2021 年度（令和 3 年度）

創造工学セミナーII　Final Report

人物認識によりジェスチャーで操作する移動ロボットの開発

研究メンバー

S5-18053 堀越優来

S5-18058 丸山拓己

S5-18060 宮崎生望

S5-18064 目黒新大

S5-18069 吉武敏

指導教員

金丸隆志　教授

所属研究室

知能機械研究室

目次

1　緒言

1.1　研究背景　*宮崎生望*

近年の新型コロナウイルス感染拡大は、人々の生活に様々な影響を与えた。特に、図 1-1 で示すように感染防止を目的とした「人やモノとの接触に関する意識」には大きな変化があった。コロナ禍では非接触で操作する機器の重要が高まり、キャッシュレス決済や非接触タッチパネルの導入などで実店舗や施設は感染対策を行っている。しかし、そのような需要が高まる中で、まだ非接触な操作を実現できていない場所も多数存在しており、その中でも店員との接触や服への接触の問題がある試着に着目した。また、試着について図 1-2 で示すように実店舗での試着を面倒に感じる人や、店員がいると試着室に入りづらいというように、実店舗で試着することに抵抗を感じる人が多いという問題もある。これらの問題を解決するために私たちは移動ロボットの開発を行う。

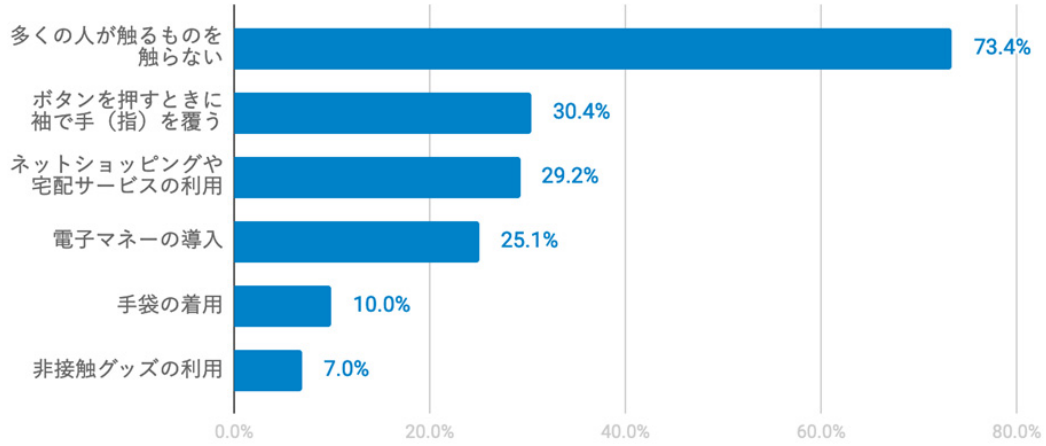


図 1-1　コロナ渦における接触を避けるために意識した、変えた行動[1-1]

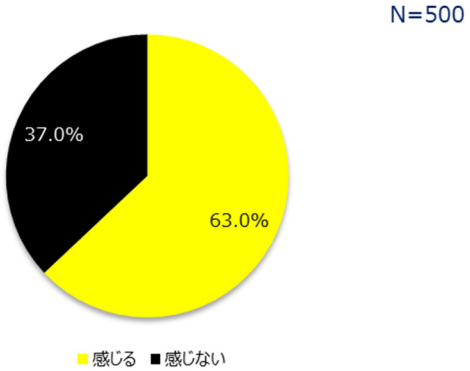


図 1-2　実店舗で試着することに抵抗を感じる割合[1-2]

1.2　先行研究　*目黒新大*

1.2.1　デプスカメラを用いた撮影補助装置の開発

本研究で制作する移動ロボットに関連し、先行研究である撮影補助装置を図　1 -3 に示す。先行研究の目的はデプスカメラである RealSense を用い追従機能を実現した撮影補助装置の開発である。移動ロボットとして Roomba を採用し、RealSense で肌色検出と顔検出と形検出を行うことで人物の認識を実現し、人物追従を可能にしている。しかし RealSense では正面からしか人物を検出することができないという問題があった。そこで我々は、Azure Kinect を用いて、様々な角度からの人物認識を実現する。また Roomba ではなく TurtleBot3 Waffle Pi を採用し、ロボット開発ミドルウェアである ROS を使用することで、制御を行う。

図　1 -3　先行研究の撮影補助装置[1-3]

1.3　研究概要　*宮崎生望*

1.3.1　研究初期の開発予定ロボットについて

研究初期に開発予定だった移動型仮想試着装置を図　1 -4 に示す。まず、人物の周りを移動ロボットが旋回する。そして、移動ロボットに搭載されている Azure Kinect で骨格情報を読み取り、その情報を基に仮想空間で試着している映像をリアルタイムで映し出す構成であった。しかし、仮想試着システムの開発で 3D モデルを作成する技術がないことや、開発が困難であったことから、ジェスチャーで操作する移動ロボットの開発に注力することにした。

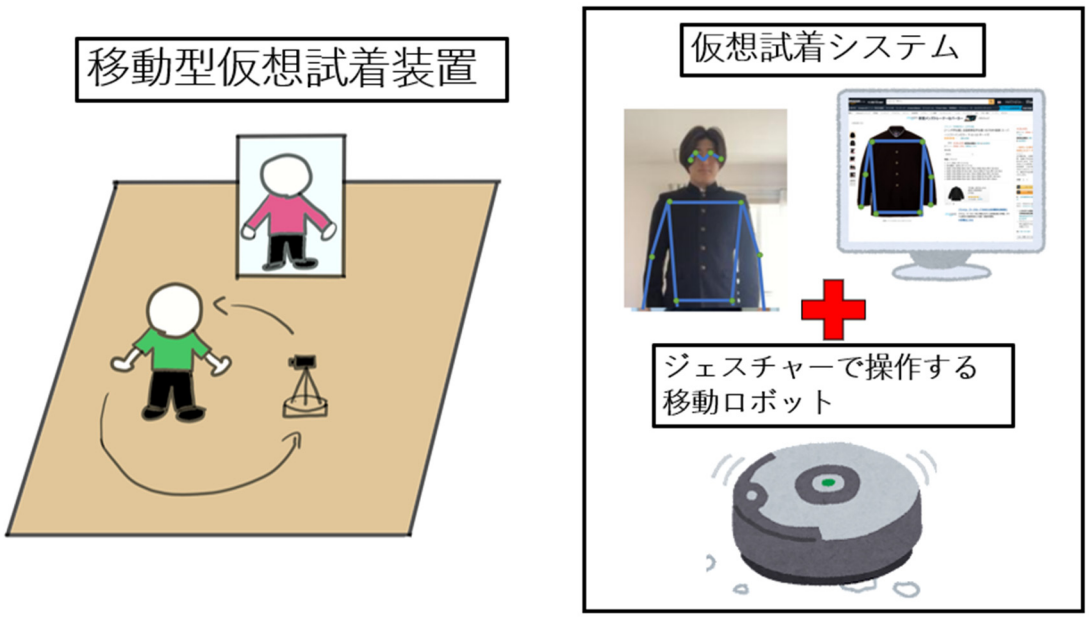


図　1 -4　移動型仮想試着装置の初期案

1.3.2　製作物の決定

我々は人物認識によりジェスチャーで操作する移動ロボットを開発する。人物認識に必要なセンサとして Azure Kinect を使用する。Azure Kinect は周囲 360 度のどこからでも骨格を読み取ることができるセンサであり、本研究では以下の 2 つの役割で使用していく。

1. 人物の骨格から人物のジェスチャーを認識してロボットを操作する役割
2. 人物までの距離を一定に保つ距離センサとしての役割

以上より、人物の周りを旋回する機能があるロボットをジェスチャーで操作することが可能となる。なお、ロボットの使用環境は室内を想定する。製作した装置を図 1-5 に示す。

図 1-5　ジェスチャーで操作する移動ロボット

1.3.3　研究の方向性

研究を進めるにあたり、我々は以下のような目標を項目ごとに設定した。

移動ロボットの設計

1. 人物の全身を映せる位置に Azure Kinect を取り付けること。
2. ロボットの移動の時の振動により映像がぶれないようにすること。
3. TurtleBot3 Waffle Pi の最大積載量を超えないようにすること。

人物認識

1. Azure Kinect により取得した三次元座標を使用し、360 度どの位置からでもジェスチャーを認識できること。
2. ロボットから人物までの距離の計測を実現すること。
3. その結果を用いて移動ロボットを実現すること。

移動ロボットの制御

1. P 制御を用いて人物の全身を常に映す制御を行うこと。
2. P 制御を用いて人物を常に画面の中心に映す制御を行うこと。
3. 右旋回や左旋回の動作を行うためのプログラムを作成すること。

UI（ユーザーインターフェース）の作成

1. 人物がロボットを操作するうえでジェスチャーの状態やロボットの位置などを分かりやすく表示するアプリケーションを作成すること。

以上の目標をそれぞれ達成することで、ジェスチャーで操作する移動ロボットを実現させる（図　1 -6）。

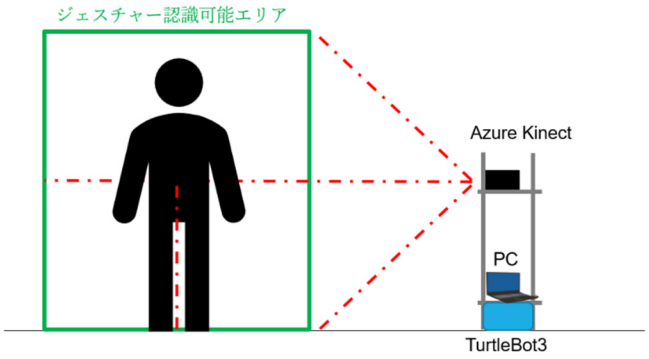


図　1 -6　各機器の設置位置

2　移動型人物認識ロボット　*宮崎生望*

2.1　移動ロボットの構造

本項では移動ロボットの構造について解説する。

図 2-1 のように TurtleBot3 Waffle Pi に Azure Kinect とノートパソコンを搭載している。また、図 2-2 で示すようにロボットの進行方向とカメラの向きが異なるものとする。その理由は、人物は移動ロボットが旋回する円の中心にいるためである。TurtleBot3 Waffle Pi は差動二輪型ロボットであり、旋回はそれぞれのタイヤの回転の速度を異なる値に設定することで実現する。そのようにして旋回中のロボットが常に人物を画面に映すことができるためには、進行方向とカメラの向きを異なるものにする必要があるのである。それぞれの搭載機器については 2.2 で解説する。

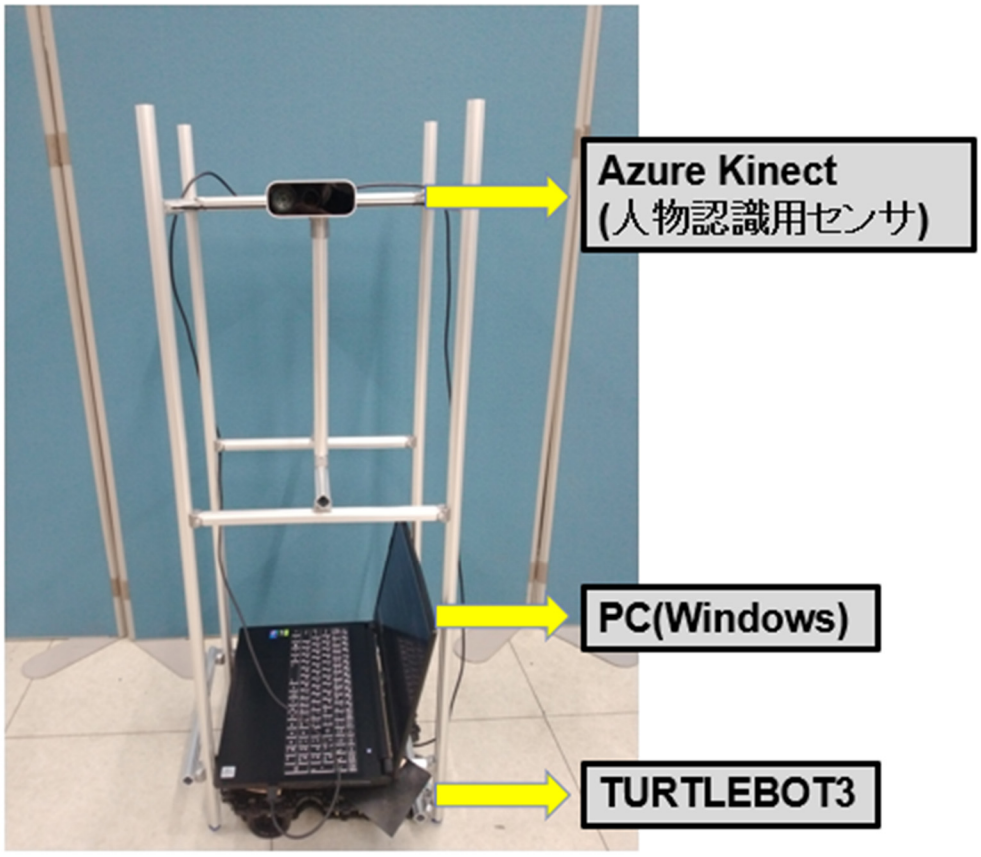


図 2-1　移動ロボット

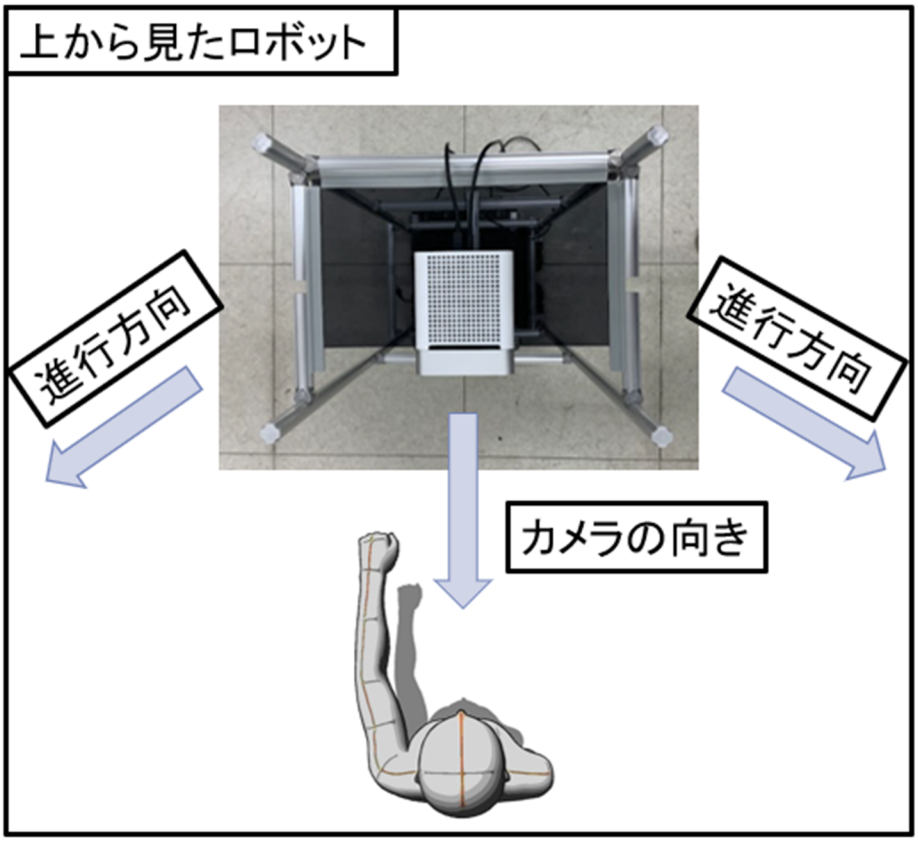


図 2-2　上から見た移動ロボット

2.1.1　人とロボットの関係

移動ロボットの動作のフローチャートを以下の図　2-3 に示す。

移動ロボットの動作の入力はユーザーが行うジェスチャーとし、出力はロボットが動くこととする。まず、ユーザーがロボットにジェスチャーをすると、その情報が Azure Kinect により読み取られる。認識されたジェスチャーによって対応するプログラムが実行され、ロボットが移動する。同時に UI アプリケーション上の情報も変化する。

