/100

指導教員氏名　高橋　直樹

情報工学実験報告書

ROBOCODE関する実験

報告者　　第4学年　出席番号　28

氏名　和田正弘

実験日

1回目　2005年10月11日(火)
2回目　2005年10月25日(火)
3回目　2005年11月 1日(火)
4回目　2005年11月 8日(火)
5回目　2005年11月15日(火)
6回目　2005年11月22日(火)
7回目　2005年12月 6日(火)
8回目　2005年12月13日(火)
9回目　2005年12月20日(火)
10回目　2005年12月21日(火)
11回目　2006年1月17日(火)
12回目　2006年1月24日(火)
13回目　2006年2月 7日(火)
14回目　2006年2月 14日(火)

実験所要時間　44時間00分

実験場所　情報通信実験室

提出期限　2006年3月3日(水) 17:00

提出日　2006年 3月 3日(火)

目 次

1 実験の目的

- Robocodeを用いてJava言語とオブジェクト指向プログラミングを理解する
- Linux上での開発を通してUNIXの使用方法を学ぶ

2 実験環境

- OS
- 開発言語
- 開発環境

3 Robocodeについて

3.1 Robocodeとは

Robocodeとは、IBM社が開発したJava言語によってロボットを開発するロボットシミュレータである。

3.2 Robocodeのルール

2体以上のロボット同士を戦わせ、ポイントをより多く獲得したロボット又は最も長く生存していたロボットが勝利となる。試合を行うバトルフィールドのサイズや同時に戦闘を行うロボットの数、ポイント制か勝ち残り制かなどは試合を開始する前に自由に設定できる。

4 ロボットについて

4.1 ロボットの構成

ロボットは戦車を模しており、レーダー部、砲台部、車体部の3つからなる。　レーダー部、砲台部、車体部はそれぞれ独立に制御できる。

4.2 レーダー部

レーダー部ではレーダーの向きを変えることによって敵ロボットを探索することができる。

4.3 砲台部

砲台部では、砲台を回転させ敵ロボットに照準をあわせ弾丸を発射することにより、敵ロボットを攻撃することができる。

4.4 車体部

車体部では、進行方向の変更や前進・後退を行いロボットを移動させることができる

5 戦略と戦術

5.1 戦略

確実に玉を当てる

5.2 戦術

敵機に十分接近し、狙いをさだめてから弾を発射する

6 射撃アルゴリズムについて

6.1 等速直線運動予測

1. 敵ロボットをスキャンして、座標、進行方向、移動速度を得る
2. スキャンした座標を (X_e, Y_e)、進行方向を θ_e、移動速度を V_e とすると、t 刻時後の座標 $(X_e^{'}, Y_e^{'})$ は以下の式であたえられる

$$X_e^{'} = V_e sin(\theta_e) \cdot t + X_e \quad (1)$$

$$Y_e^{'} = V_e cos(\theta_e) \cdot t + Y_e \quad (2)$$

3. 発射する弾丸の速度を V_b とすると t 刻時後の自ロボットと弾丸との距離 r_b は以下の式で与えられる

$$r_b = V_b \cdot t \quad (3)$$

同様に初期位相は、右回転の場合進行方向から左方向に 90 度回転させた値であり、左回転の場合右方向に 90 度回転させた値となる。そのため、右回転における初期位相 θ_{0r}、左回転に置ける初期位相 θ_{0l} は以下の式より導出される。

$$\theta_{0r} = \theta_e - 90 \tag{12}$$

$$\theta_{0l} = \theta_e + 90 \tag{13}$$

7. 中心座標への方向 θ_r、半径 r、敵ロボットの座標 (X_e, Y_e) より中心座標 (X_c, Y_c) を求める。中心座標は (X_c, Y_c) から θ_r 方向に r の距離の場所にあるはずなので以下の式によって与えられる。

$$X_c = X_e + rsin\theta_r \tag{14}$$

$$Y_c = Y_e + rcos\theta_r \tag{15}$$

8. 以上の計算によって円運動に必要なパラメーターをすべて計算できたので、円運動による刻時 t 後の座標 $(X_e^{'}, Y_e^{'})$ を求める。$(X_e^{'}, Y_e^{'})$ は以下の式によって与えられる。

$$X_e^{'} = X_0 + rsin(\theta_0 + \omega t) \tag{16}$$

$$Y_e^{'} = Y_0 + rcos(\theta_0 + \omega t) \tag{17}$$

9. 自ロボットと刻時 t 後の敵ロボットとの距離 r_e は、自ロボットの座標を $(X_m, Y_<)$ とすると以下の式で与えられる

$$r_e = \sqrt{(X_e' - X_m)^2 + (Y_e' - Y_m)^2} \tag{18}$$

10. 自ロボットと発射された弾丸との距離 r_b は弾丸の速度を V_b とすると、以下の式から与えられる

$$r_b = V_b \cdot t \tag{19}$$

11. 発射した弾丸が敵ロボットに直撃したとき、自ロボットと敵ロボットの距離と自ロボットと弾丸との距離は等しいはずなので

$$r_b = r_e \tag{20}$$

$$V_b \cdot t = \sqrt{(X_e' - X_m)^2 + (Y_e' - Y_m)^2} \tag{21}$$

$$= \sqrt{(X_e' - (X_0 + rsin(\theta_0 + \omega t)))^2 + (Y_e' - (Y_0 + rcos(\theta_0 + \omega t)))^2} \tag{22}$$

となる t を求める。t の計算には同様にセカント法を用いた

12. 求めた刻時 t を用いて、t 刻時後の座標 $(X_e^{'}, Y_e^{'})$ を計算しその座標に向けて弾丸を発射する

7 セカント法について

4. 自ロボットの座標を (X_m, Y_m) とすると式 (2)、(3) より自ロボットと敵ロボットとの距離 r_e は以下の式で与えられる

$$r_e = \sqrt{(X_e' - X_m)^2 + (Y_e' - Y_m)^2)} \quad (4)$$

$$= \sqrt{(V_e sin(\theta_e) \cdot t + X_e - X_m)^2 + (V_e cos(\theta_e) \cdot t + Y_e - Y_m)^2} \quad (5)$$

5. 発射した弾丸が敵ロボットに直撃するときには、自ロボットと弾丸との距離と自ロボットと敵ロボットの距離が等しくなるはずなので

$$r_b = r_e \quad (6)$$

$$V_b \cdot t = \sqrt{(V_e sin(\theta_e) \cdot t + X_e - X_m)^2 + (V_e cos(\theta_e) \cdot t + Y_e - Y_m)^2} \quad (7)$$

となる刻時 t を求める。今回のプログラムではセカント法を用いて計算した。

6. 求めた刻時 t を用いて、t 刻時後の座標 (X_e', Y_e') を計算しその座標に向けて弾丸を発射する

6.2 円運動予測

1. 敵ロボットをスキャンし、敵ロボットの進行方向 θ_e、スキャンした時刻 t を得る

2. 再度敵ロボットをスキャンし、敵ロボットの進行方向 θ_e'、座標 (X_e, Y_e)、移動速度 V_e を得る。

3. 1 度めのスキャンから得た進行方向と 2 度目のスキャンから得た進行方向を用いて角速度 ω を計算する。ω は以下の計算によって求めることができる。

$$\omega = \frac{\theta_e' - \theta_e}{t' - t} \quad (8)$$

4. 角速度と移動速度の関係から円運動の半径 r を求める。半径 r は以下の式より与えられる

$$r = \frac{V_e}{\omega} \quad (9)$$

5. 角速度の符号より右回転か左回転かを判断する。角速度が正の値なら角度が増加する方向にロボットが回転しているので右回転となる。また、角速度が負なら角度が減少する方向に回転しているので左方向への回転となる。

6. 次に初期位相 θ_0 と中心への角度 θ_c を計算する。中心座標は右回転なら右方向へ 90 度回転させた方向に、左回転なら左方向に 90 度回転させた方向にある。現在の敵ロボットの進行方向を θ_e とすると、右回転に置ける中心への角度 θ_{cr}、左回転に置ける中心への角度 θ_{cl} は以下のようにして求めることができる。

$$\theta_{cr} = \theta_e + 90 \quad (10)$$

$$\theta_{cl} = \theta_e - 90 \quad (11)$$

7.1 セカント法とは

セカント法とはコンピューターを用いた方程式の解法の一種である。セカント法はニュートン法を単純化した解法である。ニュートン法では解を見つけるために関数 f とその導関数 $\dot{f}$ を必要とする。関数によっては導関数 $\dot{f}$ を計算するのに非常に時間がかかる場合があるのでセカント法では導関数 $\dot{f}$ を以下の式で近似させる。t_n、t_{n-1} はｎ回めのｔの値を表す

$$\dot{f} = \frac{f(t_n) - f(t_{n-1})}{t_n - t_{n-1}} \tag{23}$$

上式において $f(t_n)$、$f(t_{n-1})$ は関数 f の値を求める際にも計算するのでその値を利用することによって高速に導関数を計算することができる。

関数 f、上式によって計算された導関数の近似式 $/dotf$ を用いてニュートン法を行う。

8 回避運動について

8.1 敵機の砲撃の感知方法

敵機のエネルギーの変化を読み取ることによって、敵機が砲撃を行ったかどうかを判断する。自機から発射した弾丸が着弾していないにもかかわらず敵機のエネルギーが減少していた場合は、敵機が砲撃を行ったと考える。

8.2 実装した回避アルゴリズム

弾丸は常に自機に向かって発射されると仮定する。このとき、弾丸のスピードは敵機のエネルギーの変化量から計算できるので刻時も計算することができる。　着弾寸前に前進することによって敵弾を回避し、同時に敵機へも接近することができる。

9 模擬戦の結果

開発者	千葉	早川	吉田	和田	小田	高橋	浅野	土屋	中田	NT x
千葉	＼	○○○	○○○	×××	×××	×××	××○	○×○	××○	×××
早川	×××	＼	×○×	×××	×××	×××	×××	○××	○○×	×××
吉田	×××	○×○	＼	×××	×××	×××	×××	×××	×××	×○×
和田	○○○	○○○	○○○	＼	○○×	○××	○○○	○○○	○○○	×××
小田	○○○	○○○	○○○	××○	＼	○×○	○○○	○○○	○○○	×××
高橋	○○○	○○○	○○○	×○○	×○×	＼	○○○	○○○	○○○	×××

表 1 :模擬戦結果

自分の結果　21 勝 6 敗　1 位

10 感想

オブジェクト指向言語を用いて何かを作るということは初めてだったので非常に良い経験となった。これまで、オブジェクト指向とはどのようなものか漠然としかわからなかったがこの実験を通して理解することができたと思う。この実験で学んだことを卒業研究や今後の実験などにいかしていきたいと思う。

11 参考文献

参考文献

[1] Wrox Press 闘え、ROBOCODE(ロボコード)! (http://www-06.ibm.com/jp/developerworks/java/020329/j_j-robocode.html)

[2] Wrox Press 闘え、ROBOCODE(ロボコード): 第2ラウンド (http://www-06.ibm.com/jp/developerworks/java/020705/j_j-robocode2.html)

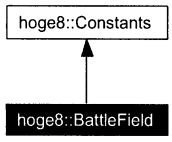
[3] lae Marsh|. Eivind Bjarte. 他 外套と砲塔: ロボコードの達人たちから秘訣を学ぶ (http://www-06.ibm.com/jp/developerworks/java/020726/CloakandTurret.pdf)

[4] 立堀 道昭 立堀の講義ページ (http://lecture.ecc.u-tokyo.ac.jp/ cmich/)

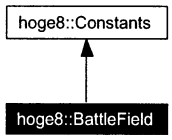
12 作成したロボットの構成について

12.1 クラス hoge8::BattleField

hoge8::BattleField に対する継承グラフ



hoge8::BattleField のコラボレーション図



Public メソッド

- **BattleField** (AdvancedRobot r)
- double **getWidth** ()
- double **getHeight** ()
- double **getMinX** ()
- double **getMinY** ()
- double **getMaxX** ()
- double **getMaxY** ()
- double **getCenterX** ()
- double **getCenterY** ()
- double **getRobotSize** ()

12.1.1 コンストラクタとデストラクタ

12.1.1.1 hoge8::BattleField::BattleField (AdvancedRobot *r*) `[inline]` 各種オブジェクト・変数の宣言や初期化を行う

引数:

← ***r*** 自機オブジェクト

12.1.2 関数

12.1.2.1 double hoge8::BattleField::getCenterX () [inline] バトルフィールドの X 軸における中心値をかえす

戻り値:
X 軸の中心値

12.1.2.2 double hoge8::BattleField::getCenterY () [inline] ボトルフィールドの Y 軸における中心値を返す

戻り値:
Y 座標の中心値

12.1.2.3 double hoge8::BattleField::getHeight () [inline] バトルフィールドの高さを返す

戻り値:
バトルフィールドの高さ

12.1.2.4 double hoge8::BattleField::getMaxX () [inline] ロボットが移動可能な最大の X 座標を返す

戻り値:
移動可能な X 座標の最大値

12.1.2.5 double hoge8::BattleField::getMaxY () [inline] ロボットが移動可能な最大の Y 座標を返す

戻り値:
移動可能な Y 座標の最大値

12.1.2.6 double hoge8::BattleField::getMinX () [inline] ロボットが移動可能な最小の X 座標を返す

戻り値:
移動可能な X 座標の最小値

12.1.2.7 double hoge8::BattleField::getMinY () [inline] ロボットが移動可能な最小の Y 座標を返す

戻り値:
移動可能な Y 座標の最小値

12.1.2.8 double hoge8::BattleField::getRobotSize () [inline] ロボットのサイズを返す

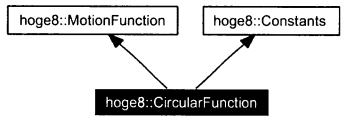
戻り値:
ロボットのサイズ

12.1.2.9 double hoge8::BattleField::getWidth () [inline] バトルフィールドの幅を返す

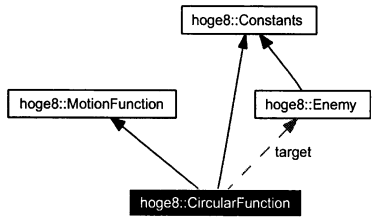
戻り値:
バトルフィールドの幅

12.2 クラス hoge8::CircularFunction

hoge8::CircularFunction に対する継承グラフ



hoge8::CircularFunction のコラボレーション図



Public メソッド

- **CircularFunction** (**Enemy** er)
- void **update** (**Enemy** t)
- double **getNextX** (double t)
- double **getNextY** (double t)

変数

- **Enemy target**
  敵機オブジェクトを格納する.

