# 第２回空戦 AI チャレンジ

# 配布シミュレータの取扱説明書

主催 防衛装備庁 委託 SIGNATE Empowering Your Potential

# 目次

# 1 配布シミュレータの構成と環境構築

配布シミュレータ(以下、本シミュレータという)のディレクトリ構成と環境構築の方法を説明する。

## 1.1 ディレクトリ構成

配布データの simulator_dist.zip をそのまま解凍すると、以下のようなディレクトリが作成される。root 以下が本体となる。

```
simulator_dist
├root
│ ├addons：本シミュレータの追加機能として実装されたサブモジュール
│ │ ├AgentIsolation：行動判断モデルをシミュレータ本体と隔離して動作させるための評価用機能
│ │ ├HandyRLUtility：HandyRL を用いて学習を行うための拡張機能
│ │ ├MatchMaker：複数の行動判断モデルを用いて Self-Play 等の対戦管理を行うための拡張機能
│ │ ├rayUtility：ray RLlib を用いて学習を行うための拡張機能
│ │ └torch_truncnorm：PyTorch で切断ガウス分布を用いるための拡張機能
│ ├ASRCAISim1：最終的に Python モジュールとしての本シミュレータが格納されるディレクトリ
│ ├include：コア部を構成するヘッダファイル
│ ├sample：戦闘環境を定義し学習を行うためのサンプル
│ │ ├HandyRLSample：HandyRL を用いて基本的な強化学習を行うためのサンプル
│ │ ├MinimumEvaluation：各行動判断モデルを隔離して対戦させる最小限の評価環境
│ │ ├OriginalModelSample：独自の Agent クラスと報酬クラスを定義するためのサンプル
│ │ └raySample：ray を用いて基本的な強化学習を行うためのサンプル
│ ├src：コア部を構成するソースファイル
│ └thirdParty：外部ライブラリの格納場所(同梱は改変を加えたもののみ)
│   └include
│     └pybind11_json  ※オリジナルを改変したものを同梱
├Dockerfile：環境構築に必要な Dockerfile
└requirements.txt ： 環境構築に必要な Python ライブラリ一覧
```

## 1.2 環境の構築

### 1.2.1 方法 1: Docker による環境構築

こちらが推奨される方法。まず、自分の OS 環境に応じて Docker を導入する。導入方法については、以下のサイトを参照。

https://docs.docker.com/get-docker/

Docker の導入ができたら、simulator_dist.zip を解凍して Dockerfile が存在するディレクトリ(そのまま解凍したなら simulator_dist)に移動して、以下のコマンドを実行する。

```
docker build .
```

コンテナの立ち上げには時間がかかることに注意。コンテナ環境の中に入り、以下動作確認用サンプルを実行しエラーが出なければ成功である。

```
docker run -it IMAGEID
...
cd /path/to/sample/MinimumEvaluation
python evaluator.py "Rule-Fixed" "Rule-Fixed" -n 1 -l "./result"
```

この方法で構築した環境は GUI は非対応で、投稿したときに動作するサーバーと同じ環境である。

### 1.2.2 方法 2: ソースコードからのビルド

ソースコードからビルドしたい場合は以下の手順に従う。

#### 1.2.2.1 推奨動作環境の準備(主要な項目のみ)

本シミュレータの推奨動作環境は以下の通りである。

OS：Ubuntu 18.04 LTS 又は Ubuntu 20.04 LTS(gcc と Python のバージョンにより後者を推奨)
Python 3.8.11 (3.6 や 3.7 でも動作可能)
Tensorflow：2.7.0
PyTorch：1.11.0+cuda11.3
ray：1.13.0(>=1.13,<2.0)
gcc：11.2.0(>=9)
Eigen：3.4.0 (>=3.4.0)
pybind11：2.8.0 (2.6.2 以上で動作確認済)
nlohmann’s json：3.9.1 において動作確認済
pybind11_json：0.2.11(を改変)
NLopt：2.6.2 において動作確認済
Magic Enum C++：0.7.3
Boost：1.65.1 において動作確認済
CMake：3.21 (>=3.10)

なお、C++コンパイラについては C++17 の機能を使用しているため、以下の通りのバージョン要求がある。(以下は最も要求バージョンが厳しい Magic Enum C++による。)

clang/LLVM >= 5
MSVC++ >= 14.11 / Visual Studio >=2017
Xcode >= 10
gcc >= 9
MinGW >= 9

上記に含まれる C++依存ライブラリのインストールに関する説明は以下の通り。

#### 1.2.2.2 C++依存ライブラリのインストール

本シミュレータは以下の C++ライブラリを使用している。

(1) NLopt https://nlopt.readthedocs.io/en/latest/

NLopt は非線形最適化ライブラリである。上記の web ページの内容に従いインストールされたい。本シミュレータでは/usr/local 以下にインストールすることを想定している。もし異なるディレクトリにインストールする場合は、本シミュレータの CMakeLists.txt においてパスを適切に設定されたい。

(2) Eigen https://eigen.tuxfamily.org/

Eigen はヘッダオンリーの線形代数ライブラリである。上記の web ページの内容に従いダウンロードされたい。本シミュレータでは CMake を用いて eigen のインストールを行うことを想定している。

なお、Eigen は一部に LGPL のコードを含んでおり、本シミュレータではコンパイラオプションにて-DEIGEN_MPL2_ONLY を指定しているため MPL2 の部分のみを使用するようにしている。

(3) pybind11 https://pybind11.readthedocs.io/en/stable/

pybind11 はヘッダオンリーの C++・Python 連接用ライブラリである。本シミュレータでは pip によりインストールすることを想定している。

(4) nlohmann’s json https://github.com/nlohmann/json

nlohmann’s json はヘッダオンリーの json ライブラリである。上記の web ページの内容に従いダウンロードされたい。本シミュレータではダウンロードした nlohmann フォルダを/usr/local/include/にコピーして使用することを想定している。もし異なるディレクトリにインストールする場合は、本シミュレータの CMakeLists.txt においてパスを適切に設定されたい。

(5) pybind11_json https://github.com/pybind/pybind11_json

pybind11_json は nlohmann’s json を pybind11 に対応させるためのヘッダオンリーライブラリである。本シミュレータにおいては Release 0.2.11 に対して以下の変更を行っており、変更済みのファイルを本シミュレータの root/thirdParty/include/pybind11_json に同梱している。

・numpy 行列から json array への変換を追加(オリジナルはリストからの変換のみ)

・nlohmann’s json オブジェクトを Python 側から Mutable な形で参照できるようにプリミティブ型への自動変換を無効化

(6) Magic Enum C++ https://github.com/Neargye/magic_enum
Magic Enum C++は enum クラスの取り扱いを便利にするためのヘッダオンリーライブラリである。上記の web ページの内容に従いダウンロードされたい。本シミュレータではダウンロードした magic_enum.hpp を/usr/local/include/magic_enum/magic_enum.hpp にコピーして使用することを想定している。もし異なるディレクトリに保存する場合は、本シミュレータの CMakeLists.txt においてパスを適切に設定されたい。

(7) thread-pool https://github.com/bshoshany/thread-pool
thread-pool は C++17 においてスレッドプールの機能を提供するヘッダオンリーライブラリである。本シミュレータではヘッダファイルを/usr/local/include/thread-pool/thread_pool.hpp に置くことを想定している。もし異なるディレクトリに保存する場合は、本シミュレータの CMakeLists.txt においてパスを適切に設定されたい。

(8) Boost https://www.boost.org/
本シミュレータは Boost ライブラリのうち boost::uuids、boost::math::tools::toms748_solve 及び boost::math::ellint_2 を使用している。本シミュレータでは apt-get によりインストールすることを想定しており、CMake の機能を用いてインクルードパスを検索することとしている。

#### 1.2.2.3 Python 環境の立ち上げ

Python の仮想環境を使用する場合、インストール対象の環境を予め activate しておくこと。

#### 1.2.2.4 Python モジュールとしてのビルド及びインストール

本シミュレータの root ディレクトリ上で単に

```
pip install .
```

を実行することでインストールが可能である。
なお、このようにしてインストールしたモジュールはサンプルモジュールを含まない。サンプルモジュールのビルド及びインストールについては後述。

##### 1.2.2.4.1 インストールオプションについて

(1) アドオンの指定
デフォルトでは root/setup.py の冒頭において include_addons に記載されているもののみがインストールされる。アドオンを追加したい場合は

```
pip install . --global-option="--addons=addon1,addon2"
```

のようにコンマ区切りで指定して追加することができる。また、アドオンを除外したい場合は

```
pip install . --global-option="--ex-addons=addon3,addon4"
```

のように指定して除外することができる。なお、両方のオプションに同じアドオンを指定した場合は除外の方が優先される。
(2) C++部分のデバッグビルド
デフォルトではリリースビルドとして C++部分がビルドされるが、gdb 等を用いてデバッグを行いたい場合のためにデバッグビルドにも対応している。

```
pip install . --global-option="--Debug"
```

とするとデバッグビルドとなる。
(3) C++部分で MSYS を使用する場合
Windows 上で使用する場合でビルドツールに MSYS を使用したい場合は、

```
pip install . --install-option="--MSYS"
```

とする。

#### 1.2.2.5 深層学習用フレームワークのインストール

本シミュレータ及びサンプルの全ての機能を使用するためには PyTorch と Tensorflow(Tensorboard)が必要であるため、各ユーザーは自身の所望のバージョンのこれらのフレームワークを別途インストールしておく必要がある。なお、PyTorch については一部のアドオンの requirements に含まれているため、未インストールだった場合は pip 経由で自動的にインストールされる。

#### 1.2.2.6 サンプルモジュールと追加ライブラリのビルド及びインストール

本シミュレータは独自の行動判断モデルや報酬関数等を定義するためのサンプルとして、独立した Python モジュールとして動作する形式のサンプルを root/sample/OriginalModelSample として同梱している。学習サンプルを動作させるためにはインストールが必須であり、シミュレータ本体と同様に、root/sample/OriginalModelSample 上で `pip install .` と実行することでインストールできる。また、simulator_dist/requirements.txt に記載のライブラリを simulator_dist ディレクトリ直下で `pip install -r requirements.txt` と実行することでインストールする。

#### 1.2.2.7 動作確認用サンプルの実行

本シミュレータの root/sample/MinimumEvaluation 上で端末を開き、以下の通り実行する(GUI 対応ありの環境)。

```
python evaluator.py "Rule-Fixed" "Rule-Fixed" -n 1 -v -l "./result"
```

可視化しながら 1 回戦闘が行われ、その結果を保存したログが./result に保存されていれば正しく動作している。GUI の無い環境で使用したい場合は、-v のオプションを省略することで可視化を無効化できる。また、このスクリプトの詳細は 7.３項に記載している。

# ２本シミュレータの機能概要

本項では本シミュレータの全体像について概要をまとめる。

本シミュレータは、内部の処理を C++で記述したものを pybind11 経由で Python モジュールとして公開する形で作成されている。

## 2.1 クラス、モデル、ポリシーの考え方及び Factory によるインスタンス生成

本シミュレータにおいて、クラス、モデル、ポリシーについて以下のように定義するものとする。

クラス　…プログラム上で実装されたクラスそのもの
（例：レーダクラス、誘導弾クラス）

モデル　…あるクラスに対して、特定のパラメータセットを紐付けたもの
（例：50km 見えるレーダ、100km 見えるレーダ）

ポリシー…与えられた Observation に対して対応する Action を計算するもの
（例：ニューラルネットワーク、ルールベースなど）

このうち、クラスとモデルは本シミュレータの内部で管理されるものであり、ポリシーは外部の、例えば強化学習ライブラリによって管理されるものである。

ある登場物のインスタンスを生成する際にはクラスまたはモデルを指定することで行うものとし、本シミュレータはクラスとモデルを登録して共通のインターフェースでインスタンスを生成できる Factory クラスを実装している。また、インスタンスの生成時にはモデルのパラメータセットのみでなく、そのインスタンスの初期状態や「親」となる登場物等、インスタンス固有のパラメータセットも必要となるため、本シミュレータにおいてはモデルのパラメータセットを modelConfig という名称の json 型変数で、インスタンス固有のパラメータセットを instanceConfig という名称の json 型変数で取り扱うこととしている。クラス名を指定してインスタンスを生成する場合は Factory に対してクラス名とともに modelConfig と instanceConfig を与え、モデル名を指定して生成する場合はモデル名とともに instanceConfig を与えることで行う。登録方法、生成方法の詳細は 4.１項に記載する。

## 2.2 シミュレーションの登場物

本シミュレータの登場物は大きく 2 種類に分けられる。一つはシミュレーション中に実際に様々な行動をとる主体となる Asset であり、もう一つは勝敗の判定や得点、報酬の計算、あるいはログの出力や場面の可視化等、Asset の動きに応じて様々な処理を行いシミュレーションの流れを管理する Callback である。両者はともに Factory に登録して生成するものとし、２．１項で述べた modelConfig と instanceConfig をコンストラクタ引数にとる共通の基底クラス Entity を継承したものとして実装されている。Asset と Callback はそれぞれ図２．２－１のように、役割に応じてもう一段階細かい分類をしている。

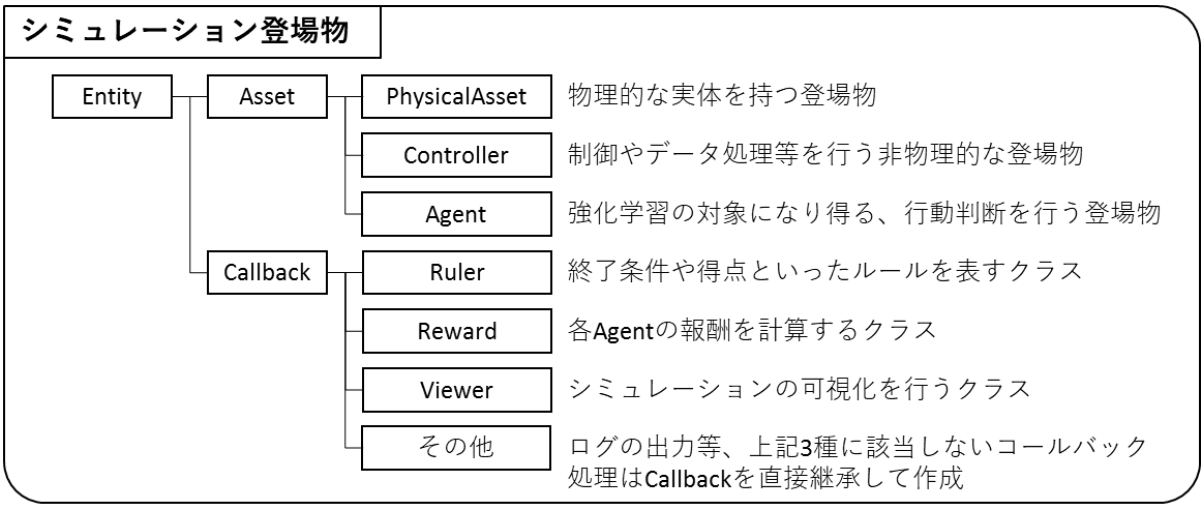


図 2.２－１　シミュレーション登場物の分類

### 2.2.1 Asset

Asset は、観測(perceive)、制御(control)、行動(behave)の 3 種類の処理をそれぞれ指定された周期で実行することで環境に作用する。また、各 Asset の処理の中には他の Asset の処理結果に依存するものがあるため、同時刻に処理を行うこととなった際の処理優先度を setDependency 関数によって指定することができる。

Asset は自身の生存状態を返す関数 bool isAlive()と、生存状態を false にするための関数 void kill()を持つ。生存中のみ前述の 3 種類の処理が実行され、一度生存状態が false となった Asset はそのエピソード中に復活することはない。ただし、インスタンスとしてはエピソードの終了まで削除せずに維持しており、例えば Callback 等から特定の処理のために変数を参照することができるようにしている。

また、各 Asset は共通のメンバ変数として、「他者が観測してよい自身の情報」である observables と、「自身の行動を制御するための情報」である commands を json 型変数として持ち、他の Asset が具体的にどのようなクラスであるかを気にせずに相互作用を及ぼすインターフェースとして用いることができる。ただし、他 Asset との相互作用を必ずしもこれらの変数を介して行わなければならないものではなく、各クラスの定義次第では直接互いのメンバ変数や関数を参照することも許容される。

Asset は大きく PhysicalAsset、Contoller、Agent の 3 種類のサブクラスに分類され、それぞれの概要は以下の通りである。

#### 2.2.1.1 PhysicalAsset

PhysicalAsset は Asset のうち物理的実体を持つものを表すクラスであり、独立した Asset として生成されるものと、他の PhysicalAsset に従属してその「子」として生成されるものとする。

自身の位置や速度、姿勢といった運動に関する状態量(4.2 項)を持つほか、従属関係にあるものについてはその運動が「親」に束縛されているか否かを指定することができる。また、自らの「子」となる PhysicalAsset と Controller を生成することができる。

#### 2.2.1.2 Controller

Controller は Asset のうち物理的実体を持たないものを表すクラスであり、一つの PhysicalAsset に従属してその「子」として生成されるものである。PhysicalAsset の複雑な制御を実現するために用いることを想定しているが、control だけでなく、必要に応じて perceive と behave で処理を行うことも許容される。

また、自らの「子」となる Controller(仕様上は PhysicalAsset も)を生成することができる。

#### 2.2.1.3 Agent

Agent は Controller と同様に物理的実体を持たないものであり、一つ以上の PhysicalAsset に従属してその「子」として生成されるものとする。Controller との最大の違いは、強化学習の対象として、一定周期でシミュレータの外部に Observation を供給し、外部から Action を受け取る役割を持っていることと、「親」である PhysicalAsset の実体にはアクセスできず、前述の observables と commands を介して作用することしか認められていないことである。

Agent は親 Asset の observables を読み取って Observation を生成する makeObs 関数と、Action を引数として受け取り自身のメンバ変数 commands を更新する deploy 関数を持っており、親 Asset は Agent の commands を参照して自身の動作を決定する。また、Agent は perceive、control、behave の処理も行えるため、これらを活用して Observation、Action の変換のために様々な中間処理を行うことも許容される。

### 2.2.2 Callback

Callback は、シミュレーション中に周期的に呼び出される処理を記述してシミュレーションの流れを制御するクラスであり、次の 11 種類のタイミングで対応するメンバ関数が呼び出される。

(1) onEpisodeBegin…各エピソードの開始時(＝reset 関数の最後)
(2) onValidationEnd…各エピソードの validate 終了時
(3) onDeployAction…step 関数の開始時に、外部から与えられた Action を各 Agent に配分する前
(4) onStepBegin…step 関数の開始時に、外部から与えられた Action を Agent が受け取った直後

(5) onInnerStepBegin…各 tick の開始時(＝Asset の control の前)
(6) onInnerStepEnd…各 tick の終了時(＝Asset の perceive の後)
(7) onStepEnd…step 関数の終了時（＝step 関数の戻り値生成の前または後)
(8) onMakeObs…step 関数の戻り値として observation を生成する直前
(9) onEpisodeEnd…各エピソードの終了時（step 関数の戻り値生成後)
(10) onGetObservationSpace…get_observation_space において戻り値を返す直前
(11) onGetActionSpace…get_action_space において戻り値を返す直前

Callback は大きく Ruler、Reward、Viewer、その他の 4 種類に分けられ、それぞれの概要は以下の通りである。

#### 2.2.2.1 Ruler

Ruler は、エピソードの終了判定の実施と各陣営の得点の計算を主な役割としている、名前の通りシミュレーションのルールを定義するクラスである。そのため、単一のエピソード中に存在できる Ruler インスタンスは一つのみに限られる。また、Ruler は Asset と同様に「観測してよい情報」である observables を json 型変数として持っており、Agent クラスからは得点以外には observables にしかアクセスできないようにしている。

#### 2.2.2.2 Reward

Reward は、エピソードにおける各 Agent の報酬の計算を行うためのクラスであり、Ruler と異なり、単一のエピソード中に複数存在させることができる。Reward は陣営単位の報酬を計算するものと Agent 単位の報酬を計算するもの 2 種類に分類でき、生成時に計算対象とする陣営や Agent の名称を与えることで個別に報酬関数をカスタマイズできる。

#### 2.2.2.3 Viewer

Viewer は、エピソードの可視化を行うためのクラスである。Viewer は Ruler と同様、単一のエピソード中にただ一つのインスタンスのみ存在できる。

#### 2.2.2.4 その他

その他の Callback は特別な共通サブクラスはなく、Callback クラスを直接継承してよい。例えば、ログの出力や、次のエピソードのコンフィグの書き換えを行うために用いることができる。更には、前述の Ruler や Reward の処理結果や Asset の状態量を強引に改変したり、Agent から本来はアクセス出来ない情報を Agent に伝達したりすることも可能な作りとなっており、かなり広い範囲にわたってシミュレーションの挙動に干渉することができるものとしている。

また、その他に該当する Callback のうち、ログ出力に該当するものを Logger として通常の Callback とは明示的に分けて生成する。これは Logger が Viewer の可視化結果を参照できるようにするためであり、シミュレーション中は Logger 以外の Callback→Viewer→Logger の順で処理される。

## 2.3 シミュレーションの処理の流れ

本項では、シミュレーションの処理の流れの概要をまとめる。一般的な OpenAI Gym 環境と同じく、本シミュレータの外部から見た場合に各エピソードは初期 Observation を返す reset 関数から始まり、その後は Action を入力して Observation、Reward、Done、Info の 4 種類の値を返す step 関数を繰り返し、全ての Agent に対する Done が True となった時点で終了となる。2.2 2.2.1 項に挙げた Asset と、2.1 項で挙げたポリシーに関するエピソード中の処理フローは図 2.3－1 の通りである。現時点での実装では、全ての Agent の行動判断周期(ポリシーと Observation、Action をやりとりする周期)は同一とし、$n$[tick]ごとに行うものとしている。また、図中青色で示している step 関数内の $n$[tick]分の時間進行処理は、必ずしも全ての Asset が1[tick]ごとに処理を行うものではなく、各 Asset クラスまたはモデルごとに指定された周期で、実行すべき時刻が来たときに実行される。

また、2.2.2 項に挙げた Callback の各処理の実行タイミングは Callback の種類によって異なり、図 2.3－2 に示すタイミングで実行される。

