FLOWRRA Technical Documentation

Flow Reconfiguration-Reconfiguration Agent

Overview

FLOWRRA is a multi-agent system that achieves temporal harmonization through repulsive density fields, wave function collapse recovery mechanisms, and reinforcement learning. The system enables swarm robotics to maintain coherence while adapting to dynamic environments.

Core Architecture

1. Density Function Estimator (`DensityFunctionEstimator_RL.py`)

Purpose: Implements speed-aware repulsive density fields with "comet-tail" projection to guide node movement and prevent collisions.

Mathematical Foundation

Repulsion Kernel Projection: For each detected object with position p and velocity v, the system projects repulsion kernels forward in time:

$$\text{future\_pos}(k) = p + v * k$$

Where $k \in [0, k_f]$ represents future timesteps.

Gaussian Kernel Application: Each future position receives a weighted repulsion value:

$$R(x,y,k) = \eta * \gamma_f^k * \exp(-0.5 * (k/\sigma_f)^2)$$

Where:

- η (eta): Learning rate for splatting repulsion (default: 0.5)
- γ_f (gamma_f): Decay factor for forward projection (default: 0.9)
- σ_f (sigma_f): Width of Gaussian kernel (default: 0.05)
- k_f: Number of future steps projected (default: 5)

Local Grid Computation: Each node maintains a local 4×4 repulsion grid instead of processing a global field:

```python
def get_repulsion_potential_for_node(self, node_pos, repulsion_sources):
    local_grid = np.zeros((4, 4))

    for source in repulsion_sources:
        for k in range(self.k_f):
            future_pos = source['pos'] + source['velocity'] * k

            # Map to local grid coordinates
            local_x = clip((future_pos[0] - node_pos[0] + 0.5) * 4, 0, 3)
            local_y = clip((future_pos[1] - node_pos[1] + 0.5) * 4, 0, 3)

            kernel_value = exp(-0.5 * (k / sigma_f)^2)
            local_grid[local_y, local_x] += eta * (gamma_f^k) * kernel_value

    return beta * local_grid
```

Field Decay: The global field (used for visualization) undergoes exponential decay:

$$R(t+1) = R(t) * (1 - \lambda)$$

Where (λ) (decay_lambda) = 0.01 (default).

---

2. Node Position System (`NodePosition_RL.py`)

Purpose: Manages individual node dynamics, sensing, and discrete action execution.

Node Dynamics

Position Update with Toroidal Wrapping:

$$pos_{new} = mod(pos_{old} + \Delta pos * dt, 1.0)$$

Velocity Calculation:

```python
```

```
def velocity(self):
    delta = self.pos - self.last_pos
    # Handle toroidal boundary conditions
    toroidal_delta = mod(delta + 0.5, 1.0) - 0.5
    return toroidal_delta
```

Action Space

Position Actions (4 discrete choices):

- 0: Move left: `Δpos = [-move_speed, 0]`
- 1: Move right: `Δpos = [+move_speed, 0]`
- 2: Move up: `Δpos = [0, +move_speed]`
- 3: Move down: `Δpos = [0, -move_speed]`

Angle Actions (4 discrete choices):

- 0: Rotate left: `angle_idx -= rotation_speed`
- 1: Rotate right: `angle_idx += rotation_speed`
- 2: Move forward in current direction
- 3: No-op

Sensing Model

Distance Calculation with Toroidal Geometry:

python

```
def sense_nodes(self, all_nodes):
    for other_node in all_nodes:
        delta = other_node.pos - self.pos
        toroidal_delta = mod(delta + 0.5, 1.0) - 0.5
        distance = norm(toroidal_delta)

        if distance < sensor_range:
            bearing_rad = arctan2(toroidal_delta[1], toroidal_delta[0])
            # Record detection...
```

Sensor Parameters:

- `sensor_range`: 0.15 (15% of environment width)

- Returns: position, distance, bearing, relative velocity, type

---

3. Wave Function Collapse (`WaveFunctionCollapse_RL.py`)

Purpose: Implements temporal coherence recovery through historical state analysis and manifold smoothing.

Collapse Detection

Coherence Threshold: System triggers collapse when coherence < 0.15 for sustained period.

Historical Analysis: Maintains sliding window of `history_length` states (default: 200) with coherence scores.

