

**POLITECHNIKA BIAŁOSTOCKA**  
**WYDZIAŁ MECHANICZNY**

**PRACA DYPLOMOWA INŻYNIERSKA**

**TEMAT:** Rozpoznawanie i estymacja pozycji obiektu z użyciem systemu wizyjnego w zadaniu chwytania realizowanego przez manipulator

**WYKONAWCA:** Adam Wołkowycki

**PODPIS:** .....

**PROMOTOR:** Dr inż. Adam Wolniakowski

**PODPIS:** .....

**BIAŁYSTOK 2022**

## Abstract

In the recent years, a rising trend in application of autonomous robots can be observed. Nowadays, many solutions intended for this part of market are developed and each of these is based on solutions that are hard to algorithmize. The examples of this kind of tasks are image processing, object position estimation and configuration of kinematics. I am going to focus on them in my thesis.

The purpose of this thesis is to develop a solution that allows for recognition of a given object and perform grasping of it by a manipulator. The further development of the problem known as *bin picking* aims at increasing a scope of industrial robots' operation in a context of their work. It is quintessential in many modern robotic applications targeted for being more "flexible" and intuitive.

Because the tasks I mentioned are often performed by a particular type of robots, in my work I am going to use UR5 manipulator by Danish manufacturer *Universal Robots*. There will be implemented algorithm on it that will be able to define a location of seen objects based on the data from the computer vision sensors. In my work I will be using code examples written in Python. Thanks to this, it will be possible to thoroughly verify in practice the working of a robot.

## Streszczenie

W ostatnich latach można zaobserwować wzrastający trend na zastosowanie robotów o działaniu autonomicznym. Powstaje obecnie wiele rozwiązań przeznaczonych dla tego segmentu rynku, a każdy z nich opiera swoje działanie o rozwiązania, które są przyjęte jako trudno algorytmizowalne. Przykładami takich zadań są przetwarzanie obrazu, estymacja położenia obiektów i konfiguracja kinematyki robota. Właśnie na nich zamieram skupić się w mojej pracy.

Celem pracy jest opracowanie rozwiązania odpowiedzialnego za rozpoznawanie wskazanego obiektu i realizację chwycenia go przy wykorzystaniu manipulatora. Rozwinięcie problemu znanego pod nazwą *bin picking* ma przede wszystkim na celu zwiększenie zakresu działania robotów przemysłowych w kontekście ich pracy. Stanowi to kwintesencję w wielu współczesnych zastosowaniach robotyki dążącej do bycia bardziej "elastyczną" i intuicyjną.

Ponieważ zadania, o których wspomniałem są często realizowane przez określony rodzaj robotów, w swojej pracy zamierzam wykorzystać manipulator UR5 duńskiego producenta Universal Robots. Na nim zostanie zaimplementowany algorytm, który w oparciu o dane z systemu wizyjnego będzie w stanie określić położenie widzianych przedmiotów. W pracy posłużę się programami stworzonymi w języku Python. Dzięki temu działanie robota będzie można gruntownie zweryfikować w praktyce.

## Spis treści

<b>1. Wprowadzenie</b>	<b>6</b>
1.1. Manipulacja a percepcaja . . . . .	7
1.2. Cele manipulacji . . . . .	9
1.3. Afordancje . . . . .	10
1.4. Zarys pracy . . . . .	12
<b>2. Przegląd dotychczasowych rozwiązań</b>	<b>15</b>
2.1. Widzenie maszynowe . . . . .	15
2.2. Sztuczne sieci neuronowe . . . . .	16
2.3. Uczenie przez wzmacnianie . . . . .	20
<b>3. Estymacja położenia obiektu</b>	<b>23</b>
3.1. Wyznaczenie najbliższego punktu . . . . .	23
3.2. Algorytm najbliższego sąsiada . . . . .	24
3.3. Transformacje . . . . .	25
3.4. Rozkład macierzy według wartości osobliwych . . . . .	27
3.5. Interpretacja geometryczna rozkładu SVD . . . . .	31
<b>4. Algorytm Iterative Closest Point</b>	<b>33</b>
4.1. Implementacja algorytmu . . . . .	36
4.2. Działanie robota w praktyce . . . . .	39
<b>5. Podsumowanie i wnioski</b>	<b>44</b>
5.1. Dalszy rozwój . . . . .	45
<b>Bibliografia</b>	<b>48</b>

## Wykaz oznaczeń

Podczas pisania pracy została użyta podana notacja.

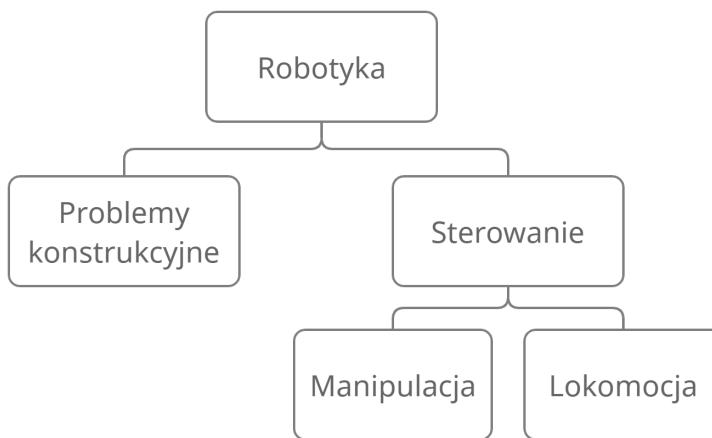
- $\mathbf{A}$  - macierze oznaczone zostały pogrubionymi wielkimi literami
- $\mathbf{v}$  - wektory oznaczone zostały pogrubionymi małymi literami
- $s$  - wartości skalarne oznaczone zostały małymi literami
- $\mathbb{R}$  - zbiory liczb oznaczone zostały konturowymi wielkimi literami
- $\mathbf{R}$  - macierz rotacji (obrotu)  $3 \times 3$
- $\mathbf{t}$  - wektor translacji (przesunięcia)  $1 \times 3$
- $\mathbf{T}$  - macierz transformacji  $4 \times 4$

## 1. Wprowadzenie

Definiując robotykę jako dziedzinę wiedzy mamy na uwadze zajmowanie się urządzeniami mechanicznymi, które mogą być przystosowane do realizacji wielu różnych zadań przez zmianę programu nimi sterującego. Jest to podstawowa różnica pozwalająca wyodrębnić robotykę na tle sztywnej automatyzacji zajmującej się maszynami przeznaczonymi do wykonywania jednego rodzaju zadań przez dłuższy czas. Współczesna robotyka stanowi przede wszystkim zbiór pewnych rozwiązań ukierunkowanych na tworzenie i sterowanie robotami. Mając to na uwadze, możemy wyróżnić dwa problemy: natury konstrukcyjnej i sterowania. Ponieważ wielu producentów z branży robotyki oferuje szeroką gamę rozwiązań spełniających potrzeby niniejszej pracy, posłużymy się manipulatorem przegubowym - najbardziej powszechną odmianą robota przemysłowego, w którego konstrukcję nie będziemy ingerować. Dlatego, aby zrozumieć koncepcje zawarte w tej pracy nie jest wymagana dogłębiańska znajomość budowy robota. Natomiast, jeśli chodzi o sterowanie to możemy ponownie wyodrębnić dwie podrzędne kategorie: manipulację i lokomocję.

- **Manipulację** definiujemy jako zdolność do wykonywania precyzyjnych ruchów przez efektor końcowy robota w celu osiągnięcia wyznaczonego celu, np. przeniesienia przedmiotu procesowanego - tzw. zadanie *pick and place*. Roboty posiadające tą cechę nazywamy mianem manipulatorów.
- **Lokomocja** z kolei, jak sama nazwa wskazuje, opisuje zdolność do przemieszczania się platformy mobilnej. Jest kluczowa m.in. dla pojazdów AVG (*Automated Guided Vehicles*), łazików czy robotów kroczących. Ponieważ w swojej pracy zamierzam skupić się na robotach stacjonarnych, nie będę wracał więcej do tego tematu.

Musimy jednak mieć na uwadze, że jest to dość ogólna i luźna klasyfikacja. Pierwotnie, definicję wyodrębniającą robotykę i automatyzację zaproponował John J. Craig, natomiast podział na manipulację i lokomocję został zastosowany w książceMarca H. Raiberta *Legged Robots That Balance* w ostatnim rozdziale w pytaniu *Do Locomotion and Manipulation Have a Common Ground?* Tam też problem został rozwinięty w kontekście badań nad lokomocją i manipulacją oraz tego jak te dwie dziedziny wpływają



Rysunek 1: Podział robotyki na odrębne zagadnienia.

na siebie nawzajem.

## 1.1. Manipulacja a percepceja

Gdy mamy już wyjaśnione ogólne zagadnienia możemy skupić się na danym zadaniu manipulatora. Założymy, że zadaniem robota jest pobranie kilku kostek z podajnika w miejscu A i przeniesienie ich w odpowiednie miejsca na palecie B, tzw. paletyzacja. Aby wykonać to zadanie możemy podejść do niego na dwa sposoby: zapewniając maszynie odpowiednią percepcję lub nie.

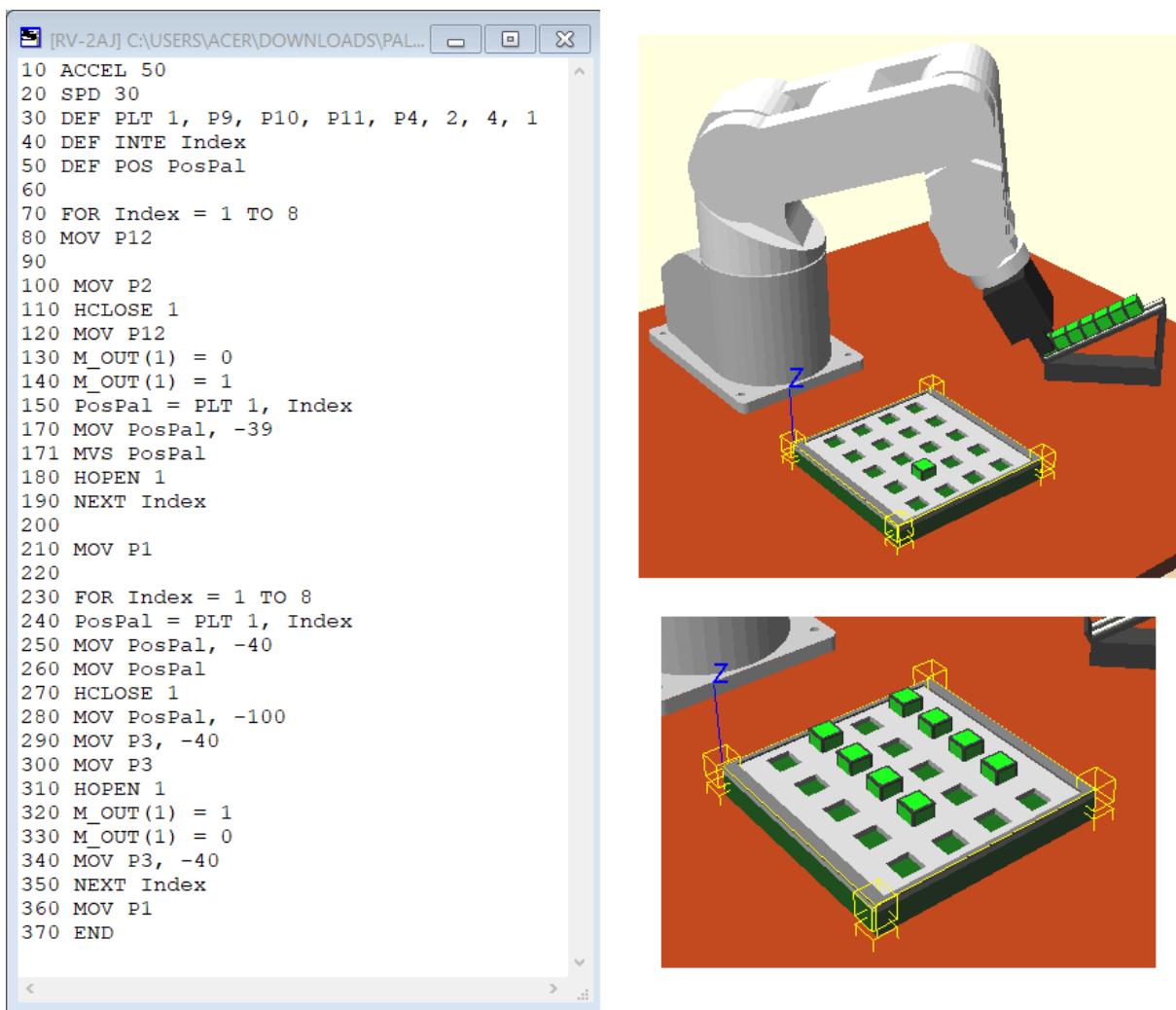
- **Przypadek 1. Brak percepacji**

Brak percepacji oznacza brak danych wejściowych. Operujemy jedynie na wielkościach, które muszą zostać założone z góry. W tym przykładzie będą to: pozycje bazy i kłuci manipulatora, podajnika, palety oraz rozmiar kostki. Z tego powodu, robot pozbawiony percepji zawsze musi otrzymać od nas dokładne informacje odnośnie trajektorii ruchu.

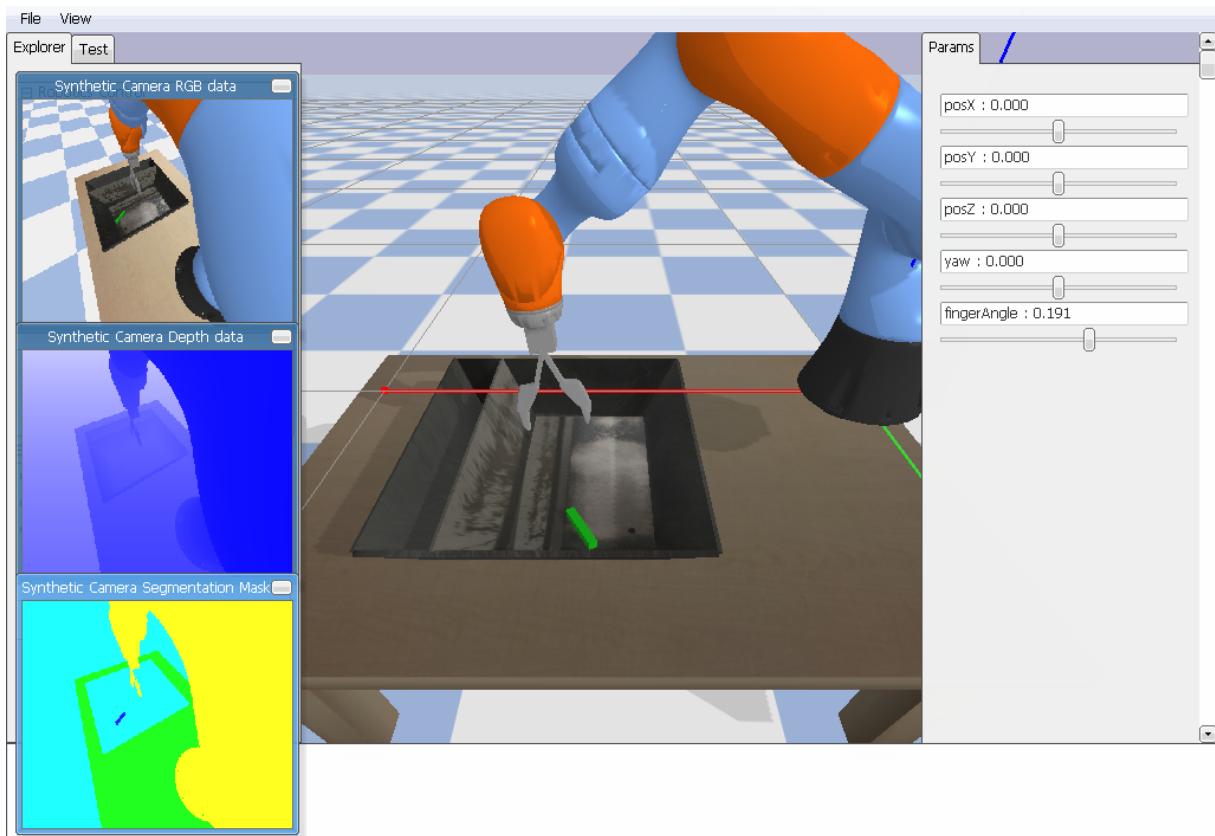
- **Przypadek 2. Percepceja z użyciem systemu wizyjnego**

Przez system wizyjny możemy rozumieć dowolny rodzaj kamery, jak również projektorzy chmur punktów i urządzenia nakładające na siebie kolory z danymi o głębokości obrazu, zwane kamerami RGBD. W takim przypadku operujemy znaczną ilością danych, dzięki czemu znajomość wymienionych wyżej wartości nie musi być konieczna - możemy oszacować te wartości używając metod widzenia maszynowego

(*Computer Vision*). Zastosowanie systemu wizyjnego możemy ponownie podzielić na dwa typy: z kamerą zamontowaną nieruchomo względem bazy robota oraz na jego narzędziu. Różnica między nimi polega głównie na tym, że w pierwszym przypadku widziane obiekty procesowane, jeśli robot ich bezpośrednio nie dotyka, pozostają w bezruchu względem kamery, a przez to nie zmienia się ich obraz. Przypadek z kamerą zamontowaną na kiści jest w stanie zapewnić więcej użytecznych ujęć, ale przez to jest też bardziej skomplikowany do zaimplementowania.



