空戦 AI チャレンジ

内容説明 Detail 版

主催 防衛装備庁

委託 Nishika 株式会社 Nishika

目次

1 本シミュレータの構成・使用方法

本シミュレータの構成及び使用方法について説明する。

1.1 ディレクトリ構成

simulator.zip として配布される本シミュレータのディレクトリ構成は以下の通り。

```
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
 │ └rayUtility：ray RLlib を用いて学習を行うための拡張機能
 ├ASRCAISim1：本シミュレータの Python モジュールを構成する一式(一部はビルド時に生成)
 ├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
 │ ├config：サンプル用のコンフィグファイル
 │ ├MinimumEvaluation：各エージェント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
 │ └Standard：初期行動判断モデルを対戦相手とし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
```

1.2 環境の構築

1.2.1 環境構築手順

1.2.1.1 手法 1 Ubuntu ベースのコンテナ環境を構築

simulator.zip を解凍し、解凍した simulator の root に Dockerfile を置き、以下を実行する。

```
docker build .
```

この環境が提出されたエージェント同士の対戦環境にもなっている。

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問題なければ成功である。

```
cd sample/Standard
python FirstSample.py
```

上記コンテナ環境は GUI 非対応であるため、後に示す、戦闘画面の可視化を行いたい場合は手法 2 を推奨する。

1.2.1.2 手法 2 ホスト OS 上で直接環境構築（Linux, macOS） ※推奨

Linux の場合は py37_linux_whl.zip、macOS の場合は py37_mac_whl.zip を解凍し、2 つの whl ファイルが存在することを確認する。

Python3.7 がインストールされた仮想環境等にて、以下を実行する。

```
pip install "ray[default, tune, rllib]==1.9.1"
pip install ASRCAISim1-1.0.0-py3-none-any.whl
pip install OriginalModelSample-1.0.0-py3-none-any.whl
```

nlopt をインストールする。

```
curl -sSL https://github.com/stevengj/nlopt/archive/v2.6.2.tar.gz | tar xz
cd nlopt-2.6.2 && cmake . && make && make install
```

