User

**You**

Can you interpret this code for me? The main function is run\_simulation import numpy as np from matplotlib.patches import Polygon, Circle import matplotlib.pyplot as plt from palettable.colorbrewer.qualitative import Set1\_9 import copy import pandas as pd import itertools def softmax(x): """Compute softmax values for each sets of scores in x.""" if x.ndim == 1: x = x.reshape(1, -1) # Convert 1D array to 2D array e\_x = np.exp(x) return e\_x / np.sum(e\_x, axis=1, keepdims=True) def log\_stable(x): """Compute log values for each sets of scores in x.""" return np.log(x + np.exp(-16)) def simulate\_observation(true\_position, observation\_error\_std=2.0, sim\_type='A', observed\_agent= None): """Simulate noisy observation of another agent's position.""" observation = {'position': np.zeros((2,))-1e6, 'heading': float(-1e6), 'type': sim\_type, 'observed\_agent': observed\_agent} if sim\_type == 'self' or 'sim': # Assume no noise for self-observation or simulated observation observation['position'] = true\_position[:2] observation['heading'] = true\_position[2] else: observation['position'] = true\_position[:2] + np.random.normal(0, observation\_error\_std, true\_position[:2].shape) observation['heading'] = true\_position[2] + np.random.normal(0, observation\_error\_std\*0.1) return observation def calculate\_kl\_divergence(p, q): """Calculate KL divergence between two probability distributions.""" return np.sum(p \* log\_stable(p / (q + np.exp(-16))))/16 #Divide by 16 to normalize the KL divergence def calculate\_shannon\_entropy(p): """Calculate Shannon entropy of a probability distribution.""" return -np.sum(p \* log\_stable(p)) def max\_entropy\_prior(distribution): """Calculate the max entropy prior for a given distribution.""" entropy = np.sum([calculate\_shannon\_entropy(d) for d in distribution]) #Use entropies as weights for each element of the distribution return entropy def get\_likelihood(robot\_id,observations,goals,system\_dict,max\_distance\_measure=1.0,use\_ep=False): """Calculate the likelihood of a goal being the target based on agent positions.""" robot\_pose = observations[robot\_id]['position'] if not use\_ep: likelihood = np.zeros(goals.shape[0]).flatten() for observation in observations: tmp\_obs\_type = observation['type'] likelihood += salience(robot\_pose,observation, tmp\_obs\_type, goals, max\_distance\_measure).flatten() else: likelihood = np.tile(np.zeros(goals.shape[0]).flatten(),(len(observations),1)) robot\_type = system\_dict['agent\_types'][robot\_id] for predict\_id, predict\_type in enumerate(system\_dict['agent\_types']): for obs\_id,observation in enumerate(observations): tmp\_obs\_type = ('self' if predict\_id == obs\_id else predict\_type) tmp\_obs\_type = ('none' if (robot\_type in ['B'] and predict\_type in ['A']) else tmp\_obs\_type) likelihood[predict\_id] += salience(observations[predict\_id]['position'], observation, tmp\_obs\_type,goals,max\_distance\_measure).flatten() return softmax(likelihood) def salience(robot\_pose, observation, predict\_type, goals, max\_distance\_measure=1.0): """Calculate the salience of each goal based on depth and azimuth measurements from the robot.""" goal\_vectors = goals - robot\_pose[:2] goal\_azimuths = np.arctan2(goal\_vectors[:, 1], goal\_vectors[:, 0]) saliences = np.zeros(len(goals)) if predict\_type in ['A','self']: # For depth, compute how close each goal is to the observed depth depth\_differences = np.linalg.norm(observation['position'] - goals, axis=1) depth\_salience = 2\*np.exp(-depth\_differences / max\_distance\_measure) # exponential decay based on depth difference saliences += depth\_salience if predict\_type in ['B']: # For azimuth, compute alignment of each goal with the observed azimuth observed\_azimuth = np.arctan2(observation['position'][1]-robot\_pose[1], observation['position'][0]-robot\_pose[0]) relative\_azimuths = np.abs((goal\_azimuths - observed\_azimuth + np.pi) % (2 \* np.pi) - np.pi) azimuth\_salience = 1./8 \* np.exp(- relative\_azimuths / np.pi) # normalize and invert to make smaller angles more salient saliences += azimuth\_salience # Compute if observed robot is heading towards the goal # heading\_to\_goal = (np.arctan2(goals[:, 1] - observation['position'][1], goals[:, 0] - observation['position'][0]) - observation['heading'] + np.pi) % (2 \* np.pi) - np.pi # heading\_salience = 1./8 \* np.exp(- np.abs(heading\_to\_goal) / np.pi) # saliences += heading\_salience return saliences def predict\_agent\_position(agent\_position, velocity, heading): """Predict agent's next position based on chosen velocity and heading.""" agent\_prediction = np.copy(agent\_position) agent\_prediction[0] += velocity \* np.cos(heading) agent\_prediction[1] += velocity \* np.sin(heading) agent\_prediction[2] = heading return agent\_prediction def evaluate\_state(prior, posterior): """Evaluate the desirability of a state based on the posterior probability of goals.""" # Here we use negative entropy as a simple scoring mechanism: lower entropy means higher certainty/quality of the state. entropy = calculate\_shannon\_entropy(posterior) kl\_divergence = calculate\_kl\_divergence(prior,posterior) return entropy + kl\_divergence # This function can be changed -- recommend using a neural network to predict the likelihood of a goal being the target # TODO: make more general for many different rollouts def extended\_greedy\_rollout(args, system\_dict, num\_rollout\_steps=5): """Perform a greedy rollout considering the actions of all agents.""" agent\_id = args['agent\_id'] current\_positions = np.copy(args['agent\_positions']) goals = args['goals'] all\_actions = list(itertools.product(args['velocity\_options'], args['heading\_options'])) best\_action = None best\_score = float('-inf') # Consider all possible actions for the focal agent for action in all\_actions: total\_score = 0 for step in range(num\_rollout\_steps): simulated\_positions = np.copy(current\_positions) # Assume each agent, including the focal agent, might take any action for other\_agent\_id in range(len(current\_positions)): if other\_agent\_id == agent\_id: velocity, heading = action else: # Assume a random or heuristic-based action for other agents velocity, heading = np.random.choice(all\_actions) predicted\_position = predict\_agent\_position(simulated\_positions[other\_agent\_id], velocity, heading) simulated\_positions[other\_agent\_id] = predicted\_position # Evaluate the system state after all agents have moved predicted\_observations = [simulate\_observation(pos, args['observation\_error\_std'], args['agent\_types'][idx], idx) for idx, pos in enumerate(simulated\_positions)] posteriors = [get\_likelihood(simulated\_positions[i], predicted\_observations, goals, args['max\_distance\_measure'], system\_dict, args['use\_ep'], predict\_self=True) for i in range(len(simulated\_positions))] system\_score = sum(evaluate\_state(posterior, goals) for posterior in posteriors) total\_score += system\_score # Update best action based on the highest cumulative score over the rollout steps if total\_score > best\_score: best\_score = total\_score best\_action = action return best\_action def parse\_observations(observations): """Parse observations to get agent positions.""" agent\_positions = np.zeros((len(observations), len(observations[0]['position']) + 1)) for idx, observation in enumerate(copy.deepcopy(observations)): agent\_positions[idx] = np.concatenate((observation['position'], [observation['heading']]),axis=0) return agent\_positions.copy() def make\_decision(args,system\_dict, use\_ep=True): """Agent decision-making based on active inference to encourage convergence on a shared goal.""" #Parse arguments agent\_id = args['agent\_id'] agent\_type = args['agent\_type'] agent\_positions = args['agent\_positions'] prior = args['prior'] goals = args['goals'] velocity\_options = args['velocity\_options'] heading\_options = args['heading\_options'] max\_distance\_measure = args['max\_distance\_measure'] observation\_error\_std = args['observation\_error\_std'] # Initialize variables for decision-making best\_action = None best\_score = np.inf goal\_scores = [] observations = [] # Start decision loops for each possible action for idx in range(agent\_positions.shape[0]): if idx == agent\_id: observations.append(simulate\_observation(agent\_positions[idx], 0, 'self',idx)) else: observations.append(simulate\_observation(agent\_positions[idx], observation\_error\_std, agent\_type,idx)) observed\_positions = parse\_observations(observations) # Calculate the likelihood of each goal being the target based on agent measurements -> May be able to turn into NN for velocity in velocity\_options: for heading in heading\_options: predicted\_position = predict\_agent\_position(observed\_positions[agent\_id], velocity, heading) new\_agent\_positions = np.copy(agent\_positions) new\_agent\_positions[agent\_id] = predicted\_position