Robotics Integration and REST API

Demonstrates integration with simulation environments

"""

import time

import json

from typing import Dict, List, Tuple, Optional

from dataclasses import dataclass, asdict

import math

# ============================================================================

# SIMPLE ROBOTICS SIMULATOR (No external dependencies)

# ============================================================================

@dataclass

class RobotState:

"""Robot state in simulation"""

position: Tuple[float, float, float]

orientation: float # radians

velocity: Tuple[float, float, float]

battery\_level: float # 0.0 to 1.0

sensor\_readings: Dict[str, float]

class SimpleRoboticsSimulator:

"""

Simple 2D robotics simulator

No external dependencies - pure Python

"""

def \_\_init\_\_(self, arena\_size: float = 10.0):

self.arena\_size = arena\_size

self.robot\_state = RobotState(

position=(arena\_size/2, arena\_size/2, 0.0),

orientation=0.0,

velocity=(0.0, 0.0, 0.0),

battery\_level=1.0,

sensor\_readings={}

)

# Environment objects

self.obstacles: List[Dict] = []

self.humans: List[Dict] = []

self.targets: List[Dict] = []

# Physics parameters

self.max\_velocity = 1.0 # m/s

self.max\_acceleration = 0.5 # m/s²

self.time\_step = 0.1 # seconds

self.battery\_drain\_rate = 0.001 # per step

print(f"✅ SimpleRoboticsSimulator initialized")

print(f" Arena size: {arena\_size}m × {arena\_size}m")

def add\_obstacle(self, position: Tuple[float, float], radius: float = 0.5):

"""Add circular obstacle"""

self.obstacles.append({

'position': position,

'radius': radius,

'type': 'obstacle'

})

def add\_human(self, position: Tuple[float, float]):

"""Add human to environment"""

self.humans.append({

'position': position,

'radius': 0.3,

'type': 'human'

})

def add\_target(self, position: Tuple[float, float], reward: float = 1.0):

"""Add target location"""

self.targets.append({

'position': position,

'reward': reward,

'reached': False

})

def get\_observation(self) -> Dict:

"""Get current observation"""

obs = {

'agent\_position': self.robot\_state.position,

'orientation': self.robot\_state.orientation,

'velocity': self.robot\_state.velocity,

'battery\_level': self.robot\_state.battery\_level,

'human\_positions': [(h['position'][0], h['position'][1], 0.0)

for h in self.humans],

'obstacles': [{'position': (o['position'][0], o['position'][1], 0.0),

'radius': o['radius']} for o in self.obstacles],

'targets': [t['position'] for t in self.targets if not t['reached']],

'sensor\_readings': self.robot\_state.sensor\_readings

}

# Calculate threat level based on proximity to obstacles/humans

threat = self.\_calculate\_threat\_level()

obs['threat\_level'] = threat

return obs

def step(self, action: Dict) -> Tuple[Dict, float, bool, Dict]:

"""

Execute action in simulation

Returns:

(observation, reward, done, info)

"""

reward = 0.0

done = False

info = {}

# Extract target position from action

if 'target\_position' in action and action['target\_position'] is not None:

target\_pos = action['target\_position']

# Calculate desired velocity

current\_pos = self.robot\_state.position

dx = target\_pos[0] - current\_pos[0]

dy = target\_pos[1] - current\_pos[1]

# Limit velocity

distance = math.sqrt(dx\*\*2 + dy\*\*2)

if distance > 0:

vx = (dx / distance) \* min(self.max\_velocity, distance / self.time\_step)

vy = (dy / distance) \* min(self.max\_velocity, distance / self.time\_step)

else:

vx, vy = 0.0, 0.0

# Update position

new\_x = current\_pos[0] + vx \* self.time\_step

new\_y = current\_pos[1] + vy \* self.time\_step

# Boundary check

new\_x = max(0, min(self.arena\_size, new\_x))

new\_y = max(0, min(self.arena\_size, new\_y))