- double **radius**
  円形運動の半径

- double **InitialHeading**
  円形運動の初期位相

- double **CenterX**
  円形運動の中心の X 座標

- double **CenterY**
  円形運動の中心の Y 座標

- double **heading**
  敵機の進行方向.

12.2.1 説明

円形運動の予測砲撃を行うクラス

12.2.2 コンストラクタとデストラクタ

12.2.2.1 hoge8::CircularFunction::CircularFunction (Enemy *er*) `[inline]` 各種オブジェクト・変数の宣言や初期化を行う

引数:

← ***er*** 敵機オブジェクト

12.2.3 関数

12.2.3.1 double hoge8::CircularFunction::getNextX (double *t*) `[inline]` 刻時 t 後の敵ロボットの自ロボットに対する相対敵な X 座標を返す。

引数:

← ***t*** 予測座標を計算したい刻時 t

戻り値:

刻時 t 後の相対的な X 座標

hoge8::MotionFunction (p. 42) を実装しています.

12.2.3.2 double hoge8::CircularFunction::getNextY (double *t*) [inline] 刻時 t 後の敵ロボットの自ロボットに対する相対敵な Y 座標を返す。

引数:

← *t* 予測座標を計算したい刻時 t

戻り値:

刻時 t 後の相対的な Y 座標

hoge8::MotionFunction (p. 42) を実装しています.

12.2.3.3 void hoge8::CircularFunction::update (Enemy *t*) [inline] 敵機に関する情報を更新する

引数:

← *t* 敵機オブジェクト

hoge8::MotionFunction (p. 42) を実装しています.

12.2.4 変数

12.2.4.1 double hoge8::CircularFunction::CenterX [package] 円形運動の中心の X 座標

12.2.4.2 double hoge8::CircularFunction::CenterY [package] 円形運動の中心の Y 座標

12.2.4.3 double hoge8::CircularFunction::heading [package] 敵機の進行方向.

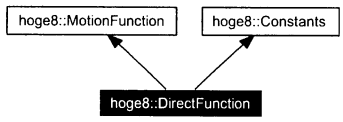
12.2.4.4 double hoge8::CircularFunction::InitialHeading [package] 円形運動の初期位相

12.2.4.5 double hoge8::CircularFunction::radius [package] 円形運動の半径

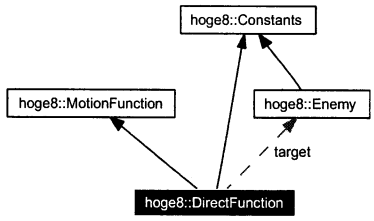
12.2.4.6 Enemy hoge8::CircularFunction::target [package] 敵機オブジェクトを格納する.

12.3 クラス hoge8::DirectFunction

hoge8::DirectFunction に対する継承グラフ



hoge8::DirectFunction のコラボレーション図



Public メソッド

- **DirectFunction** (**Enemy** er)
- void **update** (**Enemy** er)
- double **getNextX** (double t)
- double **getNextY** (double t)

変数

- **Enemy target**
  敵機オブジェクトを格納する.
- double **X**
  敵機の自機に対する相対的な X 座標.
- double **Y**
  敵機の自機に対する相対的な Y 座標.

12.3.1 説明

現在の座標に直接砲撃を行うクラス

12.3.2 コンストラクタとデストラクタ

12.3.2.1 hoge8::DirectFunction::DirectFunction (Enemy *er*) [inline] 各種オブジェクト・変数の宣言や初期化を行う

引数:

← ***er*** 敵機オブジェクト

12.3.3 関数

12.3.3.1 double hoge8::DirectFunction::getNextX (double *t*) [inline] 敵機の相対的なX座標を返す

戻り値:

相対的なX座標

hoge8::MotionFunction (p. 42) を実装しています.

12.3.3.2 double hoge8::DirectFunction::getNextY (double *t*) [inline] 敵機の相対的なY座標を返す

戻り値:

相対的なY座標

hoge8::MotionFunction (p. 42) を実装しています.

12.3.3.3 void hoge8::DirectFunction::update (Enemy *er*) [inline] 敵機に関する情報を更新する

引数:

← ***er*** 敵機オブジェクト

hoge8::MotionFunction (p. 42) を実装しています.

12.3.4 変数

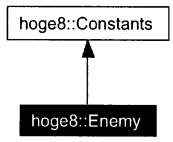
12.3.4.1 Enemy hoge8::DirectFunction::target [package] 敵機オブジェクトを格納する.

12.3.4.2 double hoge8::DirectFunction::X [package] 敵機の自機に対する相対的なX座標.

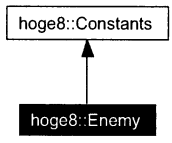
12.3.4.3 double hoge8::DirectFunction::Y [package] 敵機の自機に対する相対的な Y 座標.

12.4 クラス hoge8::Enemy

hoge8::Enemy に対する継承グラフ



hoge8::Enemy のコラボレーション図



Public メソッド

- **Enemy** (ScannedRobotEvent e, AdvancedRobot me)
- double **getRelativeX** ()
- double **getRelativeY** ()
- double **getX** ()
- double **getY** ()
- double **getBearing** ()
- double **getDistance** ()
- double **getEnergy** ()
- double **getLastEnergy** ()
- double **getSpeed** ()
- double **getAngularSpeed** ()
- double **getHeading** ()
- double **getScanTime** ()
- double **getLastScanTime** ()
- double **calcAngularSpeed** (double **current_heading**, double **last_heading**, double current_time, double last_time)
- double **calcRelativeX** (double **bearing**, double **distance**)
- double **calcRelativeY** (double **bearing**, double **distance**)
- double **calcx** (double **x**, double **bearing**, double **distance**)

- double **calcy** (double **y**, double **bearing**, double **distance**)
- void **updateEnergy** (double energy)
- void **update** (ScannedRobotEvent e, AdvancedRobot me)

変数

- double **bearing**
  敵機との角度 *(360 度)*.
- double **distance**
  敵機との距離.
- double **current_energy**
  敵機の現在のエネルギー.
- double **last_energy**
  敵機の以前のエネルギー.
- double **current_speed**
  敵機の現在の速度.
- double **last_speed**
  敵機の以前の速度.
- double **angularSpeed**
  敵機の角速度.
- double **current_acc**
  敵機の現在の加速度.
- double **last_acc**
  敵機の以前の加速度.
- double **current_heading**
  敵機の現在の向き.
- double **last_heading**
  敵機の以前の向き.
- double **x**
  敵機の現在の X 座標.
- double **y**
  敵機の現在の Y 座標.

- double **RelativeX**
  適機の現在の相対的な X 座標.

- double **RelativeY**
  適期の現在の相対的な Y 座標.

- double **scantime**
  最後に敵機をスキャンした時間

- double **last_scantime**
  一回前に敵機をスキャンした時間

- AdvancedRobot **myrobot**
  自機オブジェクト

12.4.1 説明

敵機の情報を保存するクラス

12.4.2 コンストラクタとデストラクタ

12.4.2.1 hoge8::Enemy::Enemy (ScannedRobotEvent *e*, AdvancedRobot *me*) `[inline]` 各種変数に初期データを代入

引数:

← ***e*** ScannedRobotEvent 型オブジェクト

← ***me*** 自機オブジェクト

12.4.3 関数

12.4.3.1 double hoge8::Enemy::calcAngularSpeed (double *current_heading*, double *last_heading*, double *current_time*, double *last_time*) `[inline]` 角速度を計算する

引数:

← ***current_heading*** 現在の進行方向

← ***last_heading*** 以前の進行方向

← ***current_time*** 現在の時刻

← ***last_time*** 以前の時刻

戻り値:

角速度

12.4.3.2 double hoge8::Enemy::calcRelativeX (double *bearing*, double *distance*) [inline] 敵機の相対的なX座標を計算する

引数:

← ***bearing*** 自機と敵機との間の角度

← ***distance*** 自機と敵機との間の距離

戻り値:

相対的なX座標

12.4.3.3 double hoge8::Enemy::calcRelativeY (double *bearing*, double *distance*) [inline] 敵機の相対的なX座標を計算する

引数:

← ***bearing*** 自機と敵機との間の角度

← ***distance*** 自機と敵機との間の距離

戻り値:

相対的なX座標

12.4.3.4 double hoge8::Enemy::calcx (double *x*, double *bearing*, double *distance*) [inline] 敵機のX座標をかえす

引数:

← ***x*** 自機のX座標

← ***bearing*** 自機と敵機との間の角度

← ***distance*** 自機と敵機との間の距離

戻り値:

敵機のX座標

12.4.3.5 double hoge8::Enemy::calcy (double *y*, double *bearing*, double *distance*) [inline] 敵機のY座標をかえす

引数:

← ***y*** 自機のY座標

← ***bearing*** 自機と敵機との間の角度

← ***distance*** 自機と敵機との間の距離

戻り値:

敵機のY座標

12.4.3.6 double hoge8::Enemy::getAngularSpeed () [inline] 敵機の角速度を返す

戻り値:

敵機の角速度

12.4.3.7 double hoge8::Enemy::getBearing () [inline] 敵機との角度 (360 度) を返す

戻り値:

敵機との角度

12.4.3.8 double hoge8::Enemy::getDistance () [inline] 敵機との距離をもどす

戻り値:

敵機との距離

12.4.3.9 double hoge8::Enemy::getEnergy () [inline] 敵機の現在のエネルギーをかえす

戻り値:

敵機の現在のエネルギー

12.4.3.10 double hoge8::Enemy::getHeading () [inline] 敵機の現在の方向 (360 度) を返す

戻り値:

敵機の現在の方向

12.4.3.11 double hoge8::Enemy::getLastEnergy () [inline] 敵機の以前のエネルギーをかえす

戻り値:

敵機の以前のエネルギー

12.4.3.12 double hoge8::Enemy::getLastScanTime () [inline] 一回前にスキャンした時間をかえす

戻り値:

一回前にスキャンした時間

12.4.3.13 double hoge8::Enemy::getRelativeX () [inline] 敵機と自機の相対的な X 座標を返す

戻り値:

相対的な X 座標

12.4.3.14 double hoge8::Enemy::getRelativeY () [inline] 敵機と自機の相対的な Y 座標を返す

戻り値:

相対的な Y 座標

12.4.3.15 double hoge8::Enemy::getScanTime () [inline] 最後にスキャンした時間をかえす

戻り値:
最後にスキャンした時間

12.4.3.16 double hoge8::Enemy::getSpeed () [inline] 敵機の現在の速度を返す 敵機の現在の速度

12.4.3.17 double hoge8::Enemy::getX () [inline] 敵機の X 座標を返す

戻り値:
敵機の X 座標

12.4.3.18 double hoge8::Enemy::getY () [inline] 敵機の Y 座標を返す

戻り値:
敵機の Y 座標

12.4.3.19 void hoge8::Enemy::update (ScannedRobotEvent *e*, AdvancedRobot *me*) [inline] 敵機の情報の更新を実行

引数:
← *e* ScannedRObotEvent 型オブジェクト
← *me* 自機オブジェクト

12.4.3.20 void hoge8::Enemy::updateEnergy (double *energy*) [inline] 敵機のエネルギーを更新する

引数:
← *energy* エネルギー変化分

12.4.4 変数

12.4.4.1 double hoge8::Enemy::angularSpeed [package] 敵機の角速度

12.4.4.2 double hoge8::Enemy::bearing [package] 敵機との角度 (360 度).

12.4.4.3 double hoge8::Enemy::current_acc [package] 敵機の現在の加速度.

12.4.4.4 double hoge8::Enemy::current_energy [package] 敵機の現在のエネルギー.

12.4.4.5 double hoge8::Enemy::current_heading [package] 敵機の現在の向き.

12.4.4.6 double hoge8::Enemy::current_speed [package] 敵機の現在の速度.

12.4.4.7 double hoge8::Enemy::distance [package] 敵機との距離.

12.4.4.8 double hoge8::Enemy::last_acc [package] 敵機の以前の加速度.

12.4.4.9 double hoge8::Enemy::last_energy [package] 敵機の以前のエネルギー.

12.4.4.10 double hoge8::Enemy::last_heading [package] 敵機の以前の向き.

