Aufgabe 5 : Pong

33.Bundeswettbewerb Informatik 2014/'15

Der Script-Tim

Inhaltsverzeichnis

1 Allgemein

Mein Benutzername im Turniersystem ist „TiHoX“.

2 Problem

Das Problem hierbei besteht darin, dass für das erfolgreiche Abwehren des Balles dessen genaue Position und Bewegungsrichtung ermittelt werden muss, aber weder Geschwindigkeit noch Bewegungswinkel/Steigungsfaktor und die Position des Balles lediglich auf Ganzzahlen gerundet übergeben ist. Es ist daher notwendig, diese Informationen durch Vergleichen der zurückgelegten Punkte zu ermitteln. Dazu müssen noch die Spielfeldbegrenzungen berücksichtigt werden. Zudem wird eine allein defensiv reagierende KI nicht weit kommen. Das Vorausberechnen eines gegnerischen Schlages ist durch Zufallswerte hier beinahe unmöglich, es sei denn, er schlägt mit dem mittleren Drittel. Die KI muss die verschiedenen Runden voneinander trennen können bzw. gezielt auf eine Veränderung der Punktzahl achten, da sich für die KI nichts außer die Insgesamtpunktzahl verändert.

3 Lösungsidee

3.1 Errechnen von Steigungsfaktor und Position des Balles

Das Herzstück meiner KI besteht aus der Errechnung der Position und der Steigungsgerade des Balles aus den letzten 3 bis 10 gespeicherten Positionen.

Da ich zu Recht einen linearen Zusammenhang der Punkte vermuten kann, bietet sich zur Errechnung der Position und des Steigungsfaktors eine lineare Regression an.

Welche Punkte? Für die Regression verwende ich die letzten 3, 5, 7 oder 10 Punkte des Balles, sofern diese x-y-linear sind, d.h. sich in einer Linie befinden;

$$x_i \leq x_{i+1} \leq x_{i+2} \ldots \text{ ODER } x_i \geq x_{i+1} \geq x_{i+2} \ldots$$

UND

$$y_i \leq y_{i+1} \leq y_{i+2} \ldots \text{ ODER } y_i \geq y_{i+1} \geq y_{i+2} \ldots$$

Eine Regression aus nur drei Punkten ist zwar ungenau, wird aber auch nur dann ausgeführt, wenn noch keine fünf linearen Punkte gesammelt wurden, damit eine schnellere Reaktion auf einen entgegenkommenden Ball möglich ist und sich der Schläger schon einmal in die grobe Richtung bewegen kann.

Woher? Die Positionen werden in zehn Membervariablen gespeichert. Bei jedem neuen Zug werden die Punkte „weitergreicht“;

$$position_5 = position_4, position_4 = position_3, \ldots\ position_1 = aktuellePosition$$

Hierbei entfällt logischer Weise die letzte Position.

Wie? Die Regression errechnet die Variablen a und b, die eingesetzt in die allgemeine Formel der linearen Steigung

$$f(x) = a + b \cdot x$$

eine Linie ergeben, die näherungsweise alle Punkte durchläuft.

a und b ergeben sich durch folgende Formeln:

$$a(n, (x_1|y_1)\dots(x_n|y_n)) = \frac{n\sum_{i=1}^n x_i y_i - \sum_{i=1}^n x_i \sum_{i=1}^n y_i}{n\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2} \quad (1)$$

$$b(n, (x_1|y_1)\dots(x_n|y_n)) = \frac{\sum_{i=1}^n x_i^2 \sum_{i=1}^n y_i - \sum_{i=1}^n x_i \sum_{i=1}^n x_i y_i}{n\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2} \quad (2)$$

Da ich den Punkt errechnen will, an dem der Ball auf die Wand trifft - $f(0)$ - gilt:

$$\text{auftreffpunkt} = f(0) = a + \cancel{b \cdot 0} = a$$

a könnte > 60 oder < 1 sein, die Spielfeldbegrenzungen werden später behandelt.

Theoretisch wäre meine KI so in der Lage, jeden beliebigen Ball aus jedem Winkel aufzuhalten, wenn dessen Geschwindigkeit nicht zunehmen würde.

3.2 Defensivverhalten

Zu mir hin Wenn sich der Ball in Richtung des Schlägers meiner KI bewegt, errechnet sie wie beschrieben aus den letzten 3 bis 10 gespeicherten Positionen die erwähnte Regressionsgerade und damit den Auftriffspunkt auf ihrer Wand. Daraufhin versucht sie, diesen Punkt zu erreichen. Zusätzlich kommt hierbei noch die Angriffslogik (siehe unten) zum Einsatz.

