Technische Universität Graz

Institut für Grundlagen der Informationsverarbeitung

Bachelorarbeit

Ein Vergleich von Lernalgorithmen für Parametersuche im hochdimensionalen Raum

Bearbeiter:	Thomas K. Wiesner
Aufgabensteller:	DI Elmar Rückert, DI Gerhard Neumann
Betreuer:	Mag. Dr. Stefan Häusler
Abgabedatum:	18.April 2011

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Graz, den 18.April 2011

. . . . . .

(Unterschrift Thomas Wiesner)

Kurzfassung

Viele Probleme in der Robotik, wie zum Beispiel das Generieren von Bewegungen, die Navigation durch ein Gebäude oder das logische Planen von Aktionen können als Parametersuchprobleme formuliert werden. Dazu definiert man für diese Aktionen eine Reihe von Parametern, wobei die Suche nach einem guten Parameterset zur Aufgabe von Suchalgorithmen gemacht werden kann. Dies ist eine komplexe Aufgabe und entspricht typischerweise einer Suche im hochdimensionalen Raum (z.B.: 10 Parameter $\widehat{=}$ 10^{10} dimensionalen Raum). Derzeit gibt es verschiedene Algorithmen um Parameter im hochdimensionalen Raum zu suchen und zu optimieren.
Die hier vorliegende Arbeit soll einen Vergleich dreier Algorithmen, CMA-ES, Particle Swarm Optimization sowie SIMPLEX geben. Ziel dieser Arbeit ist es vorhandene Algorithmen in Hinsicht auf Lerngeschwindigkeit, Qualität der Lösung und Anwendbarkeit zu prüfen. Die Algorithmen sollen Parameter finden, sodass ein simuliertes Modell eines Schwimmers möglichst schnell von einem Punkt zu einem anderen schwimmt. Wir werden hier zuerst näher auf die Aufgabenstellung eingehen und werden zeigen mit welchen Problemen man konfrontiert ist, wenn die Suche sich über einen hochdimensionalen Raum erstreckt. Weiters wird dann verdeutlicht, dass es mithilfe der Algorithmen verschieden-gut und verschieden-schnell möglich ist, solche Parameter zu finden.

1 Einleitung

1.1 Problembeschreibung

Es gibt verschiedene Ansätze zur Suche optimaler Parameter. Während die Suche nach einer guten, wenn nicht sogar optimalen Lösung bei einigen wenigen Parametern oft recht trivial gelöst werden kann, ist die Suche nach optimalen Parametern im hochdimensionalen Raum viel anspruchsvoller. Das spezielle Problem dieser Aufgabenstellung war die Suche nach 10 Parametern ($\widehat{=} 10^{10}$ dimensionalen Suchraum) für die Simulation eines Schwimmers, welcher wiederrum eine Performance zurückliefert, die angibt wie effizient geschwommen wurde. Dabei kann der Parameterraum nicht beliebig ausgenutzt werden, sondern ist, wie ein realer Schwimmer, nach oben und unten begrenzt. Beispielsweise kann ein Gelenk keine beliebig große Frequenz erreichen.
Wie Abbildung 1.1 zeigt, muss sich der Schwimmer möglichst weit in einer vorgegebenen Zeit entlang des schwarzen Pfeils in x-Richtung bewegen um eine möglichst große Performance zu erreichen. Wie er das macht - also ob vorwärts, rückwärts, in Kreisbewegungen, ... - ist vollkommen egal, hautsache er überschreitet nicht maximalen Parametergrenzen. Die optimalen Parameter zu finden, die den Bewegungsablauf so gestalten, das die Performance maximal wird, ist das Problem, welches durch die Algorithmen gelöst werden soll.

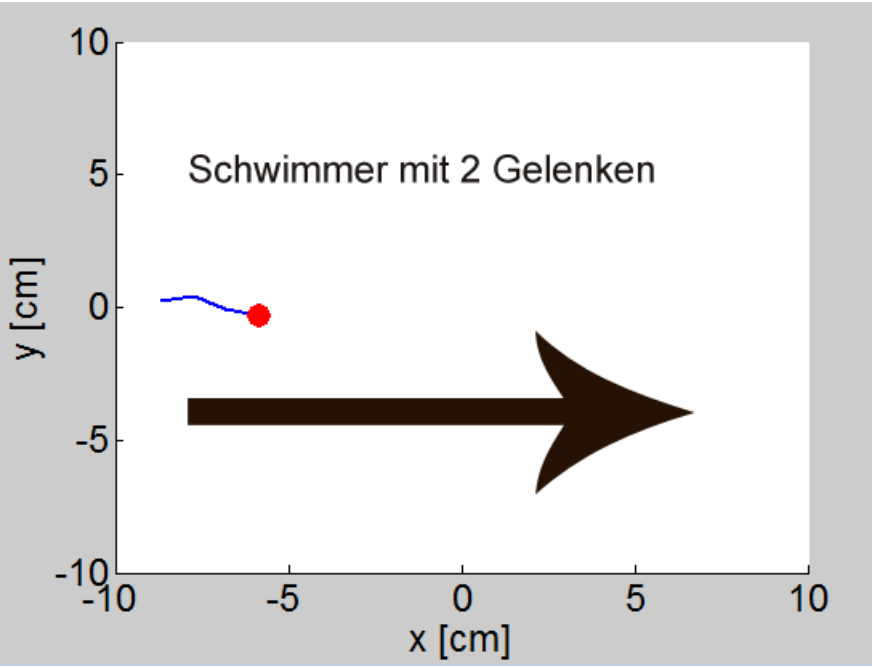

Abbildung 1.1: Schwimmer mit 2 Gelenken muss möglichst weit in x-Richtung schwimmen

1.2 Lernframework

Das Lernframework besteht aus einem Policylearner welcher die gelernte Policy an einem Schwimmer zur Anwendung bringt und durch die zurückgelieferten Kosten bzw. Performance sieht, wie gut die Policy funktioniert. Der Schwimmer selbst besteht aus dem Generieren der grundlegenden dynamischen Bewegungen (*Dynamic Movement Primitives - DMPs*), dem linearen PD Controller und dem Berechnen der für den Schwimmsimulationsvorgang notwendigen Kosten (bzw. die Performance= -Kosten). Das gesamte Framework ist schematisch in Abbildung 1.2 dargestellt. Der Policylearner, welcher in diesem Fall entweder der CMA-ES, Particla Swarm Optimizer oder SIMPLEX ist, lernt die Parameter für die Berechnung der DMPs: $\Theta_{t+1} = f(\Theta_t, hist)$. Der lineare PD Controller, welcher gebraucht wird um der Trajektorie zu folgen, die von den DMPs generiert worden ist, ist wie folgt aufgebaut: $u = k_{pos}(\vec{y}(t) - \vec{q}(t)) + k_{vel}(\dot{\vec{y}}(t) - \dot{\vec{q}}(t))$. Die Parameter k_{pos} und k_{vel} wurden dabei mit $k_{pos} = 2$ und $k_{vel} = 0.3$ empirisch ermittelt, könnten aber natürlich auch durch den Policylearner gelernt werden.

