Bachelorarbeit

Untersuchung zur Sensordatenfusion durch einen Partikelfilter

Patrizia Weidinger

Hochschule	Technische Hochschule Ingolstadt
Fakultät	Informatik
Studiengang	Informatik
Erstprüfer	Prof. Dr. Jörg Hunsinger
Zweitprüferin	Prof. Dr. Katherine Roegner
Betreuer	Dr. Florian Großmann
Ausgabedatum	18.12.2020
Abgabedatum	25.01.2021

Erklärung

Ich erkläre hiermit, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Schrobenhausen, 25.01.2021

Patrizia Weidinger

Danksagung

Verfasst wurde diese Arbeit in Kooperation mit der MBDA Deutschland für den Abschluss meines Studiums an der Technischen Hochschule Ingolstadt. Wissenschaftlicher Betreuer seitens der Hochschule war Herr Prof. Dr. Jörg Hunsinger, bei dem ich mich an dieser Stelle für die gute Betreuung bedanken möchte.

Auch bei meinem betrieblichen Betreuer Herrn Dr. Floian Großmann, der mir bei Fragen stets mit einem guten Rat zur Seite stand, möchte ich mich an dieser Stelle ganz besonders bedanken. Durch seine inhaltlichen und strukturellen Anregungen konnte ich sehr viel dazulernen.

Ein großer Dank geht außerdem an meine Kollegen in der Abteilung EWG3, die nicht nur fachlich zum Gelingen der Arbeit beigetragen haben, sondern auch durch das angenehme Arbeitsumfeld, das sie geschaffen haben. Besonders bedanken möchte ich mich dabei bei Thomas Friedrich und Philipp von Perponcher, die mir durch ihre Korrekturen zahlreiche Tipps geben konnten.

Einen ganz besonderen Dank möchte ich an Nico Borgsmüller richten, der mir bei meiner Bachelorarbeit durch fachliche Tipps und Korrekturen sehr weitergeholfen hat. Aber nicht nur dafür schulde ich ihm meinen größten Dank, sondern auch für die Unterstützung während meines gesamten Studiums, in dem er mir immer mit einem offenen Ohr zur Seite stand und durch dessen Feedback ich auf fachlicher und persönlicher Ebene sehr viel dazulernen konnte.

Zuletzt möchte ich noch meiner Familie und meinen Freunden danken, die mich beim Schreiben der Arbeit motiviert und unterstützt haben und die während der Verfassung der Arbeit sehr viel Geduld mit mir hatten.

Abstract

Für das Tracking von Objekten gilt der Kalmanfilter nur so lange als optimaler Algorithmus, wie seine restriktiven Annahmen erfüllt sind. Da in der Luftverteidigungsbranche hochagile Ziele getrackt werden, löst der Kalmanfilter das Trackingproblem nicht mehr ideal und die Suche nach alternativen Algorithmen rückt immer mehr in den Fokus. Eine dieser Alternativen ist der Partikelfilter.

Die vorliegende Arbeit stellt eine erste Untersuchung zur Verwendung eines Partikelfilters bei der MBDA Deutschland GmbH dar. Dabei liegt der Fokus auf der Fusion von Sensordaten. Das Ziel der Arbeit ist es, einen Partikelfilter zur Positionsbestimmung von mehreren Objekten in dem Phasenraum eines mathematischen Pendels durch mehrere Sensoren zu implementieren und die Ergebnisse der Datenfusion zu untersuchen.

Aus den theoretischen Grundlagen des Kalman- und des Partikelfilters ergeben sich die drei wichtigsten Bestandteile eines Partikelfilters: Das Bewegungsmodell für die Positionsschätzung der Partikel, das Messmodell für die Bewertung der Schätzung aufgrund der Messung und das Resampling für das Ersetzen schlecht bewerteter Partikel durch gut bewertete. Anschließend werden drei Leitfragen für die Arbeit formuliert, aus welchen sich die Anforderungen für die darauffolgende Implementierung ableiten. Für die Realisierung des Partikelfilters werden zunächst Aufbau und Ablauf des Partikelfilters festgelegt. Im Anschluss daran wird das Bewegungsmodell definiert, welches das Runge-Kutta-Verfahren zur Lösung der Differentialgleichung eines Pendels verwendet. Das Messmodell wird daraufhin so umgesetzt, dass alle Partikel, die sich in der Nähe der Sensormessung befinden, ein Resampling erhalten. Dieses schiebt die Partikel in die Richtung der Messung. Partikel, die lange kein Resampling erhalten haben, werden gelöscht. Die Trackbildung wird im nächsten Schritt mithilfe einer Self Organizing Map realisiert. Zuletzt sorgt die Visualisierung für ein übersichtliches Testen der Ergebnisse. Die Implementierung zeigt, dass durch den Partikelfilter Daten von mehreren Sensoren fusioniert werden können, um Tracks von verschiedenen Objekten zu berechnen. Da diese Tracks in jedem Zeitschritt berechnet werden, können auch „Split Track-“ und „Merge Track Events“ festgestellt werden. Durch die anschließenden Tests kann gezeigt werden, dass die Anzahl der Tracks in 86.3 % der Fälle korrekt bestimmt wird. Außerdem ist die Abweichung der berechneten zur tatsächlichen Position der Objekte in knapp 90 % der Fälle niedriger als 0.15 Meter. Eine Verbesserung der Trackbildung ist durch eine Anpassung der Neuronen der Self Organizing Map möglich. Die Untersuchung der Zusammenhänge zwischen der Sensorgenauigkeit und der Qualität der Ergebnisse zeigt, dass die Ergebnisse besser sind, je höher die Sensorgenauigkeit ist und je länger das Tracking andauert. Eine hohe Sensorgenauigkeit führt allerdings zu einer deutlich längeren Laufzeit. Dies kann durch eine Anpassung der Neuronen der Self Organizing Map verbessert werden. Die maximale Abweichung der berechneten von der tatsächlichen Position liegt bei einer mittleren Sensorgenauigkeit im Schnitt bei 0.19 Metern.

Durch die Arbeit wird gezeigt, dass der Partikelfilter eine gute Alternative für den Kalmanfilter darstellen kann, wenn er auf geeignete Weise zur Anwendung gebracht wird. Die Sensordatenfusion ist erfolgreich gelungen. Auch die Ergebnisse der Trackbildung sind zufriedenstellend und können durch eine Anpassung der Parameter der Self Organizing Map weiter verbessert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Thematischer Hintergrund

Die MBDA Deutschland GmbH ist ein deutsches Verteidigungsunternehmen, welches zu der europäischen MBDA Gruppe gehört. Sie entwickelt und produziert Lenkflugkörper- und Luftverteidigungssysteme, sowie Komponenten und Subsysteme für Luftwaffe, Marine und Heer.[4]

Um eine gute Verteidigung im Luftbereich zu ermöglichen, ist es von großer Bedeutung, sämtliche fliegenden Objekte zu erkennen und deren Position möglichst genau zu ermitteln. Dazu sind nicht nur qualitativ hochwertige Radare nötig, sondern auch Algorithmen, welche die eingehenden Daten verarbeiten, Robustheit gegenüber Messfehlern herstellen und fehlerhafte Daten herausfiltern können. Dieser Prozess der Datenverarbeitung und Positionsschätzung wird Tracking genannt und mithilfe eines Tracking-Algorithmus durchgeführt. Das Ergebnis, das sich daraus ergibt, ist eine Aneinanderreihung von geschätzten Positionen bis zum aktuellen Zeitpunkt und wird als Track bezeichnet. Das möglichst korrekte Berechnen eines Tracks, basierend auf ungenauen und fehlerhaften Messdaten, stellt das Tracking-Problem dar. [10, S.14]

1.2. Motivation und Zielsetzung

Im wehrtechnischen Bereich sind Luftziele besonders schwer zu tracken. Ein Grund dafür ist, dass hauptsächlich Radare als Informationsquellen dienen, welche nach [6, S.1] oft oft die Schwierigkeit haben, dass die Zeitskala der Abtastung nicht sehr viel größer ist als die Zeitskala für signifikante Änderung des Bewegungszustandes durch Manöver. Dies führt dazu, dass es bei Zielen mit hohen Geschwindigkeiten oder großer Manövrierfähigkeit zu signifikanten Informationsverlusten kommt. Der zweite Grund ist, dass die Ziele hochagil, sodass sich ihre Bewegungen nicht mit einer linearen Funktion beschreiben lassen. Somit ist nicht mehr garantiert, dass der Kalmanfilter, welcher in vielen Systemen als Tracking-Algorithmus verwendet wird, das Tracking-Problem ideal löst [10, S.8]. Partikelfilter-Algorithmen können, mithilfe von sogenannten Partikeln, nicht nur die wahrscheinlichste Position des getrackten Objekts angeben, sondern die Wahrscheinlichkeit des Aufenthaltsortes des Objekts auf der gesamten Fläche darstellen. Der Vorteil des Kalman Filters ist sein geringer Rechenbedarf, weshalb er in der Vergangenheit praktisch alternativlos war. Leistungsfähigere Computer ermöglichen heutzutage auch rechenintensive Partikelfilter in Erwägung zu ziehen, womit ihre Vorteile gegenüber dem Kalman zum Tragen kommen. Für die Lösung des Tracking-Problems mithilfe eines Partikelfilters soll diese Bachelorarbeit einen ersten Ansatz liefern. Dabei liegt der Fokus der Arbeit auf dem Aspekt der Sensordatenfusion.

1.3. Aufgabenstellung

Wie gut die Sensordatenfusion mithilfe eines Partikelfilters umsetzbar ist, soll am Beispiel eines mathematischen Pendels im Phasenraum untersucht werden. Dieses Konstrukt wird in Kapitel 3.1 erläutert. Anhand dieses Anwendungsfalles soll erkannt werden, wie gut der Partikelfilter unter idealen Umständen funktioniert. Dies bietet die Grundlage, um den Partikelfilter in einer späteren Untersuchung an einem realen Beispiel anzuwenden. In dieser Arbeit liegt das Ziel darin, auf folgende drei Leitfragen eine Antwort geben zu können:

1. **Ist ein Partikelfilter geeignet, um eine Sensordatenfusion für unterschiedliche Tracks mit verschiedenen Sensoren im Phasenraum zu realisieren?**
   Diese Leitfrage beinhaltet zwei wesentliche Aspekte. Zuerst soll untersucht werden, ob die von mehreren Sensoren gesendeten Daten zusammengeführt werden können. Diese haben verschiedene, jedoch bekannte Messungenauigkeiten. Zweitens soll der Algorithmus so implementiert werden, dass er erkennt, wie viele Tracks vorhanden sind und wo sich diese Tracks befinden. Dabei sollen auch die Ereignisse „Merge Track“ und „Split Track“ berücksichtigt werden, welche zwei verschiedene Tracks zusammenführen oder trennen, die zu dem selben Objekt gehören.
2. **Wie lässt sich eine Trackbildung realisieren?**
   Das Problem der Trackbildung wird erst deutlich, wenn mehrere Tracks vorhanden sind. Es stellt sich hier die Frage, ab wann verschiedene Partikel oder Partikelwolken zwei verschiedenen Tracks zugeordnet werden können und wann zwei verschiedene Messungen als zwei unterschiedliche Tracks und nicht als Messungenauigkeit von einem Track erkannt werden.
3. **Wie hängt die Sensorgenauigkeit mit dem Fusionsergebnis zusammen?**
   Die dritte Frage zielt darauf ab, herauszufinden, wie viel Einfluss die Sensorungenauigkeit auf das Fusionsergebnis hat, beziehungsweise wie gut der Algorithmus Sensorungenauigkeiten ausgleichen kann.

Der Partikelfilter-Algorithmus soll wie eine Blackbox funktionieren, in die nach und nach die Messpunkte der Radare mit einer zugehörigen Fehlerwahrscheinlichkeit eingelesen werden. Am Ende soll eine Trackliste zurückgegeben werden, welche möglichst genau und möglichst korrekt die verschiedenen Tracks anzeigt. Um dies zu erreichen, sollen zunächst in Kapitel 2 die Theorie des Kalmanfilters und darauf aufbauend die theoretischen Grundlagen des Partikelfilters untersucht werden. Basierend darauf wird in Kapitel 3 ein Konzept zur praktischen Umsetzung des Partikelfilters erstellt. Darin wird zunächst die Wahl des Anwendungsbeispiels erklärt. Danach werden die Anforderungen an den Partikelfilter von den Leitfragen abgeleitet. Daraus kann schließlich ein Vorgehen zur Umsetzung entwickelt werden, welches in Kapitel 4 erläutert wird. Zunächst werden der Aufbau und Ablauf des Partikelfilters beschrieben und anschließend die wichtigsten Funktionen, also das Bewegungsmodell, das Messmodell und das Resampling, erklärt. Zuletzt folgen die Trackbildung und die Visualisierung und. In Kapitel 5 werden die Tests vorgestellt, anhand derer die Erfüllung der Anforderungen untersucht wird. Die Ergebnisse davon werden in Kapitel 6 dargestellt. Abschließend erfolgt in Kapitel 7 ein Fazit der Untersuchung und ein Ausblick auf die weitere Analyse des Partikelfilters.

1.4. Abgrenzung

Für diese Bachelorarbeit wurde nicht der gesamte Programmcode selbst erarbeitet. Es waren bereits mehrere Hilfsklassen von Dr. Florian Großmann gegeben. Dazu gehören eine Winkel-, Vektor- und eine Matrizenklasse, sowie eine Klasse, welche Zufallszahlen mit unterschiedlicher Verteilung generiert. Außerdem ist eine Visualisierung durch die Vektorgrafik-Sprache „Asymptote“ gegeben, welche die Fortschritte des Partikelfilters in mehreren aufeinanderfolgenden Bildern festhält. Diese werden in einem PDF-Dokument zusammengefasst. Im Rahmen dieser Arbeit werden die Architektur und die Implementierung des Partikelfilter-Algorithmus erarbeitet. Außerdem werden Messdaten generiert, welche der Bewegung eines mathematischen Pendels entsprechen. Diese werden passend zur Sensorgenauigkeit verrauscht. Auch die Trackbildung wird selbst erarbeitet. Bei der Implementierung ist die Programmiersprache C++ vorgegeben, was sich dadurch ergibt, dass die vorhandenen Hilfsklassen zur numerischen Integration der Differentialgleichungen bereits in C++ implementiert wurden.

2. Der Partikelfilter in der Theorie

Im Bereich des Tracking ist der Kalmanfilter derzeit einer der bekanntesten Bayes'schen Filter-Algorithmen. Die Änderung der Bewegung wird dabei durch eine lineare Funktion beschrieben, wobei sowohl der Systemfehler beim Berechnen des nächsten Schrittes, als auch der Messfehler, zum Beispiel durch Radare, eine gaußverteilte Wahrscheinlichkeit haben.[7, S.74]
Falls die Bewegung nicht durch eine lineare Funktion beschreibbar oder die Fehlerwahrscheinlichkeiten nicht gaußverteilt sind, löst der Kalmanfilter das Trackingproblem nicht mehr optimal. Der Partikelfilter kann sowohl nichtgaußverteilte Fehler, als auch nichtlineare Bewegungsupdates verarbeiten, weswegen er im Bereich des Trackings eine interessante Rolle spielen könnte. Die theoretischen Grundlagen und Zusammenhänge der beiden Filter werden in diesem Kapitel genauer dargestellt.

2.1. Der Kalmanfilter

Der Kalmanfilter oder auch Kalman-Bucy-Filter wurde in den 1960er Jahren von Rudolf Kalman und Richard Bucy unabhängig voneinander entwickelt und gilt nach [11, S.2] als der „am häufigsten verwendete probabilistische Filter". Es ist ein Algorithmus, welcher, verteilt über einen gewissen Zeitraum, ungenaue und teils unvollständige Informationen über einen Datensatz, wie zum Beispiel den Aufenthaltsort eines Objektes, erhält. Diese Informationen werden so verarbeitet, dass der Kalmanfilter bei jedem Zeitschritt eine genauere Schätzung des unbekannten Wertes zurückgeben kann. Aufgrund seiner Echtzeitfähigkeit und Kosteneffizienz findet er hohen Gebrauch in Tracking-Anwendungen. [7, S.74]

Der Kalmanfilter soll den dynamischen Zustand eines Systems zum Zeitpunkt k berechnen. Dafür werden zwei Gleichungen benötigt:

$$x_k = A \cdot x_{k-1} + w_k \tag{2.1}$$

$$z_k = H \cdot x_k + v_k \tag{2.2}$$

Die Gleichung (2.1) beschreibt dabei den Systemzustand zum Zeitpunkt k. Dieser ist eine lineare Zusammensetzung aus der Zustandsübergangsmatrix A, welche als Ausgangspunkt den vorherigen Zustand x_{k-1} hat und einen System-Rausch-Vektor w_k. Die Gleichung (2.2) beschreibt die Korrektur des Systemzustands x_k aufgrund der Messung zum Zeitpunkt k, welche aus der Beobachtungsmatrix H und dem Beobachtungs-Rausch-Vektor v_k zusammengesetzt ist. w_k und v_k sind voneinander unabhängige, gaußverteilte Rauschterme mit den bekannten Kovarianzen Q_k und R_k. Der Filterprozess besteht nun aus zwei Schritten. Zuerst wird der Systemzustand zum Zeitpunkt k geschätzt und anschließend wird die Schätzung mithilfe der nächsten Messung korrigiert. [10, S.4-7] [1, S.175]

Die Vorhersage des Systemzustands wird die A-priori Wahrscheinlichkeitsverteilung $p(x_k|Z_{k-1})$ genannt. Die Notation $p(a|b)$ beschreibt dabei die bedingte Wahrscheinlichkeit, dass a eintritt, wenn b bereits eingetreten ist. Somit beschreibt A-priori Wahrscheinlichkeitsverteilung die Wahrscheinlichkeit, dass ein beliebiger Punkt x_k die korrekte Position ist, basierend darauf, dass alle verfügbaren Messungen Z zum Zeitpunkt $k-1$ eingetreten sind. Durch die Korrektur, basierend auf der neuen Messung zum Zeitpunkt k, entsteht die A-posteriori Wahrscheinlichkeitsverteilung $p(x_k|Z_k)$.

Vorhersage:

Um den nächsten Zeitschritt k vorhersagen zu können, werden die A-posteriori Wahrscheinlichkeitsverteilung des letzten Zeitschritts $p(x_{k-1}|Z_{k-1})$ und die Systemgleichung (2.1), welche die Eintrittswahrscheinlichkeit $p(x_k|x_{k-1})$ besitzt, benötigt. Die vorherigen Systemzustände werden nicht benötigt, da die Systemgleichung einen Markov-Prozess der 1. Ordnung darstellt. Das bedeutet, dass der zukünftige Zustand eines Systems nur durch den aktuellen und nicht durch die vergangenen Zustände beeinflusst wird. Der Übergang von einem Zustand in den nächsten erfolgt bei solchen Prozessen durch die sogenannte Chapman-Kolmogorow-Gleichung.
Dies führt zu folgender Gleichung für das Update des Systemzustands [10, S.5]:

$$p(x_k|Z_{k-1}) = \int p(x_k|x_{k-1}) \cdot p(x_{k-1}|Z_{k-1})\, dx_{k-1} \tag{2.3}$$

Korrektur:

Sobald die Messung zum Zeitschritt k verfügbar ist, wird die A-priori Wahrscheinlichkeitsverteilung mittels der Bayes'schen Regel korrigiert. Somit ergibt sich die folgende A-posteriori Verteilung [10, S.5]:

$$\begin{aligned} p(x_k|Z_k) &= p(x_k|z_k, Z_{k-1}) \\ &= \frac{p(z_k|x_k, Z_{k-1}) \cdot p(x_k|Z_{k-1})}{p(z_k|Z_{k-1})} \\ &= \frac{p(z_k|x_k) \cdot p(x_k|Z_{k-1})}{p(z_k|Z_{k-1})} \end{aligned} \tag{2.4}$$

Die Vereinfachung zum finalen Term ist möglich, da das System nur vom letzten Zeitschritt abhängig ist und somit nicht von allen vergangenen Messungen Z_{k-1}. $p(z_k|x_k)$ ist hier die sogenannte Likelihood-Funktion. Diese setzt die A-priori Verteilung und die neuen Messungen ins Verhältnis, um zu beurteilen, wie präzise die Schätzung des nächsten Zustands vorhergesagt wurde. Sie wird durch das Messmodell aus Gleichung (2.2) definiert. $p(z_k|Z_{k-1})$ ist eine Normalisierungskonstante, welche durch die A-priori Wahrscheinlichkeitsverteilung und die Likelihood-Funktion definiert ist [10, S.5]:

$$p(z_k|Z_{k-1}) = \int p(z_k|x_k) \cdot p(x_k|Z_{k-1})\, dx_k \tag{2.5}$$

Bei dem Ansatz des Kalmanfilters wird davon ausgegangen, dass jede A-posteriori Wahrscheinlichkeit wieder gaußverteilt ist, weswegen sie nur von zwei Parametern abhängig ist, welche aktualisiert werden müssen: Mittelwert und Kovarianz. Somit können die Schritte (2.3) und (2.4) des Kalmanfilter-Algorithmus durch folgende Gleichungen vereinfacht werden, die in jedem Zeitschritt ausgeführt werden. [10, S.8][1, S.175]:

$$p(x_{k-1}|Z_{k-1}) = \mathcal{N}(x_{k-1}; \hat{x}_{k-1|k-1}, P_{k-1|k-1}) \quad (2.6)$$
$$p(x_k|Z_{k-1}) = \mathcal{N}(x_k; \hat{x}_{k|k-1}, P_{k|k-1}) \quad (2.7)$$
$$p(x_k|Z_k) = \mathcal{N}(x_k; \hat{x}_{k|k}, P_{k|k}) \quad (2.8)$$

Dabei ist $\mathcal{N}(x; m, P)$ die Wahrscheinlichkeitsdichte einer Gauß'sche Verteilung mit Argument x, Mittelwert m und Kovarianz P, welche durch folgenden Ausdruck definiert ist:

$$\mathcal{N}(x; m, P) \triangleq |2\pi P|^{-1/2} \exp(-\frac{1}{2}(x-m)^T P^{-1}(x-m)) \quad (2.9)$$

$(x-m)^T$ steht dabei für den transponierten Vektor $(x-m)$. Die Mittelwerte und Kovarianzen müssen in jedem Zeitschritt mit folgenden Formeln aktualisiert werden[10, S.8][1, S.176]. Update Mittelwert $\hat{x}_{k|k-1}$ und Kovarianz $P_{k|k-1}$ aus Gleichung (2.8):

$$\hat{x}_{k|k-1} = F_{k-1}\hat{x}_{k-1|k-1} \quad (2.10)$$
$$P_{k|k-1} = Q_{k-1} + F_{k-1}P_{k-1|k-1}F_{k-1}^T \quad (2.11)$$

Update Mittelwert $\hat{x}_{k|k}$ und Kovarianz $P_{k|k}$ aus Gleichung (2.7):

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \cdot (z_k - H_k\hat{x}_{k|k-1}) \quad (2.12)$$
$$P_{k|k} = [I - K_k H_k]P_{k|k-1} \quad (2.13)$$

Dabei ist I die Einheitsmatrix der Größe $n_x \times n_x$ und K_k die Kalman-Gain-Matrix, welche das Rauschen der Messung mit dem Update der Partikel ins Verhältnis setzt. Sie ist folgendermaßen definiert:

$$K_k = P_{k|k-1}H_k^T S_k^{-1} \quad (2.14)$$

S_k ist dabei die Kovarianz des Rauschterms, welche durch folgenden Ausdruck beschrieben ist:

$$S_k = H_k P_{k|k-1}H_k^T + R_k \quad (2.15)$$

Den Mittelwert und die Kovarianz nach diesem Schema rekursiv zu berechnen ist nach [10, S.8] die „optimale Lösung für das Tracking Problem - wenn die (sehr restriktiven) Annahmen gelten".

2.2. Der Partikelfilter

Neben dem Kalmanfilter, welcher das Filter-Problem im linearen Bereich optimal lösen kann, existieren noch eine ganze Reihe an sogenannten suboptimalen Filtern. Diese nähern die A-posteriori Verteilung lediglich an, können dafür aber auch komplexere und nicht-gaußverteilte A-posteriori Wahrscheinlichkeiten abbilden. Einer dieser Filter ist der Partikelfilter, den es in Ansätzen bereits seit den fünfziger Jahren gibt. Dieser wurde jedoch lange Zeit ignoriert, da damals nicht genug Rechenleistung zur Verfügung stand. Diese Einschränkung wurde mittlerweile behoben, weshalb sich Partikelfilter immer weiter verbreiten. [5, S.54] [10, S.35]

2.2.1. Übersicht

Bevor die Mathematik erklärt wird, die dem Partikelfilter zugrunde liegt, wird eine kurze Übersicht über die Idee und die einzelnen Schritte des Partikelfilters gegeben. Der vollständige Zyklus eines Partikelfilters ist in Abbildung 2.1 zu sehen. Dabei werden $N = 10$ Partikel gleichmäßig mit einem Gewicht von $1/N$ initialisiert. Die Likelihood Funktion, welche auf der Gleichung (2.2) basiert, gibt daraufhin an, wie gut die Position eines Partikels in Abhängigkeit von der neuen Messung bestimmt wurde. Dadurch werden die Partikel neu gewichtet und die A-Posteriori Funktion entsteht. Anschließend findet ein Resampling statt, welches die Partikel auf einer schlechten Position durch Partikel auf einer guten Position ersetzt. Zu Beginn des nächsten Zeitschritts erhalten alle Partikel ein Bewegungsupdate. Dadurch wird die A-Priori Verteilung des nächsten Zeitschritts beschrieben und der Zyklus kann von vorne beginnen.

