Übungsaufgaben
Künstliche Intelligenz
Serie 2

Mario Krell Berit Grußien Volker Grabsch

13. Dezember 2006

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

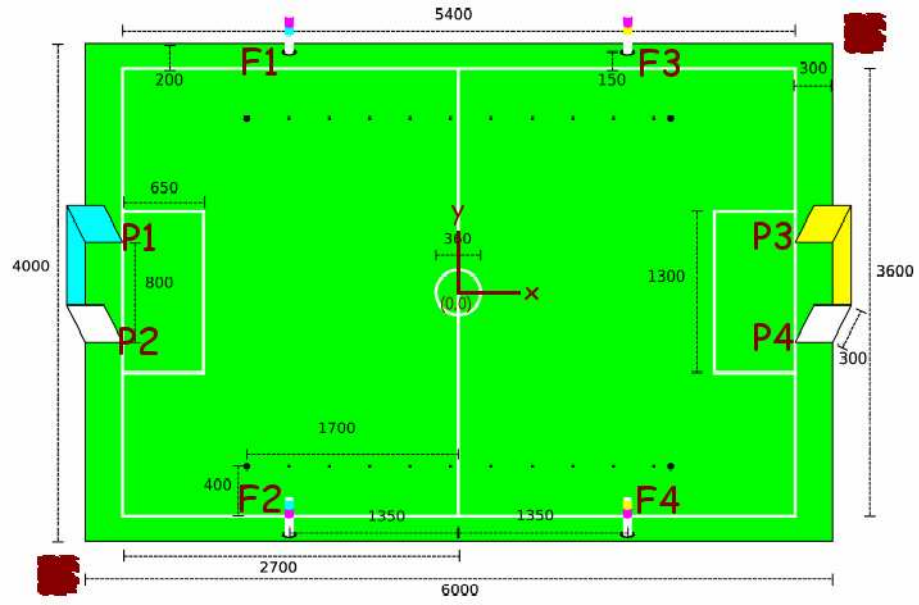
Inhaltsverzeichnis

1 Annahmen über den Imageprozessor

Zunächst gehen wir davon aus, dass der Imageprozessor sämtliche 4 Flaggen und 4 Torpfosten unterscheiden kann. Im folgenden „Objekt“ genannt. Da die 4 Flaggen unterschiedliche Farbkombinationen und die 2 Tore verschiedene Farben haben und von außen weiß sind, sollte das möglich sein. Im folgenden wird

$$i \in \{F1, ..., F4, P1, ..., P4\}$$

für eine bestimmte **F**lagge bzw. einen bestimmten **P**fosten stehen:



Wenn auf dem Kamerabild ein Objekt vollständig zu sehen ist, so nimmt es auf dem Bild in etwa eine Rechteckform an. Wir gehen weiterhin davon aus, dass uns der Imageprozessor im diesem Fall die x-y-Koordinaten in Grad dieser 4 Ecken liefern kann. Diese Koordinaten werden mit

$$K_{i1} = (\alpha_{i1}, \beta_{i1})$$
$$K_{i2} = (\alpha_{i2}, \beta_{i2})$$
$$K_{i3} = (\alpha_{i3}, \beta_{i3})$$
$$K_{i4} = (\alpha_{i4}, \beta_{i4})$$

bezeichnet.
Dabei sind die Ecken wie folgt nummeriert:

1 – Ecke links oben

2 – Ecke links unten

3 – Ecke rechts oben

4 – Ecke rechts unten

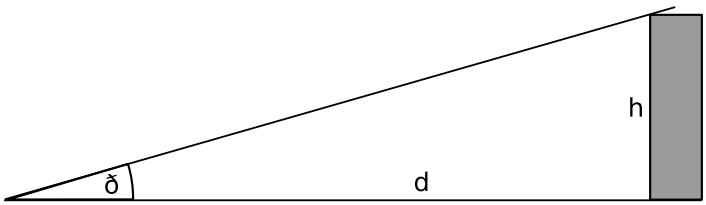
2 Aufgabenteil a)

Sei h_i die reale Höhe in mm des Objektes i und δ_i die gesehene Höhe des Objektes i im Bild:

$$\delta_i = \sqrt{(\alpha_{i1} - \alpha_{i2})^2 + (\beta_{i1} - \beta_{i2})^2}$$

Dies gilt unabhängig von dem Neigungswinkel der Kamera.

