Protein Folding

Chains

```
main_chain = "APRLRFY"
side_chains = [""] * 7
```

The main_chain variable stores the sequence of amino acids in the main chain of a protein. In this example, the main chain sequence is "APRLRFY".

The side_chains variable is a list that represents the side chains attached to each residue of the main chain. Side chains of maximum length 1 are only supported for residues within the main chain, excluding the first and last residues. In this example we do not consider any side chains to keep the real structure of the neuropeptide.

Interaction between Aminoacids

```
random_interaction = RandomInteraction()
mj_interaction = MiyazawaJerniganInteraction()
```

RandomInteraction Class

The "RandomInteraction" class provides a random energy matrix for amino acid pairs, which does not follow any specific pattern or statistical potential. This class has **calculate_energy_matrix** method, which takes a residue_sequence parameter, which is a dummy residue sequence used to infer the length of the peptide chain. It generates a random energy matrix. The method returns a numpy array of shape (chain_len + 1, 2, chain_len + 1, 2) and assigns random values between -1 and -5 to represent the pair energies between amino acids. Each dimension of the array corresponds to a specific aspect of the interaction: the index of the first residue (main chain), the state of the first residue (side chain or not), the index of the second residue (main chain), and the state of the second residue (side chain or not).

MiyazawaJerniganInteraction Class

The "MiyazawaJerniganInteraction" class represents the Miyazawa-Jernigan interaction between beads of a peptide in the protein folding model. It provides a method called **calculate_energy_matrix** that calculates the energy matrix for the Miyazawa-Jernigan interaction based on the Miyazawa-Jernigan potential file. It uses statistical potentials derived using the quasi-chemical approximation, as introduced by Miyazawa and Jernigan. These potentials capture the inter-residue contacts observed in proteins.

Physical Constraints

```
penalty_back = 10
penalty_chiral = 10
penalty_1 = 10

penalty_terms = PenaltyParameters(penalty_chiral, penalty_back, penalty_1)
```

The physical constraints in the protein folding model are enforced through penalty functions. These penalty functions help ensure that the folding process respects certain physical properties. The model uses several penalty terms to address different constraints:

- **penalty_chiral**: This penalty parameter is used to impose the correct chirality in the protein structure. This penalty term helps maintain the correct orientation of amino acids in the folded protein.
- **penalty_back**: This penalty parameter penalizes turns along the same axis. It prevents the chain from folding back into itself by disallowing the selection of the same axis twice in a row. This constraint ensures that the protein maintains a relatively linear structure without self-intersections.
- **penalty_1**: This penalty parameter penalizes local overlap between beads within a nearest neighbor contact. It aims to avoid steric clashes or unfavorable interactions that may occur when two amino acids are positioned too closely to each other in the folded protein structure.

Peptide Definition

```
peptide = Peptide(main_chain, side_chains)
side_chains = peptide.get_side_chains()
side_chain_hot_vector = peptide.get_side_chain_hot_vector()
main_chain = peptide.get_main_chain
print(side_chain_hot_vector)
```

The "Peptide' class represents a protein and contains information about its beads (amino acids) and chains. It has methods to retrieve the side chains (get_side_chains()) and their presence as a one-hot encoding (get_side_chain_hot_vector()). It also provides access to the main chain of the peptide (get_main_chain()).

Protein Folding Problem

```
protein_folding_problem = ProteinFoldingProblem(peptide, mj_interaction,
    penalty_terms)
qubit_op = protein_folding_problem.qubit_op()
print(qubit_op)
```

The `ProteinFoldingProblem` class defines a protein folding problem that can be passed to algorithms. It handles the construction of the qubit operator for the problem Hamiltonian and provides methods for interpretation of the algorithm results.

- `qubit_op()`: This method builds a qubit operator for the Hamiltonian encoding the protein folding problem. It uses the `QubitOpBuilder` class to construct the qubit operators for the Hamiltonian terms. The method first calls the `_qubit_op_full()` method to build the full qubit operator without optimization. It then applies qubit number reduction using the `qubit_number_reducer` module to remove unused qubits and optimize the number of qubits needed for the problem. The optimized qubit operator is returned.

  Within the `_qubit_op_full()` method, the `_create_h_back()`, `_create_h_chiral()`, `_create_h_bbbb()`, `_create_h_bbsc_and_h_scbb()`, and `_create_h_short()` functions from the `QubitOpBuilder` class are used to build different components of the qubit operator for the protein folding problem's Hamiltonian. These functions create specific Hamiltonian terms that capture penalties and interactions relevant to the folding problem.

