Ein Regler zur Kollisionsvermeidung von Flugrobotern

Jürgen Eckert, Bernd Hügel, Reinhard German und Falko Dressler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Informatik 7, Martensstraße 3, 91058 Erlangen
juergen.eckert@cs.fau.de, bernd.huegel@web.de, german@cs.fau.de,
dressler@cs.fau.de

Zusammenfassung. Fliegende Roboter werden in letzter Zeit vermehrt nicht nur im Außenbereich, sondern auch innerhalb von Gebäuden eingesetzt. Durch den fehlenden Bodenkontakt können Hindernisse wie z.B. Treppen ohne mechanischen Aufwand überwunden werden. Für rollende Plattformen triviale Operationen, wie über einem Ort in der Luft stehen zu bleiben und nicht mit Hindernissen zu kollidieren, sind hier allerdings komplexe Aufgaben. Schwebende Flugobjekte befinden sich auf einem Luftkissen, das vor allem innerhalb von Gebäuden durch starke Verwirbelungen sehr instabil ist. Dies macht selbst eine stabiles Halten der Position ohne zusätzliche Sensorik oder ohne externes Positionierungssystem schwierig bis unmöglich. In dieser Arbeit wird eine Sensorik vorgestellt, die so genannte Quadrokopter (vier-rotoriges Flugobjekt) erweitert, so dass Hindernisse erkannt und Kollisionen vermieden werden können. Neben einer kostengünstigen und leichtgewichtigen Hardware wurde ebenfalls ein einfacher Regler (auf die Rechenkomplexität bezogen) implementiert, der selbst auf sehr kleine Mikrocontroller portiert werden kann. Aufgrund der geringen Sensorreichweite ist die maximale Geschwindigkeit des fliegenden Objekts stark eingeschränkt.

1 Einleitung

Fliegende Roboter werden seit einigen Jahren nicht nur für die autonome Erkundung im Außenbereich, sondern auch für den Einsatz in Gebäuden immer interessanter. Ihr Vorteil gegenüber fahrenden Systemen im unwegsamen Gelände ist schnell ersichtlich. Innerhalb von Gebäuden gibt es ebenfalls eine Reihe von Hindernissen, die rollende Einheiten nur schwer überwinden können. Um beispielsweise Treppen befahren zu können, bedarf es eines enormen zusätzlichen mechanischen Aufwands. Für fliegende Objekte sind solche Hindernisse nicht relevant. Allerdings birgt diese neue Technologie durch den physikalisch sehr instabilen Zustand des Fliegens oder Schwebens nicht nur Vorteile. Scheinbar triviale Aktionen wie anhalten und in einem Punkt verharren werden hier zur Herausforderung.

Bevorzugt werden für die Luftaufklärung sogenannte Quardrokopter verwendet. Zwei gegenläufige Rotorpaare (somit insgesamt vier Rotoren) gleichen jeweils gegenseitig ihr Drehmoment aus, wodurch auf einen Heckrotor verzichtet

werden kann. Das energiesparende und gleichzeitig kompakte Konzept ist aus physikalischer Sicht allerdings höchst instabil. Ohne eine Sensorik zur Lageregelung kann es vom Menschen nicht gesteuert werden. Um alle drei Freiheitsgrade (Winkel in 3D) stabilisieren zu können, wird üblicherweise eine Kombination aus Beschleunigungs- und Drehwinkelsensoren verwendet. Diese beobachten den ersten und zweiten Zustand des Systems. Ein Regler reagiert auf unerwünschte Lageänderungen, indem er die Drehzahl (und somit indirekt den Schub) der Rotoren kontinuierlich anpasst. Diese stabilisierte Plattform kann relativ leicht für Szenarien verwendet werden, die autonomes Verhalten der Flugroboter verlangen. Anders als bei konventionellen Helikoptern, besitzen sie einen zentralen Prozessor, der alle Aktionen koordiniert. Dieser verfügt meistens standardmäßig über eine digitale Kommunikationsschnittstelle, über welche Steuerbefehle gesendet und Informationen abgerufen werden können. Zusätzlicher Konstruktionsaufwand wird somit minimiert.