Sei weiterhin d_i der Abstand des Roboterhundes Bello zum Objekt i:



Dann gilt $\cot \delta_i = \frac{d_i}{h_i}$. Dies ist eine Näherung, die aber gerechtfertigt ist, da wir die Objekte nur betrachten, wenn wir sie ganz sehen. Dazu muss Bello einen großen Abstand zum Objekt haben, wodurch die Höhe der Kamera nicht mehr relevant ist und für die Formel auf 0 gesetzt wurde. Nun folgt:

$$\begin{aligned} d_i &= h_i \cot \delta_i \\ &= h_i \cot \sqrt{(\alpha_{i1} - \alpha_{i2})^2 + (\beta_{i1} - \beta_{i2})^2} \\ &= h_i \cot |K_{i1} - K_{i2}| \end{aligned}$$

Wenn $z_i = (x_i, y_i)$ die Koordinaten der Objekte i auf dem Spielfeld sind und $z = (x, y)$ die Koordinaten von Bello, dann kann man die Constraints folgendermaßen definieren:

$$C_i = \{z = (x, y) | \, |z - z_i| = h_i \cdot \cot |K_{i1} - K_{i2}|\} \, , i \in \{F1, ..., P4\}$$

Für diese Berechnungen ist die Kamerastellung irrelevant. Diese könnte man dann benutzen um festzustellen, wohin Bello blickt, was ja noch nicht Teil der Aufgabe ist. Hierzu müsste man dann auch den Neigungswinkel mit einberechnen.

3 Aufgabenteil b)

Seien $\Delta\alpha_i = \alpha_{i1} - \alpha_{i2}$ und $\Delta\beta_i = \beta_{i1} - \beta_{i2}$. Der Fehler sei $\varepsilon := \pm 0{,}25°$.
Da die Tangensfunktion streng monoton wachsend und die Funktion $f(x) = \frac{1}{x}$ streng monoton fallend, kann man den Fehler ausrechnen:

$$d_{i\min} \leq |z - z_i| \leq d_{i\max}$$

mit

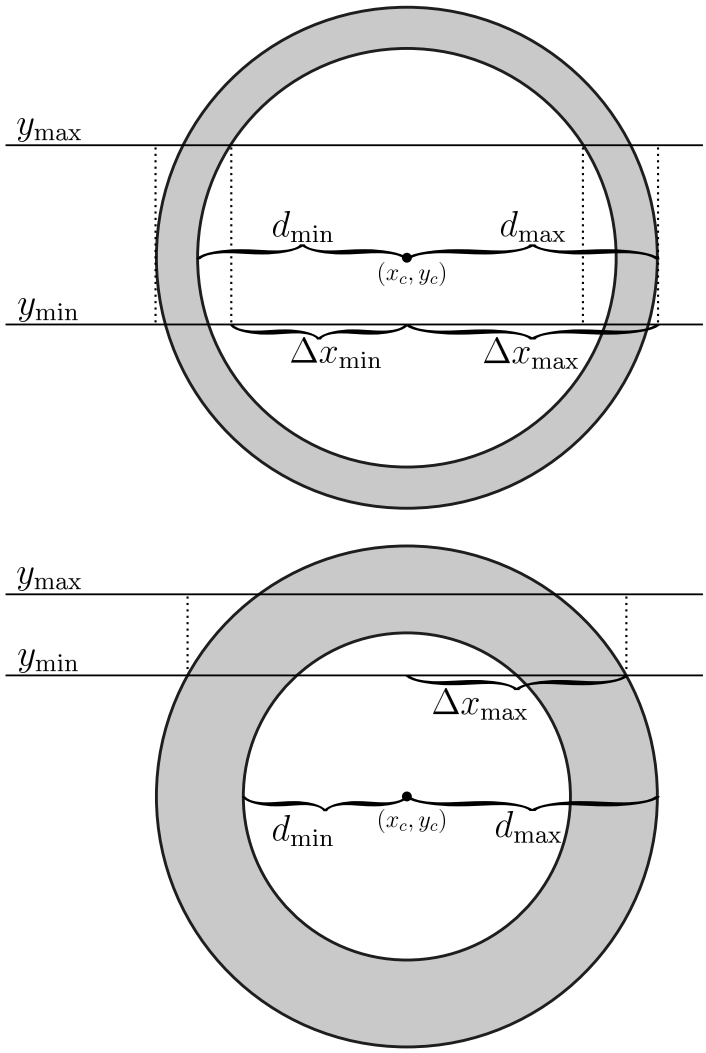
$$\begin{aligned} d_{i\min} &= h_i \cot \sqrt{(\Delta\alpha_i + \varepsilon)^2 + (\Delta\beta_i + \varepsilon)^2} \\ d_{i\max} &= h_i \cot \sqrt{(\Delta\alpha_i - \varepsilon)^2 + (\Delta\beta_i - \varepsilon)^2} \end{aligned}$$