Von mir weg Wenn sich der Ball von meiner KI entfernt, berechnet sie den Auftreffpunkt beim Gegner. Wenn sich dort der gegnerische Schläger befindet, berechnet sie entsprechend dem Drittel den Abprallwinkel und den Auftreffpunkt bei ihr selbst und begibt sich dorthin; sie berechnet den Schlag des Gegners quasi voraus. Dies ist nur möglich, wenn sich der gegnerische Schläger bereits früh an den Auftreffpunkt begibt und sich das Drittel, das auf dem Auftreffpunkt liegt, nicht verändert. Jetzt verstehe ich den Satz in der Aufgabenstellung:

> Man kann dem Gegner eine Überraschung bescheren, indem man sich im letzten Moment entscheidet, den Ball nicht mit der Mitte, sondern mit einem äußeren Drittel anzunehmen.

→ Meine KI kann indirekt in eine falsche Richtung gelockt werden; dies bin ich bereit, in Kauf zu nehmen.

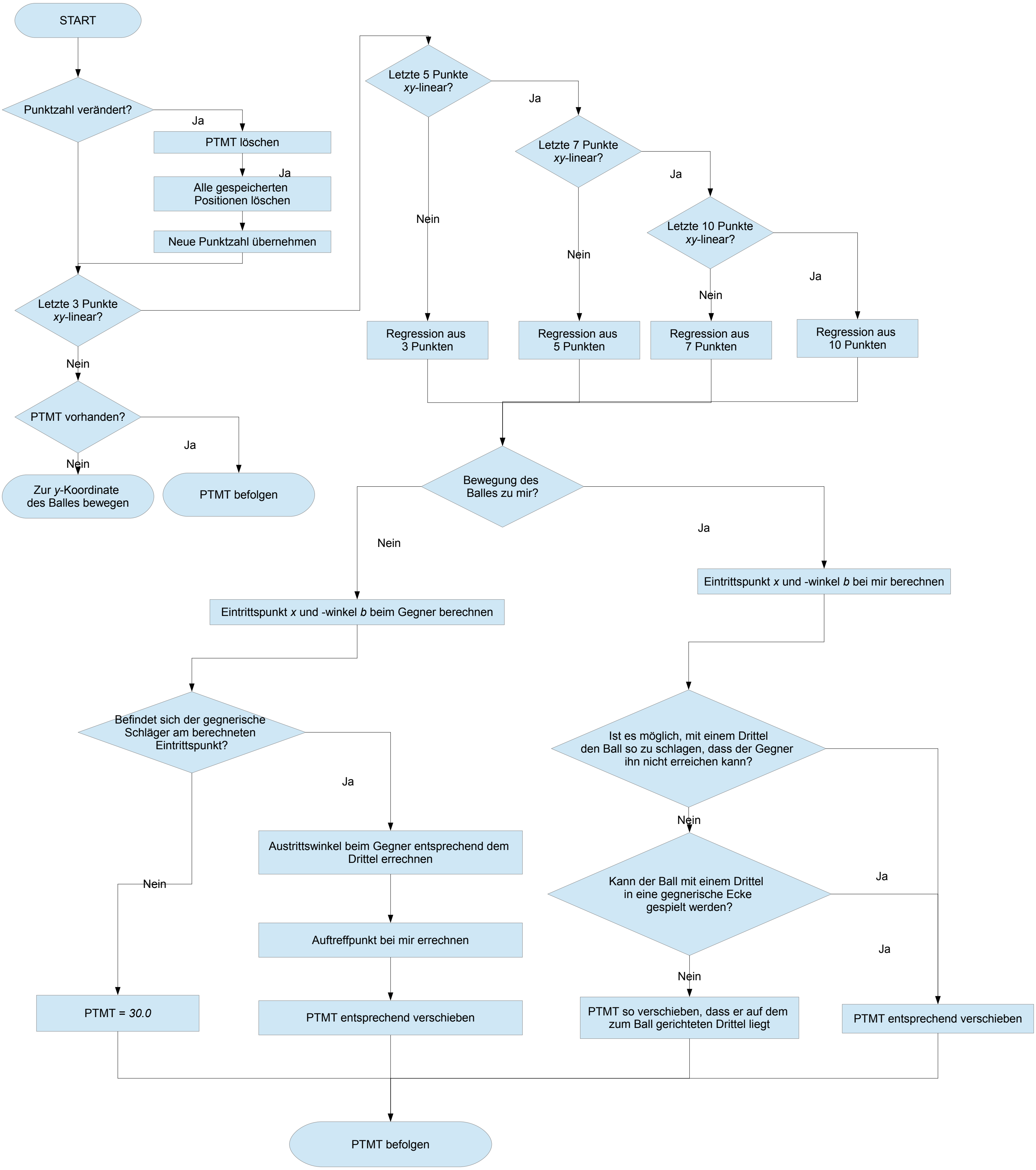
3.3 Offensivverhalten; Angriffslogik

Meine KI ist nicht allein defensiv; ich habe ein Feature implementiert, das ich „Todesstoß“ nenne. Wenn der Ball auf meine KI zukommt, überprüft sie, ob der Ball so gespielt werden kann, dass der gegnerische Schläger nicht genug Zeit hat, den Punkt zu erreichen, an dem der Ball voraussichtlich auftreffen wird und das Abwehren somit unmöglich ist. Das Problem besteht darin, dass die äußeren Drittel ja einen zufälligen Winkel erzeugen, welche folglich nicht berechenbar sind. Meine KI verwendet für ihre Berechnungen den Mittelwert aus $-a$ und $\tan(70°)$ bzw. $\tan(-70°)$, sprich, den möglichen Abprall-Winkeln. Deshalb ist klar, dass Todesstöße nicht gezielt einsetzbar sind, sondern eher eine Möglichkeit darstellen, die eintreffen kann, aber nicht muss. Der Todesstoß kommt vor allem dann zum Einsatz, wenn die Bälle mit der Zeit immer schneller werden. Bis dahin verlässt sich meine KI auf ihre Defensivlogik. Wenn ein Todesstoß nicht möglich ist, wird überprüft, ob der Ball nicht in eine