Проект InMotion

Результат разработки фильтра Калмана

Выполнил: Чашков М. С.

©FabLiteElectronics
2020

Оглавление

Общее описание алгоритма фильтра Калмана

Данный документ содержит пояснения использования Фильтра Калмана (ФК) для сглаживания триангулированных данных. Работа выполнена с использованием [3], [2], [4]

ФК построен на итеративном повторе двух процедур:

1. *Процедура предсказания*. На основе математической модели поведения системы и вектора состояния системы в предыдущий момент времени выполняется предсказания вектора состояния системы в текущий момент времени.
2. *Процедура коррекции.* На основе измеренных данных корректируется предсказанный вектор состояния системы.

Опишем обе эти процедуры математически.

Пусть x – вектор состояния системы, тогда

$$s_k = F_k s_{k-1} + B_k u_{k-1} + w_k \tag{1}$$

$$z_k = H_k s_k + v_k \tag{2}$$

В этих формулах:

s_k — вектор состояния системы в текущий момент времени размерностью n

F_k – матрица описывающая переход наблюдаемого процесса из состояния s_{k-1} в состояние s_k. Размерность матрицы n x n

u_k – вектор управляющих воздействий на процесс, размерностью k

B_k – матрица отображает вектор управляющих воздействий в изменение состояния s_k. Размерность n x k.

w_k - случайная величина описывающая погрешность исследуемого процесса. Причем плотность распределения вероятности (ПР) (pdf) случайной величины должна многомерной гауссовой с нулевым математическим ожиданием [1]

$$p(w) = N(0, Q) \tag{3}$$

где[5]

$$N(x, \mu, Q) = (2\pi)^{-\frac{n}{2}} |Q|^{-\frac{1}{2}} e^{-\frac{1}{2}(x-\mu)^T Q^{-1}(x-\mu)} \tag{4}$$

Здесь Q – ковариационная матрица погрешностей системы

z_k – вектор измерений системы, размерностью m

H_k – матрица отображения состояния системы в вектор измерений. Размерность матрицы H m x n

v_k - случайная величина описывающая погрешность исследуемого процесса. Причем ПР случайной величины также должна являться многомерной гауссовой с нулевым математическим ожиданием

$$p(v) = N(0, P) \tag{5}$$

Здесь P – ковариационная матрица погрешностей измерения

В качестве модели движения объекта будем рассматривал модель движения с равномерным ускорением (4 порядок)

$$r_k = r_{k-1} + v_{k-1} dt_k + a_{k-1} \frac{dt_k^2}{2} + e_{k-1} \frac{dt_k^3}{6} \quad (6)$$

Поскольку дополнительная информация о процессе движения отсутствует, считаю, что управляющего воздействия нет. Следовательно матрица B = 0

Введем матрицу ошибки изменения компонента e в виде

$$err_k = \begin{bmatrix} errE_k^x \\ errE_k^y \end{bmatrix} \quad (7)$$

Будем считать, что ошибка изменения компоненты e является двумерной гауссовой случайной величиной со средним квадратом σ_e. Иными словами

$$p(err_k^{x,y}) = N(0, \sigma_e) \quad (8)$$

С учетом предположений уравнение (3) примет следующий вид

$$s_k = F_k s_{k-1} + G_k \sigma_e^2 \quad (9)$$

Где

$$s_k = [x_k, v_k^x, a_k^x, e_k^x, y_k, v_k^y, a_k^y, e_k^y] \quad (10)$$

Временной интервал *dt* составляет 100 мс

$$F = \begin{bmatrix} 1 & dt & \frac{dt^2}{2} & \frac{dt^3}{6} & 0 & 0 & 0 & 0 \\ 0 & 1 & dt & \frac{dt^2}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & dt & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & dt & \frac{dt^2}{2} & \frac{dt^3}{6} \\ 0 & 0 & 0 & 0 & 0 & 1 & dt & \frac{dt^2}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & dt \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (11)$$

$$Q = G_k \sigma_e^2 \quad (12)$$

$$G = g\,g^T = \begin{bmatrix} \frac{\partial x}{\partial e} & \frac{\partial v}{\partial e} & \frac{\partial a}{\partial e} & \frac{\partial e}{\partial e} \end{bmatrix} \begin{bmatrix} \frac{\partial x}{\partial e} & \frac{\partial v}{\partial e} & \frac{\partial a}{\partial e} & \frac{\partial e}{\partial e} \end{bmatrix}^T =$$

$$= \begin{bmatrix} \frac{dt^3}{6} & \frac{dt^2}{2} & dt & 1 \end{bmatrix} \begin{bmatrix} \frac{dt^3}{6} & \frac{dt^2}{2} & dt & 1 \end{bmatrix}^T =$$

$$G = \begin{bmatrix} \frac{dt^6}{36} & \frac{dt^5}{12} & \frac{dt^4}{6} & \frac{dt^3}{6} & 0 & 0 & 0 & 0 \\ \frac{dt^5}{12} & \frac{dt^4}{4} & \frac{dt^3}{2} & \frac{dt^2}{2} & 0 & 0 & 0 & 0 \\ \frac{dt^4}{6} & \frac{dt^3}{2} & dt^2 & dt & 0 & 0 & 0 & 0 \\ \frac{dt^3}{6} & \frac{dt^2}{2} & dt & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{dt^6}{36} & \frac{dt^5}{12} & \frac{dt^4}{6} & \frac{dt^3}{6} \\ 0 & 0 & 0 & 0 & \frac{dt^5}{12} & \frac{dt^4}{4} & \frac{dt^3}{2} & \frac{dt^2}{2} \\ 0 & 0 & 0 & 0 & \frac{dt^4}{6} & \frac{dt^3}{2} & dt^2 & dt \\ 0 & 0 & 0 & 0 & \frac{dt^3}{6} & \frac{dt^2}{2} & dt & 1 \end{bmatrix} \quad (13)$$