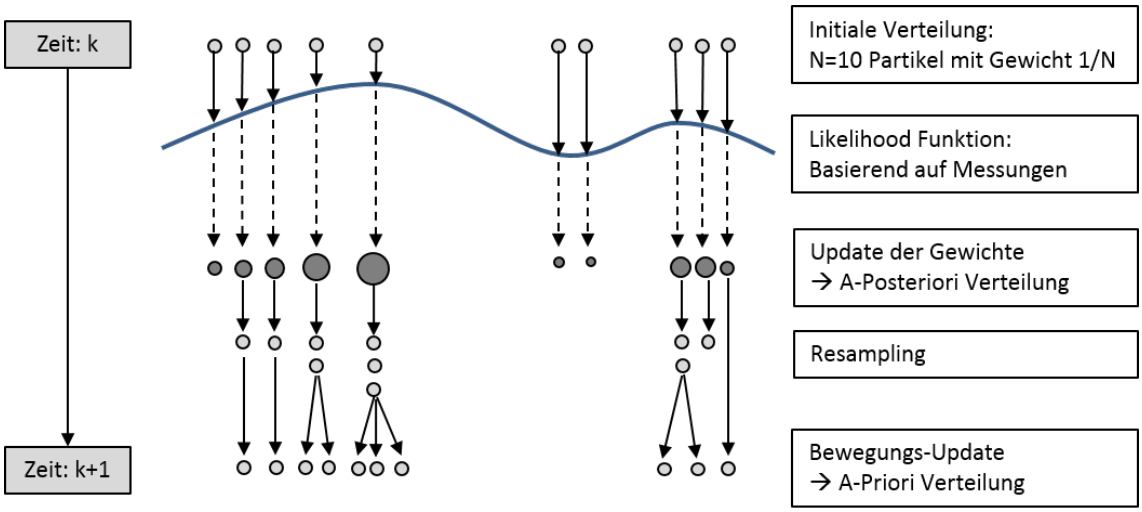


Abbildung 2.1.: Einzelner Zyklus eines Partikelfilters in Anlehnung an [10, S.44]

2.2.2. Mathematische Entstehung

Im folgenden Abschnitt wird die mathematische Entstehung der einzelnen Schritte des Partikelfilters erklärt.

Bayes'sches Theorem:

Die Basis des Partikelfilters ist das Bayes'sche Theorem, welches auch bei Kalmanfiltern Verwendung findet. Da die A-posteriori Verteilung bei Partikelfiltern $p(x_{0:k}|Z_k)$ den Verlauf des Systemzustands über den gesamten Zeitraum darstellt, muss deren Marginalverteilung berechnet werden, um den Systemzustand zum Zeitpunkt k zu erhalten. Dies erweist sich allerdings als sehr schwierig, da dafür komplexe und hochdimensionale Integrale berechnet werden müssen. Die Lösung für dieses Problem bietet die Monte-Carlo-Integration, eine numerische Methode um mehrdimensionale Integrale anzunähern.

Monte-Carlo Integration:

Angenommen das gegebene Integral $I = \int g(x)\,dx$ soll bestimmt werden, dann sieht die Monte-Carlo Integrations-Methode vor, $g(x)$ in zwei Funktionen zu unterteilen:

$$g(x) = f(x) \cdot \pi(x) \tag{2.16}$$

Dabei ist π eine Wahrscheinlichkeitsdichte-Funktion mit $\pi(x) \geqslant 0$ und $\int \pi(x)\,dx = 1$. In dem Kontext der Bayes'schen Regel stellt π die Dichtefunktion der A-posteriori Verteilung dar.
$f(x)$ muss so gewählt werden, dass es die Funktion $g(x)$ annähert, jedoch leichter zu integrieren ist. Die Wahl von $f(x)$ ist also abhängig von dem Problem, das gelöst werden soll. [13, S.141f] Im nächsten Schritt werden N zufällige und voneinander unabhängige Partikel x_i nach der Dichtefunktion π verteilt. Daraufhin kann das Integral

$$I = \int f(x) \cdot \pi(x)\,dx \tag{2.17}$$

durch

$$I_N = \frac{1}{N} \sum_{i=1}^{N} f(x_i) \tag{2.18}$$

angenähert werden. Für ein großes N konvergiert I_N zu I. [13, s.142][3, S.7][10, S.36]

Die Herausforderung bei dem Monte-Carlo Verfahren liegt darin, effizient Samples von der Wahrscheinlichkeits-Verteilung $\pi(x)$ zu erzeugen, da diese nicht eindeutig, gleichmäßig oder nach einem bestimmten Standard verteilt ist. Aus diesem Grund wird das Importance Sampling angewendet.

Importance Sampling:

Dazu wird $\pi(x)$ um eine Funktion $q(x)$ erweitert, welche ähnlich zu $\pi(x)$ ist, aus der aber einfacher Samples erzeugt werden können. $q(x)$ wird als Importance Density oder Wichtigkeitsdichte bezeichnet.
(2.17) kann somit durch $q(x)$ umgeschrieben werden zu:

$$I = \int f(x) \cdot \pi(x)\,dx = \int f(x) \cdot \frac{\pi(x)}{q(x)} \cdot q(x)\,dx \tag{2.19}$$

Durch die Verteilung von N zufälligen und voneinander unabhängigen Partikeln x_i nach $q(x)$, kann das Integral I als folgende gewichtete Summe dargestellt werden:

$$I_N = \frac{1}{N} \sum_{i=1}^{N} f(x_i) \cdot w(x_i) \tag{2.20}$$

Diese haben folgende Gewichte:

$$\tilde{w}(x_i) = \frac{\pi(x_i)}{q(x_i)} \tag{2.21}$$

Die Gewichte geben an, wie stark die Wichtigkeitsdichte $q(x)$ am Punkt x_i von der tatsächlichen A-posteriori Funktion $\pi(x)$ abweicht. Die Normalisierung der Gewichte erfolgt durch folgende Rechnung [10, S.37]:

$$w(x_i) = \frac{\tilde{w}(x_i)}{\sum_{j=1}^{N} \tilde{w}(x_j)} \tag{2.22}$$

Durch die Normalisierung ergibt die Summe der Gewichte den Wert 1. Dies ist notwendig, da sich das Objekt auf jeden Fall innerhalb der Messweite des Sensors befindet und deshalb die Summe der Wahrscheinlichkeiten der möglichen Aufenthalsorte bei 100 % liegt.

Da beim Tracking immer wieder neue Messungen vorhanden sind, ist es für das Importance Sampling erforderlich, dass für jeden Zeitschritt alle Gewichte neu berechnet werden. Dies würde zu einem viel zu hohen Rechenaufwand führen[3, S.9]. Deshalb wurde der Importance Sampling Algorithmus zu einem Sequentiellen Importance Sampling Algorithmus (SIS) abgeändert.

Sequentielles Importance Sampling:

Dabei werden die errechneten Gewichte bis zum Zeitpunkt $k-1$ wiederverwendet und durch die neue Messung angepasst. Das Ziel ist es, die A-posteriori Wahrscheinlichkeitsdichte zum Zeitpunkt k durch N gewichtete Partikel $\{x_k^i, w_k^i\}_{i=1}^N$ anzunähern. Da bei Partikelfiltern die A-posteriori Wahrscheinlichkeitsdichte $p(x_{0:k}|Z_k)$ den Verlauf des Systemzustands über die ganze Zeit angibt, muss die Marginalverteilung $p(x_k|Z_k)$ berechnet werden. Zunächst wird die A-posteriori Funktion durch folgende diskrete, gewichtete Summe angenähert:

$$p(x_{0:k}|Z_k) \approx \sum_{i=1}^{N} w_k^i \delta(x_{0:k} - x_{0:k}^i) \tag{2.23}$$

Dabei ist $\delta(x_{0:k} - x_{0:k}^i)$ die sogenannte Dirac-Delta-Funktion. Diese hat den Wert 1, falls das Argument $x_{0:k} - x_{0:k}^i$ 0 ist. Andernfalls ist das Ergebnis der Dirac-Delta Funktion 0.

Nach dem Prinzip des Importance Samplings sind die Punkte x_i nach einer Wichtigkeitsdichtefunktion $q(x_{0:k}|Z_k)$ verteilt. Bei dem SIS-Algorithmus sollen die Gewichte der Partikel jedoch nicht neu berechnet werden, sondern durch Wiederverwendung des bisherigen Verlaufs $q(x_{0:k-1}|Z_{k-1})$ und eines Gewichtsupdates $q(x_k|x_{0:k-1}, Z_{k-1})$, basierend auf diesem Verlauf, beschrieben werden. Durch mehrere Rechenschritte, welche in [10, S.37ff] und [3, S.9] beschrieben sind, entsteht die nachfolgende Funktion, durch welche die Gewichte in jedem Zeitschritt aktualisiert werden können:

$$w_k^i \propto w_{k-1}^i \cdot \frac{p(z_k|x_k^i) \cdot p(x_k^i|x_{k-1}^i)}{q(x_k^i|x_{k-1}^i, z_k)} \tag{2.24}$$

$p(z_k|x_k^i)$ ist, analog zum Kalmanfilter, die Likelihood Funktion, welche durch das Messmodell in Gleichung (2.2) definiert wird. $p(x_k^i|x_{k-1}^i)$ wird durch die Systemübergangsmatrix (2.1) beschrieben. Mithilfe der Gewichte kann daraufhin die Marginalverteilung berechnet werden:

$$p(x_k|Z_k) \approx \sum_{i=1}^{N} w_k^i \cdot \delta(x_k - x_k^i) \tag{2.25}$$

Durch eine gute Wahl von $q(x)$ in Gleichung (2.24), kann die Berechnung der Gewichte sogar noch vereinfacht werden. Eine häufige Wahl für $q(x_k^i|x_{k-1}^i, z_k)$ ist die Zustandsübergangsfunktion $p(x_k|x_{k-1})$ [10, S.47]. Dadurch wird (2.24) zu:

$$w_k^i \propto w_{k-1}^i \cdot p(z_k|x_k^i) \tag{2.26}$$

Resampling:

Das Problem des SIS-Algorithmus ist, dass er sehr ungenau wird, je mehr Zeitschritte vergehen. Dies liegt an der Degeneration der Partikel. Das bedeutet, dass nach einigen Iterationen nur noch ein Partikel ein so großes Gewicht hat, dass er für den Algorithmus relevant ist. Alle anderen Gewichte sind so klein, dass sie vernachlässigt werden können.

Wenn ein Partikel durch die fortgesetzte Iteration ein hohes Gewicht bekommen hat, beginnt er die Verteilung zu dominieren. Diese Dominanz ist nicht zweckdienlich, da dadurch wesentliche Eigenschaften der Verteilung verloren gehen. Ändert sich der Zustand, so bedarf es vieler neuer Messungen, um das zu große Gewicht des Partikels wieder abzusenken. Ein Partikel kann also schon veraltet sein und trotzdem noch ein hohes Gewicht haben. [13, S.146][5, S.58]
Die Messgröße, anhand welcher die Größe des Degenerationsproblems gemessen wird, ist die effektive Sample Größe:

$$N_{eff} = \frac{1}{\sum_{i=1}^{N}(w_k^i)^2} \tag{2.27}$$

N_{eff} variiert zwischen den Werten 1, wenn, ausgenommen von einem einzigen Partikel, alle Partikel das Gewicht 0 haben, und N, wenn alle Gewichte gleich groß sind. Je niedriger N_{eff} ist, desto größer ist das Degenerationsproblem. Um gegen diesen unvermeidbaren Effekt anzukämpfen, wurde ein Resampling Schritt eingeführt, der immer dann ausgeführt wird, wenn N_{eff} unter einen bestimmten Wert N_{thr} fällt. Bei dem Resampling Schritt werden Partikel mit niedriger Gewichtung gelöscht und stattdessen Partikel mit höherer Gewichtung dupliziert. Nach dem Resampling werden alle Gewichte wieder auf den Initialwert $\frac{1}{N}$ gesetzt, da die Information, welche sich hinter den Gewichten verbirgt, bereits verarbeitet wurde. [10, S.40f][1, S.179f]

3. Konzept zur praktischen Umsetzung eines Partikelfilters

Um eine Untersuchung des Partikelfilters in der Praxis durchführen zu können, muss im ersten Schritt ein passender Anwendungsfall bestimmt werden. Für diesen können daraufhin die Anforderungen ermittelt werden, welche sich durch die Leitfragen der Bachelorarbeit ergeben. Mithilfe dieser Basis kann schließlich ein Konzept zur praktischen Umsetzung eines Partikelfilters entworfen werden.

3.1. Auswahl eines Anwendungsbeispiels

Um die Brauchbarkeit und die Qualität des Algorithmus zu testen, ist es sinnvoll, den Partikelfilter nicht sofort an einem realen Beispiel zu testen. Dies hat mehrere Gründe. Zunächst einmal ist es sehr aufwändig, ein reales Bewegungs- und Messmodell mathematisch zu beschreiben. Dafür müssen viele Faktoren berücksichtigt und die Parameter gut ausgewählt werden. Die Zeit, die benötigt wird, um diese geeignet zu bestimmen ist unverhältnismäßig hoch, weil lediglich getestet werden soll, wie gut der Algorithmus in der Praxis umsetzbar ist und wie er sich für mehrere Eingabeparameter verhält. Zweitens führen komplexe Modelle zu mehr potenziellen Fehlerquellen. Es wäre dann schwer zu beurteilen, wie gut der Algorithmus tatsächlich funktioniert, da Fehler oder schlechte Laufzeiten sowohl durch den Partikelfilter-Ansatz, als auch durch die Modelle oder Parameter entstehen können. Zuletzt ist es schwer einzuschätzen, wie hoch die Qualität des Fusions-Ergebnisses ist, wenn reale Tracking-Daten benutzt werden. Das liegt daran, dass die Daten fehlerbehaftet sind und die korrekte Position nicht erkennbar ist. Somit kann die Größe des Fehlers nicht beurteilt werden.
Aus diesen Gründen wird für die Bachelorarbeit ein vereinfachter Anwendungsfall verwendet. In dem ausgewählten Beispiel soll die Position eines mathematischen Pendels bestimmt werden. Dieses ist für die Untersuchung des Partikelfilters gut geeignet, da das Bewegungsmodell nur von zwei Parametern abhängig ist: Von dem Winkel ϕ und der zugehörigen Winkelgeschwindigkeit ω. Die Vorhersage der Bewegung ist deshalb mit weniger potenziellen Fehlerquellen behaftet. Außerdem können sehr leicht Testdaten generiert und verrauscht werden. Somit ist eine präzise Beurteilung der Qualität des Fusionsergebnisses möglich.

3.2. Anforderungsanalyse

In diesem Kapitel werden die drei anfangs definierten Leitfragen genauer untersucht. Dabei soll festgestellt werden, worauf die jeweilige Frage abzielt und welche Anforderungen erfüllt sein müssen, um diese zu beantworten.

1. **Ist ein Partikelfilter geeignet, um eine Sensordatenfusion für unterschiedliche Tracks mit verschiedenen Sensoren im Phasenraum zu realisieren?**
   Diese Frage betrifft vorwiegend die Implementierung des Partikelfilters. Dabei muss die Möglichkeit geboten werden, dass sich mehrere Objekte im Phasenraum des Pendels befinden. Das führt dazu, dass der Output möglicherweise mehrere Tracks zurückgeben muss. Außerdem muss beim Input der Messdaten darauf geachtet werden, dass die Messungen von verschiedenen Sensoren stammen können. Diese haben unterschiedliche Ungenauigkeiten, was auch bei dem Update der Gewichte berücksichtigt werden muss. Die Anforderungen an den Algorithmus, die durch die Frage entstehen, sind folgende:

   - Der Input des Partikelfilters umfasst mehrere Sensoren.
   - Der Output des Partikelfilters umfasst mehrere Tracks.
   - Der Input wird durch den Partikelfilter fusioniert und führt zu dem Output.

   Die Leitfrage ist beantwortet, wenn feststeht, welche der drei Anforderungen durch die Implementierung erfüllt werden können.

2. **Wie lässt sich eine Trackbildung realisieren?**
   Diese Leitfrage ist eine Erweiterung der ersten Frage. Nachdem der Input und der Output das richtige Format haben, muss nun geprüft werden, ob die Daten auch richtig verarbeitet werden. Diese Leitfrage zielt deshalb darauf ab, einen Algorithmus zu finden, welcher aus der Partikelwolke einen oder mehrere Tracks generiert. Dabei muss auch beachtet werden, dass sich die Anzahl der Tracks durch die Ereignisse „Split Track“ und „Merge Track“ verändern kann. Dieser Algorithmus muss daher folgende Anforderungen erfüllen:

   - Die Anzahl der Tracks wird von dem Algorithmus richtig festgestellt.
   - Die Tracks werden von dem Algorithmus an der richtigen Stelle erkannt.
   - Der Algorithmus kann die Fusion und Trennung von Tracks durchführen.

   Die Leitfrage ist beantwortet, wenn ein Algorithmus gefunden wurde, der die Anforderungen erfüllt.

3. **Wie hängt die Sensorgenauigkeit mit dem Fusionsergebnis zusammen?**
   Bei dieser Leitfrage, soll untersucht werden, wie sich die Genauigkeit des Sensors auf das Ergebnis auswirkt. Es soll festgestellt werden, wie gut der Algorithmus Sensorungenauigkeiten ausgleichen kann und ab wann er an seine Grenzen stößt. Dabei muss nicht nur beachtet werden, dass ein ungenauer Sensor Auswirkungen auf das Ergebnis haben kann, sondern auch ein zu genauer Sensor. Dies kann zutreffen, wenn der Algorithmus davon ausgeht, dass eine Ungenauigkeit vorliegt, obwohl keine existiert.

Die Leitfrage kann also auf folgende Fragen heruntergebrochen werden:

- Ab welcher Sensorungenauigkeit ist ein korrektes Tracking nicht mehr möglich?
- Welche Auswirkungen hat ein sehr genauer Sensor auf das Ergebnis?
- Wie gut ist das Fusionsergebnis bei durchschnittlicher Sensorungenauigkeit?

Die Leitfrage ist beantwortet, wenn eine Antwort auf diese drei Fragen gegeben werden kann.

3.3. Vorgehensweise

Nachdem ein Anwendungsbeispiel ausgewählt und die Anforderungen daran beschrieben wurden, kann nun die Implementierung eines Partikelfilters geplant werden. Das weitere Vorgehen, um alle Anforderungen zu berücksichtigen, wird im Folgenden beschrieben.
Als Erstes soll ein Aufbau und Ablaufplan für den Partikelfilter-Algorithmus festgelegt werden. Dabei müssen auch Designentscheidungen getroffen werden, welche angeben, ob und wie die Gewichte der Partikel aktualisiert werden und wie oft ein Resampling stattfindet. Zunächst soll dabei nur ein Track als Output ausgegeben werden.

Im zweiten Schritt werden die Generierung der Inputdaten und das Bewegungsupdate für die einzelnen Partikel festgelegt. Beide simulieren ein mathematisches Pendel im Phasenraum, welches, mithilfe des Runge-Kutta-Verfahrens, von seiner derzeitigen Position im Phasenraum um ein Zeitintervall nach vorne propagiert wird. Beim Einlesen und Verrauschen des Inputs muss die erste Leitfrage berücksichtigt werden. Diese schreibt vor, dass die Messdaten von mehreren Sensoren stammen können.

Anschließend werden das Messmodell und die Art des Resamplings erläutert. Das Messmodell beschreibt, basierend auf der neuen Messung, wie gut die einzelnen Partikel positioniert sind. Darauf aufbauend werden die Partikel, im Resampling Schritt gelöscht, Hinzugefügt oder Verschoben.

Sobald bis zu diesem Schritt alles funktioniert, kann der Output um mehrere Tracks erweitert werden. Dies hat zur Folge, dass ein Algorithmus entworfen werden muss, welcher die Trackbildung für mehr als einen Track durchführt. Bei diesem müssen die Anforderungen der zweiten Leitfrage berücksichtigt werden.

Anschließend wird die Visualisierung der (Zwischen-)Ergebnisse implementiert. Dies macht das Testen der Ergebnisse im letzten Schritt einfacher. Dabei werden die Fusionsergebnisse, basierend auf der zweiten und dritten Leitfrage, beurteilt. Es soll herausgefunden werden, ob der Partikelfilter korrekt funktioniert und wie gut die Ergebnisse sind. Falls diese nicht zufriedenstellend sind, muss zusätzlich noch geprüft werden, ob der Algorithmus noch optimiert werden kann oder ob einige Parameter noch angepasst werden müssen.

4. Der Partikelfilter in der Praxis

Basierend auf dem Vorgehen, welches in Kapitel 3.3 beschrieben wurde, wird nun die Implementierung eines Partikelfilters erläutert. Zunächst werden in Kapitel 4.1 der Aufbau und Ablauf eines Partikelfilters für die Positionsbestimmung im Phasenraum eines Pendels beschrieben. In der Praxis wurden diese beiden Aspekte im Vergleich zur Theorie leicht abgewandelt. Die Unterschiede und Gründe für die Änderung werden ebenfalls in Abschnitt 4.1 erläutert. Daraufhin wird in Kapitel 4.2 der Phasenraum eines Pendels genauer erklärt und beschrieben, wie darin Punkte nach vorne propagiert werden. Dieses Vorgehen wird sowohl für das Erzeugen der Messdaten, als auch für das Bewegungsupdate der einzelnen Partikel verwendet. Der Abschnitt 4.3 beschreibt die Umsetzung des Messmodells, welches die Partikel mit der aktuellen Messung in Verhältnis setzt. Anschließend wird der Partikelfilter um mehrere Tracks als Output erweitert. Dazu wird in Abschnitt 4.4 der Algorithmus zur Trackbildung erläutert. Zum Schluss wird in Kapitel 4.5 eine Visualisierung des Partikelfilters im Phasenraum beschrieben. Diese wurde nicht selbst implementiert, ist jedoch sehr nützlich, um zu testen, wie gut der Partikelfilter funktioniert. Das Testen des Partikelfilters wird anschließend in den Kapiteln 5 und 6 beschrieben. In den folgenden Kapiteln werden mehrere Codeabschnitte gezeigt. Diese sind oft auf das Wichtigste reduziert, um den Text übersichtlich zu halten. Der vollständige Code befindet sich im Anhang.

4.1. Aufbau und Ablauf des Partikelfilters

Bevor die tatsächliche Implementierung erfolgt, müssen erst Aufbau und Ablauf des Partikelfilters festgelegt werden. Zunächst wird der Aufbau anhand eines Klassenmodells dargestellt, welches in Abbildung 4.1 zu sehen ist. Anschließend wird der Ablauf des Partikelfilters anhand der Abbildung 4.2 beschrieben.

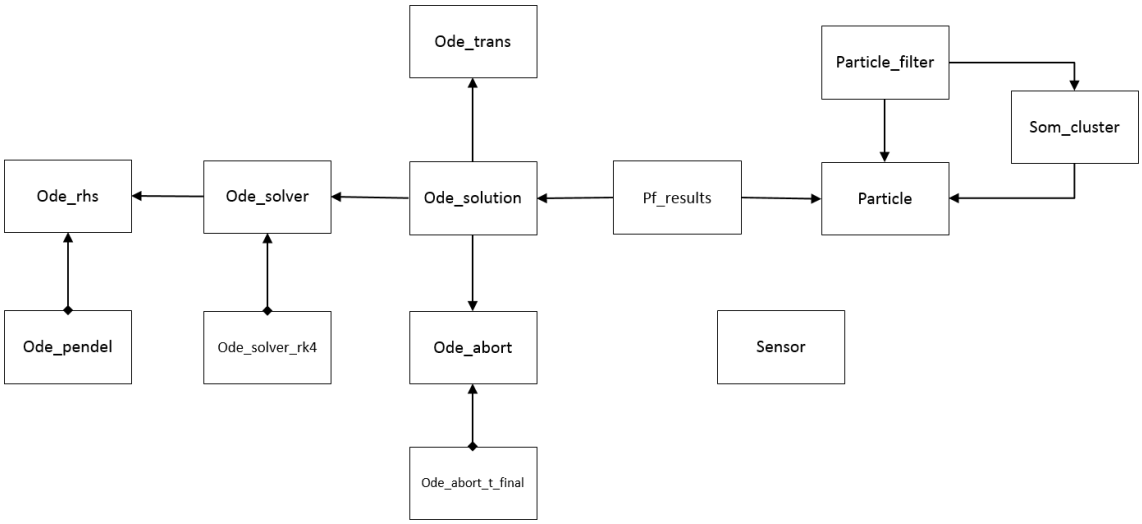


Abbildung 4.1.: Aufbau des Partikelfilters

Die Klassen des Partikelfilters aus Abbildung 4.1 können grob in zwei Teile untergliedert werden. Auf der linken Seite sind alle `Ode`-Klassen zu sehen, wobei „Ode“ die Kurzform für „ordinary differential equation“ ist. Diese Klassen haben den Zweck, die Pendelbewegung zu integrieren. Die Lösung der Integration, welche die Position des Pendels im nächsten Zeitschritt darstellt, wird in der Klasse `Ode_solution` gespeichert. Die Klasse `Ode_abort`, beziehungsweise deren Kindklasse `Ode_abort_t_final`, legt die Abbruchkriterien für die Integration fest. In diesem Fall wird die Integration beendet, wenn der nächste Zeitschritt erreicht ist. Die Hilfsklasse `Ode_trans` transformiert `x` nach `y` zum Beispiel über eine Skalierung. Die Klasse `Ode_solver` berechnet die Lösung einer Differenzialgleichungen. Dies kann auf mehrere Arten geschehen. Die Kindklasse `Ode_solver_rk4` verwendet das in Kapitel 4.2 dargestellte Runge-Kutta-Verfahren, um eine numerische Lösung zu erhalten. Die Klasse `Ode_rhs` unterstützt die Lösung der Differenzialgleichung, indem sie die rechte Seite der Differenzialgleichung löst. Ihre Kindklasse liefert dabei das Ergebnis für ein Pendel.