Gegnerische Ecke gespielt werden kann, da der Gegner mit zunehmender Ballgeschwindigkeit weniger Zeit hat, sich in die weit entfernten Ecken zu bewegen. Hat er den Ball in der Ecke abgewehrt, kann meine KI den zurückkommenden Ball in die entgegengesetzte Ecke spielen, da dem Gegner zu wenig Zeit bleibt, den kompletten Weg von einer in die andere Ecke zurückzulegen.

4 Algorithmus

Es folgt ein Flussdiagramm des Algorithmus' meiner KI. $PTMT$ bezeichnet hier eine Membervariable des selben Namens, die Zug übergreifend den errechneten Punkt speichert, an den sich der eigene Schläger bewegen soll. (**P**oint **T**o **M**ove **T**o).

Flussdiagramm KI „TiHoX“ v8 – Der Script-Tim



Start- bzw. Endpunkt. Eine Bewegung (*nachUnten()*/*nachOben()*) ist einem Endpunkt gleichzusetzen, da nach der durch „IchFalleVomDreirad“ veranlassten Änderung der Spiellogik bei jeder Bewegung die Ausführung der *zug*-Funktion sofort beendet wird.

5 Umsetzung

5.1 Programmiersprache

Alle meine (4) KIs habe ich in Java geschrieben.

5.2 Funktionen

Neben den Funktionen *getMe()*,*getEnemy()* und *getBall()* der Beispiel-KI habe ich mehrere andere Funktion definiert

ausgabe gibt einen Text unter Angabe der Rundennummer aus (das hat mir an *zug::ausgabe* gefehlt)

```
/* Gibt unter Angabe der Rundennummer den Text 'text' aus */
public void ausgabe(String text, Spiel.Zug zug){
    zug.ausgabe("[" + this.runde + "] " + text);
}
```

doMove führt eine Bewegung zu einem übergebenen Punkt *ptmt* aus.

```
/* Bewegt den Schläger in Richtung des ptmt */
public void doMove(double ptmt, Spiel.Zustand.Schlaeger me, Spiel.Zug zug){
    double meinePosition = me.yKoordinate() + 2.5;

    ptmt = Math.round(ptmt);
    meinePosition = Math.round(meinePosition);

    if (meinePosition < ptmt) zug.nachUnten();
    if (meinePosition > ptmt) zug.nachOben();
    return; //Nötig; falls meinePosition == ptmt
}
```