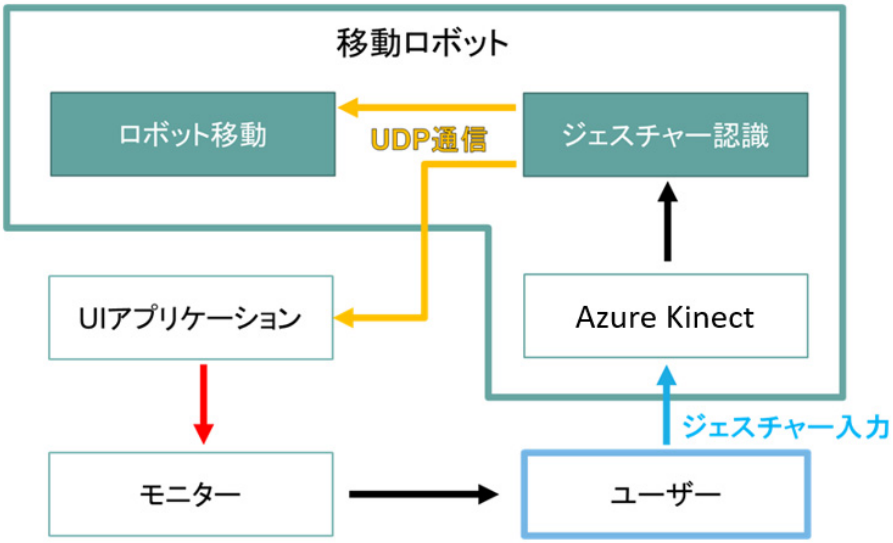


図　2-3　移動ロボットの動作のフローチャート

2.2　使用機器

2.2 では TurtleBot3 Waffle Pi、ROS 用 PC(Ubuntu)の詳細を述べる。また、TurtleBot3 Waffle Pi に搭載されている Raspberry Pi と OpenCR についての詳細を述べる。

2.2.1　TurtleBot3 Waffle Pi

ROS の入門者向けの標準的なロボットプラットフォームである TurtleBot3 Waffle Pi (図　2 -4)には移動ロボットに必要な標準装備が整っており、開発が容易となるため使用する。

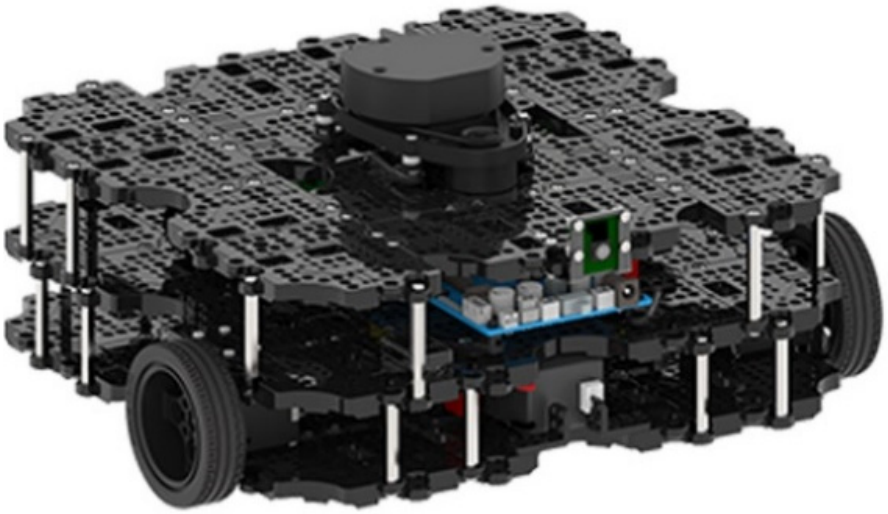


図　2 -4　TurtleBot3 Waffle Pi[2-1]

2.2.1.1　Raspberry Pi

図　2 -5 で示した Raspberry Pi はシングルボードコンピューターと呼ばれる超小型コンピューターであり、TurtleBot3 に搭載されている。TurtleBot3 上の Raspberry Pi が外部のパソコンからの命令を受け取ることで、TurtleBot3 が制御されるのである。

Raspberry Pi の OS として Ubuntu MATE 16.04 LTS を用いる。なお Ubuntu MATE 16.04 は 2016 年 2月に発表された Raspberry Pi 3 Model B までしかサポートしておらず、その後に発表された Raspberry Pi 3 Model B+や Raspberry Pi 4 では動作しないので注意が必要である。

この Ubuntu MATE 上に、Robot Operating System(以下、ROS) をインストールして利用する。ROS とはロボット用のソフトウェアプラットフォームのことである。

図　2 -5　Raspberry Pi 3 Model B[2-2]

2.2.1.2　OpenCR

図　2-6 に示したのが OpenCR と呼ばれるマイコンボードである。TurtleBot3 には OpenCR がロボット制御のため搭載されている。ROS がインストールされた Raspberry Pi が OpenCR に命令を送信し、OpenCR がロボットを制御する仕組みである。

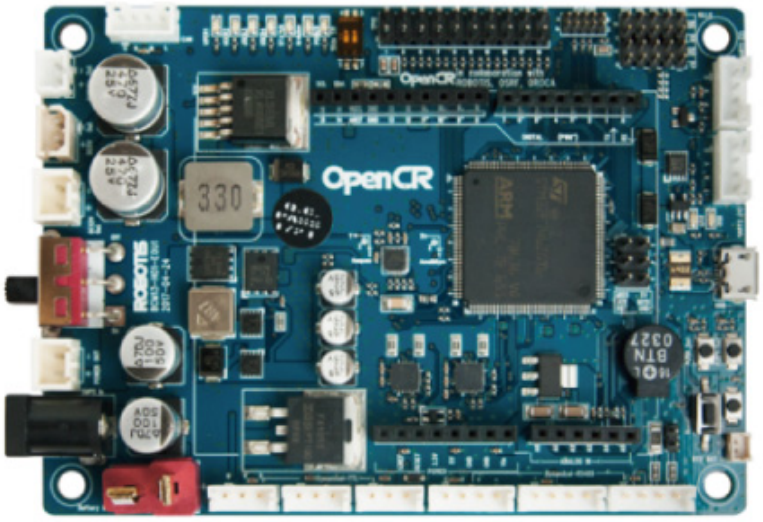


図　2-6　OpenCR[2-3]

2.3　ロボットに搭載する機器　*丸山拓己*

2.3.1　Azure Kinect 用 PC

表 2-1 に使用したノートパソコンの仕様を示す。このノートパソコン（図 2-7）は高性能な GPU を搭載しているため、Azure Kinect で取得した 3 次元座標位置の計算やプログラムで必要とする演算処理等ができる。さらに、Anaconda と Python をインストールしており Azure Kinect を使用するための環境が構築されている。

図 2-7　Azure Kinect 用 PC（iiyama 製）

表 2-1　Azure Kinect 用 PC の仕様

OS	Windows 10
CPU	Intel® CORE™ i7-10750H
RAM	16.0GB
GPU	NVIDIA GeForce GTX 1660 Ti

2.3.2　Azure Kinect

人物認識用センサとして Microsoft 社製の Azure Kinect（図　2 -8）を使用する。3 次元認識能力と深度測定能力のある AI カメラを搭載した開発者向けのデバイスであり、深度センサ、マイク、RGB カメラ、姿勢センサを搭載し、USB 給電で動作する。また、複数のモード、オプション、およびソフトウェア開発キットが用意されており、3D で体をトラッキングするための Body Tracking SDK などが開発環境として提供されている。また、ToF（Time-of-Flight）センサ技術を搭載しており深度センサによる奥行き検知精度を実現し、高性能な 3D アプリケーションを開発することができる。主な仕様を表　2 -2 に示す。

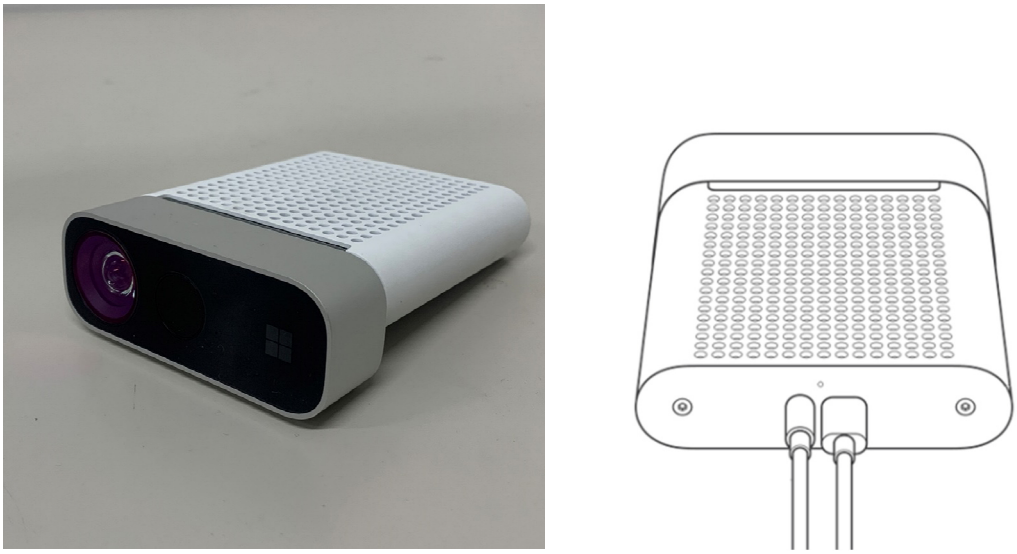


図　2 -8　Azure Kinect（斜め前図：後ろ図）[2-4]

図　2 -8 の右図のように Azure Kinect には電源ケーブル(電源に接続)と USB-C データケーブル(PC に接続)がある。

表 2-2 Azure Kinect の仕様[2-4/2-5]

寸法		103×39×126mm
重量		440g
最大消費電力		5.9W
関節		32 関節
入力/出力/接続性		USB-C データコネクタ USB-C または外部 PSU 経由での電力 複数の装置を同期する同期コネクタ
RGB センサ	最大解像度	4093×3072
	最大視野	90×74.3°
	最大 FPS	30
深度センサ	最大解像度	1024×1024
	動作範囲	0.5～5.46
	最大 FPS	30

図 2-9 のように Azure Kinect では 32 か所の関節座標を取得することができる。また図 2-10 が示すように Azure Kinect は三次元座標を取得することができる。この機能によって 360 度どの位置からでもジェスチャー認識が可能になる。

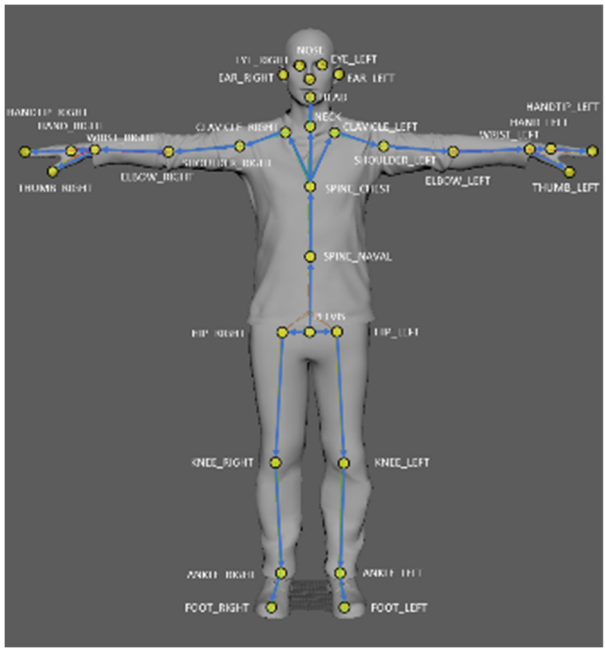


図 2-9 Azure Kinect が取得する関節[2-6]

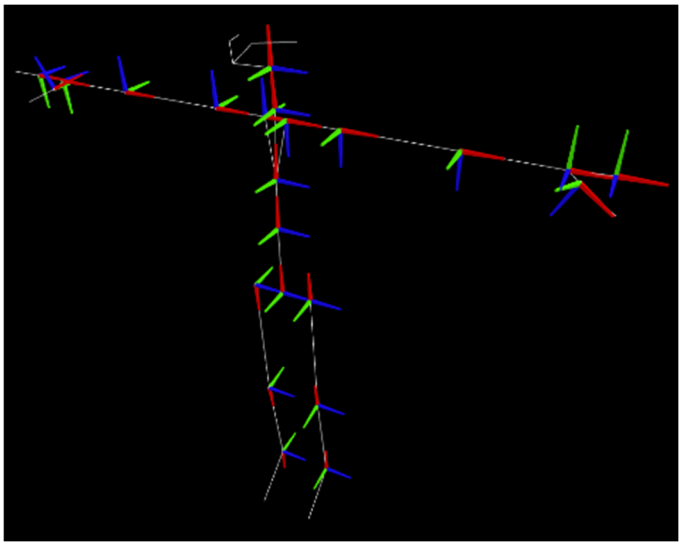


図　2-10　取得する関節の 3 次元座標イメージ(赤=X 軸,緑=Y 軸,青=Z 軸)[2-6]

2.3.3　バッテリー

Azure Kinect は図　2 -8 のように電源ケーブルをコンセントに接続して電力供給を行わなければならない。しかし、ロボットは人の周りを 360 度移動するため、コンセントからケーブルで繋いでしまうとケーブルがロボットの移動を妨げてしまう恐れがある。この問題を解決するために株式会社バファロー社製の BSMPB67182 Series モバイルバッテリー（図　2 -11）を使用する。主な仕様を表　2 -3 に示す。

図　2 -11　BSMPB67182 Series モバイルバッテリー[2-7]

表　2 -3　モバイルバッテリーの仕様[2-7]

動作環境	湿度 5℃～40℃、湿度 10%～85%
充電時間	約 2.44 時間
バッテリータイプ	充電式リチウムイオン電池
バッテリー容量	DC5.0V/6700mAh
入力	DC5.0V/3.0V
出力	[AUTO POWER SELECY モード] Type-C/Type-A ポート　合計 DC5.0V/3.0A
外寸(幅 x 高さ x 奥行)	45×115×24mm
質量	約 150g

2.3.4　ROS 用 PC(Ubuntu)

TurtleBot3 を制御するためには、図 2-7 の Azure Kinect 用 PC から、図 2-5 の Raspberry Pi へと命令を送信すればよい。そのためには、Raspberry Pi 上で命令を受け付けるプログラムを開発する必要がある。しかし、Raspberry Pi にはディスプレイやキーボードが接続されておらず、プログラム開発を行う難易度がやや高い。

そこで、Azure Kinect 用 PC と Raspberry Pi の間に図 2-12 に示す ROS インストール済ノートパソコン（Panasonic 製 Let's note CF-SZ5）を挿入することにする。この ROS 用 PC の OS としては Ubuntu16.04 LTS を用いる。そうすることで、

- Azure Kinect 用 PC→（命令）→Raspberry Pi

という命令の流れが以下のように変わる。

- Azure Kinect 用 PC→（命令）→ROS 用 PC→（ROS 命令）→Raspberry Pi

それにより、Azure Kinect 用 PC からの命令を受け付けるプログラムの開発を図 2-12 の ROS 用 PC で行えることになり、開発難易度を下げられるというメリットがある。なお、ROS 用 PC で実行した ROS 命令は、ROS の仕組みによりあたかも Raspberry Pi 上で実行したかのように振る舞う。

なお、Azure Kinect 用 PC は移動ロボット上に搭載されるが、この ROS 用 PC は移動ロボットの外部にあり、命令の受付や ROS 命令の実行は全て Wifi 経由で行われる。さらに、開発が完了すればこの ROS 用 PC の役割を全て Raspberry Pi に担わせることが可能である。

図 2-12　ROS 用 PC（Panasonic 製 Let's note CF-SZ5）

2.4　ハードウェアの開発　*堀越優来・吉武敏*

移動ロボットにAzure Kinectを取り付ける際、人間の全身が映るようにする必要がある。そのために、SUS株式会社のG-funシリーズを用いて図 2-13のように移動ロボットのフレームを製作した。

図 2-13　移動ロボットのフレーム

2.4.1　移動ロボットの要求仕様

以下に、Azure Kinect を取り付ける装置に必要な仕様を以下に示す。それぞれの詳細を 2.4.2 以降で述べる。

- フレーム同士が固定されており、さらに TurtleBot3 に組み立てたフレームが固定されていること
- 全身が映るように Azure Kinect を適切な高さに取り付けること
- 移動中にロボットが倒れないこと
- Azure Kinect がロボットから落下することを防ぐこと
- 必要な機器が搭載できる場所を確保すること

2.4.2　フレームの固定

フレームを組み立てるには、G-fun マルチコネクタインナーを用いる。このコネクタは G-fun のフレームであれば、図 2-14 のように至るところに直角に接続して固定することが可能である。

TurtleBot3 と骨組みの固定には TurtleBot3 にある既存の穴に G-fun シリーズのボードホルダを用いて 6 か所で固定する（図 2-15）。

