LEGO-Spurter mit PD-Regler

Ronald Hecht

ronald.hecht@gmx.de

23. März 2006

Wer nicht so sicher mit dem Lötkolben umgehen kann und lieber die Finger von Transistoren, Operations- und Leistungsverstärkern lässt, wird in dieser Bauanleitung vielleicht auf seine Kosten kommen. Wir werden ein Spurtmodell mit dem LEGO® MINDSTORMS® System aufbauen.

Der ein oder andere hat sich bestimmt schon daran versucht und geärgert, dass ein exakt und ruhig fahrendes Modell einfach nicht hinzubekommen ist. Es geht aber doch! Mit einem sogenannten PD-Regler werden wir das Hin- und Herpendeln vollständig unterdrücken. Diesen programmieren wir in der Programmiersprache C. Das Modell wird exakt und dennoch zügig den Spurt-Parcours bewältigen. Natürlich werden wir dabei wieder auf die altbewährte differentielle Ansteuerung der Räder zurückgreifen. Allerdings werden wir auch sehen, dass das Bremsen nicht ganz so einfach wird.

1 Das Grundprinzip

Wie auch bei den Spurtmodellen mit Kopfhörerverstärker [5] ist es die Idee, mit nur einem Sensor die Drehzahl des linken und gleichzeitig des rechten Rades so zu regeln, dass das Modell ohne zu pendeln exakt auf der Bahn fährt. Abbildung 1 zeigt die Ansteuerung der Räder bei der Fahrt auf der Bahn. Befindet sich der Sensor genau mittig über der schwarz-weißen Trennungslinie, soll das Modell geradeaus fahren. Beide Motoren müssen mit gleicher Drehzahl laufen. Verschiebt sich die Trennungslinie nach links, handelt es sich um eine Linkskurve. Der Sensor empfängt mehr Licht. Das linke Rad muss langsamer und das rechte schneller laufen. Bei einer Verschiebung der Trennungslinie nach rechts befindet sich das Modell in einer Rechtskurve. Der Sensor empfängt weniger Licht. Das linke Rad muss schneller laufen und das rechte langsamer.

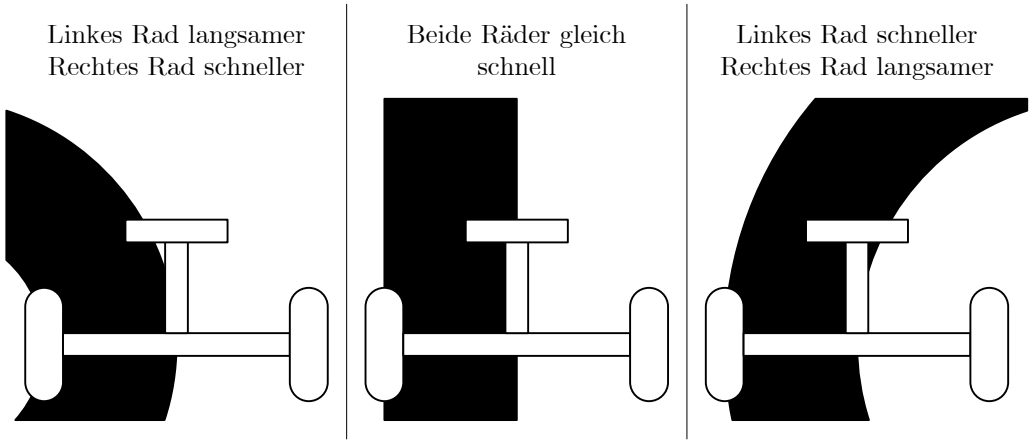


Abbildung 1: Grundprinzip des Modells

Da hier die Drehzahl der Räder entgegengesetzt erhöht und verringert wird, spricht man von einer differentiellen Ansteuerung. Dabei bestimmt das arithmetische Mittel der Drehzahlen der beiden Motoren die Geschwindigkeit des Modells und die Differenz der Drehzahlen die Enge der Kurve. Es ist aber nicht unbedingt erforderlich, bei der Linkskurve das rechte Rad und bei der Rechtskurve das linke Rad schneller laufen zu lassen. Man kann auch einfach die Drehzahl beibehalten und jeweils nur das andere Rad bremsen. Das Modell wird dann aber in der Kurve langsamer.