Rysunek 2: Przykład zadania paletyzacji jako manipulacji przy braku percepcji. Po lewej widoczny jest kod programu stworzony w języku MELFA-BASIC IV. Po prawej na górze - ujęcie wykonane podczas pracy robota Kawasaki, na dole - widok palety do ukończeniu zadania. Symulacja została wykonana w środowisku Cosimir.



Rysunek 3: Przykład zadania *bin picking* wykonywanego przez robota KUKA w oparciu o dane z systemu wizyjnego. Widoczne po lewej obrazy zostały wygenerowane sztucznie jako hipotetyczne dane z kamery umieszczonej nieruchomo względem bazy robota. Od góry - ujęcie powstałe jako bezpośrednią projekcję, obraz niosący dane o głębi i obraz poddany segmentacji. Symulacja została stworzona w języku Python z wykorzystaniem biblioteki *pybullet*.

## 1.2. Cele manipulacji

Chociaż pierwsze w pełni sprawne manipulatory istnieją w sektorze industrialnym od dekad, wykorzystanie ich w zadaniu manipulacji stanowi wyzwanie, ponieważ często stosowane rozwiązania nie wykorzystują w pełni ich potencjału. Dlatego chcąc określić potrzebę implementacji omawianych tu rozwiązań warto jest przyjrzeć się celom przed, którymi staje współczesna robotyka. Przede wszystkim chcemy, aby dane rozwiązanie było możliwe jak najbardziej **uniwersalne**, to znaczy działało poprawnie w różnych środowiskach, było odporne na zaszumienia i efektywne. Aby te cechy mogły zostać spełnione należy pewne kwestie muszą zostać rozwiązane, a należą do nich:

- **Zdolność do operowania wieloma różnymi obiekttami.** Uniwersalność może zostać osiągnięta przez adaptację, czyli trwający w czasie proces, którego celem jest

przystosowanie danego podmiotu do obserwowanego środowiska. Algorytmy działające zgodnie z tą zasadą stają się dość popularne szczególnie w ostatnich latach, a techniki wiodące prym na tym polu określane są mianem *Reinforcement Learning* (w języku polskim przyjęły się nazwy uczenia ze wspomaganiem i uczenia z krytykiem). Nie zawsze stanowią jednak wydajne rozwiązań zadania, głównie ze względu na wspomniany wyżej czas potrzebny do przetrenowania algorytmu. Problematyczne mogą być również dość chaotyczne, a przez to nieoptymalne i trudne do przewidzenia trajektorie jakie algorytmy tego typu generują.

- **Sterowanie we wszystkich stopniach swobody.** Manipulator przemysłowy wykorzystany w tej pracy charakteryzuje się sześcioma stopniami swobody (bez chwytaka), co sprawia, że dla niektórych pozycji zadanie kinematyki odwrotnej będzie miało więcej niż jedno rozwiązanie i wiele możliwych trajektorii, z których nie wszystkie są optymalne.
- **Odporność na zaszumione lub zniekształcone dane wejściowe.** Ograniczona rozdzielcość kamer RGBD, zaszumienie spowodowane światłem zewnętrznym i różne możliwe rozrzucenie obiektów komplikuje działanie algorytmu i aby proces przebiegał płynnie - musi zostać obsłużone.

Do powyższej listy moglibyśmy także dopisać zdolność do przenoszenia zachowań z symulacji do rzeczywistego świata, ale ponieważ nie we wszystkich problemach stosuje się metody symulacji - nie dopisywałem jej.

### 1.3. Afordancje

Zdolność oddziaływania na jakiś obiekt lub środowisko została określona mianem **afordancji** przez psychologa Jamesa J. Gibsona w 1966 roku w książce *The Senses Considered as Perceptual Systems* i opisana w artykule *The Theory of Affordance. In: Perceiving, Acting and Knowing Toward an Ecological Psychology*. Definicja ta przyjęła szerokie znaczenie w dziedzinach tj. psychologia, kognitywistyka, sztuczna inteligencja czy robotyka. Możemy zdefiniować afordancje na przykładzie ludzi, ale człowiek ze względu na swoją złożoność często pozostaje niewdzięcznym modelem. Dlatego dla naszych rozwia-

żań posłużymy się morskim bezkręgowcem - krabem *Limulus polyphemus*. Przykład ten został opisany m.in. w *Biocybernetycy* Ryszarda Tadeusiewicza. Korzystnymi cechami tego stawonoga z naszego punktu widzenia są posiadanie ośrodkowego układu nerwowego oraz oczu, z których impulsy jesteśmy w stanie śledzić. Tak więc, jesteśmy w stanie w przybliżeniu zobrazować jego pole widzenia.

Co wobec tego widzi krab? Przede wszystkim **obiekty** stanowiące najczęściej zagrożenie lub będące przedmiotem łowów niezależnie od tła. W warunkach laboratoryjnych możemy także osiągnąć pobudzenie neuronalne za pomocą punktowego oświetlenia lub gwałtownych zmian światła. Krab nie widzi natomiast równomiernych gradientów oświetlenia czy monotonnego pastelowego tła. Jest to biologicznie uzasadnione, ponieważ informacje te nie są dla niego w żaden sposób niezbędne do życia. Presja ewolucyjna wywierana na mózgi zwierząt kształtowała się z najbardziej podstawowych powodów: żeby umożliwić im lepszą zdolność poruszania się i odpowiednią zdolność postrzegania otoczenia. Te umiejętności znajdowały odzwierciedlenie w najbardziej fundamentalnych cechach życiowych, czyli poszukiwaniu pożywienia i schronienia oraz ucieczki przed zagrożeniem. Dlatego jeśli prześledzimy drogę ewolucji prowadzącą do prostych bezkręgowców tj. glisty czy małże, zauważymy, że ich ośrodkowy układ nerwowy odpowiada głównie za kontrolę ruchu przez pobudzanie właściwych mięśni. A ponieważ nawet u najbardziej prymitywnych zwierząt ruch wykształcał się w parze ze zdolnością postrzegania środowiska - receptory reagujące na sygnały zewnętrzne (substancje chemiczne lub światło) przesyłają impulsy elektryczne do nerwów odpowiedzialnych za poruszanie się.

Przykład kraba, mimo że może wydawać się surowy, niesie ze sobą istotne wskaźówki na temat tego co powinniśmy rozumieć przez afordancje i jaka jest ich rola. Nikt dokładnie nie wie jak wyglądała droga rozwoju układu nerwowego prowadząca do zorganizowania jego struktury w mózg dorosłego człowieka, w którym ponad połowa neuronów odpowiada za kontrolę motoryczną i zmysły. Aby mieć władzę nad ciałem, mózg tworzy pewne odwzorowanie, które "mapuje" poszczególne części ciała na powierzchni kory mózgowej w miejscu zwojów *gyrus postcentralis* (pół Brodmanna 1, 2 i 3). Mając taką mapę możemy zauważać, że łączna powierzchnia dłoni i twarzy zajmuje na niej znacznie więcej miejsca niż pozostała reszta ciała. Innymi słowy, czynności wy-

konywane twarzą i dłońmi, tj. mimika, mowa czy manipulacja, angażują znaczne ilości neuronów co stanowi pewną miarę ich skomplikowania. Ostatnie badania stosujące techniki neuroobrazowania sugerują, że umiejętność "obchodzenia się z narzędziami", czyli zdolność do projektowania, planowania i korzystania z narzędzi wyewoluowała w lewej półkuli mózgu. Dlatego też pacjenci, u których zdiagnozowano uszkodzenia tej części, mimo że potrafią rozpoznać dany przedmiot, nie potrafią sobie wyobrazić jak go użyć i nie radzą sobie w tej kwestii lepiej niż szymbansy.

Tak więc, aby mogło zostać zrealizowane chwytanie - wykorzystywane są informacje zawarte w obrazie. Pozwala to tłumaczyć dlaczego niektóre gatunki, przykładem tu mogą być wczesne homidy, wykształciły bardziej rozwinięty wzrok konieczny do wytypowania elementów otoczenia. Podobnie w robotyce, robot musi nauczyć się jak chwytać i operować uchwyconymi obiekty, tak aby osiągnąć wyznaczony cel. Różne przedmioty, np. młotek mogą być uchwycone na wiele różnych sposobów natomiast liczba optymalnych chwytów jest ograniczona w kontekście danego zadania.

## 1.4. Zarys pracy

Mając wprowadzenie za sobą możemy skupić się na wymienieniu kolejnych etapów pracy. Głównym jej dążeniem jest znalezienie znanego przedmiotu i uchwycenie go za pomocą manipulatora. Aby to osiągnąć najpierw możemy rozbić ten problem na mniejsze, do których należą estymacja położenia i orientacji obiektu, zadanie kinematyki odwrotnej oraz chwytanie.

- **Estymacja położenia i orientacji obiektu**

Do rozwiązania tego zadania zostanie zaproponowana metoda inteligencji obliczeniowej znana pod nazwą ICP (ang. *Iterative Closest Point*). W oparciu o model obiektu znajduje ona transformację jakiej ten model trzeba poddać, aby znalazł się w pozycji szukanego przedmiotu. Jej szczegółowe działanie zostanie szerzej opisane w rozdziale czwartym. Zostaną tam wyłożone matematyczne podstawy algorytmu, jego implementacja oraz jak to wygląda w praktyce.

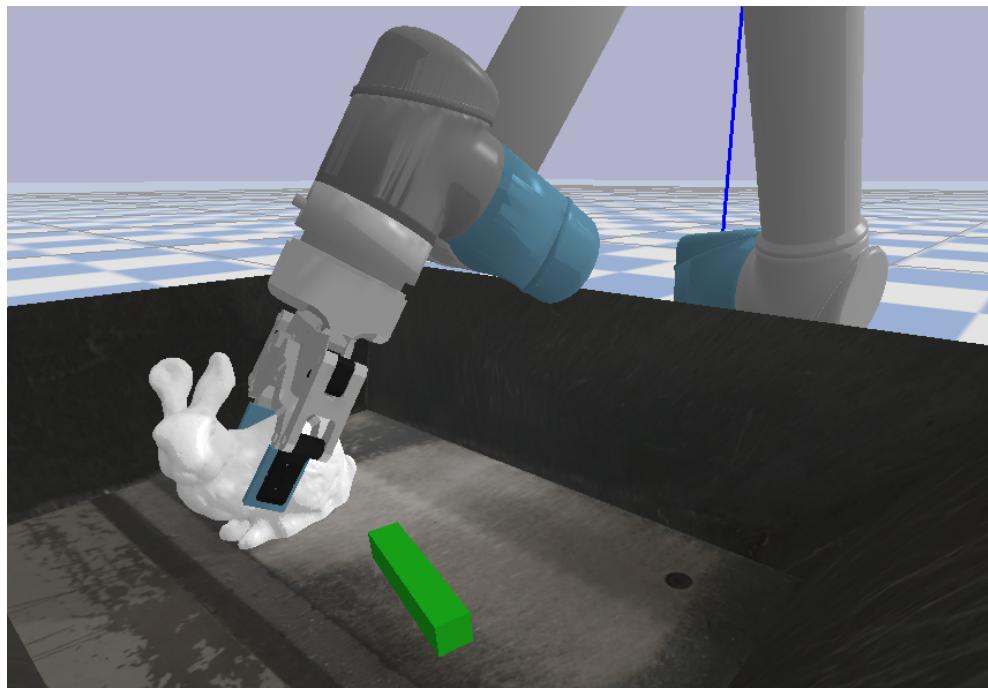
- **Zadanie kinematyki odwrotnej**

Problem znalezienia odpowiednich wielkości opisujących manipulator jako łańcuch

kinematyczny dzielimy na dwa mniejsze: zadanie kinematyki prostej i odwrotnej. Proste zadanie kinematyki odpowiada na pytanie w jakim miejscu znajduje się efektor końcowy dla danej pozy manipulatora. Natomiast kinematyka odwrotna zakłada, że znane jest docelowe położenie efektora końcowego a szukana jest pozycja (lub pozycje) robota, które pozwolą mu je osiągnąć, dlatego też zostanie wykorzystana w tej pracy. Mając na uwadze, że większość robotów przemysłowych posiada wbudowaną implementację kinematyki odwrotnej nie będziemy wchodzić w jej szczegóły.