Abgesehen von der Einhaltung der Flughöhe, die mit den verbauten Sensoren nicht genügend überwacht werden kann, bedarf es mehr für die Positionshaltung als eine ständige (perfekte) horizontale Lage des Systems. Zum einen driftet der Quadrokopter selbst unter optimaler Einhaltung der Horizontalen, da er noch einen gewissen Restimpuls besitzt. Zum anderen kommt es durch Messfehler und durch Luftverwirbelungen ständig zu Korrekturen, die eine Positionsverschiebung mit sich führen. Vor allem in engen Bereichen innerhalb von Gebäuden wird das auf einem Luftkissen schwebende Objekt sehr stark durch sich bildende Luftwirbel abgelenkt.

Daher ist ein System zur Kollisionsvermeidung von besonderem Interesse. Prinzipiell wird zwischen zwei Hindernisarten unterschieden. Statische oder unbewegliche Hindernisse sind Gegenstände wie Wände oder Säulen, die mittels einer *a priori* vorgegebenen Karte und einer Positionserkennung erkannt und umflogen werden können. Die zweite Art von Hindernissen sind dynamische oder bewegliche Hindernisse wie Personen oder auch Türen. Da diese nicht im Vorfeld behandelt werden können, muss ihnen situationsbedingt ausgewichen werden.

Die Grundidee dieser Arbeit ist es einen Quadrokopter so zu erweitern, dass autonome Flüge ohne Kollisionen ermöglicht werden. Neben den für fliegende Objekte typischen Restriktionen wie Gewicht und Energieeffizienz wird im folgenden noch zusätzlich eine kostengünstige und einfache Methodik betrachtet. Die benötigten Softwarekomponenten sind möglichst einfach aufgebaut, um selbst auf kleinen 8 Bit-Systemen mit der benötigten Geschwindigkeit ausgeführt werden zu können.

2 Verwandte Arbeiten

Kollisionsvemeidende Plattformen stehen in den letzten Jahren immer häufiger im Mittelpunkt der Forschungsarbeiten. Zumeist bestehen sie aus einem 360°-Laserscanner und einer sehr leistungsstarken on-board Recheneinheit [1,2]. Üblicherweise wird die Hardware dann nicht nur zur Kollisionsvermeidung verwendet, sondern zur Selbstlokalisation und zur Kartenbildung (SLAM). Dafür

sehr gut geeignete Hardware ist schwer und konsumiert viel Energie, was die Flugzeit negativ beeinflusst. Deshalb wird in dieser Arbeit untersucht, welche Komponenten (Hardware und Software) für ein minimales System nötig sind.

Java-basierte Sensorknoten eignen sich nicht sehr gut für Echtzeit Systeme, wie Chen *et al.* [3] aufzeigen. Viele zeitkritische Schranken können wegen der automatisierten Speicherverwaltung nicht vollständig eingehalten werden. Im weiteren Verlauf wollen wir allerdings aufzeigen, dass nicht jede Frist strikt eingehalten werden muss, da die Massenträgheit des Systems einige Zeitfehler kompensieren kann.

3 Erweiterung des Quadrokopters

Handelsübliche Qudarokopter verfügen nicht über die Möglichkeit ihr Umfeld wahrzunehmen. Die verbaute Sensorik beschränkt sich rein auf die Lage- und Zustandsdetektion. Gesteuert werden sie über eine Fernbedienung aus dem Modellflugbereich. Wie Eingangs bereits erwähnt, verfügen die meisten Modelle (wie die von uns verwendeten Exemplare) standardmäßig über einen seriellen Eingang, über welchen Flugbefehle übermittelt werden können. Die von uns zur Demonstration verwendet Quadrokopter sind der Hummingbirg von AscTec und der M3-MK von Mikrokopter. Trotz des erhöhten Abfluggewichts weisen beide (je nach Akkuleistung) eine Schwebeflugzeit von 10 min bis 15 min auf.

