ロボカップサッカーシミュレーション 2D リーグ
必勝ガイド

バージョン 1.0

秋山英久

目 次

第1章
はじめに

本書は，RoboCup サッカーシミュレーション 2D リーグの解説書です．本書の内容に入る前に，RoboCup について概要を簡単に説明します．

Section 1.1
RoboCup とは

RoboCup 公式サイト[1)]のトップページには以下の文が掲載されています．

> “By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.”

日本語に訳すと「西暦 2050 年までにサッカーの世界チャンピオンチームに勝てる完全自律型ヒューマノイドロボットのチームを作る」となります．これが RoboCup が掲げる最終目標です．非常に壮大な目標ですが，これを実現するだけが RoboCup の目的ではありません．目標実現への過程で新しい研究課題と成果を生み出し，関連技術を進歩促進させることも RoboCup の目的のひとつです．

本書執筆時点では，RoboCup には以下の競技部門があります

- サッカー
  シミュレーション，小型，中型，四足，ヒューマノイド
- レスキュー
  シミュレーション，実機

[1)] http://www.robocup.org/

- ジュニア
- @Home

サッカー以外にも，大規模災害へのロボット技術の応用としてレスキュー，教育とホームエンターテインメント向けのジュニア，実世界での生活空間における人間とロボットの共存を扱う@Home などがあります.

本書で扱う内容は，これらのうちサッカーシミュレーションリーグです.

Section 1.2
RoboCup の歴史

RoboCup 世界大会は 1997 年の第一回大会以来，以下のように毎年開催されています.

第 1 回大会	RoboCup 1997 (名古屋)
第 2 回大会	RoboCup 1998 (パリ，フランス)
第 3 回大会	RoboCup 1999 (ストックホルム，スウェーデン)
第 4 回大会	RoboCup 2000 (メルボルン，オーストラリア)
第 5 回大会	RoboCup 2001 (シアトル，アメリカ)
第 6 回大会	RoboCup 2002 (福岡)
第 7 回大会	RoboCup 2003 (パドヴァ，イタリア)
第 8 回大会	RoboCup 2004 (リスボン，ポルトガル)
第 9 回大会	RoboCup 2005 (大阪)
第 10 回大会	RoboCup 2006 (ブレーメン，ドイツ)

1996 年にプレロボカップというサッカーシミュレーションのみの競技会も開催されたそうです．そのため，サッカーシミュレーションは最も古いリーグであると言われています.

筆者自身は 2000 年のメルボルン大会からサッカーシミュレーションリーグに参加しています.

Section 1.3
サッカーシミュレーションリーグ

1.3.1 マルチエージェントシステムとしての RoboCup

自律的に動作して外界と相互作用するプログラムをエージェントと呼ぶことがあります．エージェントとは，日本語で言うと代理人などの意味になります．コンピュータの分野でエージェントと言うといろいろな意味で使用されていて明確な定義があるわけでは無いそうで，“自律性を持った行動主体” という点が定義として共通する点だそうです．

人工知能分野では，単一のエージェントでは解決の難しい問題を複数のエージェントが協調して解決するマルチエージェントシステムが研究されています．RoboCup サッカーシミュレーションは，まさにこのマルチエージェントシステムであり，マルチエージェントシステムのテストベッドと呼ばれています．

1.3.2 競技内容

サッカーシミュレーションリーグは全競技の中でも特に参加チームが多く，リーグ内部で，2D，3D，コーチの 3 種類の競技が行われています．

2D リーグは最初期から存在している競技です．高さの概念が無いため，物体は平面上を滑るように移動します．しかし，それ以外はかなりそれらしく 11 対 11 のサッカーを再現しています．古くから存在しているだけあって，チームプレイの完成度は RoboCup 全競技の中でも郡を抜いています．本書ではこの 2D リーグを扱います．

3D リーグは 2004 年から開始された新しい競技です．高さの概念を加え，実時間シミュレーションに近付いたシミュレータを使用しています．今後は 2D リーグを縮小し，3D リーグをメインに据えようというのがコミュニティの流れとなってます．しかし，まだまだシミュレータも開発途上で満足に試合が行える状態ですらありません．今後の進化に期待したいところですが，2D リーグの域に達するにはまだ時間がかかりそうです．

コーチリーグは，チームの監督の能力を測ることに注目した競技です．試合内容の大局的な分析を行い，適切なアドバイスを与えることでチームのパフォーマ

ンスをリアルタイムに改善することを目的としています．コーチリーグでは 2D リーグと同一のシミュレータを使用しています．

1.3.3 最新情報を入手するには

RoboCup は毎年のルール変更が当り前であるため，頻繁に情報が変更されます．

Web サイト

以下の Web サイトをこまめのチェックしましょう．

- http://sourceforge.net/projects/sserver/
  サッカーシミュレーションリーグ公式開発サイト．最新のシミュレータのソースコードはここから入手できます．
- http://rc-oz.sourceforge.jp/
  サッカー/レスキューシミュレーションリーグ日本公式サイト．日本国内で行われるイベントなどの案内が掲載されます．

メーリングリスト

関連メーリングリストへの入会を強くお薦めします．競技会やキャンプの情報が最も早く流れるのはメーリングリストです．それに続いて，Web サイトが更新されるはずですが，すぐに更新されるとは限りません[2)]．

- robocup-sim (https://mailman.cc.gatech.edu/mailman/listinfo/robocup-sim)
  サッカーシミュレーションリーグ国際公式メーリングリスト．世界大会や最新のシミュレータの情報のほとんどはここで流れます．
- SimJP (http://lists.sourceforge.jp/mailman/listinfo/rc-oz-simjp)
  サッカー/レスキューシミュレーションリーグ日本語専用メーリングリスト．国内のイベントの情報などが流れます．質問も可です．
- RoboCup-J (http://www.robocup.or.jp/ml.html)
  RoboCup 関連の日本公式メーリングリスト．シミュレーション関連の情報はほとんど流れません．

[2)] 場合によってはイベント自体終了してからとか…

Section 1.4
本書掲載・付属のプログラムについて

本書では，サッカーシミュレーション 2D リーグ用サッカーエージェント開発のためのライブラリや関連プログラムをソースコードで提供しています．本書掲載・付属のプログラムは開発継続中のものであるため，今後もバージョンアップを予定しています．最新バージョンのリリースは以下の筆者の開発サイトで行う予定です．

http://rctools.sourceforge.jp/

バグレポートなどもこちらまでお願いします．

本書で使用するプログラミング言語は C++です．ある程度の C++の知識が前提となるため，C++未経験の読者には難易度が高いかもしれません．C++自体が難解な言語と言われていますが，最近は良い本が多く出ています．以下の本の内容を理解していれば，本書のプログラムやその書き方の意図も理解できるでしょう．

- 「Accelerated C++」[19]
  C は知っている，C++みたいな C なら書けるという初心者向けの本です．
- 「Effective C++」 [22]
  C++を正しく使うための Tips 集です．一度は読みましょう．
- 「Exceptional C++」[17]
  同じく C++を正しく使うための Tips 集です．一度は読みましょう．

また，本書で扱うプログラム中では STL や Boost といったテンプレートライブラリも多用しています．それぞれの参考書としては以下の本がお薦めです．

- 「STL 標準講座」[16]
  STL を一通り網羅し，なおかつ豊富なサンプルコードが掲載されています．これから STL を学ぶ人に適しているでしょう．
- 「Effective STL」 [23]
  STL を正しく使うための Tips 集です．一度は読みましょう．
- 「Boost C++ Library プログラミング」 [21]
  膨大な Boost を網羅しています．

本書掲載および付属 CD-ROM 収録のプログラムの使用にあたっては，LGPL に従うものとします.

第2章

RoboCupサッカーシミュレータ

本章では，サッカーシミュレーションの実行環境となる RoboCup サッカーシミュレータについて，その仕組みからインストール方法と実際の使用方法までを説明します.

Section 2.1
RoboCup サッカーシミュレータとは

RoboCup サッカーシミュレータとは，その名のとおり，コンピュータ上でサッカーシミュレーションを実行するためのソフトウェアです．英語では The RoboCup Soccer Simulator と書き，RCSoccerSim と略して表記されます．サッカーサーバシステム (Soccer Server System) と呼ばれることもありますが，現在はサッカーシミュレータと呼ぶことの方が多いようです．RoboCup サッカーシミュレータはオープンソースのプログラムで，無償で利用できます.

2.1.1 歴史

元祖のサッカーシミュレータは 1993 年に電子技術総合研究所 (現在は産業技術研究所に統合) の野田五十樹氏によって開発されました．以後，RoboCup の公式サッカーシミュレータとして採用され，さまざまな人の手による改良を経て，2005 年 7 月にバージョン 10.0.7 が公開されました.

バージョン 10 をもって，RoboCup 公式競技のための積極的なルール変更や機能追加は停止されることになっています．しかし，これは逆に，シミュレータが成熟し，チーム開発者は新機能に追従するための苦労から開放されたことを意味

します．マルチエージェントシステムとしてチームが成熟してくるのはこれからなのです．

2.1.2 シミュレータの構成

RoboCup サッカーシミュレータは単体のプログラムではなく，複数のプログラムを連携させてシミュレーション環境を提供する統合システムとなっています．RCSoccerSim と呼ぶ場合，主に以下のプログラムパッケージが含まれます．

- rcssbase
  RCSoccerSim に含まれる各プログラムが使用する基本ライブラリ．
- rcssserver
  シミュレータ本体．
- rcssmonitor
  画面表示プログラム．
- rcsslogplayer
  ログ再生プログラム．

2.1.3 シミュレータの仕組み

サッカーシミュレータにおいて，実際のシミュレーションを担当するプログラムは rcssserver というプログラムです．この名前から想像がつくとおり，rcssserver はサーバプログラムです．

RoboCup サッカーシミュレータでは，サーバクライアント方式によって分散マルチエージェントシミュレーションを実現しています．サーバクライアント方式 (またはクライアントサーバ方式) とは，プログラムをサーバとクライアントに分け，それぞれのプロセスで役割分担をして処理を行う仕組みのことです．サーバは情報を集中管理し，クライアントはその情報を利用するという，ネットワークプログラムにおいては極めて一般的なモデルです．RoboCup サッカーシミュレータで動作するサッカーエージェントは，rcssserver と通信するクライアントプログラムとなります．

シミュレーション実行時の各プログラムの関係は図 2.1 のようになります．

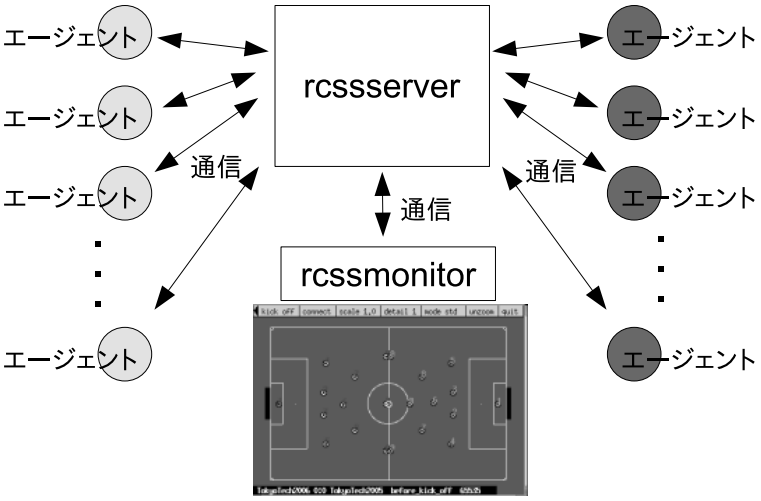


図 2.1 シミュレーション実行時のプログラムの関係図

サッカーエージェントは知覚情報となるメッセージを rcssserver から受け取り，それに応じて自分の行動コマンドのメッセージを rcssserver へ送信します．このメッセージの送受信を繰り返すことでフィールド上の物体の状態が変化し，シミュレーションが進行します．

個々のエージェントが独立して通信を行う点にも注目してください．これは，サッカーエージェントは全て独立して制御されることを意味します．公式競技においては，ひとつのクライアントプログラムが制御できるのはひとつのエージェントのみと規定されています．そして，エージェント間の情報共有は rcssserver を介したコミュニケーションでしか許可されていません．このようにして，完全な分散マルチエージェントシミュレーションが実現されています．

Section 2.2
シミュレータのインストール

2.2.1 実行環境

本書ではシミュレータおよびサッカーエージェントプログラムの実行環境として Linux を想定しています．Linux を使用する理由は，公式競技において使用される OS が Linux であるからです．公式に配布されるサッカーシミュレータも，Linux ならば間違いなく安定動作します．

RoboCup サッカーシミュレーションに本気で取り組むつもりならば，まずは Linux や FreeBSD などの UNIX 系 OS を用意してください．Mac OS X でも動くようです．どうしても Windows しか使えない場合は，VMware のような仮想ソフトや coLinux[1] などの解決法があります．ただし，サッカーシミュレーションの実行にはそれなりのマシンパワーが要求される (2GHz クラス以上の CPU が望ましい) ため，これらのソフトを利用しても恐らく満足いく結果は得られないでしょう．

筆者は Gentoo Linux[2] の使用をお薦めします．インストールには非常に手間がかかりますが，後の運用はとても楽です．RoboCupPortag というソフトウェアを利用して，シミュレータ一式をコマンドひとつでインストール，アップデートできてしまいます．

2.2.2 Linux へのインストール

ここでは，一般的な Linux ディストリビューションでのインストール手順を説明します．プログラムバイナリなどは自分のホームディレクトリにインストールします．これは，rcssbase が独自に Boost ライブラリを取り込んでおり，これを/usr/local へインストールすると他のプログラムのコンパイル時に不具合を発生するためです．

[1] Windows 上で Linux カーネルを実行できる．http://www.colinux.org/
[2] http://www.gentoo.org/

前準備

ホームディレクトリにある .bash_profile というファイルを編集して，実行パスとライブラリパスを追加します．以下の 2 行をファイルの末尾に追加してください．

```
export PATH=$HOME/rcss/bin:$PATH
export LD_LIBRARY_PATH=$HOME/rcss/lib:$LD_LIBRARY_PATH
```

一旦ログアウトしてログインし直します．

シミュレータの入手

サッカーシミュレータのプログラムパッケージはサッカーシミュレーションリーグの公式サイトで配布されています．配布ページは以下になります．

http://sourceforge.net/projects/sserver/

「Download The RoboCup Soccer Simulator」というボタンを押すと，Latest File Releases という一覧が表示されます．この中から最低限以下のものをダウンロードします．

- RCSSBase Official Release, Release 10.0.11
- RCSSServer Official Release, Release 10.0.7
- RCSSMonitor Release Candidate, Release 10.0.0

それぞれ，rcssbase-10.0.11.tar.gz，rcssserver=10.0.7，rcssmonitor-10.0.0.tar.gz というファイルをダウンロードします．これらのファイルよりも新しいバージョンがリリースされていれば，そちらを使用してください．

rcssbase のビルド，インストール

以下のコマンドを順に実行するだけです．

```
$ tar xzvf rcssbase-10.0.11.tar.gz
$ cd rcssbase-10.0.11
$ ./configure --prefix=$HOME/rcss
$ make
$ make install
```

rcssserver のビルド，インストール

以下のコマンドを順に実行します.

```
$ tar xzvf rcssserver-10.0.7.tar.gz
$ cd rcssserver-10.0.7
$ ./configure --prefix=$HOME/rcss RCSSBASE=$HOME/rcss
$ make
$ make install
```

~/rcss/bin/rcsoccersim というファイルを開き，以下の行を探してください.

```
kill -s SIGINT $PID
```

この行を以下のように書き換えてください.

```
kill -s INT $PID
```

rcssmonitor のビルド，インストール

ソースアーカイブを展開します.

```
$ tar xzvf rcssmonitor-10.0.0.tar.gz
```

gcc 3.4 以降を使用している場合はパッチを当てる必要があります[3].

```
$ patch -p0 < rcssmonitor-10.0.0-gcc34.patch
```

GNU Autotools 関連のファイルがいくつパッケージに含まれていないので，**autoreconf** コマンドを実行してシステムからコピーしてからインストール作業を行います.

```
$ cd rcssmonitor-10.0.0
$ autoreconf -i
$ ./configure --prefix=$HOME/rcss
$ make
$ make install
```

[3] http://cvs.sourceforge.jp/cgi-bin/viewcvs.cgi/rc-oz/RoboCupPortage/portage/sci-misc/rcssmonitor/files/ よりパッチファイルを入手可能.

確認

以下のように **which** コマンドを実行し，エラーメッセージが表示されなければインストール成功です．

```
$ which rcssserver
$ which rcssmonitor
$ which rcsoccersim
```

2.2.3 シミュレータの起動

それでは，いよいよシミュレータを起動します．**rcsoccersim** というコマンドを実行してください．

```
$ rcsoccersim
rcssserver-10.0.7

Copyright (C) 1995, 1996, 1997, 1998, 1999 Electrotechnical Laboratory.
2000, 2001, 2002, 2003, 2004 RoboCup Soccer Server Maintenance Group.

Using rcssbase-10.0.11

Trying to create configuration file: /home/akiyama/.rcssserver/server.conf
Created configuration file: /home/akiyama/.rcssserver/server.conf
Hetero Player Seed: 314226
wind factor: rand: 0.000000, vector: (0.000000, 0.000000)

Hit CTRL-C to exit
Copyright (c) 1999 - 2001, Artur Merke <amerke@ira.uka.de>
Copyright (c) 2001 - 2002, The RoboCup Soccer Server Maintainance Group.
        <sserver-admin@lists.sourceforge.net>
reading options from file: /home/akiyama/.rcssmonitor.conf
a new (v2) monitor connected
```

以上のように表示され，図 2.2 のようなウインドウが表示されれば成功です．右上の “quit” をクリックして終了しましょう．

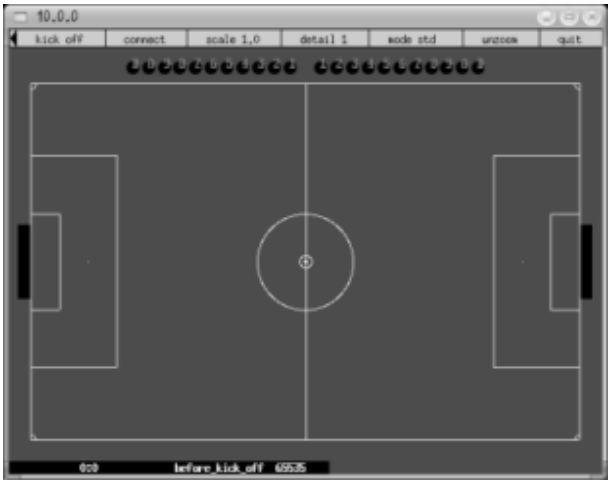


図 2.2 rcsoccersim の実行直後

2.2.4 試合実行

シミュレータが起動できただけでは試合を開始できません．動くチームを入手して動かしてみましょう．

サッカーシミュレーションリーグの世界大会では，競技終了後，チームの実行バイナリをリリースすることが義務づけられています．これらは以下のアドレスから入手できます．

http://www.uni-koblenz.de/ maas/RC2005/Download/Binaries/2D/

http://www.uni-koblenz.de/ maas/Download/Binaries/2D/

http://www.uni-koblenz.de/ fruit/orga/rc03/binaries/

ここでは，2005 年の世界大会参加チームから，`tokyotech_rc2005_2d_bin.tar.gz` と `trilearn_rc2005_2d_bin.tar.gz` をダウンロードしたとします．

ダウンロードしたファイルを展開し，`start.sh` というファイルが存在することを確認してください．

```
$ tar xzvf tokyotech_rc2005_2d_bin.tar.gz
$ ls tokyotech_rc2005_2d/
formations  helios_coach  helios_player  player.conf  start.sh
$ tar xzvf trilearn_rc2005_2d_bin.tar.gz
$ ls trilearn_rc2005_bin/
README           player.conf  trilearn_coach
formations.conf  start.sh     trilearn_player
```

ターミナルを 3 つ開き，ひとつで **rcsoccersim** を実行します．シミュレータの起動が確認できたら，残りふたつで各チームの起動スクリプトをそれぞれ実行してください．

```
$ rcsoccersim
```

```
$ cd tokyotech_rc2005_2d
$ ./start.sh
```

```
$ cd trilearn_rc2005_bin
$ ./start.sh
```

全ての起動に成功すれば，順番にプレイヤがフィールド上に現れ，図 2.3 のような状態になります．

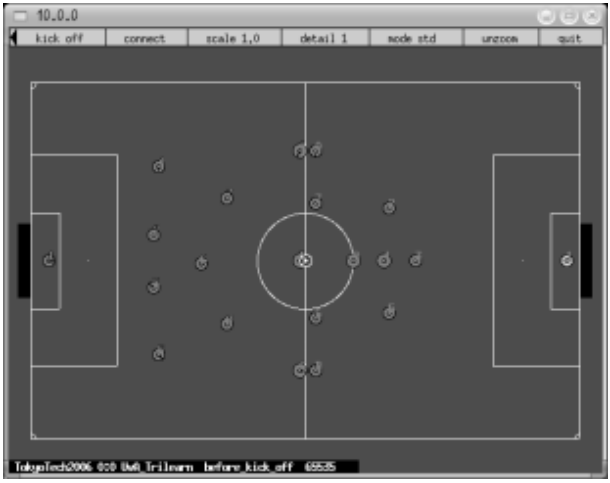


図 2.3 チームを接続した状態

それでは，左上の “kick off” をクリックしてください．試合が始まってプレイヤが動き始めましたか？rcssmonitor の下部にはチーム名，現在のスコア，現在のサイクル数が表示されています．試合の審判は rcssserver が自動で行ってくれます．3000 サイクル経過すると前半終了です．再び “kick off” ボタンを押すと後半を開始できます．試合が終了したら，“quit” をクリックして終了してください．

2.2.5 分散実行

rcssserver と各サッカーエージェントとのメッセージ送受信は，ネットワーク越しでの通信を想定されたものです．そのため，rcssserver と各サッカーエージェントを異なるホストマシン上で分散実行することが可能となっています．実際に試した人はほとんどいないと思いますが，遠隔地で動作する rcssserver とサッカーエージェントとを通信させることも原理的に可能です．一般的には，Ethernet で接続された LAN 内のコンピュータを用いて rcssserver とサッカーエージェントを分散動作させます．

ほとんどのチームのチーム起動スクリプトには，rcssserver が実行されているホスト名を指定するオプションが用意されています．例えば，前節の TokyoTechSFC の場合，server1 というホスト名を持つコンピュータ上で動いている rcssserver に接続するには，以下のようにチームを起動します．

```
$ ./start.sh --host server1
```

ホスト名の文字列部分には IP アドレスも使用できます．

Section 2.3
シミュレータの制御

2.3.1 rcssmonitor の操作

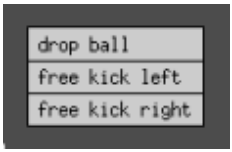
メインメニュー

それぞれ以下の機能を持ちます.

ボタン	説明
kick off	試合を開始する.
connect	rcssserver へ接続し直す.
scale	物体を拡大表示する．マウスの第 1 第 3 ボタンクリックで変更できる.
detail	物体の詳細情報表示レベルを変更する．マウスの第 1 第 3 ボタンクリックで変更できる．レベル 2 でプレイヤのスタミナとボールの速度を表示，レベル 3 でプレイヤのタイプを表示する.
mode	動作モードを変更する．マウスの第 1 第 3 ボタンクリックで変更できる．std は通常モード． move ではプレイヤを任意の位置へマウスドラッグで移動できるようになる．viwe ではプレイヤの視界エリアを表示できるようになる.
unzoom	フィールドをウインドウの大きさに収まるように自動調整する.
quit	ウインドウを閉じ，プログラムを終了する.

ドロップダウンメニュー

rcssmonitor は人間が rcssserver に介入するためのインタフェースを備えています．mode が std のとき，rcssmonitor のフィールド上でマウスの第 2 ボタンまたは第 3 ボタンをクリックすると，以下のようなメニューが表示されます.



上から順に，マウスクリックした位置にボールを落とす，左チームにフリーキックを与える，右チームにフリーキックを与える，という効果があります．これらは，チーム開発時のデバッグのため，rcssserver 組み込みの自動審判では判定できないファウルを処理するためなどに使用されます.

その他の操作

フィールド上でマウスドラッグすることによって選択した領域を拡大表示することができます．他にも，以下のようなマウスやキーボードによる操作が可能となっています．

操作	効果
f	描画を止める．
i	フィールドの大きさを自動調整する．
k	キーの割り当てをコンソールに表示する．
q	rcssmonitor を終了する．
+	ズームイン．
-	ズームアウト．
クリック	描画中心位置をクリック位置へ変更．
ドラッグ	領域指定，拡大．
b	現在のマウスポインタ位置へボールをドロップ．
c	rcssserver に接続．
l	現在のマウスポインタ位置で左チームにフリーキックを与える．
r	現在のマウスポインタ位置で右チームにフリーキックを与える．
s	試合開始 (kick off)．
m	mode を切り替える．
p	物体の現在位置をコンソールに出力する．
v	ボールの予測起動を表示する．
t	操作対象チームを切り替える．
数字,a,b	対応する背番号のプレイヤを現在のマウスポインタ位置へ移動する．0 または a は 10 番，b は 11 番を意味する．

起動オプション

rcssmonitor はホームディレクトリに `.rcssmnitor.conf` という設定ファイルを自動作成します．このファイルを編集するか，コマンドラインオプションでの直接指定によって，rcssmonitor 起動時の設定を変更することができます．ウインドウのサイズ，フォント，プレイヤの色などの基本的な設定変更はここで行います．

2.3.2 rcssserver の起動オプション

rcssserver は非常に多くのオプションを持っています．rcssserver はホームディレクトリに .rcssserver というディレクトリを作成し，更にその下に各種設定ファイルを作成します．これらのファイルを変更するか，コマンドラインオプションでの直接指定によって rcssserver 起動時の設定を変更することができます．

どの環境であっても，最低限以下の二つのファイルが生成されます．

- server.conf
- player.conf

server.conf に含まれるオプションは，server:: という名前空間に含まれており，player.conf に含まれるオプションは player:: という名前空間に含まれています．それぞれ，コマンドラインオプションで使用する場合は以下のように使用します．

```
$ rcssserver server::game_logging = 1 server::text_logging = 1 \
player::random_seed = 1208
```

これらオプションには，プログラムの挙動を変更するオプションだけでなく，シミュレーションの物理モデルや時間モデルを変更するパラメータも含まれています．最も良く利用するのは，ログファイルの保存に関するものでしょう．rcssserver が提供するオプション，パラメータの一覧を付録 A.2 に掲載しています．詳しくはそちらを参照してください．

Section 2.4
ログファイル

2.4.1 ログファイルの種類

シミュレーション実行後，rcssserver は以下の拡張子を持つ 2 種類のログファイルを生成します．

- .rcl (RoboCup Text Log)
  テキストファイル．主にサッカーエージェントが rcssserver へ送信したコマ

ンド文字列，審判の発したメッセージを記録する，エージェントのデバッグや動作確認のためのログファイル．rcssserver のオプションによっては，rcssserver 自身のデバッグ情報を含めることもある．

- `.rcg` (RoboCup Game Log)
  バイナリファイル．シミュレーションサイクルごとの物体の位置情報などを記録した，試合再生のためのログファイル．

デフォルトの設定では，これらのファイルは **rcsooccersim** コマンドを実行したディレクトリに，“日時+チーム名+スコア” というファイル名で保存されます．ログファイル名のフォーマットを変更したい，ログを保存するディレクトリパスを固定にしたい，またはログ自体を保損じたくない場合などは `server.conf` を編集し，該当オプションを変更します．

2.4.2 試合再生

`rcg` ファイルによる試合再生を行うためのプログラムとして，rcsslogplayer というプログラムがサッカーシミュレーション公式サイトで配布されています．rcsslogplayer を利用するには，公式サイトのリリースファイル一覧から “RCSSLogPlayer: Rel Candidate” を選択し，`rcsslogplayer-10.0.1.tar.gz` というファイルをダウンロードします．後は，他のパッケージと同様に以下のコマンドを順に実行するだけです．

```
$ tar xzvf rcsslogplayer-10.0.1.tar.gz
$ cd rcsslogplayer-10.0.1
$ ./configure --prefix=$HOME/rcss
$ make
$ make install
```

rcsslogplayer でログを再生するには，**rcsslogplay** というコマンドを，再生したい `rcg` ファイル名を指定して実行します．

```
$ rcsslogplay <ログファイル名>
```

すると，ビデオの操作パネルのようなウインドウと rcssmonitor が現れ，パネルを操作することで rcssmonitor 上に試合が再生されます．

rcsslogplayer では圧縮ファイルを使用できないので注意してください．また，環境によっては rcsslogplayer 自体のコンパイルに失敗することがあります．

2.4.3 サードパーティ製ログプレイヤ

公式のログプレイヤである rcsslogplayer はあまり使い勝手の良いものではありません．そのため，競技参加者が独自のログプレイヤを開発，公開していることがあります．以下に筆者も利用しているものを上げます．

- Soccer Viewer
  http://www.i.his.fukui-u.ac.jp/~shimora/RoboCup/
  福井大学の下羅弘樹氏によって開発された高機能ビューワです．C++で開発されており，Linux，FreeBSD などで動作します．

- SoccerScope
  http://ne.cs.uec.ac.jp/~newone/SoccerScope2003/
  東京大学 (電気通信大学) 竹内研究室によって開発された高機能ビューワです．Java で開発されているため，プラットフォームを選ばずに動作します．

- MagicBox
  http://wrighteagle.org/appdev/
  中国，科学技術大学が開発した三次元ビューワです．自動実況機能もあります．Windows 用です．プレゼンテーションには最も適しているでしょう．

- SoccerWindow
  http://sourceforge.jp/projects/rctools/
  筆者が既存の Windows 用ログプレイヤを改造して作成したビューワです．Windows 用です．

- soccerwindow2
  http://sourceforge.jp/projects/rctools/
  筆者が現在開発継続中の高機能ビューワです．Linux で動作確認していますが，将来的にあマルチプラットフォームなアプリケーションを目指しています．

これらのビューワプログラムは rcssmonitor の代替プログラムとして使用することもできます．図 2.1 から分かるとおり，rcssmonitor もクライアントプログラムとして動作します．よって，rcssserver との通信プロトコルを理解できれば，独自のモニタクライアントプログラムを使用できるのです．何より，現在公式に使用されている rcssmonitor 自体が，元々は参加者によって開発されたモニタプログラムです．

2.4.4 ログコンバータ

rcg ファイルを Flash に変換するコンバータとして robocup2flash[4] というものがあります．これを使えば，Flash プラグインがインストールされている Web ブラウザで試合を再生できます．

筆者の開発サイトでは rcg ファイルをテキストデータへと変換するツールプログラムなども公開しています．参考にしてみてください．

Section 2.5
rcssserver の仕様

2.5.1 クライアントプログラムの種類

rcssserver が受け付けるクライアントプログラムは，以下の 4 種類です．

- プレイヤエージェント
  フィールド上のプレイヤを制御する．フィールド上の情報を部分的にしか得られず，その情報はノイズを含む．
- コーチエージェント
  フィールド上の物体の完全な位置情報を得られる．チームに対してアドバイスを与えられる．プレイヤを直接的に制御することはできない．試合で使用できる．
- トレーナエージェント
  コーチの能力に加え，審判同様に試合を制御することもできる．試合では使用できない．オフラインコーチとも呼ばれ卯．
- モニタ

本書では，プレイヤエージェント，コーチエージェント，トレーナエージェントの 3 種をまとめてサッカーエージェントと呼びます．

[4] http://sourceforge.net/projects/robolog/

2.5.2 時間モデル

rcssserver は離散時間シミュレータです．定期的に状態を更新し，物体の位置はそのときに限り更新されます．rcssmonitor に表示される試合時間がこのシミュレーションに該当します．ただし，サイクル更新は定期的ですが，各エージェントとのメッセージ送受信はこのサイクルとは非同期に行われます．

通常，100 ミリ秒に一回，フィールドの状態が更新されることになっています．よって，理想的には，rcssmonitor は秒間 10 フレームで表示を更新していきます．rcssse4rver の時間モデルについては，6 章で本書のライブラリと合わせて解説します．

2.5.3 物理モデル

rcssserver の物理モデルは実世界とかなりの部分で異なっています．例えば，物体と地面との摩擦係数は考慮されておらず，速度減衰率というパラメータで置き換えられています．これは，実世界の忠実なシミュレータでは無く，あくまでマルチエージェントシミュレーション環境として適切にかつ軽快に動作するように rcssserver が設計されたためです．

rcssserver の物理モデルについては，**??**章でプレイヤエージェントの行動コマンドと合わせて解説します．

2.5.4 フィールドの座標系

rcssserver 内部で使用される座標系は，左手座標系です．rcssmonitor でフィールドを表示した場合，フィールド中央を原点とし，右が X 軸正方向，下が Y 軸正方向になります．角度の計算もそれに準じます．すなわち，フィールド中心から，右サイドのゴール方向が 0 度，真下の方向が 90 度，真上の方向が -90 度，そして左サイドのゴール方向が 180 度 となります．角度を扱う場合は，常に時計回りが正の方向になるように考えます．

図で表すと図 2.4 のようになります．フィールドの大きさは長さ $105m$，幅 $64m$ です．フィールド中央が原点のため，各コーナーの座標は図に示すよう値になります．

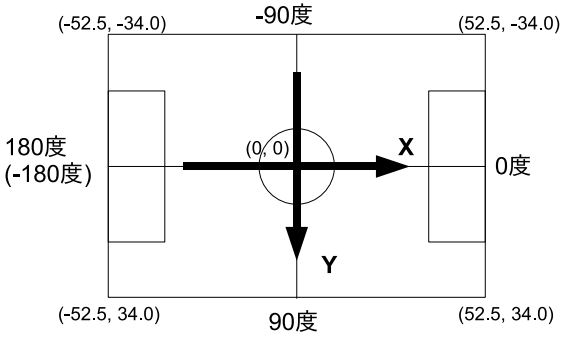


図 2.4 rcssserver の座標系

プレイヤの座標系

プレイヤエージェントのための座標系は rcsserver の座標系とは若干異なります．左サイドのチームは rcssserver 内部の座標系と全く同じです．しかし，右サイドのチームはそれらを全て反転して使用することになっています．すなわち，フィールド中央を原点とし，左が X 軸正方向，上が Y 軸正方向になり，角度も反転されます．混乱しやすいですが，“敵ゴール方向が X 軸正方向”，“敵ゴールに向いて右手側が角度の正方向” と覚えておけば良いでしょう．

コーチとトレーナの座標系

コーチエージェントとトレーナエージェントは常に rcssserver と同じ左手座標系を使用します．

2.5.5 審判

サッカーの試合を行うための審判機能は rcssserver に組み込まれています．曖昧な判定は無く，1 ミリも見逃すことはありません．ただし，悪質なプレイに対するファウルの判定はできません．

審判は自動化されており，ボールやプレイヤの位置，状態に応じて適切にプレイモードを変更し，ボールやプレイヤを再配置し，全てのエージェントに新しいプレイモードを通知します．ルールは，実際の人間のサッカーのルールに可能な

限り近くなるように設定されています．rcssserver で使用されるプレイモードの一覧を付録 A.1 に掲載しています．

ほとんどの場合，人間は試合開始のキックオフを行えば良いだけとなっています．しかし，組み込みの審判では判断できない状況が発生すれば人間が介入せざるを得ません．例えば，公式競技において以下のような状況が発生した場合は人間が rcssmonitor を介して試合に介入しても良いことになっています．

- 複数のプレイヤがボールに群がり，ボールが動かせない．
- ボールが一定位置からほとんど動かない
- 両チームがフリーキックの失敗を繰り返す，キーパがキャッチを繰り返すなどして試合が進まない

他にも，非紳士的であると認められる行為を行った場合は何らかの処分が下されます．

Section 2.6
マニュアルの入手

サッカーシミュレータの仕様のほとんどは公式マニュアルに掲載されています．シミュレータのプログラムパッケージと同様，マニュアルも公式サイトのリリースファイルに含まれています．公式マニュアルは英語で書かれていますが，これを日本語訳したものを筆者の開発サイトで公開しています．

http://sourceforge.jp/projects/rctools/

ただし，その内容はバージョン 7 当時のシミュレータのマニュアルに基づいており，最新のシミュレータの仕様に沿ったものではありません．基本的な物理モデルなどの参考にとどめてください．

第3章

チーム開発

本章では，チーム開発に必要となるライブラリ利用のための知識や，チームとして協調的な動作を実現するために必要なノウハウを解説します．rcssserver の物理モデルや個々のプレイヤの細かい動作制御に関して詳しく知りたい場合は，4章を参照してください．

Section 3.1
開発時の心得

チームの作り方を説明する前に，筆者が心がけている開発時における心得のようなものを紹介しようと思います．

- 恵まれ過ぎた環境を使わない．

  CPU が遅い，ネットワークが不安定といった貧弱な環境での動作に耐えられるようにプログラムを作りましょう．計算資源を浪費するプログラムを作ってしまうと，他の環境では重すぎてまともに動かないという事態になりかねません．

- 自分のプログラムを信じない．

  何かを実装するたびにしつこいくらいテストしましょう．発生頻度が稀なバグであっても，結果に致命的な影響を及ぼしかねません．もし，“よくわからないけど動いているからいいや” という考え方をするようであれば，RoboCup やマルチエージェントなどと言う以前にプログラミングという作業自体への適性に問題があります．

- ビジュアルデバッガを使う．

  テストとも関連しますが，サッカーシミュレーションにおいては，自分が実装した内容が本当に機能しているのかどうかの確認作業が非常に繁雑なものになります．この確認作業を効率化するにはビジュアルデバッガの利用が必須です．ビジュアルデバッガについては 3.8 節を参照してください．

- コーディングスタイルを守る．

  勢いで書いたプログラムは後で自分が読んでも理解できません．一定の書き方を守り，書式を整えることでプログラムの理解しやすさは格段に向上します．特に，関数名や変数名の命名規則とインデント幅の明文化は重要です．

- コメントを書く．

  逐一コメントを書く必要はありませんが，最低限 Doxygen[2] スタイルのコメントを書いておけば後々自分が助かります．

- バージョン管理する．

  CVS[18] や Subversion[13, 14] などのバージョン管理システムを使いましょう．本書では Subversion をお薦めします．

- 試合直前に変更を加えない．

  変更内容をテストする充分な時間が無いときに思いつきでプログラムに変更を加えても良い結果が得られることはまずありません．逆に，バグを仕込んで悪化するだけです．実際，直前の作り込みで自滅したチームは過去に多数存在します．深刻な不具合を修正する以外，試合直前に変更を加えるべきではありません．

普通のプログラム開発における心構えと何ら変わりません．しかし，これらがきちんと実践できているか否かで，できあがるチームのパフォーマンスには大きな違いが現れることでしょう．

Section 3.2
開発環境の準備

ここではサンプルチームを実行するまでの手順を解説します．サッカーシミュレータ一式がインストールできた環境ならば，ほとんど問題は無いはずです．

3.2.1 必須ライブラリ，ツール

本書で使用するプログラムでは以下のライブラリやツールプログラムを必要とします．まずはこれらがインストールされているか確認してください．

- gcc 3.0 以降

  http://gcc.gnu.org/

  本書のライブラリでは C++テンプレートを多用しています．また，後述の Boost を正しく導入，利用するためにもなるべく標準準拠度の高いコンパイラを使用してください．gcc 以外のコンパイラ，gcc でも 3.0 以前のバージョンを使用する場合は動作の保証ができません．

- GNU Autotools

  http://www.gnu.org/software/autoconf/

  http://www.gnu.org/software/automake/

  http://www.gnu.org/software/libtool/

  所謂 “configure” を利用するためのツールです．Autoconf，Automake，Libtool をセットにして使用し，プログラムをコンパイルするための環境のチェックや設定を自動化してくれます．大抵の Linux ディストリビューションなら標準でインストールされているはずです．

- Boost 1.32 以降

  http://boost.org/

  準標準と言われる C++テンプレートライブラリです．大抵の Linux ディストリビューションには専用のインストール用パッケージが用意されています．

- wxGTK 2.6.1 以降

  http://www.wxwidgets.org/

  本書で紹介するビジュアルデバッガ，soccerwindow2 を使用するために必須のライブラリです．soccerwindow2 を使用しないのであれば必要ありませんが，導入を強く推奨します．

- Doxygen

  http://www.stack.nl/ dimitri/doxygen/

  プログラムソースファイル中のコメント文からドキュメントを自動生成してくれるツールです．必須ではありませんが，本書のプログラムライブラリのリファレンスを生成するために必要になります．リファレンスが無ければ開発効率が著しく低下すると予想されるので，是非導入してください．

3.2.2 プログラム構成

本書で提供するプログラムは以下の 3 つのパッケージに分かれています．

- librcsc

  サッカーシミュレーションのチームプログラムやツールプログラムの開発に使用する基本ライブラリです．以下の 2 つのプログラムで使用します．

- soccerwindow2

  rcssmonitor 互換のモニタプログラムです．単体でログプレイヤとして動作するだけでなく，高機能なビジュアルデバッガとしても使用できます．チーム開発が本格的になってくれば必ず使用しましょう．このツールを使わなければ，強いチームを作ることは不可能です．

- agent2d

  チームとして最小限動く状態にまとめたプログラムソースです．チーム開発を開始する際のテンプレートとして使用します．チーム開発時にはこのパッケージ内のソースを編集します．

これらのプログラムパッケージは本書付録の CD-ROM に納められています．また，筆者の開発サイトでも公開し，開発を継続しています．

http://rctools.sourceforge.jp/

パッケージ内部のディレクトリ構成は以下のようになっています.

```
librcsc-x.x.x/       : configure スクリプトなど
 +-- config/         : Autotools の設定ファイル
 +-- example/        : librcsc を利用したサンプルプログラム
 +-- rcsc/           : ライブラリのディレクトリ
 |    +-- action/    : プレイヤエージェントの基本スキル
 |    +-- client/    : 基本クライアントライブラリ
 |    +-- coach/     : コーチエージェント関連ライブラリ
 |    +-- geom/      : 幾何計算ライブラリ
 |    +-- gz/        : gzip ストリーム
 |    +-- monitor/   : モニタクライアント用ライブラリ
 |    +-- net/       : 通信ライブラリ
 |    +-- param/     : シミュレータのパラメータライブラリ
 |    +-- player/    : プレイヤエージェント関連ライブラリ
 |    +-- rcg/       : ゲームログ解析用ライブラリ
 |    +-- trainer/   : トレーナエージェント関連ライブラリ
 |    +-- util/      : その他のユーティリティー
 +-- src/            : librcsc を利用したツールプログラム
```

```
soccerwindow2-x.x.x/ : configure スクリプトなど
 +-- config/         : Autotools の設定ファイル
 +-- src/            : ソースファイル一式
      +-- xpm/       : アイコンなどの画像ファイル一式
```

```
agent2d-x.x.x/        : configure スクリプトなど
 +-- config/         : Autotools の設定ファイル
 +-- src/            : ソースファイル一式
```

3.2.3 コンパイル，インストール

各プログラムパッケージは GNU Autotools を使用してパッケージングされています．いずれも，configure スクリプトを実行し make という流れでコンパイル

を行います．'--help' というオプションを付けて configure スクリプトを実行すると，利用可能なオプションが表示されます．

librcsc のインストール

librcsc を入手して，展開してください．後は通常の手順でインストールするだけです．インストール先を'--prefix' というオプションで指定できます．指定が無ければ /usr/local 以下にインストールされます．/usr/local でも問題はありませんが，サッカーシミュレータと同じ場所にインストールしておいた方が管理が楽です．

```
$ tar xzvf librcsc-x.x.x.tar.gz
$ cd librcsc-x.x.x
$ ./configure --prefix=$HOME/rcss
$ make
$ make install
```

これで，ヘッダファイル，ライブラリファイル，そしてツールプログラムがインストールされます．

soccerwindow2 のインストール

soccerwindow2 を入手して，展開してください．librcsc を標準以外の場所へインストールした場合は fileconfigure の'--with-librcsc' というオプションで librcsc へのパスを指定します．

```
$ tar xzvf soccerwindow2-x.x.x.tar.gz
$ cd soccerwindow2-x.x.x
$ ./configure --prefix=$HOME/rcss --with-librcsc=$HOME/rcss
$ make
$ make install
```

これで，実行ファイル soccerwindow2 と，簡易起動スクリプト sswindow2 がインストールされます．

soccerwindow2 を rcssmonitor の代わりのデフォルトモニタとする場合は，環境変数 RCSSMONITOR を変更します．ホームディレクトリの .bash_profile に以下の内容を追記してください．

```
export RCSSMONITOR=sswindow2
```

soccerwindow2 の起動オプションを追加した独自のスクリプトを指定することももちろん可能です．'--help' オプションをつけて soccerwindow2 を起動すると，soccerwindow2 が受け付けるオプションの一覧が表示されます．

サンプルチームのコンパイル

agent2d を入手して，展開してください．soccerwindow2 と同様，必要に応じて'--with-librcsc' というオプションを指定します．チームプログラムはコンパイルのみでインストールされないので，'–prefix' オプションは必要ありません．

```
$ tar xzvf agent2d-x.x.x.tar.gz
$ cd agent2d-x.x.x
$ ./configure  --with-librcsc=$HOME/rcss
$ make
```

アンインストール

librcsc と soccerwindow2 をアンインストールする場合はパッケージのディレクトリで以下のコマンドを実行してください

```
$ make uninstall
```

3.2.4 サンプルチームの実行

agent2d のコンパイルに成功すれば，チームが実行できる状態になっています．別のターミナルで先にシミュレータを起動しておき，src ディレクトリに移動して，チーム起動スクリプト start.sh を実行します．

```
$ rcsoccersim
```

```
$ cd agent2d-x.x.x/src
$ ./start.sh
```

プレイヤがモニタ上に現れ，初期フォーメーションを形成すれば成功です．

3.2.5 Doxygen によるリファレンス生成

librcsc のソースファイルには Doxygen 形式でのコメントがつけられています．librcsc のパッケージディレクトリに `Doxyfile` というファイルが存在することを確認してください．確認できたら `doxygen` コマンドを実行します．

```
$ ls Doxyfile
Doxyfile
$ doxygen
Warning: The selected output language "japanese" has not been updated
since release 1.3.9.  As a result some sentences may appear in English.

Searching for include files...
Searching for example files...
Searching for images...
Searching for dot files...
...
...
...
Generating namespace member index...
Generating page index...
Generating graph info page...
Generating graphical class hierarchy...
$ ls doc
html
```

`doc` というディレクトリが作成され，更にそれ以下に `html` というディレクトリが作成されます．`html` ディレクトリ内の `index.html` を Web ブラウザで表示させると，ライブラリのリファレンスを閲覧できます．

3.2.6 ソースファイルの追加，削除

agent2d では，GNU Autotools を使用した `Makefile` の自動生成を行っています．独自のヘッダファイルやソースファイルを追加する場合の手順は以下のようになります．

1. ソースディレクトリの `Makefile.am` を編集．
   既存のヘッダファイルとソースファイルがリストされているので，追加したファイルをリストに追加します．ここでは，`new_file.h` と `new_file.cpp` という二つのファイルを追加するものとします．

```
sample_player_SOURCES = \
  sample_player.cpp \
  new_file.cpp

sample_palyer_HEADERS = \
  sample_player.h \
  new_file.h
```

バックスラッシュの有無に注意してください．リストの最後のファイルの後にバックスラッシュをつけると，正しい Makefile が生成できません．

2. bootstrap の実行．
パッケージのトップディレクトリで以下のように bootstrap スクリプトを実行します．

```
$ ./bootstrap
```

3. 再度，configure と make を実行します．

```
$ ./configure --with-librcsc=$HOME/rcss
$ make
```

以前のビルド過程で生成されたキャッシュファイルなどが原因で configure や make が上手くいかないことがあります．そのような場合は，以下のように distclean することで一旦全ての生成ファ イルを削除してください．

```
$ make distclean
```

distclean が完了したら，再度 configure からやり直してください．

ソースファイルをパッケージから削除したい場合は，追加とは逆にリストから該当ファイルを消去してください．その他の手順は全く同じです．

Makefile の生成をより細かく制御したい場合は，GNU Autotools のマニュアルを参照するか，独自に Makefile を作成するなどしてください．

Section 3.3
フォーメーションの変更

サンプルチームのプレイヤエージェントは，ボールを持っていないときにはフォーメーションを維持するためのポジショニング動作を実行します．まずはこのフォーメーションを変更してみましょう．

3.3.1 フォーメーションの基礎

個々で説明するフォーメーションは， formation.conf というファイルに記述されたパラメータで管理されています．このファイルを編集することでフォーメーションを変更することができます．

formation.conf の中を覗いてみてください．以下のような数値が並んでいるのを見つけることができるでしょう．

```
# Formation 2 = FT_433_offensive
2
-50.0 -16.5 -21.0 -15.0 -16.5   0.0   0.0  -3.0  15.0  18.0  18.0  # X_pos
  0.0  10.0   0.0  0.0  -10.0 -11.0  11.0   0.5  -0.5  19.0 -19.0  # Y_pos
    1     4     3     2     4     6     6     5     8     7     7   # P_type
  0.0   0.1   0.7  0.65   0.7  0.65   0.7   0.5  0.6                # X_attr
  0.0   0.1   0.2   0.4 0.25  0.3  0.25   0.3  0.25               # Y_attr
    0     1     1     1     0     0     0     0     0           # Behind_ball
  0.0 -50.5 -42.0 -47.0 -45.0 -36.0 -36.0  -2.0  -2.0               # X_min
  0.0 -30.0   0.0   2.0   2.0  42.0  42.0   44.0 44.0               # X_max
```

このパラメータセットを使用してチームを起動するには，以下のように--sbspというオプションを付けてチームの起動スクリプトを実行してください．

```
$ cd agent2d-x.x.x/src
$ ./start.sh --sbsp
```

フォーメーションパラメータのフォーマットは，UvA Trilearn[15] というチームが採用しているものと全く同じにしています．このようなパラメータセットでフォーメーションを管理する手法は SBSP(Situation Based Strategic Positioning)[9] と呼ばれています．SBSP は 2000 年の RoboCup 世界大会で優勝した FC Portugal[4] というチームが提案した手法で，パラメータとシンプルなルールとを組み合せて

フォーメーションを形成し，更に，状況に応じてパラメータセットを切り替えることで柔軟な戦略変更を可能にしています．

パラメータから移動位置を算出する原理は非常に簡単です．プレイヤエージェントの移動位置座標は以下の式で求められます．

$$PlayerPosX = X\_Pos + BallPosX \times X\_attr$$

$$PlayerPosY = Y\_Pos + BallPosY \times Y\_attr$$

パラメータセット中の X_Pos と Y_Pos の位置を基準とし，ボール位置がフィールド中央 (原点) から移動するに従い，それに引き寄せられるようにして移動位置座標が決定します．ボールにどれだけ引き寄れるかは X_attr と Y_attr によって決定されます．

パラメータの意味を順に説明します．

X_Pos と Y_Pos の行は，上式のパラメータとしてそのまま使用されます．P_type の行はプレイヤの役割 (キーパ，ディフェンダ，フォワードなど) の ID を意味します．これら 3 種は各プレイヤごとに設定するパラメータで，一行あたりのパラメータ数は 11 個です．

以降のパラメータは，プレイヤごとでは無く，役割ごとに設定するパラメータです．ここでは，役割の種類はは合計 9 タイプとしているので，1 行あたりのパラメータ数も 9 個となります．X_attr と Y_attr の行は，上式で使用されているパラメータです．Behind_ball は，ボールよりも後ろの位置を維持するかどうかを決定します．通常，ディフェンダはボールよりも後ろにいないと危険なため，上式を無視してボールよりも後ろに下がっておく方が自然です．Behind_ball を有効 (=1) にしておけば，移動位置座標はボールよりも後ろに補正されます．X_min と X_max はプレイヤが移動する範囲を制限します．移動位置の計算方法自体が単純なため，X_attr の値によっては移動位置がフィールドの外になることもありますが，X_min と X_max によって強制的な制限をかけることができます．

動作原理が分かれば後は簡単です．変更が大きく影響を及ぼすのは，X_Pos，Y_Pos，X_attr，Y_attr です．まずはこれらを修正してみてください．ファイルを保存したらチームを起動し直して確認してみましょう．思いどおりにフォーメーションが変更されましたか？

3.3.2 より高度なフォーメーション作成

SBSP の長所は何と言ってもその手軽さです．実装が容易で，パラメータ数も人間が把握できる程度に抑えられています．原理が簡単な割に効果が非常に高い

ため，サッカーシミュレーションリーグ参加チームには広く利用されています．

しかしながら，SBSP は単純で扱いやすい半面，調整作業に苦労が伴います．経験と勘が要求されるだけでなく，動作確認のためのテストに膨大な時間を費さざるを得ません．また，状況に応じた配置を実現するにはそれだけの数のパラメータセットを用意し，フォーメーションを使い分けなければなりません．現実的には，管理できるパラメータセットの数は 3 つから 4 つ程度が限度です．

このように，SBSP では柔軟性と管理コストにおいて限界が見えています．そこで，本書ではより柔軟にフォーメーションを作成するために soccerwindow2 のフォーメーション編集機能を利用する方法を紹介します．詳しくは 3.20 節で解説します．

Section 3.4
librcsc の利用

ここでは，チーム開発時に最低限必要となる librcsc の利用方法を解説します．

3.4.1 名前空間 rcsc

librcsc に含まれる全ての変数，関数，クラスは名前空間 `rcsc` に含まれています．チームプログラムでこれらを利用するには，以下のようにスコープ解決演算子 (::) を使用してください．

```
rcsc::Vector2D first_vel( 1.0, 0.0 );
rcsc::Vector2D travel_vec
  = rcsc::inertia_final_travel( first_vel,
                          rcsc::ServerParam::i().ballDecay() );
```

または，以下のように `using` 構文を使用するとスコープ解決演算子を省略できます．

```
using namespace rcsc;
Vector2D first_vel( 1.0, 0.0 );
Vector2D travel_vec
  = inertia_final_travel( first_vel,
                          ServerParam::i().ballDecay() );
```

タイプ数が少なくなるので一見良さそうですが，この書き方は非推奨です．名前の衝突を回避するためには，面倒でも毎回スコープ解決演算子で rcsc を指定する方が安全です．

3.4.2 よく使うデータ型

使用頻度の高い，librcsc 独自のデータ型を説明します．

SideID

値として LEFT, RIGHT, NEUTRAL を定義した列挙型です．主に，プレイヤが屬するチームを確認するために使用します．

HeteroID

ヘテロジニアスプレイヤの ID を定義した列挙型です．値として，Hetero_Unknown, Hetero_Default, Hetero_1 . . . Hetero_6 を取ります．ヘテロジニアスプレイヤについては 3.4.4 節や 5 章を参照してください．

GameTime

シミュレーション時間を格納するためのクラスです．内部に long 型変数を二つ持ち，一つ目は通常のシミュレーションサイクル，二つ目はシミュレーションサイクルが止まっている間の隠れたサイクルのカウントを保持します．例えば，キックオフ前やオフサイド直後などはモニタ上のシミュレーションサイクルのカウントは進んでいませんが，rcssserver 内部では通常どおり時間が進んでいます．これが隠れたサイクルです．通常の試合進行時では，隠れたサイクルの値は常に 0 になります．GameTime クラスの持つ情報を参照するには，以下のメンバ関数を使用します．

GameTime メンバ
const long & cycle() const; 通常のサイクルの値を返す.
const long & stopped() const; 隠れたサイクルの値を返す．通常の試合進行時には常に 0.

PlayMode

rcssserver で定義されているプレイモードと同じ順序，同じ値で定義された列挙型です．しかし，通常は以下の GameMode を使用します.

GameMode

rcssserver のプレイモードを管理するためのクラスです．このクラス内部では，PlayMode が GameMode::Type と SideID のペアに変換されて格納されます．これによって，プレイモードのタイプとプレイモードのサイドの情報を独立して扱うことができます．GameMode::Type は，GameMode クラス内部で定義されている列挙型です．GameMode クラスは rcssserver で定義されているプレイモード文字列を解析して適切なモードに更新する機能も持っており，審判から配信される情報の解析に使用できます．GameMode クラスの持つ情報を参照するには，以下のメンバ関数を使用します.

GameMode メンバ
GameMode::Type type() const; プレイモードのタイプを返す.
SideID type() const; プレイモードのサイドを返す.
int getScoreLeft() const; 左チームの得点を返す.
int getScoreRight() const; 右チームの得点を返す.

`bool isServerCycleStoppedMode() const;` シミュレーションサイクルが停止するタイプのプレイモードであれば `true` を返す.
`bool isGameEndMode() const;` 試合が終了していれば `true` を返す.
`bool isPenaltyKickMode() const;` ペナルティキック中なら `true` を返す.
`bool isOurSetPlay( const SideID ourside ) const;` 味方のセットプレイ中なら `true` を返す.
`PlayMode getPlayMode() const;` `rcssserver` 上の `PlayMode` に変換された値を返す.

ViewWidth

プレイヤエージェントの視野角を保持するためのクラスです．rcssserver 上でプレイヤが取りうる視野角は 3 タイプしか存在しないため，数値では無く列挙型，`ViewWidth::Type` で視野角を管理しています．`ViewWidth::Type` は `ViewWidth` クラス内部で定義されており，値として `NARROW`，`NORMAL`，`WIDE` を取ります．`ViewWidth` クラスが持つ情報を参照するには、以下のメンバ関数を使用します．

`ViewWidth` メンバ
`ViewWidth::Type type() const;` 視野角のタイプ (`NARROW`,`NORMAL`,`WIDE`) を返す.
`double getWidth() const;` 視野角の大きさを度数で返す.

3.4.3 幾何計算クラスライブラリ

`geom` ディレクトリ以下には幾何計算用のクラスライブラリが収められています．その中でも，`AngleDeg` と `Vector2D` は極めて頻繁に利用するため，全てのメンバ関数を把握しておきたいところです．

AngleDeg

angle_deg.h で宣言されている，平面上で-180 度から 180 度の度数を扱うためのクラスです．コンストラクタと代入演算子では double 型の値を使用して初期化できます．その際，値は自動的に $[-180, 180]$ に正規化されて保持されます．

AngleDeg の使用例を以下に示します．

```
#include <rcsc/geom/angle_deg.h>
int main() {
  rcsc::AngleDeg angle1( 100.0 ); // 通常のコンストラクタ
  rcsc::AngleDeg angle2 = 90.0;   // 代入演算子でも初期化可能
  angle1 = 200.0;      // double 型の直接代入も可能
                       // この場合更に正規化も行われ,
                       // angle1 == -160 度となる.
  angle1 = angle1 + angle2;  // 加算，減算の演算子が使用可能.
  angle1 = angle1 - angle2;
  angle1 += angle2;   // 複合代入演算子が使用可能
  std::cout << angle1 << std::endl; // ストリームに直接出力
  double deg = angle1.degree();  // 度数値を取得
  dobule rad = angle1.radian();  // ラジアン値を取得
  double cosine = angle1.cos();  // コサイン値を取得
  // static メンバ関数で二等分角を取得
  rcsc::AngleDeg bisect_angle
    = rcsc::AngleDeg::bisect_angle( angle1, angle2 );
  // ある方向を基準とした左右を判定
  if ( bisect_angle.isLeftOf( angle2 ) )
    std::cout << "bisect is left side of angle2" << std::endl;
}
```

Vector2D

vector_2d.h で宣言されている，平面上のベクトルを扱うためのクラスです．厳密に言えば，ただの座標値である点と方向を持つベクトルは区別して扱うべきですが，librcsc ではいずれも Vector2D で処理することにしています．

Vector2D の使用例を以下に示します．

```
#include <rcsc/geom/vector_2d.h>
int main() {
  rcsc::Vector2D point1( 0.0, 0.0 ); // 通常のコンストラクタ
  // 極座標で初期化
  rcsc::Vector2D point2( rcsc::Vector2D::POLE, 10.0, 90.0 );
  point1 = point1 + point2;  // 加算，減産の演算子が使用可能.
  point1 = point1 - point2;
  point1 += point2;   // 複合代入演算子が使用可能
  point1 *= 2.0;  // スカラー倍
  double len = point1.r();  // ベクトルの長さ (半径) を得る
  rcsc::AngleDeg dir = point1.th();  // ベクトルの方向を得る
  double naiseki = point1.innerProduct( point2 ); // 内積
  double gaiseki = point1.outerProduct( point2 ); // 外積
  point1.normalize();  // 長さ 1 に正規化
  point1.setLength( 10.0 ); // 方向は変えず，長さを 10 に
  point1.rotate( 20.0 ); // 長さは変えず，方向を 20 度回転
  std::cout << point1 << point2 << std::endl; // ストリームへ出力
}
```

その他のクラス

AngleDeg と Vector2D 以外にも，幾何計算を容易にする以下のようなクラスを用意しています.

クラス名	説明
Line2D	無限の長さを持つ直線を扱うクラスです．他の直線との交点や，指定の点を通る垂線を求めることができます.
Ray2D	ひとつの端点を始点としてある方向へと無限に伸びる半直線を扱うクラスす．直線との交点を求めることができます.
Segment2D	二点を結ぶまっすぐな線分を扱うクラスです．他の直線や線分との交点，垂直二等分線を求めることができます.