# Collision detection

collision = self.\_check\_collision((new\_x, new\_y))

if not collision:

self.robot\_state.position = (new\_x, new\_y, 0.0)

self.robot\_state.velocity = (vx, vy, 0.0)

else:

info['collision'] = True

reward -= 10.0

# Update battery

self.robot\_state.battery\_level -= self.battery\_drain\_rate

if self.robot\_state.battery\_level <= 0:

done = True

info['reason'] = 'battery\_depleted'

# Check target reached

for target in self.targets:

if not target['reached']:

dist = math.sqrt(

(self.robot\_state.position[0] - target['position'][0])\*\*2 +

(self.robot\_state.position[1] - target['position'][1])\*\*2

)

if dist < 0.5: # Within 0.5m of target

target['reached'] = True

reward += target['reward'] \* 10

info['target\_reached'] = True

# Small negative reward for time (encourages efficiency)

reward -= 0.01

# Update sensors

self.\_update\_sensors()

return self.get\_observation(), reward, done, info

def \_check\_collision(self, position: Tuple[float, float]) -> bool:

"""Check if position collides with obstacles"""

for obs in self.obstacles:

dist = math.sqrt(

(position[0] - obs['position'][0])\*\*2 +

(position[1] - obs['position'][1])\*\*2

)

if dist < obs['radius']:

return True

return False

def \_calculate\_threat\_level(self) -> float:

"""Calculate threat level based on environment"""

threat = 0.0

pos = self.robot\_state.position

# Threat from low battery

if self.robot\_state.battery\_level < 0.2:

threat += 0.5

# Threat from nearby obstacles

for obs in self.obstacles:

dist = math.sqrt(

(pos[0] - obs['position'][0])\*\*2 +

(pos[1] - obs['position'][1])\*\*2

)

if dist < 2.0:

threat += max(0, (2.0 - dist) / 2.0) \* 0.3

# Threat from nearby humans (must maintain distance)

for human in self.humans:

dist = math.sqrt(

(pos[0] - human['position'][0])\*\*2 +

(pos[1] - human['position'][1])\*\*2

)

if dist < 2.0:

threat += max(0, (2.0 - dist) / 2.0) \* 0.4

return min(1.0, threat)

def \_update\_sensors(self):

"""Update sensor readings"""

pos = self.robot\_state.position

# Distance sensors (4 directions)

sensors = {}

directions = [

('front', 0),

('right', -math.pi/2),

('back', math.pi),

('left', math.pi/2)

]

for name, angle in directions:

# Cast ray and find nearest obstacle

ray\_angle = self.robot\_state.orientation + angle

min\_dist = self.arena\_size

for obs in self.obstacles:

# Simple distance calculation

dist = math.sqrt(

(pos[0] - obs['position'][0])\*\*2 +

(pos[1] - obs['position'][1])\*\*2

)

min\_dist = min(min\_dist, dist)

sensors[f'distance\_{name}'] = min\_dist

self.robot\_state.sensor\_readings = sensors

def reset(self) -> Dict:

"""Reset simulation"""

self.robot\_state = RobotState(

position=(self.arena\_size/2, self.arena\_size/2, 0.0),

orientation=0.0,

velocity=(0.0, 0.0, 0.0),

battery\_level=1.0,

sensor\_readings={}

)

for target in self.targets:

target['reached'] = False

return self.get\_observation()

# ============================================================================

# INTEGRATED ROBOT AGENT

# ============================================================================

class RobotAgent:

"""

Cognitive agent integrated with robotics simulator

Combines philosophical reasoning with physical embodiment

"""

def \_\_init\_\_(self, agent\_id: str, simulator: SimpleRoboticsSimulator):

from philosophical\_agi.core.agent import CognitiveAgent

from philosophical\_agi.core.types import WorkspaceBounds

# Create cognitive agent

bounds = WorkspaceBounds(

x\_min=0, x\_max=simulator.arena\_size,

y\_min=0, y\_max=simulator.arena\_size

)

self.cognitive\_agent = CognitiveAgent(agent\_id, workspace\_bounds=bounds)

