Contents

1 /

1.1 README.md

```
# Controller-Agnostic Orchestration Architecture (COACH)
## A Framework for Training Semi-Autonomous Agents and Directors Leveraging Existing Simulations

COACH is a system to turn (PettingZoo)[https://pettingzoo.farama.org/index.html] (PZ) compatible multiagent Gymnasium environments into agent orchestration and planning problems, using a director/actor hierarchical framework. Given any PZ environment and a set of agents trained to execute actions on that enviornemnt given asynchronous direction in terms of a __plan__," the COACH repo provides a wrapper to turn the problem into a planning simulation and a wrapper to turn such planning simulations into PZ compatible orchestration environments for raining director agents. In addition, it provides a prefab agents and an interactive plan visualization system using DASH.

This library can be used as is, but has been deisgned as a starting point for interactive planning problems.

<img src="examples/Multipolicy/Interface.png">

## Termanology:
A lot of the terminology around RL becomes contested when you move into the multiagent setting. For this library we will nail down the following conventions, with the explicit understand that they are not universal. We have tried, as much as possible, to keep the language as close to any standard we could find as possible.

* Simulation - A (multiagent) simulation environment conforming to the (PettingZoo interface)[https://pettingzoo.farama.org/index.html]
* Agent - A piece of code that takes observations and returns actions.
* Actor - An agent assigned to a specific role in multiagent simulation, and equiped with an instruction set (interface) for directing the actor. Actors are "interactive" or "directable" agents. Importantly, actors may be RL policy based, have multiple policies, use good-old-fashioned-ai, or be generated using any other method.
* Course Of Action (COA) - A set of directions for a particular actor, tagged to specific times in the language of the agents interface.
* Plan - A set of courses of action, one for each role in the simulation.
* COACH Environment - An environment for planning problems: Given a simulation and a set of actors filling the required roles, a directing agent generates a course of action for each agent to follow. The simulation is then run forward a number of steps (fixed, agent depenant, or agent dependant with blackout conditions), and the courses of action may be updated.
* Director - An agent that takes observations recorded by actors, and generates a courses of action.
* Trajectory - Action/observation/rewards for all agents over a period of time.


# Getting started

For a good introduction, take a look at the examples provided in the examples folder. They give a demonstration of how to train and deploy a multi-policy actor with a director choosing which policy to use at a given time.

## Examples:

### Multiploicy Director

In this example, we use StableBaselines3 and Supersuit to train two policies on the Waterworld environment from PettingZoo. One we train in a sparse food environment as an "explorer" agent, the other we train in a dense food and dense poison environment as a "dense" avoider agent. Finally, we train a director to select the policies each agent should use for the next 10 turns given their current observations.

In this example we see how to train policies on a PettingZoo environment, use those policies and that PettingZoo environment to establish a planning problem using the coach environment. We then set up a communications schedule to allow communication every 10 turns and train the director agent to select between the policies.

Finally, we construct a Dash app to display our solution.

This example uses all most entirely default functionality. For more advanced functionality see the MAInspection example.

### Multiagent Inspection

This example shows how to solve a simple satelite inspection problem in two steps:
* First we train an agent not to solve the problem, but to use the environment to train an agent to go from one waypoint to another.
* After this "waypointer"
agent is trained the director is then trained to select waypoints that easily solve the problem.

This example also demonstrates how to create custom agents, a custom coach environment, custom
```

~~~
trajectory classes and custom trajectory visualizations.

## Overview of Environment Files:

* `coach.py` - Contains the class that wraps a PZ env, turning it into a planning environment.
```
COACHEnvironemnt(
        env_creator: callable,
        parameter_generator:BaseParameterGenerator,
        agents=dict(),
        fill_random=True,
        comm_schedule:Timeline=None,
        seed=6234235,
        TrajectoryClass = Trajectory,
)
```
* * `env_creator` - A function that creates a parallel PettingZoo env.
* * `parameter_generator` - A parameter generator object. Simulations call the parameter generator for samples. The BaseParameterGenerator simply returns the same parameters each time, the RandomParameterGenerator returns the same parameters but different seeds.
* * `agents` - A dictionary of `role:agent` pairs. If not specified the DefaultActor agent will be used.
* * `fill_random` - If `True` any agent roles not specified above will be set to the RandomActor agent.
* * `comm_schedule` - A `utilities.planning.CommunicationsSchedule` object that determines contact times and blackout times for director/agent communication.
* * `seed` - Random seed.
* * `TrajectoryClass` - Trajectories save agent history. By default all actions and step returns are saved, but if other information is required a different class can be passed.

* `env.py` - PettingZoo env wrapping COACHEnvironemnt for training director agents.
```
COACH_PettingZoo(env_creator: callable, COACHEnvClass: COACHEnvironment)
```
* * `env_creator` - A function that creates a parallel PettingZoo env.
* * `COACHEnvClass` - COACHEnvironemnt class that will be wrapped.

## Overview of Auxiliary Files
* `agents.py` - Library of agents, sample agent classes include RandomActor, DefaultActor and SB_PPOPoliciesActor
* `params.py` - Standard MarcoPolo params functionality for evolving parameters
* `utilities.iotools.py` - Tools for saving and reloading
* `utilities.planning.py` - Holds timeline classes specifically for COACH, including `CommunicationsSchedule` and `COA`
* `utilities.PZParams.py` - Parameter file for environments from the PettingZoo library
* `utilities.PZWrapper.py` - Wrapper for environments from the PettingZoo library
* `utilities.timelines.py` - Timeline classes
* `utilities.tools.py` - Miscellaneous tools used by parts of the coach env.

* `DASH` - Interactive Course Of Action viewer. See Multipolicy Example for more.
* `examples` - Examples of how to use the coach environment.
~~~

1.2 agents.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import sys

from utilities.timelines import TimelineEvent
from utilities.planning import COA
from numpy.random import PCG64DXSM, Generator
import numpy as np
import copy
import logging
from gymnasium.spaces import Box

logger = logging.getLogger(__name__)

from collections import OrderedDict
from stable_baselines3 import PPO
import sys, inspect

##################################################
##
## Interfaces
##
##################################################

class ActionBox(Box):
    def __init__(self, low=[], high=[], shape=[], default=[], description=None):
        if len(low)>0:
            super().__init__(low=low, high=high, shape=shape)
        else:
            self.low = []
            self.high = []

        self.description = description

        if len(default) == len(low):
            self.default = default
        elif len(default) == 0:
            default = []
            for l, h in zip(low, high):
                # If the bounds are finite, take the average.
                if np.isfinite(l) and np.isfinite(h):
                    default.append((l+h)/2)
                # If one is infinte, we're going to assume the other is the default value
                elif np.isinf(l) and np.isfinite(h):
                    default.append(h)
                elif np.isfinite(l) and np.isinf(h):
                    default.append(l)
                ## Otherwise, assume 0
                else:
                    default.append(0)

            self.default = default
        else:
            raise Exception("Action Box has default values of length different than box.")

    def __repr__(self) -> str:
        return f"{self.description}: low {self.low}, high {self.high}"

    def items(self):
        actions = []
        for i in range(len(self.low)):
            actions.append(
                ActionBox(
                    low = np.array([self.low[i]]),
                    high = np.array([self.high[i]]),
                    shape = (1,),
                    description = self.description[i]
                )
            )

        return actions

    def equals(self, other):
        # All defualt interfaces are the same
        if not (type(other) is type(self)):
            return False
        if not all([a==b for a,b in zip(self.low, other.low)]):
            return False
        if not all([a==b for a,b in zip(self.high, other.high)]):
            return False
        if not all([a==b for a,b in zip(self.shape, other.shape)]):
            return False
        if not all([a==b for a,b in zip(self.default, other.default)]):
            return False

        return True


class TrivialInterface:
    def __init__(self, role, env):
        self.env_observation_space = copy.deepcopy(env.observation_space(role))
        self.env_action_space = copy.deepcopy(env.action_space(role))

        self.action_space = self.env_action_space
        self.observation_space = self.env_observation_space

        self.action_dictionary = {"default": ActionBox()}

        self.name = self.__class__.__name__

    def action_names(self):
        return list(self.action_dictionary.keys())

    def get_action_descriptions(self):
        # Return human readable description of the sceion parameters.
        return self.action_description

    def get_action_dictionary(self):
        # Return the dctionary of possible actions with their parameter
        # spaces
        return self.action_dictionary

    # We may want to verify and error check that an interface matches an expected
    # interface, espeaclly if the interface takes parameters.
    def equals(self, other):
        # All defualt interfaces are the same
        if type(other) is type(self):
            return True
        return False


class DefaultInterface(TrivialInterface):
    def __init__(self, role, env, max_action_len=None):
        super().__init__(role, env)

        if not max_action_len:
            max_action_len = np.inf

        # The burn is "legnth" + "action space"
        burn = ActionBox(
            low=np.concatenate([(0,), self.env_action_space.low]),
            high=np.concatenate([(max_action_len,), self.env_action_space.high]),
            shape=(1 + np.prod(self.env_action_space.shape),),
            description = ["Length"] + ["Unknown"] * len(self.env_action_space.low)
        )

        self.action_dictionary = {"burn": burn}

    def equals(self, other):
        # All defualt interfaces are the same
        if type(other) is type(self):
            return self.action_dictionary["burn"].equals(self.action_dictionary["burn"])

        return False


class RandomActionInterface(TrivialInterface):
    def __init__(self, role, env):
        super().__init__(role, env)

        self.action_dictionary = {"random": ActionBox()}


class SBPolicyInterface(TrivialInterface):
    def __init__(self, role, env, n_policies, max_action_len):
        super().__init__(role, env)

        one_hot_low = [0]*n_policies
        one_hot_high = [1]*n_policies
        desc = ["Action Length"] + [f"policy_{i}" for i in range(n_policies)]

        policies = ActionBox(
            low=np.array([0] + one_hot_low),
            high=np.array([max_action_len] + one_hot_high),
            shape=(len(one_hot_low) + 1,),
            description=desc
        )

        self.action_dictionary = {"Policies": policies}

    def equals(self, other):
        if type(other) is type(self):
            return self.action_dictionary["Policies"].equals(other.action_dictionary["Policies"])
        return False


##################################################
##
## Actors
##
##################################################


class BasicActor:
    InterfaceType = TrivialInterface

    def __init__(self, role, reference_interface:TrivialInterface = None):
        self.role = role
        self.reference_interface = reference_interface

    def action_names(self):
        return list(self.interface.action_dictionary.keys())

    def get_action_dictionary(self):
        # Return the dctionary of possible actions with their parameter
        # spaces
        return self.interface.action_dictionary

    def get_action_descriptions(self):
        # Return human readable description of the sceion parameters.
        return self.interface.action_description

    def get_action(self, obs, t, mean_mode=False):
        return self.none_action, {"acting": False, "coa_done": True}

    def process_observations(self, obs, t):
        # Agents may communicate different information than what they observe
        return obs

    def update_coa(self, coa):
        self.coa.add_timeline_to(coa, allow_dup_labels=False)

    def reset(self, env):
        self.env = env
        self.interface = BasicActor.InterfaceType(self.role, env)
        if self.reference_interface is not None:
            if not self.interface.equals(self.reference_interface):
                raise Exception("Actor interface does not match reference interface.")

        self.coa = COA()

        self.none_action = np.zeros(self.interface.action_space.shape)


class DefaultActor(BasicActor):
    InterfaceType = DefaultInterface

    def __init__(self, role, max_action_len=None):
        super().__init__(role)
        self.max_action_len = max_action_len

    def __str__(self):
        return f"DefaultActor: {self.role}"

    def __repr__(self):
        return self.__str__()

    def get_action(self, obs, t, mean_mode=False):
        if t in self.coa.timeline.keys():
            e = self.coa.get(time = t, label = "burn")
            self.burn_time = np.floor(e.parameters[0])
            self.current_burn = np.array(e.parameters[1:]).reshape(
                self.interface.action_space.shape
            )

        time_stamps = list(self.coa.timeline.keys())
        if len(time_stamps) == 0:
            coa_done = True
        else:
            last_t = max(time_stamps)
            last_duration = self.coa.get(time = last_t, label= "burn").parameters[0]

            if t >= last_t + last_duration:
                coa_done = True
            else:
                coa_done = False

        if self.burn_time > 0:
            self.burn_time -= 1
            return self.current_burn, {"acting": True, "coa_done": coa_done}
        else:
            return self.none_action, {"acting": False, "coa_done": coa_done}

    def reset(self, env=None):
        super().reset(env)

        self.interface = DefaultActor.InterfaceType(self.role, env, max_action_len=self.max_action_len)

        self.current_burn = None
        self.burn_time = 0


class RandomActor(BasicActor):
    InterfaceType = RandomActionInterface

    def __init__(self, role, seed=23143):
        super().__init__(role)
        self.seed = seed

    def __str__(self):
        return f"RandomActor: {self.role}"

    def __repr__(self):
        return self.__str__()

    def get_action(self, action, parameters):
        r = self.np_random.uniform(
            low=self.interface.action_space.low, high=self.interface.action_space.high
        )

        return r, {"acting": True, "coa_done": False}

    def reset(self, env):
        super().reset(env)

        self.interface = RandomActor.InterfaceType(self.role, env)
        self.np_random = Generator(PCG64DXSM(seed=self.seed))


class SB_PPOPoliciesActor(BasicActor):
    InterfaceType = SBPolicyInterface

    def __init__(self, role, policy_paths: dict, max_action_len: int, interface:SBPolicyInterface=None):
        super().__init__(role)

        self.max_action_len = max_action_len

        self.policies = dict()
        for policy, model_path in policy_paths.items():
            # For some reason SB doesn't want to zip on the end
            if model_path[-3:] == "zip":
                model_path = model_path.split(".")[0]
            self.policies[policy] = PPO.load(model_path)

        self.policy_names = list(self.policies.keys())

    def __str__(self):
        return f"PPOPoliciesActor: {self.role}, Policies: {self.policies.keys()}"

    def get_action(self, obs, t, mean_mode=False):
        bad_command = False
        acting = False
        coa_done = False

        if t in self.coa.timeline.keys():
            ## Start new policy
            e = self.coa.get(time = t, label = "Policies")
            self.time_remaining = e.parameters[0]*self.max_action_len
            logger.debug("Agent: start new policy %s", e) # DEBUG
            logger.debug("policy: %s, timeleft: %s",
                 self.policy_names[int(np.argmax(e.parameters[1:]))], self.time_remaining)  # DEBUG

            self.current_policy = self.policy_names[int(np.argmax(e.parameters[1:]))]

        # self.next_waypoint_abs = np.array([7.08763559,  -25.61246231, -167.51736098])
        if (self.current_policy is not None) and (self.time_remaining > 0):
            acting = True
            action, _ = self.policies[self.current_policy].predict(obs, deterministic=True)
            self.time_remaining += -1
        else:
            action = copy.copy(self.none_action)

        if self.time_remaining == 0:
            coa_done = True

        return action, {
            "acting": acting,
            "coa_done": coa_done,
            "bad_command": bad_command,
        }

    def reset(self, env):
        super().reset(env)
        self.time_remaining = -1
        self.current_policy = None

        self.interface = SB_PPOPoliciesActor.InterfaceType(
            self.role,
            env,
            n_policies=len(self.policy_names),
            max_action_len=self.max_action_len
        )

        if self.reference_interface is not None:
            if not self.interface.equals(self.reference_interface):
                raise Exception("Actor interface does not match reference interface.")


classes = [
    cls_obj
    for cls_name, cls_obj in inspect.getmembers(sys.modules[__name__])
    if inspect.isclass(cls_obj) and cls_obj.__module__ == __name__
]

Interfaces = {}
for c in classes:
    # BasicActor
    if issubclass(c, TrivialInterface):
        Interfaces[c] = []

Agents = {}
for c in classes:
    # BasicActor
    if issubclass(c, BasicActor):
        Interfaces[c.InterfaceType].append(c)
        Agents[c.__name__] = c
```

1.3 coach.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from typing import Any
import numpy as np
import copy
import logging
logger = logging.getLogger(__name__)
import argparse

import matplotlib.pyplot as plt
from numpy.random import PCG64DXSM, Generator

import agents as agents
from agents import (
    BasicActor,
    DefaultActor,
    RandomActor
)

from utilities.timelines import Timeline, Timelines
from utilities.planning import (
    COA,
    CommunicationSchedule,
    BaseParameterGenerator,
    SeededParameterGenerator,
    State,
    Trajectory,
    Telemetry,
)

from matplotlib.backends.backend_agg import (
    FigureCanvasAgg as FigureCanvas,  # type: ignore[import]
)

# %%

def get_env_class(args: argparse.Namespace):
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env objects
    """

    return COACHEnvironment

class COACHEnvironment:
    agent_selection = agents

    def __init__(
        self,
        env_creator: callable,
        parameter_generator:BaseParameterGenerator,
        agents=dict(),
        fill_random=True,
        comm_schedule:Timeline=None,
        seed=6234235,
        TrajectoryClass = Trajectory,
    ):
        self.env_creator = env_creator
        self.env = env_creator()
        self.parameter_generator = parameter_generator

        self.stored_agents = agents
        self.fill_random = fill_random
        self.seed = seed
        self.TrajectoryClass = TrajectoryClass

        if comm_schedule is None:
            comm_schedule = Timelines(labels=["blackout", "checkins"])
            comm_schedule.checkins.add_event(time=0, )
        self.comm_schedule = comm_schedule

        self.rendering=False

        self.reset()

        for k in agents.keys():
            if not k in self.env.possible_agents:
                raise Exception(
                    "Passed actor name is not in enviroments possible actors"
                )

    def __deepcopy__(self, memo):
        tmp_agents = {k: copy.deepcopy(a) for k, a in self.stored_agents.items()}

        tmp = self.__class__(
            self.env_creator,
            self.parameter_generator,
            tmp_agents,
            copy.deepcopy(self.fill_random),
            copy.deepcopy(self.comm_schedule),
            copy.deepcopy(self.seed),
        )

        tmp.setstate(self.state)
        return tmp

    def _fillagents(self, fill_random=None):
        if not fill_random:
            fill_random = self.fill_random

        self.action_spaces = {}
        self.observation_spaces = {}

        self.agents = dict()

        for agt in self.env.possible_agents:
            if agt in self.stored_agents.keys():
                self.agents[agt] = self.stored_agents[agt]
            else:
                if fill_random:
                    self.agents[agt] = RandomActor(agt)
                else:
                    self.agents[agt] = DefaultActor(agt)

            self.agents[agt].reset(self.env)
            self.action_spaces[agt] = self.agents[agt].interface.action_dictionary
            self.observation_spaces[agt] = self.agents[agt].interface.observation_space

    def possible_models(self):
        default = BasicActor, DefaultActor, RandomActor
        return {role: default for role in self.env.possible_agents}

    def set_coa(self, coas):
        for agt, coa in coas.items():
            self.agents[agt].update_coa(coa)

    def get_coa(self):
        return {role: agt.coa for role, agt in self.agents.items()}

    def simulate(
        self,
        coas=None,
        parameters=None,
        agents=None,
        state=None,
        time_steps=None,
        comm_steps=None,
        render=False
    ):
        if comm_steps and time_steps:
            logger.info(
                "Only one of `time_steps` and `comm_steps` can be defined at a time"
            )
            return

        tmp_env = copy.deepcopy(self)
        if agents is not None:
            tmp_env.stored_agents = agents

        if render:
            tmp_env.start_rendering()

        # Set new env params:
        if parameters:
            tmp_env.augment(parameters)

        tmp_env.reset()
        tmp_agts = tmp_env.agents

        if state:
            if parameters:
                logger.info(
                    "Note: State parameters will overwrite passed parameters."
                )
            tmp_env.setstate(state)

        # Set new COA for agents
        if coas:
            for agt, coa in coas.items():
                tmp_agts[agt].update_coa(coa)

        # Run simulation, recording trajectory
        if time_steps:
            tmp_env.step_env(steps=time_steps)

        elif comm_steps:
            for _ in range(comm_steps):
                tmp_env.step()

        else:
            ended = False
            while not ended:
                tmp_env.step()
                terms = list(tmp_env.step_return[-1][2].values())
                truncs = list(tmp_env.step_return[-1][3].values())
                if all([a or b for a, b in zip(terms, truncs)]):
                    ended = True

        # Since trajectory starts at 0'th step
        trajectory = tmp_env.state.trajectory

        del tmp_env
        del tmp_agts

        return trajectory

    def start_rendering(self, steps_per_frame = 10):
        self.steps_per_frame = steps_per_frame
        self.rendering=True

    def stop_rendering(self):
        self.rendering=False

    def augment(self, parameters):
        self.parameter_generator = parameters

    def reset(self):
        seed, parameters = self.parameter_generator.sample()
        self.setup_env(seed, parameters)
        self.state.cummulative_rews = {agt: 0 for agt in self.env.possible_agents}

    def setup_env(self, seed, parameters):
        self.env.augment(parameters)
        self.env.seed(int(seed))
        obs, info = self.env.reset()
        self.step_return = [(obs, None, None, None, info)]

        # Fill agents based on reset env
        self._fillagents()

        ## Reset agents, get their initial conditions.
        self.agent_info = [
            {role: agt.reset(env=self.env) for role, agt in self.agents.items()}
        ]

        initial_frame = None
        if self.rendering:
            initial_frame = self.env.render()

        traj = self.TrajectoryClass(
            self.env,
            initial_return=self.step_return[0],
            intial_frame=initial_frame
        )

        self.state = State(parameters, seed, traj, current_t=0)
        self.alive = {agt:True for agt in self.agents.keys()}


    def setstate(self, state: State):
        self.setup_env(state.seed, state.parameters)

        for step in state.trajectory[1:]:
            step_return = self.env.step(step["action"])
            self.step_return.append(step_return)

            self.state.trajectory.add(
                env=self.env,
                action=step["action"],
                agent_info=step["agent_info"],
                step_return=step_return,
            )

        self.state.current_t = len(state.trajectory) - 1


    def last(self):
        return self.step_return[-1]

    def step(self, coas=None):
        ## This is the step through the coa
        if coas:
            self.set_coa(coas)

        next_comm, _ = self.comm_schedule.checkins.next_event(self.state.current_t)
        if next_comm:
            next_comm = next_comm - self.state.current_t

        logger.debug("current_t: %s, next_com: %s", self.state.current_t, next_comm)    # DEBUG
        self.step_env(steps=next_comm)

        return self.step_return[-1], self.step_end

    def step_env(self, steps=1, coas=None):
        self.step_end = {
            "coa_done": [],
            "term_or_trunc": False,
            "steps_reached": False
        }

        if coas:
            self.set_coa(coas)

        t = self.state.current_t-1
        running = True

        while running:
            t += 1
            logger.debug("Env Step: %s", t)  # DEBUG

            action = dict()
            agent_info = dict()

            for role, agt in self.agents.items():
                if self.alive[role]:
                    action[role], agent_info[role] = agt.get_action(
                        self.step_return[-1][0][role], t
                    )
                else:
                    action[role] = None
                    agent_info[role] = "not_alive"
            logger.debug("Env Action: %s", action)    # DEBUG

            obs, rwds, terms, truncs, info = self.env.step(action)

            frame = None
            if self.rendering:
                if t % self.steps_per_frame == 0:
                    frame = self.env.render()

            returns = (
                {role: self.agents[role].process_observations(o, t) for role, o in obs.items()},
                rwds, terms, truncs, info
            )

            for role, rwd in returns[1].items():
                self.state.cummulative_rews[role] += rwd

            self.agent_info.append(agent_info)
            self.step_return.append(returns)
            self.state.trajectory.add(
                env=self.env,
                action=action,
                step_return=returns,
                agent_info=agent_info,
                frame=frame)

            self.alive = {agt: not (returns[2][agt] or returns[3][agt]) for agt in self.agents.keys()}

            break_for_new_COA = False

            if self.comm_schedule.ALLOW_AGENT_BREAK:
                for role in self.agents.keys():
                    if self.comm_schedule.blackouts.get(t, role) is not None:
                        if self.agent_info[-1][role]["coa_done"]:
                            self.step_end["coa_done"].append(role)
                            break_for_new_COA = True
                            running = False

            if not any(self.alive.values()):
                self.step_end["term_or_trunc"] =  True
                running = False

            if steps:
                if t >= (self.state.current_t + steps):
                    running = False

        if steps:
            if t == self.state.current_t + steps - 1:
                # We reached the end, even if other things would have terminated it
                self.step_end["steps_reached"] = True

        logger.debug("Env step end: %s", self.step_end)  # DEBUG
        self.state.current_t = t + 1

#####################################
# COA Level Interface:
#####################################

    def get_traj_from_coas(self, coas, params=None, from_state=None, render=False):
        traj = self.simulate(
            coas,
            parameters=params,
            state=from_state,
            render=render
            )
        return traj

    def get_traj_from_plan(self,
        plan,
        params=None,
        from_state=None,
        render=False
    ):
        coas = {role: coa for role, coa in plan.items()}

        agent_dict = dict()
        for role, model_class in plan.model_classes.items():
            agent_dict[role] = model_class(role, **plan.model_params[role])

        traj = self.simulate(
            coas,
            parameters=params,
            agents=agent_dict,
            state=from_state,
            render=render
            )

        return traj

    def trajectory_telemetry(
        self,
        traj,
        components=None,
        labels=None,
        players=None,
        ao="observations",
        plan=None,
        env=None,
        cmap=None
    ):

        if players is None:
            players = list(traj.step_returns[0][0].keys())

        if cmap is None:
            default_cycle = plt.rcParams['axes.prop_cycle'].by_key()['color']
            cmap = lambda i: default_cycle[i%len(default_cycle)]

        traj_len = len(traj)

        rewards = Telemetry(
            name="Reward",
            title="Reward",
            xscale = np.array(range(traj_len)),
            xlabel = "Time Step",
            ylabel = "Reward",
        )

        rewards.set_data(np.zeros([len(players),traj_len]))

        for i, p in enumerate(players):
            rewards.data_labels[i] = p
            rewards.colors[i] = cmap(i)

            for j, t in enumerate(traj.step_returns[1:]): # No intial reward
                rewards.data[i, j] = t[1][p]

        return [rewards]

    # Note: This should be able to handle plotting
    # without having to have the origional env spun up.
    def plot_trajectory_component(
            self,
            traj,
            components=[0],
            labels=None,
            players=None,
            ao="observations",
            plan=None,
            env=None,
            render_mode="rgb_array" # Union["rgb_array", "matplotlib", "plotly"]
        ):

        if not players:
            players = list(traj.step_returns[0][0].keys())

        if not type(players) is list:
            players = [players]

        create_labels = False
        if not labels:
            labels = dict()
            create_labels = True

        if ao == "observations":
            data = {}
            all_components = False
            if not components:
                all_components = True

            for p in players:
                action_list = []
                for t in traj.step_returns:
                    t = t[0]
                    if all_components:
                        components = list(range(len(np.array(t[p]).reshape(-1))))

                    action_list.append(t[p][components])

                data[p] = np.array(action_list)
                if create_labels:
                    labels[p] = [f"{p}: {c}" for c in components]

        if ao == "actions":
            data = {}
            all_components = False
            if not components:
                all_components = True

            for p in players:
                action_list = []
                for t in traj.actions[1:]:
                    if all_components:
                        components = list(range(len(np.array(t[p]).reshape(-1))))

                    action_list.append(t[p][components])

                data[p] = np.array(action_list)
                if create_labels:
                    labels[p] = [f"{p}: {c}" for c in components]


        if ao == "rewards":
            components = [0]
            data = traj.step_returns
            xs = np.zeros([len(players), len(components), len(data)-1])

            for j, t in enumerate(data):
                if j > 0: # The initial state has no reward
                    for i, p in enumerate(players):
                        xs[i, :, j] = t[1][p]

        n_axes = sum([s.shape[1] for s in data.values()])
        # Setup Figures
        w = n_axes // 2
        w_m = w + n_axes % 2

        f = plt.gcf()
        axes = []
        for i in range(w):
            for j in range(2):
                axes.append(plt.subplot2grid((w_m, 4), (i, 2 * j), 1, 2))

        if n_axes % 2:
            axes.append(plt.subplot2grid((w + 1, 4), (w, 1), 1, 2))


        # Get Data

        idx = 0
        for p in players:
            for j in range(data[p].shape[1]):
                axes[idx].plot(data[p][:,j])
                axes[idx].title.set_text(labels[p][j])
                idx += 1

        # l5 = axes[0].legend(
        #     bbox_to_anchor=(0.5, -0.05), loc="lower center", bbox_transform=f.transFigure
        # )

        plt.subplots_adjust(left=0.1, right=0.9, hspace=0.3, wspace=0.5)

        fig = plt.gcf()
        if render_mode == "matplotlib":
            return fig

        canvas = FigureCanvas(fig)
        canvas.draw()

        data = np.frombuffer(fig.canvas.tostring_rgb(), dtype=np.uint8)
        data = data.reshape(fig.canvas.get_width_height()[::-1] + (3,))
        plt.close(fig)

        if render_mode == "rgb_array":
            return data

        if render_mode == "plotly":
            import plotly.express as px
            fig = px.imshow(data)
            return fig





#################################################################
## Example Usage
#################################################################

if __name__ == "__main__":
    from examples.MAInspection.env import MultInspect

    ## Environemntal Parameters
    env_params = {"_OBS_REWARD":.01, "num_deputes": 3}

    # Actor Parameters
    COACH_params = {
        "Agents": {
            "player_0": {
                "class_name":"DefaultActor"
                # params: {"parameter": "value"}    # If the agent has setup parameters
                },
            "player_1": {"class_name":"RandomActor"},
            "player_2": {"class_name":"DefaultActor"},
        }
    }

    # Create actors from parameters
    agent_dict = dict()
    if "Agents" in COACH_params.keys():
        for role, agent in COACH_params["Agents"].items():
            agent_class = COACHEnvironment.agent_selection.Agents[agent["class_name"]]
            agent_dict[role] = agent_class(role, **agent.get("params", dict()))

    # Example communication scheudle.
    comms = CommunicationSchedule.repeating(checkin_frequency=10)

    # Wrap parameters in parameter generator
    parameter_generator = SeededParameterGenerator(23423, env_params)

    env = COACHEnvironment(
        env_creator = MultInspect,
        default_parameters = parameter_generator,
        comm_schedule = comms,
        fill_random = True,
        agents = agent_dict
    )

    for agt in env.agents.values():
        print(agt.interface)
    input()

    for i in range(10):
        print(env.step())
        for role, actions in env.action_spaces.items():
            print(f"{role}\t {actions}")
        input("Press Enter For Next Observation")
```