nlopt の共有オブジェクトの場所を指定する。Linux の場合は、
```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```
macOS の場合は、
```
export LIBRARY_PATH=$LIBRARY_PATH:/usr/local/lib
```
により指定できる。
simulator.zip を解凍し、以下動作確認用サンプルを実行し問題なければ成功である。
```
cd sample/Standard
python FirstSample.py
```

1.2.1.3 手法 2 ホスト OS 上で直接環境構築（Windows, MSVC） ※非推奨

py37_win_whl.zip を解凍し、2 つの whl ファイルが存在することを確認する。
Python3.7 がインストールされた仮想環境等にて、以下を実行する。
```
pip install "ray[default, tune, rllib]==1.9.1"
pip install ASRCAISim1-1.0.0-py3-windows-any.whl
pip install OriginalModelSample-1.0.0-py3-windows-any.whl
```

nlopt をインストールする。
```
mkdir c:¥simulator;cd simulator
curl -sSL -o nlopt-2.6.2.tar.gz https://github.com/stevengj/nlopt/archive/v2.6.2.tar.gz
tar -xzf nlopt-2.6.2.tar.gz
cd nlopt-2.6.2
mkdir build;cd build
cmake ..
```
Visual Studio を起動して、nlopt.sln を開いて Release モードでビルドする。

システム環境変数 Path に、以下の nlopt.dll へのパスと ASRCAISim1 の Core.dll へのパス（自身のパスに読み替えること）を追加する。
C:¥simulator¥nlopt-2.6.2¥build¥Release
C:¥Users¥...¥lib¥site-packages¥ASRCAISim1

simulator.zip を解凍し、以下動作確認用サンプルを実行し問題なければ成功である。
```
cd sample/Standard
python FirstSample.py
```

1.2.1.4 手法 2 ホスト OS 上で直接環境構築（Windows, MSYS） ※非推奨

py37_win_msys_whl.zip を解凍し、2 つの whl ファイルが存在することを確認する。
Python3.7 がインストールされた仮想環境等にて、以下を実行する。
```
pip install numpy
pip install "ray[default, tune, rllib]==1.9.1"
pip install ASRCAISim1-1.0.0-py3-none-any.whl
pip install OriginalModelSample-1.0.0-py3-none-any.whl
```

nlopt をインストールする。
```
mkdir c:¥simulator;cd simulator
curl -sSL -o nlopt-2.6.2.tar.gz https://github.com/stevengj/nlopt/archive/v2.6.2.tar.gz
tar -xzf nlopt-2.6.2.tar.gz
cd nlopt-2.6.2
mkdir build;cd build
cmake .. -G "MSYS Makefiles"
make; make install
```

システム環境変数 Path に、以下の nlopt.dll へのパスと ASRCAISim1 の Core.dll へのパス（自身のパスに読み替えること）を追加する。
C:¥simulator¥nlopt-2.6.2¥build¥Release
C:¥Users¥...¥lib¥site-packages¥ASRCAISim1

simulator.zip を解凍し、以下動作確認用サンプルを実行し問題なければ成功。
```
cd sample/Standard
python FirstSample.py
```

1.2.1.5 ソースコードからのビルドを行いたい場合

ソースコードからのビルドを行いたい場合は、１．２．３項に示す依存ライブラリをインストールした上で、setup.py や CMakeLists.txt のある root 直下のディレクトリにおいて
```
pip install .
cd sample/OriginalModelSample
pip install .
```
とすることによりビルド及びインストールが可能である。
なお、Windows で MSYS を用いる場合は、
```
pip install . --install-option=”--MSYS”
cd sample/OriginalModelSample
pip install . --install-option=”--MSYS”
```
のようにオプションを追加する必要がある。

1.2.2 動作確認環境(主要な項目のみ)

本シミュレータは、以下の環境で動作確認を行っている。
- OS：Ubuntu 20.04 LTS
- Python 3.7.12
- Tensorflow：2.6.2
- PyTorch：1.9.1+cpu
- ray：1.9.1
- gcc：11.2.0(>=9)
- Eigen：3.4.0 (>=3.3.9)
- pybind11：2.8.0 (>=2.6.2)
- nlohmann’s json：3.9.1
- pybind11_json：0.2.11(を改変)
- NLopt：2.6.2
- Magic Enum C++：0.7.3
- Boost：1.65.1
- CMake：3.21 (>=3.12)

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

1.2.3 C++依存ライブラリのダウンロード

本シミュレータは以下の C++ライブラリを使用している。whl からインストールする場合は(1)、ソースからビルドする場合は(1)～(7)のダウンロード及びインストールが必要となる。

(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) 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。

1.2.4 深層学習用フレームワークのインストール

本シミュレータは純粋な OpenAI Gym 環境ではなく、マルチエージェントのシミュレーションを前提としていることから ray RLlib を強化学習フレームワークとして想定しており、RLlib の MultiAgentEnv クラスにインターフェースを合わせている。１．２．３項における本シミュレータのインストール作業において、依存ライブラリとして RLlib を指定しているため RLlib は自動的にインストールされるが、深層学習フレームワークについては自動的にはインストールされない。RLlib は Tensorflow または PyTorch のいずれかを使用することを前提に実装されており、本シミュレータの学

習サンプルは RLlib による学習を実施するため、どちらもインストールされていない場合は別途インストールすることを推奨する。

2 本シミュレータの機能・処理の流れ

本シミュレータの機能・処理の流れについて説明する。

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、例えば強化学習フレームワーク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。

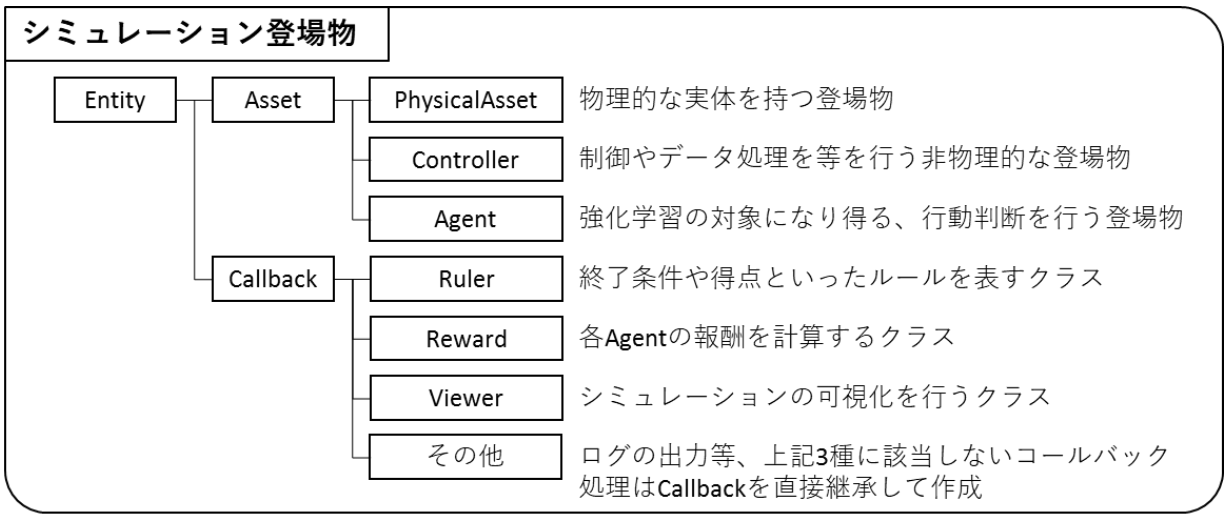

図 ２．２－１　シミュレーション登場物の分類

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 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、「観測してよい情報」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メンバ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「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「子」となる PhysicalAsset と Controller 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 observable 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 6 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) onStepBegin…step 関数の開始時(＝外部から与えられた Action を Agent が受け取った直後)
(3) onInnerStepBegin…各 tick の開始時(＝Asset の control の前)
(4) onInnerStepEnd…各 tick の終了時(＝Asset の perceive の後)

(5) onStepEnd…step 関数の終了時（＝step 関数の戻り値生成の前または後※1）
(6) onEpisodeEnd…各エピソードの終了時（step 関数の戻り値生成後）

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 observable 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 observable 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。２．２．１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 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。ただし、現時点で実装済のクラス及びモデルは全て1[tick]ごとに処理するものとして実装されている。

また、２．２．２に挙げた Callback の 6 種類の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。

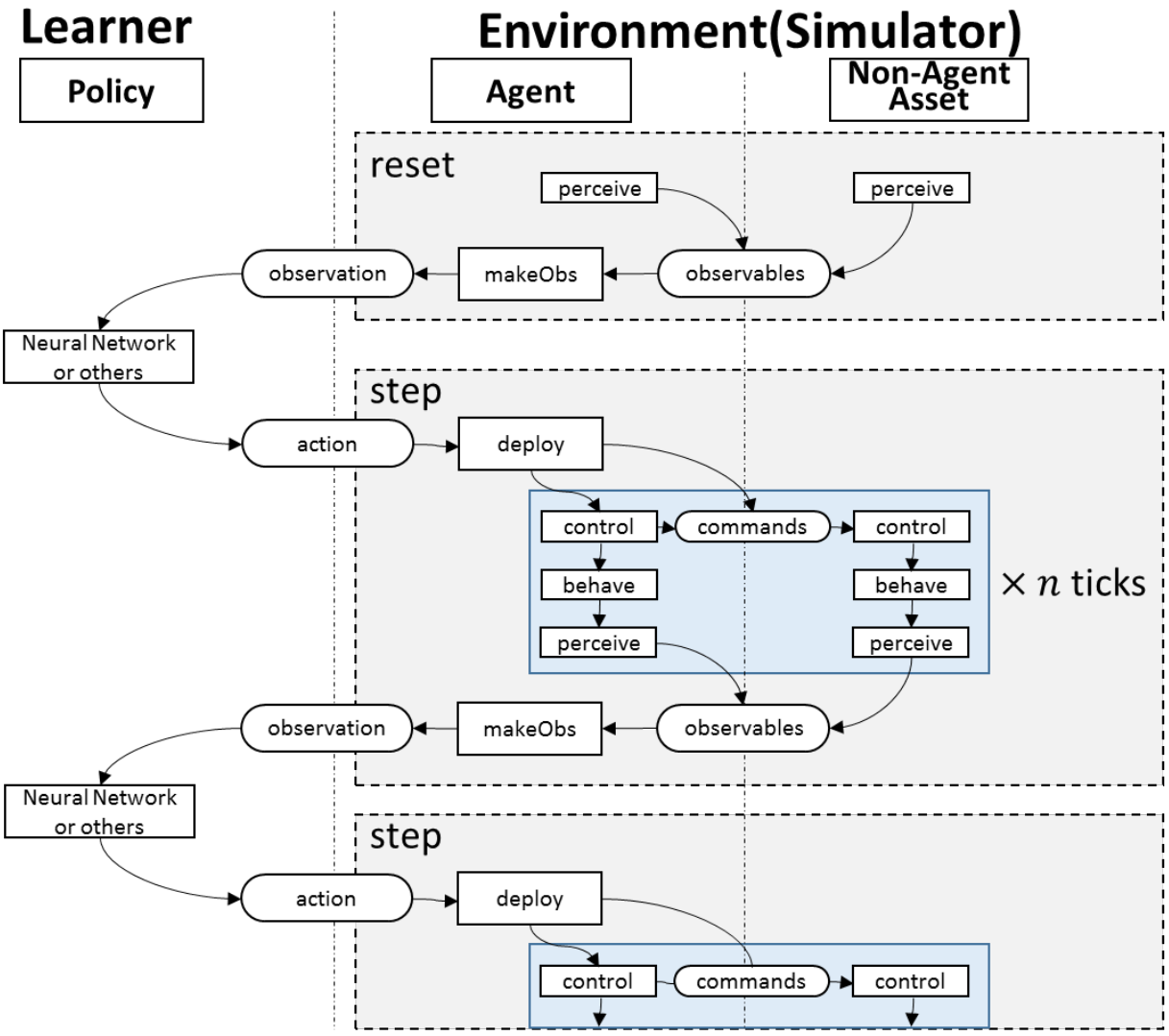

図 ２．３－１　シミュレーション中の処理の流れ(行動判断に関するもののみ)

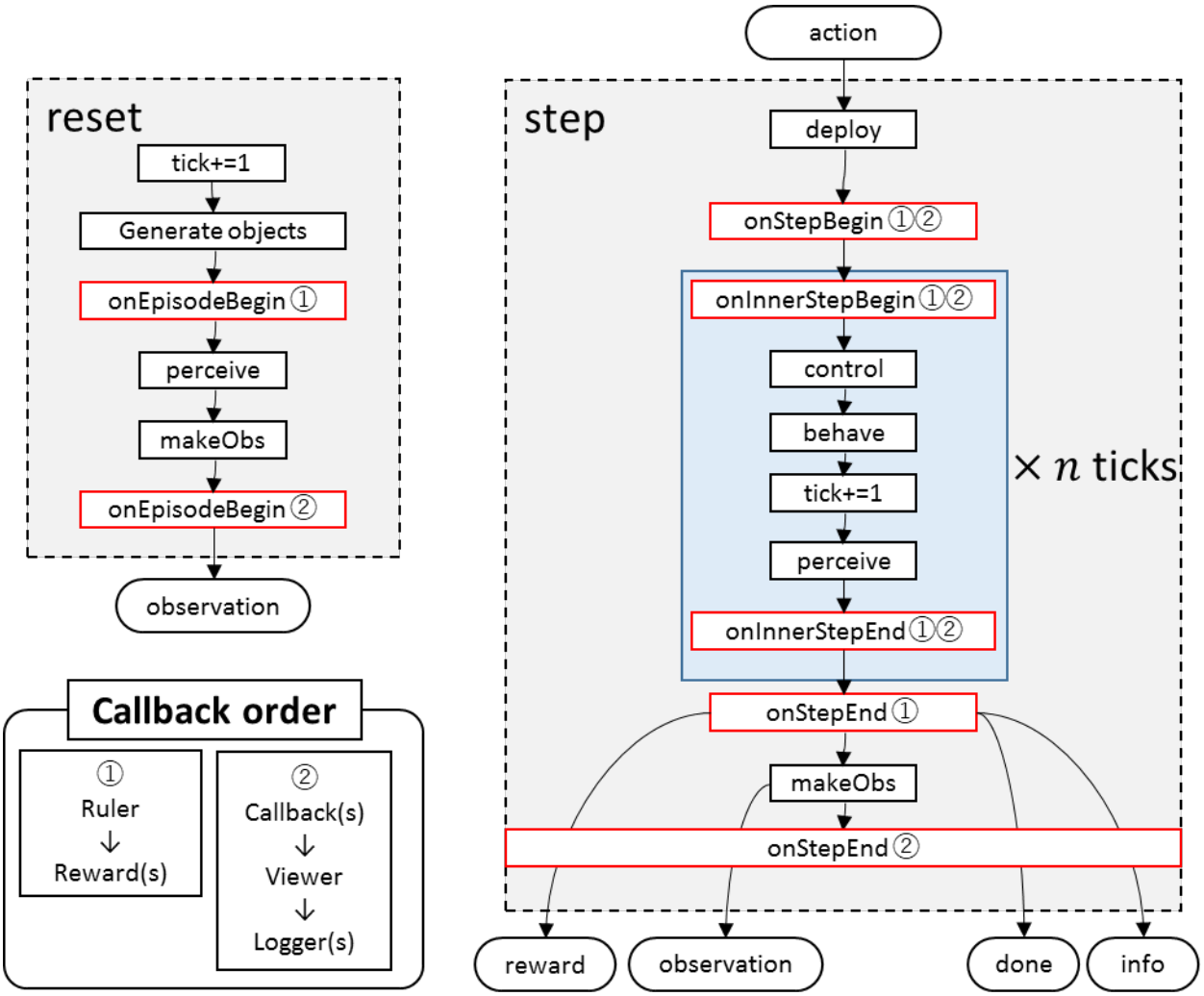

図　2．3－2　コールバックの処理タイミング

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シミュレーションの処理の流れ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を共通化するために、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 RayManager

RLlib の MultiAgentEnv を継承したマルチエージェント環境としてのラッパークラスであり、通常はこのクラスを RLlib に登録して使用するものである。通常の gym.Env との大きな違いは、reset 関数と step 関数の引数と戻り値が各 Agent 名をキーとした dict になっていることである。また、コンストラクタ引数 context は dict を継承した EnvContext 型とされているため、SimulationManager のコンストラクタが要求する４つの引数は EnvContext のコンストラクタにおいて以下のように記述する。