Circle2D	円を扱うクラスです．内外判定や直線との交点を求めることができます．
Rect2D	矩形を扱うクラスです．内外判定や直線との交点を求めることができます．
Size2D	長さと幅を扱うクラスです．矩形のサイズを指定するために使用します．
Sector2D	扇型を扱うクラスです．内外判定ができます．
Triangle2D	三角形を扱うクラスです．内外判定や，内心，外心などを求めることができます．

3.4.4 rcssserver のパラメータ

rcssserver の時間モデル，物理モデルに関するパラメータは ServerParam，PlayerParam，PlayerType というクラスで管理されています．

ServerParam

rcssserver の設定パラメータのほぼ全てを格納しています．そのため，ServerParam のメンバ変数は膨大な数に及んでおり，プレイヤの意志決定には全く関わりの無いパラメータも多く含まれています．これらの全てを把握する必要はありません．利用する機会が多いのは，フィールドのサイズやボールの設定パラメータでしょう．

Serverparam では Singleton パターン [3] を使用しているため，プログラム中のどこからでも使用できます．Singleton のインスタンスにアクセスするには，static メンバ変数 i() を使用します．ServerParam の使用例を以下に示します．

```
#include <rcsc/geom/rect_2d.h>
#include <rcsc/param/server_param.h>
...
  // フィールドの矩形を作成
  rcsc::Rect2D pitch( -rcsc::ServerParam::i().pitchHalfLength(),
                      -rcsc::ServerParam::i().pitchHalfWidth(),
                      rcsc::ServerParam::i().pitchLength(),
                      rcsc::ServerParam::i().pitchWidth() );
  // 自陣のペナルティエリアの矩形を作成
  rcsc::Rect2D our_penalty
    ( -rcsc::ServerParam::i().pitchHalfLength(),
      -rcsc::ServerParam::i().penaltyAreaHalfWidth(),
      rcsc::ServerParam::i().penaltyAreaLength(),
      rcsc::ServerParam::i().penaltyAreaWidth() );
  // ボールの半径を得る
  double ball_size = rcsc::ServerParam::i().ballSize();
  // ボールの最大スピードを得る
  double ball_speed_max = rcsc::ServerParam::i().ballSpeedMax();
...
```

rcssserver のパラメータ一覧は付録 A.2 にも掲載しているので，そちらも参考にしてください．

PlayerParam

rcssserver は，その起動時に身体能力パラメータが異なる 7 種類のプレイヤタイプを生成します．これらはヘテロジニアスプレイヤと呼ばれ，例えば，加速性能が高い代わりにスタミナの回復量が小さいなどのトレードオフに基づいてパラメータが設定されます．PlayerParam はこのヘテロジニアスプレイヤの生成に関するパラメータを格納しています．PlayerParam は rcssserver のためのパラメータセットであるため，サッカーエージェントの開発において使用する機会はまず無いでしょう．

PlayerType

PlayerType クラスは，個々のヘテロジニアスプレイヤのパラメータを格納します．プレイヤエージェントが自分自身の動作の予測を行うときには必ず参照する情報です．含まれるパラメータは全て把握しておきましょう．以下のメンバ関数で各パラメータを参照できます．

PlayerType メンバ
`HeteroID id();` ヘテロジニアスプレイヤのタイプ識別 ID を返す.
`const double & playerSpeedMax();` スピードの最大値を返す．プレイヤのスピードはこの値以下に制限される．ただし，ノイズによってこの値を越えることはある.
`const double & staminaIncMax();` 1 サイクルのスタミナ回復量の最大値を返す.
`const double & playerDecay();` 速度減衰率を返す.
`const double & inertiaMoment();` 慣性モーメント値を返す．turn コマンドに影響する.
`const double & dashPowerRate();` dash コマンドの効果率を返す.
`const double & playersize();` プレイヤの半径を返す.
`const double & kickableMargin();` キック可能領域のマージンを返す.
`const double & kickRand();` キックまたはタックル実行時にボールに加わるノイズ率を返す.
`const double & extraStamina();` スタミナ値 0 のときに追加で利用できるスタミナ量を返す..
`const double & effortMax();` effort の最大値を返す..
`const double & effortMin();` effort の最小値を返す..
`const double & realSpeedMax();` `const double & realSpeedMax2();`

この PlayerType に含まれる能力で到達できる最大スピードを返す．playerSpeedMax() の値よりも小さくなることがある． realSpeedMax2() は realSpeedMax() を二乗した値．

パラメータの値を得る以外にもいくつかのメンバ関数を用意しています．詳細はソースファイルか Doxygen で生成されるリファレンスを参照してください．

各パラメータの意味，及ぼす影響については章を参照してください．ヘテロジニアスプレイヤの詳細は 5 章でも解説しています．

3.4.5 PlayerAgent

サンプルエージェントは SampleAgent クラスで実装されており，これは PlayerAgent クラスからの public 派生クラスとなっています．そのため，SampleAgent からは PlayeAgent の protected と public なインタフェースにアクセス可能です．

プレイヤエージェントプログラムから rcssserver への情報の送信は，全て PlayerAgent クラスを介して行われます．rcssserver へ行動のコマンドを送信することで初めて，プレイヤエージェントが rcssserver 上で動作することができます．行動コマンドについての詳細は**??**節や 6.2 節を参照してください．

次節で PlayerAgent が持つ内部モデルの参照方法について説明します．

Section 3.5
内部モデルの参照

3.5.1 WorldModel

librcsc では，プレイヤエージェントの内部モデルを管理する WorldModel というクラスを用意しています．WorldModel は PlayerAgent クラスのメンバ変数として保持されます．PlayerAgent やその派生クラスがフィールドの情報へアクセスする場合，この WorldModel が全ての窓口となります．WorldModel のメンバ変数として，プレイヤエージェント自身の情報を管理する SelfObjec クラス，ボール情報を管理する BallObject クラス，他のプレイヤを管理する PlayerObject ク

ラスのコンテナ，が含まれています．その他，現在時刻，推定オフサイドライン，ボール所有者判定情報を格納する InterceptTable クラスなども含まれています．

3.5.2 SelfObject

プレイヤエージェント自身の情報を管理するクラスです．SelfObject は以下のようなメンバ関数を持っています．

SelfObject メンバ
int unum() const; 背番号
bool goalie() const; キーパか否か
const PlayerType & playerType() const; ヘテロジニアスプレイヤのパラメータ．
cons Vector2D & pos() const; 位置座標．
const Vector2D & vel() const; 速度．
const AngleDeg & body() const; 体の絶対方向．
const AngleDeg & neck() const; 体の方向に対する首の相対角度．
const AngleDeg & face() const; 首の絶対方向．
const ViewWidth & viewWidth() const; 現在の視野角の大きさ．
const GameTime & catchTime() const; 最後にボールをキャッチした時間．
const double & stamina() const; スタミナ値．
const double & effort() const; dash コマンドの効果に関連するスタミナ値．
bool isKickable() const;

ボールをキック可能か否か.
`const double & kickRate() const` kick コマンドの効果率．ボールとの位置関係と `playerType()` のパラメータによって決定される.
`const double & dashRate() const` dash コマンドの効果率．`effort()` と `playerType()` のパラメータによって決定される.
`const double & tackleProbability()` 現在の推定タックル成功確率．`[0,1]` の実数値.

他にもいくつかのメンバ関数があります．詳しくは Doxygen によるリファレンスなどを参照してください.

3.5.3 BallObject

観測されたボールの情報を管理するクラスです．`BallObject` は以下のようなメンバ関数を持っています.

`BallObject` メンバ
`const Vector2D & pos() const;` 位置座標.
`int posCount() const;` 最後に観測してからの経過サイクル．位置の信頼性情報として使う.
`const Vector2D & rpos() const;` `SelfObject` からの相対位置座標.
`const Vector2D & vel() const;` 速度.
`int velCount() const;` 最後に速度を観測してからの経過サイクル．速度の信頼性情報として使う.
`const double & distFromSelf() const;` `SelfObject` からの距離.
`const double & distFromSelf() const;` `SelfObject` からの絶対方向.

`Vector2D inertiaTravel( int cycle ) const;` 追加加速度が無い場合，cycle サイクルによる総移動ベクトル
`Vector2D inertiaPoint( int cycle ) const;` 追加加速度が無い場合，cycle サイクル後の位置.
`Vector2D inertiaFinalTravel() const;` 追加加速度が無い場合，速度が 0 になるまでの総移動ベクトル
`Vector2D inertiaFinalPoint() const;` 追加加速度が無い場合，速度が 0 になったときの位置座標.

他にもいくつかのメンバ関数があります．詳しくは Doxygen によるリファレンスなどを参照してください．

3.5.4 PlayerObject

観測された他のプレイヤを管理するクラスです．WorldModel クラス内では STL コンテナによって保持されています．

PlayerObject は以下のようなメンバ関数を持っています．

PlayerObject メンバ
`int unum() const;` 背番号を返す．背番号が不明の場合は-1 を返す.
`bool goalie() const;` キーパか否か
`const Vector2D & pos() const;` 位置座標.

`int posCount() const;`
最後に観測してからの経過サイクル．位置の信頼性情報として使う.
`const Vector2D & rpos() const;`
SelfObject からの相対位置座標.
`const Vector2D & vel() const;`
速度.

`int velCount() const;`

最後に速度を観測してからの経過サイクル．速度の信頼性情報として使う．
`const double & distFromSelf() const;` `SelfObject` からの距離．
`const double & distFromSelf() const;` `SelfObject` からの絶対方向．
`const double & distFromBall() const;` `BallObject` からの距離．
`const AngleDeg & body() const;` 体の絶対方向．
`const AngleDeg & face() const;` 首の絶対方向．
`int faceCount() const;` 最後に体と首の向きを観測してからの経過サイクル．方向の信頼性情報として使う．

他にもいくつかのメンバ関数があります．詳しくは Doxygen によるリファレンスなどを参照してください．

3.5.5 PlayerObject コンテナ

`WorldModel` はメンバ変数として `PlayerObject` を保持します．観測できるプレイヤは複数存在するため，`PlayerObject` を STL コンテナに格納し，管理しています．

タイプ量を減らすために，`PlayerObject` のコンテナとして以下の二種類が `typedef` されています．

```
typedef std::list< PlayerObject > PlayerCont;
```

```
typedef std::vector< PlayerObject * > PlayerPtrCont;
```

PlayerCont は PlayerObject の実体を保持するために使用します．要素の挿入や削除が頻繁に行われるため，std::list をコンテナとして使用します．

PlayerPtrCont は PlayeCont に格納されている実体を参照するポインタを保持します．高速な参照を可能にするために，std::vector を使用しています．プレイヤエージェントの内部状態が更新されるたびに，すなわち PlayerCont の内容に変更が加わるたびに PlayerPtrCont は新しく作り直されます．格納される値がポインタである点に注意してください．

3.5.6 InterceptTable

InterceptTable クラスは，WorldModel 内に保持されている情報から各プレイヤがボール捕捉に必要なサイクル数を予測し，その結果を保持するために使用されます．プレイヤエージェントの意志決定においては，このボール捕捉サイクル数を参照することによっていずれのチームの誰がボールを所有しているを判断します．

InterceptTable は以下のようなメンバ関数を持っています．

WorldModel メンバ
int getSelfReachCycle() const; SelfObject(プレイヤエージェント自身) の予測ボール捕捉サイクル数.

int getTeammateReachCycle() const;
味方プレイヤの予測ボール捕捉サイクル数の中で最小の値.
int getOpponentReachCycle() const;
敵プレイヤの予測ボール捕捉サイクル数の中で最小の値.

const PlayerObject * getFastestTeammate() const; 最も早くボールを捕捉できると予測された味方プレイヤへのポインタ.
const PlayerObject * getFastestOpponent() const; 最も早くボールを捕捉できると予測された味方プレイヤへのポインタ.

3.5.7 WorldModel からのアクセス

WorldModel は以下のようなメンバ関数を持っています．これらによってフィールド上の様々な情報を参照できます．

WorldModel メンバ
const GameTime & getTime() const; 現在のシミュレーションサイクルを返す.
const GameMode & getGameMode() const; 試合の現在のプレイモードの情報を返す.
SideID getOurSide() const; 自分のチームのサイド (LEFT または RIGHT) を返す.
const SelfObject & self() const; プレイヤエージェント自身の情報.
const BallObject & ball() const; ボールの情報.
const PlayerCont & teammates() const; 背番号が判明している味方プレイヤ.
const PlayerCont & unknownTeammates() const; 背番号が不明な味方プレイヤ.
const PlayerCont & opponents() const; 背番号が判明している敵プレイヤ.
const PlayerCont & unknownOpponents() const; 背番号が不明な敵プレイヤ.
const PlayerCont & unknownPlayers() const; 完全に識別不能なプレイヤ.
const PlayerPtrCont & getTeammatesFromSelf() const; プレイヤエージェントから近い順にソートされた味方プレイヤ.
const PlayerPtrCont & getOpponentsFromSelf() const; プレイヤエージェントから近い順にソートされた敵プレイヤ. 識別不能なプレイヤを含む.
const PlayerPtrCont & getTeammatesFromBall() const; ボールから近い順にソートされた味方プレイヤ
const PlayerPtrCont & getOpponentsFromBall() const; ボールから近い順にソートされた敵プレイヤ. 識別不能なプレイヤを含む.
const PlayerObject * getOpponentGoalie() const; 敵キーパへのポインタを返す. 存在しない場合は NULL を返す.

const PlayerObject * getTeammateNearestTo(const Vector2D & point, const int count_thr, double * dist_to_point) const; 信頼性が count_thr 以下である，point へ最も近い味方プレイヤへのポインタを返す．条件を満たすプレイヤが存在しない場合は NULL を返す．dist_to_point != NULL ならば，得られるプレイヤから point までの距離が格納される．
const PlayerObject * getOpponentNearestTo(const Vector2D & point, const int count_thr, double * dist_to_point) const; getTeammateNearestTo() の敵プレイヤ版．
const PlayerObject * getTeammate(const int unum) const; 背番号が unum の味方プレイヤへのポインタを返す．存在しない場合は NULL を返す．
const PlayerObject * getOpponent(const int unum) const getTeammate() の敵プレイヤ版．
bool existKickableTeammate() const; ボールをキック可能な味方プレイヤが村内すると推定される場合，true を返す．
bool existKickableOpponent() const; existKickableTeammate() の敵プレイヤ版
const InterceptTable * getInterceptTable() const; InterceptTable の実体への const ポインタを返す．
int getDirCount(const AngleDeg & angle) const; 最後に angle の方向を観測してからの経過サイクル．フィールド上の領域の信頼性情報として使う．
HeteroID getTeammateHeteroID(const int unum) const; 背番号が unum の味方プレイヤのプレイヤタイプ ID を返す．
const double & getOffsideLineX() const; 推定されるオフサイドラインの X 座標値．

表にも現れているように，WorldModel で管理される PlayerObject には背番号やチームが不明なものが含まれます．また，全ての物体には観測されてからの経過サイクルが保持されており，これを使って移動範囲の予測を行うこともできます．プレイヤエージェントの意志決定においては，このような不確実な情報を考慮しなければなりません．

WorldModel クラスにはここでは列挙しきれないほど多くのメンバ関数があります．詳しくは Doxygen によるリファレンスなどを参照してください．

3.5.8 ActionEffector

WorldModel が現在のフィールド上の状態を管理するのに対し，ActionEffector はプレイヤエージェントが実行した (またはする予定の) 行動を記録しておき，その効果を推定するために使用されます．例えば，体の向きを変えながらある位置に首を向けようとするならば，最終的に体がどの方向へ向くのかを把握してから首を振る必要があります．このような，rcssserver 上にはまだ反映されていないかもしれないが，実行した (するつもりの) 行動の管理を ActionEffector が担当します．

PlayerAgent クラスは ActionEffector クラスをメンバ変数として持っています．プレイヤの意志決定時に使用する情報を参照するには，以下のメンバ関数を使用します．

ActionEffector メンバ
AngleDeg queuedNextMyBody() const; 次サイクルの体の方向.
Vector2D queuedNextMyPos() const; 次サイクルの位置座標.
Vector2D queuedNextBallPos() const; 次サイクルのボール位置.
Vector2D queuedNextBallVel() const; 次サイクルのボール速度.
AngleDeg queuedNextAngleFromBody(const Vector2D & target) const; 次サイクルの target 位置の体からの相対方向.
ViewWidth queuedNextViewWidth() const; 次サイクルの視野角.

チーム開発時に使用する機会はありませんが，ActionEffector クラスには他にも多くのメンバ関数があります．詳しくは Doxygen によるリファレンスなどを参照してください．

3.5.9 PlayerAgent からのアクセス

PlayerAgent クラスは WorldModel クラスと ActionEffector クラスをメンバ変数として持っています．SampleAgent からこれらを参照するには，以下の PlayerAgent のメンバ関数を使用します．

PlayerAgent メンバ
const WorldModel & world() const; WorldModel クラスの実体への const 参照を返す．
const ActionEffector & effector() const; ActionEffector クラスの実体への const 参照を返す．

Section 3.6
動作クラスの利用

3.6.1 動作クラス

librcsc では，プレイヤエージェントの動作をクラスライブラリとして実装，管理しています．プレイヤエージェントの動作は，BodyAction, TurnNeckAction, ChangeViewAction そして SoccerBehavior の 4 つの基底クラスを頂点とし，それらの派生クラスとして実装されます．名前からも分かるとおり，これらは，体を動かす動作，首振り行動，視野変更動作，そして，それら 3 つを同時に実行する複合的な動作[1)]，となります．ただし，これら 4 つの基底クラスに機能的な差異はありません．コードの可読性を向上させるために 4 つに分類しているだけに過ぎません．

これら基底クラスでは bool execute(PlyaerAgent *) という純粋仮想関数がインタフェースとして宣言されています．これら基底クラスを継承した動作クラスによって，行動コマンドよりも抽象的な，サッカープレイヤとして意味のある動作が定義されています．例えば，目標位置へ移動する，目標位置へある速度でボールを蹴り出す，目標位置へドリブルする，などといった動作が該当します．

[1)] librcsc で定義済みの SoccerBehavior の具象クラスは体と首の制御だけしか行っていません．ChangeViewAction の呼び出しを忘れずに実行してください．

個々の動作クラスの実装は独立しているため，必要な動作クラスを宣言したヘッダファイルを include することで，各動作クラスをグローバル関数のように扱うことができます．実装済みの各動作クラスのクラス名には，基底クラスに応じて，Body_, Neck_, View_, Bhv_という接頭子をつけています．これによって，各動作クラスがどの基底クラスから派生しているかを一目で判別できるようになっています．

3.6.2 動作の登録

PlayerAgent クラスには，首振り動作と視野変更動作を登録するための以下のような関数を用意しています．登録対象となるのは首振り動作と視野変更動作のみです．

PlayerAgent メンバ
void setTurnNeck(TurnNeckAction * ptr); 首振り動作を予約する．ptr は new で新しく割り当てた動作オブジェクト．
void setChangeView(ChangeViewAction * ptr); 視野変更動作を予約する．ptr は new で新しく割り当てた動作オブジェクト．

これらの動作登録関数の呼出し時には，ActionEffector へのコマンド登録は行われません．実行予定の動作オブジェクトを予約しておくのみで，実際のコマンド登録は後でその動作オブジェクトが改めて行います．このようなまわりくどい方法を取る理由は，首振り動作にあります．プレイヤエージェントが首を振る場合，自身の体の向きと視野角の大きさを考慮しなければなりません．そのため，意志決定過程においてどの方向に首を向けるかを決めることはできても，最終的な体の向きと視野角の大きさが決まらなければ，具体的に何度首を回転させれば良いかを決めることができません．動作予約関数はこの問題を解決してくれます．

3.6.3 意図クラス

librcsc では，プレイヤエージェントの意図を扱う SoccerIntention というクラスを導入しています．librcsc において，意図とは継続実行する動作を意味してお

り，一旦意図が登録されると，その意図の目的を達成するまで意思決定における他の選択肢は排除されます．この意図を利用することで，プレイヤエージェントの制御が容易になる場合があります．例えば，敵プレイヤのマークを外すような短時間だけの集中的な移動動作を行わせたい場合，サイクルごとに意思決定するだけでは状態の判断と管理が繁雑なものになりますが，意図の仕組みはこの問題を簡単に解決してくれます．

SoccerIntention クラスは，前出の動作クラスと同様に `bool execute(PlayerAgent *)` という純粋仮想関数を持っており，動作クラスの一種と言えます．ただし，通常の動作クラスとは異なり，`bool finished(const PlayerAgent *)` という純粋仮想関数も持っています．`finished()` は，その意図が終了したかどうかを判断するために使用されます．派生クラスで `finished()` を適切に定義すれば，始めに設定した目的を達成するまで継続して実行する動作クラスを実現することができます．多くの場合，意図の終了条件として時間制限が設定されます．これは，一定時間，特定の動作を強制的に継続させることを目的としています．

実際に意図クラスを利用するには，プレイヤエージェントの意思決定過程において，`PlayerAgent::setIntention()` によって具象意図クラスのオブジェクトを PlayerAgent に登録します．プレイヤエージェントの意思決定時，登録された意図オブジェクトが存在しているかどうか，存在していればその意図が終了しているかがまず判定されます．もし継続中の意図があれば，プレイヤエージェントはその意図に基づく動作を実行し，意思決定を終了します．登録された意図はその意図が終了したと判定されるまで自動的に実行され続け，終了したと判定されれば PlayerAgent から削除されます．

3.6.4 意図の登録

PlayerAgent クラスには，意図を登録するための以下の関数を用意しています．

PlayerAgent メンバ
`void setIntention( SoccerIntention * ptr );` 意図を登録する．ptr は new で新しく割り当てた意図オブジェクト．

`setIntention()` によって登録された意図は次のサイクルから効果を発揮することに注意してください．現在のサイクルのための動作は現在のサイクルで実行しなければなりません．

Section 3.7
librcsc に含まれる動作クラス

librcsc では，サッカープレイヤとして振る舞うために必要な基本動作スキルを既にいくつか用意しています．以下に，用意してある動作クラスを列挙し，必要なヘッダファイル，コンストラクタ，そして，その動作がどのような効果をもたらすかを簡単に説明します．

3.7.1 体を動かす動作

```
#include <rcsc/action/body_advance_ball.h>
Body_AdvanceBall();
```
敵陣方向へボールを送る．

```
#include <rcsc/action/body_clear_ball.h>
Body_ClearBall();
```
ボールをクリアする．

```
#include <rcsc/action/body_dribble.h>
Body_Dribble( const Vector2D & target_point,
              const double & dist_thr,
              const double & dash_power,
              const int dash_count,
              const bool dodge = true )
```
target_point の位置へ向かって，dash_power のパワーで距離が dist_thr 以下になるま でドリ ブルする．キック後に dash_count 回のダッシュを実行できるようにボールを蹴る．dodge = true の場合は敵プレイヤを自動的に回避する．

動作が完了するまではドリブル動作は自動的に継続して実行される．

```
#include <rcsc/action/body_go_to_point.h>
Body_GoToPoint( const Vector2D & target_point,
                const double & dist_thr,
                const double & dash_power,
                const int cycle = 100,
                const bool back_mode = false,
                const bool save_recovery = true,
                const double & dir_thr = 12.0 );
```

target_point の位置へ，dash_power のパワーで距離が dist_thr 以下になるまで移動する．cycle サイクル後にちょうども句評位置へ到達するように，ダッシュパワーは自動調整される．back_mode = true であれば，後方ダッシュを実行する．save_recovery = true であれば，*recover* の値を減らさないようにダッシュパワーが自動調整される．移動中の target_point の方向と体の方向との誤差が dir_thr は以下である．

```
#include <rcsc/action/body_hold_ball.h>
Body_HoldBall( const bool do_turn = false,
               const Vector2D & face_target
                                = Vector2D( 0.0, 0.0 ) );
```

その場に留まりながら，敵プレイヤに奪われないようにボールを制御する．do_turn = true の場合，回転する余裕があれば face_target へ向けて体を回転させる．

```
#include <rcsc/action/body_intercept.h>
Body_Intercept( const bool save_recovery = true );
```

ボールを追いかける．save_recovery = true であれば，*recover* を減らさないようにダッシュパワーを自動調整する．

```
#include <rcsc/action/body_kick_collide_with_ball.h>
Body_KickCollideWithBall();
```

ボールを蹴って自分自身と意図的に衝突させる．

```
#include <rcsc/action/body_kick_multi_step.h>
Body_KickMultiStep( const Vector2D & target_point,
                    const double & first_speed,
                    const bool enforce = false );
```

target_point へ向かって first_speed の初速で蹴り出す．至近距離に存在する敵プレイヤを考慮して，キック可能領域内でのボールの移動経路をプランニングする．. 必要に応じて複数回のキックをプランニングする．ただし，キック動作は継続実行されないので，次のサイクルでは改めてキックのプランニングをやり直さなければならない．enforce = true であれば，目標速度に到達させられない場合でも 2 ステップ以内で強制的に蹴り出すプランニングを行う．

```
#include <rcsc/action/body_kick_one_step.h>
Body_KickOneStep( const Vector2D & target_point,
                  const double & first_speed );
```

target_point へ向かって first_speed の初速で蹴り出す．キックは 1 回のみ．1 回のキックで目標速度に到達させられない場合は，現在の位置関係でボールをキープする．

```
#include <rcsc/action/body_kick_to_relative.h>
Body_KickToRelative( const double & target_dist,
                     const AngleDeg & target_angle_rel,
                     const bool stop );
```

自分自身の中心から target_dist の距離で，体の正面から target_angle_rel の角度の位置へボールをキックする．必要に応じて複数回のキックをプランニングする．ただし，キック動作は継続実行されないので，次のサイクルでは改めてキックのプランニングをやり直さなければならない．stop = true であれば，目標位置でボール速度を 0 にできるようにプランニングを行う．

```
#include <rcsc/action/body_kick_two_step.h>
Body_KickTwoStep( const Vector2D & target_point,
                  const double & first_speed,
```

```
                    const bool enforce = false );
```

target_point へ向かって first_speed の初速で蹴り出す．キックは最大で 2 ステップ分まで考慮する．ただし，キック動作は継続実行されないので，次のサイクルでは改めてキックのプランニングをやり直さなければならない．1 回のキックで目標速度に到達させられるならば，キックは 1 回だけになる．enforce = true ならば，2 回のキックで目標速度に到達させられない倍でも 2 ステップで強制的に蹴り出すプランニングを行う．

```
#include <rcsc/action/body_pass.h>
Body_Pass();
```

味方プレイヤへのパスを実行する．パスコース探索は自動的に行われ，探索した中で最も評価の高いパスコースが自動的に選択される．パスコースが見付からなかった場合は何の動作も実行されず，Body_Pass::execute() は false を返す．この動作を呼び出す前に，static メンバ関数 Body_Pass::get_best_pass() を呼び出して適切なパスコースが存在するかどうかを確認することができる．

```
#include <rcsc/action/body_shoot.h>
Body_Shoot();
```

シュートコースが存在すればシュートを実行する．シュートコースの探索は自動的に行われる．シュートコースが見付からなかった場合は何の動作も実行されず，Body_Shoot::execute() は false を返す．この動作は，プレイヤエージェントの意思決定処理の最初の部分で一度だけ呼び出せば充分である．

```
#include <rcsc/action/body_stop_ball.h>
Body_StopBall();
```

プレイヤエージェントとボールとの現在の位置関係を維持するようにボールをキックする．ただし，ボールと衝突しないように，また，キック不可能な状態にならないように，自動的な調整が行われる

```
#include <rcsc/action/body_stop_dash.h>
```

```
Body_StopDash( const bool save_recovery );
```
プレイヤエージェントの速度が 0 になるようにダッシュする．ただし，プレイヤエージェントの体の方向の速度成分しか考慮されないため，体と垂直方向への速度を変化させることはできない．save_recovery = true であれば，*recover* を減少させないようにダッシュパワーが自動調整される．

```
#include <rcsc/action/body_turn_to_ball.h>
Body_TurnToBall( const int cycle = 1 );
```
cycle サイクル後のボール位置へ体を向けられるように回転動作を実行する．自分自身の慣性による移動も考慮される．1 回の回転動作で目標の回転角度を達成できるとは限らない．

```
#include <rcsc/action/body_turn_to_point.h>
Body_TurnToPoint( const Vector2D & target_point,
                  const int cycle = 1 )
```
cycle サイクル後に target_point へ体を向けられるように回転動作を実行する．自分自身の慣性による移動が考慮される．1 回の回転動作で目標位置へ体を向けられるとは限らない．

3.7.2 首振り動作

```
#include <rcsc/action/neck_scan_field.h>
Neck_ScanField();
```
首振りのみで周囲を見回す．最後に観測してからの経過時間が最も長い方向が優先される．

```
#include <rcsc/action/neck_turn_to_ball.h>>
Neck_TurnToBall();
```
次のサイクルの予測ボール位置へ視界を向ける．

```
#include <rcsc/action/neck_turn_to_ball_or_scan.h>
Neck_TurnToBallOrScan();
```

ボール情報の信頼性が低い場合は次サイクルの予測ボール位置へ視界を向ける．ボール情報の信頼性が高い場合は周囲を見回す．自分自身以外のボールをキック可能なプレイヤが存在すれば，そのプレイヤのキックに寄ってボールが移動する範囲を考慮して，ボールに注目するか否かを決定する．

```
#include <rcsc/action/neck_turn_to_goalie_or_scan.h>
Neck_TurnToGoalieOrScan();
```

敵キーパ情報の信頼性が低い場合はキーパの位置へ視界を向ける．ボール情報の信頼性が高い場合，または，敵キーパの位置へ視界を向けることが不可能な場合は Neck_TurnToBallOrScan を実行する．

```
#include <rcsc/action/neck_turn_to_low_conf_teammate.h>
Neck_TurnToLowConfTeammate();
```

情報の信頼性が低い味方プレイヤを優先してその位置へ視界を向ける．注目すべき味方プレイヤが存在しない場合は Neck_TurnToBallOrScan を実行する．

```
#include <rcsc/action/neck_turn_to_point.h>
Neck_TurnToPoint( const Vector2D & target_point );
```

target_point へ視界を向ける．次サイクルの自分自身の位置と体の向きが自動的に反映される．

```
#include <rcsc/action/neck_turn_to_relative.h>
Neck_TurnToRelative( const AngleDeg & rel_angle );
```

体の向きから相対に rel_angle の角度へ首を回転させる．

3.7.3 視界モード変更

```
#include <rcsc/action/view_normal.h>
View_Normal();
```
視界モードを (normal, high) の状態にする.

```
#include <rcsc/action/view_synch.h>
View_Synch();
```
視覚情報の受信頻度を rcssserver のサイクルと同期させ, 視覚情報を毎サイクル受信できるように視界モードを変更する. 通常, (normal, high) を 1 サイクル, (narrow, high) を 2 サイクルという 3 サイクル 1 セットの視界モード変更を繰り返す.

```
#include <rcsc/action/view_wide.h>
View_Wide();
```
視界モードを (wide, high) の状態にする.

3.7.4 複合的な動作

```
#include <rcsc/action/bhv_before_kick_off.h>
Bhv_BeforeKickOff( const Vector2D & pos );
```
試合開始前, または, ゴール直後の動作. pos の位置へジャンプし, その位置で周囲の確認を実行し続ける.

```
#include <rcsc/action/bhv_body_neck_to_ball.h>
Bhv_BodyNeckToBall();
```
次サイクルの予測ボール位置へ体と視界を向ける.

```
#include <rcsc/action/bhv_body_neck_to_point.h>
Bhv_BodyNeckToPoint( const Vector2D & target_point );
```

target_point の位置へ体と視界を向ける．

```
#include <rcsc/action/bhv_go_to_point_look_ball.h>
Bhv_GoToPointLookBall( const Vector2D & target_point,
                       const double & dist_thr,
                       const double & dash_power,
                       const double & back_power_rate = 0.7 );
```

target_point の位置へ,. dash_power のパワーで距離が dist_thr 以下になるまで移動する．ただし，常にボールの方向へ視界を向けながら移動する．そのために，必要に応じて後方ダッシュを実行する．後方ダッシュを実行する場合，使用するダッシュパワーは $dash\_power \times back\_power\_rate$ になる．**recover** が減少しないように，ダッシュパワーの自動調整が行われる．

```
#include <rcsc/action/bhv_neck_body_to_ball.h>
Bhv_NeckBodyToBall();
```

次サイクルの予測ボール位置へ首と体を向ける．首の回転だけでボールを視界に入れることが出来るならば，体の回転動作は実行されない．

```
#include <rcsc/action/bhv_neck_body_to_point.h>
Bhv_NeckBodyToPoint( const Vector2D & target_point );
```

target_point の位置へ首と体を向ける．首の回転だけでその位置を視界に入れることが出来るならば，体の回転動作は実行されない．

```
#include <rcsc/action/bhv_scan_field.h>
Bhv_ScanField();
```

周囲を見渡す．ボールの位置情報が得られている場合，体は 90 度ずつ回転させ，首振りは Neck_ScanField によって制御する．ボールの位置情報が得られていない場合，ボール探索を開始する．その際，視界モードを (wide, high) に変更し，ボールが見つかるまで，視界方向を 180 度反転させる動作を繰り返す．

3.7.5 意図

```
#include <rcsc/action/intention_dribble.h>
IntentionDribble( const Vector2D & target_point,
                  const double & dist_thr,
                  const int turn_step,
                  const int dash_step,
                  const double & dash_power_abs,
                  const bool back_dash,
                  const GameTime & start_time );
```

ドリブルの継続実行用の意図クラス．Body_Dribble から自動的に呼び出されるので，チーム開発時に使用することは無い．

```
#include <rcsc/action/intention_kick.h>
IntentionKick( const Vector2D & target_point,
               const double & first_speed,
               const int kick_step,
               const bool enforce,
               const GameTime & start_time );
```

キックの継続実行用の意図クラス．target_point の位置へ，キック回数が kick_step 以内で初速 first_speed でボールを蹴り出せるようにキックを実行する．enforce = true であれば，目標速度に到達させられなくとも，強制的に目標方向へとボールを蹴り出す．start_time は意図の開始時間で，動作継続時間の確認のために使用される．

```
#include <rcsc/action/intention_time_limit_action.h>
IntentionTimeLimitAction( BodyAction * body_action,
                          TurnNeckAction * neck_action,
                          ChangeViewAction * view_action,
                          const long & max_step,
                          const GameTime & start_time );
```

時間制限付きの汎用意図クラス．body_action, neck_action, view_action のそれぞれに，new で動的に確保した動作クラスを与える．意図の継続時間は start_time から max_step サイクルである．

Section 3.8
デバッグ

上手く動くチームを作るには，チーム開発者の意図どおりに実装が行われているかどうかの動作確認が必要です．しかし，サッカーエージェントプログラムは rcssserver と連携させてリアルタイムに動作させなければならないため，実際に動作させながらの動作確認が難しくなっています．そこで，プログラムをいったん動かしてログを残し，後から (オフラインで) トレースする，という方法を取ります．更に，このトレース作業を省力化するための特別なデバッガツールも使用します．

3.8.1 Logger の利用

librcsc では，プレイヤエージェントの動作内容，特に意思決定に関する記録を出力するために，Logger というクラスを用意しています．Logger を使うことで，試合時間とともに任意のメッセージを外部のファイルに出力できます．また，記録するメッセージごとにログレベルを設定することができます．

ログレベルとは，トレース作業時の情報量の調節を可能にするパラメータです．例えば，サッカーチームとしての動作をトレースしたい場合に，rcssserver との通信や同期に関する処理の記録が混在していては，トレース作業は非常に繁雑になり極めて非効率的です．ファイル出力は負荷が高いため，不要な情報を初めから出力させないという制御もしたいところです．このように，必要に応じて情報の取捨選択を可能にするのがログレベルを導入する目的です．

Logger クラスには，インタフェースとして汎用的に使用できる以下のメッセージ出力関数を用意しています．

Logger メンバ
`double addText( const boost::int32_t id, char * msg, ...);` ログレベルが id のメッセージを出力する．printf() のように可変個引数を取ることができる．

将来的には，他にもいくつかの特定メッセージ出力用の関数を用意する予定です．例えば，フィールド上の位置とともに円や直線などの図形の情報を記録しておき，後述のビジュアルデバッガ上に表示できるようにする，といった機能の追加を想定しています．

出力するファイルの管理などは `PlayerAgent` クラスに既に実装されており，チーム開発時に意識する必要はありません．デフォルトでは，ログファイルは`/tmp` 以下に，“チーム名-背番号.log” というファイル名で生成されます．出力先ディレクトリは，`PlayerConfig` の `log_dir` オプションによって変更できます．必要に応じて，`player.conf` や軌道スクリプトを編集してください．ただし，NFS 環境などで動かす場合も考慮すると，`/tmp` をそのまま使用することを推奨します．

ログレベルの有効無効の切替えは，`PlayerConfig` クラスに用意されているスイッチを切替えることで行います，本書執筆時点の `PlayerConfig` によって切替え可能なログレベルのオプションは，以下の 13 種類です．

オプション名	内容
debug	これが無効の場合，全てのログメッセージが出力されなくなる．何らかのログを記録するには，必ずこのオプションを有効にしなければならない．
debug_system	rcssserver との同期に関する情報など，サッカーエージェントプログラムとしての基本的な動作内容の記録．
debug_sensor	rcssserver からのセンサ情報解析に関する記録．
debug_world	プレイヤエージェントの内部モデル管理に関する記録．
debug_action	サッカープレイヤとしての基本的な動作に関する記録．
debug_intercept	インターセプト動作に関する記録．
debug_kick	キックのプランニングに関する記録．
debug_dribble	ドリブルのプランニングに関する記録．
debug_pass	パス動作に関する記録．
debug_cross	ゴール前のクロスに関する記録．
debug_shoot	シュート動作に関する記録．
debug_clear	ボールクリア動作に関する記録．
debug_team	チームレベルの意思決定に関する記録．
debug_role	役割に応じた意思決定に関する記録．

以上は暫定的に用意したログレベルとそのオプション名であり，必要ならば変

更/追加してください．これらのログレベルは Logger クラスで宣言されているので，まずはそちらを変更し，それに合わせて PlayerConfig クラスを編集してください．後でも触れますが，ログレベルは最大 32 種類まで用意することが出来ます．ただし，筆者の経験上，ログレベルをあまり細かく分類してもかえって繁雑になるだけなので，なるべくシンプルな分類を維持しておくことをお勧めします．

実際にログメッセージをを出力させたい箇所では，以下のように書きます．

```
dlog.addText( Logger::ACTION,
              "%s:%d: Body_HoldBall"
              ,__FILE__, __LINE__ );
```

librcsc のソースを覗けば，いたるところで使用例を見つけることが出来るでしょう．

dlog は，logger.{h,cpp} で宣言されているグローバル変数で，Logger クラスの実体です．logger.h を include しておけば，どこからでも参照できます．

addText() の第一引数にはログレベルの Id を指定しますが，これには logger.h で宣言されている値を使用します．logger.h を見ると，各ログレベルが以下のようにビットで表現されているのを見つけることが出来ます．

```
static const boost::int32_t LEVEL_00 = 0x00000000;
static const boost::int32_t LEVEL_01 = 0x00000001;
static const boost::int32_t LEVEL_02 = 0x00000002;
static const boost::int32_t LEVEL_03 = 0x00000004;
...
// 省略
...
static const boost::int32_t LEVEL_30 = 0x20000000;
static const boost::int32_t LEVEL_31 = 0x40000000;
static const boost::int32_t LEVEL_32 = 0x80000000;
static const boost::int32_t LEVEL_ANY = 0xffffffff;

static const boost::int32_t SYSTEM    = LEVEL_01;
static const boost::int32_t SENSOR    = LEVEL_02;
static const boost::int32_t WORLD     = LEVEL_03;
static const boost::int32_t ACTION    = LEVEL_04;
static const boost::int32_t INTERCEPT = LEVEL_05;
static const boost::int32_t KICK      = LEVEL_06;
static const boost::int32_t DRIBBLE   = LEVEL_07;
static const boost::int32_t PASS      = LEVEL_08;
static const boost::int32_t CROSS     = LEVEL_09;
static const boost::int32_t SHOOT     = LEVEL_10;
static const boost::int32_t CLEAR     = LEVEL_11;
static const boost::int32_t TEAM      = LEVEL_12;
static const boost::int32_t ROLE      = LEVEL_13;
```

boost::int32_t は，その名のとおり 32 ビットの整数型です．addText() で使用するのは，SYSTEM から ROLE までの static メンバ変数です．

チーム開発時には，Logger::TEAM と，Logger::ROLE の二つだけで充分です．これらを使って，開発者の意図どおりにプレイヤエージェントの意思決定のフローが進んでいるかどうかを確認するために，いたるところに addText() によるメッセージ出力を埋め込んでください．

3.8.2 ビジュアルデバッガの利用

プレイヤエージェントの開発には特別なデバッグツールが必要不可欠です．サッカーシミュレーションの世界では，ビジュアルデバッガという，描画機能を持つデバッグツールが存在します．

このようなビジュアルデバッガでは，通常のモニタの表示に加えて，プレイヤエージェントの内部状態の表示を可能にしています．これによって，プレイヤエージェントが認識している物体の位置のずれを確認する，現在のプレイヤエージェントの意思決定内容を確認する，といった作業を視覚的に行えるようになります．テキストデータで数値を見ているだけでは気づけないような現象を発見することも容易になります．

ビジュアルデバッガとしては，SoccerScope や Soccer Viewr などが有名です．いずれも国産のツールで，その仕様やソースが公開されています．他にも，世界で名の知れたチームはまず間違いなく何らかのビジュアルデバッガを使っています．しかし，それらはほとんど公開もされておらず，その仕様が各チーム独自のものになっているのが現状です．筆者自身は長い間 Soccer Viewer を愛用していましたが，現在は筆者自身で開発した soccerwindow2 を使用しています．

soccerwindow2 は以下のような機能を持っています．librcsc を使用すれば，これらの機能をすぐに使うことが出来ます．

- Soccer Viewr 互換のデバッグサーバ
- プレイヤエージェントの内部状態表示
- `Logger` が出力したログメッセージの読み込みと表示
- ログレベルの切替え

デバッグサーバとは，プレイヤエージェントを rcssserver と接続するのと同時に soccerwindow2 にも接続させ，サイクルごとの情報をオンラインで送受信することを可能にしたシステムです．デバッグサーバの利点は，ディスク上の入出力を無くなることによる負荷の軽減です．一般的にファイルへの出力は非常に負荷が大きいため，プレイヤエージェントが毎サイクルファイルへの出力を行っていると，その動作にしばしば多大な影響を与えてしまいます．具体的には，センサ情報やコマンドのロストが発生してしまいます．動作確認のためのログ出力が不具合を発生させては意味がありません．

soccerwindow2 をデバッグサーバとして使用するには，以下の手順を踏む必要があります．

- rcssserver とサッカーモニタモードの soccerwindow2 を起動する．
- soccerwindow2 でデバッグサーバを起動する．
- チームの各プレイヤをデバッグクライアントモードで起動する．

soccerwindow2 をサッカーモニタモードで起動するには，rcssserver を同じホスト上で実行している状態で，以下のコマンドを実行します．

```
$ soccerwindow2 -c
```

もちろん，他のホストで動作する rcssserver に接続することも出来ます．’–host’ オプションに引数でホスト名を指定すれば，異なるホスト上の rcssserver に接続します．詳しくは，soccerwindow2 に’–help’ オプションを付けて起動すると表示されるヘルプメッセージを参照してください．

無事起動すれば，続いてデバッグサーバを起動します．これは，soccerwindow2 のメニューで実行できます (図 3.1)．

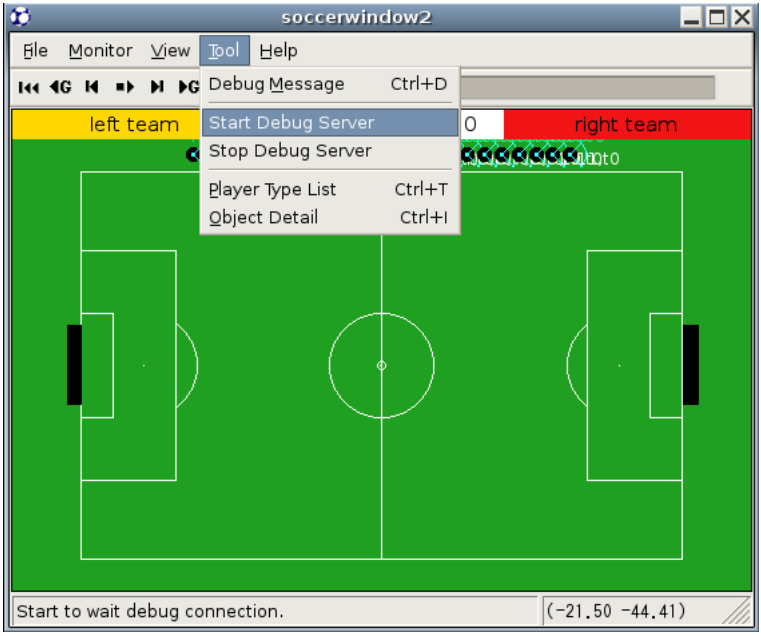


図 3.1 soccerwindow2 上でのデバッグサーバの起動

または，以下のようにコマンドラインオプションに'-d' を与えます．

```
$ soccerwindow2 -c -d
```

デバッグサーバの使用が中心であれば，以上の二つのオプションを含んだ起動スクリプトを用意しておくと良いでしょう．

プレイヤエージェントの起動時には，以下のオプションを与えます．必要に応じて，`player.conf` か起動スクリプトを編集してください．

オプション名	内容
debyg_connect	有効にすると，soccerwindow2 に接続し，内部状態の送信を自動的に開始する．

無事チームが起動すれば，内部状態を参照したいプレイヤを選択してみてください．選択は，対応する背番号の数字キーを押すか，メニューの “View” → “Preference” で出てくるダイアログを利用して行ってください．デバッグサーバとの接続がうまくいっていれば，通常の表示に加えて，図 3.2 のように多くの追加情報が表示されます．

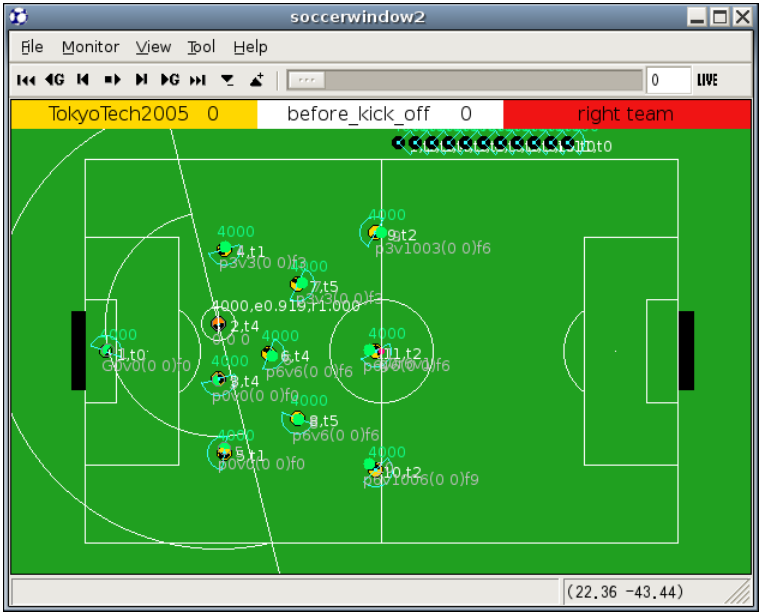


図 3.2 内部状態の表示

プレイヤエージェントの内部状態のうち，他の物体の位置情報は自動的に出力されるようになっています．チーム開発時に追加する必要があるのは，その他のサッカープレイヤとしての意思決定に関わる情報です．librcsc では，デバッグサーバへ情報を送るためのクライアント機能の実装として，DebugClient というクラスを用意しています．

DebugClient クラスは以下のメンバ変数を持っています．

DebugClient メンバ
`void addMessage( char * msg, ... );` 任意のメッセージを出力する．可変個引数を取ることが出来る．1 サイクル中に複数回使用できる．
`void setTarget( const int unum );` 行動の対となる味方プレイヤ象 (パス相手など) の背番号をセットする． 1 サイクルに 1 回しか使用できない．
`void setTarget( const Vector2D & p );`

行動の目標位置をセットする．1 サイクルに 1 回しか使用できない．

```
void addLine( const Vector2D & from, const Vector2D & to );
```

任意の二点間を結ぶ線分の描画を要求する． 1 サイクル中に複数回使用できる．

1 サイクルに 1 回しか使用できない関数があることに注意してください．もし，複数回呼び出してしまった場合は，最後に呼び出した内容によって出力すべき内容が上書きされます．また，複数回利用できる関数に関しても，呼び出せる回数は可能な限り抑えるべきです．デバッグサーバへ送信するメッセージ長は 8192 文字以内に制限されており，それ以上の長さのメッセージ町が送られた場合の動作は不定であるためです．

実際の使用例は以下のようになります．

```
agent->debugClient().setTarget( drib_target );
agent->debugClient().addMessage( "DribAtt%d", dash_count );

agent->debugClient().addMessage( "MidPass(1)" );

// draw offside line
agent->debugClient().addLine
  ( Vector2D( agent->world().getOffsideLineX(),
              agent->world().self().pos().y - 15.0 ),
    Vector2D( agent->world().getOffsideLineX(),
              agent->world().self().pos().y + 15.0 ) );
```

このように，PlayerAgent クラスには debugClient() というメンバ関数があり，これは DebugClient の実態への参照を返します．

Section 3.9
スタミナを考慮したポジショニング

通常の試合進行時，各プレイヤはフォーメーションを維持するための移動動作を行います．そして，必要となれば，ボールを全力で追いかけなければなりません．

しかし，プレイヤエージェントのスタミナは限られているため，移動の動作時に無駄に素早く動くと肝心な時に動けなくなってしまいます．このような事態を避けるために，普段はスタミナ消費を抑えて移動しなければなりません．rcssserverにおけるプレイヤの移動とスタミナのモデルについては 4.2.3 節で詳しく説明しているので，rcssserver の仕様に関する知識を持っていない場合はまずそちらを参照してください．

3.9.1 目標位置との距離の閾値

フォーメーションを維持する場合に限らず，プレイヤの移動動作には常に目標位置が設定されます．プレイヤが移動した結果，目標位置との距離の差が充分小さくなれば，それ以上移動する必要が無くなります．つまり，距離の差の許容値が小さすぎると，いつまでも移動し続けることになります．

注意すべきは，プレイヤエージェントの視覚センサから得られる情報には誤差が含まれることです．librcsc の実装においては，まず，自分自身の位置の測定において最大で $0.5m$ 程度の誤差が発生します．更に，移動物体の観測誤差は，対象物体との距離の約 0.1 倍となっており，$100m$ 離れれば $\pm 5m$ ほどの誤差が発生します．視覚センサに含まれる誤差に付いては，6.5 節で詳しく解説しています．

ここで，フォーメーションはボール位置を中心にして維持されることを思い出してください．これは，ボールの位置が変化すれば，目標位置も変化するということです．もしも，移動目標位置との距離の許容値がボールの観測誤差よりも小さければ，プレイヤエージェントはボールの観測後左に振り回されて，細かい動きを繰り返すことになります．そもそも，ボールから遠くにいるプレイヤが積極的に移動する必要はありません．このような無駄な動きを減らすことが効率的なチームプレイへの第一歩です．

3.9.2 ダッシュパワーの調整

ボールとの位置関係によっては積極的に移動する必要はありません．実際の人間のサッカーにおいても，ボール周辺のプレイに関わっていないプレイヤは歩いていることあります．このように，不必要にスタミナを消費しないように，ダッシュパワーを調整し，移動速度を抑えることが必要です．

スタミナ管理のためのダッシュパワーの調整を実現する最も簡単な方法は，ボールとの位置関係に応じたルールを書いてしまうことです．もちろん，何らかの関数近似を採用した法がスマートな実装になります．しかし，スタミナ管理がブラッ

クボックス化してしまうと，現在の戦術に応じた柔軟な対応が出来なくなる上に，ヘテロジニアスプレイヤへの対応が難しくなってしまいます．よって，ルールでの表現が最も適しているのではないかと思います．決定木学習を採り入れるのも良いでしょう．

3.9.3　スタミナ回復

普段はスタミナを温存するように移動していたとしても，突発的に素早く移動しなければならない状況がしばしば発生します．例えば，ボールを追いかけるとき，ディフェンスラインの間をボールが抜けてしまった場合などは，プレイに関係しそうなプレイヤは全力で移動しなければなりません．すると，状況が落ち着いた時にはスタミナが相当減っていることになります．このような状況で通常どおりのダッシュパワーで移動することは得策ではありません．

スタミナが著しく減少している場合は，通常よりも更にダッシュパワーを抑え，回復に専念し，スタミナが充分に回復すれば，再び通常のスタミナ管理ルールに戻すという方法が効率的です．スタミナが切れた状態のまま動きまわろうとすると，使用できるスタミナ量が小さいために結局ダッシュパワーが小さくなり，単位時間あたりの移動量が減少します．結果的に，全く回復する機会の無いまま移動し続ける羽目になります．

3.9.4　スタミナ管理の実装例

簡単なスタミナ管理の実装例を以下に示します．この例では，回復モードのフラグをローカル static 変数で管理しています．この `getDashPower()` という関数が呼ばれるたびに状態のチェックがなされ，回復モードが更新されます．

```
double getDashPower( const PlayerAgent * agent,
                     const Vector2D & target_point ) {
  // ローカル static 変数で回復モードを管理する
  static bool s_recover_mode = false;
  const WorldModel & wm = agent->world();
  if ( wm.self().stamina() < ServerParam::i().staminaMax() * 0.5 ) {
    s_recover_mode = true;
  } else if ( wm.self().stamina()
              > ServerParam::i().staminaMax() * 0.7 ) {
    s_recover_mode = false;
  }
  double dash_power = ServerParam::i().maxPower();
  // 1 サイクルあたりのスタミナ回復量
  const double my_inc
    = wm.self().playerType().staminaIncMax() * wm.self().recovery();
  if ( wm.getDefenseLineX() > wm.self().pos().x
       && wm.ball().pos().x < wm.getDefenseLineX() + 20.0 ) {
    // ディフェンスラインよりも後方にいる場合
    dash_power = ServerParam::i().maxPower();
  } else if ( s_recover_mode ) {
    // 回復モードでは 1 サイクル 25 以上の回復量を確保する
    dash_power = std::max( 0.0, my_inc - 25.0 );
  } else if ( wm.existKickableTeammate()
              && wm.ball().distFromSelf() < 20.0 ) {
    // 味方が近くでボールを持っていれば，移動速度を早める
    dash_power = std::min( my_inc * 1.1,
                           ServerParam::i().maxPower() );
  } else if ( wm.self().pos().x > wm.getOffsideLineX() ) {
    // オフサイド位置であれば全力で戻る
    dash_power = ServerParam::i().maxPower();
  } else {
    // 通常時は回復量よりも大きいダッシュパワーで移動する
    dash_power = std::min( my_inc * 1.7,
                           ServerParam::i().maxPower() );
  }
  return dash_power;
}
```

ダッシュパワーとして，プレイヤエージェントの 1 サイクルあたりのスタミナ回復量を常に使用すれば，スタミナ消費は 0 になります．しかし，それではあまりに移動速度が遅いので，スタミナ回復量よりは大きい値を使用すべきです．大抵の場合，ダッシュだけでなく回転動作も含まれるので，チューニングがうまくいっていれば，スタミナ不足になることはあまりありません．

ただし，ヘテロジニアスプレイヤに関しては注意が必要です．ヘテロジニアスプレイヤのパラメータによっては，通常のプレイヤよりも大幅に燃費が悪いことがあります．そのようなパラメータを持つプレイヤは，なるべく移動させず，重要な場面に飲み集中的に働かせるような工夫が必要です．

Section 3.10
ボール所有者判定

プレイヤエージェントがボールを持っていない場合，プレイヤエージェントは常にボール所有者判定を行わなければなりません．特に，現在誰もボールを持っていないときには次に誰がもっとも早くボールに辿り着けるかを予測し，攻撃的状況か守備敵状況かを判断します．もしプレイヤエージェント自身が最も早くボール捕捉できるのであれば，インターセプト動作を開始しなければなりません．

3.10.1 インターセプトサイクルの参照

ボール所有者判定は，各プレイヤの予測インターセプトサイクルを比較することで行います．インターセプトサイクルとは，現在のサイクルからボール捕捉を完了させるまでに要するサイクル数を意味します．この情報は，`InterceptTable` クラスによってサイクルごとの意思決定直前に自動的に更新されます．各プレイヤのインターセプトサイクルは以下のようにして参照できます．

```
const WorldModel & wm = agent->world();
int self_min = wm.getInterceptTable()->getSelfReachCycle();
int mate_min = wm.getInterceptTable()->getTeammateReachCycle();
int opp_min = wm.getInterceptTable()->getOpponentReachCycle();
```

それぞれ，プレイヤエージェント自身，味方プレイヤ，敵プレイヤの最小インターセプトサイクルを意味します．既にボールを持っているプレイヤのインターセプトサイクルは 0 になります．

3.10.2 ボール所有者の推定

ボール所有者の推定は，各最小インターセプトサイクルの比較によって行われます．単純に考えれば，self_min, mate_min, opp_min の中から最小値のプレイヤを選択すれば良いように思えます．しかし，実際には状況やプレイヤエージェントに割当てられた役割に応じて，判定基準を変更させなければなりません．何故なら，インターセプトサイクルの情報は，プレイヤエージェントが観測した情報に基づいて推定した情報に過ぎないため，その精度はそれほど高くないためです．

例えば，プレイヤエージェントに守備的役割を与えている場合に，プレイヤエージェントのインターセプトサイクルが敵プレイヤのものよりも 1 サイクルだけ小さい状況を想像してみてください．たった 1 サイクルの差では，観測誤差やその後の行動の伴うノイズのために，最終的なボール所有者は簡単に逆転してしまうでしょう．守備的役割のプレイヤがわざわざこのような挑戦的な行動を取ることは非常に危険です．そのため，守備的役割のプレイヤは，自分の方が確実に早くボールに追い付くことができると通常よりも余裕を持って判断できた場合にのみインターセプト動作を開始すべきです．逆に，攻撃的役割を与えられた場合は，味方プレイヤの中で最も早くボール捕捉できると判断できれば，敵プレイヤの情報を無視して常にボールを追いかけた方が良いでしょう．

以下に，守備的役割の場合の実装例を示します．

```
if ( self_min < mate_min && self_min < opp_min - 3 ) {
  Body_Intercept().execute( agent );
  agent->setNeckAction( new Neck_TurnToBall() );
  return;
}
```

このように，敵プレイヤのインターセプトサイクルの比較に数サイクルの余裕を持たせると守備が安定します．

プレイヤエージェントが攻撃的役割でボールが敵ゴールの近くにある場合は，以下のようにボールの近くにいれば積極的にインターセプトを実行すると効果的です．

```
if ( self_min < mate_min
     || ( ! wm.existKickableTeammate()
          && self_min <= 6
          && wm.ball().pos().dist( home_pos ) < 10.0 ) ) {
  Body_Intercept().execute( agent );
  agent->setNeckAction( new Neck_TurnToBall() );
  return;
}
```

通常のプログラミングにおいては，このようなマジックナンバーを埋め込む実装は推奨されません．しかし，ボール所有者判定は状況によってその判断基準が大きく変化するため，一定のルールや関数で記述することが非常に困難です．よほど良いモデルの構築に成功しない限り，必要に応じてコードにパラメータを直接埋め込む方が実装とテストの効率が良いのが現状です．

Section 3.11
動的なポジショニング

3.3 節では，SBSP というフォーメーションシステムを説明しました．この SBSP では，個々のプレイヤエージェントは他のプレイヤの動きを考慮しておらず，ボールの位置だけを入力として移動目標位置が決定されます．これは，真に協調的なポジショニングを行っているわけではなく，あくまで見かけ上協調しているように見えるに過ぎないことを意味します．よって，他のプレイヤの動きを考慮した動的なポジショニングは，フォーメーションシステムとは別に実装しなければなりません．

本節で説明するポジショニング動作には，どちらかといえば個人スキルに近いものもあります．しかし，動作の実装においてはチーム全体のバランスを考慮した調整が必要であるため，チーム開発の一部として取り上げています．

3.11.1 敵プレイヤのマーク

特定の敵プレイヤにぴったりと張りついて移動するマーク動作は，味方ゴール前でのプレイ時に必要とされます．もちろん，それ以外の状況でマーク動作を実行しても良いのですが，プレイヤのスタミナが持たないでしょう．フォーメーションを大きく崩してしまい危険な状況を招きかないとも限りません．味方ゴール前以外では，厳密なマークではなく若干近寄る程度にとどめておいた方が無難です．

味方ゴール前において敵プレイヤをマークする場合，ゴール前でのパスレシーブ動作を妨害することがこの動作の最大の目的となります．これを実現するには，マーク対象プレイヤよりも少し味方ゴール寄り，かつ，少しボール寄りの位置への移動動作が必要になります．

マーク動作の移動目標位置を決める前に，まずは，マーク対象となる敵プレイヤを決定します．以下に実装例を示します．

最初に，候補となる敵プレイヤを探します．

```
const PlayerObject * getMarkTarget( const WorldModel & wm,
                                    const Vector2D & home_pos ) {
  double dist_opp_to_home = 1000.0;
  const PlayerObject * opp
    = wm.getOpponentNearestTo( M_home_pos, 1, &dist_opp_to_home );
  if ( ! opp
       || ( wm.existKickableTeamamte()
            && opp->distFromBall() > 2.5 ) ) {
    return NULL;
  }
  if ( dist_opp_to_home > 7.0 && home_pos.x < opp->pos().x ) {
    return NULL;
  }
  const PlayerPtrCont::const_iterator
    end = wm.getTeammatesFromSelf().end();
  for ( PlayerPtrCont::const_iterator
          it = wm.getTeammatesFromSelf().begin();
        it != end;
        ++it ) {
    if ( (*it)->pos().dist( opp->pos() ) < dist_opp_to_home ) {
      return NULL;
    }
  }
  return opp;
}
```

続いて，マーク対象となる敵プレイヤに対する移動位置を決定します．opp を getMarkTarget で得た PlayerObject へのポインタとすると，

```
AngleDeg block_angle = ( wm.ball().pos() - opp->pos() ).th();
Vector2D block_point
  = opp->pos() + Vector2D::polar2vector( 0.2, block_angle );
block_point.x -= 0.1;
if ( block_point.x < wm.self().pos().x - 1.5 ) {
  block_point.y = wm.self().pos().y;
}
```

最後の Y 座標の修正は，余計な回転動作を入れずにとにかく後ろに下がることを最優先するために必要です．ここでは，プレイヤエージェント自身の位置の Y 座標値としましたが，静的な固定値を使用しても良いでしょう．

以上の実装では，プレイヤエージェント自身のホームポジションに最も近い敵プレイヤをマーク対象としています．フォーメーションや役割配分，他の味方プレイヤとのバランス，マークによって生まれるスペースなどを考慮すると，より複雑な実装が必要になります．しかし，マーク対象プレイヤの決定を作り込みのルールだけで記述するのはかなり難しいようです．特に，他のプレイヤの識別が不十分な場合，味方プレイヤのマークを始めてしまうことがあります．より確実なマーク動作を実現するには，コミュニケーションを活用した情報共有が必要不可欠でしょう．

3.11.2 パスコースのブロック

パスコースをブロックし，敵チームの行動の選択肢を減らしていくことは重要な守備戦術です．しかし，パス動作はボールを持っているプレイヤが主導権を握るため，どれだけパスコースを潰したつもりでも，よほど組織的にバランスよくプレイヤを配置していなければ，パスコースをうまくブロックすることはできません．また，ボールは目まぐるしく動くため，そのボールの動きに合わせてブロックすべきパスコース上へと毎回移動するわけにもいきません．よって，この動作に関してもマーク動作と同様で，味方ゴール前での守備的状況での使用機会が多いでしょう．

マーク動作は味方ゴール前でのパスレシーブ動作を妨害することが目的でしたが，更にその前にパスコースを潰すことで敵チームの行動の選択肢を減らすことが目的になります．マーク動作と合わせて二重の守備を行うことで，より安全になります．その代わり，マークを実行するプレイヤに加えて，パスコースをブロックするプレイヤが新たに必要になります．マークとパスコースブロックとをどのように役割分担するかは，チームの戦略や戦術に依存します．

以下に，コーナー方向からゴール前へのパスコースをブロックするための移動位置決定アルゴリズムの実装例を示します．

```
const BallObject & ball = agent->world().ball();
Vector2D center( -41.5, 0.0 );
if ( ball.pos().x < -45.0 ) {
  center.x = std::max( -48.0, ball.pos().x + 1.0 );
}
Line2D block_line( wm.ball().pos(),
                   ( center - wm.ball().pos() ).th() );
Vector2D block_point;
block_point.y = std::max( 15.0, ball.pos().absY() - 10.0 );
if ( ball.pos().y < 0.0 ) block_point.y *= -1.0;
block_point.x = block_line.getX( block_point.y );
```

