S4 Simulation System

Version 6.2

psim 言語リファレンスマニュアル

Copyright © 2008-2022 NTT DATA Mathematical Systems, Inc. All Rights Reserved.

目 次

第 1 章　はじめに **10**

1.1　準備 10

第 2 章　シミュレーション記述 **11**

2.1　プロセス 11

2.1.1　プロセスの定義 11

2.1.2　プロセスの登録 12

2.1.3　シミュレーションの開始 14

2.1.4　シミュレーションの停止 14

2.2　待ち受け 14

2.2.1　待ち受けの実行 14

2.3　時間 15

2.3.1　シミュレーション時間の取得 15

2.3.2　時間待ち受け 15

2.4　イベント 15

2.4.1　イベントの定義 15

2.4.2　イベント待ち受け 16

2.4.3　イベント発火 16

2.4.4　イベントの状態取得 16

2.4.5　イベントの時系列モニター 17

2.5　ファシリティ 18

2.5.1　ファシリティの定義 18

2.5.2　ファシリティの取得 19

2.5.3　ファシリティの解放 19

2.5.4　ファシリティの利用可能性確認 20

2.5.5　ファシリティの解放ハンドラ 20

2.5.6　ファシリティの状態取得 20

2.5.7　ファシリティの時系列モニター 21

2.6　段取替ファシリティ 22

2.6.1　ファシリティの定義 22

2.6.2　段取替ファシリティの取得 23

2.6.3　段取替ファシリティの状態取得 24

2.6.4　段取替ファシリティの時系列モニター 25

2.7 タンク 26
2.7.1 タンクの定義 26
2.7.2 タンクへの追加 27
2.7.3 タンクからの取得 27
2.7.4 タンクの状態取得 27
2.7.5 タンクの時系列モニター 28
2.8 ストア 29
2.8.1 ストアの定義 29
2.8.2 ストアへの追加 30
2.8.3 ストアからの取得 31
2.8.4 ストアの状態取得 32
2.8.5 ストアの時系列モニター 34
2.9 複合待ち受け 34
2.9.1 OR 待ち受け 35
2.9.2 AND 待ち受け 35
2.9.3 逐次待ち受け 36
2.9.4 ガード式のキャンセル 36
2.9.5 結合優先度 36
2.10 複合イベント 37
2.10.1 OR 複合イベント 37
2.10.2 AND 複合イベント 38
2.10.3 逐次複合イベント 39
2.10.4 結合優先度 40
2.11 複合ファシリティ 40
2.11.1 OR 複合ファシリティ 41
2.11.2 AND 複合ファシリティ 42
2.11.3 逐次複合ファシリティ 43
2.11.4 結合優先度 45
2.11.5 複合ファシリティの解放 45
2.12 複合タンク 45
2.12.1 OR 複合タンク 45
2.12.2 AND 複合タンク 47
2.12.3 逐次複合タンク 49
2.12.4 結合優先度 51
2.13 複合ストア 52
2.13.1 OR 複合ストア 52
2.13.2 AND 複合ストア 55
2.13.3 逐次複合ストア 57
2.13.4 結合優先度 60

2.14 連続シミュレーション 61
2.14.1 連続変数の定義 62
2.14.2 補助変数の定義 62
2.14.3 連続シミュレーションの実行 64
2.14.4 連続シミュレーションの状態待ち受け 68
2.14.5 連続シミュレーションの結果 69
2.14.6 連続変数の時系列モニター 69
2.14.7 補助変数の時系列モニター 70
2.14.8 常微分方程式ソルバー 70
2.15 条件式 71
2.15.1 恒真 71
2.15.2 恒偽 71
2.15.3 条件判定 71
2.15.4 遅延条件判定 71
2.16 プロセス制御 72
2.16.1 プロセスの状態取得 73
2.16.2 プロセスの終了待ち受け 73
2.16.3 プロセスの一時停止待ち受け 74
2.16.4 プロセスの再開待ち受け 74
2.16.5 プロセスの終了 74
2.16.6 プロセスの一時停止 75
2.16.7 プロセスの再開 75
2.16.8 自プロセスの終了 75
2.16.9 自プロセスの一時停止 76
2.16.10 プロセスの終了ハンドラの登録 76
2.16.11 プロセスの一時停止ハンドラの登録 76
2.16.12 プロセスの再開ハンドラの登録 76
2.16.13 自プロセスの継続オブジェクトの取得 77
2.16.14 プロセス制御の例 78
2.17 サブプロセス 79
2.17.1 サブプロセスの定義 79
2.17.2 サブプロセスの実行 80
2.17.3 複合サブプロセス 81
2.18 状態遷移 83
2.18.1 状態遷移の実行 83
2.18.2 サブ状態 85
2.18.3 並行状態 87
2.18.4 履歴 88
2.18.5 ガード式の末尾 91

2.19 アクション . . . 91
2.19.1 アクションの挿入 . . . 91
2.20 記録 . . . 92
2.20.1 値の記録 . . . 92
2.20.2 時間の記録 . . . 92
2.21 ハンドラ . . . 93
2.21.1 ハンドラの登録 . . . 93
2.22 待ち受け結果 . . . 94
2.22.1 待ち受け結果の取得 . . . 94
2.22.2 待ち受けの成立検査 . . . 94
2.22.3 発火値の取得 . . . 95
2.22.4 ガード式の発火時間 . . . 96
2.22.5 ガード式の成立順番 . . . 96
2.22.6 単一ガード式の場合の発火値の取得 . . . 97
2.22.7 全てのファシリティロックの解放 . . . 98
2.23 直前の待ち受け結果 . . . 100
2.23.1 直前の待ち受け結果の取得 . . . 100
2.24 キュー . . . 100
2.24.1 FIFO キュー . . . 101
2.24.2 PriorityQ . . . 101
2.24.3 Stack . . . 101
2.25 デバッグ . . . 101

第 3 章 エージェントベースシミュレーション記述 104
3.1 概要 . . . 104
3.2 環境 . . . 104
3.2.1 環境の基底クラス . . . 104
3.2.2 グラフ構造を表す環境の基底クラス . . . 106
3.2.3 カスタムグラフ . . . 109
3.2.4 GNM グラフ . . . 110
3.2.5 GNP グラフ . . . 110
3.2.6 Barabasi Albert グラフ . . . 110
3.2.7 Powerlaw Cluster グラフ . . . 111
3.2.8 完全グラフ . . . 111
3.2.9 格子グラフ . . . 111
3.2.10 GEXF フォーマットグラフ . . . 112
3.2.11 GraphML フォーマットグラフ . . . 113
3.2.12 ユークリッド 2D . . . 113
3.2.13 ソーシャルフォースモデル環境 . . . 113
3.2.14 ネットワークシミュレーション環境 . . . 137

3.3 エージェント集合 140
3.3.1 エージェント集合の基底クラス 140
3.3.2 同期エージェント集合 143
3.3.3 非同期エージェント集合 143
3.3.4 ソーシャルフォースモデルエージェント集合 143
3.3.5 ネットワークエージェント集合 149
3.3.6 粒子フィルタエージェント集合 150
3.4 エージェント 152
3.4.1 エージェントの基底クラス 152
3.4.2 同期エージェント 154
3.4.3 非同期エージェント 155
3.4.4 ソーシャルフォースモデルエージェント 155
3.4.5 ネットワークエージェント 163
3.4.6 粒子フィルタエージェント 165
3.5 エージェントパネル 166
3.6 エージェントスクリーン 167
3.6.1 共通 170

第 4 章 エージェントプラナー 172
4.1 概要 172
4.2 基本概念 172
4.3 同期型/非同期型プロセスの動作比較 173
4.4 API 175
4.4.1 プラナー 175
4.4.2 プロセス 176
4.4.3 タスク 177
4.4.4 状態 177
4.5 動作例 179
4.5.1 問題設定 179
4.5.2 状態遷移図 180
4.5.3 コード例 181
4.5.4 ステップ実行 182
4.5.5 サブ状態 183
4.6 代表的なパターン 186
4.6.1 基本状態遷移 186
4.6.2 自分への状態遷移 186
4.6.3 タスクの終了 186
4.6.4 サブ状態 187
4.6.5 サブ状態が完了したら、別の状態に遷移 187
4.6.6 より複雑な手順 (サブ状態のリスト) に遷移 188

4.6.7 同一プロセス内の優先度によるタスク実行 188
4.6.8 別プロセスによるタスク実行 188

第 5 章 フローベース・フレームワーク 190
5.1 シミュレーター 190
5.1.1 シミュレーター 190
5.1.2 一括実行シミュレーター 191
5.1.3 シミュレーションパラメータ 191
5.2 フロー部品 191
5.2.1 部品 191
5.2.2 フローアイテム 192
5.2.3 フローアイテムリスト 193
5.2.4 リンク 193
5.2.5 出力ポート 193
5.2.6 入力ポート 194
5.3 セレクタ 194
5.3.1 ランダム (一様) 195
5.3.2 ランダム (重み付き) 195
5.3.3 ラウンドロビン 195
5.3.4 固定 195
5.3.5 最短キュー 195
5.3.6 強化学習 196

第 6 章 データ 197
6.1 モニターの操作 197
6.1.1 モニターの作成 197
6.1.2 モニターへの観測結果の記録 198
6.1.3 時系列モニターの作成 198
6.1.4 時系列モニターへの観測結果の記録 199
6.1.5 観測列の型 199
6.1.6 モニターからの観測列の取得 199
6.1.7 時系列モニターからの観測列の取得 200
6.1.8 時系列モニターからの時刻列の取得 200
6.1.9 列指定子 200
6.1.10 行指定子 201
6.1.11 モニターからの部分抽出 (列指定) 202
6.1.12 時系列モニターからの部分抽出 (列指定) 202
6.1.13 モニターからの部分抽出 (行、列指定) 202
6.1.14 時系列モニターからの部分抽出 (行、列指定) 203
6.1.15 モニターへの部分更新 (列指定) 203

6.1.16 時系列モニターへの部分更新 (列指定) 204
6.1.17 モニターへの部分更新 (行、列指定) 204
6.1.18 時系列モニターへの部分更新 (行、列指定) 205
6.1.19 モニターからの部分削除 (列指定) 205
6.1.20 時系列モニターからの部分削除 (列指定) 205
6.1.21 モニターからの部分削除 (行指定) 206
6.1.22 時系列モニターからの部分削除 (行指定) 206
6.1.23 モニターの保存 206
6.1.24 時系列モニターの保存 206
6.1.25 モニターのその他の操作 206
6.2 モニターの列の操作 207
6.2.1 観測列の作成 208
6.2.2 観測列の基本統計量の取得 208
6.2.3 観測列の基本統計量のサマリ 211
6.2.4 観測列のヒストグラム 212
6.2.5 時系列観測列からの時刻列の取得 213
6.2.6 観測列の 2 項演算 213
6.2.7 時系列観測列の 2 項演算 214
6.2.8 観測列の累算演算 215
6.2.9 時系列観測列の累算演算 215
6.2.10 観測列の単項演算 216
6.2.11 時系列観測列の単項演算 216
6.2.12 観測列の数値演算 216
6.2.13 時系列観測列の数値演算 217
6.2.14 観測列の比較演算 217
6.2.15 時系列観測列の比較演算 218
6.2.16 観測列の部分抽出 218
6.2.17 時系列観測列の部分抽出 218
6.2.18 観測列の部分更新 218
6.2.19 時系列観測列の部分更新 219
6.2.20 観測列の部分削除 219
6.2.21 時系列観測列の部分削除 219
6.2.22 観測列のその他の操作 219
6.2.23 時系列観測列のその他の操作 223

第 7 章 乱数 227

7.1 種 227
7.2 乱数生成器 227
7.2.1 固定 228
7.2.2 指数分布 228

7.2.3 正規分布 . . . 228
7.2.4 対数正規分布 . . . 229
7.2.5 一様分布 . . . 229
7.2.6 ベータ分布 . . . 229
7.2.7 ガンマ分布 . . . 230
7.2.8 アーラン分布 . . . 230
7.2.9 パレート分布 . . . 231
7.2.10 ワイブル分布 . . . 231
7.2.11 カイ二乗分布 . . . 231
7.2.12 F 分布 . . . 232
7.2.13 ロジスティック分布 . . . 232
7.2.14 非心カイ二乗分布 . . . 232
7.2.15 非心 F 分布 . . . 233
7.2.16 コーシー分布 . . . 233
7.2.17 レイリー分布 . . . 233
7.2.18 t 分布 . . . 234
7.2.19 三角分布 . . . 234
7.2.20 二項分布 . . . 235
7.2.21 幾何分布 . . . 235
7.2.22 超幾何分布 . . . 235
7.2.23 負の二項分布 . . . 236
7.2.24 ポアソン分布 . . . 236
7.2.25 経験分布 . . . 237
7.2.26 密度分布 . . . 237
7.2.27 サンプリング . . . 238
7.2.28 再生 . . . 238
7.2.29 ステップ . . . 238
7.2.30 変化率ステップ . . . 238

第 8 章 強化学習 239

8.1 概要 . . . 239
8.2 強化学習モデル . . . 239
8.3 強化学習関連のオブジェクト . . . 247
8.3.1 観測値・行動値集合 . . . 247
8.3.2 観測値・行動値 . . . 249
8.3.3 状態価値関数 . . . 250
8.3.4 行動価値関数, preference, numerator, denominator . . . 252
8.3.5 方策・Actor . . . 257
8.3.6 Critic . . . 261
8.3.7 更新手法 . . . 264

8.3.8 トレース . . . 271
8.3.9 報酬決定関数 . . . 271
8.3.10 その他 . . . 272
8.4 出力されるモデルの利用方法 . . . 272

第 9 章 粒子フィルタ 275
9.1 概要 . . . 275
9.2 粒子群クラス . . . 275
9.3 粒子フィルタの処理 . . . 279

第 10 章 地図エディタ 285
10.1 Python コンソール . . . 285
10.2 API . . . 285

第 11 章 その他 291
11.1 その他 . . . 291
11.1.1 バージョン番号の取得 . . . 291

関連図書 292

第1章　はじめに

psim 言語は、Python 上で動作するシミュレーションの記述、実行、分析をサポートするための言語である。

主な機能としては、以下がある。

- 離散イベントシミュレーションを記述、実行する機能
- エージェントベースシミュレーションを記述、実行する機能
- フローベースでシミュレーションを記述、実行する機能
- 分析機能
- 乱数生成機能
- 強化学習機能

S^4 Simulation System の様々な機能は psim 言語を使う事で実現されている。

1.1　準備

psim の機能を使えるようにするには、最初に、

コード例
```
from psim import *
```

を実行する。

また、初期化のために、

コード例
```
initialize()
```

を実行する。

API　initialize()

シミュレーションの内部状態を全て初期化する。

第2章　シミュレーション記述

2.1　プロセス

2.1.1　プロセスの定義

プロセスは、Python のジェネレータ関数として定義する。

コード例

```
def proc():
    ...
    yield ガード式
    ...
```

Python では、関数内に yield がひとつでも含まれればジェネレータ関数となる。

プロセス内には、任意の Python のコードが書ける。また、プロセス内には待ち受けを実行するために、複数の

コード例

```
    yield ガード式
```

を記述する事が出来る。待ち受けは、任意の Python の制御構造内に記述する事が出来る。

コード例

```
def proc():
    ...
    for i in range(100):
        yield ガード式
    ...
    if someCondition():
        yield ガード式
    ...
```

しかしながら、ひとつも yield がない場合、無効なプロセスとなる。(プロセスを登録する時にエラーが発生する)

また、以下のように proc 内で proc2 を呼出し、proc2 内で yield を書いてもエラーにはならないが、proc2 内で yield を実行しても、スケジューラに命令が渡らないため、待ち受けは実行されない。(サブプロセスを実現したいのであれば、2.17 章を参照)

エラーコード例

```
def proc():
    ...
    proc2()
    ...

def proc2():
    ...
    yield ガード式
    ...

activate(proc)()
```

プロセスはインスタンスメソッドとして定義する事も可能である。

コード例

```
class Person:
    def proc(self):
        ...

person = Person()
activate(person.proc())()
```

2.1.2 プロセスの登録

プロセスをシミュレーターに登録するには、activate を実行する。

API activate(process, at = False, delay = False, cond = False,priority = 0, name = ””, terminatedHandler = nopHandler,suspendedHandler = nopHandler, resumedHandler = nopHandler)(*args,**keys)

プロセス process を起動する。起動時間 at と起動遅延時間 delay とガード式 cond はオプショナルであるが、複数を指定した場合はエラーとなる。at が指定された場合は、シミュレーション時間 at にプロセスを起動する。delay が指定された場合は、プロセスの起動を指定時間遅らせる。cond が指定された場合、まず cond の成立を待ち受ける。何れも指定されなかった場合は、起動前の待ち受けは行われない。起動条件成立後、プロセス名 name のプロセス process を実行する。at もしくは delay が指定された時のみ、優先度 priority は有効となり、同時刻の起動優先度を示す。終了、一時停止、再開ハンドラをそれぞれ、terminatedHandler,suspendedHandler, resumedHandler で指定する事が出来る。各ハンドラは、1 引数の関数で、プロセス状態が変化した時に、発火値の設定された単一の待ち受け結果を引数として呼出される。ハンドラを明示的に指定しなかった場合は、空のハンドラが設定される。(*args, **keys) に書かれた引数は全て process に送られる。返り値はプロセスの実行状態を現す継続オブジェクトである。

コード例

```
def proc(delay, msg):
    yield pause(delay)
    print(msg)

activate(proc, at = 10, priority = 3,
         name = "process foo")(3, "Hello world!")

start()
```

なお、たとえ、プロセスが無引数だったとしても、activate のふたつ目の括弧は省略出来ない。

エラーコード例

```
def proc():
    ...

activate(proc)
```

は、エラーとはならないが、無処理となる。

2.1.3 シミュレーションの開始

シミュレーションを開始するには、start を実行する。

API start(until = None, odesolver = dopri54Solver())

シミュレーションを開始する。until が指定された場合は、シミュレーション時間が until になったら、シミュレーションを停止する。until が指定されなかった場合は、スケジューラのキューが空になるまで実行し、キューが空になると停止する。odesolver は連続シミュレーションを実行する際の常微分方程式ソルバーを指定する。常微分方程式ソルバーは数種類用意されており、詳細は、2.14.8 章 を参照の事。

2.1.4 シミュレーションの停止

API finish()

シミュレーションを停止する。シミュレーション実行中にシミュレーションを停止したい場合は、プロセス内で、finish() を呼ぶと、シミュレーションが停止する。

2.2 待ち受け

2.2.1 待ち受けの実行

プロセス内では様々な待ち受けを行う事が出来る。待ち受けの条件をガード式と呼ぶ。

API yield ガード式

プロセス定義内に、この記述があると、プロセスの実行を一旦停止し、スケジューラにガード式が戻る。スケジューラはガード式によって、プロセスの実行を制御する。

2.3 時間

2.3.1 シミュレーション時間の取得

API now()

現在のシミュレーション時間を返す。

2.3.2 時間待ち受け

API yield pause(delay = 0, priority = 0, name = "")

シミュレーション時間 delay だけプロセスの実行を停止するガード式である。優先度は priority、ガード名は name となる。同時刻に複数の時間待ち受けが成立した場合、優先度 priority が高いものが先に実行される。発火値は None である。

2.4 イベント

2.4.1 イベントの定義

API Event(name = "", queue = FIFO, monitor = False)

名前 name のイベントオブジェクトを作成する。キューの種別は queue となる。キューについては、2.24 を参照。monitor が真の場合は、イベントの待ち行列数の時系列変化を時系列モニターに記録する。

2.4.2　イベント待ち受け

API yield　event.wait(priority = 0, name = ””, canceled = None)

イベントオブジェクト event が発火するまで、プロセスの実行を停止するガード式である。優先度は priority、ガード名は name となる。同時刻に複数のイベント待ち受けが成立した場合、キュー種別によって、実行順が定まる。発火値はイベント発火時に指定された値となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.4.3　イベント発火

API　event.signal(val = None, n = False)

イベントオブジェクト event を発火する。event で待ち受けている全てのイベント待ち受け状態のプロセスを再開する。val が指定された場合、イベント待ち受けの発火値に指定された値が渡る。n が指定された場合は最大 n 個のイベント待ち受け状態のプロセスを再開し、そうでなければ、全てのイベント待ち受け状態のプロセスを再開する。

API yield　event.signal(val = None, n = False)

イベント発火 e.signal(val, n) を実行後、常に成立する待ち受けを行う。

2.4.4　イベントの状態取得

API　event.name

イベントオブジェクト event の名前を取得する。

API event.sizeWaitQueue()

イベントオブジェクト event を待ち受けている数を返す。

2.4.5 イベントの時系列モニター

API event.monitor

イベントオブジェクトの時系列変化を記録した時系列モニターを返す。イベントオブジェクトで時系列モニターを記録した場合のみ、参照可能となる。列数は 1 であり、発火待ち数を示す。

2.5 ファシリティ

2.5.1 ファシリティの定義

API Facility(capacity = 1, name = ””, queue = FIFO, monitor = False, schedule = [], offset = 0.0, repeat = False, gfailure = None, grepair = None, releaseIfStopped = False)

容量が capacity で、名前 name のファシリティオブジェクトを作成する。キューの種別は queue となる。キューについては、2.24 を参照。monitor が真の場合は、ファシリティの待ち行列数の時系列変化を時系列モニターに記録する。

schedule は時間と容量からなるタプルのリスト [(時間$_1$, 容量$_1$), (時間$_2$, 容量$_2$), ...] である。offset には開始時間を指定する。つまり、offset + 時間$_i$ に、容量が容量$_i$ に変化する。repeat に数値が指定された場合、任意の k $\geq$ 0 に対し、offset + repeat * k + 時間$_i$ に、容量が容量$_i$ に変化する。ただし repeat に数値を指定する場合は、schedule 内の最大時間以上でなくてはならない。また、時間$_i$ ¡ 時間$_{i+1}$ でなくてはならない。schedule が長さ 0 のリストであった場合は、容量変化スケジュールは登録されない。

gfailure にジェネレータが指定された場合、故障の発生間隔を返す乱数生成器として利用される。grepair にジェネレータが指定された場合、故障発生後に回復するまでの時間を返す乱数生成器として利用される。gfailure にジェネレータが指定され、grepair にジェネレータが指定されていない場合は、一度故障すると回復しない。

releaseIfStopped が指定されていた場合、ファシリティロックが利用不可になった時に、強制的に release を実行する。ファシリティロックを利用しているプロセスは、facilityLock.isAvailable() にて、現在利用可能か否かを確認する事ができる。もしくはガード式 facilityLock.stopped() を使う事で、利用不可能になる事を待ち受ける事も出来る。releaseIfStopped が指定されていない場合は、強制的な release は発生しない。その場合、一時的に facility.sizeBuffer() $>$ facility.capacity - facility.nfailure になる場合がある。

2.5.2 ファシリティの取得

API yield facility.request(priority = 0, name = ””, canceled = None)

ファシリティ facility からロックを取得出来るまで、プロセスの実行を停止するガード式である。優先度は priority、ガード名は name となる。発火値としてファシリティロックオブジェクトが返される。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.5.3 ファシリティの解放

API facilityLock.release()

ファシリティのロックを解放する。ファシリティを待ち受けているファシリティ待ち受け状態のプロセスが再開する。プロセスの再開順序は、ファシリティの取得待ち受けで指定された優先度の高いもの、優先度が同じ場合は作成順番が早いものとなる。

API yield facilityLock.release()

ファシリティのロックの解放 facilityLock.release() を実行後、常に成立する待ち受けを行う。

コード例

```
result = yield facility.request(name = "request")
yield pause(100)
result["request"].release()
```

コード例

```
result = yield facility.request()
yield pause(100)
result.val.release()
```

あるいは、2.22.7 章の API も利用可能である。

コード例

```
result = yield facility.request()
yield pause(100)
result.releaseAllFacilities()
```

2.5.4 ファシリティの利用可能性確認

API facilityLock.isAvailable()

ファシリティロックが現在利用可能かどうかを調べ、真偽を返す。利用不可能とは、ファシリティ容量が減少し、利用不可能になった場合と、故障が発生し、利用不可能になった場合がある。

API yield facilityLock.stopped(name = ””)

ファシリティロックが利用不可能になるのを待ち受けるガード式である。利用不可能とは、ファシリティ容量が減少し、利用不可能になった場合と、故障が発生し、利用不可能になった場合がある。ガード名は name となる。

2.5.5 ファシリティの解放ハンドラ

API releaseIfRequested

ファシリティの取得時に canceled ハンドラとして、releaseIfRequested を指定すると、キャンセルされた場合に、自動的にファシリティを解放する。

2.5.6 ファシリティの状態取得

API facility.name

ファシリティオブジェクト facility の名前を取得する。

API facility.capacity

ファシリティオブジェクト facility の容量を取得する。

API facility.sizeBuffer()

ファシリティオブジェクト facility の現在のバッファ数を返す。ロックが満杯の時、facility.sizeBuffer() == facility.capacity となる。

API facility.nfailure

ファシリティオブジェクト facility の現在停止中のバッファ数を返す。

API facility.sizeWaitQueue()

ファシリティオブジェクト facility を待ち受けている数を返す。

2.5.7 ファシリティの時系列モニター

API facility.monitor

ファシリティオブジェクトの時系列変化を記録した時系列モニターを返す。ファシリティオブジェクトで時系列モニターを記録した場合のみ、参照可能となる。列数は 2 であり、それぞれ、要求待ち数、バッファ数 (利用可能なファシリティ数) を示す。

2.6 段取替ファシリティ

2.6.1 ファシリティの定義

API ChangeoverFacility(products, product, changeoverMat, product, name = ””, queue = FIFO, monitor = False, schedule = [], offset = 0.0, repeat = False, gfailure = None, grepair = None, releaseIfStopped = False)

段取替に対応した、容量が 1 固定の、名前 name のファシリティオブジェクトを作成する。キューの種別は queue となる。キューについては、2.24 を参照。monitor が真の場合は、ファシリティの待ち行列数の時系列変化を時系列モニターに記録する。
products には、このファシリティが対応する品種のリストを指定する。
product には、初期の品種を指定する。products の要素でなくてはならない。
changeoverMat には、段取替時間を行列形式 (リストのリスト) で指定する。品種 A から 品種 B への段取替時間を changeoverMat[品種 A のインデックス][品種 B のインデックス] に指定する。ここで品種 A のインデックスとは、products リスト内の品種 A のインデックスである。なお、changeoverMat[i][i] は指定しても利用されない。
品種 product の段取替時間 time を指定する。既に指定されている品種の場合はエラーとなるので、一度削除する必要がある。
その他のオプションは、Faciltiy と同じである。

2.6.2 段取替ファシリティの取得

API yield changeoverFacility.request(product, priority = 0, name = "", canceled = None)

段取替ファシリティ changeoverFacility からロックを取得出来るまで、プロセスの実行を停止するガード式である。優先度は priority、ガード名は name となる。発火値としてファシリティロックオブジェクトが返される。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。段取替ファシリティは最初に段取替を実施する。品種 product が現時点で設定されている品種である場合は、何もしない。そうでない場合、指定された品種へ段取替を実施する。
段取り替え中は、段取替ファシリティをロックする。

段取替ファシリティ changeoverFacility からロックを取得出来るまで、プロセスの実行を停止するガード式である。優先度は priority、ガード名は name となる。発火値としてファシリティロックオブジェクトが返される。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

段取替ファシリティは最初に段取替を実施する。品種 product が現時点で設定されている品種である場合は、何もしない。そうでない場合、指定された品種へ段取替を実施する。

段取り替え中は、ファシリティをロックする。

キャンセルの動作

request 中にスケジュールによるキャパシティの減少、故障が発生した場合、

- 段取替中なら、段取替は中止され、品種は段取替前の状態から変化しない。
- ファシリティロックの取得待ち状態なら、取得待ちを続ける。ただし、releaseIfStopped が指定されている場合は強制的に開放する (従来の仕様)

request 中にキャンセルが発生した場合、

- 段取替中なら、段取替は中止され、品種は段取替前の状態から変化しない。
- ファシリティロックの取得待ち状態なら、待ち受けを停止する。

判定方法

- 段取替要求は外部要因 (スケジュールや故障) で失敗する事がある。要求後は、必ず品種が切り替わったかどうかを確認し、適切な処理を書く必要がある。

コード例

```
result = yield changeoverFacility.request("A",
                                          name = "request")
if changeoverFacility.product == "A":
    # 段取替成功
else:
    # 段取替失敗
```

2.6.3 段取替ファシリティの状態取得

API changeoverFacility.product

現時点の品種を返す。(読み込み専用)

API changeoverFacility.name

段取替ファシリティオブジェクト changeoverFacility の名前を取得する。

API changeoverFacility.capacity

段取替ファシリティオブジェクト changeoverFacility の容量を取得する。

API changeoverFacility.sizeBuffer()

段取替ファシリティオブジェクト changeoverFacility の現在のバッファ数を返す。ロックが満杯の時、changeoverFacility.sizeBuffer() == changeoverFacility.capacity となる。

API changeoverFacility.nfailure

段取替ファシリティオブジェクト changeoverFacility の現在停止中のバッファ数を返す。

API changeoverFacility.sizeWaitQueue()

段取替ファシリティオブジェクト changeoverFacility を待ち受けている数を返す。

2.6.4 段取替ファシリティの時系列モニター

API changeoverFacility.monitor

段取替ファシリティオブジェクトの時系列変化を記録した時系列モニターを返す。ファシリティオブジェクトで時系列モニターを記録した場合のみ、参照可能となる。列は それぞれ、要求待ち数、バッファ数 (利用可能なファシリティ数)、容量、停止中数、段取替中数、品種を示す。

2.7 タンク

2.7.1 タンクの定義

API Tank(capacity = 1, buffer = 0, name = ””, putQueue = FIFO, getQueue = FIFO, monitor = False)

容量が capacity で、初期量が buffer で、名前が name のタンクオブジェクトを作成する。追加操作用のキューの種別は putQueue となる。取得操作用のキューの種別は getQueue となる。キューについては、2.24 章を参照。monitor が真の場合は、タンクの待ち行列数の時系列変化を時系列モニターに記録する。
schedule は時間と容量からなるタプルのリスト [(時間 $_1$, 容量 $_1$), (時間 $_2$, 容量 $_2$), ...] である。offset には開始時間を指定する。つまり、offset + 時間 $_i$ に、容量が容量 $_i$ に変化する。repeat に数値が指定された場合、任意の k $\geq$ 0 に対し、offset + repeat * k + 時間 $_i$ に、容量が容量 $_i$ に変化する。ただし repeat に数値を指定する場合は、schedule 内の最大時間以上でなくてはならない。また、時間 $_i$ ¡ 時間 $_{i+1}$ でなくてはならない。schedule が長さ 0 のリストであった場合は、容量変化スケジュールは登録されない。
overflow は、2 引数の関数あるいは None を指定する。None でない場合、現在のバッファ数が容量を越えてしまった場合に、第 1 引数がタンクオブジェクト、第 2 引数が容量を越えてしまった量が設定されて呼ばれる。

2.7.2　タンクへの追加

API yield　tank.put(val, priority = 0, name = ””, cancled = None)

タンクに val を加えるガード式である。優先度は priority、ガード名は name となる。タンクに空きがある場合は、即座に発火する。タンク容量を越えてしまいタンクに追加が出来ない場合は、空きが出来るまでプロセスを停止する。空きが出来たら、キューの種別によって実行順が定まる。発火値は None となる。val は 0 以上でなくてはならない。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.7.3　タンクからの取得

API yield　tank.get(val, priority = 0, name = ””, canceled = None)

タンクから val を取得するガード式である。優先度は priority、ガード名は name となる。タンク容量が足らず取得が出来ない場合は、取得出来るまでプロセスを停止する。実行順はキューの種別によって定まる。発火値は None となる。val は 0 以上でなくてはならない。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.7.4　タンクの状態取得

API　tank.name

タンクオブジェクト tank の名前を取得する。

API tank.capacity

タンクオブジェクト tank の容量を取得する。

API tank.sizeBuffer()

タンクオブジェクト tank の現在のバッファ量を返す。タンクが空の時、tank.sizeBuffer() == 0 となり、タンクが満杯の時、tank.sizeBuffer() == tank.capacity となる。

API tank.sizeGetQueue()

タンクオブジェクト tank からの取得を待ち受けている数を返す。

API tank.sizeGetQueueSum()

タンクオブジェクト tank からの取得を待ち受けている各要求の合計要求量を返す。

API tank.sizePutQueue()

タンクオブジェクト tank への追加を待ち受けている数を返す。

API tank.sizePutQueueSum()

タンクオブジェクト tank への追加を待ち受けている各要求の合計要求量を返す。

2.7.5 タンクの時系列モニター

API tank.monitor

タンクオブジェクトの時系列変化を記録した時系列モニターを返す。タンクオブジェクトで時系列モニターを記録した場合のみ、参照可能となる。列数は 5 であり、それぞれ、追加待ち数、追加待ち合計、取得待ち数、取得待ち合計、バッファ量を示す。

2.8 ストア

2.8.1 ストアの定義

API Store(buffer = [], capacity = False, name = ””, putQueue = FIFO, getQueue = FIFO, monitor = False, init = [], schedule = [], offset = 0.0, repeat = False, overflow = None)

オブジェクトリストが buffer で、容量が capacity で、名前が name のストアオブジェクトを作成する。buffer はリストである。capacity はリストの最大サイズを指し、False が指定された場合は、容量制限は無しとなる。追加操作用のキューの種別は putQueue となる。取得操作用のキューの種別は getQueue となる。キューについては、2.24 章を参照。monitor が真の場合は、ストアの待ち行列数の時系列変化を時系列モニターに記録する。init 引数には値と優先度からなるタプルのリスト [(値 1, 優先度 1), (値 2, 優先度 2), ...] を指定する事が出来る。空リストでない場合、シミュレーション開始時に各要素に対し、store.put1(値, priority = 優先度) が実行される。容量以上の要素を指定した場合、追加キューに並ぶ。
schedule は時間と容量からなるタプルのリスト [(時間$_1$, 容量$_1$), (時間$_2$, 容量$_2$), ...] である。offset には開始時間を指定する。つまり、offset + 時間$_i$ に、容量が容量$_i$ に変化する。repeat に数値が指定された場合、任意の k $\geq$ 0 に対し、offset + repeat * k + 時間$_i$ に、容量が容量$_i$ に変化する。ただし repeat に数値を指定する場合は、schedule 内の最大時間以上でなくてはならない。また、時間$_i$ ¡ 時間$_{i+1}$ でなくてはならない。schedule が長さ 0 のリストであった場合は、容量変化スケジュールは登録されない。
overflow は、2 引数の関数あるいは None を指定する。None でない場合、現在のバッファ数が容量を越えてしまった場合に、第 1 引数がストアオブジェクト、第 2 引数が容量を越えてしまった数が設定されて呼ばれる。

2.8.2　ストアへの追加

API yield store.put(buffer, priority = 0, name = ””, canceled = None, disposal = -1, disposed = None)

ストアのオブジェクトリストに buffer を加えるガード式である。優先度は priority、待ち受け名は name となる。buffer は追加するオブジェクトのリストである。ストア容量を越えてしまいオブジェクト追加が出来ない場合は、空きが出来るまでプロセスを停止する。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。disposal が 0 以上の値が設定されていた場合、disposal 時間経過後に、ストアの追加待ち行列もしくはストア内に残っていた場合強制廃棄される。disposed に 1 引数の関数が指定されていた場合、廃棄されたオブジェクトを引数に、disposed 関数がそれぞれ呼び出される。ストアの追加待ちの状態で破棄された場合、store の追加処理は終了するが、待ち受け結果には ”disposed” が含まれる。

注意: 破棄されると、disposed 関数が呼ばれ、ストアから除去されるが、対象となるオブジェクト自体が破棄されるわけでない。オブジェクトを参照しているプロセスが存在する場合、ユーザーコードで後始末を行う必要がある。

API yield store.put1(obj, priority = 0, name = ””, canceled = None, disposal = -1, disposed = None)

ストアのオブジェクトリストに obj を加えるガード式である。優先度は priority、待ち受け名は name となる。obj は追加するオブジェクトである。ストア容量を越えてしまいオブジェクト追加が出来ない場合は、空きが出来るまでプロセスを停止する。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。disposal が 0 以上の値が設定されていた場合、disposal 時間経過後に、ストアの追加待ち行列もしくはストア内に残っていた場合強制廃棄される。disposed に 1 引数の関数が指定されていた場合、廃棄されたオブジェクトを引数に、disposed 関数が呼び出される。ストアの追加待ちの状態で破棄された場合、store の追加処理は終了するが、待ち受け結果には ”disposed” が含まれる。

注意: 破棄されると、disposed 関数が呼ばれ、ストアから除去されるが、対象となるオブジェクト自体が破棄されるわけでない。オブジェクトを参照しているプロセスが存在する場合、ユーザーコードで後始末を行う必要がある。

2.8.3 ストアからの取得

API yield store.get(n, condition = None, priority = 0, name = ””, canceled = None)

ストアのオブジェクトリストから n 個のオブジェクトを取得するガード式である。condition に 1 引数の関数を指定した場合、condition が成立するようなオブジェクトのみを取得する。優先度は priority、待ち受け名は name となる。待ち受け結果は取得したリストである。オブジェクト数が足らず取得出来ない場合は、取得出来るまでプロセスを停止する。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield store.get1(condition = None, priority = 0, name = ””, canceled = None)

ストアのオブジェクトリストから ひとつのオブジェクトを取得するガード式である。condition に 1 引数の関数を指定した場合、condition が成立するようなオブジェクトのみを取得する。優先度は priority、待ち受け名は name となる。待ち受け結果は取得したオブジェクトである。オブジェクト数が足らず取得出来ない場合は、取得出来るまでプロセスを停止する。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.8.4 ストアの状態取得

API store.name

ストアオブジェクト store の名前を取得する。

API store.capacity

ストアオブジェクト store の容量を取得する。

API store.sizeBuffer()

ストアオブジェクト store の現在のバッファ数を返す。ストアが空の時、store.sizeBuffer() == 0 となり、ストアが満杯の時、store.sizeBuffer() == store.capacity となる。

API store.sizeGetQueue()

ストアオブジェクト store からの取得を待ち受けている数を返す。

API store.sizeGetQueueSum()

ストアオブジェクト store からの取得を待ち受けている各要求の合計要求量を返す。

API store.sizePutQueue()

ストアオブジェクト store への追加を待ち受けている数を返す。

API store.sizePutQueueSum()

ストアオブジェクト store への追加を待ち受けている各要求の合計要求量を返す。

API store.buffer

ストアのオブジェクトリストを返す。この値を書き換える場合は、他のプロセスも参照する可能性もあるので、必ずアトミックに行わなくてはならない。つまり、store.buffer を読み取った時点と、書き込む時点間に、プロセスの切り替えが発生してはならない。store.replace を利用した場合は、アトミック性が保証される。

API store.replace(filter)

ストアのオブジェクトリストを引数に filter 関数を呼び出し、その結果であるリストが、ストアのリストに置き換わる。

コード例

```
store = Store()
def proc():
    yield store.put1(3)
    yield store.put1(4)
    yield store.put1(2)
def proc2():
    yield pause(1)
    gotlist = []
    def filter(lst):
        rest = []
        for elem in lst:
            if elem == 4:
                gotlist.append(elem)
            else:
                rest.append(elem)
        return rest
    store.replace(filter)
    print("rest", store.buffer) # (=> [3, 2])
    print("gotlist", gotlist) # (=> [4])
activate(proc)()
activate(proc2)()
start()
```

は、ストアの中から、値が 4 の要素のみを取り出す。

2.8.5 ストアの時系列モニター

API store.monitor

ストアオブジェクトの時系列変化を記録した時系列モニターを返す。ストアオブジェクトで時系列モニターを記録した場合のみ、参照可能となる。列数は 5 であり、それぞれ、追加待ち数、追加待ち合計、取得待ち数、取得待ち合計、バッファ数を示す。

2.9 複合待ち受け

複数のガード式を組み合わせて、より複雑な待ち受けを実行する事が出来る。

2.9.1 OR 待ち受け

API yield ガード式 $_1$ | ガード式 $_2$

ガード式 $_1$ と、ガード式 $_2$ を OR 待ち受けするガード式である。ガード式 $_1$ かガード式 $_2$ のどちらか一方が成立すれば、残りのガード式は除去 (2.9.4 章) され、このガード式が成立する。

API yield anyOf(ガード式 $_1$, ガード式 $_2$,···, ガード式 $_n$)

ガード式 $_1$ からガード式 $_n$ を OR 待ち受けするガード式である。あるガード式 $_i$ $(1 \leq i \leq n)$ が成立すれば、残りのガード式 $_j$ $(j \neq i)$ は除去 (2.9.4 章) され、このガード式が成立する。

2.9.2 AND 待ち受け

API yield ガード式 $_1$ & ガード式 $_2$

ガード式 $_1$ と、ガード式 $_2$ を AND 待ち受けするガード式である。ガード式 $_1$ とガード式 $_2$ の両方が成立すれば、このガード式が成立する。なお、AND 待ち受けは、待ち受けの完了を同期化するもので、待ち受けの成立時間を同期化するものではない。

API yield allOf(ガード式 $_1$, ガード式 $_2$, ···, ガード式 $_n$)

ガード式 $_1$ から ガード式 $_n$ を AND 待ち受けするガード式である。全ての ガード式 $_i$ $(1 \leq i \leq n)$ が成立すれば、このガード式が成立する。なお、AND 待ち受けは、待ち受けの完了を同期化するもので、待ち受けの成立時間を同期化するものではない。

2.9.3 逐次待ち受け

API yield ガード式 $_1$ >> ガード式 $_2$

ガード式 $_1$ と、ガード式 $_2$ を 逐次待ち受けするガード式である。まずガード式 $_1$ を待ち受けし、ガード式 $_1$ が成立したら、ガード式 $_2$ の待ち受けを開始する。ガード式 $_2$ が成立すれば、このガード式が成立する。

API yield sequenceOf(ガード式 $_1$, ガード式 $_2$, $\cdots$, ガード式 $_n$)

ガード式 $_1$ から ガード式 $_n$ を 逐次待ち受けするガード式である。まず ガード式 $_1$ を待ち受けし、ガード式 $_1$ が成立したら、ガード式 $_2$ の待ち受けを開始する。同様にガード式 $_i$ $(1 \leq i \leq n-1)$ が成立したら、ガード式 $_{i+1}$ の待ち受けを開始する。ガード式 $_n$ が成立すれば、このガード式が成立する。

2.9.4 ガード式のキャンセル

OR 待ち受けのある子ガード式のひとつが成立すると、OR 待ち受けが成立する。その時、残りのガード式は除去される。除去とは、以下を指す。

1. そのガード式がまだ待ち受け登録されていない場合 (逐次待ち受けの後半など)、待ち受け登録の予約が除去される。
2. そのガード式が既に登録されているが、まだ成立していない場合 (資源待ち受けなど)、その資源待ち受けが除去される。
3. そのガード式が既に成立している場合 (逐次待ち受けの前半など)、何もしない。

ただし、資源待ち受け (ファシリティの取得、タンクからの取得など) の場合、キャンセルハンドラを指定する事が出来る。キャンセルハンドラを指定している場合、上記 3 の場合に、キャンセルハンドラが呼ばれる。

2.9.5 結合優先度

結合優先度は Python の演算子の優先度に従う。つまり、優先度の高い順に、`>>` ，`&` ，`|` になる。例えば、`a`，`b`，`c`，`d`，`e` はガード式とすると、

コード例
```
yield a >> b | c >> d & e >> f
```

は、

コード例
```
yield (a >> b) | ((c >> d) & (e >> f))
```

と解釈される。

2.10 複合イベント

複数のイベントオブジェクトを組み合わせて、複合イベントオブジェクトを作る事が出来る。また、複合イベントオブジェクトに対して、待ち受けを実行する事が出来る。

2.10.1 OR 複合イベント

API $\texttt{event}_1$ | $\texttt{event}_2$

イベントオブジェクト $\texttt{event}_1$ とイベントオブジェクト $\texttt{event}_2$ の OR 複合イベントオブジェクトを作成する。

API yield ($\texttt{event}_1$ | $\texttt{event}_2$).wait(priority = 0,name = "", canceled = None)

$\texttt{event}_1$ と $\texttt{event}_2$ の OR 複合イベントオブジェクトを待ち受けるガード式である。$\texttt{event}_1$ と $\texttt{event}_2$ のどちらかが発火するまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API anyEventOf($\texttt{event}_1$, $\texttt{event}_2$, $\cdots$, $\texttt{event}_n$)

イベントオブジェクト $\texttt{event}_1$ からイベントオブジェクト $\texttt{event}_n$ の OR 複合イベントオブジェクトを作成する。

API yield anyEventOf($\texttt{event}_1$, ⋯, $\texttt{event}_n$).wait(priority = 0, name = "", canceled = None)

$\texttt{event}_1$ から $\texttt{event}_n$ の OR 複合イベントオブジェクトを待ち受けるガード式である。ある $\texttt{event}_i$ $(1 \leq i \leq n)$ が発火するまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.10.2 AND 複合イベント

API $\texttt{event}_1$ & $\texttt{event}_2$

イベントオブジェクト $\texttt{event}_1$ とイベントオブジェクト $\texttt{event}_2$ の AND 複合イベントオブジェクトを作成する。

API yield ($\texttt{event}_1$ & $\texttt{event}_2$).wait(priority = 0, name = "", canceled = None)

$\texttt{event}_1$ と $\texttt{event}_2$ の AND 複合イベントオブジェクトを待ち受けるガード式である。$\texttt{event}_1$ と $\texttt{event}_2$ の両方が発火するまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API allEventOf($\texttt{event}_1$, $\texttt{event}_2$, ⋯, $\texttt{event}_n$)

イベントオブジェクト $\texttt{event}_1$ からイベントオブジェクト $\texttt{event}_n$ の AND 複合イベントオブジェクトを作成する。

API yield allEventOf(event$_1$, ⋯, event$_n$).wait(priority = 0, name = "", canceled = None)

event$_1$ から event$_n$ の AND 複合イベントオブジェクトを待ち受けるガード式である。全ての event$_i$ $(1 \leq i \leq n)$ が発火するまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.10.3 逐次複合イベント

API event$_1$ >> event$_2$

イベントオブジェクト event$_1$ とイベントオブジェクト event$_2$ の逐次複合イベントオブジェクトを作成する。

API yield (event$_1$ >> event$_2$).wait(priority = 0, name = "", canceled = None)

event$_1$ と event$_1$ の逐次複合イベントオブジェクトを待ち受けるガード式である。まず event$_1$ の待ち受けを開始する。event$_1$ が発火すると event$_2$ の待ち受けを開始し、event$_2$ が発火するまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API sequenceEventOf(event$_1$, event$_2$, ⋯, event$_n$)

