Übungsaufgaben
Künstliche Intelligenz
Serie 3

Mario Krell Berit Grußien Volker Grabsch

5. Februar 2007

http://www.profv.de/uni/

Inhaltsverzeichnis

1 Literarisches Programmieren

Um die Qualität der Programm-Dokumentation zu verbessern, verfolgen wir in dieser Lösung den Ansatz des literarischen Programmierens.[1]

Üblicherweise wird die Dokumentation in den Programm-Code eingebunden, in Form von Kommentaren, Docstrings und ähnlichem. Zusätzlich gibt es eine zusammenhängende Erklärung, also eine separate Dokumentation. Dort wird all das erklärt, was sich in den Kommentarzeilen nur schwer unterbringen lässt.

Beim literarischen Programmieren hingegen wird der Programm-Code in die Dokumentation eingebettet. So dominiert die Struktur der Erklärungen und Kommentare über die Struktur des Programms. Quelltext und Dokumentation werden nicht mehr separiert. Das Programmieren verschmilzt mit dem Erklären.

Damit diese Dokumentation ein in sich geschlossenes Werk ist, gehen wir noch einen Schritt weiter:

- Wir bereiten die Ausgabe des Programmes grafisch auf und lassen sie wieder in dieses Dokument einfließen.
- Wir erklären alle dafür notwendigen Schritte.
- Wir erklären außerdem sämliche Hilfs-Scripte, die uns das literarische Programmieren ermöglichen.
- Auch diese werden literarisch programmiert. Ja, das ist ein Henne-Ei-Problem.[2]

Idealerweise würde ein Interpreter das Dokument direkt einlesen und den eingebetteten Code ausführen. Solche Interpreter gibt es aber noch nicht. Also brauchen wir einen Extraktor, der das Programm aus unserem Dokument holt. Dann starten wir es und verarbeiten die Ausgabe bis zum fertigen PDF-Dokument.

Doch ein Extraktor ist Stuss, wenn er sich selbst extrahieren muss. Deshalb ist er so gestaltet, dass er auch mit einem einfachen Suchbefehl aus dem Dokument geholt werden kann. Jede Codezeile mit einer Markierung versehen, so lösen wir das Henne-Ei-Problem.

Wenn wir erst einmal den Extraktor haben, wozu noch die Schritte bis zum fertigen PDF-Dokument per Hand ausführen? Wir extrahieren sie einfach. Das erledigt ein kleines Shell-Script. Es holt den Extraktor mit dem Suchbefehl `grep`, benutzt ihn zum Extrahieren der Bau-Anweisungen und führt sie aus:

```
#!/bin/sh
set -e

# extract extractor and build instructions
grep '# extract.py$' <aufgabe3.tex >extract.py
python extract.py build-instructions.sh

# execute build instructions
sh -e build-instructions.sh
```

Durch die Option `-e` bricht es im Problemfall sofort ab, wie bei einem Makefile.

So reduziert sich der gesamte Aufwand, vom Holen des Extraktors über das Starten des Programms bis zum fertigen PDF-Dokument, auf:

```
sh build.sh
```

[1] auch: „literate programming“, siehe http://de.wikipedia.org/wiki/Literate_programming

[2] Es ähnelt einem Film, der seine eigenen Dreharbeiten dokumentieren soll.

2 Bello, lauf!

2.1 Starten des Programms

Entsprechend der Aufgabenstellung liegt das Programm als extra Datei vor. Es wurde folgendermaßen aus diesem Dokument extrahiert:

```
python extract.py aufgabe3.py
```

Das Programm ist in Python geschrieben und folgt der Unix-Konvention, von der Standardeingabe zu lesen und in die Standardausgabe zu schreiben. Es kann auf dem Rechner `tegel.informatik.hu-berlin.de` gestartet werden via:

```
python aufgabe3.py <percepts.txt >position.txt 2>probleme.txt
```

Die `build.sh` funktioniert dort ebenfalls.

Erste Zeilen der erzeugten `position.txt`:

```
140;2130.95;-1574.77;;
144;2155.36;-1346.11;;
163;533.49;-1354.42;;
164;533.49;-1208.96;;
165;342.49;-1376.58;;
```

2.2 Grundlegende Annahmen

Wir entwickeln eine Spezialanwendung für unseren Robocup-Hund Bello.

Die Sichtdaten sind fehlerbehaftet. Da die Fehler nicht explizit bekannt sind, muss man sie abschätzen und experimentell, d.h. am Beispiel, erproben.

Bello erwartet nur einen Punkt als Ergebnis. Wenn er dazu eine gewisse Fehlerangabe kriegt, weiß er, ob er sich mehr umschauen müsste oder getrost weiter handeln kann.

Bello ist ein braver Hund und verhält sich ordentlich. Das heißt, wir gehen unter anderem davon aus, dass mehrfache Positionsangaben zu einem Zeitpunkt nicht interessant für Bello sind. Da die gegebene `percepts.txt` sehr viele Informationen enthält und wir 6 wohlunterscheidbare Landmarken auf dem Feld haben, sowie Linienpunkte, gehen wir davon aus, dass nach einer gewissen Orientierungsphase am Anfang keine Notwendigkeit besteht, Alternativen zu betrachten.