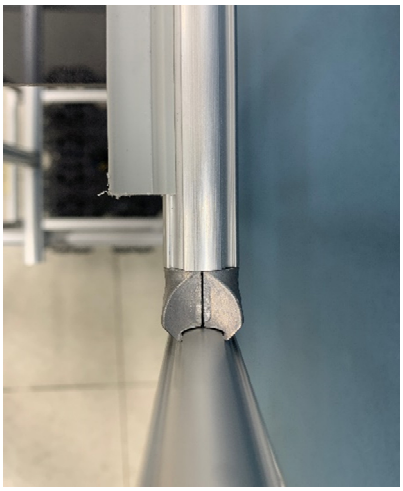


図 2-14　フレーム同士の固定

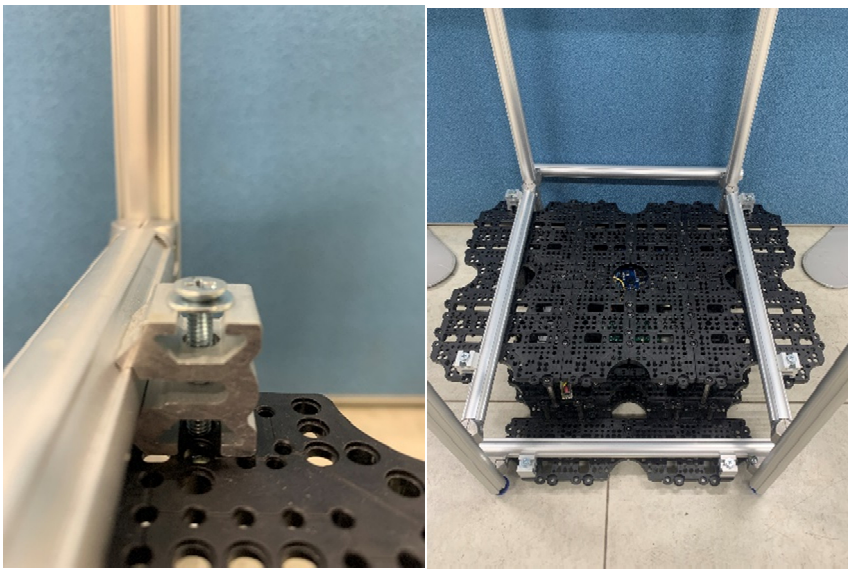


図 2-15　TurtleBot3 との固定方法

2.4.3　移動ロボットが展開する環境

我々が製作した移動ロボットによって実現される環境のイメージを図 2-16 に示す。Azure Kinect によって人のジェスチャーを認識するため、全身が映るように Azure Kinect を設置する必要がある。

ユーザーのジェスチャーを常に認識できるようにするために、以下の条件を満たす位置に Azure Kinect を設置する必要がある。

1．ユーザーの全身が常時 Azure Kinect の認識領域に含まれるよう距離をとること
2．Azure Kinect 内の映像で頭と足が見切れないように高さを設定すること

条件 1 を満たすために、ユーザーと Azure Kinect が最低 1700mm 離れるようにした。

条件 2 を満たすために、Azure Kinect を床から 920mm の高さに設置した。これ以上下げると、身長 180 ㎝以上のユーザーの頭が見切れてしまうためである。

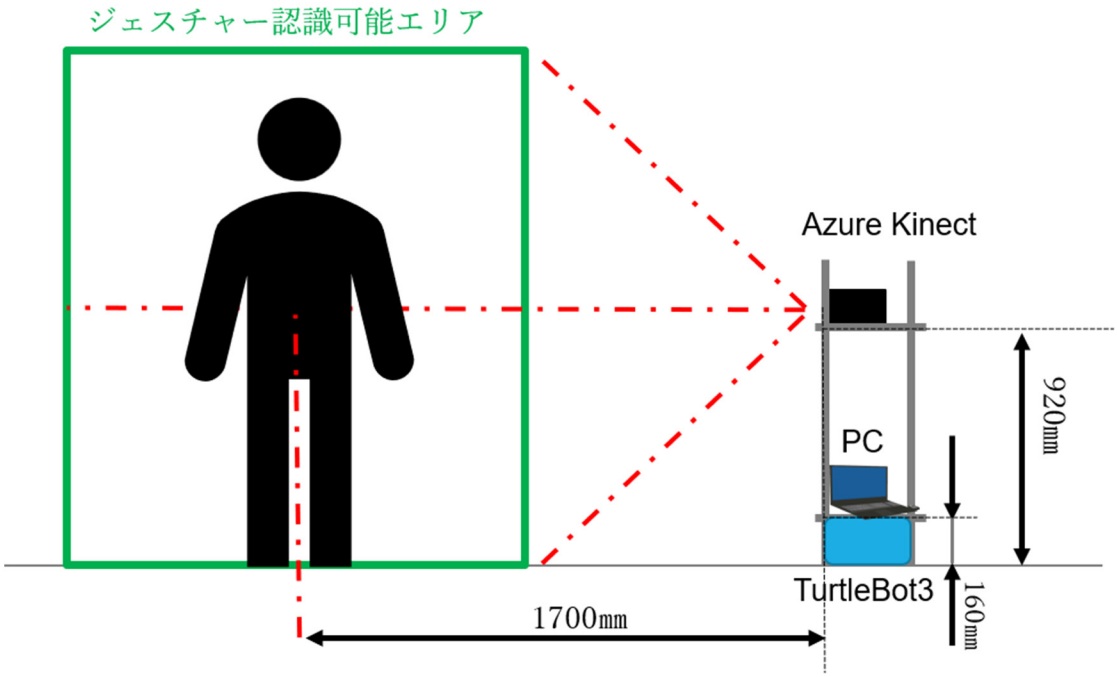


図 2-16　各機器の設置位置（寸法あり）

2.4.4　Azure Kinect の取り付け

移動ロボットに図 2-17（左）のようにアクリル板を取り付けた。

図 2-17（右上）のように Azure Kinect には M6 用のネジ穴があるため、アクリル板に M6 のネジが取り付けられるように穴を開けることで、Azure Kinect とアクリル板を固定した（図 2-18）。また図 2-18 の赤丸のように下からアクリル板を棒で抑えることで、ロボットの移動中に Azure Kinect の振動を抑制することができた。これにより Azure Kinect がロボットから落下することを防いでいる。

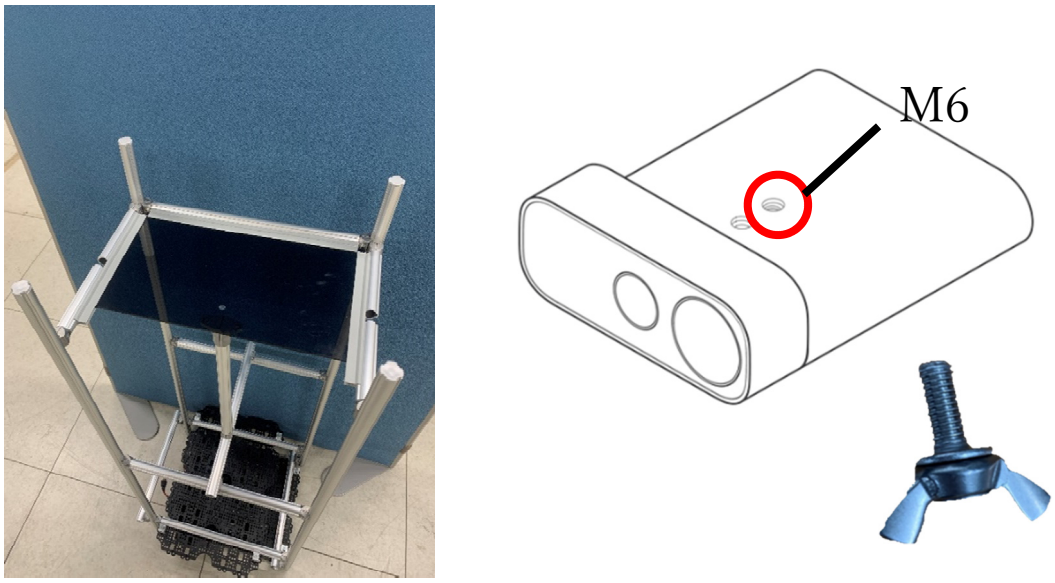


図 2-17　ネジ穴を開けたアクリル板（左）、Azure Kinect の裏面（右上）、M6 ねじ（右下）

図 2-18　Azure Kinect を取り付けたロボット

2.4.5　移動中にロボットが倒れることを防ぐ

移動ロボット上で Azure Kinect は高さ 920mm の位置にあるため、図 2-18 のように縦の寸法が横の寸法に対して非常に大きく、ロボットの動作によって横に揺れてしまう場合がある。そのため、図 2-19 のように支柱の下部の部分にボールキャスターを取り付けた。それによりフレーム自体を地面に接地させ、フレームの重量が TurtleBot3 だけにかかることを防ぎ、横に揺れることを軽減している。

図 2-19　支柱下部に取り付けたボールキャスター

2.5　ロボットに必要な機能　*丸山拓己*

ロボットを人の周りを自由に動かすには以下 4 つの機能 (図　2 -20) が必要であると考えた。

① 静止
② 時計回りに旋回
③ 半時計回りに旋回
④ ロボット移動モードのオン・オフ

それぞれについて以下に順に解説する。

①静止
ロボットを静止させたいときに必要である。
②&③時計回りに旋回&半時計回りに旋回
ロボットを自由に移動させるために必要である。
④ロボット移動モードのオン・オフ
仮想試着装置としての利用を想定すると、ロボットの移動を無効化させ、ユーザーが様々なポーズを取れるようにする仕組みが必要である。この仕組みがないと、意図せぬユーザーの動きに反応してロボットが動き出してしまう可能性がある。

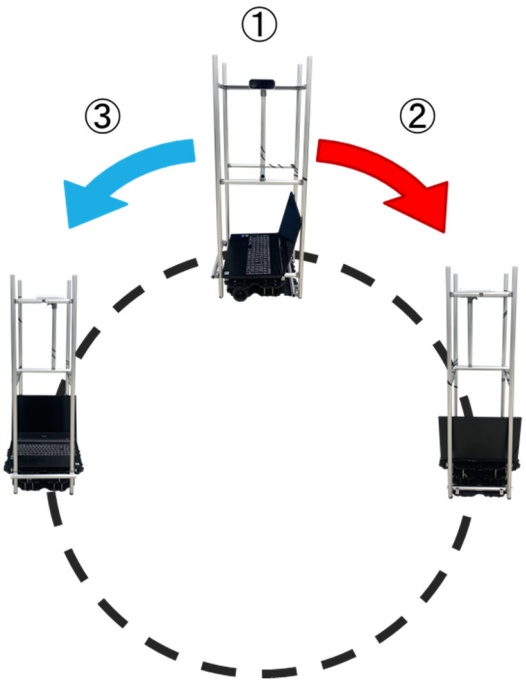


図　2 -20　ロボットの移動の模式図

2.6　UDP 通信

UDP 通信はジェスチャーの認識結果と、人物とロボットの距離に基づく制御データを送受信するために必要となる。無線 LAN 経由で IP アドレスとポート番号を用いて通信をする。TCP 通信と比べて信頼性は高くないが、高速であるというメリットがある。そのためリアルタイム性のあるデータを転送するのに適している。以下の図　2 -21 は機器間の通信関係を示したものである。本研究では、異なる PC（Azure Kinect 用 PC と Raspberry Pi）間だけではなく、同一 PC 内の異なるプログラム間で通信を行う際にも UDP 通信を用いていることが示されている。

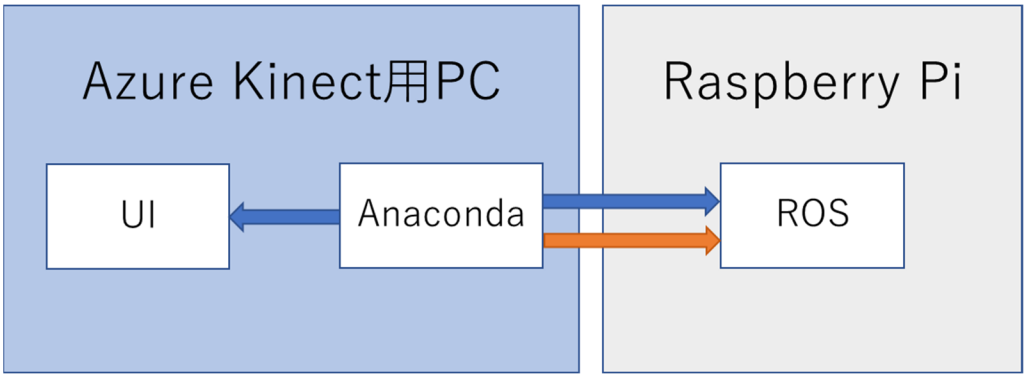


図　2 -21　UDP 通信の関係

Azure Kinect 用 PC 内では Python（Anaconda）から Unity（UI）にジェスチャーデータを送信する。そして Azure Kinect 用 PC の Python（Anaconda）から Raspberry Pi の ROS にはジェスチャーデータ（pose2）と人とロボットの距離に基づく制御データ（ds）を送信する（図　2 -22）。

```
ros_serv_address = ('192.168.1.8', 50002)#ROS用PC(IPアドレス,ポート番号)
serv_address = ('127.0.0.1', 50003) #Azure Kinect用PC(IPアドレス,ポート番号)

pose = (result[0])
pose_byte = bytes(result)
send_len = sock.sendto(pose_byte, serv_address)#Azure Kinect用PCにジェスチャーを送信

ds = struct.pack('>d',pid_dis(z0, x0))
pose2 = struct.pack('>d',pose)

send_len = sock.sendto(ds, ros_serv_address)#ROS用PCに距離を送信
send_len = sock.sendto(pose2, ros_serv_address)#ROS用PCにジェスチャーを送信
```

図　2 -22　送信側(Python)のプログラム

なお、2.3.4 で述べたように、図　2 -21 の Azure Kinect 用 PC と Raspberry Pi の間には、開発中には ROS 用 PC が挿入されている。図　2 -22 に「ROS 用 PC」との記載があるのは

そのためである。

このUDP通信により以下3つのことが可能になる。

- ジェスチャーによる移動ロボットの操作(4.1)
- 移動ロボットのP制御(4.2)
- ユーザーインターフェースの変化(5.2)

3　Azure Kinect を用いた人物認識

第 3 章では Azure Kinect を用いた人物認識について説明する。主に以下 3 項目を説明する。

- 使用するジェスチャーとコマンド設定(3.1)
- ジェスチャー認識(3.2)
- 認識精度実験(3.3)

3.1　使用するジェスチャー　*丸山拓己*

実店舗での仮想試着装置を使用することを想定し、以下の要件を満たすジェスチャーを選定した。

- ジェスチャーが感覚的でわかりやすいこと
- ジェスチャーが 360 度すべての角度から認識できること
- ジェスチャーが画面内に収まること

この条件を満たすものとして直立、右手前、左手前、腰に両手の 4 つのジェスチャーを使用する。そして直立を gesture1、右手前を gesture2、左手前を gestture3、腰に両手を gesture4 と設定する。ジェスチャーと移動ロボットの機能は以下の通りである（図　3 -1/P.26）。

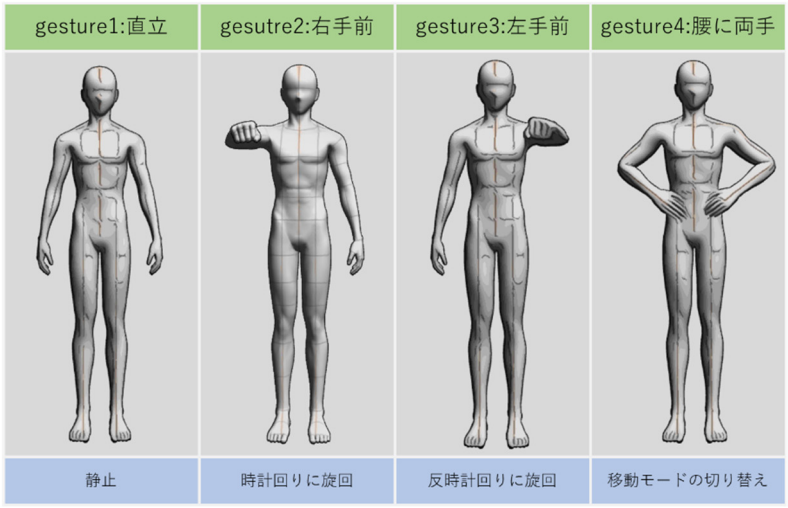


図　3 -1　各ジェスチャーと機能の対応

3.2　ジェスチャー認識

ジェスチャー認識を可能にするには事前に識別器にジェスチャーを学習させる必要がある。データ学習用プログラムでジェスチャー三次元座標データを取得し、学習用プログラムでそれぞれのジェスチャーを判別できるように学習する。その後、認識用プログラムを使うとジェスチャーを認識できるようになる。

Ⅰ　データ保存用プログラムで選定したジェスチャーの座標データを取得する（3.2.1）。
Ⅱ　学習用プログラムを用いて取得した座標データを学習させる（3.2.2）。
Ⅲ　認識用プログラムに学習データを取り込み、ジェスチャーを認識させる（3.2.3)。

3.2.1　座標データの取得

下の図　3 -2 はデータ保存用プログラムでデータ保存をしている画面である。左上に見える赤い数字は記録しているデータ数を表している。 4 つのジェスチャーに対して 100 個ずつデータを取得した。この時、1 個のデータに対して 32 か所の関節の三次元座標 (X,Y,Z) 合計 96 個の座標を取得している。

