, OP2_info_3, OP2_info_4,
OP2_info_5))

    else:
        print("ERROR: This plugin cannot handle more than 5 chains.")
        return

    # Create a file to write non scissile phosphate data to
    with open(directory + '\\%s.txt' % (pdb + " Non-Scissile Phosphate NBO Contacts"), "w") as
f_non_scissile:

        # Creates a header
        f_non_scissile.write("***%s Contacts***\n" % (pdb + " Non-Scissile Phosphate NBO
Contacts"))
        f_non_scissile.write("aoiResi" + '\t' + "aoiAtom" + '\t' + "caIndex" + '\t' + "caAtom" + '\t' +
"caResi" + '\t'
                             + "caChain" + '\t' + "Dist" + '\n')

        # Create a file to write scissile phosphate data to
        f_scissile = open(directory + '\\%s.txt' % (pdb + " Scissile Phosphate NBO Contacts"), "w")

        # Creates a header
        f_scissile.write("***%s Contacts***\n" % (pdb + " Scissile Phosphate NBO Contacts"))
        f_scissile.write("aoiResi" + '\t' + "aoiAtom" + '\t' + "caIndex" + '\t' + "caAtom" + '\t' +
"caResi" + '\t' +
            "caChain" + '\t' + "Dist" + '\n')
        # Iterates through all the stored OP1s
        for a in range(len(OP1_info)):

            # Give a default chain letter of Z if no chain letter was previously assigned
            if len(OP1_info[a][5]) < 1:
                OP1_info[a][5] = 'Z'

            # Find and stores all contacts within 5 angstroms of each OP1
            stored.CA_info = []
            cmd.iterate(("all within 5 of index " + str(OP1_info[a][0])),
"stored.CA_info.append((index, name, elem, "
                                                  "resn, resi, chain, b))")
            CA_info = np.array(stored.CA_info)

            for b in range(len(CA_info)):
                if len(CA_info[b][5]) < 1:
                    CA_info[b][5] = 'Z'

            # Determines the CA distances to the OP1 at the scissile phosphate and writes them to the
scissile phosphate
            # text file
            if OP1_info[a][4] == plus_one_nucleotide and (OP1_info[a][5] == chain_array[0] or
chain_array[0] == 'NONE'):

                for b in range(len(CA_info)):
                    dist = cmd.distance(("index %s" % OP1_info[a][0]), ("index %s" % CA_info[b][0]))

                    f_scissile.write(OP1_info[a][3] + OP1_info[a][4] + '\t' + OP1_info[a][1] + '\t' +
str(CA_info[b][0])
                             + '\t' + CA_info[b][1] + '\t' + CA_info[b][3] + CA_info[b][4] + '\t' +
                             CA_info[b][5] + '\t' + str(dist) + '\n')

                continue

            # Determines the CA distances to the OP1 and writes them to the non-scissile text file
            for b in range(len(CA_info)):
                dist = cmd.distance(("index %s" % OP1_info[a][0]), ("index %s" % CA_info[b][0]))

                f_non_scissile.write(OP1_info[a][3] + OP1_info[a][4] + '\t' + OP1_info[a][1] + '\t' +
str(CA_info[b][0])
                             + '\t' + CA_info[b][1] + '\t' + CA_info[b][3] + CA_info[b][4] + '\t' +
                             CA_info[b][5] + '\t' + str(dist) + '\n')

        # Deletes all distance objects in PyMOL to prevent PyMOL from crashing by storing too
many objects
```

```
        cmd.delete("dist*")

        # Iterates through all the stored OP2s
        for a in range(len(OP2_info)):

            # Give a default chain letter of Z if no chain letter was previously assigned
            if len(OP2_info[a][5]) < 1:
                OP2_info[a][5] = 'Z'

            # Find and stores all contacts within 5 angstroms of each OP2
            stored.CA_info = []
            cmd.iterate(("all within 5 of index " + str(OP2_info[a][0])),
"stored.CA_info.append((index, name, elem, "
                                                            "resn, resi, chain, b))")
            CA_info = np.array(stored.CA_info)

            for b in range(len(CA_info)):
                if len(CA_info[b][5]) < 1:
                    CA_info[b][5] = 'Z'

            # Determines the CA distances to the OP2 at the scissile phosphate and writes them to the
scissile phosphate
            # text file
            if OP2_info[a][4] == plus_one_nucleotide and (OP2_info[a][5] == chain_array[0] or
chain_array[0] == 'NONE'):

                for b in range(len(CA_info)):
                    dist = cmd.distance(("index %s" % OP2_info[a][0]), ("index %s" % CA_info[b][0]))

                    f_scissile.write(OP2_info[a][3] + OP2_info[a][4] + '\t' + OP2_info[a][1] + '\t' +
str(CA_info[b][0])
                                     + '\t' + CA_info[b][1] + '\t' + CA_info[b][3] + CA_info[b][4] + '\t' +
                                     CA_info[b][5] + '\t' + str(dist) + '\n')

                continue

            # Determines the CA distances to the OP2 and writes them to the non-scissile text file
            for b in range(len(CA_info)):
                dist = cmd.distance(("index %s" % OP2_info[a][0]), ("index %s" % CA_info[b][0]))

                f_non_scissile.write(OP2_info[a][3] + OP2_info[a][4] + '\t' + OP2_info[a][1] + '\t' +
str(CA_info[b][0])
                                     + '\t' + CA_info[b][1] + '\t' + CA_info[b][3] + CA_info[b][4] + '\t' +
                                     CA_info[b][5] + '\t' + str(dist) + '\n')
        # Deletes all distance objects in PyMOL to prevent PyMOL from crashing by storing too
many objects
        cmd.delete("dist*")

        f_scissile.close()

        print("NBO Data Collection Complete")

def find_O5_CA(plus_one_nucleotide, chain_array):

    stored.pdb = ""
    cmd.iterate("index 1", "stored.pdb = model")
    pdb = stored.pdb

    stored.O5_info_1 = []
    stored.O5_info_2 = []
    stored.O5_info_3 = []
    stored.O5_info_4 = []
    stored.O5_info_5 = []
    stored.C2_info_1 = []
    stored.C2_info_2 = []
    stored.C2_info_3 = []
    stored.C2_info_4 = []
    stored.C2_info_5 = []

    # Generates list(s) of all O5's within each nucleotide linkage for the chains specified by the
user
    if chain_array[0] == 'NONE':
        cmd.iterate("name O5'",
                "stored.O5_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2'",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        O5_info_1 = np.array(stored.O5_info_1)
        C2_info_1 = np.array(stored.C2_info_1)

        # Removes the first O5' if it resides on the first residue
        if C2_info_1[0][4] == O5_info_1[0][4] and C2_info_1[0][5] == O5_info_1[0][5]:
            O5_info_1 = np.delete(O5_info_1, 0, 0)
        O5_info = O5_info_1

    elif len(chain_array) == 1:
        cmd.iterate("name O5' and (chain " + chain_array[0] + ")",
                "stored.O5_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        O5_info_1 = np.array(stored.O5_info_1)
        C2_info_1 = np.array(stored.C2_info_1)

        # Removes the first O5' if it resides on the first residue
        if C2_info_1[0][4] == O5_info_1[0][4] and C2_info_1[0][5] == O5_info_1[0][5]:
            O5_info_1 = np.delete(O5_info_1, 0, 0)
        O5_info = O5_info_1

    elif len(chain_array) == 2:
        cmd.iterate("name O5' and (chain " + chain_array[0] + ")",
                "stored.O5_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name O5' and (chain " + chain_array[1] + ")",
                "stored.O5_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
        O5_info_1 = np.array(stored.O5_info_1)
        O5_info_2 = np.array(stored.O5_info_2)
        C2_info_1 = np.array(stored.C2_info_1)
        C2_info_2 = np.array(stored.C2_info_2)

        # Removes the first O5' if it resides on the first residue
        if C2_info_1[0][4] == O5_info_1[0][4] and C2_info_1[0][5] == O5_info_1[0][5]:
            O5_info_1 = np.delete(O5_info_1, 0, 0)
        if C2_info_2[0][4] == O5_info_2[0][4] and C2_info_2[0][5] == O5_info_2[0][5]:
            O5_info_2 = np.delete(O5_info_2, 0, 0)
        O5_info = np.concatenate((O5_info_1, O5_info_2))

    elif len(chain_array) == 3:
        cmd.iterate("name O5' and (chain " + chain_array[0] + ")",
                "stored.O5_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name O5' and (chain " + chain_array[1] + ")",
                "stored.O5_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name O5' and (chain " + chain_array[2] + ")",
                "stored.O5_info_3.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
                "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
        O5_info_1 = np.array(stored.O5_info_1)
        O5_info_2 = np.array(stored.O5_info_2)
        O5_info_3 = np.array(stored.O5_info_3)
        C2_info_1 = np.array(stored.C2_info_1)
        C2_info_2 = np.array(stored.C2_info_2)
```

```
        C2_info_3 = np.array(stored.C2_info_3)

