ChatGPT

# MycoBeaver Simulator Design Plan

## 1. Grid Environment & Hydrology Dynamics

The MycoBeaver world is a 2D grid (e.g. an `NxN` grid) with various fields stored per cell. Key state layers include: **elevation** (static), **water depth** `h_i` (m of water) [1], **vegetation biomass** `v_i` (available food/wood) [2], **soil moisture** `m_i` (0–1) [3], **dam permeability** `d_i` (range 0–1, where 0 = fully blocked dam, 1 = no dam) [4], and optional **lodge indicator** (beaver home) [5]. These are maintained in the environment's state, for example as NumPy 2D arrays (`elevation[y,x]`, `water[y,x]`, etc.). The environment's **Gym interface** returns observations composed of these fields (or local excerpts) to agents.

**Hydrological flow:** Each time-step, water flow is computed based on differences in water surface height between neighboring cells. Define water surface height `H_i = elevation_i + h_i`. If cell *i* has higher water level than neighbor *j*, water flows from *i* to *j* with flux `q_ij = g_ij * (H_i - H_j)` [6] [7]. Here `g_ij` is a conductance factor modulated by dams: we take `g_ij = g0 * f(d_i, d_j)` [8], where `g0` is a base conductance and `f(d_i,d_j)` is a decreasing function of dam presence (e.g. `f = 0.5 * (d_i + d_j)` means a low `d` (strong dam) reduces flow) [9]. The environment integrates these flows to update water depth: each cell's new water depth `h_i(t+1)` = old depth + Δt * (rainfall + boundary inflow + **inflows** from neighbors – **outflows** – **losses**) [10] [11]. Loss terms include evaporation and seepage (modeled as `ℓ(h_i) = α_evap * h_i + α_seep * h_i`, a linear decay) [12]. Rainfall `r_i` and any external inflows `b_i` can be added to cells as exogenous inputs each step [13] [14]. This hydrology model ensures water accumulation behind dams (low `d` yields low conductance and thus water backs up) and can simulate floods or droughts. The **Environment** class will implement a method like `update_water()` performing these calculations each step, using vectorized operations for efficiency.

**Vegetation & soil:** Vegetation biomass `v_i` can regrow each step and be consumed by beavers. Soil moisture `m_i` increases with water presence (e.g. proportional to water depth or flooding) and decays over time. We implement a simple soil-water coupling: when water is present in a cell or neighboring cell, soil moisture trends toward 1; otherwise it dries toward 0. Vegetation regrowth rate may depend on soil moisture (e.g. higher moisture = faster regrowth). These dynamics are updated in an `update_vegetation_and_soil()` step: for each cell, increase `v_i` by a regrowth function (capped at some max) and update `m_i` by diffusing current moisture toward an equilibrium based on water.

**Dams and lodges:** A **dam** in the grid is represented by a cell (or edge) with reduced permeability `d_i < 1`. When beavers build a dam at a cell, `d_i` is decreased (tending toward 0 as dam fully blocks water) [4]. The environment could treat dams as occupying a line of cells (e.g. across a stream); but for implementation simplicity we can allow any cell to have a dam strength value that influences flow to its neighbors. **Lodges** are special cells where beavers live (analogous to "hive" or colony centers). A lodge cell can grant safety or warmth (reducing beaver wetness penalty when they occupy it), and it serves as a hub for communication (as in bee waggle dances). Lodges are placed via scenario configuration or built by agents; the environment tracks lodge positions (e.g. a boolean grid or list of lodge cells). Lodges do not directly affect hydrology but are important for agent behavior.

**Environment class design:** We encapsulate all these features in a class `MycoBeaverEnv(gym.Env)`. It contains: - **Grid state arrays**: `elevation`, `water_depth`, `vegetation`, `soil_moisture`, `dam_permeability`, `lodge_map`, etc. - **Methods for dynamics**: `flow_water()` (compute flows and update water depths), `evaporate_seep()` (apply losses), `grow_vegetation()`, `update_soil()`. - **Exogenous input management**: e.g. apply rainfall patterns or inflows each step (could be random or scenario-defined). - **Reset**: Initialize all fields (e.g. set initial water levels, vegetation distribution, etc., perhaps from a Scenario). - **Step**: Integrate one time-step (calls internal update methods as well as processes agent actions, described later). - It also holds subcomponents like the **pheromone field**, **physarum network**, and list of **agents** (or their states), as described in sections below.

The environment will produce observations for agents. Likely each agent gets a local observation (e.g. an `r x r` grid around it for local terrain and resources, plus some global signals). We implement a function `get_observation(agent)` that gathers the necessary data from the environment for a given agent (this can be used in the Gym `step` to return all agents' observations). The environment's `info` or a separate metrics module will also track global metrics like total vegetation, flood occurrence, etc., for analysis and for Overmind input (see Overmind section).

## 2. Beaver Agents: State, Decisions, and Actions

We simulate a colony of `K` beaver agents. Each **BeaverAgent** is an object with internal state variables capturing its physical condition and behavioral disposition [15] [16]. Important state fields include: - **Energy** (`e_k`): a non-negative value representing stamina. It decreases with exertion (moving, building) and partially restores with rest or feeding. - **Satiety (hunger)** (`s_k` in [0,1]): fraction of full stomach [17]. 1 means well-fed, 0 means starving. Satiety decreases over time; if it drops too low, the agent's energy use rises or it may start losing health. - **Wetness/Cold** (`w_k` in [0,1]): exposure level [18]. If a beaver spends time in water or rain, this increases, imposing penalties (e.g. faster energy drain or risk of hypothermia). Sheltering in a lodge or dry area reduces `w_k`. - **Role** (`r_k`): the behavioral role or caste, e.g. `scout`, `worker`, `guardian`, etc. [16]. Roles determine default tendencies (scouts explore, workers gather/build, guardians defend). Agents can change roles based on colony needs (e.g. an agent might switch to scout if new projects are needed). - **Task response thresholds** (`θ_{k,m}`): each agent has a threshold for responding to a stimulus or task *m* (for m=1…M task types) [19]. For example, one threshold might govern when it starts building a dam (e.g. respond if water flow or Overmind signal exceeds θ), another for when to forage for food, etc. These thresholds can be innate or learned, and the Overmind may globally adjust them (e.g. lower the threshold for dam-building if a flood risk is detected). - **Current project assignment** (`p_{k}^{proj}`): either the ID of a project the agent is working on, or ∅ if it's not currently committed [20]. This indicates if the beaver is part of a cooperative building task (like a dam construction project).

Each agent perceives its **local environment** and possibly colony signals. We implement an agent method `perceive(obs)` that takes in its observation (from the environment, e.g. nearby cell data and any pheromone or "wisdom" signals present) and updates its internal state or neural network input accordingly. Agents have a decision policy to choose an action each step given their state and observations: - **Greedy policy**: This is a reactive policy that selects actions to immediately satisfy the agent's needs (e.g. if hungry, go forage; if cold, seek lodge; if nothing pressing, possibly explore or follow a pheromone trail to resources). It doesn't explicitly plan for long-term colony benefits. - **Contemplative policy**: A more strategic policy that weighs colony-level goals (possibly influenced by Overmind's wisdom signal or anticipated future states). For example, a contemplative agent might delay eating to help build a dam if it predicts that will improve habitat (and thus future food) [21] [22]. Technically, this could be implemented via a different neural policy network that includes global signals or a value function considering long-term rewards.