Die Blickwinkel-Berechnung ist fehlerbehaftet und könnte über Constraints berechnet werden, was jedoch zu aufwändig für Bello ist. Stattdessen berechnen wir den Winkel aus den zuletzt gegeben Daten. Wir gehen davon aus, dass diese ausreichen, um einen hinreichend guten Winkel auszugeben.

Wir gehen davon aus, dass die Daten innerhalb eines Zeitabschnittes gleichwertig sind, sich also auf genau ein Bild beziehen.

2.3 Hilfsklasse: Rechteck

```
from math import hypot
```

Wir arbeiten intensiv mit Rechtecken:

```
class Rechteck:

    def __init__(self, x_min, x_max, y_min, y_max):
        self.x_min = x_min
```

```
        self.x_max = x_max
        self.y_min = y_min
        self.y_max = y_max
```

Zwei Rechtecke werden koordinatenweise verglichen (von `loese_constraints` benötigt):

```
    def __cmp__(self, other):
        return cmp((self.x_min, self.x_max, self.y_min, self.x_max),
                   (other.x_min, other.x_max, other.y_min, other.x_max))
```

Zur Positionsausgabe benötigen wir den Mittelpunkt:

```
    def mittelpunkt(self):
        x = (self.x_min + self.x_max) / 2
        y = (self.y_min + self.y_max) / 2
        return x, y
```

... und zur Fehlerabschätzung den Umkreisradius:

```
    def umkreis_radius(self):
        dx = self.x_max - self.x_min
        dy = self.y_max - self.y_min
        return hypot(dx, dy) * 0.5
```

Zum Lösen der Constraints wollen wir Rechtecke miteinander schneiden:

```
    def schneide(self, other):
        return Rechteck(max(self.x_min, other.x_min),
                        min(self.x_max, other.x_max),
                        max(self.y_min, other.y_min),
                        min(self.y_max, other.y_max))
```

Beim Schneiden disjunkter Rechtecke entstehen widersprüchliche Koordinaten. Dies kann über folgende Funktion festgestellt werden:

```
    def widerspruechlich(self):
        return (self.x_min > self.x_max
                or self.y_min > self.y_max)
```

Weiterhin benötigen wir für Vereinigungs-Constraints die Möglichkeit, zwei Rechtecke zu einem einzigen zusammen zu fassen. Da die Vereinigung meistens kein Rechteck mehr ist, berechnen wir hier nicht wirklich die Vereinigung, sondern ihr umschließendes Rechteck. Widersprüchliche Rechtecke behandeln wir als leere Menge, d.h. sie beeinflussen die Vereinigung nicht:

```
    def vereinige(self, other):
        if self.widerspruechlich():
            return other
        elif other.widerspruechlich():
            return self
        else:
            return Rechteck(min(self.x_min, other.x_min),
                            max(self.x_max, other.x_max),
                            min(self.y_min, other.y_min),
                            max(self.y_max, other.y_max))
```

2.4 Bellos Welt

Größe des Spielfelds:

```
spielfeld = Rechteck(-3000, 3000, -2000, 2000)
```

Positionen der Landmarken (Flaggen und Tore):

```
landmarken_koordinaten = {
    'FPY': ( 1350,  1950),
    'FPB': (-1350,  1950),
    'FYP': ( 1350, -1950),
    'FBP': (-1350, -1950),
    'GY':  ( 2700,  0),
    'GB':  (-2700,  0),
}
```

Zur einfachen Berechnung verlängern wir alle Linien. Y-Koordinaten der waagerechten Linien:

```
linienpunkte_y = [-1800, -650, 650, 1800]
```

X-Koordinaten der senkrechten Linien:

```
linienpunkte_x = [-2700, -2050, 0, 2050, 2700]
```

```
from math import radians
```

Relativer Messfehler der Entfernungen und Winkel zu den gesehenen Objekten:

```
mess_fehler = 0.15                  # 15%
winkel_mess_fehler = radians(0.25)  # +/-  1/4 Grad
```

Bello hat eine Höchstgeschwindigkeit von 300mm pro Sekunde. Die Sensordaten gehen über 236 Zeitabschnitte, was ca. 60 Sekunden entspricht. Ein Zeitabschnitt entspricht also ca. einer viertel Sekunde. Somit läuft Bello maximal $\frac{300}{4}$mm pro Zeiteinheit (viertel Sekunde).

```
bewegungs_unschaerfe = 300.0/4
```

Maximaler absoluter Fehler der ermittelten Position in mm, bei dem Bello seinem Orientierungssinn noch traut:

```
max_fehler = 800.0
```

3 Constraints

3.1 Constraint-Löser

Zum Lösen der Constraints durch die Landmarken, verwenden wir den Constraintlöser aus Aufgabe 2.

Dabei nehmen wir unser Startrechteck und mehrere Constraints und schränken unser Rechteck immer wieder hintereinander durch diese ein, bis sich nichts mehr ändert.

```
def loese_constraints(start, constraints):
    aktuell = start
    while True:
        vorher = aktuell
        for constraint in constraints:
            aktuell = constraint.schneide(aktuell)
        if aktuell == vorher:
            return aktuell
```

Dabei kann es leider zu einer Konvergenz kommen, die nicht ganz das minimale Rechteck berechnet. In unserem Fall haben wir aber meist schon eine gute Prognose.

