```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: 12/18/22
# Cα Cβ Sγ

import proteusPy
from proteusPy import *

from proteusPy.DisulfideExceptions import *
from proteusPy.DisulfideGlobals import *
from proteusPy.proteusGlobals import *
#from proteusPy.disulfide import Disulfide

from Bio.PDB import Select, Vector, PDBParser
from Bio.PDB.vectors import calc_dihedral
import math

import pyvista as pv

# float init for class
_FLOAT_INIT = -999.9

# tqdm progress bar width
_PBAR_COLS = 100

#
#
class DisulfideList(UserList):
    '''
    Class provides a sortable list for Disulfide objects.
    Indexing and slicing are supported, and normal list operations like
.insert, .append and .extend.
    The DisulfideList object must be initialized with an iterable (tuple,
list) and a name.

    Example:
        from proteusPy.disulfide import DisulfideList, Disulfide,
DisulfideLoader

        # make some empty disulfides
        ss1 = Disulfide('ss1')
        ss2 = Disulfide('ss2')

        # make a DisulfideList containing ss1, named 'tmp'
        sslist = DisulfideList([ss1], 'tmp')
```



```python
        sslist.append(ss2)

        # load the PDB Disulfide database
        PDB_SS = None
        PDB_SS = DisulfideLoader(verbose=True, modeldir=MODELS)

        # extract a disulfide with typical index
        ss1 = PDB_SS[0]
        print(f'{ss1.pprint_all()}')

        # grab a subset via slicing
        subset = DisulfideList(PDB_SS[0:10],'subset')


    '''
    def __init__(self, iterable, id):
        self.pdb_id = id
        super().__init__(self.validate_ss(item) for item in iterable)

    def __getitem__(self, item):
        if isinstance(item, slice):
            indices = range(*item.indices(len(self.data)))
            name = self.data[0].pdb_id
            sublist = [self.data[i] for i in indices]
            return DisulfideList(sublist, name)
        return UserList.__getitem__(self, item)

    def __setitem__(self, index, item):
        self.data[index] = self.validate_ss(item)

    def insert(self, index, item):
        self.data.insert(index, self.validate_ss(item))

    def append(self, item):
        self.data.append(self.validate_ss(item))

    def extend(self, other):
        if isinstance(other, type(self)):
            self.data.extend(other)
        else:
            self.data.extend(self._validate_ss(item) for item in other)

    def validate_ss(self, value):
        if isinstance(value, (proteusPy.disulfide.Disulfide)):
            return value
        raise TypeError(f"Disulfide object expected, got
{type(value).__name__}")
```

```
    def set_id(self, value):
        self.pdb_id = value

    def get_id(self):
        return self.pdb_id

# Class definition for a Disulfide bond.
class Disulfide:
    """
    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_vector = numpy.array((_FLOAT_INIT, _FLOAT_INIT, _FLOAT_INIT, _FLOAT_INIT, _FLOAT_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)
        self.c_dist = Vector(0,0,0)
        self.o_dist = Vector(0,0,0)

        # local coordinates for the Disulfide, computed using the Turtle3D in
        # Orientation #1 these are generally private.

        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)
        self._c_dist = Vector(0,0,0)
        self._o_dist = Vector(0,0,0)

        # Dihedral angles for the disulfide bond itself, set to _FLOAT_INIT
        self.chi1 = _FLOAT_INIT
        self.chi2 = _FLOAT_INIT
        self.chi3 = _FLOAT_INIT
        self.chi4 = _FLOAT_INIT
        self.chi5 = _FLOAT_INIT

        # I initialize an array for the torsions which will be used for comparisons
        self.dihedrals = numpy.array((_FLOAT_INIT, _FLOAT_INIT, _FLOAT_INIT, _FLOAT_INIT, _FLOAT_INIT), "d")

    def internal_coords(self) -> numpy.array:
        res_array = numpy.zeros(shape=(6,3))

        res_array = numpy.array((
            self._n_prox.get_array(),
```

```
            self._ca_prox.get_array(),
            self._c_prox.get_array(),
            self._o_prox.get_array(),
            self._cb_prox.get_array(),
            self._sg_prox.get_array(),
            self._n_dist.get_array(),
            self._ca_dist.get_array(),
            self._c_dist.get_array(),
            self._o_dist.get_array(),
            self._cb_dist.get_array(),
            self._sg_dist.get_array(),
        ))
        return res_array

    def internal_coords_res(self, resnumb) -> numpy.array:
        res_array = numpy.zeros(shape=(6,3))

        if resnumb == self.proximal:
            res_array = numpy.array((
                self._n_prox.get_array(),
                self._ca_prox.get_array(),
                self._c_prox.get_array(),
                self._o_prox.get_array(),
                self._cb_prox.get_array(),
                self._sg_prox.get_array(),
            ))
            return res_array
        elif resnumb == self.distal:
            res_array = numpy.array((
                self._n_dist.get_array(),
                self._ca_dist.get_array(),
                self._c_dist.get_array(),
                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 reset(self):
        self.__init__(self)

    def render(self, pvp=pv.Plotter(), cpk=False, bs_scale=.2, spec=.8,
specpow=6):
        ''' Update the passed pyVista plotter() object with the mesh data
```



```
for the input Disulfide Bond
            Arguments:
                pvp: pyvista.Plotter() object
                cpk: Whether to render as CPK object or ball-and-stick
(bool)
        '''

        cyl_radius = .6 * bs_scale
        coords = self.internal_coords()
        atoms = ('N', 'C', 'C', 'O', 'C', 'SG', 'N', 'C', 'C', 'O', 'C',
'SG')

        newplt = pvp.copy()
        # bond connection table with atoms in the specific order shown
