Робастное управление с возмущениями

1. Теоретический анализ

1.1 Система

Манипулятор может быть описан уравнением:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + D\dot{q} + F_c(\dot{q}) = u \tag{1.1.1}$$

$$M(q)\ddot{q} + (C(q,\dot{q}) + D)\dot{q} + g(q) + F_c(\dot{q}) = u \tag{1.1.2}$$

где:

- $M(q)$ - неопределенная (для нее известна только примерная оценка) матрица инерции системы
- $C(q,\dot{q})$ - неопределенная матрица центробежных и кориолисовых сил
- $g(q)$ - неопределенный вектор сил гравитации
- D - неизвестная диагональная матрица вязкого трения
- F_c - неизвестный вектор сил кулоновского трения
- u - наше управление моментами (силами)

1.2 Линеаризация системы

Может быть проведена посредством использования управления в виде с использованием оценок параметров системы:

$$u = \hat{M}(q)v + (\hat{C}(q,\dot{q}) + \hat{D})\dot{q} + \hat{g}(q) + \hat{F}_c(\dot{q}), \tag{1.2.1}$$

где значения с шапкой - оценочные значения для реальных параметров системы, используемые для компенсации нелинейности системы, а v - непосредственное управление.

Если поаставить такое управление в уравнение динамики, $\ddot{q}$ может быть выражено как:

$$\ddot{q} = M^{-1}(q) \cdot (\widetilde{C}(q,\dot{q})\dot{q} + \widetilde{D}\dot{q} + \tilde{g}(q) + \widetilde{F_c}(\dot{q})) + M^{-1}(q)\hat{M}(q)v \tag{1.2.2}$$

Которое можно свернуть в:

$$\ddot{q} = f(q,\dot{q}) + B(q)v \tag{1.2.3}$$

где:

$$f(q,\dot{q}) = M^{-1}(q) \cdot (\widetilde{C}(q,\dot{q})\dot{q} + \widetilde{D}\dot{q} + \tilde{g}(q) + \widetilde{F_c}(\dot{q})) \tag{1.2.4}$$

- $\widetilde{C} = C - \hat{C}$
- $\widetilde{D} = D - \hat{D}$
- $\tilde{g} = g - \hat{g}$
- $\widetilde{F_c} = F_c - \hat{F}_c$
  и

$$B(q)v = M^{-1}(q)\hat{M}(q)v \tag{1.2.5}$$

1.3 Sliding surface

Задается уравнением:

$$S = \dot{\tilde{q}} + \lambda\tilde{q}, \tag{1.3.1}$$

где:

- $\tilde{q} = q_{target} - q$ - ошибка по позиции суставов
- $\dot{\tilde{q}} = \dot{q}_{target} - \dot{q}$ - ошибка по скорости суставов
- λ - матрица коэффициэнтов

Здесь, в качестве кандидата Ляпунова можно рассмотреть функцию:

$$V = \frac{1}{2}\|S\|^2 \tag{1.3.2}$$

Преобразуем функцию, подставив в нее выражение (1.2.3):

$$\dot{V} = S^T\dot{S} = S^T(\ddot{\tilde{q}} + \lambda\dot{\tilde{q}}) = S^T(\ddot{q}_{target} - \ddot{q} + \lambda\dot{\tilde{q}}) = S^T(\ddot{q}_{target} - f(q,\dot{q}) - B(q)v + \lambda\dot{\tilde{q}}) \tag{1.3.3}$$

Нам нужно чтобы $\dot{V} < 0$, этого можно достичь, если:

$$\ddot{q}_{target} - f(q,\dot{q}) - B(q)v + \lambda\dot{\tilde{q}} = -k\frac{S}{\|S\|} \tag{1.3.4}$$

Поскольку в этом случае:

$$\dot{V} = S^T\dot{S} = S^T(\ddot{q}_{target} - f(q,\dot{q}) - B(q)v + \lambda\dot{\tilde{q}}) = -k\frac{S^TS}{\|S\|} = -k\frac{\|S\|^2}{\|S\|} = -k\|S\| < 0 \tag{1.3.5}$$

Таким образом получаем, что система стабильна по Ляпунову, поскольку нашелся подходящий кандидат.