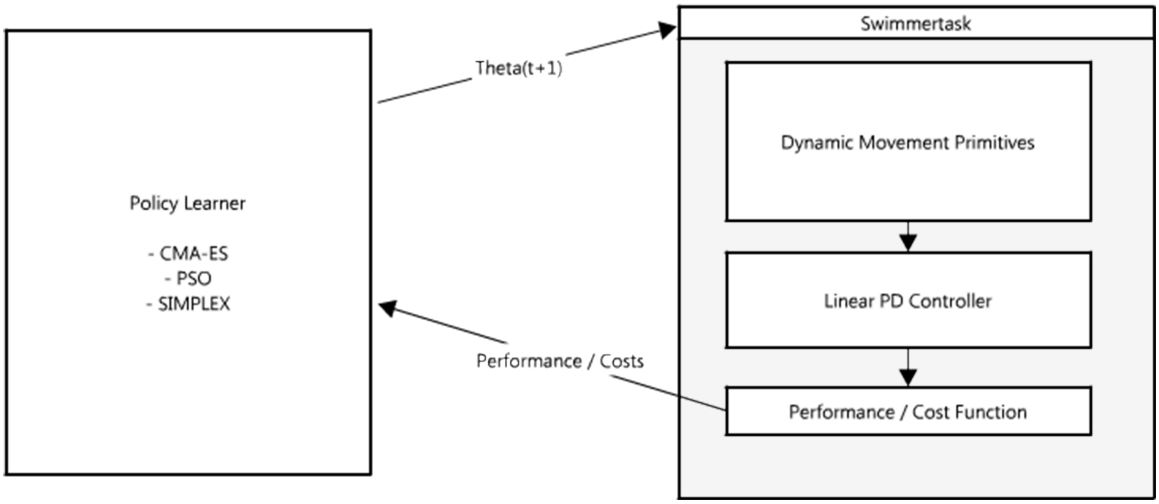

Abbildung 1.2: Schematische Darstellung des Lernablaufs

1.3 Schwimmerparametrisierung

Der Schwimmer besteht aus drei Verbindungsstücken welche durch zwei Gelenke zusammengehalten werden, wobei per Definition beim roten Punkt „vorne“ ist. Eine Vielzahl von Parametern bestimmen das Verhalten des Schwimmers. Durch Zuweisung beispielsweise der Gewichte für die Korellation der Oszillatoren, des Amplitudenanstiegs, der Frequenz, der Winkeländerung, kann eine Schwimmbewegung modelliert werden. Einfach gesagt könnte man sich das Schwimmverhalten in etwa so vorstellen, als würde man einer Kaulquappe von oben beim Schwimmen zusehen.
Zum Modellieren werden *Dynamic movement primitives (DMPs)* verwendet die eine Reihe von Differentialgleichungen beschreiben (Schaal u. a. (2005)). Es stehen insgesamt 18 Parameter zur Verfügung, wobei einige vorweg mit Konstanten befüllt werden, sodass der Lernal-

gorithmus ein Set von 10 Parametern suchen muss. Diese 10 Parameter sind beschränkt (siehe Policy (1.5)). Beispielsweise gibt es die Phasenverschiebung von -180° bis +180°, statt von -10 bis +10. Am Ende sollen die beiden Gelenke so mit einer bestimmten Frequenz und Phasenverschiebung oszillieren (siehe Abbildung 1.3), dass eine Schwimmbewegung ausgeführt wird, die die Kostenfunktion minimiert, bzw die Performancefunktion maximiert. Für die Generierung der Bewegung der Oszillatoren werden folgende Gleichungen aufgestellt (Pouya u. a. (2010)):

$$\dot{\phi} = 2\pi \cdot \omega_i + K_i \tag{1.1}$$

$$K_i = \sum_j w_{ij} \cdot r_j \cdot \sin(\phi_j - \phi_i - \psi_{ij}) \tag{1.2}$$

$$\dot{r} = a_i(R_i - r_i) \tag{1.3}$$

$$\theta_i = r_i \cdot \sin(\phi_i) + X_i \tag{1.4}$$

Pouya u. a. (2010) beschreibt noch weitere Gleichungen für Bewegungsabläufe, jedoch sind für dieses Experiment nur oszillierende Bewegungen von Nöten. Die Variablen r_i und ϕ_i sind Zustandsvariablen und sind für Amplitude und Phase der Oszillation verantwortlich. Die Parameter w_{ij} und ψ_{ij} sind die Gewichte bzw. die Phasenverschiebung zwischen den Oszillatoren i und j. a_i ist eine positive Konstante, die die Anstiegszeit der Amplitude bis zum Wert R_i festlegt. Die Parameter R_i, X_i und ψ_{ij} sind offene Parameter. Die gezeigte Struktur in Gleichungen (1.1) und (1.3) wäre auch fähig Sensorenfeedback zu verarbeiten, dies wird aber nicht benötigt. Schließlich beschreibt Gl. (1.4) das endgültige Oszillationsverhalten.

