PML-Bericht-no-interactive

February 18, 2022

1 Bericht Projektlabor Maschinelles Lernen (PML)

Gruppenmitglied	Matrikelnummer
Christian Singer	2161064
Domenic Gosein	2160647
Lukas Burger	2150580
Maximilian Kürschner	2160331

Betreuender Dozent:Dr.-Ing. Wei Yap TanFakultät der Informationstechnik

Hochschule MannheimWintersemester 2021/22

1.1 Inhalt

1. Einleitung
2. Das Kalman-Filter
3. 1D Radarsensor Experiment
    1. Transitionmodell
    2. Übergangsmatrix/Messfunktion
    3. Experimente
    4. Metrik
4. 3D Radarsensor Experiment
    1. 3D Radarsensor Experiment ohne DBScan
    2. Verschiede Parameterwerte beim 3D Experiment
    3. Der DBScan Algorithmus
    4. 3D Radarsensor Experiment mit DBScan
    5. Vergleich von Sensor und Kalman-Filter
    6. Nutzung der Radialgeschwindigkeit zur Verbesserung des Kalman-Filters
5. Interaktiver Teil (Jupyter Notebook)
    1. Interaktives Kalman-Filter 1D
    2. Interaktiver DBScan
    3. Interaktiver Kalman-Filter mit DBScan 3D
6. Schlussfolgerung und Ausblick

7. Verwendete Literatur
8. Anhang
    1. GitHub Workflow
    2. Jupyter Notebook

Wenn Sie diesen Bericht mit Juypter Notebook ausführen, müssen Sie als erstes folgende Zeilen ausführen, um alle nötigen Module zu importieren. Falls das Notebook auf dem eigenen Rechner ausgeführt wird muss die Bibliothek "ipympl" wahrscheinlich noch heruntergeladen werden, via pip oder conda install.

[1]:
```
# Import aller benötigten Module

# Eigene Module
import DataGenerationRadar1D as gen_1D
from DataGenerationRadar3D import *
from DBScan import *
from ui import interactive1DExperiment, interactiveDBScan,␣
 ↪interactive3DExperiment

# Externe Module
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import copy
from collections import deque
from ipywidgets import *
from collections import deque
%matplotlib widget
plt.style.use('classic')
```

1.2 Einleitung

Nach einer Einführung in maschinelles Lernen, war es unsere Aufgabe das Kalman-Filter und den DBSCAN Algorithmus für Daten aus einem 1D und anschließend 3D Radarsensor zu implementieren.

Um ein besseres Verständis für das Kalman-Filter zu erlangen, haben wir uns zunächst mit der Theorie dahinter beschäftigt. Im Zuge dieses Prozesses sind wir auf das α-β-Filter gestoßen. Das α-β-Filter bildet die Grundlage für eine Reihe von Filtern, darunter auch das Kalman Filter. Wir haben uns daher dazu entschlossen, dieses zu Übungszwecken zu implementieren.

[2]:
```
class abFilter:
    def __init__(self, x_0, dx, a, b, dt):
        self.x_est = x_0 # initial state value
        self.dx = dx # inital change rate
        self.a = a # a scale factor
        self.b = b # b scale factor
        self.dt = dt # time step
```

```
    def step(self, values):
        ests = []
        preds = []
        for z in values:
            # Predict
            x_pred = self.x_est + (self.dx * self.dt)
            preds.append(x_pred)
            self.dx = self.dx
            # Update
            residual = z - x_pred
            self.dx += self.b * (residual)/self.dt
            self.x_est = x_pred + self.a * residual
            ests.append(self.x_est)
        return np.array(ests), np.array(preds)
```

In unserem Beispiel verwenden wir das Filter dazu, das Körpergewicht einer Person vorherzusagen.

Unser Filter verwendet dazu folgende Parameter: * `x_0` als initialen Zusatandswert (in unserem Fall das Anfangsgewicht) * `dx` als initiale Änderungsrate des Gewichts z. B. +0.5 kg/Tag * `a` als Faktor für die Veränderung der Gewichtsmessung * `b` als Faktor für die Änderungsrate des Gewichts * `dt` für das Zeitintervall

sowie `values` für unsere Messwerte.

Nachdem das α-β-Filter initialisiert wurde, führt es folgende Schritte aus:

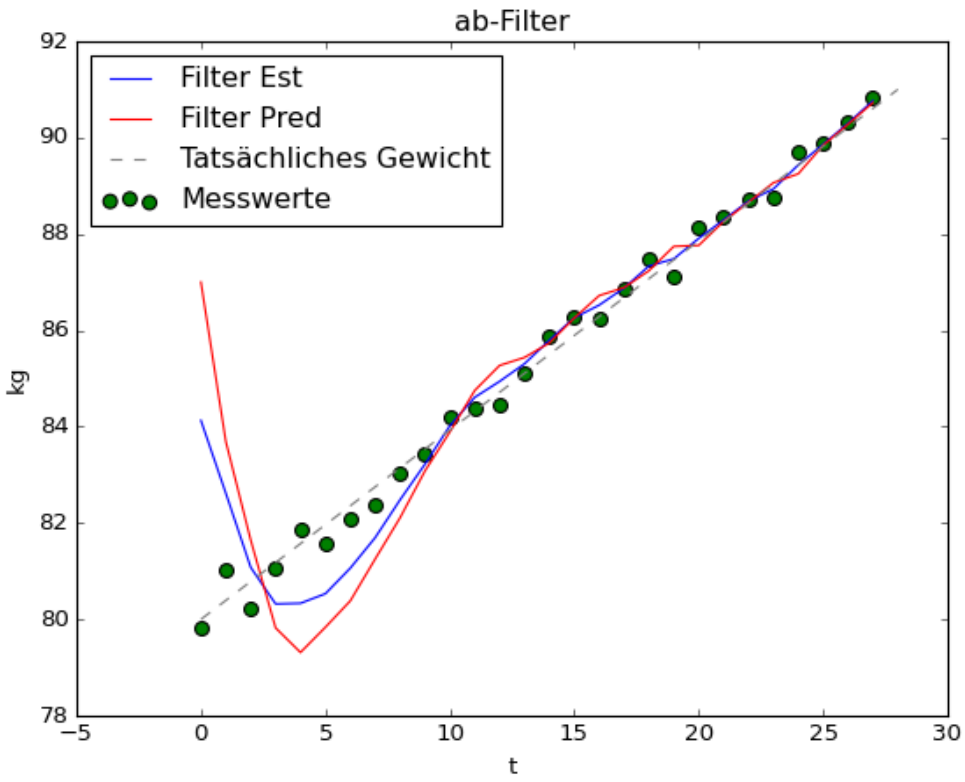
1. Berechnung der Vorhersage im nächsten Zeitintervall basierend auf aktuellem Schätzwert, Änderungsrate und Zeitintervall
2. Berechnen der Differenz aus aktuellem Messwert und Vorhersage
3. Anpassung der neuen Änderungsrate mit Faktor `b`, des Restwerts aus Schritt 2 und dem Zeitintervall
4. Berechnung des neuen Schätzwerts mittels Vorhersage, Faktor `a` und Restwert

Um das Filter zu testen, haben wir mittels einer Funktion 14 Messwerte generiert und diese an den Filter übergeben. Zur Initialisierung haben wir zudem 86 kg, eine Änderungsrate von +1 kg/Tag, einen α-Wert von 0.4 und einen β-Wert von 0.2 und einem Zeitintervall von 1 übergeben. Auf passende Werte für `a` und `b` sind wir durch Ausprobieren gestoßen.

[3]:
```
# Daten: Körpergewichte über n Tage verteilt gemessen
count = 28
def data_generator(x_0, dx, count, noise_factor):
    return [x_0 + dx * i + np.random.randn() * noise_factor for i in␣
 ↪range(count)]

gewichte = data_generator(79.9, 0.4, count, 0.3) # kg
#print(gewichte)
zeitabstaende = [i for i in range(28)] # n Tage
print()
# Initialisierung und Ausführung des ab-Filters
gewicht_filter = abFilter(x_0=86, dx=1, a=0.4, b=0.2, dt=1.)
```

```
pr = gewicht_filter.step(values=gewichte)

# Ploten der Filter Ergebnisse im Vergleich zu den den echten Werten
fig, ax = plt.subplots(figsize=(8, 6), facecolor='w')
ax.plot(pr[0], label='Filter Est')
ax.plot(pr[1], label='Filter Pred', color='r')
ax.scatter(zeitabstaende, gewichte, s=50, facecolor='C1', edgecolor='k',␣
 ↪label='Messwerte')
ax.plot([0,count], [80., 91.], label='Tatsächliches Gewicht', linestyle='--',␣
 ↪color='grey')
ax.set_xlabel('t')
ax.set_ylabel('kg')
ax.set_title("ab-Filter")
ax.legend(loc='upper left')
```

[3]: <matplotlib.legend.Legend at 0x17fdbec6e80>



Die von uns gewählte initiale Schätzung von 86 kg war mit Absicht sehr hoch gewählt. Wir sehen daher einen großen Ausschlag zu Beginn, schließlich benötigt das Filter einige Iterationen, um seine

Werte anzupassen, sodass das Filter zum Ende hin eine deutlich bessere Schätzung ausgibt.

Im nächsten Kapitel widmen wir uns nun dem Kalman-Filter.

1.3 Das Kalman-Filter

Das Kalman-Filter ist ein Algorithmus, der anhand einer Reihe von Messungen über eine gewisse Zeit unbekannte Variablenwerte eines Systems schätzt. Dabei versucht das Filter Unsicherheiten, die z. B. durch den Luftwiderstand entstehen, durch statistisches Rauschen und den Einbezug eines zu Grunde liegenden physikalischen Modells zu reduzieren.

In unserem Fall versuchen wir mit das Kalman-Filter die Position eines Objekts erst im eindimensionalen Raum, dann im dreidimensionalen Raum vorherzusagen. Die dafür notwendigen Messwerte liefert uns dabei eine Radarsensor Simulation.

Die Klasse `KalmanFilter` implementiert das Kalman-Filter und besteht aus zwei Methoden. Über die `__init__(self, s_hat, transition_model, H, Q, R)` Methode kann das Kalman-Filter mit folgenden Parametern initialisiert werden: * `s_hat`: Position des Objekts, also x bzw. x, y und z Koordiante(n) * `P_hat`: Kovarianz des Zustands * `transition_model`: zu Grunde liegenes physikalisches Modell * `H`: Messfunktion * `Q`: Prozessrauschen * `R`: Messrauschen

Nach der Initialisierung kann die `step(self,z)` Funktion mit den Messwerten `z` aufgerufen werden, um den zuvor initialisierten Kalman-Filter Algorithmus auszuführen.

[4]:
```
class KalmanFilter:
    # Initialisierung von Kalman-Filter
    def __init__(self, s_hat, transition_model, H, Q, R):
        self.s_hat = s_hat
        self.P_hat = np.eye(len(s_hat)) * 100
        self.model = transition_model
        self.H = H # Measurement Function
        self.Q = Q # Process Noise
        self.R = R # Measurement Noise

    # Kalman-Filter Algorithmus
    def step(self, z):
        # if only positions are to be predicted by the Kalman-Filter.
        if z.shape[0] == 1:
            H = np.array([self.H[0]])
        # if velocity should be predicted as well.
        else:
            H = self.H
        # Prediction
        s_hat_p = self.model @ self.s_hat
        self.P_hat_p = self.model @ self.P_hat @ self.model.T + self.Q
        # Calculate Kalman Gain
        K = self.P_hat_p @ self.H.T @ np.linalg.inv(self.H @ self.P_hat_p @␣
↪self.H.T + self.R)
        # Update covariance of estimation error
        # self.P_hat = self.P_hat - K @ self.H @ self.P_hat
```

```
        self.P_hat = (np.eye(len(self.s_hat)) - K @ self.H) @ self.P_hat_p
        # Improve estimate
        e_m_p = z - self.H @ s_hat_p
        self.s_hat = s_hat_p + K @ e_m_p
        return self.s_hat
```

Im Detail führt die `step(self,z)` Methode dann folgende Schritte nacheinander aus:

1. Vorhersage der neuen Position
2. Berechnung der Kovarianz des Zustands
3. Berechnung des Kalman Gain
4. Aktualisierung der Kovarianz des Zustands
5. Verbesserung der Schätzung und Rückgabe der neuen Position

Der dabei entstehende Kalman Gain `K`, gibt an, wie sehr wir der Vorhersage im Vergleich zur Messung vertrauen. Im Gegensatz zum α-β Filter wird beim Kalman-Filter also mithilfe von `K` der Einfluss der Sensormessung und der Vorhersage des Models auf die Vorhersage des Filters dynamisch reguliert. Je nachdem ob der Kalman-Filter nur die Position oder auch deren Ableitungen (Geschwindigkeit etc.) bestimmen soll wird eine passende `H` matrix gewählt.