この例では，ゴール前中央の固定位置を設定し，その位置とボール位置とを結ぶ直線上にプレイヤエージェントが移動するようにしています．ブロック位置の Y 座標はボールから 10m 内側で，Y 軸からの最小距離は 15m としています．ボール位置の Y 座標値が 15m 以下の場合には，他のルールを採用しければならない点に役に注意してください．X 座標はその Y 座標値と直線の方程式から求められます．マジックナンバーをいくつか埋め込んでしまっているのであまり良い実装とは言えませんが，この程度の実装でも，ゴール前のパスのブロックにはかなりの効果を発揮します．

特定の敵プレイヤに対するパスコースをブロックする場合は，ボールとそのプレイヤとを結ぶ直線上に移動することになります．しかし，ボールもプレイヤもそれぞれが移動するので，ブロックのための移動目標位置は激しく変化します．そもそも，その直線上に移動したとしても他にもパスコースは存在するので，あまり意味がありません．パスコースのブロックにおいては，移動位置をある程度固定的に決定する方が良い結果をもたらすでしょう．

3.11.3 敵プレイヤ前方への先回り

敵プレイヤがドリブルを始めると，単純にインターセプト動作を実行しただけでは上手くボールを奪うことができません．最悪の場合，敵プレイヤのドリブルを後ろから追いかけているだけという状態になりかねません．そこで，敵プレイヤの前へ先回りし，妨害する動作が新しく必要になります．

実装例は以下のようになります．

```
const WorldModel & wm = agent->world();
const PlayerObject * opp
 = wm.getInterceptTable()->getFastestOpponent();
if ( ! opp ) {
  return;
}
Vector2D block_point = opp->pos();
if ( wm.self().pos().x > opp->pos().x ) {
  block_point.x -= 5.0;
} else {
  block_point.x -= 2.0;
}
Body_GoToPoint( block_point,
                1.0,   // 距離の誤差は 1m まで許容する
                ServerParam::i().maxPower(),
                5,     // 5 サイクル後に目標位置に到達するように
                false, // 後方ダッシュは使用しない
                true,  // recover を消費しない
                40.0   // 方向の誤差は 40 度まで許容する
                ).execute( agent );
```

この実装では，目標位置の導出方法が単純過ぎると思うかもしれません．確かに単純過ぎます 。しかし，中途半端に敵プレイヤの動きを予測したところで上手くいくものでもありません．かえってバグを仕込み，無駄な動きを増やしてしまいます．余計なことをするくらいなら，単純な方が良いのです．

とは言え，やはり無駄な回転動作は実行してしまうかもしれないので，一点だけ工夫します．それは，Body_GoToPoint による移動動作で，方向の誤差を大きめに設定することです．これによって，多少のずれは無視して目標位置へ向かうことができるので，無駄な回転動作を減らすことができます．

この先回り動作は，攻撃的役割のプレイヤに実行させると効果的です．後の節でも解説しますが，守備敵役割のプレイヤは複数人数で壁を作っておき，敵プレイヤの足を止めている間に攻撃敵役割のプレイヤは先回りしてボールを奪うという戦術を取ると，ボール奪取の成功率が向上します．

3.11.4 マークを外す動き

味方ゴール前で敵プレイヤをマークするのと同様に，敵ゴール前では味方プレイヤがマークされます．敵チームの戦術によっては，ゴール前以外でもマークされるかもしれません．マークを受けると，パスを受けるのが難しくなるだけでなく，パスを出す側のプレイヤもパスコースを発見できなくなってしまいます．そこで，敵プレイヤが密着してきたり，自分がこれから向かう先に既に敵プレイヤが存在している場合などは，移動目標位置をずらす必要があります．

動的な探索

移動目標位置をずらす方法として，フィールド上のポテンシャルを計算する方法が取られることが多いようです．ここでは，フィールドのポテンシャルとして，ある位置におけるプレイヤの密集度を求めます．密集度とは以下のような実装で得られる値とします．

```
Vector2D target_point( 45.0, 0.0 );
double congestion = 0.0;
const PlayerPtrCont::const_iterator
  end = agent->world().getOpponentsFromSelf().end();
for ( PlayerPtrCont::const_iterator
        o = agent->world().getOpponentsFromSelf().begin();
      o != opps_end;
      ++o ) {
  congestion += 1.0 / (*o)->pos().dist2( target_point );
}
```

この例では，target_point における敵プレイヤの密集度 congestion を求めています．複数の位置座標に対してこの計算を行い，congestion の値が最小になった位置を新しい目標位置とします．

さて，このアルゴリズムで良さそうに思えますが，実際にはそう上手くいきません．プレイヤエージェントの視野は制限されており，更に，他のプレイヤはそれぞれが自分の意思で移動しているため，新しい目標位置は常に変化するためです．このことは，プレイヤエージェントの移動動作に回転動作が多く含まれてしまい，結果，移動効率が低下することを意味します．

解決方法としては，一度目標位置を決めたら一定時間その位置への移動動作を継続する，という方法が考えられます．これは，制限時間付きの意図を登録することで容易に実現できます．

しかしながら，意図を利用することで移動動作の無駄を軽減したとしても，移動目標位置の動的な変更はそれほど効果を発揮しないようです．より確実な効果を期待するならば，次に説明する静的なルールを組み込む方法を採用すると良いでしょう．

静的なルールによる実現

戦術レベルの静的なポジショニングルールによってマークを外す動作を実現することもできます．この方法では，状況に応じた固定目標位置を静的に決めておき，移動すべき状況になればプレイヤエージェントはその位置への移動を開始します．正確に言うと，これは動的なポジショニングではなく，次節で説明する局所的な状況のための戦術的なポジショニングなのですが，便宜上，ここで説明します．

例えば，プレイヤエージェントの役割がサイドのフォワードの場合，以下のような実装によって，マークを振り切ってゴール前に移動できることが多くなります．

```
const WorldModel & wm = agent->world();
Vector2D target_point = home_pos;
if ( wm.ball().pos().y * wm.self().pos().y < 0.0
     && wm.ball().pos().x > 40.0
     && 6.0 < wm.ball().pos().absY()
     && wm.ball().pos().absY() < 17.0 ) {
  Rect2D goal_area( Vector2D( 52.5 - 8.0, -6.0 ),
                    Vector2D( 52.5, 6.0 ) );
  if ( ! wm.existTeammateIn( goal_area, 10, true ) ) {
    target_point.x = wm.ball().pos().x + 2.0;
    if ( target_point.x > wm.getOffsideLineX() - 1.0 ) {
      target_point.x = wm.getOffsideLineX() - 1.0;
    }
    if ( target_point.x > 49.0 ) target_point.x = 49.0;
    target_point.y = -1.0;
    if ( wm.ball().pos().y > 0.0 ) target_point.y *= -1.0;
  }
}
```

この実装例は，ボールが逆サイドの特定領域にある場合に移動目標位置を敵ゴール中央付近へと強制的に変更するルールとなっています．このままではマジックナンバーばかりで汚い実装なので，実際の利用においてはもう少し工夫すべきです．

このようなルールを組み込むことで，プレイヤエージェントの動作に無駄が無くなり，マークを外すのに成功しやすくなります．しかし，移動目標位置を静的に決定するにはチームの戦略や戦術との調整が必要不可欠であるため，一度チームを調整してしまうと移動目標位置の変更が難しくなってしまうという欠点もあります．また，移動した先に敵プレイヤが存在する場合には全く役に立ちません．実際には，固定目標位置を複数用意しておき，それらの中から選択する，という工夫が必要でしょう．

3.11.5 体の向きの調節

プレイヤエージェントの移動動作において，既に目標位置に到達しており特にやるべきことが無い場合には，更に体の向きを調節しておくと次のプレイで有利になります．

攻撃的役割のプレイヤの場合は，敵ゴールライン方向に体を向けるだけで充分です．これは，以下のような実装で簡単に実現できます．

```
if ( ! Body_GoToPoint( target_point, dist_thr, dash_power
                       ).execute( agent ) ) {
  Vector2D face_point( 100.0, wm.self().pos().y );
  Body_TurnToPoint( face_point ).execute( agent );
}
```

更に，攻撃的役割のプレイヤが移動目標位置から少しだけ前に出過ぎていた場合は，体の向きを変えずに後方ダッシュによる位置調整を行うと効果的です．

```
if ( wm.self().pos().x > target_point.x + dist_thr*0.5
     && std::fabs( wm.self().pos().x - target_point.x ) < 3.0
     && wm.self().body().abs() < 10.0 ) {
  double back_dash_power
    = wm.self().getSafetyDashPower( -dash_power );
  agent->doDash( back_dash_power );
}
```

守備的役割のプレイヤの場合は，インターセプト動作に有理になるような工夫が必要です．例えば，プレイヤエージェントがディフェンスラインを構成する役割の場合は，左右に動いて敵プレイヤの進路を妨害する動きが必要になるため，以下のように実装します．

```
if ( ! Body_GoToPoint( target_point, dist_thr, dash_power
                       ).execute( agent ) ) {
  Vector2D face_point( wm.self().pos().x, 0.0 );
  face_point.y = ( wm.ball().pos().y < wm.self().pos().y
                   ? -40.0 : 40.0 );
  Body_TurnToPoint( body_point ).execute( agent );
}
```

また，ボールをいきなり蹴られても対処しやすいように，ボール方向に対して垂直に体を向けておくことも必要になるでしょう．これは敵チームのセットプレイにおいて効果を発揮します．

```
if ( ! Body_GoToPoint( target_point, dist_thr, dash_power
                       ).execute( agent ) ) {
  AngleDeg body_angle = wm.ball().angleFromSelf();
  if ( body_angle.degree() < 0.0 ) body_angle -= 90.0;
  else body_angle += 90.0;
  Vector2D face_point = wm.self().pos();
  face_point += Vector2D::polar2vector( 10.0, body_angle );
  Body_TurnToPoint( face_point ).execute( agent );
}
```

ただし，いずれの場合もボールの位置を確認ができるように考慮しておかなければなりません．よって，これらの体の向きを調節する動作は，首を振るだけでボールの位置を確認できる場合にのみ実行すべきです．

Section 3.12
局所的状況でのポジショニング

チームとしてのフォーメーションを維持するだけでは，チャンスにもピンチにも適切に対処することが難しくなります．ある特定の状況においては，チーム全体のフォーメーションを基本にしつつも特殊な配置状態を意図的に作り出すことが要求されます．前節で説明した静的なルールによるマークを外す動作は，このような局所的状況でのポジショニングの副産物です．

3.12.1 中盤でのディフェンスライン

中盤における守備では，早い段階でボールを取り返すことが最大の目的となります．そのためには，相手の攻撃を遅れさせ，パスカットやドリブルを妨害する機会を増やすことが重要です．しかし，中盤でのマーク動作は，相手プレイヤにあわせた移動では移動距離があまりに大きくなるため，スタミナ管理の点から効率的ではありません．相手の攻撃を遅れさせるには，マークよりもボールを持った敵プレイヤの進路を塞ぐような動きを優先すべきです．そのためには，ディフェンダのプレイヤは敵プレイヤの前に壁を作るように移動しなければなりません．

このような動作は，適切なフォーメーションが設定されていれば，そのフォーメーションを維持するだけでもほぼ実現できます．しかし，本書におけるフォーメーションとは，ボール位置を中心にしたポジショニングシステムであるため，ボールの位置が下がればそれぞれのプレイヤの位置も下がってしまいます．これを解決するためには，ディフェンスラインを構成するプレイヤの移動目標位置のX 座標をある程度固定的にします．

通常，フォーメーションによる各プレイヤの移動位置は，入力をボール位置座標とした連続な関数からの出力と考えられます．以下のような実装によって，出力である移動位置の X 座標を階段状に変更させることができます．ここでは，home_pos をフォーメーションによる移動位置とします．

```
const WorldModel & wm = agent->world();
Vector2D target_point = home_pos;
if ( wm.self().pos().x > home_pos.x
     && wm.ball().pos().x > wm.self().pos().x + 3.0 ) {
  double tmpx = std::floor( wm.ball().pos().x * 0.1 ) * 10.0 - 3.0;
  if ( tmpx > 0.0 ) tmpx = 0.0;
  if ( tmpx > home_pos.x + 5.0 ) tmpx = home_pos.x + 5.0;
  target_point.x = tmpx;
  if ( wm.getDefenseLineX() < target_point.x - 1.0 ) {
    target_point.x = wm.getDefenseLineX() + 1.0;
  }
  target_point.y = home_pos.y;
  double home_y_diff = wm.ball().pos().y - home_pos.y;
  target_point.y += home_y_diff * 0.1;
}
```

この実装例では，X 座標の $10m$ ごとにディフェンスラインを想定しており，ボールがそのラインよりも下がらないうちはそのライン上に目標移動位置を設定します．更に，ボール位置との Y 座標の差を少しだけ値縮めることに寄って，自然にボールの前に集まるようにしています．

ディフェンスラインを構成するプレイヤエージェントにこのような移動ルールを設定することで，全体としては敵プレイヤの進路を妨害する壁を作るような動きになります．すると，敵プレイヤは動きを鈍らさざるを得ないため，中盤や前線の味方プレイヤが戻ってくる時間を稼げるようになり，人数をかけてのボール奪取の機会を増やすことが出来ます．

ディフェンダプレイヤはあまり積極的にボールを奪おうとすべきではありません．もちろん，1 対 1 でボールを奪うだけのスキルを持っていれば挑戦しても良いのでしょう．しかし，通常，ボールを持っている側の方がプレイの主導権を握るため，ほんの 1 サイクルでもずれが生じれば，敵プレイヤによるディフェンスラインの突破を許してしまうかもしれません．ディフェンダプレイヤはより安全で確実性の高い意思決定を行うべきなのです．

3.12.2 味方ゴールのブロック

味方ゴール前に攻め込まれた場合，守備的役割のプレイヤは味方ゴール前に集まらなければなりません．それだけで無く，敵プレイヤによるシュートコースを塞ぐような位置に移動しておくことが重要です．特に，キーパー人では対処しきれないボール隅を塞ぐような位置に移動しておくと，敵プレイヤによるシュートを止められることがあります．このような動作は，サイドのディフェンダプレイヤが実行することになります．

実装例は以下のようになります．

```
Vector2D block_point( -49.0,
                      ServerParam::i().goalHalfWidth() - 3.0 );
if ( wm.self().pos().x < -45.0 ) {
  if ( wm.ball().pos().y * home_pos.y < 0.0 // ボールが逆サイド
       && wm.existKickableOpponent()
       && wm.ball().pos().x < -40.0
       && 5.0 < wm.ball().pos().absY() ) {
    block_point.y = 0.5;
  }
  if ( wm.ball().pos().y * home_pos.y < 0.0 // ボールが同じサイド
       && wm.existKickableOpponent()
       && wm.ball().pos().x < -40.0
       && 6.0 < wm.ball().pos().absY()
       && wm.ball().pos().absY() < 12.5 ) {
    block_point.y = ServerParam::i().goalHalfWidth() - 1.5;
  }
}
if ( home_pos.y < 0.0 ) block_point.y *= -1.0;
```

ルールの条件部がやや複雑になっていますが，これはチームの戦術やフォーメーションのバランスに応じて調節が必要です．一見してわかるように，かなり汚い実装となっています．わざわざルールで書かなくとも，フォーメーションによるポジショニングシステムとして実現できれば良いのですが，このような微調整が必要な動作はルールとして作り込まざるを得ないのが現状です．

3.12.3　敵ゴール前の布陣

一般的なサッカーのフォーメーションでは，フォワードのプレイヤが 1 人から 3 人程度存在します．現在のシミュレーションサッカーにおいてもこれはほぼ同様で，敵ゴール前に攻め込んだ場合，フォワードプレイヤの働きが重要になります．しかし，近年は組織的な守備が非常に発達してきており，高々3 人程度の連携プレイでは点を取ることが難しくなっています．ペナルティエリア内に守備のプレイヤが 7 人以上いることも珍しくなく，ひどいチームになると，5 人のディフェンダがほぼ完全にゴール前を塞いでしまっていることもあります[2)]．このような状況を打開するには，攻撃側も人数を増やし，数で対抗しなくてはなりません．敵ゴール周辺に最低でも 5 人は集めなければ，得点のチャンスを掴むことは難しいでしょう．

しかし，人数を集めただけでは役に立ちません．ゴールを狙うチャンスがより増えるように味方プレイヤを配置しなければなりません．味方プレイヤを単に横に並べるだけでは簡単にパスコースをブロックされてしまうので，各プレイヤの移動目標位置 X 座標をずらしおくと効果的です．これは，元々フォワードのプレイヤはより前方に，ミッドフィールダのプレイヤはやや後方に配置することで自然に実現できます．

ここで，最も重要になるのは中央付近でポストプレイを行うプレイヤの位置です．現在の rcssserver の仕様では，ポストプレイを行うために適切な位置はほぼ決定してしまっています．位置座標としては，両ゴールポストから 6m 程度手前の (46.0, ±8.0) 付近になります．rcssserver におけるキーパの能力の性質上，ゴールライン際深くまでボールを運ばれると，キーパはゴールポスト近くに張りつかざるを得ません．そのキーパからほんの少しだけ離れたこの位置に陣どることによって，サイドからのパスを受けた直後に直接シュートを狙えるだけなく，更に逆サイドにボールを振るチャンスも生まれます．

逆に考えると，味方ゴール前での守備においてもこの位置は重要です．この位置にスペースが出来たとき，失点することが多いはずです．

---

2) ゴール前を常に塞ぐことは反則ですが，常時でなければ反則とは取られないようです．

Section 3.13
キーパのポジショニング

キーパの戦術としては，積極的に前に出る，または，最小限の動きで堅実にセーブする，という二種類のものがあります．積極的な動きの方が実現は難しくなります．隙が生まれないように非常に精度の高い状況判断能力が要求されるためです．しかし，実際にはそのような状況判断能力の実装は非常に困難ですキーパの実装に充分な時間をかけられないのであれば，最小限の動きにとどめておいた方が無難です．本書でも，最小限の動きについてのみ説明します．

3.13.1 アルゴリズム

キーパのボールキャッチ能力は非常に強く，ボールをキャッチ可能な距離にとらえれば 100%キャッチできます．そして，rcssserver の仕様上，プレイヤエージェントは体の方向に対して前後にしか加速することが出来ません．この 2 点から，ゴール前でゴールラインに対して並行に体を向け，普段は左右に移動するだけ，という動作が最も効率的であることが分かります．

アルゴリズムは以下のようになります．

1. 目標位置に到達していれば，その位置を維持できるように自分自身の速度を抑える．
2. 危険な状況であれば，多少の X 座標のずれは無視して通常の移動を実行する．
3. 移動目標位置との X 座標のずれが大きければ，通常の移動を実行する．
4. 体の向きを修正する．
5. ボール位置座標に合わせて，前後に移動する．

3.13.2 移動位置の求め方

いくらキーパのキャッチ能力が高くとも，移動すべき位置が適切でなければボールに追い付くことが出来ません．この目標移動位置の計算アルゴリズムとしてい

くつかの方法がありますが，ここではその一例を示します．

実装例は以下のようになります．

```
const WorldModel & wm = agent->world();
int ball_reach_step = 0;
if ( ! wm.existKickableTeammate()
     && ! wm.existKickableOpponent() ) {
  ball_reach_step
    = std::min( wm.getInterceptTable()->getTeammateReachCycle(),
                wm.getInterceptTable()->getOpponentReachCycle() );
}
const Vector2D base_point( -ServerParam::i().pitchHalfLength() - 6.0,
                           0.0 );
Vector2D ball_point;
if ( wm.existKickableOpponent() ) {
  ball_point = wm.ball().pos();
} else {
  ball_point = wm.ball().inertiaPoint( ball_reach_step );
}
Line2D ball_line( ball_point, base_point );
double move_y = ball_line.getY( -50.5 );
if ( move_y > ServerParam::i().goalHalfWidth() - 0.5 ) {
  move_y = ServerParam::i().goalHalfWidth() - y_buf;
}
if ( move_y < -ServerParam::i().goalHalfWidth() + 0.5 ) {
  move_y = -ServerParam::i().goalHalfWidth() + y_buf;
}
Vector2D target_point( base_move_x, move_y );
```

この実装では，予測ボール位置とある固定位置とを結ぶ直線上を維持するように移動位置が求められています．固定位置として，ゴールよりも更に後ろの位置座標が設定されています．マジックナンバーが大量に埋め込まれていますが，実際には変更可能なパラメータとして実装すべきであることに注意してください．

この実装はあくまで一例であることにも注意してください．これが最適なアルゴリズムというわけではありません．より適切な移動位置を獲得するには，何らかの近似アルゴリズムによる関数の獲得が必要でしょう．

3.13.3 危険な状況への対処

敵プレイヤがゴールポスト近くまでドリブルで進入してきた場合，ゴールポストとキーパの間をすり抜けるようなシュートを打たれ，それが決まってしまうことがあります．キーパのキャッチ可能範囲とゴールとの隙間を狙われた場合，キーパの体の向きが適切でなければ対処が遅れてしまうためです．また，キーパのキャッチ可能範囲をボールが通っている場合でもゴールが決まってしまうことがあります．これは，rcssserver が離散時間シミュレータであることが原因で起こる現象です．幾何学的にはキャッチか納涼いき上を通過していたとしても，サイクル更新時にボールがキャッチ可能領域内に無ければボールをキャッチすることは出来ないのです．

よって，ボールがゴールライン近くに存在する場合は通常よりもゴールラインよりに移動しなければなりません．更に，シュートに素早く反応できるように，ゴールラインと垂直に近い方向に体の向きを変更すべきです．

以下に移動位置の計算の実装例を示します．

```
double sign = ball_pos.y > ServerParam::i().goalhalfWidth() + 3.0
              ? 1.0 : -1.0;
Vector2D pole( -ServerParam::i().pitchHalfLength(),
               sign * ServerParam::i().goalHalfWidth() );
AngleDeg angle_to_pole = ( pole - ball_pos ).th();
if ( sign > 0.0
     && -140.0 < angle_to_pole.degree()
     && angle_to_pole.degree() < -90.0 )
   || ( sign < 0.0
        && -90.0 < angle_to_pole.degree()
        && angle_to_pole.degree() < 140.0 )
{
  return Vector2D( danger_move_x,
                   sign * ( ServerParam::i().goalHalfWidth() + 0.5 ) );
}
if ( ball_pos.x < -ServerParam::i().pitchHalfLength() + 8.0
     && ball_pos.absY() > ServerParam::i().goalHalfWidth() + 2.0 ) {
  Vector2D target_point( base_move_x,
                         ServerParam::i().goalHalfWidth() + 0.2 );
  if ( ball_pos.y < 0.0 ) {
    target_point.y *= -1.0;
  }
  return target_point;
}
```

Section 3.14
ボール所有時の意思決定

プレイヤエージェントがボールを持っているとき，実行できる動作はシュート，パス，ドリブル，キープ，クリアなどであり，動作の種類の選択肢はそれほど多くありません．しかし，それぞれの動作のパターンは無数に生成できるため，動作の種類だけでなく，それらの中からどのパターンを実行するかも決定しなけれ

ばなりません.

3.14.1 シュートの評価

本書執筆時点の librcsc では，シュートコースの生成と評価が暫定的にですが実装されています．暫定的とは言っても，2005 年度の TokyoTechSFC で実際に使用していたものなので，それなりの精度は実現されています．シュートコースの生成方法については 節を参照してください.

librcsc では，シュートコースを以下の二項目で評価してします.

- ボールの初速度の大きさ
- 敵プレイヤがインターセプトに要するサイクル

しかし，これだけでは評価基準としては不十分です．他にも，ゴールに入る確率を高めるために，ゴール中央を狙うシュートコースを優先するなどの評価基準が必要です．実際に，ゴール隅を狙ったものの観測誤差やボール動きのの伊豆などのためにゴールからはずれてしまうことはしばしばあります．UvA Trilearn というチームはシュートコースの統計を取り，具体的な数値として評価を得る実験を行っています．シュートは得点に絡む重要な動作であるため，より高い精度を得るための改良が必要でしょう.

3.14.2 パスの評価

本書執筆時点の librcsc では，パスコースの生成と評価が暫定的にですが実装されています．パスコースの生成方法については 節を参照してください.

生成されたパスコースが成功する，すなわち，敵プレイヤに妨害されず味方プレイヤに渡ると予測されれば，次に，それらの中から適切なものを選択しなければなりません．これは，非常に解決の難しい問題です．人間であれば，チームの戦略や戦術だけでなく，試合の状況や現在のプレイヤの配置状態までも考慮して，直感的に選択してしまうところですが，プログラムではそうはいきません.

本書執筆時点の librcsc におけるパスの実装では，以下のような評価基準に基づいてパスコースをスコア付けし，最も高い評価を得たパスコースを選択するようにしています.

- レシーブ位置と敵プレイヤとの距離

- レシーブ位置の x 座標
- レシーバの移動距離
- レシーバの位置の信頼性
- パスコースの方向の信頼性
- レシーブ位置前方にスペースがあるか否か

この実装は，`Body_Pass::evaluate_routes()` で見つけることが出来ます．しかし，この実装は筆者の勘による調整と繰り返しのテストの結果できあがったものに過ぎず，適切な評価を行えているとは言えません．例えば，戦略や戦術を全く考慮できていません．

解決方法としては，何らかの近似アルゴリズムによって多次元入力の関数を獲得させる方法が良いのではないかと思いますが，評価基準のブラックボックス化や，計算資源の浪費が予想されます．パスコースの評価は，今後も検討が必要な課題です．

3.14.3　ドリブルの方向

ドリブルに関しては，敵プレイヤの回避動作が考慮されていれば，敵ゴール方向かそのまま前方に進むかのいずれかを選択すれば充分です．敵プレイヤの回避はライブラリレベルで実装済なのであまり考慮する必要はありませんが，ドリブルを実行するか否かの判断についてはチーム開発者の責任です．すぐ目の前に敵プレイヤがいるような状況では，ほとんどの場合そもそもドリブルを実行すべきではありません．

そこで，ドリブルの進行方向に敵プレイヤが存在するかどうかの判断をまず行います．簡易な計算方法として，ドリブル方向に扇型の領域を作り，その中に敵プレイヤが存在するかどうかで判断する実装例を以下に示します．

```
const WorldModel & wm = agent->world();
Vector2D drib_target( 50.0, 0.0 );
AngleDeg drib_angle = ( drib_target - wm.self().pos() ).th();
const Sector2D sector( wm.self().pos(),
                       0.5, 15.0,
                       drib_angle - 30.0, drib_angle + 30.0);
if ( ! wm.existOpponentIn( sector, 10, true ) ) {
  Body_Dribble( drib_target, 1.0,
                ServerParam::i().maxPower(), 5
              ).execute( agent );
}
```

この例では，敵ゴール中央付近を目標位置とし，その方向に左右 30 度以内，距離 0.5 から 15m の範囲に敵プレイヤが存在しなければドリブルを実行する，というじっそうになっています．実際にはもう少し細かい条件の確認が必要です．また，非常に簡易なアルゴリズムであるため，精度は当てになりません．ドリブルに関しても，パスやシュートと同様に厳密に計算することは可能なので，計算資源に余裕があれば，可能な限り厳密に予測すべきです．

ドリブルの目標方向または目標位置の決定方法としては，プレイヤエージェントの体の向きを優先させる方法も考えられます．ドリブル開始時に回転動作が入るとそれだけボールを奪われる危険が大きくなるため，戦術的な目標方向との角度のずれが小さければそのままドリブルを実行してしまった方が効率が良くなります．

以下に実装例を示します．

```
if ( wm.self().body().abs() < 70.0 ) {
  const Vector2D drib_target
    = wm.self().pos()
      + Vector2D::polar2vector( 5.0, wm.self().body() );
  if ( drib_target.x < ServerParam::i().pitchHalfLength() - 1.0
       && drib_target.absY()
          < ServerParam::i().pitchHalfWidth() - 1.0 ) {
    const Sector2D sector( wm.self().pos(),
                           0.5, 10.0,
                           wm.self().body() - 30.0,
                           wm.self().body() + 30.0 );
    if ( ! wm.existOpponentIn( sector, 10, true ) ) {
      Body_Dribble( drib_target, 1.0,
                    ServerParam::i().maxPower(), 2
                  ).execute( agent );
    }
}
```

この例では，ドリブル目標方向はゴールライン方向，すなわち 0 度の方向を想定しています．自分自身の体の方向の延長線上に目標位置を設定し，そこに向かってのドリブルとなります．角度のずれは ±70 度 まで許容するので，ほとんど横に向かって進むこともありますが，それが良い結果を生むこともしばしばあります．ドリブルという動作では，とにかく前へボールを運ぼうとすることが重要です．

3.14.4　緊急回避のためのキック

敵プレイヤに囲まれてしまい，シュート，パス，ドリブルのいずれも実行できず，その場でのキープも難しくなった場合，残る選択肢はクリアです．ただし，クリアと言っても，状況によってその性質を変化させるとより効果的です．

まず，守備的状況の場合は普通にクリアすることが望ましいでしょう．この場合，ボールを蹴り出す目標方向は安全であれば安全であるほど良い結果が得られます．そして，ボールの初速度の大きさは可能な限り大きくすべきです．

次に，中盤から攻撃的状況の場合は，クリアでは無く次の展開をを考慮した位置ボールを蹴り出すと，結果的に有利な展開になることがあります．特に，敵陣側のフィールドコーナーに向かってボールを蹴ると良い結果になることが多くあ

ります．ペナルティエリア内に向かって蹴っても敵キーパが簡単にキャッチしてしまいますが，コーナーであればよほど積極的なキーパでない限り手は出しません．また，敵ディフェンダプレイヤも用心のためにゴール前へ戻ることが多いので，結局味方ボールのまま展開が続行されやすいようです．

コーナーに向かってボールを蹴る場合は，クリアと違ってボールの初速度の大きさを調整します．コーナー直前でボールが止まるような強さが理想的です．以下に実装例を示します．

```
const WorldModel & wm = agent->world();
Vector2D left_corner( ServerParam::i().pitchHalfLength() - 10.0,
                      -ServerParam::i().pitchHalfWidth() + 8.0 );
Vector2D right_corner( ServerParam::i().pitchHalfLength() - 10.0,
                       ServerParam::i().pitchHalfWidth() - 8.0 );
Vector2D target_point = wm.self().pos().y < 0.0
                        ? left_corner : right_corner;
double first_speed
  = calc_first_term_geom_series_last
    ( 0.1,
      agent->world().ball().pos().dist( target_point ),
      ServerParam::i().ballDecay() );
first_speed = std::min( first_speed, ServerParam::i().ballSpeedMax() );
Body_KickMultiStep( target_point,
                    first_speed,
                    false  ).execute( agent );
```

この実装では，プレイヤエージェント自身がフィールドのゴールよりも左側にいれば左コーナーへ，逆なら右コーナーへとボールを蹴ります．そして，目標位置に到達したときにボールのスピードが 0.1 になるように初速度の大きさを求めます．初速度の大きさはは，librcsc が提供する `calc_first_term_geom_series_last` という関数で求めることが出来ます．

この例では敵プレイヤの存在を全く考慮していない点に注意してください．実際の使用においては，ボールの軌道上に敵プレイヤが存在するか否かの確認や，必要であれば目標位置の修正なども行わなければなりません．

3.14.5　キックの競合

他のプレイヤと同時にボールをキック可能な状態になった場合，相手に応じた適切な対処が必要となります．相手が味方プレイヤの場合は，お互いの動作を邪魔しないように，どちらがボール所有者であるかを推定しなければなりません．相手が敵プレイヤであれば，強制的なキックを実行しなければそのままボールを取られてしまうかもしれません．

いずれの場合も，同時にキック可能になったか否かはあくまで観測に基づく推定の結果でしかないので，推定結果が誤っていれば望む結果が得られないことに注意してください．

相手が味方プレイヤの場合

単純に，よりボールに近い方をボール所有者と判断すれば充分です．ボールとの距離がほとんど同じある場合や互いに味方であると認識できていない場合にはうまくいきませんが，視覚による観測だけではどうしようもありません．より確実にボール所有者を決めたい場合は，コミュニケーションの利用が必須です．

相手が敵プレイヤの場合

最も確実なのは，敵ゴール方向に向かって最大パワーでキックを実行することです．相手のキックによっては上手くこぼれ玉になるかもしれず，また，相手がキックを実行しなければただのシュートになるので非常に安全です．

以下に実装例を示します．

```
if ( agent->world().getGameMode().type() == GameMode::PlayOn
     && ! agent->world().self().goalie()
     && agent->world().self().isKickable()
     && agent->world().existKickableOpponent() ) {
  Vector2D goal_pos( ServerParam::i().pitchHalfLength(), 0.0 );
  if ( agent->world().self().pos().x > 36.0
       && agent->world().self().pos().absY() > 10.0 ) {
    goal_pos.x = 45.0;
  }
  Body_KickOneStep( goal_pos,
                    ServerParam::i().ballSpeedMax()
                  ).execute( agent );
}
```

このルールをプレイヤエージェントの意思決定処理の先頭近くに置くように注意してください．さもなければ，キック動作のたびに逐一キックの競合を確認して上記コードを挿入しなければならなくなります．

3.14.6 キック動作の選択

キック動作の選択はチームの戦術の大部分を決定します．そのためには，パス主体かドリブル主体か，また，プレイヤによってどの動作を優先的に選択するかといった問題を解決しなければなりません．

キック動作の種類は，シュート，パス，ドリブル，キープ，クリアの 5 つです．これらの中から，シュートとクリアを他の動作と比較する必要はありません．シュートについては，成功すると予測されれば常に最優先で実行すべきです．そのため，プレイヤエージェントがボールをキック可能であれば，シュートの成功判定をまず第一に実行します．そして，パス，ドリブル，キープの 3 つの動作について予測結果を評価し，いずれも実行不可能だった場合の最終手段としてクリアを実行することになります．

パス，ドリブル，キープの 3 種からの選択に関する意思決定は，チームの戦略や戦術に大きく依存します．また，各動作内でも目標位置などの選択が必要であり，二段階の意思決定が要求されます．チーム開発を柔軟に行うには，これら 3 種の動作とそのパターン全てを同列に比較できることが望ましいのですが，本書執筆時点の librcsc ではそのような比較ができるような実装になっていません．非

常に泥くさいやり方ですが，条件分岐を細かく作り込むことでチームを作り上げるようになっています．

今後は，より柔軟な設計へと改良していく予定ですが，入門用としてはルールで作り込むやり方を体験しておくことも必要だと思うので，一度は作りこんでみることを推奨します．人手で作りこむことによって，人手では何が出来て何が出来ないのかがはっきりと見えるようになってきます．

Section 3.15
戦術的なタックル

プレイヤエージェントは，キックだけでなくタックルによってもボールへ加速度を与えることができます．タックルの有効範囲は通常のキックよりも大きいので，敵プレイヤがボールをキープしている場合には非常に有効です．タックルについての詳細は，4.2.5 節を参照してください．

3.15.1 タックルすべき状況

タックルを実行すべき状況とは，具体的には以下の条件を満たした場合です．

- プレイヤエージェントはボールをキックできない
- プレイヤエージェントのタックル成功確率が 0 よりも大きい
- 敵プレイヤがボールをキック可能な状態である，または，敵プレイヤの方が先にボールに追い付くと予測される

プレイヤエージェントの移動動作に関する意思決定では，ボール所有者の推定の直後にタックルを実行すべきかどうかを判断します．

3.15.2 タックル成功確率の考慮

ボールとプレイヤエージェントとの位置関係に応じてタックルの成功確率が決まります．rcssserver 内部では厳密に計算されていますが，プレイヤエージェント

が得られるタックル成功確率は，プレイヤエージェント自身が観測した情報に基づいた推定値でしかないことに注意してください．

タックル成功確率の推定値は，`SelfObject::tackleProbability()` で得られます．成功確率が 0 のときにタックルをしても全く意味がありません．成功確率が 0 よりも大きければタックルが成功する可能性があります．しかし，成功する可能性があるからといって闇雲にタックルに挑戦すべきではありません．タックルを実行すると，成功不成功に関係なく 10 サイクル動けなくなるという制約があるためです．

攻撃的役割のプレイヤであればこのペナルティを考慮する必要はあまりありませんが，守備的役割のプレイヤは味方ゴール前で成功確率の低いタックルに挑戦すべきではありません．経験的には，`SelfObject::tackleProbability()` の値が 0.8 程度以上であれば，どのような状況でもタックルに挑戦しても良いようです．成功確率が 0.9 以上となると，ボールがほぼキック可能領域内に入ってしまっているのであまり意味がありません．

3.15.3 タックル方向の選択

タックルは，プレイヤエージェントの体の方向にボールの加速度を生成する動作です．**tackle** コマンドに与えるパワーの符号を負にすると，逆方向への加速度をせいせいすること賀できます．つまり，タックルには方向の選択肢が 2 つしかありませんが，どちらを選択するかは戦術的に極めて重要です．

基本的な考え方としては，敵陣方向に近い方を選択すべきですすなわち，プレイヤエージェントの体の向きが $[-90, 90]$ であれば前方 (正のパワー) を，それ以外であれば後方 (負のパワー) を選択します．

実装例は以下のようになります．

```
double min_prob = 0.75;
double body_thr = 80.0;

const WorldModel & wm = agent->world();
if ( wm.existKickableOpponent()
     && wm.self().tackleProbability() > min_prob ) {
   double tackle_power = ServerParam::i().maxPower();
   if ( wm.self().body().abs() < body_thr ) {
     agent->doTackle( tackle_power );
     agent->setNeckAction( new Neck_TurnToBallOrScan() );
   } else if ( wm.self().body().abs() > 180.0 - body_thr ) {
     agent->doTackle( -tackle_power );
     agent->setNeckAction( new Neck_TurnToBallOrScan() );
   }
}
```

この例では，タックル成功確率が 0.75 以上で，体の向きの絶対値が 80 度以上であれば前方へのタックル，体の向きの絶対値が 100 度以上であれば後方へのタックルを実行しています．タックルのパワーは常に最大パワー (`ServerParam::i().maxPower()`) です．タックルにおいて，パワーを調整する必要は全くありません．

中盤であればこれでも充分ですが，特に味方ゴール前においては工夫の余地があります．以下に実装例を示します．

```
const WorldModel & wm = agent->world();
if ( wm.ball().pos().absY()
     > ServerParam::i().goalHalfWidth() + 5.0 ) {
  double power = ServerParam::i().maxPower();
  if ( wm.self().body().degree() * wm.ball().pos().y < 0.0 ) {
    power *= -1.0;
  }
  agent->doTackle( power );
} else {
  double power_sign = 0.0;
  if ( wm.ball().pos().y > 0.0 ) {
    if ( 0.0 < wm.self().body().degree()
         && wm.self().body().degree() < 90.0 ) {
      power_sign = 1.0;
    }  else if ( -180.0 < wm.self().body().degree()
                 && wm.self().body().degree() < -90.0 ) {
      power_sign = -1.0;
    }
  } else {
    if ( -90.0 < wm.self().body().degree()
         && wm.self().body().degree() < 0.0 ) {
      power_sign = 1.0;
    } else if ( 90.0 < wm.self().body().degree()
                && wm.self().body().degree() < 180.0 ) {
      power_sign = -1.0;
    }
  }
  if ( power_sign != 0.0 ) {
    agent->doTackle( power_sign * ServerParam::i().maxPower() );
  }
}
```

この例では，以下のルールでタックルの方向を決定しています．

- 味方ゴール前から少しずれた範囲であれば，ボールを中心としてゴールと逆のサイドライン方向へ．

- 味方ゴール前であれば，敵陣方向へかつゴールから遠ざかる方向へ

Section 3.16
戦術的な情報収集

プレイヤエージェントは，受信した視覚情報を分析することで周囲の状態を認識します．しかし，rcssserver 上のプレイヤの視界の範囲は制限されているため，必要に応じて視界の広さや方向を変化させ，積極的に情報収集を行わなければなりません．

プレイヤの視界の広さについては 6.4.3 節で，視界方向の変更については 4.2.8 節で，それぞれその仕様を説明しています．不明な点があれば，そちらを参照してください．

3.16.1 視界モードの変更

rcssserver 上のプレイヤは，視界の広さとして 45 度，90 度，180 度の 3 タイプを使用できます．視界を広くすると視覚情報の受信頻度が下がり，逆に，視覚情報の受信頻度を高めると視界は狭くなります．この特性を理解した上で状況に応じて視界の広さを変更させると，プレイヤエージェントの状況認識の精度が高まり，より高いパフォーマンスを発揮できるようになります．

例えば，ボールが近くに存在する場合などは常にボールを監視してその位置を把握しておかなければなりません．さもなければ，他のプレイヤによってボールが蹴られたときにすぐに反応することができません．よって，ボールが近くに存在すれば，視野を狭めて見るべき対象をボールに絞りこむ必要があります．逆に，ボールとの距離が大きい場合や，ボールがプレイヤエージェントの背後にあるために視界内に入れることが不可能な場合には，視界を広げて一度に得られる情報を多くするのが良いでしょう．

以下に実装例を示します．

```
if ( ! agent->world().ball().posValid() ) {
  return agent->doChangeView( ViewWidth::WIDE );
}
const double ball_dist = agent->world().ball().distFromSelf();
if ( ball_dist > 40.0 ) {
  return agent->doChangeView( ViewWidth::WIDE );
}
if ( ball_dist > 20.0 ) {
  return agent->doChangeView( ViewWidth::NORMAL );
}
if ( ball_dist > 10.0 ) {
  AngleDeg ball_angle
    = agent->effector().queuedNextAngleFromBody
                       ( agent->effector().queuedNextBallPos() );
  if ( ball_angle.abs() > 120.0 ) {
    return agent->doChangeView( ViewWidth::WIDE );
  }
}
double teammate_ball_dist = 1000.0;
double opponent_ball_dist = 1000.0;
if ( ! agent->world().getTeammatesFromBall().empty() ) {
  teammate_ball_dist
    = agent->world().getTeammatesFromBall().front()->distFromBall();
}
if ( ! agent->world().getOpponentsFromBall().empty() ) {
  opponent_ball_dist
    = agent->world().getOpponentsFromBall().front()->distFromBall();
}
if ( teammate_ball_dist > 5.0 && opponent_ball_dist > 5.0
     && ball_dist > 10.0 ) {
  return agent->doChangeView( ViewWidth::NORMAL );
}
return View_Synch().execute( agent );
```

まず最初に，ボールの位置を見失っている場合は視界を ***wide*** モードにして，ボールを探索する準備をして終了しています．次に，単純にボールとの距離で視

界の広さを切替えています．ボールとの距離が 10m から 20m の間では，背後にあるボールを観測するために，やはり ***wide*** モードに変更します．最後に，他のプレイヤとボールとの距離を調べ，ボールがすぐに蹴られることがなさそうであれば ***normal*** モードに変更します．それ以外の場合は，同期モードに変更します．同期モードについては**??**

3.16.2 視界方向の変更

周囲の情報を収集するために，プレイヤエージェントは首振り動作によって視界方向を適切に変更しなければなりません．ほとんどの場合，周囲を広く見渡し続けておけば充分なので，`Neck_TurnToBallOrScan|` ，または，ボール所有者であれば `Neck_ScanField` を実行しておけば良いでしょう．しかし，状況によっては，特定の情報を得るために視界方向を限定すると，より良い結果が得られます．

例えば，プレイヤエージェントがボールを所有しており，かつ，敵ゴールの近くにいる場合には，敵キーパの確認を優先するとシュートの精度を高めることができます．また，逆サイドの確認をしておけば，ゴール前でボールを左右に振るパスの精度を高めることもできます．以下に実装例を示します．

```
const WorldModel & wm = agent->world();
if ( wm.self().pos().x < 35.0 ) {
  agent->setNeckAction( new Neck_ScanField() );
  return;
}
const PlayerObject * opp_goalie = wm.getOpponentGoalie();
if ( opp_goalie && opp_goalie->posCount() > 2 ) {
  agent->setNeckAction( new Neck_TurnToGoalieOrScan() );
  return;
}
Vector2D opposite_pole( 46.0, 7.0 );
if ( wm.self().pos().y > 0.0 ) opposite_pole *= -1.0;
AngleDeg opposite_pole_angle = ( opposite_pole - wm.self().pos() ).th();
if ( wm.getDirCount( opposite_pole_angle ) <= 1 ) {
  agent->setNeckAction( new Neck_TurnToGoalieOrScan() );
  return;
}
AngleDeg angle_diff
  = agent->effector().queuedNextAngleFromBody( opposite_pole );
if ( angle_diff.abs() > 100.0 ) {
  agent->setNeckAction( new Neck_TurnToGoalieOrScan() );
  return;
}
agent->setNeckAction( new Neck_TurnToPoint( opposite_pole ) );
```

このように，目標位置や目標方向の信頼性をチェックし，信頼性が低ければその方向へ視界を向ける，というルールを書きます.

Section 3.17
状況に応じた意思決定

フィールド全体で同じ意思決定ルールを適用してもチームとしてのパフォーマ

ンスは向上しません．自分やボールが存在する位置などの状況を考慮して振る舞いの特徴を変化させることが重要です．すなわち，これはチームの戦術を作ることを意味します．

3.17.1　フィールドの分割

意思決定における状態を分割する最も簡単な方法は，フィールドをいくつかのサブ領域に分割し，ボールが存在する領域に応じて使用するルールを切替える方法です．フィールドの分割方法は無数に考えられますが，サッカーというゲームの特徴を考えると，図 3.3 のような分割を基本として考えると良いでしょう．図中の破線は分割された領域の境界線です．

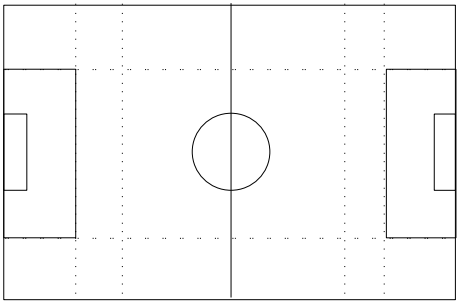


図　3.3　フィールドの分割例

分割をどのように行うかはどのような戦術の実装を目指しているかによって変化するでしょう．しかし，結論から言うと，フィールドの分割は図中に示された形状と数だけでも充分です．あまりに細かく分割したり，複雑な形状の領域を作成したところで管理しきれなくなり，バグを仕込む元になるためです．領域の分割や管理を自動化する手法を用意しない限り，フィールドの分割はシンプルで最小限なものにとどめておくべきです．

実装例は以下に示します．まず，ソース中で座標値を直接書いていては可読性が悪いので，列挙型で領域ごとに以下のような名前を付けておきます．

```
enum BallArea {
  BA_DefenseAreaSide, BA_DefenseAreaCenter,
  BA_OurMidfieldBackSide, BA_OurMidfieldBackCenter,
  BA_OurMidfieldForwardSide, BA_OurMidfieldForwardCenter,
  BA_OppMidfieldBackSide, BA_OppMidfieldBackCenter,
  BA_OppMidfieldForwardSide, BA_OppMidfieldForwardCenter,
  BA_AttackAreaSide, BA_AttackAreaCenter,
  BA_None,
};
```

そして，ボールの位置座標を入力として，これらのうちのひとつを返り値とする関数を作ります.

```
BallArea get_ball_area( const Vector2D & ball_pos ) {
  if ( ball_pos.x > 36.0 ) {
    if ( ball_pos.absY() > 20.0 ) {
      return BA_AttackAreaSide;
    } else {
      return BA_AttackAreaCenter;
    }
  }
  if ( ball_pos.x > 25.0 ) {
    if ( ball_pos.absY() > 20.0 ) {
      return BA_OppMidfieldForwardSide;
    } else {
      return BA_OppMidfieldForwardCenter;
    }
  }

  ...

  if ( ball_pos.absY() > 20.0 ) {
    return BA_DefenseAreaSide;
  }
  return BA_DefenseAreaCenter;
}
```

これらは，後述する Strategy クラスで実装すると良いでしょう．

3.17.2 新しい動作の作成

分割領域ごとにプレイヤエージェントの意思決定ルールを実装し始めると，複数のプレイヤに同じルールを適用できることがあります．そのような場合，共通部分をサブルーチン化する構造化プログラミングが必要ですが，本書では，librcsc の設計に倣って新しい動作クラスを作ることを推奨します．

例として，マークの動作クラス Bhv_MarkOpponent を新たに作成するとします．まず，bhv_mark_opponent.h というヘッダファイルを新規作成し，クラスを宣言します．

```
#ifndef BHV_MARK_CLOSE_OPPONENT_H
#define BHV_MARK_CLOSE_OPPONENT_H
#include <rcsc/geom/vector_2d.h>
#include <rcsc/player/soccer_action.h>
class Bhv_MarkCloseOpponent : public SoccerBehavior {
private:
    const rcsc::Vector2D M_home_pos;
public:
    Bhv_MarkCloseOpponent( const rcsc::Vector2D & home_pos )
        : M_home_pos( home_pos )
      { }
    bool execute( PlayerAgent * agent );
};
#endif
```

続いて，bhv_mark_opponent.cpp にクラスを定義します．実装内容に応じて，include すべきヘッダファイルは変化します．必要に応じて追加してください．

```
#include <rcsc/player/player_agent.h>
#include <rcsc/action/basic_actions.h>
#include "bhv_mark_close_opponent.h"
bool Bhv_MarkCloseOpponent::execute( PlayerAgent * agent ) {
  ...
  return true;
}
```

最後に，Makefile.am にこれらののファイルを追加します．

```
PLAYERSOURCES = \
  ...
  bhv_mark_opponent.cpp \
  ...

PLAHERHEADERS = \
  ...
  bhv_mark_opponent.h \
  ...
```

パッケージのトップディレクトリに戻り，以下のコマンドを実行します．

```
$ ./bootstrap
$ ./configure --with-librcsc=$HOME/rcss
$ make
```

bootstrap と configure によって Makefile が生成し直されます．ファイルを追加，削除した際は，必ずこの手順を踏んでください．

Section 3.18
セットプレイ

3.18.1 プレイモードの判定

rcssserver には多くのプレイモードが定義されています．それらのうちいくつかは実機リーグで利用されることを想定して導入されたため，実際には使用されません．プレイモードの一覧表は付録 A.1 にあるので，そちらも参照してください．

プレイモードの判定は以下のように行います．

```
const GameMode & gm = agent->world().getGameMode();
switch ( gm.type() ) {
case rcsc::GameMode::KickOff_:
  if ( gm.side() == agent->world().getOurSide() ) {
    ...
  } else {
    ...
  }
  break;
case rcsc::GameMode::KickIn_:
  if ( gm.side() == agent->world().getOurSide() ) {
    ...
  } else {
    ...
  }
  break;
  ...
default:
  break;
}
```

switch 文と if 文の組合せでは実装が汚いと感じるかもしれません．余裕があれば，オブジェクト指向プログラミングを実践するのも良いでしょう．

各プレイモードにおいて，プレイモードが味方チームのセットプレイである場合は，チームとして適切に対処しなければなりません対処する必要があるプレイモードは，キックイン，コーナーキック，ゴールキック，直接フリーキック，間接フリーキック，そして，キーパのキャッチ後のフリーキック，です．キーパのキャッチ後のフリーキック以外は，誰がキッカーを務めるか決定しなければなりません．そして，キッカーとなったプレイヤは，適切にボールを蹴ってプレイを再開させなければなりません．

3.18.2 キッカーの選択

チーム内でのフォーメーションと役割配分の情報が静的に決定されており，プレイヤ間で共有されていれば，その戦略に基づいてキッカーを選択すると良いで

しょう．ただし，プレイヤのクライアントプログラムが落ちるなどの障害には弱くなります．ここでは，より実装が簡単でロバストな，最も近いプレイヤをキッカーとして選択する実装例を示します．

```
bool isKicker( const PlayerAgent * agent,
               const Vector2D & home_pos ) {
  const WorldModel & wm = agent->world();
  const long wait_cycle = 30;
  if ( wm.getTeammatesFromBall().empty() ) return true;
  const PlayerObject * nearest_mate
    = nearest_mate = wm.getTeammatesFromBall().front();
  if ( wm.getSetPlayCount() < wait_cycle
       || ( nearest_mate
            && nearest_mate->distFromBall()
               < wm.ball().distFromSelf() * 0.9 )
     ) {
    return false;
  }
  return true;
}
```

この例では，自分がキッカーかどうかを判定する関数，isKicker を実装しています．単純な実装ですが，ほとんどの場面でそこそこ上手く動いてくれます．

関数内で宣言している変数 wait_cycle は，キッカーを選択するまでに待機するサイクル数を指定します．まず，プレイモードが変わってから wait_cycle の間はキッカーの判定を実行しません．これによって，本来のホームポジションへ移動する時間を確保します．wait_cycle 経過後は，最もボールに近い味方プレイヤと自分とを比較し，自分の方がボールに近ければ，自分をキッカーと判定します．

3.18.3 キックの準備

自分がキッカーであると判定した場合は，ボールを蹴るための位置へ移動しなければなりません．このとき，ボールと自分とを適切な位置関係にしておくと，実際にボールを蹴るときのミスを減らすことが出来ます．

適切な位置関係とは，例えばキックイン実行時にはフィールドの外側からボールを蹴った方が有利です．フィールド内に視界を向けつつ，自分の体の正面でボールを蹴ることができるからです．

このような動作は以下の実装で実現できます．

```
AngleDeg ball_place_angle = ... ; // 目標となるボールの相対方向
const WorldModel & wm = agent->world();
const double dir_margin = 15.0;
AngleDeg angle_diff = wm.ball().angleFromSelf()
                      - ball_place_angle;
if ( angle_diff.abs() < dir_margin
     && ( wm.ball().distFromSelf()
          < ( wm.self().playerType().playerSize()
              + ServerParam::i().ballSize() + 0.08 ) ) ) {
  return; // already there
}
Vector2D sub_target
  = wm.ball().pos()
    + Vector2D::polar2vector( 2.3, ball_place_angle + 180.0 );
double dash_power = ServerParam::i().maxPower() * 0.2;
if ( agent->world().ball().distFromSelf() > 2.0 ) {
  dash_power = wm.self().playerType().staminaIncMax()
               * wm.self().recovery();
}
if ( angle_diff.abs() > dir_margin ) {
  Body_GoToPoint( sub_target, 0.1, dash_power, 3
                ).execute( agent );
} else {
  if ( ( wm.ball().angleFromSelf() - wm.self().body() ).abs()
       > 5.0 ) {
    Body_TurnToBall().execute( agent );
  } else {
    agent->doDash( dash_power );
  }
}
```

後は，状況に応じたボールとの位置関係を設定し，そのときのボールの相対方向を引数として与えられるように関数として実装すれば，汎用的に使用できます．

更に，ボールをキック可能な状態なっても周囲の確認のために数サイクルは待機すると，より安全なセットプレイを実現できます．このような動作は以下の実装で実現できます．

```
static int s_wait_cycle = -1;
const WorldModel & wm = agent->world();
if ( s_wait_cycle < 0 ) {
  s_wait_cycle = 20;
}
if ( s_wait_cycle == 0 ) {
  if ( wm.self().stamina() < ServerParam::i().staminaMax() * 0.9
       || wm.getSeeUpdateTime() != wm.getTime() ) {
    s_wait_cycle = 1;
  }
}
if ( s_wait_cycle > 0 ) {
  Body_TurnToBall().execute( agent );
  agent->setNeckAction( new Neck_ScanField() );
  s_wait_cycle--;

  return;
}
s_wait_cycle = -1;
... // ここで何らかのキック動作を実行
```

ローカルな static 変数をカウンタとして用意し，状態を監視します．一定サイクル (ここでは 20) 経過するまでは周囲を見渡す動作を続け，その後，キック動作を実行します．

3.18.4 キーパによるボールキャッチ後

キーパがボールをキャッチすると，プレイモードは通常のフリーキックと同じ状態になります．このとき，キーパは自陣ペナルティエリア内で 2 回まで瞬間移

動できますが，ボールもキーパにくっついた状態で同時に移動します．そのため，普通はキーパがキッカーを担当します．

最大 2 回の瞬間移動とその後のキックを実行するために，キャッチ後の状態の管理が必要になります．これは以下の実装で実現できます．

```
static bool s_first_move = false;
static bool s_second_move = false;
static int s_second_wait_count = 0;
const WorldModel & wm = agent->world();
const long time_diff = wm.getTime().cycle()
                     - agent->effector().getCatchTime().cycle();
if ( time_diff <= 2 ) {
  s_first_move = = s_second_move = false;
  s_second_wait_count = 0;
  ... // 待機動作を実行
  return;
}
if ( ! s_first_move ) {
  s_first_move = true;
  Vector2D move_target = ... ;// 1 回目の移動位置を決定
  agent->doMove( move_target.x, move_target.y );
  return;
}
if ( time_diff < 50 ) {
  ... // 待機動作を実行
  return;
}
if ( ! s_second_move ) {
  s_second_move = true;
  Vector2D kick_point = ... ; // ボールをキックする位置を決定
  agent->doMove( kick_point.x, kick_point.y );
  return;
}
s_second_wait_count++;
if ( s_second_wait_count < 5 )
  ... // 待機動作を実行
  return;
}
s_first_move = s_second_move = false;
s_second_wait_count = 0;
 ... // キック動作を実行
```

この例では 3 つの static 変数を使って，2 回の瞬間移動と最後のキック直前の待機のための状態管理を実現しています．より安全なキックを行うためには，他のフィールドプレイヤの状態の確認も必要です．セットプレイの内容によっては，自陣ペナルティエリア内に味方プレイヤが残っている場合は待機時間を延長する，といった工夫が必要となるでしょう．

Section 3.19
戦略の作成

本書では，戦略とはフォーメーションと役割配分を意味するものとします．これらは試合開始前に充分に調整し，静的に決定しておくことが必要です．ここでは，サンプルチームで提供する枠組みの使用方法を交えて解説を進めます．

3.19.1 フォーメーションの管理

ここでは，サンプルチームにおけるフォーメーションを管理するデータ構造に着いて説明します．ただし，ここからは，SBSP による単純なフォーメーションシステムではなく，本書で紹介する FormationEditor によって作成されるフォーメーションについて説明します．

FormationEditor を用いたフォーメーション作成方法については 3.20 節で解説します．もちろん，SBSP のフォーメーションシステムをそのまま使用しても問題ありません．ただし，直接のパラメータ編集による方法では，フォーメーション作成とその調整には相当の時間が必要であることに注意してください．

FormationParam クラス

個々のプレイヤエージェントのポジショニングルールは，FormationParam クラスによって管理されます．プレイヤエージェントが異動すべきホームポジションを得るには，以下の getPosition メンバ関数を使用します．

```
rcsc::Vector2D
```

```
getPosition( const rcsc::Vector2D & ball_pos,
             const Type type ) const;
```
ball_pos は入力となるボール位置座標．Type は formation.h で宣言されている列挙型で，SIDE，MIRROR，CENTER のいずれか．返り値として移動すべき位置座標が得られる．

チーム開発時，この関数を直接使用することはありません．プレイヤエージェントからのインタフェースは次に解説する Formation クラスであり，この関数は Formation クラスから呼び出されるためです．

注意すべき点は，プレイヤエージェントの役割が同じであれば，FormationParam の実体が共有されるということです．これは，役割が重複している場合，通常は左右均等にその役割のプレイヤが配置されるためです．

FormationParam クラス内部では，3 層のニューラルネットワークによる関数近似によってポジショニングルールを実現しています．プレイヤの役割が同じであれば，同じ結合荷重を持ったネットワークが使用されることになります．引数 type によって左右いずれの配置かを指定し，出力を切り替えることができるようになっています．

Formation クラス

Formation クラスは FormationParam クラスを保持し，管理します．更に，プレイヤエージェントがフォーメーションの情報を得るためのインタフェースも提供します．しかし，通常は以下の getPosition() メンバ関数のみを使用します．

```
rcsc::Vector2D
getPosition( const int unum,
             const rcsc::Vector2D & ball_pos ) const;
```
unum はプレイヤエージェントの背番号．ball_pos は入力となるボール位置座標．返り値として移動すべき位置座標が得られる．

FormationParam の関数とほとんど同じですが，引数が背番号とボール位置になり，より使い易くなっています．これは，Formation クラスが背番号と役割との対応を自動的に解決するようになっているためです．

3.19.2 役割の作成

サンプルチームのソースでは，Role_Sample という役割を実装済みです．サンプルチームにおいて，役割は動作クラスと同様にクラスとして管理されています．よって，新しく役割クラスを作成する場合，それらは全て SoccerRole クラスからの派生クラスとなります．

サンプルの状態では全てのプレイヤが Role_Sample に割り当てられているため，個々のプレイヤは全く同じ意思決定を行ってしまいます．一つの役割の中で戦術を作り込むことは，実装の手間が増えるだけでなく，ソースの管理を難しくしてしまいます．役割をいくつか用意することで，これらの問題が解決できます．つまり，役割は，戦術と戦略の間の橋渡しとして機能します．

新しい役割の作成手順は，新しい動作の作成手順とほぼ同じです．ヘッダファイルとソースファイルを新規作成し，Makefile を作り直します．詳しい手順は，3.17 節を参照してください．

SoccerRole から派生される役割クラスでは，以下の 4 つの関数が宣言，定義されていなければなりません．

- 1 引数のコンストラクタ
- メンバ関数 execute()
- static メンバ関数 name()
- static メンバ関数 create()

ヘッダファイルの実装例は以下のようになります．

```
#ifndef ROLE_CENTER_FORWARD_H
#define ROLE_CENTER_FORWARD_H
#include "soccer_role.h"
class RoleCenterForward : public SoccerRole {
public:
  explicit
  RoleCenterForward( boost::shared_ptr< const Formation > f )
    : SoccerRole( f )
    { }
  virtual
  void execute( PlayerAgent * agent );
  static
  std::string name()
    {
        return std::string( "CenterForward" );
    }
  static
  SoccerRole * create( boost::shared_ptr< const Formation > f )
    {
        return ( new RoleCenterForward( f ) );
    }
};
#endif
```

まず，コンストラクタについて説明します．サンプルチームにおける役割クラスは，Formation クラスのスマートポインタをコンストラクタの引数として取ります．そして，基底クラスである SoccerRole が Formation のスマートポインタを保持するためのメンバ変数を持っています．Formation とは，チームのフォーメーションを管理するためのクラスで，フォーメーションのパラメータとホームポジションを導出するルールを備えています．プレイヤエージェントが役割クラスによって意思決定を行う際は，この Formation クラスからホームポジションの情報が得られます．

execute() 関数は意思決定時に呼び出されます．動作クラスに置ける execute() 関数とほとんど同じと考えて問題ありません．役割クラス作成における作業のほとんどはこの関数の実装になります．

name() 関数は，その役割クラスの名前文字列を std::string 型で返します．こ

の間数は次で説明する Strategy クラスで利用されます．この関数で得られる名前はプログラム起動直後のパラメータ初期化時に使用されるため，他の役割と同じにならないよう，一意な名前を与えなければなりません．

create() 関数は，このクラスの実体を生成するための Factory となっています．この関数は次で説明する Strategy クラスで利用されます．

3.19.3 フォーメーションと役割の管理

サンプルチームでは，Strategy というクラスによってフォーメーションと役割の実体を管理しています．更に，外部の設定ファイルを読み込むことで既存の戦略を再現することができるようにもなっています．Strategy クラスの実体は，SampleAgent クラスのメンバ変数として宣言されています．

