Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik

Lösung eines Kontrollproblems mittels Gaußscher Prozesse

Bachelorarbeit
Dezember 2009

Leo Bronstein
Matrikel–Nummer 310230

Prüfer und Betreuer Prof. Dr. Manfred Opper
Zweitprüfer Prof. Dr. Klaus Obermayer

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt
Berlin, den 28.12.2009

....

Kurzzusammenfassung Die (optimale) Steuerung von dynamischen Systemen ist eine bereits seit längerem betrachtete Problemstellung. In der vorliegenden Arbeit wurde ein von Deisenroth, Rasmussen und Peters entwickeltes und veröffentlichtes Verfahren zur Lösung dieses Problems implementiert und an einem einfachen Beispiel getestet. Dieses Verfahren ist in der Lage, die Dynamik des zu steuernden Systems anhand von Beobachtungen zu Lernen. Es ist nicht darauf angewiesen, ein vollständiges (durch eine Differentialgleichung gegebenes) Modell des Systems zu besitzten.

Abstract The (optimal) control of dynamical systems is a task which has been considered for some time now. The thesis at hand deals with a method for solving this problem which has been developed and published by Deisenroth, Rasmussen and Peters. This method was implemented and tried out on a simple example problem. The method is capable of learning the dynamics of the system by example. A complete model of the system dynamics (such as the one given by a differential equation) is not required.

Danksagung

Ich bedanke mich bei Herrn Prof. Manfred Opper und bei Herrn Dr. Andreas Ruttor für die freundliche Unterstützung beim Schreiben dieser Bachelorarbeit.

Mein Dank gilt auch meinen Eltern, die mich während meines Studiums immer unterstützt haben.

Inhaltsverzeichnis

Einleitung **1**
Aufbau der Arbeit 1

1 Optimalsteuerung und Dynamische Programmierung **3**
1.1 Problemstellung 3
1.2 Der DP-Algorithmus 4

2 Regression mit Gaußschen Prozessen **6**
2.1 Problemstellung und Grundlagen 6
2.2 Kovarianzfunktionen 8
2.3 Wahl der Hyperparameter 11
2.4 Ein einfaches Beispiel 11

3 Gaussian Process Dynamic Programming **13**
3.1 Von Dynamischer Programmierung zu GPDP 13
3.2 Verallgemeinern der Policy 15
3.3 Lernen der Dynamik des Systems 16

4 Experimente **17**
4.1 Versuchsaufbau 17
4.2 Versuchsdurchführung 20
4.3 GPDP mit bekannter Dynamik 23
4.3.1 Abhängigkeit von der Anzahl der Trainingspunkte 23
4.3.2 Abhängigkeit vom Optimierungshorizont 26
4.3.3 Einfluss von zufälligen Störungen der Kostenfunktion 28
4.3.4 Zusammenfassung von GPDP bei bekannter Dynamik 29
4.4 GPDP mit gelernter Dynamik 29
4.4.1 Abhängigkeit von der Anzahl der Trainingspunkte 29
4.5 Allgemeine Bemerkungen und Vergleich mit Deisenroth et al. 33

5 Implementierung **35**

6 Zusammenfassung und Ausblick **37**

Literaturverzeichnis **39**

Abbildungsverzeichnis **39**

Tabellenverzeichnis **42**

A Quellcode **43**

Einleitung

Die optimale Steuerung von dynamischen Systemen ist eine Aufgabenstellung, die bereits seit längerem untersucht wird. Es existieren verschiedene Lösungsansätze, unter anderem die sogenannte Dynamische Programmierung (DP).
Ausgehend hiervon haben Deisenroth, Rasmussen und Peters in [1] und [2] ein Verfahren, genannt Gaussian Process Dynamic Programming (GPDP), entwickelt, das Dynamische Programmierung auf mehrere Weisen verbessert:
Erstens ist es möglich, den für die Lösung des Problems notwendigen Rechenaufwand (sowohl bezüglich Rechenzeit als auch bezüglich Speicher) zu reduzieren. Die „Qualität“ der Lösung sinkt dabei nur minimal.
Zweitens kann man darauf verzichten, dass zu steuernde dynamische System explizit (durch eine Differentialgleichung) zu beschreiben. Stattdessen reicht es aus, einige Beispiele für die Dynamik des Systems bereitzustellen.

Das Ziel der vorliegenden Arbeit war es, dieses Verfahren zu implementieren und anhand des in [1] und [2] aufgeführten Beispiels experimentell nachzuvollziehen. Dabei wurde versucht, die Rollen verschiedener Parameter des Verfahrens genauer zu untersuchen, als dies in [1] und [2] der Fall ist.

Aufbau der Arbeit

Im ersten Kapitel wird die Problemstellung erläutert und der klassische DP-Algorithmus vorgestellt. Es wird auf die Probleme eingegangen, die sich stellen wenn man versucht Dynamische Programmierung auf Systeme mit überabzählbarem Zustandsraum und mit überabzählbar vielen möglichen Steuerungen anzuwenden. Dies motiviert den Aufbau der Arbeit.
Im zweiten Kapitel wird die Regression mit Gaußschen Prozessen (GP) erklärt. Zum besseren Verständnis wird ein sehr einfaches Beispiel einer Anwendung besprochen.
GP-Regression wird für den im dritten Kapitel beschriebenen GPDP-Algorithmus benötigt. Hier wird auch besprochen, auf welche Weise mit unbekannter Dynamik

des zu steuernden Systems umgegangen werden kann.
Im vierten Kapitel wird GPDP auf ein einfaches mechanisches System angewandt. Vorher muss geklärt werden, auf welche Weise das zu steuernde System simuliert werden kann.
Im fünften Kapitel wird kurz auf Implementierungsaspekte eingegangen. Insbesondere wird hier ein Problem, welches bei der Implementierung auftauchte und möglicherweise die in dieser Arbeit erhaltenen Ergebnisse beeinflusste, angesprochen.
Schließlich folgt ein Ausblick auf Fragen, die sich im Anschluss an die vorliegende Arbeit stellen könnten.

Kapitel 1

Optimalsteuerung und Dynamische Programmierung

Dynamische Programmierung (DP) ist ein Standard-Verfahren zur Lösung von Optimierungsproblemen. Dieses Kapitel enthält eine kurze Zusammenfassung der wichtigsten Fakten, die in vielen Büchern nachgelesen werden können. Die folgende Darstellung orientiert sich an [3], wurde aber teilweise vereinfacht.

1.1 Problemstellung

Wir betrachten das folgende Optimierungsproblem:
Gegeben ist ein dynamisches System in diskreter Zeit. Das System kann sich zu jedem Zeitpunkt einer endlichen Menge T von Zeitpunkten in einem Zustand $\mathbf{x}$ einer evtl. überabzählbaren Zustandsmenge X befinden. Weiterhin kann zu jedem Zeitpunkt eine Steuerung $\mathbf{u}$ aus einer im Allgemeinen ebenfalls überabzählbaren Menge U gewählt werden, welche einen Einfluss auf das Verhalten des Systems während des nächsten Zeitschritts hat. Das System ist durch eine Gleichung der Form

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k),$$

gegeben, wobei $k = 0, \ldots, N-1$ die Zeitschritte zählt, $\mathbf{x}_k \in X$ den Zustand des Systems zum Zeitpunkt k angibt und $\mathbf{u}_k \in U$ die zum Zeitpunkt k angewandte Steuerung ist. Es ist zu beachten, dass das Verhalten des Systems zufälligen Störungen unterworfen sein kann, so dass die Zustände $\mathbf{x}_k$ Zufallsvariablen sind.
Das Ziel ist es, dieses System durch die Auswahl der passenden Steuerung „optimal“ zu steuern. Dabei wird durch eine Kostenfunktion $g(\mathbf{x}, \mathbf{u})$ festgelegt, wie „gut“ die Entscheidung ist, die Steuerung $\mathbf{u}$ anzuwenden, wenn sich das System im Zustand $\mathbf{x}$ befindet ist. Im letzten Zeitschritt N fallen zusätzliche Kosten an, abhängig vom Zustand $\mathbf{x}$, in dem sich das System befindet. Diese werden mit $g_{term}(\mathbf{x})$

bezeichnet.
Um das System zu steuern, muss eine Funktion $\mathbf{u} = \pi(\mathbf{x})$ (engl. „policy“) angegeben werden, die zu einem gegebenen Zustand $\mathbf{x}$ des Systems die ausgewählte Steuerung $\mathbf{u}$ angibt. Hat man solch eine Funktion gegeben, so wird durch

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \pi(\mathbf{x}_k))$$

die Entwicklung des Systems bestimmt. Dabei sind die $\mathbf{x}_k$ wieder Zufallsvariablen. Befindet sich das System zum Zeitpunkt 0 im Zustand $\mathbf{x}_0$, so sind bei gegebener Steuerfunktion π die erwarteten Kosten eindeutig bestimmt [3, S. 10 – 12]:

$$V^\pi(\mathbf{x}_0) := \mathbb{E}\left[g_{term}(\mathbf{x}_N) + \sum_{k=0}^{N-1} g(\mathbf{x}_k, \pi(\mathbf{x}_k))\right]$$

Es muss ein Erwartungswert berechnet werden, da die Folge $(\mathbf{x}_k)$ der Zustände auch von zufälligen Störungen abhängt.
Das Ziel ist es nun, diejenige Steuerfunktion π^* zu bestimmen, welche (für einen gegebenen Startzustand $\mathbf{x}_0$) die erwarteten Kosten $V^{\pi^*}(\mathbf{x}_0)$ minimiert. Diese minimalen Kosten werden mit $V^*(\mathbf{x}_0)$ bezeichnet.

1.2 Der DP-Algorithmus

Eine Möglichkeit, dass beschriebene Problem zu lösen, ist die sogenannte *Dynamische Programmierung*. Dieses Verfahren kann folgendermaßen beschrieben werden [3, S. 18 – 19]: Man definiert

$$V_N^*(\mathbf{x}_N) := g_{term}(\mathbf{x}_N) \tag{1.1}$$

und

$$V_k^*(\mathbf{x}_k) := \min_{\mathbf{u} \in U} \mathbb{E}\left[g(\mathbf{x}_k, \mathbf{u}_k) + V_{k+1}^*(f(\mathbf{x}_k, \mathbf{u}_k))\right] \tag{1.2}$$

für $k = 0, \ldots, N-1$. Dann gilt für jeden Startzustand $\mathbf{x}_0$

$$V^*(\mathbf{x}_0) = V_0^*(\mathbf{x}_0),$$

d.h. das in (1.1) und (1.2) definierte Iterationsverfahren berechnet die minimalen Kosten für jeden Startzustand. Außerdem gilt: Eine Policy π^*, welche für jeden Zustand $\mathbf{x}_k$ dasjenige $\mathbf{u}_k = \pi(\mathbf{x}_k)$ wählt, welches die rechte Seite von (1.2) minimiert, realisiert die minimalen Kosten.

Wenn die Zustandsmenge X und die Menge der Steuerungen U endlich sind, ist dieses Verfahren (zumindest im Prinzip) direkt anwendbar. In diesem Fall sei $\mathcal{X} := X$

und $\mathcal{U} := U$. Der Ablauf des Algorithmus kann zum Beispiel so erfolgen (angepasst aus [1]):

Algorithmus 1 Dynamische Programmierung, Teil 1

1: **for all** $\mathbf{x} \in \mathcal{X}$ **do**
2: $\quad V_N^*(\mathbf{x}) = g_{term}(\mathbf{x})$
3: **end for**

Zuerst wird V_N^* mit den terminalen Kosten initialisiert. Nun beginnt die Iteration über alle Zeitpunkte:

Algorithmus 2 Dynamische Programmierung, Teil 2

1: **for** $k = N-1$ to 0 **do**
2: $\quad$ **for all** $\mathbf{x} \in \mathcal{X}$ **do**
3: $\quad\quad$ **for all** $\mathbf{u} \in \mathcal{U}$ **do**
4: $\quad\quad\quad Q_k^*(\mathbf{x}, \mathbf{u}) := g(\mathbf{x}, \mathbf{u}) + \mathbb{E}\left[V_{k+1}^*(f(\mathbf{x}, \mathbf{u}))\right]$
5: $\quad\quad$ **end for**
6: $\quad\quad \pi_k^*(\mathbf{x}) \in \arg\min_{\mathbf{u}} Q_k^*(\mathbf{x}, \mathbf{u})$
7: $\quad\quad V_k^*(\mathbf{x}) := Q_k^*(\mathbf{x}, \pi_k^*(\mathbf{x}))$
8: $\quad$ **end for**
9: **end for**

Für alle Kombinationen aus Zustand $\mathbf{x}$ und Steuerung $\mathbf{u}$ wird $Q_k^*(\mathbf{x}, \mathbf{u})$ berechnet, die erwarteten Kosten, wenn zum Zeitpunkt k im Zustand $\mathbf{x}$ die Steuerung $\mathbf{u}$ gewählt wird, und ab dem Zeitpunkt $k+1$ die optimale Steuerung. Durch die Minimierung von $Q_k^*(\mathbf{x}, \mathbf{u})$ bezüglich $\mathbf{u}$ lässt sich für jedes $\mathbf{x}$ die optimale Steuerung im Zeitschritt k finden. Für $k = 0$ erhält man schließlich die gesuchte optimale Steuerung des Systems.

Der Rechenaufwand für dieses Verfahren beträgt $\mathcal{O}(|\mathcal{X}||\mathcal{U}|)$ Operationen, wenn der Aufwand für das Berechnen des Erwartungswertes in Zeile 4 von Algorithmus 2 nicht berücksichtigt wird: Die innere Schleife wird für jedes $\mathbf{x} \in \mathcal{X}$ einmal ausgeführt, und enthält selbst $\mathcal{O}(|\mathcal{U}|)$ Operationen. Ebenso benötigt die Minimierung in Zeile 6 von Algorithmus 2 $\mathcal{O}(|\mathcal{U}|)$ Operationen.

Oben wurde insbesondere der Fall endlicher Mengen X und U behandelt. Für die in der vorliegenden Arbeit betrachtete Problemstellung ist der Fall $X = \mathbb{R}^{n_x}, U = \mathbb{R}^{n_u}$ relevant. Es stellt sich die Frage, auf welche Weise die beschriebene Iteration sowie die darin enthaltene Minimierung durchgeführt werden können. Mit dieser Frage beschäftigt sich Kapitel 3. Ein hierfür benötigtes Verfahren wird im nächsten Kapitel behandelt.