Преобразуя, получим:

$$B(q)v = \ddot{q}_{target} - f(q,\dot{q}) + k\frac{S}{\|S\|} + \lambda\dot{\tilde{q}} \tag{1.3.6}$$

Исходя из предроложения, что оценочные значения близки к истинным можно сказать, что значение $f(q,\dot{q}) \to 0$, а матрица $B(q) \to I$

Таким образом уравнение (1.3.6) упростится до:

$$v = \ddot{q}_{target} + k\frac{S}{\|S\|} + \lambda\dot{\tilde{q}} \tag{1.3.7}$$

Управление v может быть разложено на две составляющие:

$$v = v_n + v_s, \tag{1.3.8}$$

где:

- $v_n = \ddot{q}_{target} + \lambda\dot{\tilde{q}}$ - номинальная составляющая управления
- $v_s = k\frac{S}{\|S\|}$ - составляющая управления для прихода на поверхность

1.4 Sliding condition

Чтобы система сходилась с определенной скоростью, можно ввести условие:

$$\frac{1}{2}\frac{d}{dt}\|S\|^2 = S^T\dot{S} < -\eta\|S\| \tag{1.4.1}$$

Производная по времени от S выглядит соответственно:

$$\dot{S} = \ddot{\tilde{q}} + \lambda\dot{\tilde{q}} = v_n - \ddot{q} = v_n - f - B(v_n + v_s) = w - Bv_s \tag{1.4.2}$$

где:

$$w = (I - B)v_n - f \tag{1.4.3}$$

Подставляя (1.4.2) в (1.4.1) получим:

$$S^T(w - Bv_s) \le \|S\|\|w\| - S^TBv_s \le -\eta\|S\| \tag{1.4.4}$$

1.5 Робастный контроллер

Контроллер v можно выбрать как:

$$v_s = \frac{k}{\sigma_{max}}\hat{M}^{-1}\frac{S}{\|S\|} = \rho\frac{S}{\|S\|} \tag{1.5.1}$$

где σ_{max} - максимальное сингулярное значение матрицы M^{-1}
Благодаря выбранному таким образом v_s:

$$\|S\|\|w\| - S^TBv_s \leq \|S\|\|w\| + \frac{k}{\lambda_{max}^2\|S\|}S^TM^{-1}S \leq \|S\|\|w\| + k\|S\| < -\eta\|S\| \tag{1.5.2}$$

Система будет сходиться при $k > \|w\| + \eta$

1.6 Итоговая система

$$\begin{cases} u = \hat{M}(q)v + (\hat{C}(q,\dot{q}) + \hat{D})\dot{q} + \hat{g}(q) + \hat{F}_c(\dot{q}) \\ v = \ddot{q}_{target} + \lambda\dot{\tilde{q}} + v_s \\ v_s = \frac{k}{\sigma_{max}}\hat{M}^{-1}\frac{S}{\|S\|} = \rho\frac{S}{\|S\|} \\ S = \dot{\tilde{q}} + \lambda\tilde{q} \end{cases}.$$

2. Реализация и анализ эффективности

2.1 Модификация модели робота UR5

- Дополнительная масса на энд-эффекторе задается строкой:

```
sim.modify_body_properties("end_effector", mass=2)
```

- Коэффициэнты демпфирования задаются в виде numpy.array:

```
D = np.array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5])
```

```
и устанавливаются с помощью:
```

```
sim.set_joint_damping(D)
```

- Кулоновское трение задается постоянным для каждого сустава:

```
F_c = np.array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5])
```

```
и устанавливается с помощью:
```