Recovery Mechanism

Coherent Tail Search:

```python
def find_coherent_tail(self):
    for i in range(len(history) - tail_length):
        tail = history[i:i + tail_length]
        if all(h['coherence'] >= collapse_threshold for h in tail):
            return tail
    return None
```

Manifold Smoothing: Uses Gaussian-weighted temporal averaging to compute stabilized positions:

```python
def smooth_positions(coherent_tail):
    positions_over_time = array([snapshot['positions'] for snapshot in tail])

    # Gaussian kernel with recent bias
    weights = exp(-0.5 * arange(tail_length)[::-1]^2 / (tail_length/4)^2)
    weights /= sum(weights)

    # Weighted temporal average
    smoothed_positions = einsum('t,tni->ni', weights, positions_over_time)
    return smoothed_positions
```

Fallback Strategy: If no coherent tail exists, applies small random perturbation:

```
pos_new = mod(pos_old + N(0, 0.05^2), 1.0)
```

---

4. RL Integration (`FLOWRRA_RL.py`)

State Representation

Per-Node State Vector (36 dimensions):

- Node position: `[x, y]` (2D)
- Node velocity: `[vx, vy]` (2D)
- Neighbor data: `[distance, bearing, vx, vy]` × 2 neighbors (8D)
- Obstacle data: `[distance, bearing, vx, vy]` × 2 obstacles (8D)
- Local repulsion grid: flattened 4×4 = 16D

Total State Size: `num_nodes × 36`

Reward Structure

Coherence Reward:

```python
def calculate_coherence(self):
    coherences = []
    for node in nodes:
        repulsion_at_pos = get_local_repulsion(node.pos)
        coherence = 1.0 / (1.0 + repulsion_at_pos)
        coherences.append(coherence)
    return mean(coherences)
```

Exploration Reward:

```python
```

```
def calculate_exploration_reward(self):
    # Movement incentive
    movements = [norm(pos_current - pos_previous) for node in nodes]
    movement_reward = mean(movements) * 57.0

    # Spread reward (optimal distance ≈ 0.3)
    pairwise_distances = compute_all_distances(nodes)
    spread_reward = max(0, 1.0 - abs(mean(pairwise_distances) - 0.3))

    # Velocity alignment
    alignment = compute_velocity_alignment(nodes)
    alignment_reward = max(0, alignment) * 0.5

    return movement_reward + spread_reward * 1.5 + alignment_reward
```

Total Reward:

```
R_total = (coherence_pos + coherence_final)/2 + exploration_reward + baseline_reward
```

Collapse Penalty: `-2.0` to `-3.0` for states requiring WFC intervention.

Two-Stage Action Execution

1. **Position Stage**: Apply position actions, update density field, check coherence
2. **Angle Stage**: Apply angle actions, final density update, compute final reward

This staged approach allows the system to recover from position-based instabilities before committing to angular changes.

---

5. Environment Management

Environment A (`EnvironmentA_RL.py`)

- Manages node initialization and state transitions
- Maintains simulation history for analysis
- Handles random seed management for reproducible experiments

Environment B (`EnvironmentB_RL.py`)

- Simulates external obstacles (fixed and moving)

- Grid-based discrete obstacle movement with collision avoidance
- Converts discrete grid positions to continuous coordinates for node sensing

---

Key Mathematical Properties

Temporal Coherence

The system maintains coherence through:

$$C(t) = (1/N) \Sigma[1/(1 + R_i(t))]$$

Where $R_i(t)$ is the repulsion potential at node i's position at time t.

Stability Condition

System remains stable when:

$$C(t) \geq \tau_{collapse} \text{ for sustained period}$$

Default: $\tau_{collapse} = 0.15$

Recovery Dynamics

WFC recovery seeks states where:

$$\forall t \in [t_{tail} - L, t_{tail}]: C(t) \geq \tau_{collapse}$$

With manifold smoothing providing:

$$pos_{recovered} = \Sigma\, w(t) * pos(t)$$

Where $w(t)$ follows Gaussian weighting favoring recent stable states.

Computational Complexity

- **Per-Node State Computation**: O(k_f × num_detections)
- **Global System Update**: O(N × M) where N = nodes, M = average detections
- **WFC Recovery**: O(history_length × tail_length) worst case

- **RL Training**: Standard DQN complexity with experience replay

---

Applications

Warehouse Robotics

- Autonomous navigation with collision avoidance
- Swarm coordination for package handling
- Recovery from system failures without human intervention

Satellite Industries

- Formation flying with station-keeping
- Debris avoidance through predictive repulsion
- Autonomous reconfiguration after perturbations

Smart Cities (Future)

- Traffic flow optimization
- Sensor network coordination
- Infrastructure resilience through coherence monitoring

This architecture represents a novel approach to multi-agent systems that prioritizes temporal coherence and graceful recovery over pure optimization, enabling robust autonomous behavior in dynamic environments.