Bachelor's Thesis
Academic Year 2022

Modeling Head-Bobbing in Pigeon Locomotion using Reinforcement Learning

Faculty of Environment and Information Sciences,
Keio University

Mioto Takahashi

A Bachelor's Thesis
submitted to Faculty of Environment and Information Sciences, Keio University
in partial fulfillment of the requirements for the degree of
BACHELOR of Environment and Information Sciences

Mioto Takahashi

Thesis Committee:

Professor Tatsuya Hagino (Supervisor)
Professor Takashi Hattori (Co-Supervisor)

Abstract of Bachelor's Thesis of Academic Year 2022

Modeling Head-Bobbing in Pigeon Locomotion using Reinforcement Learning

Category: Science / Engineering

Summary

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In efficitur porta augue, at interdum nunc lobortis at. Morbi feugiat facilisis justo, vitae maximus dolor. Cras convallis at elit in porta. Fusce lobortis tortor nibh, quis imperdiet arcu luctus quis. Mauris imperdiet urna eu mauris aliquet, vitae tincidunt orci dapibus. Vestibulum convallis elit ut velit accumsan cursus. Pellentesque lacus lacus, blandit eu felis vitae, pellentesque dignissim est.

Keywords:

Reinforcement Learning, Biomimetic, Pigeon, Locomotion

Faculty of Environment and Information Sciences, Keio University

Mioto Takahashi

Contents

List of Figures

List of Tables

Chapter 1

Background

1.1. Reinforcement Learning

Reinforcement learning is a type of control system that attempts to execute tasks described by manually set cost functions by minimizing them using optimization algorithms or learning algorithms. In the case of deep reinforcement learning, the controller is modeled using deep neural networks and the cost function is minimized using gradient descent algorithms.

Reinforcement learning divides control systems into an agent and an environment. The agent acts the controller of the system that inspects its environment's state and sends output signals, or actions, that affect the environment. This mutual interactions creates a feedback loop between the two modules.

In the context of a pigeon tasked to move forward, the pigeon's brain and its nervous system connected to each limb act as the agent, and its surroundings, such as the ground and arbitrary objects on it, act as the environment. The environment outputs a state, such as the global position of the pigeon, which is used as the input for the agent. Using the provided state, the agent calculates and outputs an action, such as the torque of each joint in the pigeon's body. The action alters the state of the environment, and the environment outputs a new state and a reward. The reward describes how well the pigeon was able to execute its task, such as the current position of the pigeon relative to its previous timestep. The agent is updated to output sequences of actions that maximizes the cumulative reward, or return. The return can be interpreted as the inverted

or negated cost function.

Chapter 2

Preliminary Research

2.1. Modeling Biological Phenomena using Robotics

As Webb discusses the topic of utilizing robotics to aid research in biology [3], where he argues that robotic modeling can be used for proposing or testing hypotheses. This can be accomplished by depicting the hypotheses in the form of algorithms or hardware configurations in robots and observing their emergent behaviors. The generated behaviors can then be compared with their biological counterparts. In particular, robotic models implemented based on preliminary hypotheses can be used as null hypotheses, which can be validated after observing behaviors exhibited by them.

While Webb argues about the potential of using robotics for research in biology, he also highlights the limitations of such methods. Since robotic models naturally contain information on all of the assumptions made on the hypotheses of the biological phenomena they reference, discrepancies between the model and the phenomena must be carefully taken into consideration during experiments. For example, if a bipedal robot were to be used to model a pigeon's forward locomotion, it should account for noises and disturbances in the pigeon's sensory input signals and neurological output signals that stimulate their muscles. Additionally, environmental factors such as rough terrain that pigeons walk on should be simulated in the experiment.

From such perspective algorithmic models are inferior to models that utilize real robots since, as Webb states, they cannot confront problems seen in physical

environments, such as noises or disturbances, that are not seen in virtual environments, such as physics engines. As such, algorithms in biology are often used for modeling neurological functionalities. In the context of modeling pigeons' locomotion, algorithmic models can represent the neurological controllers for the pigeons. In this case, the pigeons' interaction between their brains, their morphology, and their surrounding environments can be seen as a control system.

Webb identifies incremental modeling as a method to tackle the discrepancies between complexity of biological phenomena and their model counterparts. Incremental modeling builds multiple models for the same hypotheses, where each model gets more complex with biological or physical details.

