```python
# Implementation for a Disulfide Bond Class object.
# Based on the original C/C++ implementation by Eric G. Suchanek
# Part of the program Proteus, a program for the analysis and modeling of
# protein structures, with an emphasis on disulfide bonds.
# Author: Eric G. Suchanek, PhD
# Last revision: 1/11/2023

import math
from math import cos

import pickle
from tqdm import tqdm

from proteusPy import *
from proteusPy.atoms import *
from proteusPy.proteusGlobals import *
from proteusPy.DisulfideExceptions import *
from proteusPy.DisulfideGlobals import *
from proteusPy.DisulfideList import DisulfideList

from Bio.PDB import Vector, PDBParser, PDBList
from Bio.PDB.vectors import calc_dihedral

import pandas as pd
import pyvista as pv

# float init for class
_FLOAT_INIT = -999.9
_ANG_INIT = -180.0

# tqdm progress bar width
_PBAR_COLS = 100

# columns for the torsions file dataframe.
Torsion_DF_Cols = ['source', 'ss_id', 'proximal', 'distal', 'chi1',
'chi2', 'chi3', 'chi4', \
           'chi5', 'energy', 'ca_distance', 'phi_prox', 'psi_prox',
'phi_dist',\
           'psi_dist']

# make a colormap in vector space from starting color to
# ending color

#from proteusPy import cmap_vector


# Class definition for a Disulfide bond.
class Disulfide:
```



```python
    """
    This class provides an object representing a physical disulfide bond
    that is either extracted from the RCSB protein databank or built
    using the proteusPy.Turtle3D. The Disulfide Bond is characterized
    by the atomic coordinates N, Cα, Cβ, C', Sγ for both residues, the
    dihedral angles X1 - X5 for the disulfide bond conformation,a name,
    proximal resiude number and distal residue number, and conformational
    energy. All atomic coordinates are represented by the BIO.PDB.Vector
    class. The class uses the internal methods to initialize dihedral
    angles and approximate energy upon initialization.

    """
    def __init__(self, name="SSBOND"):
        """
        Initialize the class. All positions are set to the origin.
        The optional string name may be passed.
        """
        self.name = name
        self.proximal = -1
        self.distal = -1
        self.energy = _FLOAT_INIT
        self.proximal_chain = str('')
        self.distal_chain = str('')
        self.pdb_id = str('')
        self.proximal_residue_fullid = str('')
        self.distal_residue_fullid = str('')
        self.PERMISSIVE = bool(True)
        self.QUIET = bool(True)
        self.ca_distance = _FLOAT_INIT
        self.torsion_array = numpy.array((_ANG_INIT, _ANG_INIT, _ANG_INIT,
                                          _ANG_INIT, _ANG_INIT))
        self.phiprox = _ANG_INIT
        self.psiprox = _ANG_INIT
        self.phidist = _ANG_INIT
        self.psidist = _ANG_INIT

        # global coordinates for the Disulfide, typically as returned from
the PDB file
        self.n_prox = Vector(0,0,0)
        self.ca_prox = Vector(0,0,0)
        self.c_prox = Vector(0,0,0)
        self.o_prox = Vector(0,0,0)
        self.cb_prox = Vector(0,0,0)
        self.sg_prox = Vector(0,0,0)
        self.sg_dist = Vector(0,0,0)
        self.cb_dist = Vector(0,0,0)
        self.ca_dist = Vector(0,0,0)
        self.n_dist = Vector(0,0,0)
```

```
94          self.c_dist = Vector(0,0,0)
95          self.o_dist = Vector(0,0,0)
96
97          # set when we can't find previous or next prox or distal
98          # C' or N atoms.
99          self.missing_atoms = False
100
101         # need these to calculate backbone dihedral angles
102         self.c_prev_prox = Vector(0,0,0)
103         self.n_next_prox = Vector(0,0,0)
104         self.c_prev_dist = Vector(0,0,0)
105         self.n_next_dist = Vector(0,0,0)
106
107         # local coordinates for the Disulfide, computed using the Turtle3D
    … in
108         # Orientation #1 these are generally private.
109
110         self._n_prox = Vector(0,0,0)
111         self._ca_prox = Vector(0,0,0)
112         self._c_prox = Vector(0,0,0)
113         self._o_prox = Vector(0,0,0)
114         self._cb_prox = Vector(0,0,0)
115         self._sg_prox = Vector(0,0,0)
116         self._sg_dist = Vector(0,0,0)
117         self._cb_dist = Vector(0,0,0)
118         self._ca_dist = Vector(0,0,0)
119         self._n_dist = Vector(0,0,0)
120         self._c_dist = Vector(0,0,0)
121         self._o_dist = Vector(0,0,0)
122
123         # need these to calculate backbone dihedral angles
124         self._c_prev_prox = Vector(0,0,0)
125         self._n_next_prox = Vector(0,0,0)
126         self._c_prev_dist = Vector(0,0,0)
127         self._n_next_dist = Vector(0,0,0)
128
129         # Dihedral angles for the disulfide bond itself, set to _ANG_INIT
130         self.chi1 = _ANG_INIT
131         self.chi2 = _ANG_INIT
132         self.chi3 = _ANG_INIT
133         self.chi4 = _ANG_INIT
134         self.chi5 = _ANG_INIT
135
136         # Initialize an array for the torsions which will be used for
    … comparisons
137         self.dihedrals = numpy.array((_ANG_INIT, _ANG_INIT, _ANG_INIT,
138                                       _ANG_INIT, _ANG_INIT), "d")
139
```



```
140     def internal_coords(self) -> numpy.array:
141         res_array = numpy.zeros(shape=(16,3))
142
143         res_array = numpy.array((
144             self._n_prox.get_array(),
145             self._ca_prox.get_array(),
146             self._c_prox.get_array(),
147             self._o_prox.get_array(),
148             self._cb_prox.get_array(),
149             self._sg_prox.get_array(),
150             self._n_dist.get_array(),
151             self._ca_dist.get_array(),
152             self._c_dist.get_array(),
153             self._o_dist.get_array(),
154             self._cb_dist.get_array(),
155             self._sg_dist.get_array(),
156             self._c_prev_prox.get_array(),
157             self._n_next_prox.get_array(),
158             self._c_prev_dist.get_array(),
159             self._n_next_dist.get_array()
160         ))
161         return res_array
162
163     @property
164     def cofmass(self) -> numpy.array:
165         res = numpy.zeros(shape=(16,3))
166         res = self.internal_coords()
167         return res.mean(axis=0)
168
169     def internal_coords_res(self, resnumb) -> numpy.array:
170         res_array = numpy.zeros(shape=(6,3))
171
172         if resnumb == self.proximal:
173             res_array = numpy.array((
174                 self._n_prox.get_array(),
175                 self._ca_prox.get_array(),
176                 self._c_prox.get_array(),
177                 self._o_prox.get_array(),
178                 self._cb_prox.get_array(),
179                 self._sg_prox.get_array(),
180             ))
181             return res_array
182         elif resnumb == self.distal:
183             res_array = numpy.array((
184                 self._n_dist.get_array(),
185                 self._ca_dist.get_array(),
186                 self._c_dist.get_array(),
187                 self._o_dist.get_array(),
```

```
                self._cb_dist.get_array(),
                self._sg_dist.get_array(),
            ))
            return res_array
        else:
            mess = f'-> Disulfide.internal_coords(): Invalid argument. \
             Unable to find residue: {resnumb} '
            raise DisulfideConstructionWarning(mess)

    def get_chains(self):
        prox = self.proximal_chain
        dist = self.distal_chain
        return tuple(prox, dist)

    def same_chains(self) -> bool:
        (prox, dist) = self.get_chains()
        if prox == dist:
            return True
        else:
            return False

    def reset(self) -> None:
        self.__init__(self)

    def copy(self):
        return copy.deepcopy(self)

    def compute_extents(self, dim='z'):
        ic = self.internal_coords()
        # set default index to 'z'
        idx = 2

        if dim == 'x':
            idx = 0
        elif dim == 'y':
            idx = 1
        elif dim == 'z':
            idx = 2

        _min = ic[:, idx].min()
        _max = ic[:, idx].max()
        return _min, _max

    def bounding_box(self):
        res = numpy.zeros(shape=(3, 2))
        xmin, xmax = self.compute_extents('x')
        ymin, ymax = self.compute_extents('y')
        zmin, zmax = self.compute_extents('z')
```



```

        res[0] = [xmin, xmax]
        res[1] = [ymin, ymax]
        res[2] = [zmin, zmax]

        return res

    def _render(self, pvplot: pv.Plotter(), style='bs', plain=False,
            bondcolor=BOND_COLOR, bs_scale=BS_SCALE, spec=SPECULARITY,
            specpow=SPEC_POWER) -> pv.Plotter:
        '''
        Update the passed pyVista plotter() object with the mesh data for
the
        input Disulfide Bond.

