```python
import pyrosetta
from pyrosetta import *
from pyrosetta.rosetta.protocols.docking import *
from pyrosetta.rosetta.core.scoring import *
from pyrosetta.rosetta.protocols.moves import *
from pyrosetta.rosetta.core.pack.task import TaskFactory
from pyrosetta.rosetta.core.pack.task.operation import RestrictToRepackingRLT
from pyrosetta.rosetta.protocols.minimization_packing import PackRotamersMover
import numpy as np
from dataclasses import dataclass
from typing import List, Dict, Tuple, Optional
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class JAZProtein:
    """Class to store JAZ protein information"""
    name: str
    sequence: str
    pdb_path: str
    jas_start: int  # Start residue of JAS motif
    jas_end: int    # End residue of JAS motif

class MYC2JAZAnalyzer:
    def __init__(self, myc2_pdb_path: str, reference_complex_path: str):
        """
        Initialize the analyzer with MYC2 structure and reference complex.
        """
        pyrosetta.init(extra_options="-ex1 -ex2 -use_input_sc -docking:dock_pert 3 8")

        self.myc2_original = pose_from_pdb(myc2_pdb_path)
        self.reference_complex = pose_from_pdb(reference_complex_path)

        self.charged_residues = [131, 133, 135, 136, 137, 140, 141]
        self.replacement_aas = ['iALA', 'VAL', 'ILE', 'LEU']

        self.scorefxn = create_score_function('ref2015_docking')
        self.setup_docking_protocol()

        self.reference_geometry = self.extract_interface_geometry(self.reference_complex)

    def setup_docking_protocol(self):
        """Configure high-resolution docking protocol."""
        self.docking_protocol = DockingProtocol()
        self.docking_protocol.set_scorefxn(self.scorefxn)

        tf = TaskFactory()
        tf.push_back(RestrictToRepackingRLT())
        self.docking_protocol.set_task_factory(tf)

    def create_mutation(self, pose: Pose, position: int, new_aa: str) -> Tuple[Pose, bool]:
        """
        Create and validate a single point mutation with improved backbone relaxation.
        """
        mutant = pose.clone()

        # Perform mutation
        mutate = MutateResidue(position, new_aa)
        mutate.apply(mutant)

        # Set up localized relaxation
        movemap = MoveMap()
        movemap.set_bb(False)
        movemap.set_chi(True)
        for i in range(max(1, position - 5), min(mutant.total_residue(), position + 5)):
            movemap.set_bb(i, True)  # Allow backbone movement locally

        minmover = MinMover(movemap, self.scorefxn, 'lbfgs_armijo_nonmonotone', 0.001, True)
        minmover.apply(mutant)

        is_folded = self.validate_fold(mutant, position)
        return mutant, is_folded

    def validate_fold(self, pose: Pose, mutated_position: int) -> bool:
        """
        Validate the fold of a mutated structure.
        """
        score = self.scorefxn(pose)
        if score > 0:
            return False

        # Use DSSP to validate helix integrity
        dssp = pyrosetta.rosetta.core.scoring.dssp.Dssp(pose)
        dssp.apply(pose)
        structure = dssp.get_dssp_secstruct()
        helix_region = structure[128:146]
        if helix_region.count('H') < len(helix_region) * 0.8:
            return False

        # Check local geometry
        mut_xyz = pose.residue(mutated_position).xyz("CA")
        for i in range(1, pose.total_residue() + 1):
            if i != mutated_position and pose.residue(i).xyz("CA").distance(mut_xyz) < 2.0:
                return False
        return True

    def dock_with_jaz(self, myc2_pose: Pose, jaz_pose: Pose, jaz_protein: JAZProtein) -> Optional[Dict]:
        """
        Perform docking and analysis with error handling.
