Gruppe: Jakob Duebel, Jonas Luft, Andreas Gerken, Tanja Kaiser

RoboNewbie Übung

Omnidirektionales Laufen

Der Roboter kann sich omnidirektional, also in x-, y- und alpha-Richtung, fortbewegen. Dies wurde durch die Implementierung der Parallelkinematik auf Basis einer Positionskontrolle erreicht.
Um die Stabilität des Roboters zu erhöhen, geht der Roboter bildlich gesprochen in die Knie. Die Pitch-Gelenke von Hüfte, Knie und Fuß werden dazu auf einen festen Wert *offset* gesetzt.

Um das Laufen möglich zu machen, verschiebt der Roboter die Hüfte abwechselnd nach Links und Rechts (Hip und Foot Roll Gelenke) um seinen Schwerpunkt zu verlagern. Die Verschiebung geschieht auf Basis einer Sinusfunktion, wobei durch den Wert *shift* die maximale Verschiebung der Hüfte angegeben wird.
Das momentan nicht belastete Bein wird danach angehoben. Dies geschieht basierend auf einer cosinus-Funktion und durch die Verwendung der Pitch-Gelenke von Fuß, Knie und Hüfte. Wie hoch das Bein gehoben wird, ist durch die Variable *stepheight* steuerbar. Die so erreichte Position ist die Ausgangsposition für Bewegungen in alle Richtungen.

y-Richtung

Um eine Fortbewegung in y-Richtung zu bewirken, also vorwärts oder rückwärts zu laufen, wird das Pitch-Gelenk des Fußes und der Hüfte weiter bewegt als das Gelenk des Knies. Dadurch entsteht eine Verschiebung in y-Richtung. Diese Fortbewegung basiert auf einer Cosinus-Funktion, die für das linke und rechte Bein um eine halbe Phase verschoben ist. Die Weite des Schritts kann durch die Variable *steplength* variiert werden.

x-Richtung

Werden Hüfte und Fuß des Roboters verschoben (Roll-Gelenke), wenn der Fuß angehoben ist, wird eine Fortbewegung in x-Richtung erreicht. Dazu kommen für den linken und rechten Fuß um eine halbe Phase verschobene cosinus-Funktionen zum Einsatz. Die Schrittweite kann mit *step_sidewards* festgelegt werden.

alpha-Richtung

Eine Drehung des Roboters um sich selbst kann durch die Verwendung des YawPitch-Gelenks der Hüfte erreicht werden. Dabei wird je nach Drehrichtung entweder das linke oder rechte Bein im Winkel *alpha* zur Seite gestreckt und wieder abgesetzt. Damit der Roboter nicht umkippt, muss ein Ausgleich um den gleichen Winkel am Pitch-Gelenk der Hüfte erfolgen. Auch in diesem Fall erfolgt Ansteuerung über eine phasenverschobene cosinus-Funktion.

Stabilisierung

Als Ausgangspose lässt der Roboter die Arme hängen und winkelt sie etwa 30° vom Körper ab. Zur Stabilisierung des Laufes werden nur die Arme bewegt. Droht der Roboter nach hinten zu kippen, werden beide Arme nach vorn gestreckt. Umgekehrt werden die Arme nach hinten bewegt, wenn er nach vorn kippt. Zur Stabilisierung in y-Achse, werden analog beide gestreckte Arme entgegen der Kippbewegung bewegt.
Accelero- und Gyrometer fließen zu gleichen Teilen in die Berechnung der Körperlage ein. Die Werte werden geglättet und auf die voreingestellte Winkel-Position addiert.

Evolution

Sowohl für das orthogonale Laufen wie auch für die Stabilisierung werden Variablen eingeführt, die so eingestellt werden sollen, dass eine stabile aber auch möglichst schnelle Gangart entsteht. Die Variablen sind vermutlich in einer unbekannten Weise voneinander abhängig und es ist schwer die Variablen “per Hand” durch probieren einzustellen. Um den Vorgang zu automatisieren wurde ein Algorithmus implementiert der Agenten mit Variablen einzustellen, der die biologische Evolution nachahmt. Das Prinzip dieser Algorithmen ist es durch zufällige Mutation der Variablen neue Agenten zu erzeugen und zu überprüfen, wie “gut” sie sind. Die besten Individuen überleben und können weiter mutiert werden.