        Arguments:
            pvpplot: pyvista.Plotter() object
            style: 'bs', 'st', 'cpk', 'plain': Whether to render as CPK,
            ball-and-stick or stick. Bonds are colored by atom color,
unless 'plain'
            is specified.

        Returns:
            Updated pv.Plotter() object.
        '''

        bradius = BOND_RADIUS
        coords = self.internal_coords()
        missing_atoms = self.missing_atoms

        atoms = ('N', 'C', 'C', 'O', 'C', 'SG', 'N', 'C', 'C', 'O', 'C',
                 'SG', 'C', 'N', 'C', 'N')
        pvp = pvplot

        # bond connection table with atoms in the specific order shown
above:
        # returned by ss.get_internal_coords()

        def draw_bonds(pvp, bradius=BOND_RADIUS, style='sb',
                       bcolor=BOND_COLOR, missing=True):
            bond_conn = numpy.array(
            [
                [0, 1], # n-ca
                [1, 2], # ca-c
                [2, 3], # c-o
                [1, 4], # ca-cb
                [4, 5], # cb-sg
                [6, 7], # n-ca
```

```
                [7, 8], # ca-c
                [8, 9], # c-o
                [7, 10], # ca-cb
                [10, 11], # cb-sg
                [5, 11],  # sg -sg
                [12, 0],  # cprev_prox-n
                [2, 13],  # c-nnext_prox
                [14,6],   # cprev_dist-n_dist
                [8,15]    # c-nnext_dist
            ])

            # colors for the bonds. Index into ATOM_COLORS array
            bond_split_colors = numpy.array(
                [
                    ('N', 'C'),
                    ('C', 'C'),
                    ('C', 'O'),
                    ('C', 'C'),
                    ('C', 'SG'),
                    ('N', 'C'),
                    ('C', 'C'),
                    ('C', 'O'),
                    ('C', 'C'),
                    ('C', 'SG'),
                    ('SG', 'SG'),
                    # prev and next C-N bonds - color by atom Z
                    ('C', 'N'),
                    ('C', 'N'),
                    ('C', 'N'),
                    ('C', 'N')
                ]
            )
            # work through connectivity and colors
            orig_col = dest_col = bcolor

            for i in range(len(bond_conn)):
                if i > 10 and missing_atoms == True: # skip missing atoms
                    continue

                bond = bond_conn[i]

                # get the indices for the origin and destination atoms
                orig = bond[0]
                dest = bond[1]

                col = bond_split_colors[i]

                # get the coords
```



```
                prox_pos = coords[orig]
                distal_pos = coords[dest]

                # compute a direction vector
                direction = distal_pos - prox_pos

                # and vector length. divide by 2 since split bond
                height = math.dist(prox_pos, distal_pos) / 2.0

                # the cylinder origins are actually in the
                # middle so we translate

                origin = prox_pos + 0.5 * direction # for a single plain
…bond
                origin1 = prox_pos + 0.25 * direction
                origin2 = prox_pos + 0.75 * direction

                bondradius = bradius

                if style == 'plain':
                    orig_col = dest_col = bcolor

                # proximal-distal red/green coloring
                elif style == 'pd':
                    if i <= 4 or i == 11 or i == 12:
                        orig_col = dest_col = 'red'
                    else:
                        orig_col = dest_col= 'green'
                    if i == 10:
                        orig_col = dest_col= 'yellow'
                else:
                    orig_col = ATOM_COLORS[col[0]]
                    dest_col = ATOM_COLORS[col[1]]

                if i >= 11: # prev and next residue atoms for phi/psi
…calcs
                    bondradius = bradius * .75 # make smaller to
…distinguish

                cap1 = pv.Sphere(center=prox_pos, radius=bondradius)
                cap2 = pv.Sphere(center=distal_pos, radius=bondradius)

                cyl = pv.Cylinder(origin, direction, radius=bondradius,
…height=height*2.0)
                cyl1 = pv.Cylinder(origin1, direction, radius=bondradius,
…height=height)
                cyl2 = pv.Cylinder(origin2, direction, radius=bondradius,
…height=height)
```

```
371
372                 if style == 'plain':
373                     pvp.add_mesh(cyl, color=orig_col)
374                 else:
375                     pvp.add_mesh(cyl1, color=orig_col)
376                     pvp.add_mesh(cyl2, color=dest_col)
377
378                 pvp.add_mesh(cap1, color=orig_col)
379                 pvp.add_mesh(cap2, color=dest_col)
380
381             return pvp # end draw_bonds
382
383         if style=='cpk':
384             i = 0
385             for atom in atoms:
386                 rad = ATOM_RADII_CPK[atom]
387                 pvp.add_mesh(pv.Sphere(center=coords[i], radius=rad),
    color=ATOM_COLORS[atom],
388                              smooth_shading=True, specular=spec,
    specular_power=specpow)
389                 i += 1
390
391         elif style=='cov':
392             i = 0
393             for atom in atoms:
394                 rad = ATOM_RADII_COVALENT[atom]
395                 pvp.add_mesh(pv.Sphere(center=coords[i], radius=rad),
    color=ATOM_COLORS[atom],
396                              smooth_shading=True, specular=spec,
    specular_power=specpow)
397                 i += 1
398
399         elif style == 'bs': # ball and stick
400             i = 0
401             for atom in atoms:
402                 rad = ATOM_RADII_CPK[atom] * bs_scale
403                 if i > 11:
404                     rad = rad * .75
405
406                 pvp.add_mesh(pv.Sphere(center=coords[i], radius=rad),
407                              color=ATOM_COLORS[atom], smooth_shading=True,
408                              specular=spec, specular_power=specpow)
409                 i += 1
410             pvp = draw_bonds(pvp, style='bs')
411
412         elif style == 'sb': # splitbonds
413             pvp = draw_bonds(pvp, style='sb', missing=missing_atoms)
414
```



```
415         elif style == 'pd': # proximal-distal
416             pvp = draw_bonds(pvp, style='pd', missing=missing_atoms)
417
418         else: # plain
419             pvp = draw_bonds(pvp, style='plain', bcolor=bondcolor,
420                              missing=missing_atoms)
421
422         return pvp
423
424     def display(self, single=True, style='sb'):
425         src = self.pdb_id
426         enrg = self.energy
427         title = f'{src}:
…  {self.proximal}{self.proximal_chain}-{self.distal}{self.distal_chain}:
…  {enrg:.2f} kcal/mol'
428
429         near_range, far_range = self.compute_extents()
430
431         if single == True:
432             _pl = pv.Plotter(window_size=WINSIZE)
433             _pl.add_title(title=title, font_size=FONTSIZE)
434             _pl.enable_anti_aliasing('msaa')
435             _pl.add_camera_orientation_widget()
436             _pl.view_isometric()
437             _pl = self._render(_pl, style=style,
438                                bs_scale=BS_SCALE, spec=SPECULARITY,
    specpow=SPEC_POWER)
439             _pl.reset_camera()
440             _pl.show()
441
442         else:
443             _WINSIZE = (1024, 1024)
444             pl = pv.Plotter(window_size=_WINSIZE, shape=(2,2))
445             pl.subplot(0,0)
446
447             #pl.add_axes()
448             pl.add_title(title=title, font_size=FONTSIZE)
449             pl.enable_anti_aliasing('msaa')
450
451             pl.add_camera_orientation_widget()
452             self._render(pl, style='cpk', bondcolor=BOND_COLOR,
453                                bs_scale=BS_SCALE, spec=SPECULARITY,
    specpow=SPEC_POWER)
454
455             pl.subplot(0,1)