1.4 1D Radarsensor Experiment

Die Simulation für den 1D Radarsensor gibt uns Messzustände vom **Abstand** und **Geschwindigkeit** eines Objektes zum Radarsensor und biete die Möglichkeit zwischen fünf verschiedenen Bewegungsarten zu wählen: *Static*, *Constant Velocity*, *Constant Acceleration*, *Sinus* und *Triangle*. Zusätzlich lassen sich folgende Sensor-Eigenschaften für die Experimente anpassen (Default-Wert in Klammern):

- "initialDistance": Start Position des Objektes ($8m$),
- "stopTime": Messzeit des Sensors ($1s$),
- "SporadicError": Anzahl der Außreißer (5),
- "initialVelocity": Start Geschwindigkeit des Objektes ($3m/s$)
- "movementRange": Amplitude der Sinus/Trianglebewegung (1),
- "frequency": Periode der Sinus/Trianglebeweung (2),
- "velocity": Konstate Geschwindigkeit ($3m/s$)
- "acceleration": Konstatne Beschleunigung ($3m/s^2$)

1.4.1 Transitionmodell

Der Radasensor misst die Position und die Geschwindigkeit eines Objektes mit einer gegebenen Taktfrequenz, welches zu Modellierung des Weg-Zeit-Gesetzes geführt hat:

- Beschleunigung: $a = const$
- Geschwingkeits-Zeit-Gesetz: $v(t) = \dot{s}(t) = at + v_0$
- Weg-Zeit Gesetz: $s(t) = \frac{a}{2}t^2 + v_0 t + s_0$

Für die Anwendung des Modells muss die Messgeschwindigkeit des Sensors berücksichtigt werden, wodurch der Zeitschritt(dt) sich wie folgt berechnen lässt:

$$dt = \frac{1}{Taktfrequenz} \tag{1}$$

Bei einer Taktfrequenz von $100Hz$ führt dies zu folgendem Transitionsmodell:

[5]:
```
dt = 1 / gen_1D.measurementRate # 100 Hz
transition_model = np.array([[1, dt, dt/2],
                             [0,   1, dt  ],
                             [0,   0, dt ]])
transition_model
```

[5]:
```
array([[1.   , 0.01 , 0.005],
       [0.   , 1.   , 0.01 ],
       [0.   , 0.   , 0.01 ]])
```

Das Transitionmodell kann für die unterschiedlichen Bewegungsarten, falls diese im vorraus als bekannt vorrausgesetz sind auch weiter vereinfacht bzw. angepasst werden.

1.4.2 Übergangsmatrix/Messfunktion

Um die im Kalman-Filter geschätzten Werte mit den Messwerten vergleichen zu können, muss die Messfunktion auf

- Postion
- Geschwindigkeit

begrenzt werden. Hierbei entsteht folgende Übergangsmatrix:

[6]:
```
H = np.array([[1., 0., 0.],
              [0., 1., 0.]])
print(H)
```

```
[[1. 0. 0.]
 [0. 1. 0.]]
```

1.4.3 Experimente

Im Folgenden wird der Code für ein Experiment, anhand der Bewegungsart *Static*, erklärt. Für die darauf folgenden Bewegungsarten werden anschließend statische Bilder aus dem Unterorder *figs* geladen und erläutern. Diese können im interaktiven Teil des 1D-Radarsensor nachgestellt werden.

1.4.4 Static

Zuerst wird der Radarsensor mit den Default-Werten initialisiert und die gewünschte Bewegungsart ausgewählt.

[7]:
```
opt = {
    "initialDistance": 8,
    "stopTime": 1,
    "SporadicError": 5,
}
timeAxis, distValues, velValues, truthDistValues, truthVelValues = gen_1D.
 ↪GenerateData(type="Static", options=opt)
```

Anschließend wird das Messrauschen R ermittelt. Diese errechnet sich aus der Sensorgenauigkeit, die durch die Standardabweichung des Messwerts vom wahren Wert(während einer Kalibrierung) dargestellt wird. Dabei kann die Sensorgenauigkeit entweder beim Hersteller erfragt oder durch eigene Messreihen einer statischen Szene, wie folgt geschätzt werden:

[8]:
```
# Sensor ohne vereinzelne Fehler
opt_zero_err = {
    "initialDistance": 8,
    "stopTime": 1,
    "SporadicError": 0,
}
# Führe x-Messreihen durch und nehme die Varianz in den Daten auf
dist_mesurment_err = list()
vel_mesurment_err = list()
for i in range(1000):
    gen_1D.seed = i
    _, distValues_test, velValues_test, _, _ = gen_1D.
 ↪GenerateData(type="Static",

 ↪options=opt_zero_err)
    dist_mesurment_err.append(np.var(distValues_test))
    vel_mesurment_err.append(np.var(velValues_test))
pos_var = np.mean(dist_mesurment_err)
vel_var = np.mean(vel_mesurment_err)
print(f"Positions-Genauigkeit: +-{pos_var} m")
print(f"Geschwindigkeits-Genauigkeit: +-{vel_var} m/s")
```

```
Positions-Genauigkeit: +-0.0001318894150845477 m
Geschwindigkeits-Genauigkeit: +-8.243652403552338e-06 m/s
```

Somit kann R initialisiert werden:

[9]:
```
R = np.diag([pos_var,vel_var])
R
```

[9]:
```
array([[1.31889415e-04, 0.00000000e+00],
       [0.00000000e+00, 8.24365240e-06]])
```

Da auch das physikalische Modell für die Zustandsermittlung des Objektes gestört werden kann, wird hierfür das Prozessrauschen Q verwendet. Diese ist stark von dem zugrunde liegenden Transitionsmodell und seiner genauen physikalischen Beschreibung abhängig. Je höher Q, desto mehr Gewicht haben die verrauschten Messungen und die Schätzgenauigkeit wird beeinträchtigt. Im Falle eines niedrigeren Q wird eine bessere Schätzgenauigkeit erreicht und Verzögerungen in den Messwerten können besser toleriert werden.

Bei einem nicht realen Problemen kann davon ausgegangen werden, dass Q eine Nullmatrix ist. Jedoch hilft eine Q-Matrix von null verschiedenen bessere Konvergenzeigenschaften zu erhalten.

[10]:
```
Q = np.diag([0, 0, 0])
Q
```

[10]:
```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Am Ende müssen noch die Startzustände s_0 des Kalman-Filter initialisiert werden. Diese können entweder mit hohen/niedrigen Werten gesetzte werden, welches einen Einfluss auf die Konvergenz der Kalman-Verstärkung mit sich führt oder den Messdaten wird mehr Vertrauen geschenkt, sodass die ersten Messerwerte als Initialisierung des Kalman-Filters verwendet werden können.

[11]:
```
s0 = np.array([distValues[0], velValues[0], 0]) # initalisierung der Startwerte␣
 ↪mit ersten Messwerten
s0
```

[11]:
```
array([ 7.99498160e+00, -1.26359182e-03,  0.00000000e+00])
```

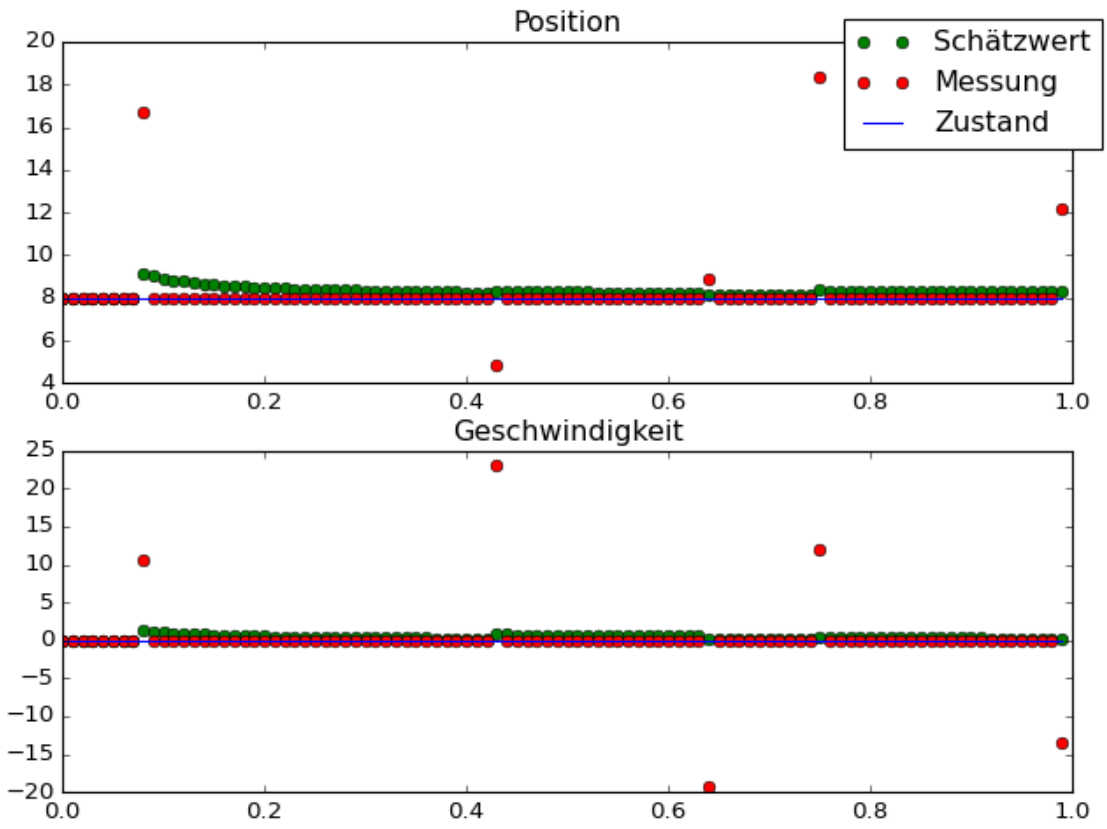
Nun kann der Kalman-Filter auf den Messdaten verwendet werden und die korrigierten Messungen werden in `Predictions` zwischengespeichert.

[12]:
```
kalmanFilter1D = KalmanFilter(s0, transition_model, H, Q, R)
Predictions = [s0]
for i in range(1, np.size(timeAxis)):
    s = np.array([distValues[i], velValues[i]])
    pred = kalmanFilter1D.step(s)
    Predictions.append(pred)
```

Zur **Visualisierung** des 1D Kalman-Filters werden jeweils zwei Grafiken mit der Postion und Geschwindigkeit gezeigt: * Blaue Linie: Zustand (Ground Truth) * Rote Punkte: Messwerte * Grüne Punkte: Geschätzte Werte

[13]:
```
fig, axs = plt.subplots(2,facecolor='w')
fig.tight_layout()
Predictions = np.array(Predictions)