2 Die Mechanik

Schauen wir in unseren LEGO MINDSTORMS Kasten, finden wir eigentlich alles, was wir für unser Spurtmodell brauchen: Lichtsensor, Motoren, Räder und den RCX-Mikrocomputer. Wir sollten aber auch hier zuerst das Ziel verfolgen, so einfach, klein und stabil wie möglich zu bauen. Bei dem Modell darf nichts wackeln, insbesondere nicht die Räder und der Sensor. Abbildung 2 zeigt eine mögliche Aufbauvariante. Die Räder sind direkt auf den Motorachsen montiert. Beide Motoren sitzen eng beieinander. Auf der

Unterseite ist ein rundes Plättchen als Stütze angebracht. Ein oder zwei Sensoren können vorn auf den Winkel gesteckt werden. Oben wird der RCX-Mikrocomputer aufgesetzt. Er sorgt zusammen mit den Halblochbalken an der Seite für zusätzliche Stabilität.

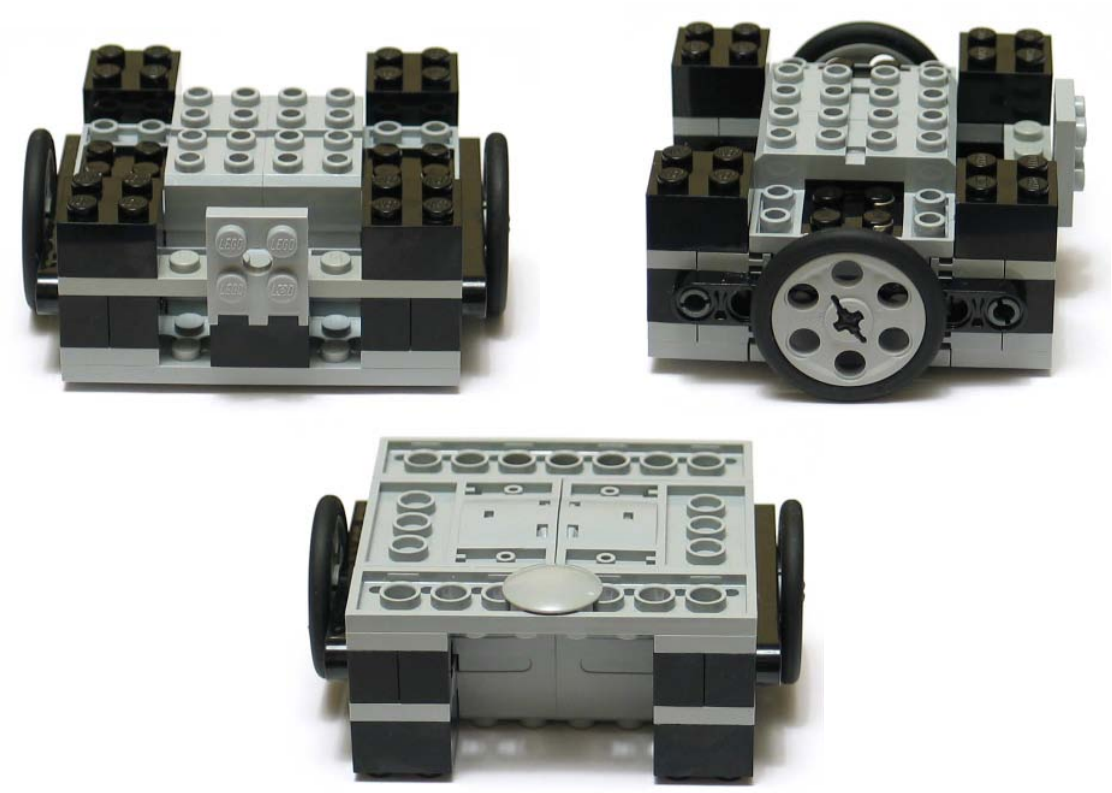


Abbildung 2: Die Mechanik des Modells

Natürlich kann dieses Modell noch nicht den Rundenrekord von 5 Sekunden sprengen. Die Motoren drehen bei Vollaussteuerung mit etwa $n = 300$ Umdrehungen pro Minute. Die Räder haben einen Durchmesser von $d = 3$ cm. Bei einer Bahnlänge von $s = 4{,}6$ m erhält man also bestenfalls eine Rundenzeit von

$$\begin{aligned} v &= \frac{s}{t} = \pi \cdot d \cdot n \\ t &= \frac{s}{\pi \cdot d \cdot n} \\ &= \frac{4{,}6 \text{ m}}{\pi \cdot 3 \text{ cm} \cdot 300 \text{ min}^{-1}} \\ &= 9{,}8 \text{ s} \end{aligned}$$

Das ist aber immerhin schon unter 10 Sekunden – also recht schnell. An dieser Stelle sei jeder aufgefordert, eine bessere Mechanik zu bauen. Viel Erfolg!