        """
        try:
            docking_pose = Pose()
            docking_pose.append_pose_by_jump(myc2_pose, 1)
            docking_pose.append_pose_by_jump(jaz_pose, 2)

            self.docking_protocol.apply(docking_pose)

            interface_metrics = self.analyze_interface(docking_pose, jaz_protein)
            geometry_rmsd = self.compare_to_reference(self.extract_interface_geometry(docking_pose))

            return {
                'total_score': self.scorefxn(docking_pose),
                'interface_metrics': interface_metrics,
                'geometry_rmsd': geometry_rmsd,
                'pose': docking_pose
            }
        except Exception as e:
            logger.error(f"Docking failed: {e}")
            return None

    def analyze_interface(self, pose: Pose, jaz_protein: JAZProtein) -> Dict:
        """
        Analyze the interface between MYC2 and JAZ.
        """
        hbond_cutoff = 3.5
        salt_bridge_cutoff = 4.0
        hydrophobic_cutoff = 5.0

        interactions = {'hydrogen_bonds': 0, 'salt_bridges': 0, 'hydrophobic_contacts': 0}
        for i in range(jaz_protein.jas_start, jaz_protein.jas_end + 1):
            for j in range(129, 147):
                dist = pose.residue(i).xyz("CA").distance(pose.residue(j).xyz("CA"))
                if dist < hbond_cutoff:
                    interactions['hydrogen_bonds'] += 1
                if dist < salt_bridge_cutoff and pose.residue(i).is_charged() and pose.residue(j).is_charged():
                    interactions['salt_bridges'] += 1
                if dist < hydrophobic_cutoff and pose.residue(i).is_hydrophobic() and pose.residue(j).is_hydrophobic():
                    interactions['hydrophobic_contacts'] += 1

        return {
            'interface_energy': self.scorefxn(pose),
            'interactions': interactions
        }

    def compare_to_reference(self, test_geometry: Dict) -> float:
        """
        Compare test interface geometry to reference geometry.
        """
        rmsd = 0.0
        for metric in self.reference_geometry:
            if metric in test_geometry:
                rmsd += (self.reference_geometry[metric] - test_geometry[metric]) ** 2
        return np.sqrt(rmsd / len(self.reference_geometry))

# Example usage
if __name__ == "__main__":
    jaz_proteins = [
        JAZProtein(
            name="JAZ1",
            sequence="EXAMPLE_SEQUENCE",
            pdb_path="jaz1.pdb",
            jas_start=218,
            jas_end=239
        ),
    ]

    analyzer = MYC2JAZAnalyzer("myc2.pdb", "reference_complex.pdb")
    for jaz in jaz_proteins:
        results = analyzer.analyze_mutation_set(jaz)
        for r in results[:5]:
            print(r)



    def calculate_local_strain(self, pose: Pose, mutation_pos: int, radius: float = 5.0) -> float:
        local_energy = 0.0
        mut_xyz = pose.residue(mutation_pos).xyz("CA")
        for i in range(1, pose.total_residue() + 1):
            if i != mutation_pos and pose.residue(i).xyz("CA").distance(mut_xyz) <= radius:
                local_energy += pose.energies().residue_total_energy(i)
        return local_energy

    def calculate_helix_integrity(self, pose: Pose, start: int, end: int) -> float:
        from pyrosetta.rosetta.core.scoring.dssp import Dssp
        dssp = Dssp(pose)
        structure = dssp.get_dssp_secstruct()
        helix_residues = sum(1 for i in range(start, end + 1) if structure[i - 1] == 'H')
        return (helix_residues / (end - start + 1)) * 100

    def validate_in_vivo_viability(self, pose: Pose) -> Dict[str, bool]:
        folding_energy = self.scorefxn(pose)
        wildtype_score = self.scorefxn(self.myc2_original)
        minimal_deviation = abs(folding_energy - wildtype_score) < 5.0
        return {
