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

Отчёт по практическому заданию

Вариант Bio_2 / CKF / PF (boot)

Выполнила:
студентка 416 группы
Гореленкова А. П.

Преподаватель:
Борисов А. В.

Москва
2025

Содержание

1 Теоретическая часть

В минимально-инвазивной робохирургии возникает проблема с фильтрацией данных о состоянии тканей: результаты должны быть достаточно точными и при этом расчёты необходимо производить в реальном времени. Joshua B. Gafford в статье 'Real-Time Parameter Estimation of Biological Tissue Using Kalman Filtering' предлагает решать эту задачу с помощью различных фильтров Калмана (Extended, Unscented and Adaptive Fading Extended). В данной работе рассматривается применение к той же модели кубатурного фильтра Калмана (Cubature Kalman Filtering, CKF) и фильтра частиц (Particle Filter, PF) с bootstrap.

1.1 Постановка задачи

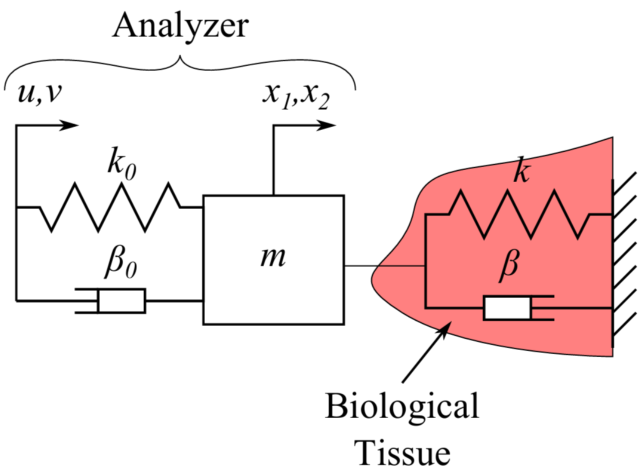

Рис. 1: Модель взаимодействия инструмент — ткань, рассмотренная статье

В качестве модели вязкопластичной ткани выбрана модель Кельвина — Фойгта (Kelvin — Voight) с параметрами упругости (stiffness) k и вязкости (damping) β, а инструмент представлен как массивная система с пружиной и амортизатором (mass-spring-damper system) с известными параметрами: массой m, коэффициентом упругости (spring constant) k_0 и коэффициентом затухания (damping coefficient) β_0. Внешнее воздействие на систему моделируется как известные сдвиг $u(t)$ и скорость $\dot{u}(t)$.

$$\begin{cases} m\ddot{x}(t) = \underbrace{k_0(u(t) - x(t)) + \beta_0(\dot{u}(t) - \dot{x}(t))}_{\text{инструмент}} + \underbrace{(-kx(t) - \beta\dot{x}(t))}_{\text{биологическая ткань}} \\ F(t) = k_0(x(t) - u(t)) \end{cases} \tag{1}$$

Координата $x(t)$ - ненаблюдаемое состояние системы, а доступные наблюдения - сила взаимодействия между инструментом и тканью $F(t)$.

1.2 Уравнение динамики и модель наблюдений

Введём следующие обозначения:

$$x_{1,t} = x_1(t) = x(t), \quad x_{2,t} = x_2(t) = \dot{x}(t)$$
$$k_t = k(t) = k = Const, \quad \beta_t = \beta(t) = \beta = Const$$
$$y_t = F(t)$$

Тогда если $X_t = [x_{1,t}, x_{2,t}, k_t, \beta_t]^T$ - расширенный вектор состояний, а $Y_t = [y_t]$ - вектор наблюдений, то систему (1) можно переписать в следующем виде:

$$\begin{cases} dX_t = F_t(X_t)dt + \hat{Q}dW_t \\ Y_t = h_t(X_t) + R\mathcal{V}_t \end{cases} \tag{2}$$

где $X_0 \sim \mathcal{N}(\hat{X}_0, \hat{P}_0)$,

$$F_t(X_t) = F_t(\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ k_t \\ \beta_t \end{bmatrix}) = \begin{bmatrix} x_{2,t} \\ \frac{1}{m}\left[k_0(u(t) - x_{1,t}) + \beta_0(\dot{u}(t) - x_{2,t}) - k_t x_{1,t} - \beta_t x_{2,t}\right] \\ 0 \\ 0 \end{bmatrix},$$

$h_t(X_t) = k_0(x_{1,t} - u(t))$,

$$\hat{Q} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

W_t - стандартный винеровский процесс,
$\mathcal{V}_t$ - последовательность н.о.р.с.в. из $\mathcal{N}(0,1)$,
$\{X_0, W_t, \mathcal{V}_t\}$ - независимы в совокупности.

Дискретизируя систему (2) методом Эйлера — Муруямы с шагом Δt, получим:

$$\begin{cases} X_t = f_t(X_{t-1}) + Q\mathcal{W}_t \\ Y_t = h_t(X_t) + R\mathcal{V}_t \end{cases} \tag{3}$$

где

$$f_t(X_{t-1}) = X_{t-1} + F_{t-1}(X_{t-1})\Delta t =$$
$$= \begin{bmatrix} x_{1,t-1} + x_{2,t-1}\Delta t \\ x_{2,t-1} + \frac{\Delta t}{m}\left[k_0(u(t-1) - x_{1,t-1}) + \beta_0(\dot{u}(t-1) - x_{2,t-1}) - k_{t-1}x_{1,t-1} - \beta_{t-1}x_{2,t-1}\right] \\ k_{t-1} \\ \beta_{t-1} \end{bmatrix},$$

$\mathcal{W}_t$ - н.о.р.с.в. из $\mathcal{N}(0, I_4)$,
$Q = \hat{Q}\sqrt{\Delta t}$.

Также в статье показано, что в качестве подающейся на вход функции $u(t)$ можно взять $A\sin(wt)$.

1.3 CKF

Шаг прогноза:

$$\breve{X}_t = \int_{R^N} a_t(x)\,\mathcal{N}(x, \bar{X}_{t-1}, \bar{k}_{t-1})\,dx \tag{4}$$

$$\breve{k}_t = \int_{R^N} a_t(x)a_t^T(x)\mathcal{N}(x, \bar{X}_{t-1}, \bar{k}_{t-1})\,dx - \breve{X}_t\breve{X}_t^T + b_t b_t^T \tag{5}$$

Шаг коррекции:

$$\breve{Y}_t = \int_{R^N} A_t(x)\,\mathcal{N}(x, \breve{X}_t, \breve{k}_t)\,dx \tag{6}$$

$$\breve{\varkappa}_t = \int_{R^N} A_t(x)A_t^T(x)\mathcal{N}(x, \breve{X}_t, \breve{k}_t)\,dx - \breve{Y}_t\breve{Y}_t^T + B_t B_t^T \tag{7}$$

$$\breve{\mu}_t = \int_{R^N} xA_t(x)\,\mathcal{N}(x, \breve{X}_t, \breve{k}_t)\,dx - \breve{X}_t\breve{Y}_t^T \tag{8}$$

$$\bar{X}_t = \breve{X}_t + \breve{\mu}_t\breve{\varkappa}_t^{-1}(Y_t - \breve{Y}_t) \tag{9}$$

$$\bar{k}_t = \breve{k}_t - \breve{\mu}_t\breve{\varkappa}_t^{-1}\breve{\mu}_t^T \tag{10}$$

Для вычисления интегралов вида

$$I = \int_{R^N} f(x)N(x, m, k)dx,$$

где $f(x)$ - некоторая гладкая функция, а $\mathcal{N}(x, m, k)$ - гауссовская квадратура (плотность гауссовского распределения со средним m и невырожденной ковариационной матрицей k), в CKF (формулы (4) - (10)) используются квадратуры Гаусса-Эрмита

$$H_n(x) = (-1)^n e^{x^2}\frac{d^n}{dx^n}e^{-x^2},$$

где $x_i, i = 1, ..., n$ - корни, а $w_i = \frac{2^{n-1}n!\sqrt(\pi)}{n^2(H_{n-1}(z_i))^2}$ - соответствующие веса.

В N-мерном случае для полиномов Гаусса - Эрмита порядка n приближённое вычисление будет выглядеть так:

$$I \approx \pi^{-N/2}\sum_{i_1=1}^{n}...\sum_{i_N=1}^{n} w_{i_1}...w_{i_N} f(k^{\frac{1}{2}}[x_{i_1}, ...x_{i_N}]^T + [m_1, ..., m_N]^T)$$

В этой задаче

- $N = 4 : [x_1, x_2, \theta_1, \theta_2]^T$
- $n = 3$
- $a_t(X) = f_t(X, u(t))$
- $A_t(X) = h_t(X, u(t))$

1.4 PF (boot)

Так как распределения компонент k и β - вырожденные (параметры константны и не меняются со временем), то просто так фильтром пользоваться не получится. Чтобы с этим бороться прибавим к данным дополнительную матрицу с шумом: $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 0.02 \end{bmatrix}$

Итого, бутстреп-фильтр имеет следующий вид:

$$\Theta(x_t|x_0, \dots, x_{t-1}; y_{\delta_3}, \dots, y_t) = \pi(x_t|x_{t-1}) \sim \mathcal{N}(f_t(x_{t-1}), QQ^T),$$

а весовые коэффициенты пересчитываются так:

$$\tilde{w}_t^{(i)} = \pi(Y_t|X_t^{(i)}) w_{t-1}^{(i)}, \quad \pi(Y_t|X_t^{(i)}) \sim \mathcal{N}(h_t(X_t^{(i)}, R^2))$$

$$w_t^{(i)} = \frac{\tilde{w}_t^{(i)}}{\sum_{j=1}^N \tilde{w}_t^{(j)}}$$

2 Расчётная часть

2.1 Значения параметров

Параметры системы (не в точности как в статье, чтобы масштаб на графиках был лучше):

$$m = 0.04\,\text{кг}, \quad k_0 = 970\,\frac{\text{Н} \times \text{м}}{\text{с}}, \quad \beta_0 = 0.4\,\frac{\text{Н}}{\text{м}}$$

Параметры внешнего воздействия $u(t) = A\sin(\omega t)$:

$$A = 0.1, \quad \omega = 30$$

Параметры начального состояния (согласно статье допустимы любые параметры для x_1 и x_2, кроме $x_1 = x_2 = 0$ - случая, когда система статична):

$$X_0 = \begin{bmatrix} 0 \\ 5 \\ 500 \\ 15 \end{bmatrix}, \quad \hat{X}_0 = \begin{bmatrix} 0 \\ 4 \\ 450 \\ 10 \end{bmatrix}, \quad \hat{P}_0 = \begin{bmatrix} 0.001^2 & 0 & 0 & 0 \\ 0 & 1^2 & 0 & 0 \\ 0 & 0 & 50^2 & 0 \\ 0 & 0 & 0 & 5^2 \end{bmatrix}$$

Параметры шума:

$$q = 0.01, \quad R = 0.5$$

Моделирование выполнено со следующими шагами по времени ($\Delta t = 0.5$ с):
$\delta_1 = \frac{\Delta t}{10^6}$ - шаг, с которым моделируется "непрерывная"траектория;
$\delta_2 = \frac{\Delta t}{10^4}$ - шаг, который применяется в алгоритмах фильтрации для прогноза;
$\delta_3 = \frac{\Delta t}{10^2}$ - шаг, с которым поступают наблюдения.

