Московский государственный университет
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Отчёт по статье «Passive Underwater Target Tracking: Conditionally Minimax Nonlinear Filtering with Bearing-Doppler Observations»

Экзаменационное задание по курсу «Дополнительные главы случайных процессов»

Выполнил: студент 416 группы
Шурыгин Всеволод Евгеньевич

Москва
2024

Содержание

1 Постановка задачи

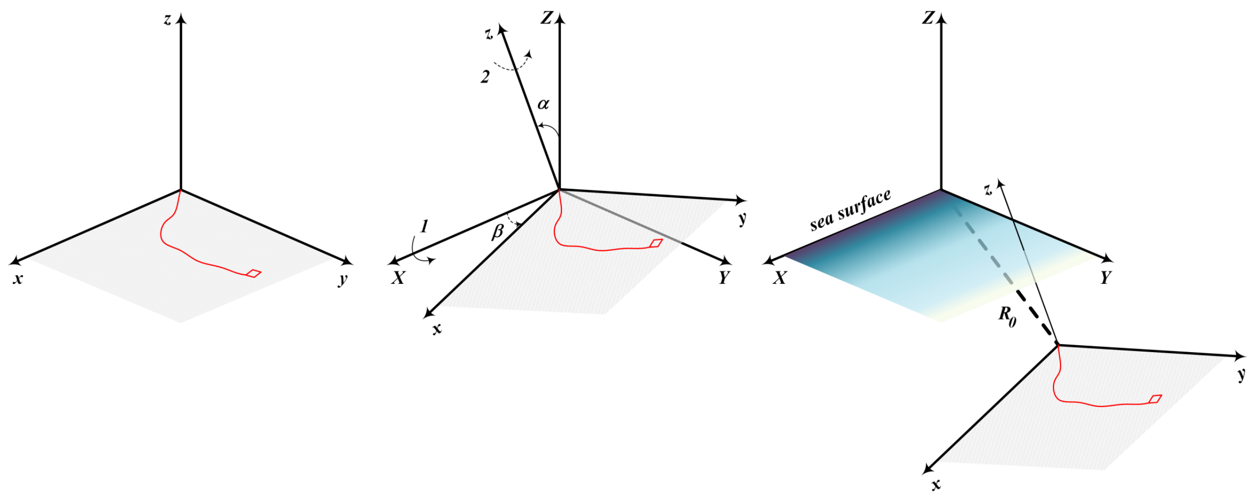
1.1 Описание модели

Решается задача о слежении за маневрированием подводного аппарата. Движение аппарата описывается следующей системой:

$$\begin{cases} dX_t = dx_t = v_t \cos\varphi_t dt \\ dY_t = dy_t = v_t \sin\varphi_t dt \\ dZ_t = dz_t = 0 \\ dv_t = 0 \\ d\varphi_t = \frac{a_t^n}{v_t} dt \\ da_t^n = (-\lambda a_t^n + \nu)\, dt + \mu dW_t \end{cases}$$

x_t, y_t, z_t — координаты подводного аппарата в его системе отсчета, X_t, Y_t, Z_t — координаты системы отсчёта в системе наблюдателя. Для простоты возьмём единичную матрицу поворота, а сдвиг пусть отвечает начальным условиям на x_t, y_t, z_t.

эти константы приведёны в разделе 2.4



1.2 Система наблюдения

Каждый из датчиков наблюдает следующие величины:

$$\xi^i_{t_k} = \frac{Z_{t_k} - Z^i}{R^i_{t_k}} + v^{\xi^i}_k \tag{1.1}$$

$$\eta^i_{t_k} = \frac{X_{t_k} - X^i}{r^i_{t_k}} + v^{\eta^i}_k \tag{1.2}$$

где $i = 1, .., N$ и

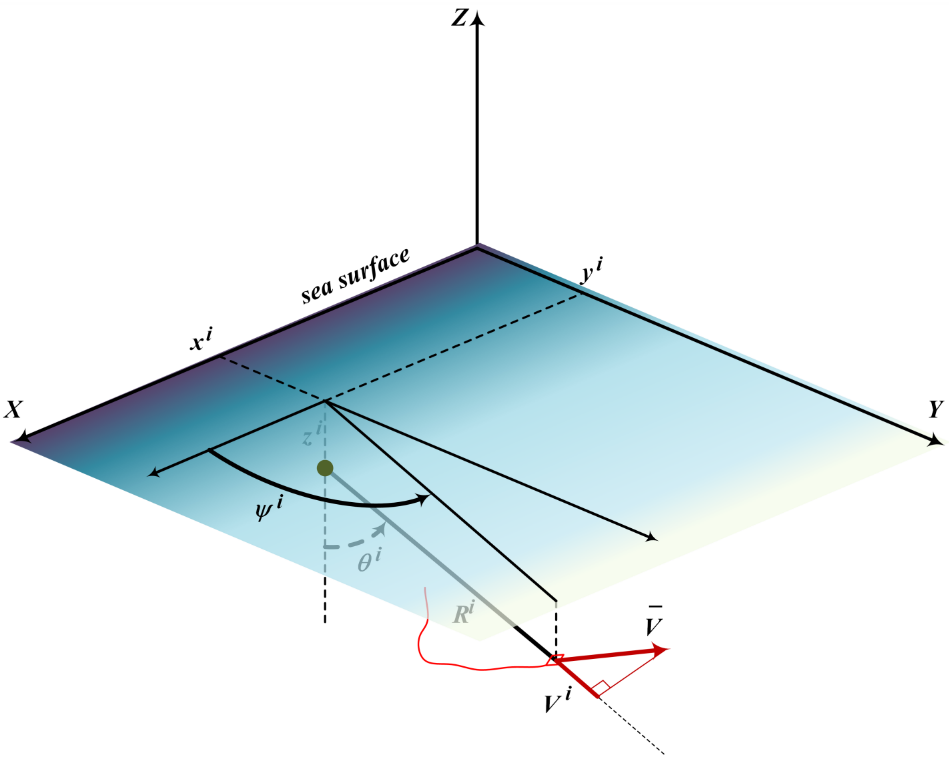
$$R^i_{t_k} = \sqrt{(X_{t_k} - X^i)^2 + (Y_{t_k} - Y^i)^2 + (Z_{t_k} - Z^i)^2},$$
$$r^i_{t_k} = \sqrt{(X_{t_k} - X^i)^2 + (Y_{t_k} - Y^i)^2}$$

расстояние от наблюдателя до цели и длина проекции на плоскость $Z = 0$ соответственно.

Последняя компонента наблюдений описывает эффект Доплера из-за движения аппарата относительно сенсора.

$$\omega^i_{t_k} = \frac{\omega_0}{1 - V^i_{t_k}/C} + v^{\omega^i}_k \tag{1.3}$$

где $V^i_{t_k} = \frac{V^X_{t_k}\left(X_{t_k} - X^i\right) + V^Y_{t_k}\left(Y_{t_k} - Y^i\right) + V^Z_{t_k}\left(Z_{t_k} - Z^i\right)}{R^i_{t_k}}$ – скорость аппарата относительно наблюдателя. Последовательность $\left(v^{\xi^1}_k, v^{\eta^1}_k, v^{\omega^1}_k, \ldots, v^{\xi^N}_k, v^{\eta^N}_k, v^{\omega^N}_k\right)^T$ описывает погрешность показаний датчиков и состоит из независимых и одинаково распределенных векторов с некоррелированными компонентами.



1.3 Формализация

Получили задачу фильтрации с дискретно-непрерывной системой:

$$d\mathcal{X}(t) = a(\mathcal{X}(t))dt + b(t)dW_t,$$

$$\mathcal{X}(0) = \mathcal{X}_0 \sim \pi_0 : \mathbb{E}\left[\mathcal{X}_0\right] = \mu_0$$

$$\operatorname{cov}\left(\mathcal{X}_0, \mathcal{X}_0\right) = \mathbf{D}_0$$

$$\mathcal{Y}\left(t_k\right) = A\left(\mathcal{X}\left(t_k\right)\right) + B_{t_k} W_{t_k}, k \in \mathbb{N}$$

В нашей задаче

$$a\left(\mathcal{X}_t\right) = \begin{pmatrix} v_t \cos\left(\varphi_t\right) \\ v_t \sin\left(\varphi_t\right) \\ 0 \\ 0 \\ \frac{a_t^n}{v_t} \\ -\lambda a_t^n + \nu \end{pmatrix}$$

$$b(t) = b = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \mu \end{pmatrix}$$

Комбинированные пеленгово-доплеровские наблюдения:

$$Y_k = \operatorname{col}\left(\left(\xi_{t_k}^i, \eta_{t_k}^i, \omega_{t_k}^i\right)^T\right)$$

$$A\left(\mathcal{X}_t\right) = \operatorname{col}\left(\left\{\left(\frac{Z_t - Z^i}{R_t^i}, \frac{X_t - X^i}{r_t^i}, \frac{\omega_0}{1 - V_t^i/C}\right)\right\}\right),$$

$$B_{t_k} = B = \operatorname{diag}\left(\operatorname{col}\left(\left\{\left(\sigma_{i,1}^2, \sigma_{i,2}^2, \sigma_{i,3}^2\right)\right\}\right)\right), i = 1, \ldots, N$$

где col() — конкатенация векторов в один, $i = 1, \ldots, N$, N — число датчиков.
Посчитаем для реализации EKF-1 матрицы Якоби для a и A:

$$\frac{\partial a\left(\mathcal{X}_t\right)}{\partial\left(\mathcal{X}_t\right)} = \begin{pmatrix} 0 & 0 & 0 & \cos\varphi_t & -v_t \sin\varphi_t & 0 \\ 0 & 0 & 0 & \sin\varphi_t & v_t \cos\varphi_t & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -a_t^n/v_t^2 & 0 & 1/v_t \\ 0 & 0 & 0 & 0 & 0 & -\lambda \end{pmatrix}$$

$$\frac{\partial A\left(\mathcal{X}_t\right)}{\partial\left(\mathcal{X}_t\right)} = \operatorname{col}\left(\left\{\begin{pmatrix} \xi_X^i & \xi_Y^i & \xi_Z^i & 0 & 0 & 0 \\ \eta_X^i & \eta_Y^i & 0 & 0 & 0 & 0 \\ \omega_X^i & \omega_Y^i & \omega_Z^i & \omega_v^i & \omega_\phi^i & 0 \end{pmatrix}\right\}_{i=1}^N\right),$$

где производные, обозначенные в матрице Якоби $\frac{\partial A(\mathcal{X}_t)}{\partial (\mathcal{X}_t)}$:

$$\xi_X^i = -\frac{\left(Z_t - Z^i\right)\left(X_t - X^i\right)}{\left(R_t^i\right)^3}$$

$$\xi_Y^i = -\frac{\left(Z_t - Z^i\right)\left(Y_t - Y^i\right)}{\left(R_t^i\right)^3}$$

$$\xi_Z^i = \frac{R^2 - \left(Z_t - Z^i\right)^2}{\left(R_t^i\right)^3}$$

$$\eta_X^i = \frac{\left(r_t^i\right)^2 - \left(X_t - X^i\right)^2}{\left(r_t^i\right)^3}$$

$$\eta_Y^i = -\frac{\left(X_t - X^i\right)\left(Y_t - Y^i\right)}{\left(r_t^i\right)^3}$$

$$\omega_X^i = -\omega_0 \frac{V_{t_k}^X \left(X_t - X^i\right) - V_{t_k}^i R}{R^2 C \left(1 - V_{t_k}^i / C\right)^2}$$

$$\omega_Y^i = -\omega_0 \frac{V_{t_k}^Y \left(Y_t - Y^i\right) - V_{t_k}^i R}{R^2 C \left(1 - V_{t_k}^i / C\right)^2}$$

$$\omega_Z^i = -\omega_0 \frac{\left(Z_t - Z^i\right) V_{t_k}^i}{R^2 C \left(1 - V_{t_k}^i / C\right)^2}$$

$$\omega_v^i = \omega_0 \frac{V_{t_k}^i}{v_t C \left(1 - V_{t_k}^i / C\right)^2}$$

$$\omega_\phi^i = \omega_0 \frac{\left(v_t \cos\varphi_t\right)_t \left(Y_t - Y^i\right) - \left(v_t \sin\varphi_t\right)_t \left(X_t - X^i\right)}{RC \left(1 - V_{t_k}^i / C\right)^2}$$

2 Алгоритмы фильтрации

2.1 EKF-1 — Расширенный фильтр Калмана первого порядка

Расширенный фильтр Калмана — это линеаризованный фильтр Калмана, в котором в качестве опорной траектории выступает оценка на предыдущем шаге, т.е. $_t = \hat{X}(t-1)$.

В нашей задаче используется EKF первого порядка для непрерывно-дискретных систем наблюдения, в котором уравнение динамики является СДУ, а наблюдения дискретны.

Для моделирования динамики и прогнозов используется метод Эйлера-Маруямы.

1. Начальное условие

$$\hat{X}(0) = m_0,$$
$$k(0) = D_0.$$

2. Шаг прогноза.

$$d\hat{X}(t) = a(\hat{X}(t))dt, \quad t \in (t_{k-1}, t_k) \tag{2.1}$$

$$\dot{k}(t) = \left.\frac{\partial a(x)}{\partial x}\right|_{\hat{X}(t)} k(t) + k(t)\left(\left.\frac{\partial a(x)}{\partial x}\right|_{\hat{X}(t)}\right)^T + b(t)b^T(t), \quad t \in (t_{k-1}, t_k) \tag{2.2}$$

3. Шаг коррекции.

$$\hat{X}\left(t_k\right) = \hat{X}\left(t_{k-}\right) + k\left(t_{k-}\right) \times$$
$$\times \left(\left.\frac{\partial A_t(x)}{\partial x}\right|_{\hat{X}(t_{k-})}\right)^T \left(\left.\frac{\partial A_t(x)}{\partial x}\right|_{\hat{X}(t_{k-})} k\left(t_{k-}\right)\left(\left.\frac{\partial A_t(x)}{\partial x}\right|_{\hat{X}(t_{k-})}\right)^T + B_{t_{k-}}B^T_{t_{k-}}\right)^{-1} \times \Big(Y_k - A_t\left(\hat{X}\left(t_{k-}\right)\right)\Big) \tag{2.3}$$

$$k\left(t_k\right) = k\left(t_k\right) - k\left(t_{k-}\right) \times$$
$$\times \left(\left.\frac{\partial A_t(x)}{\partial x}\right|_{\hat{X}(t_{k-})}\right)^T \left(\left.\frac{\partial A_t(x)}{\partial x}\right|_{\hat{X}(t_{k-})} k\left(t_{k-}\right)\left(\left.\frac{\partial A_t(x)}{\partial x}\right|_{\hat{X}(t_{k-})}\right)^T + B_{t_{k-}}B^T_{t_{k-}}\right)^{-1} \times \frac{\partial A_t(x)}{\partial x}\Big|_{\hat{X}(t_{k-})} k\left(t_{k-}\right) \tag{2.4}$$

2.2 Корневой аналог расширенного фильтра Калмана

1. Начальное условие

$$\hat{X}(0) = m_0,$$
$$k(0) = D_0.$$

2. Шаг прогноза.

$$\tilde{X}_t = a_t \hat{X}_{t-1} + c_t \tag{2.5}$$

$$N\{[\underbrace{\tilde{s}_t}_{N} \underbrace{0}_{M}] = \begin{bmatrix} a_t s_{t-1} & \left(b_t b_t^T\right)^{\frac{1}{2}} \end{bmatrix} \tilde{T}_t \tag{2.6}$$

где $\tilde{T}_t$ - ортогональная матрица, обеспечивающая выполнение равенства. В работе найдена с помощью процедуры Грама-Шмидта, а точнее с помощью использований QR-разложений для ортогонализации (реализация в коде ниже).