Im Gegensatz zur Aufgabe 2 haben wir jedoch nicht nur Abstands-Constraints, sondern auch Vereinigungen von nicht-zusammenhängenden Constraints. Diese benötigen wir für die Linienpunkte.

Der Algorithmus kommt mit jedem Constraint zurecht, dass sich mit einem Rechteck schneiden kann. Das heißt, unsere Constraint-Klassen müssen lediglich eine `schneide`-Methode anbieten, die ein Rechteck nimmt und ein (kleineres) Rechteck zurückliefert. Den Rest erledigt dann unser Constraint-Löser.

`Rechteck` ist offensichtlich solch eine Constraint-Klasse. Die anderen beiden beschreiben wir in den folgenden Kapiteln.

3.2 Abstands-Constraint

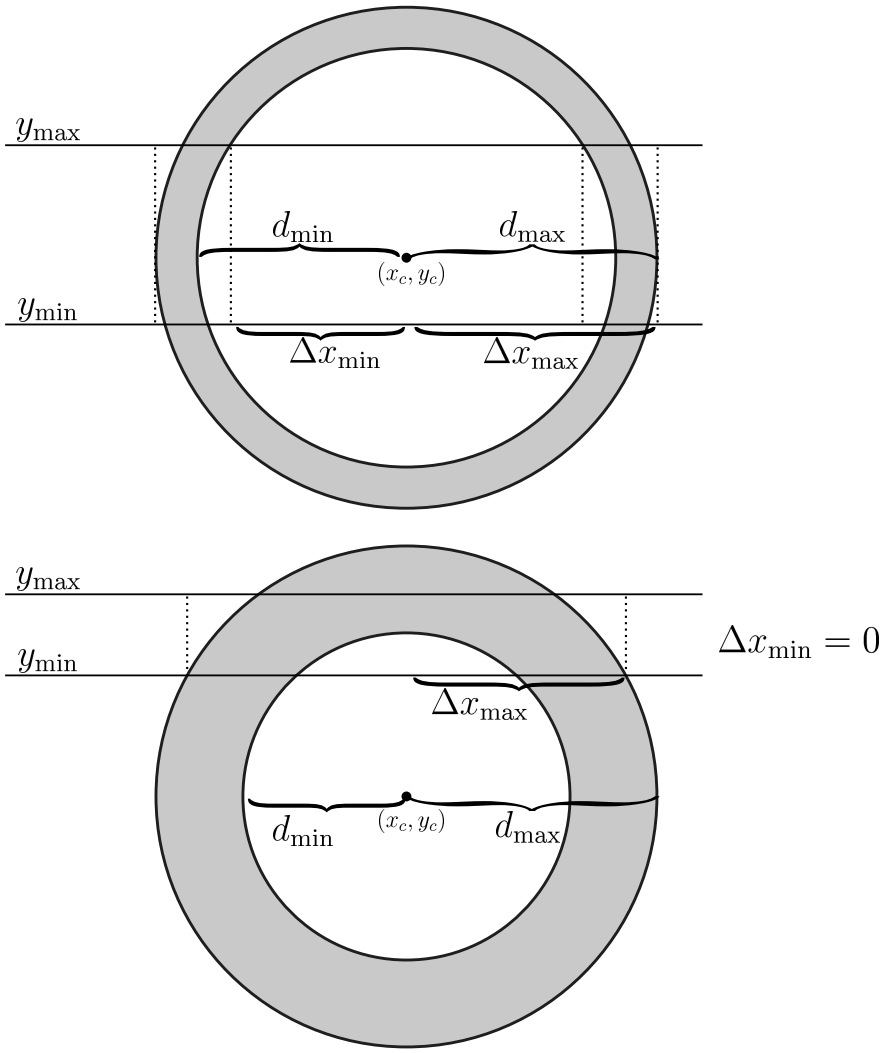
```
from math import sqrt
```

Fehlerbehaftete Abstands-Constraints entsprechen Kreisringen:

```
class AbstandsConstraint:

    def __init__(self, center_x, center_y, dist_min, dist_max):
        assert(dist_min < dist_max)
        self.center_x = center_x
        self.center_y = center_y
        self.dist_min = dist_min
        self.dist_max = dist_max
```

Um diesen mit einem Rechteck schneiden zu können, ermiteln wir $\Delta x_{\min}$ und $\Delta x_{\max}$ durch Schnitt mit einem y-Schlauch. Dabei muss man ein paar Fälle unterscheiden:



Eine leichte Überlegung zeigt:

$$\Delta x_{\min} = \min\left\{\sqrt{d_{\min}^2 - (y_{\min} - y_c)^2}, \sqrt{d_{\min}^2 - (y_{\max} - y_c)^2}\right\}$$
$$\Delta x_{\max} = \max\left\{\sqrt{d_{\max}^2 - (y_{\min} - y_c)^2}, \sqrt{d_{\max}^2 - (y_{\max} - y_c)^2}\right\}$$

- Falls eine der Wurzeln nicht definiert ist, setzen wir das entsprechende $\Delta x_{\min}$ bzw. $\Delta x_{\max}$ auf 0.
- Für $y_{\min} \leq y_c \leq y_{\max}$ setzen wir $\Delta x_{\max} := d_{\max}$.

```
    def delta_x_min(self, dist, center_y, y_min, y_max):
        return sqrt(max(0, min(dist**2 - (y_min - center_y)**2,
                               dist**2 - (y_max - center_y)**2)))

    def delta_x_max(self, dist, center_y, y_min, y_max):
        if y_min <= center_y <= y_max:
            return dist
        return sqrt(max(0, dist**2 - (y_min - center_y)**2,
                           dist**2 - (y_max - center_y)**2))
```