456             pl.add_title(title=title, font_size=FONTSIZE)
457             self._render(pl, style='pd', bondcolor=BOND_COLOR,
458                                bs_scale=BS_SCALE, spec=SPECULARITY,
```

```
458…specpow=SPEC_POWER)
459             pl.view_isometric()
460
461             pl.subplot(1,0)
462             pl.add_title(title=title, font_size=FONTSIZE)
463             self._render(pl, style='bs', bondcolor=BOND_COLOR,
464                          bs_scale=BS_SCALE, spec=SPECULARITY,
…specpow=SPEC_POWER)
465             pl.view_isometric()
466
467             pl.subplot(1,1)
468             pl.add_title(title=title, font_size=FONTSIZE)
469             self._render(pl, style='sb', bondcolor=BOND_COLOR,
470                          bs_scale=BS_SCALE, spec=SPECULARITY,
…specpow=SPEC_POWER)
471             pl.view_isometric()
472
473             pl.link_views()
474             pl.reset_camera()
475             pl.show()
476         return
477
478     def screenshot(self, single=True, style='sb', fname='ssbond.png',
479                    verbose=False):
480         src = self.pdb_id
481         enrg = self.energy
482         title = f'{src}:
…{self.proximal}{self.proximal_chain}-{self.distal}{self.distal_chain}:
…{enrg:.2f} kcal/mol'
483
484         near_range, far_range = self.compute_extents()
485
486         if single:
487             print('entered')
488             pl = pv.Plotter(window_size=WINSIZE)
489             pl.add_title(title=title, font_size=FONTSIZE)
490             pl.enable_anti_aliasing('msaa')
491             pl.add_camera_orientation_widget()
492             pl = self._render(pl, style=style, bondcolor=BOND_COLOR,
493                          bs_scale=BS_SCALE, spec=SPECULARITY,
…specpow=SPEC_POWER)
494             pl.view_isometric()
495             pl.reset_camera()
496             pl.show(auto_close=False)
497             pl.screenshot(fname)
498             pl.clear()
499
500         else:
```



```
501             _WINSIZE = (1024, 1024)
502             pl = pv.Plotter(window_size=_WINSIZE, shape=(2,2))
503             pl.subplot(0,0)
504
505             pl.add_title(title=title, font_size=FONTSIZE)
506             pl.enable_anti_aliasing('msaa')
507
508             pl.add_camera_orientation_widget()
509             self._render(pl, style='cpk', bondcolor=BOND_COLOR,
510                          bs_scale=BS_SCALE, spec=SPECULARITY,
511                          specpow=SPEC_POWER)
512             pl.view_isometric()
513
514             pl.subplot(0,1)
515             pl.add_title(title=title, font_size=FONTSIZE)
516             self._render(pl, style='pd', bondcolor=BOND_COLOR,
517                          bs_scale=BS_SCALE, spec=SPECULARITY,
518                          specpow=SPEC_POWER)
519             pl.view_isometric()
520
521             pl.subplot(1,0)
522             pl.add_title(title=title, font_size=FONTSIZE)
523             self._render(pl, style='bs', bondcolor=BOND_COLOR,
524                          bs_scale=BS_SCALE, spec=SPECULARITY,
525                          specpow=SPEC_POWER)
526
527             pl.subplot(1,1)
528             pl.add_title(title=title, font_size=FONTSIZE)
529             self._render(pl, style='sb', bondcolor=BOND_COLOR,
530                          bs_scale=BS_SCALE, spec=SPECULARITY,
531                          specpow=SPEC_POWER)
532             pl.view_isometric()
533
534             pl.link_views()
535             pl.reset_camera()
536             pl.show(auto_close=False)
537             pl.screenshot(fname)
538
539         if verbose:
540             print(f'Saved: {fname}')
541
542     # comparison operators, used for sorting. keyed to SS bond energy
543     def __lt__(self, other):
544         if isinstance(other, Disulfide):
545             return self.energy < other.energy
546
547     def __le__(self, other):
548         if isinstance(other, Disulfide):
```

```
            return self.energy <= other.energy

    def __gt__(self, other):
        if isinstance(other, Disulfide):
            return self.energy > other.energy

    def __ge__(self, other):
        if isinstance(other, Disulfide):
            return self.energy >= other.energy

    def __eq__(self, other):
        if isinstance(other, Disulfide):
            return self.energy == other.energy

    def __ne__(self, other):
        if isinstance(other, Disulfide):
            return self.energy != other.energy

    # repr functions. The class is large, so I split it up into sections
    def repr_ss_info(self):
        """
        Representation for the Disulfide class
        """
        s1 = f'<Disulfide {self.name} SourceID: {self.pdb_id} Proximal:
{self.proximal} {self.proximal_chain} Distal: {self.distal}
{self.distal_chain}'
        return s1

    def repr_ss_coords(self):
        s2 = f'\nProximal Coordinates:\n   N: {self.n_prox}\n   Cα:
{self.ca_prox}\n   C: {self.c_prox}\n   O: {self.o_prox}\n   Cβ:
{self.cb_prox}\n   Sγ: {self.sg_prox}\n   Cprev {self.c_prev_prox}\n
Nnext: {self.n_next_prox}\n'
        s3 = f'Distal Coordinates:\n   N: {self.n_dist}\n   Cα:
{self.ca_dist}\n   C: {self.c_dist}\n   O: {self.o_dist}\n   Cβ:
{self.cb_dist}\n   Sγ: {self.sg_dist}\n   Cprev {self.c_prev_dist}\n
Nnext: {self.n_next_dist}\n\n'
        stot = f'{s2} {s3}'
        return stot

    def repr_ss_conformation(self):
        s4 = f'Conformation: (X1-X5):  {self.chi1:.3f}°, {self.chi2:.3f}°,
{self.chi3:.3f}°, {self.chi4:.3f}° {self.chi5:.3f}° '
        s5 = f'Energy: {self.energy:.3f} kcal/mol'
        stot = f'{s4} {s5}'
        return stot

    def repr_ss_local_coords(self):
```



```
        """
        Representation for the Disulfide class, internal coordinates.
        """
        s2i = f'Proximal Internal Coordinates:\n   N: {self._n_prox}\n
Cα: {self._ca_prox}\n   C: {self._c_prox}\n   O: {self._o_prox}\n   Cβ:
{self._cb_prox}\n   Sγ: {self._sg_prox}\n   Cprev {self.c_prev_prox}\n
Nnext: {self.n_next_prox}\n'
        s3i = f'Distal Internal Coordinates:\n   N: {self._n_dist}\n   Cα:
{self._ca_dist}\n   C: {self._c_dist}\n   O: {self._o_dist}\n   Cβ:
{self._cb_dist}\n   Sγ: {self._sg_dist}\n   Cprev {self.c_prev_dist}\n
Nnext: {self.n_next_dist}\n\n'
        stot = f'{s2i} {s3i}'
        return stot

    def repr_ss_chain_ids(self):
        return(f'Proximal Chain fullID: <{self.proximal_residue_fullid}>
Distal Chain fullID: <{self.distal_residue_fullid}>')

    def __repr__(self):
        """
        Representation for the Disulfide class
        """

        s1 = self.repr_ss_info()
        res = f'{s1}>'
        return res

    def pprint(self):
        """
        pretty print general info for the Disulfide
        """

        s1 = self.repr_ss_info()
        s4 = self.repr_ss_conformation()
        res = f'{s1} {s4}>'
        return res

    def pprint_all(self):
        """
        pretty print all info for a Disulfide
        """

        s1 = self.repr_ss_info() + '\n'
        s2 = self.repr_ss_coords()
        s3 = self.repr_ss_local_coords()
        s4 = self.repr_ss_conformation()
        s5 = self.repr_chain_ids()
        res = f'{s1} {s5} {s2} {s3} {s4} >'
```

```
629         print(res)
630
631     def _handle_SS_exception(self, message):
632         """Handle exception (PRIVATE).
633
634         This method catches an exception that occurs in the Disulfide
635         object (if PERMISSIVE), or raises it again, this time adding the
636         PDB line number to the error message.
637         """
638         # message = "%s at line %i." % (message)
639         message = f'{message}'
640
641         if self.PERMISSIVE:
642             # just print a warning - some residues/atoms may be missing
643             warnings.warn(
644                 "DisulfideConstructionException: %s\n"
645                 "Exception ignored.\n"
646                 "Some atoms may be missing in the data structure."