3 Die Programmierung des RCX

Mit der im Baukasten beiliegenden Software kann man den RCX-Mikrocomputer auf sehr einfache Weise programmieren. Man muss nur die richtigen Blöcke auf dem Bildschirm zusammenschieben, und schon macht der Roboter, was man will. Allerdings hat diese Herangehensweise Grenzen. Die Sensoren lassen sich nur recht ungenau auslesen. Die Leistung der Motoren kann lediglich in acht Stufen eingestellt werden. Tatsächlich lässt sich die Drehzahl damit kaum regeln. Auch mit der Programmiersprache NQC ist das nicht anders [3]. Zudem ist beides recht langsam. Wenn man ein schnelles Modell bauen will, muss man in Bruchteilen von Sekunden reagieren und die Drehzahl der Motoren exakt regeln können.

Die Lösung für diese Probleme heißt BrickOS [1]. Das ist ein spezielles, sehr kleines Betriebssystem für den RCX. Es stellt einen Ersatz für die originale Firmware dar. Mit BrickOS kann man in der Programmiersprache C programmieren. Dadurch werden die Programme sehr schnell ausgeführt. Außerdem kann man mit BrickOS die Leistung der Motoren in 256 Stufen einstellen. Das lässt doch hoffen!

Die Installation von BrickOS unter Windows ist recht einfach. Man sollte sich zuerst BricxCC installieren [2]. Das ist eine Entwicklungsumgebung, welche die Programmierung vereinfacht. Auf der Webseite von BricxCC findet man auch zwei Installationsdateien für BrickOS. Wenn man diese auch installiert hat, sollte der C-Programmierung nichts mehr im Wege stehen. BrickOS gibt es natürlich auch für Linux. Wer mit der Installation überhaupt nicht klar kommt, schreibt mir einfach eine E-Mail.

4 Erste Experimente

Versuchen wir, uns schrittweise dem Ziel zu nähern. Lesen wir erst einmal die Sensorwerte aus und zeigen sie auf der Anzeige an. Nun muss man wissen, dass die Sensorauflösung 10 Bit ist. Es gibt also $2^{10} = 1024$ verschiedene Werte. Um auf die Zahlenwerte 0 bis 1023 zu kommen, muss man den ausgelesenen Wert noch durch 64 dividieren oder mit der Operation `>> 6` um 6 Bits nach rechts verschieben. Um einen größeren Bereich auf der Bahn zu sehen, werden in dieser Bauanleitung, wie auf Seite 1 zu sehen ist, zwei voreinander gesetzte Sensoren verwendet. Die gemessenen Sensorwerte werden einfach addiert und in der Variable SollWert gespeichert.

```
#include <conio.h>
#include <unistd.h>
#include <dsensor.h>

int main(int argc, char *argv[]) {

  // Variable für den ausgelesenen Sensorwert
  int SollWert;

  // Sensoren einschalten
  ds_active(&SENSOR_1); ds_active(&SENSOR_3);
```

```
  // Sensorwert auslesen und anzeigen
  cputs("SOLL"); sleep(2);
  SollWert = (SENSOR_1 >> 6) + (SENSOR_3 >> 6);
  lcd_int(SollWert);

  return 0;
}
```

Den gespeicherten Sensorwert werden wir nun verwenden, um während der Fahrt die Abweichung von der schwarz-weißen Trennungslinie zu berechnen. Wenn das Modell losfahren soll, wird es also zuerst genau mittig über der Trennungslinie platziert. Das Programm wird gestartet, der Mittenwert ermittelt und in SollWert gespeichert. Die Abweichung wird während der Fahrt in der Variable Abweichung abgelegt. Die Motoren werden nun unter Verwendung der Abweichung entgegensetzt angesteuert. Mit dem Faktor K_p kann man einstellen, wie stark sich die Abweichung auf die Motoransteuerung auswirken soll.

```
#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>

#define Kp 4

#define SENSOR ((SENSOR_1 >> 6) + (SENSOR_3 >> 6))

int main(int argc, char *argv[]) {

  int IstWert, SollWert, Abweichung;
  int Links, Rechts;

  // Sensoren einschalten
  ds_active(&SENSOR_1); ds_active(&SENSOR_3);

  // Sensorwert auslesen, merken und anzeigen
  cputs("SOLL"); sleep(2);
  SollWert = SENSOR;
  lcd_int(SollWert); sleep(2);

  // Solange bis Programm gestoppt
  while (!shutdown_requested()) {
    // Sensoren auslesen
    IstWert = SENSOR;
    // Abweichung von der Trennungslinie ermitteln
    Abweichung = SollWert - IstWert;

    // Linken Motor ansteuern
    Links = 128 - Kp * Abweichung;
```

```
    // Auf Wertebereich begrenzen
    if (Links > 255) Links = 255;
    if (Links < 0) Links = 0;
    motor_a_dir(fwd);
    motor_a_speed(Links);

    // Rechten Motor ansteuern
    Rechts = 128 + Kp * Abweichung;
    // Auf Wertebereich begrenzen
    if (Rechts > 255) Rechts = 255;
    if (Rechts < 0) Rechts = 0;
    motor_c_dir(fwd);
    motor_c_speed(Rechts);

    // 5 Millisekunden warten
    msleep(5);
  }

  return 0;
}
```