Вектор измерения представляется следующим образом

$$z_k = \begin{bmatrix} x_k & y_k \end{bmatrix} \quad (14)$$

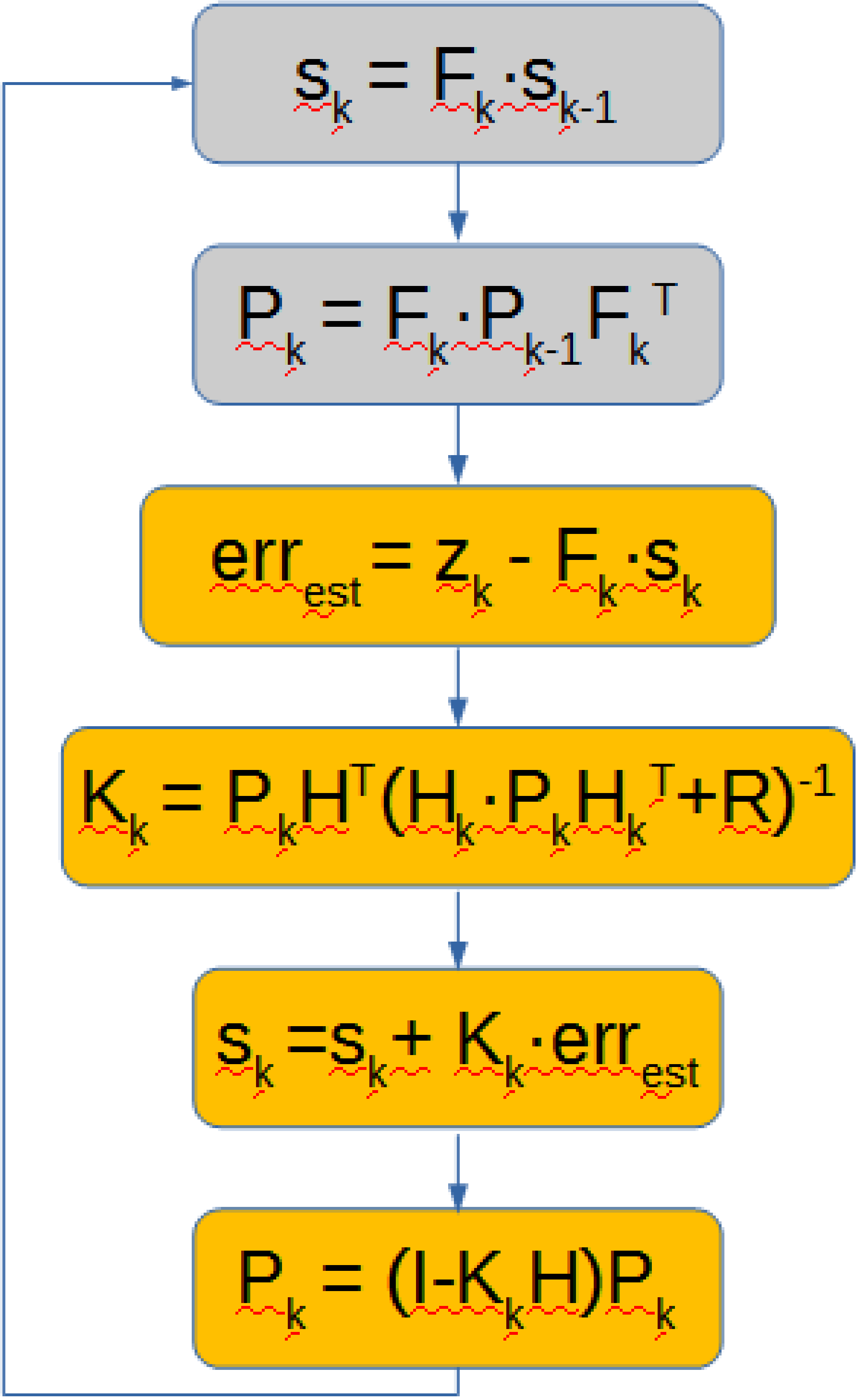
Тогда матрица измерений

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad (15)$$

Ковариационная матрица погрешности измерений R задается так

$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \sigma_m^2 \quad (16)$$

Блок схема базового фильтра Калмана

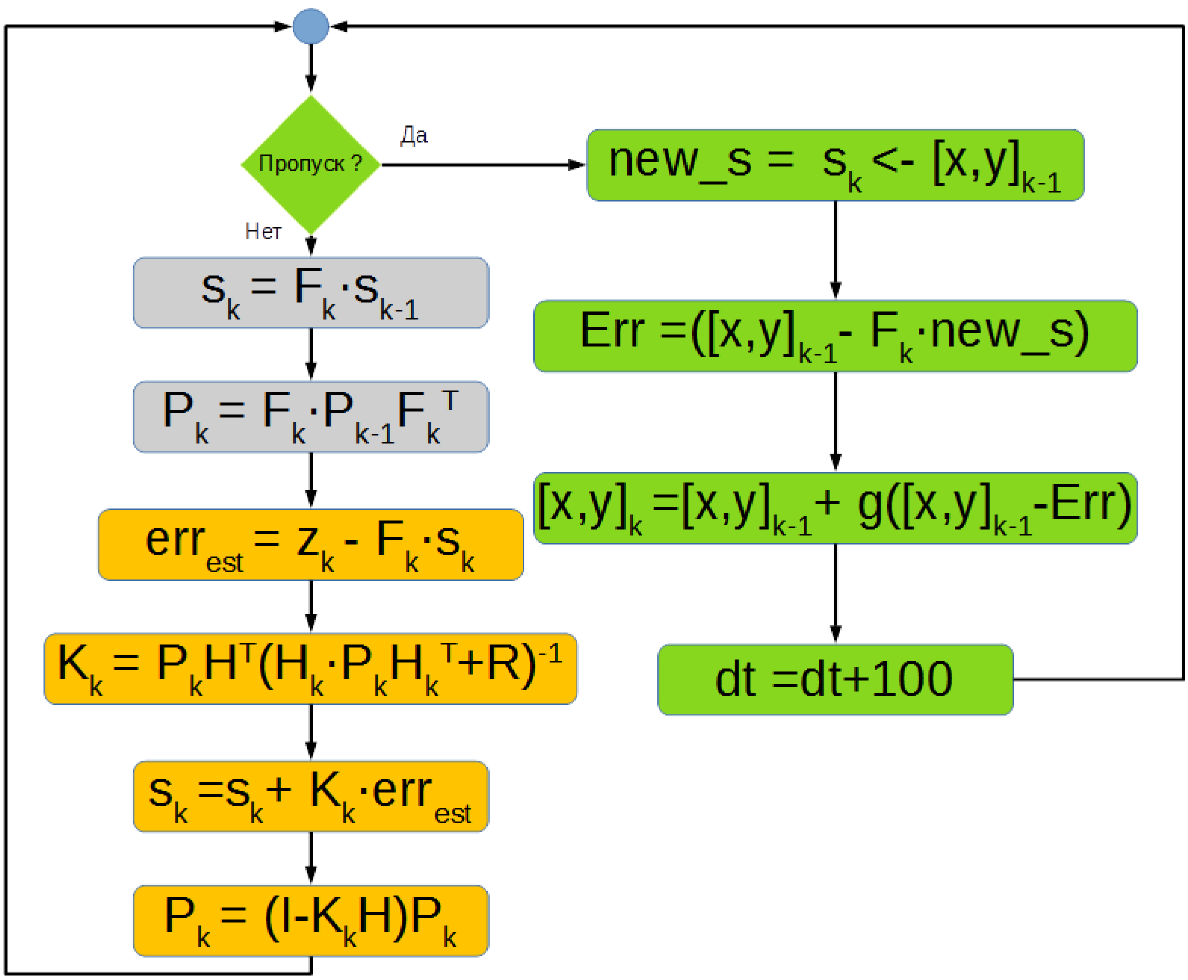


Восстановление пропусков

Проблема заключалась в том, что часть триангулированных точек теряются перед обработкой. Когда эти потери приходятся на линейный участок алгоритм предсказания их восстанавливает. Когда потерянные точки приходятся на криволинейный участок траектории возникают значительные ошибки. Для уменьшения влияния для восстановления применил g-h фильтр. Его основная идея может быть записана математически так

$$news_k = news_{k-1} + g \cdot (news_{k-1} - F_k \cdot news_{k-1}) \quad (17)$$

В этом случае блок схема алгоритма выглядит так



Настройка фильтра

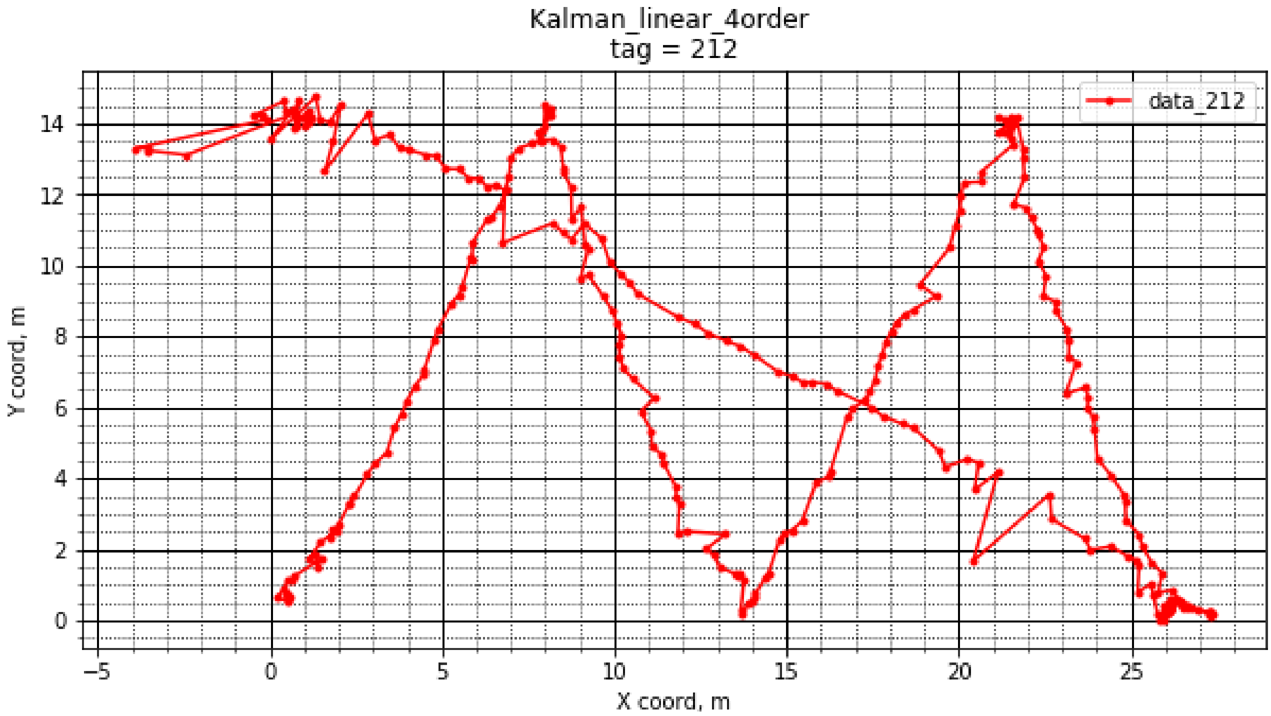
При настройке необходимо определить следующие параметры

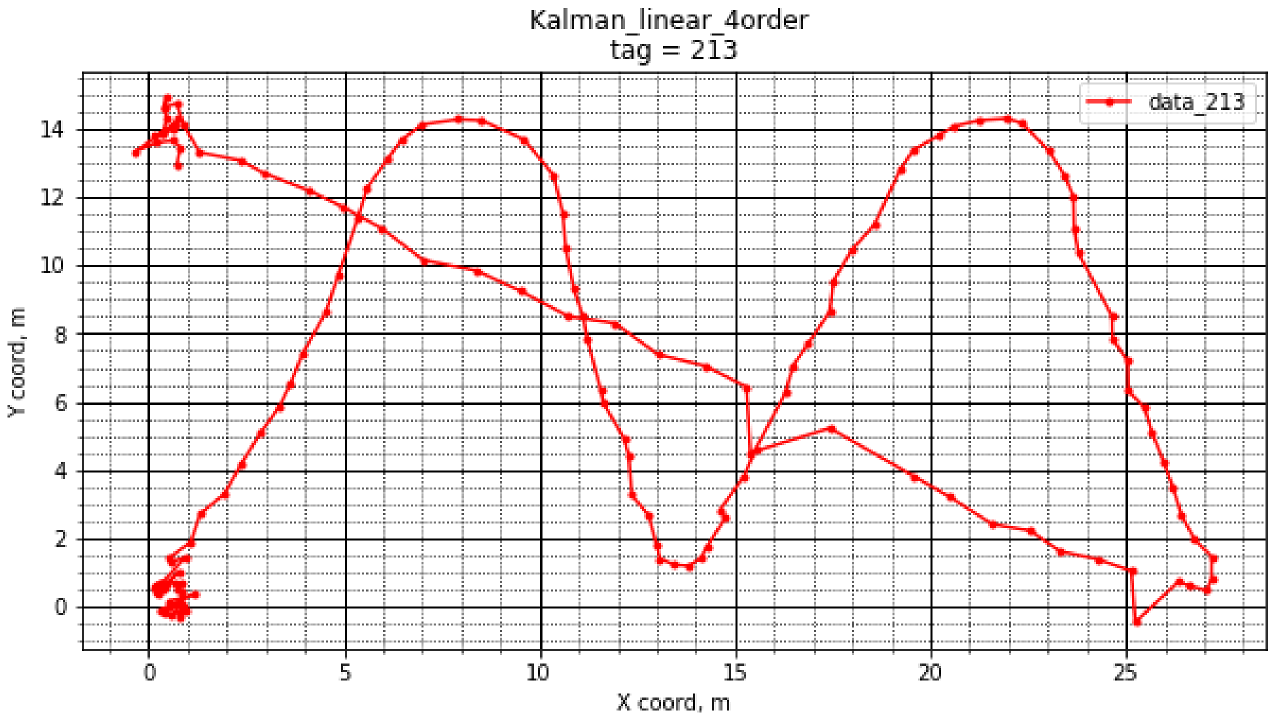
Известен интервал времени прихода триангулированных данных — 100 мс, следовательно dt = 100