predicted\_observation = copy.deepcopy(observations) predicted\_observation[agent\_id]['position'] = np.copy(predicted\_position[:2]) predicted\_observation[agent\_id]['heading'] = predicted\_position[2] #Predict how what you would do would differ from the prior (what the system is doing - complexity) posterior = get\_likelihood(agent\_id, predicted\_observation, goals, system\_dict, max\_distance\_measure, use\_ep) kl\_divergence = calculate\_kl\_divergence(prior, posterior) kl\_divergence = 0.0 entropy = max\_entropy\_prior(posterior) # Estimate how both agents are aligned with reaching the current goal goal\_scores.append((entropy + kl\_divergence, velocity, heading, entropy, kl\_divergence, posterior)) # Choose the action (for the current goal) that minimizes the combined distance best\_action\_for\_goal = min(goal\_scores, key=lambda x: x[0]) # Update best action if this goal is more attainable than previous best if best\_action\_for\_goal[0] < best\_score: best\_score = best\_action\_for\_goal[0] best\_action = best\_action\_for\_goal[1], best\_action\_for\_goal[2] return best\_action, observations, goal\_scores def parse\_args\_by\_agent(args): """Parse arguments by agent.""" agent\_vars = [] for agent\_id in range(len(args['agent\_positions'])): agent\_dict = {} agent\_dict['agent\_id'] = agent\_id agent\_dict['agent\_type'] = args['agent\_types'][agent\_id] agent\_dict['agent\_positions'] = args['agent\_positions'] agent\_dict['goals'] = args['goals'] agent\_dict['velocity\_options'] = args['velocity\_options'] agent\_dict['heading\_options'] = args['heading\_options'] agent\_dict['max\_distance\_measure'] = args['max\_distance\_measure'] agent\_dict['observation\_error\_std'] = args['observation\_error\_std'] agent\_dict['prior'] = args['prior'] if args['use\_ep']: agent\_dict['prior'] = np.tile(args['prior'], (len(args['agent\_positions']), 1)) agent\_vars.append(agent\_dict) return agent\_vars def run\_simulation(args, max\_iterations=100): """Run the simulation until both agents converge to the same goal or max iterations reached.""" return\_args = {} agent\_vars = parse\_args\_by\_agent(args) num\_agents = len(args['agent\_positions']) current\_positions = np.copy(args['agent\_positions']) true\_positions = np.copy(current\_positions) goals = args['goals'] return\_args['positions'] = [np.copy(current\_positions)] free\_energy\_scores = [] use\_ep = args['use\_ep'] converged\_count = 0 for iteration in range(max\_iterations): # Make decisions for all agents # print(f"Iteration {iteration}") if iteration == max\_iterations - 1: print("Max iterations reached.") decisions = [] observations = [] if iteration > 8: s = 1 for idx in range(num\_agents): decision, observation, free\_energy\_score = make\_decision(agent\_vars[idx],args, use\_ep) # Make decision for agent decisions.append(decision) # Save decisions for all agents observations.append(observation) # Save observations for all agents free\_energy\_scores.append((free\_energy\_score, idx, iteration)) # Save free energy scores for all agents # Update agent positions based on their decisions for agent\_id, (velocity, heading) in enumerate(decisions): dx = velocity \* np.cos(heading) dy = velocity \* np.sin(heading) true\_positions[agent\_id] += np.array([dx, dy, 0]) true\_positions[agent\_id][2] = heading # decided\_velocities = [decision[0] for decision in decisions] for agent\_id in range(num\_agents): agent\_vars[agent\_id]['agent\_positions'] = np.copy(true\_positions) observations[agent\_id][agent\_id]['position'] = true\_positions[agent\_id][:2] observations[agent\_id][agent\_id]['heading'] = true\_positions[agent\_id][2] agent\_vars[agent\_id]['prior'] = get\_likelihood(agent\_id,observations[agent\_id], goals, args, args['max\_distance\_measure'], use\_ep) # Check if agents have converged to the same goal current\_positions = np.copy(true\_positions) distances\_to\_goals = [np.linalg.norm(goals - pos[:2], axis=1) for pos in current\_positions] distances\_to\_selected\_goal = [np.min(distances) for distances in distances\_to\_goals] selected\_goal = [np.argmin(distances) for distances in distances\_to\_goals] all\_same\_goal = [selected\_goal[0] == which\_goal for which\_goal in selected\_goal] # Save current positions for plotting return\_args['positions'].append(np.copy(current\_positions)) if (np.array(distances\_to\_selected\_goal)<args['max\_distance\_measure']/10).all() and