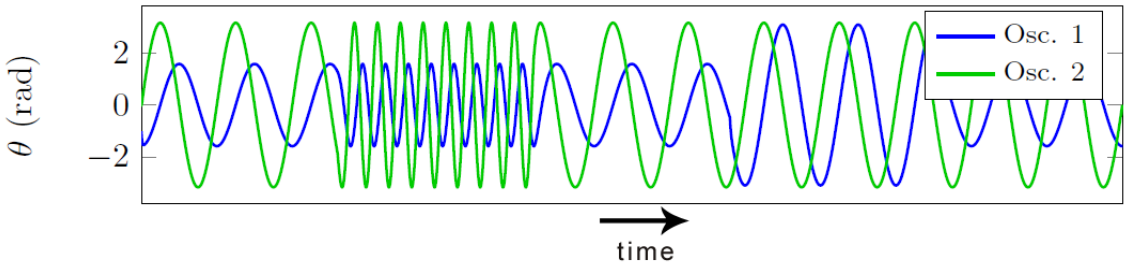

Abbildung 1.3: Synchronisationverhalten von zwei gekoppelten Oszillatoren (Pouya u. a. (2010))

Die Schwimmerbewegung wird mit zwei Oszillatoren modelliert, wobei die Policy für die Parametersuche wie folgt aufgebaut ist:

$$\Theta = [Gewichte \quad Kreisfrequenz \quad Phasenverschiebung \quad Amplitude]$$

$$\Theta = [w_1 \quad w_2 \quad w_3 \quad w_4 \quad \omega_2 \quad \omega_1 \quad \phi_1 \quad \phi_2 \quad R_1 \quad R_2] \tag{1.5}$$

Diese Parameter sind folgendermaßen beschränkt:

$$\Theta_{min} = [-10 \quad -10 \quad -10 \quad -10 \quad -\pi \quad -\pi \quad -\pi \quad -\pi \quad -10 \quad -10]$$

$$\Theta_{max} = [+10 \quad +10 \quad +10 \quad +10 \quad +\pi \quad +\pi \quad +\pi \quad +\pi \quad +10 \quad +10]$$

Die restlichen Parameter wurden manuell festgelegt:

$\psi_{11} = 3.7089$, $\psi_{12} = 5.8108$, $\psi_{21} = 4.4158$, $\psi_{22} = 3.3912$

$a_1 = 2.2955$, $a_2 = 2.2906$

$X_1 = X_2 = 0$

1.4 Gewählte Methoden

Es werden drei verschiedene, robuste Algorithmen verglichen, welche sich für die Suche im hochdimensionalen Raum anbieten. CMA-ES, welcher ein sehr populärer Algorithmus ist, Particle Swarm Optimizer, welcher leicht zu implementieren ist, in Pouya u. a. (2010) als Optimierungsalgorithmus bereits zum Einsatz kommt und unter gewissen Umständen sehr gute Ergebnisse liefern kann und SIMPLEX, welcher in einer bereits vorgefertigten Matlab-Funktion vorliegt und sich durch seine Einfachheit anbietet. Die einzelnen Algorithmen werden im Folgenden näher beschrieben.

2 Methoden

2.1 Allgemeines

Im Allgemeinen kann eine Parametersuche folgermaßen beschrieben werden: Es gibt eine Performancefunktion r und eine Policy Θ, die direkt miteinander zusammenhängen. Die Performancefunktion r liefert die aktuelle Performance zu der jeweiligen Policy. Eine neue Policy Θ_{i+1} hängt von allen bereits evaluierten Policies ab $\Theta_{0...i}$ und von allen Performanceergbnissen aus diesen ($r_{0...i}$).

2.2 CMA-ES

Wie in Hansen (2006) beschrieben, wird durch diese Methode eine gewisse Funktion minimiert, wobei die einzige Information dieser Funktion die zugänglich ist, die Suchwerte sind mit der die Funktion aufgerufen wird. *Covariance Matrix Adaption - Evolution Strategie (CMA-ES)* gehört in die Klasse der *Estimation of Distribution Algorithms (EDAs)* - Algorithmen die versuchen eine Verteilung der Parameter zu bestimmen. CMA-ES verwendet Normalverteilungen.
Jede Normalverteilung, wie in Hansen (2006) beschrieben, $\mathcal{N}(m, C)$, ist eindeutig durch ihren Mittelwert $m \in \mathbb{R}^n$ und ihre symmetrische und positiv endliche Kovarianzmatrix $C \in \mathbb{R}^{nXn}$ bestimmt. Kovarianzmatrizen erlauben eine ansprechende geometrische Interpretation als (Hyper-) Ellipsoid, wie in Abbildung 2.1

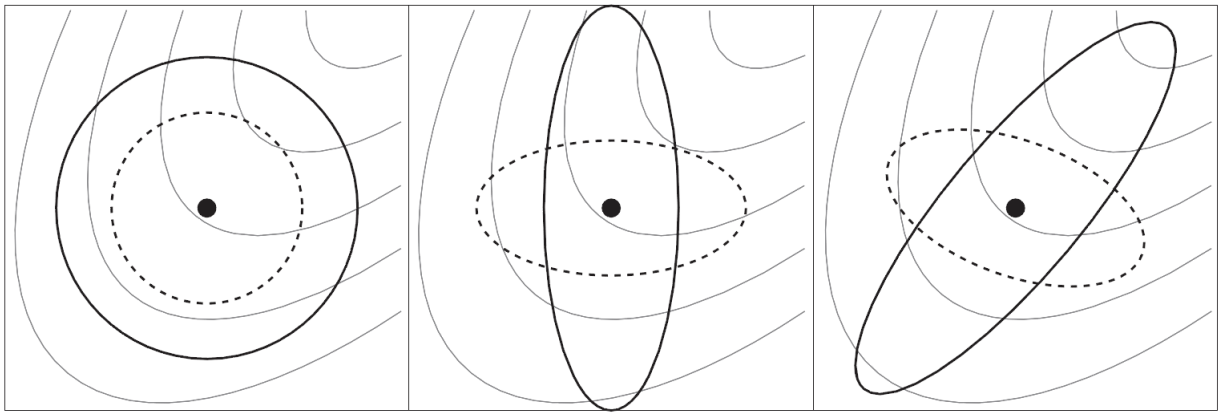

Abbildung 2.1: Hansen (2006): Sechs Ellipsen, Darstellung ein-σ Linien von gleicher Dichte von sechs verschiedenen Normalverteilungen.

Das Ziel der Kovarianzanpassung (*Covariance Matrix Adaption*) ist es, die Suchverteilung

so den Konturlinien anzupassen, dass die Zielfunktion minimiert wird. Analog könnte man Abb 2.1 als Darstellung von Höhenlinien bezeichnen, wobei es die Aufgabe des Algorithmus ist den „Berg“ bis zum Gipfel zu besteigen, obwohl man diesen nicht sieht.