647                 % message,
648                 DisulfideConstructionWarning,
649             )
650         else:
651             # exceptions are fatal - raise again with new message
    … (including line nr)
652             raise DisulfideConstructionException(message) from None
653
654     def print_compact(self):
655         return(f'{self.repr_ss_info()} {self.repr_ss_conformation()}')
656
657     def repr_conformation(self):
658         return(f'{self.repr_ss_conformation()}')
659
660     def repr_coords(self):
661         return(f'{self.repr_ss_coords()}')
662
663     def repr_internal_coords(self):
664         return(f'{self.repr_ss_local_coords()}')
665
666     def repr_chain_ids(self):
667         return(f'{self.repr_ss_chain_ids()}')
668
669     def set_permissive(self, perm: bool) -> None:
670         self.PERMISSIVE = perm
671
672     def get_permissive(self) -> bool:
673         return self.PERMISIVE
674
675     def get_full_id(self):
```



```
676         return((self.proximal_residue_fullid, self.distal_residue_fullid))
677
678     def initialize_disulfide_from_chain(self, chain1, chain2, proximal,
679                                         distal, quiet=True):
680         '''
681         Initialize a new Disulfide object with atomic coordinates from the
    … proximal and
682         distal coordinates, typically taken from a PDB file.
683
684         Arguments:
685             chain1: list of Residues in the model, eg: chain = model['A']
686             chain2: list of Residues in the model, eg: chain = model['A']
687             proximal: proximal residue sequence ID
688             distal: distal residue sequence ID
689
690         Returns: none. The internal state is modified.
691         '''
692
693         id = chain1.get_full_id()[0]
694         self.pdb_id = id
695
696         chi1 = chi2 = chi3 = chi4 = chi5 = _ANG_INIT
697
698         prox = int(proximal)
699         dist = int(distal)
700
701         prox_residue = chain1[prox]
702         dist_residue = chain2[dist]
703
704         if (prox_residue.get_resname() != 'CYS' or
    … dist_residue.get_resname() != 'CYS'):
705             print(f'build_disulfide() requires CYS at both residues:
    … {prox} {prox_residue.get_resname()} {dist} {dist_residue.get_resname()}
    … Chain: {prox_residue.get_segid()}')
706
707         # set the objects proximal and distal values
708         self.set_resnum(proximal, distal)
709
710         self.proximal_chain = chain1.get_id()
711         self.distal_chain = chain2.get_id()
712
713         self.proximal_residue_fullid = prox_residue.get_full_id()
714         self.distal_residue_fullid = dist_residue.get_full_id()
715
716         if quiet:
717             warnings.filterwarnings("ignore",
    … category=DisulfideConstructionWarning)
718         else:
```

```
719             warnings.simplefilter("always")
720
721         # grab the coordinates for the proximal and distal residues as
    … vectors
722         # so we can do math on them later
723         # proximal residue
724
725         try:
726             n1 = prox_residue['N'].get_vector()
727             ca1 = prox_residue['CA'].get_vector()
728             c1 = prox_residue['C'].get_vector()
729             o1 = prox_residue['O'].get_vector()
730             cb1 = prox_residue['CB'].get_vector()
731             sg1 = prox_residue['SG'].get_vector()
732
733         except Exception:
734             raise DisulfideConstructionWarning(f"Invalid or missing
    … coordinates for proximal residue {proximal}") from None
735
736         # distal residue
737         try:
738             n2 = dist_residue['N'].get_vector()
739             ca2 = dist_residue['CA'].get_vector()
740             c2 = dist_residue['C'].get_vector()
741             o2 = dist_residue['O'].get_vector()
742             cb2 = dist_residue['CB'].get_vector()
743             sg2 = dist_residue['SG'].get_vector()
744
745         except Exception:
746             raise DisulfideConstructionWarning(f"Invalid or missing
    … coordinates for distal residue {distal}") from None
747
748         # previous residue and next residue - optional, used for phi, psi
    … calculations
749         try:
750             prevprox = chain1[prox-1]
751             nextprox = chain1[prox+1]
752
753             prevdist = chain2[dist-1]
754             nextdist = chain2[dist+1]
755
756             cprev_prox = prevprox['C'].get_vector()
757             nnext_prox = nextprox['N'].get_vector()
758
759             cprev_dist = prevdist['C'].get_vector()
760             nnext_dist = nextdist['N'].get_vector()
761
762             # compute phi, psi for prox and distal
```



```
763             self.phiprox = numpy.degrees(calc_dihedral(cprev_prox, n1,
    … ca1, c1))
764             self.psiprox = numpy.degrees(calc_dihedral(n1, ca1, c1,
    … nnext_prox))
765             self.phidist = numpy.degrees(calc_dihedral(cprev_dist, n2,
    … ca2, c2))
766             self.psidist = numpy.degrees(calc_dihedral(n2, ca2, c2,
    … nnext_dist))
767
768         except Exception:
769             mess = f'Missing coords for: {id} {prox-1} or {dist+1} for SS
    … {proximal}-{distal}'
770             cprev_prox = nnext_prox = cprev_dist = nnext_dist =
    … Vector(-1.0, -1.0, -1.0)
771             self.missing_atoms = True
772             warnings.warn(mess, DisulfideConstructionWarning)
773
774         # update the positions and conformation
775         self.set_positions(n1, ca1, c1, o1, cb1, sg1, n2, ca2, c2, o2,
    … cb2,
776                            sg2, cprev_prox, nnext_prox, cprev_dist,
    … nnext_dist)
777
778         # calculate and set the disulfide dihedral angles
779         self.chi1 = numpy.degrees(calc_dihedral(n1, ca1, cb1, sg1))
780         self.chi2 = numpy.degrees(calc_dihedral(ca1, cb1, sg1, sg2))
781         self.chi3 = numpy.degrees(calc_dihedral(cb1, sg1, sg2, cb2))
782         self.chi4 = numpy.degrees(calc_dihedral(sg1, sg2, cb2, ca2))
783         self.chi5 = numpy.degrees(calc_dihedral(sg2, cb2, ca2, n2))
784
785
786
787         self.ca_distance = distance3d(self.ca_prox, self.ca_dist)
788         self.torsion_array = numpy.array((self.chi1, self.chi2, self.chi3,
    … self.chi4,
789                                           self.chi5))
790
791         # calculate and set the SS bond torsional energy
792         self.compute_torsional_energy()
793
794         # compute and set the local coordinates
795         self.compute_local_coords()
796
797     def set_chain_id(self, chain_id):
798         self.chain_id = chain_id
799
800     def set_positions(self, n_prox: Vector, ca_prox: Vector, c_prox:
    … Vector,
```

```
                        o_prox: Vector, cb_prox: Vector, sg_prox: Vector,
                        n_dist: Vector, ca_dist: Vector, c_dist: Vector,
                        o_dist: Vector, cb_dist: Vector, sg_dist: Vector,
                        c_prev_prox: Vector, n_next_prox: Vector,
                        c_prev_dist: Vector, n_next_dist: Vector
                        ):
        '''
        Sets the atomic positions for all atoms in the disulfide bond.
        Arguments:
            n_prox
            ca_prox
            c_prox
            o_prox
            cb_prox
            sg_prox
            n_distal
            ca_distal
            c_distal
            o_distal
            cb_distal
            sg_distal
        Returns: None
        '''

        # deep copy
        self.n_prox = n_prox.copy()
        self.ca_prox = ca_prox.copy()
        self.c_prox = c_prox.copy()
        self.o_prox = o_prox.copy()
        self.cb_prox = cb_prox.copy()
        self.sg_prox = sg_prox.copy()
        self.sg_dist = sg_dist.copy()
        self.cb_dist = cb_dist.copy()
        self.ca_dist = ca_dist.copy()
        self.n_dist = n_dist.copy()
        self.c_dist = c_dist.copy()
        self.o_dist = o_dist.copy()

        self.c_prev_prox = c_prev_prox.copy()
        self.n_next_prox = n_next_prox.copy()
        self.c_prev_dist = c_prev_dist.copy()
        self.n_next_dist = n_next_dist.copy()

    def set_dihedrals(self, chi1, chi2, chi3, chi4, chi5):
        '''
        Sets the 5 dihedral angles chi1 - chi5 for the Disulfide object
and
        computes the torsional energy.