Für die Erkennung des Umfeldes wird, wie in Abschnitt 2 bereit erwähnt, meist ein Rundum-Laser-Abstandsscanner verwendet. Diese sind allerdings nicht nur relativ schwer ($\geq$141 g), sondern auch vergleichsweise teuer. Zudem könnten langsame Prozessoren durch die hohe Genauigkeit und schnelle Messfrequenz überlastet werden. Daher fiel unsere Wahl auf günstige Abstandssensoren von Sharp (GP2Y0A02YK [4]). Ihr Messbereich liegt zwischen 20 cm und 120 cm. Die analoge Ausgangsspannung ist indirekt proportional zum gemessenen Abstand. Vorteil dieses Sensors ist, dass die Messungen kontinuierlich durchgeführt werden und die Ergebnisse zu einer beliebigen Zeit abgerufen werden können. Nachteilig wirkt sich das sehr geringe Gefälle der Spannungs-Abstandskurve in der letzten Hälfte aus. Dies hat zur Folge, dass Entfernungen $\geq$80 cm nicht mehr vernünftig gemessen werden können. Das Problem liegt teils im Sensor selbst und teils in der unsauberen Versorgungsspannung (die Rotoren setzen circa 100 W um) begründet. Der Sensortyp hat eine internes festes Messintervall von 38.3 ms $\pm$ 9.6 ms. Für das Rotieren des Sensors wie bei einem Radargerät ist er zu langsam. Daher wurden acht Sensoren des gleichen Typs auf einem Ring mit 15 cm Durchmesser gleichmäßig (45° zueinander) verteilt und auf dem Quadrokopter befestigt. Um die analogen Werte zu digitalisieren wird ein dedizierter Prozessor verwendet, der mittels einer I^2C-Schnittstelle angesprochen werden kann.

Als Hauptprozessor wurde ein SunSpot der Firma Sun Microsystems verwendet [5]. Die eigentlich als Sensorknoten vorgesehene Plattform verfügt neben der klassischen IEEE 802.15.4 Funkschnittstelle noch über alle weiteren hier benötigten Schnittstellen. Als Betriebssystem kommt eine Java VM (Virtual Machine) zum Einsatz. Java gilt, nicht zuletzt wegen des Garbage Collectors, nicht als

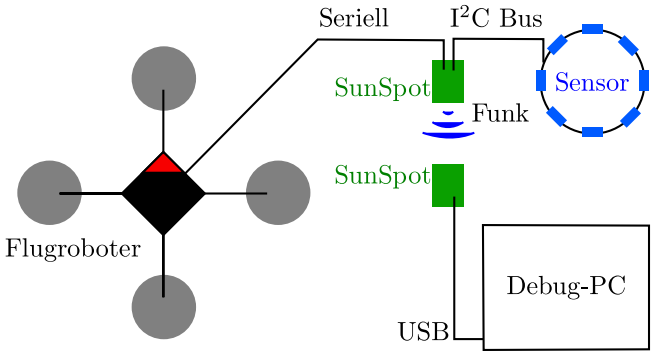

Abb. 1. Schematischer Aufbau aller Komponenten

echtzeitfähig. Aus diesem Grund wird in Abschnitt 5 genauer auf die Echtzeitfähigkeit des Knotens und die Anwendbarkeit auf dieses Systems eingegangen.

Abbildung 1 zeigt den schematischen Aufbau aller Komponenten. Der SunSpot ist wohl mit der Sensorik als auch mit der Aktorik (Quadrokopter) verbunden. Die Funkschnittstelle der Knotens wird als kabellose Testüberwachung des Systems verwendet.

4 Kollisionsvermeidung

Für die Kollisionsvermeidung benötigt man ein in sich geschlossenes System bestehend aus Sensoren und Aktoren (*hier:* Quadrokopter). Sensoren messen den physikalischen Ist-Zustand, dieser wird mit einer Referenz (Sollwert) verglichen. Daraus resultiert dann schließlich eine Stellgröße für die Aktoren (*hier:* Neigungswinkel). Ein solches System wird als Regelkreis bezeichnet. Folgende Bezeichnungen und weiterführende Informationen zur Regelungstechnik sind dem Buch von Horn und Dourdmoumas [6] zu entnehmen.