```
context = EnvContext(
    {
        "config":config,
        "overrider":overrider
    },
    worker_index,
    vector_index
)
```

2.5.1.2 GymManager

ray を使用せずに本シミュレータを使用したいという場合を想定し、コンストラクタ引数 context を ray の EnvContext 型でなく通常の dict 型とした GymManager クラスも実装している。コンストラクタに渡す dict には、EnvContext がメンバとして有していた worker_index と vector_index を同名のキーで指定する必要がある。また、他のインターフェースは RayManager クラスと同一であるため、Observation 等は dict 型である。

2.5.1.3 RaySinglizedEnv, SinglizedEnv

登場する Agent のうち指定した一つのみの Observation, Action を入出力するように制限したシングルエージェント環境としてのラッパークラスであり、摸倣学習時に対象の Agent を限定するために用い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 RayManager, 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 RaySinglizedEnv, SinglizedEnv の context に追加すべき項目

キー名	型	概要
target	Union[Callable[[str,bool],str]]	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	Optional[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

2.5.1.4 RaySimpleEvaluator, 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クラスである。

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”Auto”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 クラ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た overrider によって実現する。使い方としては、複数の SimulationManager を並列化して並列実行する際にインスタンスごとに異なる設定としたい場合に worker_index と vector_index の値に応じて書き換えるために用いることを想定している。これにより、RLlib の Trainer に与える”env_config”は単一の config で済むようになる。

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、前エピソードの終了時までに、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 に対して適用する 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いる。

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２．５．１．３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には環境側が主となってポリシーの Action 計算処理を呼び足す必要がある。

そのため、本シミュレータでは StandalonePolicy クラスとしてポリシーの共通インターフェースを定義している。Standalone 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、 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.proxy として扱われる。

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だが 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うる。

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　getValueFromJson の引数と関数末尾の文字に示す通り 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> 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> const ValueType& 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”, ” value”:<value>}	<value>で指定した値
一様分布	{“type”:” uniform”, ” dtype”:<dtype>, ” low”:<low>, ” high”:<high>}	<dtype>は” int”または” float”とし、省略した場合は” float”とみなす。 <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。 “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。 また、要素数１の次元は削除される。
正規分布	{“type”:” normal”, ” mean”:<mean>, ” 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”, ” candidates”:<candidates>, ” 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。 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。 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に対応している。この機能は３．６．１項～３．６．３項の３つのクラスによって実現されている。また、厳密にはシミュレータ側の機能ではないが、RLlib において模倣学習アルゴリズムとして実装されている MARWIL(及びその派生としての Behavior Cloning)が RNN に対応していないため、オリジナルの MARWIL を改変して RNN に対応したものを RNNMARWIL として実装している。

3.6.1 ExpertWrapper

ExpertWrapper クラスは Agent クラスの派生クラスであり、内部に expert(模倣される側)と imitator(模倣する側)の 2 つの Agent モデルを保持する。外部に対しては expert の Observation と Action を入出力する。「親」となる PhysicalAsset を含む他の Asset に対しては、expert と imitator のいずれか指定した一方が生成した commands と observables を適用する。また、imitator の action は expert の出力結果を最も類似するものに変換する関数をユーザーが定義することで計算する。詳細は４．５．５項に示す。

3.6.2 ExpertTrajectoryWriter

ExpertWrapper クラスは、それ単体では模倣する側の行動軌跡を記録しない。そのため、Callback クラスを継承した Experttrajectory クラスを用いて行動軌跡を記録する。行動軌跡の形式は RLlib の SmapleBatch に準じており、json ファイルとして出力される。出力先は、modelConfig のキー”prefix”で指定したパスの末尾に ExpertWrapper の instanceConfig にキー”identifier”で与えた名称を付加したフォルダとなり、ExpertWrapper のインスタンスごと異なるフォルダとなる。

3.6.3 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。詳細は４．５．６項に示す。

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)
```