Der zweite Teil der Klassen implementiert den eigentlichen Ablauf des Partikelfilters. Die Lösung befindet sich in der Klasse `PF_results`, welche die Partikel in mehrere Arten unterteilt: Die wahre Position der Objekte, die Messungen und Sigmas der Sensoren, die Partikel mit ihrem Alter, die Trajektorien und die Tracks. Die, in der `Particle`-Klasse erzeugten Partikel, werden über die Klasse `PF_results` an die Klasse `Ode_solution` weitergegeben, welche die Position der Partikel aktualisiert. Die Klasse `Particle_filter` übernimmt die Organisation der Partikel anhand des Messmodells und das Resampling. Beides wird in Kapitel 4.3 erläutert. Die Organisation der Partikel basiert darauf, ob sich die Partikel in der Nähe einer Messung befinden. Daraus resultieren die Aktualisierung der Partikel und das Löschen veralteter Partikel. Außerdem gibt die `Particle_filter`-Klasse den Anstoß für die Trackbildung durch die Klasse `Som_Cluster`, welche die Tracks mithilfe einer sogenannten Self Organizing Map bestimmt. Dieses Prinzip wird in Kapitel 4.4 genauer erläutert. Zusätzlich gibt es noch eine `Sensor`-Klasse, welche die Sensor-Objekte zur Verfügung stellt. Aus Gründen der Übersichtlichkeit werden weitere, nicht selbst-implementierte Hilfsklassen für Winkel, Matrizen, Vektoren und Generierung von Zufallszahlen nicht in der Grafik abgebildet.

Nachdem alle nötigen Bestandteile des Algorithmus festgelegt wurden, wird als nächstes der Ablauf des Partikelfilters, welcher in Abbildung 4.2 dargestellt ist, Schritt für Schritt erläutert. Dabei sind Input und Output grau gekennzeichnet, das Update der Bewegung und der Zeit in Blau und das Messmodell zusammen mit dem Resampling in Rot.

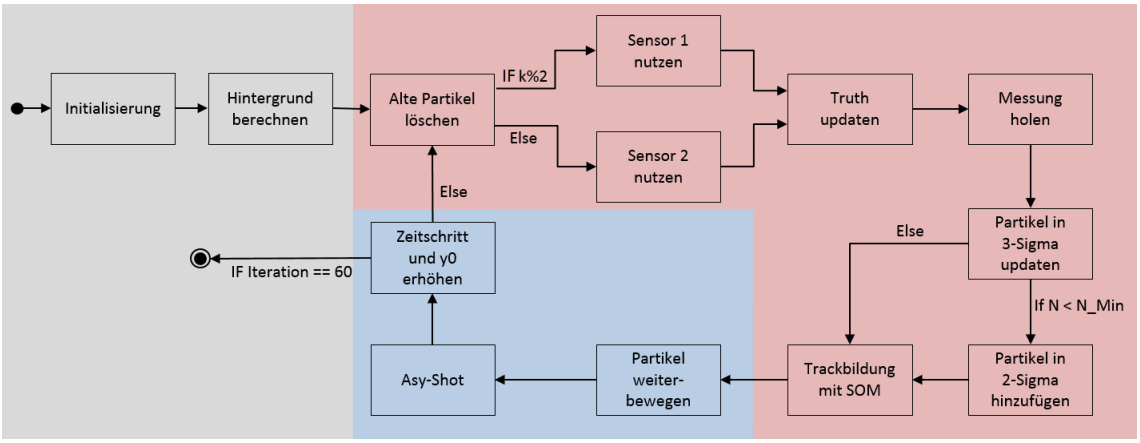


Abbildung 4.2.: Ablauf des Partikelfilters

Wie sich erkennen lässt, weist dieses Modell einige Unterschiede zu dem theoretischen Aufbau eines Partikelfilters auf. Diese Designentscheidungen wurden bewusst getroffen, da sie die Resultate verbessern und deren Berechnung vereinfachen. Der erste Unterschied liegt in der Gewichtung der Partikel. In der Theorie werden die Gewichte der Partikel höher, je näher sie der Messung sind. In der Umsetzung gibt es lediglich das Alter der Partikel, welches als Gewichtung bezeichnet werden kann. Je niedriger das Alter ist, desto eher wird es bei der Trackbildung berücksichtigt. Sind die Partikel zu alt, werden sie gelöscht. Natürlich spielt die Nähe der Partikel zu der Messung weiterhin eine Rolle, da die Aktualisierung der Partikel und somit das Alter davon abhängig sind.

Der zweite Unterschied liegt darin, dass in der Umsetzung, anders als in der Theorie, das Messmodell und das Resampling nicht eindeutig trennbar sind. Deshalb ist das Resampling in Abbildung 4.2 nicht explizit aufgeführt. Der Grund für die Mischung der beiden Funktionen liegt darin, dass sich die Anzahl der Partikel aufgrund der Gewichtung durch das Alter ändert. Das bedeutet, dass nicht an einer Stelle des Algorithmus die Partikel verschoben und alle Gewichte zurückgesetzt werden, wie es die Theorie vorsieht. Stattdessen werden am Anfang jeder Iteration die veralteten Partikel gelöscht und am Ende der Anwendung des Messmodells wieder Partikel in die 2-Sigma Umgebung hinzugefügt, falls zu wenige vorhanden sind, die der Messung sehr nah sind. Auf diese Weise wird sichergestellt, dass vorhergesagte Informationen, die mit der Messung verträglich sind, berücksichtigt werden. Weiterhin werden Informationen aus der Messung durch den Versatz der Partikel in Richtung des Messergebnisses absorbiert. Gleichzeitig werden vorhergesagte Informationen, die nicht mit der Messung verträglich sind, verworfen, sobald sie zu alt werden und zwischenzeitlich keine Unterstützung durch Messergebnisse erfahren haben. Dieses Vorgehen entspricht qualitativ einem Sequential Importance Sampling, weil die Partikeldichte in vergleichbarer Weise verändert wird.

Nachdem die Unterschiede zwischen Theorie und Praxis diskutiert wurden, wird nun der Ablauf des Partikelfilters erklärt. Der erste Schrittist die Initialisierung des Partikelfilters mithilfe einiger Variablen. Dies sind einerseits die Anfangszeit `t0`. die Fallbeschleunigung `g`, die Länge `L` des Pendelstabs, `PI_0` für die numerische Berechnung des Integrals, die aktuelle Zeit `t`, die Länge der Zeitintervalle `Delta_t`, die Zahl der Iterationen `iterations` und die tatsächliche Position des Objekts für die Integration `y0`. Zum Anderen sind das Objekte, die erzeugt werden müssen, um den Algorithmus durchführen zu können. Dies sind jeweils ein Objekt der Klassen `Particle_filter`, `Ode_pendel`, `grfl_ranx` (zur Generierung von Zufallszahlen), sowie drei Objekte der `Sensor`-Klasse und eine Liste an Objekten der Klasse `Ode_solution`.

```
1  //Variables
2  double t0 = 1.0;            //starting time in s
3  double g = 9.81;            //gravitational acceleration in m/s^2
4  double L = 9.81;            //length of pendulum in m
5  double PI_0 = t0*t0*g/L;    //for integration
6
7  double t = 0.0;             //time t
8  double Delta_t = 2.0;       //delta t
9  int iterations = 60;        //number of iterations
10
11 //Starting position of the true position of the object
12 std::deque<grfl_vector> y0;
13 y0.push_back(grfl_vector(std::vector<double> {0.0,+1.5}));
14 y0.push_back(grfl_vector(std::vector<double> {0.0,-1.0}));
```

```
15
16  //Objects
17  cl_particle_filter pf = cl_particle_filter();   //particle filter
18  cl_ode_pendel pendel = cl_ode_pendel(PI_0);     //solving pendulum ode
19  grfl_ranx rg = grfl_ranx();                     //random number
20
21  //Sensor objects
22  const cl_sensor sr_sensor = cl_sensor(0.2);     //sensor 1
23  const cl_sensor fc_sensor = cl_sensor(0.1);     //sensor 2
24  cl_sensor sensor = sr_sensor;                   //chosen sensor(alternates between both)
25
26  //Background trajectories
27  std::deque<cl_ode_solution> kulisse;
```

Danach wird der Hintergrund für die Visualisierung des Partikelfilters berechnet. Dies erfolgt mithilfe der Funktion `background_trajectories`. Sie wird im Kapitel 4.5 „Visualisierung“ erklärt.

Ab nun beginnt die Iteration des Pendels über sechzig Zeitschritte. Dafür werden zunächst alle Partikel gelöscht, die das letzte Mal vor mehr als drei Zeitschritten aktualisiert wurden. Der Code dafür wird in Kapitel 4.3 über das Messmodell erklärt. Als nächstes wird abwechselnd einer der beiden Sensoren gewählt, um die weiteren Berechnungen durchzuführen. Es ist auch möglich, jede Iteration mit beiden Sensoren zu berechnen. Dies führt allerdings zu einem höheren Rechenaufwand, weshalb sich für die erste Variante entschieden wurde. Um den tatsächlichen Aufenthalt des Pendels im Phasenmodell im Auge zu behalten, wird als nächstes die wahre Position im nächsten Zeitschritt berechnet. Dazu wird die wahre Position aus dem letzten Zeitschritt mithilfe des Runge-Kutta-Verfahrens, welches in Kapitel 4.2 erklärt wird, integriert.

```
1   //the result of the integration is stored here with type cl_pf_results
2   cl_pf_results res = cl_pf_results();
3   res.clear();
4
5   //step size for integrating the truth to time t
6   double t_step = 0.03;
7   //initialize the rk4-solver for the integration step
8   cl_odesolver_rk4 rk4 = cl_odesolver_rk4(y0.front().size(),t_step);
9
10  //integration stops when reaching next time step
11  cl_ode_abort_t_final abort = cl_ode_abort_t_final(t+Delta_t);
12
13  //truth is stored in a list of true solutions of the results object
14  res.truth().push_back(cl_ode_solution());
15
16  //determine the truth trajectory by integrating a pendel object
17  //with initial condition y0, from t'=t to t'=t+Delta_t
18  res.truth().back().integrate(y0,t,rk4,pendel,abort);
```

Danach wird die nächste Messung des ausgewählten Sensors erfasst, in das Format eines `Particle`-Objekts umgewandelt und im `res`-Objekt gespeichert.

```
//store the measurement coordinates here
double x_measure = 0.0;
double y_measure = 0.0;

//get coordinates of next measurement and store them in the declared variables
sensor.measure(y[0],y[1],x_measure,y_measure,rg);

//make a particle for the measurement
cl_particle p_measure = cl_particle(x_measure,y_measure,t);

//save the measurement particle in the results object
res.measurements().push_back(p_measure);
//Also save the sigma of the sensor
res.sigmas().push_back(sensor.tell_sigma());
```

Da nun die Messung und die Sensorungenauigkeit bekannt sind, können alle Partikel aktualisiert werden, die sich in der 3-Sigma Umgebung der Messung befinden. Die Berechnung des Resamplings und der zugehörige Code sind in Kapitel 4.3 beschrieben. Sollten sich zu wenige Partikel in der 3-Sigma Umgebung der Messung befinden, müssen so viele Partikel hinzugefügt werden, dass die Mindestanzahl erreicht ist. Dies ist nötig, da zu Beginn der nächsten Iteration Partikel gelöscht werden und das Ergebnis verfälscht werden könnte, wenn die Berechnungen auf zu wenigen Partikeln basieren. Die neuen Partikel werden zufällig innerhalb der 2-Sigma Umgebung verteilt, um sowohl das hinzugekommene Messergebnis als auch seine Unschärfe in der neuen Partikeldichte zu repräsentieren. Sie setzen sich aus einem zufällig bestimmten Winkel und Abstand zu der Messung zusammen.

```
//minimum number of Particles in 3-Sigma environment
int N_min = 40;

//avoid drastic outliers when adding particles,
//cut off at 2 sigma, which covers 95 percent
double cut_off_sigma = 2.0*sensor.tell_sigma();

//coordinates of the new particle
double x = 0.0;
double y = 0.0;

//distance to the measurement
double z = 0.0;

//add particles as long as n < n_min
for (int i=N; i < N_min; ++i){
    //make a gaussian distributed random number that lies within the cut_off
    do{
        z = fabs(rg.gauss(0.0,sigma));
    }while(fabs(cut_off_sigma) < z);

    //direction in which the particle lies
    cl_angle beta = cl_angle(M_PI*rg.ran1_pm(),angle::radian);

```

```
25      //calculate x- and y-coordinate of new particle
26      x_out = x_measure+z*cos(beta.rad());
27      y_out = y_measure+z*sin(beta.rad());
28
29      //add new particle to list of particles
30      pf.push_back(cl_particle(x,y,t));
31  }
```

Im nächsten Schritt erfolgt die Trackbildung mithilfe einer sogenannten Self Organizing Map. Wie das funktioniert und wie die Implementierung aussieht, wird in Kapitel 4.4 zur Trackbildung erklärt.

Nachdem die Trackbildung erfolgt ist, können alle Partikel um einen Zeitschritt weiterbewegt und deren letzte Resampling-Zeit aktualisiert werden. Die Integration, um die nächste Position zu erhalten, erfolgt erneut mit dem Runge-Kutta-Verfahren, welches in Kapitel 4.2 zum Bewegungsupdate eines Pendels erklärt wird.

Um den neuen Zustand festzuhalten, wird er mithilfe von „Asymptote“ visualisiert und als PDF-Datei gespeichert. Wie die Visualisierung funktioniert und aussieht, wird im nächsten Kapitel 4.5 gezeigt. Im letzten Schritt wird die Zeit erhöht und der wahre Zustand `y0` für die nächste Iteration aktualisiert.

```
1  //update time
2  t+=Delta_t;
3
4  //update the next initial state
5  y[0]=results.truth().back()[0];
6  y[1]=results.truth().back()[1];
```

Sind alle sechzig Iterationen durchgelaufen, endet der Algorithmus. Als Output ergibt sich eine Liste von Partikeln, mit einem Track der daraus generiert wurde und eine Visualisierung der (Zwischen-)Ergebnisse.

4.2. Bewegung im Phasenraum eines Pendels

In diesem Abschnitt wird erklärt, was der Phasenraum eines Pendels ist, was darin dargestellt wird und wie eine Bewegung darin berechnet wird. Anschließend wird erläutert, wofür dieses Modell in dem hier umgesetzten Partikelfilter verwendet wird.

Wie bereits erwähnt, ist ein mathematisches Pendel ein optimales Pendel. Faktoren, wie der Luftwiderstand werden dabei nicht beachtet. Aus diesem Grund ist es nur von zwei Variablen abhängig: Dem Winkel ϕ und der zugehörigen Winkelgeschwindigkeit ω. Diese beiden Größen werden in einem Phasenraum dargestellt. Dabei stellt die x-Achse den Winkel und die y-Achse die Winkelgeschwindigkeit dar. Darauf aufbauend ist der Phasenraum in Abbildung 4.3 dargestellt, welcher die Bewegung eines Pendels visualisiert.

Die grünen Punkte stellen die Ruhepositionen eines Pendels dar, in denen es senkrecht nach unten hängt ohne zu Schwingen. Dabei befindet sich das Pendel in einem stabilen Zustand. Auch bei den blauen Punkten ist das Pendel in einer Ruheposition. Hierbei handelt es sich allerdings um einen instabilen Zustand, weil das Pendel senkrecht nach oben gerichtet ist. Die kleinste Änderung bringt es zu Fall. In der Realität existiert dieser Zustand nicht. Bei einem idealen Pendel ist das jedoch möglich. Je größer die Auslenkung des Pendels ist, desto größer ist der dargestellte Kreis, beziehungsweise die Ellipse. Je näher das Pendel an die instabile Ruheposition gelangt, desto elliptischer wird die dargestellte Kurve. Die in Rot dargestellten Schwingungen entstehen, wenn das Pendel durchschwingt.

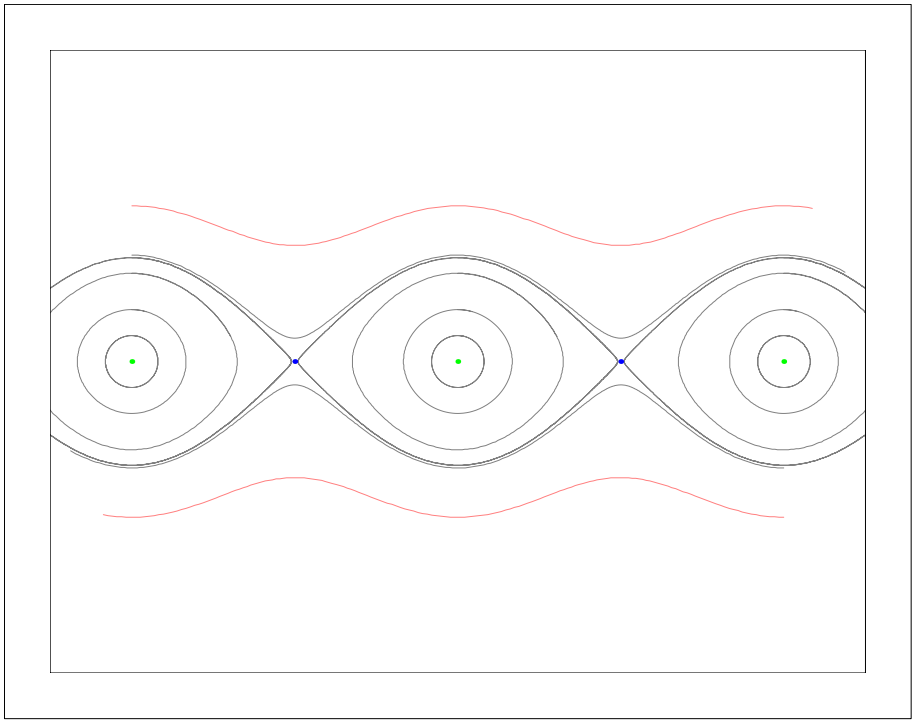


Abbildung 4.3.: Phasenmodell eines mathematischen Pendels
Winkel ϕ auf der x-Achse und Winkelgeschwindigkeit ω auf der y-Achse. Die grünen Punkte stellen die stabile Ruhelage des Pendels dar. Die blauen Punkte zeigen die instabile Ruhelage des Pendels. Je größer der Kreis ist, desto höher die Auslenkung des Pendels. Die rote Schwingung entsteht, wenn das Pendel durchschwingt.

Mathematisch betrachtet lässt sich die Bewegung des Pendels durch zwei gekoppelte Differenzialgleichungen erster Ordnung beschreiben:

$$\dot{\phi} = \omega \tag{4.1}$$

$$\dot{\omega} = -\frac{g}{L}\sin\phi \tag{4.2}$$

Dabei ist ϕ der Winkel des Pendels und ω dessen Winkelgeschwindigkeit. $\dot{\phi}$ und $\dot{\omega}$ stellen deren Ableitungen dar, g bezeichnet die Gewichtskraft und L ist die Länge des Pendelstabs.[9]

Da dies Differenzialgleichungen erster Ordnung sind, kann das sogenannte Runge-Kutta-Verfahren angewendet werden, um die Position im nächsten Zeitschritt zu berechnen. Es wird auch als RK4-Verfahren bezeichnet, da genau vier Schritte nötig sind, um eine Differenzialgleichung erster Ordnung numerisch zu lösen. Dabei werden die Hilfssteigungen k_1 bis k_4 gebildet, welche folgendermaßen berechnet werden:

$$k_1 = f(t_i, y_i) \tag{4.3}$$
$$k_2 = f(t_i + \tfrac{h}{2}, y_i + \tfrac{h}{2} \cdot k_1) \tag{4.4}$$
$$k_3 = f(t_i + \tfrac{h}{2}, y_i + \tfrac{h}{2} \cdot k_2) \tag{4.5}$$
$$k_4 = f(t_i + h, y_i + h \cdot k_3) \tag{4.6}$$

Dabei ist $f(t_i, y_i)$ die erste Ableitung von y_i nach t_i und h die Größe des Zeitschritts. Der nächste Schritt y_{i+1} wird dann durch folgende Rechnung bestimmt:

$$y_{i+1} = y_i + h \cdot \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{4.7}$$

[8, S.2]

Bevor die Implementierung des RK4-Verfahrens dargestellt wird, muss zunächst eine wichtige Hilfsfunktion erklärt werden. Diese leitet einen Vektor $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ab. y_1 ist dabei der Winkel ϕ und y_2 dessen Ableitung $\dot{\phi} = \omega$. Die Ableitung erfolgt über zwei Schritte. Erst wird $y1$ abgeleitet und anschließend y_2. Die Werte werden in einem übergebenen Vektor namens `rhs` gespeichert. `rhs` beinhaltet nach der Berechnung die Werte $\dot{\phi}$ und $\dot{\omega}$, welche durch die Gleichungen (4.1) und (4.2) gegeben sind. Nachfolgend ist die Implementierung der Funktion `compute_rhs` zu sehen.

```
bool cl_ode_pendel::compute_rhs(const grfl_vector& y, const double t, grfl_vector& rhs) {
        // y = (phi,phi_dot)
        // rhs = y_dot = (phi_dot, -(g/L) * sin(phi))
        rhs[0] = y[1];
        rhs[1] = -(g/l)*sin(y[0]);

        //success
        return(true);
}
```

Bei der Implementierung des RK4-Verfahrens wird zunächst k_1, also die Ableitung von y_i nach t_i, berechnet. Dies geschieht durch die oben erklärte Hilfsfunktion `compute_rhs`. Das Ergebnis wird in der Variable `k1` gespeichert.

```
//calculate k1: derive y_0 with respect to t and store result in k1
ode_rhs.compute_rhs(y_0,t,k1);
```

Anschließend wird k_2 analog zu (4.4) berechnet. Dafür wird zuerst die Funktion $y_i + \frac{h}{2} \cdot k_1$ aufgestellt, die schließlich nach $t_i + \frac{h}{2}$ abgeleitet wird.

```
//calculate k2:

//construct y_1 = y_0 + h/2 * k1
y_1 = y_0;
y_1.scale_add(0.5*h,k1); //add 0.5*h*k1
```

```
6
7  //derive k2 with respect to t+h/2
8  ode_rhs.compute_rhs(y_1,t+0.5*h,k2);
```

Dieses Vorgehen wird für die Berechnung von k_3 und k_4 nach den Formeln (4.5) und (4.6) wiederholt:

```
1  //calculate k3:
2
3  //construct y_2 = y_0 + h/2 *k2
4  y_2 = y_0;
5  y_2.scale_add(0.5*h,k2);
6
7  //derive k3 with respect to t+h/2
8  ode_rhs.compute_rhs(y_2,t+0.5*h,k3);
9
10 //calculate k4:
11
12 //construct y_3 = y_0 + h*k3
13 y_3 = y_0;
14 y_3.scale_add(h,k3);
15
16 //derive k4 with respect to t+h
17 ode_rhs.compute_rhs(y_3,t+h,k4);
```

Abschließend kann die neue Position des Pendels mithilfe der Gleichung (4.7) berechnet werden:

```
1  //sum up the resulting new state: y_{n+1} =  y_0 + h/6 * (y'_0 + 2 (y'_1+y'_2) + y'_3)
2  y_new[i] = y_0[i] + h/6 * ( k1[i] + 2.0 * (k2[i] + k3[i]) + k4[i]);
```

Um den abgebildeten Code auf das Wesentliche zu beschränken, wurden in der Darstellung einige Abbruchkriterien weggelassen. Der vollständige Code ist im Anhang zu finden.

Diese Methode wird sowohl für die Generierung der Messdaten, als auch für das Bewegungsupdate der einzelnen Partikel eingesetzt. Der einzige Unterschied zwischen den Verwendungen besteht darin, dass die Messdaten zusätzlich noch verrauscht werden. Dies geschieht durch Hinzufügen einer gaußverteilten Zufallszahl.