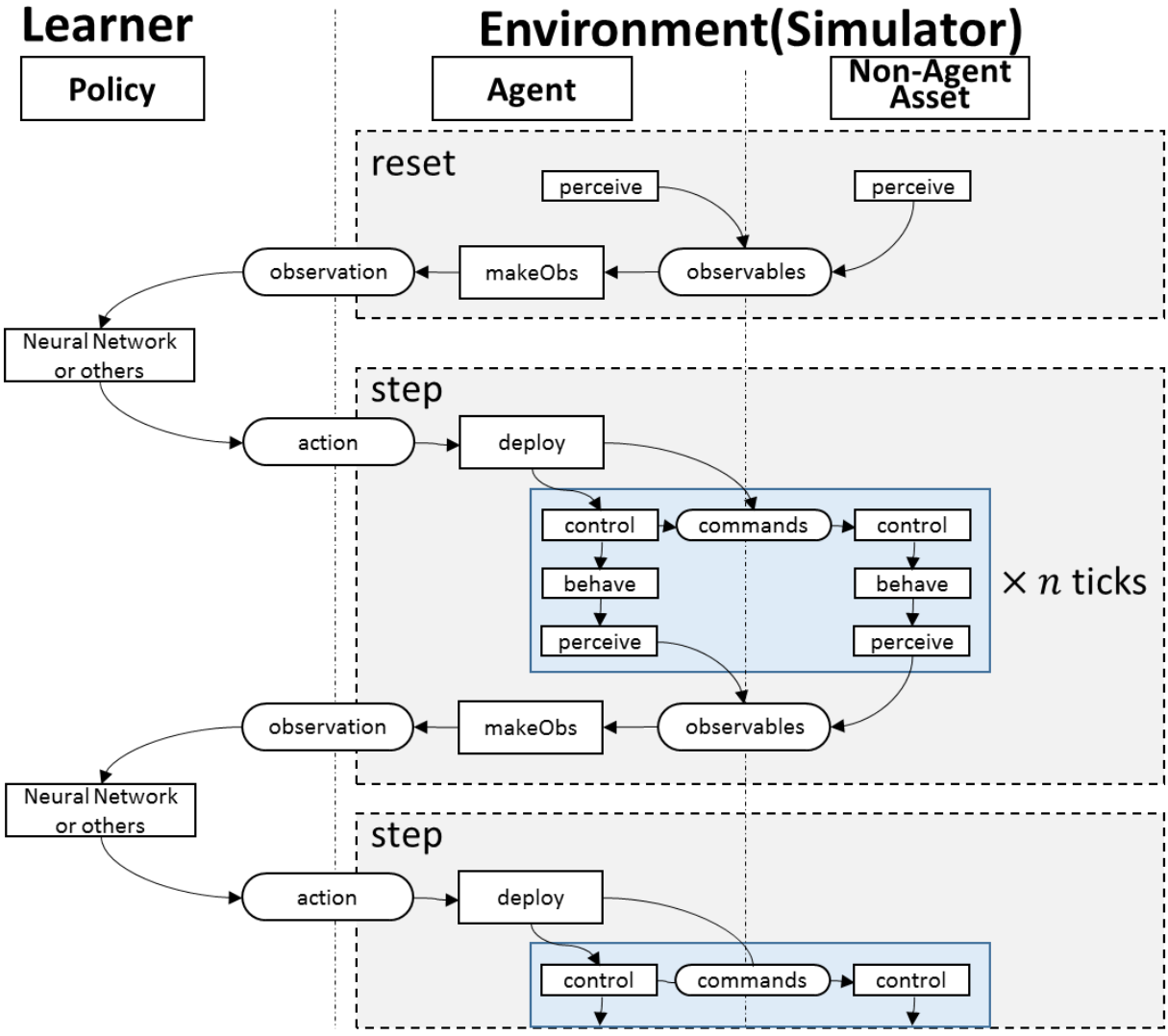


図 2.3－1　シミュレーション中の処理の流れ(行動判断に関するもののみ)

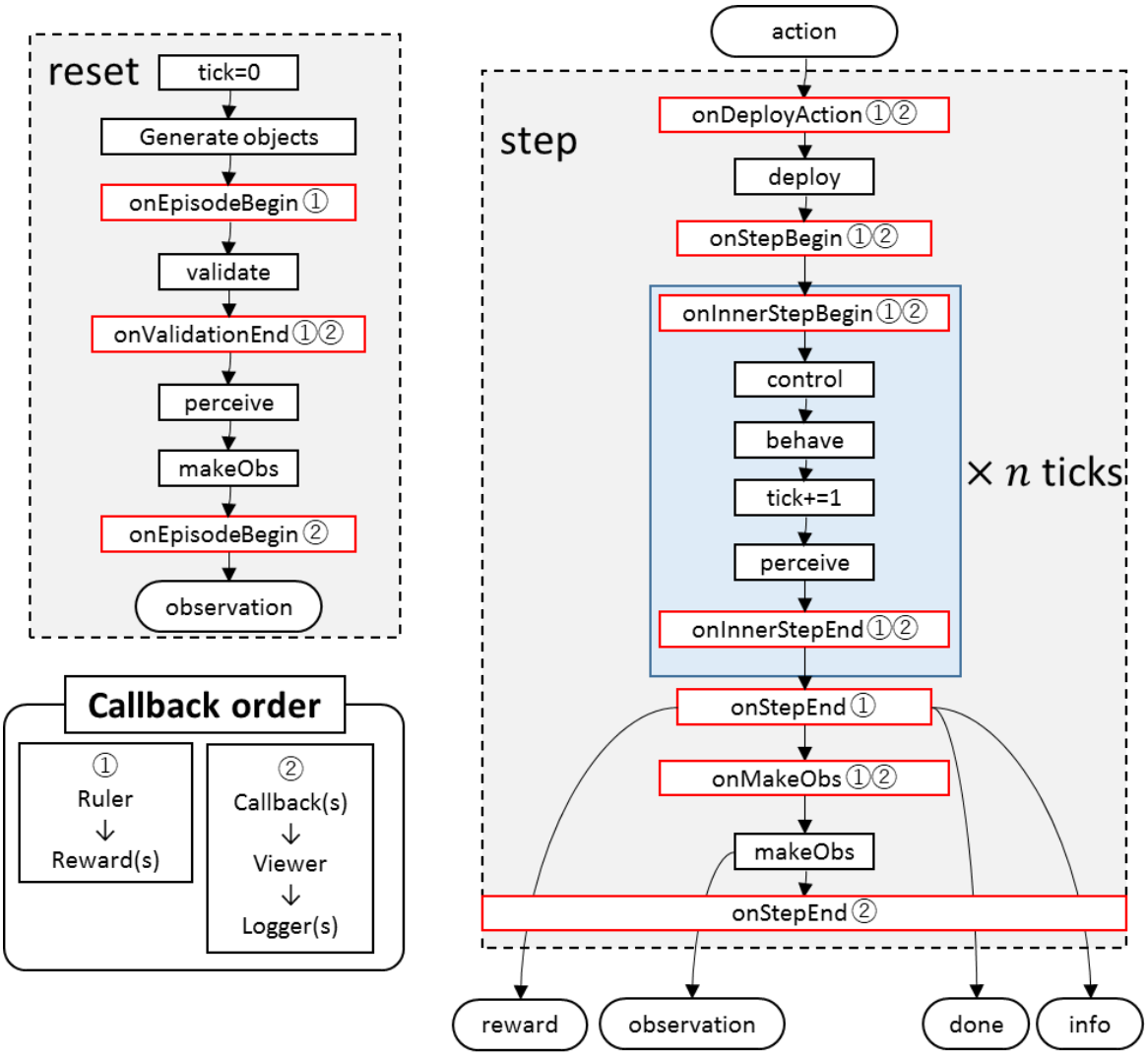


図 2.３－２　コールバックの処理タイミング

## 2.4 非周期的なイベントの発生及びイベントハンドラ

本シミュレータでは 2.３項で述べた周期的な処理のほかに、突発的なイベントを発生させ、それをトリガーとしたイベントハンドラの処理を実行する機能を有している。イベントハンドラの登録は任意のイベント名とコールバック関数オブジェクトの組を与えることで行い、イベントを発生させる際にはイベント名とイベントハンドラに対する引数の組を与えることで行う。イベントハンドラの引数はインターフェース共通化のため、json 型の変数一つとしている。

例えば、誘導弾の命中時や戦闘機・無人機の墜落時の得点増減は Asset からイベントを発生させ Ruler が登録したイベントハンドラを呼ぶことによって実現されている。

なお、イベントハンドラの登録は Callback からのみ可能であり、イベントの発生は Callback、PhysiacalAsset または Controller のいずれかからのみ可能である。

## 2.5 シミュレーションの実行管理及び外部とのインターフェース

前項までの登場物や処理フローの管理は SimulationManager クラスのインスタンスによって一元的に管理されており、ユーザーは SimulationManager のコンストラクタに json 形式のコンフィグを与えることで登場物やルール等の設定を行うことができる。また、現時点では一つの SimulationManager インスタンスはシングルスレッドで動作する。コンストラクタ引数は次の 4 個であり、config 以外は省略可能である。

(1)config…json 型(object または string、もしくはそれらの array)であり、string の場合はファイルパスとみなして json ファイルを読み込む。array の場合は各要素を順番に merge_patch で結合していく。最終的に得られる json は object でなければならず、キー"Manager"に自身のコンフィグを、キー"Factory"に Factory に登録するモデルのコンフィグを持っているものとする。
(2)worker_index…ワーカープロセスのインデックスであり、RLlib で使用されている通り。
(3)vector_index…プロセス内のインデックスであり、RLlib で使用されている通り。
(4)overrider…コンストラクタ内でコンフィグを置き換えるための関数オブジェクトであり、config,worker_index,vector_index を引数として新たな config を返すものとする。

### 2.5.1 Python インターフェース

本シミュレータの使用時には、SimulationManager クラスのインスタンスを直接生成するのではなく、以下のラッパークラスによる Python インターフェースを使用することを想定している。

#### 2.5.1.1 GymManager

gym.Env を継承したマルチエージェント環境として GymManager クラスを実装している。GymManager クラスのコンストラクタには、SimulationManager のコンストラクタが要求する４つの引数に相当する値を以下のような単一の dict として与える。

```
context = {
  "config": json(dict, str, or list[dict and/or str]),
  "overrider": Callable[[json, int, int], json]
  "worker_index": int,
  "vector_index": int
)
```

#### 2.5.1.2 SinglizedEnv

登場する Agent のうち指定した一つのみの Observation, Action を入出力し、それ以外の Agent の行動を環境内部で計算するようにしたシングルエージェント環境として SinglizedEnv クラスを実装している。マルチエージェント環境に対応していない強化学習ライブラリを使用したり、模倣学習時に学習対象の Agent を限定するために使用したりすることを想定している。入出力の対象とならなかった Agent については、コンストラクタにおいて 2.６項で示す StandalonePolicy オブジェクトを渡すことにより内部で Action の計算を行う。また、対象とする Agent はその fullName によって判定するものとし、コンストラクタに与えた判定用の関数で最初に True となった Agent を対象とする。コンストラクタ引数 context には GymManager が必要とするものに加え以下の値を追加したものである。

表 2.５－１　SinglizedEnv の context に追加すべき項目

| キー名 | 型 | 概要 |
|---|---|---|
| target | Union[<br>Callable[[str,bool],str],<br>str<br>] | 入出力対象の Agent を特定するための関数であり、Agent の fullName を引数にとって bool を返す関数。文字列を直接指定してもよく、その場合はその文字列で始まるかどうかを判定条件として扱う。 |
| policies | dict[str,StandalonePolicy] | ポリシー名をキーとした、内部で Action を計算するための StandalonePolicy オブジェクトの dict。使用しない場合は省略可能。 |
| policyMapper | Callable[[str],str] | Agent の fullName を引数にとり対応するポリシー名を返す関数。省略時は fullName の末尾の policyName にあたる部分がそのまま使用される。 |
| exposeImitator | bool | ExpertWrapper を target とする際、Expert 側でなく Imitator 側の値を出力するか否か。デフォルトは False。 |
| runUntilAllDone | bool | 対象とした Agent が done となった場合でも全 Agent が done となるまで内部の計算を続行するか否か。デフォルトは True。 |

#### 2.5.1.3 SimpleEvaluator

最小限の評価用環境として、全 Agent の行動判断を StandalonePolicy によって内部で処理し、GymManager を用いたエピソードの実行を自動で行う SimpleEvaluator クラスを実装している。

### 2.5.2 Agent 名の書式について

本シミュレータにおける Agent の命名規則は、インスタンス名、モデル名、ポリシー名を":"で繋いだものとする。例えばインスタンス名が"Agent1"、モデル名が"AgentModel1"、ポリシー名が"Policy1"である場合、Agent 名は"Agent1:AgentModel1:Policy1"となる。

また、Policy 名について"Internal"は予約語としており、Action の計算が不要で本シミュレータの内部で動作が完結するタイプの Agent を示すものとして扱っている。

### 2.5.3 コンフィグの書き換えについて

本シミュレータは、同一の SimulationManager インスタンスを複数エピソードにわたって繰り返し使い続けるものとしているが、複数のエピソードを並列に実行したい場合や、一定エピソードごとに登場物やルール、報酬等を変更したい場合も想定される。そのような場合に SimulationManager インスタンスを再生成することなく対応するために、SimulationManager インスタンス生成時と各エピソード開始時にコンフィグを書き換える機能を有している。書き換え対象は SimulationManager そのもののコンフィグと、Factory に登録しているモデルのコンフィグであり、クラスの置換には対応していない。

#### 2.5.3.1 インスタンス生成時の書き換え

SimulationManager のコンストラクタに与える config を複数使い分けることでもインスタンスごとに異なる環境として生成することが可能であるが、強化学習ライブラリの仕様によってはコンストラクタ引数を変えることが困難であることも想定される。そのため、SimulationManager のコンストラクタ引数として、config を書き換えるための overrider を与えることによって、コンストラクタ内部で config を書き換える処理を行うことを可能としている。

#### 2.5.3.2 エピソード開始時の書き換え

エピソード開始時の書き換えは、reset 関数の呼び出し前までに、SimulationManager の requestReconfigure 関数に Manager と Factory 登録モデルそれぞれの置換用 json を与えることで、次の reset 関数の先頭で行われるようになる。置換用 json は、元の json に対して適用すべき merge_patch として与えるものとする。使用方法としては、対戦ごとに各陣営の Agent を変更したり、学習の進捗に応じてルールや報酬を変更したりするような場合を想定しており、環境の外部から呼び出すことも、Callback クラスによって内部から呼び出すことも可能である。

## 2.6 ポリシーの共通フォーマット

通常、OpenAI gym 環境においてエピソードの実行は外部から制御されるため、ポリシーが Action を計算する処理の実装方法は制約が無い。しかし、例えば異なる手法で学習された複数の行動判断モデルどうしを戦わせて評価したい場合や、2.５.１.２項で述べたように一部の Agent について環境の内部で処理を済ませたい場合を想定して、本シミュレータでは StandalonePolicy クラスとしてポリシーの推論のための共通インターフェースを定義している。StandalonePolicy クラスは引数、返り値なしの reset 関数と、Observation 等を入力して Action を出力する step 関数の二つを有するものとし、それぞれ環境側の reset,step と対になるものである。step 関数の引数として与えられる変数は以下の通りである。

表 2.６－１　StandalonePolicy の step 関数の引数

| 引数 | 型 | 概要 |
|---|---|---|
| observation | Any | 環境から得られた、計算対象の Agent の observation |
| reward | float | 環境から得られた、計算対象の Agent の reward |
| done | bool | 環境から得られた、計算対象の Agent の done |
| info | Any | 環境から得られた、計算対象の Agent の info |
| agentFullName | str | 計算対象の Agent の完全な名称であり、<br>agentName:modelName:policyName の形をとる |
| observation_space | gym.spaces.Space | 与えられた Observation の Space |
| action_space | gym.spaces.Space | 計算すべき Action の Space |

# ３本シミュレータのその他の機能・仕様

本項では、本シミュレータが提供するユーティリティ機能や、実装上の細部仕様について概要をまとめる。

## 3.1 オブジェクトの保持方法

本シミュレータでは一部の純粋データ型として扱うクラスを除いて、shared_ptr として生成することとしており、Python 側で生成した場合も同様である。また、相互の循環参照を解決するために weak_ptr も用いており、これは Python 側では weakref.ref 互換のインターフェースにより()演算子を用いて本体にアクセスできるようになっている。

## 3.2 ・Accessor クラスによるアクセス制限

2.2.1.3 項で少し触れたが、メンバ変数、メンバ関数について他のオブジェクトからのアクセスを制限したい場合が存在する。しかし、Python 側からの自由な継承を可能としている本シミュレータの実装では、Python 側からアクセスできる変数、関数は原則として public 指定とする必要がある。このような状況でアクセス制限をかける方法として、オブジェクト本体への参照を用いるのではなく自身に紐づいた Accessor クラスを介してアクセスを提供する機能を実装している。

### 3.2.1 SimulationManager のアクセス制限

各登場物(Entity)は多くの場合、自身の処理を行うために SimulationManager の持つ情報や関数にアクセスする必要がある。しかし、Entity の種類によって適切なアクセス範囲は異なる。例えば時刻情報は基本的に全ての Asset に開示してよいし、場に存在する Asset の情報は Ruler に対してであれば全て開示すべきであるが Agent に対してはその Agent の「親」となる PhysicalAsset 以外の情報は開示すべきではない。このようなアクセス制限を実現するため、各 Entity には SimulationManager 本体への参照を保持させるのではなく、Entity の種類に応じた Accessor クラスを用意して Accessor 経由でアクセスさせることとしている。

### 3.2.2 Entity のアクセス制限

アクセス制限は SimulationManager に対してだけではなく、Ruler と PhysicalAsset に対してもかけられるようにしている。これは特に、Agent 側から本来はアクセス不可の情報に不正にアクセスできてしまわないようにし、observables と commands による制限された情報共有を実現するためのものである。ただし、この制限はあくまで Agent 単体での不正アクセスを防止するだけのものであり、Callback を上手く使えばほぼ任意の情報を Agent に与えることが可能であるため、学習を加速させるため等にそのような利用方法をとることは許容される。

## 3.3 json 経由での参照渡し

本シミュレータでは多くの機能で json 形式でのデータ授受を採用しているが、json では非プリミティブ型の参照渡しができない。本シミュレータではこれを実現するために、次の条件を満たすクラスのインスタンスについて、shared_ptr と weak_ptr を json と相互変換できるようにしている。

(1) std::enable_shared_from_this<T>を継承している。
(2) T を別途 PtrBaseType として typedef している。

相互変換は生ポインタを intptr_t にキャストして強引に実現しているものであるため、扱いには気をつける必要がある。また、当然ながらアドレス空間を共有しているオブジェクトどうしでしかこの交換は意味をなさない。

なお、現時点で上記の条件を満たす基底クラスは Entity、SimulationManager 及びこれらの Accessor である。

## 3.4 json による確率的パラメータ設定

ランダム要素を含んだ登場物を単一のコンフィグファイルから動的に生成するための仕組みとして、getValueFromJsonX というユーティリティ関数を実装している。X には第二引数以降の与え方によって表 3.４－１に示す通り K,R,D のいずれか 0～3 文字が入る。また、第一引数は値の生成元となる json であり、値の生成方法に応じて表 3.４－２のように記述する。

表 3.４－１　getValueFromJson の引数と関数末尾の文字の関係

| 引数の順序 | 関数名の X | 対応する引数 | 意味 |
|---|---|---|---|
| 1 | − | const nl::json& j | 生成元となる json。 |
| 2 | K | const std::string& key | 生成元 json のうち値生成のために参照するキーを与える。そのキーに対応する値の書式はキーを指定しなかった場合の生成元 json と同じ。 |
| 3 | R | <template class URBG><br>URBG& gen | 乱数生成器を与える。通常は Entity クラスのメンバ変数 random として保持している std::mt19937 を与えることを想定している。もし numpy.random を使用したい場合は別途 numpy.random を C++側から使用するラッパークラスを定義し、std::mt19937 と同等のインターフェースをもたせる必要がある。 |
| 4 | D | <template ValueType><br>const ValueType&<br>defaultValue | 生成元 json が指定したキーを持っていなかった場合のデフォルト値を与える。 |

表 3.４－２　生成元 json の記述方法

| 生成方法 | 書式（ユーザ指定の変数を<>で表す） | 生成される値 |
|---|---|---|
| 定数 | {"type":"direct",<br>"value":<value>} | <value>で指定した値 |
| 一様分布 | {"type":"uniform",<br>"dtype":<dtype>,<br>"low":<low>,<br>"high":<high>} | <dtype>は"int"または"float"とし、省略した場合は"float"とみなす。<br><low>と<high>はスカラー、ベクトルまたは行列値が許容され、要素ごとに対応する区間の値が一様にサンプリングされる。<br>"int"のときは<low>以上<high>以下の整数となり、"float"のときは<low>以上<high>未満の実数となる。<br>また、要素数１の次元は削除される。 |
| 正規分布 | {"type":"normal",<br>"mean":<mean>,<br>"stddev":<stddev>} | 平均<mean>、標準偏差<stddev>の正規分布に従いサンプリングされる。<mean>と<stddev>はスカラー、ベクトルまたは行列値であり、一様分布の場合と同様に要素ごとに対応する分布からサンプリングされる。 |
| 択一 | {"type":"choice",<br>"candidates":<candidates>,<br>"weights":<weights>} | <candidates>から重み<weights>に従い一つ選択し、選択されたものに対し再帰的に getValueFromJsonR を適用した値が返される。<candidates>と<weights>はいずれもベクトルで与える。 |
| 直接 | 上記以外の任意の値 | 生成元 json が object の場合、各キーに対応する値に再帰的に getValueFromJsonR を適用した object が返される。<br>生成元 json が array の場合、各要素に再帰的に getValueFromJsonR を適用した array が返される。<br>それ以外の場合、生成元 json の値がそのまま返される。 |

## 3.5 CommunicationBuffer による Asset 間通信の表現

本シミュレータにおいて、Asset 間通信の簡易的な模擬として、CommunicationBuffer を実装している。CommunicationBuffer は一つの json(object)を共有バッファとして参加中の Asset 間で読み書きを行うものとなっている。データの送信時は send 関数を用いて送信データを json 形式でバッファの更新方法(REPLACE または MERGE)とともに与える。データの受信時は receive 関数に受信したいキーを与えることで、該当するキーの更新時刻とデータを得ることができる。現時点では、キーの指定はバッファの最上位階層のみであり、子階層の指定には対応していない。

CommunicationBuffer の生成には、participants と inviteOnRequest を指定する必要がある。いずれも json(array)であり、前者は無条件で参加する Asset のリスト、後者は Asset 側から要求があった場合のみ参加させる Asset のリストである。これらのリストの要素は、Asset 種別と Asset 名（4.４.１項における fullName）をコロンで区切った文字列(例："PhysicalAsset:team/group/name")とする。Asset 名については正規表現に対応しており、マッチした全ての Asset を対象に含める。

必ずしも全ての Asset 間情報共有に CommunicationBuffer を用いる必要はなく、親子関係にある Asset 間で親が子のメンバ変数を直接参照するというようなことも許容される。また、より複雑に通信の遅延や距離による接続可否等を模擬したい場合は、ネットワーク状況を管理する PhysicalAsset を新たに定義し、それらの間で CommunicationBuffer を繋いだうえでそれらの処理の中で遅延等の表現を行うことで実現可能となる。