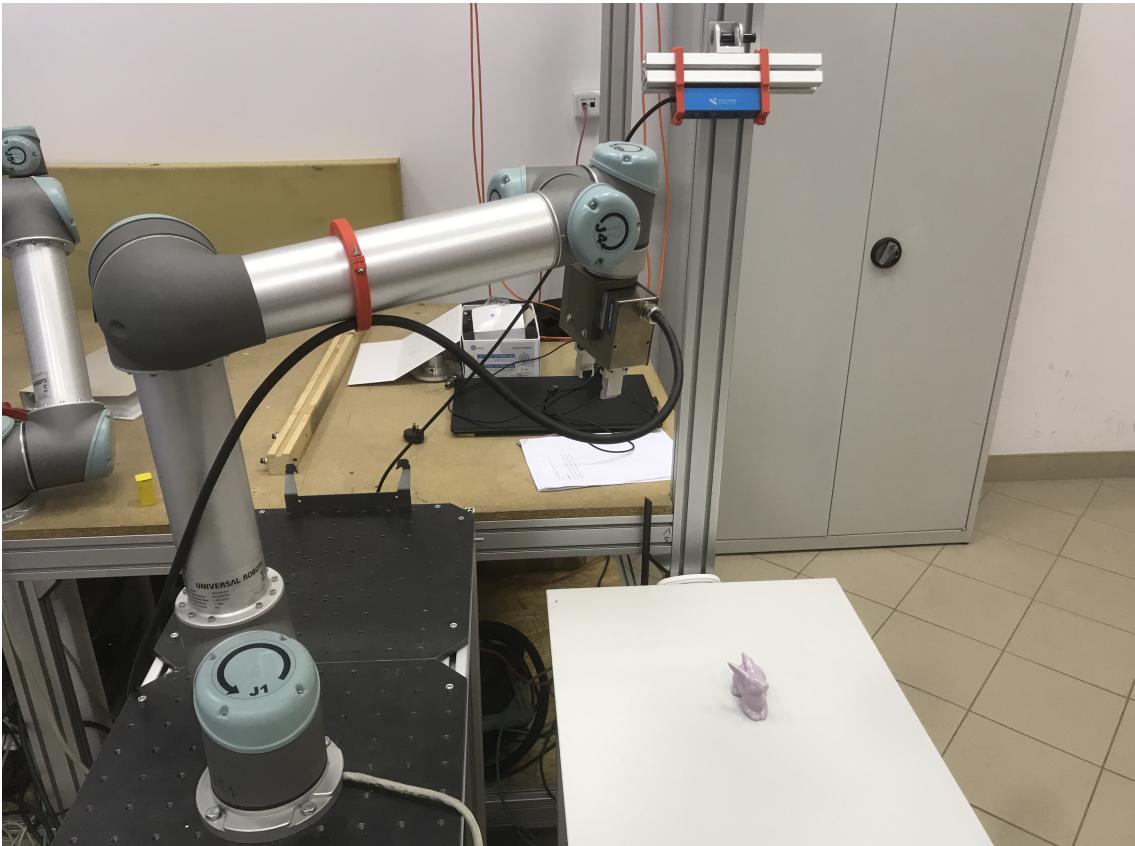
- **Chwytyanie**

Do realizacji chwytyania posłużymy się chwytkiem, którego orientacja wobec chwytanego przedmiotu zostanie dobrana z góry. Stanowić to może pewien mankament, ponieważ działanie uzależniane jest tym samym od ingerencji człowieka. Z drugiej strony, daje to większe prawdopodobieństwo pewniejszego i bardziej stabilnego chwytu, który może okazać się kluczowy w praktycznych zastosowaniach. Taki nadzór pozwala również stwierdzić czy stosowany chwytek jest odpowiedni i prawidłowo spełnia swoją rolę. Realizacja chwytu manipulatorem UR5 zostanie pokazana pod koniec rozdziału czwartego.



Rysunek 4: Ręczne zorientowanie szczęk chwytaka robota UR5 wobec obiektu. Wizualizacja została wykonana na potrzeby pracy z użyciem biblioteki *pybullet*.

Zanim przejdziemy do przedstawienia proponowanego rozwiązania przez rozwinięcie powyższych kwestii zostanie zrobiony przegląd najważniejszych obecnie stosowanych metod. Następnie będzie miał miejsce teoretyczny opis pewnych zagadnień. Zostaną pokazane matematyczne podstawy algorytmu *Iterative Closest Point* i jego implementacja. Pod koniec zostanie zrobione podsumowanie, podczas którego będą wyciągnięte wnioski odnośnie działania proponowanego rozwiązania oraz możliwości dalszego rozwoju.



Rysunek 5: Stanowisko z robotem UR5. Na niebiesko jest widoczna umieszczona pod kątem kamera *Intel RealSense* skierowana na stół.

## 2. Przegląd dotychczasowych rozwiązań

Problem sterowania robotami z wykorzystaniem informacji zwrotnej z otoczenia sięga początków współczesnej robotyki. Chcąc zapewnić możliwość pracy robota w środowiskach, w których nie mamy wystarczającej wiedzy na temat położenia obiektów z jakimi ma wchodzić w interakcje, konieczne jest zaimplementowanie podstawowej percepji. Najprostszy przypadek może stanowić robot wyposażony w czujnik koloru w zadaniu sortowania kolorowych kulek. Wiele rozwiązań jednak opiera się na wykorzystaniu danych i przetworzeniu obrazu z kamery i czujników w taki sposób, aby robot był w stanie rozpoznać lub zlokalizować wskazane obiekty. W tym rozdziale przyjrzymy się obecnie stosowanym rozwiązaniom, które wpisują się w popularne trendy na tym polu. Aby wprowadzić pewien porządek, wyodrębnione zostały trzy najbardziej istotne grupy: widzenie maszynowe, sieci neuronowe oraz uczenie przez wzmacnianie. Ponieważ są to tematy dosyć obszerne, uwaga zostanie skupiona na podstawowych zasadach ich działania bez szczególnego wchodzenia obliczenia i metody matematyczne, które za nimi stoją - takie rozważania wychodziły poza temat tej pracy. Nabycie takiej intuicji okaże się pomocne przy późniejszej ocenie proponowanego rozwiązania.

### 2.1. Widzenie maszynowe

Analizę i przetwarzanie obrazu nazywany widzeniem maszynowym lub krócej CV od angielskich słów *Computer Vision*. Stało się ono popularne szczególnie na początku lat '00 wraz ze wzrostem wydajności obliczeniowej ówczesnych komputerów. Wyróżniamy kilka typów problemów związanych z obrazami, takich jak: detekcja, lokalizacja i segmentacja.

- **Detekcja** dostarcza nam informację czy wskazany obiekt znajduje się na obrazie, a także w niektórych przypadkach, otrzymujemy prawdopodobieństwo jego wystąpienia. Nie zapewnia nam jednak informacji, gdzie dokładnie ten obiekt się znajduje.
- **Lokalizacja** często pojawia się w parze z detekcją. Wynika to z faktu, że gdy mamy rozpoznany dany obiekt, chcemy pozyskać także informację o jego położeniu, co otrzymujemy w postaci współrzędnych lokalizacji, a czasem także wielkości.

- **Semantyczna segmentacja** niesie nam odpowiedź o położeniu oraz kształcie obiektów. Założymy, że mamy trzy różne obiekty w puli do rozpoznania i wszystkie znajdują się na jednym obrazie, z czego dwa z nich są na nim po jednej sztuce, a ostatni – w dwóch. Semantyczna segmentacja umożliwia nam znalezienie położenia ich wszystkich, natomiast nadal nie wiemy czy jakiś z nich nie wystąpił w więcej niż jednej sztuce, przez co niemożliwe jest określenie ilości.
- **Segmentacja z uwzględnieniem instancji** powstała jako rozwiązanie problemu opisanego powyżej. Niesie nam odpowiedź, która oprócz podania dokładnego położenia i kształtu obiektów także uwzględnia ich ilość.

Wymienione wyżej problemy należą do dość złożonych zadań wymagających sporej mocy obliczeniowej. Z pomocą przychodzą programy takie jak SIFT i algorytm Violi-Jonesa (w celu detekcji i lokalizacji danych obiektów na podstawie podobieństwa cech) czy metody uczenia nienadzorowanego (pomocne przy segmentacji). Często jednak stosowane są sztuczne sieci neuronowe, a szczególnie popularne są sieci zawierające operację splotu, czyli sieci splotowe.

## 2.2. Sztuczne sieci neuronowe

Mówiąc o sieciach neuronowych w kontekście przedmiotów zajmujących się przetwarzaniem sygnałów mamy na myśli struktury matematyczne, które w pewnym stopniu przypominają budowę i działanie ludzkiego mózgu. Są one jedną z najbardziej popularnych technik współczesnej sztucznej inteligencji. Stosujemy je przede wszystkim wtedy, gdy nie znamy reguł rozwiązania danego problemu, ale dysponujemy zbiorem przykładowych zadań (zbiorem uczącym), które zostały poprawnie rozwiązane.

Sztuczne sieci neuronowe składają się z wzajemnie połączonych neuronów, które są uproszczonymi modelami biologicznych odpowiedników. Jest to spowodowane tym, że rzeczywiste neurony są tworami niezmiernie skomplikowanymi i wymagającymi pod względem obliczeniowym. Gdy przedstawimy taką sieć w postaci grafu - jego wierzchołki będą reprezentować neurony, a krawędzie - poszczególne wagie sieci. Jej trening polega na znalezieniu takich wartości wag, które zapewnią poprawne rozwiązanie problemu. Zazwyczaj odbywa się to za pomocą algorytmu wstecznej propagacji błędów (ang. *back-*

*propagation*). Trenując sieć, generalizuje ona wiedzę na podstawie zbioru uczącego, od którego zależy jej późniejsze działanie.

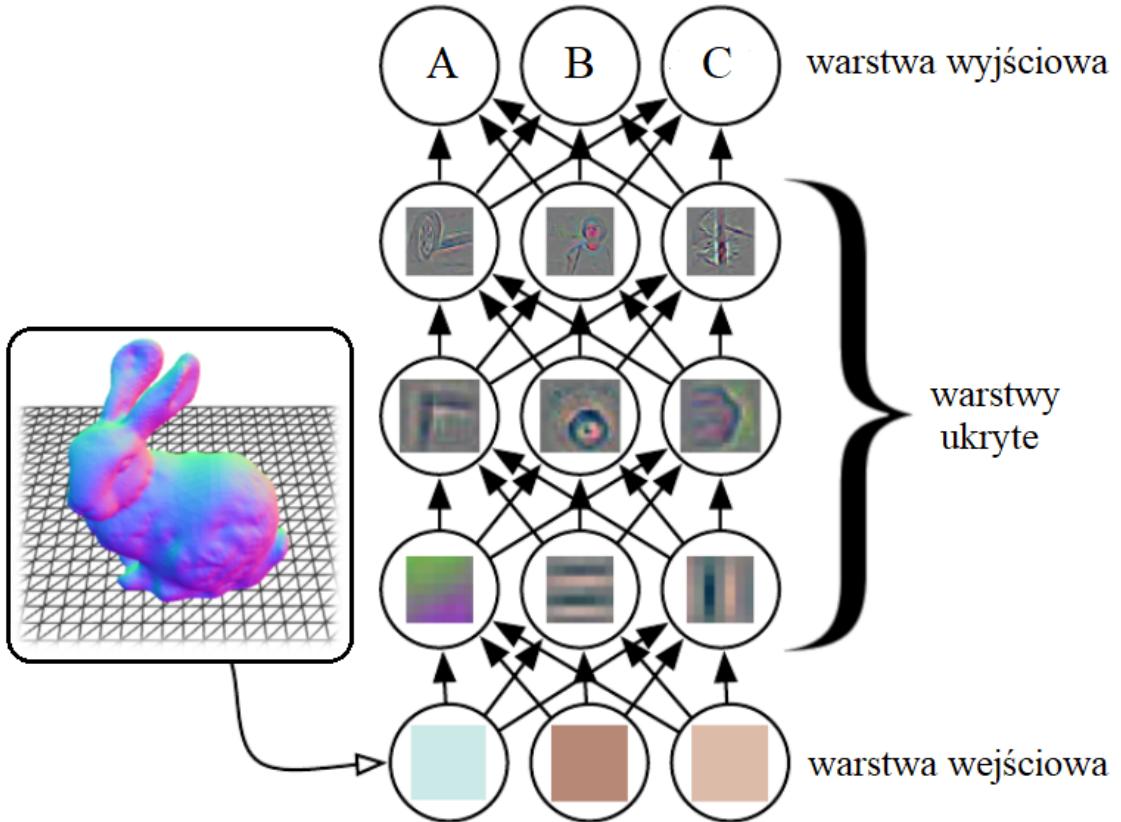
### 2.2.1. Sieci jednokierunkowe

Sztandarowym przykładem jest tu wielowarstwowa sieć jednokierunkowa, zwana również perceptronem wielowarstwowym lub MLP (ang. *Multi-Layer Perceptron*). Jak sama nazwa wskazuje jest to struktura, w której zostały wyodrębnione **warstwy**, z których pierwszą nazywamy wejściową, środkowe - ukrytymi, a ostatnią - wyjściową. Warstwy te reprezentują kolejne poziomy abstrakcji tworząc model hierarchiczny. Najniższe z nich wykrywają proste cechy sygnału wejściowego, a najwyższe, opierając się o informacje z poprzednich warstw - są w stanie znaleźć pewne coraz bardziej abstrakcyjne cechy.

Posługując się przykładem, założymy, że chcemy przetrenować pięciowarstwową sieć pod kątem zdolności do rozpoznawania obrazu. Przechodząc przez warstwy ukryte w najbardziej powszechnym przypadku pierwsza z nich wykryje krawędzie na podstawie różnic kolorów między pikselami, druga - korzystając z informacji o krawędziach będzie w stanie znaleźć kontury i zaokrąglenia, trzecia - pewne relacje pomiędzy nimi. Na ich podstawie z większym prawdopodobieństwem będzie mogła stwierdzić do jakiej klasy należy widziany obraz. W ten sposób, najczęściej w postaci rozkładu prawdopodobieństwa, dostaniemy informację o rozpoznanym obiekcie.

### 2.2.2. Sieci splotowe

Rodzaj sieci, na który szczególnie warto jest zwrócić uwagę stanowią sieci splotowe zwane częściej sieciami konwolucyjnymi, w skrócie CNN (ang. *Convolutional Neural Network*). Te sieci zawdzięczają swoją popularność przede wszystkim dzięki wykorzystaniu do problemów związanych z analizą obrazów, przy których ich możliwości przewyższają zwykłe sieci jednokierunkowe. Ich nazwa wzięła się od operacji splotu (konwolucji), która jest wykorzystywana do ekstrakcji cech przez splot obrazu z filtrem. Sieci splotowe są z powodzeniem stosowane w rozpoznawaniu, klasyfikacji i segmentacji obrazów. Ponieważ możliwe jest przeprowadzenie operacji splotu na danych o zarówno jednym, dwóch oraz trzech wymiarach, użycie sieci CNN da się poszerzyć do analizy danych takich jak



Rysunek 6: Idea działania sieci neuronowej w zadaniu rozpoznawania obiektu. Przedstawiona została sieć składająca się z pięciu warstw: wejściowej, trzech ukrytych i wyjściowej. Na wejście trafiają dane, np. obraz lub chmura punktów, którą chcemy rozpoznać. Następnie kolejne warstwy ukryte wydobywają z nich coraz bardziej abstrakcyjne cechy. Pod koniec, informacja o przynależności do danej klasy A, B lub C jest zwracana na wyjściu sieci. Obraz został zaczerpnięty z książki *Deep Learning: Systemy uczące się*.

dźwięk, obrazy RGBD czy chmury punktów, a to z kolei implikuje ich przydatność w robotyce.

### 2.2.3. GraspNet

Daleko idącym przykładem wykorzystania sieci neuronowych w robotyce jest projekt *GraspNet*. Celem przedsięwzięcia było przetrenowanie modelu sztucznej inteligencji na zbiorze obrazów pochodzących z kamer RGBD przedstawiających przedmioty z możliwymi chwytami wyznaczonymi analitycznie. Twórcom projektu udało się zgromadzić 97280 obrazów RGBD zawierających ponad miliard możliwych pozycji chwytu. Zbiór ten został ujawniony i opublikowany pod nazwą *GraspNet-1Billion*. W rzeczywistości *GraspNet* jest modelem składającym się z kilku mniejszych sieci, który oprócz nich

wykorzystuje funkcje tj. grupowanie, wyśrodkowywanie i filtrowanie danych. Tak więc, dla danych wejściowych w postaci chmury punktów przyporządkowuje pewną ilość chwytów jakie mogą zostać zastosowane podczas próby uchwycenia obiektów w niej przedstawionych. Możliwe chwyty zostają wyznaczone w sposób, na który mogą zostać przeprowadzone z użyciem chwytyaka dwupalcistego.