Kapitel 2

Regression mit Gaußschen Prozessen

Gaußsche Prozesse sind eine „State-of-the-Art"-Methode für Regression und Klassifikation. Dieser Abschnitt gibt einen kurzen Überblick der wichtigsten Fakten zur Regression mit Gaußschen Prozessen.

2.1 Problemstellung und Grundlagen

Betrachtet wird das folgende Problem (siehe [4, S. 8]):
Gegeben sei eine Menge von Beobachtungen

$$\mathcal{D} = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_n, t_n)\} \subset \mathbb{R}^D \times \mathbb{R}.$$

Man geht davon aus, dass es sich um Paare von Eingaben $\mathbf{x}_i$ und (evtl. verrauschten) Ausgaben t_i einer unbekannten Funktion $y(\mathbf{x})$ handelt,

$$t_i = y_i + \varepsilon_i = y(\mathbf{x}_i) + \varepsilon_i,$$

wobei die ε_i unabhängig und identisch normalverteilte Zufallsvariablen sind.
Das Ziel ist es, die zu einem weiteren Eingabewert $\mathbf{x}_*$ gehörende Ausgabe $y_* = y(\mathbf{x}_*)$ oder Beobachtung t_* möglichst genau vorherzusagen.

Die Regression mit Gaußschen Prozessen ist ein Verfahren der Bayes'schen Statistik. Dies bedeutet (siehe [5]), dass zunächst eine Menge von (stochastischen) Modellen für die Daten festgelegt wird. Ein Element M_α dieser Menge hat dann eine Prior-Wahrscheinlichkeit $P(M_\alpha)$. Unter der Annahme, das Modell M_α sei korrekt, haben die vorliegenden Daten $\mathcal{D}$ wiederum die Wahrscheinlichkeit $P(\mathcal{D}|M_\alpha)$. Berücksichtigt man nun sowohl die vorliegenden Daten als auch die Annahme über das Modell, ergibt sich nach dem Satz von Bayes

$$P(M_\alpha|\mathcal{D}) = \frac{P(M_\alpha)P(\mathcal{D}|M_\alpha)}{\sum_\beta P(M_\beta)P(\mathcal{D}|M_\beta)}$$

als Wahrscheinlichkeit dafür, dass Modell M_α gilt, wenn die Daten $\mathcal{D}$ vorliegen. Vorhersagen für den unbekannten Wert y_* an einem Punkt $\mathbf{x}_*$ lassen sich mit Hilfe der Verteilung

$$P(y_*) = \sum_\beta P(y_*|\mathbf{x}_*, M_\beta)P(M_\beta|\mathcal{D})$$

berechnen.
(Hier wurden Wahrscheinlichkeiten und Summen verwendet. Diese müssen evtl. durch Wahrscheinlichkeitsdichten und Integrale ersetzt werden.)

Im Kontext der Regression mit Gaußschen Prozessen nimmt dieses Verfahren die im Folgenden beschriebene Form an. Zunächst definiert man (siehe [6, S. 305])

Definition 1 *Ein Gaußscher Prozess ist eine Wahrscheinlichkeitsverteilung über Funktionen* $y(\mathbf{x})$*, wobei für beliebige* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *die Funktionswerte* $y(\mathbf{x}_1), \ldots, y(\mathbf{x}_n)$ *gemeinsam Normalverteilt sind.*

Gaußsche Prozesse sind bereits eindeutig durch die Angabe von Mittelwert

$$m(\mathbf{x}) = \mathbb{E}\left[y(\mathbf{x})\right]$$

und Kovarianzfunktion

$$k(\mathbf{x}, \mathbf{z}) = \mathbb{E}\left[(y(\mathbf{x}) - m(\mathbf{x}))(y(\mathbf{z}) - m(\mathbf{z}))\right]$$

festgelegt [4, S. 13]. Die Kovarianzfunktion wird auch Kernel genannt.
Nun nimmt man an, dass die Daten durch einen Gaußschen Prozess mit einer bestimmten Erwartungswert- und Kovarianzfunktion erzeugt werden. Dieser Gaußsche Prozess wird im Folgenden auch als „GP-Prior“ bezeichnet. Für den Erwartungswert wird meistens auf Grund mangelnder Informationen $m(\mathbf{x}) \equiv 0$ gesetzt [6, S. 305]. Für die Kovarianz gibt es verschiedene Möglichkeiten. Einige von ihnen sind im Abschnitt 2.2 beschrieben. Damit lässt sich schließlich die Verteilung $p(t_*|\mathbf{x}_*, \mathcal{D})$ bestimmen. Es zeigt sich, dass es sich hierbei um eine Normalverteilung mit Mittelwert

$$\mu(\mathbf{x}_*) = \mathbf{k}^T\mathbf{C}^{-1}\mathbf{t} \tag{2.1}$$

und Varianz

$$\sigma^2(\mathbf{x}_*) = c - \mathbf{k}^T\mathbf{C}^{-1}\mathbf{k}. \tag{2.2}$$

handelt [6, S. 307 – 308]. Dabei ist $\mathbf{C} = [C_{ij}] \in \mathbb{R}^{n\times n}$, $\mathbf{k} = [k_i] \in \mathbb{R}^n$ und

$$\begin{aligned} C_{ij} &= k(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij}\beta^{-1} \\ k_i &= k(\mathbf{x}_i, \mathbf{x}_*) \\ c &= k(\mathbf{x}_*, \mathbf{x}_*) + \beta^{-1} \\ \mathbf{t} &= (t_1, \ldots, t_n) \end{aligned}$$

Hier ist β^{-1} die Varianz des normalverteilten Rauschens, also

$$\varepsilon_i \sim \mathcal{N}(0, \beta^{-1}).$$

β^{-1} ist ein Hyperparameter, dessen Wert zusammen mit den Werten der Hyperparameter der Kovarianzfunktion gefunden werden muss, siehe Abschnitt 2.2.

Der Rechenaufwand für die Regression mit Gaußschen Prozessen wird hauptsächlich durch die Inversion der $(n \times n)$–Matrix $\mathbf{C}$ in den Gleichungen 2.1 und 2.2 bestimmt, was $\mathcal{O}(n^3)$ Operationen benötigt. Allerdings muss dies für eine gegebene Menge $\mathcal{D}$ von Beobachtungen nur einmal durchgeführt werden. Das Vorhersagen des Funktionswertes y_* an einem Punkt $\mathbf{x}_*$ benötigt lediglich $\mathcal{O}(n^2)$ Operationen für eine Matrix-Vektor-Multiplikation. Dennoch kann der Aufwand, wenn die Menge $\mathcal{D}$ groß ist, unpraktikabel werden. In diesem Fall müssen approximative Methoden genutzt werden [6, S. 309–310].

Im Gegensatz zu vielen anderen Regressionsverfahren hat man im Fall von Gaußschen Prozessen auch Informationen über die Varianz der Vorhersage und damit darüber, wie „sicher“ die Vorhersage an einem gegebenen Punkt ist.

2.2 Kovarianzfunktionen

Wie oben beschrieben kann die Funktion, welche die Kovarianz der Funktionswerte zwischen zwei Punkten angibt (unter Berücksichtigung bestimmter Anforderungen) frei gewählt werden. Hierdurch gibt es die Möglichkeit, Annahmen über die zu schätzende Funktion in das Regressionsverfahren hinein zu kodieren [4, S. 79].

Drei typische Kovarianzfunktionen sind im Folgenden beschrieben. Zunächst sei mit $\mathbf{x} = (x_1, \dots, x_D)$ und $\mathbf{z} = (z_1, \dots, z_D)$

$$r := \sqrt{\sum_j \theta_j (x_j - z_j)^2}$$

Dabei sind $\theta_1, \dots, \theta_D$ Hyperparameter. Mögliche Kernel sind zum Beispiel (vergleiche [4, S. 85 – 86])

1. Der *Squared-Exponential-Kernel*

$$k_{SE}(\mathbf{x}, \mathbf{z}) = e^{-r^2} \tag{2.3}$$

2. Der *Matérn-Kernel*

$$k_M(\mathbf{x}, \mathbf{z}) = (1 + \sqrt{3}r)e^{-\sqrt{3}r} \tag{2.4}$$

3. Der *Exponential-Kernel*
$$k_E(\mathbf{x}, \mathbf{z}) = e^{-r}$$

Es können auch verschiedene Kernel miteinander kombiniert werden. So ist die Summe von je zwei der genannten Kernelfunktionen wieder eine Kernelfunktion. Außerdem kann es sinnvoll sein, zu den oben genannten Kerneln einen Term der Form
$$a(\mathbf{x}, \mathbf{z}) := \theta + \theta' \mathbf{x}^T \mathbf{z}$$
zu addieren [6, S. 296, 307]. θ und θ' sind hierbei wieder Hyperparameter.

Um zu sehen, wie die Funktionen aussehen, die von einem Gaußschen Prozess mit einer gegebenen Kovarianzfunktion erzeugt werden, kann man Funktionen der durch den Gaußschen Prozess gegebenen Prior-Verteilung zufällig ziehen. Hierzu müssen zunächst die Punkte, an denen die Funktion ausgewertet werden soll, ausgewählt werden. Seien
$$\mathbf{x}_*^1, \ldots, \mathbf{x}_*^n \in \mathbb{R}^D$$
diese Punkte. Man bildet die Kovarianzmatrix
$$\mathbf{K} = [K_{ij}] = [k(\mathbf{x}_*^i, \mathbf{x}_*^j)] \in \mathbb{R}^{n \times n},$$
wobei $k(\cdot, \cdot)$ die Kernelfunktion ist, und zieht einen normalverteilten Zufallsvektor $(y_*^1, \ldots, y_*^1) \sim \mathcal{N}(\mathbf{0}, \mathbf{K})$. Der Funktionswert der zufällig gezogenen Funktion am Punkt $\mathbf{x}_*^i$ ist dann y_*^i. Siehe hierzu [4, S. 14].
Die folgende Grafik zeigt exemplarisch einige auf die beschriebene Weise gezogene Funktionen für verschiedene Kernel. Die Auswertung der Funktionen erfolgte dabei an 2000 auf dem Intervall $[0, 10]$ gleichmäßig verteilten Punkten.

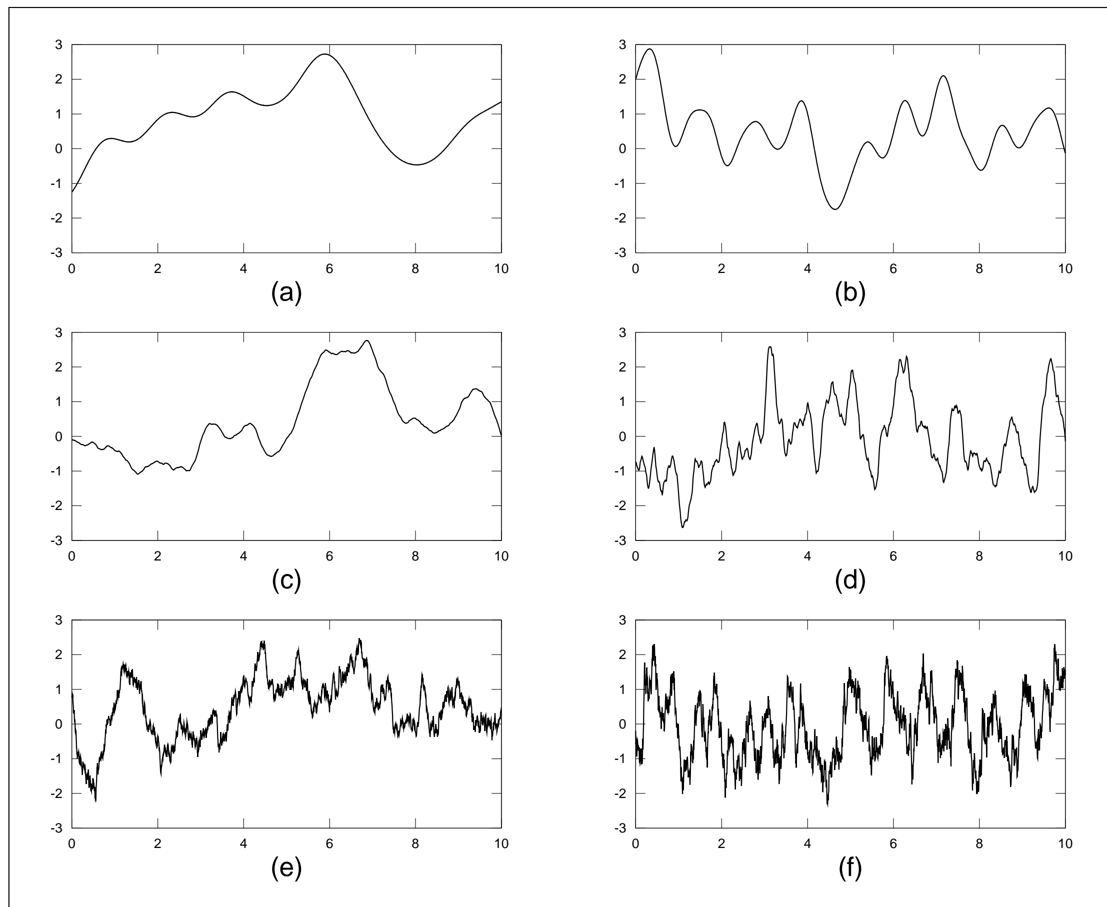

Abbildung 2.1: Zufällig gezogene Funktionen aus einem GP-Prior mit verschiedenen Kovarianzfunktionen.