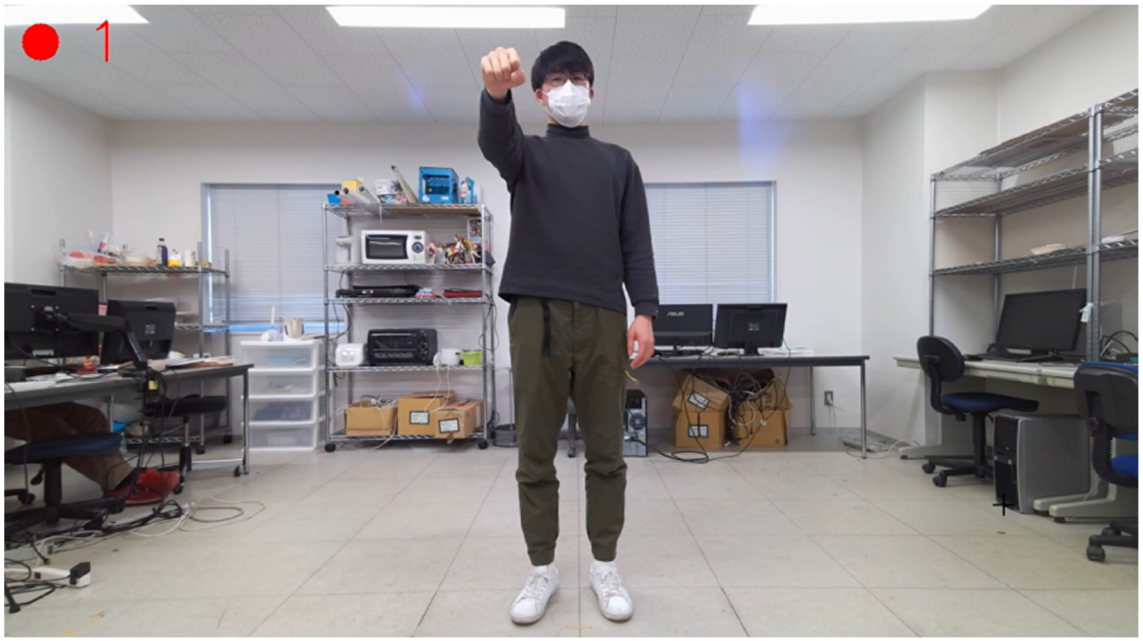


図　3 -2　データ保存用プログラムによるデータ取得時の PC 画面

3.2.2　ジェスチャー学習

取得した 400 個のデータを学習プログラムで学習させる。この時、ジェスチャーを 360 度から認識できるようにするため学習用プログラム内で 5 度ずつ座標データを回転させている（図　3 -3)。

```
for angle in range(0, 360, 5):
    cosval = math.cos(math.radians(angle))
    sinval = math.sin(math.radians(angle))
    for joint in range(num_joints):
        data2[joint, 0] = cosval*data[joint, 0] - sinval*data[joint, 2]
        data2[joint, 2] = sinval*data[joint, 0] + cosval*data[joint, 2]
        data2[joint, 1] = data[joint, 1]
```

図　3 -3　学習用プログラムの座標データ回転部分

3.2.3　ジェスチャー認識

認識用プログラムに学習したデータを読み込ませると、ジェスチャーを認識できるようになる。Azure Kinect の正面からそれぞれのジェスチャーを取った結果、図　3 -4 のようになった。左上に表示されているのがそれぞれのジェスチャー番号である。
3.1 で述べたように直立は gesture1、右手前は gesture2、左手前は gesture3、腰に両手は gesture4 と認識されているのがわかる。

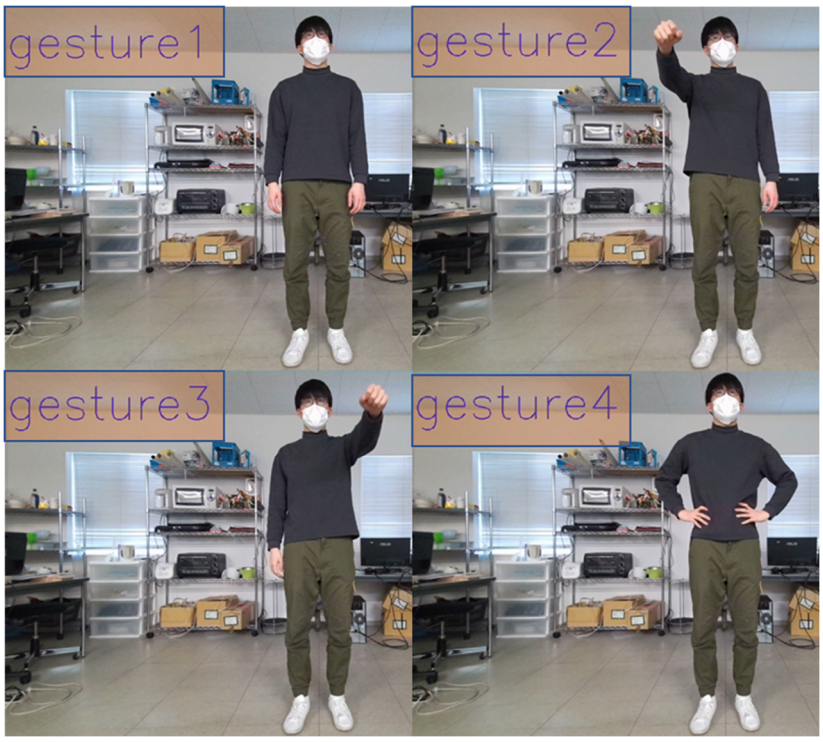


図　3 -4　認識用プログラムによるジェスチャー認識

3.3　認識率実験　*吉武敏*

我々の移動ロボットは、人物の周りを 360 度旋回することを想定しているため、360 度の周囲から人を認識したときの認識率を調べる。実験環境は図　3 -5 のように被験者と Azure Kinect の距離を 1700mm、Azure Kinect の高さを 920mm に設定した。被験者のつま先から頭までの全身を Azure Kinect が収めるためである（図　3 -6)。

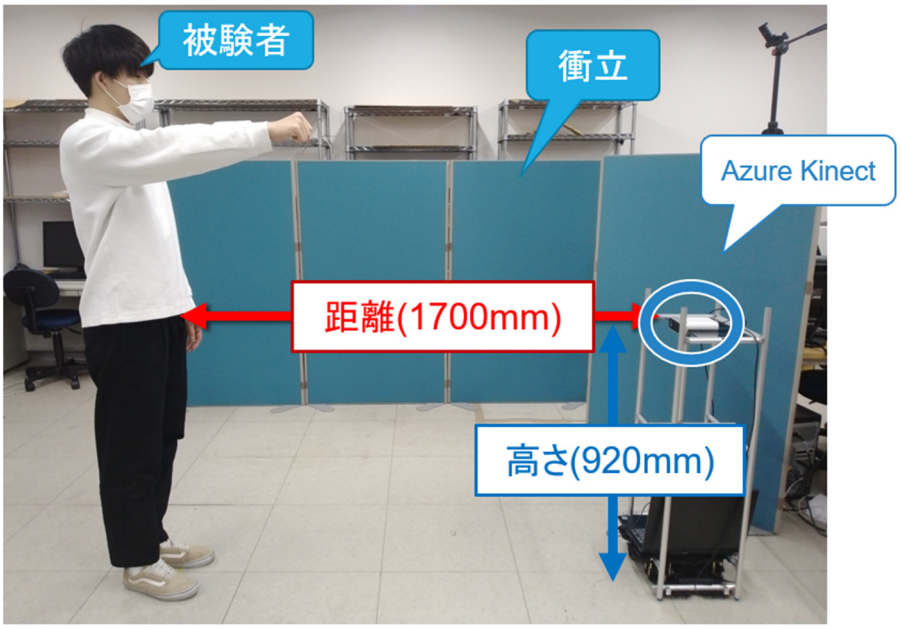


図　3 -5　人物の全身が映るよう Azure Kinect を設置した様子

図　3 -6　Azure Kinect から映した映像

以上の実験環境で認識率実験を行う。本実験の目的は以下の 2 つである。

1　4 つのジェスチャーの 360 度の周囲から人を認識したときの認識率を検証すること。

2　識別器にジェスチャーを登録しているユーザー（以下、登録者）と登録していないユーザー（以下、非登録者）の違いが認識精度に影響を及ぼすのかを検証すること。

3.3.1　実験方法

図 3-7（左）のように直立からそれぞれのジェスチャーを行い、3 秒間正しいジェスチャー認識が継続したら認識成功とみなす。それぞれ 100 回ずつジェスチャーを行った。

加えて、360 度の周囲から人を認識した時の認識率も検証するため、図 3-7（右）のように 360 度を 45 度ずつに分け、それぞれの地点からの認識率も調べた。

以上より、本実験はⅠ～Ⅴの手順で行う。

Ⅰ．4 つのジェスチャーが登録されている識別器を用意する。

Ⅱ．ジェスチャー 1 の直立状態から認識したいジェスチャーを行う。

Ⅲ．認識したいジェスチャーをとった状態から 3 秒間認識正しいジェスチャーの認識が継続したら、認識成功とみなす。

Ⅳ．100 回の実験が終了したら、角度を 45 度変更して同じジェスチャーを行う。

Ⅴ．0 度から 315 度まで計測できたら、ジェスチャーを変えⅡから行う。

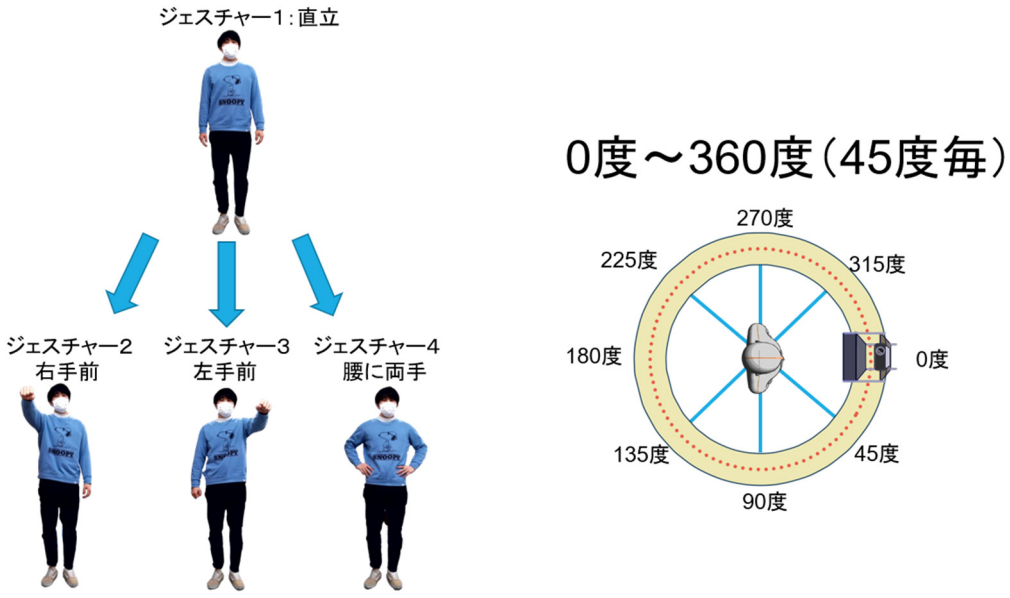


図 3-7　ジェスチャーの認識順序(左) 45 度毎に分け上から認識したイメージ図(右)

3.3.2　認識率結果

図　3-8～図　3-11 はジェスチャー登録者と非登録者の 4 つのジェスチャーの認識率を示している。「直立」と「腰に両手」は登録者、非登録者ともに全角度の認識率が 100%である。「右手前」(図　3-9)と「左手前」(図　3-10)は殆どの角度で 100%であるが、「右手前」は 225 度、「左手前」は 135 度の時、誤認識が見られた。

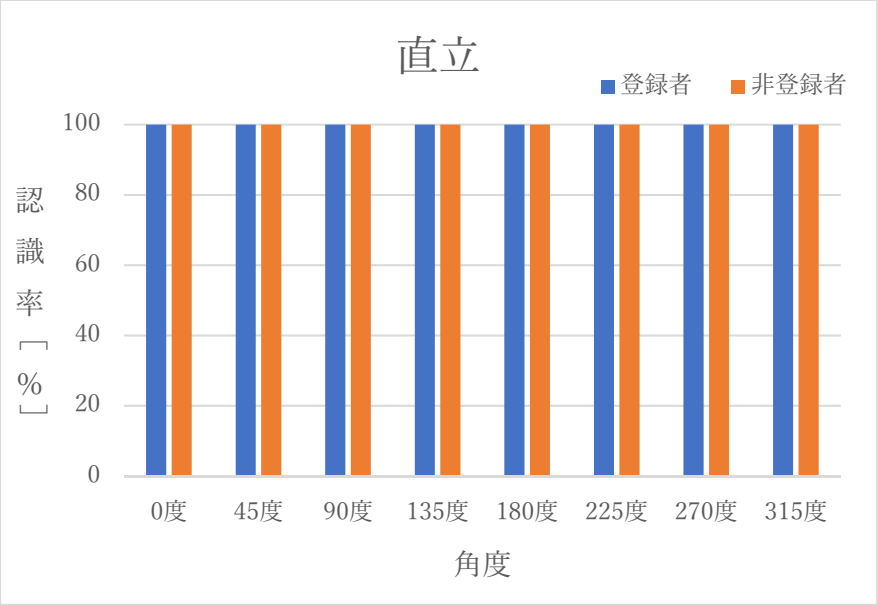


図　3-8　45 度毎の「直立」の認識率結果

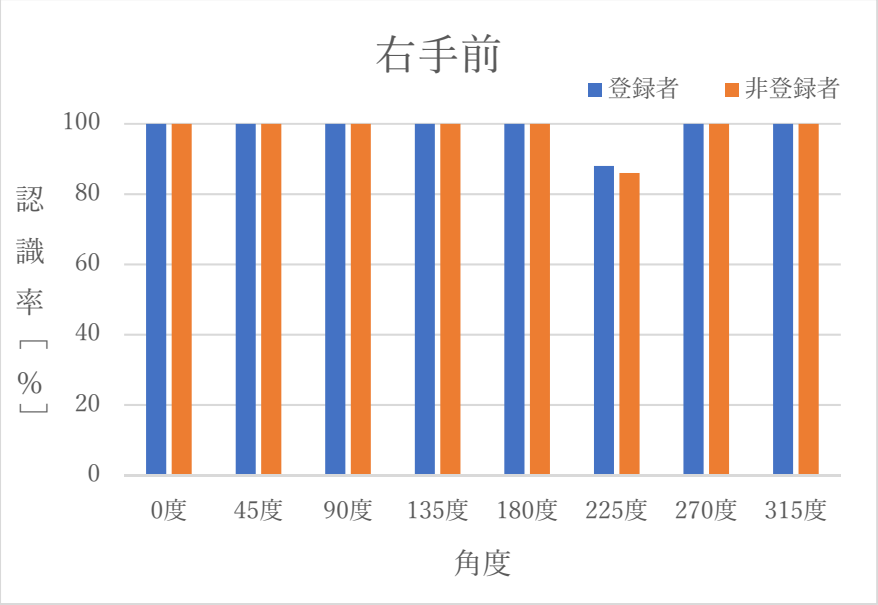


図　3-9　45 度毎の「右手前」の認識率結果

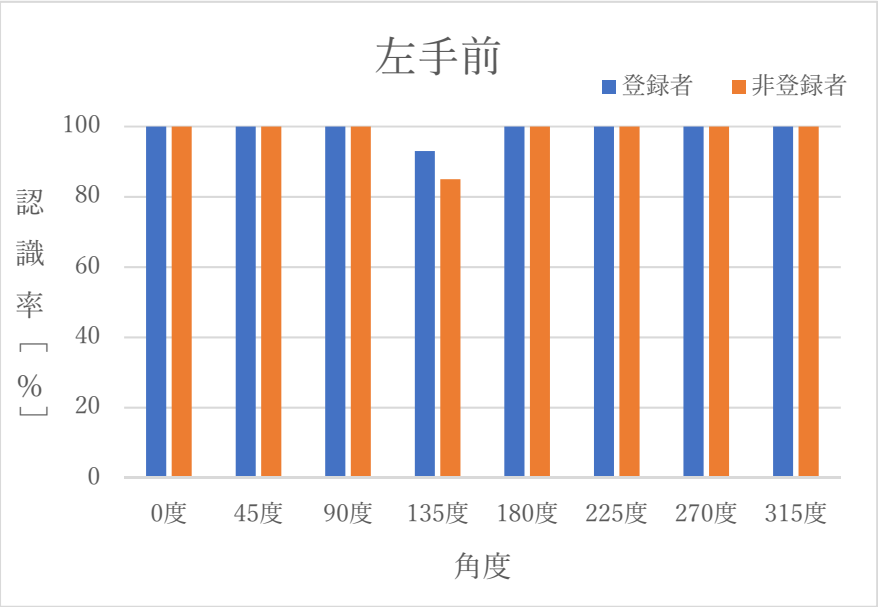


図　3-10　45 度毎の「左手前」の認識率結果

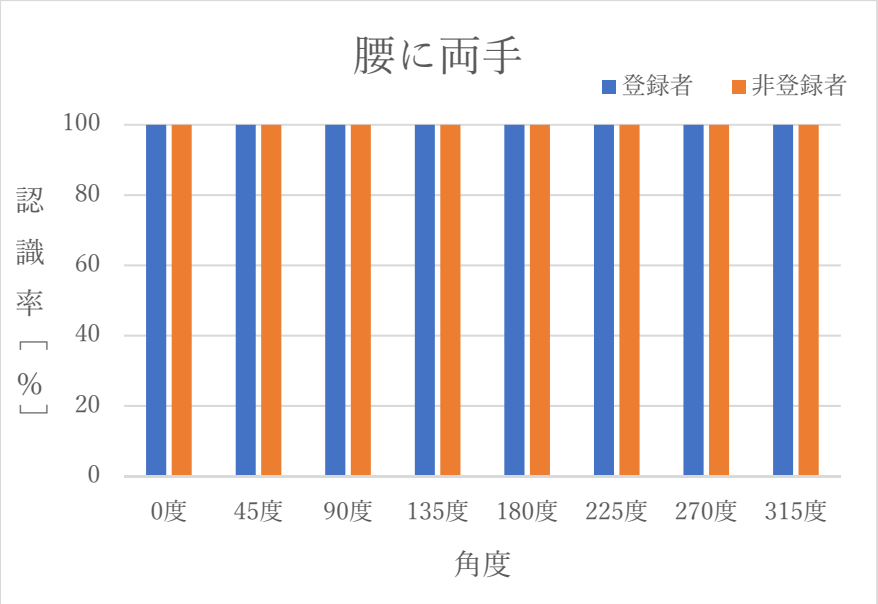


図　3-11　45 度毎の「腰に両手」の認識率結果

図　3-12 と図　3-13 は誤認識があった 225 度と 135 度の地点からジェスチャーを認識している様子である。225 度の時は「右手前」、135 度の時は「左手前」のジェスチャーをしているにも関わらず、共に「直立」と表示されている。この現象は前に出している腕が体の影になり見えていないため、「直立」と誤認識されていることがわかる。