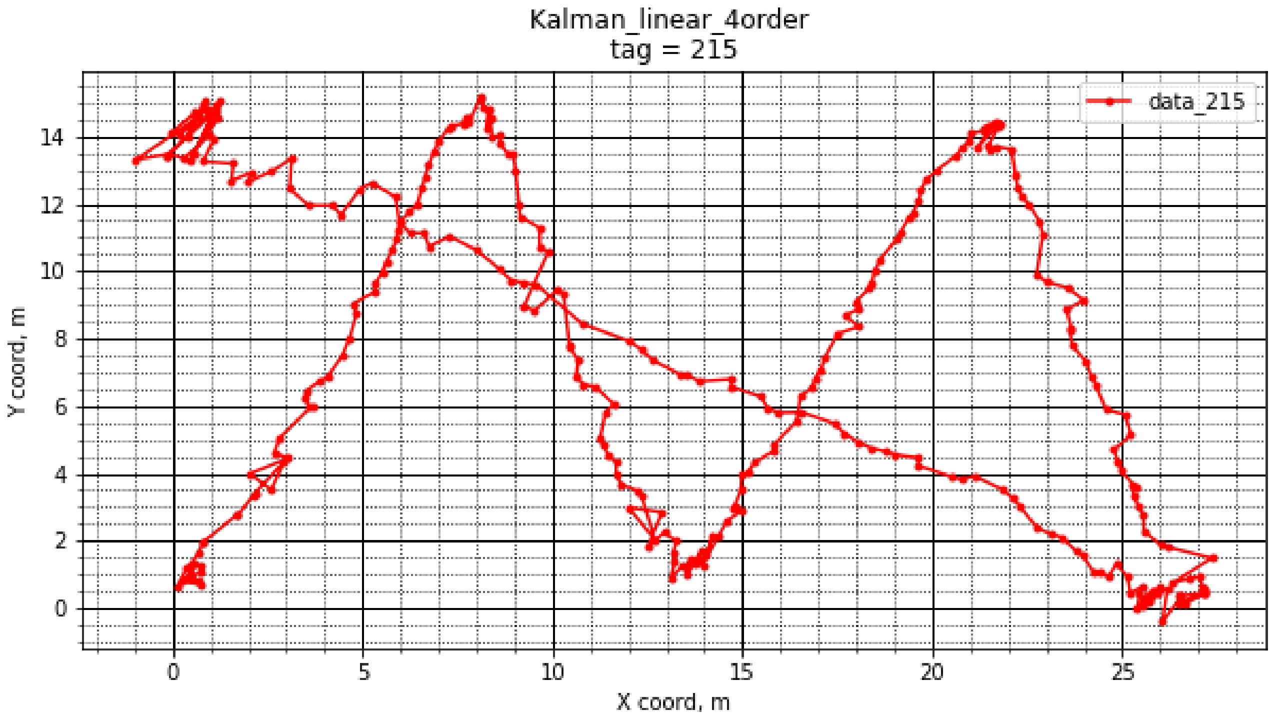
Задаем СКО измерения координаты σ_m = 3 см ($3\sigma_m$ = 9 см)

Осталось определить еще два параметра — σ_e и параметр g

Настройку производил по результатам эксперимента «бабочка» трех меток. Ниже графики каждой из меток до фильтрации.



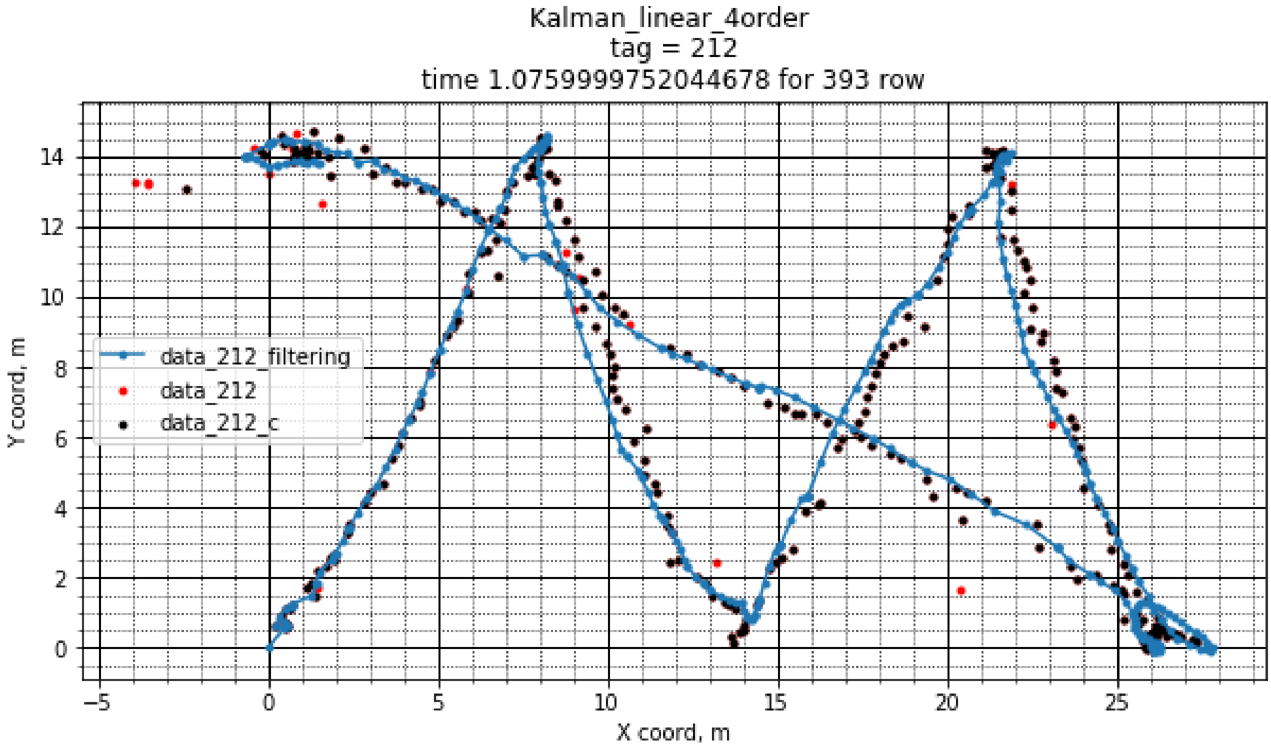


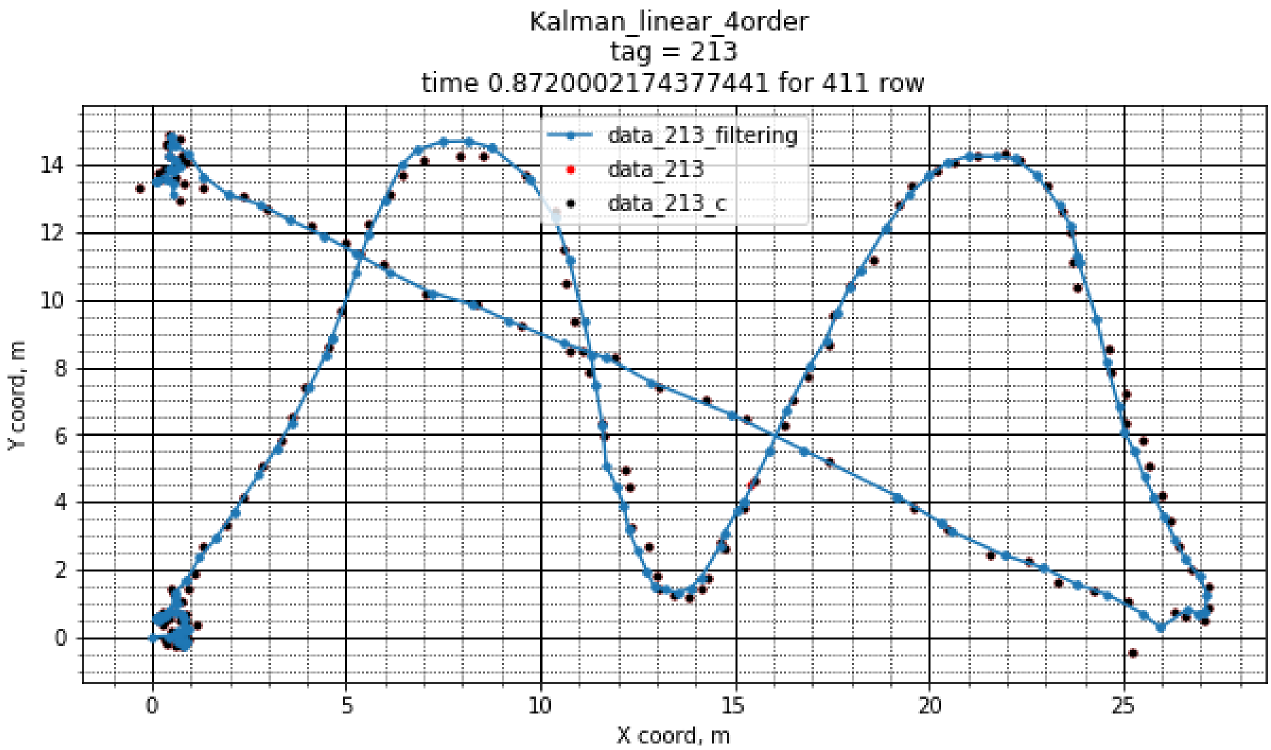

Kalman_linear_4order
tag = 215
data_215
Y coord, m
X coord, m
14
12
10
8
6
4
2
0
0
5
10
15
20
25