1.4 directors.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from env import COACH_PettingZoo
from stable_baselines3 import PPO

import copy

import logging
logger = logging.getLogger(__name__)


class SB3_PPO_Director:
    def __init__(self, env_creator, COACHEnvClass, params, model_path):
        self.env_creator = env_creator
        self.COACHEnvClass = COACHEnvClass
        self.params = params

        self.env = COACH_PettingZoo(env_creator=env_creator, COACHEnvClass=COACHEnvClass)
        self.env.augment(params)

        if model_path.endswith(".zip"):
            model_path = model_path[:-4]

        self.policy = PPO.load(model_path)

    def generate_coas(self, params=None):
        if not params:
            params = self.params

        self.env.augment(params)

        obs, info = self.env.reset()
        running = True

        while running:
            act = {"director": self.policy.predict(obs["director"], deterministic=False)[0]}

            # logger.debug("Obs From Director: %s", obs["director"]) # DEBUG
            # logger.debug("Action From Director: %s", act) # DEBUG

            obs, reward, term, trunc, info = self.env.step(act)

            if all([a or b for a,b in zip(term.values(), trunc.values())]):
                running = False

        coas = dict()
        for role, agent in self.env.coa_env.agents.items():
            coas[role] = agent.coa

        traj = copy.deepcopy(self.env.coa_env.state.trajectory)
        return coas, traj
```

1.5 env.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import sys

from gymnasium.spaces import Box
from pettingzoo.utils.env import ParallelEnv
import gymnasium as gym
from typing import Any
import numpy as np
import argparse

from coach import (
    SeededParameterGenerator,
    BaseParameterGenerator,
)

from utilities.planning import COA

from coach import COACHEnvironment, CommunicationSchedule

import logging
logger = logging.getLogger(__name__)


def get_env_class(args: argparse.Namespace):
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env objects
    """
    return COACH_PettingZoo


class COACH_PettingZoo(gym.Wrapper,ParallelEnv):
    def __init__(self, env_creator, COACHEnvClass):
        self.env_creator = env_creator
        self.COACHEnvClass = COACHEnvClass
        self.fake_render_mode = "rgb_array"

    def _setup(self, params):
        self.current_params = params
        self.COACH_params = COACH_params = params["COACH_params"]
        self.env_params = env_params = params["env_params"]

        # Set up parameter generator
        if COACH_params["stochastic"]:
            self.parameter_generator = SeededParameterGenerator(23423, env_params)
        else:
            self.parameter_generator = BaseParameterGenerator(23423, env_params)

        # Set up comm schedule
        if "FIXED_STEPS_PER_COM" in COACH_params.keys():
            schedule_param = COACH_params["FIXED_STEPS_PER_COM"]
            self.comm_schedule = CommunicationSchedule.repeating(**schedule_param)
        else:
            logger.info("Communication Schedule Is Non-repeating")
            self.comm_schedule = CommunicationSchedule(length=0)

        self.ACTION_PADDING = COACH_params.get("ACTION_PADDING", 0)
        self.MIN_NEXT_ACTION_TIME = COACH_params.get("MIN_NEXT_ACTION_TIME", 1)
        self.MAX_NEXT_ACTION_TIME = COACH_params.get("MAX_NEXT_ACTION_TIME", np.inf)

        # Setup Agents
        agent_dict = dict()

        if "Agents" in self.COACH_params.keys():
            for role, agent in self.COACH_params["Agents"].items():
                logger.info(f"############ {role} {agent}")
                agent_class = self.COACHEnvClass.agent_selection.Agents[agent["class_name"]]
                agent_dict[role] = agent_class(role, **agent.get("params", dict()))

        # Create COA Env
        self.coa_env = self.COACHEnvClass(
            env_creator=self.env_creator,
            parameter_generator=self.parameter_generator,
            agents=agent_dict,
            fill_random=False,
            comm_schedule=self.comm_schedule,
            seed=COACH_params["seed"],
        )

        self.coa_env.augment(self.parameter_generator)
        self.coa_env.reset()

        self.stochastic = COACH_params["stochastic"]

        self.players = list(self.coa_env.agents.keys())
        self.player_actions = self.coa_env.action_spaces

        self.possible_agents = ["director"]
        self.agents = ["director"]

        # Action spaces
        self.player_actions = dict()
        all_lows = []
        all_highs = []
        self.action_indexs = []
        idx = 0
        for role in self.players: # Preserve order. Dicts should do this now but just to be safe
            # Process actions
            actions = self.coa_env.action_spaces[role]
            lows = []
            highs = []
            self.action_indexs
            for label, action in actions.items():
                # Need to add an inital entry for the start time
                lows.append(np.concatenate([[self.MIN_NEXT_ACTION_TIME],
                    np.array(action.low).reshape(-1)]))
                highs.append(np.concatenate([[self.MAX_NEXT_ACTION_TIME],
                    np.array(action.high).reshape(-1)]))

                L = len(np.array(action.low).reshape(-1)) + 1
                self.action_indexs.append((role, label, idx, idx + L))
                idx = idx + L

            lows = np.concatenate(lows)
            highs = np.concatenate(highs)
            all_lows.append(lows)
            all_highs.append(highs)
            self.player_actions[role] = Box(low = lows, high = highs)

        self.action_spaces = {"director": Box(
                low = np.concatenate(all_lows),
                high = np.concatenate(all_highs)
            )
        }

        # Observation spaces
        self.player_observations = dict()
        all_lows = []
        all_highs = []
        for role in self.players: # Preserve order. Dicts should do this now but just to be safe
            # Process actions
            observations = self.coa_env.observation_spaces[role]
            low = np.array(observations.low).reshape(-1)
            high = np.array(observations.high).reshape(-1)

            all_lows.append(low)
            all_highs.append(high)
            self.player_observations[role] = observations
        self.observation_spaces = {"director": Box(
                low = np.concatenate(all_lows),
                high = np.concatenate(all_highs)
            )
        }

############################################################
# Standard PettingZoo Interface Functions
############################################################
    def observation_space(self, role):
        return self.observation_spaces[role]

    def action_space(self, role):
        return self.action_spaces[role]

    def reset(self, seed=0, options=None):
        self.coa_env.reset()
        self.steps = 0
        self.cummulative_rew = np.zeros(len(self.possible_agents))

        self.coas = dict()
        for role in self.players:
            self.coas[role] = COA()

        return (
            {"director": self._process_observations(self.coa_env.last())},
            {},
        )

    def augment(self, params):
        self._setup(params)

    def seed(self, seed):
        self.parameter_generator.setseed(seed)

    def render(self, components=None, ao="rewards"):
        return self.coa_env.plot_trajectory_component(
            self.coa_env.state.trajectory,
            components = components,
            ao = ao
        )

    def step(self, action, render=False):
        self._process_actions(action)
        logger.debug("Director Step: COA: %s", self.coas)
        last_returns, step_end = self.coa_env.step(self.coas)
        logger.debug("Time: %s, Step End: %s", self.coa_env.state.current_t, step_end)

        # Process terminations and tructions
        term = False
        trunc = False

        if step_end["term_or_trunc"]:
            if all(list(last_returns[2].values())):
                # Unless everyone terminates, somebody must have trucated.
                term = True
            else:
                trunc = True

        # Process reward
        reward_til_now = sum(self.coa_env.state.cummulative_rews.values())
        step_reward = reward_til_now - self.cummulative_rew
        self.cummulative_rew = reward_til_now

        return (
            {"director": self._process_observations(last_returns)},
            {"director": step_reward},  ## Reward
            {"director": term},  ## Term
            {"director": trunc},  ## Trunc
            {"director": {}},  ## Info
        )

############################################################
# Converstion between actions and COAs
############################################################

    def _process_actions(self, action):
        action = action["director"]

        coas = {role:[] for role in self.players}

        logger.debug("Processing Action to COA")
        for role, label, i0, i1 in self.action_indexs:
            logger.debug("%s, %s, %s, %s, %s, %s", role, label, i0, i1, np.floor(action[i0]) + self.coa_env.state.current_t, action[i0+1:i1])
            event = {
                "start": np.floor(action[i0]) + self.coa_env.state.current_t,
                "label": label,
                "parameters": action[i0+1:i1],
                "role": role,
            }
            coas[role].append(event)

        for role in self.players:
            self.coas[role].add_events_from_dict(coas[role])

    def _process_observations(self, last_returns):
        # When working with a specific env you almost certanly want to change this
        # as there may be a ton of redudenent information in the combined observation
        # space
        obs = np.concatenate([last_returns[0][role].reshape(-1) for role in self.players])
        return obs

############################################################
# Wrapper Functions
############################################################

    def __getattr__(self, name: str) -> Any:
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""

        if name == "coa_env":
            if "coa_env" not in self.__dict__.keys():
                self.__dict__["coa_env"] = None
            return self.__dict__["coa_env"]

        if name in self.__dict__:
            return self.__dict__[name]

        if name == "unwrapped":
            return self.coa_env.env

        if name == "parallel_env":
            return self

        return getattr(self.coa_env.env, name)

############################################################
# DASH Viewer Functions
############################################################

    def set_fake_render_mode(self, fake_render_mode):
        self.fake_render_mode = fake_render_mode

# %%
if __name__ == "__main__":
    params = {
        "COACH_params":{
            "stochastic": True,
            # "FIXED_STEPS_PER_COM": {
            #     "checkin_frequency": 10
            # },
            "ACTION_PADDING": 0,
            "MIN_NEXT_ACTION_TIME": 1,
            "MAX_NEXT_ACTION_TIME": 10,
            "Agents": {
                "pursuer_0": {
                    "class_name": "DefaultActor",
                    "params": {"max_action_len":6}
                },
                "pursuer_1": {
                    "class_name": "DefaultActor",
                    "params": {"max_action_len":5}
                },
            },
            "seed": 453413,
        },

        "env_params": {"n_pursuers":2}
    }

    env = COACH_PettingZoo(env_creator=PettingZooEnv, COACHEnvClass=COACHEnvironment)

    env.augment(params)
    env.reset()

    print("#"*20, "COA Gym Information", "#"*20)
    print("Players:", env.players)
    print("Observation Space:", env.observation_spaces["director"].shape)
    print("Action Space:", env.action_spaces["director"].shape)
    print("Sample action:", env.action_spaces["director"].sample())

    for i in range(50):
        print("Turn", i)
        obs,rew,term,trunc,info = env.step({"director": env.action_spaces["director"].sample()})
        if all([a or b for a,b in zip(term.values(), trunc.values())]):
            print("Environment has terminated.")
            break

    env.render(ao="actions")
# %%
```

1.6 params.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from typing import Any, Type

def get_env_param_class(args: Any) -> Type:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env parameter objects
    """
    return EnvParams


"""Defines base class for environment parameter classes."""

from typing import Any, Optional


class EnvParams:
    """Base class for environment params."""

    def __init__(self, args: Any, param_section_name: str = "env_params") -> None:
        self.args = args
        self._params = getattr(args, param_section_name, {})

    def __getitem__(self, key: str) -> Any:
        """Return value stored for key from the params dict."""
        return self._params[key]

    def __setitem__(self, key: str, value: Any) -> None:
        """Set the value for key in the params dict."""
        self._params[key] = value

    def get(self, key: str, default: Optional[Any] = None) -> Any:
        """Return value for key from the params dict, or None if it doesn't exist."""
        try:
            return self._params[key]
        except KeyError:
            return default

    def get_mutated_params(self) -> "EnvParams":
        """Return a mutated copy of the params"""
        raise NotImplementedError(
            f"get_mutated_params has not been implemented in {type(self)}"
        )

    def checkpoint(self, folder: str) -> None:
        """Save a checkpoint in the given folder."""
        raise NotImplementedError(
            f"checkpoint has not been implemented in {type(self)}"
        )

    def reload(self, folder: str) -> None:
        """Read a checkpoint from the given folder."""
        raise NotImplementedError(f"reload has not been implemented in {type(self)}")
```

1.7 requirements.txt

```
astropy==5.3.4
dash==2.15.0
gymnasium==0.29.1
imageio==2.34.0
matplotlib==3.8.2
networkx==3.2.1
numpy==1.26.4
pandas==2.2.0
pettingzoo==1.24.3
Pillow==10.2.0
plotly==5.18.0
poliastro==0.17.0
PyYAML==6.0.1
PyYAML==6.0.1
scipy==1.12.0
stable_baselines3==2.2.1
SuperSuit==3.9.2
tensorflow==2.15.0.post1
torch==2.2.0
tqdm==4.66.2
```

1.8 texify.py

```
#!/usr/bin/env python3

import os, re, argparse

latex_header = r'''
\documentclass{article}
\usepackage[margin=1in]{geometry}
\usepackage{textcomp}
\usepackage{listingsutf8}
\usepackage{hyperref}
\usepackage[dvipsnames]{xcolor}
\definecolor{darkgreen}{rgb}{0,0.5,0}
\definecolor{lightblue}{rgb}{0.2,0.5,1}
\hypersetup{colorlinks=true, linkcolor=blue}
\lstset{
    numbers=left,
    upquote=true,
    breaklines=true,
    tabsize=4,
    showstringspaces=false,
    showspaces=false,
    breakatwhitespace=true,
    <SYNTAX_HIGHLIGHTING>
}
\begin{document}
\tableofcontents
\newpage
'''

styles = {
'default': r'''
    basicstyle=\ttfamily\scriptsize,
    keywordstyle=\ttfamily,
    commentstyle=\ttfamily\color{darkgreen},
    stringstyle=\ttfamily\color{blue},
''',
'dark': r'''
    backgroundcolor=\ttfamily\color{black},
    basicstyle=\ttfamily\color{white}\scriptsize,
    keywordstyle=\ttfamily,
    commentstyle=\ttfamily\color{green},
    stringstyle=\ttfamily\color{lightblue},
''' # xterm-mode
}

# Governs syntax highlighting
file_types = {
    '.py': 'Python',
     '.c': 'C',
     '.d': 'C',
     '.m': 'Matlab',
     '.r': 'R',
     '.sh': 'bash',
     '.bash': 'bash',
     '.cpp': 'C++',
     '.cc': 'C++',
     '.pl': 'Perl',
     '.tex': 'TeX',
     '.f': 'Fortran',
     '.for': 'Fortran',
     '.ftn': 'Fortran',
     '.f90': 'Fortran',
     '.f95': 'Fortran',
     '.f03': 'Fortran',
     '.f08': 'Fortran',
     '.csh': 'csh',
     '.ksh': 'ksh',
     '.lisp': 'lisp',
     '.lsp': 'lisp',
     '.cl': 'lisp',
     '.l': 'lisp',
     '.scm': 'lisp',
     '.go': 'Go',
     '.hs': 'Haskell',
     '.lhs': 'Haskell',
     '.bat': 'command.com',
     '.awk': 'Awk',
}
```

```python

def main() -> None:
    parser = argparse.ArgumentParser(usage='%s [-d DIR] [-i extension ...]\n' % __file__
        + 'example: %s -d ./src -i foo.m -i makefile .c .d .py\n\n' % __file__,
        description='Will search under DIR for all source files with the specified file extensions, and compile them into a LaTeX file.')
    parser.add_argument('--dir', '-d', help='root directory under which to search', default='.')
    parser.add_argument('--include', '-i', action='append', help="Explicitly include a file even if it doesn't match the extension list", default=[])
    parser.add_argument('--style', default='default', choices=styles.keys(), help='Changes syntax highlighting, etc.')
    parser.add_argument('extension', nargs='+', help='Only files with these extensions will be included (leading dot optional)')
    args = parser.parse_args()

    # Permit valid extensions to be input with or without the dot
    args.extension = [ a if ('.' == a[0]) else '.%s' % a
        for a in args.extension ]

    # Make relative to base path, escape underscores
    def format_path(path: str) -> str:
        if path == args.dir: return '/'
        assert (path[0:len(args.dir)+1] == args.dir + '/') or (path[0:len(args.dir)+1] == args.dir + '\\')
        return re.sub('_', r'\_', path[len(args.dir)+1:])

    # Print single file
    def dumpsrc(dirpath: str, fname: str) -> str:
        path = '%s/%s' % (dirpath, fname)
        escaped = format_path(path)
        print(r'\subsection[%s]{%s}' % (os.path.basename(escaped), escaped))
        ext = os.path.splitext(f)[1]
        if ext in file_types:
            s = r'\lstinputlisting[language=%s]{%s}' % (file_types[ext], path)
        else:
            s = r'\lstinputlisting{%s}' % path
        return '%s\n%s\n' % (s, r'\newpage')

    def print_header() -> None:
        s = latex_header.replace(r'<SYNTAX_HIGHLIGHTING>', styles[args.style].strip(), 1)
        print(s.strip())

    print_header()

    dirs = {dirpath:fnames for dirpath, _, fnames in os.walk(args.dir)}
    includes = { os.path.realpath(f):f for f in args.include }
    for dirpath in sorted(dirs):
        fnames = dirs[dirpath]
        src = sorted([f for f in fnames
            if (os.path.splitext(f)[1] in args.extension) or (os.path.realpath(f) in includes)])
        if 0 == len(src): continue

        print(r'\section{%s}' % format_path(dirpath))
        for f in src:
            print(dumpsrc(dirpath, f))

            # Don't include files twice just because they're explicitly included with -i
            f = os.path.realpath(f)
            if f in includes:
                del includes[f]

    # Any explicitly included files that weren't already covered (i.e. those outside args.path)
    if len(includes):
        print(r'\section{Miscellaneous}')
        for _,f in includes.items():
            f = args.dir + '/' + os.path.relpath(f, args.dir)
            print(dumpsrc(os.path.dirname(f), os.path.basename(f)))

    print(r'\end{document}')

main()
```

2 DASH

2.1 DASH/app.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import os
import re
import sys


# Utility Libraries
import numpy as np
import copy
import json

# Import Dash Things
from dash import Dash, html, dcc, Input, Output, callback, State, ALL, no_update, ctx, MATCH
import plotly.express as px
import plotly.graph_objects as go

# Import Custom Things
from DASH.html_objects import app_layout, plan_menu, actions_display
from DASH.dash_utilities import callback_tools as ct

# Import Env Things:
from DASH.coach_integration import COACHIntegration
from coach import COACHEnvironment

# Logging
import logging

logging.basicConfig(level=logging.DEBUG, format="%(levelname)s:%(name)s:%(message)s")
logger = logging.getLogger(__name__)

pymunk_loggers = [logging.getLogger(name) for name in logging.root.manager.loggerDict if name.startswith("pymunk")]

for log_handler in pymunk_loggers:
    log_handler.setLevel(logging.INFO)

##########################################################
# Inputs
##########################################################


### Plan Selector
@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input({"type":"plan", "id": ALL}, "n_clicks"),
    prevent_initial_call=True,
)

def select_plan(plan):
    logger.debug("INPUT: select_plan")

    plan_id = ctx.triggered_id['id']
    logger.debug("\t %s %s %s","select_plan", plan, plan_id)
    if plan[plan_id] > 0:
        ct.env_factory.plans.set_current_plan(plan_id)

    return True



### Plan View Checkboxes
@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input({"type":"plan_view", "id": ALL}, "value"),
    prevent_initial_call=True,
)
def view_plan(plan_checks):
    logger.debug("INPUT: view_plan")

    plan_id = ctx.triggered_id['id']
    logger.debug("\t %s %s %s", "view_plan", plan_checks, plan_id)
    checked = len(plan_checks[plan_id])>0

    if checked:
        ct.env_factory.plans.active.add(plan_id)
        logger.debug("\t %s %s", "plan added, active plans:", ct.env_factory.plans.active)
    else:
        if plan_id in env_factory.plans.active:
            env_factory.plans.active.remove(plan_id)
            logger.debug("\t %s %s", "plan removed, active plans:", ct.env_factory.plans.active)
        else:
            return no_update

    return True

### Interface Selector
@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input({"type": "agent_interface", "role": ALL}, "value"),
    prevent_initial_call=True,
)
def choose_interface(agent_interface):
    logger.debug("INPUT: choose_interface")

    # Get the role that changed.
    role = ctx.triggered_id['role']
    interface = ct.get_role_from_callback(role, agent_interface)
    logger.debug("\t %s %s","choose_interface:", role)

    logger.debug("\t %s %s %s %s %s %s","Interface Selector:", role, "new interface:", interface, "old interface:", ct.env_factory.plans.current.get_interface(role)[0])
    if interface == ct.env_factory.plans.current.get_interface(role)[0]:
        logger.debug("INPUT: choose_interface - no_update")
        return no_update
    else:
        logger.debug("INPUT: choose_interface - updated")
        ct.env_factory.plans.current.set_interface(role, interface)
        return True



### Model Selector
@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input({"type": "onboard_model", "role": ALL}, "value"),
    prevent_initial_call=True,
)
def select_model(onboard_model):
    logger.debug("INPUT: select_model")
    role = ctx.triggered_id['role']

    model = ct.get_role_from_callback(role, onboard_model)
    if model == ct.env_factory.plans.current.get_model(role):
        return no_update
    else:
        logger.debug("\t Setting Model for Plan %s %s %s %s %s", ct.env_factory.plans.current.id, "for role", role, "to", model)
        ct.env_factory.plans.current.set_model(role, model)
        return True



### Action Parameters
@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input({"type": "actionparam", "plan": ALL, "time": ALL, "type":ALL, "index": ALL, "role": ALL}, "value"),
    State({"type": "actionparam", "plan": ALL, "time": ALL, "type":ALL, "index": ALL, "role": ALL}, "id"),
    prevent_initial_call=True,
)
def change_action_params(params, id):
    logger.debug("INPUT: change_action_params")
    logger.debug("\t %s %s", "calling id:", id)

    if ctx.triggered_id is None:
        ## This got called by a plan change
        return no_update

    logger.debug("\t %s %s", "ctx.triggered_id:", ctx.triggered_id)

    role = ctx.triggered_id['role']
    logger.debug("\t %s %s %s %s","New Params:", params, "role:", role)

    locks["action_card"] = True
    logger.debug("\t %s","LOCKING ACTION PARAMS")

    i = id.index(ctx.triggered_id)
    plan = ct.env_factory.plans.get(ctx.triggered_id["plan"])
    coa = plan.coas[ctx.triggered_id["role"]]
    event = coa.get(time = ctx.triggered_id["time"], label = ctx.triggered_id["type"])

    logger.debug("\t %s %s","Param_Update: Current COA" , coa)

    event.parameters[ctx.triggered_id["index"]] = params[i]

    logger.debug("\t %s %s","Param_Update: New COA" , coa)

    return True

### Timeline Selector

@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input({"type": "timeline", "role": ALL}, "value"),
    prevent_initial_call=True,
)

def edit_timeline(new_timelines):
    role = ctx.triggered_id['role']

    logger.debug("\t %s","INPUT: edit_timeline")
    logger.debug("\t %s %s","new_timelines", new_timelines)
    logger.debug("\t %s %s","Old COA:", ct.env_factory.plans.current.coas[role])

    # Figure out which timeline event changed
    changed_timeline = ct.get_role_from_callback(role, new_timelines)
    logger.debug("\t %s %s %s","updated timeline:", role, changed_timeline)

    if len(changed_timeline)==0:
        logger.debug("INPUT: edit_timeline - no_update")
        return no_update

    old_timeline = ct.env_factory.plans.current.get_dash_timelines()[role]

    # If we drag a timepoint across another it will change it's position in the ordering
    # so we need some extra logic around that
    old = []
    new = []

    for j_old, j_new in zip(old_timeline, changed_timeline):
        if not j_old == j_new:
            old.append(j_old)
            new.append(j_new)

    if len(old)==0:
        logger.debug("INPUT: edit_timeline - no_update")
        return no_update

    # Check if we've moved one point past another
    if len(old)>1:
        if new[0] == old[1]:
            j_new = new[1]
            j_old = old[0]
        else:
            j_new = new[0]
            j_old = old[1]
    else:
        j_new = new[0]
        j_old = old[0]

        # Check if we're moving one point on top of another
        if j_new in old_timeline:
            # Kind of dumb trick to make it bounce to the side you're dragging it from.
            sign = -(j_new - j_old)/abs(j_new - j_old)
            while j_new in changed_timeline:
                j_new += sign*1

    logger.debug("\t %s %s","New Timelines:", changed_timeline)
    logger.debug("\t %s %s","Old Timelines:", old_timeline)
    logger.debug("\t %s %s","Change Index:", j_old, j_new)

    ct.env_factory.plans.current.coas[role].move_events(time=j_old, to=j_new)

    logger.debug("\t %s %s","New COA:", ct.env_factory.plans.current.coas[role])

    logger.debug("INPUT: edit_timeline - Update Backend")
    return True

##########################################################
# Buttons
##########################################################

### New Plan
@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input("button_new_plan", "n_clicks"),
    prevent_initial_call=True,
)
def new_plan(nclicks):
    logger.debug("BUTTON: new_plan")
    ct.env_factory.plans.new_plan()

    return True

### New Command
@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input({"type":"button_add_new_command", "role":ALL}, "n_clicks"),
    prevent_initial_call=True,
)
def new_command(nclicks):
    logger.debug("BUTTON: new_command")
    role = ctx.triggered_id['role']
    current_plan = ct.env_factory.plans.current
    current_plan.new_event(role)

    locks["action_card"] = True

    return True

### Simulate Plan
@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input("button_simulate_plan", "n_clicks"),
    prevent_initial_call=True,
)
def simulate_plan(nclicks):
    logger.debug("BUTTON: simulate_plan")
    current_plan = ct.env_factory.plans.current
    ct.env_factory.run_plan(current_plan)

    return True



### Generate Plan
@callback(
    Output("update_backend_coa", "data", allow_duplicate=True),
    Input("button_generate_plan", "n_clicks"),
    prevent_initial_call=True,
)
def simulate_plan(nclicks):
    logger.debug("BUTTON: generate_plan")
    ct.env_factory.generate_plan()
    return True


##########################################################
# Outputs
##########################################################

@callback(
    Output({"type": "onboard_model", "role": ALL}, "options", allow_duplicate=True),
    Output({"type": "onboard_model", "role": ALL}, "value", allow_duplicate=True),
    Output({"type": "timeline", "role": ALL}, "value", allow_duplicate=True),
    Output({"type": "agent_interface", "role": ALL}, "value", allow_duplicate=True),
    Output({"type": "actions_card", "role": ALL}, "children", allow_duplicate=True),
    Output("table_plans", "children", allow_duplicate=True),
    Output('button_generate_plan', 'disabled'),
    Input("update_frontend_elements", "data"),
    prevent_initial_call=True,
)

def display_choose_model(update):
    logger.debug("OUTPUT: display_choose_model")

    possible_models = ct.make_returns_from_dict(ct.env_factory.get_current_models())
    timelines = ct.make_returns_from_dict(ct.env_factory.plans.current.get_dash_timelines())
    interface = ct.make_returns_from_dict(ct.env_factory.plans.current.interfaces)
    actions = actions_display(ct.env_factory)
    plans = plan_menu(ct.env_factory)
    generator = not ct.env_factory.generator_available()

    selected_model = [ct.env_factory.plans.current.models[role] for role in ct.env_factory.roles]

    logger.debug("\t %s %s","Models", possible_models)
    logger.debug("\t %s %s","Selected Model", selected_model)
    logger.debug("\t %s %s","Timelines", timelines)
    logger.debug("\t %s %s","Interfaces", interface)

    # return possible_models, Tb_children, timelines, interface
    return possible_models, selected_model, timelines, interface, actions, plans, generator


## Update COA Visualizations
@callback(
    Output("plotly_visialization", "figure", allow_duplicate=True),
    Input('update_frontend_elements', 'data'),
    prevent_initial_call=True,
)

def visualization(update_frontend_elements):
    logger.debug("OUTPUT: visualization")
    if ct.env_factory.plans.current.visualizations is not None:
        return ct.env_factory.plans.current.visualizations
    else:
        return no_update



## Update Telemetry

@callback(
    Output("stats-graphic-1", "figure", allow_duplicate=True),
    Output("stats-graphic-2", "figure", allow_duplicate=True),
    Output("stats-graphic-3", "figure", allow_duplicate=True),
    Input('update_frontend_elements', 'data'),
    prevent_initial_call=True,
)

def telemetry(update_frontend_elements):
    logger.debug("OUTPUT: telemetry")
    figures = []

    for plan_id in ct.env_factory.plans.active:
        logger.debug("\t %s %s", "active plan_id", plan_id)
        plan = ct.env_factory.plans.get(plan_id)

        if plan.telemetry is not None:
            for i, t in enumerate(plan.telemetry[:3]):
                if len(figures) <= i:
                    logger.debug("\t %s %s %s", "adding figure", len(figures), i)
                    f = go.Figure()
                    f.update_layout(
                        margin={"l": 5, "b": 0, "t": 0, "r": 80},
                        hovermode="closest",
                        showlegend=False,
                        xaxis_title=t.xlabel,
                        yaxis_title=t.ylabel,
                    )
                    figures.append(f)

                t = t.as_df()
                opacity = .2
                if plan_id == ct.env_factory.plans.current.id:
                    logger.debug("\t %s %s %s", "active plan_id", plan_id, "this plan is the current one.")
                    opacity = 1

                for col in t.columns:
                    figures[i].add_trace(go.Scatter(x=t.index,y=t[col],
                        mode='lines',
                        name=f"{plan_id}_{col}",
                        opacity=opacity
                        ))

    if len(figures) == 0:
        return no_update

    while len(figures)<3:
        figures.append(go.Figure())

    return figures

##########################################################
# Update Backend
##########################################################

@callback(
    Output("update_frontend_elements", "data", allow_duplicate=True),
    Input("update_backend_coa", "data"),
    # prevent_initial_call=True,
    prevent_initial_call='initial_duplicate'
)

def update_coa(coa_update):
    logger.debug("BACKEND: update_coa")
    return True

##########################################################
# Setup
##########################################################

locks = {
            "action_card": False
        }

if __name__ == "__main__":
    logger.debug("app - ################### RELOADING DASH APP #########################")
    # external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']
    env_factory = env_creator(get_env_class)
    ct.env_factory = env_factory

    ## App Layout
    app = Dash(__name__)
    app.layout = app_layout(env_factory)
    app.run(debug=True)
```