axs[0].set_title('Position')
axs[1].set_title('Geschwindigkeit')
kalman_1D_dist, = axs[0].plot(timeAxis, Predictions[:, 0],␣
 ↪'go',label="Schätzwert")
kalman_1D_vel, = axs[1].plot(timeAxis, Predictions[:, 1], 'go')
data_dist, = axs[0].plot(timeAxis, distValues, 'ro',label="Messung")
data_vel, = axs[1].plot(timeAxis, velValues, 'ro')
true_dist, = axs[0].plot(timeAxis, truthDistValues, 'b',label="Zustand")
true_vel, = axs[1].plot(timeAxis, truthVelValues, 'b')
fig.legend()
```

[13]:
```
<matplotlib.legend.Legend at 0x17fdc2746a0>
```



Metrik

Zur besseren Verständnis und Überblick haben wir den Mean-Square-Error (MSE) zwischen den vom Kalman-Filter geschätzten Werten ($pred$) und den Messwerten($meas$) zu der Ground Truth(GT) berechnet.

$$mse_{pred} = \frac{1}{n}\sum_{1}^{n}(pred - GT)^2$$

$$mse_{meas} = \frac{1}{n}\sum_{1}^{n}(meas - GT)^2$$

Um eine Verbesserung bzw. Verschlechterung der geschätzten Werten nachvollziehen zu können, wurde anschließend das Verhätlniss zwischen den mse_{pred} zu mse_{meas} berechnet:

$$1 - \frac{mse_{pred}}{mse_{meas}}$$

Somit erreichen wir im vorherigen Experiment folgende Metriken:

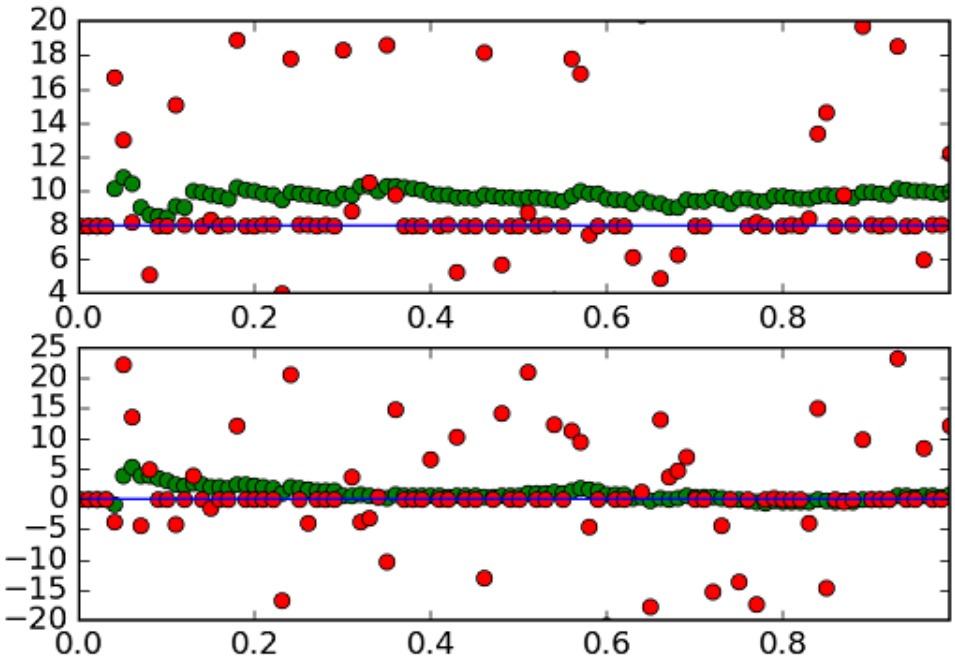
[14]:
```
pos_pred,pos_sens,pos_ratio =  gen_1D.compute_mse(Predictions[:, 0],␣
 ↪truthDistValues, distValues)
vel_pred,vel_sens,vel_ratio =  gen_1D.compute_mse(Predictions[:, 1],␣
 ↪truthVelValues, velValues)
print(f"MSE der Messwerten : \t Pos: {pos_sens:>10.5f}  \t Vel: {vel_sens:>10.
 ↪5f}")
print(f"MSE der Schätzwerte: \t Pos: {pos_pred:>10.5f}  \t Vel: {vel_pred:>10.
 ↪5f}")
print(f"Verbesserung       : \t Pos: {pos_ratio:>10.5f} \t Vel: {vel_ratio:>10.
 ↪5f}")
```

```
MSE der Messwerten :     Pos:    2.10459         Vel:   13.33246
MSE der Schätzwerte:     Pos:    0.14132         Vel:    0.23643
Verbesserung       :     Pos:    0.93285         Vel:    0.98227
```

Ist nun die Anzahl der zufällig aufgetretenen **Fehler sehr hoch**(50), so sind die Messwerte nicht mehr zuverlässig und das Prozessrauschen muss im Kalman-Filter neu angepasst werden.

$$Q = \begin{pmatrix} 0.00001 & 0 & 0 \\ 0 & 0.00005 & 0 \\ 0 & 0 & 0.0005 \end{pmatrix}$$

$$R = \begin{pmatrix} 0.0279 & 0 \\ 0 & 0.103 \end{pmatrix}$$



In der Grafik kann gut erkannt werden, dass das Kalman-Filter gegenüber dem starken Prozessrauschen lernt(siehe ab 0.6 Sekunde der Geschwindigkeit) und gute Vorhersagen treffen kann, welches sich in den folgenden Metriken auch widerspiegelt:

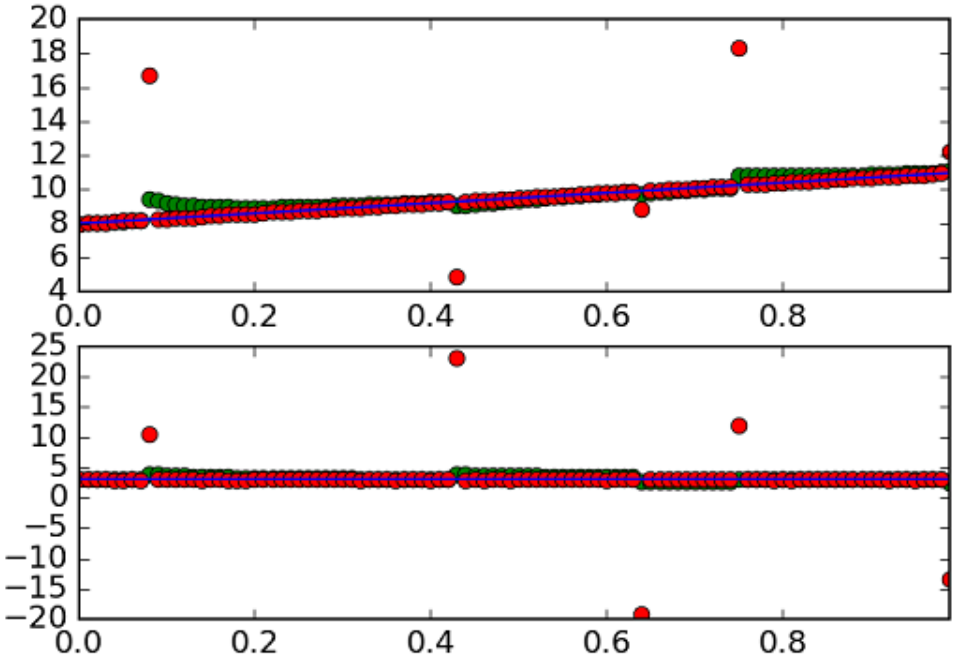
	Position	*Geschwindigkeit*
$MSE_{Messwerte}$	30.78603	74.18581
$MSE_{Schtzwerte}$	2.98777	2.10236
$Verbesserung$	0.90295	0.97166

1.4.5 Constant Velocity

Mit dem oben vorgestellen Transitionmodell erhalten wir folgende Ergebnisse:

$$Q = \begin{pmatrix} 0.000001 & 0 & 0 \\ 0 & 0.00000001 & 0 \\ 0 & 0 & 0.0000005 \end{pmatrix}$$

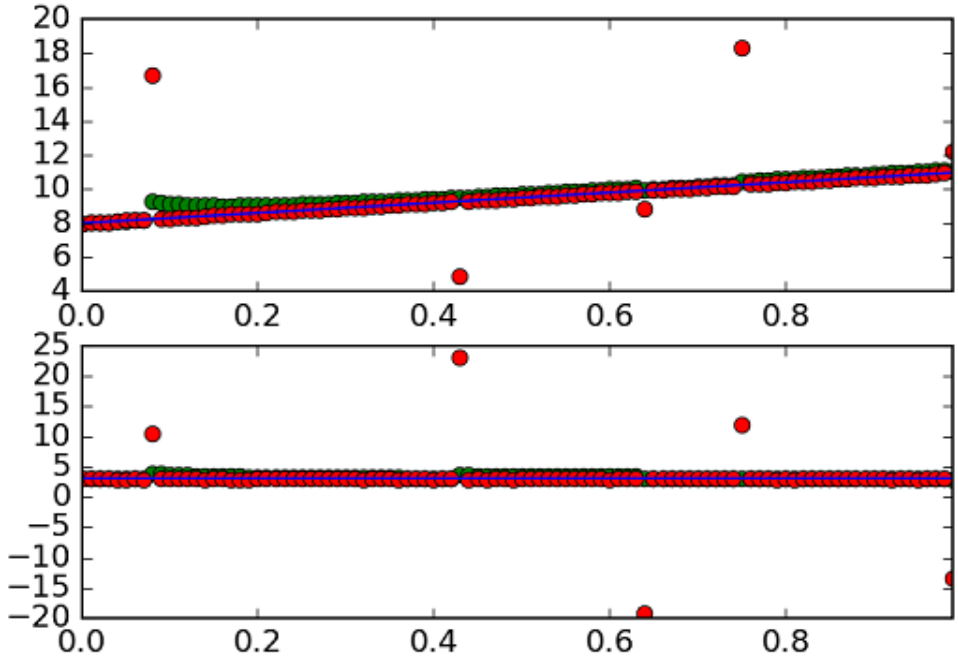
$$R = \begin{pmatrix} 0.0001333 & 0 \\ 0 & 0.000008333 \end{pmatrix}$$



	Position	*Geschwindigkeit*
$MSE_{Messwerten}$	1.58898	13.01375
$MSE_{Schtzwerte}$	0.10319	0.15943
$Verbesserung$	0.93506	0.98775

Dabei zeigt sich, dass das Kalman-Filter die zufälligen Fehler immer besser herrausfilter kann und sich die Genauigkeit über alle Messpunkte stark verbessert. Ist nun die Bewegungsart im Vorfeld bekannt, kann das Transitionsmodell vereinfacht werden, da die Beschleunigung und Geschwindigkeit sich über die Zeit nicht verändern. Es gilt somit:

- $s = v \cdot t$
- $v = const$
- $a = const$



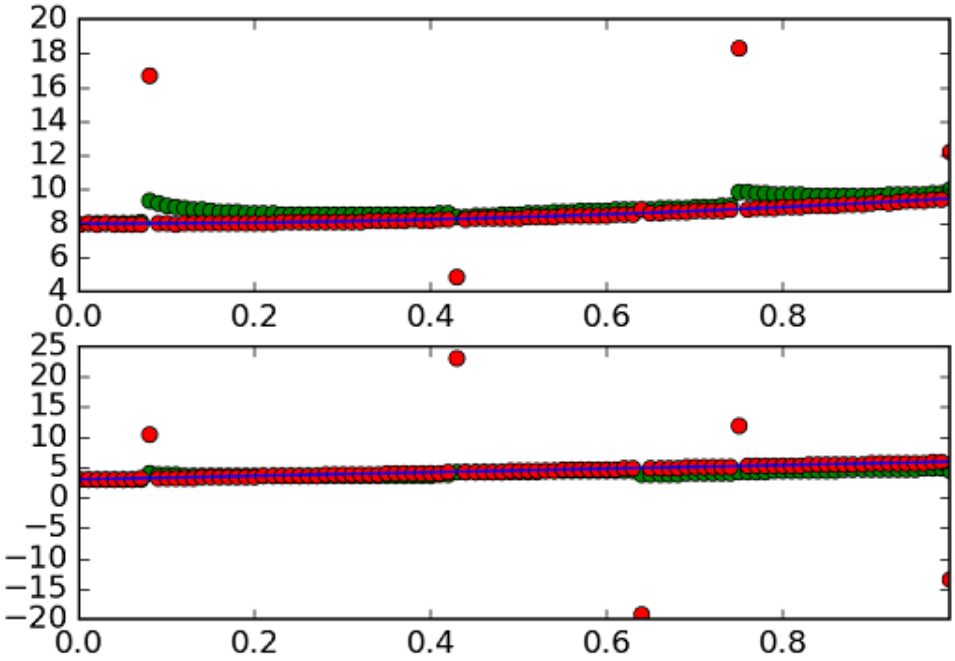
	Position	*Geschwindigkeit*
$MSE_{Messwerten}$	1.58898	13.01375
$MSE_{Schtzwerte}$	0.10623	0.12643
$Verbesserung$	0.93315	0.99029

1.4.6 Constant Acceleration

Wir erhalten mit dem im Weg-Zeit Transitionmodell folgende Ergebnisse:

$$Q = \begin{pmatrix} 0.000001 & 0 & 0 \\ 0 & 0.000000005 & 0 \\ 0 & 0 & 0.0000001 \end{pmatrix}$$

$$R = \begin{pmatrix} 0.0001333 & 0 \\ 0 & 000008333 \end{pmatrix}$$



	Position	*Geschwindigkeit*
$MSE_{Messwerten}$	1.84500	14.08193
$MSE_{Schtzwerte}$	0.26117	0.40673
$Verbesserung$	0.85844	0.97112

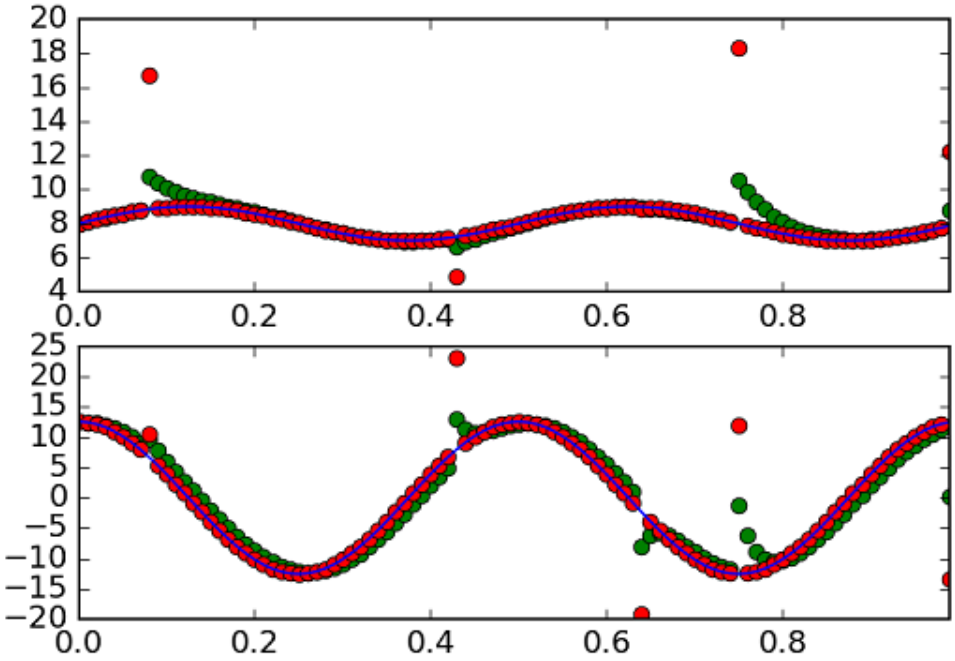
Auch hier erhalten wir für die Geschwindigkeit gute Verbesserung zu den Messdaten. Bei der Positionsbestimmung, bleiben die Messwerte jedoch stärker gewichtet, wodurch die zufälligen Ausreißer den Kalamn-Fitler mehr beeinflussen.

1.4.7 Sinus

Für die Sinus-Bewegung mussten wir die Q-Matrix stärker anpassen und erhalten folgende Ergebnisse:

$$Q = \begin{pmatrix} 0.00001 & 0 & 0 \\ 0 & 0.000003 & 0 \\ 0 & 0 & 0.00005 \end{pmatrix}$$

$$R = \begin{pmatrix} 0.0001333 & 0 \\ 0 & 0.000008333 \end{pmatrix}$$



	Position	*Geschwindigkeit*
$MSE_{Messwerten}$	1.92603	17.96788
$MSE_{Schtzwerte}$	0.27105	5.63545
$Verbesserung$	0.85927	0.68636

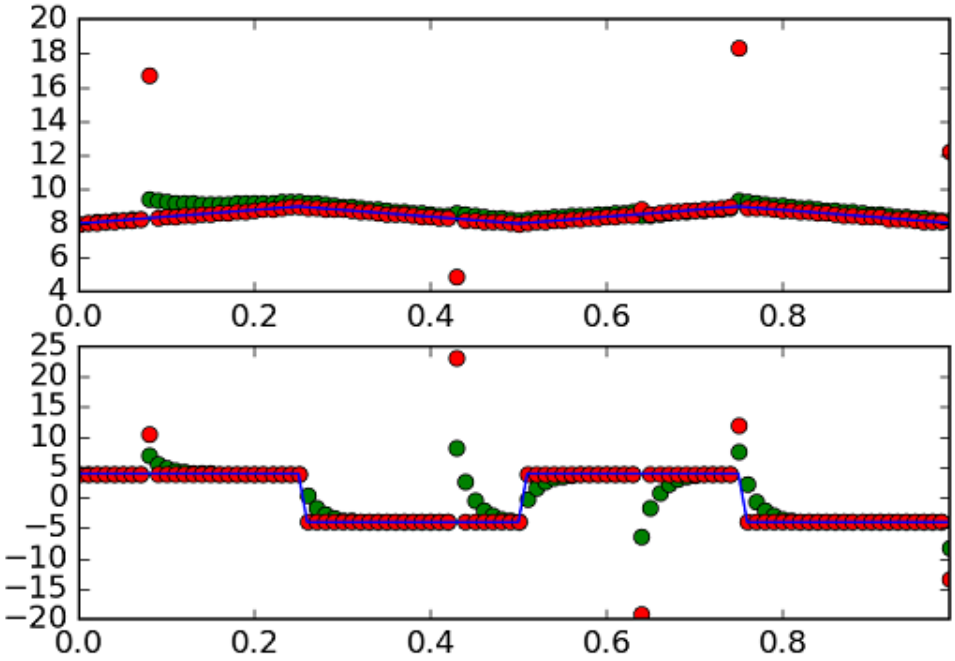
Für die Position-Messungen erhalten wir durch die Erkennugng der Außreißer deutlich bessere Ergebnisse. Jedoch reicht das Modell für die Geschwindigkeitsverbesserung nicht aus und wir erhalten bei Außreißern immer noch größere Unterschiede zum wahren Zustand.

1.4.8 Triangle

Für die Tirangle-Bewegung erhalten sind unsere besten Ergebnisse folgenden:

$$Q = \begin{pmatrix} 0.0000001 & 0 & 0 \\ 0 & 0.000003 & 0 \\ 0 & 0 & 0.001 \end{pmatrix}$$

$$R = \begin{pmatrix} 0.0001333 & 0 \\ 0 & 0.000008333 \end{pmatrix}$$



	Position	*Geschwindigkeit*
$MSE_{Messwerten}$	2.98635	19.48293
$MSE_{Schtzwerte}$	0.14755	6.09658
$Verbesserung$	0.95059	0.68708

Ähnlich wie die Sinus-Bewegung zeigt sich eine deutliche Besserung und interpetation von Mess-Ausreißer über die Zeit und eine schlechtere bei der Geschwindigkeit. Dies liegt wahrscheinlich daran, dass das Modell nicht in der lage ist die Sprünge zwischen den Ausreßern aussreichen zu approximieren.

1.5 3D Radarsensor Experiment

Um die Ergebnisse und den Vorteil des DBScan Algorithmus zu verdeutlichen, haben wir das Experiment in zwei Abschnitte unterteilt. Im ersten Abschnitt betrachten wir die unterschiedlichen Ergebnisse des Experiments nur mit dem Kalman-Filter, im zweiten Abschnitt dann mit vorgelagertem DBScan Algorithmus.

1.5.1 3D Radarsensor Experiment ohne DBScan

Zu Beginn wissen wir noch nicht, welche Werte für den 3D-Fall gut geeignet sind. Wir initialisieren das erste Kalman-Filter deshalb mit für uns plausiblen Werten und führen in damit aus.

Unser **Übergangsmodell** basiert auf dem im Skript gegebenen physikalischen Model der gleichmäßig beschleunigten Bewegung mit drei Parametern: * Beschleunigung * Geschwindigkeit * Position

Bei einer *measurementRate* von *100 Hz* sieht das zugrunde liegende Gleichungssystem demnach wie folgt aus:

$$\begin{pmatrix} 1 & 0.01 & 0.01/2 \\ 0 & 1 & 0.01 \\ 0 & 0 & 0.01 \end{pmatrix}$$

Für andere Werte für *measurementRate* lässt sich das Transitionsmodell problemlos anpassen. Im Originalen Skript wurde eine *measurementRate* von *30 Hz* verwendet.

[15]:
```
transition_model = np.array([[1, 0.01, 0.01/2],
                             [0, 1,    0.01  ],
                             [0, 0,    0.01 ]])
print(transition_model)
```