12.4.4.11 double hoge8::Enemy::last_scantime [package] 一回前に敵機をスキャンした時間

12.4.4.12 double hoge8::Enemy::last_speed [package] 敵機の以前の速度.

12.4.4.13 AdvancedRobot hoge8::Enemy::myrobot [package] 自機オブジェクト

12.4.4.14 double hoge8::Enemy::RelativeX [package] 適機の現在の相対的な X 座標.

12.4.4.15 double hoge8::Enemy::RelativeY [package] 適期の現在の相対的な Y 座標.

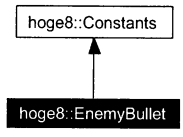
12.4.4.16 double hoge8::Enemy::scantime [package] 最後に敵機をスキャンした時間

12.4.4.17 double hoge8::Enemy::x [package] 敵機の現在の X 座標.

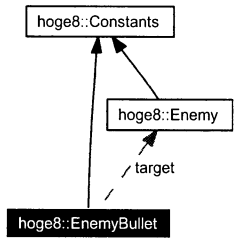
12.4.4.18 double hoge8::Enemy::y [package] 敵機の現在の Y 座標.

12.5 クラス hoge8::EnemyBullet

hoge8::EnemyBullet に対する継承グラフ



hoge8::EnemyBullet のコラボレーション図



Public メソッド

- **EnemyBullet** (**Enemy** er, AdvancedRobot me)
- double **getSpeed** ()
- double **getDirection** ()
- double **getDistance** ()
- double **calcBulletSpeed** (**Enemy target**)
- boolean **getHit** ()
- void **update** (AdvancedRobot me)

変数

- double **speed**
  弾丸のスピード.
- double **X**
  弾丸の X 座標.
- double **Y**
  弾丸の Y 座標.
- double **distance**
  弾丸と自機との距離.
- double **direction**
  弾丸の進行方向.
- double **FireTime**
  弾丸の発射時刻.
- double **LastUpdateTime**
  最後に弾丸の情報が更新された時刻

- boolean **hit** = false
  この弾丸が自機にあたるかどうか *(当たる:true、外れる:false)*

- **Enemy target**
  敵機のオブジェクト.

- AdvancedRobot **myrobot**
  実機のオブジェクト

12.5.1 コンストラクタとデストラクタ

12.5.1.1 hoge8::EnemyBullet::EnemyBullet (Enemy *er*, AdvancedRobot *me*) `[inline]` 各種オブジェクト・変数の初期化を行う

引数:

← ***er*** 敵機オブジェクト

← ***me*** 自機オブジェクト

12.5.2 関数

12.5.2.1 double hoge8::EnemyBullet::calcBulletSpeed (Enemy *target*) `[inline]` 敵機のエネルギー変化から弾丸のスピードを計算する

引数:

← ***target*** 敵ロボットのオブジェクト

戻り値:

弾丸のスピード

12.5.2.2 double hoge8::EnemyBullet::getDirection () `[inline]` 弾丸の進行方向をかえす

戻り値:

弾丸の進行方向

12.5.2.3 double hoge8::EnemyBullet::getDistance () `[inline]` 弾丸と自機との距離をかえす

戻り値:

弾丸と自機との距離

12.5.2.4 boolean hoge8::EnemyBullet::getHit () [inline] 弾丸がヒットするかどうかをかえす

戻り値:

弾丸がヒットするとき true を返す。 それ以外の時は false を返す

12.5.2.5 double hoge8::EnemyBullet::getSpeed () [inline] 弾丸のスピードをかえす

戻り値:

弾丸のスピード

12.5.2.6 void hoge8::EnemyBullet::update (AdvancedRobot *me*) [inline] 弾丸の情報を更新する

引数:

← *me* 自機のオブジェクト

12.5.3 変数

12.5.3.1 double hoge8::EnemyBullet::direction [package] 弾丸の進行方向.

12.5.3.2 double hoge8::EnemyBullet::distance [package] 弾丸と自機との距離.

12.5.3.3 double hoge8::EnemyBullet::FireTime [package] 弾丸の発射時刻.

12.5.3.4 boolean hoge8::EnemyBullet::hit = false [package] この弾丸が自機にあたるかどうか (当たる:true、外れる:false)

12.5.3.5 double hoge8::EnemyBullet::LastUpdateTime [package] 最後に弾丸の情報が更新された時刻

12.5.3.6 AdvancedRobot hoge8::EnemyBullet::myrobot [package] 実機のオブジェクト

12.5.3.7 double hoge8::EnemyBullet::speed [package] 弾丸のスピード.

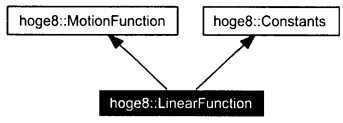
12.5.3.8 Enemy hoge8::EnemyBullet::target [package] 敵機のオブジェクト.

12.5.3.9 double hoge8::EnemyBullet::X [package] 弾丸の X 座標.

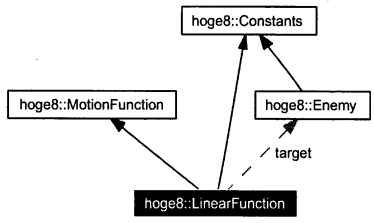
12.5.3.10 double hoge8::EnemyBullet::Y [package] 弾丸の Y 座標.

12.6 クラス hoge8::LinearFunction

hoge8::LinearFunction に対する継承グラフ



hoge8::LinearFunction のコラボレーション図



Public メソッド

- **LinearFunction** (**Enemy** t)
- void **update** (**Enemy** t)
- double **getNextX** (double t)
- double **getNextY** (double t)

変数

- **Enemy target**
  敵機オブジェクトを格納する.
- double **X**
  敵機の自機に対する相対的な X 座標.
- double **Y**
  敵機の自機に対する相対的な X 座標.

12.6.1 説明

円形運動の予測砲撃を行うクラス

12.6.2 コンストラクタとデストラクタ

12.6.2.1 hoge8::LinearFunction::LinearFunction (Enemy *t*) [inline] 各種オブジェクト・変数の宣言や初期化を行う

引数:

← ***t*** 敵機オブジェクト

12.6.3 関数

12.6.3.1 double hoge8::LinearFunction::getNextX (double *t*) [inline] 刻時 t 後の敵機の自機に対する相対敵な X 座標を返す。

引数:

← ***t*** 予測座標を計算したい刻時 t

戻り値:

刻時 t 後の相対的な X 座標

hoge8::MotionFunction (p. 42) を実装しています.

12.6.3.2 double hoge8::LinearFunction::getNextY (double *t*) [inline] 刻時 t 後の敵機の自機に対する相対敵な Y 座標を返す。

引数:

← ***t*** 予測座標を計算したい刻時 t

戻り値:

刻時 t 後の相対的な Y 座標

hoge8::MotionFunction (p. 42) を実装しています.

12.6.3.3 void hoge8::LinearFunction::update (Enemy *t*) [inline] 敵機に関する情報を更新する

引数:

← ***t*** 敵機オブジェクト

hoge8::MotionFunction (p. 42) を実装しています.

12.6.4 変数

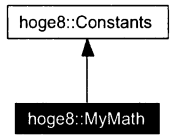
12.6.4.1 Enemy hoge8::LinearFunction::target [package] 敵機オブジェクトを格納する.

12.6.4.2 double hoge8::LinearFunction::X [package] 敵機の自機に対する相対的な X 座標.

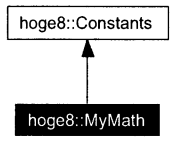
12.6.4.3 double hoge8::LinearFunction::Y [package] 敵機の自機に対する相対的な X 座標.

12.7 クラス hoge8::MyMath

hoge8::MyMath に対する継承グラフ



hoge8::MyMath のコラボレーション図



Static Public メソッド

- static double **to360** (double angle)
- static double **to180** (double fromAngle, double toAngle)
- static double **calcHeading** (double fromX, double fromY, double toX, double toY)
- static double **calcDistance** (double fromX, double fromY, double toX, double toY)
- static double **calcNextX** (double x, double distance, double angle)
- static double **calcNextY** (double y, double distance, double angle)
- static double **calcBulletSpeed** (double Power)
- static double **calcBulletDamage** (double Power)
- static double **calcBulletTrt** (double Power)
- static double **calcBulletPower** (**Enemy** target)
- static double **sin** (double angle)
- static double **cos** (double angle)
- static double **tan** (double angle)
- static double **atan** (double data)
- static double **asin** (double data)

12.7.1 説明

各種数学的関数を集めたクラス

12.7.2 関数

12.7.2.1 static double hoge8::MyMath::asin (double *data*) `[inline, static]` 指定した値の asin をとる (角度)

引数:

← ***data*** asin をとる値

戻り値:

角度（360 度形式）

12.7.2.2 static double hoge8::MyMath::atan (double *data*) `[inline, static]` 指定した値の atan をとる (角度)

引数:

← ***data*** atan をとる値

戻り値:

角度（360 度形式）

12.7.2.3 static double hoge8::MyMath::calcBulletDamage (double *Power*) `[inline, static]` 弾丸のパワーから被弾時のダメージを計算する

引数:

← ***Power*** 弾丸のパワー

戻り値:

被弾時のダメージ

12.7.2.4 static double hoge8::MyMath::calcBulletPower (Enemy *target*) `[inline, static]` 自機と敵機との距離から適切な弾丸のパワーを計算する

引数:

← ***target*** 敵機オブジェクト

戻り値:

弾丸のパワー

12.7.2.5 static double hoge8::MyMath::calcBulletSpeed (double *Power*) `[inline, static]` 弾丸のパワーから弾丸のスピードを計算する

引数:

← ***Power*** 弾丸のパワー

戻り値:

弾丸のスピード

12.7.2.6 static double hoge8::MyMath::calcBulletTrt (double *Power*) `[inline, static]` 弾丸のパワーから着弾時の回復量を計算する

引数:

← ***Power*** 弾丸のパワー

戻り値:

着弾時の回復量

12.7.2.7 static double hoge8::MyMath::calcDistance (double *fromX*, double *fromY*, double *toX*, double *toY*) `[inline, static]` 指定された 2 点間の距離を計算する

引数:

← ***fromX*** 基準となる点の X 座標

← ***fromY*** 基準となる点の Y 座標

← ***toX*** 目的となる点の X 座標

← ***toY*** 目的となる点の Y 座標

戻り値:

指定された 2 点間の距離

12.7.2.8 static double hoge8::MyMath::calcHeading (double *fromX*, double *fromY*, double *toX*, double *toY*) `[inline, static]` 指定された 2 点間の角度 (360 度) を計算する

引数:

← ***fromX*** 基準となる点の X 座標

← ***fromY*** 基準となる点の Y 座標

← ***toX*** 目的となる点の X 座標

← ***toY*** 目的となる点の Y 座標

戻り値:

指定された 2 点間の角度 (360 度形式)

12.7.2.9 static double hoge8::MyMath::calcNextX (double *x*, double *distance*, double *angle*) `[inline, static]` 現在のX座標、移動距離、進行方向から移動先のX座標を計算する

引数:

← ***x*** 現在のX座標

← ***distance*** 移動距離

← ***angle*** 進行方向

戻り値:

移動先のX座標

12.7.2.10 static double hoge8::MyMath::calcNextY (double *y*, double *distance*, double *angle*) `[inline, static]` 現在のX座標、移動距離、進行方向から移動先のY座標を計算する

引数:

← ***y*** 現在のY座標

← ***distance*** 移動距離

← ***angle*** 進行方向

戻り値:

移動先のY座標

12.7.2.11 static double hoge8::MyMath::cos (double *angle*) `[inline, static]` 指定した角度(360度)のcosをとる

引数:

← ***angle*** 角度（360度形式）

戻り値:

cos(angle)の計算結果

12.7.2.12 static double hoge8::MyMath::sin (double *angle*) `[inline, static]` 指定した角度(360度)のsinをとる

引数:

← ***angle*** 角度（360度形式）

戻り値:

sin(angle)の計算結果

12.7.2.13 static double hoge8::MyMath::tan (double *angle*) [inline, static] 指定した角度 (360 度) の tan をとる

引数:

← ***angle*** 角度（360 度形式）

戻り値:

tan(angle) の計算結果

12.7.2.14 static double hoge8::MyMath::to180 (double *fromAngle*, double *toAngle*) [inline, static] 渡された二つの角度の差を+-180 度形式に変換

引数:

← ***fromAngle*** 基準となる角度（360 度形式）

← ***toAngle*** 差を計算する対象の角度 (360 度形式)

戻り値:

+-180 度形式に変換されたふたつの角度の差

12.7.2.15 static double hoge8::MyMath::to360 (double *angle*) [inline, static] 渡された角度を 360 度形式に変換

引数:

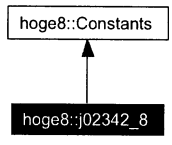
← ***angle*** 角度（+-180 度形式）

戻り値:

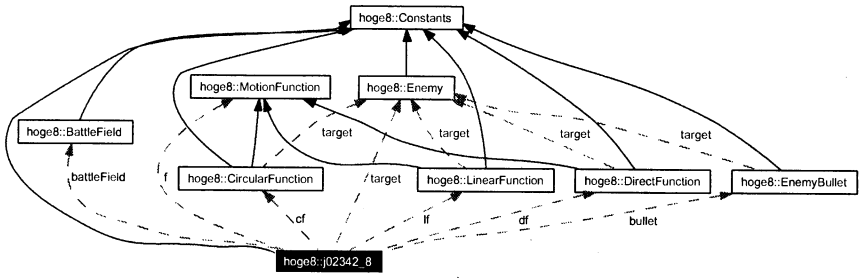
360 度形式に変換された角度

12.8 クラス hoge8::j02342_8

hoge8::j02342_8 に対する継承グラフ



hoge8::j02342_8 のコラボレーション図



Public メソッド

- void **run** ()
- void **init** ()
- void **initFunction** (**Enemy** er)
- void **updateFunction** (**Enemy** er)
- void **selectFunction** (**Enemy** er)
- void **search** ()
- void **move** ()
- void **aim** ()
- void **ahead** (double distance)
- void **setAhead** (double distance)
- void **setTurnRadortoEnemy** (double distance, **Enemy** target, AdvancedRobot me)
- void **setTurnRightAngle** ()
- void **avoid** (**Enemy** target)
- boolean **isHit** (**EnemyBullet** bullet)
- boolean **isSouth** ()
- boolean **isWest** ()
- boolean **isEast** ()
- boolean **isNorth** ()
- boolean **isCircular** (**Enemy** er)
- boolean **isLinear** (**Enemy** er)
- boolean **isDirect** (**Enemy** er)
- double **getEstimatedGunBearing** (**MotionFunction** f, double BP, double min, double max)
- double **getEstimatedImpactTime** (**MotionFunction** f, double BP, double min, double max)
- double **getEstimatedDistance** (**MotionFunction** f, double BP, double t)
- void **onScannedRobot** (ScannedRobotEvent e)
- void **onHitByBullet** (HitByBulletEvent e)
- void **onBulletHit** (BulletHitEvent event)

- void **onWin** (WinEvent event)
- void **onDeath** (DeathEvent event)
- void **onCustomEvent** (CustomEvent ev)

12.8.1 関数

12.8.1.1 void hoge8::j02342_8::ahead (double *distance*) `[inline]` 回避運動時に呼び出され前進する

12.8.1.2 void hoge8::j02342_8::aim () `[inline]` 敵ロボットに照準をあわせる

12.8.1.3 void hoge8::j02342_8::avoid (Enemy *target*) `[inline]` 敵ロボットから発射された弾丸を回避する

引数:

← ***target*** 敵ロボットのオブジェクト

12.8.1.4 double hoge8::j02342_8::getEstimatedDistance (MotionFunction *f*, double *BP*, double *t*) `[inline]` 弾丸と敵機との距離を計算する

引数:

← ***f*** 砲撃用オブジェクト

← ***BP*** 弾丸のパワー

← ***t*** 距離を計算する刻時

戻り値:

刻時 t における弾丸と敵機との距離を返す。

12.8.1.5 double hoge8::j02342_8::getEstimatedGunBearing (MotionFunction *f*, double *BP*, double *min*, double *max*) `[inline]` 予測砲撃を行う際に砲台を回転させる角度を計算する

引数:

← ***f*** 砲撃用オブジェクト

← ***BP*** 弾丸のパワー

← ***min*** 着弾までの最小の刻時

← ***max*** 着弾までの最大の刻時

戻り値:

予測した座標がバトルフィールド内の時は砲台を回転させる角度（0〜360 度）を返す。バトルフィールド外のときは-1 をかえす。

12.8.1.6 double hoge8::j02342_8::getEstimatedImpactTime (MotionFunction *f*, double *BP*, double *min*, double *max*) [inline] 敵ロボットに着弾するまでの時間を計算する

引数:

← ***f*** 砲撃用オブジェクト

← ***BP*** 弾丸のパワー

← ***min*** 着弾までの最小の刻時

← ***max*** 着弾までの最大の刻時

戻り値:

敵ロボットに着弾するまでの刻時を返す。

12.8.1.7 void hoge8::j02342_8::init () [inline] 各種オブジェクトの初期化処理を行う

12.8.1.8 void hoge8::j02342_8::initFunction (Enemy *er*) [inline] 砲撃用オブジェクトの初期化を行う

引数:

← ***er*** 砲撃対象となる敵機オブジェクト

12.8.1.9 boolean hoge8::j02342_8::isCircular (Enemy *er*) [inline] 敵機が円形運動かどうかを判断する

引数:

← ***er*** 敵ロボットのオブジェクト

戻り値:

敵機が円形運動を行ってるとき true を返す。 それ以外の時は false を返す。

12.8.1.10 boolean hoge8::j02342_8::isDirect (Enemy *er*) [inline] 敵機が減速中かどうかを判断する

引数:

← ***er*** 敵機のオブジェクト

戻り値:

敵機が減速中のとき true を返す。 それ以外の時は false を返す。

12.8.1.11 boolean hoge8::j02342_8::isEast () [inline] 自機がバトルフィールドの東側にいるかどうか判断する

戻り値:

東側にいるときは true を返す それ以外の時は false を返す

12.8.1.12 boolean hoge8::j02342_8::isHit (EnemyBullet *bullet*) [inline] 敵ロボットから発射された弾丸が自ロボットに当たるかどうかを判断する

引数:

← ***bullet*** 敵機から発射された弾丸のオブジェクト

戻り値:

当たる場合は ture を返す。 外れる場合は false を返す。

12.8.1.13 boolean hoge8::j02342_8::isLinear (Enemy *er*) [inline] 敵機が等速直線運動かどうかを判断する

引数:

← ***er*** 敵機のオブジェクト

戻り値:

敵機が等速直線運動を行ってるとき true を返す。 それ以外の時は false を返す。

12.8.1.14 boolean hoge8::j02342_8::isNorth () [inline] 自機がバトルフィールドの北側にいるかどうか判断する

戻り値:

北側にいるときは true を返す それ以外の時は false を返す

12.8.1.15 boolean hoge8::j02342_8::isSouth () [inline] 自機がバトルフィールドの南側にいるかどうかを判断する

戻り値:

南側にいるときは true を返す。 それ以外のと気は false を返す。

12.8.1.16 boolean hoge8::j02342_8::isWest () [inline] 自機がバトルフィールドの西側にいるかどうか判断する

戻り値:

西側にいるときは true を返す それ以外の時は false を返す

12.8.1.17 void hoge8::j02342_8::move () [inline] 敵の弾丸を回避する

12.8.1.18 void hoge8::j02342_8::onBulletHit (BulletHitEvent *event*) [inline] 敵機に着弾した際に呼び出され、敵機オブジェクトのエネルギーを更新する

引数:

← ***event*** BulletHitEvent 型のオブジェクト

12.8.1.19 void hoge8::j02342_8::onCustomEvent (CustomEvent *ev*) [inline] 各種カスタムイベントが定義されている

引数:

← *ev* CustomEvent 型のオブジェクト

12.8.1.20 void hoge8::j02342_8::onDeath (DeathEvent *event*) [inline] 敗北した時に呼び出され、各種オブジェクトを破棄する

引数:

← *event* DeathEvent 型のオブジェクト

12.8.1.21 void hoge8::j02342_8::onHitByBullet (HitByBulletEvent *e*) [inline] 自機が被弾した際に呼び出され、bullet オブジェクトを破棄し、敵機オブジェクトのエネルギーを更新する

引数:

← *e* HitByBulletEvent HitByBulletEvent 型のオブジェクト

12.8.1.22 void hoge8::j02342_8::onScannedRobot (ScannedRobotEvent *e*) [inline] レーダーで敵機を発見した際に呼び出され、敵機オブジェクトの作成・更新、砲撃用オブジェクトの作成・選択、レーダーの回転、敵機の砲撃判定を行う

引数:

← *e* ScannedRobotEvent 型のオブジェクト

12.8.1.23 void hoge8::j02342_8::onWin (WinEvent *event*) [inline] 勝利した時に呼び出され、各種オブジェクトを破棄する

引数:

← *event* WinEvent 型のオブジェクト

12.8.1.24 void hoge8::j02342_8::run () [inline] ロボットのメイン処理

12.8.1.25 void hoge8::j02342_8::search () [inline] 敵機を探索する

12.8.1.26 void hoge8::j02342_8::selectFunction (Enemy *er*) [inline] 敵ロボットの運動に適した射撃用オブジェクトを選択する

引数:

← *er* 砲撃対象となる敵機オブジェクト

12.8.1.27 void hoge8::j02342_8::setAhead (double *distance*) [inline]

12.8.1.28 void hoge8::j02342_8::setTurnRadortoEnemy (double *distance*, Enemy *target*, AdvancedRobot *me*) [inline] レーダーを敵ロボットへ向ける

引数:

← ***distance*** 敵機との距離

← ***target*** 敵機オブジェクト

← ***me*** 自機オブジェクト

12.8.1.29 void hoge8::j02342_8::setTurnRightAngle () [inline] 自機の進行方向を敵機に対して 80 度の角度を保つように回転する

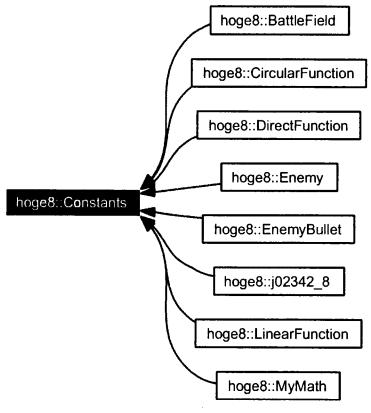
12.8.1.30 void hoge8::j02342_8::updateFunction (Enemy *er*) [inline] 砲撃用オブジェクトの更新を行う

引数:

← ***er*** 砲撃対象となる敵機オブジェクト

12.9 インタフェース hoge8::Constants

hoge8::Constants に対する継承グラフ



Static Public 変数

- static final double **MAX_BULLET_SPEED** = 20
  弾丸の最高速度.
- static final double **MAX_SPEED** = 8.0
  ロボットの最高速度
- static final double **SCAN_ARC** = 45.0

レーダーの振り幅

- static final double **HALF_SCAN_ARC** = **SCAN_ARC** / 2
  レーダーの振り幅（片側）

- static final long **MAX_SCAN_AGE** = 8
  スキャンした情報の寿命

- static final double **MINX** = 20.0
  ロボットが移動する最小のX座標

- static final double **MINY** = 20.0
  ロボットが移動する最小の Y 座標

- static final double **PI** = 3.1415
  π の値

- static final double **SAFE_MARGIN** = 20
  発射された弾丸を被弾するかどうか判断するマージン.

- static final double **AVOID_MARGIN** = 20
  回避行動を開始する弾丸到着までの時間

- static final double **EXT_DIS** = 20
  回避行動の際に移動するマージン

- static final double **MISS** = 0.001
  方程式の解の誤差の範囲.

- static final double **D_L** = 700
  遠距離と中距離との境界値

- static final double **D_S** = 150
  中距離と近距離との境界値.

12.9.1 変数

12.9.1.1 final double hoge8::Constants::AVOID_MARGIN = 20 `[static]` 回避行動を開始する弾丸到着までの時間

12.9.1.2 final double hoge8::Constants::D_L = 700 `[static]` 遠距離と中距離との境界値

12.9.1.3 final double hoge8::Constants::D_S = 150 [static] 中距離と近距離との境界値.

12.9.1.4 final double hoge8::Constants::EXT_DIS = 20 [static] 回避行動の際に移動するマージン

12.9.1.5 final double hoge8::Constants::HALF_SCAN_ARC = SCAN_ARC / 2 [static] レーダーの振り幅（片側）

12.9.1.6 final double hoge8::Constants::MAX_BULLET_SPEED = 20 [static] 弾丸の最高速度.

12.9.1.7 final long hoge8::Constants::MAX_SCAN_AGE = 8 [static] スキャンした情報の寿命

12.9.1.8 final double hoge8::Constants::MAX_SPEED = 8.0 [static] ロボットの最高速度

12.9.1.9 final double hoge8::Constants::MINX = 20.0 [static] ロボットが移動する最小のX座標

12.9.1.10 final double hoge8::Constants::MINY = 20.0 [static] ロボットが移動する最小の Y 座標

12.9.1.11 final double hoge8::Constants::MISS = 0.001 [static] 方程式の解の誤差の範囲.

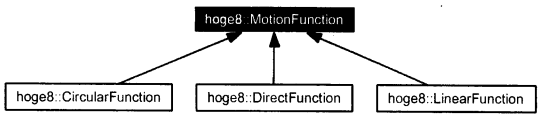
12.9.1.12 final double hoge8::Constants::PI = 3.1415 [static] πの値

12.9.1.13 final double hoge8::Constants::SAFE_MARGIN = 20 [static] 発射された弾丸を被弾するかどうか判断するマージン.

12.9.1.14 final double hoge8::Constants::SCAN_ARC = 45.0 [static] レーダーの振り幅

12.10 インタフェース hoge8::MotionFunction

hoge8::MotionFunction に対する継承グラフ



Public メソッド

- double **getNextX** (double t)

関数

- double **getNextY** (double t)
- void **update** (**Enemy** t)

12.10.1 説明

砲撃用クラスのインターフェース

12.10.2 関数

12.10.2.1 double hoge8::MotionFunction::getNextX (double *t*) **hoge8::CircularFunction** (p. 13), **hoge8::DirectFunction** (p. 16), と **hoge8::LinearFunction** (p. 28) で実装されています.