Rysunek 7: Wizualizacja możliwych chwytów wykrytych przez sieć *GraspNet*. Na niebiesko oznaczone jest położenie szczęk chwytyaka.

#### 2.2.4. Zalety sieci neuronowych

Oprócz zdolności do rozwiązywania problemów, które nie są efektywnie algorytmizowalne w oparciu o matematyczne modelowanie wiedzy, do zalet sieci neuronowych możemy zaliczyć ich wszechstronność. To znaczy, dana struktura sieci może zostać przetrenowana do wielu różnych niezwiązanych ze sobą zadań. Jest to w pewnym sensie cecha wspólna wszystkich technik uczenia maszynowego wyróżniająca je na tle innych algorytmów.

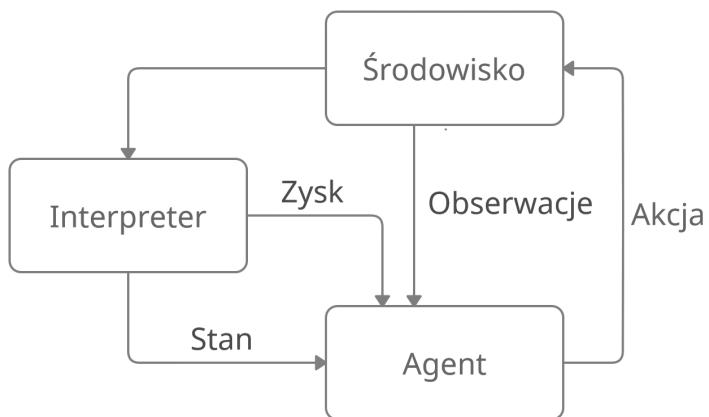
#### 2.2.5. Wady sieci neuronowych

Sieci neuronowe mają wiele wad - trenowane algorytmem *backpropagation* dokonują coraz mniejszych zmian w kolejnych warstwach idących wgłąb sieci (problem zanikającego gradientu) oraz łatwo przeuczają się, ponieważ większa liczba zmieniających się podczas uczenia sieci parametrów sprzyja przetrenowaniu. Skutkiem tego jest mniejsza wydajność proces ekstrakcji cech występujących w zbiorze uczącym. Trenując sieci neuronowe, w szczególności duże modele, nie do końca wiemy w jaki sposób przetwarzają one informacje. Pociąga to za sobą wiele implikacji, a patrząc z technicznej perspektywy,

często nie jesteśmy pewni czy zastosowany przez nas model sieci nie mógłby zostać uproszczony. Gdybyśmy dysponowali większą wiedzą na temat przetwarzania przez nie informacji, najprawdopodobniej możliwa była by redukcja czasu, danych i zasobów obliczeniowych potrzebnych do ich uczenia. Są to najbardziej istotne czynniki wpływające na jakość działania modelu.

### 2.3. Uczenie przez wzmacnianie

Uczenie przez wzmacnianie zwane też uczeniem z krytykiem lub *Reinforcement Learning* opisuje metody uczenia maszynowego, w którym optymalne rozwiązanie danego zadania powstaje na skutek interakcji agenta ze środowiskiem. W zastosowaniach tego podejścia w robotyce, środowiskiem jest symulacja lub rzeczywiste otoczenie, z którym agent (robot) wchodzi w interakcję. Celem jest maksymalizacja zysku ściśle związanego z pożądanym rezultatem. W ten sposób robot próbując osiągnąć jak najwyższy wynik uczy się rozwiązywać zadany problem.



Rysunek 8: Przedstawienie zasady działania metod *Reinforcement Learning* w postaci grafu: agent podejmuje działanie w danym środowisku co jest interpretowane jako zysk i reprezentacja stanu, które trafiają z powrotem do agenta. W dalej idących implementacjach algorytmu występują także obserwacje, czyli dostarczane agentowi informacje na temat środowiska.

Istnieje wiele wariantów algorytmów wykorzystujących *Reinforcement Learning*, które możemy podzielić ze względu na kryterium optymalizacji na oparte na polityce lub funkcji zmiany stanu.

### 2.3.1. Oparte na strategii

W tym podejściu przyjmowana jest strategia, którą należy zoptymalizować. Strategię możemy definiować jako odpowiedzialną za zachowanie agenta funkcję, która na wejściu przyjmuje obserwację, a na wyjściu zwraca akcję.

### 2.3.2. Oparte na funkcji wartości stanu

Funkcja wartości stanu  $V_\pi(s)$  określa jak wysoki będzie zysk, gdy zaczniemy ze stanu  $s$  i będziemy opierać się o politykę  $\pi$ .

$$V_\pi(s) = \mathbb{E}[R|s_0 = s] = \mathbb{E} \left[ \sum_{t=0}^n \gamma^t r_t | s_0 = s \right]$$

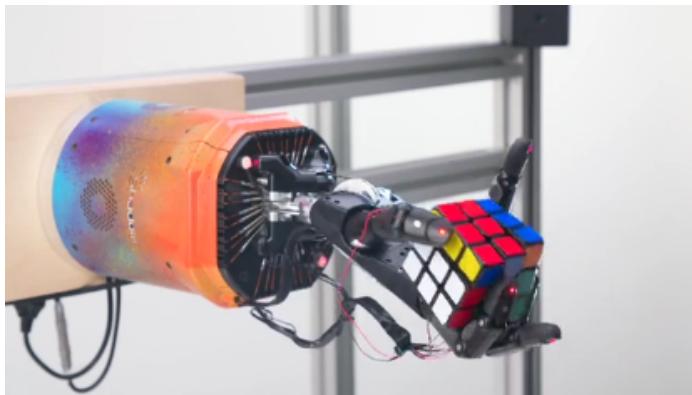
W tym kryterium zostaje wprowadzona wartość  $R$  jako całkowity **zysk**, który jest zdefiniowany jako suma poszczególnych przyszłych zysków

$$R = r_0 + \gamma r_1 + \gamma^2 r_2 + \dots + \gamma^n r_n = \sum_{t=0}^n \gamma^t r_t \quad \text{gdzie } \gamma \in \langle 0, 1 \rangle$$

Ponieważ  $\gamma$  jest mniejsza niż 1, odległe przyszłe zdarzenia są mniej istotne od zdarzeń w bliskiej i niedalekiej przyszłości. Jest to istotne, ponieważ zależy nam na najszybszym możliwym uzyskaniu zysku.

### 2.3.3. Deep Reinforcement Learning

Związek uczenia przez wzmacnianie z robotyką stale się zacieśnia, zwłaszcza popularne w ostatnich latach stały się techniki wykorzystujące głębokie uczenie przez wzmacnianie, czyli *Deep Reinforcement Learning*. Jest to w pewnym sensie połączenie dwóch omawianych wyżej metod: sieci neuronowych i *Reinforcement Learningu* stosowane w sytuacjach, gdy mamy do czynienia z bardziej złożonymi problemami. Posługując się konkretnym przykładem, założmy że chcemy, aby robot na podstawie obrazu chmury punktów pochodzącej z kamery RGBD znalazł zadany przedmiot i go uchwycił. Jest to problem, na którym skupia się ta praca, szerzej znany pod nazwą *bin picking*. Wówczas, gdy mamy wielowymiarowe obserwacje w postaci chmury punktów - niemożliwe się staje rozwiązanie go za pomocą tradycyjnego *Reinforcement Learningu*.



Rysunek 9: Robot wykorzystujący techniki uczenia ze wzmacnianiem podczas układania kostki Rubika. Eksperyment został opisany w publikacji *Solving Rubik's Cube with a Robot Hand* i miał na celu przetrenowanie sztucznej inteligencji pod kątem umiejętności motorycznych - praca zbiorowa OpenAI.

#### 2.3.4. Zalety uczenia przez wzmacnianie

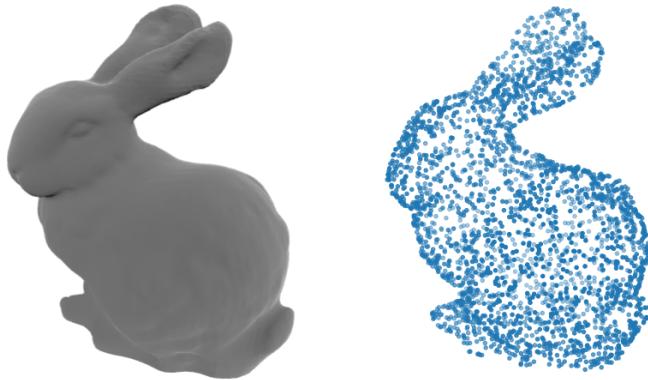
Do niewątpliwych zalet uczenia przez wzmacnianie należy zdolność do tworzenia zachowań **emergentnych**. Przez emergencję rozumiemy powstawanie zachowań, do których instrukcje nie zostały bezpośrednio udzielone, a znalezione metodą prób i błędów i następnie z powodzeniem stosowane. Drugą ważną zaletą jest jego uniwersalność - metody oparte na *Reinforcement Learning* mogą zostać z lepszym lub gorszym skutkiem użyte w większości problemów związanych z motoryką w robotyce, tj. nauka chodzenia, chwytania itp.

#### 2.3.5. Wady uczenia przez wzmacnianie

Jak już zostało wspomniane we wprowadzeniu, do wad podejścia wykorzystującego uczenie przez wzmacnianie w robotyce zaliczamy czas i często zasoby obliczeniowe potrzebne do przetrenowania algorytmu oraz niewydajne i często chaotyczne ruchy, które są jego rezultatem.

### 3. Estymacja położenia obiektu

Aby przejść do zadania estymacji położenia obiektu najpierw należy określić czy dysponujemy jego kształtem. Ponieważ w wielu zastosowaniach industrialnych przedstawionego algorytmu cecha ta jest wiadoma z góry, dla potrzeby tej pracy możemy przyjąć, że posiadamy taki model. Mamy więc podane dwa kształty: rzeczywisty model obiektu pobrany z kamery RGBD jako chmura punktów oraz wyidealizowany model, który posłuży nam za szablon. Oba są reprezentowane jako zbiory punktów, z których każdy opisany jest przez współrzędne  $x$ ,  $y$ ,  $z$ . W tym rozdziale omówiony szerzej zostanie problem estymacji położenia, zanim jednak do tego przejdziemy skupimy się na podstawowych założeniach i sposobach, które do niego prowadzą. Zostaną opisane także konstrukcje matematyczne, tj. elementarne macierze transformacji, kwaterniony czy rozkład według wartości osobliwych, które pozwolą lepiej zrozumieć temat przewodni pracy.



Rysunek 10: Model obiektu i utworzona sztucznie chmura punktów.

#### 3.1. Wyznaczenie najbliższego punktu

Najbardziej fundamentalny przykład sprowadza się do znalezienia długości odcinka łączącego dwa punkty  $\mathbf{p}_1 = (x_1, y_1, z_1)$  i  $\mathbf{p}_2 = (x_2, y_2, z_2)$ . Odległość ta jest dana w metryce euklidesowej wzorem

$$d(\mathbf{p}_1, \mathbf{p}_2) = \|\mathbf{p}_1 - \mathbf{p}_2\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Kolejnym krokiem ku estymacji położenia jednego obiektu względem drugiego wydaje się być przeprowadzenie tej operacji dla wszystkich punktów. W tym miejscu

nasuwa się jednak pewien problem. Mianowicie, nie tylko nie wiemy czy dany punkt ma swojego odpowiednika, ale też czy ich liczba w obu zbiorach się zgadza. W rzeczywistości próbujemy znaleźć najkrótszą drogę między punktami zbioru A a punktami zbioru B do czego posłuży algorytm najbliższego sąsiada.

### 3.2. Algorytm najbliższego sąsiada

Algorytm najbliższego sąsiada (ang. *Nearest Neighbour algorithm*) jest jedną z najbardziej podstawowych technik wykorzystywanych do rozpoznawania wzorców we współczesnym uczeniu maszynowym. Jest to algorytm zachłanny o złożoności czasowej  $O(n^2)$  o czym świadczy zagnieżdżenie dwóch pętli. Może zostać wykorzystany zarówno do klasyfikacji, regresji jak i do rozwiązania problemu komiwojażera.

```
1 def knn(points, p, k=1):
2     distances = []
3     for idx in points:
4         for coord in points[idx]:
5
6             dist = math.sqrt((coord[0] - p[0])**2
7                               + (coord[1] - p[1])**2)
8             distances.append((dist, idx))
9
10    distances = sorted(distances)[:k]
11    return distances
```

Opisując powyższy kod, dana jest funkcja *knn* przyjmująca trzy argumenty: zbiór punktów w postaci listy, punkt *p* oraz liczbę najbliższych sąsiadów domyślnie ustawioną na jeden. Działanie funkcji zaczyna się od zdefiniowania pustej listy, w której będą przechowywane poszczególne odległości. Poniżej znajduje się zagnieżdżenie dwóch pętli, z których pierwsza iteruje po punktach, a druga - po ich współrzędnych. W linijkach 6-7 liczona jest odległość euklidesowa między danym punktem w zbiorze, a punktem *p*, która następnie zapisywana jest do listy *distances*. Po wyjściu z obu pętli, lista ta jest sortowana i zwracana na wyjściu funkcji.

### 3.3. Transformacje

Mianem transformacji określana jest funkcja przekształcająca jeden obiekt w drugi. Ze względu, że przez cały czas trwania tej pracy będziemy rozważać przedmioty jako bryły sztywne - omówione zostaną jedynie transformacje, które nie zmieniają kształtu ani rozmiaru obiektu.

$$\mathbf{R} = \begin{bmatrix} R_{xx} & R_{xy} & R_{xz} \\ R_{yx} & R_{yy} & R_{yz} \\ R_{zx} & R_{zy} & R_{zz} \end{bmatrix} \quad \mathbf{t}^\top = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

#### 3.3.1. Macierz rotacji

W zależności od tego jakim sposobem chcemy wykonywać działania mamy do wyboru trzy równoważne opcje obrotu danego kształtu w przestrzeni. Możemy w tym celu posłużyć się macierzą opartą o kąty obrotu wokół wszystkich trzech osi w układzie kartezjańskim, macierzą opartą o oś obrotu i kąt o jaki dany obiekt ma zostać obrócony lub kwaternionem. Dla zadanej rotacji wszystkie trzy sposoby z większą lub mniejszą dokładnością zwrócią numerycznie te same wyniki.