Der CMA-ES hat weiters eine Schrittweitenkontrolle (*Step Size Control*), die in den meisten EDAs für gewöhnlich fehlt. Wir werden eine Schrittweite von 0.1 verwenden.
Die Populationsrate (*population size*), λ, ist immer ein Kompromiss zwischen einer schnellen Konvergenz, bzw. das Verhindern vom auffinden des lokalen Optimums. Die Populationsrate wird in diesem Beispiel anhand des Matlab-Beispiels 2.1 zugewiesen.

```
1 n = size(action,2); %=10
2 lambda = round(4 + 3 * log(n)); %=11
```

Listing 2.1: Zuweisung der Populationsrate anhand der Parameteranzahl in learnCMAES.m, lambda = λ

Die Populationsgröße λ hat mehrere Auswirkungen: Es werden die standard-Parameter mit der der Algorithmus betrieben wird aufgrund von λ berechnet, außerdem werden neue Suchpunkte $x_k^{(g+1)} \sim \mathcal{N}(m^{(g)}, (\sigma^{(g)})^2 C^{(g)})$ für $k = 1, ..., \lambda$ berechnet. Es kommt dann zu einer Selektion und Rekombination, Schrittweitenkontrolle und schlussendlich wird die Kovarianzmatrix adaptiert.(Hansen (2006))

Externe Parameter von diesem Algorithmus sind λ, μ, $\omega_{i=1...\mu}$, c_σ, d_σ, c_c, μ_{COV} und c_{COV}. Diese werden allerdings durch standard-Parameter ersetzt, welche durch (2.1) bis (2.5) berechnet werden. Es wird von Hansen (2006) davon abgeraten, die standard-Parameter manuell einzustellen, da sich der Algorithmus mit den Parametern, so wie sie aufgrund der Populationsgröße λ berechnet werden, auf eine große Palette von Problemen anwenden lässt. Wenn man die Polulationsgröße λ manuell verändert, so hat dies natürlich gravierende Auswirkungen auf die anderen Parameter, da diese von λ weitgehend abhängen.

Selektion und Rekombination:

$$\lambda = 4 + \lfloor 3 \ln n \rfloor, \quad \mu = \lfloor \lambda/2 \rfloor \tag{2.1}$$

$$\omega_i = \frac{\ln(\mu + 1) - \ln i}{\sum_{j=1}^{\mu} (\ln(\mu + 1) - \ln j)}, \quad i = 1, ..., \mu \tag{2.2}$$

Schrittweitenkontrolle:

$$c_\sigma = \frac{\mu_{eff} + 2}{n + \mu_{eff} + 3}, \quad d_\sigma = 1 + 2 \max(0, \sqrt{\frac{\mu_{eff} - 1}{n + 1}}) + c_\sigma \tag{2.3}$$

Kovarianzmatrix Adaption:

$$c_c = \frac{4}{n + 4}, \quad \mu_{COV} = \mu_{eff} \tag{2.4}$$

$$c_{COV} = \frac{1}{\mu_{COV}} \frac{2}{(n + \sqrt{2})^2} + (1 - \frac{1}{\mu_{COV}}) \min(1, \frac{2\mu_{eff} - 1}{(n + 2)^2 + \mu_{eff}}) \tag{2.5}$$

Der CMA ist fähig auch für schlecht formulierte oder kaum trennbare Probleme gute Lösungen zu finden. Es ist ein hochwertiger state-of-the-art Algorithmus für kontinuierliche Daten, der dadurch, dass er quasi parameterfrei ist, auf eine große Palette von Problemen leicht anwendbar ist. (Hansen (2006))

Wichtige Eigenschaften:

- Nur 1 Parameter
- Verwendet volle Kovarianz
- Anzahl der Offsprings ist logarithmisch abhängig von der Anzahl der Parameter

2.3 Particle Swarm Optimization

Die verwendete Variante des PSO-Algorithmus (*Particle Swarm Optimization Algorithm*) ist eine relativ einfache. Grundsätzlich basiert sie auf Kennedy und Eberhart (1995), und beschreibt dass es eine Population gibt, die Schwarm (*Swarm*) genannt wird, von vielen möglichen Lösungen, die Partikel (*Particles*) genannt werden. PSO kommt auch bei Pouya u. a. (2010) zum Einsatz. Die Partikel werden im Suchraum herumgezogen, wobei die Methode relativ simpel ist: Der Partikel mit der besten Position - also mit dem besten Parameterset - behält seine Position und zieht die anderen Partikel in seine Richtung. Dies wird solange fortgesetzt bis entweder eine neue beste Position gefunden wird oder ein Abbruchkriterium erreicht ist. Eine optimale Lösung ist möglich, jedoch kann nicht einmal eine zufriedenstellende Lösung *garantiert* werden.

Es sei $f : \mathbb{R}^n \to \mathbb{R}$ eine Kostenfunktion die minimiert werden soll. Weiters sei S der Suchraum. f nimmt als Argument einen Vektor von Zahlen aus dem reelen Zahlenbereich auf und gibt eine reelle Zahl zurück, die die Kosten darstellt. Der Gradient von f ist nicht bekannt. Das Ziel der Optimierung ist eine Lösung $\mathbf{a}$ zu finden für die gilt: $f(\mathbf{a}) \leq f(\mathbf{b}) \forall \mathbf{b} \in S$.

Diese einfache Form des PSO-Algorithmus hat leider eine Reihe von Parametern die nicht online mitoptimiert werden, sondern vorher per Hand eingestellt werden müssen. Die Geschwindigkeit mit der die Partikel in eine Richtung gezogen werden, besteht aus der bereits ausgeführten Geschwindigkeit multipliziert mit einem Parameter ω sowie den Anteilen für die beste Position für den einen, gerade betrachteten Partikel und die beste gefundene Position (global bester Partikel). Formal kann die Geschwindigkeit zum Iterationsschritt i folgendermaßen geschrieben werden: $v_i \leftarrow \omega * v_i + \phi_p * r_p * (p_i - x_i) + \phi_g * r_g * (g - x_i)$, wobei r_p und r_g eine Zufallszahl zwischen 0 und 1 ist.
Für die Geschwindigkeit mit der andere Partikel in die Richtung des Partikels mit der besten Lösung gezogen werden, sind drei Parameter maßgeblich: ω, ϕ_p und ϕ_g.
Wählt man diese Parameter geschickt, kann man schnell zu einer Lösung kommen, jedoch ist dies in keinem Fall garantiert. Wählt man sie zu klein, könnte der Algorithmus einfach stehen bleiben. Wählt man sie zu groß, kann es passieren, dass er über das globale Minimum hinausschnellt und nur ein lokales Minimum liefert.
Initialisiert wird der Algorithmus mit Zufallszahlen zwischen den Parametergrenzen. Auch dies stellt keinesfalls sicher, dass überhaupt eine zufriedenstellende Lösung auch bei langer Laufzeit gefunden wird, macht es aber bei steigender Partikelanzahl durchaus plausibel.