2.2 DASH/coach_integration.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import sys, inspect

from utilities.planning import COA, TimelineEvent
from coach import CommunicationSchedule
import plotly.express as px

from argparse import Namespace
from utilities.iotools import NumpyDecoder
import json
import os
import copy
import numpy as np
from itertools import product
import logging
logger = logging.getLogger(__name__)

from coach import (
    SeededParameterGenerator,
    BaseParameterGenerator,
)

import agents as agents_module
import directors as directors_module


def role_type(role):
    return role.split("_")[0]

class COACHIntegration:
    def __init__(self,
        env_creator,
        COACHEnvClass,
        parameters,
        agents_module=agents_module
    ) -> None:
        self.agents_module=agents_module
        self.env_creator = env_creator
        self.COACHEnvClass = COACHEnvClass
        self.parameters = parameters
        self.render_gif = False

        self.library = dict()
        self._id_to_interface = dict()
        self._interface_to_id = dict()

        self.setup()

        self.plans = PlanLibrary(
            self,
            default_interfaces=self.default_interfaces,
            default_models=self.default_models,
        )

        self.actions_container = {}

    def get_current_models(self):
        return {role: self.get_interface_models(interface) for role, interface in self.plans.current.interfaces.items()}

    # def get_interfaces(self):
    #     return list(self.interfaces.keys())

    def set_current_interface(self, role, interface):
        self.current_interface[role] = interface

    def get_interfaces_by_role(self, role):
        return list(self.interfaces_by_role[role].keys())

    def get_current_interface(self, role):
        return self.current_interface[role]

    def get_interface_models(self, interface):
        return list(self.model_params_by_interface[interface].keys())

    def get_current_model(self):
        return self.current_model

    def set_current_model(self, role, model):
        self.current_model[role] = model

    def actions(self, role):
        agt = self.coach_env.agents[role]
        return agt.interface.action_dictionary

    # Deal with Plan Library
    def new_plan(self, name = None):
        self.plans.new_plan(name)

    def get(self, id):
        return(self.library[id])

    def keys(self):
        return(self.library.keys())

    def items(self):
        return(self.library.items())

    def values(self):
        return(self.library.values())

    def by_interface(self, interface):
        return(self._interface_to_id[interface])

    def get_by_interface(self, interface, id):
        return(self._interface_to_id[interface][id])

    def plot_COA(self, COA):

        logger.debug("coach_integration.plot_COA - Generated COA Trajectory: %s", COA)
        traj = self.coach_env.get_traj_from_coas({"player_0": COA})

        logger.debug("coach_integration.plot_COA - Generated COA Trajectory: %s", len(traj))

        return plot_trajectory_component(traj, {"player_0": COA}, self.coach_env.env)


    def setup(self):
        ## This should set up the game, get the information about it, and fix things like the
        ## which roles can take which models.
        ## COA_Env has agents, need to extract them from env.
        # Create COA Env

        self.COACH_params = COACH_params = self.parameters["COACH_params"]
        self.env_params = env_params = self.parameters["env_params"]
        self.actor_params = actor_params = self.parameters["actor_params"]

        # Set up parameter generator
        if COACH_params["stochastic"]:
            self.parameter_generator = SeededParameterGenerator(23423, env_params)
        else:
            self.parameter_generator = BaseParameterGenerator(23423, env_params)

        # Set up comm schedule
        if "FIXED_STEPS_PER_COM" in COACH_params.keys():
            schedule_param = COACH_params["FIXED_STEPS_PER_COM"]
            self.comm_schedule = CommunicationSchedule.repeating(**schedule_param)
        else:
            logger.info("Communication Schedule Is Non-repeating")
            self.comm_schedule = CommunicationSchedule(length=0)

        self.ACTION_PADDING = COACH_params.get("ACTION_PADDING", 0)
        self.MIN_NEXT_ACTION_TIME = COACH_params.get("MIN_NEXT_ACTION_TIME", 1)
        self.MAX_NEXT_ACTION_TIME = COACH_params.get("MAX_NEXT_ACTION_TIME", np.inf)

        self.coach_env = self.COACHEnvClass(
            env_creator=self.env_creator,
            parameter_generator=self.parameter_generator,
            # agents=agent_dict,
            fill_random=False,
            comm_schedule=self.comm_schedule,
            seed=COACH_params["seed"],
        )

        self.coach_env.augment(self.parameter_generator)
        self.coach_env.reset()

        self.roles = self.coach_env.env.agents
        self.role_to_idx = {role:idx for idx, role in enumerate(self.roles)}
        self.idx_to_role = {idx:role for idx, role in enumerate(self.roles)}

        self.role_types = list(set([role_type(role) for role in self.roles]))
        self.possible_models = self.coach_env.possible_models()


        # Setup Actor Interfaces
        self.interfaces_by_role = {role: dict() for role in self.roles}
        self.roles_by_interface = dict()
        self.interfaces = dict()
        self.model_params_by_interface = dict()
        self.model_params = dict()
        self.model_classes = dict()
        self.model_to_interface = dict()

        for if_name, if_params in self.actor_params["interfaces"].items():
            # Get the interface class
            if_Class = self.agents_module.__dict__.get(if_params["interface_class"])
            logger.debug("Current Class: %s", if_Class)
            logger.debug("Current Params: %s", if_params)
            # if_Class = self.coach_env.__class__.agent_selection.Interfaces[if_params["interface_class"]]

            # Get a reference interface, this is make sure that any model class instiated has the correct interface
            role = if_params.get("roles", self.roles)[0] # Get one applicable role
            if_reference = if_Class(role, self.coach_env.env, **if_params.get("iterface_parameters", dict()))
            logger.debug("Current Class: %s", if_reference)

            # Set up references to interfaces
            self.interfaces[if_name] = if_reference

            # Get roles for which the interface is valid
            for role in if_params.get("roles", self.roles):
                self.interfaces_by_role[role][if_name] = if_reference
                if if_name not in self.roles_by_interface.keys():
                    self.roles_by_interface[if_name] = []
                self.roles_by_interface[if_name].append(role)

            # Setup reference to model creation information
            self.model_params_by_interface[if_name] = if_params["models"]

            self.model_classes[if_name] = dict()
            for model_name, model_params in if_params["models"].items():
                self.model_to_interface[model_name] = if_name
                self.model_classes[if_name][model_name] = self.agents_module.__dict__.get(model_params["class_name"])
                self.model_params[model_name] = model_params

        # Setup Directors
        directors = self.actor_params["directors"]
        self.directors = dict()
        self.directors_allow = dict()

        for dr_name, dr_params in directors.items():

            self.default_models = {role: params["classes"][0] for role, params in dr_params['roles'].items()}
            self.default_interfaces = {role: self.model_to_interface[model] for role, model in self.default_models.items()}

            tmp_params = copy.copy(self.parameters)

            for role, params in dr_params["roles"].items():
                tmp_params["COACH_params"]["Agents"][role] = self.model_params[params["classes"][0]]

            dr_class = directors_module.__dict__.get(dr_params["class_name"])
            self.directors[dr_name] = dr_class(
                env_creator = self.env_creator,
                COACHEnvClass = self.COACHEnvClass,
                params = tmp_params,
                model_path = dr_params["path"]
            )

            class_iter = product(*[classes["classes"] for classes in dr_params["roles"].values()])

            for t in class_iter:
                self.directors_allow[tuple(t)] = self.directors[dr_name]
                self.model_to_interface

    def generator_available(self):
        return tuple(self.plans.current.models.values()) in self.directors_allow.keys()


    def generate_plan(self):
        current_models = self.plans.current.models
        director = self.directors_allow[tuple(current_models.values())]
        coas, traj = director.generate_coas(self.parameters)

        tmp = self.plans.new_plan()
        tmp.coas = coas
        tmp.trajectory = traj

        tmp.interfaces = copy.copy(self.plans.current.interfaces)
        tmp.models = copy.copy(self.plans.current.models)
        tmp.model_classes = copy.copy(self.plans.current.model_classes)
        tmp.model_params = copy.copy(self.plans.current.model_params)


    def run_plan(self, plan):
        logger.debug("env_factory: run_plan")
        logger.debug("\t plan info: %s %s %s", plan.name, plan.interfaces, plan.models)

        if self.render_gif:
            traj = self.coach_env.get_traj_from_plan(plan, render=True)
            frames = np.array(traj.frames)
            plan.visualizations = px.imshow(frames, animation_frame=0, binary_string=True, labels=dict(animation_frame="slice"))
        else:
            traj = self.coach_env.get_traj_from_plan(plan, render=False)

            plan.visualizations = self.coach_env.plot_trajectory_component(
                traj,
                plan=plan,
                env=self.coach_env.env,
                render_mode="plotly"
            )

        plan.trajectory = traj
        plan.telemetry = self.coach_env.trajectory_telemetry(traj, plan, self.coach_env.env)


class Plan:
    id = 0

    @staticmethod
    def fromCOAs(coas, name=None):
        plan = Plan(coas.keys())
        if name is not None:
            plan.name = name

        plan.coas = coas
        return plan

    def __init__(self,
        roles,
        name = None,
        interfaces_by_role=None,
        model_classes_library=None,
        model_params_library=None
        ):
        self.id = copy.copy(Plan.id)
        if name is None:
            name = f"Plan_{Plan.id}"
            Plan.id += 1

        self.interfaces_by_role = interfaces_by_role
        self.model_classes_library = model_classes_library
        self.model_params_library = model_params_library

        self.name = name
        self.roles = list(roles)

        self.coas = {role: COA() for role in self.roles}
        self.interfaces = {role: None for role in self.roles}
        self.models = {role: None for role in self.roles}
        self.model_classes = {role: None for role in self.roles}
        self.model_params = {role: dict() for role in self.roles}
        self.trajectories = {role: None for role in self.roles}

        self.visualizations = None
        self.telemetry = None

    def __getitem__(self, role):
        return self.coas[role]

    def items(self):
        return self.coas.items()

    def values(self):
        return self.coas.values()

    def keys(self):
        return self.coas.keys()

    def set_interface(self, role, interface):
        if hasattr(interface, "__len__"):
            if len(interface) == 0:
                interface = None
        self.interfaces[role] = interface
        self.models[role] = None
        self.coas[role] = COA()

    def get_interface(self, role):
        interface = self.interfaces.get(role,None)
        instance = self.interfaces_by_role[role][interface]

        return interface, instance

    def set_model(self, role, model):
        self.models[role] = model
        self.model_classes[role] = self.model_classes_library[self.interfaces[role]][model]
        self.model_params[role] = self.model_params_library[self.interfaces[role]][model].get("params", dict())

    def get_model(self, role):
        return self.models[role]

    def get_timelines(self):
        return {role: self.coas[role].to_dict() for role in self.roles}

    def get_dash_timelines(self):
        timelines = self.get_timelines()
        for k,v in timelines.items():
            if len(v) == 0:
                timelines[k] = []
            else:
                timelines[k] = list(v.keys())
        return timelines


    def new_event(self, role):
        logger.debug("Adding new event to COA: %s", self.coas[role])
        action_name, act_box = list(self.get_interface(role)[1].action_dictionary.items())[0]

        ## [{"start":int, "action": str, "parameters":np.array}]
        last_t = -1
        if len(list(self.coas[role].timeline.keys()))>0:
            last_t = list(self.coas[role].timeline.keys())[-1]

        logger.debug("Action Name: %s", action_name)

        tmp = TimelineEvent(
            label=action_name,
            parameters=copy.copy(act_box.default),
            tags=role
        )

        self.coas[role].add_event(time=last_t + 1, event=tmp)
        logger.debug("Added new event to COA: %s", self.coas[role])


class PlanLibrary:
    def __init__(self,
        env_factory,
        plans = [],
        default_interfaces=None,
        default_models=None
        ):
        self.roles = env_factory.roles
        self._plans: list[Plan] = plans
        self.current: Plan = None
        self.active = set()
        self.interfaces_by_role = env_factory.interfaces_by_role
        self.model_classes_library = env_factory.model_classes
        self.model_params_library = env_factory.model_params_by_interface

        if not default_interfaces:
            self.default_interfaces = {role: env_factory.get_interfaces_by_role(role)[0] for role in self.roles}
        else:
            self.default_interfaces = default_interfaces

        if not default_models:
            self.default_models = {role: env_factory.get_interface_models(self.default_interfaces[role])[0] for role in self.roles}
        else:
            self.default_models = default_models

        self.new_plan()
        self.current = self.all()[0]
        self.active.add(self.current.id)

    def ids(self):
        return list(self._id_to_plan.keys())

    def get(self, id):
        return self._id_to_plan[id]

    def set_current_plan(self, id):
        self.current = self._id_to_plan[id]

    def new_plan(self, name=None):
        tmp = Plan(
            self.roles,
            name=name,
            interfaces_by_role = self.interfaces_by_role,
            model_classes_library=self.model_classes_library,
            model_params_library=self.model_params_library
        )

        for role, interface in self.default_interfaces.items():
            tmp.set_interface(role, interface)
            tmp.set_model(role, self.default_models[role])

        self.add_plan(tmp)
        return tmp

    def add_plan(self, plan: Plan):
        self._plans.append(plan)
        self._id_to_plan = {plan.id:plan for plan in self._plans}

    def all(self):
        return self._plans



class COALibrary:
    def __init__(self, roles, coas: dict[str, COA] = None):
        self.roles = roles
        self.role_types = list(set([role_type(role) for role in roles]))

        self._library = {role_type:[] for role_type in self.role_types}

        if coas is not None:
            for role_tag, coa in coas.items:
                self.add_coa(role_tag, coa)

    def add_coa(self, role_tag, coa):
        if role_tag in self.roles:
            role_type = role_tag
        elif role_type(role_tag) in self.roles:
            role_type = role_type(role_tag)
        else:
            raise Exception(f"Role type {role_tag} cannot be found. Valid role types are {list(self._library.keys())}.")

        if coa not in self._library[role_type]:
            self._library[role_type].append(coa)

    def __setitem__(self, key, item):
        self._library[key] = item

    def __getitem__(self, key):
        return self._library[key]

    def values(self):
        return self._library.values()

    def items(self):
        return self._library.items()

    def keys(self):
        return self._library.keys()
```

2.3 DASH/dash_utilities.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

class callback_tools:
    env_factory = None

    @staticmethod
    def list_to_role(entry_list):
        for i, entry in enumerate(entry_list):
            if entry is not None:
                return i, callback_tools.env_factory.roles[i], entry

    @staticmethod
    def make_role_returns(role, r):
        returns = [None]*len(callback_tools.env_factory.roles)
        i = callback_tools.env_factory.roles.index(role)
        returns[i] = r
        return(returns)

    @staticmethod
    def get_role_from_callback(role, callback):
        # print("Roles:", env_factory.role_to_idx)
        return callback[callback_tools.env_factory.role_to_idx[role]]

    @staticmethod
    def make_returns_from_dict(r):
        returns = [None]*len(callback_tools.env_factory.roles)
        for i, role in enumerate(callback_tools.env_factory.roles):
            if role in r.keys():
                returns[i] = r[role]

        return callback_tools.nones_to_empty_list(returns)

    @staticmethod
    def make_returns(r):
        return [r]*len(callback_tools.env_factory.roles)

    @staticmethod
    def nones_to_empty_list(r):
        if type(r) is list:
            return [callback_tools._ntol(v) for v in r]
        if type(r) is dict:
            return {k:callback_tools._ntol(v) for k, v in r.items()}

    @staticmethod
    def _ntol(r):
        if r is None:
            return []

        return r
```

2.4 DASH/html_objects.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from dash import Dash, html, dcc, Input, Output, callback, State

def app_layout(env_factory):
    ## The role interface select
    RoleInterfaceSelect = []

    for role in env_factory.roles:
        RoleInterfaceSelect.append(html.Div(
                [
                    html.Div([
                        role,
                        dcc.Dropdown(
                            env_factory.get_interfaces_by_role(role),
                            placeholder=f"Interface",
                            id={"type": "agent_interface", "role": role}
                            ),
                        dcc.Dropdown(
                            id={"type": "onboard_model", "role": role},
                            placeholder=f"Model"
                            ),
                        html.Button("Add New Command", id={"type": "button_add_new_command", "role": role}),
                        ],
                        className="left_card",
                        style={"float": "left", "margin": "auto"},
                    ),
                    html.Div(
                        className="right_card",
                        id={"type": "actions_card", "role": role},
                        style={"float": "left", "margin": "auto"},
                    ),
                ],
                className="row agent_card",
            ))

    RoleInterfaceSelect = html.Div(RoleInterfaceSelect, className="action_cards")

    ## The Role Timeline Selection
    TimeLines = []
    for role in env_factory.roles:
        TimeLines += [
            html.Tr([
                html.Td(role, style={"float": "left", "margin": "auto", "width": 100}),
                html.Td(
                    dcc.RangeSlider(
                        0,
                        100,
                        value=[],
                        tooltip={"placement": "bottom", "always_visible": True},
                        id={"type": "timeline", "role": role},
                    ),
                style={"margin": "auto", "width": "100%"}
                ),
            ])
        ]

    if len(TimeLines)==1:
        TimeLines = TimeLines[0]

    TimeLines = html.Table(TimeLines)

    ## App Layout
    app_layout = html.Div(
        [
            html.Div(
                [
                    html.Div(
                        [
                            dcc.Graph(
                                id="plotly_visialization",
                                style={"width": "600px", "height": "600px"},
                            ),
                        ],
                        style={"width": "48%", "display": "inline-block"},
                    ),
                    html.Div(
                        [
                            dcc.Graph(id="stats-graphic-1", style={"height": "150px"}),
                            dcc.Graph(id="stats-graphic-2", style={"height": "150px"}),
                            dcc.Graph(id="stats-graphic-3", style={"height": "150px"}),
                        ],
                        style={"width": "48%", "float": "right", "display": "inline-block"},
                    ),
                ]
            ),
            TimeLines,
            html.Button("New Plan", id="button_new_plan"),
            html.Button("Generate Plan", id="button_generate_plan"),
            html.Button("Simulate Plan", id="button_simulate_plan"),
            ## These are True False values, they signle that the corresponding object should be updated.
            dcc.Store(id="update_backend_coa"),
            dcc.Store(id="update_frontend_elements"),
            ### Agent Card:
            html.Div(RoleInterfaceSelect),
            html.Div([
                "Plans",
                html.Table(id="table_plans")
            ], id="div_plans", style = {"width":"15%", "float": "right", "display": "inline-block", "padding":"10px", "border":"1px solid black"})
        ],
    )

    return app_layout


def action_table_row(time, event, interface, plan="", role=""):
    CELL_WIDTH = "80px"

    ## Action Label:
    html_objects = []

    ## Create the header describing the actions
    action_head = [
        html.Td("", style={"padding-left": "25px", "width": "200px"}),
        html.Td("Time", style={"width": CELL_WIDTH, 'font-size': 10, 'font-style': 'italics'}),
    ]

    action_params = interface.action_dictionary[event["label"]]

    for param in action_params.items():
        # Give the name in the heading
        action_head.append(html.Td(param.description, style={"width": CELL_WIDTH, 'font-size': 10, 'font-style': 'italics'}))

    # Store header
    html_objects.append(html.Tr(action_head))

    TDs = [
        html.Td(
            [
                html.Div(
                    className=f"rc-slider-handle rc-slider-handle-{time}",
                    style={"margin-top": "0px", "margin-right": "5px"},
                ),
                html.Span(event["label"], style={"padding-left": "25px"}),
            ],
            style={"width": "200px"},
        )
    ]

    TDs.append(
        html.Td(
            dcc.Input(
                id={
                    "type": "actionstart",
                    "plan": plan,
                    "time": time,
                    "type": event['label'],
                    "role": role
                },
                type="number",
                value=time,
                placeholder=f"Step",
                style={"width": CELL_WIDTH},
                readOnly=True,
                debounce = True,
            )
        )
    )

    # print("Event Parameters:", event["parameters"])

    for i, param in enumerate(action_params.items()):
        for j in range(param.shape[0]):
            TDs.append(
                html.Td(
                    dcc.Input(
                        id={
                            "type": "actionparam",
                            "plan": plan,
                            "time": time,
                            "type": event['label'],
                            "index": i,
                            "role": role
                        },
                        type="number",
                        min=param.low[j],
                        max=param.high[j],
                        value=event["parameters"][i],
                        placeholder=f"[{param.low[j]:.3}, {param.high[j]:.3}]", # the :.3 is the number of significant figures
                        style={"width": CELL_WIDTH},
                        debounce = True,
                    )
                )
            )

    # print("Event Parameters:", TDs)
    html_objects.append(html.Tr(TDs))

    return html_objects


def actions_display(env_factory):
    current_plan = env_factory.plans.current
    html_objects = []
    for role in env_factory.roles:
        ## Render COA Actions
        print("\t",f"Calling Model Card for {role}")

        timelines = env_factory.plans.current.get_timelines()
        interface_name, interface = current_plan.get_interface(role)

        html_sub_objects = []

        for i, (time, events) in enumerate(timelines[role].items()):
            for j, event in enumerate(events.values()):
                html_sub_objects += action_table_row(time, event, interface, plan=current_plan.id, role=f"{role}")

        html_objects.append(html.Div(html_sub_objects))

    return html_objects


def plan_menu(env_factory):
    html_objects = []
    for plan in env_factory.plans.all():
        if plan == env_factory.plans.current:
            display_class = "current_plan"
            style = {"background-color":"#ccc", "border":0, "margin":0, "padding": "10px 40px 10px 40px"}
        else:
            display_class = "archived_plan"
            style = {"background-color":"#eee", "border":0, "margin":0, "padding": "10px 40px 10px 40px"}

        print("\t","plan_menu: plan", plan.name, display_class)

        active = []
        if plan.id in env_factory.plans.active:
            active = [""]

        html_objects.append( html.Tr(
                [
                    html.Td(html.A(plan.name, id={"type":"plan", "id": plan.id}, n_clicks=0),
                         className=display_class, style=style),
                    html.Td(dcc.Checklist([""], active, id={"type":"plan_view", "id":plan.id}))
                ]
            ))

    return html.Tbody(html_objects)
```

3 DASH/Documentation

3.1 DASH/Documentation/interface.txt

```

### Interface Selector
@callback(
    Output("onboard_model", "options", allow_duplicate=True),
    Output("model_table_body", "children", allow_duplicate=True),
    Output("model_table_body", "value", allow_duplicate=True),
    Input("agent_interface", "value"),
    prevent_initial_call=True,
)
def choose_interface(value):
        return [o.__name__ for o in idx], Tb_children, []

### Model Selector
@callback(
    Output("model_table_head", "children", allow_duplicate=True),
    Output("model_table_subhead", "children"),
    Output("coa_update", "data", allow_duplicate=True),
    Input("onboard_model", "value"),
    prevent_initial_call=True,
)
def select_model(value):
    return table_head, table_subhead, True


### Add Action Button
@app.callback(
    Output("my-range-slider", "value"),
    Output("model_table_body", "children"),
    Input("AddNewCommand", "n_clicks"),
    State("my-range-slider", "value"),
    State("model_table_body", "children"),
    State("agent_interface", "value"),
)
def add_COA_command(val, slider_values, children, interface):
        return slider_values, children


### Slider Action
@callback(
    Output({"type": "actionstart", "index": ALL}, "value"),
    Output("coa_update", "data", allow_duplicate=True),
    Input("my-range-slider", "drag_value"),
    State("agent_interface", "value"),
    prevent_initial_call=True,
)
def update_output(drag_value, interface):
    return drag_value, True


### Parameter Update
@callback(
    Output("coa_update", "data", allow_duplicate=True),
    Input({"type": "actionparam", "index": ALL}, "value"),
    State("agent_interface", "value"),
    prevent_initial_call=True,
)
def update_coa_parameters(value, interface):
    return interface