above:
        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
            ])

        if cpk:
            i = 0
            for atom in atoms:
                rad = ATOM_RADII_CPK[atom]
                newplt.add_mesh(pv.Sphere(center=coords[i], radius=rad),
color=ATOM_COLORS[atom], smooth_shading=True, specular=spec,
specular_power=specpow)
                i += 1
        else: # ball and stick
            i = 0
            for atom in atoms:
                rad = ATOM_RADII_CPK[atom] * bs_scale
                sphr = pv.Sphere(center=coords[i], radius=rad)
                newplt.add_mesh(sphr, color=ATOM_COLORS[atom],
smooth_shading=True, specular=spec, specular_power=specpow)
                i += 1
            # now do the bonds.
            for i in range(len(bond_conn)):
```

```
                bond = bond_conn[i]
                orig = bond[0]
                dest = bond[1]
                prox_pos = coords[orig]
                distal_pos = coords[dest]
                direction = distal_pos - prox_pos
                height = math.dist(prox_pos, distal_pos)
                origin = prox_pos + 0.5 * direction # the cylinder origin is actually in the middle so we translate
                cyl = pv.Cylinder(origin, direction, radius=cyl_radius, height=height)
                newplt.add_mesh(cyl)
        return newplt

    # comparison operators, used for sorting. keyed to SS bond energy
    def __lt__(self, other):
        if isinstance(other, Disulfide):
            return self.energy < other.energy

    def __le__(self, other):
        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\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\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\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'
        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
        """
```

```
344
345         s1 = self.repr_ss_info()
346         s4 = self.repr_ss_conformation()
347         res = f'{s1} {s4}>'
348         return res
349
350     def pprint_all(self):
351         """
352         pretty print all info for a Disulfide
353         """
354
355         s1 = self.repr_ss_info() + '\n'
356         s2 = self.repr_ss_coords()
357         s3 = self.repr_ss_local_coords()
358         s4 = self.repr_ss_conformation()
359         s5 = self.repr_chain_ids()
360         res = f'{s1} {s5} {s2} {s3} {s4} >'
361         return res
362
363     def _handle_SS_exception(self, message):
364         """Handle exception (PRIVATE).
365
366         This method catches an exception that occurs in the Disulfide
367         object (if PERMISSIVE), or raises it again, this time adding the
368         PDB line number to the error message.
369         """
370         # message = "%s at line %i." % (message)
371         message = f'{message}'
372
373         if self.PERMISSIVE:
374             # just print a warning - some residues/atoms may be missing
375             warnings.warn(
376                 "DisulfideConstructionException: %s\n"
377                 "Exception ignored.\n"
378                 "Some atoms may be missing in the data structure."
379                 % message,
380                 DisulfideConstructionWarning,
381             )
382         else:
383             # exceptions are fatal - raise again with new message
… (including line nr)
384             raise DisulfideConstructionException(message) from None
385
386     def print_compact(self):
387         return(f'{self.repr_ss_info()} {self.repr_ss_conformation()}')
388
389     def repr_conformation(self):
390         return(f'{self.repr_ss_conformation()}')
```



```
391
392     def repr_coords(self):
393         return(f'{self.repr_ss_coords()}')
394
395     def repr_internal_coords(self):
396         return(f'{self.repr_ss_local_coords()}')
397
398     def repr_chain_ids(self):
399         return(f'{self.repr_ss_chain_ids()}')
400
401     def set_permissive(self, perm: bool) -> None:
402         self.PERMISSIVE = perm
403
404     def get_permissive(self) -> bool:
405         return self.PERMISIVE
406
407     def set_quiet(self, perm: bool) -> None:
408         self.QUIET = perm
409
410     def get_quiet(self) -> bool:
411         return self.QUIET
412
413     def get_full_id(self):
414         return((self.proximal_residue_fullid, self.distal_residue_fullid))
415
416     def initialize_disulfide_from_chain(self, chain1, chain2, proximal,
… distal):
417         '''
418         Initialize a new Disulfide object with atomic coordinates from the
… proximal and
419         distal coordinates, typically taken from a PDB file.
420
421         Arguments:
422             chain1: list of Residues in the model, eg: chain = model['A']
423             chain2: list of Residues in the model, eg: chain = model['A']
424             proximal: proximal residue sequence ID
425             distal: distal residue sequence ID
426
427         Returns: none. The internal state is modified.
428         '''
429
430         id = chain1.get_full_id()[0]
431
432         self.pdb_id = id
433
434         # create a new Disulfide object
435         chi1 = chi2 = chi3 = chi4 = chi5 = _FLOAT_INIT
436
```

```
        prox = int(proximal)
        dist = int(distal)

        prox_residue = chain1[prox]
        dist_residue = chain2[dist]

        if (prox_residue.get_resname() != 'CYS' or
dist_residue.get_resname() != 'CYS'):
            print(f'build_disulfide() requires CYS at both residues:
{prox} {prox_residue.get_resname()} {dist} {dist_residue.get_resname()}
Chain: {prox_residue.get_segid()}')

        # set the objects proximal and distal values
        self.set_resnum(proximal, distal)

        self.proximal_chain = chain1.get_id()