We will implement these policies as PyTorch neural networks (e.g. using `torch.nn.Module`). For example, `AgentPolicyNet` could be a network that takes as input the agent's local observation (nearby water, food, etc.), its internal state (satiety, energy, role), and any global signals (like Overmind's current wisdom or project recruitment info), and outputs an action distribution. We can have two variants or modes of this network to represent "greedy" vs "contemplative" (or a single network trained differently depending on reward structure). The architecture might be a small CNN or MLP processing the local grid and a concatenated vector of agent state. The **action space** for each beaver agent includes: - **Move** in one of the 4 cardinal directions (or 8 directions if diagonal moves allowed), or stay put. Movement costs energy and time. - **Forage/Eat**: Harvest vegetation at the current cell to gain food (increasing satiety) and collect wood. Mechanically, this reduces `v_i` in the cell and increases the agent's satiety. If vegetation is ample, multiple forage actions might yield diminishing returns until it regrows. - **Carry resources**: If implemented, an agent might pick up a piece of wood (from foraging) to carry to a build site. We could simulate a simple carrying capacity (like one wood unit at a time). - **Build Dam**: If at a location designated for dam construction (project site) and carrying materials, the agent can expend those materials to decrease the cell's `d` value (strengthening the dam) [4]. This also costs energy. The environment will apply the effect by reducing `dam_permeability` in that cell (or multiple adjacent cells if the dam spans them). - **Build Lodge**: Possibly, if conditions allow, an agent could construct a new lodge on a suitable cell (e.g. along a pond). This would be a complex action requiring wood and time. For now, lodge building can be considered a special project type handled similarly to dams. - **Communication (social signals)**: A scout at a lodge might perform a "waggle dance" (or equivalent communication) to advertise a project site to idle workers. We don't need a literal dance animation, but this can be an action like `advertise(project_id)` which triggers an update to the recruitment signal for that project in the environment (detailed in the next section). - **Rest/Idle**: If the agent chooses no active task (or explicitly rests), it can recover energy faster or reduce wetness (especially if resting in lodge).

Each **BeaverAgent** class instance holds its own PyTorch policy (or references a shared policy model if using parameter sharing). The class provides: - `decide_action(obs)`: feed observation to the policy network to sample or argmax an action. - `reset(position)`: place the agent at a start position with full or default internal state. - `update_internal_state(dt)`: update energy (decrease based on activity, increase if resting), satiety (decrease each step, or set to 0 if the agent has starved), wetness (maybe decrease gradually if not in water or if in lodge). - `is_alive()`: check if conditions like energy or hunger indicate death; if so, the agent is removed from active list. - Learning hooks: e.g. `learn(reward, next_obs)` to update the policy (for on-policy RL this might store trajectory data; for off-policy it might do a gradient step each time). We integrate PyTorch optimizers for agent policies here or in a separate trainer.

**Roles and differentiation:** We may implement role-specific behavior either via input features or separate networks. A simple approach is to embed the role (scout/worker) as part of the observation input (e.g. one-hot role input) so that one policy network can learn different behaviors. Alternatively, we could maintain separate policy networks for scouts vs workers if their tasks are very distinct. Initially, roles can be assigned (e.g. a fraction of agents are scouts that explore outward, others workers that mostly stay near the lodge until recruited) and agents can switch roles by some logic (like if a scout finds a project, it might become a worker on that project after advertising). The **response thresholds** `θ_{k,m}` tie into this: they act like personality parameters – e.g. an agent with a low threshold for building tasks will spontaneously start building when seeing moderate stimulus (like minor floods), whereas a high-threshold agent waits for a strong stimulus (major flood or explicit Overmind signal). We can use these thresholds within the policy or as heuristic triggers: the environment can provide "stimulus" values (like current flood level or project need) and the agent will take on a task if stimulus > θ. This mechanism can produce specialization and division of labor. For implementation, each agent can store its θ values (drawn from some distribution at init for diversity), and the decision logic (either in policy network or in a higher-level behavior controller) checks stimuli vs thresholds to decide actions if using a more rule-based approach.

# 3. Ant-Style Pheromone Routing System

To facilitate pathfinding and coordination, the simulation includes an **ant-inspired pheromone field** on the grid. We represent pheromone trails as a scalar value `τ_{ij}` on each directed edge from cell *i* to neighbor *j* [23]. In practice, we can maintain a pheromone matrix or dictionary: e.g. for each cell index *i*, store `pheromone[i][j]` for each neighbor *j*. If the grid has adjacency lists, we can store `pheromone_trails = { (i,j): τ_ij }`. Initially all τ are zero (no trails).

**Pheromone update rules:** Each time-step, the pheromone levels evaporate and are reinforced by agent movements: - **Evaporation:** For every edge, pheromone decays by a factor `(1 - ρ)` per step. For example, `τ_{ij}(t+1) = (1 - ρ) * τ_{ij}(t) + ...` deposition term [24]. The evaporation rate ρ ∈ (0,1] is set in config (e.g. ρ = 0.1 means 10% trail loss per step) [25]. This prevents stale trails from persisting indefinitely, forcing adaptation to changing environments. - **Deposition:** When agents move along an edge, they deposit pheromone. If a set `K_{ij}(t)` of agents traveled from *i* to *j* at time *t*, then we add Δτ to that edge. A simple scheme is a fixed deposit per agent: e.g. `Δτ_{ij}(t) = ∑_{k ∈ K_{ij}} δ_k`, where δ_k might be a constant small amount for any moving agent [26] [27]. We can also make δ depend on success: for instance, if an agent reaches a food source or successfully delivers material on that route, deposit a larger amount (reinforcing useful paths) [28]. This mimics ants laying more pheromone on routes that lead to food. - The pheromone class (`PheromoneField`) will handle these updates each step via `update_pheromones(movements)` where `movements` is a list of tuples (i,j) for each agent movement. It multiplies all τ by (1-ρ) then adds contributions for each movement.

**Routing influence:** Pheromone trails influence agent movement decisions similarly to ant foraging behavior. When an agent (not in a specific project) is moving, instead of random exploration it will bias toward cells with higher pheromone. For example, if an agent at cell *i* has neighbor cells *j1, j2,...*, we can compute a heuristic weight η_{i→j} for each direction based on pheromone: e.g. η_{ij} = (τ_{ij} + ε)^α * (h_{ij})^β, where ε is a tiny baseline to allow exploration, and h_{ij} is some static heuristic (like preference to move downhill if looking for water, or towards home if carrying food). We then sample a direction with probability proportional to η. In practice, we might not hand-code this in final form; instead, we can include pheromone values in the agent's observation and let the policy network learn to favor higher τ. However, encoding known pheromone-following behavior can improve efficiency: for example, the agent's policy could implement a softmax over neighbor pheromone levels (this is akin to an *ant-like movement policy* [29] ).

