```
# ------
# Mobile Radiopharmaceutical Synthesis Unit (MRSU) Pipeline v5
# (Forward-Deployed "Nuclear Medical Heavy Tank")
#
# This script simulates a full 4-stage "on-demand" pipeline for a
# forward-deployed, heavily-shielded mobile unit.
#
# The goal is to synthesize short-lived radiopharmaceuticals from
# stable precursors and immediately use them for diagnostics or
# therapy (Theranostics).
#
# 1. (Stage 0) Ligand Analysis:
#    Analyzes the guiding molecule (peptide/ligand) for chelation
properties.
#
# 2. (Stage 1) Ligand Conformation:
#    Uses quantum-informed lasers to fold the ligand into the optimal
#    3D shape to accept a radioactive isotope.
#
# 3. (Stage 2) Radiosynthesis (Tagging):
#    The "hot cell" stage. Selects a radioactive isotope (e.g., Ga-68)
#    and "tags" it to the folded ligand, primed by S1 results.
#
# 4. (Stage 3) Targeted Theranostics:
#    The "patient" stage. Uses the just-synthesized drug for
#    diagnostics (imaging) or bimodal therapy (activating the
#    drug's radiation at the target site). Primed by S2 purity.
# ------

import math
import random
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from typing import List, Tuple, Dict, Any

# --------------------------------
# --- SHARED/HYBRID CLASSES ---
# --------------------------------

class TargetMolecule:
    """Represents the target for the *entire* pipeline."""
    def __init__(self, ligand_name: str, synthesis_target: str,
medical_procedure: str):
        self.ligand_name = ligand_name           # e.g., "DOTA-TATE
Peptide"
        self.synthesis_target = synthesis_target # e.g., "Ga-68
Tagging"
```

```
        self.medical_procedure = medical_procedure # e.g., "Tumor
Imaging"

# ---------------------------------------------------------------
# --- STAGE 0: LIGAND ANALYSIS (formerly Hydrocarbon Sequencing) ---
# ---------------------------------------------------------------

class LigandSignature:
    """Represents the sequenced properties of the guiding molecule."""
    def __init__(self, peptide_length: int, chelation_sites: int,
steric_hindrance: float):
        self.peptide_length = peptide_length       # e.g., number of
amino acids
        self.chelation_sites = chelation_sites   # e.g., "arms" of the
DOTA cage
        self.steric_hindrance = steric_hindrance # How physically
"crowded" the tag site is

def analyze_ligand_precursor(molecule: TargetMolecule) ->
LigandSignature:
    """Simulates a rapid analysis of the guiding molecule (ligand)."""
    print(f"[SCAN-S0] Analyzing ligand: {molecule.ligand_name}...")
    # Simulate properties based on name
    if "DOTA" in molecule.ligand_name:
        length = random.randint(8, 12)
        sites = random.randint(3, 6) # DOTA-TATE
        hindrance = random.uniform(0.1, 0.3)
    else:
        length = random.randint(5, 20)
        sites = random.randint(1, 4)
        hindrance = random.uniform(0.2, 0.5)

    print(f"[SCAN-S0] Peptide Length: {length} | Chelation Sites:
{sites} | Steric Hindrance: {round(hindrance, 2)}")
    return LigandSignature(length, sites, hindrance)

def interpret_folding_priming(sig: LigandSignature) -> Dict[str,
float]:
    """Uses Ligand data to prime the initial S1 Folding parameters."""

    # More chelation sites require a more specific, curved "pocket"
    primed_curvature = 0.1 + (sig.chelation_sites * 0.05)

    # Longer peptides are less rigid, requiring more entropy control
    primed_entropy_bias = 0.8 - (sig.peptide_length * 0.01) -
sig.steric_hindrance

    print(f"[PRIME-S0->S1] Primed Fold Curvature:
```

```
{round(primed_curvature, 2)} | Entropy Bias:
{round(primed_entropy_bias, 2)}")
    return {"curvature": primed_curvature, "entropy_bias":
primed_entropy_bias}

# ------------------------------------------------------------
# --- STAGE 1: LIGAND CONFORMATION (formerly Matter Folding) ---
# ------------------------------------------------------------

class GammaSignature:
    """Represents the gamma backscatter signature for folding."""
    def __init__(self, energy: float, angle: float):
        self.energy = energy
        self.angle = angle

class FoldingParams:
    """Internal parameters derived from the gamma scan."""