Probiert man dieses Programm aus, wird man feststellen, dass das Modell ins Pendeln kommt und schließlich die Bahn verlässt. Wählt man K_p zu klein, kann es sich gar nicht auf der Bahn halten. Ist K_p zu groß, pendelt es stärker. Die Hauptursache dafür ist, dass die Motoren nicht gebremst werden. Ist der Wert Links oder Rechts klein, wird der Motor zwar nur schwach angesteuert aber eben nicht gebremst. Er läuft dann fast frei.

5 Bremsen ja – aber wie?

Die Leistungssteuerung von Motoren wird heute normalerweise nicht mit analogen Verstärkern realisiert, wie es in der Anleitung mit Kopfhörerverstärker [5] gezeigt ist. Vielmehr wird dazu der Motor sehr schnell hintereinander ein- und wieder ausgeschaltet. Je nachdem wie lang die Ein- und Ausphase ist, wird dem Motor mehr oder weniger Leistung zugeführt. Dies wird Pulsweitenmodulation [7] genannt und wird häufig zusammen mit einer sogenannten H-Brücke kombiniert. Mit ihr kann man die Drehrichtung des Motors auch umkehren. Genau eine solche Schaltung ist auch im RCX zu finden.

Bremsen kann man damit natürlich auch. Am einfachsten geht das, wenn man den Motor einfach umgekehrt ansteuert. So kann man einerseits die Bremskraft einstellen, aber auch rückwärts fahren oder das Modell auf der Stelle drehen lassen. Genau das ist im folgenden Programmteil umgesetzt.

```
    // Linken Motor ansteuern
    Links = 128 − Kp * Abweichung;
    // Auf Wertebereich begrenzen
    if (Links > 255) Links = 255;
    if (Links < −255) Links = −255;
    // Auf Richtung prüfen
```

```
if (Links < 0 ) {
  motor_a_dir(rev);
  motor_a_speed(−Links);
} else {
  motor_a_dir(fwd);
  motor_a_speed(Links);
}

// Rechten Motor ansteuern
Rechts = 128 + Kp * Abweichung;
// Auf Wertebereich begrenzen
if (Rechts > 255) Rechts = 255;
if (Rechts < −255) Rechts = −255;
// Auf Richtung prüfen
if (Rechts < 0 ) {
  motor_c_dir(rev);
  motor_c_speed(−Rechts);
} else {
  motor_c_dir(fwd);
  motor_c_speed(Rechts);
}
```

Wenn man das in sein Programm einbaut, wird man feststellen, dass das Bremsen zwar funktioniert, sich das Modell aber noch ruckartiger bewegt. Zumindest hält es schon mal die Spur! Der entscheidende Fehler hierbei ist, dass erst gebremst wird, wenn der Sensor schon viel zu weit von der schwarz-weißen Trennungslinie abgewichen ist. Wir müssen also irgendwie früher bremsen, also bevor der Sensor die Spur verlässt. Ein Vergrößern des Faktors K_p macht die Sache aber leider auch nicht besser. Was dann?

6 PID-Regelung

Das Geheimnis vieler gut funktionierender Regelschaltungen ist ein PID-Regler. Dabei steht P für Proportional, I für Integral und D für Differential. Klingt kompliziert und nach Fachchinesisch. Im Prinzip ist es aber ganz einfach und recht leicht in Software zu programmieren.

