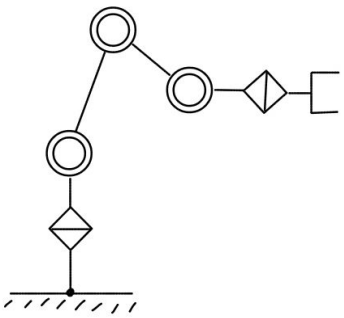
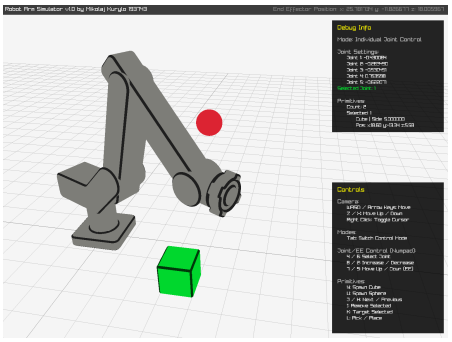
Projekt OOP I
Symulacja 5DOF ramienia robota

Autor:	Mikołaj Kuryło
Numer albumu:	193743
Kierunek:	ACiR 2B
Repozytorium:	https://github.com/3hoot/5DOF-Arm-simulator



Rys. 1: Schemat przegubowy robota



Rys. 2: Obraz programu

Cele projektu

Celem projektu było stworzenie symulatora 5-przegubowego ramienia robota (5DOF), który umożliwia:

- testowanie irozwijanie algorytmu kinematyki odwrotnej (Inverse Kinematics),
- wizualizację pozycji ramienia w przestrzeni 3D,
- możliwość interaktywnego sterowania ramieniem za pomocą GUI.

Projekt ma również na celu rozwinięcie umiejętności programistycznych w Pythonie, modelowania matematycznego manipulatorów oraz implementacji algorytmów sterowania.

Wykorzystane oprogramowanie i narzędzia

- **VS Code** – kodowanie i debugowanie,
- **Fusion 360** – tworzenie modeli 3D przegubów,
- **C++ / OpenGL / Raylib** – logika i grafika symulacji,
- **Eigen** – obliczenia macierzowe i geometryczne,
- **CMake** – zarządzanie kompilacją.

Konstrukcja robota

Ramię robota składa się z 5 segmentów, które tworzą przegubowy układ przestrzenny. Każde połączenie odpowiada jednemu stopniowi swobody, pozwalając na ruchy obrotowe w różnych osiach. Geometria i długości członów są zdefiniowane w kodzie, co umożliwia elastyczne modyfikacje.

Modele geometryczne zostały stworzone w programie **Fusion 360**, co umożliwiło przetestowanie układu przed implementacją symulacji.

Struktura projektu i obiektowość

Każdy plik źródłowy w połączeniu ze swoim odpowiednikiem w pliku nagłówkowym implementują klasę o określonej funkcjonalności służącej w symulowaniu robota. Poniżej przedstawiono strukturę projektu z opisem indywidualnym klas.
Zastosowanie podejścia obiektowego w projekcie umożliwia jego łatwe rozbudowanie, np. można łatwo dodać kolejny stopień swobody edytując zawartość plików *config*.

```
5DOF-Arm-simulator/
├── include/
│   └── robot_arm/
│       ├── core/
│       │   ├── config.hpp
│       │   ├── robot.hpp
│       │   ├── joint.hpp
│       │   └── utils.hpp
│       └── sim/
│           ├── config.hpp
│           ├── simulator.hpp
│           ├── primitive.hpp
│           └── utils.hpp
├── source/
│   ├── core/
│   │   ├── robot.cpp
│   │   ├── joint.cpp
│   │   └── utils.cpp
│   ├── sim/
│   │   ├── simulator.cpp
│   │   ├── primitive.cpp
│   │   └── utils.cpp
│   └── main.cpp
├── resources/
│   ├── primitives/
│   │   ├── sphere.obj
│   │   └── cube.obj
│   ├── J0.obj
│   ├── J1.obj
│   ├── J2.obj
│   ├── J3.obj
│   ├── J4.obj
│   └── J5.obj
└── CMakeLists.txt
```