    def __init__(self, phase_shift: float, curvature: float,
entropy_bias: float):
        self.phase_shift = phase_shift
        self.curvature = curvature         # How tightly folded the
chelation pocket is
        self.entropy_bias = entropy_bias   # Stability of the fold

class LaserConfig:
    """Configuration for the ligand-folding laser."""
    def __init__(self, wavelength: float, pulse_width: float,
coherence: float):
        self.wavelength = wavelength
        self.pulse_width = pulse_width
        self.coherence = coherence

def capture_gamma_backscatter(molecule: TargetMolecule) ->
GammaSignature:
    """Simulates gamma scan for initial folding."""
    print(f"[SCAN-S1] Scanning ligand conformation:
{molecule.ligand_name}")
    energy = round(random.uniform(1.0, 1.5), 2)
    angle = round(random.uniform(30.0, 60.0), 2)
    return GammaSignature(energy, angle)

def interpret_folding_params(sig: GammaSignature, priming_data:
Dict[str, float]) -> FoldingParams:
    """Interprets gamma signature AND S0 priming data to get folding
parameters."""
    phase_shift = sig.energy * 0.85
    # Use primed data as the starting point
    curvature = priming_data.get("curvature", 0.2) +
```

```
math.tan(math.radians(sig.angle)) * 0.1
    entropy_bias = priming_data.get("entropy_bias", 0.7) / (1.0 +
sig.energy)
    return FoldingParams(phase_shift, curvature, entropy_bias)

def derive_laser_geometry(params: FoldingParams) -> LaserConfig:
    """Configures laser based on folding parameters."""
    wavelength = 400.0 + params.phase_shift * 10.0
    pulse_width = 50.0 - params.curvature * 5.0
    coherence = 1.0 - params.entropy_bias
    return LaserConfig(wavelength, pulse_width, coherence)

def recalculate_band_states(params: FoldingParams, config:
LaserConfig) -> Tuple[float, float]:
    """Recalculates electron band states during folding."""
    print("[BAND-S1] Recalculating ligand electron band states...")
    band_gap = max(1.5 - (params.curvature * 2.0 + params.entropy_bias
* 1.2), 0.5)
    orbital_shift = config.coherence * 0.3
    print(f"[BAND-S1] Band Gap: {round(band_gap, 3)} eV | Orbital
Shift: {round(orbital_shift, 3)} units")
    return band_gap, orbital_shift

def simulate_conformation_skyrmions(iteration: int) -> float:
    """Simulates skyrmion tunneling dynamics for folding stability."""
    print(f"[SKYRMION-S1] Simulating conformation dynamics at
iteration {iteration}...")
    position = np.sin(iteration * 0.5) * 5
    tunneling_current = 0.8 + np.cos(iteration * 0.3) * 0.2
    spin_density = np.exp(-abs(position)) * tunneling_current
    print(f"[SKYRMION-S1] Position: {round(position,2)} | Current:
{round(tunneling_current,2)} | Spin Density: {round(spin_density,3)}")
    return spin_density

def fold_ligand(config: LaserConfig) -> bool:
    """Executes the laser-based ligand folding."""
    if config.coherence < 0.5:
        print("[ERROR-S1] Laser coherence too low. Folding
aborted.\n")
        return False
    print("[SUCCESS-S1] Folding completed. Ligand chelation site
exposed.\n")
    return True

def run_conformation_stage(molecule: TargetMolecule, priming_data:
Dict[str, float]) -> Tuple[FoldingParams, List[float], float, bool]:
    """The adaptive loop for Stage 1: Folding."""
    sig = capture_gamma_backscatter(molecule)
```

```
    params = interpret_folding_params(sig, priming_data)
    config = derive_laser_geometry(params)

    band_gap_history = []
    spin_density_history = []
    success_flag = False

    for i in range(3): # Short loop
        print(f"[LOOP-S1] Iteration {i+1}: Evaluating conformation
stability...")

        spin_density = simulate_conformation_skyrmions(i)
        spin_density_history.append(spin_density)

        band_gap, _ = recalculate_band_states(params, config)
        band_gap_history.append(band_gap)

        if spin_density < 0.7:
            print("[ADAPT-S1] Spin density low. Enhancing curvature to
stabilize pocket.")
            params.curvature *= 1.1
            config = derive_laser_geometry(params)

        if config.coherence < 0.6 or band_gap < 0.7:
            print("[ADAPT-S1] Coherence or band gap unstable.
