'''
```
#########################################################################
#                                                       #
# HwRMSD README                                                 #
#                                                       #
# email: Nickolay (@umich.edu) Khazanov                         #
# authors: Kelly Damm-Ganamet, Nickolay Khazanov, Daniel Quang       #
# University of Michigan                                        #
# Carlson Lab                                                   #
# Last modified: January 2011                                   #
#                                                       #
# HwRMSD bundles the wRMSD superposition method with 'needle'        #
# EMBOSS sequence alignment tool and SE structural alignment tool.   #
# The default implementation consists of the following steps:        #
#                                                       #
# Step I:  Initial sequence alignment (needle)                  #
# Step II: Standard superposition (sRMSD)                       #
# Step III:Weighted superposition (wRMSD)                       #
# Step IV: Structural alignment (SE)                    #
# Step V:  Local version of weighed superposition               #
#########################################################################

Required Software:

- SE Seed Extension alignment
- EMBOSS needle alignment

Required Python Libraries:

- NumPy 1.5.1 & SciPy 0.9.0
- BioPython 1.56 (1.50+ probably OK, but not tested)
      - Bio.PDB, Bio.SCOP

Installation:
------------

For full functionality you must install 'needle' EMBOSS and SE tools, as
the
HwRMSD method uses those algorithms for Step I and Step II respectively.
However, installing 'needle' EMBOSS is optional if you plan to use your
own
pre-computed sequence alignments (see Usage Cases below)



1. Copy HwRMSD.py and Alignment.py to a desired directory.

2. Install 'needle' EMBOSS:
      You can download EMBOSS from: [http://emboss.sourceforge.net/]
      or from: [http://www.ebi.ac.uk/Tools/emboss/]
      Installation instructions can be found here:
[http://emboss.sourceforge.net/download/#Gettingstarted]

3. Install SE:
```

You can download SE from:
[http://rex.nci.nih.gov/RESEARCH/basic/lmb/mms/sedownload.html]

4. Edit HwRMSD.py to contain proper paths to 'needle' and SE

Set the NEEDLE_PATH constant in HwRMSD.py to the absolute path to the 'needle' program,
usually it is <EMBOSS_install_directory>/mEMBOSS/needle.exe on Windows and
<EMBOSS_install_directory>/emboss/needle on a Unix platform.

If desired, specify the sequence alignment parameters, namely the scoring matrix (M), the gap open
penalty (GO) and the gap extension penalty (GE). The default parameters are 'EBLOSUM50', 10 and 2
respectively

Set the SE_PATH constant in HwRMSD.py to point to the SE program. Usually <SE_install_directory>/test/se.exe
on Windows or <SE_install_directory>/test/se on a Unix platform.

Usage:

>>python HwRMSD.py PDB_FILE_X Chain_X PDB_FILE_Y Chain_Y

for example:

>>python HwRMSD.py 1FJM.pdb A 1TCO.pdb A

this will run all four steps of the method and output the following files:

1. PDBX_PDBY_ARGS.needle - where ARGS will be the parameters used to run needle.
2. PDBX.fasta and PDBY.fasta - the sequences extracted from the PDB files (used by 'needle')
3. sRMSD_PDBX_PDBY.pdb - the standard superposition structue of PDBX
4. wRMSD_PDBX_PDBY.pdb - the wighted superposition structue of PDBX
5. wRMSD_PDBX_PDBY.se - the SE alignment based on the weighted superposition
6+.wRMSD_l_N_wSUM_PDBX_PDBY.pdb - any number of local wRMSD alignments

If you've already generated an intitial sequence alignment,
or want to use one of your own, you can simply supply it as an extra argument. Currently on 'needle' and 'fasta' formats are supported. File extensions
should be *.needle or *.fasta respectively

>>python HwRMSD.py 1FJM.pdb A 1TCO.pdb A 1FJM_1TCO.needle

```
When an alignment file is supplied, the 'needle' alignment is not
performed.
You can skip additional steps of the algorithm as needed with one the
following flags:

-s          don't do the standard superposition (Step II)
-e          don't do the SE sequence alignment (Step IV)
-l          don't do the local weighted superposition (Step V)

You can skip the steps in any combination. If you skip a step, the
assosciated
output files will not be written. Flags can be grouped together after the
dash. For example:

>>python HwRMSD.py -ls 1FJM.pdb A 1TCO.pdb A

This run of HwRMSD will not perform the standard superposition, and won't
attempt
to do the local alignments. Combinations of three flags are also allowed.
Doing this might
not result in a good weighted superposition (since the proteins weren't
brought into initial alignment);
so a typically you would only skip the sRMSD if your input PDB files
already contain a superimposed pair
of structures.