        # Removes the first O5' if it resides on the first residue
        if C2_info_1[0][4] == O5_info_1[0][4] and C2_info_1[0][5] == O5_info_1[0][5]:
            O5_info_1 = np.delete(O5_info_1, 0, 0)
        if C2_info_2[0][4] == O5_info_2[0][4] and C2_info_2[0][5] == O5_info_2[0][5]:
            O5_info_2 = np.delete(O5_info_2, 0, 0)
        if C2_info_3[0][4] == O5_info_3[0][4] and C2_info_3[0][5] == O5_info_3[0][5]:
            O5_info_3 = np.delete(O5_info_3, 0, 0)
        O5_info = np.concatenate((O5_info_1, O5_info_2, O5_info_3))

    elif len(chain_array) == 4:
        cmd.iterate("name O5' and (chain " + chain_array[0] + ")",
                    "stored.O5_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name O5' and (chain " + chain_array[1] + ")",
                    "stored.O5_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name O5' and (chain " + chain_array[2] + ")",
                    "stored.O5_info_3.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name O5' and (chain " + chain_array[3] + ")",
                    "stored.O5_info_4.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                    "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                    "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
                    "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
        cmd.iterate("name C2' and (chain " + chain_array[3] + ")",
                    "stored.C2_info_4.append((index, name, elem, resn, resi, chain, b))")
        O5_info_1 = np.array(stored.O5_info_1)
        O5_info_2 = np.array(stored.O5_info_2)
        O5_info_3 = np.array(stored.O5_info_3)
        O5_info_4 = np.array(stored.O5_info_4)
        C2_info_1 = np.array(stored.C2_info_1)
        C2_info_2 = np.array(stored.C2_info_2)
        C2_info_3 = np.array(stored.C2_info_3)
        C2_info_4 = np.array(stored.C2_info_4)

        # Removes the first O5' if it resides on the first residue
        if C2_info_1[0][4] == O5_info_1[0][4] and C2_info_1[0][5] == O5_info_1[0][5]:
            O5_info_1 = np.delete(O5_info_1, 0, 0)
        if C2_info_2[0][4] == O5_info_2[0][4] and C2_info_2[0][5] == O5_info_2[0][5]:
            O5_info_2 = np.delete(O5_info_2, 0, 0)
        if C2_info_3[0][4] == O5_info_3[0][4] and C2_info_3[0][5] == O5_info_3[0][5]:
            O5_info_3 = np.delete(O5_info_3, 0, 0)
        if C2_info_4[0][4] == O5_info_4[0][4] and C2_info_4[0][5] == O5_info_4[0][5]:
            O5_info_4 = np.delete(O5_info_4, 0, 0)
    O5_info = np.concatenate((O5_info_1, O5_info_2, O5_info_3, O5_info_4))

elif len(chain_array) == 5:
    cmd.iterate("name O5' and (chain " + chain_array[0] + ")",
                "stored.O5_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O5' and (chain " + chain_array[1] + ")",
                "stored.O5_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O5' and (chain " + chain_array[2] + ")",
                "stored.O5_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O5' and (chain " + chain_array[3] + ")",
                "stored.O5_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name O5' and (chain " + chain_array[4] + ")",
                "stored.O5_info_5.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[0] + ")",
                "stored.C2_info_1.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[1] + ")",
                "stored.C2_info_2.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[2] + ")",
                "stored.C2_info_3.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[3] + ")",
                "stored.C2_info_4.append((index, name, elem, resn, resi, chain, b))")
    cmd.iterate("name C2' and (chain " + chain_array[4] + ")",
                "stored.C2_info_5.append((index, name, elem, resn, resi, chain, b))")
    O5_info_1 = np.array(stored.O5_info_1)
    O5_info_2 = np.array(stored.O5_info_2)
    O5_info_3 = np.array(stored.O5_info_3)
    O5_info_4 = np.array(stored.O5_info_4)
    O5_info_5 = np.array(stored.O5_info_5)
    C2_info_1 = np.array(stored.C2_info_1)
    C2_info_2 = np.array(stored.C2_info_2)
    C2_info_3 = np.array(stored.C2_info_3)
    C2_info_4 = np.array(stored.C2_info_4)
    C2_info_5 = np.array(stored.C2_info_5)

    # Removes the first O5' if it resides on the first residue
    if C2_info_1[0][4] == O5_info_1[0][4] and C2_info_1[0][5] == O5_info_1[0][5]:
        O5_info_1 = np.delete(O5_info_1, 0, 0)
    if C2_info_2[0][4] == O5_info_2[0][4] and C2_info_2[0][5] == O5_info_2[0][5]:
        O5_info_2 = np.delete(O5_info_2, 0, 0)
    if C2_info_3[0][4] == O5_info_3[0][4] and C2_info_3[0][5] == O5_info_3[0][5]:
        O5_info_3 = np.delete(O5_info_3, 0, 0)
    if C2_info_4[0][4] == O5_info_4[0][4] and C2_info_4[0][5] == O5_info_4[0][5]:
        O5_info_4 = np.delete(O5_info_4, 0, 0)
    if C2_info_5[0][4] == O5_info_5[0][4] and C2_info_5[0][5] == O5_info_5[0][5]:
        O5_info_5 = np.delete(O5_info_5, 0, 0)
    O5_info = np.concatenate((O5_info_1, O5_info_2, O5_info_3, O5_info_4, O5_info_5))
    else:
        print("ERROR: This plugin cannot handle more than 5 chains.")
        return

    # Create a file to write non scissile phosphate data to
    with open(directory + '\\%s.txt' % (pdb + " Non-Scissile Phosphate O5' Contacts"), "w") as
f_non_scissile:

        # Creates a header
        f_non_scissile.write("***%s Contacts***\n" % (pdb + " Non-Scissile Phosphate O5'
Contacts"))
        f_non_scissile.write("aoiResi" + '\t' + "aoiAtom" + '\t' + "caIndex" + '\t' + "caAtom" + '\t' +
"caResi" + '\t'
                             + "caChain" + '\t' + "Dist" + '\n')

        # Iterates through all the stored O5's
        for a in range(len(O5_info)):

            # Give a default chain letter of Z if no chain letter was previously assigned
            if len(O5_info[a][5]) < 1:
                O5_info[a][5] = 'Z'

            # Find and stores all contacts within 5 angstroms of each O5'
            stored.CA_info = []
            cmd.iterate(("all within 5 of index " + str(O5_info[a][0])),
"stored.CA_info.append((index, name, elem, "
                                                              "resn, resi, chain, b))")
            CA_info = np.array(stored.CA_info)

            for b in range(len(CA_info)):
                if len(CA_info[b][5]) < 1:
                    CA_info[b][5] = 'Z'

            # Determines the CA distances to the O5' at the scissile phosphate and writes them to a
separate text file
            if O5_info[a][4] == plus_one_nucleotide and (O5_info[a][5] == chain_array[0] or
chain_array[0] == 'NONE'):
                with open(directory + '\\%s.txt' % (pdb + " Scissile Phosphate O5' Contacts"), "w") as
f_scissile:

                    f_scissile.write("***%s Contacts***\n" % (pdb + " Scissile Phosphate O5'
Contacts"))
                    f_scissile.write("aoiResi" + '\t' + "aoiAtom" + '\t' + "caIndex" + '\t' + "caAtom" + '\t'
+ "caResi"
                                     + '\t' + "caChain" + '\t' + "Dist" + '\n')
                    for b in range(len(CA_info)):
                        dist = cmd.distance(("index %s" % O5_info[a][0]), ("index %s" %
CA_info[b][0]))

                        f_scissile.write(O5_info[a][3] + O5_info[a][4] + '\t' + O5_info[a][1] + '\t' +
                                    str(CA_info[b][0]) + '\t' + CA_info[b][1] + '\t' + CA_info[b][3] +
                                    CA_info[b][4] + '\t' + CA_info[b][5] + '\t' + str(dist) + '\n')

                continue

            # Determines the CA distances to the O5' and writes them to the non-scissile text file
            for b in range(len(CA_info)):
                dist = cmd.distance(("index %s" % O5_info[a][0]), ("index %s" % CA_info[b][0]))

                f_non_scissile.write(O5_info[a][3] + O5_info[a][4] + '\t' + O5_info[a][1] + '\t' +
str(CA_info[b][0]) +
                                '\t' + CA_info[b][1] + '\t' + CA_info[b][3] + CA_info[b][4] + '\t' +
CA_info[b][5]
                                + '\t' + str(dist) + '\n')

    # Deletes all distance objects in PyMOL to prevent PyMOL from crashing by storing too
many objects
    cmd.delete("dist*")

    print("O5' Data Collection Complete")
```

IV. Scissile Phosphate Downstream Processing Script

```
# -*- coding: utf-8 -*-
# This script calculate stats on the crystal structures
import os
import numpy as np
import csv
import matplotlib.pyplot as plt
from matplotlib.path import Path
from scipy.misc import imread
import sys
import warnings
import matplotlib as mpl

mpl.rc('font', family='arial')

# These commands make it so we can print the angstrom sign
reload(sys)
sys.setdefaultencoding('utf8')

# Don't show numpy deprecation warnings
warnings.filterwarnings("ignore", category=np.VisibleDeprecationWarning)


def extract_resi_number(resn):
    # This function accepts a residue (Ex: G33) as an argument and will return its
    # residue number (33) as the output
    resi = []
    for n in range(1, len(resn)):
        try:
            resi.append(int(resn[n]))
        except ValueError:
            resi = []
    resi = ''.join(map(str, resi))

    if len(resi) == 0:
        return -10
    return int(resi)


def PDBtoRBZ():
    # This function takes no arguments and iterates through all filenames in the cwd.
    # It then identifies a given rbz based on the PDBs present in the filenames. The function
returns the rbz name,
    # modlist and mutant list. Modlist is a list of two lists of residue names. Modlist is used to
handle numbering
```

```
    # discrepancies. So if a given residue is present in the first of two lists, the corresponding
value in the second
    # list is assigned to the residue. The mutant list of each rbz is a list containing all mutant
residue names
    # so that they can be identified