Die benutzten Kernelfunktionen waren

a) $k(x, z) = e^{-1\cdot(x-z)^2}$

b) $k(x, z) = e^{-5\cdot(x-z)^2}$

c) $k(x, z) = (1 + 1 \cdot \sqrt{3}|x - z|)e^{-1\cdot\sqrt{3}|x-z|}$

d) $k(x, z) = (1 + 5 \cdot \sqrt{3}|x - z|)e^{-5\cdot\sqrt{3}|x-z|}$

e) $k(x, z) = e^{-1\cdot|x-z|}$

f) $k(x, z) = e^{-5\cdot|x-z|}$

2.3 Wahl der Hyperparameter

Die aufgeführten Kernel hängen von der Wahl einiger Hyperparameter ab. Es stellt sich die Frage, wie diese gewählt werden müssen. Eine einfache Methode hierzu ist Maximum-Likelihood. Dabei wählt man diejenigen Hyperparameter $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n)$, für welche der Logarithmus der Likelihood-Funktion $p(\mathbf{t}|\boldsymbol{\theta})$ maximal wird. Das Maximum kann dabei an mehreren Punkten $\boldsymbol{\theta}$ angenommen werden. Es stellt sich heraus (siehe [6, S. 311]), dass

$$p(\mathbf{t}|\boldsymbol{\theta}) = -\frac{1}{2}\left[\ln|\mathbf{C}| + \mathbf{t}^T\mathbf{C}^{-1}\mathbf{t} + N\ln(2\pi)\right].$$

Um die Optimierung effizient durchführen zu können, ist es nützlich auch den Gradienten von $p(\mathbf{t}|\boldsymbol{\theta})$ nach $\boldsymbol{\theta}$ zu kennen. Es ergibt sich

$$\frac{\partial}{\partial\theta_i}\ln p(\mathbf{t}|\boldsymbol{\theta}) = \frac{1}{2}\left[\mathbf{t}^T\mathbf{C}^{-1}\frac{\partial\mathbf{C}}{\partial\theta_i}\mathbf{C}^{-1}\mathbf{t} - \mathrm{Tr}\left(\mathbf{C}^{-1}\frac{\partial\mathbf{C}}{\partial\theta_i}\right)\right]$$

Hierbei ist es offenbar notwendig, die Ableitungen der Kovarianzfunktion berechnen zu können.

2.4 Ein einfaches Beispiel

Um die Regression mit Gaußschen Prozessen zu demonstrieren, folgt hier ein einfaches Beispiel mit eindimensionalem Eingaberaum.
Man betrachtet die Funktion $h(x) = e^{-x/5}\sin(x)$ und wählt 6 Punkte im Intervall $[1, 10]$ aus. Zu den Funktionswerten an diesen Stellen wird unabhängiges, normalverteiltes Rauschen mit Mittelwert 0 und Standardabweichung 0.05 addiert. Die Regression wird basierend auf diesen 6 Trainingspunkten durchgeführt. Als Kovarianzfunktion wird der Squared-Exponential-Kernel verwendet und die Hyperparameter mit Hilfe der Maximum-Likelihood-Methode gefunden. Das Ergebnis ist in Grafik 2.2 dargestellt.

Es ist zu sehen, dass in dem Bereich, in dem Trainingspunkte vorliegen, die Funktion gut approximiert wird. In unmittelbarer Umgebung eines Trainingspunktes ist die Unsicherheit sehr klein. In den Bereichen ohne Trainingspunkte steigt die Unsicherheit der Vorhersage hingegen schnell an. In den Bereichen, in denen keine Trainingspunkte mehr liegen (das Intervall $[11, 14]$) kann der Verlauf der Funktion natürlich nicht mehr vorhergesagt werden.

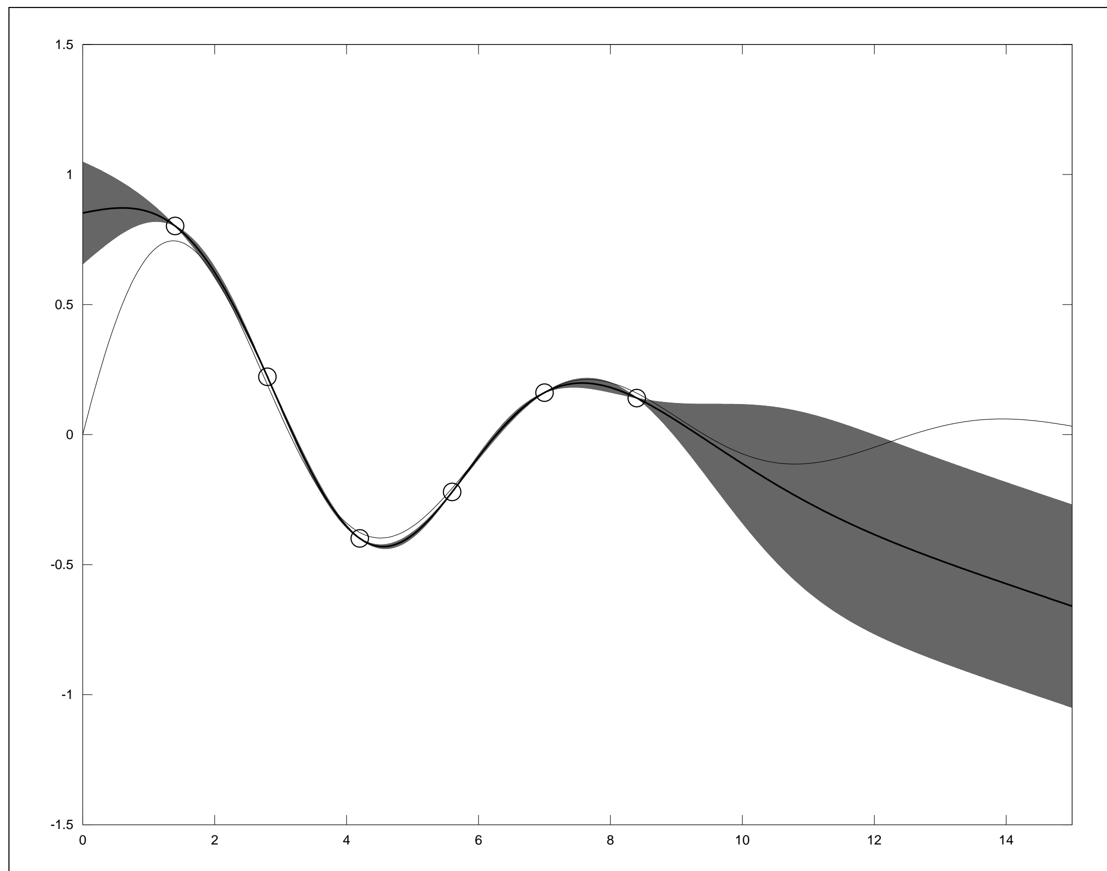

Abbildung 2.2: Beispiel zur Regression mit Gaußschen Prozessen.
Dünne Kurve: Zu approximierende Funktion $h(x)$.
Durch Kreise markierte Punkte: Trainingspunkte.
Dicke Kurve: Durch Regression bestimmte Funktion (Mittelwert der Vorhersage).
Schattierter Bereich: $\pm$ 1 Standardabweichung um den Mittelwert.

Kapitel 3

Gaussian Process Dynamic Programming

Bei der Anwendung des im letzten Kapitel beschriebenen Verfahrens ergibt sich ein Problem: Wie verfährt man, wenn der Zustandsraum des Systems und die Menge der möglichen Steuerungen überabzählbar unendlich sind?
Eine Möglichkeit ist es, diese Mengen zu diskretisieren, d.h. in endlich viele, hinreichend kleine Teile einzuteilen. Problematisch hieran ist, dass die Anzahl der Teile mit der Dimension des Raumes exponentiell ansteigt, was die praktische Anwendung erschwert.
Um mit diesem Problem umzugehen, wurden verschiedene Methoden entwickelt, die auf der Approximation der bei DP auftretenden Funktionen beruhen. Ein solches Verfahren, genannt „Gaussian Process Dynamic Programming“ (GPDP), wurde von Deisenroth, Rasmussen und Peters in [1] vorgestellt und in [2] erweitert. Im Folgenden wird dieses Verfahren erläutert. Dabei gibt das gesamte Kapitel Inhalte aus [1] und [2] wieder. Allerdings wurde die Darstellung an einigen Stellen leicht verändert.

3.1 Von Dynamischer Programmierung zu GPDP

Die Grundidee des Verfahrens ist relativ einfach: Anstatt den Phasenraum und die Menge der Steuerungen mit einem engmaschigen Gitter zu überziehen, um diese so zu diskretisieren und dann den üblichen DP-Algorithmus anzuwenden, wird ein sehr grobes Gitter verwendet. Die Funktionswerte der V- und Q-Funktionen werden dann mit Hilfe der Regression mit Gaußschen Prozessen auf den relevanten Teil des Raumes verallgemeinert.
Man beginnt, indem die Teilmengen $\mathcal{X}_{ctl}$ des Phasenraums und $\mathcal{U}_{ctl}$ der Steuerungen gewählt werden, die als „Stützstellen“ für den Algorithmus fungieren. Für diese

Stützstellen werden die Werte der V- und Q-Funktionen für die jeweilige Iteration des Algorithmus wie beim klassischen DP bestimmt. Anschließend werden diese Funktionen auf den gesamten relevanten Teil des Raumes verallgemeinert. Hierbei kommt die Regression mit Gaußschen Prozessen zum Einsatz. In der darauf folgenden Iteration des Algorithmus können dann die notwendigen Werte an jeder Stelle des (relevanten Teil–) Raumes bestimmt werden. Außerdem können bei der Minimierung der Q-Funktion alle Punkte der Menge der Steuerungen berücksichtigt werden.
Der GPDP-Algorithmus lässt sich wie folgt formulieren:

Algorithmus 3 GPDP, Teil 1

1: **for all** $\mathbf{x} \in \mathcal{X}_{ctl}$ **do**
2: $\quad V_N^*(\mathbf{x}) = g_{term}(\mathbf{x})$
3: **end for**
4: $V_N^*(\cdot) \sim \mathcal{GP}$

Zuerst wird wie beim DP-Algorithmus V_N^* mit den terminalen Kosten initialisiert. Dabei enthält die Menge $\mathcal{X}_{ctl}$ im Allgemeinen deutlich weniger Elemente als für DP. In Zeile 4 wird V_N^* ausgehend von den vorher berechneten Punkten verallgemeinert. Die verwendete Notation stammt aus [1].

Algorithmus 4 GPDP, Teil 2

1: **for** $k = N-1$ to 0 **do**
2: $\quad$ **for all** $\mathbf{x} \in \mathcal{X}_{ctl}$ **do**
3: $\quad\quad$ **for all** $\mathbf{u} \in \mathcal{U}_{ctl}$ **do**
4: $\quad\quad\quad Q_k^*(\mathbf{x}, \mathbf{u}) := g(\mathbf{x}, \mathbf{u}) + \mathbb{E}\left[V_{k+1}^*(f(\mathbf{x}, \mathbf{u}))\right]$
5: $\quad\quad$ **end for**
6: $\quad\quad Q_k^*(\mathbf{x}, \cdot) \sim \mathcal{GP}$
7: $\quad\quad \pi_k^*(\mathbf{x}) \in \arg\min_{\mathbf{u}} Q_k^*(\mathbf{x}, \mathbf{u})$
8: $\quad\quad V_k^*(\mathbf{x}) = Q_k^*(\mathbf{x}, \pi_k^*(\mathbf{x}))$
9: $\quad$ **end for**
10: $\quad V_k^*(\cdot) \sim \mathcal{GP}$
11: **end for**
12: **return** π_0^*

Wie beim DP-Algorithmus wird rückwärts über alle Zeitschritte iteriert. Alle dabei berechneten Funktionen V_k^* werden auf den gesamten Phasenraum verallgemeinert, bevor sie in der nächsten Iteration benutzt werden. Dies geschieht in Zeile 10. Das selbe geschieht mit den Q_k^*. Es ist zu beachten, dass Q_k^* nicht mit einer einzigen Regression (mit Paaren $(\mathbf{x}, \mathbf{u})$ als Eingaben) verallgemeinert wird.

Stattdessen wird für jeden Zustand $\mathbf{x} \in \mathcal{X}_{ctl}$ eine eigene Regression durchgeführt. In [1] wird dies damit begründet, dass sonst die Anzahl der Trainingspunkte sehr groß sein müsste. Außerdem kann Q sowohl als Funktion von $\mathbf{x}$, als auch als Funktion von $\mathbf{u}$ unstetig sein. Dadurch, dass man Q_k^* für verschiedene Zustände einzeln behandelt, kann eine Quelle von Unstetigkeit ausgeschlossen werden.

Der Mittelwert des GP-Priors der in den Zeilen 6 und 10 erstellten Modelle wird nicht, wie in Kapitel 2 beschrieben, konstant gleich 0 gesetzt, sondern in der k-ten Iteration konstant gleich k. Laut [1] hat dies zur Folge, dass Zustände, die sich weiter von den Trainingspunkten entfernt befinden, als weniger günstig angesehen werden.

Das vorgestellte Verfahren benötigt $\mathcal{O}(|\mathcal{X}_{ctl}||\mathcal{U}_{ctl}|^3 + |\mathcal{X}_{ctl}|^3)$ Operationen:
$\mathcal{O}(|\mathcal{U}_{ctl}|^3)$ Operationen für das Erzeugen des GP-Modells für Q in Zeile 6, was einmal für jedes $\mathbf{x} \in \mathcal{X}_{ctl}$ durchgeführt werden muss. Außerdem $\mathcal{O}(|\mathcal{X}_{ctl}|^3)$ Operationen für das GP-Modell für V in Zeile 10. Allerdings sind die Mengen $\mathcal{X}_{ctl}$ und $\mathcal{U}_{ctl}$ im Allgemeinen wesentlich kleiner als die Mengen $\mathcal{X}$ und $\mathcal{U}$ im klassischen DP-Algorithmus.

3.2 Verallgemeinern der Policy

Der Algorithmus liefert eine Policy π_0^* als Ergebnis. Diese gibt die Steuerung für jeden der endlich vielen Trainingspunkte an. Um diese Funktion zur Steuerung des Systems anwenden zu können, ist es notwendig sie auf den gesamten Eingaberaum zu verallgemeinern. Wie dies geschieht, wird von dem konkret vorliegenden System abhängen. Die einfachste Möglichkeit, nämlich eine Regression durchzuführen, hat den Nachteil, dass die optimale Steuerfunktion im Allgemeinen unstetig sein wird. Im Fall von Gaußschen Prozessen z.B. ist es schwierig, eine passende Kovarianzfunktion zu finden. Deshalb wird in [1] ein anderes Verfahren vorgeschlagen:

Bei der Steuerung von mechanischen Systemen ist es vorteilhaft, wenn die Steuerfunktion glatt ist, da dies die Aktoren des Systems schützt. Deshalb geht man davon aus, dass die (angenähert) optimale Steuerfunktion zumindest stückweise glatt ist, und Unstetigkeiten höchstens an den Stellen auftreten, an denen das Vorzeichen der Steuerung wechselt. Hier wird angenommen, dass die Steuerung ein Skalar ist. In diesem Fall bietet es sich an, ausgehend von der Policy π_0^* zwei verschiedene Steuerfunktionen zu lernen: Eine für die Punkte des Eingaberaums, an denen die Steuerung positiv ist, und eine für diejenigen Punkte, an denen die Steuerung negativ ist. Um für einen gegebenen Zustand $\mathbf{x}$ des Systems die Steuerung zu bestimmen, ist es dann notwendig festzustellen,

welche der beiden Steuerfunktionen gewählt werden muss. Dies kann mit Hilfe eines binären Klassifikators entschieden werden.