PyMol Visualization of Weights
------------------------------
Any of the output .pdb files will have the weight values for the C-alpha
atoms written to the
b-factor column. To visualize these b-factors in PyMol, you must color
the c-alpha atoms by the
b-factor codes using the spectrum command:

pymol>> hide all
pymol>> show cartoon, all
pymol>> spectrum b, rainbow_rev, n. CA

This coloring is consistent with that in the HwRMSD publications - higher
weight values (smaller distances)
are in blue, lower weight values (larger distances) are in red. The scale
is from 0 to 1.
'''


'''
Edit the paths and parameters as needed here:
-----------------------------------------------
'''
#needle path
NEEDLE_PATH = #something like : "/data/people/you/software/bin/needle"
M = "EBLOSUM50" #the scoring matrix to use with needle
GO = 10         #the gap opening penalty
GE = 2          #the gap extension penalty
```

```
#SE path
SE_PATH = #something like: "/data/people/you/software/bin/se"
'''
-----------------------------------------------
'''

'''
Don't edit these names unless you want to or have to:
These are temporaty files written by the HwRMSD code
-----------------------------------------------
'''
FASTA_FILEX = 'ProteinX.fasta'
FASTA_FILEY = 'ProteinY.fasta'
NULL_FILE = 'log.null'
'''
-----------------------------------------------
'''


import sys, os, re
from math import floor
from subprocess import call
from Bio.PDB.StructureBuilder import StructureBuilder
from Bio.PDB.PDBExceptions import PDBConstructionException
from Bio.PDB.PDBParser import PDBParser
from Bio.PDB.PDBIO import PDBIO
from Bio.PDB.Polypeptide import PPBuilder
from Bio.SCOP.Raf import to_one_letter_code
import numpy


def mean(first,n):
    return [sum(each)/n for each in first]

def nested_list(name,n):
    first1 = name[0:n]
    first2 = name[n:2*n]
    first3 = name[2*n:3*n]
    first_translated =[first1,first2,first3]
    return first_translated

def translation(first,second):
    c = 0
    k = []
    q = len(first)
    while c < q:
        for each in first[c]:
            subtr = each - second[c]
            k.append(subtr)
        c = c + 1
    return k

def normalize(a):
```

```
    B_0 = (a[0])/(numpy.sqrt((a[0]**2)+(a[1]**2)+((a[2])**2)))
    B_1 = (a[1])/(numpy.sqrt((a[0]**2)+(a[1]**2)+((a[2])**2)))
    B_2 = (a[2])/(numpy.sqrt((a[0]**2)+(a[1]**2)+((a[2])**2)))
    return [B_0, B_1, B_2]

def add_coords(first,second):
    c = 0
    k = []
    q = len(first)
    while c < q:
        for each in first[c]:
            add = each + second[c]
            k.append(add)
        c = c + 1
    return k

def sqr(matrix):
    k = []
    for each in matrix:
        sq = (each)**2
        k.append(sq)
    return k

def sqroot(matrix):
    j = []
    for each in matrix:
        sqroot = numpy.sqrt(each)
        j.append(sqroot)
    return j

def Gaussian(first,z):
    value = []
    for each in first:
        weight = (-((each)**2)/z)
        value.append(weight)
    return value

def Gaussian2(first):
    value = []
    for each in first:
        weight = numpy.exp(each)
        value.append(weight)
    return value



def sRMSD(first,second):
    subtr = numpy.subtract(first,second)
    subtr_s = sqr(subtr)
    sum_subtr_s = sum(subtr_s)
    d = sqroot(sum_subtr_s)
    sq_d = sqr(d)
    s_sq_d = sum(sq_d)
    tot = (len(first[0]))
```

```
    value = numpy.sqrt(s_sq_d/tot)
    return value

def wRMSD(first,second,constant):
    subtr = numpy.subtract(first,second)
    subtr_s = sqr(subtr)
    sum_subtr_s = sum(subtr_s)
    d = sqroot(sum_subtr_s)
    weighted_d = Gaussian(d,constant)
    weights = Gaussian2(weighted_d)
    sq_d = sqr(d)
    wd = numpy.multiply(sq_d,weights)
    s_wd = sum(wd)
    n = len(d)
    s_sq_d_divide = s_wd/n
    value = numpy.sqrt(s_sq_d_divide)
    return value

def wSUML(first,second,constant,atoms):
    subtr = numpy.subtract(first,second)
    subtr_s = pow(subtr,2)
    sum_subtr_s = subtr_s.sum(axis=0)
    d = numpy.sqrt(sum_subtr_s)
    weighted_d = Gaussian(d,constant)
    weighted_d = Gaussian2(weighted_d)
    sum_weighted_d = sum(weighted_d)
    value = sum_weighted_d/atoms
    return value



def weight_trans(first,weight,n):
    mult = numpy.multiply(first, weight)