## 3.6 模倣学習のための機能

本シミュレータは、異なる Observation、Action 形式を持つ Agent の行動を模倣するような模倣学習を行いやすくするための機能を有している。この機能の根幹をなすのは Agent クラスの派生クラスとして実装された ExpertWrapper クラスである。ExpertWrapper クラスはメンバ変数として expert(模倣される側)と imitator(模倣する側)の２つの Agent インスタンスを保持し、環境外部のポリシーから供給された expert の Action を最も類似した imitator の Action に変換したうえで両 Agent の処理ループを回し、両方又は一方の observation を環境外部に返すものである。step 関数における ExpertWrapper の処理フローは図 3.６－１の通りである。

Action の変換にあたっては、親 Asset に渡される commands に加え、その行動の意図を何かしらの形で表現する decision(4.５.３項を参照)を observables として生成しておき、これらを imitator 側でオーバーライドした convertActionFromAnother 関数に与えることで imitator 側の行動空間内で最も類似した Action を計算し、メンバ変数として保持する仕様としている。

親 Asset に渡される commmands は、expert が生成したものと imitator が生成したものの好きな方を選択できるようになっている。また、imitator の control 関数を実行する際に、expert の decision と commands を反映して修正を行いたい場合も想定されるが、そのような場合には imitator 側のクラスで controlWithAnotherAgent 関数をオーバーライドしておくことで対応可能となっている。なお、もしオーバーライドされていなければ通常の control 関数がそのまま呼び出される。

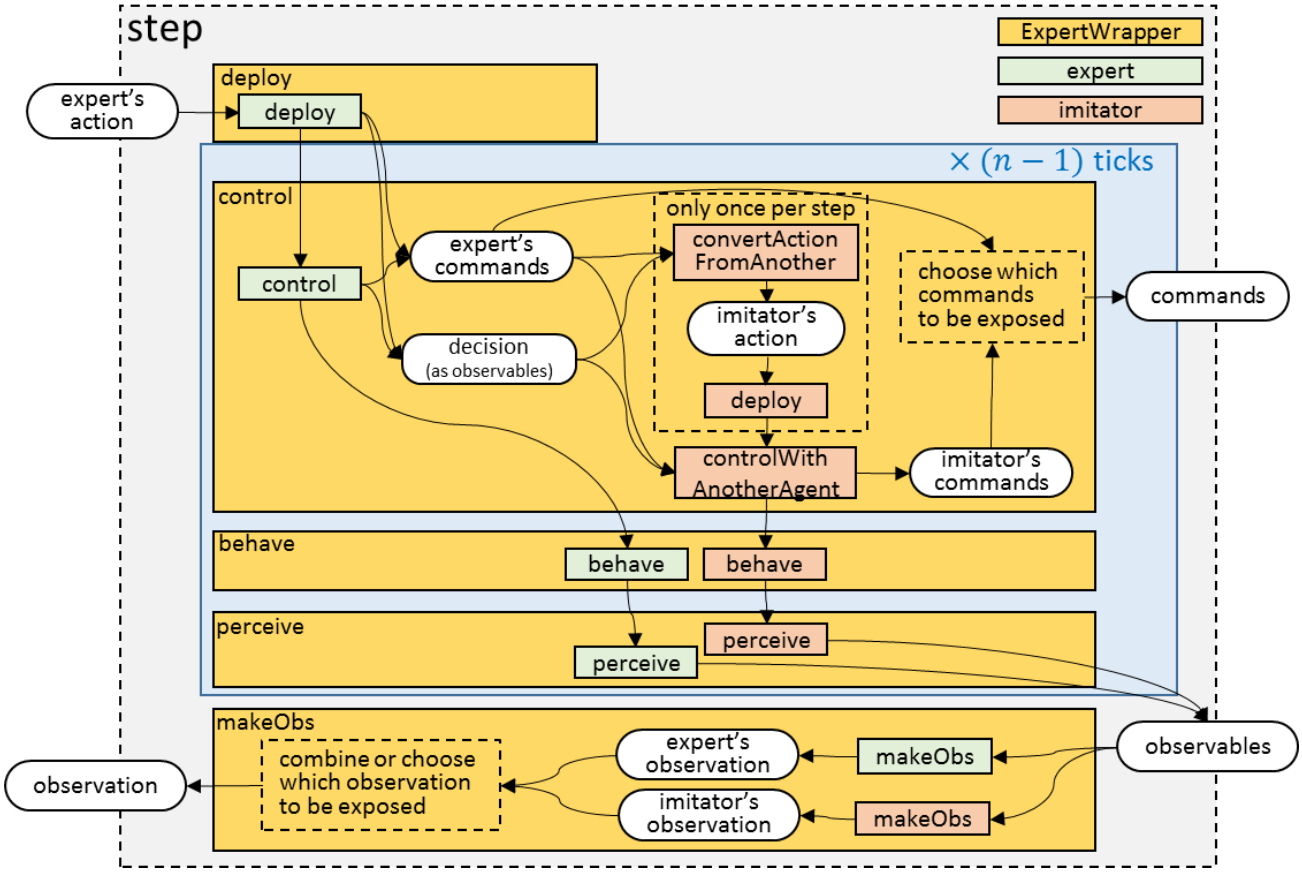


図 3.6－1　ExpertWrapper の step 関数中の処理フロー

## 3.7 複数の行動判断モデルを用いた対戦管理機能(MatchMaker)

本シミュレータのアドオンとして、複数の行動判断モデルを用いて Self-Play 等の対戦管理を行うための機能を root/addons/MatchMaker に実装している。ただし、多くの処理を実際に学習を行うプログラム側に依存するため、本追加機能のみでは実現できない。同梱の学習サンプルにおいて ray RLlib と HandyRL で MatchMaker を使用する例を実装しているため参考とされたい。

### 3.7.1 MatchMaker 及び MatchMonitor

複数の Policy の重み保存、読み込みを管理しつつ、その勝率等に応じたエピソードごとの対戦カードを生成するための基底クラス群を root/addons/MatchMaker/MatchMaker.py に実装しており、以下の 2 つのクラスからなる。

(1) 一つだけ生成し、全体の管理を行う MatchMaker

(2) 実際にエピソードを実行するインスタンスごとに生成し、対戦結果を抽出して MatchMaker に渡す result を生成する MatchMonitor

#### 3.7.1.1 使用方法

(1) MatchMaker のコンストラクタ引数について

MatchMaker クラスのコンストラクタは以下のような dict 型の引数 config をとる。必須のキーは以下の通り。

```
config={
  "restore": None or str, #チェックポイントを読み込む場合、そのパスを指定。
  "weight_pool": str, #重みの保存場所。<policy>-<id>.dat のように Policy 名と
                      #重み番号を示すファイル名で保存される。
  "policy_config": { #Policy に関する設定。Policy 名をキーとした dict で与える。
    <Policy's name>: Any, #派生クラスごとの様式に従った各 Policy の設定
    ...
  },
  "match_config": Any #対戦カードの生成に関する指定。派生クラスで自由に定める。
}
```

(2) MatchMaker と MatchMonitor の生成

学習プログラムにおいて、メインプロセスで MatchMaker インスタンスを一つだけ生成する必要がある。また、MatchMonitor はエピソード生成を行う Worker インスタンスごとに生成する必要がある。

(3) result の伝達

学習プログラムの仕様に適した方法にて、各 MatchMonitor の onEpisodeEnd が抽出したエピソードの結果を MatchMaker の onEpisodeEnd に供給する必要がある。

(4) 対戦カードを環境に反映するための機構の準備

学習プログラムにおいて、生成された対戦カードを用いて環境の生成を行う機構も必要である。

(5) 重みの読み書きや初期化を実際に行う処理は学習プログラム側で準備する必要がある。

#### 3.7.1.2 カスタマイズの方法

実際に MatchMaker を使用するうえでは、環境の仕様に応じて以下の関数群を適宜オーバーライドして使用する。

(1) MatchMaker.makeNextMatch のオーバーライド

makeNextMatch は対戦グループ(〇〇リーグのようなイメージ)を表す何らかの変数 matchType を引数にとり、何らかの対戦カードを返す関数であり、各ユーザーが独自の対戦方式を実装するものである。対戦カードは dict で表現され、場に登場するチームごとにどの Policy のどの重みを使うかをキー"Policy"と"Weight"により指定する。また、同名の Policy で異なる重みを複数同時に使用したい場合等のために、それらを区別するための"Suffix"を指定可能である。重み番号は基本的に、

負数(-1)･･･学習中の現物であり、この Policy が経験した軌跡が学習に使用される

0･･･学習中重みの最新のコピー(随時更新される)

自然数･･･過去のある時点で保存されたコピー

とすることを想定しているが、MatchMaker を使用する学習プログラムの書き方次第である。また、これら以外にも必要な情報があれば適宜追加してもよい。まとめると、以下のような dict となる。

```
matchInfo={
  <Team's name>:{
    "Policy": str, #team を動かす Policy の名前
    "Weight": int #team を動かす Policy の重み番号
    "Suffix": str #team を動かす Policy の接尾辞
  },
  …
}
```

(2) MatchMaker.onEpisodeEnd のオーバーライド

いずれかの対戦が終わったときに呼び出し、その対戦を実施したインスタンスにおける次の対戦カードを生成して返す関数。引数は、終了した対戦の matchInfo と、対応する MatchMonitor の onEpisodeEnd で生成された result を与えるものとする。また、返り値として、重み保存要否を記した dict を返すこととしている。基本的には重み保存を行う Policy のみを対象として以下のような dict として記述することを想定しており、学習プログラム側で重みの保存や初期化を行うものとしている。

```
{
  <Policy's name>:{
    "weight_id": int, #保存先の重み番号
    "reset": bool #保存後に重みをリセットするかどうか
  },
  …
}
```

(3) MatchMonitor.onEpisodeEnd のオーバーライド

環境インスタンスごとに対戦結果を抽出して MatchMaker に渡る result を生成する関数。MatchMaker の onEpisodeEnd で重み保存の判定や次回の対戦カードの生成を行うために必要な情報があれば、環境インスタンスから抽出して返す。

(4) MacthMaker.checkInitialPopulation のオーバーライド

学習開始時の重みを weight_pool に追加するかどうかを返す関数。返り値の形式は onEpisodeEnd と同じ。用途としては、別の学習済モデルを読み込んで開始したときにその初期状態を対戦候補に含めたいような場合等が考えられる。

(5) MatchMaker.get_metrics のオーバーライド
Tensorboard ログ等に記録するための値を格納した dict を返す関数として、必要に応じてオーバーライドする。

(6) MatchMaker.initialize, load, save のオーバーライド
MatchMaker の初期化やチェックポイントの生成、読み込みを行うための各関数を必要に応じてオーバーライドする。

### 3.7.2 BVRMatchMaker 及び BVRMatchMonitor

MatchMaker 及び MatchMonitor を継承し、2 陣営による空対空目視外戦闘を対象とした対戦管理を行うサンプルとして、root/addons/MatchMaker/BVRMatchMaker.py に BVRMatchMaker 及び BVRMatchMonitor を実装している。

#### 3.7.2.1 使用時の前提

(1) 陣営名は"Blue"と"Red"とすること。SimulationManager の Ruler の設定もこれに合わせること。
(2) 学習対象の Policy は"Learner"とし、初期行動判断モデルは"Initial"とする。
SimulationManager の AgentConfigDispatcher にはサンプルに示すような形式で各 Policy 名に対応する alias を登録しておくこと。
(3) 一つの陣営を操作する Policy は一種類とする。つまり、SimulationManager 側で対応する Agent も一種類となる。
(4) 各 Policy は 1 体で 1 陣営分を動かしても、1 体で 1 機を動かしてもよく、対戦カードにはその設定を表す bool 型の"MultiPort"キーを追加している。
(5) 使用する環境クラス(GymManager 又はその派生クラス)を wrapEnvForBVRMatchMaking でラップしたものを使用し、学習プログラムにおいて env.setMatch 関数を呼び出して対戦カードの反映を行うこと。
(6) 対戦カードの Suffix は、学習中重みの場合は""(空文字列)、過去の重みを Blue 側で使用する場合は"_Past_Blue"、過去の重みを Red 側で使用する場合は"_Past_Red"とする。これらの Suffix を付加した名称に対応する Policy インスタンスを学習プログラム側で用意すること。

#### 3.7.2.2 ログの記録

このサンプルでは対戦ログの記録についても実装例を示している。学習の進捗チェック用に、AlphaStar[1]で用いられている PayOff と、イロレーティングを算出しており、それらを get_metrics で Tensorboard ログとして使用できる形式で出力するとともに、各エピソードの結果を csv ファイルに出力する。

[1] Vinyals, Oriol, t al. "Grandmaster level in StarCraft Ⅱ using multi-agent reinforcement learning," Nature 575.7782 (2019): 350-354.

#### 3.7.2.3 対戦カードの生成方式

このサンプルでは、以下のように対戦カードを選択する。
Blue 側･･･常に学習中の"Learner"
Red 側･･･一定エピソード数(warm_up_episodes)の経過前は常に初期行動判断モデル("Initial")とし、経過後は(1)学習中重みの直近のコピー、(2)初期行動判断モデル("Initial")、(3)過去の保存された重みから一様分布で選択、の 3 種類を 4:4:2 の割合で選択

### 3.7.2.4 config の書式

BVRMatchMaker クラスのコンストラクタに与える引数の書式は以下の通りである。

```
config={
  # 基底クラスで指定されたもの
  "restore": None or str, #チェックポイントを読み込む場合、そのパスを指定。
  "weight_pool": str, #重みの保存場所。<policy>-<id>.dat のように Policy 名と
                      #重み番号を示すファイル名で保存される。
  "policy_config": { #Policy に関する設定。Policy 名をキーとした dict で与える。
    <Policy's name>: {
      "active_limit": int, #保存された過去の重みを使用する数の上限を指定する。
      "is_internal": bool, #SimulationManager における Internal な Policy かどうか。
      "populate": None or { #重み保存条件の指定
        "firstPopulation": int, #初回の保存を行うエピソード数。
                                #0 以下の値を指定すると一切保存しない。
        "interval": int, #保存間隔。0 以下の値を指定すると一切保存しない。
        "on_start": bool, #開始時の初期重みで保存するかどうか。省略時は False。
        "reset": float, #重み保存時のリセット確率(0～1)
      },
      "rating_initial": float, #初期レーティング
      "rating_fixed": bool, #レーティングを固定するかどうか。
      "initial_weight": None or str, #初期重み(リセット時を含む)のパス
      "multi_port": bool, #1 体で 1 陣営分を動かすか否か。省略時は False。
    }
    ...
  },
  "match_config": {#対戦カードの生成に関する指定。
    "warm_up_episodes": int, #学習初期に対戦相手を"Initial"に固定するエピソード数。
  },
  # このクラスで追加されたもの
  "seed": None or int, #MatchMaker としての乱数シードを指定。
  "log_prefix": str, 全対戦結果を csv 化したログの保存場所。
}
```

# ４主要クラスの使用方法

本項では、主要なクラスの使用方法の概要をまとめる。

## 4.1 Factory

Factory は登録されたクラスとモデルを、対応する基底クラスごとにグループ分けして管理する。グループ名は PhysicalAsset、Controller、Agent、Ruler、Reward、Viewer、Callback の 7 種類であり、同一グループの中では名称の重複は認められない。また、本シミュレータの実装においてはこのグループ名を baseName と称している。

クラスの登録は static メンバとして同一アドレス空間内の全 SimulationManager インスタンスで共有されるが、モデルの登録についてはクラスと同じく static メンバとして登録する方法と、各 SimulationManager の config 経由でインスタンス固有のモデルとして登録する方法の 2 種類が用意されている。登録は以下のように行う。

### 4.1.1 クラスの登録

グループ名…PhysicalAsset、クラス名…ClassName の場合、以下のように登録する。

(1) C++からの登録

```
#include <ASRCAISim1/Factory.h>
#include <ASRCAISim1/PhysicalAsset.h>
class ClassName:PhysicalAsset{
};
FACTORY_ADD_CLASS(PhysicalAsset,ClassName)
```

(2)Python からの登録

```
from ASRCAISim1.common import addPythonClass
from ASRCAISim1.libCore import PhysicalAsset
class ClassName(PhysicalAsset):
  ...
addPythonClass("PhysicalAsset","ClassName",ClassName)
```

### 4.1.2 モデルの登録

グループ名…PhysicalAsset、クラス名…ClassName、モデル名…ModelName の場合、

```
modelConfig={...}
json={
  "PhysicalAsset":{
    "ModelName":{
      "class":"ClassName",
      "config":modelConfig
    }
  }
}
```

として、C++、Python それぞれ以下のように登録する。

(1)C++からの static メンバへの登録

```
Factory::addModel("GroupName","ModelName",modelConfig)
Factory::addDefaultModelsFromJson(json)
Factory::addDefaultModelsFromJsonFile(filepath)
```

のいずれかの方法で登録する。

(2)Python からの static メンバへの登録

```
Factory.addModel("GroupName","ModelName",modelConfig)
Factory.addDefaultModelsFromJson(json)
Factory.addDefaultModelsFromJsonFile(filepath)
```

のいずれかの方法で登録する。

(3)SimulationManager インスタンスの固有モデルとして登録する場合
C++、Python 問わず、

```
config={
  "Manager":{...},
  "Factory":{
    "PhysicalAsset":{
      "ModelName":{
        "class":"ClassName",
        "config":modelConfig
      }
    }
  }
}
```

のように、SimulationManager に与える config に"Factory"キーでモデルの情報を記述することで登録可能である。

### 4.1.3 モデルの処理周期の指定

各 Asset モデルは、perceive,control,behave の処理周期を modelConfig で指定することが可能であり、その記述方法は表 4.１－１のとおりである。また、各 Callback モデルは、onInnerStepBegin/End の処理周期を instanceConfig 又は modelConfig で指定することが可能であり、その記述方法は表 4.１－２のとおりである。

表 4.１－１　Asset の modelConfig における処理周期の記述方法

| キー名(インデントは階層を表す) | 型 | 概要 | 省略時 |
|---|---|---|---|
| firstTick | object | 初回処理タイミングに関する指定 | |
| unit | string | 指定方法。以下のいずれか。<br>"tick"･･･シミュレーション tick で指定。<br>"time"･･･シミュレーション時刻で指定。最も近い tick に変換される。 | "tick" |
| perceive | double/int | perceive の初回処理タイミング | 1[tick]相当 |
| control | double/int | control の初回処理タイミング | 0 |
| behave | double/int | behave の初回処理タイミング | 0 |
| interval | object | 処理間隔に関する指定 | |
| unit | string | 指定方法。以下のいずれか。<br>"tick"･･･シミュレーション tick で指定。<br>"time"･･･シミュレーション時刻で指定。最も近い tick に変換される。 | "tick" |
| perceive | double/int | perceive の処理間隔 | 1[tick]相当 |
| control | double/int | control の処理間隔 | 1[tick]相当 |
| behave | double/int | behave の処理間隔 | 1[tick]相当 |

表 4.1－2　Callback の instanceConfig/modelConfig における処理周期の記述方法

| キー名(インデントは階層を表す) | 型 | 概要 | 省略時 |
|---|---|---|---|
| firstTick | object | 初回処理タイミングに関する指定 | |
| unit | string | 指定方法。以下のいずれか。<br>"tick"･･･シミュレーション tick で指定。<br>"time"･･･シミュレーション時刻で指定。最も近い tick に変換される。 | "tick" |
| value | double/int | onInnerStepBegin の初回処理タイミング | 0 |
| interval | object | 処理間隔に関する指定 | |
| unit | string | 指定方法。以下のいずれか。<br>"tick"･･･シミュレーション tick で指定。<br>"time"･･･シミュレーション時刻で指定。最も近い tick に変換される。 | "tick" |
| value | double/int | onInnerStepBegin の処理間隔 | 1[tick]相当 |

### 4.1.4 モデルの置き換え

現在の Factory インスタンスが保持しているモデル情報を一度 json に変換し、それに対して与えられた置換用 json による merge_patch を適用する。マージ後の json から再度モデル情報を読み込み直すことでモデルの置き換えが完了する。

## 4.2 MotionState

本シミュレータにおいて一般的な運動状態は MotionState クラスにより表現するものとし、位置、速度、姿勢、角速度及び生成時刻を保持するものとして扱う。また、姿勢に関する追加情報として、方位角、ピッチ角の情報と、方位角をそのままに x-y 平面を水平面と一致させた座標系(局所水平座標系)の情報を付加するものとする。また、基本的な observables として得られるものは慣性系における値とし、内部の状態量として使用するものは親 Asset の座標系における値とする。運動状態を json 化した際の表現は表 4.2－1 の通りとする。なお、MotionState クラスは純粋データ型として扱われ、shared_ptr としてでなく値として生成される。

表 4.2－1　MotionState クラスの json 表現

| キー名 | 本文中の記号 | 型 | 概要 |
|---|---|---|---|
| pos | | array(double) | 位置ベクトル |
| vel | | array(double) | 速度ベクトル |
| omega | | array(double) | 角速度ベクトル |
| q | | array(double) | 現在の姿勢。クォータニオンを実部⇒虚部の順に並べた 4 次元ベクトルとして記述。 |
| qh | | array(double) | 現在の局所水平座標系を表すクォータニオンを実部⇒虚部の順に並べた 4 次元ベクトルとして記述。 |
| az | | double | 現在の方位角(真北を 0 として東側を正) |
| el | | double | 現在のピッチ角(下向きを正) |
| time | | double | この MotionState を生成した時刻 |

### 4.2.1 時刻の外挿

MotionState は生成時刻の情報を保持しており、指定した時間 dt だけ外挿する extrapolate(dt)と、指定した時刻 dstTime まで外挿する extrapolateTo(dstTime)の 2 種類によって状態量を外挿することができる。

### 4.2.2 座標変換

MotionState は自身の座標系を B、親の座標系を P、局所水平座標系を H として、相対位置、絶対位置、速度、角速度ベクトルの座標変換を行う関数を持つ。その一覧は表 4.２－２の通り。

表 4.２－２　MotionState クラスの座標変換関数の一覧

| 関数名 | 引数 | 概要 |
|---|---|---|
| relBtoP<br>relPtoB<br>relHtoP<br>relPtoH | const Eigen::Vector3d &v | 相対位置ベクトル v の変換<br>原点の平行移動を無視する |
| absBtoP<br>absPtoB<br>absBtoH<br>absHtoB | const Eigen::Vector3d &v | 絶対位置ベクトル v の変換<br>原点の平行移動を考慮する |
| velBtoP<br>velPtoB | const Eigen::Vector3d &v<br>const Eigen::Vector3d &r | 元座標系で位置 r にある点の速度 v の変換<br>原点まわりの回転を考慮する<br>r を省略した場合は r=0 とする |
| omegaBtoP<br>omegaPtoB | const Eigen::Vector3d &v | 角速度ベクトルの変換 |