Количество частиц в PF: $N = 1000$.

2.2 Сравнение результатов расчётов

На графиках (2), (3), (4) и (5) приведены значения параметров $x(t), \dot{x}(t), k$ и β, полученные разными способами. Траектории достаточно хорошо совпадают (особенно для первых двух компонент).

Посмотрим подробнее на соответствующие графики ошибок: (6) - (13) (так как тривиальные оценки имеют много большие ошибки по сравнению с другими методами, то для каждого параметра график сначала приведён полностью, а потом в другом масштабе, чтобы были видны отличия фильтров). Как видно, наилучший результат показывает CKF, PF (boot) работает несколько хуже, а тривиальная оценка, очевидно, имеет огромную погрешность (на порядок больше остальных).

Аналогичные выводы можно сделать по графикам с выборочными ошибками (результаты получены по нерасходящимся траекториям из пучка в 10000 траекторий): (14), (15), (16) и (17).

Более строго выводы подтверждаются данными из таблицы 1. При этом можно заметить, что для параметра x_2 доля расходящихся траекторий у CKF выше.

Метод	Параметр	Доля расх. траекторий	СКО $t = 0.25$ с	СКО $t = 0.5$ с
Трив. оценка	x_1	0.01	0.00692219	0.00015079
	x_2	0.13	0.18217142	0.21481203
	k	0	50	50
	β	0	5	5
CKF	x_1	0	0.00005339	0.00003136
	x_2	0.05	0.00889350	0.00926300
	k	0	0.86446479	0.58702221
	β	0	0.04399797	0.02256929
PF (boot)	x_1	0.03	0.00039594	0.00010413
	x_2	0.01	0.02200189	0.01085255
	k	0.02	13.8392319	1.79883389
	β	0.02	0.54218990	0.19274808

Таблица 1: Сравнение методов

2.3 Выводы

Таким образом,

1. и CKF, и PF(boot) лучше тривиальной оценки;
2. у CKF лучшее качество (при условии сходимости траекторий), причём такой фильтр требует существенно меньше вычислительных ресурсов, чем PF (boot).