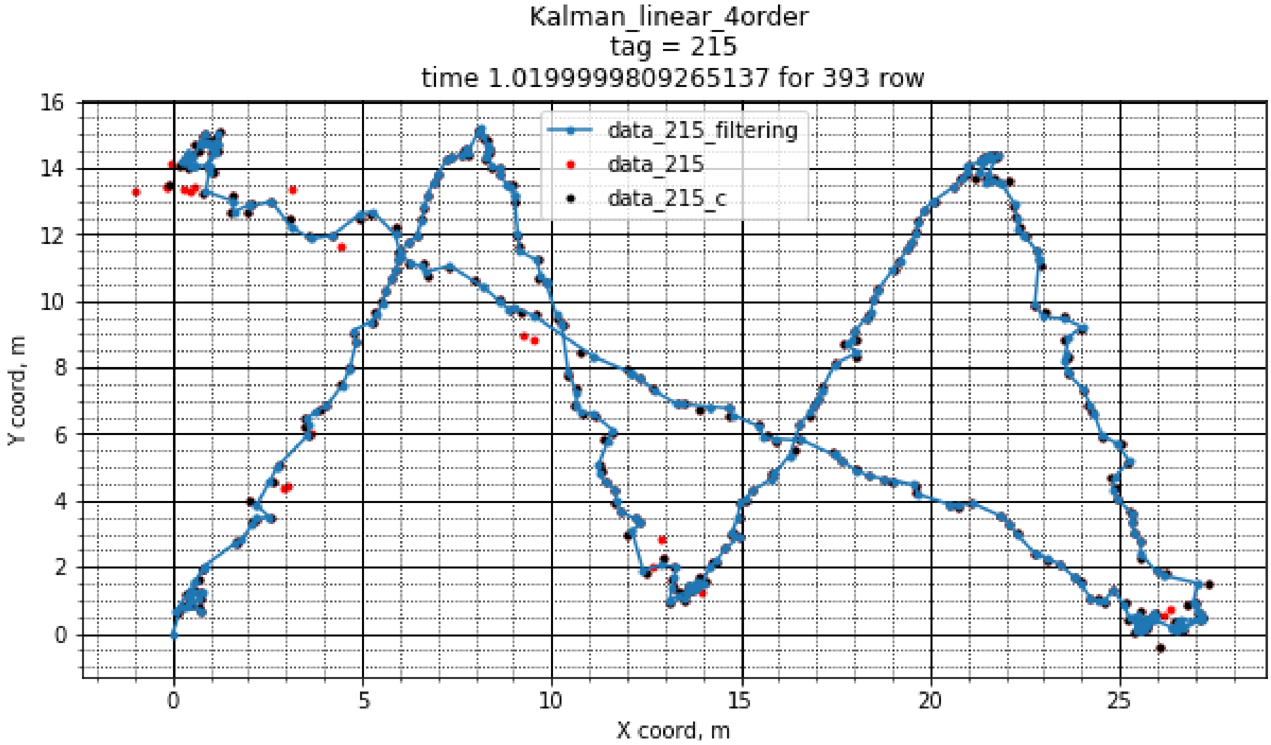
Подбор параметра σ_e

Подбор параметра осуществлялся эмпирически. Здесь следует отметить, что увеличение параметра приводит к ухудшению сглаживающих свойств фильтра, тогда как уменьшение к значительным потерям динамики фильтруемых сигналов. Ниже приведены результаты фильтрации заниженным параметром σ_e

Занижение параметра

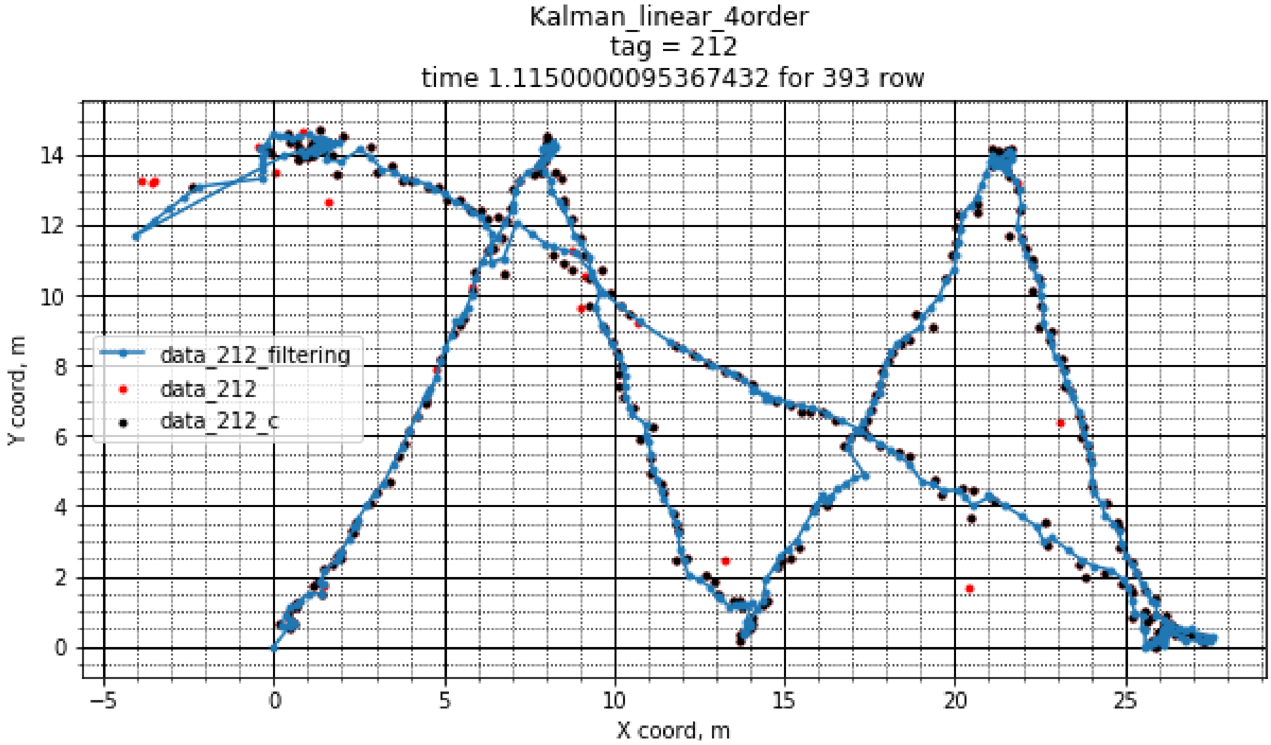


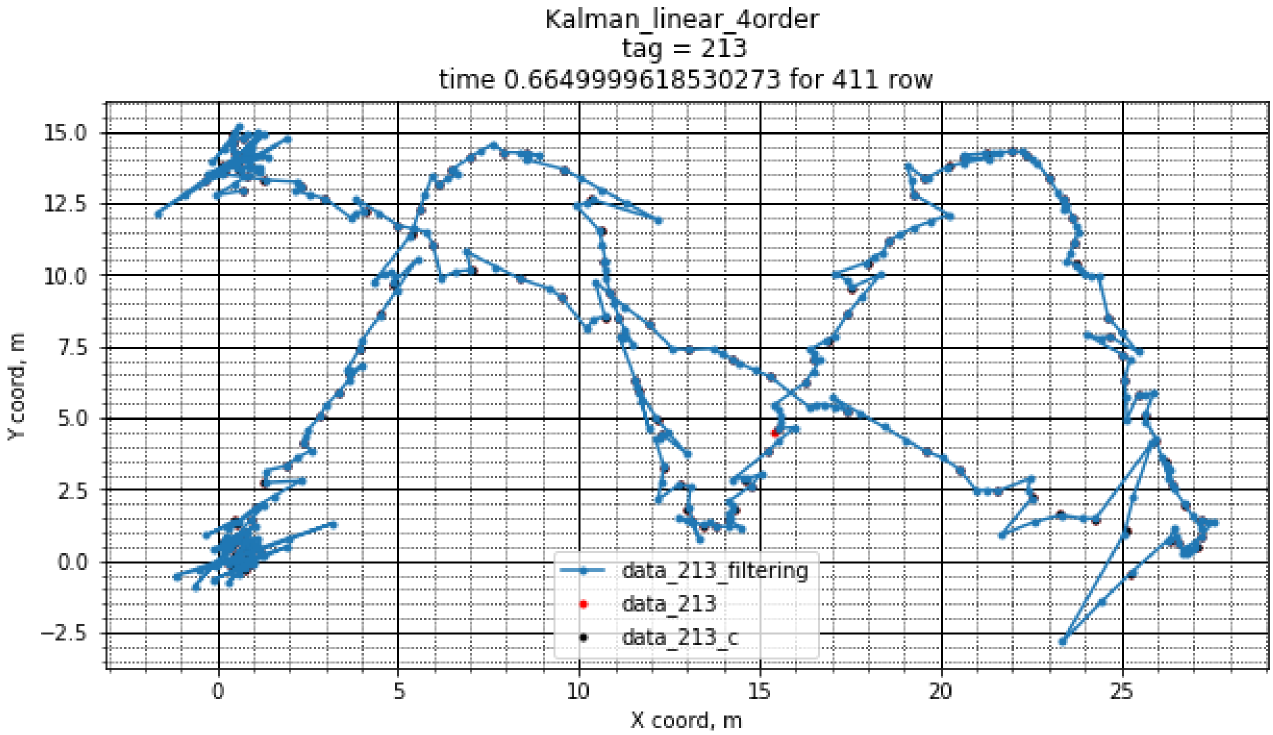


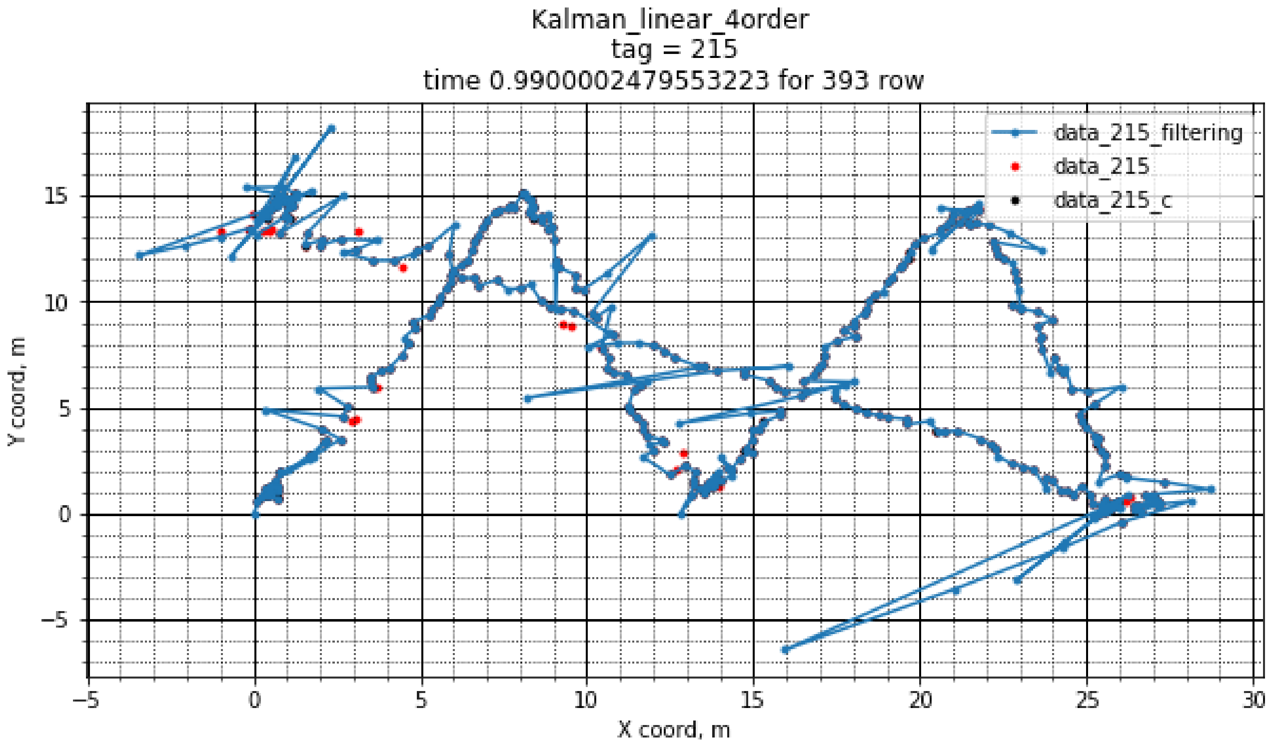

Kalman_linear_4order
tag = 215
time 1.0199999809265137 for 393 row
data_215_filtering
data_215
data_215_c
Y coord, m
X coord, m
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25

Подбор параметра g

Без использования предложенного алгоритма восстановления базовый алгоритм способен восстановить пропуски, но при этом возникает накопление ошибки. Результат такого восстановления приведен ниже.