Adjusting entropy bias.")
            params.entropy_bias *= 0.9
            config = derive_laser_geometry(params)

        if fold_ligand(config):
            print(f"[RESULT-S1] {molecule.ligand_name} successfully
folded.\n")
            success_flag = True
            break
        print("[RETRY-S1] Folding failed. Retrying...\n")

    final_spin_density = spin_density_history[-1] if
spin_density_history else 0.0
    return params, band_gap_history, final_spin_density, success_flag

# ----------------------------------------------------------------------
# --- STAGE 2: RADIOSYNTHESIS (TAGGING) (formerly MOC Cyclization) ---
# ----------------------------------------------------------------------

class RadioComplex:
    """Defines the selected radiosynthesis 'kit'."""
    def __init__(self, isotope: str, ligand_type: str,
chelation_affinity: float, base_stability: float):
```

```
        self.isotope = isotope                  # e.g., 'Ga-68',
'Tc-99m', 'Lu-177'
        self.ligand_type = ligand_type          # e.g., 'DOTA', 'PSMA'
        self.chelation_affinity = chelation_affinity # Base binding
strength
        self.base_stability = base_stability    # Base radiolytic
stability

class RadioReactionState:
    """Parameters defining the 'hot cell' chemical environment."""
    def __init__(self, tagging_efficiency: float,
radiolytic_stability: float):
        self.tagging_efficiency = tagging_efficiency     # Speed of
the tagging reaction
        self.radiolytic_stability = radiolytic_stability # How well
the drug resists self-destruction

class SpectrumSignature:
    """Represents the CD signature of the FOLDED ligand."""
    def __init__(self, ellipticity: float, absorption_peak: float):
        self.ellipticity = ellipticity
        self.absorption_peak = absorption_peak

class MetamaterialParams:
    """Parameters for the 'hot cell' stabilization scaffold."""
    def __init__(self, tensile_stress: float, pore_density: float,
confinement_rigidity: float):
        self.tensile_stress = tensile_stress
        self.pore_density = pore_density
        self.confinement_rigidity = confinement_rigidity

class SkyrmionGateConfig:
    """Config for the field controlling the tagging reaction."""
    def __init__(self, magnetic_field: float, skyrmion_density: float,
optical_polarization: float):
        self.magnetic_field = magnetic_field
        self.skyrmion_density = skyrmion_density
        self.optical_polarization = optical_polarization

def select_radiosynthesis_kit(synthesis_target: str, fold_params:
FoldingParams) -> RadioComplex:
    """Selects isotope and ligand, primed by S1 fold quality."""
    print(f"[KIT-S2] Selecting radiosynthesis kit for:
{synthesis_target}")

    # Base affinity and stability for the isotope
    if "Ga-68" in synthesis_target:
        isotope, ligand, base_affinity, base_stability = 'Ga-68',
```

```
'DOTA', 1.2, 0.8
    elif "Tc-99m" in synthesis_target:
        isotope, ligand, base_affinity, base_stability = 'Tc-99m',
'HYNIC', 1.0, 0.9
    elif "Lu-177" in synthesis_target:
        isotope, ligand, base_affinity, base_stability = 'Lu-177',
'DOTA', 1.3, 0.7 # Therapeutic, less stable
    else:
        isotope, ligand, base_affinity, base_stability = 'F-18',
'FDG-like', 0.8, 0.95

    # --- SALIENT COMBINATION (S1->S2) ---
    # A better fold (high curvature) from S1 *directly* improves
chelation affinity
    curvature_boost = fold_params.curvature * 0.5
    primed_affinity = base_affinity + curvature_boost
    print(f"[PRIME-S1->S2] S1 Fold Curvature
({round(fold_params.curvature, 2)}) primed Chelation Affinity to
{round(primed_affinity, 2)}")

    return RadioComplex(isotope, ligand, primed_affinity,
base_stability)

def capture_cd_spectrum(molecule: TargetMolecule) ->
SpectrumSignature:
    """Scans the folded ligand to confirm conformation."""
    print(f"[CD-SCAN-S2] Scanning folded ligand:
{molecule.ligand_name}...")
    ellipticity = round(random.uniform(-15.0, 15.0), 1)
    peak = round(random.uniform(280.0, 320.0), 1)
    return SpectrumSignature(ellipticity, peak)

def interpret_tagging_params(sig: SpectrumSignature, complex:
RadioComplex) -> Tuple[MetamaterialParams, RadioReactionState]:
    """Interprets CD scan to derive metamaterial and reaction
parameters."""