Hier Schneiden wir mit dem y-Schlauch:

```
    def cut_x(self, dist_min, dist_max, center_x, center_y,
                    x_min, x_max, y_min, y_max):
```

```
        dx_min = self.delta_x_min(dist_min, center_y, y_min, y_max)
        dx_max = self.delta_x_max(dist_max, center_y, y_min, y_max)

        if x_min <= center_x - dx_max:
            x_min = center_x - dx_max
        elif center_x - dx_min <= x_min <= center_x + dx_min:
            x_min = center_x + dx_min

        if center_x + dx_max <= x_max:
            x_max = center_x + dx_max
        elif center_x - dx_min <= x_max <= center_x + dx_min:
            x_max = center_x - dx_min

        return x_min, x_max
```

Die Berechnung mit einem x-Schlauch erfolgt analog. Beides zusammen ergibt dann schon unsere `schneide`-Methode:

```
    def schneide(self, rechteck):
        x_min, x_max = self.cut_x(self.dist_min, self.dist_max,
                                  self.center_x, self.center_y,
                                  rechteck.x_min, rechteck.x_max,
                                  rechteck.y_min, rechteck.y_max)
        y_min, y_max = self.cut_x(self.dist_min, self.dist_max,
                                  self.center_y, self.center_x,
                                  rechteck.y_min, rechteck.y_max,
                                  rechteck.x_min, rechteck.x_max)
        return Rechteck(x_min, x_max, y_min, y_max)
```

3.3 Vereinigungs-Constraint

Ein Vereinigungs-Constraint ist eigentlich ein Meta-Constraint. Er besteht aus mehreren Teil-Constraints, z.B. aus Rechtecken und Abstands-Constraints.

Sein Konstruktor nimmt entsprechend eine Liste von Constraints entgegen:

```
class VereinigungsConstraint:

    def __init__(self, constraints):
        self.constraints = constraints
```

Er delegiert das Schneiden an seine Teil-Constraints entsprechend dem Distributivgesetz für Mengen,

$$(C_1 \cup C_2 \cup C_3) \cap R = (C_1 \cap R) \cup (C_2 \cap R) \cup (C_3 \cap R)$$

mit dem Unterschied, dass die Vereinigung zu einem Rechteck abgeschlossen wird.

```
    def schneide(self, rechteck):
        ergebnis = Rechteck(1, 0, 0, 0)     # leere Menge
        assert(ergebnis.widerspruechlich)

        for constraint in self.constraints:
            ergebnis = ergebnis.vereinige(constraint.schneide(rechteck))

        return ergebnis
```

4 Verarbeitung

4.1 Einlesen der Sensordaten

Die Sensordaten werden in der Reihenfolge geliefert, wie sie gesehen werden. Deshalb macht es Sinn, sie im Hauptprogramm sequentiell abzuarbeiten:

```
import sys

def main():
    t = 0
    landmarken   = []
    linienpunkte = []
    wo_ist_bello = spielfeld
    for line in sys.stdin:
        percept = iter(line.split(";"))
        bezeichnung = str  (percept.next())
        t_neu       = int  (percept.next())
        entfernung  = float(percept.next())
        winkel      = float(percept.next())
```

Erreichen wir einen neuen Zeitpunkt, so werden die Daten des alten verarbeitet:

```
        while t < t_neu:
            wo_ist_bello = orientiere(landmarken, linienpunkte,
                                      t, wo_ist_bello)
            t += 1
            landmarken   = []
            linienpunkte = []
```

Da Landmarken und Linienpunkte unterschiedlich behandelt werden, trennen wir sie beim sequentiellen Lesen der Daten. Zudem muss bei Landmarken ja noch die Normalenrichtung gespeichert werden:

```
        if bezeichnung == "LNP":
            normale = float(percept.next())
            linienpunkte += [(entfernung, winkel, normale)]
        else:
            landmarken += [(entfernung, winkel, bezeichnung)]
```

Am Ende verarbeiten wir noch die Informationen des letzten Zeitpunktes:

```
    orientiere(landmarken, linienpunkte, t, wo_ist_bello)
```

4.2 Verarbeiten der Sensordaten eines Zeitpunktes

```
def orientiere(landmarken, linienpunkte, t, wo_ist_bello):
```

Hier beschreibt `wo_ist_bello` das Positionsrechteck des letzten Zeitpunktes.

Zuerst erweitern wir unser Positionsrechteck um die potentielle Fortbewegung von Bello:

```
    wo_ist_bello = bewegungs_abschaetzung(wo_ist_bello)
```

Dann erzeugen wir aus den Landmarken und Linienpunkten unsere Constraints:

```
    constraints = []
    constraints += landmarken_constraints(landmarken)
    constraints += linienpunkte_constraints(linienpunkte)
```

Nun wenden wir unseren Constraint-Löser an.