auftreffpunkt errechnet den Auftreffpunkt des Balles unter Einbeziehung der Banden

```
/* Berechnet den Punkt, an dem der Ball an der Wand auftreffen wird */
public double auftreffpunkt(double punkt){
    while (punkt < 0 || punkt > 60){

        if(punkt < 0){
            punkt = -punkt;
        }
        if (punkt > 60){
            punkt = 60 - (punkt - 60);
        }
    }
    return punkt;
}
```

inEinerReihe überprüft, ob die (in einem ArrayList-Container) übergebenen Punkte x-y-linear sind, d.h., sich in einer durchgehenden Reihe/Linie befinden.

```java
/* Gibt zurück, ob die Punkte in 'liste' xy-linear sind */
public boolean inEinerReihe(ArrayList<Spiel.Zustand.Ball> liste){

    for(int i = 0; i < liste.size(); i++){
        if (liste.get(i) == null) return false;
    }

    boolean OKx1 = true;
    boolean OKx2 = true;
    boolean OKy1 = true;
    boolean OKy2 = true;

    for(int i = 0; i < liste.size(); i++){
        if (i >= 2){
            if (liste.get(i).xKoordinate() > liste.get(i-2).xKoordinate()) OKx1 = false;
        }
    }

    for(int i = 0; i < liste.size(); i++){
        if (i >= 2){
            if (liste.get(i).xKoordinate() < liste.get(i-2).xKoordinate()) OKx2 = false;
        }
    }

    if (!OKx1 && !OKx2) return false;

    for(int i = 0; i < liste.size(); i++){
        if (i >= 2){
            if (liste.get(i).yKoordinate() > liste.get(i-2).yKoordinate()) OKy1 = false;
        }
    }

    for(int i = 0; i < liste.size(); i++){
        if (i >= 2){
            if (liste.get(i).yKoordinate() < liste.get(i-2).yKoordinate()) OKy2 = false;
        }
    }

    if (!OKy1 && !OKy2) return false;
    return true;
}
```

Auf die Funktionsweise der einzelnen Funktionen gehe ich später genauer ein.

5.3 Membervariablen

Membervariablen sind die einzige Möglichkeit, Informationen Zugübergreifend zu speichern. Hierzu gehören z.B. die letzten 10 Positionen und der *PTMT*.

```

    public int runde = 0;

    public int verschiebung = 1;

    public Spiel.Zustand.Ball position1;
    public Spiel.Zustand.Ball position2;
    public Spiel.Zustand.Ball position3;
    public Spiel.Zustand.Ball position4;
    public Spiel.Zustand.Ball position5;
    public Spiel.Zustand.Ball position6;
    public Spiel.Zustand.Ball position7;
    public Spiel.Zustand.Ball position8;
    public Spiel.Zustand.Ball position9;
    public Spiel.Zustand.Ball position10;

    public double ptmt;
    public int punktzahl;
```

5.4 *AI::zug()*

Die Ausführung der KI beginnt hier

```
    public void zug(int id, Spiel.Zustand zustand, Spiel.Zug zug){
```

5.4.1 Punktzahl verändert?

Zuerst wird geprüft, ob sich die aktuelle Punktzahl von der des letzten Zuges unterscheidet → neue Runde. Dafür habe ich die Membervariable *AI::punktzahl* definiert; sie enthält den letzten Punktestand.

```
        /* Bei neuer Runde alles löschen */
        int punktzahl = this.getMe(zustand, id).punktzahl() + this.getEnemy(zustand, id).punktzahl();
        if (punktzahl != this.punktzahl){
            this.position10 = null;
            this.position9 = null;
            this.position8 = null;
            this.position7 = null;
            this.position6 = null;
            this.position5 = null;
            this.position4 = null;
            this.position3 = null;
            this.position2 = null;
            this.position1 = null;
            this.ptmt = 0;
            this.punktzahl = punktzahl;
```

```
41
42             this.ausgabe("========== Punktzahl geändert", zug);
43         }
```

Ich betrachte hierbei logischer Weise die Insgesamtpunktzahl, die sich aus Addition der einzelnen Schlägerpunkte ergibt [Z.27]. Sollte sie sich von der letzten Punktzahl unterscheiden, hat anscheinend eine neue Runde angefangen und alle gespeicherten Punkte werden gelöscht [Z.29-38] sowie der *PTMT* [Z.39]. Die neue Punktzahl wird übernommen [Z.40]. Darauf folgt eine (für den Algorithmus unnötige) Benachrichtigung an mich, dass sich die Punktzahl geändert hat (was ich wahrscheinlich auch von alleine gemerkt hätte)[Z.42].