4.1 Interpretation der Messwert

Gemäß der Ausrichtung $\overrightarrow{e}_i$ des i-ten Sensors (Sicht von oben auf den Flugroboter) wird dessen Abstandswert d_i in seinen Roll- und Nick-Anteil zerlegt und mit allen anderen verrechnet. Dadurch entsteht ein zweidimensionaler (Nick/Roll) Flucht-Vektor $\overrightarrow{y}$:

$$\overrightarrow{y} = \sum_{i=0}^{7} \overrightarrow{e}_i \cdot d_i \tag{1}$$

$$\overrightarrow{e}_i = \begin{pmatrix} \cos\varphi_i \\ \sin\varphi_i \end{pmatrix};\ \varphi_i = \left\{ \frac{\pi i}{4} | 0 \leq i \leq 7 \right\} \tag{2}$$

Dieser zeigt an in welcher Richtung sich am wenigsten Hindernisse befinden. Die Länge lässt Rückschlüsse auf die Dringlichkeit des Ausweichens zu. Besitzt er eine Länge von Null, so befinden sich keine Hindernisse in messbarer Reichweite.

4.2 Abweichung vom Standard-Regelverhalten

Die Nick-Achse ist orthogonal zur Roll-Achse. Durch diese Unabhängigkeit können beide Ebenen getrennt voneinander betrachtet werden. Der Flucht-Vektor $\overrightarrow{y}$ wird in seine Komponenten aufgeteilt. Zwei unabhängige, aber identisch parametrisierte Reglerinstanzen, wirken je auf der Roll- und Nick-Achse.

Der klassische Regelkreis ist so konzipiert, dass für den Messwert y ein Arbeitspunkt gemäß einer charakteristischen Einschwingkurve erreicht und dieser dort kontinuierlich gehalten wird. Der Arbeitspunkt kann durch das Referenzsignal r beeinflusst werden. In der hier dargestellten Anwendung kann der Regler allerdings nur aktiv werden, wenn eine Kollision zu erwarten ist. Der Zustand zwischen zwei Kollisionen kann als Hysterese angesehen werden. Hier können keine Abstände gemessen werden (die Distanz zu den Objekten ist größer als die maximale Messdistanz). Daraus resultierend kann auch die Fluggeschwindigkeit nicht bestimmt werden. Auf diesen Zustand wird später nochmals genauer eingegangen (siehe Abschnitt 4.3).

Das Gesamtsystem muss zwei Ziele verfolgen: Primär muss der Kontakt mit dem Hindernis vermieden werden. Ein nur in eine Richtung aussteuernder PD-Regler kann diese Aufgabe übernehmen. Er wird aktiv, sobald die ihm zugehörige Komponente des Flucht-Vektors von Null verschieden ist. Der Quadrokopter neigt sich vom Hindernis weg bis die Regeldifferenz wieder Null wird. Während dieser Zeit wird die Plattform nicht nur abgebremst, sondern nimmt auch Fahrt in die entgegengesetzte Richtung zum Hindernis auf. Das System baut so lange Fahrt auf bis das Hindernis aus dem Detektionsbereich der Sensoren verschwunden ist oder die Mitte zwischen zwei Hindernissen erreicht wurde (die Flucht-Vektor Komponente wird wieder Null). Gerade im ersten Fall driftet der Flugroboter trotz einer stabilen geraden Lage mit hoher Geschwindigkeit. Die sekundäre Aufgabe des Systems muss es nun sein, diese nicht messbare Geschwindigkeit abzubauen. Hierfür verantwortlich ist die Gegensteuerung, auf die im Folgenden genauer eingegangen wird.

4.3 Gegensteuerung

Nach der erfolgreichen Kollisionsvermeidung befindet sich der Flugroboter meist in einem nicht messbaren Zustand. Da die zuvor aufgenommene Geschwindigkeit nicht mehr festgestellt werden kann, ist es möglich, dass der Roboter mit einer hohen Geschwindigkeit in ein anderes Objekt fliegt, wenn keine Gegenmaßnahmen eingeleitet werden.