Auch wenn wir es nicht hoffen, können Sensordaten auch grobe Fehler produzieren. Sei es durch Kommafehler oder Fehlinterpretation von Objekten durch den Image-Prozessor. Um trotzdem ordentlich weiterarbeiten zu können, wird unser Rechteck auf das ursprüngliche zurückgesetzt, falls unser Schnittmenge einmal leer wird.

```
    alt = wo_ist_bello
    wo_ist_bello = loese_constraints(wo_ist_bello, constraints)
    if wo_ist_bello.widerspruechlich():
        print >>sys.stderr, "Widerspruechliche Daten zu t =", t
        wo_ist_bello = alt
```

Es tauchten zu folgenden Zeitpunkten Probleme auf:

```
Widerspruechliche Daten zu t = 148
Widerspruechliche Daten zu t = 152
Widerspruechliche Daten zu t = 229
```

Falls möglich, berechnen wir mithilfe seiner Position und einer Landmarke seinen aktuellen Blickwinkel:

```
    winkel = blickwinkel(wo_ist_bello, landmarken)
```

Sind die Daten verarbeitet, erfolgt eine Ausgabe und die Verarbeitung des nächsten Zeitpunktes.

```
    positions_ausgabe(wo_ist_bello, winkel, t)
    return wo_ist_bello
```

4.3 Bewegungs-Abschätzung

Wir erweitern nach jedem Zeitabschnitt unser Positionsrechteck um die potentielle Fortbewegung von Bello. Hier könnte man später auch noch die vermutete Bewegungsrichtung einbinden und vielleicht sogar die Geschwindigkeit, um diese Vergrößerung abzuschwächen.

```
def bewegungs_abschaetzung(alt):
    neu = Rechteck(alt.x_min - bewegungs_unschaerfe,
                   alt.x_max + bewegungs_unschaerfe,
                   alt.y_min - bewegungs_unschaerfe,
                   alt.y_max + bewegungs_unschaerfe)
```

Bello würde niemals unerlaubt das Spielfeld verlassen. Deshalb ist das neue Rechteck immer mit dem Spielfeld-Rechteck zu schneiden.

```
    neu = neu.schneide(spielfeld)
    return neu
```

4.4 Landmarken

Die Landmarken liefern ein Kreisringconstraint, dass wir mit unserem aktuellen Positionsrechteck schneiden. Dies Kreisringconstraints entstehen dadurch, dass die Entfernungsangaben fehlerbehaftet sind.

```
def landmarken_constraints(landmarken):
    constraints = []
```

```
    for entfernung, winkel, bezeichnung in landmarken:
        entfernung_min = (1 - mess_fehler) * entfernung
        entfernung_max = (1 + mess_fehler) * entfernung
        x, y = landmarken_koordinaten[bezeichnung]
        constraints += [AbstandsConstraint(
                            x, y,
                            entfernung_min, entfernung_max)]
    return constraints
```

4.5 Linienpunkte

```
from math import cos
from math import pi
```

```
def linienpunkte_constraints(linienpunkte):
    constraints = []
```

Linienpunkte erfordern eine gesonderte Behandlung. Wir nehmen die Daten zu einem Linienpunkt,

```
    for entfernung, winkel, normale in linienpunkte:
        teilconstraints = []
```

berechnen daraus den Abstand von Bello zu der zugehörigen Linie, und beachten den Messfehler.

```
        nwinkel_min = normale - winkel_mess_fehler
        nwinkel_max = normale + winkel_mess_fehler
        cos1 = abs(cos(nwinkel_min))
        cos2 = abs(cos(nwinkel_max))

        if ((nwinkel_min < -pi < nwinkel_max) or
            (nwinkel_min <  pi < nwinkel_max)):
            cos_min = 0
        else:
            cos_min = min(cos1, cos2)

        if nwinkel_min < 0 < nwinkel_max:
            cos_max = 1
        else:
            cos_max = max(cos1, cos2)

        abstand_min = (1 - mess_fehler) * entfernung * cos_min
        abstand_max = (1 + mess_fehler) * entfernung * cos_max
```

Da wir nicht wissen, um welche Linie es sich handelt, stellen wir $9 \cdot 2$ Rechteck-Constraints für die geraden Linien und $2 \cdot 2$ Abstands-Constraints für die Kreislinie in der Spielfeldmitte auf. Damit erhalten wir das neue Positionsrechteck.

Das führt zu folgenden Rechteck-Constraints für die waagerechten Linien:

```
        for y in linienpunkte_y:
            teilconstraints += [
                Rechteck(spielfeld.x_min, spielfeld.x_max,
                         y + abstand_min, y + abstand_max),
                Rechteck(spielfeld.x_min, spielfeld.x_max,
                         y - abstand_max, y - abstand_min)]
```

Rechteck-Constraints für die senkrechten Linien:

```
        for x in linienpunkte_x:
            teilconstraints += [
                Rechteck(x + abstand_min, x + abstand_max,
                         spielfeld.y_min, spielfeld.y_max),
                Rechteck(x - abstand_max, x - abstand_min,
                         spielfeld.y_min, spielfeld.y_max)]
```

Abstands-Constraints für die Kreislinien:

```
        teilconstraints += []  # nicht implementiert :-(
```

Diese Constraints vereinigen wir:

```
        constraints += [VereinigungsConstraint(teilconstraints)]
```

```
    return constraints
```