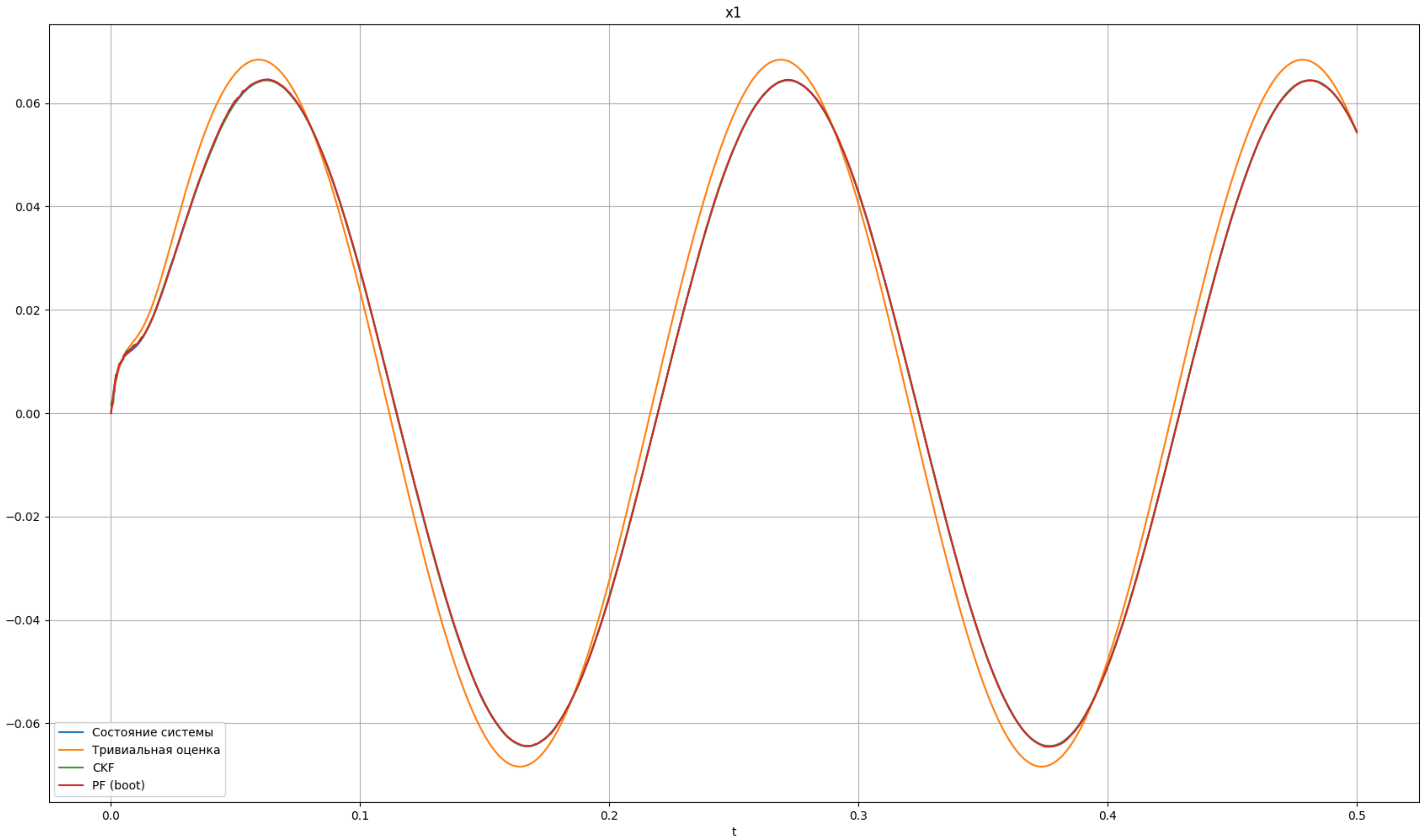

Рис. 2: Значение параметра x_1

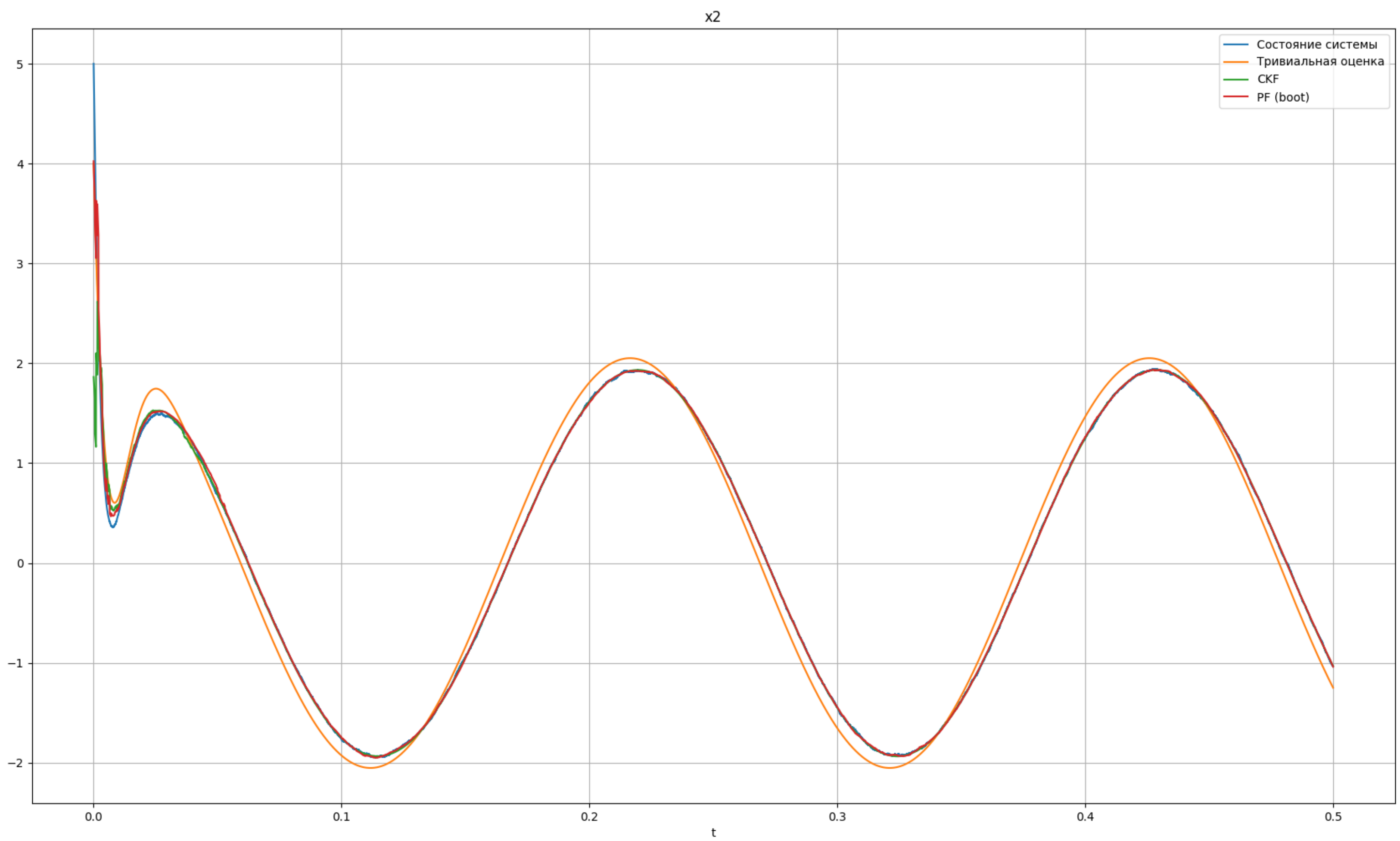

Рис. 3: Значение параметра x_2

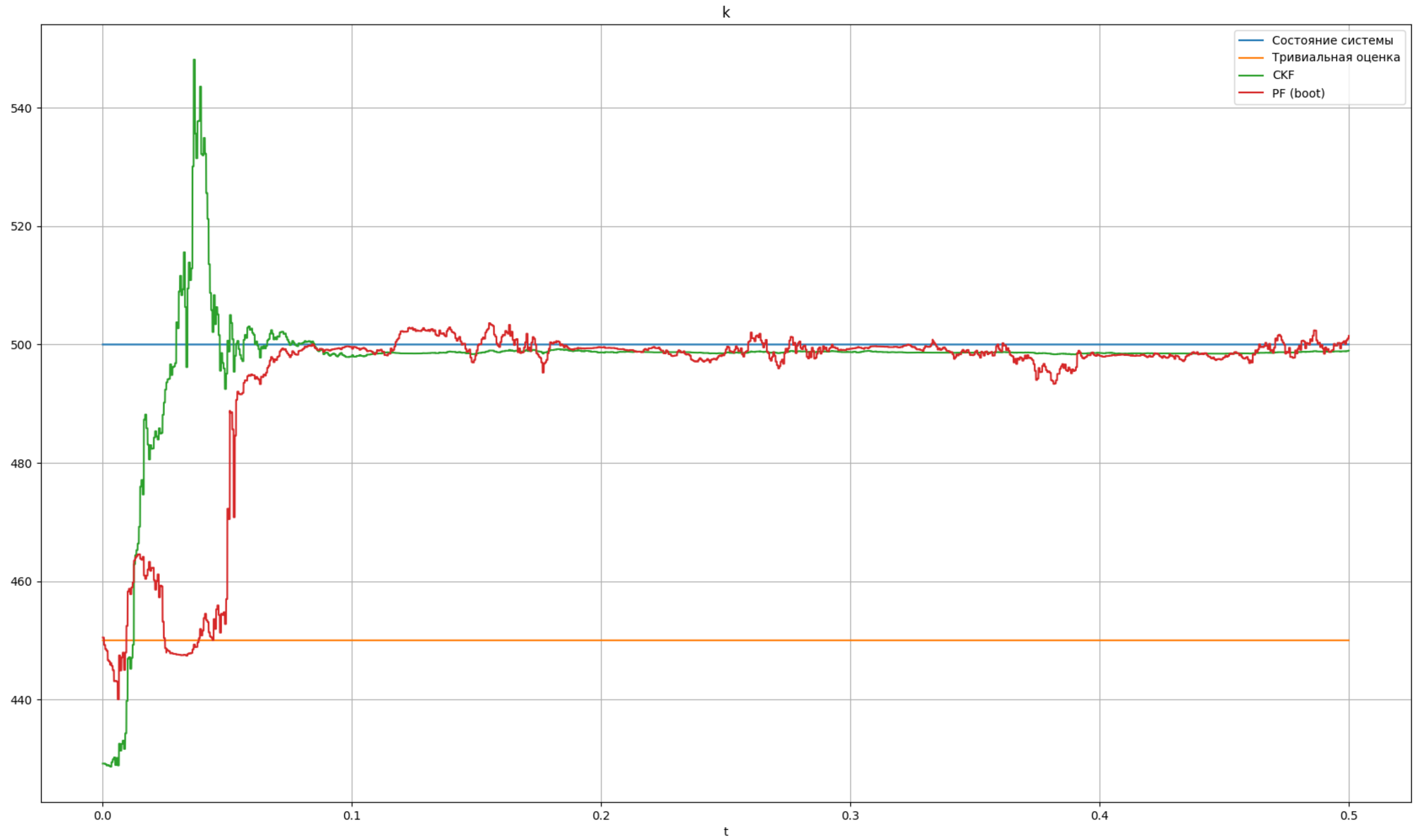

Рис. 4: Значение параметра k

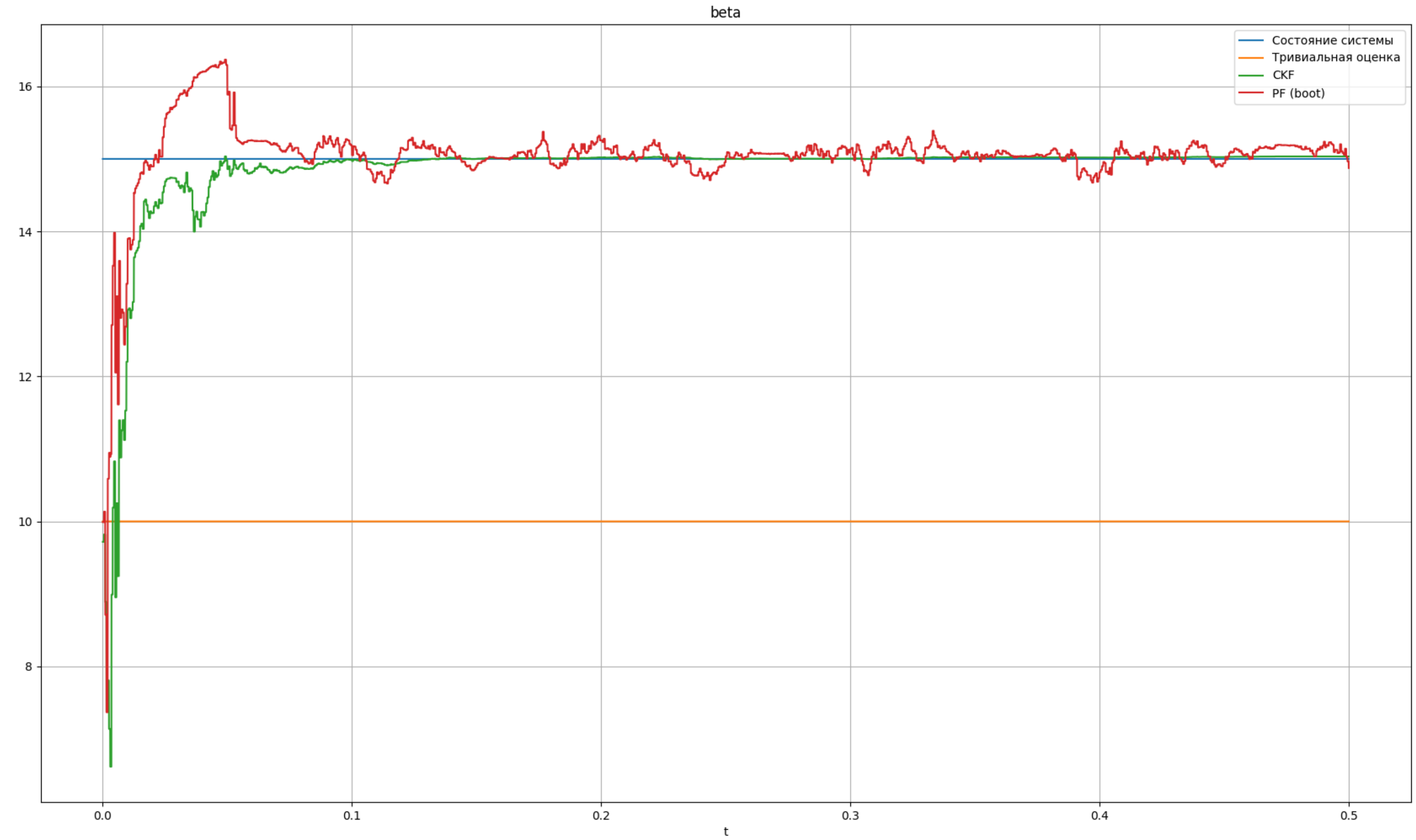

Рис. 5: Значение параметра β