サンプルチームの Strategy クラスでは，いくつかの規定のフォーメーションを持たせています．それらの切替えのルールも既に実装されていますが，あくまでサンプルなので，必要に応じて変更してください．ただし，あまり多くのフォーメーションを持たせても管理しきれなくなるだけなので，初めのうちは既存のフォーメーションをそのまま利用することを推奨します．

さて，SoccerRole::create() 関数を Strategy クラスで利用すると前述しました．Strategy クラスでは関数ポインタによって SoccerRole::create() を格納するように実装しています．strategy.h で以下のような typedef と map 変数の宣言を見つけることができます．

```
typedef SoccerRole * (*RoleCreator)( boost::shared_ptr< const Formation > );
std::map< std::string, RoleCreator > M_role_factory;
```

まず 1 行目で，RoleCreator という関数ポインタ型を宣言しています．この関数ポインタに SoccerRole::create() へのポインタが保持されます．2 行目の map 変数は，その名とおり，まさしく役割の Factory となっています．map の key には役割の名前が使われます．ここで SoccerRole::name() が使用されます。

この役割の Factory の初期化は以下のように行います．

```
M_role_factory[RoleSample::name()] = &RoleSample::create;
M_role_factory[RoleGoalie::name()] = &RoleGoalie::create;
M_role_factory[RoleCenterBack::name()] = &RoleCenterBack::create;
M_role_factory[RoleSideBack::name()] = &RoleSideBack::create;
M_role_factory[RoleDefensiveHalf::name()] = &RoleDefensiveHalf::create;
M_role_factory[RoleOffensiveHalf::name()] = &RoleOffensiveHalf::create;
M_role_factory[RoleSideForward::name()] = &RoleSideForward::create;
M_role_factory[RoleCenterForward::name()] = &RoleCenterForward::create;
```

strategy.cpp を参照してみてください．以上のような実装を Strategy クラスのコンストラクタの定義で見つけることができます．Strategy クラスは，外部の設定ファイルを読み，そこに書かれた役割の名前から対応する RoleCreator を呼び出し，役割の実体を生成することができます．

役割の生成が各役割クラスに任されているため，チーム開発において新しい役割を作成した場合は，Strategy クラスのコンストラクタに 1 行追加するだけで良いようになっています．逆に，この 1 行を書かなければいくら役割を作っても実際に使用することが出来ないので注意してください．

3.19.4 フォーメーションの選択

サンプルの Strategy クラスでは，以下の 8 種類のフォーメーションを使用しています．括弧内は対応する設定ファイル名です．

- キックオフ前 (before-kick-off.conf)
- 敵チームのゴールキック (goal-kick-opp.conf)
- 味方チームのゴールキック (goal-kick-our.conf)
- 敵キーパのボールキャッチ後 (goalie-catch-opp.conf)
- 味方キーパのボールキャッチ後 (goalie-catch-our.conf)
- 味方チームのキックイン (kickin-our-formation.conf)
- 自陣にボールが存在する場合 (defense-formation.conf)
- 敵陣にボールが存在する場合 (offense-formation.conf)

上 5 つと下 3 つではファイルのフォーマットが異なることに注意してください．下 3 つは FormationEditor によって作成したフォーメーションパラメータ，すなわち，FormationParam クラスを使用するパラメータセットです．それに対して，上 5 つは 11 人のプレイヤに対して，ボール位置に関わらず固定的な位置座標を指定しています．このような仕様にしている理由は，上 5 つの状況では複雑なパラメータセットを用意するまでもなく，むしろ，固定位置へ移動させた方が都合が良いためです．

これらフォーメーションパラメータは，Strategy::read() メンバ関数に寄ってそれぞれの設定ファイルから読み込まれます．ファイルの内容に不具合があれば，プレイヤエージェントのプログラムはエラーメッセージを出力して即座に終了します．不正な設定ファイルを作成しないように注意してください．

そして，プレイヤエージェントは Formation::getPosition() という関数によってホームポジションを得ると前述しましたが，プレイモードが ***play_on*** 以外の場合は Stragety::getSetPlayPosition() という関数によって移動すべき位置座標が得られるようになっています．この関数は SampleAgent クラス内から呼び出すことを想定しています．使用例は以下のようになります．

```
rcsc::Vector2D move_pos
  = M_strategy.getSetPlayPosition( config().playerNumber(),
                                   role_ptr->formation(),
                                   world() );
```

role_ptr は，現在使用中の役割クラスのポインタです．

3.19.5 動的な役割生成

Strategy クラスには役割を生成する機能が実装されていることは前述しました．この機能を応用すると，プレイヤエージェントに割り当てる役割を動的に生成することが可能になります．例えば，人間のサッカーにおいては，一時的にポジションを入れ換えて役割を交換してしまうようなプレイをすることがあります．Strategy クラスは，このような試合中の役割変更を可能にします．sample_agent.cpp の doAction() で以下の実装を見つけることができます．

```
boost::shared_ptr< SoccerRole >
  role_ptr = M_strategy.createRole( config().playerNumber(),
                                    world() );
if ( ! role_ptr ) {
  setServerAlive( false );
  return;
}
if ( ! role_ptr->hasFormation() ) {
  setServerAlive( false );
  return;
}
```

このように，サンプルにおける実装では，毎サイクルの意思決定直前に役割生成の処理を実行させています．そして，役割の生成に失敗したり，役割がフォーメーションを持っていない場合はエラーとしてプログラムを終了させます．

これによって，状況に応じて適切な役割を生成する準備が整いました．ただし，現在のサンプルの実装では背番号に応じて役割を生成しているだけなので，毎サイクル同じ役割が生成されています．役割の動的生成は解決が難しい問題です．実装は大変なものになるでしょうが，やるだけの価値はあるはずなので，是非挑戦してみてください．

Section 3.20
FormationEditor の利用

チームフォーメーションの作成は戦略の作成を意味します．プレイヤの配置は個々の戦術へ強く影響を及ぼすため，フォーメーション作成はトップダウン的に戦術を決定すると言えます．

3.20.1 FormationEditor

FormationEditor は soccerwindow2 の一部として実装されており，以下のような機能を有しています．

- マウスドラッグによるボールとプレイヤの移動が可能
- ボールとプレイヤの配置状態を訓練データとして記録可能
- 蓄積した訓練データによるフォーメーション学習が可能
- ボールを移動させた際に，獲得済みのフォーメーションによって各プレイヤが自動再配置される．
- 訓練データの再編集，管理が容易

3.20.2 使用方法

FormationEditor を実行するには，以下のように`--editor-mode` オプションを付けて soccerwindow2 を起動します．

```
$ soccerwindow2 --editor-mode
```

起動後，メニューから “New Formation” を選択すると，画面が図 3.4 のような状態になります．

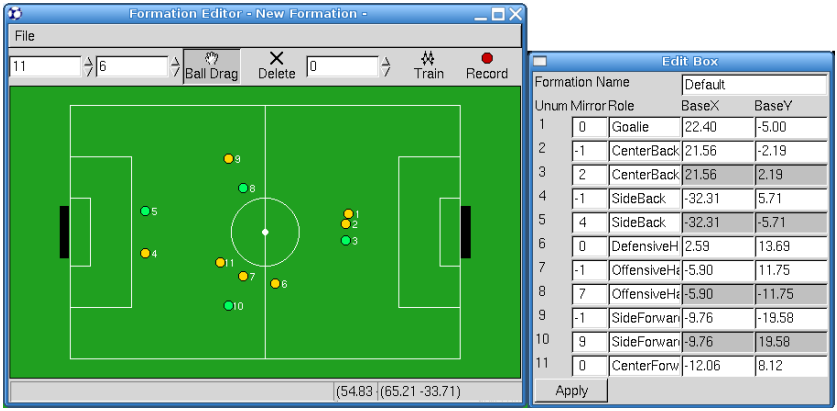


図 3.4 FormationEditor の実行画面

ダイアログに表示されている役割の名前は，役割クラスで定義している名前に対応しています．必要に応じて変更してください．また，役割配分が望みのものと異なるのであれば，ダイアログを操作して変更してください．

後は以下の手順を実行するだけです．

1. ボールを移動
2. プレイヤを移動
3. “Record” ボタンで訓練データ保存
4. “Train” ボタンで学習を実施
5. 訓練データ作成と学習を繰り返し実行
6. メニューから保存して終了

作成したフォーメーションのファイルは，サンプルチームのフォーメーションとして使用可能です．サンプルチームの `Strategy` クラスをそのまま使うなら，味方チームのキックイン，ボールが敵陣に存在する場合，ボールが自陣に存在する場合，の 3 種類のフォーめー書ンを作成することになります．

3.20.3 作成のコツ

訓練データは最大で 50 個までしか保存できないので，無駄な訓練データは適宜削除してください．ツールバーのスピンボタンで訓練データを呼び出せます．“Delete” ボタンを押せば，現在表示中の訓練データがメモり上から削除されます．

FormationEditor におけるフォーメーションの学習は，かなり単純なニューラルネットワークで実現されています．そのため，訓練データそのものの位置へ移動するような近似結果を得ることはできません．上手くいかないからといって訓練データを大量に作ったところで，過学習を引き起こすだけです．フォーメーション作成は細かい精度の調整では無く，大まかな動きや配置の調整であることを念頭に置いてください．細かい位置調整は動的なポジショニングで解決すべき問題です．

とは言え，精度の荒さは FormationEditor の欠点であることに変わりは無いため，異なる手法によるフォーメーション形成手法が必要でしょう．本書執筆時点で既に新たな手法を実装，テスト中なので，今後のバージョンアップを予定しています

どのようなフォーメーションが良いかは，どのような戦術を採用するかで変わってきます．戦術のパフォーマンスを上げるには，状況ごとに異なる戦術の間の移動の無駄を無くす必要があります．そのための戦略レベルでの配置調整を意識することが重要です．単にバランスを調整するだけでなく，採用する戦術を想定してフォーメーションを作成すると良い結果が得られやすいでしょう．

Section 3.21
コミュニケーション

rcssserver 上のプレイヤの視界は非常に限られているため，視覚センサからだけではフィールド上の一部の情報しか得られません．そのため，他のプレイヤとのコミュニケーションを通した情報の共有や意思の疎通が重要になりますコミュニケーションは必須ではありませんが，使わなければ情報量が圧倒的に不利になります．特にパス動作の精度向上において，コミュニケーションは重要です．

3.21.1 利用できるメッセージ

librcsc では，プレイヤ間の簡単なコミュニケーション機能を既に実装しています．本書執筆時点では，以下の内容を通信できます．

- ボールの位置と速度
- パスレシーバの背番号と予想されるレシーブ位置
- 敵キーパの背番号と位置

コミュニケーションに関する意思決定は，`SampleAgent::doCommunication()` で実装されています．これは既に実装済みなので，そのままで自動的に実行されます．ただし，工夫の余地は充分にあるので，必要に応じて変更を加えてください．

3.21.2 メッセージの圧縮

rcssserver 上のプレイヤ間のコミュニケーションでは，一度に送受信できるメッセージ長が 10 文字以内に制限されています．そのため，メッセージの圧縮を行わなければ充分な情報を送信することが難しくなっています．

librcsc では，`AudioCodec` という簡単なメッセージ圧縮、展開ライブラリを実装しています．`AudioCodec` で圧縮できる情報は数値のみです．特に，位置座標や速度成分を圧縮することを想定しています．コミュニケーションで使用できる文字を利用して，実数を 73 進数に変換することで圧縮を実現しています．

メッセージ圧縮のアルゴリズムについては本書では詳しく解説しません．詳しくはソースパッケージの audio_codec.{h,cpp} を参照してください．あまり良いライブラリとは言えませんが，実用できる程度のものにはなっています．

3.21.3 課題

筆者自身，コミュニケーションに力を入れていなかったため充分なライブラリを提供できていません．本書執筆時点で利用できるコミュニケーションメッセージは非常に少なく，情報共有の意味でも不十分です．例えば，オフサイドラインを知らせる，近付いている敵プレイヤを教える，パスを受ける準備が整っていることを背後から教える，などといった応用が考えられます．

コミュニケーションプロトコルの拡張，そして，無駄無く情報をやりとりするためのコミュニケーションモデルの確立が必要です．特に日本はこの分野では世界に対して遅れを取っているので，今後の発展が望まれます．

Section 3.22
より高度な意思決定に向けて

本章ではチーム開発に関わる知識の解説，具体的な実装のテクニックや実例をいくつか解説しました．これらは研究的観点から見れば新規性はほとんど無いに等しく，ただプログラムを作り込んだと言われるかも知れません．しかしながら，チームの戦略や戦術のレベルにおいては，このような作り込みのチームがいまだに強いのも事実です．

これに対して，Priority Confidence Model[6] や [11] のようなプレイヤの意思決定機構の提案がいくつかなされています．これらの手法の利点は，少ないモデルで全体の制御が可能であることです．プレイヤエージェントの制御を作り込みで実現しようとすると，ソースコードが肥大化する一方ですが，これらの手法はこの問題を解決してくれます．ただし，チームのパフォーマンスという点ではルールの作りこみに比べて大きく劣っています．また，チームを作る上ではルールで作りこむ部分もやはり必要です．

プレイヤエージェントがより高度な意思決定を行うには，両方の利点を上手く取り込んだ意思決定モデルの確立が必要でしょう．

第4章

基本スキル開発

個々のプレイヤエージェントの動作精度，すなわちサッカープレイヤとしての基本スキルをより向上させることはチームの強さに直結します．小学生の良くまとまったチームとプロの寄せ集めのチーム，どちらが強いかは結果を見るまでもなく明らかでしょう個々の基本スキルの差が大きすぎては，チームプレイをどれだけ成熟させたところで勝つことは難しいのです．

本章では，シミュレータの物理モデルとプレイヤの行動モデルに関する知識，そして，より良い基本スキルを開発するためのテクニックを解説します．

Section 4.1
rcssserver の物理モデル

プレイヤエージェントを上手く制御するには，シミュレータ内部の物理モデルをよく知らなければなりません．

4.1.1 物体の移動

rcssserver のシミュレーションサイクル更新時，物体の移動に関する情報更新は以下の順序で行われます．

1. 加速度ベクトルの大きさがその物体の最大加速度を越えないように修正．
   $a^t = a^t \times min(|a^t|, accelMax)/|a^t|$
   a^t はサイクル t の加速度，$accelMax$ はその物体の加速度の大きさの最大値．

2. 加速度ベクトルを速度ベクトルに追加.
   $v^{t+1} = v^t + a^t$
   v^t はサイクル t の速度，v^{t+1} はサイクル $t+1$ の速度.

3. 速度ベクトルの大きさが最大スピードを越えないように修正.
   $v^{t+1} = v^{t+1} \times min(|v^{t+1}|, speedMax)/|v^{t+1}|$
   $speedMax$ はその物体の最大スピード.

4. 速度ベクトルにノイズを追加.
   $v^{t+1} = v^{t+1} + noise$
   $noise$ はその物体のノイズパラメータによって求まるノイズベクトル.

5. 速度ベクトルに風ベクトルを追加.
   現在のルールでは考慮する必要が無い.

6. ゴールポストとの衝突をチェック.
   衝突時の位置関係に応じて位置と速度が修正される.

7. 位置座標に速度ベクトルを追加.
   $p^{t+1} = p^t + v^t$
   p^t はサイクル t の位置，p^{t+1} はサイクル $t+1$ の位置.

8. 速度を減衰.
   $v^{t+1} = v^{t+1} \times decay$
   $decay$ はその物体の速度減衰率パラメータ.

9. 加速度を 0 にする.
   $a^{t+1} = (0, 0)$
   a^{t+1} はサイクル $t+1$ の加速度.

10. 全物体の位置更新終了後，物体同士の衝突をチェック.

ボールとプレイヤのいずれもこの手順で位置と速度が更新されます．式中に現れた各パラメータは，それぞれボールとプレイヤの対応するパラメータが使用されます.

ノイズの追加後，最大スピードのチェックが行われない点に注意してください．これは，ノイズによってその物体の最大スピードを越えるかもしれないことを意味します.

4.1.2 移動ノイズ

物体の移動を予測不能なものにするために，物体の移動にはノイズが追加されます．$random(x, y)$ を $[x, y]$ の一様乱数を返す関数とすると，ノイズベクトル $noise$ を得る計算式は，前節の記号を用いて以下のように書けます．

$$randMax = randParam \times |v^{t+1}|$$
$$noise_x = drand(-randMax, randMax)$$
$$noise_y = drand(-randMax, randMax)$$

$randParam$ は物体ごとに設定されたノイズパラメータで，ボールは 0.05(***ball_rand***)，プレイヤは 0.1(***player_rand***) に設定されています．$drand()$ は指定範囲内で一様分布の乱数を生成する関数です．ノイズの大きさは XY 成分それぞれに独立して生成されます．そのため，最大のノイズが加わり続けると，ボールは自然に加速し続けることができます[1)]．何故こんな仕様になっているかは不明です．

移動ノイズは物体の位置座標更新にのみ伴うノイズである点に注意してください．このノイズとは別に，プレイヤの行動コマンドに伴うノイズが追加されます．

4.1.3 物体の衝突

移動物体同士の衝突

全てのボールとプレイヤの移動が完了した後，物体同士の衝突がチェックされます．もし物体の重なりが検出されれば，物体の重なりが無くなる位置までそれぞれ移動し，速度が -0.1 倍されます．

rcssserver における衝突モデルは非常に簡素化されており，衝突の検出を厳密に計算していません．そのため，物体の重なりが発生しない限り衝突は発生せず，物体同士がすり抜けることが可能となっています．この特性はプレイヤによるボールコントロール時に意味を持ちます．

ゴールポストとの衝突

ボールやプレイヤといった移動物体同士の衝突に比べ，移動物体とゴールポストとの衝突は若干厳密なモデルとなっています．物体とゴールポストの位置が重なった場合，その速度ベクトルは衝突時の位置関係に応じて回転します．

[1)] $\sqrt{0.05^2 + 0.05^2} \simeq 0.071$．ボールの速度減衰率は 0.94 = 減衰量は 0.06 なので，ノイズの大きさが減衰量を上回ることがありえます

Section 4.2
プレイヤの行動モデル

4.2.1 利用できる行動コマンド

プレイヤエージェントが意志決定を行うと，その動作を実現するために必要な行動コマンドを rcssserver へ送信します．rcssserver はプレイヤの行動として以下のコマンドを受け付けます．

コマンド	効果
kick	ボールを任意の方向へ加速する．
dash	自分自身を体の方向へ加速する．
turn	体の方向を回転させる．
tackle	ボールを体の方向へ加速する．
catch	(キーパのみ) ボールを手でキャッチする．
move	指定位置へ瞬間移動する．

以上のコマンドはプレイヤエージェントの体を動かすコマンドです．体を動かすコマンドは，1 サイクルに一回，排他的にしか実行できません．例えば，rcssserver があるプレイヤからの **kick** コマンドを受け付けると，そのサイクルが終了するまでそのプレイヤは新しく体を動かすコマンドを実行することができなくなります．**dash** と **turn** が同時に使えないので，斜めに移動することもできません．

更に，以下の補助行動コマンドを使用できます．

コマンド	効果
turn_neck	首を回転させ，視界の方向を変える．
change_view	視界モードを変更する．
say	コミュニケーションメッセージを発する．
pointto	特定位置を指さす．
attentionto	特定プレイヤからのコミュニケーションに注意する．

補助行動コマンドは体を動かすコマンドと同時に使用できます．また，補助行動動コマンドはそれぞれが独立しており，同一サイクル内で同時に組み合わせて使用できます．

コマンドのフォーマットに関する詳細は 6.2 節を参照してください．

プレイヤエージェントが使用できるのはこれらの非常に基本的なコマンドのみです．ドリブルのようなサッカープレイヤとしての動作は，これらコマンドを連続的に実行した結果得られるものです．プレイヤエージェントをサッカープレイヤとして動かすには，コマンド列のプランニングが重要となります．

4.2.2 kick モデル

プレイヤがボールをキック可能な場合，**kick** コマンドによってボールに任意の方向の加速度を与えることができます．ボールとプレイヤとの間の距離がプレイヤの ***kickable_margin*** 以下であれば，プレイヤはボールをキック可能です．ここで言う距離とは，ボールとプレイヤそれぞれの外接円の間の距離を意味します．キック可能領域の半径 $kickable\_area$ は以下の計算で求められます．

$$kickable\_area = ball\_size + player\_size + kickable\_margin$$

すなわち，ボールの中心位置とプレイヤの中心位置との距離が $kickable\_area$ 以下であれば，プレイヤはボールをキック可能になります (図 4.1)．***player_size*** と ***kickable_margin*** はヘテロジニアスプレイヤのパラメータで，`PlayerType` クラスのメンバとして保持されています．

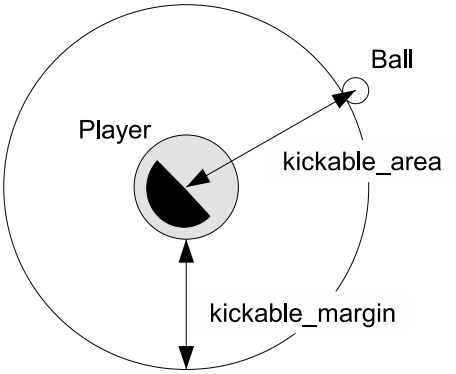


図 4.1 プレイヤエージェントのキック可能領域

kick コマンドは $power$ と rel_dir の二つの引数を取ります．$power$ はボールを蹴る力で，-100(***minpower***) から 100(***maxpower***) の範囲の値を取ります．$power$

が負の場合，加速度の方向が 180 度反転します．rel_dir はプレイヤの体の方向からの相対角度で，-180(***minmoment***) 度から 180 度 (***maxmoment***) の範囲の値を取ります．任意の方向を指定できるので，負の $power$ を使う必要はありません．

kick コマンドが rcssserver に受理され，そのプレイヤがボールをキック可能な状態であれば，キックが実行されます．追加される加速度の大きさは $power$ で決定されますが，その効果率はプレイヤとボールの位置関係によって変化します．rcssserver では，以下の計算式でキックによって与えられる加速度の大きさを求めます．

$$effective\_power = power \times kick\_power\_rate \times (1.0 - decay)$$

$effective\_power$ がボールに与えられる加速度の大きさになります．***kick_power_rate*** の値は 0.027 で，`ServerParam` クラスのメンバとして保持されている全プレイヤで共通のパラメータです．$decay$ がプレイヤとボールの位置関係によって変化し，キック効果率はボールがプレイヤの正面に密着する位置にある場合に最大 ($decay = 0$) となります．$power$ の最大値は 100 なので，与えられる加速度の大きさは 2.7 が最大となります．

$decay$ は以下の計算式で求められます．

$$decay = 1.0 - 0.25 \times \frac{dir\_diff}{180} - 0.25 \times \frac{dist\_ball}{kickable\_margin}$$

dir_diff はボール位置のプレイヤの正面からの角度の絶対値，$ball\_dist$ はボールの外接円からプレイヤの外接円までの距離です．ボールがプレイヤから離れるほど，またプレイヤの正面から角度がずれるほど，キックパワーの効果率は線形に減少します．ボールがプレイヤの真後ろで距離が ***kickable_area*** のときにキック効果率は最低となり，$power$ の 50%しか作用しません (図 4.2，図中の数値はその位置関係における大まかなキック効果率)．

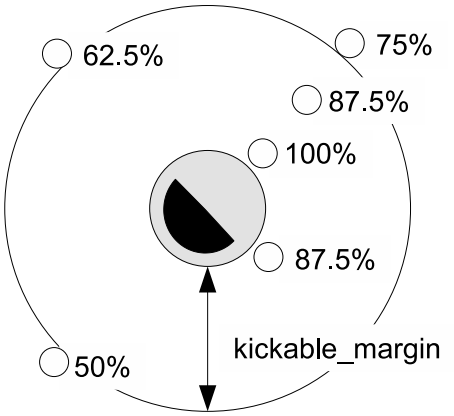


図 4.2 ボールとの位置関係によるキック効果率の変化

kick コマンドによって与えられる加速度とは別に，キックによるノイズが発生します．このノイズの大きさは ***kick_rand*** パラメータと $power$ の大きさによって決定されます．***kick_rand*** はヘテロジニアスプレイヤのパラメータで，`PlayerType` クラスのメンバとして保持されています．デフォルトプレイヤの場合，***kick_rand*** の値は 0 でキックに関するノイズは一切発生しません．他のタイプのプレイヤの場合，***kickable_margin*** の大きさ ***kick_rand*** の大きさは双対となっています．キックによるノイズのベクトル $noise$ は以下の計算式で求められます．

$$randMax = kick\_rand \times power / maxpower$$
$$noise_x = drand(-randMax, randMax)$$
$$noise_y = drand(-randMax, randMax)$$

複数のプレイヤボールをキック可能な場合，それら全てのプレイヤが同時にボールをキックすることが可能です (タックルについても同様です)．**kick** コマンドによる加速度は累積され，それらのベクトル和が最終的な加速度になります．ただし，ボールの加速度の大きさの上限は ***ball_accel_max*** で制限されており，加速度ベクトルはこの大きさ以下に正規化されます．また，ボールの最大速度は 2.7(***ball_speed_max***) に制限されています．ボールが最大速度で蹴り出された場合，ボールの移動距離は $45m$ です．

ボールがキック可能である限り, プレイヤは **kick** コマンドによってボールに加速度を与えることができます．これは，適切なキックを連続して実行すれば，

kickable_area 内でボールを制御し続けられることを意味します．ボールをトラップし，より有利な位置は配置してから望ましい方向へ蹴るといったことが可能です．連続キックを行うことで，ボールを体の正面に置くこと無く最大速度まで加速することも可能です．

4.2.3 dash モデル

dash モデル

dash コマンドはプレイヤをその体の方向へ加速します．**dash** コマンドは引数として *power* ひとつだけを取り，プレイヤが得られる加速度の大きさは *power* の大きさに依存します．*power* は −100(***minpower***) から 100(***maxpower***) の範囲の値を取ります．*power* が負の場合，加速度の方向が 180 度反転し，後向きにダッシュします．

dash コマンドが rcssserver に受理されると，その *power* に使って加速度の大きさが計算されます．rcssserver では，以下の計算式でダッシュの効果，すなわち加速度の大きさを求めます．

$$effector\_power = effort \times dash\_power\_rate \times power$$

$effort$ はプレイヤが持つダッシュの効果率に関わるスタミナパラメータで，***effort_min*** と ***effort_max*** の範囲の値を取ります．これらはヘテロジニアスプレイヤのパラメータで，`PlayerType` クラスのメンバとして保持されています．***dash_power_rate*** も同じくヘテロジニアスプレイヤのパラメータで，ダッシュの効果率を意味します．デフォルトプレイヤの場合，$effort$ の最大値は 1，***dash_power_rate*** は 0.006 で，得られる最大の加速度の大きさは 0.6 となります．デフォルトプレイヤの ***player_decay*** は 0.4 であるため，デフォルトプレイヤの最大スピードは 1.0 に収束します．

プレイヤの加速度には，$effective\_dash\_power$ とプレイヤの体の向きで決定されるベクトルが加算されます．通常，プレイヤは **dash** コマンド以外で加速度を得ることはありません．

スタミナモデル

プレイヤのスタミナに関して，*stamina*, *recovery*，$effort$ の 3 つの重要なパラメータがあります．これらのパラメータは全てのプレイヤが独立して持っており，シミュレーション中，常に更新され続けます．

プレイヤは一定量の *stamina* を持っており，**dash** コマンドによって *stamina* を消費します．**dash** コマンドの *power* が正 (前方に加速した場合) の場合，*stamina* の減少量は *power* になります．*power* が負 (後方に加速した場合) の場合，*stamina* の減少量は $-2 \times power$ になります．そのため，後向きのダッシュは *stamina* の消費が非常に大きくなります．プレイヤの *stamina* が **dash** コマンドに必要とされる量よりも少ない場合，rcssserver 側で残り *stamina* で可能な値まで *power* が修正されます．ヘテロジニアスプレイヤには ***extra_stamina*** というパラメータが設定されています．これは通常の *stamina* が尽きた場合でも余分で使えるスタミナ量で，***extra_stamina*** 分のスタミナは常に使えることを意味します．デフォルトプレイヤの sparamextra_stamina は 0 に設定されています．

stamina はダッシュによって減少し，毎サイクル少しずつ回復します．*recovery* はスタミナの回復率を意味し、$effort$ はダッシュの効果率に関係します．rcssserver のスタミナ更新アルゴリズムは以下のようになります．

```
if ( stamina <= recover_dec_thr * stamina_max )
{
  if ( recovery > recover_min )
    recover -= recover_dec;
  if ( recovery < recover_min )
    recover = recover_min;
}
if ( stamina <= effort_dec_thr * stamina_max )
{
  if ( effort > effort_min )
    effort -= effort_dec;
  if ( effort < effort_min )
    effort = effort_min;
}
if ( stamina >= effort_inc_thr * stamina_max )
{
  if ( effort < effort_max )
  {
    effort += effort_inc;
    if ( effort > effort_max )
      effort = effort_max
  }
}
stamina += recovery * stamina_inc_max;
```

以上の更新がプレイヤそれぞれに適用されます．***stamina_max***(4000), ***recover_dec_thr***(0.3)
recover_dec(0.002), ***recover_min***(0.5), ***effort_dec_thr***(0.3), ***effort_inc_thr***(0.6),
effort_dec(0.005), ***effort_inc***(0.01) は全プレイヤで共通のパラメータで，ServerParam
で保持されています (括弧内は実際に使用される値)．***effort_min***, ***effort_max*** は
ヘテロジニアスプレイヤのパラメータのため，プレイヤによって異なります．

これらのスタミナ関連のパラメータは，各ハーフの最初 (正確には KickOff の
瞬間) に ***stamina_max***, ***recover_init***(1.0), ***effort_max*** に初期化されます．試合
進行中，$stamina$ と $effort$ は回復できますが，$recovery$ は一度減少すると回復
しません．$recovery$ と $effort$ のいずれも減少の閾値率が 0.3 で，$stamian$ に換
算すると 1200 が減少の閾値となります．$recovery$ や $effort$ の減少はプレイヤの
動作パフォーマンスに大きく影響するため，$stamina$ が 1200 を下回らないよう

に注意を払うことが必要です.

4.2.4 turn モデル

turn コマンドはプレイヤの体の向きを回転させます. **turn** コマンドによる回転は 1 サイクルで完結し，回転の動きに慣性が働くことはありません. **turn** コマンドは引数として *moment* ひとつだけを取り，これはプレイヤが体の向きを回転させる角度を意味します. *moment* は -180(***minmoment***) から 180(***maxmoment***) の範囲の値を取るため，プレイヤは任意の方向へ瞬時に体を向けようとすることができます. ただし，プレイヤの回転には移動速度の慣性による影響が導入されており、プレイヤの速度の大きさに応じて回転の効果が低くなるように設定されています. プレイヤの実際の回転角度 *actual_angle* は以下の式で求められます.

$$actual\_angle = moment/(1.0 + inertia\_moment \times player\_speed)$$

inertia_moment はヘテロジニアスプレイヤのパラメータで，`PlayerType` クラスで保持されています. プレイヤのスピード *player_speed* が 0 の場合，プレイヤの回転角度は *moment* に等しくなります.

デフォルトプレイヤの ***inertia_moment*** は 5.0 で，プレイヤのスピードが 1.0 であれば，可能となる最大の回転角度は ± 30 度となります. ただし，同じサイクルで **dash** コマンドと **turn** コマンドを実行することはできないので，プレイヤが **turn** を実行するときは，必ず ***player_decay*** によって速度が減衰された後です. よって，**turn** 実行時のデフォルトプレイヤの最大スピードは 0.4 となり，このときの可能な回転角度は ± 60 度となります. rcssserver-10.0.7 の仕様では，ヘテロジニアスプレイヤの ***inertia_moment*** はデフォルトプレイヤよりも常に大きくなります. これは，ヘテロジニアスプレイヤはデフォルトプレイヤよりも小回りが利かないことを意味します.

プレイヤを回転させたい場合，慣性による影響を考慮して目標回転各度が得られるようにコマンドの引数を修正しなければなりません. プレイヤを *target_moment* 度回転させたい場合，コマンドの引数として与える値 *command_moment* は以下の式で求められます.

$$command\_moment = target\_moment \times (1.0 + player\_speed \times inertia\_moment)$$

プレイヤの回転にはノイズが含まれます. ノイズの割合は 0.1(***player_rand***) で，回転角度に直接かけられます. 最終的なプレイヤの回転各度は以下のようになります.

$noise = drand(-player\_rand, player\_rand)$
$actual\_angle = actual\_angle \times (1.0 + noise)$

$drand()$ は指定範囲内で一様分布の乱数を生成する関数です．よって，プレイヤが速度 0 で 180 度回転しようとした場合，±18 度もの誤差が発生することを意味します．プレイヤの移動においては，**turn** コマンド実行によるノイズへの頑健性が重要となります．

4.2.5 tackle モデル

tackle コマンドはプレイヤの体の向きへボールを加速します．タックル可能な領域は，プレイヤの体の前方に 2.0(***tackle_dist***)，体の後方に 0.5(***tackle_back_dist***)，体の左右に幅 1.0(***tackle_width***) の矩形です (図 4.3)．ただし，**kick** とは異なり，この領域内にボールが存在してもタックルは必ずしも成功しません．タックルの成功確率はボールとプレイヤとの位置関係によって，以下の式で計算されます．

ボールがプレイヤの前方にある場合のタックル失敗確率：
$fail\_prob = (|player\_to\_ball.x|/tackle\_dist)^6 + (|player\_to\_ball.y|/tackle\_width)^6$

ボールがプレイヤの後方にある場合のタックル失敗確率：
$fail\_prob = (|player\_to\_ball.x|/tackle\_back\_dist)^6 + (|player\_to\_ball.y|/tackle\_width)^6$

タックル成功確率：$tackle\_prob = 1.0 - fail\_prob$

$player\_to\_ball$ はプレイヤからボールへの相対ベクトルで，X 軸がプレイヤの体の方向へ回転させられています．タックルの成功確率が得られ，その値が 1.0 以下であれば，ベルヌーイ分布によって **tackle** コマンドの最終的な成否が決定されます．

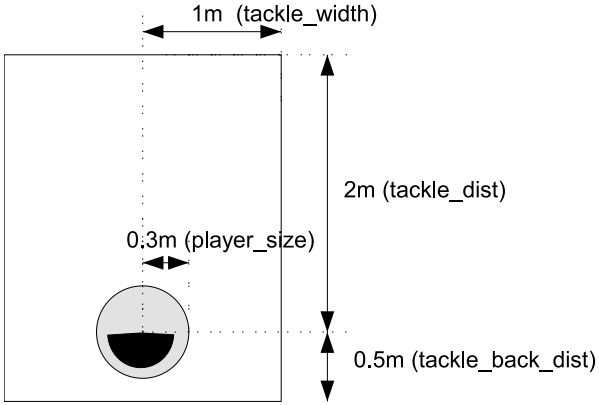


図 4.3 タックル可能領域

tackle コマンドは引数として *power* ひとつだけを取り，これはボールを蹴る力を意味します．*power* は -100(***minpower***) から 100(***maxpower***) の範囲の値を取り，*power* が負の場合，加速度の方向が 180 度反転します．**kick** の場合とは異なり，加速度の方向の反転は大きな意味を持ちます．**tackle** によるボールへの加速度はプレイヤの体の方向にしか与えられないため，ボールを前方と後方のどちらに蹴り飛ばすかは非常に重要な選択となります．

ボールへ与えられる加速度の大きさ $effective\_power$ はボールとプレイヤとの位置関係に依存せず，*power* の大きさによってのみ決定されます．

$$effective\_power = power \times tackle\_power\_rate$$

tackle_power_rate の値は 0.027 で，`ServerParam` クラスのメンバとして保持されている全プレイヤで共通のパラメータです．すなわち，ボールの速度が 0 の状態に *power* が 100 の **tackle** コマンドが成功すると，ボールは最大速度まで一瞬で加速されます．**kick** の場合と同様，**tackle** によってもノイズが発生します．ノイズの計算方法は **kick** と全く同じであるため、省略します．ボールの加速度に関する制約も **kick** の場合と同様です．

プレイヤが **tackle** コマンドを実行すると，成功不成功に関わらず 10([tackle_cycles]) サイクルの間，プレイヤエージェントは動けなくなります．

4.2.6 catch モデル

キーパとして rcssserver と接続したプレイヤエージェントは，手を使ってボールをキャッチすることができます．**catch** コマンドはこのキャッチを行うためのコマンドです．プレイモードが ‘play_on’ で，ボールが自陣ペナルティエリア内にあり，ボールがキーパのキャッチ可能領域内にあれば，キーパはボールをキャッチできます．一度ボールをキャッチすれば，ボールが蹴られるまではボールはキーパの手の中にあり，敵に奪われることはありません．***back_passes*** が有効で審判がバックパス違反を取る場合は，最後にボールに触ったプレイヤが味方であれば，ボールをキャッチした瞬間にバックパス違反を取られます．

catch コマンドは引数として *rel_dir* ひとつだけを取り，これはキャッチを試みる方向を意味します．*rel_dir* は −180(***minmoment***) から 180(***maxmoment***) の範囲の値を取るため，プレイヤは任意の方向のボールをキャッチすることができます．キャッチ可能領域は，*rel_dir* の方向に長さ 2.0(***catcahble_area_l***)，幅 1.0(***catchable_area_w***) の矩形です (図 4.4)．**catch** コマンド実行時，この矩形領域内にボールがあれば，***catchalbe_probability*** の確率でボールをキャッチできます．ただし，***catchalbe_probability*** は 1 に設定されているため，ボールがキャッチ可能な位置にあれば，キーパは必ずボールをキャッチすることができます．

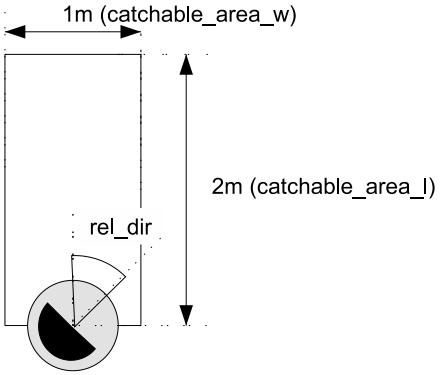


図 4.4 rel_dir 方向でのキャッチ可能領域

キーパがボールキャッチに成功した場合，次の 5(***catch_ban_cycle***) サイクルの間，**catch** コマンドを実行することができません．もしこの間に **catch** コマンドを rcssserver へ送信しても，何の効果も得られません．ボールキャッチに失敗した場合，この制限は受けません．

キーパによるボールキャッチ後，ボールは自動的にキーパの正面に密着した状態 (**kick** コマンドの *power* 効果率が 100%になる位置関係) になります．キーパが移動してもボールは自動的にその位置で保持されます．そして，プレイモードはすぐに ‘goalie_catch_ball_{l|r}’ へと変更され，同一サイクル内に再び ‘free_kick_{l|r}’ へと変更されます．この 2 回のプレイモード変更は **catch** コマンドが rcssserver に受理された直後のサイクル途中に行われます．そのため，もしボールキャッチ直後にプレイモードの変更を認識できていない味方プレイヤの誰かがボールをキックしてしまえば，プレイモードは即座に ‘play_on’ へと更に変更されます．

ボールキャッチ後のキーパは，ボールを持ったまま **move** コマンドによって自陣ペナルティエリア内を 2(***goalie_max_moves***) 回まで瞬間移動できます．**move** コマンドの使用回数制限を越えると，**move** コマンドを実行しても何の効果も得られず，“**(error too_many_moves)**” という応答が返されます．通常の **dash** コマンドや **turn** コマンドによる移動も可能です．ただし，この移動中にボールをペナルティエリアの外に出してしまうと，‘catch_fault_{l|r}’ が取られ，相手チームへフリーキックが与えられます．**catch**，**move**，小さいキックを繰り返して何度も移動するような行為は非紳士的なプレイとみなされるので注意してください．

キャッチ可能領域は矩形であるため，実際のキャッチ可能な距離 $catchable\_length$ は矩形の対角線から求められます．矩形の幅は半分になるので，

$$catchable\_length = \sqrt{(catchable\_area\_l)^2 + (catchable\_area\_w/2)^2}$$

キャッチの効果はは任意の方向に作用させられるので，結局，$catchable\_length$ の円領域をキャッチ可能領域とみなすことが出来ます (図 4.5)．

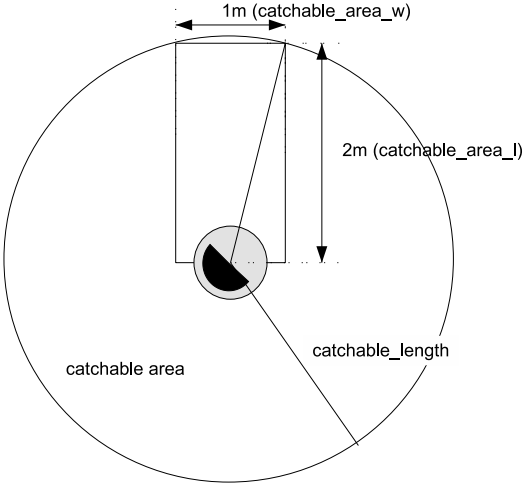


図 4.5 キャッチ可能領域

ただし，*catchable_length* を有効距離としてキャッチを試みる場合，**catch** コマンドの引数 *rel_dir* を以下のように補正しなければなりません．

$$rel\_dir = rel\_dir + \arctan(catchable\_area\_w/2/catchable\_area\_l)$$

4.2.7 move モデル

move コマンドによってプレイヤはフィールド上を瞬間移動できます．ただし，**move** コマンドは試合進行を円滑にする目的で導入されているコマンドであるため，使用できる状況は非常に限られています．**move** コマンドが使用できるのは，キックオフ前 (‘`before_kick_off`’) とゴール直後 (‘`goal_{l|r}_?`’)，そしてボールキャッチ直後のキーパのみです．

move コマンドは x と y のふたつの引数を取り，これらは移動先絶対位置座標を意味します．座標系は，フィールド中央を原点とし，敵ゴール方向が X 軸正方向，敵ゴール方向に対して右手側が Y 軸正方向となります．他の移動に関するコマンドとは異なり，移動には誤差が含まれず，コマンドで指定した位置へ正確に移動できます．x と y にはフィールド上の任意の位置を指定できますが，キックオフ前とゴール直後には自陣エリア内のみを，ボールキャッチ後のキーパは自陣のペナルティエリア内のみを指定すべきです．

キックオフ前とゴール直後には，移動回数の制限はありません．ただし，ゴール直後のプレイモードは 5 秒間しか継続しないため，実際の実行回数は約 50 回までとなります．

4.2.8 turn_neck モデル

turn_neck コマンドはプレイヤの首の向きを回転させます．首の角度の回転は体の動きとは独立しており，プレイヤは体の向きと無関係に首の向きを変更することができます．**turn_neck** コマンドは **kick**, **dash**, **turn** などの体を動かすコマンドと同時に使用することもできます．

プレイヤの首の向きは，体の向きに対して-90(***minneckang***) 度から 90(***maxneckang***) 度までです．プレイヤの首の向きは，プレイヤの体の向きに対して相対に管理されているため，**turn_neck** コマンドを実行していなくても，**turn** コマンドが実行されればプレイヤの首の絶対方向は自動的に変更されます．

turn_neck コマンドは引数として *moment* ひとつだけを取り，これはプレイヤの首の向きを回転させる角度を意味します．*moment* は −180(***minneckmoment***) から 180(***maxneckmoment***) の範囲の値を取ります．首の回転には誤差は含まれず，プレイヤは正確に首を回転させることができます．プレイヤの首の角度の範囲が [-90, 90] であるの対して，回転角度の範囲が [-180, 180] である点に注意してください．首の向きを-90 度から 90 度へ回転させる場合，その回転角度は 180 度になります．しかし，首の向きを 0 度から 90 度へ回転させる場合，回転角度に 180 度を指定しても首の向きは 90 度以上にはならず，強制的に 90 度に修正されます．

4.2.9 change_view モデル

change_view コマンドはプレイヤの視界モード (視野角の広さと，視覚情報に含まれる情報の精度) を変更します．視野角の広さは，45 度 (**narrow**)，90 度 (**normal**)，180 度 (**wide**) の 3 種類が使用できます．情報の精度には **high** と **low** の二種類がありますが，通常，変更する必要はありません．

change_view コマンドはプレイヤエージェントが受け取る視覚情報の頻度を変更するため，非常に扱いが難しいコマンドです．ライブラリで提供している行動クラス以外で視界モードを変更することは，推奨されません．

change_view コマンドとプレイヤの視界モードに関しては 6.4.3 節以降で詳しく解説しています．

4.2.10 say モデル

say コマンドを使用することで，プレイヤは他のプレイヤへコミュニケーションメッセージを配信することができます．**say** コマンドは引数として文字列 *message* を取り，この内容がそのまま他のプレイヤへと配信されます．ただし，*message* の文字列長は 10(***say_msg_size***) 文字以内に制限されており，また使用できる文字は `[-0-9a-zA-Z ().+*/?<>_]` の 73 種類のみです．メッセージを発したプレイヤから $50m$(***audio_cut_dist***) 以内であれば，敵味方を問わずメッセージが配信されます．

say コマンドが rcssserver に受理されると，実際のメッセージの配信は次サイクルの最初に行われます．各プレイヤは *hear_capacity* というパラメータを持っており，プレイヤがメッセージを聞けるかどうかはこの値に依存します．プレイヤの *hear_capacity* は最大 1(***hear_max***) で，メッセージを聞くたびに 1(***hear_decay***) だけ減少します．そして，*hear_capacity* はサイクル更新時に 1(***hear_inc***) だけ回復します．rcssserver-10.0.7 の設定では，プレイヤは 1 サイクルにひとつだけメッセージを聞くことができます．*hear_capacity* を越えて受信したメッセージにはメッセージ本体の文字列が含まれず，メッセージが配信されたということしか判断できません．プレイヤが受け取るコミュニケーションメッセージのフォーマットについては，6.3.2 節を参照してください．

他のプレイヤからのコミュニケーションメッセージを受信するか否かをプレイヤ自身の意志で変更することも可能です．そのためには，**ear** コマンドを使用します．**ear** コマンドでは，メッセージ自体を聞くか否か，敵または味方からのメッセージを聞くか否か，完全なメッセージのみを聞くか否か，などを設定できます．表 4.1 にコマンドの実例とその効果を示します．

コマンド	効果
(ear on)	全てのメッセージを受信.
(ear off)	全てのメッセージを受信しない.
(ear on our)	味方からの全てのメッセージを受信.
(ear off opp)	敵からの全てのメッセージを受信しない.
(ear on opp complete)	敵からの完全なメッセージを受信.
(ear on our partial)	味方からの不完全メッセージを受信.
(ear off opp complete)	敵からの不完全メッセージを受信しない.

表 4.1 ear コマンドの例とその効果

デフォルトでは，敵味方両方からの完全なメッセージのみを受信する設定になっています．通常は以下の **ear** コマンドを実行し，味方からのメッセージのみを聞くように設定しておくのが良いでしょう．

```
(ear off opp)
```

4.2.11 pointto モデル

pointto コマンドは指定位置を指さすためのコマンドです．**pointto** コマンドは引数として *dist* と *rel_dir* のふたつ，または **off** ひとつを取ります．*dist* と *rel_dir* が指定されると，プレイヤの現在位置から *dist* の距離，プレイヤの体の向きからの相対方向が *rel_dir* の位置を指さします．他のプレイヤはこの指さし動作を視覚情報として観測できます．一度指さし動作を行うと，コマンド実行から 20(***point_to_duration***) サイクルの間は自動的に指さし動作が継続されます．その間，プレイヤが移動しても指さし位置は変化せず，指さし方向が rcssserver 内部で自動更新され，プレイヤは固定位置を指さし続けます．引数として **off** を指定すると，現在行っている指さし動作を止めます．また，一度指さし動作を行うと，最低 5(***point_to_ban***) サイクルは途中で指さし位置を変更することができず，指さし動作自体を止めることもできません．

4.2.12 attentionto モデル

atteintionto コマンドによって，特定のプレイヤからの **say** メッセージに注意を向けることができます．注意を向けていない状態では，複数のプレイヤが同時に **say** コマンドを実行して，複数のメッセージが配信されると，プレイヤが受信するメッセージはその中からランダムに選択されます．**atteintionto** コマンドはそのランダム性を無くし、コミュニケーション相手をプレイヤ自身の意志で決定することができます．

attentionto コマンドは引数として *side* と *unum* のふたつ，または **off** ひとつを取ります．*side* はチームを表す文字列 (**our** または **opp**)，*unum* は背番号です．**off** が指定されると，現在の注意を無効にします．一度注意を向けると，新しい **attentionto** コマンドを実行するまでその注意対象は変更されません．

attentionto コマンドには使用制限が無く，いつでも何度でも実行でき，効果が即座に現れます．

Section 4.3
PlayerAgent クラスのインタフェース

PlayerAgent クラスには，前節の各行動コマンドを生成するための以下のような関数を用意しています．それぞれ，プレイヤエージェントのコマンドを作成し，ActionEffector クラスに登録します．登録されたコマンドは，自動的に適切なタイミングで rcssserver へ送信されます．

PlayerAgent メンバ
`bool doKick(const double & power, const AngleDeg & rel_dir );` kick コマンドを登録する．power と rel_dir はコマンド引数に相当．ボールがキック可能でない，またはタックルの影響で動けない場合は false を返す．
`bool doDash(const double & power );` dash コマンドを登録する．power はコマンド引数に相当．ただし，無駄な power は自動で削減される．タックルの影響で動けない場合は false を返す．
`bool doTurn( const AngleDeg & moment );` turn コマンドを登録する．moment は目標回転角度．moment 度回転するために必要なコマンド引数は自動的に計算される．moment 度回転できないときは最大限回転しようとする．タックルの影響で動けない場合は false を返す．
`bool doTackle( const double & power );` power はコマンド引数に相当．タックルの影響で動けない場合は false を返す．
`bool doCatch();` コマンド引数は自動的に計算される．キャッチできない状況，またはタックルの影響で動けない場合は false を返す．
`bool doMove( const double & x, const double & y ));` x, y はコマンド引数に相当．瞬間移動できない状況，またはタックルの影響で動けない場合は false を返す．
`bool doTurnNeck( const AngleDeg & moment );` moment はコマンドの引数に相当．

`bool doChangeView( const ViewWidth & width );` `width` はコマンド引数の視野角の設定に相当．視覚情報の精度は変更できない．既に指定の視野角になっている場合，または指定の視野角に変更すべき状況でない場合は `false` を返す．詳しくは 6.4 節を参照．
`bool doSay( const std::string & msg );` `msg` はコマンドの引数に相当．`PlayerAgent` がコミュニケーションを使用しない設定になっているとき，または `msg` の文字列長がルール制限を越えていれば `false` を返す．
`bool doPointto( const double & x, const double & y ));` `(x, y)` の位置を指さすようにコマンド引数は自動的に計算される．指さしを変更できない状況，または自分自身の位置が不明な場合は `false` を返す．
`bool doPointtoOff();` 指さしを止める．指さしを変更できない状況であれば `false` を返す．
`bool doAttentionto( SideID side, const int unum );` `side` と `unum` はコマンド引数に相当．注意対象の情報が不正であれば `false` を返す．
`bool doAttentiontoOff();` 注意対象を無効にする．

Section 4.4
動作クラスの実装

3.6 節でも述べたとおり，librcsc ではプレイヤエージェントの動作をクラスライブラリとして実装，管理しており，各動作クラスをグローバル関数のように扱うことができます．この設計によって，各動作クラスの `execute()` 関数が他の動作クラスへと処理を委譲することができます．処理の委譲は，引数である `PlayerAgent` のポインタをより単純な動作を実装した動作クラスへと渡していき，最終的に `PlayerAgent` クラスのコマンド登録関数を呼び出して完了となります．例えば，`Body_TurnToBall` クラスは `Body_TurnToPoint` クラスへ処理を委譲し，`Body_TurnToPoint` クラスによって `PlayerAgent` クラスのメンバ関数である

doTurn() が呼び出されます.

このような設計になっている理由は，プレイヤエージェントの動作の追加を柔軟に行えるようにするためです．例えば，同様の実装を PlayerAgent クラスのメンバ関数のみで行う設計を考えてみてください．同等のことは実現できるものの，PlayerAgent クラスの実装が著しく肥大化することが容易に想像できます．また，新しい動作の関数を追加するたびに PlayerAgent クラスに依存する全てのソースファイルの再コンパイルが必要となり，開発効率も低下します．グローバル関数のように振る舞う動作クラスは，これらの問題を解決してくれます.

動作クラスの実装において注意すべき点は，処理の委譲に循環構造が発生しないようにすることです．これは通常のプログラミングにおいても同様の注意点ですが，動作クラスを用いた設計ではその実装が分散してしまうため，実装の依存関係を把握しづらくなっています．動作クラスの実装時には動作の依存関係の確認を怠らないようにしなければなりません.

意図クラスにおける execute() メンバ関数の実装は，動作クラスにおける execute() の実装と大差ありません．意図クラスの実装においては，finished() メンバ関数を適切に実装することが重要です.

Section 4.5
目標位置への移動

4.5.1 目標方向への回転

プレイヤエージェントは自分の体の方向にしか加速度を生成できないため，目標位置へ移動する場合には，まずその方向へ体を回転させなければなりません．単純に考えれば，以下のような実装になります.

```
const SelfObject & self = agent->world().self();
Vector2D rel_pos = target_pos - self.pos();
AngleDeg turn_angle = rel_pos.th() - self.pos();
agent->doTurn( turn_angle );
```

しかし，プレイヤエージェントの慣性による移動のために，これでは希望どおりの結果は得られないでしょう．プレイヤエージェントが速度を持っているのであれば，次のサイクルの予測位置に基づいて回転角度を求めなければなりません.

```
Vector2D my_next = self.pos() + self.vel();
Vector2D rel_pos = target_pos - my_next;
AngleDeg turn_angle = rel_pos.th() - self.pos();
agent->doTurn( turn_angle );
```

一見良さそうですが，これでも不十分です．プレイヤエージェントの移動が 1 サイクルで完了することはほとんど無く，目標方向へ体を向けた後，複数回のダッシュが必要とされます．最終的な必要サイクル数 cycle を見積もることができれば，慣性による最終移動位置も予測できます．図 4.6 に，慣性による移動のために必要となる回転角度が変化する様子を示します．

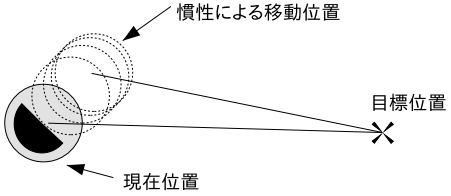


図 4.6 慣性による移動のために変化する回転角度

最終的な実装は以下のようになります．

```
Vector2D my_point = self.playerType().inertiaPoint( self.pos(),
                                                    self.vel(),
                                                    cycle );
Vector2D rel_pos = target_point - my_point;
AngleDeg turn_angle = rel_pos.th() - self.body();
agent->doTurn( turn_angle );
```

これとほぼ同じコードを Body_TurnToPoint クラスの実装で見つけることができます．inertiaPoint() は慣性のみによる移動位置を求める，PlayerType クラスのメンバ関数です．inertiaPoint() によって，cycle 後の移動位置を予測し，更に必要な回転角度を求めることができます．

プレイヤエージェントのスピードによっては，目標方向への回転が 1 サイクルで完了しないかもしれないことに注意してください．

4.5.2 回転角度の閾値

プレイヤエージェントが既に目標方向へ向いているのであれば，回転する必要はありません．このとき，目標方向と体の向きとの差を何度まで許容するかの閾値を決めなければなりません．この角度の閾値 `turn_thr` は，目標位置までの距離の閾値 `dist_thr` を用いて計算できます (図 4.7).