A biological model of a pigeon with high biological and physical details would consist of the musculoskeletal mechanism of pigeons, a spiking neural network to control all of the muscles, and a surrounding environment with complex details, such as bumpy terrain and colors. This would be too complicated of a problem to tackle in a single research paper, especially considering that the learning and control mechanism of spiking neural networks is currently unclear. Even if the musculoskeletal model were to be controlled by a fully connected deep neural network, the biological and environmental details, including the colors and the sheer number of controllable muscles, would heighten the input and output dimension of the neural network, making training it using deep reinforcement learning extremely difficult.

Our research specifically focuses on a simplified model of the pigeon, which includes simplified representations of limb control and retinal input.

2.2. Head-Bobbing in Pigeons

The "head-bob" behavior in pigeons consists of stabilizing the global location and orientation of the head and altering them periodically. Such are dubbed as the hold phase and the thrust phase, respectively [1].

Frost and Davies' have proposed hypotheses regarding the functionalities of such behavior have been proposed [2] [1]. Both proposals highlight the use of the hold phase as a means to stabilize vision and the use of the thrust phase as a means to detect motion parallax and determine the distance between objects.

2.2.1 Hold Phase

Frost's hypothesis links the functionality of the hold phase to the detection of backward motion within the eye [2]. Pigeons' heads, while flying or moving forward, would be detecting objects whose movements can be distinguished from the surrounding stationary objects. Since stationary objects would be moving backwards relative to the pigeons' eyes, desensitizing backward motion would be necessary for such distinctions to be detected. However, this desensitization would be detrimental to the pigeons' object recognition while the pigeons' heads are stationary. The hypothesis highlights the existence of "backward notch" cells which counteract the aforementioned desensitization. Such cells would be activated when the pigeons' vision is stabilized and allowing them to distinguish objects moving backward relative to stationary objects, hence the necessity of the hold phase during locomotion.

Davies' hypothesis challenges this notion and highlights the lack of necessary conditions in Frost's hypothesis to induce a hold phase by stating that "they would fail to detect objects moving backwards through the visual field at velocities similar to that of the bird, as their responses could not be discriminated from those caused by self-induced motion" [1]. Davies proposes the existence of cells that detect objects' movement relative to stationary backgrounds regardless of their directions.

In the context of our model, combining the two hypotheses leads to a mechanism that stabilizes the head of the pigeon relative to arbitrary stationary objects and activate cells that detect arbitrary motion during the hold phase.

2.2.2 Thrust Phase

Frost proposes a hypothesis which links the behavior of thrusting the head forward to depth perception [2]. He takes the two most common methods of depth perception in biology, stereopsis and motion parallax, into consideration, and points out that, upon examining the anatomy of birds' heads, the eyes are seen to be placed on the opposite sides of the skull instead of having both of them be placed on the frontal area. Such positioning of the eyes in pigeons implies that the overlapping areas in visions of both eyes are reduced compared to animals that

mainly use stereopsis for identifying depth of objects. Given such observation, he argues that pigeons are more likely to resort to utilizing motion parallax over the alternative, given the lateral eye placement in pigeons' heads.

Davies later elaborates on the hypothesis by modeling the retinal information of objects [1].

Davies models the depth perception caused by motion parallax as

$$\dot{\theta} = \frac{v \sin \theta}{x} \tag{2.1}$$

where θ is the angle of the object relative to the eye, v is the velocity of the pigeon's head, and x is the distance between the object and the head, or the depth of the object. The equation indicates that the depth of the object can be derived if the velocity of the head and the angular velocity of the object within the retina are given.

Davies further extends this model to account for detecting the difference in depth between objects. Davies differentiates the angular velocity of a single object and the relative angular velocity between an object at depth x and an object at depth y with the notation Δ.

$$\Delta\dot{\theta} = v \sin \theta \left(\frac{1}{x} - \frac{1}{y} \right) \tag{2.2}$$

Given the two equations, Davies argues that maximizing the speed of head would result in better differentiation between objects with different depths. By thrusting the head forward, the pigeon would be able to detect stationary objects in different depths, which cannot be detected during the hold phase. As an example Davies mentions the role of the eye in detection of food, such as the detection of bread crumbs lying on the ground close by as opposed to lampposts seen far away.

