**Tab 1**

* import json  
  import numpy as np  
  import math  
  from typing import Dict, List, Tuple, Any  
  from dataclasses import dataclass  
  import logging  
    
  @dataclass  
  class SimulationResults:  
   """Container for simulation results"""  
   entanglement\_values: List[float]  
   correlation\_coefficients: List[float]  
   numerical\_mappings: Dict[str, Any]  
   lambda\_contributions: Dict[str, float]  
   polarization\_data: List[Dict[str, Any]]  
    
  class QuantumEntanglementSDKPSimulator:  
   """  
   Multi-dimensional entanglement simulator using SDKP, SD&N, VEI, and QF methodologies  
   """  
    
   def \_\_init\_\_(self, config: Dict[str, Any]):  
   self.config = config  
   self.setup\_logging()  
   self.validate\_config()  
     
   def setup\_logging(self):  
   """Initialize logging if enabled"""  
   if self.config.get('parameters', {}).get('logging', False):  
   # Set logging level to INFO and include time and level for clarity  
   logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s: %(message)s')  
   self.logger = logging.getLogger(\_\_name\_\_)  
   else:  
   self.logger = None  
     
   def validate\_config(self):  
   """Validate simulation configuration"""  
   required\_keys = ['polarization\_angles\_deg', 'numerical\_signatures', 'lambda\_weights']  
   params = self.config.get('parameters', {})  
     
   for key in required\_keys:  
   if key not in params:  
   raise ValueError(f"Missing required parameter: {key}")  
     
   # Validate lambda weights sum to 1  
   weights = params['lambda\_weights']  
   total\_weight = sum(weights.values())  
   if abs(total\_weight - 1.0) > 1e-6:  
   self.log(f"Warning: Lambda weights sum to {total\_weight:.6f}, normalizing...")  
   for key in weights:  
   weights[key] /= total\_weight  
     
   # Validate required weight keys  
   required\_weight\_keys = ['C\_SDN', 'VEI\_delta', 'QF\_delta']  
   for key in required\_weight\_keys:  
   if key not in weights:  
   raise ValueError(f"Missing lambda weight: {key}. Ensure 'C\_SDN', 'VEI\_delta', 'QF\_delta' are defined.")  
    
   def log(self, message: str):  
   """Log message if logging is enabled"""  
   if self.logger:  
   self.logger.info(message)  
   else:  
   print(message)  
    
   def calculate\_numerical\_signature\_value(self, signature: str) -> float:  
   """  
   Convert numerical signature to quantum-relevant value  
   Uses digit sum and positional weighting.  
   """  
   # Handle non-numeric characters  
   clean\_signature = ''.join(filter(str.isdigit, str(signature)))  
     
   if not clean\_signature:  
   self.log(f"Warning: No digits found in signature '{signature}', using default value 0.5 for calculation.")  
   return 0.5  
     
   digit\_sum = sum(int(d) for d in clean\_signature)  
   positional\_weight = sum(int(d) \* (i + 1) for i, d in enumerate(clean\_signature))  
     
   # Enhanced normalization with signature length consideration  
   # Normalize by typical length (e.g., assuming average signature length is 10 for scaling)  
   length\_factor = len(clean\_signature) / 10.0   
   normalized\_value = (digit\_sum + positional\_weight \* 0.1 + length\_factor) % 1.0  
     
   return normalized\_value  
    
   def calculate\_mass\_basis(self, dim\_index: int, signature\_value: float) -> float:  
   """  
   Calculate mass basis value based on dimensional index and signature.  
   This function is currently not integrated into the main simulation flow  
   but can be used for future extensions related to dimensional modeling.  
   """  
   model = self.config['parameters'].get('mass\_basis\_model', 'dim\_index\_dependent')  
     
   if model == 'dim\_index\_dependent':  
   return (dim\_index + 1) \* signature\_value \* 0.1  
   elif model == 'uniform':  
   return signature\_value  
   else:  
   # Default fallback with enhanced scaling  
   return signature\_value \* (1 + dim\_index \* 0.05)  
    
   def apply\_mapping\_function(self, value: float, mapping\_type: str) -> float:  
   """  
   Apply transformation mapping to values.  
   Ensures value is within [0, 1] for consistent mapping behavior.  
   """  
   # Ensure value is in valid range for mapping functions like sin  
   value = max(0.0, min(1.0, value))  
     