```



```

        Arguments: chi, chi2, chi3, chi4, chi5 - Dihedral angles in
degrees
        (-180 - 180) for the Disulfide conformation.

        Returns: None
        '''
        self.chi1 = chi1
        self.chi2 = chi2
        self.chi3 = chi3
        self.chi4 = chi4
        self.chi5 = chi5
        self.dihedrals = list([chi1, chi2, chi3, chi4, chi5])
        self.compute_torsional_energy()

    def set_name(self, namestr="Disulfide"):
        '''
        Sets the Disulfide's name
        Arguments: (str)namestr
        Returns: none
        '''

        self.name = namestr

    def set_resnum(self, proximal, distal):
        '''
        Sets the Proximal and Distal Residue numbers for the Disulfide
        Arguments:
            Proximal: Proximal residue number
            Distal: Distal residue number
        Returns: None
        '''

        self.proximal = proximal
        self.distal = distal

    def Distance_RMS(self, other):
        '''
        Calculate the RMS distance between the internal coordinates
        of self and another Disulfide
        '''
        ic1 = self.internal_coords()
        ic2 = other.internal_coords()

        totsq = 0.0
        for i in range(12):
            p1 = ic1[i]
            p2 = ic2[i]
```

```
895             totsq += math.dist(p1, p2)**2
896
897         totsq /= 12
898
899         return(math.sqrt(totsq))
900
901     def compute_torsional_energy(self):
902         '''
903         Compute the approximate torsional energy for the Disulfide's
    … conformation.
904         Arguments: chi1, chi2, chi3, chi4, chi5 - the dihedral angles for
    … the Disulfide
905         Returns: Energy (kcal/mol)
906         '''
907         # @TODO find citation for the ss bond energy calculation
908         chi1 = self.chi1
909         chi2 = self.chi2
910         chi3 = self.chi3
911         chi4 = self.chi4
912         chi5 = self.chi5
913
914         energy = 2.0 * (cos(torad(3.0 * chi1)) + cos(torad(3.0 * chi5)))
915         energy += cos(torad(3.0 * chi2)) + cos(torad(3.0 * chi4))
916         energy += 3.5 * cos(torad(2.0 * chi3)) + 0.6 * cos(torad(3.0 *
    … chi3)) + 10.1
917
918         self.energy = energy
919
920     def compute_local_coords(self):
921         """
922         Compute the internal coordinates for a properly initialized
    … Disulfide Object.
923
924         Arguments: SS initialized Disulfide object
925
926         Returns: None, modifies internal state of the input
927         """
928
929         turt = Turtle3D('tmp')
930         # get the coordinates as numpy.array for Turtle3D use.
931         cpp = self.c_prev_prox.get_array()
932         nnp = self.n_next_prox.get_array()
933
934         n = self.n_prox.get_array()
935         ca = self.ca_prox.get_array()
936         c = self.c_prox.get_array()
937         cb = self.cb_prox.get_array()
938         o = self.o_prox.get_array()
```



```
939         sg = self.sg_prox.get_array()
940
941         sg2 = self.sg_dist.get_array()
942         cb2 = self.cb_dist.get_array()
943         ca2 = self.ca_dist.get_array()
944         c2 = self.c_dist.get_array()
945         n2 = self.n_dist.get_array()
946         o2 = self.o_dist.get_array()
947
948         cpd = self.c_prev_dist.get_array()
949         nnd = self.n_next_dist.get_array()
950
951         turt.orient_from_backbone(n, ca, c, cb, ORIENT_SIDECHAIN)
952
953         # internal (local) coordinates, stored as Vector objects
954         # to_local returns numpy.array objects
955
956         self._n_prox = Vector(turt.to_local(n))
957         self._ca_prox = Vector(turt.to_local(ca))
958         self._c_prox = Vector(turt.to_local(c))
959         self._o_prox = Vector(turt.to_local(o))
960         self._cb_prox = Vector(turt.to_local(cb))
961         self._sg_prox = Vector(turt.to_local(sg))
962
963         self._c_prev_prox = Vector(turt.to_local(cpp))
964         self._n_next_prox = Vector(turt.to_local(nnp))
965         self._c_prev_dist = Vector(turt.to_local(cpd))
966         self._n_next_dist = Vector(turt.to_local(nnd))
967
968         self._n_dist = Vector(turt.to_local(n2))
969         self._ca_dist = Vector(turt.to_local(ca2))
970         self._c_dist = Vector(turt.to_local(c2))
971         self._o_dist = Vector(turt.to_local(o2))
972         self._cb_dist = Vector(turt.to_local(cb2))
973         self._sg_dist = Vector(turt.to_local(sg2))
974
975     def build_model(self, turtle: Turtle3D):
976         """
977         Build a model Disulfide based on the internal dihedral angles.
978         Routine assumes turtle is in orientation #1 (at Ca, headed toward
979         Cb, with N on left), builds disulfide, and updates the object's
    … internal
980         coordinate state. It also adds the distal protein backbone,
981         and computes the disulfide conformational energy.
982
983         Arguments: turtle: Turtle3D object properly oriented for the
    … build.
984         Returns: None. The Disulfide object's internal state is updated.
```

```
        """

        tmp = Turtle3D('tmp')
        tmp.copy_coords(turtle)

        n = Vector(0, 0, 0)
        ca = Vector(0, 0, 0)
        cb = Vector(0, 0, 0)
        c = Vector(0, 0, 0)

        self.ca_prox = tmp._position
        tmp.schain_to_bbone()
        n, ca, cb, c = build_residue(tmp)

        self.n_prox = n
        self.ca_prox = ca
        self.c_prox = c

        tmp.bbone_to_schain()
        tmp.move(1.53)
        tmp.roll(self.chi1)
        tmp.yaw(112.8)
        self.cb_prox = tmp._position

        tmp.move(1.86)
        tmp.roll(self.chi2)
        tmp.yaw(103.8)
        self.sg_prox = tmp._position

        tmp.move(2.044)
        tmp.roll(self.chi3)
        tmp.yaw(103.8)
        self.sg_dist = tmp._position

        tmp.move(1.86)
        tmp.roll(self.chi4)
        tmp.yaw(112.8)
        self.cb_dist = tmp._position

        tmp.move(1.53)
        tmp.roll(self.chi5)
        tmp.pitch(180.0)
        tmp.schain_to_bbone()
        n, ca, cb, c = build_residue(tmp)

        self.n_dist = n
        self.ca_dist = ca
        self.c_dist = c
```



```

        self.compute_torsional_energy()

# Class defination ends
def Torsion_RMS(ss1, ss2):
    '''
    Calculate the 5D Euclidean distance for 2 Disulfide torsion_vector
    objects. This is used to compare Disulfide Bond torsion angles to
    determine their torsional 'distance'.

    Arguments: p1, p2 Vector objects of dimensionality 5 (5D)
    Returns: Distance
    '''

    _p1 = ss1.torsion_array
    _p2 = ss2.torsion_array
    if (len(_p1) != 5 or len(_p2) != 5):
        raise ProteusPyWarning("--> distance5d() requires vectors of
length 5!")
    d = math.dist(_p1, _p2)
    return d

def Distance_RMS(ss1, ss2):
    '''
    Calculate the RMS distance between the internal coordinates between
    two Disulfides
    '''
    ic1 = ss1.internal_coords()
    ic2 = ss2.internal_coords()

    totsq = 0.0
    # only take coords for the proximal and distal disfulfides, not the
    # prev/next residues.

    for i in range(12):
        p1 = ic1[i]
        p2 = ic2[i]
        totsq += math.dist(p1, p2)**2

    totsq /= 12

    return(math.sqrt(totsq))

def distance3d(p1: Vector, p2: Vector):
    '''
    Calculate the 3D Euclidean distance for 2 Vector objects

    Arguments: p1, p2 Vector objects of dimensionality 3 (3D)
```

```
1080     Returns: Distance
1081     '''
1082     _p1 = p1.get_array()
1083     _p2 = p2.get_array()
1084     if (len(_p1) != 3 or len(_p2) != 3):
1085         raise ProteusPyWarning("--> distance3d() requires vectors of
   … length 3!")