    RBZlist = []

    glmS = ["2GCS", "2H0S", "2H0W", "2H0Z", "2HO6", "2H07", "2NZ4", "3g8s", "3G9C",
"3g8t", "3l3c", "3g96",
            "2Z75", "2Z74", "2H0X", "3B4A", "3B4B", "3B4C", "2GCV"]
    HH = ["1MME", "299D", "300D", "301D", "359D", "2OEU", "3ZD4", "3ZD5", "3ZP8",
"5DI2", "5DI4", "5DQK", "5DH6",
          "5DH7", "5DH8", "5EAO", "5EAQ", "1NYI", "1Q29", "2QUS", "1HMH", "1RMN",
"2QUW"]
    Twister = ["4RGE", "4RGF", "4OJI", "4QJD", "4QJH", "5DUN", "ACTI", "NEUT", "TS"]
    Hairpin = ["2D2K", "1X9K", "1X9C", "2D2L", "2OUE", "1ZFT", "2FGP", "1ZFV", "1ZFX",
"3B5A", "3B58", "3B5F", "3B5S",
               "3B91", "3BBI", "3BBK", "3BBM", "3CR1", "3I2Q", "3I2R", "3I2S", "3I2U",
"2NPY", "2NPZ", "2P7D", "2P7E",
               "2P7F", "1M5K", "1M5O", "1M5V", "3CQS", "4G6P", "4G6R", "4G6S", "3GS1",
"3GS8", "2BCY", "2BCZ", "3GS5",
               "1M5P"]
    Pistol = ["5K7C", "5K7D", "5K7E", "5KTJ"]
    HDV = ["2OJ3", "2OIH", "1SJ3", "1VBX", "1VBY", "1VBZ", "1VC6", "1SJF", "1VC0",
"1VC5", "1SJ4", "4PRF", "4PR6"]
    VS = ["4R4P", "4R4V", "5V3I"]
    twister_sister = ["5T5A"]

    for filename in os.listdir(os.getcwd()):
        if filename == "Search.txt" or ".txt" not in filename:
            continue
        PDB = filename.split("_")[0]
        PDB = PDB.upper()
        # Change the name of file_name if working with Darrin's files
        if "Active" in filename:
            PDB = "ACTI"
        elif "Neutral" in filename:
            PDB = "NEUT"
        elif "TS" in filename:
            PDB = "TS"

        if PDB in glmS:
            RBZlist.append("glmS")
        elif PDB in HH:
            RBZlist.append("HH")
        elif PDB in Twister:
            RBZlist.append("Twister")
        elif PDB in Hairpin:
            RBZlist.append("Hairpin")
        elif PDB in Pistol:
            RBZlist.append("Pistol")
        elif PDB in HDV:
            RBZlist.append("HDV")
        elif PDB in twister_sister:
            RBZlist.append("Twi_sis")
        elif PDB in VS:
            RBZlist.append("VS")

    r_b_z = RBZlist[0]

    # Check to see if all structures are from the same ribozyme
    uniform = 1
    for n in range(0, len(RBZlist)):
        if RBZlist[n] != r_b_z:
            uniform = 0
            break
    if uniform == 1:
        r_b_z = RBZlist[0]
    else:
        r_b_z = "UND"
        print("ERROR: Not all PDB's are recognized")

    # The left part of the array is the part to be changed, the right is the new part
    glms_mutant_list = ["G33A", "Glc6P", "6MN"]
    glms_mod_list = np.array(
        [["2AD0", "A2M", "G39", "G40", "U51", "U67", "G65", "G6P", "GLP", "HOH", "A32",
"A33", "A38", "A40", "6MN12",
          "G66", "A-1", "3AD-1"],
         ["A0", "A0", "G32", "G33", "U43", "U59", "G57", "Glc6P", "GlcN6P", "HOH", "G33A",
"G33A", "A31", "G33A",
          "6MN", "G59", "A0", "A0"]])

    hh_mutant_list = ["G12A"]
    hh_mod_list = np.array(
        [["HOH", "Mn", "Mg", "Na", "A36", "G39", "C38", "A39", "A19", "G21", "G20", "A22",
"G36", "OMC16", "CVC7",
          "U8", "DC7", "OMC6", "C7"],
         ["HOH", "MT", "MT", "Na", "G12A", "G12", "C35", "G12A", "A18", "G8", "G8", "A21",
"G12", "U16", "U16",
          "C17", "U16", "U16", "C17"]])
    twister_mutant_list = []
    twister_mod_list = np.array(
        [["HOH", "Mn", "Mg", "Na", "GX145", "A29", "G40", "A10", "AP37", "A7", "U40",
"U45", "U6", "A6", "G62", "A61",
          "A63", "A39", "DU5", "G45", "OMU5", "U9", "G65"],
         ["HOH", "MT", "MT", "Na", "G48", "A30", "G48", "A1", "A1", "A1", "U44", "U44", "U-
1", "A1", "G48", "A47",
          "A41", "A47", "U-1", "G48", "U-1", "U-1", "G48"]])

    hairpin_mutant_list = ["G8A", "N6G8", "I8", "A38G"]
    hairpin_mod_list = np.array(
        [["HOH", "Mn", "Mg", "Na", "A8", "G38", "A57", "C15", "G12", "N6G38", "A39", "I12",
"3AD5", "P5P38", "MTU8"],
         ["HOH", "MT", "MT", "Na", "G8A", "A38G", "A38", "C8", "G8", "N6G8", "A9", "I8",
"A5", "A38P", "G8AP"]])

    pistol_mutant_list = ["A32G"]
    pistol_mod_list = np.array([["HOH", "MN", "Mg", "Na", "DG53", "U11", "G10", "NCO",
"G32"],
                               ["HOH", "MT", "MT", "Na", "G53", "U54", "G53", "NCO", "A32G"]])

    hdv_mutant_list = ["C75U"]
    hdv_mod_list = np.array([["HOH", "Mn", "Mg", "Na", "U163"], ["HOH", "MT", "MT", "Na",
"C75U"]])

    vs_mutant_list = ["A756G", "G638A"]
    vs_mod_list = np.array([["HOH", "Mn", "Mg", "Na", "G756", "A638", "G751", "A751"],
                           ["HOH", "MT", "MT", "Na", "A756G", "G638A", "A756", "A756"]])

    twi_sis_mutant_list = []
    twi_sis_mod_list = np.array([["HOH", "MN", "Mg", "Na"], ["HOH", "MT", "MT", "Na"]])

    if r_b_z == "glmS":
        mod_list = glms_mod_list
        mutant_list = glms_mutant_list
    if r_b_z == "HH":
        mod_list = hh_mod_list
        mutant_list = hh_mutant_list
    if r_b_z == "Twister":
        mod_list = twister_mod_list
        mutant_list = twister_mutant_list
    if r_b_z == "Hairpin":
        mod_list = hairpin_mod_list
        mutant_list = hairpin_mutant_list
    if r_b_z == "Pistol":
        mod_list = pistol_mod_list
        mutant_list = pistol_mutant_list
    if r_b_z == "HDV":
        mod_list = hdv_mod_list
        mutant_list = hdv_mutant_list
    if r_b_z == "VS":
        mod_list = vs_mod_list
        mutant_list = vs_mutant_list
    if r_b_z == "Twi_sis":
        mod_list = twi_sis_mod_list
        mutant_list = twi_sis_mutant_list
    if r_b_z == "UND":
        mod_list = []
        mutant_list = []

    return r_b_z, mod_list, mutant_list


def mut_to_WT(resn):
    # This function accepts a residue name as an argument and converts mutant residue names to
wild type residue names
    change_log = [["G8A", "N6G8", "I12", "A38G", "G12A", "G33A", "Glc6P", "G638A",
"C75U"],
                  ["G8", "G8", "G8", "G8", "G12", "G33", "GlcN6P", "G638", "C75"]]
    for x in range(len(change_log[0])):
        if change_log[0][x] == resn:
            return change_log[1][x]
    return ""


def ArrayStats(atom, resn, dist, bfact, angle_one, RBZStats, contacts_to_parent_mol, modlist,
active_site):
    # This function serves to add entries to the list of contacts (RBZStats) with sums of distances,
angles,
    # b-factors, and number of times the contact was observed (occurrence).
    # It then outputs RBZStats, as well as contacts_to_parent_mol, which is the same as RBZStats
but it consists
    # solely of contacts made to the -1/1 bases.

    # Find the residue number of the incoming contact atom
    resi = extract_resi_number(resn)

    # Define the height of temp_array
    row_height = 9

    # Make array of proper size to hold data for RBZStats
    # Later on this will be used to add the contact row to the total for that contact
    temp_array = np.chararray(row_height, itemsize=10)
    temp_array[:] = ""
    temp_array[0] = atom
    temp_array[1] = resn
    temp_array[2] = angle_one
    temp_array[3] = dist
    temp_array[4] = bfact
    temp_array[5] = '1'  # Occurrence angle_one
    temp_array[6] = '1'  # StdDev dist
    temp_array[7] = '1'  # StdDev bfact
    temp_array[8] = '1'  # Times found
    temp_array = np.vstack(temp_array)

    # Write is a variable to let the loops below communicate on whether or not to write
    write = True

    # If the aoi is the incoming atom, do not add its data to the arrays
    if "AOI" in atom:
        return RBZStats, contacts_to_parent_mol

    # If a given contact is found in RBZStats and the contact is not to the -1/1 bases, add its data to
RBZStats
    for n in range(len(RBZStats[0])):
        if atom == RBZStats[0][n] and resn == RBZStats[1][n] and resi not in active_site and write:
            RBZStats[2][n] = str(float(RBZStats[2][n]) + float(angle_one))
            RBZStats[3][n] = str(float(RBZStats[3][n]) + float(dist))
            RBZStats[4][n] = str(float(RBZStats[4][n]) + float(bfact))
            RBZStats[8][n] = str(int(RBZStats[8][n]) + 1)
            if float(angle_one) > 0:
                RBZStats[5][n] = str(int(RBZStats[5][n]) + 1)
            write = False

    # If a given contact is found in contacts_to_parent_mol and the contact is made to either the -1
or 1 base, add
    # its data to contacts_to_parent_mol
    for n in range(len(contacts_to_parent_mol[0])):
        if atom == contacts_to_parent_mol[0][n] and resn == contacts_to_parent_mol[1][n] \
                and resi in active_site and write:
            contacts_to_parent_mol[2][n] = str(float(contacts_to_parent_mol[2][n]) +
float(angle_one))
            contacts_to_parent_mol[3][n] = str(float(contacts_to_parent_mol[3][n]) + float(dist))
            contacts_to_parent_mol[4][n] = str(float(contacts_to_parent_mol[4][n]) + float(bfact))
            contacts_to_parent_mol[8][n] = str(int(contacts_to_parent_mol[8][n]) + 1)
            if float(angle_one) > 0:
                contacts_to_parent_mol[5][n] = str(int(contacts_to_parent_mol[5][n]) + 1)
            write = False
    # If the contact hasn't been found, add it to the appropriate list
    if write and resi not in active_site:
        RBZStats = np.append(RBZStats, temp_array, 1)
        write = False
    if write and resi in active_site:
        contacts_to_parent_mol = np.append(contacts_to_parent_mol, temp_array, 1)
        write = False
    return RBZStats, contacts_to_parent_mol


def txt_to_graph(new_dir):
    # This function serves to identify relevant contacts to a given aoi in a given rbz. Both the aoi
and ribozyme
    # are specified by the new directory argument (new_dir).