4.3. Messmodell und Resampling

Das Messmodell beschreibt die Korrektur der Partikel, sobald eine neue Messung verfügbar ist. Es wird festgelegt, wie wahrscheinlich es ist, dass ein Partikel die richtige Position hat, wenn der Messpunkt des Sensors an einem bestimmten Ort liegt. Je näher die Partikel an diesem Messpunkt liegen, desto wahrscheinlicher ist es, dass die Position des Partikels korrekt ist. Dieses Messmodell wird auf alle Partikel angewendet. Dadurch entsteht eine Gewichtung, welche angibt, wie gut die Position der Partikel ist.

Hoch gewichtete Partikel werden weiterverarbeitet, wohingegen niedrig Gewichtete verworfen werden. In diesem Anwendungsfall wurde das Messmodell so umgesetzt, dass das entscheidende Gewicht der Anzahl der Zeitschritte entspricht, die vergangen sind, seit der Partikel das letzte Mal aktualisiert wurde. Eine Aktualisierung, auch Resampling genannt, findet immer dann statt, wenn sich der Partikel innerhalb des 3-Sigma Radius der aktuellen Messung befindet. Das Sigma beschreibt die Ungenauigkeit des Sensors.
Der 3-Sigma Radius ist also dreimal so groß wie die Sensorungenauigkeit. Sobald mehr als 3 Zeitschritte vergangen sind, seit sich der Partikel das letzte Mal in der 3-Sigma Umgebung einer Messung befunden hat, wird der Partikel gelöscht. In der Implementierung wurde dies folgendermaßen umgesetzt:

```
1  //maximum lifetime of a particle is 3 time steps
2  double max_lifetime = 3*Delta_t;
3
4  //check lifetime of all particles
5  for (unsigned int i=0; i < particles.size(); ++i) {
6      //particles that have not been updated by more than max_lifetime, are removed.
7      if (max_lifetime < (t - pf[i].tell_last_update_time())){
8          remove(i);
9      }
10 }
```

Bei dem Resampling wird der Partikel ein Stück in die Richtung der Messung verschoben. Die Berechnung der neuen Position p' des Partikels erfolgt durch folgende Rechnung:

$$p' = p + |x|e_q + n \tag{4.8}$$

p ist dabei die aktuelle Position des Partikels. Das Zufallsrauschen n ist kreisförmig gaußverteilt mit 0.1· Sigma und dient dazu, dass sich nicht alle Partikel auf dem Punkt r niederlassen, der deutlich kleiner als Sigma ist. $|x|$ ist der Wert einer Zufallszahl x, welche normalverteilt um 0 ist und für die eine Standardabweichung von 0.3· Sigma festgelegt wurde. Um den Wert dieser Zufallszahl wird der Partikel in die Richtung der Variable e_q versetzt, welche angibt, wo die Messung r liegt. e_q kann folgendermaßen berechnet werden:

$$e_q = \frac{r - p}{|r - p|} \tag{4.9}$$

Bei der Implementierung werden zunächst alle Partikel gespeichert, welche sich in der 3-Sigma Umgebung einer Messung befinden:

```
1  //particles, that are going to be updated are stored here
2  std::deque<unsigned int> v;
3
4  //function to determine all particles within a radius of 3 sigma
5  //stores result in v
6  vicinity(r,3.0*sigma,v);
```

Diese werden dann nacheinander mithilfe der Rechnung 4.8 aktualisiert. Als erstes werden dafür $|x|$ und e_q berechnet:

```
//gaussian random number with 0.3 sigma
grfl_ranx rg;
//calculate |x|
double x = fabs(rg.gauss(0.3*sigma));
//distance between measurement and particle v[i]
double d = r.distance(pf[v[i]]);
//calculate e_q = (r-p)/(|r-p|):
double e_q_x = (r.tell_x() - pf[v[i]].tell_x()) / d;
double e_q_y = (r.tell_y() - pf[v[i]].tell_y()) / d;
```

Als nächstes wird das Zufallsrauschen n berechnet:

```
//dice some random noise with 0.1 sigma
double R = fabs(rg.gauss(0.1*sigma));

//noise has to be circular
double phi = rg.ran1()*2.0*M_PI;

//x- and y-coordinates of noise
double noise_x = R*cos(phi);
double noise_y = R*sin(phi);
```

Nun kann die neue Position p' mithilfe der Gleichung (4.8) aus den berechneten Bestandteilen zusammengesetzt werden:

```
//calculate new position according to: p' = p + |x| e_q + n
double new_x= pf[v[i]].tell_x() + x * e_q_x + noise_x;
double new_y= pf[v[i]].tell_y() + x * e_q_y + noise_y;

//overwrite the position of the particle
pf[v[i]].set_x(new_x);
pf[v[i]].set_y(new_y);
```

Zuletzt wird die Zeit des letzten Resamplings auf die aktuelle Zeit gesetzt.

```
//last update was now
pf[v[i]].set_last_update_time(update_time);
```

4.4. Trackbildung

Von etablierten Cluster Methoden erschien die Affinity Propagation zunächst als die geeignetste, weil sie im Gegensatz zu vielen anderen Methoden in der Lage ist, selbstständig die Anzahl der Tracks zu erkennen. Leider ist sie sehr rechenintensiv und enthält Metaparameter, die geeignet erraten werden müssen.[14] Im Endeffekt erwies sich die Affinity Propagation im Gegensatz zu den Self Organizing Maps(SOMs) als ungeeignet. Aus diesem Grund wird im folgenden nur die Trackbildung über SOMs disktuiert. SOMs sind eine Methode aus dem Unsupervised Machine Learning, deren Ziel es ist, die Inputdaten in mehrere Klassen einzuteilen und somit zu clustern. Dazu werden Neuronen über eine zweidimensionale Fläche verteilt, was in Abbildung 4.4 zu sehen ist. Dabei sind zwei Neuronen in verschiedenen Farben dargestellt und mehrere Input-Partikel.

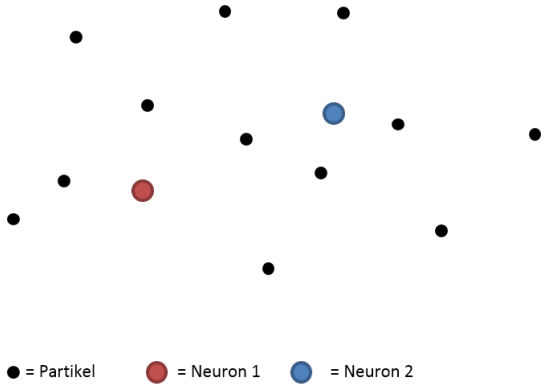


Abbildung 4.4.: Initialisierung der Self Organizing Map

Je näher sich zwei Neuronen sind, desto mehr beeinflussen sie sich gegenseitig. Nach und nach werden nun die einzelnen Inputdaten in zufälliger Reihenfolge ausgewählt. Jedes Neuron bewegt sich auf das Input Datum zu. Es bewegt sich stärker darauf zu, je näher es dem Datum ist. Kommen sich dabei Neuronen zu nahe, fusionieren sie zu einem. Auf diese Weise ist es möglich, dass viele Neuronen zu nur wenigen Tracks fusionieren. Das Verfahren ist konvergiert, wenn die Neuronen aufhören, sich zu bewegen.

Wurde dies hinreichend lange mit allen Inputdaten durchgeführt, entsteht am Ende ein Cluster. Dieses ist in so viele Klassen unterteilt, wie Neuronen übrig geblieben sind. Im Zuge der Iterationen konvergieren diese auf den Track im Zentrum der Partikel. Das Ergebnis des Clustering ist in Abbildung 4.5 zu sehen. [12]

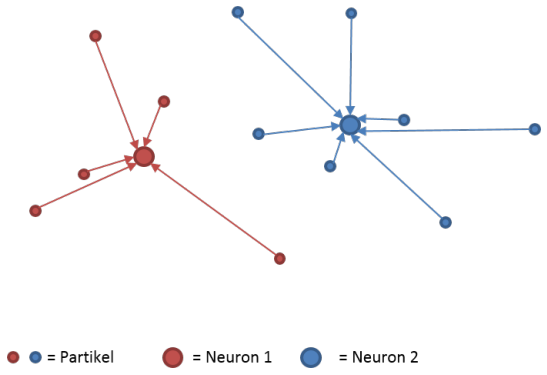


Abbildung 4.5.: Fertiges Cluster

Für die Umsetzung in unserem Anwendungsfall werden nur Partikel berücksichtigt, die erst im letzten Schritt aktualisiert wurden. Damit wird verhindert, dass alte Informationen zur Trackbildung herangezogen werden.

```
//list of particles
std::vector<cl_particle> pf;
//maximum age of particles
int dead_time_limit = 1;

//only accept those particles that have been updated recently
for (auto it=pf.cbegin(); it!=pf.cend(); ++it){
        if ( it->dead_time(t,Delta_t) < dead_time_limit){
                som.push_back(*it);
        }
}
```

Im nächsten Schritt werden gleichmäßig Neuronen im Phasenmodell verteilt. Diese haben einen Abstand von 3· Sigma zueinander. Das Sigma der Self Organizing Map wurde etwas höher festgelegt, als das des Sensors, um zu verhindern, dass Neuronen so zwischen zwei Partikeln liegen, dass diese zwar nur knapp daneben liegen, jedoch nicht trainiert oder gelöscht werden.

```
//define size of the pendulum area
double x_min = -2.0*PI, y_min = -2.0*PI;
double x_max = +2.0*PI, y_max = +2.0*PI;
//noise of the SOM
double SOM_sigma = 1.3 * sensor_sigma;
//list of neurons
std::deque<cl_particle> neurons;

//evenly distribute neurons over area
for (double x=x_min; x < x_max+SOM_sigma; x+=3.0*SOM_sigma){
    for (double y=y_min; y < y_max+SOM_sigma; y+=3.0*SOM_sigma){
        neurons.push_back(cl_particle(x,y));
    }
}
```

Alle Neuronen, welche keinen Partikel in ihrer 3-Sigma Umgebung haben, werden anschließend enfernt. Dadurch kann Rechenzeit gespart werden, da es keine überflüssigen Neuronen mehr gibt.

```
double radius_of_vicinity = 3*SOM_sigma;//size of vicinity
int min_number_of_particles = 1;        //no. of particles needed in vicinity
int particle_counter = 0;               //counter for particles
bool remove_neuron = true;              //gets false if there are enough particles

//check for each neuron, if there are enough particles in vicinity
for (auto n=neurons.begin(); n!=neurons.end(); ++n){
    particle_counter = 0;
```

```
    remove_neuron = true;
    for (auto particle=p.begin(); particle!=p.end(); ++particle){
        //if particle is in vicinity, increase counter
        if ((*n).distance(*particle) < radius_of_vicinity){
            ++particle_counter;
            //break for-loop, if the counter is high enough
            if ( min_number_of_particles <= particle_counter){
                remove_neuron = false;
                break;
            }
        }
    }
    //remove the neuron
    if (remove_neuron){
        (*n) = neurons.back();
        --n;
        neurons.pop_back();
    }
}
```

Bevor das eigentliche Clustern beginnen kann, müssen noch zwei wichtige Variablen deklariert werden. Die erste Variable ist die Lernrate `delta`. Sie beeinflusst, wie stark sich die Neuronen in einem Durchlauf an die richtige Stelle annähern. Je höher die Lernrate ist, desto größer sind die Schritte des Neurons in die Richtung des Partikels. Die Wahl der Lernrate ist sehr wichtig. Ist sie zu hoch, dann konvergiert die Map nicht zu einem endgültigen Ergebnis, sondern verschiebt sich immer weiter. Ist die Lernrate allerdings zu niedrig, dauert es sehr lange bis die Map konvergiert. Beides führt zu ungenauen Ergebnissen. Ein guter Mittelweg ergibt sich, wenn anfangs eine etwas höhere Lernrate gewählt und diese mit jeder Iteration etwas verringert wird. So dauert die Berechnung der Map nicht zu lange und trotzdem konvergiert das Netz schließlich zu einem Ergebnis.

```
//learning rate in the beginning
double delta_0 = 0.1

for(int i=0; i<iterations; i++){
    //a slowly decaying learn rate
    double delta = 100.0 * delta_0 / double( i + 100);
}
```

Die zweite wichtige Variable ist die kumulative Verschiebung der Neuronen `cumul_displacement`. Diese gibt an, wie stark sich die Position der Neuronen noch verändert. Durch die immer niedriger werdende Lernrate werden die Neuronen zu einer bestimmten Position konvergieren. Wenn dies der Fall ist, wird sich die kumulative Verschiebung nicht mehr spürbar ändern. Dann ist die Map fertig berechnet. Die Variable zeigt also an, wenn das Ende der Berechnung erreicht ist. Immer, wenn Neuronen in die Richtung der Partikel verschoben werden, oder mehrere Neuronen zusammengefasst werden, wird die kumulative Verschiebung um die Größe der Verschiebung erhöht.

```
    //distance between old and new Position of the Neuron
    double delta_x, delta_y;

    //keep track of the cumulative displacement
    cumul_displacement += delta_x*delta_x+delta_y*delta_y;

    //abort, when the SOM stops changing
        if (cumul_displacement < 1e-6 ) {
                break;
        }
```

Nun kann mit dem Clustern begonnen werden. Dafür wurden maximal 1000 Iterationen festgelegt, in denen jeweils für alle Partikel die Neuronen aktualisiert werden. Diese Anzahl ist zunächst für den Proof of Concept willkürlich auf "groß genug"gesetzt. Die Aktualisierung wird auch als Training der Neuronen bezeichnet. Die Funktion, die dafür verantwortlich ist, bekommt als Input den betrachteten Partikel, das Sigma der Neuronen und die Lernrate übergeben. Der einzige Rückgabewert ist die kumulative Verschiebung. Diese wird zu dem derzeitigen Wert dazu addiert.

```
//p: list of all considered particles
//train each training data
for (auto it=p.cbegin(); it!=p.cend(); ++it) {
    cumul_displacement += train_neurons(*it,sigma,Delta);
}
```

Innerhalb der Funktion wird für jedes Neuron der Abstand zum betrachteten Partikel berechnet. Liegt der Partikel innerhalb der 3-Sigma Umgebung des Neurons, bewegt sich das Neuron einen Schritt auf den Partikel zu. Wie groß dieser Schritt ist, ist abhängig von der Lernrate und der Distanz zwischen Partikel und Neuron. Nachdem das Neuron auf die neue Position gesetzt wurde, wird die kumulative Verschiebung um die Größe des Schritts erhöht. Sobald die Berechnung für alle Neuronen abgeschlossen ist, wird die kumulative Verschiebung zurückgegeben.

```
double cl_som_cluster::train_neurons(const cl_particle& td, double sigma, double delta){
    //cumulative displacement
    double cumul_displacement = 0.0;

    //train the neurons
    for (auto it=neurons.begin(); it!=neurons.end(); ++it) {
        //distance between particle and neuron
        double d = it->distance(td);

        //if the training data is within the 3 sigma environment of a neuron update neuron
        if ( (d/sigma) < 3.0 ) {
            //distance in x- and y-direction
            double delta_x = td.tell_x()-(*it).tell_x();
            double delta_y = td.tell_y()-(*it).tell_y();


```

```
17
18              //the smaller the distance between the particle and the neuron,
19              //the larger the step should be
20              double V_x = delta_x * exp(-0.5*d*d);
21              double V_y = delta_y * exp(-0.5*d*d);
22
23              //size of the step also depends on the learning rate delta
24              delta_x = delta * V_x;
25              delta_y = delta * V_y;
26
27              //update the position of the neuron
28              (*it).set_x((*it).tell_x() + delta_x);
29              (*it).set_y((*it).tell_y() + delta_y);
30
31              //keep track on the cumulative displacement
32              cumul_displacement += delta_x*delta_x+delta_y*delta_y;
33              }
34      }
35      return(cumul_displacement);
36  }
```

Nachdem das Update aller Neuronen erfolgt ist, kann es passieren, dass einige davon sehr nah zusammenliegen. Um einen doppelten Rechenaufwand ohne Mehrwert zu vermeiden, werden diese zu einem Neuron zusammengefasst. Dies geschieht, indem eines der beiden Neuronen in die Mitte zwischen ihnen gesetzt wird. Das andere Neuron wird gelöscht. Auch hier wird die kumulative Verschiebung aktualisiert.

```
1   //minumum distance between two neurons: the 2-Sigma environments touch
2   double min_distance = 4.0*sigma
3
4   //cumulative displacement
5   double cumul_displacement = 0.0;
6
7   //get distance between each of the neurons
8   for (auto n1=neurons.begin(); n1!=neurons.end(); ++n1) {
9       for (n2=n1+1; n2!=neurons.end(); ++n2) {
10          //fuse neurons, which are closer than min_distance
11          if ( n1->distance(*n2) < fabs(min_distance)) {
12              //calculate middle between n1 and n2
13              double new_x = 0.5* ((*n1).tell_x() + (*n2).tell_x());
14              double new_y = 0.5* ((*n1).tell_y() + (*n2).tell_y());
15
16              //keep track of the cumulative displacement
17              double delta_x = new_x - (*n1).tell_x();
18              double delta_y = new_y - (*n1).tell_x();
19              cumul_displacement += delta_x*delta_x+delta_y*delta_y;
20
21              //set n1 in the middle between n1 and n2
22              (*n1).set_x( new_x);
23              (*n1).set_y( new_y);
24
```

```
25
26              //remove n2
27              //set n2 as the last neuron in the list
28              (*n2) = neurons.back();
29              //remove the last neuron
30              neurons.pop_back();
31
32              //keep n2 on the neuron after n2 in the for()
33              --n2;
34          }
35      }
36  }
```

Bei jeder hundertsten Iteration werden alle Neuronen entfernt, die weniger als drei Partikel in ihrer 3-Sigma Umgebung haben, um nicht unnötige Rechenzeit zu verbrauchen. Am Ende jeder Iteration wird geprüft, wie stark sich die Neuronen bewegt haben. Ist die Kumulative Verschiebung kleiner als 10^{-6}, ändert sich das SOM nichtmehr und das Clustern wird beendet. Ist dies nicht der Fall, so wird mit der nächsten Iteration weitergemacht. Nachdem die letzte Iteration vorbei ist oder das Clustern abgebrochen wurde, werden noch einmal alle Neuronen entfernt, in dessen 3-Sigma Umgebung sich kein Partikel befindet. Anschließend werden die Cluster der Neuronen extrahiert. Dies bedeutet, dass jeder Partikel dem Neuron zugeordnet wird, in dessen 3-Sigma Umgebung er liegt.

```
1   for (auto it=neurons.cbegin(); it!=neurons.cend(); ++it) {
2       //for every neuron, there is a cluster
3       clusters.push_back(*(new std::deque<cl_particle>));
4
5       //assign all particles to the neuron, which are in the 3 sigma environment
6       for (auto jt=particles.cbegin(); jt!=particles.cend(); ++jt) {
7           if (it->distance(*jt) < 3*sigma){
8               clusters.back().push_back(*jt);
9           }
10      }
11  }
```

Dadurch, dass immer wieder unwichtige Neuronen gelöscht werden, bleiben am Ende des Algorithmus nur noch wenige davon übrig. Die Anzahl an Neuronen steht für die Anzahl an Tracks, die sich innerhalb der betrachteten Fläche befinden. Diese Neuronen sind so platziert, dass sie eine zusammengehörige Gruppe an Partikeln repräsentieren. Die Position der Neuronen gibt also die Position der gesuchten Objekte wieder.
Somit ist die Berechnung der Self Organizing Map der letzte Schritt des Trackings, in dem alle Informationen gebündelt werden und der Output des Partikelfilters berechnet wird.

4.5. Visualisierung

Für die Visualisierung des Partikelfilters wird die Vektorgrafik-Sprache „Asymptote“ verwendet. Sie bietet die Möglichkeit, koordinatenbasierte Zeichnungen zu erstellen und ist daher für den Zweck der Bachelorarbeit gut geeignet. [2] Um die Visualisierung, welche in Grafik 4.6 abgebildet ist, mit „Asymptote“ zu erreichen, werden zusätzlich drei Funktionen implementiert.

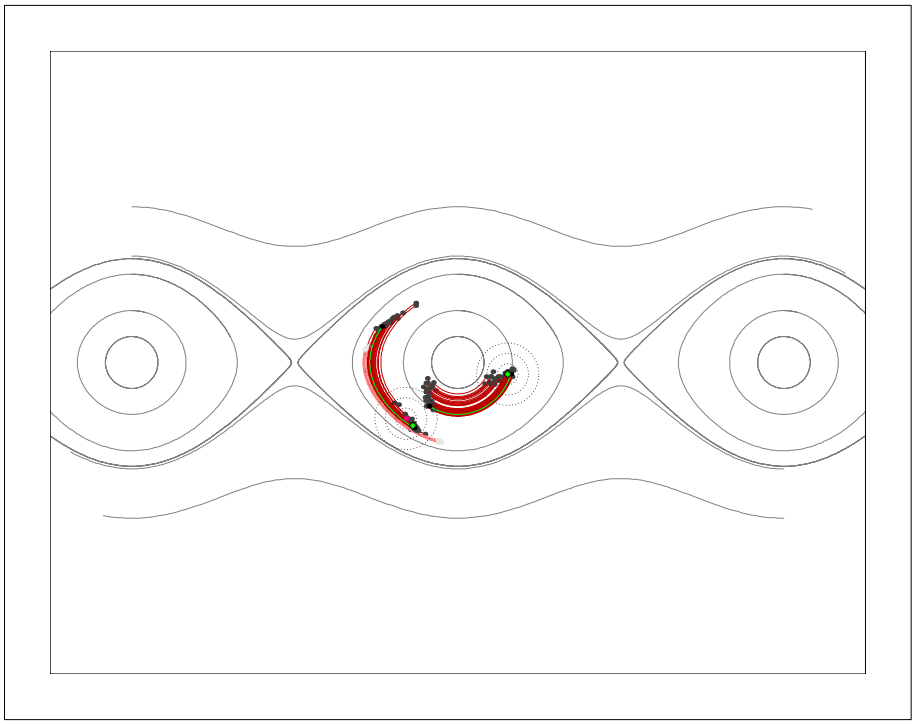


Abbildung 4.6.: Visualisierung des Partikelfilters
In grün sieht man die Truth Trajektorie. Sie wird im Uhrzeigersinn durchlaufen. Die Partikel bzw. ihre Verteilung reräsentieren die Messung und deren Ungenauigkeit. Zustände d.h. Partikel werden über die Integration der Bewegungsgleichung des Pendels in die Zukunft vorhergesagt.

1. `solution_to_asypath()`
   Diese Funktion konvertiert eine Pendelbahn in einen String, welcher eine koordinatenbasierte Beschreibung der Bahn darstellt. Der String wird so aufgebaut, dass er als „Asymptote“-Befehl verarbeitet werden kann. Einzelne aufeinanderfolgende Koordinaten der Pendelbahn werden dabei wie folgt aneinandergereiht:
   path *Name*=(x_0, y_0)- -(x_1, y_1)- -...- -(x_N, y_N);

2. `background_trajectories()`
   Diese Funktion berechnet die sechs Pendelbahnen, auch Trajektorien genannt, welche im Hintergrund zu sehen sind. Dafür werden verschiedene Startkoordinaten mit dem RK4-Verfahren so integriert, dass sich die Pendelbahnen bilden. Die äußeren Trajektorien, welche das Durchschwingen des Pendels darstellen, werden je zweimal integriert. Einmal für die Darstellung der oberen Schwingung und einmal für die der unteren Schwingung. Dafür wird die Startkoordinate in x- und y-Richtung negiert. Um die äußeren Pendelbahnen vollständig darstellen zu können, ist es notwendig, dass die vier inneren Trajektorien jeweils dreimal integriert werden. Somit sind sie in der Grafik dreimal zu sehen.