(2)Python からの static メンバへの登録

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

(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 の config に”Factory”キーでモデルの情報を記述する。

4.1.3 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)	現在の方位をx軸正方向として$x-y$平面を水平にとった座標系。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 relPtoB relHtoP relPtoH	const Eigen::Vector3d &v	相対位置ベクトル v の変換 原点の平行移動を無視する
absBtoP absPtoB absBtoH absHtoB	const Eigen::Vector3d &v	絶対位置ベクトル v の変換 原点の平行移動を考慮する
velBtoP velPtoB	const Eigen::Vector3d &v const Eigen::Vector3d &r	元座標系で位置 r にある点の速度 v の変換 原点まわりの回転を考慮する r を省略した場合は r=0 とする
omegaBtoP omegaPtoB	const Eigen::Vector3d &v	角速度ベクトルの変換

4.3 Track3D 及び Track2D

本シミュレータにおいて、センサによって捉えた航跡は、3 次元航跡を表す Track3D クラスと 2 次元航跡を表す Track2D クラスによって表現される。

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、誤相関は発生しないものとして扱う。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、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、誤相関は発生しないものとして扱う。2 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 の構成要素(json)

キー名(インデントは階層を表す)	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。	○
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 team group 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。	○

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。 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。 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。 例えば Fighter の場合、以下の 3 項目を記述する。 pos･･･初期位置(3-dim array(double)) vel･･･初期速度(double) heading･･･真北を 0、時計回りを正とした初期方位(double) 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	模倣学習用 Agent のうち、親 Asset との相互作用に expert の出力を用いるもの。
ExpertI	模倣学習用 Agent のうち、親 Asset との相互作用に imitator 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	type	“Internal”	×
	model	モデル名(string)	×
External	type	“External”	×
	model	モデル名(string)	×
	policy	ポリシー名(string)。省略時は”Auto”となる。	○
ExpertE/ExpertI	type	“ExpertE”または”ExpertI”	×
	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	模倣される側ののポリシー名(string)。省略時は”Auto”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	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のを 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、例えば本シミュレータの初期行動判断モデルは戦闘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 行動意図表現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 とあ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

模倣学習をする際は、模倣用の教師データの作成と、教師データを使用した学習の 2 段階で実施する必要がある。教師データの作成時は、SimulationManager の config において、表 4.5-2 に示した”ExpertE”または”ExpertI”を指定し、必要事項を記述するとともに、Callback または Logger として ExpertTrajectoryWriter クラスのインスタンスを生成するように設定しておく必要がある。また、imitator 側の Agent クラスにおいて、expert の decision と commands を自身の action_space で最も近い Action に変換する以下のメンバ関数をオーバーライドしておく必要がある。

```
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
```

一方で学習時には、本番の強化学習と同様、”External”を指定し模倣する側の Agent モデルを直接記述する必要がある。

4.5.6 MultiPortCombiner

3.6.3 項で触れた 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。

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	×
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 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 の名称又は名称のリスト。 ”All”とすると場に存在する者すべてを対象とする。	×

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。 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 名は正規表現による指定が可能である。	○
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。 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 名は正規表現による指定が可能である。	○

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 “names”：array(string) “element”：object “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。 <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。 <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) “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” “names”：array(string) “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。 <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 “index”：int または指定なし “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。 <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。 <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。 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”AssetConfigDispatcher”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(1／2)

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。
○	○	○	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。 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。
	○	○	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 のテラブルを返す。
	○	○	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。
		○	getManualConfig()	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。

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

対象クラス			関数名	概要
Agent	PhysicalAsset /Controller	Callback		
○	○		generateAgent(agentConfig, agentName, 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(baseName, modelName, 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(baseName, className, modelConfig, 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(bufferName, 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(name participants, 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(baseName, modelName, instanceConfig)	モデル名を指定して、Asset 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(baseName, className, modelConfig, instanceConfig)	クラス名を指定して、Asset 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。
		○	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。
		○	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(managerReplacer, 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、本シミュレータによる空対空目視外戦闘場面の表現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
	AircraftRadar	Fighter	戦闘機搭載センサ(レーダ)
	MWS		戦闘機搭載センサ(MWS)
	Propulsion		戦闘機用ジェットエンジン (CoordinatedFighter のみ)
	Missile		誘導弾
	MissileSensor	Missile	誘導弾搭載センサ(シーカ)
Controller	SensorDataSharer	Fighter	戦闘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	戦闘機の行動判断エージェント

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。

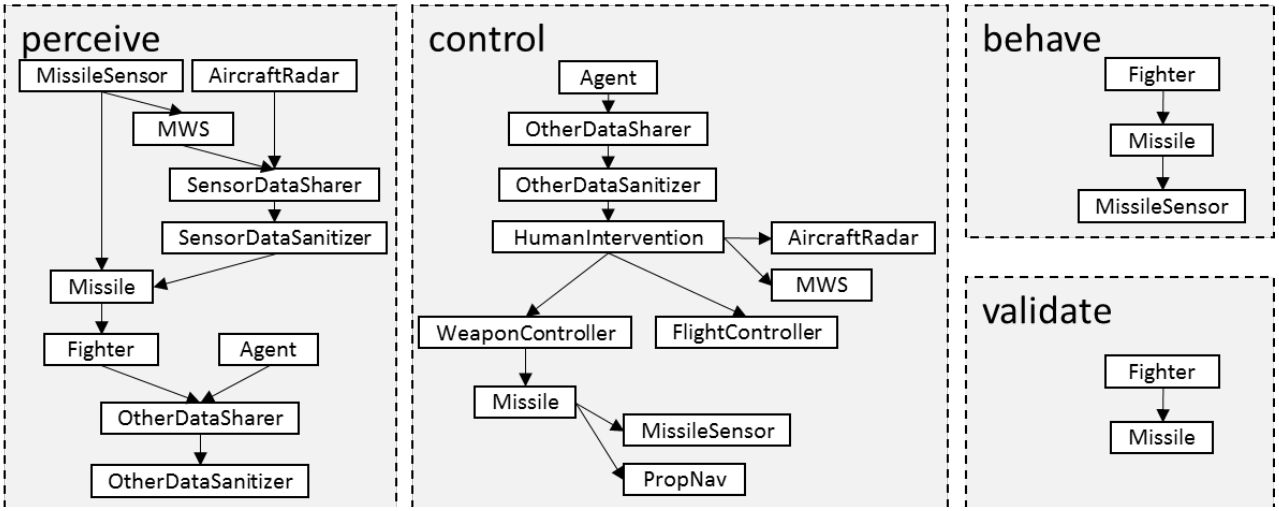

図 4.7－1　各処理における Asset の処理順序の依存関係

4.7.3 戦闘機モデルの種類

現バージョンにおいて戦闘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 CoordinatedFighter クラスと MassPointFighter クラスの 2 種類を実装している。