    # Change directory to the new directory
    os.chdir(new_dir)

    # split start should be
    #    8 for files made after July 2016
    #    7 for files treated with DuplicateHandler
    #    9 for files from July 2016 and June 2017
    splitStart = 9
    outcount = 0
    rowHeight = 9
    minfound = 0.25
    step_num = 7
    Top = 15  # Variable to tell the program how many contacts to sort
    size = 500
    size_ave = 4 * size
    yticks = np.linspace(2, 5.5, 8)

    # Verts are points to be followed when creating bars for graphs
    verts = [
        (0., 0.),
        (0., 0.5),
        (6, 0.5),
        (6, 0.),
        (0., 0.),
    ]

    verts_ave = [
        (15.0, 0.0),
        (30.0, 0.0),
        (30, 0.5),
        (15, 0.5),
        (0., 0.5),
        (0.0, 0.0),
        (15.0, 0.0)
    ]

    verts_mutant = [
        (0., 0.),
        (0., 1.0),
        (4, 1.0),
        (4, 0.),
        (0., 0.),
    ]

    # Codes are instructions for how to traverse the verts
    codes = [Path.MOVETO,
             Path.LINETO,
             Path.LINETO,
             Path.LINETO,
             Path.CLOSEPOLY,
             ]

    codes_ave = [Path.MOVETO,
                 Path.LINETO,
                 Path.LINETO,
                 Path.LINETO,
                 Path.LINETO,
                 Path.LINETO,
                 Path.CLOSEPOLY,
                 ]

    # Create a path from the codes and vertices defined above
    # The path will be used to make a marker for the graph
    path = Path(verts, codes)
    path_ave = Path(verts_ave, codes_ave)
    path_mutant = Path(verts_mutant, codes)

    # The list of all the row labels. These are used in the text file
    array_descriptor = np.chararray(rowHeight, itemsize=20)
    array_descriptor[0] = "Atom        :"
    array_descriptor[1] = "Resn        :"
    array_descriptor[2] = "Ave Angle 1 :"
    array_descriptor[3] = "Ave dist    :"
    array_descriptor[4] = "Ave bfact   :"
    array_descriptor[5] = "Occur Ang 1 :"
    array_descriptor[6] = "StdDev Dist :"
```

```
    array_descriptor[7] = "StdDev Bfact :"
    array_descriptor[8] = "Times found :"

    sugar_list = np.array(["C1'", "C2'", "O2'", "C3'", "O3'", "O4'", "C4'", "C5'", "OP1", "OP2"])

    rbz_data, contacts_to_parent_mol = np.chararray((rowHeight, 1), itemsize=5),
np.chararray((rowHeight, 1),
                                                                         itemsize=5)
    rbz_data[:], contacts_to_parent_mol[:] = "", ""

    rbz_name, modify_list, mutant_list = PDBtoRBZ()
    non_cat_list = ["1MME", "299D", "300D", "301D", "359D", "4QJD", "1NYI", "1Q29",
"1HMH"]
    vanadate_rbz_list = ["5EAO", "5EAQ", "2P7E", "1M5O", "3B58", "3B5F", "3B91", "3BBK",
"3I2R", "3I2U", "2P7F",
                         "3B4B", "3B4C", "3CQS", "3GS8", "TS"]
    unmod_aoi_list = ["O2'", "OP1", "OP2", "O5'"]

    pdb_list = ['2GCS', '2H0S', '2H0Z', '2HO7', '2H0X', '2NZ4', '3G8S', '3G8T', '3L3C', '3G96',
'2Z75', '2Z74', '3B4A',
                '3B4B', '3B4C', '4MEH', '4MEG', '2GCV', '2H0W', '2HO6', '3G9C', '2OEU', '2QUS',
'2QUW', '3ZD4', '3ZD5',
                '3ZP8', '5DI2', '5DI4', '5DQK', '5DH6', '5DH7', '5DH8', '379D', '1NYI', '1HMH',
'1Q29', '1RMN', '5EAO',
                '5EAQ', '1MME', '299D', '300D', '301D', '359D', '4RGE', '4RGF', '4OJI', '4QJD',
'4QJH', '5DUN',
                'Triple_RS_Active', 'Triple_RS_Neutral', 'Triple_TS', '3GS1', '3GS5', '3GS8', '4G6P',
'4G6R', '4G6S',
                '2BCY', '2BCZ', '3CQS', '2D2K', '1X9K', '1X9C', '2D2L', '2OUE', '1ZFT', '2FGP',
'1ZFV', '1ZFX', '3B5A',
                '3B58', '3B5F', '3B5S', '3B91', '3BBK', '3BBM', '3CR1', '3I2Q', '3I2R', '3I2S', '2NPY',
'2NPZ', '1M5K',
                '1M5O', '3BBI', '3I2U', '2P7D', '2P7E', '2P7F', '1M5V', '1M5P', '5K7C', '5K7D', '5K7E',
'5KTJ']

    res_list = ['2.1', '2.35', '2.7', '2.9', '2.3', '2.5', '3.1', '3', '2.85', '3.01', '1.7', '2.2', '2.7', '2.7', '3',
                '3.12', '3.1', '2.1', '2.4', '2.8', '2.9', '2', '2.4', '2.2', '2.2', '2.2', '1.55', '2.99', '2.95',
                '2.71', '2.78', '3.06', '3.3', '3.1', '2.85', '2.6', '3', 'N/A', '2.99', '3.2', '3.1', '3', '3', '3',
                '2.9', '2.89', '3.2', '2.3', '3.1', '3.88', '2.64', 'n/a', 'n/a', 'n/a', '2.85', '2.75', '2.85', '2.64',
                '2.83', '2.84', '2.7', '2.4', '2.8', '2.65', '3.17', '2.19', '2.5', '2.05', '2.33', '2.4', '2.4',
                '2.38', '2.35', '2.65', '2.7', '2.25', '2.75', '2.75', '2.65', '2.25', '2.9', '2.8', '2.75', '2.65',
                '3.35', '2.4', '2.2', '2.35', '2.8', '2.25', '2.05', '2.35', '2.4', '2.6', '2.73', '2.68', '3.27',
                '2.97']

    # Initialize aoi_list. This will hold the list of aoi's in the CWD and will later be used to assign
a filename
```

```
# for the graph
aoi_list = []

# Open every file in the current directory to extract data
for file_name in os.listdir(os.getcwd()):
    active_site_resi_num = []
    outcount += 1
    pdb = file_name[:4].upper()

    if file_name == "Search.txt" or ".txt" not in file_name or pdb in non_cat_list:
        outcount -= 1
        continue

    # Split up the filename to extract the atom of interest
    file_name_split = file_name.split('_')
    if file_name_split[1] in unmod_aoi_list:
        aoi_list.append(file_name_split[1])
    elif rbz_name == "Twister":
        aoi = [x for x in unmod_aoi_list if x in file_name]
        aoi_list.append(aoi[0])
    elif rbz_name == "VS":
        aoi_list.append(file_name_split[2])
    else:
        aoi_list.append(file_name_split[1])

    with open(file_name) as File:
        raw = File.read()
        split = raw.split()

        for n in range(splitStart, len(split), step_num):

            # Ignore contacts greater than 5A away
            if float(split[n + 3]) > 5.0:
                continue

            # Change name of resn to account for numbering discrepancies
            for x in range(len(modify_list[0])):
                if modify_list[0][x] in split[n + 1]:
                    split[n + 1] = modify_list[1][x]
                    break
                for c in ["MN", "MG", "TB", "CA", "Co"]:
                    if c in split[n + 2]:
                        split[n + 2] = "MT"
                        split[n + 1] = "MT"
                        break
            # Rename VS according to its special way (Add 5 to the residue number if it's >= 676
                if "VS" in rbz:
                    resi_num = extract_resi_number(split[n + 1])
                    if resi_num >= 676:
                        new_resi_num = resi_num + 5
                        split[n + 1].replace(str(resi_num), str(new_resi_num))
                if "AOI" in split[n + 2] and len(active_site_resi_num) < 1:
                    atom_name_stripped = split[n + 2].replace("AOI", "")
                    atom_name_stripped = atom_name_stripped.replace("_", "")

                    if atom_name_stripped in ["O2'", "N", "O01"]:
                        active_site_resi_num.append(extract_resi_number(split[n + 1]))  # -1 base
                        active_site_resi_num.append(active_site_resi_num[0] + 1)  # +1 base
                        continue

                    else:
                        active_site_resi_num.append(extract_resi_number(split[n + 1]))  # -1 base
                        active_site_resi_num.append(active_site_resi_num[0] - 1)  # +1
                        continue
            for n in range(splitStart, len(split), step_num):
                rbz_data, contacts_to_parent_mol = ArrayStats(split[n + 2], split[n + 1], split[n + 3],
                                                              split[n + 5], split[n + 4], rbz_data,
                                                              contacts_to_parent_mol, modify_list,
active_site_resi_num)
    # Average the angles, distances and b-factors
    for n in range(2, 5):
        for t in range(1, len(rbz_data[0])):
            rbz_data[n][t] = str(float(rbz_data[n][t]) / float(rbz_data[8][t]))

    for n in range(3, 5):
        for t in range(1, len(contacts_to_parent_mol[0])):
            contacts_to_parent_mol[n][t] = str(
                float(contacts_to_parent_mol[n][t]) / float(contacts_to_parent_mol[8][t]))
            if n > 3:
                if float(contacts_to_parent_mol[5][t]) > 1.0:
                    contacts_to_parent_mol[2][t] = str(
                        float(contacts_to_parent_mol[2][t]) / float(contacts_to_parent_mol[5][t]))