### Process COA Update
@callback(
    Output("indicator-graphic", "figure"),
    Output("stats-graphic-vel", "figure"),
    Output("stats-graphic-fuel", "figure"),
    Output("stats-graphic-acc", "figure"),
    Input("coa_update", "data"),
    State("agent_interface", "value"),
)
def process_coa_update(value, interface):
    return fig, fig1, fig2, fig3


### Generate COA
@callback(
    Output("memory", "data"),
    Input("GenerateCOA", "n_clicks"),
    State("agent_interface", "value"),
    prevent_initial_call=True,
)
def generate_coa(value, interface):
    return "generated"


### Parameter Update
@callback(
    Output("model_table_body", "children", allow_duplicate=True),
    Output("my-range-slider", "value", allow_duplicate=True),
    Output("coa_update", "data", allow_duplicate=True),
    Input("memory", "data"),
    State("model_table_body", "children"),
    State("my-range-slider", "value"),
    State("agent_interface", "value"),
    prevent_initial_call=True,
)
def load_coa(coa_name, table_body, slider, interface):
    return table_body, slider_values, True
```

3.2 DASH/Documentation/network_view.py

```
# %%
import networkx as nx
import matplotlib.pyplot as plt

s = ""
with open("../app.py", "r") as f:
    in_callback = False
    callbacks = []
    s = ""

    for line in f:
        if in_callback:
            s += line

            if line.startswith("def"):
                in_callback = False
                callbacks.append((line , s))


        if line.startswith("@callback"):
            s = ""
            in_callback = True



# %%
info = dict()


for d, c in callbacks:
    Outputs = []
    Input = []
    State = []

    c = c.strip()
    cs = c.split(",")
    i = 0
    while i < len(cs):
        st = cs[i].strip()

        if st.startswith("Output"):
            target = st.split("(")[1].replace("'","").replace('"',"")
            atr = cs[i+1].replace("'","").replace('"',"").replace(")","")
            Outputs.append([target,atr])
            i+=1

        if st.startswith("Input"):
            target = st.split("(")[1].replace("'","").replace('"',"")
            atr = cs[i+1].replace("'","").replace('"',"").replace(")","")
            Input.append([target,atr])
            i+=1

        if st.startswith("State"):
            target = st.split("(")[1].replace("'","").replace('"',"")
            atr = cs[i+1].replace("'","").replace('"',"").replace(")","")
            State.append([target,atr])
            i+=1

        i+=1

    name = d.split()[1].split("(")[0]
    info[name] = {
                    "Input": Input,
                    "Output": Outputs,
                    "State": State
                }

info
# %%
edges = []
functions = []
inputs = []
outputs = []
states = []
for name, s in info.items():
    functions.append(name)

    for i in s["Input"]:
        inputs.append(i[0])
        edges.append( (i[0], name) )

    for i in s["Output"]:
        outputs.append(i[0])
        edges.append( (name, i[0]) )

    for i in s["State"]:
        states.append(i[0])
        edges.append( (i[0], name) )

edges
# %%
G = nx.Graph().to_directed()

# %%
color_map = []
for n in G:
    n = str(n)
    if n in functions:
        color_map.append("blue")
    elif n in inputs:
        color_map.append("green")
    elif n in outputs:
        color_map.append("red")
    elif n in states:
        color_map.append("orange")
    else:
        color_map.append("black")

color_map

# %%
for e in edges:
    G.add_edge(e[0], e[1])
# G = nx.from_edgelist(edges).to_directed()

f = plt.figure(figsize=[12,12])
ax = f.gca()
pos = nx.nx_agraph.graphviz_layout(G, prog="dot")
nx.draw(
    G,
    pos=pos,
    ax=ax,
    with_labels=False,
    node_color=color_map
)
text = nx.draw_networkx_labels(G, pos)
for _, t in text.items():
    t.set_rotation(20)

print(text)
# %%
```

4 DASH/assets

4.1 DASH/assets/typography.css

```
body {
    font-family: sans-serif;
}

h1, h2, h3, h4, h5, h6 {
    color: black
}

.resume {
    display: none;
}

.button4 {
    background-color: white; /* Green */
    border: none;
    color: black;
    padding: 15px 32px;
    text-align: left;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
    width: 100%;
  }

.button4:hover {
    background-color: #DDDDDD; /* Green */
    color: white;
  }


.float-container {
    border: 3px solid #fff;
    padding: 20px;
    width: 600px;
    margin:0;
    height:300px;
}

.float-child {
    width: 50%;
    height:100%;
    float: left;
    padding: 20px;
    border: 2px solid red;
}

.agent_card {
    float: left;
    padding: 10px;
    height: 150px;
    width: 100%;
    margin: auto;
  }

.action_cards {
    float: left;
    padding: 10px;
    width: 80%;
    margin: auto;
  }

.left_card {
    float: left;
    width: 25%;
    height: 100%;
    background-color: #aaa;
    margin:auto;
  }

.right_card {
    float: left;
    width: 70%;
    height: 100%;
    background-color: #bbb;
    margin:auto;
  }

  .coa_card {
    float: left;
    width: 15%;
    height: 100%;
    background-color: #bbb;
    margin:auto;
  }

.current_plan tr {
    background-color: #ccc;
}

.archived_plan tr {
    background-color: #eee;
}

.archived_plan tr:hover {
    background-color: #ddd;
}

/* Clear floats after the columns */
.agent_card_container:after {
    content: "";
    display: table;
    clear: both;
  }


/* Colors for sliders */

.rc-slider-handle-1 {
    border-color: #01befe;
}

.rc-slider-handle-1.rc-slider-handle-click-focused:focus {
    border-color: #01befe;
}

.rc-slider-handle-1:hover {
    border-color: #01befe;
}

/* ===================== */

.rc-slider-handle-2 {
    border-color: #ffdd00;
}

.rc-slider-handle-2.rc-slider-handle-click-focused:focus {
    border-color: #ffdd00;
}

.rc-slider-handle-2:hover {
    border-color: #ffdd00;
}

/* ===================== */

.rc-slider-handle-3 {
    border-color: #ff7d00;
}

.rc-slider-handle-3.rc-slider-handle-click-focused:focus {
    border-color: #ff7d00;
}

.rc-slider-handle-3:hover {
    border-color: #ff7d00;
}

/* ===================== */

.rc-slider-handle-4 {
    border-color: #ff006d;
}

.rc-slider-handle-4.rc-slider-handle-click-focused:focus {
    border-color: #ff006d;
}
```

```

.rc-slider-handle-4:hover {
    border-color: #ff006d;
}

/* ====================== */

.rc-slider-handle-5 {
    border-color: #adff02;
}

.rc-slider-handle-5.rc-slider-handle-click-focused:focus {
    border-color: #adff02;
}

.rc-slider-handle-5:hover {
    border-color: #adff02;
}

/* ====================== */

.rc-slider-handle-6 {
    border-color: #8f00ff;
}

.rc-slider-handle-6.rc-slider-handle-click-focused:focus {
    border-color: #8f00ff;
}

.rc-slider-handle-6:hover {
    border-color: #8f00ff;
}

/* ====================== */
```

5 examples/MAInspection

5.1 examples/MAInspection/Readme.md

```
# Multiagent Inspection

This example shows how to solve a simple satelite inspection problem in two steps:
* First we train an agent not to solve the problem, but to use the environment to train an agent to go
     from one waypoint to another.
* After this "waypointer"
agent is trained the director is then trained to select waypoints that easily solve the problem.

This example also demonstrates how to create custom agents, a custom coach environment, custom
     trajectory classes and custom trajectory visualizations.

<img src="Interface.png">
```

5.2 examples/MAInspection/app.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import sys
import glob
import yaml
sys.path.insert(0, "../../")

from DASH.app import *
from utilities.PZWrapper import PettingZooEnv

from examples.MAInspection.Environments import env as Multinspect
from examples.MAInspection.Environments.MAIcoach import MAI_COACH
import examples.MAInspection.Environments.MAIagents as MAIagents

# Logging
import logging
logging.basicConfig(level=logging.DEBUG, format="%(levelname)s:%(name)s:%(message)s")
logger = logging.getLogger(__name__)

pymunk_loggers = [logging.getLogger(name) for name in logging.root.manager.loggerDict if name.startswith("pymunk")]
numba_loggers = [logging.getLogger(name) for name in logging.root.manager.loggerDict if name.startswith("numba")]
for log_handler in pymunk_loggers + numba_loggers:
    log_handler.setLevel(logging.INFO)


def get_env():
    return PettingZooEnv(PZGame=Multinspect)

if __name__ == "__main__":
    print("app - ################### RELOADING DASH APP #########################")

    with open(os.path.join("Experiment", "train_director.yaml"), "r") as f:
        try:
            params = yaml.safe_load(f)
        except yaml.YAMLError as exc:
            print(exc)


    with open(os.path.join("Experiment", "dash_app.yaml"), "r") as f:
        try:
            params["actor_params"] = yaml.safe_load(f)
        except yaml.YAMLError as exc:
            print(exc)

    # external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']
    env_factory = COACHIntegration(
        env_creator=get_env,
        COACHEnvClass=MAI_COACH,
        parameters=params,
        agents_module=MAIagents,
        )

    ct.env_factory = env_factory

    locks = {
                "action_card": False
            }

    ## App Layout
    proxy_url = re.sub("{{port}}", "8050", os.environ["VSCODE_PROXY_URI"])
    proxy_url = re.sub(os.environ["ACEHUB_BASEURL"], "", proxy_url)
    app = Dash(serve_locally=True, requests_pathname_prefix=proxy_url)

    app.layout = app_layout(env_factory)

    app.run(debug=True)
```

5.3 examples/MAInspection/train_agents.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from __future__ import annotations
import sys
sys.path.insert(0, "../../")

import examples.MAInspection.Environments.env as TrainingEnv
import supersuit as ss
from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy
from stable_baselines3.common.callbacks import EvalCallback
from stable_baselines3.common.vec_env import VecVideoRecorder
from PIL import Image
import os
import time
import glob
from tqdm import tqdm
import numpy as np
import yaml
import copy

import torch
CORES = 7
torch.set_num_threads(CORES)
torch.set_num_interop_threads(CORES)

# Code adapted from https://pettingzoo.farama.org/tutorials/sb3/waterworld/

## We're going to train two policies, one that assumes a densely poisonous env
## and one that requires sparse exploration.

def train_butterfly_supersuit(
    env_fn,
    steps: int = 10_000,
    seed: int | None = 0,
    num_vec_envs=1,
    num_cpus=CORES,
    learning_rate=1e-3,
    batch_size=256,
    model_dir="",
    name="",
    **env_kwargs
):
    model_path = os.path.join(model_dir, name)
    os.makedirs(model_path,exist_ok=True)

    # Train a single model to play as each agent in a cooperative Parallel environment
    env = env_fn(**env_kwargs)
    env.reset(seed=seed)

    print(f"Starting training on {str(env.metadata['name'])}.")

    env = ss.pettingzoo_env_to_vec_env_v1(env)
    env = ss.concat_vec_envs_v1(env, num_vec_envs=num_vec_envs, num_cpus=num_cpus, base_class="stable_baselines3")

    # Note: Waterworld's observation space is discrete (242,) so we use an MLP policy rather than CNN
    model = PPO(
        MlpPolicy,
        env,
        verbose=3,
        learning_rate=learning_rate,
        batch_size=batch_size,
        tensorboard_log=os.path.join("./tensorboard_log/", name)
    )

    eval_env = env_fn(**env_kwargs)
    eval_env.reset(seed=seed)
    eval_env = ss.pettingzoo_env_to_vec_env_v1(eval_env)
    eval_env = ss.concat_vec_envs_v1(eval_env, num_vec_envs=1, num_cpus=num_cpus, base_class="stable_baselines3")

    eval_callback = EvalCallback(eval_env, verbose=1, eval_freq=1000)

    weight_name = f"{env.unwrapped.metadata.get('name')}_{time.strftime('%Y%m%d-%H%M%S')}"
    weight_path = os.path.join(model_path, weight_name)
    os.makedirs(weight_path,exist_ok=True)

    model_args = {
        "name": name,
        "load_class": "SB_PPOWaypointerActor",
        "env_params": env_kwargs
    }

    with open(os.path.join(weight_path, "params.yaml"), "w") as f:
        yaml.dump(model_args, f, default_flow_style=False)

    model.learn(total_timesteps=steps, callback=eval_callback)
    model.save(os.path.join(weight_path, "model"))

    print("Model has been saved.")
    print(f"Finished training on {str(env.unwrapped.metadata['name'])}.")

    env.close()

    return os.path.join(weight_path, "model")


def eval(
    env_fn,
    model_path,
    num_games: int = 5,
    render_mode: str | None = "rgb_array",
    render = True,
    **env_kwargs
):
    # Evaluate a trained agent vs a random agent
    env = env_fn(render_mode=render_mode, **env_kwargs)

    print(
        f"\nStarting evaluation on {str(env.metadata['name'])} (num_games={num_games}, render_mode={render_mode})"
    )

    print(model_path)

    model = PPO.load(model_path)

    total_rewards = []

    # Note: We train using the Parallel API but evaluate using the AEC API
    # SB3 models are designed for single-agent settings, we get around this by using he same model for every agent
    for i in tqdm(range(num_games)):
        obs, infos = env.reset(seed=i)
        cumm_rewards = {agent: 0 for agent in env.possible_agents}
        running = True
        frames = []

        while running:
            acts = dict()
            for role in env.agents:
                acts[role] = model.predict(obs[role], deterministic=True)[0]

            obs, rewards, terminations, truncations, infos = env.step(acts)

            for role in env.agents:
                cumm_rewards[role] += rewards[role]

            if all([a or b for a,b in zip(terminations.values(), truncations.values())]):
                running = False
                total_rewards += list(cumm_rewards.values())

            if render:
                frames.append(env.render())

        if render:
            imgs = [Image.fromarray(frame) for frame in frames]
            # duration is the number of milliseconds between frames; this is 40 frames per second
            model_dir = model_path.split(".")[0]
            imgs[0].save(model_dir + f"_eval_run_{i}.gif", save_all=True, append_images=imgs[1:], duration=500, loop=0)

    env.close()

    avg_reward = np.mean(total_rewards)
    std_reward = np.std(total_rewards)

    with open(model_path + ".txt","a") as f:
        f.writelines(f"\n\n{env_kwargs}\n")
        f.writelines(f"\t Avg reward: {avg_reward}, std: {std_reward}\n")
        f.writelines(f"\t Rewards: {total_rewards}\n")

    print(f"\t Avg reward: {avg_reward}, std: {std_reward}")
    return avg_reward


if __name__ == "__main__":

    with open("Experiment/train_agents.yaml", "r") as stream:
        try:
            experiments = yaml.safe_load(stream)
        except yaml.YAMLError as exc:
            print(exc)

    STEPS = 2000

    # Train Waypoint Agent
    model_paths = dict()
    for name, experiment in experiments.items():
        model_paths[name] = train_butterfly_supersuit(
            env_fn=TrainingEnv.parallel_env,
            steps=STEPS,
            model_dir=name,
            **experiment["env_params"]
            )

    for name, model_path in model_paths.items():#, model_path_explore]:
        eval(env_fn=TrainingEnv.parallel_env,
            model_path=model_path,
            **experiments[name]["env_params"]
            )
```

5.4 examples/MAInspection/train_director.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from __future__ import annotations
import os
import time
import sys
import glob
from tqdm import tqdm
import numpy as np
from PIL import Image
import yaml

sys.path.insert(0, "../../")

from utilities.PZWrapper import PettingZooEnv
from examples.MAInspection.Environments.MAIcoach import MAI_COACH
from examples.MAInspection.Environments import env as Multinspect
from env import COACH_PettingZoo

from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy
from stable_baselines3.common.callbacks import EvalCallback
import supersuit as ss

import torch
CORES = 6
torch.set_num_threads(CORES)
torch.set_num_interop_threads(CORES)

import logging
logging.basicConfig()
logging.getLogger().setLevel(logging.INFO)
logger = logging.getLogger(__name__)

pymunk_loggers = [logging.getLogger(name) for name in logging.root.manager.loggerDict if name.startswith("pymunk")]
for log_handler in pymunk_loggers:
    log_handler.setLevel(logging.INFO)

# SB code adapted from https://pettingzoo.farama.org/tutorials/sb3/waterworld/

## We're going to train two policies, one that assumes a densely poisonous env
## and one that requires sparse exploration.
# %%
def train(
    env_fn,
    steps: int = 10_000,
    seed: int | None = 0,
    num_vec_envs=1,
    num_cpus=1,
    learning_rate=1e-3,
    batch_size=256,
    model_dir=""
):
    os.makedirs(model_dir,exist_ok=True)

    # Train a single model to play as each agent in a cooperative Parallel environment
    env = env_fn()
    env.reset(seed=seed)
    print(env.possible_agents)

    logger.info(f"Starting training on {str(env.metadata['name'])}.")

    env = ss.pettingzoo_env_to_vec_env_v1(env)
    env = ss.concat_vec_envs_v1(env, num_vec_envs=num_vec_envs, num_cpus=num_cpus, base_class="stable_baselines3")

    model = PPO(
        MlpPolicy,
        env,
        verbose=3,
        learning_rate=learning_rate,
        batch_size=batch_size,
        tensorboard_log=os.path.join("./tensorboard_log/", model_dir.split("/")[-1])
    )

    eval_env = env_fn()
    eval_env = ss.pettingzoo_env_to_vec_env_v1(eval_env)
    eval_env = ss.concat_vec_envs_v1(eval_env, num_vec_envs=num_vec_envs, num_cpus=num_cpus, base_class="stable_baselines3")
    eval_callback = EvalCallback(eval_env, verbose=1, eval_freq=500)

    model.learn(total_timesteps=steps, callback=eval_callback)

    model_name = f"{env.unwrapped.metadata.get('name')}_{time.strftime('%Y%m%d-%H%M%S')}"
    model_path = os.path.join(model_dir, model_name)
    model.save(model_path)

    logger.info("Model has been saved.")
    logger.info(f"Finished training on {str(env.unwrapped.metadata['name'])}.")

    env.close()

    return model_path + ".zip"

def eval(
    env_fn,
    model_path,
    num_games: int = 10,
    render = False,
    **env_kwargs
):
    # Evaluate a trained agent vs a random agent
    env = env_fn(**env_kwargs)

    logger.info(
        f"\nStarting evaluation on {str(env.metadata['name'])} (num_games={num_games}, render={render})"
    )

    model = PPO.load(model_path)

    cum_rewards = []

    # Note: We train using the Parallel API but evaluate using the AEC API
    # SB3 models are designed for single-agent settings, we get around this by using he same model for every agent
    for i in tqdm(range(num_games)):
        if render:
            env.coa_env.start_rendering()

        obs, info = env.reset(seed=i)

        rewards = {agent: 0 for agent in env.possible_agents}
        running = True
        j = 0
        while running:
            j += 1
            act = {"director": model.predict(obs["director"], deterministic=False)[0]}

            logger.debug("Obs From Director: %s", obs["director"]) # DEBUG
            logger.debug("Action From Director: %s", act) # DEBUG

            obs, reward, term, trunc, info = env.step(act)
            for a in env.agents:
                rewards[a] += reward[a]

            if all([a or b for a,b in zip(term.values(), trunc.values())]):
                for a in env.agents:
                    cum_rewards.append(rewards[a])

                running = False

        if render:
            frames = env.coa_env.state.trajectory.frames
            imgs = [Image.fromarray(frame) for frame in frames]
            # duration is the number of milliseconds between frames; this is 40 frames per second
            model_dir = model_path.split(".")[0]
            imgs[0].save(model_dir + f"_eval_run_{i}.gif", save_all=True, append_images=imgs[1:], duration=500, loop=0)

    env.close()

    avg_reward = np.mean(cum_rewards)
    std_reward = np.std(cum_rewards)

    with open(model_path + ".txt","a") as f:
        f.writelines(f"\n\n{env_kwargs}\n")
        f.writelines(f"\t Avg reward: {avg_reward}, std: {std_reward}\n")
        f.writelines(f"\t Rewards: {cum_rewards}\n")

    logger.info(f"\t Avg reward: {avg_reward}, std: {std_reward}")
    return avg_reward


def get_env():
    return PettingZooEnv(PZGame=Multinspect)

if __name__ == "__main__":
    WAYPOINTER_DIR = "waypointer"
    WAYPOINTER_PATH = min(glob.iglob('waypointer/*'), key=os.path.getctime)

    WAYPOINTER_PATH = min(glob.iglob(os.path.join(WAYPOINTER_PATH, '*.zip')), key=os.path.getctime)

    with open(os.path.join("Experiment", "train_director.yaml"), "r") as stream:
        try:
            params = yaml.safe_load(stream)
        except yaml.YAMLError as exc:
            print(exc)

    for agent, param in params["COACH_params"]["Agents"].items():
        param["params"]["policy_path"] = WAYPOINTER_PATH

    env = COACH_PettingZoo(env_creator=get_env, COACHEnvClass=MAI_COACH)

    def get_env_pz():
        env = COACH_PettingZoo(env_creator=get_env, COACHEnvClass=MAI_COACH)
        env.augment(params)
        env.reset()
        return env

    model_path = train(
        get_env_pz,
        steps=500,
        seed=0,
        model_dir="director"
        )

    model_path = min(glob.iglob(os.path.join('director', '*.zip')), key=os.path.getctime)
    rew = eval(
        get_env_pz,
        model_path=model_path,
        num_games=1,
        render=True
        )

    with open(os.path.join("Experiment", "dash_app_template.yaml"), 'r') as f:
        with open(os.path.join("Experiment", "dash_app.yaml"), 'w') as g:
            for line in f.readlines():
                line = line.replace("<WAYPOINTER_PATH>", WAYPOINTER_PATH)
                line = line.replace("<DIRECTOR_PATH>", model_path)
                g.write(line)

# %%
```

6 examples/MAInspection/Environments

6.1 examples/MAInspection/Environments/MAIagents.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import numpy as np
import copy
import logging
from gymnasium.spaces import Box
logger = logging.getLogger(__name__)
from numpy.random import PCG64DXSM, Generator
from agents import *

class WaypointInterface(TrivialInterface):
    def __init__(self, role, env):
        super().__init__(role, env)

        desc = ["rel frame x", "rel frame y", "rel frame z"]

        policies = ActionBox(
            low=-np.ones(3)*env._MAX_OUTER_PERIMETER.value,
            high=np.ones(3)*env._MAX_OUTER_PERIMETER.value,
            shape=(3,),
            description=desc
        )

        self.action_dictionary = {"Waypoint": policies}

    def equals(self, other):
        if type(other) is type(self):
            return self.action_dictionary["Waypoint"].equals(other.action_dictionary["Waypoint"])
        return False


class SB_PPOWaypointActor(BasicActor):
    InterfaceType = WaypointInterface
    def __init__(self, role, policy_path: str, interface:WaypointInterface=None):
        super().__init__(role)

        if policy_path[-3:] == "zip":
            policy_path = policy_path.split(".")[0]
        self.policy = PPO.load(policy_path)
        self.next_waypoint = None

    def __str__(self):
        return f"SB_PPOWaypointActor: {self.role}"

    def get_action(self, obs, t, mean_mode=False):
        bad_command = False
        acting = False
        coa_done = False
        action = copy.copy(self.none_action)

        if t in self.coa.timeline.keys():
            ## Start new waypoint
            e = self.coa.get(time = t, label = "Waypoint")
            # self.time_remaining = e.parameters
            self.next_waypoint = e.parameters
            logger.debug("Agent: start new waypoint %s", e) # DEBUG


        if self.next_waypoint is not None:
            i = self.env.possible_agents.index(self.role)

            pos = self.env.orb[self.role].r.to(self.env._OU_DIS).value
            pos_c = self.env.orb["chief"].r.to(self.env._OU_DIS).value

            if self.env.OBSERVATION_FRAME == "Hills":
                frame = self.env.hills_frame(self.env.orb[self.role])
            else:
                frame = self.env.ori[i]

            rel_waypt = frame.T @ (self.next_waypoint + pos_c - pos).reshape(-1,1)

            agent_obs = np.concatenate([obs[:6*self.env.num_deputies],rel_waypt.reshape(-1)])
            if np.linalg.norm(rel_waypt) < self.env._WAYPOINT_ARRIVAL_PROX.value:
                acting = False
                coa_done = True
                self.next_waypoint = None
            else:
                acting = True
                action, _ = self.policy.predict(agent_obs, deterministic=True)

        return action, {
            "acting": acting,
            "coa_done": coa_done,
            "bad_command": bad_command,
        }

    def reset(self, env):
        super().reset(env)
        self.time_remaining = -1
        self.current_policy = None

        self.interface = SB_PPOWaypointActor.InterfaceType(
            self.role,
            env
        )

        self.env = env

        if self.reference_interface is not None:
            if not self.interface.equals(self.reference_interface):
                raise Exception("Actor interface does not match reference interface.")


classes = [
    cls_obj
    for cls_name, cls_obj in inspect.getmembers(sys.modules[__name__])
    if inspect.isclass(cls_obj) and cls_obj.__module__ == __name__
]

Interfaces = {}
for c in classes:
    # BasicActor
    if issubclass(c, TrivialInterface):
        Interfaces[c] = []