イベントオブジェクト event$_1$ からイベントオブジェクト event$_n$ の逐次複合イベントオブジェクトを作成する。

API yield `sequenceEventOf(event`$_1$`, ···, event`$_n$`).wait(priority = 0, name = "", canceled = None)`

event$_1$ から event$_n$ の逐次複合イベントを待ち受けるガード式である。まず event$_1$ の待ち受けを開始する。event$_1$ が発火すると event$_2$ の待ち受けを開始する。同様に event$_i$ ($1 \leq i \leq n-1$) が発火すると、event$_{i+1}$ の待ち受けを開始する。event$_n$ が発火するまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.10.4 結合優先度

結合優先度は Python の演算子の優先度に従う。つまり、優先度の高い順に、>> , & , | になる。例えば、a, b, c, d, e はイベントとすると、

コード例
```
yield (a >> b | c >> d & e >> f).wait()
```
は、

コード例
```
yield ((a >> b) | ((c >> d) & (e >> f))).wait()
```
と解釈される。

2.11 複合ファシリティ

複数のファシリティオブジェクトを組み合わせて、複合ファシリティオブジェクトを作る事が出来る。また、複合ファシリティオブジェクトに対して、待ち受けを実行する事が出来る。

2.11.1 OR 複合ファシリティ

API `facility`$_1$ `|` `facility`$_2$

ファシリティオブジェクト facility$_1$ とファシリティオブジェクト facility$_2$ の OR 複合ファシリティオブジェクトを作成する。

API yield (facility$_1$ | facility$_2$).request(priority = 0, name = "", canceled = None)

facility$_1$ と facility$_2$ の OR 複合ファシリティオブジェクトを待ち受けるガード式である。facility$_1$ と facility$_2$ のどちらかからロックを取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API anyFacilityOf(facility$_1$, facility$_2$, $\cdots$, facility$_n$)

ファシリティオブジェクト facility$_1$ からファシリティオブジェクト facility$_n$ の OR 複合ファシリティオブジェクトを作成する。

API yield anyFacilityOf($\texttt{facility}_1$, ⋯, $\texttt{facility}_n$).request(priority = 0, name = "", canceled = None)

$\texttt{facility}_1$ から $\texttt{facility}_n$ の OR 複合ファシリティオブジェクトを待ち受けるガード式である。ある $\texttt{facility}_i$ $(1 \leq i \leq n)$ のロックが取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.11.2 AND 複合ファシリティ

API $\texttt{facility}_1$ & $\texttt{facility}_2$

ファシリティオブジェクト $\texttt{facility}_1$ とファシリティオブジェクト $\texttt{facility}_2$ の AND 複合ファシリティオブジェクトを作成する。

API yield ($\texttt{facility}_1$ & $\texttt{facility}_2$).request(priority = 0,name = "", canceled = None)

$\texttt{facility}_1$ と $\texttt{facility}_2$ の AND 複合ファシリティオブジェクトを待ち受けるガード式である。$\texttt{facility}_1$ と $\texttt{facility}_2$ の両方のロックが取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API allFacilityOf(facility$_1$, facility$_2$, $\cdots$, facility$_n$)

ファシリティオブジェクト facility$_1$ からファシリティオブジェクト facility$_n$ の AND 複合ファシリティオブジェクトを作成する。

API yield allFacilityOf(facility$_1$, $\cdots$, facility$_n$).request(priority = 0, name = "", canceled = None)

facility$_1$ から facility$_n$ の AND 複合ファシリティオブジェクトを待ち受けるガード式である。全ての facility$_i$ $(1 \leq i \leq n)$ のロックが取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.11.3 逐次複合ファシリティ

API facility$_1$ >> facility$_2$

ファシリティオブジェクト facility$_1$ とファシリティオブジェクト facility$_2$ の逐次複合ファシリティオブジェクトを作成する。

API yield ($\texttt{facility}_1$ >> $\texttt{facility}_2$).request(priority = 0, name = "", canceled = None)

$\texttt{facility}_1$ と $\texttt{facility}_1$ の逐次複合ファシリティオブジェクトを待ち受けるガード式である。まず $\texttt{facility}_1$ の待ち受けを開始する。$\texttt{facility}_1$ のロックが取得出来たら $\texttt{facility}_2$ の待ち受けを開始し、$\texttt{facility}_2$ のロックが取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API sequenceFacilityOf($\texttt{facility}_1$, $\texttt{facility}_2$, $\cdots$, $\texttt{facility}_n$)

ファシリティオブジェクト $\texttt{facility}_1$ からファシリティオブジェクト $\texttt{facility}_n$ の逐次複合ファシリティオブジェクトを作成する。

API yield sequenceFacilityOf($\texttt{facility}_1$, $\cdots$, $\texttt{facility}_n$).request(priority = 0, name = "", canceled = None)

$\texttt{facility}_1$ から $\texttt{facility}_n$ の逐次複合ファシリティを待ち受けるガード式である。まず $\texttt{facility}_1$ の待ち受けを開始する。$\texttt{facility}_1$ のロックが取得出来たら $\texttt{facility}_2$ の待ち受けを開始する。同様に $\texttt{facility}_i$ $(1 \leq i \leq n-1)$ のロックが取得出来たら、$\texttt{facility}_{i+1}$ の待ち受けを開始する。$\texttt{facility}_n$ のロックが取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.11.4 結合優先度

結合優先度は Python の演算子の優先度に従う。つまり、優先度の高い順に、>> , & , | になる。例えば、a, b, c, d, e はファシリティとすると、

コード例
```
yield (a >> b | c >> d & e >> f).request()
```

は、

コード例
```
yield ((a >> b) | ((c >> d) & (e >> f))).request()
```

と解釈される。

2.11.5 複合ファシリティの解放

複合ファシリティのロックを取得した場合、解放忘れがあると、永久にファシリティを占有してしまうので注意が必要である。必ず 2.22.7 章 の方法で全てのファシリティを解放する必要がある。

コード例
```
result = yield (f1 >> f2 | f3 >> f4).request()
result.releaseAllFacilities()
```

あるいは、ファシリティの取得の canceled ハンドラに releaseIfRequested (2.5.5 章) を指定しておく事も出来る。

2.12 複合タンク

複数のタンクオブジェクトを組み合わせて、複合タンクオブジェクトを作る事が出来る。また、複合タンクオブジェクトに対して、待ち受けを実行する事が出来る。

2.12.1 OR 複合タンク

API $\texttt{tank}_1$ | $\texttt{tank}_2$

タンクオブジェクト $\texttt{tank}_1$ とタンクオブジェクト $\texttt{tank}_2$ の OR 複合タンクオブジェクトを作成する。

API yield ($tank_1$ | $tank_2$).get(val, priority = 0,name = "", canceled = None)

$tank_1$ と $tank_2$ の OR 複合タンクオブジェクトを待ち受けるガード式である。$tank_1$ と $tank_2$ のどちらかから取得出来るまで、プロセスの実行を停止する。指定した val を複数のタンクに分散して成立する事はなく、あるタンクに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield ($tank_1$ | $tank_2$).put(val, priority = 0,name = "", canceled = None)

$tank_1$ と $tank_2$ の OR 複合タンクオブジェクトを待ち受けるガード式である。$tank_1$ と $tank_2$ のどちらかへ追加出来るまで、プロセスの実行を停止する。指定した val を複数のタンクに分散して成立する事はなく、あるタンクに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API anyTankOf($tank_1$, $tank_2$, ···, $tank_n$)

タンクオブジェクト $tank_1$ からタンクオブジェクト $tank_n$ の OR 複合タンクオブジェクトを作成する。

API yield anyTankOf(tank$_1$, ···, tank$_n$).get(val, priority = 0, name = "", canceled = None)

tank$_1$ から tank$_n$ の OR 複合タンクオブジェクトを待ち受けるガード式である。ある tank$_i$ $(1 \leq i \leq n)$ から取得出来るまで、プロセスの実行を停止する。指定した val を複数のタンクに分散して成立する事はなく、あるタンクに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield anyTankOf(tank$_1$, ···, tank$_n$).put(val, priority = 0, name = "", canceled = None)

tank$_1$ から tank$_n$ の OR 複合タンクオブジェクトを待ち受けるガード式である。ある tank$_i$ $(1 \leq i \leq n)$ へ追加出来るまで、プロセスの実行を停止する。指定した val を複数のタンクに分散して成立する事はなく、あるタンクに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.12.2 AND 複合タンク

API tank$_1$ & tank$_2$

タンクオブジェクト tank$_1$ とタンクオブジェクト tank$_2$ の AND 複合タンクオブジェクトを作成する。

API yield ($tank_1$ & $tank_2$).get(val, priority = 0,name = "", canceled = None)

$tank_1$ と $tank_2$ の AND 複合タンクオブジェクトを待ち受けるガード式である。$tank_1$ と $tank_2$ の両方から取得出来るまで、プロセスの実行を停止する。指定した val を複数のタンクに分散して成立する事はなく、全てのタンクに対して成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield ($tank_1$ & $tank_2$).put(val, priority = 0,name = "", canceled = None)

$tank_1$ と $tank_2$ の AND 複合タンクオブジェクトを待ち受けるガード式である。$tank_1$ と $tank_2$ の両方へ追加出来るまで、プロセスの実行を停止する。指定した val を複数のタンクに分散して成立する事はなく、全てのタンクに対して成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API allTankOf($tank_1$, $tank_2$, $\cdots$, $tank_n$)

タンクオブジェクト $tank_1$ からタンクオブジェクト $tank_n$ の AND 複合タンクオブジェクトを作成する。

API yield `allTankOf(tank`$_1$`, ···, tank`$_n$`).get(val, priority = 0, name = "", canceled = None)`

tank$_1$ から tank$_n$ の AND 複合タンクオブジェクトを待ち受けるガード式である。全ての tank$_i$ $(1 \leq i \leq n)$ から取得出来るまで、プロセスの実行を停止する。指定した val を複数のタンクに分散して成立する事はなく、全てのタンクに対して成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield `allTankOf(tank`$_1$`, ···, tank`$_n$`).put(val, priority = 0, name = "", canceled = None)`

tank$_1$ から tank$_n$ の AND 複合タンクオブジェクトを待ち受けるガード式である。全ての tank$_i$ $(1 \leq i \leq n)$ へ追加出来るまで、プロセスの実行を停止する。指定した val を複数のタンクに分散して成立する事はなく、全てのタンクに対して成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.12.3 逐次複合タンク

API tank$_1$ >> tank$_2$

タンクオブジェクト tank$_1$ とタンクオブジェクト tank$_2$ の逐次複合タンクオブジェクトを作成する。

API yield (tank$_1$ >> tank$_2$).get(val, priority = 0, name = "", canceled = None)

tank$_1$ と tank$_1$ の逐次複合タンクオブジェクトを待ち受けるガード式である。まず tank$_1$ の待ち受けを開始する。tank$_1$ から取得出来たら tank$_2$ の待ち受けを開始し、tank$_2$ から取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield (tank$_1$ >> tank$_2$).put(val, priority = 0, name = "", canceled = None)

tank$_1$ と tank$_1$ の逐次複合タンクオブジェクトを待ち受けるガード式である。まず tank$_1$ の待ち受けを開始する。tank$_1$ へ追加出来たら tank$_2$ の待ち受けを開始し、tank$_2$ へ追加出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API sequenceTankOf(tank$_1$, tank$_2$, ···, tank$_n$)

タンクオブジェクト tank$_1$ からタンクオブジェクト tank$_n$ の逐次複合タンクオブジェクトを作成する。

API yield `sequenceTankOf(tank`$_1$`, ···, tank`$_n$`).get(val, priority = 0, name = "", canceled = None)`

tank$_1$ から tank$_n$ の逐次複合タンクを待ち受けるガード式である。まず tank$_1$ の待ち受けを開始する。tank$_1$ から取得出来たら tank$_2$ の待ち受けを開始する。同様に tank$_i$ $(1 \leq i \leq n-1)$ から取得出来たら、tank$_{i+1}$ の待ち受けを開始する。tank$_n$ から取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield `sequenceTankOf(tank`$_1$`, ···, tank`$_n$`).put(val, priority = 0, name = "", canceled = None)`

tank$_1$ から tank$_n$ の逐次複合タンクを待ち受けるガード式である。まず tank$_1$ の待ち受けを開始する。tank$_1$ へ追加出来たら tank$_2$ の待ち受けを開始する。同様に tank$_i$ $(1 \leq i \leq n-1)$ へ追加出来たら、tank$_{i+1}$ の待ち受けを開始する。tank$_n$ へ追加出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.12.4 結合優先度

結合優先度は Python の演算子の優先度に従う。つまり、優先度の高い順に、>> , & , | になる。例えば、a, b, c, d, e はタンクとすると、

コード例
```
yield (a >> b | c >> d & e >> f).get()
```

は、

コード例

```
yield ((a >> b) | ((c >> d) & (e >> f))).get()
```

と解釈される。

2.13 複合ストア

複数のストアオブジェクトを組み合わせて、複合ストアオブジェクトを作る事が出来る。また、複合ストアオブジェクトに対して、待ち受けを実行する事が出来る。

2.13.1 OR 複合ストア

API $\texttt{store}_1$ | $\texttt{store}_2$

ストアオブジェクト $\texttt{store}_1$ とストアオブジェクト $\texttt{store}_2$ の OR 複合ストアオブジェクトを作成する。

API yield ($\texttt{store}_1$ | $\texttt{store}_2$).get(n, priority = 0,name = "", canceled = None)

$\texttt{store}_1$ と $\texttt{store}_2$ の OR 複合ストアオブジェクトを待ち受けるガード式である。$\texttt{store}_1$ と $\texttt{store}_2$ のどちらかから取得出来るまで、プロセスの実行を停止する。指定した n を複数のストアに分散して成立する事はなく、あるストアのみに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield (store$_1$ | store$_2$).get1(priority = 0,name = "", canceled = None)

store$_1$ と store$_2$ の OR 複合ストアオブジェクトを待ち受けるガード式である。store$_1$ と store$_2$ のどちらかから取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield (store$_1$ | store$_2$).put(buffer, priority = 0,name = "", canceled = None)

store$_1$ と store$_2$ の OR 複合ストアオブジェクトを待ち受けるガード式である。store$_1$ と store$_2$ のどちらかへ追加出来るまで、プロセスの実行を停止する。指定した buffer を複数のストアに分散して成立する事はなく、あるストアのみに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API anyStoreOf(store$_1$, store$_2$, $\cdots$, store$_n$)

ストアオブジェクト store$_1$ からストアオブジェクト store$_n$ の OR 複合ストアオブジェクトを作成する。

API yield anyStoreOf(store$_1$, ⋯, store$_n$).get(n, priority = 0, name = "", canceled = None)

store$_1$ から store$_n$ の OR 複合ストアオブジェクトを待ち受けるガード式である。ある store$_i$ $(1 \le i \le n)$ から取得出来るまで、プロセスの実行を停止する。指定した n を複数のストアに分散して成立する事はなく、あるストアのみに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield anyStoreOf(store$_1$, ⋯, store$_n$).get1(n, priority = 0, name = "", canceled = None)

store$_1$ から store$_n$ の OR 複合ストアオブジェクトを待ち受けるガード式である。ある store$_i$ $(1 \le i \le n)$ から取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield anyStoreOf(store$_1$, ⋯, store$_n$).put(buffer, priority = 0, name = "", canceled = None)

store$_1$ から store$_n$ の OR 複合ストアオブジェクトを待ち受けるガード式である。ある store$_i$ $(1 \le i \le n)$ へ追加出来るまで、プロセスの実行を停止する。指定した buffer を複数のストアに分散して成立する事はなく、あるストアのみに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.13.2 AND 複合ストア

API $store_1$ & $store_2$

ストアオブジェクト $store_1$ とストアオブジェクト $store_2$ の AND 複合ストアオブジェクトを作成する。

API yield ($store_1$ & $store_2$).get(n, priority = 0,name = "", canceled = None)

$store_1$ と $store_2$ の AND 複合ストアオブジェクトを待ち受けるガード式である。$store_1$ と $store_2$ の両方から取得出来るまで、プロセスの実行を停止する。指定した n を複数のストアに分散して成立する事はなく、全てのストアのみに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield ($store_1$ & $store_2$).get1(priority = 0,name = "", canceled = None)

$store_1$ と $store_2$ の AND 複合ストアオブジェクトを待ち受けるガード式である。$store_1$ と $store_2$ の両方から取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield $(\texttt{store}_1$ & $\texttt{store}_2)$`.put(buffer, priority = 0,name = "", canceled = None)`

$\texttt{store}_1$ と $\texttt{store}_2$ の AND 複合ストアオブジェクトを待ち受けるガード式である。$\texttt{store}_1$ と $\texttt{store}_2$ の両方へ追加出来るまで、プロセスの実行を停止する。指定した buffer を複数のストアに分散して成立する事はなく、全てのストアのみに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API allStoreOf($\texttt{store}_1$, $\texttt{store}_2$, $\cdots$, $\texttt{store}_n$)

ストアオブジェクト $\texttt{store}_1$ からストアオブジェクト $\texttt{store}_n$ の AND 複合ストアオブジェクトを作成する。

API yield allStoreOf($\texttt{store}_1$, $\cdots$, $\texttt{store}_n$)`.get(n, priority = 0, name = "", canceled = None)`

$\texttt{store}_1$ から $\texttt{store}_n$ の AND 複合ストアオブジェクトを待ち受けるガード式である。全ての $\texttt{store}_i$ $(1 \leq i \leq n)$ から取得出来るまで、プロセスの実行を停止する。指定した n を複数のストアに分散して成立する事はなく、全てのストアのみに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield `allStoreOf(store`$_1$`, ⋯, store`$_n$`).get1(priority = 0, name = "", canceled = None)`

store$_1$ から store$_n$ の AND 複合ストアオブジェクトを待ち受けるガード式である。全ての store$_i$ ($1 \le i \le n$) から取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield `allStoreOf(store`$_1$`, ⋯, store`$_n$`).put(buffer, priority = 0, name = "", canceled = None)`

store$_1$ から store$_n$ の AND 複合ストアオブジェクトを待ち受けるガード式である。全ての store$_i$ ($1 \le i \le n$) へ追加出来るまで、プロセスの実行を停止する。指定した buffer を複数のストアに分散して成立する事はなく、全てのストアのみに対してのみ成立する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

2.13.3 逐次複合ストア

API store$_1$ >> store$_2$

ストアオブジェクト store$_1$ とストアオブジェクト store$_2$ の逐次複合ストアオブジェクトを作成する。

API yield (store$_1$ >> store$_2$).get(n, priority = 0, name = "", canceled = None)

store$_1$ と store$_1$ の逐次複合ストアオブジェクトを待ち受けるガード式である。まず store$_1$ の待ち受けを開始する。store$_1$ から取得出来たら store$_2$ の待ち受けを開始し、store$_2$ から取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield (store$_1$ >> store$_2$).get1(priority = 0, name = "", canceled = None)

store$_1$ と store$_1$ の逐次複合ストアオブジェクトを待ち受けるガード式である。まず store$_1$ の待ち受けを開始する。store$_1$ から取得出来たら store$_2$ の待ち受けを開始し、store$_2$ から取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield (store$_1$ >> store$_2$).put(buffer, priority = 0, name = "", canceled = None)

store$_1$ と store$_1$ の逐次複合ストアオブジェクトを待ち受けるガード式である。まず store$_1$ の待ち受けを開始する。store$_1$ へ追加出来たら store$_2$ の待ち受けを開始し、store$_2$ へ追加出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API sequenceStoreOf(store$_1$, store$_2$, ⋯, store$_n$)

ストアオブジェクト store$_1$ からストアオブジェクト store$_n$ の逐次複合ストアオブジェクトを作成する。

API yield sequenceStoreOf(store$_1$, ⋯, store$_n$).get(n, priority = 0, name = "", canceled = None)

store$_1$ から store$_n$ の逐次複合ストアを待ち受けるガード式である。まず store$_1$ の待ち受けを開始する。store$_1$ から取得出来たら store$_2$ の待ち受けを開始する。同様に store$_i$ $(1 \leq i \leq n-1)$ から取得出来たら、store$_{i+1}$ の待ち受けを開始する。store$_n$ が取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield `sequenceStoreOf(`store$_1$`, ···, `store$_n$`).get1(priority = 0, name = "", canceled = None)`

store$_1$ から store$_n$ の逐次複合ストアを待ち受けるガード式である。まず store$_1$ の待ち受けを開始する。store$_1$ から取得出来たら store$_2$ の待ち受けを開始する。同様に store$_i$ $(1 \leq i \leq n-1)$ から取得出来たら、store$_{i+1}$ の待ち受けを開始する。store$_n$ が取得出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。canceled には 1 引数の関数、あるいは、None を指定する。None でない場合、複合待ち受けにおいて、この待ち受け操作が完了した後にその待ち受けパスがキャンセル (2.9.4 章) された場合、待ち受け結果を引数として canceled ハンドラが呼ばれる。

API yield `sequenceStoreOf(`store$_1$`, ···, `store$_n$`).put(buffer, priority = 0, name = "", canceled = None)`

store$_1$ から store$_n$ の逐次複合ストアを待ち受けるガード式である。まず store$_1$ の待ち受けを開始する。store$_1$ へ追加出来たら store$_2$ の待ち受けを開始する。同様に store$_i$ $(1 \leq i \leq n-1)$ へ追加出来たら、store$_{i+1}$ の待ち受けを開始する。store$_n$ へ追加出来るまで、プロセスの実行を停止する。優先度は priority、ガード名は name となる。

2.13.4 結合優先度

結合優先度は Python の演算子の優先度に従う。つまり、優先度の高い順に、>> , & , | になる。例えば、a, b, c, d, e はストアとすると、

コード例
```
yield (a >> b | c >> d & e >> f).get1()
```

は、

コード例
```
yield ((a >> b) | ((c >> d) & (e >> f))).get1()
```

と解釈される。

2.14 連続シミュレーション

ここで言う連続シミュレーションとは、常微分方程式（ODE, Ordinary Differential Equation）の初期値問題（IVP, Initial Value Problem）の数値解法の事を指し、時間 t_0 で、y が y_0 であった場合に、$y = y(t) \quad (t_0 \leq t \leq T)$ の近似解を求める問題である。

$$\begin{aligned} \frac{dy}{dt} &= f(t, y) \quad (t_0 \leq t \leq T) \\ y(t_0) &= y_0 \end{aligned}$$

k 階の常微分方程式は以下のように表される。

$$\begin{aligned} y^{(k)} &= f(t, y, y', y'', \cdots, y^{(k-1)}) \\ y(t_0) &= y_0 \\ y'(t_0) &= y'_0 \\ y''(t_0) &= y''_0 \\ \vdots &= \vdots \\ y^{(k-1)}(t_0) &= y_0^{(k-1)} \end{aligned}$$

ここで、$y = y^0, y' = y^1, \cdots, y^{(k-1)} = y^{k-1}$ と置けば、次のような 1 階常微分方程式に変換する事が出来る。

$$\begin{aligned} (y^{k-1})' &= f(t, y^0, y^1, y^2, \cdots, y^{k-1}) \\ (y^{k-2})' &= y^{k-1} \\ \vdots &= \vdots \\ (y^1)' &= y^2 \\ (y^0)' &= y^1 \end{aligned}$$

連続シミュレーションの実行（2.14.3 章）では、1 階常微分方程式のみを指定可能なので、高階常微分方程式の場合はこのような変換を手動で行う必要がある。

2.14.1 連続変数の定義

API `Value(buffer = 0, lower = None, upper = None, name = "", monitor = False, item = None, localid = None)`

初期値が buf で、最小値が lower で、最大値が upper で、名前が name の連続変数オブジェクトを作成する。最小値が None の場合は最小値は設定されない。最大値が None の場合は最大値は設定されない。monitor が真の場合は、変数値の時系列変化を時系列モニターに記録する。localid が None の場合、この連続変数はグローバル変数扱いになり、localid が None でなかった場合、この連続変数はローカル変数扱いになる。ローカル変数は、continuous 待ち受けで特殊な処理が行われる。ローカル変数の場合、item 引数にフローアイテムを設定する事で、そのローカル変数にフローアイテムを結びつける事が出来る。

2.14.2 補助変数の定義

API `Expression(args, expr, name = "", monitor = False)`

名前が name の補助変数オブジェクトを作成する。補助変数とは、複数の連続変数の関数として表現される変数である。args には、補助変数が参照する連続変数のリストを指定する。expr はスカラー値を返す関数であり、第 1 引数は、必ずシミュレーション時間を指す。第 2 引数以降は、args で指定した連続変数オブジェクトの値が渡される。つまり引数の数は、1 + len(args) となる。monitor が真の場合は、変数値の時系列変化を時系列モニターに記録する。

コード例

```
a = Value(3.0, name = "a")
b = Value(4.0, name = "b")
c = Value(5.0, name = "c")
e = Expression([a, b, c],
               lambda t, a, b, c: a + b * c,
               name = "e")
```

コード例

```
import math
def calc(t, a, b, c):
    return math.sin(a) + math.exp(b) * c
a = Value(3.0, name = "a")
b = Value(4.0, name = "b")
c = Value(5.0, name = "c")
e = Expression([a, b, c],
               calc,
               name = "e")
```

2.14.3 連続シミュレーションの実行

API yield `continuous(vars, args, f, name = "")`

連続シミュレーションを行うガード式である。vars には更新する連続変数のリストを指定する。args には参照する連続変数あるいは補助変数のリストを指定する。f には、vars の各変数の導関数を指定する。第 1 引数は、必ずシミュレーション時間を指す。第 2 引数以降には、vars で指定した連続変数の値、続いて、args で指定した連続変数あるいは補助変数の値が続く。ただし、i 番目の連続変数 vars[i] がローカル変数であった場合、同一の localid であるローカル変数値からなるリストが渡され、それに次いで、そのローカル変数のそのリスト上でのインデックス番号が渡される。つまり引数の数は、ローカル変数の個数を len(localvars) とすると、1 + len(vars) + len(localvars) + len(args) となる。返り値は、要素数が len(vars) 個のリストであり、それぞれが、vars の各変数の微係数である。vars で指定した変数のうちひとつでも、上下限に達した場合は、その時点で連続シミュレーションを停止する。ガード名は、name となる。発火値として、連続シミュレーション結果オブジェクトを返す。f に渡されるローカル変数は、同一の localid であるローカル変数値からなるリストであるが、「ローカル変数.items」は特殊なリストを返し、各ローカル変数に割当てられているフローアイテムのリストである。

以下のような 3 変数の連立常微分方程式を考える。(伝染病のモデルとして有名な SIR モデル)

$$
\begin{aligned}
\frac{dS}{dt} &= -\beta SI \\
\frac{dI}{dt} &= \beta SI - \gamma I \\
\frac{dR}{dt} &= \gamma I
\end{aligned}
$$

それぞれの、時間 0 における値を以下とする。

$$
\begin{aligned}
S(0) &= 999 \\
I(0) &= 1
\end{aligned}
$$

$$R(0) = 0$$

このような微分方程式をシミュレーションするコードは以下のように表現する事が出来る。

コード例

```
S = Value(999, name = "S", monitor = True)
I = Value(1, name = "I", monitor = True)
R = Value(0, name = "R", monitor = True)
def proc():
    beta = 0.001
    gamma = 0.2
    yield continuous([S, I, R],
                     [],
                     lambda T, S, I, R: (
                         -beta * S * I,
                         beta * S * I - gamma * I,
                         gamma * I))
activate(proc)()
start()
```

次に、タンカーモデルを考える。石油タンクが平均 10 の指数分布間隔で港に到着し、それぞれタンカーには固有のタンク容量と、注入速度が与えられている。港には石油タンクがあり、タンカーが到着すると、石油を与えられている注入速度で注入する。このようなモデルは以下のように記述する事が出来る。

コード例

```
initialize()

tank = Value(0.0, lower = 0.0, monitor = True)

class Tanker:
    def __init__(self):
        self.capacity = next(exponentialDistribution(
                                     mean = 10.0))
        self.velocity = next(exponentialDistribution(
                                     mean = 1.0))

def tanker():
    gen = exponentialDistribution(mean = 10.0)
    while True:
        yield pause(next(gen))
        yield subactivate(putting)(Tanker())

def putting(tanker):
    localtank = Value(tanker.capacity,
                      lower = 0.0,
                      upper = tanker.capacity)
    yield continuous([tank, localtank],
                     [],
                     lambda t, tank, lockaktank: (
                         tanker.velocity,
                         -tanker.velocity))

activate(tanker)()
start(until = 100)
```

この例では、動的に作成される一時的な変数 `localtank` が出現している。しかしながら、ローカル変数としては定義していない。

次に、基本的には前述のモデルと同様だが、港の石油タンクに注入する速度には上限があり、上限を越えた場合は、各注入速度が按分されるようなモデルを考える。そのようなモデルは以下のように、ローカル変数を使う事で容易に記述する事が出来る。

コード例

```
initialize()
setGlobalSeed(0)

tank = Value(0.0, lower = 0.0, monitor = True)

class Tanker:
    def __init__(self):
        self.capacity = next(exponentialDistribution(
                                 mean = 10.0))
        self.velocity = next(exponentialDistribution(
                                 mean = 1.0))

def tanker():
    gen = exponentialDistribution(mean = 10.0)
    while True:
        yield pause(next(gen))
        yield subactivate(putting)(Tanker())

def putting(tanker):
    localtank = Value(tanker.capacity,
                      lower = 0.0,
                      upper = tanker.capacity,
                      localid = "localtank",
                      item = tanker)
    def calc(t, tank, localtank, i):
        maxv = 2.0
        s = sum([item.velocity for item in localtank.items])
        if s >= maxv:
            return (tanker.velocity * maxv / s,
                    -tanker.velocity * maxv / s)
        else:
            return (tanker.velocity,
                    -tanker.velocity)
    yield continuous([tank, localtank],
                     [],
                     calc)

activate(tanker)()
start(until = 100)
```

この例では、localtank は、ローカル変数と定義している。そのため、calc の第3引数 localtank は、その時点で注入している全てのタンカーの石油残量のリストを示しており、その中の i 番目の値が、そのタンカーの石油残量を示している。localtank.items は、個々のタンカーの特性を示す Tanker オブジェクトのリストを示している。つまり、[item.velocity for item in localtank.items] は注入速度のリストを示し、sum([item.velocity for item in localtank.items]) は、注入速度の合計値を示す。

2.14.4 連続シミュレーションの状態待ち受け

API yield continuousWait(args, until, name = "")

連続シミュレーション中の状態を待ち受けるガード式である。args には参照する連続変数あるいは補助変数のリストを指定する。until には、スカラー値を返す関数を指定する。この関数値が正になると、このガード式は発火する。until の第1引数は、必ずシミュレーション時間を指す。第2引数以降には、args で指定した連続変数を指定する。ただし、i 番目の連続変数 vargs[i] がローカル変数であった場合、同一の localid であるローカル変数値からなるリストが渡され、それに次いで、そのローカル変数のそのリスト上でのインデックス番号が渡される。つまり引数の数は、ローカル変数の個数を len(localargs) とすると、1 + len(args) + len(localargs) となる。ガード名は、name となる。

コード例

```
S = Value(999, name = "S", monitor = True)
I = Value(1, name = "I", monitor = True)
R = Value(0, name = "R", monitor = True)
def proc():
    beta = 0.001
    gamma = 0.2
    yield (continuous([S, I, R],
                      [],
                      lambda T, S, I, R: (
                          -beta * S * I,
                          beta * S * I - gamma * I,
                          gamma * I)) |
           continuousWait([I, R],
                          lambda T, I, R: I + R - v))
activate(proc)()
start()
```

2.14.5 連続シミュレーションの結果

API `continuousResult.upperbound(value)`

連続変数 `value` が上限に達した為に、連続シミュレーションが停止した場合に真となる。

API `continuousResult.lowerbound(value)`

連続変数 `value` が下限に達した為に、連続シミュレーションが停止した場合に真となる。

2.14.6 連続変数の時系列モニター

API `value.monitor`

連続変数オブジェクトの時系列変化を記録した時系列モニターを返す。連続変数オブジェクトで時系列モニターを記録した場合のみ、参照可能となる。列数は `1` であり、連続変数値を示す。

2.14.7 補助変数の時系列モニター

API `expression.monitor`

補助変数オブジェクトの時系列変化を記録した時系列モニターを返す。補助変数オブジェクトで時系列モニターを記録した場合のみ、参照可能となる。列数は 1 であり、補助変数値を示す。

2.14.8 常微分方程式ソルバー

常微分方程式 (Ordinary Differential Equation, ODE) ソルバーとしては、以下が用意されている。これらは、2.1.3 章の start (シミュレーションの開始 API) の odesolver 引数に指定する事が出来る。

API `dopri54Solver(dt = 1.0e-1, rtol = 1e-3, atol = 1e-6)`

Dormand-Prince の 5(4) 次の陽的 Runge-Kutta 法によるソルバー。dt は時間刻み幅。rtol は相対許容誤差。atol は絶対許容誤差。

API `dop853Solver(dt = 1.0e-1, rtol = 1e-3, atol = 1e-6)`

Dormand-Prince の 8(5,3) 次の陽的 Runge-Kutta 法によるソルバー。dt は時間刻み幅。rtol は相対許容誤差。atol は絶対許容誤差。

API `adamsSolver(dt = 1.0e-1, rtol = 1e-3, atol = 1e-6)`

Adams 法によるソルバー (non-stiff な問題用)。dt は時間刻み幅。rtol は相対許容誤差。atol は絶対許容誤差。

API `bdfSolver(dt = 1.0e-1, rtol = 1e-3, atol = 1e-6)`

後退差分法 (BDF) 法によるソルバー (stiff な問題用)。dt は時間刻み幅。rtol は相対許容誤差。atol は絶対許容誤差。

2.15 条件式

2.15.1 恒真

API yield `alwaysTrue()`

常に成立するガード式である。

2.15.2 恒偽

API yield `alwaysFalse()`

常に成立しないガード式である。

2.15.3 条件判定

API yield `case(cond, name = "")`

条件式 `cond` を評価し、真なら成立し、偽なら成立しないガード式である。評価は待ち受け依頼時に一度だけ行われる。評価後に、条件が変わっても発火する事はない。もし、待ち受け依頼時と、待ち受け実行時で条件の真偽が変わる可能性がある場合は、2.15.4 章のガード式を利用する。

2.15.4 遅延条件判定

API yield `delayedCase(condFunc, name = "")`

判定関数 `condFunc` を評価し、真なら成立し、偽なら成立しないガード式である。判定関数の評価は待ち受け実行時に一度だけ行われる。評価後に、条件が変わっても発火する事はない。待ち受け依頼時と、待ち受け実行時で条件の真偽が変わる可能性がない場合は、2.15.3 のガード式を利用する事も出来る。

2.16 プロセス制御

プロセスの状態には、活性状態（*active*）、一時停止状態（*suspended*）、終了状態（*terminated*）がある。活性状態のサブ状態として、実行待ち状態（*waiting*）、実行状態（*running*）がある。実行待ち状態とは、まだプロセスの起動条件を満たしていない状態であり、起動条件を満たすと自動的に実行状態に遷移する。

プロセスの状態によって、終了（`terminate`）、一時停止（`suspend`）、再開（`resume`）する事が出来る。定義されていない遷移はエラーとなる。

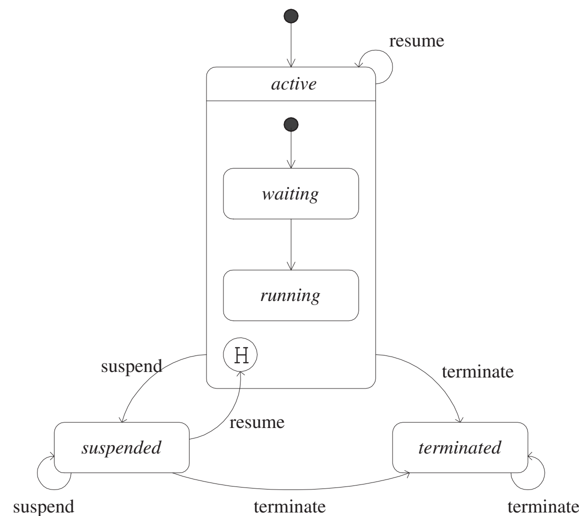

図 2.1: プロセスの状態遷移

プロセスの終了とは、そのプロセスが実行している全ての待ち受けが削除され、また、そのプロセスに登録されている全てのハンドラが削除される。サブプロセスも終了する。終了したプロセスは再開する事は出来ない。

プロセスの一時停止とは、そのプロセスが実行している全ての待ち受けが一時停止し、また、そのプロセスに登録されている全てのハンドラも一時停止する。サブプロセスも一時停止する。

プロセスの再開とは、一時停止しているプロセスの一時停止状態の全ての待ち受けが再開し、また、一時停止しているプロセスの一時停止状態の全てのハンドラが再開する。サブプロセスも再開する。実行待ち状態で、一時停止していた場合は、再開すると、実行待ち状態に戻る。実行状態で、一時停止していた場合は、再開すると、実行状態に戻る。

2.16.1 プロセスの状態取得

API `cont.isWaiting()`

プロセスの継続オブジェクト `cont` が実行待ち状態かどうかを調べる。

API `cont.isRunning()`

プロセスの継続オブジェクト `cont` が実行状態かどうかを調べる。

API `cont.isActive()`

プロセスの継続オブジェクト `cont` が活性状態かどうかを調べる。

API `cont.isSuspended()`

プロセスの継続オブジェクト `cont` が一時停止状態かどうかを調べる。

API `cont.isTerminated()`

プロセスの継続オブジェクト `cont` が終了状態かどうかを調べる。

2.16.2 プロセスの終了待ち受け

API yield `cont.terminated()`

プロセスの継続オブジェクト `cont` の終了を待ち受けるガード式である。対象プロセスが終了状態以外から終了状態に遷移した時に成立する。待ち受け開始時に終了状態であった場合は成立しない。外部プロセスが `cont.terminate(val)` を実行して終了した場合は、発火値は `val` となる。ただし、自プロセスの終了を待ち受ける事は出来ない。自プロセスの終了を待ち受けるには、2.16.10 章のハンドラ登録を利用する。

2.16.3 プロセスの一時停止待ち受け

API yield `cont.suspended()`

プロセスの継続オブジェクト cont の一時停止を待ち受けるガード式である。対象プロセスが一時停止状態以外から一時停止状態に遷移した時に成立する。待ち受け開始時に一時停止状態であった場合は成立しない。外部プロセスが cont.suspend(val) を実行して一時停止した場合は、発火値は val となる。ただし、自プロセスの一時停止を待ち受ける事は出来ない。自プロセスの一時停止を待ち受けるには、2.16.11 章のハンドラ登録を利用する。

2.16.4 プロセスの再開待ち受け

API yield `cont.resumed()`

プロセスの継続オブジェクト cont の再開を待ち受けるガード式である。対象プロセスが活性状態以外から活性状態に遷移した時に成立する。待ち受け開始時に活性状態であった場合は成立しない。外部プロセスが cont.resume(val) を実行して再開した場合は、発火値は val となる。ただし、自プロセスの再開を待ち受ける事は出来ない。自プロセスの再開を待ち受けるには、2.16.12 章のハンドラ登録を利用する。

2.16.5 プロセスの終了

API `cont.terminate(val = None)`

プロセスの継続オブジェクト cont を終了する。既に終了状態であった場合は何もしない。また、自プロセスを終了させる事は出来ない。自プロセスを終了するには、2.16.8 章のガード式を利用する。終了する前に、発火値を val として、対象プロセスの終了ハンドラが実行され、対象プロセスの終了を待ち受けているガード式が発火する。

2.16.6 プロセスの一時停止

API `cont.suspend(val = None)`

活性状態のプロセスの継続オブジェクト cont を一時停止する。既に一時停止状態であった場合は何もしない。また、自プロセスを一時停止させる事は出来ない。自プロセスを一時停止するには、2.16.9 章のガード式を利用する。一時停止する前に、発火値を val として、対象プロセスの一時停止ハンドラが実行され、対象プロセスの一時停止を待ち受けているガード式が発火する。

2.16.7 プロセスの再開

API `cont.resume(val = None)`

一時停止状態のプロセスの継続オブジェクト cont を再開する。既に活性状態であった場合は何もしない。再開止する前に、発火値を val として、対象プロセスの再開ハンドラが実行され、対象プロセスの再開を待ち受けているガード式が発火する。

2.16.8 自プロセスの終了

API yield `terminate(val = None)`

現在実行中のプロセスを終了するガード式である。終了する前に、発火値を val として、対象プロセスの終了ハンドラが実行され、対象プロセスの終了を待ち受けているガード式が発火する。

2.16.9 自プロセスの一時停止

API yield `suspend(val = None)`

現在実行中のプロセスを一時停止するガード式である。他プロセスが再開させるまでプロセスの実行を停止する。一時停止する前に、発火値を `val` として、対象プロセスの一時停止ハンドラが実行され、対象プロセスの一時停止を待ち受けているガード式が発火する。

2.16.10 プロセスの終了ハンドラの登録

API `cont.setTerminatedHandler(handler)`

プロセスの継続オブジェクト `cont` の終了ハンドラを設定する。プロセスが終了状態以外から終了状態に遷移した時に `handler` が実行される。`handler` は 1 引数の関数で、プロセスが終了した時に、発火値の設定された単一の待ち受け結果を引数として呼出される。

2.16.11 プロセスの一時停止ハンドラの登録

API `cont.setSuspendedHandler(handler)`

プロセスの継続オブジェクト `cont` の一時停止ハンドラを登録する。プロセスが一時停止状態以外から一時停止状態に遷移した時に `handler` が実行される。`handler` は 1 引数の関数で、プロセスが一時停止した時に、発火値の設定された単一の待ち受け結果を引数として呼出される。

2.16.12 プロセスの再開ハンドラの登録

API `cont.setResumedHandler(handler)`

プロセスの継続オブジェクト `cont` の再開ハンドラを登録する。プロセスが活性状態以外から活性状態に遷移した時に `handler` が実行される。`handler` は 1 引数の関数で、プロセスが再開した時に、発火値の設定された単一の待ち受け結果を引数として呼出される。

2.16.13 自プロセスの継続オブジェクトの取得

API yield `currentProc()`

自プロセスの継続オブジェクトを取得するガード式である。
待ち受け結果は自プロセスの継続オブジェクトとなる。

以下のコードは同じ動作をする。

コード例

```
def handler(result):
    print("terminated")
def proc():
    cont = yield currentProc()
    cont.setTerminatedHandler(handler)
    yield pause(100)
activate(proc)()
start()
```

コード例

```
def handler(result):
    print("terminated")
def proc():
    yield pause(100)
activate(proc, terminatedHandler = handler)()
start()
```

コード例

```
def handler(result):
    print("terminated")
def proc():
    yield pause(100)
cont = activate(proc)()
cont.setTerminatedHandler(handler)
start()
```

2.16.14 プロセス制御の例

コード例

```
initialize()

def printMessage(msg):
    print(msg)
def terminatedHandler(result):
    printMessage("terminated %s at %s" % (result.val, now()))
def suspendedHandler(result):
    printMessage("suspended %s at %s" % (result.val, now()))
def resumedHandler(result):
    printMessage("resumed %s at %s" % (result.val, now()))
def proc1():
    yield pause(1000)
def proc2(p):
    yield (p.terminated() >> runAction(printMessage)(
           "terminated signal was observed by proc2") |
           p.suspended() >> runAction(printMessage)(
           "suspended signal was observed by proc2") |
           p.resumed() >> runAction(printMessage)(
           "resumed signal was observed by proc2"))
    yield go(proc2)(p)
def proc3(p):
    for i in range(3):
        p.suspend("by proc3")
        yield pause(3)
        p.resume("by proc3")
        yield pause(3)
    p.terminate("by proc3")
p = activate(proc1)()
p.setTerminatedHandler(terminatedHandler)
p.setSuspendedHandler(suspendedHandler)
p.setResumedHandler(resumedHandler)
activate(proc2)(p)
activate(proc3)(p)
start()
```

2.17 サブプロセス

サブプロセスとは、プロセス内から再帰的に呼び出すプロセスの事である。

2.17.1 サブプロセスの定義

サブプロセスは通常のプロセスと同様に定義する事が出来る。サブプロセスの実行を OR 複合待ち受けと併用して利用する場合で、サブプロセスの終了より先に、他の OR 複合待ち受けが成立した場合、サブプロセスの継続オブジェクトに対し、終了シグナルを送り、サブプロセスを呼出した親プロセスは継続の実行を進める。そのため、OR 複合待ち受けと併用して利用する場合、サブプロセスは終了される事を想定しておく必要がある。

コード例

```
def proc():
    ...
    yield call(subProc)() | cancelEvet.wait()
    ...

def subProc()
    for count in range(10):
        yield pause(10)
        yield event.wait()
        print(count)

activate(proc)()

start()
```

2.17.2 サブプロセスの実行

API yield `call(process, delay = 0, priority = 0, name = "",terminatedHandler = nopHandler, suspendedHandler = nopHandler,resumedHandler = nopHandler) (*args, **keys)`

サブプロセスを起動し、サブプロセスの完了を待ち受けるガード式である。シミュレーション時間 `delay` 後に、プロセス名 `name` のサブプロセス `process` を実行し、その完了を待ち受ける。`delay` が指定された時のみ、優先度 `priority` は有効となり、同時刻の起動優先度を示す。終了、一時停止、再開ハンドラをそれぞれ、`terminatedHandler`, `suspendedHandler`, `resumedHandler` で指定する事が出来る。各ハンドラは、1 引数の関数で、プロセス状態が変化した時に、発火値の設定された単一の待ち受け結果を引数として呼出される。ハンドラを明示的に指定しなかった場合は、空のハンドラが設定される。(`*args,**keys`) に書かれた引数は全て `process` に送られる。なお、引数が評価されるのは、待ち受け実行時ではなく、待ち受け依頼時である。サブプロセスは他のガード式と同様に、複合待ち受けの一部として指定された場合、並行に待ち受けを行う。

API yield subactivate(process, delay = False, cond = False, priority =0, name = "", terminatedHandler = nopHandler, suspendedHandler =nopHandler, resumedHandler = nopHandler)(*args, **keys)

サブプロセスを起動するガード式である。このガード式自体は即座に成立する。起動遅延時間 delay とガード式 cond はオプショナルであるが、両方を指定した場合はエラーとなる。delay が指定された場合は、起動を指定時間遅らせる。cond が指定された場合、まず cond の成立を待ち受ける。何れも指定されなかった場合は、起動前の待ち受けは行われない。起動条件成立後、プロセス名 name のサブプロセス process を実行する。delay が指定された時のみ、優先度 priority は有効となり、同時刻の起動優先度を示す。終了、一時停止、再開ハンドラをそれぞれ、terminatedHandler,suspendedHandler, resumedHandler で指定する事が出来る。各ハンドラは、1 引数の関数で、プロセス状態が変化した時に、発火値の設定された単一の待ち受け結果を引数として呼出される。ハンドラを明示的に指定しなかった場合は、空のハンドラが設定される。(*args, **keys) に書かれた引数は全て process に送られる。なお、引数が評価されるのは、待ち受け実行時ではなく、待ち受け依頼時である。待ち受け結果はサブプロセスの継続オブジェクトである。サブプロセスを実行したプロセスが終了したら、サブプロセスも終了する。