    # findTop searches through an existing list of contacts, taking the list of contacts (like
rbz_data),
    # top positions (like top 5), minimum number of times a contact should be found (Like 25%),
and whether
    # or not the incoming list is the list of parent contacts
    def findTop(list_of_contacts, top, occ_threshold, is_parent, outer_count):
        count = 0
        list_of_positions = np.ones(top)  # list_of_positions is the position in rbz_data
        list_of_positions[:] = 0
        top_average_distances = np.ones(top)
        temp_ave = np.ones(top)
        temp_pos = np.ones(top)
        top_average_distances[:] = 5
        if is_parent:
            exclude_list = ["C1'", "C2'", "O2'", "C3'", "O3'", "O4'", "C4'", "C5'", "O5'", "P", "OP1",
"OP2", "H", "C"]
            include_list = []
        else:
            exclude_list = ["C", "H", "P"]
            include_list = ["OP1", "OP2"]

        for n in range(1, len(list_of_contacts[0])):

            # This loop serves to compare the rbz_data array and the top_average_distances array
            for c in range(top):
                # Exclude certain contacts
                panic = False
                for exclude_atom in exclude_list:
                    if str(exclude_atom) in list_of_contacts[0][n] or "NCO" in list_of_contacts[1][n] \
                            and "Co" not in list_of_contacts[0][n]:
                        panic = True
                for include_atom in include_list:
                    if str(include_atom) in list_of_contacts[0][n]:
                        panic = False
                if panic:
                    break

                # If the average distance is lower than the current value
                # AND the contact is found in 25% of the structures
                # Shift list_of_positions and top_average_distances down one

                if float(list_of_contacts[3][n]) < top_average_distances[c] and
int(list_of_contacts[8][n]) >= int(
                        occ_threshold * outer_count):
                    # Assign the placeholder
                    for t in range(0, top):
                        temp_ave[t] = top_average_distances[t]
                        temp_pos[t] = list_of_positions[t]
                    top_average_distances[c] = float(list_of_contacts[3][n])
                    list_of_positions[c] = n
                    # This loop serves to bump down all values
                    for t in range(c + 1, top):
                        top_average_distances[t] = temp_ave[t - 1]
                        list_of_positions[t] = temp_pos[t - 1]
                    break
        list_of_positions = [int(x) for x in list_of_positions if x > 1]
        return list_of_positions

    # Sort rbz_data
    parent_contact_positions = findTop(contacts_to_parent_mol, Top, minfound, True, outcount)
    # Include contacts_to_parent_mol if relevant
    for x in range(0, len(parent_contact_positions)):
        temp_array = np.chararray(rowHeight, itemsize=10)
        # temp_array[:] = ""
        if float(contacts_to_parent_mol[2][parent_contact_positions[x]]) > 140.0:
            for t in range(rowHeight):
                # Make array of proper size to hold data for rbz_data
                # Later on this will be used to add the contact row to the total for that contact
                temp_array[t] = contacts_to_parent_mol[t][parent_contact_positions[x]]
                temp_array = np.vstack(temp_array)
            rbz_data = np.append(rbz_data, temp_array, 1)

    # Sort rbz_data
    top_contact_positions = findTop(rbz_data, Top, minfound, False, outcount)

    # Create .csv file for excel/kaleidagraph
    fl = open("TextSearcher_1.7_" + str(rbz_name) + ".csv", 'wb')
    writer = csv.writer(fl)

    # Create lists for x's and y's to be plotted with matplotlib
    y_list = []
    angle_plot_list = []
    y_labels = []
    y_list_std_dev = []
    y_list_mutant = []
    y_mutant_labels = []
    non_cat_y_list = []
    non_cat_label_list = []
    marker_color_list = []

    # This section of code serves to loop through all the important contacts and find all of the
observed distances
    for t in range(0, len(top_contact_positions)):
        dist_list = np.array([])
        bfact_list = np.array([])
        count_list = np.array([])
        angle_one_list = np.array([])
        color_list = []

        # This loop opens every file in the current directory except for unrelated files
        for file_name in os.listdir(os.getcwd()):
```

```
        count = 0
        pdb = file_name[:4].upper()
        if pdb == "TRIP":
            pdb = file_name.split('_')
            pdb = pdb[1]

        # If file_name isn't one of the text files with data, skip it (continue)
        if file_name == "Search.txt" or ".txt" not in file_name or pdb in non_cat_list:
            continue

        with open(file_name) as File:
            raw = File.read()
            split = raw.split()

            # This loop serves to iterate through contacts in file_name
            for n in range(splitStart, len(split), step_num):

                # Ignore contacts greater than 5A away
                if float(split[n + 3]) > 5.0:
                    continue

                # Rename VS according to its special way (Add 5 to the residue number if it's >=
676
                if "VS" in rbz:
                    resi_num = extract_resi_number(split[n + 1])
                    if resi_num >= 676:
                        new_resi_num = resi_num + 5
                        split[n + 1].replace(str(resi_num), str(new_resi_num))

                # Split[n + 1] and split[n] are renamed to make things easier to follow
                atom = split[n + 2]
                resn = split[n + 1]

                # Instead of split[n], we need to use the condensed version so that we can find the
renamed mutants
                # and re-numbered bases
                for c in range(len(modify_list[0])):
                    if modify_list[0][c] in resn:
                        resn = modify_list[1][c]
                        break
                for c in ["MN", "MG", "CA", "TB"]:
                    if c in atom:
                        atom = "MT"
                        resn = "MT"

                # Append each observed value to the array if it is found in the rbz_data array
```

```
                    if atom == rbz_data[0][top_contact_positions[t]] and resn in
rbz_data[1][top_contact_positions[t]]:
                        angle_one = split[n + 4]
                        angle_one_list = np.append(angle_one_list, float(angle_one))
                        dist_list = np.append(dist_list, float(split[n + 3]))
                        bfact_list = np.append(bfact_list, float(split[n + 5]))
                        count += 1

                        # If the observed value is from a vanadate structure, color it magenta
                        if pdb not in vanadate_rbz_list and len(pdb) > 3:
                            color_list.append('k')
                        elif pdb in vanadate_rbz_list:
                            color_list.append('m')
                        else:
                            color_list.append('y')

                count_list = np.append(count_list, count)
            marker_color_list.append(color_list)
            y_labels.append(rbz_data[0][top_contact_positions[t]] + '\n' +
rbz_data[1][top_contact_positions[t]])
            angle_plot_list.append(list(angle_one_list))
            y_list.append(list(dist_list))
            y_list_std_dev.append([np.std(dist_list) + np.mean(dist_list), np.mean(dist_list) -
np.std(dist_list)])
            std_dev_dist = np.std(dist_list)
            std_dev_bfact = np.std(bfact_list)
            writer.writerow([])
            writer.writerow(rbz_data[0][top_contact_positions[t]] +
rbz_data[1][top_contact_positions[t]])
            writer.writerow(dist_list)
            writer.writerow(angle_one_list)
            writer.writerow(count_list)
            rbz_data[6][top_contact_positions[t]] = std_dev_dist
            rbz_data[7][top_contact_positions[t]] = std_dev_bfact

        mutant_contacts = []
        # Write mutant contacts to csv
        for n in range(len(mutant_list)):
            for t in range(0, len(rbz_data[1])):
                if mutant_list[n] in rbz_data[1][t]:
                    mutant_contacts.append(t)

        writer.writerow(["MUTANT CONTACTS"])
        for t in range(0, len(mutant_contacts)):
            # Initialize numpy arrays
            angle_list, dist_list, bfact_list, count_list = np.array([]), np.array([]), np.array([]),
np.array([])

            # This loop opens every file in the current directory except for unrelated files
            for file_name in os.listdir(os.getcwd()):
                count = 0

                # If file_name isn't one of the text files with data, skip it (continue)
                if file_name == "Search.txt" or ".txt" not in file_name:
                    continue

                with open(file_name) as File:
                    raw = File.read()
                    split = raw.split()