Agents = {}
for c in classes:
    # BasicActor
    if issubclass(c, BasicActor):
        Interfaces[c.InterfaceType].append(c)
        Agents[c.__name__] = c
```

6.2 examples/MAInspection/Environments/MAIcoach.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import sys
sys.path.insert(0, "../../")

import numpy as np
import copy
import logging
from gymnasium.spaces import Box

logger = logging.getLogger(__name__)
from collections import OrderedDict
import matplotlib.pyplot as plt
from numpy.random import PCG64DXSM, Generator
import matplotlib as mpl
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from mpl_toolkits.mplot3d.art3d import Line3DCollection
import plotly.express as px
import plotly.graph_objects as go
import pandas as pd
import examples.MAInspection.Environments.MAIagents as agents
import io
from PIL import Image

# %matplotlib inline
from coach import (
    COACHEnvironment,
    COA,
    State,
    Trajectory,
    BaseParameterGenerator,
    SeededParameterGenerator,
    CommunicationSchedule,
    Timeline
)

# %%
def plot_orbits(orbits):
    trace = go.Scatter3d(
        x=orbits[:, 0],
        y=orbits[:, 1],
        z=orbits[:, 2],
        mode="markers",
        marker=dict(color=orbits[:, 3:], size=5),
    )

    return trace

    # ax.scatter(orbits[:,0], orbits[:,1], orbits[:,2], c = orbits[:,3:], alpha = alpha)


def plot_points(p_array, c):
    # c = np.zeros([seen.shape[0], 3])
    c = mpl.colors.to_rgba_array(c)
    tmp = np.zeros([p_array.shape[0], 7])
    tmp[:, :3] = p_array
    tmp[:, 3:] = c

    t_seen = go.Scatter3d(
        x=tmp[:, 0],
        y=tmp[:, 1],
        z=tmp[:, 2],
        mode="markers",
        marker=dict(color=tmp[:, 3:], size=5),
    )

    return t_seen


def ms(x, y, z, radius, resolution=20):
    """Return the coordinates for plotting a sphere centered at (x,y,z)"""
    u, v = np.mgrid[0 : 2 * np.pi : resolution * 2j, 0 : np.pi : resolution * 1j]
    X = radius * np.cos(u) * np.sin(v) + x
    Y = radius * np.sin(u) * np.sin(v) + y
    Z = radius * np.cos(v) + z
    return (X, Y, Z)


def plotly_fig2array(fig):
    # convert Plotly fig to  an array
    fig_bytes = fig.to_image(format="png")
    buf = io.BytesIO(fig_bytes)
    img = Image.open(buf)
    return np.asarray(img)


class MAI_Trajectory(Trajectory):
    def __init__(self, env, initial_return, intial_frame=None):
        self.pos = []
        self.ori = []
        self.pts = []
        self.unobserved_points = []
        self.cumm_rewards = []

        super().__init__(env, initial_return, intial_frame=intial_frame)

    def add(self, env, action, step_return, agent_info, frame=None):
        super().add(env, action, step_return, agent_info, frame=frame)
        frame = env.hills_frame(env.orb["chief"])

        # Here: would be frame.T because we're mapping from absolute coords to
        # chief frame, so right mul is the same
        # for orientation, recall that self.ori[k, axis, :] is the k'th
        # agent's axis, with axis 0 being the heading.

        self.pos.append(copy.copy(env.pos) @ frame)
        self.ori.append(copy.copy(env.ori) @ frame)
        self.pts.append(copy.copy(env.pts) @ frame)
        self.unobserved_points.append(copy.copy(env.unobserved_points))
        self.cumm_rewards.append(copy.copy(env.cum_rewards))


class MAI_COACH(COACHEnvironment):
    agent_selection = agents

    def __init__(
        self,
        env_creator: callable,
        parameter_generator:BaseParameterGenerator,
        agents=dict(),
        fill_random=True,
        comm_schedule:Timeline=None,
        seed=6234235,
        TrajectoryClass = MAI_Trajectory,
    ):
        super().__init__(env_creator,
            parameter_generator,
            agents=agents,
            fill_random=fill_random,
            comm_schedule=comm_schedule,
            seed=seed,
            TrajectoryClass = TrajectoryClass
        )

    def plot_trajectory_component(
        self,
        traj,
        coas=None,
        env=None,
        plan=None,
        render_mode="plotly",
        simulate_data=False
    ):
        if plan:
            coas = plan.coas

        players = traj.players
        cmap = plt.get_cmap("tab10")
        event_cmap = plt.get_cmap("Set1")
        player_to_color = {p: i / 10 for i, p in enumerate(players)}

        game_len = len(traj)
        num_points = env.num_points
        chief_perim = env._CHIEF_PERIMETER

        # 3 dim: pos, 4 dim: color
        xs = np.zeros([game_len, len(players), 7])
        xs[:,:,:3] = np.array([pos.to(env._OU_DIS).value for pos in traj.pos])

        # Plot Cone Information
        CONE_FREQ = 5
        CONE_OPATICY = 0.5

        n_cones = int(np.ceil(game_len/CONE_FREQ))
        cones = np.zeros([n_cones*len(players), 10])

        # Stack is going to concatenate long the first axis, combine all
        # players info into one stack
        cones[:, :3] = np.vstack(xs[::CONE_FREQ,:,:3])
        cones[:, 3:6] = np.vstack(np.array(traj.ori)[::CONE_FREQ,:,:])[:,0,:]
        pcolors = [cmap(player_to_color[p]) for p in players]
        cones[:, 6:] = np.repeat(np.array(pcolors),n_cones, axis=0)
        cones[:, -1] = CONE_OPATICY

        # Seperate into drifts and burns
        for p_idx, p in enumerate(players):
            c = cmap(player_to_color[p])

            if p in coas.keys():
                coa = coas[p]
            else:
                coa = COA()

            event_time_left = 0

            for i in range(game_len):
                if i in coa.timeline.keys():
                    for event in coa.timeline[i]:
                        # event = coa.timeline[i][action]
                        event_time_left = event.parameters[0]
                        print(event.label)
                        c = event_cmap(event.id)
                        print(c)

                if event_time_left <= 0:
                    c = cmap(player_to_color[p])

                xs[i,p_idx,3:] = c
                event_time_left -= 1

        ## Get Seen Points
        pts = traj.pts[0]
        u_set = traj.unobserved_points[0]

        unseen = np.zeros(pts.shape[0], dtype=bool)
        unseen[list(u_set)] = True

        seen = pts[~unseen]
        unseen = pts[unseen]

        traces = []
        traces.append(plot_orbits(np.vstack(xs)))

        if unseen.shape[0] > 0:
            traces.append(plot_points(unseen, "#000000"))

        if seen.shape[0] > 0:
            traces.append(plot_points(seen, "#FFFFFF"))

        # # Draw Chief Sphere
        (x_pns_surface, y_pns_surface, z_pns_suraface) = ms(0, 0, 0, chief_perim)
        traces.append(
            go.Surface(
                x=x_pns_surface,
                y=y_pns_surface,
                z=z_pns_suraface,
                opacity=0.5,
                showscale=False,
            )
        )

        # View Cones:
        traces.append(
            go.Cone(
                x=cones[:, 0],
                y=cones[:, 1],
                z=cones[:, 2],
                u=cones[:, 3],
                v=cones[:, 4],
                w=cones[:, 5],
                opacity=0.1,
                showscale=False,
                showlegend=False,
                sizemode="absolute",
                sizeref=10,
                anchor="tip",
            )
        )

        fig = go.Figure(data=traces)
        fig.update_layout(
            scene_aspectmode="cube",
            showlegend=False,
            scene=dict(
                xaxis=dict(
                    nticks=4,
                    range=[-300, 300],
                ),
                yaxis=dict(
                    nticks=4,
                    range=[-300, 300],
                ),
                zaxis=dict(
                    nticks=4,
                    range=[-300, 300],
                ),
            ),
            width=700,
            margin={"l": 40, "b": 40, "t": 10, "r": 0},
            hovermode="closest",
        )

        if render_mode == "plotly":
            return fig

        if render_mode == "rgb_array":
            return plotly_fig2array(fig)


#################################################################
## Example Usage
#################################################################

if __name__ == "__main__":
    from examples.MAInspection.env import MultInspect

    ## Environemntal Parameters
    env_params = {"_OBS_REWARD":.01, "num_deputies": 2}

    # Actor Parameters
    COACH_params = {
        "Agents": {
            "player_0": {
                "class_name":"SB_PPOWaypointActor",
                "params": {"policy_path": "/root/coach/examples/MAInspection/waypointer/MAInspect_20240205-234515.zip"}    # If the agent has setup parameters
                },
            "player_1": {
                "class_name":"SB_PPOWaypointActor",
                "params": {"policy_path": "/root/examples/MAInspection/waypointer/MAInspect_20240205-234515.zip"}    # If the agent has setup parameters
                },
        }
    }

    # Create actors from parameters
    agent_dict = dict()
```

```python
    if "Agents" in COACH_params.keys():
        for role, agent in COACH_params["Agents"].items():
            agent_class = MAI_COACH.agent_selection.Agents[agent["class_name"]]
            agent_dict[role] = agent_class(role, **agent.get("params", dict()))

    # Example communication scheudle.
    comms = CommunicationSchedule.repeating(checkin_frequency=10)

    # Wrap parameters in parameter generator
    parameter_generator = SeededParameterGenerator(23423, env_params)

    def env_creator():
        return MultInspect(**env_params)

    env = MAI_COACH(
        env_creator = MultInspect,
        parameter_generator = parameter_generator,
        comm_schedule = comms,
        fill_random = True,
        agents = agent_dict
    )

    for agt in env.agents.values():
        print(agt.interface)

    for i in range(2):
        env.step()
    traj = env.state.trajectory
    coas = {role: agt.coa for role, agt in env.agents.items()}
    fig = env.plot_trajectory_component(traj, coas, env=env.env)
    fig.show()
# %%
```

6.3 examples/MAInspection/Environments/env.py

```
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# %%

"""Multi-Inspection Environment"""

import argparse  # for type hinting
import copy
import logging
from collections.abc import Callable  # for type hinting
from typing import Any, Optional, TypeVar, Union, cast, NewType

import gymnasium as gym
import imageio
import matplotlib  # type: ignore[import]
import matplotlib.lines as mlines  # type: ignore[import]
import matplotlib.pyplot as plt  # type: ignore[import]
import numpy as np

# from pettingzoo.utils.env import ParallelEnv
from matplotlib.backends.backend_agg import (
    FigureCanvasAgg as FigureCanvas,  # type: ignore[import]
)
from matplotlib.collections import LineCollection
from mpl_toolkits.mplot3d.art3d import Line3DCollection  # type: ignore[import]
from numpy.random import PCG64DXSM, Generator
from scipy.spatial.transform import Rotation  # type: ignore[import]
from scipy.stats import multivariate_normal, ortho_group  # type: ignore[import]

import sys

from numpy.typing import NDArray
FloatArray = NDArray[np.float_]
RenderFrame = NDArray[np.uint8]
Role = NewType("Role", str)

from astropy import units as u
from poliastro.maneuver import Maneuver
from poliastro.bodies import Earth, Mars, Sun
from poliastro.twobody import Orbit

matplotlib.use("agg")
logger = logging.getLogger(__name__)

from pettingzoo import ParallelEnv

########################################################
## Factories
########################################################
def get_env_class(args: argparse.Namespace) -> Callable[..., Any]:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env objects
    """
    return MultInspect

def unit(v):
    if len(v.shape) == 1:
        v = v.reshape(1,-1)
    return v/np.linalg.norm(v,axis=1)[:,None]

def proj(v, onto):
    return onto * np.dot(v, onto) / np.dot(onto, onto)

def frame(r, v):
    r = unit(r)
    v = unit(v)
    n = np.cross(r,v, axis=1)

    return np.stack([r,v,n],axis=2)

def parallel_env(**kwargs):
    return MultInspect(**kwargs)

class MultInspect(ParallelEnv):
    """
    We assuime six axis movement:
        action space: [X-Thrust, Y-Thrust, Z-Thrust,
                       X-Clockwise Torque, Y-Clockwise Torque, Z-Clockwise Torque]

    Some notes on linear algebra conventions and frames:

    All frames have columns as vectors in the background coordinate system R, so
    for any frame M, M * v maps v to R. To traslate back, a vector in w in R is mapped to
    M^T * w.

    A good rule of thumb here is to understand every matrix as having units
    xf, where x is standard coords and f is frame coords. Since frames are orthonormal
    M^-1 = M^T. If we have a vcetor in frame 1 coords and we want it in frame 2
    coords, we can just pass through the standard coords: M_1: v_f1 -> v_x,
    M_2^T: v_x -> v_f2, so M_2^T M_1 * v_f1 = v_f2 is a appropreate transform

    A source for physics: http://control.asu.edu/Classes/MMAE441/Aircraft/441Lecture9.pdf

    We're implementing a Petting Zoo interface, details can be found here:
    https://pettingzoo.farama.org/api/parallel/#parallelenv


    Attributes
    -------------

    agents: list[AgentID]
        A list of the names of all current agents, typically integers. May changed as environment progresses
    num_agents: int
        The length of the agents list.
    possible_agents: list[AgentID]
        A list of all possible_agents the environment could generate. Equivalent to the list of agents in the observation and action spaces. This cannot be changed through play or resetting.
    max_num_agents: int
        The length of the possible_agents list.
    observation_spaces: dict[AgentID, gym.spaces.Space]
        A dict of the observation spaces of every agent, keyed by name. This cannot be changed through play or resetting.
    action_spaces: dict[AgentID, gym.spaces.Space]
        A dict of the action spaces of every agent, keyed by name. This cannot be changed through play or resetting.


    Methods
    -------------

    step(actions: dict[str, ActionType])     tuple[dict[str, ObsType], dict[str, float], dict[str, bool], dict[str, bool], dict[str, dict]]
        Receives a dictionary of actions keyed by the agent name.
        Returns the observation dictionary, reward dictionary, terminated dictionary, truncated dictionary and info dictionary, where each dictionary is keyed by the agent.
    reset(seed: int | None = None, options: dict | None = None)     dict[str, ObsType]
        Resets the environment.
    seed(seed=None)
        Reseeds the environment (making it deterministic).
    render()     None | np.ndarray | str | list
        Displays a rendered frame from the environment, if supported.
    close()
        Closes the rendering window.
    state()     ndarray
        Returns the state.
    observation_space(agent: str)     Space
        Returns the observation space for given agent.
    action_space(agent: str)     Space
    """

    metadata = {"render_modes": ["human", "rgb_array"], "render_fps": 50, "name": "MAInspect"}


    def __init__(self, render_mode="rgb_array", **kwargs) -> None:
        """ """
        self.augment(params = kwargs)
        self.render_mode = render_mode

    def parallel_env(self, **kwargs):
        return self

    def _setup_gym_env(self, env_params: dict) -> None:
        ## Setup parameters. All of this may be overwritten by the
        ## params dictionary if the appropreate key exists.
        ## The list below provides the default parameters

        self.verbose: bool = False
        self.six_axis: bool = True
        self.num_deputies: int = 3
        self.max_episode_steps: int = 500
        self.num_points: int = 20

        self.MAXTIME = 1 << u.h
        self.TIME_PER_STEP = 1 << u.min

        # Compatability
        # This disable some agents terminating before others.
        self._SB_SAFTY_MODE = True

        # Game Mode:
        self._TRAIN_WAYPOINTER = True
        self._WAYPOINT_ARRIVAL_PROX = 10 << u.m
        self._WAYPOINT_ARRIVE_REWARD = 1

        # Initial Conditions
        self.INIT_ALT = 700 << u.km
        self.INIT_ALT_OFFSET = 50 << u.m
        self.RAAN = 0 << u.deg ## Angle Around the Circular Orbit
        self.ARGLAT = 0 << u.deg ## Latitude updown
        self.INC = 0 << u.deg ## Direction of orbit
        self._STARTING_DISTANCE_FROM_CHIEF = 150 << u.m

        self.offset_angle = (1/10)*(self._STARTING_DISTANCE_FROM_CHIEF /
             self.INIT_ALT).decompose().value << u.rad

        # Action Frame
        self.ACTION_FRAME = "Hills"          # Options should be Hills and Orientation
        self.OBSERVATION_FRAME = "Hills"     # Options should be Hills and Orientation
        self._OU_DIS = u.m
        self._OU_TIM = u.s
        self._OU_VEL = self._OU_DIS/self._OU_TIM

        self._CHIEF_PERIMETER: float = 50 << u.m
        self._DEPUTY_PERIMETER: float = 150  << u.m
        self._DEPUTY_MASS: float = 1 << u.kg
        self._DEPUTY_RADIUS: float = 1 << u.m
        self._DEPUTY_THRUST_COEEF: float = 0.01

        self._SIM_OUTER_PARAMETER: float = 200 << u.m #
        self._MAX_OUTER_PERIMETER: float = 300 << u.m # If you leave here, you truncate
        self._STARTING_VEL_NORM: float = 0 << u.m/u.s # 0.001

        self._TIME_PER_STEP: float = 90 << u.s
        self._DELTA_V_PER_THRUST: float = 0.1 << u.m/u.s
        self._DELTA_T: float = 90 << u.s
        self._TAU_per_THRUST: float = 0.1
        self._USE_ANGULAR_MOMENTUM = False

        self._OBS_REWARD: float = 0.02  # 20
        self._REWARD_FOR_SEEING_ALL_POINTS: float = 1  # 200
        self._CRASH_REWARD: float = 0
        self._REWARD_PER_STEP: float = 0.0   # -.1  # Reward per tick
        self._REWARD_FOR_LEAVING_PARAMETER: float = 0
        self._REWARD_FOR_LEAVING_OUTER_PERIMETER: float = 0

        self._SOLID_CHIEF: bool = True
        self._MAX_BURN: float = 10

        self._REW_COV_MTRX = 300**2
        self._PROX_RWD_SCALE = 100000

        # Visualization
        self._VIS_NUM_CONE_LINES = 8
        self._VIS_CONE_LEN = 40 << u.m

        # Vision Cone:
        self._MIN_VISION: float = 0 << u.m
        self._MAX_VISION: float = np.inf << u.m
        self._VISION_ARC: float = np.pi / 8 << u.rad

        # Apply parameters from input
        self._apply_parameters(params=env_params)
        if hasattr(env_params,"args"):
            self.master_seed: int = env_params.args.master_seed
        else:
            self.master_seed: int = 487924
        self.seed(seed=self.master_seed)


        if self._TRAIN_WAYPOINTER:
            self.num_points = self.num_deputies

        # Check parameters from input
        assert self._MIN_VISION >= 0, "_MIN_VISION must be >= 0"
        assert self._MAX_VISION > self._MIN_VISION, "_MAX_VISION must be > _MIN_VISION"
        assert (self._VISION_ARC >= 0) and (
            self._VISION_ARC <= np.pi << u.rad
        ), "_VISION_ARC must be [0, pi]"

        ## Setup actual Gym Env based on the above
        self.agents: list[Role] = [Role(f"player_{i}") for i in range(self.num_deputies)]

        # Agent Velocity, Other Dep. Rel Position and Vel, Point Rel Position
        self._VISION: dict[Role, list[float]] = {
            role: [self._VISION_ARC, self._MIN_VISION, self._MAX_VISION]
            for role in self.agents
        }

        in_size: int = (
            6 + 6 * (self.num_deputies - 1) + 3 * self.num_points + self.num_points
        )

        if self._TRAIN_WAYPOINTER:
            raw_low_values: list[float] = [
                -self._MAX_OUTER_PERIMETER.value * np.ones(3),  # Chief Position
                -self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * self.num_deputies),  # Deputy velocity
                -2
                * self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * (self.num_deputies - 1)),  # Dep. Positions, except me
                -2
                * self._MAX_OUTER_PERIMETER.value
                * np.ones(3),  # Single Waypoint,
            ]

            raw_high_values: list[float] = [
                self._MAX_OUTER_PERIMETER.value * np.ones(3),  # Chief Position
                self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * self.num_deputies),  # Deputy velocity
                2
                * self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * (self.num_deputies - 1)),  # Dep. Positions, except me
                2
                * self._MAX_OUTER_PERIMETER.value
                * np.ones(3),  # Single Waypoint,
            ]
        else:
            raw_low_values: list[float] = [
                -self._MAX_OUTER_PERIMETER.value * np.ones(3),  # Chief Position
                -self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * self.num_deputies),  # Deputy velocity
                -2
                * self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * (self.num_deputies - 1)),  # Dep. Positions, except me
                -2
                * self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * (self.num_points)),  # Point Pos,
                np.zeros(self.num_points),
            ]

            raw_high_values: list[float] = [
                self._MAX_OUTER_PERIMETER.value * np.ones(3),  # Chief Position
                self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * self.num_deputies),  # Deputy velocity
                2
                * self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * (self.num_deputies - 1)),  # Dep. Positions, except me
                2
                * self._MAX_OUTER_PERIMETER.value
                * np.ones(3 * (self.num_points)),  # Point Pos,
                np.ones(self.num_points),
            ]

        low = np.concatenate(raw_low_values, dtype=np.float32)
        high = np.concatenate(raw_high_values, dtype=np.float32)

        obs_space = gym.spaces.Box(low=low, high=high, dtype=np.float32)

        if self.six_axis:
            low = np.array([-10, -10, -10, -np.pi, -np.pi, -np.pi], dtype=np.float32)
            act_space = gym.spaces.Box(low=low, high=-low, dtype=np.float32)
        else:
            # Thrust, rotate Left/Right, rotate UP/Down
            act_space = gym.spaces.Box(
                low=-np.ones(3, dtype=np.float32),
                high=np.ones(3, dtype=np.float32),
                dtype=np.float32,
            )

        self.possible_agents = self.agents

        self.observation_spaces = {role: obs_space for role in self.possible_agents}
        self.action_spaces = {role: act_space for role in self.possible_agents}

        self.ori = np.zeros([self.num_deputies, 3, 3])       # Depute Orientation relative to absolute earth frame
        self.rot = np.zeros([self.num_deputies, 3])          # Depute Angular Momentum

        self.pos = np.zeros([self.num_deputies, 3])          # Depute Position Holder
        self.vel = np.zeros([self.num_deputies, 3])          # Depute Velocity Holder

        self.chief_frame = np.zeros([3, 3])                 # Current frame for the cheif relative to absolute earth frame
        self.frames = np.zeros([self.num_deputies, 3, 3])    # Current frame for the deputies relative to absolute earth frame
        self.pts = np.zeros([self.num_deputies, 3])          # Chief Inspection Points
        self.nor = np.zeros([self.num_deputies, 3])          # Chief Inspection Normals

        self.sim_steps = 0

        self.unobserved_points = set(range(self.num_points))

        self.render_path: Optional[str] = None
        self.frame_store: list[RenderFrame] = []
        self.cum_rewards = np.zeros(self.num_deputies)

    def _apply_parameters(self, params: dict) -> None:
        for k, v in params.items():
            self.__dict__[k] = copy.copy(v)

    def augment(self, params: dict) -> None:
        self._setup_gym_env(env_params=params)

    def seed(self, seed: int) -> None:
        self.np_random = Generator(PCG64DXSM(seed=seed))

    def observation_space(self, agent: Role) -> gym.spaces.Box:
        return self.observation_spaces[agent]

    def action_space(self, agent: Role) -> gym.spaces.Box:
        return self.action_spaces[agent]

    ## Reset
    def reset(
        self,
        *,
        seed: Optional[int] = None,
        options: Optional[dict[Any, Any]] = None,
        render_path: Optional[str] = None,
    ) -> tuple[dict[Role, Any], dict[Role, dict[str, Any]]]:
        """ """
        self.sim_steps = 0

        # Reset Seed
        local_seed = seed if (seed is not None) else self.master_seed
        self.seed(seed=local_seed)
        local_og = ortho_group(dim=3, seed=self.np_random)

        # Create Orbits for centers of mass
        self.orb = dict()
        self.orb_hist = []

        # Construct Chief
        self.orb[f"chief"] = Orbit.circular(Earth, alt=self.INIT_ALT)
        if self._TRAIN_WAYPOINTER:
            self.pts = self._random_sphere(self.num_points, radius=self._DEPUTY_PERIMETER)
        else:
            self.pts = self._random_sphere(self.num_points, radius=self._CHIEF_PERIMETER)

        self.chief_angular_momentum = None
        # self.chief_angular_momentum = Rotation.from_euler("xyz", [1,0,0]).as_matrix()

        # Construct Deputies
        self._active = np.array([role in self.agents for role in self.possible_agents])

        # Deputy Orbits
        for i in range(self.num_deputies):
            self.orb[f"player_{i}"] = Orbit.circular(Earth, alt=self.INIT_ALT + i*self.INIT_ALT_OFFSET,
                 raan = self.offset_angle)

        self.ori = np.stack([local_og.rvs() for i in range(self.num_deputies)])
        self.rot = self._random_sphere(self.num_deputies, radius=self._CHIEF_PERIMETER)

        # Make Initial Observations

        # Compute rewards
        if self._TRAIN_WAYPOINTER:
            obs = self._make_waypoint_observations()
        else:
            obs = self._make_observations()

        self.unobserved_points = set(range(self.num_points))
        self.truncated = {Role(f"player_{i}"): False for i in range(self.num_deputies)}

        # Initial relative positions and velocities
        self.pos = np.stack([self.orb[role].r-self.orb["chief"].r for role in self.possible_agents])
        self.vel = np.stack([self.orb[role].v-self.orb["chief"].v for role in self.possible_agents])

        # Setup Reward Structure for waypoints
        self.pt_prox = []
        for i in range(self.num_points):
            self.pt_prox.append(multivariate_normal(mean=self.pts[i], cov=self._REW_COV_MTRX))

        self.cum_rewards = np.zeros(self.num_deputies)

        # Setup Longterm Rendering
        self.render_path = render_path
        if render_path:
            frame = self.render()
            assert frame is not None
            self.frame_store = [frame]

        self.render_data = dict()

        return obs, {agt: dict() for agt in self.possible_agents}

    def step(
        self, action: dict[str, FloatArray]
    ) -> tuple[
        dict[Role, Any],
        dict[Role, float],  # reward
        dict[Role, bool],  # terminated
        dict[Role, bool],  # truncated
        dict[Role, dict[str, Any]],  # info
    ]:
        """ """
        # truncated = self.truncated
        self.terminated_this_step = False

        # Propagate Motion In Time
        self._propagate_objects(action)