2.17.3 複合サブプロセス

単一の call 待ち受けを実行した場合、サブプロセスが完了するまで、親プロセスの実行が停止する。

複合待ち受けの一部に call 待ち受けが含まれていた場合、他の待ち受けと同様に、並行観測を行う。つまり、複数の call 待ち受けが、ガード式に含まれていた場合、一時的に複数のサブプロセスが起動する事になる。

OR 複合サブプロセス

複数の `call` を `OR` 結合した場合、複数のプロセスが起動する。それぞれのプロセスは独立に処理を進めるが、あるプロセスが終了したタイミングで、ガード式は成立し、他のプロセスには終了シグナルが送られる。

AND 複合サブプロセス

複数の `call` を `AND` 結合した場合、複数のプロセスが起動する。それぞれのプロセスは独立に処理を進めるが、全てのプロセスが終了したタイミングで、ガード式は成立する。

逐次複合サブプロセス

複数の `call` を逐次結合した場合、順番にプロセスが起動する。全てのプロセスが終了したタイミングで、ガード式は成立する。

2.18 状態遷移

2.18.1 状態遷移の実行

API yield `go(process, delay = 0, priority = 0, name ="")(*args,**keys)`

現在の実行中のプロセスを、指定されたプロセスに置き換えるガード式である。現在の実行中のプロセスは終了され、シミュレーション時間 delay 後に、サブプロセス process を実行する。ガード名は name となる。delay が指定された時のみ、優先度 priority は有効となり、同時刻の起動優先度を示す。go はスタックを消費しない。(*args, **keys) に書かれた引数は全て process に送られる。なお、引数が評価されるのは、待ち受け実行時ではなく、待ち受け依頼時である。ただし、移動するタイミングは、go が含まれるガード式全体が成立した時である。go は状態遷移を記述するために利用される。例えば、yield R1 >> go(p1)() | R2 >> go(p2)() とした場合、R1 が成立すれば p1 に遷移し、R2 が成立すれば p2 に遷移する。なお、yield R1 >> go(p1)() & R2 >> go(p2)() や yieldgo(p1) >> go(p2) はエラーとはならないが、go がガード式の末尾以外 (2.18.5 章参照) で利用された場合の動作は未定義である。go で遷移しても、直前の待ち受け結果 (2.23 章参照) は保存され、遷移先で、yield last() を実行すれば、go を含めた待ち受け結果が得られる。

以下のような例を考える。

> 電話オペレーターは電話がなると応答する。応答には 3 分を要する。しかし応答中に何らかの緊急割り込み (上司からの要求かもしれないし、腹痛かもしれない) が発生する場合がある。すると、1 分間の処理時間を要する。処理が終わったら、残りの応答を行い、電話待ち状態に戻る。しかしながら、割り込み処理の最中に電話が切られてしまう可能性もある。その場合は、電話待ち状態に戻る。

この例の状態遷移図を書くと図 2.2 のようになる。
それを状態遷移の実行 go を使って書くと以下のようになる。

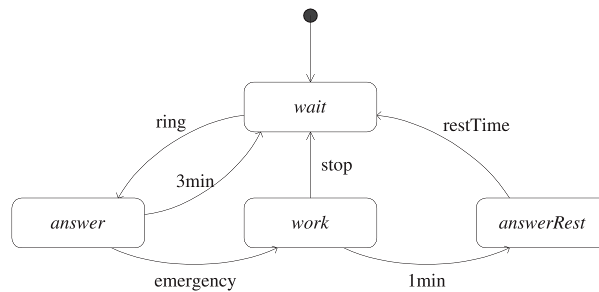

図 2.2: 状態遷移図

コード例

```
ring = Event()
stop = Event()
emergency = Event()

class TelephoneOperator:
    def wait(self):
        print("state: wait at %d" % now())
        yield ring.wait() >> go(self.answer)()
    def answer(self):
        print("state: answer at %d" % now())
        start = now()
        yield (pause(3) >> go(self.wait)() |
               emergency.wait() >> go(self.work)(start))
    def work(self, start):
        print("state: work at %d" % now())
        rest = 3 - (now() - start)
        yield (stop.wait() >> go(self.wait)() |
               pause(1) >> go(self.answerRest)(rest))
    def answerRest(self, rest):
        print("state: answerRest at %d" % now())
        yield pause(rest) >> go(self.wait)()

def proc():
    yield pause(1)
    ring.signal()
    yield pause(1)
    emergency.signal()

operator = TelephoneOperator()
activate(operator.wait)()
activate(proc)()
start()
```

ただし、後述のサブ状態と絡むが、以下のように、状態遷移はプロセスメソッド内に内包してしまう記法を推奨する。self の記述を最低限に抑え、状態遷移のみに着目してモデリングする事が出来る。状態遷移内でのメンバー参照は、「self. メンバー名」で可能である。

コード例

```
class TelephoneOperator:
    def office(self):
        def wait():
            print("state: wait at %d" % now())
            yield ring.wait() >> go(answer)()
        def answer():
            print("state: answer at %d" % now())
            start = now()
            yield (pause(3) >> go(wait)() |
                   emergency.wait() >>
                       go(work)(start))
        def work(start):
            print("state: work at %d" % now())
            rest = 3 - (now() - start)
            yield (stop.wait() >> go(wait)() |
                   pause(1) >> go(answerRest)(rest))
        def answerRest(rest):
            print("state: answerRest at %d" % now())
            yield pause(rest) >> go(wait)()
        yield go(wait)()
```

2.18.2 サブ状態

ここで、先の電話オペレーターの状態の更に上位の状態として、出社状態、帰宅状態を考える。それぞれ 1000 分で状態が切り替わるとする。出社状態はコンポジット状態であり、その中には、先の状態遷移が内包されている。それらをサブ状態と呼ぶ。

このようなサブ状態を含む状態遷移は以下のようにして記述する事が出来る。

メインの状態として、office と home があり、相互に切り替わる。office 状態のサブ状態の起動には call を利用すれば良い。

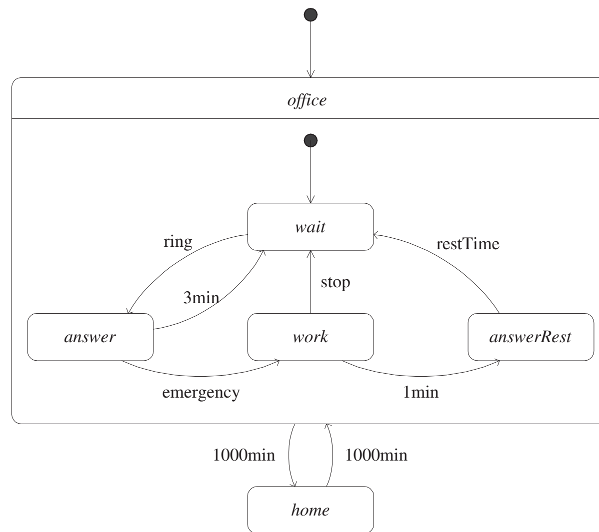

図 2.3: 状態遷移図 (サブ状態)

コード例

```
class TelephoneOperator:
    def start(self):
        def office():
            def wait():
                print("state: office.wait at %d" % now())
                yield ring.wait() >> go(answer)()
            def answer():
                print("state: office.answer at %d" % now())
                start = now()
                yield (pause(3) >> go(wait)() |
                       emergency.wait() >>
                           go(work)(start))
            def work(start):
                print("state: office.work at %d" % now())
                rest = 3 - (now() - start)
                yield (stop.wait() >> go(wait)() |
                       pause(1) >> go(answerRest)(rest))
            def answerRest(rest):
                print("state: office.answerRest at %d" %
                      now())
                yield pause(rest) >> go(wait)()
            print("state: office at %d" % now())
            yield (call(wait)() |
                   pause(1000) >> go(home)())
        def home():
            print("state: home at %d" % now())
            yield pause(1000) >> go(office)()
        yield go(office)()
```

2.18.3 並行状態

ここで、電話オペレーターの出社状態では、通常の業務を行うと共に、400 分単位で、スキルが向上するとする。このような並行して動作する状態遷移は並行状態と呼ぶ。

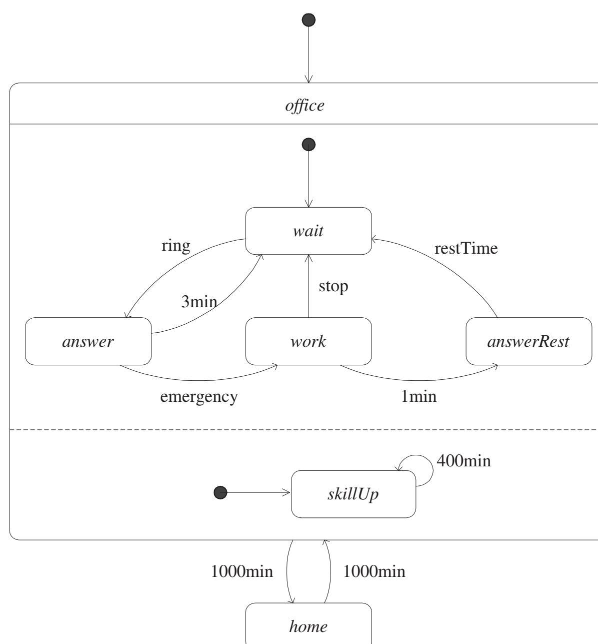

図 2.4: 状態遷移図 (並行状態)

このような並行状態は以下のようにして記述する事が出来る。

通常の業務とスキル向上は並行して進むプロセスなので、`call` を使って、`OR` 並列プロセスとして起動すれば良い。(この例の場合、無限ループなので、`AND` 並列でも構わない)

コード例

```
class TelephoneOperator:
    def start(self):
        def office():
            def wait():
                print("state: office.wait at %d" % now())
                yield ring.wait() >> go(answer)()
            def answer():
                print("state: office.answer at %d" % now())
                start = now()
                yield (pause(3) >> go(wait)() |
                       emergency.wait() >>
                           go(work)(start))
            def work(start):
                print("state: office.work at %d" % now())
                rest = 3 - (now() - start)
                yield (stop.wait() >> go(wait)() |
                       pause(1) >> go(answerRest)(rest))
            def answerRest(rest):
                print("state: office.answerRest at %d" %
                      now())
                yield pause(rest) >> go(wait)()
            def skillUp():
                print("state: office.skillUp at %d" %
                      now())
                yield pause(400) >> go(skillUp)()
            print("state: office at %d" % now())
            yield (call(wait)() |
                   call(skillUp)() |
                   pause(1000) >> go(home)())
        def home():
            print("state: home at %d" % now())
            yield pause(1000) >> go(office)()
        yield go(office)()
```

2.18.4 履歴

先の例と同じ例を考えるが、出社状態に戻った時は以前の業務の続きから行うとする。スキル向上も総業務時間で発生するとする。状態を抜ける時に

履歴を残しておき、再度その状態に戻った時に、その履歴と同じ状態に復帰するというモデリングである。

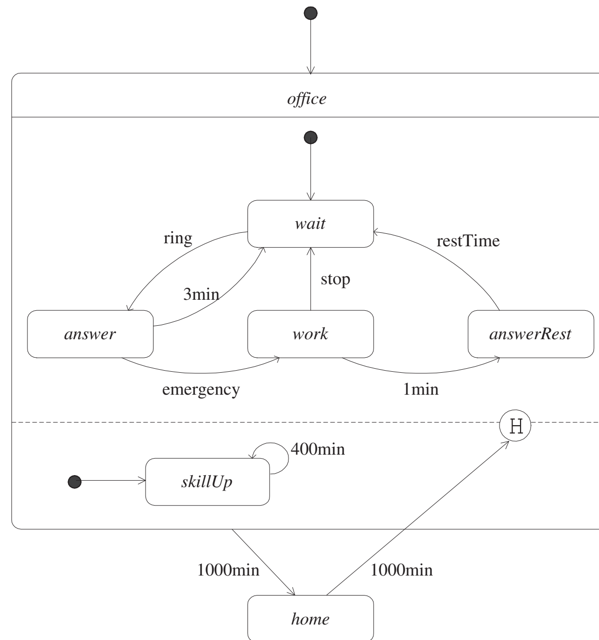

図 2.5: 状態遷移図 (履歴)

このような並行状態は以下のようにして記述する事が出来る。

履歴を実現するには、`go` と `call` だけでは実現出来ない。履歴については、プロセスの `suspend/resume` を利用する事が出来る。

コード例

```
class TelephoneOperator:
    def start(self):
        def officeWork():
            def wait():
                print("state: office.wait at %d" % now())
                yield ring.wait() >> go(answer)()
            def answer():
                print("state: office.answer at %d" % now())
                start = now()
                yield (pause(3) >> go(wait)() |
                       emergency.wait() >>
                           go(work)(start))
            def work(start):
                print("state: office.work at %d" % now())
                rest = 3 - (now() - start)
                yield (stop.wait() >> go(wait)() |
                       pause(1) >> go(answerRest)(rest))
            def answerRest(rest):
                print("state: office.answerRest at %d" %
                      now())
                yield pause(rest) >> go(wait)()
            def skillUp():
                print("state: office.skillUp at %d" %
                      now())
                yield pause(400) >> go(skillUp)()
            yield (call(wait)() |
                   call(skillUp)())
        def office():
            print("state: office at %d" % now())
            self.officeProc.resume()
            yield pause(1000)
            self.officeProc.suspend()
            yield go(home)()
        def home():
            print("state: home at %d" % now())
            yield (pause(1000) >> go(office)())
        self.officeProc = activate(officeWork)()
        yield go(office)()
```

2.18.5 ガード式の末尾

ガード式 X がガード式 Y の末尾である事を tail(X,Y) と書くとすれば、以下のように定義される。

$$
\begin{aligned}
\texttt{tail}(W, W_1 \mid W_2) &= \texttt{tail}(W, W_1) \vee \texttt{tail}(W, W_2) \\
\texttt{tail}(W, W_1 \ \&\ W_2) &= F \\
\texttt{tail}(W, W_1 >> W_2) &= \texttt{tail}(W, W_2) \\
\texttt{tail}(W, W) &= T \\
\texttt{tail}(W, W') &= F \quad (W \neq W')
\end{aligned}
$$

2.19 アクション

2.19.1 アクションの挿入

API yield `runAction(action, name = "")(*args, **keys)`

関数 action を実行するガード式である。action は、任意の関数である。単体で利用した場合、AND/OR 式の中に含めても即座に実行されるので、あまり意味がない。逐次待ち受けの場合に、独自の処理を挿入したい場合などに、利用出来る。(*args, **keys) に書かれた引数は全て action に送られる。なお、引数が評価されるのは、待ち受け実行時ではなく、待ち受け依頼時である。発火値はアクションの実行結果となる。

コード例

```
e1 = Event()
e2 = Event()

def action():
    print("e1 fired")

def proc1():
    ...
    yield sequenceOf(e1.wait(name = "e1"),
                     runAction(action)(),
                     e2.wait(name = "e2"))
    ...

def proc2():
    ...
    e1.signal()
    ...
    e2.signal()
    ...
```

2.20 記録

2.20.1 値の記録

API yield `recordValue(value, name = "")`

値 `value` を発火値とするガード式である。このガード式自体は即座に発火する。

2.20.2 時間の記録

API yield `recordNow(name = "")`

現在のシミュレーション時間を発火値とするガード式である。このガード式自体は即座に発火する。逐次待ち受けの一部で利用した場合、`yield` を実行した時刻ではなく、`recordNow` ガード式が実行された時刻が発火値となる。

2.21 ハンドラ

2.21.1 ハンドラの登録

API yield `addHandler(handler, ガード式)`

指定したガード式のハンドラを登録する。この式自体もガード式であるが、即座に成立する。`handler` は 1 引数の関数で、ガード式が成立したら、待ち受け結果を引数として呼出される。ガード式が成立したら登録されたハンドラは除去される。また、`addHandler` を実行したプロセスが終了した時に、登録されたハンドラは除去される。発火値はハンドラオブジェクトであり、`removeHandler` で除去する事が出来る。

コード例

```
def handler(result):
    if "e1" in result:
       (e1 が発火)
    else:
       (e2 が発火)

e1 = Event()
e2 = Event()

def proc1():
    ...
    yield addHandler(handler,
                     (e1.wait(name = "e1") |
                      e2.wait(name = "e2")))
    ...

def proc2():
    ...
    e1.signal()
    e2.signal()
    ...
```

API `addGlobalHandler(handler, ガード式)`

指定したガード式のグローバルハンドラを登録する。`addHandler` とは違い、この式自体はガード式ではない。`handler` は 1 引数の関数で、ガード式が成立したら、待ち受け結果を引数として呼出される。ガード式が成立したら登録されたハンドラは除去される。`addHandler` と違い、`addGlobalHandler` を実行したプロセスが終了しても、登録されたハンドラは除去されない。返り値はハンドラオブジェクトであり、`removeHandler` で除去する事が出来る。

API `handlerObject.removeHandler()`

登録されたハンドラを除去する。

2.22 待ち受け結果

2.22.1 待ち受け結果の取得

待ち受け結果は、

コード例

```
result = yield ガード式
```

のようにして取得する事が出来る。
`result` は任意の変数であり、成立した待ち受けの結果が含まれている。

2.22.2 待ち受けの成立検査

ガード式に `OR` 待ち受けが含まれていた場合、どちらの待ち受けが成立したのかを取得する事が出来る。ガード式にはガード名を指定出来るが、待ち受けの成立検査を行うには、必ずガード式にガード名を指定しておかなくてはならない。

API `"ガード名" in 待ち受け結果`

待ち受け結果の中に、ガード名で示される待ち受けの結果が含まれていた場合に真を返す。そうでなければ偽を返す。

コード例

```
result = event.wait(name = "fired") | \
         pause(100, name = "timeout")
if "fired" in result:
    (event が発火)
else:
    (timeout が発生)
```

2.22.3 発火値の取得

イベントの発火で val を指定した場合や、ファシリティのロック取得で、個々の待ち受けの発火値を取得する事が出来る。ガード式にはガード名を指定出来るが、発火値の取得を行うには、必ずガード式にガード名を指定しておかなくてはならない。

API 待ち受け結果 ["ガード名"]

待ち受け結果の中に、ガード名で示される待ち受けの結果が含まれていた場合、その発火値を返す。

コード例

```
event = Event()

def proc1():
    result = event.wait(name = "fired")
    print(result["fired"])

def proc2():
    event.signal("fire")
```

コード例

```
def proc3():
    result = yield facility.request(name = "request")
    yield pause(100)
    result["request"].release()
```

2.22.4　ガード式の発火時間

各ガード式の発火時間を得る事が出来る。ガード式にはガード名を指定出来るが、ガード式の発火時間を得るには、必ずガード式に待ち受け名を指定しておかなくてはならない。

API　`待ち受け結果.time("ガード名")`

待ち受け結果の中に、待ち受け名で示される待ち受けの結果が含まれていた場合、その待ち受けが成立したシミュレーション時間を返す。

2.22.5　ガード式の成立順番

待ち受けが複合待ち受けだった場合、ガード式の成立順番を得る事が出来る。ガード式にはガード名を指定出来るが、ガード式の成立順番を得るには、必ずガード式にガード名を指定しておかなくてはならない。

API　`待ち受け結果.index("ガード名")`

待ち受け結果の中に、待ち受け名で示される待ち受けの結果が含まれていた場合、その発火順序を示すインデックスを返す。最後に成立した待ち受けのインデックスは `0` で、インデックスが大きくなる程過去に成立した事を示す。ただし、AND/逐次待ち受けが含まれていた場合、単純に個々の待ち受けが成立した順番を示しているわけではなく、ガード式の成立した順番を示す。

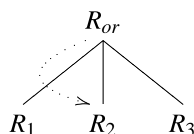

図 2.6: OR 待ち受けの成立順番 (R_2 が発火した場合)

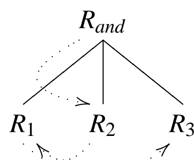

図 2.7: AND 待ち受けの成立順番 (R_3, R_1, R_2 の順で発火した場合)

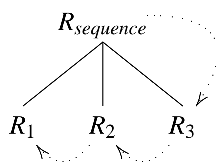

図 2.8: 逐次待ち受けの成立順番

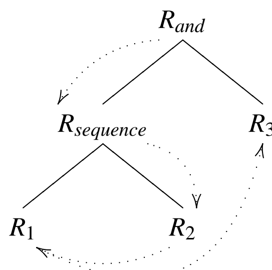

図 2.9: 複合待ち受けの成立順番 (R_1, R_3, R_2 の順で発火した場合)

コード例

```
result = event.wait(name = "fired") & \
         pause(100, name = "timeout")
if result.index("fired") < result.index("timeout"):
    (timeout が発生してから event が発火した)
else:
    (event が発火してから timeout が発生した)
```

2.22.6 単一ガード式の場合の発火値の取得

API 待ち受け結果.val

待ち受けが複合待ち受けでなく、単一待ち受けであった場合、待ち受け結果の val メンバーに、単一待ち受けの発火値が入っている。また、この方法を利用する場合、単一待ち受けのガード名は省略する事が出来る。ただし、複合待ち受けの結果の場合は、この方法は使用出来ない。

コード例

```
event = Event()

def proc1():
    result = event.wait()
    print(result.val)

def proc2():
    event.signal("fire")
```

コード例

```
def proc3():
    result = yield facility.request()
    yield pause(100)
    result.val.release()
```

2.22.7 全てのファシリティロックの解放

API 待ち受け結果.releaseAllFacilities(facility = None)

facility が None なら待ち受け結果に含まれる全てのファシリティロックを解放する。facility が 指定されているなら待ち受け結果に含まれる全ての facility のファシリティロックを解放する。ファシリティロックがひとつも含まれていない場合は何もしない。待ち受け結果に含まれる全てのファシリティロックが解放済みなら True を返す。そうでなければ False を返す。

以下のように使用する事が出来る。

コード例

```
def proc4():
    result = yield ((facility1 & facility2).request() |
                    pause(10, name = "timeout"))
    if "timeout" in result:
        ...
    else:
        ...
    result.releaseAllFacilities()
```

facility1 と facility2 の双方のファシリティロックを取得しようとするが、一定時間取得出来ない場合はタイムアウト処理を行う例であるが、このような場合は、releaseAllFacilities を使わないと、問題が生じる。

この例の場合は、4 つのケースがある。

1. facility1 のロックが時間内に取得出来、その後タイムアウトが発生。
2. facility2 のロックが時間内に取得出来、その後タイムアウトが発生。
3. facility1 と facility2 のロックが時間内に取得出来た。
4. どちらのロックも取得出来ずにタイムアウトが発生。

どのようなケースでも、releaseAllFacilities を実行しておけば、全てのファシリティロックを解放してくれる。

もし、ファシリティロックの解放忘れがあると、永久にファシリティを占有してしまうので、注意が必要である。

あるいは、ファシリティの取得の canceled ハンドラに releaseIfRequested (2.5.5 章) を指定しておく事も出来る。

コード例

```
def proc5():
    result = yield ((facility1 & facility2).request(
                        canceled = releaseIfRequested) |
                      pause(10, name = "timeout"))
    if "timeout" in result:
        (タイムアウトが発生した場合は、)
        (必ず全てのファシリティを自動解放している)
        ...
    else:
        (タイムアウトが発生なかった場合は、)
        (必ず両方のロックを取得している)
        ...
        result.releaseAllFacilities()
```

2.23 直前の待ち受け結果

2.23.1 直前の待ち受け結果の取得

API yield `last()`

直前の待ち受け結果を返すガード式である。直前の待ち受け結果とは、そのプロセスがひとつ前に実行した「yield ガード式」の結果である。得られた結果には 2.22 章の操作が可能である。go で状態遷移した場合も、直前の待ち受け結果は保存される。つまり、go で遷移した直後に yield last() を実行すれば、go を実行したガード式の結果が得られる。

2.24 キュー

キューには、FIFO, PriorityQ, Stack がある。

2.24.1 FIFO キュー

API `FIFO`

先入れ、先出しのキューを示す。First In, First Out の略である。1, 2, 3 の順番で入力された場合、1, 2, 3 で出力する。優先度 priority は無視される。

2.24.2 PriorityQ

API `PriorityQ`

優先度付きキューを示す。入力された順番とは関係なく、優先度 priority の高い順番に出力する。

2.24.3 Stack

API `Stack`

後入れ、先出しのキューを示す。LIFO（Last In, FirstOut）キューとも言う。1, 2, 3 の順番で入力された場合、3, 2, 1 で出力する。優先度 priority は無視される。

2.25 デバッグ

API `psimDebug()`

ガード式の処理内部でエラーが発生した場合、スタックトレースはスケジューラ内部の行数を示すため、ユーザコードのどの部分でエラーが発生したかが分からない。start() を実行する前に psimDebug() を呼んでおけば、スタックトレースにユーザコード上の行数を表示するようになる。

以下のコードは pause の引数にエラーがある。

エラーコード例

```
from psim import *

initialize()

def proc():
    yield pause("abc")

activate(proc)()
start()
```

実行するとエラーが発生しスタックトレースが表示されるが、どこでエラーが発生しているかが分からない。

```
Traceback (most recent call last):
  ...
  File "psim\core.py", line ..., in registerRequest
RuntimeError: unsupported operand type(s) for +: 'float' and 'str'
(try psimDebug() before start() to show detail)
```

以下のコードのように psimDebug() を書いておく。

エラーコード例

```
from psim import *

initialize()

psimDebug()

def proc():
    yield pause("abc")

activate(proc)()
start()
```

実行すると、8 行目でエラーが発生している事が分かる。

```
Traceback (most recent call last):
  ...
  File "psim\core.py", line ..., in registerRequest
RuntimeError: unsupported operand type(s) for +: 'float' and 'str'
  ...
```

```
File "....py", line 8, in proc
  yield pause("abc")
...
```

第3章　エージェントベースシミュレーション記述

3.1　概要

S^4 Simulation System にてエージェントベースモデリングをする際は、大きくわけて、

- 環境
- エージェント集合
- エージェント

に抽象化する。

環境は、エージェント行動の基盤となるジオメトリ情報や、全エージェントのテーブルなどを管理するものである。

エージェント集合は、互いに影響を及ぼしあいながら行動する複数のエージェントの集りを表現するものである。エージェント集合は必ず環境に属す。

エージェントは、個々の自律的に動作する主体を表現するものである。エージェントには、固定ステップで同期的に動作する同期エージェント、個々のエージェントが非同期に動作する非同期エージェントがある。エージェントは必ずエージェント集合に属す。

また、エージェント結果を表示するために、エージェントパネルが用意されている。

3.2　環境

環境は、エージェント行動の基盤となるジオメトリ情報や、全エージェントのテーブルなどを管理するものである。

3.2.1　環境の基底クラス

個々のエージェントは独自の属性を持つ。そのため、位置情報などをエージェントの内部属性として保持する事は禁止はしていない。しかしながら、

位置情報はエージェント間で共有する必要があるので、位置情報に関しては、必ず環境の setPositionm，getPosition を経由してやりとりする事を推奨する。

API EnvironmentBase(name, **keys)

環境を作成する。name には名前を指定する。
環境とは、エージェント行動の基盤となるジオメトリ情報や、全エージェントのテーブルなどを管理するものである。
エージェントは必ずひとつの環境を持つ。各エージェントには、環境の位置属性値が、エージェントの属性値として、付与される。

API environmentBase.initAfter(**keys)

環境生成後に呼ばれる。

API environmentBase.initAttribute(agent)

新規のエージェントが作成されたら、必ずこのメソッドが呼ばれる。
エージェントは必ずひとつの環境に属す。
各エージェントに環境に固有の属性を設定する。

API environmentBase.setPosition(agent, pos)

エージェント agent の環境に固有の位置属性を更新する。
エージェントの環境上の位置属性が変わったら、必ずこのメソッドが呼ばれる。
pos は、環境上の位置属性 (座標やノード番号など) であり、そのフォーマットは、環境ごとに異なる。

API environmentBase.getPosition(agent)

エージェント agent の環境に固有の位置属性を返す。
環境上の位置属性 (座標やノード番号など) であり、そのフォーマットは、環境ごとに異なる。

API environmentBase.getRandomPosition()

環境内のランダムな位置属性を返す。
環境上の位置属性 (座標やノード番号など) であり、そのフォーマットは、環境ごとに異なる。

API environmentBase.findNeighborAgents(srcAgent, d = 1, weight = None, withDistance = False)

環境内のエージェントの中で、srcAgent からの距離が、d 以内のエージェントのリストを返す。
widthDistance が True の場合は、距離とエージェントタプルのリストを返す。

API environmentBase.draw(panel)

この環境を panel 上に描画する。

API environmentBase.drawAgents(panel, agents)

この環境上で、エージェントのリスト agents を panel 上に描画する。

3.2.2 グラフ構造を表す環境の基底クラス

API GraphBase(name, graph, **keys)

グラフ構造を表すための EnvironmentBase を継承した環境クラス。graph は NetworkX のグラフオブジェクトを指定する。無向グラフなら networkx.Graph，有向グラフなら networkx.DiGraph が指定可能である。

API graphBase.graph

グラフ構造を表す。NetworkX の Graph もしくは DiGraph のオブジェクトである。

NetworkX の様々な API が利用可能である。基本的な操作は以下になる。

- graph.nodes(): ノードのリストを返す

- graph.edges(): エッジのリストを返す。(ふたつのノードからならタプル)
- graph.node[v]: ノード v の実体を返す。
- graph.node[v][ATTRNAME]: ノード v の属性 ATTRNAME を返す。(左辺値の場合は設定する)
- graph[u][v]: エッジ（u, v）の実体を返す。
- graph[u][v][ATTRNAME]: エッジ（u, v）の属性 ATTRNAME を返す。(左辺値の場合は設定する)

より詳細な使い方は、NetworkX のマニュアルを参照の事。

API graphBase.layout

このメンバー値にキーがノード番号、値が x と y からなるタプルであるような辞書を指定していた場合、draw メソッドでは、その位置にプロットを行う。

API graphBase.distance(srcAgent = None, tgtAgent = None, weight = None, cutoff = None)

最小経路長を返す。
srcAgent と tgtAgent が指定された場合は、その距離を返す。
srcAgent のみが指定された場合は、グラフ上の全ノードまでの距離を、ノード番号 (位置属性) をキーとした辞書の形式で返す。
srcAgent も tgtAgent も指定されなかった場合は、グラフ上の全ノード間の距離を、第 1 キーが起点ノード番号、第 2 キーが終点ノード番号とした、辞書の形式で返す。
cutoff は、探索を停止する距離である。

API graphBase.setPosition(agent, pos)

エージェント agent の環境に固有の位置属性を更新する。エージェントの環境上の位置属性が変わったら、必ずこのメソッドが呼ばれる。
pos が node を示している場合、エージェントはノード node 上にいる事を示す。
pos が (edge[0], edge[1], d) の形式の場合、エージェントはエッジ (edge[0], edge[1]) 上にいる事を示す。
edge[0] の位置を v0, edge[1] の位置を v1 とすると、(1 - d) * v0 + d * v1 の位置にいる事を示す。

setPosition 内部では、findNeighborAgents が高速に動作出来るように、内部にエージェントの位置情報を保持している。具体的には、以下のような実装になっている。この他の処理を加えても良いが、通常ここにあるよう処理は残して置く必要がある。

コード例

```
def setPosition(self, agent, pos):
    try:
        origPos = self._agentAttrs[agent]
        if isinstance(origPos, (list, tuple)):
            (エッジ上)
            self.graph[origPos[0]][origPos[1]]
                ["agents"].remove(agent)
        else:
            (ノード上)
            self.graph.node[origPos]
                ["agents"].remove(agent)
    except:
        pass
    self._agentAttrs[agent] = pos
    if isinstance(pos, (list, tuple)):
        (エッジ上)
        self.graph[pos[0]][pos[1]]
            ["agents"].append(agent)
    else:
        (ノード上)
        self.graph.node[pos]["agents"].append(agent)
```

API graphBase.getPosition(agent)

エージェント agent の環境に固有の位置属性を返す。
エージェントの環境上の位置属性が変わったら、必ずこのメソッドが呼ばれる。
位置属性が node を示している場合、エージェントはノード node 上にいる事を示す。
位置属性が (edge[0], edge[1], d) の形式の場合、エージェントはエッジ (edge[0], edge[1]) 上にいる事を示す。edge[0] の位置を v0, edge[1] の位置を v1 とすると、(1 - d) * v0 + d * v1 の位置にいる事を示す。

getPosition の実装は以下のようになっている。基本的には setPosition された値をそのまま返却している。

コード例

```
def getPosition(self, agent):
    return self._agentAttrs[agent]
```

3.2.3 カスタムグラフ

API CustomGraphBase(name, directed = False, **keys)

空のグラフを作成する。directed が True なら有向グラフ、そうでないなら無向グラフになる。ユーザーがグラフを自分で初期化する必要がある。

例えば、以下のように initAfter メソッドで初期化する事ができる。

コード例

```
def initAfter(self, **keys):
    self.graph.add_node(0)
    self.graph.add_node(1)
    self.graph.add_node(2)
    self.graph.add_node(3)
    self.graph.add_edge(0, 1)
    self.graph.add_edge(0, 2)
    self.graph.add_edge(0, 3)
    self.graph.add_edge(1, 2)
    self.graph.add_edge(1, 3)
    self.graph.add_edge(2, 3)
```

3.2.4 GNM グラフ

API `GNMRandomGraphBase(name, n = 10, m = 20, directed = False, seed = None, **keys)`

ノード数 n, エッジ数 m のランダムグラフを作成する。ただし、m が完全グラフのエッジ数を越えた場合は、完全グラフを返す。directed が True なら有向グラフ、そうでないなら無向グラフになる。seed は乱数の種を指定する。しかし seed が None ならグローバル系列から乱数の種を初期化する。

3.2.5 GNP グラフ

API `GNPRandomGraphBase(name, n = 10, p = 0.1, directed = False, seed = None, **keys)`

ノード数 n のランダムグラフを作成する。おのおののエッジは確率 p で生成される。directed が True なら有向グラフ、そうでないなら無向グラフになる。seed は乱数の種を指定する。しかし seed が None ならグローバル系列から乱数の種を初期化する。

3.2.6 Barabasi Albert グラフ

API `BarabasiAlbertRandomGraphBase(name, n = 10, m = 2, seed = None, **keys):`

Barabasi-Albert モデルに従ったスケールフリーネットワークを作成する。m 個のノードからなら空のグラフから開始し、ノード数が n になるまで成長させる。ノードを追加する時、既に存在する m 個のノードにエッジを張るが、この時、エッジが張られる確率は、それぞれのノードのその時点での次数に比例する。グラフは無向グラフになる。seed は乱数の種を指定する。しかし seed が None ならグローバル系列から乱数の種を初期化する。

3.2.7 Powerlaw Cluster グラフ

API `PowerlawClusterRandomGraphBase(name, n = 10, m = 2, p = 0.2, seed = None, **keys)`

スケールフリー性（次数分布のべき乗則）を持ったネットワークを作成する。m 個のノードからなら空のグラフから開始し、ノード数が n になるまで成長させるのは Barabasi-Albert モデルと同様だが、ノードの追加時に、確率 p で、triangle を生成する。グラフは無向グラフになる。seed は乱数の種を指定する。しかし seed が None ならグローバル系列から乱数の種を初期化する。

3.2.8 完全グラフ

API `CompleteGraphBase(name, n = 10, directed = False, gnodeOrEdge = None, **keys)`

ノード数 n の完全グラフを作成する。directed が True なら有向グラフ、そうでないなら無向グラフになる。

3.2.9 格子グラフ

API `LatticeGraphBase(name, width = 10, height = 10, type = 8, gnode = None, directed = False, **keys)`

幅 width、高さ height の格子グラフを作成する。type が 4 なら 4 方格子、6 なら 6 方格子、8 なら 8 方格子になる。directed が True なら有向グラフ、そうでないなら無向グラフになる。

API `latticeGraphBase.xy2node`

4 方格子もしくは 8 方格子の場合、キーが座標（x, y）で、値がノード番号であるような辞書を返す。座標値は、左下が（0, 0）である。

API latticeGraphBase.node2xy

4 方格子もしくは 8 方格子の場合、キーがノード番号で、値が座標（x, y）であるような辞書を返す。座標値は、左下が（0, 0）である。

3.2.10 GEXF フォーマットグラフ

API GEXFFormatGraphBase(name, basename = "", **keys)

GEXF フォーマットのファイルからグラフを作成する。data フォルダの下の、basename よりデータを読み込む。

GEXF フォーマットでは、配置情報を記録する仕様が別途定義されている。Gephi で自動配置した結果を GEXF フォーマットでエクスポートすると、viz/position/x と viz/position/y にそれぞれ配置位置情報が入っている。そのため、initAfter メソッドは以下のようにしている。

コード例

```
    def initAfter(**keys):
        (Gephi でエクスポートした gexf ファイルの場合、)
        (ノード属性の viz/position/x と viz/position/y にそれぞれ、)
        (配置位置情報が入っている。)
        self.layout = {}
        try:
            for v in self.graph.nodes_iter():
                attrs = self.graph.node[v]
                x = attrs["viz"]["position"]["x"]
                y = attrs["viz"]["position"]["y"]
                self.layout[v] = (x, y)
        except:
            (おそらく、Gephi 以外で作成されたもので、配置位置情報が無い、)
            (あるいは、別の場所にある。)
            (表示したい場合は、適切なレイアウト情報を設定する必要がある。)
            pass
```

3.2.11 GraphML フォーマットグラフ

API `GraphMLFormatGraphBase(name, basename = "", **keys)`

GraphML フォーマットのファイルからグラフを作成する。data フォルダの下の、basename よりデータを読み込む。

3.2.12 ユークリッド 2D

API `Euclid2DBase(name, x0, x1, y0, y1, **keys)`

2 次元ユークリッド空間を作成する。$x \in [x0, x1], y \in [y0, y1]$ の範囲の空間となる。トーラスとして実装しており、必ず範囲内に丸められる。$x \leftarrow x0+(x-x1)mod(x1-x0), y \leftarrow y0+(y-y1)mod(y1-y0)$

API `euclid2DBase.distance(srcAgent = None, tgtAgent = None)`

srcAgent から tgtAgent までのユークリッド距離を返す。

3.2.13 ソーシャルフォースモデル環境

ソーシャルフォースモデル

ソーシャルフォースモデル（Social Force Model, SFM）とは、群集行動の力学ベースモデルのひとつである。各歩行者は質量を持つ質点として表され、平面内で運動する粒子とみなす。各歩行者は、目的地を持つが、他の歩行者や、障害物から相互に干渉を受けながら、それぞれが運動するようなモデルである。

質量 m_i を持つ歩行者 i は、以下の運動方程式に従う。

$$m_i \frac{d\vec{v}_i}{dt} = m_i \frac{v_i^0 \vec{e}_i(t) - \vec{v}_i(t)}{\tau_i} + R(c, \sum_{j(\neq i)} \vec{f}_{ij} + \sum_{W} \vec{f}_{iW})$$

ここで、$\vec{e}_i$ は目的地に向かうベクトル、v_i^0 は歩行者の最適な速度、$\vec{v}_i(t)$ は現在の速度であり、τ_i は加速時間である。$R(c, p)$ は、平均 p、分散共分散共分散行列が $\begin{pmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{pmatrix}, \sigma = c||p||$ の多変量正規分布に従う乱数である。

$\vec{f}_{ij}$ は歩行者 j から 歩行者 i に与える外力であり、以下のように表される。

$$\begin{aligned}\vec{f}_{ij} &= \left\{A_i \exp\left(\frac{r_i + r_j - d_{ij}}{B_i}\right) + kg(r_i + r_j - d_{ij})\right\}\vec{n}_{ij} \\ &+ \kappa g(r_i + r_j - d_{ij})\left\{(\vec{v}_j - \vec{v}_i) \cdot \vec{t}_{ij}\right\}\vec{t}_{ij}\end{aligned}$$

r_i，r_j は歩行者の半径で、d_{ij} は歩行者 i と j の距離で、k は弾性係数で、κ は散逸係数で $\vec{n}_{ij}$ は歩行者 j から i に向かう単位ベクトルで、$\vec{t}_{ij}$ は $(-n_{ij}^2, n_{ij}^1)$ である。g は以下とする。

$$g(x) = \begin{cases} x & x \geq 0 \\ 0 & \texttt{otherwise} \end{cases}$$

$\vec{f}_{iW}$ は障害物 W から 歩行者 i に与える外力であり、以下のように表される。

$$\begin{aligned}\vec{f}_{iW} &= \left\{A_i \exp\left(\frac{r_i - d_{iW}}{B_i}\right) + kg(r_i - d_{iW})\right\}\vec{n}_{iW} \\ &+ \kappa g(r_i - d_{iW})\left\{\vec{v}_i \cdot \vec{t}_{iW}\right\}\vec{t}_{iW}\end{aligned}$$

ここで、r_{iW} は壁までの距離、$\vec{n}_{iW}$ は壁への法線方向の単位ベクトルであり、$\vec{t}_{iW}$ は $(-n_{iW}^2, n_{iW}^1)$ である。A_i は歩行者 i の相互作用の強さ、B_i は歩行者 i の相互作用の範囲である。

障害物

シミュレーション空間は 2 次元の平面であり、左下の座標 (x_0, y_0) と、右上の座標 (x_1, y_1) で定められる。

その中に任意の障害物を配置させる事ができる。障害物は任意の多角形として指定する事が出来る。

ホールにも対応するが、SFM では、障害物を越える事ができないため、連結していない領域間の移動は、別途処理が必要となる。

PGM(Path Graph Method): 経路グラフによる経路計算

SFM では、各エージェントが (可視な) 単一の目的地を持つような場合に、エージェント間障害物間との相互干渉をモデリングする。つまり SFM だけでは、目的地が可視でない場合、エージェントがスタックしてしまうような現象が容易に発生する。そこで、経路グラフによる経路探索モデル（PGM）で

は、SFM とは別に、複数の経路ポイントを経由した歩行者の行動もサポートするように設計されている。

エージェントの通過する可能性のある地点を経路地点と呼ぶ。経路地点は面積を持つ円である。

互いに経路地点の中心を視認出来る経路地点の組はエッジで結ばれる。そのようにして作成された無向 (もしくは有向) グラフを経路グラフと呼ぶ。

SFM では、障害物を越えて目的地に到達する事ができない。視認できない目的地が設定された場合は、経路グラフを元に途中の経路が選択される。そのため、シミュレーション空間のあらゆる位置から視認確認可能な経路地点が少なくともひとつは存在しなくてはならない。

経路グラフの作成方法には幾つかの方法がある。

- 格子状の自動初期化方法

一番簡単な経路グラフ作成方法である。

格子状に経路地点を自動登録し、可視な隣接するエッジは自動接続される。

コード例

```
env.initLatticelPathGraph(nx, ny, r)
```

メリット:	設定が簡単
	疎な経路グラフになるので効率が良い
デメリット:	格子幅が大きいと、不自然な移動になり易い

- 経路地点手動登録、エッジ自動接続方法

経路地点を手動で登録する。エッジに関しては、可視なエッジが自動的に接続される。

コード例

```
v1 = env.addPathPoint(x1, y1, r1)
v2 = env.addPathPoint(x2, y2, r2)
...
env.connectPathGraphAuto()
```

メリット:	エッジの設定が簡単
	適切な経路点が設定できる
	エッジ間の移動はなめらかになる
デメリット:	エッジ数が多いと、メモリ効率、実行効率が非常に悪くなる

- 経路地点手動登録、エッジ手動接続方法

経路地点及びエッジを全て手動で登録する。

コード例

```
v1 = env.addPathPoint(x1, y1, r1)
v2 = env.addPathPoint(x2, y2, r2)
...
e1 = env.addPathEdge(v1, v2)
e2 = env.addPathEdge(v2, v4)
...
```

メリット:　適切な経路点が設定できる
　　　　　適切なエッジが設定できる
デメリット:　設定が大変

PGM における経路選択

経路選択モデルは、ユーザが任意に修正可能な仕組みにするが、基本モデルとして、目的地までの最短経路距離を効用とするロジットモデルが実装されている。

経路選択関数は、SFM エージェントが、ある経路地点に到達した時に呼ばれ、次に選択すべき経路地点を返す関数である。

基本実装されている経路選択モデルでは、次の経路地点 k を選択する確率は以下となる。

$$P_k = \frac{\exp(-\mu D_{kG})}{\sum_{k' \in R} \exp(-\mu D_{k'G})}$$

D_{uv} は経路地点 u から v の最短距離、μ はスケールパラメータである。μ が大きくなると、ほぼ最適解を選ぶようになり、μ が 0 に近づくと、最適でない解を選ぶ確率が増大する。

他の効用を利用した経路選択モデルに拡張したり、全く別のモデルを実装する事も可能である。

SFM エージェントに、現在地と目的地が指定されて配置されると、目的地に向かうために最初に向かうべき経路地点を選択する。その時、この経路選択モデルが適用される。また、ある経路地点に到達したら、次の経路地点を選択するために、この経路選択モデルが適用される。

経路地点を用いない経路計算手法

PGM とは別の経路地点を用いない経路計算手法も実装されている。経路地点を用いないことには次のような利点がある:

- 経路地点やエッジを設定する手間が不要になる（ただしエージェントの出現地点などを指定する際に経路地点を用いる場合、経路地点の設定は必要となる）。
- PGM の場合経路地点を辿っていくため経路地点の配置によっては不自然な経路になることがあるのに対し、空間上の任意の点を通るため自然な経路となる。
- PGM の場合次の経路地点が不可視となった場合に特別な対処が必要となる（最悪の場合迷子になり移動できなくなる）のに対し、任意の点同士の経路を求められるため、目的地を見失った場合でも単純に経路を再計算するだけで復帰できる。

CMM(Corridor Map Method) による経路計算

SFM 初期化時に移動可能領域のボロノイ図を作り、保持しておく。エージェントごとの経路計算において、出発地・目的地から最も近いボロノイ点までの直線経路およびボロノイ図中の境界線に沿って移動する経路を求めることによって出発地から目的地までの経路を得る。経路計算の際、ボロノイ図境界線のうちエージェント自身の直径よりも狭い幅を持つ通路は除いており、エージェントが SFM モデルに従って実際に動作した際に細い詰まってしまうことを回避する。

TIM(Trajectory Improvment Method) による経路計算

CMM で経路を求めたうえで、最適化計算によって経路長の短いパスを求める（CMM は領域の中心を経路の基本とするため、不自然な大回りをする傾向がある）。具体的には次のようなステップで経路を計算する：