   if mapping\_type == 'linear':  
   return value  
   elif mapping\_type == 'nonlinear':  
   # Example: A sine-squared curve that peaks at 0.5  
   return np.sin(value \* np.pi) \*\* 2  
   elif mapping\_type == 'hybrid':  
   # Example: A blend of linear and non-linear behavior  
   return 0.5 \* value + 0.5 \* np.sin(value \* np.pi) \*\* 2  
   else:  
   self.log(f"Warning: Unknown mapping type '{mapping\_type}', using linear mapping.")  
   return value  
    
   def calculate\_entanglement\_measure(self, angle\_deg: float, signature: str) -> Tuple[float, Dict[str, float]]:  
   """  
   Calculate entanglement measure for given angle and numerical signature.  
   Combines components from SD&N (C\_SDN), VEI (VEI\_delta), and QF (QF\_delta).  
   Includes detailed logging of calculation steps.  
   """  
   self.log(f"\n--- Calculating Entanglement for Angle {angle\_deg}° and Signature '{signature}' ---")  
     
   angle\_rad = math.radians(angle\_deg)  
   self.log(f"Step 1: Convert angle {angle\_deg}° to radians: {angle\_rad:.4f} rad")  
     
   signature\_value = self.calculate\_numerical\_signature\_value(signature)  
   self.log(f"Step 2: Calculated numerical signature value for '{signature}': {signature\_value:.4f}")  
     
   # Calculate individual raw components based on conceptual models  
   # C\_SDN (Symmetry, Duality, Negation): Cosine correlation with signature influence  
   C\_SDN = abs(np.cos(angle\_rad)) \* signature\_value  
   self.log(f"Step 3a: Raw C\_SDN = |cos({angle\_rad:.4f})| \* {signature\_value:.4f} = {abs(np.cos(angle\_rad)):.4f} \* {signature\_value:.4f} = {C\_SDN:.4f}")  
     
   # VEI\_delta (Conceptual placeholder, needs specific definition): Sine-squared dependence  
   VEI\_delta = np.sin(angle\_rad) \*\* 2 \* (1 - signature\_value)  
   self.log(f"Step 3b: Raw VEI\_delta = sin({angle\_rad:.4f})^2 \* (1 - {signature\_value:.4f}) = {np.sin(angle\_rad)\*\*2:.4f} \* {(1-signature\_value):.4f} = {VEI\_delta:.4f}")  
     
   # QF\_delta (Conceptual placeholder, needs specific definition): Exponential decay with sine modulation  
   QF\_delta = np.exp(-signature\_value) \* abs(np.sin(angle\_rad \* 2))  
   self.log(f"Step 3c: Raw QF\_delta = exp(-{signature\_value:.4f}) \* |sin({angle\_rad\*2:.4f})| = {np.exp(-signature\_value):.4f} \* {abs(np.sin(angle\_rad\*2)):.4f} = {QF\_delta:.4f}")  
     
   # Apply mapping transformations to each component  
   mapping\_type = self.config['parameters'].get('mapping\_type', 'linear')  
   self.log(f"Step 4: Applying '{mapping\_type}' mapping function to components.")  
   C\_SDN\_mapped = self.apply\_mapping\_function(C\_SDN, mapping\_type)  
   self.log(f" Mapped C\_SDN: {C\_SDN:.4f} -> {C\_SDN\_mapped:.4f}")  
   VEI\_delta\_mapped = self.apply\_mapping\_function(VEI\_delta, mapping\_type)  
   self.log(f" Mapped VEI\_delta: {VEI\_delta:.4f} -> {VEI\_delta\_mapped:.4f}")  
   QF\_delta\_mapped = self.apply\_mapping\_function(QF\_delta, mapping\_type)  
   self.log(f" Mapped QF\_delta: {QF\_delta:.4f} -> {QF\_delta\_mapped:.4f}")  
     