    # Metamaterial Derivation (Physical confinement in hot cell)
    rigidity = 1.0 + abs(sig.ellipticity) / 10.0
    tensile_stress = sig.absorption_peak / 300.0 * 1.2
    pore_density = 0.5 + sig.absorption_peak * sig.ellipticity *
0.0001
    meta_params = MetamaterialParams(tensile_stress, pore_density,
rigidity)

    # Reaction State Derivation (Chemical environment)
    # High affinity (from MOC) = high tagging efficiency
    tagging_efficiency = 1.0 + complex.chelation_affinity * 0.2
```

```
    # Trade-off: High-energy isotopes have *lower* stability
    radiolytic_stability = complex.base_stability * (1.0 -
(complex.chelation_affinity * 0.1))

    reaction_state = RadioReactionState(tagging_efficiency,
radiolytic_stability)
    print(f"[REACTION-S2] Tagging Efficiency:
{round(tagging_efficiency, 2)} | Radiolytic Stability:
{round(radiolytic_stability, 2)}")

    return meta_params, reaction_state

def derive_skyrmion_gate_geometry(params: MetamaterialParams, sig:
SpectrumSignature) -> SkyrmionGateConfig:
    """Derives magnetic field and optical gate parameters."""
    magnetic_field = 0.8 + params.confinement_rigidity * 0.3
    skyrmion_density = math.log(params.tensile_stress + 1.0) * 5.0
    optical_polarization = -sig.ellipticity / 15.0
    return SkyrmionGateConfig(magnetic_field, skyrmion_density,
optical_polarization)

def recalculate_tagging_field(meta_params: MetamaterialParams,
skyrmion_config: SkyrmionGateConfig, reaction_state:
RadioReactionState) -> Tuple[float, float, float]:
    """Determines optical mode coupling and transport loss."""
    print("[RECAL-S2] Recalculating radiolytic stabilization
field...")

    base_coupling = min(skyrmion_config.skyrmion_density / 8.0, 0.98)
* (1.0 - abs(skyrmion_config.optical_polarization) * 0.1)

    # Chemical Influence Factor (Tagging Eff. vs Stability)
    chemical_influence = 1.0 + (reaction_state.tagging_efficiency -
(1.0 - reaction_state.radiolytic_stability)) * 0.1
    combined_coupling = min(base_coupling * chemical_influence, 0.99)
    transport_loss = max(0.01 + 1.0 /
(meta_params.confinement_rigidity * 10.0), 0.05)

    print(f"[RECAL-S2] Chemical Influence: {round(chemical_influence,
3)} | Combined Coupling: {round(combined_coupling, 3)}")
    return combined_coupling, transport_loss, chemical_influence

def simulate_radiotagging_dynamics(iteration: int, chemical_influence:
float, final_fold_spin_density: float) -> float:
    """
    Simulates the efficiency of the field-driven tagging.
    *** SALIENT COMBINATION (S1->S2) ***
```

```
    Uses final_fold_spin_density from S1 to provide a "spin boost".
    """
    print(f"[TAGGING-S2] Simulating radiotagging field efficiency
(Iter: {iteration})...")
    skyrmion_stability = 0.8 + np.sin(iteration * 0.4) * 0.15

    # --- SALIENT COMBINATION ---
    spin_boost = final_fold_spin_density * 0.1

    tunneling_efficiency = max(0.9 * skyrmion_stability *
chemical_influence + spin_boost - iteration * 0.05, 0.5)

    print(f"[TAGGING-S2] (Fold Spin Boost: {round(spin_boost, 3)}) |
Tagging Field Efficiency: {round(tunneling_efficiency, 3)}")
    return tunneling_efficiency

def execute_radiotagging(config: SkyrmionGateConfig, efficiency:
float) -> Tuple[bool, float, float]:
    """Attempts the final tagging, incorporating field efficiency."""

    # Radiochemical Purity (RCP)
    radiochemical_purity = efficiency * random.uniform(0.95, 0.99)
    # Specific Activity (SA)
    specific_activity = (efficiency +
abs(config.optical_polarization)) * random.uniform(50.0, 80.0)

    if efficiency < 0.75:
        print("[ERROR-S2] Tagging Field Efficiency too low. Synthesis
aborted.\n")
        return False, 0.0, 0.0

    print(f"[SUCCESS-S2] Radiosynthesis complete.")
    return True, radiochemical_purity, specific_activity

def run_radiosynthesis_stage(molecule: TargetMolecule, fold_params:
FoldingParams, final_fold_spin_density: float) -> Tuple[float, float,
List[float]]:
    """The iterative loop for Stage 2: Radiosynthesis."""