Ein Beobachter kann die aufgenommene Geschwindigkeit schätzen und nach dem Ausregeln den Roboter anhand dieser Zustandsschätzung anschließend wieder abbremsen. Dazu wird, wie in Abbildung 2 veranschaulicht, die Fläche unter dem Antikollisionsregler Ausgang u berechnet und es wird um die gleiche Fläche linear wachsend gegengesteuert, sobald der Regelwert wieder zu Null wird. Kommt es in diesem Zeitraum zu einer weiteren Kollisionsmöglichkeit, wird der Wert der Gegensteuerung verworfen bzw. neu gestartet. Die Gegensteuerung ist somit nur nach den Kollisionsszenarien und während der Blindflugphase aktiv.

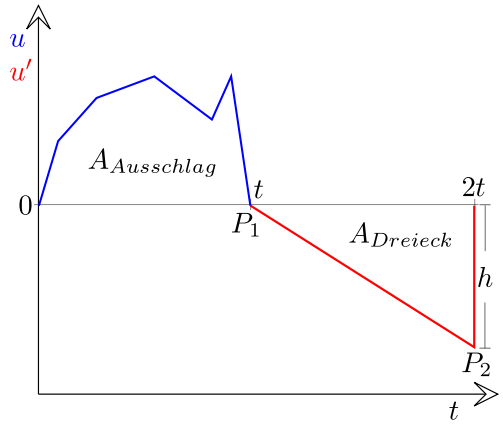

Abb. 2. Skizzierte Gegensteuerung

Die Berechnung der Gegensteuerung geschieht über die Fläche und über den gleichen Zeitraum des eigentlichen Ausweichvorgangs:

$$\frac{1}{2}t \cdot h = A_{Dreieck} = A_{Ausschlag} = \int_0^t u(x)\,dx \tag{3}$$

Mit den erwünschten Punkten $P_1(t, 0)$, $P_2(2t, h)$ der Gegensteuerung ergibt sich die einfache lineare Funktion für den Bremsvorgang:

$$u'(x) = -\frac{h}{t} \cdot x \tag{4}$$

Diese kann additiv dem eigentlichen Reglerausgang hinzugefügt werden. Eine linear steigende Kurve wurde gewählt, da der Beobachter rein über die Stellgröße das System beobachtet und nicht über den tatsächlichen Ist-Zustand. Dies wäre zwar möglich, jedoch bedarf es hierfür wieder eines erhöhten Kommunikationsaufwands mit dem Quadrokopter. Die Stellgröße kann im Gegensatz zu Hardware beliebig schnell die Neigungswerte ändern. Um den Beobachtungsfehler minimal zu halten und Energie zu sparen, wird nun im Ausgleichszustand versucht, möglichst kontinuierlich gegenzuwirken.

4.4 Steuerwert-Einkopplung

Bis jetzt bietet das geschlossene System keine Möglichkeit aktiv in den Flug einzugreifen. Eine Steuerung wie üblicherweise über den Referenzwert ist nicht machbar. Zum Einen gibt es im normalen Flug keine Rückmeldung der Sensoren, was ein Übersteuern des Reglers nach sich ziehen würde. Zum Anderen müsste der Beobachter übergangen werden. Deshalb ist eine Einspeisung der Steuerdaten $\overrightarrow{s}$ direkt vor der Strecke (dem Quadrokopter) sinnvoll.

Solange keine Hindernisse erkannt werden (Flucht-Vektor $\overrightarrow{y} = 0$) liegt der Regler brach und eine reine Steuerung ist aktiv. Kontraproduktiv ist es hingegen,

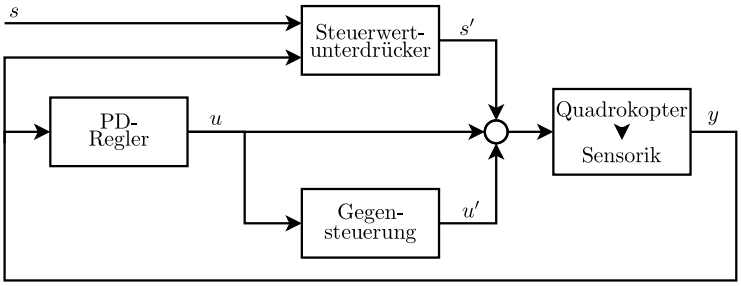

Abb. 3. Antikollisions-Regelkreis

wenn ein Hindernis entdeckt wird und der Operator, sei es ein Mensch oder ein Algorithmus, steuert weiterhin unvermindert auf des Hindernis zu. Sollte dieser Fall eintreten muss die Steuerung ausgeschaltet bzw. unterdrückt werden und die Regelung muss komplett die Kontrolle übernehmen.