```
[[1.    0.01  0.005]
 [0.    1.    0.01 ]
 [0.    0.    0.01 ]]
```

Die **Q-Matrix** und damit unser Prozessrauschen setzen wir zunächst auf einen sehr kleinen Wert `0.02` für alle drei Parameter. Damit wollen wir erreichen, dass der Filter unserem Modell mehr vertraut als den Messungen.

$$\begin{pmatrix} 0.02 & 0 & 0 \\ 0 & 0.02 & 0 \\ 0 & 0 & 0.02 \end{pmatrix}$$

[16]:
```
Q = np.diag([0.002, 0.002, 0.002])
print(Q)
```

```
[[0.002 0.    0.   ]
 [0.    0.002 0.   ]
 [0.    0.    0.002]]
```

Die **R-Matrix** initialisieren wir mit folgendem Wert, wobei $\sigma = rangeAccuracy$, für das Messrauschen des Radarsensors:

$$\frac{\sigma^2}{3}$$

Wir wählen diesen Wert, da die Fehler des Sensors uniform verteilt sind und die Varianz einer uniform verteilten Zufallsvariable sich wie folgt berechnen lässt:

Sei $x \sim U[a, b]$, dann gilt $\mathrm{Var}(x) = \frac{1}{12} \cdot (b - a)^2$ da in unserem Falle $b = \sigma$ und $a = -\sigma$ ergibt sich die oben abgebildete Formel.

Da die vom Sensor gemessene Radialgeschwindigkeit nur zur Clusterbildung mit Hilfe des DBScan Algorithmus genutzt wird, spielt der *velocityAccuracy* Hyperparameter zur Berechnung von **R** keine Rolle.

[17]:
```
R = np.diag([rangeAccuracy**2])/3
print(R)
```

```
[[0.00083333]]
```

Zum Schluss initialisieren wir die **H-Matrix** wie folgt, sodass sie (alle drei) Parameter berücksichtigt:

$$\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$$

[18]:
```
H =  np.array([[1, 0, 0]])
print(H)
```

```
[[1 0 0]]
```

Die Einstellungen für unser *Target* belassen wir bei den uns gegebenen Werten:

[19]:
```
path = [[0,5,0],
        [0,5,0.5],
        [1,5,1],
        [1,5,0.5],
        [0.5, 2, 0.1]]

vel = 3 * numpy.ones((1,5))
vel[0,2] = 1

InitialPosition = numpy.array([-1,5,0])

opt = {
    'InitialPosition' : InitialPosition,
    'Path' : numpy.array(path).transpose(),
    'Velocities' : vel
}

x = Target(opt)

targets = [x]
```

Lediglich bei den Einstellungen des Radarsensors nehmen wir eine kleine Anpassung vor. Um die *False Detections* zu reduzieren, haben wir einen zusätzlichen Parameter zu den Optionen des 3D Radarsensors hinzugefügt. Der neue Parameter `falseDetectionsRange` bestimmt die *Range* der *False Detections* im *DataGenerationRadar3D* Skript.

[20]:
```
optRadar = {
    'Position' : numpy.array([0,0,0.5]),
    'OpeningAngle' : numpy.array([120,90]), # [Horizontal, Vertical]
    'FalseDetection': True,
    'falseDetectionsRange' : 10 # neuer Parameter für Range
```

