

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import
gaussian_filter1d
from qutip import basis, sigmaz, sigmam,
mesolve, Options
from scipy.interpolate import interp1d
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.tensorboard import
SummaryWriter
from stl import mesh
from flask import Flask, jsonify, request

device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")

# Constants
phi = (1 + np.sqrt(5)) / 2 # Golden ratio  $\approx$ 
1.618
```

```
golden_angle = 360 / phi**2 #  $\approx 137.508^\circ$   
psi = 144.0 # Resonance frequency  
alpha_inv_real = 137.036 # Fine-  
structure constant inverse  
chevron_angle = 60 * np.pi / 180 #  
Chevron V-angle (in radians)  
chi = 2 * np.pi / chevron_angle #  
Modulation frequency
```

```
# Agent-Glyph: Handles decay and  
diagnostics
```

```
class AgentGlyph:
```

```
    def glyph_decay_map(self, expect_sz,  
modulated_radII):
```

```
        expect_sz = torch.tensor(expect_sz,  
device=device)
```

```
        modulated_radII =  
torch.tensor(modulated_radII,  
device=device)
```

```
        interp = interp1d(np.linspace(0, 1,  
len(expect_sz.cpu()))), expect_sz.cpu())
```

```
    decay_factor =  
torch.tensor(interp(np.linspace(0, 1,  
len(modulated_radii.cpu()))),  
device=device)  
    distorted_radii = modulated_radii * (1  
- (1 - decay_factor) * 0.5)  
    return distorted_radii.cpu().numpy()
```

```
def integrity_track(self, expect_sz):  
    states = []  
    for val in expect_sz:  
        if val > 0.8:  
            states.append("Stable Glyph")  
        elif 0.5 < val <= 0.8:  
            states.append("Phase Fracture")  
        else:  
            states.append("Anima Collapse")  
    return states
```

```
def codex_name_decay(self,  
mean_decay):
```

```
    return "Stable Harmony" if  
mean_decay > 0.8 else "ψ/φ Fracture" if  
mean_decay > 0.5 else "Anima Collapse"
```

```
# Agent-Scroll: Generates harmonic  
waveforms
```

```
class AgentScroll:
```

```
    def generate_scroll_wave(self, t,  
omega_n=2 * np.pi * 10, delta=0,  
tau_i=0.01):
```

```
        n = np.arange(len(t))
```

```
        psi_t = np.sin(2 * np.pi * psi * t)
```

```
        theta_n = n * golden_angle * np.pi /  
180
```

```
        r_n = phi * (1 + 0.5 * np.sin(theta_n +  
np.sin(theta_n / chi)))
```

```
        exp_term = np.exp(1j * omega_n * t)
```

```
        envelope = 1 + 0.2 * np.sin(0.5 * t +  
delta)
```

```
        gate = (tau_i > 0.007) &  
(np.abs(np.gradient(psi_t, t)) >
```

```
np.std(np.gradient(psi_t, t)) * 1.5)
    g_t = gate[:len(t)] * envelope
    return r_n * np.real(g_t * exp_term)
```

# Agent-Brane: Manages toroidal geometry

```
class AgentBrane:
```

```
    def toroidal_helix(self, t, R=3, r=1,
mode='toroid'):
```

```
        if mode == 'toroid':
```

```
            theta = t
```

```
            phi_t = golden_angle * np.pi / 180
```

```
* t
```

```
            x = (R + r * np.cos(theta)) *
```

```
np.cos(phi_t)
```

```
            y = (R + r * np.cos(theta)) *
```

```
np.sin(phi_t)
```

```
            z = r * np.sin(theta)
```

```
        else:
```

```
            x = np.cos(t)
```

```
            y = np.sin(t)
```

```
z = t  
return x, y, z
```

```
def brane_collision(self, x,  
noise_scale=0.01):  
    x_shifted, _, _ =  
self.toroidal_helix(np.linspace(0, 10 *  
np.pi, len(x)) + np.pi)  
    return x + x_shifted +  
np.random.normal(scale=noise_scale,  
size=len(x))
```

```
def export_3d_model(self, x, y, z,  
filename='helix.obj'):  
    vertices = np.vstack((x, y, z)).T  
    faces = np.arange(len(x)) # Stub for  
line mesh  
    model =  
mesh.Mesh(np.zeros(faces.size,  
dtype=mesh.dtype))  
    for i, f in enumerate(faces):
```

```
    for j in range(3):
        model.vectors[i][j] = vertices[f[j], :]
    model.save(filename)
```

# Agent-RL: Reinforcement adaptation

```
class AgentRL:
```

```
    def __init__(self):
```

```
        self.model = nn.Linear(2, 1).to(device)
```

```
        self.optimizer =
```

```
optim.Adam(self.model.parameters(),
lr=0.01)
```

```
        self.writer =
```

```
SummaryWriter(log_dir="runs/glyph_rl")
```

```
    def forward(self, state):
```

```
        return self.model(state)
```

```
    def train_episode(self, mean_sz,
gamma_rate, ep, integrity_states):
```

```
        reward = mean_sz + (sum(1 for s in
enumerate(integrity_states) if s ==
```