    mean = [sum(each)/n for each in mult]
    return mean

def weight(first,second,constant):
    subtr = numpy.subtract(first,second)
    subtr_s = sqr(subtr)
    sum_subtr_s = sum(subtr_s)
    d = sqroot(sum_subtr_s)
    weighted_d = Gaussian(d,constant)
    weighted_d = Gaussian2(weighted_d)
    return weighted_d



class HwRMSDAlignment(object):
    '''
    Class for handling sequence to structure correspondence
    '''

    def calcRMSD(self):
        x = numpy.transpose(self.aligned_coordinates_X)
```

```
        y = numpy.transpose(self.aligned_coordinates_Y)
        rmsd = sRMSD(x,y)
        return rmsd

    def calcwRMSD(self):
        x = numpy.transpose(self.aligned_coordinates_X)
        y = numpy.transpose(self.aligned_coordinates_Y)
        rmsd = wRMSD(x,y,self.scaling_factor)
        return rmsd

    def calcwSUM(self):
        x = numpy.transpose(self.aligned_coordinates_X)
        y = numpy.transpose(self.aligned_coordinates_Y)
        wsum = wSUML(x,y,self.scaling_factor,x.shape[1])
        return wsum

    def updateX(self,newCACoords):
        allCAs =[]
        for model in self.X.get_list():
            chain = model.get_list()
            for each in chain:
                for res in each.get_list():
                    for x in res.get_list():
                        allCAs.append(x)
        if len(allCAs) != len(newCACoords):
            sys.exit("\nERROR: Wrong number of atoms .. structure
had",len(allCAs),"you gave me",len(newCACoords))
        for newCoords,ca in zip(newCACoords,allCAs):
            ca.set_coord(newCoords)

    def dowRMSD(self, x = None, y = None, scaling_factor = None):
        if (x is None) or (y is None):
            x = numpy.transpose(self.aligned_coordinates_X)
            y = numpy.transpose(self.aligned_coordinates_Y)
        else:
            x = numpy.transpose(x)
            y = numpy.transpose(y)

        all = self.get_all_original_coordsX()
        n = x.shape[1]

        #WEIGHTED RMSD CALCULATION, z = # of iterations
        j= 1
        weighted_rmsds = []

        w_metric = []
        weights = []

        #SET APPROPRIATE SCALING FACTOR
        if scaling_factor == None and self.scaling_factor == None:
            scaling_factor = self.calcRMSD()
            self.scaling_factor = scaling_factor
            print "Set scaling factor to %s based on
sRMSD."%(scaling_factor)
```

```
        elif self.scaling_factor is not None:
            scaling_factor = self.scaling_factor
            print "Set scaling factor to %s."%(scaling_factor)

        while j <= self.MAX_ITER:
            #TRANSLATE WEIGHTED CENTROIDS TO ORIGIN
            #CALCULATE WEIGHTS (protein1, protein2, scaling_factor)
            weights = weight(x,y,scaling_factor)
            weighted_x_mean =  weight_trans(x,weights,n)
            weighted_y_mean =  weight_trans(y,weights,n)
            x_trans = translation(x, weighted_x_mean)
            y_trans = translation(y, weighted_y_mean)
            x_translated = nested_list(x_trans,n)
            y_translated = nested_list(y_trans,n)

            #CALCULATE WEIGHTED COVARIENCE MATRIX
            weighted_rot =
weight(x_translated,y_translated,scaling_factor)
            weighted_x_translated =
numpy.multiply(weighted_rot,x_translated)
            wx_transpose = numpy.transpose(weighted_x_translated)
            R = numpy.dot(y_translated,wx_transpose)
            R_transpose = numpy.transpose(R)
            R2 = numpy.dot(R_transpose, R)

            #DETERMINE THE EIGENVECTORS AND EIGENVALUES of R2
            mu,A = numpy.linalg.eig(R2)
            A = numpy.transpose(A)

            #SORT EIGENVECTORS IN DECREASING ORDER OF EIGENVALUES
            a = [(mu[i],A[i]) for i in range(len(A))]
            a.sort()
            a.reverse()
            mu = [x[0] for x in a]
            A = [x[1] for x in a]

            #DETERMINE RIGHT-HANDED SYSTEM
            A_3 = numpy.cross(A[0], A[1])
            A = [A[0], A[1], A_3]

            #CALCULATE B, NORMALIZED PRODUCT OF (RxA)
            B_1 = numpy.dot(R, A[0])
            B_2 = numpy.dot(R, A[1])
            norm_B_1 = normalize(B_1)
            norm_B_2 = normalize(B_2)
            norm_B_3 = numpy.cross(norm_B_1,norm_B_2)
            B = [norm_B_1,norm_B_2,norm_B_3]