    # S1->S2 Link: Pass fold_params to kit selection
    complex = select_radiosynthesis_kit(molecule.synthesis_target,
fold_params)

    sig = capture_cd_spectrum(molecule)
    meta_params, reaction_state = interpret_tagging_params(sig,
complex)
    skyrmion_config = derive_skyrmion_gate_geometry(meta_params, sig)
```

```
    efficiency_history = []
    final_rcp, final_sa = 0.0, 0.0

    for i in range(5):
        print(f"[LOOP-S2] Iteration {i+1}: Optimizing radiotagging
field...")

        combined_coupling, _, chemical_influence =
recalculate_tagging_field(meta_params, skyrmion_config,
reaction_state)

        # S1->S2 Link: Pass spin_density to field simulation
        tunneling_efficiency = simulate_radiotagging_dynamics(i,
chemical_influence, final_fold_spin_density)
        efficiency_history.append(tunneling_efficiency)

        if tunneling_efficiency < 0.85:
            print("[ADAPT-S2] Low efficiency. Boosting field to
compensate for instability.")
            reaction_state.tagging_efficiency *= 1.05
            skyrmion_config.skyrmion_density *= 1.05

        elif combined_coupling < 0.8:
            print("[ADAPT-S2] Low coupling. Tightening confinement.")
            meta_params.confinement_rigidity *= 1.02
            skyrmion_config =
derive_skyrmion_gate_geometry(meta_params, sig)

        success, current_rcp, current_sa =
execute_radiotagging(skyrmion_config, tunneling_efficiency)

        if success and current_rcp > 0.95: # Target >95% RCP
            final_rcp = current_rcp
            final_sa = current_sa
            print(f"[RESULT-S2] {complex.isotope} tagged to
{molecule.ligand_name} at optimal purity.")
            print(f"    Radiochemical Purity (RCP):
{round(final_rcp*100, 1)}% | Specific Activity (SA): {round(final_sa,
1)} GBq/µmol\n")
            return final_rcp, final_sa, efficiency_history

        print("[RETRY-S2] Synthesis not optimal. Retrying...\n")

    return final_rcp, final_sa, efficiency_history

# ---------------------------------------------------------------------
# --- STAGE 3: TARGETED THERANOSTICS (formerly Medical Intervention)
---
```

```
# ----------------------------------------------------------------------

class BioSignature:
    """Represents the bio-physical state of the target tissue."""
    def __init__(self, impedance_contrast: float, biomarker_level:
float):
        self.impedance_contrast = impedance_contrast
        self.biomarker_level = biomarker_level

class TheranosticProtocol:
    """Parameters for the diagnostic scan or bimodal therapy."""
    def __init__(self, required_radiation_dose: float,
scan_calibration_factor: float, scaffold_rigidity: float):
        self.required_radiation_dose = required_radiation_dose # Dose
for therapy, or 0 for diagnosis
        self.scan_calibration_factor = scan_calibration_factor #
Primed by S2 purity
        self.scaffold_rigidity = scaffold_rigidity

class MIMOArrayConfig:
    """Configuration for the exciton array (scanner or
potentiator)."""
    def __init__(self, exciton_wavelength: float,
phase_lock_threshold: float, array_gain: float):
        self.exciton_wavelength = exciton_wavelength
        self.phase_lock_threshold = phase_lock_threshold
        self.array_gain = array_gain

def capture_bio_signature(molecule: TargetMolecule) -> BioSignature:
    """Scans the target tissue in the forward-deployed environment."""
    print(f"[SCAN-S3] Acquiring bio-signature for:
{molecule.medical_procedure} at target site.")
    impedance = random.uniform(0.5, 2.0)
    biomarker = random.uniform(1.0, 10.0)
    return BioSignature(impedance, biomarker)

def interpret_theranostic_protocol(sig: BioSignature, rcp: float, sa:
float, procedure: str) -> TheranosticProtocol:
    """
    Derives the intervention protocol.
    *** SALIENT COMBINATION (S2->S3) ***
    Uses Radiochemical Purity (RCP) and Specific Activity (SA) from S2
    to calibrate the scan.