all(all\_same\_goal): converged\_count += 1 if converged\_count >= 5: print(f"Agents have converged to Goal {selected\_goal[0]} after {iteration + 1} iterations.") return current\_positions, selected\_goal[0], iteration, return\_args, [agent\_vars[i]['prior'] for i in range(num\_agents)], free\_energy\_scores print("Agents did not converge to the same goal within the maximum iterations.") return current\_positions, None, iteration, return\_args, [agent\_vars[i]['prior'] for i in range(num\_agents)], free\_energy\_scores class PlotSim: """Class to plot the simulation of agents moving towards a goal.""" def \_\_init\_\_(self, num\_agents, goals, env\_size=15, padding = 5): # Initialize plotting objects self.fig, self.ax = plt.subplots() plt.xlim(-5, env\_size+padding) plt.ylim(-5, env\_size+padding) self.cmap = Set1\_9.mpl\_colors self.agent\_paths = [self.ax.plot([], [], 'o-', markersize=3, linewidth=1, alpha=0.5, color=self.cmap[i])[0] for i in range(num\_agents)] self.goal\_plots = [self.ax.plot(goal[0], goal[1], 'x', markersize=10, color='purple')[0] for goal in goals] # Plot goals self.all\_rosbots = [] # Dictionary to store all rosbot elements def init(self): """Initialize the background of the plot.""" agent\_id = 0 for agent\_path in self.agent\_paths: agent\_path.set\_data([], []) self.all\_rosbots.append(self.init\_robot(self.cmap[agent\_id])) agent\_id += 1 return self.agent\_paths def update(self, frame, args): """Update the plot data for each agent.""" agent\_positions = args['positions'][frame] # Update plot data for each agent for agent\_id, agent\_path in enumerate(self.agent\_paths): xnew, ynew, heading = agent\_positions[agent\_id] self.update\_robot(self.all\_rosbots[agent\_id], (xnew,ynew), heading, color=self.cmap[agent\_id]) xdata, ydata = agent\_path.get\_data() xdata = np.append(xdata, xnew) ydata = np.append(ydata, ynew) agent\_path.set\_data(xdata, ydata) #Update title self.ax.set\_title(f'Iteration {frame}') return self.agent\_paths def init\_robot(self, color='r'): """Initialize parts of the Robot with placeholders for their geometric data""" elements = { 'red\_box': self.ax.fill([], [], color=color, alpha=0.3)[0], 'left\_tires': self.ax.fill([], [], color='k')[0], 'right\_tires': self.ax.fill([], [], color='k')[0], 'light\_bar': self.ax.fill([], [], color='k', alpha=0.5)[0], } return elements def update\_robot(self, elements, position, heading\_ang, color, scale=0.3): """Update the position and orientation of the Robot parts""" rosbot\_scale = scale # Define the parts with their unscaled coordinates parts = { 'red\_box': np.array([[-2, -2, 2, 2], [1.6, -1.6, -1.6, 1.6]]), 'left\_tires': np.array([ [0.35, 0.35, 2, 2, 0.35, -0.35, -2, -2, -0.35, -0.35], [1.6, 1.9, 1.9, 1.6, 1.6, 1.6, 1.6, 1.9, 1.9, 1.6] ]), 'right\_tires': np.array([ [0.35, 0.35, 2, 2, 0.35, -0.35, -2, -2, -0.35, -0.35], [-1.6, -1.9, -1.9, -1.6, -1.6, -1.6, -1.6, -1.9, -1.9, -1.6] ]), 'light\_bar': np.array([[-0.6, -0.6, -1.25, -1.25], [1.2, -1.2, -1.2, 1.2]]), } # Apply scaling, rotation, and translation rotation = np.array([ [np.cos(heading\_ang), -np.sin(heading\_ang)], [np.sin(heading\_ang), np.cos(heading\_ang)] ]) for name, part in parts.items(): scaled\_part = rosbot\_scale \* part transformed = rotation @ scaled\_part + np.array(position)[:, None] elements[name].set\_xy(transformed.T) if name in ['red\_box']: elements[name].set\_facecolor(color) # Apply color only to these parts def parse\_free\_energy\_scores(free\_energy\_scores, num\_iterations): """Parse the free energy scores for each agent over time.""" res\_list = []; ending\_energy = [] for entry in free\_energy\_scores: avg\_scores = np.min([score[0] for score in entry[0]]) agent = entry[1] iteration = entry[2] res\_list.append([avg\_scores, agent, iteration]) if iteration == num\_iterations: ending\_energy.append(avg\_scores) return res\_list, ending\_energy def plot\_energy(data, num\_agents): """Plot the free energy scores for each agent over time.""" cmap = Set1\_9.mpl\_colors fig, ax = plt.subplots() df = pd.DataFrame(data, columns=['Avg Free Energy', 'Agent', 'Iteration']) # Plot the free energy scores for each agent for i in range(num\_agents): agent\_data = df[df['Agent'] == i] ax.plot(agent\_data['Iteration'], agent\_data['Avg Free Energy'], label=f'Agent {i}', color=cmap[i]) ax.set\_xlabel('Iteration') ax.set\_ylabel('Free Energy') ax.set\_title('Free Energy Scores for Each Agent') ax.legend() # Show plot plt.show()