CoordinatedFighter クラスは航空機の運動を横滑り角=0 として簡略化したモデルを実装したクラスであり、基本的にはこのクラスを戦闘機モデルとして用いる。

MassPointFighter クラスは、Agent にとって学習の難易度を下げるために、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として実装している。空対空戦闘に関する行動判断の学習をはじめて試行する際は、まずは難易度の低い MassPointFighter クラスを使用したモデルを使用して「飛びたいように飛べる」状況で行うことも想定している。

4.7.4 Agent が入出力すべき observables と commands

4.7.4.1 CoordinatedFighter クラスを用いる場合

Agent が CoordinatedFighter クラス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、CoordinatedFighter クラスが Agent に対して出力することを要求する commands は、CoordinatedFighter::setFlightControllerMode(mode)で指定した飛行制御モードに応じて異なり、mode=”direct”とした場合の形式は表 4.7-3、mode=”fromDirAndVel”とした場合の形式は表 4.7-4 のとおりである。

表 4.7-2 CoorinatedFighter クラスの 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
rollMax	double	ロール角速度の上限値[rad/s]
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。
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。 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 observable であり、各 parent の fullName をキーとした object として生成される。
fighter	object	味方戦闘機の observables。自身を含む各機の名称をキーとして各機の observable 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。 ・/shared 以下全て ・/sensor/track ・/sensor/trackSource

表 4.7-3 飛行制御モードが”direct”の場合の commands の形式

キー名(インデントは階層を表す)	型	概要	備考
motion	object	運動に関する値	
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
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	

表 4.7-4 飛行制御モードが”fromDirAndVel”の場合の commands の形式

キー名(インデントは階層を表す)	型	概要	備考
motion	object	運動に関する値	
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。)	
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	

４.７.４.２ MassPointFighterFighter クラスを用いる場合

Agent が MassPointFighter クラスから受け取ることのできる observables の形式は表 ４．７−５のとおりである。また、MassPointFighter クラスが Agent に対して出力することを要求する commands は表４．７−６のとおりである。

表 ４．７−５　MassPointFighter クラスの observables の形式(CoordinatedFighter との差分のみ)

キー名(インデントは階層を表す)	型	概要
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
vMin	double	速度の下限値[m/s]
vMax	double	速度の上限値[m/s]
aMin	double	加速度の下限値[m/s^2]
aMax	double	加速度の上限値[m/s^2]
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]
その他		CoordinateFighter の場合と同様

表 ４．７−６　MassPointFighter クラスが Agent 側に要求する commands の形式

キー名(インデントは階層を表す)	型	概要	備考
motion	object	運動に関する値	
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	[aMin, aMax]を[−1,1]に正規化した目標加速度	いずれか一つを指定
throttle	double	[aMin, aMax]を[0,1]に正規化した目標加速度	
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	

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い場合があるため、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 で最も近い Action を計算する convertActionFromAnother 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、Factory にモデル登録ができるようにする。
(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 としてスーパークラスで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() #RulerAccessor 型で得られる
  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):
  #1tick 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)6 種類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、各 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)6 種類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。
(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)
    #スーパークラスにおいて 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)
    #スーパークラスにおいて 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、スーパー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。
  #スーパー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、スーパー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()):
        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 基本サンプルの概要

本項では、root/sample/Standard に格納されている基本的なサンプルの使用方法の概要についてまとめる。

6.1 基本的な使用方法

6.1.1 サンプルモジュールのビルド及びインストール

本シミュレータのコア部分をインストールした後、root/sample/OriginalModelSample 上で

```
pip install .
```

により Agent 及び Reward のサンプルをインストールする。

6.1.2 OpenAI gym 環境としてのインターフェース確認

root/sample/Standard 上で

```
python SecondSample.py
```

を実行すると、サンプルの Agent モデル 2 種(Blue 側は 1 機ずつ行動、Red 側は 2 機分一纏めに行動)どうしのランダム行動による対戦が実行され、Observation や Action の定義例が確認できる。

なお、このサンプルはコンフィグを変えた後等の Agent,Policy 割当の確認の際にも用いることができる。

6.1.3 ray RLlib を用いた学習

root/sample/Standard に同梱しているサンプルでは、ルールベースの初期行動判断モデルを教師役として模倣学習を行うフェーズと、模倣学習済のモデルを読み込んで初期行動判断モデルを対戦相手とした強化学習を行うフェーズに分かれている。なお、以下のサンプルにおいては深層学習フレームワークに PyTorch を使用している。

6.1.3.1 模倣学習用の教師データの取得

root/sample/Standard 上で

```
python ExpertTrajectoryGatherer.py Gather2v2.json
```

を実行すると、./experts/2vs2/以下にルールベースの行動に相当する教師データが保存される。

6.1.3.2 模倣学習の実施

教師データの取得後、root/sample/Standard 上で

```
python ImitationSample.py Imitate2v2.json
```

を実行すると、./experts/2vs2/以下の教師データを入力として模倣学習が行われ、
完了時には./policies/Imitated2v2.dat に重みが保存される。また、実行中のチェックポイントは./results/Imitate2v2/run_YYYY-mm-dd-HH-MM-SS/以下に保存される。

6.1.3.3 強化学習の実施

模倣学習の実施後、root/sample/Standard 上で

```
python LearningSample.py IMPALA2v2.json
```

を実行すると、./policies/Imitated2v2.dat を初期重みとして IMPALA による強化学習が行われ、
完了時には./policies/IMPALA2v2.dat に重みが保存される。また、実行中のチェックポイントは./results/IMPALA2v2/run_YYYY-mm-dd-HH-MM-SS/以下に保存される。
なお、初期重みを使用しない場合は json 中のパスを指定している部分を null とすればよい。
学習済モデルの評価
模倣学習の実施後は root/sample/Standard 上で

```
python ImitatedTest.py Imitate2v2.json ./policies/Imitated2v2.dat
```

強化学習の実施後は root/sample/Standard 上で

```
python LearnedTest.py IMPALA2v2.json ./policies/IMPALA2v2.dat
```

を実行すると、学習済モデルを読み込みルールベースモデルと対戦させ、その結果を記録することができる。

6.1.3.4 学習サンプルのバリエーション

上記のサンプルは、1 体のエージェントにつき 1 機を動かすモデルを学習するものとなっているが、root/sample/Standard にはこれ以外に以下の 2 系統のサンプルが含まれており、コマンドライン引数で指定する json ファイルを変更することで使用可能である。