    """

    # --- SALIENT COMBINATION ---
    # A purer, more active drug (high RCP, high SA) gives a *much*
cleaner
```

```
    # signal, requiring less gain.
    scan_calibration_factor = 1.0 + (rcp * sa * 0.01)
    print(f"[PRIME-S2->S3] S2 Purity/Activity (RCP: {round(rcp,2)},
SA: {round(sa,1)})")
    print(f"[PRIME-S2->S3] Primed Scan Calibration Factor:
{round(scan_calibration_factor, 2)}")

    # If it's just imaging, dose is 0. If therapy, calculate dose.
    if "Imaging" in procedure:
        required_dose = 0.0
    else:
        required_dose = 3.0 / (sig.impedance_contrast + 0.1)

    scaffold_rigidity = 1.0 / (1.0 + sig.biomarker_level * 0.05)

    return TheranosticProtocol(required_dose, scan_calibration_factor,
scaffold_rigidity)

def derive_mimo_array_config(protocol: TheranosticProtocol, sig:
BioSignature) -> MIMOArrayConfig:
    """Configures the exciton array based on scan calibration."""
    wavelength = 980.0 - (sig.biomarker_level * 10.0)
    threshold = 0.95 - (protocol.scaffold_rigidity * 0.1)
    # Calibrated scan factor reduces the required gain
    array_gain = 5.0 / (protocol.scan_calibration_factor + 0.1)

    return MIMOArrayConfig(wavelength, threshold, array_gain)

def simulate_theranostic_field(protocol: TheranosticProtocol, config:
MIMOArrayConfig, iteration: int) -> Tuple[float, float, float]:
    """Simulates the exciton field (for imaging or therapy)."""
    print(f"[FIELD-S3] Simulating theranostic field (Iter:
{iteration})...")

    base_penetration = 100.0 * config.array_gain
    phase_scatter = 0.1 + (iteration * 0.02) /
(protocol.scaffold_rigidity)

    # Targeting success is based on scan calibration and low scatter
    targeting_success = max(0.99 - phase_scatter, 0.1) *
(protocol.scan_calibration_factor / 1.5)

    print(f"[FIELD-S3] Penetration: {round(base_penetration, 1)}µm |
Scatter: {round(phase_scatter, 3)} | Targeting Success:
{round(targeting_success, 3)}")
    return base_penetration, phase_scatter, targeting_success

def execute_theranostic_procedure(protocol: TheranosticProtocol,
```

```
success_factor: float, procedure: str) -> Tuple[float, float]:
    """Executes the final diagnostic scan or bimodal therapy."""

    if "Imaging" in procedure:
        # For imaging, "Efficacy" is the Signal-to-Noise Ratio (SNR)
        # "Damage" is the systemic radiation load (low, as it's
diagnostic)
        snr = success_factor * random.uniform(20.0, 30.0)
        systemic_load = 1.0 / (success_factor + 0.1) *
random.uniform(0.5, 1.0)
        print(f"[SUCCESS-S3] Diagnostic scan complete. SNR:
{round(snr, 1)}")
        return snr, systemic_load
    else:
        # For therapy, "Efficacy" is the Tumor Uptake Ratio
        # "Damage" is the systemic load (higher, as it's therapeutic)
        uptake_ratio = success_factor * random.uniform(0.9, 1.2)
        systemic_load = (protocol.required_radiation_dose * 0.5) /
(success_factor + 0.1)
        print(f"[SUCCESS-S3] Bimodal therapy complete. Tumor Uptake
Ratio: {round(uptake_ratio, 2)}")
        return uptake_ratio, systemic_load

def run_theranostic_stage(molecule: TargetMolecule, rcp: float, sa:
float) -> Tuple[float, float, List[float]]:
    """The iterative loop for Stage 3: Theranostics."""

    sig = capture_bio_signature(molecule)

    # S2->S3 Link: Pass RCP and SA to protocol interpretation
    protocol = interpret_theranostic_protocol(sig, rcp, sa,
molecule.medical_procedure)

    config = derive_mimo_array_config(protocol, sig)

    success_history = []
    final_efficacy, final_damage = 0.0, 0.0

    for i in range(5):
        print(f"[LOOP-S3] Iteration {i+1}: Locking on target...")

        _, _, targeting_success = simulate_theranostic_field(protocol,
config, i)
        success_history.append(targeting_success)

        if targeting_success < 0.8:
            print("[ADAPT-S3] Low targeting success. Increasing array
gain.")