**Pheromone use cases:** - **Resource trails:** When one beaver finds a rich patch of vegetation, as it travels back and forth to the lodge with food, it will leave a pheromone trail. Other hungry beavers can detect and follow this trail to quickly reach the food source instead of random searching. - **Material delivery:** If multiple trips are needed to carry wood to a dam site, pheromone trails can form between the lodge (or wood source) and the project location, guiding workers along efficient routes. - **Dynamic updating:** If a route becomes inefficient (e.g. flooded or blocked by a new dam), and agents stop using it, its pheromone will evaporate and alternative routes will emerge (typical ant colony optimization behavior).

The pheromone system's implementation is modular: we can easily **disable** it (e.g. set ρ=1 to instantly evaporate or simply not update τ) to test ablations. It interacts with agents through the observation (agents can query `env.pheromone.get_level(i,j)` for neighbors) and through the environment step (after agents move, `env` calls `pheromone_field.update()` to evaporate and deposit). We ensure thread-safety or order consistency: update pheromones after all agents have moved in a step to include everyone's contributions from that step.

## 4. Bee-Style Project Recruitment (Waggle Dance Dynamics)

MycoBeaver agents coordinate on large construction tasks (like building dams or lodges) via a **project recruitment system** inspired by honeybee waggle dances. We define a set of possible projects `P` (work sites or tasks) with unique IDs [30]. At any time, a project *p* has: - A **region of interest** `I_p` (one or more grid cells where the work happens) [31]. For a dam, this might be the cells along a stream that need blocking; for a lodge, a specific pond location. - An **estimated quality** `Q_p(t)` [32] – a value that represents the expected benefit of the project. Scouts evaluate this when they discover the site. For example, `Q_p` could be computed as a weighted sum of factors like resource availability, hydrological impact, and safety [33] [34]. In practice: *ResourceAbundance* might be the total nearby wood/food that would be gained or the materials available; *HydroImpact* might estimate how much the dam would improve water distribution or reduce flood risk; *Safety* could reflect distance from predators or stability of the site; *DistanceCost* subtracts the travel cost from the colony to the site [35]. The spec suggests a formula: *Q_p* = *w1*Resource + w2*HydroImpact* + *w3*Safety – w4*DistanceCost* [33] [36]. We can implement this calculation in the environment when a scout surveys a site. - A **recruitment intensity** `R_p(t)` (the "waggle dance" signal strength) [37]. This represents how enthusiastically the colony is being recruited to project *p*. It starts at 0 and increases when scouts promote the project.

**Project discovery and advertising:** Scout agents roam the environment (beyond immediate food needs) searching for opportunities. If a scout finds a promising location (e.g. it reaches a river segment that could be dammed and notes abundant trees there), it will evaluate the site's quality `Q_p` using the above criteria [38] [39]. A new project entry is created in the environment's project list with that `Q_p`. The scout then returns to a lodge (or the colony area) to **advertise** the project. In simulation, we can say when a scout arrives at a lodge cell with a project idea, it performs a waggle dance. This will **increase the recruitment signal** `R_p` for that project. Formally, each timestep, we update recruitment as:

$$R_p(t+1) = (1-\kappa)\, R_p(t) + \sum_{k \in S_p(t)} \gamma_{\text{dance}}\, f(Q_p(t))$$

where $S_p(t)$ is the set of scout agents that returned to advertise project *p* at time *t* [40] [41]. Here $ \kappa$ is a decay factor (dance enthusiasm evaporates over time) [42], $\gamma_{\text{dance}}$ is a gain scaling, and $f(Q)$ is an increasing function of project quality (for example $f(Q)=\exp(\lambda Q)$ to make high-quality projects get disproportionately higher recruitment) [43] [44]. In implementation, each step we can decay all projects' $R_p$ by $(1-\kappa)$, then for each scout that is currently advertising (we might mark the scout with the project it's promoting and how long it will dance), we add $\gamma_{\text{dance}} f(Q_p)$ to that project's $R$. This mimics the bee waggle dance: more scouts and better sites -> stronger recruitment signal.

We also optionally implement **cross-inhibition**: to sharpen consensus, a very strong signal for one project can suppress others. E.g., $R_q(t+1) \leftarrow R_q(t+1) - \chi R_p(t)$ for all $q \neq p$ being advertised [45] [46]. The parameter $\chi$ controls how much success of one project's recruitment comes at the expense of others. This encourages the colony to eventually focus on the best project rather than splitting efforts indefinitely.

**Agent recruitment and project work:** Idle worker agents at the colony pay attention to the waggle dances (recruitment signals). In our model, an agent can sense the current $R_p$ values (either directly as part of observation or implicitly via some probabilistic rule). We design the decision such that the probability an idle worker chooses to join project *p* increases with $R_p$. For example, each free agent could pick a project with probability $P(\text{join }p) = R_p / \sum_{q} R_q$, or if there's a threshold, join when $R_p$ exceeds some value. In practice, we might encode $R_p$ signals in the agent observation as part of a "colony signals" vector (or the agent at the lodge could observe the dances). Simpler: we can automatically assign a certain number of nearest idle workers to a project once $R_p$ crosses a threshold (like how bees implement quorum). But a probabilistic ongoing recruitment better captures partial commitments.

Once an agent commits to a project, we set its `p_k^proj = p` (and possibly role becomes "builder"). It will travel to the project site (following pheromones or known route, or using Physarum network guidance) carrying any needed materials, and start executing build actions (e.g. placing sticks to reduce `d` in that region's cells). Multiple agents on the same project coordinate implicitly by working in the same region (we can simulate that each build action by any agent contributes to the project's progress). The project could have a required total work or materials; we track a progress metric. For example, if a dam project needs 100 units of wood placed, each build action uses 1 wood and we increment progress until 100 reached, then mark project complete.

**Project completion and lifecycle:** When a project is finished (dam built), we set some indicator (dams fully constructed might be reflected by `d_i = 0` in those cells, and perhaps log an event). Completed projects could be removed from active list or marked inactive; their $R_p$ would drop to 0 (no one should recruit to a finished project). If a project is abandoned (no dance support and no agents working on it for some time), we also remove it. Agents on a completed or aborted project revert to idle (or original role). New projects can continually arise as scouts find new opportunities, thus the system is dynamic.

All project logic resides in a **ProjectManager** within the environment. This could be a class or just part of the environment state: - It holds a dict of active projects with fields: `region_cells`, `Q`, `R`, `progress`, `status`. - Method `add_project(region, Q)` to create a new entry (called by a scout's action). - Method `update_recruitment()` to do the $R_p$ update each step (using lists of advertising scouts). - Method `assign_agents_to_projects()` which can either be deterministic or called via agent policies. We might simply feed $R$ into agent observations and let their policy decide to navigate to the project. For more deterministic recruitment in initial implementation, we can do: if a project has $R$ above a threshold and still needs workers, automatically flag some idle agent's internal state to join it. - It will also handle checking completion: e.g. if `progress >= required` or if dam permeability in region has hit target, then finalize project. - Optionally handle failure: if environment changes such that $Q$ becomes very low (e.g. the site is no longer useful or got destroyed), we could drop the project and maybe penalize wasted effort.

This bee-like recruitment system ensures not all agents randomly build their own dams; instead the colony can focus on the most promising site identified by scouts. It also introduces positive feedback: a good site attracts more workers, which finishes it faster – analogous to consensus building. We will include configuration flags to disable or modify this system for ablation (for example, turning off cross-inhibition or limiting number of projects to test sequential vs parallel project building).