```

```python
        self.pos = np.stack([self.orb[role].r-self.orb["chief"].r for role in self.possible_agents])
        self.vel = np.stack([self.orb[role].v-self.orb["chief"].v for role in self.possible_agents])

        # Compute Which Points Seen
        just_seen = self._detect_points()

        # Compute rewards
        if self._TRAIN_WAYPOINTER:
            self._compute_waypoint_reward(just_seen)
            obs = self._make_waypoint_observations()
        else:
            self._compute_reward(just_seen)
            obs = self._make_observations()

        # Check if all of the deputes have truncated.
        if all(self.truncated.values()):
            terminated = True

        if self.render_path:
            frame = self.render()
            assert frame is not None
            self.frame_store.append(frame)

            if terminated:
                imageio.mimwrite(
                    uri=self.render_path,
                    ims=self.frame_store,
                    fps=int(60 / 10),  # type: ignore[arg-type]
                    loop=0,
                )

        self.cum_rewards += self.sra

        self.obs = obs
        self.reward = {role: rwd[0] for role, rwd in self.step_rewards.items()}

        self.info: dict[Role, dict[str, Any]] = {
            agt: dict() for agt in self.possible_agents
        }

        self.terminated = {agt: self.terminated_this_step for agt in self.possible_agents}

        # Remove Terminated and Truncated agents from avaiable agent lists
        if self._SB_SAFTY_MODE:
            self.agents = self.possible_agents
        else:
            self.agents = []
            for role in self.possible_agents:
                if not (self.terminated[role] or self.truncated[role]):
                    self.agents.append(role)


        # Set the internal active players array
        self._active = np.array([role in self.agents for role in self.possible_agents])

        return (self.obs, self.reward, self.terminated, self.truncated, self.info)


    def _compute_reward(self, just_seen: list[list[int]]) -> float:
        self.sra = np.zeros(self.num_deputies) # Step Reward Array
        self.step_rewards = {role: self.sra[i:i+1] for i, role in enumerate(self.agents)} # Slices are pointers to the reward array

        self._insepct_reward(just_seen)
        self._chief_prox_reward()
        self._game_length_exceed()
        self._seeing_all_points()
        self._leaving_outer_perimeter()


    def _compute_waypoint_reward(self, just_seen: list[list[int]]) -> float:
        self.sra = np.zeros(self.num_deputies) # Step Reward Array
        self.step_rewards = {role: self.sra[i:i+1] for i, role in enumerate(self.agents)} # Slices are pointers to the reward array

        self._go_towards_waypoint()
        self._game_length_exceed()
        self._leaving_outer_perimeter()
        self._chief_prox_reward()
    def _go_towards_waypoint(self):
        # We're going to change the reward structure to include waypointing
        for i, role in enumerate(self.agents):
            self.step_rewards[role] += self.pt_prox[i].pdf(self.pos[i]) * self._PROX_RWD_SCALE
            if np.linalg.norm(self.pos[i] - self.pts[i]) < self._WAYPOINT_ARRIVAL_PROX:
                self.step_rewards[role] += self._WAYPOINT_ARRIVE_REWARD
                self.terminated_this_step = True



    def _seeing_all_points(self):
        if len(self.unobserved_points) == 0:
            R = (self.max_episode_steps - self.sim_steps) / self.max_episode_steps
            self.sra += R * self._REWARD_FOR_SEEING_ALL_POINTS
            self.terminated_this_step = True



    def _leaving_outer_perimeter(self):
        I = np.where(np.linalg.norm(self.pos, axis=1) > self._MAX_OUTER_PERIMETER)[0]

        # print(np.linalg.norm(self.pos, axis=1) > self._MAX_OUTER_PERIMETER)
        # print(I, self.possible_agents, self.pos)
        for i in I:
            role = Role(f"player_{i}")
            if not self.truncated[role]:
                self.truncated[role] = True
                self.step_rewards[role] += self._REWARD_FOR_LEAVING_OUTER_PERIMETER



    def _game_length_exceed(self):
        self.sim_steps += 1
        if self.sim_steps > self.max_episode_steps:
            ## Technically we were truncated
            self.truncated = {
                Role(f"player_{i}"): True for i in range(self.num_deputies)
            }
            self.terminated_this_step = True



    def _insepct_reward(self, just_seen):
        """ """
        for i, role in enumerate(self.possible_agents):
            seen_points = just_seen[i]
            # Initialize reward
            #  Start with the survival reward
            reward: float = self._REWARD_PER_STEP

            # Check if seen points are new
            new_pts: set = self.unobserved_points.intersection(seen_points)

            # Check if new points were seen
            if new_pts:
                # Reward for seeing points for the first time
                reward += len(new_pts) * self._OBS_REWARD
                # Update unseen points
                self.unobserved_points = self.unobserved_points.difference(new_pts)

            self.step_rewards[role] += reward



    def _chief_prox_reward(self):
        # Are we active and outside the safe zone?
        outside_safe_zone = self._active & (np.linalg.norm(self.pos, axis=1) > self._SIM_OUTER_PARAMETER)
        # Are we active and inside the chief?
        inside_chief = self._active & (np.linalg.norm(self.pos, axis=1) < self._CHIEF_PERIMETER)

        # Adjust reward
        self.sra[outside_safe_zone] += self._REWARD_FOR_LEAVING_PARAMETER
        self.sra[inside_chief] += self._CRASH_REWARD

        # Kill agents inside chief
        self._active[inside_chief] = False

        for i in np.where(inside_chief)[0]:
            self.terminated_this_step = True


    def _random_sphere(self, n_points: int, radius: float) -> FloatArray:
        """ """
        ## Generate random points on a sphere
        pts: FloatArray = -1 + 2 * self.np_random.normal(size=[n_points, 3])
        pts = pts / np.linalg.norm(pts, axis=1).reshape(-1, 1)
        pts = pts * radius

        return pts

    # Physics goes here
    def _propagate_objects(self, actions):
        # Poliastro assumes earth centered coordinates. It fixes an x,y, and z. So
        # vectors in poliastro objects are in terms of of an absolute coordinate system.

        # Translate actions into action frame and apply impulses

        for player, (role, act) in enumerate(actions.items()):
            # Compute Velocity Change
            dv = act[:3].reshape(-1,1) * self._DELTA_V_PER_THRUST
            if self.ACTION_FRAME == "Hills":
                frame = self.hills_frame(self.orb[role])
            else:
                frame = self.ori[role]

            imp = Maneuver.impulse((frame @ dv).reshape(-1))
            self.orb[role] = self.orb[role].apply_maneuver(imp)

            # Apply rotations
            # Sanity Check: if frame is ori, than an action of [1,0,0]
            # fixes ori[:,0] and rotates ori[:,1] and ori[:,2]
            # Here, R has units ff, mapping from frame coords to frame coords
            # with left multiplaction assumed for standard clockwise rotation

            tau = self._TAU_per_THRUST * act[-3:]


            if self._USE_ANGULAR_MOMENTUM:
                self.rot[player] += tau
                tau = self.rot

            R = Rotation.from_euler("xyz", tau)
            T = frame @ R.as_matrix() @ frame.T # x columns to frame, rotate, then back to x

            self.ori[player, :, :] = T @ self.ori[player, :, :]

        # Rotate Chief Points:
        if self.chief_angular_momentum is not None:
            R = self.chief_angular_momentum
            self.pts = R @ self.pts

        # Propagate Orbits
        for role, orb in self.orb.items():
            self.orb[role] = orb.propagate(self._TIME_PER_STEP)

        # for role in self.possible_agents:
        #     print(role, self.orb[role].r, np.linalg.norm(self.orb[role].r - self.orb["chief"].r))

    def _detect_points(self):
        just_seen = []
        for player, role in enumerate(self.possible_agents):
            # Position relvative to chief
            pos = self.orb[role].r - self.orb["chief"].r

            # Detect Points:
            #  This handles occlusion by the chief
            #  Basically, what points are on the same side of the chief as you
            seeable_points = np.where(
                (self.pts * (pos - self.pts)).sum(axis=1) > 0
            )[0]

            ##########
            # Spherical Vision
            #########
            # point holder
            seen_points: list[int] = []

            # loop over possible points
            for point in seeable_points:
                # Cone References
                # cone:
                    https://stackoverflow.com/questions/12826117/how-can-i-detect-if-a-point-is-inside-a-cone-or-not-in-3
                #  We're going to not use the cone, and switch to using a shell
                #  The initial distance calc though is fine, and easy
                #
                # Shell References
                #  This is a pain in the a$$ - I hate math vs physics
                #  Used all of these references to generate a consensus algorithm
                #  https://en.wikipedia.org/wiki/Spherical_coordinate_system
                #   This one doesn't fully work
                #  https://en.wikipedia.org/wiki/Atan2
                #   This explains why atan2 vs atan
                #  https://mathworld.wolfram.com/SphericalCoordinates.html
                #   https://stackoverflow.com/questions/4116658/faster-numpy-cartesian-to-spherical-coordinate-conversion
                #   THIS IS MY FAVORITE ONE - uses physics notation
                #   https://math.libretexts.org/Bookshelves/Calculus/Calculus_(OpenStax)/12%3A_Vectors_in_Space/12.07%3A_
                #   This supports the stackoverflow, just in math notation

                # Position relative to agent
                v = self.pts[point, :] - pos

                # Distance relative to agent
                #  this is the same as the distance in the deputy frame, but
                #  doesn't require the transformation
                p_dist = np.linalg.norm(v)

                # Check if distance is within shell
                #  Ignoring angles at the moment
                #  Less than max, more than min
                #   I assume we're usually too far away
                if (p_dist < self._VISION[role][2]) and (
                    p_dist >= self._VISION[role][1]
                ):
                    # Orient point in deputy frame
                    pdf = self.ori[player].T @ v

                    # theta and phi
                    #  This is physics notation
                    #  theta = polar/zenith angle, [0, pi], z-axis is 0
                    #  phi = azimuth angle, (-pi, pi], x-axis is 0
                    theta = np.arccos(pdf[2] / p_dist)
                    phi = np.arctan2(pdf[1], pdf[0])

                    # theta needs to measure from x-axis, not z-axis
                    #  calculate the complementary angle
                    #  theta = [pi/2, -pi/2], x-axis is 0
                    theta = (np.pi / 2 << u.rad) - theta

                    # Check if point is within cone
                    #  Assume circular directions, so abs() it
                    # NOTE: We could implement different ranges for theta and phi
                    #       This would give a non-circular viewing angle
                    if (np.abs(phi) < self._VISION[role][0]) and (
                        np.abs(theta) < self._VISION[role][0]
                    ):
                        # you can see it!
                        seen_points.append(point)

            just_seen.append(seen_points)
            if self.verbose:
                print(f"{player} seeable: {seeable_points}, seen: {seen_points}")

        return just_seen

    def accel(self, t0, state, k, rate=1e-5):
        """Constant acceleration aligned with the velocity. """
        v_vec = state[3:]
        #v_vec = state[:3]
        norm_v = (v_vec * v_vec).sum() ** 0.5
        return -rate * v_vec / norm_v

    def f(self, t0, u_, k):
        # t_0: time to evaluate at
        # u_ = [x,y,z,vx,vy,vz] in earth coords, rather annoyingly unitless.
        # Assumed units (I've tested this) are km and km/s

        # U.append(u_)
        du_kep = func_twobody(t0, u_, k)
        #ax, ay, az = self.accel(t0, u_, k, rate=1e-5)
        ax, ay, az = self.acc
        du_ad = np.array([0, 0, 0, ax, ay, az])
        return du_kep + du_ad

    def hills_frame(self, orb):
        r = orb.r
        v = orb.v
        v_p = v - proj(v, onto=r)
        return frame(r,v_p)[0].decompose().value

    def _make_observations(self, obs_frame=None) -> dict[Role, FloatArray]:
        """ """
        # Return dictionary
        # Play role will be keys, observations will be values
        obs = dict()

        if obs_frame is None:
            obs_frame = self.OBSERVATION_FRAME

        # Setup point mask
        #  hide points that have been observed
        observedPoints = list(set(range(self.num_points)) - self.unobserved_points)
        pt_mask = np.ones(self.num_points)
        pt_mask[observedPoints] = 0

        poss = np.stack([self.orb[role].r for role in self.possible_agents]).T.to(self._OU_DIS).value
        vels = np.stack([self.orb[role].v for role in self.possible_agents]).T.to(self._OU_VEL).value
        pos_c = self.orb["chief"].r.reshape(-1,1).to(self._OU_DIS).value
        vel_c = self.orb["chief"].v.reshape(-1,1).to(self._OU_VEL).value
        pos_p = self.pts.T.to(self._OU_DIS).value

        # loop over all deputies
        for i, role in enumerate(self.possible_agents):
            # Recall: Moving from absolute coords into a frame is left multiplacation
            # by the transpose

            # Velocities
            # Velocities of all deputies in my frame
            if obs_frame == "Hills":
                frame = self.hills_frame(self.orb[role])
            else:
                frame = self.ori[i]

            vel_in_frame = frame.T @ (vels - vels[:,[i]])
            chief_vel_in_frame = frame.T @ (vel_c - vels[:,[i]])

            #  Relative velocities of all other deputies in my frame
            rel_vel = np.delete(arr=vel_in_frame, obj=i, axis=1)

            # Relative positions of all other deputies in my frame
            rel_pos = (
                frame.T @ np.delete(arr=poss - poss[:,[i]], obj=i, axis=1)
            )

            dchief = pos_c - poss[:,[i]]

            # Relative positions of all points in my frame
            rel_pts = frame.T @ (pos_p + dchief)

            # Relative position of chief in my
            rel_chief = frame.T @ dchief

            # Put the observation space together
            tmp = np.concatenate(
                [
                    rel_chief,
                    chief_vel_in_frame,
                    rel_vel,
                    rel_pos,
                    rel_pts,
                ], axis=1
            )

            # Store observation space under appropriate role
            obs[Role(f"player_{i}")] = np.concatenate([tmp.reshape(-1), pt_mask])

        return obs

    def _make_waypoint_observations(self, obs_frame=None) -> dict[Role, FloatArray]:
        """ """
        # Return dictionary
        # Play role will be keys, observations will be values
        obs = dict()

        if obs_frame is None:
            obs_frame = self.OBSERVATION_FRAME

        # Setup point mask
        #  hide points that have been observed
        observedPoints = list(set(range(self.num_points)) - self.unobserved_points)
        pt_mask = np.ones(self.num_points)
        pt_mask[observedPoints] = 0

        poss = np.stack([self.orb[role].r for role in self.possible_agents]).T.to(self._OU_DIS).value
        vels = np.stack([self.orb[role].v for role in self.possible_agents]).T.to(self._OU_VEL).value
        pos_c = self.orb["chief"].r.reshape(-1,1).to(self._OU_DIS).value
        vel_c = self.orb["chief"].v.reshape(-1,1).to(self._OU_VEL).value
        pos_p = self.pts.T.to(self._OU_DIS).value


        # loop over all deputies
        for i, role in enumerate(self.possible_agents):
            # Recall: Moving from absolute coords into a frame is left multiplacation
            # by the transpose

            # Velocities
            # Velocities of all deputies in my frame
            if obs_frame == "Hills":
                frame = self.hills_frame(self.orb[role])
            else:
                frame = self.ori[i]

            vel_in_frame = frame.T @ (vels - vels[:,[i]])
            chief_vel_in_frame = frame.T @ (vel_c - vels[:,[i]])

            #  Relative velocities of all other deputies in my frame
            rel_vel = np.delete(arr=vel_in_frame, obj=i, axis=1)

            # Relative positions of all other deputies in my frame
            rel_pos = (
                frame.T @ np.delete(arr=poss - poss[:,[i]], obj=i, axis=1)
            )

            dchief = pos_c - poss[:,[i]]

            # print("dchief", dchief)
            # print(self.pts.T.to(self._OU_DIS).value + dchief)
            # Relative positions of all points in my frame
            rel_pts = frame.T @ (pos_p[:,[i]] + dchief)

            # Relative position of chief in my
            rel_chief = frame.T @ dchief

            # Put the observation space together
            tmp = np.concatenate(
                [
                    rel_chief,
                    chief_vel_in_frame,
                    rel_vel,
                    rel_pos,
                    rel_pts,
                ], axis=1
            )

            # Store observation space under appropriate role
            obs[Role(f"player_{i}")] = tmp.reshape(-1)

        return obs



    ## Visualizations
    def render(
        self,
        render_mode: str = "rgb_array",
        vision: Optional[int] = 0,  # player number, but not their role, just the number
        rotate: bool = False,
        elev: float = 45,  # I think it's degrees?
        close: bool = False,
        **kwargs: Any,
    ) -> Union[RenderFrame, None]:
        """ """

        if close:
            self.close()
            return None

        # init figure and canvas
        fig = plt.figure(figsize=[20,5])
        canvas = FigureCanvas(fig)

        poss = np.stack([self.orb[role].r-self.orb["chief"].r for role in self.possible_agents]).T.to(self._OU_DIS).value
        vels = np.stack([self.orb[role].v-self.orb["chief"].v for role in self.possible_agents]).T.to(self._OU_VEL).value

        ##########
        # Title
        ##########
        # plot scores as title
        score_title = "    ".join(
            [f"Player {i}: {self.cum_rewards[i]:.3f}" for i in range(self.num_deputies)]
        )
        fig.suptitle(
            t=score_title, horizontalalignment="center", verticalalignment="center"
        )

        ##########
        # Setup Plot Area
        ##########
        # Plot just the chief, or the chief and one agent-centric view?
        if vision is not None:
            ax = fig.add_subplot(141, projection="3d")
        else:
            ax = plt.axes(projection="3d")

        ##########
        # First Sub Area
        ##########
        # Orientation
        deg = 45
        if (vision is not None) and rotate:
            x, y = poss[vision, 0:2] / np.linalg.norm(poss[vision, 0:2])
            deg = np.degrees(np.arctan2(y, x))

        ax.view_init(elev=elev, azim=deg)

        # limits
        # ax.set_xlim([-300, 300])
        # ax.set_ylim([-300, 300])
        # ax.set_zlim([-300, 300])

        ##########
        # Deputies
        ##########
        # I think all of this just plots deputies - JB

        # Deputy body
        for i in range(self.num_deputies):
            x, y, z = poss[:,[i]]
            ax.plot3D(x, y, z, marker="^", linewidth=0, label=f"player_{i}")

        # Velocity
        # Plot line collection: LC = np.array([N,S,D]), num lines, num of points
        # per line, dim of space, so LC[2,0,:] is point 0 on line 2


        TPS = (self.TIME_PER_STEP * self._OU_VEL/self._OU_DIS).decompose().value

        segs = np.stack([poss.T, poss.T + TPS * vels.T], axis=1)
        line_segments = Line3DCollection(
            segs, linestyle="solid", label="Velocity", color="k"
        )
        ax.add_collection(line_segments)

        # Orientation
        colors = ["r", "g", "b"]
        labels = ["Roll/Heading", "Yaw", "Pitch"]
        for axis in range(len(labels)):
            segs = np.stack([poss.T, poss.T + 30 * self.ori[:, axis, :]], axis=1)
            line_segments = Line3DCollection(
                segs, linestyle="solid", color=colors[axis], label=labels[axis]
            )
            ax.add_collection(line_segments)

        ##########
        # Points on Chief
        ##########
        # Shift for plotting
        view_vec = 100 * np.ones(3) / np.sqrt(3)

        # Unobserved points
        if len(self.unobserved_points) > 0:
            # get positions of points
            unobserved_pos = self.pts[list(self.unobserved_points), :]
            # calc vector for point size
            s = np.linalg.norm(unobserved_pos.to(self._OU_DIS).value + view_vec, axis=1) - 40
            # get in euclidean space
            x, y, z = unobserved_pos.T
            # plot
            ax.scatter(x, y, z, label="Unseen", s=s, color="m")
        else:
            # plot for legend
            #  60: comes from s, when unobserved_points is the empty set
            ax.scatter([], [], [], label="Unseen", s=60, color="m")

        # Observed points
        observed_pts = set(range(self.num_points)) - self.unobserved_points
        if len(observed_pts) > 0:
            # get positions of points
            observed_pos = self.pts[list(observed_pts), :]
            # calc vector for point size
            s = np.linalg.norm(observed_pos.to(self._OU_DIS).value + view_vec, axis=1) - 40
            # get in euclidean space
            x, y, z = observed_pos.T
            # plot
            ax.scatter(x, y, z, color="c", s=s, label="Seen")
        else:
            # plot for legend
            #  60: comes from s, when observed_points is the empty set
            ax.scatter([], [], [], color="c", s=60, label="Seen")

        ##########
        # Legend
        ##########
        # get legend handles
        # handles, _ = ax.get_legend_handles_labels()

        # Style
        fig.legend(loc="lower center", ncol=3, fancybox=True)  # , handles=handles

        ##########
        # Chief Sphere
        ##########
        u, v = np.mgrid[0 : 2 * np.pi : 20j, 0 : np.pi : 10j]  # type: ignore[misc]
        x = self._CHIEF_PERIMETER * np.cos(u) * np.sin(v)
        y = self._CHIEF_PERIMETER * np.sin(u) * np.sin(v)
        z = self._CHIEF_PERIMETER * np.cos(v)
        ax.plot_wireframe(x, y, z, color="k", alpha=0.2)

        mins = np.min(poss.T, axis=0) - 10
        maxes = np.max(poss.T, axis=0) + 10

        widths = []
        centers = []
        for i, s in enumerate(['x', 'y', 'z']):
            lim = getattr(ax, f"get_{s}lim")()
            lim = [
                min(max(mins[i], -300), -self._CHIEF_PERIMETER.to(self._OU_DIS).value+10),
                max(min(maxes[i], 300), self._CHIEF_PERIMETER.to(self._OU_DIS).value+10)
            ]
            widths.append(lim[1]-lim[0])
            centers.append((lim[1]+lim[0])/2)

        w = max(widths)/2
        for i, s in enumerate(['x', 'y', 'z']):
            getattr(ax, f"set_{s}lim")([centers[i] - w, centers[i] + w])


        ##########
        # Render Agent View
        ##########
        if vision is not None:
            ax2 = fig.add_subplot(142, projection="3d")
            all_obs = self._make_observations(obs_frame="Orientation")
            role = cast(Role, f"player_{vision}")
            obs = all_obs[role]

            mask = obs[-self.num_points :]
            obs = obs[:-self.num_points].reshape(3,-1)
            x = obs[:,-self.num_points:]

            # Orientation
            z = np.zeros(2)
            o = 30 * np.array([0, 1])
            ax2.plot(o, z, z, color="r")
            ax2.plot(z, o, z, color="g")
            ax2.plot(z, z, o, color="b")



            seen = x[:, mask == 0]
            unseen = x[:, mask == 1]
            ax2.scatter(seen[0], seen[1], seen[2], color="c")
            ax2.scatter(unseen[0], unseen[1], unseen[2], color="m")
            ax2.set_xlim([-100, 200])
            ax2.set_ylim([-100, 200])
            ax2.set_zlim([-100, 200])


        # Render Earth Frame

        ax3 = fig.add_subplot(143)

        N = len(self.orb.keys())

        poss = np.stack([orb.r for orb in self.orb.values()]).to(self._OU_DIS).value

        if "pos_hist" not in self.render_data.keys():
            self.render_data["pos_hist"] = []

        self.render_data["pos_hist"].append(poss)
        pos_hist = np.array(self.render_data["pos_hist"])

        for i in range(N):
            ax3.plot(pos_hist[:, i, 0], pos_hist[:, i, 1])


        ax3.set_title("Earth Centered Frame")


        # Render Hill Frame

        ax4 = fig.add_subplot(144)

        if "frames" not in self.render_data.keys():
            self.render_data["frames"] = []

        if "chief_rel_poss" not in self.render_data.keys():
            self.render_data["chief_rel_poss"] = []

        c_idx = list(self.orb.keys()).index("chief")
        frame = self.hills_frame(self.orb["chief"])

        chief_rel_poss = (frame.T @ poss.T).T
        chief_rel_poss = chief_rel_poss - chief_rel_poss[c_idx,:]

        self.render_data["chief_rel_poss"].append(chief_rel_poss)
        self.render_data["frames"].append(frame)

        hills = np.array(self.render_data["chief_rel_poss"])

        for i in range(N):
            ax4.plot(hills[:, i, 1], hills[:, i, 0])

        chief = plt.Circle((0, 0), self._CHIEF_PERIMETER.to(self._OU_DIS).value, color='k', fill=False)
        ax4.add_patch(chief)
        ax4.set_title("Chief Centered Hill Frame")

        # Draw

        if render_mode == "rgb_array":
            canvas.draw()

            data = np.frombuffer(fig.canvas.tostring_rgb(), dtype=np.uint8)
            data = data.reshape(fig.canvas.get_width_height()[::-1] + (3,))
            plt.close(fig)
```

```python

            return data

        plt.close(fig)
        return None

    def close(self) -> None:
        pass

    def state(self) -> None:
        raise NotImplementedError("State Not Implemented.")


if __name__ == "__main__":
    from tqdm import tqdm
    from PIL import Image

    env = MultInspect(
        num_deputies=4,
        _SIM_OUTER_PARAMETER=100<<u.m,
        _CHIEF_PERIMETER = 100 << u.m
    )
    env.seed(0)
    env.reset()

    frames = []
    # for i in tqdm(range(env.max_episode_steps)):
    for i in tqdm(range(10)):
        act = {role: np.array([0,0,0,0,0,0]) for role in env.possible_agents}
        env.step(act)
        frames.append(env.render())

    # Image.fromarray(frames[0])

    imgs = [Image.fromarray(frame) for frame in frames]
    # duration is the number of milliseconds between frames; this is 40 frames per second
    model_dir = ""
    imgs[0].save(model_dir + f"_eval_run_{i}.gif", save_all=True, append_images=imgs[1:], duration=500, loop=0)
# %%
```

7 examples/MAInspection/Experiment

7.1 examples/MAInspection/Experiment/dash_app.yaml

```
interfaces:
  Default: # Name of Interface
    interface_class: DefaultInterface
    models:
      Default:
        class_name: DefaultActor

  Waypointer: # Name of Interface
    interface_class: WaypointInterface

    roles: [player_0] # If this is not supplied, assumed all roles
    models:
      PPOWaypointer:
        class_name: SB_PPOWaypointActor
        params:
          policy_path: waypointer/MAInspect_20240213-193621/model.zip

directors:
  PPO_Director:
    class_name: SB3_PPO_Director
    path: director/MAInspect_20240206-211327.zip
    roles:
      player_0:
        classes: [PPOWaypointer]
```

7.2 examples/MAInspection/Experiment/dash_app_template.yaml

```
interfaces:
  Default: # Name of Interface
    interface_class: DefaultInterface
    models:
      Default:
        class_name: DefaultActor

  Waypointer: # Name of Interface
    interface_class: WaypointInterface

    roles: [player_0] # If this is not supplied, assumed all roles
    models:
      PPOWaypointer:
        class_name: SB_PPOWaypointActor
        params:
          policy_path: <WAYPOINTER_PATH>

directors:
  PPO_Director:
    class_name: SB3_PPO_Director
    path: <DIRECTOR_PATH>
    roles:
      player_0:
        classes: [PPOWaypointer]
```

7.3 examples/MAInspection/Experiment/train_agents.yaml

```
waypointer:
  env_params:
    num_deputies: 1
    _SB_SAFTY_MODE: true
    _TRAIN_WAYPOINTER: true
    _REWARD_FOR_LEAVING_PARAMETER: 0
    _REWARD_FOR_LEAVING_OUTER_PERIMETER: 0
    _CRASH_REWARD: 0
    _REW_COV_MTRX: 22500 # 150^2
    _PROX_RWD_SCALE: 10000
```

7.4 examples/MAInspection/Experiment/train_director.yaml

```
env_params:
  num_deputies: 1
  max_episode_steps: 150
  _TRAIN_WAYPOINTER: false
COACH_params:
  stochastic: true
  FIXED_STEPS_PER_COM:
    checkin_frequency: 20
    allow_agent_break: False
  ACTION_PADDING: 0
  MIN_NEXT_ACTION_TIME: 19
  MAX_NEXT_ACTION_TIME: 20
  seed: 453413
  Agents:
    player_0:
      class_name: "SB_PPOWaypointActor"
      params:
        policy_path:
```

8 examples/MAInspection/assets

8.1 examples/MAInspection/assets/typography.css

```
body {
    font-family: sans-serif;
}

h1, h2, h3, h4, h5, h6 {
    color: black
}

.resume {
    display: none;
}

.button4 {
    background-color: white; /* Green */
    border: none;
    color: black;
    padding: 15px 32px;
    text-align: left;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
    width: 100%;
  }

.button4:hover {
    background-color: #DDDDDD; /* Green */
    color: white;
  }


.float-container {
    border: 3px solid #fff;
    padding: 20px;
    width: 600px;
    margin:0;
    height:300px;
}

.float-child {
    width: 50%;
    height:100%;
    float: left;
    padding: 20px;
    border: 2px solid red;
}

.agent_card {
    float: left;
    padding: 10px;
    height: 150px;
    width: 100%;
    margin: auto;
  }

.action_cards {
    float: left;
    padding: 10px;
    width: 80%;
    margin: auto;
  }

.left_card {
    float: left;
    width: 25%;
    height: 100%;
    background-color: #aaa;
    margin:auto;
  }

.right_card {
    float: left;
    width: 70%;
    height: 100%;
    background-color: #bbb;
    margin:auto;
  }

  .coa_card {
    float: left;
    width: 15%;
    height: 100%;
    background-color: #bbb;
    margin:auto;
  }

.current_plan tr {
    background-color: #ccc;
}

.archived_plan tr {
    background-color: #eee;
}

.archived_plan tr:hover {
    background-color: #ddd;
}

/* Clear floats after the columns */
.agent_card_container:after {
    content: "";
    display: table;
    clear: both;
  }


/* Colors for sliders */

.rc-slider-handle-1 {
    border-color: #01befe;
}

.rc-slider-handle-1.rc-slider-handle-click-focused:focus {
    border-color: #01befe;
}

.rc-slider-handle-1:hover {
    border-color: #01befe;
}

/* ===================== */

.rc-slider-handle-2 {
    border-color: #ffdd00;
}

.rc-slider-handle-2.rc-slider-handle-click-focused:focus {
    border-color: #ffdd00;
}

.rc-slider-handle-2:hover {
    border-color: #ffdd00;
}

/* ===================== */

.rc-slider-handle-3 {
    border-color: #ff7d00;
}

.rc-slider-handle-3.rc-slider-handle-click-focused:focus {
    border-color: #ff7d00;
}

.rc-slider-handle-3:hover {
    border-color: #ff7d00;
}

/* ===================== */

.rc-slider-handle-4 {
    border-color: #ff006d;
}

.rc-slider-handle-4.rc-slider-handle-click-focused:focus {
    border-color: #ff006d;
}
```

```

.rc-slider-handle-4:hover {
    border-color: #ff006d;
}

/* ====================== */

.rc-slider-handle-5 {
    border-color: #adff02;
}

.rc-slider-handle-5.rc-slider-handle-click-focused:focus {
    border-color: #adff02;
}

.rc-slider-handle-5:hover {
    border-color: #adff02;
}

/* ====================== */

.rc-slider-handle-6 {
    border-color: #8f00ff;
}

.rc-slider-handle-6.rc-slider-handle-click-focused:focus {
    border-color: #8f00ff;
}

.rc-slider-handle-6:hover {
    border-color: #8f00ff;
}

/* ====================== */
```

9 examples/Multipolicy

9.1 examples/Multipolicy/Readme.md

```
# Multiploicy Director

In this example, we use StableBaselines3 and Supersuit to train two policies on the Waterworld environment from PettingZoo. One we train in a sparse food environment as an "explorer" agent, the other we train in a dense food and dense poison environment as a "dense" avoider agent. Finally, we train a director to select the policies each agent should use for the next 10 turns given their current observations.

In this example we see how to train policies on a PettingZoo environment, use those policies and that PettingZoo environment to establish a planning problem using the coach environment. We then set up a communications schedule to allow communication every 10 turns and train the director agent to select between the policies.

Finally, we construct a Dash app to display our solution.

This example uses all most entirely default functionality. For more advanced functionality see the MAInspection example.

<img src="Interface.png">
```

9.2 examples/Multipolicy/app.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import sys
import glob
import yaml
sys.path.insert(0, "../../")

from DASH.app import *

from pettingzoo.sisl import waterworld_v4
from utilities.PZWrapper import PettingZooEnv

# Logging
import logging
logging.basicConfig(level=logging.DEBUG, format="%(levelname)s:%(name)s:%(message)s")
logger = logging.getLogger(__name__)

pymunk_loggers = [logging.getLogger(name) for name in logging.root.manager.loggerDict if name.startswith("pymunk")]
for log_handler in pymunk_loggers:
    log_handler.setLevel(logging.INFO)


def get_env():
    return PettingZooEnv(PZGame=waterworld_v4)

if __name__ == "__main__":
    print("app - ################### RELOADING DASH APP #########################")