Künstliche evolutionäre Algorithmen benötigen einige Features:

- **Fitness Funktion**: Die Güte eines jeden Individuum muss gemessen werden können.
- **Mutation:** Die Variablen müssen zufällig mutiert werden können. Wie sehr sie verändert werden wird über eine Schrittweite angegeben. Es ist sehr wichtig, dass sich die Fitness der Agenten auf diese Mutation stark kausal verhalten. Das heißt, dass eine kleine Änderung der Variablen auch eine kleine Veränderung im Fitness Wert ergeben sollte.
- **Selektion:** Die Agenten einer Population müssen nach dem Fitnesstest selektiert werden. Das heißt dass einige Agenten “sterben” und andere überleben.
- **Mehrfache Messung (optional):** Je nach Umwelt kann das Ergebnis des Fitnesstests bei mehrfacher Durchführung schwanken. Deshalb kann der Test mehrfach durchgeführt werden und der Durchschnitt oder Median der Werte verwendet werden.

Um die Features zu implementieren wurden zwei Klassen erstellt. Eine, die die Population verwaltet und den eigentlichen Algorithmus anwendet und eine, die einen evolutionären Agenten vorbereitet. Von dieser Klasse kann geerbt werden um einen beliebigen Agenten einzustellen.
Bei der Implementierung gibt es einige hervor zu hebende Details:

- **Fitness Funktion:** Da der rcs-server von einer lokalen Sicht der Roboter ausgeht, muss diese zunächst verwendet werden um eine globale Position zu finden. Die Implementierung verwendet die Entfernung zu den Torpfosten. Instabile Individuen sollen bestraft werden. Das funktioniert durch das Auslesen des Gyroskops.

- **Mutation:** Die Mutationen werden in der Implementierung über eine Gaussverteilung mit einer bestimmten Standardabweichung erstellt. Die Standardabweichung ist pro Variable in jedem Individuum gespeichert und wird vor jeder Mutation entweder mit μ^1 oder μ ^(-1) multipliziert. An dieser Stelle gibt es möglicherweise noch Verbesserungsbedarf, da in manchen Fällen die Schrittweite sich selbst extrem hoch eingestellt hat.
- **Mehrfache Messung:** Damit die Laufzeit nicht explodiert wurden alle Individuen zunächst nur einfach getestet. Danach wurde ein Bruchteil (40%) selektiert, der mehrfach getestet wurde. Mit den genaueren Werten wurde dann so selektiert, dass noch ~27% der Anfangspopulation als neue Gründerpopulation weiterbesteht.
- **Servergeschwindigkeit:** Der Server läuft normalerweise in Echtzeit, was bedeutet, dass jeder Versuch etwa 10 Sekunden dauert. Bei 50 Individuen (40% Dreifachtest) in jeder Generation dauert jede Generation etwa 15 Minuten was zu langsamen Ergebnissen führt. Deshalb wurde der Server so eingestellt, dass er sich nicht mehr an die Echtzeit hält sondern sich mit dem Agenten synchronisiert.

Probleme: Der Algorithmus produziert meistens anfangs verbesserte Agenten, fängt jedoch auf einmal an, ganze Generationen von instabilen Agenten zu entwickeln. Diese werden dann für die Mutation verwendet und es entstehen keine Stabilen Agenten mehr. Möglicherweise kommt das daher, dass die Umgebung der Agenten nicht genug stark kausal ist. Eine andere Erklärung wäre die mutative Schrittweiten einstellung. Eine intelligentere Vorgehensweise zum Beispiel mit einer geschachtelten Strategie mit Fortschrittsgeschwindigkeitsmessung wäre möglicherweise von Vorteil.