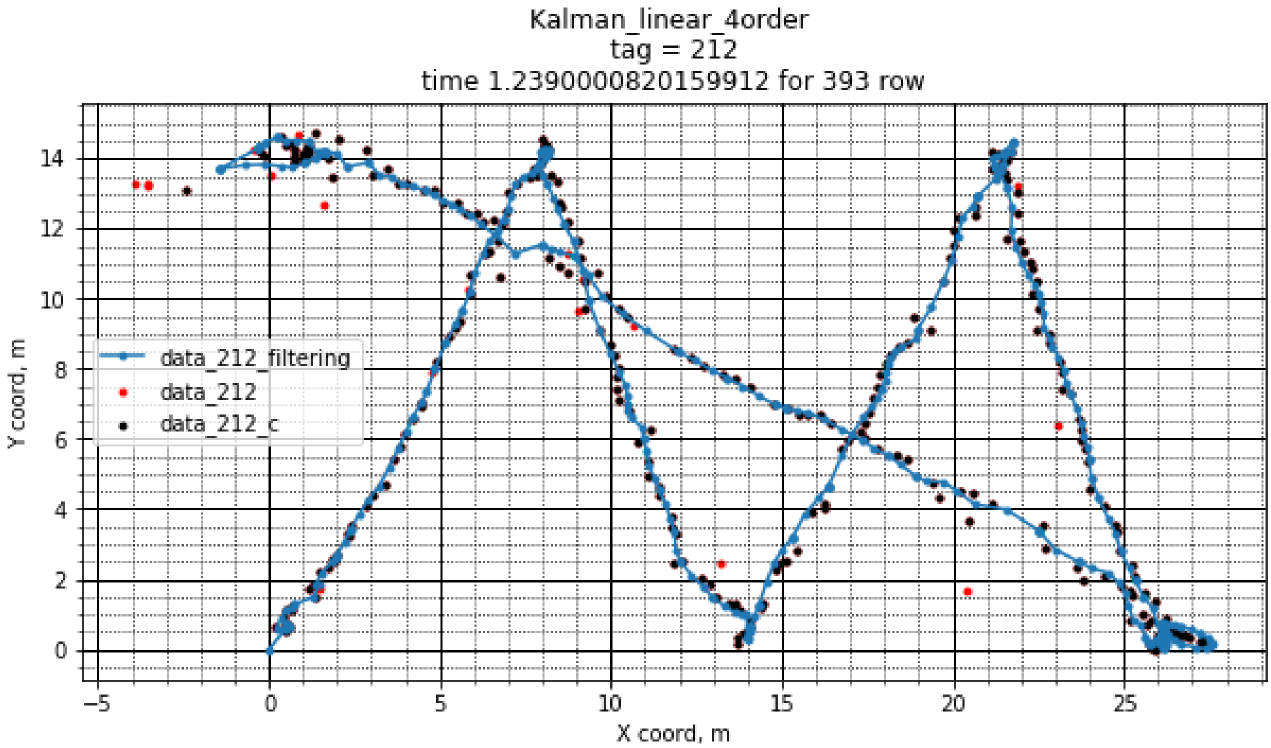


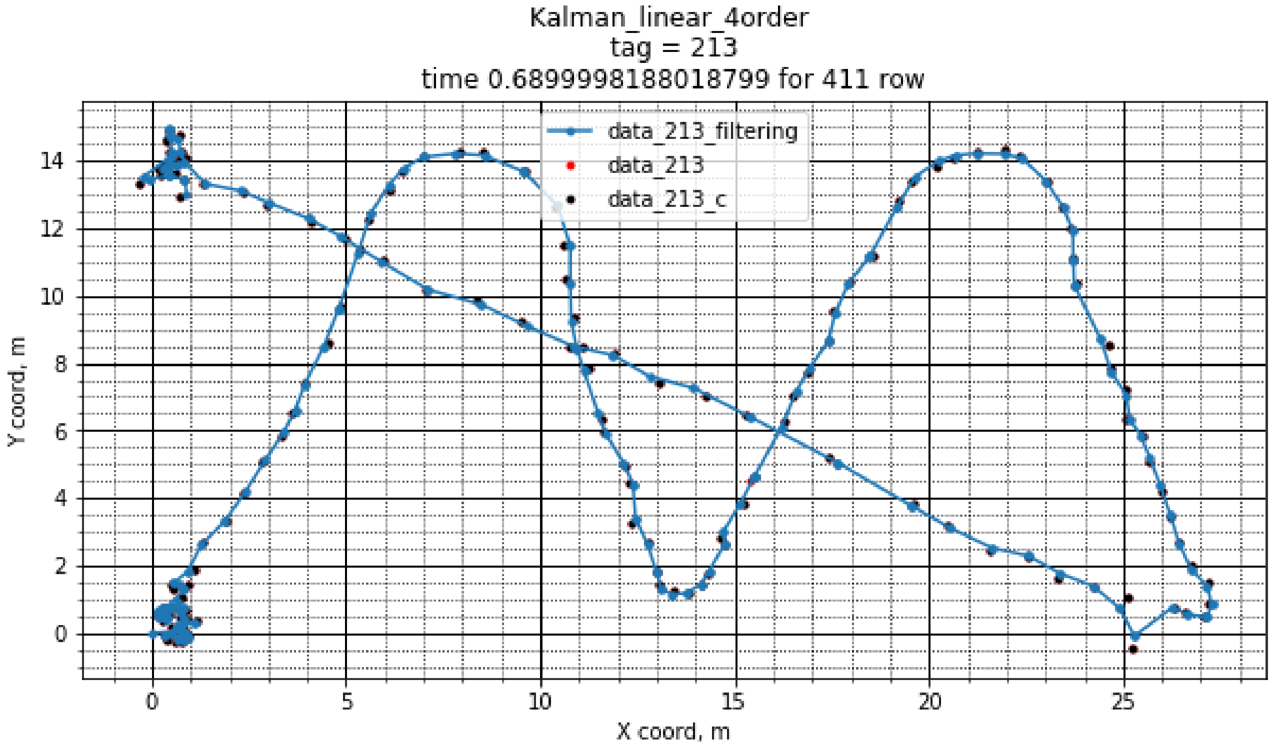
Для корректной работы алгоритма необходимо подобрать параметр g.

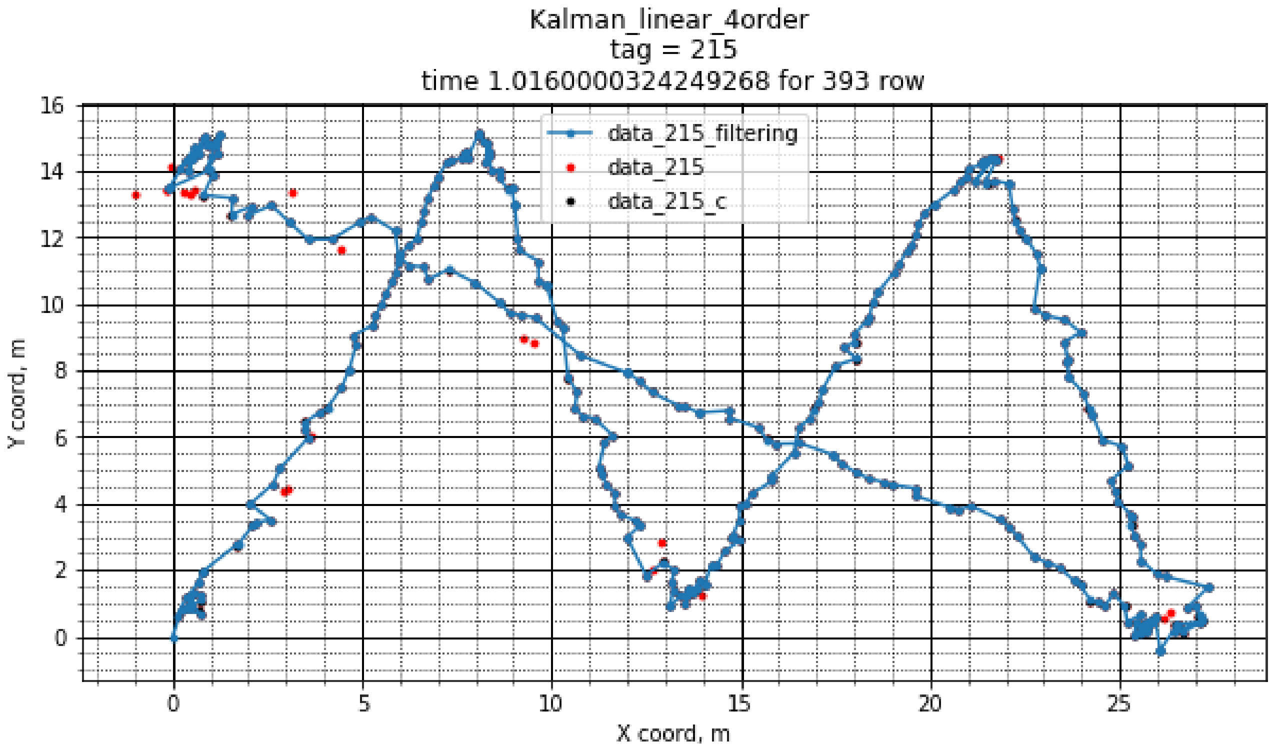
Подбор осуществлялся эмпирически. Здесь следует отметить, что увеличение параметра практически не влияет на линейные участки траектории, но вносит сильные искажения на поворотах. Снижение параметра приводит к тому, что фильтр перестает восстанавливать потерянные точки.