Man kann sich die Frage stellen, ob es sinnvoll wäre die Steuerung für einen Zustand $\mathbf{x}$ zu bestimmen, in dem man das (gewichtete) Mittel der beiden Steuerfunktionen bildet. Dies ist, wie in [1] angemerkt wird, im Allgemeinen nicht sinnvoll.

3.3 Lernen der Dynamik des Systems

Das oben beschriebene Verfahren wurde in [2] noch einmal erweitert. Vorher wurde angenommen, dass die Dynamik des zu steuernden Systems bekannt ist, d.h. in Form einer Differentialgleichung gegeben. Stattdessen wird nun davon ausgegangen, dass lediglich Beispiele für das Verhalten des Systems vorliegen, d.h. eine endliche Menge

$$\mathcal{S} = \{((\mathbf{x}_1, \mathbf{u}_1), f(\mathbf{x}_1, \mathbf{u}_1)), \ldots, ((\mathbf{x}_m, \mathbf{u}_m), f(\mathbf{x}_m, \mathbf{u}_m))\} \subset \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_x}$$

gegeben ist. Dies ist folgendermaßen zu verstehen: Befindet sich das System zu einem bestimmten Zeitpunkt t im Zustand $\mathbf{x}$, und wird während des Zeitintervalls $(t, t + \Delta t)$ die Steuerung $\mathbf{u}$ angewendet, dann befindet sich das System zum Zeitpunkt $t + \Delta t$ im Zustand $f(\mathbf{x}, \mathbf{u})$. Hierbei kann $f(\mathbf{x}, \mathbf{u})$ verrauscht sein, also nur approximativ den wahren Zustand des Systems angeben.
Anhand dieser Trainingsdaten wird das Verhalten des Systems mit Hilfe der Regression mit Gaußschen Prozessen auf den gesamten Eingaberaum verallgemeinert. Dieses „gelernte“ Modell wird im GPDP-Algorithmus anstelle des durch eine Differentialgleichung beschriebenen exakten Modells verwendet. Dies hat zur Folge, dass der in Zeile 4 von Algorithmus 4 benötigte Erwartungswert nicht mehr analytisch berechnet werden kann. Stattdessen kann er z.B. mit Hilfe einer Monte-Carlo Integration numerisch bestimmt werden. Falls aber genügend Trainingsdaten vorliegen, geht die Unsicherheit im Modell für f gegen Null. In diesem Fall ist

$$\mathbb{E}\left[V_{k+1}^*(f(\mathbf{x}, \mathbf{u}))\right] = \mu_v(\mu_f(\mathbf{x}, \mathbf{u})). \tag{3.1}$$

Das gesamte Verfahren hat somit die folgende Form:

1. Lernen der Dynamik des zu steuernden Systems anhand der Trainingsdatenmenge $\mathcal{S}$.

2. Auffinden der (auf $\mathcal{X}$ beschränkten) Policy π_0^* mit Hilfe des GPDP-Algorithmus.

3. Verallgemeinern der von GPDP gefundenen Policy auf den gesamten Zustandsraum.

Kapitel 4

Experimente

In diesem Kapitel wird ein Experiment beschrieben, anhand dessen das in Kapitel 3 erläuterte Verfahren überprüft werden soll. Dieser Versuchsaufbau orientiert sich an [1] und [2].

Das Vorgehen ist wie folgt: Zunächst wird das dynamische System beschrieben, das gesteuert werden soll. Danach wird untersucht, wie gut das in Kapitel 3 beschriebene Verfahren funktioniert, wenn die exakte Dynamik des Systems bekannt ist. Im Anschluss wird überprüft, inwiefern sich die Ergebnisse ändern, wenn die Dynamik des Systems unbekannt ist und erst aus Beispielen gelernt werden muss. Schließlich werden die erhaltenen Resultate mit denen aus [1] und [2] verglichen.

4.1 Versuchsaufbau

Betrachtet wird ein Pendel, das an einem masselosen Stab in einer Ebene schwingt (siehe Abbildung 4.1). Sei φ die Auslenkung des Pendels aus der instabilen Gleichgewichtslage, d.h. es ist $\varphi = 0$ in der oberen und $\varphi = \pi$ in der unteren Position. Die Bewegungsgleichung lautet dann

$$\ddot{\varphi}(t) = \frac{-\mu\dot{\varphi}(t) + mgl\sin(\varphi(t)) + u(t)}{ml^2}$$

Dabei ist $l = 1\,\mathrm{m}$ die Länge des Stabes, $m = 1\,\mathrm{kg}$ die Masse des Pendels, $\mu = 0.05\,\mathrm{kg}\cdot\mathrm{m}^2/\mathrm{s}$ eine Reibungskonstante und $g = 9.81\,\mathrm{m}^2/\mathrm{s}$ die Gravitationskonstante. Das Drehmoment $u(t)$ kann sich in dem Bereich $[-5, 5]$Nm bewegen. Es lässt sich beliebig festlegen und das Pendel dadurch beeinflussen.

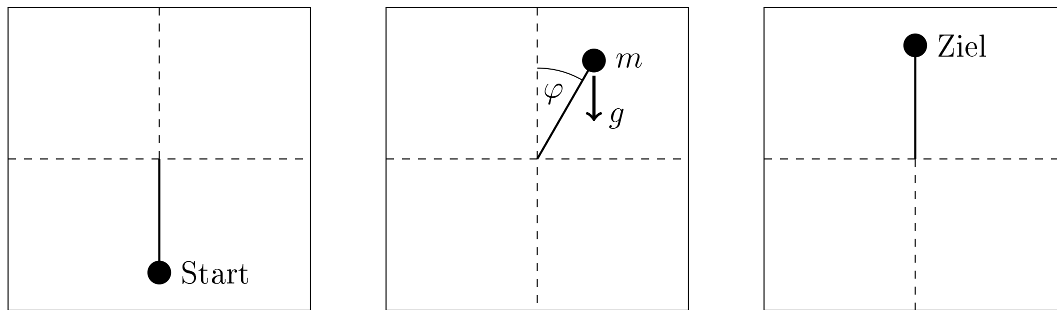

Abbildung 4.1: Skizze des Pendels sowie Start- und Zielposition.

Die zu lösende Aufgabe lautet, eine Steuerfunktion $u(t)$ zu bestimmen, die das Pendel aus der stabilen in die unstabile Gleichgewichtslage bringt und dort balanciert. Zusätzlich soll die hierfür benötigte Zeit so klein wie möglich sein. Ein entscheidendes Merkmal der Aufgabe ist, dass das maximale Drehmoment nicht ausreicht, um das Pendel „in einem Schwung“ nach oben zu befördern. Folglich muss das Pendel zuerst in eine Richtung angeschwungen werden, um dann in die entgegengesetzte Richtung in die Zielposition gebracht zu werden.

Dieses System wurde nicht tatsächlich nachgebaut, sondern auf dem Computer simuliert. Dafür ist es notwendig, die Bewegungsgleichung des Pendels zu diskretisieren. Dies geschieht mittels der Gleichungen

$$\begin{aligned} \varphi_{k+1} &= \varphi_k + \dot{\varphi}_k dt + \ddot{\varphi}_k \frac{dt^2}{2} \\ \dot{\varphi}_{k+1} &= \dot{\varphi}_k + \ddot{\varphi}_k dt \end{aligned}$$

wobei $\varphi_k = \varphi(kdt)$ ist und dt (hier eine endliche Größe) passend gewählt werden muss. Dabei zählt $k \in \mathbb{N}_0$ die Zeitschritte der Simulation.
Ob ein gegebenes dt geeignet ist, lässt sich zumindest heuristisch überprüfen, indem die freie Schwingung des Pendels aus einer vorgegebenen Lage betrachtet wird. Die folgende Grafik zeigt den Winkel φ abhängig von der Zeit t für vier verschiedene Werte von dt, wenn das Pendel aus der Position $\varphi = \pi/2$, $\dot{\varphi} = 0$ fallengelassen wird.

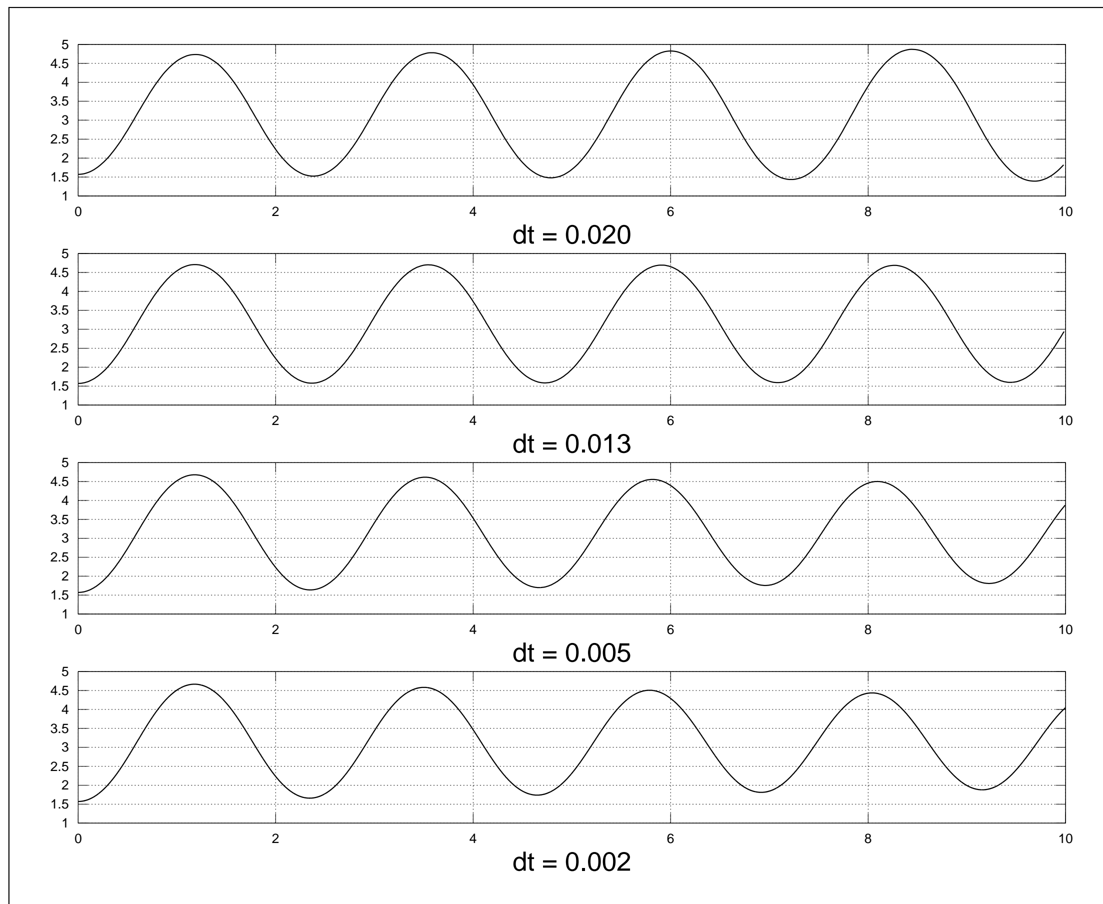

Abbildung 4.2: Verlauf der Auslenkung des Pendels über die Zeit bei verschiedenen Werten für dt. Das Pendel wird aus der Position $\varphi = \pi/2$, $\dot{\varphi} = 0$ fallen gelassen. Die Steuerung ist konstant gleich 0.

Es ist zu sehen, dass für $dt = 0.020\,\mathrm{s}$ die Amplitude der Schwingungen zunimmt, obwohl Reibung existiert und die Steuerung u konstant gleich 0 ist. Dies kann für ein reales Pendel offenbar nicht gelten. Für $dt = 0.013\,\mathrm{s}$ bleibt die Amplitude zumindest konstant. Im Gegensatz hierzu nimmt die Amplitude der Schwingungen für $dt = 0.005\,\mathrm{s}$ mit der Zeit ab, was bei einem realen Pendel auch zu erwarten ist. Für $dt = 0.002\,\mathrm{s}$ unterscheidet sich die Kurve nicht mehr wesentlich von dem Fall $dt = 0.005\,\mathrm{s}$. Aus diesem Grund wurde für die folgenden Versuche $dt = 0.005\,\mathrm{s}$ gewählt.

Betrachtet man das beschriebene Problem, so fällt auf, dass es nicht die in den Kapiteln 1 und 3 verlangte Form hat. Insbesondere gibt es weder eine „natürliche“ Diskretisierung, noch einen „natürlich“ vorgegebenen Zeitpunkt, zu dem das System angehalten wird. Stattdessen gilt die Aufgabe dann als gelöst, wenn das

Pendel in der Zielposition balanciert wird – unabhängig davon, wie lange es dauert diesen Zustand zu erreichen.
Um das vorliegende Problem auf die benötigte Form zu reduzieren, ist es notwendig ein Zeitintervall Δt festzulegen, welches angibt, in welchen Abständen die Steuerung u neu gewählt werden kann. Wird zu einem gegebenen Zeitpunkt t eine Steuerung u gewählt, so bleibt sie im Intervall $(t, t + \Delta t)$ konstant gleich u.
Außerdem muss festgelegt werden, über wie viele Zeitschritte die Iteration im GPDP-Algorithmus ausgeführt werden soll. Diese Zahl (im GPDP-Algorithmus als N bezeichnet) muss groß genug sein, damit die optimale Steuerung tatsächlich gefunden werden kann.
Das Produkt $h = N\Delta t$ wird als Optimierungshorizont bezeichnet.

Der Phasenraum dieses Systems ist $[-\pi, \pi) \times \mathbb{R}$. Allerdings genügt es, lediglich Winkelgeschwindigkeiten im Bereich $[-7, 7]$rad/s zu betrachten, wie dies auch in [1] und [2] getan wird. Andererseits ist es nützlich, den Winkel φ auch Werte außerhalb $[-\pi, \pi)$ annehmen zu lassen, obwohl dann die Zuordnung zwischen den Zuständen des Systems und den Punkten des Phasenraums nicht mehr eineindeutig ist.