## 5. Physarum-Inspired Adaptive Transport Network

On a layer beneath agent activities, we run a **Physarum polycephalum-inspired network** that continuously computes efficient pathways for resource transport. We model the grid as a graph $G=(I, E)$ where nodes $I$ are the cells and edges $E$ connect neighboring cells [47] [48]. Each edge $(i,j)$ has: - A **conductivity** $D_{ij}(t) \ge 0$ [49], which represents the capacity or "thickness" of a tubular connection between cells. This is analogous to slime mold's protoplasmic tube thickness. - A **length/cost** $L_{ij} > 0$ [50], representing the distance or difficulty of that connection. We initialize $L_{ij}$ based on geometric distance (e.g. 1 for orthogonal neighbors) and can add terrain factors (e.g. climbing cost proportional to elevation difference, or water cost) [51]. For instance, $L_{ij}(t)$ might be dynamically increased by large elevation differences or unfavorable terrain [51] (so the physarum avoids steep climbs or certain areas if possible).

The Physarum network updates by solving flow distribution problems between multiple sources and sinks: - We define a set of **source nodes** $S$ and **sink nodes** $T$ based on the current state [52]. The choice of sources/sinks depends on the "commodity" we are optimizing: - For **food transport**: sources could be cells with high food/wood availability (vegetation clusters), and sinks could be the lodge location(s) where food is needed. - For **building material transport**: sources might overlap with food (wood is gathered from trees) or could be specific storage sites, and sinks are active dam project sites needing material. - For **water flow optimization** (if we treat water as a commodity for physarum): sources could be water input cells (spring or river inflow) and sinks could be downstream targets or the area we want to keep watered (wetlands). However, since we explicitly simulate water physics, we might focus the physarum on resource (log/food) transport paths. - We assign a net injection $b_i$ for each node: $b_i = +I_s$ if i is a source (inject amount $I_s$), $b_i = -I_t$ if i is a sink (demand $I_t$), and 0 otherwise [53] [54]. We ensure total supply equals total demand for flow conservation [55] (in practice, we can set each source injection to 1 and each sink demand such that sum equals those sources).

Given current conductivities $D_{ij}(t)$, we compute the steady-state flow (like an electrical circuit): each edge behaves according to Darcy's or Ohm's law with $D_{ij}$ as conductance. The node potentials Φ are found by solving the linear system [56] [57]:

$$\sum_{j:(i,j)\in E} D_{ij}(t)\frac{1}{L_{ij}}\left(\Phi_i - \Phi_j\right) = b_i, \quad \forall i,$$

which basically means flow conservation at each node (Kirchhoff's law) [58] [59]. We solve for potentials Φ_i (e.g. by matrix inversion or iterative solvers). Then flow on each edge is $Q_{ij}(t) = D_{ij} \frac{1}{L_{ij}} (\Phi_i - \Phi_j)$ (this is the flux from i to j).

Next, we **adapt conductivities** based on flow: edges carrying more flow get reinforced (thickened), while low-flow edges decay. We use an update like Tero's model [60] [61]:

$$D_{ij}(t+1) = D_{ij}(t) + \Delta t\big(\alpha_D\, g(|Q_{ij}(t)|) - \beta_D\, D_{ij}(t)\big).$$

Here $\alpha_D$ is a reinforcement rate, $\beta_D$ a decay rate, and $g(|Q|)$ is an increasing function of flux magnitude [62] [63] (commonly $g(|Q|)=|Q|^\gamma$ for some $0<\gamma\le1$). Intuitively, an edge used heavily will see $D$ increase, unused edges will slowly vanish [64] [65]. We ensure $D_{ij}$ stays within reasonable bounds (e.g. clamp to max or min >0 to avoid numerical issues). Over time, this process yields a sparse network of high-conductivity edges connecting sources to sinks via near-optimal paths [66] – effectively finding good routes for resource transport that adapt if sources/sinks move.

We embed this Physarum logic in a class `PhysarumNetwork`. It contains: - Data: current `D` for each edge (could store as a dict of edge->conductivity or a matrix shape [N, N] if N small and we treat absent edges as 0). - Methods: `compute_flow(sources, sinks, injections)` – sets up and solves the linear system for Φ (likely using `numpy.linalg.solve` on a sparse matrix). `update_conductivity(Q)` – applies the above rule to adjust each D. - It might also have a method `adapt_costs(env)` to update edge lengths `L_ij` based on environment context. For example, we can make $L_{ij}$ dynamic: if an area is flooded or terrain changes (like new obstacles), increase $L$ to discourage physarum from using that path [67] [68]. The spec suggests: $L_{ij}(t) = L_{ij}(0) + \lambda_h \bar{h}_{ij}(t) + \lambda_z |z_i - z_j|$ [68] – i.e. if the path goes through water depth $\bar{h}$ or uphill (difference in elevation $|z_i-z_j|$), it becomes effectively longer cost. We can periodically update $L$ using current water levels or other factors, making the slime mold avoid, say, deep water or steep climbs unless necessary.

We run the Physarum update at each simulation step (or maybe every few steps if performance is an issue). The environment will do: 1. Determine `sources` and `sinks` given current context (e.g. if a project is active, its region center is a sink for wood; the trees around are sources; plus lodge as sink for food). This logic can be in the Scenario or environment config. 2. Call `physarum.compute_flow()` to get flows. 3. Call `physarum.update_conductivity()`. 4. Use the updated D values to influence agent behavior or environment.

**Using Physarum for agent guidance:** The high-conductivity edges essentially mark efficient “highways” for transport [69]. We integrate this with agent pathfinding: an agent carrying food or wood can query the physarum network to decide its route. Practically, an agent at cell i can look at neighbor j and consider the current $D_{ij}$ and $L_{ij}$. We define a **movement desirability** metric η that combines pheromone and physarum suggestions. For instance, $\eta_{ij} \propto \frac{D_{ij}(t)}{L_{ij}(t)}$ [70] – meaning edges that physarum has made thick (high $D$) and that are short (low $L$) are very attractive. This was suggested in the spec: “beavers interpret high-conductivity edges as recommended transport corridors” [69]. In implementation, we can include $D_{ij}$ (or a normalized form of it) in the agent's observation of each neighbor, or bias the agent's movement probabilities accordingly. For example, multiply the earlier pheromone weight by $(1 + \kappa_D \frac{D_{ij}}{L_{ij}})$ for some scaling $\kappa_D$. This way, even without explicit global planning, agents will naturally follow the efficient routes the Physarum layer discovers. Physarum is **differentiable and global**, so in principle if training end-to-end, agents could learn to utilize it (though here we treat it as a fixed algorithmic module).

We can optionally run **multiple Physarum layers (multi-commodity)** [71] [72], one for each resource type (e.g. one optimizing water distribution, one for food, one for wood). Each would have its own $D^{(c)}$ and flows. This adds complexity, so initially one unified network or a primary one for food/wood transport suffices. If implemented, agents could then consider multiple $D^{(c)}$ depending on context (e.g. use the food-transport network when hungry, etc.). The code can be structured to allow plugging in multiple layers if needed (each with its own source/sink sets).