5.4.2 Punkte „weiterreichen“

Unabhängig davon werden die gespeicherten Punkte um die neue (aktuelle) Position des Balles erweitert; dafür entfällt die letzte gespeicherte Position $position_{10}$ - falls vorhanden - und die Werte dazwischen werden „weitergereicht“.

```
45         this.position10 = this.position9;
46         this.position9 = this.position8;
47         this.position8 = this.position7;
48         this.position7 = this.position6;
49         this.position6 = this.position5;
50         this.position5 = this.position4;
51         this.position4 = this.position3;
52         this.position3 = this.position2;
53         this.position2 = this.position1;
54         this.position1 = this.getBall(zustand);
```

5.4.3 Regression möglich?

Das Programm stellt daraufhin fest, ob genügend Punkte vorhanden sind, um die Bewegung des Balles zu ermitteln und ggf. eine ausreichend genaue Regression errechnen zu können.

Dafür wird zunächst ein *ArrayList*-Container (die Klasse muss extra eingebunden werden)

```
1 import java.util.ArrayList;
```

erstellt und mit den ersten drei gespeicherten Positionen - unerheblich, ob vorhanden - gefüttert:

```
56         ArrayList<Spiel.Zustand.Ball> tempListe = new ArrayList<Spiel.Zustand.Ball>();
57         tempListe.add(this.position1);
58         tempListe.add(this.position2);
59         tempListe.add(this.position3);
```

Dieser Container ist notwendig, um die Positionen an die Funktion *AI::inEinerReihe()* übergeben zu können;

```
61         if (inEinerReihe(tempListe)){
```

Die Funktion *AI:inEinerReihe()* überprüft, ob die übergebenen Punkte

1. Überhaupt vorhanden sind ($\neq$ null), [Z.332 ff.] und
2. x-y-linear (in einer Linie) sind [Z.336 - 368]

Andernfalls kann logischer Weise keine lineare Regression errechnet werden.

5.4.4 Regression errechnen

Sind mindestens drei Punkte vorhanden und x-y-linear, kann eine Regression errechnet werden. Es wird überprüft, wieviele Punkte für eine lineare Regression verwendet werden können; dafür wird die Liste um diejenigen Punkte erweitert und dann per *AI::inEinerReihe()* überprüft.

```
70          tempListe.add(this.position4);
71          tempListe.add(this.position5);
72
73          if (this.inEinerReihe(tempListe)){ /* 5 Punkte? */
74              tempListe.add(this.position6);
75              tempListe.add(this.position7);
76
77              if (this.inEinerReihe(tempListe)){ /* 7 Punkte? */
78                  tempListe.add(this.position8);
79                  tempListe.add(this.position9);
80                  tempListe.add(this.position10);
81
82                  if (this.inEinerReihe(tempListe)){ /* 10 Punkte? */
83                      /* Regression aus 10 Punkten */
```

Entsprechend der Anzahl der Punkte werden die einzelnen Summen $t_1 \dots t_6$ Errechnet. Die aktuelle Geschwindigkeit des Balles lässt sich durch

$$\frac{|\Delta x|}{\text{AnzahlDerPunkte}}$$

ermitteln. Hier z.B. die Summen der Regression aus 5 Punkten:

```
139                 /* Regression aus 5 Punkten */
140                 double x1 = this.position1.xKoordinate();
141                 double y1 = this.position1.yKoordinate();
142                 double x2 = this.position2.xKoordinate();
143                 double y2 = this.position2.yKoordinate();
144                 double x3 = this.position3.xKoordinate();
145                 double y3 = this.position3.yKoordinate();
146                 double x4 = this.position4.xKoordinate();
147                 double y4 = this.position4.yKoordinate();
148                 double x5 = this.position5.xKoordinate();
149                 double y5 = this.position5.yKoordinate();
150
151                 geschwindigkeit = (double) Math.abs(x1 - x5)/5;
152
153                 t1 = x1 + x2 + x3 + x4 + x5;
```

```java
				t2 = y1 + y2 + y3 + y4 + y5;
				t3 = x1 * y1 + x2 * y2 + x3 * y3 + x4 * y4 + x5 * y5;
				t5 = x1 * x1 + x2 * x2 + x3 * x3 + x4 * x4 + x5 * x5;
				t6 = 5;
```

$$t_1 \widehat{=} \sum_{i=1}^{n} x_i, t_2 \widehat{=} \sum_{i=1}^{n} y_i, t_3 \widehat{=} \sum_{i=1}^{n} x_i y_i, t_5 \widehat{=} \sum_{i=1}^{n} x_i^2, t_6 \widehat{=} n$$