    MODEL_PATH = ""
    DENSE_PATH = min(glob.iglob(os.path.join(MODEL_PATH,'dense/*.zip')), key=os.path.getctime)
    EXPLORE_PATH = min(glob.iglob(os.path.join(MODEL_PATH,'explore/*.zip')), key=os.path.getctime)

    with open(os.path.join("Experiment","train_director.yaml"), "r") as f:
        try:
            params = yaml.safe_load(f)
        except yaml.YAMLError as exc:
            print(exc)


    with open(os.path.join("Experiment","dash_app.yaml"), "r") as f:
        try:
            params["actor_params"] = yaml.safe_load(f)
        except yaml.YAMLError as exc:
            print(exc)

    # external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']
    env_factory = COACHIntegration(
        env_creator=get_env,
        COACHEnvClass=COACHEnvironment,
        parameters=params
        )

    ct.env_factory = env_factory

    locks = {
                  "action_card": False
            }

    ## App Layout
    proxy_url = re.sub("{{port}}", "8050", os.environ["VSCODE_PROXY_URI"])
    proxy_url = re.sub(os.environ["ACEHUB_BASEURL"], "", proxy_url)
    app = Dash(serve_locally=True, requests_pathname_prefix=proxy_url)

    app.layout = app_layout(env_factory)

    app.run(debug=True)
```

9.3 examples/Multipolicy/train_agents.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from __future__ import annotations

from pettingzoo.sisl import waterworld_v4
import supersuit as ss
from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy
from stable_baselines3.common.callbacks import EvalCallback
import os
import time
import glob
from tqdm import tqdm
import numpy as np
import yaml
import copy

import torch
CORES = 9
torch.set_num_threads(CORES)
torch.set_num_interop_threads(CORES)


# Code adapted from https://pettingzoo.farama.org/tutorials/sb3/waterworld/

## We're going to train two policies, one that assumes a densely poisonous env
## and one that requires sparse exploration.

def train_butterfly_supersuit(
    env_fn,
    steps: int = 10_000,
    seed: int | None = 0,
    num_vec_envs=1,
    num_cpus=CORES,
    learning_rate=1e-3,
    batch_size=256,
    model_dir="",
    name="",
    **env_kwargs
):
    model_path = os.path.join(model_dir, name)
    os.makedirs(model_path,exist_ok=True)

    # Train a single model to play as each agent in a cooperative Parallel environment
    env = env_fn.parallel_env(**env_kwargs)
    env.reset(seed=seed)

    print(f"Starting training on {str(env.metadata['name'])}.")

    env = ss.pettingzoo_env_to_vec_env_v1(env)
    env = ss.concat_vec_envs_v1(env, num_vec_envs=num_vec_envs, num_cpus=num_cpus, base_class="stable_baselines3")

    # Note: Waterworld's observation space is discrete (242,) so we use an MLP policy rather than CNN
    model = PPO(
        MlpPolicy,
        env,
        verbose=3,
        learning_rate=learning_rate,
        batch_size=batch_size,
        tensorboard_log=os.path.join("./tensorboard_log/", name)
    )

    eval_env = env_fn.parallel_env(**env_kwargs)
    eval_env.reset(seed=seed)
    eval_env = ss.pettingzoo_env_to_vec_env_v1(eval_env)
    eval_env = ss.concat_vec_envs_v1(eval_env, num_vec_envs=1, num_cpus=num_cpus, base_class="stable_baselines3")

    eval_callback = EvalCallback(eval_env, verbose=1, eval_freq=10000)

    weight_name = f"{env.unwrapped.metadata.get('name')}_{time.strftime('%Y%m%d-%H%M%S')}"
    weight_path = os.path.join(model_path, weight_name)

    model_args = {
        "name": name,
        "load_class": "SB_PPOPoliciesActor",
        "env_params": env_kwargs
    }

    with open(weight_path + ".yaml", "w") as f:
        yaml.dump(model_args, f, default_flow_style=False)

    model.learn(total_timesteps=steps, callback=eval_callback)
    model.save(weight_path)


    print("Model has been saved.")
    print(f"Finished training on {str(env.unwrapped.metadata['name'])}.")

    env.close()

    return weight_path


def eval(
    env_fn,
    model_path,
    num_games: int = 10,
    render_mode: str | None = None,
    **env_kwargs
):
    # Evaluate a trained agent vs a random agent
    env = env_fn.env(render_mode=render_mode, **env_kwargs)

    print(
        f"\nStarting evaluation on {str(env.metadata['name'])} (num_games={num_games}, render_mode={render_mode})"
    )

    model = PPO.load(model_path)

    cum_rewards = []

    # Note: We train using the Parallel API but evaluate using the AEC API
    # SB3 models are designed for single-agent settings, we get around this by using he same model for every agent
    for i in tqdm(range(num_games)):
        env.reset(seed=i)
        rewards = {agent: 0 for agent in env.possible_agents}

        for agent in env.agent_iter():
            obs, reward, termination, truncation, info = env.last()

            for a in env.agents:
                rewards[a] += env.rewards[a]
            if termination or truncation:
                for a in env.agents:
                    cum_rewards.append(rewards[a])
                break
            else:
                act = model.predict(obs, deterministic=True)[0]

            env.step(act)
    env.close()

    avg_reward = np.mean(cum_rewards)
    std_reward = np.std(cum_rewards)

    with open(model_path + ".txt","a") as f:
        f.writelines(f"\n\n{env_kwargs}\n")
        f.writelines(f"\t Avg reward: {avg_reward}, std: {std_reward}\n")
        f.writelines(f"\t Rewards: {cum_rewards}\n")

    print(f"\t Avg reward: {avg_reward}, std: {std_reward}")
    return avg_reward

```

```

if __name__ == "__main__":
    env_fn = waterworld_v4
    env_kwargs = {}

    STEPS = 1000

    ## Train Dense Eater
    ## Note: We want one eater here so that it doesn't learn to just sit there and hope for food.
    model_path_dense = train_butterfly_supersuit(
        env_fn=env_fn,
        steps=STEPS,
        seed=0,
        n_pursuers=1,
        n_evaders=10,
        n_poisons=40,
        model_dir="",
        name="dense"
        )

    ## Train Explore Eater
    model_path_explore = train_butterfly_supersuit(
        env_fn=env_fn,
        steps=STEPS,
        seed=0,
        n_pursuers=3,
        n_evaders=5,
        n_poisons=5,
        model_dir="",
        name="explore")

    for model_path in [model_path_dense, model_path_explore]:#, model_path_explore]:
        eval(env_fn=env_fn,
            model_path=model_path,
            n_pursuers=3,
            n_evaders=20,
            n_poisons=30,
            )

        eval(env_fn=env_fn,
            model_path=model_path,
            n_pursuers=3,
            n_evaders=5,
            n_poisons=5,
            )

        eval(env_fn=env_fn,
            model_path=model_path,
            n_pursuers=3,
            n_evaders=15,
            n_poisons=15,
            )
```

9.4 examples/Multipolicy/train_director.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
from __future__ import annotations
import sys
sys.path.insert(0, "../../")

import os
import time
import glob
from tqdm import tqdm
import numpy as np
from PIL import Image
import yaml

from utilities.PZWrapper import PettingZooEnv
from coach import COACHEnvironment
from env import COACH_PettingZoo

from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy
from stable_baselines3.common.callbacks import EvalCallback
import supersuit as ss

from pettingzoo.sisl import waterworld_v4

import torch
CORES = 6
torch.set_num_threads(CORES)
torch.set_num_interop_threads(CORES)

import logging
logging.basicConfig()
logging.getLogger().setLevel(logging.INFO)
logger = logging.getLogger(__name__)

pymunk_loggers = [logging.getLogger(name) for name in logging.root.manager.loggerDict if name.startswith("pymunk")]
for log_handler in pymunk_loggers:
    log_handler.setLevel(logging.INFO)

# SB code adapted from https://pettingzoo.farama.org/tutorials/sb3/waterworld/

## We're going to train two policies, one that assumes a densely poisonous env
## and one that requires sparse exploration.
# %%
def train(
    env_fn,
    steps: int = 10_000,
    seed: int | None = 0,
    num_vec_envs=1,
    num_cpus=1,
    learning_rate=1e-3,
    batch_size=256,
    model_dir=""
):
    os.makedirs(model_dir,exist_ok=True)

    # Train a single model to play as each agent in a cooperative Parallel environment
    env = env_fn()
    env.reset(seed=seed)
    print(env.possible_agents)

    logger.info(f"Starting training on {str(env.metadata['name'])}.")

    env = ss.pettingzoo_env_to_vec_env_v1(env)
    env = ss.concat_vec_envs_v1(env, num_vec_envs=num_vec_envs, num_cpus=num_cpus, base_class="stable_baselines3")

    # Note: Waterworld's observation space is discrete (242,) so we use an MLP policy rather than CNN
    model = PPO(
        MlpPolicy,
        env,
        verbose=3,
        learning_rate=learning_rate,
        batch_size=batch_size,
        tensorboard_log=os.path.join("./tensorboard_log/", model_dir.split("/")[-1])
    )

    eval_env = env_fn()
    eval_env = ss.pettingzoo_env_to_vec_env_v1(eval_env)
    eval_env = ss.concat_vec_envs_v1(eval_env, num_vec_envs=num_vec_envs, num_cpus=num_cpus, base_class="stable_baselines3")
    eval_callback = EvalCallback(eval_env, verbose=1, eval_freq=500)

    model.learn(total_timesteps=steps, callback=eval_callback)

    model_name = f"{env.unwrapped.metadata.get('name')}_{time.strftime('%Y%m%d-%H%M%S')}"
    model_path = os.path.join(model_dir, model_name)
    model.save(model_path)

    logger.info("Model has been saved.")
    logger.info(f"Finished training on {str(env.unwrapped.metadata['name'])}.")

    env.close()

    return model_path + ".zip"

def eval(
    env_fn,
    model_path,
    num_games: int = 10,
    render = False,
    **env_kwargs
):
    # Evaluate a trained agent vs a random agent
    env = env_fn(**env_kwargs)

    logger.info(
        f"\nStarting evaluation on {str(env.metadata['name'])} (num_games={num_games}, render={render})"
    )

    model = PPO.load(model_path)

    cum_rewards = []

    # Note: We train using the Parallel API but evaluate using the AEC API
    # SB3 models are designed for single-agent settings, we get around this by using he same model for every agent
    for i in tqdm(range(num_games)):
        if render:
            env.coa_env.start_rendering()

        obs, info = env.reset(seed=i)

        rewards = {agent: 0 for agent in env.possible_agents}
        running = True
        j = 0
        while running:
            j += 1
            act = {"director": model.predict(obs["director"], deterministic=False)[0]}

            logger.debug("Obs From Director: %s", obs["director"]) # DEBUG
            logger.debug("Action From Director: %s", act) # DEBUG

            obs, reward, term, trunc, info = env.step(act)
            for a in env.agents:
                rewards[a] += reward[a]

            if all([a or b for a,b in zip(term.values(), trunc.values())]):
                for a in env.agents:
                    cum_rewards.append(rewards[a])

                running = False

        if render:
            frames = env.coa_env.state.trajectory.frames
            imgs = [Image.fromarray(frame) for frame in frames]
            # duration is the number of milliseconds between frames; this is 40 frames per second
            model_dir = model_path.split(".")[0]
            imgs[0].save(model_dir + f"_eval_run_{i}.gif", save_all=True, append_images=imgs[1:], duration=500, loop=0)

    env.close()

    avg_reward = np.mean(cum_rewards)
    std_reward = np.std(cum_rewards)

    with open(model_path + ".txt","a") as f:
        f.writelines(f"\n\n{env_kwargs}\n")
        f.writelines(f"\t Avg reward: {avg_reward}, std: {std_reward}\n")
        f.writelines(f"\t Rewards: {cum_rewards}\n")

    logger.info(f"\t Avg reward: {avg_reward}, std: {std_reward}")
    return avg_reward


def get_env():
    return PettingZooEnv(PZGame=waterworld_v4)

if __name__ == "__main__":
    MODEL_PATH = ""
    DENSE_PATH = min(glob.iglob(os.path.join(MODEL_PATH,'dense/*.zip')), key=os.path.getctime)
    EXPLORE_PATH = min(glob.iglob(os.path.join(MODEL_PATH,'explore/*.zip')), key=os.path.getctime)

    with open(os.path.join("Experiment", "train_director.yaml"), "r") as stream:
        try:
            params = yaml.safe_load(stream)
        except yaml.YAMLError as exc:
            print(exc)

    for agent, param in params["COACH_params"]["Agents"].items():
        param["params"]["policy_paths"]["dense"] = DENSE_PATH
        param["params"]["policy_paths"]["explore"] = EXPLORE_PATH

    env = COACH_PettingZoo(env_creator=get_env, COACHEnvClass=COACHEnvironment)

    def get_env_pz():
        env = COACH_PettingZoo(env_creator=get_env, COACHEnvClass=COACHEnvironment)
        env.augment(params)
        env.reset()
        return env

    model_path = train(
        get_env_pz,
        steps=100,
        seed=0,
        model_dir="director"
        )

    model_path = min(glob.iglob(os.path.join(MODEL_PATH,'director/*.zip')), key=os.path.getctime)
    rew = eval(
        get_env_pz,
        model_path=model_path,
        num_games=1,
        render=True
        )

    with open(os.path.join("Experiment", "dash_app_template.yaml"), 'r') as f:
        with open(os.path.join("Experiment", "dash_app.yaml"), 'w') as g:
            for line in f.readlines():
                line = line.replace("<DENSE_PATH>", DENSE_PATH)
                line = line.replace("<EXPLORE_PATH>", EXPLORE_PATH)
                line = line.replace("<DIRECTOR_PATH>", model_path)
                g.write(line)

# %%
```

10 examples/Multipolicy/Experiment

10.1 examples/Multipolicy/Experiment/dash_app.yaml

```yaml
interfaces:
  Default: # Name of Interface
    interface_class: DefaultInterface
    models:
      Default:
        class_name: DefaultActor

  Random: # Name of Interface
    interface_class: RandomActionInterface
    models:
      Random:
        class_name: RandomActor

  Dense_v_Sparce: # Name of Interface
    interface_class: SBPolicyInterface
    iterface_parameters:
      n_policies: 2
      max_action_len: 10

    roles: [pursuer_0, pursuer_1, pursuer_2] # If this is not supplied, assumed all roles
    models:
      Dense_v_Sparse_v1:
        class_name: SB_PPOPoliciesActor
        params:
          policy_paths:
            dense: dense/waterworld_v4_20240213-185458.zip
            explore: explore/waterworld_v4_20240213-185504.zip
          max_action_len: 10

      Dense_v_Sparse_v2:
        class_name: SB_PPOPoliciesActor
        params:
          policy_paths:
            dense: dense/waterworld_v4_20240213-185458.zip
            explore: explore/waterworld_v4_20240213-185504.zip
          max_action_len: 10

  Tall_v_Short:
    interface_class: SBPolicyInterface
    iterface_parameters:
      n_policies: 2
      max_action_len: 10

    models:
      Tall_v_Short:
        class_name: SB_PPOPoliciesActor
        params:
          policy_paths:
            dense: dense/waterworld_v4_20240213-185458.zip
            explore: explore/waterworld_v4_20240213-185504.zip
          max_action_len: 10
    roles: [pursuer_0, pursuer_1, pursuer_2] # If this is not supplied, assumed all roles

directors:
  PPO_Director:
    class_name: SB3_PPO_Director
    path: director/waterworld_v4_20240213-190312.zip
    roles:
      pursuer_0:
        classes: [Dense_v_Sparse_v1, Dense_v_Sparse_v2]
      pursuer_1:
        classes: [Dense_v_Sparse_v1, Dense_v_Sparse_v2]
      pursuer_2:
        classes: [Dense_v_Sparse_v1, Dense_v_Sparse_v2]
```

10.2 examples/Multipolicy/Experiment/dash_app_template.yaml

```yaml
interfaces:
  Default: # Name of Interface
    interface_class: DefaultInterface
    models:
      Default:
        class_name: DefaultActor

  Random: # Name of Interface
    interface_class: RandomActionInterface
    models:
      Random:
        class_name: RandomActor

  Dense_v_Sparce: # Name of Interface
    interface_class: SBPolicyInterface
    iterface_parameters:
      n_policies: 2
      max_action_len: 10

    roles: [pursuer_0, pursuer_1, pursuer_2] # If this is not supplied, assumed all roles
    models:
      Dense_v_Sparse_v1:
        class_name: SB_PPOPoliciesActor
        params:
          policy_paths:
            dense: <DENSE_PATH>
            explore: <EXPLORE_PATH>
          max_action_len: 10

      Dense_v_Sparse_v2:
        class_name: SB_PPOPoliciesActor
        params:
          policy_paths:
            dense: <DENSE_PATH>
            explore: <EXPLORE_PATH>
          max_action_len: 10

  Tall_v_Short:
    interface_class: SBPolicyInterface
    iterface_parameters:
      n_policies: 2
      max_action_len: 10

    models:
      Tall_v_Short:
        class_name: SB_PPOPoliciesActor
        params:
          policy_paths:
            dense: <DENSE_PATH>
            explore: <EXPLORE_PATH>
          max_action_len: 10
    roles: [pursuer_0, pursuer_1, pursuer_2] # If this is not supplied, assumed all roles

directors:
  PPO_Director:
    class_name: SB3_PPO_Director
    path: <DIRECTOR_PATH>
    roles:
      pursuer_0:
        classes: [Dense_v_Sparse_v1, Dense_v_Sparse_v2]
      pursuer_1:
        classes: [Dense_v_Sparse_v1, Dense_v_Sparse_v2]
      pursuer_2:
        classes: [Dense_v_Sparse_v1, Dense_v_Sparse_v2]
```

10.3 examples/Multipolicy/Experiment/train_agents.yaml

```
train_agents:
  dense:
    env_params:
      n_evaders: 10
      n_poisons: 40
      n_pursuers: 1
  explore:
    env_params:
      n_evaders: 5
      n_poisons: 5
      n_pursuers: 3
```

10.4 examples/Multipolicy/Experiment/train_director.yaml

```
env_params:
  n_pursuers: 3
  max_cycles: 30
COACH_params:
  stochastic: true
  FIXED_STEPS_PER_COM:
    checkin_frequency: 10
    allow_agent_break: False
  ACTION_PADDING: 0
  MIN_NEXT_ACTION_TIME: 1
  MAX_NEXT_ACTION_TIME: 10
  seed: 453413
  Agents:
    pursuer_0:
      class_name: "SB_PPOPoliciesActor"
      params:
        max_action_len: 10
        policy_paths:
          dense:
          explore:
    pursuer_1:
      class_name: "SB_PPOPoliciesActor"
      params:
        max_action_len: 10
        policy_paths:
          dense:
          explore:
    pursuer_2:
      class_name: "SB_PPOPoliciesActor"
      params:
        max_action_len: 10
        policy_paths:
          dense:
          explore:
```

11 examples/Multipolicy/assets

11.1 examples/Multipolicy/assets/typography.css

```css
body {
    font-family: sans-serif;
}

h1, h2, h3, h4, h5, h6 {
    color: black
}

.resume {
    display: none;
}

.button4 {
    background-color: white; /* Green */
    border: none;
    color: black;
    padding: 15px 32px;
    text-align: left;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
    width: 100%;
  }

.button4:hover {
    background-color: #DDDDDD; /* Green */
    color: white;
  }


.float-container {
    border: 3px solid #fff;
    padding: 20px;
    width: 600px;
    margin:0;
    height:300px;
}

.float-child {
    width: 50%;
    height:100%;
    float: left;
    padding: 20px;
    border: 2px solid red;
}

.agent_card {
    float: left;
    padding: 10px;
    height: 150px;
    width: 100%;
    margin: auto;
  }

.action_cards {
    float: left;
    padding: 10px;
    width: 80%;
    margin: auto;
  }

.left_card {
    float: left;
    width: 25%;
    height: 100%;
    background-color: #aaa;
    margin:auto;
  }

.right_card {
    float: left;
    width: 70%;
    height: 100%;
    background-color: #bbb;
    margin:auto;
  }

  .coa_card {
    float: left;
    width: 15%;
    height: 100%;
    background-color: #bbb;
    margin:auto;
  }

.current_plan tr {
    background-color: #ccc;
}

.archived_plan tr {
    background-color: #eee;
}

.archived_plan tr:hover {
    background-color: #ddd;
}

/* Clear floats after the columns */
.agent_card_container:after {
    content: "";
    display: table;
    clear: both;
  }


/* Colors for sliders */

.rc-slider-handle-1 {
    border-color: #01befe;
}

.rc-slider-handle-1.rc-slider-handle-click-focused:focus {
    border-color: #01befe;
}

.rc-slider-handle-1:hover {
    border-color: #01befe;
}

/* ===================== */

.rc-slider-handle-2 {
    border-color: #ffdd00;
}

.rc-slider-handle-2.rc-slider-handle-click-focused:focus {
    border-color: #ffdd00;
}

.rc-slider-handle-2:hover {
    border-color: #ffdd00;
}

/* ===================== */

.rc-slider-handle-3 {
    border-color: #ff7d00;
}

.rc-slider-handle-3.rc-slider-handle-click-focused:focus {
    border-color: #ff7d00;
}

.rc-slider-handle-3:hover {
    border-color: #ff7d00;
}

/* ===================== */

.rc-slider-handle-4 {
    border-color: #ff006d;
}

.rc-slider-handle-4.rc-slider-handle-click-focused:focus {
    border-color: #ff006d;
}
```

```

.rc-slider-handle-4:hover {
    border-color: #ff006d;
}

/* ====================== */

.rc-slider-handle-5 {
    border-color: #adff02;
}

.rc-slider-handle-5.rc-slider-handle-click-focused:focus {
    border-color: #adff02;
}

.rc-slider-handle-5:hover {
    border-color: #adff02;
}

/* ====================== */

.rc-slider-handle-6 {
    border-color: #8f00ff;
}

.rc-slider-handle-6.rc-slider-handle-click-focused:focus {
    border-color: #8f00ff;
}

.rc-slider-handle-6:hover {
    border-color: #8f00ff;
}

/* ====================== */
```

12 utilities

12.1 utilities/PZParams.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""PettingZoo Parameter Class """

import argparse
import logging
from argparse import Namespace  # for type hinting
from collections.abc import Callable  # for type hinting
from typing import Any, Optional  # for type hinting

class EnvParams:
    """Base class for environment params."""

    def __init__(self, args: Any, param_section_name: str = "env_params") -> None:
        self.args = args
        self._params = getattr(args, param_section_name, {})

    def __getitem__(self, key: str) -> Any:
        """Return value stored for key from the params dict."""
        return self._params[key]

    def __setitem__(self, key: str, value: Any) -> None:
        """Set the value for key in the params dict."""
        self._params[key] = value

    def get(self, key: str, default: Optional[Any] = None) -> Any:
        """Return value for key from the params dict, or None if it doesn't exist."""
        try:
            return self._params[key]
        except KeyError:
            return default

    def get_mutated_params(self) -> "EnvParams":
        """Return a mutated copy of the params"""
        raise NotImplementedError(
            f"get_mutated_params has not been implemented in {type(self)}"
        )

    def checkpoint(self, folder: str) -> None:
        """Save a checkpoint in the given folder."""
        raise NotImplementedError(
            f"checkpoint has not been implemented in {type(self)}"
        )

    def reload(self, folder: str) -> None:
        """Read a checkpoint from the given folder."""
        raise NotImplementedError(f"reload has not been implemented in {type(self)}")


###############################################################################
## Auxiliary Functions
###############################################################################
def get_env_param_class(args: Namespace) -> Callable[..., Any]:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env parameter objects
    """
    return PettingZooEnvParams


################################################################################
## Main Class
################################################################################
class PettingZooEnvParams(EnvParams):
    """Parameters for PettingZoo Environments"""
```

12.2 utilities/PZWrapper.py

```
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from Farama-Foundation/PettingZoo
# (https://github.com/Farama-Foundation/PettingZoo)
# under the MIT License.

"""PettingZoo Wrapper"""

import argparse  # for type hinting
import logging
from collections.abc import Callable  # for type hinting
from typing import Any, Union, cast  # for type hinting

from gymnasium.wrappers import FlattenObservation
from pettingzoo.classic import connect_four_v3 as PZGame  # type: ignore[import]
from pettingzoo.sisl import waterworld_v4 as PZGame

logging.getLogger("pettingzoo.utils.env_logger").setLevel(logging.WARNING)
logger = logging.getLogger(__name__)

import copy

import numpy as np
from gymnasium.spaces import Box, Discrete, flatten, flatten_space, unflatten
from numpy.typing import NDArray
from pettingzoo.utils.conversions import aec_to_parallel
from pettingzoo.utils.env import AECEnv, ParallelEnv

from .PZParams import PettingZooEnvParams


################################################################################
## Auxiliary Functions
################################################################################
def softmax(x):
    exp = np.exp(x)
    return exp / cast(float, np.exp(x).sum())


@unflatten.register(Discrete)
def _unflatten_discrete(
    space: Discrete, x: Union[NDArray[np.int64], NDArray[float]]
) -> np.int64:
    nonzero = np.nonzero(x)
    if len(nonzero[0]) == 0:
        raise ValueError(
            f"{x} is not a valid one-hot encoded vector and can not be unflattened to space {space}. "
            "Not all valid samples in a flattened space can be unflattened."
        )

    act = np.argmax(x)

    return space.start + act


#######################################################
## Factories
#######################################################
def get_env_class(args: argparse.Namespace) -> Callable[..., Any]:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env objects
    """
    return PettingZooEnv


## This is what you want to edit if you add wrappers, or change
## things about the env parameters.
def env_creator(**kwargs):
    return PZGame.env(**kwargs)


################################################################################
## Main Class
################################################################################
class PettingZooEnv:
    def __init__(self, PZGame) -> None:
        """ """
        def env_creator(**kwargs):
            return PZGame.parallel_env(**kwargs)

        self.env_creator = env_creator
        # If you have wrappers define a function to set it up properly
        self.standard_params = {"render_mode": "rgb_array"}
        self.current_params = None

    def __getattr__(self, name):
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
        if name == "env":
            if "env" not in self.__dict__.keys():
                self.__dict__["env"] = None
            return self.__dict__["env"]

        return getattr(self.env, name)

    def unflatten_or_none(self, action_space, action):
        if action is not None:
            return unflatten(action_space, action)
        else:
            return None

    def step(self, act):
        act = {agt: self.unflatten_or_none(self.env.action_space(agt), v) for agt, v in act.items()}

        # logger.info(f"action: {str(act)}")
        if self.parallel:
            return self.env.step(act)
        else:
            self.env.step(act[self.env.agent_selection])

        obs = {
            agt: flatten(self.env.observation_space(agt), self.env.observe(agt))
            for agt in self.env.possible_agents
        }

        # logger.info(f"action: {str(obs)}, {self.env.terminations}, {self.env.truncations}")
        return (
            obs,
            self.env.rewards,
            self.env.terminations,
            self.env.truncations,
            self.env.infos,
        )

    def reset(self):
        if self.parallel:
            return self.env.reset()

        self.env.reset()
        # print(obs)
        obs = {
            agt: self.env.observation_space(agt)
            for agt in self.env.possible_agents
        }
        # return obs, info
        return obs, self.env.info

    def augment(self, params):
        self.current_params = copy.copy(self.standard_params)

        for k, v in params.items():
            self.current_params[k] = v

        self.env = self.env_creator(**self.current_params)
        if isinstance(self.env, ParallelEnv):
            self.parallel = True
        else:
            self.parallel = False

        self.env.reset()

        self.action_spaces = dict()
        self.observation_spaces = dict()

        self.action_type = dict()
        for agt in self.env.possible_agents:
            self.action_spaces[agt] = flatten_space(self.env.action_space(agt))
            self.observation_spaces[agt] = flatten_space(
                self.env.observation_space(agt)
            )
            self.action_type[agt] = type(self.env.action_space(agt))

    def seed(self, seed):
        if hasattr(self.env, "seed"):
            self.env.seed(seed)

    def render(self, *args, **kwargs):
        if kwargs.get("close", False):
            self.env.close()
            return

        return self.env.render()
```