#### 3.3.2. Elementarne macierze rotacji

Najprostszym sposobem na obrócenie obiektu w przestrzeni trójwymiarowej jest skorzystanie z elementarnych macierzy transformacji. Jest to podejście, którego używamy gdy dane są trzy kąty położone na prostopadłych płaszczyznach, wokół których należy obrócić obiekt. kąty Eulera. Czasami można spotkać się także z nazwami *pitch-roll-yaw* oznaczającymi to samo. Obrót o każdy taki kąt możemy zapisać w postaci macierzy

$$\mathbf{R}_z(\gamma) = \begin{bmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{bmatrix}$$

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix}$$

Gdy chcemy dokonać rotacji wokół więcej niż jednej osi składamy poszczególne obroty mnożąc je przez siebie. Mamy więc trzy obroty, które składają się na wynikową rotację.

$$\mathbf{R} = \mathbf{R}_z(\gamma)\mathbf{R}_y(\beta)\mathbf{R}_x(\alpha)$$

Po przemnożeniu przez siebie wszystkich trzech obrotów składowych macierz rotacji przyjmuje postać

$$\mathbf{R} = \begin{bmatrix} \cos \gamma \cos \beta & \cos \gamma \sin \beta \sin \alpha - \sin \gamma \cos \alpha & \cos \gamma \sin \beta \cos \alpha + \sin \gamma \sin \alpha \\ \sin \gamma \cos \beta & \sin \gamma \sin \beta \sin \alpha + \cos \gamma \cos \alpha & \sin \gamma \sin \beta \cos \alpha - \cos \gamma \sin \alpha \\ -\sin \beta & \cos \beta \sin \alpha & \cos \beta \cos \alpha \end{bmatrix}$$

### 3.3.3. Oś obrotu i kąt

Podczas rozpisywania złożonych transformacji za pomocą elementarnych macierzy rotacji możemy zauważyc, że w rzeczywistości zamiast obracać obiekt trzykrotnie wokół różnych osi - moglibyśmy tak obrać jedną oś i kąt obrotu wokół niej, aby wynikowa rotacja pozostała tożsama. Dlatego w tym podejściu zakładamy, że dysponujemy taką osią i wykonujemy już tylko jeden obrót. Oś obrotu możemy przedstawić w postaci wektora jednostkowego  $\mathbf{u} = [u_x, u_y, u_z]$  takiego, że

$$u_x^2 + u_y^2 + u_z^2 = 1$$

Dla ułatwienia oznaczmy  $c = \cos \theta$  i  $s = \sin \theta$

$$\mathbf{R} = \begin{bmatrix} c + u_x^2(1 - c) & u_x u_y(1 - c) - u_z s & u_x u_z(1 - c) + u_y s \\ u_y u_x(1 - c) - u_z s & c + u_y^2(1 - c) & u_y u_z(1 - c) + u_x s \\ u_z u_x(1 - c) - u_y s & u_z u_y(1 - c) - u_x s & c + u_z^2(1 - c) \end{bmatrix}$$

### 3.3.4. Kwaterion

Podejście wykorzystujące kwaterion różni się od dwóch powyższych głównie tym, że nie są tu używane funkcje trygonometryczne. Jest też pod pewnym względem podobne do przypadku wykorzystującego os obrotu i kąt, ponieważ ich obu nie da się rozbić na mniejsze obroty składowe. Same kwateriony są przykładem liczb hiperzespolonych z jedną wartością rzeczywistą i trzema urojonymi. Kwaterion możemy zapisać w postaci sumy algebraicznej jako

$$q = a \cdot \mathbf{e} + b \cdot \mathbf{i} + c \cdot \mathbf{j} + d \cdot \mathbf{k} \quad \text{gdzie } a, b, c, d \in \mathbb{R}$$

Zaś  $\mathbf{e}, \mathbf{i}, \mathbf{j}, \mathbf{k}$  to pewne jednostki urojone, między którymi zachodzi zależność

$$i^2 = j^2 = k^2 = -1$$

Wówczas transformację tą możemy zapisać równoważnie w postaci macierzy

$$\mathbf{R} = \begin{bmatrix} a^2 + b^2 - c^2 - d^2 & 2(bc - ad) & 2(bd + ac) \\ 2(bc + ad) & a^2 + c^2 - b^2 - d^2 & 2(cd - ab) \\ 2(bd - ac) & 2(cd + ab) & a^2 + d^2 - b^2 - c^2 \end{bmatrix}$$

## 3.4. Rozkład macierzy według wartości osobliwych

Celem rozkładu według wartości osobliwych, w skrócie rozkładu SVD (ang. *Singular Value Decomposition*) w proponowanym algorytmie jest znalezienie macierzy rotacji, która stanowi transformację między chmurą punktów a szablonem, do którego próbujemy ją dopasować. Umożliwi to późniejsze uchwycenie przedmiotu przez chwytkę manipulatora, ponieważ szablon jest obiektem, który jest znany, a więc możemy zamodelować poprawny proces chwytania.

Opisany w ten sposób problem stanowi jedynie pewien szczególny przypadek zastosowania rozkładu SVD. Ogólnie rzecz biorąc, SVD jest metodą pozwalającą na znalezienie rzędu macierzy, co teoretycznie może zostać wyznaczone metodą eliminacji Gaussa. To podejście jest jednak niepraktyczne, gdy ma zostać zrealizowane za pomocą metod numerycznych. Błędy biorą się często z niedoszacowań z powodu zaokrąglania

wartości. Na przykład gdy rząd macierzy  $A$  jest niedoszacowany a  $U$  stanowi obliczoną numerycznie postać schodkową, wówczas możliwe jest, że  $U$  będzie miało niepoprawną liczbę niezerowych wierszy.

Najprostszym sposobem na zrozumienie rozkładu według wartości osobliwych jest zapisanie macierzy rzeczywistej  $A$  w postaci iloczynu trzech czynników, z których środkowy to macierz z pewnymi wartościami rosnącymi wzduż przekątnej. Taki rozkład jest zawsze możliwy, co można zapisać jako

$$A = UDV^\top$$

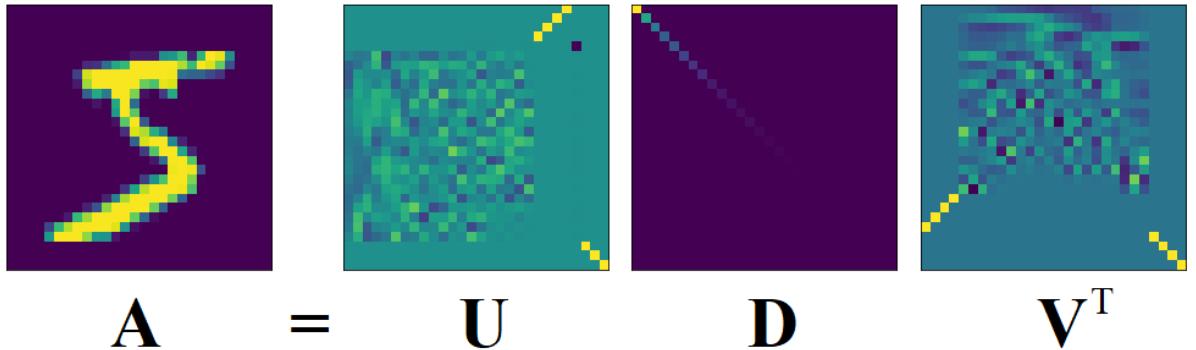
Zakładając, że  $A$  jest macierzą  $m \times n$ , wówczas  $U$  stanie się macierzą ortogonalną o wymiarach  $m \times m$ ,  $V$  - macierzą ortogonalną o wymiarach  $n \times n$ , a  $D$  macierzą  $m \times n$ , której wartości nieleżące na przekątnej są równe 0, a wartości na przekątnej spełniają zależność

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0$$

$$D = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix}$$

Wartości  $\sigma_i$  określone przez rozkład są różne i są nazywane **wartościami osobliwymi** macierzy  $A$ . Liczba niezerowych wartości osobliwych jest równa rzędowi macierzy  $A$ , a ich wielkość stanowi miarę jak bardzo ta macierz różni się od macierzy niższego rzędu. Z kolei kolumny macierzy  $U$  stanowią **lewostronne wektory osobliwe** a kolumny  $V$  - **prawostronne wektory osobliwe**.

Twierdzenie związane z SVD zakłada, że jeśli  $A$  jest macierzą o wymiarach  $m \times n$  możliwy jest jej rozkład według wartości osobliwych. Aby to wykazać, posłużmy się macierzą  $A^\top A$ . Jest to macierz symetryczna, toteż wszystkie jej wartości własne są liczbami rzeczywistymi i istnieje taka macierz ortogonalna  $V$ , która ją diagonalizuje. Co więcej, jej wartości własne muszą być nieujemne. Aby to zobaczyć, niech  $\lambda$  będzie



Rysunek 11: Graficzne przedstawienie zasady działania SVD na przykładzie obrazu ręcznie pisanej cyfry pięć o wymiarach  $28 \times 28$  pikseli. Kolory zbliżone do fioletu reprezentują niskie wartości, a jaskrawe zbliżone ku żółci - wysokie. Możemy zauważyć, że sposób w jakim na obrazie macierzy  $\mathbf{D}$  układają się coraz ciemniejsze kolory jest zgodny z założonym wzorem  $\sigma_i \geq \sigma_{i+1} \geq 0$ .

wartością własną  $\mathbf{A}^\top \mathbf{A}$  a  $\mathbf{x}$  wektorem własnym związanym z  $\lambda$ . Wynika z tego, że

$$\|\mathbf{Ax}\|^2 = \mathbf{x}^\top \mathbf{A}^\top \mathbf{Ax} = \lambda \mathbf{x}^\top \mathbf{x} = \lambda \|\mathbf{x}\|^2$$

Aby wyznaczyć  $\lambda$  z powyższego równania wykonujemy dzielenie

$$\lambda = \frac{\|\mathbf{Ax}\|^2}{\|\mathbf{x}\|^2} \geq 0$$

Podniesienie do kwadratu dowodzi, że  $\lambda$  jest dodatnia. Założymy, że kolumny macierzy  $\mathbf{V}$  zostały uporządkowane tak, że odpowiednie wartości własne spełniają zależność

$$\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n \geq 0$$

Wartości własne macierzy  $\mathbf{A}$  są dane

$$\sigma_i = \sqrt{\lambda_i} \quad i = 1, 2, \dots, n$$

Niech  $r$  oznacza rząd macierzy  $\mathbf{A}$ . Należy zauważyć, że macierz  $\mathbf{A}^\top \mathbf{A}$  też będzie rzędu  $r$ . Ponieważ  $\mathbf{A}^\top \mathbf{A}$  jest symetryczna jej rząd będzie równy liczbie niezerowych wartości własnych.

$$\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_r \geq 0 \quad \lambda_{r+1} = \lambda_{r+2} = \dots = \lambda_n \geq 0$$

Ta sama zależność zachodzi dla wartości osobliwych

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r \geq 0 \quad \sigma_{r+1} = \sigma_{r+2} = \dots = \sigma_n \geq 0$$

Dla naszych dalszych rozważań konieczne jest przedstawienie poszczególnych czynników  $\mathbf{U}$ ,  $\mathbf{D}$  i  $\mathbf{V}$  w rozbiciu na mniejsze macierze, z których są złożone. Zaczynając od macierzy  $\mathbf{V}$  mamy

$$\mathbf{V}_1 = (\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_r), \quad \mathbf{V}_2 = (\mathbf{v}_{r+1}, \mathbf{v}_{r+2}, \dots, \mathbf{v}_n)$$

$$\mathbf{D}_1 = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_r \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} \mathbf{D}_1 & 0 \\ 0 & 0 \end{bmatrix}$$

Wektory  $\mathbf{V}_2$  stanowią wektory własne macierzy  $\mathbf{A}^\top \mathbf{A}$  związane z  $\lambda = 0$

$$\mathbf{A}^\top \mathbf{A} \mathbf{x}_i = 0 \quad i = r+1, r+2, \dots, n$$

$$\mathbf{A} \mathbf{V}_2 = 0$$

Ponieważ  $\mathbf{V}$  jest macierzą ortogonalną możemy zapisać

$$\mathbf{I} = \mathbf{V} \mathbf{V}^\top$$

$$\mathbf{A} = \mathbf{A} \mathbf{I} = \mathbf{A} \mathbf{V} \mathbf{V}^\top$$

Zostało nam jeszcze skonstruować macierz ortogonalną  $\mathbf{U}$  o wymiarach  $m \times m$  taką, że

$$\mathbf{A} = \mathbf{U} \mathbf{D} \mathbf{V}^\top$$

Co po przeniesieniu  $\mathbf{V}$  na lewą stronę możemy równoważnie zapisać jako

$$\mathbf{A} \mathbf{V} = \mathbf{U} \mathbf{D}$$

Porównując pierwsze r kolumn po obu stronach możemy zauważyc

$$\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i \quad i = 1, 2, \dots, n$$

Zatem jeśli, ustalimy

$$\mathbf{u}_i = \frac{1}{\sigma_i} A \mathbf{v}_i \quad i = 1, 2, \dots, n$$

$$\mathbf{U}_1 = (\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_r)$$

To będzie wynikać z tego

$$\mathbf{A}\mathbf{V}_1 = \mathbf{U}_1 \mathbf{D}_1$$

$$\mathbf{u}_i^\top \mathbf{u}_j = \left( \frac{1}{\sigma_i} \mathbf{v}_i^\top \mathbf{A}^\top \right) \left( \frac{1}{\sigma_j} \mathbf{A} \mathbf{v}_j \right) = \frac{1}{\sigma_i \sigma_j} \mathbf{v}_i^\top (\mathbf{A}^\top \mathbf{A} \mathbf{v}_j) = \frac{\sigma_j}{\sigma_i} \mathbf{v}_i^\top \mathbf{v}_j = \delta_{ij}$$

Jeśli  $\mathbf{u}_1, \dots, \mathbf{u}_m$  tworzy bazę ortonormalną dla  $\mathbb{R}^m$  to  $\mathbf{U}$  jest macierzą ortogonalną.

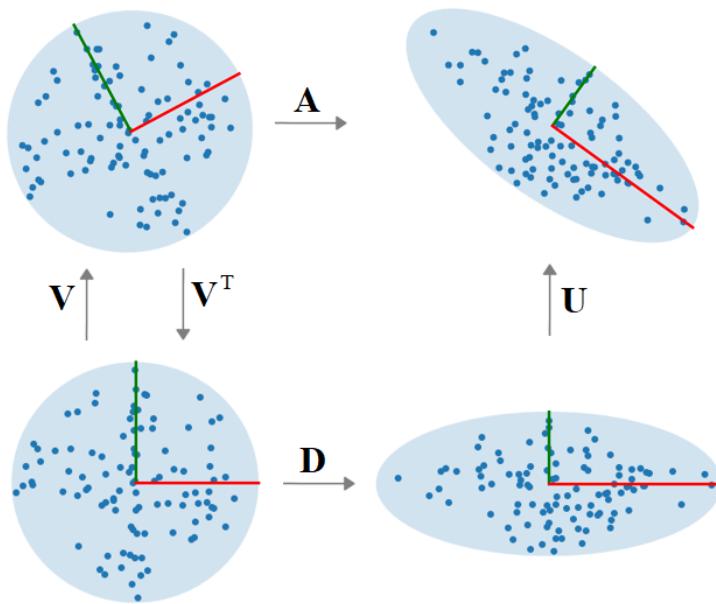
Wówczas ostatnim krokiem jest wykazanie, że  $\mathbf{A}$  rzeczywiście jest równe  $\mathbf{UDV}^\top$

$$\mathbf{UDV}^\top = [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \mathbf{D}_1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{V}_1^\top \\ \mathbf{V}_2^\top \end{bmatrix} = \mathbf{U}_1 \mathbf{D}_1 \mathbf{V}_1^\top = \mathbf{AV}_1 \mathbf{V}_1^\top = \mathbf{A}$$

### 3.5. Interpretacja geometryczna rozkładu SVD

W interpretacji geometrycznej wartości osobliwe mogą być rozumiane jako długości półosi elipsy na płaszczyźnie, co pokazano na rysunku. Wówczas zakładając, że  $\mathbf{A}$  stanowi przekształcenie liniowe przestrzeni - macierze  $\mathbf{U}$  i  $\mathbf{V}$  reprezentują rotacje i odbicie tej przestrzeni, a  $\mathbf{D}$  odpowiada za skalowanie. Innymi słowy, rozkład SVD rozkłada dowolne przekształcenie liniowe na złożenie funkcji trzech przekształceń: obrotu lub odbicia ( $\mathbf{V}$ ), skalowania ( $\mathbf{D}$ ) i kolejnego obrotu lub odbicia ( $\mathbf{U}$ ). W szczególności, jeśli wyznacznik poddowanej przekształcaniu macierzy jest dodatni, wówczas macierze  $\mathbf{U}$  i  $\mathbf{V}$  mogą odpowiadać zarówno za rotację jak i odbicie. Jeśli wyznacznik jest ujemny tylko jedna z nich odpowiada za odbicie, jeśli jest równy zero - odbicia nie ma wcale.