1086     d = math.dist(_p1, _p2)
1087     return d
1088
1089 def name_to_id(fname: str):
1090     '''return an entry id for filename pdb1crn.ent -> 1crn'''
1091     ent = fname[3:-4]
1092     return ent
1093
1094 def torad(deg):
1095     return(numpy.radians(deg))
1096
1097 def todeg(rad):
1098     return(numpy.degrees(rad))
1099
1100 def parse_ssbond_header_rec(ssbond_dict: dict) -> list:
1101     '''
1102     Parse the SSBOND dict returned by parse_pdb_header.
1103     NB: Requires EGS-Modified BIO.parse_pdb_header.py
1104
1105     Arguments:
1106         ssbond_dict: the input SSBOND dict
1107     Returns: a list of tuples representing the proximal, distal residue
1108              ids for the disulfide.
1109
1110     '''
1111     disulfide_list = []
1112     for ssb in ssbond_dict.items():
1113         disulfide_list.append(ssb[1])
1114
1115     return disulfide_list
1116
1117 #
1118 # function reads a comma separated list of PDB IDs and download the
   … corresponding
1119 # .ent files to the PDB_DIR global.
1120 # Used to download the list of proteins containing at least one SS bond
1121 # with the ID list generated from: http://www.rcsb.org/
1122 #
1123
1124 def Download_Disulfides(pdb_home=PDB_DIR, model_home=MODEL_DIR,
1125                         verbose=False, reset=False) -> None:
```



```
1126     '''
1127     Function reads a comma separated list of PDB IDs and downloads them
1128     to the pdb_home path.
1129
1130     Used to download the list of proteins containing at least one SS bond
1131     with the ID list generated from: http://www.rcsb.org/
1132     '''
1133
1134     start = time.time()
1135     donelines = []
1136     SS_done = []
1137     ssfile = None
1138
1139     cwd = os.getcwd()
1140     os.chdir(pdb_home)
1141
1142     pdblist = PDBList(pdb=pdb_home, verbose=verbose)
1143     ssfilename = f'{model_home}{SS_ID_FILE}'
1144     print(ssfilename)
1145
1146     # list of IDs containing >1 SSBond record
1147     try:
1148         ssfile = open(ssfilename)
1149         Line = ssfile.readlines()
1150     except Exception:
1151         raise DisulfideIOException(f'Cannot open file: {ssfile}')
1152
1153     for line in Line:
1154         entries = line.split(',')
1155
1156     print(f'Found: {len(entries)} entries')
1157     completed = {'xxx'} # set to keep track of downloaded
1158
1159     # file to track already downloaded entries.
1160     if reset==True:
1161         completed_file = open(f'{model_home}ss_completed.txt', 'w')
1162         donelines = []
1163         SS_DONE = []
1164     else:
1165         completed_file = open(f'{model_home}ss_completed.txt', 'w+')
1166         donelines = completed_file.readlines()
1167
1168     if len(donelines) > 0:
1169         for dl in donelines[0]:
1170             # create a list of pdb id already downloaded
1171             SS_done = dl.split(',')
1172
1173     count = len(SS_done) - 1
```

```
1174        completed.update(SS_done) # update the completed set with what's
    … downloaded
1175
1176        # Loop over all entries,
1177        pbar = tqdm(entries, ncols=_PBAR_COLS)
1178        for entry in pbar:
1179            pbar.set_postfix({'Entry': entry})
1180            if entry not in completed:
1181                if pdblist.retrieve_pdb_file(entry, file_format='pdb',
    … pdir=pdb_home):
1182                    completed.update(entry)
1183                    completed_file.write(f'{entry},')
1184                    count += 1
1185
1186        completed_file.close()
1187
1188        end = time.time()
1189        elapsed = end - start
1190
1191        print(f'Overall files processed: {count}')
1192        print(f'Complete. Elapsed time: {datetime.timedelta(seconds=elapsed)}
    … (h:m:s)')
1193        os.chdir(cwd)
1194        return
1195
1196    def build_torsion_df(SSList: DisulfideList) -> pd.DataFrame:
1197        # create a dataframe with the following columns for the disulfide
1198        # conformations extracted from the structure
1199
1200        SS_df = pd.DataFrame(columns=Torsion_DF_Cols)
1201
1202        pbar = tqdm(SSList, ncols=_PBAR_COLS, miniters=400000)
1203        for ss in pbar:
1204            #pbar.set_postfix({'ID': ss.name}) # update the progress bar
1205
1206            new_row = [ss.pdb_id, ss.name, ss.proximal, ss.distal, ss.chi1,
    … ss.chi2,
1207                    ss.chi3, ss.chi4, ss.chi5, ss.energy, ss.ca_distance,
1208                    ss.psiprox, ss.psiprox, ss.phidist, ss.psidist]
1209            # add the row to the end of the dataframe
1210            SS_df.loc[len(SS_df.index)] = new_row.copy() # deep copy
1211
1212        return SS_df.copy()
1213
1214    def Extract_Disulfides(numb=-1, verbose=False, quiet=True, pdbdir=PDB_DIR,
1215                           modeldir=MODEL_DIR, picklefile=SS_PICKLE_FILE,
1216                           torsionfile=SS_TORSIONS_FILE,
1217                           problemfile=PROBLEM_ID_FILE,
```



```
1218                           dictfile=SS_DICT_PICKLE_FILE) -> None:
1219        '''
1220        This function creates .pkl files needed for the DisulfideLoader class.
1221        The Disulfide objects are contained in a DisulfideList object and
1222        Dict within these files. In addition, .csv files containing all of
1223        the torsions for the disulfides and problem IDs are written.
1224
1225        Arguments:
1226            numb:           number of entries to process, defaults to all
1227            verbose:        more messages
1228            quiet:          turns of DisulfideConstruction warnings
1229            pdbdir:         path to PDB files
1230            modeldir:       path to resulting .pkl files
1231            picklefile:     name of the disulfide .pkl file
1232            torsionfile:    name of the disulfide torsion file .csv created
1233            problemfile:    name of the .csv file containing problem ids
1234            dictfile:       name of the .pkl file
1235
1236        Example:
1237            from proteusPy.Disulfide import Extract_Disulfides,
    … DisulfideLoader, DisulfideList
1238
1239            Extract_Disulfides(numb=500, pdbdir=PDB_DIR, verbose=False,
    … quiet=True)
1240
1241            SS1 = DisulfideList([],'All_SS')
1242            SS2 = DisulfideList([], '4yys')
1243
1244            PDB_SS = DisulfideLoader()
1245            SS1 = PDB_SS[0]       <-- returns a Disulfide object at index 0
1246            SS2 = PDB_SS['4yys']  <-- returns a DisulfideList containing
    … disulfides
1247                                      for 4yys
1248            SS3 = PDB_SS[:10]     <-- returns a DisulfideList containing the
    … slice
1249        '''
1250
1251        entrylist = []
1252        problem_ids = []
1253        bad = 0
1254
1255        # we use the specialized list class DisulfideList to contain our
    … disulfides
1256        # we'll use a dict to store DisulfideList objects, indexed by the
    … structure ID
1257        All_ss_dict = {}
1258        All_ss_list = []
1259
```

```
    start = time.time()
    cwd = os.getcwd()

    # Build a list of PDB files in PDB_DIR that are readable. These files were downloaded
    # via the RCSB web query interface for structures containing >= 1 SS Bond.

    os.chdir(pdbdir)

    ss_filelist = glob.glob(f'*.ent')
    tot = len(ss_filelist)

    if verbose:
        print(f'PDB Directory {pdbdir} contains: {tot} files')

    # the filenames are in the form pdb{entry}.ent, I loop through them and extract
    # the PDB ID, with Disulfide.name_to_id(), then add to entrylist.

    for entry in ss_filelist:
        entrylist.append(name_to_id(entry))

    # create a dataframe with the following columns for the disulfide conformations
    # extracted from the structure

    SS_df = pd.DataFrame(columns=Torsion_DF_Cols)