            B_transpose = numpy.transpose(B)

            #CALCULATE WEIGHTED ROTATION MATRIX, U
            U = numpy.dot(B_transpose, A)
            x_rot = numpy.dot(U,x_translated)

            #CALCULATE WEIGHTED RMSD
```

```
            weighted_rmsds.append(wRMSD(x_rot,
y_translated,scaling_factor))
            w_metric.append(wSUML(x_rot, y_translated,scaling_factor,n))

            #DETERMINE IF CONVERGENCE IS REACHED
            if j > 1:
                wrmsd_diff = weighted_rmsds[-2] - weighted_rmsds[-1]
            else:
                wrmsd_diff = []

            if 0 < wrmsd_diff < self.MIN_DIFF:

                #ADD MEAN VALUES OF PROTEIN Y TO ROTATED COORDINATES OF
PROTEIN X
                x_coords = add_coords(x_rot, weighted_y_mean)
                x = nested_list(x_coords,n)

                #TRANSFORM ALL COORDINATES OF PROTEIN X
                all_trans = translation(all, weighted_x_mean)
                k = len(all[0])
                all_translated = nested_list(all_trans,k)
                all_rot = numpy.dot(U,all_translated)

                #ADD ALL MEAN VALUES OF PROTEIN Y TO ALL ROTATED
COORDINATES OF PROTEIN X
                all_coords = add_coords(all_rot, weighted_y_mean)
                all = nested_list(all_coords,k)
                allrot = numpy.transpose(all)
                #print "Final wRMSD = %s"%weighted_rmsds[-1]
                #print "Final wSUM = %s"%w_metric[-1]

                #setAll(set,allrot)
                #weights = weight(x,y,scaling_factor)
                #set = setBfactors(set,xID,weights)
                #filename = "%s_wRMSD_with_b-
factors.pdb"%(sRMSD_filename.split('_')[0])
                #writeStructure(set,filename)
                self.updateX(allrot)
                x_CA_names,x_CA_coords = self.get_CA_coordsX(self.chainX)
                self.aligned_coordinates_X =
map(x_CA_coords.get,self.aligned_IDs_X)
                print "Weighted RMSD alignment performed, converged after
%s iterations"%j

                break


            else:
                #ADD MEAN VALUES OF PROTEIN Y TO ROTATED COORDINATES OF
PROTEIN X
                x_coords = add_coords(x_rot, weighted_y_mean)
                x = nested_list(x_coords,n)

                #TRANSFORM ALL COORDINATES OF PROTEIN X
```

```
                all_trans = translation(all, weighted_x_mean)
                k = len(all[0])
                all_translated = nested_list(all_trans,k)
                all_rot = numpy.dot(U,all_translated)

                #ADD ALL MEAN VALUES OF PROTEIN Y TO ALL ROTATED
COORDINATES OF PROTEIN X
                all_coords = add_coords(all_rot, weighted_y_mean)
                all = nested_list(all_coords,k)
                j = j + 1
        else:
            print "Alignment stopped after 5000 iterations, convergence
was never reached"
            allrot = numpy.transpose(all)


    def dosRMSD(self):
        x = numpy.transpose(self.aligned_coordinates_X)
        y = numpy.transpose(self.aligned_coordinates_Y)
        all = self.get_all_original_coordsX()

        #Initial standard alignment without weight
        #TRANSLATE PROTEINS X AND Y TO CENTER
        n = x.shape[1]

        x_mean = x.mean(axis=1) # mean(x,n)
        y_mean = y.mean(axis=1) # mean(y,n)

        x_trans = translation(x, x_mean)
        x_translated = nested_list(x_trans,n)
        y_trans = translation(y, y_mean)
        y_translated = nested_list(y_trans,n)
        x_transpose = numpy.transpose(x_translated)


        #CALCULATE COVARIANCE MATRIX (y_translated *x_translated^t)
        R = numpy.dot(y_translated, x_transpose)
        R_transpose = numpy.transpose(R)
        R2 = numpy.dot(R_transpose, R)

        #DETERMINE THE EIGENVECTORS AND EIGENVALUES of R2
        #mu,A = LinearAlgebra.eigenvectors(R2)
        mu,A = numpy.linalg.eig(R2)
        A = numpy.transpose(A)

        #SORT EIGENVECTORS IN DECREASING ORDER OF EIGENVALUES
        a = [(mu[i],A[i]) for i in range(len(A))]
        a.sort()
        a.reverse()
        mu = [x[0] for x in a]
        A = [x[1] for x in a]

        #DETERMINE RIGHT-HANDED SYSTEM
        A_3 = numpy.cross(A[0], A[1])
```

```
        A = [A[0], A[1], A_3]

        #CALCULATE B, NORMALIZED PRODUCT OF (RxA)