```
}
sensor = RadarSensor(optRadar)
```

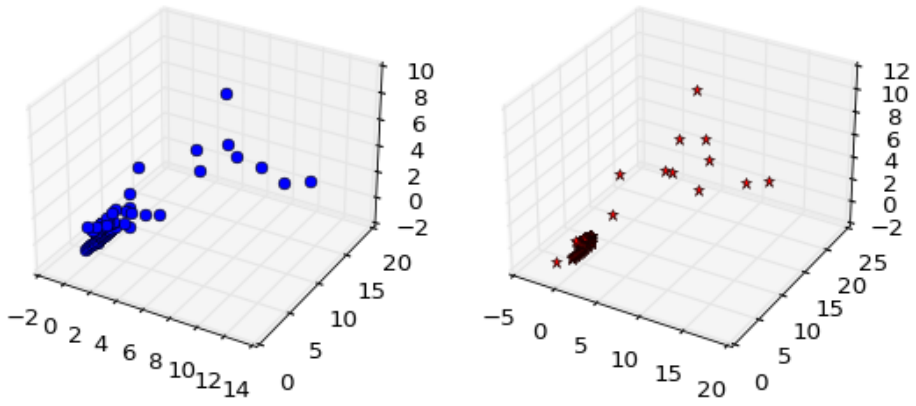
Anschließend iterieren wir mit nachfolgender Schleife über die *Detections* des 3D Radarsensors und plotten das Ergebnis:

[21]:
```
# Initialisierung benötigter Variablen
Detections = np.array([0,0,0,0])
i = 0
pred = []

getNext = True
while(getNext == True):

    for target in targets:
        target.Step(1/sensor.opt['MeasurementRate'])
        getNext = getNext & ~target.reachedEnd

    dets = sensor.Detect(targets)
    for det in dets:
        Detections = np.vstack((det, Detections))

    s0 = np.vstack((Detections[0,:-1], np.zeros((2,3))))

    if i == 0:
        f = KalmanFilter(s0, transition_model, H, Q, R)
        pred.append(s0[0,:])

    s = Detections[0,:-1].reshape(1,3)
    s_hat = f.step(s)
    pred = np.vstack((s_hat[0,:], pred))
    i += 1
```

In den beiden Abbildungen sehen wir die *Predictions* des Kalman-Filters getrennt von den *Detections* der Simulation:

[22]:
```
T1 = pred[:-1]

fig = plt.figure(figsize=(8, 4),facecolor='w')
ax1 = fig.add_subplot(121, projection='3d')
ax2 = fig.add_subplot(122, projection='3d')

ax1.plot3D(T1[:,0], T1[:,1], T1[:,2], 'bo')
ax2.plot3D(Detections[:,0], Detections[:,1], Detections[:,2], 'r*')
```

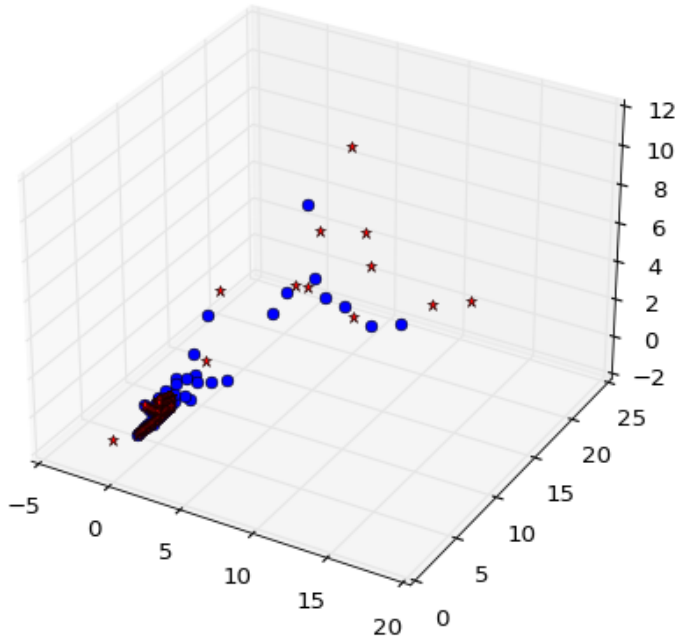
[22]:
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x17fdd5f1fa0>]
```



Legen wir beide Abbildungen zusammen erhalten wir folgenen Plot:

[23]:
```
fig = plt.figure(facecolor='w')
ax = plt.axes(projection='3d')
ax.plot3D(T1[:,0], T1[:,1], T1[:,2], 'bo')
ax.plot3D(Detections[:,0], Detections[:,1], Detections[:,2], 'r*')
```

[23]:
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x17fdd734c40>]
```



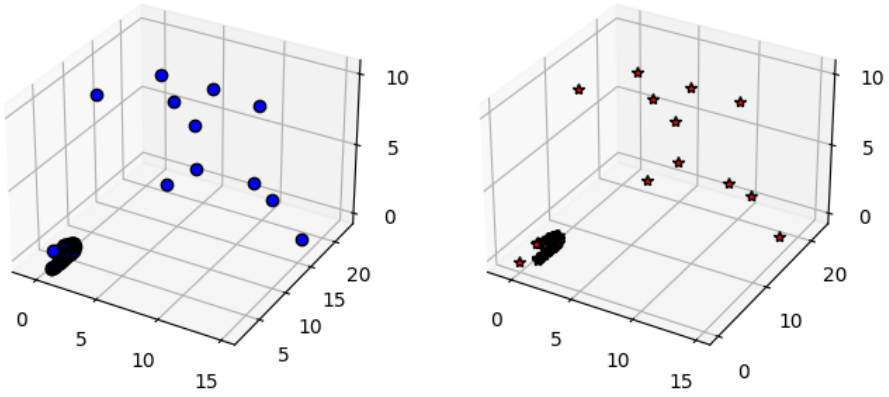
In diesem zusammengefügten Plot können wir nun deutlich erkennen, wie gut oder schlecht unser Kalman- Filter mit seiner aktuellen Initialisierung die Position unseres Objekts vorhersagt. Je näher die blauen Punkte des Kalman-Filters am Pfad liegen, desto besser funktioniert die initialisierte **Q-/R-Matrix**.

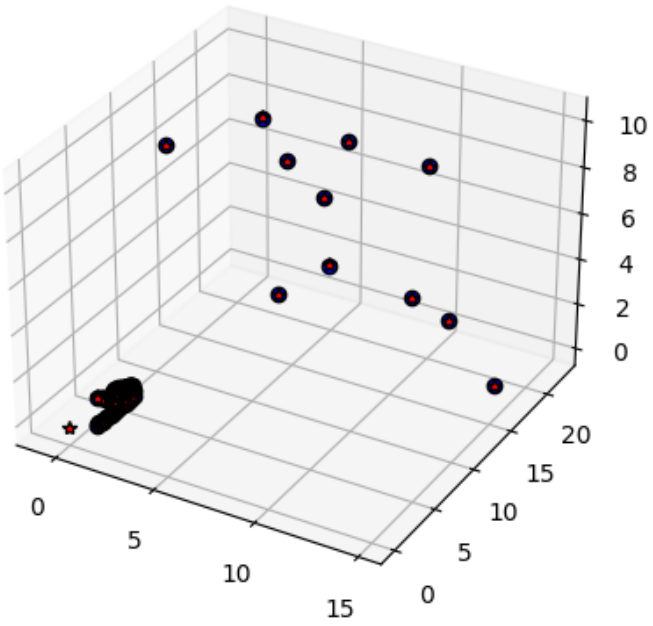
Im Folgenden werden wir die einzelnen Parameter des Kalman-Filter verändern und das Ergebnis dabei untersuchen, bis wir eine überzeugende Einstellung gefunden haben.

1.5.2 Verschiede Parameterwerte beim 3D Experiment

Als erstes verändern wir die **Q-Matrix**. Indem wir deren Werte erhöhen und damit mehr Rauschen zum Prozess hinzufügen.

```
Q = np.diag([0.5, 0.5, 0.5])
```

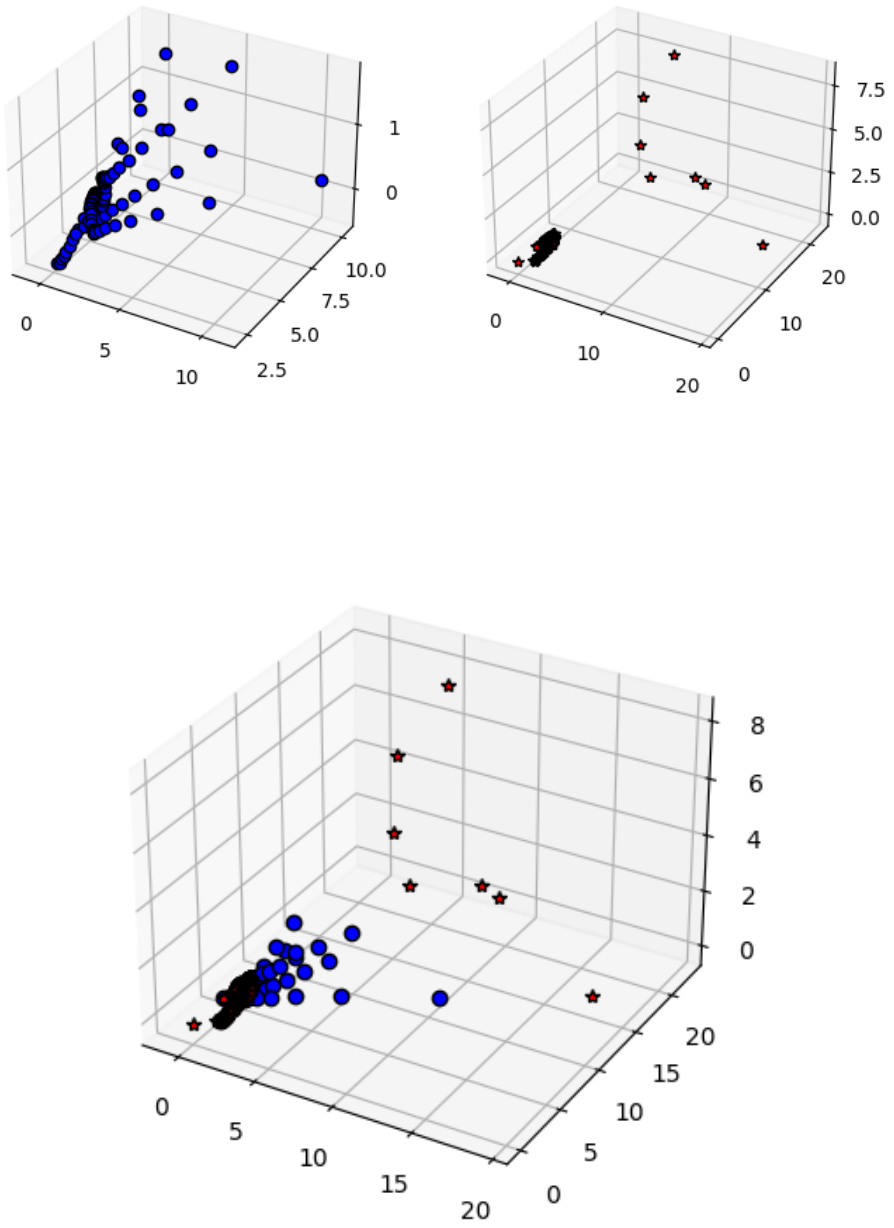




Wie man sehen kann, führt dies dazu, dass die blauen Punkte des Filters nahe an den roten Sternen des Radarsensors liegen, diese Einstellung für die **Q-Matrix** eignet sich also nicht besonders gut, da sie den Messungen ein zu großes Vertrauen schenkt.

Belassen wir nun die **Q-Matrix** bei einem Wert von `0.02` und erhöhen den Wert der **R-Matrix** auf `0.05`, sehen wir, dass die Vorhersagen des Filters bevorzugt werden. Wir sehen also sehr deutlich, dass die blauen Punkte des Filters nah am Pfad und weit von den Messungen des Sensors liegen. Dennoch ist die Einstellung nicht optimal, da sie zu großen Abweichungen vom Pfad führt.