1. あらかじめ領域を離散化した各点と障害物までの距離を求めておく
2. CMM の解から曲がり角などを抽出して適当な折れ線を得て、初期解とする。
3. 経路長と障害物までの距離、慣性の項からなる目的関数を定式化し、これを最適化することで障害物に近すぎない範囲で経路長が短い経路を求める（最適化には `scipy.optimize.leastsq` を用いている）。

CMM 同様任意の 2 点間の経路を求めることができ、経路グラフの枝を必要としない。また、経路計算後に左側通行/右側通行を指向する後処理を加えることができる。この処理は得られた経路の各点を進行方向に対して左（右）に指定の距離だけ摂動させることによって行われる。ただし、摂動によって障害物に近づきすぎる場合は摂動しない。

CMM/TIM を選択した場合の SFM モデル上でのエージェントの移動

CMM/TIM の経路計算の結果、経路情報として次の 2 つが得られる:

points: 適当に離散化して得られる経路上の点列

rs: 各点に対し、その点から最も近い障害物までの距離

エージェントは開始位置から points を順に辿っていくことで目的地に向かっていくが、毎時刻ごとに points の中で現在地から可視なうち最も遠い(インデックスの大きい)点を目標位置とする。この際、可視かどうかは points を中心とし rs を半径とする円の中にエージェントの現在位置が含まれるかどうかで判定する(計算速度のため)。もし目標位置が不可視となった場合、エージェントは経路の再計算を行う。

TIM のパラメータおよび設定の方針

経路探索手法に TIM を選んだ場合、次のパラメータを指定することができる:

epsilon エージェントが障害物から保とうとする距離 (1.0m)

agentrate エージェントごとに経路をばらつかせるかどうかのフラグ(bool 値)。デフォルトでは False。True の場合経路長の短さ重視の解/もとの CMM の解の間でランダムに経路を選択するようになる (False)。

res 領域を離散化する際の粒度(0.1m)

oneside 片側通行するかどうか (False)。(片側通行に関するパラメータは、経路計算手法に CMM を選んだ場合も適用される。)

left 片側通行する際にどちらにずらすかのフラグ。True:左側通行、False:右側通行(True)。

leftdist 片側通行を指向する際に、元の経路からその向きにずらす最大距離(1.0m)。デフォルトでは上記の epsilon と同じ値に設定される。

const_sigma 最適化計算時に経路長を短くする/障害物を回避する、のどちらを重視するかをコントロールするパラメータ。経路計算がうまくいかない(障害物を突き抜ける経路が頻繁に得られる)場合、この値を小さくする(0.04)。

設定の方針は、以下のように考えるとよい:

- epsilon は入力する地図の通路幅によって決める。最も細い通路の幅に対して、その半分以下に設定する必要がある。

- res はエージェントの動作をコントロールしたい粒度に従って決める。例えば少なくとも 10 ｃｍ単位でコントロールしたい場合は 0.1m に設定する。プログラムの使用メモリが足りない場合、res を大きくすることで対処する。
- const_sigma はとりあえずデフォルト値（0.04）を使い、障害物を突き抜けるケースが多い場合（出力に「経路計算に失敗。CMM 経路を…」と表示される）0.03、0.02... と小さくする。障害物を突き抜けない範囲で大きくすると、より短い経路を求めることができるようになる。
- その他 agentrate，oneside などのパラメータは、シミュレーションしたい内容・状況に即して選択する。leftdist は、epsilon と同程度かそれ以下に取るのがよい（経路が壊れる可能性があるので大きすぎる値を入れてはいけない）。

PGM/CMM/TIM の比較

3 つのうちのどの経路計算手法を選ぶべきかは、かけられる手間やシミュレーションしたい状況によって決まる。以下、参考までに 3 つの手法の比較について述べる

- 計算速度については、PGM，CMM，TIM の順に高速である。
- PGM は経路地点・エッジの設定に依存（経路地点を経由して進んでいく）のに対して CMM，TIM は経路地点やエッジによらず、設定の手間が無い（エージェントの出現地点に経路地点を用いる場合は、経路地点のみ設定の必要がある）。
- PGM は途中の経路地点を必ず経由し、CMM は移動可能領域の中間を通ろうとするため、不自然に見える経路を求めることがあるのに対し、TIM は最適化で経路を求めるため、人の目で見て自然な経路を選ぶ傾向がある。
- TIM もしくは CMM の片側通行のオプションを用いると対行するエージェントがうまくすれ違い、混雑が生じにくくなる。
- TIM は複数のパラメータを指定する必要があるのに対し、PGM/CMM はそのような必要が無い。

API `SFMEnvironmentBase(name, x0, x1, y0, y1, A, B, k, kappa, r0, T, v0, v1, r, tau, m, c, visR, visTheta, dispR, dispV, dispF, dispW, dispP, dispC, dispB, dispU, dispS, method, res, stayType, **keys)`

2 次元ユークリッド空間上の、Social Force Model 環境を作成する。$x \in [x0, x1], y \in [y0, y1]$ の範囲の空間となる。

各引数の意味は以下である。

(x0, y0) 画面左下の座標

(x1, y1) 画面右上の座標

A 相互作用の強さ (N)

B 相互作用の範囲 (m)

k 弾性係数 (kg/s2)

kappa 散逸係数 (kg/ms)

r0 最大影響半径 (m)

T 経路再探索間隔 (s)

v0 最適速度 (m/s)

v1 最高速度 (m/s)

r 歩行者の半径 (m)

tau 加速時間 (s)

m 体重 (kg)

c 外力の変動係数

visR 視野制限距離 (m) (0 以下で制限なし)

visTheta 視野仰角 (degree)

dispR 歩行者半径表示

dispV 速度ベクトル表示

dispF 歩行者間外力表示

- dispW 障害物外力表示
- dispP 経路グラフ表示
- dispC Corridor Map 表示
- dispB 背景画像表示
- dispU ユーザー定義領域表示
- dispS エージェント視野表示
- method 経路探索手法の指定（"PGM", "CMM", "TIM"から選ぶ）
- res 領域を離散化する際の粒度（0.1m）
- stayType 滞留時の動作種類

  「滞留時の動作種類」は次から選ぶことができる:

  - out シミュレーションから一時的に除外され、他エージェントと干渉しなくなる。
  - flaot 他エージェントと干渉して動く。
  - fix 体重を一時的に増加させて他エージェントと干渉しても動かなくなる。
  - return 他エージェントと干渉して動くが、常にもとの位置に戻ろうとする。

また、経路探索手法に TIM を選んだ場合は前項で述べた各パラメータを引数として与え、設定することができる。

API sfmEnvironmentBase.getAllLayers()

すべてのレイヤーを含むリストを返す。

API sfmEnvironmentBase.addLayer(name = None)

名前が name であるレイヤーを追加する。name = None の場合、「レイヤー X」（X は添え字）という名前が自動的に振られる。

API sfmEnvironmentBase.removeLayer(layer)

レイヤー layer を取り除く。

API `sfmEnvironmentBase.addUserDefinedRegion(layer, sign, points, attrs = None)`

正負が sign の真偽で、形状が (x, y) 座標のリスト points で表現され、属性が attrs であるユーザー定義領域を作成し、 layer に追加する。

API `sfmEnvironmentBase.removeUserDefinedRegion(uregion)`

ユーザー定義領域 uregions を削除する。

API `sfmEnvironmentBase.removeUserDefinedRegions(uregions)`

ユーザー定義領域のリストを受け取り、各領域を削除する。各ユーザー定義領域は所属レイヤーが異なっていてもよい。

API `sfmEnvironmentBase.groupUserDefinedRegions(uregions, attrs)`

ユーザー定義領域のリスト uregions をグループ化する。attrs はグループの属性となる。 uregions の各要素が同一のレイヤーに属していない場合、エラーとなる。

API `sfmEnvironmentBase.ungroupUserDefinedRegion(ugroup)`

ユーザー定義領域のグループ ugroup を解除する。

API `sfmEnvironmentBase.getUserDefinedRegionTable(layer)`

ユーザー定義領域のレイヤー layer に対応する属性テーブルを返す。

API `sfmEnvironmentBase.getAllPathPoints()`

すべての経路地点を含むリストを返す。

API `sfmEnvironmentBase.addPathPoint(x, y, r, attrs = None)`

x, y に半径 r の経路地点を加え、属性 attrs を持たせる。経路地点番号を返す。経路エリアは障害物や他の経路エリアと重なってはいけない。重なった経路エリアは登録されずに、None を返す。登録された場合は経路地点を返す。

API `sfmEnvironmentBase.removePathPoint(pathPoint)`

経路地点 pathPoint を削除する。

API `sfmEnvironmentBase.removePathPoints(pathPoints)`

経路地点のリスト pathPoints を受け取り、各経路地点を削除する。

API `sfmEnvironmentBase.getPathPointTable()`

経路地点の属性テーブルを返す。

API `sfmEnvironmentBase.getPathPoints(attr)`

属性 attr のついている経路点のリストを返す（これは属性名 attr に対する属性値が "1" であることを意味する）。存在しない場合は空リストを返す。

API `sfmEnvironmentBase.getAllPathEdges()`

すべてのエッジを含むリストを返す。

API `sfmEnvironmentBase.addPathEdge(u, v, attrs = None, d = None, r = None)`

u, v を接続する。お互いに可視でない場合は経路として登録されずに、 None を返す。d には u, v 間の距離を指定する。指定しなかった場合は、ユークリッド距離が設定される。r は可視を判定する半幅を意味する。指定しなかった場合は、歩行者の半径を利用する。あまり狭いエッジをつないでしまうと、エージェントがスタックする可能性があるので注意が必要である。

API sfmEnvironmentBase.removePathEdge(pathEdge)

エッジ pathEdge を削除する。

API sfmEnvironmentBase.removePathEdges(pathEdges)

エッジのリスト pathEdges を受け取り、各エッジを削除する。

API sfmEnvironmentBase.getPathEdgeTable()

エッジの属性テーブルを返す。

API sfmEnvironmentBase.addObstacle(polygon)

障害物を加える。輪郭（(x, y) のリスト）もしくは Polygon オブジェクトを指定する。

Polygon オブジェクトとは、Polygon モジュールの Polygon オブジェクトである。

コード例

```
q = Polygon.Polygon(((0.0, 0.0), (10.0, 0.0),
                     (10.0, 5.0), (0.0, 5.0)))
t = Polygon.Polygon(((1.0, 1.0), (3.0, 1.0),
                     (2.0, 3.0)))
a = q - t
```

q は、(0.0, 0.0), (10.0, 0.0), (10.0, 5.0), (0.0, 5.0) を頂点とする長方形を作成する。

t は、(1.0, 1.0), (3.0, 1.0), (2.0, 3.0) を頂点とする三角形を作成する。

a は q から t を引いた形状を表す Polygon となる。つまり、長方形内に三角形の穴があいたような形状となる。

他の、Polygon オブジェクトに対する操作は、Polygon モジュールのマニュアルを参照の事。

例えば、a.isInside(x, y) は、座標 (x, y) が形状 a に含まれるか否かを検査する。上記の例では、a.isInside(2, 1.5) は偽、a.isInside(5, 3) は真となる。

API `sfmEnvironmentBase.convertPathGraphDirected()`

経路グラフを有向グラフに変換する。既に無向な経路が設定されていた場合は、双方向な経路に変換されるが、変換後に経路を加えた場合は、有向グラフとなる。

このメソッドを呼び出さなかった場合は、経路グラフは無向グラフとなる。

API `sfmEnvironmentBase.connectPathGraphAuto(r = None)`

お互いに可視な経路地点間を自動的に接続する。r は可視を判定する半幅を意味する。指定しなかった場合は、歩行者の半径を利用する。あまり狭いエッジをつないでしまうと、エージェントがスタックする可能性があるので注意が必要である。

API `sfmEnvironmentBase.initLatticePathGraph(nx, ny, r, s = None)`

シミュレーショ空間上に、8 方格子で x 軸方向に nx 個、y 軸方向に ny 個、経路地点を配置する。経路エリアの半径は r となる。障害物と重なった経路エリアは登録されない。隣接する可視なエッジは自動的に接続される。登録された経路地点番号と、その座標で構成されるタプルのリスト [(v, x, y), ...] を返す。s は可視を判定する半幅を意味する。指定しなかった場合は、歩行者の半径を利用する。あまり狭いエッジをつないでしまうと、エージェントがスタックする可能性があるので注意が必要である。

API `sfmEnvironmentBase.nearestPathPoint(self, p, num=1, visible=True, proj=None, return_distance=False)`

座標 p=(x, y) に最も近い可視な経路地点番号を近い順に num 個返す (num>=2 のときは返り値がリストになる)。visible=False のとき可視化どうかによらず近くの経路地点を求めて返す。p を座標値ではなく緯度経度として与える場合は proj に射影方法を指定する。return_distance=True の場合経路地点と距離のタプルを返す。

API `sfmEnvironmentBase.innerPathPoints(self, x0, y0, x1, y1, proj=None)`

長方形領域に含まれる経路地点のリストを返す。x,y を座標値ではなく緯度経度として与える場合は proj に射影方法を指定する。

API `sfmEnvironmentBase.inSight(p, v, r = 1.0e-8)`

座標 p から経路地点 v が視野内にあるかどうかを判定する。視野領域は p から v へ向かう幅 2 * r の長方形として、視野領域が、移動可能空間に完全に包含されるかどうかを判定する。

API `sfmEnvironmentBase.inSightPoint(p, p2, r = 1.0e-8, strict=True)`

座標 p から座標 p2 が視野内にあるかどうかを判定する。視野領域は p から p2 へ向かう幅 2 * r の長方形として、視野領域が、移動可能空間に完全に包含されるかどうかを判定する。strict=False とすると障害物を格子に離散化した配列を用いて計算を行う。離散化に伴う誤差が生じるが、障害物の形状が複雑な場合はこちらのほうが高速に計算できる。

API `sfmEnvironmentBase.inSightSector(p, p2, v, visR = -1, visTheta = 90.0, r = 1.0e-8, strict = True)`

座標 p から p2 が視野 (v 方向を中心軸とする半角 theta の扇形) 内にあるか判定する。strict = True だと厳密に判定し、False だと格子で粗く高速に判定する。

API `sfmEnvironmentBase.inSightByGrid(p, p2)`

座標 p から座標 p2 が視野内にあるかどうかの判定を、SDF を用いて粗く (格子幅ぶんの誤差を伴って) 高速に判定する。

API `sfmEnvironmentBase.sampleInnerPathPoint(v)`

経路地点 v のエリア内にある点をサンプリングし、タプル (x,y) を返す。

API `sfmEnvironmentBase.paths(v)`

最短距離を返す辞書 P を返す。P[u][v] が u, v 間の最短距離。

API `sfmEnvironmentBase.pathgraph`

経路グラフを返す。NetworkX のグラフオブジェクトである。無向グラフの場合 networkx.Graph、有向グラフの場合 networkx.DiGraph のオブジェクトである。このオブジェクトは読み込み専用で、変更してはならない。

ノードの属性値は以下である。

- ‘‘p’’: 座標 (numpy.array)
- ‘‘r’’: 半径

エッジの属性値は以下である。

- ‘‘d’’: 距離

API `sfmEnvironmentBase.draw(panel)`

この環境を panel 上に描画する。dispP の場合、経路グラフが表示される。

API `sfmEnvironmentBase.drawAgents(panel, agents)`

この環境上で、エージェントのリスト agents を panel 上に描画する。dispR なら、エージェントの半径を表示する。dispV なら速度ベクトルを表示する。dispW なら、歩行者間外力を表示する。dispF なら障害物外力を表示する。

API `sfmEnvironmentBase.distance(srcAgent = None, tgtAgent = None, weight = None, cutoff = None)`

SFM では利用不可である。

API sfmEnvironmentBase.findNeighborAgents(srcAgent, d = 1, weight = None, withDistance = False)

SFM では利用不可である。

API sfmEnvironmentBase.setPosition(agent, pos)

SFM では利用不可である。

API sfmEnvironmentBase.getPosition(agent)

SFM では利用不可である。

API sfmEnvironmentBase.getRandomPosition()

SFM では利用不可である。

API SFMEnvironmentBase.setFacility(name, capacity)

名称 name，容量 capacity の施設を設定する。エージェントはこの name を用いてメソッド setStaying(t, facility=name) とすることで施設を t 秒間利用するようになる。

API SFMEnvironmentBase.readOSM(fname, lon0, lat0, lon1, lat1, abs_path=False, r=1.0, proj="EPSG:2451", margin=1.0, highway="car", maxspeed="motorway:60,trunk:50,primary:50,secondary:40,tertiary:30,motorway_li only_maximum_component=True, weakly_connected=False, clear_cache=False, image_provider="osm",)

OpenStreetMap が提供する pbf ファイルを読みこんで経路グラフに設定する。image_provider を指定した場合同時に背景画像も設定する。

pbf ファイルは例えば http://download.geofabrik.de/ から取得する。pbf ファイルのサイズが大きいため計算には数分程度の時間を要するが、計算結果を保存しておくため同じ設定で行った場合は次回以降の処理は高速化される。

得られた経路グラフのエッジは次の属性を持つ：

highway 道路種別

maxspeed 制限速度 (km/h)

lanes 車線数

tt 通過予想時間 (sec)(制限速度で移動した場合にかかる秒数)

車線数に関してはhttps://wiki.openstreetmap.org/wiki/JA:Key:lanesを、一方通行に関してはhttps://wiki.openstreetmap.org/wiki/JA:Key:oneway をそれぞれ参照して自動で反映している。ネットワークデータを細かく制御したい場合は GIS ソフトなどで編集した上で GeoJSON 形式に変換し、後述の readGeoJSON メソッドで読み込むことを推奨する。

各引数の意味は以下の通り：

- fname : str
  対象ファイルパス． プロジェクトの入力フォルダ (simulator.inputDir の値。通常はプロジェクトディレクトリの inputdefault ディレクトリ) からの相対パスで指定
- lon0, lon1, lat0, lat1 : float
  緯度経度の minmax
- abs_path : bool
  True の場合、fname を絶対パスで指定できるようになる。
- r : float
  経路地点の半径のデフォルト値重なる場合は小さく調整される
- proj : pyproj.Proj
  緯度経度→座標の変換． None のとき恒等変換になる。地域ごとの設定種類については後述する。
- margin : float
  上下左右に設けるマージン [m]
- highway: str
  読み込む道路の種別を指定する。"car"で車道を、"walk"で人が歩ける道を、None を指定した場合は全ての道路を取得する。具体的には、"car"においてはmotorway, trunk, primary, secondary, tertiary, motorway_link, trunk_link, primary_link, secondary_link, tertiary_link, unclassified, residential "walk"においてはtrunk, primary, secondary, tertiary, trunk_link, primary_link, secondary_link, tertiary_link, unclassified, residential, living_street, service, pedestrian, track, footway, steps, corridor, path の属性が付いた道路を取得する。上記のようにカンマ区切りで取得する道路を列挙することでも指定できる。

- maxspeed : str
  制限速度 (km/h) のデフォルト値。カンマ区切りで"motorway:60,trunk:50..."のように各道路種別ごとの値を指定する。
- only_maximum_component : bool
  True にすると得られた道路ネットワークのうち最大の連結成分のみを抽出し、ネットワークが連結になるようにする。(元々連結のときは何もしない)
- weakly_connected : bool
  上記連結性の判定の際、True であれば弱連結性、False のとき強連結性を基準にする
- clear_cache : bool
  True のとき前回の計算結果ファイルが存在しても更新する
- image_provider : str or None
  文字列で背景種類を指定． "osm", "std", "pale", "eng", "photo" から選択できる． None の場合背景なし
- zoomlevel : int
  地図ズームレベル． 指定しない場合タイル数が 20 以内に収まる最大値に調整される (推奨)

平面直角座標の座標系は日本国内でも地域によって異なるため、proj の値はシミュレーションしたい地域によって適切に指定する必要がある。具体的には以下のように対応づけられる:

座標系	EPSG 表記	地域
Japan Plane Rectangular CS I	EPSG:2443	長崎県鹿児島県のうち北方北緯 32 度南方北緯 27 度西方東経 128 度 18 分東方東経 130 度を境界線とする区域内（奄美群島は東経 130 度 13 分までを含む。）にあるすべての島、小島、環礁及び岩礁
Japan Plane Rectangular CS II	EPSG:2444	福岡県　佐賀県　熊本県　大分県　宮崎県　鹿児島県（I 系に規定する区域を除く。）

座標系	EPSG 表記	地域
Japan Plane Rectangular CS III	EPSG:2445	山口県　島根県　広島県
Japan Plane Rectangular CS IV	EPSG:2446	香川県　愛媛県　徳島県　高知県
Japan Plane Rectangular CS V	EPSG:2447	兵庫県　鳥取県　岡山県
Japan Plane Rectangular CS VI	EPSG:2448	京都府　大阪府　福井県　滋賀県　三重県　奈良県 和歌山県
Japan Plane Rectangular CS VII	EPSG:2449	石川県　富山県　岐阜県　愛知県
Japan Plane Rectangular CS VIII	EPSG:2450	新潟県　長野県　山梨県　静岡県
Japan Plane Rectangular CS IX	EPSG:2451	東京都（XIV 系、XVIII 系及び XIX 系に規定する区域を除く。）　福島県　栃木県　茨城県　埼玉県千葉県　群馬県　神奈川県
Japan Plane Rectangular CS X	EPSG:2452	青森県　秋田県　山形県　岩手県　宮城県
Japan Plane Rectangular CS XI	EPSG:2453	小樽市　函館市　伊達市　北斗市　北海道後志総合振興局の所管区域　北海道胆振総合振興局の所管区域のうち豊浦町、壮瞥町及び洞爺湖町　北海道渡島総合振興局の所管区域　北海道檜山振興局の所管区域
Japan Plane Rectangular CS XII	EPSG:2454	北海道（XI 系及び XIII 系に規定する区域を除く。）

座標系	EPSG 表記	地域
Japan Plane Rectangular CS XIII	EPSG:2455	北見市　帯広市　釧路市　網走市　根室市　北海道オホーツク総合振興局の所管区域のうち美幌町、津別町、斜里町、清里町、小清水町、訓子府町、置戸町、佐呂間町及び大空町　北海道十勝総合振興局の所管区域　北海道釧路総合振興局の所管区域　北海道根室振興局の所管区域
Japan Plane Rectangular CS XIV	EPSG:2456	東京都のうち北緯 28 度から南であり、かつ東経 140 度 30 分から東であり東経 143 度から西である区域
Japan Plane Rectangular CS XV	EPSG:2457	沖縄県のうち東経 126 度から東であり、かつ東経 130 度から西である区域
Japan Plane Rectangular CS XVI	EPSG:2458	沖縄県のうち東経 126 度から西である区域
Japan Plane Rectangular CS XVII	EPSG:2459	沖縄県のうち東経 130 度から東である区域
Japan Plane Rectangular CS XVIII	EPSG:2460	東京都のうち北緯 28 度から南であり、かつ東経 140 度 30 分から西である区域
Japan Plane Rectangular CS XIX	EPSG:2461	東京都のうち北緯 28 度から南であり、かつ東経 143 度から東である区域

※ 背景画像は OpenStreetMap や国土地理院からのデータを用いるため、これらを含んだ可視化を行う際は出典の表示が必要となる。readOSM/readGeoJSON

を行った場合は環境部品の「環境上のエージェントの可視化」においてデフォルトでこの表示を行うようになっている。ユーザが可視化コードを書き換える場合、この部分を変更しないように注意すること。

API `SFMEnvironmentBase.readGeoJSON(self, fname, abs_path=False, proj = "EPSG:2451", encoding = "utf-8", r = 0.5, margin = 1.0, epsilon = 1e-2, only_maximum_component = True, weakly_connected=False, oneway_key=None, oneway_values=None, image_provider="osm", **keys):`

ローカルに保存してある geojson ファイルを読み込み、その道路ネットワーク情報を抽出して経路グラフに設定する。image_provider を指定した場合同時に背景画像も設定する。

具体的には、geojson ファイルの Point，LineString タイプのオブジェクト (および MultiPoint，MultiLineString) をそれぞれノード・エッジとするグラフを作る (必要に応じて LineString オブジェクトの端点もノードに追加する)。各オブジェクトの属性も経路グラフのノード・エッジの属性として保持する。

各引数の意味は以下の通り:

- fname : str
  対象ファイルパス． プロジェクトの入力フォルダ (simulator.inputDir の値。通常はプロジェクトディレクトリの input/default ディレクトリ) からの相対パスで指定
- abs_path : bool
  True の場合、fname を絶対パスで指定できるようになる。
- proj : str or None
  緯度経度の座標変換方式。pyproj.Proj の初期化に渡る。None の場合恒等変換 (座標変換しない)。具体的な指定方法については readOSM メソッドの項を参照。
- encoding : str
  geojson ファイルのエンコーディング
- r : float
  経路地点の半径のデフォルト値。重なる場合は小さく調整される

- epsilon : float
  座標値の許容誤差。epsilon 以内の距離の点 (point, line の端点) はマージされる
- margin : float
  上下左右に設ける余白 [m]
- only_maximum_component : bool
  True にすると得られた道路ネットワークのうち最大の連結成分のみを抽出し、ネットワークが連結になるようにする。(元々連結のときは何もしない)
- weakly_connected : bool
  上記連結性の判定の際、True であれば弱連結性、False のとき強連結性を基準にする
- oneway_key : str or None
  片道通行のキー
- oneway_values : str or list of str
  LineSring の attr において oneway_key の値がこの値 (ないしこのリストに含まれる) 場合に片道通行として扱う
- image_provider : str or None
  文字列で背景種類を指定． "osm", "std", "pale", "eng", "photo" から選択できる． None の場合背景なし
- zoomlevel : int
  地図ズームレベル． 指定しない場合タイル数が 20 以内に収まる最大値に自動調整される (推奨)

※ 背景画像は OpenStreetMap や国土地理院からのデータを用いるため、これらを含んだ可視化を行う際は出典の表示が必要となる。readOSM/readGeoJSON を行った場合は環境部品の「環境上のエージェントの可視化」においてデフォルトでこの表示を行うようになっている。ユーザが可視化コードを書き換える場合、この部分を変更しないように注意すること。

API SFMEnvironmentBase.readTile(self, lon0, lat0, lon1, lat1, zoomlevel=None, image_provider="osm")

各種プロバイダからシミュレーション表示範囲の地図画像をダウンロードし、背景に設定する。readGeoJSON や readOSM から呼ばれるが、ユーザがこれを直接呼び出して背景画像を設定することも可能。

各引数の意味は以下の通り：

- lon0, lat0, lon1, lat1 : float
  緯度経度の上下限
- zoomlevel : int
  地図ズームレベル． 指定しない場合タイル数が 20 以内に収まる最大値に調整される
- image_provider : str or None
  文字列で背景種類を指定． "osm", "std", "pale", "eng", "photo" から選択できる． None の場合背景なし

※ 背景画像は OpenStreetMap や国土地理院からのデータを用いるため、これらを含んだ可視化を行う際は出典の表示が必要となる。readOSM/readGeoJSON を行った場合は環境部品の「環境上のエージェントの可視化」においてデフォルトでこの表示を行うようになっている。ユーザが可視化コードを書き換える場合、この部分を変更しないように注意すること。

レイヤー、ユーザー定義領域、経路地点、エッジ、属性テーブルはそれぞれクラス Layer, UserDefinedRegion, PathPoint, PathEdge, AttributeTable で表現される。 Layer は list を継承しており、各要素が UserDefinedRegion になっている。 PathPoint は int を継承しており、 sfmEnvironmentBase.pathgraph のノード番号として参照される。 PathEdge は tuple を継承しており、第一、第二要素がそれぞれ端点の PathPoint になっている。属性テーブルは dict を継承しており、属性名と属性値がそれぞれ key と value になっている。また、これらは下記に示す独自のメソッドも有している。下記以外の方法でこれらのオブジェクトを変更してはならない。

API Layer.getName()

レイヤーの名前を返す。

API Layer.setName()

レイヤーの名前を name にする。

API UserDefinedRegion.getPolygon()

領域の形状を表す多角形オブジェクトを返す。

API UserDefinedRegion.setPolygon(polygon)

領域の形状を表す多角形オブジェクトをセットする。

API `UserDefinedRegion.getAttrs()`

領域の属性辞書を返す。返り値は変更禁止。

API `UserDefinedRegion.setAttr(attr, value)`

既存の属性 `attr` に値 `value` をセットする。

API `UserDefinedRegion.includes(x, y)`

領域が座標（x, y）を含んでいれば `True` を、さもなくば `False` を返す。境界上の場合はどちらの値にもなりうる。

API `UserDefinedRegion.isGroup()`

領域がグループなら `True` を、さもなくば `False` を返す。

API `UserDefinedRegionGroup.getUserDefinedRegions()`

グループを構成しているユーザー定義領域のリストを返す。

API `PathPoint.getPos()`

経路地点の座標をタプル形式（x, y）で返す。

API `PathPoint.setPos(x, y)`

経路地点の座標をセットする。

API `PathPoint.getRadius()`

経路地点の半径を返す。

API `PathPoint.setRadius(r)`

経路地点の半径をセットする。

API `PathPoint.getAttrs()`

経路地点の属性辞書を返す。

API `PathPoint.setAttr(attr, value)`

既存の属性 `attr` に値 `value` をセットする。

API `PathEdge.getPathPoints()`

エッジの両端の経路地点をタプル形式で返す。

API `PathEdge.getAttrs()`

エッジの属性辞書を返す。

API `PathEdge.setAttr(attr, value)`

既存の属性 `attr` に値 `value` をセットする。

API `AttributeTable.add(attr, defaultValue)`

属性 `attr` を追加し、デフォルト値を `defaultValue` とする。

API `AttributeTable.set(attr, defaultValue)`

既存の属性 `attr` にデフォルト値 `defaultValue` をセットする。

API `AttributeTable.remove(attr)`

属性 `attr` を削除する。

3.2.14 ネットワークシミュレーション環境

ネットワークシミュレーションモデル

ネットワークシミュレーションモデルは、ソーシャルフォースモデルよりも広範囲の人やモノ (=エージェント) の移動を大まかに再現するためのモデルある。ネットワークはノードとそれらをつなぐエッジからなり、エージェ

ントは出発地ノードから目的地ノードに向けてネットワークのエッジ上を移動する。

S4 のネットワークシミュレーションモデルでは、自然なエージェントの移動・渋滞を表現するために以下の機能が備わっている:

速度計算 エージェントの移動速度を周辺の人口密度から算出する。人口密度から移動速度の計算はデフォルトでは歩行者の移動速度モデルである Greenberg の式によって行われるが、NW エージェント部品の「エージェント速度の計算処理」から変更可能。

人口密度の計算 上記の人口密度は自身がいるサブエッジ (=エッジを数メートル単位で細かく分けたもの。分ける単位はパラメータで指定可能。) に存在するエージェント数とサブエッジの長さ・エッジ幅から算出される。この際、エージェントの移動方向ごとに別々に人口密度を求めるか、双方向を足し合わせて求めるかを指定できる。このパラメータは NWAgentSet が持っており、NWAgentSetBase.setEdgeInfo() から変更できるほか、NW 環境部品からも編集可能。例えば道路の車を考えるときは前者で、一本の歩道上の歩行者を考えるときは後者の設定を使うことになる。

ノード容量・通過時間 ネットワークの作り方によってはノードは交差点を表す場合もあり、そのようなときはノードを通過するのに時間を要するのが自然である。ノードに容量・通過時間を設けられており、エージェントはノードに到達すると次のノードに向かうまでに通過時間経過するまでそこで待機する。また、ノード容量ぶんのエージェントがそのノードに存在する場合、エージェントはノードに入る前にその場で待機することになる。これは交差点が満員で侵入できない状況を表現できる。

通過時間·容量は NWAgentSet がパラメータとして持っており、NWAgentSetBase.setNodeInfo() から変更できるほか、NW 環境部品からも編集可能。なお通過時間を 0 に設定すれば上記の処理は行われない。

過度な混雑による渋滞 前述したサブエッジごとに上限の人口密度が設けられており、上限に達しているサブエッジには後から来たエージェントは侵入できず、サブエッジ境界で停止する。これが連鎖することで各サブエッジ境界にエージェントが溜まっていき、渋滞が実現する。サブエッジの人口密度上限は NWAgentSet がパラメータとして持っており、NWAgentSetBase.setEdgeInfo() から変更できるほか、NW 環境部品からも編集可能。

信号 ノードに信号を配置し、エッジからエッジへの移動を制限できる。上記のノード通過時間を経過したエージェントが次のエッジに移ろうとした

時点で、信号が存在してかつ赤の場合は青になるまで停止する。信号は SFMAgentSet の API(setSignal など) から配置・設定できる。

ゲート 施設の入り口が閉鎖されている状況を表現する。エッジに対して開・閉の時間を指定することができ、閉じているエッジにはエージェントは侵入できないようになり、境界のノードで停止する。内部的には閉じているエッジは長さが十分大きいものとして扱われるので、通常は経路選択で回避されるが、そのエッジを通らないと目的地に到達できないような場合 (施設の中に入りたい、等) はその手前のノードで停止することになる。設定は SFMAgentSetBase.setGate などから行える。

施設利用 目的地に到達したエージェントが、容量の決まった施設を一定時間利用することを表現する。あらかじめノードに施設を配置しておき、利用したいエージェントがノードに到達すると、容量が空いていれば一定時間専有したのち開放し、空いていなければ空くまで待機する。飲食店に並んで利用するような状況を表現できる。SFMEnvironmentBase.setFacility(name, capacity) で配置し、SFMAgentBase.setStaying(facility=name) として利用する。

上記の設定を行う API は環境・エージェント集合に定義されている。なお信号・ゲート・施設利用に関しては SFM 環境でも同様に利用することができる (そのため API は SFMEnvironment や SFMAgentSet に定義されている)。

API NWEnvironmentBase(name, x0 = 0.0, x1 = 1.0, y0 = 0.0, y1 = 1.0, v0 = 0.6,（最適速度（m/s））v1 = 1.4,（最高速度（m/s））r = 0.3,（歩行者の半径（m））visR = 5,（視野制限距離（m）（0 以下で制限なし））visTheta = 45.0,（視野仰角（degree））dispR = True,（歩行者半径表示）dispP = False,（経路グラフ表示）dispB = False,（背景画像表示）dispU = False,（ユーザー定義領域表示）dispS = False,（エージェント視野表示）basename = None,（地図ファイル）maxNum = 10000,（エージェント最大同時出現数）subnodeIntval = 5.0,（サブエッジ長）edgeWidth = 3.0,（エッジ幅）isDirected = False,（True のとき逆向きエッジで人口を足し合わせない）nodeCapacity = 10,（ノード容量）nodeTime = 0,（ノード通過時間）edgeThred = 5.0,（エッジ人口密度上限）**keys)

ネットワークシミュレーション環境を生成する。引数は SFM と同様だが、ネットワークシミュレーション特有のものは以下 maxNum, subnodeIntval, edgeWidth, isDirected, nodeCapacity, nodeTime, edgeThred がある。

API SFMEnvironmentBase.setFacility(name, capacity)

名称 name, 容量 capacity の施設を設定する。エージェントはこの name を用いてメソッド setStaying(t, facility=name) とすることで施設を t 秒間利用するようになる。

3.3 エージェント集合

3.3.1 エージェント集合の基底クラス

API AgentSetBase(name, env, **keys)

エージェント集合を作成する。
このエージェント集合は、generateAgents が作成する複数のエージェントを管理する。
各エージェントは、環境 env に属す。

API `agentSetBase.env`

エージェント集合が属す環境を返す。

API `agentSetBase.agents`

エージェント集合を構成するエージェントのリストを返す。

API `agentSetBase.initAfter(**keys)`

エージェント集合の作成後にこのメソッドが呼ばれる。
エージェント集合固有の初期化を行う。
もし、エージェント集合で固有の属性 (エージェント全体で共有したい属性) がある場合は、このメソッド内で初期化を行う。
通常は、この処理の最後に、`generateAgents` メソッドを使って、エージェントの生成を行う。

API `agentSetBase.generateAgents(n, *args, **keys)`

エージェントを、`n` 個作成する。
`*args, **keys` は、各エージェントのコンストラクタに渡される。

API `agentSetBase.remove(agent)`

エージェント `agent` を削除する。

API `agentSetBase.findNeighborAgents(agent, d = 1, weight = None, withDistance = False)`

エージェント集合に属すエージェントの中で、`agent` からの距離が、`d` 以内のエージェントのリストを返す。
`widthDistance` が `True` の場合は、距離とエージェントタプルのリストを返す。

API agentSetBase.findVisibleAgents(agent, p = None, v = None, visR = None, visTheta = None, r = 1.0e-8, strict = True, withDistance = False)

agent の視野（位置 p から方向 visDir を向いた半角 visTheta の扇形範囲）内に存在するエージェント id のリストを返す。p を与えない場合は agent の現在位置（agent.p）を用いる。withDistance を True にすると距離とエージェント id のタプルのリストを返す。agent, withDistance 以外の引数は agent.inSightSector() メソッドに渡される。

API agentSetBase.start()

エージェント集合の動作を開始する。

API agentSetBase.getAgentScreen(interval = 1, xlim = None, ylim = None, title = "agent frame", size = (700, 700)

エージェントを描画するスクリーンを返す。interval は描画間隔を指定する。xlim, ylim には、それぞれ x 軸と y 軸の表示範囲をタプルで指定する。title にはフレームの名前を指定する。size にはフレームのサイズをタプルで指定する。

API agentSetBase.view()

エージェントの描画を開始する。

通常は以下のようなコードになる。

コード例

```
def view(self):
    interval = 1 (表示間隔)
    screen = self.getAgentScreen(interval = interval,
        xlim = None, ylim = None)
    screen.addAgentSet(self)
    screen.start()
```

もし、表示を行いたくない場合は、以下のように空のメソッドにする。

コード例
```
def view(self):
    pass
```

3.3.2 同期エージェント集合

API SynchronousAgentSetBase(name, env, interval = 1, **keys)

同期エージェント集合を作成する。
このエージェント集合は、generateAgents が作成する複数のエージェントを管理する。
各エージェントは、環境 env に属す。

API synchronousAgentSetBase.step()

エージェント集合のステップ処理を行う。

3.3.3 非同期エージェント集合

API AsynchronousAgentSetBase(name, env, **keys)

非同期エージェント集合を作成する。
このエージェント集合は、generateAgents が作成する複数のエージェントを管理する。
各エージェントは、環境 env に属す。

3.3.4 ソーシャルフォースモデルエージェント集合

API SFMAgentSetBase(name, env, interval = 1, **keys)

Social Force Model エージェント集合を作成する。同期エージェント集合を継承している。
このエージェント集合は、generateAgents が作成する複数のエージェントを管理する。
各エージェントは、環境 env に属す。必ず env は、SFMEnvironmentBase のオブジェクトを指定しなくてはならない。

API `sfmAgentSetBase.step()`

各 SFM エージェントを、SFM 環境のルールに従って、動かす。動かした後、各 SFM エージェントの step 関数を呼び出す。

API `agentSetBase.findNeighborAgents(agent, d = 1, weight = None, withDistance = False)`

エージェント集合に属すエージェントの中で、agent からの距離が、d 以内の可視なエージェントのリストを返す。weight は無視される。widthDistance が True の場合は、距離とエージェントタプルのリストを返す。

API `agentSetBase.closeGate(u, v, d = None)`

u->v のエッジを通行止めにする。d を指定した場合、内部的にエッジをその長さとして扱う。既に閉じている場合は何もしない。

API `agentSetBase.openGate(u, v)`

u->v のエッジの通行止めを解除する。d を指定した場合、内部的にエッジをその長さとして扱う。既に開いている場合は何もしない。

API `sfmAgentSetBase.setGate(u, v, pattern)`

エッジ u->v の open/closeGate() を計画的に実行する。pattern は辞書で {"open": [t,..], "close": [t,...]} のように指定することで、開/閉の時間を一括で指定できる。

API `sfmAgentSetBase.setSignal(u, v, w, green, yellow, red, start = 0)`

エッジ u->v からエッジ v->w に遷移するときの信号を設定する。green, yello, red で指定した秒数だけ信号が切り替わる。start で指定した時刻に青になったとする。既に同じ箇所に信号が置かれている場合、上書きする。

API `sfmAgentSetBase.removeSignal(u, v, w)`

上記で設定した信号を削除する。

API `sfmAgentSetBase.setSignalIntersection(u, v, w, green, yellow, red, start=0, interval=2)`

次数 4 の頂点 v に対して v を交差点とみなし、反対向き、交差する向きにまとめて設定する。start に指定した時刻で (u,v) から (v,w) への信号が青になる。interval で両方向とも赤の時間を設定する。

API `sfmAgentSetBase.setSignalIntersectionWalk(u, v, w, z, green, yellow, red, start=0, interval=2)`

交差点を囲む 4 辺 (u,v,w,z) に対して歩行者用信号を設定する。start に指定した時刻で (u,v) に侵入する箇所の信号が青になる。interval で両方向とも赤の時間を設定する。

API `sfmAgentSetBase.setSignalCrossWalk(u, v, green, yellow, red, start = 0, directed = False)`

辺 (u,v)(u から v への移動) を制限する信号を設定する。start に指定した時刻で (u,v) が渡れるようになる。directed=False だと双方向とも制限される

API `sfmAgentSetBase.getSignalColor(u, v, w, t = None)`

辺 (u,v) から (v,w) に設置された信号のシミュレーション時刻 t における色 ("green", "yellow", "red") を返す。t=None の場合現在時刻が使われる。

API `sfmAgentSetBase.getNextGreen(u, v, w, t)`

現在時刻 t に対し、枝 (u,v) から枝 (v,w) に行くときの次の青信号までの待ち時間を返す。

API `sfmAgentSetBase.setAgentProfile(agentid, assettype = 1, appearance = None, skin = None, clothes = None)`

3D アニメーション表示の際のエージェントのプロファイル（見た目）を設定する。

各引数はつぎの項目を指定する。

agentid プロファイルを指定したいエージェントの固有 ID

assettype エージェントの 3D モデル種別を指定する。現行は 1 のみが指定できる。

appearance 見た目の性別を指定する。"m"（男性）または "f"（女性）を選ぶことができる。

skin エージェントの肌と髪の色を指定する。指定できる選択肢については後述する。

clothes エージェントの服装を指定する。指定できる選択肢については後述する。

エージェントのプロファイルは appearance 引数の値に対して、指定できる skin および clothes の組み合わせが決まっている。組み合わせのリストをつぎに挙げる。

appearance = "m" の場合。

skin

- "BrnSkin_BaldGoatee"
- "DkBrnSkin_Beard"
- "DkSkin_HairGoatee"
- "FairSkin_Bald"
- "FairSkin_BaldGoatee"
- "FairSkin_GryHairBeard"
- "FairSkin_HairDkBrn"
- "LightTan_Bald"
- "LightTan_BaldGoatee"
- "LightTan_Hair"
- "LightTan_HairBeard"

- "Sallow_HairDkBrn"
- "Tan_HairTash_Brown"

clothes

- "Suit_Beige_Full"
- "Suit_Beige_Open"
- "Suit_Beige_ShirtOnly"
- "Suit_BeigeBlue_Open"
- "Suit_Black_Full"
- "Suit_Blue_ShirtOnly"
- "Suit_BlueRed_Casual"
- "Suit_Brown_Full"
- "Suit_LtGreenBlack_WaistcoatTie"
- "Suit_Navy_NoTie"
- "Suit_Navy_Open"
- "Suit_PurpleBlack_WaistcoatTie"
- "Suit_RedBlue_ShirtTie"

appearance = "f" の場合、clothes のタイプとして Skirt と Trouser があり、それぞれについて対応できる skin が決まっている。
Skirt タイプ
skin

- "DarkBlack"
- "DarkBlackShort"
- "DkTanBlackBobbed"
- "SoftTanLtBrown"
- "TanBlack"
- "WhiteBlonde"

clothes

- "SkirtCasual_BeigeCream"
- "SkirtCasual_BluePurple"
- "SkirtCasual_Greens"
- "SkirtSuit_BlackPink"
- "SkirtSuit_LtBlueRed"

Trouser タイプ
skin

- "BlackBobbed"
- "CreamBrown"
- "DarkTanBlackShort"
- "LtTanBlonde"
- "TanBrown"
- "WhiteBlack"

clothes

- "TrouserCasual_Beige"
- "TrouserCasual_BlueBlack"
- "TrouserCasual_Purple"
- "TrouserSuit_BlueBalck"
- "TrouserSuit_BlueWhite"
- "TrouserSuit_GreenBlue"
- "TrouserSuit_PinkBeige"
- "TrouserSuit_RedBeige"

引数の文字列に誤りがあると 3D アニメーションへプロファイル設定が適切に反映されないので注意すること。

3.3.5 ネットワークエージェント集合

ネットワーク上を移動するエージェントをまとめるクラス。SFMAgentSetBase を継承しており、各メソッドを同じように使うことができる。

API `NWAgentSetBase(name, env, interval = 1, **keys)`

ネットワークエージェント集合の初期化を行う。
このエージェント集合は、generateAgents が作成する複数のエージェントを管理する。
各エージェントは、環境 env に属す。env は、NWEnvironmentBase のオブジェクトを指定しなくてはならない。

NWAgentSetBase は SFMAgentSetBase を継承しており、各メソッドを同じように使うことができるが、追加として以下のメソッドを持つ:

API `NWAgentSetBase.setNodeInfo(node, capacity=None, time=None)`

ノード node に容量 capacity と通過時間 time を設定する。

API `NWAgentSetBase.setEdgeInfo(u, v, width=None, directed=None, thred=None)`

エッジ (u,v) に幅・人口密度計算時の方向の有無・人口密度上限を設定する。

3.3.6 粒子フィルタエージェント集合

API ParticleFilterAgentSetBase(name, env, monitor, timecolumn, interval = 1, save_particle_history = True, save_association_history = True, save_stats_history = True, **keys)

粒子フィルタエージェント集合を作成する。同期エージェント集合を継承している。
このエージェント集合は、generateAgents が作成する複数のエージェントを管理する。
各エージェントは、環境 env に属す。
monitor には、フィッティング対象となるデータをモニタまたは時系列モニタで指定する。データは時間列に関して昇順に並んでいる必要がある。timecolumn には、monitor の時間列を列名または列番号で指定する。monitor に時系列モニタを指定した場合には無視される。
save_particle_history が True の場合は、particleHistory のモニタに粒子の状態が記録される。save_associaton_history が True の場合は、associateHistory のモニタに対応付けの結果が記録される。save_stats_history が True の場合は、statsHistory のモニタに粒子の状態の統計量が記録される。各モニタを出力する際には、キーワード引数 simulator を初期化の際に与える必要がある。

API particleFilterAgentSetBase.initAfter(*args, **keys)

エージェントが作成され、environment.initAttribute の呼出し後に、このメソッドが呼ばれる。
もし、エージェントごとに固有の属性がある場合は、このメソッド内で初期化を行う。
初期観測値は keys["observations"] に格納されている。