        B_1 = numpy.dot(R, A[0])
        B_2 = numpy.dot(R, A[1])
        norm_B_1 = normalize(B_1)
        norm_B_2 = normalize(B_2)
        norm_B_3 = numpy.cross(norm_B_1,norm_B_2)
        B = [norm_B_1,norm_B_2,norm_B_3]
        B_transpose = numpy.transpose(B)


        #CALCULATE ROTATION MATRIX, U
        U = numpy.dot(B_transpose, A)

        x_rot = numpy.dot(U,x_translated)

        #CALCULATE STANDARD RMSD
        #standard_RMSD = sRMSD(x_rot, y_translated)

        #ADD MEAN VALUES OF PROTEIN Y TO ROTATED COORDINATES OF PROTEIN X
        x_coords = add_coords(x_rot, y_mean)
        x = nested_list(x_coords,n)

        #TRANSLATE ALL COORDINATES OF PROTEIN X
        all_trans = translation(all, x_mean)
        k = len(all[0])
        all_translated = nested_list(all_trans,k)
        all_rot = numpy.dot(U,all_translated)

        #ADD ALL MEAN VALUES OF PROTEIN Y TO ALL ROTATED COORDINATES OF
PROTEIN X
        all_coords = add_coords(all_rot, y_mean)
        all = nested_list(all_coords,k)
        allrot = numpy.transpose(all)

        self.updateX(allrot)
        x_CA_names,x_CA_coords = self.get_CA_coordsX(self.chainX)
        self.aligned_coordinates_X =
map(x_CA_coords.get,self.aligned_IDs_X)

        print "Standard RMSD alignment performed"


    def output_with_b_factors(self,filename=None):
        #OUTPUT STANDARD RMSD ALIGNMENT
        if not filename:
            filename =
"%s_%s_sRMSD.pdb"%(self.originalX.get_id(),self.originalY.get_id())
        io = PDBIO()
        io.set_structure(self.X)
        io.save(filename)

    def get_all_original_coordsX(self):
```

```
        result = []
        for model in self.originalX.get_list():
            chain = model.get_list()
            for each in chain:
                for res in each.get_list():
                    for x in res.get_list():
                        result.append(x.get_coord())
        x_t = numpy.transpose(result)
        return x_t


    def get_CA_coordsX(self,X_chain):
        X_names_dict = {}
        X_coords_dict = {}
        #assume only one model
        for model in self.X.get_list():
            chain = model[X_chain]
            for residue in chain.get_list():
                #residue must have a C-alpha atom and either have an ATOM
type
                #or be in the SCOP.Ref.to_one_letter_code list of
modified residues)
                if residue.has_id("CA") and (residue.get_id()[0] == ' '
or residue.get_resname() in to_one_letter_code.keys()):
                    try:
                        X_coords_dict[residue.get_id()[1]] =
residue["CA"].get_coord()
                        X_names_dict[residue.get_id()[1]] =
to_one_letter_code[residue.get_resname()]
                    except:
                        pass
        return X_names_dict, X_coords_dict

    def get_CA_original_coords(self,X_chain, Y_chain):
        X_names_dict = {}
        X_coords_dict = {}
        Y_names_dict = {}
        Y_coords_dict = {}
        #assume only one model
        for model in self.originalX.get_list():
            chain = model[X_chain]
            for residue in chain.get_list():
                #residue must have a C-alpha atom and either have an ATOM
type
                #or be in the SCOP.Ref.to_one_letter_code list of
modified residues)
                if residue.has_id("CA") and (residue.get_id()[0] == ' '
or residue.get_resname() in to_one_letter_code.keys()):
                    try:
                        X_coords_dict[residue.get_id()[1]] =
residue["CA"].get_coord()
                        X_names_dict[residue.get_id()[1]] =
to_one_letter_code[residue.get_resname()]
                    except:
```

```
                    pass
        for model in self.originalY.get_list():
            chain = model[Y_chain]
            for residue in chain.get_list():
                #residue must have a C-alpha atom and either have an ATOM
type
                #or be in the SCOP.Ref.to_one_letter_code list of
modified residues)
                if residue.has_id("CA") and (residue.get_id()[0] == ' '
or residue.get_resname() in to_one_letter_code.keys()):
                    try:
                        Y_coords_dict[residue.get_id()[1]] =
residue["CA"].get_coord()
                        Y_names_dict[residue.get_id()[1]] =