3. Шаг коррекции (вариант последовательной обработки скалярных измерений):

$$i = 1, \ldots, M, \quad s_t^0 = \tilde{s}_t, \quad s_t = s_t^M, \quad \hat{X}_t^0 = \tilde{X}_t, \quad \hat{X}_t = \hat{X}_t^M :$$
$$D_t^i = A_t^i s_t^{i-1},$$

$$\mu_t^i = \left(D_t^i \left(D_t^i\right)^T + B_t^1 \left(B_t^1\right)^T\right)^{-1}$$

$$v_t^i = \left(1 + \sqrt{\mu_t^i B_t^1 \left(B_t^1\right)^T}\right)^{-1}$$

$$\hat{X}_t^i = \hat{X}_t^{i-1} + \mu_t^i s_t^{i-1} \left(D_t^i\right)^T \left(Y_t^i - A_t^i \hat{X}_t^{i-1} - C_t^i\right) \tag{2.7}$$

$$s_t^i = s_t^{i-1} - \mu_t^i v_t^i s_t^{i-1} \left(D_t^i\right)^T D_t^i \tag{2.8}$$

2.3 CMNF — Условно-минимаксный нелинейный фильтр

1. Начальное условие.

Моделируется выборка синтетического состояния $\left\{X_0^{(i)}\right\}_{i=\overline{1,N}} \sim \pi_0(x)$. Вычисляются $\hat{X}_0$ - целевая оценка фильтрации в начальный момент времени, $\hat{K}_0^{XX}$ - ковариационная матрица ошибки оценки фильтрации в начальный момент времени. Формируется выборка синтетических оценок фильтрации в начальный момент времени $\left\{\hat{X}_0^{(i)}\right\}_{i=\overline{1,N}} : \hat{X}_0^{(i)} \equiv \hat{X}_0$.

2. Шаг прогноза.

Пусть на предыдущем шаге $k-1$ имеются оценка состояния $\hat{X}_{k-1}$, выборки синтетических состояний $\left\{X_{k-1}^{(i)}\right\}_{i=\overline{1,N}}$ и синтетических оценок фильтрации $\left\{\hat{X}_{k-1}^{(i)}\right\}_{i=\overline{1,N}}$.

2.1. Методом Эйлера-Маруямы с малым шагом моделируем выборку синтетического состояния $\left\{X_k^{(i)}\right\}_{i=\overline{1,N}}$ в следующий момент времени t_k. По ней строим наблюдения $\left\{Y_k^{(i)}\right\}_{i=\overline{1,N}}$.

2.2. Вычисляем выборки синтетических базовых прогнозов $\left\{\alpha_k\left(\hat{X}_{k-1}^{(i)}\right)\right\}_{i=\overline{1,N}}$ и синтетических

$$\hat{X}_0 = \frac{1}{N}\sum_{i=1}^{N} X_0^{(i)}, \quad \hat{K}_0^{XX} = \frac{1}{N-n}\sum_{i=1}^{N}\left(X_0^{(i)} - \hat{X}_0\right)\left(X_0^{(i)} - \hat{X}_0\right)^T \tag{2.9}$$

базовых коррекций $\left\{\gamma_k\left(\hat{X}_{k-1}^{(i)}, \beta_k\left(Y_k^{(i)}\right)\right)\right\}_{i=\overline{1,N}}$.

2.3. По пучку синтетических объектов

Строим выборочные моменты $m_k, \bar{R}_k$. В качестве верхней оценки в работе взяли R с занесением отрицельных элементов

2.4. По целевой оценке фильтрации $\hat{X}_{k-1}$ и наблюдениям Y_k вычисляем целевой базовый прогноз $\alpha_k\left(\hat{X}_{k-1}\right)$ и целевую базовую коррекцию $\gamma_k\left(\hat{X}_{k-1}, \beta_k\left(Y_k\right)\right)$.

2.5. Модифицируем выборку синтетических прогнозов $\left\{\alpha_k\left(\hat{X}_{k-1}^{(i)}\right)\right\}_{i=\overline{1,N}}$ и синтетических коррекцию $\gamma_k\left(\hat{X}_{k-1}, \beta_k\left(Y_k\right)\right)$

3. Шаг коррекции.

3.1. По выборке синтетических объектов

строим выборку синтетических оценок фильтрации $\left\{\hat{X}_k^{(i)}\right\}_{i=\overline{1,N}}$.

3.2. По модифицированным целевым прогнозу и коррекции $\begin{bmatrix} \breve{X}_k \\ \breve{\gamma}_k \end{bmatrix}$ строим оценку состояния $\hat{X}_k$ и ковариационную матрицу ее ошибки $\bar{R}_k^{XX}$.

2.4 Необходимые для реализации дополнительные сведения

Моделируются состояния с помощью метода Эйлера-Маруямы с шагом h :

$$0 = t_0 < t_1 < \ldots < t_N = T, \quad h = T/N$$

Получаем стохастическую динамическую систему наблюдения с дискретным временем:

$$X_0 = X(0)$$

$$b\left(X_t\right) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \mu \end{pmatrix}, X_t \sim \mathcal{N}(0,1)$$

$$X_{t+1} = X_t + a\left(X_t\right) \cdot h_1 + b\left(X_t\right) * V_t \cdot \sqrt{h} \tag{2.10}$$

- Шаг для наблюдений шаг $\delta_3 = 1$ секунда.
- Шаг фильтрации для прогноза шаг $\delta_2 = 10^{-2}$ секунд.
- Шаг для моделирования траектории $\delta_1 = 10^{-3}$ секунд.
- Размер обучающей выборки CMNF составляет N = 10^3 частиц.
- Для вычисления СКО моделируем пучок из 10^4 траекторий.
- Критерий развала фильтра — превышение 5 аппроксимаций СКО.

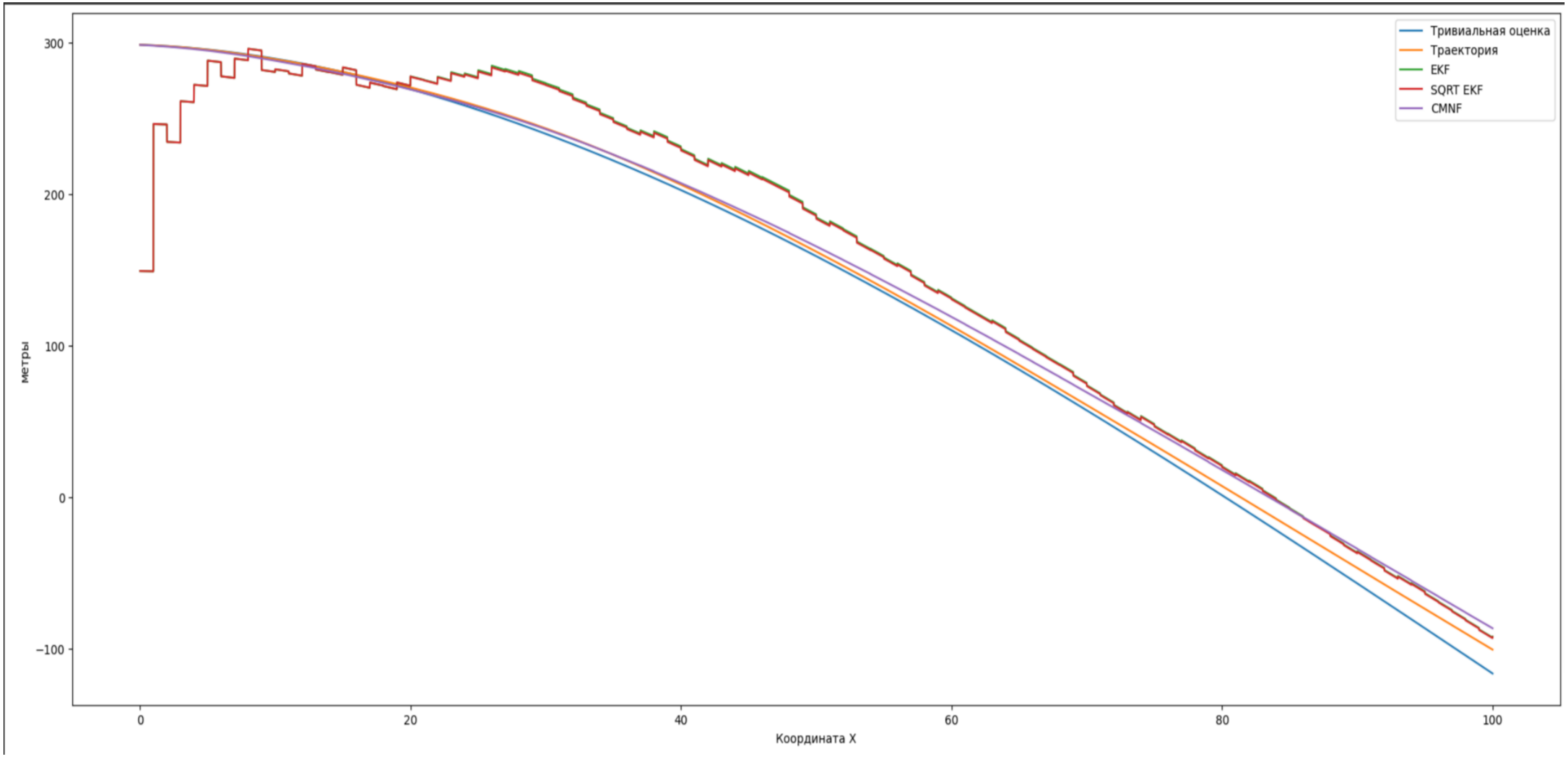
Константы из статьи.

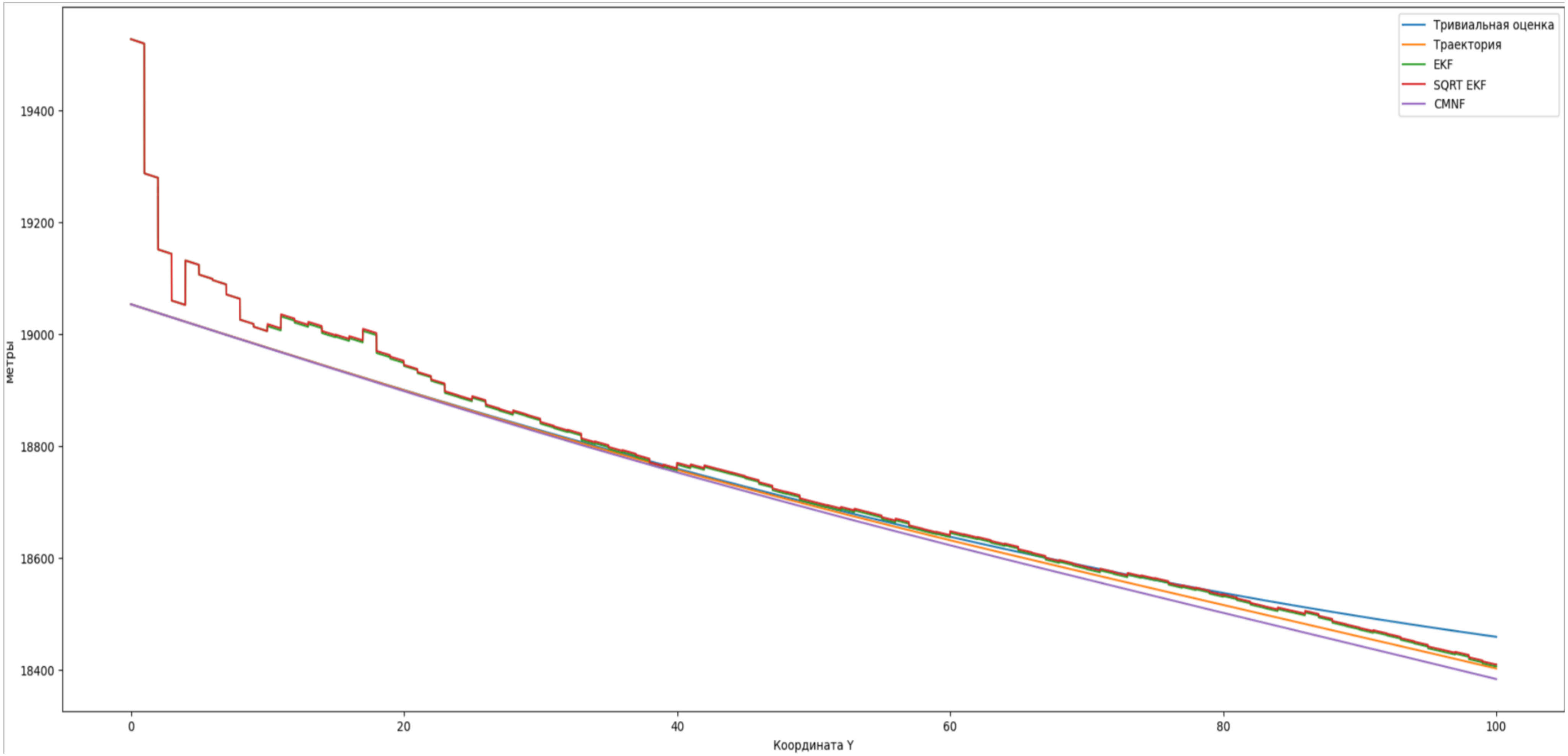
$\mu_{01} = 0$ м — среднее для 1-й компоненты
$\sigma_{01} = 1000$ м — отклонение для 1-й компоненты
$\mu_{02} = 20000$ м — среднее для 2-й компоненты
$\sigma_{02} = 1000$ м — отклонение для 2-й компоненты
$\mu_{03} = -1000$ м — среднее для 3-й компоненты
$\sigma_{03} = 100$ м — отклонение для 3-й компоненты
$v_{min} = 5$ м/с — минимальная скорость
$v_{max} = 12$ м/с — максимальная скорость
$\mu_{0\phi} = -\frac{\pi}{2}$ рад — среднее начального градуса
$\sigma_{0\phi} = 0.1$ рад — отклонение начального градуса
$a_{min} = -0.2$ — минимальное нормальное ускорение
$a_{max} = 0.2$ — максимальное нормальное ускорение
$\lambda = 0.01$
$\mu = 0.01$
$\nu = 0$
$freq = 1$ гц — частота измерений
$TotalT = 100$ с — длительность манёвра