The PhysarumNetwork class is integrated into the environment: e.g. `env.physarum` object. In the `env.step()`, after agents act and environment updates, we call `env.physarum.tick()` to update flows. The module is independent of agent policy (no learning parameters, purely adaptive), which makes it a robust backbone the agents can rely on. For ablation, we can disable it by freezing $D$ or not using it in agent decisions to observe differences.

## 6. Overmind & Wisdom Signals (Global Meta-Control)

Above all agents, we implement an **Overmind** entity – essentially a global controller that observes the ecosystem and influences high-level parameters. The Overmind produces a **wisdom signal** each step that reflects the overall health and stability of the environment [73] [74]. It can also take **actions** in the form of adjusting meta-parameters of the system to guide the colony.

**Overmind observation:** Rather than raw grid data, the Overmind gets aggregate features of the system. We compile a feature vector $\Xi_t$ including metrics like [75] [76]: - Average water level in a core area and its variability (e.g. $\bar{h}_{core}(t)$ and $\sigma(t)$) [77] to gauge flood stability in the main habitat zone. - Flood indicators $C_{flood}(t)$ and drought indicators $C_{drought}(t)$ [78] – e.g. counts or severity of any cells that have too high water or too low water respectively. - $C_{failure}(t)$ [79] – a measure of recent failures (like dam breaches or agent deaths). For instance, each time a dam is breached or an agent dies, this counter could increment or be a rolling average of “failure events.” - Habitat richness $R_{habitat}(t)$ [80] – e.g. a composite score of biodiversity or environment quality. We could use total vegetation biomass or number of water-adjacent green cells as a proxy for habitat health. - Colony population or activity: e.g. the fraction of alive agents ($|A_t|/K$) and possibly distribution of their roles [81]. - Current project recruitment levels ${R_p(t)}_{p\in P}$ [82] – indicating where colony effort is focused. - Possibly physarum network metrics (like number of thick edges or network redundancy) to capture infrastructure complexity.

This observation is essentially a summary of the state at time *t*. It is fed into the Overmind’s policy network.

**Wisdom signal:** Based on the above features, the Overmind computes a scalar *wisdom* $w_t$. This is akin to a global reward signal for ecosystem wellbeing. We define $w_t$ to increase when the environment is stable and thriving, and decrease with instability or calamity. For example [22] [83]:

$$w_t = -\big(\lambda_\sigma \sigma_{h,core}(t) + \lambda_F C_{flood}(t) + \lambda_D C_{drought}(t) + \lambda_B C_{failure}(t)\big) + \lambda_H R_{habitat}(t).$$

In words, $w_t$ is higher when water levels are consistent (low variance, no extreme floods/droughts) and no major failures, and when habitat is rich [83]. It’s lower if there are wild swings in water (floods/dry spells) or if the colony or network is collapsing. This wisdom signal can be computed directly from the metrics each step. It serves two purposes: (1) as a **global performance measure** to evaluate how well the simulation is doing, and (2) as the **Overmind’s reward** if we treat the Overmind as a learning agent. Indeed, we can set the Overmind’s objective to maximize the expected discounted sum of wisdom signals [84] [85] (analogous to a return in RL).

**Overmind actions (Meta-parameters):** The Overmind can influence the colony by tuning parameters that individual agents or processes use. We allow it to adjust: - **Pheromone evaporation rate ρ:** Increasing ρ makes pheromones evaporate faster (forcing exploration), decreasing $\rho$ preserves trails longer [86]. - **Recruitment decay or intensity ($\kappa$ or $\gamma$_dance):** The Overmind could tweak how easily the colony focuses on one project vs keeps options open. E.g., lowering $\kappa$ (less decay) keeps dances going longer, potentially causing multiple projects to compete; increasing $\gamma$_dance intensifies recruitment so consensus is reached faster [87] [88]. - **Threshold adjustments {S_m}:** The Overmind might raise or lower the colony-wide sensitivity for tasks m [87] [88]. For instance, if it perceives an upcoming flood season (e.g. high variance in water), it could globally lower the threshold for dam-building so more agents start building proactively. - **Network decay rates:** It could adjust $\beta_D$ in the physarum layer, influencing how quickly old paths are pruned. Or adjust which commodities the physarum prioritizes if multiple. - **Role reassignments (indirectly):** Overmind could bias the ratio of scouts to workers by setting a parameter that agents use to decide switching (for example, Overmind could signal "we need more scouts now" if habitat richness is dropping). - Other possible parameters: agent metabolism rates (slow down energy use if resources scarce), priorities for habitat vs survival, etc. We denote the vector of all such meta-parameters as $\psi_t$ [89].

In each step, the Overmind's **policy** $\pi_{OM}$ takes the observation $\Xi_t$ and outputs a choice of meta-parameters $\psi_t$ [90]. For implementation, we define an `Overmind` class with: - A PyTorch neural network (policy) that inputs the feature vector and outputs a set of continuous action values (some might be bounded 0–1, e.g. $\rho_t \in [0,1]$, which we can enforce by a sigmoid layer or clipping). - Possibly a value network if using actor-critic for Overmind (to estimate expected return from current state). - A method `adjust_parameters(env, agents)` that takes the chosen $\psi$ and applies it: e.g. set `env.pheromone.evap_rate = ρ_t`, set `env.recruitment.kappa = new κ` or `env.recruitment.gamma_dance = new γ`, broadcast new thresholds to agents (agents could each have their $\theta_{k,m}$ scaled by an Overmind factor $S_m(t)$ for task m). - The Overmind also could directly broadcast signals to agents – for example, in the "wisdom signal grid" implementation, Overmind might emit spatial signals (like a gradient or marker in areas that need attention). The provided code snippet suggests something called `WisdomSignalGrid` and agents emitting/receiving signals. We can interpret this as the Overmind possibly seeding some global field that agents sense. However, to keep it simple: Overmind mostly works through parameter tuning and the scalar reward.

Training the Overmind: we will treat it as an RL agent in a meta-control loop. During training episodes, after each step, we can give the Overmind a reward = $w_t$ (or perhaps a shaped reward that also includes a small penalty for drastic parameter changes if we want to smooth its actions). Overmind's policy is updated to maximize accumulated wisdom. This creates a hierarchy: the beaver agents handle day-to-day actions for survival and building (their learning is mostly about immediate tasks), while the Overmind learns higher-level strategies (like balancing exploration vs exploitation by tuning pheromone evaporation, or maintaining diversity vs focus by tuning recruitment signals). For example, if it notices the physarum network collapsing to a single path (low redundancy), Overmind might deliberately increase pheromone evaporation and decrease recruitment intensity to push agents to explore new routes and projects [91] [92]. Or if the colony is too scattered, it might boost one project's signal to concentrate efforts.

The Overmind class will also contain: - Storage of recent observations and actions (to feed into RL updates). - A `reset()` to clear any history at episode start, possibly with an initial default parameter set. - Logging of chosen parameters for analysis (to see how it's intervening).

The **Overmind and agents interaction** is designed carefully to avoid non-stationarity issues in training. Since Overmind's actions change the environment's dynamics from the agents' perspective, we effectively have a two-level game. We might train the agents' policies and Overmind's policy alternately or simultaneously. A likely training strategy: train beaver agents on a relatively short horizon (e.g. many episodes with a fixed Overmind or with a slowly adapting Overmind), and train Overmind on a longer horizon with the agents' behavior model considered part of the environment. For now, we assume both can be trained together with proper RL algorithms (multi-agent or hierarchical RL approach), or we can simplify by fixing one while training the other in phases.