Der Startpunkt für die Integration wird dann auf der x-Achse jeweils um 2π nach rechts beziehungsweise links bewegt. Bei der Implementierung der Berechnung einer Trajektorie müssen zunächst einige Integrationsparameter festgelegt werden, welche im folgenden Code-Abschnitt aufgeführt werden.

```
//integration parameters
static constexpr double t_start = 0.0;          //starting time
static constexpr double t_step = 0.03;          //integration step width
grfl_vector y0 = std::vector<double>{0,0.5};    //starting position
cl_ode_solution solution = cl_ode_solution();   //save solution of integration
cl_odesolver_rk4 rk4 = cl_odesolver_rk4(y0.size(),t_step); //init rk4 method solver
```

Als nächstes wird die Startposition bis zu einem Abbruchkriterium integriert, welches hier das Erreichen der Zeiteinheit 13 ist, da die Trajektorie dann vollständig beerechnet ist. Die Lösung wird einer Liste hinzugefügt, in der alle Trajektorien gespeichert sind. Bevor die nächste Integration beginnt, müssen die Lösungsvariable wieder zurückgesetzt und die Startvariable `y0` aktualisiert werden.

```
//inner circle in the middle
//integrate: starting position y0, starting time t_start,
//use RK4-method to calculate a pendulum, end if t=13.0
solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(13.0));

//add solution to list of trajectories
traj.push_back(solution);

//reset solution variable
solution.clear();

//update start vector for inner left circle
y0 = std::vector<double>{-2.0*M_PI,0.5};
```

3. `asy()`

Diese Funktion übernimmt die eigentliche Visualisierungsarbeit. Dabei wird beschrieben, welche Objekte in dem Bild, wie dargestellt werden sollen. Diese Informationen werden dann in eine .asy-Textdatei geschrieben, welche daraufhin das fertige Bild erzeugt. Zunächst wird pdflatex als Dateiformat des fertigen Bildes eingestellt und die Größe des Bildes auf 20 Zentimeter festgelegt.

```
//set filename to pf_000.asy, pf_001.asy,...,pf_030.asy(One for each iteration)
ostringstream filename;
filename.fill('0');
filename<< "pf_" << std::setw(3) << k << ".asy";

//open .asy-file
fstream        fileout;
fileout.open(filename,ios::out);


```

```
11 //general settings
12 fileout << "size(20cm);"<<endl;
13 fileout << "settings.tex=\"pdflatex\";"<<endl;
```

Im zweiten Schritt wird der Hintergrund gezeichnet. Dafür werden die Trajektorien, welche zuvor mithilfe der bereits erklärten Funktion `background_trajectories` berechnet und in einer Liste namens `traj` gespeichert wurden, in grau dargestellt.

```
1  //counter needed for name of trajectory paths
2  unsigned int counter=1;
3
4  //for each trajectory
5  for (auto t=traj.cbegin(); t!=traj.cend(); ++t,++counter){
6      //set name of trajectory to p1, p2...p8
7      ostringstream name;
8      name<< "p" << counter;
9      //make asy-string out of trajectory and write it into file
10     fileout << solution_to_asypath(name.str(),*t) << endl;
11     //write asy draw-command for the trajectory into file
12     fileout << "draw(" <<name.str() << ",gray);" << endl;
13 }
```

Als nächstes werden die einzelnen Partikel eingezeichnet. Diese werden in vier unterschiedlich hellen Grautönen dargestellt. Je dunkler das Grau, desto weniger Zeitschritte sind seit dem letzten Update des Partikels vergangen. Jeder Partikel wird in der Grafik zweimal dargestellt. Einmal auf seiner derzeitigen Position und einmal an der Stelle, an der er sich im nächsten Zeitschritt befinden wird. Die beiden Punkte werden durch einen Track verbunden, der in der Abbildung in Rot dargestellt ist. Die Helligkeit des Rottons entspricht der des Grautons.

```
1  //counter needed for name of trajectory paths
2  counter = 0;
3
4  //list of timesteps since last update of each particle
5  auto d = res.dead_times().cbegin();
6
7  //color of the pen
8  std::string pen = "red";       //red for trajectories
9  std::string pend = "gray";     //grey for particles
10
11 //for each particle, draw dots and trajectory
12 for (auto t=res.traj().cbegin(); t!=res.traj().cend() && d!=res.dead_times().cend();
13         ++t,++d,++counter){
14
15     //name for the trajectory part0, part1...
16     ostringstream name;
17     name<< "part" << counter;
18
19
```

```
20      //make asy-string out of trajectory and write it into file
21      fileout << solution_to_asypath(name.str(),*it)<<endl;
22
23      //get the right color based on time since last update
24      switch ( (*d)){
25          case 0:
26              pen = "heavyred";
27              pend = "heavygray";
28              break;
29          case 1:
30              pen = "red";
31              pend = "gray";
32              break;
33          case 2:
34              pen = "lightred";
35              pend = "lightgray";
36              break;
37          case 3:
38              pen = "palered";
39              pend = "palegray";
40      }
41
42      //write asy draw-command for trajectory
43      fileout << "draw("<<name.str()<<","<<pen<<");"<<endl;
44      //write asy draw-command for position of particle in this time step
45      fileout << "dot(( "<<(*it).front()[0]<<","<<(*it).front()[1]<<"),"<<pend<<");";
46      //write asy draw-command for position of particle in next time step
47      fileout << "dot(( "<<(*it).back()[0]<<","<<(*it).back()[1]<<"),"<<pend<<");";
48  }
```

Der tatsächliche Aufenthaltsort des Pendels wird durch einen schwarzen Punkt gekennzeichnet, der durch eine grüne Linie mit dem Aufenthaltsort im nächsten Zeitschritt verbunden ist. Die Berechnung des Partikelfilters wird durch ein grünes Quadrat visualisiert. Nachdem der Track einen Namen erhalten hat und zu einem String umgewandelt wurde, wird der Zeichenbefehl in die Datei geschrieben:

```
1   //write asy draw-command for...
2
3   //...truth as black dots
4   fileout << "dot(( " << (*it).front()[0]<< ","<< (*it).front()[1]<<"),black);"<<endl;
5   fileout << "dot(( " << (*it).back()[0]<< ","<< (*it).back()[1]<<"),black);"<<endl;
6
7   //...green trajectory of truth
8   fileout << "draw("<<pathname.str()<<",green);"<<endl;
9
10  //calculated position as green square
11  fileout << "filldraw(shift(("<< it->tell_x() << ","<< it->tell_y()
12      <<"))*scale(0.1)*shift((-0.5,-0.5))*unitsquare,green+linewidth(2));"<<endl;
```

Nun müssen nur noch die Messungen der Sensoren eingezeichnet werden. Diese werden durch einen pinken Punkt dargestellt. Um die Messung herum werden drei gepunktete Kreise gezeichnet. Diese stellen die 1-Sigma, 2-Sigma und 3-Sigma Umgebungen dar.

```
//draw the measurements: dot and 1,2,3 sigma environment
auto s = res.sigmas().cbegin();  //iteration over List of 3 sigmas

//draw for each measurement...
for (auto m=res.measurements().cbegin(); m!=res.measurements().cend()
        && s!=res.sigmas().cend();++m,++s){
    //...measurement as pink dot
    fileout << "dot(("<< m->tell_x() << "," << m->tell_y() << "),fuchsia);" << endl;

    //...1-Sigma environment of measurement black dotted
    fileout << "draw(shift(("<< m->tell_x() << "," << m->tell_y() <<
        "))*scale("<<(*s)<<")*unitcircle,black+dotted);"<<endl;

    //...2-Sigma environment of measurement black dotted
    fileout << "draw(shift(("<< it->tell_x() << "," << it->tell_y() <<
        "))*scale("<<2.0*(*kt)<<")*unitcircle,black+dotted);"<<endl;

    //...3-Sigma environment of measurement black dotted
    fileout << "draw(shift(("<< it->tell_x() << "," << it->tell_y() <<
        "))*scale("<<3.0*(*kt)<<")*unitcircle,black+dotted);"<<endl;
}
```

Zuletzt wird der Bereich des Koordinatensystems festgelegt, der in der Grafik abgebildet werden soll. Dieser beträgt 2.5π in die negative und positive x-Richtung und den Wert 6 in die negative und positive y-Richtung. Der Rest wird abgeschnitten.

```
//Set size of the boundingbox
fileout << "real a=2.5*pi;"<<endl;
fileout << "real b=6;"<<endl;

//write asy-draw command for the boundingbox into the file
fileout << "path mybbox=yscale(2*b)*xscale(2*a)*shift(-0.5,-0.5)*unitsquare;"<<endl;

//draw the boundingbox
fileout << "draw(mybbox);" << endl;

//do the clipping
fileout << "clip (mybbox);"<<endl;

//set border size
fileout << "shipout(bbox(1cm));" << endl;

//close file
fileout.close();
```

Tabelle 4.1 gibt zusammenfassend eine Übersicht darüber, wie die verschiedenen Elemente visualisiert werden.

Element	**Visualisierung**
Berechnete Positionen der Objekte durch den Algorithmus	Grüne Quadrate
Messungen der Sensoren	Pinke Punkte
Partikel im jetzigen und im nächsten Zeitschritt	Graue Punkte, je dunkler desto aktueller
Partikeltracks vom jetzigen zum nächsten Zeitschritt	Rote Linien, je dunkler desto aktueller
Sigma Umgebungen der Messungen	Schwarz gepunktete Kreise
Tatsächlicher Aufenthaltsort der Objekte im jetzigen und im nächsten Zeitschritt	Schwarze Punkte
Tatsächlicher Track der Objekte vom jetzigen zum nächsten Zeitschritt	Grüne Linien
Trajektorien im Hintergrund	Graue Linien

Tabelle 4.1.: Visualisierung der einzelnen Elemente

5. Durchführung von Tests

In Kapitel 3 wurde ein Konzept zur Umsetzung eines Partikelfilters erarbeitet. Dabei wurden die drei Leitfragen und die Anforderungen, die sich dadurch für die Umsetzung des Anwendungsbeispiels ergeben, vorgestellt. In diesem Kapitel werden Tests definiert, durch welche geprüft werden soll, ob alle Anforderungen erfüllt werden konnten. Zunächst wird für jede der Fragen untersucht, welche Anforderungen bereits durch die Implementierung abgedeckt wurden. Daraufhin wird für die restlichen Anforderungen festgelegt, welche Tests möglich sind, um deren Erfüllungsgrad zu beurteilen. Die Auswertung der Tests erfolgt in Kapitel 6.

1. **Ist ein Partikelfilter geeignet, um eine Sensordatenfusion für unterschiedliche Tracks mit verschiedenen Sensoren im Phasenraum zu realisieren?**
   Die erste Anforderung, die erfüllt sein muss, um diese Frage beantworten zu können, ist, dass der Input des Partikelfilters mehrere Sensoren umfasst. Dies wurde bereits in der Implementierung umgesetzt, da dort abwechselnd die Messungen von zwei verschiedenen Sensorobjekten mit unterschiedlicher Messungenauigkeit erfasst wurden.

   Für die zweite Anforderung muss es möglich sein, mehrere Tracks als Output zu erhalten. Auch dies wurde in der Implementierung bereits gelöst. Die Klasse PF_results, welche die Lösung des Partikelfilters darstellt, besitzt als Attribut eine Liste, welche eine beliebige Anzahl an Tracks speichern kann. Diese wird in der Klasse Particle_filter mit den Tracks befüllt, die in der Klasse SOM_Cluster gefunden wurden.

   Auch die letzte Anforderung wird bereits bei der Implementierung erfüllt. Diese sagt aus, dass die Inputdaten fusioniert werden müssen, um den Output zu erhalten. Diese Bedingung wird eingehalten, da die Position der Partikel durch die Messung der Sensoren beeinflusst wird, indem die Partikel in der 3-Sigma Umgebung ein Update bekommen. So werden sie etwas näher an die Messung verschoben und innerhalb der nächsten drei Schritte nicht gelöscht. Außerdem werden neue Partikel in der 2-Sigma Umgebung der Messung hinzugefügt. Die Trackbildung wiederum basiert auf der Position der Partikel, da sich die trackbildenden Neuronen nur dort befinden, wo auch Partikel sind. Somit wird der Output durch den Input generiert. Insgesamt muss also für die Bestätigung der Anforderungen, die sich durch die erste Leitfrage ergeben, kein Test mehr durchgeführt werden.

2. **Wie lässt sich eine Trackbildung realisieren?**
   Die erste Anforderung, die sich durch diese Leitfrage ergibt, ist, dass die Anzahl der Tracks von dem Algorithmus richtig erkannt wird. Dies ist allein durch die Implementierung noch nicht ersichtlich. Um zu testen, ob diese Anforderung erfüllt ist, wird die Anzahl der Tracks stufenweise verändert. Dies geschieht, indem der Liste mit den Startwerten y0 Elemente hinzugefügt oder gelöscht werden. Für einen bis fünf Tracks werden jeweils zehn Durchläufe gemacht, um zu sehen, ob der Algorithmus die Menge an Tracks richtig identifizieren kann.

Auch die letzte Anforderung, die Durchführung von „Split Track“ und „Merge Track“ Events, soll hier auf seine Erfüllung getestet werden.

Die zweite Anforderung besagt, dass die Tracks von dem Algorithmus an der richtigen Stelle erkannt werden müssen. Auch das ist durch die Implementierung allein noch nicht bestätigt und muss erst getestet werden. Der Track, der vom Algorithmus bestimmt wird, befindet ich an der richtigen Stelle, wenn er sich in der 1-Sigma Umgebung des tatsächlichen Tracks befindet. Der Wert des Sigmas wurde auf 0.15 Meter festgelegt. Als Grundlage für diesen Test dienen die Ergebnisse aus dem letzten Test. Die Testfälle werden genauso übernommen, mit dem Unterschied, dass nicht die richtige Anzahl der Tracks, sondern deren Genauigkeit entscheidend ist.

Insgesamt müssen also noch zwei Tests durchgeführt werden, um zu bestimmen, ob die Anforderungen durch den Trackbildungs-Algorithmus erfüllt und somit die Leitfrage beantwortet wurde.

3. **Wie hängt die Sensorgenauigkeit mit dem Fusionsergebnis zusammen?**
   Die erste Frage, die sich durch diese Leitfrage ergibt, ist, ab welcher Sensorungenauigkeit ein korrektes Tracking nicht mehr möglich ist. Dies kann getestet werden, indem die Sensorungenauigkeit schrittweise erhöht wird. So wird irgendwann eine Grenze erreicht, bei der der berechnete Track standardmäßig außerhalb der Sigma Umgebung des tatsächlichen Tracks liegt. Ab dieser Ungenauigkeit ist ein korrektes Tracking nicht mehr möglich. Zunächst wird nur ein Sensor verwendet. Die Sensorungenauigkeit startet bei 0.2 Metern und wird dann schrittweise um 0.1 Meter erhöht, bis eine Ungenauigkeit von 0.7 Metern erreicht ist. Für jede Ungenauigkeitsstufe werden fünf Testdurchläufe gemacht, um sicherzustellen, dass ein Ergebnis nicht nur zufällig schlecht oder gut ist. Nachdem dies mit einem Sensor getestet wurde, wird ein zweiter Sensor hinzugefügt. Beide Sensoren starten wieder bei einer Ungenauigkeit von 0.2 Metern. Zuerst wird nur ein Sensor schrittweise um 0.1 Meter bis zu einer Grenze von 0.7 Metern ungenauer, der andere bleibt gleich. Danach wird der zweite Sensor schrittweise um 0.1 Meter ungenauer und der andere bleibt bei hoher Ungenauigkeit. Im dritten Durchlauf werden beide Sensoren gleichmäßig schrittweise wieder von 0.2 Meter auf 0.7 Meter verschlechtert. Bei diesen Tests soll außerdem darauf geachtet werden, ob sich die Ergebnisse verbessern, je mehr Iterationen der Partikelfilter durchläuft oder ob sie gleichbleibend sind.

   Die zweite Frage ist, welche Auswirkungen ein sehr genauer Sensor auf das Ergebnis hat. Dazu wird ein Vorgehen verwendet, das dem aus dem letzten Test ähnelt. Zunächst wird wieder nur ein Sensor verwendet, welcher bei einer Ungenauigkeit von 0.2 Metern startet. Diese wird um 0.05 Meter verringert, bis eine Genauigkeit von 0.01 Meter erreicht ist. Eine Ungenauigkeit von 0 Metern wird von dem Algorithmus nicht zugelassen. Anschließend wird ein zweiter Sensor hinzugefügt. Beide starten bei einer Ungenauigkeit von 0.2 Metern. Die Ungenauigkeit des ersten Sensors wird schrittweise herabgesetzt bis 0.01 Meter erreicht sind, der andere bleibt gleich. Danach wird der zweite Sensor heruntergesetzt und der erste Sensor bleibt bei einer Ungenauigkeit von 0.01 Metern. Im letzten Schritt werden beide Sensoren gleichmäßig von 0.2 Meter auf 0.01 Meter herabgesetzt. Auch bei diesen Tests werden für jede Ungenauigkeit fünf Durchläufe gemacht.

Als letztes soll getestet werden, wie gut das Fusionsergebnis bei durchschnittlicher Sensorungenauigkeit ist. Hierzu werden fünfzig Durchläufe mit zufälligen Ungenauigkeiten zwischen 0.1 Metern und 0.5 Metern gemacht. Es werden bei allen Durchläufen beide Sensoren verwendet. Es wird ermittelt, um welchen Wert das Ergebnis im Durchschnitt von der realen Position abweicht und ob nach mehreren Iterationen eine Verbesserung festzustellen ist.

Insgesamt werden somit drei Tests benötigt, um die Leitfrage beantworten zu können.

6. Ergebnisse

- **Test 1: Richtige Anzahl an Tracks**
  Bei diesem Test wurde die Anzahl der Tracks zwischen eins und fünf variiert. Das Ziel dieses Tests war es, herauszufinden, ob die Anzahl an Tracks durch den Algorithmus richtig festgestellt wird. Die Erkennung der richtigen Menge ist beispielhaft für fünf Tracks in Abbildung 6.1 dargestellt. Welche Elemente wie visualisiert sind, kann der Übersicht in Tabelle 4.1 entnommen werden.

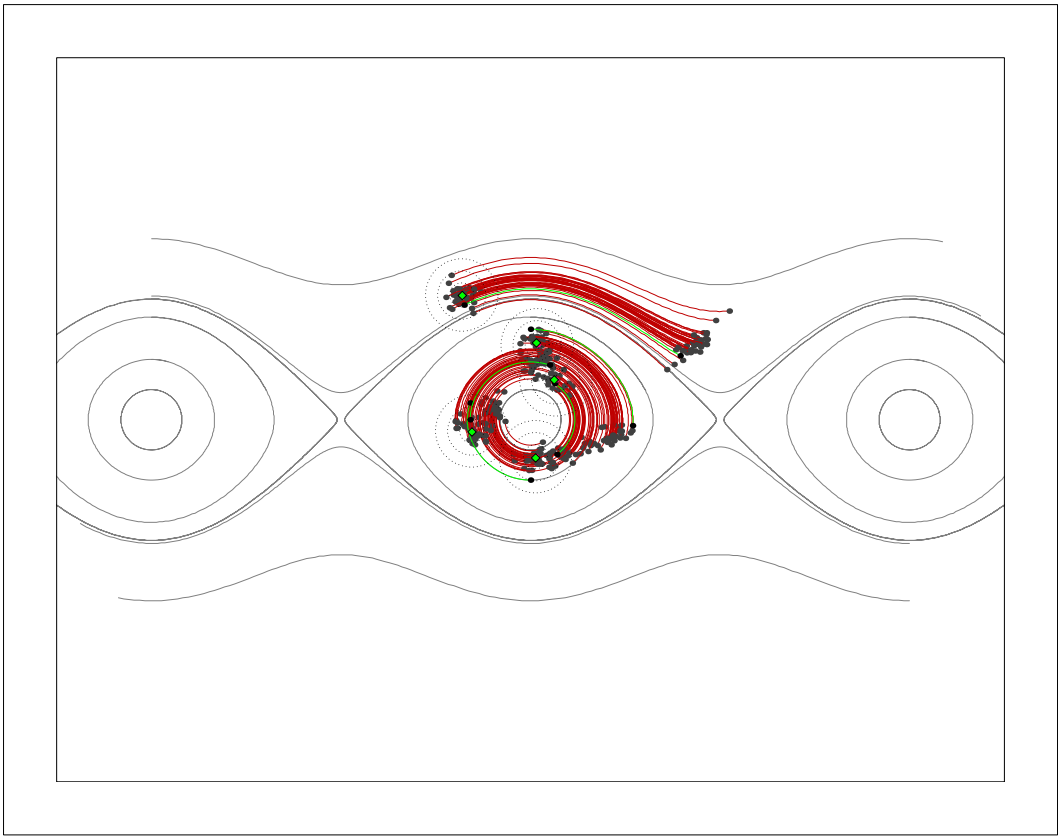


Abbildung 6.1.: Erkennung der richtigen Anzahl an Tracks

Insgesamt wurden für jede Anzahl zehn Durchläufe gemacht. Jeder der insgesamt fünfzig Durchläufe wurde ausgewertet, indem die Anzahl der berechneten Tracks auf jedem Bild gezählt wurde. Pro Durchlauf wurden dreissig Bilder generiert, was dazu führt, dass insgesamt 1500 Bilder verwertbar sind. Insgesamt wurde auf 1294 Bildern die richtige Anzahl an Tracks berechnet, was etwa einer Erfolgsquote von 86,3 % entspricht. Heruntergebrochen auf die einzelnen Mengen an Tracks, ist zu beobachten, dass die Fehlerrate steigt, je größer die Anzahl an Tracks ist, die berechnet werden soll. Bei einem einzelnen Track liegt die Erfolgsrate bei 99 %, wohingegen es bei fünf Tracks nur 76.4 % sind. Ein möglicher Grund dafür liegt darin, dass die Tracks bei einer höheren Anzahl automatisch näher zusammenliegen.

Da die Neuronen bei der Trackbildung gleichmäßig über die gesamte Fläche verteilt sind, ist es sehr wahrscheinlich, dass die Partikel der entsprechenden Tracks dem gleichen Neuron zugeordnet werden. Dadurch werden sie zu einem gemeinsamen Track zusammengefasst.

Unterstützt wird diese Hypothese dadurch, dass die Anzahl der Tracks bei einer größeren Menge deutlich öfter zu niedrig als zu hoch berechnet wird. Bei einem einzelnen Track liegt die Fehlerrate für eine zu niedrige Anzahl bei 0.35 % und für eine zu hohe Anzahl bei 0.65 %. Es wird hier also eher eine zu hohe Anzahl berechnet. Auch bei zwei Tracks wird mit einer Fehlerquote von 3.7 % mehr als doppelt so oft eine zu hohe als eine zu niedrige Anzahl berechnet, welche hier bei 1.6 % der Fälle auftritt. Dies ändert sich sprungartig ab einer Anzahl von drei Tracks. In 6 % der Fälle wird die Anzahl der Tracks hier zu hoch geschätzt. Mehr als doppelt so oft, in 15.6 % der Fälle, wird die Anzahl der Tracks zu niedrig geschätzt. Bei fünf Tracks wird die Anzahl knapp fünfmal so oft zu niedrig als zu hoch geschätzt. Eine Zusammenfassung der Ergebnisse ist in Tabelle 6.1 zu sehen.

Anzahl Tracks	Korrektes Ergebnis	Ergebnis zu hoch	Ergebnis zu niedrig
1	99 %	0.65 %	0.35 %
2	94.7 %	3.7 %	1.6 %
3	78.4 %	6 %	15.6 %
4	83 %	7 %	10 %
5	76.4 %	4 %	19.6 %

Tabelle 6.1.: Ergebnisse aus Test 1

Umgekehrt bedeutet dies auch, dass eine zu hohe Anzahl an Tracks geschätzt wird, wenn Partikel, die zu einem Track gehören zu weit auseinander liegen. Dadurch werden sie bei der Trackbildung zwei verschiedenen Neuronen zugeordnet und es entstehen zu viele Tracks. Eine zu hohe Distanz zwischen zwei Partikeln kann durch zwei Gründe entstehen. Zum einen besteht die Möglichkeit, dass der verwendete Sensor sehr ungenau ist. Dies führt dazu, dass die 3-Sigma Umgebung größer wird und somit mehr Partikel aktualisiert werden. Die breite Fächerung der Partikel führt dazu, dass sie verschiedenen Neuronen zugeordnet werden und somit zu viele Tracks entstehen. Der zweite Grund liegt in der Beschaffenheit des Phasenmodells. Die Pendelbahnen werden von den instabilen Ruhepositionen angezogen, wodurch sich die elliptische Form der äußeren Kreisbahnen ergibt. Dies führt dazu, dass zwei nebeneinanderliegende Bahnen an den seitlichen Rändern außeinandergezogen werden und sich ihr Abstand vergrößert. Die Partikel werden also an den seitlichen Rändern immer etwas auseinandergezogen und es wird somit wahrscheinlicher, dass sie verschiedenen Neuronen zugeordnet werden. Dadurch werden mehrere Tracks, statt nur einem gebildet.

Dieses Problem kann gelöst werden, indem die Neuronen bei der Trackbildung nicht gleichmäßig über die gesamte Fläche verteilt werden, sondern beispielsweise gaußförmig. Dadurch sind in der Mitte, wo die Abstände zwischen den Partikeln tendenziell geringer sind, mehr Neuronen. Das führt dazu, dass die Partikel nicht denselben Neuronen zugeteilt werden und die Anzahl der Tracks nicht mehr zu gering geschätzt wird. An den äußeren Stellen, wo die Partikel auseinandergezogen werden, sind weniger Neuronen vorhanden und die Partikel werden dem gleichen Neuron zugeordnet. Dadurch werden seltener zu viele Tracks berechnet.

Zusätzlich sollten weniger Neuronen auf der Fläche verteilt werden, wenn die Sensorungenauigkeit höher ist, da zusammengehörige Partikel dort weiter gestreut sind. Je höher die Anzahl der Tracks ist, desto mehr Neuronen sollten jedoch hinzugefügt werden, da es dann leichter wird, die Partikel der verschiedenen Tracks den verschiedenen Neuronen zuzuordnen. Es gibt also keine perfekte Lösung für die Anpassung der Neuronenanzahl, sondern es muss ein Kompromiss für die gegebenen Bedingungen eingegangen werden.

Zusammenfassend zeigt der Test, dass die Anzahl der Tracks in 86 % der Fälle richtig erkannt wird. Durch die vorgeschlagenen Optimierungen kann dies noch deutlich verbessert werden. Die Anforderung, dass die Anzahl der Tracks richtig erkannt wird, ist somit erfüllt. Außerdem zeigt der Test, dass die Anzahl der Tracks in jedem Zeitschritt neu berechnet wird. Somit sind durch den Algorithmus auch die Ereignisse „Split Track“ und „Merge Track“ durchführbar.