```

```
            config.array_gain *= 1.1
        elif targeting_success > 0.98:
            print("[ADAPT-S3] Target locked.")
            final_efficacy, final_damage =
execute_theranostic_procedure(protocol, targeting_success,
molecule.medical_procedure)
            break

        if i == 4: # Final attempt
            final_efficacy, final_damage =
execute_theranostic_procedure(protocol, targeting_success,
molecule.medical_procedure)

    return final_efficacy, final_damage, success_history

# ------------------------------
# Visualization Routines
# ------------------------------

def visualize_conformation_skyrmions(molecule: TargetMolecule,
spin_density_history: List[float]):
    """Visualizes the 3D evolution of Skyrmion dynamics from Stage
1."""
    if not spin_density_history: return
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')
    iterations = list(range(1, len(spin_density_history) + 1))
    positions = [np.sin(i * 0.5) * 5 for i in iterations]
    currents = [0.8 + np.cos(i * 0.3) * 0.2 for i in iterations]
    ax.plot(positions, currents, spin_density_history, marker='o',
color='cyan')
    ax.set_title(f"S1: Ligand Conformation Stability:
{molecule.ligand_name}")
    ax.set_xlabel("Position"), ax.set_ylabel("Tunneling Current"),
ax.set_zlabel("Spin Density")
    plt.tight_layout(), plt.show()

def visualize_radiotagging_field(molecule: TargetMolecule,
efficiency_history: List[float]):
    """Visualizes the Skyrmion-Optics Tunneling Efficiency from Stage
2."""
    if not efficiency_history: return
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')
    iterations = list(range(1, len(efficiency_history) + 1))
    skyrmion_density = [4.0 + np.sin(i*0.6) * 1.5 + i*0.5 for i in
iterations]
    chemical_influence = [0.8 + np.cos(i*0.3) * 0.2 for i in
```

```
iterations]
    ax.plot(skyrmion_density, chemical_influence, efficiency_history,
marker='o', color='lime')
    ax.set_title(f"S2: Radiotagging Field Efficiency:
{molecule.synthesis_target}")
    ax.set_xlabel("Skyrmion Density"), ax.set_ylabel("Chemical
Influence"), ax.set_zlabel("Tagging Field Efficiency")
    plt.tight_layout(), plt.show()

def visualize_theranostic_targeting(molecule: TargetMolecule,
success_history: List[float]):
    """Visualizes the Exciton Targeting Success from Stage 3."""
    if not success_history: return
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')
    iterations = list(range(1, len(success_history) + 1))
    penetration = [80.0 + i*5 + np.sin(i*0.5)*10 for i in iterations]
    scatter = [0.2 - i*0.02 + np.cos(i*0.5)*0.05 for i in iterations]
    ax.plot(penetration, scatter, success_history, marker='o',
color='magenta')
    ax.set_title(f"S3: Theranostic Targeting Success:
{molecule.medical_procedure}")
    ax.set_xlabel("Field Penetration (μm)"), ax.set_ylabel("Phase
Scatter"), ax.set_zlabel("Targeting Success Factor")
    plt.tight_layout(), plt.show()

# ------------------------------
# Master Commentary
# ------------------------------

def master_commentary(molecule, s0_results, s1_results, s2_results,
s3_results):
    """Provides a unified summary of the full 'Heavy Tank' run."""

    (s0_sig, s0_prime) = s0_results
    (s1_params, _, s1_spin, s1_success) = s1_results
    (s2_rcp, s2_sa, _) = s2_results
    (s3_efficacy, s3_damage, _) = s3_results


print(f"\n\n============================================================")
    print(f"MRSU 'HEAVY TANK' MISSION REPORT: {molecule.ligand_name}")
    print(f"DEPLOYMENT: Forward-Deployed (Simulated)")

print(f"============================================================\n")

    print("--- STAGE 0: LIGAND ANALYSIS ---")
    print(f"[STATUS] Ligand {molecule.ligand_name} analyzed.")
```

```
    print(f"    Discovered Chelation Sites: {s0_sig.chelation_sites}")
    print(f"    Salient Link (S0->S1): Primed S1 Fold Curvature to
{round(s0_prime.get('curvature', 0), 2)}")

    print("\n--- STAGE 1: LIGAND CONFORMATION ---")
    if not s1_success:
        print("[STATUS] FOLDING FAILED. Proceeding with sub-optimal
conformation.")
    else:
        print("[STATUS] FOLDING SUCCESSFUL.")
    print(f"    Final Fold Curvature: {round(s1_params.curvature,
3)}")
    print(f"    Final S1 Spin Density: {round(s1_spin, 3)}")
    print(f"    Salient Link (S1->S2): Primed S2 Chelation Affinity
and Tagging Efficiency.")

    print("\n--- STAGE 2: RADIOSYNTHESIS (TAGGING) ---")
    if s2_rcp > 0.9:
        print(f"[STATUS] SYNTHESIS SUCCESSFUL.