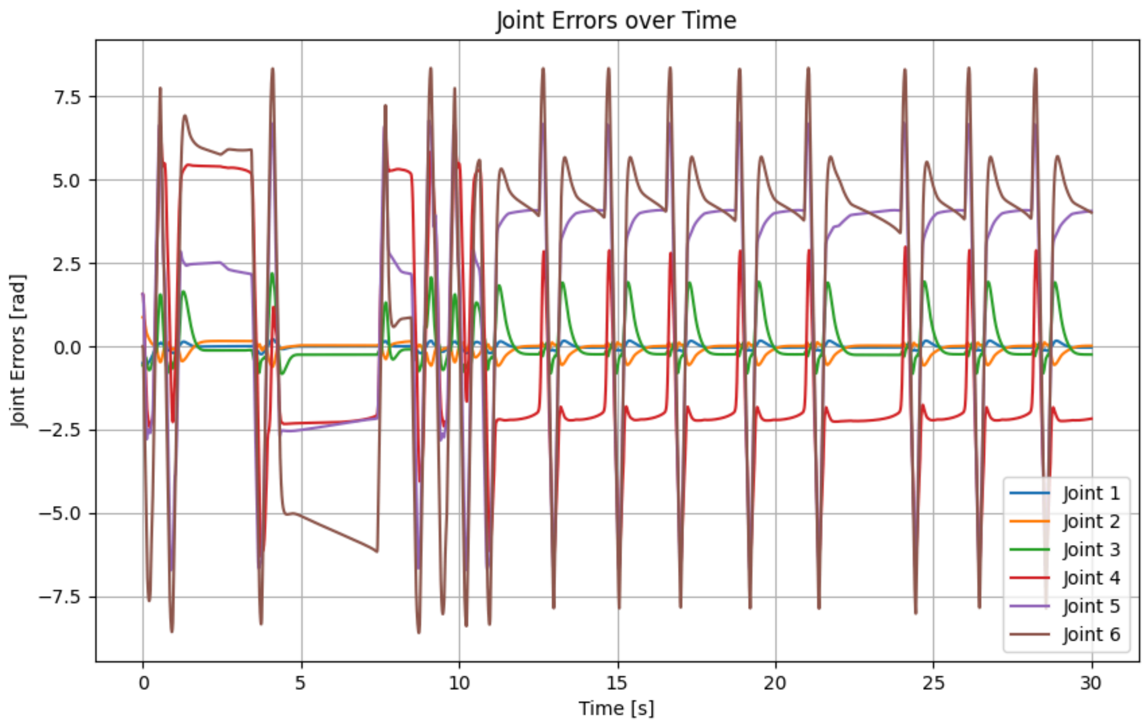
```
sim.set_joint_friction(F_c)
```

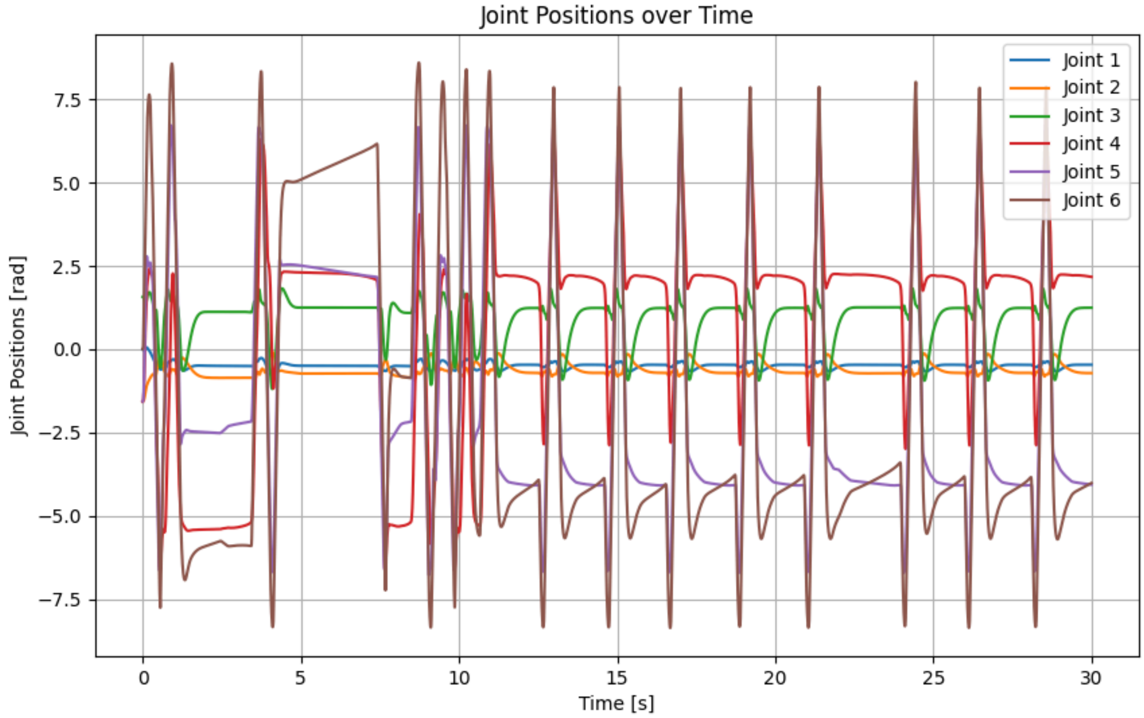
2.2 Реализация контроллеров

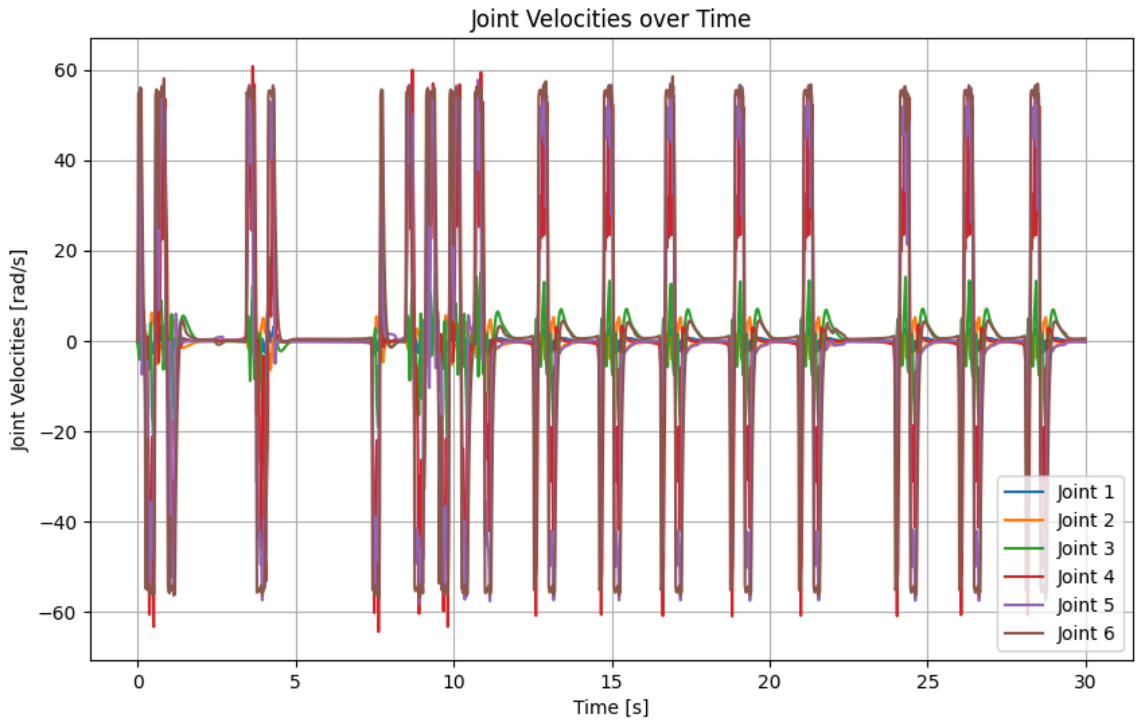
2.2.1 Реализация PD контроллера

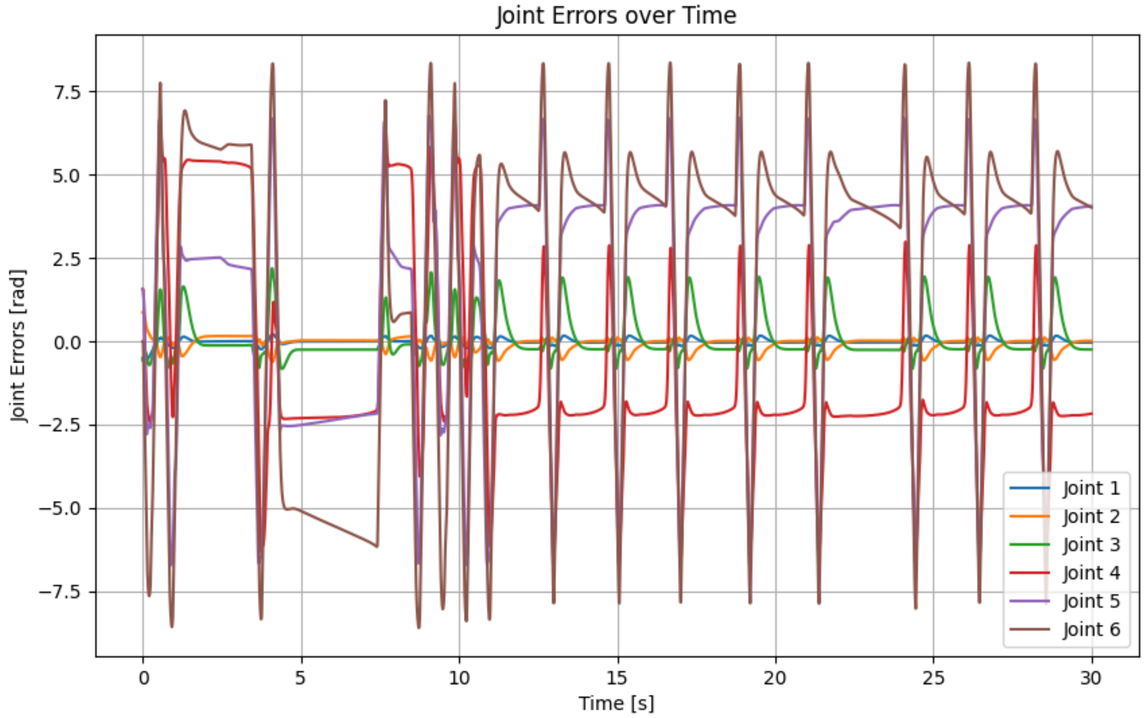
```
def get_worse_parameters(M, C, g, D, F_c):
    result_disp = 0.01
    result = ((result_disp * 2 * np.random.random(5) - result_disp) + 1)

    M_hat   = result[0] * M
    C_hat   = result[1] * C
    g_hat   = result[2] * g
    D_hat   = result[3] * D
    F_c_hat = result[4] * F_c

    return M_hat, C_hat, g_hat, D_hat, F_c_hat




def controller(q: np.ndarray, dq: np.ndarray, t: float) -> np.ndarray:
    t_history.append(t)
    joint_position_history.append(q)
```