    # define a tqdm progressbar using the fully loaded entrylist list.
    # If numb is passed then
    # only do the last numb entries.

    if numb > 0:
        pbar = tqdm(entrylist[:numb], ncols=_PBAR_COLS)
    else:
        pbar = tqdm(entrylist, ncols=_PBAR_COLS)

    # loop over ss_filelist, create disulfides and initialize them
    for entry in pbar:
        pbar.set_postfix({'ID': entry, 'Bad': bad}) # update the progress bar

        # returns an empty list if none are found.
        sslist = DisulfideList([], entry)
        sslist = load_disulfides_from_id(entry, model_numb=0, verbose=verbose,
                                         quiet=quiet, pdb_dir=pdbdir)
```



```
        if len(sslist) > 0:
            for ss in sslist:
                All_ss_list.append(ss)
                new_row = [ss.pdb_id, ss.name, ss.proximal, ss.distal,
                           ss.chi1, ss.chi2, ss.chi3, ss.chi4, ss.chi5,
                           ss.energy, ss.ca_distance, ss.phiprox,
                           ss.psiprox, ss.phidist, ss.psidist]

                # add the row to the end of the dataframe
                SS_df.loc[len(SS_df.index)] = new_row.copy() # deep copy
            All_ss_dict[entry] = sslist
        else:
            # at this point I really shouldn't have any bad non-parsible file
            bad += 1
            problem_ids.append(entry)
            os.remove(f'pdb{entry}.ent')

    if bad > 0:
        prob_cols = ['id']
        problem_df = pd.DataFrame(columns=prob_cols)
        problem_df['id'] = problem_ids

        print(f'Found and removed: {len(problem_ids)} problem structures.')
        print(f'Saving problem IDs to file: {modeldir}{problemfile}')

        problem_df.to_csv(f'{modeldir}{problemfile}')
    else:
        if verbose:
            print('No problems found.')

    # dump the all_ss array of disulfides to a .pkl file. ~520 MB.
    fname = f'{modeldir}{picklefile}'
    print(f'Saving {len(All_ss_list)} Disulfides to file: {fname}')

    with open(fname, 'wb+') as f:
        pickle.dump(All_ss_list, f)

    # dump the all_ss array of disulfides to a .pkl file. ~520 MB.
    dict_len = len(All_ss_dict)
    fname = f'{modeldir}{dictfile}'

    print(f'Saving {len(All_ss_dict)} Disulfide-containing PDB IDs to file: {fname}')

    with open(fname, 'wb+') as f:
        pickle.dump(All_ss_dict, f)
```

```
1347
1348        # save the torsions
1349        fname = f'{modeldir}{torsionfile}'
1350        print(f'Saving torsions to file: {fname}')
1351
1352        SS_df.to_csv(fname)
1353
1354        end = time.time()
1355        elapsed = end - start
1356
1357        print(f'Disulfide Extraction complete! Elapsed time:\
1358             {datetime.timedelta(seconds=elapsed)} (h:m:s)')
1359
1360        # return to original directory
1361        os.chdir(cwd)
1362        return
1363
1364    # NB - this only works with the EGS modified version of
      …  BIO.parse_pdb_header.py
1365    def load_disulfides_from_id(struct_name: str,
1366                                pdb_dir = '.',
1367                                model_numb = 0,
1368                                verbose = False,
1369                                quiet=False,
1370                                dbg = False) -> list:
1371        '''
1372        Loads all Disulfides by PDB ID and initializes the Disulfide objects.
1373        Assumes the file is downloaded in the pdb_dir path.
1374
1375        NB: Requires EGS-Modified BIO.parse_pdb_header.py
1376
1377        Arguments:
1378            struct_name: the name of the PDB entry.
1379
1380            pdb_dir: path to the PDB files, defaults to PDB_DIR
1381
1382            model_numb: model number to use, defaults to 0 for single
1383            structure files.
1384
1385            verbose: print info while parsing
1386
1387        Returns: a list of Disulfide objects initialized from the file.
1388        Example:
1389          Assuming the PDB_DIR has the pdb5rsa.ent file in place calling:
1390
1391          SS_list = []
1392          SS_list = load_disulfides_from_id('5rsa', verbose=True)
1393
```



```
1394          loads the Disulfides from the file and initialize the disulfide
      …  objects, returning
1395          them in the result. '''
1396
1397        i = 1
1398        proximal = distal = -1
1399        SSList = DisulfideList([], struct_name)
1400        _chaina = None
1401        _chainb = None
1402
1403        parser = PDBParser(PERMISSIVE=True)
1404
1405        # Biopython uses the Structure -> Model -> Chain hierarchy to organize
1406        # structures. All are iterable.
1407
1408        structure = parser.get_structure(struct_name,
      …  file=f'{pdb_dir}pdb{struct_name}.ent')
1409        model = structure[model_numb]
1410
1411        if verbose:
1412            print(f'-> load_disulfide_from_id() - Parsing structure:
      …  {struct_name}:')
1413
1414        ssbond_dict = structure.header['ssbond'] # NB: this requires the
      …  modified code
1415
1416        # list of tuples with (proximal distal chaina chainb)
1417        ssbonds = parse_ssbond_header_rec(ssbond_dict)
1418
1419        with warnings.catch_warnings():
1420            if quiet:
1421                #warnings.filterwarnings("ignore",
      …  category=DisulfideConstructionWarning)
1422                warnings.filterwarnings("ignore")
1423            for pair in ssbonds:
1424                # in the form (proximal, distal, chain)
1425                proximal = pair[0]
1426                distal = pair[1]
1427                chain1_id = pair[2]
1428                chain2_id = pair[3]
1429
1430                if not proximal.isnumeric() or not distal.isnumeric():
1431                    mess = f' -> Cannot parse SSBond record (non-numeric
      …  IDs):\
1432                     {struct_name} Prox: {proximal} {chain1_id} Dist: {distal}
      …  {chain2_id}, ignoring.'
1433                    warnings.warn(mess, DisulfideConstructionWarning)
1434                    continue
```

```
            else:
                proximal = int(proximal)
                distal = int(distal)

            if proximal == distal:
                mess = f' -> Cannot parse SSBond record (proximal ==
distal):\
                 {struct_name} Prox: {proximal} {chain1_id} Dist: {distal}
{chain2_id}, ignoring.'
                warnings.warn(mess, DisulfideConstructionWarning)
                continue

            _chaina = model[chain1_id]
            _chainb = model[chain2_id]

            if (_chaina is None) or (_chainb is None):
                mess = f' -> NULL chain(s): {struct_name}: {proximal}
{chain1_id}\
                 - {distal} {chain2_id}, ignoring!'
                warnings.warn(mess, DisulfideConstructionWarning)
                continue

            if (chain1_id != chain2_id):
                if verbose:
                    mess = (f' -> Cross Chain SS for: Prox: {proximal}
{chain1_id}\
                     Dist: {distal} {chain2_id}')
                    warnings.warn(mess, DisulfideConstructionWarning)
                    pass # was break

            try:
                prox_res = _chaina[proximal]
                dist_res = _chainb[distal]

            except KeyError:
                mess = f'Cannot parse SSBond record (KeyError):
{struct_name} Prox:\
                  {proximal} {chain1_id} Dist: {distal} {chain2_id},
ignoring!'
                warnings.warn(mess, DisulfideConstructionWarning)
                continue

            # make a new Disulfide object, name them based on proximal and
distal
            # initialize SS bond from the proximal, distal coordinates

            if _chaina[proximal].is_disordered() or
_chainb[distal].is_disordered():
```



```
                mess = f'Disordered chain(s): {struct_name}: {proximal}
{chain1_id}\
                 - {distal} {chain2_id}, ignoring!'
                warnings.warn(mess, DisulfideConstructionWarning)
                continue
            else:
                if verbose:
                    print(f' -> SSBond: {i}: {struct_name}: {proximal}
{chain1_id}\
                     - {distal} {chain2_id}')
                ssbond_name =
f'{struct_name}_{proximal}{chain1_id}_{distal}{chain2_id}'
                new_ss = Disulfide(ssbond_name)
                new_ss.initialize_disulfide_from_chain(_chaina, _chainb,
proximal,
                 distal, quiet=quiet)
                SSList.append(new_ss)
        i += 1
    return SSList

def check_header_from_file(filename: str, model_numb = 0,
                           verbose = False, dbg = False) -> bool:

    '''
    Loads all Disulfides by PDB ID and initializes the Disulfide objects.