- `interpret(raw_result)`: This method interprets the raw algorithm result in the context of the protein folding problem and returns a `ProteinFoldingResult` object. The `ProteinFoldingResult` object contains the protein folding result, which includes the optimized qubit operator, unused qubits, peptide information, and the best turn sequence obtained from the algorithm result. This method allows for further analysis and visualization of the protein folding solution.

- `unused_qubits`: This property returns the list of indices for qubits in the original problem formulation that were removed during the qubit number reduction process in the `qubit_op()` method. These unused qubits are not required for the solution and their removal helps optimize the qubit resources.

- `peptide`: This property returns the `Peptide` object defining the protein subject to the folding problem. The `Peptide` object contains the primary amino acid sequence and side chain information of the protein.

The output of `print(qubit_op)` will be a textual representation of the qubit operator. The qubit operator represents the Hamiltonian of the protein folding problem and describes the energy of the system based on qubit interactions and constraints. The qubit operator is expressed as a linear combination of Pauli strings, where each Pauli string represents a product of Pauli operators (X, Y, Z) acting on specific qubits. Each line in the output corresponds to a term in the qubit operator, consisting of a coefficient and a Pauli string. The coefficient represents the weight of the term in the Hamiltonian, indicating its contribution to the overall energy. The Pauli string describes the qubit interactions or constraints associated with the term. For example, a line like "+ 487.5 * IIIIIIZII" indicates a term with a coefficient of 487.5 and a Pauli string "IIIIIZII". The presence of Pauli operators (X, Y, Z) in the string represents specific qubit interactions or constraints within the protein folding problem.

Using VQE with CVaR expectation value for the solution of the problem

```
from qiskit.circuit.library import RealAmplitudes
from qiskit.algorithms.optimizers import COBYLA
from qiskit.algorithms import NumPyMinimumEigensolver
from qiskit.algorithms.minimum_eigensolvers import SamplingVQE
from qiskit import execute, Aer
```

```python
from qiskit.primitives import Sampler

# set classical optimizer
optimizer = COBYLA(maxiter=50)

# set variational ansatz
ansatz = RealAmplitudes(reps=1)

counts = []
values = []

def store_intermediate_result(eval_count, parameters, mean, std):
    counts.append(eval_count)
    values.append(mean)

# initialize VQE using CVaR with alpha = 0.1
vqe = SamplingVQE(
    Sampler(),
    ansatz=ansatz,
    optimizer=optimizer,
    aggregation=0.1,
    callback=store_intermediate_result,
)

raw_result = vqe.compute_minimum_eigenvalue(qubit_op)
print(raw_result)
```

The “SamplingVQE” class is a variant of the Variational Quantum Eigensolver (VQE) algorithm, optimized for diagonal Hamiltonians. It find the minimum eigenvalue of a given diagonal Hamiltonian operator by using a hybrid quantum-classical approach, where a parameterized quantum circuit (ansatz) is used to prepare a trial state, and a classical optimizer is employed to minimize the objective function based on the chosen aggregation method.

Visualizing the answer

```python
result = protein_folding_problem.interpret(raw_result=raw_result)
print(
    "The bitstring representing the shape of the protein during optimization is: ",
    result.turn_sequence,
)
print("The expanded expression is:", result.get_result_binary_vector())
print(
    f"The folded protein's main sequence of turns is: {result.protein_shape_decoder.
    main_turns}"
)
print(f"and the side turn sequences are: {result.protein_shape_decoder.side_turns}")
print(result.protein_shape_file_gen.get_xyz_data())
fig = result.get_figure(title="Protein Structure", ticks=False, grid=True)
fig.get_axes()[0].view_init(10, 70)
```

The ‘`ProteinFoldingResult`‘ class represents the result of solving the protein folding problem. It interprets the encoded bitstring representing the turns of the protein and provides methods for analysis and visualization of the folded protein structure.

- `protein_shape_decoder`: Interprets the result bitstring and provides the decoded information about the protein's turns.
- `protein_shape_file_gen`: Generates a .xyz file with Cartesian coordinates of the folded protein.
- `turn_sequence`: Returns the best sequence of turns obtained from the solution.
- `get_result_binary_vector()`: Returns the original solution vector, accounting for compressed optimization.
- `save_xyz_file()`: Saves the folded protein structure as a .xyz file.
- `get_figure()`: Generates a 3D plot of the folded protein structure using matplotlib.