   # Weighted combination of mapped components  
   weights = self.config['parameters']['lambda\_weights']  
   self.log(f"Step 5: Applying lambda weights: C\_SDN={weights['C\_SDN']:.2f}, VEI\_delta={weights['VEI\_delta']:.2f}, QF\_delta={weights['QF\_delta']:.2f}")  
     
   entanglement\_value = (  
   weights['C\_SDN'] \* C\_SDN\_mapped +   
   weights['VEI\_delta'] \* VEI\_delta\_mapped +   
   weights['QF\_delta'] \* QF\_delta\_mapped  
   )  
   self.log(f" Calculation: ({weights['C\_SDN']:.2f} \* {C\_SDN\_mapped:.4f}) + "  
   f"({weights['VEI\_delta']:.2f} \* {VEI\_delta\_mapped:.4f}) + "  
   f"({weights['QF\_delta']:.2f} \* {QF\_delta\_mapped:.4f})")  
   self.log(f" Final Entanglement Value: {entanglement\_value:.4f}")  
     
   components = {  
   'C\_SDN': C\_SDN\_mapped,  
   'VEI\_delta': VEI\_delta\_mapped,  
   'QF\_delta': QF\_delta\_mapped,  
   'signature\_value': signature\_value,  
   'raw\_C\_SDN': C\_SDN,  
   'raw\_VEI\_delta': VEI\_delta,  
   'raw\_QF\_delta': QF\_delta  
   }  
     
   return entanglement\_value, components  
    
   def calculate\_correlation\_coefficient(self, entanglement\_values: List[float]) -> float:  
   """  
   Calculate auto-correlation coefficient at lag 1 for entanglement values.  
   Handles edge cases for constant or insufficient data.  
   """  
   if len(entanglement\_values) < 2:  
   self.log("Correlation: Not enough data points (<2) for correlation calculation. Returning 0.0.")  
   return 0.0  
     
   # If all values are identical, it's perfect correlation  
   if all(v == entanglement\_values[0] for v in entanglement\_values):  
   self.log("Correlation: All entanglement values are identical. Returning 1.0 (perfect correlation).")  
   return 1.0   
     
   try:  
   # Shifted array for auto-correlation (lag 1, circular)  
   shifted\_values = entanglement\_values[1:] + [entanglement\_values[0]]  
   self.log(f"Correlation: Original values: {[f'{v:.4f}' for v in entanglement\_values]}")  
   self.log(f"Correlation: Shifted values: {[f'{v:.4f}' for v in shifted\_values]}")  
   correlation\_matrix = np.corrcoef(entanglement\_values, shifted\_values)  
     
   # The correlation coefficient is at [0, 1] in a 2x2 matrix  
   correlation = correlation\_matrix[0, 1]  
   self.log(f"Correlation: Calculated numpy correlation matrix: \n{correlation\_matrix.round(4)}")  
   self.log(f"Correlation: Resulting coefficient: {correlation:.4f}")  
   return correlation if not np.isnan(correlation) else 0.0  
   except Exception as e:  
   self.log(f"Warning: Correlation calculation failed: {e}. Returning 0.0.")  
   return 0.0  
    
   def run\_simulation(self) -> SimulationResults:  
   """  
   Run the complete entanglement simulation process.  
   Iterates through defined angles and numerical signatures,  
   calculating entanglement and correlation measures.  
   """  
   self.log("\n========================================================")  
   self.log("=== Starting Quantum Entanglement SDKP Simulation ===")  
   self.log("========================================================")  
     
   angles = self.config['parameters']['polarization\_angles\_deg']  
   signatures = self.config['parameters']['numerical\_signatures']  
   threshold = self.config['parameters'].get('entanglement\_threshold', 0.75)  
     
   self.log(f"Simulation Configuration: Angles={angles}, Signatures={signatures}")  
   self.log(f"Processing {len(angles)} angles with {len(signatures)} numerical signatures.")  
   self.log(f"Total individual measurements planned: {len(angles) \* len(signatures)}")  
   self.log(f"Entanglement threshold for 'high entanglement': {threshold}")  
   self.log(f"Mapping type: '{self.config['parameters'].get('mapping\_type', 'linear')}'")  
   self.log(f"Lambda Weights: {self.config['parameters']['lambda\_weights']}")  
     
   all\_entanglement\_values = []  
   all\_correlation\_coefficients = []  
   numerical\_mappings = {}  
   polarization\_data = []  
     
   for i, angle in enumerate(angles):  
   angle\_entanglements = []  
     
   for j, signature in enumerate(signatures):  
   entanglement\_val, components = self.calculate\_entanglement\_measure(angle, signature)  
   angle\_entanglements.append(entanglement\_val)  
     