Um die Funktionsweise eines solchen Reglers zu verstehen, schauen wir uns erst einmal an, was wir bisher gemacht haben. Abbildung 3 zeigt das Funktionsprinzip unseres Spurtmodells. Wir vergleichen ständig unseren zuvor gemessenen Sollwert mit dem aktuellen Sensorwert – dem Istwert. Die Abweichung multiplizieren wir mit dem Faktor K_p. Damit steuern wir unsere beiden Motoren entgegengesetzt an. Den Geschwindigkeitswert haben wir bisher immer auf 128 gelassen. Man kann hier natürlich auch mal mit anderen Werten experimentieren.

Auch wenn das Bild ganz genau zeigt, wie das Modell funktioniert, ist es etwas unübersichtlich. Man kann es, wie in Abbildung 4 zu sehen ist, auch vereinfacht darstellen. So wird das häufig in der Regelungstechnik gemacht [8]. Der Istwert wird mit dem Sollwert verglichen. Die Abweichung wird in einen Regler gegeben, der daraus eine

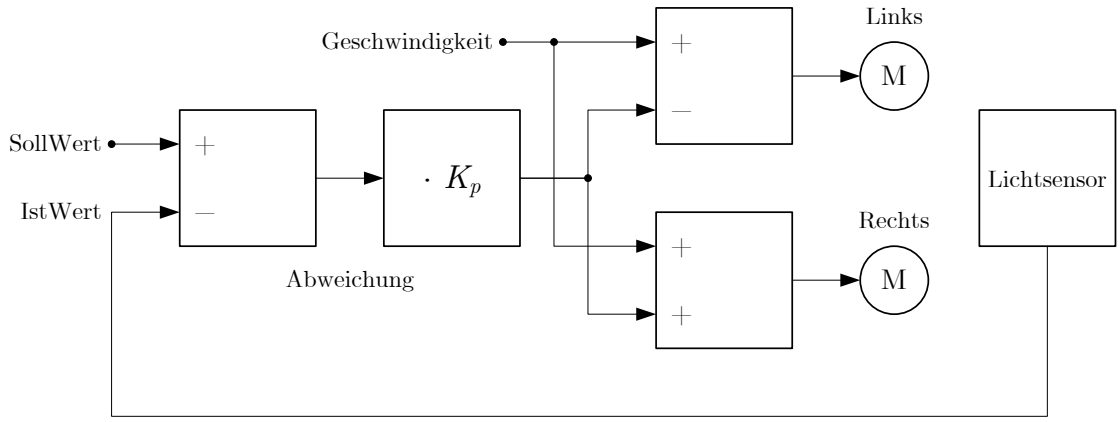
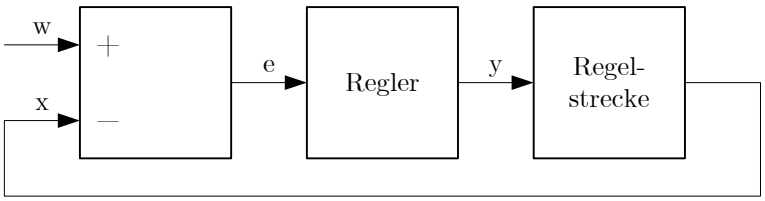


Abbildung 3: Das Spurtmodell bisher



w: Sollwert x: Istwert e: Regelabweichung y: Stellgröße

Abbildung 4: Prinzip einer Regelung

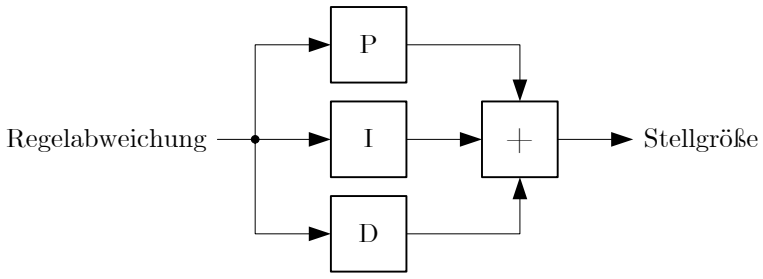


Abbildung 5: Aufbau eines PID-Reglers

Stellgröße berechnet. Diese Stellgröße wird in unserem Fall auf die Motoren gegeben. Der Lichtsensor ermittelt den Istwert. Unsere Regelstrecke setzt sich also aus den Motoren, dem Lichtsensor und den mechanischen Eigenschaften des Modells zusammen.