Damit erhält man ein Intervall $d_i \in [d_{i\min}, d_{i\max}]$ für den möglichen Abstand d_i von Bello zum Objekt i. Es folgt:

$$C_i = \{z = (x, y) | d_{i\min} \leq |z - z_i| \leq d_{i\max}\} \, , i \in \{F1, ..., P4\}$$

4 Aufgabenteil c)

Das Programm verwendet den Constraintpropagierungsalgorithmus aus der Vorlesung, welcher mit den lokal maximal konsistenten Einschränkungen arbeitet. Allerdings verlangen wir, dass der Definitionsbereich rechteckig bleibt. Start-Definitionsbereich ist das gesamte Spielfeld. Aus den Bildern haben wir 3 Constraints extrahiert, jeweils für F1, F2 und F3. Unser Programm geht diese nun nacheinander durch und schränkt zuerst den x-Bereich und analog den y-Bereich ein. Nachdem es alle Constraints abgearbeitet hat, prüft es, ob sich etwas zum Anfangsrechteck geändert hat und fängt, falls dies der Fall war, wieder von vorne an mit dem eingeschränkten Rechteck.
Unsere Constraints entsprechen Kreisringen, wenn man x- und y-Koordinaten beliebig hat. Dazu ermiteln wir $\Delta x_{\min}$ und $\Delta x_{\max}$ durch Schnitt mit einem y-Schlauch. Dabei muss man dann ein paar Fälle unterscheiden:



Es kann passieren, dass unsere Einschränkung nicht maximal ist und wir unser Rechteck in zwei aufspalten müssten. Dieser Fall sollte sich aber erledigen, nachdem man die anderen Constraints betrachtet hat, was bedingt ist durch die verschiedenen Positionen der Objekte. Eine Möglichkeit zur maximalen Einschränkung wäre eine Problemzerlegung, da man 2 Rechtecke erhält, und dann kann man den Algorithmus auf die beiden Rechtecke loslassen und die Vereinigung aller Endrechtecke, wäre dann das Ergebnis, was man als Liste zum Beispiel ausgeben könnte.
Da dies in unserem Fall aber keine besseren Ergebnisse liefern würde, beschränken wir uns auf einfache Rechtecke.
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 $\Delta x_{\max} := d_{\max}$.

Der Algorithmus hat die positive Eigenschaft, dass er uns einen Bereich angibt, in dem Bello stehen kann unter den gegebenen Fehlern. Das heißt, er liefert uns nicht nur eine Position, sondern auch eine Fehlerabschätzung. Ist der Fehler zu groß, weiß Bello, dass er wohl noch weitere Bilder braucht.[1] Ist der Fehler klein genug, kann Bello seine Position als Rechteck-Mittelpunkt annehmen und damit weiter rechnen.

[1] Was für ein schlaues Kerlchen!

4.1 Programm-Ablauf

Namen der Variablen im Programm:

$$
\begin{aligned}
x_{\min} &= \texttt{x\_min} \\
x_{\max} &= \texttt{x\_max} \\
y_{\min} &= \texttt{y\_min} \\
y_{\max} &= \texttt{y\_max} \\
\Delta x_{\min} &= \texttt{dx\_min} \\
\Delta x_{\max} &= \texttt{dx\_max} \\
x_c &= \texttt{center\_x} \\
y_c &= \texttt{center\_y} \\
d_{\min} &= \texttt{dist\_min} \\
d_{\max} &= \texttt{dist\_max} \\
\varepsilon &= \texttt{abs\_error}
\end{aligned}
$$