Aus diesen Summen wird dann die Regression errechnet, vgl. **3.1** Formeln (1) und (2).

```java
		double b = (double) (((t6 * t3) - (t1 * t2))/((t6 * t5) - (t1 * t1)));
		double a = (double) (((t5 * t2) - (t1 * t3))/((t6 * t5) - (t1 * t1)));



		//Der Ball bewegt sich auf der Regressionsgerade a + b * x mit der Geschwindigkeit 'geschwindigkeit' x pro Zug
```

5.4.5 Angriffslogik

Wenn sich der Ball zu mir bewegt,

```java
		if (this.position1.xKoordinate() < this.position3.xKoordinate()){
```

wird zuerst der Punkt errechnet, an dem der Ball vermutlich bei mir aufkommen wird:

```java
190			//Auftreffpunkt und -Winkel berechnen
191			double x = a;
192			int anzahl = 0;
193			while(x < 0 || x > 60){
194				if (x < 0){ /* obere Bande */
195					x = -x;
196				}else if (x > 60){ /* untere Bande */
197					x = 60 - ( x - 60);
198				}
199				anzahl++;
200			}
201			if (anzahl % 2 != 0){ b = -b; }
202
203			//b: Winkel, mit dem der Ball bei mir ankommt
204			//x: Punkt, an dem er ankommt
```

Der Punkt a ist ja bereits aus der Regression bekannt; er kann aber noch > 60 oder < 1 sein. Deshalb wird mit einer *while*-Schleife das Abprallen an der Wand simuliert [Z.193-200]; mit *anzahl* wird gezählt, wie oft er abprallt [Z.199]. Wenn der Ball in einer geraden Anzahl abprallt, heben sich die Veränderungen des Winkels auf; wenn er ungerade oft abprallt, wird der Winkel „umgekehrt“ [Z.201].

Hierbei ist zu beachten, dass es sich bei b nicht um einen Winkel im Sinne des Gradmaßes, sondern um einen Steigungsfaktor handelt. Der Winkel 70° entspricht einem Steigungsfaktor von $\tan(70) \approx 2.747477419$.

Der Ball wird also bei Punkt x mit dem Winkel b ankommen. Wir überprüfen nun, ob ein Todesstoß möglich ist, d.h. ob ein Drittel meines Schlägers den Ball so schlagen