Werfen wir nun einen genauen Blick auf den Regler. Bisher haben wir dort nur die Multiplikation mit K_p. In einem PID-Regler werden jedoch drei Operationen durchgeführt, deren Ergebnisse anschließend aufaddiert werden (vgl. Abbildung 5). Diese drei Operationen – P, I und D – sollen nun erklärt werden. Dazu werden wir das Beispiel einer Füllstandsregelung wie in Abbildung 6 heranziehen. Das ist noch einfacher zu verstehen, als unser Spurtmodell. Ein Behälter mit Wasser wird von einem Füllstandssensor überwacht. Mit einem Regler soll nun der Zulauf so gesteuert werden, dass sich eine bestimmte Füllhöhe einstellt. Aus dem Behälter kann auch Wasser ablaufen. Dabei soll die Füllhöhe ebenfalls konstant gehalten werden.

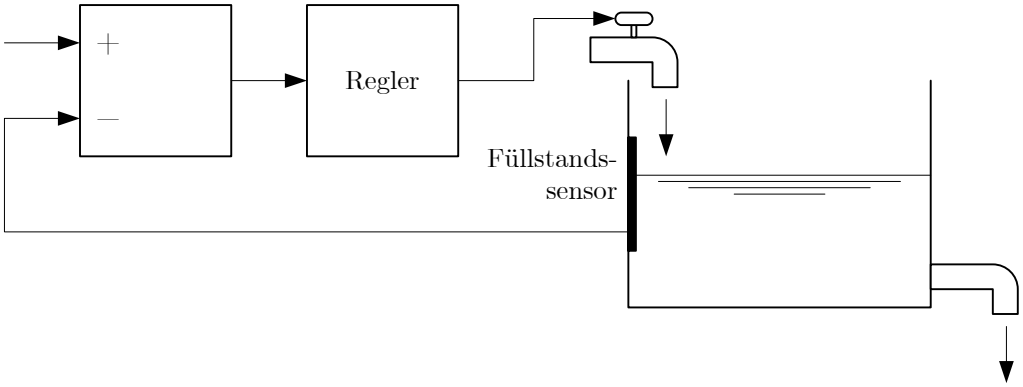


Abbildung 6: Füllstandsregelung

6.1 P – Der Proportionalteil

Den Proportionalteil eines PID-Reglers haben wir eigentlich schon kennengelernt. Das ist die Multiplikation mit K_p. Die Stellgröße wird proportional zur Regelabweichung verändert. Bei der Füllstandsregelung bedeutet dies, dass der Wasserhahn mehr aufgedreht wird, wenn eine große Abweichung vom Sollfüllstand vorliegt. Ist die Abweichung nur sehr gering, wird der Hahn nur ein wenig aufgedreht. Klingt eigentlich plausibel. In einem Programm sieht das dann so aus:

```
StellGroesse = Kp * Abweichung;
```

Angenommen, der aktuelle Füllstand ist zu niedrig. Dann wird der Wasserhahn voll aufgedreht. Wasser läuft in den Behälter. Langsam nähert sich der Wasserspiegel dem gewünschten Füllstand. Gleichzeitig wird durch den Proportionalteil des Reglers langsam der Hahn zugedreht, bis schließlich der Sollfüllstand erreicht ist.

6.2 I – Der Integralteil

Um zu verstehen wozu man den Integralteil braucht, stellen wir uns vor, dass ständig Wasser aus dem Behälter abfließt. Der Füllstand wird erstmal sinken. Der Proportionalteil sorgt dafür, dass der Hahn aufgedreht wird. Wasser fließt nach. Es wird sich dann ein Zustand einstellen, bei dem genauso viel Wasser in den Behälter rein- wie auch herausfließt. Der Sollfüllstand wird aber nicht wieder erreicht. Das liegt daran, dass der Wasserhahn nur dann aufgedreht wird, wenn eine Abweichung vom Sollfüllstand vorliegt. Der Wasserhahn muss aber immer aufgedreht sein, da ständig Wasser abfließt. Es ergibt sich also eine bleibende Regelabweichung.

Mit dem Integralteil eines PID-Reglers kann man diese bleibende Regelabweichung beseitigen. Er addiert Regelabweichungen ständig auf. Bei einer bleibenden Regelabweichung wird diese Summe also immer größer. Die Summe, oder auch das Integral, wird mit einem Faktor K_i multipliziert und, wie Abbildung 5 zeigt, zum Proportionalteil addiert. In Software kann man das zum Beispiel so realisieren:

```
// Integrieren (Aufsummieren)
Summe = Summe + Abweichung;
// Begrenzen
if (Summe < −255) Summe = −255;
if (Summe > 255) Summe = 255;
// Proportional− und Integralteil addieren
StellGroesse = Kp ∗ Abweichung + Ki ∗ Ta ∗ Summe;
```

Hier wird der Integralteil noch begrenzt. Das sollte man unbedingt machen, damit kein Wertüberlauf entsteht. Zusätzlich zu K_i wurde die Summe noch mit T_a multipliziert. Das ist die Abtastzeit, also die Zeit zwischen den Messintervallen. Bei uns waren das bisher 5 ms. Da hier Proportional- *und* Integralteil für die Stellgröße verwendet werden, nennt man das einen PI-Regler.