図　3-12　225 度の時の認識映像（注：pose1=gesture1）

図　3-13　135 度の時の認識映像（注：pose1=gesture1）

図　3-14 はジェスチャー登録者と非登録者の角度ごとの４つのジェスチャーの認識率を平均した結果である。殆ど 100%となっており、登録者と非登録者の認識率の違いがみられない。また、それぞれ 90%以上の認識率であり、どの地点からでもジェスチャーの認識が可能である。そのため、我々が製作した移動ロボットの操作に選定した４つのジェスチャーを用いても、ロボットの操作に対する影響は軽微であると考える。

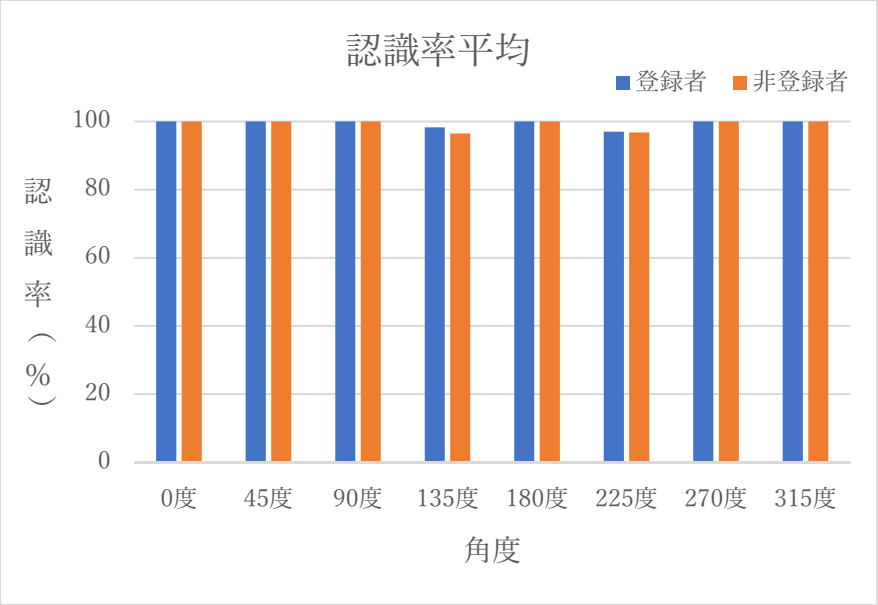


図　3-14　４つのジェスチャーの認識率を平均した結果

4　移動ロボットの制御　*堀越優来*

4.1　移動ロボットの制御

TurtleBot3 を制御するためには、ROS 上で実行される Python プログラムを実行する必要がある。2.3.4 で述べたように、このプログラムが実行されるのは、ROS 用 PC または Raspberry Pi 上である。このプログラムは Azure Kinect 用 PC から UDP 通信により送られてきたデータに基づき、TurtleBot3 を制御する。

プログラムのうち、TurtleBot3 に制御命令を送信している部分を抜粋したものが以下である。

```
set_data()                                          #データの設定
    if status==True:                               #移動モード ON
        print('True')
        cmd_vel_pub.publish(move_cmd)              #移動命令の送信
    else:                                          #移動モード OFF
        cmd_vel_pub.publish(stop_cmd)
```

変数 status は移動モードがオンかオフかの状態を格納しており、オンの時のみ動作命令（move_cmd）を発行している。オフの時は強制的に静止命令（stop_cmd）を発行している。

変数 move_cmd や stop_cmd は、TurtleBot3 を速度制御するために用いられている。以下のように前後移動と回転移動の速度に対してそれぞれ非 0 の値を設定することで移動ロボットは円軌道を描いて移動する。

```
move_cmd.linear.x          #前後移動
move_cmd.angular.z         #回転移動
```

この考え方を用いて、以下のように Azure Kinect から得られた認識結果をジェスチャー番号（value）として、移動ロボットの制御を行う。value の値で条件分岐することで、TurtleBot3 の動作を変更している。

なお、速度の値はここでは固定値を記したが、実際には TurtleBot3 と人物との距離を一定に保つため、回転速度の値（move_cmd.angular.z）は動作中に変動する。その点については次節で述べる。

```
def set_data():
    if value ==1:                          #直立（静止）
        move_cmd.linear.x =0
        move_cmd.angular.z =0
        count=0
    elif value ==2:                          #右手前（時計回りに旋回）
        move_cmd.linear.x =-0.15
        move_cmd.angular.z = 0.26
        count=0
    elif value ==3:                          #左手前（半時計回りに旋回）
        move_cmd.linear.x = 0.15
        move_cmd.angular.z = 0.26
        count=0
    elif value ==4:                          #腰に両手（移動モードの切り替え）
        move_cmd.linear.x =0
        move_cmd.angular.z =0
        switch_mode()
    else:
        move_cmd.linear.x =0
        move_cmd.angular.z =0
        count=0
```

移動ロボットの移動モードのオン・オフ切り替えは以下のようになっている。「腰に両手」のジェスチャーが 40 回連続で認識されると status の True と False が切り替わる。その結果 status が True となった場合、移動ロボットをジェスチャーによって動作させることができる。status が False となった場合、移動ロボットはどんなジェスチャーにも反応しない。

```
def switch_mode():
    global status
    global count
    if count>40:
        if status==False:
            status=True
            count=0
        else:
            status=False
            count=0
    count+=1
```

4.2　速度の制御方法

4.2.1　移動速度、回転速度の設定

研究初期の段階では、移動ロボットを一定の速度で旋回させるため、試行錯誤により速度の値を以下のように設定していた。この値はおよそ半径 1700mm の円軌道を描く値になっている。しかし、床の状態やわずかな段差により、厳密な円形を保って旋回することができなかった。それにより、ユーザーが映像範囲からはみ出してしまい、誤認識や周辺環境への衝突が生じてしまった。

```
move_cmd.linear.x=0.15          #前後移動
move_cmd.angular.z=0.26         #回転移動
```

上記の問題を解決するために必要な制御の条件を以下の通りに定める。

- ユーザーを常に映像範囲の中央に捉えること。
- ユーザーとの距離を常に一定に保つこと。

これらの条件を満たすため P 制御を用いた制御を導入する。

4.2.2　P 制御について

P 制御は目標値と取得値の偏差に対して比例ゲイン Kp を用いて操作量を調節してフィードバック制御をおこなう制御方式である。映像の中心に人物を収める制御、人物との距離を一定に保つ制御の両方において P 制御を行う。

映像の中心に人物を収める場合、図　4 -1 のように映像の中心とのズレが発生し、映像範囲の左側にユーザーがいるときロボットが左旋回することでユーザーが映像範囲の中心に戻るように制御する。

距離を一定に保つ場合も同様に、図　4 -2 のように目標値とのズレが発生したとき、遠い場合は人物に近づく方向に旋回し、近い場合は遠い方向に旋回する制御を行う。

P 制御は以下の式を用いた。

$$操作量=Kp\times(目標値-取得値)$$

この式をもとに操作量を以下のように設定した。

$$\begin{aligned}操作量=&Kp_{distance}\times(距離の目標値-距離の取得値)\\&+Kp_{center}\times(映像範囲の中心目標値-座標の値)\\=&\ Kp_{distance}\times距離の偏差\ +Kp_{center}\times中心からの偏差\end{aligned}$$

操作量を転回移動での速度の値とした。

```
def pid(dis_Actual,cen_Actual):
    dis_kp=0.001                    #距離の偏差に対する比例ゲイン
    cen_kp=0.001                    #中心からの偏差に対する比例ゲイン

    dis_b1=(dis_Actual-1700)
    dis_p=dis_kp*dis_b1             #距離の偏差に対する P 制御式

    cen_b1=(cen_Actual-0)
    cen_p=cen_kp*cen_b1             #中心からの偏差に対する P 制御式

    return dis_p+cen_p              # 制御量
```

この計算は Azure Kinect による認識プログラム内で行い、計算結果を ROS 用 PC に UDP

通信で送信する。

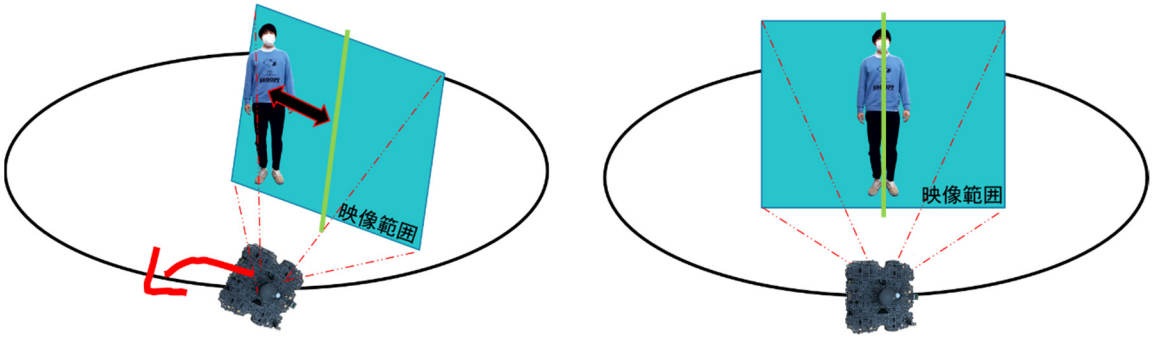


図　4 -1　被験者を映像中心に収める制御

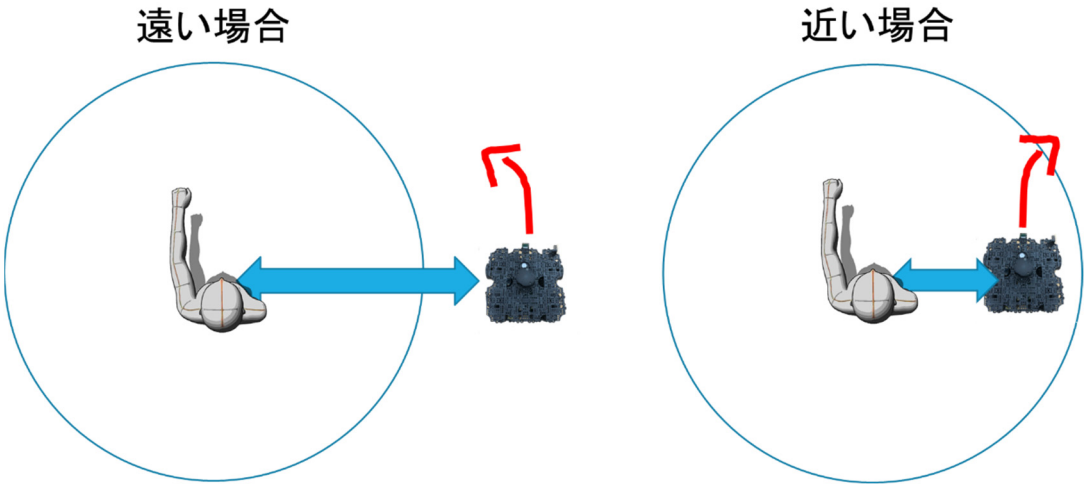


図　4 -2　一定の距離を保つ制御

4.3　評価実験

4.3.1　評価方法

考案した制御方法を評価するため、被験者の周りをロボットに旋回させ、P 制御ありと P 制御なしの条件で 10 回ずつ実験を行う。Azure Kinect によって得られた骨盤までの距離を移動ロボットと人物の間の距離として計測する。

4.3.2　実験環境

図　4 -3 のように被験者と Azure Kinect の距離を 1700mm、Azure Kinect の高さを 920mm に設定した。この実験環境は、認識実験と同様であり、被験者のつま先から頭までの全身を Azure Kinect がとらえられるように定めた。

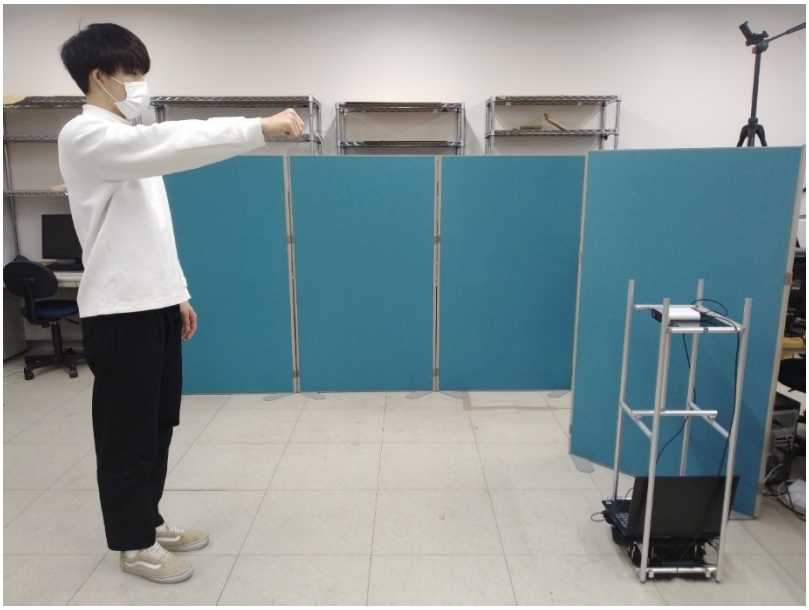


図　4 -3　実験環境

4.3.3　測定結果

実験の結果を以下の図　4 -4、図　4 -5 に示す。横軸は時間に相当し、縦軸の距離が 1700mm に保たれていることが望ましい挙動である。

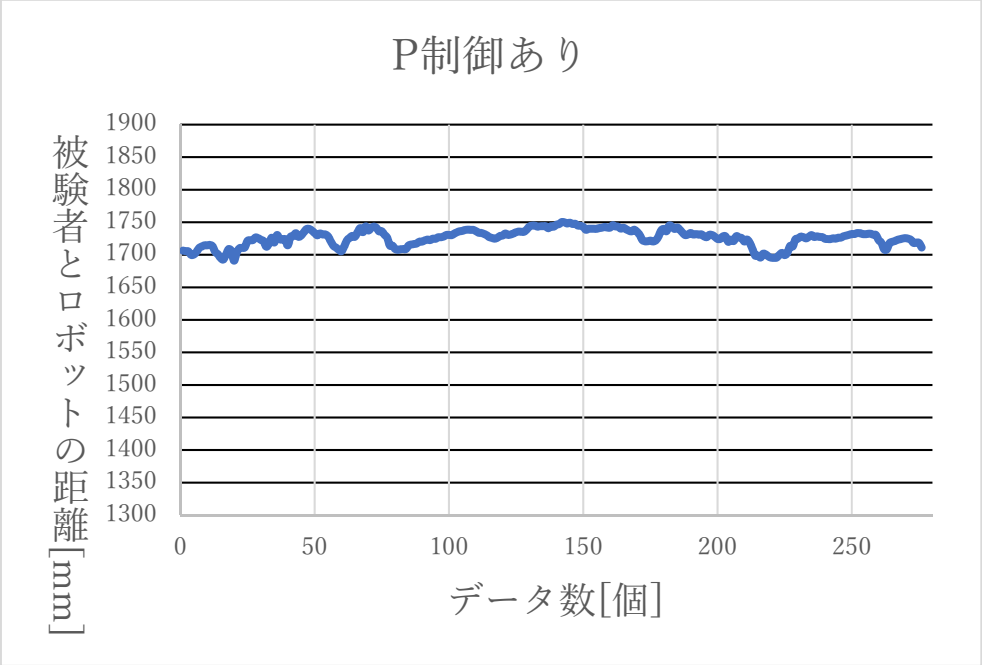


図　4 -4　P 制御あり実験の測定結果(一例)