In summary, the Overmind provides a top-down adaptation mechanism making the simulation **adaptive and resilient**. Its presence is optional: we can run experiments with Overmind disabled (it would then use default parameters and no global feedback) to see how much the colony benefits from meta-control.

## 7. Reward Structure and Objectives

The simulator uses a reward structure that balances individual survival with collective eco-engineering success. We outline the main components: - **Survival Reward/Penalty:** Each agent staying alive contributes to reward, while death incurs a penalty. For instance, an agent might get +1 reward for each time step it remains alive (encouraging basic survival behaviors like eating and avoiding hazards). If an agent dies (starvation or accident), a substantial negative reward (e.g. -100) is given, either to that agent or even to all agents if we use a shared reward (since we want the colony to avoid losing members). This emphasizes self-preservation. - **Hydrological Stability Reward:** We provide reward for maintaining water levels in a desired range. This ties closely to the Overmind's wisdom components. For example, every time step without a flood (no cell exceeds a certain water depth) could give a small positive reward, and every flood event could give a negative reward. Similarly avoiding drought (no critical water source goes dry) yields reward. We can also use a continuous measure: e.g. reward -= (variance of water distribution) or penalize each unit of overflow water. In practice, the Overmind's wisdom signal already encodes a penalty for flood variance [22], so if we choose to share a global reward, we might directly use $w_t$ as a component of the agents' reward too (scaled down). - **Habitat/Biodiversity Reward:** If the environment becomes more lush and supportive of life (due to beaver activity creating wetlands), that's positive. We can measure total vegetation or the number of cells with both water and vegetation (indicating a healthy wetland). The reward can include a term like $+\lambda$ * (fraction of grid with high moisture & vegetation). Overmind's wisdom includes $+ \lambda_H R_{habitat}$ [83], which covers this. So again $w_t$ encapsulates it. But to ensure agents directly care, we might give them a direct reward for planting/regrowing vegetation or expanding water coverage. - **Project Completion Reward:** Successfully building a dam or lodge is a major colony achievement and should yield a positive reward. We can give a one-time bonus when a project finishes (shared among agents or to those involved). This encourages agents to cooperate on large tasks even if they are costly short-term. - **Resource Gathering Reward:** To drive foraging, we can give a small reward to an agent when it eats or collects wood. This ensures they learn to gather food to stay alive (intrinsic via survival) but also possibly to store resources. However, to avoid selfish hoarding, the reward for gathering could be modest, and long-term survival reward already incentivizes it. - **Penalty for Inefficiencies or Damage:** We may add minor penalties for things like wasted materials (if an agent drops wood or builds a failed structure that later breaks). If a dam fails (gets washed out), that might register in $C_{failure}$ and thus reduce $w_t$ (Overmind's penalty) [93]. We could explicitly penalize that event to the agents as well to reinforce building sturdy dams or maintaining them. - **Time or Efficiency:** To motivate timely action, we might include a small per-step penalty to encourage the colony to achieve goals faster (or simply rely on discount factor in RL for that effect). Alternatively, no explicit time penalty, just let discounting handle it.

**Global vs individual reward:** Given this is a cooperative multi-agent system aiming to improve the environment, a shared reward approach is reasonable. We can define the **colony reward at time t** as the

Overmind's wisdom signal $w_t$ (or a scaled version). Then each agent receives that as their reward at each step (plus maybe tiny personal terms for satiety to ensure they don't ignore eating). This aligns everyone with the same objective: maximize ecosystem health, which implicitly requires them to survive and build. The Overmind itself, if trained, gets reward = $w_t$ by design.

However, training purely on such a global signal can be hard for the agents (sparse credit assignment). In practice we might use a **combined reward**: e.g. *Agent reward = α * (survival/food reward for that agent) + β * (global habitat reward)*. That way they take care of basics (eat, don't drown) but also contribute to global improvements. For example, if α=1 and β=1, an agent gets a reward for being alive and additional reward when the habitat gets better due to a dam (even if that agent didn't directly build it, if using shared reward). We can tune these weights.

**Value functions:** We will incorporate critics (value networks) for both the agents and Overmind if using actor-critic methods, or use policy gradient with reward-to-go for agents. The reward structure must be properly fed into these learning updates.

**Metrics and logging:** We implement a `MetricsComputer` that at episode end (or during) computes things like survival rate (percent of agents alive), mean energy, cooperation metrics (how many agents participated in projects), habitat quality, etc. The prompt mentions metrics like *field_coherence*, *ethical_score* in code – possibly related to how cohesive the network is or whether agents avoided unnecessary destruction. We will focus on clearly defined ones: - *Survival_rate* = (# agents alive at end) / K. - *Mean_energy* = average final energy. - *Cooperation_rate* = maybe proportion of agents that worked on a collective project vs solitary. - *Habitat_score* = normalized habitat richness. - *Flood_frequency* or *Water_variance* to capture stability. - We output these in logs for analysis (and Overmind observation partly overlaps with them).

The **reward and termination** logic is handled in the environment's `step()`. Typically, an episode could run for a fixed number of steps (e.g. simulate a season or year) or until a failure condition (all agents died or certain time passed after which evaluation happens). We will have `done` = True if all agents die or if we reached max_steps. In cooperative training, we might run to the full length regardless to collect data, unless a catastrophic failure (e.g. 0 agents or 100% flood) occurs.

Finally, all these reward components are configurable. One can turn off the habitat reward to see if agents still build dams just from survival necessity (perhaps not). Or turn off survival reward to see if global reward alone is enough (risky because agents might then sacrifice themselves for the greater good, which isn't desirable). The code will allow easy toggling of these via the config.

## Integration of Components and System Flow

With all major components defined, we structure the system into modules and outline the execution cycle. The **module structure** (as Python classes/modules) will be roughly: - `environment.py` – defines `MycoBeaverEnv(gym.Env)` containing grid state and sub-systems. It implements `reset()` and `step()`. - `agents.py` – defines `BeaverAgent` class, and possibly different policy classes or network architectures for greedy/contemplative. - `pheromone.py` – defines `PheromoneField` class (with methods for update and query). - `projects.py` – defines `Project` class and `ProjectManager` that holds active projects and updates recruitment. - `physarum.py` – defines `PhysarumNetwork` class and related data structures. - `overmind.py` – defines `Overmind` class with its policy network and parameter adjustment logic. - `metrics.py` – defines `MetricsComputer` and data classes for storing metrics (e.g. a `ColonyMetrics` dataclass to aggregate results). - `training_pipeline.py` – optional module to handle running multiple episodes, optimizing the PyTorch models (agents and Overmind) with chosen RL algorithms, saving checkpoints, etc. - `config.py` – defines configuration structures (using perhaps `dataclass` for sections: environment config, agent config, overmind config, etc.), including random seeds for reproducibility, toggles for components (use_pheromone, use_physarum, etc.), and hyperparameters (learning rates, discount, etc.). - `visualization.py` (or integrated in env) – functions to render the grid state for visualization.