to_one_letter_code[residue.get_resname()]
                    except:
                        pass

        return X_names_dict, X_coords_dict, Y_names_dict, Y_coords_dict

    def load_alignment(self, filepath, X_chain, Y_chain):
        '''
        pre-condition:
            Loaded X and Y structures
            Pass in path to alignment file and chain IDs

        post-condition:
            Loaded Coordinates and Names for X and Y coordinated residues
        '''

        try:
            input = open(filepath,'r')
            filestring = input.read()
            input.close()
        except:
            sys.exit("\nERROR: Couldn't open alignment file >>
%s"%filepath)

        #The following parses the alignment and the starting residue IDs
        #It is specific to the output format generated by the "needle"
        #tool of the EMBOSS tool package
        if filepath.split('.')[-1] == 'needle':
            filestring = re.sub("\n","~",filestring)
            match = re.search("Score(.*?)={2,40}(.*?)#-
{2,40}",filestring)
            if match:
                alignment = match.group(2)
            alignment_list = alignment.split("~")
            x_string = "".join(alignment_list[2:len(alignment_list):4])
            y_string = "".join(alignment_list[4:len(alignment_list):4])
            x_string_list = x_string.split()
            y_string_list = y_string.split()
            X_start = int(x_string_list[0])
            Y_start = int(y_string_list[0])
```

```
            X_alignment_string =
"".join(x_string_list[1:len(x_string_list):3])
            Y_alignment_string =
"".join(y_string_list[1:len(y_string_list):3])
            X_alignment_string.upper()
            Y_alignment_string.upper()
        elif filepath.split('.')[-1] == 'fasta':
            filestring = re.sub("\n","~",filestring)
            match = re.match(">(.*?)>(.*)",filestring)
            if match:
                Y_alignment_string =
"".join(match.group(1).split('~')[1:])
                X_alignment_string =
"".join(match.group(2).split('~')[1:])
        else:
            sys.exit("ERROR: Unsupported alignment format. Make sure the
alignment is in .needle or .fasta format")

        #Check that the lengths of the alignment strings agree
        if len(X_alignment_string) != len(Y_alignment_string):
            sys.exit("\nERROR: Alignment lengths don't agree! %s
%s"%(len(X_alignment_string),len(Y_alignment_string)))


        #Try assigning residue IDs to the alignments, and return the
aligned IDs
        x_CA_names, x_CA_coords, y_CA_names, y_CA_coords =
self.get_CA_original_coords(X_chain, Y_chain)


        #X_aligned_IDs, Y_aligned_IDs =
inferAlignedResidues(X_alignment_string, Y_alignment_string, X_start,
Y_start, x_CA_names, y_CA_names)
        #def inferAlignedResidues(X_string, Y_string, X_start, Y_start,
X_names, Y_names):
        x_ID = X_start
        y_ID = Y_start
        for i in range(0,len(X_alignment_string),1):
            while x_ID not in x_CA_names.keys():
                x_ID += 1
                if x_ID > max(x_CA_names.keys()):
                    break
            while y_ID not in y_CA_names.keys():
                y_ID += 1
                if y_ID > max(y_CA_names.keys()):
                    break
            if X_alignment_string[i] == '-':
                y_ID += 1
            elif Y_alignment_string[i] == '-':
                x_ID += 1
            else:
                if X_alignment_string[i] != x_CA_names[x_ID] or
Y_alignment_string[i] != y_CA_names[y_ID]:
```

```
                    sys.exit("\nERROR: Alignment residue name doesn't
match structure residue name. X: alignment %s vs structure %s Y:
alignment %s vs structure
%s"%(X_alignment_string[i],x_CA_names[x_ID],Y_alignment_string[i],y_CA_na
mes[y_ID]))
                self.aligned_IDs_X.append(x_ID)
                self.aligned_IDs_Y.append(y_ID)
                x_ID += 1
                y_ID += 1


        #GET COORDINATES THAT COORESPOND TO POSITIONS FROM SEQUENCE
ALIGNMENT
        self.aligned_coordinates_X =
map(x_CA_coords.get,self.aligned_IDs_X)
        self.aligned_coordinates_Y =
map(y_CA_coords.get,self.aligned_IDs_Y)

        #FINAL CHECK TO ENSURE RESIDUE CORRESPONDENCE