12.10.2.2 double hoge8::MotionFunction::getNextY (double *t*) [package] **hoge8::CircularFunction** (p. 14), **hoge8::DirectFunction** (p. 16), と **hoge8::LinearFunction** (p. 28) で実装されています.

12.10.2.3 void hoge8::MotionFunction::update (Enemy *t*) [package] **hoge8::CircularFunction** (p. 14), **hoge8::DirectFunction** (p. 16), と **hoge8::LinearFunction** (p. 28) で実装されています.

13 ソースコード

13.1 BattleField.java

```
package hoge8;
```

```
import robocode.*;
/** @file
 *  @brief バトルフィールドのデータを格納するクラス
 *  @author 和田　政弘
 */

/**
 * バトルフィールドのデータを格納するクラス
 * /
public class BattleField implements Constants
{
private static double   width     = 0.0; // バトルフィールドの幅
private static double   height    = 0.0; // バトルフィールドの高さ
private static double   CenterX   = 0.0; // バトルフィールドのX軸の中心
private static double   CenterY   = 0.0; // バトルフィールドのY軸の中心
private static double   robotSize = 0.0; // ロボットのサイズ

private static double   minX      = 0.0; // ロボットが移動できる最小のX座標
private static double   minY      = 0.0; // ロボットが移動できる最小のY座標
private static double   maxX      = 0.0; // ロボットが移動できる最大のX座標
private static double   maxY      = 0.0; // ロボットが移動できる最大のY座標

private static AdvancedRobot    myRobot;        // 自機オブジェクト

/**
 *  各種オブジェクト・変数の宣言や初期化を行う
 *  @param [in] r 自機オブジェクト
 */
public BattleField(AdvancedRobot r) {

    myRobot = r;

    //ロボットのサイズを計算
    robotSize = Math.min((myRobot.getWidth() / 2),(myRobot.getHeight() / 2));

    //バトルフィールドの大きさを計算

    width   = myRobot.getBattleFieldWidth();
    height  = myRobot.getBattleFieldHeight();

    //ロボットが移動可能な範囲を決定
    minX    = MINX;
    minY    = MINY;
```

```
        maxX    = width - MINX;
        maxY    = height - MINY;

        //中心座標を計算
        CenterX = maxX / 2;
        CenterY = maxY / 2;
    }

    /**
     * バトルフィールドの幅を返す
     * @return バトルフィールドの幅
     */
    public double getWidth() {
        return width;
    }

    /**
     * バトルフィールドの高さを返す
     * @return バトルフィールドの高さ
     */
    public double getHeight() {
        return height;
    }

    /**
     * ロボットが移動可能な最小の X 座標を返す
     *  @return 移動可能な X 座標の最小値
     */
    public double getMinX() {
        return minX;
    }

    /**
     * ロボットが移動可能な最小の Y 座標を返す
     * @return 移動可能な Y 座標の最小値
     */
    public double getMinY() {
        return minY;
    }

    /**
     * ロボットが移動可能な最大の X 座標を返す
     * @return 移動可能な X 座標の最大値
```

```
 */
public double getMaxX() {
    return maxX;
}

/**
 *  ロボットが移動可能な最大の Y 座標を返す
 *  @return 移動可能な Y 座標の最大値
 */
public double getMaxY() {
    return maxY;
}

/**
 *  バトルフィールドの X 軸における中心値をかえす
 *  @return X 軸の中心値
 */
public double getCenterX() {
    return CenterX;
}

/**
 *  ボトルフィールドの Y 軸における中心値を返す
 *  @return Y 座標の中心値
 */
public double getCenterY() {
    return CenterY;
}

/**
 *  ロボットのサイズを返す
 *  @return ロボットのサイズ
 */
public double getRobotSize() {
    return robotSize;
}

}
```

13.2 CircularFunction.java

```
package hoge8;

import robocode.*;

.
/** @file
 *  @brief 円形運動の予測砲撃を行うクラス
 *  @author wada
 */



/**
 *  円形運動の予測砲撃を行うクラス
 */
public class CircularFunction implements MotionFunction,Constants {
    Enemy target;   ///< 敵機オブジェクトを格納する
    double radius;
             ///< 円形運動の半径
    double InitialHeading;  ///< 円形運動の初期位相
    double CenterX;         ///< 円形運動の中心の X 座標
    double CenterY;         ///< 円形運動の中心の Y 座標
    double heading;         ///< 敵機の進行方向

    target = null;
    /**
     *  各種オブジェクト・変数の宣言や初期化を行う
     *  @param [in] er 敵機オブジェクト
     */
    public CircularFunction(Enemy er) {
        target = er;
        heading = target.getHeading();

        double signedRadius = 0;

        //敵機の速度と角速度の関係から円形運動の半径を計算する
        if(target.getAngularSpeed() != 0)
            signedRadius = target.getSpeed() / target.getAngularSpeed();
        else
            signedRadius = 0.0;
        radius = Math.abs(signedRadius);

        //ロボットの進行方向から初期位相と中心への角度を計算する
```

```
        InitialHeading = (signedRadius > 0 ) ? heading - 90 : heading + 90;
        double bearingToCenter = (signedRadius > 0) ? heading + 90 : heading - 90;

        //円形運動の中心座標を計算する
        CenterX = MyMath.calcNextX(target.getRelativeX(),radius,bearingToCenter);
        CenterY = MyMath.calcNextY(target.getRelativeY(),radius,bearingToCenter);

    }


    /**
     *  敵機に関する情報を更新する
     *  @param [in] t 敵機オブジェクト
     */
    public void update(Enemy t) {
        target = t;
        heading = target.getHeading();
        double signedRadius = 0;

        //敵機の速度と角速度の関係から円形運動の半径を計算する。
        if(target.getAngularSpeed() != 0)
            signedRadius = target.getSpeed() / target.getAngularSpeed();
        else
            signedRadius = 0.0;
        radius = Math.abs(signedRadius);

        //ロボットの進行方向から初期位相と中心への角度を計算する
        InitialHeading = (signedRadius > 0 ) ? heading - 90 : heading + 90;
        double bearingToCenter = (signedRadius > 0) ? heading + 90 : heading - 90;

        //円形運動の中心座標を計算する
        CenterX = MyMath.calcNextX(target.getRelativeX(),radius,bearingToCenter);
        CenterY = MyMath.calcNextY(target.getRelativeY(),radius,bearingToCenter);
    }

    /**
     *  刻時 t 後の敵ロボットの自ロボットに対する相対敵な X 座標を返す。
     *  @param [in] t 予測座標を計算したい刻時 t
     *  @return 刻時 t 後の相対的な X 座標
     */
    public double getNextX(double t) {
        return CenterX + radius * Math.sin(Math.toRadians(InitialHeading)
+ target.getAngularSpeed() * t);
```

```
    }

    /**
     *  刻時 t 後の敵ロボットの自ロボットに対する相対敵な Y 座標を返す。
     *  @param [in] t 予測座標を計算したい刻時 t
     *  @return 刻時 t 後の相対的な Y 座標
     */
    public double getNextY(double t) {
        return CenterY + radius * Math.cos(Math.toRadians(InitialHeading)
 + target.getAngularSpeed() * t);
    }

}
```

13.3 DirectFunction.java

```
package hoge8;
import robocode.*;
/** @file
 *  @brief 現在の座標に直接砲撃を行うクラス
 *  @author wada
 */

/**
 *  現在の座標に直接砲撃を行うクラス
 */
public class DirectFunction implements MotionFunction , Constants {
    Enemy target;    ///< 敵機オブジェクトを格納する
    double X;        ///< 敵機の自機に対する相対的な X 座標
    double Y;        ///< 敵機の自機に対する相対的な Y 座標

    target = null;

    /**
     *  各種オブジェクト・変数の宣言や初期化を行う
     *  @param [in] er 敵機オブジェクト
     */
    public DirectFunction(Enemy er) {
        target = er;
```

```
        //敵機の相対座標を代入する
        X = target.getRelativeX();
        Y = target.getRelativeY();

    }

    /**
     *  敵機に関する情報を更新する
     *  @param [in] er 敵機オブジェクト
     */
    public void update(Enemy er) {
        target = er;

        //敵機の相対座標を代入する
        X = target.getRelativeX();
        Y = target.getRelativeY();
    }

    /**
     * 敵機の相対的な X 座標を返す
     *  @return 相対的な X 座標
     */
    public double getNextX(double t) {
        return target.getRelativeX();
    }

    /**
     *  敵機の相対的な Y 座標を返す
     *  @return 相対的な Y 座標
     */
    public double getNextY(double t) {
        return target.getRelativeY();
    }
}
```

13.4 Enemy.java

```
package hoge8;
import robocode.*;

/**
```

```
 * MyClass - a class by (your name here)
 */
public class Enemy implements Constants {
    double bearing; ///< 敵機との角度 (360 度)
    double distance; ///< 敵機との距離

    double  current_energy; ///< 敵機の現在のエネルギー
    double last_energy; ///< 敵機の以前のエネルギー

    double current_speed; ///< 敵機の現在の速度
    double last_speed; ///< 敵機の以前の速度

    double  angularSpeed;

    double current_acc; ///< 敵機の現在の加速度
    double last_acc; ///< 敵機の以前の加速度

    double current_heading; ///< 敵機の現在の向き
    double last_heading; ///< 敵機の以前の向き

    double x; ///< 敵機の現在の X 座標
    double y; ///< 敵機の現在の Y 座標

    double RelativeX;               ///< 適機の現在の相対的な X 座標
    double RelativeY;               ///< 適期の現在の相対的な Y 座標

    double scantime; ///< 最後に敵機をスキャンした時間
    double last_scantime; ///< 一回前に敵機をスキャンした時間

    AdvancedRobot myrobot; ///< 自機オブジェクト


    /**
     *  各種変数に初期データを代入
     *  @param [in] e ScannedRobotEvent 型オブジェクト
     *  @param [in] me 自機オブジェクト
     */
    public Enemy(ScannedRobotEvent e,AdvancedRobot me) {
        ///< myrobot に実機のデータを代入
        myrobot = me;

        ///< 敵機との角度、距離を代入
        bearing = MyMath.to360(e.getBearing() + myrobot.getHeading());
```

```
        distance = e.getDistance();

        ///< 敵機のエネルギーを代入
        current_energy = e.getEnergy();
        last_energy = current_energy;

        ///<
        angularSpeed = 0.0;

        ///< 敵機の現在の速度、以前の速度を代入
        current_speed = e.getVelocity();
        last_speed = 0.0;

        ///< 敵機の現在の加速度を計算
        current_acc = (current_speed - last_speed)/e.getTime();
        last_acc = 0;

        ///< 敵機の向きを代入
        current_heading = e.getHeading();
        last_heading = 0;

        ///< 敵ロボットの相対座標を計算
        RelativeX = calcRelativeX(bearing,distance);
        RelativeY = calcRelativeY(bearing,distance);

        ///< 敵機の X,Y 座標を計算する
        x = calcx(myrobot.getX(),bearing,distance);
        y = calcy(myrobot.getY(),bearing,distance);

        ///< スキャンした時間を保存
        scantime = e.getTime();
    }

    /**
     *  敵機と自機の相対的な X 座標を返す
     *  @return 相対的な X 座標
     */
    public double getRelativeX() {
        return RelativeX;
    }

    /**
     *  敵機と自機の相対的な Y 座標を返す
```

```
     *  @return 相対的なY座標
     */
    public double getRelativeY() {
        return RelativeY;
    }

    /**
     *  敵機のX座標を返す
     *  @return 敵機のX座標
     */
    public double getX() {
        return x;
    }

    /**
     *  敵機のY座標を返す
     *  @return 敵機のY座標
     */
    public double getY() {
        return y;
    }

    /**
     * 敵機との角度(360度)を返す
     * @return 敵機との角度
     */
    public double getBearing() {
        return bearing;
    }

    /**
     *  敵機との距離をもどす
     *  @return 敵機との距離
     */
    public double getDistance() {
        return distance;
    }

    /**
     *  敵機の現在のエネルギーをかえす
     *  @return 敵機の現在のエネルギー
     */
    public double getEnergy() {
```

```
    return current_energy;
}

/**
 *  敵機の以前のエネルギーをかえす
 *  @return 敵機の以前のエネルギー
 */
public  double getLastEnergy() {
    return last_energy;
}

/**
 *  敵機の現在の速度を返す
 *  敵機の現在の速度
 */
public double getSpeed() {
    return current_speed;
}

/**
 *  敵機の角速度を返す
 *  @return   敵機の角速度
 */
public  double getAngularSpeed() {
    return angularSpeed;
}


/**
 *  敵機の現在の方向 (360 度) を返す
 *  @return 敵機の現在の方向
 */
public double getHeading() {
    return current_heading;
}

/**
 *  最後にスキャンした時間をかえす
 *  @return 最後にスキャンした時間
 */
public double getScanTime() {
    return scantime;
}
```

```
    /**
     *  一回前にスキャンした時間をかえす
     *  @return 一回前にスキャンした時間
     */
    public double getLastScanTime() {
        return last_scantime;
    }

    /**
     *  角速度を計算する
     *  @param [in] current_heading 現在の進行方向
     *  @param [in] last_heading 以前の進行方向
     *  @param [in] current_time 現在の時刻
     *  @param [in] last_time 以前の時刻
     *  @return 角速度
     */
    public double calcAngularSpeed(double current_heading,double last_heading
,double current_time,double last_time) {
        return Math.toRadians((current_heading - last_heading) / (current_time - last_time))
    }

    /**
     *  敵機の相対的なX座標を計算する
     *  @param [in] bearing 自機と敵機との間の角度
     *  @param [in] distance 自機と敵機との間の距離
     *  @return 相対的なX座標
     */
    public double calcRelativeX(double bearing,double distance) {
        return distance * Math.sin(Math.toRadians(bearing));
    }

    /**
      *  敵機の相対的なX座標を計算する
      *  @param [in] bearing 自機と敵機との間の角度
      *  @param [in] distance 自機と敵機との間の距離
      *  @return 相対的なX座標
      */
    public double calcRelativeY(double bearing,double distance) {
        return distance * Math.cos(Math.toRadians(bearing));
    }

    /**
```

```
 *  敵機のX座標をかえす
 *  @param [in] x 自機のX座標
 *  @param [in] bearing 自機と敵機との間の角度
 *  @param [in] distance 自機と敵機との間の距離
 *  @return 敵機のX座標
 */
public double calcx(double x,double bearing,double distance) {
    return x + calcRelativeX(bearing,distance);
}

/**
 *  敵機のY座標をかえす
 *  @param [in] y 自機のY座標
 *  @param [in] bearing 自機と敵機との間の角度
 *  @param [in] distance 自機と敵機との間の距離
 *  @return 敵機のY座標
 */
public double calcy(double y,double bearing,double distance) {
    return y + calcRelativeY(bearing,distance);
}

/**
 *  敵機のエネルギーを更新する
 *  @param [in] energy エネルギー変化分
 */
public void updateEnergy(double energy) {
    last_energy = current_energy;
    current_energy += energy;
}

/**
 *  敵機の情報の更新を実行
 *  @param [in] e ScannedRObotEvent型オブジェクト
 *  @param [in] me 自機オブジェクト
 */
public void update(ScannedRobotEvent e,AdvancedRobot me) {

    myrobot = me;

    ///< 更新時間の更新
    last_scantime = scantime;
    scantime = e.getTime();
```

```
        ///< 敵機の絶対角度 (360 度) を計算
        bearing = MyMath.to360(e.getBearing() + myrobot.getHeading());
        distance = e.getDistance();

        ///< 敵機のエネルギー残料を更新
        last_energy = current_energy;
        current_energy = e.getEnergy();

        ///< 敵機のスピードを更新
        last_speed = current_speed;
        current_speed = e.getVelocity();

        ///< 敵機の加速度を更新
        last_acc = current_acc;
        current_acc = (current_speed - last_speed)/(scantime - last_scantime);

        ///< 敵機の向きを更新
        last_heading = current_heading;
        current_heading = e.getHeading();


        angularSpeed = calcAngularSpeed(current_heading,last_heading,
scantime,last_scantime);

        RelativeX = calcRelativeX(bearing,distance);
        RelativeY = calcRelativeY(bearing,distance);

        ///< 敵機の座標を更新
        x = calcx(myrobot.getX(),bearing,distance);
        y = calcy(myrobot.getY(),bearing,distance);

    }

}
```