        self.distal_chain = chain2.get_id()

        self.proximal_residue_fullid = prox_residue.get_full_id()
        self.distal_residue_fullid = dist_residue.get_full_id()


        # grab the coordinates for the proximal and distal residues as
vectors so we can do math on them later

        # proximal residue
        if self.QUIET:
            warnings.filterwarnings("ignore",
category=DisulfideConstructionWarning)
        try:
            n1 = prox_residue['N'].get_vector()
            ca1 = prox_residue['CA'].get_vector()
            c1 = prox_residue['C'].get_vector()
            o1 = prox_residue['O'].get_vector()
            cb1 = prox_residue['CB'].get_vector()
            sg1 = prox_residue['SG'].get_vector()

        except Exception:
            raise DisulfideConstructionWarning(f"Invalid or missing
coordinates for proximal residue {proximal}") from None

        # distal residue
        try:
            n2 = dist_residue['N'].get_vector()
            ca2 = dist_residue['CA'].get_vector()
            c2 = dist_residue['C'].get_vector()
            o2 = dist_residue['O'].get_vector()
            cb2 = dist_residue['CB'].get_vector()
```



```
            sg2 = dist_residue['SG'].get_vector()

        except Exception:
            raise DisulfideConstructionWarning(f"Invalid or missing
coordinates for proximal residue {distal}") from None

        # update the positions and conformation
        self.set_positions(n1, ca1, c1, o1, cb1, sg1, n2, ca2, c2, o2,
cb2, sg2)

        # calculate and set the disulfide dihedral angles
        self.chi1 = numpy.degrees(calc_dihedral(n1, ca1, cb1, sg1))
        self.chi2 = numpy.degrees(calc_dihedral(ca1, cb1, sg1, sg2))
        self.chi3 = numpy.degrees(calc_dihedral(cb1, sg1, sg2, cb2))
        self.chi4 = numpy.degrees(calc_dihedral(sg1, sg2, cb2, ca2))
        self.chi5 = numpy.degrees(calc_dihedral(sg2, cb2, ca2, n2))

        self.ca_distance = distance3d(self.ca_prox, self.ca_dist)
        self.torsion_array = numpy.array((self.chi1, self.chi2, self.chi3,
self.chi4, self.chi5))

        # calculate and set the SS bond torsional energy
        self.compute_torsional_energy()

        # compute and set the local coordinates
        self.compute_local_coords()

    def set_chain_id(self, chain_id):
        self.chain_id = chain_id

    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):
        '''
        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
```

```
523             cb_distal
524             sg_distal
525         Returns: None
526         '''
527
528         # deep copy
529         self.n_prox = n_prox.copy()
530         self.ca_prox = ca_prox.copy()
531         self.c_prox = c_prox.copy()
532         self.o_prox = o_prox.copy()
533         self.cb_prox = cb_prox.copy()
534         self.sg_prox = sg_prox.copy()
535         self.sg_dist = sg_dist.copy()
536         self.cb_dist = cb_dist.copy()
537         self.ca_dist = ca_dist.copy()
538         self.n_dist = n_dist.copy()
539         self.c_dist = c_dist.copy()
540         self.o_dist = o_dist.copy()
541
542     def set_conformation(self, chi1, chi2, chi3, chi4, chi5):
543         '''
544         Sets the 5 dihedral angles chi1 - chi5 for the Disulfide object
    … and computes the torsional energy.
545
546         Arguments: chi, chi2, chi3, chi4, chi5 - Dihedral angles in
    … degrees (-180 - 180) for the Disulfide conformation.
547         Returns: None
548         '''
549
550         self.chi1 = chi1
551         self.chi2 = chi2
552         self.chi3 = chi3
553         self.chi4 = chi4
554         self.chi5 = chi5
555         self.dihedrals = list([chi1, chi2, chi3, chi4, chi5])
556         self.compute_torsional_energy()
557
558     def set_name(self, namestr="Disulfide"):
559         '''
560         Sets the Disulfide's name
561         Arguments: (str)namestr
562         Returns: none
563         '''
564
565         self.name = namestr
566
567     def set_resnum(self, proximal, distal):
568         '''
```



```
569         Sets the Proximal and Distal Residue numbers for the Disulfide
570         Arguments:
571             Proximal: Proximal residue number
572             Distal: Distal residue number
573         Returns: None
574         '''
575
576         self.proximal = proximal
577         self.distal = distal
578
579     def TorsionDistance(p1: Vector, p2: Vector):
580         '''
581         Calculate the 5D Euclidean distance for 2 Disulfide torsion_vector
    … objects. This is used
582         to compare Disulfide Bond torsion angles to determine their
    … torsional
583         'distance'.
584
585         Arguments: p1, p2 Vector objects of dimensionality 5 (5D)
586         Returns: Distance
587         '''
588         _p1 = p1.get_array()
589         _p2 = p2.get_array()
590         if (len(_p1) != 5 or len(_p2) != 5):
591             raise ProteusPyWarning("--> distance5d() requires vectors of
    … length 5!")
592         d = math.dist(_p1, _p2)
593         return d
594
595     def compute_torsional_energy(self):
596         '''
597         Compute the approximate torsional energy for the Disulfide's
    … conformation.
598         Arguments: chi1, chi2, chi3, chi4, chi5 - the dihedral angles for
    … the Disulfide
599         Returns: Energy (kcal/mol)
600         '''
601         # @TODO find citation for the ss bond energy calculation
602         chi1 = self.chi1
603         chi2 = self.chi2
604         chi3 = self.chi3
605         chi4 = self.chi4
606         chi5 = self.chi5
607
608         energy = 2.0 * (cos(torad(3.0 * chi1)) + cos(torad(3.0 * chi5)))
609         energy += cos(torad(3.0 * chi2)) + cos(torad(3.0 * chi4))
610         energy += 3.5 * cos(torad(2.0 * chi3)) + 0.6 * cos(torad(3.0 *
    … chi3)) + 10.1