## 4.3 Track3D 及び Track2D

本シミュレータは、各種センサによって捉えた航跡を表す基底クラスとして、3 次元航跡を表す Track3D クラスと 2 次元航跡を表す Track2D クラスを実装している。これらを継承し、又は全く異なる航跡クラスを使用することも許容される。

### 4.3.1 3 次元航跡の表現

3 次元航跡を表す Track3D クラスは、慣性系での位置、速度及び生成時刻を保持するものとして扱う。また、航跡は必ずそれがどの Asset を指したものかを示す情報を付加しており、誤相関が発生しない理想的なものとして扱うこともできるようになっている。3 次元航跡を json 化した際の表現は表 4.３－１の通りとする。

表 4.３－１　Track3D クラスの json 表現

| キー名 | 本文中の記号 | 型 | 概要 |
|---|---|---|---|
| truth | | str | この 3 次元航跡が指す対象の Asset を特定する UUID(バージョン 4)を表す文字列。 |
| time | | array(double) | この航跡を生成した時刻 |
| pos | | array(double) | 位置ベクトル(慣性系) |
| vel | | array(double) | 速度ベクトル(慣性系) |
| buffer | | array(object) | この 3 次元航跡と同一の対象を指すものとして外部から追加された 3 次元航跡のリスト。merge 関数によって平均値をとる際に用いられる。 |

### 4.3.2 2 次元航跡の表現

2 次元航跡を表す Track2D クラスは、慣性系での観測点の位置、観測点から見た目標の方向及び角速度並びに生成時刻を保持するものとして扱う。また、Track3D と同様に、どの Asset を指したものかを示す情報を付加している。次元航跡を json 化した際の表現は表 4.３－２の通りとする。

表 4.3－2　Track2D クラスの json 表現

| キー名 | 本文中の記号 | 型 | 概要 |
|---|---|---|---|
| truth | | str | この 2 次元航跡が指す対象の Asset を特定する UUID(バージョン 4)を表す文字列。 |
| time | | array(double) | この航跡を生成した時刻 |
| dir | | array(double) | 方向ベクトル(慣性系) |
| origin | | array(double) | 観測者の位置ベクトル(慣性系) |
| omega | | array(double) | 角速度ベクトル(慣性系) |
| buffer | | array(object) | この 2 次元航跡と同一の対象を指すものとして外部から追加された 2 次元航跡のリスト。merge 関数によって平均値をとる際に用いられる。 |

### 4.3.3 時刻の外挿

Track3D と Track2D は MotionState と同様に生成時刻の情報を保持しており、指定した時間 dt だけ外挿する extrapolate 関数と、指定した時刻 dstTime まで外挿する extrapolateTo 関数の 2 種類によって状態量を外挿することができる。

### 4.3.4 航跡のマージ

Track3D と Track2D はそれぞれ、複数の同種の航跡の平均を取ってマージすることができる。元の航跡に addBuffer 関数によってマージ対象の航跡を追加していき、最後に merge 関数を呼ぶことでそれらの平均をとったものとして更新される。

### 4.3.5 同一性の判定

Track3D と Track2D は、簡略化のために誤相関は発生しないものとして扱うことを可能としており、それらがどの Asset を指した航跡なのかという情報を真値で保持している。また、ある航跡に対して他の航跡または Asset を与えられたとき、同一の Asset を指しているか否か、あるいは有効な航跡か否かを正しく判定することができるものとしている。同一性の判定は isSame 関数で、有効性の判定は is_none 関数で行う。

## 4.4 Asset

### 4.4.1 Asset 名の規則

シミュレーションに登場する Asset は重複のない名称で生成・管理されるが、その名称(fullName)は"/"区切りの多階層で表現される。最上層の名を陣営名(team)、最下層の名称をインスタンス名(name)、インスタンス名の直前までをグループ名(group)として扱う。個別の Asset を特定する際は fullName を使用し、複数の Asset を指定する際は team または group を使用することを想定している。

### 4.4.2 子 Asset の生成

Asset の中には、子 Asset を持つものが存在する。子 Asset の生成処理は makeChildren 関数に記述する。makeChildren は SimulationManager の管理下で自身が生成された場合、その直後に呼ばれる。

子 Asset 生成の際はメンバ変数 manager を経由して SimulationManager の generateAsset 関数または generateAssetByClassName 関数を呼び出すことで行う。モデル名を指定して生成する場合は前者を、クラス名を指定して生成する場合は後者を呼び出せばよい。

### 4.4.3 他 Asset に依存する初期化処理

Asset の初期化処理の中には、他の Asset の生成が完了してからでないと行えないものが存在し得る。このような処理はインスタンス生成と分離するために、validate 関数に記述するものとする。validate 関数は reset 関数内で全インスタンスの生成後に自動的に呼ばれる。

### 4.4.4 処理順序の解決

同時刻に複数の Asset が処理を行うこととなった場合の優先順位は、メンバ変数 dependencyChecker が持つ 2 種類の addDependency 関数によって指定する。

(1) Asset オブジェクトで指定する場合

```
void addDependency(SimPhase phase, std::shared_ptr<Asset> asset)
  @param[in] phase 指定対象とする処理。SimPhase は enum class であり、
                   VALIDATE, PERCEIVE, CONTROL, BEHAVE のいずれかを指定する。
  @param[in] asset 先に処理されるべき Asset。
```

(2) Asset の fullName で指定する場合

```
void addDependency(SimPhase phase, const std::string& fullname)
  @param[in] phase 指定対象とする処理。SimPhase は enum class であり、
                   VALIDATE, PERCEIVE, CONTROL, BEHAVE のいずれかを指定する。
  @param[in] fullName 先に処理されるべき Asset の fullName。
```

なお、Agent は他の Asset の dependencyChecker にアクセスできないため、親となる PhysicalAsset 側で優先順位を指定するものとする。Agent 以外の Asset は前後どちらの Asset から指定しても差し支えない。

### 4.4.5 config の記述方法

PhysicalAsset、Controller、Agent クラスには modelConfig で指定すべき必須項目は存在しない。instanceConfig については、PhysicalAsset と Controller の場合、表 4.4－1 に示す要素が必要である。Agent の instanceConfig については原則として自動的に生成されるため省略する。

なお、上記の instanceConfig を直接指定するのは makeChildren 関数内で子 Asset として生成する場合のみである。それ以外の Asset(＝親 Asset の無い独立した PhysicalAsset)については SimulationManager の config に記述して 4.6.3 項に示す要領で生成されるため、その記述方法は表 4.4－2 の通りである。

表 4.4－1　PhysicalAsset と Controller の instanceConfig の必須構成要素(json)

| キー名 | 対応する変数名 | 型 | 概要 | 省略可否 |
|---|---|---|---|---|
| seed | | | Asset 固有の乱数生成器のシード。省略した場合は std::random_device()()によって初期化される。 | ○ |
| manager | | intptr_t | SimulationManager への Accessor へのポインタであり、SimulationManager によって自動的に設定される。 | ○ |
| baseTimeStep | | double | 1[tick]の時間[s]を表し、SimulationManager によって自動的に設定される。 | ○ |
| dependencyChecker | | intptr_t | 処理順序指定用の変数へのポインタであり、SimulationManager によって自動的に設定される。 | ○ |
| fullName | fullName<br>team<br>group<br>name | string | Asset の名称。 | × |
| parent | parent | intptr_t | 親となる Asset へのポインタであり、省略した場合は nullptr となる。 | ○ |
| isBound | isBound | bool | parent に固定されているか否か。PhysicalAsset のみ使用する。省略した場合は false となる。 | ○ |

表 4.４－２　SimulationManager の config における PhysicalAsset の記述方法

| キー | 値 | 省略可否 |
|---|---|---|
| type | ″PhysicalAsset″ | × |
| model | モデル名(string) | × |
| Agent | 自身に対応する Agent を指定する object。<br><br>表 4.５－２に示す object または同等の object を表 4.６－５に示す ConfigDispatcher に与えることが可能なものとする。<br>省略した場合は Agent 無しとなる。 | ○ |
| instanceConfig | 表 4.４－１に示す項目以外の instanceConfig。<br>例えば Fighter の場合、以下の 3 項目を記述する。<br>pos･･･初期位置(3-dim array(double))<br>vel･･･初期速度(double)<br>heading･･･真北を 0、時計回りを正とした初期方位(double)<br>省略した際の振る舞いは各クラスの実装による。 | ○ |

## 4.5 Agent

本項では、Agent クラスの使用方法について概要をまとめる。

### 4.5.1 Agent の種類について

本シミュレータでは、Agent の種類を、表 4.５－１の通りに分類する。また、SimulationManager の config で生成する Agent について記述する際にはこの分類に従い、

表 4.５－２に示す要素を持つ object として表 4.４－２に示す通り各 PhysicalAsset の config 中に記述する。

表 4.５－１　Agent の種類

| 名称 | 概要 |
|---|---|
| Internal | Observation と Action が不要な、Agent クラス単体で行動判断が完結する Agent を指し、ポリシー名が自動的に"Internal"となる。step 関数に与える Action には、このような Agent に対応する Action を含めなくてもよい(省略しても任意の値を与えてもよい)。 |
| External | Observation と Action の入出力が必要な Agent クラス。 |
| ExpertE | 3.６項で述べた ExpertWrapper のうち、親 Asset への command と環境外部への observation として expert の出力を用いるもの。 |
| ExpertI | 3.６項で述べた ExpertWrapper のうち、親 Asset への command と環境外部への observation として imitator の出力を用いるもの。 |
| ExpertBE | 3.６項で述べた ExpertWrapper のうち、親 Asset への command として expert の出力を用い、環境外部への observation として両方の出力を用いるもの。 |
| ExpertBI | 3.６項で述べた ExpertWrapper のうち、親 Asset への command として imitator の出力を用い、環境外部への observation として両方の出力を用いるもの。 |

表 4.5－2　SimulationManager の config における Agent の記述方法

| Agent の種類 | キー | 値 | 省略可否 |
|---|---|---|---|
| Internal | model | モデル名(string) | × |
| External | model | モデル名(string) | × |
| | policy | ポリシー名(string) | × |
| ExpertE<br>ExpertI<br>ExpertBE<br>ExpertBI | type | Agent の種類と同一の文字列(string) | × |
| | imitatorModel | 模倣する側のモデル名(string) | × |
| | expertModel | 模倣される側のモデル名(string) | × |
| | expertPolicy | 模倣される側のポリシー名(string)。省略時は"Internal"となる。 | ○ |
| | identifier | 模倣用の軌跡データを出力する際の識別名。省略した場合は imitatorModelName と同一となる。 | ○ |
| 共通 | type | Agent の種類と同一の文字列(string) | × |
| | name | インスタンス名。 | × |
| | port | port 名。省略した場合は"0"とみなされる。 | ○ |

### 4.5.2 observables、commands の記述方法

Agent が保持する observables と commands は、親 Asset の fullName をキーとして各親ごとに値を持つ object とする。これにより、親 Asset は Agent の observations と commands から自身に関連する情報を抽出することができるようになる。

なお、commands だけでなく observables も使用する目的は、親 Asset を介して味方 Agent との情報共有を行えるようにするためであり、例えば空対空目視外戦闘の初期行動判断モデルは戦闘機・無人機モデルを介して味方 Agent の observables を参照するような設計としている。

### 4.5.3 Agent クラス間に共通の行動意図表現について

Action の値が何を意味しているかは Agent クラスによって異なること、commands は Asset の制御用に各 tick の出力値として加工された値に過ぎないことから、いずれも他の Agent クラスにとっては行動の意図を表現している情報ではない。そのため、他の Agent クラスを模倣した学習を実現するためには、両クラスの間で共通の行動意図表現が必要となる。本シミュレータではその行動意図表現を decision として、observables の一要素として記述することとしており、3.６項で述べた ExpertWrapper クラスによる模倣学習を行いやすくするための機能で使用している。

本シミュレータの初期行動判断モデル及びサンプル Agent で用いている、空対空目視外戦闘用の行動意図表現は表 4.５－３の通りである。ただし、模倣する側とされる側の間で情報を伝達するための単なるパイプとして捉え、必要に応じて任意の変数を追加で共有してしまっても差し支えない。

表 4.５－３　空対空目視外戦闘用の基本となる decision の構成要素

| キー名(インデントは階層を表す) | 値 | 概要 |
|---|---|---|
| Roll | ["Don't care"] | ロール方向の回転について指定なしであることを表す。 |
| | ["Angle",<value>] | ロール方向の回転について目標ロール角<value>への回転を意図していることを表す。 |
| | ["Rate",<value>] | ロール方向の回転について指定角速度<value>での回転を意図していることを表す。 |
| Horizontal | ["Don't care"] | 水平方向の旋回について指定なしであることを表す。 |
| | ["Az_NED",<value>] | 水平方向の旋回について NED 座標系(慣性系)での指定方位<value>への旋回を意図していることを表す。 |
| | ["Az_BODY",<value>] | 水平方向の旋回について機体座標系での指定方位<value>への旋回を意図していることを表す。 |
| | ["Rate",<value>] | 水平方向の旋回について指定角速度<value>での旋回を意図していることを表す。 |
| Vertical | ["Don't care"] | 垂直方向の上昇・下降について指定なしであることを表す。 |
| | ["El",<value>] | 垂直方向の上昇・下降について指定経路角(下向き正)<value>での上昇・下降を意図していることを表す。 |
| | ["Pos",<value>] | 垂直方向の上昇・下降について目標高度<value>への上昇・下降を意図していることを表す。 |
| | ["Rate",<value>] | 垂直方向の上昇・下降について指定角速度<value>での上昇・下降を意図していることを表す。 |
| Throttle | ["Don't care"] | 加減速について指定なしであることを表す。 |
| | ["Vel",<value>] | 加減速について目標速度<value>への加減速を意図していることを表す。 |
| | ["Accel",<value>] | 加減速について指定加速度<value>での加減速を意図していることを表す。 |
| | ["Throttle",<value>] | 加減速について指定スロットルコマンド(0～1)<value>での加減速を意図していることを表す。 |
| Fire | [<launchFlag>,<target>] | <launchFlag>は bool 型で、True のときに<target>に射撃する意図を持っていることを表す。<target>は Track3D の json 表現とする。 |

### 4.5.4 親 Asset へのアクセス方法

Agent が親 Asset にアクセスするためにはメンバ変数 parents を用いる。これは fullName をキーとし、std::shared_ptr<AssetAccessor>を値とした Map となっている。この Accessor が提供する機能は表 4.5－4の通りである。なお、Fighter クラスの Accessor で提供される機能を C++側で用いる際には dynamic_pointer_cast で FighterAccessor クラスにキャストする必要がある。

表 4.5－4　親 Asset への Accessor が提供する機能

| 実装クラス | 変数/関数名 | 変数の型/戻り値型 | 概要 |
|---|---|---|---|
| Asset | getFactoryBaseName() | std::string | 親 Asset が Factory により生成された際のグループ名 |
| | getFactoryClassName() | std::string | 親 Asset が Factory により生成された際のクラス名 |
| | getFactoryModelName() | std::string | 親 Asset が Factory により生成された際のモデル名 |
| | isAlive() | bool | 親 Asset の生存状態を返す。 |
| | getTeam() | std::string | 親 Asset の陣営名を返す。 |
| | getGroup() | std::string | 親 Asset のグループ名を返す。 |
| | getName() | std::string | 親 Asset のインスタンス名を返す。 |
| | getFullName() | std::string | 親 Asset の fullName を返す。 |
| | observables | const nl::json& | 親 Asset の observables の const 参照 |
| | <template class T> isinstance<T>() | bool | 親 Asset がクラス T のインスタンスか否かを返す。 |
| | isinstancePY(cls) (C++から) or isinstance(cls) (Python から) | bool | 親 Asset が Python クラスオブジェクト cls のインスタンスか否かを返す。Python 側からの使用を想定しているものである。 |
| Fighter | setFlightControllerMode(name) | void | 飛行制御の方法をその名前で指定する。 |
| | getRmax(rs,vs,rt,vt) | double | 搭載誘導弾の推定射程を返す。計算条件は、位置 rs、速度 vs で飛行中の機体から位置 rt、速度 vt で飛行中の機体に射撃し、射撃された側が等速直線運動を継続した場合の射程である。 |
| | getRmax(rs,vs,rt,vt,aa) | double | 搭載誘導弾の推定射程を返す。計算条件は、位置 rs、速度 vs で飛行中の機体から位置 rt、速度 vt で飛行中の機体に射撃し、射撃された側が直ちにアスペクト角が aa となる方位に等速直線運動を継続した場合の射程である。 |

### 4.5.5 模倣学習のための関数オーバーライド

3.6項で述べた機構を用いて模倣学習をする際は、SimulationManager の config において、

表 4.5－2に示した"ExpertE"、"ExpertI"、"ExpertBE"、"ExpertBI"のいずれかを指定してExpertWrapperクラスのインスタンスとして生成することとなるが、imitator(模倣する側)のクラスで二つのメンバ関数をオーバーライドしておく必要がある。

一つ目はconvertActionFromAnother関数であり、expert(模倣される側)により生成されたdecisionとcommandsを自身のaction_spaceで最も近いActionに変換するものである。

```
py::object Agent::convertActionFromAnother(const nl::json& decision,
                                           const nl::json& command)
  @param[in] decision expertのobservation
  @param[in] command expertからparentへの制御出力
  @return py::object 計算されたimitatorのAction
```

二つ目はcontrolWithAnotherAgent関数であり、control関数の処理内容expert(模倣される側)により生成されたdecisionとcommandsを参照してその内容に応じて変更したい場合にオーバーライドする必要がある。オーバーライドしない場合は、単に通常のcontrol関数がそのまま呼ばれることとなる。

```
void Agent::controlWithAnotherAgent(const nl::json& decision,
                                    const nl::json& command)
  @param[in] decision expertのobservation
  @param[in] command expertからparentへの制御出力
```

### 4.5.6 MultiPortCombiner

2.2.1.3項で触れた通り、一つのAgentが複数のPhysicalAssetを親として持つことが認められている。そのため、単一の親を持つ複数のAgentの組合せを模倣対象として、複数の親を持つ単一のAgentを学習したい場合が生じ得る。これに対応するために、複数のAgentを束ねて一つのAgentのように扱うためのMultiPortCombinerクラスをAgentクラスの派生クラスとして実装している。

MultiPortCombinerを使用する際には、modelConfigにおいて自身の各portに対応する子Agentのインスタンス名、モデル名及びport名を与えることで必要な子Agentが生成される。ただし、MultiPortCombinerクラスをそのまま使用できるのはObservationとActionが不要な"Internal"なAgentモデルを子とする場合のみであり、そうでない場合は各ObservationとActionの具体的な束ね方についてユーザーが定義しmakeObs関数とactionSplitter関数をオーバーライドした派生クラスを実装して使用する必要がある。最もシンプルな派生クラスとして、単に全ての子のObservationとActionをdictとして束ねるSimpleMultiPortCombinerクラスを実装済である。

### 4.5.7 SingleAssetAgent

SingleAssetAgentは単一のAssetを親として持つAgentである。これは、都度port名を指定して親Assetにアクセスするのは手間がかかるため、その単一の親に別のメンバ変数parentとしてアクセス可能にしたものである。configの記述方法は通常のAgentと同じである。

## 4.6 SimulationManager

本項では、SimulationManagerクラスの使用方法についてまとめる。

### 4.6.1 configの記述方法

SimulationManagerのconfigは表 4.6－1～表 4.6－4の通り記述する。

表 4.6－1　SimulationManagerのconfigの記述方法

| キー名(インデントは階層を表す) | 型 | 概要 | 省略可否 |
|---|---|---|---|
| TimeStep | | | |
| baseTimeStep | double | シミュレーション時刻の最小単位(=1tick) | × |
| agentInterval | unsigned int | エージェントの行動判断の周期 | × |
| measureTime | | 処理に要した時間を計測・出力するかどうか。省略時はfalseとなる。 | ○ |
| numThread | unsigned int | 使用するスレッド数。省略時は1となる。 | ○ |

| | | | |
|---|---|---|---|
| exposeDeadAgentReward | bool | 生存していない Agent の報酬も出力し続けるかどうか。省略時は true となる。 | ○ |
| ViewerType | string | 使用する Viewer モデルの名称。省略した場合は"None"(非表示)となる。 | ○ |
| Ruler | string | 使用する Ruler モデルの名称 | × |
| Rewards | array(object) | 使用する Reward モデルのリスト。各要素の記述方法は表 4.6－2の通り。省略時は model="ScoreReward"、"target"="All"とした要素一つとみなし、各陣営の得点をそのまま報酬とするようになる。 | ○ |
| Callbacks | array(object) | 使用する Callback モデルの一覧。各要素の記述方法は表 4.6－3の通り。インスタンス名をキー、クラス/モデル名と config の組を値とした object として与える。 | ○ |
| Loggers | array(object) | 使用する Logger モデルの一覧。与え方は Callbacks と同じ。 | ○ |
| CommunicationBuffers | array(object) | 生成する CommunicationBuffer のリスト。各要素の記述方法は表 4.6－4の通り。 | ○ |
| Assets | object | 独立した PhysicalAsset として生成する全ての PhysicalAsset の config を陣営名からインスタンス名までの多階層で記述した object<br>または同等の object に ConfigDispacher によって変換可能な object。 | × |
| AssetConfigDispatcher | object | PhysicalAsset を生成するための ConfigDispatcher の初期化用 alias を列挙した object。 | × |
| AgentConfigDispatcher | object | Agent を生成するための ConfigDispatcher の初期化用 alias を列挙した object。 | × |

表 4.6－2　SimulationManager における Reward モデルの記述方法

| キー名 | 型 | 概要 | 省略可否 |
|---|---|---|---|
| model | string | モデル名 | × |
| target | string 又は array(string) | 計算対象の陣営又は Agent の名称を指定する文字列又はそのリスト。<br>指定方法は以下のいずれか。<br>"Team:REGEX_PATTERN"：REGEX_PATTERN に正規表現としてマッチする陣営<br>"Agent:REGEX_PATTERN"：REGEX_PATTERN に正規表現としてマッチする Agent<br>"All"：場に存在する者すべての陣営及び Agent | × |

表 4.6－3　SimulationManager における Callback/Logger モデルの指定方法

| キー名 | 型 | 概要 | 省略可否 |
|---|---|---|---|
| class | string | クラス名で指定する場合のクラス名。 | △ |
| model | string | モデル名で指定する場合のモデル名 | △ |
| config | object | クラス名で指定する場合は modelConfig に相当する config を与え、モデル名で指定する場合は instanceConfig に相当する config のうちユーザー定義の要素を与える。 | × |

表 4.6－4　SimulationManager における CommunicationBuffer モデルの記述方法

| キー名 | 型 | 概要 | 省略可否 |
|---|---|---|---|
| participants | string または array(string) | 無条件で参加する Asset の名称またはそのリスト。<br>各要素は 3.5項に示した通り、Asset 種別と Asset 名(fullName)をコロンで区切った文字列とし、Asset 名は正規表現として解釈される。 | ○ |
| inviteOnRequest | string または array(string) | Asset 側から要求があった場合のみ参加させる Asset の名称またはそのリスト。<br>各要素は 3.5項に示した通り、Asset 種別と Asset 名(fullName)をコロンで区切った文字列とし、Asset 名は正規表現として解釈される。 | ○ |