4.6 Blickwinkel

Wir bestimmen den Blickwinkel nur, wenn Bello mindestens eine Landmarke (Flagge oder Tor) sieht. War dies nicht möglich, liefern wir `None` statt des Winkels zurück:

```
def blickwinkel(wo_ist_bello, landmarken):
    if not landmarken:
        return None
    landmarke = landmarken[0]
```

Zunächst muss der Abstandsvektor $d = (d_1, d_2)$ von Bellos Position zu der Position der Landmarke bzw. des Tors berechnet werden. Seien x, y die Koordinaten der Landmarke bzw. des Tors und x_B, y_B die Koordinaten von Bello.

$$d_1 = x - x_B$$

$$d_2 = y - y_B$$

Der Winkel Φ zwischen dem Abstandsvektor d und einem Vektor in Nullrichtung $d_0 = (1, 0)$ ergibt sich durch

$$\Phi = cos^{-1}(\frac{d * d_0}{|d|}) = cos^{-1}(\frac{d_1}{\sqrt{d_1^2 + d_2^2}})$$

Dieser Winkel hat noch nicht die geforderte Richtung. Um diese zu bestimmen muss zwischen Landmarken und Toren unterschieden werden.

Bei Landmarken gilt folgendes: Wenn Bello eine untere Landmarke sieht (d.h mit negativer y- Koordinate) $y < 0$ ist

$$\Phi = 2 * \pi - \Phi$$

Sieht Bello ein Tor: Dann gilt $\Phi = 2 * \pi - \Phi$, wenn $y_B > 0$ Bello in der oberen Spielfeldhälfte steht.

Da Bello, wenn er ein Tor oder eine Landmarke sieht, nicht direkt darauf ausgerichtet ist, sondern in einem Winkel ϕ zu der Landmarke bzw. dem Tor ausgerichtet ist, ergibt sich nur der eigentliche Blickwinkel α aus

$$\alpha = \Phi - \phi$$

Zuletzt muss $\alpha \in [-\pi, 3\pi]$ noch in den Wertebereich von $[-\pi, \pi]$ gebracht werden.

```
def blickwinkel(wo_ist_bello, landmarken):
    return None  # nicht implementiert :-(
```

5 Ausgabe

5.1 Positionsausgabe

Die Ausgabe der vermutlichen Position erfolgt als Mittelpunkt des Rechtecks mit einem gewissen Fehler am Ende jedes Zeitabschnitts. Da uns dies aber durch das geforderte Format nicht möglich ist, liefern wir einfach nur Positionspunkte, wenn Bello sich seiner Position relativ sicher ist, d.h. der Fehler bzw. die Rechteckdiagonale klein genug. Der Blickwinkel wird auch ausgegeben, falls er berechnet werden konnte.

```
def positions_ausgabe(wo_ist_bello, blickwinkel, t):
    x, y = wo_ist_bello.mittelpunkt()
    fehler = wo_ist_bello.umkreis_radius()
    if fehler < max_fehler:
        if blickwinkel is None:
            print "%i;%.2f;%.2f;;" % (t, x, y)
        else:
            print "%i;%.2f;%.2f;%.4f;" % (t, x, y, blickwinkel)
```