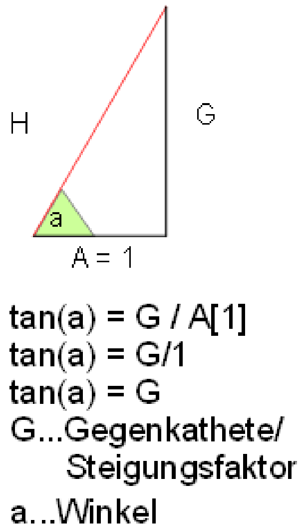


Abbildung 1: Errechnung des Steigungsfaktors G aus Gradmaß-Winkel a

kann, dass er für den Gegner nicht mehr erreichbar ist. Sollte dies nicht der Fall sein, wird zusätzlich noch geprüft, ob der Ball nicht „zumindest“ in eine gegnerische Ecke gespielt werden kann.

```
            double benoetigteZuege = (double) 65/geschwindigkeit;

            double b1 = 2.747477419 - (Math.abs(2.747477419 - (-b))/2); // oberes Drittel
            double b2 = (-2.747477419) + (Math.abs((-2.747477419) - (-b))/2); // unteres Drittel
            double b3 = -b; //mittleres Drittel

            double auftreffpunkt1 = a + b1 * 65;
            double auftreffpunkt2 = a + b2 * 65;
            double auftreffpunkt3 = a + b3 * 65;

            if (benoetigteZuege < (Math.abs(this.auftreffpunkt(auftreffpunkt1) - (this.getEnemy(zustand, id).yKoordinate() + 2.5))) ){
                //Todesstoß mit dem oberen Drittel
                this.ptmt += this.verschiebung; //Das obere Drittel zum PTMT verschieben
                //this.ausgabe("Todesstoß mit dem oberen Drittel.", zug);
            }else if(benoetigteZuege < (Math.abs(this.auftreffpunkt(auftreffpunkt2) - (this.getEnemy(zustand, id).yKoordinate() + 2.5)))){
                //Todesstoß mit dem unteren Drittel
                this.ptmt -= this.verschiebung; //Das untere Drittel zum PTMT verschieben
                //this.ausgabe("Todesstoß mit dem unteren Drittel.", zug);
```

```
            }else if(benoetigteZuege < (Math.abs(this.auftreffpunkt(auftreffpunkt3
                ) - (this.getEnemy(zustand, id).yKoordinate() + 2.5)))){
                //Todesstoß mit der Mitte
                //Keine Aktion nötig, da ptmt = x
            }else{
                //Ist es möglich, in die Ecken zu spielen?
                if (this.auftreffpunkt(auftreffpunkt1) <= 8 || this.auftreffpunkt(
                    auftreffpunkt1) >= 52 ){
                    //oberes Drittel benutzen
                    this.ptmt += this.verschiebung;
                }else if(this.auftreffpunkt(auftreffpunkt2) <= 8 || this.
                    auftreffpunkt(auftreffpunkt2) >= 52 ){
                    //unteres Drittel benutzen
                    this.ptmt -= this.verschiebung;
                }else if(this.auftreffpunkt(auftreffpunkt3) <= 8 || this.
                    auftreffpunkt(auftreffpunkt3) >= 52 ){
                    //Keine Aktion nötig
                }else{
                    //Das zum Ball gerichtete äußere Drittel benutzen
                    this.ptmt += (b > 0) ? this.verschiebung : -this.verschiebung;
                }
             }
```

Für den zufälligen Ausfallwinkel des oberen Drittels gilt immer

$$\tan(70°) \ldots (-b)$$

und für den des unteren

$$(-b) \ldots \tan(-70°)$$

Für die Berechnung nehme ich immer den Mittelwert der Differenz

$$\text{Winkel oben} = \tan(70°) - \frac{|\tan(70°) - (-b)|}{2}$$

```
            double b1 = 2.747477419 - (Math.abs(2.747477419 - (-b))/2); // oberes
                Drittel
```

$$\text{Winkel unten} = \tan(-70°) + \frac{|(-b) - \tan(-70°)|}{2}$$

```
            double b2 = (-2.747477419) + (Math.abs((-2.747477419) - (-b))/2); //
                unteres Drittel
```

$$\text{Winkel mitte} = (-b)$$

```
            double b3 = -b; //mittleres Drittel
```

Aus diesen Winkeln errechne ich dann den möglichen Auftreffpunkt

```java
214                 double auftreffpunkt1 = a + b1 * 65;
215                 double auftreffpunkt2 = a + b2 * 65;
216                 double auftreffpunkt3 = a + b3 * 65;
```

Wenn dann einer der Auftreffpunkte mehr als *benoetigteZuege* vom Gegnerischen Schläger entfernt ist, wird der $PTMT$ so verschoben, dass das entsprechende Drittel auf x und damit auf dem Auftreffpunkt bei mir liegt. Das Ausmaß der Verschiebung lege ich global in der Membervariable *this.verschiebung* fest. Zur y-Koordinate des gegnerischen Schlägers muss noch 2.5 addiert werden, da *this.getEnemy(zustand,id).yKoordinate()* den oberen der sechs Punkte zurückgibt. [vgl. Z.218 - 229]

Wenn kein Winkel diese Bedingungen erfüllt, wird überprüft, ob es möglich ist, in eine gegnerische Ecke zu spielen, unberücksichtigt, ob der Gegner diesen Punkt erreichen kann, oder nicht. [vgl. Z.230 - 238].

Sollte dies alles nicht möglich sein, schlägt meine KI mit demjenigen Drittel, das zum Ball gerichtet ist, da so die meist möglichen Winkel entstehen und der Ball so schwieriger für den Gegner zu berechnen ist. [vgl. Z.240f.]

5.4.6 Defensivlogik

Wenn sich der Ball von mir wegbewegt, wird mit Hilfe der ausgeführten Regression sowohl der Auftreffpunkt beim Gegner als auch der -winkel berechnet.

```java
245                 //Auftreffpunkt und -winkel beim Gegner berechnen
246                 double gegnerPosition = this.getEnemy(zustand, id).yKoordinate();
247                 double x = a + b * 65;
248                 int anzahl = 0;
249                 while(x < 0 || x > 60){
250                     if (x < 0){
251                         x = -x;
252                     }else if (x > 60){
253                         x = 60 - ( x - 60);
254                     }
255                     anzahl++;
256                 }
257                 if (anzahl % 2 != 0){ b = -b; }
258 
259                 double auftreffpunktBeimGegner = x;
260                 double auftreffpunktBeimGegnerGerundet = Math.round(x);
```

Diese Methodik ist bereits aus der Angriffslogik bekannt.