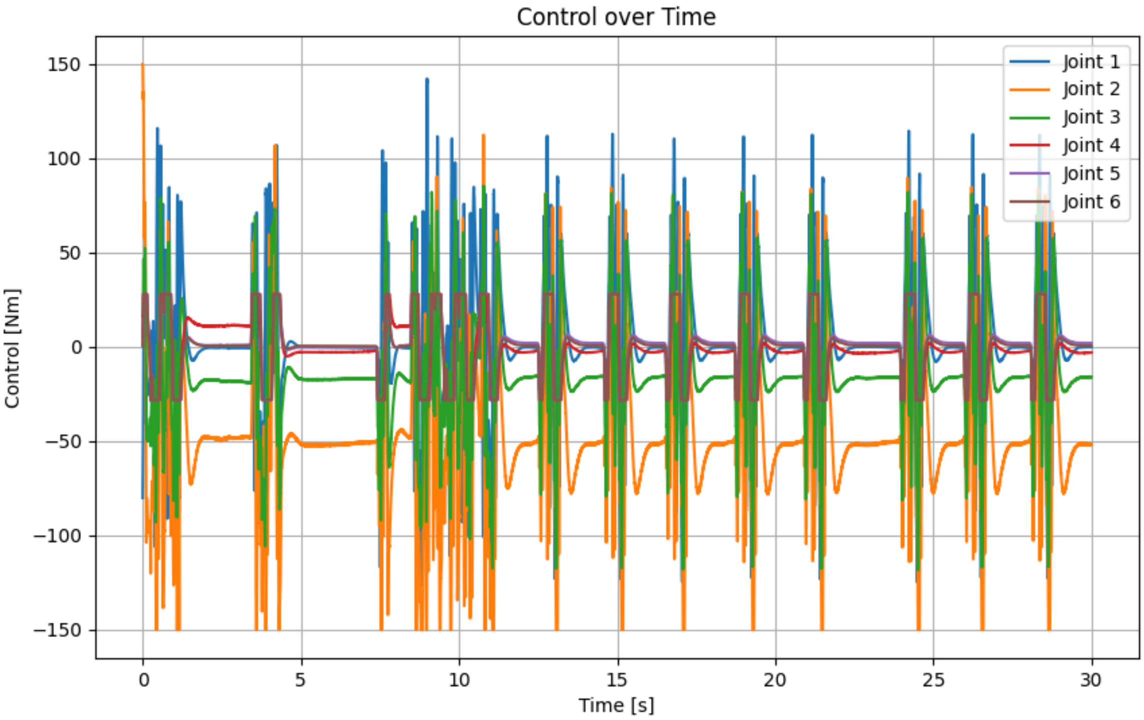
```python
    joint_velocity_history.append(dq)

    pin.computeAllTerms(model, data, q, dq)

    M_hat, C_hat, g_hat, D_hat, F_c_hat = get_worse_parameters(data.M, data.C, data.g, D, F_c)

    q_t = np.array([-0.5, -0.7, 1.0, 0.0, 0.0, 0.0], dtype=float)
    d_q_t = np.zeros(6, dtype=float)
    dd_q_t = np.zeros(6, dtype=float)

    q_err = q_t - q
    error_history.append(q_err)

    d_q_err = d_q_t - dq

    kp = 100
    kd = 20

    v = kp * q_err + kd * d_q_err + dd_q_t
    u = M_hat @ v + (C_hat + D_hat) @ dq + g_hat + F_c_hat
    u_history.append(u)

    return u
```




Joint Positions over Time
Joint Positions [rad]
Time [s]
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
0
5
10
15
20
25
30


Joint Velocities over Time
Joint Velocities [rad/s]
Time [s]
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6
60
40
20
0
−20
−40
−60
0
5
10
15
20
25
30