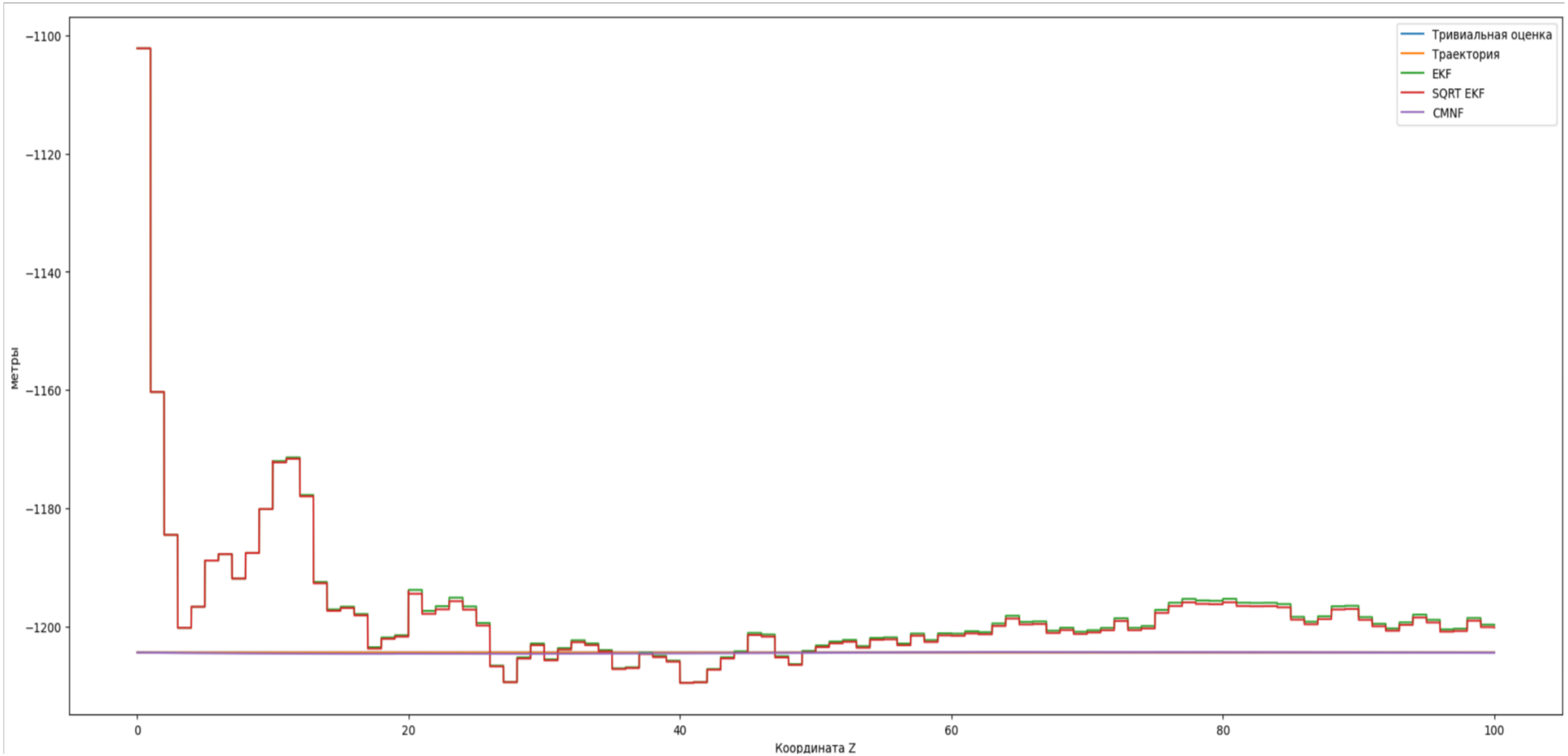
3 Эксперименты

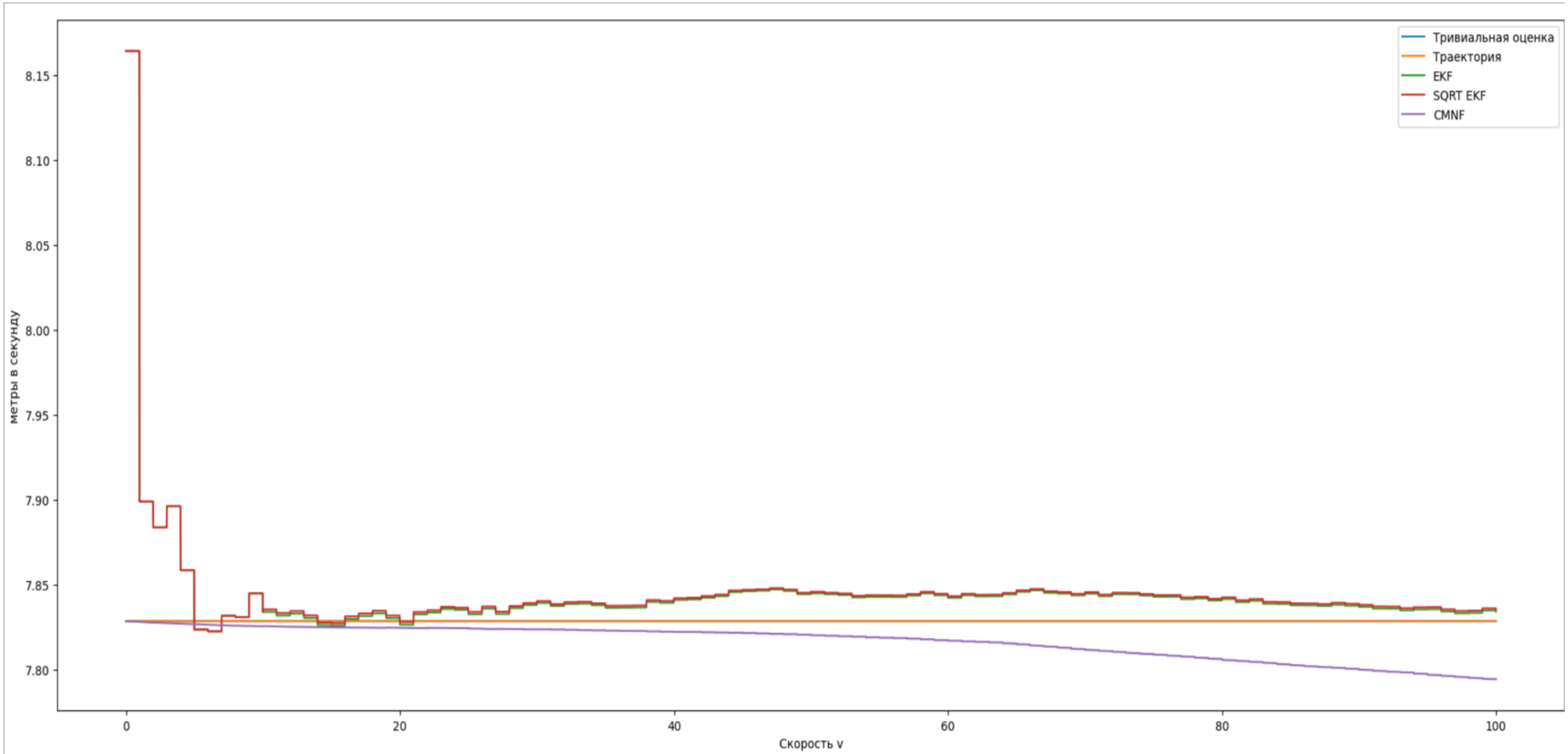
3.1 Компоненты и их оценки

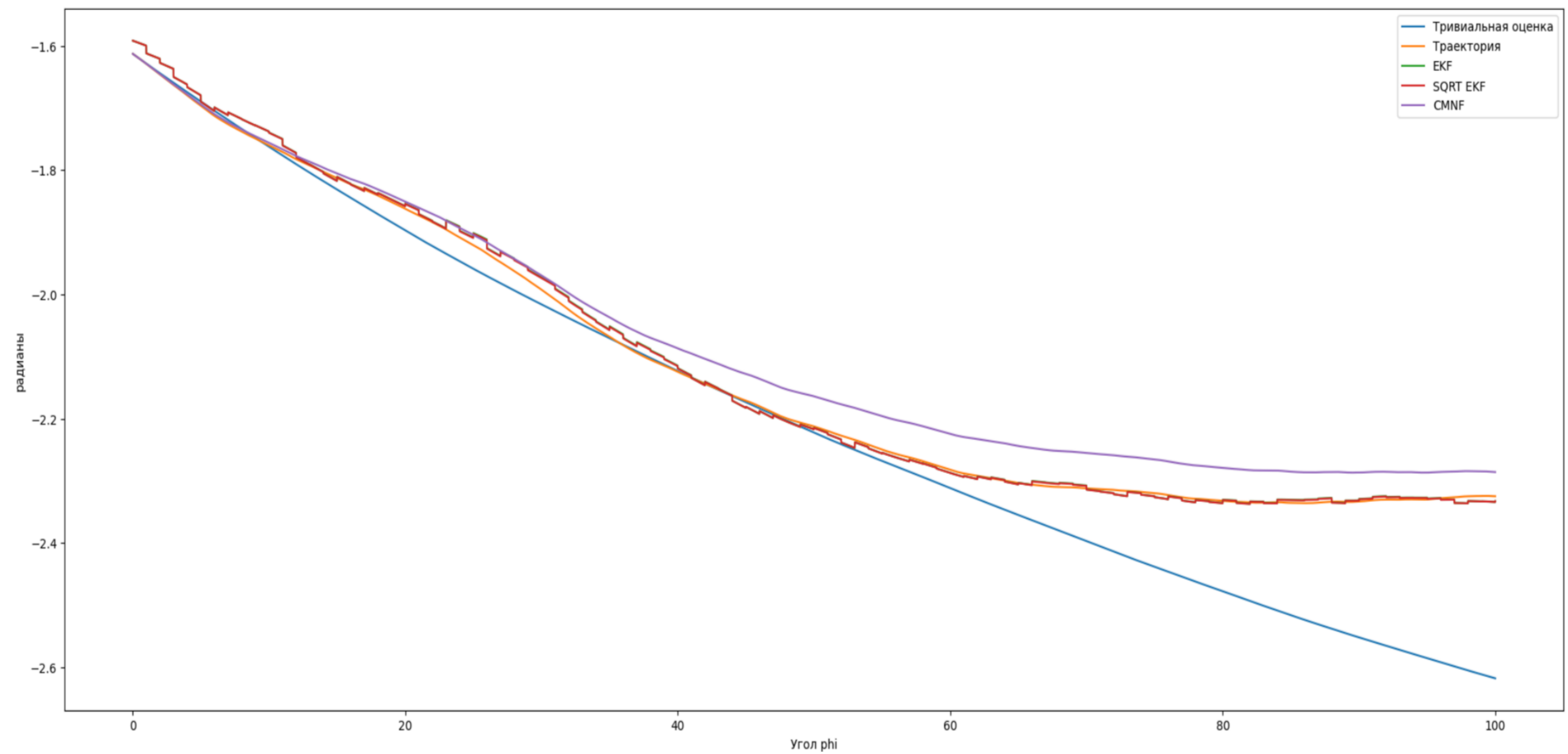
Замечание. Во всех экспериментах ниже результаты работы фильтров Калмана практически сливаются.

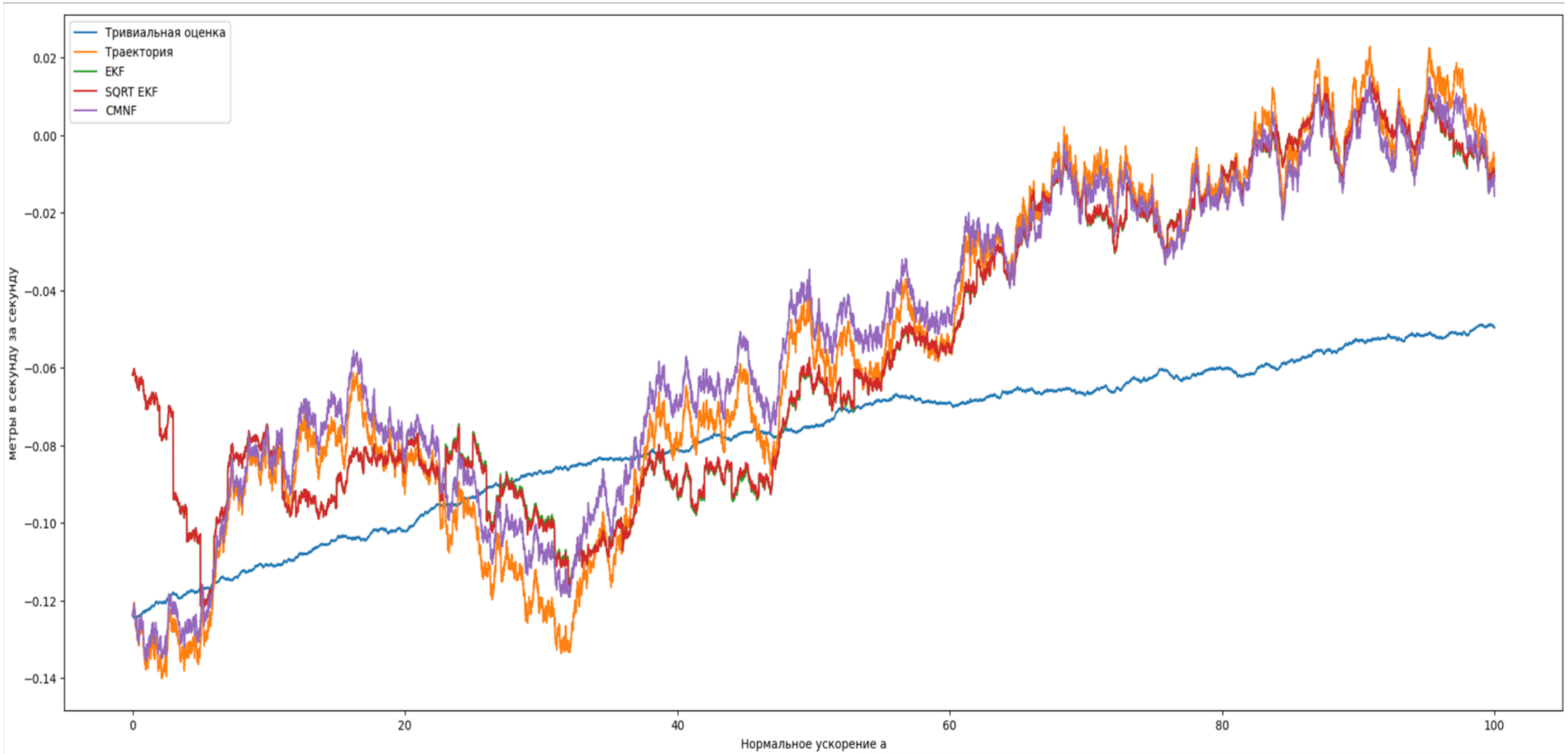



Тривиальная оценка
Траектория
EKF
SQRT EKF
CMNF
метры
Координата Y
19400
19200
19000
18800
18600
18400
0
20
40
60
80
100