            "similarity_to_wild_type": minimal_deviation,
            "retention_of_function": True,  # Placeholder: validate via interaction predictions
            "feasibility_for_dcas13": True  # Placeholder: ensure mutations align with CRISPR constraints
        }

    def analyze_mutation(self, res_pos: int, new_aa: str, jaz_pose: Pose, jaz_protein: JAZProtein):
        mutant_pose, is_folded = self.create_mutation(self.myc2_original, res_pos, new_aa)
        if is_folded:
            folding_energy = self.scorefxn(mutant_pose)
            local_strain = self.calculate_local_strain(mutant_pose, res_pos)
            helix_integrity = self.calculate_helix_integrity(mutant_pose, 129, 147)
            viability_metrics = self.validate_in_vivo_viability(mutant_pose)
            docking_results = self.dock_with_jaz(mutant_pose, jaz_pose, jaz_protein)

            return {
                'mutation': f"{self.myc2_original.residue(res_pos).name3()} → {new_aa}",
                'position': res_pos,
                'folding_energy': folding_energy,
                'local_strain': local_strain,
                'helix_integrity': helix_integrity,
                'viability_metrics': viability_metrics,
                'docking_score': docking_results.get('total_score', float('inf')),
                'interface_metrics': docking_results.get('interface_metrics', {}),
                'geometry_rmsd': docking_results.get('geometry_rmsd', float('inf'))
            }
        return None




def generate_ramachandran_plot(pose: Pose, region: Tuple[int, int], output_file: str):
    """
    Generate a Ramachandran plot for a specific region of the protein.
    Args:
        pose: PyRosetta Pose object
        region: Tuple indicating start and end residues (inclusive)
        output_file: Path to save the plot
    """
    import matplotlib.pyplot as plt

    phi_angles = []
    psi_angles = []

    for res in range(region[0], region[1] + 1):
        phi = pose.phi(res)
        psi = pose.psi(res)
        phi_angles.append(phi)
        psi_angles.append(psi)

    # Plot the Ramachandran plot
    plt.figure(figsize=(8, 6))
    plt.scatter(phi_angles, psi_angles, c='blue', alpha=0.6, label="Residues")
    plt.axhline(0, color='black', linewidth=0.5)
    plt.axvline(0, color='black', linewidth=0.5)
    plt.xlabel("Phi (φ) Angle")
    plt.ylabel("Psi (ψ) Angle")
    plt.title("Ramachandran Plot")
    plt.legend()
    plt.grid(alpha=0.3)
    plt.savefig(output_file)
    plt.show()



if __name__ == "__main__":
    jaz_proteins = [
        JAZProtein(name="JAZ1", sequence="...", pdb_path="jaz1.pdb", jas_start=218, jas_end=239),
        JAZProtein(name="JAZ2", sequence="...", pdb_path="jaz2.pdb", jas_start=215, jas_end=236),
        # Add additional JAZ proteins (JAZ3, JAZ9, JAZ10, JAZ12)
    ]
    analyzer = MYC2JAZAnalyzer("myc2.pdb", "reference_complex.pdb")

    for jaz in jaz_proteins:
        for res_pos in analyzer.charged_residues:
            for new_aa in analyzer.replacement_aas:
                result = analyzer.analyze_mutation(res_pos, new_aa, pose_from_pdb(jaz.pdb_path), jaz)

                if result:
                    logger.info(f"Mutation: {result['mutation']}")
                    logger.info(f"Folding Energy: {result['folding_energy']:.2f}")
                    logger.info(f"Local Strain: {result['local_strain']:.2f}")
                    logger.info(f"Helix Integrity: {result['helix_integrity']:.2f}%")
                    logger.info(f"Viability: {result['viability_metrics']}")
                    logger.info(f"Docking Score: {result['docking_score']:.2f}")
                    logger.info(f"Interface Metrics: {result['interface_metrics']}")
                    logger.info(f"Geometry RMSD: {result['geometry_rmsd']:.2f}")
                    logger.info("-" * 40)
```