API `particleFilterAgentSetBase.generateAgents(n, *args, **keys)`

エージェントを n 個作成する。また、各エージェントに対応する粒子群を作成する。
`*args, **keys` は、各エージェントのコンストラクタに渡される。
具体的には、次の処理が行われる。

1. n 個のエージェントを生成して、エージェント集合に追加する。
2. n 個のエージェントそれぞれについて `makeParticles` を呼び出し、粒子群 `particles` を作成する。作成の際には、キーワード引数 `observation` に設定された初期観測値を使用する。
3. 得られた `particles` について次の状態を計算し、`particles.nextStates` を更新する。
4. 得られた `particles` を `filter` に追加する。

API `particleFilterAgentSetBase.remove(agent)`

エージェント `agent` を削除する。同時に、対応する粒子群も `filter` から削除される。既に粒子群が削除されていた場合には、エージェントの削除のみを行う。

API `particleFilterAgentSetBase.filter`

粒子フィルタを表す `ssm.ParticleFilter` のオブジェクトを返す。

API `particleFilterAgentSetBase.monitor`

フィッティング対象のデータを表すモニタまたは時系列モニタを返す。

API `particleFilterAgentSetBase.particleHistory`

粒子の状態を記録したモニタを返す。
初期化の際に save_paritcle_history を False とした場合には None を返す。
モニタの形式は、8.3 節 particleFilter.particleHistory と同様である。

API `particleFilterAgentSetBase.associateHistory`

対応付けの結果を記録したモニタを返す。
初期化の際に save_association_history を False とした場合には None を返す。
モニタの形式は、8.3 節 particleFilter.associateHistory と同様である。

API `particleFilterAgentSetBase.statsHistory`

粒子の状態の各統計量を記録したモニタを返す。
初期化の際に save_stats_history を False とした場合には None を返す。
モニタの形式は、8.3 節 particleFilter.statsHistory と同様である。

3.4 エージェント

3.4.1 エージェントの基底クラス

API `AgentBase(agentset, *args, **keys)`

エージェントを作成する。
このエージェントは、エージェント集合 agentset に属す。
ただし、通常はこのコンストラクタは呼ばれずに、エージェント集合の generateAgents メソッド経由で作成される。

API `agentBase.agentset`

エージェントの属すエージェント集合を返す。

API `agentBase.initAfter(*args, **keys)`

エージェントが作成され、`environmentBase.initAttribute` の呼び出し後に、このメソッドが呼ばれる。
もし、エージェントごとに固有の属性がある場合は、このメソッド内で初期化を行う。

API `agentBase.setPosition(pos)`

エージェントの環境上の位置属性を変更する場合は、このメソッドを呼ぶ。
通常は、`environmentBase.setPosition` が呼ばれる。

API `agentBase.getPosition()`

エージェントの環境上の位置属性を返す。
通常は、`environmentBase.getPosition` が呼ばれる。

API `agentBase.getRandomPosition()`

エージェントの環境上のランダムな位置属性を返す。
通常は、`environmentBase.getRandomPosition` が呼ばれる。

API `agentBase.findNeighborAgents(d = 1, weight = None, withDistance = False)`

エージェント集合に属すエージェントの中で、自身からの距離が、`d` 以内のエージェントのリストを返す。
`widthDistance` が `True` の場合は、距離とエージェントタプルのリストを返す。
通常は、`environmentBase.findNeighborAgents` が呼ばれる。

API `agentBase.agentid`

シミュレーション内でユニークなエージェント番号

API `agentBase.screenColor`

エージェントスクリーンにこのエージェントが表示される時の色

API `agentBase.screenSize`

エージェントスクリーンにこのエージェントが表示される時のサイズ

API `agentBase.screenMarker`

エージェントスクリーンにこのエージェントが表示される時のマーカー

API `agentBase.screenAlpha`

エージェントスクリーンにこのエージェントが表示される時の不透明度

3.4.2 同期エージェント

API `SynchronousAgentBase(agentset, *args, **keys)`

同期エージェントを作成する。
このエージェントは、エージェント集合 `agentset` に属す。ただし、通常はこのコンストラクタは呼ばれずに、エージェント集合の `generateAgents` メソッド経由で作成される。

API `synchronousAgentBase.step()`

エージェントのステップ処理を行う。

3.4.3 非同期エージェント

API `AsynchronousAgentBase(agentset, *args, **keys)`

非同期エージェントを作成する。
このエージェントは、エージェント集合 agentset に属す。
ただし、通常はこのコンストラクタは呼ばれずに、エージェント集合の generateAgents メソッド経由で作成される。

API `asynchronousAgentBase.run()`

エージェントのプロセス処理を行う。

3.4.4 ソーシャルフォースモデルエージェント

API `SFMAgentBase(agentset, *args, **keys)`

Social Force Model エージェントを作成する。同期エージェントを継承している。
このエージェントは、エージェント集合 agentset に属す。
ただし、通常はこのコンストラクタは呼ばれずに、エージェント集合の generateAgents メソッド経由で作成される。

以下のキーワード引数が有効である。

p 現在地

A 相互作用の強さ (N)

B 相互作用の範囲 (m)

v0 最適速度 (m/s)

v1 最高速度 (m/s)

r 歩行者の半径 (m)

tau 加速時間 (s)

m 体重 (kg)

c 外力の変動係数

visR 視野制限距離（m）（0 以下で制限なし）

visTheta 視野仰角（degree）

method 経路計算方法（「TIM」「CMM」「PGM」から選ぶ）

stayType 滞留時の動作（"out", "float", "fix", "return"から選ぶ。詳細は SFM 環境の引数リストを参照）

上記引数はそのまま属性として保持される。これら以外に以下の属性を持つ：

v 速度ベクトル

visDir 視線方向

いずれの引数も、指定されなかった場合は環境オブジェクト（SFMEnvironmentBase）の値が利用される。
経路計算方法に TIM を選んだ場合、その他 TIM 経路計算時に用いられるパラメータ（後述）も指定できる。指定できる経路探索手法およびそれらの概要は次のとおりである（3.2.13 節の SFM 環境の項も参照）。

PGM (Path Graph Method) 従来の経路グラフを用いた経路計算。高速だがエージェントの次の経路地点が不可視になった場合に再探索する必要がある。また、経路グラフの枝を設定する必要がある。

CMM (Corridor Map Method) 移動可能領域のボロノイ図を作り、領域中に得られる中線に沿って移動する経路を求める。エージェント自身の直径よりも狭い通路幅の通路は除いて経路計算を行う。経路地点に頼らず、任意の 2 点間の経路を求めることができる。経路グラフの枝を設定する必要がない。

TIM (Trajectory Improvment Method) CMM で経路を求めたうえで、最適化計算によって経路長の短いパスを求める（CMM は領域の中心を経路の基本とするため、不自然な大回りをする傾向がある）。あらかじめ領域を離散化した各点と障害物までの距離を求めておいて、最適化の各ステップで障害物に近すぎない範囲で経路長が短くなるように更新が行われる。CMM 同様任意の 2 点間の経路を求めることができ、経路グラフの枝を必要としない。また、経路計算後に左側通行/右側通行を指向する後処理を加えることができる。

経路計算法に TIM を用いた場合、SFMAgentBase の初期化時に以下のパラメータを追加で与えることができる（カッコ内はデフォルト値を与える）：

epsilon エージェントが障害物から保とうとする距離（1.0m）

res 領域を離散化する際の粒度（0.1m）

agentrate エージェントごとに経路をばらつかせるかどうかのフラグ（bool 値）。デフォルトでは False。True の場合経路長の短さ重視の解/もとの CMM の解の間でランダムに経路を選択するようになる。

oneside 片側通行するかどうか (False)。（片側通行に関するパラメータは、経路計算手法に CMM を選んだ場合も適用される。）

left 片側通行する際にどちらにずらすかのフラグ。True:左側通行、False:右側通行（True）。

leftdist 片側通行を指向する際に、元の経路からその向きにずらす最大距離（1.0m）。デフォルトでは上記の epsilon と同じ値に設定される。

const_sigma 最適化計算時に経路長を短くする/障害物を回避する、のどちらを重視するかをコントロールするパラメータ。経路計算がうまくいかない（障害物を突き抜ける経路が頻繁に得られる）場合、この値を小さくする。

なお、これらの値はエージェントの他のパラメータと同様、SFMAgentBase の初期化時に与えられていない場合は環境オブジェクトの値が利用される。

API `sfmAgentBase.setDestination(v = None, p = None, method = None, delayed = False)`

目的地を経路地点 v に設定する。method の指定によって経路探索手法を切り替える。method が指定されない場合、エージェントの初期化時に指定された値を用いる（初期化時にも与えていない場合は、環境部品の値を用いる）。座標 p を指定した場合その点を起点として動作する。delayed を True に設定した場合、現在までに delayed=True で指定した setStaying，setDestinaion の動作を全て終えたあとに今回指定する目的地設定が行われる。delayed を False に設定した場合、今回指定する目的地設定が直ちに行われる（過去の指定はクリアされる）。

API `sfmAgentBase.setStaying(t = None, stayType = None, p = None, facility = None, delayed = False)`

エージェントをその場で t 秒間滞留させる。座標 p を指定した場合エージェントをその点に移して停止させる。facility を指定した場合、その名前の資源を t 秒間利用しおえるまでの間その場で待機する。stayType は"out", "float", "fix", "return"から指定できる（詳細は sfm 環境の項を参照）。指定しない場合はエージェントに設定されたデフォルト値が用いられる。delayed を True に設定した場合、現在までに delayed=True で指定した setStaying, setDestinaion の動作を全て終えたあとに今回指定する滞留が実行される。delayed を False に設定した場合、今回指定する滞留が直ちに行われる（過去の指定はクリアされる）。

API `sfmAgentBase.getDestination()`

現在の目標経路地点を返す。設定されていない場合は None を返す。

API sfmAgentBase.setTravel(transitList, order="ordered", method=None, stayTime=100, stayType=None, stayingKwargs=None, delayed=False)

transitList で与えた経路地点を method で指定した方法でエージェントを巡回させる。巡回の際、stayTime で指定した時間だけ各経路地点で滞留する。transitList は経路地点のリストである必要がある。stayTime は int 型かつ非負である。method は"ordered", "random"から指定できる。"ordered"を指定した場合、transitList で指定した経路地点の順番にエージェントを巡回させる。"random"を指定した場合、transitList で指定した経路地点をランダムな順番で巡回させる。method は setDestination のキーワード引数である method に相当する。stayType は setStaying のキーワード引数である stayType に相当する。stayingKwargs はキーが経路地点で値が setStaying に渡すキーワード引数の辞書である。エージェントが到達した経路地点ごとに滞留時間を変えたい場合や環境部品の setFacility で設定した資源を使いたい場合などに用いる。stayTime や stayType など滞留に関するキーワード引数は stayingKwargs で指定したものが優先される。delayed を True に設定した場合、現在までに delayed=True で指定した動作を全て終えたあとに今回指定する巡回が実行される。delayed を False に設定した場合、今回指定する巡回が直ちに行われる（過去の指定はクリアされる）。

API `sfmAgentBase.setRandomTransition(pattern, startPoint=None, goalPoints=None, until=1000, method=None, stayTime=100, stayType=None, stayingKwargs=None, delayed=False)`

pattern で与えた経路地点間の遷移確率に従ってをエージェントをランダムに移動させる。ある経路地点に到達した際には stayTime で指定した時間だけその経路地点で滞留する。エージェントが goalPoints を指定した経路地点に到達する、もしくは until で指定した時間が経過するとランダムな遷移を終える。pattern はキーが経路地点または経路地点のタプルで値が「キーが経路地点または経路地点のタプルで値が float 型の辞書」の辞書である。タプルを与えた場合、遷移が起こるたびにタプルのうちランダムな経路地点を選択してそこへ移動する。startPoint は pattern のキーのうちの 1 つである。startPoint が指定された場合、エージェントは startPoint まで移動してから遷移を開始する。startPoint が指定されない場合、エージェントの現在地が pattern のキーの経路地点のどこかに含まれる場合はその地点から遷移を開始する。startPoint が指定されていないかつ、エージェントの現在地が pattern のキーの経路地点のどれにも含まれない場合、pattern のキーのうちランダムな地点まで移動してから遷移を開始する。goalPoints は pattern のキーのリストである。until は数値型かつ非負である。stayTime は int 型かつ非負である。method は setDestination のキーワード引数である method に相当する。stayType は setStaying のキーワード引数である stayType に相当する。stayingKwargs はキーが経路地点で値が setStaying に渡すキーワード引数の辞書である。エージェントが到達した経路地点ごとに滞留時間を変えたい場合や環境部品の setFacility で設定した資源を使いたい場合などに用いる。stayTime や stayType など滞留に関するキーワード引数は stayingKwargs で指定したものが優先される。delayed を True に設定した場合、現在までに delayed=True で指定した動作を全て終えたあとに今回指定する巡回が実行される。delayed を False に設定した場合、今回指定する巡回が直ちに行われる（過去の指定はクリアされる）。

API `sfmAgentBase.inObstacle()`

障害物上にいるかどうかを調べる。

API `sfmAgentBase.nearestPathPoint(self, p=None, num=1, visible=True, proj=None, return_distance=False)`

座標 p=(x, y) に最も近い可視な経路地点番号を近い順に num 個返す (num>=2 のときは返り値がリストになる)。座標値を指定しない場合、エージェントの現在地が使用される。visible=False のとき可視化どうかによらず近くの経路地点を求めて返す。p を座標値ではなく緯度経度として与える場合は proj に射影方法を指定する。具体的な指定方法については readOSM メソッドの項を参照。return_distance=True の場合経路地点と距離のタプルを返す。

API `sfmAgentBase.inArea(v, p = None)`

自身が経路地点 v の中かどうかを判定する。p を指定した場合は、その座標点で判定する。

API `sfmAgentBase.inSight(v, p = None, r = 1.0e-8)`

自身から経路地点 v が視野内にあるかどうかを判定する。p を指定した場合は、その座標点で判定する。

API `sfmAgentBase.inSight(p0, p = None, r = 1.0e-8, strict = True)`

自身から座標 p0 が視野内にあるかどうかを判定する。p を指定した場合は、その座標点で判定する。strict=False とすると障害物を格子に離散化した配列を用いて計算を行う。離散化に伴う誤差が生じるが、障害物の形状が複雑な場合はこちらのほうが高速に計算できる。

API `sfmAgentBase.inSightSector(p0, p = None, v = None, visR = None, visTheta = None, r = 1.0e-8, strict = True)`

座標 p0 から p が視野 (v 方向を中心軸とする半角 theta の扇形) 内にあるか判定する。strict = True だと厳密に判定し、False だと格子で粗く高速に判定する。

API `sfmEnvironmentBase.findVisibleAgents(p = None, visDir = None, visR = None, visTheta = None, r = 1.0e-8, strict = True, withDistance = False)`

視野（位置 p から方向 visDir を向いた半角 visTheta の扇形範囲）内部のエージェント id のリストを返す。withDistance を True にすると距離とエージェント id のタプルのリストを返す。agent 以外の引数は sfmAgentBase.inSightSector() メソッドに渡される。

API `sfmAgentBase.selectNextNode(G, p = None)`

経路地点 G へ向かうための次の経路地点を返す。次の経路地点は視野内になくてはならない。p が指定されている場合は、その座標点を起点とする。

デフォルトの実装では、p が経路地点 v 内である場合、以下の確率で次の経路地点 k を選択する。

$$P_k = \frac{\mathrm{exp}\left\{-\mu(d_{vk} + D_{kG})\right\}}{\sum_{(v,k')\in E} \mathrm{exp}\left\{-\mu(d_{vk'} + D_{k'G})\right\}}$$

ここで、E は経路グラフのエッジ集合、d_{uv} は経路地点 u から v の距離、D_{uv} は経路地点 u から v の最短距離、μ はスケールパラメータである。μ が大きくなると、ほぼ最適解を選ぶようになり、μ が 0 に近づくと、最適でない解を選ぶ確率が増大する。

p がどの経路地点 v の内部点でない場合、以下の確率で次の経路地点 k を選択する。

$$P_k = \frac{\mathrm{exp}\left\{-\mu(||p_k - p|| + D_{kG})\right\}}{\sum_{k'\in S(p)} \mathrm{exp}\left\{-\mu(||p'_k - p|| + D_{k'G})\right\}}$$

ここで、$S(p)$ は p から直線視野内にある経路地点の集合で、p_k は経路地点 k の座標である。

API `sfmAgentBase.isStopping()`

エージェントが停止状態か否かを返す。

API `sfmAgentBase.isInErrorState()`

エージェントがエラー状態か否かを返す。目的地に到達不能だった場合などに発生する。

API `sfmAgentBase.findNeighborAgents(d = 1, weight = None, withDistance = False)`

自分からの距離が、`d` 以内の可視なエージェントのリストを返す。`weight` は無視される。`widthDistance` が `True` の場合は、距離とエージェントタプルのリストを返す。

API `sfmAgentBase.setPosition(pos)`

`SFM` では利用不可である。

API `sfmAgentBase.getPosition()`

`SFM` では利用不可である。

API `sfmAgentBase.getRandomPosition()`

`SFM` では利用不可である。

API `sfmAgentBase.setAgentProfile(assettype = 1, appearance = None, skin = None, clothes = None)`

3D アニメーション表示の際のエージェントのプロファイル（見た目）を設定する。

各引数の詳細については、`sfmAgentSetBase` の項を参照。

3.4.5 ネットワークエージェント

ネットワーク上を移動するエージェントを表すクラス。`SFMAgentBase` を継承しており、基本的な属性・メソッドを共有している。ただし、次の点が異なる:

- 加速時間 tau は考慮しない (速度が不連続に変化する)。
- ネットワーク上の移動状態を現す情報として以下の属性を持つ:

API `NWAgentBase.e`

移動中エッジの経路地点 (u,v) のペア

API `NWAgentBase.p`

現在座標

API `NWAgentBase.waitTime`

残りノード通過待ち時間

API `NWAgentBase.signalTime`

残り信号待ち時間

API `NWAgentBase.signalTime`

残り信号待ち時間

API `NWAgentBase.inArea(v)`

経路地点 v に存在するかどうかを返す。

- 一部メソッドの仕様が異なる。具体的には、setDestination メソッドの引数 p(移動開始位置) として、座標値ではなく経路地点番号を指定する。詳細は以下の通り:

API `NWAgentBase.setDestination(v = None, p = None, delayed = False)`

目的地を経路地点 v に設定する。p として経路地点番号を与えた場合、。delayed を True に設定した場合、現在までに delayed=True で指定した setStaying, setDestinaion の動作を全て終えたあとに今回指定する目的地設定が行われる。delayed を False に設定した場合、今回指定する目的地設定が直ちに行われる（過去の指定はクリアされる)。

なお、setStaying メソッドの引数 p(滞留位置) については SFM 同様に座標値で指定する:

API NWAgentBase.setStaying(t = None, stayType = None, p = None, facility = None, delayed = False)

エージェントをその場で t 秒間滞留させる。座標 p を指定した場合エージェントをその点に移して停止させる。facility を指定した場合、その名前の資源を t 秒間利用しおえるまでの間その場で待機する。stayType は"out", "float", "fix", "return"から指定できる（詳細は sfm 環境の項を参照）。指定しない場合はエージェントに設定されたデフォルト値が用いられる。delayed を True に設定した場合、現在までに delayed=True で指定した setStaying, setDestinaion の動作を全て終えたあとに今回指定する滞留が実行される。delayed を False に設定した場合、今回指定する滞留が直ちに行われる（過去の指定はクリアされる）。

3.4.6 粒子フィルタエージェント

API ParticleFilterAgentBase(agentset, *args, **keys)

粒子フィルタエージェントを作成する。同期エージェントを継承している。
このエージェントは、エージェント集合 agentset に属す。ただし、通常はこのコンストラクタは呼ばれずに、エージェント集合の generateAgents メソッド経由で作成される。

API `particleFilterAgentBase.makeParticles(observation)`

粒子群を生成する。
本メソッドは初期化処理 `initAfter` の後に呼び出される。
`observation` には、生成に使用する観測値を 1 行からなるモニタで指定する。モニタの各列が観測値の各次元を表す。
返り値として、粒子群をあらわすクラスのオブジェクトを返すとする。このクラスを取得するためには、自身のエージェント集合がもつ `_defParticles` メソッドを使用すればよい。

通常は以下のようなコードになる。

コード例

```
def makeParticles(self, observation):
    AgentParticles = self.agentset._defParticles()
    initial_states = ...    (初期状態の生成)
    return AgentParticles(initial_states, observation)
```

API `particleFilterAgentBase.particles`

エージェントに対応した粒子群を表す `ssm.Particles` のオブジェクトを返す。

3.5 エージェントパネル

API `panel.add_screen(rectOrScreen, zorder = 1, draw_once = False)`

子スクリーンを追加する。`rectOrScreen` には、(`left, bottom, right, top`) の形式で指定する。各数値は `0` から `1` の間に正規化された値で、(`0, 0, 1, 1`) は全範囲を示す。`zorder` は描画順序を示す。数値が高い方が上に描画される。`draw_once` が `True` なら、ステップごとに再描画されない。

API `panel.clear()`

全てのスクリーンをクリアする。

API `panel.draw()`

全てのスクリーンを描画する。

3.6 エージェントスクリーン

API `screen.clear()`

スクリーンをクリアする。

API `screen.set_axis_off()`

軸を表示しない。

API `screen.set_axis_on()`

軸を表示する。

API `screen.set_title(label, fontproperties = None)`

タイトル label を表示する。

API `screen.set_xlabel(label, fontproperties = None)`

x 軸のラベル label を表示する。

API `screen.set_ylabel(label, fontproperties = None)`

y 軸のラベル label を表示する。

API `screen.set_xlim(xlim)`

x 軸の範囲を xlim に設定する。xlim = (xmin, xmax) である。None を指定した場合は自動スケーリングになる。

API `screen.set_ylim(ylim)`

y 軸の範囲を ylim に設定する。ylim = (ymin, ymax) である。None を指定した場合は自動スケーリングになる。

API screen.colormap(name)

名前 name のカラーマップを返す。

API screen.colorbar(left, bottom, width, height, cmap, vmin = 0, vmax = 1, orientation = "vertical")

左下（left, bottom）から、幅 width, 高さ height の領域に、カラーマップ cmap のカラーバーを表示する。座標系は、左下が（0, 0)、右上が（1, 1）である。最小値が vmin で、最大値が vmax になる。向きは orientation で "vertical"/"horizontal" を指定する。カラーバーの最小値が vmin、最大値が vmax になる。

API screen.imshow(data, cmap = None)

配列 data をスクリーン上にプロットする。cmap が指定された場合、data[x][y] は float(0 から 1）であり、cmap(data[x][y]）の色で描画される。cmap が None なら、data[x][y] は、(R, G, B）である。

API screen.line(x0, y0, x1, y1, linewidth = 1, color = 'b', linestyle = "solid", cmap = None, alpha = 1.0, zorder = 1)

(x0, y0）から（x1, y1）に線を描画する。linewidth は線の太さ、color は色、linestyle は線種、cmap はカラーマップ, alpha は不透明度、zorder は描画順序を指定する。

API screen.lines(points, linewidth = 1, color = 'b', linestyle = "solid", cmap = None, alpha = 1.0, zorder = 1)

points = [(x0, y0), (x1, y1), ..., (xn, yn)] に線を描画する。linewidth は線の太さ、color は色、linestyle は線種、cmap はカラーマップ, alpha は不透明度、zorder は描画順序を指定する。

API `screen.point(x, y, size = 5, color = 'b', edgecolor = 'k', linewidth = 1, marker = 'o', cmap = None, alpha = 1.0, zorder = 1)`

(x, y) に marker を描画する。size はサイズ、color は色、edgecolor はエッジの色、linewidth はエッジの太さ、cmap はカラーマップ，alpha は不透明度、zorder は描画順序を指定する。

API `screen.text(x, y, s, fontsize = 10, color = 'k', fontproperties = None, zorder = 1)`

(x, y) に文字列 s を描画する。fontsize はフォントのサイズ、color は色、zorder は描画順序を指定する。

API `screen.arrow(x, y, dx, dy, width = 1.0, **args)`

(x, y) を起点とする矢印を描画する。(dx, dy) は、向きと長さを示す。width は矢印の太さを示す。edgecolor はエッジの色、color は中身の色、linewidth は線の太さ、cmap はカラーマップ、alpha は不透明度、zorder は描画順序を指定する。

API `screen.ellipse(x, y, width, height, angle = 0.0, **args)`

(x, y) を中心とする楕円を描画する。width は水平方向の直径、height は垂直方向の直径、angle は回転角 (反時計回り) を度で示す。edgecolor はエッジの色、color は中身の色、linewidth は線の太さ、cmap はカラーマップ、alpha は不透明度、zorder は描画順序を指定する。

API `screen.polygon(points, closed = True, **args)`

points で構成される多角形を描画する。closed が真なら、points の最後と最初の点をつなぐ。edgecolor はエッジの色、color は中身の色、linewidth は線の太さ、cmap はカラーマップ、alpha は不透明度、zorder は描画順序を指定する。

API screen.rectangle(x, y, width, height, **args)

左下が（x, y）の長方形を描画する。width は幅、height は高さを示す。edgecolor はエッジの色、color は中身の色、linewidth は線の太さ、cmap はカラーマップ、alpha は不透明度、zorder は描画順序を指定する。

API screen.draw()

スクリーンを描画する。

3.6.1 共通

color には以下が指定できる。

- "b": blue
- "g": green
- "r": red
- "c": cyan
- "m": magenta
- "y": yellow
- "k": black
- "w": white
- "#RRGGBB"
- (R, G, B): (float, float, float)

linestyle には、以下が指定できる。

- "solid"/"dashed"/"dashdot"/"dotted"
- "-"/"--"/"-."/":"

marker には以下が指定できる。

- "8": octagon
- "d": thin_diamond

- "<": triangle_left
- "h": hexagon1
- "+": plus
- "o": circle
- "p": pentagon
- "s": square
- "v": triangle_down
- "x": x
- "^": triangle_up
- ">": triangle_right

第4章　エージェントプラナー

4.1　概要

S4 Simulation System にてエージェントベースモデリングをする際に、同期エージェントを記述時に、状態遷移が複雑であったり、状態の遷移に複数ステップかかる場合、ロジックの記述が非常に難しくなる。そのような際に、エージェントプラナーが利用可能である。

4.2　基本概念

planner は、同期型のエージェントの行動計画スケジューラである。

基本的には、プラナー、プロセス、タスク、状態 の 4 階層で表わされる。

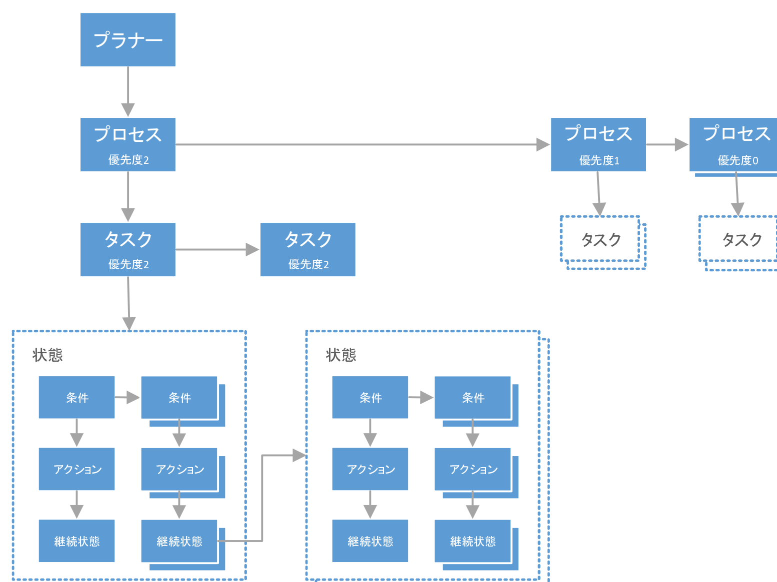

図 4.1: HTN 構成

- プラナー
  - 概要：　エージェントの行動計画スケジューラ。
  - 内容：　優先順位のつけられた複数のプロセスを持つ。

- 動作： 実行すると、保持する全てのプロセスを優先順位順に実行する。
- 操作： プロセスを動的に追加、削除できる。

• プロセス

- 概要： エージェントの目標を達成するために必要なタスクをスケジューリングする。それぞれのプロセスは独立に並行実行可能である。
- 内容： 優先順位のつけられた複数のタスクを持つ。
- 動作： 実行すると、保持するタスクを優先順位順に実行する。あるタスクにて、現ステップでの実行を終了する特殊な状態（TICK(skip=True)）が実行されたら、実行を停止する。
- 操作： タスクを動的に追加、削除できる。

• タスク

- 概要： プロセスを構成する、個々の行動単位を表す。
- 内容： 現在実行中の継続状態を持つ。
- 動作： 実行すると、保持している継続状態を実行する。その結果によって、自身の継続状態が差し替えられる。時間経過を示す特殊な状態（TICK）が実行されるまで、実行を繰り返す。
- 操作： 継続状態は、実行結果によって自動的に差し替えられる。

• 状態

- 概要： タスクの状態を表す。
- 内容： 優先順位のつけられた複数の「条件とアクションと継続状態」の組を持つ。
- 動作： 条件が成立すればアクションを実行し、タスクに継続状態を返す。
- 操作： 状態の定義は、Python の関数の形式で行う。

4.3 同期型/非同期型プロセスの動作比較

非同期型のプロセススケジューラは、以下のように動作する。

各プロセスは、非同期に発生する可能性のあるイベントを待ち受ける。イベントが発生するまでは何もしない。

例：（非同期版）event を待ち受け、evnet が発生したら ‘‘event fired’’ を表示する。

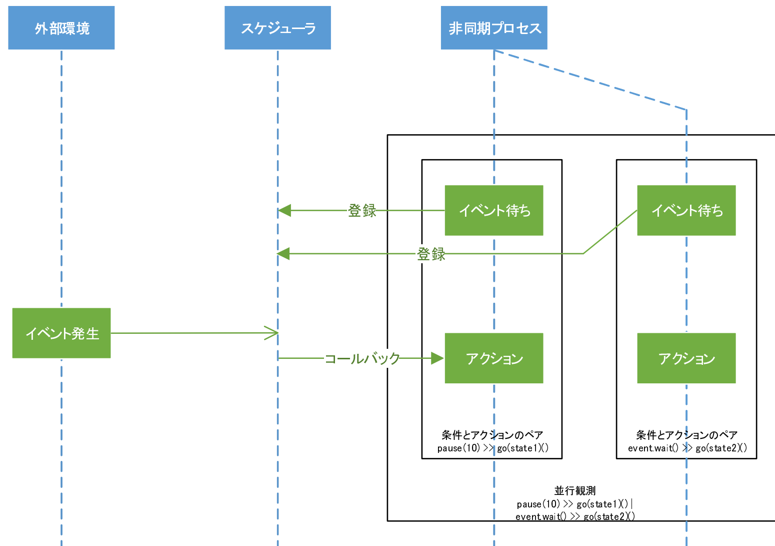

図 4.2: 非同期型プロセスの実行方式

コード例

```
def proc():
    while True:
        yield event.wait()
        print("event fired")
activate(proc)()
start()
```

一方で、同期型のプロセススケジューラは、以下のように動作する。

各プロセスは、各ステップごとにイベントが発生しているかどうかをポーリングする。

例：（同期版）event を待ち受け、evnet が発生したら ‘‘event fired’’ を表示する。

コード例

```
def proc():
    if len(events) > 0:
        event.pop(0)
        print("event fired")
process = planner.add()
process.add(proc)()
while True:
    planner.step()
    ステップが進む
```

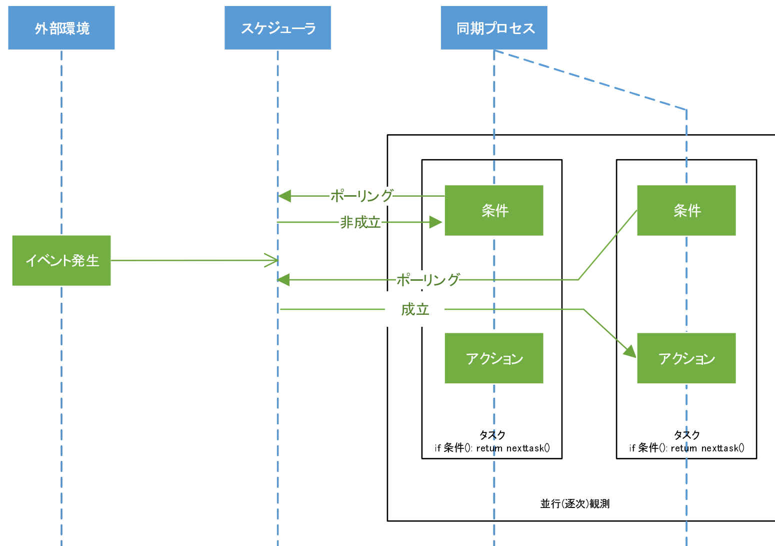

図 4.3: 同期型プロセスの実行方式

4.4 API

4.4.1 プラナー

プラナーはエージェントの行動計画スケジューラである。優先順位のつけられた複数のプロセスを持つ。実行すると、保持する全てのプロセスを優先順位順に実行する。プロセスを動的に追加、削除できる。

API `Planner(name = None)`

プラナーを作成する。名前は `name` となる。

API `planner.add(name = None, priority = 0)`

プラナー上にプロセスを作成し、プロセスを返す。プロセスの名前は `name`、優先度は `priority` となる。

API `planner.remove(name)`

プラナーから、名前が `name` のプロセスを削除する。存在しない場合は何もしない。

API `planner.iter()`

プラナーが保持しているプロセスのイテレータを返す。順番は保証されない。

API `planner.step()`

プラナー上の全てのプロセスをステップ実行する。実行順は、優先度に従う。

4.4.2 プロセス

プロセスは、エージェントの目標を達成するために必要なタスクをスケジューリングする。それぞれのプロセスは独立に並行実行可能である。優先順位のつけられた複数のタスクを持つ。実行すると、保持するタスクを優先順位順に実行する。あるタスクにて、現ステップでの実行を終了する特殊な状態 (`TICK(skip=True)`) が実行されたら、実行を停止する。タスクを動的に追加、削除できる。

API `process.add(func, name = None, priority = 0)(*args, **keys)`

プロセス上に、タスクを追加し、タスクを返す。`func` には、初期状態定義関数名を指定する。タスクの名前は、`name`、優先度は `priority` となる。`args, keys` は、`func` の引数に渡される。

API `process.remove(name)`

プロセスから、名前が `name` のタスクを削除する。存在しない場合は何もしない。

API `process.iter()`

プロセスが保持しているタスクのイテレータを返す。順番は保証されない。

API `process.name`

プロセスの名前を返す。

API `process.priority`

プロセスの優先度を返す。

API `process.step()`

プロセス上の全てのタスクをステップ実行する。実行順は、優先度に従う。ただし、特殊な継続状態 `tick` が呼ばれたら、残りのタスクは実行されない事もある。

4.4.3 タスク

タスクは、プロセスを構成する、個々の行動単位を表す。現在実行中の継続状態を持つ。実行すると、保持している継続状態を実行する。その結果によって、自身の継続状態が差し替えられる。時間経過を示す特殊な状態（`TICK`）が実行されるまで、実行を繰り返す。継続状態は、実行結果によって自動的に差し替えられる。

API `task.name`

タスクの名前を返す。

API `task.priority`

タスクの優先度を返す。

API `task.step()`

タスクの保持する状態を実行する。

4.4.4 状態

状態は、タスクの状態を表す。優先順位のつけられた複数の「条件とアクションと継続状態」の組を持つ。条件が成立すればアクションを実行し、タスクに継続状態を返す。状態の定義は、`Python` の関数の形式で行う。

状態は `Python` の関数として、以下のように定義する。

コード例

```
def 状態(...):
    if 条件1():
        アクション1()
        return 継続状態1()
    elif 条件2():
        アクション2()
        return 継続状態2()
    ...
    else:
        アクションn()
        return 継続状態n()
```

状態は、「条件とアクションと継続状態」の組を、Python の構文に従って定義される。

継続状態は、次に実行すべき状態を示し、以下のような形式で指定する事ができる。

- STATE(状態定義関数名)(*args, **keys)

  単一の継続状態を作成する。状態を実行するタイミングで実体化される。

- (継続状態$_1$, 継続状態$_2$, ..., 継続状態$_n$)

  逐次実行する継続状態の列。

  最初に、継続状態$_1$ が実体化される。その時、継続状態$_n$ (n > 1) は実体化されていない。

  継続状態$_i$ が完了すると、継続状態$_{i+1}$ が実体化される。

- None

  継続状態が無い事を示す。状態定義関数で None を返した場合、そのタスクは完了した事を示す。

- REPEAT()

  特殊な継続状態で、自身を示す継続状態を示す。

- TICK(step = 1, skip = False)

  特殊な継続状態で、step 時間後に完了する。現ステップでのこのタスクの実行を停止する。skip が True の場合は、このタスクを実行したプロセスの残りのタスクの実行も停止する。skip が False の場合は、遺りのタスクの実行を行う。step 時間後まで、このタスクはブロックする。

4.5 動作例

4.5.1 問題設定

- 空間

  ランダムに作成されたグラフ空間。

  - 5 つのエリアがある。
  - それぞれのエリアは、50 ノード、100 エッジの連結ランダムグラフ。
  - 各エリアにはひとつの入口ノードがあり、各入口ノードは、必ず接続されている。(つまり、全体でも連結グラフになる)
  - エリア内には施設がランダムに配置される。

- 願望、エリア、施設

  - エージェントはランダムな複数の願望を持つ。
  - エージェントが施設に訪れる事で、ある願望を満す。
  - 空間には、幾つか地図があり、地図では、以下の情報が得られる。
    * エリアの位置とエリア内の施設情報 (及び、どのような願望を満す施設かであるかの情報
    * エリア内の各施設の位置情報

- エージェントの行動

  - 基本動作
    * エージェントは次に満すべき願望を選択する。
    * 願望を満すためのエリアを知っているなら、そのエリアに向かって移動する。(知らないなら、ランダムウォークする)
    * エリアに到着したら、願望を満す施設を探す。
    * 願望を満すための施設を知っているなら、その施設に向かって移動する。(知らないなら、エリア内をランダムウォークする)
    * 施設に到着したら、次の願望を選択する。
  - 観測動作

    エージェントは移動するごとに、空間に設定された施設情報を学習する。
  - スケジュールされた動作
    * 各エージェントにはそれぞれ帰宅時間が設定されており、その時間になったら、全ての行動をキャンセルし、指定された場所に移動する。

4.5.2 状態遷移図

プロセスがふたつある。

- メインプロセス
- 観測プロセス

メインプロセスには、ふたつのタスクがある。

- 基本タスク
- 予定実行タスク

基本タスクは、エージェントが願望を満すように行動するルールが定められている。

予定実行タスクは、指定時間に帰宅動作に切り換えるようなルールが定められている。

予定実行タスクの優先度が高い。予定実行タスクが開始したら、基本タスクは実行されない。

観測タスクは、メインプロセスとは独立な観測プロセスにて実行される。

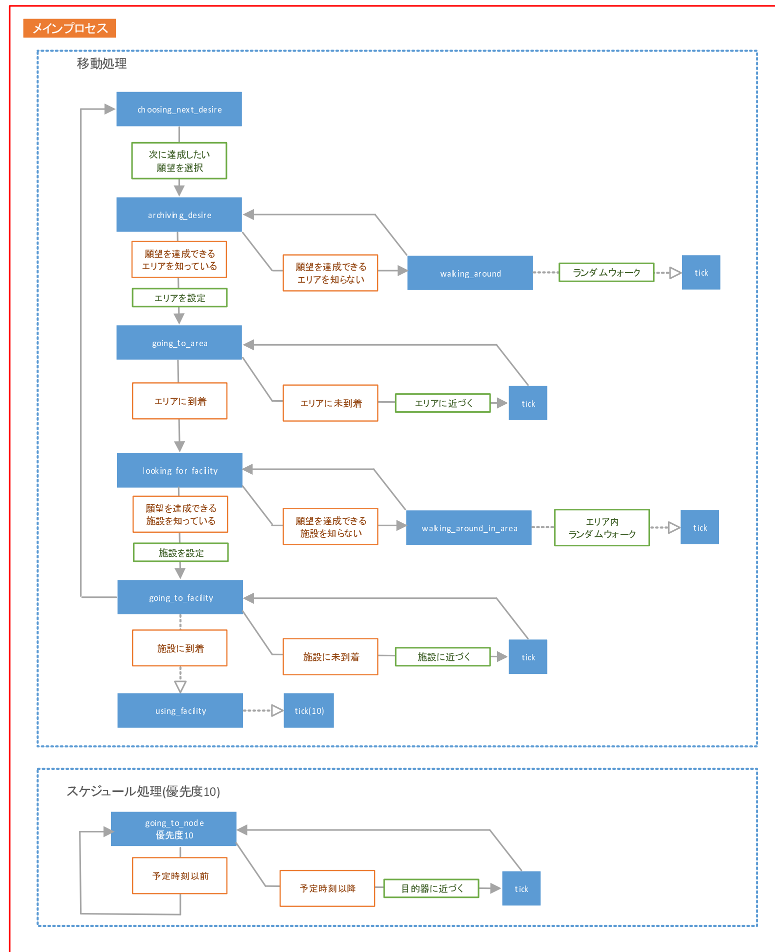

図 4.4: メインプロセス

4.5.3 コード例

- 初期化

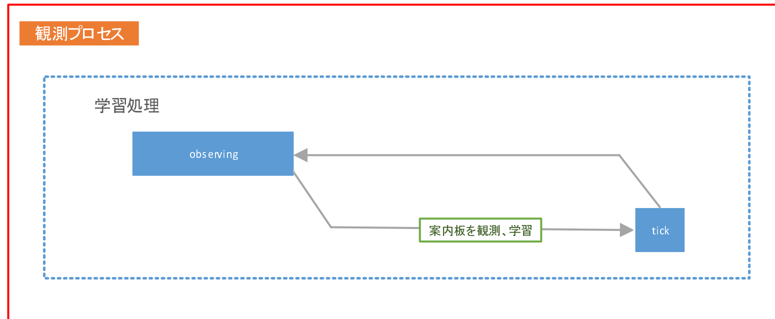

図 4.5: 観測プロセス

コード例

```
def __init__(self):
    self.planner = Planner()

    self.proc1 = self.planner.add("main")
    self.proc1.add(self.choosing_next_desire)()
    self.proc1.add(self.going_to_node, priority = 10)(
        50, goal_node)

    self.proc2 = self.planner.add("observing")
    self.proc2.add(self.observing)()
```

4.5.4 ステップ実行

コード例

```
def step(self):
    self.planner.step()
```

4.5.5 サブ状態

コード例

```
def walking_around(self):
    self.position = random.choice(graph.neighbors(self.position))
    return TICK()
def walking_around_in_area(self):
    self.position = random.choice(graph.neighbors(self.position))
    while True:
        self.position = random.choice(graph.neighbors(
            self.position))
        if graph.node[self.position]["area"] == self.goal_area:
            break
    return TICK()
def walking_to_position(self, dest):
    path = networkx.shortest_path(graph, self.position, dest)
    self.position = path[1]
    return TICK()
```

- 基本タスク

コード例

```
def choosing_next_desire(self):
    self.goal_desire = random.choice(self.desire_list)
    self.goal_area = None
    self.goal_facility = None
    return STATE(self.archiving_desire)()
def archiving_desire(self):
    areas = self.known_areas.get(self.goal_desire, [])
    if len(areas) > 0:
        self.goal_area = random.choice(list(areas))
        return STATE(self.going_to_area)()
    else:
        return STATE(self.walking_around)(), REPEAT()
def going_to_area(self):
    if graph.node[self.position]["area"] == self.goal_area:
        return STATE(self.looking_for_facility)()
    else:
        return STATE(self.walking_to_position)(
            area_entrance[self.goal_area]), REPEAT()
def looking_for_facility(self):
    try:
        vs = self.known_facilities[self.goal_area][
            self.goal_desire]
    except:
        vs = []
    if len(vs) > 0:
        self.goal_facility = random.choice(list(vs))
        return STATE(self.going_to_facility)()
    else:
        return STATE(self.walking_around_in_area)(), REPEAT()
def going_to_facility(self):
    if self.position == self.goal_facility:
        return STATE(self.using_facility)(), STATE(
            self.choosing_next_desire)()
    else:
        return STATE(self.walking_to_position)(
            self.goal_facility), REPEAT()
def using_facility(self):
    return TICK(10)
```

- 予定実行タスク

コード例

```
def going_to_node(self, time, dest):
    if now[0] < time:
        return TICK(), REPEAT(priority = 10)
    else:
        if self.position == dest:
            self.planner.remove("main")
            self.planner.remove("observing")
            return None
        else:
            path = networkx.shortest_path(
                graph, self.position, dest)
            self.position = path[1]
            return TICK(skip = True), REPEAT()
```

- 観測タスク

コード例

```
def observing(self):
    print("goal (desire: %s, area: %s, facility: %s), %s" % (
        self.goal_desire, self.goal_area, self.goal_facility,
            self.position))
    if "facility-ad" in graph.node[self.position]:
        area, facilities = graph.node[self.position][
            "facility-ad"]
        for v, desire in facilities:
            self.known_facilities.setdefault(area, {})
            self.known_facilities[area].setdefault(
                 desire, set())
            self.known_facilities[area][desire].add(v)
    if "area-ad" in graph.node[self.position]:
        for desire, areas in desire2areas.iteritems():
            self.known_areas.setdefault(desire, set())
            for area in areas:
                self.known_areas[desire].add(area)
    return TICK(), REPEAT()
```

4.6 代表的なパターン

4.6.1 基本状態遷移

コード例

```
def state1():
    action()
    return STATE(state2)()
```

state1 に入ると、action を実行し、state2 に遷移する。

もし、この遷移に時間を要する場合は、以下のように TICK() を入れる必要がある。

コード例

```
def state1():
    action()
    return TICK(), STATE(state2)()
```

TICK() があると、このタスクのこのステップでの実行は完了する。(他のタスクがある場合は、それぞれ実行される)

4.6.2 自分への状態遷移

コード例

```
def state1():
    if condition1():
        action()
        return STATE(state2)()
    else:
        return TICK(), REPEAT()
```

ある条件が成立するまで待ち受けるようなパターンは、このように書く。自分に戻る時は、通常、TICK() が入る。これがないと、無限ループになる。

4.6.3 タスクの終了

コード例

```
def state1():
    action()
    return None
```

次に遷移する状態がない、つまり、タスクが完了する場合、None を返す。python の仕様上、何も返さなければ return None と同一の意味を持つ。

4.6.4 サブ状態

コード例

```
def substate():
    subaction()
    return None
```

サブ状態は、別の状態から再帰的に呼び出される状態の事である。コンテキストが違うだけで、サブ状態も状態である。

上の substate は、subaction を実行した後に完了するサブ状態である。

サブ状態もまた、複数の状態を持っても構わない。TICK() のように時間経過をするような処理が入る場合もある。

サブ状態は、複数の状態から呼び出される可能性がある場合に、非常に便利である。通常、メインの状態遷移は複数の状態の有向グラフの形で表現される。サブ状態もまた、入口と出口の定義された有向グラフであるが、入口と出口は他の状態と動的に結合される。