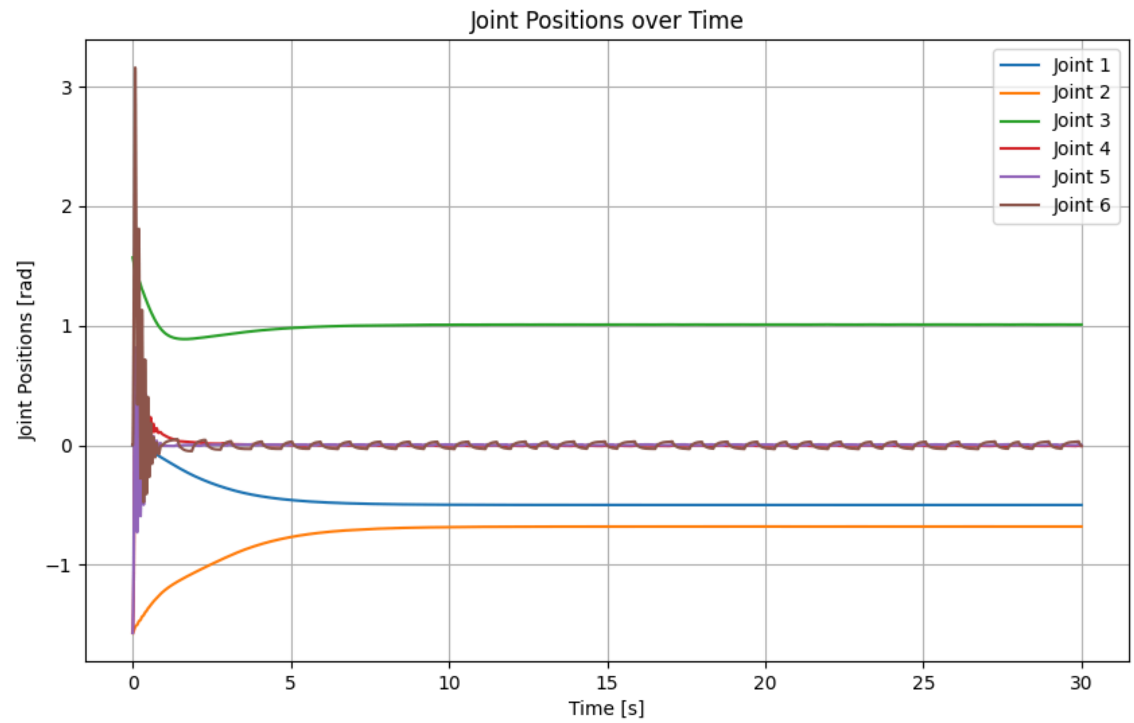


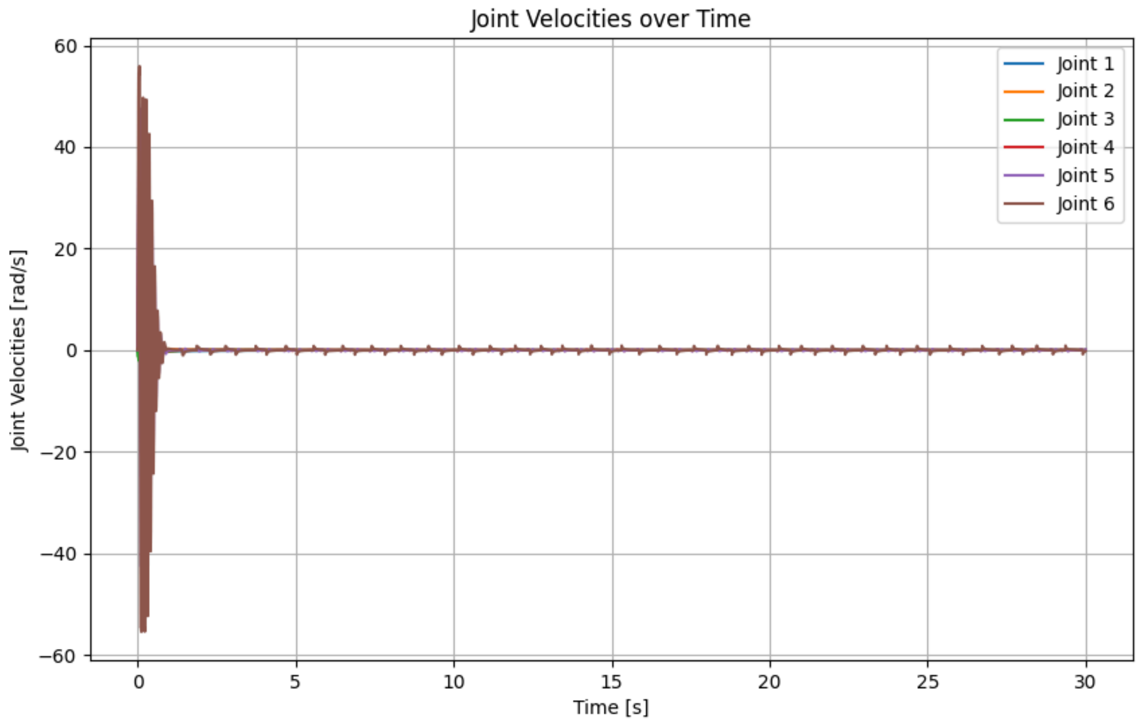


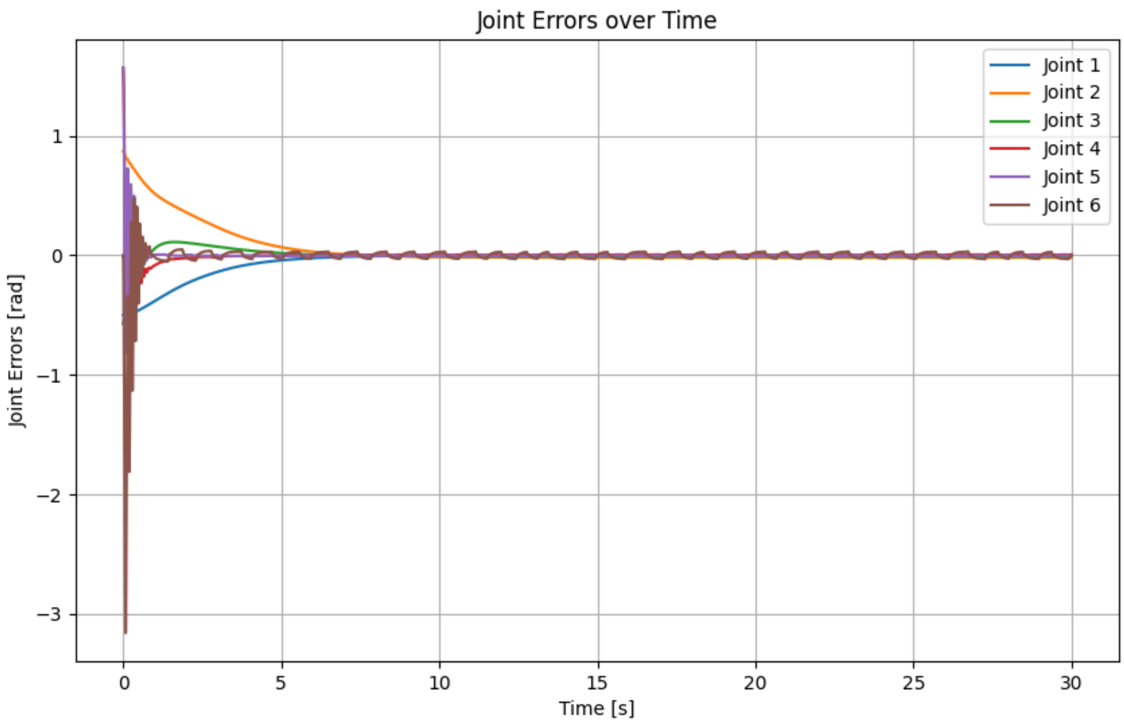
2.2.2 Реализация робастного контроллера

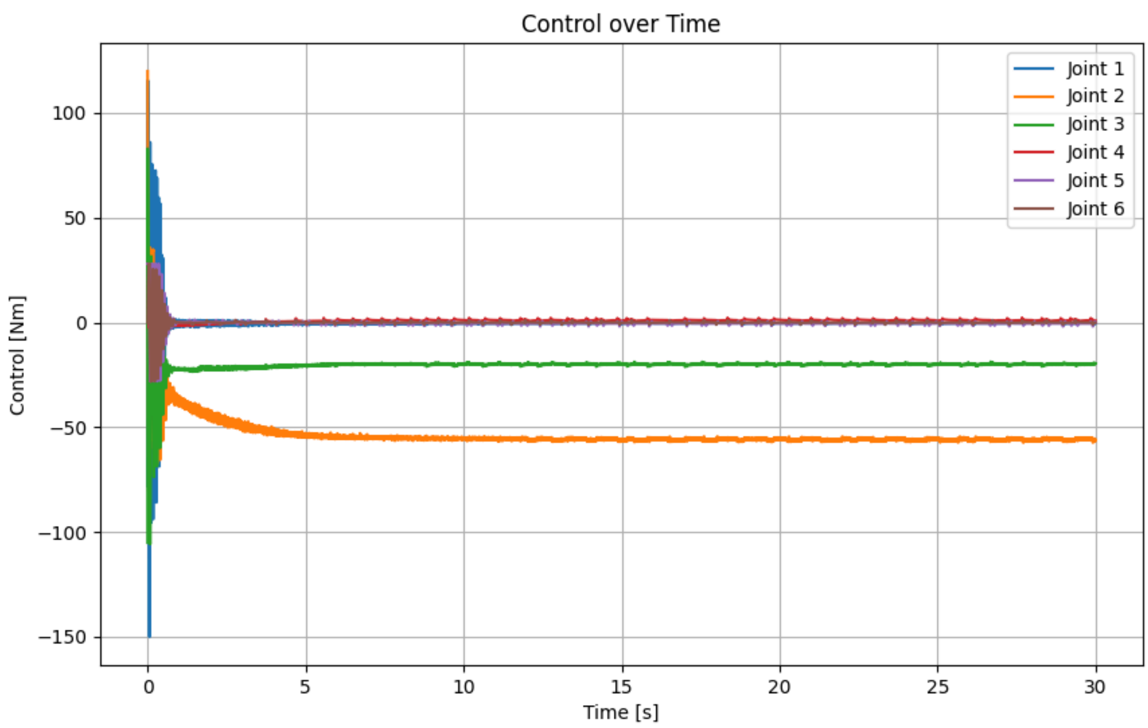
```
def get_worse_parameters(M, C, g, D, F_c):
    result_disp = 0.01
    result = ((result_disp * 2 * np.random.random(5) - result_disp) + 1)

    M_hat   = result[0] * M
    C_hat   = result[1] * C
    g_hat   = result[2] * g
    D_hat   = result[3] * D
    F_c_hat = result[4] * F_c
```

```python
    return M_hat, C_hat, g_hat, D_hat, F_c_hat

def V_s(M_hat, S):
    k = 1000
    epsilon = np.array([200, 200, 100, 100, 10, 100])
    sigma_max = 5

    S_norm = np.ones(6) * np.linalg.norm(S)

    S_norm = np.array(list([(eps * np.sign(S_norm[i]) if S_norm[i] <= eps else S_norm[i]) for i, eps
in enumerate(epsilon)]))

    rho = (k / sigma_max) * np.linalg.pinv(M_hat)

    v_s = rho @ (S / np.mean(S_norm))

    return v_s

def controller(q: np.ndarray, dq: np.ndarray, t: float) -> np.ndarray:
    t_history.append(t)