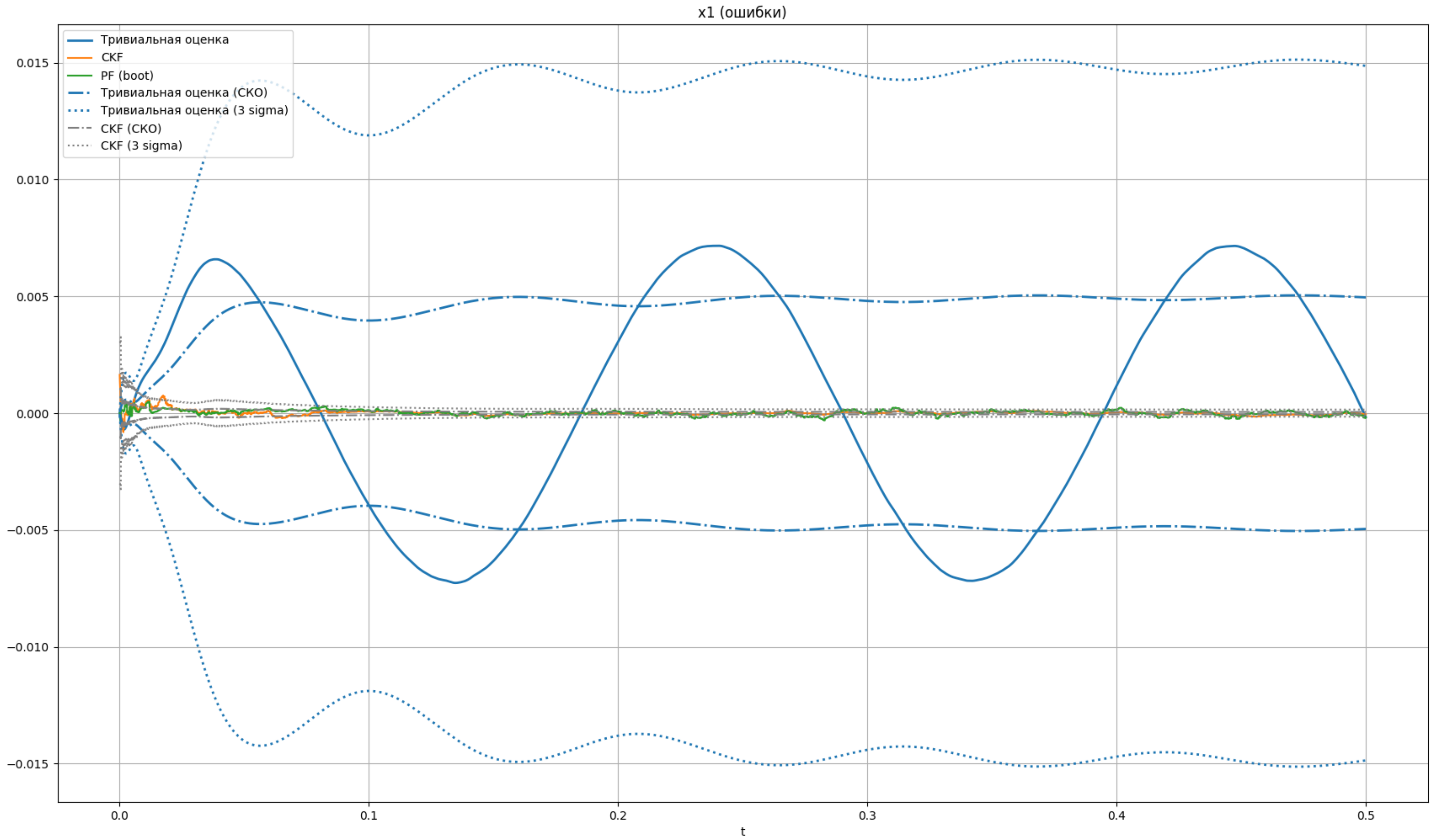

Рис. 6: Ошибки оценивания x_1

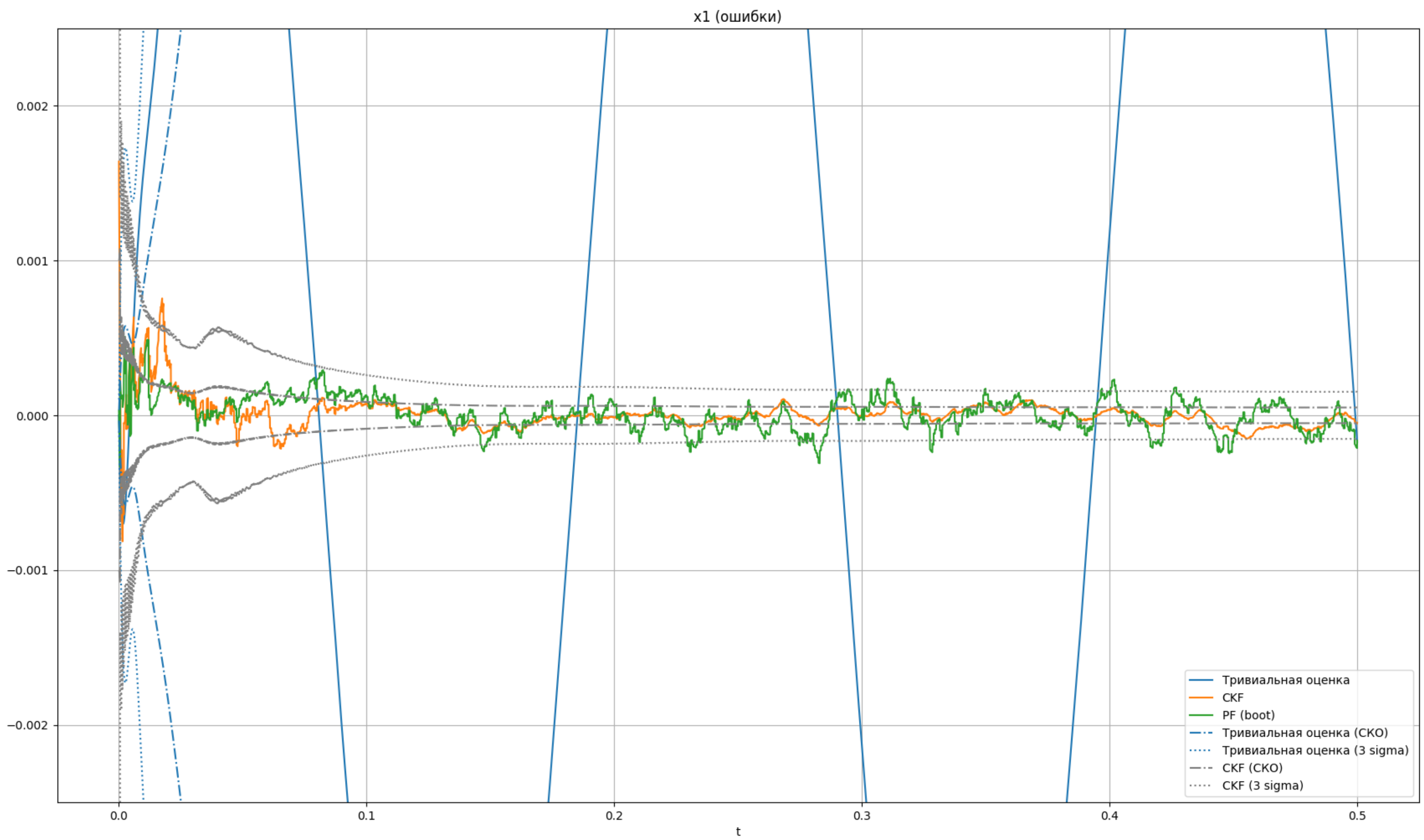

Рис. 7: Ошибки оценивания x_1 (другой масштаб)

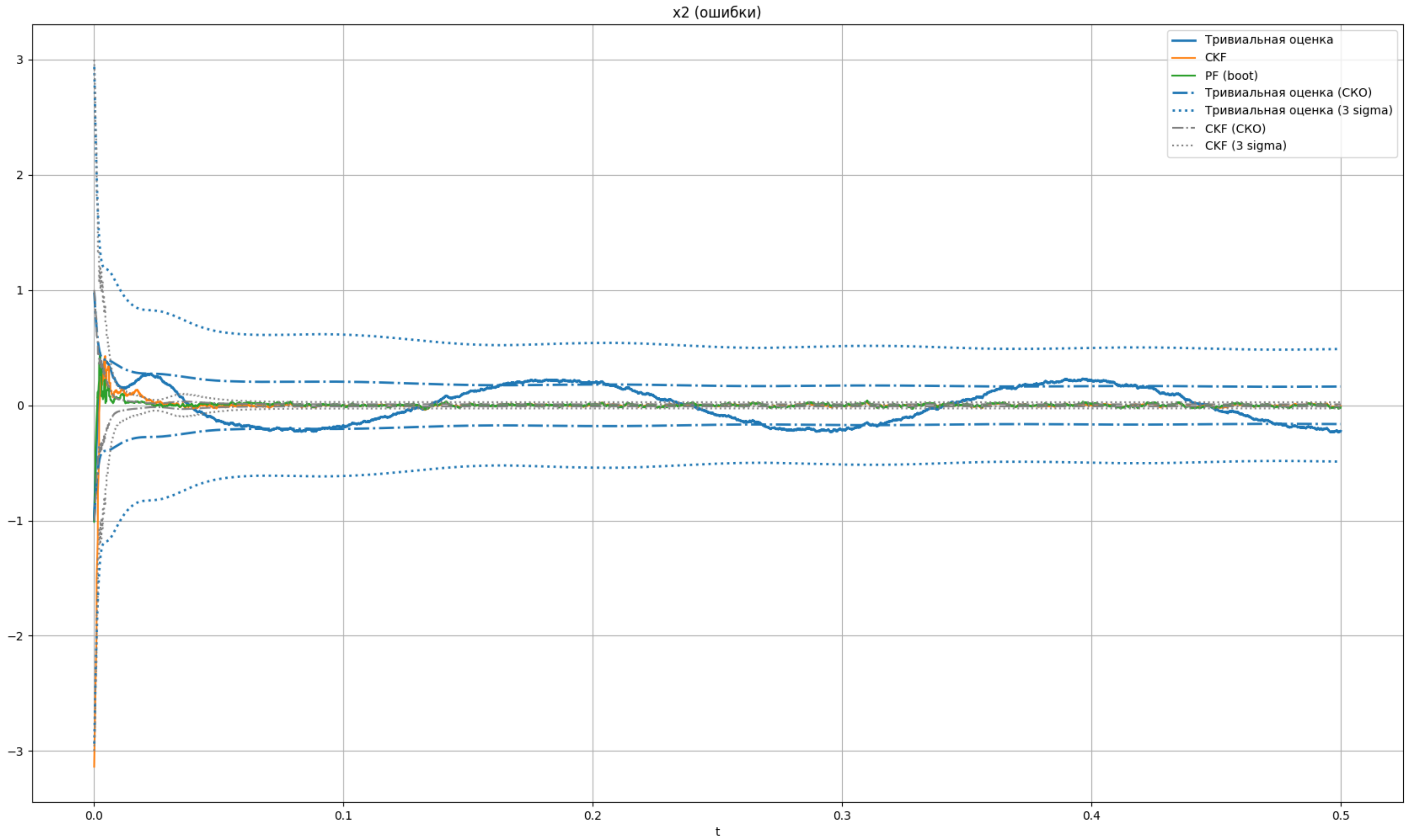

Рис. 8: Ошибки оценивания x_2

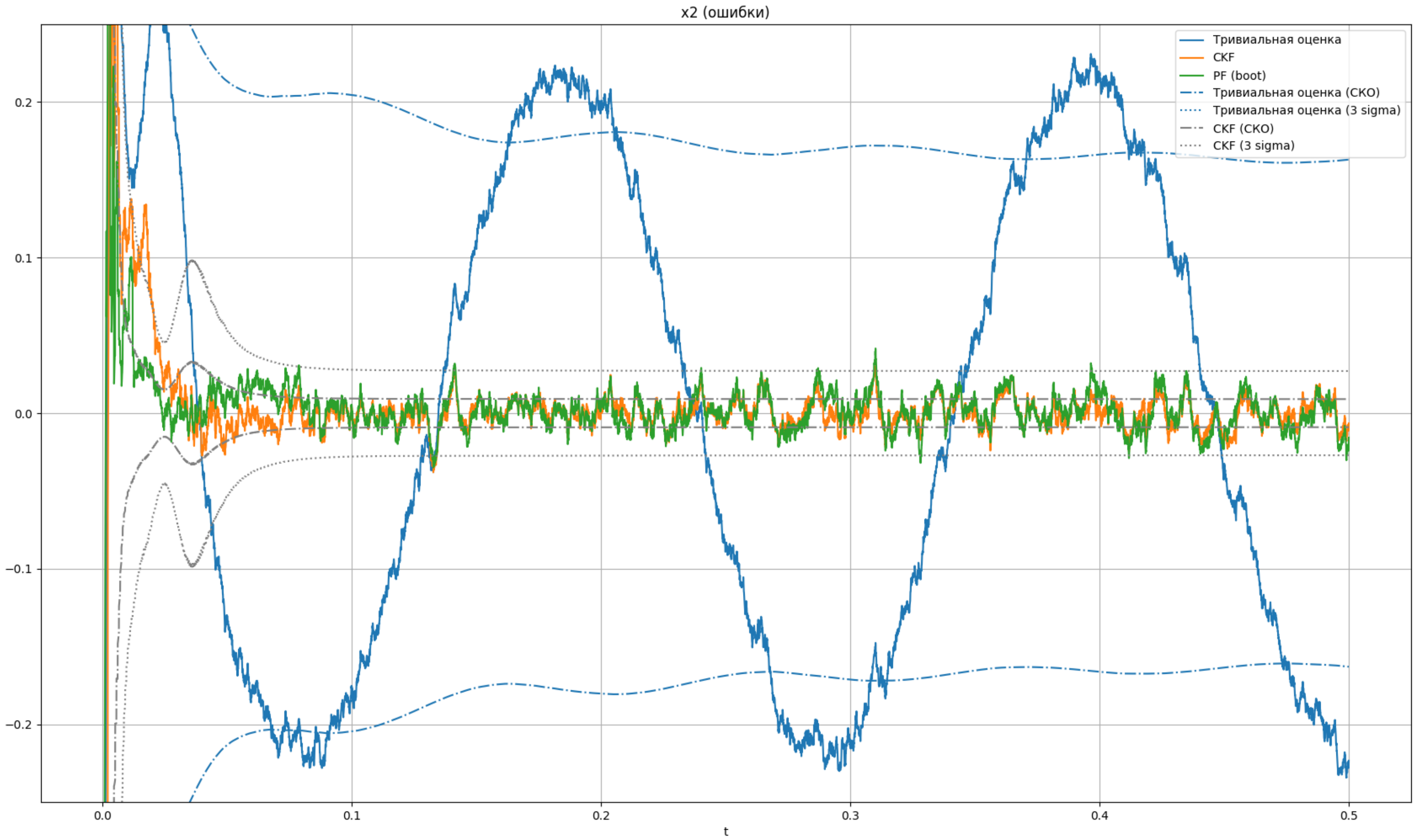

Рис. 9: Ошибки оценивания x_2 (другой масштаб)