```

```
611
612         self.energy = energy
613
614     def compute_local_coords(self):
615         """
616         Compute the internal coordinates for a properly initialized
    … Disulfide Object.
617         Arguments: SS initialized Disulfide object
618         Returns: None, modifies internal state of the input
619         """
620
621         turt = Turtle3D('tmp')
622         # get the coordinates as numpy.array for Turtle3D use.
623         n = self.n_prox.get_array()
624         ca = self.ca_prox.get_array()
625         c = self.c_prox.get_array()
626         cb = self.cb_prox.get_array()
627         o = self.o_prox.get_array()
628         sg = self.sg_prox.get_array()
629
630         sg2 = self.sg_dist.get_array()
631         cb2 = self.cb_dist.get_array()
632         ca2 = self.ca_dist.get_array()
633         c2 = self.c_dist.get_array()
634         n2 = self.n_dist.get_array()
635         o2 = self.o_dist.get_array()
636
637         turt.orient_from_backbone(n, ca, c, cb, ORIENT_SIDECHAIN)
638
639         # internal (local) coordinates, stored as Vector objects
640         # to_local returns numpy.array objects
641
642         self._n_prox = Vector(turt.to_local(n))
643         self._ca_prox = Vector(turt.to_local(ca))
644         self._c_prox = Vector(turt.to_local(c))
645         self._o_prox = Vector(turt.to_local(o))
646         self._cb_prox = Vector(turt.to_local(cb))
647         self._sg_prox = Vector(turt.to_local(sg))
648
649         self._n_dist = Vector(turt.to_local(n2))
650         self._ca_dist = Vector(turt.to_local(ca2))
651         self._c_dist = Vector(turt.to_local(c2))
652         self._o_dist = Vector(turt.to_local(o2))
653         self._cb_dist = Vector(turt.to_local(cb2))
654         self._sg_dist = Vector(turt.to_local(sg2))
655
656     def build_model(self, turtle: Turtle3D):
657         """
```



```
658         Build a model Disulfide based on the internal dihedral angles.
659         Routine assumes turtle is in orientation #1 (at Ca, headed toward
660         Cb, with N on left), builds disulfide, and updates the object's
    … internal
661         coordinate state. It also adds the distal protein backbone,
662         and computes the disulfide conformational energy.
663
664         Arguments: turtle: Turtle3D object properly oriented for the
    … build.
665         Returns: None. The Disulfide object's internal state is updated.
666         """
667
668         tmp = Turtle3D('tmp')
669         tmp.copy_coords(turtle)
670
671         n = Vector(0, 0, 0)
672         ca = Vector(0, 0, 0)
673         cb = Vector(0, 0, 0)
674         c = Vector(0, 0, 0)
675
676         self.ca_prox = tmp._position
677         tmp.schain_to_bbone()
678         n, ca, cb, c = build_residue(tmp)
679
680         self.n_prox = n
681         self.ca_prox = ca
682         self.c_prox = c
683
684         tmp.bbone_to_schain()
685         tmp.move(1.53)
686         tmp.roll(self.chi1)
687         tmp.yaw(112.8)
688         self.cb_prox = tmp._position
689
690         tmp.move(1.86)
691         tmp.roll(self.chi2)
692         tmp.yaw(103.8)
693         self.sg_prox = tmp._position
694
695         tmp.move(2.044)
696         tmp.roll(self.chi3)
697         tmp.yaw(103.8)
698         self.sg_dist = tmp._position
699
700         tmp.move(1.86)
701         tmp.roll(self.chi4)
702         tmp.yaw(112.8)
703         self.cb_dist = tmp._position
```

```
704
705         tmp.move(1.53)
706         tmp.roll(self.chi5)
707         tmp.pitch(180.0)
708         tmp.schain_to_bbone()
709         n, ca, cb, c = build_residue(tmp)
710
711         self.n_dist = n
712         self.ca_dist = ca
713         self.c_dist = c
714
715         self.compute_torsional_energy()
716
717 # Class defination ends
718
719 class DisulfideLoader():
720     '''
721     This class loads .pkl files created from the ExtractDisulfides()
    routine
722     and initializes itself with their contents. The Disulfide objects are
    contained
723     in a DisulfideList object and Dict. This makes it possible to access
    the disulfides by
724     array index or PDB structure ID.\n
725
726     Example:
727         from proteusPy.disulfide import DisulfideList, Disulfide,
    DisulfideLoader
728
729         SS1 = DisulfideList([],'tmp1')
730         SS2 = DisulfideList([],'tmp2')
731
732         PDB_SS = DisulfideLoader()
733         SS1 = PDB_SS[0]         <-- returns a Disulfide object at index 0
734         SS2 = PDB_SS['4yys']    <-- returns a DisulfideList containing all
    disulfides for 4yys
735         SS3 = PDB_SS[:10]       <-- returns a DisulfideList containing the
    slice
736     '''
737
738     def __init__(self, verbose=True, modeldir=MODEL_DIR,
    picklefile=SS_PICKLE_FILE,
739                  pickle_dict_file=SS_DICT_PICKLE_FILE,
740                  torsion_file=SS_TORSIONS_FILE):
741         self.ModelDir = modeldir
742         self.PickleFile = f'{modeldir}{picklefile}'
743         self.PickleDictFile = f'{modeldir}{pickle_dict_file}'
744         self.TorsionFile = f'{modeldir}{torsion_file}'
```



```
745         self.SSList = DisulfideList([], 'ALL_PDB_SS')
746         self.SSDict = {}
747         self.TorsionDF = pd.DataFrame()
748         self.TotalDisulfides = 0
749         self.IDList = []
750
751         # create a dataframe with the following columns for the disulfide
    conformations extracted from the structure
752         df_cols = ['source', 'ss_id', 'proximal', 'distal', 'chi1',
    'chi2', 'chi3', 'chi4', 'chi5', 'energy']
753         SS_df = pd.DataFrame(columns=df_cols, index=['source'])
754         _SSList = DisulfideList([], 'ALL_PDB_SS')
755
756         idlist = []
757         if verbose:
758             print(f'Reading disulfides from: {self.PickleFile}')