   # Store detailed numerical mapping for each specific angle-signature pair  
   key = f"angle\_{angle}\_sig\_{signature}"  
   numerical\_mappings[key] = {  
   'entanglement': float(entanglement\_val),  
   'components': {k: float(v) for k, v in components.items()}, # Ensure float conversion  
   'above\_threshold': bool(entanglement\_val > threshold) # Ensure boolean type  
   }  
     
   # Calculate correlation for this angle across all signatures  
   self.log(f"\n--- Calculating Correlation for Angle {angle}° Entanglement Values ---")  
   correlation = self.calculate\_correlation\_coefficient(angle\_entanglements)  
   all\_correlation\_coefficients.append(correlation)  
   all\_entanglement\_values.extend(angle\_entanglements) # Flatten list of all entanglement values  
     
   # Collect aggregated statistics for the current angle  
   angle\_stats = {  
   'angle\_deg': angle,  
   'entanglement\_values': [float(x) for x in angle\_entanglements], # Ensure float conversion  
   'mean\_entanglement': float(np.mean(angle\_entanglements)),  
   'std\_entanglement': float(np.std(angle\_entanglements)),  
   'min\_entanglement': float(np.min(angle\_entanglements)),  
   'max\_entanglement': float(np.max(angle\_entanglements)),  
   'correlation': float(correlation),  
   'high\_entanglement\_count': sum(1 for e in angle\_entanglements if e > threshold)  
   }  
   polarization\_data.append(angle\_stats)  
     
   self.log(f"\nSummary for Angle {angle:3.0f}°: Mean Entanglement={np.mean(angle\_entanglements):.4f}, "  
   f"Std Dev={np.std(angle\_entanglements):.4f}, Correlation={correlation:.4f}")  
   self.log("--------------------------------------------------------") # Separator for each angle summary  
     
   # Calculate overall lambda contributions based on average component values  
   self.log("\n--- Calculating Overall Lambda Contributions ---")  
   weights = self.config['parameters']['lambda\_weights']  
   lambda\_contributions = {}  
     
   for weight\_key in weights.keys():  
   component\_values = []  
   # Extract relevant component values from all numerical mappings  
   for mapping in numerical\_mappings.values():  
   if weight\_key in mapping['components']:  
   component\_values.append(mapping['components'][weight\_key])  
     
   if component\_values:  
   # Calculate weighted average contribution  
   mean\_component = np.mean(component\_values)  
   lambda\_contributions[weight\_key] = float(weights[weight\_key] \* mean\_component)  
   self.log(f" {weight\_key}: Weight={weights[weight\_key]:.2f} \* Mean\_Component={mean\_component:.4f} = {lambda\_contributions[weight\_key]:.4f}")  
   else:  
   lambda\_contributions[weight\_key] = 0.0 # Default if no data for component  
   self.log(f" {weight\_key}: No component values found. Contribution = 0.0.")  
    
   self.log("--------------------------------------------------------")  
     
   results = SimulationResults(  
   entanglement\_values=all\_entanglement\_values,  
   correlation\_coefficients=all\_correlation\_coefficients,  
   numerical\_mappings=numerical\_mappings,  
   lambda\_contributions=lambda\_contributions,  
   polarization\_data=polarization\_data  
   )  
     
   self.log("\n========================================================")  
   self.log("=== Simulation completed successfully! ===")  
   self.log("========================================================\n")  
   return results  
    
   def export\_results(self, results: SimulationResults, filename: str = None) -> str:  
   """  
   Export simulation results to JSON format.  
   Provides a comprehensive summary, lambda contributions,  
   detailed polarization analysis, and a sample of detailed mappings.  
   """  
   # Calculate comprehensive summary statistics  
   entanglement\_values = results.entanglement\_values  
   correlation\_coefficients = results.correlation\_coefficients  
   threshold = self.config['parameters']['entanglement\_threshold']  
     