                    # This loop serves to iterate through contacts in file_name
                    for n in range(splitStart, len(split), step_num):
                    # Rename VS according to its special way (Add 5 to the residue number if it's >=
676
                    if "VS" in rbz:
                        resi_num = extract_resi_number(split[n + 1])
                        if resi_num >= 676:
                            new_resi_num = resi_num + 5
                            split[n + 1].replace(str(resi_num), str(new_resi_num))
                    # Instead of split[n], we need to use the condensed version so that we can find the
renamed mutants
                    # and re-numbered bases
                    atom = split[n + 2]
                    resn = split[n + 1]

                    for c in range(0, len(modify_list[0])):
                        if modify_list[0][c] in resn:
                            resn = modify_list[1][c]
                            break
                    if atom == rbz_data[0][mutant_contacts[t]] and resn ==
rbz_data[1][mutant_contacts[t]] \
                            and "C" not in atom and "H" not in atom:
                        angle_list = np.append(angle_list, float(split[n + 4]))
                        dist_list = np.append(dist_list, float(split[n + 3]))
                        bfact_list = np.append(bfact_list, float(split[n + 5]))
                        count += 1
                count_list = np.append(count_list, count)
            y_list_mutant.append(list(dist_list))
            y_mutant_labels.append(rbz_data[0][mutant_contacts[t]] + '\n' +
rbz_data[1][mutant_contacts[t]])
            std_dev_dist = np.std(dist_list)
            std_dev_bfact = np.std(bfact_list)

            writer.writerow([])
            writer.writerow(rbz_data[0][mutant_contacts[t]] + rbz_data[1][mutant_contacts[t]])
            writer.writerow(dist_list)
            writer.writerow(angle_list)
            writer.writerow(count_list)

            rbz_data[6][mutant_contacts[t]] = std_dev_dist
            rbz_data[7][mutant_contacts[t]] = std_dev_bfact

        writer.writerow(["PARENT MOL CONTACTS"])


        for t in range(len(parent_contact_positions)):
            # Initialize numpy arrays
            angle_list, dist_list, bfact_list, count_list = np.array([]), np.array([]), np.array([]),
np.array([])

            # This loop opens every file in the current directory except for unrelated files
            for file_name in os.listdir(os.getcwd()):
                count = 0

                # If file_name isn't one of the text files with data, skip it (continue)
                if file_name == "Search.txt" or ".txt" not in file_name:
                    continue

                with open(file_name) as File:
                    raw = File.read()
                    split = raw.split()

                    # This loop serves to iterate through contacts in file_name
                    for n in range(splitStart, len(split), step_num):

                    # Instead of split[n], we need to use the condensed version so that we can find the
renamed mutants
                    # and re-numbered bases
                    atom = split[n + 2]
                    resn = split[n + 1]

                    for c in range(0, len(modify_list[0])):
                        if modify_list[0][c] in resn:
                            resn = modify_list[1][c]
                            break

                    # Only search through non-cat list once
                    if t == len(parent_contact_positions) - 1:
                        # Add non-catalytic contacts to a list
                        for non_cat_pdb in non_cat_list:
                            if non_cat_pdb in file_name.upper():
                                for x in range(len(top_contact_positions)):
                                    if atom == rbz_data[0][top_contact_positions[x]] and resn in rbz_data[1][
                                        top_contact_positions[x]]:
                                        non_cat_y_list.append([split[n + 3]])
                                        non_cat_label_list.append(atom + '\n' + resn)

                    if atom == contacts_to_parent_mol[0][parent_contact_positions[t]] and resn == \
                            contacts_to_parent_mol[1][
                                parent_contact_positions[t]]:
                        angle_list = np.append(angle_list, float(split[n + 4]))
                        dist_list = np.append(dist_list, float(split[n + 3]))
                        bfact_list = np.append(bfact_list, float(split[n + 5]))
                        count += 1

                count_list = np.append(count_list, count)

            writer.writerow([])
            writer.writerow(
                contacts_to_parent_mol[0][parent_contact_positions[t]] + contacts_to_parent_mol[1][
                    parent_contact_positions[t]])
            writer.writerow(dist_list)
            writer.writerow(angle_list)
            writer.writerow(count_list)

        fl.close()

        f = open('Search.txt', 'w')
        f.write("Native Contact List" + '\n')
        for n in range(0, rowHeight):
            f.write(array_descriptor[n] + '\t')
            for t in range(0, len(top_contact_positions), 1):
                f.write(rbz_data[n][top_contact_positions[t]] + '\t')
            f.write('\n')
        f.write('\n')

        f.write("Substituted Contact List" + '\n')
        for n in range(0, rowHeight - 2):
            f.write(array_descriptor[n] + '\t')
            for t in range(0, len(mutant_contacts)):
                f.write(rbz_data[n][mutant_contacts[t]] + '\t')
            f.write('\n')
        f.write('\n')
```

```
    # Don't write any contacts that are on the sugarlist to the text file
    no_write_list = []
    for t in range(1, len(contacts_to_parent_mol[0])):
        if contacts_to_parent_mol[0][t] in sugar_list:
            no_write_list.append(t)

    f.write("Contacts to Parent Molecule" + '\n')
    for n in range(0, rowHeight):
        f.write(array_descriptor[n] + '\t')
        for t in parent_contact_positions:
            f.write(contacts_to_parent_mol[n][t] + '\t')
        f.write('\n')

    f.close()

    # ******************** Plot Data ********************* #
    # Create x-axis ticks, 5 numbers evenly spaced between 2.5 and 5
    x_list = []
    x_list_std_dev_bars = np.ones(2)
    y_list_count = 0
    plotted_mutant_labels = []
    y_list = list(filter(None, y_list))
    angle_plot_list = list(filter(None, angle_plot_list))
    # Count the number of non-zero elements in y_list, this makes it so we can plot contacts for
AOIs with <5
    # significant contacts
    for x in y_list:
        if np.count_nonzero(x) < 1:
            break
        y_list_count += 1
    if y_list_count > 5:
        y_list_count = 5
    x_axis_tick_num = np.linspace(2.5, 5, y_list_count)

    # Do not print any mutant points by themselves if the wild type is in the top 5
    y_label_len = y_list_count
    label_count = 0

    # Iterate through y_labels to see if a mutant resn is present
    while label_count < y_label_len:
        label = y_labels[label_count].split()
        if len(label) < 1:
            label_count += 1
            continue
    # Check to see if a mutant is present in the top 5
    mutant_resn = ""
    mutant_atom = ""
    for x in mutant_list:
        if label[1] == x:
            mutant_resn = x
            mutant_atom = label[0]
    # If a mutant is present, check to see if the WT exists in the top 5
    if mutant_resn:
        wt_resn = mut_to_WT(mutant_resn)
        for t in range(y_list_count):
            entry = y_labels[t].split()
            if len(entry) < 1:
                continue
            elif len(entry[0]) < 2:
                continue
            # If the WT is present, delete the mutant from the list of normal contacts
            if wt_resn == entry[1] and entry[0][-1] in mutant_atom:
                del y_list[label_count]
                del y_labels[label_count]
                del angle_plot_list[label_count]
                del y_list_std_dev[label_count]
                if y_label_len < 5:
                    y_label_len -= 1
                label_count -= 1
                break

    label_count += 1

# Create a list of average angles for contacts
# Don't average any angles that are equal to zero
new_pl = []
for x in range(len(angle_plot_list)):
    if np.mean(angle_plot_list[x]) > 0:
        test = filter(lambda a: a != 0, angle_plot_list[x])
    else:
        test = angle_plot_list[x]
    new_pl.append(test)

# Round angles and shift all the angles' x-values
ave_angle_list = [int(round(np.mean(new_pl[x]), 0)) for x in range(y_list_count)]
angle_x_values = x_axis_tick_num[:] - 0.144

# Put angles above each column
for x in range(y_list_count):
    if 140 > ave_angle_list[x] > 1:
            plt.text(angle_x_values[x], 5.2, str(ave_angle_list[x]) + '°', fontsize=12,
fontname="arial", color='r')
        elif ave_angle_list[x] >= 140:
            plt.text(angle_x_values[x], 5.2, str(ave_angle_list[x]) + '°', fontsize=12,
fontname="arial")
        else:
            plt.text(angle_x_values[x], 5.2, "N/A", fontsize=12, fontname="arial", color="#4F4747")

    # Create a list of x's for all the y's using x_axis_tick_num
    for x in range(y_list_count):
        x_list_outer = []
        for t in range(0, len(y_list[x])):
            x_list_outer.append(x_axis_tick_num[x] - 0.07754)  # The subtraction aligns the points
        x_list.append(x_list_outer)
    # Plot mutant and non-catalytic points
    for x in range(y_list_count):
        label = y_labels[x].split()
        # If a label position is empty, skip it
        if len(label) < 1:
            continue

        # Search through the list of contacts identified in non-cat structures, if they show up in the
top 5,
        # plot them with their corresponding catalytic contact
        for t in range(len(non_cat_label_list)):
            non_cat_label = non_cat_label_list[t].split()
            if len(non_cat_label) < 1:
                continue

            if label[1] == non_cat_label[1] and label[0] == non_cat_label[0]:
                x_list_non_cat = [x_axis_tick_num[x] - 0.077546 for x_tick in non_cat_y_list[t]]
                plt.scatter(x_list_non_cat, non_cat_y_list[t], marker=path_mutant, s=size,
color='#FF4500')
        for t in range(len(y_mutant_labels)):
            mutant_label = y_mutant_labels[t].split()
            if len(mutant_label) < 1:
                continue

            mutant_label_except = [["GlcN6P", "G8"], ["Glc6P", "N6G8"]]
            # Don't plot a mutant's average and standard deviation bars if the WT is in the top 5
            plot_alone = False
            # If the mutant is in the top 5 proceed further
            if label[0][-1] in mutant_label[0][-1] and label[1] in mutant_label[1] or label[1] in
mutant_label_except[
                0] and mutant_label[1] in mutant_label_except[1] and label[0][1] in
mutant_label[0][1]:
```