- **Test 2: Richtige Position der Tracks**
  In diesem Test sollte festgestellt werden, wie präzise die berechneten Positionen der Tracks sind, die für Test 1 generiert wurden. Für jede Anzahl an Tracks wurde untersucht, wie groß die Distanz zwischen realen und berechneten Standpunkten ist. Diese Abstände wurden in fünf Stufen untergliedert:
  - **Stufe 0:** Berechnung entspricht dem realen Standpunkt
  - **Stufe 1:** Berechnung liegt innerhalb des 1-Sigma Radius um den realen Standpunkt
  - **Stufe 2:** Berechnung liegt innerhalb des 2-Sigma Radius um den realen Standpunkt
  - **Stufe 3:** Berechnung liegt innerhalb des 3-Sigma Radius um den realen Standpunkt
  - **Stufe 4:** Berechnung liegt außerhalb des 3-Sigma Radius um den realen Standpunkt

  Der Test zeigt, dass die Berechnung im Durchschnitt im 0.73 Sigma Radius der realen Position liegt. Dies bedeutet, dass das Ergebnis des Partikelfilters im Schnitt eine geringere Abweichung als 0.11 Meter hat. Zum Vergleich: der innerste Kreis des Phasenmodells in der Visualisierung hat einen Durchmesser von einem Meter. Die genutzen Sensoren hatten eine Standardabweichung von 0.1 und 0.2 Metern. Das zeigt, dass der Algorithmus den schlechteren Sensor ausgleichen konnte, sodass das Ergebnis etwa genauso gut ist, wie das des besseren Sensors. Außerdem bedeutet das, dass Fehlinformationen oder Messfehler der Sensoren durch den Partikelfilter abgedämpft wurden. Im Verlauf der einzelnen Durchgänge ist zu sehen, dass die Stellen, an denen die Ergebnisse schlechter als der Durchschnitt sind, Stellen sind, an denen die Messung des Sensors ebenfalls schlecht war. Die Grafik 6.2 stellt den Abstand der Messpunkte und der berechneten Ergebnisse zur tatsächlichen Position des Objektes über 30 Zeitschritte dar. Der Abstand wurde mithilfe der verschiedenen Stufen beurteilt. Die blaue Linie zeigt die Entfernung der Messung zur wahren Pendelposition und die rote Linie den Abstand der berechneten Positionen zur Pendelposition. Die gepunkteten Linien zeigen den jeweiligen durchschnittlichen Wert.

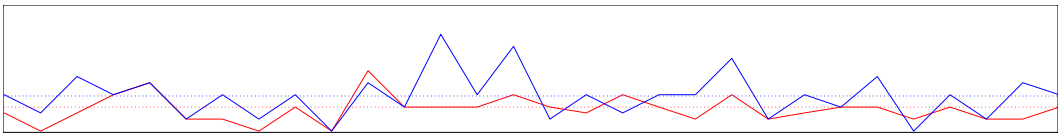


Abbildung 6.2.: Abstand von Messpunkt(blau) und Ergebnis(rot) zur wahren Position

Werden die jeweiligen Ergebnisse der verschiedenen Track-Anzahlen betrachtet, ist zu sehen, dass sich die Durchschnittswerte kaum voneinander unterscheiden. Im Mittel liegen die berechneten Standpunkte alle innerhalb der 1-Sigma Umgebung der realen Position. Eine leichte Verschlechterung bei der Erhöhung der Anzahl der Tracks ist zwar zu erkennen, jedoch ist diese so gering, dass sie vernachlässigt werden kann. Der einzige Unterschied ergibt sich bei der Verteilung der einzelnen Ergebnisse auf die verschiedenen Stufen. Je mehr Tracks hinzukommen, desto weniger Ergebnisse befinden sich in Stufe 0, wohingegen die Anzahl der Tracks in Stufe 1 zunimmt. Die restlichen Stufen bleiben fast unverändert. Werden im Vergleich dazu die Ergebnisse aus dem ersten Test betrachtet, lässt sich ein Zusammenhang zwischen der Bestimmung von Position und Anzahl der Tracks erkennen. Auch die Berechnung der richtigen Anzahl der Tracks wurde schlechter, je mehr Tracks hinzugefügt wurden. Der Grund für beide Beobachtungen liegt darin, dass der Abstand zwischen den Tracks geringer wird, je mehr Tracks sich auf der Fläche befinden. Dadurch kommt es häufiger vor, dass zwei verschiedene Tracks zu einem fusioniert werden, welcher in der Mitte der beiden tatsächlichen Tracks platziert wird. Das erklärt auch, warum sich nur die Anzahl der Tracks in Stufe 0 und Stufe 1 verändern und in den anderen nicht. Da das Zusammenfassen von mehreren Tracks aufgrund einer niedrigen Distanz passiert, sind nur die Stufen betroffen, die eine niedrige Distanz widerspiegeln. Eine Zusammenfassung der Ergebnisse ist in Tabelle 6.2 zu sehen.

Anzahl Tracks	Stufe 0	Stufe 1	Stufe 2	Stufe 3	Stufe 4	∅ Stufe
1	47.5 %	41.9 %	9.4 %	0.6 %	0.6 %	0.65
2	42.6 %	45.0 %	10.3 %	1.6 %	0.5 %	0.72
3	40.9 %	49.9 %	8.4 %	0.6 %	0.2 %	0.69
4	31.8 %	57.7 %	9.1 %	1.3 %	0.1 %	0.80
5	33.8 %	56.2 %	8.5 %	1.3 %	0.2 %	0.78

Tabelle 6.2.: Ergebnisse aus Test 2

Im Durchschnitt liegen die berechneten Ergebnisse zu knapp 90 % der Zeit mindestens in der 1 Sigma Umgebung. Die Sensorungenauigkeit kann insgesamt von 0.15 Metern auf 0.11 Meter eingegrenzt werden. Damit ist die hier getestete Anforderung erfüllt, da die Tracks von dem Algorithmus an der richtigen Stelle erkannt werden.

- **Test 3: Niedrige Sensorgenauigkeit**
  Bei diesem Test wurden die Ergebnisse für niedrige Sensorgenauigkeiten getestet. Dafür wurde zunächst ein Sensor mit einer Ungenauigkeit von 0.2 Meter verwendet. Diese wurde schrittweise um 0.1 erhöht bis eine Ungenauigkeit von 0.7 Meter erreicht war. Die Ergebnisse der Berechnungen mit einem Sensor haben im Schnitt eine Genauigkeit von mindestens 1.87× Sigma, was einer maximalen Abweichung von etwa 0.28 Metern entspricht.

Da das Rauschen des Sensors durchschnittlich bei 0.45 Metern lag, konnte der Algorithmus die Abweichung der Messungen folglich um knapp 40 % verringern.

Dabei zeigte der Test, dass die Ergebnisse schlechter wurden je ungenauer der Sensor wurde. Bei einer Sensorgenauigkeit von 0.2 Metern, lag die durchschnittliche Abweichung der Ergebnisse von der wahren Position noch bei maximal 0.13 Metern, während sie bei einem Sensor mit einer Ungenauigkeit von 0.7 Metern bei 0.44 Metern lag. Dabei gab es vor Allem bei der Erhöhung der Sensorungenauigkeit von 0.4 auf 0.5 eine starke Verschlechterung. Die Anzahl an Tracks, die genau an die richtige Stelle berechnet wurden, sank von 17 auf 2 und die Anzahl der Tracks in der 1-Sigma Umgebung der wahren Poition halbierte sich. Stattdessen stieg die Menge an Tracks in der 3-Sigma Umgebung von 29 auf 43 und die Tracks der Stufe vier war mehr als dreimal so hoch. Die durchschnittliche Abweichung stieg von 0.24 Meter auf 0.34 Meter. Die Ergebnisse des Tests mit einem Sensor sind in Tabelle 6.3 dargestellt.

Sensorgenauigkeit (in Metern)	Stufe 0	Stufe 1	Stufe 2	Stufe 3	Stufe 4	∅ Stufe
0.2	26.1 %	62.1 %	10.5 %	1.0 %	0.3 %	0.87
0.3	7.7 %	65.4 %	21.0 %	2.2 %	3.7 %	1.29
0.4	6.8 %	45.8 %	31.2 %	11.5 %	4.7 %	1.62
0.5	0.8 %	24.5 %	39.1 %	17.8 %	17.8 %	2.27
0.6	0.9 %	24.4 %	39.6 %	20.0 %	15.1 %	2.24
0.7	0.5 %	9.1 %	25.7 %	25.7 %	39.0 %	2.94

Tabelle 6.3.: Ergebnisse aus Test 3 mit einem Sensor

Anschließend wurde ein zweiter Sensor hinzugefügt. Im Schnitt lagen die berechneten Ergebnisse bei Variation der Sensorgenauigkeiten innerhalb eines Radius von 1.84 Sigma um die reale Position, was einer maximalen Abweichung von 0.27 Metern entspricht und damit etwa genauso hoch ist, wie bei der Verwendung von nur einem Sensor. Bei einer gleichmäßigen Steigerung beider Sensorungenauigkeiten von 0.2 auf 0.7 Meter waren die Ergebnisse ähnlich, wie bei der Verwendung von nur einem Sensor. Die Ergebnisse waren besser, wenn bei nur einem Sensor die Genauigkeit verschlechtert wurde und der andere Sensor eine Genauigkeit von 0.2 Metern behielt. Obwohl der zweite Sensor bis zu einer Ungenauigkeit von 0.9 Metern heruntergesetzt wurde, also um 0.2 Meter mehr als bei dem vorhergehenden Test, lag die durchschnittliche Abweichung bei 0.23 Metern. Das bedeutet, dass die Messungen durch die Sensoren um etwa 40 % verbessert werden konnten. Vor allem die Anzahl an berechneten Tracks, die in Stufe 0 eingestuft wurden, ist deutlich langsamer gesunken, als bei nur einem Sensor oder zwei gleich guten Sensoren. Die Ergebnisse sind zusammenfassend in Tabelle 6.4 dargestellt.

Analog dazu waren die berechneten Tracks des Partikelfilters schlechter, wenn ein Sensor durchgehend eine Ungenauigkeit von 0.7 Metern hatte und der andere Sensor verändert wurde. Die durchschnittliche Abweichung von der tatsächlichen Position lag bei 0.34 Metern. Dies ist zwar deutlich schlechter als bei den anderen Tests, aber besser, als die durchschnittliche Sensorungenauigkeit von 0.58 Metern. Vor Allem die Anzahl an korrekt berechneten Ergebnisse der Stufe 0 ist von Anfang an sehr niedrig. Die Ergebnisse, die außerhalb der 3-Sigma Umgebung der tatsächlichen Position des Objekts liegen sind dagegen schon bei einer Sensorgenauigkeit von 0.2 Metern vergleichsweise hoch. Die Ergebnisse sind zusammenfassend in Tabelle 6.5 dargestellt.

Sensorgenauigkeit (in Metern)	Stufe 0	Stufe 1	Stufe 2	Stufe 3	Stufe 4	∅ Stufe
0.2	26.5 %	63.8 %	9.1 %	0.6 %	0 %	0.84
0.3	17.5 %	47.6 %	28.7 %	5.8 %	0.4 %	1.24
0.4	14.2 %	56.7 %	22.2 %	4.7 %	2.2 %	1.24
0.5	10.1 %	57.7 %	23.6 %	5.6 %	3.0 %	1.34
0.6	7.9 %	32.5 %	40.2 %	14.3 %	7.1 %	1.76
0.7	7.0 %	32.7 %	41.3 %	12.0 %	7.0 %	1.79
0.8	6.6 %	23.3 %	42.3 %	16.3 %	11.5 %	2.03
0.9	5.9 %	26.6 %	33.0 %	14.5 %	20.0 %	2.16

Tabelle 6.4.: Ergebnisse aus Test 3 mit einem Sensor der Genauigkeit 0.2 und einem Sensor mit variabler Genauigkeit

Sensorgenauigkeit (in Metern)	Stufe 0	Stufe 1	Stufe 2	Stufe 3	Stufe 4	∅ Stufe
0.2	6.3 %	37.7 %	34.7 %	13.8 %	7.5 %	1.79
0.3	3.8 %	26.1 %	33.8 %	20.5 %	15.8 %	2.18
0.4	0.9 %	18.0 %	44.6 %	23.6 %	12.9 %	2.3
0.5	0.5 %	17.1 %	32.0 %	23.8 %	26.6 %	2.59
0.6	1.5 %	12.4 %	31.2 %	27.2 %	27.7 %	2.67
0.7	0.5 %	6.8 %	26.7 %	29.1 %	36.9 %	2.95

Tabelle 6.5.: Ergebnisse aus Test 3 mit einem Sensor der Genauigkeit 0.7 und einem Sensor mit variabler Genauigkeit

Mit diesem Test sollte untersucht werden, ab welcher Sensorungenauigkeit ein korrektes Tracking nicht mehr möglich ist. Ein konkreter Schwellwert kann dabei nicht genannt werden, da der Algorithmus die Abweichung der Sensormessungen bei jeder Ungenauigkeit verringert. Unabhängig von der Genauigkeit der Sensormessung, kann bei der Verwendung von einem einzelnen Sensor allerdings eine deutliche Verschlechterung bei einer Sensorgenauigkeit von 0.5 Metern erkannt werden. Bei zwei Sensoren liegen diese Schwellen bei den Genauigkeiten von 0.2 und 0.8 Metern, sowie bei 0.7 und 0.3 Metern. Da diese im Durchschnitt auch eine Genauigkeit von 0.5 ergeben, kann eine mittlere Sensorgenauigkeit von 0.5 Metern, als Grenze angesehen werden, bei der die Ergebnisse des Algorithmus deutlich schlechter werden.

- **Test 4: Hohe Sensorgenauigkeit**
  Bei diesem Test sollte festgestellt werden, welche Auswirkungen ein sehr genauer Sensor auf das Fusionsergebnis hat. Dazu wurde zuerst ein Sensor verwendet, dessen Genauigkeit schrittweise erhöht wurde. Dabei wurden bei dem Ergebnis der Fusion keine Besonderheiten erkannt, die von dem bisher bekannten Muster abweichen. Je genauer der Sensor wurde, desto genauer war auch das Ergebnis. Für eine Sensorgenauigkeit von 0.2 Metern liegt das berechnete Ergebnis noch innerhalb eines Radius von 0.86× Sigma, wohingegen es bei 0.1 Metern nur noch im Umkreis von 0.35× Sigma liegt und bei 0.05 Metern innerhalb eines Radius von 0.11× Sigma. Bei einer Sensorgenauigkeit von 0.01 Metern liegt es schließlich bei 0× Sigma. Auch das Hinzufügen eines zweiten Sensors erzeugte keine ungewöhnlichen Ergebnisse.

Die einzige Auswirkung, die dieser Test beobachten konnte, war die Erhöhung der Laufzeit, je genauer die Sensoren waren. Um das genauer zu untersuchen, wurden jeweils fünf Durchläufe mit einer Sensorgenauigkeit zwischen 0.5 und 0.01 durchgeführt, bei denen die Laufzeit gemessen wurde. Die durchschnittlichen Ergebnisse sind in Tabelle 6.6 dargestellt.

Sensorgenauigkeit (in Metern)	Laufzeit (in Sekunden)
0.5	1.701432
0.4	1.683948
0.3	1.717678
0.2	1.793418
0.1	2.18088
0.05	3.878952
0.01	65.2408

Tabelle 6.6.: Laufzeiten in Abhängigkeit von der Sensorgenauigkeit

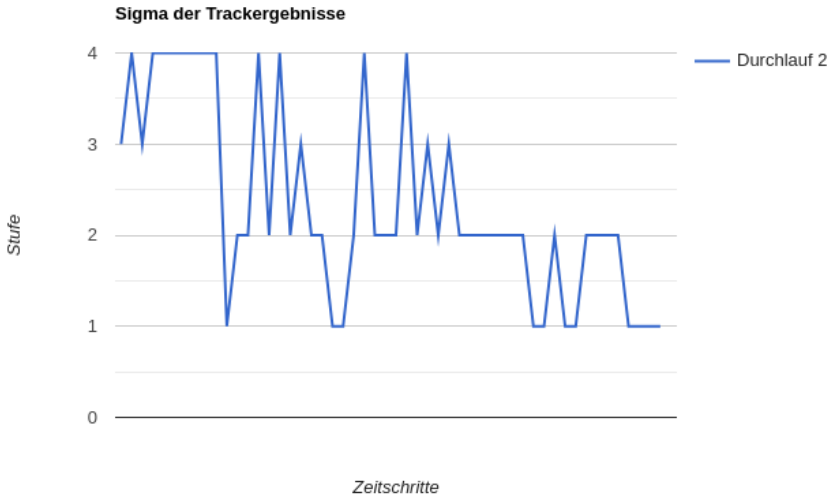
Bis zu einer Sensorgenauigkeit von 0.2 Metern ergeben sich ähnlich lange Laufzeiten zwischen 1.68 und 1.79 Sekunden. Bei einer Genauigkeit von 0.1 Metern steigt die Laufzeit auf über zwei Sekunden und bei 0.01 Metern auf knapp vier Sekunden. Im letzten Schritt, bei einer Sensorgenauigkeit von 0.01 Metern, steigt die Laufzeit deutlich auf über eine Minute. Die Ursache dafür liegt in der Trackbildung. Die Neuronen, die dafür nötig sind, werden über die gesamte Kreisfläche mit einem Durchmesser von $4 \cdot \pi$ Metern verteilt. Der Abstand zwischen den Neuronen beträgt die dreifache Sensorgenauigkeit. Je genauer der Sensor ist, desto mehr Neuronen werden also generiert. Bei dieser Fläche und einer Sensorgenauigkeit von 0.5 Metern sind das 496 Neuronen. Wird die Sensorgenauigkeit auf 0.01 Meter erhöht, sind das $137'812$ Neuronen. Die meisten dieser Neuronen werden im ersten Schritt bereits wieder gelöscht, da sich kein Partikel in ihrer 3-Sigma Umgebung befindet. Um zu erfahren, welche Neuronen gelöscht werden können, muss die Distanz zwischen jedem Neuron und jedem der mindestens 40 Partikel berechnet werden. Das sind wenigstens $40 \cdot 137'812 = 5'512'480$ Distanzen, die berechnet werden müssen.

Eine Lösung für dieses Problem wäre es, die Anzahl an Neuronen nach oben zu begrenzen. Da dies dazu führen könnte, dass kein Neuron übrigbleibt, in dessen 3-Sigma Umgebung sich ein Partikel befindet, muss auch ein Mindestbereich festgelegt werden, in dem nach einem Partikel gesucht wird. Dieser muss entsprechend an die Maximalanzahl an Neuronen angepasst werden.

Insgesamt kann die Frage, welche Auswirkungen ein sehr genauer Sensor auf das Ergebnis hat, so beantwortet werden, dass sich das Ergebnis weiterhin gleich verhält und stetig besser wird. Allerdings hat eine sehr hohe Genauigkeit Auswirkungen auf die Laufzeit.

- **Test 5: Durchschnittliche Sensorgenauigkeit**
  Für diesen Test wurden zwei Sensoren genutzt, deren Messungenauigkeit zwischen 0.1 und 0.5 Metern lag. Im Schnitt hat der Sensor eine Ungenauigkeit von 0.3 Metern. Die Genauigkeit des Messergebnisses lag im Mittel bei 1.27× Sigma, was einer Messungenauigkeit von 0.19 Metern entspricht. Das bedeutet, dass der Partikelfilter-Algorithmus die Messwerte im Schnitt um etwa ein Drittel verbessern konnte.

Mehr als die Hälfte der Tracks lag in der 1-Sigma Umgebung der wahren Position und 15 % der Tracks wurden auf die richtige Stelle berechnet. Ein knappes Viertel der berechneten Tracks wurde mit Stufe 2 bewertet, was einer Distanz von maximal 0.3 Metern entspricht. Damit sind insgesamt 91,4 % der berechneten Ergebnisse genauer als der Sensor. Nur 5.9 % wurden mit Stufe 3 und 2.8 % mit Stufe 4 bewertet. Dieses Ergebnis zeigt, dass der Algorithmus in der Lage ist, fehlerhafte Messungen abzudämpfen und somit die Ergebnisse der Sensoren zu verbessern. Außerdem ist festzustellen, dass das Ergebnis immer besser wurde, je länger der Algorithmus lief. Wenn eine Messung sehr weit von der richtigen Stelle entfernt war, wurde das Ergebnis wieder etwas schlechter, hat sich dann aber im Laufe der Zeit wieder verbessert. Eine solche Verbesserung ist exemplarisch für zwei Durchläufe, bei denen die Fusionsergebnisse zu Beginn sehr schlecht waren, in Abbildung 6.3 dargestellt. Dabei sind für je 60 Zeitschritte die Stufen er Ergebnisse angegeben.



(a)

(b)

Abbildung 6.3.: Verbesserung der Ergebnisse über die Zeit

Die Frage, wie gut das Fusionsergebnis bei durchschnittlicher Sensorgenauigkeit ist, kann also so beantwortet werden, dass es die Ungenauigkeit des Sensors auf durchschnittlich 0.19 Meter eingrenzen kann. Außerdem ist eine Verbesserung der Ergebnisse über die Zeit zu erkennen.

7. Fazit und Ausblick

Fazit:

Das Ziel dieser Bachelorarbeit war es, die Sensordatenfusion mithilfe eines Partikelfilters zu untersuchen. Dazu wurden anfangs drei Leitfragen gestellt, deren Antworten durch die Implementierung und das Testen eines Partikelfilters erarbeitet wurden.

Die Implementierung beantwortet die erste Leitfrage, ob ein Partikelfilter dazu geeignet ist, eine Sensordatenfusion für unterschiedliche Tracks mit verschiedenen Sensoren im Phasenraum zu realisieren. Das konnte bestätigt werden, da der Partikelfilter mehrere Sensoren als Input und mehrere Tracks als Output besitzt. Der Input wird dabei durch den Partikelfilter fusioniert und verarbeitet und führt so zu dem Output. Die Fusion der Sensordaten erfolgt dabei durch abwechselnde Verwendung der Sensoren, welche die Position der Partikel beeinflussen.

Durch die Durchführug der ersten beiden Tests kann die zweite Leitfrage folgendermaßen beantwortet werden: Die Trackbildung lässt sich mithilfe einer Self Organizing Map realisieren. Durch die Verarbeitung der Daten zu Tracks mithilfe einer Self Organizing Map, ist der Partikelfilter dazu in der Lage, die Anzahl an Tracks in 86% der Fälle richtig zu erkennen. Dabei gibt es noch ein sehr großes Verbesserungspotential durch eine Anpassung der Neuronenanzahl und -verteilung. Zusätzlich zeigen die Tests, dass der Partikelfilter „Split Track“ und „Merge Track“ Events realisieren kann. Die Abweichung der berechneten Position der Tracks zu der tatsächlichen Stelle ist etwa 25 % niedriger, als das Rauschen des Sensors und kann noch weiter verringert werden, wenn die Berechnung der Anzahl an Tracks genauer wird.

Die letzten drei Tests untersuchten die Auswirkungen der Sensorgenauigkeit auf das Fusionsergebnis. Zusammmenfassend ist festzuhalten, dass das Fusionsergebnis besser ist, je genauer die Sensoren sind. Im ersten Test wurde festgestellt, dass die Qualität des Trackingergebnisses ab einer durchschnittlichen Sensorungenauigkeit von 0.5 Metern stark sinkt, ein Tracking jedoch immernoch möglich ist. Ein sehr genauer Sensor hat keine besonderen Auswirkungen auf die Qualität des Ergebnisses. Allerdings steigt die Rechenzeit bei Erhöhung der Sensorgenauigkeit stark an. Durch die Anpassung der initialen Verteilung der Neuronen, kann dieses Problem gelöst werden. Wie gut das Fusionsergebnis bei einer durchschnittlichen Sensorgenauigkeit von etwa 0.3 Metern ist, wurde im Letzten Versuch getestet. Das Ergebnis zeigt, dass der Algorithmus die Sensorungenauigkeit auf mindestens 0.19 Meter eingrenzen kann. Außerdem kann erkannt werden, dass sich das Ergebnis über die Zeit stetig verbessert, bis eine ungenaue Messung das Ergebnis wieder negativ beeinflusst.

Ausblick:

Insgesamt konnten in dieser Arbeit alle Leitfragen beantwortet und deren Anforderungen erfüllt werden. Mithilfe dieser Ergebnisse und dem gewonnenen Wissen über Schwächen und Stärken des Partikelfiler-Algorithmus, kann dieser nun weiter optimiert werden. Sobald das erfolgt ist, kann er an einem realen Trackingbeispiel mit realen Daten angewendet werden.

Literatur

[1] M. S. Arulampalam u. a.
„A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking“.
In: *IEEE Transactions on Signal Processing* 50.2 (2002), S. 174–188.
DOI: 10.1109/78.978374.

[2] John Bowman und Andy Hammerlindl. „Asymptote: A vector graphics language“.
In: *TUGboat* 29.2 (2008), S. 288–294.
URL: http://www.math.ualberta.ca/~bowman/publications/asyTUG.pdf.

[3] Arnaud Doucet, Nando de Freitas und Neil Gordon.
„An Introduction to Sequential Monte Carlo Methods“. In:
Sequential Monte Carlo Methods in Practice.
Hrsg. von Arnaud Doucet, Nando de Freitas und Neil Gordon.
New York, NY: Springer New York, 2001, S. 3–14. ISBN: 978-1-4757-3437-9.
DOI: 10.1007/978-1-4757-3437-9_1.
URL: https://doi.org/10.1007/978-1-4757-3437-9_1.