2.2.3 Regarding Kinematic Functionalities

When building our pigeon model, in addition to the hypotheses proposed regarding the hold phase and the thrust phase, we must take the effect of the torques of the neck joints and the movement of the head generated by them. Intuitively

such motion would alter the balance of the entire pigeon, leading to mutual adjustments between the bipedal walk cycle and the neck control for head positioning. Additionally, the head-bobbing motion could be hypothesized to function as a means to balance the pigeon's forward locomotion. However as Davies argues in his paper [1], since head-bobbing is not exhibited during fast forward locomotion, such as flying, it is unlikely that such behavior has kinematic purposes. Frost's findings [2] further support this idea by demonstrating that pigeons stabilize their head in one global coordinate regardless of the body's global velocity, it is likely that head-bobbing's functionalities are solely based on vision.

Chapter 3

Approach

We define a simplified 2-dimensional model of pigeons based on incremental modeling. The pigeon model consists of 3 joints connecting one body representing the head, 2 bodies representing the neck, and one body representing the torso. The model's physics, mainly the collision and gravity, is simulated in a 2 dimensional physics engine. The torso's orientation and y-position is fixed while its x-position is incremented by a constant value. This represents forward locomotion at a constant speed.

Additionally, we build control systems for the model using reinforcement learning. By using reinforcement learning, we can train the controller to maximize reward functions that represent hypotheses or manually-defined trajectories for the bodies in the model to follow.

As the baseline for the model's control system, we attempt to recreate the head-bob movement by setting a target position for the head's position to match every timestep. The target position is first defined at a set location in front of the pigeon model T relative to the position of its torso. The target then acts as a static position in the global coordinate for the head to follow. If the distance between the target position and the torso's position goes below a set threshold value, the target is repositioned at the same location T relative to the torso's position.

We compare the trajectories of the bodies in the baseline control system to those generated by the control system that represents preliminary hypotheses.

Chapter 4

Experiments

Chapter 5

Analysis

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In efficitur porta augue, at interdum nunc lobortis at. Morbi feugiat facilisis justo, vitae maximus dolor. Cras convallis at elit in porta. Fusce lobortis tortor nibh, quis imperdiet arcu luctus quis. Mauris imperdiet urna eu mauris aliquet, vitae tincidunt orci dapibus. Vestibulum convallis elit ut velit accumsan cursus. Pellentesque lacus lacus, blandit eu felis vitae, pellentesque dignissim est.

References

[1] DAVIES, M. N., and GREEN, P. R. Head-bobbing during walking, running and flying: relative motion perception in the pigeon. *Journal of Experimental Biology 138*, 1 (1988), 71–91.

[2] Frost, B. The optokinetic basis of head-bobbing in the pigeon. *Journal of Experimental Biology 74*, 1 (1978), 187–195.

[3] Webb, B. What does robotics offer animal behaviour? *Animal behaviour 60*, 5 (2000), 545–558.