ChatGPT

**ChatGPT**

The provided Python code simulates a multi-agent system where agents interact within a defined space and make decisions based on observations of each other and predefined goals. The main function **run\_simulation** orchestrates the simulation based on input parameters and decision-making functions. Here's a breakdown of the key components:

**Functions**

1. **Utility Functions**:
   * **softmax**, **log\_stable**: These functions handle numerical calculations for probability distributions and safe logarithm calculations to avoid numerical issues.
2. **Agent Observation Simulation**:
   * **simulate\_observation**: Simulates noisy or perfect observations of agent positions and headings based on their true state and a defined error standard deviation.
3. **Probability Calculations**:
   * **calculate\_kl\_divergence**, **calculate\_shannon\_entropy**, **max\_entropy\_prior**: Functions to calculate Kullback-Leibler divergence, Shannon entropy, and a maximum entropy prior from probability distributions, useful for decision-making and evaluating state desirability.
4. **Decision-Making Mechanics**:
   * **get\_likelihood**: Calculates the likelihood of each goal being the target based on observations, using both salience and agent-specific information.
   * **salience**: Determines how "important" or relevant each goal is based on spatial and observational parameters like azimuth and depth.
   * **predict\_agent\_position**: Predicts the next position of an agent based on its current velocity and heading.
   * **evaluate\_state**: Scores the system state by combining entropy and KL divergence, indicating the quality of a decision.
   * **extended\_greedy\_rollout**, **make\_decision**: These functions are at the core of the agent's decision-making process, allowing them to select actions based on predicted outcomes and the maximization of certain utility functions.
   * **parse\_observations**, **parse\_args\_by\_agent**: Helper functions to structure observations and arguments for processing.
5. **Simulation Control**:
   * **run\_simulation**: Controls the simulation loop, calling decision-making functions iteratively and updating agent states until convergence criteria are met or maximum iterations are reached.