Aus diesem Grund werden die Steuerdaten nicht direkt in des System gegeben sondern indirekt über einen Steuerwertunterdrücker. Dieser dämpft die Steuerwerte proportional zur Länge des Flucht-Vektors in negativer Richtung ab. Sollte der Operator auf ein Hindernis aktiv zusteuern, werden die Steuerwerte linear bis auf 0 % gedämpft, so das keine zusätzliche Kollisionsgefahr entsteht. Steuert der Operator vom Hindernis weg (in Richtung des Flucht-Vektors) findet keine Dämpfung statt.

4.5 Regelkreismodell

Abbildung 3 zeigt den Informationsfluss aller vorgestellter Komponenten. Die Steuerwerte gelangen unmodifiziert zum Quadrokopter, wenn der Flucht-Vektor gleich Null ist . Sollte der Flucht-Vektor einen von Null verschiedenen Wert aufweisen, wird die Steuerung immer mehr unterdrückt und die Regelung gewinnt die Oberhand.

Durch die Mischung von Steuerung und Regelung und in Kombination mit der Bildung des Flucht-Vektors, eignet sich das Gesamtsystem für den Flug innerhalb von Gebäuden. Es fliegt immer mittig durch Türen oder durch Gänge und minimiert so das Risiko des Kontakts mit Hindernissen.

5 Java in Echtzeitsystemen

Fliegende Roboter sind Echtzeit Systeme, die üblicherweise in C programmiert werden. Im Gegensatz dazu sind in Java Speicherverwaltung und Fehlerbehandlung komfortabler, jedoch fehlt es dieser Sprache im Vergleich zu C an Codeoptimierung und der direkteren Kontrolle der Abläufe. Gerade das Einplanen der Threads, das von der Java VM übernommen wird, ist als Programmierer schwer zu überprüfen und stellt somit eine schwer zu überwindende Hürde dar. Hinzu kommt die Laufzeit des Garbage Collectors.

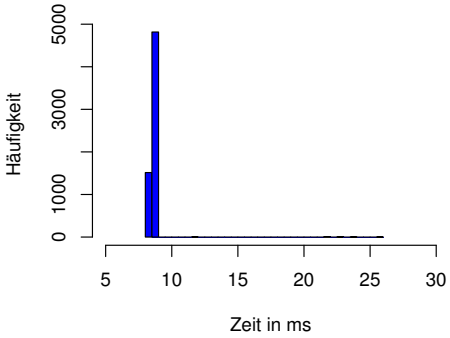

Abb. 4. Reglerlaufzeit ohne Last

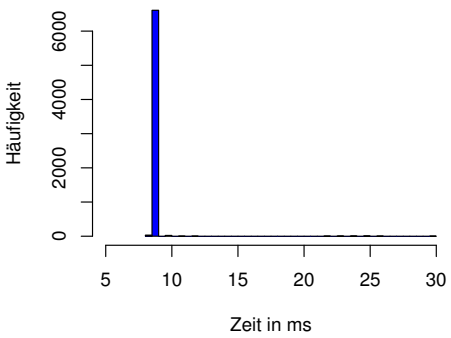

Abb. 5. Reglerlaufzeit mit Last

Diese und andere Gründe verwehren es normalerweise Anwendungen wie diese in Java zu implementieren. Hier wird allerdings ausgenutzt, dass der Quadrokopter bereits mit durch einen dedizierten Mikrocrontroller stabilisiert wird. Also ist die harte Echtzeit Anwendung bereits ausgelagert.