Appendix

A. OpenAI Gym for Simplified Model of Pigeons' Head Control

A.1 Manually Defined Head Trajectory (Baseline)

```
from Box2D import *
import gym
from gym import spaces

from math import sin, pi, sqrt
import numpy as np
from copy import copy, deepcopy

# anatomical variables ("macros")
BODY_WIDTH = 10
BODY_HEIGHT = 5

LIMB_WIDTH = 5
LIMB_HEIGHT = 2

HEAD_WIDTH = 3

ANGLE_FREEDOM = 0.6

# control variables/macros
MAX_JOINT_TORQUE = 200 #70
```

```
MAX_JOINT_SPEED = 5 #10
VELOCITY_WEIGHT = 1.0 #0.9
LIMB_DENSITY = 0.1 ** 3
LIMB_FRICTION = 5

VIEWPORT_SCALE = 6.0
FPS = 60

HEAD_OFFSET_X = 10
HEAD_OFFSET_Y = 2

class PigeonEnv3Joints(gym.Env):
    metadata = {"render.modes": ["human", "rgb_array"], "video.
        frames_per_second": FPS}

    def __init__(self,
                 body_speed = 0,
                 reward_code = "head_stable_manual_reposition",
                 max_offset = 0.5):
        """
        Action and Observation space
        """

        # 3-dim joints' torque ratios
        self.action_space = spaces.Box(
            np.array([-1.0] * 3).astype(np.float32),
            np.array([1.0] * 3).astype(np.float32),
        )
        # 2-dim head location;
        # 1-dim head angle;
        # 3x2-dim joint angle and angular velocity;
        # 1-dim x-axis of the body
        # [NEW] 2-dim target head location
        high = np.array([np.inf] * 12).astype(np.float32) # formally 10
        self.observation_space = spaces.Box(-high, high)
```

```
    """
    Box2D Pigeon Model Params and Initialization
    """
    self.world = b2World()                      # remove in Framework
    self.body = None
    self.joints = []
    self.head = None
    self.bodyRef = [] # for destruction
    self.body_speed = body_speed
    self._pigeon_model()

    """
    Box2D Simulation Params
    """
    self.timeStep = 1.0 / FPS
    self.vel_iters, self.pos_iters = 10, 10

    self.viewer = None

    """
    Assigning a Reward Function
    """
    self._assign_reward_func(reward_code, max_offset)

"""
Define Reward Function and Necessary Parameters
"""
def _assign_reward_func(self, reward_code, max_offset):
    if "head_stable_manual_reposition" in reward_code:
        self.max_offset = max_offset

        self.relative_repositioned_head_target_location = np.array(
            self.head.position) - np.array([0, HEAD_OFFSET_Y])
        self.head_target_location = self.
            relative_repositioned_head_target_location + np.array(
            self.body.position)
```

```
        self.head_target_angle = self.head.angle
        self.reward_function = self._head_stable_manual_reposition

        if "strict_angle" in reward_code:
            self.reward_function = self.
                _head_stable_manual_reposition_strict_angle

    else:
        raise ValueError("Unknown reward_code")

"""
Box2D Pigeon Model
"""
def _pigeon_model(self):
    # params
    body_anchor = np.array([float(-BODY_WIDTH), float(BODY_HEIGHT)])
    limb_width_cos = LIMB_WIDTH / sqrt(2)

    self.bodyRef = []
    # body definition
    self.body = self.world.CreateKinematicBody(
        position = (0, 0),
        shapes = b2PolygonShape(box = (BODY_WIDTH, BODY_HEIGHT)), #
            x2 in direct shapes def
        linearVelocity = (-self.body_speed, 0),
        angularVelocity = 0,
        )
    self.bodyRef.append(self.body)

    # neck as limbs + joints definition
    self.joints = []
    current_center = deepcopy(body_anchor)
    current_anchor = deepcopy(body_anchor)
    offset = np.array([-limb_width_cos, limb_width_cos])
    prev_limb_ref = self.body
    for i in range(2):
```

```
    if i == 0:
        current_center += offset

    else:
        current_center += offset * 2
        current_anchor += offset * 2

    tmp_limb = self.world.CreateDynamicBody(
        position = (current_center[0], current_center[1]),
        fixtures = b2FixtureDef(density = LIMB_DENSITY,
                                friction = LIMB_FRICTION,
                                restitution = 0.0,
                                shape = b2PolygonShape(
                                    box = (LIMB_WIDTH, LIMB_HEIGHT))