```
"Stable Glyph") * 0.1)
    state = torch.tensor([mean_sz,
gamma_rate],
dtype=torch.float32).to(device)
    pred_reward = self.forward(state)
    loss = (pred_reward - reward) ** 2
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
    self.writer.add_scalar("Reward",
reward, ep)
    self.writer.add_scalar("GammaRate",
gamma_rate, ep)
    self.writer.add_scalar("MeanSz",
mean_sz, ep)
    return gamma_rate

def close(self):
    self.writer.close()

# Agent-AI: Multi-mode embedding
```



```
class AgentAI:
    def embed_input(self, input_data,
mode='text'):
        if mode == 'text':
            return np.sin(np.linspace(0, 2 *
np.pi, len(input_data)))
        elif mode == 'audio':
            return np.fft.fft(input_data)
[:len(input_data)//2].real
        elif mode == 'dna':
            mapping = {'A': 0, 'C': 90, 'G': 180, 'T':
270}
            return
np.array([mapping.get(base.upper(), 0) for
enumerate(base in input_data)]) * np.pi /
180
        elif mode == 'ai':
            return
np.random.normal(size=len(input_data))
            return np.zeros(len(input_data))
```

```
# Phase Collapse Diagnostics
def classify_collapse(mean_decay,
mean_sz, integrity_states):
    if mean_decay < 0: return "Loop
Overload"
    elif mean_sz < -0.1: return "Brane
Interference"
    elif "Anima Collapse" in integrity_states:
return "Glyph Resonance Failure"
    else: return "Decoherence Drift"
```

```
# Main simulation
def run_simulation(mode='toroid',
input_mode='text', input_data=None,
episodes=5):
    agent_glyph = AgentGlyph()
    agent_scroll = AgentScroll()
    agent_brane = AgentBrane()
    agent_rl = AgentRL()
    agent_ai = AgentAI()
```

```
t = np.linspace(0, 10 * np.pi, 1000)
```

```
t_list = np.linspace(0, 10, 100)
```

```
omega = 2 * np.pi * psi
```

```
# AI embedding
```

```
psi_t =
```

```
agent_ai.embed_input(input_data,  
mode=input_mode) if input_data else  
np.sin(2 * np.pi * psi * t)
```

```
# Scroll wave
```

```
scroll_wave =
```

```
agent_scroll.generate_scroll_wave(t)  
scroll_mean = np.mean(scroll_wave)
```

```
# RL Adapt Decay
```

```
gamma_rate = 0.1 # Default
```

```
for ep in enumerate(range(episodes)):
```

```
    expect_sz =
```

```
qutip_psi_evolution(omega, t_list,  
gamma_rate=gamma_rate)
```

```
    qutip_mean = np.mean(expect_sz)
    gamma_rate =
agent_rl.train_episode(qutip_mean,
gamma_rate, ep,
agent_glyph.integrity_track(expect_sz))

# Qutip evolution
expect_sz =
qutip_psi_evolution(omega, t_list,
gamma_rate=gamma_rate)
    qutip_mean = np.mean(expect_sz)

# Helix/Toroid
base_radius = phi
modulated = base_radius * (1 +
np.sin(t) * 0.05 * phi)
    distorted =
agent_glyph.glyph_decay_map(expect_sz,
modulated)

x, y, z = agent_brane.toroidal_helix(t,
```

```
mode=mode)
    x_distorted =
agent_brane.brane_collision(x)

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111,
projection='3d')
    ax.plot(x_distorted, y, z, label=f'AEON-
M v1.9 {mode}')
    ax.set_title(f"AEON-M v1.9 Toroidal
Helix with Glyph Decay")
    ax.legend()
    plt.show()

# Diagnostics
integrity_states =
agent_glyph.integrity_track(expect_sz)
    decay_name =
agent_glyph.codex_name_decay(qutip_m
ean)
    collapse_type =
```

```
classify_collapse(qutip_mean,  
qutip_mean, integrity_states)
```

```
    agent_rl.close()  
    print("Scroll Wave Mean:", scroll_mean)  
    print("Qutip Mean  $\langle \sigma_z \rangle$ :",  
round(qutip_mean, 5))  
    print("Decay State:", decay_name)  
    print("Phase Collapse:", collapse_type)  
    print("Integrity States (Sample):",  
integrity_states[:5])
```

```
# Flask GUI
```

```
app = Flask(__name__)
```

```
@app.route('/simulate', methods=['GET'])  
def simulate():  
    run_simulation()  
    return jsonify({"status": "Simulation  
complete"})
```

```
if __name__ == "__main__":  
    app.run(debug=True)
```