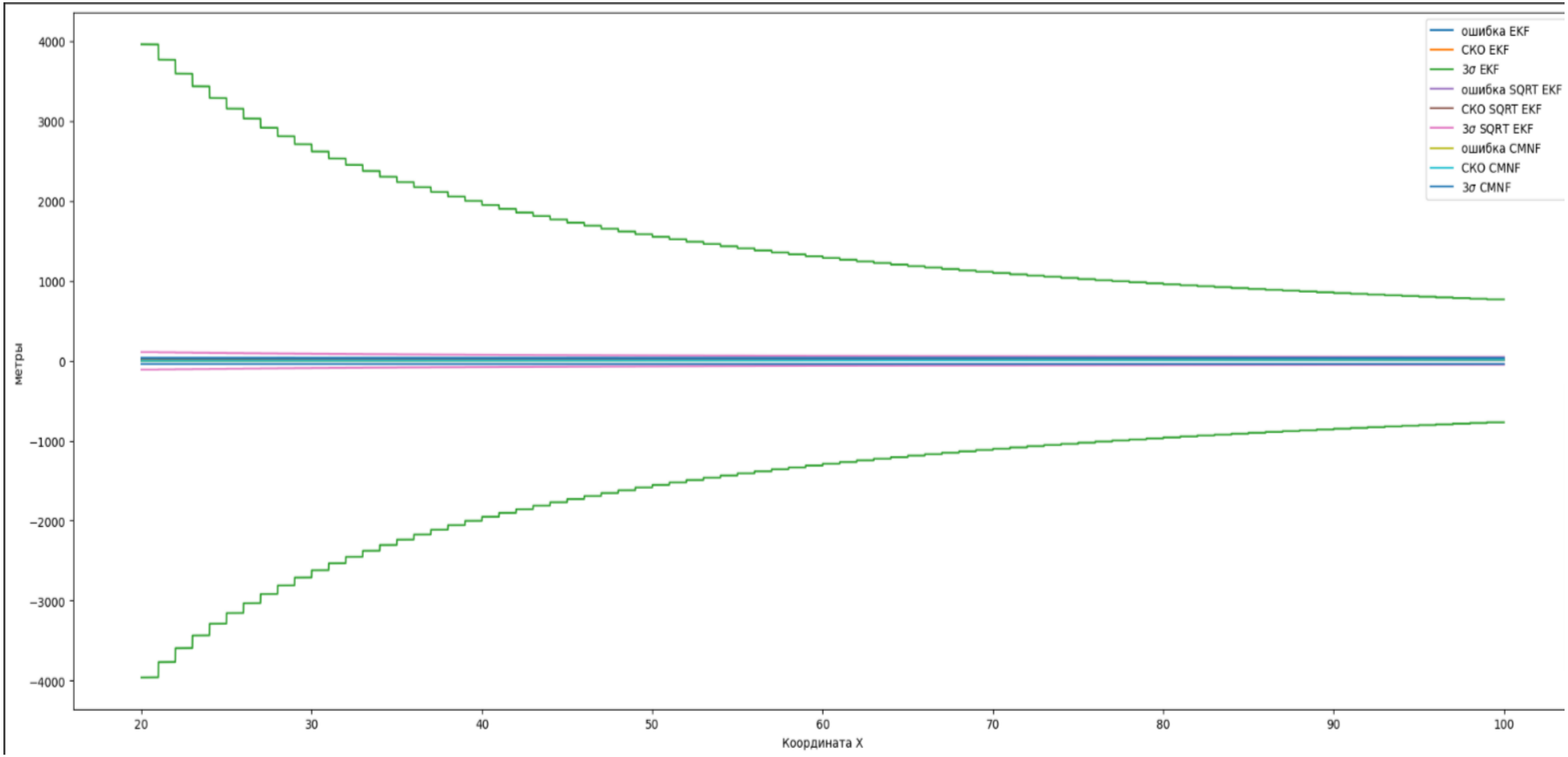

Тривиальная оценка
Траектория
EKF
SQRT EKF
CMNF
метры
Координата Z
−1100
−1120
−1140
−1160
−1180
−1200
0
20
40
60
80
100

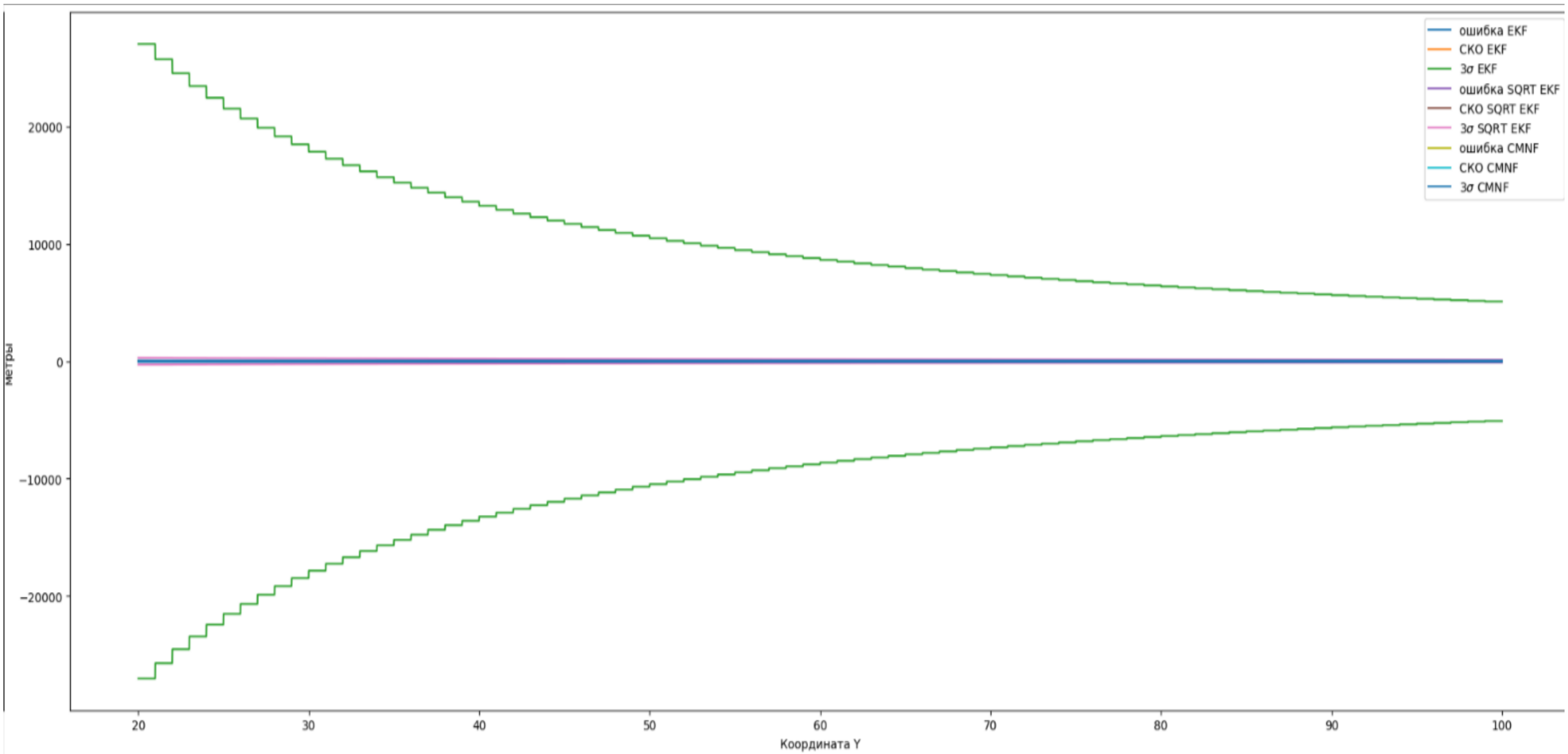

Тривиальная оценка
Траектория
EKF
SQRT EKF
CMNF
метры в секунду
Скорость v
8.15
8.10
8.05
8.00
7.95
7.90
7.85
7.80
0
20
40
60
80
100


Тривиальная оценка
Траектория
EKF
SQRT EKF
CMNF
−1.6
−1.8
−2.0
−2.2
−2.4
−2.6
радианы
0
20
40
60
80
100
Угол phi

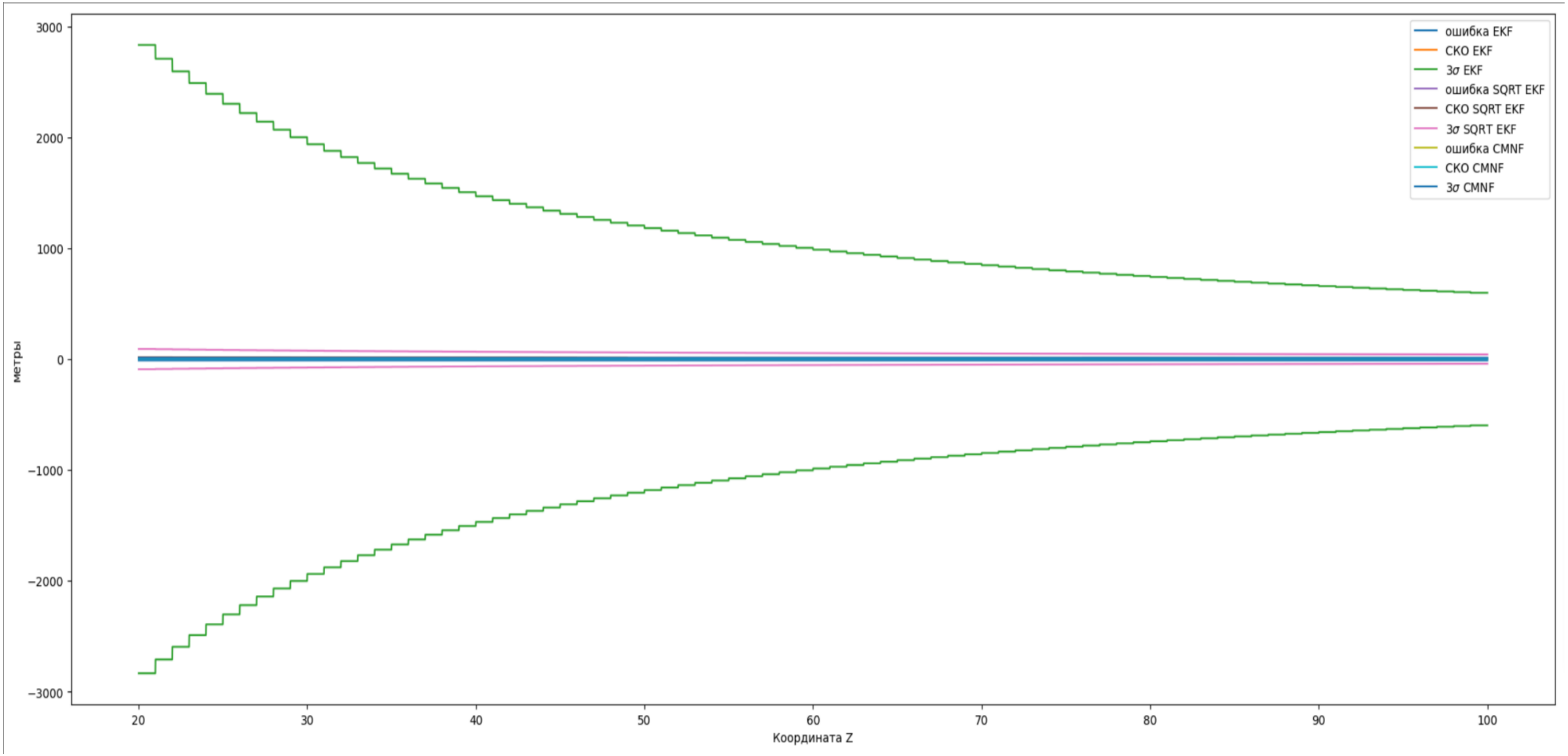

Тривиальная оценка
Траектория
EKF
SQRT EKF
CMNF
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
−0.14
метры в секунду за секунду
0
20
40
60
80
100
Нормальное ускорение a

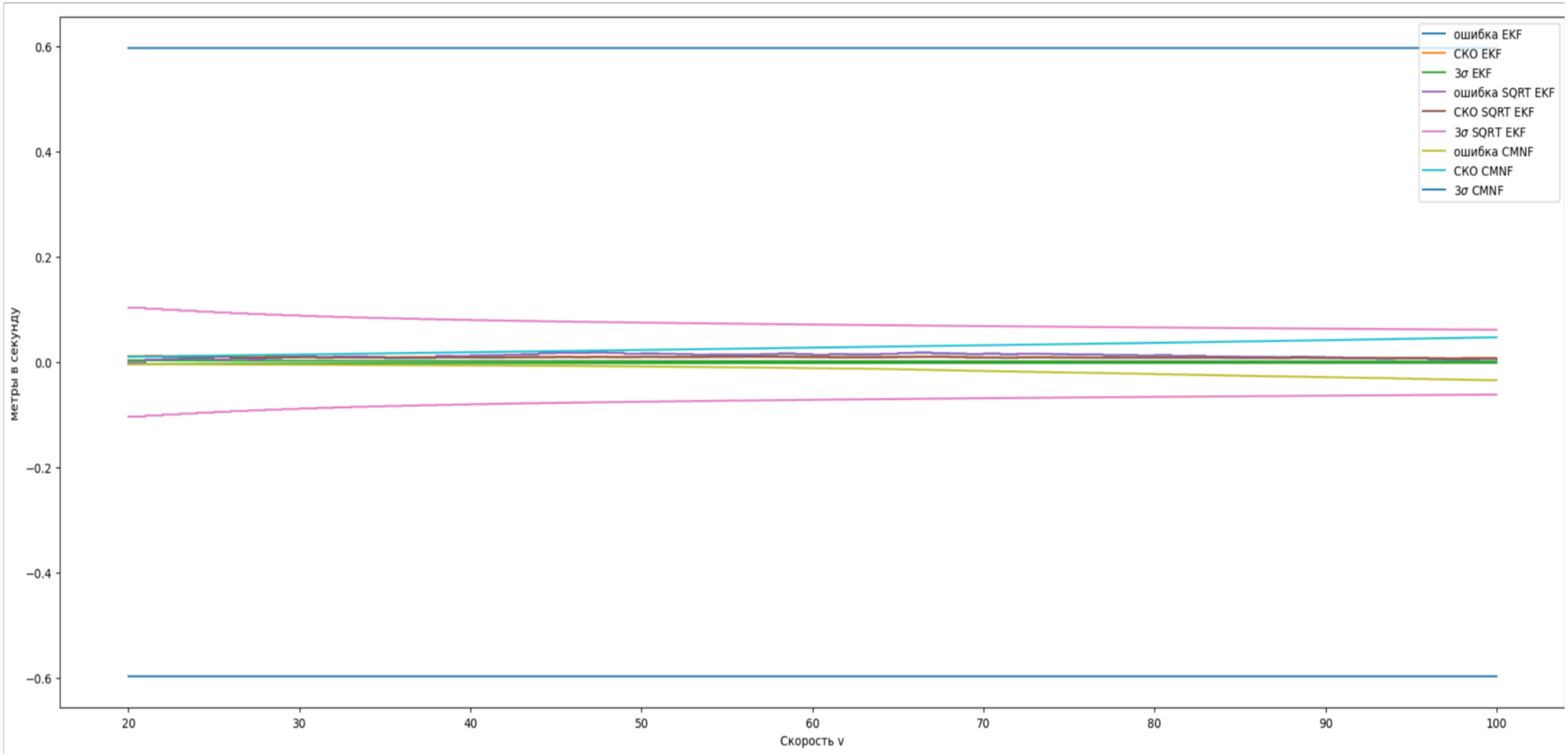
3.2 Ошибки оценивания и оценки СКО по алгоритмам