Das Programm ist in Python geschrieben. Es kann auf dem Rechner `tegel.informatik.hu-berlin.de` gestartet werden via:

```
python aufgabe2.py
```

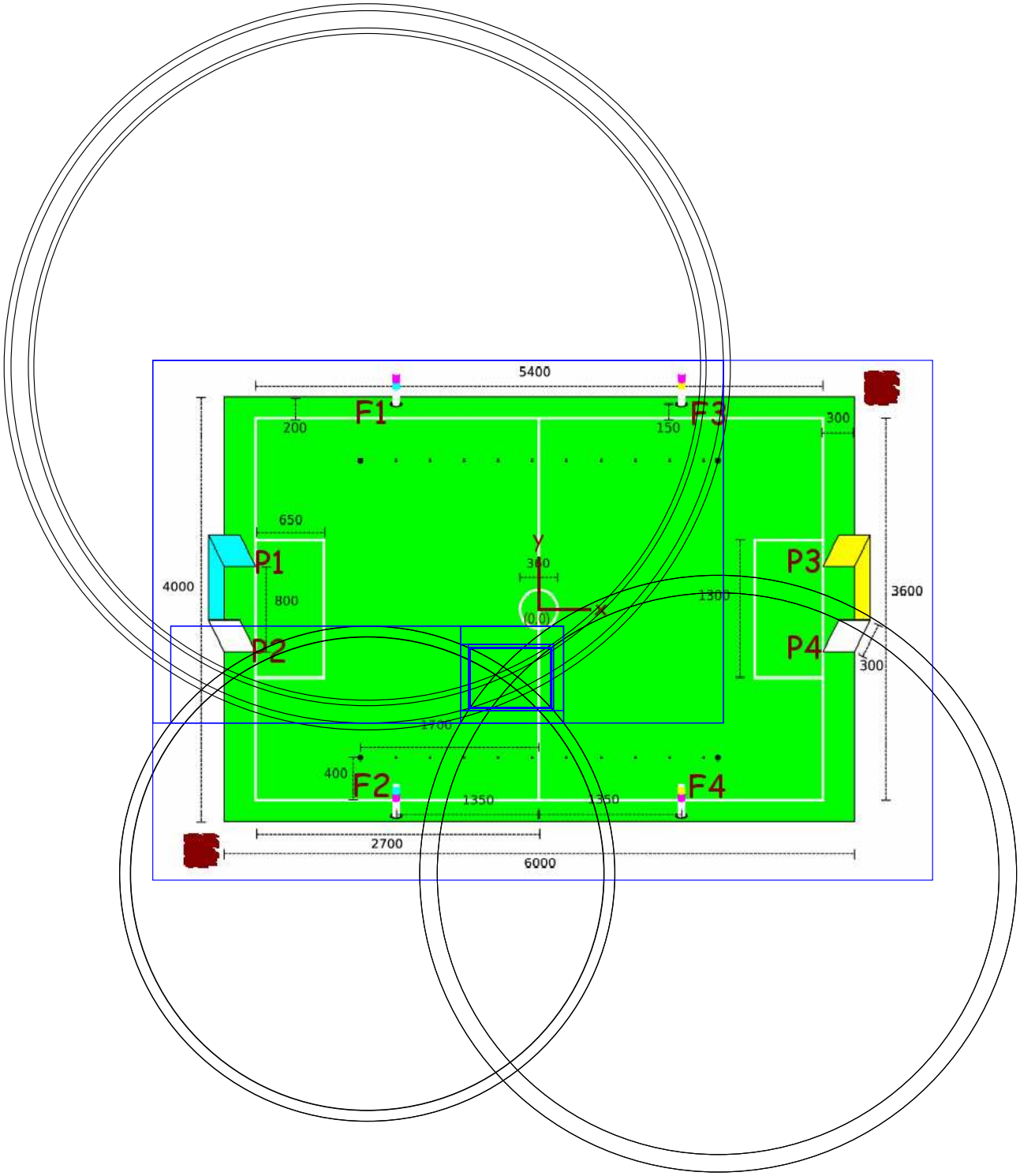
Es gibt dann auf der Konsole folgendes aus:

```
Vermutete Position von Bello:
x = -247.6 mm  (+/- 622.571720 mm)
y =  445.9 mm  (+/- 456.288184 mm)

Male SVG-Bildchen: aufgabe2-loesung.svg
```

Es erzeugt eine SVG-Datei, die z.B. in Inkscape geöffnet werden kann:

```
inkscape aufgabe-loesung.svg
```



Man sieht hier sehr schön, dass das Verfahren zwar eine maximal konsistente Einschränkung gefunden hat, diese jedoch nur lokal ist. Eine globale Einschränkung hingegen müsste den Schnitt aller Kreisscheiben genau umfassen.