   # Avoid division by zero if no measurements  
   total\_measurements = len(entanglement\_values)  
   high\_entanglement\_fraction = (sum(1 for e in entanglement\_values if e > threshold) / total\_measurements) \  
   if total\_measurements > 0 else 0.0  
    
   export\_data = {  
   'simulation\_name': self.config.get('simulation\_name', 'Unnamed Simulation'),  
   'description': self.config.get('description', 'No description provided.'),  
   'simulation\_config': self.config['parameters'], # Export only parameters for conciseness  
   'results': {  
   'summary\_statistics': {  
   'total\_individual\_measurements': total\_measurements,  
   'mean\_entanglement\_across\_all': float(np.mean(entanglement\_values)) if entanglement\_values else 0.0,  
   'std\_entanglement\_across\_all': float(np.std(entanglement\_values)) if entanglement\_values else 0.0,  
   'min\_entanglement\_across\_all': float(np.min(entanglement\_values)) if entanglement\_values else 0.0,  
   'max\_entanglement\_across\_all': float(np.max(entanglement\_values)) if entanglement\_values else 0.0,  
   'mean\_correlation\_across\_angles': float(np.mean(correlation\_coefficients)) if correlation\_coefficients else 0.0,  
   'std\_correlation\_across\_angles': float(np.std(correlation\_coefficients)) if correlation\_coefficients else 0.0,  
   'high\_entanglement\_threshold': threshold,  
   'fraction\_above\_threshold': high\_entanglement\_fraction  
   },  
   'lambda\_contributions': results.lambda\_contributions, # Already float-converted  
   'polarization\_analysis\_per\_angle': results.polarization\_data, # Already float-converted  
   'detailed\_mappings\_sample': {k: v for k, v in list(results.numerical\_mappings.items())[:20]} # Limit to first 20 for readability in export  
   }  
   }  
     
   if filename:  
   try:  
   with open(filename, 'w') as f:  
   json.dump(export\_data, f, indent=4)  
   self.log(f"Results successfully exported to '{filename}'")  
   return f"Results exported to {filename}"  
   except IOError as e:  
   self.log(f"Error: Could not write to file {filename}: {e}")  
   return json.dumps({"error": f"Could not write to file: {e}", "data": export\_data}, indent=2)  
   else:  
   return json.dumps(export\_data, indent=2)  
    
  # Example usage and configuration  
    
  def create\_default\_config():  
   """Create the default simulation configuration"""  
   return {  
   "simulation\_name": "Quantum\_Entanglement\_SDKP\_Study",  
   "description": "Configuration for multi-dimensional entanglement simulations using SDKP, SD&N, VEI, and QF.",  
   "parameters": {  
   "polarization\_angles\_deg": [0, 30, 45, 60, 90, 120, 135, 150, 180],  
   "numerical\_signatures": ["7146", "1467", "4671", "6714", "999988889999", "6", "7", "ABCD123", "0"],  
   "lambda\_weights": {  
   "C\_SDN": 0.4,  
   "VEI\_delta": 0.3,  
   "QF\_delta": 0.3  
   },  
   "mapping\_type": "hybrid",  
   "entanglement\_threshold": 0.75,  
   "mass\_basis\_model": "dim\_index\_dependent", # This parameter is currently not used in the main simulation flow  
   "output\_format": "json",  
   "logging": True,  
   "max\_simulations": 100 # This parameter is currently not used in the simulation flow  
   }  
   }  
    
  # Example simulation run  
    
  if \_\_name\_\_ == "\_\_main\_\_":  
   # Create configuration  
   config = create\_default\_config()  
    
   # Initialize simulator  
   simulator = QuantumEntanglementSDKPSimulator(config)  
    
   # Run simulation  
   results = simulator.run\_simulation()  
    
   # Export results  
   json\_output = simulator.export\_results(results, filename="entanglement\_simulation\_results.json")  
   print("\n=== SIMULATION RESULTS ===")  
   print(json\_output)  
    
   # Example of retrieving and printing specific data  
   print("\n--- Example: First few detailed mappings ---")  
   for k, v in list(results.numerical\_mappings.items())[:3]:  
   print(f" {k}: Entanglement={v['entanglement']:.4f}, Above Threshold={v['above\_threshold']}")  
    
   print("\n--- Example: Lambda Contributions ---")  
   for k, v in results.lambda\_contributions.items():  
   print(f" {k}: {v:.4f}")  
    
   print("\n--- Example: Mean Entanglement across all measurements ---")  
   print(f" {np.**mean**(results.entanglement\_values):.4f}")

**Tab 2**

[Gemini AI research about quantum entanglement predictions](https://docs.google.com/document/d/1Hi5fRqEF1SiSWs_mmlGTZrGuxlOVH605seTA5P2VL3I/edit?usp=drivesdk)