                                        ,
                                ),
        angle = -pi / 4
    )
    self.bodyRef.append(tmp_limb)

    tmp_joint = self.world.CreateRevoluteJoint(
        bodyA = prev_limb_ref,
        bodyB = tmp_limb,
        anchor = current_anchor,
        lowerAngle = -ANGLE_FREEDOM * b2_pi, # -90 degrees
        upperAngle = ANGLE_FREEDOM * b2_pi, # 90 degrees
        enableLimit = True,
        maxMotorTorque = MAX_JOINT_TORQUE,
        motorSpeed = 0.0,
        enableMotor = True,
    )

    self.joints.append(tmp_joint)
    prev_limb_ref = tmp_limb

# head def + joints
```

```
        current_center += offset
        current_anchor += offset * 2
        self.head = self.world.CreateDynamicBody(
            position = (current_center[0] - HEAD_WIDTH, current_center
                [1]),
            fixtures = b2FixtureDef(density = LIMB_DENSITY,
                                    friction = LIMB_FRICTION,
                                    restitution = 0.0,
                                    shape = b2PolygonShape(
                                        box = (HEAD_WIDTH, LIMB_HEIGHT)),
                                    ),
        )
        self.bodyRef.append(self.head)

        head_joint = self.world.CreateRevoluteJoint(
            bodyA = prev_limb_ref,
            bodyB = self.head,
            anchor = current_anchor,
            lowerAngle = -ANGLE_FREEDOM * b2_pi, # -90 degrees
            upperAngle = ANGLE_FREEDOM * b2_pi, # 90 degrees
            enableLimit = True,
            maxMotorTorque = MAX_JOINT_TORQUE,
            motorSpeed = 0.0,
            enableMotor = True,
        )
        self.joints.append(head_joint)

        # head tracking
        self.head_prev_pos = np.array(self.head.position)
        self.head_prev_ang = self.head.angle

    def _destroy(self):
        for body in self.bodyRef:
            # all associated joints are destroyed implicitly
            self.world.DestroyBody(body)
```

```
def _get_obs(self):
    # (self.head{relative}, self.joints -> obs) operation
    obs = np.array(self.head.position) - np.array(self.body.position)
    obs = np.concatenate((obs, self.head.angle), axis = None)
    for i in range(len(self.joints)):
        obs = np.concatenate((obs, self.joints[i].angle), axis = None
            )
        obs = np.concatenate((obs, self.joints[i].speed), axis = None
            )
    obs = np.concatenate((obs, self.body.position[0]), axis = None)

    # complement a target position
    obs = np.concatenate((obs, self.head_target_location - np.array(
        self.body.position)),
                         axis = None)

    obs = np.float32(obs)
    assert self.observation_space.contains(obs)
    return obs

def reset(self):
    self._destroy()
    self._pigeon_model()
    return self._get_obs()

def _head_target_reposition_mechanism(self):
    # detect whether the target head position is behind the body edge
         or not
    if self.head_target_location[0] > self.body.position[0] - float(
        BODY_WIDTH + HEAD_OFFSET_X):
        self.head_target_location = np.array(self.body.position) + \
            self.relative_repositioned_head_target_location

    head_dif_loc = np.linalg.norm(np.array(self.head.position) - \
            self.head_target_location)
    head_dif_ang = abs(self.head.angle - self.head_target_angle)
```

```
    return head_dif_loc, head_dif_ang

"""
Modular Reward Functions
"""
def _head_stable_manual_reposition(self):
    # This method is separated from step(), since there are variables
        used
    # that are only defined in with this strain of reward functions
    head_dif_loc, head_dif_ang = self.
        _head_target_reposition_mechanism()

    reward = 0
    # threshold reward function with static offset
    if head_dif_loc < self.max_offset:
        reward += 1 - head_dif_loc/self.max_offset

        if head_dif_ang < np.pi / 6: # 30 deg
            reward += 1 - head_dif_ang/ np.pi

    return reward

def _head_stable_manual_reposition_strict_angle(self):
    head_dif_loc, head_dif_ang = self.
        _head_target_reposition_mechanism()