    joint_position_history.append(q)
    joint_velocity_history.append(dq)

    pin.computeAllTerms(model, data, q, dq)

    M_hat, C_hat, g_hat, D_hat, F_c_hat = get_worse_parameters(data.M, data.C, data.g, D, F_c)

    q_t = np.array([-0.5, -0.7, 1.0, 0.0, 0.0, 0.0], dtype=float)
    d_q_t = np.zeros(6, dtype=float)
    dd_q_t = np.zeros(6, dtype=float)

    lambdas = np.array([400, 400, 400, 100, 100, 10], dtype=float)

    q_err = q_t - q
    error_history.append(q_err)

    d_q_err = d_q_t - dq

    S = d_q_err + lambdas * q_err

    V = dd_q_t + lambdas * d_q_err + V_s(M_hat, S)

    u = M_hat @ V + (C_hat + D_hat) @ dq + g_hat + F_c_hat * np.sign(dq)

    u_history.append(u)

    return u
```


Joint Positions over Time
Joint Positions [rad]
Time [s]
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6
3
2
1
0
−1
0
5
10
15
20
25
30


Joint Velocities over Time
Joint Velocities [rad/s]
Time [s]
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6
60
40
20
0
−20
−40
−60
0
5
10
15
20
25
30





2.2.3 Сравнение PD и Robust контроллеров

Видно, что в случае robust контроллера система сходится и остается в стабильном положении, в то время как PD не обеспечивает в данном случае даже зануления ошибки, а приводит к некоторому колебательному процессу

Реализация пограничного слоя

Пограничный слой реализован в виде выражения:

$$v_s = \begin{cases} \rho \dfrac{S}{\|S\|}, \ \|S\| > \epsilon \\ \rho \dfrac{S}{\epsilon}, \ \|S\| \leq \epsilon \end{cases}$$

В коде это реализовано отдельно для каждого сустава и выглядит как:

```
def V_s(M_hat, S):
    k = 1000
    epsilon = np.array([200, 200, 100, 100, 10, 100])
    sigma_max = 5

    S_norm = np.ones(6) * np.linalg.norm(S)

    S_norm = np.array(list([(eps * np.sign(S_norm[i]) if S_norm[i] <= eps else S_norm[i]) for i, eps
in enumerate(epsilon)]))

    rho = (k / sigma_max) * np.linalg.pinv(M_hat)

    v_s = rho @ (S / np.mean(S_norm))

    return v_s
```

Вывод

Robust Control значительно лучше работает в условиях, когда параметры системы известны лишь приблизительно.