### 4.6.2 ConfigDispatcher による json object の再帰的な変換

本シミュレータは、config として与えた json から確率による選択や並び替え、要素の複製や別名参照による置換等を経て、複雑な登場物の生成を記述しやすくするための ConfigDispatcher クラスを実装しており、これを用いて PhysicalAsset と Agent の動的生成を実現している。ConfigDispatcher クラスそのものの機能は json を与えると一定の規則に従って再帰的に変換を行い一つの json を生成するものである。

ConfigDispatcher の変換対象物及び生成物となる json の記述方法は表 4.6－5の通りであり、初期化の際は alias として登録したい変換対象物又は生成物を列挙した object を与える。

変換の実行は ConfigDisptcher のメンバ変数 run に生成元 json を引数として与えることで行う。

表 4.6－5　ConfigDispatcher の変換対象物及び生成物の記述方法

| キー"type"の値 | 他のパラメータ | 生成されるもの |
|---|---|---|
| "alias" | "alias": string | 自身の属する ConfigDispatcher から<alias>で与えられた名称を持つ object を参照して複製を生成する。 |
| "broadcaster" | "number": unsinged int<br>"names": array(string)<br>"element": object<br>"dispatchAfterBroadcast": bool (default=false) | <element>で与えられた要素を<number>個複製し、<names>で与えられたキーに対応付けて<type>=="group"相当の object として生成する。<br><names>を省略した場合は 1 番から順に"Element1", "Element2"・・・のように命名される。<br><dispatchAfterBroadcast>が true のときは複製してから各要素を dispatch し、false のときは dispatch 後のものを複製する。 |
| "choice" | "weights": array(double)<br>"candidates": array(object) | <weights>で与えられた重みに従い、<candidates>で与えられた要素から一つ選択して生成する。 |
| "concatenate" | "elements": array(object) | <elements>で与えられた要素を結合して一つの<type>=="group"相当の object を生成する。 |
| "direct" | "value": object | <value>で与えられた値そのものとして生成する。 |
| "group" | "order": "fixed" or "shuffled"<br>"names": array(string)<br>"elements": array(object) | <elements>で与えられた要素を、<order>が"fixed"のときはそのまま、"shuffled"のときはランダムに並べ替えて、<names>で与えられたキーに対応付けた object として生成する。<br><names>を省略した場合は 1 番から順に"Element1", "Element2"・・・のように命名される。 |
| "none" | なし | 空っぽの要素を返す。最終生成物からは削除されるが、例えば choice において「選ばない」ということを表現するために用いる。 |
| 上記以外の任意の文字列 | | <type>=="direct"とみなし、その値自身が<value>に与えられていたものとして扱う。 |
| "type"キーが無い場合、または object 型でない場合 | | <type>=="direct"とみなし、その値自身が<value>に与えられていたものとして扱う。 |
| 共通 | "instance": str または指定なし<br>"index": int または指定なし<br>"overrider": object, array または指定なし | <instance>が与えられた場合、その名称において一つのインスタンスを保持し、他の場所で同名の<instance>が指定された際には同一のオブジェクトとして得られる。<br><index>が与えられた場合、Broadcaster と Group の複数の要素のうち、<index>番目の要素を抽出して返す。<br><overrider>が与えられた場合、本体側の生成を終えたあと、<overrider>に対しても同様に dispatch を行い、得られた object で本体の object を merge_patch する。<br>本体の生成物が<type>=="group"相当の object であるとき、<overrider>の生成物も要素数が同じ<type>=="group"相当の object でなければならない。 |

### 4.6.3 PhysicalAsset の生成

PhysicalAsset の生成は、SimulationManager の config における"AssetConfigDispatcher"キーに対応する値で初期化した ConfigDispatcher の run 関数に、"Assets"キーに対応する値を引数として与えて得られた json object を用いて行う。

得られる json object は多階層の object であり、最上位層のキーは陣営名を表し、表 4.４－２の書式に合致する値を持つ層のキーはインスタンス名を表しているものとする。

### 4.6.4 Agent の生成

Agent の生成は、次のように行われる。

(1)SimulationManager の config における"AgentConfigDispatcher"キーに対応する値で初期化した ConfigDispatcher の run 関数に、PhysicalAsset の生成時にキー"Agent"で指定されていた json object を引数として与える。

(2)得られた json object は

表 4.5－2の書式に合致する値を持つが、単一の Agent が複数の PhysicalAsset から指定されている場合があるため、同一 Agent を指しているか否かを判定して生成が必要な Agent インスタンス数の特定を行うと同時に、port と parents の対応付けを行う。
(3)対応付けが終わったら Agent インスタンスを生成し、各 PhysicalAsset に対して対応付けられた Agent インスタンスを通知する。

### 4.6.5 各登場物から SimulationManager へのアクセス方法

各登場物が SimulationManager にアクセスするためには、メンバ変数 manager を用いる。これはその登場物の種類に応じて異なる Accessor クラスの shared_ptr であり、提供されている機能はそれぞれの通りである。

表 4.6－6　SimulationManager への Accessor が提供する機能(1／3)

| 対象クラス | | | 関数名 | 概要 |
|---|---|---|---|---|
| Agent | PhysicalAsset /Controller | Callback | | |
| ○ | ○ | ○ | getTime() | 現在のシミュレーション時刻[s]を返す。 |
| ○ | ○ | ○ | getBaseTimeStep() | 現在のエピソードにおける 1[tick]の時間[s]を返す。 |
| ○ | ○ | ○ | getTickCount() | 現在の時刻を[tick]単位で返す。 |
| ○ | ○ | ○ | getAgentInterval() | 現在のエピソードにおける Agent の行動判断周期[tick]を返す。 |
| ○ | ○ | ○ | getStepCount() | 現在の step 数を返す。 |
| ○ | ○ | ○ | getTeams() | 現在のエピソードにおいて存在する陣営名のリストを返す。 |
| ○ | ○ | ○ | getRuler() | 現在のエピソードで使用されている Ruler を返す。<br>Agent に対しては observables のみ参照可能な Accessor を返す。 |
| | | ○ | getViewer() | 現在のエピソードで使用されている Viewer を返す。 |
| | | ○ | getRewardGenerator(idx) | 現在のエピソードで使用されている Reward のうちインデックスが idx であるものを返す。 |
| | | ○ | getRewardGenerators() | 現在のエピソードで使用されている全 Reward のイテラブルを返す。 |
| | ○ | ○ | getAsset(fullName) | 名称が fullName である PhysicalAsset を返す。 |
| | ○ | ○ | getAssets() | 現在のエピソードに存在する全 Asset のイテラブルを返す。 |
| | ○ | ○ | getAssets(matcher) | 現在のエピソードに存在する全 Asset のうち matcher に該当するものを対象としたイテラブルを返す。matcher は Asset を引数にとり bool を返す関数オブジェクトである。 |
| | ○ | ○ | getAgent(fullName) | 名称が fullName である Agent を返す。 |
| | ○ | ○ | getAgents() | 現在のエピソードに存在する全 Agent のイテラブルを返す。 |
| | ○ | ○ | getAgents(matcher) | 現在のエピソードに存在する全 Agent のうち matcher に該当するものを対象としたイテラブルを返す。matcher は Agent を引数にとり bool を返す関数オブジェクトである。 |
| | ○ | ○ | getController(fullName) | 名称が fullName である Controller を返す。 |
| | ○ | ○ | getControllers() | 現在のエピソードに存在する全 Controller のイテラブルを返す。 |

表 4.6－6　SimulationManager への Accessor が提供する機能(2／3)

| 対象クラス | | | 関数名 | 概要 |
|---|---|---|---|---|
| Agent | PhysicalAsset /Controller | Callback | | |
| | ○ | ○ | getControllers(matcher) | 現在のエピソードに存在する全 Controller のうち matcher に該当するものを対象としたイテラブルを返す。matcher は Controller を引数にとり bool を返す関数オブジェクトである。 |
| | ○ | ○ | getCallback(name) | 名称が name である Callback を返す。 |
| | ○ | ○ | getCallbacks() | 現在のエピソードに存在する全 Callback のイテラブルを返す。 |

| | | | | |
|---|---|---|---|---|
| | ○ | ○ | getCallbacks(matcher) | 現在のエピソードに存在する全 Callback のうち matcher に該当するものを対象としたイテラブルを返す。matcher は Callback を引数にとり bool を返す関数オブジェクトである。 |
| | ○ | ○ | getLogger(name) | 名称が name である Logger を返す。 |
| | ○ | ○ | getLoggerks() | 現在のエピソードに存在する全 Logger のイテラブルを返す。 |
| | ○ | ○ | getLoggers(matcher) | 現在のエピソードに存在する全 Logger のうち matcher に該当するものを対象としたイテラブルを返す。matcher は Logger を引数にとり bool を返す関数オブジェクトである。 |
| | | ○ | getManagerConfig() | 現在の SimulationManager 本体の config を返す。 |
| | | ○ | getFactoryConfig() | 現在の Factory に登録されたモデルの一覧を config 形式で返す。 |
| | | ○ | worker_index() | worker_index の値を返す。 |
| | | ○ | vector_index() | vector_index の値を返す。 |
| | ○ | | generateAgent(<br>agentConfig,<br>agentName,<br>parents) | Agent を SimulationManager の管理下で生成する。ユーザーが直接使用することは想定していない。 |
| | ○ | | generateAsset(<br>baseName,<br>modelName,<br>instanceConfig) | モデル名を指定して、PhysicalAsset または Controller を SimulationManager の管理下で生成する。PhysicalAsset と Controller の makeChildren で使用することを想定している。 |
| | ○ | | generateAssetByClassName(<br>baseName,<br>className,<br>modelConfig,<br>instanceConfig) | クラス名を指定して、PhysicalAsset または Controller を SimulationManager の管理下で生成する。PhysicalAsset と Controller の makeChildren で使用することを想定している。 |
| ○ | ○ | | requestInvitationToCommunicationBuffer(<br>bufferName,<br>asset) | bufferName で指定した CommunicateBuffer に対して、asset で指定した Asset の参加を要請し、その結果を bool で返す。 |
| | ○ | | generateCommunicationBuffer(<br>name<br>participants,<br>inviteOnRequest) | CommunicationBuffer を生成する。PhysicalAsset と Controller の makeChildren で使用することを想定している。 |
| ○ | ○ | ○ | generateUnmanagedChild(<br>baseName,<br>modelName,<br>instanceConfig) | モデル名を指定して、Asset や Callback を SimulationManager の管理下で生成する。ユーザーが直接使用することは想定していない。 |
| | ○ | ○ | generateUnmanagedChildByClassName(<br>baseName,<br>className,<br>modelConfig,<br>instanceConfig) | クラス名を指定して、Asset や Callback を SimulationManager の管理下で生成する。ユーザーが直接使用することは想定していない。 |
| | | ○ | addEventHandler(name,handler) | handler をイベント名 name のイベントハンドラとして追加する。 |
| | | ○ | triggerEvent(name,args) | イベント名 name を引数 args で発生させる。 |
| | | ○ | observation_space() | 全 Agent の observation_space を取得する。 |
| | | ○ | action_space() | 全 Agent の action_space を取得する。 |
| | | ○ | observation() | 全 Agent の直近の observation を取得する。 |
| | | ○ | action() | 全 Agent の直近の action を取得する。 |
| | | ○ | dones() | 全 Agent の done(終了フラグ)を取得する。 |
| | | ○ | scores() | 全陣営の得点を取得する。 |
| | | ○ | rewards() | 全 Agent の報酬を取得する。 |
| | | ○ | totalRewards() | 全 Agent の累積報酬を取得する。 |

表 4.6－6　SimulationManager への Accessor が提供する機能(3／3)

| 対象クラス | | | 関数名 | 概要 |
|---|---|---|---|---|
| Agent | PhysicalAsset /Controller | Callback | | |
| | | ○ | experts() | 現在のエピソードに存在する ExpertWrapper の一覧を Agent 名をキーとした Map で返す。 |
| | | ○ | manualDone() | 手動終了フラグの状態を返す。 |
| | | ○ | setManualDone(b) | 手動終了フラグの状態を設定する。Viewer の画面を閉じた時点で終了させる場合などに用いる。 |

| | | | | |
|---|---|---|---|---|
| | | ○ | requestReconfigure(<br>managerReplacer,<br>factoryReplacer) | 引数で与えたパッチで次のエピソード開始時に config の再設定を予約する。 |

## 4.7 空対空目視外戦闘場面のシミュレーション

本項では、前項までの機能を用いて本シミュレータ上で空対空目視外戦闘場面を表現する方法について概要をまとめる。

### 4.7.1 空対空戦闘場面を構成する Asset

本シミュレータで空対空目視外戦闘場面のシミュレーションを行う際、Agent が行動判断を行う対象は戦闘機・無人機を表す Fighter のみであり、場に独立して存在する PhysicalAsset 及び Controller は Fighter のみであるが、子 Asset として表 4.7－1 に示すクラスのインスタンスが生成される。

表 4.7－1　空対空目視外戦闘の場面に登場する Asset クラスの一覧

| 種類 | 名称(総称として記載) | 親 | 概要 |
|---|---|---|---|
| PhyisicalAsset | Fighter | なし | 戦闘機・無人機 |
| | AircraftRadar | Fighter | 戦闘機・無人機搭載センサ(レーダ) |
| | MWS | | 戦闘機・無人機搭載センサ(MWS) |
| | Propulsion | | 戦闘機・無人機用ジェットエンジン |
| | Missile | | 誘導弾 |
| | MissileSensor | Missile | 誘導弾搭載センサ(シーカ) |
| Controller | SensorDataSharer | Fighter | 戦闘機・無人機のセンサ探知データを編隊内で共有する処理の送信側 |
| | SensorDataSanitizer | | 戦闘機・無人機のセンサ探知データを編隊内で共有する処理の受信側 |
| | OtherDataSharer | | 戦闘機・無人機のセンサ探知データ以外を編隊内で共有する処理の送信側 |
| | OtherDataSanitizer | | 戦闘機・無人機のセンサ探知データ以外を編隊内で共有する処理の受信側 |
| | HumanIntervention | | 戦闘機・無人機の射撃行為に関する人間の介入を模した処理 |
| | WeaponController | | 人間の介入を受けた後の射撃判断結果に基づく誘導弾発射等の処理 |
| | FlightController | | 戦闘機・無人機の飛行制御 |
| | PropNav | Missile | 比例航法による誘導弾飛翔制御 |
| Agent | Agent | なし | 戦闘機・無人機の行動判断モデル |

### 4.7.2 Assetの処理順序

空対空目視外戦闘の場面に登場するAsset間の、perceive、control、behave、validateの各処理における処理順序の依存関係は図 4.7－1に示す通りである。線で繋がれていないものどうしはどちらを先に処理してもよいものとしている。また、図に登場していないクラスは該当する処理を行わないか、依存関係がなく任意の順序で処理可能なものである。

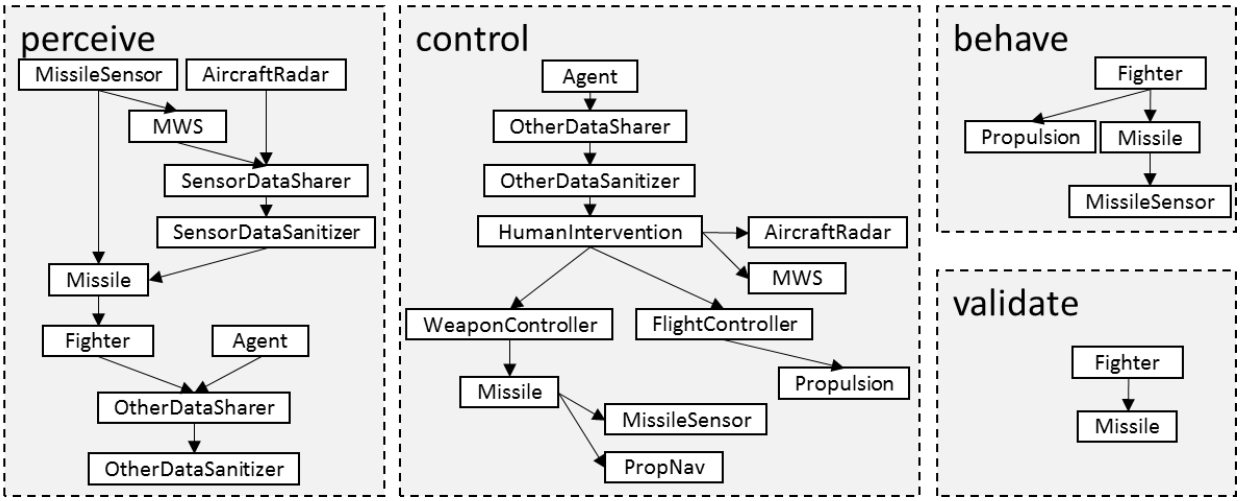


図 4.7－1　各処理におけるAssetの処理順序の依存関係

### 4.7.3 戦闘機・無人機モデルの種類

現バージョンにおいて戦闘機・無人機を表す基底クラスはFighterクラスとして実装しているが、機体の運動に関する部分のみ派生クラスで実装することとしている。現バージョンにおいてはSixDoFFighterクラス、CoordinatedFighterクラス、MassPointFighterクラスの3種類を実装している。

SixDoFFighterクラスは航空機の運動を6自由度の運動方程式で表現し、舵面の操作による姿勢制御を表現したモデルであり基本的にはこれを用いることとなる。なお、SixDoFFighterクラスは抽象クラスとなっており、空力モデルを実装した具象クラスを定義して使用する必要があるが、現バージョンにおいてはStevensFighterクラスとMorelliFighterクラスの2種類が実装されている。

CoordinatedFighterクラスは航空機の運動を横滑り角=0として、回転運動及び姿勢制御を簡略化したモデルを実装したクラスである。

MassPointFighterクラスは、CoordinatedFighterより更に簡略化し、重力や空気力を無視し速度と角速度を一定の範囲で直接操作できるような運動モデルである。

### 4.7.4 Agentが入出力すべきobservablesとcommands

Agentが戦闘機・無人機モデルから受け取ることのできるobservablesの形式は表 4.7－2のとおりである。また、Agentが戦闘機・無人機モデルに対して出力しなければならないcommandsの形式は表 4.7－3のとおりであるが、運動に関するobservables、commandsは使用するFighterクラス及び飛行制御モードによって異なる。具体的には、observables["spec"]["dynamics"]とcommands["motion"]が該当する。

表 4.7－2　戦闘機・無人機モデルの observables の形式

| キー名(インデントは階層を表す) | 型 | 概要 |
|---|---|---|
| isAlive | bool | 生存中か否か |
| spec | object | 性能に関する値（戦闘中不変） |
| dynamics | object | 運動性能に関する値。使用する Fighter クラスにより異なる。 |
| propulsion | object | 推進系(エンジン)の性能に関する値 |
| fuelCapacity | double | 燃料タンクの容量[kg] |
| optCruiseFuelFlowRatePerDistance | double | 最適な(燃料消費が最小な)巡航高度・速度で水平飛行した際の単位距離あたりの燃料消費量[kg/m] |
| weapon | object | 武装に関する値 |
| numMsls | unsigned int | 初期誘導弾数 |
| stealth | object | 被探知性能に関する値 |
| rcsScale | double | RCS スケール(無次元量として扱う) |
| sensor | object | センサに関する値 |
| radar | object | レーダに関する値 |
| Lref | double | 基準探知距離[m] |
| thetaFOR | double | 探知可能覆域[rad] |
| mws | object | MWS に関する値　※該当なし |
| motion | object | 現在の運動状態に関する値。表 4.2－1 に示す MotionState クラスの json 表現である。 |
| propulsion | object | 推進系(エンジン)の現在の状態量 |
| fuelRemaining | double | 残燃料[kg] |
| sensor | object | 現在の探知状況に関する値 |
| radar | object | 自機レーダの探知状況に関する値 |
| track | array(object) | 自機レーダが探知した 3 次元航跡のリスト。各要素は表 4.3－1 に示す Track3D クラスの json 表現である。 |
| track | array(object) | 編隊内で共有し統合されたレーダ航跡のリスト。各要素は表 4.3－1 に示す Track3D クラスの json 表現である。 |
| trackSource | array(array(string)) | 編隊内で共有し統合された 3 次元航跡それぞれの、統合元となった機体名のリストのリスト。 |
| mws | object | 自機 MWS の探知状況に関する値 |
| track | array(object) | 自機 MWS が探知した 2 次元航跡のリスト。<br>各要素は表 4.3－2 に示す Track2D クラスの json 表現である。 |
| weapon | object | 現在の武装状況に関する値 |
| remMsls | unsigned int | 現在の残弾数 |
| nextMsl | unsigned int | 次に射撃する誘導弾の ID |
| launchable | bool | 現在射撃可能な状態か否か。残弾数が 0 でなく、かつ人間介入モデルの記憶容量が上限に達していない場合に可となる。 |
| missiles | array(object) | 各誘導弾に関する observables の配列。その内訳は後述する誘導弾モデルに記載のとおり。 |
| shared | object | 味方と共有された情報 |
| agent | object | 味方と共有された、Agent に関する情報 |
| obs | object | Agent 自身が生成した固有の observables であり、各 parent の fullName をキーとした object として生成される。 |
| fighter | object | 味方機の observables。自身を含む各機の名称をキーとして各機の observables の一部が格納される。含まれないものは以下の 3 要素である。<br>・/shared 以下全て<br>・/sensor/track<br>・/sensor/trackSource |

表 4.７－３　戦闘機・無人機モデルの commands の形式

| キー名(インデントは階層を表す) | 型 | 概要 |
|---|---|---|
| motion | object | 運動に関する値。使用する Fighter クラスにより異なる。 |
| weapon | object | 射撃に関する値 |
| launch | bool | 射撃するか否か |
| target | object | 射撃する対象の 3 次元航跡の json 表現 |

#### 4.7.4.1 SixDoFFighter クラスを用いる場合

SixDoFFighter クラスは、observables["spec"]["dynamic"]に固有の要素を持たない。commands["motion"]については、Fighter::setFlightControllerMode(mode)で指定した飛行制御モードに応じて異なり、飛行制御モードを"fromDirAndVel"とした場合の要素は表 4.７－８のとおりであり、"fromManualInput"又は"direct"とした場合の要素は表 4.７－５のとおりである。

表 4.７－４　飛行制御モードが fromDirAndVel の場合の commands["motion"]の要素

| キー名(インデントは階層を表す) | 型 | 概要 | 備考 |
|---|---|---|---|
| dstDir | array(double) | 目標進行方向(単位ベクトル) | いずれか一つを指定 |
| dstTurnRate | array(double) | 目標旋回率[rad/s ] | |
| dstAlt | double | 目標高度[m] | dstDir を指定した場合のみ |
| dstV | double | 目標速度[m/s] | いずれか一つを指定 |
| dstAccel | double | 目標加速度[$m/s^2$] | |
| dstThrust | double | 目標推力[N] | |
| dstThrottle | double | 目標スロットル(0～1 で正規化) | |