Schaltet man bei der Füllstandsregelung mit reinem Proportionalregler nun den Integralteil hinzu, würde dieser durch das ständige Aufsummieren die bleibende Regelabweichung erkennen und den Wasserhahn noch weiter aufdrehen, so lange bis der Sollfüllstand erreicht ist. Der Proportionalteil würde auf Null gehen. Der Integralteil sorgt aber weiterhin für nachfließendes Wasser.

6.3 D – Der Differentialteil

Der Differentialteil macht den PID-Regler schnell und lässt ihn vorausschauend arbeiten. Entnehmen wir kurzzeitig eine große Wassermenge aus dem Behälter, sollte der Sollfüllstand so zügig wie möglich wieder erreicht werden. Der Differentialteil ermittelt dazu die Änderung der Regelabweichung in einem bestimmten Zeitintervall. Er berechnet also den Anstieg, oder auch Differential genannt. In Software kann man das so umsetzen:

```
// Proportional− und Differentialteil addieren
StellGroesse = Kp ∗ Abweichung + Kd ∗ (Abweichung − AbweichungAlt) / Ta;
// Abweichung für nächstes Zeitintervall merken
AbweichungAlt = Abweichung;
```

Für die Berechnung der Differenz muss man sich die aktuelle Regelabweichung für das nächste Mal merken. Hier taucht auch wieder die Abtastzeit T_a auf. Die Stellgröße wird in Kombination mit dem Proportionalteil berechnet. Man spricht deshalb auch von einem PD-Regler.

Ändert sich also der Füllstand sehr schnell, wird schon mal vorausschauend der Wasserhahn weit aufgedreht. Das passiert auch dann, wenn noch gar keine große Abweichung vom Sollfüllstand vorliegt. Es geht hier ja um die Änderung des Füllstandes pro Zeiteinheit! Das hört sich fast wie künstliche Intelligenz an. Tatsächlich machen wir es genauso: auf dem Fahrrad, im Auto, beim zu Fuß gehen und sogar unter der Dusche. Merken wir, dass das Wasser nur etwas kälter wird, drehen wir schon mal vorsichtshalber mehr warmes hinzu.

6.4 Fassen wir zusammen

Ein PID-Regler setzt sich also aus drei Teilen zusammen, die aufsummiert werden. Der Proportionalteil ermittelt proportional zur Regelabweichung die Stellgröße. Er kann aber keine bleibende Regelabweichung korrigieren, die durch eine Störung verursacht wurde. Dies lässt sich mit dem Integralteil realisieren. Der Differentialteil macht den Regler schnell und vorausschauend. In Software kann man einen PID-Regler etwa so programmieren:

```
// Integrieren (Aufsummieren)
Summe = Summe + Abweichung;
// Begrenzen
if (Summe < −255) Summe = −255;
if (Summe > 255) Summe = 255;
// Proportional−, Integral− und Differentialteil addieren
StellGroesse = Kp * Abweichung
             + Ki * Ta * Summe
             + Kd * (Abweichung − AbweichungAlt) / Ta;
// Abweichung für nächstes Zeitintervall merken
AbweichungAlt = Abweichung;
```

Alle drei Teile können mit den Faktoren K_p, K_i und K_d eingestellt werden. Das geht natürlich nur in bestimmten Grenzen. Zuerst sollte man K_i und K_d auf Null setzen und K_p soweit erhöhen, dass das System gerade nicht instabil wird, also nicht schwingt. Dann kann man den Integralteil zuschalten und zum Schluss den Differentialteil. Es gibt eine Vielzahl von Einstellregeln. Wer diese genauer kennen lernen will, schaut unter [8] nach.