4.2 Quellcode

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-


from math import sqrt
from math import radians
from math import tan
from math import hypot


#--------------------
#  Constraint-Löser
#--------------------

class Rectangle:

    """Rechteckiger Bereich."""

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

    def __cmp__(self, other):
        """Vergleiche zwei Rechtecke."""
        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))

    def to_svg(self, color=u"blue"):
        """SVG-Darstellung"""
        return u"""
            <rect x="%f" y="%f" width="%f" height="%f"
                  style="fill: none; stroke: %s; stroke-width: 7"/>""" \
            % (self.x_min, self.y_min,
               self.x_max - self.x_min, self.y_max - self.y_min, color)


class DistanceConstraint:

    """Constraint, das durch eine Abstands-Ungleichung beschrieben wird."""

    def __init__(self, center_x, center_y, dist_min, dist_max):
        """Initialisierung."""
        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

    def delta_x_min(self, dist, center_y, y_min, y_max):
```

```
    return sqrt(min(0, max(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))

def cut_x(self, dist_min, dist_max, center_x, center_y,
                x_min, x_max, y_min, y_max):
    """Ermittle den x-Bereich für die maximal konsistente Einschränkung.

    Funktioniert aus Symmetriegründen genauso auch für den y-Bereich.
    """
    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

def cut(self, rect):
    """Ermittle die maximal konsistente Einschränkung eines Rechtecks.

    Geometrisch entspricht dies dem umschließenden Rechteck der
    Schnittmenge zwischen diesem Constraint und dem Ausgangs-Rechteck.

    Argumente
    =========
        - rect -- Ausgangs-Rechteck, das eingeschränkt wird
    """
    x_min, x_max = self.cut_x(self.dist_min, self.dist_max,
                              self.center_x, self.center_y,
                              rect.x_min, rect.x_max,
                              rect.y_min, rect.y_max)
    y_min, y_max = self.cut_x(self.dist_min, self.dist_max,
                              self.center_y, self.center_x,
                              rect.y_min, rect.y_max,
                              rect.x_min, rect.x_max)
    return Rectangle(x_min, x_max, y_min, y_max)

def to_svg(self, color=u"black"):
    return u"""
        <circle cx="%f" cy="%f" r="%f"
```

```
                style="fill: none; stroke: %s; stroke-width: 7"/>
            <circle cx="%f" cy="%f" r="%f"
                style="fill: none; stroke: %s; stroke-width: 7"/>""" \
            % (self.center_x, self.center_y, self.dist_min, color,
               self.center_x, self.center_y, self.dist_max, color)


def solve_constraints(start, constraints):
    """Löse ein System von Constraints.

    Liefert ein Rectangle-Objekt zurück. Es beschreibt das kleinste
    Recheck, das die Lösungsmenge des Constraint-Systems umfasst.

    Außerdem werden die benötigten Zwischenschritte zurückgegeben.

    Argumente
    =========
        - start -- Ausgangs-Rechteck
        - constraints -- Liste von zu erfüllenden Constraints
    """
    steps = [start]
    rect = start
    while True:
        prev = rect
        for constraint in constraints:
            rect = constraint.cut(rect)
            steps.append(rect)
        if rect == prev:
            return rect, steps


#---------------------------
#  Constraint-Ermittelung
#---------------------------

class MapObject:

    """Ein Objekt (z.B. Flagge, Pfosten) auf dem Spielfeld."""

    def __init__(self, (x, y), h):
        self.x = x
        self.y = y
        self.h = h


def distance_to(height, angle):
    """Distanz zu einem gesehenen Objekt.

    Argumente
    =========
        - size      -- reale Größe des Objektes
        - angle     -- Winkelgröße des Objektes in Grad
    """
```

```
    return height / tan(radians(angle))


def post_constraints(obj, (a1, b1), (a2, b2), abs_error):
    """Constraints für einen vom Imageprozessor erkannten Pfosten.