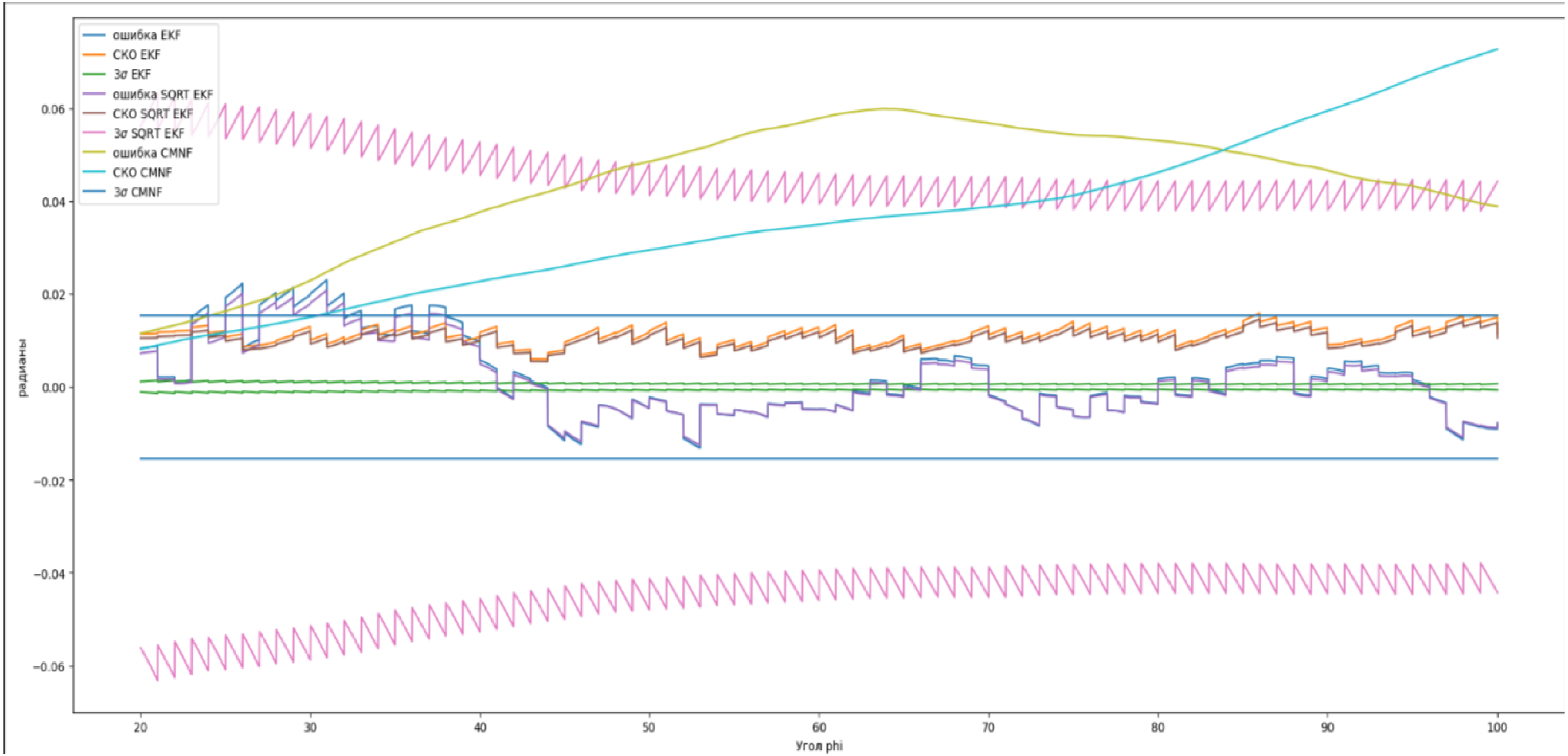




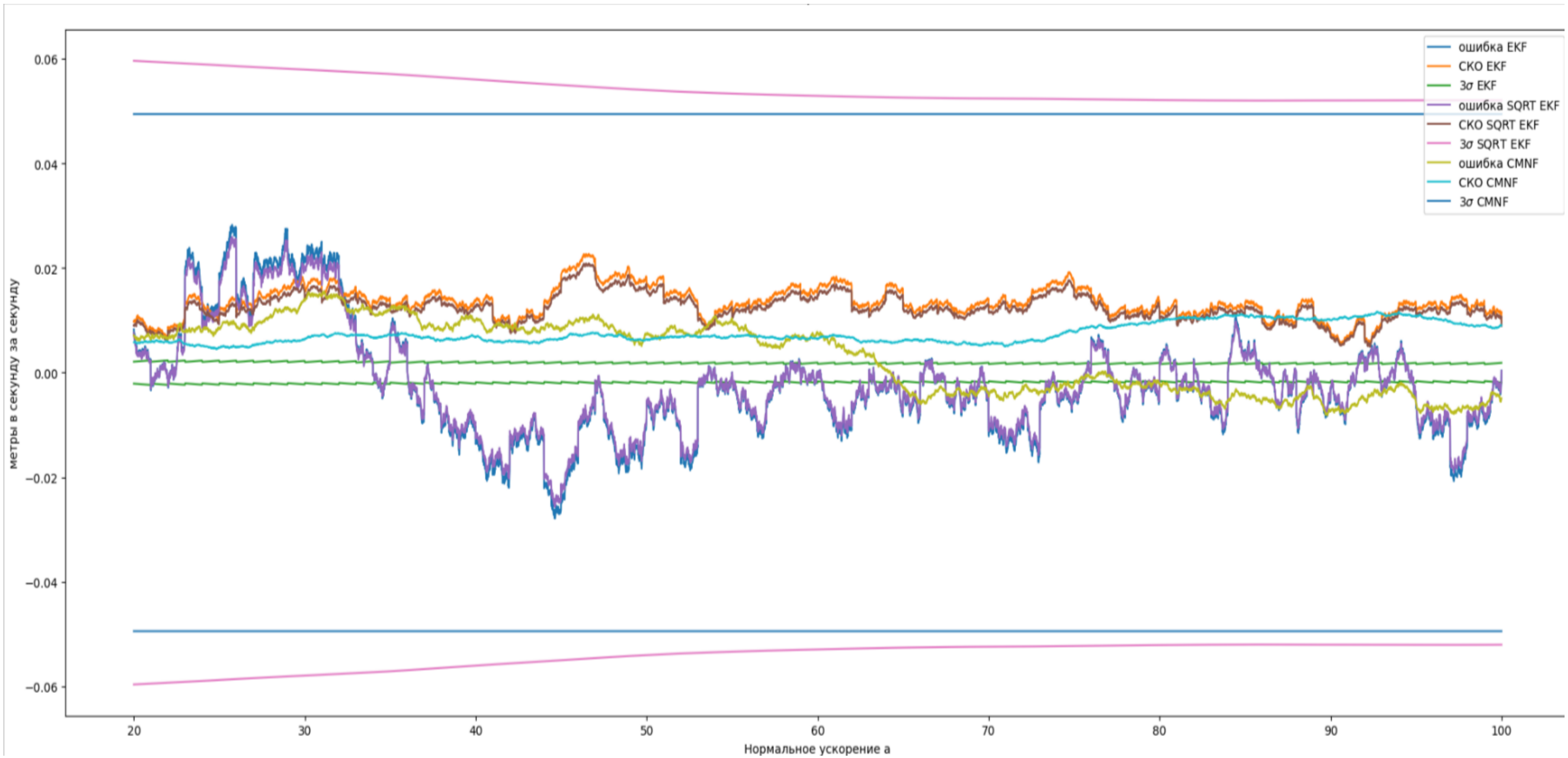
ошибка EKF
СКО EKF
3σ EKF
ошибка SQRT EKF
СКО SQRT EKF
3σ SQRT EKF
ошибка CMNF
СКО CMNF
3σ CMNF
метры
Координата Y
20000
10000
0
−10000
−20000
20
30
40
50
60
70
80
90
100


ошибка EKF
СКО EKF
3σ EKF
ошибка SQRT EKF
СКО SQRT EKF
3σ SQRT EKF
ошибка CMNF
СКО CMNF
3σ CMNF
метры
Координата Z
3000
2000
1000
0
−1000
−2000
−3000
20
30
40
50
60
70
80
90
100

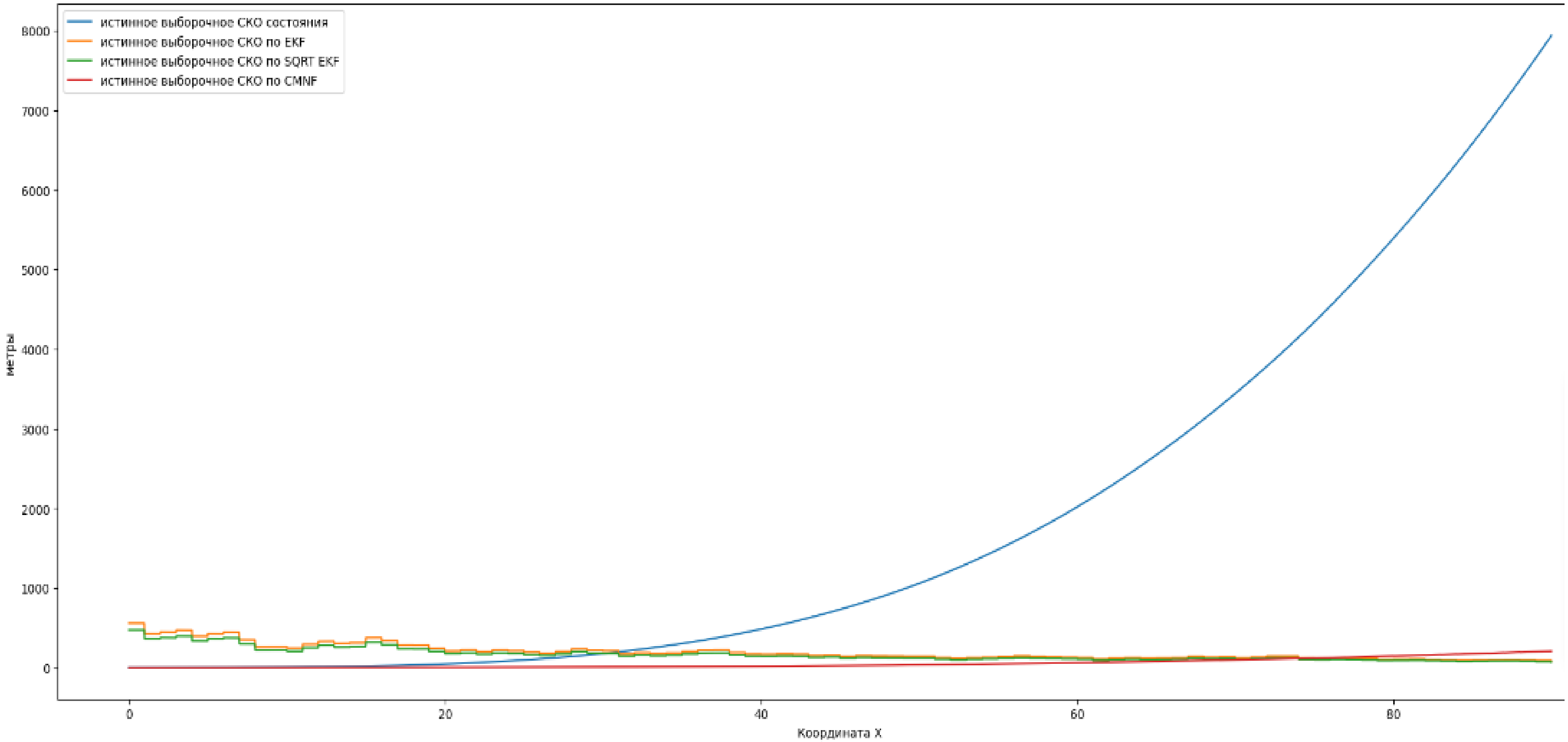

ошибка EKF
СКО EKF
3σ EKF
ошибка SQRT EKF
СКО SQRT EKF
3σ SQRT EKF
ошибка CMNF
СКО CMNF
3σ CMNF
метры в секунду
Скорость v
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
20
30
40
50
60
70
80
90
100