7 Die Regelung des Spurtmodells

Wenden wir uns wieder dem Spurtmodell zu. Was für einen Regler brauchen wir nun? Mit dem Proportionalteil sind wir schon recht weit gekommen. Allerdings haben wir festgestellt, dass immer erst gebremst wird, wenn es schon zu spät ist. Wir müssen also vorausschauend reagieren. Wie wir gerade gelernt haben, geht das mit dem Differentialteil. Bauen wir den in unser Programm ein, könnte das so aussehen:

```
#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>

#define Kp 4
#define Kd 20

#define SENSOR ((SENSOR_1 >> 6) + (SENSOR_3 >> 6))

int main(int argc, char *argv[]) {

  int IstWert, SollWert, StellGroesse;
  int Abweichung = 0, AbweichungAlt = 0;
  int Links, Rechts;

  // Sensoren einschalten
  ds_active(&SENSOR_1); ds_active(&SENSOR_3);

  // Sensorwert auslesen, merken und anzeigen
  cputs("SOLL"); sleep(2);
  SollWert = SENSOR;
  lcd_int(SollWert); sleep(2);

  // Solange bis Programm gestoppt
  while (!shutdown_requested()) {
    // Sensoren auslesen
    IstWert = SENSOR;
    // Abweichung ermitteln
    Abweichung = SollWert - IstWert;

    // PD Regler
    StellGroesse = Kp * Abweichung + Kd * (Abweichung - AbweichungAlt);

    // Abweichung merken
    AbweichungAlt = Abweichung;

    // Linken Motor ansteuern
    Links = 255 - StellGroesse;
    // Auf Wertebereich begrenzen
    if (Links > 255) Links = 255;
    if (Links < -255) Links = -255;
    // Auf Richtung prüfen
    if (Links < 0 ) {
      motor_a_dir(rev);
      motor_a_speed(-Links);
    } else {
      motor_a_dir(fwd);
      motor_a_speed(Links);
```

```
    }

    // Rechten Motor ansteuern
    Rechts = 255 + StellGroesse;
    // Auf Wertebereich begrenzen
    if (Rechts > 255) Rechts = 255;
    if (Rechts < -255) Rechts = -255;
    // Auf Richtung prüfen
    if (Rechts < 0 ) {
      motor_c_dir(rev);
      motor_c_speed(-Rechts);
    } else {
      motor_c_dir(fwd);
      motor_c_speed(Rechts);
    }

    // 5 Millisekunden warten
    msleep(5);
  }

  return 0;
}
```

Und es funktioniert! Das Modell fährt, ohne zu schwingen, genau über der Trennungslinie. Wenn es nicht gleich klappt, oder man nur einen Lichtsensor hat, muss man mit den Faktoren K_p und K_d experimentieren. Den Integralteil benötigen wir beim Spurtmodell nicht, da uns die bleibende Regelabweichung in der Kurve nicht wirklich stört. Nur wenn man will, dass der Sensor auch in der Kurve genau mittig über der Trennungslinie bleiben soll, muss man den Integralteil einbauen. Das ist aber nicht so einfach.

8 Wie geht's weiter?

Natürlich ist dieses Modell noch weit entfernt vom Optimum und dem Bahnrekord. Da die Drehzahl der Motoren begrenzt ist, muss man eine andere Mechanik bauen – größere Räder oder ein Getriebe. Sinnvoll ist sicherlich auch, die Parameter K_p und K_d über Tasten einstellbar zu machen. Dafür gibt es die Funktionen `getchar()`, `dkey_pressed()` und `dkey_released()`. Wenn man die Räder nicht unbedingt rückwärts drehen lassen muss, kann man zum Bremsen auch mal `motor_a_dir(brake)` ausprobieren.

Eine gute Regelung steht und fällt mit der Sensorik. Die Lichtsensoren aus dem Baukasten sind nicht gerade das Ideale. Entweder man baut sich selbst einen [4] oder besorgt sich einen besseren [6].

Literatur

[1] *BrickOS*. http://brickos.sourceforge.net/.

[2] *Bricx Command Center*. http://bricxcc.sourceforge.net/.

[3] *NQC – Not Quite C*. http://bricxcc.sourceforge.net/nqc/.

[4] Gasperi, M.: *MindStorms RCX Sensor Input Page*. http://www.plazaearth.com/usr/gasperi/lego.htm.

[5] Hecht, R.: *Spurtmodelle mit Kopfhörerverstärker*. http://spurt.uni-rostock.de/.

[6] HiTechnic. http://www.hitechnic.com/.

[7] RoboterNETZ: *Pulsweitenmodulation*. http://www.roboternetz.de/wissen/.

[8] RoboterNETZ: *Regelungstechnik*. http://www.roboternetz.de/wissen/.