**Class interactions / contracts:** - The `MycoBeaverEnv` holds instances of `BeaverAgent` for each agent, an instance of `PheromoneField`, an instance of `ProjectManager`, and `PhysarumNetwork`, and `Overmind`. Each of these has an interface: - `BeaverAgent.decide_action(observation)` returns an action (which could be an enum or structured e.g. (`move_dir`, `forage`, etc.)). - `PheromoneField.update(agent_moves)` requires the list of moves executed in this step to deposit pheromones. - `ProjectManager.update(agents, scouts_advertising)` updates all projects (recruitment signals, checks completion) and possibly returns a list of assignments (but likely assignment is handled by agents themselves via signals). - `PhysarumNetwork.step(sources, sinks)` updates the physarum network. The environment must determine sources and sinks each step (e.g. all current food sources and needs). - `Overmind.act_and_adjust(env_metrics)` produces new meta-parameter values and applies them. - The `BeaverAgent` has some reference or method to interact with environment: e.g., when an agent performs a build dam action, it should call something like `env.project_manager.contribute(project_id, material=1)` to update dam progress and environment dam cells. For foraging, agent calls `env.harvest(x,y)` to reduce vegetation and maybe increase soil moisture (simulate disturbance). We can handle these in environment's `step()` when applying actions for consistency (the environment can interpret each action type appropriately).

**Time-step update flow:** We break the `env.step()` into a sequence of phases: 1. **Overmind phase:** At the very start of a step, the environment compiles the Overmind observation (global metrics from last step) and passes it to `Overmind.decide_action()`. The Overmind returns a set of meta-parameter adjustments. Environment then applies these: e.g. update pheromone evaporation rate, update dance gain, broadcast any threshold changes to agents (if needed). This way, the current step's behaviors will reflect the new parameters. *(If Overmind is not used, this step is skipped or does nothing.)* 2. **Observation phase:** For each agent, assemble its observation. This could include: - Local grid slice (e.g. a radius-2 view of terrain and resources around the agent's position, maybe encoded as a matrix or channels). - Pheromone info (e.g. values of pheromone on outgoing edges from agent's cell). - Any nearby pheromone from other agents? (Probably just edges from current cell is enough). - Project signals: if the agent is at a lodge or otherwise, perhaps the top few $R_p$ values could be included (or a simplified "most advertised project ID"). - Overmind wisdom signals: possibly a general colony state indicator. We might not feed $w_t$ directly to agents to avoid trivializing the task (and $w_t$ is more a performance metric). Instead, if Overmind wants to directly influence agents, it should do so via parameters or via some "wisdom field". If we use the concept of a **wisdom signal grid** (like a spatial field of signals), the Overmind could place gradients that agents sense (e.g. marking areas of interest). The provided code snippet had agents getting `signal_map = signal_grid.get_signals_at(agent.x, agent.y)` and `agent.emit_signals()` – which suggests a two-way communication field. For now, we can omit this complexity or treat it as pheromone-like signals that Overmind might add. - Internal state: agent's own energy, hunger, etc., so the policy knows its needs. We package this into a structured observation (could be a dict with e.g. `{"local_grid": array, "pheromone": vector, "global_signals": vector}`). 3. **Agent decision phase:** Each agent (if alive) uses its policy to choose an action. We gather all actions into a list or dict. (If using a vectorized policy for all agents, we could concatenate observations and run a batch through a shared network, but initially per-agent calls are fine.) 4. **Apply actions (Environment dynamics):** The environment processes all agent actions, ideally simultaneously: - Movement: compute new positions for all agents. We might need to handle conflicts (if two agents move into the same cell, is it allowed? Given beavers are not too territorial with each other, we might allow multiple agents per cell, but perhaps not two lodges or something. To simplify, we can allow multiple agents in one cell, representing working side by side). - Forage: for any agent that chose to forage in its cell, reduce the vegetation there by a certain amount and increase the agent's satiety. If multiple agents forage the same cell, the vegetation is reduced multiple times (bounded by available biomass). - Build dam: for actions targeting a project region, we update that project's progress. If the action is to build at the agent's current cell and that cell is part of project p's region, reduce `d` at that cell (e.g. $d_i = \max(0, d_i - \Delta d)$ for some small $\Delta d$ per action). Also increment project p's work counter. If the agent needed to expend a carried wood resource, mark that resource as used. - Other possible actions: If any agent did a `advertise(project)` action (scout dancing), we mark that agent as part of $S_p(t)$ for the recruitment update. In practice, we might model the dance as taking up that step for the scout (it doesn't move or do other work, just signals). - Update agent internal states: increase wetness for those in water cells, reduce energy according to what they did (moving and building cost more than resting). If any agent's energy or satiety fell below 0, mark it as dead (it won't act further). - The environment also checks if any external events happen due to actions (e.g. if a dam was built that stops water, the water update next will reflect it). 5. **Environment dynamics update:** Now we update the environment state for the next step: - Water flow: use the updated dam states (permeabilities) and compute the new water distribution (as described in section 1). This is done after building actions so the benefits of dams (raising water upstream) materialize gradually. Also apply rainfall for this step. - Vegetation growth: after any foraging, regrow some vegetation in all cells. - Soil moisture: update moisture based on new water distribution. - Pheromone evaporation/deposition: Call `pheromone_field.update()` with the list of actual moves from step 4. This will evaporate trails and add new pheromone where agents moved [24] [25]. - Physarum network update: Determine sources and sinks (e.g. now if some trees were depleted by foraging, their source capacity might reduce; if new project started, add it as a sink). Run `physarum.step()` to recalc flows and update conductivities. - Project recruitment signals: Call `project_manager.update_recruitment()`. Scouts that danced in step 4 contribute to boosting $R_p$ [40] [41]; also decay all $R$ by (1-κ). If any project reached completion criteria, mark it finished (which could trigger a one-time reward). - Role updates: possibly reassign roles if needed. For example, if a project got enough workers and a scout is no longer needed, it might revert to scout role or go idle. 6. **Reward calculation:** Compute the reward for each agent (and Overmind): - Compute global metrics (flood count, etc.) and then $w_t$ (wisdom) [22] [83]. - If using shared reward: assign each living agent a reward = some function of $w_t$. We might normalize $w_t

```
or just use it directly if it's already scaled
appropriately. Additionally, add any individual component: e.g. +1 if the agent ate
this step, -100 if agent died this step, etc.
   - If using individual rewards: for each agent, sum the relevant parts (we ensure
the global parts are included).
   - Overmind's reward = $w_t$ (by definition of its objective) [84] .
   - If a project completed this step, we might add a bonus to all agents or those
who worked on it.
   - These rewards are returned or passed to learning algorithm.
7. **Learning updates:** If doing online learning (like in the same step):
                     -     For     each     agent,     call
```

agent.learn(reward, next_obs) `where` next_obs `is the new observation after environment update. This could be`