[4] MBDA Deutschland GmbH. *Das Unternehmen*. 2020.
URL: https://www.mbda-deutschland.de/das-unternehmen/ (besucht am 28. 11. 2020).

[5] Fredrik Gustafsson.
„Particle filter theory and practice with positioning applications“.
In: *IEEE Aerospace and Electronic Systems Magazine* 25.7 (2010), S. 53–82.
DOI: 10.1109/MAES.2010.5546308.

[6] Daekeun Jeon u. a.
„Nonlinear aircraft tracking filter utilizing a point mass flight dynamics model“.
In: *Proceedings of the Institution of Mechanical Engineers Part G Journal of Aerospace Engineering* 227 (Nov. 2013). DOI: 10.1177/0954410012463641.

[7] Q. Li u. a. „Kalman Filter and Its Application“. In: *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*. 2015, S. 74–77.
DOI: 10.1109/ICINIS.2015.35.

[8] A. Nurhakim u. a.
„Modified Fourth-Order Runge-Kutta Method Based on Trapezoid Approach“.
In: *2018 4th International Conference on Wireless and Telematics (ICWT)*. 2018,
S. 1–5. DOI: 10.1109/ICWT.2018.8527811.

[9] Markus Otto. *Rechenmethoden für Studierende der Physik im ersten Jahr*.
Heidelberg: Spektrum Akademischer Verlag, 2011, S. 317–318.
ISBN: 978-3-8274-2455-6. DOI: 10.1007/978-3-8274-2456-3.

[10] Branko Ristic, Sanjeev Arulampalam und Neil Gordon.
Beyond the Kalman filter. particle filters for tracking applications.
Boston, MA: Artech House, 2004. ISBN: 1-58053-631-X.

[11] P. Rohal und J. Ochodnicky. „Radar target tracking by Kalman and particle filter“.
In: *2017 Communication and Information Technologies (KIT)*. 2017, S. 1–4.
DOI: 10.23919/KIT.2017.8109459.

[12] Towards Data Science.
Kohonen Self-Organizing Maps. A special type of Artificial Neural Network. 2019.
URL: https://towardsdatascience.com/kohonen-self-organizing-maps-a29040d688da (besucht am 22. 01. 2021).

[13] Maarten Speekenbrink. „A tutorial on particle filters“.
In: *Journal of mathematical psychology* 73 (2016), S. 140–152.
DOI: 10.1016/j.jmp.2016.05.006.

[14] Harshita Vemula. *How Affinity Propagation works?* 2019.
URL: https://towardsdatascience.com/math-and-intuition-behind-affinity-propagation-4ec5feae5b23 (besucht am 22. 01. 2021).

A. Source Code

Übersicht

In Tabelle A.1 sind alle Klassen des Source Code mit einer kurzen Erklärung und der Information, ob sie hier abgedruckt sind, enthalten. Einige Dateien mit Hilfsklassen, die nicht selbst implementiert wurden, wurden nicht abgedruckt, da sie Informationen enthalten, welche nicht veröffentlicht werden dürfen.

Datei	**Enthaltene Klassen**	**Hier abgedruckt?**	**Erklärung**
cl_angle.cpp	cl_angle	Nein	Winkeloperationen
cl_angle.h	cl_angle	Nein	Header: Winkeloperationen
cl_ode.cpp	cl_ode_solution	Nein	Lösung von Differenzialgleichungen erster Ordnung
cl_ode.h	cl_ode_abort cl_ode_abort_t_final cl_ode_rhs cl_ode_solution cl_odesolver cl_ode_trans	Nein	Hilfsfunktionen und virtuelle Klassen zur Lösung von Differenzialgleichungen erster Ordnung
cl_ode_pendel.h	cl_ode_pendel	Ja	Lösung der Differentialgleichung eines Pendels
cl_ode_rk4.h	cl_odesolver_rk4	Ja	Runge-Kutta-Verfahren zur Lösung von Differenzialgleichungen erster Ordnung
cl_particle.cpp	cl_particle_filter	Ja	Hauptaufgaben des Partikelfilters
cl_particle.h	cl_particle cl_particle_filter	Ja	Attribute der Partikel und Hauptaufgaben des Partikelfilters
cl_sensor.h	cl_sensor	Ja	Attribute und wichtigste Funktionen eines Sensors
cl_som_cluster.cpp	cl_som_cluster	Ja	Self Organizing Map
cl_som_cluster.h	cl_som_cluster	Ja	Header: Self Organizing Map
CMakeLists.txt		Nein	Build Konfiguration
grfl_matrix.cpp	grfl_matrix	Nein	Matrixoperationen
grfl_matrix.h	grfl_matrix	Nein	Header: Matrixoperationen
grfl_vector.cpp	grfl_vector	Nein	Vektoroperationen
grfl_vector.h	grfl_vector	Nein	Header: Vektoroperationen
grfl_ranx.cpp	grfl_ranx	Nein	Generieren von Zufallszahlen
grfl_ranx.h	grfl_ranx	Nein	Header: Generieren von Zufallszahlen
join.pl		Nein	Skript zum Visualisieren und Zusammensetzen der Ergebnisse
main.cpp	cl_pf_results	Ja	Programm-Ablauf und Klasse mit Ergebnissen
math_ext.h		Nein	Mathematische Beschränkungen und Vorzeichen-Funktion

Tabelle A.1.: Datei-Übersicht

Source Code

Source Code A.1.: cl_ode_pendel

```
#pragma once
#include "cl_ode.h"

class cl_ode_pendel : public cl_ode_rhs{
private:
        double        m_PI;

public:
        cl_ode_pendel(double PI) : m_PI(PI) {}
        cl_ode_pendel(const cl_ode_pendel& x) : m_PI(x.m_PI) {}

        ~cl_ode_pendel(){}

        bool compute_rhs(const grfl_vector& y, const double t, grfl_vector& rhs)const;
        bool jacobian(const grfl_vector& y, grfl_matrix& J)const{return(false);}
        bool sanity_check(grfl_vector& y)const{return(true);}
        double tell_PI()const{return(m_PI);}
};

bool cl_ode_pendel::compute_rhs(const grfl_vector& y, const double t,
grfl_vector& rhs)const{
        // Y= (phi, phi_dot)
        // Y= (y1, y2)
        // rhs = Y_dot = (y2, -PI * sin(y1))
        if (y.size()!=2) return(false);
        rhs.resize(y.size());

        rhs[0]=y[1];
        rhs[1]=-m_PI*sin(y[0]);

        //success
        return(true);
}
```

Source Code A.2.: cl_ode_rk4

```
#pragma once
#include "cl_ode.h"

class cl_odesolver_rk4 : public cl_odesolver{
private:
        double m_step;
        grfl_vector y_1;
        grfl_vector y_2;
        grfl_vector y_3;

        grfl_vector y_prime_0;
```

```
12          grfl_vector y_prime_1;
13          grfl_vector y_prime_2;
14          grfl_vector y_prime_3;
15
16
17  public:
18          cl_odesolver_rk4(const unsigned int dim_of_y) : m_step(1.0){
19                  y_1.resize(dim_of_y);
20                  y_2.resize(dim_of_y);
21                  y_3.resize(dim_of_y);
22
23                  y_prime_0.resize(dim_of_y);
24                  y_prime_1.resize(dim_of_y);
25                  y_prime_2.resize(dim_of_y);
26                  y_prime_3.resize(dim_of_y);
27          }
28
29          cl_odesolver_rk4(const unsigned int dim_of_y,
30          const double stepsize) : m_step(stepsize){
31                  y_1.resize(dim_of_y);
32                  y_2.resize(dim_of_y);
33                  y_3.resize(dim_of_y);
34
35                  y_prime_0.resize(dim_of_y);
36                  y_prime_1.resize(dim_of_y);
37                  y_prime_2.resize(dim_of_y);
38                  y_prime_3.resize(dim_of_y);
39          }
40
41          ~cl_odesolver_rk4(){ }
42
43          bool step(const grfl_vector& y_0, double& t, const cl_ode_rhs& ode_rhs,
44          grfl_vector& y_1);
45
46          //accessor methods
47          void set_stepsize(const double stepsize){m_step=stepsize;}
48          double tell_stepsize()const{return(m_step);}
49  };
50
51  inline bool cl_odesolver_rk4::step(const grfl_vector& y_0, double& t,
52  const cl_ode_rhs& ode_rhs, grfl_vector& y_new){
53          //abort if y_0 does not have the proper dimension specified
54          //when constructing the solver
55          if ( y_1.size()!=y_0.size() ) return(false);
56
57          const double half_step=0.5*m_step;
58          const double full_step=m_step;
59          const double sixth_step=m_step/6.0;
60
61          //y_1=y_0 + h/2 * y_prime(y_0,t)
62          if (not ode_rhs.compute_rhs(y_0,t,y_prime_0)) return(false);
63          y_1=y_0;
```

```
64          y_1.scale_add(half_step,y_prime_0);
65
66          //y_2=y_0 + h/2 * y_prime(y_1,t+h/2)
67          if (not ode_rhs.compute_rhs(y_1,t+half_step,y_prime_1)) return(false);
68          y_2=y_0;
69          y_2.scale_add(half_step,y_prime_1);
70
71          //y_3=y_0 + h * y_prime(y_2,t+h/2)
72          if (not ode_rhs.compute_rhs(y_2,t+half_step,y_prime_2)) return(false);
73          y_3=y_0;
74          y_3.scale_add(full_step,y_prime_2);
75
76          if (not ode_rhs.compute_rhs(y_3,t+full_step,y_prime_3)) return(false);
77
78          //sum up the resulting new state:
79          //y_{n+1}= y_0 + h/6 (y'_0 + 2 (y'_1+y'_2) + y'_3)
80          y_new.resize(y_0.size());
81
82          for (unsigned int i=0; i < y_0.size(); i++){
83                  y_new[i]= y_0[i] + sixth_step * ( y_prime_0[i] + 2.0 *
84                  (y_prime_1[i] + y_prime_2[i]) + y_prime_3[i]);
85          }
86
87          //return the two return values
88          t+=full_step;
89          return(true);
90  }
```

Source Code A.3.: cl_particle.cpp

```
1   #include<vector>
2   #include<deque>
3   #include<iostream>
4   #include<iomanip>
5   #include"cl_particle.h"
6   #include"cl_afp_cluster.h"
7   #include"cl_som_cluster.h"
8
9   using namespace std;
10
11  inline std::ostream& operator<<(std::ostream& lhs, const cl_particle& p){
12          lhs << "(" << std::setw(15) << p.x <<"," << std::setw(15) << p.y << ","
13          << std::setw(15) << p.last_update_time <<")";
14          return(lhs);
15  }
16
17  inline std::ostream& operator<<(std::ostream& lhs, const cl_particle_filter& pf){
18          unsigned int i = 0;
19          for(auto it=pf.cbegin(); it!=pf.cend(); ++it,++i){
20                  lhs << std::setw(5)<< i << "   " << *it << std::endl;
21          }
```

```
22          return(lhs);
23  }
24
25  void cl_particle_filter::vicinity(const cl_particle& p, double R,
26  std::deque<unsigned int>& v){
27          //determine the elements in the vicinity of p with radius R
28          for (unsigned int i=0; i < pf.size(); ++i){
29                  if (p.distance(pf[i]) <R){
30                          v.push_back(i);
31                  }
32          }
33  }
34
35  unsigned int cl_particle_filter::update(const cl_particle& p, double R,
36  double update_time){
37          //all particles in the R-vicinity of p are updated to move towards p a little bit
38          //return values is the number of particles updated
39          std::deque<unsigned int> v;
40
41          //determine all particles within a Radius of R
42          vicinity(p,R,v);
43
44
45          //all particles in the vicinity are attracted by p a little bit
46          //according to an E(x)= - 1/cosh(x) potential
47          //with E'(x) = + sinh(x)/cosh^2(x) and
48          //a step delta_x =def= -0.1 * E'(d)
49          //where d is the Euclidian distance between particles
50          //we update the position of the particle i according to:
51          //r_i' = r_i + (delta_x / x) * (r_i - r_p)
52          //which translates to:
53          //x_i' = x_i + (delta_x / x) * (x_i - r_p)
54          //y_i' = y_i + (delta_x / x) * (y_i - r_p)
55          for (unsigned int i=0; i < v.size(); ++i){
56                  double d = p.distance(pf[v[i]]);
57                  if (0 < d){
58                          double temp = cosh(d);
59                          double delta_d = -0.3*sinh(d)/(temp*temp);
60                          double step = delta_d/d;
61
62                          //overwrite the particle
63                          double x = (1.0+step)*pf[v[i]].tell_x() - step * p.tell_x();
64                          double y = (1.0+step)*pf[v[i]].tell_y() - step * p.tell_y();
65                          pf[v[i]].set_x(x);
66                          pf[v[i]].set_y(y);
67                          pf[v[i]].set_last_update_time(update_time);
68                  }
69          }
70          //return the number of particles updated
71          return(v.size());
72  }
73
```

```
74  unsigned int cl_particle_filter::update_noise(const cl_particle& r, const double sigma,
75  const double update_time, grfl_ranx& rg){
76          //all particles in the R-vicinity of r are updated
77          //to move towards r a little bit plus some smaller random noise
78          //return values is the number of particles updated
79          std::deque<unsigned int> v;
80
81          //determine all particles within a Radius of 3 sigma
82          vicinity(r,3.0*sigma,v);
83
84          //we update the position of all particles p in vincinity v according to:
85          //p' = p + |x| e_q         + n
86          //where p' is the new particle position
87          //where p is the current particle position
88          //|x| is the value of a random number x, which is normal distributed arround zero
89          //with standard deviation sigma
90          //         e_q = r-p / |r-p|
91          //the random step in e_q direction goes a random distance x (sigma distributed)
92          //towards particle r
93          //the ranomd noise n is circular gaussian distributed with 0.3 sigma
94          //and serves to not let all particles settle down on point r
95          //much smaller than sigma
96          for (unsigned int i=0; i < v.size(); ++i){
97                  double        d=r.distance(pf[v[i]]);
98                  if (0 < d){
99                          //dice some directed step
100                         double x = fabs(rg.gauss(0.3*sigma));
101                         double e_q_x= (r.tell_x() - pf[v[i]].tell_x() ) / d;
102                         double e_q_y= (r.tell_y() - pf[v[i]].tell_y() ) / d;
103
104                         //dice some random noise with sigma/3
105                         double R = fabs(rg.gauss(0.1*sigma));
106                         double phi = rg.ran1()*2.0*M_PI;
107                         double noise_x = R*cos(phi);
108                         double noise_y = R*sin(phi);
109
110                         //overwrite the particle
111                         double new_x = pf[v[i]].tell_x() + x * e_q_x + noise_x;
112                         double new_y = pf[v[i]].tell_y() + x * e_q_y + noise_y;
113                         pf[v[i]].set_x(new_x);
114                         pf[v[i]].set_y(new_y);
115                         pf[v[i]].set_last_update_time(update_time);
116                 }
117
118         }
119         //return the number of particles updated
120         return(v.size());
121 }
122
123 void cl_particle_filter::thin_out(const double epsilon){
124         //remove all the particles, which are closer than epsilon to another particle
125         for(unsigned int i=0; i < pf.size(); ++i){
```

```
126                 for(unsigned int j=i+1; j < pf.size(); ++j){
127                         if(pf[i].distance(pf[j]) < epsilon){
128                                 remove(j);
129                         }
130                 }
131         }
132 }
133
134 void cl_particle_filter::die_out(const double t, const double max_lifetime){
135         //particles that have not been update by more than max_lifetime, are removed.
136         for(unsigned int i=0; i < pf.size();){
137                 if( max_lifetime < (t - pf[i].tell_last_update_time())){
138                         remove(i);
139                         continue;
140                 }
141                 ++i;
142         }
143 }
144
145 unsigned int cl_particle_filter::determine_tracks_afp(double t, double Delta_t,
146 std::deque<cl_particle>& tracks){
147         //determine all recently updated particles in the pf
148         //and put them in the afp cluster algorithm
149         static constexpr unsigned int dead_time_limit  =1;
150         static constexpr unsigned int iteration_steps = 100;
151         static constexpr double delta = 0.5;
152         static constexpr double lambda = 2.0;
153         cl_afp_cluster afp = cl_afp_cluster();
154         for(auto it=pf.cbegin(); it!=pf.cend(); ++it){
155                 //only accept those particles that have been updated recently
156                 if(it->dead_time(t,Delta_t) < dead_time_limit){
157                         afp.push_back(*it);
158                 }
159         }
160
161         afp.cluster(iteration_steps,delta,lambda);
162         tracks.clear();
163         afp.get_cluster_centers(tracks);
164
165         return(tracks.size());
166 }
167
168 unsigned int cl_particle_filter::determine_tracks_som(const double t,
169 const double Delta_t, const double sigma,
170 std::deque<cl_particle>& tracks,
171 grfl_ranx& rg, unsigned int number_of_neurons){
172         //determine all recently updated particles in the pf
173         //and put them in the som cluster algorithm
174         static constexpr unsigned int dead_time_limit = 1;
175         static constexpr unsigned int iteration_steps = 1000;
176         static constexpr double delta = 0.1;
177         cl_som_cluster som = cl_som_cluster();
```

```
178
179         //setup the particles
180         for(auto it=pf.cbegin(); it!=pf.cend(); ++it){
181                 //only accept those particles that have been updated recently
182                 if ( it->dead_time(t,Delta_t) < dead_time_limit){
183                         som.push_back(*it);
184                 }
185         }
186
187         //take a slightly larger sigma for the SOM to avoid false tracks:
188         const double SOM_sigma=1.3*sigma;
189
190         som.init_grid(-2.0*M_PI, +2.0*M_PI,-2.0*M_PI,+2.0*M_PI, SOM_sigma);
191
192         //do the clustering
193         som.cluster(iteration_steps,SOM_sigma,delta);
194
195         //return the tracks
196         tracks.clear();
197         for (auto it=som.neurons_cbegin(); it!=som.neurons_cend();++it){
198                 tracks.push_back(*it);
199         }
200
201         return(tracks.size());
202 }
```

Source Code A.4.: cl_particle.h

```
1  #pragma once
2  #include<vector>
3  #include<deque>
4  #include<iostream>
5  #include<iomanip>
6  #include<math.h>
7  #include<grfl_ranx.h>
8
9  class cl_particle{
10 private:
11         friend std::ostream& operator<<(std::ostream& lhs, const cl_particle& p);
12         double x;
13         double y;
14         double last_update_time;
15
16 public:
17         cl_particle() : x(0),y(0),last_update_time(0){}
18         cl_particle(double an_x, double a_y) : x(an_x), y(a_y), last_update_time(0.0){}
19         cl_particle(double an_x, double a_y, double a_t) : x(an_x), y(a_y),
20         last_update_time(a_t){}
21         cl_particle(const cl_particle& p) : x(p.x), y(p.y),
22         last_update_time(p.last_update_time){}
23         ~cl_particle(){}
```

```
24
25          cl_particle& operator=(const cl_particle p){x=p.x; y=p.y;
26                  last_update_time=p.last_update_time; return(*this);}
27          double distance(const cl_particle& p)const{
28                  return( sqrt( (x-p.x)*(x-p.x) + (y-p.y)*(y-p.y)));
29          };
30          unsigned int dead_time(double t, double delta_t)const{
31                  return( (0 < fabs(delta_t)) ? ceil((t-last_update_time)/delta_t) : 0);
32          }
33          bool is_epsilon_equal(const cl_particle& p, const double epsilon)const{
34                  return (fabs(x-p.x) < epsilon && fabs(y-p.y) < epsilon);
35          }
36
37          //getter,setter
38          void set_last_update_time(double t){last_update_time=t;};
39          double tell_last_update_time()const{return(last_update_time);};
40          double tell_x()const{return(x);}
41          double tell_y()const{return(y);}
42          void set_x(double u){x=u;}
43          void set_y(double u){y=u;}
44  };
45
46  class cl_particle_filter{
47  private:
48          friend std::ostream& operator<<(std::ostream& lhs, const cl_particle_filter& pf);
49          std::vector<cl_particle> pf;
50
51  public:
52          cl_particle_filter(){pf.reserve(1000);}
53          cl_particle_filter(const cl_particle_filter& p){pf.reserve(1000);pf=p.pf;}
54          ~cl_particle_filter(){}
55
56          void reserve(unsigned int N ) {pf.reserve(N);}
57          void clear(){pf.clear();}
58          unsigned int size()const{return(pf.size());}
59
60          void push_back(const cl_particle& x){pf.push_back(x);}
61          void pop_back(){if (0 < pf.size()) {pf.resize(pf.size()-1);}}
62
63          void remove(unsigned int idx){
64                  if (0<=idx && idx<pf.size()){
65                          pf[idx]=pf[pf.size()-1]; pop_back();
66                  }
67          }
68
69          auto begin(){return(pf.begin());}
70          auto end(){return(pf.end());}
71          auto cbegin()const{return(pf.cbegin());}
72          auto cend()const{return(pf.cend());}
73
74          //determine all particles in the ball of radius R around p and return them in v
75          void vicinity(const cl_particle& p, double R, std::deque<unsigned int>& v);
76
```

```
77          //all particles in the R-vicinity of p are updated to move towards p a little bit
78          unsigned int update(const cl_particle& p, double R, double update_time);
79
80          //a better stochastic update variant
81          unsigned int update_noise(const cl_particle& r, const double sigma,
82          const double update_time, grfl_ranx& rg);
83
84          //remove all the particles, which are closer than epsilon to another particle
85          void thin_out(const double epsilon);
86
87          //particles that have not been update by more than max_lifetime, are removed.
88          void die_out(const double t, const double max_lifetime);
89
90          //determine all recently updated particles in the pf
91          //and put them in the afp cluster algorithm
92          unsigned int determine_tracks_afp(double t, double Delta_t,
93          std::deque<cl_particle>& tracks);
94
95          //determine all recently updated particles in the pf
96          //and put them in the SOM cluster algorithm
97          unsigned int determine_tracks_som(const double t, const double Delta_t,
98          const double sigma, std::deque<cl_particle>& tracks,
99          grfl_ranx& rg, unsigned int number_of_neurons);
100 };
```

Source Code A.5.: cl_sensor.h

```
1   #pragma once
2   #include "cl_angle.h"
3
4   class cl_sensor{
5   private:
6           double sigma;
7
8   public:
9           cl_sensor(double a_sigma) : sigma(a_sigma){}
10          ~cl_sensor(){}
11
12          //do a gaussian distributed measurement
13          void measure(double x_in, double y_in, double& x_out, double& y_out,
14          grfl_ranx& rg)const{
15                  cl_angle beta = cl_angle( M_PI*rg.ran1_pm(),angle::radian);
16                  double z = fabs(rg.gauss(0.0,sigma));
17                  x_out = x_in+z*cos(beta.rad());
18                  y_out = y_in+z*sin(beta.rad());
19          }
20
21          //do a gaussian distributed measurement, but guarantee that z < cut_off_sigma
22          void measure_cut_off(double x_in, double y_in, double& x_out, double& y_out,
23          double cut_off_sigma, grfl_ranx& rg)const{
24                  cl_angle beta = cl_angle( M_PI*rg.ran1_pm(),angle::radian);
```

```
                double z = 0.0;
                do{
                        z = fabs(rg.gauss(0.0,sigma));
                }
                while(fabs(cut_off_sigma) < z);
                x_out=x_in+z*cos(beta.rad());
                y_out=y_in+z*sin(beta.rad());
        }
        double tell_sigma()const{return(sigma);}
};
```