Wichtige Eigenschaften:

- Einfach zu implementieren
- Einfach zu parallelisieren
- Parameter schwer zu tunen

2.4 SIMPLEX

Wir verwenden hier die sogenannte Nealder-Mead Methode, oder auch „downhill simplex“ Methode genannt. Sie ist eine oft verwendete Methode der nichtlinearen Optimierungstechnik. Zu Beginn wird ein Simplex aus N+1 Punkten aufgespannt, welcher den N-dimensionalen Parameterraum darstellt. Zu jedem Punkt wird ein Funktionswert berechnet. Der Algorithmus ermittelt dann den schlechtesten Punkt aufgrund der Kostenfunktion. Zum nächsten Iterationsschritt gelangt man, indem man den schlechtesten Punkt gegen einen neuen Punkt erstetzt, der *hoffentlich* besser ist. Der *beste* Punkt ist immer die beste Lösung.

Grundsätzlich funktioniert der Algorithmus so:
Zu Beginn wählt man die N+1 Anfangspunkte. Dazu gibt es mehrere Möglichkeiten, wobei in dieser Arbeit die Startparameter der Punkte gleichverteilt zwischen -10 und 10 gewählt werden. Es werden die Performancewerte für diese Punkte berechnet. Normalerweise gibt es dann ein Abbruchkriterium. Dieses würde den Algorithmus hier beenden lassen, sofern es jetzt schon erfüllt wird. Es gibt bei der Anwedung in diesem Fall nur ein einziges Abbruchkriterium: Die Anzahl der Funktionsaufrufe der Evaluierungsfunktion. Da diese mit 500 Aufrufen noch nicht überschritten ist, ist das Abbruchkriterium hier noch nicht erfüllt. Dann wird der Punkt mit der schlechtesten Performance durch einen neuen Punkt ausgetauscht, der *hoffentlich* eine bessere Performance liefert.

Nach jedem Iterationsschritt wird ein neuer Simplex gebildet. Dazu gibt es folgende Regell (siehe auch Abbildung 2.2): Es wird immer der schlechteste Punkt am Mittelpunkt des restlichen Simplex reflektiert. Dies passiert unter Berücksichtigung des Faktors *alpha*. Daraus ergeben sich vier Möglichkeiten: 1. der neue Punkt ist besser als alle anderen, 2. der Punkt ist wenigstens besser als der zweitschlechteste, 3. der Punkt ist besser als vor der Reflektion, 4. der Punkt ist weiterhin der Schlechteste. Ist der Punkt besser als alle anderen (Fall 1), wird der Schritt, den der Punkt gemacht hat, nochmals um einen Faktor *gamma* vergrößert (Expansion), woraus sich wieder zwei Möglichkeiten ergeben: 1a. der neue Punkt nach der Expansion ist noch besser, dann wird dieser akzeptiert, 1b. der neue Punkt nach der Expansion ist nicht besser als hätte man ihn nur reflektiert, so wird der reflektierte Punkt genommen. Nach 1a und 1b wird wieder von vorne begonnen. Tritt Fall 2 in Kraft, nämlich der reflektierte Punkt ist wenigstens besser als der Zweitschlechteste, so tausche ihn. Fall 3, der Punkt ist wenigstens besser als vor der Reflektion, akzeptiere wenigstens den reflektierten Punkt.
Nun kommt eine weitere Regel für die Fälle zwei bis vier: Kontraktion. Es wird ein neuer Punkt berechnet, indem ein Faktor *beta* mit dem schlechtesten Punkt multipliziert wird. Die Multiplikation mit dem Faktor *beta* bringt den Punkt näher an den Mittelpunkt der anderen Punkte. Ist der Punkt nun besser, akzeptiere ihn und starte von vorne. Ist er noch immer

nicht besser, werden alle Punkte P_i durch $\frac{(P_i+P_{best})}{2}$ ersetzt. Dadurch wird eine Komprimierung vorgenommen und man beginnt wieder von vorne.(Nelder und Mead (1965))