表 4.７－５　飛行制御モードが fromManualInput 又は direct の場合の commands["motion"]の要素

| キー名(インデントは階層を表す) | 型 | 概要 | 備考 |
|---|---|---|---|
| roll | double | ロールに関する指示([-1,+1]で正規化し、負側を左旋回とする。) | |
| pitch | double | ピッチに関する指示([-1,+1]で正規化し、負側を下降とする。) | |
| yaw | double | ヨーに関する指示([-1,+1]で正規化し、負側を左旋回とする。) | |
| accel | double | 加減速による速度指示([-1,+1]で正規化し、負側を減速とする。) | いずれか一つを指定 |
| throttle | double | スロットルによる速度指示([0,+1]で正規化する。) | |

#### 4.7.4.2 CoordinatedFighter クラスを用いる場合

CoordinatedFighter クラスは、observables["spec"]["dynamic"]として表 4.７－６に示す要素を持つ。commands["motion"]については、Fighter::setFlightControllerMode(mode)で指定した飛行制御モードに応じて異なり、飛行制御モードを"fromDirAndVel"とした場合の要素は表 4.７－７のとおりであり、"direct"とした場合の要素は表 4.７－８のとおりである。

表 4.7－6　CoordinatedFighter クラスの observables["spec"]["dynamics"]の要素

| キー名(インデントは階層を表す) | 型 | 概要 |
|---|---|---|
| rollMax | double | ロール角速度の上限値[rad/s] |

表 4.7－7　飛行制御モードが"fromDirAndVel"の場合の commands["motion"]の要素

| キー名(インデントは階層を表す) | 型 | 概要 | 備考 |
|---|---|---|---|
| dstV | double | 目標速度[m/s] | いずれか一つを指定 |
| dstAccel | double | 目標加速度[$m/s^2$] | |
| dstThrust | double | 目標推力[N] | |
| dstThrottle | double | 目標スロットルコマンド | |
| dstDir | array(double) | 目標進行方向(単位ベクトル) | いずれか一つを指定 |
| dstTurnRate | array(double) | 目標角速度[rad/s ] | |
| dstAlpha | double | 目標迎角[rad] | |
| ey | array(double) | 目標とする機体$y$軸方向(単位ベクトル) | dstAlpha を指定した場合のみ |

表 4.7－8　飛行制御モードが"direct"の場合の commands["motion"]の要素

| キー名(インデントは階層を表す) | 型 | 概要 | 備考 |
|---|---|---|---|
| roll | double | ロールに関する指示([-1,+1]で正規化し、負側を左旋回とする。) | |
| pitch | double | ピッチに関する指示([-1,+1]で正規化し、負側を下降とする。) | |
| accel | double | 加減速による速度指示([-1,+1]で正規化し、負側を減速とする。) | いずれか一つを指定 |
| throttle | double | スロットルによる速度指示([0,+1]で正規化する。) | |

#### 4.7.4.3 MassPointFighterFighter クラスを用いる場合

MassPointFighter クラスは、observables["spec"]["dynamic"]として表 4.７－９に示す要素を持ち、commands["motion"]として表 4.７－１０に示す要素を持つ。

表 4.７－９　MassPointFighter クラスの observables["spec"]["dynamics"]の要素

| キー名(インデントは階層を表す) | 型 | 概要 |
|---|---|---|
| vMin | double | 速度の下限値[m/s] |
| vMax | double | 速度の上限値[m/s] |
| rollMax | double | ロール角速度の上限値[rad/s] |
| pitchMax | double | ピッチ角速度の上限値[rad/s] |
| yawMax | double | ヨー角速度の上限値[rad/s] |

表 4.７－１０　MassPointFighter クラスの commands["motion"]の要素

| キー名(インデントは階層を表す) | 型 | 概要 | 備考 |
|---|---|---|---|
| roll | double | ロール角速度[rad/s ] | |
| pitch | double | ピッチ角速度[rad/s ] | |
| yaw | double | ヨー角速度[rad/s ] | |
| accel | double | 加減速による速度指示([-1,+1]で正規化し、負側を減速とする。) | いずれか一つを指定 |
| throttle | double | スロットルによる速度指示([0,+1]で正規化する。) | |

# ５独自クラス、モデルの実装方法について

本項では、ユーザーが頻繁に独自クラス、モデルを定義して使用することになるものについて実装方法の概要をまとめる。

## 5.1 Agent

独自の Agent クラスを実装する場合の大まかな流れは以下の通りである。
(1)単一の PhysicalAsset を対象とする場合は SingleAssetAgent クラスを、それ以外の場合は Agent クラスを継承する。
(2)Observation と Action の形式を決め、get_observation_space 関数と get_action_space 関数をオーバーライドする。
(3)observables から Observation を生成する makeObs 関数と、Action から decision と commands を生成する deploy 関数をオーバーライドする。このとき、これらの関数だけでは難しくより細かい周期で処理する必要がある場合も想定されるため、必要に応じて perceive、control、behave 関数をオーバーライドする。
(4)modelConfig として設定可能とするパラメータを選択し、他の登場物に依存する初期化処理が必要な場合は validate 関数もオーバーライドし、初期化処理を記述する。
(5)模倣学習を使用する場合は、模倣対象として想定する Agent クラスの decision と commands から、自身の action_space の中で最も類似している Action を計算する convertActionFromAnother 関数をオーバーライドする。また、control 関数の中身を模倣対象の decision と commands に応じて変更したい場合、controlWithAnotherAgent 関数をオーバーライドする。
(6)メンバ変数 observables と commands が適切に計算されているかを確認する。特に、このクラスが模倣される側となることを想定する場合は observables に decision が含まれていることを確認する。
(7) C++で実装した場合、Pybind11 を用いて Python 側へ公開する。
(8) Factory へクラスを登録する処理をどのタイミングで実行するかを決定する。インポート時に呼ばれるように記述してもよいし、ユーザーがインポート後に手動で登録するものとしてもよい。
(9)json ファイル等を用意して modelConfig を記述し、Factory にモデル登録ができるようにする。このとき、perceive、control、behave をオーバーライドした場合であって、1[tick]ごとではない処理周期としたい場合には、4.１.３項に従い modelConfig に処理周期に関する記述を追加する。
(10)以上により、SimulationManager の config["AgentConfigDispatcher"]に登録したモデル名を指す alias 要素を記述することで独自の Agent が使用可能となる。

Python クラスとして Agent クラスを実装する際のひな形を以下に示す。

```
class UserAgent(Agent): # Agent の代わりに SingleAssetAgent を継承してもよい
  def __init__(self, modelConfig: nljson, instanceConfig: nljson):
    super().__init__(modelConfig, instanceConfig))
    if(self.isDummy):
      return #Factory によるダミー生成のために空引数でのインスタンス化に対応させる
    # 以上３行の呼び出しは原則として必須である。

    #modelConfig から値を取得する場合、直接[]でアクセスすると nljson 型で得られる。
    #Python のプリミティブ型にするためには()を付けて__call__を呼ぶ必要がある。
    self.hoge = self.modelConfig["hoge"]()

    #3.４項のユーティリティを用いて確率的な選択やデフォルト値の設定も可能。
    #その場合の出力は Python プリミティブ型となるため()の付加は不要。
    #現バージョンでは乱数生成器には std::mt19937 しか使用できないが、
    #self.randomGen として基底クラスで予め生成されているためこれを使用する。
    self.withRandom = getValueFromJsonKR(self.modelConfig,"R",self.randomGen)
    defaultValue = 1234
```

```
    self.withDefault = getValueFromJsonKD(self.modelConfig, "D", defaultValue)
    self.withRandomAndDefault = getValueFromJsonKRD(
      self.modelConfig, "RD", self.randomGen, defaultValue)
  def action_space(self) -> gym.spaces.Space:
    # 行動空間の定義(必須)
    return gym.spaces.MultiDiscrete([3,3,3,3]) #所要の Space を返す。

  def observation_space(self) -> gym.spaces.Space:
    # 状態空間の定義(必須)
    return gym.space.Box(low=0.0, high=1.0, shape=(100,)) #所要の Space を返す。

  def makeObs(self) -> object:
    # Observation の生成(必須)

    #時刻の取得方法
    time = self.manager.getTime()

    #Ruler の情報の取得方法
    ruler = self.manager.getRuler()() #getRuler()で RulerAccessor 型の weak_ptr が
                                      #得られるため、更に()で本体を取得する
    rulerObs = ruler.observables #nljson 型で得られる

    #parent の情報の取得方法
    #(1)SingleAssetAgent から継承した場合、self.parent を使用する。
    parentObs = self.parent.observables #nljson 型で得られる
    motion = MotionState(parentObs["motion"]) #運動情報は MotionState として取得する
    #(2)Agent から継承した場合、dict 型の self.parents を使用する。
    for parentFullName, parent in self.parents.items():
      parentObs = parent.observables #nljson 型で得られる
    return np.zeros([100]) #observables を加工し、所要の Observation を返す。

  def deploy(self,action: object):
    #Action の解釈と decision and/or commands の生成 (1step に 1 回実行)(必須)

    #decision は Agent 自身の observables の一部として記述する。
    #使用しない場合は省略しても差し支えない。
    self.observables[self.parent.getFullName()]["decision"]={
      "Roll":["Don' t care"],
      "Horizontal":["Az_BODY",0.0],
      "Vertical":["El",0.0],
      "Throttle":["Vel",300.0],
      "Fire":[False,Track3D()]
    }

  def validate(self):
    #コンストラクタ外の初期化処理(必須ではない)
```

```
    #ruler や parent の observables に依存するものがあるような場合を想定している。
    pass


  def perceive(self):
    #より細かい tick 単位の処理(perceive)を記述(必須ではない)
    #decision and/or commands の複雑な生成処理を行う場合等に用いる。
    pass

  def control(self):
    #1tick 単位の処理(control)を記述(必須ではない)
    #decision and/or commands の複雑な生成処理を行う場合等に用いる。
    #commands は deploy で計算してもよいが、control でより高頻度に計算してもよい。
    self.commands[self.parent.getFullName()] = {
      "motion": { #機動の指定。以下の指定方法は一例。
        "dstDir": np.array([1.0, 0.0, 0.0]), #進みたい方向を指定
        "dstV": 300.0 #進みたい速度を指定
      },
      "weapon": {
        "launch": False, #射撃要否を bool で指定
        "target": Track3D().to_json() #射撃目標の Track3D を json 化して指定
      }
    }

  def behave(self):
    #1tick 単位の処理(behave)を記述(必須ではない)
    #decision and/or commands の複雑な生成処理を行う場合等に用いる。
    pass
```

## 5.2 Ruler

独自の Ruler クラスを実装する場合の大まかな流れは以下のとおりである。基本となる基底クラスは Ruler クラスである。
(1)各種のコールバック関数のうち、必要なものをオーバーライドする。
(2)必要に応じてイベントハンドラを定義し、適切な場所(通常は onEpisodeBegin)で addEventHandler を呼んで SimulationManager に登録する。
(3)checkDone 関数をオーバーライドし、終了判定と勝敗判定を記述する。
(4)modelConfig として設定可能とするパラメータを選択し、他の登場物に依存する初期化処理が必要な場合は validate 関数もオーバーライドし、初期化処理を記述する。
(5)メンバ変数 score、stepScore、dones、winner、endReason が適切に計算されているかを確認する。endReason は enum class とし、基底クラスのものをそのまま使わない場合は再定義する。
(6)C++で実装した場合、Pybind11 を用いて Python 側へ公開する。
(7) Factory へクラスを登録する処理をどのタイミングで実行するかを決定する。インポート時に呼ばれるように記述してもよいし、ユーザーがインポート後に手動で登録するものとしてもよい。
(8)json ファイル等を用意し、Factory にモデル登録ができるようにする。
(9)以上により、登録したモデル名を SimulationManager の config["Ruler"]に記述することで独自の Ruler が使用可能となる。

## 5.3 Reward

独自の Reward クラスを実装する場合の大まかな流れは以下の通りである。
(1)陣営単位での報酬とするのか、Agent ごとの報酬とするのかに応じて、TeamReward クラスと AgentReward クラスのいずれかを基底クラスとして継承する。
(2)各種コールバック関数のうち、必要なものをオーバーライドする。
(3)必要に応じてイベントハンドラを定義し、適切な場所(通常は onEpisodeBegin)で addEventHandler を呼んで SimulationManager に登録する。
(4)modelConfig として設定可能とするパラメータを選択し、他の登場物に依存する初期化処理が必要な場合は validate 関数もオーバーライドし、初期化処理を記述する。
(5)メンバ変数 reward、totalReward が適切に計算されているかを確認する。
(6)C++で実装した場合、Pybind11 を用いて Python 側へ公開する。
(7) Factory へクラスを登録する処理をどのタイミングで実行するかを決定する。インポート時に呼ばれるように記述してもよいし、ユーザーがインポート後に手動で登録するものとしてもよい。
(8)json ファイル等を用意し、Factory にモデル登録ができるようにする。
(9)以上により、登録したモデル名を SimulationManager の config["Rewards"]に記述することで独自の Reward が使用可能となる。

Python クラスとして Reward クラスを実装する際のひな形を以下に示す。

```
class UserReward(TeamReward):
  #チーム全体で共有する報酬は TeamReward を継承し、
  #個別の Agent に与える報酬は AgentReward を継承する。
  def __init__(self, modelConfig: nljson, instanceConfig: nljson):
    super().__init__(modelConfig, instanceConfig))
    if(self.isDummy):
      return #Factory によるダミー生成のために空引数でのインスタンス化に対応させる
    #以上 3 行の呼び出しは原則として必須である。
    #また、modelConfig の読み込み等は Agent クラスと共通である。

  def onEpisodeBegin(self):
    #エピソード開始時の処理(必要に応じてオーバーライド)
    #基底クラスにおいて config に基づき報酬計算対象の設定等が行われるため、
    #それ以外の追加処理や設定の上書きを行いたい場合のみオーバーライドする。
    super().onEpisodeBegin()

  def onStepBegin(self):
    #step 開始時の処理(必要に応じてオーバーライド)
    #基底クラスにおいて reward(step 報酬)を 0 にリセットしているため、
    #オーバーライドする場合、基底クラスの処理を呼び出すか、同等の処理が必要。
    super().onEpisodeBegin()

  def onInnerStepBegin(self):
    #インナーステップ開始時の処理(必要に応じてオーバーライド)
    #デフォルトでは何も行わないが、より細かい報酬計算が必要な場合に使用可能。
    pass
```

```
def onInnerStepEnd(self):
  #インナーステップ終了時の処理(必要に応じてオーバーライド)
  #デフォルトでは何も行わないが、より細かい報酬計算が必要な場合に使用可能。
  pass

def onStepEnd(self):
  #step 終了時の処理(必要に応じてオーバーライド)
  #主にここで報酬の計算を実施することになる。
  #基底クラスにおいて累積報酬の更新を行っているため、
  #オーバーライドする場合、基底クラスの処理を呼び出すか、同等の処理が必要。
  for team in self.reward:
    #team に属している Asset(Fighter)を取得する例
    for f in self.manager.getAssets(
                        lambda a:a.getTeam()==team and isinstance(a,Fighter)):
      if(f().isAlive()): # ここでの f は weak_ptr となるので()が必要
        self.reward[team] += 0.1 #例えば、残存数に応じて報酬を与える場合
  super().onStepEnd()

def onEpisodeEnd(self):
  #エピソード終了時の処理(必要に応じてオーバーライド)
  #デフォルトでは何も行わないが、より細かい報酬計算が必要な場合に使用可能。
  pass
```

## 5.4 その他の Callback

独自の Callback クラスを実装する場合の大まかな流れは Ruler や Reward と同様である。

# 6 同梱サンプルの概要

本項では、root/sample に格納されている各種サンプルの使用方法の概要についてまとめる。

## 6.1 基本的な空対空目視外戦闘シミュレーションのために必要なクラスの一覧

基本的な空対空目視外戦闘のシミュレーションを実行するために必須なクラスは表 6.1－1 の通りである。これらに対応するモデルも予め基準となるデフォルト値により定義済であり、本シミュレータモジュールをインポートした際に自動的に Factory に登録される。

表 6.1－1　必須クラスの一覧

| 種類 | クラス名 | 概要 | 実装言語 |
|---|---|---|---|
| Ruler | R4BVRRuler01 | 基本的な空対空目視外戦闘のルールを定義したもの | C++ |
| PhysicalAsset | Fighter | 戦闘機・無人機モデルの構成要素のうち運動モデル以外の共通部分を実装した基底クラス | |
| | SixDoFFighter | 6 自由度の運動モデルで動く戦闘機・無人機 | |
| | StevensFighter | SixDoFFighter のうち、空力モデルをテーブルデータで表現したもの | |
| | MorelliFighter | SixDoFFighter のうち、空力モデルを多項式モデルで表現したもの | |
| | CoorinatedFighter | 横滑りを無視し、回転運動を簡略化した戦闘機・無人機 | |
| | MassPointFighter | 質点モデルで動く戦闘機・無人機 | |
| | Missile | 簡略化した誘導弾モデルを実装したもの | |
| | AircraftRadar | 簡略化した戦闘機・無人機センサ(レーダ) | |
| | MWS | 簡略化した戦闘機・無人機センサ(MWS) | |
| | MissileSensor | 簡略化した誘導弾センサ | |
| | Propulsion | 戦闘機・無人機の推進系(エンジン)の基底クラス | |
| | IdealDirectPropulsion | MassPointFighter で使用する、推力を直接指定できる理想的なエンジンモデルを実装したもの | |
| | SimpleFighterJetEngine | SixDoFFighter 及び CoorinatedFighter で使用する、簡略化したジェットエンジンモデルを実装したもの | |
| Controller | Fighter:: SensorDataSharer | 戦闘機・無人機のセンサ探知データを編隊内で共有する処理の送信側を簡易的に実装したもの | |
| | Fighter:: SenorDataSanitizer | 戦闘機・無人機のセンサ探知データを編隊内で共有する処理の受信側を簡易的に実装したもの | |
| | Fighter:: OtherDataSharer | 戦闘機・無人機のセンサ探知データ以外を編隊内で共有する処理の送信側を簡易的に実装したもの | |
| | Fighter:: OtherDataSanitizer | 戦闘機・無人機モデルのセンサ探知データ以外を編隊内で共有する処理の受信側を簡易的に実装したもの | |
| | Fighter:: HumanIntervention | 戦闘機・無人機モデルの射撃行為に関する人間の介入モデルを簡易的に実装したもの | |
| | Fighter:: WeaponController | 人間の介入を受けた後の射撃判断結果に従い、誘導弾の発射等の処理を行うモデルを実装したもの | |
| | SixDoFFighter:: FlightController | commands に従い SixDoFFighter の飛行制御を行うモデルを実装したもの | |
| | CoordinatedFighter:: FlightController | commands に従い CoodinatedFighter の飛行制御を行うモデルを実装したもの | |
| | MassPointFighter:: FlightController | commands に従い MassPointFighter の飛行制御を行うモデルを実装したもの | |
| | PropNav | 誘導弾飛翔制御のための単純な比例航法を実装したもの | |
| Agent | R4InitialFighterAgent01 | 基本的なルールベースによって空対空目視外戦闘を行う初期行動判断モデルを実装したもの | |

## 6.2 Reward の実装例

報酬を計算する Reward については、表 6.２－１に示す 5 種類をサンプルとして実装している。

表 6.２－１　Reward の実装例

| クラス名 | 概要 | パス | 実装言語 |
|---|---|---|---|
| ScoreReward | Ruler の得点をそのまま報酬として扱う例 | root/include/Reward.h<br>root/src/Reward.cpp | C++ |
| R4BVRBasicReward01 | R4BVRRuler01 の得点計算と同様の観点で即時報酬を与える例 | root/include/R4BVRBasicReward01.h<br>root/src/R4BVRBasicReward01.cpp | C++ |
| R4RewardSample01 | 得点とは直接関係しない運動や探知、射撃の状況に応じた報酬を、陣営単位で与える例 | root/sample/OriginalModelSample/include/R4RewardSample01.h<br>root/sample/OriginalModelSample/src/R4RewardSample01.cpp | C++ |
| R4PyRewardSample01 | | root/sample/OriginalModelSample/OriginalModelSample/R4PyRewardSample01.py | Python |
| R4RewardSample02 | 得点とは直接関係しない運動や探知、射撃の状況に応じた報酬を、Agent 単位で与える例 | root/sample/OriginalModelSample/include/R4RewardSample02.h<br>root/sample/OriginalModelSample/src/R4RewardSample02.cpp | C++ |
| R4PyRewardSample02 | | root/sample/OriginalModelSample/OriginalModelSample/R4PyRewardSample02.py | Python |
| WinLoseReward | 勝敗のみに基づく報酬を与える例 | root/sample/OriginalModelSample/WinLoseReward.h<br>root/sample/OriginalModelSample/WinLoseReward.cpp | C++ |

## 6.3 Agent の実装例

深層強化学習を行うための Agent の実装例は表 6.３－１に示す通り、一機分の行動判断を行うクラスと陣営全機分の行動判断を行うクラスを C++と Python の両方で実装している。モデルの登録は C++版のみを行っているが、Python 版も modelConfig は同一であり、Python 版を使用したい場合はクラス名を指定している部分を書き換えればよい。Observation と Action の詳細や modelConfig で設定可能なパラメータについてはソースコード中の説明を参照されたい。

表 6.３－１　Agent の実装例

| クラス名 | 概要 | パス | 実装言語 |
|---|---|---|---|
| R4AgentSample01S | 1 体の Agent につき 1 機分の行動判断を行う例 | root/sample/OriginalModelSample/include/R4AgentSample01S.h<br>root/sample/OriginalModelSample/src/R4AgentSample01S.cpp | C++ |
| R4PyAgentSample01S | | root/sample/OriginalModelSample/OriginalModelSample/R4PyAgentSample01S.py | Python |
| R4AgentSample01M | 1 体の Agent で陣営全体の機体の行動判断を行う例。各機の Observation と Action は 1 機分の Agent と同一であり、単にそれを全機分並べただけである。 | root/sample/OriginalModelSample/include/R4AgentSample01M.h<br>root/sample/OriginalModelSample/src/R4AgentSample01M.cpp | C++ |
| R4PyAgentSample01M | | root/sample/OriginalModelSample/OriginalModelSample/R4PyAgentSample01M.py | Python |

## 6.4 Viewer の実装例

戦闘場面の可視化を行う Viewer の例として、pygame と OpenGL を用いた単純な可視化を行う GodView クラスを root/ASRCAISim1/viewer/GodView.py に Python クラスとして実装している。6.4 図 6.4－1 に示すように、戦域を真上から見た平面図と、真南から見た鉛直方向の断面図が表示される。また、画面の左上には時刻、得点、報酬の状況が表示される。