759         with open(self.PickleFile, 'rb') as f:
760             self.SSList = pickle.load(f)
761
762         self.TotalDisulfides = len(self.SSList)
763
764         if verbose:
765             print(f'Disulfides Read: {self.TotalDisulfides}')
766             print(f'Reading disulfide dict from: {self.PickleDictFile}')
767
768         with open(self.PickleDictFile, 'rb') as f:
769             self.SSDict = pickle.load(f)
770             for key in self.SSDict:
771                 idlist.append(key)
772             self.IDList = idlist.copy()
773             totalSS_dict = len(self.IDList)
774
775         if verbose:
776             print(f'Reading Torsion DF {self.TorsionFile}.')
777
778         self.TorsionDF = pd.read_csv(self.TorsionFile)
779
780         if verbose:
781             print(f'Read torsions DF.')
782             print(f'PDB IDs parsed: {totalSS_dict}')
783             print(f'Total Space Used: {sys.getsizeof(self.SSList) +
    sys.getsizeof(self.SSDict) + sys.getsizeof(self.TorsionDF)} bytes.')
784         return
785
786     # overload __getitem__ to handle slicing and indexing
787     def __getitem__(self, item):
788         if isinstance(item, slice):
789             indices = range(*item.indices(len(self.SSList)))
```

```
            # return [self.SSList[i] for i in indices]
            name = self.SSList[0].pdb_id
            sublist = [self.SSList[i] for i in indices]
            return DisulfideList(sublist, name)

        if isinstance(item, int):
            if (item < 0 or item >= self.TotalDisulfides):
                mess = f'DisulfideDataLoader error. Index {item} out of
range 0-{self.TotalDisulfides - 1}'
                raise DisulfideException(mess)
            else:
                return self.SSList[item]

        try:
            res = self.SSDict[item]
        except KeyError:
            mess = f'! Cannot find key {item} in SSBond dict!'
            raise DisulfideException(mess)
        return res

    def __setitem__(self, index, item):
        self.SSList[index] = self.validate_ss(item)

    def getlist(self):
        return self.SSList.copy()

    def getdict(self) -> dict:
        return copy.deepcopy(self.SSDict)

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

    def validate_ss(self, value):
        if isinstance(value, (Disulfide)):
            return value
        raise TypeError(
            f"Disulfide object expected, got {type(value).__name__}"
        )
# class ends

class CysSelect(Select):
    def accept_residue(self, residue):
        if residue.get_name() == 'CYS':
            return True
        else:
            return False

def distance3d(p1: Vector, p2: Vector):
```



```
    '''
    Calculate the 3D Euclidean distance for 2 Vector objects

    Arguments: p1, p2 Vector objects of dimensionality 3 (3D)
    Returns: Distance
    '''
    _p1 = p1.get_array()
    _p2 = p2.get_array()
    if (len(_p1) != 3 or len(_p2) != 3):
        raise ProteusPyWarning("--> distance3d() requires vectors of
length 3!")
    d = math.dist(_p1, _p2)
    return d


def name_to_id(fname: str):
    '''return an entry id for filename pdb1crn.ent -> 1crn'''
    ent = fname[3:-4]
    return ent

def torad(deg):
    return(numpy.radians(deg))

def todeg(rad):
    return(numpy.degrees(rad))

def parse_ssbond_header_rec(ssbond_dict: dict) -> list:
    '''
    Parse the SSBOND dict returned by parse_pdb_header.
    NB: Requires EGS-Modified BIO.parse_pdb_header.py

    Arguments:
        ssbond_dict: the input SSBOND dict
    Returns: a list of tuples representing the proximal, distal residue
             ids for the disulfide.

    '''
    disulfide_list = []
    for ssb in ssbond_dict.items():
        disulfide_list.append(ssb[1])

    return disulfide_list

#
# function reads a comma separated list of PDB IDs and download the
corresponding
# .ent files to the PDB_DIR global.
# Used to download the list of proteins containing at least one SS bond
```

```
# with the ID list generated from: http://www.rcsb.org/
#

def DownloadDisulfides(pdb_home=PDB_DIR, model_home=MODEL_DIR,
                       verbose=False, reset=False) -> None:
    '''
    Function reads a comma separated list of PDB IDs and downloads them
    to the pdb_home path.

    Used to download the list of proteins containing at least one SS bond
    with the ID list generated from: http://www.rcsb.org/
    '''

    start = time.time()
    donelines = []
    SS_done = []
    ssfile = None

    cwd = os.getcwd()
    os.chdir(pdb_home)

    pdblist = PDBList(pdb=pdb_home, verbose=verbose)
    ssfilename = f'{model_home}{SS_ID_FILE}'
    print(ssfilename)

    # list of IDs containing >1 SSBond record
    try:
        ssfile = open(ssfilename)
        Line = ssfile.readlines()
    except Exception:
        raise DisulfideIOException(f'Cannot open file: {ssfile}')

    for line in Line:
        entries = line.split(',')

    print(f'Found: {len(entries)} entries')
    completed = {'xxx'} # set to keep track of downloaded

    # file to track already downloaded entries.
    if reset==True:
        completed_file = open(f'{model_home}ss_completed.txt', 'w')
        donelines = []
        SS_DONE = []
    else:
        completed_file = open(f'{model_home}ss_completed.txt', 'w+')
        donelines = completed_file.readlines()

    if len(donelines) > 0:
```



```
        for dl in donelines[0]:
            # create a list of pdb id already downloaded
            SS_done = dl.split(',')

    count = len(SS_done) - 1
    completed.update(SS_done) # update the completed set with what's
downloaded

    # Loop over all entries,
    pbar = tqdm(entries, ncols=_PBAR_COLS)
    for entry in pbar:
        pbar.set_postfix({'Entry': entry})