Damit ist das Programm fertig!

```
if __name__ == "__main__":
    main()
```

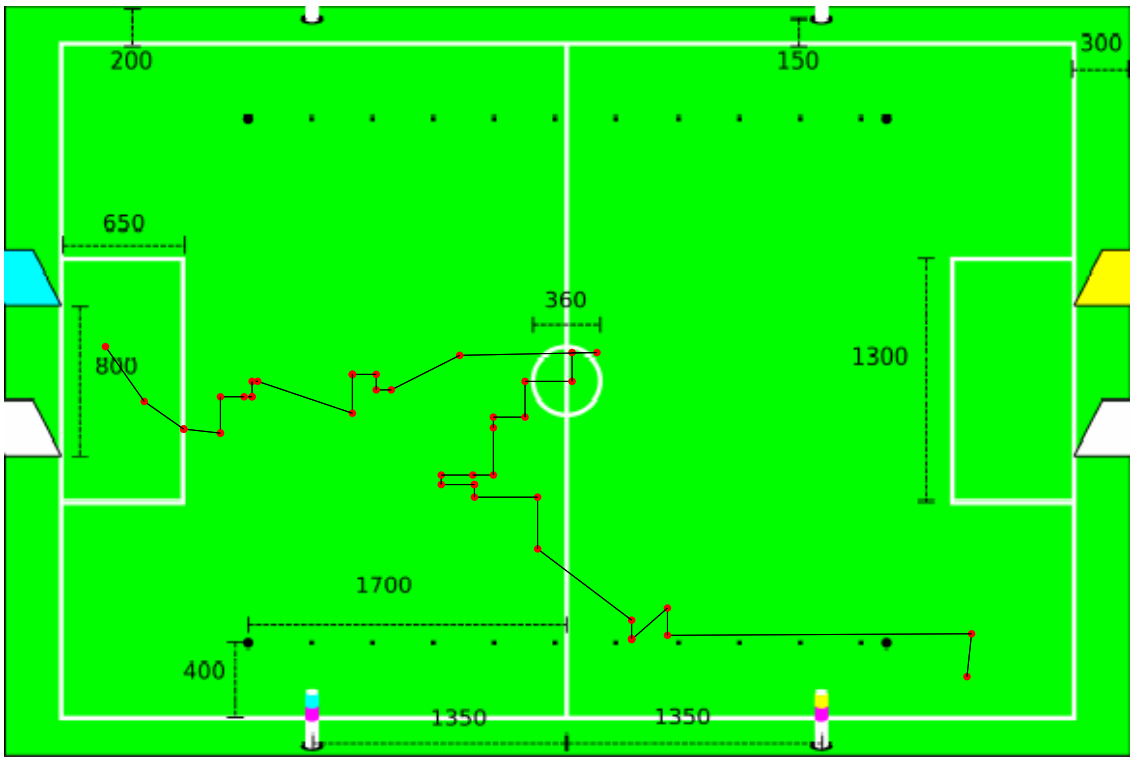
5.2 Grafische Ausgabe

Nun werden wir die erzeugte `position.txt` visualisieren, damit Bello mit ihr auch etwas anfangen kann. Dies erledigt ein extra Script, das wie üblich aus dem Dokument extrahiert werden kann:

```
python extract.py pos2svg.py
```

Es erzeugt aus den Positions-Daten eine SVG-Grafik. Aufruf:

```
python pos2svg.py <position.txt >position.svg
```



Wie arbeitet das Programm? Nun, zuerst wird der übliche SVG-Header ausgegeben:

```
print '<?xml version="1.0"?>'
print '<!DOCTYPE svg'
print '          PUBLIC "-//W3C//DTD SVG 1.1//EN"'
print '          "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">'
print
print '<svg version="1.1"'
print '     width="600px" height="400px"'
print '     viewBox="-3000 -2000 6000 4000"'
print '     xmlns="http://www.w3.org/2000/svg"'
print '     xmlns:xlink="http://www.w3.org/1999/xlink">'
print
```

Wir laden die Spielfeld-Skizze als Hintergrund-Grafik:

```
print '    <image x="-3000" y="-2000"'
print '           width="6000" height="4000"'
print '           xlink:href="spielfeld.png"/>'
print
```

Wir müssen aufpassen! Das SVG-Koordinatensystem hat gespiegelte Y-Koordinaten, diese ëntspiegeln"wir zunächst:

```
print '    <g transform="scale(1 -1)">'
```

Dann holen wir von der Standard-Eingabe die Positions-Informationen:

```
import sys
altes_x = None
altes_y = None
for line in sys.stdin:
    t, x, y, winkel, br = line.split(";")
```

Die Positionen werden der Reihe nach durch schwarze Linien miteinander verbunden:

```
    if altes_x is not None:
        print '        <line',
        print 'x1="%s"' % altes_x,
        print 'y1="%s"' % altes_y,
        print 'x2="%s"' % x,
        print 'y2="%s"' % y,
        print 'style="stroke: black; stroke-width: 5"/>'
    altes_x = x
    altes_y = y
```

Jede Position wird durch einen kleinen roten Kreis dargestellt:

```
    print '        <circle',
    print 'cx="%s"' % x,
    print 'cy="%s"' % y,
    print 'r="20"',
    print 'style="fill: red"/>'
```

Fertig!

```
print '    </g>'
print '</svg>'
```

6 Weitere Informationen

6.1 Erzeugen des PDF-Dokuments

LaTeX benötigt alle Grafiken im EPS-Format. Die Rastergrafiken konvertieren wir mit ImageMagick:

```
convert spielfeld.png spielfeld.eps
```

Die Vektorgrafiken konvertieren wir mit Inkscape:

```
inkscape -E skizze1.eps skizze1.svg
inkscape -E skizze2.eps skizze2.svg
inkscape -E position.eps position.svg
```

Wenn alle Grafiken nach EPS konvertiert sind, starten wir LaTeX. Das machen wir mehrmals, damit Inhaltsverzeichnis und Referenzen stimmen:

```
latex aufgabe3.tex
latex aufgabe3.tex
```

LaTeX erzeugt eine DVI-Datei, die wir in eine PostScript-Datei ausdrucken:

```
dvips -j0 -o aufgabe3.ps aufgabe3.dvi
```

Zum Schluss verwandeln wir PostScript in PDF:

```
ps2pdf aufgabe3.ps
```

Unser PDF-Dokument `aufgabe3.pdf` ist nun fertig.

6.2 Erzeugen der Folien

Konvertieren der zusätzlichen Vektorgrafiken nach EPS:

```
inkscape -B -E skizze_a2_loesung.eps skizze_a2_loesung.svg
inkscape -B -E skizze_a2_loesung_2.eps skizze_a2_loesung_2.svg
inkscape -E skizze_hund_winkel.eps skizze_hund_winkel.svg
inkscape -E skizze_linienpunkte_h.eps skizze_linienpunkte_h.svg
inkscape -E skizze_linienpunkte_hv.eps skizze_linienpunkte_hv.svg
inkscape -E skizze_linienpunkte.eps skizze_linienpunkte.svg
inkscape -E skizze_schneiden_kreisring_1.eps skizze_schneiden_kreisring_1.svg
inkscape -E skizze_schneiden_kreisring_2.eps skizze_schneiden_kreisring_2.svg
inkscape -E skizze_schneiden_vereinigung_1.eps skizze_schneiden_vereinigung_1.svg
inkscape -E skizze_schneiden_vereinigung_2.eps skizze_schneiden_vereinigung_2.svg
inkscape -T -B -E skizze_hund_fragen.eps skizze_hund_fragen.svg
inkscape -E skizze_hund_bewegung_1.eps skizze_hund_bewegung_1.svg
inkscape -E skizze_hund_bewegung_2.eps skizze_hund_bewegung_2.svg
inkscape -T -B -E skizze_hund_denken.eps skizze_hund_denken.svg
```