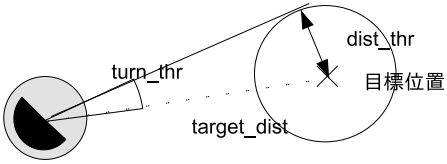


図 4.7 回転の閾値

要求される回転角度が求めた閾値よりも大きければ **turn** を実行することになります．ただし，実際には，アークサインで求めた $turn\_thr$ の値が非常に小さくなることがあります．この場合，回転動作に含まれる誤差のために目的の回転角度が得られず，無駄な回転を繰り返す可能性があります．この問題には，閾値の最小値 min_thr をあらかじめ設定しておくことで対処できます．

実装は以下のようになります．これとほぼ同じコードを `Body_GoToPoint` クラスの実装で見つけることができます．

```
AngleDeg turn_angle = rel_pos.th() - self.body();
double target_dist = rel_pos.r();
double turn_thr = 180.0;
if ( dist_thr < target_dist ) { // まだ目標位置へ到達していない
  turn_thr = AngleDeg::asin_deg( dist_thr / target_dist );
}
turn_thr = std::max( turn_thr, min_thr );
if ( turn_angle.abs() > turn_thr ) {
  agent->doTurn( turn_angle );
}
```

4.5.3 目標位置へのダッシュ

ダッシュにおいても，回転の場合と同様，慣性による移動を考慮します．rcssserver の物体の移動モデルでは，物体の速度は速度減衰率によって減少していくという単純なモデルとなっています．よって，速度減衰率 $decay$ の物体が初速 $first\_speed$ で移動を始めたとき，n サイクル後の総移動距離 $inertia\_move\_dist$ は以下のように等比数列の和として求められます．

$$inertia\_move\_dist = first\_speed \times (1 - decay^n)/(1 - decay)$$

この式から，総移動距離が判明していれば，必要な初速は以下の式で簡単に得られることが分かります．

$$first\_speed = inertia\_move\_dist \times (1 - decay)/(1 - decay^n)$$

必要な初速が分かれば，次は必要な加速度を求めます．加速度 $accel$ は，以下の式のように目標スピードと現在のスピードの差を求めるだけで得られます．

$$accel = first\_speed - current\_speed$$

この $accel$ の大きさがプレイヤエージェントが 1 サイクルで生成できる加速度の大きさより小さければ，たった 1 回の **dash** コマンドを実行するだけで，目標位置へ n サイクルで到達することができます．逆に，$accel$ が生成できる加速度の大きさを上回っていれば，目標位置へ到達するために複数回の **dash** コマンドを実行しなければなりません．

ここで，プレイヤは体の向きにしか加速度を生成できないことを思い出して下さい．$accel$ による加速度ベクトルは，体の向きにしか生成できません．よって，速度と加速度のベクトルを，プレイヤの体の向きを軸とする座標系に変換すると計算が簡単になります (図 4.8)．

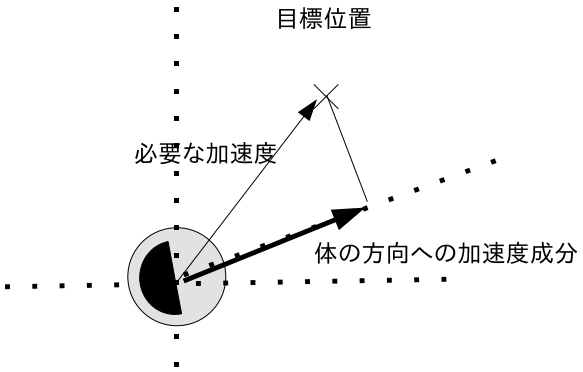


図 4.8 ダッシュによる加速度の求め方

必要な加速度の大きさが分かれば，プレイヤエージェントが持つダッシュ効果率から **dash** コマンドに必要なダッシュパワーが求められます．実装は以下のようになります．これとほぼ同じコードを Body_GoToPoint クラスの実装で見つけることができます．

```
rel_pos.rotate( - self.body() ); // 体の方向を X 軸に
double first_speed = calc_first_team_teom_series(
                       rel_pos.x,
                       self.playerType().playerDecay(),
                       cycle );
first_speed = min_max( - self.playerType().playerSpeedMax(),
                       first_speed,
                       self.playerType().playerSpeedMax() );
Vector2D rel_vel = self.vel().rotatedVector( self.body() );
double accel = first_speed - rel_vel.x;
accel = min_max( - ServerParam::i().playerAccelMax()
                 accel,
                 ServerParam::i().playerAccelMax() );
double dash_power = accel / self.dashRate();
dash_power = ServerParam::i().normalizePower( dash_power );
agent->doDash( dash_power );
```

まず初めに，目標までの相対位置を自分の体の向きを X 軸とする座標系へ回転さ

せます．次に，必要な初速を calc_first_team_teom_series() によって求めます．これは，librcsc で定義されている等比数列の第一項を求める関数です．第一引数の rel_pos.x が必要な総移動距離です．続いて，自分自身の現在の速度も体の向きを X 軸とする座標系へ回転させます．必要な初速と自分自身の現在の速度から，必要な加速度が得られます．後は，プレイヤエージェント自身の現在のダッシュ効果率から必要なダッシュパワーを求めるだけです．

この実装によって，プレイヤエージェントの体の方向に対する前後のずれが最小に抑えられます．

4.5.4 目標位置での停止

目標位置に既に到達しており，現在位置に即座に停止したい場合は，プレイヤエージェント自身の速度を 0 に近づけなければなりません．このとき必要な加速度は，現在の速度を-1 倍することで得られます．後は通常のダッシュと同じです．

実装は以下のようになります．これとほぼ同じコードを Body_StopDash クラスの実装で見つけることができます．

```
const SelfObject & self = agent->world().self();
Vector2D rel_vel = self.vel().rotatedVector( - self.body() );
double dash_power = - ( rel_vel.x / self.dashRate() );
dash_power = ServerParam::i().normalizePower( dash_power );
agent->doDash( dash_power );
```

Section 4.6
インターセプト

インターセプトとは，プレイヤが動いているボールを追いかける動作です．インターセプトと言うと一般的にはパスカット動作を意味しますが，サッカーシミュレーションにおいては，慣習的にボール捕捉動作を全てインターセプトと呼んでいます．インターセプトの精度はチームの強さを大きく左右します．また，もっとも早くボールに追い付くのは誰か，そしてそれは何サイクル後かという情報は，

プレイヤエージェントの意思決定に必要不可欠です．そのため，インターセプトはプレイヤの動作の中でも最も重要なスキルのひとつとなっています．

4.6.1 予測手順

プレイヤエージェント自身のインターセプト予測は可能な限り厳密に行われます．インターセプト予測の手順は以下のようになります．

1. 1 サイクルでボール捕捉できるかどうかを判定
   捕捉可能な場合，ボールトラップ位置が最適になるようにダッシュパワーを調整して終了
2. 最短捕捉サイクル n を推定
3. n サイクル後のボール位置 $bpos_n$ を予測
4. プレイヤエージェントが $bpos_n$ に n サイクルで到達できるかどうかを予測．厳密には，制御可能な距離にボールを置けるかどうかを判定する．
   (a) $bpos_n$ へ体を向けるのに必要なサイクル数 n_t を推定
   (b) $bpos_n$ へ到達するのに必要なダッシュ回数 n_d を推定
   (c) $n <= n_t + n_d$ であれば，ボールを制御可能距離内に置けるかどうかを判定
5. $n = n + 1$ として，手順 3 から繰り返し．

4.6.2 1 サイクルの予測

次サイクルにボールを捕捉できる場合，続くトラップ動作をより有理に行えるように特別な計算を行います．1 サイクルの予測は，`self_intercept.cpp` の `SelfIntercept::predictOneStep()` で実装されています．

捕捉条件

プレイヤエージェントは自分自身の体の方向にしか加速度を生成できません．1 サイクル後にボールを制御下におくためには，少なくとも以下の条件を満たさなければなりません．

- 次サイクルのプレイヤエージェントの慣性による移動後の位置を原点とし，プレイヤエージェントの体の方向を X 軸とする．この座標系での次サイクルのボール位置座標を $brpos$，プレイヤエージェントのキック可能距離 (キャッチ可能距離) を $ctrl\_dist$，プレイヤが生成できる加速度の大きさの最大値を max_accel とすると，
  - $|brpos.y| <= ctrl\_dist$
  - $|brpos.x| <= ctrl\_dist + max\_accel$
- 特に，$|brpos.y| <= ctrl\_dist$ かつ $|brpos.x| <= max\_accel$ であれば，次サイクルに必ずボールを制御下に置くことができる．

図 4.9 に例を示します．

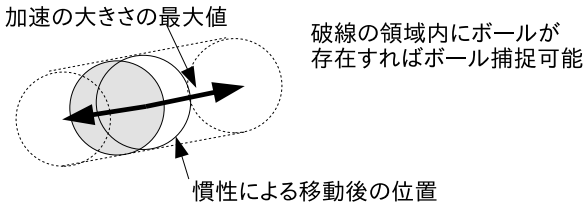


図 4.9 1 サイクル後にボール捕捉できる可能性がある状態

最適なトラップ位置の推定

次サイクルにボールを制御範囲内に置けるのであれば，より制御しやすい位置でのボールトラップが望まれます．ボール捕捉の確実性を高めつつボールとの衝突を避けるには，プレイヤエージェントの ***kickable_margin*** の中間位置が最適と言えます．このときのボールとプレイヤエージェントとの間の距離 $trap\_dist$ は，

$$trap\_dist = \boldsymbol{player\_size} + \boldsymbol{kickable\_margin}/2$$

ただし，$|brpos.y| > trap\_dist$ であれば，$trap\_dist$ でのトラップは不可能です．この場合，可能な限りボールに近付くようにするには，ボールをプレイヤの真横に置くようにダッシュしなければなりません．また，max_accel の大きさによっても，$trap\_dist$ でのトラップを実現できない場合があります．この場合は，max_accel

を生み出す **dash** コマンドを実行することで，可能な限りボールに近付くことになります．

図 4.10 に例を示します．図に示すように，最適なトラップ位置は最大で 2 つ存在します．このとき，よりキックパワー効果率の高い方を選択すべきでしょう．

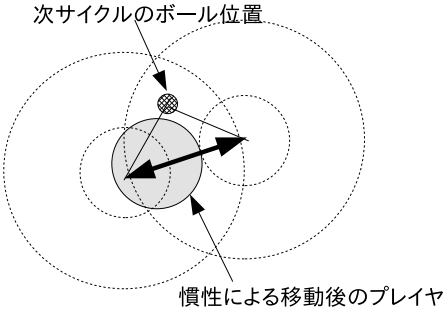


図 4.10 最適なトラップ位置候補

ボールを制御化に置くだけであれば，トラップ位置を調整するように **dash** を実行するだけで良いでしょう．しかし，トラップ位置が最適でないとしても、充分安全にボール捕捉ができると予測できたならば，**dash** ではなく **turn** を実行すべきです．これによって，次の目標位置へ向かう準備をしつつ，ボールトラップも同時に行えます．この判定は，`SelfIntercept::predictNoDash()` で実装されています．

4.6.3 複数サイクルの予測

1 サイクルでのトラップが不可能な場合，複数サイクル後の状態の予測が必要になります．これはサイクル数をループ変数とするループ処理になります．複数サイクルの予測は，`self_intercept.cpp` の `SelfIntercept::predictLongStep()` で実装されています．

最短捕捉サイクルの推定

ボール捕捉に最低限必要なサイクルを推定できれば，ループ開始のサイクル数を決定し，無駄な計算を省略できます．最低限必要なサイクルとは，プレイヤエー

ジェントがボールに到達できる最短移動距離から求められます．librcsc では，プレイヤエージェントからボールの軌道までの距離を，この最短移動距離としています．よって，プレイヤエージェントからボールの軌道までの距離を $bline\_dist$ とすると，推定最短捕捉サイクル min_cycle は，

$$min\_cycle = ceil(bline\_dist/\mathbf{real\_speed\_max})$$

$ceil(x)$ は，x よりも大きい最小の整数値を返す関数，***real_speed_max*** はプレイヤエージェントが到達できる最大スピードです．図 4.11 に例を示します．

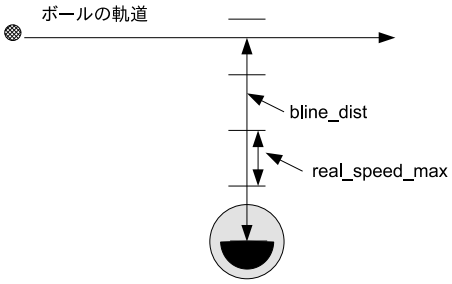


図 4.11 最短捕捉サイクルの推定

最短捕捉サイクルをより厳密に求めることはもちろん可能ですが，計算量と実装の手間を考えると，上記手法が適切でしょう．

ループ変数 n の初期値が求まれば，ループの開始です．n サイクル後の予測ボール位置座標 $bpos_n$ は簡単に計算できます．後は，プレイヤエージェントが n サイクルで $bpos + n$ 近傍に到達し，ボールを制御可能範囲内に置くことができるかどうかの判定の繰り返しです．

turn 回数の予測

プレイヤエージェントの移動の予測においては，前述のように，$bpos_n$ へ体を向けるのに必要なサイクル数 n_t をまず推定します．回転に必要なサイクル数の推定は，`SelfIntercept::predictTurnCycle()` で実装されています．

ここで必要となるのは，要求される **turn** コマンドの回数の推定です．4.2.4 節で説明したように，プレイヤエージェントのスピードに応じて実際の回転角度はコマンドの引数よりも小さくなるため，目標方向への回転が 1 サイクルで完了し

ないことがあります．SelfIntercept::predictTurnCycle() では，回転のシミュレーションを行うことで，**turn** コマンドの回数を求めています．ここでは，単純に目標方向と体の向きとの角度差が閾値を下回るまで回転を繰り返し，このループ回数を **turn** コマンドの回数としています．

以下は，この実装の例です．SelfIntercept::predictTurnCycle() においても，ほぼ同じ実装を見つけることができます．

```
int n_turn = 0; // turn コマンドの回数
// self：SelfObject の参照
const PlayerType & my_type = self.playerType();
// target_angle：目標方向
double angle_diff = ( target_angle - self.body() ).degree();
double speed = self.vel.r();
// turn_margin：角度差の閾値
while ( angle_diff > turn_margin ) {
  double max_turnable
    = my_type.effectiveTurn( ServerParam::i().maxMoment(), speed );
  angle_diff -= max_turnable;
  speed *= my_type.playerDecay();
  ++n_turn;
}
```

角度差の閾値は，プレイヤエージェントのキック可能距離 (キャッチ可能距離) から逆算できます．これは，通常の目標位置への移動と同じアルゴリズムで実現できます．

dash による到達距離の予測

次に，ダッシュによって到達できる距離を推定します．**dash** コマンドを実行できる回数 n_d は，回転の実行回数 n_t から，

$$n_d = n - n_t$$

n_d 回の **dash** コマンドの実行によって，ボールを制御距離範囲内に置ける位置へ移動できるならば，インターセプト成功と判断されます．この判定は，SelfIntercept::canReachAft
で実装されています．

ダッシュの予測においては，プレイヤエージェントの残りスタミナも考慮しなければなりません．そのため，基本的には **dash** コマンドによる速度，位置，ス

タミナの更新を逐一シミュレートすることになります．最終的に，ボールを制御可能距離内に置くことができる位置に到達できるか，または．$bpos_n$ よりも遠くに到達できるならば，インターセプト成功となります (図 4.12)．

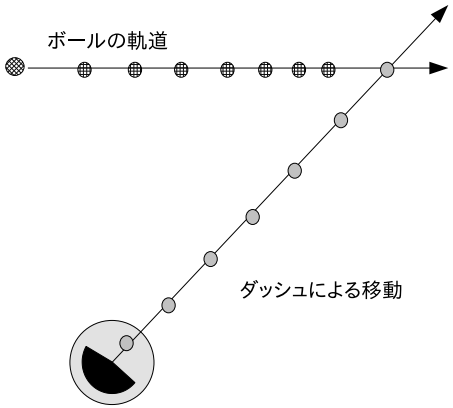


図 4.12 dash による到達距離の予測

後方ダッシュの利用

`SelfIntercept` クラスには，後方ダッシュによるインターセプト予測も含まれています．後方ダッシュによるインターセプトは，プレイヤの回転によるノイズの影響をより小さくすることを目的としています．4.2.4 節で説明したように，プレイヤの回転にはノイズが含まれます．このノイズを予測することはできないため，要求される回転角度が大きくなる程，回転に要する推定サイクル数の信頼性は低下します．よって，後方ダッシュを使った方が，要求される回転角度がより小さくなることがあることは容易に想像できます．ただし，後方ダッシュには通常の 2 倍のスタミナ消費が発生するため，**dash** コマンドの実行回数が充分小さい場合にのみ，後方ダッシュを使用すべきです．librcsc では，後方ダッシュを使うかどうかの判定には埋め込みのマジックナンバーを使っています[2)]．あまり良い実装とは言えませんが，現状では特に不具合は発生していないため，あえて修正する必要は無いでしょう．

[2)] ボール捕捉位置との方向差が 100 度以上，距離がが 5m 以内 (キーパの場合は 10m 以内)

4.6.4 予測の実行タイミング

ボールの軌道予測，プレイヤエージェント自身の移動予測を可能な限り正確に行う必要があるため，他の動作に比べて計算量が多くなります．ボールを持っていないプレイヤは常にインターセプトを意識しなければならないため，インターセプトの予測は常に行われています．そこで，librcsc では，計算の無駄を省くためにプレイヤエージェントの意志決定直前にインターセプトに関する予測をあらかじめ実施するようにしています．そのため，このインターセプト予測に限っては，実装ポリシーが他の動作クラスと異なります．インターセプト動作自体は `Body_Intercept` クラスで実行しますが，この動作を行うための予測は `SelfIntercept` クラスが担当しています．

プレイヤエージェント自身のインターセプト予測と同時に他のプレイヤのインターセプト予測も行われます．他のプレイヤについては，`PlayerIntercept` クラスが予測を担当します．インターセプト予測のアルゴリズムは，`SelfIntercept` とほぼ同じです．ただし，プレイヤエージェント自身の場合と比べて，他のプレイヤの情報の多くは欠落しているため，予測のための計算はかなり簡略化されたものになっています．

インターセプト予測の結果は，`InterceptTable` クラスに保持されます．インターセプト予測の実行は `WorldModel::updateJustBefore()` 内で `InterceptTable::update()` を呼び出すことで行われます．プレイヤエージェント自身のインターセプト予測結果は `InterceptInfo` クラス型の変数として `InterceptTable` に保持されます．`InterceptInfo` では，インターセプトに要するサイクル数，必要なダッシュパワー，***recover*** を消費すること無く動作完了できるかどうか，などの情報が保持されています．`InterceptTable` クラス自体は `WorldModel` クラスのメンバ変数として用意されているので，`PlayerAgent` からも予測結果を自由に参照できます．

Section 4.7
スマートインターセプト

前節で説明した方法でインターセプト成否の判定が可能になりました．しかし，以下のような理由のために，単純にもっとも早くボール捕捉できる動作を実行すべきではありません．このような戦略的，戦術的な先読みを含んだインターセプトを，本書ではスマートインターセプトと呼びます．

- 観測誤差や物体の移動ノイズによって，予測結果どおりにインターセプトが成功するとは限らない．
- 意図的にボール捕捉を遅れさせた方が，戦略的に有理になる場合がある．

ただし，他のプレイヤのインターセプト予測サイクルよりも，プレイヤエージェントのインターセプト予測サイクルが小さくなければ，スマートインターセプトを実行すべきではありません．

4.7.1 精度の向上

理由のひとつめは，インターセプト動作の精度を向上させる意味で重要です．観測誤差や移動ノイズは想像以上に大きいものであるため，より確実にボールを捕捉するには，余裕を持って先回りする動きが必要になります．一般的には，**turn** コマンドの実行回数がより少ない方が，インターセプトの失敗は減るでしょう (図 4.13)．

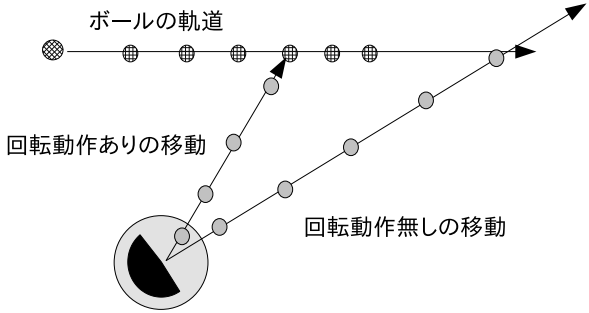


図 4.13 turn を実行しないインターセプト

4.7.2 戦略，戦術の考慮

理由の二つめは，例えば，フォワードプレイヤはより前方でボールを受けた方がより有利になります (図 4.14)．また，既にボール捕捉できる位置に到達していれば，スタミナ消費を抑えるために無駄な **dash** コマンドを実行せずに待機するといった戦術も考えられます (図 4.15)．

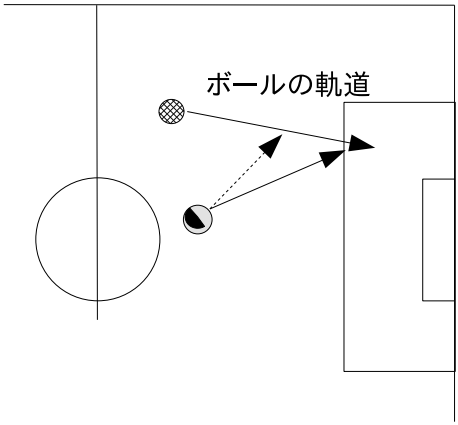


図 4.14 より前方でボールを捕える

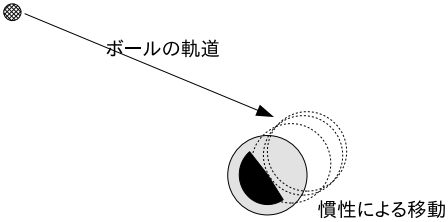


図 4.15 インターセプト位置での待機

4.7.3 改善すべき点

スマートインターセプトは，Body_Intercept クラスで既に実装されています．しかし，インターセプト位置の決定アルゴリズムには改善すべき点が多く残っており，本来の目的を完全に達成しているとは言い難い状態です．

本書執筆時の実装では，ルールを埋め込んで，それにマッチしたインターセプト位置を選択するという方法を取っており，柔軟性に乏しくなっています．より良いアルゴリズムを見つけたならば，ぜひ改良してみてください．例えば，各インターセプト位置の評価関数を用意し，その返り値による評価値を比較して最良

のものを選択する，といった方法が考えられます．C4.5[8] のような決定木アルゴリズムも適しているでしょう．

Section 4.8
キックによるボールの加速

目標の方向へ目標の初速で蹴り出す，という基本的な動作ができなければ，サッカープレイヤとして満足に動作することはできません．本節では，適切な **kick** コマンド引数を生成するために必要な知識を解説します．

4.8.1　加速度の求め方

ボールの目標初速度 v_t が決まっており，ボールの現在の速度 v_c が分かっているならば，v_t を達成するために必要な加速度 $accel$ は以下の式で得られます．

$$accel = v_t - v_c$$

非常に簡単なベクトルの計算であることが分かります．これを図で書くと，図 4.16 のようになります．

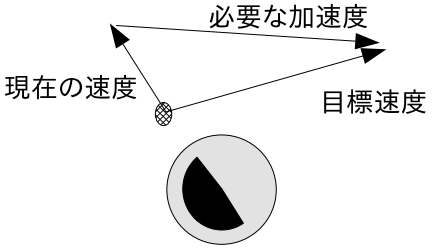


図　4.16　必要な加速度ベクトルの求めかた

4.8.2 キックパワーの求め方

必要な加速度ベクトルが得られれば，その大きさから **kick** コマンドに必要なパワーが求められます．キックの効果率の求め方は 4.2.2 節で説明したとおりですが，librcsc では現在のキック効果率を得るための関数を `SelfObject` のメンバとして用意しています．これによって，以下のような実装で必要なキックパワーを求めることができます．

```
Vector2D target_vel( 1.0, 0.0 );
Vector2D accel = target_vel - agent->world().ball().vel();
double kick_power = accel.r() / agent->world().self().kickRate();
```

ここで，**kick** コマンドの引数には，[-100,100] の範囲の値しか指定できないことに注意してください．もし，`kick_power` が 100 よりも大きければ，一回のキックで目標速度を達成できないことになります．すなわち，目標速度を達成するためには，キックを連続して実行し，加速度を累積させなければならない場合があります．複数回の連続キックのプランニングについては 4.11 節で説明します．

4.8.3 速度の方向の調整

速度の大きさには多少の誤差があっても良いので，方向だけは合わせたいという場合もあります．そのような場合は，図 4.17 のような，最大加速度の大きさを半径とする円を考えると計算が簡単になります．

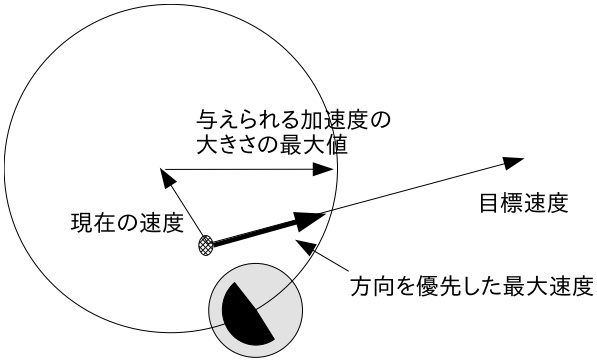


図 4.17 速度の方向の調整

これは，Body_KickOneStep::get_max_possible_vel() で実装されています．

Section 4.9
キッカブルエリア内でのボール制御

キッカブルエリア内でのボール制御スキルは，ほとんどのキック関連スキルで必要とされます．ボールを蹴り出すのでは無く常に一定範囲内に置くという点で，他のキック動作とは異なります．このボール制御動作は Body_KickToRelative クラスで実装されています．

4.9.1 目標位置へのボール移動

現在のボールの相対位置 $bpos_c$ から，次のサイクルにキッカブルエリア内の相対位置 $bpos_n$ へボールを移動させる場合，移動後のボールの速度 $bvel_n$ は，

$$bvel_n = (bpos_n + self\_vel - bpos_c) \times \textbf{ball\_decay}$$

となります．$self\_vel$ はプレイヤエージェントの現在の速度です．よって，$bvel_n / \textbf{ball\_decay}$ を目標初速度とするキックと考えれば，前節と全く同じ考え方で必要なキックパワーを求められます．

4.9.2 サブターゲットの生成

必要な加速度が現在のキック効果率では到達できない大きさだった場合，目標位置との中間位置にサブターゲットを生成し，一回のキックを複数回のキックへと分割します．図 4.18 に示すように，一回のキックでの移動が不可能であれば，その中間位置をサブターゲットとします．もし，このサブターゲットにも一回のキックで到達できないならば，更に再帰的にサブターゲットを生成します．この方法によって，目標位置へのボールの移動が確実に実行できます．

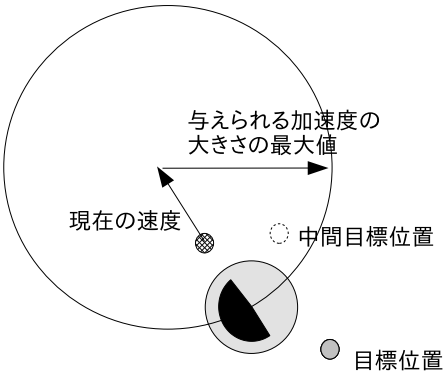


図 4.18 サブターゲットの生成

Section 4.10
ボールキープ

プレイヤエージェントのキッカブルエリア内でボールをキープする動作は，敵プレイヤからボールを守りつつ周囲を確認する余裕を生み出す，重要なスキルです．ボールキープ動作は `Body_HoldBall` クラスで実装されています．

4.10.1 キープ位置

ボールキープは敵プレイヤの回避が目的です．ここで，どこにボールを置くべきか？という問題が発生します．ボールの位置としては以下の 3 箇所が考えられます．

- プレイヤエージェントの体正面
- 敵プレイヤの体の向きからずらした位置
- 敵プレイヤから最も遠い位置

体正面でのキープ

体の正面にボールを置くことのメリットは，キック効果率が最大に近くなることです．これは，1 回のキックで目標速度に到達できる確率が増加するため，敵プレイヤが至近距離に近付いた場合にも素早く対応できることを意味します．どこかへ蹴り出すのに複数回のキックが必要となった場合，そのキックの途中で敵プレイヤに妨害される可能性を 0 にすることはできません．よって，より素早く，シンプルに動作できる体正面でのキープは非常に効果的と言えます．

敵プレイヤの状態を考慮できる場合

敵プレイヤの体の向きや速度が分かっている場合，そのプレイヤが次のサイクルでは決してボールに振れることができない位置を求めることができます．プレイヤは体の向きにしか加速度を生成できないので，敵プレイヤの中心を通り体の向きに伸びる直線からの距離がキック可能距離以上である位置は，少なくとも次サイクルには安全な位置と言えます (図 4.19)．敵プレイヤが至近距離に近付いて体正面でのキープが危険になった状況では，このキープ位置は高い効果をもたらします．

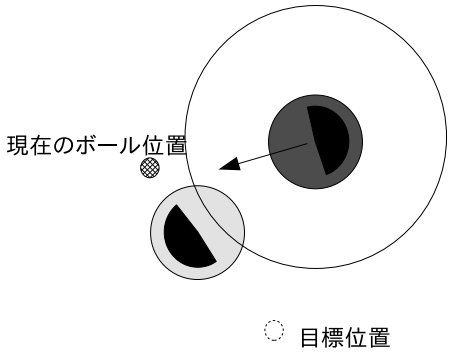


図　4.19　敵プレイヤの体の向きからずらした位置でのキープ

敵プレイヤの状態を考慮できない場合

敵プレイヤの体の向きや速度が不明で，位置座標しか得られなければ，次の敵プレイヤの動きは全く予測できません．この場合，敵プレイヤから最も遠い位置，

すなわち，自分を中心として敵プレイヤと対称な位置が最も安全な位置となります (図 4.20).

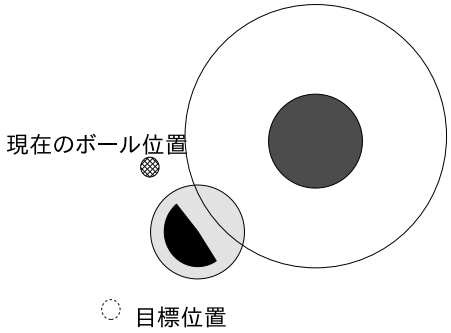


図 4.20 敵プレイヤから最も遠い位置でのキープ

このキープ位置では，次サイクルに敵プレイヤにボールに触れられてしまう可能性は 0 ではないことに注意してください．rcssserver の物理モデルでは物体同士がすり抜けることが可能なので，敵プレイヤがプレイヤエージェントの体を飛び越してボール側に到達してしまうことがあるためです.

4.10.2 改善すべき点

本書執筆時の `Body_HoldBall` によるボールキープの実装では，最近傍の敵プレイヤしか考慮していません．そのため，複数の敵プレイヤが至近距離に近付いた場合には全く対応できません．より確実性を求めるならば複数プレイヤも考慮すべきですが，現実にはそのような状況はまれにしか発生しないため，実装していません.

また，敵プレイヤの状態の推定や動きの予測もそれほど厳密には行っていません．厳密にやったところでそれほど効果が無いだろうと考えたためですが，より厳密に動きを予測すれば，キープの精度は向上するかもしれません.

そして，キープの安全性をより高めるには，プレイヤエージェントは一定位置に留まるべきではありません．必要に応じてボールをスペースへ蹴り出し，プレイヤエージェント自身が移動するようにした方がより安全です．このような動作はドリブルに近いものであるため，戦略レベルの意思決定でドリブルと組み合わせることで代用するのも手です.

Section 4.11
スマートキック

前述のように，一回のキックではボールを目標速度に到達させられない場合があります．このような場合，連続してボールを蹴り，加速度を累積することで目標速度に到達させることができますが，ここでキック列のプランニングという問題が発生します．これまでに，キック列のプランニングを組合せ最適化問題，または，経路探索問題として捉えることで解決が試みられてきました．このようなやや複雑なキック動作を本書ではスマートキックと呼びます．

4.11.1 状態の離散化

rcssserver 上の仮想フィールドは連続空間であり，プレイヤのキッカブルエリア連続値を持ちます．そのため，キック列の組合せは無数に存在します．まずはこれを適度に離散化する必要があります 4.21．これは，キッカブルエリア内に固定の目標位置を設定し，ボールはその位置のみを経由するように制限されることを意味します．この状態の離散化の精度が粗ければそれだけキック動作の柔軟性は乏しくなります．逆に，精度が細かければ柔軟性には富むものの，計算量が爆発的に増大してしまいます．状態の離散化の精度をどの程度にするかはプランニングに使用するアルゴリズムによって変更しなければなりません．

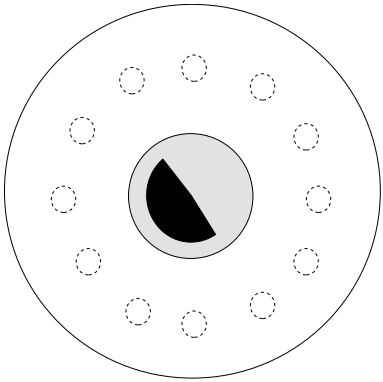


図 4.21 キッカブルエリア内での固定目標位置の設定

4.11.2 単純な方法

最も単純で実装が簡単なのは，全ての組合せをしらみつぶしに探索する方法です．本書執筆時点の librcsc では，ほぼこの方法での探索を行っています．`Body_KickOneStep`, `Body_KickTwoStep`, `Body_KickMultiStep` がキックプランニングの実装となっています．

必要なボールの初速度を生成できるか否かは，ボールリリース位置でのキック効果率と最後のキック直前のボール速度のみが関係します．すなわち，最後の 2 回のキックの組合せが重要と言えます．librcsc では，最後の 2 回のキックを準備するためのキックと合わせて最大 3 回までのキックを考慮することにしています．

librcsc の実装では，キッカブルエリア内に 12 個の固定目標位置を設定しています．よって，キック列の組合せの総数は，

$$12^2 + 12 + 1$$

となります．組合せと言っていましたが，実際には順列の計算になります．上式の第一項はキックを 3 回実行する場合の全組合せ[3)]，第二項はキックを 2 回実行する場合の組合せ[4)]，そして，第三項は 1 回のキックで目標の初速を達成できる場合です．最大で 157 通りの組合せを探索することになります．実際にはより少ないキック回数のものから探索を実行し，その時点で目標の書速度を達成できる

[3)] 2 箇所の固定目標位置を通過するので，組合せの総数は 12^2 となります．
[4)] 1 箇所の固定目標位置を通過するので，組合せの総数は 12 となります．

ものが発見されればそこで探索は打ち切られます．このため，実際には全組合せを探索するわけではありません．この程度であれば，計算量もそれほど大きくなく，現実的な時間で解が得られます．

4.11.3 敵プレイヤ回避とフィールドの考慮

ボールを加速している間に敵プレイヤが触れられる位置にボールを置いてしまっては，その敵プレイヤに妨害されて目標の初速を達成することはできないでしょう．キッカブルエリア内の固定目標位置が敵プレイヤのキッカブルエリアに入っていないかのチェックが必要です．これは簡単に実現できます．

また，キッカブルエリア内でボールを移動させている間にフィールドの外にボールを出してしまっては意味がありません．そのため，各固定目標位置でフィールドの内外判定を行い，フィールドの内側の目標位置のみを使用するように制限します．これも簡単に実現できます．

4.11.4 より高度な方法

前述の単純な探索方法では，橘花ブルエリアを高々12 分割しかできませんでした．離散化の精度をより細かくすることは簡単ですが，探索空間は爆発的に大きくなり，計算量は指数的に増加してしまいます．探索空間を増やしても実際の探索量を抑えるための工夫が必要です．現在，最も良い方法と考えられるのは，RoboCup2001,2002 を連覇した TsinghuAeolus というチームが導入した，強化学習と A*探索を利用した手法でしょう．

強化学習とは，人工知能分野では広く知られた学習手法で，試行錯誤を通じて環境に適応する学習制御の枠組みです．状態の入力に対する出力を明示的に教示しない，教師無し学習の一種です [10]．

A*探索とは人工知能分野においては古典的な経路探索手法で，現在ではゲームにおける経路探索などにも広く応用されている手法です [1]．

TsinghuAeolus の手法でも，キッカブルエリア内に固定目標位置を設定することには変わりありません．ただし，ある目標位置からもう一つの目標位置を結ぶエッジに評価値を設定します．この評価値は強化学習によるオフライントレーニングで獲得されます．TsinghuAeolus は，このオフライントレーニングに Q 学習を採用していたようです．そして，この評価値をヒューリスティック関数として A*探索を実行します．同時に，敵プレイヤによる妨害とフィールドの内外判定のチェックも行います．

この手法はオフライントレーニングとオンライン探索を組み合わせて使用しており，非常に効果的かつ頑健性も高いと言えます．

4.11.5 改善すべき点

本書執筆時のスマートキック実装はかなり単純なアルゴリズムを採用しています．実用上，ほとんど支障は出ていませんが，敵プレイヤの回避という点ではやや精度が落ちます．回避の精度を上げるには探索空間を大きくしなければならないため，A*探索のような手法を組み合わせて使用する必要があるでしょう．このような実装は既に librcsc にも実装を始めているものの，まだテスト段階で実用には至っていません．しかし，今後の更新によって既存の実装から置き換えられ，実用される可能性があります．

Section 4.12
ドリブル

ドリブルという動作は，キックの後にボール捕捉動作を行って再び自分自身でキック可能な状態にする，という一連の動作の繰り返しです．すなわち，ボールをキック可能な場合はボールを目標位置へキック， キック可能でない場合はインターセプト，というルールだけで最も単純なドリブルが実現できます．しかしながら，これでは動きに無駄が多く，敵プレイヤがボールを奪う隙を与えてしまうでしょう．ドリブルの実行においては，より無駄無く効率的に，そして，安全にボールを運べるように，ボールを上手くコントロールすることが重要です．

4.12.1 基本的なアルゴリズム

librcsc で実装しているドリブルのアルゴリズムは以下のようになります．

1. ドリブルを妨害する敵プレイヤが存在する場合，
   (a) 回避目標方向を探索
   (b) 新しい目標方向へ向かって改めてドリブル

2. プレイヤエージェントの体が目標位置へ向いていない場合,
   (a) 慣性による移動後にボールがキック可能であれば，回転
   (b) そうでなければ，回転完了後にキック可能な状態になるようにボールをキック
3. プレイヤエージェントの体が目標位置へ向いている場合,
   (a) 1 回のダッシュ後にキック可能な状態であれば，ダッシュして終了
   (b) n 回のダッシュ後のボールとの位置関係を決定
   (c) n 回のダッシュ後のプレイヤエージェントの位置を推定
   (d) n 回のダッシュ後，適切な位置関係でキック可能な状態になるようにボールをキック
   (e) ドリブルの意図を登録して終了

以上の実装は，`Body_Dribble` クラスで見つけることができます．`Body_Dribble` では，ドリブルのために必要なキック動作の推定と敵プレイヤを回避する方向の探索に多くの計算資源が消費されています．

ここで，以上のアルゴリズムではキック後の回転やダッシュ動作が考慮されていない点に注意してください．`Body_Dribble` で行われるのは，ドリブルのために必要な動作のプランニング，そして，最初のキックだけです．ドリブル実行中の移動動作は `Intention_Dribble` クラスで実装されています．`Intention_Dribble` は意図クラスであり，実行予定の **turn** と **dash** のそれぞれの回数が与えられます．そして，ドリブル動作の最後に `PlayerAgent` に登録されます．

`Intention_Dribble` では，以下のようなアルゴリズムが実装されています．

1. ドリブルを妨害する敵プレイヤが存在すれば，意図を中断して終了
2. **turn** の実行回数が残っている場合,
   (a) 既に目標位置の方向への回転が完了していれば，予測と結果が異なるので意図を中断して終了
   (b) 目標位置の方向へ回転するための **turn** コマンドを実行し，**turn** の実行回数をデクリメントして終了
3. **dash** の実行回数が残っている場合,

(a) ボールとプレイヤエージェントとの左右のずれが大きくなっていれば，ダッシュ完了後にキック可能な状態になれないので，意図を中断して終了

(b) ボールを追い越してしまい，キック可能状態にもなれない場合は，意図を中断して終了

(c) 指定のパワーで **dash** コマンドを実行し，**dash** の実行回数をデクリメントして終了

このように，エラーチェックをしつつ，最初にプランニングした回転とダッシュを遂行しようとします．プランニングと実際の行動の実装が分離されているため，比較的ソースコードの見通しが良くなっているのではないかと思います[5]．ドリブルは，意図クラスを利用することで実装を容易にすることができる代表的な動作でしょう．

4.12.2 ボール位置の推定

`Body_Dribble` においては，以下の二種類のキックの予測が必要とされます．

- 回転中にキック可能な状態を維持するための調整キック
- 複数回ダッシュ後に適切な位置でのボールトラップを可能にするキック

いずれも，n サイクル後にボールが指定位置に到達するようにボールをキックするという点で同じです．必要なボールの初速度 $bvel_0$ は以下の計算で求めることができます．n サイクル後のボールの総移動ベクトルを $ball\_move$ とすると，

$$ball\_move = bvel_0 \times (1 + ball\_decay + \ldots + ball\_decay^n)$$

よって，

$$bvel_0 = \frac{ball\_move}{((1 - ball\_decay^n)/(1 - ball\_decay))}$$

$ball\_move$ は，プレイヤエージェント自身の移動ベクトルから求めることができます．実装においては，n の値を間違えないように注意してください．特に，最初のキック 1 回分を考慮するのを忘れがちです．

[5] ただし，本書執筆時のコードは読みやすいとは言い難い状態です．後々修正していければと考えています．

以上が基本的なボール位置推定の考え方です．後は，次のボールトラップ時の位置関係を適切に設定すれば，ドリブルの動きは相当スムーズになるでしょう．次のボールトラップ時の位置関係としては，プレイヤエージェントの斜め 45 度前方でキック可能領域の中間あたりが良いでしょう．

また，この方法ではサイクル数に応じたボール位置推定を行っているため，ドリブル実行時に何回ダッシュを行うかを自由に変更できるようになります．ダッシュ回数を変えられるということは，それだけ柔軟な対応が可能になるということを意味します．ただし，ダッシュ回数が増すと物体の移動ノイズも増すため，ボールコントロールの精度が落ちてしまい，ボールを横にこぼす可能性が高くなる点に注意してください．

より高度な動き，例えば，常にボールをキック可能量域内に置いたまま移動する，などといった動きを実現することも可能ですが，本書執筆時の librcsc では実装されていません．実現のためには，次のボールトラップ時の位置関係やダッシュ回数まで含めたプランニングが必要になります．

4.12.3 ボールとの衝突の利用

プレイヤエージェントの体の向きが目標位置へ向いていない場合，最初に回転動作が必要です．しかし，慣性モーメントのために 1 サイクルで回転が完了できないことがあります．特に，ドリブルはインターセプト直後の動作になるので，プレイヤエージェントがトップスピードの状態であることも珍しくありません．この問題はスマートインターセプトによってある程度解決できますが，ドリブル途中で方向転換する場合には，素早い切り返しができなければ敵プレイヤにボールを奪われやすくなってしまいます．

より少ないサイクル数で回転を完了させるには，プレイヤエージェントのスピードが小さい方が有利です．ここで，rcssserver における物体の衝突モデルを思い出してください．このモデルでは，移動物体動詞が衝突すると，いずれの物体も速度が-0.1 倍されます．すなわち，通常の速度減衰よりもはるかに大きい速度減衰が得られるのです．`Body_Dribble` では，この仕様を逆手に取って，プレイヤエージェントとボールとを意図的に衝突させるようにしています (図 4.22)．意図的な衝突は，次サイクルのボール位置を，次サイクルのプレイヤエージェントの中心位置になるようにボールをキックするだけで実現できます．実装は，`Body_Dribble::collideWithBall()` で見つけることができます．

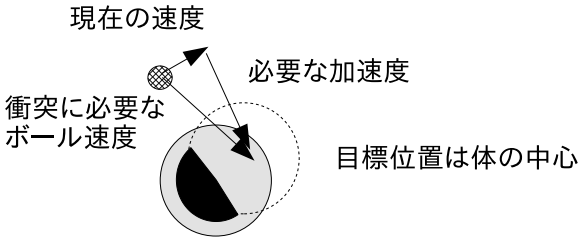


図 4.22 ボールとの意図的な衝突

4.12.4 敵プレイヤの回避

守備の上手いチームのプレイヤは，ドリブルしているプレイヤの進行方向へ先回りし，進路を妨害しようとします．このような進路妨害への対応として，Body_Dribble には敵プレイヤをを回避する動きがすでに組み込まれています．回避のための動作は，Body_Dribble::doDodge() で実装されています．

Body_Dribble における敵プレイヤ回避は，以下の条件を満たしたときに発動します．

- 敵プレイヤがプレイヤエージェントの進行方向の扇型内に存在する
- ドリブル中にその敵プレイヤがボールに追い付く可能性がある

ただし，本書執筆時では，敵プレイヤのボール捕捉の予測を正確に行っていません．より正確な予測を行った方が良いのは間違いありませんが，実装と検証の手間がかかるため，単純な実装でごまかしている状態です．

回避すべきプレイヤが存在していた場合は，続いて回避する方向を決定します．この方向の計算は，Body_Dribble::getAvoidAngle() で実装されています．360 度全てを探すわけにはいかないので，ここでは，自分の周囲を分割して離散化し，その中から最も安全そうな方向を選択するという簡単な探索を行っています．各方向の検証では，その方向に一定距離進んだ先で一定半径の円領域を考え，その中に敵プレイヤが存在しなければその方向を安全とみなします．Body_Dribble::getAvoidAngle() では，最初の目標方向から近い順に探索を実行することで，目標位置に最も近い回避方向が得られるようになっています (図 4.23)．

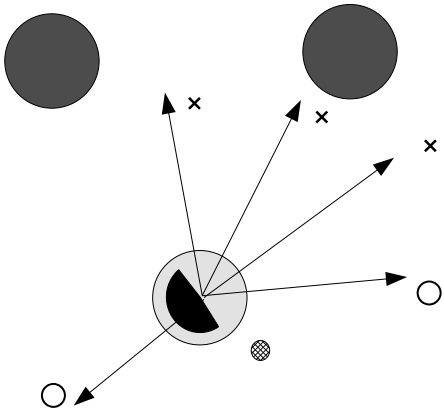


図 4.23 回避方向の探索

Body_Dribble::getAvoidAngle() では，回避方向として，プレイヤエージェントの現在の体の前後の方向も探索します．プレイヤの回転には 1 サイクル以上の時間が必要であるため，回転無しで回避した方がより安全です．多くの場合，後向きに一瞬下がることで敵プレイヤを回避することができます．Body_Dribble ではバックダッシュによるドリブルも可能となっているので，利用してみてください．

4.12.5 改善すべき点

Body_Dribble は任意のダッシュ回数を指定できるという柔軟なドリブル動作を実現しているものの，ボールキープの精度はあまり高くありません．そのため，不用意にドリブルを始めると，敵プレイヤにボールを奪われてしまうことが多いでしょう．より上位の戦略や戦術レベルでの意思決定で解決できなくはありませんが，反射のレベルで回避できた方が楽なのは間違いありません．よって，常にボールをキック可能状態に置くドリブルの実装も必要と言えるでしょう．

ボールコントロールの精度も良いとは言えません．ボールをこぼしてしまう場面をしばしば見かけることでしょう．より精度の良いボールコントロールを実現するには，より厳密な予測を行う必要があります．または，何らかの学習手法を採り入れるのも良いかもしれません．ただし，その際はヘテロプレイヤによる能力の違いを忘れずに考慮してください．固定能力のプレイヤのドリブル動作の学習であればそれほど難しくはありませんが，どのような能力のプレイヤであって

も使えるような汎化能力の高い学習モデルの構築と実装はかなり難しい問題と考えられます.

更に，より高度な回避動作を実現するのも面白いでしょう．敵プレイヤの回避のために最初の目標とは異なる方向へ移動すると，スタミナを無駄に消費します．キック可能領域内でフェイントを仕掛けて敵プレイヤを惑わし，その隙に走り抜けるという動作ができれば理想的です．ただし，実現のためには敵プレイヤの情報管理を相当注意深く行わなければならないでしょう.

ドリブル動作はまだまだ改善できる余地が多く，また，敵プレイヤとの相互作用を考慮してプランニングを行わなければならないという点で非常に面白い問題だと思います．良いチームを作るには良いドリブラが必須なので，工夫してみる価値はあるでしょう.

Section 4.13
パス，シュート

パスとシュートの動作には，その動作が成功するか否かの予測が必要です．そのためには，誰が最も早くボールに追い付くかを予測するボール所有者判定を行うことになります．これは，インターセプトの予測方法を流用することで解決できる問題です．しかしながら，これらのキック動作には無数の選択肢 (ボール初速度) が存在するため，通常のインターセプト予測計算を行っていては，計算資源が不足してしまいます.

また，動作の選択肢が無数に存在するということは，結果が成功と予測される動作も無数に存在することになります．よって，これらの中からどれを選択し，実行するかという戦術的な意思決定に関わる問題も発生します.

4.13.1 成功判定の高速計算

全てのプレイヤは，ボールの速度を観測してからでなければインターセプト動作を開始することが出来ません．そのため，プレイヤエージェントがボールをキックしている間，他のプレイヤはボールを追いかけることが出来ません．ボールがリリースされて初めて，各プレイヤはボールの速度を観測でき，インターセプト動作を開始できます．ここで，プレイヤがボールをキックしている間はボールを

取られない，他のプレイヤは常にボールを観測し続けている，と仮定します．すると，パスとシュートの成功判定は，

- ボールのリリース直後，ボールが敵プレイヤの制御距離範囲内に存在する
- ボールのリリース後，敵プレイヤが最も早くボールに到達できる．

という 2 つの条件を検証することで実現できます (図 4.24)

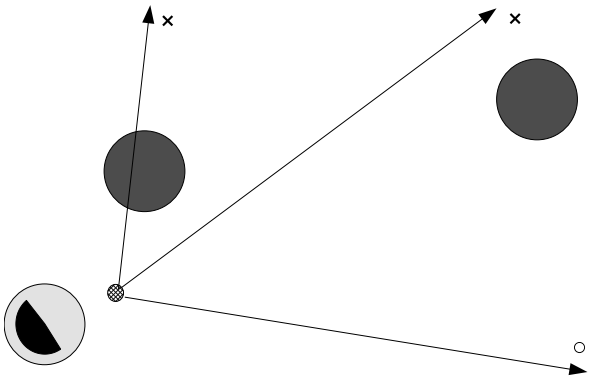


図 4.24 パスとシュートの成功判定

一つ目の条件の検証は簡単です．リリース後のボール位置と他のプレイヤの予測位置との間の距離を求めるだけです．一方，二つ目の条件の検証にはかなりの計算資源を必要とします．そこで，精度をある程度犠牲にした高速な計算手法が必要になります．librcsc では二種類の高速計算アルゴリズムを実装しています．

単純な方法

ここで説明するアルゴリズムは，実用においてそこそこの結果を出せ，しかも高速に動作させることができます．ただし，成功判定を適切に行っているものではありません．本書執筆時点の librcsc では実際に利用していますが，今後は利用されなくなっていく予定のアルゴリズムです．あくまで参考程度にとどめるよう注意してください．

判定の手順は以下のようになります：

- ボールの速度方向を前方とした場合にボールの移動開始位置よりも後方に存在する敵プレイヤは無視する (パスやシュート時のボールの速度はプレイヤの移動速度よりも速いと仮定).
- ボールの軌道の直線に対して，敵プレイヤの最短捕捉サイクル c_o とその位置 p_o を求める．これは，4.6 節で説明した最短捕捉サイクルの求めかたと同じ方法で求められる．
- ボールが p_o まで到達するのに必要なサイクル c_b を求める．
- $c_o > c_b$ であれば，このパスやシュートは敵プレイヤに取られないと判定される．

以上の判定を全ての敵プレイヤに対して行います．この方法では，ボールを途中で敵プレイヤに取られるか否かの判定のみを行い，敵プレイヤが何サイクルでボールに追い付くかの予測は行われません．また，敵プレイヤの最短捕捉サイクルしか考慮していないため，極めて大雑把な推定と言えます．それでも，実用上はかなりの効果を上げることが出来ます．この判定方法は，`Body_Pass` におけるパスの成功判定で利用しています．

ニュートン法の利用

インターセプト予測の高速計算手法として，ニュートン法を応用した手法が知られています [11]．以下で，この手法の概要を説明します．

リリース直後のボールの初速度の大きさを $first\_speed$，t サイクル後のボールの速度を $bvel(t)$，$t = 0$ のボール位置を原点とし，ボールの速度方向を X 軸正方向の座標系を考えます．この座標系において，t サイクル後のボールの位置は X 軸上にあり，これを $bx(t)$ とします．また，敵プレイヤの初期位置を p_0 とします．

このとき，$bx(t)$ は，

$$bx(t) = first\_speed \times \frac{1 - \boldsymbol{ball\_decay}}{1 - \boldsymbol{ball\_decay}^t}$$

t サイクル後の敵プレイヤの最大移動距離を $h(t)$ とすると，

$$h(t) = \boldsymbol{player\_speed\_max} \times t$$

t サイクル後のボール位置 $bx(t)$ と敵プレイヤの初期位置 p_0 の距離を $g(t)$ とすると，

$$g(t) = \sqrt{(bx(t) - p_0.x)^2 + p_0.y^2}$$

ここで，関数 $f(t)$ を以下のように定義します (対象となる敵プレイヤがキーパの場合，***kickable_area*** はキャッチ可能距離に置き換えられます．)．

$$f(t) = g(t) - h(t) - \textbf{\textit{kickable\_area}}$$

このとき，初期状態が理想的であれば，$f(t) < 0$ である t で，敵プレイヤはボールを捕捉できることになります．そして，rcssserver は離散時間シミュレータですが，ここでは連続時間として考えると，$f(t)$ は全微分可能となります．よって，$f(t) = 0$ となる t をニュートン法で求めれば，敵プレイヤがインターセプトに要するサイクル数の推定が可能となります．

$f(t)$ に対してある初期状態を設定してグラフで表すと，図 4.25 のようになります．

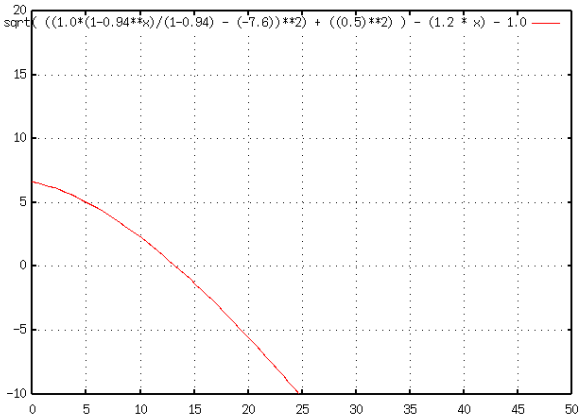


図 4.25 ニュートン法で解く関数の例 1

このような単調減少の関数であれば問題はありません．しかし，初期状態が異なれば，$f(t)$ は図 4.26 のようになることもあります．これは，こうなると，単純にニュートン法を適用するだけではうまくいかなくなります．

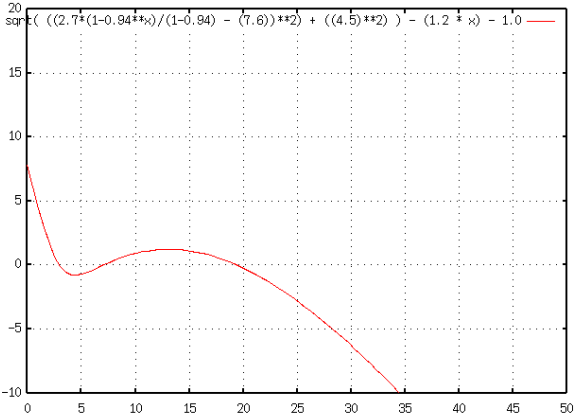


図 4.26 ニュートン法で解く関数の例 2

ニュートン法では，以下の漸化式を解くことで解が得られます．

$$t_{n+1} = t_n - \frac{f(t_n))}{f'(t_n)}$$

ここで，$f(t)$ を微分した $f'(t)$ が必要になります．

$$f'(t) = \frac{(bx(t) - p_0.x) \times bx'(t)}{g(t)} - \boldsymbol{player\_speed\_max}$$

$$bx'(t) = -\frac{first\_speed \times \boldsymbol{ball\_decay}^t \times log(\boldsymbol{ball\_decay})}{1 - \boldsymbol{ball\_decay}}$$

$bx'(t)$ では，$\boldsymbol{ball\_decay}^t$ 以外は定数なので，事前に計算しておくことが可能です．

導出過程は省略します．難しいものではないので，興味があれば自分でも計算してみてください．

数学的には以上で完了ですが，実用においてはもう少し工夫が必要です．例えば，$f(t)$ の傾きがほぼ水平で $f'(t)$ が 0 に近い場合や，$f'(t) > 0$ の場合は収束に時間がかかってしまいます．以上のような場合は，収束を早めるために特殊な計算を行う方が良いようです．

librcsc ではこのニュートン法を実装しており，シュートの成功判定に利用しています．以下のような実装を，`interception.{h,cpp}` の `Interception` というクラスで見つけることが出来ます．

```
double t = 0.0; // 経過サイクル数
double f; // f(t) の値
double f_d;  // f'(t) の値
int counter = 0;
do {
  ++counter;
  double ball_x = M_ball_x_constant * ( 1.0 - std::pow( bdecay, t) );
  double ball_x_d = M_ball_x_d_constant
                    * std::pow( ServerParam::i().ballDecay(), t );
  double dist_to_ball
     = std::sqrt( rcsc::square( ball_x - start_point.x )
                  + rcsc::square( start_point.y ) );
  f = dist_to_ball - player_max_speed * t - control_buf;
  f_d = (ball_x - start_point.x) * ball_x_d / dist_to_ball
        - player_max_speed;
  if ( ( ball_x < start_point.x && f_d != 0.0 )
       || ( ball_x > start_point.x && f_d < 0.0 ) ) {
    t = t - f / f_d;
  } else {
    t += f / player_max_speed;
    if ( f_d > 0.0 ) {
      // 傾きが正の場合，ボールはプレイヤよりも速い速度で遠ざかっ
      // ているので，強制的に一定サイクル数を追加して収束を早める.
      t += 10.0;
    }
  }
  if ( std::fabs( f ) < MIN_ERROR ) {
     break;
  }
} while ( counter < MAX_LOOP );
```

M_ball_x_constant と M_ball_x_d_constant は，ボールの初速度に対して一意に定まる定数値で，Interception クラスのコンストラクタで以下のように初期化されます.

```
Interception::Interception( const Vector2D & ball_pos,
                            const Vector2D & ball_vel )
    : M_ball_first_pos( ball_pos )
    , M_ball_first_speed( ball_vel.r() )
    , M_ball_vel_angle( ball_vel.th() )
    , M_ball_x_constant( M_ball_first_speed
                         / (1.0 - ServerParam::i().ballDecay()) )
    , M_ball_x_d_constant( (-M_ball_first_speed * logBallDecay())
                           / (1.0 - ServerParam::i().ballDecay()) )
{ }
```

4.13.2 パスコースの生成と評価

探索空間の削減

パスコースの成功判定を高速にできるとしても，無数に存在するパスコースの全てを検証することは不可能です．そこで，パスの探索空間を削減するために，離散的なパスコースを生成する必要があります．離散化のためには，パスの方向と初速度の大きさを一定単位ごとに制限する，という方法が妥当でしょう．

本書執筆時の librcsc におけるパスの実装では，以下の 3 種のパスコースを生成することにしています．

- ダイレクトパス
  レシーバの位置へ直接出すパス
- リードパス
  レシーバの位置から数メートルずらして出すパス
- スルーパス
  レシーバを前方へ走らせるパス

それぞれ，図 4.27 に示すようなパスコースになります．

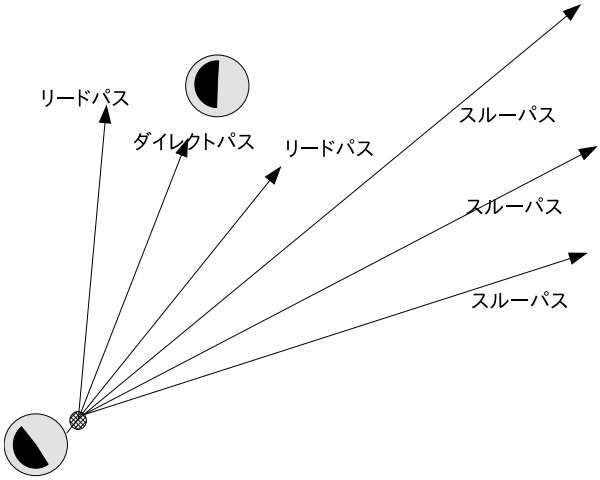


図 4.27 パスの種類

ここでは，どのパスもレシーバを中心に考えていることに注意してください．すなわち，パスコースを生成した時点でそのパスを受けるレシーバは決まっていることを意味します．librcsc では，このようにして，レシーバを探索する計算を省略しています．その代わり，探索空間が小さくなりすぎてしまい，意思決定における選択肢の幅が狭くなるという欠点を持っています．

理想的には，全方向に対して離散化したパスコースを生成し，それら全てを検証する，という実装が望ましいでしょう (図 4.28)．敵プレイヤによるインターセプトだけでなく，味方プレイヤの誰がレシーバになるかの予測も必要になりますが，離散化の精度を調整し，ニュートン法による予測計算を行えば，充分に実時間での探索が可能です．この方法であれば，意思決定における選択肢に大幅な柔軟性を持たせられます．しかし，逆に，柔軟性があるために評価が難しくなるという側面も持っています．

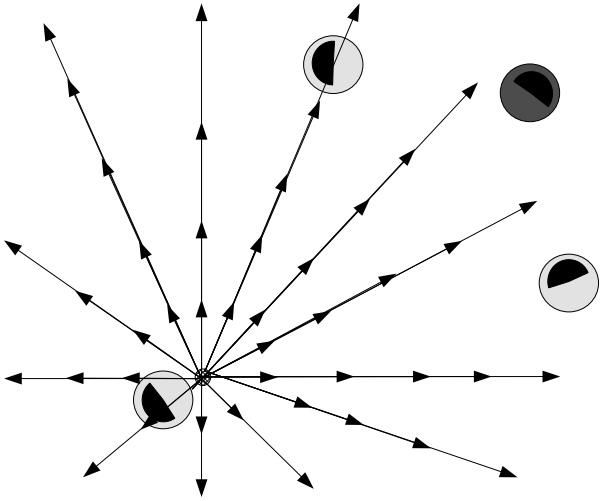


図 4.28 パスの探索

今後，librcsc では後者の探索方法へと移行していく予定です．最大の理由は，前者の方法では柔軟性が低く，チームレベルでの意思決定をシンプルなモデルにできないためです．

ルールによる排除

パスコースの探索空間を削減するためには，何らかのヒューリスティックに基づいて無駄なパスコースを事前に排除すると効果的です．例えば，以下のようなパスコースは検証するまでもなく排除できます．

- 味方ゴール前に出すパス
- 味方プレイヤが全く存在しない方向へのパス
- パスコースの方向の信頼性が著しく低い

librcsc では以上のようなルールを人手で実装しています．そのため，コードが汚く，効果も期待どおりのものではありません．この問題に対しては，決定木学習を利用してヒューリスティックなルールを獲得させると効果的であると予想さ

れます．筆者自身も以前から決定木を導入したいと考えており，今後の再実装を予定しています．

4.13.3 シュートコースの生成と評価

生成方法

シュートコースの生成においても，パスコースの生成と同様に探索空間を離散化して削減します．このとき，ボールの速度の方向は，敵ゴールの左右のポストの間に限定されます．通常は，ポストの間を 8〜10 分割程度すれば充分でしょう (図 4.29)．また，分割した結果，各コースの角度の差が小さすぎる場合は，分割の数を減らすとより効果的です．

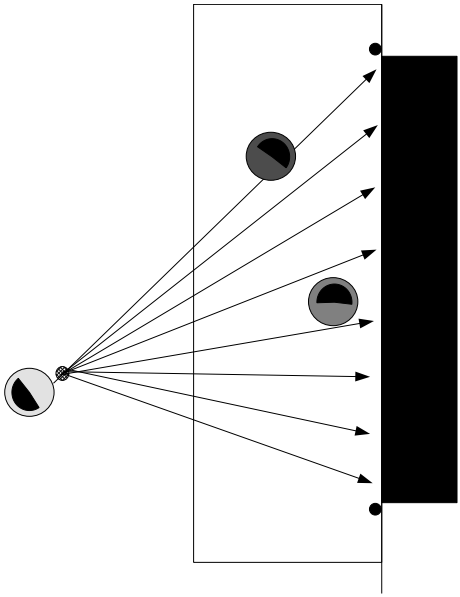


図 4.29 シュートコースの分割

初速度の大きさは大きければ大きいほど敵プレイヤに妨害されにくくなります．***ball_speed_max*** で蹴るのが理想的ですが，スピードを得るために **kick** コマンドの回数が増えてしまっては逆に妨害される可能性が高くなります．そのため，一

回の **kick** コマンドで与えられる最大のスピードを最小値とし，***ball_speed_max*** までの数段階のスピードを検証するのが妥当であると考えられます．

シュートの成功判定においては，敵プレイヤがボールに追い付くよりも先にボールがゴールラインを割れば成功とみなせます．そこで，まずはボールがゴールに配流までに必要なサイクルを推定します．そして，各敵プレイヤについてもインターセプトに要するサイクル数を推定します．特に，ペナルティエリア内のキーパのボールキャッチ能力は強力なので，敵キーパに関しては，他のフィールドプレイヤよりも厳密に推定を行う必要があります．

librcsc では，シュートコースに対するインターセプトサイクルの推定に，ニュートン法による計算方法を使用しています．実装は，`Body_Shoot::search()` で見つけることができます．

4.13.4 改善すべき点

パスとシュートには改善すべき点が多くあります．しかしながら，決め手となる手法はいまだに提案されておらず，それぞれのチームの開発者が独自のノウハウで調整を施しているのが現状です．筆者自身もまだまだ手探りの状態です．人間の持つ直感的判断能力をいかにして計算機に持たせるかという，人工知能の一般的案問題につながるテーマであるため，今後も研究を続ける必要があるでしょう．

Section 4.14
クリア

パスやシュートは敵プレイヤにボールを取られないことが前提の動作でした．しかし，クリアではこれらと同じ方法で動作のプランニングを行うことが出来ません．例えば，敵プレイヤに囲まれたときなど，パスコースがひとつも見付からなかった場合に実行すべき動作です．よって，敵プレイヤにボールを取られるかもしれないが，少なくとも現在よりは安全になる，という結果を得られるように動作のプランニングを行う必要があります．

ボールをクリアする方向を探索する場合には，

- より安全な位置へのボールの移動

- 敵プレイヤによるインターセプトを可能な限り遅れさせるボール速度

という二点が重要になります.

4.14.1 探索範囲の決定

一つ目の条件を満たすための単純な解決法として，現在のボール位置に応じて探索する方向の範囲を変化させる，という方法が考えられます．librcsc では，Body_ClearBall クラスにおいて以下のように実装しています.

```
const SelfObject & self = agent->world().self();
if ( self.pos().y > ServerParam::i().goalHalfWidth() - 1.0 ) {
   lower_angle = 0.0;
   upper_angle = 90.0;
} else if ( self.pos().y < -ServerParam::i().goalHalfWidth() + 1.0 ) {
   lower_angle = -90.0;
   upper_angle = 0.0;
} else {
   lower_angle = -60.0;
   upper_angle = 60.0;
}
```

これは，自分がゴールの幅よりも外側にいれば，その更に外側の斜め前方向の範囲を探索し，ゴールの幅よりも幅よりも内側にいれば，前方 120 度の範囲を探索する実装になっています．図 4.30 にこの場合分けの実例を示します.