        if entry not in completed:
            if pdblist.retrieve_pdb_file(entry, file_format='pdb',
pdir=pdb_home):
                completed.update(entry)
                completed_file.write(f'{entry},')
                count += 1

    completed_file.close()

    end = time.time()
    elapsed = end - start

    print(f'Overall files processed: {count}')
    print(f'Complete. Elapsed time: {datetime.timedelta(seconds=elapsed)}
(h:m:s)')
    os.chdir(cwd)
    return

def build_torsion_df(SSList: DisulfideList) -> pd.DataFrame:
    # create a dataframe with the following columns for the disulfide
conformations extracted from the structure
    df_cols = ['source', 'ss_id', 'proximal', 'distal', 'chi1', 'chi2',
'chi3', 'chi4', 'chi5', 'energy', 'ca_distance']
    SS_df = pd.DataFrame(columns=df_cols)

    pbar = tqdm(SSList, ncols=_PBAR_COLS, miniters=400000)
    for ss in pbar:
        #pbar.set_postfix({'ID': ss.name}) # update the progress bar

        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]
        # add the row to the end of the dataframe
        SS_df.loc[len(SS_df.index)] = new_row.copy() # deep copy

    return SS_df.copy()
```

```
973
974 def ExtractDisulfides(numb=-1, verbose=False, quiet=False, pdbdir=PDB_DIR,
975                         modeldir=MODEL_DIR, picklefile=SS_PICKLE_FILE,
976                         torsionfile=SS_TORSIONS_FILE,
… problemfile=PROBLEM_ID_FILE,
977                         dictfile=SS_DICT_PICKLE_FILE) -> None:
978     '''
979     This function creates .pkl files needed for the DisulfideLoader class.
… The Disulfide
980     objects are contained in a DisulfideList object and Dict within these
… files.
981     In addition, .csv files containing all of the torsions for the
… disulfides and
982     problem IDs are written.
983
984     Arguments:
985         numb:           number of entries to process, defaults to all
986         verbose:        more messages
987         quiet:          turns of DisulfideConstruction warnings
988         pdbdir:         path to PDB files
989         modeldir:       path to resulting .pkl files
990         picklefile:     name of the disulfide .pkl file
991         torsionfile:    name of the disulfide torsion file .csv created
992         problemfile:    name of the .csv file containing problem ids
993         dictfile:       name of the .pkl file
994
995     Example:
996         from proteusPy.Disulfide import ExtractDisulfides,
… DisulfideLoader, DisulfideList
997
998         ExtractDisulfides(numb=500, pdbdir=PDB_DIR, verbose=False,
… quiet=True)
999
1000        SS1 = DisulfideList([],'All_SS')
1001        SS2 = DisulfideList([], '4yys')
1002
1003        PDB_SS = DisulfideLoader()
1004        SS1 = PDB_SS[0]         <-- returns a Disulfide object at index 0
1005        SS2 = PDB_SS['4yys']    <-- returns a DisulfideList containing all
… disulfides for 4yys
1006        SS3 = PDB_SS[:10]       <-- returns a DisulfideList containing the
… slice
1007    '''
1008
1009    entrylist = []
1010    problem_ids = []
1011    bad = 0
1012
```



```
1013    # we use the specialized list class DisulfideList to contain our
… disulfides
1014    # we'll use a dict to store DisulfideList objects, indexed by the
… structure ID
1015    All_ss_dict = {}
1016    All_ss_list = []
1017
1018    start = time.time()
1019    cwd = os.getcwd()
1020
1021    # Build a list of PDB files in PDB_DIR that are readable. These files
… were downloaded
1022    # via the RCSB web query interface for structures containing >= 1 SS
… Bond.
1023
1024    os.chdir(pdbdir)
1025
1026    ss_filelist = glob.glob(f'*.ent')
1027    tot = len(ss_filelist)
1028
1029    if verbose:
1030        print(f'PDB Directory {pdbdir} contains: {tot} files')
1031
1032    # the filenames are in the form pdb{entry}.ent, I loop through them
… and extract
1033    # the PDB ID, with Disulfide.name_to_id(), then add to entrylist.
1034
1035    for entry in ss_filelist:
1036        entrylist.append(name_to_id(entry))
1037
1038    # create a dataframe with the following columns for the disulfide
… conformations extracted from the structure
1039
1040    df_cols = ['source', 'ss_id', 'proximal', 'distal', 'chi1', 'chi2',
… 'chi3', 'chi4', 'chi5', 'energy', 'ca_distance']
1041    SS_df = pd.DataFrame(columns=df_cols)
1042
1043    # define a tqdm progressbar using the fully loaded entrylist list. If
… numb is passed then
1044    # only do the last numb entries.
1045    if numb > 0:
1046        pbar = tqdm(entrylist[:numb], ncols=_PBAR_COLS)
1047    else:
1048        pbar = tqdm(entrylist, ncols=_PBAR_COLS)
1049
1050    # loop over ss_filelist, create disulfides and initialize them
1051    for entry in pbar:
1052        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]
                # 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:
        if verbose:
            print(f'Found and removed: {len(problem_ids)} problem
structures.')
        prob_cols = ['id']
        problem_df = pd.DataFrame(columns=prob_cols)
        problem_df['id'] = problem_ids

        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)

    fname = f'{modeldir}{torsionfile}'
    if True:
        print(f'Saving torsions to file: {fname}')

    SS_df.to_csv(fname)

    end = time.time()
    elapsed = end - start

    print(f'Disulfide Extraction complete! Elapsed time:
{datetime.timedelta(seconds=elapsed)} (h:m:s)')

    # return to original directory
    os.chdir(cwd)
    return

def check_chains(pdbid, pdbdir, verbose=True):
    '''Returns True if structure has multiple chains of identical length,
False otherwise'''

    parser = PDBParser(PERMISSIVE=True)
    structure = parser.get_structure(pdbid,
file=f'{pdbdir}pdb{pdbid}.ent')