(1) 1 体のエージェントで 2 機両方を動かすモデルを学習するサンプル

Gather2v2.json、Imitate2v2.json、IMPALA2v2.json をそれぞれ Gather2v2_Cent.json、Imitate2v2_Cent.json、IMPALA2v2_Cent.json と置き換えればよい。ただし、学習済モデルの評価を行う際は、ImitatedTest.py 及び LearnedTest.py 中の

```
configs=["../config/BVR2v2_rand.json","../config/Learned2v2.json"]
```

となっている部分を

```
configs=["../config/BVR2v2_rand.json","../config/Learned2v2_Cent.json"]
```

とする必要がある。

(2) 機体モデルを更に簡略化した質点モデル(MassPointFighter)を用いた環境で学習するサンプル

Gather2v2.json、Imitate2v2.json、IMPALA2v2.json をそれぞれ Gather2v2_MP.json、Imitate2v2_MP.json、IMPALA2v2_MP.json と置き換えればよい。ただし、学習済モデルの評価を行う際は、ImitatedTest.py 及び LearnedTest.py 中の

```
configs=["../config/BVR2v2_rand.json","../config/Learned2v2.json"]
```

となっている部分を

```
configs=["../config/BVR2v2_rand_MP.json","../config/Learned2v2.json","./config_MP.json"]
```

とする必要がある。

6.2 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 6．2-1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 6．2-1　必須クラスの一覧

種類	クラス名	概要	実装言語
Ruler	R3BVRRuler01	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	簡略化した戦闘機モデルの運動モデル以外の共通部分を実装した基底クラス	
	CoorinatedFighter	簡略化した戦闘機の運動モデルを実装したもの	
	MassPointFighter	CoordinatedFighter よりさらに運動モデルを簡略化し、速度、角速度ベクトルを直接操作できるようにしたもの	
	Missile	簡略化した誘導弾モデルを実装したもの	
	AircraftRadar	簡略化した戦闘機センサ(レーダ)モデルを実装したもの	
	MWS	簡略化した戦闘機センサ(MWS)モデルを実装したもの	
	MissileSensor	簡略化した誘導弾センサモデルを実装したもの	
	Propulsion	CoorinatedFighter クラスで使用する、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	戦闘機モデル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	戦闘機モデル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送信側を簡易的に実装したもの	
	Fighter:: OtherDataSanitizer	戦闘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	
	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	R3InitialFighterAgent01	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	

6.3 Rewardの実装例

報酬を計算するRewardについては、表 6.3-1に示す4種類をサンプルとして実装している。いずれも陣営単位の報酬を計算するTeamRewardの派生クラスである。

表 6.3-1 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 root/src/Reward.cpp	C++
R3BVRBasicReward01	R3BBVRRuler01 の得点計算と同様の観点で少し異なる報酬計算を行う例	root/include/ R3BVRBasicReward01.h root/src/ R3BVRBasicReward01.cpp	C++
R3RewardSample01	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与える例	root/sample/OriginalModelSample/include/R3RewardSample01.h root/sample/OriginalModelSample/src/R3RewardSample01.cpp	C++
R3PyRewardSample01		root/sample/OriginalModelSample/R3PyRewardSample01.py	Python
WinLoseReward	勝敗のみに基づく報酬を与える例	root/sample/OriginalModelSample/WinLoseReward.h root/sample/OriginalModelSample/WinLoseReward.cpp	C++

6.4 Agentの実装例

深層強化学習を行うためのAgentの実装例はに示す通り、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 6.4-1 Agentの実装例

クラス名	概要	パス	実装言語
R3AgentSample01	1体のAgentにつき1機分の行動判断を行う例	root/sample/OriginalModelSample/include/R3AgentSample01.h root/sample/OriginalModelSample/src/R3AgentSample01.cpp	C++
R3PyAgentSample01		root/sample/OriginalModelSample/R3PyAgentSample01.py	Python
R3AgentSample02	1体のAgent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/R3AgentSample02.h root/sample/OriginalModelSample/src/R3AgentSample02.cpp	C++
R3PyAgentSample02		root/sample/OriginalModelSample/R3PyAgentSample02.py	Python

6.5 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。図 6.5－1 に示すように、戦域を真上から見た図平面図と、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。

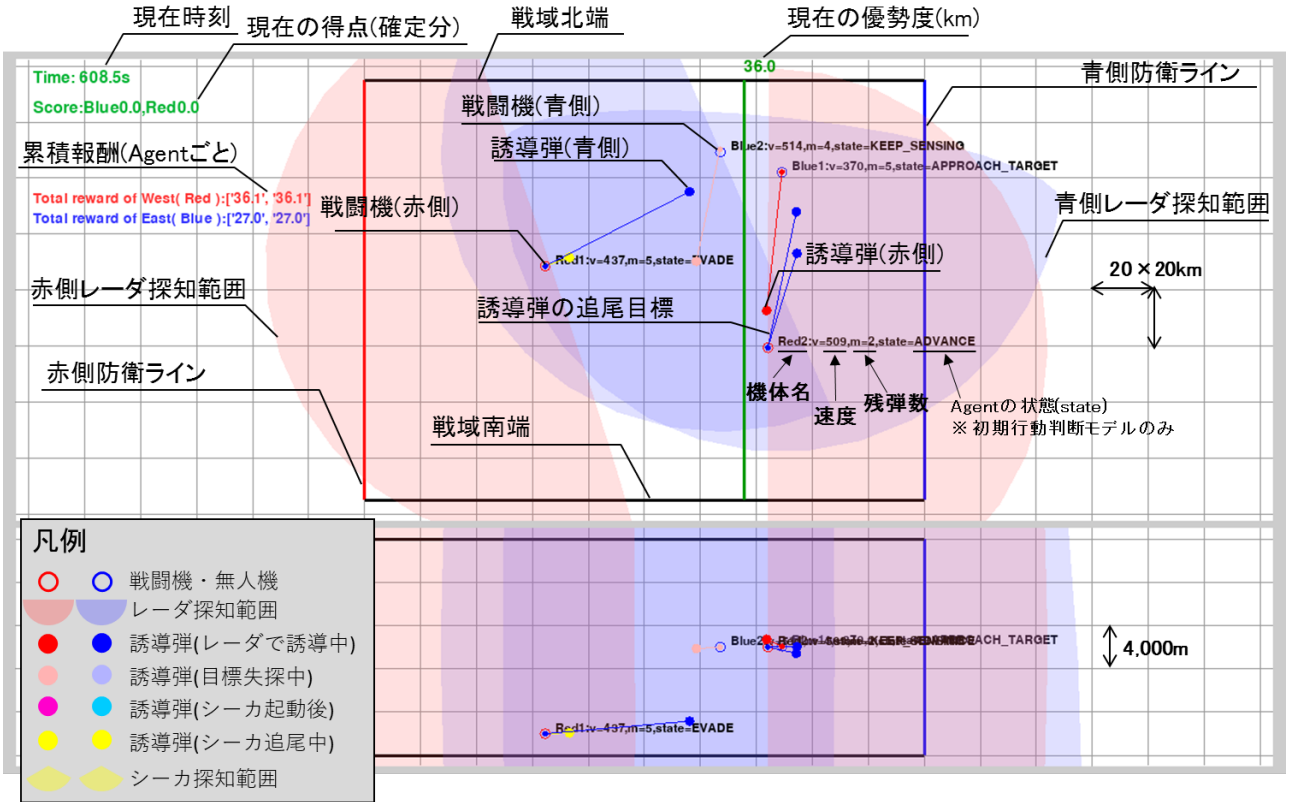

図 6.5－1　GodView の画面表示例(上側が平面図、下側が側面図)