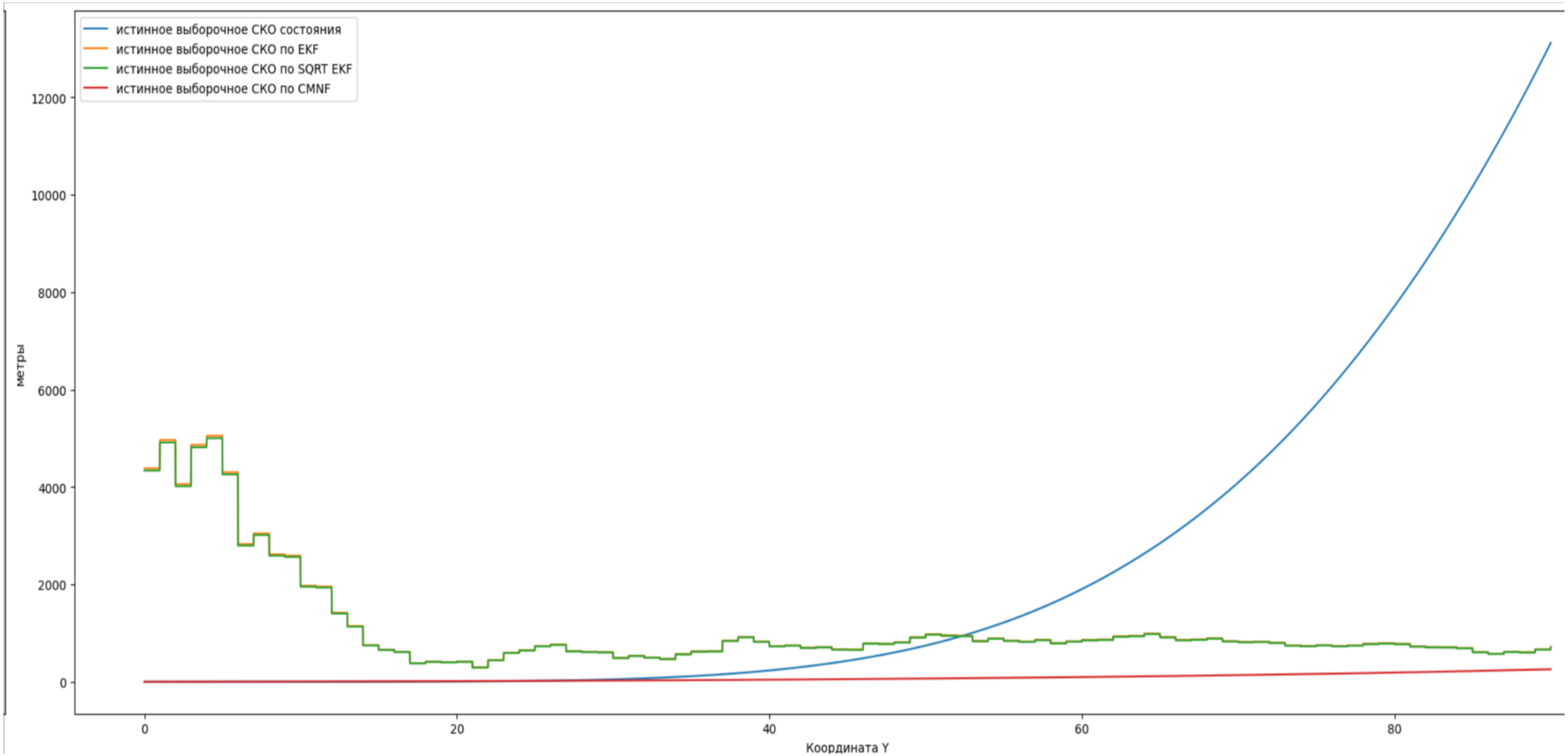


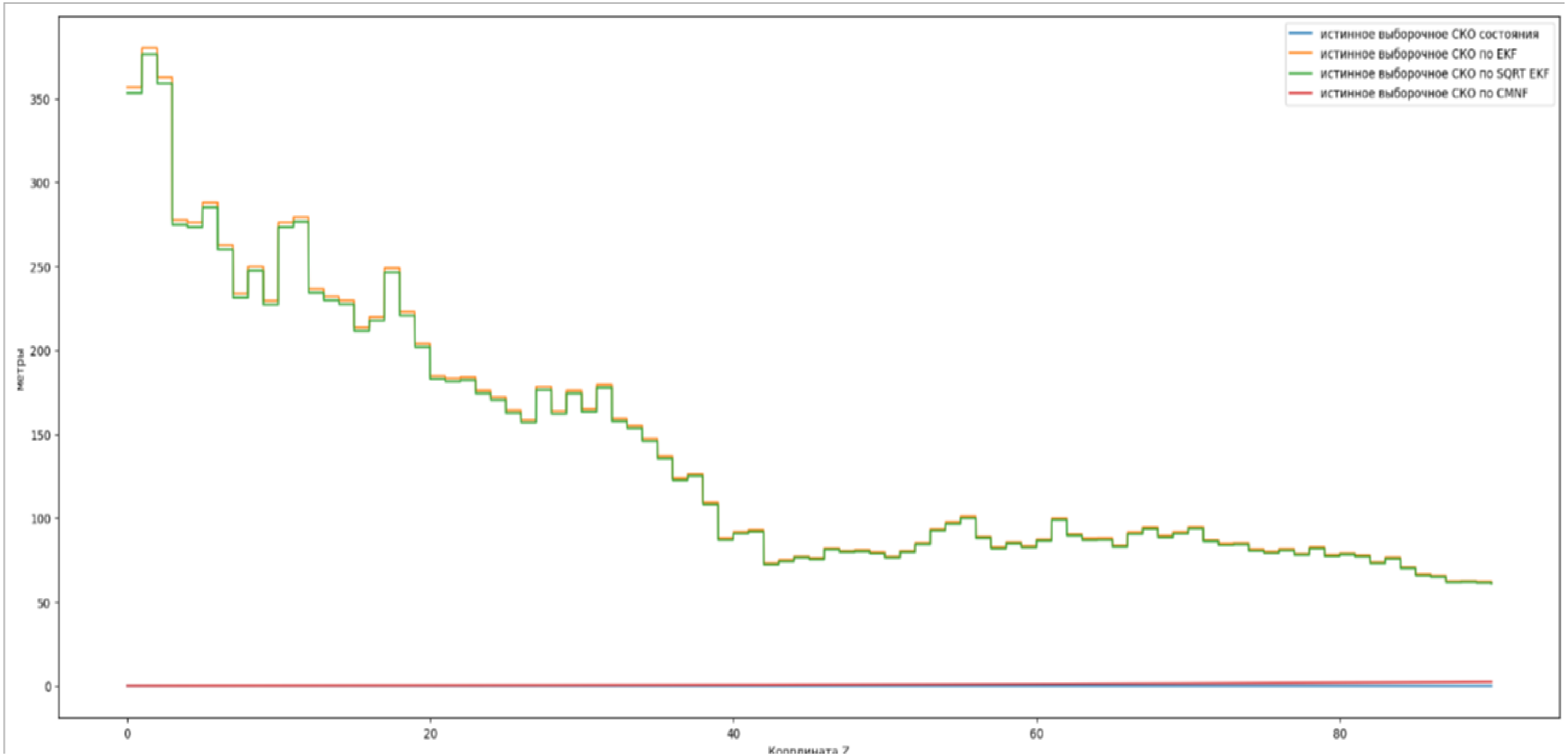
ошибка EKF
СКО EKF
3σ EKF
ошибка SQRT EKF
СКО SQRT EKF
3σ SQRT EKF
ошибка CMNF
СКО CMNF
3σ CMNF
радианы
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
20
30
40
50
60
70
80
90
100
Угол phi

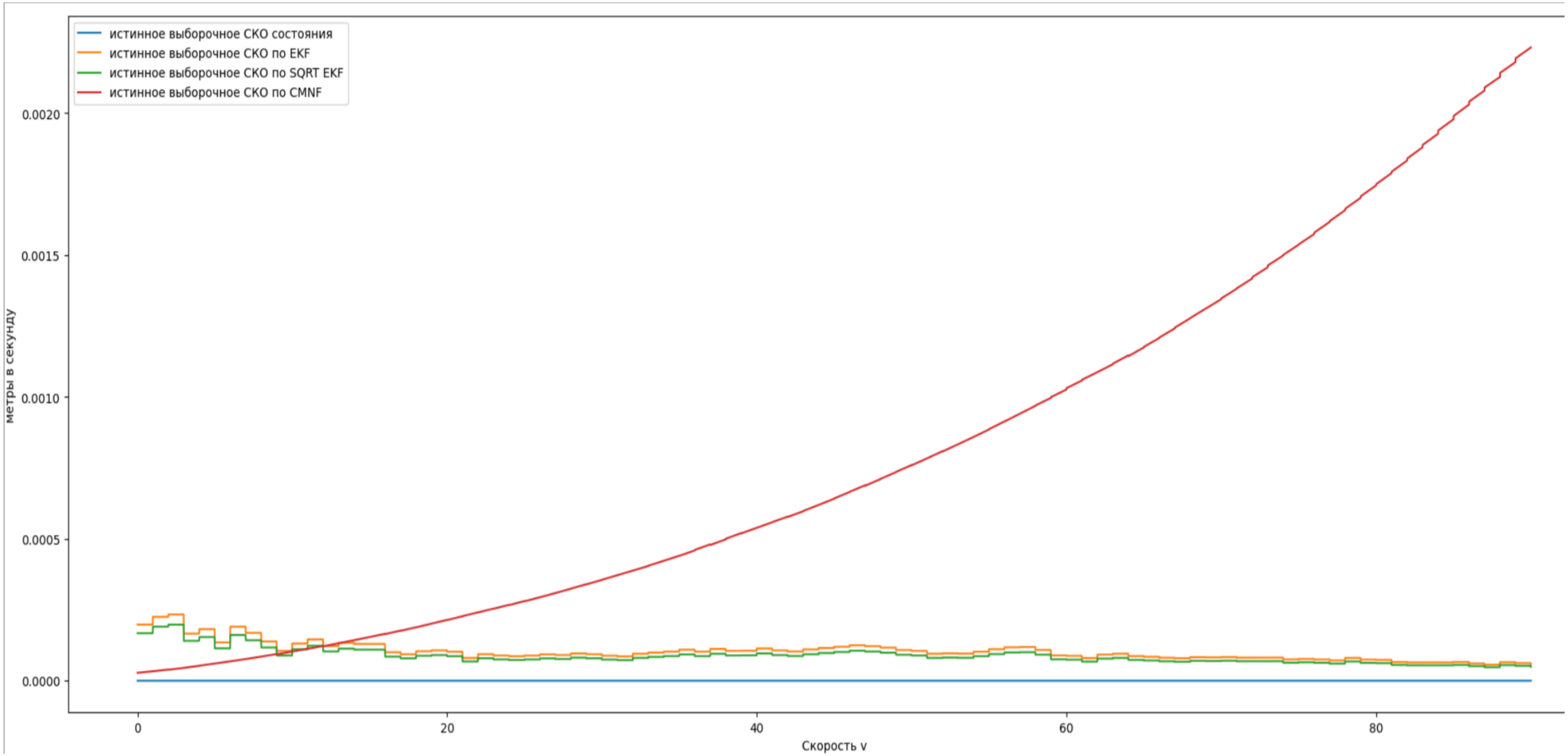

ошибка EKF
СКО EKF
3σ EKF
ошибка SQRT EKF
СКО SQRT EKF
3σ SQRT EKF
ошибка CMNF
СКО CMNF
3σ CMNF
метры в секунду за секунду
Нормальное ускорение а
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
20
30
40
50
60
70
80
90
100

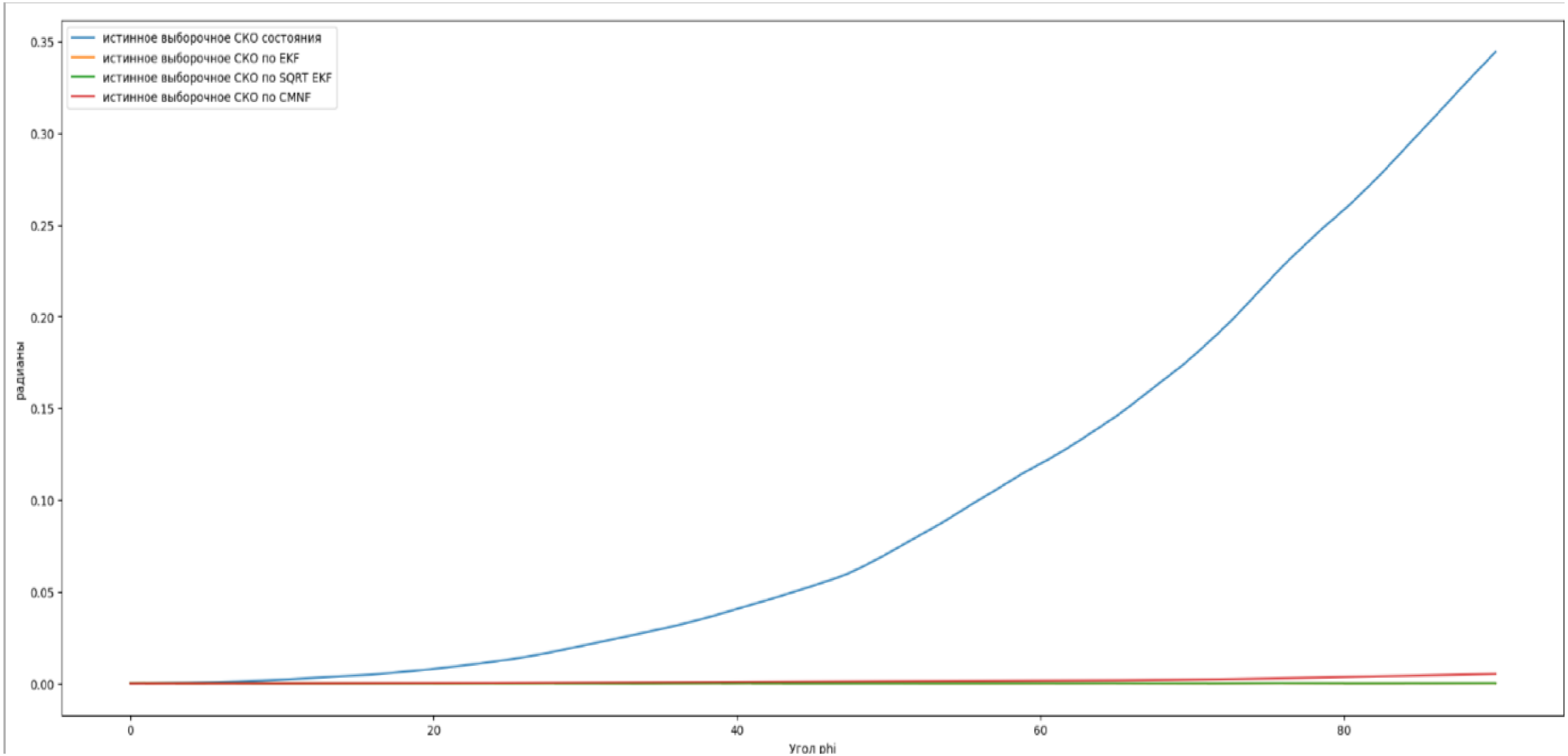
3.3 Истинные выборочные СКО

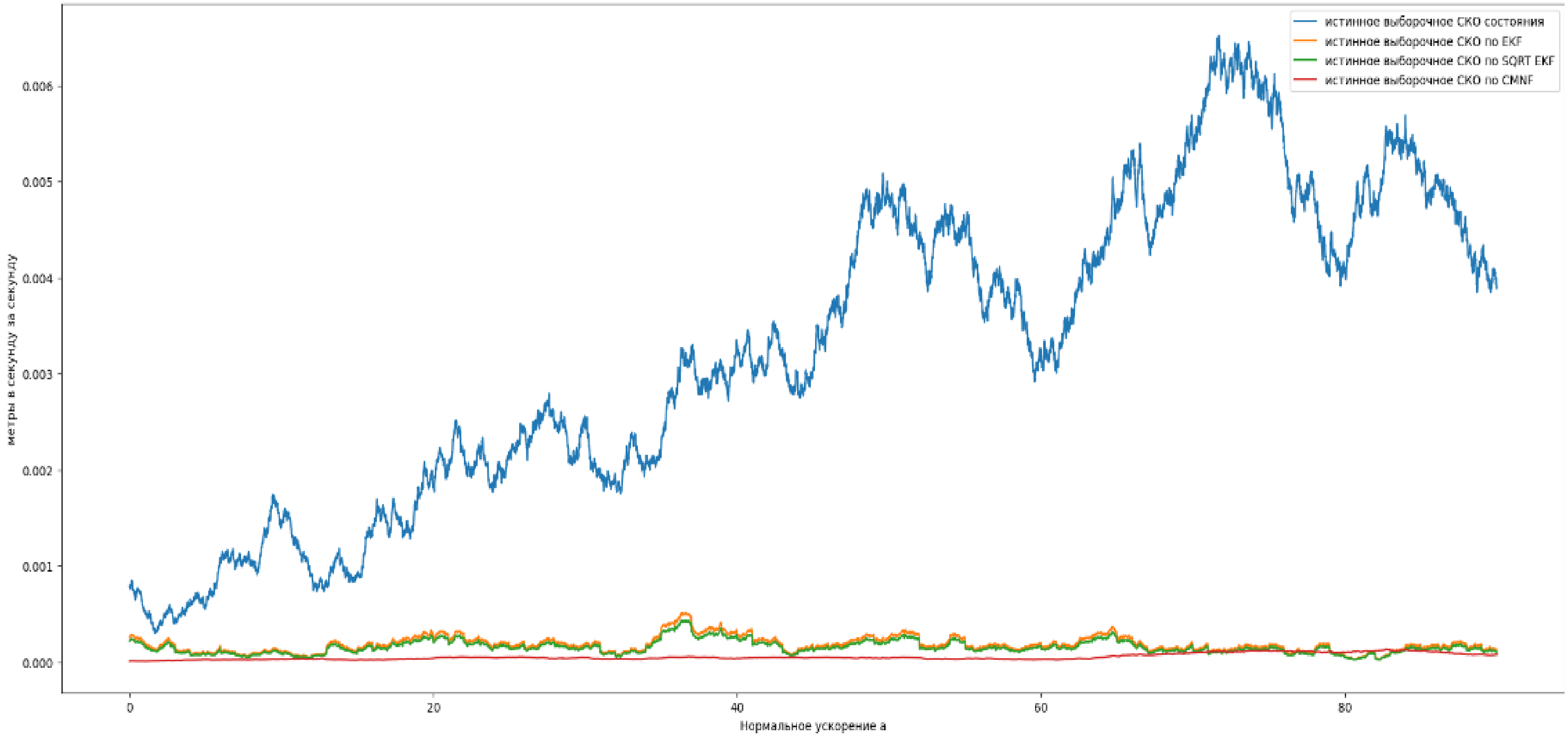



истинное выборочное СКО состояния
истинное выборочное СКО по EKF
истинное выборочное СКО по SQRT EKF
истинное выборочное СКО по CMNF
метры
Координата Y
12000
10000
8000
6000
4000
2000
0
0
20
40
60
80