Zagnieżdżenie poszczególnych plików w odpowiednich folderach w strukturze projektu zapewnia łatwe rozróżnienie do czego dany plik się odnosi, nad czym wykonuje pracę oraz dostępem do niego z poziomu kodu (np. klasy implementowane przez pliki nagłówkowe umieszczone w folderze robot_arm są dostępne w przestrzeni nazwowej robot_arm).

Moduły:

- **core** - moduł zawierający wyłącznie logikę robota, możliwa do wykorzystania poza samą symulacją.
- **sim** - moduł zawierający logikę przestrzeni symulacji robota, powiązania każdego przegubu z modelem 3D, implementacja wizualizacji oraz interakcji z prymitywami.

Pliki i klasy:

- **config** - plik kodu nie implementujący żadnej klasy a zawierający zmienne konfiguracyjne modułu w którym się znajduje.
- **utils** - plik nie implementujący żadnej klasy, zawiera przydatne funkcje wykorzystywane przez inne pliki kodu w swoim sąsiedztwie (np. funkcja tłumacząca przestrzeń roboczą robota na przestrzeń roboczą symulatora w pliku sim/utils)

- **joint** - implementuje klasę *Joint* odpowiedzialną za zarządzanie przegubami robota, wprowadza swoje parametry DH, macierz transformacji względem początku układu współrzędnych, nastawianie i limity ustawień.
- **robot -** implementuje klasę *Robot*, która realizuje całą logikę robota. Manipuluje indywidualnie przegubami i aktualizuje ich macierze transformacji oraz zawiera algorytm kinematyki wstecznej do pozycjonowania.

- **simulator** - implementuję klasę *Simulator* odpowiedzialną za wizualizację, wprowadzenie interfejsu do interakcji z robotem oraz zarządzania obiektami klasy *Primitive*.
- **primitive** - implementuje klasę *Primitive* tworzącą i zarządzającą informacjami związanymi z prymitywami.

Pliki zawarte w folderze *resources* zawierają modele 3D przegubów robota oraz prymitywów. Plik *CMakeLists.txt* służy w konfiguracji *build system'u* projektu oraz odpowiada za podłączenie niezbędnych bibliotek i plików nagłówkowych ze źródłami.

Algorytm kinematyki odwrotnej

```
std::vector<double> Robot::solveIK(const Eigen::Vector3d &target_position, const Eigen::Vector3d &approach_vector, const double approach_angle)
    {
        std::vector<double> settings = joint_settings_;

        const double offset_1 = joints_[0].getDHParameters().d +
                                joints_[1].getDHParameters().d;
        const double offset_2 = joints_[2].getDHParameters().a;

        const double a = joints_[3].getDHParameters().a;
        const double b = joints_[4].getDHParameters().a;
        const double d = std::sqrt(std::pow(target_position.x(), 2) + std::pow(target_position.y(), 2)) - offset_2;

        const double h = target_position.z() - offset_1;
        const double c = std::sqrt(std::pow(d, 2) + std::pow(h, 2));

        settings[1] = atan2(target_position.y(), target_position.x());

        double arg2 = (std::pow(b, 2) + std::pow(c, 2) - std::pow(a, 2)) / (2 * b * c);
        arg2 = std::clamp(arg2, -1.0, 1.0);
        settings[2] = -(std::numbers::pi / 2 - std::atan2(h, d) - std::acos(arg2));

        double arg3 = (std::pow(a, 2) + std::pow(b, 2) - std::pow(c, 2)) / (2 * a * b);
        arg3 = std::clamp(arg3, -1.0, 1.0);
        settings[3] = -(std::numbers::pi / 2 - std::acos(arg3));

        std::vector<Joint> joints_copy = joints_;
        Robot dummy_robot(joints_copy);
        for (size_t i = 0; i < settings.size(); ++i)
            dummy_robot.setJointSetting(i, settings[i]);

        Eigen::Vector3d joint4_to_ee = (dummy_robot.getTransformationMatrices().back().block<3, 1>(0, 3) -
                                        dummy_robot.getTransformationMatrices()[4].block<3, 1>(0, 3)).normalized
        Eigen::Vector3d approach_vector_normalized = approach_vector.normalized();
        Eigen::Vector3d joint4_axis = dummy_robot.getTransformationMatrices()[4].block<3, 1>(0, 2).normalized();

        Eigen::Vector3d joint4_to_ee_proj = (joint4_to_ee - joint4_to_ee.dot(joint4_axis) * joint4_axis).normalized();
        Eigen::Vector3d approach_proj = (approach_vector_normalized - approach_vector_normalized.dot(joint4_axis) * joint4_axis).normalized();

        double angle_to_approach = std::acos(joint4_to_ee_proj.dot(approach_proj));
        double signed_angle = std::atan2(
            joint4_axis.dot(joint4_to_ee_proj.cross(approach_proj)),
            joint4_to_ee_proj.dot(approach_proj));
        settings[4] = settings[4] + signed_angle;

        settings[5] = approach_angle;
        return settings;
    }
```