4.6.5 サブ状態が完了したら、別の状態に遷移

コード例

```
def substate():
    subaction()
    return None
def state1():
    action()
    return STATE(substate)(), STATE(state2)()
```

サブ状態は、別の状態から再帰的に呼び出される状態の事である。

上の例では、state1 に入ると、substate 状態の実行後に、state2 に遷移する。

サブ状態もまた、複数の状態を持っても構わない。TICK() のように時間経過をするような処理が入る場合もある。

4.6.6 より複雑な手順 (サブ状態のリスト) に遷移

コード例

```
def state():
    action()
    return STATE(substate1)(), STATE(substate2)(), STATE(state2)()
```

一般的には、サブ状態のリストを指定する事で、より複雑な手順を指定する事も出来る。

4.6.7 同一プロセス内の優先度によるタスク実行

サブ状態のリストを指定する事で、複雑な手順も指定できるが、より複雑な場合は、別タスクを起動し、イベントドリブン (毎ステップ監視) 式に記述したり、動的にタスクを起動する必要がある。

コード例

```
def scheduled_state():
    if condition():
        return TICK(), REPEAT()
    else:
        return STATE(state2)()
```

この例では、ある条件 `condition()` が成立するまで待ち受け、成立したら `state2` に遷移するタスクを登録している。

`condition` の中では、時間や環境などを観測したり、自分、あるいは、他のエージェントからのインタラクションなどを観測する。

同一プロセス内に複数のタスクを起動する場合、優先度が重要になる。優先度の低いタスクの実行をキャンセルしたい場合は、`TICK(skip = True)` とする。`skip` が真なら、このタスクを実行したプロセスの残りのタスクの実行も停止する。`skip` はデフォルトでは 偽である。

4.6.8 別プロセスによるタスク実行

並行に動作するタスクが複数ある場合、

- ひとつのプロセスに複数のタスクを走らせる
- 複数のプロセスで各タスクを走らせる

の選択肢がある。

タスクに優先度があり、各ステップにて、優先度の高いタスクのみを実行したい場合、前者のような設計にするのが良い。

完全に独立に動作するタスクの場合は、後者のような設計にするのが良い。

コード例

```
def observing():
    action()
    return TICK(), REPEAT()
```

第5章　フローベース・フレームワーク

2 章のシミュレーション記述を用いれば様々なシミュレーションを記述する事が出来るが、より大規模で複雑なシミュレーションをフローベースで記述するためのフレームワークが用意されている。このフレームワークは、部品、ポート、リンク、フローアイテムなどを組み合わせる事で簡単にシミュレーションを記述出来るような仕組みを提供する。S^4 `Simulation System` で自動生成されるコードはこのフレームワークを利用している。

5.1　シミュレーター

5.1.1　シミュレーター

API　`SimulatorBase(param, inputDir, outputDir, until)`

シミュレーションパラメータ `param` のシミュレーターを作成する。入力フォルダを `inputDir`、出力フォルダを `outputDir` に指定する。`until` には `until(self)` がシミュレーション終了時間になるような `lambda` 式を指定する。`SimulatorBase` は基底クラスであり必ず継承して使う。

API　`simulatorBase.run()`

シミュレーションを開始する。

API　`simulatorBase.save()`

登録されている全てのモニターを出力する。

5.1.2 一括実行シミュレーター

API BatchSimulator(simcls, paramcls, inputDir, outputDir, iters, maxcount, outputs, until)

様々なパラメータでシミュレーションを行う一括実行シミュレーターを作成する。simcls にはシミュレータークラス名を指定し、paramcls にはパラメータクラス名を指定する。inputDir, outputDir にはそれぞれ、入力フォルダ、出力フォルダを指定する。iters は、各要素がパラメータ名、列の型、動かす範囲 (g) で構成されるリストである。g(simulator) はジェネレータであり、そのジェネレータの生成する値の組み合わせを全て実行する。ただし実行回数は最大で maxcount(simulator) 回までとなる。outputs は、各要素が列名、列の型、式 (expression) で構成されるリストである。expression(simulator) で求まる結果が指定した列に記録される。各シミュレーションの終了時間は until(simulator) となる。一括実行を行うと、出力フォルダに「一括実行結果」という名前のモニターが出力される。

5.1.3 シミュレーションパラメータ

API SimulatorParam()

シミュレーションで使うパラメータを作成する。シミュレーションパラメータは任意個のメンバーを持ち、シミュレーター内の部品はそれらのメンバーをシミュレーションパラメータとして参照する。一括実行する時は、指定したパラメータを指定した範囲動かしシミュレーションを行うが、指定しなかったパラメータは毎回初期化される。

5.2 フロー部品

5.2.1 部品

API Widget(parent, name)

部品の作成する。parent にはシミュレーターを指定する。Widget は基底クラスであり必ず継承して使う。

API `widget.addMonitor()`

モニターを登録する。

API `widget.run()`

部品の動作定義。

API `widget.start()`

部品の起動する。

5.2.2 フローアイテム

API `FlowItem(param, name = None, **keys)`

名前が `name` のフローアイテムを作成する。`param` にはシミュレーションパラメータを指定する。`keys` に指定した任意個の属性名と属性値を持つ。

API `flowItem.getAttributes()`

フローアイテムの属性名のリストを返す。

API `flowItem.getTime(tag)`

フローアイテムにタグ `tag` で記録された時間を取得する

API `flowItem.recoredTime(tag, time)`

フローアイテムにタグ `tag` の時間に `time` を設定する。

5.2.3 フローアイテムリスト

API `FlowItemList(items = [], name = None, **keys)`

フローアイテムのリスト items から構成されるフローアイテムリストを作成する。name には名前を指定する。keys に指定した任意個の属性名と属性値を持つ。フローアイテムと Python の list オブジェクトを継承しており、フローアイテムの操作と list に対する全ての操作を行う事が出来る。

5.2.4 リンク

API `Link(outport, inport)`

出力ポート outport と入力ポート inport をリンクで結ぶ。

5.2.5 出力ポート

API `OutputPort(parent, name = '', size = 0, save = False, selector = None)`

部品 parent に名前 name の出力ポートを作成する。キューサイズは size となり、セレクタは selector となる。save が真ならキューの変化を記録し、シミュレーション終了後に出力される。

API `outputPort.send(item)`

出力ポートからフローアイテム item を出力する。出力ポートにキューがない場合は、セレクタによってリンクが選ばれ、リンクの先の入力ポートキューに追加するような名前 name の待ち受けを行う。出力ポートにキューがある場合は、出力ポートキューに追加するような待ち受けを行う。

5.2.6 入力ポート

API `InputPort(parent, name = '', size = 0, save = False, selector = None, init = [])`

部品 parent に名前 name の入力ポートを作成する。キューサイズは size となり、セレクタは selector となる。save が真ならキューの変化を記録し、シミュレーション終了後に出力される。init 引数には値と優先度からなるタプルのリスト [(値 1, 優先度 1), (値 2, 優先度 2), ...] を指定する事が出来る。空リストでない場合、シミュレーション開始時に各要素に対し、入力ポートのキューに対し store.put1(値, priority = 優先度) が実行される。キューサイズ以上の要素を指定した場合、追加キューに並ぶ。

API `inputPort.receive(name = 'item')`

入力ポートからフローアイテムを入力する。入力ポートと出力ポートにキューがない場合は、入力ポートキューからフローアイテムを取得するような名前 name の待ち受けを行う。入力ポートにキューがなく、出力ポートにキューがある場合は、セレクタによってリンクを選び、リンクの先の出力ポートからフローアイテムを取得するような名前 name の待ち受けを行う。入力ポートにキューがない場合は、入力ポートのキューからフローアイテムを取得するような名前 name の待ち受けを行う。

5.3 セレクタ

入力ポートのセレクタの場合、その入力ポートに入力する複数のリンク $\{L_i\}$ $(1 \leq i \leq n)$ から、セレクタはひとつの L_k を選択し、そのリンクの先の出力ポートからフローアイテムを入力する。

出力ポートのセレクタの場合、その出力ポートから出力する複数のリンク $\{L_i\}$ $(1 \leq i \leq n)$ から、セレクタはひとつの L_k を選択し、そのリンクの先の入力ポートへフローアイテムを出力する。

5.3.1 ランダム (一様)

API `selectRandomUnif(seed = None)`

確率 $P(L_i) = 1/n$ でリンク L_i を選ぶセレクタ。

5.3.2 ランダム (重み付き)

API `selectRandomProb(weights, seed = None)`

確率的にリンクを選ぶセレクタ。`weights` には重みリストを指定する。重みは `0` 以上でなくてはならない。重みを $\{w_i\}$ とすると、確率 $P(L_i) = w_i / \sum w_i$ でリンク L_i を選ぶ。

5.3.3 ラウンドロビン

API `selectRoundRobin()`

順番にリンクを選択するセレクタ。セレクタが呼ばれた回数を `k` (≥ 1) とすると `k` 番目にリンク $L_j, j = ((k-1) \bmod n) + 1$ を選ぶ。

5.3.4 固定

API `selectFix(portno)`

常に指定されたリンクを選ぶセレクタ。

5.3.5 最短キュー

API `selectShortestQueue()`

リンクの先のポートのキューが最も少ないリンクを選ぶセレクタ。最小キューサイズのリンクが複数ある場合は、セレクタが呼ばれた回数を `k` (≥ 1) とし、最小キューサイズのリンクのインデックス集合を $\{a_i\}$ とすると、リンク $L_{a_m}, m = \arg\min_i ((k - a_i + 1) \bmod n)$ を選択する。

5.3.6 強化学習

API `selectRL(model, linklist, idfunc, async_reward = False, reward_func_name = u''reward_func'')`

強化学習によりリンクを選ぶセレクタ。model には利用する強化学習モデルに対応する RLModelSet のオブジェクトを指定する。 linklist は行動とリンクの対応をあらわすリストである。 linklist[i] に i 番目の行動に対応するリンクの番号 (0 以上 $n-1$ 以下) が格納されているとする。 idfunc はフローアイテムを引数にとり、強化学習におけるエピソードを区別する文字列を返す関数である。 async_reward は即時報酬計算のタイミングをずらすかどうかをあらわす真偽値である。 reward_func_name は即時報酬計算のタイミングをずらす際に報酬決定関数をフローアイテムの属性に設定する際の属性名をあらわす名前である。

第6章　データ

6.1　モニターの操作

シミュレーション内での様々な変数を観測するために、モニターと時系列モニターがある。モニターは、任意の観測結果を記録するために用いる。時系列モニターは、任意の時間変動するパラメータの時系列変動を記録するために用いる事が出来る。

6.1.1　モニターの作成

モニターは、任意の観測結果を記録するために用いる。複数のパラメータを同時に記録する事が出来る。m 個のパラメータセットを、n 回観測した結果は以下のようになる。

	パラメータ$_0$	パラメータ$_1$	$\cdots$	パラメータ$_{m-1}$
観測$_0$	$X_{0,0}$	$X_{0,1}$	$\cdots$	$X_{0,m-1}$
観測$_1$	$X_{1,0}$	$X_{1,1}$	$\cdots$	$X_{1,m-1}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
観測$_{n-1}$	$X_{n-1,0}$	$X_{n-1,1}$	$\cdots$	$X_{n-1,m-1}$

API　`Monitor(cols = [], types = [], name = "", basedir = None)`

モニターを作成する。`cols` にはパラメータ名のリストを指定する。`types` には、各パラメータの型名のリストを指定する。型名には `"f"`, `"i"`, `"o"` を指定する事が出来る。`types` が省略された場合、全てのパラメータは `"f"` 型である事を意味する。`types` を明示的に指定する場合は、指定個数はパラメータ数と一致しなくてはならない。`name` はモニターの名前を指定する。`basedir` を指定した場合、ディレクトリ `basedir` から名前 `name` のモニターをロードする。その場合、`cols`, `types` は指定しない。

6.1.2 モニターへの観測結果の記録

API `monitor.observe(*values)`

モニターに観測結果を追加する。`values` の個数は、モニターのパラメータ数と一致しなくてはならない。

6.1.3 時系列モニターの作成

時系列モニター は、時間変動するパラメータの時系列変動を記録する事が出来る。現時刻と、複数のパラメータを同時に記録する事が出来る。m 個のパラメータセットを、n 回観測した結果は以下のようになる。

	時刻	パラメータ$_0$	パラメータ$_1$	$\cdots$	パラメータ$_{m-1}$
観測$_0$	t_0	$X_{0,0}$	$X_{0,1}$	$\cdots$	$X_{0,m-1}$
観測$_1$	t_1	$X_{1,0}$	$X_{1,1}$	$\cdots$	$X_{1,m-1}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
観測$_{n-1}$	t_{n-1}	$X_{n-1,0}$	$X_{n-1,1}$	$\cdots$	$X_{n-1,m-1}$

API `TimeMonitor(cols = [], types = [], name = "", basedir = None)`

時系列モニターを作成する。`cols` にはパラメータ名のリストを指定する。`types` には、各パラメータの型名のリストを指定する。型名には `"f"`, `"i"`,`"o"` を指定する事が出来る。`types` が省略された場合、全てのパラメータは`"f"`型である事を意味する。`types` を明示的に指定する場合は、指定個数はパラメータ数と一致しなくてはならない。`name` は時系列モニターの名前を指定する。`basedir` を指定した場合、ディレクトリ `basedir` から名前 `name`の 時系列モニターをロードする。その場合、`cols`, `types` は指定しない。

6.1.4 時系列モニターへの観測結果の記録

API `timeMonitor.observe(time, *values)`

時系列モニターに観測結果を追加する。time は時刻を指定する。必ず昇順でなくてはならない。通常は now() を指定する。values の個数は、時系列モニターのパラメータ数と一致しなくてはならない。

6.1.5 観測列の型

モニター, 時系列モニターは、複数の観測列で構成されるが、各観測列の型には、以下が用意されている。型名はモニター, 時系列モニターで指定する型名である。

型名	要素の型
"f"	浮動小数点型
"i"	整数型
"o"	文字列型

6.1.6 モニターからの観測列の取得

API `monitor[パラメータ名]`

モニターからパラメータ名で示される観測列を取得する。存在しないパラメータの場合はエラーとなる。複数存在するパラメーター名の場合、列番号の小さなものとなる。取得した観測列は、モニターの観測列のコピーである。

API `monitor[i]`

モニターから i 番目の観測列を取得する。存在しない列番号の場合はエラーとなる。負の数が指定された場合は (観測列数 + i) 列を意味する。取得した観測列は、モニターの観測列のコピーである。

いずれの方法で取得しても、取得した観測列は、モニターの観測列のコピーである。つまり、取得した観測列に変更を加えても、元のモニターは変更されない。ただし、取得した時点で全要素がコピーされるわけではない。何ら

かの観測列の更新が発生するまで、全要素のコピーは遅延する。更新がない場合は、全要素のコピーは発生しない。

6.1.7 時系列モニターからの観測列の取得

API `timeMonitor[パラメータ名]`

時系列モニターからパラメータ名で示される観測列を取得する。観測列は時系列データであり、観測列と時刻列の組で構成される。存在しないパラメータの場合はエラーとなる。複数存在するパラメーター名の場合、列番号の小さなものとなる。取得した観測列は、時系列モニターの観測列のコピーである。時刻列もコピーされる。

API `timeMonitor[i]`

時系列モニターから `i` 番目の観測列を取得する。観測列は時系列データであり、観測列と時刻列の組で構成される。存在しない列番号の場合はエラーとなる。負の数が指定された場合は（観測列数 `+ i`）列を意味する。取得した観測列は、時系列モニターの観測列のコピーである。時刻列もコピーされる。

モニターの場合と同じく、取得した観測列は、時系列モニターの観測列のコピーである。つまり、取得した観測列、時刻列に変更を加えても、元の時系列モニターは変更されない。

6.1.8 時系列モニターからの時刻列の取得

API `timeMonitor.time()`

時系列モニターの時刻列のコピーを返す。

6.1.9 列指定子

モニターの複数の観測列を指定するために、図 6.1 の列指定子がある。ここで、モニターの列数は C とする。列指定子においては負の列は最後の列からのインデックスを示す。つまり、列指定子における列 i は、実際の列 $M_C(i)$ を示す。

$$M_C(i) = \begin{cases} 0 & (i < -C) \\ C + i & (-C \le i < 0) \\ i & (0 \le i < C) \\ C & (C \le i) \end{cases}$$

intVector	整数型観測列 $\{I_k\}$ に対し、$\{M_C(I_k)\}$ が列番号列を示す。ただし、$-C \le I_j < C$ 以外の要素があった場合は無効な列指定子である。
`[I`$_1$`, I`$_2$`, ⋯, I`$_n$`]`	リストの各要素に対し、$\{M_C(I_k)\}$ が列番号列を示す。ただし、$-C \le I_j < C$ 以外の要素があった場合は無効な列指定子である。
start:end	$\{k \mid M_C(\mathrm{start}) \le k < M_C(\mathrm{end})\}$ の列番号列を示す。
start:	$\{k \mid M_C(\mathrm{start}) \le k < C\}$ の列番号列を示す。
:end	$\{k \mid 0 \le k < M_C(\mathrm{end})\}$ の列番号列を示す。
start:end:step	$\{k = M_C(\mathrm{start}) + i\,\mathrm{step} \mid M_C(\mathrm{start}) \le k < M_C(\mathrm{end}), i \ge 0\}$ の列番号列を示す。
:	全ての列を示す。
...	全ての列を示す。

図 6.1: 列指定子

6.1.10 行指定子

モニターの複数の観測行を指定するために、図 6.2 の行指定子がある。ここで、モニターの行数は R とする。行指定子においては負の行は最後の行からのインデックスを示す。つまり、行指定子における行 i は、実際の行 $M_R(i)$ を示す。

$$M_R(i) = \begin{cases} 0 & (i < -R) \\ R+i & (-R \leq i < 0) \\ i & (0 \leq i < R) \\ R & (R \leq i) \end{cases}$$

6.1.11 モニターからの部分抽出 (列指定)

単一の観測列を取得したい場合は、6.1.6 章の方法が利用出来るが、複数の観測列を取得したい場合は以下を利用する。

API `monitor[列指定子]`

monitor の列指定子で示される列を抽出し、新規のモニターを返す。列指定子としては 6.1.9 章の指定方法が利用出来る。各列はコピーされる。無効な指定子の場合はエラーとなる。

6.1.12 時系列モニターからの部分抽出 (列指定)

API `timeMonitor[列指定子]`

timeMonitor の列指定子で示される列を抽出し、新規の時系列モニターを返す。列指定子としては 6.1.9 章の指定方法が利用出来る。各列および時刻列はコピーされる。無効な指定子の場合はエラーとなる。

6.1.13 モニターからの部分抽出 (行、列指定)

モニターから一部の列のみを抽出したい場合は、6.1.11 章の方法が利用出来るが、更に行番号を指定した抽出を行いたい場合は、以下を利用する。

API `monitor[行指定子, 列指定子]`

monitor の列指定子及び行指定子で示される部分を抽出し、新規のモニターを返す。列指定子としては 6.1.9 章の指定方法が利用出来る。行指定子としては 6.1.10 章の指定方法が利用出来る。指定された部分はコピーされる。無効な列指定子あるいは行指定子の場合はエラーとなる。

なお、6.1.6 章の方法は、観測列を返すが、以上の方法による行指定も可能である。ただし、その場合の結果は、モニターでなく、観測列が返る。更に例外として、monitor[row, パラメータ名]、monitor[row, col] の場合は、指定された観測値が返る。

6.1.14 時系列モニターからの部分抽出 (行、列指定)

API timeMonitor[行指定子, 列指定子]

timeMonitor の列指定子及び行指定子で示される部分を抽出し、新規の時系列モニターを返す。列指定子としては 6.1.9 章の指定方法が利用出来る。行指定子としては 6.1.10 章の指定方法が利用出来る。指定された部分はコピーされる。時刻列も指定された行のみがコピーされる。無効な列指定子あるいは行指定子の場合はエラーとなる。

API timeMonitor[TimeInterval(開始時刻, 終了時刻), 列指定子]

timeMonitor の列指定子で指定される列の開始時刻から終了時刻までの間の部分を抽出し、新規の時系列モニターを返す。列指定子としては 6.1.9 章の指定方法が利用できる。指定された部分はコピーされる。時刻列は開始時刻から終了時刻までの時刻列が作成される。ただし、開始時刻が timeMonitor の最初の時刻より前の場合は、開始時刻は最初の時刻に置き換えられる。無効な列指定子や、終了時刻が開始時刻より前であるような場合はエラーとなる。

6.1.15 モニターへの部分更新 (列指定)

API monitor[列指定子] = $monitor_2$

monitor の列指定子で指定された部分を $monitor_2$ で置き換える。列指定子としては 6.1.9 章の指定方法が利用出来る。無効な列指定子の場合はエラーとなる。monitor と $monitor_2$ の列数あるいは行数が違う場合はエラーとなる。

API monitor[列指定子] = [[$\mathtt{val}_{0,0}$, $\mathtt{val}_{1,0}$, $\cdots$, $\mathtt{val}_{n-1,0}$], [$\mathtt{val}_{0,1}$, $\mathtt{val}_{1,1}$, $\cdots$, $\mathtt{val}_{n-1,1}$], $\cdots$, [$\mathtt{val}_{0,m-1}$, $\mathtt{val}_{1,m-1}$, $\cdots$, $\mathtt{val}_{n-1,m-1}$]]

monitor の列指定子で指定された部分を指定したリストで置き換える。列指定子としては 6.1.9 章の指定方法が利用出来る。リストは、k 列を表現するリスト [$\mathtt{val}_{0,k}$, $\mathtt{val}_{1,k}$, $\cdots$, $\mathtt{val}_{n-1,k}$] のリストである。無効な列指定子の場合はエラーとなる。monitor と $\mathtt{monitor}_2$ の列数あるいは行数が違う場合はエラーとなる。

API monitor[列指定子] = val

monitor の列指定子で指定された列の中身を全て val にする。列指定子としては 6.1.9 章の指定方法が利用出来る。無効な列指定子の場合はエラーとなる。

6.1.16 時系列モニターへの部分更新 (列指定)

6.1.15 章と同様の方法が利用出来る。

6.1.17 モニターへの部分更新 (行、列指定)

モニターの一部の列のみを更新したい場合は、6.1.15 章の方法が利用出来るが、更に行番号を指定した更新を行いたい場合は、以下を利用する。

API monitor[行指定子, 列指定子] = $\mathtt{monitor}_2$

monitor の列指定子及び行指定子で指定された部分を $\mathtt{monitor}_2$ で置き換える。列指定子としては 6.1.9 章の指定方法が利用出来る。行指定子としては 6.1.10 章の指定方法が利用出来る。無効な列指定子あるいは行指定子の場合はエラーとなる。monitor と $\mathtt{monitor}_2$ の列数あるいは行数が違う場合はエラーとなる。

API monitor[行 指 定 子, 列 指 定 子] = [[val$_{0,0}$, val$_{1,0}$, $\cdots$, val$_{n-1,0}$], [val$_{0,1}$, val$_{1,1}$, $\cdots$, val$_{n-1,1}$], $\cdots$, [val$_{0,m-1}$, val$_{1,m-1}$, $\cdots$, val$_{n-1,m-1}$]]

monitor の列指定子及び行指定子で指定された部分を monitor$_2$ で置き換える。列指定子としては 6.1.9 章の指定方法が利用出来る。行指定子としては 6.1.10 章の指定方法が利用出来る。リストは、k 列を表現するリスト [val$_{0,k}$, val$_{1,k}$, $\cdots$, val$_{n-1,k}$] のリストである。無効な列指定子あるいは行指定子の場合はエラーとなる。monitor と指定されたリストの列数あるいは行数が違う場合はエラーとなる。

API monitor[行指定子, 列指定子] = val

monitor の列指定子及び行指定子で指定された部分を val で置き換える。列指定子としては 6.1.9 章の指定方法が利用出来る。行指定子としては 6.1.10 章の指定方法が利用出来る。無効な列指定子あるいは行指定子の場合はエラーとなる。

6.1.18 時系列モニターへの部分更新 (行、列指定)

6.1.17 章と同様の方法が利用出来る。時刻列は変化しない。

6.1.19 モニターからの部分削除 (列指定)

API del monitor[列指定子]

monitor の列指定子で指定された部分を削除する。列指定子としては 6.1.9 章の指定方法が利用出来る。無効な列指定子の場合はエラーとなる。

6.1.20 時系列モニターからの部分削除 (列指定)

6.1.19 章と同様の方法が利用出来る。時刻列は変化しない。

6.1.21 モニターからの部分削除 (行指定)

API `del monitor[行指定子,:]`

`monitor` の行指定子で指定された部分を削除する。行指定子としては 6.1.10 章の指定方法が利用出来る。無効な行指定子の場合はエラーとなる。

API `del monitor[行指定子,...]`

`monitor` の行指定子で指定された部分を削除する。行指定子としては 6.1.10 章の指定方法が利用出来る。無効な行指定子の場合はエラーとなる。

6.1.22 時系列モニターからの部分削除 (行指定)

6.1.21 章と同様の方法が利用出来る。指定された行の時刻列も削除される。

6.1.23 モニターの保存

API `monitor.save(basedir = ".")`

`monitor` をディレクトリ `basedir` に保存する。名前は `monitor` の名前になる。

6.1.24 時系列モニターの保存

API `timeMonitor.save(basedir = ".")`

`monitor` をディレクトリ `basedir` に保存する。名前は `monitor` の名前になる。

6.1.25 モニターのその他の操作

API `monitor.ncol()`

`monitor` の観測列数を返す。

API `monitor.nrow()`

`monitor` の観測数を返す。

API `monitor.dim()`

`monitor` の観測列数と観測数をタプルの形で返す。

API `monitor.names()`

`monitor` の観測列の名前のリストを返す。

API `monitor.setName(i, name)`

`monitor` の `i` 番目の観測列の名前を `name` に変更する。

API `monitor.toList()`

`monitor` の リ ス ト 表 現 [[val$_{0,0}$, val$_{1,0}$, $\cdots$, val$_{n-1,0}$], [val$_{0,1}$, val$_{1,1}$, $\cdots$, val$_{n-1,1}$], $\cdots$, [val$_{0,m-1}$, val$_{1,m-1}$, $\cdots$, val$_{n-1,m-1}$]] を返す。

6.2 モニターの列の操作

通常、観測列は、6.1.6 章、6.1.7 の方法で、モニターあるいは時系列モニターから取得する。

観測列には、

- `FloatVector`
- `IntVector`
- `ObjectVector`
- `BoolVector`

がある。時系列観測列には、

- `TimeFloatVector`
- `TimeIntVector`

- TimeObjectVector
- TimeBoolVector

がある。特殊な時系列観測列として、

- TimeVector

がある。TimeVector は時刻列の型であるが、TimeFloatVector の特殊な場合で、自分の時刻列が自分自身であるような観測列である。

6.2.1 観測列の作成

API FloatVector(init = [])

浮動小数点型観測列を作成する。init が指定された場合は、そのリストの内容で初期化される。

API IntVector(init = [])

整数型観測列を作成する。init が指定された場合は、そのリストの内容で初期化される。

API ObjectVector(init = [])

文字列型観測列を作成する。init が指定された場合は、そのリストの内容で初期化される。

6.2.2 観測列の基本統計量の取得

API floatVec.[min|max|range|count|mean|var|sd|cv]()

それぞれ、浮動小数点型観測列の最小値 (min)、最大値 (max)、範囲 (range)、観測数 (count)、平均 (mean)、分散 (var)、標準偏差 (sd)、変動係数 (cv) を返す。

観測列を $\{x_i\}$、観測数を n とすると、以下のように定義される。

$$
\begin{aligned}
\texttt{range} &= \max_{0 \le i \le n-1}(x_i) - \min_{0 \le i \le n-1}(x_i) \\
\texttt{mean} &= \frac{1}{n}\sum_{i=0}^{n-1} x_i
\end{aligned}
$$

$$\begin{aligned} \texttt{var} &= \frac{1}{n-1}\sum_{i=0}^{n-1}(x_i - \texttt{mean})^2 \\ \texttt{sd} &= \sqrt{\texttt{var}} \\ \texttt{cv} &= \frac{\texttt{sd}}{\texttt{mean}} \end{aligned}$$

API `floatVec.confInterval(p)`

浮動小数点型観測列の母平均の p * 100% 信頼区間をタプル（v1, v2）の形で返す。$(0 < p < 1)$

$t_{\nu,\alpha/2}$ は自由度 ν の t 分布の、片側確率が $\alpha/2 * 100\%$ となる t 値とすると、信頼区間は以下となる。

$$\texttt{confInterval}(p) = \texttt{mean} \pm t_{\texttt{count}-1,(1-p)/2}\sqrt{\frac{\texttt{var}}{\texttt{count}}}$$

API `intVec.[min|max|range|count|mean|var|sd|cv]()`

それぞれ、整数型観測列の最小値 (min)、最大値 (max)、範囲 (range)、観測数 (count)、平均 (mean)、分散 (var)、標準偏差 (sd)、変動係数 (cv) を返す。

計算式は浮動小数点型観測列の場合と同じである。

API `intVec.confInterval(p)`

整数型観測列の母平均の p % 信頼区間をタプル（v1, v2）の形で返す。$(0 < p < 1)$

計算式は浮動小数点型観測列の場合と同じである。

API `objectVec.[mode|modefreq|freq]()`

それぞれ、文字列型観測列の最頻値 (mode)、最頻値の頻度 (modefreq)、観測数 (count) を返す。

API `timeFloatVec.[min|max|range|count|mean|var|sd|cv]()`

それぞれ、浮動小数点型時系列観測列の最小値 (min)、最大値 (max)、範囲 (range)、有効観測数 (count)、平均 (mean)、分散 (var)、標準偏差 (sd)、変動係数 (cv) を返す。

観測列を $\{x_i\}$、時間列を $\{t_i\}$、観測数を n とすると、有効観測数は n-1 となり、以下のように定義される。

$$
\begin{aligned}
\texttt{count} &= n-1 \\
\texttt{range} &= \max_{0\le i\le n-2}(x_i) - \min_{0\le i\le n-2}(x_i) \\
\texttt{mean} &= \frac{1}{t_{n-1}-t_0}\sum_{i=0}^{n-2}(t_{j+1}-t_j)x_i \\
\texttt{var} &= \frac{t_{n-1}-t_0}{(t_{n-1}-t_0)^2-\sum_{i=0}^{n-2}(t_{i+1}-t_i)^2}\sum_{i=0}^{n-2}(t_{i+1}-t_i)(x_i-\texttt{mean})^2 \\
\texttt{sd} &= \sqrt{\texttt{var}} \\
\texttt{cv} &= \frac{\texttt{sd}}{\texttt{mean}}
\end{aligned}
$$

API `timeFloatVec.confInterval(p)`

浮動小数点型時系列観測列の母平均の p % 信頼区間をタプル（v1, v2）の形で返す。($0 < p < 1$)

計算式は、浮動小数点型観測列の場合と同様であるが、前述の基本統計量を用いる。

API `timeIntVec.[min|max|range|count|mean|var|sd|cv]()`

それぞれ、整数型時系列観測列の最小値 (min)、最大値 (max)、範囲 (range)、観測数 (count)、平均 (mean)、分散 (var)、標準偏差 (sd)、変動係数 (cv) を返す。

計算式は浮動小数点型時系列観測列の場合と同じである。

API `timeIntVec.confInterval(p)`

整数型時系列観測列の母平均の p% 信頼区間をタプル（v1, v2）の形で返す。($0 < p < 1$)

計算式は浮動小数点型時系列観測列の場合と同じである。

API `timeObjectVec.[mode|modefreq|count]()`

それぞれ、文字列型時系列観測列の最頻値 (mode)、最頻値の頻度 (modefreq)、観測数 (count) を返す。

観測列の基本統計量の結果は可能な限りキャッシュされる。観測列をコピーしても基本統計量に変化はないので、キャッシュ結果が利用される。観測列に変更があった場合は、再度計算を行う。また、最小値、最大値、平均は同

時に求められるので、最小値を取得した後に、最大値を取得した場合はキャッシュ結果が利用される。

6.2.3 観測列の基本統計量のサマリ

API `floatVec.summary()`

浮動小数点型観測列の基本統計量のサマリを文字列で返す。

API `intVec.summary()`

整数型観測列の基本統計量のサマリを文字列で返す。

API `objectVec.summary()`

文字列型観測列の基本統計量のサマリを文字列で返す。

API `timeFloatVec.summary()`

浮動小数点型時系列観測列の基本統計量のサマリを文字列で返す。

API `timeIntVec.summary()`

整数型時系列観測列の基本統計量のサマリを文字列で返す。

API `timeObjectVec.summary()`

文字列型時系列観測列の基本統計量のサマリを文字列で返す。

浮動小数点型と、整数型の場合、最小値、最大値、範囲、観測数、平均、分散、標準偏差、変動係数、信頼区間が含まれる。

文字列型の場合、観測数、種別数、最頻値、最頻値の頻度が含まれる。

時系列型の基本統計量は、出現頻度ではなく、観測列に含まれる値をとった時間を元に算出される。

6.2.4 観測列のヒストグラム

API `floatVec.histogram(nbin = False, frm = False, to = False)`

浮動小数点型観測列の最小階級値 `frm` から 最大階級数 `to` までを `nbin` 個に分割したヒストグラムオブジェクトを返す。`nbin` が省略された場合は $\lfloor 1 + \log_2(n) \rfloor$ となる。`frm` が省略された場合は観測列の最小値、`to` が省略された場合は観測列の最大値となる。

API `intVec.histogram(nbin = False, frm = False, to = False)`

整数型観測列の最小階級値 `frm` から 最大階級数 `to` までを `nbin` 個に分割したヒストグラムオブジェクトを返す。`nbin` が省略された場合は $\lfloor 1 + \log_2(n) \rfloor$ となる。`frm` が省略された場合は観測列の最小値、`to` が省略された場合は観測列の最大値となる。

API `objectVec.histogram(maxfactors = False)`

文字列型観測列の出現頻度の多い順に並び換えたヒストグラムオブジェクトを返す。`maxfactors` は最大数を指定する。指定されなかった場合は全てのオブジェクトを対象とする。

API `timeFloatVec.histogram(nbin = False, frm = False, to =False)`

浮動小数点型時系列観測列の最小階級値 `frm` から 最大階級数 `to` までを `nbin` 個に分割した滞在時間のヒストグラムオブジェクトを返す。`nbin` が省略された場合は $\lfloor 1 + \log_2(n) \rfloor$ となる。`frm` が省略された場合は観測列の最小値、`to` が省略された場合は観測列の最大値となる。

API timeIntVec.histogram(nbin = False, frm = False, to =False)

整数型時系列観測列の最小階級値 frm から 最大階級数 to までを nbin 個に分割した滞在時間のヒストグラムオブジェクトを返す。nbin が省略された場合は $\lfloor 1+\log_2(n) \rfloor$ となる。frm が省略された場合は観測列の最小値、to が省略された場合は観測列の最大値となる。

API timeObjectVec.histogram(maxfactors = False)

文字列型時系列観測列の滞在時間の長い順に並び換た滞在時間のヒストグラムオブジェクトを返す。maxfactors は最大数を指定する。指定されなかった場合は全てのカテゴリー文字列を対象とする。

時系列型の観測列のヒストグラムは、出現頻度ではなく、観測列に含まれる値をとった時間を元に算出される。

ヒストグラムオブジェクトを print すればヒストグラムが表示される。

6.2.5 時系列観測列からの時刻列の取得

API timeVec.time()

時系列観測列の時刻列のコピーを返す。

6.2.6 観測列の 2 項演算

API vec_1 · vec_2

観測列の各要素を 2 項演算子 · で演算し、その結果の観測列を返す。2 項演算子には、+, --, *, /, %,**, &, |, ^ , >>, << がある。結果の観測列の型は vec_1 の型となる。

API vec_1 · [val_1, val_2, $\cdots$, val_n]

観測列 vec_1 と、リスト [val_1, val_2, $\cdots$, val_n] の各要素を 2 項演算子 · で演算し、その結果の観測列を返す。2 項演算子には、+, --, *, /, %, **, &, |, ^, >>, << がある。結果の観測列の型は vec_1 の型となる。

API vec$_1$ · val

観測列 vec$_1$ の各要素と val を 2 項演算子 · で演算し、その結果の観測列を返す。2 項演算子には、+, --,*, /, %, **, &, |, ^, >>, << がある。結果の観測列の型は vec$_1$ の型となる。

API [val$_1$, val$_2$, ···, val$_n$] · vec$_1$

リスト [val$_1$, val$_2$, ···, val$_n$] と、観測列 vec$_1$ の各要素を 2 項演算子 · で演算し、その結果の観測列を返す。2 項演算子には、+, --, *, /, %, **, &, |, ^, >>, << がある。結果の観測列の型は vec$_1$ の型となる。

API val · vec$_1$

val と 観測列 vec$_1$ の各要素を 2 項演算子 · で演算し、その結果の観測列を返す。2 項演算子には、+, --,*, /, %, **, &, |, ^, >>, << がある。結果の観測列の型は vec$_1$ の型となる。

6.2.7 時系列観測列の 2 項演算

時系列観測列は、観測列と時刻列の組、すなわち、

$$(\{x_i\}, \{t_i\})$$

で表わされるが、

$$x(t) = x_i \quad (i = \max_{t_k \leq t} k)$$

のような関数としても表記する事が出来る。

ここで、時系列観測列同士の 2 項演算子 · を以下のように定義する。

任意の $(\{x_i\}, \{t_i\})$、$(\{y_i\}, \{u_i\})$ に対し、$(\{x_i\}, \{t_i\}) \cdot (\{y_i\}, \{u_i\}) = (\{z_i\}, \{w_i\})$ とすると、$(\{z_i\}, \{w_i\})$ は以下を満たす。

$$\begin{aligned} x(t) \cdot y(t) &= x_j \cdot y_l \quad (j = \max_{t_k \leq t} k, l = \max_{u_k \leq t} k) \\ &= z_i \quad (i = \max_{w_k \leq t} k) \end{aligned}$$

必ず、$\|\{w_i\}\| = \|\{z_i\}\| \leq \|\{t_i\} \cup \{u_i\}\|$ となる。

この定義の下、時系列観測列に対して、6.2.6 章と同様の 2 項演算が可能である。

時系列観測列 · 観測列、あるいは、観測列 · 時系列観測列の場合は、双方を同じ時刻列のものとして演算を行う。

6.2.8 観測列の累算演算

API vec$_1$ ·= vec$_2$

観測列の vec$_1$ と vec$_2$ の各要素を 2 項演算子 · で演算し、その結果を vec$_1$ に反映する。2 項演算子に = をつなげたものを累積演算子と呼ぶ。累積演算子には、+=,--=, *=, /=, &=, |=, ^, >>=, <<= がある。結果の観測列の型は vec$_1$ の型となる。

API vec$_1$ ·= [val$_1$, val$_2$, ···, val$_n$]

観測列の vec$_1$ とリスト [val$_1$, val$_2$, ···, val$_n$] の各要素を 2 項演算子 · で演算し、その結果を vec$_1$ に反映する。2 項演算子に = をつなげたものを累積演算子と呼ぶ。累積演算子には、+=,--=, *=, /=, &=, |=, ^, >>=, <<= がある。結果の観測列の型は vec$_1$ の型となる。

API vec$_1$ ·= val

観測列の vec$_1$ の各要素 と val を 2 項演算子 · で演算し、その結果を vec$_1$ に反映する。2 項演算子に = をつなげたものを累積演算子と呼ぶ。累積演算子には、+=, --=,*=, /=, &=, |=, ^, >>=, <<= がある。結果の観測列の型は vec$_1$ の型となる。

6.2.9 時系列観測列の累算演算

6.2.8 章と同様の方法が利用出来る。時系列は変化しない。

6.2.10 観測列の単項演算

API +vec

観測列 vec を返す。

API -vec

観測列 vec の各要素の符号を反転した観測列を返す。

API vec

観測列 vec の各要素のビットを符号を反転した観測列を返す。

API abs(vec)

観測列 vec の各要素の絶対値で構成される観測列を返す。

6.2.11 時系列観測列の単項演算

時系列観測列に対しても、6.2.10 章の演算が可能である。

6.2.12 観測列の数値演算

API op(vec)

観測列 vec の各要素 x_i に対し、op(x_i) の演算を行った観測列を返す。op には、ceil, fabs, floor, exp, log10,sqrt, acos, atan, cos, sin, tan, degrees, raidans, cosh, sinh, tanh がある。

API op2(vec_1, vec_2)

観測列 vec_1 の各要素 x_i と観測列 vec_2 の各要素 y_i に対し、op2(x_i, y_i) の演算を行った観測列を返す。op には、log, fmod, atan2, hypot がある。

6.2.13 時系列観測列の数値演算

時系列観測列に対しても、6.2.12 章の演算が可能である。

2 項演算の場合の演算方法は 6.2.7 章と同様である。時系列観測列と観測列の組み合わせの場合は、双方を同じ時刻列のものとして演算を行う。

6.2.14 観測列の比較演算

観測列には比較演算を行う事が出来る。結果は BoolVector である。BoolVector を用いて、モニターや観測列の抽出を行う事が出来る。

API vec_1 · vec_2

観測列の各要素を比較演算子 · で比較し、その結果の BoolVector を返す。比較演算子には、>, >=, <,<=, ==, != がある。

API vec_1 · [val_1, val_2, ···, val_n]

観測列 vec_1 と、リスト [val_1, val_2, ···, val_n] の各要素を比較演算子 · で比較し、その結果の BoolVector を返す。比較演算子には、>, >=, <, <=, ==, != がある。

API vec_1 · val

観測列 vec_1 の各要素と val を比較演算子 · で比較し、その結果の BoolVector を返す。比較演算子には、>, >=, <, <=, ==, != がある。

API [val_1, val_2, ···, val_n] · vec_1

リスト [val_1, val_2, ···, val_n] と、観測列 vec_1 の各要素を比較演算子 · で比較し、その結果の BoolVector を返す。比較演算子には、>, >=, <, <=, ==, != がある。

API val · vec_1

val と 観測列 vec_1 の各要素を比較演算子 · で比較し、その結果の BoolVector を返す。比較演算子には、>, >=, <, <=, ==, != がある。

6.2.15 時系列観測列の比較演算

時系列観測列に対しても、6.2.14 章の方法が利用出来る。ただし、結果は `BoolVector` に時刻列を付加された `TimeBoolVector` が返る。

時系列観測列同士の比較の場合は、2 項演算と同様に、行拡張が発生する。

時系列観測列 · 観測列、あるいは、観測列 · 時系列観測列の場合は、双方を同じ時刻列のものとして比較を行う。

6.2.16 観測列の部分抽出

API `vector[行指定子]`

`vector` の行指定子で示される部分を抽出し、新規の `Vector` を返す。行指定子としては 6.1.10 章の指定方法が利用出来る。指定された部分はコピーされる。無効な行指定子の場合はエラーとなる。

6.2.17 時系列観測列の部分抽出

6.2.16 章と同様の方法が利用出来る。時系列も抽出される。その他に次の方法も使用できる。

API `timeVector[TimeInterval(開始時刻, 終了時刻)]`

`timeVector` の開始時刻から終了時刻までの間の部分を抽出し、新規の時系列観測列を返す。指定された部分はコピーされる。時刻列は開始時刻から終了時刻までの時刻列が作成される。ただし、開始時刻が `timeVector` の最初の時刻より前の場合は、開始時刻は最初の時刻に置き換えられる。終了時刻が開始時刻より前であるような場合はエラーとなる。

6.2.18 観測列の部分更新

API `vector[行指定子] = vector`$_2$

`vector` の行指定子で指定された部分を `vector`$_2$ で置き換える。行指定子としては 6.1.10 章の指定方法が利用出来る。無効な行指定子の場合はエラーとなる。`vector` と `vector`$_2$ の行数が違う場合はエラーとなる。

API `vector[行指定子] = [val`$_1$`, val`$_2$`, ⋯,val`$_n$`]`

vector の行指定子で指定された部分を指定したリストで置き換える。行指定子としては 6.1.10 章の指定方法が利用出来る。無効な行指定子の場合はエラーとなる。vector とリストの行数が違う場合はエラーとなる。

API `vector[行指定子] = val`

vector の行指定子で指定された行の中身を全て val にする。行指定子としては 6.1.10 章の指定方法が利用出来る。無効な行指定子の場合はエラーとなる。

6.2.19 時系列観測列の部分更新

6.2.18 章と同様の方法が利用出来る。時系列は変化しない。

6.2.20 観測列の部分削除

API `del vector[行指定子]`

vector の行指定子で指定された部分を削除する。行指定子としては 6.1.10 章の指定方法が利用出来る。無効な行指定子の場合はエラーとなる。

6.2.21 時系列観測列の部分削除

6.2.20 章と同様の方法が利用出来る。時系列は変化しない。指定された行の時刻列も削除される。

6.2.22 観測列のその他の操作

階差

API `vector.diff(lag = 1, difference = 1)`

vector の階差数列を返す。lag は階差を取る間隔、difference は何回階差を取るかを指定する。lag と difference は 1 以上の整数でなくてはならない。なお、文字列型観測列については、この処理は行えない。

元の vector を $a_i^{(0)}\,(0 \le i < n)$ とし、lag を l,difference を d とすると、

$$a_i^{(d)} = a_{i+l}^{(d-1)} - a_i^{(d-1)}$$

になる。

部分和

API `vector.cumsum()`

vector の部分和の数列を返す。なお、文字列型観測列については、この処理は行えない。

元の vector を $a_i\,(0 \le i < n)$ とすると、

$$a'_i = \sum_{k=0}^{i} a_k$$

になる。

移動平均

API `vector.movingAverage(n)`

vector の単純移動平均 (直近の n 個の平均) の列を返す。なお、文字列型観測列については、この処理は行えない。

元の vector を $a_i\,(0 \le i < n)$ とすると、

$$a'_i = \frac{\sum_{k=i-n+1}^{i} a_k}{n}$$

になる。

分位値

API `vector.quantile(qs)`

vector の分位値のリストを返す。qs は 0 以上 1 以下の確率のリストを指定する。なお、文字列型観測列については、この処理は行えない。

q 分位値は、分布を q:(1-q) に分割する値である。$(0 \le q \le 1)$

$$
\begin{aligned}
Q_q &= \begin{cases} x_k & (k_0 = k_1) \\ (k_1 - k)x_{k_0} + (k - k_0)x_{k_1} & (k_0 \neq k_1) \end{cases} \\
k &= q(n-1) \\
k_0 &= \lfloor k \rfloor \\
k_1 &= \lceil k \rceil
\end{aligned}
$$