Ниже приведены графики с завышенным и заниженным параметром g

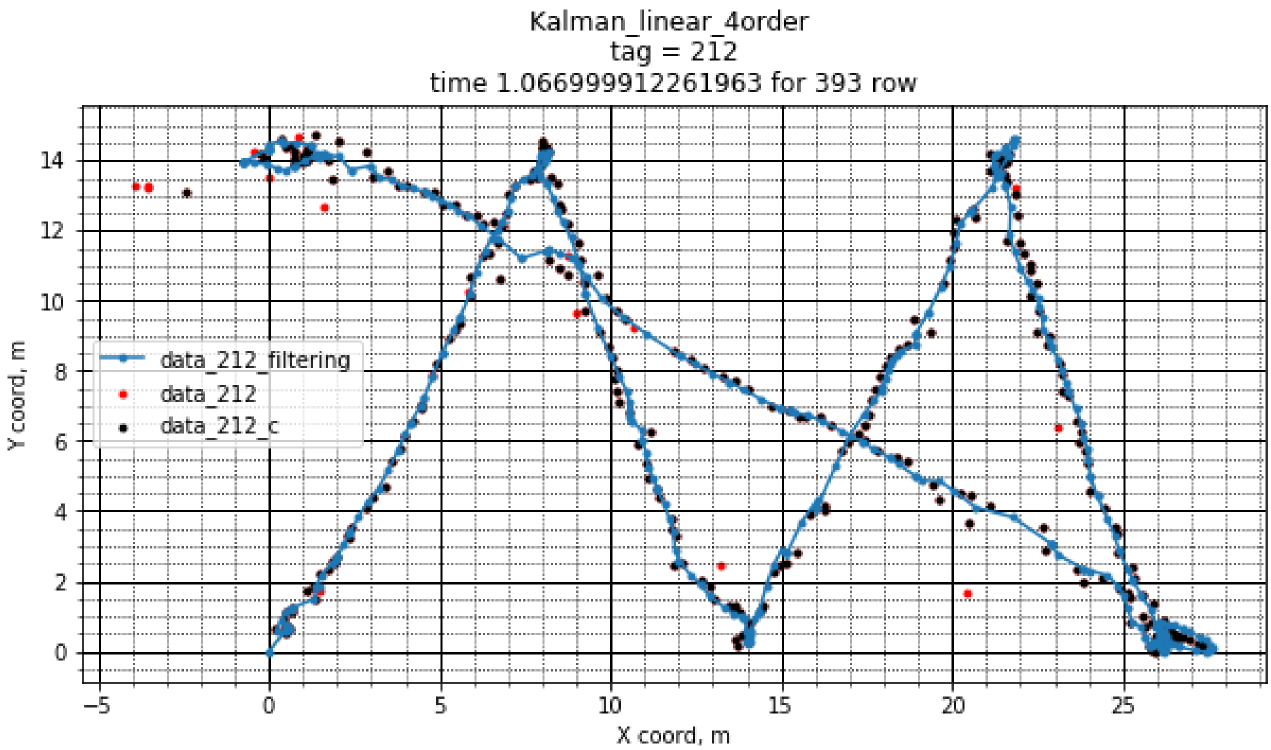
Заниженный параметр g

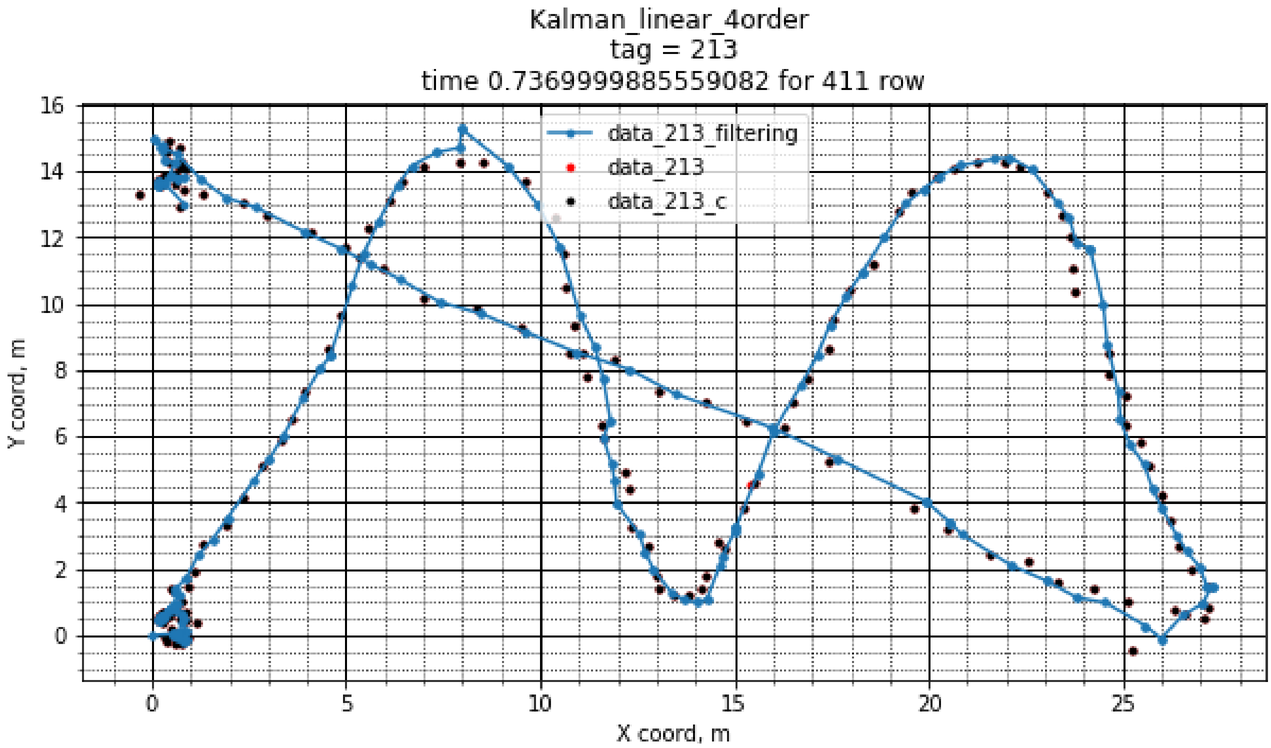

Kalman_linear_4order
tag = 212
time 1.2390000820159912 for 393 row
data_212_filtering
data_212
data_212_c
Y coord, m
X coord, m
14
12
10
8
6
4
2
0
−5
0
5
10
15
20
25

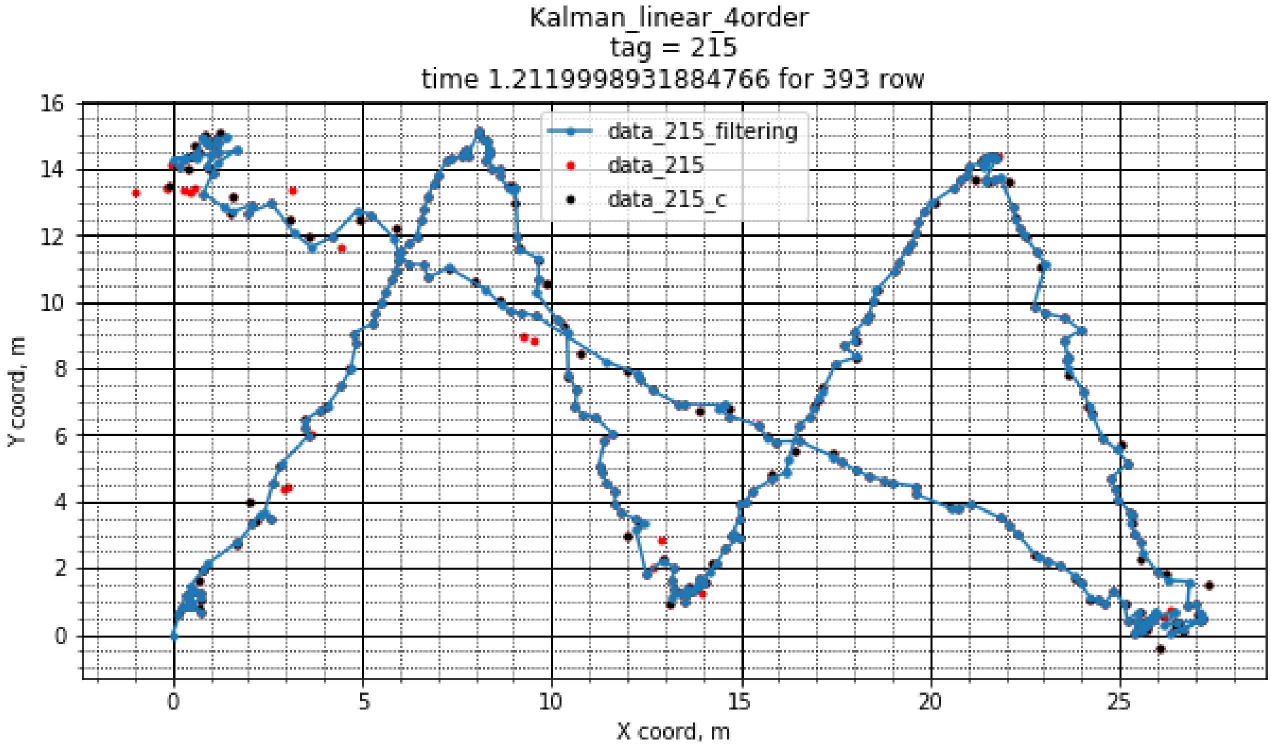

Kalman_linear_4order
tag = 213
time 0.6899998188018799 for 411 row
data_213_filtering
data_213
data_213_c
Y coord, m
X coord, m
14
12
10
8
6
4
2
0
0
5
10
15
20
25