Anschließend wird überprüft, ob und welches gegnerische Schlägerdrittel sich am berechneten Auftreffpunkt befindet. Entsprechend den Drittel wird ein Austrittswinkel ähnlich der Angriffslogik aus dem Mittelwert der möglichen Winkel errechnet und dann derejenige Punkt errechnet, an dem der Ball von dort aus auf meiner Wand aufkommen wird. Die KI bewegt sich dann dorthin bzw. in die Richtung.

```java
262                 /* Ist der gegnerische Schläger am berechneten Punkt? */
263                 if (auftreffpunktBeimGegnerGerundet == gegnerPosition ||
                        auftreffpunktBeimGegnerGerundet == gegnerPosition + 1){
264                     //Oberes Drittel
```

```
265                 double ausfallswinkel = 2.747477419 - (Math.abs(2.747477419 - (-b))/2);
266                 this.ptmt = this.auftreffpunkt(auftreffpunktBeimGegner + ausfallswinkel * 65);
267             }else if (auftreffpunktBeimGegnerGerundet == gegnerPosition + 2 || auftreffpunktBeimGegnerGerundet == gegnerPosition + 3){
268                 //mittleres Drittel
269                 double ausfallswinkel = -b;
270                 this.ptmt = this.auftreffpunkt(auftreffpunktBeimGegner + ausfallswinkel * 65);
271             }else if (auftreffpunktBeimGegnerGerundet == gegnerPosition + 4 || auftreffpunktBeimGegnerGerundet == gegnerPosition + 5){
272                 //unteres Drittel
273                 double ausfallswinkel = (-2.747477419) + (Math.abs((-2.747477419) - (-b))/2);
274                 this.ptmt = this.auftreffpunkt(auftreffpunktBeimGegner + ausfallswinkel * 65);
275             }else{
276                 this.ptmt = 30.0; //Anscheinend ist der gegnerische Schläger nicht am Auftreffpunkt
277             }
278             this.ptmt = 30.0 + ((this.ptmt - 30.0)/2);
```

Damit sich mein Schläger nicht zu sehr in eine Ecke bewegt, die womöglich falsch ist, bewegt er sich nur halb so weit von der Mitte weg, wie er sich laut Berechnung bewegen sollte vgl Z.278.

5.4.7 Bewegung

Die Angriffs- und Verteidigungslogik setzen einen Punkt *PTMT* fest. Die KI bewegt sich dann dorthin

```
        this.doMove(this.ptmt, this.getMe(zustand, id), zug);
        return;
```

Der PTMT steht als Membervariable auch dem nachfolgenden Zug zur Verfügung, wenn der Ball an einer Bande abprallen sollte und somit keine lineare Bewegung mehr vollzieht, siehe folgendes:

5.5 Zu wenige Punkte für eine Regression

Wenn zu wenige Punkte für eine Regression vorhanden sind($\rightarrow$ weniger als 3 oder nicht linear) , bewegt die KI den Schläger entweder in Richtung des gespeicherten PTMT oder - falls dieser noch nicht vorhanden ist - in Richtung der y-Koordinate des Balles. So bewegt sich der Schläger bei einem direkt bewegenden Ball schon einmal in die richtige Richtung; bewegt sich der Ball nicht direkt auf mich zu, hat die KI durch die Abprallungen an den Banden eh genug Zeit, den Schläger zu korrigieren.

```
    }else{
        if (this.ptmt != 0){
            //PTMT befolgen
```

```
            this.doMove(this.ptmt, this.getMe(zustand, id), zug);
        }else{
            //Auch kein PTMT vorhanden
            //-> zum Ball bewegen
            this.doMove(this.getBall(zustand).yKoordinate(), this.getMe(zustand,
                id), zug);
            return;
        }
    }
```

6 Effizienz

Meine KI ist sicher nicht die effizienteste; hierzu wird sicherlich die Regression aus 10 Punkten und die Vorausberechnung des gegnerischen Zuges beitragen. In einer Runde verbraucht meine KI zwischen ca 10000 und 13500 von 30000 Rechenpunkten, das sind $\approx$ 30-45%.(!)

7 Beispiele

Beispiele kann ich hier schlecht in Bildform geben; schauen Sie sich doch ein Paar meiner Challenges im Turniersystem an!