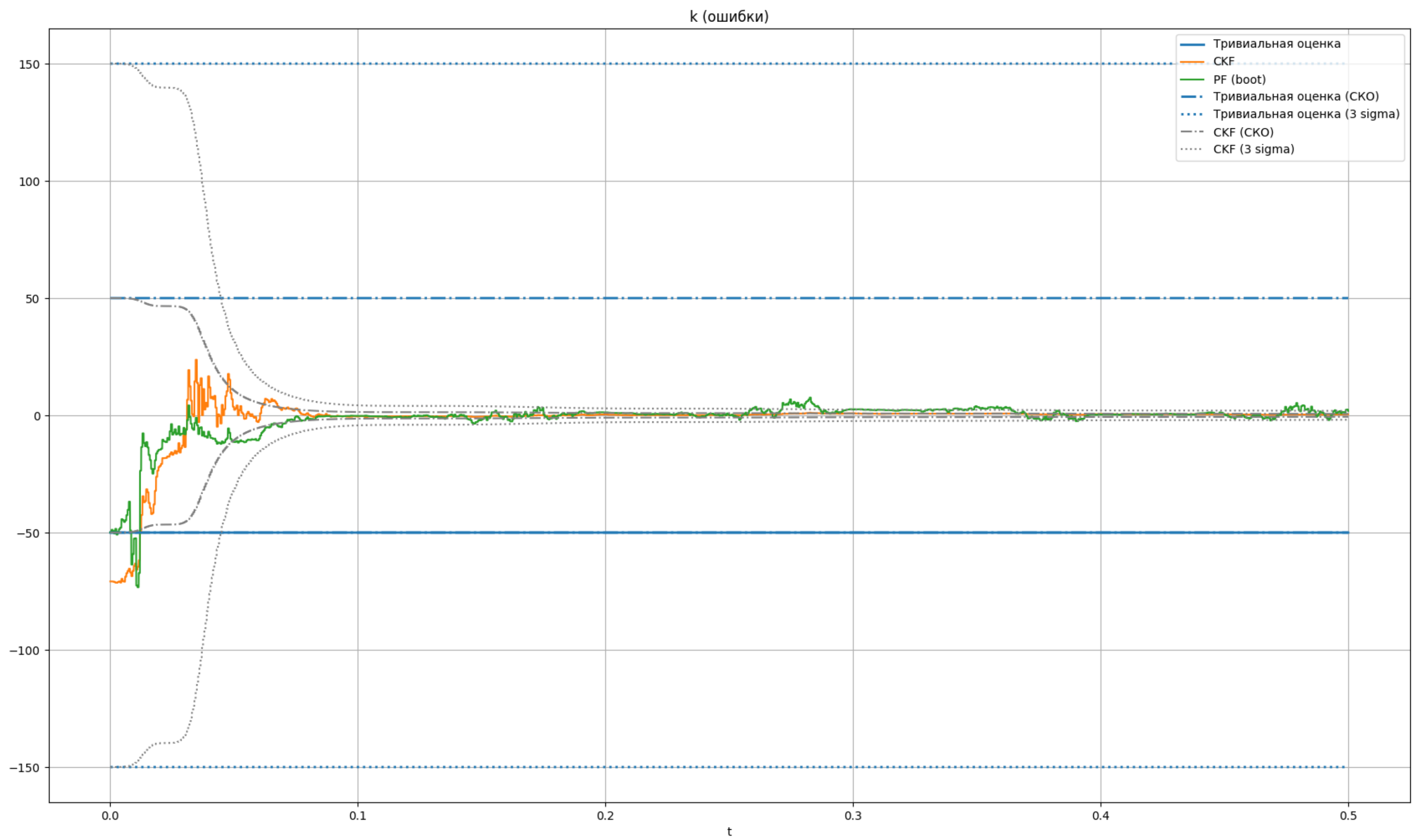

Рис. 10: Ошибки оценивания k

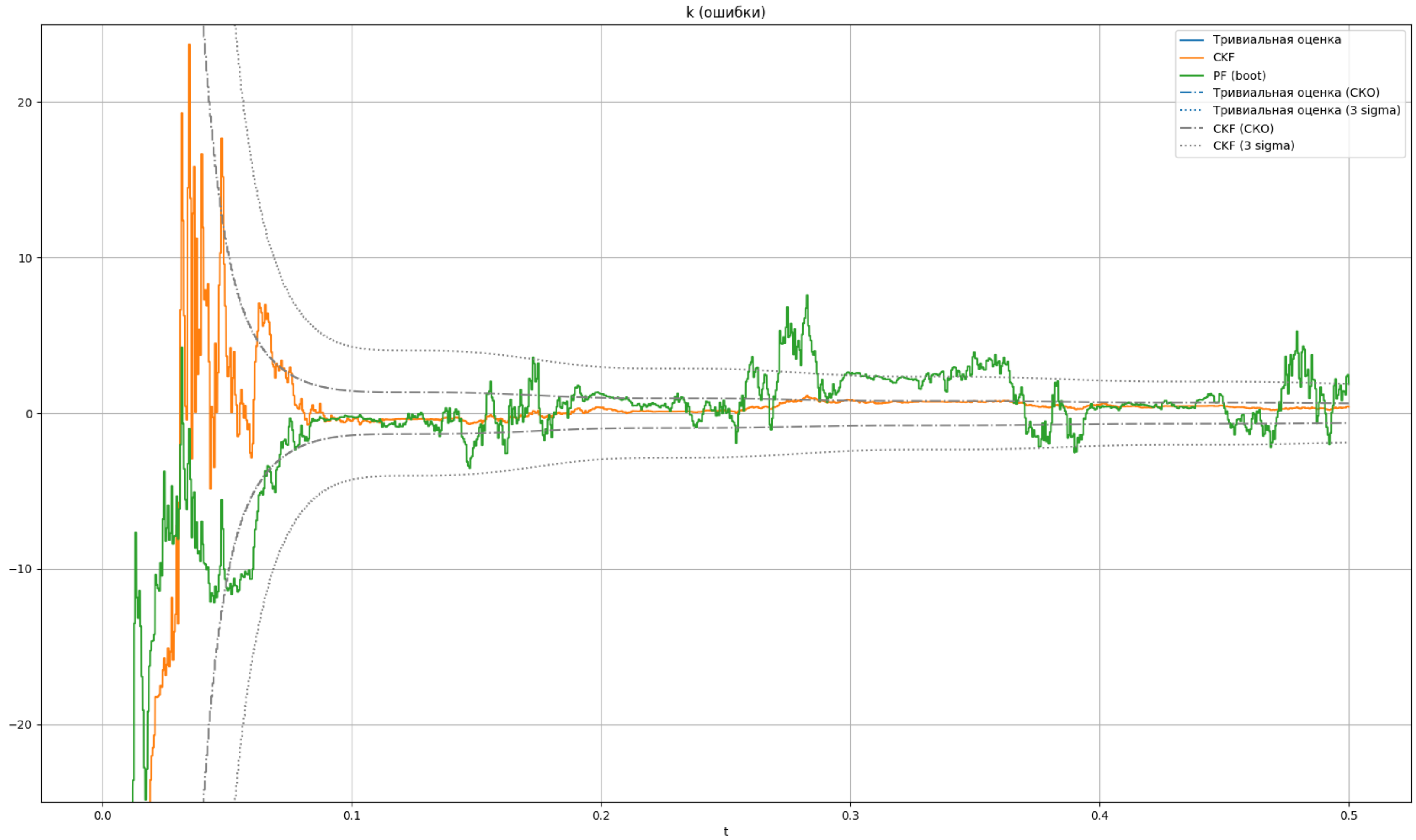

Рис. 11: Ошибки оценивания k (Другой масштаб)