最も近い 2 点の値から、線形補完で内挿する。
例えば、$\{1, 2, 3, 4\}$ の中央値 $Q_{0.5} = 2.5$ となる。

挿入

API `vector.insert(n, val)`

`vector` の `n` 番目に、`val` を加える。この操作は破壊的である。

追加

API `vector.append(val)`

`vector` の末尾に、`val` を加える。この操作は破壊的である。

拡張

API `vector.extend(vector)`

`vector` の末尾に、`vector` あるいはリストを加える。この操作は破壊的である。

反転

API `vector.reverse()`

`vector` の順番を反転する。この操作は破壊的である。

整列

API `vector.sort(cmp = None, reverse = False)`

vector の順番を並び換える。cmp が指定されなかった場合、昇順に並び換える。cmp に-1, 0,1 を返す関数が指定された場合、その順番に並び換えられる。reverse が True の場合、順序は反転される。この操作は破壊的である。

共分散

API vector.cov($\mathtt{vector}_2$)

vector と $\mathtt{vector}_2$ の共分散を返す。vector と $\mathtt{vector}_2$ の長さは同じでなくてはならない。なお、文字列型観測列については、この処理は行えない。

vector を $x_i\,(0 \le i < n)$、$\mathtt{vector}_2$ を $y_i\,(0 \le i < n)$ とすると、

$$\frac{1}{n-1}\sum_{i=0}^{n-1}(x_i - \bar{x})(y_i - \bar{y})$$

になる。

相関係数

API vector.cor($\mathtt{vector}_2$)

vector と $\mathtt{vector}_2$ の積率相関係数を返す。vector と $\mathtt{vector}_2$ の長さは同じでなくてはならない。なお、文字列型観測列については、この処理は行えない。

vector を $x_i\,(0 \le i < n)$、$\mathtt{vector}_2$ を $y_i\,(0 \le i < n)$ とすると、

$$\frac{\sum_{i=0}^{n-1}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=0}^{n-1}(x_i - \bar{x})^2}\sqrt{\sum_{i=0}^{n-1}(y_i - \bar{y})^2}}$$

になる。

6.2.23 時系列観測列のその他の操作

階差

API `timeVector.diff(weightFirst = False)`

timeVector の階差モニターを返す。階差モニタは値列と重み列からなる。weightFirst が False なら、モニターの列は（値、重み）となり、True なら、モニターの列は（重み、値）となる。なお、文字列型時系列観測列については、この処理は行えない。

元の timeVector を $(\{x_i\}, \{t_i\})\,(0 \leq i < n)$ とし、重みを w_i，値を v_i とすると、

$$
\begin{aligned}
w_i &= \Delta t_i = t_{i+1} - t_i \\
v_i &= x_i
\end{aligned}
$$

となるような、モニターを返す。

移動平均

API `timeVector.movingAverage(time)`

timeVector の移動平均を返す。なお、文字列型時系列観測列については、この処理は行えない。

元の timeVector を $(\{x_i\}, \{t_i\})\,(0 \leq i < n)$ とし、time を s とすると、

$$
\begin{aligned}
x(t) &= x_i \quad (i = \max_{t_k \leq t} k) \\
T &= \max_{t_k \leq t_0 + s} k \\
x'_i &= \frac{1}{s} \int_{t_{i+t} - s}^{t_{i+t}} x dt \\
t'_i &= t_{i+T}
\end{aligned}
$$

となるような、timeVector (x'_i, t'_i) を返す。

分位値

API `timeVector.quantile(qs)`

`timeVector` の分位値のリストを返す。`qs` は `0` 以上 `1` 以下の確率のリストを指定する。なお、文字列型観測列については、この処理は行えない。

整列

API `timeVector.sort(cmp = None, reverse = False)`

`timeVector` の順番を並び換える。`cmp` が指定されなかった場合、昇順に並び換える。`cmp` に`-1, 0,1` を返す関数が指定された場合、その順番に並び換えられる。`reverse` が `True` の場合、順序は反転される。この操作は破壊的である。

元の `timeVector` を $(\{x_i\}, \{t_i\})\,(0 \le i < n)$ とする。
ここで、x_i を昇順に並びかえる。つまり、

$$x_{S_i} \le x_{S_j} \quad (0 \le i \le j \le n-2)$$

を見たすような、$\{S_i\}$ を求める。

$$\begin{aligned} x'_i &= x_{S_i} \\ u_i &= t_{S_{i+1}} - t_{S_i} \\ t'_i &= \sum_{k=0}^{i} u_i \end{aligned}$$

となるような、`timeVector` (x'_i, t'_i) になる。

共分散

API `timeVector.cov(timeVector`$_2$`)`

`timeVector` と `timeVector`$_2$ の共分散を返す。なお、文字列型観測列については、この処理は行えない。

`timeVector` を $(\{x_i\}, \{t_i\})\,(0 \le i < n)$、`timeVector`$_2$ を $(\{y_i\}, \{u_i\})\,(0 \le i < n)$ とする。

$(\{x_i\},\{t_i\})$ と $(\{y_i\},\{u_i\})$ を結合し、時間 $\max(\min_i t_i, \min_i u_i)$ から $\min(\max_i t_i, \max_i u_i)$ までの区間のみを取り出し $(\{x'_i\},\{y'_i\},\{t'_i\})\,(0 \le i < n')$ とする。その上で、以下を共分散とする。

$$\frac{t'_{n-1} - t'_0}{(t'_{n-1} - t'_0)^2 - \sum_{i=0}^{n'-2}(t'_{i+1} - t'_i)^2} \sum_{i=0}^{n-2} (t'_{i+1} - t'_i)(x'_i - \bar{x'})(y'_i - \bar{y'})$$

相関係数

API `timeVector.cor(timeVector₂)`

`timeVector` と `timeVector₂` の積率相関係数を返す。なお、文字列型観測列については、この処理は行えない。

`timeVector` を $(\{x_i\},\{t_i\})\,(0 \le i < n)$、`timeVector₂` を $(\{y_i\},\{u_i\})\,(0 \le i < n)$ とする。

$(\{x_i\},\{t_i\})$ と $(\{y_i\},\{u_i\})$ を結合し、時間 $\max(\min_i t_i, \min_i u_i)$ から $\min(\max_i t_i, \max_i u_i)$ までの区間のみを取り出し $(\{x'_i\},\{y'_i\},\{t'_i\})\,(0 \le i < n')$ とする。

その上で、以下を相関係数とする。

$$\frac{\sum_{i=0}^{n'-2} (t'_{i+1} - t'_i)(x'_i - \bar{x'})(y'_i - \bar{y'})}{\sqrt{\sum_{i=0}^{n'-2} (t'_{i+1} - t'_i)(x'_i - \bar{x'})^2}\sqrt{\sum_{i=0}^{n'-2} (t'_{i+1} - t'_i)(y'_i - \bar{y'})^2}}$$

重み付き積分

API `timeVector.integrate(beta=0.0, start=None, end=None)`

時系列観測列を階段関数とみて、次の積分を計算する。

$$\int \exp(-\beta t) v(t) dt$$

ここで、積分区間は `[start, end]` である。また、 β は引数の `beta` で非負とする。`v` は `timeVector` をあらわす。`start` と `end` が `None` の場合は、時系列観測列の時刻の最小値と最大値が使用される

I	整数 I に対し、$M_R(I)$ の行を示す。ただし、$-R \leq I < R$ でない場合は無効な行指定子である。
intVector	整数型観測列 $\{I_k\}$ に対し、$\{M_R(I_k)\}$ が行番号列を示す。ただし、$-R \leq I_j < R$ 以外の要素があった場合は無効な行指定子である。
`[I`$_1$`, I`$_2$`, ⋯, I`$_n$`]`	リストの各要素に対し、$\{M_R(I_k)\}$ が行番号列を示す。ただし、$-R \leq I_j < R$ 以外の要素があった場合は無効な行指定子である。
start:end	$\{k \mid M_R(\text{start}) \leq k < M_R(\text{end})\}$ の行番号列を示す。
start:	$\{k \mid M_R(\text{start}) \leq k < R\}$ の行番号列を示す。
:end	$\{k \mid 0 \leq k < M_R(\text{end})\}$ の行番号列を示す。
start:end:step	$\{k = M_R(\text{start}) + i\,\text{step} \mid M_R(\text{start}) \leq k < M_R(\text{end}), i \geq 0\}$ の行番号列を示す。
:	全ての行を示す。
...	全ての行を示す。
boolVector	ブール型観測列 $\{B_i\}$ に対し、$\{k \mid B_k = T\}$ の行番号列を示す。$\|\{B_i\}\| \neq R$ の場合は無効な行指定子である。
Interval(index, minval, maxval)	列 index の値が minval 以上、maxval 以下の行を示す。ただし、モニターは列 index に関して昇順にソート済みであることを要求する。オプションとして exclude_min=True を与えると、列 index の値が minval より大きい行を示す。また、exclude_max=True を与えると、列 index の値が maxval 未満の行を示す。

図 6.2: 行指定子

第7章　乱数

7.1　種

疑似乱数の系列には、

- グローバル系列
- 個別系列

がある。

グローバル系列の乱数の種は、`setGlobalSeed` で指定する事が出来る。

API　`setGlobalSeed(a)`

`a` が `None` の場合、現在のシステム時間によって種が初期化される。`a` が `None` でない場合、その値によって定められる乱数系列が設定される。同じ種の場合、同じ乱数系列を返す。

個別系列の乱数の種は各乱数生成器で指定する事が出来るが、

- グローバル系列を利用
- 独自系列を利用

を指定する事が出来る。グローバル系列を利用した場合、グローバル系列を用いて各個別乱数を生成する。そのため、複数の並列するプロセスから乱数を生成した場合、呼出し順番によって、個々の乱数系列は変わってしまう。

独自系列を利用した場合、グローバル系列とは独立な、指定した値によって定められる一意な独自系列から乱数が生成される。同一の値を指定した場合同一の系列から乱数が生成される。

7.2　乱数生成器

乱数生成器は全て Python のジェネレータ関数である。`next()` を呼ぶ事で次の乱数を取得する事が出来る。

API `next(乱数生成器)`

指定した乱数生成器の次の乱数を返す。

7.2.1 固定

API `constantValue(val)`

常に同じ値 `val` を返すジェネレータを返す。

7.2.2 指数分布

API `exponentialDistribution(mean = 1.0, seed = None)`

平均 `mean` の指数分布に従う乱数ジェネレータを返す。

確率密度関数

平均を μ とすると、確率密度関数は以下になる。

$$f(x;\mu) = \frac{e^{-\frac{x}{\mu}}}{\mu} \quad (x \geq 0, \mu > 0)$$

7.2.3 正規分布

API `normalDistribution(mean = 1.0, sd = 1.0, seed = None)`

平均`mean`、標準偏差 `sd` の正規分布に従う乱数ジェネレータを返す。

確率密度関数

平均を μ、標準偏差を σ とすると、確率密度関数は以下になる。

$$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad (\sigma > 0)$$

7.2.4 対数正規分布

API `logNormalDistribution(mean = 0.0, sd = 1.0, seed = None)`

平均 `mean`, 標準偏差 `sd` の対数正規分布に従う乱数ジェネレータを返す。

確率密度関数

平均を μ、標準偏差を σ とすると、確率密度関数は以下になる。

$$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-\frac{(\log x-\mu)^2}{2\sigma^2}} \quad (\sigma > 0)$$

7.2.5 一様分布

API `uniformDistribution(a = 0.0, b = 1.0, seed = None)`

`a` から `b` までの一様分布に従う乱数ジェネレータを返す。

確率密度関数

最小を a、最大を b とすると、確率密度関数は以下になる。

$$f(x;a,b) = \frac{1}{b-a} \quad (a \leq x < b)$$

7.2.6 ベータ分布

API `betaDistribution(alpha = 1.0, beta = 1, seed = None)`

`alpha,beta` を形状とする β 分布に従う乱数ジェネレータを返す。

確率密度関数

形状 1 を α、形状 2 を β とすると、確率密度関数は以下になる。

$$f(x;\alpha,\beta) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1} \quad (0 \le x \le 1, \alpha > 0, \beta > 0)$$

7.2.7 ガンマ分布

API `gammaDistribution(alpha = 1.0, beta = 1.0, seed = None)`

形状 `alpha`，尺度 `beta` のガンマ分布に従う乱数ジェネレータを返す。

確率密度関数

形状を α、尺度を β とすると、確率密度関数は以下になる。

$$f(x;\alpha,\beta) = x^{\alpha-1} \frac{e^{-x/\beta}}{\Gamma(\alpha)\,\beta^{\alpha}} \quad (x \ge 0, \alpha > 0, \beta > 0)$$

7.2.8 アーラン分布

API `gammaDistribution(alpha = 1, beta = 1.0, seed = None)`

形状 `alpha`，尺度 `beta` のアーラン分布に従う乱数ジェネレータを返す。

確率密度関数

形状を α、尺度を β とすると、確率密度関数は以下になる。

$$f(x;\alpha,\beta) = x^{\alpha-1} \frac{e^{-x/\beta}}{(\alpha-1)!\,\beta^{\alpha}} \quad (x \ge 0, \alpha > 0, \alpha \in \mathbf{N}, \beta > 0)$$

7.2.9　パレート分布

API `paretoDistribution(a = 1.0, b = 1.0, seed = None)`

形状 `a`，最小値 `b` のパレート分布に従う乱数ジェネレータを返す。

確率密度関数

形状を a、最小値を b とすると、確率密度関数は以下になる。

$$f(x;a,b) = \frac{ab^a}{x^{a+1}} \quad (x \geq b, a > 0, b > 0)$$

7.2.10　ワイブル分布

API `weibullDistribution(alpha = 1.0, beta = 1.0, seed = None)`

形状 `alpha`，尺度 `beta` のワイブル分布に従う乱数ジェネレータを返す。

確率密度関数

形状を α、尺度を β とすると、確率密度関数は以下になる。

$$f(x;\alpha,\beta) = \frac{\alpha}{\beta}\left(\frac{x}{\beta}\right)^{\alpha-1} e^{-\left(\frac{x}{\beta}\right)^{\alpha}} \quad (x \geq 0, \alpha > 0, \beta > 0)$$

7.2.11　カイ二乗分布

API `chiSquareDistribution(df = 1, seed = None)`

自由度 `df` のカイ二乗分布に従う乱数ジェネレータを返す。

確率密度関数

自由度を m とすると、確率密度関数は以下になる。

$$f(x;m) = \frac{(1/2)^{m/2}}{\Gamma(m/2)} x^{m/2-1} e^{-x/2} \quad (m > 0)$$

7.2.12 F 分布

API `fDistribution(dfnum = 1, defden = 1, seed = None)`

分子の自由度 `dfnum`、分母の自由度 `dfden` の F 分布に従う乱数ジェネレータを返す。

確率密度関数

分子の自由度を m_1、分母の自由度を m_2 とすると、確率密度関数は以下になる。

$$f(x; m_1, m_2) = \frac{\Gamma(\frac{m_1+m_2}{2})(\frac{m_1}{m_2})^{\frac{m_1}{2}} x^{\frac{m_1}{2}-1}}{\Gamma(\frac{m_1}{2})\Gamma(\frac{m_2}{2})(1+\frac{m_1}{m_2}x)^{\frac{m_1+m_2}{2}}} \quad (m_1 > 0, m_2 > 0)$$

7.2.13 ロジスティック分布

API `logisticDistribution(loc = 0.0, scale = 1.0, seed = None)`

位置 `loc`、尺度 `scale` のロジスティック分布に従う乱数ジェネレータを返す。

確率密度関数

位置を μ、尺度を s とすると、確率密度関数は以下になる。

$$f(x; \mu, s) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2}$$

7.2.14 非心カイ二乗分布

API `noncentralChiSquareDistribution(df = 2, nonc = 1.0, seed = None)`

自由度 `df`、非心度 `nonc` の非心カイ二乗分布に従う乱数ジェネレータを返す。

確率密度関数

自由度を m、非心度を δ とすると、確率密度関数は以下になる。

$$f(x;m,\delta) = \frac{e^{-\frac{x+\delta}{2}}}{2^{\frac{m}{2}}} \sum_{j=0}^{\infty} (\frac{\delta}{4})j \frac{x^{\frac{m}{2}+j-1}}{j!\Gamma(\frac{m}{2}+j)} \quad (m > 0, \delta \geq 0)$$

7.2.15 非心 F 分布

API `noncentralFDistribution(dfnum = 2, dfden = 2, nonc = 1.0, seed = None)`

分子の自由度 `dfnum`、分母の自由度 `dfden`、非心度 `nonc` の非心 F 分布に従う乱数ジェネレータを返す。

確率密度関数

分子の自由度を m_1、分母の自由度を m_2、非心度を δ とすると、確率密度関数は以下になる。

$$f(x;m_1,m_2,\delta) = \sum_{j=0}^{\infty} \frac{e^{-\frac{\delta}{2}}(\frac{\delta}{2})^j}{j!} \frac{(m_1+2j)^{\frac{m_1+2j}{2}} m_2^{\frac{m_2}{2}}}{B(\frac{m_1+2j}{2},\frac{m_2}{2})} x^{\frac{m_1+2j}{2}-1} (m_2+(m_1+2j)x)^{-\frac{(m_1+2j)+m_2}{2}}$$

7.2.16 コーシー分布

API `cauchyDistribution(seed = None)`

標準コーシー分布に従う乱数ジェネレータを返す。

確率密度関数

確率密度関数は以下になる。

$$f(x) = \frac{1}{\pi(1+x^2)}$$

7.2.17 レイリー分布

API `rayleighDistribution(scale = 1.0, seed = None)`

尺度 `scale` のレイリー分布に従う乱数ジェネレータを返す。

確率密度関数

尺度を θ とすると、確率密度関数は以下になる。

$$f(x;\theta) = \frac{x}{\theta^2} e^{-\frac{x^2}{2\theta^2}}$$

7.2.18 t 分布

API `tDistribution(df = 2, seed = None)`

自由度 `df` の `t` 分布に従う乱数ジェネレータを返す。

確率密度関数

自由度を m とすると、確率密度関数は以下になる。

$$f(x;m) = \frac{\Gamma((m+1)/2)}{\sqrt{m\pi}\,\Gamma(m/2)}(1+x^2/m)^{-(m+1)/2}$$

7.2.19 三角分布

API `triangularDistribution(left = 0.0, mode = 0.5, right = 1.0, seed = None)`

最小値 `left`、最頻値 `mode`、最大値 `right` の三角分布に従う乱数ジェネレータを返す。

確率密度関数

最小値を α、最頻値を γ、最大値を β とすると、確率密度関数は以下になる。

$$f(x;\alpha,\gamma,\beta) = \begin{cases} \frac{2(x-\alpha)}{(\beta-\alpha)(\gamma-\alpha)} & (\alpha \le x \le \gamma) \\ \frac{2(\beta-x)}{(\beta-\alpha)(\beta-\gamma)} & (\gamma \le x \le \beta) \end{cases}$$

7.2.20 二項分布

API `binomialDistribution(n = 1.0, p = 0.1, seed = None)`

試行回数 `n`，確率 `p` の二項分布に従う乱数ジェネレータを返す。

確率関数

試行回数を n、確率を p とすると、確率関数は以下になる。

$$f(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} \quad (n > 0, 0 \leq p \leq 1)$$

7.2.21 幾何分布

API `geometricDistribution(p = 1, seed = None)`

確率 `p` の幾何分布に従う乱数ジェネレータを返す。

確率関数

確率を p とすると、確率関数は以下になる。

$$f(k; p) = (1-p)^{k-1} p \quad (0 \leq \leq 1)$$

7.2.22 超幾何分布

API `hyperGeometricDistribution(ngood = 1, nbad = 1, nsample = 2, seed = None)`

ある属性を持つ要素数 `ngood`、それ以外の要素数 `nbad`、サンプル数 `nsample` の超幾何分布に従う乱数ジェネレータを返す。

確率関数

ある属性を持つ要素数 m、それ以外の要素数を n、サンプル数を N とすると、確率関数は以下になる。

$$f(x;n,m,N) = \frac{\begin{pmatrix} m \\ x \end{pmatrix}\begin{pmatrix} n \\ N-x \end{pmatrix}}{\begin{pmatrix} m+n \\ N \end{pmatrix}} \quad (0 \leq x \leq m, n+m-N \leq x \leq n)$$

7.2.23 負の二項分布

API `negativeBinomialDistribution(n = 1, p = 0.5, seed = None)`

試行回数 `n`，確率 `p` の負の二項分布に従う乱数ジェネレータを返す。

確率関数

試行回数を n、確率を p とすると、確率関数は以下になる。

$$f(x;n,p) = \begin{pmatrix} n+x-1 \\ n-1 \end{pmatrix} p^n(1-p)^x \quad (n > 0, 0 \leq p \leq 1)$$

7.2.24 ポアソン分布

API `poissonDistribution(lmd = 1.0, seed = None)`

平均 `lmd` のポアソン分布に従う乱数ジェネレータを返す。

確率関数

平均を λ とすると、確率関数は以下になる。

$$f(x;\lambda) = \frac{\lambda^x e^{-\lambda}}{x!}$$

7.2.25 経験分布

API `empiricalDistribution(weights = [[`w_1`, `v_1`],` `[`w_2`,`v_2`], ...])`

確率 $w_i/\sum w_i$ で v_i を返すような乱数ジェネレータを返す。

確率関数

重み列を $\{w_i\}$、値列を $\{v_i\}$ とすると、確率関数は以下になります。

$$f(v_i;\{w_j\}) = \frac{w_i}{\sum_j w_j} \quad (w_i \geq 0)$$

7.2.26 密度分布

API `densityDistribution(weights = [[`w_1`, `v_1`],` `[`w_2`,`v_2`], ...],kernel = u' ガ ウ ス 関 数', width = 1.0 , seed = None)`

確率 $w_i/\sum w_i$ で v_i を中心とした `width` をもつ `kernel` 関数に従う乱数ジェネレータを返す。

確率関数

重み列を $\{w_i\}$、値列を $\{v_i\}$ とすると、確率関数は以下になります。カーネル関数を K とすると、カーネル密度推定では以下の式により密度を推定します。

$$f(x) = \sum_{i=1}^{N} \frac{w_i}{\sum_j w_j} K(\frac{x - v_i}{\omega})$$

ここで ω は間隔幅（`width`）です。

カーネル関数は、それぞれ以下で定義されます。

- ガウス関数

$$f(x;\mu,\omega) = \frac{1}{\sqrt{2\pi}\omega} e^{-\frac{(x-\mu)^2}{2\omega^2}} \qquad (\omega > 0)$$

- 三角関数

$$f(x;\mu,\omega) = \frac{1}{\omega^2}(x-\mu) + \frac{1}{\omega} \qquad (\omega > 0, \mu - \omega \leq x \leq \mu + \omega)$$

- 矩形関数

$$f(x;\mu,\omega) = \frac{1}{2\omega} \qquad (\omega > 0, \mu - \omega \le x \le \mu + \omega)$$

- コサイン関数

$$f(x;\mu,\omega) = \frac{1}{2\omega}[1 + \cos(\frac{\pi(x-\mu)}{\omega})] \qquad (\omega > 0, \mu - \omega \le x \le \mu + \omega)$$

7.2.27 サンプリング

API `sample(vec = [`v_1`, `v_2`, ...], seed = None)`

均一な確率で v_i を返すような乱数ジェネレータを返す。

7.2.28 再生

API `replay(vec = [v1, v2, ..], repeat = Ture)`

vec の順番で v_i を返すようなジェネレータを返す。repeat が True なら繰り返す。

7.2.29 ステップ

API `replayStep(begin = 1, end = 2, step = 1, repeat =False)`

begin から end 未満までを step おきに返すようなジェネレータを返す。repeat が True なら繰り返す。

7.2.30 変化率ステップ

API `replayStepByPercentage(baseValue=1, maxPer=100, minPer=-100, step=50, typeFun=float, repeat=False)`

次の値を返すジェネレータを返す：$baseValue(1 + \frac{minPer}{100}), baseValue(1 + \frac{step+minPer}{100}), baseValue(1 + \frac{2step+minPer}{100}), \ldots, baseValue(1 + \frac{maxPer}{100})$。ただし、各値は typeFun で指定した関数で変換されて返される。repeat が True なら繰り返す。

第8章　強化学習

8.1　概要

S^4 Simulation System で強化学習を利用する際は、強化学習モデルを定義する必要がある。強化学習モデルは強化学習に関する設定をまとめたものである。

8.2　強化学習モデル

強化学習モデルは具体的には `RLModel` クラスを継承したクラスとして定義される。`RLModel` クラスを継承した上で、次のクラス変数とメソッドを定義する必要がある。

API　`OBS_SPEC`

観測値集合を定めるクラス変数である。観測値集合は整数・実数値の閉区間または文字列の有限集合の、積集合としてあらわされるものとする。`OBS_SPEC` は、観測値集合の各次元をあらわすタプルのリストである。タプルは次の 3 つの要素を持つ。

1. 次元名である。
2. 型を表す文字列であり、次のいずれかである。実数は `"f"` であり、整数は `"i"` であり、文字列は `"o"` である。
3. 取りうる値をあらわすリスト・タプルである。第二要素が `"f"`, `"i"` の場合は最小値と最大値のタプルで、`"o"` の場合は取りうる文字列のリストとする。なお、与えられる文字列はアルファベット文字からなる文字列のみとする。

API ACT_SPEC

観測値集合を定めるクラス変数である。形式は OBS_SPEC と同様である。

API SMDP

semi-Markov Decision Process でモデル化するかどうかをあらわすブール値のクラス変数である。真ならば、semi-Markov Decision Process でモデル化し、学習することを意味する。偽ならば、通常の Markov Decision Process でモデル化される。

API observation(self, simulator, item)

観測値を計算するメソッドである。各引数の意味は次の通りである。

simulator

シミュレータ本体であり、 simulator を通して現在のシミュレーション状況を観測し、観測値を計算する。

item

後述する getAction における item が渡される。フローアイテムのリンク選択で selectRL (5.3.6) を使用した場合は、対象のフローアイテムが item に格納される。

返り値として、観測値集合の各次元の名前がキーであり、値が次元の値である辞書を返すとする。

API `reward(self, simulator, item, prev_time)`

即時報酬値を計算するメソッドである。各引数の意味は次の通りである。

`simulator`

シミュレータ本体であり、 simulator を通して、現在のシミュレーション状況を観測し、即時報酬値を計算する。

`item`

後述する getAction における item が渡される。フローアイテムのリンク選択で selectRL (5.3.6) を使用した場合は、対象のフローアイテムが item に格納される。

`prev_time`

即時報酬計算のタイミングを次回の行動決定時点とする際 (後述する getAction における async_reward が False) には、直前の行動を決定してから今回の行動を決定するまでの経過時間である。このタイミングをずらす際 (getAction における async_reward が True) には、行動を決定してから即時報酬を確定させるまでの経過時間となる。

返り値は即時報酬値である。

API action(self, simulator, obs, act, item)

強化学習によって提案された行動に問題がある場合に、このメソッドで修正する。各引数の意味は次の通りである。

simulator

シミュレータ本体である。

obs

観測値である。

act

強化学習によって提案された行動値である。

item

後述する getAction における item が渡される。フローアイテムのリンク選択で selectRL（5.3.6）を使用した場合は、対象のフローアイテムが item に格納される。

引数の act があらわす行動に問題がある場合は、act の各次元の値を修正した上で、 act を返すとする。問題が無ければ act をそのまま返すとする。

API __init__

__init__ メソッドでは次を行う。

1. RLModel.__init__(self, simulator) を呼び出し
2. 強化学習の学習対象（行動価値関数など）の作成

前者は親クラスの __init__ の呼び出しである。引数の simulator はシミュレータ本体である。後者は行動価値関数などの学習対象の作成や、更新手法の作成を行うことに対応する。メンバ変数として次の名称のオブジェクトを定義する必要がある。オブジェクトの具体的な作成の仕方は 8.3 章 を参照のこと。

vector

状態価値関数をあらわす。状態価値関数を使用する手法の場合にのみ定義する必要がある。シミュレーション終了後に学習結果として保存される。状態価値関数の作成方法については 8.3.3 章 を参照のこと。

table

行動価値関数をあらわす。行動価値関数を使用する手法の場合にのみ定義する必要がある。シミュレーション終了後に学習結果として保存される。行動価値関数の作成方法については 8.3.4 章 を参照のこと。

preference

Actor Critic における preference をあらわす。Actor Critic 法の場合にのみ定義する必要がある。シミュレーション終了後に学習結果として保存される。preference の作成方法については 8.3.4 章 を参照のこと。

numerator

Monte Calro 法における numerator をあらわす。Monte Carlo 法の場合にのみ定義する必要がある。シミュレーション終了後に学習結果として保存される。numerator の作成方法については 8.3.4 章 を参照のこと。

denominator

Monte Carlo 法における denominator をあらわす。Monte Carlo 法の場合にのみ定義する必要がある。シミュレーション終了後に学習結果として保存される。denominator の作成方法については 8.3.4 章 を参照のこと。

critic

Actor Critic における critic をあらわす。Actor Critic 法の場合にのみ定義する必要がある。critic の作成方法については 8.3.6 章 を参照のこと。

control

方策・Actor をあらわす。方策・Actor の作成方法については 8.3.5 章 を参照のこと。

make_updater

学習対象の更新手法をあらわすオブジェクトを返す関数である。返すのは 8.3.7 章 のオブジェクトとする。

RLModel.__init__ の実行後には、以下のメンバ変数が定義される。

API obsset

観測値集合をあらわす。

API actset

行動値集合をあらわす。

定義された強化学習モデル (RLModel クラスを継承したクラス) は通常は直接使用せず、 RLModelSet というクラスを通して利用する。RLModelSet は次のメソッドを持つ。

API RLModelSet(simulator, cls, name, save_history=True, save_model=True)

RLModelSet を作成する。各引数の意味は次の通り。

simulator

シミュレータ本体である。

cls

強化学習モデルをあらわす RLModel クラスを継承したクラスである。

name

強化学習モデルの名前である。後述する履歴やモデルの出力の際に、ファイル名の一部として使用される。

save_hisotry

真ならば、シミュレーション終了後に強化学習の観測値・行動値などの履歴をまとめたモニタを出力する。モニタは「name - 履歴」という名前で出力される。ここで、name は引数で与える name である。偽ならば出力されない。

save_model

真ならば、シミュレーション終了後に強化学習の学習結果を出力する。学習結果は「name - 学習モデル」という名前で出力される。ここで、name は引数で与える name である。偽ならば出力されない。

API RLModelSet.getAction(model_id, episode_id, item, async_reward = False)

強化学習モデルを用いて行動を取得する。各引数の意味は次の通り。

model_id

行動を提示するモデルを表す文字列である。以前に同一の model_id で getAction を呼び出していれば、その際と同一のモデルが使用される。以前に同一の model_id で getAction が呼び出されていなければ RLModelSet の作成時に与えられた cls を元に強化学習モデルが作成され、そのモデルを使用して行動が提示される。

episode_id

強化学習におけるエピソード ID をあらわす文字列である。

item

RLModel クラスの observation 、action 、 reward メソッドの引数に item として渡される。

async_reward

偽ならば、返り値の行動に対する即時報酬値を次回の行動の取得時点で確定する。真ならば、報酬決定関数 (8.3.9) の呼び出し時点で確定する。

即時報酬計算のタイミングを、次回の行動の取得時点からずらさない際には、返り値は行動である。行動の各次元の値には 行動 [' 次元名'] でアクセスできる。ずらす際には、返り値は行動と報酬決定関数 (8.3.9) のタプルである。なお、getAction を呼び出した後で、必ず setActualAction も呼び出すこと。

API `RLModelSet.setActualAction(model_id, episode_id, act)`

getAction で行動を取得した後で、実際に取られた行動を強化学習モデルに通知する。各引数の意味は次の通り。

model_id

行動を提示したモデルを表す文字列である。引数の act を取得した際の getAction と同一の文字列を与えること。

episode_id

強化学習におけるエピソード ID をあらわす文字列である。引数の act を取得した際の getAction と同一の文字列を与えること。

act

実際に取った行動に対応する行動値である。

返り値は無い。

8.3 強化学習関連のオブジェクト

強化学習関連の各種オブジェクトの作成方法や、オブジェクトの持つメソッドについて順に説明する。

8.3.1 観測値・行動値集合

観測値・行動値集合は、整数・実数値の閉区間またはアルファベット文字列の有限集合の積集合として構成される。

整数・実数の閉区間と、アルファベット文字列の有限集合は次で作成されるオブジェクトであらわされる。

API `rein.IntegerDimension(name, minvalue, maxvalue)`

整数値の閉区間をあらわす。引数の意味は次の通りである。

`name`

次元の名前

`minvalue`

取りうる値の下限

`maxvalue`

取りうる値の上限

API `rein.RealDimension(name, minvalue, maxvalue)`

実数値の閉区間をあらわす。引数の意味は次の通りである。

`name`

次元の名前

`minvalue`

取りうる値の下限

`maxvalue`

取りうる値の上限

API `rein.NominalDimension(name, values)`

アルファベット文字列値をとる集合をあらわす。引数の意味は次の通りである。

`name`

次元の名前

`values`

取りうる値（文字列）のリスト

上記で作成したオブジェクトを `*` 演算子により結合したものが観測値・行動値集合となる。なお、観測値・行動値集合が 1 次元の場合は、以下を用いて作成する。

API `rein.ValueSet(dim)`

`dim` があらわす次元からなる観測値・行動値集合を作成する。

8.3.2 観測値・行動値

観測値、行動値は次で作成される。

API `rein.Value(valueset)`

`valuset` が表す集合の元をあらわす。観測値や行動値に対応する。

観測値、行動値の値は次で得る。

API `value['次元名']`

次元名に対応する値が返される。

観測値、行動値の値は次で設定する。

API `value['次元名'] = 値`

次元名に対応する次元に値が設定される。

8.3.3 状態価値関数

状態価値関数の作成方法やメソッドについて、状態価値関数の種別毎に順に説明する。また、種別に関わらず備わっている共通の機能について本説の最後に述べる。

近似表現を使用しない状態価値関数

近似表現を使用しない状態価値関数は次の関数により作成できる。

API `rein.NonApproximatedVector(valueset, initial_value)`

引数の意味は次の通りである。

`valueset`

状態価値関数が定義される観測値集合を表す。実数値をとる次元の無い集合を与えること。

`initial_value`

初期値である。全ての値はこれで初期化される。

近似表現を使用しない場合にのみ、次の関数で明示的に値を設定できる。

API `rein.NonApproximatedVector.setAt(index, value)`

指定した観測値に価値を設定する。

`index`

観測値

`value`

設定する値

NeuralNet を使用する場合

近似関数として `Neural Net` を使用した状態価値関数は次の関数により作成できる。

API `rein.NeuralNetVector(valueset, hidden_layer_sizes, min_weight, max_weight)`

引数の意味は次の通りである。

`valueset`

状態価値関数が定義される観測値集合をあらわす。

`hidden_layer_sizes`

隠れ層のニューロン数のリストである。

`min_weight`

重みの初期値の最小値である

`max_weight`

重みの初期値の最大値である

重みは最小値から最大値までの範囲でランダムに設定される。

Hashtilng を使用する場合

近似関数として `Hashtiling` を使用した状態価値関数は次の関数により作成できる。

API `rein.LinearFunctionVector(rein.HashTilingFeatureVector(num_tiling, memory_size), normalize_type, has_bias`

引数の意味は次の通りである。

`num_tiling`

1 度の計算でアクティブになるタイルの個数である。

`memory_size`

タイルの総数である。

`normalize_type`

パラメータベクトルを正規化する方法をあらわす。8.3.10 章 の定数が指定できる。

`has_bias`

真ならばバイアス項を持ち。偽ならば持たないことになる。

共通の機能

いずれの形式の状態価値関数でも価値は次で得られる。

API `vector[obs]`

`vector` は状態価値関数で、 `obs` は観測値である。観測値に対応する価値が返される。

8.3.4 行動価値関数, preference, numerator, denominator

行動価値関数や `preference`, `numerator`, `denominator` の作成方法やメソッドについて、種別毎に順に説明する。また、種別に関わらず備わっている共通の機能について本説の最後に述べる。

近似表現を使用しない場合

近似表現を使用しない場合は次の関数により作成できる。

API `rein.NonApproximatedTable(row_valueset, column_valueset, initial_value)`

引数の意味は次の通りである。

`row_valueset`

定義される観測値集合を表す。実数値をとる次元の無い集合を与えること。

`column_valueset`

定義される行動値集合を表す。実数値をとる次元の無い集合を与えること。

`initial_value`

初期値である。全ての値はこれで初期化される。

近似表現を使用しない場合にのみ、次の関数で明示的に値を設定できる。

API `rein.NonApproximatedTable.setAt(row, column, value)`

指定した観測値・行動値に対応する値を設定する。

`row`

観測値

`column`

行動値

`value`

設定する値

NeuralNet を使用する場合

近似関数として `Neural Net` を使用する場合は次の関数により作成できる。

API rein.NeuralNetTable(row_valueset, column_valueset, hidden_layer_sizes, min_weight, max_weight)

引数の意味は次の通りである。

row_valueset

定義される観測値集合を表す。

column_valueset

定義される行動値集合を表す。

hidden_layer_sizes

隠れ層のニューロン数のリストである。

min_weight

重みの初期値の最小値である

max_weight

重みの初期値の最大値である

重みは最小値から最大値までの範囲でランダムに設定される。

Hashtilng を使用する場合

近似関数として Hashtiling を使用した状態価値関数は次の関数により作成できる。

API `rein.LinearFunctionTable(row_valueset, column_valueset, rein.HashTilingFeatureVector2D(num_tiling, memory_size), normalize_type, has_bias`

引数の意味は次の通りである。

`row_valueset`

定義される観測値集合を表す。実数値をとる次元の無い集合を与えること。

`column_valueset`

定義される行動値集合を表す。実数値をとる次元の無い集合を与えること。

`num_tiling`

1 度の計算でアクティブになるタイルの個数である。

`memory_size`

タイルの総数である。

`normalize_type`

パラメータベクトルを正規化する方法をあらわす。8.3.10 章 の定数が指定できる。

`has_bias`

真ならばバイアス項を持ち。偽ならば持たないことになる。

共通の機能

いずれの形式でも値は次で得られる。

API `table[obs, act]`

table は行動価値関数（または preference, numerator, denominator）で、obs は観測値、act は行動値である。対応する値が返される。

8.3.5 方策・Actor

方策・Actor の作成方法やメソッドについて、種別毎に順に説明する。また、種別に関わらず備わっている共通の機能について本節の最後に述べる。

Epsilon-Greedy

ε-greedy な方策は次の関数により作成できる。

API `rein.control.EpsilonGreedy(table, epsilon)`

引数の意味は次の通りである。

table

観測値と行動から、その行動の好ましさが定まるテーブルである。必ずしも行動価値関数である必要はない。

8.3.4 章 で作成されるオブジェクトが指定できる。

epsilon

ε である。0 以上 1 以下の実数とする。確率 ε で最適な行動を選択し、 $1-\varepsilon$ でランダムに行動を選択する。

生成後に、 ε の値は次のプロパティから設定・変更できる。

API `control.epsilon`

control は生成された ε-greedy な方策である。

Softmax

Softmax 方策は次の関数により作成できる。

API `rein.control.Softmax(table, temperature)`

引数の意味は次の通りである。

`table`

観測値と行動から、その行動の好ましさが定まるテーブルである。必ずしも行動価値関数である必要はない。
8.3.4 章 で作成されるオブジェクトが指定できる。

`temperature`

温度である。正の実数とする。
観測値を o 、行動を a とすると、各行動は $\exp(table[o,a]/temperature)$ の比率で確率的に選択される。

生成後に、 温度 `temperature` の値は次のプロパティから設定・変更できる。

API `control.temperature`

`control` は生成された `Softmax` な方策である。

Neural Net を使用したガウス型の方策

`Neural Net` を使用したガウス型の方策とは平均ベクトルがニューラルネットであるガウス分布による方策である。観測値 s における行動値 a の選択確率がガウス分布 $\pi(s,a) = \frac{1}{Z}exp((a-\mu(s))^T\Sigma^{-1}(a-\mu(s))$ により定まる。ただし、 平均 $\mu(s)$ は（フィードフォワード）ニューラルネットにより定まる。
次の関数により作成できる。

API `rein.control.GaussianWithNeuralNet(observation_set, action_set, hidden_layer_sizes, min_weight, max_weight, sigma)`

引数の意味は次の通りである。

`observation_valueset`

定義される観測値集合を表す。

`action_valueset`

定義される行動値集合を表す。

`hidden_layer_sizes`

隠れ層のニューロン数のリストである。

`min_weight`

重みの初期値の最小値である

`max_weight`

重みの初期値の最大値である

`sigma`

ガウス分布の分散共分散行列を表す。スカラー値を指定すると、$\sigma^2 I$ が初期値として指定される。スカラー値のリストを指定するとリストの成分を $\sigma_1, \ldots, \sigma_n$ として、対角成分が $\sigma_1^2, \ldots, \sigma_n^2)$ である対角行列が初期値として指定される。リストのリストを指定するとそのまま初期分散共分散行列として指定される。

重みは最小値から最大値までの範囲でランダムに設定される。

Hashtiling を使用したガウス型の方策

Hashtiling を使用したガウス型の方策は次の関数により作成できる。

API rein.control.GaussianWithFeatureVector(observation_set, action_set, rein.HashTilingFeatureVector2D(num_tiling, memory_size), sigma)

引数の意味は次の通りである。

observation_valueset

定義される観測値集合を表す。

action_valueset

定義される行動値集合を表す。

num_tiling

1 度の計算でアクティブになるタイルの個数である。

memory_size

タイルの総数である。

sigma

ガウス分布の分散共分散行列を表す。スカラー値を指定すると、$\sigma^2 I$ が初期値として指定される。スカラー値のリストを指定するとリストの成分を $\sigma_1, \ldots, \sigma_n$ として、対角成分が $\sigma_1^2, \ldots, \sigma_n^2)$ である対角行列が初期値として指定される。リストのリストを指定するとそのまま初期分散共分散行列として指定される。

共通の機能

いずれの形式でも、行動の選択確率が次で得られる。

API `control.getProbability(observation, action)`

`control` は方策である。引数の意味は次の通りである。

`observation`

観測値である。

`action`

行動値である。

返り値として `observation` を観測した時に、行動 `action` を取る確率密度または質量が返される。行動集合が連続的である場合は確率密度であり、有限離散集合である場合は確率質量となる。

8.3.6 Critic

Critic の作成方法やメソッドについて、種別毎に順に説明する。また、種別に関わらず備わっている共通の機能について本説の最後に述べる。

TD(λ)

`TD(λ)` による `Critic` は次で作成できる。

API `rein.TDLambdaCritic(statevalue, discount_rate etrace, step_size)`

引数の意味は次の通りである。

`statevalue`

状態価値関数である

`discount_rate`

即時報酬の割引率である

`etrace`

トレース (8.3.8 章) である

`step_size`

状態価値の更新に関する学習率である

Average Reward TD(λ)

Average Reward TD(λ) による Critic は次で作成できる。

API `rein.AverageRewardTDLambdaCritic(statevalue, etrace, step_size, average_reward_step_size)`

引数の意味は次の通りである。

`statevalue`

状態価値関数である

`etrace`

トレース (8.3.8 章) である

`step_size`

状態価値の更新に関する学習率である

`average_reward_step_size`

即時報酬の時間平均の推定に関する学習率である

また、Average Reward TD(λ) による Critic は次のプロパティを持つ。

API `critic.average_reward_step_size`

即時報酬の時間平均の推定に関する学習率をあらわす。参照・設定の両方が可能である。

API `critic.average_reward`

即時報酬の時間平均の推定値をあらわす。参照・設定の両方が可能である。

共通の機能

いずれの形式でも、以下のプロパティを持つ。

API `critic.step_size`

状態価値の更新に関する学習率である。参照・設定の両方が可能である。

8.3.7 更新手法

各種更新手法を表すオブジェクトの作成方法やメソッドについて、種別ごとに説明する。また、種別にかかわらず備わっている共通の機能について本節の最後に述べる。

Q 学習

Q 学習をあらわすオブジェクトは次の関数により作成できる。

API `rein.QUpdater(value_function, discount_rate, step_size)`

引数の意味は次の通りである。

`value_function`

行動価値関数

`discount_rate`

即時報酬の割引率。0 以上 1 以下の実数値。

`step_size`

学習率。0 以上 1 以下の実数値。

Q 学習をあらわすオブジェクトは次のプロパティを持つ。

API `updater.step_size`

学習率をあらわす。参照・設定の両方が可能である。

TD(λ) (Sarsa(λ))

TD(λ)（ または Sarsa(λ) ）をあらわすオブジェクトは次の関数により作成できる。

API `rein.SarsaLambdaUpdater(value_function, discount_rate, etrace, step_size)`

引数の意味は次の通りである。

`value_function`

行動価値関数

`discount_rate`

即時報酬の割引率。0 以上 1 以下の実数値。

`etrace`

トレース (8.3.8 章) である

`step_size`

学習率。0 以上 1 以下の実数値。

TD(λ)（または Sarsa(λ)）をあらわすオブジェクトは次のプロパティを持つ。

API `updater.step_size`

学習率をあらわす。参照・設定の両方が可能である。

Monte Carlo 法

Monte Carlo 法をあらわすオブジェクトは次の関数により作成できる。

API `rein.OffPolicyMonteCarloUpdater(value_function, discount_rate, every_visit, evaluate_policy, control_policy, numerator, denominator)`

引数の意味は次の通りである。

`value_function`

行動価値関数

`discount_rate`

即時報酬の割引率である。 0 以上 1 以下の実数値とする。

`every_visit`

真偽値をとる。真ならば every-visit Monte Carlo になり、偽なら first-visit Monte Carlo により更新される。

`evaluate_policy`

評価対象の方策である。省略することもできる。省略された場合は行動価値関数から定まる greedy な方策が与えられたとみなす。

`control_policy`

エピソードを生成している方策である。省略することも出来る。省略された場合は evaluate_policy と control_policy が一致すると見なされる。

`numerator`

numerator である。指定しなくとも良い。指定しない場合は全成分が 0 で初期化される。

`denominator`

denominator である。指定しなくとも良い。指定しない場合は全成分が 0 で初期化される。

Monte Carlo 法をあらわすオブジェクトは次のプロパティを持つ。

API `updater.numerator`

`numerator` をあらわす。取得のみが可能である。

API `updater.denominator`

`denominator` をあらわす。取得のみが可能である。

GQ(λ)

GQ(λ) 法をあらわすオブジェクトは次の関数によりさくせいできる。

API `rein.GQLambdaUpdater(value_function, discount_rate, etrace, step_size, relative_step_size, evaluate_policy, control_policy)`