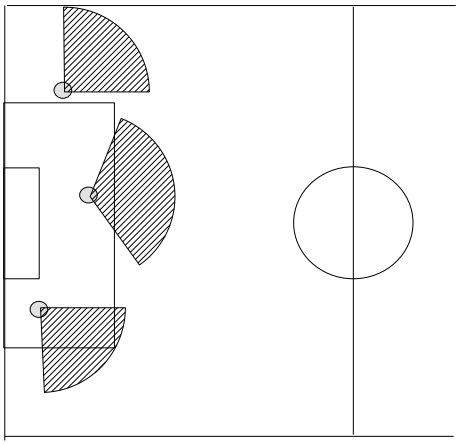


図 4.30 クリア方向の探索範囲

あまりに単純過ぎて不安を感じるかもしません．しかし，実用上はこれでほぼ充分です．クリアが必要な状況では周囲の確認も祿にできない状態であることも珍しくないため，余計なことはせず，なるべく単純で成功確率の高い範囲を探索した方が良い結果が得られやすいのです．

4.14.2 クリア方向の評価

二つめの条件を満たすための決定的なアルゴリズムはいまだ存在していません．librcsc では，厳密な計算は行わず，敵プレイヤの方向とクリア方向との角度差に基づいてスコアを付けるという単純な方法を採用しています．実装は以下のようになります．

```
double score = 1.0;
const Line2D angle_line( agent->world().self().pos(),
                         target_angle );
const AngleDeg target_left_angle = target_angle - 30.0;
const AngleDeg target_right_angle = target_angle + 30.0;
const PlayerPtrCont::const_iterator end
  = agent->world().getOpponentsFromSelf().end();
for ( PlayerPtrCont::const_iterator
        it = agent->world().getOpponentsFromSelf().begin();
      it != end;
      ++it ) {
  if ( (*it)->angleFromSelf().isWithin( target_left_angle,
                                        target_right_angle ) ) {
    Vector2D project_point = angle_line.projection( (*it)->pos() );
    double width = (*it)->pos().dist( project_point );
    double dist = agent->world().self().pos().dist( project_point );
    score *= width / dist;
  }
}
```

最初に score を 1 で初期化し，ボールの速度の方向と敵プレイヤの方向との角度差に応じて，この値を減衰させます．クリアでは，より前方にボールを運べた方が良いため，以下のような補正を更に行います．

```
  score *= ( 0.5
             * ( AngleDeg::sin_deg(1.5 * target_angle.abs() + 45.0 )
                 + 1.0) );
```

探索範囲内で 8 度単位で角度を変化させ，以上のようなスコア付けのループを行い，最終的に最もスコアの高かった方向をクリア方向として決定します．

4.14.3 改善すべき点

クリア方向の評価は非常に難しく，ルールで定義しきれるものではありません．解決方法としては，何らかの関数近似手法によって多次元入力の関数を獲得させる方法が良いのではないかと思います．ニューラルネットを利用するのが簡単で

しょうが，パスなどの場合と同様，計算量の問題が新たに発生するため，慎重に実装する必要があります．

Section 4.15
首振りによる視界方向の変更

プレイヤエージェントの首振り動作は，視界を特定の方向へ向けて情報収集するための重要な動作です．プレイヤエージェントは体の動きと独立して首を動かすことができますが，体の方向に対して相対に変化させることしかできません．よって，体の方向が変化すると，首の首の絶対方向も影響を受けます．首振り動作には体の動作による影響の考慮が常に必要です．

4.15.1 特定位置への首振り

特定位置への首振り動作の実装は以下のようになります．

```
AngleDeg target_rel_angle
  = agent->effector().queuedNextAngleFromBody( target_point );
target_rel_angle
  = ServerParam::i().normalizeNeckAngle( target_rel_angle.degree() );
agent->doTurnNeck( target_rel_angle - agent->world().self().neck() );
```

`queuedNextAngleFromBody()` によって，`target_point` への次サイクルの体からの相対方向が得られます．この関数は，**turn** コマンドによる体の方向の変化だけでなく，慣性による移動も考慮します．得られた値は，そのまま目標となる首の相対角度となります．ただし，プレイヤエージェントの首の角度の範囲は [-90,90] に制限されているので，その範囲を越えないように修正します．最後に，現在の首角度と目標首角度との差の分だけ首を回転させれば完了です．

4.15.2 ボールへの首振り

ボールへの首振りもほぼ同様です．ただし，現在のボール位置では無く，予測される次サイクルのボール位置へと首を向けなければなりません．実装は以下のようになります．

```
const Vector2D ball_next = agent->effector().queuedNextBallPos();
const AngleDeg ball_rel_angle_next
   = agent->effector().queuedNextAngleFromBody( ball_next );
ball_rel_angle_next
  = ServerParam::i().normalizeNeckAngle( target_rel_angle.degree() );
agent->doTurnNeck( ball_rel_angle_next - agent->world().self().neck() );
```

queuedNextBallPos() によって，次サイクルのボール位置座標が得られます．この関数は，慣性によるボールの移動だけでなく，プレイヤエージェント自身が実行した **kick** コマンドによる加速の影響も考慮します．ただし，他のプレイヤがボールをキックした場合は予測しようがないため全く考慮していません．後は，通常の特定位置への首振りと同じです．

4.15.3 首振りによる情報収集

プレイヤエージェントが得る環境からの情報は，そのほとんどが視覚センサからの情報です．よって，首振り動作によって視界の方向を変化させることで周囲の情報を積極的に入手しなければ，適切な意思決定を期待することは出来ません．

特に注目すべき対象が無い場合は，可能な限り平均的に周囲の情報を入手しておくことが望ましいでしょう．すなわち，最後に観測してからの経過時間が長い領域に対して優先的に視界を向けるようにするべきです．このような首振りによるフィールドスキャン動作は，Neck_ScanField クラスで実装されています．

フィールドスキャンを行うためには，WorldModel クラスに格納されている，方向の信頼性の情報を利用します．まずは，次サイクルのプレイヤエージェントの体の向きと視界の広さから，視界として取りうる範囲を求めます．実装は以下のようになります．

```
double next_view_width
  = agent->effector().queuedNextViewWidth().getWidth();
const Vector2D my_next = agent->effector().queuedNextMyPos();
const AngleDeg body_next = agent->effector().queuedNextMyBody();
const AngleDeg next_limit_min
  = body_next + ( ServerParam::i().minNeckAngle()
                  - (next_view_width * 0.5 + 1.0) );
const double next_neck_range
  = ( ( ServerParam::i().maxNeckAngle()
        - ServerParam::i().minNeckAngle() )
      + next_view_width - 2.0 );
```

WorldModel::getDirCount() によって，方向の信頼性の情報，すなわち，最後にその方向を見てからの経過サイクルを得ることが出来ます．この情報を利用し，next_limit_min から next_neck_range の範囲内で信頼性の低い方向を探します．このとき，一度に調べる広さは next_view_width です．

最初に，next_limit_min から next_view_width の範囲内で，WorldModel::getDirCount() の値を累積します．この値を保持しておき，WorldModel::DIR_STEP の分だけ角度を追加し，同様の計算を行います．つまり，next_limit_min+WorldModel::DIR_STEP から next_view_width の範囲内で，WorldModel::getDirCount() の値を累積します (図 4.31)．これを，next_neck_range の間だけ繰り返します．

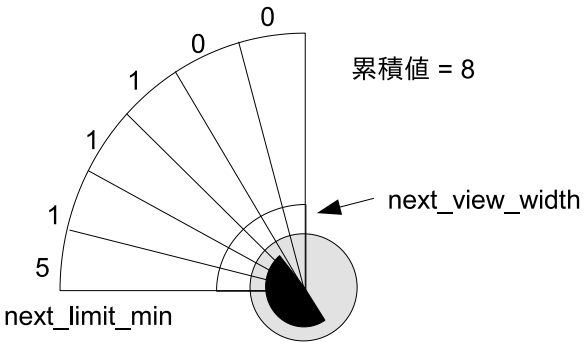


図 4.31 視界範囲での信頼性情報の累積

プレイヤエージェントがフィールドのライン近くにいる場合，首の方向によってはほとんどフィールドの外しか見ていないことがあります．これでは状況判断

に必要な情報がほとんど得られないなので，確認する方向の視界範囲がほとんどフィールドの外に出てしまう場合は，その方向はキャンセルされます．視界範囲がフィールドの外に出ているかどうかは，視界方向先 20 メートルの位置がフィールドの外に出ているかどうかで判断しています．

最終的に，刻み幅を `WorldModel::DIR_STEP` として，複数の視界方向とそれぞれの視界範囲での信頼性の累積情報が得られます．得られた複数の視界方向に対する信頼性の情報を比較し，最も信頼性の低い方向が選択されます．

第5章
コーチの利用

rcssserver 上では，プレイヤエージェント以外にコーチとトレーナの 2 種類のエージェントを使用できます．これらは，名前どおり，プレイヤエージェントに対して指示を与えることを目的としています．これらを上手く活用することで，チーム開発をより円滑に進めたり，より高度な戦略を実現することが可能になります．

また，ヘテロジニアスプレイヤの利用にはコーチエージェントが必要であるため，ヘテロジニアスプレイヤを使用するためだけにコーチエージェントを利用する例も少なくありません．

本章では，コーチエージェントとトレーナエージェントの仕様，そして，その活用方法に着いて解説します．

Section 5.1
コーチエージェント

以前には，コーチエージェントはオンラインコーチとも呼ばれていました．サッカーシミュレーションリーグの初期の頃は，コーチと言えばトレーナエージェントのことを意味していましたが，今ではコーチと言えばオンラインコーチを意味します．トレーナとは異なり，実際の競技でもコーチを使用することが許可されています．

コーチエージェントは，フィールド上の移動物体の完全な位置情報を得ることができます．そして，それらの情報を分析した結果から，自分が屬するチームのプレイヤに対してアドバイスを送信することが出来ます．コーチエージェントを利用することで，大局的な試合分析と動的な適応など，より抽象的で高度な意思決定が可能になります．

また，ヘテロジニアスプレイヤを実際の試合で利用するには，コーチエージェントを使わざるを得ません．ヘテロジニアスプレイヤのためだけにコーチエージェントを使うこともちろん可能です．

5.1.1 コーチのコマンド

能力的には，コーチエージェントはトレーナエージェントのサブセットです．コーチができることはトレーナも実行できます．コーチエージェントが利用できるコマンドのフォーマットについては，6.2 節で解説しているので，詳しくはそちらを参照してください．librcsc ではコーチエージェントが利用できるコマンドをクラスライブラリとして提供しています．各コマンドクラスの宣言と実装は，coach_command.{h,cpp} で見つけることができるので，実際の利用方法についてはそちらも参照してください．

5.1.2 CoachAgent クラスのインタフェース

コーチエージェントプログラムから rcssserver への情報の送信は，全て CoachAgent クラスを介して行われます．CoachAgent クラスには，トレーナエージェントのコマンドを送信するための関数が定義されています．以下にその一覧を示します．

CoachAgent メンバ
`bool doCheckBall();` checkball コマンドを送信する．本来はボールの状態を確認するために使用するが，rcssserver との接続確認の目的にも使用できる．
`bool doLook();` look コマンドを送信する．
`bool doTeamNames();` team_names コマンドを送信する．

`bool doEye( bool on );`
eye コマンドを送信する．on の値によって視覚情報を受信するかどうかを切り替える．
`bool doChangePlayerType( const int unum, const HeteroID type );`

change_player_type コマンドを送信し，指定したプレイヤのプレイヤタイプを変更する．type は，ヘテロジニアスプレイヤの Id.
bool doSay(const std::string & msg); say コマンドを実行し，メッセージを rcssserver へ送信する．ただし，他のサッカーエージェントとことなり，コーチエージェントの say コマンドではコミュニケーションのためのメッ セージは送信されず，コーチ言語によるアドバイスメッセージが配信される．

これらの関数を呼び出すと，コマンド送信も同時に実行されます．プレイヤエージェントの場合はコマンド送信タイミングの調整を行いますが，コーチエージェントの場合はコマンド送信タイミングを考慮しません．

Section 5.2
コーチ言語 (CLang)

コーチ言語とは，プレイヤに対して試合中にアドバイスを与えるためのメッセージフォーマットで，CLang とも呼ばれています．そのメッセージフォーマットはプロダクションルール形式[1)]であり，また，拡張性を備えた言語仕様となっています．例えば，フィールド上の特定領域を表現するメッセージに対して名前を付け，rcssserver 上でその名前を再利用することができます．この仕様によってメッセージの構造化が可能になり，メッセージの冗長化をさけることができます．

このように，コーチ言語は高機能な言語仕様を備えており，サッカーシミュレーションリーグのコーチ競技では，このコーチ言語を利用した動的な適応や試合分析に注目した競技が行われています．

上手く使えば研究としても面白いと思うのですが，残念ながら，本書執筆時点の librcsc ではコーチ言語に対応できていません．もし，コーチ言語に興味があるなら，rcssserver のソースパッケージにもコーチ言語のライブラリが含まれているので，それらを取り込んで利用するのが良いでしょう．

[1)] If-Then で記述できるルール

Section 5.3
ヘテロジニアスプレイヤの活用

ヘテロジニアスプレイヤとは，能力が少しずつ異なるプレイヤです．このヘテロジニアスプレイヤの能力パラメータは，`PlayerType` クラスで管理されます．ヘテロジニアスプレイヤは，デフォルトタイプを含めては 7 タイプ生成されます．残り 6 タイプの能力は，デフォルトタイプの能力を基準として，いくつかの能力パラメータの間でトレードオフを実行することで決定されます．例えば，加速能力の高いプレイヤタイプは，加速性能が高くなるほどスタミナの回復量が小さくなるというトレードオフが設定されています．rcssserver のソースを直接読むと，このトレードオフのルールについての理解が深まるでしょう．興味があれば，rcssserver のソースパッケージの `hetroplayer.{h,C}` を参照してみてください．`HetroPlayer` クラスのコンストラクタ内でパラメータ生成の実装を見つけることができます．

実際にヘテロジニアスプレイヤを利用する際には，プレイヤエージェントの役割に適した能力パラメータを持つプレイヤタイプを割り当てなければなりません．ここで，ヘテロジニアスプレイヤ利用に置ける制約を考慮しなければなりません．ヘテロジニアスプレイヤの利用に関しては，以下の制約があります．

- キーパにはデフォルトタイプしか使用できない．
- デフォルトタイプ以外のプレイヤタイプは，同時に 3 人までしか同一のものを割り当てられない．
- 試合開始前はプレイヤタイプの変更回数は無制限．
- 試合中は 3 回までしかプレイヤタイプを変更できない．

同じ能力パラメータのプレイヤタイプは 3 人までにしか割り当てられないため，チーム内の役割配分もこの仕様に多少影響を受けるかもしれません．

さて，プレイヤタイプごとに能力が異なるわけですが，どのような能力を持っていればより有利になるでしょうか？ほとんどの場合，足の早いタイプ，特に，加速性能が高いタイプは非常に有利です．能力パラメータの組合せによっては，足が早いにも関わらずスタミナの減りが鈍いという, 非常にパフォーマンスの高いプレイヤが生成されることもあります．このような能力を持つプレイヤタイプを適切に判別しなれば，試合では極めて不利になります．

一般的に，攻撃的な役割のプレイヤ，つまり，フォーワードに加速性能の高いプレイヤタイプを割り当てると，有利に試合を展開することができます．プレイヤタイプの割り当てにおいては，フォワードのプレイヤから先に割り当てを行うと良いでしょう．フォワードに割り当てるプレイヤタイプを選択するルールは以下のようになるでしょう．

- `PlayerType::realSpeedMax()` が最大か，最大に近い,.
- `PlayerType::staminaIncMax()` が一定値以上.
- 加速性能が高い.

あまり良い実装ではありませんが，サンプルチームには，このようなルールによってフォワードのためのプレイヤタイプを選択するコーチエージェントを含めています．参考にしてみてください．

Section 5.4
トレーナエージェント（オフラインコーチ）

トレーナはオフラインコーチとも呼ばれます．能力はコーチの上位セットとなっており，コーチができることはすべて実行可能です．それに加えて，試合やフィールド上の物体を自由に制御することが出来ます．そのため，実際の試合で使用することは禁じられており，あくまで試合前の訓練や実験のための利用に限られています．

トレーナエージェントを利用することで，実験の自動化が可能になります．多くの場合，機械学習を行うためのエピソードの繰り返し実行の制御，そして，プレイヤエージェントの行動内容の評価を自動化する目的に使用されます．

5.4.1 トレーナのコマンド

トレーナエージェントが利用できるコマンドのフォーマットについては，6.2 節で解説しているので，詳しくはそちらを参照してください．librcsc ではトレーナエージェントが利用できるコマンドをクラスライブラリとして提供しています．各コマンドクラスの宣言と実装は，`trainer_command.{h,cpp}` で見つけることができるので，実際の利用方法についてはそちらも参照してください．

5.4.2 TrainerAgent クラスのインタフェース

トレーナエージェントプログラムから rcssserver への情報の送信は，全て TrainerAgent クラスを介して行われます．TrainerAgent クラスには，トレーナエージェントのコマンドを送信するための関数が定義されています．以下にその一覧を示します．

TrainerAgent メンバ
bool doCheckBall(); checkball コマンドを送信する．
bool doLook(); look コマンドを送信する．
bool doTeamNames(); team_names コマンドを送信する．
bool doEye(bool on); eye コマンドを送信する．on の値によって視覚情報を受信するかどうかを切り替える．
bool doEar(bool on); ear コマンドを送信する．on の値によって視覚情報を受信するかどうかを切り替える．
bool doKickOff(); start コマンドを送信する．
bool doMoveBall(const Vector2D & pos, const Vector2D & vel); move コマンドを送信し，ボールの位置と速度を変化させる．
bool doMovePlayer(const std::string & teamname, const int unum, const Vector2D & pos); move コマンドを送信し，指定したプレイヤの位置を変化させる．
bool doRecover(); recover コマンドを送信し，全てのプレイヤのスタミナを初期値まで回復させる．
bool doChangeMode(const PlayMode mode); change_mode コマンドを送信し，プレイモードを変更する．
bool doChangePlayerType(const std::string & teamname, const int unum, const HeteroID type);

change_player_type コマンドを送信し，指定したプレイヤのプレイヤタイプを変更する．
bool doSay(const std::string & msg); say コマンドによってコミュニケーションメッセージを送信する．

これらの関数を呼び出すと，コマンド送信も同時に実行されます．プレイヤエージェントの場合はコマンド送信タイミングの調整を行いますが，トレーナエージェントの場合はコマンド送信タイミングを考慮しません．

5.4.3 実験時に利用するコマンド

学習実験などのためにあるエピソードを繰り返し実行する場合などによく利用するコマンドについて解説します．

TrainerKickOffCommand

TrainerAgent::doKickOff() によって送信されます．通常，試合を開始するには人間のオペレータが rcssmonitor の kick off ボタンを押さなければなりません．このコマンドによって，kick off ボタンを押さずとも試合を開始することが出来ます．

実験時には試合が自動的に始まって欲しいので，このコマンドを利用します．ただし，rcssserver-9.0.2 以降では，rcssserver 自体に自動モードが搭載されたため，このコマンドを利用する機会は少なくなりました．rcssserver の自動モードについては 5.5.1 節を参照してください．

TrainerChangeModeCommand

TrainerAgent::doChangeMode() によって送信され，現在のプレイモードを変更します．一つのエピソード終了後はプレイモードをフリーキックモードなどにしておき，エピソードを再開するときには ***play_on*** に変更する，という手順を取ることが多いでしょう．

TrainerMoveBallCommand

`TrainerAgent::doMoveBall()` によって送信され，ボールの位置と速度を変更します．一つのエピソードを開始する前に物体の初期配置を整えるために使用することが多いでしょう．ボールの位置だけでなく速度も変更できるため，インターセプト動作の調整を行う場合には特に重宝します．

TrainerMovePlayerCommand

`TrainerAgent::doMovePlayer()` によって送信され，プレイヤの位置と速度を変更します．使用目的は `TrainerMoveBallCommand` とほとんど同じです．複数のプレイヤを同時に移動させたい場合は，連続して `TrainerAgent::doMovePlayer()` を呼び出します．この呼び出しのたびにコマンドメッセージの送信が実行されます．複数のコマンドをまとめて送信することはできません．

TrainerRecoverCommand

`TrainerAgent::doReocver()` によって送信され，フィールド上に存在する全てのプレイヤのスタミナを完全に回復します．特定のプレイヤのみを対象とすることは出来ません．通常の ***stamina*** 値だけでなく，***reocver*** と ***effort*** も初期値の状態に戻ります．一つのエピソードを開始する前にスタミナを回復しておくことで，同じ条件での試行を可能にします．

TrainerSayCommand

`TrainerAgent::doSay()` によって送信され，フィールド上に存在する全てのプレイヤへ聴覚情報としてメッセージを送信します．トレーナエージェントからプレイヤエージェントへ実験の評価値を伝える目的に使用できます．トレーナエージェントに限っては rcssserver を介さずに直接メッセージを送受信しても問題は無いので，特に積極的に利用する必要はありません．他のメッセージ送受信と同じ通信経路を使用できる分，実装の手間がかからないという利点があります．

Section 5.5
実験の効率化

rcssserver を用いた実験では，トレーナエージェントを利用してシミュレーションの制御を行うことが多いでしょう．しかし，本説ではそれに加えて，実験を効率的に行うための rcssserver の設定方法を解説します．

5.5.1 自動モードの利用

オフラインの実験においては，何度も試合を自動的に繰り返す設定で実験を行う機会も多いでしょう．そのような場合，試合開始のキックオフをどのように実行するかが問題でした．以前は，`TrainerAgent::doKickOff()` による **start** コマンドの送信によってこの問題を解決していました．現在では，rcssserver 自体に自動モードという機能が実装されており，試合開始を完全に自動化することができるようになっています．もちろん，現在でもトレーナエージェントが試合開始を制御する方がよいことも多いのですが，rcssserver 組み込みの自動モードの方がはるかに手軽で簡単です．

この自動モードは，そもそも実験の自動化のために導入されたものではありませんでした．RoboCup2003 以降，試合進行を円滑にするために，最初に組んだスケジュールどおりに試合を自動進行させる運営方法が採用されるようになりました．rcssserver への自動モードは，競技運営のために導入されたものだったのです．とは言え，やはり，自動モードは繰り返し行う実験にも非常に役に立つものです．使用方法も簡単なので，是非活用しましょう．

自動モードを利用するには，`/.rcsssserver/server.conf` 内，または，コマンドライン引数によって ***auto_mode*** を有効にします．すると，rcssserver は プレイヤが接続するまで ***connect_wait*** サイクル待機し，キックオフ前に ***kick_off_wait*** サイクル待機します．***nr_normal_halfs*** ハーフ終了後，rcssserver はプロセスを終了する前に ***game_over_wait*** サイクル待機します．延長戦は行われません．自動モードが有効な場合，***team_l_start*** と ***team_r_start*** によって，チームの起動コマンドを指定することもできます．

まとめると，自動モードの実行においては，以下のサーバパラメータが利用されます．

パラメータ名	説明
auto_mode	自動モードを使用するかいなかを決定する.
connect_wait	プレイヤが接続するまでの最大待機サイクル数. この時間を経過してもプレイヤの接続が無い場合は, 無視して試合が開始される.
kick_off_wait	プレイヤが接続されてから試合開始までの待機サイクル数. この時間が経過すると, 試合が自動的に開始される.
game_over_wait	試合終了から rcssserver によるプロセス終了までの待機サイクル数. この時間が経過すると, rcssserver はチーム起動スクリプトを実行したプロセスへ SIGINT を送り, 自分自身のプロセスも終了させる.
team_l_start	左チームの起動コマンドのパスを指定できる. このパラメータの文字列が空でなければ, rcssserver はその文字列を左チームを起動するためのコマンドとして実行する. コマンド文字列には空白も使用できる.
team_r_start	対象が右チームである以外は ***team_l_start*** と同じ目的で使用する.

これらのパラメータは `/.rcssserver/server.conf` を編集することでも変更できますが, rcssserver のコマンドライン引数として与える方が簡単です. その場合, ***team_l_start*** と ***team_r_start*** の指定には注意が必要です. まず, チームの起動スクリプトを指定する場合, そのスクリプトファイルへの絶対パスを指定しなければなりません. そして, 起動スクリプトを使用するのであれば, そのスクリプトが rcssserver から送られる SIGINT 割り込みを受け付けられなければなりません.

rcssserver 起動時のコマンドラインオプションの例を以下に示します.

```
$ rcssserver server::auto_mode = 1 \
    server::team_l_start = '/home/robocup/team1/start.sh' \
    server::team_r_start = '/home/robocup/team2/start.sh' \
```

この例では, 各チームは `start.sh` という起動スクリプトで起動すると想定しています. しかし, 実際にはスクリプトへのコマンドライン引数を与えることも必要になります. すると, 空白の取り扱いなどがややわかりにくくなります. そのような場合は, rcssserver 起動用のスクリプトを新たに作り, 以下のように記述

しておくと若干管理が楽になります.

```
#!/bin/sh
START_L="/tmp/start_l.$$"
echo "$/home/robocup/team1/start.sh server1" > ${START_L}
chmod +x $START_L
TEAM_L_START=="server::team_l_start = \"${START_L}\""
START_R="/tmp/start_r.$$"
echo "$/home/robocup/team2/start.sh server1" > ${START_L}
chmod +x $START_R
TEAM_R_START="server::team_r_start = \"${START_R}\""

rcssserver $TEAM_L_START $TEAM_R_START &
```

最後に，チームの起動スクリプトが SIGINT 割り込みを受け付ける方法を説明します．これは以下のようなシェルスクリプトを用意することで実現できます.

```
#!/bin/sh
player=/home/robocup/team1/player
trap kill_team INT
kill_team()
{
    echo "Killing Team"
    killall ‘basename $player‘
    exit 0
}
$options=...
$player $options &
$player $options &
$player $options &
...
$player $options &
wait
```

本書で提供する librcsc によって動作するクライアントプログラムは，rcssserver との通信が途絶えると自動終了するように作られています．しかし，より確実に

プロセスを終了させるために，このようなスクリプトを用意することが推奨されています．

5.5.2 同期モードの利用

実験で試行を繰り返す場合，特に遺伝的アルゴリズムなどの膨大な回数の試行が必要な実験においては，高速なシミュレーションが実行できると著しく効率が上がります．実機の制約の無いシミュレーションなのですから，実時間の制約も取り払えて然るべきです．rcssserver には同期モードという機能が実装されており，このモードを利用することで高速シミュレーションが可能になります．

rcssserver における同期モードでは，サーバとクライアントとの通信が 00%成功するものと仮定し，センサ情報とコマンドの送受信の間の待ち時間を無くしてしまいます．これによって，見かけ上のシミュレーション時間を圧縮します．計算資源を最大限に使いきってシミュレーションを実行するため，通常であれば前後半で 10 分必要な試合を数十秒程度で完了させることができます．その代わり，人間の目では追い付かないほどに高速になってしまうため，rcssmonitor でリアルタイムに観戦することは難しくなります．

同期モードを利用するには，サーバパラメータの ***synch_mode*** を有効にします．これは， `/.rcssserver/server.conf` を編集するか，rcssserver のコマンドライン引数で指定することで設定できます．`server.conf` を編集する方が楽でよいでしょう．

同期モードを利用する場合，クライアントプログラム側が対応していなければならない点に注意してください．librcsc とサンプルチームは同期モードに完全に対応しているので，安心して使用してください．もし，同期モードに対応できていないチームを同期モードで動作する rcssserver に接続してしまうと，rcssserver もクライアントプログラム側も正しく動作することができず，シミュレーションが全く進まなくなってしまいます．これは，プレイヤエージェントだけでなく，コーチエージェント，そして，トレーナエージェントのプログラムでも同様です．

同期モードで動いている rcssserver は，クライアントプログラムに **(think)** というメッセージを送信します．このメッセージは，クライアントプログラムがそれぞれのコマンドを rcssserver へと送信するタイミングであることを通知しています．**(think)** を受信したクライアントプログラムは，意思決定に関わるコマンドを送信するか否かに関わらず，**(done)** というコマンドを送信しなければなりません．rcssserver は，接続中の全てのクライアントプログラムから **(done)** が帰ってくるのを一定時間待機します．そのため，**(done)** を適切に送信できないクライ

アントプログラムが存在すると，rcssserver は正しく動作できなくなってしまいます．

本書執筆時点で同期モードに対応できているチーム，ライブラリはいくつか存在します．特に，UvA Trilearn とそのライブラリは安定して使えるので，同期モードでの対戦相手として使い易いでしょう．

5.5.3 Keepaway モードの利用

Keepaway モードとは，フィールド中央に固定サイズの矩形領域を設定し，その領域内でボールを奪い合う Keepaway という実験を行うためのモードです [12]．2 対 2 や 2 対 3 程度の小人数でのシミュレーションを実行することで，通常のサッカーの試合よりも小さい問題設定としています．状況が限定的であるため，さまざまな学習手法を実時間の範囲で適用することが可能になります．

Keepaway モードでは，ボールを守る Keeper と，ボールを奪う Taker とに分かれ，Taker 側がボールを奪うか，ボールが矩形領域の外に出るまでを 1 エピソードとしてシミュレーションを繰り返すことができます．この判定は rcssserver の審判機能として組み込まれているため，わざわざトレーナエージェントを作る必要はありません．

同期モードを利用するには，サーバパラメータの ***keepaway*** を有効にします．これは，`/.rcssserver/server.conf` を編集するか，rcssserver のコマンドライン引数で指定することで設定できます．`server.conf` を編集する方が楽でよいでしょう．同時に，rcssmonitor の設定も変更しておく必要があります．`/.rcssmonitor.conf` を編集し，`keepaway` の値を 1 に設定します．この状態で rcssmonitor を起動すると，図 5.1 のような画面が現れます．このように，通常とは異なり，フィールド中央に矩形が表示されます．

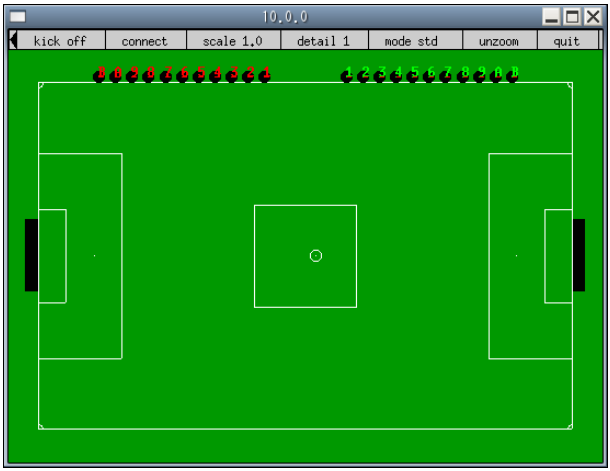


図　5.1　Keepaway 領域を表示した rcssmonitor

第6章

librcsc詳説

本章では，librcsc をより深く解説し，また関連する rcssserver の仕様についても説明します．

プレイヤやコーチとしての意思決定を行うだけでは，サッカーエージェントが rcssserver と連携して動作することはできません．サッカーエージェントには rcssserver と通信するクライアント機能が必須となります．クライアントとして動作するためには，適切なプロトコルと適切なタイミングで rcssserver とのメッセージの送受信ができなければなりません．更に，rcssserver から送られてきたメッセージを基に，サッカーエージェント内部で仮想フィールドの状態を再構築しなければなりません．これらが正しく行われなければ，プレイヤエージェントは自分が今フィールド上のどこにいるのかすら知ることができないのです．

これらの問題はマルチエージェントシステムという観点からはあまり本質的ではありません．実際，作っていてあまり面白い部分でもありません．しかしながら，実世界のような不確実性の伴う環境に対する頑健性を考慮する上では，適度に擬似的な問題設定となっているのではないかと思います．

Section 6.1
rcssserver との通信

試合実行中，サッカーエージェントは rcssserver との通信を繰り返しています．この通信でのメッセージのやりとりによってのみ，サッカーエージェントは rcssserver 内部での仮想フィールドの状態を知ることができます．そのため，サッカーエージェントが正しく動作するためには，通信の正確性やコマンド送信のタイミングが非常に重要になります．

6.1.1 UDP/IP 通信

rcssserver とサッカーエージェントとの間の通信は UDP/IP で行われます．UDP という用語自体をあまり聞き慣れないかもしれませんが，一般的なインターネット通信に利用されている通信プロトコルで，正しくは User Datagram Protocol と言います．IP とは，Internet Protocol の略で，IP によって世界規模で相互接続されたコンピュータネットワークがインターネットやイントラネットと呼ばれています．よって，IP によってネットワーク接続され，IP アドレスを割り当てられたコンピュータの間であれば，rcssserver とサッカーエージェントプログラムの通信が可能となり，分散して動作させることができます．

6.1.2 UDP の問題

UDP には，転送速度は高いが信頼性が低いという特徴があります．そのため，rcssserver とサッカーエージェントとのメッセージ送受信の信頼性もそれほど高くありません．例えば，エージェントが送信したコマンドが届かない，rcssserver からのセンサ情報が届かないなどの障害が発生する可能性があり，通信が著しく遅延してもそれを確認する手段がありません．

この UDP を利用することによって rcssserver とサッカーエージェントとの非同期通信が実現されています．rcssserver からサッカーエージェントへのセンサ情報送信と，サッカーエージェントから rcssserver へのコマンド送信とは完全に独立して行われるため，サッカーエージェントからのコマンド送信が遅れたとしても，rcssserver はそれをまったく考慮せずにシミュレーションを進めてしまいます．互いのメッセージの不達などお構い無しでそれぞれが完全に独立して動作しているのです．

一台のコンピュータ上では正しく動作していたがネットワーク上で分散動作させると挙動がおかしくなった，といったトラブルは良くある話です．サッカーエージェントの開発においては，通信障害の可能性を常に考慮していなければならないのです．

6.1.3 UDPSocket クラス

librcsc では，UDP によるメッセージの送受信に `sendto()` や `recvfrom()` などのシステムコール関数を使用しています．これらは POSIX(Portable Operating System Interface for UNIX) と呼ばれる UNIX システムが守るべき標準インタ

フェースとされているので，Linux や FreeBSD，Cygwin などで同様に使用できるはずです．

librcsc では，これらのシステムコール処理をユーザから隠蔽する UDPSocket という通信クラスライブラリを用意しています．このクラスを用いることで通信機能の実装と利用の手間を省略し，rcssserver とのメッセージ送受信を手軽に行うことができます．UDPSocket クラスは以下の public なインタフェースを提供しています．

UDPSocket メンバ
UDPSocket(const char * hostname, const int port); コンストラクタ．クライアント用ソケットを作成する．hostname:接続先リモートホスト名または IP アドレス．port:接続先ポート番号．
UDPSocket(const int port); コンストラクタ．サーバ用ソケットを作成する．hostname:接続先リモートホスト名または IP アドレス．port:接続先ポート番号．
bool isOpen() const; ソケットが使用できる状態になっているかどうかを調べる．返り値 ： 使用できるなら true，できないなら false を返す．
int send(const char * msg, const int len); 接続先ポートへメッセージを送信する．msg:送信する文字配列へのポインタ．len:送信する文字列の長さ．返り値:送信に成功した場合，送信した文字列の長さを返す．失敗した場合，-1 を返す．
int receive(char * buf, std::size_t len); 接続先ポートからのメッセージを受信する．buf:メッセージを格納する文字配列へのポインタ．len:buf に割り当てられた文字配列の長さ．返り値:受信に成功した場合，受信した文字列の長さを返す．ソケット上に利用できるデータが無い場合，0 を返す．エラーが発生した場合，-1 を返す．

UDPSocket を作成して接続の準備が整えば，後は UDPSocket のメンバ関数である send() と receive() を使うだけです．実際の使用は以下のようになります．

```
UDPSocket socket( "192.168.1.100", 6000 ); // ソケット作成
std::string command( "(init TEAM)" ); // 送信コマンドメッセージ
// コマンド送信
soket.send( command.c_str(), command.length() + 1 );
char buffer[8192];// 受信用バッファ
// キューにたまっている受信メッセージが無くなるまでループ
while ( socket.receive( buffer, 8192 ) > 0 ) {
   analyzeMessage( buffer );
}
```

send() の第二引数で文字列長に+1 しています．これは C スタイル文字列のヌル文字 ('\0') まで含めて送信するためです．ヌル文字終端されていないメッセージが届くと，rcssserver は “(warning message_not_null_terminated)” という警告メッセージを返信してきます．

この UDPSocket クラスはサッカーエージェントに特化した使用を想定しています．利用時の混乱を無くすためにクラスのインタフェースを必要最小限にしており，ライブラリとしての汎用性は乏しくなっています．UDP 通信を行う他のアプリケーション開発での利用には向かないので，注意してください．本書では基本的な通信技術に関してはこれ以上解説しません．詳しくは [20] などの解説書を参照してください．

6.1.4 メッセージ受信の検出

UDPSocket の receive() 関数を呼び出す無限ループを実行するだけでは CPU を 100%使い続け，計算資源を無駄遣いしてしまいます．よって，必要無いときはプログラムをスリープさせておき，rcssserver からメッセージが届いたときにのみ receive() メソッドを呼び出すというイベントドリブン方式の採用が望ましいと言えます．これは，select() という POSIX 準拠のシステムコール関数によって解決できます．select() を使うことでソケットへの入力（メッセージ受信）が発生するまではプログラムをスリープさせ，計算資源を他のプロセスへ解放することができます．

他にも，シグナル処理を利用してソケットへの入出力を検出する方法があります．どちらの方法でもほぼ同じことが実現できますが，シンプルできれいな実装ができるので，librcsc では select() を利用する方法を採用しています．

メッセージ受信の検出とプログラムスリープだけでは，サッカーエージェントとして頑健に動作するにはまだ充分ではありません．rcssserver からのメッセージが欠落もしくは遅延していると判断されれば，それまでに得られた情報による推測に基づいて，意思決定とコマンド送信を行わなければなりません．そもそも，rcssserver からプレイヤエージェントへ送られる視覚情報の送信タイミングと rcssserver 内部のシミュレーションサイクルとは非同期であるため，プレイヤエージェントは rcssserver からのメッセージとは独立して意思決定のタイミングを取れるように設計されていなければならないのです．

この問題を解決するためには，定期的に意思決定の機会をチェックする機構が必要となります．メッセージ受信機会に反応するイベントドリブンに対して，このように定期的なチェックを行う方法をポーリング方式と言います．これもまた，`select()` 関数で実現できます．`select()` はソケットへの入力を待機するだけでなく，一定時間何の入力もなければタイムアウトさせることもできます．

`select()` を使うと，プログラムのメインループを以下のように書けます．

```
UDPSocket socket( "192.168.1.100", 6000 );
const int INTERVAL_MSEC = 10; // タイムアウトまでの待ち時間 [ミリ秒]
struct timeval interval; // タイムアウトまでの待ち時間保持用
// select() を使うためのおまじない
fd_set read_fds;
fd_set read_fds_back;
FD_ZERO( &read_fds );
FD_SET( socket.fd(), &read_fds );
read_fds_back = read_fds;

while ( isServerAlive() ) // サーバが有効と判断できる間はループ
{
  read_fds = read_fds_back;
  interval.tv_sec = INTERVAL_MSEC / 1000;
  interval.tv_usec = ( INTERVAL_MSEC % 1000 ) * 1000;

  // interval の有効期限つきで socket への入力を待機
  int ret = ::select( socket.fd() + 1, &read_fds,
                      static_cast< fd_set * >( 0 ),
                      static_cast< fd_set * >( 0 ),
                      &interval );
  if ( ret < 0 ) { // エラー
      perror( "select" );
      break;
  } else if ( ret == 0 ) { // 受信無し．タイムアウト
      handleTimeout(); // タイムアウト時の処理
  } else { // メッセージ受信
      handleMessage(); // メッセージ受信後の処理
  }
}
```

6.1.5 BasicClient クラス

librcsc では，プログラムメインループ処理をラップした BasicClient というクラスを提供しています．この BasicClient が全てのサッカーエージェントの基

底クラスになっています．PlayerAgent，CoachAgent，TrainerAgent は，いずれも BasicClient からの派生クラスです．BasicClient は以下の public なインタフェースを持っています．

BasicClient メンバ
bool init(const int argc, const char * const * argv); コマンドラインオプションで初期化する．argc:main 関数の argc を渡す．argv:main 関数の argv を渡す．返り値:初期化が成功すれば true，失敗すれば false を返す．
void run(); プログラムメインループ．

init() によって，プログラム起動時のコマンドラインオプションを使用し，サッカーエージェントを初期化できます．init() は内部で doInit() メンバ関数を呼び出します．doInit() は純粋仮想関数として宣言されており，PlayerAgent，CoachAgent，TrainerAgent などの各派生クラス内で実際の定義がなされています．init() はエージェントの内部変数の初期化を行うのみで，rcssserver への接続は行いません．

run() 内部では，最初に rcssserver との接続を行い，続いてメッセージ待機のループを開始します．このループは，rcssserver が実行中と推定される間は無限に実行されます．メインループ内では handndleMessage() と handleTimeout() を呼び出します．rcssserver との接続が断たれたと判断されるとループを終了し，handleExit() 関数を呼び出します．これらも純粋仮想関数として宣言されており，各派生クラスで実際の定義がなされています．

プログラムの main 関数では init() と run() の 2 つの関数だけを呼び出します．以下のように，init() による初期化を行い，続いて run() を呼び出してループを実行することになります（実際には，Ctrl-C や kill コマンドなどによる強制終了への対応も必要です）．

```
int main( int argc, char ** argv ) {
  PlayerAgent agent;
  agent.init( argc, argv );
  agent.run();
  return 0;
}
```

Section 6.2
コマンド

サッカーエージェントは，意思決定に基づく行動やクライアント制御情報を rcssserver へ伝えるために，規定のコマンドを使用します．このコマンドは人間が読んでも理解できるようなテキスト文字列のフォーマットとなっています．この節では，主にプレイヤエージェントが使用するコマンドについて説明します．

6.2.1 コマンドクラス

コマンド文字列のフォーマットを覚えておくのは大変です．そこで，librcsc ではコマンド文字列生成のためのクラスライブラリを用意しています．全てのコマンドクラスは以下のインタフェースを持っています．

コマンドクラスメンバ
`std::ostream & toStr( std::ostream & os ) const;` ストリームにコマンド文字列を出力する．os:出力先ストリームへの参照．返り値:出力先ストリームへの参照.

例えば，プレイヤエージェントが rcssserver へ接続するためには **init** というコマンドを送信しますが，この **init** コマンド文字列は PlayerInitCommand というクラスのオブジェクトで生成できます．コマンドクラスは以下のように使用します．

```
std::string teamname( "TeamName" );
double version = 9.0;
bool goalie = true;
PlayerInitCommand com( teamname, version, goalie );

std::ostringstream os;
com.toStr( os ); // "(init TeamName (version 9) (goalie))"を出力
sendMessage( os.str().c_str() );
```

librcsc では，rcssserver が受け付ける全てのコマンドに対応するクラスを用意しています．全てのコマンドクラスは，PlayerCommand，CoachCommand，TrainerCommand という基底クラスからの派生クラスです．これらは，player_command.h，coach_command.h，trainer_command.h で宣言されています．

6.2.2 プレイヤの接続コマンド

rcssserver にサッカーエージェントとして接続するには，UDPSocket による通信経路を利用して初期化コマンドを送信しなければなりません．プレイヤエージェントの場合は，初期化コマンドとして **init** と **reconnect** を使用します．また逆に，サッカーエージェントの動作を終了する際には切断コマンドを送信して rcssserver へ自らの終了を伝えることもできます．切断断のコマンドとして bye が使えます．

librcsc では，初期化と切断のコマンドは PlayerAgent などのクラスが自動で行うようになっているので，普段はあまり意識する必要はありません．しかし，最初に行われていることを知っておくことはトラブル対応の助けとなるでしょう．

以下，接続に関するコマンドクラスのコンストラクタを示し，生成されるコマンド文字列のフォーマットについて説明します．

PlayerInitCommand

```
PlayerInitCommand( const std::string & team_name,
                   const double & version = 3.0,
                   const bool goalie = false );
```

生成されるコマンド文字列:
(init *team_name* [**(version** *version*)] [**(goalie)**])

team_name には，英数字，ハイフン (-) とアンダーバー (_) が使えます．空白は使えません．16 文字以上のチーム名はログファイルに正しく記録できないので注意してください．*version* はクライアントが受け付けるプロトコルバージョンを指定します．指定しなかった場合，自動的にプロトコルバージョン 3 のクライアントして rcssserver に扱われます．古いバージョンを指定しても良いことは何もありません．librcsc は最新のプロトコルをサポートしているので，最新のバージョンを必ず指定してください．本書執筆時では，プロトコルバージョンは 9 が最新です．最後の (goalie) は，キーパとして動作させたい場合に使用し，コンス

トラクタの第 3 引数が *true* であれば生成されますキーパはボールを手でキャッチできる唯一のプレイヤですが，1 チームに 1 体しか使用できません．**init** コマンドが成功すると，rcssserver は以下のメッセージを返信してきます．

(**init** { **l** | **r** } *unum PlayModeString*)

'l' または'r' はチームのサイドを示す文字，*unum* はプレイヤに割り当てられた背番号，*PlayModeString* は現在のプレイモードを示す文字列です．

更に，rcssserver の設定パラメータとして以下のメッセージが送信されてきます．

(**server_param** *Parameters* ...)

(**player_param** *Parameters* ...)

(**player_type** *Id Parameters* ...)

初期化に問題があれば，以下のエラーメッセージが返されます．

(**error no_more_team_or_player_or_goalie**)

rcsserver から返信されてくるメッセージの種類とその解析については 6.3 節で解説しています．

PlayerReconnectCommand

```
PlayerReconnectCommand( const std::string & team_name,
                        const int unum );
```
生成されるコマンド文字列:
(**reconnect** *team_name unum*)

reconnect はプレイヤエージェントの再接続を行うためのコマンドです．何らかのエラーによって通信が途切れてしまった場合や，異なる設定でプレイヤエージェントを起動し直したい場合などに，rcssserver を再起動すること無くプレイヤを再接続するために利用できます．*team_name* は **init** と同様，自分のチーム名の文字列です．*unum* で再接続対象となるプレイヤの背番号を指定します．**reconnect** コマンドが成功すると，**init** コマンドの場合と同様に，rcssserver が以下のメッセージを返信してきます．

(**reconnect** { **l** | **r** } *PlayModeString*)

PlayerByeCommand

```
PlayerByeCommand();
```
生成されるコマンド文字列:
(bye)

rcssserver との接続を切断し，クライアントとしての動作を終了するときに使用します．rcssserver がこのコマンドを受け付けると，コマンドを送信したプレイヤエージェントにメッセージが届かなくなり，プレイヤエージェントからのコマンドも受け付けられなくなります．更に，rcssmonitor 上でもプレイヤがフィールドから退場させられます．

6.2.3 プレイヤの行動コマンド

以下，プレイヤエージェントの行動コマンドのコンストラクタを示し，生成されるコマンド文字列のフォーマットについて説明します．コマンドクラス名の後に Body と付いているものは体を動かすコマンド，Support と付いているものは補助行動コマンドを意味します．各コマンドの作用や使用制限については，4.2 を参照してください．

PlayerKickCommand (Body)

```
PlayerKickCommand( const double & power,
                   const double & rel_dir );
```
生成されるコマンド文字列:
(kick *power rel_dir***)**

rel_dir は −180 から 180 までの実数の度数で，プレイヤエージェントの体の向きに相対な角度を入力とします．*power* は −100 から 100 の実数ですが，任意の方向への加速が可能であるため，負の値を使う必要はありません．

PlayerDashCommand (Body)

```
PlayerDashCommand( const double & power );
```
生成されるコマンド文字列:
(dash *power***)**

power は −100 から 100 までの実数です.

PlayerTurnCommand (Body)

```
PlayerTurnCommand( const double & moment );
```
生成されるコマンド文字列:
(turn *moment***)**

moment は −180 から 180 までの実数の度数です.

PlayerTackleCommand (Body)

```
PlayerTackleCommand( const double & power );
```
生成されるコマンド文字列:
(tackle *power***)**

power は −100 から 100 までの実数です.

PlayerCatchCommand (Body)

```
PlayerCatchCommand( const double & rel_dir );
```
生成されるコマンド文字列:
(catch *rel_dir***)**

キーパのための特別なコマンドです．*rel_dir* は −180 から 180 までの実数の度数で，プレイヤエージェントの体の向きに相対な角度を入力とします．

PlayerMoveCommand (Body)

```
PlayerMoveCommand( const double & x, const double & y );
```
生成されるコマンド文字列:
(move x y**)**

x と y にはフィールド上の任意の位置を指定できますが，キックオフ前とゴール直後には自陣エリア内のみで移動し，キャッチ後のキーパは自陣のペナルティエリア内でのみ移動すべきです．

PlayerTurnNeckCommand (Support)

```
PlayerTurnNeckCommand( const double & moment );
```
生成されるコマンド文字列:
(turn_neck *moment*)

moment は −180 から 180 までの実数の度数です．ただし，プレイヤの首の角度は −90 から 90 に制限されているため，それ以上の角度へ回転させようとしても無効になります．**turn_neck** コマンドは 1 サイクルに 1 回しか実行できませんが，他のコマンドとの同時実行は可能です．

PlayerChangeViewCommand (Support)

```
PlayerChangeViewCommand( const ViewWidth & width,
                         const ViewQuality & quality )
```
生成されるコマンド文字列:
(change_view *width quality*)

width は **narrow**，**normal**，**wide** のいずれか，*quality* は **high** または **low** のいずれかです．

コンストラクタの引数で指定する ViewWidth と ViewQuality は，プレイヤエージェントの視界モードの取り扱いを容易にするために librcsc で定義されているクラスです．

このコマンドはいつでも実行でき，即座に効果が現れます．

PlayerSayCommand (Support)

```
PlayerSayCommand( const char * message,
                  const double & version );
```
生成されるコマンド文字列:
(**say** "*message*") または (**say** *message*)

message に使用可能な文字は英数字に記号を含めた 73 種類，文字列長は 10 文字までと rcssserver 側で制限されています．このコマンドは他の動作コマンドと同時に使用でき，またいつでも使用できますが，同一サイクル内に複数回の **say** コマンドを実行すると以前のメッセージが上書きされてしまいます．

コンストラクタの *version* 引数でプロトコルバージョンを指定できます．バージョン 8 以降のプロトコルでは *message* が二重引用符 (") で囲われます．コマンド送信やメッセージ配信がより安全になるので，バージョン 8 以降の使用を推奨します．

PlayerPointtoCommand (Support)

```
PlayerPointtoCommand( const double & dist,
                      const double & rel_dir )
```
生成されるコマンド文字列:
(**pointto** *dist rel_dir*)

dist は任意の実数，*rel_dir* は −180 から 180 までの実数の度数で，プレイヤエージェントの体の方向に相対な角度を入力とします．

```
PlayerPointtoCommand();
  生成されるコマンド文字列:
  (pointto off)
```

“off” が指定された場合は，現在継続中の指さし動作を止めます.

PlayerAttentiontoCommand (Support)

```
PlayerAttentiontoCommand( const SideType side,
                          const int unum )
  生成されるコマンド文字列:
  (attentionto side unum)
```

side は”our” もしくは”opp” のいずれか[1)]，*unum* は背番号です.

コンストラクタの引数で指定する OurSide は，PlayerAttentiontoCommand 内部で定義されている列挙型で，OUR または OPP という値を取ります.

```
PlayerAttentiontoCommand()
  生成されるコマンド文字列:
  (attentionto off)
```

現在行っている指さし動作を止める “off” モードのコマンド文字列を生成します.

6.2.4 プレイヤのその他のコマンド

プレイヤが実行可能なコマンドが他にもいくつかあります．これらは動作制御では無く，プレイヤエージェントの状態確認や rcssserver 内部での設定変更を目的とするコマンドです．チーム開発時にこれらのコマンドを使用することはほと

[1)] 実際にはもっと多くの形式が利用可能ですが，この二種類で充分です.

んど無いため，各コマンドクラスのコンストラクタは省略し，生成されるコマンド文字列のみを説明します．

PlayerEarCommand

(ear (on|off our|opp partial|complete))

プレイヤエージェントの聴覚センサの状態を変更します．他のプレイヤからの **say** メッセージを受信するかどうか，完全な **say** メッセージのみを受信するかどうかを選択できます．味方とのコミュニケーションを意図的にカットしたい場合などに使用できます．通常は librcsc の PlayerAgent が自動で適切なモードを設定してくれるので，チーム開発者がこのコマンドを実行する必要はありません．

PlayerSenseBodyCommand

(sense_body)

sense_body メッセージを要求します．**sense_body** は毎サイクル自動的に送信されてくるので，使用する必要はありません．

PlayerScoreCommand

(score)

現在の得失点情報を要求します．ゴール直後に総得点情報が審判から自動的に送信されるので，実行する必要はありません．

PlayerCLangCommand

(**clang** (**ver** *min max*))

サポートするコーチ言語のバージョン範囲を *min* と *max* で設定します．コーチ言語を使用しない場合は実行する必要はありません．

PlayerCompressionCommand

(**compression** *level*)

rcssserver と送受信されるメッセージを gzip アルゴリズムを使って圧縮転送します．*level* で 0 から 9 の圧縮レベルを選択できます．0 は無圧縮，すなわち通常の送受信を意味します．圧縮メッセージはコーチ言語のような非常に長いメッセージが要求される場面での使用が想定されています．プレイヤエージェントが使用するメリットは無いでしょう．

PlayerDoneCommand

(**done**)

rcssserver が ***synch_mode*** を有効にして動作している場合，プレイヤエージェントが意思決定を行うべきタイミングになると，rcssserver からプレイヤエージェントへ”(think)” というメッセージが送信されてきます．そのとき，プレイヤエージェントは通常の動作コマンドに加えて，この **done** コマンドを最後に追加して送信しなければなりません．librcsc の PlayerAgent は ***synch_mode*** を検出し，自動で **done** コマンドを送信してくれます．

6.2.5 コーチのコマンド

試合中，コーチが実行するコマンドは非常に限られており，フィールド上に影響を与えるのは，プレイヤタイプ変更 (**change_player_type** とコーチ言語の発話 (**say**) しかありません.. ここでは，各コマンドクラスのコンストラクタは省略し，生成されるコマンド文字列のみを説明します.

CoachInitCommand

(**init** $TeamName$ [$CoachName$] $Version$)

$TeamName$ はプレイヤエージェントが使用したチーム名文字列と同じ文字列を使用します．$CoachName$ はコーチエージェント自身の名前を指定し，この名前をログファイル名に残すことができます．$Version$ はクライアントが受け付けるプロトコルバージョンを指定します.

CoachByeCommand

(**bye**)

rcssserver との接続を切断します．rcssserver がこのコマンドを受け付けると，コマンドを送信したコーチエージェントにメッセージが届かなくなり，コーチエージェントからのコマンドも受け付けられなくなります.

CoachCheckBallCommand

(**check_ball**)

ボールの状態を確認するためのコマンドです．通常，使用する必要はありません.

CoachLookCommand

(look)

see_global とは別に視覚センサ情報を要求するコマンドです．コマンドが成功すると，**(ok look ...)** というメッセージとともに視覚センサ情報が返信されてきます．

CoachTeamNamesCommand

(team_names)

量チームのチーム名文字列を要求するコマンドです．コマンドが成功すると，チーム名文字列を含んだメッセージが返信されてきます．

CoachEyeCommand

(eye { on | off })

視覚センサ情報を受信するかどうかのモードを変更します．デフォルトではコーチエージェントは視覚センサ情報を得ることができないため，**on** モードのコマンドを送信する必要があります．

CoachChangePlayerTypeCommand

(change_player_type $Unum$ $Type$**)**

背番号が $Unum$ のプレイヤタイプを $Type$ 番に変更します

CoachSayCommand

(**say** *message*)

アドバイスメッセージをプレイヤに向けて発信します．*message* はコーチ言語のフォーマットを満たしていなければなりません．

CoachTeamGraphicCommand

(**team_graphic** (*X Y* "*XPMLine*" ... "*XPMLine*"))

タイル分割された XPM 形式のチームアイコンを送信します．rcsserver 経由でモニタプログラムがこのチームアイコンを利用することができます．

CoachCompressionCommand

(**compression** *level*)

rcssserver と送受信されるメッセージを gzip アルゴリズムを使って圧縮転送します．使用方法はプレイヤエージェントと同じです．コーチ言語の文字列が長くなり，パケットのバッファサイズを越えそうなときにを使用します．

CoachDoneCommand

(**done**)

rcssserver が ***synch_mode*** を有効にして動作している場合に使用します．使用方法はプレイヤエージェントと同じです．

6.2.6 トレーナのコマンド

トレーナエージェントはコーチエージェントが使用できるコマンドに加え，審判と同様に試合を制御するコマンドなども使用できます．ここでは，それらの追加コマンドを説明します．コマンドクラスのコンストラクタは省略し，生成されるコマンド文字列のみを説明します．

TrainerEarCommand

(ear { on|off})

聴覚センサ情報を受信するかどうかのモードを変更します．

TrainerKickOffCommand

(start)

試合を開始します．

TrainerChangeModeCommand

(change_mode $PlayModeString$**)**

プレイモードを変更します．

TrainerMoveBallCommand

(move (ball) X Y **[0** $VelX$ $VelY$**]))**

ボールを任意の位置へドロップします．ドロップ後の速度を指定することもできます．

TrainerMovePlayerCommand

(**move** (**player** $TeamName$ $Unum$) X Y [$BodyAngle$ [$VelX$ $VelY$]]))

指定のプレイヤを任意の位置へドロップします．ドロップ後の体の向きと速度を指定することもできます．

TrainerRecoverCommand

(**recover**)

全てのプレイヤのスタミナを完全に回復します．

TrainerSayCommand

(**say** $Message$)

コミュニケーションメッセージを配信します．$Message$ はコーチ言語ではありません．

Section 6.3
メッセージ解析

rcssserver からサッカーエージェントへはさまざまなメッセージが送信されてき

ます．フィールドやエージェント自身の状態を知るためのセンサ情報だけでなく，送信したコマンドに対するエラーや警告の通知なども含まれます．

rcssserver から送信されるメッセージは全て S 式で記述されています．S 式とは LISP で使用される論理記述方式です．S 式の詳しい説明は省略しますが，要するに括弧で要素の論理構造が表現される記述形式です．ほとんどの場合，括弧内の第一要素が識別子，残りの要素がその識別子に対応するパラメータを意味します．

サッカーエージェントは，これらのメッセージを受信したらそのメッセージ内容の解析をまず第一に行わなければなりません．解析のためのパーサライブラリは librcsc で提供しています[2)]が，メッセージフォーマットを知っておいて損はありません．なお，本節で解説するメッセージフォーマットは，rcssserver-10.0.7 時点のものです．

6.3.1 see メッセージの解析

see は，プレイヤエージェントがフィールド上で見た物体の情報が得るための視覚センサ情報で，プレイヤエージェントの制御においては最も重要なセンサ情報です．このメッセージが得られなければ，プレイヤエージェントはフィールドの状態をほとんど何も知ることができません (自分の位置すら分かりません)．

see にはプレイヤエージェントの視界内に存在する物体についての情報が含まれます．

フィールド上の物体には，静止物体としてライン，フラッグとゴール，移動物体としてボールと他のプレイヤが含まれます．プレイヤエージェントは静止物体を観測することで自分自身の位置測定を行い，その結果に基づいて各移動物体のフィールド上での位置を推定することになります．

see の解析処理は VisualSensor というクラスが担当します．PlayerAgent クラスは，analyzeSee() メンバ関数内で VisualSensor を使用します．

プレイヤエージェントが受信する **see** のフォーマットは以下のようになります．このフォーマットはどのプロトコルバージョンであっても変わりありません．

(**see** *Time Object Object* ...)

プロトコルバージョン 8 以降では，*Object* は次のフォーマットのいずれかとなります．

[2)] librcsc で提供するパーサライブラリは，期待の動作はするもののお世辞にも良い実装とは言えません．今後，は flex&bison のようなパーサジェネレータを利用した実装に置き換えられる可能性が高いため，参考程度にとどめてください．

(*Name Dist Dir* [*DistChange DirChange* [*BodyDir HeadDir* [*ArmDir*] [**t**]]]) : 物体がプレイヤの場合

(*Name Dist Dir* [*DistChange DirChange*]) : 物体がボールまたはフラッグの場合

(*Name Dir*) : ViewQuality が Low の場合

Name は次のフォーマットのうちのいずれかです.

(**p** [**"***TeamName***"** [*Unum* [**g**]]]) または (**P**)

(**b**) または (**B**)

(**f** *FlagName*) または (**F**)

(**g** {**l**|**r**}) または (**G**)

p, **P** はプレイヤ，**b B** はボール，**f**, **F** はフラッグ，**g**, **G** はゴールを表す識別子です[3]．識別子が大文字の場合，その物体はプレイヤエージェントの視界には直接入っていないがすぐ近くに存在しており，"感じられた" ことを意味します．"感じられた" 物体からは距離と角度の情報しか得られません．*FlagName* はフラッグを識別するための名前です.

各物体について得られる情報の量は，プレイヤエージェントの視界モード，および，プレイヤエージェントからその物体までの距離に依存します．ViewQuality が Low であれば，距離の情報すら得られません．また，物体との距離が離れるほど得られる情報は少なくなります．特に，距離情報に関しては，実際の値に量子化という処理を施してからプレイヤへ送信されます．距離情報の量子化については 6.5 節で詳しく説明します.

see メッセージに含まれる各パラメータの詳細を表 6.4 に記します.

[3] プロトコルバージョン 6 以前の see メッセージではメッセージに含まれる物体の識別子のフォーマットが現在のものと少し異なります (ボール: ball, フラッグ: flag など)．しかし，今後古いプロトコルを使う機会は無いので本書では省略します.

パラメータ	意味
Dist	物体への距離．量子化による誤差を含む
Dir	プレイヤエージェントの首の向きから物体への相対方向．精度は 1.
DistChange	物体の相対速度成分．量子化よる誤差を含む．任意の実数値.
DirChange	物体の相対速度成分．精度は 0.1.
BodyDir	プレイヤエージェントの首の向きから対象プレイヤの体の向きへの相対角度．精度は 1.
HeadDir	プレイヤエージェントの首の向きから対象プレイヤの首の向きへの相対角度．精度は 1.
ArmDir	プレイヤエージェントの首の向きから対象プレイヤの指さし方向への相対角度．精度は 1.
t	対象プレイヤがタックルしている，またはタックル直後で動けない.

表 6.4 see メッセージに含まれる物体の情報

6.3.2 hear メッセージの解析

hear は，サッカーエージェントがフィールド上で聞くことができる聴覚センサ情報です．審判，オンラインコーチ，審判，そして他のプレイヤからの say メッセージが含まれます．**hear** は定期的には送信されず，送信すべきイベントが発生した場合にのみ送信されます．ここでは，プレイヤエージェントが受信する **hear** について説明します.