Завышенный параметр g



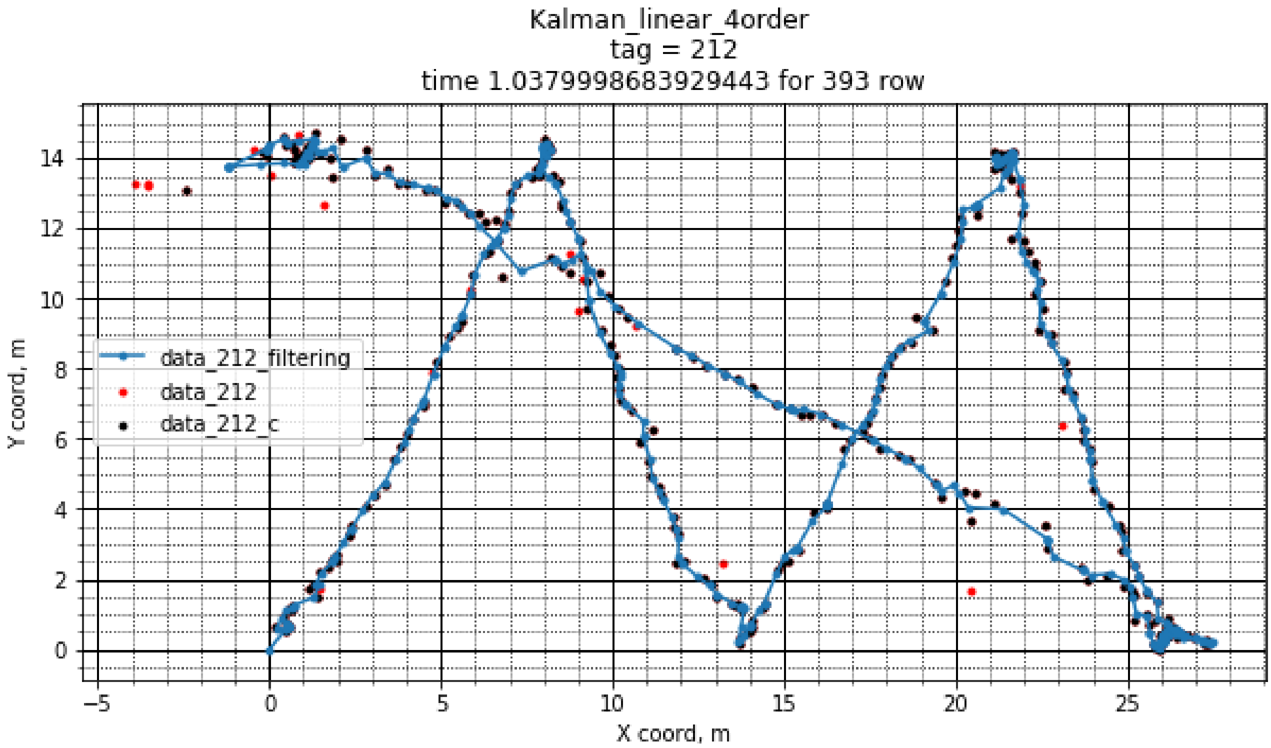


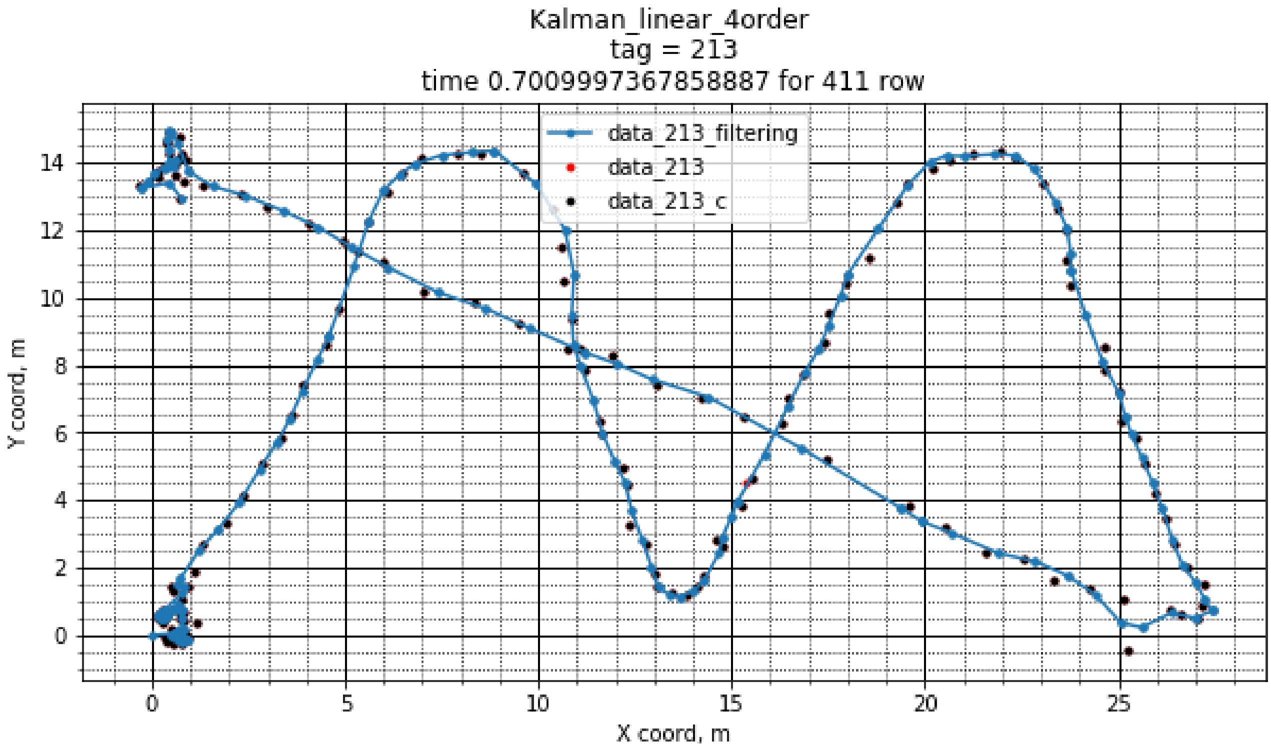


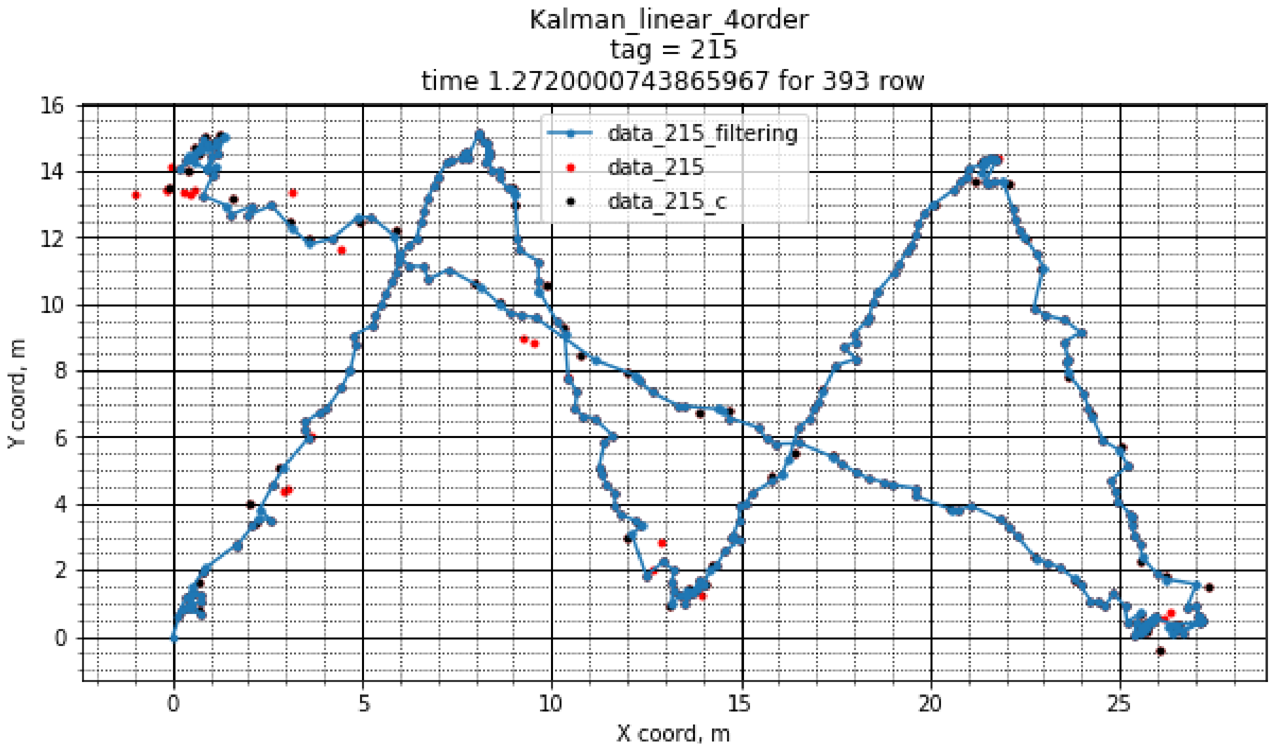
Настройки при которых фильтр ведет себя корректно такие:

$$\sigma_e^2 = 4 \cdot 10^{-21} \quad g = 0.2 \tag{18}$$

Результаты фильтрации


Kalman_linear_4order
tag = 212
time 1.0379998683929443 for 393 row
data_212_filtering
data_212
data_212_c
Y coord, m
X coord, m


Kalman_linear_4order
tag = 213
time 0.7009997367858887 for 411 row
data_213_filtering
data_213
data_213_c
Y coord, m
X coord, m


Kalman_linear_4order
tag = 215
time 1.2720000743865967 for 393 row
data_215_filtering
data_215
data_215_c
Y coord, m
X coord, m
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25

Библиография

1: Листеренко Р. Р., Применение фильтра Калмана для обработки последовательности GPS-координат, 2015
2: Gabriel A. Terejanu, Unscented Kalman Filter Tutorial,
3: Greg Welch, Gary Bishop, An Introduction to the Kalman Filter, 2001
4: Maria Isabel Ribeiro, Kalman and Extended Kalman Filters:Concept, Derivation and Properties, 2004
2: Roger R Labbe Jr, Kalman and Bayesian Filters in Python,