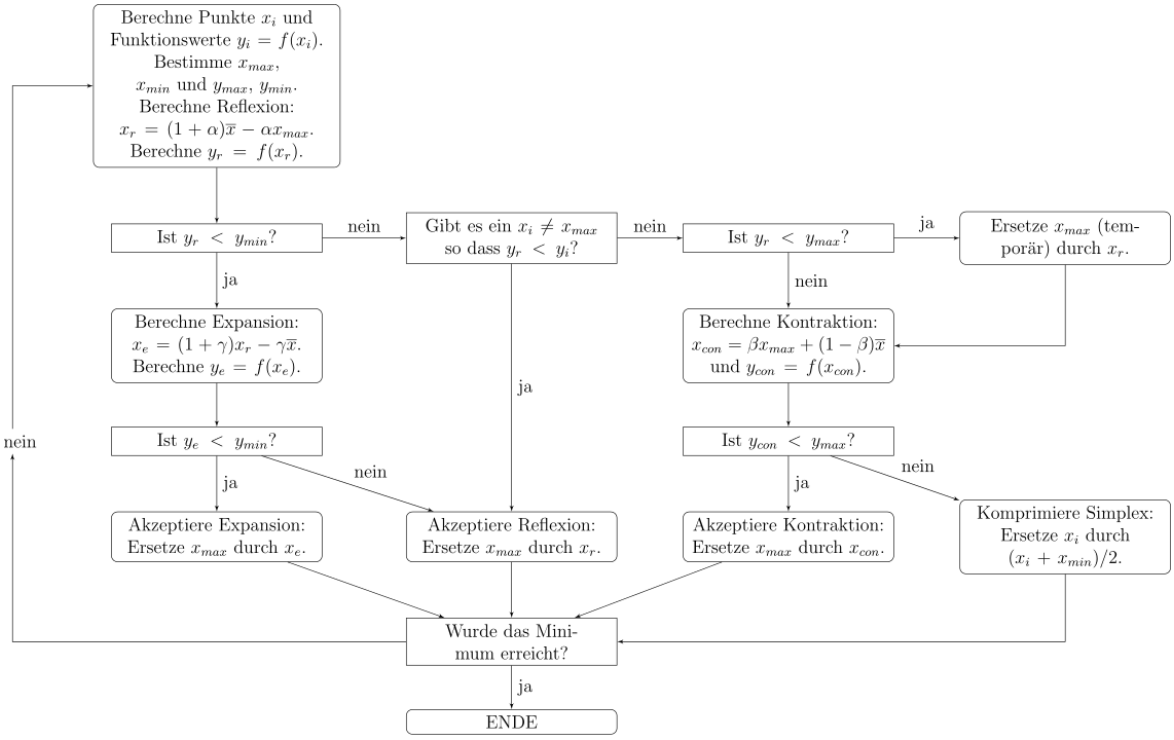

Abbildung 2.2: Flowchart wie Simplex arbeitet. Quelle: Wikipedia

2.4.1 Probleme

Es gibt einige Probleme die mit der bereits vorgefertigten Matlab-Implementation aufgetreten sind. So kann zum Beispiel der Wertebereich, in dem gesucht wird, nicht eingeschränkt werden. Aus diesem Grund wurden verschiedene Methoden implementiert, die den Algorithmus dazu veranlassen, den Wertebereich von -10 bis 10 nicht zu unter-, bzw. überschreiten. Weiters gibt es eine Differenz zwischen der Iterationsanzahl und der Anzahl der Evaluierungsfunktionsaufrufe, da für die Expansion bzw Kontraktion extra überprüft werden muss, ob sich die Performance verändert hat. Da es zwar keine Iteration im Sinne vom Algorithmus ist, wurde dies in dieser Arbeit allerdings sehrwohl als Iteration gewertet, da die Ergebnisse, die verglichen werden, von der Anzahl der berechneten Performancewerten abhängen. Weiters wurde die Matlabfunktion nicht verändert, sondern die standard-Parameter für *alpha = 0.5*, *beta = 0.5* und *gamma = 2* zur Ausführung verwendet. Durch die Initialisierung mit einem gleichverteilten Anfangszustand, kann es vorkommen, dass der Algorithmus schnell, langsam oder gar keine gute Lösung findet. Das Konvergenzverhalten ist nicht vorhersagbar, da es sein kann, dass er in einem Tal *fast* stehen bleibt oder durch die Initialisierung mit Zufallszahlen schnell eine akzeptable Lösung findet. Das spiegelt sich auch in den Ergebnissen in einer großen Standardabweichung wieder.

3 Experimente

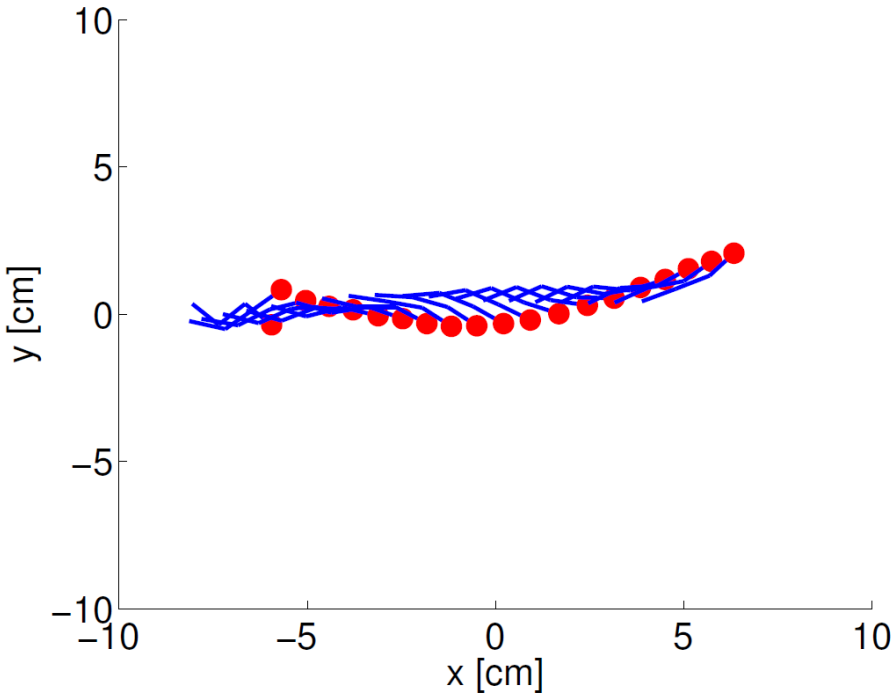

Abbildung 3.1: Symboldarstellung des Schwimmers, der von links nach rechts schwimmt. Rückert (2010)

3.1 Taskbeschreibung

Um die verschiedenen Algorithmen zu vergleichen wird die dynamische Simulation eines Schwimmers verwendet. Dieser Schwimmer (wie in Abb 3.1) soll entsprechend der Kostenfunktion (3.1), welche normalisiert ist ($\frac{1}{N}$), von links nach rechts so weit wie möglich schwimmen. v_i^x ist dabei die Geschwindigkeit in x-Richtung vom Massenmittelpunkt (*Center of Mass - COM*). $u_i^{[1]}$ und $u_i^{[2]}$ beschreiben die zwei Controls für die Verbindungen (Oszillatoren) 1 und 2 im Zeitschritt i.

$$c = \frac{1}{N} \cdot dt \cdot \sum_{i=1}^{N} (-v_i^x + u_i^{[1]} + u_i^{[2]})^2 \tag{3.1}$$

Durch die Vielfalt der möglichen Parameter treten auch vollkommen unterschiedliche Schwimmbewegungen auf. Auch bei zu großer Wahl des Zeitschritts dt kann es vorkommen, dass die

Simulation ausbricht (siehe Abb 3.2 links) und eine entsprechend schlechte Performance liefern würde oder die Lernalgorithmen gar keine Parameter finden können und damit auch nicht in der Lage sind zu konvergieren. Gewollt ist eine gute Schwimmbewegung (schematisch in Abb 3.2 rechts).