    ssbond_dict = structure.header['ssbond'] # dictionary of tuples with
SSBond prox and distal

    if verbose:
        print(f'ssbond dict: {ssbond_dict}')

    same = False
    model = structure[0]
    chainlist = model.get_list()

    if len(chainlist) > 1:
        chain_lens = []
        if verbose:
            print(f'multiple chains. {chainlist}')
        for chain in chainlist:
            chain_length = len(chain.get_list())
            chain_id = chain.get_id()
            if verbose:
                print(f'Chain: {chain_id}, length: {chain_length}')
            chain_lens.append(chain_length)
```

```
1139
1140        if numpy.min(chain_lens) != numpy.max(chain_lens):
1141            same = False
1142            if verbose:
1143                print(f'chain lengths are unequal: {chain_lens}')
1144        else:
1145            same = True
1146            if verbose:
1147                print(f'Chains are equal length, assuming the same.
{chain_lens}')
1148    return(same)
1149
1150# NB - this only works with the EGS modified version of
BIO.parse_pdb_header.py
1151def load_disulfides_from_id(struct_name: str,
1152                            pdb_dir = '.',
1153                            model_numb = 0,
1154                            verbose = False,
1155                            quiet=False,
1156                            dbg = False) -> list:
1157    '''
1158    Loads all Disulfides by PDB ID and initializes the Disulfide objects.
1159    Assumes the file is downloaded in the pdb_dir path.
1160
1161    NB: Requires EGS-Modified BIO.parse_pdb_header.py
1162
1163    Arguments:
1164        struct_name: the name of the PDB entry.
1165
1166        pdb_dir: path to the PDB files, defaults to PDB_DIR
1167
1168        model_numb: model number to use, defaults to 0 for single
1169        structure files.
1170
1171        verbose: print info while parsing
1172
1173    Returns: a list of Disulfide objects initialized from the file.
1174    Example:
1175      Assuming the PDB_DIR has the pdb5rsa.ent file in place calling:
1176
1177      SS_list = []
1178      SS_list = load_disulfides_from_id('5rsa', verbose=True)
1179
1180      loads the Disulfides from the file and initialize the disulfide
objects, returning
1181      them in the result. '''
1182
1183    i = 1
```



```
1184    proximal = distal = -1
1185    SSList = DisulfideList([], struct_name)
1186    _chaina = None
1187    _chainb = None
1188
1189    parser = PDBParser(PERMISSIVE=True)
1190
1191    # Biopython uses the Structure -> Model -> Chain hierarchy to organize
1192    # structures. All are iterable.
1193
1194    structure = parser.get_structure(struct_name,
file=f'{pdb_dir}pdb{struct_name}.ent')
1195    model = structure[model_numb]
1196
1197    if verbose:
1198        print(f'-> load_disulfide_from_id() - Parsing structure:
{struct_name}:')
1199
1200    ssbond_dict = structure.header['ssbond'] # NB: this requires the
modified code
1201
1202    # list of tuples with (proximal distal chaina chainb)
1203    ssbonds = parse_ssbond_header_rec(ssbond_dict)
1204
1205    with warnings.catch_warnings():
1206        if quiet:
1207            #warnings.filterwarnings("ignore",
category=DisulfideConstructionWarning)
1208            warnings.filterwarnings("ignore")
1209        for pair in ssbonds:
1210            # in the form (proximal, distal, chain)
1211            proximal = pair[0]
1212            distal = pair[1]
1213            chain1_id = pair[2]
1214            chain2_id = pair[3]
1215
1216            if not proximal.isnumeric() or not distal.isnumeric():
1217                mess = f' -> Cannot parse SSBond record (non-numeric IDs):
{struct_name} Prox: {proximal} {chain1_id} Dist: {distal} {chain2_id},
ignoring.'
1218                warnings.warn(mess, DisulfideConstructionWarning)
1219                continue
1220            else:
1221                proximal = int(proximal)
1222                distal = int(distal)
1223
1224            if proximal == distal:
1225                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)
                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
      them in the result. '''

    i = 1
    proximal = distal = -1
    SSList = []
    _chaina = None
    _chainb = None

    parser = PDBParser(PERMISSIVE=True)

    # Biopython uses the Structure -> Model -> Chain hierarchy to organize
```

```
    # structures. All are iterable.

    structure = parser.get_structure('tmp', file=filename)
    struct_name = structure.get_id()

#    structure = parser.get_structure(struct_name,
file=f'{pdb_dir}pdb{struct_name}.ent')
    model = structure[model_numb]

    if verbose:
        print(f'-> check_header_from_file() - Parsing file: {filename}:')

    ssbond_dict = structure.header['ssbond'] # NB: this requires the
modified code

    # list of tuples with (proximal distal chaina chainb)
    ssbonds = parse_ssbond_header_rec(ssbond_dict)

    for pair in ssbonds:
        # in the form (proximal, distal, chain)

        proximal = pair[0]
        distal = pair[1]

        if not proximal.isnumeric() or not distal.isnumeric():
            if verbose:
                print(f' ! Cannot parse SSBond record (non-numeric IDs):
{struct_name} Prox:  {proximal} {chain1_id} Dist: {distal} {chain2_id}')
            continue # was pass
        else:
            proximal = int(proximal)
            distal = int(distal)

        chain1_id = pair[2]
        chain2_id = pair[3]

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

        if (chain1_id != chain2_id):
            if verbose:
                print(f' -> Cross Chain SS for: Prox: {proximal}
{chain1_id} Dist: {distal} {chain2_id}')
                pass # was break

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