Kod implementacji kinematyki wstecznej robota

Funkcja Robot::solveIK() wyznacza kąty przegubów potrzebne do osiągnięcia przez końcówkę ramienia zadanej pozycji i orientacji. Główne kroki:

1. **Parametry wejściowe:**
   - target_position – docelowa pozycja końcówki w przestrzeni,
   - approach_vector – wektor podejścia (orientacja końcówki),
   - approach_angle – kąt końcowego obrotu.
2. **Obliczenia geometryczne:**
   - Obliczane są wartości d, h, c tworzące trójkąt dla przegubów 2 i 3.
   - Za pomocą funkcji trygonometrycznych (gł. acos, atan2) wyznaczane są kąty dla przegubów tak, aby ramię sięgnęło do celu.
3. **Dopasowanie orientacji końcówki:**
   - Obliczany jest obrót czwartego przegubu, aby końcówka była zgodna z wektorem podejścia.
   - Ustalany jest także kąt przegubu 5 w oparciu o approach_angle.
4. **Normalizacja i kontrola błędów:**
   - Wartości argumentów acos są ograniczane (clamp) do zakresu [-1, 1] by uniknąć błędów numerycznych. Wektorowe obliczenia orientacji zapewniają precyzyjne dopasowanie.

Funkcja zwraca wektor wartości kątowych do ustawienia przegubów. W kodzie istnieje również tworzenie tymczasowego robota w celu walidacji transformacji końcowej.

Wnioski

1. Projekt zrealizował wszystkie założenia: działa wizualizacja 3D, GUI oraz algorytm IK.
2. Zastosowanie OpenGL oraz Raylib pozwoliło na stworzenie przejrzystej i interaktywnej symulacji.
3. Implementacja algorytmu IK jest poprawna matematycznie i zapewnia szybkie obliczenia.
4. Projekt ma modularną budowę – łatwo można go rozbudować o kolejne stopnie swobody, kolizje czy obsługę trajektorii.

Możliwości rozbudowy:

- dodanie systemu kolizji i detekcji kolizji między członami,
- rozbudowa GUI o zapis i odczyt trajektorii ruchu,
- wdrożenie obsługi joysticka lub manipulatora zewnętrznego,
- migracja do środowiska ROS w celu integracji ze sprzętem rzeczywistym.

Bibliografia

- en.wikipedia.org
- www.roboticsunveiled.com
- stackoverflow.com
- robotics101-viscircuit.web.app
- www.raylib.com
- eigen.tuxfamily.org