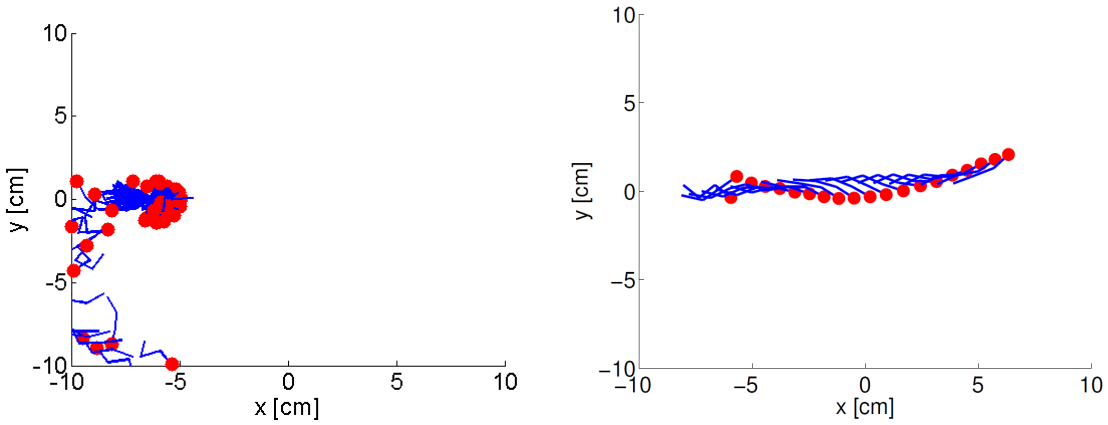

Abbildung 3.2: Schwimmerabbildung links: Schwimmer bricht aus, schlechte Parameterwahl. Schwimmerabbildung rechts: Schwimmer hat gute Schwimmbewegung.

3.2 Auswertung

In den gezeigten Error-Plot Diagrammen findet man jeweils als durchgehende Linie in horizontaler Richtung den Mittelwert über 10 Durchläufe (*Runs*). Dazu in vertikaler Richtung jeweils die Standardabweichung in den jeweiligen Punkten. Die horizontale Achse beschreibt die Iterationsanzahl, die vertikale Achse ist die erreichte Performance.

3.2.1 CMA-ES

Der **CMA-ES** liefer auch im Mittel über 10 Runs eine gute Performance nach relativ kurzer Zeit (siehe dazu: 3.4 - links). Auffällig ist sofort, dass sich die Standardabweichung bei steigender Iterationsanzahl immer kleiner wird. Der Algorithmus findet bei den verschiedenen Aufrufen immer wieder ein ähnlich gutes Ergebnis - und das schon nach ca 200 Iterationen (Vergleich dazu 3.3).

3.2.2 Particle Swarm Optimizer

Der **Particle Swarm Optimizer** liefert im Mittel schon zu Beginn ein gutes Ergebnis (siehe 3.3 Vergleich zwischen CMA und PSO), allerdings mit einer riesen Standardabweichung über 10 Runs (siehe dazu den Plot: 3.4 - mitte). Dies ist damit zu begründen, dass 50 Partikel zufällig initialisiert werden und einer möglicherweise (zufällig) gute Parameter gefunden hat. Die Lernrate ist eher gering, da 50 Partikel zum dem Partikel „hingezogen“ werden, der das

beste Parameterset gefunden hat. Die Standardabweichung würde auch über die doppelte Anzahl an Iterationen kaum besser werden und hängt generell stark von der Initialisierung ab. Es wurden 50 Partikel gewählt, weil die Chance so relativ groß ist, dass ein Partikel ein gutes Parameterset trifft.

3.2.3 SIMPLEX

Der Error-Plot des **SIMPLEX** zeigt, dass dieser Algorithmus noch stärker von der Initialisierung abhängt. Die auf und absteigende Performance begründet sich daraus, dass auch Performances berücksichtigt wurden, die eigentlich nur zum Testen für Expand oder Reduce gegen die Evaluierungsfunktion gedacht waren. Da eine Iteration aber sehrwohl ein Aufruf der Evaluierungsfunktion ist, sind diese ebenfalls enthalten. Anhand von 3.3 sieht man, dass der SIMPLEX eindeutig auch bei besten Performances durch eine zufällig gute Initialisierung das schlechteste Ergebnis der drei Algorithmen liefert. Es gibt zwar eindeutig eine steigende Performance mit steigender Iterationszahl, jedoch ist diese Lernrate bestenfalls als mäßig zu bezeichnen.