```
        except KeyError:
            print(f' ! Cannot parse SSBond record (KeyError):
{struct_name} Prox:  <{proximal}> {chain1_id} Dist: <{distal}>
{chain2_id}')
            continue

        # make a new Disulfide object, name them based on proximal and
distal
        # initialize SS bond from the proximal, distal coordinates
        if (_chaina is not None) and (_chainb is not None):
            if _chaina[proximal].is_disordered() or
_chainb[distal].is_disordered():
                continue
            else:
                if verbose:
                    print(f' -> SSBond: {i}: {struct_name}: {proximal}
{chain1_id} - {distal} {chain2_id}')
        else:
            if dbg:
                print(f' -> NULL chain(s): {struct_name}: {proximal}
{chain1_id} - {distal} {chain2_id}')
        i += 1
    return True

def check_header_from_id(struct_name: str,
                             pdb_dir = '.',
                             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: True if the proximal and distal residues are CYS and there
are no cross-chain SS bonds

```

```
    Example:
      Assuming the PDB_DIR has the pdb5rsa.ent file in place calling:

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

    '''

    parser = PDBParser(PERMISSIVE=True, QUIET=True)
    structure = parser.get_structure(struct_name,
file=f'{pdb_dir}pdb{struct_name}.ent')
    model = structure[0]

    ssbond_dict = structure.header['ssbond'] # NB: this requires the
modified code

    bondlist = []
    i = 0

    # get a list of tuples containing the proximal, distal residue IDs for
all SSBonds in the chain.
    bondlist = parse_ssbond_header_rec(ssbond_dict)

    if len(bondlist) == 0:
        if (verbose):
            print(f'-> check_header_from_id(): no bonds found in
bondlist.')
        return False

    for pair in bondlist:
        # in the form (proximal, distal, chain)
        proximal = pair[0]
        distal = pair[1]
        chain1 = pair[2]
        chain2 = pair[3]

        chaina = model[chain1]
        chainb = model[chain2]

        try:
            prox_residue = chaina[proximal]
            dist_residue = chainb[distal]

            prox_residue.disordered_select("CYS")
            dist_residue.disordered_select("CYS")

            if prox_residue.get_resname() != 'CYS' or
dist_residue.get_resname() != 'CYS':
```



```
                if (verbose):
                    print(f'build_disulfide() requires CYS at both
residues: {prox_residue.get_resname()} {dist_residue.get_resname()}')
                return False
        except KeyError:
            if (dbg):
                print(f'Keyerror: {struct_name}: {proximal} {chain1} -
{distal} {chain2}')
                return False

        if verbose:
            print(f' -> SSBond: {i}: {struct_name}: {proximal} {chain1} -
{distal} {chain2}')

        i += 1
    return True

# Class defination for a structure-based Disulfide Bond.


def render_ss(ss: Disulfide, pvp: pv.Plotter(), cpk=False, bs_scale=.2,
spec=.8, specpow=6):
    ''' Update the passed pyVista plotter() object with the mesh data for
the input Disulfide Bond
        Arguments:
            pvp: pyvista.Plotter() object
            cpk: Whether to render as CPK object or ball-and-stick (bool)
    '''

    cyl_radius = .6 * bs_scale
    coords = ss.internal_coords()
    atoms = ('N', 'C', 'C', 'O', 'C', 'SG', 'N', 'C', 'C', 'O', 'C', 'SG')

    # bond connection table with atoms in the specific order shown above:
    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
        ])
```

```
1480
1481        if cpk:
1482            i = 0
1483            for atom in atoms:
1484                rad = ATOM_RADII_CPK[atom]
1485                pvp.add_mesh(pv.Sphere(center=coords[i], radius=rad),
…   color=ATOM_COLORS[atom], smooth_shading=True, specular=spec,
…   specular_power=specpow)
1486                i += 1
1487        else: # ball and stick
1488            i = 0
1489            for atom in atoms:
1490                rad = ATOM_RADII_CPK[atom] * bs_scale
1491                pvp.add_mesh(pv.Sphere(center=coords[i], radius=rad),
…   color=ATOM_COLORS[atom], smooth_shading=True, specular=spec,
…   specular_power=specpow)
1492                i += 1
1493
1494            for i in range(len(bond_conn)):
1495                bond = bond_conn[i]
1496                orig = bond[0]
1497                dest = bond[1]
1498                prox_pos = coords[orig]
1499                distal_pos = coords[dest]
1500                direction = distal_pos - prox_pos
1501                height = math.dist(prox_pos, distal_pos)
1502                origin = prox_pos + 0.5 * direction # the cylinder origin is
…   actually in the middle so we translate
1503                cyl = pv.Cylinder(origin, direction, radius=cyl_radius,
…   height=height)
1504                pvp.add_mesh(cyl)
1505        return
1506
1507    # End of file
1508
```