    Assumes the file is downloaded in the pdb_dir path.

    NB: Requires EGS-Modified BIO.parse_pdb_header.py

    Arguments:
        struct_name: the name of the PDB entry.

        pdb_dir: path to the PDB files, defaults to PDB_DIR

        model_numb: model number to use, defaults to 0 for single
        structure files.

        verbose: print info while parsing

    Returns: a list of Disulfide objects initialized from the file.
    Example:
      Assuming the PDB_DIR has the pdb5rsa.ent file in place calling:

      SS_list = []
      SS_list = load_disulfides_from_id('5rsa', verbose=True)

      loads the Disulfides from the file and initialize the disulfide
objects, returning
```

```
1518        them in the result. '''
1519
1520      i = 1
1521      proximal = distal = -1
1522      SSList = []
1523      _chaina = None
1524      _chainb = None
1525
1526      parser = PDBParser(PERMISSIVE=True)
1527
1528      # Biopython uses the Structure -> Model -> Chain hierarchy to organize
1529      # structures. All are iterable.
1530
1531      structure = parser.get_structure('tmp', file=filename)
1532      struct_name = structure.get_id()
1533
1534      model = structure[model_numb]
1535
1536      if verbose:
1537          print(f'-> check_header_from_file() - Parsing file: {filename}:')
1538
1539      ssbond_dict = structure.header['ssbond'] # NB: this requires the
…  modified code
1540
1541      # list of tuples with (proximal distal chaina chainb)
1542      ssbonds = parse_ssbond_header_rec(ssbond_dict)
1543
1544      for pair in ssbonds:
1545          # in the form (proximal, distal, chain)
1546          proximal = pair[0]
1547          distal = pair[1]
1548
1549          if not proximal.isnumeric() or not distal.isnumeric():
1550              if verbose:
1551                  mess = f' ! Cannot parse SSBond record (non-numeric IDs):\
1552                   {struct_name} Prox:  {proximal} {chain1_id} Dist:
…  {distal} {chain2_id}'
1553                  warnings.warn(mess, DisulfideParseWarning)
1554              continue # was pass
1555          else:
1556              proximal = int(proximal)
1557              distal = int(distal)
1558
1559          chain1_id = pair[2]
1560          chain2_id = pair[3]
1561
1562          _chaina = model[chain1_id]
1563          _chainb = model[chain2_id]
```



```
1564
1565          if (chain1_id != chain2_id):
1566              if verbose:
1567                  mess = f' -> Cross Chain SS for: Prox: {proximal}
…  {chain1_id} Dist:\
1568                       {distal} {chain2_id}'
1569                  warnings.warn(mess, DisulfideParseWarning)
1570                  pass # was break
1571
1572          try:
1573              prox_res = _chaina[proximal]
1574              dist_res = _chainb[distal]
1575          except KeyError:
1576              print(f' ! Cannot parse SSBond record (KeyError):
…  {struct_name} Prox:\
1577                <{proximal}> {chain1_id} Dist: <{distal}> {chain2_id}')
1578              continue
1579
1580          # make a new Disulfide object, name them based on proximal and
…  distal
1581          # initialize SS bond from the proximal, distal coordinates
1582          if (_chaina is not None) and (_chainb is not None):
1583              if _chaina[proximal].is_disordered() or
…  _chainb[distal].is_disordered():
1584                  continue
1585              else:
1586                  if verbose:
1587                     print(f' -> SSBond: {i}: {struct_name}: {proximal}
…  {chain1_id}\
1588                      - {distal} {chain2_id}')
1589          else:
1590              if dbg:
1591                  print(f' -> NULL chain(s): {struct_name}: {proximal}
…  {chain1_id}\
1592                   - {distal} {chain2_id}')
1593          i += 1
1594      return True
1595
1596  def check_header_from_id(struct_name: str, pdb_dir='.', model_numb=0,
1597                           verbose=False, dbg=False) -> bool:
1598      '''
1599      Loads all Disulfides by PDB ID and initializes the Disulfide objects.
1600      Assumes the file is downloaded in the pdb_dir path.
1601
1602      NB: Requires EGS-Modified BIO.parse_pdb_header.py
1603
1604      Arguments:
1605          struct_name: the name of the PDB entry.
```

```
1606
1607        pdb_dir: path to the PDB files, defaults to PDB_DIR
1608
1609        model_numb: model number to use, defaults to 0 for single
1610        structure files.
1611
1612        verbose: print info while parsing
1613
1614    Returns: True if the proximal and distal residues are CYS and there
1615             are no cross-chain SS bonds
1616
1617    Example:
1618      Assuming the PDB_DIR has the pdb5rsa.ent file in place calling:
1619
1620      SS_list = []
1621      goodfile = check_header_from_id('5rsa', verbose=True)
1622
1623    '''
1624
1625    parser = PDBParser(PERMISSIVE=True, QUIET=True)
1626    structure = parser.get_structure(struct_name,
1627
…  file=f'{pdb_dir}pdb{struct_name}.ent')
1628    model = structure[0]
1629
1630    ssbond_dict = structure.header['ssbond'] # NB: this requires the
…  modified code
1631
1632    bondlist = []
1633    i = 0
1634
1635    # get a list of tuples containing the proximal, distal residue IDs for
1636    # all SSBonds in the chain.
1637    bondlist = parse_ssbond_header_rec(ssbond_dict)
1638
1639    if len(bondlist) == 0:
1640        if (verbose):
1641            print(f'-> check_header_from_id(): no bonds found in
…  bondlist.')
1642        return False
1643
1644    for pair in bondlist:
1645        # in the form (proximal, distal, chain)
1646        proximal = pair[0]
1647        distal = pair[1]
1648        chain1 = pair[2]
1649        chain2 = pair[3]
1650
```



```
1651        chaina = model[chain1]
1652        chainb = model[chain2]
1653
1654        try:
1655            prox_residue = chaina[proximal]
1656            dist_residue = chainb[distal]
1657
1658            prox_residue.disordered_select("CYS")
1659            dist_residue.disordered_select("CYS")
1660
1661            if prox_residue.get_resname() != 'CYS' or
…  dist_residue.get_resname() != 'CYS':
1662                if (verbose):
1663                    print(f'build_disulfide() requires CYS at both
…  residues:\
1664                     {prox_residue.get_resname()}
…  {dist_residue.get_resname()}')
1665                return False
1666        except KeyError:
1667            if (dbg):
1668                print(f'Keyerror: {struct_name}: {proximal} {chain1} -
…  {distal} {chain2}')
1669                return False
1670
1671        if verbose:
1672            print(f' -> SSBond: {i}: {struct_name}: {proximal} {chain1} -
…  {distal}\
1673             {chain2}')
1674
1675        i += 1
1676    return True
1677
1678def Check_chains(pdbid, pdbdir, verbose=True):
1679    '''Returns True if structure has multiple chains of identical length,\
1680     False otherwise'''
1681
1682    parser = PDBParser(PERMISSIVE=True)
1683    structure = parser.get_structure(pdbid,
…  file=f'{pdbdir}pdb{pdbid}.ent')
1684
1685    # dictionary of tuples with SSBond prox and distal
1686    ssbond_dict = structure.header['ssbond']
1687
1688    if verbose:
1689        print(f'ssbond dict: {ssbond_dict}')
1690
1691    same = False
1692    model = structure[0]
```

```
1693     chainlist = model.get_list()
1694
1695     if len(chainlist) > 1:
1696         chain_lens = []
1697         if verbose:
1698             print(f'multiple chains. {chainlist}')
1699         for chain in chainlist:
1700             chain_length = len(chain.get_list())
1701             chain_id = chain.get_id()
1702             if verbose:
1703                 print(f'Chain: {chain_id}, length: {chain_length}')
1704             chain_lens.append(chain_length)
1705
1706         if numpy.min(chain_lens) != numpy.max(chain_lens):
1707             same = False
1708             if verbose:
1709                 print(f'chain lengths are unequal: {chain_lens}')
1710         else:
1711             same = True
1712             if verbose:
1713                 print(f'Chains are equal length, assuming the same.
…{chain_lens}')
1714     return(same)
1715
1716 # End of file
1717
```