Есть система координат TEME с центром в Земле. Есть системе координат параллельная этой, но с центром в спутнике. И есть собственная система координат спутника. Необходимо научится с помощью трех (позже возможно будет 4) маховиков **гасить** произвольное вращение, которое возникает из-за раскрытия антенн, батарей… (возможно есть те эффекты, которые мы просто так не учтем) и **следить** за точками, которые заданы, например в TEME.
Есть несколько способов определить ориентацию:

1. Солнечные датчики (вектор на Солнце), точность до условно до 10 угловых минут (можно пробовать и точнее)
2. Звездные датчики (вектор на область неба + поворот вокруг этого вектора), можно очень точно, до обычно около 1 секунды
3. Датчики магнитного поля (вектор по магнитному полю), можно достичь хорошей точности, десятки минут, но довольно сложные пересчет в TEME + работает до 2000 км (первые миссии планируются на 500 км)
4. «Датчики» Земли (вектор на геометрический центр Земли). Камеры или ИК-датчики, которые определяют направление на центр Земли. Точность небольшая на низких орбитах около градуса. Для более уже на 40000 км будет хорошо работать, можно пробовать точность 1 минуту получить.
5. 2 (или более) GPS модуля, определяют относительно положение, из этого ориентацию. Точность — десятки минут (работает тоже где-то до 2000 км)
6. Можно отдельно выделить яркие источники (Луна, яркие планеты, может несколько звезд). Точность около 1 минуты (можно попробовать точнее). Получаем просто вектор на эти объекты.

Все можно разделить на 3 части:

1. Датчики (дают какой-то вектор, может еще дополнительную информацию)
2. Матмодель (программа, которая производит все преобразования)
3. Система ориентации (микроконтроллер с моторами, который получает данные от матмодели, по которой наводится)

(Датчики → матмодель → система ориентации)
Над датчиками сейчас работает, примитивные уже есть, еще чуть-чуть дописать и будем вектор получать. Система ориентации из моторов тоже какая-то есть (дорабатываем). Теперь нужно реализовать матмодель.
Есть собственная система координат спутника $(\vec{x};\vec{y};\vec{z})$ - это базис спутника. В нем мы получаем 2 вектора $\vec{s}$ и $\vec{g}$, выраженные через базис спутника.
Пусть:
S — система координат спутника $(\vec{x};\vec{y};\vec{z})$,
O — промежуточная система координат $(\vec{s};\vec{g};[\vec{s}\times\vec{g}])$ (вектор на Солнце, на Землю или еще какой-то вспомогательный и их векторное произведение)
G — TEME с центром в спутнике.
Пусть q_1 кватернион поворота S → O, q_2 кватернион поворота O → G
Тогда поворот S→ O→ G описывается кватернионом $q_3 = q_2 \cdot q_1$
Есть алгоритмы нахождения кватерниона через базисные вектора (deepseek что-то написал, надо изучить, приведу его ответ внизу).
Для ориентации нам нужно будет задать один (или несколько, чтоб отслеживать) вектор с углом (вращение вокруг этого вектора).
Чтоб сделать вращение S→S’ можем получить кватернион $q = q'_2 \cdot q^{-1}_1 \cdot q_2 \cdot q_1$
Чтобы погасить произвольное вращение, можно небольшими промежутками времени брать Δt для них получать кватернионы и гасить это вращение (способ надо обдумать, потому что это скорее идея)

Чтобы прогнозировать вращение есть 2 основных способа интерполяции: SLERP (кусочно линейная) и SQUAD (гладкая, непрерывность C^1)
Наверно имеет смысл использовать SQUAD, т. к. более физично. Deepseek выдает 2 реализации, одна выглядит довольно просто. Получаем сразу вектор, угловую скорость и ускорение, которое нужно погасить. Но тоже надо подробнее разобраться, т. к. есть ращные системы отсчета

```
import numpy as np
from scipy.spatial.transform import Rotation, RotationSpline

# 1. Создаем сплайн по ключевым точкам
times = [0, 1, 2]
quaternions = [
    [0, 0, 0, 1],        # t=0: identity
    [0.5, 0.5, 0.5, 0.5], # t=1: 120° вокруг (1,1,1)
    [1, 0, 0, 0]          # t=2: 180° вокруг x
]
spline = RotationSpline(times, Rotation.from_quat(quaternions))

# 2. Функция для получения угловой скорости и ускорения
def get_angular_kinematics(t):
    # Угловая скорость в системе тела [rad/s]
    ω_body = spline(t, 1)

    # Угловое ускорение в системе тела [rad/s²]
    α_body = spline(t, 2)

    # Матрица поворота для преобразования в инерциальную систему
    R = spline(t).as_matrix()

    # Преобразование в инерциальную систему
    ω_inertial = R @ ω_body
    α_inertial = R @ α_body

    return {
        'quaternion': spline(t).as_quat(),
        'omega_body': ω_body,
        'alpha_body': α_body,
        'omega_inertial': ω_inertial,
        'alpha_inertial': α_inertial
    }

# 3. Пример использования для t=0.7
t = 0.7
result = get_angular_kinematics(t)

print(f"Время t = {t:.1f} s:")
print(f"Кватернион: {result['quaternion']}")
print(f"Угловая скорость (тело): {result['omega_body']} rad/s")
print(f"Угловое ускорение (тело): {result['alpha_body']} rad/s²")
print(f"Угловая скорость (инерц.): {result['omega_inertial']} rad/s")
```

```
print(f"Угловое ускорение (инерц.): {result['alpha_inertial']} rad/s²")
```