PDF-LaTeX benötigt die Grafiken im PDF-Format:

```
epstopdf skizze1.eps
epstopdf skizze2.eps
epstopdf position.eps
epstopdf skizze_a2_loesung.eps
epstopdf skizze_a2_loesung_2.eps
epstopdf skizze_hund_winkel.eps
epstopdf skizze_linienpunkte_h.eps
```

```
epstopdf skizze_linienpunkte_hv.eps
epstopdf skizze_linienpunkte.eps
epstopdf skizze_schneiden_kreisring_1.eps
epstopdf skizze_schneiden_kreisring_2.eps
epstopdf skizze_schneiden_vereinigung_1.eps
epstopdf skizze_schneiden_vereinigung_2.eps
epstopdf skizze_hund_fragen.eps
epstopdf skizze_hund_bewegung_1.eps
epstopdf skizze_hund_bewegung_2.eps
epstopdf skizze_hund_denken.eps
```

PDF-Dokument bauen:

```
pdflatex aufgabe3_folien_tiefblau.tex
```

6.3 Erzeugen der ZIP-Archive

Zuerst packen wir alle wichtigen Quell-Dateien:

```
zip aufgabe3-src.zip aufgabe3.tex  \
                     aufgabe3_folien_tiefblau.tex  \
                     percepts.txt  \
                     spielfeld.png \
                     skizze*.svg \
                     formel_*.png \
                     Hund_*.png \
                     build.sh
```

In das abzugebene ZIP-Archiv packen wir das PDF-Dokument, das extrahierte Python-Programm und das Quell-Archiv:

```
zip aufgabe3.zip aufgabe3.pdf     \
                 aufgabe3.py      \
                 aufgabe3-src.zip
```

6.4 Extrahieren des Quellcodes

Unser Extraktor ist ein Python-Script. Er wird wie folgt aufgerufen:

```
python extract.py [Dateiname]
```

Im LaTeX-Quellcode des Dokuments befinden sich spezielle Markierungen. Diesen folgt jeweils eine LaTeX-Umgebung, die ein Stück vom Quellcode beinhaltet:

```
% Dateiname
\begin{...}
Quellcode-Stück
\end{...}
```

Der Extraktor erkennt diese Markierungen und holt die Quellcode-Stücke heraus. Dazu öffnet er zunächst den LaTeX-Quellcode und die Ziel-Datei:

```
import sys                                                    # extract.py
src  = file("aufgabe3.tex", "r")                              # extract.py
dest = file(sys.argv[1],    "w")                              # extract.py
```

Die Verarbeitung ist als Zustandsautomat implementiert. Das heißt, wir durchlaufen wir die `src`-Datei zeilenweise und wechseln zwischendurch unseren Zustand, der gegeben ist durch:

`in_code` – ob wir uns gerade im Quellcode-Bereich befinden

`found_marker` – ob wir eine spezielle Markierung vor kurzem gesehen haben

```
in_code      = False                                          # extract.py
found_marker = False                                          # extract.py
for line in src:                                              # extract.py
```

Pro Zeile entsorgen wir zuerst die störenden Zeilentrenner:

```
    line = line.rstrip("\n\r")                                # extract.py
```

Sobald die Zeile eine Umgebung schließt, gehen wir davon aus, dass wir uns nicht mehr im Quellcode-Bereich befinden. Die letzte Markierung verliert ihre Gültigkeit:

```
    if line.startswith("\\end{"):                             # extract.py
        in_code = False                                       # extract.py
        found_marker = False                                  # extract.py
```

Wir müssen uns merken, falls wir über eine spezielle Markierung stolpern:

```
    if line == "% "+dest.name:                                # extract.py
        found_marker = True                                   # extract.py
```

Jenachdem, ob wir uns im Quellcode-Bereich befinden oder nicht, übernehmen wir die Zeile oder kommentieren sie aus:

```
    if in_code:                                               # extract.py
        print >>dest, line                                    # extract.py
    else:                                                     # extract.py
        print >>dest, "# "+line                               # extract.py
```

So stimmen die Zeilennummern zwischen der Quelldatei und der generierten Datei überein. Das ist vorallem bei Fehlermeldungen wichtig.

Variante: Wollen wir Quellcode extrahieren, bei dem Kommentare nicht mit „#“ eingeleitet werden, fügen wir stattdessen einfach leere Zeilen ein. Das ist nicht so schön, sorgt aber ebenfalls für korrekte Zeilennummern:

```
    if in_code:
        print >>dest, line
    else:
        print >>dest
```

Zum Schluss überprüfen wir, ob eine Umgebung nach einer speziellen Markierung geöffnet wurde. Dann befinden wir uns ab der nächsten Zeile im Quellcode-Bereich:

```
    if (line.startswith("\\begin{")                    # extract.py
        and found_marker):                             # extract.py
        in_code = True                                 # extract.py
```

Jede Zeile des Extraktors endet auf „`# extract.py`“. Dadurch kann er via

```
grep '# extract.py$'
```

aus dem Dokument herausgeholt werden.