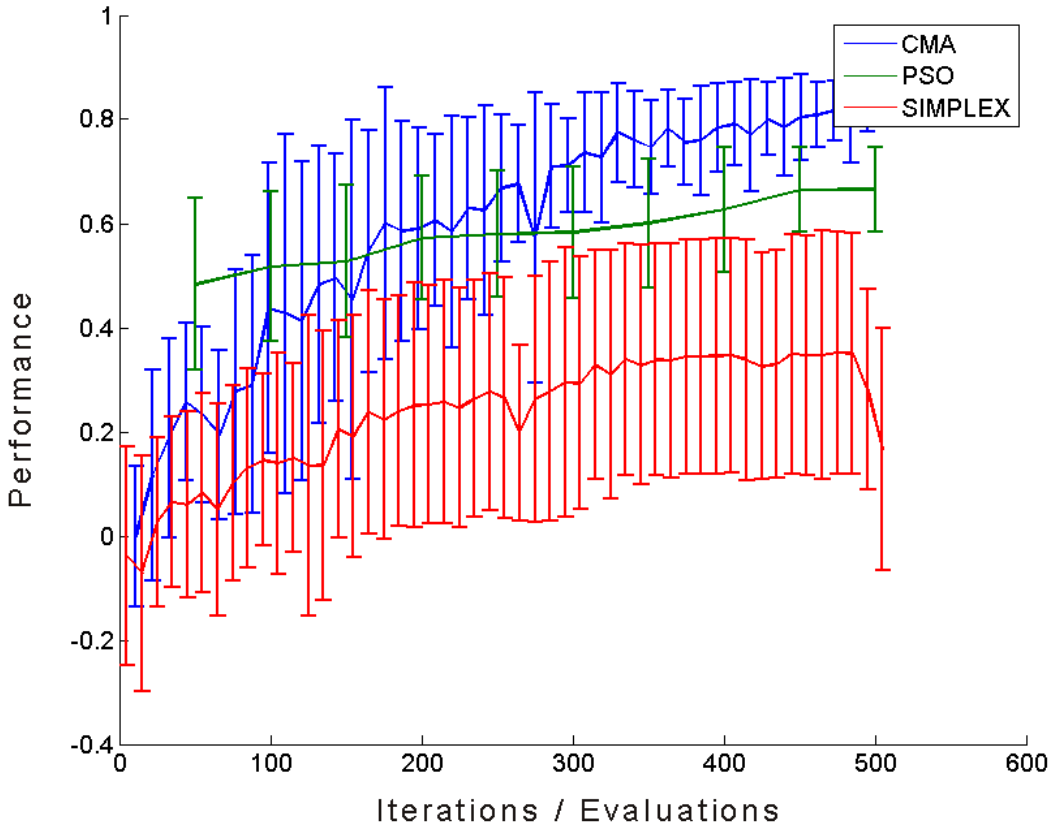

Abbildung 3.3: Vergleich der Algorithmen übereinandergelegt.

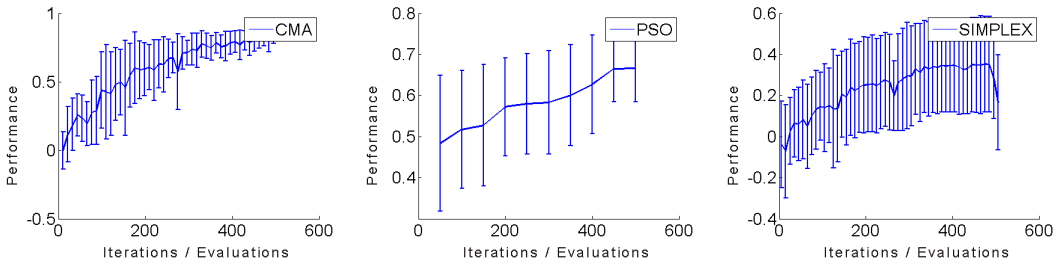

Abbildung 3.4: Diagramme der der verglichenen Lernalgorithmen. Links: CMA-ES, mitte: PSO, rechts: SIMPLEX

4 Zusammenfassung und Fazit

4.1 Zusammenfassung

Zuerst wurde das Problem der hochdimensionalen Suche im Parameterraum definiert und geeignete Algorithmen zur Suche ausgewählt. Anschließend wurde ein Framework geschaffen, welches einen Schwimmer simuliert, wobei 10 Parameter zu finden waren, damit dieser Schwimmer schwimmen „lernt“. Dies war wiederrum die Aufgabe der drei ausgewählten Algorithmen, CMA-ES, PSO und SIMPLEX, welche alle drei 500 Iterationen für die Suche nach einem guten Parameterset Zeit hatten. Anschließend wurden die Ergebnisse verglichen und ausgewertet.

4.2 Fazit

Drei verschiedene Algorithmen wurden zur Auffindung eines guten Parametersets für die Simulation eines Schwimmers verwendet. Die Parametersuche erfolgte im hochdimensionalen Raum, wobei das Parameterset 10 Parameter umfasste. Dies entspricht der Suche in einem 10^{10} dimensionalen Raum. Aus den Ergebnissen sieht man, dass der CMA-ES das beste Ergebnis liefert, dicht gefolgt vom Particle Swarm Optimizer. Der PSO macht sich das „Wissen vieler“ zu Nutze und findet unter gegebenenfalls guter Initialisierung auch ein gutes Parameterset. Die zusätzliche Schwierigkeit war möglicherweise dadurch gegeben, dass es kein sehr eindeutiges optimales Parameterset gibt, sondern viele gute und einige sehr gute Sets. Wagt man den Vergleich der Parametersuche mit dem Besteigen eines Berges, so stünde der Algorithmus vor vielen Hügeln, wobei er nicht weiß, welcher Hügel der höchste ist. Und genau hier liegt die Schwierigkeit für den Algorithmus, welcher dann durch Probieren herausfinden muss, welche Parameter ihn zum Ziel führen. SIMPLEX funktioniert zwar auch, jedoch konvergiert er sehr sehr langsam und liefert auch nur sehr schwer in kurzer Zeit ein annehmbares Ergebnis. Die Schwierigkeit beim PSO ist die Einstellung der 3 Parameter, welche bestimmen wie stark sich die Partikel aufeinander zubewegen und gegenseitig beeinflussen. CMA-ES liefert damit nicht nur das beste Ergebnis, sondern ist durch die quasi einstellungsfreie Implementierung meines Erachtens sogar sehr zu empfehlen.

Literaturverzeichnis

[Hansen 2006] Hansen, N.: The CMA evolution strategy: a comparing review. In: Lozano, J.A. (Hrsg.) ; Larranaga, P. (Hrsg.) ; Inza, I. (Hrsg.) ; Bengoetxea, E. (Hrsg.): *Towards a new evolutionary computation. Advances on estimation of distribution algorithms.* Springer, 2006, S. 75–102

[Kennedy und Eberhart 1995] Kennedy, J. ; Eberhart, R.: Particle swarm optimization. In: *Neural Networks, , IEEE International Conference on* Bd. 4, nov/dec 1995, S. 1942 –1948 vol.4

[Nelder und Mead 1965] Nelder, J. A. ; Mead, R.: A Simplex Method for Function Minimization. In: *The Computer Journal* 7 (1965), Nr. 4, S. 308–313

[Pouya u. a. 2010] Pouya, S. ; Kieboom, J. van den ; Spröandwitz, A. ; Ijspeert, A.J.: Automatic gait generation in modular robots: to oscillate or to rotate; that is the question. In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, oct. 2010, S. 514 –520. – ISSN 2153-0858

[Rückert 2010] Rückert, Elmar: Maschinelles Lernen B, WS 2010/11. 2010

[Schaal u. a. 2005] Schaal, Stefan ; Peters, Jan ; Nakanishi, Jun ; Ijspeert, Auke: Learning Movement Primitives. In: Dario, Paolo (Hrsg.) ; Chatila, Raja (Hrsg.): *Robotics Research* Bd. 15. Springer Berlin / Heidelberg, 2005, S. 561–572