13.5 EnemyBullet.java

```
package hoge8;
import robocode.*;
/** @file
 *  @brief 敵機が発射した断間に付いてのクラスを記述したファイル
```

```
 *  @author
 */

/** @class
 *  @brief 敵機が発射した寒暖についてのクラス
 */
public class EnemyBullet implements Constants {
    double speed; ///< 弾丸のスピード
    double X; ///< 弾丸のX座標
    double Y; ///< 弾丸のY座標
    double distance; ///< 弾丸と自機との距離
    double direction; ///< 弾丸の進行方向
    double FireTime; ///< 弾丸の発射時刻
    double LastUpdateTime; ///< 最後に弾丸の情報が更新された時刻

    boolean hit = false; ///< この弾丸が自機にあたるかどうか(当たる:true、外れ
る:false)

    Enemy target; ///< 敵機のオブジェクト
    AdvancedRobot myrobot; ///< 実機のオブジェクト

    /**
     *  各種オブジェクト・変数の初期化を行う
     *  @param [in] er 敵機オブジェクト
     *  @param [in] me 自機オブジェクト
     */
    public  EnemyBullet(Enemy er,AdvancedRobot me) {
        ///< 敵機、実機のオブジェクトを代入
        target = er;
        myrobot = me;

        speed = calcBulletSpeed(target); ///< 弾丸のスピードを計算
        FireTime = er.getLastScanTime(); ///< 弾丸の発射時刻(最速時刻)
        LastUpdateTime = er.getScanTime(); ///< 更新時刻を更新

        ///< 敵機が自機に向けて弾丸を発射したとして進行方向を計算
        direction = MyMath.calcHeading(er.getX(),er.getY(),myrobot.getX(),myrobot.getY());

        ///< X座標、Y座標を計算
        X = MyMath.calcNextX(er.getX(),speed*(LastUpdateTime - FireTime),direction);
        Y = MyMath.calcNextY(er.getY(),speed*(LastUpdateTime - FireTime),direction);

        ///< 弾丸と自機との距離を計算
```

```
        distance = MyMath. calcDistance(X,Y,myrobot.getX(),myrobot.getY());

        ///< 自機に向けて発射したのであたる
        hit = true;

    }

    /**
     *  弾丸のスピードをかえす
     *  @return 弾丸のスピード
     */
    public double getSpeed() {
        return speed;
    }

    /**
     *  弾丸の進行方向をかえす
     *  @return 弾丸の進行方向
     */
    public double getDirection() {
        return direction;
    }

    /**
     *  弾丸と自機との距離をかえす
     *  @return 弾丸と自機との距離
     */
    public double getDistance() {
        return distance;
    }

    /**
     *  敵機のエネルギー変化から弾丸のスピードを計算する
     *  @param [in] target 敵ロボットのオブジェクト
     *  @return 弾丸のスピード
     */
    public double calcBulletSpeed(Enemy target) {
        return MyMath.calcBulletSpeed(target.getLastEnergy() - target.getEnergy());
    }

    /**
     *  弾丸がヒットするかどうかをかえす
     *  @return 弾丸がヒットするとき true を返す。
```

```
     *  それ以外の時はfalseを返す
     */
    public boolean getHit() {
        return hit;
    }

    /**
     *  弾丸の情報を更新する
     *  @param [in] me 自機のオブジェクト
     */
    public void update(AdvancedRobot me) {
        ///< 自機オブジェクトを代入
        myrobot = me;

        ///< X座標、Y座標を更新
        X = MyMath.calcNextX(X,(speed * (me.getTime() - LastUpdateTime)),direction);
        Y = MyMath.calcNextY(Y,(speed * (me.getTime() - LastUpdateTime)),direction);

        // 弾丸と自機との距離を更新
        distance = MyMath.calcDistance(X,Y,myrobot.getX(),myrobot.getY());

        // 弾丸の進行方向と弾丸と自機との間の角度が等しければヒットすると判断する
        if(Math.abs(direction -MyMath.calcHeading(X,Y,myrobot.getX(),myrobot.getY()))
 > SAFE_MARGIN) {
            hit = false;
        } else {
            hit = true;
        }

        ///< 更新時刻の更新
        LastUpdateTime = me.getTime();
    }

}
```

13.6 LinearFunction.java

```
package hoge8;

import robocode.*;
```

```
/** @file
 *  @brief 円形運動の予測砲撃を行うクラス
 *  @author wada
 */

/**
 *  円形運動の予測砲撃を行うクラス
 */
public class LinearFunction implements MotionFunction , Constants {
    Enemy target;    ///< 敵機オブジェクトを格納する
    double X;        ///< 敵機の自機に対する相対的なX座標
    double Y;        ///< 敵機の自機に対する相対的なX座標

    target = null;
    /**
     *  各種オブジェクト・変数の宣言や初期化を行う
     *  @param [in] t 敵機オブジェクト
     */
    public LinearFunction(Enemy t) {
        target = t;

        //敵機の相対座標を代入する
        X = target.getRelativeX();
        Y = target.getRelativeY();
    }

    /**
     *  敵機に関する情報を更新する
     *  @param [in] t 敵機オブジェクト
     */
    public void update(Enemy t) {
        target = t;

        //敵機の相対座標を代入する
        X = target.getRelativeX();
        Y = target.getRelativeY();
    }

    /**
     *  刻時t後の敵機の自機に対する相対敵なX座標を返す。
     *  @param [in] t 予測座標を計算したい刻時t
     *  @return 刻時t後の相対的なX座標
     */
```

```
    public double getNextX(double t) {
        return MyMath.calcNextX(X,target.getSpeed() * t,target.getHeading());
    }

    /**
    *  刻時 t 後の敵機の自機に対する相対敵な Y 座標を返す。
    *  @param [in] t 予測座標を計算したい刻時 t
    *  @return 刻時 t 後の相対的な Y 座標
    */
    public double getNextY(double t) {
        return MyMath.calcNextY(Y,target.getSpeed() * t,target.getHeading());
    }

}
```

13.7 MyMath.java

```
package hoge8;

/** @file
 *  @brief 各種数学的関数を集めたファイル
 *  @author wada
 */

/**
 *  各種数学的関数を集めたクラス
 */
public class MyMath implements Constants {
    /**
     *  渡された角度を 360 度形式に変換
     *  @param [in] angle 角度 (+-180 度形式)
     *  @return 360 度形式に変換された角度
     */
    public static double to360(double angle) {
        if(angle > 0)
            return angle % 360;
        else
            return ((angle % 360) + 360);
    }

    /**
```

```
 *  渡された二つの角度の差を+-180 度形式に変換
 *  @param [in] fromAngle 基準となる角度 (360 度形式)
 *  @param [in] toAngle 差を計算する対象の角度 (360 度形式)
 *  @return +-180 度形式に変換されたふたつの角度の差
 */
public static double to180(double fromAngle,double toAngle) {
    double tmp;  // 角度計算用のテンポラリ変数

    tmp  = (toAngle - fromAngle) % 360;

    if (tmp < -180)
        return tmp + 360;
    else if (tmp > 180)
        return tmp - 360;
    else
        return tmp;
}

/**
 *  指定された 2 点間の角度 (360 度) を計算する
 *  @param [in] fromX  基準となる点の X 座標
 *  @param [in] fromY  基準となる点の Y 座標
 *  @param [in] toX 目的となる点の X 座標
 *  @param [in] toY 目的となる点の Y 座標
 *  @return 指定された 2 点間の角度 (360 度形式)
 */
public static double calcHeading(double fromX,double fromY,double toX,double toY) {
    double tmpX; // 2 点の X 座標の差を格納
    double tmpY; // 2 点の Y 座標の差を格納

    //2 点の座標の差を計算
    tmpX = toX - fromX;
    tmpY = toY - fromY;

    //|2 点間の角度を計算
    if(toX == 0) {
        return (tmpY > 0.0 ? 360 : 180);
    } else {
        return (tmpX > 0.0 ? 90 : 270) - atan(tmpY / tmpX);
    }
}

/**
```

```
     * 指定された２点間の距離を計算する
     * @param [in] fromX 基準となる点のＸ座標
     * @param [in] fromY 基準となる点のＹ座標
     * @param [in] toX 目的となる点のＸ座標
     * @param [in] toY 目的となる点のＹ座標
     * @return 指定された２点間の距離
     */
    public static double calcDistance(double fromX,double fromY,double toX,double toY) {
        return Math.sqrt(Math.pow(toY - fromY, 2) + Math.pow(toX - fromX, 2));
    }

    /**
     * 現在のＸ座標、移動距離、進行方向から移動先のＸ座標を計算する
     * @param [in] x 現在のＸ座標
     * @param [in] distance 移動距離
     * @param [in] angle 進行方向
     * @return 移動先のＸ座標
     */
    public static double calcNextX(double x,double distance,double angle) {
        return x + distance * sin(angle);
    }

    /**
    * 現在のＸ座標、移動距離、進行方向から移動先のＹ座標を計算する
    * @param [in] y 現在のＹ座標
    * @param [in] distance 移動距離
    * @param [in] angle 進行方向
    * @return 移動先のＹ座標
    */
    public static double calcNextY(double y,double distance,double angle) {
        return y + distance * cos(angle);
    }

    /**
     * 弾丸のパワーから弾丸のスピードを計算する
     * @param [in] Power 弾丸のパワー
     * @return 弾丸のスピード
     */
    public static double calcBulletSpeed(double Power) {
        return MAX_BULLET_SPEED - (3 * Power);
    }

    /**
```

```
 *  弾丸のパワーから被弾時のダメージを計算する
 *  @param [in] Power 弾丸のパワー
 *  @return 被弾時のダメージ
 */
public static double calcBulletDamage(double Power) {
    if(Power > 1) {
        return 4 * Power + 2 * (Power - 1);
    } else {
        return 4 * Power;
    }
}

/**
 *  弾丸のパワーから着弾時の回復量を計算する
 *  @param [in] Power 弾丸のパワー
 *  @return 着弾時の回復量
 */
public static double calcBulletTrt(double Power) {
    return 3.0 * Power;
}

/**
 *  自機と敵機との距離から適切な弾丸のパワーを計算する
 *  @param [in] target 敵機オブジェクト
 *  @return 弾丸のパワー
 */
public static double calcBulletPower(Enemy target) {
    double distance; // 自機と敵機との距離

    distance = target.getDistance();

    //距離に応じてパワーを設定する
    if(distance > D_L) {
        return 0.5;
    } else if(distance < D_S) {
        return 3.0;
    } else {
        return 1.0;
    }
}


/**
```

```
 *  指定した角度 (360 度) の sin をとる
 *  @param [in] angle 角度 (360 度形式)
 *  @return sin(angle) の計算結果
 */
public static double sin(double angle) {
    return Math.sin(Math.toRadians(angle));
}

/**
 *  指定した角度 (360 度) の cos をとる
 *  @param [in] angle 角度 (360 度形式)
 *  @return cos(angle) の計算結果
 */
public static double cos(double angle) {
    return Math.cos(Math.toRadians(angle));
}


/**
 *  指定した角度 (360 度) の tan をとる
 *  @param [in] angle 角度 (360 度形式)
 *  @return tan(angle) の計算結果
 */
public static double tan(double angle) {
    return Math.tan(Math.toRadians(angle));
}


/**
 *  指定した値の atan をとる (角度)
 *  @param [in] data atan をとる値
 *  @return 角度 (360 度形式)
 */
public static double atan(double data) {
    return Math.toDegrees(Math.atan(data));
}


/**
 *  指定した値の asin をとる (角度)
 *  @param [in] data asin をとる値
 *  @return 角度 (360 度形式)
 */
```

```
    public static double asin(double data) {
        return Math.toDegrees(Math.asin(data));
    }

}
```