12.3 utilities/iotools.py

```
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Contains various utility functions/classes used in the project"""
import csv
import json
import os
from collections.abc import Callable  # for type hinting
from typing import Any, Union, cast, NewType  # for type hinting

import numpy as np

EnvId = NewType("EnvId", str)
PathString = str


################################################################################
## Auxiliary Functions
################################################################################
### Turn dictionary with keyed tuple into csv
def save_keyed_tuple(
    dct: dict[tuple[EnvId, EnvId], float],
    filename: PathString,
    do_sort: bool = True,
) -> None:
    """
    Writes a csv file of a dict that is indexed by a tuple (a matrix)

    Parameters
    ----------
    dct : dict
        Dict of values with keys given as tuples
    filename : str
        Name of file to save to
    do_sort : boolean, optional
        whether to sort the output keys
        This is assuming they are of the form "Env_#"

    Side-Effects
    ------------
    None

    Returns
    -------
    None

    Notes
    -----
    As an example, for a dict of:
    dct = {(a,a): val_aa, (a,b): val_ab,
           (b,a): val_ba, (b,b): val_bb,
           (c,a): val_ca, (c,b): val_cb,
           (d,a): val_da, (d,b): val_db}
    The output is:
        ,a,b
        a,val_aa,val_ab
        b,val_ba,val_bb
        c,val_ca,val_cb
        d,val_da,val_db
    Note that the ',' at the start of the first line is intentional
    to provide an empty cell in the csv format so the data forms a
    rectangular matrix.
    """
    # split (l, r) keys into lists of l and r
    l_keys = list({first for first, second in dct})
    r_keys = list({second for first, second in dct})

    # sort by the numeric part of Env_X
    if do_sort:
        l_keys.sort(key=lambda x: int(x.split("_")[1]))
        r_keys.sort(key=lambda x: int(x.split("_")[1]))

    with open(filename, mode="w", newline="", encoding="utf8") as file:
        csvfile = csv.writer(
            file, delimiter=",", quoting=csv.QUOTE_MINIMAL, lineterminator="\n"
        )
        header = [""] + r_keys  # [""] to add blank entry to csv row
        csvfile.writerow(header)

        for first in l_keys:
            row = [first] + [dct.get((first, second), "") for second in r_keys]
            csvfile.writerow(row)


def load_keyed_tuple(
    filename: PathString,
    format_function: Callable[[str], Any] = lambda x: x,
) -> dict[tuple[EnvId, EnvId], Any]:
    """
    Read a tuple-indexed dict (a matrix) from a csv file.

    This is the reverse of save_keyed_tuple
    The format_function is used to convert the values from string to
    whatever format is desirable. The default leaves it as a string.
    A typical choice would be float.

    Parameters
    ----------
    filename : str
        Name of file to save to
    format_function: callable, optional
        This is applied to values (but not keys) read from the file

    Returns
    -------
    dict
        The dict that was read

    Notes
    -----
    For a file with the following:
        ,a,b
        a,val_aa,val_ab
        b,val_ba,val_bb
        c,val_ca,val_cb
        d,val_da,val_db
    The output is:
    dct = {(a,a): val_aa, (a,b): val_ab,
           (b,a): val_ba, (b,b): val_bb,
           (c,a): val_ca, (c,b): val_cb,
           (d,a): val_da, (d,b): val_db}
    """
    dct: dict[tuple[EnvId, EnvId], Any] = {}
    with open(filename, mode="r", newline="", encoding="utf8") as file:
        csvfile = csv.reader(file, delimiter=",")
        # first line has an empty spot to account for alignment
        # of columns. So, we ignore that
        r_keys = next(csvfile)[1:]
        for l_key, *values in csvfile:
            for r_key, value in zip(r_keys, values):
                # appease type checker
                l_key = cast(EnvId, l_key)
                r_key = cast(EnvId, r_key)
                dct[(l_key, r_key)] = format_function(value)
    return dct


################################################################################
## Numpy Encoders for JSON
################################################################################
## Recursively encodes objects with a reprJSON function
# https://stackoverflow.com/questions/5160077/encoding-nested-python-object-in-json
#
# Encdoing numpy objects:
# https://stackoverflow.com/questions/26646362/numpy-array-is-not-json-serializable
#
# Decoding objects
```

```python
# https://stackoverflow.com/questions/48991911/how-to-write-a-custom-json-decoder-for-a-complex-object

# Usage
# with open(filename, 'w') as jsonfile:
#     json.dump(edge, jsonfile, cls=NumpyEncoder)
# with open(filename, 'r') as jsonfile:
#     edge1 = json.load(jsonfile, cls=NumpyDecoder)


class NumpyEncoder(json.JSONEncoder):
    """Encode numpy data for JSON writer"""

    def default(self, o):  # type: ignore  # this is called by the JSON library
        if isinstance(o, np.integer):
            return {"np.integer": int(o)}
        if isinstance(o, np.floating):
            return {"np.floating": float(o)}
        if isinstance(o, np.ndarray):
            return {"np.array": o.tolist()}
        return json.JSONEncoder.default(self, o)


class NumpyDecoder(json.JSONDecoder):
    """Decode numpy data from JSON"""

    def __init__(self, *args, **kwargs):  # type: ignore  # this is called by the JSON library
        json.JSONDecoder.__init__(self, object_hook=self.numpy_hook, *args, **kwargs)

    def numpy_hook(self, dct):  # type: ignore  # this is called by the JSON library
        """Convert dict with numpy data to numpy object"""
        if "np.integer" in dct:
            return np.int_(dct["np.integer"])
        if "np.floating" in dct:
            return np.float_(dct["np.floating"])
        if "np.array" in dct:
            return np.array(dct["np.array"])
        return dct


################################################################################
## TensorFlow Logging
################################################################################
class TBWriter:
    def __init__(self, args: dict[str, Any]) -> None:
        """ """
        import time

        from tensorflow import summary  # type: ignore[import]

        self.summary = summary
        log_dir = args["log_dir"]

        now = time.localtime()
        subdir = time.strftime("%d-%b-%Y_%H.%M.%S", now)

        self.summary_dir = os.path.join(log_dir, subdir)
        self.summary_writer: dict[str, summary.SummaryWriter] = {}

    def create_scalar_writer(self, name: str) -> None:
        new_dir = os.path.join(self.summary_dir, name)
        self.summary_writer[name] = self.summary.create_file_writer(new_dir)

    def write_item(self, name: str, label: str, data: Any, step: int) -> None:
        if label not in self.summary_writer.keys():
            self.create_scalar_writer(label)

        with self.summary_writer[label].as_default():
            if isinstance(data, (np.ndarray, np.generic)):
                data = data.item()

            if hasattr(data, "__len__"):
                data = data[0]

            self.summary.scalar(name=name, data=data, step=step)
        self.summary_writer[label].flush()

    def write_items(self, name: str, data: dict[str, list[float]], step: int) -> None:
        for label in data.keys():
            self.write_item(name, label, data[label], step)


```

```
class Telemetry:
    def __init__(self) -> None:
        self.loggers: list[TBWriter] = []

    def add_logger(self, logger: TBWriter) -> None:
        self.loggers.append(logger)

    def write_item(self, name: str, label: str, data: Any, step: int) -> None:
        for logr in self.loggers:
            logr.write_item(name, label, data, step)

    def write_items(self, name: str, data: dict[str, list[float]], step: int) -> None:
        for logr in self.loggers:
            logr.write_items(name, data, step)
```

12.4 utilities/planning.py

```
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from .timelines import Timeline, Timelines, TimelineEvent
import numpy as np
from numpy.random import Generator, PCG64DXSM
import copy
import pandas as pd

class State:
    def __init__(self, parameters, seed, trajectory, current_t=0):
        self.parameters = parameters
        self.seed = seed
        self.trajectory = trajectory
        self.current_t = current_t
        self.cummulative_rews = dict()

    def __repr__(self):
        return self.__str__()

    def __str__(self):
        return f"parameters: {self.parameters}, seed: {self.seed}, trajectory len: {len(self.trajectory)}"


class Trajectory:
    def __init__(self, env, initial_return, intial_frame=None):
        self.players = copy.copy(env.possible_agents)
        self.trajectory = []
        self.actions = []
        self.step_returns = []
        self.agent_info = []
        self.add(
            env, None, initial_return, None
        )
        self.frames = []
        if intial_frame is not None:
            self.frames.append(intial_frame)

    def add(self, env, action, step_return, agent_info, frame=None):
        self.trajectory.append(
            {
                "action": action,
                "step_return": step_return,
                "agent_info": agent_info
             }
        )
        self.actions.append(action)
        self.step_returns.append(step_return)
        self.agent_info.append(agent_info)

        if frame is not None:
            self.frames.append(frame)

    def __getitem__(self, key):
        return self.trajectory[key]

    def __len__(self):
        return len(self.trajectory)


## Telemetry Object:
class Telemetry:
    def __init__(self, name, **kwargs):
        self.data = None
        self.data_labels = None
        self.xscale = None
```

```python
        self.title = None
        self.name = None
        self.xlabel = None
        self.ylabel = None
        self.colors = None

        for k,v in kwargs.items():
            if k in self.__dict__.keys():
                self.__dict__[k] = v
            else:
                raise Exception(f"Keyword argument {k} not in telemetry class.")

    def __str__(self) -> str:
        shape = getattr(self.data, "shape", None)
        return f"Telemetry - name: {self.name}, shape: {shape}"

    def __repr__(self) -> str:
        return self.__str__()

    def set_data(self, data):
        self.data = data
        self.data_labels = [""] * data.shape[0]
        self.colors = [None] * data.shape[0]

    def as_df(self):
        # print("Telemtry:", self.data, self.data_labels, self.xscale)

        tmp = pd.DataFrame(self.data.T)

        # print("Telemtry:", tmp)
        columns = copy.copy(self.data_labels)
        for i in range(len(columns)):
            if columns[i] is None:
                if self.name is not None:
                    columns[i] = self.name
                else:
                    columns[i] = ""

        tmp.columns = columns

        # print("Telemtry:", tmp)
        tmp.index = self.xscale
        return tmp

# Returns a fixed seed and fixed set of parameters
class BaseParameterGenerator:
    def __init__(self, seed, param):
        self.seed = seed
        self.param = param

    def sample(self):
        return (self.seed, self.param)

    def setseed(self, seed):
        pass

    def update_param(self, param):
        self.param = param

    def freeze(self):
        pass

    def thaw(self):
        pass

# This returns a random seed but a fixed set of parameters
class SeededParameterGenerator:
    def __init__(self, seed, param):
        self.np_random = Generator(PCG64DXSM(seed=seed))
        self.param = param
        self._frozen = False

    def sample(self):
        if self._frozen:
            return (self._frozen_random, self.param)
        else:
            return (self.np_random.integers(low=2**32), self.param)

    def setseed(self, seed):
        self.np_random = Generator(PCG64DXSM(seed=seed))
    def update_param(self, param):
        self.param = param

    def freeze(self):
        self._frozen = True
        self._frozen_random = self.np_random.integers(low=2**32)

    def thaw(self):
        self._frozen = False

# %%
class CommunicationSchedule(Timelines):
    @staticmethod
    def repeating(
        checkin_frequency=1,
        checkin_offset=0,
        blackout_frequency=1,
        blackout_length=0,
        blackout_offset=0,
        allow_agent_break=True,
    ):
        ## This is going to secretely overload the standard timeline behavior to make it
        ## infinite
        tmp = CommunicationSchedule(length=1, allow_agent_break=allow_agent_break)
        tmp.__dict__["checkin_frequency"] = checkin_frequency
        tmp.__dict__["checkin_offset"] = checkin_offset
        tmp.__dict__["blackout_frequency"] = blackout_frequency
        tmp.__dict__["blackout_length"] = blackout_length
        tmp.__dict__["blackout_offset"] = blackout_offset
        tmp.__dict__["allow_agent_break"] = allow_agent_break,
        tmp.__dict__["repeating"] = True

        blackout_event = TimelineEvent(parameters=None, label="blackouts")
        checkin_event = TimelineEvent(parameters=None, label="checkins")

        def next_event(time):
            t1 = (checkin_frequency - time + checkin_offset) % checkin_frequency
            if t1 == 0:
                t1 = checkin_frequency
            return t1 + time, checkin_event

        tmp.checkins.__dict__["next_event"] = next_event

        def get(time, role):
            if ( (time - blackout_offset)  % blackout_frequency) < blackout_length:
                return True

            return False

        tmp.blackouts.__dict__["get"] = get

        return tmp


    @staticmethod
    def from_lists(length, blackouts = [], checkins = [], allow_agent_break=True):
        tmp = CommunicationSchedule(length, allow_agent_break)
        blackout_event = TimelineEvent(parameters=None, label="blackouts")
        checkin_event = TimelineEvent(parameters=None, label="checkins")

        for t in blackouts:
            tmp.blackouts.add_events(t, blackout_event)

        for t in checkins:
            tmp.checkins.add_events(t, checkin_event)

        return tmp

    def __init__(self, length, allow_agent_break=True):
        super().__init__(labels=["blackouts", "checkins"], length=length)
        self.ALLOW_AGENT_BREAK = allow_agent_break
        self.repeating = False

        def get(time, role):
            return False

        self.blackouts.__dict__["get"] = get
        # self.checkins.add_event(time=0, event=TimelineEvent(parameters=None, label="checkin"))

    def __getstate__(self):
        return self.__dict__

    def __setstate__(self, state):
        self.__dict__ = state


    def __deepcopy__(self, memo):
        if self.repeating:
            return CommunicationSchedule.repeating(
                checkin_frequency=copy.copy(self.checkin_frequency),
                checkin_offset=copy.copy(self.checkin_offset),
                blackout_frequency=copy.copy(self.blackout_frequency),
                blackout_length=copy.copy(self.blackout_length),
                blackout_offset=copy.copy(self.blackout_offset),
                allow_agent_break=copy.copy(self.allow_agent_break),
            )

        tmp = self.__class__(labels=[])
        tmp.timelines = copy.deepcopy(self.timelines)

        return(tmp)

class COA(Timeline):
    def to_dict(self):
        d = {
            time: {event.label: event.to_dict() for event in events} for time, events in self.timeline.items()
            }
        return d

    def get_actions(self, t):
        return self.timeline.get(t, default=None)
```

12.5 utilities/timelines.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import numpy as np
import matplotlib.pyplot as plt
import json
import copy

from typing import Any


## Aux Classes:
# Note: Everything here has been written so that it (at present)
# deepcopies. If you do anything to make a model not deepcopy
# by default you'll need to add a __deepcopy__ function

# %%
#######################
class TimelineEvent:
    id_idx = 0
    @staticmethod
    def from_dict(event):
        tmp = TimelineEvent(
            event["label"],
            event["parameters"],
            tags = {k:v for k,v in event.items() if (k not in {"label", "parameters", "id"})},
            id = event.get("id", None)
            )

        return tmp

    def __init__(self, label, parameters, tags: dict = dict(), id=None):
        if id == None:
            self.id = TimelineEvent.id_idx
            TimelineEvent.id_idx += 1

        self.label = label
        self.parameters = parameters
        self.tags = tags

    def to_dict(self):
        return {
            "label": self.label,
            "parameters": self.parameters,
            "tags": self.tags
        }

    def __repr__(self) -> str:
        return str(self)

    def __str__(self) -> str:
        return f"label: {self.label}, params: {self.parameters}, tags: {self.tags}"

    def copy(self):
        return copy.copy(self)


class TimelineInterval(TimelineEvent):
    def __init__(self, *args, extent=1):
        super().__init__(*args)
        self.extent = extent
        self.range = range(self.start, self.start + extent + 1)


class Timeline:
    def __init__(self, length=0, fixedlength=None):
        self.timeline = dict()
        self.events = dict()
        self.labels = dict()

        self.fixedlength = False
        self.length = length
        self.allow_dup_labels = False

        if fixedlength is not None:
            if length>0:
                raise Exception("Both length and fixedlength are specified. Only one should be.")
            self.fixedlength = True
            self.length = fixedlength

    def __repr__(self) -> str:
        return f"Timeline: {str(self.events)}"

    def __len__(self):
        return self.length

    def __deepcopy__(self, memo):
        return(copy.deepcopy(self.__dict__))

    def add_events_from_dict(self, events):
        # Expected format:
        # event = {
        #     "start": int,
        #     "action": str,
        #     "parameters": np.array,
        #     "role": str,
        # }

        for event in events:
            self.add_event(event["start"], TimelineEvent.from_dict(event))


    def add_events(self, events):
        if type(events) is TimelineEvent:
            self.add_event(events)
        else:
            for t, e in events:
                self.add_event(t, e)

    def add_event(self, time, event: TimelineEvent):
        if self.fixedlength:
            if time > self.length:
                raise Exception(f"Timeline has fixed length {self.length} but event added at {time}.")
        else:
            self.length = max(self.length, time)

        if not self.allow_dup_labels:
            mark_for_delete = []
            for e in self.timeline.get(time, []):
                if e.label == event.label:
                    mark_for_delete.append(e)

            for e_del in mark_for_delete:
                self.delete_event_at(time=time, event=e_del)


        if time not in self.timeline.keys():
            self.timeline[time] = set()

        if event not in self.events.keys():
            self.events[event] = set()

        if event.label not in self.labels.keys():
            self.labels[event.label] = set()

        self.timeline[time].add(event)
        self.timeline = dict(sorted(self.timeline.items()))

        self.events[event].add(time)
        self.labels[event.label].add((time, event))

    def delete_event_at(self, event, time):
        self.events[event].remove(time)
        if len(self.events[event]) == 0:
            del self.events[event]

        self.timeline[time].remove(event)
        if len(self.timeline[time]) == 0:
```

```python
            del self.timeline[time]
            if time == self.length-1 and not self.fixedlength:
                self.length = max(self.timeline.keys())

        self.labels[event.label].remove((time, event))
        if len(self.labels[event.label]) == 0:
            del self.labels[event.label]

    def delete_all_occurences(self, event):
        while event in self.event.keys():
            t = list(self.events[event])[0]
            self.delete_event_at(event=event, time=t)

    def move_events(self, time, to, events=None):
        if events is None:
            events = copy.copy(self.timeline[time]) # we're modifying the object as we iterate through it

        for event in events:
            self.delete_event_at(event = event, time=time)
            self.add_event(event=event, time=to)



    def add_interval(self, event: TimelineInterval):
        for t in event.range:
            self.add_event(t, event)

    def first(self, event):
        return(min(self.events[event]))

    def last(self, event):
        return(max(self.events[event]))

    def interval_start(self, event, t):
        if t not in self.events[event]:
            return None

        while t in self.events[event]:
            t += -1

        return t + 1

    def interval_end(self, event, t):
        if t not in self.events[event]:
            return None

        while t in self.events[event]:
            t += 1

        return t - 1

    def get_all_labeled(self, label):
        return self.labels[label]

    def get_events_at_time(self, time):
        return self.timeline[time]

    def get(self, time, label):
        r = []
        if label not in self.labels.keys():
            raise Exception(f"'{label}' not found in known labels: {list(labels.keys())}")

        for t, event in self.labels[label]:
            if t == time:
                r.append(event)

        if len(r) == 1:
            return r[0]

        if len(r) == 0:
            return None

        if len(r) > 1:
            raise Exception(f"Multiple events with same label at same time: {r}")

    def itterate_over_label(self, label):
        tmp = dict()
        for time, event in self.labels[label]:
            if time not in tmp.keys():
                tmp[time] = []

            tmp[time].append(event)

        # Returns a key sorted list of tuples (t, [e1, e2,...])
        return iter(sorted(tmp.items()))

    def next_event(self, time):
        L = list(self.timeline.keys())
        if len(L) == 0:
            return None, None

        last_high = L[-1]
        if time >= last_high:
            return None, None
        if time < L[0]:
            return L[0], self.timeline[L[0]]

        # Go through timeline vis bisection. Probably overkill
        while len(L)>1:
            half_idx = int(len(L)/2)

            if L[half_idx] > time:
                last_high = L[half_idx]
                L = L[:half_idx]
            else:
                L = L[half_idx:]
                if L[-1] > time:
                    last_high = L[-1]

        return last_high, self.timeline[last_high]

    def __iter__(self):
        ## Should already be sorted
        return iter(sorted(self.timeline.items()))

    def add_timeline_to(self, timeline, allow_dup_labels=True):
        ## Add new coa elements
        for event, times in timeline.events.items():
            if event not in self.events.keys():
                self.events[event] = set()

            to_update = times - self.events[event]

            for t in to_update:
                self.add_event(time=t, event=event)

    def display(self):
        PADDING = 3
        LINEWIDTH = 64
        LABELLENGTH = 8
        TLSPACE = LINEWIDTH - PADDING - LABELLENGTH

        if self.length < TLSPACE:
            step = 1
            LINEWIDTH = LABELLENGTH + self.length
            TLLENGTH = self.length
        else:
            step = int(np.ceil(self.length/(TLSPACE)))
            TLLENGTH = int(np.ceil(self.length/step))

        ## Time:
        ticklabs = [" "]*LINEWIDTH
        ticklabs[0:5] = list("Time:")
        ticks = [" "]*LABELLENGTH + ["-"]*(TLLENGTH+1)

        demarks = list(range(0,self.length, step*10))

        for j, i in enumerate(demarks):
            j = LABELLENGTH + j*10

            c = len(str(i))
            tmp = [" "]*7
            if c < 8:
                tmp[(3-int((c-1)/2)):(3-int((c-1)/2)) + c] = str(i)
            else:
                tmp = "{:.2E}".format(c)

            ticklabs[j-3:j+4] = tmp

            ticks[j] ="|"

        ticks[-1] ="|"

        ## Lines

        lines = {
            label: list(str(label)[:LABELLENGTH]) + [" "] * max(0,LABELLENGTH - len(label)) + ["  "]*TLLENGTH
            for label in self.labels.keys()
            }

        for label, events in self.labels.items():
            for t, event in events:
                lines[label][LABELLENGTH + int(t/step)] = "o"

        lines = ["".join(line) for line in lines.values()]
        lines.insert(0, "".join(ticklabs))
        lines.insert(1, "".join(ticks))

        return "\n".join(lines)


class Timelines:
    def __init__(self, labels, length=0, fixedlength = None):
        self.timelines = {label:Timeline(length=length, fixedlength=fixedlength) for label in labels}

    def get_events(self, time):
        return {label: tl.get_events_at_time(time) for label, tl in self.timeslines.items()}

    def __repr__(self) -> str:
        return str(self.timelines)

    def __getattr__(self, __name):
        return self.timelines[__name]

    # def __deepcopy__(self, memo):
    #     tmp = self.__class__(labels=[])
    #     tmp.timelines = copy.deepcopy(self.timelines)

    #     return(tmp)

    def __getstate__(self):
        return self.timelines

    def __setstate__(self, d):
        self.timelines = d
```

12.6 utilities/tools.py

```python
# Copyright (c) 2024 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from typing import Any


class MultiDict:
    def __init__(self, tags: dict = dict()):
        self.tags = tags # tags - str(type): list(values)
        self._vals = dict()
        self._refs = {tag: {val:[] for val in values} for tag, values in tags.items()}

    def __getattr__(self, __name: str) -> Any:
        if __name in self.tags.keys():
            return SliceMakerDict(__name, self)

    def __getitem__(self, select_name: dict):
        new_tags = dict(self.tags)

        for tag_name in select_name.keys():
            del new_tags[tag_name]

        new_mdct = MultiDict(new_tags)

        items = [set(self._refs[tag_name][tag]) for tag_name, tag in select_name.items()]
        items = set.intersection(*items)

        for item in items:
            new_tag = dict(self._vals[item])
            for tag_type_name in select_name.keys():
                del new_tag[tag_type_name]

            new_mdct[new_tag] = item

        return new_mdct


    def __setitem__(self, key: dict, value: Any):
        # Following are sufficient to check all keys supplied:
        if not len(key.keys()) == len(self.tags.keys()):
            raise Exception(f"Mismatched key lengths between {key.keys()} and {self.tags.keys()}")

        for tag_type, tag in key.items():
            try:
                tags = self._refs[tag_type]
            except:
                raise Exception(f"Trying to set tag_type {tag_type} but this is not found in {self.tags.keys()}")

            if tag not in tags:
                self.tags[tag_type].append(tag)
                tags[tag] = list()

            tags[tag].append(value)

        self._vals[value] = key

    def __repr__(self) -> str:
        s = ""
        tab_count = []
        for tag_type in self.tags.keys():
            count = int(np.floor(len(tag_type) / 8)) + 1
            tab_count.append(count)
            s += f"{tag_type}\t"

        s += "value: \n"

        for item, tag in self._vals.items():

            # Keep the same order as above
            for i, tag_type in enumerate(self.tags.keys()):
                max_len = tab_count[i]*8 - 2
                tag_len = len(str(tag[tag_type]))

                if tag_len > max_len:
                    sp = str(tag[tag_type])[tab_count[i]*8-2] + "\t"
                else:
                    count = int(np.floor((max_len - tag_len)/8) + 1)
                    sp = str(tag[tag_type]) + "\t"*count

                s += sp

            s += str(item)
            s += "\n"

        return s

class SliceMakerDict:
    def __init__(self, tag_type_name, parent: MultiDict):
        self.parent: MultiDict = parent
        self.tag_type_name = tag_type_name

    def __repr__(self) -> str:
        return str(self.parent.tags[self.tag_type_name])

    def __getitem__(self, key):
        new_tags = dict(self.parent.tags )
        del new_tags[self.tag_type_name]

        new_mdct = MultiDict(new_tags)
        for item, tag in self.parent._vals.items():
            if tag[self.tag_type_name] == key:
                new_tag = dict(tag)
                del new_tag[self.tag_type_name]

                new_mdct[new_tag] = item

        return new_mdct






def unit_vector(vector):
    """Returns the unit vector of the vector."""
    return vector / np.linalg.norm(vector)


def angle_between(v1, v2):
    """Returns the angle in radians between vectors 'v1' and 'v2'::

    >>> angle_between((1, 0, 0), (0, 1, 0))
    1.5707963267948966
    >>> angle_between((1, 0, 0), (1, 0, 0))
    0.0
    >>> angle_between((1, 0, 0), (-1, 0, 0))
    3.141592653589793
    """
    v1_u = unit_vector(v1)
    v2_u = unit_vector(v2)
    return np.arccos(np.clip(np.dot(v1_u, v2_u), -1.0, 1.0))


def plot_traj(p, q, v, traj=None, w=[0, 0], ax=None):
    if ax is None:
        ax = plt.gca()
    if not traj is None:
        ax.plot(traj[:, 0], traj[:, 1], "--")
    ax.plot(p[0], p[1], "or")
    ax.plot(q[0], q[1], "og")
    ax.quiver(p[0], p[1], v[0], v[1], angles="xy", scale_units="xy", scale=1)
    ax.quiver(
        p[0], p[1], float(w[0]), float(w[1]), angles="xy", scale_units="xy", scale=1
    )
    # ax.set_xlim(-60,10)
    # ax.set_ylim(-10,10)
    return ax

```

```python
def plot_traj_3d(p, q, v, traj=None, w=[0, 0], ax=None):
    if ax is None:
        ax = plt.figure().add_subplot(projection="3d")
    if not traj is None:
        ax.plot(traj[:, 0], traj[:, 1], traj[:, 2], "--")
    ax.plot(p[0], p[1], p[2], "or")
    ax.plot(q[0], q[1], q[2], "og")
    ax.quiver(p[0], p[1], p[2], v[0], v[1], v[2])
    ax.quiver(p[0], p[1], p[2], float(w[0]), float(w[1]), float(w[2]))
    return ax


def r(t, w, p, v):
    return p + (t**2) * w / 2 + t * v


def r_p(t, w, v):
    return t * w + v


def g(t, w, t1, p, v):
    if t < t1:
        return r(t, w, p, v)
    else:
        return r(t1, w, p, v) + (t - t1) * r_p(t1, w, v)


def dg(t, w, t1, p):
    return r(t1, w, p) + t * r_p(t1, w, p)


def rotation_matrix(theta):
    theta

    rotMatrix = np.array(
        [[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]]
    )

    return rotMatrix
```