4.2 Versuchsdurchführung

Das in Kapitel 3 beschriebene Verfahren hängt von einer Reihe von Parametern ab. Insbesondere müssen die folgenden Punkte geklärt werden:

- Wie viele Trainingspunkte für GPDP werden verwendet? Wo im Phasenraum und im Raum der Steuerungen sind diese platziert?
- Welche Kovarianzfunktion wird für die Regressionen im GPDP-Algorithmus verwendet?
- Wie wird die von GPDP gefundene Policy von der endlichen Menge der Trainingsdaten auf den gesamten Phasenraum verallgemeinert?
- Ist die in Zeile 2 von Algorithmus 3 und Zeile 4 von Algorithmus 4 (siehe S. 14) benötigte Kostenfunktion mit Rauschen behaftet? Falls ja, wie stark ist dieses?

Soll auch die Dynamik des Pendels aus Beispielen gelernt werden, kommen unter anderem die folgenden Punkte hinzu:

- Wie viele Trainingspunkte werden für das Lernen der Dynamik des Systems verwendet und wo werden sie platziert?

- Welche Kovarianzfunktion wird für das Lernen der Dynamik verwendet?
- Sind die Trainingsdaten für die Dynamik verrauscht? Wie stark?

Bei den folgenden Versuchen wurde typischerweise einer dieser Parameter variiert, während alle anderen unverändert blieben. Deswegen wird hier eine Auflistung derjenigen Parameterwerte gegeben, die genutzt wurden, wenn in der Beschreibung des jeweiligen Versuchs nichts anderes behauptet wird.

Im Weiteren benutzte ich folgende
Notation Für $a < b$ und $n \in \mathbb{N}$ sei $[a, b]_n := \left\{a + \frac{b-a}{n} j \mid j = 0, \ldots, n\right\}$.

Für den GPDP-Algorithmus musste die Anzahl und Position der Trainingspunkte festgelegt werden. Hierfür wurde zunächst der relevante Teil des Phasenraums und der Menge der Steuerungen mit einem gleichmäßigen Punktgitter überzogen. Diese waren von der Form

$$\mathcal{X}_1 := [-6\pi/5, 6\pi/5]_p \times [-7, 7]_q$$

und

$$\mathcal{U}_{ctl} := [-5, 5]_{25}.$$

Die Parameter p und q, welche die Anzahl der Trainingspunkte im Phasenraum festlegen, wurden je nach Versuch unterschiedlich gewählt. Beim Phasenraum wurde um den Zielzustand $\varphi = 0$, $\dot{\varphi} = 0$ zusätzlich ein dichter liegendes Punktgitter der Form

$$\mathcal{X}_2 := [-6\pi/25, 6\pi/25]_5 \times [-7/5, 7/5]_7$$

eingefügt, so dass die endgültige Trainingspunktmenge

$$\mathcal{X}_{ctl} := \mathcal{X}_1 \cup \mathcal{X}_2$$

war. Man beachte: Der relevante Teil des Phasenraums ist in Richtung der $\dot{\varphi}$-Koordinate fast doppelt so lang wie in Richtung der φ-Koordinate. Dennoch stellte es sich heraus, dass es nützlich ist, die Punkte in einer Richtung dichter als in der anderen liegen zu lassen.
Ähnlich wurde mit den Trainingspunkten für das Lernen der Dynamik des Pendels verfahren. Hier wurde ebenfalls ein gleichmäßiges Gitter von Punkten

$$\mathcal{S} := \{((\mathbf{x}, \mathbf{u}), f(\mathbf{x}, \mathbf{u})) | \mathbf{x} \in \mathcal{X}_{dyn}, \mathbf{u} \in \mathcal{U}_{dyn}\}.$$

verwendet, wobei

$$\mathcal{X}_{dyn} := [-6\pi/5, 6\pi/5]_a \times [-7, 7]_b \tag{4.1}$$

und

$$\mathcal{U}_{dyn} := [-5, 5]_c \tag{4.2}$$

war. Für die Parameter a, b und c wurden je nach Versuch unterschiedliche Werte gewählt.
Bei jedem Durchlauf des Algorithmus wurde jeder dieser Punkte um eine zufällige, normalverteilte Korrektur mit Mittelwert 0 und Standardabweichung 0.1 unabhängig von einander in Richtung von jeder der drei Koordinatenachsen φ, $\dot{\varphi}$ und u verschoben. Da die Ergebnisse des Algorithmus auch von der Position der Trainingspunkte abhängen, wird hiermit sichergestellt, das ausreichend viele unterschiedliche Positionen probiert werden.

Für den Optimierungshorizont wird in [1] und [2] $h = 2\,\mathrm{s}$ bei einem Diskretisierungsschritt von $\Delta t = 200\,\mathrm{ms}$ vorgeschlagen. Dies ergibt $N = 10$ Iterationsschritte für den GPDP-Algorithmus. Diese Werte wurden für die meisten der folgenden Versuche beibehalten. In Abschnitt 4.3.2 wurde aber auch untersucht, welche Auswirkungen Änderungen dieser Parameter haben.

Als Kovarianzfunktion für GPDP wurde der Matérn-Kernel (Formel 2.4) verwendet. Dieser lieferte etwas „bessere“ Ergebnisse als der Squared-Exponential-Kernel (Formel 2.3). Wie die Ergebnisse bewertet wurden wird dabei weiter unten erklärt. Im Gegensatz hierzu war für das Lernen der Dynamik des Systems der Squared-Exponential-Kernel besser geeignet und wurde deshalb bei den beschriebenen Versuchen verwendet.

Die Kosten in Zeile 5 von Algorithmus 4 waren mit normalverteiltem Rauschen mit Mittelwert 0 und Standardabweichung 0.001 behaftet. Eine Außnahme bildet Abschnitt 4.3.3, wo überprüft wurde, inwiefern sich stärker verrauschte Kosten auf den Algorithmus auswirken.

Auch die Trainingsdaten für das Lernen der Dynamik waren unabhängigem, normalverteilten Rauschen mit Mittelwert 0 und Standardabweichung 0.001 unterworfen.

Als Kostenfunktion wird in [2]

$$g(\mathbf{x}, u) = 1 - \exp(-\alpha\varphi^2 - \beta\dot{\varphi}^2 - \gamma u^2)$$

mit $\mathbf{x} = (\varphi, \dot{\varphi})$ vorgeschlagen, was hier mit den Werten $\alpha = 1$, $\beta = \frac{2}{5}\,\mathrm{s}^2$ sowie $\gamma = \frac{3}{10}\,\mathrm{N}^{-2}\mathrm{m}^{-2}$ für alle folgenden Versuche übernommen wurde. Diese Werte liefern gute Ergebnisse, allerdings sind auch andere Varianten zielführend. Des Weiteren wurde $g_{term}(\mathbf{x}) = g(\mathbf{x}, 0)$ gesetzt.

Mit dem oben beschriebenen Vorgehen unterliegt somit jeder Durchlauf des Algorithmus den folgenden zufälligen Einflüssen:

- Die Position der Trainingspunkte für GPDP ist zufällig.
- Die Kostenfunktion ist mit zufälligem Rauschen behaftet.

Im Fall von gelernter Dynamik tritt noch hinzu:

- Die Position der Trainingspunkte für das Lernen der Dynamik ist zufällig.
- Die Trainingsdaten für das Lernen der Dynamik sind verrauscht.

Jeder in den folgenden Abschnitten beschriebene Versuch wurde 30 mal durchgeführt. Bei der Auswertung dieser Durchläufe wurden drei mögliche Ausgänge unterschieden:
Erstens kann das Hochschwingen und Balancieren des Pendels in der minimal möglichen Anzahl an Schwüngen gelingen. Dies wird als „erfolgreich“ bezeichnet. Der genaue Verlauf der Trajektorie ist dabei nicht relevant. Der Anteil der Versuche, die „erfolgreich“ waren, wird als Erfolgsrate bezeichnet.
Zweitens kann das Hochschwingen und Balancieren gelingen, jedoch muss öfter „Schwung geholt“ werden, als eigentlich nötig ist. Dies wird als „teilweise erfolgreich“ bezeichnet. Werden bei der Beschreibung eines Versuchs „teilweise erfolgreiche“ Durchläufe nicht erwähnt, so traten sie nicht auf. Auch hier wird der genaue Verlauf der Trajektorie ignoriert.
Drittens kann das Hochschwingen oder Balancieren nicht gelingen.

4.3 GPDP mit bekannter Dynamik

In diesem Abschnitt wird untersucht, wie gut sich das oben beschriebene System mit Hilfe von GPDP steuern lässt, wenn die Dynamik des Pendels als bekannt vorausgesetzt wird.

4.3.1 Abhängigkeit von der Anzahl der Trainingspunkte

Hier soll überprüft werden, wie sich die Anzahl der Trainingspunkte auf die Leistung des Verfahrens auswirkt. Dafür wurden Versuche mit verschiedenen Kombinationen aus Parameterwerten für p und q durchgeführt. Es wurde untersucht, wie oft das Verfahren erfolgreich ist, d.h. bei wie vielen Versuchen das Pendel hoch geschwungen und balanciert werden kann. Das Ergebnis zeigt die folgende Tabelle:

p	q	Anzahl Trainingspunkte	Erfolgsrate
13	15	230	40%
16	18	323	63%
19	21	434	87%
22	24	563	93%

Tabelle 4.1: Erfolgsrate in Abhängigkeit von der Anzahl der Trainingspunkte im Phasenraum.

Bereits mit ca. 200 Trainingspunkten kann das Problem erfolgreich gelöst werden. Allerdings ist es in diesem Fall evtl. nötig, den Algorithmus mehrere Male auszuführen. Mit steigender Anzahl von Trainingspunkten steigt die Erfolgsrate schnell an.
Die Trajektorien der erfolgreichen Versuche unterscheiden sich allerdings nicht wesentlich. Als Beispiel sind in den Abbildungen 4.3, 4.4 und 4.5 die Verläufe von Auslenkung, Winkelgeschwindigkeit und Steuerung jeweils für $(p, q) = (13, 15)$ und für $(p, q) = (22, 24)$ aufgeführt.

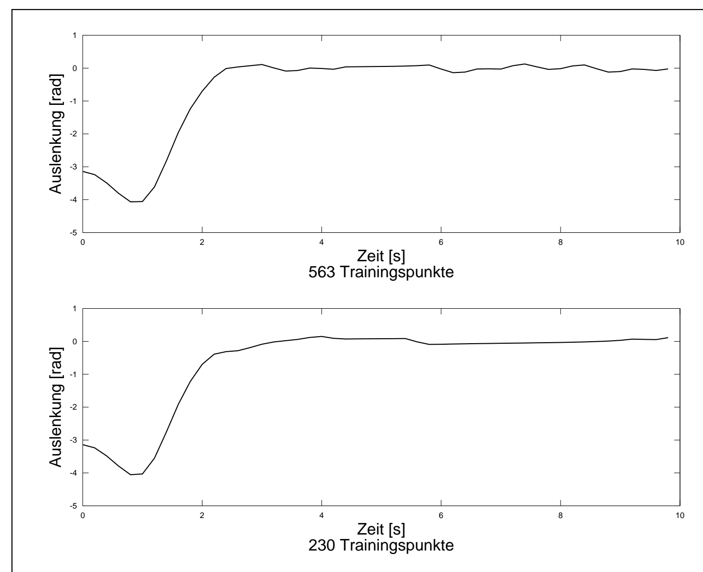

Abbildung 4.3: Verlauf der Auslenkung des Pendels über die Zeit. Das Pendel wird aus der Ruheposition $\varphi = -\pi$, $\dot{\varphi} = 0$ hochgeschwungen.
Oben: $(p, q) = (22, 24)$. Unten: $(p, q) = (13, 15)$.

Mit mehr Trainingspunkten kann das Pendel etwas früher hochgeschwungen werden. Eine Auslenkung von $\varphi = 0$ wird mit 563 Trainingspunkten nach ca. 2 s erreicht. Bei 230 Trainingspunkten ist dies erst nach ca. 3 s der Fall. Abgesehen hiervon sind die Trajektorien sehr ähnlich.

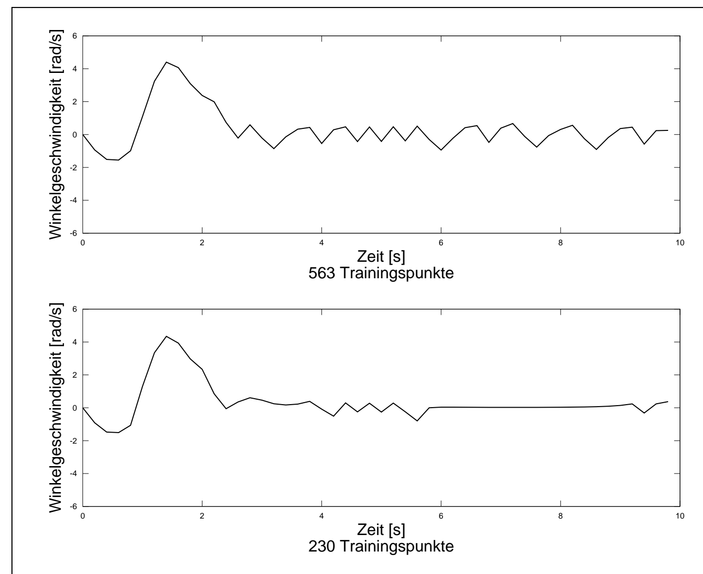

Abbildung 4.4: Verlauf der Winkelgeschwindigkeit über die Zeit. Das Pendel wird aus der Ruheposition $\varphi = -\pi$, $\dot{\varphi} = 0$ hochgeschwungen.
Oben: $(p, q) = (22, 24)$. Unten: $(p, q) = (13, 15)$.