13.8 j02342_8.java

```
package hoge8;
import robocode.*;
import java.awt.Color;
//import java.awt.Color;

/** @file
 *  @brief j02342_8 ロボットのメイン処理
 *
 *  @author 和田政弘
 */

/** @class
 *  @brief j02342_8 ロボットのメイン処理を行う
 */
public class j02342_8 extends AdvancedRobot implements Constants {
    /**
     * 各種変数の宣言
     */

    private static Enemy target = null; // 敵機オブジェクト
    private static BattleField battleField = null; // バトルフィールドオブジェクト
    private static EnemyBullet bullet = null; // 弾丸オブジェクト

    private static boolean MotionDirection = true; // 進行方向フラグ（前身:true、後退:false)
        private static boolean GunMoving = false;
                // 砲台移動フラグ（移動中:true、停止中:false)
    private static boolean MotionState = false;             //
    private static boolean pos = false;

    private static LinearFunction lf = null;                // 直線運動予測オブジェクト
```

```
    private static CircularFunction cf = null;
                // 円形運動予測オブジェクト
    private static DirectFunction df = null;
                  // 直接射撃オブジェクト
    private static MotionFunction f = null;
                   // 運動予測オブジェクト格納用変数

    private static double avoid_distance;
                // 回避運動時に移動する距離
    private static double avoid_angle;                          // 回避運動時に
移動する方向

    private static double BulletPower = 0.0;
                  // 弾丸のパワー


    //メイン処理
    public void run() {
            //自機のカラー指定（白）
        setColors(Color.white,Color.white,Color.white);
        //初期化ルーチンの起動
        init();

        //各種カスタムイベントの定義

        //自機が移動可能な最小のX座標に到達
        addCustomEvent(
            new Condition("ReachMinX") {
                public boolean test() {
                    return isWest();
                };
            }
        );

        //自機が移動可能な最大のX座標に到達
        addCustomEvent(
            new Condition("ReachMaxX") {
                public boolean test() {
                    return isEast();
                };
            }
        );
```

```
        //自機が移動可能な最小のY座標に到達
        addCustomEvent(
            new Condition("ReachMinY") {
                public boolean test() {
                    return isSouth();
                };
            }
        );

        //自機が移動可能な最大のY座標に到達
        addCustomEvent(
            new Condition("ReachMaxY") {
                public boolean test() {
                    return isNorth();
                };
            }
        );

        //メインループ
        while(true) {
                //敵ロボット探索サブルーチンを起動
            search();

                //回避運動サブルーチンの起動
            move();

                //射撃サブルーチンの起動
            aim();

                //ロボットの行動を開始
            execute();
        }
    }

    /**
     *  各種オブジェクトの初期化処理を行う
    */

    public void init() {
            //回避距離、回避角度の設定
        avoid_distance = getHeight() / 2 + EXT_DIS;
        avoid_angle = 15;
```

```
        //レーダーと砲台の方向を下層から独立させる
        setAdjustGunForRobotTurn(true);
        setAdjustRadarForGunTurn(true);

        //battleField オブジェクトが存在指定ないなら作成する
        if(battleField == null) {
            battleField = new BattleField(this);
        }
    }

    /**
    *  砲撃用オブジェクトの初期化を行う
    *  @param [in] er 砲撃対象となる敵機オブジェクト
    */
    public void initFunction(Enemy er) {
        df = new DirectFunction(er);
        lf = new LinearFunction(er);
        cf = new CircularFunction(er);
    }

       /**
    *  砲撃用オブジェクトの更新を行う
    *  @param [in] er 砲撃対象となる敵機オブジェクト
    */
    public void updateFunction(Enemy er) {
        df.update(er);
        lf.update(er);
        cf.update(er);
    }

    /**
    * 敵ロボットの運動に適した射撃用オブジェクトを選択する
    * @param [in] er 砲撃対象となる敵機オブジェクト
    */
    public void selectFunction(Enemy er) {
                //敵機が円運動時
        if(isCircular(er)) {
            f = cf;
        }
                //敵機が等速直線運動時
        else if(isLinear(er)) {
            f = lf;
        }
```

```
            //敵機が減速、静止時
    else if(isDirect(er)) {
        f = df;
    }
}

/**
*  敵機を探索する
*/
public void search() {
        //レーダーを振る角度
    double angle  = 0.0;
        //現在レーダーの向いている方向
    double heading  = getRadarHeading() % 360;

    //敵機オブジェクトがないとき
    if(target == null) {
            //レーダーを３６０度回転させる
        setTurnRadarRight(360);
    } else {
            //一定時間敵ロボットをスキャンしていないとき
        if((getTime() - target.getScanTime()) > MAX_SCAN_AGE) {
            setTurnRadarRight(360);
        } else {
                //敵機を中心に一定幅でレーダーを振りつづける
            angle = MyMath.to180(heading,target.getBearing());
            angle += angle > 0 ? +HALF_SCAN_ARC : -HALF_SCAN_ARC;
            setTurnRadarRight(angle);
        }
    }
}


    /**
*  敵の弾丸を回避する
*/
public void move() {
    //既に bullet オブジェクトが存在するなら情報を更新する
    if(bullet != null) {
        bullet.update(this);
    }

    //もし bullet オブジェクトが自機にヒットするなら回避行動を実行する
```

```
        if(isHit(bullet)) {
            bullet = null;
            avoid(target);
        }
    }

    /**
    *  敵ロボットに照準をあわせる
    */
    public void aim() {

        //砲台を回転させる角度
        double tmp_angle;

        //砲撃用オブジェクトが既に存在しGunMovingがfalseのとき敵ロボットに照準
をあわせる
        if((f != null) && !GunMoving) {
            GunMoving = true;

            BulletPower = MyMath.calcBulletPower(target);
            tmp_angle = getEstimatedGunBearing(f,BulletPower,0,100);

            if(tmp_angle < 0) {
                GunMoving = false;
            } else {
                setTurnGunRight(MyMath.to180(getGunHeading(),tmp_angle));
            }
        }

        //照準があわせられたら砲撃を行う
        if((getGunTurnRemaining() <= 0.05) && GunMoving) {
            fire(BulletPower);
            f.incFc();
            f.decEn();
            GunMoving = false;
        }
    }

    /**
     *  移動中フラグを立て前進する
     *  @param [in] distance 距離
     */
    public void ahead(double distance) {
```

```
    if(distance > 0) {
        MotionDirection = true;
    } else {
        MotionDirection = false;
    }

    super.ahead(distance);
}

/**
 *  移動フラグを立て前進する
 *  @param [in] distance 距離
 */
public void setAhead(double distance) {
    if(distance > 0) {
        MotionDirection = true;
    } else {
        MotionDirection = false;
    }

    super.setAhead(distance);
}

    /**
*  レーダーを敵ロボットへ向ける
*  @param [in] distance 敵機との距離
*  @param [in] target 敵機オブジェクト
*  @param [in] me 自機オブジェクト
*/
public void  setTurnRadortoEnemy(double distance,Enemy target,AdvancedRobot me) {
            double currentAngle;  // 現在のレーダーの向いている方向
              double nextAngle;
      // レーダーを向けるべき方向
    double nextX;
           // 自機から敵機までのX座標の距離
    double nextY;
         // 自機から敵機までのY座標の距離

    currentAngle  = me.getRadarHeading();

    nextX = MyMath.calcNextX(me.getX(),distance,me.getHeading());
    nextY = MyMath.calcNextY(me.getY(),distance,me.getHeading());
```

```
        nextAngle = MyMath.calcHeading(nextX,nextY,target.getX(),target.getY());

        setTurnRadarRight(nextAngle - currentAngle);
    }

    /**
    *  自機の進行方向を敵機に対して 80 度の角度を保つように回転する
    */
    public void  setTurnRightAngle() {
        //敵機との相対角度を求める
        double RelativeAngle = MyMath.to180(target.getBearing(),getHeading());

        if(RelativeAngle < 0) {
            setTurnLeft((-90.0 - RelativeAngle) + 10);
        } else {
            setTurnRight((90.0 - RelativeAngle) - 10);
        }
    }

    //敵機の弾を回避するサブルーチン
        /**
    *  敵ロボットから発射された弾丸を回避する
    *  @param [in] target 敵ロボットのオブジェクト
    */
    /**/
    public void avoid(Enemy target) {
        double tmp_distance = avoid_distance;
        double tmp_angle = avoid_angle;

        if((Math.random() - DirectionRate) > 0) {
            tmp_distance = tmp_distance;
        } else {
            tmp_distance = -tmp_distance;
        }

        setTurnRadortoEnemy(tmp_distance,target,this);
        setAhead(tmp_distance);
    }

    /**
    *  敵ロボットから発射された弾丸が自ロボットに当たるかどうかを判断する
    *  @param [in] bullet 敵機から発射された弾丸のオブジェクト
    *  @return 当たる場合は ture を返す。
```

```
    *  外れる場合はfalseを返す。
    */
    public boolean isHit(EnemyBullet bullet) {
                        //弾丸が発射されてないときはfalseを返す
        if(bullet == null) {
            return false;
        }

                        //弾丸が自機に向かって移動しており回避可能な限界の距離まで
接近
        if(bullet.getHit() &&
 ((bullet.getDistance() / bullet.getSpeed()) < (Math.sqrt(2 * avoid_distance)))) {
            return true;
        } else {
            return false;
        }
    }

    /**
     *  自機がバトルフィールドの南側にいるかどうかを判断する
    *  @return 南側にいるときはtrueを返す。
    *  それ以外のと気はfalseを返す。
    */
    public boolean isSouth() {
            double w;   // バトルフィールドの幅
            double h;   // バトルフィールドの高さ
             double pxy; // 自機のX座標とY座標の和
             double mxy; // 自機のX座標とY座標の差
             double mwh; // バトルフィールドの幅と高さの差

        w = battleField.getMaxX();
        h = battleField.getMaxY();
        pxy = getX() + getY();
        mxy = getX() - getY();
        mwh =  w - h;

        return ((0.0 <= mxy) && (pxy < w) && (getY() < battleField.getMinY()));
    }

        /**
    *  自機がバトルフィールドの西側にいるかどうか判断する
    *  @return 西側にいるときはtrueを返す
    *  それ以外の時はfalseを返す
```

```
*/
public boolean isWest() {
        double w;   // バトルフィールドの幅
        double h;   // バトルフィールドの高さ
         double pxy; // 自機のX座標とY座標の和
         double mxy; // 自機のX座標とY座標の差
         double mwh; // バトルフィールドの幅と高さの差

    w = battleField.getMaxX();
    h = battleField.getMaxY();
    pxy = getX() + getY();
    mxy = getX() - getY();
    mwh =  w - h;

    return ((pxy < h) && (mxy < 0.0) && (getX() < battleField.getMinX()));
}

    /**
*  自機がバトルフィールドの東側にいるかどうか判断する
*  @return 東側にいるときはtrueを返す
*  それ以外の時はfalseを返す
*/
public boolean isEast() {
        double w;   // バトルフィールドの幅
        double h;   // バトルフィールドの高さ
         double pxy; // 自機のX座標とY座標の和
         double mxy; // 自機のX座標とY座標の差
         double mwh; // バトルフィールドの幅と高さの差

    w = battleField.getMaxX();
    h = battleField.getMaxY();
    pxy = getX() + getY();
    mxy = getX() - getY();
    mwh =  w - h;

    return ((mwh <= mxy) && (pxy >= w) && (getX() > battleField.getMaxX()));

}

    /**
*  自機がバトルフィールドの北側にいるかどうか判断する
*  @return 北側にいるときはtrueを返す
*  それ以外の時はfalseを返す
```

```
*/
public boolean isNorth() {
        double w;   // バトルフィールドの幅
        double h;   // バトルフィールドの高さ
         double pxy; // 自機の X 座標と Y 座標の和
         double mxy; // 自機の X 座標と Y 座標の差
         double mwh; // バトルフィールドの幅と高さの差

    w = battleField.getMaxX();
    h = battleField.getMaxY();
    pxy = getX() + getY();
    mxy = getX() - getY();
    mwh =  w - h;

    return ((mxy < mwh) && (pxy >= h) && (getY() > battleField.getMaxY()));

}

    /**
*  敵機が円形運動かどうかを判断する
*  @param [in] er 敵ロボットのオブジェクト
*  @return 敵機が円形運動を行ってるとき true を返す。
*  それ以外の時は false を返す。
*/
public boolean isCircular(Enemy er) {
           //角速度が 0 でないときは円運動を行っていると判断する
    if((er.getAngularSpeed() != 0)) {
        return true;
    } else
        return false;
}

    /**
*  敵機が等速直線運動かどうかを判断する
*  @param [in] er 敵機のオブジェクト
*  @return 敵機が等速直線運動を行ってるとき true を返す。
*  それ以外の時は false を返す。
*/
public boolean isLinear(Enemy er) {
           //加速度が 0 の時等速直線運動であると判断する
    if((er.getAcc() == 0))
        return true;
    else
```

```
            return false;
    }

        /**
    *  敵機が減速中かどうかを判断する
    *  @param [in] er 敵機のオブジェクト
    *  @return 敵機が減速中のとき true を返す。
    *  それ以外の時は false を返す。
    */
    public boolean isDirect(Enemy er) {
        if((er.getAcc() < 0))
            return true;
        else
            return false;
    }


    /**
    *  予測砲撃を行う際に砲台を回転させる角度を計算する
    *  @param [in] f 砲撃用オブジェクト
    *  @param [in] BP 弾丸のパワー
    *  @param [in] min 着弾までの最小の刻時
    *  @param [in] max 着弾までの最大の刻時
    *  @return 予測した座標がバトルフィールド内の時は砲台を回転させる角度（0～360
度）を返す。
    *  バトルフィールド外のときは-1 をかえす。
    */
    public double getEstimatedGunBearing(MotionFunction f,double BP,
double min,double max) {
        double t;   // 着弾までにかかる刻時
        double dx;  // 刻時 t 後の X 座標
        double dy;
          // 刻時 t 後の Y 座標
                  double b;   // 砲台を回転させる角度

        t = getEstimatedImpactTime(f,BP,min,max);
        dx = f.getNextX(t);
        dy = f.getNextY(t);

        b = 90 - Math.toDegrees(Math.atan2(dy,dx));

        dx += getX();
        dy += getY();
```

```
        if((dx < 0) || (battleField.getWidth() < dx) || (dy < 0)
 || (battleField.getHeight() < dy)) {
            return -1;
        } else {
            return MyMath.to360(b);
        }
    }

        /**
     *  敵ロボットに着弾するまでの時間を計算する
     *  @param [in] f 砲撃用オブジェクト
     *  @param [in] BP 弾丸のパワー
     *  @param [in] min 着弾までの最小の刻時
     *  @param [in] max 着弾までの最大の刻時
     *  @return 敵ロボットに着弾するまでの刻時を返す。
     */
    public double getEstimatedImpactTime(MotionFunction f,double BP,
double min,double max) {
        double t0;   // 方程式の解が存在する範囲の最下限
        double t1;   // 方程式の解が存在する範囲の最上限
        double d0;   // t0 における弾丸と敵機との距離

        t0 = min;
        t1 = max;
        d0 = getEstimatedDistance(f,BP,t0);

        for(int i = 0;i < 20;i++) {
            if(Math.abs(t0 - t1) < MISS)
                break;
            double d1 = getEstimatedDistance(f,BP,t1);
            double t2 = (t0 * d1 - t1 * d0)/(d1 - d0);
            t0 = t1;
            t1 = t2;
            d0 = d1;
        }
        return t1;
    }

        /**
     *  弾丸と敵機との距離を計算する
     *  @param [in] f 砲撃用オブジェクト
     *  @param [in] BP 弾丸のパワー
```

```
*  @param [in] t 距離を計算する刻時
*  @return 刻時 t における弾丸と敵機との距離を返す。
*/
public double getEstimatedDistance(MotionFunction f,double BP,double t) {
    double bs;  // 弾丸のスピード
    double dx;  // 刻時 t における敵機の X 座標
    double dy;  // 刻時 t における敵機の Y 座標

    // 弾丸のパワーから弾丸のスピードを計算
    bs = MyMath.calcBulletSpeed(BP);

    dx = f.getNextX(t);
    dy = f.getNextY(t);

    return Math.sqrt(dx*dx + dy*dy) - bs * t;
}


/**
 * レーダーで敵機を発見した際に呼び出され、敵機オブジェクトの作成・
 * 更新、砲撃用オブジェクトの作成・選択、レーダーの回転、敵機の砲撃判定を行う
 * @param [in] e ScannedRobotEvent 型のオブジェクト
 */
public void onScannedRobot(ScannedRobotEvent e) {
    double difEnergy;  //敵ロボットのエネルギーの差を保存

    difEnergy = 0.0;

    //敵機オブジェクトが作成されてなかったら作成する
    if(target == null) {
        target = new Enemy(e,this);
    } else {
        //敵機の情報を更新する
        target.update(e,this);

        //砲撃用オブジェクトが作成されてなかったら初期化する
        if(f == null) {
            initFunction(target);
        }
        //砲撃用オブジェクトの情報を更新する
        else {
            updateFunction(target);
        }
```

```
            //適切な砲撃用オブジェクトを選択する
            selectFunction(target);
            //レーダーを敵ロボットに向ける
            setTurnRightAngle();

            //敵ロボットのエネルギー差を計算
            difEnergy = target.getLastEnergy() - target.getEnergy();

            //弾丸オブジェクトが作成されておらずエネルギー差があった場合、砲撃が
行われたと判断する
            if(bullet == null && (0 < difEnergy)) {
                bullet = new EnemyBullet(target,this);

            }
        }
    }


    /**
     * 自機が被弾した際に呼び出され、bullet オブジェクトを破棄し、敵機オブジェクト
のエネルギーを更新する
     *  @param [in] e HitByBulletEvent HitByBulletEvent 型のオブジェクト
     */
    public void onHitByBullet(HitByBulletEvent e) {
        bullet = null;
        target.updateEnergy(MyMath.calcBulletTrt(e.getPower()));
    }

    /**
     *  敵機に着弾した際に呼び出され、敵機オブジェクトのエネルギーを更新する
     *  @param [in] event BulletHitEvent 型のオブジェクト
     */
    public void onBulletHit(BulletHitEvent event) {
        target.updateEnergy(event.getEnergy() - target.getEnergy());
    }

        /**
    *  勝利した時に呼び出され、各種オブジェクトを破棄する
    *  @param [in] event WinEvent 型のオブジェクト
    */
    public void onWin(WinEvent event) {
        target = null;
```

```
        battleField = null;
        bullet = null;

        MotionDirection = true;
        MotionState     = true;
        GunMoving = false;

        f = null;
    }

        /**
    *  敗北した時に呼び出され、各種オブジェクトを破棄する
    *  @param [in] event DeathEvent 型のオブジェクト
    */
    public void onDeath(DeathEvent event) {
        target = null;
        battleField = null;
        bullet = null;

        MotionDirection = true;
        MotionState     = true;
        GunMoving = false;

        f = null;
    }

        /**
    *  各種カスタムイベントが定義されている
    *  @param [in] ev CustomEvent 型のオブジェクト
    */
    public void onCustomEvent(CustomEvent ev) {

        Condition cd;     // カスタムイベントの発生状態を保存する
        double angle;     //
        double heading;   // 自ロボットが向いている方向
        double distance; // 移動する距離
        double aaa;       // 進路を変更する角度

        cd =  ev.getCondition();
        heading = getHeading();
        distance = (MotionDirection) ? -120 : 120;
        aaa = 30;
```

```
        // 移動できる最小の X 座標に達したら呼び出される
        if (cd.getName().equals("ReachMinX")) {
            if(getY() > battleField.getCenterY())
                setTurnRight(aaa);
            else
                setTurnRight(-aaa);
            bullet = null;
            ahead(distance);
        }
                // 移動できる最大の X 座標に達したら呼び出される
        else if (cd.getName().equals("ReachMaxX")) {
            if(getY() > battleField.getCenterY())
                setTurnRight(-aaa);
            else
                setTurnRight(aaa);
            bullet = null;
            setTurnRight(30);
            ahead(distance);
        }
        // 移動できる最小の Y 座標に達したら呼び出される
        else if (cd.getName().equals("ReachMinY")) {
            if(getY() > battleField.getCenterX())
                setTurnRight(aaa);
            else
                setTurnRight(-aaa);
            bullet = null;
            setTurnRight(30);
            ahead(distance);
        }
        // 移動できる最大の Y 座標に達したら呼び出される。
        else if (cd.getName().equals("ReachMaxY")) {
            if(getY() > battleField.getCenterX())
                setTurnRight(-aaa);
            else
                setTurnRight(aaa);
            bullet = null;
            setTurnRight(30);
            ahead(distance);
        }

    }

}
```