プレイヤエージェントが受信する **hear** メッセージは，メッセージ送信者によってメッセージフォーマットが大きく異なります．そのため，プレイヤエージェントは PlayerAgent クラスの analyzeHear() メンバ関数でメッセージ送信者をまず判別し，送信者のタイプに応じて異なる解析処理を呼び出すようになっています．解析処理は GameMode クラスや AudioSensor クラスなどで実装されています.

hear メッセージはプロトコルバージョン 7 以前とバージョン 8 以降とでそのフォーマットに若干の変更が加えられています．現在はバージョン 8 以降のプロトコルの使用が推奨されており，本書でもバージョン 8 以降のプロトコルの使用

を想定しています．プロトコルバージョン 8 以降の **hear** メッセージは次のフォーマットのいずれかです．

(**hear** *Time* **referee** *PlayModeString*)：送信者は審判

(**hear** *Time* **self** “*Message*”)：送信者は自分自身

(**hear** *Time Dir* **our** *Unum* “*Message*”)：送信者は敵プレイヤ

(**hear** *Time Dir* **opp** “*Message*”)：送信者は味方プレイヤ

(**hear** *Time* **{our|opp}**)：送信者は他のプレイヤ

(**hear coach** *Time* “*Message*”)：送信者はトレーナ

(**hear** *Time* **{online_coach_left** | **online_coach_right }** *CLang*)
：送信者はコーチ

PlayModeString は現在のプレイモードを示す文字列のうちのひとつです．詳しくは付録 A.1 を参照してください．

CLang はコーチ言語のメッセージです．librcsc ではコーチ言語をサポートしていない[4]ため，オンラインコーチの **say** コマンドによるメッセージは全て無効となります．

トレーナエージェントからのメッセージのみ，*Time* の出現順序が逆であることに注意してください (仕様では他のメッセージと同様に *Time* が先に現れることになっていますが，実装では何故か逆になっています)．

送信者が他のプレイヤの場合，*Dir* によって送信してきた方向を知ることができます．更に，送信者の識別子として **our** または **opp** が使用され，送信者が味方プレイヤの場合はその背番号も知ることができます．

他のプレイヤからのメッセージが空の場合は，聴覚センサのキャパシティを越えて **hear** を受信したことを意味します．**ear** コマンドで **partial** を **off** にしておけば，このメッセージは送信されてきません．

自分自身，他のプレイヤ，またはトレーナからの *Message* は二重引用符で囲われている点にも注意してください．

6.3.3 sense_body メッセージの解析

sense_body は，現在の視界モード，残りスタミナ，自分自身のスピードとその方向のような，プレイヤエージェント自身の状態を知るためのパラメータが多

[4] 将来的にはサポートする予定です．コーチ言語のパーサとしては rcssserver 自体に含まれるライブラリを利用することも可能です．

く含まれているセンサ情報です．プレイヤエージェントが実行した各コマンドの総実行回数も得られ，rcssserver へのコマンドの不達または遅延を監視するために使用できます．

sense_body は各サイクルの開始時にプレイヤエージェントへ自動的に送信されることが保証されています．UDP のパケットロストが発生しないかぎりは，**sense_body** の受信によって新しいサイクルの開始を知ることができます．

プレイヤエージェントが受信する **sense_body** メッセージの解析は BodySensor クラスが担当します．PlayerAgent クラスは，analyzeSenseBody() メンバ関数内で BodySensor を使用します．

プロトコルバージョン 8 以降の **sense_body** メッセージのフォーマットは以下のようになります．

> (**sense_body** *Time* (**view_mode** {**high** | **low**} {**narrow** | **normal** | **wide** }) (**stamina** *Stamina Effort*) (**speed** *Speed Angle*) (**head_angle** *Angle*) (**kick** *Count*) (**dash** *Count*) (**turn** *Count*) (**say** *Count*) (**turn_neck** *Count*) (**catch** *Count*) (**move** *Count*) (**change_view** *Count*) (**arm** (**movable** *Count*) (**expires** *Count*) (**target** *Dist Dir*) (**count** *Count*)) (**focus** (**target** {**none** | { {**l** | **r**} *Unum*}}) (**count** *Count*)) (**tackle** (**expires** *Count*) (**count** *Count*)))

合計で 25 個のパラメータが含まれます．それぞれの識別子とそこに含まれる情報の意味を表 6.5 に記します．

識別子	含まれる情報
view_mode	**sense_body** メッセージ送信時の視界モード．ViewWidth と ViewQuality を示す 2 つの文字列が得られる．
stamina	プレイヤエージェントのスタミナ情報．残りスタミナと **effort** の値が誤差無しで得られる．**recover** の値は得られない．
speed	速度の大きさと首の向きに対する相対方向．
head_angle	体の向きに対する首の相対角度．
kick	成功した **kick** コマンドの総実行回数．
dash	成功した **dash** コマンドの総実行回数．
turn	成功した **turn** コマンドの総実行回数．
say	成功した **say** コマンドの総実行回数．
turn_neck	成功した **turn_neck** コマンドの総実行回数．
catch	成功した **catch** コマンドの総実行回数．
move	成功した **move** コマンドの総実行回数．
change_view	成功した **change_view** コマンドの総実行回数．
arm	**pointto** コマンドによる指さしの状態が得られる．**movable** は次に腕が動かせるようになるまでのサイクル数．**expires** は **pointto** の効果が無効になるまでのサイクル数，**count** は成功した **pointto** コマンドの総実行回数．
focus	**attentionto** コマンドに関する情報が得られる．**target** は現在の注意対象を示し，**none** は誰にも注意していない状態，**l** または **r** と *Unum* によって対象プレイヤのサイドと背番号が示す．**count** は成功した **attentionto** コマンドの総実行回数．
tackle	**tackle** コマンドに関する情報が得られる．**expires** は **tackle** の影響が解けて動けるようになるまでのサイクル数，**count** は成功した **tackle** コマンドの総実行回数．

表 6.5 sense_body メッセージに含まれる情報

6.3.4 fullstate メッセージの解析

fullstate はデバッグ目的のために使用されるセンサ情報です．このメッセージによって，プレイヤエージェントはフィールド上の全ての移動物体の完全な情報を得られます．通常の試合で使用されることはありません．

プレイヤエージェントが受信する **fullstate** メッセージの解析処理は FullStateSensor クラスが担当します．PlayerAgent クラスは，analyzeFullState() メンバ関数内で FullStateSensor を使用します．

fullstate メッセージは，プロトコルバージョン 7 以前と 8 以降で大きくフォーマットが異なります．プロトコルバージョン 8 以降の **fullstate** メッセージのフォーマットは以下のようになります．

> (**fullstate** *Time* (**pmode** *PlayModeString*) (**vmode** { **high** | **low** } { **narrow** | **normal** | **high** }) (**count** *KickCount DashCount TurnCount CatchCount MoveCount TurnNeckCount ChangeViewCount SayCount*) (**arm** (**movable** *Count*) (**expires** *Count*) (**target** *Dist Dir*) (**count** *PointtoCount*)) (**score** *OurScore OppScore*) ((**b**) *BallPosX BallPosY BallVelX BallVelY*) *Player Player Player* ...)
>
> *Player* :- ((**p** {**l**|**r**} *Unum* { **g** | *PlayerTypeId* }) *PosX PosY VelX VelY BodyDir NeckDir* [*PointtoDist PointtoDir*] (*Stamina Effort Recover*))

本書ではバージョン 8 以降のプロトコルの使用を想定しているため，バージョン 7 のメッセージフォーマットは省略します．

メッセージ中の識別子とそこに含まれる情報の意味を表 6.6 に記します．

識別子	含まれる情報
pmode	現在のプレイモード文字列.
vmode	現在の視界モード.
count	各動作コマンドの総実行回数
arm	**sense_body** に含まれる情報と同じ
score	現在の試合の得点状況
(b)	ボールの位置と速度が誤差無しで得られる.
(p ...)	各プレイヤのプレイヤタイプ，位置，速度，向き，指さし位置，スタミナの情報を誤差無しで得られる.

表 6.6 fullstate メッセージに含まれる情報

6.3.5 see_global メッセージの解析

コーチエージェントまたはトレーナエージェントが受信する視覚センサ情報です．フィールド上の全ての移動物体の位置と速度を誤差無しで得ることができます．

see_global の解析処理は GlobalVisualSensor というクラスが担当します．CoachAgent クラスまたは TrainerAgent クラスは，analyzeSeeGlobal() メンバ関数内で GlobalVisualSensor を使用します．プロトコルバージョン 7 以降の **see_global** メッセージのフォーマットは以下のようになります．

> **(see_global** *Time* **((g l) -52.5 0) ((g r) 52.5 0) ((b)** *BallPosX BallPosY BallVelX BallVelY***)** *Player Player Player* **...)**
>
> *Player* :- **((p "***TeamName***"** *Unum* **[g])** *PosX PosY VelX VelY BodyDir NeckDir* [*PointtoDir*] **[t])**

Time によるサイクル情報の後，左右のゴール位置座標が挿入されます．*PointtoDir* は指さしの絶対方向です．

6.3.6 パラメータ情報の解析

rcssserver は起動時に変更可能なオプションを多く持っています．これらには，時間モデルや物理モデルに関するパラメータのようなシミュレーションの根幹に

関わるものも多く含まれています．サッカーエージェントもこれらのパラメータの変更を知ることができなければ，正しく動くことはできません．そのため，サッカーエージェントの接続直後にそれらの設定パラメータが自動的に送信されることになっています．送信されるメッセージは以下の 3 種類です．

- **server_param**

  設定ファイルの~.rcssserver/server.conf の内容がこのパラメータセットに対応します．**server_param** メッセージは，サッカーエージェントの接続直後に一度しか送られてきません．含まれるパラメータの詳細は A.2.1 を参照してください．

- **player_param**

  ヘテロジニアスプレイヤのパラメータ生成に関するパラメータセットです．設定ファイルの~.rcssserver/player.conf の内容がこのパラメータセットに対応します．**server_param** メッセージは，サッカーエージェントの接続直後に一度しか送られてきません．含まれるパラメータの詳細は A.2.2 を参照してください．

- **player_type**

  ヘテロジニアスプレイヤのパラメータです．現在の公式設定では，デフォルトタイプを含めて合計で 7 種類のヘテロジニアスプレイヤが生成されることになっています．そのため，**player_type** メッセージは，サッカーエージェントの接続直後に合計 7 回送信されてきます．含まれるパラメータの詳細は A.2.3 を参照してください．

パラメータ情報のメッセージは，ServerParam，PlayerParam，PlayerType クラスでそれぞれ解析されます．プロトコルバージョン 8 以降の **server_param**，**player_param**，**player_type** メッセージのフォーマットは以下のようになります．

(*ParamId ParamPair ParamPair ParamPair* **...**)

ParamId :- **{ server_param** | **player_param** | **player_type }**

ParamPair :- (*ParamName ParamValue*)

最初の識別子以降は全て，パラメータの名前とその設定値とのペアが繰り返されます．パラメータの名前には，server.conf や player.conf で使用されているものと同じ名前が使用されます．

6.3.7 その他のメッセージの解析

センサ情報とパラメータ情報以外にも rcssserver からサッカーエージェントへ送信されるメッセージとして以下のようなものがあります.

- **init** メッセージ, **reconnect** メッセージ

  init コマンド，**reconnect** コマンドに対する応答メッセージです．自分のチームのサイド，背番号，プレイヤモードなどが得られます.

- **ok** メッセージ

  トレーナエージェントの試合操作コマンドや，**ear** や cmdpartscore などのプレイヤエージェントの設定変更コマンドが正しく受け付けられたことを通知するためのメッセージです．プレイヤエージェントの各動作コマンドに対して **ok** メッセージが送信されることはありません.

- **error** メッセージ

  主な目的は不正なコマンド送信に対する通知です．送られてきたコマンドのフォーマットに誤りがある場合に返信されることが多いようです.

- **warning** メッセージ

  送信される目的は **error** とほとんど違いありません．コマンドのフォーマットは正しいが，現在は使用すべきではない場合に送られることが多いようです.

これらは，サッカーエージェントが送信したコマンドに対する応答メッセージとして送信されます．フォーマットは単純なため，特定のパーサは用いず PlayerAgent の doAnalyzeInit() メンバ関数などで解析を完了させています.

ok，**warning**，**error** の各メッセージには，そのメッセージ識別子とともに rcssserver からの応答内容が含まれます．この応答内容はクライアントプログラムのデバッグに非常に有益です．メッセージのフォーマット自体は非常に単純なため，PlayerAgent などの analyzeOK(), analyzeWarning(), analyzeError() といったメンバ関数で解析処理を行っています．librcsc が提供するサッカーエージェントは，**error** メッセージと **warning** メッセージを標準エラー出力へそのまま出力します．これらのメッセージが出力がされた場合，サッカーエージェントは開発者の意図どおりに動作していないと予想されます．送信したコマンドの妥当性を必ず確認しなければなりません.

rcssserver から送られる応答メッセージの種類を表 6.7，6.8，6.9 に記します.

メッセージ内容	対応するコマンド	受信者
(ok compression *Level*)	**compression**	全て
(ok change_player_type *Unum Type*)	**change_player_type**	コーチ
(ok change_player_type *TeamName Unum Type*)	**change_player_type**	トレーナ
(ok clang (ver *Min Max*))	**clang**	プレイヤ
(ok change_mode)	**change_mode**	トレーナ
(ok move)	**move**	トレーナ
(ok say)	**say**	コーチ, トレーナ
(ok ear {on|off})	**ear**	トレーナ
(ok eye {on|off})	**eye**	コーチ, トレーナ
(ok look ...)	**look**	コーチ, トレーナ
(ok start)	**start**	トレーナ
(ok recover)	**recover**	トレーナ
(ok say)	**say**	コーチ, トレーナ
(ok team_graphic *X Y*)	**team_graphic**	コーチ
(ok team_names [(team l *TeamName*)] [(team r *TeamName*)])	**team_names**	コーチ, トレーナ

表 6.7 ok メッセージの種類

メッセージ内容	対応するコマンド	受信者
(error illegal_command_form)	全て	全て
(error only_init_allowed_on_init_port)	**init**	全て
(error no_more_team_or_player_or_goalie)	**init**	プレイヤ
(can't reconnect)	**reconnect**	プレイヤ
(error reconnect)	**reconnect**	プレイヤ
(error no team with name $TeamName$)	**ear**	プレイヤ
(error too_many_moves)	**catch** 後の **move** 使用回数制限を違反	プレイヤ (キーパ)
(error no_such_team_or_already_have_coach)	**init**	コーチ
(error could_not_parse_say)	**say**	コーチ
(error unknown_command)	全て	コーチ, トレーナ
(error said_too_many_meta_messages)	**say**	コーチ
(error said_too_many_freeform_messages)	**say**	コーチ
(error cannot_say_freeform_while_playon)	**say**	コーチ
(error said_too_many_advice_messages)	**say**	コーチ
(error said_too_many_define_messages)	**say**	コーチ
(error said_too_many_del_messages)	**say**	コーチ
(error said_too_many_info_messages)	**say**	コーチ
(error said_too_many_rule_messages)	**say**	コーチ
(error out_of_range_player_type)	**change_player_type**	コーチ, トレーナ
(error illegal_mode)	**change_mode**	トレーナ
(error illegal_object_form)	**move**	トレーナ

表 6.8 error メッセージの種類

メッセージ内容	対応するコマンド	受信者
(warning message_not_null_terminated)	全て	全て
(warning cannot_change_goalie)	**change_player_type**	コーチ
(warning cannot_sub_while_playon)	**change_player_type**	コーチ
(warning compression_unsupported)	**compression**	全て
(warning invalid_tile_location)	**team_graphic**	コーチ
(warning invalid_tile_size)	**team_graphic**	コーチ
(warning max_of_that_type_on_field)	**change_player_type**	コーチ
(warning no_team_found)	**change_player_type**	トレーナ
(warning no_such_player)	**change_player_type**	コーチ, トレーナ
(warning no_subs_left)	**change_player_type**	コーチ
(warning only_before_kick_off)	**team_graphic**	コーチ

表 6.9 warning メッセージの種類

Section 6.4
rcssserver との同期

rcssserver はサッカーエージェントからのコマンドを待ってくれません．コマンドが届かなければ届かないで，非同期にどんどんシミュレーションを進めていってしまいます．サッカーエージェントを期待どおりに動作させるには，サッカーエージェントの意思決定のタイミングを rcssserver に同期させる必要があります．しかしながら，通常の設定でシミュレーションを実行する限り，完全な同期を実現することは不可能です．そのため，サッカーエージェントの開発においては，同期の精度を高めるための工夫と同期が崩れた場合への頑健な対応が必要不可欠です．rcssserver との同期は対応が非常に難しい問題として知られています．

本節では，プレイヤエージェントを rcssserver と同期させるための基礎知識とテクニックの概要を解説します．なお，rcssserver のシミュレーション時間は server.conf 等を編集することで変更可能となっていますが，本書では公式試合で

使用される設定に基づいて解説します.

6.4.1 センサ情報の受信タイミング

プレイヤエージェントが得られるセンサ情報のうち，**sense_body** と **see** は定期的に自動送信されます．**sense_body** はサイクル更新直後に送信されることが（実際に届くかどうかは別として）保証されています．すなわち，**sense_body** は 100 ミリ秒に一回の受信機会があり，かつ，プレイヤエージェントは **sense_body** によってサイクルの開始タイミングを知ることができます．一方，**see** の送信間隔はシミュレーションサイクルとは非同期で，デフォルトでは 150 ミリ秒に一回です．そのため，プレイヤエージェントはサイクルの途中に **see** を受信することがあります．審判からの **hear** はプレイモードが変更された直後に，他のプレイヤからの **hear** はサイクル開始直後に送信されます．**fullstate** の送信機会は **sense_body** と同時です．サイクル開始直後はセンサ情報が重複して送信されることがありますが，その場合の送信順序は，審判からの **hear** → **sense_body** → **fullstate** → 他のプレイヤからの **hear** → **see** となっています.

コーチエージェントやトレーナエージェントのための **see_global** メッセージはサイクル開始直後に送信されることが保証されています.

6.4.2 意思決定のタイミング

サッカーエージェントからのコマンドは，rcssserver のサイクル更新処理が実行されるまでに送信を完了し，rcssserver に受理されなければなりません．UDP 通信の不確実性を考慮すると，かなりの余裕を持ったコマンド送信が必要です．しかし，逆にコマンド送信が早過ぎると，得られるはずだったセンサ情報を受信しないまま，不利な情報量で意思決定をしなければなりません．よって，サッカーエージェント，特にプレイヤエージェントには，サイクルが更新された時間とそのサイクルの有効期限を知る能力，そして，意思決定のタイミングを取る能力が必要となります.

プレイヤエージェントは **sense_body** によってサイクル開始時間とそのサイクルの有効期限を知ることができます．そして，**see** がそのサイクル内で受信できそうならば，受信できるまで意思決定を意図的に遅れさせる必要があります．この待機と経過時間の監視は，6.1.4 節で説明した select() によるポーリングによって実現できます．現実的には，サイクル開始から 70 ミリ秒程度経過するまでに意

思決定処理を開始することが望ましいようです．後は，待機時間中に **see** が得られるように受信タイミングを上手く調節できるか否かが鍵となります．

6.4.3 プレイヤの視界モード

rcssserver からの **see** の送信頻度はプレイヤエージェントの視界モードに応じて変化します．プレイヤエージェントは，**change_view** コマンドを実行して自分の視界モードを変更し，**see** の受信頻度を能動的に変化させることができます．

視界モードは，ViewWidth と ViewQuality の組合せで表現されます．rcssserver-10.0.7 では，ViewWidth は 3 タイプ，ViewQuality は 2 タイプあるので，合計で 6 パターンの視界モードが存在します．それぞれの特性を表 6.10 と 6.11 に記します．

タイプ	特徴
narrow	視野角 45 度．**see** の受信頻度は **normal** の 2 倍．
normal	視野角 90 度．**see** の受信頻度はデフォルト値．
wide	視野角 180 度．**see** の受信頻度は **normal** の 01/2．

表 6.10 ViewWidth のタイプ

タイプ	特徴
low	物体の距離と速度が一切観測できない．**see** の受信頻度は **high** の 2 倍．
high	通常の品質．**see** の受信頻度はデフォルト値．

表 6.11 ViewQuality のタイプ

ViewQuality のタイプによって，観測できる情報には以下の表 6.12 のような違いが現れます．low タイプでは距離と速度の情報が得られないため，通常，low タイプを使うことはあり得ません．

物体の情報	high のとき	low のとき
方向	得られる	得られる
距離	得られる	得られない
速度	得られる	得られない
チーム名・背番号	得られる	得られる

表 6.12 ViewQuality のタイプによる情報の違い

6.4.4 see の頻度

change_view コマンドを一度も実行していないデフォルトの状態でのプレイヤエージェントの視界モードは，(normal, high) となります．デフォルトの **see** の送信間隔 150 ミリ秒は，この (normal, high) に適用されます．全てのパターンでの **see** の受信頻度を表 6.13 に記します．

視界モード	see の頻度 [ミリ秒]
(narrow, high)	75 (=150/2/2)
(normal, high)	150 (=150*1*1) (デフォルト)
(wide, high)	300 (=150*2*1)
(narrow, low)	37.5 (=150/2/2)
(normal, low)	75 (=150/2*1)
(wide, low)	150 (=150*2/2)

表 6.13 視界モードと see の頻度

この受信頻度の計算方法は非常に簡単で，直感的に理解できます．ViewWidth が 2 倍になれば頻度は半分に，逆に ViewWidth が半分になれば頻度は 2 倍です．同様に、ViewQuality が high から low になれば頻度は 2 倍に，その逆は半分になります．つまり，一度により多くの情報を得ようとすれば，情報を得られる頻度が下がるように設計されています．

ただし，表中で示している頻度はあくまで理論値であって，厳密に 37.5 ミリ秒や 75 ミリ秒の間隔で see メッセージが受信できるわけではない点に注意してくだ

さい．rcssserver は独自のタイマモジュールを持っていて，表中の間隔に近くなるように擬似的な時間管理を行っています．

6.4.5 rcssserver の時間モデル

rcssserver の時間モデルは rcssserver の StandardTimer というクラスで実装されています．StandardTimer::run() という関数がプログラムメインループとなっており，大雑把に言って以下のことを行っています．

- 無限ループ．
- 通常時は sigpause でスリープ．
- 10ms(param.h の TIMEDELTA) に一回アラーム割り込み．
  - 割り込みごとに経過時間を更新．
  - 経過時間が各イベント (サイクル更新、センサ情報配信など) のタイミングに達していれば、その処理を実行．

StandardTimer::run() では，表 6.14 の変数によって各イベントのタイミングを管理しています．いずれも StandardTimer::run() 内部で定義されているローカル変数で，q_* の変数の値は、server.conf で定義されているシミュレーションステップ関連のパラメータから自動的に決定されます．ServerParam::lcmStep() の値は、各シミュレーションステップ関連パラメータの最小公倍数 (=300) です。

変数名	役割	値
lcmt	経過時間格納	割り込みごとに更新
c_simt	シミュレーションステップ更新回数カウント	発生ごとに更新
c_sent	see メッセージ送信回数カウント	発生ごとに更新
c_rect	クライアントコマンド受信回数カウント	発生ごとに更新
c_sbt	sense_body メッセージ送信回数カウント	発生ごとに更新
c_sbt	coach 用 see メッセージ送信回数カウント	発生ごとに更新
q_simt	300(ServerParam::lcmStep()) ミリ秒間でのシミュレーションステップ更新回数	3 で固定
q_sent	300(ServerParam::lcmStep()) ミリ秒間での see メッセージ送信回数	8 で固定
q_rect	300(ServerParam::lcmStep()) ミリ秒間でのクライアントコマンド受信回数	30 で固定
q_sbt	300(ServerParam::lcmStep()) ミリ秒間での sense body メッセージ送信回数	3 で固定
q_svt	300(ServerParam::lcmStep()) ミリ秒間での coach 用 see メッセージ送信回数	3 で固定

表 6.14 Stdtimer::run() のイベント管理変数

ここで，q_sent の値が 8，すなわち **see** メッセージが 3 サイクル (300 ミリ秒) に 8 回送信される機会がある，と言う点が重要です．これが意味することは，$300 = 37.5 * 8$，つまり，視界モードが (narrow, low) であれば，3 サイクルの間に 8 回 **see** が送信されるということです．理想的には，0, 37.5, 75, 112.5, 150, 187.5, 225, 262.5, 300(=0.0) のタイミングで **see** の送信処理が実行されます．しかし，前述の通り，rcssserver のアラームの割り込み間隔は 10 ミリ秒であるため，実際の送信タイミングは少しずれます．実際には，lcmt(経過時間) が 37.5 * c_sent を越えるたびに **see** の送信処理が実行されるため，図 6.1 の矢印で示す位置になります．図中の一目盛は 10 ミリ秒で,，下段の括弧内の数値は理想的な送信時間、上段の数値は実際の送信時間を意味します．シミュレーションサイクルは，0, 100, 200, 300 の時点で更新され，各物体の位置や速度などの更新も全てこのときに行われます (逆に，これら以外の時には一部例外を除いて何も更新されません)．

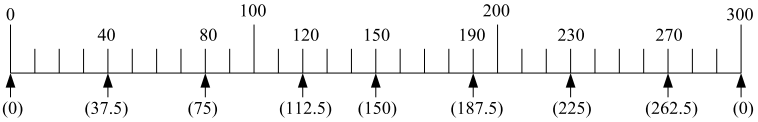


図 6.1 see の送信タイミング

この図は View Mode が (narrow, low) のときの **see** の送信タイミングを示していますが，この送信タイミング集合は任意の視界モードでの送信タイミングを包含しています．言い換えると，どの視界モードあっても，この図で示されたタイミングのいずれかでしか **see** は送信されないということです．

6.4.6 see の送信タイミング

前節で述べたように，**see** は特定のタイミングでしか送信されません．実際の送信パターンを図 6.2 と 6.3 に示します．図中の大きな矢印部分が **see** が送信されるタイミングを表しています。

(narrow, high) の場合

合計 2 パターンで，2 回に 1 回の送信機会があります．1 サイクルに最低 1 回は **see** の送信機会があることが分かります．

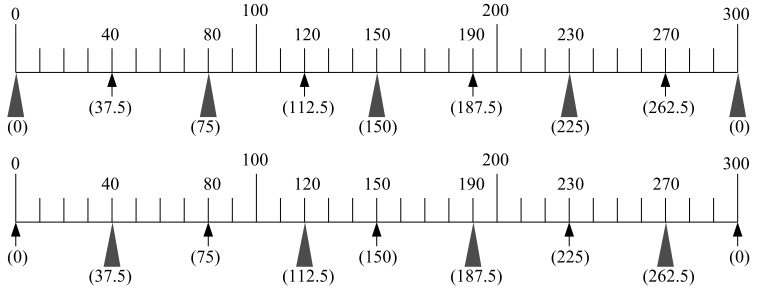


図 6.2 (narrow, hing) モードでの送信タイミング

(normal, high) の場合

合計 4 パターンで，4 回に 1 回の送信機会があります．3 サイクルに 2 回しか **see** の送信機会が無く，しかも，パターンによっては送信タイミングがサイクル終盤になってしまいます．

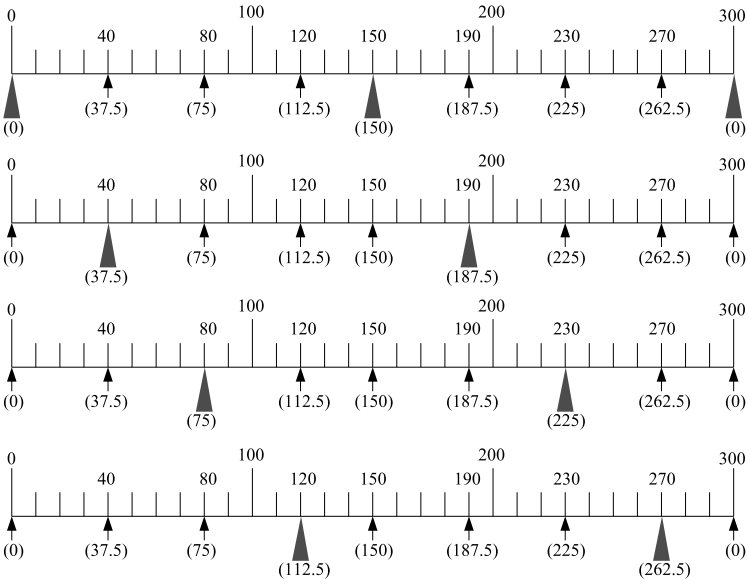


図 6.3 (normal, hing) モードでの送信タイミング

(wide, high) の場合

合計 8 パターンで，送信機会は 8 回に 1 回だけになります．図は省略します．

6.4.7 change_view コマンドの作用

6.4.5 節で書いたとおり、rcssserver が **see** を送信する機会は 300 ミリ秒間に 8 回あります．rcssserver はこの 8 回全てにおいて，**see** を送信すべきか否かのチェックを全プレイヤに対して行います．rcssserver 内部の Player オブジェクトは，それぞれが **see** 送信チェック用のカウンタを持っています．このカウンタは **see** の

送信機会ごとに（かつ、その時にのみ）インクリメントされ，カウンタが規定値以上であれば **see** が送信されます．送信が終了すると，カウンタは 0 にリセットされます．

change_view コマンドはこの規定回数を変更します．規定回数はプレイヤの現在の視界モードに応じて表 6.15 のように決められています．

視界モード	規定回数
(narrow, low)	1
(narrow, high)	2
(normal, low)	2
(normal, high)	4 (デフォルト)
(wide, low)	4
(wide, high)	8

表 6.15 see 送信チェックのカウンタ規定回数

(narrow, low) モードの規定回数は 1 なので，送信機会ごとに送信されることが分かります．各モードの回数の求め方は，(normal, high) の規定回数を基準として，表 6.10 と 6.11 に示した頻度の変更割合を掛けるのみです．興味があれば，rcssserver の src/player.cc 内で定義されている Player::change_view() という関数を参照してみてください．

ここで注意しなければならないのは，**change_view** コマンドは規定回数を変更するのみで，現在のカウンタは変更しない，という点です．つまり，**change_view** コマンドは次の **see** の送信タイミングまでに受信されれば良いことになります．以下に具体例を示します．

まずは，(narrow, high) を維持する場合のカウンタの更新の様子を図 6.4 で確認してください．図中の一目盛は 10 ミリ秒，矢印は see の送信チェックタイミング，大きな矢印は実際に see が送信されるタイミングです．矢印の下の数値は，カウンタの更新の様子を示しています．

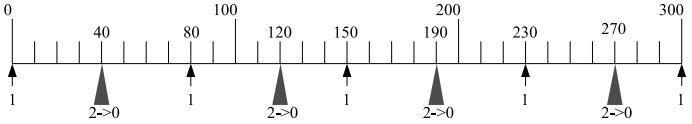


図 6.4 (narrow, high) を維持する場合

6.4 の状態で **change_view** コマンドを実行し，(normal, high) モードへ変更すると，カウンタの様子は図 6.5 のようになります．それぞれのチャートで **change_view** コマンドの受信タイミングが異なりますが，結果として得られる状態はまったく同じです．rcssserver 内部でカウンタがリセットされるまでにコマンドが受信されれば良いことが分かります．

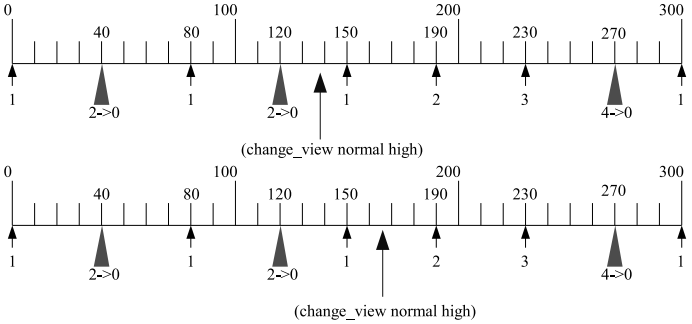


図 6.5 (narrow, high) → (normal, high) へ変更

次に，(normal, high) モードから (narrow, high) モードへ変更する場合を考えます．プレイヤエージェントからの **change_view** の送信タイミングをずらしていくと，カウンタの様子は図 6.6 のようになります．

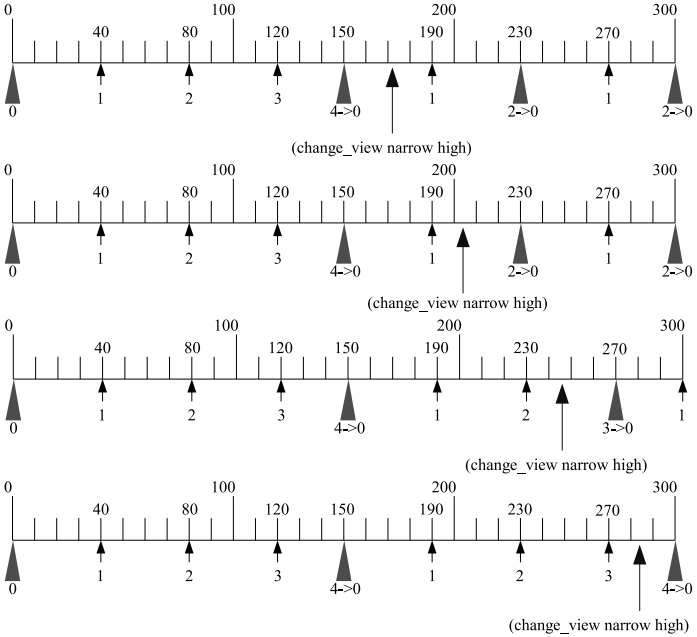


図 6.6 (normal, high) → (narrow, high) へ変更

(narrow, high) モードでのカウンタ規定回数は 2 なので、下の 2 つではカウンタが強制的にリセットされています．これは，すぐに **see** が送信されたはいいが，**see** を得る機会を減らしてしまっている，ということを意味します．**normal** から **narrow**，**wide** から **normal** のような，**see** の送信頻度がより高い視界モードへ変更する場合には注意が必要です．

6.4.8 see の同期

ここまでの知識を駆使することで，ようやく **see** のタイミング調整の方法が見えてきます．6.4.4 節の表 6.13 を良く見ると，(narrow, high)2 回と (normal, high)1 回を組み合わせると $75 + 75 + 150 = 300$ となり，300 ミリ秒 (=3 サイクル) で 3 回，すなわち毎サイクルの **see** の受信が期待できそうであることが分かります．これまでに，**see** の同期を実現する方法として主に 2 つの方法が考えられています．

同期方法その１

rcssserver の **see** 送信タイミングのうち，1 サイクル 1 回の **see** 送信を可能にするのは，図 6.7 と図 6.8 に示す 2 つのパターンが考えられます．

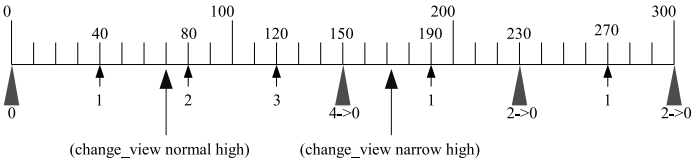


図 6.7 see の同期パターン (1)

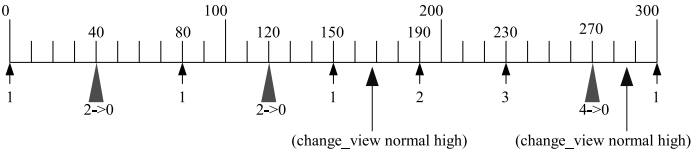


図 6.8 see の同期パターン (2)

これらのうち，図 6.7 がより望ましいパターンと言えます．理由は，図 6.8 のパターンでは see の受信がサイクル終盤 (70 ミリ秒以降) になってしまう場合があるためです．

図 6.7 のパターンでは，3 サイクルに 1 度，サイクル開始直後に see を受信する機会があります．つまり，サイクル開始直後に see を受信できるように受信間隔を調整できれば，図 6.7 の受信間隔を実現できることになります．

この調整は視界モードを (narrow, low) にすることによって行います．(narrow, low) モードであれば全ての see 送信機会の see を受信できるので，目的のタイミングで see を受信できるときが必ず訪れます．see メッセージ送信処理タイミングの図を見ると，(narrow, low) モードの場合，各サイクルでの see の送信回数は 3 回 → 3 回 → 2 回 のループになっています．この回数をチェックすることでサイクル開始直後の see か否かを判断できます．一度タイミングを取れれば，以降は現在の状態を監視しながら適切な **change_view** コマンドを適切なタイミングで送信すれば同期を維持できます．

この方法によって，少々であれば通信の遅延の影響を受けず，確実に同期できます．タイミング調整の方法を少し工夫すれば，***synch_mode*** で動作する rcssserver

でも使用できます．

その代わり，極端な通信の遅延や不達には脆く，一度同期が崩れるとどうしようもなくなります[5]．立て直すには，もう一度 (narrow, low) モードでのタイミングの監視を実行しなければなりません．そして，この手法の最大の欠点は rcssserver の仕様に依存し過ぎていることです．今後，もし rcssserver の仕様が変更されれば，全く役に立たなくなります．

同期方法その２

２つ目の方法はその１の方法に比べて原理はかなり単純です．**sense_body** 受信から **see** 受信までの経過時間に応じて **change_view** を使い分けるだけです．

sense_body はサイクル開始直後に自動的に送信されることが保障されているため，**sense_body** 受信時からの経過時間を測定することで，1 サイクル内での現在の経過時間を知ることができます．そして，各視界モードでの理想的な **see** の送信間隔は判明しているので，**see** を受信した時間を計測すると **see** の受信時間がサイクル内のどのあたりになるかを予測できます．

以上から，**see** の同期のためには，以下の条件を満たす場合に **change_view** コマンドの実行を考慮することになります．

- 現在のサイクルで **see** を受信した（**sense_body** と **see** を同じサイクル内で受信した）．
- 同サイクル内に再度 **see** を受信する可能性がある．
- ViewWidth を 1 段階広くした場合，次のサイクルの早い段階で **see** を受信できる．

疑似コードは以下の様になります．

[5] 大会などでは，この欠点のために不具合を起こすチームも見られます．

```
if ( see の受信サイクル != 現在のサイクル
     || 現在の ViewWidth != NARROW )
{
  doChangeView( NARROW );
  return;
}
msec_diff = (see 受信時間)[ms]- (sense_body 受信時間)[ms]
if ( msec_diff + 75 < 100
     && see_time + 150 < 100 + 70 )
{
   doChangeView( NORMAL );
}
```

実際の実装は多少無駄を省いたものになっていますが，原理は同じです．疑似コード中では定数のパラメータとして'70' が埋め込まれています．このパラメータは，次のサイクルの 70 ミリ秒までを **see** を受信するまでの待機時間とすることを意味しています．実際のソースでは外部設定ファイル player.conf によって変更可能なパラメータとして実装しています．動作させる環境によってはパラメータチューニングを行う必要があるでしょう．

この方法の利点はとにかく頑健であることです．その場に応じた対処がなされるため，通信が遅延しようが不達になろうがほとんど考える必要はありません．多少のことにはびくともせずに，プレイヤエージェントは動き続けられます．

欠点は同期の精度が非常に悪いことです．理想的なパターンでの **see** の受信を維持することは，恐らく難しいでしょう．センサ情報の受信時間を測定したところでそのメッセージは遅延して届いたかもしれないという問題は依然として残ります．最大の欠点は，***synch_mode*** を有効にしてで動作する rcssserver では使えないということです．***synch_mode*** が有効な場合，rcssserver は CPU を最大限に使いきって高速シミュレーションを行うため，時間計測が何の意味も持ちません．***synch_mode*** 使用時は，方法その 1 を使用せざるを得ません．

より確実な同期のために

プレイヤの意思決定にかかる計算時間とコマンド送信の遅延を考慮すると，プレイヤの意思決定は遅くとも **sense_body** 受信から 70 ミリ秒以内には開始するのが理想です．しかしながら、rcssserer とプレイヤエージェントが物理的に別の

ホストで動作する場合，通信の遅延は想像以上に発生します[6]．通信の遅延以外にも，rcssserver が動作するマシンに何らかの負荷がかかってしまい，rcssserver の処理が遅れる場合であっても影響が現れます．20 ミリ秒以上の遅延もそれほど珍しくないため，方法その 1 だけは安定した動作は望めません．そこで，同期の確実性を高めつつより頑健に動作させるには，librcsc では以下のように上記 2 つの方法を組み合わせて使用しています．

1. 方法 1 で完全な同期を実施
2. もし同期が崩れたら (**see** が遅延したら)
   (a) 一時的に方法 2 を採用
   (b) 安全な状況 (プレイモードが非 play_on のときなど) で再び方法 1 を実施

6.4.9 まとめ

本節で述べた同期方法はすでに librcsc に組み込まれており，自動的に実行されます．同期のための情報管理を SeeState クラスが行い，実際の視界モード変更は ActV_SyncView クラスが実行します．SeeState クラスでは，**see** の受診時と **change_view** コマンド送信時に状態が更新され，それに応じて使用可能な視界モードが決定されます．

チーム開発のみに集中したい場合は，細かいタイミングの管理を開発者が意識する必要はありません．しかし，**change_view** コマンドの使用制限が厳格過ぎて不便に感じることがあるかもしれません．そのような場合は，SeeState クラスの実装を改変したり，PlayerAgent クラス内で定義されている adjustSeeSyncormalMode() などの関連する関数を改変するなどすれば良いでしょう．

rcssserver との同期は非常にデリケートな問題であるため，改変する場合は本節で解説した内容を考慮し，注意して行ってください．充分に理解しないうちに変更を加えると，プレイヤエージェントが動作しなくなる可能性があります．

[6] 公式の大会ではまず間違いなく発生します．

Section 6.5
位置測定

rcssserver への接続に成功したプレイヤエージェントは自動的にセンサ情報を受信し始めます．センサ情報を受信したプレイヤエージェントが次にすべきことは，自分が今，フィールドのどこにいるのかを知ることです．これは，プレイヤエージェント内部に世界を再構築するための，最も基本的かつ重要な処理です．本節では，プレイヤエージェント，および他の物体の位置や速度を推定する方法を説明します．

6.5.1 仮想フィールド上の物体

rcssserver 内部の仮想フィールドには，複数の種類の物体が存在しています．当然ながらボールとプレイヤは物体として存在しており，プレイヤエージェントはこれらの情報を **see** で得ることができます．ボールやプレイヤはフィールド上を自由に移動する移動物体です．これらとは別に，静止物体というものが存在します．**see** によって静止物体の情報を得ることはできますが，それらに触れることはできません．ただ，“見えるだけ” の存在です．静止物体は絶対に移動しないという特徴も持っており，プレイヤエージェント自身の位置測定のためのランドマークとして利用することができます．

注意すべきは，**see** によって得られる全ての観測情報には誤差が含まれるということです．位置測定おいては，これらの情報の誤差を適切に考慮する技術が極めて重要となります．

静止物体

観測できる静止物体には，フラッグ，ゴール，ラインの 3 種類が存在します．フラッグはフィールド上に合計 53 個が点在しています．ゴールは左右のゴールの中央の位置で合計 2 個です．ラインは上下左右の 4 本のラインです．全ての静止物体には一意の名前が付いています．**see** にはこれらの名前が含まれるため，どの物体に関する情報かを識別できるようになっています．それぞれの具体的な位置は図 6.9 を参照してください．

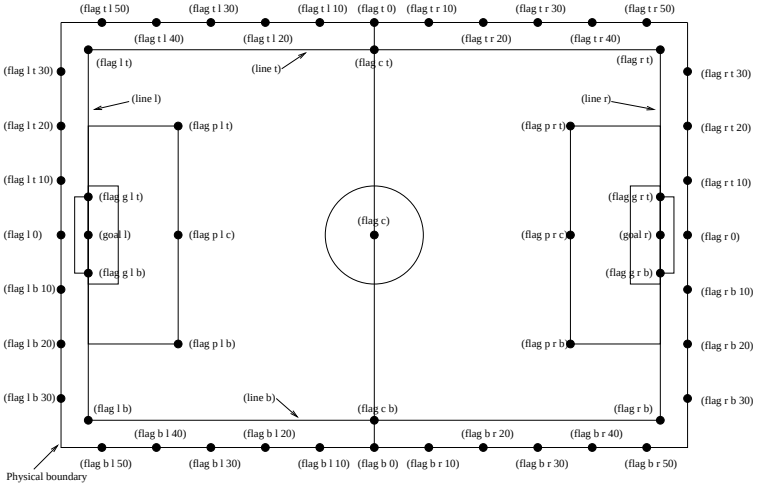


図 6.9 フィールド上の静止物体

これら静止物体の位置が今後変更される予定は無いため，librcsc では各静止物体の位置座標をソースコードに埋め込んでしまっています．これは，ObjectTable クラスの createLandmarkMap() というメンバ関数で以下のように実装されています．

```
    ...
const double rotate = ( M_our_side == LEFT ) ? 1.0 : -1.0;

M_landmark_map[Goal_L]  = Vector2D(-pitch_half_l,   0.0) * rotate;
M_landmark_map[Goal_R]  = Vector2D( pitch_half_l,   0.0) * rotate;

M_landmark_map[Flag_C]  = Vector2D(  0.0,           0.0) * rotate;
M_landmark_map[Flag_CT] = Vector2D(  0.0, -pitch_half_w) * rotate;
M_landmark_map[Flag_CB] = Vector2D(  0.0,  pitch_half_w) * rotate;
    ...
```

全ての静止物体の位置座標は，rcssserver の座標系，すなわち左チームの座標系を基準に定義されています．そのため，右チームのプレイヤエージェントは，位置測定で得られる値をすべて反転して使用しなければなりません librcsc では，初め

から自分のサイドに応じたフラッグの位置自体を保持することにしています．上記ソースコード中では，*rotate* という変数がこの反転を行っています．

全ての静止物体の具体的な位置座標が知りたい場合は，公式マニュアルか rcssserver や librcsc のプログラムソースを参照してください．しかし，静止物体の位置座標は，一度プログラムに書いてしまえば二度と書き直す必要の無い部分なので，これらを覚える必要はありません[7]．

6.5.2 物体の方向

see で得られる物体の方向は度数で得られ，その値にはほとんど誤差がありません．含まれる誤差は，実数値を整数に丸めることによって生じた丸め誤差のみです．プロトコルバージョン 7 以降では，rint() 関数を用いて近い整数へと丸められます[8]．よって，方向の誤差は常に ±0.5 度 となります

方向に関する情報はすべて，プレイヤエージェントの首の絶対方向からの相対角度で得られる点に注意してください．

6.5.3 量子化された距離情報

see で得られる物体への距離情報は，プレイヤエージェントの中心から観測対象の中心までの距離の値で，単位は (仮想フィールドにおける) メートルです．ただし，距離情報には rcssserver によって意図的な誤差が混入されます．この処理は一般に量子化 (quantization) と呼ばれており，プレイヤエージェントと物体との距離が離れれば離れるほど，得られる情報に含まれる誤差が大きくなるように設定されています．量子化の式は以下のように定義されています．

$$d' = Quantize(exp(Quantize(log(d), qstep)), qstep)$$

$$Quantize(V, Q) = rint(V/Q) \times Q$$

この式によって，線形な連続値が階段状に飛び飛びの離散値へと変換されます．d が実際の距離，d' が観測される値となります．qstep は量子化のパラメータで，静止物体には 0.01，移動物体には 0.1 が使用されます．静止物体に比べて移動物体は非常に

[7] 筆者自身も具体的な位置座標はもはやほとんど覚えていません．

[8] プロトコルバージョン 6 以前では，単純に int 型にキャストするという丸め方を行っています．期待値と誤差の範囲が全く異なってしまうため，古いプロトコルの仕様は全く推奨されません．

大きな観測誤差を含むことになります．具体的には，静止物体の場合 $100m$ 離れると $\pm 0.5m$ ほどの誤差が，移動物体の場合は $100m$ 離れると $\pm 5m$ ほどもの誤差が生じます．rcssserver のソースでは，visual.cc の VisualSenderPlaeyrV1::calcQuantDist() という関数がこの計算を担当しています．

位置測定においては，この量子化による誤差を逆算する方法を採用しています．現実世界の位置測定には全く適用できませんが，rcssserver においては間違いなく最適なやり方です．逆算方法としては，以下の二つがあります．

1. 量子化関数の逆関数を用意し，毎回計算する．

2. あらかじめテーブルを用意しておく．

(1) の方法は実にスマートで，rcssserver の仕様変更にも強いという利点があります．しかし，計算過程における誤差のために正確な値が得られない場合があります．一方，(2) の方法では仕様変更には弱いものの，正確な値が高速に得られるという利点があります．いずれの方法が良いかは議論が分かれるかと思いますが，librcsc では (2) の方法を採用しています．

実際の実装は ObjectTable クラスで行われています．以下のような大量の数値群をソースファイル中で見つけられることでしょう．ここでは std::vector に，観測距離，期待値，誤差の 3 つの情報からなる構造体を格納しています．

```
   ...
M_static_table.push_back(DataEntry(12.90, 12.935979, 0.064680));
M_static_table.push_back(DataEntry(13.10, 13.065988, 0.065330));
M_static_table.push_back(DataEntry(13.20, 13.197303, 0.065986));
M_static_table.push_back(DataEntry(13.30, 13.329938, 0.066649));
M_static_table.push_back(DataEntry(13.50, 13.463906, 0.067319));
   ...
M_movable_table.push_back(DataEntry(7.40, 7.398295, 0.369608));
M_movable_table.push_back(DataEntry(8.20, 8.176380, 0.408479));
M_movable_table.push_back(DataEntry(9.00, 9.036297, 0.451439));
M_movable_table.push_back(DataEntry(10.00, 9.986652, 0.498917));
M_movable_table.push_back(DataEntry(11.00, 11.036958, 0.551389));
   ...
```

テーブルから値を取り出す場合は，以下のように STL のアルゴリズムを用いて簡潔に書くことができます．std::lower_bound は二分探索を行うアルゴリズムです．

```
bool ObjectTable::getStaticObjInfo( const double & seen_dist,
                                    double * ave,
                                    double * err ) const
{
  std::vector< DataEntry >::const_iterator
      it = std::lower_bound( M_static_table.begin(),
                             M_static_table.end(),
                             DataEntry( seen_dist - 0.001 ),
                             DataEntryCmp() );
  if ( it == M_static_table.end() ) return false;
  *ave = it->M_average;
  *err = it->M_error;
  return true;
}
```

6.5.4 基本的な位置測定

プレイヤエージェント自身の位置測定は，概ね以下の二つのステップで行います．

1. 観測されたラインの情報を使って，首の絶対方向を求める
2. 1 の首方向と観測されたフラッグの情報を使って，絶対位置座標を求める．

一般的な位置測定では三角推量を利用しますが，rcssserver における位置測定はこのような少し特殊な方法を用います．三角推量では首の絶対方向にも距離の量子化誤差が混入してしまいますが，この方法であれば首の絶対方向の誤差は整数への丸め誤差だけとなります．

首の絶対方向

細かいことは考えず，rcssserver 内部で行われている計算の逆関数を作ります．rcssserver 内部で行われる処理については，visual.cc 内の VisualSenderPlayerV1::sendLine() という関数を参照して下さい．以下に，首の絶対方向 $face\_dir$ の計算手順を示します．

1. 観測されたラインの数が 0(フィールドの外に出て，外側に向いている) の場合，首の方向を決定できないため終了.
2. 観測されたラインの数が 2(フィールドの外に出て，内側に向いている) の場合，観測された距離の順にソートする．以降，近い方のラインの情報を使用する.
3. $face\_dir$ = 観測されたラインの方向
4. - face_dir が正の場合: $face\_dir- = 90$
   - face_dir が負の場合: $face\_dir+ = 90$
   - face_dir == 0(首の方向とラインが並行) の状態にはならないので無視
5. - 左のラインの場合: $face\_dir = 180 - face\_dir$
   - 右のラインの場合: $face\_dir = 0 - face\_dir$
   - 上のラインの場合: $face\_dir = -90 - face\_dir$
   - 下のラインの場合: $face\_dir = 90 - face\_dir$
6. 観測されたラインの数が 2 の場合， $face\_dir+ = 180$
7. $face\_dir$ を $[-180, 180]$ に正規化.

プレイヤエージェントがフィールドのラインの外側にいる場合，観測されるラインの数は 2 になります．このうち，近い方のラインは通常とは逆方向から見た状態になります．よって，通常どおりの角度を求めた後にその値を 180 度反転させれば正しい値が得られます．距離順にソートするのは，外側から見た状態になっているラインを特定するためです．近い方のラインを使うのはただの慣習で，遠い方を使っても問題はありません．その場合，角度の反転処理は不要になります.

librcsc では，LocalizeImpl クラスの getFaceDirByLines() メンバ関数が上記処理に相当します.

体の絶対方向

sense_body にはプレイヤエージェントの体の絶対方向に対する首の相対角度の情報 $sensed\_neck\_dir$ が含まれています．よって，首の絶対方向 $face\_dir$ が判明していれば，体の絶対方向 $body\_dir$ は以下のように求められます.

$$body\_dir = face\_dir + sensed\_neck\_dir$$

絶対位置

首の絶対方向が定まり，名前の分かるフラッグがひとつ以上見えていれば，プレイヤエージェント自身の絶対位置の推定が可能になります．計算手順は以下のようになります．$face\_dir$ は首の絶対方向，$seen\_flag\_dir$ は観測されたフラッグの方向，$seen\_flag\_dist$ は観測されたフラッグの距離とします．

1. プレイヤエージェントの位置からフラッグの位置への絶対方向 $flag\_dir$ を求める．
   $flag\_dir = face\_dir + seen\_flag\_dir$

2. フラッグの相対位置座標 rel_pos を求める．
   $rel\_pos.x = seen\_flag\_dist \times \cos(flsg\_dir \times \pi/180)$
   $rel\_pos.y = seen\_flag\_dist \times \sin(flsg\_dir \times \pi/180)$

3. フラッグの絶対位置座標 $flag\_pos$ は既知なので，2 の結果より，プレイヤエージェント自身の絶対位置座標 my_pos が求められる．
   $my\_pos = flag\_pos - rel\_pos$

以上が基本的な位置座標の計算手順となります．しかし，このまま単一のフラッグのみで位置測定を終了しては，得られる結果はフラッグの観測誤差の影響を強く受けてしまいます．フラッグの方向の誤差は首の絶対方向の誤差と合わせて常に ± 1 度 ですが，距離の誤差と方向の誤差を比較すると，実際には方向の誤差の方が非常に大きくなります．例えば，$100m$ 離れたフラッグの距離の誤差は $\pm 0.5m$ 程度であるのに対し，方向による誤差は $\pm 1.7m$ 以上もあります．より精度の高い位置測定を行うには，複数のフラッグ情報を利用することが必要となります．

6.5.5 位置の絞り込み

複数のフラッグが観測されていれば，それらの情報を使って更に位置の絞り込みが行えます．具体的なイメージは図 6.10 のようになります．

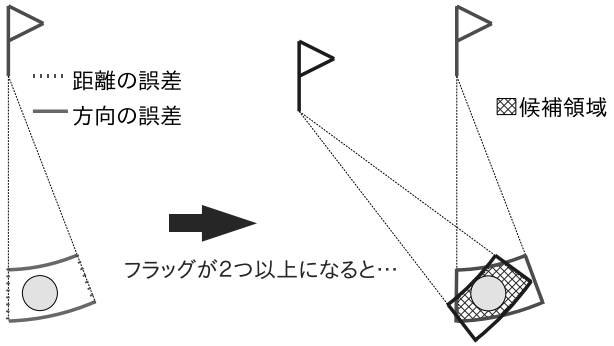


図 6.10 複数のフラッグによる位置の絞り込み

観測されたフラッグに対して方向と距離の誤差の範囲が確定しているため，プレイヤエージェント自身の位置の候補となる扇型の領域が得られます．複数のフラッグが見えていれば，この領域を重ね合わせていくことができ，残った領域がプレイヤエージェントの位置の候補となります．

位置の絞り込みにおいて，一見簡単そうで難しいのが領域の重ね合わせです．純粋に計算幾何の問題として解く [7]，パーティクルフィルターを利用する [5][15]，などのいくつかの方法が考案されてきましたが，librcsc では以下のようなもっと単純な方法を採用しています．

1. 最近傍のフラッグによって得られた初期候補領域を一定数のグリッドに分割．
2. 各グリッドが残りのフラッグによる候補領域に含まれるどうかを調べ，
   (a) 含まれていれば，そのまま残す．
   (b) 含まれていなければ，削除．
3. 最終的に残ったグリッドの平均位置を求める．

考え方としてはパーティクルフィルターと似たようなものですが，パーティクルフィルターよりもはるかに高速に動作し，充分な精度も得られます．

6.5.6 速度の更新

プレイヤエージェント自身の速度の求め方は簡単です．**sense_body** には自身のスピードの絶対値が含まれており，この値には意図的な誤差が含まれていませ

ん．なおかつ，精度も 0.01 とかなり正確な値が得られます．更に，**sense_body** によって，首の絶対方向 $face\_dir$ に対するスピードの相対方向 $sensed\_vel\_dir$ をも得られます．この方向の角度の値は整数に丸められているため，誤差の範囲は ± 0.5 度 です．ただし，実数値から int 型への単純なキャストであるため，**see** の場合とは期待値の求め方が異なります．

以上より，速度の絶対方向 vel_dir は

$$vel\_dir = face\_dir + sensed\_vel\_dir$$

となり，速度ベクトルは簡単に求められます．

この速度ベクトルの方向はプレイヤエージェントの体の絶対方向とは一致しないことに注意してください．

6.5.7 他の移動物体の位置，速度測定

位置測定

ボールやプレイヤといった他の移動物体の位置測定の計算は簡単です．**see** に含まれる距離 $dist$ と方向 dir の情報，そしてプレイヤエージェント自身の首の絶対方向 $face\_dir$ が判明していれば，観測物体の相対位置座標 rel_pos が求められます．後は，プレイヤエージェント自身の絶対位置座標 $self\_pos$ を加えれば，最終的な絶対位置座標 obj_pos が以下のように得られます．

$$rel\_pos.x = dist \times \cos((dir + face\_dir) \times \pi/180)$$

$$rel\_pos.y = dist \times \sin((dir + face\_dir) \times \pi/180)$$

$$obj\_pos = self\_pos + rel\_pos$$

速度測定

他の移動物体の速度の計算はやや複雑になります．基本的には，**see** で得られる情報から相対速度を求められ，それにプレイヤエージェント自身の絶対速度を加えることで，観測物体の絶対速度が得られます．

まずは，物体の相対速度を求めます．**see** に含まれる速度の情報は $dist\_change$ と dir_change です．これらは，観測物体とプレイヤエージェントの相対位置ベクトル方向が軸となるように，物体の相対速度ベクトルを射影したベクトル成分を意味しています．具体例を図 6.11 に示します．

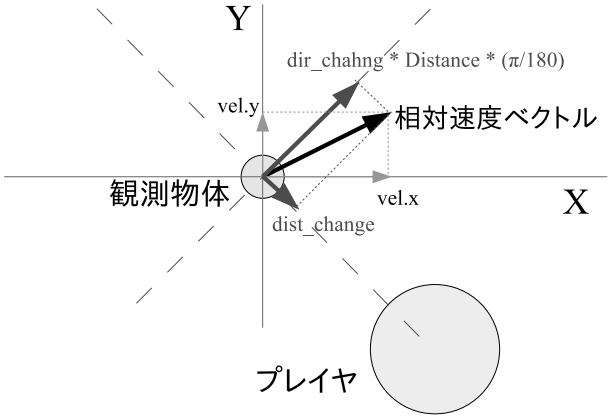


図 6.11 dist_change と dir_change

図中の $vel.x$ と $vel.y$ は，相対速度を通常の X 軸 Y 軸へ分解された成分です．一方，$dist\_change$ と $dir\_change \times distance \times (\pi/180)$ も相対速度ベクトルを分解した成分であるものの，軸の方向が回転しています．$dist\_change$ の軸の方向はプレイヤエージェントから観測物体への絶対方向，dir_change の軸はそれと直角に交わります．dir_change は角度の変化を表したものではない点に注意してください．rcssserver のソースでは，visua.cc 内の VisualSenderPlayerV1::calcVel() という関数がこれらの値を計算しています．

図を見ても分かるように，$dist\_change$ と dir_change によって構成されるベクトルと，通常の XY 成分によって構成されるベクトルとの違いは，軸が回転していることだけです．よって，$dist\_change$ と dir_change によって構成されるベクトルを適切な角度で回転させれば，通常の座標軸での相対速度ベクトル rel_vel が得られます．この回転角度として，プレイヤエージェントから観測物体への絶対方向 obj_dir が使用されます．実際の計算は以下のようになります．

$$rel\_vel.x = dist\_change$$

$$rel\_vel.y = dir\_change \times distance \times (\pi/180)$$

$$rel\_speed = \sqrt{rel\_vel.x^2 + rel\_vel.y^2}$$

$$roteted\_angel = \arctan(rel\_vel.y/rel\_vel.x) + obj\_dir \times (\pi/180)$$

$$rel\_vel.x = rel\_speed \times \cos(rotated\_angle)$$

$$rel\_vel.y = rel\_speed \times \sin(rotated\_angle)$$

$dist\_change$ と dir_change の値の計算には量子化された距離の値が使用されるため，誤差の範囲もそれに依存します．この誤差の範囲を逆算することも可能ですが，実際の利用における有用性は低いため，本書では省略します．librcsc においても，誤差を一応求めてはいますが，実際には利用してはいません．物体の速度をより正確に得るには，現在の **see** による観測値だけでなく，過去の測定結果に基づく予測値との加重平均を取る方が効果があるようです．

6.5.8 位置変化量に基づくボールの速度測定

see に速度情報が含まれるのは，観測物体がプレイヤの視界に含まれる場合のみです．一方，位置情報に関しては，視界に含まれていなくとも $3m$(**server_param** の ***visible_distance***) 以内であれば距離と方向を “感じる” ことができます．そのため，至近距離にある物体は，その存在は知ることができるが速度を得られないという状況がしばしば発生します．例えば，周囲を見渡しながらドリブルする場合などはボールがプレイヤエージェントの視界の外に置かれる状況が頻発します．ボールの速度が得られなければ，プレイヤエージェントのボールコントロールの精度が大きく低下してしまいます．

この問題を解決するために，物体の位置の変化量から速度を求める方法が考案されました[9)]．位置の変化から速度を求める最も単純な計算式は以下のようになります．

$$current\_vel = (current\_pos - previous\_pos) \times decay$$

$current\_vel$ は結果として得られる現在の速度ベクトル，$current\_pos$ は現在の観測絶対位置座標，$previous\_pos$ は 1 サイクル前の観測絶対位置座標，$decay$ はその物体の速度減衰率パラメータです．理論的にはこれだけです．しかし，位置情報として絶対位置座標を用いると，対象物体の観測誤差だけで無く，プレイヤエージェント自身の位置測定の誤差も $current\_vel$ に混入されてしまいます．そこで，自身の誤差が混入しないよう，相対位置の変化量と自身の移動ベクトル $self\_move$ を用いるように式を変形します．

$$current\_vel = (current\_rel\_pos - previous\_rel\_pos + self\_move) \times decay$$

9) むしろ，“速度が見える” 方がおかしいのであって，本来ならば，速度は位置の変化を観測することによってのみ得られるべきです．

current_rel_pos と *previous_rel_pos* は，それぞれプレイヤエージェントに対する相対位置座標です．相対位置座標を扱うことによって，推定される速度の精度ははるかに高まります．上式の内容を図で表すと図 6.12 のようになります．

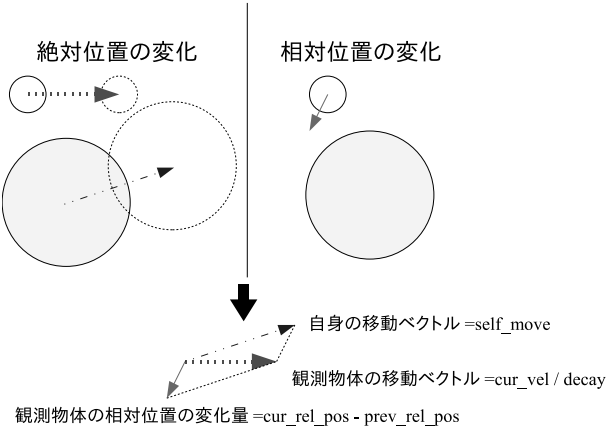


図 6.12 相対位置の変化量を用いた速度測定

この方法の欠点として，プレイヤエージェントに衝突が発生した場合に得られる速度は全く信用できないということが上げられます．また，この方法で速度を求めるには，対象となる物体の 1 サイクル前の位置情報を得られていることが前提となります．よって，**see** の同期を実現しておくか，最低限視界モードを (narrow,high) にしておく必要があります．

6.5.9 まとめ

位置測定関連の処理は，ObjectTable クラスと Localization クラスに集約されています．興味があれば，または，より良い位置測定手法の導入を検討する場合はこれらのクラスの実装を参照し，改変してください．

rcsssserver で動作するプレイヤエージェントのための位置測定手法はやや特殊です．本節で解説した位置測定手法は rcssserver の仕様を逆手に取った逆算モデルであって，実世界での応用には向かないであろう点に注意してください．

Section 6.6
世界モデル更新

センサ情報に基づいて位置や速度を測定するだけでは，フィールドの状態をプレイヤエージェントが充分に認識できたとは言えません，個々のセンサ情報を正しく解析するだけでなく，それらを適切に組み合わせ適用しなければなりません．解析されたセンサ情報は WorldModel クラスへの入力となり，WorldModel クラス内部ではそれらを使って状態の更新と管理を行います．

6.6.1 センサ情報受信後の更新手順

librcsc が提供するプレイヤエージェントは，センサ情報受信直後にその情報のタイプに応じた更新処理を即座に実行し，WorldModel へと反映します．プレイヤエージェントがセンサ情報を受信すると，以下の手順で WorldModel の更新処理を実行します．

1. センサ情報に含まれる時間情報を基に，現在のサイクルを更新．
2. sense_body の場合:
   (a) PlayerAgent::analyzeSenseBody() が起点となる．
   (b) コマンドの実行回数を確認し，1 サイクル前に実行したコマンドが機能したかどうかを確認する．もし失敗していれば，記憶しておいたコマンドの作用を消去する．(ActionEffector::checkCommandCount())
   (c) WorldModel の更新関数を呼び出す．(WorldModel::updateAfterSense())
   (d) プレイヤエージェント自身の状態を，1 サイクル前に実行したコマンドの内容に基づいて予測する．(SelfObject::updateAfterSense())
   (e) 内部に持つ情報のみを用いて各物体の位置や速度を予測し，信頼性の情報を更新する．(WorldModel::update()，各オブジェクトクラスの update())
   (f) プレイヤエージェント自身のスピードをチェックして衝突を予測する．(WorldModel::updateCollision())

3. see の場合:

   (a) PlayerAgent::analyzeSee() が起点となる.