```
Q = np.diag([0.02, 0.02, 0.02])
R = np.diag([0.05])
```



1.5.3 Der DBScan Algorithmus

Der DBSCAN Algorithmus findet vorhandene Cluster in einer Menge an Datenpunkten. Um diese Aufgabe zu erfüllen, hat der Algorithmus zwei verstellbare Parameter. Zum einen den Parameter `eps`, der bestimmt in welchem Radius um den Datenpunkt nach Nachbarn gesucht wird, und zum

anderen den Parameter `minpts`, der festlegt wie viele Punkte es minimal braucht, damit es sich um ein Kernobjekt handelt, um welches man ein Cluster aufbauen kann. Da sich die gefundenen Cluster mit dem verstellen der Parameter ändern können, müssen diese auf das jeweilige Problem angepasst werden.

Für den DBSCAN wurde eine Klasse erstellt, durch die der Algorithmus mit den beiden zuvor genannten Parametern initialisiert wird. Einmal initialisiert kann man den Algorithmus mit den gesetzten Parametern auf verschiedene Datensätze anwenden.

[24]:
```
class DBSCAN():
    def __init__(self, eps=0.5, minpts=5):
        self.eps = eps
        self.minpts = minpts
```

Um die Cluster in einem bestimmten Datensatz zu finden, besitzt die DBSCAN Klasse die Funktion `fit(self, X)`. Dabei entspricht `X` dem Datensatz, der analysiert werden soll. Innerhalb dieser Funktion wird der DBSCAN Algorithmus auf die Daten aus `X` angewendet und die einzelnen Datenpunkte werden als Kernobjekte, Dichte-erreichbare Objekte und Rauschpunkte kategorisiert.
Für die Bestimmung der Punkte Art wird der Abstand zwischen zwei Punkten benötigt. Um diesen Abstand zu bestimmen wird die Hilfsfunktion `pairwise_sq_distance` erstellt.

[25]:
```
def pairwise_sq_distance(X1, X2):
    # Calculate the pairwise distance between all pairs of points from X1 and␣
 ↪X2.
    return np.sum(X1**2, axis=1, keepdims=True) - 2*np.matmul(X1, X2.T) + np.
 ↪sum(X2**2, axis=1, keepdims=True).T
```

[26]:
```
def fit(self, X):
    dist = pairwise_sq_distance(X, X)
    neighbours = list(map(lambda d: np.arange(d.shape[0])[d < self.eps**2],␣
 ↪dist))

    # Label all points as outliers initially.
    self.assignment = np.full((X.shape[0],), -1, dtype=int)
    # Find core points.
    # Determine the number of neighbors of each point.
    N_neighbors = np.sum(dist < self.eps**2, axis=1)
    self.assignment[N_neighbors >= self.minpts] = -2

    # Create clusters.
    cluster = 0
    stack = deque()
    for p in range(X.shape[0]):
        if self.assignment[p] != -2:
            continue

        self.assignment[p] = cluster
```

```
        stack.extend(neighbours[p])
        # Expand cluster outwards.
        while len(stack) > 0:
            n = stack.pop()
            label = self.assignment[n]
            # If core point include all points in  -neighborhood.
            if label == -2:
                stack.extend(neighbours[n])
            # If not core point (edge of cluster).
            if label < 0:
                self.assignment[n] = cluster

        cluster += 1

DBSCAN.fit = fit
```

Die Funktion `fit` findet die verschiedenen Cluster und speichert diese in der Variable `assignments` ab. Das heißt, wenn wir die Methode `fit` aufrufen werden zuerst noch nicht die gefundenen Cluster zurückgegeben. Für diese Aktion existiert die `predict` Methode.
Zusätzlich zu der Basisfunktion `predict` gibt es auch die Methode `fit_predict`. Diese ruft zuerst `fit` auf, also findet die Cluster, und danach `predict`, also um die Werte zurückzugeben. Das heißt, wenn man den Algorithmus zum ersten Mal auf einen Datensatz anwendet und direkt das Ergebnis haben will, sollte die Funktion `fit_predict` verwendet werden, wenn man das Ergebnis des Algorithmus zu einem späteren Zeitpunkt nochmal benötigt muss nur noch `predict` ausgeführt werden. Kurz gesagt, kann durch die Aufsplittung von `fit` und `predict` Rechenaufwand reduziert werden. Außerdem ist diese Trennung gängige Praxis in ML Bibliotheken wie z.B scikit-learn.

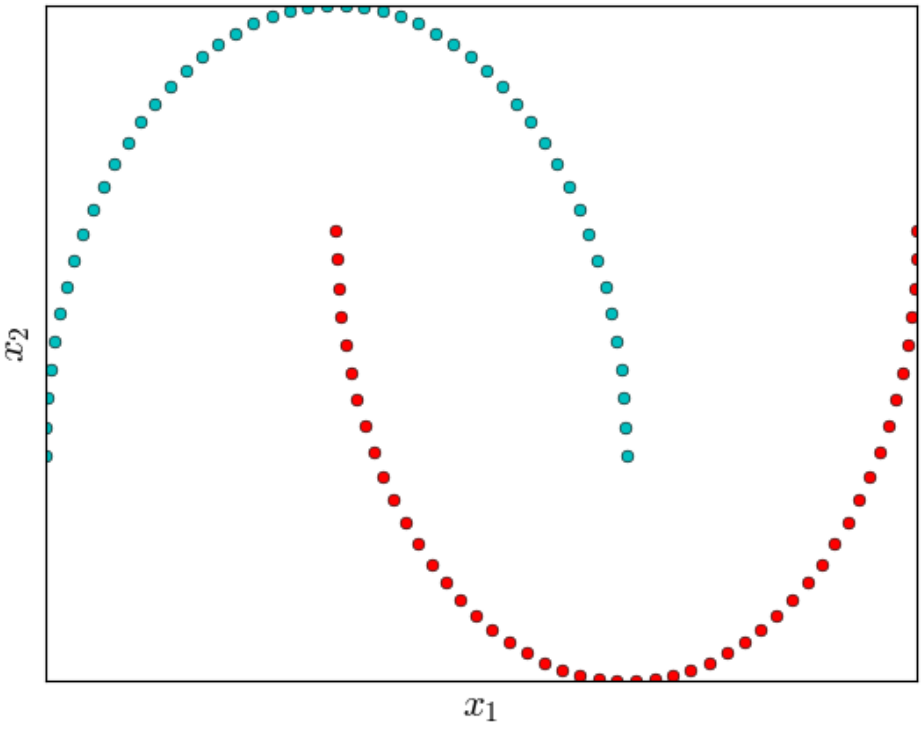
[27]:
```
def predict(self,X):
    return self.assignment

def fit_predict(self, X):
    self.fit(X)
    return self.assignment

DBSCAN.predict = predict
DBSCAN.fit_predict = fit_predict
```

Um den DBSCAN zu testen, haben wir den `make_moons` Datensatz von scikit-learn genutzt. Das Ergebnis kann man hier sehen:

[28]:
```
from sklearn.datasets import make_moons

X,y = make_moons(100)
model = DBSCAN()
preds = model.fit_predict(X)
# Either low or high values are good since DBSCAN might switch class labels.
print(f"Accuracy: {round((sum(preds == y)/len(preds))*100,2)}%")
```

```
fig= plt.figure(facecolor='w')
ax = plt.axes()
plt.plot(X[:, 0][preds==1], X[:, 1][preds==1], "co")
plt.plot(X[:, 0][preds==0], X[:, 1][preds==0], "ro")

# X contains two features, x1 and x2
plt.xlabel(r"$x_1$", fontsize=20)
plt.ylabel(r"$x_2$", fontsize=20)

# Simplifying the plot by removing the axis scales.
plt.xticks([])
plt.yticks([])

# Displaying the plot.
plt.show()
```

```
Accuracy: 0.0%
```



1.5.4 3D Radarsensor Experiment mit DBScan

In diesem Experiment sollen ein oder mehrere Ziele in einem drei dimensionalen Raum getrackt werden. Dafür werden der DBSCAN, zur Zielerkennung und der Kalman-Filter zur Laufbahnvorhersage genutzt.

In dem folgenden Code werden diese Ziele und ihre Laufbahnen angelegt:

[29]:
```
# Parameters first target.
path1 = [[0. , 5. , 0. ],
         [0. , 5. , 0.5],
         [1. , 4. , 1  ],
         [2. , 3. , 2. ],
         [1. , 5. , 3. ],
         [1. , 5. , 0.5],
         [0.5, 2. , 0.1]]

vel1 = 3 * np.ones((1,len(path1)))
vel1[0,2] = 1

InitialPosition1 = np.array([-1,5,0])

opt1 = {
    'InitialPosition' : InitialPosition1,
    'Path' : np.array(path1).transpose(),
    'Velocities' : vel1
}

# Parameters second target.
path2 = [[1. , 4. , 1. ],
         [1. , 5. , 1.7],
         [2. , 5. , 1. ],
         [3. , 4. , 2. ],
         [3. , 4. , 1.5],
         [2. , 4. , 2. ]]

vel2 = 2 * np.ones((1,len(path2)))
vel2[0,4] = 0.5

InitialPosition2 = np.array([2,4,1])

opt2 = {
    'InitialPosition' : InitialPosition2,
    'Path' : np.array(path2).transpose(),
    'Velocities' : vel2
}
```

Zusätzlich wird der Radarsensor initialisiert:

[30]:
```
'''
Setup the radar sensor
The radar sensor points always to the direction along the y axis
(see diagram in the note)
'''

optRadar = {
    'Position' : np.array([0,0,0.5]),
    'OpeningAngle' : np.array([120,90]), # [Horizontal, Vertical]
    'FalseDetection': True
}
sensor = RadarSensor(optRadar)
```

Die Bewegung der Ziele findet in einer while-Schleife statt. Dabei wird in jeder Iteration jedes Ziel einen Schritt weiter bewegt. Nachdem ein Schritt durchgeführt wurde versucht der Sensor die Ziele wahrzunehmen und mithilfe des DBSCAN die Ziele zu finden. Die Analyse durch den DBSCAN gibt allerdings erst ab einer in `pt_history` festgelegt Anzahl an Datenpunkten einen Sinn.
Sobald das erste Mal diese Anzahl erreicht wurde, wird der DBSCAN in jeder Iteration auf die letzten `pt_history` Datenpunkte angewendet. Die dabei gefundenen Cluster werden den verschiedenen Zielen aus `labeled` zugeordnet. Existiert das Ziel noch nicht in diesem Dictionary wird es erstellt.

[31]:
```
def scan(model, pt_history, targets):
    getNext = True
    detections = np.array([0,0,0,0])
    # Count number of iterations
    i = 0
    labeled = {}

    while(getNext == True):
        i += 1
        for target in targets:
            target.Step(1/sensor.opt['MeasurementRate'])
            getNext = getNext & ~target.reachedEnd

        dets = sensor.Detect(targets)
        for det in dets:
            detections = np.vstack((det, detections))

        if i >= pt_history:
            # First application of DBSCAN.
            clusters = model.fit_predict(detections[:pt_history])
            # Determine number of targets (objects tracked).
            num_objs = set(clusters)

            for j in num_objs:
```

```
                # find index of first occurence of target j in clusters. This␣
↪line is needed to filter out false detections
                obj_idx = np.where(clusters == j)[0][0]
                try:
                    last_obj_idx = np.where(clusters == j)[0][1]
                except:
                    last_obj_idx = -1

                # add new cluster if it does not exist yet
                if j not in labeled.keys():
                    labeled[j] = [detections[obj_idx]]
                else:
                    # try to check if label swap occured + correct it
                    # check if last detection is in the cluster if its not an␣
↪outlier and we had enough found clusters (last_obj_idx)
                    if (not detections[last_obj_idx] in labeled[j]) and␣
↪(last_obj_idx != -1):
                        for l in labeled.keys():
                            if detections[last_obj_idx] in labeled[l]:
                                j = l
                                break


                s = detections[obj_idx]
                labeled[j] = np.vstack((s, labeled[j]))

    return labeled
```

In einigen Fällen, beispielsweise wenn ein neues Target in dem Sichtbereich des Sensors fliegt, kann es zu einem "label swap" innerhalb des DBSCAN kommen. Dieser entsteht dadurch, dass der DBSCAN dem ersten Cluster das er findet die id 1 gibt. Wenn nun ein neues Objekt in den Sichtbereich kommt bekommt dieses automatisch die id 1, anstatt der id 2 die in diesem Falle besser wäre (da es ein neues Cluster ist). Um diese Problematik abzufangen, existiert der folgende Code innerhalb der while schleife:

```
if j not in labeled.keys():
    labeled[j] = [detections[obj_idx]]
else:
    # try to check if label swap occured + correct it
    # check if last detection is in the cluster if its not an outlier and we had enough found
    if (not detections[last_obj_idx] in labeled[j]) and (last_obj_idx != -1):
        for l in labeled.keys():
            if detections[last_obj_idx] in labeled[l]:
                j = l
                break
```

In diesem wird überprüft, zu welchem Cluster der letzte Punkt (nicht welcher jetzt klassifiziert wird) gehört. Befindet er sich im gleichen Cluster ist alles gut, befindet er sich in einem anderen

wird das Label des zu klassifizierenden Punktes angepasst!
Dadurch werden alle Datenpunkte dem jeweils richtigen Cluster zugeordnet.

Sobald die Datenpunkte visualisiert werden kann man erkennen, dass einige Datenpunkte Ausreißer sind und der Rest Pfade verschiedener Objekte darstellen. Über das folgende Dictionary wird festgelegt welches Objekt welche Farbe bekommen soll (Es sind so viele, denn je nach Einstellung der Parameter werden mehr als die zwei realen Objekte gefunden).

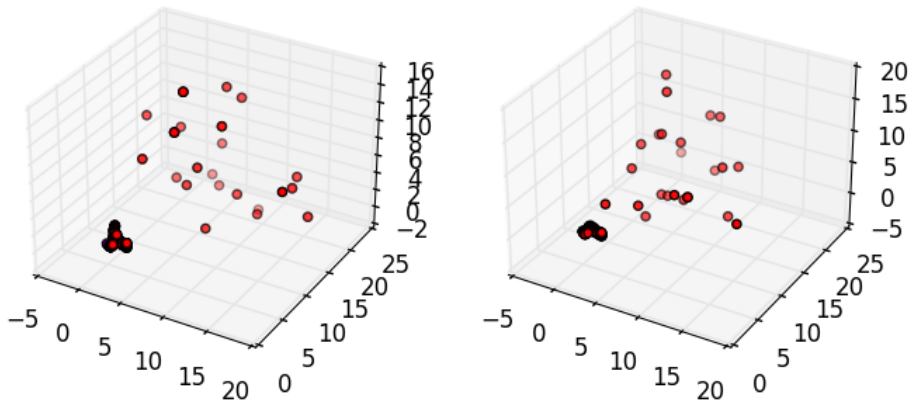
[32]:
```
colors = { -1: 'red', 0: 'green', 1: 'yellow', 2: 'blue', 3: 'purple', 4:␣
 ↪'orange', 5: 'pink', 6: 'black', 7: 'brown' }
```

Um eine gute Einstellung für den DBSCAN zu finden, werden zuerst zwei extreme Einstellungen getestet. Bei der ersten (links) Einstellungen werden die Parameter niedrig und bei der zweiten Einstellung (rechts) hoch initialisiert. Dabei wird die Variable `pt_history` bei einem Wert von konstant 20 belassen (ein Erhöhen dieser Variable entspricht einem Verringern von `minpts` und ein Verringern der Variable entspricht einer Erhöhung von `minpts` (das kann im interaktiven Beispiel am Ende der Sektion überprüft werden)).

[33]:
```
model1 = DBSCAN(eps=0.1, minpts=2)
model2 = DBSCAN(eps=1, minpts=10)
# Number of previous measurements to consider for DBSCAN().
pt_history = 20

# Instantiate targets
x = Target(opt1)
y = Target(opt2)

targets = [x, y]

# Plot Trajectories
fig= plt.figure(figsize=(8,6), dpi= 100, facecolor='w')
labeled1 = scan(model1, pt_history, targets)
print(f'Found {len(labeled1.keys())-1} clusters and {len(labeled1[-1])} outlier!
 ↪')

ax = fig.add_subplot(1, 2, 1, projection='3d')
for label in labeled1.keys():
    T = labeled1[label];
    ax.scatter(T[:,0], T[:,1], T[:,2], c=f'{colors[label]}')


# Instantiate targets
x = Target(opt1)
y = Target(opt2)

targets = [x, y]

labeled2 = scan(model2, pt_history, targets)
```

```
print(f'Found {len(labeled2.keys())-1} clusters and {len(labeled2[-1])} outlier!
 ↪')

# Plot Trajectories
ax = fig.add_subplot(1, 2, 2, projection='3d')
for label in labeled2.keys():
    T = labeled2[label];
    ax.scatter(T[:,0], T[:,1], T[:,2], c=f'{colors[label]}')

# show plot
plt.style.use('classic')
plt.show()
```

```
Found 7 clusters and 406 outlier!
Found 2 clusters and 123 outlier!
```



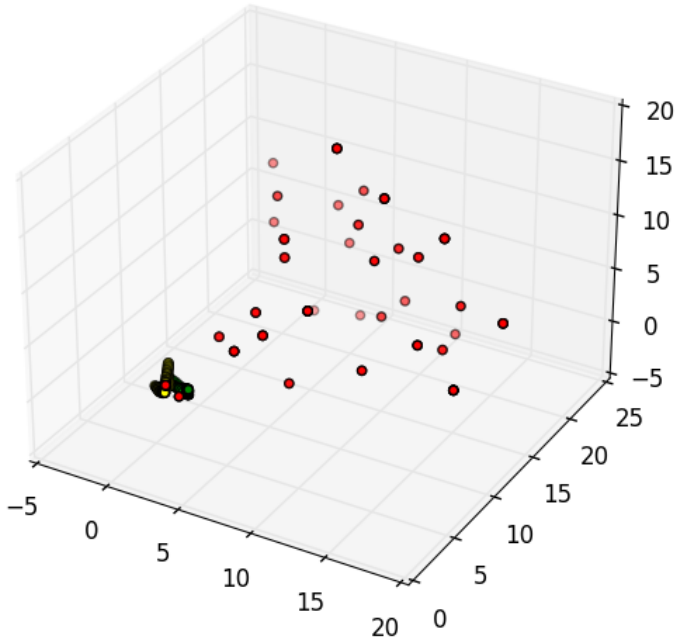
Das Ergebnis dieses Tests ist, dass bei beiden Einstellungen etwas nicht stimmt. Bei der ersten zu niedrigen Einstellung werden zu viele Cluster gefunden. Das liegt vermutlich an einem zu niedrigen Wert für `eps`, denn dadurch werden nur die Punkte mit einer so geringen Entfernung einem Cluster zu geordnet => `eps` erhöhen. Ähnliches passiert bei der zweiten Einstellung. Bei dieser kann zwar Objekte in einem größeren Umkreis finden (was auch passiert), aber er braucht deutlich mehr

`minPts` bevor diese zu einem Cluster zählen => `minPts` verringern.

Ergebend aus unseren ersten beiden Tests ergibt sich das ein für uns gutes Ergebnis zwischen den beiden Extrema liegen muss. Also `eps` hoch und `minPts` runter (bei der Einstellung von `pt_history` = 20). Durch mehrere Tests derselben Art hat sich die Einstellung `eps = 0.7` und `minPts = 2` für uns als gut ergeben.

[34]:
```
model = DBSCAN(eps=0.7, minpts=2)
# Number of previous measurements to consider for DBSCAN().
pt_history = 20

# Instantiate targets
x = Target(opt1)
y = Target(opt2)

targets = [x, y]

labeled = scan(model, pt_history, targets)
print(f'Found {len(labeled.keys())-1} clusters and {len(labeled[-1])} outlier!')

# Plot Trajectories
fig= plt.figure(figsize=(8,6), dpi= 100, facecolor='w')
ax = plt.axes(projection='3d')
for label in labeled.keys():
    T = labeled[label]
    ax.scatter(T[:,0], T[:,1], T[:,2], c=f'{colors[label]}')

# show plot
plt.show()
```

```
Found 2 clusters and 107 outlier!
```



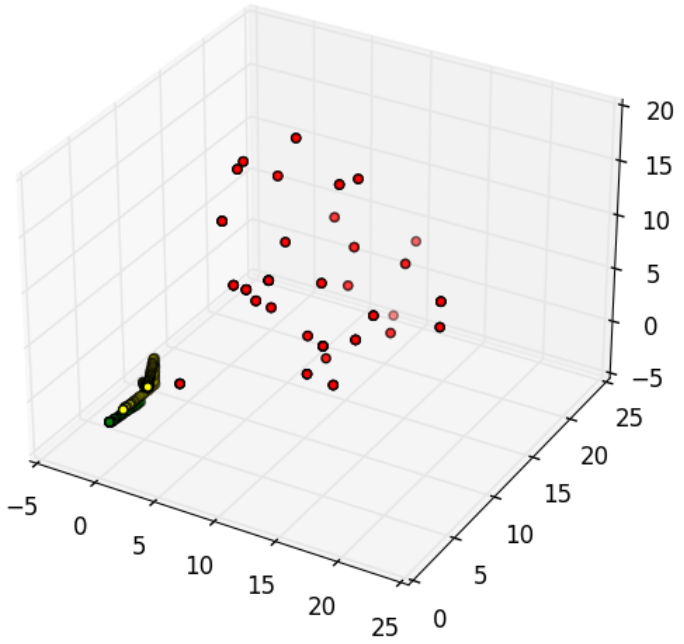
Wie zu sehen, werden die beiden Objekte gut durch den DBSCAN erkannt und die Ausreißer erfolgreich rausgefiltert.

Zusätzlich wird noch überprüft, wie sich der Algorithmus verhält, wenn ein neues Target zu einem späteren Zeitpunkt erscheint. Dafür befindet sich das Objekt zuerst außerhalb der Sicht des Sensors und fliegt dann hinein.

[35]:
```
# Parameters second target.
path3 = [[-5. , -5. , -5. ],
         [-5. , -4. , -5. ],
         [-2. , -3. , -1. ],
         [-1. , -4. , -2. ],
         [1. , 2. , 1.5],
         [1. , 2. , 2. ]]

vel3 = 2 * np.ones((1,len(path2)))
vel3[0,4] = 0.5

InitialPosition3 = np.array([2,4,1])

opt3 = {
```

```
    'InitialPosition' : InitialPosition3,
    'Path' : np.array(path3).transpose(),
    'Velocities' : vel3
}

'''
Setup the radar sensor
The radar sensor points always to the direction along the y axis
(see diagram in the note)
'''

optRadar = {
    'Position' : np.array([0,0,0.5]),
    'OpeningAngle' : np.array([120,90]), # [Horizontal, Vertical]
    'FalseDetection': True
}
sensor = RadarSensor(optRadar)

model = DBSCAN(eps=0.7, minpts=2)
# Number of previous measurements to consider for DBSCAN().
pt_history = 20

# Instantiate targets
x = Target(opt1)
y = Target(opt3)

targets = [x, y]

# labeled, detections = scan(model, pt_history, targets)
labeled = scan(model, pt_history, targets)
print(f'Found {len(labeled.keys())-1} clusters and {len(labeled[-1])} outlier!')

# Plot Trajectories
fig= plt.figure(figsize=(8,6), dpi= 100, facecolor='w')
ax = plt.axes(projection='3d')
for label in labeled.keys():
    T = labeled[label]
    ax.scatter(T[:,0], T[:,1], T[:,2], c=f'{colors[label]}')

# show plot
plt.show()
```

```
Found 2 clusters and 137 outlier!
```



Wie man sehen kann, verhält sich der Scan wie zu erwarten und klassifiziert das neue Objekt unabhängig des ersten Objekts. Dadurch ist auch zusehen, dass der “label swap” verhindert wurde.

Die Ergebnisse des DBSCAN können jetzt mit dem Kalman-Filter verbunden werden um größere Messfehler auszugleichen. Dafür wird für jedes Cluster ein neuer Filter angelegt, und zusätzlich zu den so gefundenen Datenpunkten, Vorhersagen angelegt. Die Liste detections enthält dabei für jedes Cluster eine Liste mit Sensormessungen der dazugehörigen Objekte. Die Radialgeschwindigkeit (letzter Eintrag der Messung) wird dabei nur für das clustern benötigt und somit nicht an den Filter weitergegeben. Das ganze funktioniert über die folgenden Zeilen:

```
s0 = np.vstack((detections[obj_idx,:-1], np.zeros((2,3))))
filters[j] = KalmanFilter(s0, transition_model, H, Q, R)
predictions[j] = [s0[0,:]]
```

Zusätzlich zur initialen Vorhersage werden mit den folgenden Zeilen nach jedem Schritt neue Punkte vorhergesagt:

```
s = filters[j].step(detections[obj_idx,:-1])
predictions[j] = np.vstack((s[0,:], predictions[j]))
```

[36]:
```
# Use DBSCAN to track the paths and Kalman to correct the measurment errors
def scan_with_filter(model, pt_history, targets, R, Q, transition_model, H):
    getNext = True
```

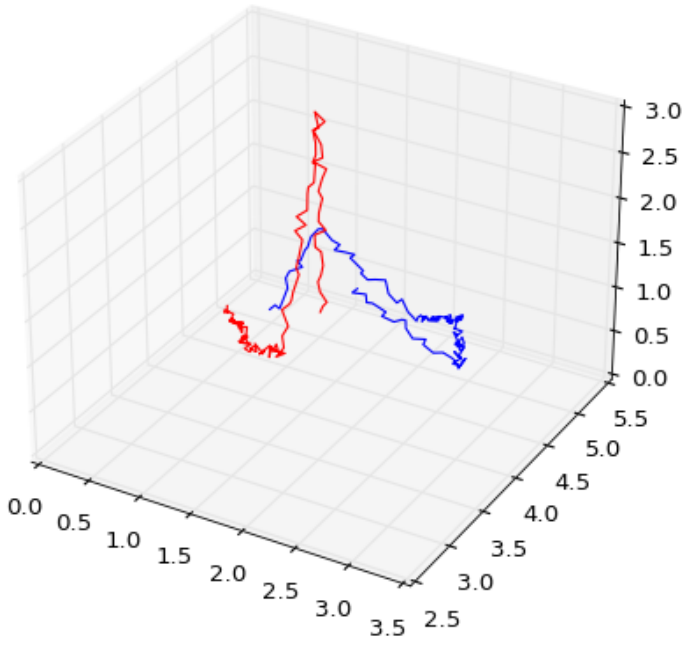
```
    detections = np.array([0,0,0,0])
    # Count number of iterations
    i = 0
    labeled = {}
    predictions = {}
    filters = {}

    while(getNext == True):
        i += 1
        for target in targets:
            target.Step(1/sensor.opt['MeasurementRate'])
            getNext = getNext & ~target.reachedEnd

        dets = sensor.Detect(targets)
        for det in dets:
            detections = np.vstack((det, detections))

        if i >= pt_history:
            # First application of DBSCAN.
            clusters = model.fit_predict(detections[:pt_history])
            # Determine number of targets (objects tracked).
            num_objs = set(clusters)

            for j in num_objs:
                # find index of first occurence of target j in clusters. This line is needed to filter out false detections
                obj_idx = np.where(clusters == j)[0][0]
                try:
                    last_obj_idx = np.where(clusters == j)[0][1]
                except:
                    last_obj_idx = -1