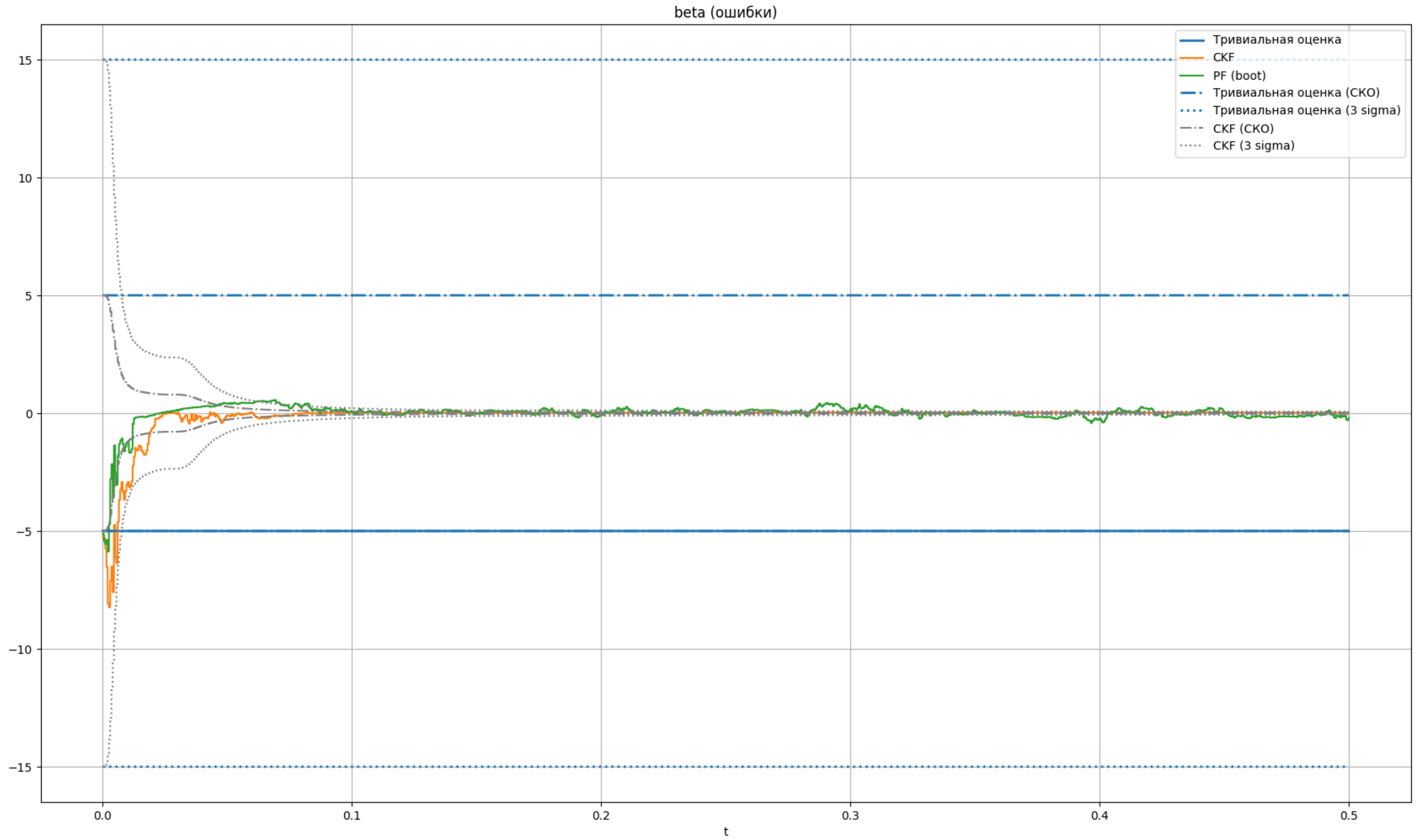

Рис. 12: Ошибки оценивания β

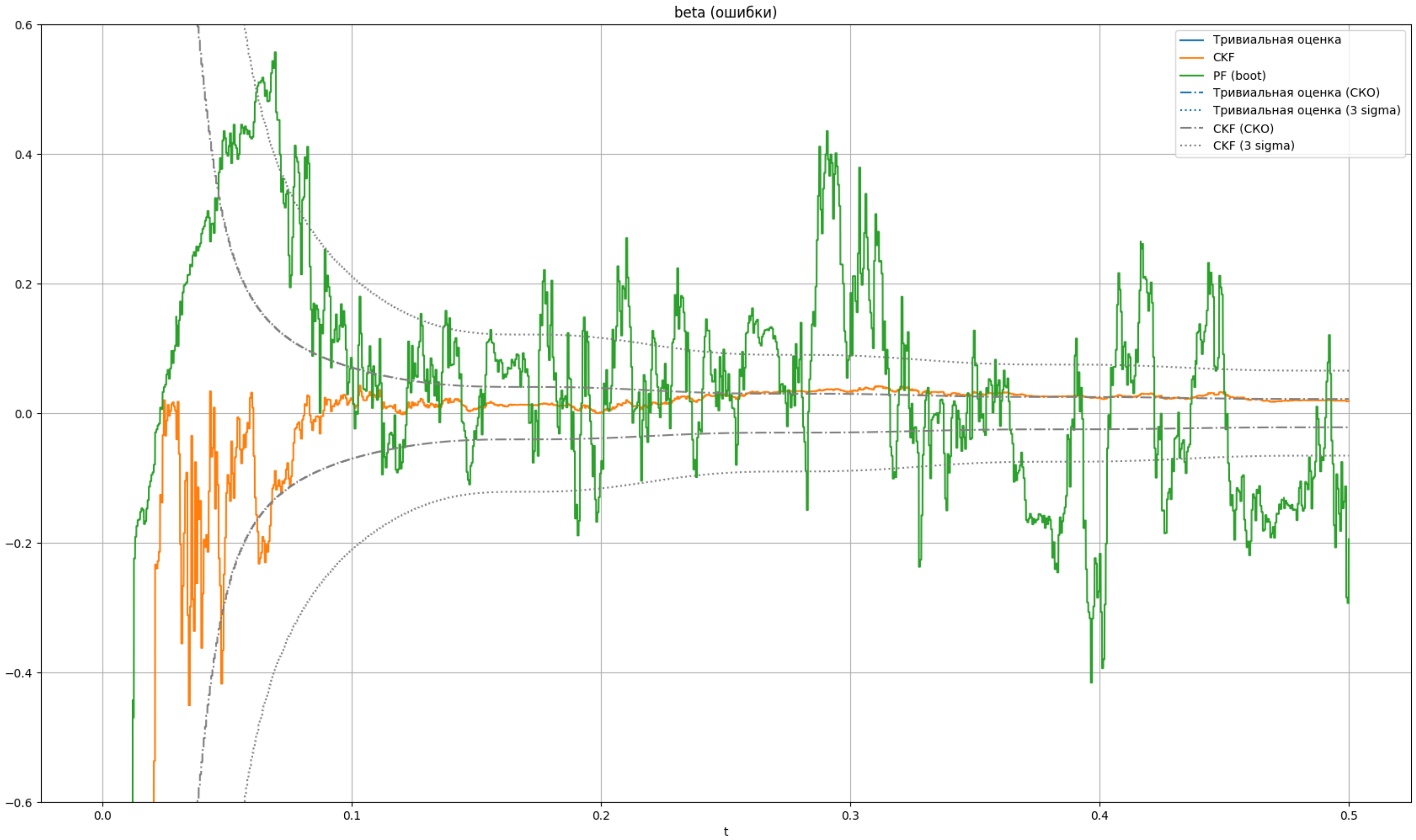

Рис. 13: Ошибки оценивания β (другой масштаб)