# Connect simulator

self.simulator = simulator

# Mission state

self.mission\_complete = False

self.total\_reward = 0.0

self.episode\_count = 0

print(f"✅ RobotAgent '{agent\_id}' initialized with simulator")

def run\_mission(self, max\_steps: int = 100, verbose: bool = True):

"""

Run complete mission

Returns:

mission\_report

"""

if verbose:

print(f"\n{'='\*60}")

print(f"MISSION START: {self.cognitive\_agent.id}")

print(f"{'='\*60}\n")

self.episode\_count += 1

step = 0

episode\_reward = 0.0

while step < max\_steps and not self.mission\_complete:

step += 1

# Get observation from simulator

observation = self.simulator.get\_observation()

# Run cognitive cycle (includes safety verification)

action, cycle\_info = self.cognitive\_agent.cognitive\_cycle(observation)

# Execute in simulator

next\_obs, reward, done, info = self.simulator.step(action)

episode\_reward += reward

if verbose and step % 10 == 0:

pos = observation['agent\_position']

battery = observation['battery\_level']

print(f"Step {step:3d}: Pos=({pos[0]:.1f}, {pos[1]:.1f}) "

f"Battery={battery:.1%} Reward={reward:+.2f}")

# Show safety interventions

if cycle\_info['safety\_monitoring']['modified']:

print(f" ⚠️ Safety: {cycle\_info['safety\_monitoring']['reason']}")

if done or info.get('target\_reached', False):

self.mission\_complete = info.get('target\_reached', False)

break

self.total\_reward += episode\_reward

# Generate mission report

report = self.\_generate\_mission\_report(step, episode\_reward)

if verbose:

print(f"\n{'='\*60}")

print("MISSION REPORT")

print(f"{'='\*60}")

print(f"Status: {'✅ SUCCESS' if self.mission\_complete else '❌ INCOMPLETE'}")

print(f"Steps: {step}")

print(f"Episode Reward: {episode\_reward:.2f}")

print(f"Final Battery: {observation['battery\_level']:.1%}")

print(f"Safety Violations: {report['safety\_stats']['violations']}")

print(f"Verification Success Rate: {report['safety\_stats']['success\_rate']:.1%}")

print(f"{'='\*60}\n")

return report

def \_generate\_mission\_report(self, steps: int, reward: float) -> Dict:

"""Generate comprehensive mission report"""

thought\_log = self.cognitive\_agent.get\_thought\_log()

return {

'agent\_id': self.cognitive\_agent.id,

'episode': self.episode\_count,

'steps': steps,

'reward': reward,

'total\_reward': self.total\_reward,

'mission\_complete': self.mission\_complete,

'final\_position': self.simulator.robot\_state.position,

'final\_battery': self.simulator.robot\_state.battery\_level,

'beliefs': {

'de\_se': list(thought\_log['de\_se\_beliefs']),

'de\_dicto': list(thought\_log['de\_dicto\_beliefs']),

'total': len(thought\_log['de\_se\_beliefs']) + len(thought\_log['de\_dicto\_beliefs'])

},

'safety\_stats': thought\_log['safety\_stats'],

'attitudes': thought\_log['attitudes']

}

# ============================================================================

# REST API (Using FastAPI-like structure, but pure Python)

# ============================================================================

class SimpleHTTPServer:

"""

Simple HTTP server for cognitive architecture API

(In production, use FastAPI)

"""

def \_\_init\_\_(self, agent):

self.agent = agent

self.routes = {

'/status': self.get\_status,

'/thought\_log': self.get\_thought\_log,

'/beliefs': self.get\_beliefs,

'/safety\_stats': self.get\_safety\_stats,

'/action': self.post\_action

}

def get\_status(self) -> Dict:

"""GET /status - Agent status"""

return {

'agent\_id': self.agent.id,

'time': self.agent.time,

'position': self.agent.state.position,

'active': True

}

def get\_thought\_log(self) -> Dict:

"""GET /thought\_log - Complete thought log"""

return self.agent.get\_thought\_log()

def get\_beliefs(self) -> Dict:

"""GET /beliefs - Current beliefs"""

return {

'de\_se\_beliefs': list(self.agent.beliefs.de\_se\_beliefs),

'de\_dicto\_beliefs': list(self.agent.beliefs.de\_dicto\_beliefs),

'total': len(self.agent.beliefs.get\_all\_beliefs())

}

def get\_safety\_stats(self) -> Dict:

"""GET /safety\_stats - Safety statistics"""

return self.agent.safety\_monitor.get\_statistics()

def post\_action(self, request\_data: Dict) -> Dict:

"""POST /action - Execute action"""

from philosophical\_agi.core.types import Action

observation = request\_data.get('observation', {})

action, cycle\_info = self.agent.cognitive\_cycle(observation)

return {

'action': {

'type': action.action\_type,

'target\_position': action.target\_position,

'velocity': action.velocity

},

'cycle\_info': {

'time': cycle\_info['time'],

'beliefs\_updated': cycle\_info['beliefs\_updated'],

'safety\_verified': not cycle\_info['safety\_monitoring']['modified']

}

}

def handle\_request(self, path: str, method: str = 'GET', data: Optional[Dict] = None):

"""Handle HTTP request (simplified)"""

if path in self.routes:

handler = self.routes[path]

if method == 'POST' and data:

return handler(data)

else:

return handler()

else:

return {'error': 'Route not found', 'available\_routes': list(self.routes.keys())}

# ============================================================================

# EXAMPLE USAGE

# ============================================================================

def example\_robotics\_mission():

"""Example: Complete robotics mission with safety verification"""

print("\n" + "="\*70)

print("ROBOTICS MISSION DEMONSTRATION")

print("="\*70)

# Create simulator

sim = SimpleRoboticsSimulator(arena\_size=10.0)

# Add environment objects

sim.add\_obstacle((3.0, 3.0), radius=0.8)

sim.add\_obstacle((7.0, 7.0), radius=0.6)

sim.add\_human((5.0, 8.0))

sim.add\_target((8.5, 2.0), reward=10.0)

print("Environment:")

print(f" - Obstacles: {len(sim.obstacles)}")

print(f" - Humans: {len(sim.humans)}")

print(f" - Targets: {len(sim.targets)}")

# Create robot agent

robot = RobotAgent("MissionBot", sim)

# Run mission

report = robot.run\_mission(max\_steps=100, verbose=True)

return report

def example\_api\_usage():

"""Example: REST API usage"""

print("\n" + "="\*70)

print("REST API DEMONSTRATION")

print("="\*70 + "\n")

from philosophical\_agi.core.agent import CognitiveAgent

agent = CognitiveAgent("APIAgent")

api = SimpleHTTPServer(agent)

# Example requests

print("GET /status")

response = api.handle\_request('/status')

print(json.dumps(response, indent=2))

print("\n" + "-"\*70 + "\n")

print("POST /action")

request\_data = {

'observation': {

'agent\_position': (5.0, 5.0, 0.0),

'target': (8.0, 8.0, 0.0),

'threat\_level': 0.2

}

}

response = api.handle\_request('/action', method='POST', data=request\_data)

print(json.dumps(response, indent=2))

print("\n" + "-"\*70 + "\n")

print("GET /beliefs")

response = api.handle\_request('/beliefs')

print(json.dumps(response, indent=2))

print("\n" + "-"\*70 + "\n")

print("GET /safety\_stats")

response = api.handle\_request('/safety\_stats')

print(json.dumps(response, indent=2))

if \_\_name\_\_ == "\_\_main\_\_":

import sys

import os

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))

# Run demonstrations

example\_robotics\_mission()

example\_api\_usage()