13.9 Constants.java

```
package hoge8;

/** @file
 *  @brief 他のクラスで必要となる定数を定義するインターフェース
 *  @author wada
 */

/** @class
 *  @brief 他のクラスで必要となる定数を定義するインターフェース
 */
public interface Constants {

    public static final double  MAX_BULLET_SPEED  = 20;   ///< 弾丸の最高速
度
    public static final double  MAX_SPEED         = 8.0;   ///< ロボットの最
高速度
    public static final double  SCAN_ARC          = 45.0;   ///< レーダーの
振り幅
    public static final double  HALF_SCAN_ARC     = SCAN_ARC / 2; ///< レー
ダーの振り幅（片側）
    public static final long    MAX_SCAN_AGE      = 8;   ///< スキャンした情
報の寿命
    public static final double  MINX   = 20.0;   ///< ロボットが移動する最小の
X座標
    public static final double  MINY   = 20.0;   ///< ロボットが移動する最小の
Y座標
    public static final double  PI   = 3.1415;   ///< πの値
    public static final double  SAFE_MARGIN   = 20;    ///< 発射された弾丸を
被弾するかどうか判断するマージン
    public static final double  AVOID_MARGIN   = 20;   ///< 回避行動を開始す
る弾丸到着までの時間
    public static final double  EXT_DIS   = 20;   ///< 回避行動の際に移動する
マージン
    public static final double MISS   = 0.001;        ///< 方程式の解の誤差の
範囲
    public static final double D_L   = 700;          ///< 遠距離と中距離との
境界値
    public static final double D_S   = 150;          ///< 中距離と近距離との
```

境界値
}

13.10 MotionFunction.java

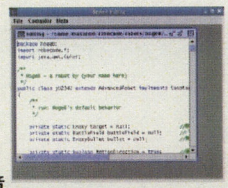
```
package hoge8;

import robocode.*;

/** @file
 *  @brief 砲撃用クラスのインターフェース
 *  @author
 */

/**
 * 砲撃用クラスのインターフェース
 */
public interface MotionFunction {
    double getNextX(double t); //刻時 t 後の X 座標を返す
    double getNextY(double t); //刻時 t 後の Y 座標を返す
    void update(Enemy t);      //敵機の情報を更新する
}
```

目的

. Robocodeを用いてJava言語とオブジェクト指向プログラミングを理解する
. Linux上での開発を通してUNIXの使用方法を学ぶ

実験環境

. OS
Fedora Core 1 Linux
. 開発言語
Java言語 J2SDK 1.4.2.04
. 開発環境
ROBOCODE開発環境下で開発



今回作成したロボットの戦略と戦術

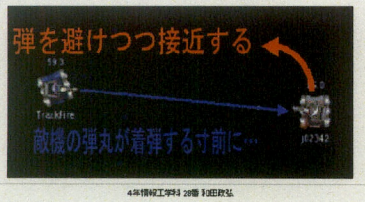
. 戦略
  . 確実に弾を当てる
. 戦術
  . 敵機に十分接近し、狙いをさだめてから弾を発射する

戦略と戦術の実現方法

. 敵機の移動を予測
. 弾丸の着弾寸前に回避
. 回避時に敵機へ前進

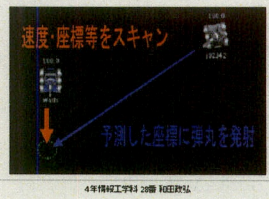
回避運動

. エネルギー差から弾丸の発射を感知
. 弾速と敵機との距離から着弾時刻を計算
. 着弾寸前に敵機へ向けて回避



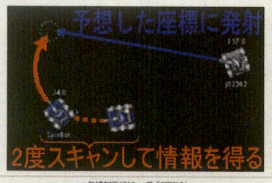
運動予測(1)-等速直線運動

. 敵機の速度と方向をスキャン
. 等速直線運動を仮定し、発射した弾丸との距離が0となる座標を求める



運動予測(2)-円形運動

. 敵機の速度と方向を2度スキャンする
. 角速度・進行方向・中心座標・半径を計算
. 円形運動を仮定し、発射した弾丸との距離が0となる座標を求める



実装した機能

. 敵機の運動の予測を実現
　→円形運動
　→等速直線運動
. 敵機への接近を実現
　→回避時に前進

実験の問題点

. 敵機の射撃判断アルゴリズムにバグ
　→敵機の射撃を感知できなくなる
. 見境なく射撃を行う
　→自機のエネルギーを無駄に使用

感想

. Java言語・オブジェクト指向プログラミングを理解するという目標を達成
. オブジェクト指向の強力さを実感
　→現実のシミュレーションが容易

最後に

和田政弘のページ

http://nt.hakodate-ct.ac.jp/~j02342/index.html