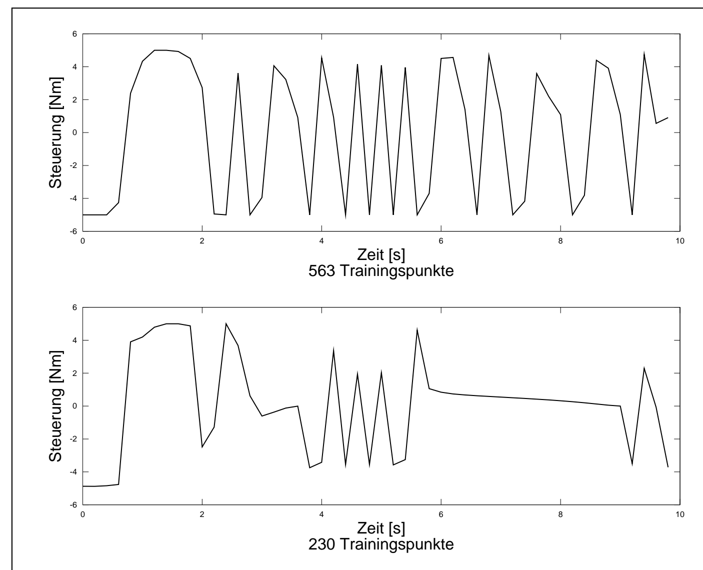

Abbildung 4.5: Verlauf der Steuerung über die Zeit. Das Pendel wird aus der Ruheposition $\varphi = -\pi$, $\dot{\varphi} = 0$ hochgeschwungen.
Oben: $(p, q) = (22, 24)$. Unten: $(p, q) = (13, 15)$.

Die Graphen für die Winkelgeschwindigkeit sind sich für die verschiedenen Anzahlen an Trainingspunkten sehr ähnlich. Für die Steuerungen gibt es zwar Unterschiede, diese haben aber keine größeren Auswirkungen auf die Leistung des Verfahrens.

Anmerkung: Der Graph der Steuerung ist so, wie er hier gezeigt ist, streng genommen nicht korrekt dargestellt. Tatsächlich ist die Steuerung stückweise konstant, wie in Abschnitt 4.1 beschrieben. Sie kann nur alle $\Delta t = 0.2\,\mathrm{s}$ geändert werden. Aus Gründen der Lesbarkeit wurde hier und im Rest der Arbeit dennoch die Darstellung als stetige Kurve gewählt.

4.3.2 Abhängigkeit vom Optimierungshorizont

Da mit größerem Optimierungshorizont h (bei gleichbleibendem Diskretisierungsintervall Δt) der Rechenaufwand für den GPDP-Algorithmus linear steigt, stellt sich die Frage, wie klein h gewählt werden kann, und ob größere Werte von h die Erfolgsrate erhöhen.
Um dies zu untersuchen, wurde folgendermaßen verfahren: Bei konstantem $\Delta t = 0.2\,\mathrm{s}$ wurde der in [1] vorgeschlagene Wert von $N = 10$ mit den Werten $N = 8$, $N = 12$ und $N = 14$ verglichen. Für die Anzahl der Trainingspunkte wurden die Parameterwerte $(p, q) = (16, 18)$ ausgewählt. Es wurden also 323 Trainingspunke für den Phasenraum verwendet. Das Ergebnis zeigt Tabelle 4.2.

Anzahl Iterationsschritte N	Optimierungshorizont h	Erfolgsrate
8	1.6 s	57%
10	2.0 s	63%
12	2.4 s	73%
14	2.8 s	70%

Tabelle 4.2: Erfolgsrate in Abhängigkeit vom Optimierungshorizont.

Es ist zu sehen, dass die Erfolgsrate mit größerem Optimierungshorizont bis zu einem Wert von $N = 12$ ebenfalls ansteigt. Insbesondere ist ein Optimierungshorizont von $N = 10$ offenbar nicht ausreichend groß, um die optimale Erfolgsrate bei den verwendeten Parameterwerten zu erreichen. Die Erfolgsrate sinkt für $N = 14$ gegenüber $N = 12$ leicht ab. Dies ist eventuell zufallsbedingt. Es bleibt aber festzuhalten, dass die Erfolgsrate zumindest nicht weiter ansteigt.

Ähnlich wie oben beschrieben ändert sich die „Qualität" der Lösungen nur geringfügig. Die folgenden Abbildungen zeigen den Verlauf von Auslenkung, Winkelgeschwindigkeit und Steuerung jeweils für die Fälle $N = 8$ und $N = 12$.

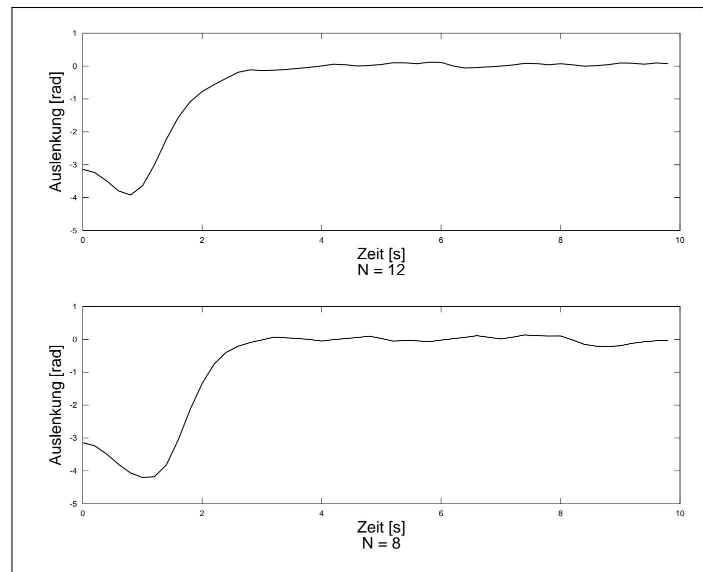

Abbildung 4.6: Verlauf der Auslenkung des Pendels über die Zeit in Abhängigkeit von der Anzahl N der Iterationsschritte.

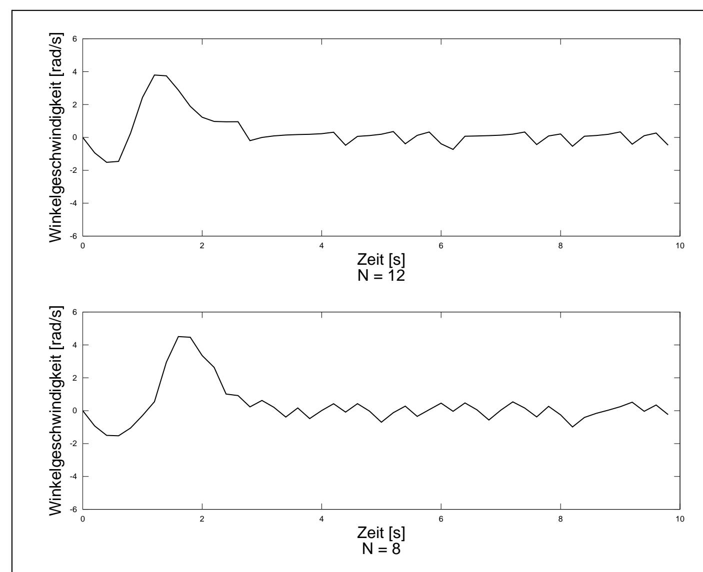

Abbildung 4.7: Verlauf der Winkelgeschwindigkeit des Pendels über die Zeit in Abhängigkeit von der Anzahl N der Iterationsschritte.

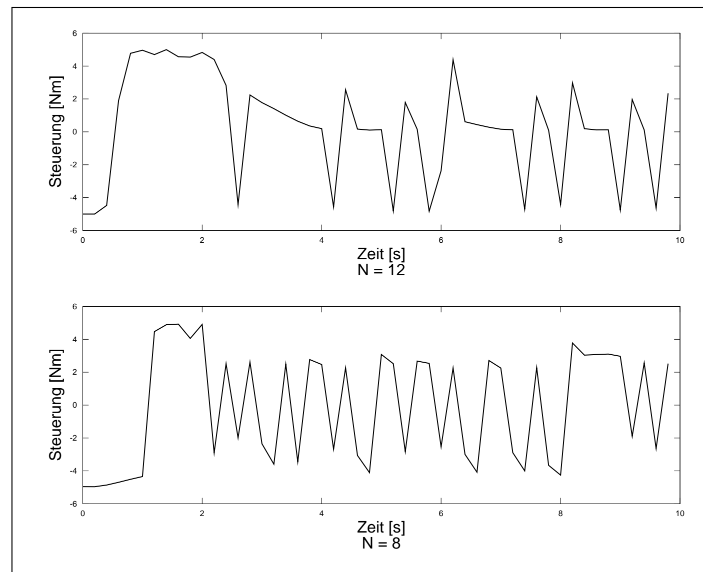

Abbildung 4.8: Verlauf der Steuerung über die Zeit in Abhängigkeit von der Anzahl N der Iterationsschritte.

Die Trajektorien unterscheiden sich kaum. Für $N = 12$ sind zwar die Schwankungen der Winkelgeschwindigkeit und der Steuerung etwas geringer, andererseits kann für $N = 8$ das Pendel etwas früher hochgeschwungen werden. Insgesamt sind die Unterschiede gering.

4.3.3 Einfluss von zufälligen Störungen der Kostenfunktion

Bei allen vorherigen Messungen war die Kostenfunktion, die in Zeile 4 von Algorithmus 4 auftritt, mit zufälligem, normalverteilten Rauschen mit Mittelwert 0 und Standardabweichung 0.001 behaftet.
In diesem Abschnitt soll untersucht werden, welchen Einfluss stärkere Störungen auf den GPDP-Algorithmus haben. Dazu wurden, bei ansonsten identischen Bedingungen, Versuche mit Standardabweichung 0.01 und mit Standardabweichung 0.1 durchgeführt. Hierbei wurde die durch die Parameter $(p, q) = (16, 18)$ charakterisierte Trainingspunktmenge verwendet. Es wurde, wie oben, der Anteil erfolgreicher Versuche bestimmt. Das Ergebnis ist in folgender Tabelle zusammengefasst:

Standardabweichung	Erfolgsrate
0.001	63%
0.01	63%
0.1	29%

Tabelle 4.3: Erfolgsrate in Abhängigkeit von der Standardabweichung der normalverteilten Störungen der Kostenfunktion

Zwischen den Werten 0.001 und 0.01 gibt es keinen Unterschied. Erst bei einer Standardabweichung von 0.1 fällt die Erfolgsrate auf ca. die Hälfte des ursprünglichen Wertes. Allerdings zeigt sich auch hier der selbe Effekt wie in den vorherigen Abschnitten: Die „Qualität" der erfolgreichen Lösungen ändert sich nicht bzw. nur unwesentlich, wenn die Standardabweichung erhöht wird. Lediglich die Erfolgsrate sinkt.

4.3.4 Zusammenfassung von GPDP bei bekannter Dynamik

In diesem Abschnitt wurde deutlich, dass GPDP in der Lage ist das vorliegende Problem zu lösen. Dies ist auch möglich, wenn nur relativ wenige Traininspunkte verwendet werden. In diesem Fall muss das Verfahren jedoch evtl. mehrmals ausgeführt werden, um eine Lösung zu finden. Mit mehr Trainingsdaten steigt die Erfolgsrate schnell an. Das Verfahren ist auch in der Lage, mit einer verrauschten Kostenfunktion umzugehen.

4.4 GPDP mit gelernter Dynamik

In diesem Abschnitt wird untersucht, wie sich das Verfahren verhält, wenn die Dynamik des Pendels nicht mehr als bekannt vorausgesetzt wird, sondern (wie in Abschnitt 3.3 beschrieben) aus Beispielen gelernt werden muss.

4.4.1 Abhängigkeit von der Anzahl der Trainingspunkte

Bei der Wahl der Trainingspunkte stellte es sich heraus, dass es für die Steuerung u ausreicht, die drei Werte -5, 0 und 5 zuzulassen, also $c = 3$ in Gleichung 4.2 zu setzen. Dies wurde in den folgenden Versuchen getan.

Zunächst wurden – bei konstanten Werten $(p, q) = (16, 18)$ für die Trainingspunkte des GPDP-Algorithmus – die Werte a und b aus Gleichung 4.1 variiert. Das Ergebnis ist in Tabelle 4.4 zu sehen.

a	b	c	Anzahl Trainingspunkte	Erfolgsrate	Anteil „teilweise erfolgreich“
4	6	3	72	13%	13%
6	8	3	144	47%	0%
8	10	3	240	50%	0%
10	12	3	360	63%	0%
Bekannte Dynamik				63%	0%

Tabelle 4.4: Erfolgsrate in Abhängigkeit von der Anzahl der Trainingspunkte für das Modell der Dynamik.

Man kann beobachten, dass die Erfolgsrate identisch zur Erfolgsrate mit bekannter Dynamik ist, wenn die Anzahl der Trainingspunkte groß genug ist. Reduziert man die Anzahl der Trainingspunkte, fällt die Erfolgsrate ab. Ab einem bestimmten Wert (hier 72 Trainingspunkte) kann man beobachten, dass auch „teilweise erfolgreiche“ Lösungen gefunden werden. Ist die Anzahl groß genug, tritt dieser Effekt nicht auf.
Somit kann man (ausgehend von einer gewissen Mindestanzahl an Trainingspunkten) den oben bereits erwähnten Effekt feststellen: Die „Qualität“ der Lösungen bleibt auch bei sinkender Anzahl an Trainingspunkten gleich. Das Verfahren muss lediglich öfter angewandt werden, um eine Lösung zu finden. Dies wird von den Abbildungen 4.9, 4.10 und 4.11 belegt, wo eine erfolgreiche Lösung mit wenigen Trainingspunkten mit einer erfolgreichen Lösung bei bekannter Dynamik verglichen wird. Hier wurden die Parameterwerte $(a, b) = (6, 8)$ verwendet.

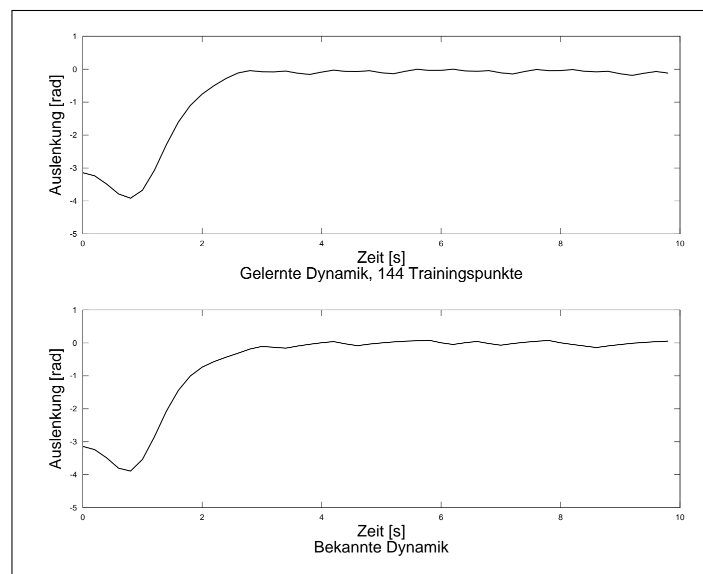

Abbildung 4.9: Verlauf der Auslenkung des Pendels über die Zeit. In Abhängigkeit davon, ob GPDP mit einem bekannten oder gelernten Dynamikmodell angewandt wurde.