Taka interpretacja może również zostać uogólniona do  $n$ -wymiarowych przestrzeni



Rysunek 12: Interpretacja geometryczna SVD macierzy  $A$  o wymiarach  $2 \times 2$ . Celem jest przekształcenie przestrzeni pokazanej w lewym górnym rogu, tak aby uzyskać obraz widoczny w prawym górnym. Transformacja zostaje rozłożona przez SVD na trzy etapy: rotację, skalowanie i ponowną rotację.

euklidesowych. Wówczas wartości osobliwe dowolnej macierzy kwadratowej  $n \times n$  staną się długościami półosi  $n$ -wymiarowej elipsoidy. Wartości osobliwe zawierają w sobie informację o długościach a wektory osobliwe - o kierunku półosi. Nie jest to jednak zbyt praktyczne rozwiązanie, dlatego mówiąc o rozkładzie SVD w proponowanym algorytmie będziemy posługiwać się uproszczonym SVD. Przede wszystkim, zakładamy, że dysponujemy szablonem, który ma dokładnie te same wymiary co szukany obiekt oraz szukany przedmiot jest bryłą sztywną (więc również jego szablon traktujemy w ten sam sposób). Te założenia sporo upraszczają. Przede wszystkim, odnosząc się do praktycznego zastosowania możemy pominąć operację skalowania.

## 4. Algorytm Iterative Closest Point

Mając teoretyczne podstawy za sobą możemy przejść do implementacji końcowego programu stworzonego w oparciu o algorytm ICP (ang. *Iterative Closest Point*). Dla danej chmury punktów jego celem jest znalezienie położenia i orientacji obiektu, który przedstawia. W tej sytuacji dane wejściowe stanowią: chmura punktów i szablon szukanego obiektu, do którego chcemy dopasować widoczny przedmiot. Przez dopasowanie rozumiemy znalezienie transformacji (obrotu i przesunięcia) wykrytego obiektu względem początkowego położenia szablonu. Możemy rozpisać pojedynczą iterację algorytmu w postaci listy kroków:

1. Wyznaczenie centroidów  $a$  i  $b$  dla obu chmur punktów  $A$  i  $B$ .
2. Utworzenie  $A'$  i  $B'$  przez standaryzację wartości w  $A$  i  $B$ . Standaryzacja odbywa się przez odjęcie od każdego punktu centroidu z całej chmury.
3. Obliczenie macierzy  $H$  jako iloczynu  $A'$  i  $B'$ .
4. Rozkład  $H$  według wartości osobliwych.
5. Wyznaczenie macierzy obrotu i jej wyznacznika w celu sprawdzenia czy nie zaszedł przypadek odbicia.
6. Obrócenie centroidu  $a$  zgodnie z macierzą rotacji i obliczenie wektora przesunięcia jako różnicy centroidów.

Dane są dwa zbiory punktów:  $A$  i  $B$  w postaci macierzy o wymiarach  $3 \times n$ . Każdy wiersz w obu z nich opisuje pojedynczy punkt w postaci współrzędnych  $x, y, z$ . Równanie opisujące przejście punktu z jednego zbioru do drugiego wygląda następująco

$$A = RB + T + N$$

Gdzie  $R$  jest macierzą rotacji o wymiarach  $3 \times 3$ ,  $T$  - wektorem przesunięcia o wymiarach  $3 \times 1$  a  $N$  - wektorem szumu. Zakładamy też, że obrót odbywa się wokół początku układu. Chcemy dobrać takie  $R$  i  $T$ , aby zminimalizować wyrażenie

$$d^2(a, b) = \sum_{i=1}^n \|b_i - (Ra_i + T)\|^2$$

Niech

$$\mathbf{a} = \frac{1}{n} \sum_{i=1}^n \mathbf{a}_i \quad \mathbf{b} = \frac{1}{n} \sum_{i=1}^n \mathbf{b}_i$$

Wówczas  $\mathbf{a}'$  i  $\mathbf{b}'$  będą zbiorami odległości poszczególnych punktów w zbiorach  $\mathbf{A}$  i  $\mathbf{B}$  od ich centroidów

$$\mathbf{a}'_i = \mathbf{a}_i - \mathbf{a} \quad \mathbf{b}'_i = \mathbf{b}_i - \mathbf{b}$$

Mamy

$$d^2(\mathbf{a}', \mathbf{b}') = \sum_{i=1}^n \|\mathbf{b}_i - \mathbf{R}\mathbf{q}_i\|^2$$

Następnym krokiem jest wyznaczenie macierzy  $3 \times 3$

$$\mathbf{H} = \sum_{i=1}^n \mathbf{a}'_i \mathbf{b}'_i^\top$$

I rozłożenie jej na wartości osobliwe

$$\mathbf{H} = \mathbf{U} \mathbf{D} \mathbf{V}^\top$$

Ponieważ, zgodnie z tym co zostało wspomniane podczas interpretacji geometrycznej rozkładu SVD, macierz  $\mathbf{D}$  jest odpowiedzialna za skalowanie - podczas wyznaczenia wynikowej rotacji możemy ją odrzucić. W ten sposób otrzymujemy wyrażenie

$$\mathbf{R} = \mathbf{V} \mathbf{U}^\top$$

Jeśli wyznacznik  $\det \mathbf{R} = 1$  to  $\mathbf{R}$  stanowi macierz obrotu. W przeciwnym wypadku, gdy mamy do czynienia z wyznacznikiem  $\det \mathbf{R} = -1$  algorytm nie zwraca rozwiązań, ale ten przypadek z reguły nie występuje. Gdy obróćmy centroid zbioru  $\mathbf{A}$  zgodnie z wyznaczoną właśnie macierzą rotacji to wektor przesunięcia  $\mathbf{t}$  będzie stanowił różnicę między dwoma centroidami.

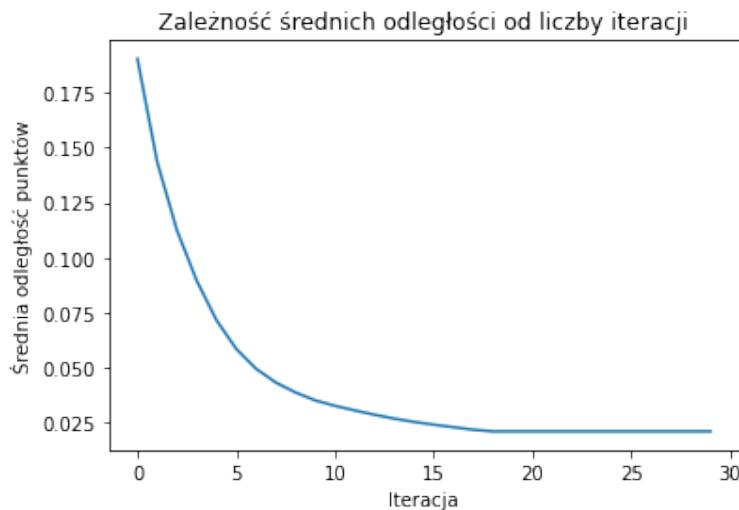
$$\mathbf{t} = \mathbf{b}^\top - \mathbf{R} \mathbf{a}^\top$$

Wynikową transformację możemy osiągnąć w odpowiedni sposób konkatenując

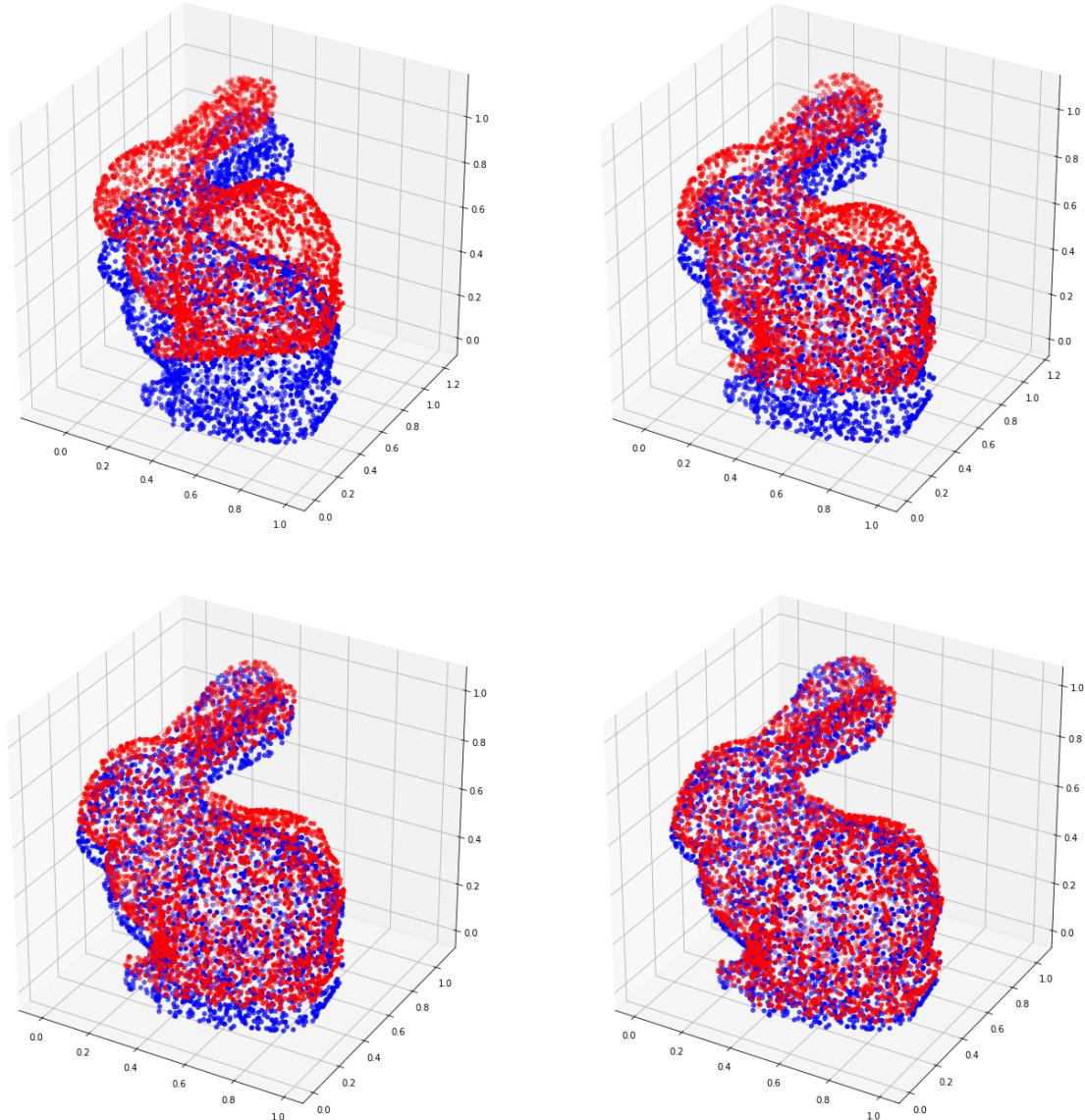
macierz obrotu z wektorem przesunięcia

$$T = \left[ \begin{array}{ccc|c} & & & \\ R & & & t^\top \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

Obliczona transformacja nie stanowi jeszcze ostatecznego rozwiązania algorytmu. Możemy jedynie stwierdzić, że poddany tej transformacji obiekt znajduje się bliżej do celowego szablonu niż był do tej pory. Dążąc do ostatecznego wyniku, potrzebujemy ponownie wykonywać poprzednie kroki do momentu, aż odległości między korespondującymi punktami w zbiorach staną się akceptowalnie małe. Każdy taki etap nosi nazwę **iteracji**, dlatego też algorytm nazywany jest iteracyjnym.



Rysunek 13: Wykres średnich odległości korespondujących punktów między dwiema chmurami przeprowadzony dla rotacji w zakresie  $0.25\pi$ . Zależność ta została powtórzona przez największą liczbę prób. Z przebiegu widać, że algorytm w największej liczbie przypadków znajdował ostateczną transformację przy osiemnastej iteracji.



Rysunek 14: Działanie algorytmu ICP na przykładzie obiektu w kształcie królika.

#### 4.1. Implementacja algorytmu

Na następnych stronach zamieszczona została minimalistyczna implementacja omówionej listy kroków w języku Python. Pełna wersja algorytmu ICP wraz z komentarzami znajduje się w sekcji *Załączniki*.

```
1 def transform_matrix(A, B):
2     n = A.shape[1]
3     centroid_A = np.mean(A, axis=0)
4     centroid_B = np.mean(B, axis=0)
5
6     A1 = A - centroid_A
7     B1 = B - centroid_B
8
9     H = np.dot(A1.T, B1)
10    U, D, Vt = np.linalg.svd(H)
11    R = np.dot(Vt.T, U.T)
12
13    if np.linalg.det(R) < 0:
14        Vt[n - 1, :] *= -1
15    R = np.dot(Vt.T, U.T)
16
17    t = centroid_B.T - np.dot(R, centroid_A.T)
18    return R, t
```

Funkcja *transform matrix* przyjmuje dwa argumenty w postaci macierzy **A** i **B**.  
i. **A** jest macierzą obiektu, który chcemy dopasować do szablonu **B**. W rozważanym przypadku obie macierze są wymiaru  $3 \times n$ , czyli składają się z trzech kolumn, z których pierwsza zawiera wartości współrzędnej x, druga - y, a trzecia - z. Każdy wiersz w obu macierzach reprezentuje jeden z  $n$  punktów. W linijkach trzeciej i czwartej następuje wyznaczenie centroidów **A** i **B** w postaci wektorów o długości trzy przez obliczenie średniej dla każdej kolumny współrzędnych. Macierze **A1** i **B1** zostają utworzone przez odjęcie od **A** i **B** ich centroidów. Pomaga nam to ustandaryzować wartości zmiennych. W linijce dziewiątej liczony jest iloczyn nowo utworzonych macierzy (transponujemy **A1**, żeby miała tyle samo kolumn co **B1** wierszy). Następnie następuje rozkład **H** według wartości osobliwych, który zwraca trzy czynniki w postaci macierzy **U**, **D** i **Vt**. Mnożąc przez siebie **U** i **Vt** (obie transponowane) otrzymujemy otrzymujemy macierz rotacji **R**. Warunek poniżej obsługuje specjalny przypadek odbicia. Wektor przesunięcia **t** jest liczony jako różnica centroidu **B** i obróconego przez **R** centroidu **A**. Funkcja zwraca

przekształcenie jako  $\mathbf{R}$  i  $\mathbf{t}$ .

Gdy mamy zaimplementowane wyznaczanie transformacji dla pojedynczej iteracji możemy przejść do ostatecznego algorytmu ICP.