   (b) WorldModel の更新関数を呼び出す. (WorldModel::updateAfterSee())

   (c) 内部状態の更新がまだであれば実行する. (WorldModel::update())

   (d) 各物体の位置測定を行う. (WorldModel::localizeSelf(), localizeBall(), localizePlayers())
   同時にプレイヤのマッチングを行う. (WorldModel::checkTeamPlayer(), WorldModel::checkUnknownPlayer())

   (e) ゴーストオブジェクトをチェックする. (WorldModel::checkGhost())

   (f) 方向の信頼性の情報を更新する. (WorldModel::updateDirCount())

4. hear の場合:

   (a) 審判からの情報はただちに適用する. (WorldModel::updatePlayMode())

   (b) 味方とのコミュニケーションで得られた情報は一旦保管しておき, 後で利用する.

更に, 意思決定の直前に WorldModel::updateJustBeforeDecision() を呼び出すことで, 以下のような推定や予測を行います.

- そのサイクルのセンサ情報をまだ一度も受信していないなら, WorldModel::update() を呼び出す.
- hear によって得られた情報を適用.
- プレイヤエージェントから近い順にソートされた他のプレイヤの配列を作成.
- ボールから近い順にソートされた他のプレイヤの配列を作成.
- オフサイドラインの推定
- ボール所有者の推定. プレイヤエージェント自身を含めてもっとも早くボールを補足できるプレイヤを予測.

意思決定直前の推定や予測は, 参照のたびにループを回して余計な計算時間の浪費することの無いように, プレイヤエージェントの意思決定中に頻繁にアクセスする情報をあらかじめ用意しておくことが目的です. 特に, ボール所有者の推定は計算量が多いため, あらかじめ計算しておき結果を保持しておくべきです.

6.6.2 プレイヤのマッチング

6.5 節で述べたように，プレイヤエージェントが **see** によって得られる物体の情報の誤差は，距離が遠くなればなるほどより大きくなります．これに加えて，観測物体がプレイヤの場合は，対象プレイヤとの距離が離れるほどその識別情報が得られにくくなります．

具体的には，次のような仕様になっています．対象プレイヤとの距離を $dist$ とし，対象プレイヤはプレイヤエージェントの視界に入っているものとすると，

- $dist \leq 20m$ であれば，背番号とチーム名の両方を常に得られる．
- $20 < dist \leq 40$ であれば，チーム名は常に得られる．しかし，$dist$ が増加するに従い，背番号が得られる確率が 1 から 0 へ線形に減少する．
- $40 < dist$ になると，背番号はもはや得られない．
- $40 < dist \leq 60$ であれば，チーム名が得られる確率が 1 から 0 へ線形に減少する．
- $60 < dist$ になると，チーム名はもはや得られない．

現在の **see** のみからプレイヤを完全に識別することは不可能です．しかし，これまでに観測された情報と照らし合わせることで，背番号やチーム名を得られなかったプレイヤを識別できる可能性があります．このプレイヤマッチングは次のような手順で行います．

1. プレイヤの背番号が見えていた場合：
   (a) 過去に観測されたプレイヤの中に背番号の一致するものがあれば，その情報を更新して終了．
   (b) 過去に観測された，背番号が見えていないがチームが同じ，または全く識別できていないプレイヤの中から，現在観測したプレイヤに最も近いものを探す．
   (c) 過去の観測位置から到達可能な距離であれば，現在の観測情報を用いて情報を更新して終了．
   (d) 到達不可能な距離であれば，新規プレイヤとして登録して終了．
2. プレイヤのチーム名が見えていた場合：

(a) 過去に観測された，チームが同じ，または全く識別できていないプレイヤの中から，現在観測したプレイヤに最も近いものを探す．

(b) 過去の観測位置から到達可能な距離であれば，現在の観測情報を用いて情報を更新して終了．

(c) 到達不可能な距離であれば，新規プレイヤとして登録して終了．

3. 背番号もチーム名も見えなかった場合：

(a) 過去に観測された全てのプレイヤの中から，現在観測したプレイヤに最も近いものを探す

(b) 過去の観測位置から到達可能な距離であれば，現在の観測情報を用いて情報を更新して終了．

(c) 到達不可能な距離であれば，新規プレイヤとして登録して終了．

WorldModel クラスの checkTeamPlayer() と checkUnknownPlayer() というメンバ関数が上記の処理に該当します．

プレイヤのマッチングは解決が難しく，最も良い手法と呼べるものがありません．上記のアルゴリズムもそれなりの精度ではあるものの，誤った識別をしてしまうこともしばしばです．このプレイヤのマッチング問題は実世界においても応用が利く問題であるため，なかなか興味深いテーマと言えるでしょう．

6.6.3 ゴーストオブジェクト

例えば，一瞬目を離した隙に他のプレイヤがボールをどこかに蹴ってしまった場合を考えてみましょう．人間であれば，ボールが無くなったことを認識し，ボールを探そうとするでしょう．では，同じことをプログラムのプレイヤエージェントに行わせるとどうなるでしょうか？何の工夫もしていなければ，存在しないボールを追いかけたり蹴ろうとしたりするかもしれません．これが所謂ゴーストオブジェクトです．

人間の場合，過去に観測から予測されるボール位置と実際の観測結果が異なることからボールを見失ったと推論を行っていると考えられます．プレイヤエージェントにも，現在の視界に含まれるはずだったが含まれていない物体をゴーストオブジェクトとして認識する能力が必要となります．

これを実装するのは難しくありません．以下の条件を満たすかどうかを確認するだけです．

- 対象物体を 1 サイクル以上観測していない.
- 対象物体の予測位置が現在の視界の範囲内に存在する.

上記の処理は WorldModel::checkGhost() という関数が担当しています．ただし，この関数では，ゴーストオブジェクトを確認するのはボールのみとしています．

プレイヤのゴーストオブジェクトを確認しないのは，ボールと比べてプレイヤは急激な移動をすることが少ないためです．プレイヤを一瞬見失ったからといって，その一瞬で見失ったプレイヤがはるか遠くへ移動しているわけではありません．“そこにはいないがその辺りにはいる” という状態を認識していなければ，思わぬピンチを招くことがあります．しかしながら，やはりそこにはいないので，曖昧な情報を上手く扱う手法が必要となります．残念ながら，筆者はそのような手法の確立にはまだ至っていません．

一瞬見失った場合の情報更新は以上のように実装されていますが，長期間観測されなかった場合はボールとプレイヤのいずれもメモリから強制的に消去されます．デフォルト設定ではボールは 20 サイクルまで，他のプレイヤは 30 サイクルまで情報を保持するようになっています．これらの消去処理は WorldModel::update() で行われています.

6.6.4 衝突の検出

速度の測定と関連しますが，プレイヤエージェントと他の物体との衝突，特にボールとの衝突の検出精度は，ボールコントールの精度に直結するため非常に重要です．衝突を 100% 検出することは不可能ですが，ある程度推定することは可能です．推定方法として最も精度が高いのは，プレイヤエージェント自身の速度の減衰量を観測する方法でしょう．プレイヤエージェントの速度減衰率は，0.4 から 0.6 となっています．一方，物体動詞の衝突が発生すると，速度は強制的に −0.1 倍されます．すなわち，以下の条件を満たせば，衝突が発生した可能性が高いと判断できます.

- プレイヤエージェント自身の観測速度と事前の予測速度の符号が逆
- 観測速度の大きさが予測値の 0.1 倍に近い.
- 密着する程度の至近距離に物体が存在する

上記の処理は SelfObject::updateAfterSense() や，WorldModel::updateCollision() で実装されています.

6.6.5 ボール所有者の推定

ボール所有者の推定は，もっとも早くボールを捕捉できるプレイヤを推定することと同値です．ボールを補足するとは，インターセプト行動に他なりません．よって，インターセプトの予測をプレイヤエージェント自身だけでなく，他のプレイヤにも同様に適用すれば，ボール所有者の推定が可能となります．

しかし，他のプレイヤの位置や速度，体の向きの情報は，プレイヤエージェント自身のものと比べてはるかに精度が劣ります．そのため，基本的な計算手順はプレイヤエージェントの場合と同じであるものの，その内容はかなり簡略化されています．この実装は，PlayerIntercept というクラスでなされています．

6.6.6 まとめ

より良い世界モデルの構築，特に他のプレイヤのマッチングやゴーストオブジェクトとなっているプレイヤの取り扱いは，今後も解決が難しい問題となるでしょう．厳密に計算しようとすれば計算量が多くなるにも関わらず，計算資源を浪費した割には大した効果がでないということも往々にしてあります．

プレイヤエージェントの世界モデルの管理は，WorldModel クラスがほぼ一手に担っています．世界モデル改善のアイデアが思い付いた場合は，このクラスを参照し，改変を試みてください．

関連図書

[1] Glenn Seemann (原著)David M. Bourg (原著), クイープ (翻訳). ゲーム開発者のための AI 入門. オライリージャパン, 2005.

[2] Doxygen:
http://www.doxygen.org/.

[3] Richard Helm Erich Gamma, Ralph Johnson, John Vlissides, 本位田 真一 (翻訳), 吉田 和樹 (翻訳). オブジェクト指向における再利用のためのデザインパターン. ソフトバンククリエイティブ, 1999.

[4] Fc Portugal:
http://www.ieeta.pt/robocup/.

[5] Nikos Vlassis Jelle R. Kok, Remco de Boer and F.C.A. Groen. Uva trilearn 2002 team description. In *Robot Soccer World Cup VI*, p. 549. Springer-Verlag, 2002.

[6] Jan Lubbers and Rogier R. Spaans. The priority/confidence model as a framework for soccer agents. *RoboCup 1998: Robot Soccer World Cup II*, pp. 162–172, 1999.

[7] Puppets Source:
http://www.i.his.fukui-u.ac.jp/~shimora/robocup/.

[8] Ross J. Quinlan. C4.5: Programs for machine learning. 1992.

[9] Luis Paulo Reis, Nuno Lau, and Eugenio Olivéira. Situation based strategic positioning for coordinating a simulated robosoccer team. *Balancing Reactivity and Social Deliberation in MAS*, pp. 175–197, 2001.

[10] 三上 貞芳 Richard S.Sutton (著), 皆川雅章. 強化学習. 森北出版, 2000.

[11] Peter Stone and David McAllester. An architecture for action selection in robotic soccer. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pp. 316–323, 2001.

[12] Peter Stone and Richard S. Sutton. Keepaway soccer: a machine learning testbed. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup-2001: Robot Soccer World Cup V*. Springer Verlag, Berlin, 2002.

[13] Subversion:
http://subversion.tigris.org/.

[14] Subversion によるバージョン管理:
http://subversion.bluegate.org/.

[15] UvA Trilearn 2003 Base Source:
http://www.science.uva.nl/~jellekok/robocup/2003/.

[16] 多摩ソフトウェア (翻訳) ハーバート シルト (著), エピステーメー (監修). STL 標準講座 標準テンプレートライブラリを利用した C++プログラミング. 翔泳社, 1999.

[17] 浜田 真理 (翻訳) ハーブ サッター (著), 浜田光之. Exceptional C++ 47 のクイズ形式によるプログラム問題と解法 C++ In Depth Series. ピアソンエデュケーション, 2000.

[18] バージョン管理システム cvs を使う:
http://radiofly.to/nishi/cvs/.

[19] Barbara E. Moo (原著) バーバラ・E. ムー (著), 小林 健一郎 (翻訳). Accelerated C++ 効率的なプログラミングのための新しい定跡 C++ In Depth Series. ピアソンエデュケーション, 2001.

[20] W. リチャード・スティーブンス. UNIX ネットワークプログラミング 第 2 版, 第 1 巻. 株式会社ピアソン・エデュケーション, 1999. 篠田陽一 訳.

[21] 稲葉一浩. Boost C++ Library プログラミング. 秀和システム, 2004.

[22] Scott Meyers(著), 吉川 邦夫 (翻訳). Effective C++ アスキーアジソンウェスレイシリーズ. アスキー, 1998.

[23] Scott Meyers(著), 細谷 昭 (翻訳). Effective STL STL を効果的に使いこなす 50 の鉄則. ピアソンエデュケーション, 2002.

付 録A
サッカーシミュレータの設定

Section A.1
プレイモード一覧

プレイモード文字列	説明
before_kick_off	試合のハーフ開始前の待機状態．auto_mode が無効の場合，人間がモニタを操作してキックオフするまで試合は始まらない．
time_over	試合終了後．
play_on	通常の試合進行時．
kick_off_l	左チームのキックオフ．
kick_off_r	右チームのキックオフ．
kick_in_l	左チームのキックイン．
kick_in_r	右チームのキックイン．
free_kick_l	左チームのフリーキック．
free_kick_r	右チームのフリーキック．
corner_kick_l	左チームのコーナーキック．
corner_kick_r	右チームのコーナーキック．
goal_kick_l	左チームのゴールキック．
goal_kick_r	右チームのゴールキック．
goal_l_?	左チームのゴール直後の状態．シミュレーションサイクルが 5 秒間止まる． 文字列末尾に総得点が付加される．
goal_r_?	右チームのゴール直後の状態．以下，同上．
drop_ball	即座に play_on に変わる．
offside_l	左チームのオフサイド違反．シミュレーションサイクルが一定時間止まり，その後 free_kick_r になる．
offside_r	右チームのオフサイド違反．シミュレーションサイクルが止まり，続いて free_kick_l になる
penalty_kick_l	使用されない．
penalty_kick_r	使用されない．
first_half_over	使用されない．

プレイモード文字列	説明
pause	使用されない.
human_judge	使用されない.
foul_charge_l	使用されない.
foul_charge_r	使用されない.
foul_push_l	使用されない.
foul_push_r	使用されない.
foul_multiple_attack_l	使用されない.
foul_multiple_attack_r	使用されない.
foul_ballout_l	使用されない.
foul_ballout_r	使用されない.
back_pass_l	左チームによるキーパへのバックパス違反. シミュレーションサイクルが一定時間止まり，その後 free_kick_r になる． ボールはペナルティエリアのコーナーに置かれる.
back_pass_r	右チームによるキーパへのバックパス違反. シミュレーションサ イクルが一定時間止まり，その後 ree_kick_l になる． 以下，同上.
free_kick_fault_l	左チームのフリーキック違反. シミュレーションサイクルが止まり，続いて free_kick_r になる.
free_kick_fault_r	右チームのフリーキック違反. 続いて free_kick_l になる
catch_fault_l	左チームのキーパのキャッチ違反. シミュレーションサイクルが一定時間止まり，その後右チームに間接フリーキックが与えられる.
catch_fault_r	右チームのキーパのキャッチ違反. シミュレーションサイクルが一定時間止まり，その後左チームに間接フリーキックが与えられる.
indirect_free_kick_l	左チームの間接フリーキック． キッカが直接ゴールを決めても得点にならない.
indirect_free_kick_r	右チームの間接フリーキック. 以下，同上.
penalty_setup_l	左チームの PK のための移動時間. このモードの間にキッカー以外のプレイヤはセンターサークル内に移動しなければならない.
penalty_setup_r	右チームの PK のための移動時間. 以下，同上.
penalty_ready_l	PK で左チームのキッカーの準備時間. このモードが終わるまでにボールを蹴り始めなければならない.
penalty_ready_r	PK で右チームのキッカーの準備時間. 以下，同上.
penalty_taken_l	左チームキッカーが PK トライ中. このモードが終わるまでんにゴールが決められなければ，自動的に PD 失敗となる.
penalty_taken_r	キッカーが PK トライ中． 以下，同上.
penalty_miss_l	左チームが PK に失敗した直後.
penalty_miss_r	右チームが PK に失敗した直後.
penalty_score_l	左チームが PK に成功した直後.
penalty_score_r	右チームが PK に成功した直後.

Section A.2
rcssserver のパラメーター一覧

A.2.1 server_param

server_param パラメータの設定ファイルは~.rcssserver/server.conf です．この内容を編集するか，起動時のコマンドラインオプションで指定することで rcssserver の設定を変更することができます．

使用例:

```
$ rcssserver server::game_logging = true \
server::game_log_dir = '~/log' server::game_log_fixed = true
```

パラメータ名 説明	デフォルト値
audio_cut_dist プレイヤの say メッセージが届く距離．	50
auto_mode true の場合，rcssserver は試合をすべて自動実行する．このモードを使用するときは ***team_l_start*** と ***team_r_start*** が設定されていなければならない．	false
back_passes true の場合，審判がバックパス違反を取るようになる．	true
ball_accel_max ボールの加速度の大きさの最大値．	2.7
ball_decay ボールの速度減衰率．	0.94
ball_rand ボールの動きに加わるノイズの割合．	0.05
ball_size ボールの半径．	0.085
ball_speed_max ボールのスピードの最大値．	2.7
ball_weight ボールの重さ．風の影響に関係するが，現在のルールでは使用されない．	0.2
catch_ban_cycle キーパがボールをキャッチすると，このサイクル経過するまでは再度 **catch** コマンドを実行しても無効になる．	5
catch_probability **catch** コマンドの成功確率．	1
catchable_area_l キャッチ可能な矩形領域の長さ．	2

catchable_area_w	1
キャッチ可能な矩形領域の幅.	
ckick_margin	1
コーナーキック時のボール配置マージン.	
clang_advice_win	1
clang_win_size 期間内に **advice** メッセージを送信できる回数.	
clang_define_win	1
clang_win_size 期間内に **define** メッセージを送信できる回数.	
clang_del_win	1
clang_win_size 期間内に **del** メッセージを送信できる回数.	
clang_info_win	1
clang_win_size 期間内に **info** メッセージを送信できる回数.	
clang_mess_delay	50
プレイモードが 'play_on' の時，CLang メッセージをコーチが送信してからこのサイクル経過した後に，プレイヤへと実際に送信される．プレイモードが 'play_on' 以外の時，CLang は直ちにプレイヤへ送信される.	
clang_mess_per_cycle	1
1 サイクルに送信される最大 CLang メッセージ数．キューに CLang メッセージがたまっていると，サイクルごとにこの個数ずつプレイヤへ送信される.	
clang_meta_win	1
clang_win_size 期間内に **meta** メッセージを送信できる回数.	
clang_rule_win	1
clang_win_size 期間内に **rule** メッセージを送信できる回数.	
clang_win_size	300
コーチが CLang を送信できる間隔．正確には，このサイクルが経過するごとに **freeform** 以外の各メッセージタイプの残り使用回数が初期化される.	
coach	false
トレーナを使用できるようにするスイッチ．`false` の場合，トレーナは rcssserver へ接続できない.	
coach_port	6001
トレーナの初期接続ポート.	
coach_w_referee	false
トレーナと審判を同時に使用できるようにするスイッチ．`false` にすると，ボールの位置によるプレイモードの判定もトレーナが行わなければならない.	
connect_wait	300
auto_mode 有効時に，プレイヤの接続を待つ最大サイクル数．このサイクル経過後にプレイヤが 22 人接続されていない場合は強制的に試合が開始される.	
control_radius	2
使用されない.	
dash_power_rate	0.006
デフォルトプレイヤの **dash** コマンドのパワー効果率.	
drop_ball_time	200
フリーキック時にボールが蹴られ g ないままこのサイクルが経過すると，ボールは適切な位置に自動的にドロップされ，プレイモードは 'play_on' になる.	
effort_dec	0.005
stamina が ***effort_dec_thr*** を下回っているときの 1 サイクルあたりの $effort$ の減少量.	
effort_dec_thr	0.3

effort が減少する *stamina* 量の閾値率.

effort_inc 0.01
stamina が ***effort_inc_thr*** を上回っているときの 1 サイクルあたりの *effort* の増加量.

effort_inc_thr 0.6
effort が増加する *stamina* 量の閾値率.

effort_init 1
デフォルトプレイヤの *effort* の初期値.

effort_min 0.6
デフォルトプレイヤの *effort* の最小値.

forbid_kick_off_offside true
`false` の場合，プレイモードが ‘`before_kick_off`’ か ‘`kick_off`’ の時にプレイヤが敵陣に侵入することが許可される．実験でプレイヤの初期配置を自由に設定したいときなどに使用する.

free_kick_faults true
`true` の場合，審判はフリーキック違反を取るようになる.

freeform_send_period 20
プレイモードが ‘`play_on`’ の時に **freeform** メッセージを送信できる期間．***freeform_wait_period*** を参照.

freeform_wait_period 600
プレイモードが ‘`play_on`’ のときは，このサイクル経過しなければコーチは **freeform** メッセージを送信できない．経過サイクルはプレイモードが ‘`play_on`’ になってからカウントされる．‘`play_on`’ になってからこのサイクル経過後，***freeform_send_period*** の間だけ **freeform** メッセージの送信が許可される．例えば、420 サイクルにプレイモードが ‘`play_on`’ になり，そのまま ‘`play_on`’ が継続した場合，コーチが **freeform** メッセージを送信できるのは 1020〜1040 の間， 1620〜1640 の間…となる．プレイモードが ‘`play_on`’ 以外のときは，この制限は働かない.

fullstate_l false
`true` の場合，左チームのプレイヤへ **fullstate** メッセージが送信される.

fullstate_r false
`true` の場合，右チームのプレイヤへ **fullstate** メッセージが送信される.

game_log_compression 0
1 以上でゲームログ (`.rcg`) ファイルを gzip 圧縮する．0–9 を指定可能で，数値は圧縮レベルを表す.

game_log_dated true
`true` の場合，ゲームログ (`.rcg`) ファイル名に試合開始日時の文字列を挿入する.

game_log_dir “./”
ゲームログ (`.rcg`) ファイルを作成するディレクトリパス．ディレクトリは自動で作成されないので，事前にディレクトリを作成しておかなければならない.

game_log_fixed false
`true` の場合，ゲームログ (`.rcg`) ファイル名は ***game_log_fixed_name*** に固定される.

game_log_fixed_name rcssserver
game_log_fixed が `true` の場合，ゲームログ (`.rcg`) ファイル名は常にこの文字列になる.

game_log_version 3
ゲームログ (`.rcg`) のバージョン．1〜3 を指定可能.

game_logging true
`true` の場合，ゲームログ (`.rcg`) ファイルを新規作成して試合内容を記録する.

game_over_wait	100
auto_mode 有効時，通常ハーフの前後半が終了してからこのサイクルが経過すると，rcssserver から ***team_l_start*** と ***team_r_start*** の各プロセスに SIGINT メッセージが送られ，rcssserver のプロセスが自動終了する．クライアント側は SIGINT を受け付けてクライアントのプロセスを kill できるようにしておかなければならない．	
goal_width	14.02
ゴールの幅．	
goalie_max_moves	2
キーパがボールキャッチ後に **move** コマンドを実行できる回数．	
half_time	300
1 ハーフのサイクル数．rcssserver 内部では，この値を 10 倍した値が使われる．**server_param** メッセージによってサッカーエージェントへ送信されるのは 10 倍されていないオリジナルの値．負の値が設定されると， 試合は無限に続行される．	
hear_decay	1
hear メッセージで他のプレイヤのコミュニケーションメッセージを受信したときに聴覚センサキャパシティが減る量．	
hear_inc	1
1 サイクルあたりの聴覚センサキャパシティの回復量．	
hear_max	1
聴覚センサキャパシティの最大値．	
inertia_moment	5
デフォルトプレイヤの慣性モーメント．**turn** コマンドの効果に影響を及ぼす．	
keepaway	false
`true` の場合，rcssserver は keepaway モードで実行される．	
keepaway_length	20
keepaway モード時の keepaway 領域の長さ．	
keepaway_log_dated	true
true の場合，keepaway ログ (`.kwy`) ファイル名に試合開始日時が挿入される．	
keepaway_log_dir	“./”
keepaway ログ (`.kwy`) ファイルを作成するディレクトリパス．ディレクトリは自動で作成されないので，事前にディレクトリを作成しておかなければならない．	
keepaway_log_fixed	false
`true` の場合，keepaway ログ (`.kwy`) ファイル名は ***keepaway_log_fixed_name*** に固定される．	
keepaway_log_fixed_name	rcssserver
keepaway_log_fixed が `true` の場合，keepaway ログ (`.kwy`) ファイル名は常にこの文字列になる．	
keepaway_logging	true
`true` の場合，keepaway ログ (`.kwy`) ファイルを新規作成し，keepaway の実行内容を記録する．	
keepaway_start	-1
keepaway モードで，‘`play_on`’ 以外のプレイモードから自動的に ‘`play_on`’ になるまでの秒数。負の場合，プレイモードは自動変更されない．	
keepaway_width	20
keepaway モード時の keepaway 領域の幅．	
kick_off_wait	100
auto_mode 有効時に，プレイヤが 22 人接続されてから自動的にキックオフされるまでのサイクル数．	
kick_power_rate	0.027

kick コマンドのパワー効果率.

kick_rand	0
デフォルトプレイヤの **kick** コマンドと **tackle** コマンドに作用するノイズの割合.	
kick_rand_factor_l	1
team_actuator_noise が `true` の場合に左チームのプレイヤの ***kick_rand*** にかけられる増加率.	
kick_rand_factor_r	1
team_actuator_noise が `true` の場合に右チームのプレイヤの ***kick_rand*** にかけられる増加率.	
kickable_margin	0.7
デフォルトプレイヤのキック可能領域のマージン.	
landmark_file	" /.rcssserver-landmark.xml"
フィールド上の静止物体を定義するファイル. 使用されていない.	
log_date_format	"%Y%m%d%H%M-"
ログファイル名に使用される日時のフォーマット. rcssserver 内部では `strftime()` 関数によって日付文字列が生成されるため, 日時のフォーマットもそれに準ずる.	
log_times	false
`true` の場合, テキストログ (`.rcl`) ファイルにシミュレーションに要した実時間を記録する.	
max_goal_kicks	3
ゴールキックやり直しの最大回数. ***proper_goal_kicks*** が `false` の場合は使用されない.	
maxmoment	180
turn コマンドの最大モーメント.	
maxneckang	90
首の最大角度.	
maxneckmoment	180
turn_neck コマンドの最大モーメント.	
maxpower	100
各行動コマンドで使用できるパワーの最大値.	
minmoment	-180
turn コマンドの最小モーメント.	
minneckang	-90
首の最小角度.	
minneckmoment	-180
turn_neck コマンドの最小モーメント.	
minpower	-100
各行動コマンドで使用できるパラーの最小値.	
nr_extra_halfs	2
延長戦のハーフ数.	
nr_normal_halfs	2
通常のハーフ数. この数のハーフ終了時に得点差があれば試合は終了する. 前半のみで終了させたい場合はこの値を 1 にする. 前後半を行わず, ペナルティキックをすぐに行いたい場合は, ***nr_normal_halfs*** と ***nr_extra_halfs*** を 0 にする.	
offside_active_area_size	2.5
オフサイド判定のためのプレイヤとボールとの距離. プレイヤとボールとの距離がこの数値以上であれば, プレイヤがオフサイドの位置に存在してもオフサイドは取られない.	
offside_kick_margin	9.15

オフサイド後のフリーキック時に，守備側チームのプレイヤの位置が不適切な場合に引き戻される距離.

olcoach_port	6002
コーチの初期接続ポート.	
old_coach_hear	false
使用されていない.	
pen_allow_mult_kicks	true
codetrue の場合，ペナルティキックでキッカに複数回のキック (ドリブル) を許可する.	
pen_before_setup_wait	30
ペナルティキック準備のための移動時間用サイクル数．一人のキックが終了するたびにこのサイクル数分待機する.	
pen_coach_moves_players	true
`true` の場合，ペナルティキック時に位置を違反しているプレイヤは適切な位置へ自動配置される．`false` の場合，プレイヤは自力で適切な位置へ移動していなければならない．**move** コマンドでの移動はできない.	
pen_dist_x	42.5
ペナルティキック開始時にボールはゴールラインからこの距離だけ離れた位置に置かれる．Y 座標は常に 0 に固定される.	
pen_max_extra_kicks	10
pen_nr_kicks 回のペナルティキックで決着がつかなかった場合の最大追加回数.	
pen_max_goalie_dist_x	14
ペナルティキック時にキッカがボールを蹴るまでに守備側キーパがゴールラインから離れられる最大距離． Y 座標の制限はゴールの幅に固定される.	
pen_nr_kicks	5
1 チームあたりのペナルティキック回数.	
pen_random_winner	false
`true` の場合，ペナルティキックで勝敗が決まらなければランダムに勝者を決定する.	
pen_ready_wait	50
ペナルティキック時にキックを開始するまでにキッカに与えられるサイクル数．プレイモードが 'penalty_ready' になってからこのサイクル以内にキックを開始しなかった場合はファウルになる.	
pen_setup_wait	100
ペナルティキック時に，プレイヤが規定の位置へ移動するための猶予サイクル．***pen_coach_moves_players*** が `false` の場合，このサイクル以内に適切な位置へ移動していなかった場合はファウルになる.	
pen_taken_wait	200
ペナルティキック時にキック開始からこのサイクル経過すると自動的にキッカのミスと判断される.	
penalty_shoot_outs	true
`true` の場合，***nr_normal_halfs***, ***nr_extra_halfs*** のハーフ終了後に同点であればペナルティキックを行う．***nr_normal_halfs*** と ***nr_extra_halfs*** をいずれも 0 にしておくと，キックオフ直後にペナルティキックが開始される.	
player_accel_max	1
プレイヤの加速度の大きさの最大値．通常この数値まで加速度を得られることはあり得ないので，気にしなくて良い.	
player_decay	0.4
デフォルトプレイヤの速度減衰率.	
player_rand	0.1

プレイヤの移動に関するノイズの割合．速度更新と **turn** コマンド実行時の回転角度に影響する．

player_size	0.3
プレイヤの半径．	
player_speed_max	1.2
デフォルトプレイヤの最大スピード．ただし，最大パワーでダッシュし続けたとしても，デフォルトプレイヤではデフォルト値の 1.2 には決して到達できない．	
player_weight	60
プレイヤの重さ．風の影響に関係するが，現在のルールでは使用されない．	
point_to_ban	5
pointto コマンド実行後に次の **pointto** を実行できるまでの最小間隔．一回指さしを行うと最低限このサイクル数はその位置を強制的に指さし続ける．指さしを無効にする場合にもこの制限を受ける．	
point_to_duration	20
指さしが自動継続されるサイクル数．	
port	6000
プレイヤ及びモニタの初期接続ポート．	
prand_factor_l	1
team_actuator_noise が `true` の場合に左チームのプレイヤの ***player_rand*** にかけられる増加率．	
prand_factor_r	1
team_actuator_noise が `true` の場合に右チームのプレイヤの ***player_rand*** にかけられる増加率．	
profile	false
`true` の場合，テキストログ (`.rcl`) ファイルにプロファイル情報を記録する．特定の処理に要した時間などを記録する，rcssserver 自体のデバッグ用スイッチ．	
proper_goal_kicks	false
`true` の場合，ゴールキックから直接 '`play_on`' に戻らなければならないという制限が加えられる．その場合，ボールがペナルティエリアの外にデル前にキッカとは別のプレイヤがボールをキックしたり，ペナルティエリアの外に出る前にボールが止まってしまうと，ゴールキックのやり直しになる．キッカがドリブルの動作を行った場合は，先にフリーキック違反がチェックされる．***free_kick_faults*** が無効の場合は，ゴールキックのやり直しになる．	
quantize_step	0.1
プレイヤが受信する **see** メッセージ内の移動物体の距離に施される量子化のパラメータ．	
quantize_step_l	0.01
プレイヤが受信する **see** メッセージ内の静止物体の距離に施される量子化のパラメータ．	
record_messages	false
`true` の場合，ゲームログ (`.rcg`) ファイルに `MSG_MODE` 情報を記録する．`MSG_MODE` 情報とは，rcssserver の `types.h` 内で `msginfo_t` という名前で定義されているデータ型で，通常はテキストログに保存される情報と考えれば良い．	
recover_dec	0.002
スタミナが ***recover_dec_thr*** を下回っているときの 1 サイクルあたりの *recovery* の減少量．*recovery* が減少すると，次のハーフ開始まで増加しない．	
recover_dec_thr	0.3
recovery が減少するスタミナ量の閾値を決定する割合．	
recover_init	1

recovery の初期値.

recover_min	0.5
recovery の最小値.	
recv_step	10
rcssserver がクライアントからの通信を処理する最小間隔．単位はミリ秒.	
say_coach_cnt_max	128
コーチが **freeform** メッセージを送信できる最大回数．***nr_normal_halfs*** 回数のハーフ終了時，この値が新たに利用可能な回数として追加される．**freeform** を一度も使用していなければ，延長開始時の **feeeform** 送信の最大回数は ***say_coach_cnt_max***∗2 回になる.	
say_coach_msg_size	128
コーチが **freeform** メッセージで使用できる文字数.	
say_msg_size	10
プレイヤの **say** コマンドで使用できるメッセージの最大長.	
send_comms	false
`true` の場合，モニタに `MSG_MODE` 情報を送信する。rcssmonitor-classic はこのメッセージを表示させるための領域を持っており，各クライアントの送信コマンドの確認できるようになっている.	
send_step	150
視界モードが **(normal, high)** のプレイヤへ **see** を送信する間隔．単位はミリ秒.	
send_vi_step	100
コーチへ **see_global** を送信する間隔．通常，***simulator_step*** と同じ値．単位はミリ秒.	
sense_body_step	100
プレイヤへ **sense_body** を送信する間隔．通常，***simulator_step*** と同じ値．単位はミリ秒.	
simulator_step	100
1 サイクルの間隔．単位はミリ秒.	
slow_down_factor	1
この値を大きくすると，センサ情報の送信間隔を大きくし，rcssserver の見かけ上の実行速度を落とすことができる．スペックが低いマシンでシミュレーションを実行するときに利用できる．影響を受けるのは，***simulator_step***，***sense_body_step***，***send_vi_step***，***send_step***，***synch_offset***．それぞれの値に ***slow_down_factor*** の値が掛けられる．***slow_down_factor***= 2 とすると，シミュレーション実行速度は半分になる.	
slowness_on_top_for_left_team	1
左チームプレイヤがフィールドの上半分にいるときに，加速度がこの割合で減少させられる.	
slowness_on_top_for_right_team	1
右チームプレイヤがフィールドの上半分にいるときに，加速度がこの割合で減少させられる.	
stamina_inc_max	45
デフォルトプレイヤの 1 サイクルあたりのスタミナ回復量の最大値.	
stamina_max	4000
プレイヤのスタミナの最大値.	
start_goal_l	0
試合開始時の左チームの得点.	
start_goal_r	0
試合開始時の右チームの得点.	

stopped_ball_vel	0.01
ゴールキック時にボールのスピードがこの値以下であればボールが止まっている (ボールがゴールエリア角に置かれたゴールキックの初期状態) とみなされる．ボールが止まっている状態でゴールキックのやり直し回数の違反を犯すと相手チームへフリーキックが与えられる．そうでない場合は、ゴールキックのやり直しになる．***proper_goal_kicks*** が `false` の場合は，何のペナルティも発生しない．	
synch_micro_sleep	1
synch_mode が有効で rcssserver が同期モードで実行されるとき，rcssserver は `usleep()` によってこの時間ずつクライアントからのコマンド送信を待機する．単位はミリ秒．	
synch_mode	false
`true` の場合，rcssserver は同期モードで実行される．	
synch_offset	60
rcssserver が同期モードで実行されるとき，サイクル開始からクライアントへ **think** メッセージを送信するまでの仮想的な待ち時間．センサ情報の送信機会を通常実行時と同等にするために使用される．単位はミリ秒．	
tackle_back_dist	0.5
タックルの有効範囲パラメータ．体の後方のタックル可能距離．	
tackle_cycles	10
tackle コマンド実行後に再び動けるようになるまでのサイクル数．	
tackle_dist	2
タックルの有効範囲パラメータ．体の前方のタックル可能距離．	
tackle_exponent	6
tackle 成功確率に関するパラメータ．	
tackle_power_rate	0.027
tackle コマンドのパワー効果率．タックルが成功した場合のボールの加速度を決定する．	
tackle_width	1
タックルの有効範囲パラメータ．タックル可能な体の中心からの幅．	
team_actuator_noise	false
`true` の場合，チーム独立のノイズパラメータが有効になる．影響を受けるのは，***prand_factor_[lr]***，***kick_rand_factor_[lr]*** .	
team_l_start	“”
auto_mode 有効時，左チームを起動するコマンドラインを指定する．	
team_r_start	“”
auto_mode 有効時，右チームを起動するコマンドラインを指定する．	
text_log_compression	0
1 以上でテキストログ (`.rcl`) ファイルを gzip 圧縮する．0–9 を指定可能で，数値は圧縮レベルを表す．	
text_log_dated	true
`true` の場合，テキストログ (`.rcl`) ファイル名に日付を付ける．	
text_log_dir	“./”
テキストログ (`.rcl`) ファイルを作成するディレクトリパス．	
text_log_fixed	false
`true` の場合，テキストログ (`.rcl`) ファイル名は text_log_fixed_name に固定される．	
text_log_fixed_name	“rcssserver”
text_log_fixed が `true` の場合，テキストログ (`.rcl`) ファイル名は常にこの文字列になる．	
text_logging	true

	`true` の場合，テキストログ (`.rcl`) ファイルを新規作成して試合内容を記録する．
use_offside	true
	`true` の場合，オフサイドルールが有効になる．keepaway モード利用時はこれを `false` にしておく方が良い．
verbose	false
	`true` の場合，冗長なデバッグメッセージを出力する．rcssserver 自体のデバッグ用スイッチ．
visible_angle	90
	プレイヤの視野角が **normal** モードの場合の視野の中心角の大きさ．単位は度数．**narrow** モードになると視野角はこの半分，**wide** モードになると視野角はこの 2 倍になる．
visible_distance	3
	視界の外にある物体の存在を “感じられる” 距離の閾値．物体の存在を感じた場合は，その距離と方向の情報しか得られない．また，物体の種類しか分からず，物体の名前を特定することはできない．
wind_ang	0
	使用されない．
wind_dir	0
	win_random が `false` の時，風のベクトルの方向はこの角度になる．
wind_force	0
	win_random が `false` の時，風のベクトルの大きさはこの値になる．
wind_none	false
	`true` の場合，風の影響は完全に 0 になる．
wind_rand	0
	風のベクトルに加わるノイズの範囲を決定するパラメータ．
wind_random	false
	`true` の場合，風のベクトルはランダムに決定される．

A.2.2 player_param

player_param パラメータの設定ファイルは`~.rcssserver/player.conf` です．この内容を編集するか，起動時のコマンドラインオプションで指定することでヘテロジニアスプレイヤに関する設定を変更することができます．しかし，**player_param** の変更はサッカーエージェントの挙動に大きく影響するため，本当に必要でない限り変更すべきではありません．

使用例:

```
$ rcssserver player::pt_max = 5
```

パラメータ名	デフォルト値
説明	
dash_power_rate_delta_max	0
dash_power_rate に追加される最大値．デフォルト値は 0 に設定されており，実質使用されていない．	

dash_power_rate_delta_min	0
dash_power_rate に追加される最小値．デフォルト値は 0 に設定されており，実質使用されていない．	
effort_max_delta_factor	-0.002
effort_max 変更用パラメータ．***extra_stamina_delta_max*** と ***extra_stamina_delta_min*** によって ***extra_stamina*** とのトレードオフが計算される．	
effort_min_delta_factor	-0.002
effort_min 変更用パラメータ．***extra_stamina_delta_max*** と ***extra_stamina_delta_min*** によって ***extra_stamina*** とのトレードオフが計算される．	
extra_stamina_delta_max	100
extra_stamina の最大値．	
extra_stamina_delta_min	0
extra_stamina の最小値．	
inertia_moment_delta_factor	25
慣性モーメント変更用パラメータ．***player_decay_delta_min*** と ***player_decay_delta_max*** によって ***player_decay*** とのトレードオフが計算される．	
kick_rand_delta_factor	0.5
kick コマンドのノイズ率変更用パラメータ．***kickable_margin_delta_factor*** によって，***kickable_margin*** とのトレードオフが計算される．	
kickable_margin_delta_max	0.2
kickable_margin に追加される最大値．	
kickable_margin_delta_min	0
kickable_margin に追加される最小値．	
new_dash_power_rate_delta_max	0.002
dash_power_rate に追加される最大値．	
new_dash_power_rate_delta_min	0
dash_power_rate に追加される最小値．	
new_stamina_inc_max_delta_factor	-10000
スタミナ回復量の変更用パラメータ．***new_dash_power_rate_delta_min*** と ***new_dash_power_rate_delta_max*** によって ***dash_power_rate*** とのトレードオフが計算される．	
player_decay_delta_max	0.2
player_decay に追加される最大値．	
player_decay_delta_min	0
player_decay に追加される最小値．	
player_size_delta_factor	-100
player_size 変更用パラメータ．***dash_power_rate_delta_min*** と ***dash_power_rate_delta_max*** によって ***dash_power_rate*** とのトレードオフが計算される．しかし，***dash_power_rate_delta_min*** と ***dash_power_rate_delta_max*** が 0 のため，実質使用されない．	
player_speed_max_delta_max	0
player_speed_max 変更用パラメータ．デフォルト値は 0 に設定されており，実質使用されていない．	
player_speed_max_delta_min	0
player_speed_max 変更用パラメータ．デフォルト値は 0 に設定されており，実質使用されていない．	

player_types	7
ヘテロジニアスプレイヤのタイプの数．変更すべきでは無い．	
pt_max	3
動タイプのプレイヤが同時にフィールド上に存在できる最大数．例えば，タイプ 1 のプレイヤは同時に 3 人までしか使用できない．デフォルトタイプはこの制限を受けない．	
random_seed	-1
ヘテロジニアスプレイヤパラメータの生成に使用される乱数の種．この値を指定すると，毎回全く同じプレイヤタイプが生成できる．この値が負の場合，パラメータは完全にランダムに生成される．	
stamina_inc_max_delta_factor	0
スタミナ回復量の変更用パラメータ ***player_speed_max_delta_min*** と ***player_speed_max_delta_max*** によって ***dash_power_rate*** とのトレードオフが計算される．***player_speed_max_delta_min***, ***player_speed_max_delta_max*** の値は 0 のため，実質使用されない．	
subs_max	3
1 ハーフ中のプレイヤ交替回数の最大値．プレイモードが ‘`before_kick_off`’ のときはこの制限を受けず，無限にプレイヤ交替できる．	

A.2.3 player_type

ヘテロジニアスプレイヤのパラメータは rcssserver の起動時に毎回新しく生成されるため，設定ファイル等で直接変更することはできません．

パラメータ名 説明
id
プレイヤタイプ番号．デフォルトタイプの番号は常に 0 になる．
player_speed_max
このプレイヤタイプの最大スピード．能力的に到達できる最大スピードとは異なり，実際の最大到達可能スピードとこの値は一致しない．能力的な最大到達スピードがこの値を越えているときは，強制的に正規化される．
stamina_inc_max
$recovery = 1$ のときの 1 サイクルあたりの *stamina* 回復量．
player_decay
速度減衰率．
inertia_moment
慣性モーメントパラメータ．
dash_power_rate
ダッシュ効果率．
player_size
プレイヤの半径．
kickable_margin
キック可能領域のマージン．
kick_rand
kick コマンドと **tackle** コマンド実行時に作用するノイズの割合．

extra_stamina プレイヤの *stamina* 値が 0 のときに追加で使えるスタミナ量.
effort_max $effort$ の最大値.
effort_min $effort$ の最小値.

Section A.3
モニタのコマンド

モニタクライアントにもサッカーエージェントと同様にコマンドが用意されています.

コマンドフォーマット 説明	対応するコマンドクラス
(dispinit[*version*])	MonitorInitCommand
rcssserver との接続を初期化するコマンド. *version* によってプロトコルバージョンを指定できる. プロトコルバージョンとして 1 と 2 の二種類が使える. *version* を省略した場合, プロトコルバージョンは自動的に 1 になる.	
(dispbye)	MonitorByeCommand
rcssserver との接続を切断するコマンド. このコマンドが rcssserver に受理されると, rcssserver から何の情報も送られてこなくなり, モニタからのコマンドも受けつけられなくなる.	
(dispstart)	MonitorKickOffCommand
'KickOff' を実行し, 試合を開始する.	
(dispfoul x y 0)	MonitorDropBallCommand
ボールを (x, y) の位置にドロップする. x と y は整数値で指定する. 実際のフィールド上の XY 座標値に SHOWINFO_SCALE($= 16.0$) の値をかけた値が使用される.	
(dispfoul x y *side*)	MonitorFreeKickCommand

ボールを (x, y) の位置にドロップし，プレイモードを ‘free_kick_l’ または ‘free_kick_r’ に変更する．*side* が’1’ のとき左サイド，’-1’ のとき右サイドへのフリーキックを与える．x と y は整数値で指定する．実際のフィールド上の XY 座標値に SHOWINFO_SCALE(= 16.0) の値をかけた値が使用される．

(dispplayer *side unum x y angle*) MonitorMovePlayerCommand

side と *unum* で指定されるプレイヤを (x, y) の位置へ移動させ，更に体の向きを *angle* にする．*side* は’1’ または’-1’ で指定し，’1’ が左サイド，’-1’ が右サイドを意味する．x と y と *angle* は整数値で指定する．実際のフィールド上の XY 座標値と角度の値に SHOWINFO_SCALE(= 16.0) の値をかけた値が使用される．

(dispdiscard *side unum*) MonitorDiscardPlayerCommand

side と *unum* で指定されるプレイヤを退場させる．ただし，仕様上そう決められているだけで，rcssserver はこのコマンドを処理することができない．

(compression *level*) MonitorDiscardPlayerCommand

通信メッセージを gzip 圧縮する．*level* は圧縮のレベルを意味し，’0’ のとき無圧縮となる．通常，モニタクライアントがこのコマンドを使う必要は無い．

Section A.4
rcssmonitor のオプション

rcssmonitor の設定ファイルは~.rcssmonitor.conf です．この内容を編集するか，起動時のコマンドラインオプションで指定することで rcssmonitor の設定を変更することができます．

使用例:

```
$ rcssmonitor -window_size_x 800 -window_size_y 600
```

パラメータ名 デフォルト値 説明
plane_origin_x 起動時にウインドウの中心となる X 座標値．指定する必要は無い．

plane_origin_y
起動時にウインドウの中心となる Y 座標値．指定する必要は無い．

plane_size_x
起動時に表示されるフィールドの長さ．指定する必要は無い．

plane_size_y
起動時に 表示されるフィールドの幅．指定する必要は無い．

window_size_x 600
起動時のウインドウの幅．

window_size_y 450
起動時のウインドウの高さ．

line_thickness 1
フィールドのラインの太さ．

font_name 6x13bold
スコアボードで使用されるフォントの名前．プロジェクタに表示する場合などはスコアボードが見え難いので大きめのフォントに変えると良い．利用できるフォントは `/usr/X11R6/lib/X11/fonts/misc/`などで見つけられる．

font_inside 6x13bold
プレイヤの背番号などの表示に使用されるフォントの名前．

port 6000
rcssserver への接続ポート番号．

host localhost
rcssserver への接続ホスト名．

version 2
使用するモニタプロトコルのバージョン．

connect_on_start 1
rcssmonitor の起動時に rcssserver へ接続するか否か．

keepaway 0
Keepaway モード用の矩形領域を表示するか否か．

keepaway_length 20
keepaway 領域の矩形の長さ．

keepaway_width 20
keepaway 領域の矩形の幅．

list_player_types 0
rcssserver に接続後，ヘテロジニアスプレイヤのパラメータリストを表示するか否か．

show_ball_collisions 0
ボールに衝突が発生した場合，その情報を表示するか否か．

just_edit 0
通常，使用する機会は無い．

scale 1
ボールとプレイヤの拡大表示率．

detail 1
起動時のボールやプレイヤの詳細情報を表示するレベルを指定．rcssmonitor の操作ボタンの’detail’ に相当．

mode std
起動時の rcssmonitor のモードを指定．rcssmonitor の操作ボタンの’mode’ に相当．

player_types 0.3
rcssserver がプレイヤの半径を通知してこなかった場合に使用されるプレイヤ半径．

kick_radius 1.085

rcssserver がプレイヤのキック可能領域情報を通知してこなかった場合に使用されるキック可能領域の半径.

player_num_pos_x	0
プレイヤの背番号の描画位置．プレイヤの中心位置からの相対座標値を指定する．使用することは無い.	
player_num_pos_y	0
プレイヤの背番号の描画位置．プレイヤの中心位置からの相対座標値を指定する．使用することは無い.	
ball_radius	0.085
rcssserver がボールの半径を通知してこなかった場合に使用されるボール半径.	
c_team_l	ffff00
左チームのフィールドプレイヤの色.	
c_goalie_l	00ff00
左チームのキーパの色.	
c_font_l	ff0000
左チームのプレイヤの背番号などを表示するフォントの色.	
c_team_r	00ffff
右チームのフィールドプレイヤの色.	
c_goalie_r	ff99ff
右チームのキーパの色.	
c_font_r	00ff00
右チームのプレイヤの背番号などを表示するフォントの色.	
c_ball	ffffff
ボールの色.	
c_field	009900
フィールドの色.	
c_line	ffffff
ラインの色.	
c_goal	000000
ゴールの色.	
c_varea_exact	00b400
プレイヤの視界領域の色．プレイヤからの距離が $20m$ 以内の領域で使用される..	
c_varea_fuzzy	00aa00
プレイヤの視界領域の色．プレイヤからの距離が $20m$ 以上の領域で使用される.	

付 録B

利用許諾

アトリビューション-ノンコマーシャル 2.1 （帰属-非営利）

クリエイティブ・コモンズ及びクリエイティブ・コモンズ・ジャパンは法律事務所ではありません。この利用許諾条項の頒布は法的アドバイスその他の法律業務を行うものではありません。クリエイティブ・コモンズ及びクリエイティブ・コモンズ・ジャパンは、この利用許諾の当事者ではなく、ここに提供する情報及び本作品に関しいかなる保証も行いません。クリエイティブ・コモンズ及びクリエイティブ・コモンズ・ジャパンは、いかなる法令に基づこうとも、あなた又はいかなる第三者の損害（この利用許諾に関連する通常損害、特別損害を含みますがこれらに限られません）について責任を負いません。

利用許諾

本作品（下記に定義する）は、このクリエイティブ・コモンズゥ僖屮螢奪ゥ イセンス日本版（以下「この利用許諾」という）の条項の下で提供される。本作品

は、著作権法及び／又は他の適用法によって保護される。本作品をこの利用許諾又は著作権法の下で授権された以外の方法で使用することを禁止する。

許諾者は、かかる条項をあなたが承諾することとひきかえに、ここに規定される権利をあなたに付与する。本作品に関し、この利用許諾の下で認められるいずれかの利用を行うことにより、あなたは、この利用許諾（条項）に拘束されることを承諾し同意したこととなる。

第 1 条　定義

この利用許諾中の用語を以下のように定義する。その他の用語は、著作権法その他の法令で定める意味を持つものとする。

1. 「二次的著作物」とは、著作物を翻訳し、編曲し、若しくは変形し、または脚色し、映画化し、その他翻案することにより創作した著作物をいう。ただし、編集著作物又はデータベースの著作物（以下、この二つを併せて「編集著作物等」という。）を構成する著作物は、二次的著作物とみなされない。また、原著作者及び実演家の名誉又は声望を害する方法で原著作物を改作、変形もしくは翻案して生じる著作物は、この利用許諾の目的においては、二次的著作物に含まれない。

2. 「許諾者」とは、この利用許諾の条項の下で本作品を提供する個人又は団体をいう。

3. 「あなた」とは、この利用許諾に基づく権利を行使する個人又は団体をいう。

4. 「原著作者」とは、本作品に含まれる著作物を創作した個人又は団体をいう。

5. 「本作品」とは、この利用許諾の条項に基づいて利用する権利が付与される対象たる無体物をいい、著作物、実演、レコード、放送にかかる音又は影像、もしくは有線放送にかかる音又は影像をすべて含むものとする。

6. 「ライセンス要素」とは、許諾者が選択し、この利用許諾に表示されている、以下のライセンス属性をいう：帰属・非営利

第 2 条　著作権等に対する制限

この利用許諾に含まれるいかなる条項によっても、許諾者は、あなたが著作権の制限（著作権法第 30 条〜49 条）、著作者人格権に対する制限（著作権法第 18

条 2 項〜4 項、第 19 条 2 項〜4 項、第 20 条 2 項)、著作隣接権に対する制限（著作権法第 102 条）その他、著作権法又はその他の適用法に基づいて認められることとなる本作品の利用を禁止しない。

第 3 条　ライセンスの付与

この利用許諾の条項に従い、許諾者はあなたに、本作品に関し、すべての国で、ロイヤリティ・フリー、非排他的で、（第 7 条 b に定める期間）継続的な以下のライセンスを付与する。ただし、あなたが以前に本作品に関するこの利用許諾の条項に違反したことがないか、あるいは、以前にこの利用許諾の条項に違反したがこの利用許諾に基づく権利を行使するために許諾者から明示的な許可を得ている場合に限る。

1. 本作品に含まれる著作物（以下「本著作物」という。）を複製すること（編集著作物等に組み込み複製することを含む。以下、同じ。)、
2. 本著作物を翻案して二次的著作物を創作し、複製すること、
3. 本著作物又はその二次的著作物の複製物を頒布すること（譲渡または貸与により公衆に提供することを含む。以下同じ。)、上演すること、演奏すること、上映すること、公衆送信を行うこと（送信可能化を含む。以下、同じ。)、公に口述すること、公に展示すること、
4. 本作品に含まれる実演を、録音・録画すること（録音ハ寝菠　鯀　修 ることを含む)、録音ハ寝菠　砲茲岨夘曚垢襪海函　　袷　　鮃圓Δ海函
5. 本作品に含まれるレコードを、複製すること、頒布すること、公衆送信を行うこと、
6. 本作品に含まれる、放送に係る音又は影像を、複製すること、その放送を受信して再放送すること又は有線放送すること、その放送又はこれを受信して行う有線放送を受信して送信可能化すること、そのテレビジョン放送又はこれを受信して行う有線放送を受信して、影像を拡大する特別の装置を用いて公に伝達すること、
7. 本作品に含まれる、有線放送に係る音又は影像を、複製すること、その有線放送を受信して放送し、又は再有線放送すること、その有線放送を受信して送信可能化すること、その有線テレビジョン放送を受信して、影像を拡大する特別の装置を用いて公に伝達すること、

上記に定められた本作品又はその二次的著作物の利用は、現在及び将来のすべての媒体・形式で行うことができる。あなたは、他の媒体及び形式で本作品又はその二次的著作物を利用するのに技術的に必要な変更を行うことができる。許諾者は本作品又はその二次的著作物に関して、この利用許諾に従った利用については自己が有する著作者人格権及び実演家人格権を行使しない。許諾者によって明示的に付与されない全ての権利は、留保される。

第 4 条　受領者へのライセンス提供

あなたが本作品をこの利用許諾に基づいて利用する度毎に、許諾者は本作品又は本作品の二次的著作物の受領者に対して、直接、この利用許諾の下であなたに許可された利用許諾と同じ条件の本作品のライセンスを提供する。

第 5 条　制限

上記第 3 条及び第 4 条により付与されたライセンスは、以下の制限に明示的に従い、制約される。

1. あなたは、この利用許諾の条項に基づいてのみ、本作品を利用することができる。
2. あなたは、本作品又は本作品の二次的著作物を利用するときは、この利用許諾の写し又は URI（Uniform Resource Identifier）を本作品の複製物に添付又は表示しなければならない。
3. あなたは、この利用許諾条項及びこの利用許諾によって付与される利用許諾受領者の権利の行使を変更又は制限するような、本作品又はその二次的著作物に係る条件を提案したり課したりしてはならない。
4. あなたは、本作品を再利用許諾することができない。
5. あなたは、本作品又はその二次的著作物の利用にあたって、この利用許諾及びその免責条項に関する注意書きの内容を変更せず、見やすい態様でそのまま掲載しなければならない。
6. あなたは、この利用許諾条項と矛盾する方法で本著作物へのアクセス又は使用をコントロールするような技術的保護手段を用いて、本作品又はその二次的著作物を利用してはならない。

7. 本条の制限は、本作品又はその二次的著作物が編集著作物等に組み込まれた場合にも、その組み込まれた作品に関しては適用される。しかし、本作品又はその二次的著作物が組み込まれた編集著作物等そのものは、この利用許諾の条項に従う必要はない。

8. あなたは、本作品又はその二次的著作物を営利目的で利用してはならない。デジタル・ファイル共有その他の手段による本作品又はその二次的著作物とその他の作品との交換は、作品の交換に関連して金銭的報酬の支払いがない限り、営利を目的とするものとはみなさない。

9. あなたは、本作品、その二次的著作物又は本作品を組み込んだ編集著作物等を利用する場合には、(1) 本作品に係るすべての著作権表示をそのままにしておかなければならず、(2) 原著作者及び実演家のクレジットを、合理的な方式で、(もし示されていれば原著作者及び実演家の名前又は変名を伝えることにより、) 表示しなければならず、(3) 本作品のタイトルが示されている場合には、そのタイトルを表示しなければならず、(4) 許諾者が本作品に添付するよう指定した URI があれば、合理的に実行可能な範囲で、その URI を表示しなければならず (ただし、その URI が本作品の著作権表示またはライセンス情報を参照するものでないときはこの限りでない。) (5) 二次的著作物の場合には、当該二次的著作物中の原著作物の利用を示すクレジットを表示しなければならない。これらのクレジットは、合理的であればどんな方法でも行うことができる。しかしながら、二次的著作物又は編集著作物等の場合には、少なくとも他の同様の著作者のクレジットが表示される箇所で当該クレジットを表示し、少なくとも他の同様の著作者のクレジットと同程度に目立つ方法であることを要する。

10. もし、あなたが、本作品の二次的著作物、又は本作品もしくはその二次的著作物を組み込んだ編集著作物等を創作した場合、あなたは、許諾者からの通知があれば、実行可能な範囲で、要求に応じて、二次的著作物又は編集著作物等から、許諾者又は原著作者への言及をすべて除去しなければならない。

第 6 条　責任制限

この利用許諾の両当事者が書面にて別途合意しない限り、許諾者は本作品を現状のまま提供するものとし、明示・黙示を問わず、本作品に関していかなる保証

（特定の利用目的への適合性、第三者の権利の非侵害、欠陥の不存在を含むが、これに限られない。）もしない。

この利用許諾又はこの利用許諾に基づく本作品の利用から発生する、いかなる損害（許諾者が、本作品にかかる著作権、著作隣接権、著作者人格権、実演家人格権、商標権、パブリシティ権、不正競争防止法その他関連法規上保護される利益を有する者からの許諾を得ることなく本作品の利用許諾を行ったことにより発生する損害、プライバシー侵害又は名誉毀損から発生する損害等の通常損害、及び特別損害を含むが、これに限らない。）についても、許諾者に故意又は重大な過失がある場合を除き、許諾者がそのような損害発生の可能性を知らされたか否かを問わず、許諾者は、あなたに対し、これを賠償する責任を負わない。

第 7 条　終了

1. この利用許諾は、あなたがこの利用許諾の条項に違反すると自動的に終了する。しかし、本作品、その二次的著作物又は編集著作物等をあなたからこの利用許諾に基づき受領した第三者に対しては、その受領者がこの利用許諾を遵守している限り、この利用許諾は終了しない。第 1 条、第 2 条、第 4 条から第 9 条は、この利用許諾が終了してもなお有効に存続する。

2. 上記 a に定める場合を除き、この利用許諾に基づくライセンスは、本作品に含まれる著作権法上の権利が存続するかぎり継続する。

3. 許諾者は、上記 a および b に関わらず、いつでも、本作品をこの利用許諾に基づいて頒布することを将来に向かって中止することができる。ただし、許諾者がこの利用許諾に基づく頒布を将来に向かって中止した場合でも、この利用許諾に基づいてすでに本作品を受領した利用者に対しては、この利用許諾に基づいて過去及び将来に与えられるいかなるライセンスも終了することはない。また、上記によって終了しない限り、この利用許諾は、全面的に有効なものとして継続する。

第 8 条　その他

1. この利用許諾のいずれかの規定が、適用法の下で無効及び／又は執行不能の場合であっても、この利用許諾の他の条項の有効性及び執行可能性には影響しない。

2. この利用許諾の条項の全部又は一部の放棄又はその違反に関する承諾は、これが書面にされ、当該放棄又は承諾に責任を負う当事者による署名又は記名押印がなされない限り、行うことができない。

3. この利用許諾は、当事者が本作品に関して行った最終かつ唯一の合意の内容である。この利用許諾は、許諾者とあなたとの相互の書面による合意なく修正されない。

4. この利用許諾は日本語により提供される。この利用許諾の英語その他の言語への翻訳は参照のためのものに過ぎず、この利用許諾の日本語版と翻訳との間に何らかの齟齬がある場合には日本語版が優先する。

第 9 条　準拠法

この利用許諾は、日本法に基づき解釈される。

本作品がクリエイティブ・コモンズ・ライセンスに基づき利用許諾されたことを公衆に示すという限定された目的の場合を除き、許諾者も被許諾者もクリエイティブ・コモンズの事前の書面による同意なしに「クリエイティブ・コモンズ」の商標若しくは関連商標又はクリエイティブ・コモンズのロゴを使用しないものとします。使用が許可された場合はクリエイティブ・コモンズおよびクリエイティブゥ灰皀鵐ゥ献碘僖 のウェブサイト上に公表される、又はその他随時要求に従い利用可能となる、クリエイティブ・コモンズの当該時点における商標使用指針を遵守するものとします。クリエイティブ・コモンズは http://creativecommons.org/から、クリエイティブ・コモンズ・ジャパンは http: //www.creativecommons.jp/から連絡することができます。

索 引