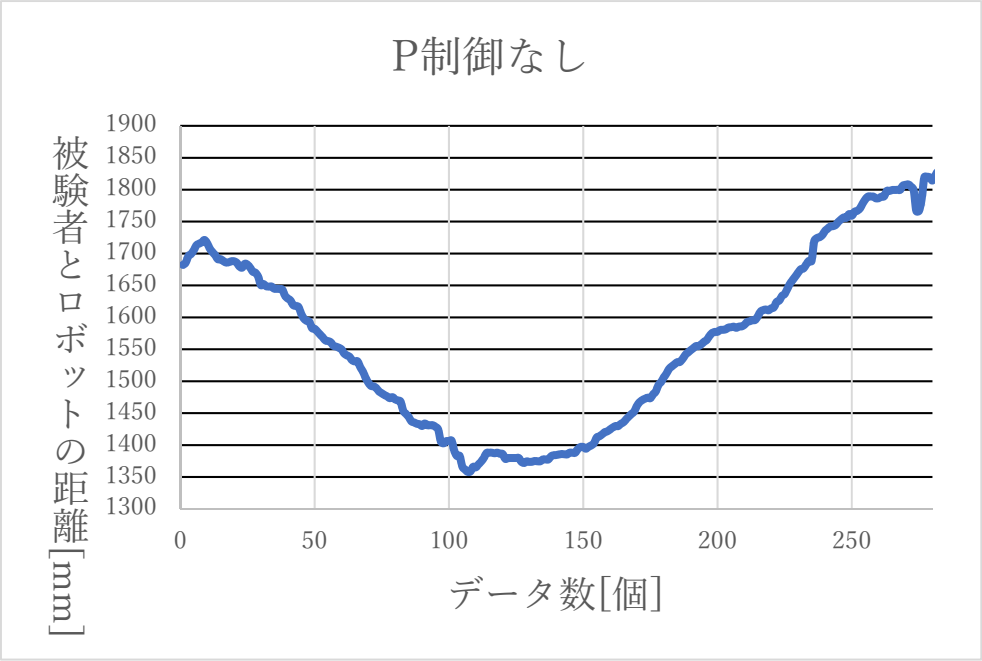


図　4 -5　P 制御なし実験の測定結果(一例)

4.4　評価

図　4 -4、図　4 -5 の実験の結果をまとめたのが図　4 -6 である。棒グラフと緑の数値が距離平均を表し、赤の数値が最大距離、青の数値が最小距離を示し、矢印が変動の幅を示している。

P 制御ありの距離平均のほうが P 制御なしの距離平均より目標値である 1700mm に近い数値になっている。また、最大距離と最小距離の変動の幅が小さく、より正確に一定の距離が保たれていることがわかる。

P 制御ありの場合でも多少の変動があるが、被験者が直立していても重心移動などでバランスを取ることがあり、体がかすかに動いてしまうことが原因であると考えられる。また、床などの環境によっても数値が変動すると考えられる。

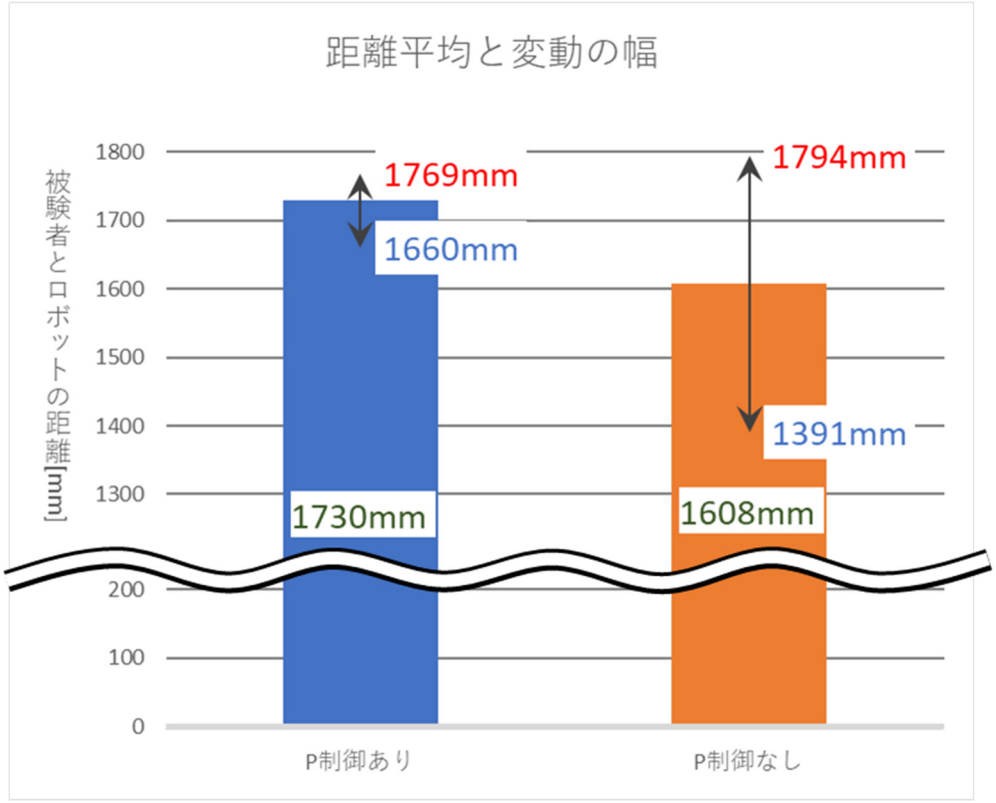


図　4 -6　距離平均と変動の幅

5　ユーザーインターフェース　*目黒新大*

5.1　ユーザーインターフェースについて

想定している移動型仮想試着装置は、図 5-1 のように Azure Kinect の映像をユーザーが見ることになるため、情報を把握しやすいようにユーザーインターフェースを作成した。以下、ユーザーインターフェースを UI と称する。

なお、図 5-1 では移動ロボット上の Azure Kinect 用 PC の映像を無線により外部モニターに転送することを想定しているが、本研究ではそこまでは実行していない。

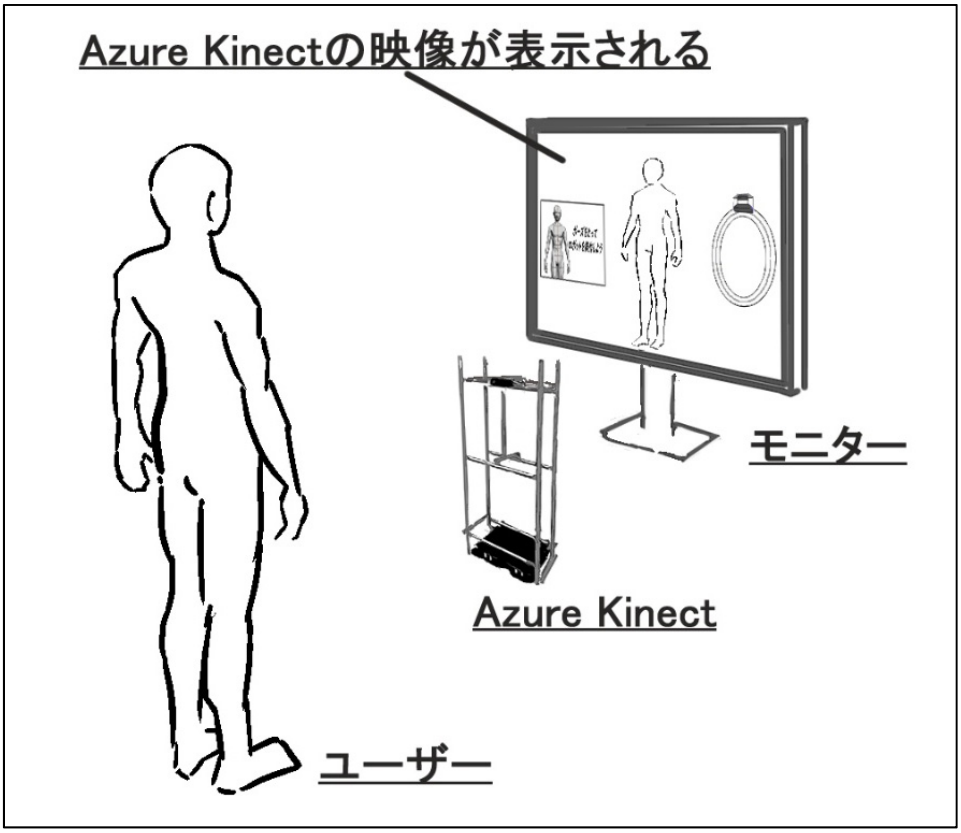


図 5-1　構想図

5.2　UI に必要な情報及び実装

本装置におけるユーザーにとって必要な最低限の情報は、「映像」、「認識されているジェスチャー」、「ロボットの位置」である。

「映像」はリアルタイムで Azure Kinect から撮影したものを表示する必要があるが、一つのデバイスを複数アプリで同時に使用できない関係上、既に Anaconda で使用している Azure Kinect を UI アプリケーションで使用することはできない。そのため、Anaconda で出力した映像ウィンドウを UI を作成する Unity へと転写することにした。

「認識されているジェスチャー」は、図 5-2 のように、どのジェスチャーが装置に入力されているかと装置がどの動きをするかをユーザーが把握するために必要である。Anaconda で得たジェスチャーデータを UDP 通信で受け取り、それに対応した画像に変化させる。

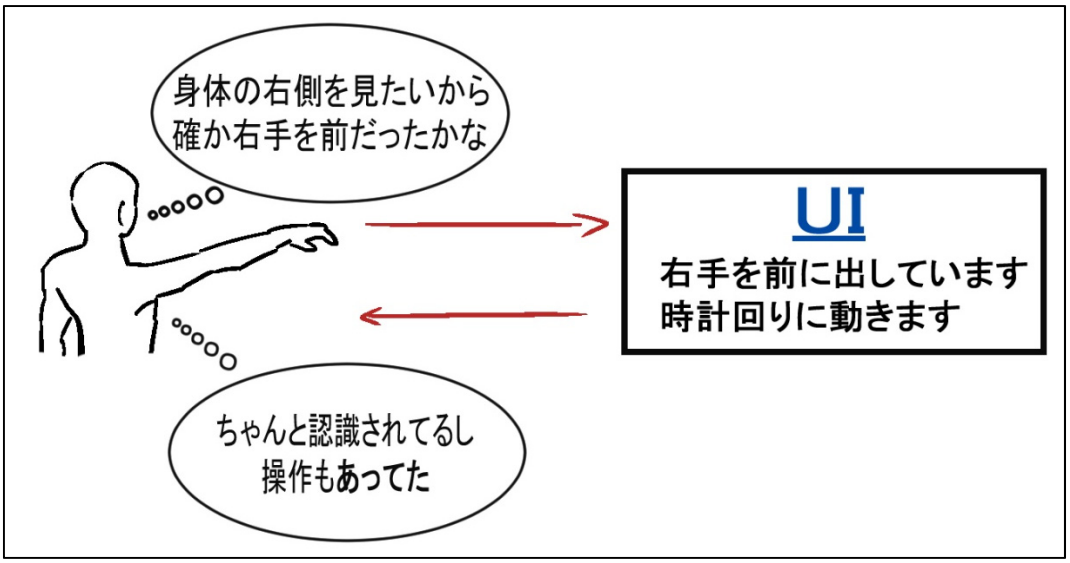


図 5-2　認識されているジェスチャー表示の必要性

「ロボットの位置」は、ユーザーの死角にロボットが移動した場合、円滑な操作をするために必要である。図 5-3 は、ユーザーが見る映像にロボット位置表示が有る場合と無い場合の比較である。ロボットの位置表示があることで、ロボット移動の時計回り・反時計回りの判断を瞬時に行うことができる。

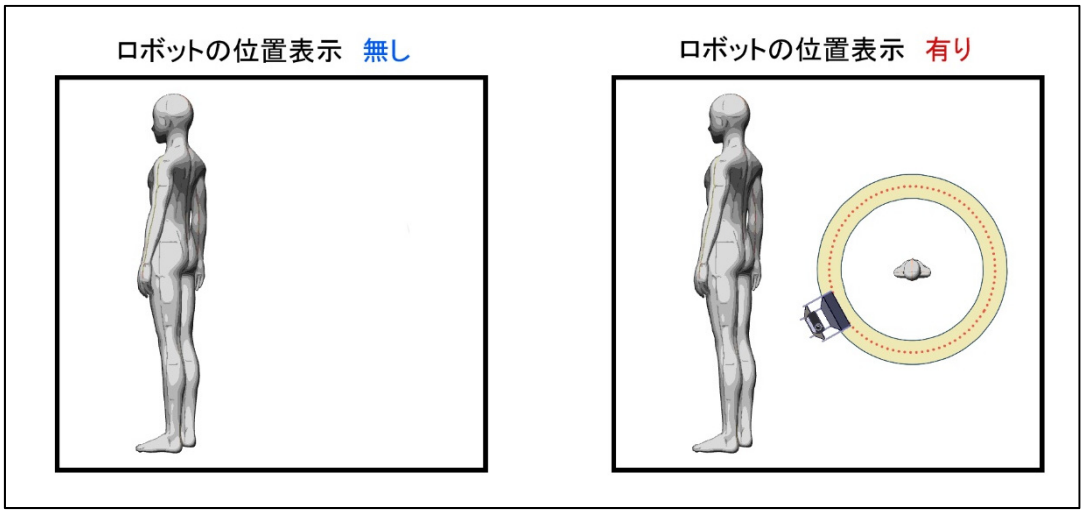


図 5-3 ロボットの位置表示の有無の比較

ロボットの位置を表示する場合、ロボットが自己位置を推定し、その結果を受信することが望ましいが、本装置に自己位置推定する機能を搭載していない。従って、ロボットに送られたジェスチャーデータの履歴からロボットの位置を推定することにした。ただし地面環境などの要因で実際のロボットの移動は毎回同じにはならない。加えて、UI 上では P 制御を考慮していないため、多少のずれが生じることに注意する必要がある。

5.3　作成した UI 概要

図　5 -4 が作成した UI である。ユーザーの正面のモニターに、UI が適用された映像が映ることを想定している。赤枠内に「現在認識されているジェスチャー」或いは「ロボットの状態」が表示される。緑枠内にはモニターの位置を上側にした真上からのロボットの推定位置が表示される。図　5 -5、図　5 -6、図　5 -7 がジェスチャーを行った画像である。人物のジェスチャーとロボットの位置に対応して左右の UI を正しく変化していることが分かる。また図　5 -7 で行っている腰に両手に関しては、ロボットの移動モードのオンとオフが可視化されるようになっている。

図　5 -4　ユーザーインターフェース画面

図　5 -5　左手を前にしたときの UI

図 5-6　右手を前にしたときの UI

図 5-7　腰に両手を置いているときの UI

6　結論　*目黒新大*

本研究では、移動型仮想試着装置における「ジェスチャーで操作する移動ロボット」の部分に注力し、大きく分けて 3 つの目標を掲げ「ジェスチャーで操作する移動ロボット」の開発を行った。

1. 様々な角度からジェスチャー認識ができること
2. 認識したジェスチャーによるロボットの操作ができること
3. ロボットと人物の距離を一定に保つ制御ができること

1 の目標については、Azure Kinect を使用したことにより、人物の周りを移動しながら人物を認識することに成功した。

2 の目標については、UDP 通信でジェスチャーデータを ROS に送り、それに対応した命令が実行されるようにしたことで達成した。

3 の目標については、UDP 通信でジェスチャーデータと同時に対象までの距離に基づく制御データを送り P 制御を行うことでロボットと人物の距離を一定に保つことに成功した。