({molecule.synthesis_target})")
        print(f"    Final Radiochemical Purity (RCP): {round(s2_rcp *
100, 1)}%")
        print(f"    Final Specific Activity (SA): {round(s2_sa, 1)}
GBq/µmol")
    else:
        print("[STATUS] SYNTHESIS FAILED. Purity < 90%.")
    print(f"    Salient Link (S2->S3): Primed S3 Scan Calibration
Factor.")

    print("\n--- STAGE 3: TARGETED THERANOSTICS ---")
    print(f"[STATUS] Procedure '{molecule.medical_procedure}'
complete.")
    if "Imaging" in molecule.medical_procedure:
        print(f"    Final Diagnostic SNR: {round(s3_efficacy, 2)}")
        print(f"    Final Systemic Radiation Load: {round(s3_damage,
2)} mSv")
    else:
        print(f"    Final Tumor Uptake Ratio: {round(s3_efficacy,
3)}")
        print(f"    Final Systemic Radiation Load: {round(s3_damage,
2)} mSv")

    print("\n--- SALIENT ANALYSIS ---")
    print(f"""
    This mission demonstrates a complete on-demand, forward-deployed
    theranostic pipeline.

    (S0->S1): Ligand analysis ({s0_sig.chelation_sites} sites)
```

```
correctly
    primed the (S1) conformation fold (Curvature:
{round(s1_params.curvature, 2)}).

    (S1->S2): The S1 fold quality and spin density ({round(s1_spin,
2)}) were
    passed to the (S2) 'hot cell', enabling successful synthesis of
    '{molecule.synthesis_target}' to {round(s2_rcp * 100, 1)}% RCP.

    (S2->S3): The high purity from S2 primed the (S3) scanner,
    allowing for a high-efficacy procedure ({round(s3_efficacy, 2)})
with
    minimal systemic load ({round(s3_damage, 2)} mSv).

    The entire 'bench-to-bedside' (or 'tank-to-patient') process
    was saliently linked and completed successfully.
    """)

# --------------------------------
# Main Execution
# --------------------------------

def main():
    print("[INIT] Mobile Radiopharmaceutical Synthesis Unit (MRSU) v5
Pipeline Active.\n")
    print("[INIT] WARNING: 'Heavy Tank' shielding is active. Running
in hot-cell simulation mode.\n")

    # Define mission targets for the "Heavy Tank"
    mission_targets = [
        TargetMolecule(
            ligand_name="DOTA-TATE Peptide",
            synthesis_target="Ga-68 Tagging",
            medical_procedure="Neuroendocrine Tumor Imaging"
        ),
        TargetMolecule(
            ligand_name="PSMA-617 Ligand",
            synthesis_target="Lu-177 Tagging",
            medical_procedure="Prostate Cancer Bimodal Therapy"
        ),
        TargetMolecule(
            ligand_name="HYNIC-TOC Peptide",
            synthesis_target="Tc-99m Tagging",
            medical_procedure="Field Trauma Bone Scan Imaging"
        )
    ]

    for mol in mission_targets:
```

```
        print(f"\n--- Processing New Mission: {mol.ligand_name} for
{mol.medical_procedure} ---")

        # --- STAGE 0 ---
        s0_sig = analyze_ligand_precursor(mol)
        s0_priming_data = interpret_folding_priming(s0_sig)
        s0_results = (s0_sig, s0_priming_data)

        # --- STAGE 1 ---
        s1_params, s1_band_gap_hist, s1_spin, s1_success =
run_conformation_stage(mol, s0_priming_data)
        s1_results = (s1_params, s1_band_gap_hist, s1_spin,
s1_success)

        # --- STAGE 2 ---
        s2_rcp, s2_sa, s2_eff_hist = run_radiosynthesis_stage(mol,
s1_params, s1_spin)
        s2_results = (s2_rcp, s2_sa, s2_eff_hist)

        # --- STAGE 3 ---
        s3_efficacy, s3_damage, s3_success_hist =
run_theranostic_stage(mol, s2_rcp, s2_sa)
        s3_results = (s3_efficacy, s3_damage, s3_success_hist)

        # --- VISUALIZATION ---
        visualize_conformation_skyrmions(mol, s1_band_gap_hist)
        visualize_radiotagging_field(mol, s2_eff_hist)
        visualize_theranostic_targeting(mol, s3_success_hist)

        # --- REPORT ---
        master_commentary(mol, s0_results, s1_results, s2_results,
s3_results)

    print("\n[END] All forward-deployed missions complete. MRSU
standing by.")

if __name__ == "__main__":
    main()
```