なお、現在の実装では 1 枚のサーフェスのまま強引に 2 種類の図を描画しているため、一方の描画範囲外になったオブジェクトがもう一方の図にはみ出て描画されてしまうことがある。

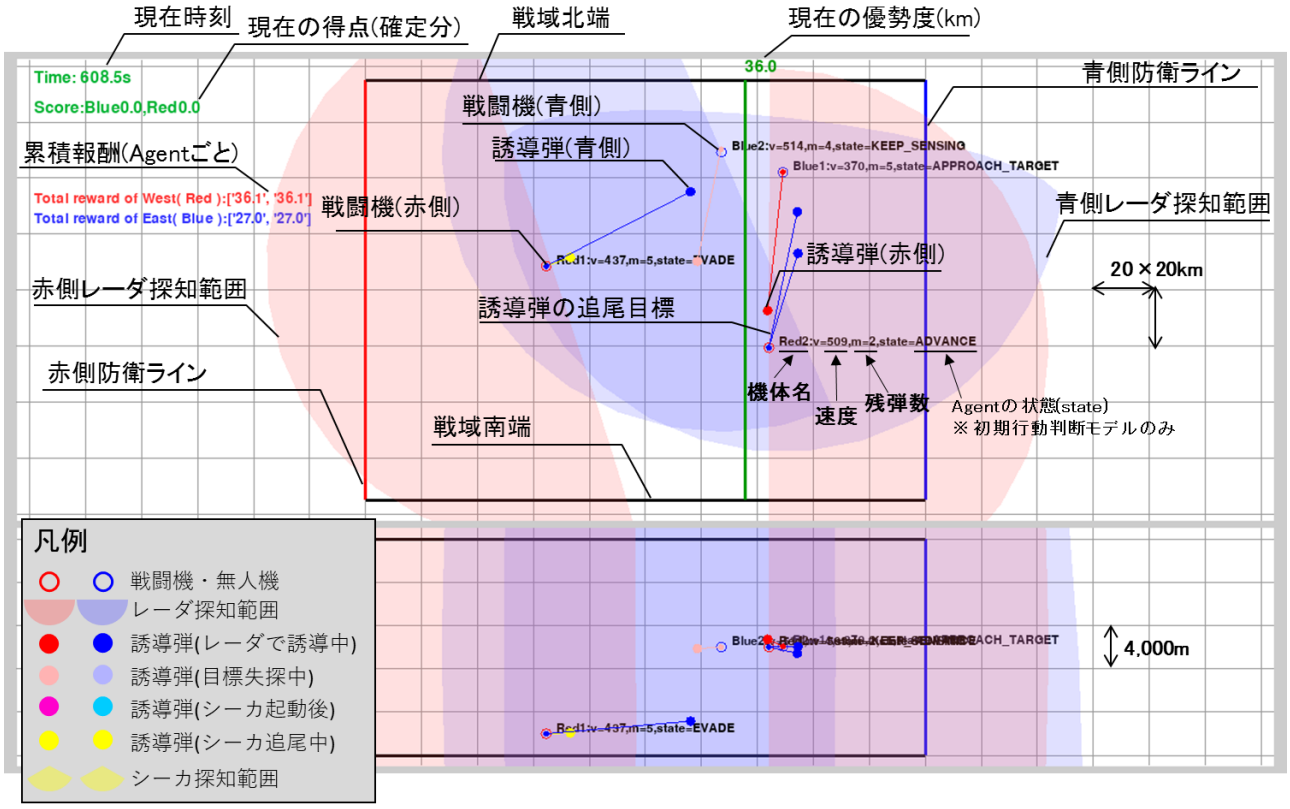


図 6.4－1　GodView の画面表示例(上側が平面図、下側が側面図)

また、表 6.5－1 に示す GodViewStateLogger を用いて戦闘場面の可視化に必要な情報をログとして出力しておき、後から SimulationManager と独立に動作可能な可視化用クラスの例として、GodViewStateLoader クラスを root/ASRCAISim1/viewer/GodViewLoader.py に実装している。

## 6.5 Loggerの実装例

戦闘結果のログ保存を行うLoggerの例として、表 6.5－1に示す5種類を実装している。

表 6.5－1　Loggerの実装例

| クラス名 | 概要 | パス | 実装言語 |
|---|---|---|---|
| BasicLogger | 指定したエピソード数ごとに、1 エピソード分の各戦闘機・無人機と誘導弾の飛行軌跡を指定したtick数おきに記録したcsvファイルを出力する例 | root/ASRCAISim1/logger/LoggerSample.py | Python |
| GodViewLogger | GodView の描画サーフェスを指定したエピソード数ごとに指定したtick数おきに連番画像ファイルとして出力する例 | root/ASRCAISim1/logger/GodViewLogger.py | Python |
| MultiEpisodeLogger | 各エピソードの得点や報酬、勝敗や直近の勝率等を指定したエピソード数おきに単一のcsv ファイルに出力する例。このクラスは並列化に対応しておらず、単一の環境インスタンスにおける複数エピソードの記録となる。 | root/ASRCAISim1/logger/MultiEpisodeLogger.py | Python |
| RayMultiEpisodeLogger | MultiEpisodeLogger を、ray の機能を用いて複数の Simulationmanager インスタンスで共有した単一のログファイルとして生成できるよう拡張したもの。 | root/addons/rayUtility/logger/RayMultiEpisodeLogger.py | Python |
| GodViewStateLogger | GodView と同等の戦闘場面描画に必要な情報をログとして出力する例。 | root/ASRCAISim1/logger/GodViewStateLogger.py | Python |
| ExpertTrajectoryWriter | ExpertWrapperが持つexpertとimitatorの情報を使用し、ray RLlibで模倣学習をするためのTrajectoryデータファイルを記録する例。 | root/addons/rayUtility/logger/ExpertTrajectoryWriter.py | Python |

## 6.6 基本的な SimulationManager の config 例

SimulationManager の config や Factory に登録するモデルの config は単一の json ファイルに記述する必要はなく、表 6.6－1 に示すように複数のファイルや dict に分けてリストで与えることによって再利用が容易になる。

表 6.6－1　SimulationManager 及び Factory 追加モデルの config 例

| 分類 | パス(root/sample/以下) | 概要 |
|---|---|---|
| 戦闘場面の指定 | MinimumEvaluation/common/R4_contest_mission_config.json | 一定の幅を持ったランダムな初期条件で、Red と Blue による 2 機種混成の 4 対 4 の対称戦を実施する例 |
| | HandyRLSample/configs/R4_contest_mission_config.json | |
| | raySample/configs/R4_contest_mission_config.json | |
| PhysicalAsset モデルの登録 | MinimumEvaluation/common/R4_contest_asset_models.json | 上記戦闘場面に登場する PhysicalAsset モデルを Factory に追加する例 |
| | HandyRLSample/configs/R4_contest_asset_models.json | |
| | raySample/configs/R4_contest_asset_models.json | |
| Agent 、 Ruler 、 Reward モデルの登録 | MinimumEvaluation/common/R4_contest_agent_ruler_reward_models.json | 上記戦闘場面に登場する Agent、Ruler、Reward モデルを Factory に追加する例。<br>0 項に従い実装した独自モデルを追加する場合はこの例に従う。 |
| | HandyRLSample/configs/R4_contest_agent_ruler_reward_models.json | |
| | raySample/configs/R4_contest_agent_ruler_reward_models.json | |
| Agent 構成、報酬体系の指定 | HandyRLSample/configs/R4_contest_learning_config_S.json | BVRMatchMaker を用いて学習するための AgentConfigDispatcher の設定と、使用する報酬モデルの設定を記述する例。<br>json ファイル名末尾の S は 1 体の Agent で 1 機を操作する行動判断モデルを学習対象とした例であることを意味する。 |
| | raySample/configs/R4_contest_learning_config_S.json | |
| | HandyRLSample/configs/R4_contest_learning_config_M.json | BVRMatchMaker を用いて学習するための AgentConfigDispatcher の設定と、使用する報酬モデルの設定を記述する例。<br>json ファイル名末尾の M は 1 体の Agent で陣営全体を操作する行動判断モデルを学習対象とした例であることを意味する。 |
| | raySample/configs/R4_contest_learning_config_M.json | |
| | raySample/configs/R4_contest_gathering_config_S.json | ray RLlib で模倣学習を行うサンプルにおいて、expert の軌跡データをログとして保存する際の Agent 構成を記述した例。<br>この例では BVRMatchMaker は使用しない。 |
| | raySample/configs/R4_contest_gathering_config_M.json | |

### 6.6.1 戦闘場面の指定に係る config

戦闘場面の指定に係る config では、時間ステップ、ルール、Asset について記述している。以下に R4_contest_mission_config.json を例に簡単な説明を示す。各項目の詳細は 4.６項を参照のこと。

```
{
  "Manager":{
    "TimeStep":{
      "baseTimeStep":0.05, #1tick の時間ステップ
      "agentInterval":20 #gym.Env として 1 回の step 関数で何 tick 進めるか
    },
    "Ruler":"R4ContestRuler", #使用する Ruler モデルの名称
    "Assets":{
      "type":"group",
      "Blue":{ #Blue 側の Asset に関する設定
        "type":"gtoup","order":"fixed",
        "names":["Blue1","Blue2","Blue3","Blue4"], #Asset 名
        "elements":[ #後述の AssetConfigDispatcher から"Fighters"を参照し、順番に割当
          #ランダム要素があっても正しく指定されるように instance と index を指定
          {"type":"alias","alias":"Fighters","instance":"Same","index":0},
          {"type":"alias","alias":"Fighters","instance":"Same","index":1},
          {"type":"alias","alias":"Fighters","instance":"Same","index":2},
          {"type":"alias","alias":"Fighters","instance":"Same","index":3}
        ],
        "overrider":[#overrider により設定を修正していく
          {"type":"broadcast","number":4,
            "element":{"type":"alias","alias":"OnEastLine"}#初期配置を東側にする
          },
          {"type":"broadcast","number":4,
            "element":{
              "type":"direct","value":{
                "instanceConfig":{"datalinkName":"BlueDatalink"}#BlueDatalink に加入
              }
            }
          },
          {"type":"group","order":"fixed","elements":[#Agent に関する設定
            #AgentConfigDispatcher の"BlueAgents"を参照するよう指定
            {"type":"direct","value":{
              "Agent":{"type":"alias","alias":"BlueAgents","instance":"g1","index":0}
            }},
            {"type":"direct","value":{
              "Agent":{"type":"alias","alias":"BlueAgents","instance":"g1","index":1}
            }},
            {"type":"direct","value":{
              "Agent":{"type":"alias","alias":"BlueAgents","instance":"g1","index":2}
            }},
            {"type":"direct","value":{
              "Agent":{"type":"alias","alias":"BlueAgents","instance":"g1","index":3}
            }},
          ]}
        ]
      },
```

```
    "Red":{ #Red 側の Asset に関する設定。記述要領は Blue 側と同様。
      "type":"gtoup","order":"fixed",
      "names":["Red1","Red2","Red3","Red4"],
      "elements":[
        {"type":"alias","alias":"Fighters","instance":"Same","index":0},
        {"type":"alias","alias":"Fighters","instance":"Same","index":1},
        {"type":"alias","alias":"Fighters","instance":"Same","index":2},
        {"type":"alias","alias":"Fighters","instance":"Same","index":3}
      ],
      "overrider":[
        {"type":"broadcast","number":4,
          "element":{"type":"alias","alias":"OnWestLine"}
        },
        {"type":"broadcast","number":4,
          "element":{
            "type":"direct","value":{
              "instanceConfig":{"datalinkName":"RedDatalink"}
            }
          }
        },
        {"type":"group","order":"fixed","elements":[
          {"type":"direct","value":{
            "Agent":{"type":"alias","alias":"RedAgents","instance":"g1","index":0}
          }},
          {"type":"direct","value":{
            "Agent":{"type":"alias","alias":"RedAgents","instance":"g1","index":1}
          }},
          {"type":"direct","value":{
            "Agent":{"type":"alias","alias":"RedAgents","instance":"g1","index":2}
          }},
          {"type":"direct","value":{
            "Agent":{"type":"alias","alias":"RedAgents","instance":"g1","index":3}
          }},
        ]}
      ]
    }
  },
  "CommunicationBuffers":{ #Asset 間の通信に関する設定。詳細は 3.５項を参照。
    "BlueDatalink":{"participants":["PhysicalAsset:Blue/Blue[0-9]+"]},
    "RedDatalink":{"participants":["PhysicalAsset:Red/Red[0-9]+"]}
  },
  "AssetConfigDispatcher":{
    #機体モデルを指定するもの
    "Large":{"type":"PhysicalAsset","model":"R4ContestFighterLarge"},
    "Small":{"type":"PhysicalAsset","model":"R4ContestFighterSmall"},
    "OnWestLine":{#初期配置(位置、速度、針路)を西側の防衛ライン上にするもの
      "type":"direct","value":{
        "instanceConfig":{
          "pos":{
            "type":"uniform","low":[-20000,-100000,-10000],"high":[20000,-100000,-10000]
          },
```

```
            "vel":300,
            "heading":90
          }
        }
      },
      "OnEastLine":{#初期配置を東側の防衛ライン上にするもの
        "type":"direct","value":{
          "instanceConfig":{
            "pos":{
              "type":"uniform","low":[-20000,100000,-10000],"high":[20000,100000,-10000]
            },
            "vel":300,
            "heading":270
          }
        }
      },
      "Fighters":{#編隊の編成を指定するもの
        "type":"group","order":"fixed",
        "elements":[
          {"type":"alias","alias":"Large"},
          {"type":"alias","alias":"Large"},
          {"type":"alias","alias":"Small"},
          {"type":"alias","alias":"Small"}
        ]
      }
    }
  }
}
```

### 6.6.2 Agent 構成、報酬体系の指定に係る config

Agent 構成、報酬体系の指定に係る config では、Reward と Agent について記述している。以下に R4_contest_learning_config_S.json を例に簡単な説明を示す。詳細は 4.6 項を参照のこと。

```
{
  "Manager":{
    "Rewards":[ #モデル名と報酬計算対象を指定する
      {"model":"MyWinLoseRewardSample,"target":"All"},
      {"model":"MyInstantZerosumScoreRewardSample","target":"All"},
      {"model":"MyTeamRewardSample","target":"All"},
      {"model":"MyAgentRewardSample","target":"All"}
    ],
    "AgentConfigDispatcher":{
      #初期行動判断モデルを指定する本体
      "Initial_e":{"type":"Internal","model":"R4RandomInitial"},
      #学習用 Agent モデルとポリシー名を指定する本体
      "Learner_e":{"type":"External","model":"MyAgentSample01S","policy":"Learner"},
      #模倣学習用 ExpertWrapper を指定する本体
      "Expert_e":{"type":"ExpertBI","imitatorModel":"MyAgentSample01S",
```

```
        "expertModel":"R4RandomInitial","identifier":"Traj1"}
      #陣営一つ分(４機分)の Agent 群として記述
      #BVRMatchMaker を使用する場合で、Internal でない Policy を用いる場合は、
      #ここの名称(例えば"Learner")と、対応する Policy 名が一致していなければならない。
      "Learner":{"type":"broadcast","number":4,
        "elements":{"type":"alias","alias":"Learner_e"}
      },
      "Initial":{"type":"broadcast","number":4,
        "element":{"type":"alias","alias":"Initial_e"}
      },
      "Expert":{"type":"broadcast","number":4,
        "element":{"type":"alias","alias":"Initial_e"}
      },
      "BlueAgents":{#Blue 側の Agent に関する設定
        #Asset 側からはこれが alias として参照され、インスタンス化されていく。
        #Agent 側としては、予め必要な数だけ AgentConfig を並べておく。
        #直接中身を記述してもよいが、本サンプルのように alias で小分けに記述してもよい。
        # BVRMatchMaker を使用する場合で、Internal でない Policy を用いる場合は、
        #ここをポリシー名と同名の alias で指定できるようにしておかなければならない。
        "type":"alias","alias":"Learner",#学習用 Agent を指定
        "overrider":[
          {"type":"group","order":"fixed","elements":[
            {"type":"direct","value":{"name":"Blue1"}}, #Agent 名と port 名を個別に指定
            {"type":"direct","value":{"name":"Blue2"}},
            {"type":"direct","value":{"name":"Blue3"}},
            {"type":"direct","value":{"name":"Blue4"}}
          ]}
        ]
      },
      "RedAgents":{#Red 側の Agent に関する設定。記述要領は Blue 側と同様。
        "type":"alias","alias":"Initial",#初期行動判断モデルを指定
        "overrider":[
          {"type":"group","order":"fixed","elements":[
            {"type":"direct","value":{"name":"Red1"}}, #Agent 名を個別に指定
            {"type":"direct","value":{"name":"Red2"}},
            {"type":"direct","value":{"name":"Red3"}},
            {"type":"direct","value":{"name":"Red4"}}
          ]}
        ]
      }
    }
  }
}
```

## 6.7 HandyRL を用いて学習を行うサンプル

root/sample/HandyRLSample には、HandyRL(を本シミュレータに合わせて改変及び機能追加したもの)を用いて行動判断モデルの学習を行うサンプルを同梱している。

### 6.7.1 元の HandyRL に対する改変・機能追加の概要

本サンプルでは HandyRL に以下のような改変・機能追加を行っている。

・Discrete 以外の action_space に対応
・MatchMaker を用いた行動判断モデルの動的選択に対応
・ExpertWrapper を用いた模倣学習に対応
・Ape-X 型の ε-greedy 探索に対応
・turn-based な環境に対する特殊化を削除

### 6.7.2 学習の実行方法

root/sample/HandyRLSample 上で、

```
python main.py R4_contest_sample_S.yaml --train
```

を実行すると、1 体で 1 機を操作する行動判断モデルについて、BVRMatchMaker を用いた Self-Play による学習が行われる。学習結果は./results/Single/YYYYmmddHHMMSS 以下に保存される。なお、R4_contest_sample_M.yaml に変更すると、1 体で陣営全体を操作する行動判断モデルとなる。

### 6.7.3 学習済モデルの評価

学習済モデルは、./results/Single/YYYYmmddHHMMSS/policies/checkpoints 以下に保存されている。このモデルの評価は root/sample/MinimumEvaluation/evaluator.py を使用することで可能である。root/sample/MinimumEvaluation/candidates.json を開き、例えば

```
"test":{
  "userModuleID":"HandyRLSample01S",
  "args":{"weightPath":<学習済の.pth ファイルのフルパス>}
}
```

のように候補を追加し、root/sample/MinimumEvaluation 上で

```
python evaluator.py "test" "Rule-Fixed" -n 10 -v -l "eval_test.csv"
```

と実行すると、初期行動判断モデルとの対戦による学習済モデルの評価を行うことができる。

### 6.7.4 yaml の記述方法

本サンプルにおける yaml の記述方法は以下の通りである。

```
save_dir: ./result/Single #学習ログの保存場所
env_args: #環境に関する設定
 env: ASRCAISim1 #環境の名称
 env_config: #2.５.１.１項の GymManager のコンストラクタに与えられる context の dict
  config: #直接記述するのは"config"のみでよい
   - ./configs/R4_contest_mission_config.json
   - ./configs/R4_contest_learning_config_S.json
   - ./configs/R4_contest_asset_models.json
   - ./configs/R4_contest_agent_ruler_reward_models.json
   - Manager:
      ViewerType: None
      Callbacks: {}
      Loggers: {} #追加で Logger を使用したい場合はここに記述する
policy_config: #3.７項の MatchMaker の policy_config＋α
 Learner:
  multi_port: false #1 体で 1 陣営分を動かすか否か。省略時は False
  active_limit: 50 #保存された過去の重みを使用する数の上限を指定する
  is_internal: false #SimulationManager における Internal な Policy かどうか
  populate: #重み保存条件の指定
   first_population: 1000 #初回の保存を行うエピソード数
   interval: 1000 #保存間隔。0 以下の値を指定すると一切保存しない
   on_start: false #開始時の初期重みで保存するかどうか。省略時は False
   reset: float #重み保存時のリセット確率(0～1)。省略時は 0
  rating_initial: 1500 #初期レーティング
  rating_fixed: false #レーティングを固定するかどうか
  initial_weight: null #初期重み(リセット時を含む)のパス
  #以下は HandyRL 側で動く Policy モデル(torch.nn.Module を継承)に関する設定
  #main.py で使用可能なクラスオブジェクトを列挙し、その中から選択する
  model_class: R4TorchNNSampleForHandyRL #クラス名
  model_config: <略> #コンストラクタに与えられる config
 Initial:
  multi_port: false
  active_limit: null
  is_internal: true
  populate: null
  rating_initial: 1500
  rating_fixed: true
  initial_weight: null
  model_class: DummyInternalModel #クラス名
  model_config: {}
```

```
train_args:
 name: Learner #Trainer の名称。MatchMaker 側に matchType として伝達される。
 match_maker_class: BVRMatchMaker #使用する MatchMaker のクラス名
 match_monitor_class: BVRMatchMonitor #使用する MatchMonitor のクラス名
 turn_based_training: false #2 プレイヤーのゼロサムゲームか否か。False とする。
 observation: false #行動無しの観測ターンを許容するか否か。False とする。
 gamma: 0.99 #報酬の割引率
 forward_steps: 64 #replay のシーケンス長
 burn_in_steps: 8 #RNN を使用する際の burn-in の長さ。forward_steps の内数。
 compress_steps: 8 #observation を圧縮する際の一塊のステップ数
 entropy_regularization: 4.0e-3 #エントロピー項の係数 1
 entropy_regularization_decay: 0.1 #エントロピー項の係数 2
 exploration_config: # ε -greedy に関する設定
  use_exploration: true # ε -greedy による探索を行うか否か
  eps_start: 0.4 # ε の初期値
  eps_end: 0.4 # ε の終端値
  eps_decay: -1 # ε の変化率
  alpha: 7.0 #Ape-X における worker ごとの ε を定める指数
  cycle: 16 # ε を定める際の見かけ上の worker 数を水増しするための係数
 update_episodes: 10 #1 エポックのエピソード数
 batch_size: 32 #バッチサイズ(長さ forward_steps のシーケンス単位)
 minimum_episodes: 10 #学習を開始する最小エピソード数
 maximimum_episodes: 3000 #replay buffer に記憶するエピソード数の上限
 epochs: -1 #実施するエポック数。負数を指定すると上限なし
 num_batchers: 8 #Batcher の数
 eval_rate: 0.1 #学習に使用しない評価用エピソードの割合
 worker:
  num_parallel: 16 #worker の数
 lambda: 0.7 #TD(λ)の λ
 policy_target: UPGO #方策の更新に使用する target
 value_target: TD #価値関数の更新に使用する target
 seed: 0 #乱数のシード
 policy_to_train: Learner #学習対象の Policy 名
 policy_to_imitate: [] #模倣対象の Policy 名のリスト
 imitation_beta: 0.6
 imitation_kl_threshold: 1.0
 imitation_loss_scale: 1.0
 imitation_loss_threshold: 1.0
 deterministic: #方策が決定論的に振舞う確率
  g: 0.0 #学習用エピソード
  e: 0.0 #評価用エピソード
worker_args:
 server_address: ''
 num_parallel: 16 #worker の数
```

```
match_maker_args: #3.7項のMatchMakerに関する設定
 seed: 12345
 match_config:
  warm_up_episodes: 0
 #weight_pool: #weight_poolはtrain.pyで自動的に生成されるので記述不要
 #log_prefix: #log_prefixはtrain.pyで自動的に生成されるので記述不要
 #policy_config: #policy_configは上のpolicy_configがコピーされるので重複記述は不要
```