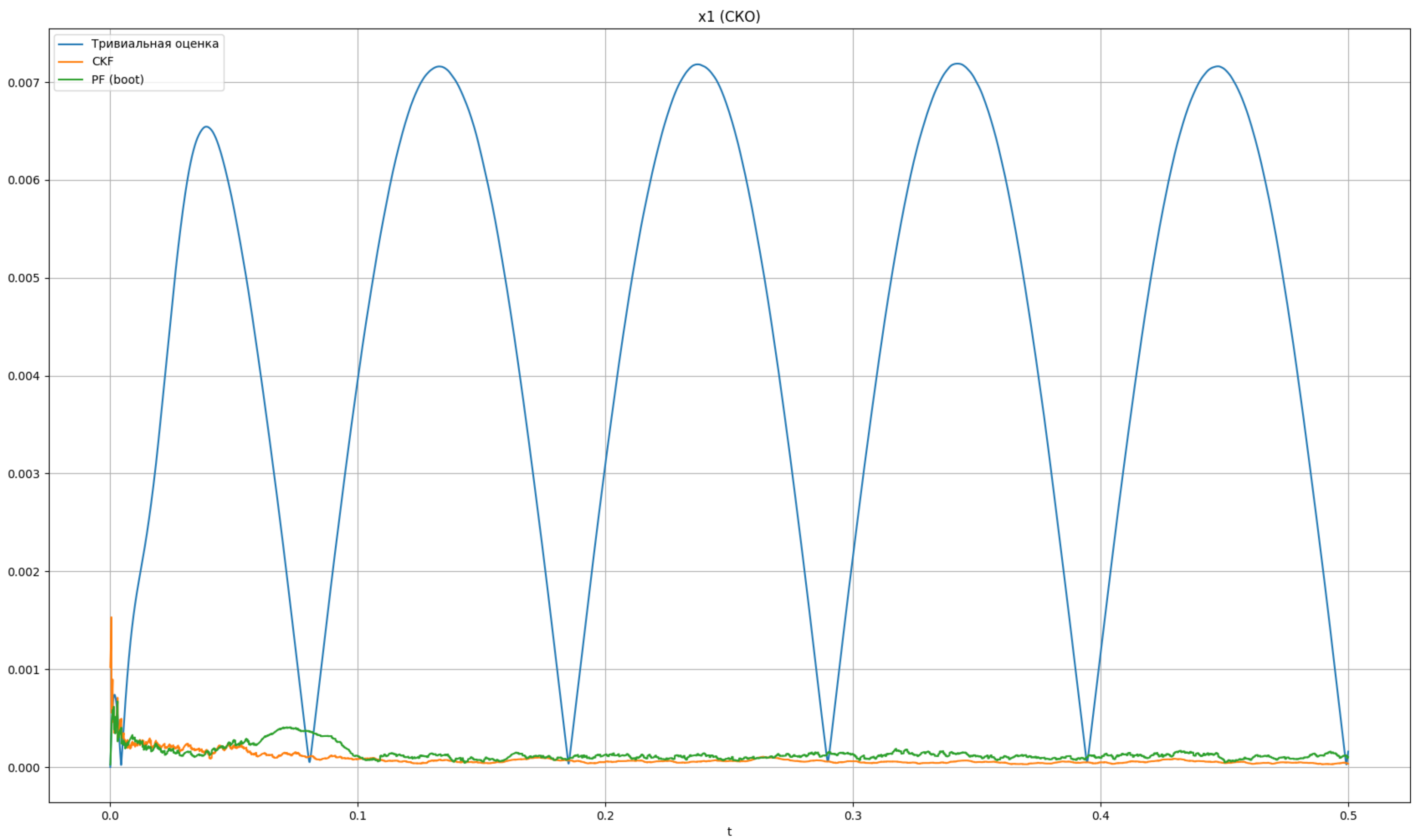

Рис. 14: Истинные выборочные СКО ошибок оценивания x_1

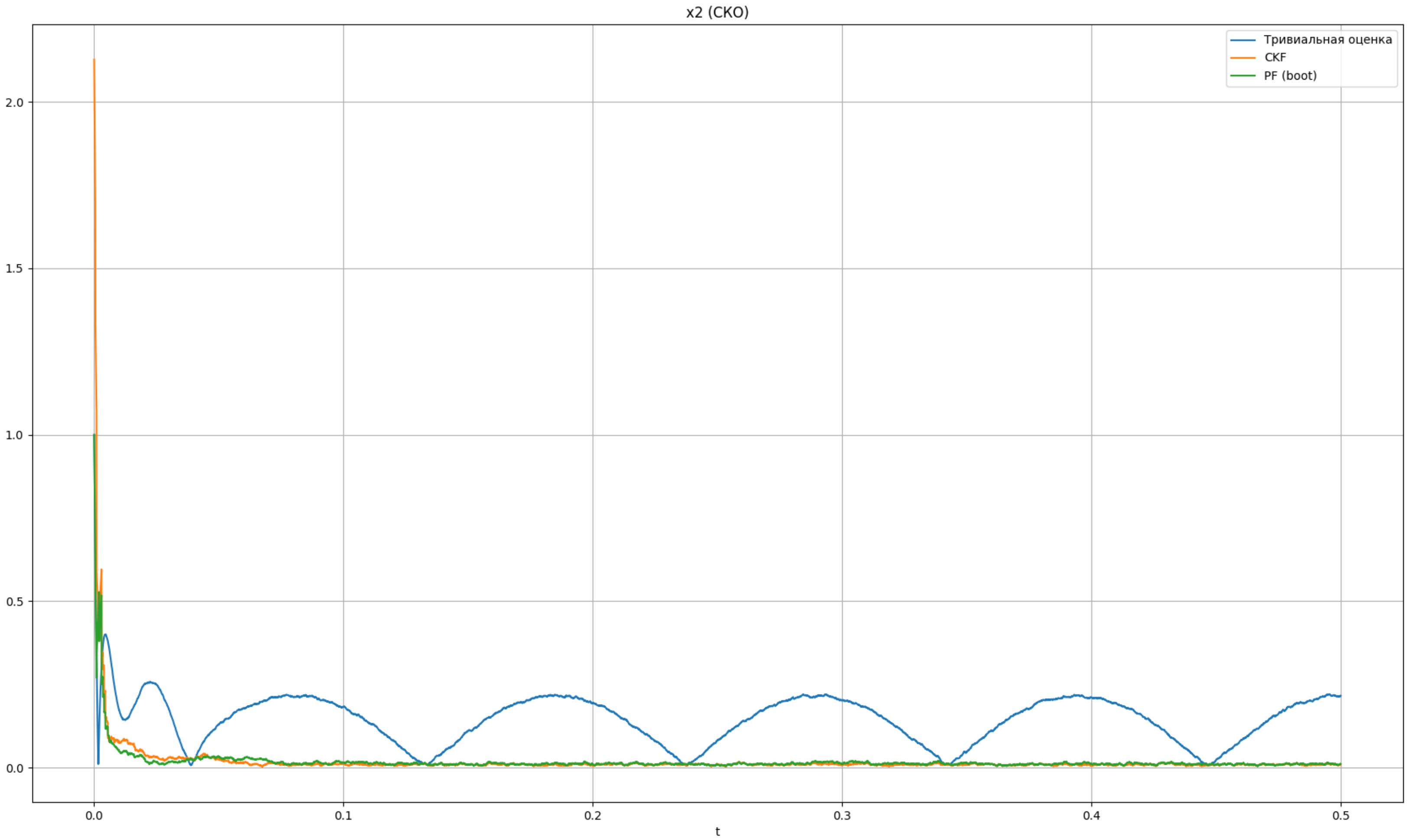

Рис. 15: Истинные выборочные СКО ошибок оцениванияя x_2

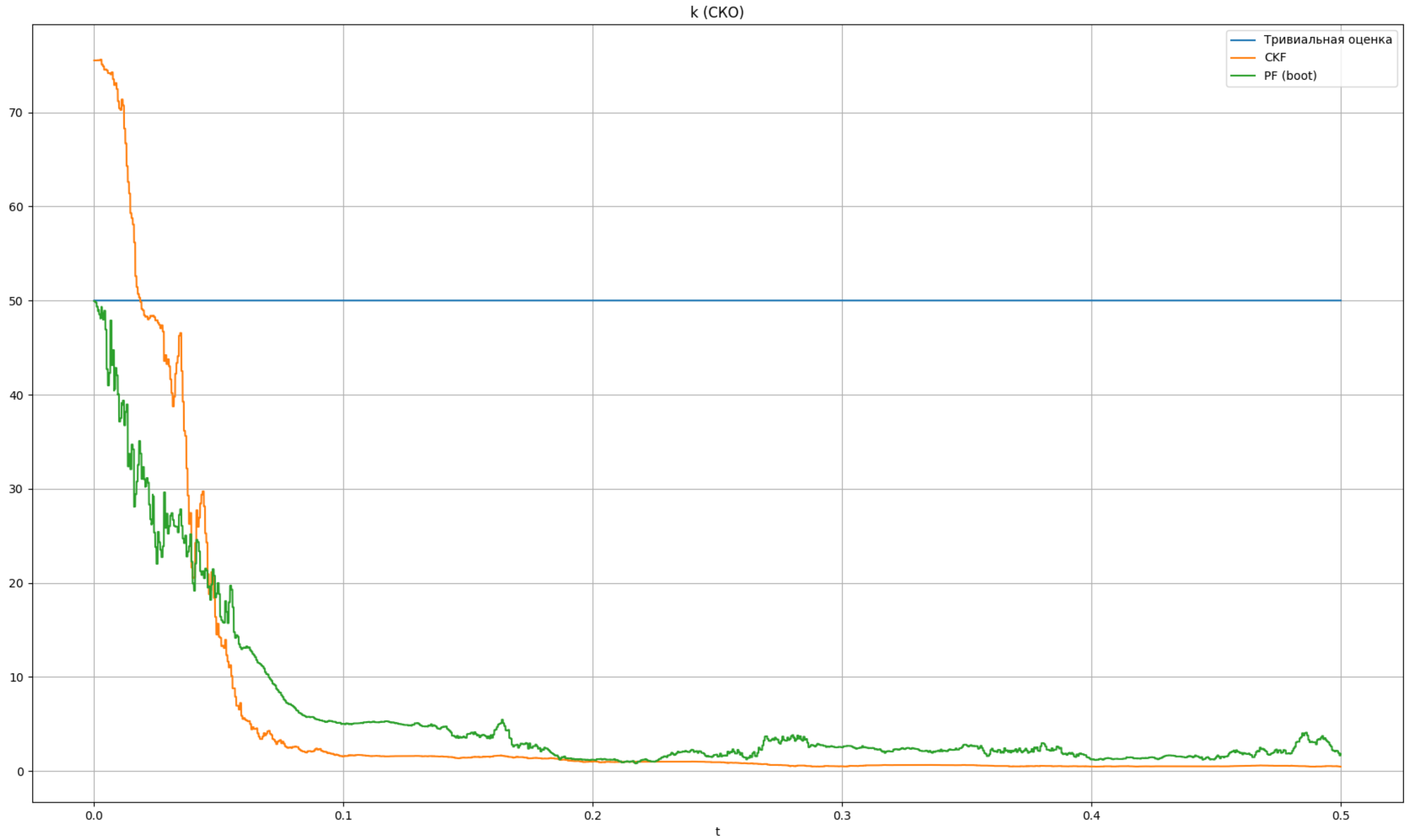

Рис. 16: Истинные выборочные СКО ошибок оценивания k

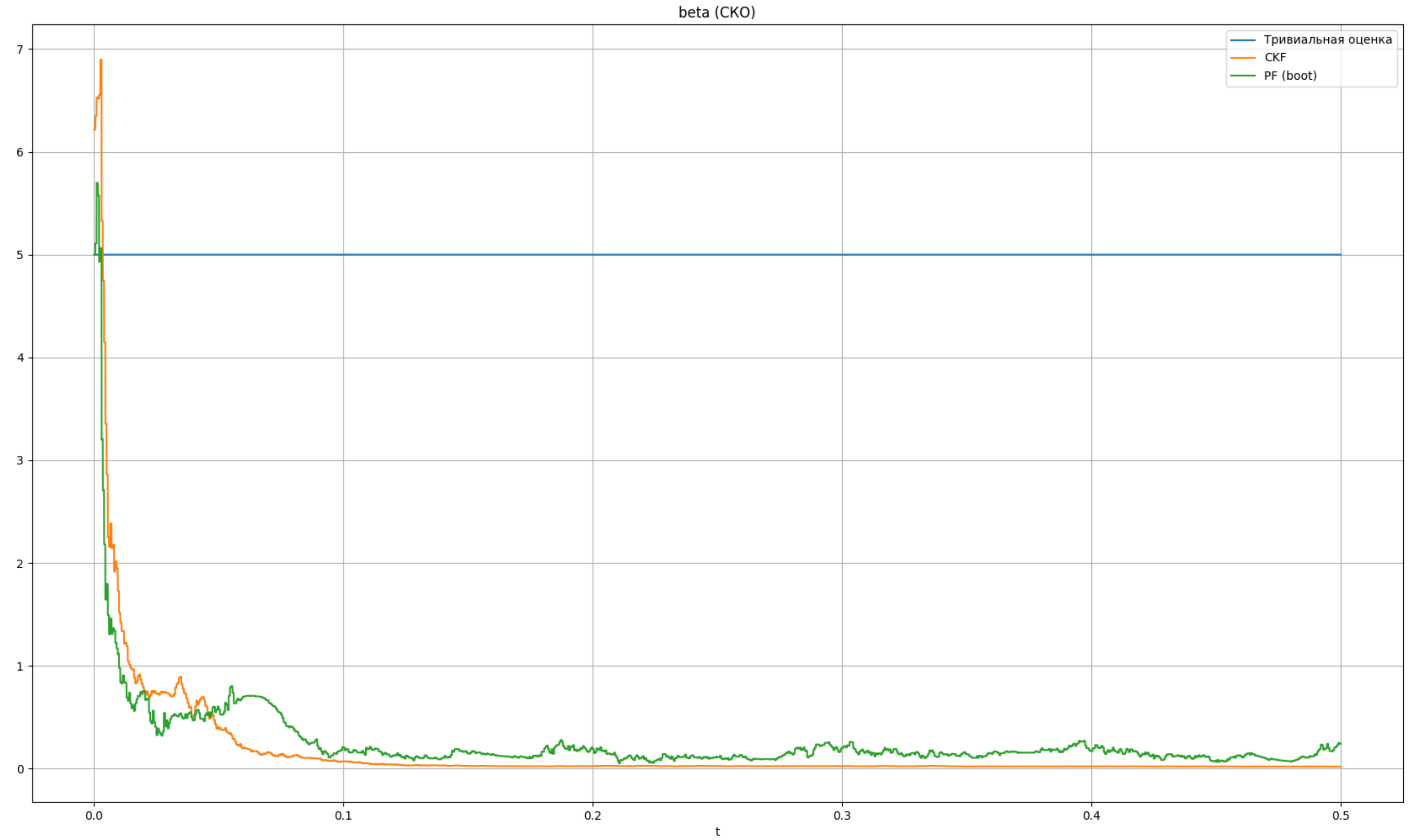

Рис. 17: Истинные выборочные СКО ошибок оценивания β

3 Приложение

3.1 Модель

```
class ToolTissueModel():
    def __init__(self, \
                 m = 0.04, k0 = 970, b0 = 0.4, \
                 u_params = (0.1, 30), \
                 T = 0.5, num_obs = 1000, pred_per_obs = 10, \
                 init_state = np.array([0, 5, 500, 15]), \
                 noise_params = (1e-2, 5e-1)):
        self.m, self.k0, self.b0 = m, k0, b0

        assert len(u_params) == 2, '2 params: A, w'
        self.A, self.w = u_params

        self.T = T
        self.num_obs = num_obs
        self.pred_per_obs = pred_per_obs
        self.cont_step = T / (num_obs * pred_per_obs ** 2)
        self.pred_step = T / (num_obs * pred_per_obs)
        self.obs_step = T / num_obs

        self.init_state = init_state

        assert len(noise_params) == 2, '2 params: Q, R'
        self.Q, self.R = noise_params

        self.cont_states = None
        self.filt_states = None
        self.observ = None

    def u(self, t):
        return self.A * np.sin(self.w * t)

    def f_func(self, x, t, time_step):
        return np.array([
            x[0] + time_step * x[1],
            x[1] + time_step / self.m * \
                (self.k0 * (self.u(t) - x[0]) + \
                 self.b0 * ((self.u(t) - self.u(t - time_step)) / \
                 time_step - x[1]) - \
                 x[2] * x[0] - x[3] * x[1]),
            x[2],
            x[3]
        ])
```

```
    def h_func(self, x, t):
        return self.k0 * (x[0] - self.u(t))

    def generate_states(self):
        states = [self.init_state]
        state_noises = stats.norm(loc=0, scale=self.Q / self.m * \
            (self.cont_step) ** 0.5).rvs(size=self.num_obs * \
            self.pred_per_obs ** 2)
        for cur_time, noise in tqdm(zip(np.linspace(0, self.T, \
            self.num_obs * self.pred_per_obs ** 2 + 1)[:-1], \
            state_noises)):
            states.append(self.f_func(states[-1], cur_time, \
                self.cont_step) + np.array([0, noise, 0, 0]))
        self.cont_states = np.array(states)
        self.filt_states = np.array(states[::int(self.pred_step / \
            self.cont_step)])

    def generate_states_for_filt(self):
        states = [self.init_state]
        state_noises = stats.norm(loc=0, scale=self.Q / self.m * \
            self.pred_step ** 0.5).rvs(size=self.num_obs * \
            self.pred_per_obs)
        for cur_time, noise in zip(np.linspace(0, self.T, \
            self.num_obs * self.pred_per_obs + 1)[:-1], \
            state_noises):
            states.append(self.f_func(states[-1], cur_time, \
                self.pred_step) + np.array([0, noise, 0, 0]))
        self.filt_states = np.array(states)

    def generate_obs(self):
        assert not (self.filt_states is None), 'No states?'
        obs_noises = stats.norm(loc=0, \
            scale=self.R).rvs(size=self.num_obs)
        obs = []
        for cur_time, noise, state in zip(np.linspace(self.obs_step, \
            self.T, self.num_obs), obs_noises, \
            self.filt_states[self.pred_per_obs::self.pred_per_obs]):
            obs.append(self.h_func(state, cur_time) + noise)
        self.observ = np.array(obs)
```

3.2 Тривиальная оценка

```
class TrivialFilter():
    def __init__(self, dyn_system, \
        init_mean = np.array([0, 4, 450, 10])):
```

```
        self.all_states = [init_mean]
        self.system = dyn_system
        self.T = self.system.T
        self.time_step = self.system.pred_step

    def step(self, cur_time):
        self.all_states.append(self.system.f_func(self.all_states[-1], \
            cur_time, self.time_step))

    def train(self):
        for cur_time in np.linspace(0, self.T, self.system.num_obs * \
            self.system.pred_per_obs + 1)[:-1]:
            self.step(cur_time)
        self.all_states = np.array(self.all_states)
```

3.3 CKF

```
def calc_int(m, k, f):
    I = 0
    n = 3
    w = [1/6, 2/3, 1/6]
    z = [-(3/2)**0.5, 0, (3/2)**0.5]
    k = sqrtm(k + 0.0001 * np.eye(4))
    for i1 in range(n):
        for i2 in range(n):
            for i3 in range(n):
                for i4 in range(n):
                    tmp = np.array(f(k @ np.array([z[i1], z[i2], z[i3], z
                    tmp *= w[i1] * w[i2] * w[i3] * w[i4]
                    I += tmp
    return I


class CubatureKalmanFilter:
    def __init__(self, dyn_system, \
                 init_st_for_filt = np.array([0, 4, 450, 10]), \
                 k_0 = np.diag(np.array([0, 0.01 / 0.5, 1.0, 1.0])) \
                 ** 0.5 * 0.03):
        self.system = dyn_system
        self.time_step = self.system.pred_step
        x0 = init_st_for_filt

        self.fit(x0, k_0)

    def a(self, x):
        time_step = self.time_step
```

```
        m = self.system.m
        k0 = self.system.k0
        b0 = self.system.b0
        t = self.cur_time
        u = self.system.u(t)
        u_prev = self.system.u(t - time_step)

        ax = np.zeros(4)
        ax[0] = x[0] + time_step * x[1]
        ax[1] = x[1] + time_step / m * (k0 * (u - x[0]) + b0 * \
        ((u - u_prev) / time_step - x[1]) - x[2] * x[0] - x[3] * x[1])