    reward = 0
    # threshold reward function with static offset
    if head_dif_loc < self.max_offset:
        if head_dif_ang < np.pi / 6: # 30 deg
            reward += 1 - head_dif_ang/ np.pi

    return reward

def _head_stable_movement_minimizer(self):
    reward = 0
```

```
    return reward

def step(self, action):
    assert self.action_space.contains(action)
    # self.world.Step(self.timeStep, self.vel_iters, self.pos_iters)
    # Framework handles this differently
    # Referenced bipedal_walker
    # self.world.Step(1.0 / 50, 6 * 30, 2 * 30)
    self.world.Step(1.0 / FPS, self.vel_iters, self.pos_iters)
    obs = self._get_obs()

    # MOTOR CONTROL
    for i in range(len(self.joints)):
        # Copied from bipedal_walker
        self.joints[i].motorSpeed = float(MAX_JOINT_SPEED * (
            VELOCITY_WEIGHT ** i) * np.sign(action[i]))
        self.joints[i].maxMotorTorque = float(
            MAX_JOINT_TORQUE * np.clip(np.abs(action[i]), 0, 1)
        )

    reward = self.reward_function()

    done = False
    info = {}
    return obs, reward, done, info

def render(self, mode = "human"):
    from gym.envs.classic_control import rendering
    if self.viewer is None:
        self.viewer = rendering.Viewer(500, 500)

        # Set ORIGIN POINT relative to camera
        self.camera_trans = b2Vec2(-250, -200) \
        + VIEWPORT_SCALE * self.bodyRef[0].position # camera moves
            with body
```

```
## Needs head_stable_manual_reposition reward function to
    execute
try:
    # init visualize max_offset
    render_target_area = rendering.make_circle( \
        radius=VIEWPORT_SCALE * self.max_offset,
        res=30,
        filled=True)
    target_translate = rendering.Transform(
        translation = VIEWPORT_SCALE * self.
            head_target_location - self.camera_trans,
        rotation = 0.0,
        scale = VIEWPORT_SCALE * np.ones(2)
    )
    render_target_area.add_attr(self.target_translate)
    render_target_area.set_color(0.0, 1.0, 0.0)
    self.viewer.add_geom(render_target_area)
except:
    pass

# init translation and rotation for each limb
self.render_polygon_list = []
self.render_polygon_rotate_list = []
self.render_polygon_translate_list = []
for body in self.bodyRef:
    polygon = rendering.FilledPolygon(
        body.fixtures[0].shape.vertices
    )
    rotate = rendering.Transform(
        translation = (0.0, 0.0),
        rotation = body.angle,
    )
    translate = rendering.Transform(
        translation = VIEWPORT_SCALE * body.position - self.
            camera_trans,
        rotation = 0.0,
```

```
                scale = VIEWPORT_SCALE * np.ones(2)
            )
            polygon.set_color(1.0, 0.0, 0.0)
            polygon.add_attr(rotate)
            polygon.add_attr(translate)
            self.render_polygon_list.append(polygon)
            self.render_polygon_rotate_list.append(rotate)
            self.render_polygon_translate_list.append(translate)
            self.viewer.add_geom(polygon)

    # Update ORIGIN POINT relative to camera
    self.camera_trans = b2Vec2(-250, -200) \
    + VIEWPORT_SCALE * self.bodyRef[0].position # camera moves with
        body

    ## Needs head_stable_manual_reposition reward function to execute
    try:
        # update max_offset shape translation
        new_target_translate = VIEWPORT_SCALE * self.
            head_target_location - self.camera_trans
        self.target_translate.set_translation(new_target_translate
            [0], new_target_translate[1])
    except:
        pass

    # update body rotation and translation
    for i, body in enumerate(self.bodyRef):
        self.render_polygon_rotate_list[i].set_rotation(body.angle)
        new_body_translate = VIEWPORT_SCALE * body.position - self.
            camera_trans
        self.render_polygon_translate_list[i].set_translation(
            new_body_translate[0], new_body_translate[1])

    return self.viewer.render(return_rgb_array = mode == "rgb_array")

def close(self):
```

```
        # self._destroy()
        # self.world = None

        if self.viewer:
            self.viewer.close()
            self.viewer = None
```