                # add new cluster if it does not exist yet
                if j not in labeled.keys():
                    labeled[j] = detections[obj_idx]
                    s0 = np.vstack((detections[obj_idx,:-1], np.zeros((2,3))))
                    filters[j] = KalmanFilter(s0, transition_model, H, Q, R)
                    predictions[j] = [s0[0,:]]
                else:
                    # try to check if label swap occured + correct it
                    # check if last detection is in the cluster if its not an outlier and we had enough found clusters (last_obj_idx)
                    if (not detections[last_obj_idx] in labeled[j]) and (last_obj_idx != -1):
                        for l in labeled.keys():
                            if detections[last_obj_idx] in labeled[l]:
                                j = l
```

```
                                break


                s = filters[j].step(detections[obj_idx,:-1])
                predictions[j] = np.vstack((s[0,:], predictions[j]))
                labeled[j] = np.vstack((detections[obj_idx], labeled[j]))

    return predictions, labeled
```

Scan und Filter werden jetzt auf zwei Targets angewendet:

[37]:
```
# Instantiate targets
x = Target(opt1)
y = Target(opt2)

targets = [x, y]

# Measurement error.
## Variance of a uniform distribution is given by (b-a)**2/12.
R = np.diag([rangeAccuracy**2])/3
# Process error.
Q = np.diag([0.05,0.05,0.05])
# Process/transition model.
transition_model = np.array([[1, 0.01, 0.01/2],
                             [0, 1, 0.01],
                             [0, 0, 0.01]])
# Transformation matrix
## Transforms predicted quantities into outputs that can be compared to the␣
 ↪measurements
H =  np.array([[1., 0., 0.]])


getNext = True
Detections = np.array([0,0,0])
model = DBSCAN(eps=0.7, minpts=2)
# Number of previous measurements to consider for DBSCAN().
ante = 20

Preds, labeled = scan_with_filter(model, ante, targets, R, Q, transition_model,␣
 ↪H)

# Visualize trajectory.
T1 = Preds[0]
T2 = Preds[1]

# Plot Trajectory
fig = plt.figure(facecolor='w')
```

```
ax = plt.axes(projection='3d')
#ax.view_init(20, 35)
ax.plot3D(T1[:,0], T1[:,1], T1[:,2], 'blue')
ax.plot3D(T2[:,0], T2[:,1], T2[:,2], 'red')

# show plot
plt.show()
```



Im Ergebnis ist zu sehen, dass der DBSCAN und der Kalman-Filter die Objekte erfolgreich tracken können.

1.5.5 Vergleich von Sensor und Kalman-Filter

Um eine quantitative Aussage über die Leistung des Kalman-Filters treffen zu können, wurde die Distanzen der vom Kalman-Filter vorhergesagten Positionen zu den wahren Positionen der targets als Kriterium ausgewählt. Die mittlere quadratische Abweichung der Distanzen dient anschließend als Grundlage für die Qualitätsbewertung. Als Benchmark wurde die mittlere quadratische Abweichung der Sensormessungen von den wahren Positionen herangezogen.

[38]:
```
def mse(paths: list, true_paths: list):
    """
```

```
    Predict the mean square error between a path prediction and the true path␣
↪of the objects

    """

    Diffs = []
    total_elements = 0
    # For-loop: Simple difference between a predicted path and a true␣
↪measurement.
    for i, true_path in enumerate(true_paths):
        diff = true_path[-len(paths[i]):] - paths[i]
        Diffs.append(diff)
        total_elements += len(diff)


    # For-loop: Square and average the differences.
    mse = 0
    for i, diff in enumerate(Diffs):
        # List diff is mapped to an integer (mse).
        mse += np.sum(diff**2)/ np.prod(diff.shape) * (len(diff)/total_elements)

    return mse
```

[39]:
```
# Sensor measurements.
## Only the position (first three coordinates) is of interest for the mse.
sensor_dets1 = labeled[0][:,:-1]
sensor_dets2 = labeled[1][:,:-1]

# True paths.
## Sensor measures position of object relative to it hence this quantity needs␣
 ↪to be subtracted.
true_path_x = np.array(x.Trajectory) - sensor.opt["Position"]
true_path_y = np.array(y.Trajectory) - sensor.opt["Position"]

T1_true = true_path_x.reshape(-1,3)
T2_true = true_path_y.reshape(-1,3)

# Determine mse.
mse_KF = mse([T1,T2],[true_path_x, true_path_y])
mse_Sensor = mse([sensor_dets1, sensor_dets2],[true_path_x, true_path_y])
print(f"Mean squared Error of Filter: {mse_KF}")
print(f"Mean squared Error of Sensor: {mse_Sensor}")
```

```
Mean squared Error of Filter: 1.2421709650079507
Mean squared Error of Sensor: 1.2417912988052668
```

Unter den Standardeinstellungen des Sensors ergibt sich kein signifikanter Unterschied zwischen dem MSE des Kalman-Filters und dem des Sensors. Erst ab *rangeAccuracy* Werten von über

0.3 m/s lässt sich eine signifikante Verbesserung der Positionsgenauigkeit durch das Kalman-Filter feststellen.

1.5.6 Nutzung der Radialgeschwindigkeit zur Verbesserung des Kalman-Filters

Um die Genauigkeit des Kalman-Filters zu verbessern, kam die Idee auf, die Radialgeschwindigkeit des Sensors zu verwenden, um davon den Geschwindigkeitsvektor der `targets` abzuleiten. Die Radialgeschwindigkeit gibt an, wie schnell sich ein Objekt direkt in Richtung des Beobachters bewegt. Da sich aus dieser eindimensionalen Größe nicht unmittelbar der dreidimensionale Geschwindigkeitsvektor berechnen lässt wurde versucht die vorletzte Messung des Sensors heranzuziehen um somit den wahren Geschwindigkeitsvektor zu approximieren.

Beispiel

[40]:
```
# j has to be an integer corresponding to a cluster generated by DBScan.
j = 1
vel = (labeled[j][0][:-1] - labeled[j][1][:-1])
print(vel.shape)
```

```
(3,)
```

Übernimmt man `vel` ohne weitere Nachbereitung als weiteren Input für den Kalman-Filter (nach Anpassung der R und H Matrix und Skalierung mit der Messfrequenz des Sensors) so führt dies trotzdem zu einer signifikanten Verschlechterung der Vorhersagen des Kalman-Filters, wobei der MSE sich zum Teil versechsfachte.

Um diesem Problem Herr zu werden, standen zwei Ansätze im Raum. Beim ersten Ansatz wurde die zu `vel` korrespondierende Radialgeschwindigkeit ermittelt und anschließend mit der vom Sensor vorhergesagten Radialgeschwindigkeit verglichen. Lag die Differenz der beiden Größen dabei unter 10 Prozent der vom Sensor vorhergesagten Radialgeschwindigkeit, so wurde `vel` als zusätzlicher Input für den Kalman-Filter verwendet. Im anderen Falle wurde nur die Sensorposition übergeben. Der Kalman-Filter wurde dahingehen erweitert, dass je nach Dimension des Inputs eine andere H und R Matrix verwendet wurde.

Beim zweiten Ansatz wurde `vel` so skaliert, dass die daraus berechnete Radialgeschwindigkeit anschließend der vom Sensor vorhergesagten Radialgeschwindigkeit entspricht. In diesem Falle wurde die skalierte Geschwindigkeit immer zusätzlich zur vom Sensor gemessenen Position an den Kalman-Filter übergeben.

Obwohl beide Ansätze dafür sorgen, dass der MSE sich im Vergleich zur Nutzung des unskalierten Geschwindigkeitsvektors nicht mehr verschlechtert, so führen sie jedoch ebenfalls nicht zu einer bemerkbaren Verbesserung im Vergleich zur ausschließlichen Nutzung der vom Sensor gemessenen Position als Eingabe für das Kalman-Filters.

1.6 Interaktiver Teil (Jupyter Notebook)

Um diesen Teil nutzen zu können, muss das Jupyter Notebook im Browser ausgeführt werden

1.6.1 Interaktives Kalman-Filter 1D

Im interaktiven Kalman-Filter 1D können sämtliche Parameter des 1D Experiments angepasst werden und dadurch die unterschiedlichen Auswirkungen auf das Ergebnis beobachtet werden.

[41]:
```
# interactive1DExperiment.plot_interactive_kalaman_filter()
```

1.6.2 Interaktiver DBScan

In dem folgenden Beispiel können die Parameter `eps`,`minPts` und `PT History` verstellt werden, um zu sehen, wie die verschiedenen Einstellungen das Ergebnis des DBScan beeinflussen.

[42]:
```
# interactiveDBScan.plot_interactive_dbscan()
```

1.6.3 Interaktiver Kalman-Filter mit DBScan 3D

Im interaktiven Kalman-Filter 3D können die Parameter des DBScan angepasst werden und dadurch die unterschiedlichen Auswirkungen auf das Ergebnis beobachtet werden. Zusätzlich können bis zu vier *Targets* gleichzeitig ausgewählt werden und für einzelne *Targets* können auch die *False Detections* angezeigt werden. Der rote Punkt zeigt die Position des Radarsensors an.

[43]:
```
# interactive3DExperiment.plot_3DExperiment()
```

1.7 Schlussfolgerung und Ausblick

Unsere Implementation des Kalman-Filters liefert enorme Verbesserungen im Vergleich zu den reinen Sensormessdaten im eindimensionalen Fall. Im dreidimensionalen Fall sind die Verbesserungen bei steigendem Messfehler des Sensors zumindest spürbar, wenn auch definitiv weniger stark als im eindimensionalen Fall. Die Kombination des Kalman-Filters mit dem DBSCAN Algorithmus eröffnet die Möglichkeiten auch bei starkem Messrauschen noch sinnvolle Positionsvorhersagen zu liefern.

Da der Kalman-Filter bei der Vorhersage des nächsten Zustands jeweils nur die letzte Observation direkt in Betracht zieht könnte es sich anbieten bei präzisionskritischen Anwendungen auch weiter zurückliegende Observationen mit in Betracht zu ziehen. Solche den Begriff “smoothing” fallende Techniken werden bereits in vielen praxiserprobte Implementationen verwendet.

Da unsere Implementation des Kalman-Filters nur mit synthetischen Daten gearbeitet hat, wäre der nächste naheliegende Schritt eine Anwendung auf echte Daten, die mit Hilfe eines eigenen Sensors generiert werden könnten.

1.8 Verwendete Literatur

1. Kalman and Bayesian Filters in Python, 2015, Roger R. Labbe
2. KalmanFilter.NET

1.9 Anhang

1.10 GitHub Workflow

Um den Code für unser Projekt zu verwalten, haben wir ein privates GitHub Repository verwendet. Sofern die Berechtigung im Voraus erteilt wurde, ist das Repository unter folgendem Link erreichbar: https://github.com/otiofrui/pml

1.11 Jupyter Notebook

Da sowohl das Kalman-Filter, als auch der DBScan und die Radarsensor Simulation viele Einstellungsmöglichkeiten bieten, haben wir uns dafür entschieden unseren Bericht mithilfe eines Juypter Notebooks interaktiv zu gestalten. Dies ermöglicht es dem Leser die Parameter der Programme selbst anzupassen und das Ergebnis so zu beeinflussen.