        ax[2] = x[2]
        ax[3] = x[3]
        return ax

    def A(self, x):
        k0 = self.system.k0
        t = self.cur_time
        u = self.system.u(t)
        return k0 * (x[0] - u)

    def fit(self, x0, k_0):
        s = self.system.pred_per_obs
        k_tot = self.system.observ.shape[0] *10

        x_wide = np.zeros((k_tot + 1, 4))
        y_wide = np.zeros(k_tot + 1)
        x_line = np.zeros((k_tot + 1, 4))
        k_line_diag = np.zeros((k_tot + 1, 4))

        x_wide[0] = x0
        x_line[0] = x0
        X_line = x0
        k_line = k_0

        b = np.diag(np.array([0, 0.01 / 0.5, 1.0, 1.0])) ** 0.5 * 0.03
        B = np.diag([self.system.R])**0.5 * 0.03

        i=1

        for ind, cur_time in enumerate(np.linspace(self.time_step, \
            self.system.T, self.system.num_obs * \
            self.system.pred_per_obs)):
            k = ind + 1

            self.cur_time = cur_time
```

```
X_wide = np.array(calc_int(f = self.a, m = X_line, \
    k=k_line))
x_wide[ind] = X_wide

def aat(x):
    self.cur_time = cur_time
    tmp = np.array(self.a(x) - X_wide).reshape((-1, 1))
    return tmp @ tmp.T

k_wide = calc_int(f = aat, m = X_line, k = k_line) + \
    b @ b.T

self.cur_time = cur_time
Y_wide = calc_int(f = self.A, m = X_wide, k = k_wide)
y_wide[k] = Y_wide

def AAt2(x):
    self.cur_time = cur_time
    tmp = np.array(self.A(x) - Y_wide).reshape((-1, 1))
    return tmp @ tmp.T

def xA(x):
    tmp1 = (x - X_wide).reshape((4, 1))
    self.cur_time = cur_time
    tmp2 = (self.A(x) - Y_wide).reshape((-1, 1))
    return tmp1 @ tmp2

if k % s == 0:
    i += 1

    kappa_wide = np.array(calc_int(f = AAt2, m = X_wide, \
        k = k_wide)  + B @ B.T)
    mu_wide = np.array(calc_int(f = xA, m = X_wide, \
        k = k_wide))

    X_line = X_wide + (mu_wide @ \
        np.linalg.pinv(kappa_wide) @ \
        np.array(self.system.observ[ind // s] - \
        Y_wide).reshape(1))
    k_line = k_wide - mu_wide @ \
        np.linalg.pinv(kappa_wide) @ mu_wide.T

    x_line[i] = X_line
    k_line_diag[i] = np.diag(k_line)
else:
    X_line = X_wide
```

```
                k_line = k_wide

        print(i)

        self.x_line = x_line
        self.x_wide = x_wide

        self.y_wide = y_wide
        self.k_line = k_line_diag
```

3.4 PF (boot)

```
class ParticleFilter_boot():
    def __init__(self, dyn_system, \
                 init_mean = np.array([0, 4, 450, 10]), \
                 init_cov = np.diag([0.001, 1, 50, 5]) ** 2, \
                 n_particles = 1000):
        self.system = dyn_system
        self.time_step = self.system.pred_step
        self.state_noise_matr = np.diag([self.system.Q / \
            self.system.m * self.time_step ** 0.5, init_cov[2, 2] ** \
            0.5 / 100, init_cov[3, 3] ** 0.5 / 100])
        self.all_obs = self.system.observ
        self.n_particles = n_particles
        self.weights = np.repeat(1 / n_particles, n_particles)
        self.particles = stats.multivariate_normal(mean=init_mean, \
            cov=init_cov).rvs(size=self.n_particles)
        self.all_states = [np.sum(self.weights[:, np.newaxis] * \
            self.particles, axis=0)]

    def generate_particles(self, cur_time):
        particles_mean = np.apply_along_axis(self.system.f_func, \
            axis=1, arr=self.particles, t=cur_time, \
            time_step=self.time_step)
        particles_noises = np.hstack((np.zeros(shape=(self.n_particles, 1
            stats.multivariate_normal(mean=np.zeros\
            (shape=(self.state_noise_matr.shape[0], )), \
            cov=self.state_noise_matr ** 2).rvs(size=self.n_particles)))
        self.particles = particles_mean + particles_noises

    def update_weights(self, cur_obs, cur_time):
        obs_func_res = np.apply_along_axis(self.system.h_func, axis=1, \
            arr=self.particles, t=cur_time)
        self.weights = stats.norm(loc=cur_obs, \
            scale=self.system.R).pdf(obs_func_res) * \
            self.weights
```

```
        self.weights /= np.sum(self.weights)

    def check_eff(self):
        return 1 / np.sum(self.weights ** 2) > self.n_particles / 10

    def particles_step(self, cur_time):
        self.particles = np.apply_along_axis(self.system.f_func, \
            axis=1, arr=self.particles, t=cur_time, \
            time_step=self.time_step)
        self.all_states.append(np.sum(self.weights[:, np.newaxis] * \
            self.particles, axis=0))

    def correct(self, cur_obs, cur_time):
        self.generate_particles(cur_time - self.time_step)
        self.update_weights(cur_obs, cur_time)
        self.all_states.append(np.sum(self.weights[:, np.newaxis] * \
            self.particles, axis=0))
        if not self.check_eff():
            self.particles = \
                self.particles[np.random.choice(a=self.n_particles, \
                size=self.n_particles, p=self.weights)]
            self.weights = np.repeat(1 / self.n_particles, \
                self.n_particles)

    def step(self, cur_time, ind):
        if ind % self.system.pred_per_obs:
            self.particles_step(cur_time - self.time_step)
        else:
            self.correct(self.all_obs[ind // \
                self.system.pred_per_obs - 1], cur_time)

    def train(self):
        for ind, cur_time in enumerate(np.linspace(self.time_step, \
            self.system.T, self.system.num_obs * \
            self.system.pred_per_obs)):
            self.step(cur_time, ind + 1)
        self.all_states = np.array(self.all_states)
```