A.2 Pigeons' Head Control Based on Retinal Inputs

```
import PigeonEnv3Joints, VIEWPORT_SCALE
import numpy as np
import gym
from gym import spaces

class PigeonRetinalEnv(PigeonEnv3Joints):

    def __init__(self,
                 body_speed = 0,
                 reward_code = "motion_parallax"):

        """
        Object Location Init (2D Tensor)
        """
        self.objects_position = np.array([[-30.0, 30.0],
                                          [-30.0, 60.0],
                                          [-60.0, 30.0],])
        self.objects_velocity = np.array([[0.0, 0.0],
                                          [1.0, 0.0],
                                          [-1.0, 0.0],])

        """
        Init based on superclass
        Reward function is defined here
        """
        super().__init__(body_speed, reward_code)
```

```
    """
    Redefining Observation space
    """
    # 2-dim head location;
    # 1-dim head angle;
    # 3x2-dim joint angle and angular velocity;
    # 1-dim x-axis of the body
    high = np.array([np.inf] * 10).astype(np.float32) # formally 10
    self.observation_space = spaces.Box(-high, high)


"""
Retinal coords (angles); Within [-np.pi, np.pi]
"""
def _get_retinal(self, object_position):
    # normalized direction of object from head
    object_direction = object_position - np.array(self.head.position)
    object_direction = object_direction / np.linalg.norm(
        object_direction)

    sign = np.ones(object_direction.shape[0])
    for i in range(sign.size):
        # is the object above or below the head?
        if object_direction[i][1] < 0:
            sign[i] = -1

    # calculate COSINE angle of object relative to head (positive if
        above, negative if below)
    # cosine_angle is of size [num_objects,]
    cosine_angle = sign * np.arccos( \
        np.dot(object_direction, np.array([-1.0, 0.0])))

    # differnce in angle between the head angle and sine_angle of
        head
    relative_angle = cosine_angle + self.head.angle
```

```
    # relative_angle should be within [-np.pi, np.pi]
    for i in range(relative_angle.shape[0]):
        if relative_angle[i] < -np.pi:
            k = 1
            while relative_angle[i] < (k + 1) * -np.pi:
                k += 1
            relative_angle[i] = relative_angle[i] + 2 * np.pi * ((k +
                 1) // 2)

        elif relative_angle[i] > np.pi:
            k = 1
            while relative_angle[i] > (k + 1) * np.pi:
                k += 1
            relative_angle[i] = relative_angle[i] - 2 * np.pi * ((k +
                 1) // 2)

    return relative_angle

def _get_angular_velocity(self, prev_ang, current_ang):
    angle_velocity = current_ang - prev_ang
    angle_speed = np.absolute(angle_velocity)
    for i in range(angle_velocity.size):
        if angle_speed[i] > np.pi:
            angle_velocity[i] = 2 * np.pi - angle_velocity[i]
        elif angle_speed[i] < -np.pi:
            angle_velocity[i] = 2 * np.pi + angle_velocity[i]
        else:
            pass
    return angle_velocity

"""
Defining Reward Functions
"""
def _assign_reward_func(self, reward_code, max_offset = None):
    self.prev_angle = self._get_retinal(self.objects_position)
```

```
    if "motion_parallax" in reward_code:
        self.reward_function = self._motion_parallax
    elif "retinal_stabilization" in reward_code:
        self.reward_function = self._retinal_stabilization
    elif "fifty_fifty" in reward_code:
        self.reward_function = self._fifty_fifty
    else:
        raise ValueError("Unknown reward_code")

def _motion_parallax(self):
    current_angle = self._get_retinal(self.objects_position)

    parallax_velocities = \
        self._get_angular_velocity(current_angle, self.prev_angle)

    reward = 0
    # sum of motion parallax magnitudes
    for i in range(parallax_velocities.size):
        for j in range(i, parallax_velocities.size):
            reward += np.abs(parallax_velocities[i] -
                parallax_velocities[j])
        # reward += parallax_velocities[i]
    return reward

def _retinal_stabilization(self):
    reward = 0
    current_angle = self._get_retinal(self.objects_position)
    relative_speeds = \
        np.absolute(self._get_angular_velocity(current_angle, self.
            prev_angle))
    reward -= np.sum(relative_speeds)
    return reward

def _fifty_fifty(self):
    reward = 0
    reward += self._retinal_stabilization()
```

```
        reward += self._motion_parallax()
        return reward

    def _get_obs(self):
        # (self.head{relative}, self.joints -> obs) operation
        obs = np.array(self.head.position) - np.array(self.body.position)
        obs = np.concatenate((obs, self.head.angle), axis = None)
        for i in range(len(self.joints)):
            obs = np.concatenate((obs, self.joints[i].angle), axis = None
                )
            obs = np.concatenate((obs, self.joints[i].speed), axis = None
                )
        obs = np.concatenate((obs, self.body.position[0]), axis = None)
        obs = np.float32(obs)
        assert self.observation_space.contains(obs)
        return obs

    def step(self, action):
        self.prev_angle = self._get_retinal(self.objects_position)
        # alter object
        self.objects_position += self.objects_velocity
        return super().step(action)

    def render(self, mode = "human"):
        from gym.envs.classic_control import rendering
        if self.viewer is None:
            self.render_objects_list = None
            self.render_objects_translate_list = None

        super().render(mode)
        # initialize object rendering pointers
        if self.render_objects_list is None:
            self.render_objects_list = []
            self.render_objects_translate_list = []
            for i in range(self.objects_position.shape[0]):
                object_render_instance = rendering.make_circle( \
```

```
            radius=0.6,
            res=30,
            filled=True)
        object_render_instance_translate = rendering.Transform(
            translation = VIEWPORT_SCALE * \
                (self.objects_position[i] - self.camera_trans),
            rotation = 0.0,
            scale = VIEWPORT_SCALE * np.ones(2)
        )
        object_render_instance.add_attr(
            object_render_instance_translate)
        object_render_instance.set_color(0.0, 1.0, 0.0)
        self.render_objects_list.append(object_render_instance)
        self.render_objects_translate_list.append(
            object_render_instance_translate)
        self.viewer.add_geom(object_render_instance)

# update object translation
new_object_translate = VIEWPORT_SCALE * self.objects_position -
    self.camera_trans
for i in range(self.objects_position.shape[0]):
    self.render_objects_translate_list[i].set_translation( \
        new_object_translate[i][0], new_object_translate[i][1])

return self.viewer.render(return_rgb_array = mode == "rgb_array")
```