Source Code A.6.: cl_som_cluster.cpp

```
#include<fstream>
#include<iomanip>
#include<deque>
#include<math.h>
#include"grfl_ranx.h"
#include"cl_som_cluster.h"

using namespace std;

void cl_som_cluster::init(unsigned int number_of_neurons, double x_min, double x_max,
double y_min, double y_max, grfl_ranx& rg){
        //dice a number of neurons into the domain
        for (unsigned int i=0; i< number_of_neurons; ++i){
                double x = x_min + rg.ran1() * (x_max-x_min);
                double y = y_min + rg.ran1() * (y_max-y_min);
                neurons.push_back(cl_particle(x,y));
        }
}

void cl_som_cluster::init_grid(double x_min, double x_max, double y_min, double y_max,
double sigma){
        for (double x=x_min; x < x_max+sigma; x+=3.0*sigma){
                for (double y=y_min; y < y_max+sigma; y+=3.0*sigma){
                        neurons.push_back(cl_particle(x,y));
                }
        }
}

double cl_som_cluster::fuse_neurons(const double min_distance){
        //fuse neurons, which are closer than min_distance
        double cumul_displacement = 0.0;
        for(auto it=neurons.begin(); it!=neurons.end(); ++it){
                auto jt=it;
                for(++jt; jt!=neurons.end(); ++jt){
                        if(it->distance(*jt) < fabs(min_distance)){
                                //update the neuron under it..
                                double new_x = 0.5*((*it).tell_x() + (*jt).tell_x());
                                double new_y = 0.5*((*it).tell_y() + (*jt).tell_y());
```

```
39
40                                  double delta_x = new_x - (*it).tell_x();
41                                  double delta_y = new_y - (*it).tell_x();
42
43                                  (*it).set_x(new_x);
44                                  (*it).set_y(new_y);
45
46                                  //..keep track of the cumulative displacement
47                                  cumul_displacement+=delta_x*delta_x+delta_y*delta_y;
48
49                                  //and move the last neuron to jt
50                                  (*jt)=neurons.back();
51
52                                  //remove the last neuron
53                                  neurons.pop_back();
54
55                                  //keep jt on the neuron after ++jt in the for()
56                                  --jt;
57                          }
58                  }
59          }
60          return(cumul_displacement);
61  }
62
63  unsigned int cl_som_cluster::cluster(unsigned int iterations, const double sigma,
64  const double delta){
65          //remove all neurons, which have no training data in their 3 sigma environment
66          remove_displaced_neurons(3.0*sigma,1);
67
68          for(unsigned int i=0; i<iterations; ++i){
69                  //a slowly vanishing learn rate
70                  double Delta = 100.0 * delta / static_cast<double>( i + 100);
71
72                  //keep track of the cululative displacement
73                  double cumul_displacement = 0.0;
74
75                  //train each training data
76                  for(auto it=p.cbegin(); it!=p.cend(); ++it){
77                          cumul_displacement+=train_neurons(*it,sigma,Delta);
78                  }
79
80                  //fuse neurons that are too close to each other
81                  //the two sigma environments touch
82                  cumul_displacement += fuse_neurons(4.0*sigma);
83
84                  if(2 < iterations && iterations%100==0){
85                          //remove all neurons, which have less or equal than
86                          //3 training data particles in their 3 sigma environment
87                          remove_displaced_neurons(3.0*sigma,3);
88                  }
89
90                  //abort, when the SOM stops changing
91                  if(cumul_displacement < 1e-6 ){
```

```
92                          cout << "SOM: stopping iteration after " << i
93                          << " iterations"<< endl;
94                          break;
95                  }
96
97          }
98
99          //remove all neurons, which have fewer than 3 training data particles
100         //in their 3 sigma environment
101         remove_displaced_neurons(3.0*sigma,3);
102
103         //extract the clusters
104         extract_cluster(sigma);
105
106         return(neurons.size());
107 }
108
109 void cl_som_cluster::remove_displaced_neurons(const double radius_of_vicinity,
110 unsigned int min_number_of_particles){
111         //remove all neurons, which have fewer particles in their vicinity
112         //than the minimum specified number
113         for (auto it=neurons.begin(); it!=neurons.end(); ++it){
114                 unsigned int particle_counter = 0;
115                 bool remove_neuron = true;
116                 for(auto jt=p.begin(); jt!=p.end(); ++jt){
117                         //don't remove neuron 'it' in case there exist a training data
118                         //in its vicinity environment
119                         if((*it).distance(*jt) < radius_of_vicinity){
120                                 ++particle_counter;
121                                 if(min_number_of_particles <= particle_counter ){
122                                         //there are enough particles in the vicinity:
123                                         //Break the for
124                                         remove_neuron = false;
125                                         break;
126                                 }
127                         }
128                 }
129                 //remove the neuron
130                 if (remove_neuron){
131                         (*it) = neurons.back();
132                         --it;
133                         neurons.pop_back();
134                 }
135         }
136 }
137
138 void cl_som_cluster::extract_cluster(const double sigma){
139         //extract the clusters found
140         clusters.clear();
141
142         for(auto it=neurons.cbegin(); it!=neurons.cend(); ++it){
143                 //for every neuron, there is a cluster
```

```
144                 clusters.push_back(*(new std::deque<cl_particle>));
145                 //assign all particles to the neuron,
146                 //which are in the 3 sigma environment
147                 for(auto jt=p.cbegin(); jt!=p.cend(); ++jt){
148                         if(it->distance(*jt) < 3*sigma){
149                                 clusters.back().push_back(*jt);
150                         }
151                 }
152         }
153 }
154
155 double cl_som_cluster::train_neurons(const cl_particle& td, const double sigma,
156 const double delta){
157         //given a training data datum td and a sigma, expose all neurons to it
158         //
159         //        V(x)= - exp( -0.5 * ( x^2 + y^2))
160         //        V_x        = dV/dx        = + x exp ( -0.5 * ( x^2 + y^2 ))
161         //        V_y        = dV/dy        = + y exp ( -0.5 * ( x^2 + y^2 ))
162         //with x = distance / sigma;
163         //delta is the learning rate
164         //if the training data is within the 3 sigma environment of a neuron,
165         //the neuron is updated in the direction of the negative gradient
166         //with the learining rate
167
168         //the return value is the sum of squared displacements inidicating motion
169         //versus stagnation
170         double        cumul_displacement = 0.0;
171
172         //for positive sigmas...
173         if(0.0 < sigma){
174                 //.. each neuron is updated..
175                 for(auto it=neurons.begin(); it!=neurons.end(); ++it){
176                         double d = it->distance(td) / sigma;
177                         //..if it is in the training data is within
178                         //its 3 sigma environment
179                         if(d < 3.0){
180                                 double        delta_x=td.tell_x()-(*it).tell_x();
181                                 double        delta_y=td.tell_y()-(*it).tell_y();
182                                 double        V_x= delta_x * exp(-0.5*d*d);
183                                 double        V_y= delta_y * exp(-0.5*d*d);
184
185                                 //compute the displacements
186                                 delta_x=delta * V_x;
187                                 delta_y=delta * V_y;
188
189                                 //make the neuron go into the direction of
190                                 //the positive gradient with learning rate delta
191                                 (*it).set_x( (*it).tell_x() + delta_x);
192                                 (*it).set_y( (*it).tell_y() + delta_y);
193
194                                 //Keep track of the cumulative displacement
195                                 cumul_displacement+=delta_x*delta_x+delta_y*delta_y;
```

```
196                         }
197                 }
198         }
199         return(cumul_displacement);
200 }
201
202 //draw cluster with asy
203 inline void cl_som_cluster::asy(const std::string filename, const double sigma){
204         fstream        fileout;
205         fileout.open(filename,ios::out);
206
207         fileout << "size(20cm);"<<endl;
208         fileout << "settings.tex=\"pdflatex\";"<<endl;
209
210         for (auto it=p.cbegin(); it!=p.cend(); ++it){
211                 fileout << "dot(("<< it->tell_x() << "," << it->tell_y() << "),black);"
212                 << endl;
213         }
214
215         for (auto it=neurons.cbegin(); it!=neurons.cend(); ++it){
216                 fileout << "filldraw(shift(("<< it->tell_x() << ","<< it->tell_y()
217                 <<"))*scale(0.05)*rotate(45)*shift((-0.5,-0.5))*unitsquare,
218                 green+linewidth(2));" << endl;
219                 fileout << "draw(shift(("<< it->tell_x() << ","<< it->tell_y()
220                 <<"))*scale("<< 1*sigma << ")*unitcircle,green+dotted);" << endl;
221                 fileout << "draw(shift(("<< it->tell_x() << ","<< it->tell_y()
222                 <<"))*scale("<< 3*sigma << ")*unitcircle,green+dashed);" << endl;
223         }
224         fileout.close();
225 }
```

Source Code A.7.: cl_som_cluster.h

```
1  #pragma once
2  #include<deque>
3  #include"grfl_ranx.h"
4  #include"cl_particle.h"
5
6  //a class for clustering with a self organizing map
7  class cl_som_cluster{
8  private:
9          std::deque<cl_particle> p;
10         std::deque<cl_particle> neurons;
11         std::deque<std::deque<cl_particle>> clusters;
12
13         double fuse_neurons(const double min_distance);
14         double train_neurons(const cl_particle& td, const double sigma,
15         const double delta);
16         void extract_cluster(const double sigma);
17
18 public:
```

```
19          cl_som_cluster(){}
20          ~cl_som_cluster(){}
21
22          void init(unsigned int number_of_neurons, double x_min, double x_max,
23          double y_min, double y_max, grfl_ranx& rg);
24          void init(){neurons=p;}
25          void init_grid(double x_min, double x_max, double y_min, double y_max,
26          double sigma);
27
28          unsigned int size()const {return(p.size());}
29          unsigned int number_of_neurons()const {return(neurons.size());}
30          void clear(){p.clear(); neurons.clear(); clusters.clear(); }
31          void push_back(const cl_particle& x){p.push_back(x);}
32
33          auto begin(){return(p.begin());}
34          auto end(){return(p.end());}
35          auto cbegin()const{return(p.cbegin());}
36          auto cend()const{return(p.cend());}
37
38          auto neurons_cbegin()const{return(neurons.cbegin());}
39          auto neurons_cend()const{return(neurons.cend());}
40          auto clusters_cbegin()const{return(clusters.cbegin());}
41          auto clusters_cend()const{return(clusters.cend());}
42
43          unsigned int cluster(unsigned int iterations, const double sigma,
44          const double delta);
45
46          //remove the neurons that do not have the minimum amount
47          //of particles of p in their vicinity environment
48          void remove_displaced_neurons(const double radius_of_vicinity,
49          unsigned int min_number_of_particles);
50          void asy(const std::string filename, const double sigma);
51  };
```

Source Code A.8.: main.cpp

```
1   #include <unistd.h>
2   #include <iostream>
3   #include <iomanip>
4   #include <math.h>
5   #include <random>
6   #include <sstream>
7   #include "grfl_ranx.h"
8   #include "cl_angle.h"
9   #include "cl_ode_pendel.h"
10  #include "cl_ode_rk4.h"
11  #include "cl_ode_rk56.h"
12  #include "cl_sensor.h"
13  #include "cl_particle.h"
14  #include  <time.h>
15
```

```
16  using namespace std;
17
18  class cl_pf_results{
19  private:
20          std::deque<cl_ode_solution> m_truth;
21          std::deque<cl_ode_solution> m_traj_bundle;
22          std::deque<unsigned int> m_dead_times;
23          std::deque<cl_particle> m_measurements;
24          std::deque<double> m_sigmas;
25          std::deque<cl_particle> m_tracks;
26
27
28  public:
29          cl_pf_results(){}
30          ~cl_pf_results(){}
31
32          void clear() {
33                  m_truth.clear();
34                  m_traj_bundle.clear();
35                  m_dead_times.clear();
36                  m_measurements.clear();
37                  m_sigmas.clear();
38                  m_tracks.clear();
39          }
40
41          //Getter
42          std::deque<cl_ode_solution>& truth() {return(m_truth);}
43          std::deque<cl_ode_solution>& traj() {return(m_traj_bundle);}
44          std::deque<unsigned int>& dead_times() {return(m_dead_times);}
45          std::deque<cl_particle>& measurements() {return(m_measurements);}
46          std::deque<double>& sigmas() {return(m_sigmas);}
47          std::deque<cl_particle>& tracks() {return(m_tracks);}
48
49          //Getter const
50          std::deque<cl_ode_solution>const& truth() const{return(m_truth);}
51          std::deque<cl_ode_solution>const& traj() const{return(m_traj_bundle);}
52          std::deque<unsigned int>const& dead_times() const{return(m_dead_times);}
53          std::deque<cl_particle>const& measurements() const{return(m_measurements);}
54          std::deque<double>const& sigmas() const{return(m_sigmas);}
55          std::deque<cl_particle>const& tracks() const{return(m_tracks);}
56  };
57
58  //converts a pendulum trajectory into an asymptote path definition string
59  std::string solution_to_asypath(const std::string name,const cl_ode_solution& sol){
60          std::ostringstream        str;
61          str.precision(3);
62          str << "path ";
63          str << name << "=";
64
65          auto it=sol.cbegin();
66          str << "(" << (*it)[0] << "," << (*it)[1]<<")";
67
```

```
68          for (auto it=sol.cbegin(); it!=sol.cend(); ++it)
69          {
70                  str << "--" << "(" << (*it)[0] << "," << (*it)[1]<<")";
71          }
72          str << ";";
73          return(str.str());
74  }
75
76  //computes the background trajectories
77  int background_trajectories(const cl_ode_pendel& pendel,
78  std::deque<cl_ode_solution>& traj){
79          //integration parameters
80          static constexpr double t_start = 0.0;
81          static constexpr double t_step = 0.03;
82
83          //the first
84          grfl_vector y0 = std::vector<double>{ 0,0.5};
85          cl_ode_solution solution = cl_ode_solution();
86          cl_odesolver_rk4 rk4 = cl_odesolver_rk4(y0.size(),t_step);
87
88          //middle
89          solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(13.0));
90          traj.push_back(solution);
91
92          //left
93          solution.clear();
94          y0 = std::vector<double>{-2.0*M_PI,0.5};
95          solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(13.0));
96          traj.push_back(solution);
97
98          //right
99          solution.clear();
100         y0 = std::vector<double>{+2.0*M_PI,0.5};
101         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(13.0));
102         traj.push_back(solution);
103
104         //the second
105         //middle
106         solution.clear();
107         y0 = std::vector<double>{0,1.0};
108         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(7.0));
109         traj.push_back(solution);
110
111         //left
112         solution.clear();
113         y0 = std::vector<double>{-2.0*M_PI,1.0};
114         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(7.0));
115         traj.push_back(solution);
116
117         //right
118         solution.clear();
119         y0 = std::vector<double>{+2.0*M_PI,1.0};
```

```
120         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(7.0));
121         traj.push_back(solution);
122 
123         //the third
124         //middle
125         solution.clear();
126         y0 = std::vector<double>{0,1.7};
127         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(10.0));
128         traj.push_back(solution);
129 
130         //left
131         solution.clear();
132         y0= std::vector<double>{-2.0*M_PI,1.7};
133         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(10.0));
134         traj.push_back(solution);
135 
136         //right
137         solution.clear();
138         y0 = std::vector<double>{+2.0*M_PI,1.7};
139         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(10.0));
140         traj.push_back(solution);
141 
142         //the fourth
143         //middle
144         solution.clear();
145         y0 = std::vector<double>{0.0,1.999};
146         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(50));
147         traj.push_back(solution);
148 
149         //left
150         solution.clear();
151         y0 = std::vector<double>{-2.0*M_PI,1.999};
152         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(50));
153         traj.push_back(solution);
154 
155         //right
156         solution.clear();
157         y0 = std::vector<double>{+2.0*M_PI,1.999};
158         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(50));
159         traj.push_back(solution);
160 
161         //the fifth
162         //bottom
163         solution.clear();
164         y0 = std::vector<double>{-2.0*M_PI,2.05};
165         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(12));
166         traj.push_back(solution);
167 
168         //top
169         solution.clear();
170         y0 = std::vector<double>{+2.0*M_PI,-2.05};
171         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(12));
```

```
172         traj.push_back(solution);
173
174         //the sixths
175         //bottom
176         solution.clear();
177         y0 = std::vector<double>{-2.0*M_PI,3};
178         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(5));
179         traj.push_back(solution);
180
181         //top
182         solution.clear();
183         y0 = std::vector<double>{+2.0*M_PI,-3.0};
184         solution.integrate(y0,t_start,rk4,pendel,cl_ode_abort_t_final(5));
185         traj.push_back(solution);
186
187         //success
188         return(0);
189 }
190
191 void asy(std::string filename, const std::deque<cl_ode_solution>& kulisse,
192 const cl_pf_results& res){
193         fstream        fileout;
194         fileout.open(filename,ios::out);
195
196         fileout << "size(20cm);"<<endl;
197         fileout << "settings.tex=\"pdflatex\";"<<endl;
198
199         //draw the kulisse
200         unsigned int counter=1;
201         for (auto it=kulisse.cbegin(); it!=kulisse.cend(); ++it,++counter){
202                 ostringstream name;
203                 name<< "p" << counter;
204                 fileout << solution_to_asypath(name.str(),*it)<<endl;
205                 fileout << "draw("<<name.str()<<",gray);"<<endl;
206         }
207
208         //draw some trajectories
209         counter = 0;
210         std::string        pen = "red";
211         std::string        pend = "gray";
212         auto jt = res.dead_times().cbegin();
213         for (auto it=res.traj().cbegin(); it!=res.traj().cend()
214         && jt!=res.dead_times().cend(); ++it,++jt,++counter){
215                 ostringstream name;
216                 name<< "part" << counter;
217                 fileout << solution_to_asypath(name.str(),*it)<<endl;
218                 switch((*jt)){
219                 case 0:
220                         pen="heavyred";
221                         pend="heavygray";
222                         break;
223                 case 1:
```

```
                        pen="red";
                        pend="gray";
                        break;
                case 2:
                        pen="lightred";
                        pend="lightgray";
                        break;
                case 3:
                        pen="palered";
                        pend="palegray";
                }
                fileout << "draw("<<name.str()<<","<<pen<<");"<<endl;
                fileout << "dot(( " << (*it).front()[0]<< ","
                << (*it).front()[1]<<"),"<<pend<<");" << endl;
                fileout << "dot(( " << (*it).back()[0]<< ","
                << (*it).back()[1]<<"),"<<pend<<");" << endl;
        }

        //draw the truth trajectories
        unsigned int i = 0;
        for (auto it=res.truth().cbegin(); it!=res.truth().cend(); ++it,++i){
                ostringstream pathname;
                pathname << "truth" << i;
                fileout << solution_to_asypath(pathname.str(),*it)<<endl;
                fileout << "draw("<<pathname.str()<<",green);"<<endl;
                fileout << "dot(( " << (*it).front()[0]<< ","
                << (*it).front()[1]<<"),black);" << endl;
                fileout << "dot(( " << (*it).back()[0]<< ","
                << (*it).back()[1]<<"),black);" << endl;
        }

        //draw the measurement: dot and 1,2,3 sigma environment
        auto kt = res.sigmas().cbegin();
        for (auto it=res.measurements().cbegin(); it!=res.measurements().cend()
        && kt!=res.sigmas().cend();++it,++kt){
                fileout << "dot(("<< it->tell_x() << "," << it->tell_y()
                << "),fuchsia);" << endl;
                fileout << "draw(shift(("<< it->tell_x() << "," << it->tell_y()
                << "))*scale("<<(*kt)<<")*unitcircle,black+dotted);"<<endl;
                fileout << "draw(shift(("<< it->tell_x() << "," << it->tell_y()
                << "))*scale("<<2.0*(*kt)<<")*unitcircle,black+dotted);"<<endl;
                fileout << "draw(shift(("<< it->tell_x() << "," << it->tell_y()
                << "))*scale("<<3.0*(*kt)<<")*unitcircle,black+dotted);"<<endl;
        }

        //draw the tracks
        for (auto it=res.tracks().cbegin(); it!=res.tracks().end(); ++it){
                fileout << "filldraw(shift(("<< it->tell_x() << ","<< it->tell_y()
                <<"))*scale(0.1)*rotate(45)*shift((-0.5,-0.5))*unitsquare,
                green+linewidth(2));" << endl;
        }

```

```
276         //draw the boundingbox & do the clipping
277         fileout << "real a=2.5*pi;"<<endl;
278         fileout << "real b=6;"<<endl;
279         fileout << "path mybbox=yscale(2*b)*xscale(2*a)*shift(-0.5,-0.5)*unitsquare;"
280         << endl;
281         fileout << "draw(mybbox);" << endl;
282         fileout << "clip (mybbox);"<<endl;
283         fileout << "shipout(bbox(1cm));" << endl;
284 
285         fileout.close();
286 }
287 
288 
289 void iterate_pf(const cl_ode_pendel& pendel, const cl_sensor& sensor,
290 const std::deque<grfl_vector>& y0, double t, double Delta_t,
291 cl_particle_filter& pf, grfl_ranx& rg, cl_pf_results& res){
292 
293         //-----------------------------Truth-------------------------
294 
295         //for integrating the truth to time t
296         static constexpr double t_step = 0.03;
297         cl_odesolver_rk4 rk4 = cl_odesolver_rk4(y0.front().size(),t_step);
298 
299         //determine the truth trajectory by integrating pendel
300         //with initial condition y0 from t'=t to t'=t+Delta_t
301         cl_ode_abort_t_final abort = cl_ode_abort_t_final(t+Delta_t);
302 
303         //obtain the true solution of the propagation
304         //from all the truth trajectories y0,t to truth(t') with t  < t' < t+Delta_t
305         res.clear();
306         for (auto it=y0.cbegin(); it!=y0.cend(); ++it){
307                 res.truth().push_back( cl_ode_solution() );
308                 res.truth().back().integrate(*it,t,rk4,pendel,abort);
309         }
310 
311         //-----------------Measurement----------------
312 
313         //measure the current state given by y0
314         double x_measure = 0.0;
315         double y_measure = 0.0;
316 
317         for (auto it=y0.cbegin(); it!=y0.cend(); ++it){
318                 //Get next measurement
319                 sensor.measure((*it)[0],(*it)[1],x_measure,y_measure,rg);
320                 cl_particle        p_measure=cl_particle(x_measure,y_measure,t);
321                 res.measurements().push_back(p_measure);
322                 res.sigmas().push_back(sensor.tell_sigma());
323 
324                 //update all particles in the 3 sigma environment
325                 unsigned int N = pf.update_noise(p_measure,sensor.tell_sigma(),t,rg);
326 
327                 //if N is smaller than N_min add new particles arround the measurement
```

```
328                 //until it is N_min
329                 static constexpr unsigned int N_min=40;
330                 for (unsigned int i=N; i < N_min; ++i){
331                         double x = 0.0;
332                         double y = 0.0;
333                         // avoid drastic outliers, cut off at 2 sigma,
334                         //which covers 95 percent
335                         sensor.measure_cut_off(x_measure,y_measure,x,y,
336                         2.0*sensor.tell_sigma(),rg);
337                         pf.push_back(cl_particle(x,y,t));
338                 }
339         }
340         //determine the tracks with the som
341         static constexpr unsigned int number_of_neurons = 1000;
342         cout << "tracks : " << pf.determine_tracks_som(t,Delta_t,0.5*sensor.tell_sigma(),
343         res.tracks(),rg,number_of_neurons) << endl;
344
345
346         //--------------------------Particles---------------------------------
347
348         //now propagate all particles in the particle filter to t+Delta_t
349         cl_ode_solution sol = cl_ode_solution();
350
351         for (auto it=pf.begin(); it!=pf.end(); ++it){
352                 grfl_vector Y0 = std::vector<double> {(*it).tell_x(), (*it).tell_y()};
353                 sol.clear();
354                 sol.integrate(Y0,t,rk4,pendel,abort);
355                 (*it).set_x(sol.back()[0]);
356                 (*it).set_y(sol.back()[1]);
357                 res.traj().push_back(sol);
358                 res.dead_times().push_back( (*it).dead_time(t,Delta_t) );
359         }
360 }
361
362
363 int main() {
364         //Time measurement variables
365         double time1 = 0.0;
366         double tstart;
367
368         //Start timer:
369         tstart = clock();
370
371         //Basic settings
372         static constexpr double t0 = 1.0; //s
373         static constexpr double g = 9.81; //m/s^2
374         static constexpr double L = 9.81; //m
375         static constexpr double PI_0 = t0*t0*g/L; //start angle
376
377         //initial condition
378         std::deque<grfl_vector> y0;
379         y0.push_back(grfl_vector(std::vector<double> {1.9, 1.7})); //Truth 1
```

```
380         y0.push_back(grfl_vector(std::vector<double> {-1.8, -1.6})); //Truth 2
381
382         //integrate the truth to time t
383         double t = 0.0;
384
385         //Delta t
386         static constexpr double Delta_t = 2.0;
387
388         //Particle filter object
389         cl_particle_filter pf = cl_particle_filter();
390
391         //For generating random numbers later
392         grfl_ranx rg = grfl_ranx();
393
394         //Sensors
395         const cl_sensor sr_sensor = cl_sensor(0.2);
396         const cl_sensor        fc_sensor = cl_sensor(0.1);
397         cl_sensor sensor = sr_sensor;
398
399         //number of Integration steps
400         static constexpr unsigned int iterations = 60;
401
402         //compute background trajectories
403         cl_ode_pendel pendel = cl_ode_pendel(PI_0);
404         std::deque<cl_ode_solution>        kulisse;
405         background_trajectories(pendel,kulisse);
406
407
408         //integrate the particle filter solution and the truth
409         for (unsigned int k=0; k< iterations; ++k){
410                 pf.die_out(t,3*Delta_t);
411
412                 if ( k%2 ==0 ){
413                         sensor=sr_sensor;
414                 }else{
415                         sensor=fc_sensor;
416                 }
417
418                 cl_pf_results results = cl_pf_results();
419                 iterate_pf(pendel,sensor,y0,t,Delta_t,pf,rg,results);
420
421                 //make an asy shot
422                 ostringstream str;
423                 str.fill('0');
424                 str<< "pf_" << std::setw(3) << k << ".asy";
425                 asy(str.str(),kulisse,results);
426
427                 //update the next initial state
428                 t += Delta_t;
429                 auto jt=results.truth().cbegin();
430                 for (auto it=y0.begin(); it!=y0.end() && jt!=results.truth().cend();
431                 ++it,++jt){
```

```
                        (*it)[0]=(*jt).back()[0];
                        (*it)[1]=(*jt).back()[1];
                }
        }
        //Stop timer
        time1 += clock() - tstart;

        //Rescale to seconds
        time1 = time1/CLOCKS_PER_SEC;

        //Print result
        cout << "  time = " << time1 << " sec." << endl;

        return(0);
}
```