また上記条件の他に、移動型仮想試着装置を想定し、ソフトウェア開発として、情報を把握しやすいように UI を作成した。

7 参考文献

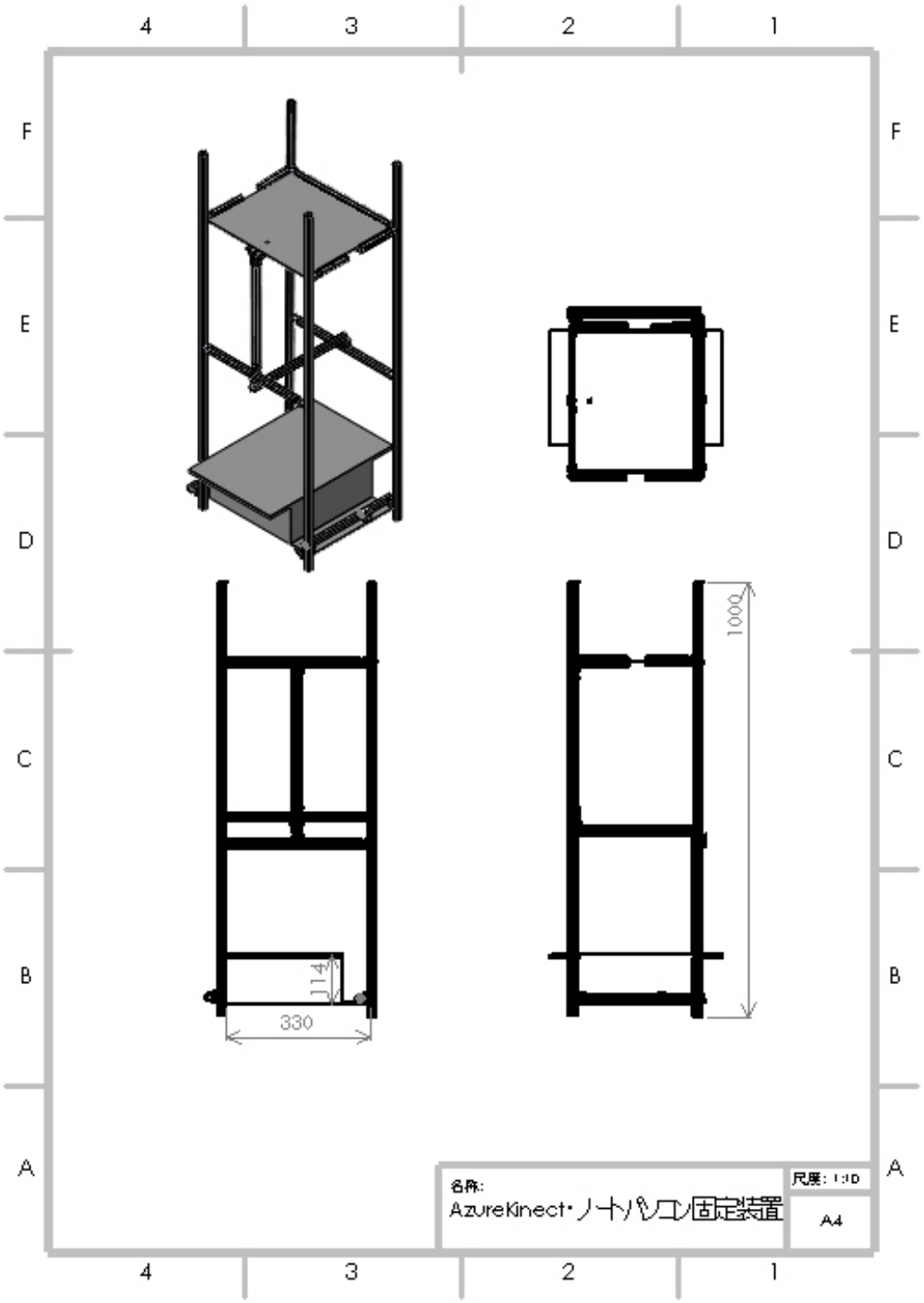
[1-1] 「コロナショック前後のモノとの接触」に関する意識調査
https://www.asukanet.co.jp/contents/news/2020/20200525.html
[1-2] 実店舗で試着することに抵抗を感じる割合
https://www.atpress.ne.jp/releases/199901/att_199901_1.pdf
[1-3] 先行研究の撮影補助装置
https://brain.cc.kogakuin.ac.jp/research/202003_RoombaThesis.pdf
[2-1] TurtleBot3 Waffle Pi
https://e-shop.robotis.co.jp/list.php?c_id=93
[2-2] Raspberry Pi 3 Model B
https://www.iodata.jp/product/pc/raspberrypi/ud-rp3/spec.htm
[2-3] OpenCR
https://emanual.robotis.com/docs/en/parts/controller/opencr10_jp/
[2-4] Azure Kinect ハードウェアの仕様
https://docs.microsoft.com/ja-jp/azure/kinect-dk/hardware-specification
[2-5] Azure Kinect
https://www.microsoft.com/ja-jp/d/azure-kinect-dk/8pp5vxmd9nhq?activetab=pivot:techspecstab
[2-6] Azure Kinect ボディトラッキングの関節
https://docs.micROSoft.com/ja-jp/azure/kinect-dk/body-joints
[2-7] BSMPB67182 Series　モバイルバッテリー
https://www.buffalo.jp/product/detail/bsmpb6718c2bk.html

謝辞

最後に、この場をお借りして本研究を進めるにあたり、2年間多大なご指導を頂きました金丸隆志教授、協力して頂いた研究室の先輩方、学科の同級生に班一同、心より感謝いたします。

付録

移動ロボットの CAD 図面　*宮崎生望*



移動ロボットのプログラム

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 28 13:24:39 2021

@author: student
"""
import socket
import struct
import rospy
from geometry_msgs.msg import Twist

h = 0
M_SIZE = 2048
status=False
count=0
#waffle_pi
cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size = 1)

rospy.init_node('move')

move_cmd = Twist()
stop_cmd = Twist()

def switch_mode():
    global status
    global count
    if count>40:
        if status==False:
            status=True
            count=0
        else:
            status=False
            count=0
    count+=1
```

```
def set_data():
    if value ==1:
        move_cmd.linear.x =0
        move_cmd.angular.z =0
        count=0
    elif value ==4:
        cmd_vel_pub.publish(stop_cmd)
        switch_mode()
    elif value ==2:
        move_cmd.linear.x =-0.20
        move_cmd.angular.z =0
        count=0
    elif value ==3:
        move_cmd.linear.x = 0.20
        move_cmd.angular.z =0
        count=0
    else:
        move_cmd.linear.x =0
        move_cmd.angular.z =0
        count=0

def testprint():
    print('OK')

def move_game():
    if move_cmd.linear.x==0 and move_cmd.angular.z==0:
        cmd_vel_pub.publish(stop_cmd)
    else:
        move_cmd.angular.z = move_cmd.angular.z - h
        print(move_cmd.angular.z)
        cmd_vel_pub.publish(move_cmd)

def main():
    host_ip = '192.168.1.8'#ros laptop
    #host_ip = '192.168.1.9'#raspi
```

```
port = 50002
client_ip = '192.168.1.64'
client_address = ('192.168.1.64', 50000)
#locaddr = (client_ip, port)
rate = rospy.Rate(10)
# ①ソケットを作成する
sock = socket.socket(socket.AF_INET, type=socket.SOCK_DGRAM)
print('create socket')

# ②自ホストで使用する IP アドレスとポート番号を指定
sock.bind(("",50002))#port のみ指定
status
while not rospy.is_shutdown():
    try:
        # ③Client からの message の受付開始
        #print('start')
        print('Waiting message dis')
        distance, cli_addr = sock.recvfrom(M_SIZE)
        distance_int=distance
        print('distance',distance_int)
        #print(str(struct.unpack('>d',distance_int)[0]))
        global h
        h=float(str(struct.unpack('>d',distance_int)[0]))
        print(h)

        print('Waiting message1')
        pose, cli_addr = sock.recvfrom(M_SIZE)
        pose_int=pose
        print('pose',pose_int)
        print(str(struct.unpack('>d',pose_int)[0]))

        #print('finish')

        global value
        value=float(str(struct.unpack('>d',pose_int)[0]))
```

```
                set_data()
                if status==True:
                    print('True')
                    move_game()
                else:
                    cmd_vel_pub.publish(stop_cmd)
                    #False の時は止まる（stop_cmd）
            except KeyboardInterrupt:
                print ('¥n . . .¥n')
                sock.close()
                break

if __name__=='__main__':
    try:
        main()
    except KeyboardInterrupt:
    #except rospy.ROSInterruptException:
        pass
```

ジェスチャー認識用データ保存プログラム

```
"""
This is a python port of the simple-sample project from.
    https://github.com/microsoft/Azure-Kinect-
Samples/blob/bf2f8cf95d969dcc7842c4c450052fe5a943c756/body-tracking-
samples/simple_sample/main.c
"""

import traceback
import sys
import ctypes
import os
import cv2
import time
import datetime
import numpy as np

# Add .. to the import path
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

import k4a

def VERIFY(result, error):
    if result != k4a.K4A_RESULT_SUCCEEDED:
        print(error)
        traceback.print_stack()
        sys.exit(1)

def print_body_information(body):
    print("Body ID: {}".format(body.id))
    for i in range(k4a.K4ABT_JOINT_COUNT):
        position = body.skeleton.joints[i].position
        orientation = body.skeleton.joints[i].orientation
        confidence_level = body.skeleton.joints[i].confidence_level
        print("Joint[{}]: Position[mm] ( {}, {}, {} ); Orientation ( {}, {}, {}, {}); Confidence
Level ({})".format(
```

```
                i, position.v[0], position.v[1], position.v[2], orientation.v[0], orientation.v[1],
orientation.v[2], orientation.v[3], confidence_level
            ))

def print_body_index_map_middle_line(body_index_map):
    print("print_body_index_map_middle_line not implemented")
    """
    uint8_t* body_index_map_buffer = k4a_image_get_buffer(body_index_map);

    // Given body_index_map pixel type should be uint8, the stride_byte should be the same
as width
    // TODO: Since there is no API to query the byte-per-pixel information, we have to
compare the width and stride to
    // know the information. We should replace this assert with proper byte-per-pixel query
once the API is provided by
    // K4A SDK.
    assert(k4a_image_get_stride_bytes(body_index_map)                                    ==
k4a_image_get_width_pixels(body_index_map));

    int middle_line_num = k4a_image_get_height_pixels(body_index_map) / 2;
    body_index_map_buffer = body_index_map_buffer + middle_line_num *
k4a_image_get_width_pixels(body_index_map);

    printf("BodyIndexMap at Line %d:¥n", middle_line_num);
    for (int i = 0; i < k4a_image_get_width_pixels(body_index_map); i++)
    {
        printf("%u, ", *body_index_map_buffer);
        body_index_map_buffer++;
    }
    printf("¥n");
    """

if __name__ == "__main__":
    device_config = k4a.K4A_DEVICE_CONFIG_INIT_DISABLE_ALL
    device_config.depth_mode = k4a.K4A_DEPTH_MODE_NFOV_UNBINNED
```

```
    device = k4a.k4a_device_t()
    VERIFY(k4a.k4a_device_open(0, ctypes.byref(device)), "Open K4A Device failed!")
    VERIFY(k4a.k4a_device_start_cameras(device, ctypes.byref(device_config)), "Start K4A cameras failed!")

    sensor_calibration = k4a.k4a_calibration_t()
    VERIFY(k4a.k4a_device_get_calibration(device, device_config.depth_mode, k4a.K4A_COLOR_RESOLUTION_OFF, ctypes.byref(sensor_calibration)), "Get depth camera calibration failed!")

    tracker = k4a.k4abt_tracker_t()
    tracker_config = k4a.K4ABT_TRACKER_CONFIG_DEFAULT
    #tracker_config.processing_mode = k4a.K4ABT_TRACKER_PROCESSING_MODE_CPU
    VERIFY(k4a.k4abt_tracker_create(ctypes.byref(sensor_calibration), tracker_config, ctypes.byref(tracker)), "Body tracker initialization failed!")

    cap = cv2.VideoCapture(0)

    isWaiting = False
    isRecording = False
    recordSkeleton = False
    start_time = 0
    now_time = 0
    prev_time = 0
    record_count = 0
    record_period = 500 # 2sec.7
    record_times = 100 # 10times

    while True:
        sensor_capture = k4a.k4a_capture_t()
        get_capture_result = k4a.k4a_device_get_capture(device, ctypes.byref(sensor_capture), k4a.K4A_WAIT_INFINITE)
        ret, frame = cap.read()

        prev_time = now_time
```

```
            now_time = time.time_ns() // 1000000    # millisecond
            if isWaiting:
                if (now_time-start_time) < 5000:
                    # A red circle blinks for 5 seconds to wait for user is posed.
                    if (now_time-start_time) % 1000 < 500:
                        cv2.circle(frame, (50,50), 20, (0, 0, 255), thickness=-1)
                else:
                    # After 5 seconds, coordinates of a skeleton are recorded every 2 seconds.
                    # Resetting start_time
                    start_time = now_time
                    isWaiting = False
                    isRecording = True
                    record_count = 0
            elif isRecording:
                # A red circle appears while the coordinates of the skeleton are recorded.
                cv2.circle(frame, (50,50), 20, (0, 0, 255), thickness=-1)
                if record_count < record_times:
                    if (prev_time-start_time) // record_period != (now_time-start_time) //
record_period:
                        record_count += 1
                        recordSkeleton = True
                    if record_count >= 1:
                        cv2.putText(frame,     '{0}'.format(record_count),     (100,     70),
cv2.FONT_HERSHEY_SIMPLEX, 2, (0, 0, 255), 2)
                else:
                    isRecording = False

            if get_capture_result == k4a.K4A_WAIT_RESULT_SUCCEEDED:
                #frame_count += 1

                #print("Start processing frame {}".format(frame_count))

                queue_capture_result      =      k4a.k4abt_tracker_enqueue_capture(tracker,
sensor_capture, k4a.K4A_WAIT_INFINITE)

                k4a.k4a_capture_release(sensor_capture)
```

```
            if queue_capture_result == k4a.K4A_WAIT_RESULT_TIMEOUT:
                # It should never hit timeout when K4A_WAIT_INFINITE is set.
                print("Error! Add capture to tracker process queue timeout!")
                break
            elif queue_capture_result == k4a.K4A_WAIT_RESULT_FAILED:
                print("Error! Add capture to tracker process queue failed!")
                break

            body_frame = k4a.k4abt_frame_t()
            pop_frame_result = k4a.k4abt_tracker_pop_result(tracker,
ctypes.byref(body_frame), k4a.K4A_WAIT_INFINITE)
            if pop_frame_result == k4a.K4A_WAIT_RESULT_SUCCEEDED:
                num_bodies = k4a.k4abt_frame_get_num_bodies(body_frame)
                #print("{} bodies are detected!".format(num_bodies))
                for i in range(num_bodies):
                    body = k4a.k4abt_body_t()
                    VERIFY(k4a.k4abt_frame_get_body_skeleton(body_frame, i,
ctypes.byref(body.skeleton)), "Get body from body frame failed!")
                    body.id = k4a.k4abt_frame_get_body_id(body_frame, i)

                    #print_body_information(body)

                    if i==0:
                        if recordSkeleton:
                            recordSkeleton = False
                            # 32 joints, 3 coordinates and confidence
                            data_array = np.empty((32, 4), float)
                            for joint in range(32):
                                data_array[joint,0] =
body.skeleton.joints[joint].position.xyz.x
                                data_array[joint,1] =
body.skeleton.joints[joint].position.xyz.y
                                data_array[joint,2] =
body.skeleton.joints[joint].position.xyz.z
                                data_array[joint,3] =
```

```
body.skeleton.joints[joint].confidence_level
                                    date_now = datetime.datetime.now()
                                    filename                                                    =
'{0}{1}.csv'.format(date_now.strftime('skeleton_%Y%m%d_%H%M_%S_'),
date_now.strftime('%f')[0:3])
                                    np.savetxt(filename, data_array, delimiter=',', fmt='%.2f')

                        body_index_map = k4a.k4abt_frame_get_body_index_map(body_frame)
                        if body_index_map:
                            #print_body_index_map_middle_line(body_index_map)
                            k4a.k4a_image_release(body_index_map)
                        else:
                            print("Error: Fail to generate bodyindex map!")