a Q-learning update or storing transition in replay memory. In a policy gradient approach, we would instead store the (obs, action, reward) and handle learning after an episode, not each step.
   - Overmind similarly can store its transition (Ξ_t, ψ_t, reward=w_t, Ξ_{t+1}) for later update.
   - We maintain trajectories for each actor to perform RL updates at the end of an episode or after a batch of episodes (depending on algorithm). The `TrainingPipeline` can orchestrate running many steps, then doing a gradient update on the neural nets (e.g. using PPO, A2C, etc., with PyTorch).
8. **Logging & Visualization hooks:** After updating, the environment can log this step's metrics (e.g. append to a list or write to tensorboard: current time, average satiety, w_t, number of active projects, etc.). If visualization is enabled (say every N steps or if running in a not-headless mode), call `env.render()`:
   - The render function might use Matplotlib to draw the grid: use color maps for water depth (blue intensity), soil/vegetation (green), maybe overlay small icons for beavers, and highlight dam cells (brown or a barrier symbol) and lodges. Alternatively, use Pygame to blit images for each entity on a grid.
   - A simple approach: create a PIL image or Matplotlib figure where each cell's color is determined by a combination of properties (e.g. terrain background by elevation, overlay blue for water, green for vegetation, etc.), and draw agents as colored dots. This can be saved as a frame or displayed live.
   - Also possibly plot some graphs of metrics over time as the simulation runs for debugging (like water level in core area vs time).
9. **Step termination:** Check if the episode should terminate. Criteria: if time_steps >= max_steps (scenario duration complete), or if all agents are dead (colony collapse), or potentially if a target condition achieved (like all projects done and environment stable – though in an open-ended sim that might not have a clear end). The `done` flag is set accordingly. If done, the environment can compile final metrics for output.

After an episode, the `SimulationResult` (with final metrics and possibly trajectories) can be stored. If training, the pipeline will then possibly reset and start a new episode (with maybe random variations or different seed if multiple seeds are run).

## Training and Extensibility Considerations

The implementation is built with **research-level code quality** in mind: modular, instrumentable, and reproducible.

- **PyTorch integration:** All learning components (agent policies, Overmind policy, value estimators) are PyTorch models, enabling end-to-end differentiable training. We will ensure the environment can provide a gradient-friendly interface if needed (for example, if we ever do differentiable physarum or differentiate through environment dynamics for planning, though currently environment is mostly not differentiable due to conditional logic). In practice, we'll use standard RL algorithms (not differentiating through the environment, just through policy networks).

- **Multi-agent training:** We can use a centralized training approach with shared policy (e.g. all beavers share one policy network to learn a general behavior, which is common in homogeneous agents MARL). The policy network can have role and perhaps agent ID as input to break symmetry if needed. Alternatively, one could assign separate networks to scouts vs workers (multi-policy MARL). Our design is flexible: the config can specify if policies are shared. The `TrainingPipeline` can collect experiences from all agents and aggregate them for a single optimizer update (this is equivalent to treating the whole colony as multiple instances of the same agent in policy gradient). This helps with sample efficiency and ensures all agents learn together. Overmind's policy is separate and has its own optimizer.
- **Gym interface compatibility:** `MycoBeaverEnv` as a `gym.Env` can be integrated with RL libraries. If we treat each agent's action as part of a joint action, the action space could be a tuple or dict. A simple method: define the action space as a dict with agent IDs mapping to sub-action spaces (Gym's newer versions or the PettingZoo library support dict observations/actions for multi-agent). Alternatively, one could flatten it by having a single vector action of length K (for K agents), but that's unwieldy. We likely will not rely on off-the-shelf library for multi-agent (since many algorithms would need modification for our hierarchy), but our custom pipeline will use the environment in a loop manually. We provide conversion if needed (like a wrapper that presents a single-agent view for centralized training).
- **Reproducibility:** The code will allow setting seeds: we set the NumPy and PyTorch random seeds at the start (and possibly random seed for environment's random events). The config will have a `seed` field, and our `reset()` uses it to seed any random number generation (e.g. randomize initial agent positions, or rainfall patterns, but in a repeatable way). We will also allow multi-seed experiment runs easily: e.g. the training pipeline can loop over several seeds and output average results.
- **Logging and observability:** We integrate Python's `logging` or printouts for key events (like "Dam project 2 started at (10,5) with quality 0.8" or "Flood detected at time 50"). The metrics recorded each step can be exported to CSV or tensorboard for analysis. Agents and Overmind can be set to verbose debug modes where they print decisions or certain internal values.
- **Visualization:** As mentioned, a render method will produce visual output. Additionally, we can produce periodic snapshots of the physarum network (since that's interesting: e.g. output an image of the conductivity network by drawing brighter lines on high D edges). This could be done via matplotlib by drawing lines between node coordinates with alpha proportional to D. The combination of grid view and network overlay will help to see how the slime mold network evolves relative to terrain and agent paths.
- **Modularity & Ablation:** Every major sub-system (pheromone, recruitment, physarum, Overmind) can be toggled or adjusted via config:
- Setting `use_pheromones=False` could cause agents to ignore pheromone (and we skip updating it).
- `use_physarum=False` could freeze D values or not instantiate PhysarumNetwork.
- `use_overmind=False` could bypass all Overmind adjustments (keeping default parameters constant) and possibly not use the wisdom signal in rewards.
- We can also run greedy vs contemplative by adjusting agent reward – e.g. contemplative might incorporate global reward whereas greedy might only care about individual reward. Or have two sets of agents and compare outcomes.

- The code is organized so that these systems interact through clear interfaces (as described). For example, even if Overmind is off, the environment can still compute $w_t$ for logging; if physarum is off, agents still function using pheromones or local heuristics.

- **Extendability:** The design allows adding more complexity:

- We could introduce predators or other species by adding another agent type or an environmental hazard that reduces beaver population, to see if beaver behavior adapts.
- We might add a plant growth model where dams actually increase vegetation downstream (which would feed back positively).
- The “ethical score” or other metrics could be incorporated if we define what they mean (e.g. minimal environmental disruption vs improvement – likely beyond our scope now).

- The multi-commodity physarum or additional role types (like a dedicated “engineer” beaver role) can be added by extending classes (e.g. PhysarumNetwork could handle multiple layers by generalizing its storage and update loop).

- **Performance:** We will profile the simulation to ensure it can run many steps reasonably. The heaviest part might be solving the linear system for physarum each step. If N (grid cells) is large (say 100x100 = 10k cells), solving a 10k linear system per step might be slow. But we can either use a sparse solver or reduce frequency (e.g. update physarum every 5 steps instead of every step, since its adaptation is slower timescale). Also, we can limit physarum to a subgraph of interest (like between certain hubs rather than entire grid). We’ll plan these optimizations as needed. Pheromone updates and local agent actions are cheap in comparison.

Finally, we ensure the **system is suitable for multi-run experiments**: we can easily run the simulation in headless mode for X episodes and aggregate metrics, perform ablations by changing config toggles, and support saving/loading learned models for analysis. All random elements (initial conditions, random weather) can be controlled to do fair comparisons across runs.

With this complete design, the MycoBeaver simulator will allow us to simulate and study the interactions of beaver-inspired agents shaping their environment, guided by ant pheromones, bee recruitment, slime-mold networks, and an Overmind’s guidance – all implemented in a maintainable, extensible codebase ready for reinforcement learning experiments. The combination of these components aims to produce a robust, adaptive colony that improves its environment while ensuring its own survival, aligning short-term needs with long-term ecosystem stability [22] [94]. All aspects are configurable and observable, enabling in-depth experimentation and verification of the bio-inspired strategies.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
91 92 93 94 mycobeaver.txt
file://file_000000001a9471fd9ff4e47304b5680d