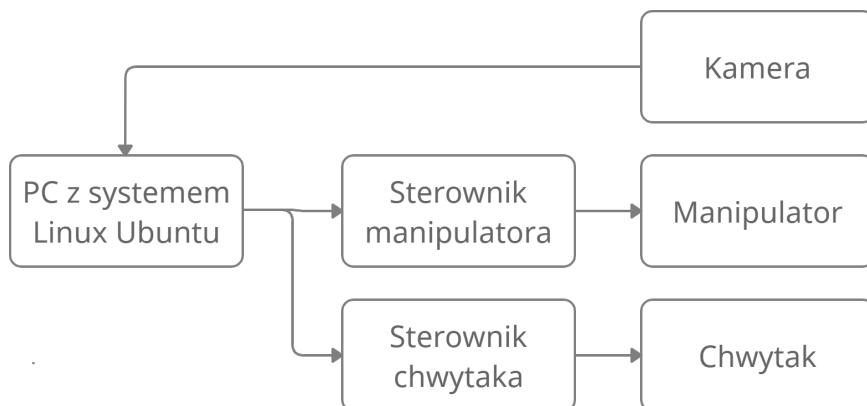
```
1 def icp(A, B, max_iterations, tolerance=0.001):
2
3     n = A.shape[1]
4     A1 = np.ones((n + 1, A.shape[0]))
5     B1 = np.ones((n + 1, B.shape[0]))
6     A1[:n, :] = np.copy(A.T)
7     B1[:n, :] = np.copy(B.T)
8
9     prev_error = 0
10    for i in range(iterations):
11        distances, indices = calculate_distance(A1[:n, :].T,
12                                                B1[:n, :].T)
13        T = transform_matrix(A1[:n, :].T, B1[:n, indices].T)
14        A1 = np.dot(T, A1)
15
16        mean_error = np.mean(distances)
17        if abs(prev_error - mean_error) < tolerance:
18            break
19        prev_error = mean_error
20
21    T = transform_matrix(A, A1[:n, :].T)
22    return T
```

Funkcja *icp* przyjmuje cztery argumenty w tym jeden domyślny. Przyjmowane są zmienne **A** i **B** jako macierze o wymiarach  $3 \times n$ , maksymalna liczba iteracji oraz domyślnie ustwiony próg tolerancji. W linijkach 4-7 tworzone są macierze **A1** i **B1** jako czterokolumnowe odpowiedniki **A** i **B**. Dzieje się to przez skopiowanie wartości **A** i **B** i dostawienie kolumny jedynek. Linijki 10-19 zawierają główną pętlę programu wykonującą się maksymalnie *max iterations* razy. Przed wejściem do pętli została zainicjalizowana zmienna *prev error*, która będzie nadpisywana przy każdej iteracji. Następnie funkcja *calculate distance* zwraca odległości euklidesowe między poszczególnymi punktami i

ich indeksy. Będą one potem nam potrzebne do wyznaczenia średniego błędu dopasowania. Wyznaczana jest najlepsza transformacja  $T$  między **A1** i **B1**, po czym zostaje ona zastosowana do przesunięcia **A1**. Pod koniec działania pętli liczony jest średni błąd jako średnia odległości między punktami. Jeśli wartość bezwzględna z różnicą średniego i poprzedniego błędu jest większa niż przyjęta toleracja, a dotychczasowa liczba iteracji mniejsza niż *max iterations* - pętla wykonuje się ponownie. W momencie znalezienia wystarczająco dokładnej transformacji przed zakończeniem pętli - jest ona przerywana.

## 4.2. Działanie robota w praktyce

Zadanie zostało zrealizowane przez manipulator UR5, którego sterownik był podłączony do jednostki centralnej komputera z systemem Linux Ubuntu. Efektor końcowy robota zakończony był chwytkiem dwupalczastym o równoległych szczękach. Jego sterownik również łączył się z jednostką centralną. Rolę systemu wizyjnego pełniła kamera *Intel RealSense* na bieżąco wysyłająca dane do komputera przez USB.



Rysunek 15: Diagram całego systemu.

Przejście od teorii do praktyki wymagało dodatkowo implementacji funkcji odpowiedzialnych za poruszanie manipulatorem. Został w tym celu użyty popularny middleware ROS (*Robot Operating System*), do którego odwołania były kierowanie z kodu Pythona. Omawiając program, możemy zwrócić uwagę, że wykorzystane zostały dwa serwisy: jeden odpowiedzialny za chwytek i drugi - za przeguby manipulatora. Funkcja *move gripper* sterująca chwytkiem przyjmuje argument logiczny otwierający go w przypadku prawdy i zamykający w przypadku przeciwnym.

```
1 def move_gripper(opened):
2     service = '/kair_ev55_2/open'
3     rospy.wait_for_service(service)
4     try:
5         moveGripper = rospy.ServiceProxy(service, Open)
6         return moveGripper(50 if opened else 0)
7     except rospy.ServiceException as e:
8         print('Service call failed: %s'%e)
```

Metoda *move* kontrolująca ustawienie manipulatora jako argument przyjmuje z kolejną tablicę sześciu wartości opisujących dane miejsce w przestrzeni przez jego współrzędne kartezjańskie i kąty Eulera.

```
1 def move(array):
2     service = '/ur5_1/move_ptp_p'
3
4     if len(array) == 6:
5         rospy.wait_for_service(service)
6         try:
7             movePtp = rospy.ServiceProxy(service, MovePTP_P)
8             req = MovePTP_PRequest()
9             req.target.data = array
10            req.a = 1
11            req.v = 1
12            req.t = 5
13            req.r = 0
14            resp = movePtp(req)
15            return resp
16        except rospy.ServiceException as e:
17            print('Service call failed: %s'%e)
18    else:
19        print('Array length must be 6')
```

Również z ROSa korzystając zostało obsłużone zachowanie kamery. W tej roli została użyta niewielkich rozmiarów kamera *Intel RealSense* zdolna zapewnić obraz w

formacie RGBD. W linijce 26. przytoczonego kodu źródłowego znajduje się funkcja odpowiedzialna za zrobienie zdjęcia. Zapisuje ona bieżącą klatkę do pliku, który następnie może zostać załadowany do programu korzystającego z ICP w celu wyznaczenia ostatecznej transformacji.

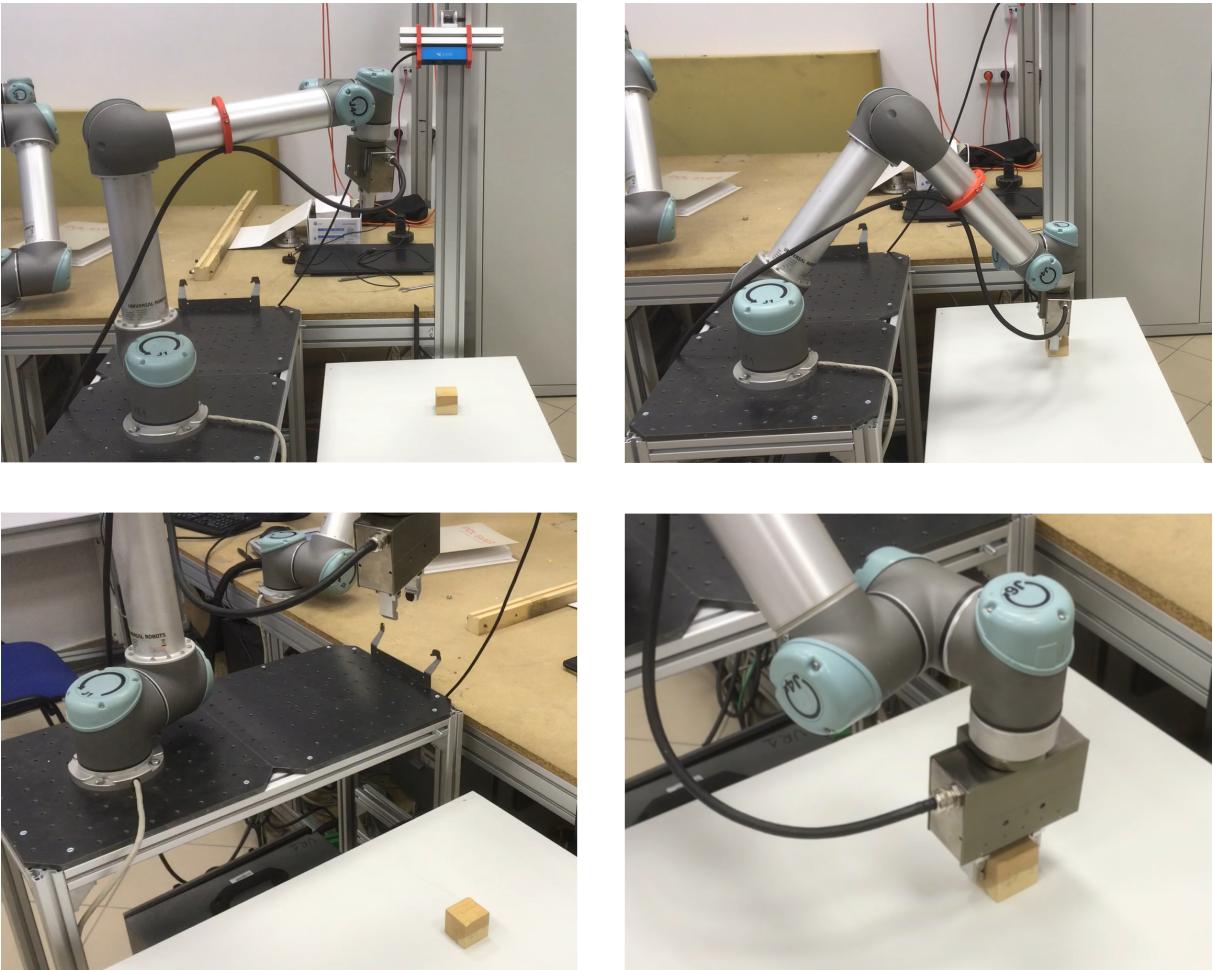
```
1 def pcl_callback(msg):
2     i = 0
3     x_values = []
4     y_values = []
5     z_values = []
6
7     while i < len(msg.data):
8         point = msg.data[i * msg.point_step:(i+1) * msg.point_step]
9
10    if len(point) >= 20:
11        x = struct.unpack('f', point[0:4])[0]
12        y = struct.unpack('f', point[4:8])[0]
13        z = struct.unpack('f', point[8:12])[0]
14
15        x_values.append(x)
16        y_values.append(y)
17        z_values.append(z)
18
19        i += msg.point_step
20
21    A = np.zeros((len(x_values), 3))
22    A[:, 0] = x_values[:]
23    A[:, 1] = y_values[:]
24    A[:, 2] = z_values[:]
25
26    with open('brick.npy', 'wb') as file:
27        np.save(file, A)
28
29 rospy.init_node('rs_pcl')
30 rospy.Subscriber('/camera/depth/color/points',
31                  PointCloud2, pcl_callback)
```

32

```
33 while not rospy.is_shutdown():
34     pass
```

Mimo wszelkich starań działanie proponowanego systemu w praktyce często różniło się od pożądanego. Głównym problemem była rozbieżność między chmurą punktów widzianą z kamery a rzeczywistym kształtem przedmiotu. Gdy ta rozbieżność była zbyt duża - algorytm nie był w stanie odnaleźć poprawnego przekształcenia. Przez to w niektórych przypadkach konieczne okazało się znajdowanie pozycji obiektu zastępując niekształtną chmurę punktów jej środkiem ciężkości. A ponieważ obiekty traktowane są jako wykonane z jednorodnego materiału, środek ten możemy nazywać centroidem. Jest to metoda działająca w przypadku odpowiednio małych obiektów, które trudno wykryć za pomocą kamery, natomiast nie dostarcza tak dokładnych informacji jak proponowane rozwiązanie.

Podejście oparte na znalezieniu centroidu stanowi pewien kompromis między potrzebą estymacji położenia a niską jakością danych z kamery. Przede wszystkim, centroid opisywany jest przez trzy współrzędne, a więc tracona jest informacja o orientacji obiektu w przestrzeni. Dlatego w przeprowadzonym doświadczeniu orientacja kostki została wcześniej skorygowana i przyjęta w postaci kątów Eulera jako  $(0, 90^\circ, 0)$ , czyli wszystkie wartości zostały zaokrąglone do kątów prostych. Następnie należało znaleźć właściwą transformację między bazą robota a centroidem jako odległości wzduż każdej osi układu kartezjańskiego. Wówczas ostatnim krokiem doświadczenia było uruchomienie manipulatora i wysłaniu mu polecenia dojechania do ustalonego punktu, zaciśnięcia chwytaka i powrotu do pozycji początkowej.



Rysunek 16: Doświadczenie wykonane przez ramię robota UR5. Drewniany klocek niewielkich rozmiarów z trudem został wyodrębniony na tle chmury punktów pobranej z kamery, natomiast ze względu na maksymalny rozstaw szczęk chytaka niemożliwe było chwytanie większych elementów.

## 5. Podsumowanie i wnioski

W tej pracy została zaproponowana metoda estymacji położenia obiektu, którego modelem dysponujemy. Rozważony problem jest kluczowy na polu współczesnej robotyki, głównie przez trend do stosowania coraz bardziej "elastycznych" sposobów sterowania manipulatorami. Problem został rozbity na podstawowe zagadnienia, których rozwiązania również zostały przeprowadzone. Ponieważ jest nie jest to temat nowy, podczas pisania pracy było wykorzystanych wiele obecnie istniejących źródeł opisujących pełne lub cząstkowe rozwiązanie problemu. Tak czy inaczej, główną motywacją było znalezienie możliwie prostego i skutecznego rozwiązania, co zostało osiągnięte.

Mając na uwadze dotychczasowe rozwiązania opisane w rozdziale drugim, pomysł na opracowanie kolejnej metody interakcji maszyn z otoczeniem może się wydawać co najmniej błahy, a w najgorszym wypadku zbędny. Takie stwierdzenia jednak tracą na sile po bliższym przyjrzeniu się motywacjom jakie przyświecały powstaniu tej pracy. Gdybyśmy chcieli wykorzystać sieci neuronowe lub uczenie przez wzmacnianie opisane w rozdziale 2. wiążałoby się to z dodatkowym czasem jaki trzeba by było poświęcić na przetrenowanie tych modeli. Metody widzenia maszynowego z kolei nie są najczęściej przystosowane do radzenia sobie z analizą przestrzeni trójwymiarowych. Poza tym, te algorytmy wydają się nadmiernie złożone, co często utrudnia pracę z nimi. Trudno jest natomiast znaleźć wymierny wskaźnik poziomu skomplikowania danego algorytmu. Moglibyśmy posłużyć się miarą złożoności Kołmogorowa, która określa jak długi musi być najkrótszy możliwy kod, aby wygenerować dany obiekt, w tym wypadku macierz transformacji. Niestety, ponieważ w składni programów znajduje się wiele odwołań i wcześniej zaimplementowanych funkcjonalności - złożoność Kołmogorowa jest w stanie jedynie zgrubnie oszacować poziom komplikacji, dlatego nie została przeze mnie zastosowana.

Z drugiej strony, istnieje wiele podobieństw między proponowaną pracą a przedstawionymi metodami uczenia maszynowego. Oba podejścia rozwiązują problem iteracyjnie próbując z każdym krokiem zmniejszyć wartość średniego błędu. W przypadku sieci neuronowych mówimy o funkcji straty, *Reinforcement Learning* posługuje się terminologią nagrody i kary, w proponowanym pomyśle opartym o SVD mamy miarę dopasowania.

W pożądanych przypadkach, miary te najczęściej maleją wraz z czasem o coraz mniejsze wartości aż funkcja stanie się niemalejąca. Można wtedy powiedzieć, że dany algorytm poprawnie wykonał swoje zadanie.