また、表 6.6-1 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。

6.6 Logger の実装例

戦闘結果のログ保存を行う Logger の例として、３．６．２項の ExpertTrajectoryWriter の他に、表６．６－１に示す 3 種類を実装している。

表 ６．６－１　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。また、本クラスは複数の SimulationManager インスタンスを並列に実行した場合でも単一のログとして生成されるように、ray ライブラリの機能を用いて実装している。	root/ASRCAISim1/logger/MultiEpisodeLogger.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

6.7 追加モジュールとしての Agent、Reward の実装

root/sample/OriginalModelSample 以下に実装した Reward と Agent のサンプル(６．３項と６．４項の一部)は、本シミュレータのコア部分と一体ではなく、独立した Python モジュールとして呼び出し可能にする形式で実装している。コア部分と同様に、root/sample/OriginalModelSample 上で

```
pip install .
```

によりビルド及びインストールが可能である。

6.8 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、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。root/sample/Standardでは表 6.8-1に示すようにconfigを分割している。

表 6.8-1 SimulationManager及びFactory追加モデルのconfig例

分類	パス	概要
戦闘場面の指定	root/sample/config/BVR2v2.json	固定された初期条件でRedとBlueの2対2の対戦場面を設定する例
	root/smple/config/BVR2v2_rand.json	ランダムな初期条件でRedとBlueの2対2の対戦場面を設定する例
	root/sample/config/BVR2v2_rand_MP.json	戦闘機モデルをMassPointFighterに置き換えてランダムな初期条件でRedとBlueの2対2の対戦場面を設定する例
Agent構成、報酬体系の指定	root/sample/config/Initial2v2.json	RedとBlue両方のAgentに初期行動判断モデルを割り当てる例
	root/sample/config/ForExpertTraj2v2.json	Redの Agentに初期行動判断モデルを、Blueの Agentに ExpertWrapperを割り当てる例
	root/sample/config/ForLearning2v2.json	Redの Agentに初期行動判断モデルを、Blueの Agentに表 6.4-1に示す Agentクラスのモデルを割り当てる例
	root/sample/config/Learned2v2.json	
	root/sample/config/*_Cent.json	対応する上記の例の Agentを、1体の Agentで編隊全機の行動判断を行うモデルに置き換えた場合の例
追加モデルの登録	root/sample/Standard/config_MP.json	MassPointFighterを用いた戦闘を行うためにFactoryにモデルを追加するためのconfig例
	root/sample/OriginalModelSample/OriginalModelSample/config/R3SampleConfig01.json	6.7項の OriginalModelSampleモジュールとしてFactoryに追加するモデルのconfig
その他の指定	root/sample/Standard/IMPALA2v2.json 等	例えば左記ファイルではSimulationManagerのコンフィグとしてViewerType、seed、Loggersを直接指定している。

6.8.1 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 BVR2v2_rand.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":0.1, #1tick の時間ステップ
      "agentInterval":10 #gym.Env として 1 回の step 関数で何 tick 進めるか
    },
    "Ruler":"R3BVRRuler01", #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":"broadcast","number":2,
        "names":["Blue1","Blue2"], #Asset 名
        "element":{"type":"alias","alias":"BlueFighter"}, #基本は後述の"BlueFighter"を使用
        "overrider":[#overrider により設定を修正していく
          {"type":"broadcast","number":2,
            "element":{"type":"alias","alias":"OnEastLine"}#初期配置を東側にする
          },
          {"type":"broadcast","number":2,
            "element":{
              "type":"direct","value":{
                "instanceConfig":{"datalinkName":"BlueDatalink"}#BlueDatalink に加入
              }
            }
          },
          {"type":"group","order":"shuffled","elements":[#Agent に関する設定
            #AgentConfigDispatcher の"BlueAgents"を参照するよう指定。
            #BlueAgents の中身は Agent 側の config で指定するものとする。
            #ランダム要素があっても陣営単位で正しく指定されるように instance と index を指定
            {"type":"direct","value":{
              "Agent":{"type":"alias","alias":"BlueAgents","instance":"g1","index":0}
            }},
            {"type":"direct","value":{
              "Agent":{"type":"alias","alias":"BlueAgents","instance":"g1","index":1}
            }}
          ]}
        ]
      },
      "Red":{ #Red 側の Asset に関する設定。記述要領は Blue 側と同様。
        "type":"broadcast","number":2,
        "names":["Red1","Red2"],
        "element":{"type":"alias","alias":"RedFighter"},
        "overrider":[
          {"type":"broadcast","number":2,
            "element":{"type":"alias","alias":"OnWestLine"}},
          {"type":"broadcast","number":2,
            "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}
            }}
          ]}
        ]
      }
    },
    "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":{
      "BlueFighter":{#Blue 側の戦闘機モデルを指定する本体
        "type":"PhysicalAsset","model":"R3Fighter00",#Factory のグループ名とモデル名に相当
        "Agent":{"type":"alias","alias":"Initial"}
      },
      "RedFighter":{#Red 側の戦闘機モデルを指定する本体
        "type":"PhysicalAsset","model":"R3Fighter00",#Factory のグループ名とモデル名に相当
        "Agent":{"type":"alias","alias":"Initial"}
      },
      "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,-14000],"high":[20000,-100000,-6000]
            },
            "vel":{"type":"uniform","low":200,"high":300},
            "heading":{"type":"uniform","low":60,"high":120}
          }
        }
      },
      "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,-14000],"high":[20000,100000,-6000]
            },
            "vel":{"type":"uniform","low":200,"high":300},
            "heading":{"type":"uniform","low":240,"high":300}
          }
        }
      }
    }
  }
}
```

6.8.2 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 ForLearning2v2.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":"R3BVRBasicReward01-1","target":"All"},
      {"model":"R3BVRBasicReward01-2","target":"All"},
      {"model":"R3RewardSample01","target":"All"}
    ],
    "AgentConfigDispatcher":{
      "Initial_e":{#初期行動判断モデルを指定する本体
        "type":"Internal","model":"R3Initial"
      },
      "Learner_e":{#学習用 Agent モデルとポリシー名を指定する本体
        "type":"External","model":"R3AgentSample01(2vs2)_single","policy":"Learner"
      },
      "Learner":{"type":"group","order":"fixed",
        #陣営一つ分の学習用 Agent 群として記述。
        #学習に SimpleRayLearner を用いる場合で、Agent の type が Internal でない場合は、
        #ここの名称("Learner")と、対応するポリシー名が一致していなければならない。
        "elements":[
          {"type":"alias","alias":"Learner_e"},
          {"type":"alias","alias":"Learner_e"}
        ]
      },
      "Initial":{"type":"group","order":"fixed",#陣営一つ分の初期行動判断モデル群として記述
        "elements":[
          {"type":"alias","alias":"Initial_e"},
          {"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。
        #ただし、学習に SimpleRayLearner を用いる場合で、
        #Agent の type が Internal でない場合は、
        #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 名を個別に指定
            {"type":"direct","value":{"name":"Blue2"}}
          ]}
        ]
      },
      "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"}}
          ]}
        ]
      }
    }
  }
}
```