**Classes**

* **PlotSim**: A class dedicated to plotting the simulation process, showing agent movements, interactions, and goal locations. It includes methods to initialize plots, update them based on agent actions, and handle detailed representations of agent positions and orientations.

**Key Concepts**

* **Agent Interaction**: Agents observe each other and make decisions based on both their own observations and those of other agents.
* **Decision Making**: Agents choose actions based on a mixture of greedy approaches (looking for immediate benefits) and predictive approaches (considering future states of the environment).
* **Convergence**: The simulation aims to reach a state where all agents agree on a common goal, indicated by their proximity to that goal and mutual agreement in their decision-making.

**Main Function: run\_simulation**

This function initializes the simulation with user-defined parameters for agents, their positions, and goals. It iteratively updates the state of the system by simulating agent decisions and movements, aiming for convergence on a common goal. The simulation can visualize the process and provide diagnostic data on the internal state of the agents, including their decision scores and position updates.

Overall, this script is an advanced example of a multi-agent simulation where decision-making is based on probabilistic models and interaction dynamics.

User

**You**

Would you say this code implements active inference mechanics?

ChatGPT

**ChatGPT**

Yes, the code does incorporate elements of active inference, a theoretical framework used in cognitive science and neuroscience for understanding how agents perceive and act in their environment based on probabilistic reasoning and minimizing surprise or prediction error. Here are a few key aspects of active inference reflected in the code:

1. **Goal-Directed Behavior**: The agents in the simulation have specific goals, and their actions are driven by predictions about how their actions will help achieve these goals. The use of a goal likelihood function (**get\_likelihood**) aligns with active inference, as it involves computing the probability of each goal being the intended target based on current observations and agent states.
2. **Probability Distributions and Updates**: Agents maintain and update beliefs about the state of the world (e.g., positions of other agents, headings, goals). These beliefs are represented as probability distributions, which are updated based on new observations (**simulate\_observation**) and predictions (**predict\_agent\_position**). The active inference framework similarly emphasizes the continuous updating of beliefs based on sensory inputs and internal models.
3. **Entropy and KL Divergence**: The simulation evaluates decisions using entropy (from the **calculate\_shannon\_entropy** function) and KL divergence (**calculate\_kl\_divergence**), which measure the uncertainty in the agents' beliefs and the divergence between their prior and posterior beliefs, respectively. Minimizing these metrics (especially KL divergence, which quantifies the "surprise" based on new data relative to prior beliefs) is central to active inference, as it suggests that agents strive to minimize surprise in their sensory encounters.
4. **Exploration and Decision Making**: The decision-making process (**make\_decision** and **extended\_greedy\_rollout**) includes exploring different possible actions and their outcomes. This decision-making is based on predicting the effects of actions on belief states and choosing actions that are expected to result in more precise (lower entropy) and accurate (minimizing KL divergence) belief updates.
5. **Free Energy Principle**: Although not explicitly named, the concepts of entropy and KL divergence are closely related to the Free Energy Principle, which posits that biological agents minimize a free energy functional of their sensory states. This principle is a cornerstone of active inference, where agents are seen as minimizing a bound on the surprise associated with sensory inputs relative to their model of the world.

Overall, the implementation uses a complex system of observations, predictions, and probability updates to guide agent behavior in a manner consistent with the principles of active inference, focusing on minimizing uncertainty and prediction error as they navigate toward their goals.