    Liefert eine Liste von Constraint-Objekten zurück.

    Argumente
    =========
        - obj       -- zugehöriges MapObject
        - (a1, b1)  -- Winkelkoordinaten der oberen Ecke in Grad
        - (a2, b2)  -- Winkelkoordinaten der unteren Ecke in Grad
        - abs_error -- absoluter Fehler von Winkeldifferenzen in Grad
    """
    assert(abs(a1-a2) > abs_error)
    assert(abs(b1-b2) > abs_error)
    dist_min = distance_to(obj.h, hypot(abs(a1-a2) + abs_error,
                                        abs(b1-b2) + abs_error))
    dist_max = distance_to(obj.h, hypot(abs(a1-a2) - abs_error,
                                        abs(b1-b2) - abs_error))
    return [DistanceConstraint(obj.x, obj.y, dist_min, dist_max)]


def flag_constraints(obj, (a1, b1), (a2, b2), (a3, b3), (a4, b4), abs_error):
    """Constraints für eine vom Imageprozessor erkannte Flagge.

    Liefert eine Liste von Constraint-Objekten zurück.

    Argumente
    =========
        - obj       -- zugehöriges MapObject
        - (a1, b1)  -- Winkelkoordinaten der linken obere Ecke in Grad
        - (a2, b2)  -- Winkelkoordinaten der linken untere Ecke in Grad
        - (a3, b3)  -- Winkelkoordinaten der rechten obere Ecke in Grad
        - (a4, b4)  -- Winkelkoordinaten der rechten untere Ecke in Grad
        - abs_error -- absoluter Fehler von Winkeldifferenzen in Grad
    """
    return post_constraints(obj, (a1, b1), (a2, b2), abs_error) \
         + post_constraints(obj, (a3, b3), (a4, b4), abs_error)


#-------------
#  SVG-Ausgabe
#-------------

def print_svg(filename, objects):
    file = open(filename, "w")
    print >>file, """<?xml version="1.0"?>
        <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
                             "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
        <svg width="100%" height="100%" version="1.1"
             xmlns="http://www.w3.org/2000/svg"
```

```
            xmlns:xlink="http://www.w3.org/1999/xlink">

          <image x="-3000" y="-2000" width="6000" height="4000"
                 xlink:href="aufgabe2-spielfeld.png"/>"""
    for obj in objects:
        print >>file, obj.to_svg()
    print >>file, """
        </svg>"""


#-----------------
#  Hauptprogramm
#-----------------

def main():
    #  Wissen über das Spielfeld
    F1 = MapObject((-1350,  1950), 400)
    F2 = MapObject((-1350, -1950), 400)
    F3 = MapObject(( 1350,  1950), 400)
    F4 = MapObject(( 1350, -1950), 400)

    P1 = MapObject((-2700,  400), 300)
    P2 = MapObject((-2700, -400), 300)
    P3 = MapObject(( 2700,  400), 300)
    P4 = MapObject(( 2700, -400), 300)

    # Größe des Spielfeld (=Ausgangsmenge)
    start = Rectangle(-3000, 3000, -2000, 2000)

    # Constraints der vom Imageprozessor erkannten Objekte
    constraints = (
        flag_constraints(F2, (21.45, 23.17), (22.55, 31.68),
                             (23.38, 22.88), (24.75, 31.20), 0.25) +
        flag_constraints(F1, (42.31,  0.83), (41.78, 12.93),
                             (44.95,  0.83), (44.16, 12.93), 0.25) +
        flag_constraints(F3, (34.64, 20.08), (32.26, 29.98),
                             (36.49, 20.63), (34.11, 30.53), 0.25))

    # Berechne Position
    pos, steps = solve_constraints(start, constraints)

    # Text-Ausgabe
    print
    print "Vermutete Position von Bello:"
    print "x = %6.1f mm  (+/- %f mm)" % ((pos.x_max + pos.x_min) / 2,
                                         pos.x_max - pos.x_min)
    print "y = %6.1f mm  (+/- %f mm)" % ((pos.y_max + pos.y_min) / 2,
                                         pos.y_max - pos.y_min)
    print

    # SVG-Ausgabe
    filename = "aufgabe2-loesung.svg"
    print_svg(filename, constraints + steps)
```

```
    print "SVG-Bildchen:", filename
    print


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