Алгоритм нахождения кватерниона через базисные вектора

Для преобразования ортонормированного базиса E = {e_1, e_2, e_3} в базис F = {f_1, f_2, f_3} с помощью кватерниона, выполните следующие шаги:

---

1. Построение матрицы поворота
Матрица R размера 3×3, столбцы которой — координаты векторов f_i в исходном базисе E:

$$R = \begin{vmatrix} | & | & | \\ f_1 & f_2 & f_3 \\ | & | & | \end{vmatrix}$$

где:
- $f_1 = (r_{11}, r_{21}, r_{31})^T$
- $f_2 = (r_{12}, r_{22}, r_{32})^T$
- $f_3 = (r_{13}, r_{23}, r_{33})^T$

Проверка условий:
- Ортогональность: $R^TR = I$
- Правовинтовость: det(R) = 1

---

2. Преобразование матрицы в кватернион
Для матрицы R с элементами:

$$R = \begin{vmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{vmatrix}$$

вычислите компоненты кватерниона q = (w, x, y, z) по устойчивому алгоритму:

```
import numpy as np

def matrix_to_quaternion(R):
    # Шаг 1: Вычисление следа
    T = R[0,0] + R[1,1] + R[2,2]

    if T > 0:
        S = np.sqrt(T + 1.0) * 2
        w = S / 4
        x = (R[2,1] - R[1,2]) / S
        y = (R[0,2] - R[2,0]) / S
        z = (R[1,0] - R[0,1]) / S

    elif (R[0,0] > R[1,1]) and (R[0,0] > R[2,2]):
        S = np.sqrt(1.0 + R[0,0] - R[1,1] - R[2,2]) * 2
```

```
        w = (R[2,1] - R[1,2]) / S
        x = S / 4
        y = (R[0,1] + R[1,0]) / S
        z = (R[0,2] + R[2,0]) / S

    elif R[1,1] > R[2,2]:
        S = np.sqrt(1.0 - R[0,0] + R[1,1] - R[2,2]) * 2
        w = (R[0,2] - R[2,0]) / S
        x = (R[0,1] + R[1,0]) / S
        y = S / 4
        z = (R[1,2] + R[2,1]) / S

    else:
        S = np.sqrt(1.0 - R[0,0] - R[1,1] + R[2,2]) * 2
        w = (R[1,0] - R[0,1]) / S
        x = (R[0,2] + R[2,0]) / S
        y = (R[1,2] + R[2,1]) / S
        z = S / 4

    q = np.array([w, x, y, z])
    return q / np.linalg.norm(q)  # Нормализация
```

---

3. Логика алгоритма

1. Выбор устойчивой ветки:
   - Сравнивается след T и диагональные элементы
   - Предотвращает деление на малые числа
   - Избегает потери точности

2. Ключевые формулы:
   - Основаны на соотношении компонент кватерниона и матрицы:

$$R = \begin{vmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 \end{vmatrix}$$

3. Нормализация:
   - Гарантирует $w^2 + x^2 + y^2 + z^2 = 1$
   - Исправляет погрешности вычислений

---

4. Пример расчета

Дано:
- Базис E: стандартный {[1,0,0], [0,1,0], [0,0,1]}
- Базис F: поворот на 90° вокруг X: {[1,0,0], [0,0,1], [0,-1,0]}

Решение:
1. Матрица R:

R = | 1 0 0 |
    | 0 0 -1 |
    | 0 1 0 |

2. След T = 1 + 0 + 0 = 1 > 0 → первая ветка:

S = sqrt(1 + 1)*2 ≈ 2.828
w = 2.828 / 4 ≈ 0.707
x = (0 - (-1)) / 2.828 ≈ 0.353
y = (0 - 0) / 2.828 = 0
z = (0 - 0) / 2.828 = 0

3. После нормализации: q = (√2/2, √2/2, 0, 0)
Соответствует повороту на 90° вокруг X.

---

5. Особые случаи
1. Тождественное преобразование:
- R = I
- q = (1, 0, 0, 0)

2. Разворот на 180°:
- Пример: поворот вокруг Z на 180°
- R = diag(-1, -1, 1)
- q = (0, 0, 0, 1)

3. Обратная связь:
- Всегда проверяйте $R^TR = I$ и det(R)=1
- Для проверки: q * vector * q^{-1} должно давать f_i

---

6. Геометрическая интерпретация
Кватернион напрямую связан с поворотом базиса:
w = cos(θ/2)
(x, y, z) = [sin(θ/2)] * (a_x, a_y, a_z)
где θ — угол поворота, a — ось вращения в исходной СК.

---

7. Преимущества перед другими методами
- Устойчивость: Алгоритм минимизирует ошибки округления
- Эффективность: 16 арифметических операций
- Совместимость: Легко интегрируется с SLERP для интерполяции
- Отсутствие сингулярностей: В отличие от углов Эйлера
Для сложных систем используйте SVD-разложение для предварительной ортогонализации матрицы.