### 6.7.5 yamlで定義可能なニューラルネットワークのサンプル

ニューラルネットワークの構造をyaml上で定義できるHandyRL用NNモデルの簡易的なサンプルクラスを、sample/OriginalModelSample/OriginalModelSample/R4TorchNNSampleForHandyRL.pyに実装している。そのためのユーティリティとしてsample/OriginalModelSample/OriginalModelSample/GenericTorchModelUtil.pyに、与えられたdictから動的にtorch.nn.Moduleを生成するGenericLayerクラスを実装している。GenericLayerは、torch.nnのクラス名とコンストラクタ引数の組を並べることで、torch.nn.Sequentialに類似した形でニューラルネットワークを生成するものとなっている。

R4TorchNNSampleForHandyRLは、このGenericLayersを部分モデルとして使用し、HandyRLで使用可能なニューラルネットワークモデルとして組み立てたものである。

### 6.7.6 カスタムクラスの使用

本サンプルは、ニューラルネットワーク(本シミュレータにおけるPolicyに相当。HandyRLにおける呼称では"model")、MatchMaker、行動確率分布についてカスタムクラスを使用できるような機構を有している。root/sample/HandyRLSample/main.pyにおいて、

```
custom_classes={
  # models
  "R4TorchNNSampleForHandyRL": R4TorchNNSampleForHandyRL,
  "DummyInternalModel": DummyInternalModel,
  # match maker
  "BVRMatchMaker": BVRMatchMaker,
  "BVRMatchMonitor": BVRMatchMonitor,
  # action distribution class getter
  "actionDistributionClassGetter": getActionDistributionClass
}
```

となっているように、識別名とクラスオブジェクトを対応付けたdictを定義しておき、yamlで指定することによって、これらの中から選択して使用できるようになっている。

ニューラルネットワークのクラスはpolicy_configにおいてPolicyごとに"model_class"キーで指定し、MatchMakerのクラスはtrain_argsにおいて"match_maker_class"キー及び"match_monitor_class"キーで指定し、行動確率分布はpolicy_configにおいてPolicyごとに"model_config"キーとして与えるdict内で"actionDistributionClassGetter"キーで指定する(省略可)。

### 6.7.7 学習ログの構成

本サンプルで保存されるログは以下のような構成となる。なお、拡張子 pth で保存される Policy の重みは torch.nn.Module.state_dict()の返り値を torch.save(path)で保存したものであり、torch.load(path)で読み込んで torch.load_state_dict に与えることで復元可能なものである。

```
<save_dir> #yaml で指定した save_dir
  YYYYmmddHHMMSS #学習開始時のタイムスタンプで作成
    policies #Policy の重みに関するログ
      checkpoints #学習対象 Policy のエポックごとの重み
        <policy>-<epoch>.pth
        <policy>-latest.pth #最新の重み
      initial_weights #各 Policy の初期重み
        <policy>.pth
    matches #MatchMaker に関するログ
      matches_YYYYmmddHHMMSS.csv #MatchMaker が生成した全対戦結果のログ
    logs #SummaryWriter による Tensorboard ログ
      events.out.tfevents.～ #Tensorboard ログ
```

### 6.7.8 模倣学習の方法

本サンプルで模倣学習を取り入れたい場合は、yaml の policy_config に"Initial"と同じ形で

```
 Expert:
  multi_port: false
  active_limit: null
  is_internal: true
  populate: null
  rating_initial: 1500
  rating_fixed: true
  initial_weight: null
  model_class: DummyInternalModel
  model_config: {}
```

と追記し、BVRMatchMaker を改変して対戦カードとして"Learner"の代わりに"Expert"が生成されるようにし、更に yaml の train_args の policy_to_imitate を[]から["Expert"]に変更することで実現できるようになっている。

## 6.8 ray RLlib を用いて学習を行うサンプル

root/sample/raySample には、ray RLlib(を本シミュレータに合わせて改変及び機能追加したもの)を用いて行動判断モデルの学習を行うサンプルを同梱している。

### 6.8.1 元の ray RLlib に対する改変・機能追加の概要

本サンプルでは ray RLlib に以下のような改変・機能追加を行っている。

・MatchMaker を用いた行動判断モデルの動的選択に対応
・ExpertWrapper を用いた模倣学習に対応
・模倣学習(MARWIL)において、recurrent な Policy に対応

### 6.8.2 学習の実行方法

root/sample/raySample 上で、

```
python LearningSample.py APPO_sample_S.json
```

を実行すると、1 体で 1 機を操作する行動判断モデルについて、BVRMatchMaker を用いた Self-Play による学習が行われる。学習結果は./results/Single/APPO/run_YYYY-mm-dd-HH-MM-SS 以下に保存される。また、MARWIL による模倣学習を行う場合は、

```
python ExpertTrajectoryGatherer.py Gather_sample_S.json
```

を実行して教師役の軌跡データを記録した後、

```
python ImitationSample.py Imitate_sample_S.json
```

を実行することで、記録されたデータを用いた模倣学習が行われる。なお、模倣学習後の行動判断モデルの重みを APPO_sample_S.json の policy_config の initial_weight に指定することで、模倣学習後の状態から強化学習を開始することができる。

また、使用する json ファイルの末尾を_S から_M に変更すると、1 体で陣営全体を操作する行動判断モデルの学習となる。

### 6.8.3 学習済モデルの評価

学習済モデルは、./results/Single/APPO/run_YYYY-mm-dd-HH-MM-SS/policies/checkpoints 以下に保存されている。このモデルの評価は root/sample/MinimumEvaluation/evaluator.py を使用することで可能である。root/sample/MinimumEvaluation/candidates.json を開き、例えば

```
"test":{
  "userModuleID":"raySample01S",
  "args":{"weightPath":<学習済の.dat ファイルのフルパス>}
}
```

のように候補を追加し、root/sample/MinimumEvaluation 上で

```
python evaluator.py "test" "Rule-Fixed" -n 10 -v -l "eval_test.csv"
```

と実行すると、初期行動判断モデルとの対戦による学習済モデルの評価を行うことができる。

### 6.8.4 LearningSample.py 及び ImitationSample.py に与える json の書式

これらの学習用スクリプトは root/addons/MatchMaker/RayLeagueLearner.py に実装されている RayLeagueLearner クラスを使用しており、コマンドライン引数の json はその__init__の引数 config として使用される。config の記述方法は以下の通り。

```
config={
  "head_ip_address":Optional[str], #ray.init に渡す、ray の Head ノードの IP アドレス
                                   #(ポート番号を含む)。"auto"とすることで
                                   #自動検索、None とすることで新規クラスタとなる。
  "entrypoint_ip_address":Optional[str], #ray.init に渡す、このインスタンスが走る
                                         #ノードの IP アドレス(ポート番号を除く)。
                                         #省略時は"127.0.0.1"となる。
  "envCreator": Optional[Callable[[EnvContext],gym.Env]], #環境インスタンスを生成
                                #する関数。省略時は RayManager を生成する関数となる。
  "register_env_as": Optional[str], #登録する環境の名称。省略時は"ASRCAISim1"
  "as_singleagent": Optional[bool], #シングルエージェントとして環境及び Trainer を
                                   #扱うかどうか。デフォルトは False だが、MARWIL に
                                   #よる摸倣学習を使用する場合には True とする必要
                                   #がある。
  "seed":Optional[int], #乱数のシード値。
  "train_steps":int, #学習のステップ数。ray の Trainer.train を呼ぶ回数を指す。
  "refresh_interval":int, #学習中に Trainer の再読込を行う周期。worker を複数使用する
                          #際に発生すると報告されているメモリリーク対策として、
                          #Trainer インスタンスの再生成による緩和を試みるもの。
  "checkpoint_interval":int, #チェックポイントの生成周期。
  "save_dir":str, #ログの保存場所。ray の全ノードから共通してアクセスできるパス
                #でなければならない。(以下全てのパスについて同様)
  "experiment_name":str, #試行に付与する名称。save_dir 以下にこの名称のディレクトリ
                         #が生成され、各試行のログはその下に run_YYYY-mm-dd_HH-MM-SS
                         #というディレクトリとして保存される。
  "restore":Optional[str], #既存のチェックポイントから読み込む場合にチェック
                           #ポイントを含むログのパスを指定する。
                           #上記の run_YYYY-mm-dd_HH-MM-SS の階層まで指定する。
  "restore_checkpoint_number":Union[int,"latest",None], #既存のチェックポイントから
                           #読み込む場合、チェックポイントの番号を指定する。
                           #"latest"と指定することで当該ログ内の最新のチェック
                           #ポイントを自動で検索する。
  "trainer_common_config":{ #各 Trainer に与えるコンフィグの共通部分を記述する。
                            #デフォルト値は ray.rllib.trainer.py を始めとする
                            #各 Trainer クラスの定義とともに示されている。
                            #以下は主要な項目を例示する。
    "env_config":{ #環境に渡されるコンフィグを記述する。
                   #詳細は 2.5.1 項を参照のこと。
      "config":Union[dict,list[Union[dict,str]], #GymManager クラスで要求要
      "overrider": Optional[Callable[[dict,int,int],dict]], #GymManager クラスで要求
      "target": Union[Callable[[str],bool],str], #SinglizedEnv クラスで要求
```

```
    "policies": Dict[str,StandalonePolicy], #SinglizedEnv クラスで要求
    "policyMapper": Optional[Callable[[str],str]], #SinglizedEnv クラスで要求
    "exposeImitator": bool (False if omitted), #SinglizedEnv クラスで要求
    "runUntilAllDone": bool (True if omitted), #SinglizedEnv クラスで要求
  },
  "input": str, #模倣学習をする場合には軌跡データの場所を glob パターンで指定
  "model":dict,#NN モデルの構造を定義する。ray.rllib.models.catalog.py に設定項目
               #の一覧とデフォルト値が示されているものを使用してもよいし、項に
               #示す独自 NN のサンプルのようにカスタムモデルを使用してもよい。
  "manual_update_for_untrainable_policies":true #MatchMaker による重み読み書きに
               #対応するために、ray.rllib の IMPALA と APPO クラスを拡張して非学習
               #対象 Policy の自動重み配信を無効化するためのフラグ
},
"trainers":{ #Trainer 名をキーとした dict により、
             #生成する Trainer インスタンスを指定する。
  <Trainer's name>: {
    "trainer_type": str, #Trainer の種類。availableTrainers のキーから選択する。
    "config_overrider":Optional[dict[str,Any]], #trainer_common_config を上書き
                      #するための dict。主に"num_workers"を上書きすることになる。
    "policies_to_train":list[str], #この Trainer により学習を行うポリシーの一覧。
    "node_designation":Optional[str], #この Trainer インスタンスを生成する
                                      #ray ノードの IP アドレス。
                                      #省略時は ray により自動で選択される。
  },
  ...
},
"policies": { #Policy に関する設定。Policy 名をキーとした dict で与える。
              #MatchMaker の policy_config としても使用される。
  <Policy's name>: {
    "multi_port": bool, #1 体で 1 陣営分を動かすか否か。省略時は False。
    "active_limit": int, #保存された過去の重みを使用する数の上限を指定する。
    "is_internal": bool, #SimulationManager における Internal な Policy かどうか。
    "populate": None or { #重み保存条件の指定
      "firstPopulation": int, #初回の保存を行うエピソード数。
                              #0 以下の値を指定すると一切保存しない。
      "interval": int, #保存間隔。0 以下の値を指定すると一切保存しない。
      "on_start": bool, #開始時の初期重みで保存するかどうか。省略時は False。
      "reset": float, #重み保存時のリセット確率(0～1)
    },
    "rating_initial": float, #初期レーティング
    "rating_fixed": bool, #レーティングを固定するかどうか。
    "initial_weight": None or str, #初期重み(リセット時を含む)のパス
    "save_path_at_the_end": None or str, #学習終了時の重みを別途保存したい場合、
                                         #そのパスを記述する。
  },
  ...
```

```
  },
  "match_maker":{ #3.7項のMatchMakerに関する設定
    "seed": int,
    "match_config":{
      "warm_up_episodes: int
    },
    #weight_pool: #自動的に生成されるので記述不要
    #log_prefix: #自動的に生成されるので記述不要
    #policy_config: #上のpolicy_configがコピーされるので重複記述は不要
  }
}
```

また、RayLeagueLearnerクラスの__init__はavailableTrainersという引数も必要であり、これはLearningSample.py及びImitationSample.pyのスクリプト中で定義されているためコマンドライン引数に与えるjsonでは指定する必要はないが、以下の要領で記述する。

```
availableTrainers={
  <trainer_type>: {
    "trainer": Trainer, #使用するTrainerクラス。
    "tf": TFPolicy, #Tensorflowを使用する場合のPolicyクラス。
    "torch": TorchPolicy, #PyTorchを使用する場合のPolicyクラス。
    "internal": Optional[Policy], #actionをInternalに計算するAgentに対応する、
                                  #ダミーのPolicyクラス。省略時は
                                  #DummyInternalRayPolicyとなる。
  }
}
```

### 6.8.5 カスタムクラスの使用

RLlibにおいて独自のニューラルネットワーククラスを使用する場合は、root/sample/OriginalModelSample/OriginalModelSample/R4TorchNNSampleForRay.pyの末尾に記述されているとおり、

```
ModelCatalog.register_custom_model("R4TorchNNSampleForRay",R4TorchNNSampleForRay)
```

のように予めRLlibのModelCatalogにクラスを登録しておく必要がある。学習実行時は、Trainerに与えるコンフィグで"model"キーとして与えるdict内に、"custom_model"キーでクラス名を、"custom_model_config"キーでそのコンストラクタに与える引数を与えることで使用可能となる。

また、本サンプルにおいてMatchMaker及びMatchMonitorクラスの指定は、LearningSample.py及びImitationSample.pyにおいて、RayLeagueLearnerクラスのコンストラクタとしてクラスオブジェクトを与えることとしている。

### 6.8.6 ExpertTrajectoryGatherer.py に与える json の書式

模倣学習用軌跡データを生成するための ExpertTrajectoryGatherer.py には、以下のような形のコンフィグを与える必要がある。

```
{
  "seed":12345, #シード
  "env_config":[2.5.1.1項に示す context の"config"に相当する部分を直接記述する
    "./configs/R4_contest_mission_config.json"
    "./configs/R4_contest_learning_config_S.json"
    "./configs/R4_contest_asset_models.json"
    "./configs/R4_contest_agent_ruler_reward_models.json"
    {"Manager":{"ViewerType":"None"}}
  ],
  "num_workers":16, #並列数
  "num_episodes_per_worker":100, #1 ワーカーあたりのエピソード数
  "save_dir":"./experts/Single/" #軌跡の保存先
}
```

### 6.8.7 json で定義可能なニューラルネットワークのサンプル

ニューラルネットワークの構造を json 上で定義できる ray RLlib 用 NN モデルの簡易的なサンプルクラスを、6.7.5項に示した HandyRL 版と同様に、
sample/OriginalModelSample/OriginalModelSample/R4TorchNNSampleForRay.py に実装している。

### 6.8.8 RayLeagueLearner クラスのログの構成

RayLeagueLearner により保存されるログは以下のような構成となる。なお、拡張子 dat で保存される Policy の重みは、ray.rllib.policy.Policy.get_weights()で得られる np.array の list 又は dict をそのまま pickle で dump したものである。

```
<save_dir> #json で指定した save_dir
  <experiment_name> #json で指定した experiment_name
    run_YYYY-mm-dd_HH-MM-SS #学習開始時のタイムスタンプで作成
      policies #Policy の重みに関するログ
        checkpoints #チェックポイント
          checkpoint_<step>
            <policy>.dat
      matches #MatchMaker に関するログ
        matches_YYYYmmddHHMMSS.csv #MatchMaker が生成した全対戦結果のログ
        checkpoint_<step> #チェックポイント
          MatchMaker-<step>.dat
      trainers #Trainer に関するログ
        <trainer> #各 Trianer 名のディレクトリを ray のチェックポイント保存先とする
          checkpoint_<step>
            checkpoint-<step>
            <other files created by ray>
```

# 7 陣営ごとに Agent や Policy を隔離した状態で対戦させるための機能

本シミュレータは、陣営ごとに Agent や Policy の動作環境や設定ファイルをパッケージ化し、互いに隔離した状態で対戦させて評価を行うことを可能にするための機能を root/addons/AgentIsolation 及び root/sample/MinimumEvaluation に実装している。

## 7.1 Agent 及び Policy のパッケージ化の方法

Agent と Policy をパッケージ化する際は、あるディレクトリに__init__.py を格納して Python モジュールとしてインポート可能な状態とし、インポートにより以下の 4 種類の関数がロードされるような形で実装されていれば、細部の実装方法は問わないものとする。
(1) getUserAgentClass(args=None)･･･Agent クラスオブジェクトを返す関数
(2) getUserAgentModelConfig(args=None)･･･Agent モデルの Factory への登録用に modelConfig を表す json(dict)を返す関数
(3) isUserAgentSingleAsset(args=None)･･･Agent の種類(一つの Agent インスタンスで 1 機を操作するのか、陣営全体を操作するのか)を bool で返す関数(True が前者)
(4) getUserPolicy(args=None)･･･2.6 項で示す StandalonePolicy を返す関数

## 7.2 Agent 及び Policy の隔離

root/addons/AgentIsolation は、Agent クラスの実装を一切変更せずに、SimulationManager 本体と隔離された環境で動作可能にするための機能を提供している。実装の概要は以下の通り。
(1) SimulationManager 本体が動作する環境(center)と隔離された環境(edge)の間は socket 通信でデータをやりとりする。
(2) center 側では本来の Agent や StandalonePolicy クラスの代わりに同等のインターフェースを持つ AgentDelegator 及び PolicyDelegator を置き、edge 側には対になる AgentDelegatee と PolicyDelegatee を置く。
(3) AgentDelegatee は本来の Agent インスタンスを生成・保持する。
(4) AgentDelegatee は SimulationManager、Ruler、Asset について元のクラスと同一のインターフェースで最小限の中身を実装した専用クラスのインスタンスを生成し、AgentDelegator から提供されるデータを用いて内部変数を制御し、本来の Agent クラスが SimulationManager、Ruler、Asset から取得できることとされている情報を同一のソースコードでアクセスできるようにする。
(4) center 側でエピソードの実行中に AgentDelegator の各種関数(makeObs や deploy など)が呼び出されたとき、AgentDelegatee に対し必要なデータとともにコマンドを送信し、受け取った AgentDelegatee は本来の Agent インスタンスの対応する関数を呼び出し、その結果を AgentDelegator に送信する。
(5) PolicyDelegatee は本来の StandalonePolicy インスタンスを生成・保持する。
(6) center 側でエピソードの実行中に PolicyDelegator の各種関数(reset や step など)が呼び出されたとき、PolicyDelegatee に対し必要なデータとともにコマンドを送信し、受け取った PolicyDelegatee は本来の StandalonePolicy インスタンスの対応する関数を呼び出し、その結果を AgentDelegator に送信する。
(7) 本来の Agent に対する実装の制約をなくすために、Agent による読み書きが許されている全ての変数を全ての処理で送受信しているため、処理負荷は高くなっている。そのため、乱用は推奨しない。

## 7.3 パッケージ化された Agent 及び Policy の組を読み込んで対戦させるサンプル

本機能を用いて対戦を行うためには、まず root/sample/MinimumEvaluation 以下に 7.1 項の仕様に合うモジュールを定義し、root/sample/MinimumEvaluation/candidates.json に、対戦候補となる行動判断モデルの一覧を以下の形式で記述しておく。args として例えば Policy の重みファイルのパスを指定したり、modelConfig を書き換えたりすることによって、単一のモジュールで複数の行動判断モデルを表現できるようになる。

```
"name":{
    "userModuleID": str, #モジュール名(ディレクトリ名)
    "args": str, # ７．１項の４種類の関数に渡される引数
    "logName": str, # 対戦結果ログを出力する際の行動判断モデルの識別名
}
```

Agent と Policy を隔離しないで動かす場合は、root/sample/MinimumEvaluation/evaluator.py を使用する。evaluator.py のコマンドライン引数は以下の通りである。

```
python evaluator.py [-h] [-n NUM_EPISODES] [-l LOG_DIR] [v VISUALIZE] blue red

positional arguments:
  blue  Blue 側の行動判断モデルの名称
  red   Red 側の行動判断モデルの名称

optional arguments:
  -h, --help                                    ArgumentParser で自動生成されるヘルプ
  -n NUM_EPISODES, --num_episodes NUM_EPISODES  対戦回数。省略時は 1 回
  -l LOG_DIR, --log_dir LOG_DIR                 対戦ログの保存先。省略時は保存しない
  -v, --visualize                               可視化する場合に指定する。
```

隔離して動かす場合は、root/sample/MinimumEvaluation/sep_main.py と sep_edge.py を使用する。root/sample/MinimumEvaluation/sep_config.json に Blue と Red のそれぞれに対応する IP アドレス、ポート名を記述しておき、Blue を動かす環境において `python sep_edge.py blue` を、Red を動かす環境において `python sep_edge.py red` をそれぞれ実行した後に、上記の evaluator.py と同様のコマンドライン引数で sep_main.py を実行することで隔離環境における対戦が実行される。

## 7.4 Agent 及び Policy のパッケージ化のサンプル

Agent 及び Policy のパッケージ化を行うサンプルとして root/sample/MinimumEvaluation には表 7.４－１に示す５種類を格納している。

表 7.４－１　Agent 及び Policy のパッケージ化を行うサンプルの一覧

| モジュール名 | 概要 |
|---|---|
| RuleBased | ルールベースの初期行動判断モデル |
| HandyRLSample01S | ・1 体で 1 機を操作する Agent モデル<br>・root/sample/HandyRLSample に同梱されたサンプルで学習した Policy |
| HandyRLSample01M | ・1 体で陣営全体を操作する Agent モデル<br>・root/sample/HandyRLSample に同梱されたサンプルで学習した Policy |
| rayRLSample01S | ・1 体で 1 機を操作する Agent モデル<br>・root/sample/raySample に同梱されたサンプルで学習した Policy |
| rayRLSample01M | ・1 体で陣営全体を操作する Agent モデル<br>・root/sample/raySample に同梱されたサンプルで学習した Policy |