```
            # If the WT is also in the top 5, don't plot contact by itself
            for c in range(y_list_count):
                label_strip = y_labels[c].split()
                WT = mut_to_WT(mutant_label[1])
                # if WT in label_strip[1] and label_strip[0][-1] in mutant_label[0][-1]
                if mutant_label == label_strip:
                    plot_alone = True

            x_list_mutant = [x_axis_tick_num[x] - 0.077546 for x_tick in y_list_mutant[t]]
            plt.scatter(x_list_mutant, y_list_mutant[t], marker=path_mutant, s=size, color='g')

            for counter in range(0, len(x_list_mutant)):
                plt.text(float(x_list_mutant[counter] + 0.2), float(y_list_mutant[t][counter]),
                         str('<' + mutant_label[1]), fontsize=8, fontname="arial")
            plotted_mutant_labels.append(mutant_label[0][-1] + mutant_label[1])

            if plot_alone:
                mean = np.mean(y_list_mutant[t])
                mutant_y_std_dev = [mean + np.std(y_list_mutant[t]), mean -
np.std(y_list_mutant[t])]
                x_list_std_dev_bars[:] = x_list[x][0] + 0.032872
                plt.plot([x_list_std_dev_bars[0] + 0.044672, x_list_std_dev_bars[1] + 0.044672],
                         mutant_y_std_dev, color='0.4')
                plt.scatter(x_list[x][0] - 0.07754, mean, marker=path_ave, s=size_ave, color='r')
                plt.scatter(x_list_std_dev_bars, mutant_y_std_dev, marker=path, s=size / 3,
color='0.4')

    # Print Hammerhead's +1 base with its correct number
    for x in range(y_list_count):
        if "U16" in y_labels[x]:
            y_labels[x].replace("U16", "U1.1")

    # Plot points
    for t in range(y_list_count):

        # Don't plot a mutant by itself if it's in the top 5
        label_strip = y_labels[t].replace("\n", "")
        label_strip = label_strip[1:]
        if label_strip in plotted_mutant_labels:
            continue
        x_list_std_dev_bars[:] = x_list[t][0] + 0.032872
        plt.scatter(x_list[t], y_list[t], marker=path, s=size, color=marker_color_list[t])
        plt.scatter(x_list[t][0] - 0.07754, np.mean(y_list[t]), marker=path_ave, s=size_ave, color='r')
        plt.scatter(x_list_std_dev_bars, y_list_std_dev[t], marker=path, s=size / 3, color='0.4')
        plt.plot([x_list_std_dev_bars[0] + 0.044672, x_list_std_dev_bars[1] + 0.044672],
                [y_list_std_dev[t][0], y_list_std_dev[t][1]], color='0.4')

    # Set background
    img = imread(
        "C:\\some directory\\image.png")
    plt.imshow(img, extent=[2, 5.5, 2, 5.5])

    # Label axes
    plt.xlabel('Contact Atoms', fontsize=14, fontname="arial", fontweight="bold")
    plt.xticks(x_axis_tick_num, y_labels, fontsize=12, fontname="arial")
    plt.ylabel('Distance (Å)', fontsize=14, fontname="arial", fontweight="bold")
    plt.yticks(yticks, fontsize=12, fontname="arial")

    # Set the bottom of the graph to be slightly higher so the x-axis label doesn't get cutoff
    plt.gcf().subplots_adjust(bottom=0.15)
    # plt.show()

    # Check for uniformity in the aoi's
    uniform = all(x == aoi_list[0] for x in aoi_list)
    if not uniform:
        if "O2'" in aoi_list:
            aoi = "O2'"
        elif "O5'" in aoi_list:
            aoi = "O5'"
        elif "OP1" or "OP2" in aoi_list:
            op1_count = 0
            op2_count = 0
            for NBO in aoi_list:
            if NBO == "OP1":
                op1_count += 1
            if NBO == "OP2":
                op2_count += 1
            if op1_count > op2_count:
            aoi = "OP1"
            else:
            aoi = "OP2"
        else:
            aoi = "Unidentified aoi"
            print(aoi_list)
    else:
        aoi = aoi_list[0]

    # Change directory to figures/graphs to change the save location
    init_cwd = os.getcwd()
    os.chdir(
        "C:\\some directory\\" + aoi)
    plt.savefig(rbz_name + ' ' + aoi + " Contacts" + ".pdf")
    plt.savefig(rbz_name + ' ' + aoi + " Contacts" + ".png", dpi=200)
    plt.clf()
    plt.close()
    os.chdir(init_cwd)


# rbz_list is a list of all the ribozymes that we want to collect data on
rbz_list = ["glmS", "Hammerhead", "Hairpin", "Twister", "Pistol", "VS"]
default_dir = "C:\\some directory"

for rbz in rbz_list:
    txt_to_graph(default_dir + '\\' + rbz + '\\' + "NBO" + '\\' + "OP1")
    txt_to_graph(default_dir + '\\' + rbz + '\\' + "NBO" + '\\' + "OP2")
    txt_to_graph(default_dir + '\\' + rbz + '\\' + "O5'")
    if rbz == "Twister Sister":
        continue
    txt_to_graph(default_dir + '\\' + rbz + '\\' + "O2'")

print("DONE" + '\n')
```

V. All Phosphate Downstream Processing Script

```
# -*- coding: utf-8 -*-
# This script calculate stats on the crystal structures
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import sys

mpl.rc('font', family='arial')


def find_sum(contact_list):
    hbond_sum = np.zeros(5)

    for n in contact_list:
        for z in range(5):
            if n == z or n >= z == 4:
                hbond_sum[z] += 1

    return hbond_sum


def find_neighbor(resn, direction, pdb):
    # This function accepts a residue (Ex: 'G33') as an argument and will return its
    # residue number ('33') as the output

    neighbor = ''
    numbers = ['-', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
    # Some hairpin residue numbers have letters in them, such as those found in 3G8S
    resi_letters = "ABCDEFGHIJKL"
    last_letter = -1

    for n in range(len(resn) - 2, -1, -1):
        # Bypasses residue numbers that end with a letter
        if resn[n] not in numbers:
            last_letter = n
            break
    if last_letter == -1:
        sys.exit("An error occurred in the find_neighbor method for " + resn)

    resi = resn[(last_letter + 1):]

    if resi[-1] not in numbers:
        if resi[-1] == 'A' and direction == 'minus':
            neighbor = str(int(resi[:-1]) - 1)
        elif resi[-1] == 'L' and direction == 'plus':
            neighbor = str(int(resi[:-1]) + 1)
        else:
            for x in range(len(resi_letters)):
                if resi[-1] == resi_letters[x]:
                    if direction == 'minus':
                        neighbor = resi[:-1] + resi_letters[x - 1]
                    if direction == 'plus':
                        neighbor = resi[:-1] + resi_letters[x + 1]
    elif direction == 'minus':
        neighbor = str(int(resi) - 1)
    elif direction == 'plus':
        neighbor = str(int(resi) + 1)

    # This is a list of PDBID's which skips residue 0
    pdb__exception__list = ['3B4A', '3B4B', '3B4C', '2Z75', '2Z74', '2H0S', '4G6P', '3GS1', '3I2Q',
'3I2S', '2OUE']

    if pdb in pdb__exception__list and neighbor == "0":
        if direction == 'minus':
            neighbor = "-1"
        else:
            neighbor = "1"

    return neighbor


def nbo_contact(split, splitStart, step_num, scissile, dist_thresh, pdb):

    exclude_list = ["C1'", "C2'", "C3'", "O3'", "O4'", "C4'", "C5'", "O5'", "P", "OP1", "OP2", "H",
"C", "O"]
    include_list = ["CA", "CO"]
    # This array holds the chains of interest for crystal structures that contain multiple copies of
the ribozyme
    # and/or one or more polypeptides
    copies_list = [['3G8S', 'EP'], ['3G8T', 'EP'], ['3L3C', 'EP'], ['3G96', 'EP'], ['2NZ4', 'FQ'],
['2QUS', 'A'],
                   ['4RGE', 'A'], ['4RGF', 'A'], ['4QJH', 'AB'], ['1M5K', 'AB'], ['1M5O', 'AB'], ['5KTJ',
'AB']]
    ns_cont_list = []
    s_count_1 = 0
    s_count_2 = 0
    s_count = 0
    current_aoi = ""
    nbo_num = "OP1"
    if scissile:
        for n in range(splitStart, len(split), step_num):
            if n == splitStart:
                nbo_num = split[n + 1]
                if '1' in split[n + 1]:
                    partner = split[n + 1].replace('1', '2')
                else:
                    partner = split[n + 1].replace('2', '1')
            # do not consider atoms that belong to neighboring ribozyme copies or polypeptides to be
contact
            # atoms
            chain_of_interest = True
            for j in range(len(copies_list)):
                if pdb == copies_list[j][0] and split[n + 5] not in copies_list[j][1]:
                    chain_of_interest = False
            minus_1_base = find_neighbor(split[n], 'minus', pdb)
            if split[n + 3] not in exclude_list and dist_thresh > float(split[n + 6]) > 0 and split[n + 4]
!= split[n] \
                    and "H" not in split[n + 3] and ("C" not in split[n + 3] or split[n + 3][:2] in
include_list) \
                    and minus_1_base not in split[n + 4] and chain_of_interest:
                if split[n + 1] == nbo_num and s_count_1 < 2:
                    s_count_1 += 1
                elif split[n + 1] == partner and s_count_2 < 2:
                    s_count_2 += 1

        s_count = s_count_1 + s_count_2

    if not scissile:

        first_iteration = True
        aoi_resi_number_from_zero = 0

        for n in range(splitStart, len(split), step_num):
            minus_1_base = find_neighbor(split[n], 'minus', pdb)

            # If this is the first time running the loop/current_aoi is empty
            if not current_aoi or n == splitStart:
                current_aoi = split[n]
                nbo_num = split[n + 1]
                if '1' in split[n + 1]:
                    partner = split[n + 1].replace('1', '2')
                else:
                    partner = split[n + 1].replace('2', '1')
            # If this is the first time seeing the current aoi
            if first_iteration:
                ns_cont_list.append(0)
                first_iteration = False