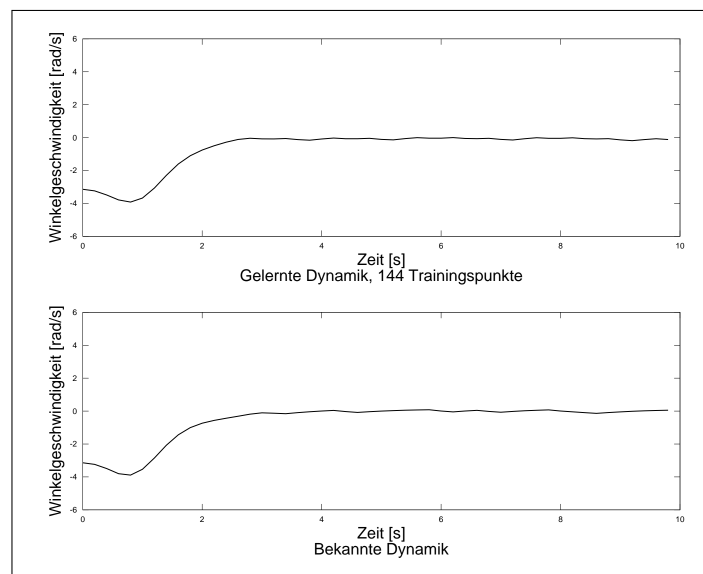

Abbildung 4.10: Verlauf der Winkelgeschwindigkeit des Pendels über die Zeit. In Abhängigkeit davon, ob GPDP mit einem bekannten oder gelernten Dynamikmodell angewandt wurde.

Die Kurven unterscheiden sich nur geringfügig.

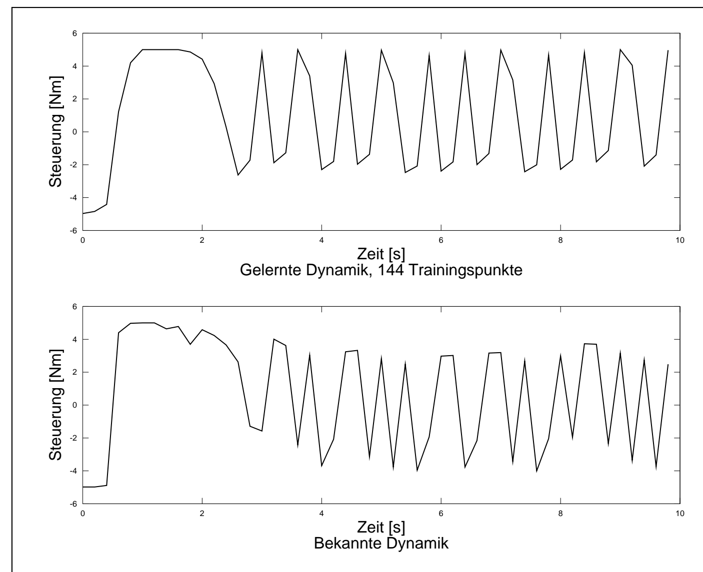

Abbildung 4.11: Verlauf der Steuerung über die Zeit. In Abhängigkeit davon, ob GPDP mit einem bekannten oder gelernten Dynamikmodell angewandt wurde.

Als nächstes sollte die folgende Fragestellung untersucht werden: Angenommen, man nutzt für das Lernen der Dynamik nur relativ wenige Trainingspunkte. Das erhaltene Modell liefert folglich eher ungenaue Vorhersagen. Kann man, durch Erhöhung der Anzahl der Trainingspunkte für den GPDP-Algorithmus, diesen Nachteil kompensieren?
Um dies festzustellen, wurden die Parameter, welche die Anzahl der Trainingspunkte für das Lernen der Dynamik bestimmen, auf $(a, b, c) = (6, 8, 3)$ festgelegt. Anschließend wurde die Anzahl der Trainingspunkte für GPDP variiert. Das Resultat zeigt folgende Tabelle:

Dynamikmodell	p	q	Anzahl Trainingspunkte	Erfolgsrate
Gelernte Dynamik	16	18	323	47%
	19	21	434	63%
	22	24	563	63%
Bekannte Dynamik	16	18	323	63%
	19	21	434	87%
	22	24	563	93%

Tabelle 4.5: Erfolgsrate in Abhängigkeit von der Anzahl der Trainingspunkte für GPDP. Jeweils bei bekannter und bei gelernter Dynamik.

Während bei bekannter Dynamik mit steigender Anzahl der Trainingspunkte auch die Erfolgsrate ansteigt, gilt dies bei gelernter Dynamik nur eingeschränkt. Statt-

dessen wird ein Punkte erreicht (hier ab 434 Trainingspunkten), ab dem die Erfolgsrate nicht mehr ansteigt. Möglicherweise beschränkt das unzureichende Modell der Dynamik die Leistung des Verfahrens.

4.5 Allgemeine Bemerkungen und Vergleich mit Deisenroth et al.

In diesem Abschnitt werden einige weitere, allgemeine Resultate präsentiert und die erzielten Ergebnisse mit den Ergebnissen von Deisenroth, Rasmussen und Peters aus [1] und [2] verglichen.

Hierbei fällt zunächst auf, dass die oben gezeigten Steuerungen bei den erfolgreichen Versuchen wesentlich weniger „glatt“ ausfallen, als dies in [1] und [2] der Fall ist. Auch die Graphen der Auslenkung und der Winkelgeschwindigkeit schwanken stärker. Es konnte nicht sicher festgestellt werden, woran dies liegt. Allerdings gibt es in der hier verwendeten Implementierung von GPDP einen Unterschied zu der Version von Deisenroth et al., was die verschiedenen Ergebnisse erklären könnte. Hierauf wird in Kapitel 5 näher eingegangen.

Es ist nützlich, die Steuerfunktionen direkt miteinander zu vergleichen. Dazu wurde in Abbildung 4.12 an jedem Punkt des relevanten Teil des Phasenraums die von GPDP gefundene Steuerung farblich dargestellt. Der Algorithmus wurde hier mit bekannter Dynamik und den Parameterwerten $(p, q) = (19, 21)$ ausgeführt. Die Anzahl der Trainingspunkte stimmt also in etwa mit der in [1] überein.

Es sind deutliche Unterschiede zu der entsprechenden Abbildung in [1] zu erkennen. Die grundlegenden Formen stimmen zwar überein, jedoch gibt es größere Unterschiede an den Rändern des betrachteten Bereiches des Phasenraums.
Auch hier lassen sich die Unterschiede möglicherweise durch die in Kapitel 5 angesprochenen Differenzen in der Implementierung erklären.

Diese Beobachtungen bleiben auch bestehen, wenn die Dynamik des Pendels gelernt werden muss. Abbildung 4.13 zeigt die Policy für $(p, q) = (19, 21)$ und $(a, b, c) = (6, 8, 3)$. Vergleicht man das Ergebnis mit der Darstellung aus [2], fallen wieder die Unterschiede an den Rändern des betrachteten Bereichs auf.

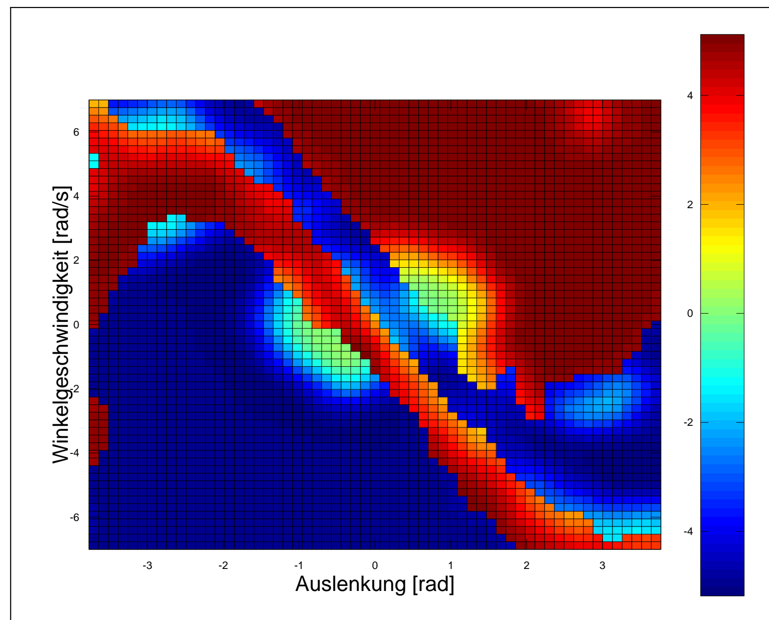

Abbildung 4.12: Von GPDP gelernte Policy. Bekanntes Dynamikmodell.

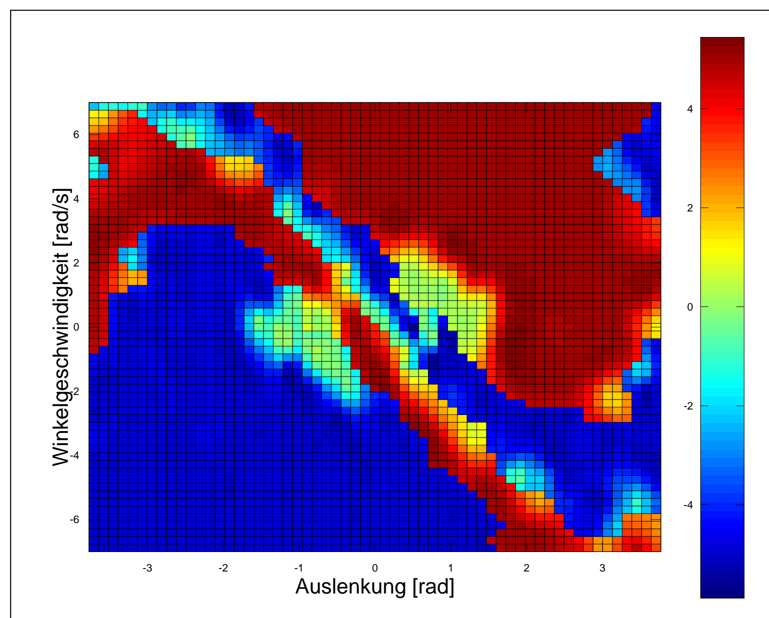

Abbildung 4.13: Von GPDP gelernte Policy. Gelerntes Dynamikmodell.

Kapitel 5

Implementierung

In diesem Kapitel wird kurz darauf eingegangen, wie die in dieser Arbeit benutzten Verfahren implementiert wurden.
Genutzt wurde die frei verfügbare Programmiersprache *Octave*[1] in der Version 3.2.2. Benötigt wurde außerdem eine Implementierung der Regression mit Gaußschen Prozessen, wie sie in Kapitel 2 beschrieben ist. Zuerst wurde versucht, diese Funktionalität selbstständig zu realisieren. Allerdings stellte sich eine in *Octave* geschriebene Implementierung als zu langsam heraus. Deshalb wurde auf das frei verfügbare Programmpacket *OctGPR*[2] in der Version 1.1.5 zurückgegriffen, welches in *FORTRAN* geschrieben ist und entsprechend schneller lief. Dies war die einzige Implementierung für *Octave*, die ausfindig gemacht werden konnte.

OctGPR stellt alle in der vorliegenden Arbeit benötigten Kovarianzfunktionen zur Verfügung. Dabei können die Hyperparameter der Kernel mit Hilfe von Maximum-Likelihood bestimmt werden. Beim Maximieren der Log-Likelihood-Funktion muss entschieden werden, wie „gründlich“ dies geschehen soll. Insbesondere muss festgelegt werden, wie viele unterschiedliche Startpositionen für den Gradientenabstieg probiert werden.
Für die verschiedenen Regressionen, die für GPDP notwendig sind, wurde dies auf unterschiedliche Weise gehandhabt. So wurden bei den GP-Modellen für die Q-Funktion nur wenige Startwerte ausprobiert, da diese Regression sehr häufig durchgeführt werden muss. Im Gegensatz hierzu konnten beim Auffinden der Hyperparameter für das Modell der Dynamik des Pendels mehr Durchläufe des Gradientenabstiegs durchgeführt werden, da dies nur ein einziges Mal, vor Beginn des GPDP-Algorithmus, geschieht.

[1] http://www.gnu.org/software/octave/
[2] http://octave.sourceforge.net/octgpr/index.html

Weiter musste entschieden werden, auf welche Weise die Minimierung der Q-Funktion in Zeile 7 von Algorithmus 4 erfolgen sollte. Hier wurde, orientiert an der Implementierung in [2], die Funktion auf einer großen Anzahl gleichmäßig auf dem relevanten Intervall $[-5, 5]$ verteilter Punkte ausgewertet. Es zeigte sich, dass diese Lösung schneller ist, als ein nichtlineares Optimierungsverfahren, wie es *Octave* zur Verfügung stellt, zu nutzen.

Das wesentliche Problem bei der Implementierung war folgendes:
Im GPDP-Algorithmus, so wie er von Deisenroth, Rasmussen und Peters beschrieben wird, ist der Mittelwert des GP-Priors der V- und Q-Funktionen auf einen anderen Wert als 0 gesetzt. Dies konnte in der vorliegenden Arbeit nicht umgesetzt werden, da wie oben beschrieben das Programmpacket *OctGPR* genutzt werden musste, dieses aber keine anderen Mittelwerte als konstant 0 für den Prior unterstützt.
Dieser Unterschied in der Implementierung kann ein Grund für die oben angesprochenen Unterschiede in den von GPDP gefundenen Lösungen sein.