истинное выборочное СКО состояния
истинное выборочное СКО по EKF
истинное выборочное СКО по SQRT EKF
истинное выборочное СКО по CMNF
метры
Координата Z
0
50
100
150
200
250
300
350
0
20
40
60
80


истинное выборочное СКО состояния
истинное выборочное СКО по EKF
истинное выборочное СКО по SQRT EKF
истинное выборочное СКО по CMNF
метры в секунду
0.0020
0.0015
0.0010
0.0005
0.0000
0
20
40
60
80
Скорость v


истинное выборочное СКО состояния
истинное выборочное СКО по EKF
истинное выборочное СКО по SQRT EKF
истинное выборочное СКО по CMNF
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
радианы
0
20
40
60
80
Угол phi


истинное выборочное СКО состояния
истинное выборочное СКО по EKF
истинное выборочное СКО по SQRT EKF
истинное выборочное СКО по CMNF
метры в секунду за секунду
0.006
0.005
0.004
0.003
0.002
0.001
0.000
0
20
40
60
80
Нормальное ускорение a

3.4 Процент расходящихся траекторий и выборочные СКО

	расходятся процентов
EKF	0.37
SQRT EKF	0.36
CMNF	0.14

Выборочные СКО сразу после начала оценивания, при t=1 (с):

	X	Y	Z	v	phi	a
по EKF	76.274745	168.902810	35.67415	0.08983141	0.018260	0.063410
по SQRT EKF	70.172403	155.390584	32.82022	0.08264490	0.016799	0.058337
по CMNF	0.118142	0.115761	0.03068806	$6.253657e-04$	0.000128	0.001551
по состояниям	0.001647	0.000179	$1.641047e-12$	$1.351242e-14$	0.000715	0.009675

Выборочные СКО при t=10 (с):

	X	Y	Z	v	phi	a
по EKF	23.637096	66.205485	18.88562	$1.406461e-02$	0.018107	0.016104
по SQRT EKF	21.746099	60.909377	17.37476	$1.294030e-02$	0.016663	0.014826
по CMNF	0.809949	1.215246	0.2452941	$5.288142e-03$	0.002781	0.003286
по состояниям	0.550414	0.074114	$1.641047e-12$	$1.351242e-14$	0.016752	0.028091

Выборочные СКО при t=50 (с):

	X	Y	Z	v	phi	a
по EKF	12.983054	27.091000	9.575006	$1.066448e-02$	0.012096	0.016953
по SQRT EKF	11.944372	24.923683	8.809018	$9.811078e-03$	0.011127	0.015596
по CMNF	4.207742	6.625001	0.7946046	$2.321271e-02$	0.029423	0.006313
по состояниям	22.002218	15.277866	$1.641047e-12$	$1.351242e-14$	0.201762	0.057276

Выборочные СКО при t=100 (с):

	X	Y	Z	v	phi	a
по EKF	9.485854	25.842247	7.876116	$7.877275e-03$	0.013892	0.011856
по SQRT EKF	8.726981	23.775082	7.246028	$7.247350e-03$	0.012782	0.010907
по CMNF	14.171700	15.830477	1.563865	$4.682883e-02$	0.071519	0.009081
по состояним	87.500077	111.775215	$1.641047e-12$	$1.351242e-14$	0.579394	0.065406

Выводы по результатам экспериментов

- Результаты работы расширенного фильтра Калмана и его корневого аналога оказались практически идентичны, так как в данной задаче, по всей видимости, не возникает проблем с неотрицательной определённостью матрицы ковариации ошибки оценки. А когда справляется обычный фильтр Калмана, тогда с аналогичной точностью справляется и его "собрат"(только с бОльшим временем вычислений).

- Фильтры справляются с поставленной задачей оценивания лучше тривиальной оценки (безусловного матожидания) для всех компонент, кроме координаты Z и скорости v, потому что по условию задачи те не меняются и околоконстантная оценка самая лучшая (подводный объект плывёт равномерно и на одной высоте). По остальным параметрам (X, Y, phi, a) преимущество особенно заметно с течением времени, когда тривиальное "усреднение"уже не позволяет обрабатывать накопившуюся ошибку, а в фильтрах шаг коррекции учитывает это. Особенно это заметно по графикам оценок нормального ускорения, которое подвергается наибольшему воздействию случайности.

- Если сравнивать результаты фильтров между собой, CMNF показал большую устойчивость по проценту расходящихся траекторий и более точную оценку координаты Z, но достаточно сильно со временем начинает ошибаться с оценкой скорости v.

Приложение 1. Код классов, реализующих фильтры Калмана

```python
from numpy.linalg import qr as qr
from numpy.linalg import pinv as pinv

def Euler_Maruyama_method(h, n, init, noise=True):
    ys = np.empty((n, init.shape[0]))
    ys[0] = init

    if noise == True:
        for i in range(n - 1):
            noise = np.array([0., 0., 0., 0., 0., np.random.normal(loc
               =0.0, scale=np.sqrt(h))])
            ys[i + 1] = ys[i] + deriv(ys[i]) * h + noise * mu
    else:
        for i in range(n - 1):
            ys[i + 1] = ys[i] + deriv(ys[i]) * h

    return ys

class EKF:
  def estimation(self, observations, n_for):
        n_dyn = int(T / h_dyn) + 1
        n = (n_dyn - n_for) // n_for

        estimation = np.empty((n_dyn, 6))
        estimation_cov = np.empty((n_dyn, 6, 6))
        estimation[0], estimation_cov[0] = initial_dynamic_mean,
           initial_dynamic_cov

        for i in range(n + 1):
            start = i * n_for
            end = start + n_for

            forecast = Euler_Maruyama_method(h_dyn, n_for + 1, estimation
               [start], False)
            forecast_cov = np.empty((n_for + 1, 6, 6))
            forecast_cov[0] = estimation_cov[start]

            for _ in range(n_for):
                j = Jacobian_m_dynamic(forecast[_])
                forecast_cov[_ + 1] = forecast_cov[_] + (j @ forecast_cov
                   [_] + forecast_cov[_] @ j.T + b @ b.T) * h_dyn

            estimation[start:end + 1], estimation_cov[start:end + 1] =
               forecast, forecast_cov

            j = Jacobian_m_observe(estimation[end])
            k = estimation_cov[end]
            gain = k @ j.T @ np.linalg.pinv(j @ k @ j.T + B @ B.T)
```

```
            estimation[end] = estimation[end] + gain @ (observations[:, i
                ] - observe(estimation[end], True))
            estimation_cov[end] = k - gain @ j @ k

        return estimation.T[:, 1:], estimation_cov.T

class SQRTEKF:
    def qr_r(self, X, Y):
      Q, R = qr(np.concatenate((X, Y), axis=0))
      return R

    def estimation(self, observations, n_for):

        n_dyn = int(T / h_dyn) + 1
        n = (n_dyn - n_for) // n_for

        estimation = np.empty((n_dyn, 6))
        estimation_cov = np.empty((n_dyn, 6, 6))
        estimation[0], estimation_cov[0] = initial_dynamic_mean, np.sqrt(
            initial_dynamic_cov)

        for i in range(n + 1):
            start = i * n_for
            end = start + n_for

            forecast = Euler_Maruyama_method(h_dyn, n_for + 1, estimation
                [start], False)
            forecast_cov = np.empty((n_for + 1, 6, 6))
            forecast_cov[0] = estimation_cov[start]

            for _ in range(n_for):
                j = Jacobian_m_dynamic(forecast[_])
                D = np.eye(6) + j * h_dyn
                forecast_cov[_ + 1] = self.qr_r(forecast_cov[_] @ (D.T),
                    b * np.sqrt(h_dyn))

            estimation[start:end + 1], estimation_cov[start:end + 1] =
                forecast, forecast_cov

            j = Jacobian_m_observe(estimation[end])
            k = estimation_cov[end]
            G = self.qr_r(k @ (j.T), B)
            gain = (pinv(G) @ (pinv(G.T) @ j) @ k.T @ k).T
            estimation[end] = estimation[end] + gain @ (observations[:, i
                ] - observe(estimation[end], True))
            estimation_cov[end] = self.qr_r(k @ (np.eye(6) - gain @ j).T
                , B @ gain.T)

        return estimation.T[:, 1:], estimation_cov.T
```