Podsumowując, praca, mimo wszelkich zalet nie stanowi samowystarczalnego systemu, który mógłby w obecnym stanie zostać wykorzystany na potrzeby przemysłu. Przede wszystkim dlatego, że obejmuje tylko część zagadnień jakie należałyby rozwiązać przed procesem komercjalizacji. Czynności takie jak nadzór działania czy zorientowanie chwytaka względem szablonu obiektu nadal wymagają interwencji człowieka. Projekt ma za to potencjał do stania się częścią większego oprogramowania, gdzie stanowiłby znaczny wkład na polu manipulacji i przemysłu 4.0.

## 5.1. Dalszy rozwój

Mimo, że zaproponowana metoda estymacji położenia obiektu wykonała założone zadanie z powodzeniem, zastosowana technika posiada również pewne ograniczenia. Do najważniejszych z nich należy brak możliwości wyłonienia pożdanego przedmiotu ze zbioru zawierającego więcej niż jeden element. Tą kwestię można oczywiście rozwiązać stosując metody segmentacji. Mielibyśmy wtedy chmurę punktów rozbitą na kilka elementów, ale rodziłoby to kolejny problem - algorytm musiałby rozpoznać, który z nich stanowi żądany obiekt. W tym miejscu mogłaby zostać zaimplementowana logika odpowiadająca za obsługę przypadków w zależności czy w danych, które są sprawdzane mamy do czynienia z jednym, kilkoma lub żadnym szukanym obiektem. Podsumowując, dalszy rozwój tej pracy mógłby obejmować:

- **Odszumienie** wejściowej chmury punktów przez odcięcie powierzchni stołu i pozbicie się nieistotnych elementów podczas przetwarzania danych.
- **Segmentację** w celu sprawdzenia z iloma elementami mamy do czynienia i gdzie się one znajdują.
- **Rozpoznanie** jak bardzo dany przedmiot odbiega od swojego szablonu. To pozwoliłoby stwierdzić czy mamy do czynienia z właściwym obiektem. Jeśli tak nie jest - dalsza estymacja położenia nie ma sensu.

Odszumienie przychodzących danych w przypadku chmur punktów nie jest łatwym zajęciem. Może zostać przeprowadzone ręcznie przez wyśrodkowanie obiektu i "przyjęcie" chmury wzdłuż jego granic. W przypadku gdy oś  $z$  jest skierowana do góry i prostopadła do stołu, punkty do niego należące mogą zostać odcięte przez usunięcie wszystkich punktów, dla których  $z$  jest większe od wysokości stołu. Wymaga to jednak pewnej interwencji człowieka. Chcąc ten proces zautomatyzować moglibyśmy posłużyć się metodą RANSAC (*Random sample consensus*) - algorytmem iteracyjnym do usuwania z danych elementów odstających. Problem ten można uprościć również zanajając pozycję kamery względem stołu, dzięki czemu stosunek liczby punktów obiektu do wszystkich punktów byłby większy.

Wracając do tematu segmentacji, zadanie to dla niewielkiej liczby elementów może zostać zrealizowane przy pomocy technik analizy skupień, do którego zaliczamy algorytmy uczenia nienadzorowanego takie jak DBSCAN, grupowanie hierarchiczne i metodę  $k$ -średnich. Jeśli podział zostałby zrealizowany dla każdej liczby klastrów od 1 do  $n$  (przy czym  $n$  to założona maksymalna liczba klastrów) - wówczas do dyspozycji mielibyśmy miarę dopasowania do każdej z tych ewentualności. Taką miarą jest najczęściej kryterium sumy kwadratów w obrębie klastra WCSS (*Within-Cluster Sum of Squared*). Z niej możliwe stałoby się odczytanie optymalnej liczby elementów.

Niestety, taka metoda jest zachłanna obliczeniowo, przez co zużywa sporo czasu i może być stosowana tylko do małych  $n$ . Wówczas, liczba klastrów rozważanych w każdej iteracji rośnie o jeden zgodnie z ciągiem arytmetycznym: 1, 2, 3, ...  $n$ . Ponieważ, w każdej iteracji jest mowa o innych klastrach ich sumaryczną ilość przedstawia ciąg: 1, 3, 6, 10, 15, ...  $\sum n$ . Innym powodem, który utrudnia pracę w wieloma elementami jest przede wszystkim to, że im większy jest ten zbiór - tym więcej charakterystycznych cech obiektów może być przysłonięte przez inne elementy. Trudno jest się z nim uporać w sposób obliczeniowy, ale może zostać w najprostszy sposób rozwiązany przez umieszczenie elementów na podajniku wibracyjnym.

Z drugiej strony do rozwiązania problemu wielu obiektów mogłyby zostać zaprężgnięte sieci neuronowe, a szczególną uwagę w tym względzie przykuwają modele tj. *Mask R-CNN* i *PointNet*. Pierwsza z nich jest przykładem sieci R-CNN (ang. *Region Based*

*Convolutional Neural Networks)* przygotowanym specjalnie pod problem segmentacji. Natomiast struktura sieci *PointNet* została przystosowana do radzenia sobie z danymi w postaci chmur punktów i była testowana w zadaniach klasyfikacji i segmentacji. Niestety, jak większość sieci neuronowych, obie są przykładem uczenia nadzorowanego, a więc ich użycie wymaga etykietowania danych, na których będą trenowane. Jeśli zbiór uczący zostanie utworzony w postaci obrazów RGBD, a każdy kształt przedmiotu oznaczymy innym kolorem, wówczas takie etykietowanie może być przeprowadzone korzystając z nałożenia na siebie informacji o kolorze (RGB) i głębokości (D). Mówiąc prościej, mamy "zlepek" różnych kształtów, który chcemy podzielić na klastry. Kompletna informacja o nich, tj. ich liczba i położenie jest już zawarta w obrazie, więc tym co możemy zrobić jest przenieść te dane przypisując do każdego widzianego punktu w przestrzeni jego klaster reprezentowany przez kolor przedmiotu, do którego należy. To oczywiście niesie za sobą komplikacje w postaci różnego oświetlenia i cieni, ale na ogólnie nie stanowią one większego problemu.

## Bibliografia

1. John. J. Craig - *Introduction to Robotics: Mechanics and Control*; wyd. polskie: *Wprowadzenie do robotyki: mechanika i sterowanie*, tłum. Józef Knapczyk, Wydawnictwa Naukowo-Techniczne, 1993
2. Marc H. Raibert - *Legged Robots That Balance*, MIT press, 1986
3. James J. Gibson - *The Senses Considered as Perceptual Systems*, Tom 2. Nr 1. Boston: Houghton Mifflin, 1966
4. Ryszard Tadeusiewicz - *Biocybernetyka*
5. R. Peeters et al. - *The Representation of Tool Use in Humans and Monkeys: Common and Uniquely Human Features*, "Journal of Neuroscience", nr 29 (wrzesień 2009), s. 11523-11539
6. Scott H. Johnson-Frey *The Neural Bases of Complex Tool Use in Humans*, "TRENDS in Cognitive Sciences", nr 8 (luty 2004), s. 71-78
7. Leonard Mlodinow, *The Upright Thinkers*; wyd. polskie: *Krótką historią rozumu. Od pierwszej myśli do rozumienia wszechświata*, tłum. Sebastian Szymański, Prószyński i S-ka, Warszawa 2016
8. David Forsyth, Jean Ponce - *Computer vision: A modern approach*, Prentice hall, 2011
9. Jan Erik Solem - *Programming Computer Vision with Python*, O'Reilly Media, czerwiec 2012
10. Ryszard Tadeusiewicz et al. - *Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#*, 2007
11. Ian Goodfellow, Yoshua Bengio, Aaron Courville - *Deep Learning*; wyd. polskie: *Deep Learning. Systemy uczące się*, tłum. Witold Sikorski, Wydawnictwo Naukowe PWN, Warszawa 2018

12. Hao-Shu Fang et al. - *Graspnet-1billion: A large-scale benchmark for general object grasping*, "Proceedings of the IEEE/CVF conference on computer vision and pattern recognition", 2020
13. Mousavian, Arsalan, Clemens Eppner, Dieter Fox - *6-dof graspnet: Variational grasp generation for object manipulation*, "Proceedings of the IEEE/CVF International Conference on Computer Vision", 2019
14. Stuart Russell, Peter Norvig - *Artificial intelligence: a modern approach*, Upper Saddle River, New Jersey 2010, s. 830-831
15. Leslie PackKaelbling, Michael L. Littman, Andrew W. Moore - *Reinforcement learning: A survey* - "Journal of artificial intelligence research", nr 4 (1996), s. 237-285
16. Ilge Akkaya et al. - *Solving Rubik's Cube with a Robot Hand*, arXiv: 1910.07113 (2019)
17. Steven J. Leon, Ion Bica, Tiina Hohn - *Linear Algebra with Applications*, Upper Saddle River, NJ: Pearson Prentice Hall, 2006
18. Sudipto Banerjee, Anindya Roy - *Linear algebra and matrix analysis for statistics*, Tom 181, Boca Raton, Floryda, USA, Crc Press, 2014
19. Dan Kalman - *A singularly valuable decomposition: the SVD of a matrix*, "The college mathematics journal" nr 27.1 (1996), s. 2-23
20. Paul J. Besl, Neil D. McKay. *Method for registration of 3-D shapes*, "Sensor fusion IV: control paradigms and data structures", Tom 1611. Spie, 1992
21. K.S. Arun, T.S. Huang, S.D. Blostein - *Least-Squares Fitting of Two 3-D Point Sets*, "IEEE Transactions on pattern analysis and machine intelligence", nr 5 (1987), s. 698-700
22. Yang Chen, Gérard Medioni - *Object modelling by registration of multiple range images* - "Image and vision computing", nr 10.3 (1992), s. 145-155
23. Zhengyou Zhang - *Iterative point matching for registration of free-form curves and surfaces* - "International journal of computer vision", nr 13.2 (1994), s. 119-152

24. Fischler, Martin A., Robert C. Bolles - *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography*, "Communications of the ACM", nr 24.6 (1981), s. 381-395
25. Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick - *Mask R-CNN*, "Proceedings of the IEEE international conference on computer vision", 2017, s. 2961-2969
26. Charles R. Qi, Hao Su, Kaichun Mo, Leonidas J. Guibas - *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*, "Proceedings of the IEEE conference on computer vision and pattern recognition", 2017, s. 652-660

## Załączniki

### Kod programu

```
1 import numpy as np
2 import trimesh
3 from sklearn.preprocessing import MinMaxScaler
4 from sklearn.neighbors import NearestNeighbors
5 from pyquaternion import Quaternion
6 import math
7 import random
8
9
10 def rotation_matrix(axis, angle):
11     """
12     Macierz rotacji dla obrotu danego przez oś i kąt
13     """
14     [a, b, c, d] = Quaternion(axis=axis, angle=angle)
15
16     R = np.array([[a*a+b*b-c*c-d*d,
17                   2 * (b*c-a*d),
18                   2 * (b*d+a*c)],
19                   [2 * (b*c+a*d),
20                   a*a+c*c-b*b-d*d,
21                   2 * (c*d-a*b)],
22                   [2 * (b*d-a*c),
23                   2 * (c*d+a*b),
24                   a*a+d*d-b*b-c*c]])
25
26
27
28 def transform_matrix(A, B):
29     """
30     Funkcja znajdująca najlepszą transformację między
31     chmurami punktów A i B metodą najmniejszych kwadratów
32     Wejście:
33         A - macierz m x n
```

```
34     B - macierz m x n
35     Wyjście:
36     T - macierz transformacji (n+1) x (n+1)
37     ...
38     n = A.shape[1]
39
40     # Wyznaczenie centroidów
41     centroid_A = np.mean(A, axis=0)
42     centroid_B = np.mean(B, axis=0)
43     # Utworzenie nowych macierzy A1 i B1 przez
44     # przedstawienie każdego punktu A i B jako
45     # jego odległości od centroidu
46     A1 = A - centroid_A
47     B1 = B - centroid_B
48
49     H = np.dot(A1.T, B1)
50     # Rozkład H według wartości osobliwych
51     U, D, Vt = np.linalg.svd(H)
52     # Wyznaczenie możliwej macierzy obrotu
53     R = np.dot(Vt.T, U.T)
54
55     # Obsługa szczególnego przypadku odbicia
56     if np.linalg.det(R) < 0:
57         Vt[n - 1, :] *= -1
58     R = np.dot(Vt.T, U.T)
59
60     # Wektor przesunięcia jako różnica centroidów
61     t = centroid_B.T - np.dot(R, centroid_A.T)
62
63     # Konkatenacja macierzy rotacji i wektora
64     # przesunięcia w celu uzyskania ostatecznej
65     # transformacji
66     T = np.identity(n + 1)
67     T[:n, :n] = R
68     T[:n, n] = t
69
```

```
70     return T
71
72
73 def calculate_distance(A, B):
74     """
75     Funkcja znajdująca najbliższego sąsiada wśród B
76     dla każdego punktu należącego do A w metryce
77     euklidesowej
78     Wejście:
79         A - macierz m x n
80         B - macierz m x n
81     Wyjście:
82         distances - odległości od najbliższego sąsiada
83         indices - indeksy najbliższego sąsiada
84     """
85     nn = NearestNeighbors(n_neighbors=1)
86     nn.fit(B)
87     distances, indices = nn.kneighbors(A, return_distance=True)
88     return distances.ravel(), indices.ravel()
89
90
91 def icp(A, B, max_iterations, tolerance=0.001):
92     """
93     Algorytm Iterative Closest Point iteracyjnie
94     znajdujący najlepszą transformację między chmurami
95     punktów A i B
96     Wejście:
97         A - macierz m x n
98         B - macierz m x n
99     Wyjście:
100        T - macierz transformacji
101    """
102    n = A.shape[1]
103
104    A1 = np.ones((n + 1, A.shape[0]))
105    B1 = np.ones((n + 1, B.shape[0]))
```

```
106     A1[:n, :] = np.copy(A.T)
107     B1[:n, :] = np.copy(B.T)
108
109     prev_error = 0
110
111     for i in range(iterations):
112         # Znalezienie najbliższego sąsiada między A1 a B1
113         distances, indices = calculate_distance(A1[:n, :].T, B1[:n, :].T)
114
115         # Obliczenie transformacji między A1 a najbliższymi punktami B1
116         T = transform_matrix(A1[:n, :].T, B1[:n, indices].T)
117
118         # Przesunięcie A1 o wyznaczoną transformację
119         A1 = np.dot(T, A1)
120
121         # Sprawdzenie błędu
122         mean_error = np.mean(distances)
123
124         if abs(prev_error - mean_error) < tolerance:
125             break
126
127         prev_error = mean_error
128
129
130
131     # Wyznaczenie ostatecznej transformacji
132     T = transform_matrix(A, A1[:n, :].T)
133
134     return T, mean_error
135
136
137
138
139
140
141
```

```
142 A = scaler.fit_transform(A)
143
144 # Próbkowanie i standaryzacja zbioru B
145 # Mimo, że zostały użyte te same funkcje co powyżej
146 # zbiory A i B są różne ze względu na losowy
147 # charakter próbkowania
148 B = mesh.sample(density)
149 B = scaler.fit_transform(B)
150
151 # Przesunięcie B o losowy wektor t
152 t = np.random.rand(dim) * translation
153 B += t
154
155 axis = np.zeros(dim)
156 axis[dim - 1] = 1
157 angle = random.random() * rotation
158 # Obrócenie B o losową macierz R
159 R = rotation_matrix(axis, angle)
160 B = np.dot(R, B.T).T
161
162 iterations = 15 # maksymalna liczba iteracji
163
164 errors = []
165 for i in range(iterations):
166     T, error = icp(B, A, iterations + 1)
167     errors.append(error)
168
169 print(T)
170 print(errors)
```