5.1 Laufzeiten der Steuerung

Die ganze Antikollisionsregelung inklusive der Sensorwert Beschaffung läuft in einem Java TimerTask mit höchster Priorität. Damit er alle 30 ms ausgeführt werden kann, muss ihm ein Hardware Timer zugeteilt werden. Eine signifikant höhere Frequenz ist wegen der Hardware-Kommunikation und der sehr langsamen Abtastfrequenz der Sensoren (38.3 ms $\pm$ 9.6 ms (Messfrequenz) + 5 ms (Signal einpendeln)) nicht sinnvoll. Wie aus der Dokumentation des SunSpot Timers [7] hervorgeht, kann Java keine festen zeitlichen Grenzen einhalten. Der in der Dokumentation erwähnte Jitter von 1 ms bis 3 ms tritt zwar auf, stellt aber nicht das Hauptproblem dar.

Abbildung 4 zeigt einen Histogrammplot der Laufzeiten des Timer Tasks. Tabelle 1 zeigt die dazugehörige Statistik. Als Zeitreferenz wurde die Systemzeit verwendet. Bei diesem Test wurde in der `main()` Schleife nur ein `sleep()` Aufruf gemacht und keine weiteren Threads erzeugt um sicherzustellen, dass die Java VM nur diesen einen Thread verwaltet. Was auffällt ist, dass für die meisten Zyklen eine CPU Zeit von 8 ms bis 9 ms benötigt wird. Jedoch wurden bei einigen Zyklen auf bis zu 24 ms ausgedehnt. Dies ist die Zeit, die der Garbage Collector für einen vollen Durchlauf benötigt. Laut Dokumentation kann dieser 20 ms bis 30 ms lang sein.

Um zu sehen, wie sich der Scheduler unter Last verhält, wurde die `main()` Schleife der Rechnereinheit mit zeitintensiven Berechnungen gefüllt. Die Ergebnisse sind in Abbildung 5 und in Tabelle 1 zu sehen. Man erkennt, dass die mittlere Laufzeiten des TimerTasks etwa um 0.2 ms nach oben geht. Die durch die Garbage Collector unterbrochenen Durchläufe sind aber auch hier das Hauptproblem. Durch die Thread-Priorisierung ist dessen Auftreten zwar minimiert,

	Minimum	1. Quantil	Median	Mittelwert	3. Quantil	Maximum
ohne Last	8.0	9.0	9.0	8.8	9.0	26.0
mit Last	8.0	9.0	9.0	9.0	9.0	30.0

Tabelle 1. Laufzeit Statistik

jedoch nicht völlig verhindert. Der Wechsel auf ein System, bei dem man die zeitlichen Abläufe, insbesondere den Scheduler, genauer kontrollieren kann, würde das System bezüglich der Reaktion auf Kollisionsgefahren jedoch nicht wesentlich verbessern.

5.2 Einplanungsungenauigkeit

Aus den Laufzeiten, den Anzahlen an durchlaufenen Zyklen und den Gesamtlaufzeiten der Tests lässt sich die mittlere Einplanungsgenauigkeit des Timer Tasks bestimmen. In Tabelle 2 sieht man, dass der Timer unabhängig von der Last etwa alle 33 ms den Task startet, obgleich der Regler alle 30 ms aktiv werden sollte. Diese Verzögerung entspricht dem maximalen angegebenen Jitter von 3 ms und lässt darauf rückschließen, dass der Jitter nicht uniform verteilt ist.

Möchte man nun mit Sicherheit sagen, dass der Timer Task zeitliche Grenzen der Ausführung einhält, so muss man die maximale Garbage Collector Durchlaufzeit, den maximalen Jitter sowie die Zeit des unverdrängten Antikollisionsreglers berücksichtigen. Es ergibt sich eine obere Schranke für die Dauer des Threads von etwa 42 ms. Dies gilt jedoch nur als absolute obere Schranke. In über 99.9 % der Fälle kann eine um eine Ordnung kleinere Schranke angenommen werden. Der Jitter kann zudem als bekannte Konstante korrigiert werden. Experimente haben gezeigt das dieses Kriterium völlig ausreichend ist, da durch die Massenträgheit des Objekts eine sehr seltene 30 ms dauernde Verzögerung das Gesamtsystem nicht beeinflusst. Wird allerdings die Priorität des Threads herabgesetzt oder gleichpriorisierte Threads verbrauchen viel CPU-Zeit, so steigt die Wahrscheinlichkeit, dass innerhalb des Regelungsthreads der Garbage Collector aktiv wird signifikant an. Dies hat zur Folge, dass das System nicht mehr flugfähig ist.