Приложение 2. Код класса, реализующего условно-минимаксный фильтр

```
class CMNF():
    def __init__(self, params, Y):
        self.params = params
        self.N_dim = params["initial_dynamic"].shape[0]
        self.observers = params["observers"]
        self.initial_mean = self.params["initial_mean"]
        self.initial_cov = self.params["initial_cov"]
        self.h_state = params["EM"]["h_state"]
        self.h_forecast = params["EM"]["h_forecast"]
        self.h_observation = params["EM"]["h_observation"]
        self.T = params["observers"]["duration"]
        self.count_observation = int(params["observers"]["duration"] /
           params["EM"]["h_observation"])
        self.count_forecast = int(params["observers"]["duration"] /
           params["EM"]["h_forecast"])
        self.count_state = int(params["observers"]["duration"] / params["
           EM"]["h_state"])
        self.b = params["b"]
        self.B = params["B"]
        self.Y = Y
        self.R = params["CMNF"]["R"]
        self.m = params["CMNF"]["m"]
        self.target = None


    def A(self, X, gaussian_error=True):
        X = X.reshape(X.shape[0], -1)
        qq = X.shape[1]
        return np.array([observe(X[:,tt], all = not gaussian_error) for
           tt in range(qq)]).T

    def a(self, X):
        return deriv(X)

    def a_ito(self, x):
        x_1 = x[0, :]
        x_2 = x[1, :]
        x_3 = x[2, :]
        x_4 = x[3, :]
        x_5 = x[4, :]
        x_6 = x[5, :]
        x_r_1 = (x_4 * np.cos(x_5)).reshape(1, -1)
        x_r_2 = (x_4 * np.sin(x_5)).reshape(1, -1)
        x_r_3 = (np.zeros(x.shape[1])).reshape(1, -1)
        x_r_4 = (np.zeros(x.shape[1])).reshape(1, -1)
        x_r_5 = (x_6 / x_4).reshape(1, -1)
        x_r_6 = (-self.params["a"]["lambda"] * x_6 + self.params["a"]["nu
           "]).reshape(1, -1)
```

```
        a_x = np.concatenate((x_r_1, x_r_2, x_r_3, x_r_4, x_r_5, x_r_6),
           axis=0)
        error = np.random.normal(loc=0.0, scale=np.sqrt(self.h_forecast),
            size=(self.N_dim, x.shape[1]))
        return a_x * self.h_forecast + x + self.b @ error

    def alpha(self, X):
        return self.a_ito(X)


    def gamma(self, X, Y, alpha=True):
        if alpha:
            alpha_t = alpha(X)
        else:
            alpha_t = X
        return Y - self.A(alpha_t)

    def Euler_Maruyama(self, x, h, count_iter):
        x_1 = x[0, :]
        x_2 = x[1, :]
        x_3 = x[2, :]
        x_4 = x[3, :]
        x_5 = x[4, :]
        x_6 = x[5, :]
        x_r_1 = (x_4 * np.cos(x_5)).reshape(1, -1)
        x_r_2 = (x_4 * np.sin(x_5)).reshape(1, -1)
        x_r_3 = (np.zeros(x.shape[1])).reshape(1, -1)
        x_r_4 = (np.zeros(x.shape[1])).reshape(1, -1)
        x_r_5 = (x_6 / x_4).reshape(1, -1)
        x_r_6 = (-self.params["a"]["lambda"] * x_6 + self.params["a"]["nu
           "]).reshape(1, -1)
        a_x = np.concatenate((x_r_1, x_r_2, x_r_3, x_r_4, x_r_5, x_r_6),
           axis=0)
        error = np.random.normal(loc=0.0, scale=np.sqrt(self.h_forecast),
            size=(self.N_dim, x.shape[1]))
        return a_x * h + x + self.b @ error

    def forecast_correction_step_synthetic(self, X_synthetic_prev,
       X_synthetic_estimate):

        X_synthetic_current = self.Euler_Maruyama(X_synthetic_prev, self.
           h_forecast, 1)
        Y_synthetic_current = self.A(X_synthetic_current)

        alpha_synthetic = self.alpha(X_synthetic_estimate)
        gamma_synthetic = self.gamma(alpha_synthetic, Y_synthetic_current
           , alpha=False)


        x_dim, a_dim, g_dim = X_synthetic_current.shape[0],
           alpha_synthetic.shape[0], gamma_synthetic.shape[0]
```

```python
    moments = np.concatenate((X_synthetic_current, alpha_synthetic,
        gamma_synthetic), axis=0)
    mk = np.sum(moments / moments.shape[1], axis=-1)
    mk_x = mk[:x_dim]
    mk_a = mk[x_dim: x_dim + a_dim]
    mk_g = mk[-g_dim:]
    Rk = self.cov_matrix(moments, mk, moments, mk)
    Rk[Rk < 0.0] = 0.0

    Rk_x_x = Rk[:x_dim, :x_dim]
    Rk_x_a = Rk[:x_dim, x_dim: x_dim + a_dim]
    Rk_a_x = Rk[x_dim: x_dim + a_dim, :x_dim]
    Rk_a_a_inv = np.linalg.pinv(Rk[x_dim: x_dim + a_dim, x_dim: x_dim
         + a_dim])
    Rk_g_a = Rk[-g_dim:, x_dim: x_dim + a_dim]
    Rk_a_g = Rk[x_dim: x_dim + a_dim, -g_dim:]
    Rk_x_g = Rk[: x_dim, -g_dim:]
    Rk_g_g = Rk[-g_dim:, -g_dim:]
    Rk_g_x = Rk[-g_dim:, : x_dim]
    tmp = (mk_x- Rk_x_a @ Rk_a_a_inv @ mk_a).reshape(alpha_synthetic.
        shape[0], -1)
    X_synthetic_modification = Rk_x_a @ Rk_a_a_inv @ alpha_synthetic
        + tmp
    tmp = Rk_g_a @ Rk_a_a_inv @ (alpha_synthetic - mk_a.reshape(
        alpha_synthetic.shape[0], 1))
    gamma_synthetic_modification = gamma_synthetic - (mk_g.reshape(
        gamma_synthetic.shape[0], 1) + tmp).reshape(gamma_synthetic.
        shape[0], -1)
    Rk_x_x_forecast = Rk_x_x - Rk_x_a @ Rk_a_a_inv @ Rk_a_x
    Rk_x_g_forecast = Rk_x_g - Rk_x_a @ Rk_a_a_inv @ Rk_a_g
    Rk_g_g_forecast = Rk_g_g - Rk_g_a @ Rk_a_a_inv @ Rk_a_g
    Rk_g_x_forecast = Rk_g_x - Rk_g_a @ Rk_a_a_inv @ Rk_a_x
    X_synthetic_correction = X_synthetic_modification +
        Rk_x_g_forecast @ np.linalg.pinv(Rk_g_g_forecast) @
        gamma_synthetic_modification
    return X_synthetic_current, X_synthetic_correction, mk, Rk


def forecast_correction_step_target(self, mk, X, Y):

    X = X.reshape(6, 1)
    alpha_target = self.alpha(X)
    x_dim, a_dim, g_dim = X.shape[0], X.shape[0], 24
    mk_x = (mk[:6]).reshape(6, 1)
    mk_a = (mk[6:12]).reshape(6, 1)
    mk_g = (mk[12:]).reshape(24, 1)
    Rk = self.R
    Rk_x_x = Rk[:x_dim, :x_dim]
    Rk_x_a = Rk[:x_dim, x_dim: x_dim + a_dim]
    Rk_a_x = Rk[x_dim: x_dim + a_dim, :x_dim]
    Rk_a_a_inv = np.linalg.pinv(Rk[x_dim: x_dim + a_dim, x_dim: x_dim
```

```
            + a_dim])
        Rk_g_a = Rk[-g_dim:, x_dim: x_dim + a_dim]
        Rk_a_g = Rk[x_dim: x_dim + a_dim, -g_dim:]
        Rk_x_g = Rk[: x_dim, -g_dim:]
        Rk_g_g = Rk[-g_dim:, -g_dim:]
        Rk_g_x = Rk[-g_dim:, : x_dim]
        Rk_x_x_forecast = Rk_x_x - Rk_x_a @ Rk_a_a_inv @ Rk_a_x
        Rk_x_g_forecast = Rk_x_g - Rk_x_a @ Rk_a_a_inv @ Rk_a_g
        Rk_g_g_forecast = Rk_g_g - Rk_g_a @ Rk_a_a_inv @ Rk_a_g
        Rk_g_x_forecast = Rk_g_x - Rk_g_a @ Rk_a_a_inv @ Rk_a_x
        X_target_modification = Rk_x_a @ Rk_a_a_inv @ alpha_target + mk_x
            - Rk_x_a @ Rk_a_a_inv @ mk_a
        if Y is None:
            return X_target_modification, Rk_x_x_forecast
        gamma_target = self.gamma(alpha_target, Y, alpha=False)
        gamma_target_modification = gamma_target - (mk_g + Rk_g_a @
           Rk_a_a_inv @ (alpha_target - mk_a))
        X_estimation_correction = X_target_modification + Rk_x_g_forecast
            @ np.linalg.pinv(Rk_g_g_forecast) @ gamma_target_modification
        Rk_k_k_estimation = Rk_x_x_forecast - Rk_x_g_forecast @ np.linalg
           .pinv(Rk_g_g_forecast) @ Rk_g_x_forecast

        return X_estimation_correction, Rk_k_k_estimation


    def create_initial(self):
        x_0_1 = np.random.normal(mu_0_1, sigma_0_1)
        x_0_2 = np.random.normal(mu_0_2, sigma_0_2)
        x_0_3 = np.random.normal(mu_0_3, sigma_0_3)
        x_0_4 = np.random.uniform(v_min, v_max)
        x_0_5 = np.random.normal(mu_0_phi, sigma_0_phi)
        x_0_6 = np.random.uniform(a_min, a_max)
        return np.array([x_0_1, x_0_2, x_0_3, x_0_4, x_0_5, x_0_6]).
           reshape(self.N_dim, )

    def cov_matrix(self, X, X_mean, Y, Y_mean):
        cov = np.zeros((X.shape[0], Y.shape[0]))
        for i in range(X.shape[1]):
            cov += (X[:, i] - X_mean).reshape(X.shape[0], 1) @ (Y[:, i] -
                Y_mean).reshape(1, Y.shape[0]) / (X.shape[1] - X.shape
               [0])
        return cov

    def synthetic(self):
        X_synthetic = np.zeros((self.N_dim, self.params["CMNF"]["count"])
           )
        for i in range(self.params["CMNF"]["count"]):
            X_synthetic[:, i] = self.create_initial()

        X_mean = np.sum(X_synthetic / X_synthetic.shape[1], axis=-1)
        self.target = X_mean
```

```
        X_synthetic_estimation = np.zeros((self.N_dim, self.params["CMNF"
            ]["count"]))
        for i in range(self.params["CMNF"]["count"]):
            X_synthetic_estimation[:, i] = X_mean
        K = self.cov_matrix(X_synthetic, X_mean, X_synthetic, X_mean)
        means = []
        R = []
        for i in tqdm.tqdm(range(self.count_forecast)):
            X_synthetic, X_synthetic_estimation, mk, Rk = self.
                forecast_correction_step_synthetic(X_synthetic,
                X_synthetic_estimation)
            means.append(mk)
            R.append(Rk)
        R = np.mean(np.array(R), axis=0)
        m = np.array(means)
        return R, m

    def estimation(self):
        Y = self.Y
        X_estimate = [self.params["initial_dynamic"]]
        k_estimate = []
        y_idx = 0
        for i in tqdm.tqdm(range(self.count_forecast)):
            if i % int(self.h_observation / self.h_forecast) == 0:
                X_target, k_target = self.forecast_correction_step_target
                    (self.m[i], X_estimate[-1], Y[:,y_idx:y_idx+1])
                y_idx += 1
            else:
                X_target, k_target = self.forecast_correction_step_target
                    (self.m[i], X_estimate[-1], None)
            for j in range(int(self.count_state / self.count_forecast)):
                X_estimate.append(X_target.reshape(6, ))
                k_estimate.append(np.diag(k_target))
        return np.array(X_estimate).T[:, 1:], np.array(k_estimate).T
```

Приложение 3. Необходимые для реализации дополнительные модули

```
def deriv(X, N=1):
    x_1, x_2, x_3, x_4, x_5, x_6 = X
    x_r_1 = x_4 * np.cos(x_5)
    x_r_2 = x_4 * np.sin(x_5)
    if N == 1:
        x_r_3 = 0
        x_r_4 = 0
    else:
        x_r_3 = [0] * N
        x_r_4 = [0] * N
    x_r_5 = x_6 / x_4
    x_r_6 = -lmbd * x_6 + v

    return np.array([x_r_1, x_r_2, x_r_3, x_r_4, x_r_5, x_r_6]).T

def observe(X, all=False):
  temp_x = np.zeros(3 * num_sensors)
  for i in range(num_sensors):
    R = np.sqrt(np.sum((X[:3] - positions[i]) ** 2))
    r = np.sqrt(np.sum((X[:2] - positions[i, :2]) ** 2))
    V_x = X[3] * np.cos(X[4])
    V_y = X[3] * np.sin(X[4])
    V = ((X[0] - positions[i, 0]) * V_x + (X[1] - positions[i, 1]) * V_y)
        / R
    temp_x[3 * i] = (X[2] - positions[i, 2]) / R # ksi
    temp_x[3 * i + 1] = (X[0] - positions[i, 0]) / r # eta
    temp_x[3 * i + 2] = w_0 / (1 - V/C) # w
    if not all:
      temp_x[3 * i] += np.random.normal(0, sigma_v_ksi)
      temp_x[3 * i + 1] += np.random.normal(0, sigma_v_eta)
      temp_x[3 * i + 2] += np.random.normal(0, sigma_v_w)
  return temp_x

def Jacobian_m_dynamic(X):
    x, y, z, v, phi, a = X

    dx_1 = np.array([0., 0., 0., np.cos(phi), -v * np.sin(phi), 0.]).
       reshape((1, -1))
    dx_2 = np.array([0., 0., 0., np.sin(phi), v * np.cos(phi), 0.]).
       reshape((1, -1))
    dx_3 = np.array([0., 0., 0., 0., 0., 0.]).reshape((1, -1))
    dx_4 = np.array([0., 0., 0., 0., 0., 0.]).reshape((1, -1))
    dx_5 = np.array([0., 0., 0., -a / (v ** 2), 0., 1 / v]).reshape((1,
       -1))
    dx_6 = np.array([0., 0., 0., 0., 0., -lmbd]).reshape((1, -1))

    return np.vstack([dx_1, dx_2, dx_3, dx_4, dx_5, dx_6])
```

```
def Jacobian_m_observe(X):

  matrix = np.zeros((3 * num_sensors, X.shape[0]))
  for i in range(num_sensors):
    R = np.sqrt(np.sum((X[:3] - positions[i]) ** 2))
    r = np.sqrt(np.sum((X[:2] - positions[i, :2]) ** 2))
    V_x = X[3] * np.cos(X[4])
    V_y = X[3] * np.sin(X[4])
    V = ((X[0] - positions[i, 0]) * V_x + (X[1] - positions[i, 1]) * V_y)
        / R
    matrix[3 * i, 0] = -(X[2] - positions[i, 2]) * (X[0] - positions[i,
       0]) / R ** 3
    matrix[3 * i, 1] = -(X[2] - positions[i, 2]) * (X[1] - positions[i,
       1]) / R ** 3
    matrix[3 * i, 2] = (R ** 2 - (X[2] - positions[i, 2]) ** 2) / R ** 3
    matrix[3 * i + 1, 0] = (r ** 2 - (X[0] - positions[i, 0]) ** 2) / r
       ** 3
    matrix[3 * i + 1, 1] = -(X[1] - positions[i, 1]) * (X[0] - positions[
       i, 0]) / r ** 3
    denom = (1 - V / C)**2
    matrix[3 * i + 2, 0] = -w_0 * (V * (X[0] - positions[i, 0]) - V_x * R
       ) / (R ** 2 * C * denom)
    matrix[3 * i + 2, 1] = -w_0 * (V * (X[1] - positions[i, 1]) - V_y * R
       ) / (R ** 2 * C * denom)
    matrix[3 * i + 2, 2] = -w_0 * (V * (X[2] - positions[i, 2])) / (R **
       2 * C * denom)
    matrix[3 * i + 2, 3] = w_0 * V / (X[3] * C * denom)
    matrix[3 * i + 2, 4] = w_0 * (V_x * (X[1] - positions[i, 1]) - V_y *
       (X[0] - positions[i, 0])) / (R * C * denom)
  return matrix

class Euler_Maruyama:
    def __init__(self, h, n, params, initial_dynamic):
        self.h = h
        self.n = n
        self.params = params
        self.N_dim = params["initial_dynamic"].shape[0]
        self.b = params["b"]

    def a(self, X):
        return deriv(X)

    def run(self):
        x = np.zeros((self.N_dim, self.n))
        x[:, 0] = self.params["initial_dynamic"]
        for i in range(self.n - 1):
            error = np.random.normal(loc=0.0, scale=np.sqrt(self.h), size
               =(self.N_dim, 1))
            b = np.diag([0, 0, 0, 0, 0, mu])
            x[:, i + 1] = ((self.a(x[:, i]) * self.h + x[:, i]).reshape(
               self.N_dim, 1) +  b @ error).reshape(self.N_dim, )
```

```
        return x

class Trivial_article:
    def __init__(self, params, Y, noise=True):
        self.count_observation = int(params["observers"]["duration"] /
            params["EM"]["h_observation"])
        self.speed = params["v"]["mean"]
        self.phi_mean = params["phi"]["mean"]
        self.T = params["observers"]["duration"]
        self.observers = params["observers"]
        self.Y = Y
        self.params = params
        self.N_dim = params["initial_dynamic"].shape[0]
        self.b = params["b"]
        self.h = params["EM"]["h_state"],
        self.n = int(params["observers"]["duration"] / params["EM"]["
            h_state"])

    def a(self, X):
        return deriv(X)

    def estimation(self):
      traj = []
      for i in range(100):
        x = np.zeros((self.N_dim, self.n))
        x[:, 0] = self.params["initial_dynamic"]
        for i in range(self.n - 1):
            error = np.random.normal(loc=0.0, scale=np.sqrt(self.h), size
               =(self.N_dim, 1))
            b = np.diag([0, 0, 0, 0, 0, mu])
            x[:, i + 1] = ((self.a(x[:, i]) * self.h + x[:, i]).reshape(
               self.N_dim, 1) +  b @ error).reshape(self.N_dim, )
        traj.append(x)
      return np.mean(np.array(y))

class Estimation:
    def __init__(self, params):
        self.params = params
        self.h_state = params["EM"]["h_state"]
        self.h_forecast = params["EM"]["h_forecast"]
        self.h_observation = params["EM"]["h_observation"]
        self.observers = params["observers"]
        self.trajectory = None
        self.trivial = None
        self.ekf = None
        self.ekf_k = None
        self.sqrtekf = None
        self.sqrtekf_k = None
        self.cmnf =  None
        self.cmnf_k = None
```

```
def A(self, X, gaussian_error=True):
    X = X.reshape(X.shape[0], -1)
    qq = X.shape[1]
    return np.array([observe(X[:,tt], all = not gaussian_error) for
       tt in range(qq)]).T

def run(self):
    self.trajectory = Trajectory(self.params).get()
    Y = self.A(self.trajectory)
    self.trivial = Trivial_article(self.params, Y).estimation()
    Y = Y[:, ::int(self.h_observation / self.h_state)]
    ekf = EKF()
    self.ekf, self.ekf_k = ekf.estimation(Y, int(self.h_observation /
        self.h_state))
    sqrtekf = SQRTEKF()
    self.sqrtekf, self.sqrtekf_k = sqrtekf.estimation(Y, int(self.
       h_observation / self.h_state))
    cmnf = CMNF(self.params, Y)
    self.cmnf, self.cmnf_k = cmnf.estimation()
```

Список литературы

[1] Borisov, A.; Bosov, A.; Miiler, B.; Miller, G. — "Passive Underwater Target Tracking: Conditionally Minimax Nonlinear Filtering with Bearing-Doppler Observations— Institute of Informatics Problems of Federal Research Center “Computer Science and Control”, 2020.

[2] Bosov, A.; Borisov, A.; Semenikhin, K. — "Conditionally Minimax Prediction in Nonlinear Stochastic Systems.— IFAC PapersOnLine, 2015.

[3] Kalman, R.E. — "A New Approach to Linear Filtering and Prediction Problems.— ASME Basic Eng, 1960.