また、模倣学習用教師データを取得する際に AgenConfigDispatcher で ExpertWrapper を指定する方法は以下の通りである。

```
{
  "Manager":{
    "AgentConfigDispatcher":{
      "Expert1":{
        "type":"ExpertI", #模倣する側の出力で行動する場合は"ExpertI"
                          #模倣される側の出力で行動する場合は"ExpertE"
        "imitatorModel":"R3AgentSample01(2vs2)_single", #模倣する側の Agent モデル
        "expertModel":"R3Initial", #模倣される側の Agent モデル
        "identifier":"Traj1" #軌跡を保存する際の識別名。
                             #同じ識別名の Agent の軌跡は同じファイルに記録される。
                             #ただし、基本的には Agent インスタンスごとに異なる
                             #識別名とすることを推奨する。
      }
    }
  }
}
```

6.9 シミュレーション・学習用のスクリプト及びコマンドライン引数

root/sample/Standard に含まれるシミュレーション・学習用のスクリプト及びコマンドライン引数の組合せの一覧は表 6．9－1 の通りである。これらは root/sample/Standard に移動してから Python スクリプトを実行することを前提としている。なお、これらのサンプルは深層学習フレームワークに PyTorch を使用している。

表 6．9－1　スクリプトとコマンドライン引数の組合せ一覧

概要	スクリプトファイル	コマンドライン引数	備考
初期行動判断モデルどうしの対戦	FirstSample.py	なし	基準
	FirstSample_MP.py	なし	MassPointFighter
模倣学習用データの生成	ExpertTrajectoryGatherer.py	Gather2v2.json	基準
		Gather2v2_Cent.json	1 体で全機分
		Gather2v2_MP.json	MassPointFighter
MARWIL による模倣学習の実施	ImitationSample.py	Imitator2v2.json	基準
		Imitator2v2_Cent.json	1 体で全機分
		Imitator2v2_MP.json	MassPointFighter
IMPALA による深層強化学習の実施	LearningSample.py	IMPALA2v2.json	基準
		IMPALA2v2_Cent.json	1 体で全機分
		IMPALA2v2_MP.json	MassPointFighter
模倣後モデルと初期行動判断モデルの対戦	ImitatedTest.py	Imitator2v2.json ./policies/Imitated2v2.dat	基準
		Imitator2v2_Cent.json ./policies/Imitated2v2_Cent.dat	1 体で全機分
		Imitator2v2_MP.json ./policies/Imitated2v2_MP.dat	MassPointFighter
学習済モデルと初期行動判断モデルの対戦	LearnedTest.py	IMPALA2v2.json ./policies/Learned2v2.dat	基準
		IMPALA2v2_Cent.json ./policies/Learned2v2_Cent.dat	1 体で全機分
		IMPALA2v2_MP.json ./policies/Learned2v2_Cent.dat	MassPointFighter

6.9.1 ExpertTrajectoryGatherer.py に与える json の書式

```
{
  "seed":12345, #シード
  "env_config":[ #2.5.1.1項に示すEnvContextの"config"に相当する部分を直接記述
    "../config/BVR2v2_rand.json",
    "../config/ForExpertTraj2v2.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/2vs2/" #軌跡の保存先(のプレフィックス)
}
```

6.9.2 ImitationSample.py 及び Learning に与える json の書式

これらの学習用スクリプトは root/sample/Standard/SimpleRayLearner.py に実装されている SimpleRayLearner 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={
  "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。ray 1.8.0時点
                          #において発生するメモリリーク対策としてインスタンス再生成
                          #によるメモリ解放を試みるもの。
  "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。
                    #詳細は各 Env クラスの定義を参照のこと。
      "config":Union[dict,list[Union[dict,str]], #GymManager クラスで要
      "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
    },
    "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。
  },
  "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。
    },
    ...
  },
  "policies": { #Policy に関する設定。Policy 名をキーとした dict で与える。
    <Policy's name>: {
      "initial_weight": None or str, #初期重み(リセット時も含む)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。
    },
    ...
  }
}
```

また、SimpleRayLearner クラスの__init__は availableTrainers という引数も必要であり、これは ImitationSample.py 及び LearningSample.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。
  }
}
```

6.9.2.1 SimpleRayLearner クラスのログの構成

SimpleRayLearner により保存されるログは

```
logdir="run_"+datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
experiment_log_path=os.path.join(config["save_dir"],
                                 config["experiment_name"],logdir)
```

として、以下のような構成となる。なお、拡張子 dat で保存される学習モデル(重み)は、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。

```
experiment_log_path
  policies #ポリシーの重みに関するログ
    initial_weight #初期重み
      <policy>.dat
    checkpoints #チェックポイント
      checkpoint_<step>
        <policy>.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 の組を一意に識別可能な名称をディレクトリ名とし、__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()･･･Agent クラスオブジェクトを返す関数
(2) getUserAgentModelConfig()･･･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()･･･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()･･･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、処理負荷は高めとなっている。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

Agent 及び Policy を隔離しない場合のサンプルを root/sample/MinimumEvaluation/evaluation.py に実装しており、run 関数の引数として Agent・Policy パッケージの識別名を二つ与えることで、それらを読み込んで互いに対戦させることができる。

また、Agent 及び Policy を隔離する場合は、sep_config.json に Blue と Red のそれぞれに対応する edge の IP アドレス、ポート名、Agent・Policy パッケージの識別名を記述しておき、Blue を動かす

edge 環境において python sep_edge.py blue を、Red を動かす edge 環境において python sep_edge.py red をそれぞれ実行した後に、center 環境において python sep_main.py を実行することで対戦させることが可能となっている。また、本サンプルでは戦闘場面は表 ６．８－１に示す BVR2v2_rand.json を root/sample/MinimumEvaluation に格納して読み込むこととしているため、異なる場面の対戦を実行したい場合は適切に変更されたい。

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 表 ７．４－１に示す 4 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

識別名	概要	4 種類の関数が返す値	
RuleBased	初期行動判断モデル	getUserAgentClass	R3InitialFighterAgent01
		getUserAgentModelConfig	config.json を読み込み
		isUserAgentSingleAsset	True
		getUserPolicy	None を返すダミーPolicy
User001	1 体で 1 機を操作する Agent モデル 及び ランダム行動をとる Policy	getUserAgentClass	R3PyAgentSample01
		getUserAgentModelConfig	config.json を読み込み
		isUserAgentSingleAsset	True
		getUserPolicy	ランダム行動を返すダミーPolicy
User002	1 体で陣営全体を操作する Agent モデル 及び ランダム行動をとる Policy	getUserAgentClass	R3PyAgentSample02
		getUserAgentModelConfig	config.json を読み込み
		isUserAgentSingleAsset	False
		getUserPolicy	ランダム行動を返すダミーPolicy
User003	1 体で 1 機を操作する Agent モデル 及び ray で学習された Policy	getUserAgentClass	R3PyAgentSample01
		getUserAgentModelConfig	config.json を読み込み
		isUserAgentSingleAsset	True
		getUserPolicy	trainer_config.json 及び weights.dat を読み込んで生成された VTraceTorchPolicy を実体として持つ StandaloneRayPolic