	Gesamtlaufzeit	Zyklen	relative Startzeit
ohne Last	210 000 ms	6350	33.0 ms
mit Last	220 000 ms	6684	32.9 ms

Tabelle 2. Durchschnittliche relative Startzeit

6 Fazit und Ausblick

Wir haben eine Möglichkeit aufgezeigt, wie handelsübliche Quadrokopter mit minimalem Aufwand zu fliegenden Robotern für die Verwendung innerhalb von Gebäuden modifiziert werden können. Durch die Verwendung analoger Sensoren und durch die Verwendung von einfachen Algorithmen ist es möglich, die komplette Regelung auf einem sehr leistungsschwachen Prozessor zu implementieren. Auch wurde mittels Java und dessen Garbage Collectors gezeigt, dass selbst für fliegende Roboter unter gewissen Bedingungen keine zwingende harte Echtzeitfähigkeit des Systems nötig ist. Jedoch können wir die Resultate von Chen *et al.* [3] bestätigen: Der SunSpot und die auf ihm laufende Version von Java sind, zumindest im momentanen Entwicklungsstand (Version 5.0), nicht für akkurate Echtzeitanwendungen geeignet. Ohne den dedizierten Mikrocontroller für die Stabilisierung der Plattform wäre der Quardokopter nicht flugfähig.

Zwischen dem Aufzeichnen einer Abstandsmessung und der Übergabe der entsprechenden Steuerbefehle an den Quadrokopter können, begründet in der Messtechnik, der Kommunikation und der Berechnung, bis zu 85 ms vergehen. Daher ergibt sich eine virtuelle „Knautschzone“ von 50 cm um das Objekt. Dies beschränkt die maximale Geschwindigkeit für ein erfolgreiches Ausweichen auf zirka $3\,\mathrm{km\,h^{-1}}$. Diese Geschwindigkeit ist angesichts einer maximalen Endgeschwindigkeit von $80\,\mathrm{km\,h^{-1}}$ sehr schnell erreicht. Aus diesem Grund beschäftigen wir uns momentan mit dem Bau eines leichten und kostengünstigen Laserscanners mit 360°-Scanfähigkeit, der einen sehr feingranularen Fluchtvektor mit größerer Messdistanz und geringerer Verzögerung bietet.

Literaturverzeichnis

1. Grzonka, S., Grisetti, G., und Burgard, W., „Towards a navigation system for autono mous indoor flying,“ in IEEE International Conference on Robotics and Automation (ICRA 2009), Kobe, Japan, Mai 2009, S. 2878-2883.
2. He, R., Prentice, S., und Roy, N., „Planning in information space for a quadrotor helicopter in a GPS-denied environment,“ in IEEE International Confonference on Robotics and Automation (ICRA 2008), Pasadena, CA, Mai 2008, S. 1814-1820.
3. Chen, L., McKerrow, P. J., und Lu, Q., „Develop Real-time Application with Java Based Sun SPOT,“ in Australasian Conference on Robotics and Automation, Canberra, Australia, Dezember 2008.
4. Sharp, „GP2Y0A02YK Long Distance Measuring Sensor,“ Datenblatt. Online: http://www.datasheetarchive.com/pdf-datasheets/Datasheets-31/DSA-612316.html
5. Smith, R., „SPOTWorld and the Sun SPOT,“ in 6th International Conference on Information Processing in Sensor Networks (IPSN 2007), Cambridge, MA, April 2007, S. 565-566.
6. Horn, M., und Dourdoumas, N., „Regelungstechnik: Rechnerunterstützter Entwurf zeitkontinuierlicher und zeitdiskreter Regelkreise,“ 1. Ausgabe. Pearson Studium, 2004.
7. Sun Microsystems, „Using the AT91 Timer/Counter,“ Datenblatt. Online: http://www.sunspotworld.com/docs/AppNotes/TimerCounterAppNote.pdf