            # If the current aoi lines up with the aoi of the current row, parse it
            if nbo_num == split[n + 1]:
                # do not consider atoms that belong to neighboring ribozyme copies or polypeptides to
be contact
                # atoms
                chain_of_interest = True
                for j in range(len(copies_list)):
                    if pdb == copies_list[j][0] and split[n + 5] not in copies_list[j][1]:
                        chain_of_interest = False
                # Add its data to _cont_list if it's not in exclude list, its distance is eligible and it's not a
                # contact to its own nucleobase
                if split[n + 3] not in exclude_list and dist_thresh > float(split[n + 6]) > 0 and split[n +
4] != \
                        split[n] and "2" not in split[n + 1] and "H" not in split[n + 3] and ("C" not in
split[n + 3] or
                        split[n + 3][:2] in include_list) and minus_1_base not in split[n + 4] and
chain_of_interest:
                    if ns_cont_list[int(aoi_resi_number_from_zero)] < 2:
                        ns_cont_list[aoi_resi_number_from_zero] += 1

            if n + step_num >= len(split):
                # Pinch off list of data
                ns_cont_list = ns_cont_list[:-1]
                break

            if split[n + step_num] != current_aoi and nbo_num == split[n + 1]:
                # Each aoi can have a max of 2 contacts
                first_iter = True
                initial = 2
                # Search for the other NBO
                for x in range(splitStart, len(split), step_num):
                    if partner == split[x + 1] and current_aoi == split[x]:
                        # Do not consider atoms that belong to neighboring ribozyme copies or
polypeptides to be contact
                        # atoms
                        chain_of_interest = True
                        for j in range(len(copies_list)):
                            if pdb == copies_list[j][0] and split[n + 5] not in copies_list[j][1]:
                                chain_of_interest = False
                        if split[x + 3] not in exclude_list and dist_thresh > float(split[x + 6]) > 0 \
                                and split[x + 4] != split[x] and "H" not in split[x + 3] and (
                    "C" not in split[x + 3] or split[x + 3][:2] in include_list) and minus_1_base
not in \
                    split[x + 4] and chain_of_interest:
                  # Go find the correct resn in resn list
                    if first_iter and ns_cont_list[int(aoi_resi_number_from_zero)] > 2:
                      sys.exit("There are more than 2 OP1 contacts before OP2 contacts were
added.")
                    if first_iter:
                      initial = ns_cont_list[aoi_resi_number_from_zero]
                      first_iter = False
                    if ns_cont_list[int(aoi_resi_number_from_zero)] < 2 + initial:
                      ns_cont_list[aoi_resi_number_from_zero] += 1

            aoi_resi_number_from_zero += 1
            ns_cont_list.append(0)

          current_aoi = split[n + step_num]

      if s_count > 4:
        sys.exit("There are more than 4 contacts to the NBOs.")
      for n in range(len(ns_cont_list)):
        if ns_cont_list[n] > 4:
          sys.exit("There are more than 4 contacts to the NBOs.")
      return s_count, ns_cont_list


def sort_data(new_dir, aoi):
      os.chdir(new_dir)
      splitStart = 13
      step_num = 7
      non_cat_list = ["1MME", "299D", "300D", "301D", "359D", "4QJD", "1NYI", "1HMH"]
      exclude_list = ["C1'", "C2'", "C3'", "O3'", "O4'", "C4'", "C5'", "O5'", "P", "OP1", "OP2", "H",
"C", "O"]
      include_list = ['CA', 'CO']
      # This array holds the chains of interest for crystal structures that contain multiple copies of
the ribozyme
      # and/or one or more polypeptides
      copies_list = [['3G8S', 'EP'], ['3G8T', 'EP'], ['3L3C', 'EP'], ['3G96', 'EP'], ['2NZ4', 'FQ'],
['2QUS', 'A'],
          ['4RGE', 'A'], ['4RGF', 'A'], ['4QJH', 'AB'], ['1M5K', 'AB'], ['1M5O', 'AB'], ['5KTJ',
'AB']]
      ns_list = []
      dist_thresh = 5.0
      s_list = []

      for file_name in os.listdir(os.getcwd()):
        # If file_name isn't one of the text files with data, skip it (continue)
        pdb = file_name[:4]
        pdb = pdb.upper()
        if file_name == "Search.txt" or ".txt" not in file_name or pdb in non_cat_list:
            continue

        with open(file_name) as File:
            raw = File.read()
            split = raw.split()

            if "Non-Scissile" in file_name:
                scissile = False
            else:
                scissile = True

            # This loop serves to iterate through contacts in file_name
            current_contact = ""
            s_count = 0
            ns_count = 0
            if aoi[0] == "NBO":
                scis, non_scis = nbo_contact(split, splitStart, step_num, scissile, dist_thresh, pdb)
                if scissile:
                    s_list.append(scis)
                else:
                    for x in non_scis:
                        ns_list.append(x)

            else:
                for n in range(splitStart, len(split), step_num):
                    # If this is the first time running the loop/current_contact is empty
                    if not current_contact or n == splitStart:
                        current_contact = split[n]
                    if aoi[0] == "O5'":
                        adjacent_nb = find_neighbor(split[n], 'minus', pdb)
                    else:
                        adjacent_nb = find_neighbor(split[n], 'plus', pdb)

                    if split[n] == current_contact:
                        # Do not consider atoms that belong to neighboring ribozyme copies or
polypeptides to be contact
                        # atoms
                        chain_of_interest = True
                        for j in range(len(copies_list)):
                            if pdb == copies_list[j][0] and split[n + 5] not in copies_list[j][1]:
                                chain_of_interest = False
                        # Add its data to a list
                        if split[n + 3] not in exclude_list and dist_thresh > float(split[n + 6]) > 0 and split[
                                n + 4] != split[n] and "H" not in split[n + 3] and (
                                "C" not in split[n + 3] or split[n + 3][:2] in include_list) and adjacent_nb not
in \
                                split[n + 4] and chain_of_interest:
                            if scissile:
                                s_count += 1
                            elif not scissile:
                                ns_count += 1

                    if n + step_num >= len(split):
                        # Pinch off list of data
                        if not scissile:
                            ns_list.append(ns_count)
                        break
                    elif split[n + step_num] != current_contact:
                        # Pinch off list of data
                        if not scissile:
                            ns_list.append(ns_count)
                        ns_count = 0
                        current_contact = split[n + step_num]

                if scissile:
                    s_list.append(s_count)

            non_scissile_sum = find_sum(ns_list)
            scissile_sum = find_sum(s_list)

            # Normalize data
            s_out = scissile_sum
            ns_out = non_scissile_sum
            scissile_sum = [x * 100.0 / np.sum(scissile_sum) for x in scissile_sum]
            non_scissile_sum = [x * 100.0 / np.sum(non_scissile_sum) for x in non_scissile_sum]
            return non_scissile_sum, scissile_sum, s_list, ns_list, s_out, ns_out


default_dir = "C:\\Directory"
aoi = ["NBO"]
rbz_list = ["glmS", "Hammerhead", "Twister", "Hairpin"]
scissile_data, non_scissile_data, all_scis, all_non_scis = [], [], [], []
init_cwd = os.getcwd()
os.chdir("C:\\Directory")
for rbz in rbz_list:
            os.chdir(init_cwd)
            nsd, sd, s_total, ns_total, s_out, ns_out = sort_data(default_dir + "\\" + rbz + "\\" + aoi[0], aoi)
            os.chdir("C:\\Directory")
    scissile_data.append(sd)
    non_scissile_data.append(nsd)
    all_scis.append(s_total)
    all_non_scis.append(ns_total)

    init_cwd = os.getcwd()
os.chdir(init_cwd)

init_cwd = os.getcwd()

# Define variables for graphs and get fig, ax objects
width = 0.2
ind = range(5)
ind = [x - width for x in ind]
y_ticks = np.arange(0, 120, 20)
colors = ["#EC60A3", "#F39940", "#008AC9", "#00B79B"]
if "Pistol" in rbz_list:
    colors = ["#783493"]
if "VS" in rbz_list:
    colors = ["#E9302E"]
patch_list = []
fig, ax = plt.subplots()

# Stop autoscaling
ax.autoscale(True, axis='x')
ax.autoscale(False, axis='y')
t = 0

print(scissile_data)
shift = [[0] * 5 for x in range(4)]
# Shift x-values to account for bars with zero height
scissile = True
if scissile:
    for c in range(len(scissile_data[0])):
        zero_count = 0
        for x in range(len(scissile_data)):
            if scissile_data[x][c] == 0:
                zero_count -= 1
            else:
                shift[x][c] = zero_count * width
else:
    for c in range(len(non_scissile_data[0])):
        zero_count = 0
        for x in range(len(non_scissile_data)):
            if non_scissile_data[x][c] == 0:
                zero_count -= 1
            else:
                shift[x][c] = zero_count * width

# Plot all data
for x in range(len(rbz_list)):
    ind = [c + width for c in ind]
    # Account for zero height bars
    ind2 = [t + c for t, c in zip(ind, shift[x])]
    if not scissile:
        non_scissile_bar = ax.bar(ind2, non_scissile_data[x], width, color='none', label=rbz_list[x],
                                  edgecolor=colors[x],
                                  linewidth=2.2)
    else:
        scissile_bar = ax.bar(ind2, scissile_data[x], width, color=colors[x], label=rbz_list[x])

# Create a legend
plt.legend(frameon=False, fontsize=12)

# Get rid of x-axis ticks but not labels
plt.tick_params(
    axis='x',
    which='both',
    bottom='off',
    top='off',
    labelbottom='on'
)

plt.yticks(y_ticks, fontsize=14)
plt.xticks(range(5), fontsize=14)
ax.set_ylabel("Percentage of " + aoi[0] + "s", fontsize=14)
ax.set_xlabel("Number of Contacts", fontsize=14)
os.chdir(
    "C:\\Directory")
if scissile:
    plt.savefig("Scissile " + aoi[0] + ".pdf")
else:
    plt.savefig("Non-Scissile " + aoi[0] + ".pdf")
plt.show()

print("DONE")
```