引数の意味は次の通りである。

`value_function`

行動価値関数

`discount_rate`

即時報酬の割引率である。 0 以上 1 以下の実数値とする。

`etrace`

トレースである

`step_size`

学習率である。0 以上 1 以下の実数値とする。

`relative_step_size`

relative step size である。 0 以上 1 以下の実数値とする。

`evaluate_policy`

評価対象の方策である。省略することもできる。省略された場合は行動価値関数から定まる greedy な方策が与えられたとみなす。

`control_policy`

行動を決定している方策である。省略することも出来る。省略された場合は `evaluate_policy` と `control_policy` が一致すると見なされる。

GQ(λ) をあらわすオブジェクトは次のプロパティを持つ。

API `updater.step_size`

学習率をあらわす。参照・設定の両方が可能である。

API `updater.relative_step_size`

relative step size をあらわす。参照・設定の両方が可能である。

Actor Critic

Actor Critic 法をあらわすオブジェクトは次の関数により作成できる。

API `rein.ActorCriticUpdater(actor, critic, step_size)`

引数の意味は次の通りである。

actor

actor（8.3.5 章）である。

critic

critic（8.3.6 章）である。

step_size

actor の学習率である

Actor Critic をあらわすオブジェクトは次のプロパティを持つ。

API `updater.step_size`

actor の学習率をあらわす。参照・設定の両方が可能である。

Natural Actor Critic

Natural Actor Critic 法をあらわすオブジェクトは次の関数により作成できる。

API rein.NaturalActorCritic(actor, critic, step_size, advantage_step_size)

actor

actor（8.3.5 章）である。

critic

critic（8.3.6 章）である。

step_size

actor の学習率である

advantage_step_size

アドバンテージパラメータの学習率である

Natural Actor Critic をあらわすオブジェクトは次のプロパティを持つ。

API updater.step_size

actor の学習率をあらわす。参照・設定の両方が可能である。

API updater.advantage_step_size

アドバンテージパラメータの学習率をあらわす。参照・設定の両方が可能である。

なお、具体的なアルゴリズムの形については [1] の Algorithm 3 を参照のこと。

8.3.8 トレース

Eligibility Trace (Accumulate Trace)

Eligibility Trace（ または Accumulate Trace）は次で作成できる。

API rein.EligibilityTrace(decay_rate)

引数の意味は次の通りである。

decay_rate

トレースの減衰率。0 以上 1 以下の実数値。

Replacing Trace

Replacing Trace は次で作成できる。

API rein.ReplacingTrace(decay_rate)

引数の意味は次の通りである。

decay_rate

トレースの減衰率。0 以上 1 以下の実数値。

8.3.9 報酬決定関数

報酬決定関数は、前述の getAction を引数 async_reward を True として呼び出した際に作成される。同時に得られた行動と紐づいており、呼び出すことで行動に対する即時報酬値を確定し、学習モデルに通知する。

API 報酬決定関数 ()

即時報酬の計算を行い、得られた値を強化学習モデルへ通知する。ただし、既に通知済みの場合はエラーとなる。

API 報酬決定関数.isCalled()

報酬決定関数が既に即時報酬値を強化学習モデルへ通知したかどうかの真偽値を取得する。
返り値として、真偽値が返される。

8.3.10 その他

ベクトルの正規化方法

いくつかの手法でベクトルの正規化手法を選択でき、各種法をあらわす定数が次のように定められている。

API `rein.NormType.NoNormalize`

正規化しない

API `rein.NormType.L1Norm`

L1 ノルムで正規化する

API `rein.NormType.L2Norm`

L2 ノルムで正規化する

8.4 出力されるモデルの利用方法

`RLModelSet` を `save_model = True` で作成した場合、学習結果のモデルが出力フォルダに出力される。出力されたモデルは以下の `RLModelDir` クラスを通して利用する。

API RLModelDir(name, basedir, id=None)

出力されたモデルのフォルダからモデルを読み込む。各引数の意味は次の通り。

name

モデルのフォルダ名である。ただし「.mon」は除く。

basedir

モデルが格納されているフォルダへのパスである。

id

モデルを区別する id である。

RLModelDir には以下のプロパティがある。それぞれ保存されたモデルに含まれるオブジェクトが得られる。modeldir は RLModelDir のオブジェクトをあらわす。

API modeldir.observation_set

観測値集合である

API modeldir.action_set

行動値集合である

API modeldir.actionvalue_function

行動価値関数である

API modeldir.preference

preference である

API modeldir.numerator

numerator である

API `modeldir.denominator`

`denominator` である

API `modeldir.statevalue_function`

状態価値関数である

API `modeldir.controller`

方策・`Actor` である

第9章　粒子フィルタ

9.1　概要

S4 Simulation SystemSimulation System で粒子フィルタ [4] を使用する場合は以下の形で各クラスを利用する。

- 粒子フィルタのみを利用する場合
  1. Particles クラスを継承して粒子群を表すクラスを定義する。
  2. ParticleFilter クラスを通して、粒子フィルタの各処理を実行する。
- 粒子フィルタを用いたエージェントシミュレーションを行う場合
  1. Particles クラスを継承して粒子群をあらわすクラスを定義する。
  2. ParticleFilterAgentSetBase、ParticleFilterAgentBase を継承したクラスを用いてエージェント集合、エージェントを定義する。各クラスの中で Particles または ParticleFilter の各機能を利用する。

本章では、粒子群を表すクラスの定義に必要な Particles クラスと、粒子フィルタの処理を提供する ParticleFilter クラスについて説明する。なお、ParticleFilterAgentSetBase、ParticleFilterAgentBase については 3.3.5 節、3.4.5 節に詳細が記載されている。

9.2　粒子群クラス

粒子群をあらわすクラスは具体的には ssm.Particles クラスを継承したクラスとして定義される。以下、Particles クラスが持つメソッドについて順に説明する。

API `ssm.Particles(initial_states, observation=None)`

粒子群オブジェクトを作成する。
`initial_states` には、粒子群の初期状態を `numpy` の行列またはモニターで指定する。各行が 1 つの状態を表す。
`observation` には、初期状態を生成する際に使用した観測値を指定する。`observation` は初期状態の粒子群を記録する場合に使用される。初期状態を記録しない場合には、`observation` を指定する必要はない。

API `particles.calcLoglikelihoods(states, observations)`

対数尤度を計算する。継承先で必ず実装しなければならい。
`states` には、粒子群の状態を `numpy` の行列またはモニターで指定する。各行が 1 つの状態を表す。
`observations` には、観測値の集合を `numpy` の行列またはモニターで指定する。各行が 1 つの観測値を表す。返り値として、各状態と各観測値の対数尤度を表した行列を返すとする。このとき、行列の各行は粒子の状態、各列は観測値に対応する。行列の行数は `states` の行数と一致し、列数は `observations` の行数と一致しなければならない。基本的には、観測値を $\{O_1, O_2, ..., O_n\}$ とし、$X_1, X_2, ..., X_m$ の粒子群の状態があれば、返り値の (i, j) 成分は $\log(Pr(O_j|X_i))$ となる。また、1 要素でも有限でない値（`Inf`、 `-Inf`、 `NaN`）を含む列があれば、その列に対応する観測値には対応付けされない。

API `particles.resample(states, loglikelihoods. observation)`

リサンプリング処理を行う。デフォルトでは、対数尤度 loglikelihoods の値に応じて states をリサンプリングする。
states には、粒子群の状態を numpy の行列またはモニターで指定する。各行が 1 つの状態を表す。
loglikelihoods には、粒子群の状態に対応する対数尤度を numpy のベクトルで指定する。各要素が粒子の対数尤度に対応する。
返り値として、リサンプリング結果を表す numpy の行列またはモニターを返すとする。各行が 1 つの状態を表す。
直接挿入法などの別の方法を使用する場合はオーバーライドする。

API `particles.calcNext(states)`

粒子群の次の状態を計算する。継承先で必ず実装しなければならない。
states には、粒子群の状態を numpy の行列またはモニターで指定する。各行が 1 つの状態を表す。
返り値として、粒子群の次の状態を表す行列を返すとする。各行が 1 つの状態を表す。None を返した場合は、粒子が消滅したことを意味する。

API `particles.calcStats(stat_name, p=None)`

粒子群の状態の統計量を、各次元ごとに計算する。計算の際には対数尤度に応じた重みがつけられる。
loglikelihoods が与えられていない場合には、全ての重みを $1/n$ として計算する。また、粒子群の状態がモニターで与えられており、特に文字列型の列を持つ場合には、その列に対応する位置に Nan が挿入される。
stat_name には計算する統計量を表す文字列を指定する。指定できる文字列には、mean（平均）、max（最大値）、min（最小値）、median（中央値）、sd（標準偏差）、percentile（パーセンタイル）がある。パーセンタイルの場合は、パーセンテージ p を $0 \leq p \leq 100$ で指定する。

各種統計量は以下のように定義される。

$$\begin{aligned}
\texttt{mean} &= \frac{\sum_{i=0}^{n-1} w_i x_i}{\sum_{i=0}^{n-1} w_i} \\
\texttt{sd} &= \sqrt{\frac{m \sum_{i=0}^{n-1} w_i (x_i - \texttt{mean})^2}{(m-1)\sum_{i=0}^{n-1} w_i}} \\
\texttt{percentile}(p) &= x_k + \frac{p - p_k}{p_{k+1} - p_k (x_{k+1} - x_k)} \\
\texttt{median} &= \texttt{percentile}(50)
\end{aligned}$$

ここで、状態の 1 つの列を $\{x_i\}$、観測数を n、状態に対する対数尤度を $\{w_i\}$、$\{w_i\}$ に含まれる非零要素数を m とする。また、k は、

$$p_k = \frac{100}{\sum_{i=0}^{n-1} w_i} (\sum_{i=0}^{k} w_i - \frac{w_k}{2})$$

に対して、$p_k \leq p < p_{k+1}$ を満たす k である。

API `particles.states`

現在の状態を返す。numpy の行列またはモニターであり、各行が 1 つの状態を表す。

API `particles.loglikelihoods`

現在対応している観測値に対する対数尤度である。numpy のベクトルであり、各要素が 1 つの状態に対する対数尤度を表す。観測値が対応していない場合は None となる。

API `particles.observation`

対応する観測値を返す。

API `particles.nextStates`

観測値による対応付けが行われる前の粒子群の状態を返す。numpy の行列またはモニターであり、各行が 1 つの状態を表す。

API `particles.isAlive`

粒子が存在していれば True、消滅していれば False を返す。

9.3 粒子フィルタの処理

粒子フィルタの各処理は ssm.ParticleFilter クラス内に定義されている。基本的には、次の流れで ParticleFilter は使用される。

1. ParticleFilter を利用する側で粒子群をあらわすオブジェクトを生成し、add を通して ParticleFilter に登録する。
2. calcNext で登録した各粒子群の次の状態を生成する。
3. 以下を繰り返す。
   (a) 消滅した（次の状態が None）である粒子群を remove で削除する。
   (b) filtrate を呼び出して尤度計算と対応付けを行う。
   (c) notAssociatedObservations と notAssociatedParticles を確認し、必要なら以下の処置をとる。
      i. 対応が付かなかった観測値に対して粒子群を生成し add により登録する。
      ii. 対応が付かなかった粒子群を remove で削除する。
   (d) calcNext でリサンプリング処理と次の状態の生成を行う。

以下、ParticleFilter が持つメソッドについて順に説明する。

API ssm.ParticleFilter(name, interval = 1, save_particle_history = True, save_association_history = True, save_stats_history = True)

粒子フィルタオブジェクトを作成する。
name には履歴を記録するモニターの接頭辞を指定する。
interval には履歴を記録する際の時間間隔を指定する。
save_particle_history が True の場合は、particleHistory のモニターに粒子の状態が記録される。
save_associaton_history が True の場合は、associateHistory のモニターに対応付けの結果が記録される。
save_stats_history が True の場合は、statsHistory のモニターに粒子の状態の統計量が記録される。

API `particleFilter.add(particles, id=None)`

粒子フィルタに粒子群 particles を追加する。
particles は ssm.Particles クラスを継承したものである必要がある。
id には、particles を区別するユニークな番号を指定する。None を指定した場合には自動的に決められる。
追加した粒子群に観測値 observation が与えられていた場合、particleHistory、associateHistory、statsHistory に初期状態を記録する。

API `particleFilter.remove(particles)`

粒子フィルタから particles を削除する。
particles は ssm.Particles クラスを継承したものである必要がある。

API `particleFilter.filtrate(observations)`

フィルタリング処理を行う。observations には、対応付けを行う観測値を numpy の行列またはモニターで指定する。各行が 1 つの観測値を表す。
具体的には、次の処理が行われる。

1. 各粒子群の状態 states を nextStates で更新する。
2. 全ての観測値･粒子の組に対して対数尤度計算を行う。対数尤度計算には各粒子の calcLoglikelihoods が使用される。
3. 観測値と粒子の対応付けを行う。対応付け後には、各粒子の loglikelihoods と observation が更新される。また、対応づけされなかった観測値は notAssociatedObservation に、対応付けされなかった粒子群は notAssociatedParticles に追加される。

API `particleFilter.calcNext()`

各粒子群の次の状態を計算する。具体的には、次の処理が行われる。

1. 観測値と対応が付いている粒子群について、リサンプリング処理を行う。ただし、直前に生成された粒子群についてはリサンプリング処理を行わない。リサンプリング処理には各粒子の `resample` が使用される。処理後は各粒子の `states` が更新される。
2. 各粒子の次の状態を計算する。次の状態の計算には各粒子の `calcNext` が使用される。処理後は各粒子の `nextStates` が更新される。

API `particleFilter.particles`

粒子フィルタに登録された粒子群を表す。粒子群の `ID` がキーであり、値が粒子群 `ssm.Particles` のオブジェクトである辞書を返す。

API `particleFilter.loglikelihood`

評価時点の全体の対数尤度を返す。

API `particleFilter.notAssociatedObservations`

対応付けされなかった観測値のリストを返す。

API `particleFilter.notAssociatedParticles`

対応付けされなかった粒子群のリストを返す。

API `particleFilter.particleHistory`

粒子の状態を記録したモニターを返す。
初期化の際に `save_paritcle_history` を `Flase` とした場合は `None` が設定されている。以下の項目が時刻ごとに記録される。

時刻 シミュレーション時刻である。

ID 粒子群の `ID` である。

尤度 対応付けされた観測値との尤度である。

粒子の状態 各粒子群における粒子の状態である。状態がモニターで与えられた場合、列名はモニターの列名と一致する。そうでない場合、列名は「状態の次元 1」、「状態の次元 2」、... となる。

API `particleFilter.associateHistory`

対応付けの結果を記録したモニターを返す。
初期化の際に `save_association_history` を `Flase` とした場合は `None` が設定されている。以下の項目が時刻ごとに記録される。

時刻 シミュレーション時刻である。

ID 粒子群の `ID` である。

観測値 粒子群に対応付けられた観測値である。観測値がモニターで与えらた場合、列名はモニターの列名と一致する。そうでない場合、列名は「観測値次元 1」、「観測値次元 2」、... となる。

API `particleFilter.statsHistory`

粒子の状態の各統計量を記録したモニターを返す。
初期化の際に save_stats_history を Flase とした場合には None が設定されている。以下の項目が時刻ごとに記録される。

時刻 シミュレーション時刻である。

ID 粒子群の ID である。

粒子数 粒子群の状態の数である。

状態の統計量 粒子群の状態の次元ごとの統計量である。統計量には、以下のものがある。

- 平均
- 標準偏差
- 95%信頼区間下限
- 95%信頼区間上限
- 変動係数
- 最小値
- 25%値
- 中央値
- 75%値
- 最大値
- レンジ

各統計量には各粒子の尤度 loglikelihoods に応じた重みがつけられる。観測値が対応付かない場合には、尤度はすべて $1/n$（n は粒子数）として計算される。
状態がモニターで与えられた場合、列名はモニターの列名と統計量の名称で設定される。そうでない場合、列名は「状態の次元 1 の平均」、「状態の次元 2 の平均」、... となる。

平均、標準偏差、各分位値の定義については、8.2 節 calcStats と同様である。また、状態の 1 つの列を $\{x_i\}$ 、平均を mean、標準偏差を sd とすると、変動係数 cv、p * 100%信頼区間 confInterval(p)、レンジ range は以下となる。

$$\mathtt{cv} = \frac{\mathtt{sd}}{\mathtt{mean}}$$

$$\begin{aligned} \texttt{range} &= \max_{0\le i\le n-1}(x_i) - \min_{0\le i\le n-1}(x_i) \\ \texttt{confInterval}(p) &= \texttt{mean} \pm t_{n-1,(1-p)/2}\frac{\texttt{sd}}{\sqrt{n}} \end{aligned}$$

ここで、$t_{\nu,\alpha/2}$ は自由度 ν の t 分布の、片側確率が $\alpha/2 * 100\%$となる t 値を表す。

第10章　地図エディタ

10.1　Python コンソール

環境アイコンの一つである SFM 地図には地図編集機能があり、地図エディタを開いてシミュレーション時に用いる地図をグラフィカルに編集することができる。地図エディタには Python コンソールも付随しており、プログラム的な操作も可能になっている。

本章では、この Python コンソールから呼び出せる API について説明する。

10.2　API

注意：下記以外の方法でこれらのオブジェクトを変更してはならない。

API `getAllLayers()`

すべてのレイヤーを含むリストを返す。

API `addLayer(name = None)`

名前が name であるレイヤーを追加する。 name = None の場合、「レイヤー X」（X は添え字）という名前が自動的に振られる。

API `removeLayer(layer)`

レイヤー layer を取り除く。

API `swapLayers(i, j)`

添え字 i のレイヤーと添え字 j のレイヤーを入れ替える。

API `addUserDefinedRegion(layer, sign, points, attrs = None)`

正負が sign の真偽で、形状が（x, y）座標のリスト points で表現され、属性が attrs であるユーザー定義領域を作成し、layer に追加する。

API `removeUserDefinedRegion(uregion)`

ユーザー定義領域 uregion を削除する。

API `removeUserDefinedRegions(uregions)`

ユーザー定義領域のリスト uregions を受け取り、各領域を削除する。各ユーザー定義領域は所属レイヤーが異なっていてもよい。

API `groupUserDefinedRegions(uregions, attrs)`

ユーザー定義領域のリスト uregions をグループ化する。attrs はグループの属性となる。 uregions の各要素が同一のレイヤーに属していない場合、エラーとなる。

API `ungroupUserDefinedRegion(ugroup)`

ユーザー定義領域のグループ ugroup を解除する。

API `getUserDefinedRegionTable(layer)`

ユーザー定義領域のレイヤー layer に対応する属性テーブルを返す。

API `getAllPathPoints()`

すべての経路地点を含むリストを返す。

API `addPathPoint(x, y, r, attrs = None)`

位置（x, y）に半径 r の、属性が attrs である経路地点を作成する。

API `removePathPoint(pathPoint)`

経路地点 pathPoint を削除する。

API `removePathPoints(pathPoints)`

経路地点のリスト pathPoints を受け取り、各経路地点を削除する。

API `getPathPointTable()`

経路地点の属性テーブルを返す。

API `getAllPathEdges()`

すべてのエッジを含むリストを返す。

API `addPathEdge(p0, p1, attrs = None)`

経路地点 p0, p1 をつなぐ、属性が attrs であるエッジを作成する。

API `removePathEdge(pathEdge)`

エッジ pathEdge を削除する。

API `removePathEdges(pathEdges)`

エッジのリスト pathEdges を受け取り、各エッジを削除する。

API `getPathEdgeTable()`

エッジの属性テーブルを返す。

レイヤー、ユーザー定義領域、経路地点、エッジ、属性テーブルはそれぞれクラス Layer, UserDefinedRegion, PathPoint, PathEdge, AttributeTable で表現される。 Layer は list を継承しており、各要素が UserDefinedRegion になっている。属性テーブルは dict を継承しており、属性名と属性値がそれぞれ key と value になっている。また、これらは下記に示す独自のメソッドも有している。

API Layer.getName()

レイヤーの名前を返す。

API Layer.setName(name)

レイヤーの名前を name にする。

API UserDefinedRegion.getPolygon()

領域の形状を表す多角形オブジェクトを返す。

API setPolygon(polygon)

領域の形状を表す多角形オブジェクトをセットする。

API UserDefinedRegion.getAttrs()

属性辞書を返す。

API UserDefinedRegion.setAttr(attr, value)

既存の属性 attr に値 value をセットする。

API UserDefinedRegion.includes(x, y)

領域が座標（x, y）を含んでいれば True を、さもなくば False を返す。境界上の場合はどちらの値にもなりうる。

API UserDefinedRegion.isGroup()

領域がグループなら True を、さもなくば False を返す。

API UserDefinedRegion.select(shift = False)

領域を選択する。 shift が False なら他の選択は解除され、 True なら解除されない。

API `UserDefinedRegionGroup.getUserDefinedRegions()`

領域を構成しているユーザー定義領域のリストを返す。

API `PathPoint.getPos()`

経路地点の座標をタプル形式（x, y）で返す。

API `PathPoint.setPos(x, y)`

経路地点の座標をセットする。

API `PathPoint.getRadius()`

経路地点の半径を返す。

API `PathPoint.setRadius(r)`

経路地点の半径をセットする。

API `PathPoint.getAttrs()`

属性辞書を返す。

API `PathPoint.setAttr(attr, value)`

既存の属性 `attr` に値 `value` をセットする。

API `PathPoint.select(shift = False)`

経路地点を選択する。 `shift` が `False` なら他の選択は解除され、 `True` なら解除されない。

API `PathEdge.getPathPoints()`

エッジがつないでいる経路地点を返す。

API `PathEdge.getAttrs()`

属性辞書を返す。

API `PathEdge.setAttr(attr, value)`

既存の属性 `attr` に値 `value` をセットする。

API `PathEdge.select(shift = False)`

経路地点を選択する。 `shift` が `False` なら他の選択は解除され、 `True` なら解除されない。

API `AttributeTable.add(attr, defaultValue)`

属性 `attr` を追加し、デフォルト値を `defaultValue` とする。

API `AttributeTable.remove(attr)`

属性 `attr` を削除する。

API `AttributeTable.set(attr, defaultValue)`

既存の属性 `attr` にデフォルト値 `defaultValue` をセットする。

第11章　その他

11.1　その他

11.1.1　バージョン番号の取得

API　`psimVersion`

バージョン番号を返す。5 つの組からなら `tuple` で、それぞれ、メジャーバージョン、マイナーバージョン、リビジョン、ビルドナンバー、ビルド日時を示す。

関連図書

[1] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee, 'Natural actor-critic algorithms,' Automatica, vol. 45, no. 11, pp. 2471-2482, Nov. 2009.

[2] Geraerts, R.J.and Overmars, M.H. 'The Corridor Map Method: A General Framework for Real-Time High-Quality Path Planning' Computer Animation and Virtual Worlds, volume 18, pp. 107 - 119. 2007.

[3] D. Helbing and P. Molnar, 'Social Force Model for Pedestrian Dynamics', Physical Review E, 51, 5 1995.

[4] 北川源四郎, 'モンテカルロ・フィルタおよび平滑化について,' 統計数理, 1996, 44(1), 31-48.

索 引

"ガード名" in 待ち受け結果, 86
+vec, 208
-vec, 208
__init__, 234
 vec, 208

abs(vec), 208
acos(vec), 208
ACT_SPEC, 231
action(), 233
activate(), 4
adamsSolver(), 62
addGlobalHandler(), 85
addLayer(), 277
addPathEdge(), 279
addPathPoint(), 278
addUserDefinedRegion(), 277
AgentBase(), 144
agentBase.agentid, 145
agentBase.agentset, 144
agentBase.findNeighborAgents(), 145
agentBase.getPosition(), 145
agentBase.getRandomPosition(), 145
agentBase.initAfter(), 145
agentBase.screenAlpha, 146
agentBase.screenColor, 146
agentBase.screenMarker, 146
agentBase.screenSize, 146
agentBase.setPosition(), 145
AgentSetBase(), 132
agentSetBase.agents, 133
agentSetBase.env, 132
agentSetBase.findNeighborAgents(), 133
agentSetBase.findVisibleAgents(), 133
agentSetBase.generateAgents(), 133
agentSetBase.getAgentScreen(), 134
agentSetBase.initAfter(), 133
agentSetBase.remove(), 133
agentSetBase.start(), 134
agentSetBase.view(), 134
allEventOf(), 30
allFacilityOf(), 34
allStoreOf(), 48
allTankOf(), 40
anyEventOf(), 29
anyFacilityOf(), 33
anyStoreOf(), 45
anyTankOf(), 38
AsynchronousAgentBase(), 147
asynchronousAgentBase.run(), 147
AsynchronousAgentSetBase(), 135
atan(vec), 208
atan2(vec, vec), 208
AttributeTable.add(), 129, 282
AttributeTable.remove(), 129, 282
AttributeTable.set(), 129, 282

BarabasiAlbertRandomGraphBase(), 102
BatchSimulator(), 183

bdfSolver(), 62
betaDistribution(), 221
binomialDistribution(), 227

cauchyDistribution(), 225
ceil(vec), 208
ChangeoverFacility(), 14
changeoverFacility.capacity, 16
changeoverFacility.monitor, 17
changeoverFacility.name, 16
changeoverFacility.nfailure, 17
changeoverFacility.product, 16
changeoverFacility.sizeBuffer(), 16
changeoverFacility.sizeWaitQueue(), 17
chiSquareDistribution(), 223
CompleteGraphBase(), 103
constantValue(), 220
cont.isActive(), 65
cont.isRunning(), 65
cont.isSuspended(), 65
cont.isTerminated(), 65
cont.isWaiting(), 65
cont.resume(), 67
cont.setResumedHandler(), 68
cont.setSuspendedHandler(), 68
cont.setTerminatedHandler(), 68
cont.suspend(), 67
cont.terminate(), 66
continuousResult.lowerbound(), 61
continuousResult.upperbound(), 61
cos(vec), 208
cosh(vec), 208
CustomGraphBase(), 101

degrees(vec), 208
del monitor[行指定子,...], 198
del monitor[行指定子,:], 198
del monitor[列指定子], 197
del vector[行指定子], 211
densityDistribution(), 229
dop853Solver(), 62
dopri54Solver(), 62

empiricalDistribution(), 229
EnvironmentBase(), 97
environmentBase.draw(panel), 98
environmentBase.drawAgents(panel, agents), 98
environmentBase.findNeighborAgents(), 98
environmentBase.getPosition(), 97
environmentBase.getRandomPosition(), 97
environmentBase.initAfter(), 97
environmentBase.initAttribute(), 97
environmentBase.setPosition(), 97
Euclid2DBase(), 105
euclid2DBase.distance(), 105
event & event, 30
event | event, 29
event >> event, 31
Event(), 7
event.monitor, 9
event.name, 8
event.signal(), 8
event.sizeWaitQueue(), 8
exp(vec), 208
exponentialDistribution(), 220
Expression(), 54

expression.monitor, 62

fabs(vec), 208
facility & facility, 34
facility | facility, 33
facility >> facility, 35
Facility(), 10
facility.capacity, 12
facility.monitor, 13
facility.name, 12
facility.nfailure, 13
facility.sizeBuffer(), 13
facility.sizeWaitQueue(), 13
facilityLock.isAvailable(), 12
facilityLock.release(), 11
fDistribution(), 224
FIFO, 93
finish(), 6
floatVec.confInterval(), 201
floatVec.count(), 200
floatVec.cv(), 200
floatVec.histogram(), 204
floatVec.max(), 200
floatVec.mean(), 200
floatVec.min(), 200
floatVec.range(), 200
floatVec.sd(), 200
floatVec.summary(), 203
floatVec.var(), 200
FloatVector(), 200
floor(vec), 208
FlowItem(), 184
flowItem.getAttributes(), 184
flowItem.getTime(), 184
flowItem.recoredTime(), 184
FlowItemList(), 185
fmod(vec, vec), 208

gammaDistribution(), 222
geometricDistribution(), 227
getAllLayers(), 277
getAllPathEdges(), 279
getAllPathPoints(), 278
getPathEdgeTable(), 279
getPathPointTable(), 279
getUserDefinedRegionTable(), 278
GEXFFormatGraphBase(), 104
GNMRandomGraphBase(), 102
GNPRandomGraphBase(), 102
GraphBase(), 98
graphBase.distance(), 99
graphBase.getPosition(), 100
graphBase.graph, 98
graphBase.layout, 99
graphBase.setPosition(), 99
GraphMLFormatGraphBase(), 105
groupUserDefinedRegions(), 278

handlerObject.removeHandler(), 86
hyperGeometricDistribution(), 227
hypot(vec, vec), 208

initialize(), 2
InputPort(), 186
inputPort.receive(), 186
intVec.confInterval(p), 201
intVec.count(), 201
intVec.cv(), 201
intVec.histogram(), 204
intVec.max(), 201
intVec.mean(), 201
intVec.min(), 201
intVec.range(), 201
intVec.sd(), 201
intVec.summary(), 203
intVec.var(), 201
IntVector(), 200

LatticeGraphBase(), 103
latticeGraphBase.node2xy, 103
latticeGraphBase.xy2node, 103
Layer.getName(), 127, 279
Layer.setName(), 127, 280
Link(), 185
log(vec, vec), 208
log10(vec), 208
logisticDistribution(), 224
logNormalDistribution(), 221

modeldir.action_set, 265
modeldir.actionvalue_function, 265
modeldir.controller, 266
modeldir.denominator, 266
modeldir.numerator, 265
modeldir.observation_set, 265
modeldir.preference, 265
modeldir.statevalue_function, 266
Monitor(), 189
monitor.dim(), 199
monitor.names(), 199
monitor.ncol(), 198
monitor.nrow(), 198
monitor.observe(), 190
monitor.save(), 198
monitor.setName(), 199
monitor.toList(), 199
monitor[i], 191
monitor[パラメータ名], 191
monitor[行指定子, 列指定子], 194
monitor[行指定子, 列指定子] = $\texttt{monitor}_2$, 196
monitor[行指定子, 列指定子] = val, 197
monitor[行指定子, 列指定子] = リスト, 197
monitor[列指定子], 194
monitor[列指定子] = $\texttt{monitor}_2$, 195
monitor[列指定子] = val, 196
monitor[列指定子] = リスト, 196

negativeBinomialDistribution(), 228
next(乱数生成器), 219
noncentralChiSquareDistribution(), 224
noncentralFDistribution(), 225
normalDistribution(), 220
now(), 7
NWAgentBase.e, 156
NWAgentBase.inArea(), 156
NWAgentBase.p, 156
NWAgentBase.setDestination(), 156
NWAgentBase.setStaying(), 157
NWAgentBase.signalTime, 156
NWAgentBase.waitTime, 156
NWAgentSetBase(SFMAgentSetBase), 141
NWAgentSetBase.setEdgeInfo(), 141
NWAgentSetBase.setNodeInfo(), 141
NWEnvironmentBase(SFMEnvironmentBase), 131

objecttVec.histogram(), 204
objectVec.freq(), 201
objectVec.mode(), 201
objectVec.modefreq(), 201
objectVec.summary(), 203
ObjectVector(), 200
OBS_SPEC, 231
observation(), 232
OutputPort(), 185
outputPort.send(), 185

panel.add_screen(), 158
panel.clear(), 158
panel.draw(), 158
paretoDistribution(), 223
particleFilter.add(), 271
particleFilter.associateHistory, 274
particleFilter.calcNext(), 272
particleFilter.filtrate(), 272
particleFilter.loglikelihood, 273
particleFilter.notAssociatedObservations, 273
particleFilter.notAssociatedParticles, 273
particleFilter.particleHistory, 273
particleFilter.particles, 273
particleFilter.remove(), 272
particleFilter.statsHistory, 274
ParticleFilterAgentBase(), 157
particleFilterAgentBase.makeParticleObservation(), 157, 158
ParticleFilterAgentSetBase(), 142
particleFilterAgentSetBase.associateHistory, 144
particleFilterAgentSetBase.filter, 143
particleFilterAgentSetBase.generateAgents(), 142
particleFilterAgentSetBase.initAfter(), 142
particleFilterAgentSetBase.monitor, 143
particleFilterAgentSetBase.particleHistory, 143
particleFilterAgentSetBase.remove(), 143
particleFilterAgentSetBase.statsHistory, 144
particles.calcLoglikelihoods(), 268
particles.calcNext(), 269
particles.calcStats(), 269
particles.isAlive, 270
particles.loglikelihoods, 270
particles.nextStates, 270
particles.observation, 270
particles.resample(), 268
particles.states, 270
PathEdge.getAttrs(), 129, 281
PathEdge.getPathPoints(), 129, 281
PathEdge.select(), 282
PathEdge.setAttr(), 129, 281
PathPoint.getAttrs(), 128, 281
PathPoint.getPos(), 128, 281
PathPoint.getRadius(), 128, 281
PathPoint.select(), 281
PathPoint.setAttr(), 129, 281
PathPoint.setPos(), 128, 281
PathPoint.setRadius(), 128, 281
Planner(), 167
planner.add(), 167
planner.iter(), 167
planner.step(), 168
poissonDistribution(), 228
PowerlawClusterRandomGraphBase(), 103
PriorityQ, 93
process.add(), 168
process.iter(), 168
process.name, 168
process.priority, 168
process.remove(), 168
process.step(), 168

psimDebug(), 93
psimVersion, 283

radians(vec), 208
rayleighDistribution(), 225
releaseIfRequested, 12
removeLayer(), 277
removePathEdge(), 279
removePathEdges(), 279
removePathPoint(), 278
removePathPoints(), 279
removeUserDefinedRegion(), 278
removeUserDefinedRegions(), 278
replay(), 230
replayStep(), 230
replayStepByPercentage(), 230
reward(), 232
RLModelDir(name, basedir, id=None), 264
RLModelSet(simulator, cls, name, save_history=True, save_model=True), 236
RLModelSet.getAction(model_id, episode_id, item, async_reward = False), 237
RLModelSet.setActualAction(model_id, episode_id, act), 239

sample(), 230
screen.arrow(), 161
screen.clear(), 159
screen.colorbar(), 160
screen.colormap(), 160
screen.draw(), 162
screen.ellipse(), 161
screen.imshow(), 160
screen.line(), 160
screen.lines(), 160
screen.point(), 160
screen.polygon(), 161
screen.rectangle(), 161
screen.set_axis_off(), 159
screen.set_axis_on(), 159
screen.set_title(), 159
screen.set_xlabel(), 159
screen.set_xlim(), 159
screen.set_ylabel(), 159
screen.set_ylim(), 159
screen.text(), 161
selectFix(), 187
selectRandomProb(), 187
selectRandomUnif(), 187
selectRL(), 188
selectRL(model, linklist, idfunc, async_reward = False, reward_func_name = u''reward_func''), 188
selectRoundRobin(), 187
selectShortestQueue(), 187
sequenceEventOf(), 31
sequenceFacilityOf(), 36
sequenceStoreOf(), 51
sequenceTankOf(), 42
setGlobalSeed(), 219
SFMAgentBase(), 147
sfmAgentBase.findNeighborAgents(), 155
sfmAgentBase.findVisibleAgents(), 154
sfmAgentBase.getDestination(), 150
sfmAgentBase.getPosition(), 155
sfmAgentBase.getRandomPosition(), 155
sfmAgentBase.inArea(), 153
sfmAgentBase.inObstacle(), 152
sfmAgentBase.inSight(), 153
sfmAgentBase.inSightPoint(), 153

sfmAgentBase.inSightSector(), 153
sfmAgentBase.isInErrorState(), 155
sfmAgentBase.isStopping(), 154
sfmAgentBase.nearestPathPoint(), 153
sfmAgentBase.selectNextNode(), 154
sfmAgentBase.setAgentProfile(), 155
sfmAgentBase.setDestination(), 149
sfmAgentBase.setPosition(), 155
sfmAgentBase.setRandomTransition(), 151
sfmAgentBase.setStaying(), 149
sfmAgentBase.setTravel(), 150
SFMAgentSetBase(), 135
sfmAgentSetBase.closeGate(), 136
sfmAgentSetBase.findNeighborAgents(), 136
sfmAgentSetBase.getNextGreen(), 137
sfmAgentSetBase.getSignalColor(), 137
sfmAgentSetBase.openGate(), 136
sfmAgentSetBase.removeSignal(), 136
sfmAgentSetBase.setAgentProfile(), 137
sfmAgentSetBase.setGate(), 136
sfmAgentSetBase.setSignal(), 136
sfmAgentSetBase.setSignalCrossWalk(), 137
sfmAgentSetBase.setSignalIntersection(), 137
sfmAgentSetBase.setSignalIntersectionWalk(), 137
sfmAgentSetBase.step(), 136
SFMEnvironmentBase(), 111
sfmEnvironmentBase.addLayer(), 113
sfmEnvironmentBase.addObstacle(), 116
sfmEnvironmentBase.addPathEdge(), 115
sfmEnvironmentBase.addPathPoint(), 114
sfmEnvironmentBase.addUserDefinedRegion(), 113
sfmEnvironmentBase.connectPathGraphAuto(), 117
sfmEnvironmentBase.convertPathGraphDirected(), 116
sfmEnvironmentBase.distance(), 119
sfmEnvironmentBase.draw, 119
sfmEnvironmentBase.drawAgents, 119
sfmEnvironmentBase.findNeighborAgents(), 120
sfmEnvironmentBase.getAllLayers(), 113
sfmEnvironmentBase.getAllPathEdges(), 115
sfmEnvironmentBase.getAllPathPoints(), 114
sfmEnvironmentBase.getPathEdgeTable(), 116
sfmEnvironmentBase.getPathPoints(), 115
sfmEnvironmentBase.getPathPointTable(), 115
sfmEnvironmentBase.getPosition(), 120

sfmEnvironmentBase.getRandomPosition(), 120

sfmEnvironmentBase.getUserDefinedRegionTable(), 114

sfmEnvironmentBase.groupUserDefinedRegions(), 114

sfmEnvironmentBase.initLatticePathGraph(), 117

sfmEnvironmentBase.innerPathPoints(), 117

sfmEnvironmentBase.inSight(), 118

sfmEnvironmentBase.inSightByGrid(), 118

sfmEnvironmentBase.inSightPoint(), 118

sfmEnvironmentBase.inSightSector(), 118

sfmEnvironmentBase.nearestPathPoint(), 117

sfmEnvironmentBase.pathgraph, 119

sfmEnvironmentBase.paths(), 119

sfmEnvironmentBase.readGeoJSON(), 125

SFMEnvironmentBase.readGeoJSON(self, fname, abs_path=False, proj = "EPSG:2451", encoding = "utf-8", r = 0.5, margin = 1.0, epsilon = 1e-2, only_maximum_component = True, weakly_connected=False, oneway_key=None, oneway_values=None, image_provider="osm", **keys): , 125

sfmEnvironmentBase.readOSM(), 120

SFMEnvironmentBase.readOSM(fname, lon0, lat0, lon1, lat1, abs_path=False, r=1.0, proj="EPSG:2451", margin=1.0, highway="car", maxspeed="motorway:60,trunk:50,; only_maximum_component=True, weakly_connected=False, clear_cache=False, image_provider="osm",), 120

sfmEnvironmentBase.readTile(), 126

sfmEnvironmentBase.removeLayer(), 113

sfmEnvironmentBase.removePathEdge(), 116

sfmEnvironmentBase.removePathEdges(), 116

sfmEnvironmentBase.removePathPoint(), 115

sfmEnvironmentBase.removePathPoints(), 115

sfmEnvironmentBase.removeUserDefinedRegion(), 114

sfmEnvironmentBase.removeUserDefinedRegions(), 114

sfmEnvironmentBase.sampleInnerPathPoint(), 118

sfmEnvironmentBase.setFacility(), 120

SFMEnvironmentBase.setFacility(name, capacity), 120, 132

sfmEnvironmentBase.setPosition(), 120

sfmEnvironmentBase.ungroupUserDefinedRegion(), 114

SimulatorBase(), 182

simulatorBase.run(), 182

simulatorBase.save(), 182

SimulatorParam(), 183

sin(vec), 208

sinh(vec), 208

SMDP, 232
sqrt(vec), 208
ssm.ParticleFilter(), 271
ssm.Particles(), 267
Stack, 93
start(), 6
store & store, 47
store | store, 44
store >> store, 49
Store(), 21
store.buffer, 25
store.capacity, 24
store.monitor, 26
store.name, 24
store.replace(filter), 25
store.sizeBuffer(), 24
store.sizeGetQueue(), 24
store.sizeGetQueueSum(), 24
store.sizePutQueue(), 25
store.sizePutQueueSum(), 25
swapLayers(), 277
SynchronousAgentBase(), 146
synchronousAgentBase.step(), 146
SynchronousAgentSetBase(), 135
synchronousAgentSetBase.step(), 135

tan(vec), 208
tanh(vec), 208
tank & tank, 39
tank | tank, 37
tank >> tank, 41
Tank(), 18
tank.capacity, 19
tank.monitor, 20
tank.name, 19
tank.sizeBuffer(), 20
tank.sizeGetQueue(), 20
tank.sizeGetQueueSum(), 20
tank.sizePutQueue(), 20
tank.sizePutQueueSum(), 20
task.name, 169
task.priority, 169
task.step(), 169
tDistribution(), 226
timeFloatVec.confInterval(), 202
timeFloatVec.count(), 201
timeFloatVec.cv(), 201
timeFloatVec.histogram(), 204
timeFloatVec.max(), 201
timeFloatVec.mean(), 201
timeFloatVec.min(), 201
timeFloatVec.range(), 201
timeFloatVec.sd(), 201
timeFloatVec.summary(), 203
timeFloatVec.var(), 201
timeIntVec.confInterval(), 202
timeIntVec.count(), 202
timeIntVec.cv(), 202
timeIntVec.histogram(), 204
timeIntVec.max(), 202
timeIntVec.mean(), 202
timeIntVec.min(), 202
timeIntVec.range(), 202
timeIntVec.sd(), 202
timeIntVec.summary(), 203
timeIntVec.var(), 202
TimeMonitor(), 190
timeMonitor.observe(), 191
timeMonitor.save(), 198
timeMonitor.time(), 192
timeMonitor[i], 192
timeMonitor[TimeInterval(開始時刻，終了時刻)，列指定子], 195
timeMonitor[パラメータ名], 192
timeMonitor[行指定子，列指定子],

195
timeMonitor[列指定子], 194
timeObjectVec.freq(), 202
timeObjectVec.histogram(), 205
timeObjectVec.mode(), 202
timeObjectVec.modefreq(), 202
timeObjectVec.summary(), 203
timeVec.time(), 205
timeVector.cor(), 217
timeVector.cov(), 216
timeVector.diff(), 215
timeVector.integrate(), 217
timeVector.movingAverage(), 215
timeVector.quantile(), 216
timeVector.sort(), 216
timeVector[TimeInterval(開始時刻, 終了時刻)], 210
triangularDistribution(), 226

ungroupUserDefinedRegion(), 278
uniformDistribution(), 221
UserDefinedRegion.getAttrs(), 128, 280
UserDefinedRegion.getPolygon(), 127, 280
UserDefinedRegion.includes(), 128, 280
UserDefinedRegion.isGroup(), 128, 280
UserDefinedRegion.select(), 280
UserDefinedRegion.setAttr(), 128, 280
UserDefinedRegion.setPolygon(), 127, 280
UserDefinedRegionGroup.getUserDefinedRegions(), 128, 280

val 2 項演算子 vec, 206
val 比較演算子 vec, 209
Value(), 54
value.monitor, 61
vec 2 項演算子 val, 206
vec 2 項演算子 vec, 205
vec 2 項演算子 リスト, 205
vec 比較演算子 val, 209
vec 比較演算子 vec, 209
vec 比較演算子 リスト, 209
vec 累積演算子 val, 207
vec 累積演算子 vec, 207
vec 累積演算子 リスト, 207
vector.append(), 213
vector.cor(), 214
vector.cov(), 214
vector.cumsum(), 212
vector.diff(), 211
vector.extend(), 213
vector.insert(), 213
vector.movingAverage(), 212
vector.quantile(), 212
vector.reverse(), 213
vector.sort(), 214
vector[行指定子], 210
vector[行指定子] = val, 211
vector[行指定子] = vector$_2$, 210

weibullDistribution(), 223
Widget(), 183
widget.addMonitor(), 184
widget.run(), 184
widget.start(), 184

yield ガード式, 6

ガード式
(event & event).wait(), 30
(event | event).wait(), 29
(event >> event).wait(), 31
(facility & facility).request(), 34
(facility | facility).request(), 33

(facility >> facility).request(), 35
(store & store).get(), 47
(store & store).get1(), 47
(store & store).put(), 47
(store | store).get(), 44
(store | store).get1(), 44
(store | store).put(), 45
(store >> store).get(), 49
(store >> store).get1(), 50
(store >> store).put(), 50
(tank & tank).get(), 39
(tank & tank).put(), 40
(tank | tank).get(), 37
(tank | tank).put(), 38
(tank >> tank).get(), 41
(tank >> tank).put(), 42
addHandler(), 85
allEventOf().wait(), 30
allFacilityOf().request(), 35
allOf(), 27
allStoreOf().get(), 48
allStoreOf().get1(), 48
allStoreOf().put(), 49
allTankOf().get(), 40
allTankOf().put(), 41
alwaysFalse(), 63
alwaysTrue(), 63
anyEventOf(), 30
anyFacilityOf().request(), 33
anyOf(), 27
anyStoreOf().get(), 45
anyStoreOf().get1(), 46
anyStoreOf().put(), 46
anyTankOf().get(), 38
anyTankOf().put(), 39
call(), 72
case(), 63
changeoverFacility.request(), 15
cont.resumed(), 66
cont.suspended(), 66
cont.terminated(), 65
continuous(), 56
continuousWait(), 60
currentProc(), 69
delayedCase(), 63
event.signal(), 8
event.wait(···), 8
facility.request(), 11
facilityLock.release(), 11
facilityLock.stopped(), 12
go(), 75
last(), 92
pause(), 7
recordNow(), 84
recordValue(), 84
runAction(), 83
sequenceEventOf().wait(), 31
sequenceFacilityOf().request(), 36
sequenceOf(), 28
sequenceStoreOf().get(), 51
sequenceStoreOf().get1(), 51
sequenceStoreOf().put(), 52
sequenceTankOf().get(), 42
sequenceTankOf().put(), 43
store.get(), 23
store.get1(), 24
store.put(), 22
store.put1(), 22
subactivate(), 72
suspend(), 68
tank.get(), 19

tank.put(), 19
terminate(), 67
ガード式 & ガード式, 27
ガード式 | ガード式, 27
ガード式 >> ガード式, 28

リスト 2 項演算子 vec, 206
リスト 比較演算子 vec, 209

待ち受け結果.index("ガード名"), 88
待ち受け結果.releaseAllFacilities(facility = None), 90
待ち受け結果.time("ガード名"), 88
待ち受け結果.val, 89
待ち受け結果 ["ガード名"], 87

0