Für die Auswertung des Erwartungswertes in Zeile 4 von Algorithmus 4 (siehe S. 14) für den Fall von gelernter Dynamik wurde die Approximation durch Gleichung 3.1 in Abschnitt 3.3 verwendet. Dafür gab es verschiedene Gründe. Einerseits sollte untersucht werden, ob mit dieser Approximation auch bei wenigen Trainingsdaten für das Dynamikmodell das System erfolgreich gesteuert werden kann. Da Deisenroth, Rasmussen und Peters stattdessen Bayesian Monte Carlo [7] verwenden, um das vorkommende Integral zu bestimmen, konnte hier eine Alternative zu dieser Methode überprüft werden.
Es sei bemerkt, dass diese Entscheidung nicht zu den oben angesprochenen Unterschieden bei den Resultaten geführt haben sollte, da diese Unterschiede bereits auftreten, wenn GPDP mit bekannter Dynamik angewandt wird.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurde das Verfahren GPDP zur (optimalen) Steuerung von dynamischen Systemen, welches von Deisenroth, Rasmussen und Peters entwickelt wurde, implementiert und anhand eines einfachen Beispiels überprüft. Dieses Verfahren erweitert den klassischen Algorithmus der Dynamischen Programmierung, um die äußert aufwendige Diskretisierung des Phasenraums und der Menge der Steuerungen zu beschleunigen.

Es wurde untersucht, welche Einflüsse die verschiedenen Parameter, von denen GPDP abhängt, auf die Leistung des Verfahrens haben. Dabei wurde unter anderem der Einfluss der Trainingsdaten und des vorhandenen Rauschens analysiert.
Außerdem wurde betrachtet, wie sich die Leistung des Verfahrens ändert, wenn die Dynamik des zu steuernden Systems nicht mehr bekannt (d.h. als Differentialgleichung gegeben) ist. In diesem Fall muss erst ein Modell der Dynamik aus Beispielen gelernt werden, bevor GPDP angewandt werden kann.

Als Ergebnis der durchgeführten Versuche kann man festhalten, dass GPDP in der Lage ist, das beschriebende Problem zu lösen. Dabei kann die Diskretisierung des Raumes in sehr kleine Abschnitte, wie sie für DP notwendig ist, durch eine sehr grobe Diskretisierung ersetzt werden. Diese Eigenschaft bleibt auch bestehen, wenn die Dynamik des Systems aus Beispielen gelernt werden muss.

Bei der Untersuchung des Einflusses der verschiedenen Parameter konnte eine interessante Beobachtung gemacht werden:
Solange bestimmte kritische Werte dieser Parameter nicht über- oder unterschritten werden, kann GPDP eine gute oder „nahezu optimale“ Lösung finden, falls man bereit ist, den Algorithmus evlt. mehrmals auszuführen. Dabei kann zum Beispiel die Anzahl der Trainingspunkte relativ klein oder die Stärke des Rauschens relativ groß sein, ohne das die „Qualität“ der gefundenen Lösungen darunter leidet.

Die Chance, die gesuchte Lösung überhaupt zu finden, wird aber entsprechend kleiner. Folglich wird das Verfahren oft mehrmals, mit verschiedenen Konfigurationen der Parameter, angewandt werden müssen, bis eine Lösung gefunden wird.

Im Vergleich zu den von Deisenroth, Rasmussen und Peters veröffentlichten Resultaten sind die von der vorliegenden Implementierung gefundenen Lösungen als schlechter einzuschätzen. Der Grund hierfür konnte nicht endgültig geklärt werden. Der Unterschied in den Mittelwerten der GP-Prior, wie er in Kapitel 5 angesprochen wurde, könnte eine Erklärung liefern.

Andererseits konnte festgestellt werden, dass trotz der Verwendung einer Approximation, die nur für den Fall vieler Trainingsdatenpunkte für das Dynamikmodell gilt, das Verfahren auch mit relativ wenigen dieser Trainingspunkte die gesuchte Lösung finden kann.

Ausgehend von den in dieser Arbeit gefundenen Resultaten kann man einige weiterführende Fragen stellen.

Erstens wäre es interessant zu untersuchen, ob die beschriebenen Unterschiede zu den Resultaten von Deisenroth, Rasmussen und Peters tatsächlich durch die andere Wahl des GP-Priors entstehen. Hierzu müsste eine Version der Regression mit Gaußschen Prozessen implementiert werden, die hinreichend schnell ist und außerdem die Nutzung von GP-Priorn mit Mittelwerten ungleich 0 erlaubt.

Zweitens bleibt zu untersuchen, wie GPDP mit Systemen mit einer höheren Anzahl an Freiheitsgraden umgehen kann. Diese Frage stellt sich insbesondere, weil der Aufwand für die Regression mit Gaußschen Prozessen kubisch in der Anzahl der Trainingsdaten ist.

Literaturverzeichnis

[1] Deisenroth, Marc P., Jan Peters und Carl E. Rasmussen.: *Approximate Dynamic Programming with Gaussian Processes.* Proceedings of the 2008 American Control Conference, June 2008. 1, 5, 13, 14, 15, 16, 17, 20, 22, 26, 33

[2] Deisenroth, Marc P., Carl E. Rasmussen und Jan Peters: *Model-Based Reinforcement Learning with Continuous States and Actions.* In: *Proceedings of the 16th European Symposium on Artificial Neural Networks (ESANN 2008)*, Seiten 19–24, Bruges, Belgium, April 2008. 1, 13, 16, 17, 20, 22, 33, 36

[3] Bertsekas, Dimitri P.: *Dynamic Programming and Optimal Control, Volume 1.* Athena Scientific, Belmont, Mass., USA, 1995. 3, 4

[4] Rasmussen, Carl E. und C. K. I. Williams: *Gaussian Processes for Machine Learning.* MIT Press, 2006. 6, 7, 8, 9

[5] Williams, C. K. I.: *Prediction With Gaussian Processes: From Linear Regression To Linear Prediction And Beyond.* In: *Learning and Inference in Graphical Models*, Seiten 599–621. Kluwer, 1997. 6

[6] Bishop, C. M.: *Pattern Recognition and Machine Learning.* Springer, 2006. 7, 8, 9, 11

[7] Rasmussen, Carl E. und Zoubin Ghahramani: *Bayesian Monte Carlo.* Advances in Neural Information Processing Systems, Seiten 489–496, 2003. 36

Abbildungsverzeichnis

2.1 Zufällig gezogene Funktionen aus einem GP-Prior mit verschiedenen Kovarianzfunktionen. 10

2.2 Beispiel zur Regression mit Gaußschen Prozessen.
Dünne Kurve: Zu approximierende Funktion $h(x)$.
Durch Kreise markierte Punkte: Trainingspunkte.
Dicke Kurve: Durch Regression bestimmte Funktion (Mittelwert der Vorhersage).
Schattierter Bereich: $\pm$ 1 Standardabweichung um den Mittelwert. . 12

4.1 Skizze des Pendels sowie Start- und Zielposition. 18

4.2 Verlauf der Auslenkung des Pendels über die Zeit bei verschiedenen Werten für dt. Das Pendel wird aus der Position $\varphi = \pi/2$, $\dot{\varphi} = 0$ fallen gelassen. Die Steuerung ist konstant gleich 0. 19

4.3 Verlauf der Auslenkung des Pendels über die Zeit. Das Pendel wird aus der Ruheposition $\varphi = -\pi$, $\dot{\varphi} = 0$ hochgeschwungen.
Oben: $(p, q) = (22, 24)$. Unten: $(p, q) = (13, 15)$. 24

4.4 Verlauf der Winkelgeschwindigkeit über die Zeit. Das Pendel wird aus der Ruheposition $\varphi = -\pi$, $\dot{\varphi} = 0$ hochgeschwungen.
Oben: $(p, q) = (22, 24)$. Unten: $(p, q) = (13, 15)$. 25

4.5 Verlauf der Steuerung über die Zeit. Das Pendel wird aus der Ruheposition $\varphi = -\pi$, $\dot{\varphi} = 0$ hochgeschwungen.
Oben: $(p, q) = (22, 24)$. Unten: $(p, q) = (13, 15)$. 25

4.6 Verlauf der Auslenkung des Pendels über die Zeit in Abhängigkeit von der Anzahl N der Iterationsschritte. 27

4.7 Verlauf der Winkelgeschwindigkeit des Pendels über die Zeit in Abhängigkeit von der Anzahl N der Iterationsschritte. 27

4.8 Verlauf der Steuerung über die Zeit in Abhängigkeit von der Anzahl N der Iterationsschritte. 28

4.9 Verlauf der Auslenkung des Pendels über die Zeit. In Abhängigkeit davon, ob GPDP mit einem bekannten oder gelernten Dynamikmodell angewandt wurde. 31

4.10 Verlauf der Winkelgeschwindigkeit des Pendels über die Zeit. In Abhängigkeit davon, ob GPDP mit einem bekannten oder gelernten Dynamikmodell angewandt wurde. 31
4.11 Verlauf der Steuerung über die Zeit. In Abhängigkeit davon, ob GPDP mit einem bekannten oder gelernten Dynamikmodell angewandt wurde. 32
4.12 Von GPDP gelernte Policy. Bekanntes Dynamikmodell. 34
4.13 Von GPDP gelernte Policy. Gelerntes Dynamikmodell. 34

Tabellenverzeichnis

4.1 Erfolgsrate in Abhängigkeit von der Anzahl der Trainingspunkte im Phasenraum. 24
4.2 Erfolgsrate in Abhängigkeit vom Optimierungshorizont. 26
4.3 Erfolgsrate in Abhängigkeit von der Standardabweichung der normalverteilten Störungen der Kostenfunktion 29
4.4 Erfolgsrate in Abhängigkeit von der Anzahl der Trainingspunkte für das Modell der Dynamik. 30
4.5 Erfolgsrate in Abhängigkeit von der Anzahl der Trainingspunkte für GPDP. Jeweils bei bekannter und bei gelernter Dynamik. 32

Anhang A

Quellcode

Dieser Anhang enthält die wichtigsten Funktionen der Implementierung.

Das Lernen der Dynamik des Systems:

```
function model = learn_dynamics (f, X, U)
  n = rows(X);
  m = rows(U);
  p = columns(X);
  q = columns(U);
  Y = zeros(n*m, p+q);     %Ein Eingabevektor fuer jede Kombination aus Zustand und Steuerung
  T = zeros(n*m, p);     %Ein Targetvektor fuer jede Kombination aus Zustand und Steuerung
  for i = 1:n
    xi = X(i,:);
    for j = 1:m
      uj = U(j,:);
      Y( (i-1)*m + j, : ) = [xi, uj];
      T( (i-1)*m + j, : ) = f(xi, uj) + normrnd([0 0], [0.001 0.001], 1, p);
    end
  end
  GPs = train_gps(Y, T);
  model = @(x,u) model_function(GPs, x, u);
end

function GPs = train_gps (Y, T)
  nm = rows(Y);
  p = columns(T);
  GPs = cell(p, 1);
  for i = 1:p
    GPs{i} = gp_train(Y, T(:,i), "gau", 1 ./ std(Y), 2);
  end
end
```

```
29 function [y, sigma] = model_function (GPs, x, U)
     m = rows(U);
31   p = rows(GPs);
     y = zeros(m, p);
33   xx = repmat(x, m, 1);
     if (nargout() == 1)
35     for i = 1:p
         y(:,i) = gp_mean(GPs{i}, [xx, U]);
37     end
     elseif (nargout() == 2)
39     for i = 1:p
         [y(:,i), sigma(:,i)] = gp_pred(GPs{i}, [xx, U]);
41     end
     end
43 end
```

Der GPDP-Algorithmus:

```
 1 function policy = gpdp (f, loss, X, U, N)
     cov = "mt3";
 3   mX = rows(X);
     mU = rows(U);
 5   V = zeros(mX, 1);
     V_GP = cell(N+1, 1);
 7   policy = zeros(mX, columns(U));
     Q = zeros(mU, 1);
 9   for i = 1:mX
         V(i) = loss.term(X(i,:));
11   end
     V_GP{N+1} = gp_train(X, V, cov, 1 ./ std(X), 5);
13   theta = V_GP{N+1}.theta;
     for k = N:-1:1
15     Q_theta = 1 ./ std(U);
       for i = 1:mX
17       xi = X(i,:);
         F = f(xi, U);    %Bei unbekannter Dynamik ist dies die
             Vorhersage
19       E = gp_pred(V_GP{k+1}, F);
         Q = loss.g(xi, U) .+ loss.gamma * E .+ normrnd(0, 0.001, mU, 1)
             ;
21       Q_GP = gp_train(U, Q, cov, [Q_theta; 1 ./ std(U)]);
         Q_theta = Q_GP.theta;
23       [policy(i,:), V(i)] = minimize(Q_GP, U);
       end
25     V_GP{k} = gp_train(X, V, cov, [theta; 1 ./ std(X)], 5);
       theta = V_GP{k}.theta;
27   end
   end
29
```

```
function [x, y] = minimize (GP, U)
  Test = linspace(-5, 5, 10001)';    %Anregung aus der Implementierung von Deisenroth et al.
  Result = gp_mean(GP, Test);
  [y, i] = min(Result);
  x = Test(i,:);
end
```

Verallgemeinern der Policy auf den gesamten Raum:

```
function [ctrl, GP_class] = optimal_control (f, loss, N, X_ctl, U_ctl, X_dyn, U_dyn)
  if(nargin() == 7)
    dynamics_model = learn_dynamics(f, X_dyn, U_dyn);
  elseif(nargin() == 5)
    dynamics_model = f;
  else
    error("Invalid number of arguments");
  end
  policy = gpdp(dynamics_model, loss, X_ctl, U_ctl, N);
  [GP_pos, GP_neg, GP_class] = create_gps(X_ctl, policy);
  ctrl = @(x) make_control_func(x, GP_pos, GP_neg, GP_class);
end

function [GP_pos, GP_neg, GP_class] = create_gps (X, policy)
  n = rows(X);
  t = zeros(n, 1);
  pos_i = neg_i = 1;
  for i = 1:n
    xi = X(i,:);
    pol_i = policy(i,:);
    if (pol_i >= 0)
      pos_X(pos_i,:) = xi;
      pos_t(pos_i,1) = pol_i;
      t(i) =  1;
      pos_i = pos_i + 1;
    else
      neg_X(neg_i,:) = xi;
      neg_t(neg_i,1) = pol_i;
      t(i) = -1;
      neg_i = neg_i + 1;
    end
  end
  GP_pos = gp_train(pos_X, pos_t, "mt3", 1 ./ std(pos_X), 7);
  GP_neg = gp_train(neg_X, neg_t, "mt3", 1 ./ std(neg_X), 7);
  GP_class = binclass_train(X, t, "mt3", 1 ./ std(X), 7);
end

function y = make_control_func (x, GP_pos, GP_neg, GP_class)
  if (binclass_pred(GP_class, x) >= 0)
```

```
40        y = gp_mean(GP_pos, x);
    else
42        y = gp_mean(GP_neg, x);
    end
44 end
```