                        k4a.k4abt_frame_release(body_frame)
                    elif pop_frame_result == k4a.K4A_WAIT_RESULT_TIMEOUT:
                        # It should never hit timeout when K4A_WAIT_INFINITE is set.
                        print("Error! Pop body frame result timeout!")
                        break
                    else:
                        print("Pop body frame result failed!")
                        break
                elif get_capture_result == k4a.K4A_WAIT_RESULT_TIMEOUT:
                    # It should never hit timeout when K4A_WAIT_INFINITE is set.
                    print("Error! Get depth frame time out!")
                    break
                else:
                    print("Get depth capture returned error: {}".format(get_capture_result))

                cv2.imshow('frame', frame)

                key = cv2.waitKey(1)
                if key & 0xFF == ord('q'):
                    break

                if key & 0xFF == ord('r') and not isWaiting and not isRecording:
```

```
            isWaiting = True
            start_time = time.time_ns() // 1000000 # millisecond

    print("Finished body tracking processing!")

    k4a.k4abt_tracker_shutdown(tracker)
    k4a.k4abt_tracker_destroy(tracker)
    k4a.k4a_device_stop_cameras(device)
    k4a.k4a_device_close(device)
```

ジェスチャー認識用学習プログラム

```
# -*- coding: utf-8 -*-
import sys
import os
import glob
import math
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
import matplotlib.pyplot as plt

# "conda install tensorflow=1.15.0    keras=2.3.1" is required

if len(sys.argv)!=2:
    print('Usage: python learn_skeletons.py savefile.h5')
    sys.exit()
savefile = sys.argv[1]

num_joints = 32
num_features = num_joints * 3 # 32 joints * 3 dimensions

# X:input vectors, y: targets
X = np.empty((0, num_features), float)
y = np.array([], int)

# checking number of classes (<=100 classes)
num_classes = 0
for i in range(100):
    # checking the existence of folders (0, 1, 2,...)
    if not os.path.exists('{0}'.format(i)):
        break
    num_classes += 1

# reading taining vectors
for gesture_class in range(num_classes):
```

```
    files = glob.glob('{0}/*.csv'.format(gesture_class))
    for file in files:
        print('Reading {0}...'.format(file))
        data = np.loadtxt(file, delimiter=',', dtype='float')
        data2 = np.empty(data.shape, float)
        # setting PELVIS as origin
        x0 = data[0, 0]
        y0 = data[0, 1]
        z0 = data[0, 2]
        for joint in range(num_joints):
            data[joint, 0] -= x0
            data[joint, 1] -= y0
            data[joint, 2] -= z0
        # normalizing |PELVIS - NECK| as 1
        length = math.sqrt(data[3, 0]**2 +   data[3, 1]**2 + data[3, 2]**2 )
        for joint in range(num_joints):
            data[joint, 0] /= length
            data[joint, 1] /= length
            data[joint, 2] /= length
        # Rotating skeleton around the vertical axis
        for angle in range(0, 360, 5):
            cosval = math.cos(math.radians(angle))
            sinval = math.sin(math.radians(angle))
            for joint in range(num_joints):
                data2[joint, 0] = cosval*data[joint, 0] - sinval*data[joint, 2]
                data2[joint, 2] = sinval*data[joint, 0] + cosval*data[joint, 2]
                data2[joint, 1] = data[joint, 1]
            # Appending learning vector
            X = np.append(X, np.array([data2[:,0:3].flatten()]), axis=0)
            y = np.append(y, gesture_class)

(num_samples, tmp) = X.shape

y_keras = keras.utils.to_categorical(y, num_classes)

model = Sequential()
```

```
model.add(Dense(units=200, activation='relu', input_shape=(num_features,)))
model.add(Dropout(0.1))
model.add(Dense(units=100, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(units=num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

#history = model.fit(X, y_keras, epochs=100, validation_split=0.1, batch_size=200,
verbose=2)
history = model.fit(X, y_keras, epochs=300, validation_split=0, batch_size=200,
verbose=2)

result = model.predict_classes(X, verbose=0)

total = len(X)
success = sum(result==y)

print('Correct rate')
print(100.0*success/total)

model.save(savefile)

plt.xlabel('time step')
plt.ylabel('loss')

#plt.ylim(0, max(np.r_[history.history['val_loss'], history.history['loss']]))
plt.ylim(0, max(history.history['loss']))
#val_loss, = plt.plot(history.history['val_loss'], c='#56B4E9')
loss, = plt.plot(history.history['loss'], c='#E69F00')
#plt.legend([loss, val_loss], ['loss', 'val_loss'])
plt.legend([loss], ['loss'])
plt.show()
```

ジェスチャー認識用プログラム

```
"""
This is a python port of the simple-sample project from.
    https://github.com/microsoft/Azure-Kinect-
Samples/blob/bf2f8cf95d969dcc7842c4c450052fe5a943c756/body-tracking-
samples/simple_sample/main.c
"""

import traceback
import sys
import ctypes
import os
import cv2
import keras
import numpy as np
import math

import openpyxl
import matplotlib.pyplot as plt
import struct
import time #added
import socket
import json
#import pickle
zsum = 0
count = 0
data_count=1
disMax = 0
disMin = 10000

M_SIZE = 4056
ros_serv_address = ('192.168.1.8', 50002)
serv_address = ('127.0.0.1', 50003)

count = 0
```

```
pose =5
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# Add .. to the import path
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

import k4a #Azure kinect

def VERIFY(result, error): #エラー
    if result != k4a.K4A_RESULT_SUCCEEDED:
        print(error)
        traceback.print_stack()
        sys.exit(1)

def print_body_information(body): #body 情報　body ID と距離と座標
    print("Body ID: {}".format(body.id))
    for i in range(k4a.K4ABT_JOINT_COUNT):
        position = body.skeleton.joints[i].position
        orientation = body.skeleton.joints[i].orientation
        confidence_level = body.skeleton.joints[i].confidence_level
        print("Joint[{}]: Position[mm] ( {}, {}, {} ); Orientation ( {}, {}, {}, {}); Confidence
Level ({})".format(
            i, position.v[0], position.varray([1]), position.varray([2]), orientation.v[0],
orientation.varray([1]), orientation.varray([2]), orientation.varray([3]), confidence_level
        ))

def pid_dis(dis_Actual,cen_Actual):
    dis_kp=0.001
    cen_kp=0.001
    dis_b1=(dis_Actual-1700)
    dis_p=dis_kp*dis_b1
    cen_b1=(cen_Actual-0)
    cen_p=cen_kp*cen_b1

    return dis_p+cen_p

def print_body_index_map_middle_line(body_index_map):
```

```
    print("print_body_index_map_middle_line not implemented")
    """
    uint8_t* body_index_map_buffer = k4a_image_get_buffer(body_index_map);

    // Given body_index_map pixel typeq should be uint8, the stride_byte should be the same as width
    // TODO: Since there is no API to query the byte-per-pixel information, we have to compare the width and stride to
    // know the information. We should replace this assert with proper byte-per-pixel query once the API is provided by
    // K4A SDK.
    assert(k4a_image_get_stride_bytes(body_index_map) == k4a_image_get_width_pixels(body_index_map));

    int middle_line_num = k4a_image_get_height_pixels(body_index_map) / 2;
    body_index_map_buffer = body_index_map_buffer + middle_line_num * k4a_image_get_width_pixels(body_index_map);

    printf("BodyIndexMap at Line %d:¥n", middle_line_num);
    for (int i = 0; i < k4a_image_get_width_pixels(body_index_map); i++)
    {
        printf("%u, ", *body_index_map_buffer);
        body_index_map_buffer++;
    }
    printf("¥n");
    """

if __name__ == "__main__":

    if len(sys.argv)==1:
        print('Usage: python recognize_skeleton.py savedfile.h5')
        sys.exit()
    savedfile = sys.argv[1]
    model = keras.models.load_model(savedfile)

    num_joints = 32
```

```
    num_features = num_joints * 3 # 32 joints * 3 dimensions

    data = np.empty((num_joints, 3), float)

    device_config = k4a.K4A_DEVICE_CONFIG_INIT_DISABLE_ALL
    device_config.depth_mode = k4a.K4A_DEPTH_MODE_NFOV_UNBINNED#深度モード
    #device_config.colorresolution_mode = k4a.K4A_COLOR_RESOLUTION_OFF



    device = k4a.k4a_device_t()
    VERIFY(k4a.k4a_device_open(0, ctypes.byref(device)), "Open K4A Device failed!")
    VERIFY(k4a.k4a_device_start_cameras(device, ctypes.byref(device_config)), "Start K4A cameras failed!")

    sensor_calibration = k4a.k4a_calibration_t()
    VERIFY(k4a.k4a_device_get_calibration(device, device_config.depth_mode, k4a.K4A_COLOR_RESOLUTION_OFF, ctypes.byref(sensor_calibration)), "Get depth camera calibration failed!")

    tracker = k4a.k4abt_tracker_t()
    tracker_config = k4a.K4ABT_TRACKER_CONFIG_DEFAULT
    #tracker_config.processing_mode = k4a.K4ABT_TRACKER_PROCESSING_MODE_CPU
    VERIFY(k4a.k4abt_tracker_create(ctypes.byref(sensor_calibration), tracker_config, ctypes.byref(tracker)), "Body tracker initialization failed!")

    cap = cv2.VideoCapture(0)

    while True:

        sensor_capture = k4a.k4a_capture_t()
        get_capture_result = k4a.k4a_device_get_capture(device, ctypes.byref(sensor_capture), k4a.K4A_WAIT_INFINITE)
        ret, frame = cap.read()
        frame2 = cv2.flip(frame,1)#added
```

```
        if get_capture_result == k4a.K4A_WAIT_RESULT_SUCCEEDED:


            #print("Start processing frame {}".format(frame_count))

            queue_capture_result = k4a.k4abt_tracker_enqueue_capture(tracker,
sensor_capture, k4a.K4A_WAIT_INFINITE)

            k4a.k4a_capture_release(sensor_capture)

            if queue_capture_result == k4a.K4A_WAIT_RESULT_TIMEOUT:
                # It should never hit timeout when K4A_WAIT_INFINITE is set.
                print("Error! Add capture to tracker process queue timeout!")
                break
            elif queue_capture_result == k4a.K4A_WAIT_RESULT_FAILED:
                print("Error! Add capture to tracker process queue failed!")
                break

            body_frame = k4a.k4abt_frame_t()
            pop_frame_result = k4a.k4abt_tracker_pop_result(tracker,
ctypes.byref(body_frame), k4a.K4A_WAIT_INFINITE)
            if pop_frame_result == k4a.K4A_WAIT_RESULT_SUCCEEDED:
                num_bodies = k4a.k4abt_frame_get_num_bodies(body_frame)
                #print("{} bodies are detected!".format(num_bodies))

                for i in range(num_bodies):
                    body = k4a.k4abt_body_t()
                    VERIFY(k4a.k4abt_frame_get_body_skeleton(body_frame, i,
ctypes.byref(body.skeleton)), "Get body from body frame failed!")
                    body.id = k4a.k4abt_frame_get_body_id(body_frame, i)

                    #print_body_information(body)
                if num_bodies == 0:
                        print("停止中")
                        pose = 5
```

```
                    pose_byte = bytes(pose)

                    send_len = sock.sendto(pose_byte, serv_address)

                elif num_bodies==1:
                    for joint in range(num_joints):
                        data[joint, 0] = body.skeleton.joints[joint].position.xyz.x
                        data[joint, 1] = body.skeleton.joints[joint].position.xyz.y
                        data[joint, 2] = body.skeleton.joints[joint].position.xyz.z

                    x0 = data[0, 0]
                    y0 = data[0, 1]
                    z0 = data[0, 2]

                    for joint in range(num_joints):
                        data[joint, 0] -= x0
                        data[joint, 1] -= y0
                        data[joint, 2] -= z0

                    # normalizing |PELVIS - NECK| as 1
                    length = math.sqrt(data[3, 0]**2 +  data[3, 1]**2 + data[3,
2]**2 )
                    for joint in range(num_joints):
                        data[joint, 0] /= length
                        data[joint, 1] /= length
                        data[joint, 2] /= length

                    X = np.array([data.flatten()])

                    result = model.predict_classes(X, verbose=0)+1

                    count+=1

                    pose = (result[0])

                    pose_byte = bytes(result)
```

```
                        send_len = sock.sendto(pose_byte, serv_address)

                        ds = struct.pack('>d',pid_dis(z0, x0))
                        pose2 = struct.pack('>d',pose)

                        send_len = sock.sendto(ds, ros_serv_address)#ポーズのソケッ
ト to ROS
                        send_len = sock.sendto(pose2, ros_serv_address)

                        print(count,"距離",z0)

                else:
                        print("複数人います")
                        pose = 6
                        pose_byte = bytes(pose)
                        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                        send_len = sock.sendto(pose_byte, serv_address)


                body_index_map = k4a.k4abt_frame_get_body_index_map(body_frame)
                if body_index_map:
                    #print_body_index_map_middle_line(body_index_map)
                    k4a.k4a_image_release(body_index_map)
                else:
                    print("Error: Fail to generate bodyindex map!")

                k4a.k4abt_frame_release(body_frame)
            elif pop_frame_result == k4a.K4A_WAIT_RESULT_TIMEOUT:
                # It should never hit timeout when K4A_WAIT_INFINITE is set.
                print("Error! Pop body frame result timeout!")
                break
            else:
                print("Pop body frame result failed!")
                break
        elif get_capture_result == k4a.K4A_WAIT_RESULT_TIMEOUT:
```

```
            # It should never hit timeout when K4A_WAIT_INFINITE is set.
            print("Error! Get depth frame time out!")
            break
        else:
            print("Get depth capture returned error: {}".format(get_capture_result))

        #cv2.imshow('frame', frame2)
        cv2.imshow('frame', frame)

        key = cv2.waitKey(1)
        if key & 0xFF == ord('q'):

            break

    print("Finished body tracking processing!")

    k4a.k4abt_tracker_shutdown(tracker)
    k4a.k4abt_tracker_destroy(tracker)
    k4a.k4a_device_stop_cameras(device)
    k4a.k4a_device_close(device
```

人物認識によりジェスチャーで操作する移動ロボットの開発

指導教員　金丸　隆志　教授
S5-18053 堀越 優来　S5-18058 丸山 拓己
S5-18060 宮崎 生望　S5-18064 目黒 新大
S5-18069 吉武 敏

1.　緒言

近年の新型コロナウイルス感染拡大は，人や物との接触に関する意識に大きな変化を与えた．したがって非接触による機器の操作需要が高まっている．我々は，その中でも店員や服への接触などの問題がある試着に着目した．この問題を解決するためにジェスチャーによって操作可能な移動型仮想試着システムの開発を行う．その中で今回私たちはジェスチャーで操作する移動ロボットに注力して開発を行う．ジェスチャーを用いた移動ロボットの設計において，Azure Kinect を人物認識センサとして用いる．

2.　ロボットの製作

Azure Kinect を設置する高さを 920mm と設定して TurtleBot3 をもとにフレームを設計した．ROS の機能を使用してロボットを動作させる.製作したロボットを図 1 に示す.

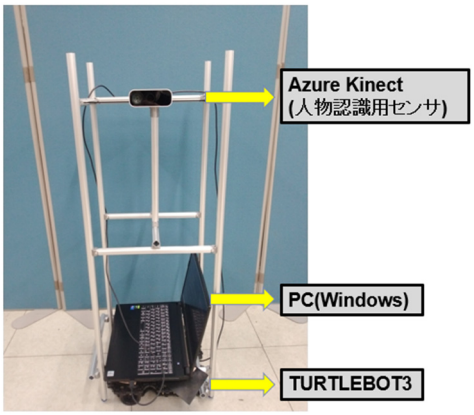


図 1　移動ロボット

3.　Azure Kinect を用いた人物認識

ジェスチャーでロボットを操作するために図 2 のような「直立」,「右手前」,「左手前」,「腰に両手」を選定した．そして，ロボットの移動コマンドに選定した 4 つのジェスチャーを割り当てた.

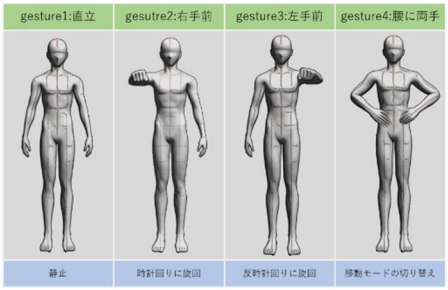


図 2　ジェスチャーとロボットの移動コマンドの対応

4.　認識率実験

認識率実験では被験者とセンサの距離を 1700mm，センサの高さを 900mm に設定し，45 度ずつ計 8 か所に分け，その地点での認識率を求めた．ジェスチャー登録者と非登録者の角度ごとの 4 つのジェスチャーの認識率の平均は概ね 100%となり，さらに登録者と非登録者ではほぼ同じ認識率を出していたため，認識精度に問題はないといえる．

5.　移動ロボットの制御

移動ロボットを適切な動きにするため，図 3 と図 4 のように被験者を映像中心に収める制御と一定の距離を保つ制御を実装した．その結果、常に被験者を映像範囲に収めることが可能になった.

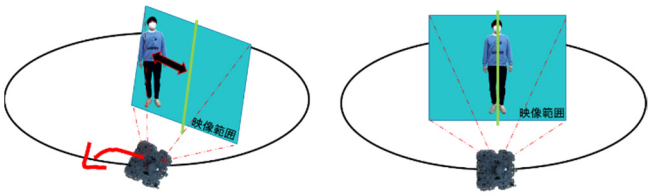


図 3　被験者を映像中心に収める制御

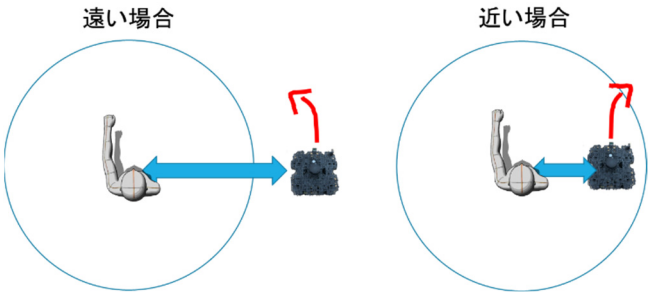


図 4　一定の距離を保つ制御

6.　結論

1. 様々な角度からのジェスチャー認識に成功した.
2. ジェスチャーでのロボット操作を可能にした.
3. ロボットと人物の距離を一定に保つ制御プログラムを開発した.

以上より,「ジェスチャーで操作する移動ロボット」を実現できたといえる.

＜参考文献＞
「コロナショック前後のモノとの接触」に関する意識調査, 2020