        if len(self.aligned_coordinates_X) !=
len(self.aligned_coordinates_Y):
            sys.exit("\nERROR:Proteins do not have same number of atoms;
Protein X has ",len(self.aligned_coordinates_X)," atoms, while Protein Y
has ",len(self.aligned_coordinates_Y)," atoms")

        #CHECK TO DETERMINE IF APPROPRIATE NUMBER OF COORDINATES PRESENT
        if (len(self.aligned_coordinates_X[0]) != 3):
            sys.exit("\nERROR:Protein X does not have a 3xn atom
coordinate set")
        if (len(self.aligned_coordinates_Y[0]) != 3):
            sys.exit("\nERROR:Protein Y does not have a 3xn atom
coordinate set")

        #CHECK TO DETERMINE IF >4 COORDINATES PRESENT FOR EACH PROTEIN
        if len(self.aligned_coordinates_X) < 5:
            sys.exit("\nERROR:Protein X has 4 or less coordinates, 5 or
more needed to perform alignment")
        if len(self.aligned_coordinates_Y) < 5:
            sys.exit("\nERROR:Protein Y has 4 or less coordinates, 5 or
more needed to perform alignment")

    def __init__(self, structureX,chainX, structureY, chainY,
sequence_alignment_path):

        '''
        Constructor accepts two structure objects, chain names, and a
path to a sequence alignment file

        structureX and structureY are BioPython PDB:Structure objects
        chainX and chainY are strings

        '''
        self.aligned_coordinates_X = numpy.array([],dtype="float64")
        self.aligned_coordinates_Y = numpy.array([],dtype="float64")
```

```
        self.aligned_IDs_X = []
        self.aligned_IDs_Y = []
        self.X = structureX
        self.originalX = structureX
        self.originalY = structureY
        self.chainX = chainX
        self.chainY = chainY
        self.MAX_ITER = 2500
        self.MIN_DIFF = 0.000001
        self.scaling_factor = None


        #The following will load the aligned_coordinates and aligned_IDs
arrays

self.load_alignment(sequence_alignment_path,self.chainX,self.chainY)
        print "Alignment loaded %s // %s coordinated residues in X and
Y"%(len(self.aligned_IDs_X),len(self.aligned_IDs_X))



def getSequence(chain):
    chain_string = ""
    for residue in chain.get_list():
        if residue.has_id("CA") and (residue.get_id()[0] == ' ' or
residue.get_resname() in to_one_letter_code.keys()):
            chain_string =
"".join((chain_string,to_one_letter_code[residue.get_resname()]))
    return chain_string

if __name__ == '__main__':

    NO_SRMSD = False
    NO_WRMSD = False
    NO_LOCAL = False
    NO_SE    = False
    afni     = None
    flag     = None

    if len(sys.argv) < 4:
        print "usage: Global_HwRMSD.py Protein_X.pdb Protein_X_ChainID
Protein_Y.pdb Protein_Y_ChainID [AlignmentFile(optional)]"
        sys.exit()
    elif len(sys.argv) == 4:
            FNX = sys.argv[1]
            FNY = sys.argv[3]
            chainX = sys.argv[2]
            chainY = sys.argv[4]
    elif len(sys.argv) > 4:
        if sys.argv[1][0] == "-":
            flag = sys.argv[1]
            FNX = sys.argv[2]
            FNY = sys.argv[4]
            chainX = sys.argv[3]
```

```
            chainY = sys.argv[5]
            afni = 6
        else:
            FNX = sys.argv[1]
            FNY = sys.argv[3]
            chainX = sys.argv[2]
            chainY = sys.argv[4]
            afni = 5

    #See if there were flags specified
    if flag:
        if 's' in flag:
            NO_SRMSD = True
        #if 'w' in flag:
        #    NO_WRMSD = True
        if 'l' in flag:
            NO_LOCAL = True
        if 'e' in flag:
            NO_SE = True

    #See if there was an alignment file specified
    try:
            alignment_filename = sys.argv[afni]
    except:
            alignment_filename = None

    #Make sure we can get the structures:
    parser = PDBParser()
    try:
        structureX = parser.get_structure(FNX.split('.')[-2], FNX)
    except:
        sys.exit("Couldn't open structure file X %s"%FNX)

    try:
        structureY = parser.get_structure(FNY.split('.')[-2], FNY)
    except:
        sys.exit("Couldn't open structure file Y %s"%FNY)



    ############################
    #Part I. Sequence Alignment
    ############################

    #Generate the sequence alignment file if none is given
    if not alignment_filename:

        try:
            #NOT using PPBuilder for this because it won't assemble all
the residues
            #    in a chain if there's a physical gap in the chain.
            stringX = getSequence(structureX[0][chainX])
            output = open(FASTA_FILEX,'w')
            output.write(str(stringX))
```

```
            output.close()

            stringY = getSequence(structureY[0][chainY])
            output = open(FASTA_FILEY,'w')
            output.write(str(stringY))
            output.close()
        except:
            sys.exit("ERROR: Couldn't generate FASTA files for
alignment")

        print "\nI. Running 'needle' to generate initial sequence
alignment"
        print "-----------------------------------------------------"
        print "'needle' parameters: Scoring Matrix = %s, Gap Open = %s,
Gap Extend = %s"%(M,GO,GE)
        alignment_filename =
'%s_%s_%s_%s_%s.needle'%(structureX.get_id(),structureY.get_id(),M,GO,GE)

        cmd_line = '%s %s %s %s -datafile %s -gapopen %s -gapextend
%s'%(NEEDLE_PATH,FASTA_FILEX,FASTA_FILEY,alignment_filename,M,GO,GE)

        try:
            FDUMP = open(NULL_FILE,'w')
            retcode = call(cmd_line, stdout=FDUMP, stderr=FDUMP,
shell=True)
            FDUMP.close()
            os.system('rm %s'%NULL_FILE)
        except OSError, e:
            sys.exit("\nERROR: Couldn't run needle >> %s"%cmd_line)
    else:
        print "\nI. Using alignment file %s"%alignment_filename
        print "-----------------------------------------------------"


    ####################
    #Initialize alignment
    ####################
    alignment = HwRMSDAlignment(structureX,chainX,structureY,chainY,
alignment_filename)

    ################################################
    #Part II. Perform sRMSD and wRMSD superpositions
    ################################################
    if not NO_SRMSD:
        print "\nII. Running initial sRMSD superposition"
        print "-----------------------------------------------------"
        alignment.dosRMSD()
        srmsd_filename =
"sRMSD_%s_%s.pdb"%(alignment.originalX.get_id(),alignment.originalY.get_i
d())
        alignment.output_with_b_factors(srmsd_filename)
        print "Standard RMSD superposition file written:
%s"%srmsd_filename
    else:
```

```
        print "\nII. Skipping initial sRMSD superposition"

    rmsd = alignment.calcRMSD()
    print "Initial RMSD (X to Y) = %s"%rmsd

    if not NO_WRMSD:
        print "\nIII. Running wRMSD superposition"
        print "-----------------------------------------------------"
        alignment.dowRMSD()
        wrmsd_filename =
"wRMSD_%s_%s.pdb"%(alignment.originalX.get_id(),alignment.originalY.get_i
d())
        alignment.output_with_b_factors(wrmsd_filename)
        print "Weighted RMSD superposition file written:
%s"%wrmsd_filename
        rmsd = alignment.calcRMSD()
        print "Final RMSD (X to Y) = %s"%rmsd
        wrmsd = alignment.calcwRMSD()
        print "Final wRMSD (X to Y) = %s"%wrmsd
        wsum = alignment.calcwSUM()
        print "Final wSUM (X to Y) = %s"%wsum
    else:
        print "\nIII. Skipping wRMSD superposition"



    ################################################
    #Part III. Run SE on the generated superposition
    ################################################
    if not NO_SE:
        print "\nIV. Running SE"
        print "-----------------------------------------------------"
        print "'SE' parameters: None"
        #print "Parameters: Scoring Matrix = %s, Gap Open = %s, Gap
Extend = %s"%(M,GO,GE)
        alignment_filename =
'wRMSD_%s_%s.se'%(structureX.get_id()[0],structureY.get_id()[0])
        cmd_line = '%s -fasta %s %s'%(SE_PATH,wrmsd_filename,FNY)
        try:
            FDUMP = open('log.null','w')
            retcode = call(cmd_line, stdout=FDUMP, stderr=FDUMP,
shell=True)
            FDUMP.close()
            os.system('rm log.null')
        except:
            sys.exit("\nERROR: Couldn't run SE >> %s"%cmd_line)
        print "'SE' alignment written to:
%s"%("".join((wrmsd_filename.split('.')[0],"-
",structureY.get_id(),".fasta")))
    else:
        print "\nIV. Skipping 'SE' alignment"


    if not NO_LOCAL:
```

```
        print "\nIII. Running Local wRMSD superpositions"
        print "-----------------------------------------------------"
        seq_length =
int(min(len(alignment.aligned_coordinates_X),len(alignment.aligned_coordi
nates_Y)))
        loc_length = round(seq_length/10)
        previous_i = 0
        for i in xrange(loc_length,seq_length,loc_length):
            print "\nRegion %s to %s"%(previous_i, i)
            alignment.dosRMSD()
            x = alignment.aligned_coordinates_X[previous_i:i]
            y = alignment.aligned_coordinates_Y[previous_i:i]
            alignment.dowRMSD(x,y)
            previous_i = i
            rmsd = alignment.calcRMSD()
            print "Local %s RMSD (X to Y) = %s"%(i,rmsd)
            wrmsd = alignment.calcwRMSD()
            print "Local %s wRMSD (X to Y) = %s"%(i,wrmsd)
            wsum = alignment.calcwSUM()
            print "Local %s wSUM (X to Y) = %s"%(i,wsum)
            wrmsd_filename =
"wRMSD_local_%s_%2f_%s_%s.pdb"%(i,wsum,alignment.originalX.get_id(),align
ment.originalY.get_id())
            alignment.output_with_b_factors(wrmsd_filename)
            print "Local weighted RMSD superposition file written:
%s"%wrmsd_filename
    else:
        print "\nV. Skipping local wRMSD superposition"
```