Приложение А Код фильтра реализованный на языке Python

Базовый класс

```
import numpy as np
import pandas as pd
from abc import ABC, abstractmethod

class KalmanFilter_EKF:
def __init__(self, dim_x, dim_z, dim_u=0):
 """
 Create a Kalman filter. You are responsible for setting the
 various state variables to reasonable values; the defaults below will
 not give you a functional filter.
 Parameters
 ----------
 dim_x : int
 Number of state variables for the Kalman filter. For example, if
 you are tracking the position and velocity of an object in two
 dimensions, dim_x would be 4.
 This is used to set the default size of P, Q, and u
 dim_z : int
 Number of of measurement inputs. For example, if the sensor
 provides you with position in (x,y), dim_z would be 2.
 dim_u : int (optional)
 size of the control input, if it is being used.
 Default value of 0 indicates it is not used.
 """

 self.x = np.zeros((dim_x, 1))  # state
 self.P = np.eye(dim_x)  # uncertainty covariance
 self.Q = np.eye(dim_x)  # process uncertainty
 self.u = np.zeros((dim_x, 1))  # motion vector
 self.B = 0  # control transition matrix
 self.JF = 0  # state F-jacobian matrix
 self.H = 0  # Measurement function
 self.R = np.eye(dim_z)  # state uncertainty

 # identity matrix. Do not alter this.
 self._I = np.eye(dim_x)

def predict(self, u=0):
 """
 Predict next position.
 Parameters
 ----------
 u : np.array
 Optional control vector. If non-zero, it is multiplied by B
 to create the control input into the system.
```

```
    """

    self.nonlinear_state()
    # self.x = np.dot(self.JF, self.x) + np.dot(self.B, u)
    self.P = self.JF.dot(self.P).dot(self.JF.T) + self.Q

def update(self, Z, R=None):
    """
    Add a new measurement (Z) to the kalman filter. If Z is None, nothing
    is changed.
    Parameters
    ---------
    Z : np.array
    measurement for this update.
    R : np.array, scalar, or None
    Optionally provide R to override the measurement noise for this
    one call, otherwise self.R will be used.
    """

    if Z is None:
        return

    if R is None:
        R = self.R
    elif np.isscalar(R):
        R = np.eye(self.dim_z) * R

    # error (residual) between measurement and prediction
    y = Z - np.dot(self.H, self.x)

    # project system uncertainty into measurement space
    S = np.dot(self.H, self.P).dot(self.H.T) + R

    # map system uncertainty into kalman gain
    K = np.dot(self.P, self.H.T).dot(np.linalg.inv(S))

    # predict new x with residual scaled by the kalman gain
    self.x = self.x + np.dot(K, y)

    I_KH = self._I - np.dot(K, self.H)
    self.P = np.dot(I_KH, self.P).dot(I_KH.T) + \
            np.dot(K, R).dot(K.T)

def filtering(self, Z):
    """
    This fuction filtering input data Z
    :param Z: pandas DataFrame input data
    :return: pandas dataFrame filtering data
    """
    coord_x_est = [0]
    coord_y_est = [0]
    time_est = [0]
```

```
    for i in range(1, len(Z.x)):
        self.recover(Z.iloc[i-1:i+1].reset_index(drop = True), coord_x_est, coord_y_est,
time_est)
        self.predict()
        mes = Z.loc[i].to_numpy()
        self.update(np.resize(mes,  (3, 1)))
        # save for latter plotting
        coord_x_est.append(self.x[2])
        coord_y_est.append(self.x[3])
        time_est.append(self.x[0])
        return pd.DataFrame({'X_f': coord_x_est,
                   'Y_f': coord_y_est,
                   'time': time_est
                   })

    @abstractmethod
    def nonlinear_state(self):
     pass

    @abstractmethod
    def predict(self, u=0):
     """
     Predict next position.
     Parameters
     ----------
     u : np.array
     Optional control vector. If non-zero, it is multiplied by B
     to create the control input into the system.
     """
     pass

    @abstractmethod
    def jacobianF(self):
     """
     Function return jacobian for k-step iteration
     """
     pass

    @abstractmethod
    def recover(self, Z, x, y, time):
    pass
```

Производный класс фильтра 4 порядка

```
import numpy as np
import pandas as pd
from . import KalmanFilter_EKF

class LinearKalman4(KalmanFilter_EKF):
def __init__(self):
 super().__init__(dim_x=8, dim_z=2)
 dt = 100
 self.x = np.array([[0], [0], [0], [0], [0], [0], [0], [0]])
 self.F = np.array([[1, dt, 1/2*(dt**2), 1/6*(dt**3), 0, 0, 0, 0],
                    [0, 1, dt, 1/2*(dt**2), 0, 0, 0, 0],
                    [0, 0, 1, dt, 0, 0, 0, 0],
                    [0, 0, 0, 1, 0, 0, 0, 0],
                    [0, 0, 0, 0, 1, dt, 1/2*(dt**2), 1/6*(dt**3)],
                    [0, 0, 0, 0, 0, 1, dt, 1/2*(dt**2)],
                    [0, 0, 0, 0, 0, 0, 1, dt],
                    [0, 0, 0, 0, 0, 0, 0, 1]]
                   )
 self.H = np.array([[1, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 1, 0, 0, 0]]
                   )
 self.R *= 9e-4

 self.Q = np.array([[1/36*(dt**6), 1/12*(dt**5), 1/6*(dt**4),   1/6*(dt**4),  0, 0, 0, 0],
                    [1/12*(dt**5), 1/4*(dt**4),  1/2*(dt**3),  1/2*(dt**2),  0, 0, 0, 0],
                    [1/6*(dt**4),   1/2*(dt**3),  (dt**2),      (dt**1),     0, 0, 0, 0],
                    [1/6*(dt**3),   1/2*(dt**2),  (dt**1),      1,           0, 0, 0, 0],
                    [0, 0, 0, 0,   1/36*(dt**6), 1/12*(dt**5), 1/6*(dt**4),  1/6*(dt**4)],
                    [0, 0, 0, 0,   1/12*(dt**5), 1/4*(dt**4),  1/2*(dt**3), 1/2*(dt**2)],
                    [0, 0, 0, 0,   1/6*(dt**4),   1/2*(dt**3),  (dt**2),     (dt**1)],
                    [0, 0, 0, 0,   1/6*(dt**3),   1/2*(dt**2),  (dt**1),     1]]
 )
 sygma = 1e-18
 self.Q *= sygma
 self.P *= 10.

def predict(self, u=0):
 """
 Predict next position.
 Parameters
 ----------
 u : np.array
 Optional control vector. If non-zero, it is multiplied by B
 to create the control input into the system.
 """

 self.x = np.dot(self.F, self.x) + np.dot(self.B, u)
 self.P = self.F.dot(self.P).dot(self.F.T) + self.Q
```

```
    def filtering(self, Z):
        """
        This fuction filtering input data Z
        :param Z: pandas DataFrame input data
        :return: pandas dataFrame filtering data
        """
        coord_x_est = [0]
        coord_y_est = [0]
        time_est = [Z[['time']].iloc[0][0]]
        for i in range(1, len(Z.x)):

            while (Z[['time']].iloc[i][0] - time_est[-1] > 150):
                self.x[0] = self.x[0] + 0.12 * (self.x[0] - np.dot(self.F, self.x)[0])
                self.x[4] = self.x[4] + 0.12 * (self.x[4] - np.dot(self.F, self.x)[4])
                coord_x_est.append(self.x[0])
                coord_y_est.append(self.x[4])
                time_est.append(time_est[-1]+100)
                # self.predict()
            self.predict()
            mes = Z[['x', 'y']].iloc[i].to_numpy()
            self.update(np.resize(mes,  (2, 1)))
            # save for latter plotting
            coord_x_est.append(self.x[0])
            coord_y_est.append(self.x[4])
            time_est.append(int(Z[['time']].iloc[i]))

        return pd.DataFrame({'X_f': coord_x_est,
                             'Y_f': coord_y_est,
                             'time': time_est
                             })

    def clear(self):
        dt = 100
        self.x = np.array([[0], [0], [0], [0], [0], [0], [0], [0]])
        self.F = np.array([[1, dt, 1 / 2 * (dt ** 2), 1 / 6 * (dt ** 3), 0, 0, 0, 0],
                           [0, 1, dt, 1 / 2 * (dt ** 2), 0, 0, 0, 0],
                           [0, 0, 1, dt, 0, 0, 0, 0],
                           [0, 0, 0, 1, 0, 0, 0, 0],
                           [0, 0, 0, 0, 1, dt, 1 / 2 * (dt ** 2), 1 / 6 * (dt ** 3)],
                           [0, 0, 0, 0, 0, 1, dt, 1 / 2 * (dt ** 2)],
                           [0, 0, 0, 0, 0, 0, 1, dt],
                           [0, 0, 0, 0, 0, 0, 0, 1]]
                          )
        self.H = np.array([[1, 0, 0, 0, 0, 0, 0, 0],
                           [0, 0, 0, 0, 1, 0, 0, 0]]
                          )
        self.R *= 9e-4

        self.Q = np.array([[1 / 36 * (dt ** 6), 1 / 12 * (dt ** 5), 1 / 6*(dt ** 4), 1 / 6*(dt ** 4), 0, 0,
0, 0],
                           [1 / 12 * (dt ** 5), 1 / 4 * (dt ** 4), 1 / 2 * (dt ** 3), 1 / 2*(dt ** 2), 0, 0, 0, 0],
                           [1 / 6*(dt ** 4), 1 / 2 * (dt ** 3), (dt ** 2), (dt ** 1), 0, 0, 0, 0],
```

```
                    [1 / 6*(dt ** 3), 1 / 2 * (dt ** 2), (dt ** 1), 1, 0, 0, 0, 0],
                    [0, 0, 0, 0, 1 / 36 * (dt ** 6), 1 / 12 * (dt ** 5), 1 / 6*(dt ** 4), 1 / 6*(dt ** 4)],
                    [0, 0, 0, 0, 1 / 12 * (dt ** 5), 1 / 4 * (dt ** 4), 1 / 2 * (dt ** 3), 1 / 2*(dt ** 2)],
                    [0, 0, 0, 0, 1 / 6*(dt ** 4), 1 / 2 * (dt ** 3), (dt ** 2), (dt ** 1)],
                    [0, 0, 0, 0, 1 / 6*(dt ** 3), 1 / 2 * (dt ** 2), (dt ** 1), 1]]
                   )

        sygma = 1e-18
        self.Q *= sygma
        self.P *= 10.

    def nonlinear_state(self):
        """
        """
        pass

    def jacobianF(self):
        """
        """
        pass

    def recover(self, Z, x, y, time):
        pass
```