

2025年度 卒業論文

画像処理およびセンサを用いた  
遠隔在庫把握システムの構築

Development of a Remote Inventory Monitoring System  
Using Image Processing and Sensors

成蹊大学 理工学部 理工学科 機械システム専攻  
流体力学研究室

S226109 前澤 栄飛  
S226003 秋元 大生

指導教授：小川 隆申

提出日 2026年 1月 29日

# 目次

<b>第1章</b>	<b>序論</b>	<b>1</b>
1.1	研究背景	1
1.2	研究目的	3
<b>第2章</b>	<b>画像処理による在庫検出</b>	<b>4</b>
2.1	画像取得に用いた機器	4
2.2	OpenCV	6
2.3	輪郭抽出	11
2.4	類似商品の識別	15
2.5	検出結果と考察	19
<b>第3章</b>	<b>超音波センサによる在庫の把握</b>	<b>23</b>
3.1	超音波センサによる在庫把握の方法	23
3.2	研究に使用した装置	24
3.3	超音波センサの配置	24
3.4	超音波センサの距離測定に使用するプログラム	25
3.5	考察	26
<b>第4章</b>	<b>デプスカメラによる在庫把握</b>	<b>28</b>
4.1	デプスカメラによる在庫把握の方法	28
4.2	デプスカメラの配置	29
4.3	デプスカメラでの距離測定に使用するプログラム	29
4.4	考察	30
<b>第5章</b>	<b>LINE Bot を用いた遠隔の在庫把握</b>	<b>31</b>
5.1	LINE Bot	31
5.2	遠隔在庫把握システムの流れ	36
5.3	公式アカウント上のメッセージ送信手法	40
5.4	システムの動作結果と考察, 課題	45
<b>第6章</b>	<b>結論</b>	<b>48</b>
<b>参考文献</b>		<b>49</b>
<b>謝辞</b>		<b>52</b>
<b>付録A</b>	<b>プログラム</b>	<b>1</b>

A.1	画像処理のソースコード	1
A.2	超音波センサのソースコード	5
A.3	LINE Bot のソースコード	6
<b>付録B</b>	<b>原稿非掲載データ</b>	<b>10</b>
B.1	実験結果	10

# 第1章 序論

## 1.1 研究背景

近年、様々な技術の発達に伴い、人々の生活はより快適になっている。その中でも、IT (Information Technology) 技術は、コンピューターおよびネットワークを利用して情報の処理や通信を行う技術の総称であり、社会的重要性が高まっている。例えば、画像データに対し情報の変換や特徴量の抽出などの処理を行い、目的に応じた分析および判定を可能にする画像処理<sup>[2]</sup>、感知器や測定器などを用いて対象の定量的な情報を取得するセンシング<sup>[1]</sup>が挙げられる。これらの技術は、農業、製造業、医療、物流など幅広い分野で活用されており、作業の効率化や生産性の向上に寄与している。

画像処理およびセンシングの活用事例として、図1.1に示す成蹊大学11号館1階に設置された無人店舗が挙げられる。この施設では、複数のAI (Artificial Intelligence) カメラおよびセンサを用いて利用者の動作検知や商品の識別を行うことで、キャッシュレス決済やデータの収集、リアルタイム処理を実現している。これにより、利用者の円滑な購買行動を促進し、効率的な店舗運営が可能になる<sup>[3]</sup>。その一方、無人店舗の運営においていくつかの課題が存在する。主要な課題の1つとして、AIカメラおよびセンサによって収集されたデータを利用する独自のシステム構築が困難である点が挙げられる。これらのデータは無人店舗を管理している企業が保有しており、成蹊大学側に提供されていないため、容易に入手することができない。また別の主要な課題として、地理的条件に加え、訪問前に在庫状況を把握できることにより、無人店舗が利用選択肢として選ばれづらい点が挙げられる。図1.2に示す成蹊大学キャンパスマップから分かる通り無人店舗はキャンパスの端に近い場所に設置されており、正門からの距離も大きく離れている。そのため、店舗を訪れたにもかかわらず、目当ての商品が売り切れや欠品によって入手できない場合、利用者が時間を浪費してしまう可能性がある。そこで独自にデータを収集、利用し、在庫状況を遠隔から手軽に確認するシステムを構築することで、利用可能なデータの入手方法の確立および無人店舗の利便性向上による利用者の増加が期待できる。



図 1.1: 大学内の無人店舗

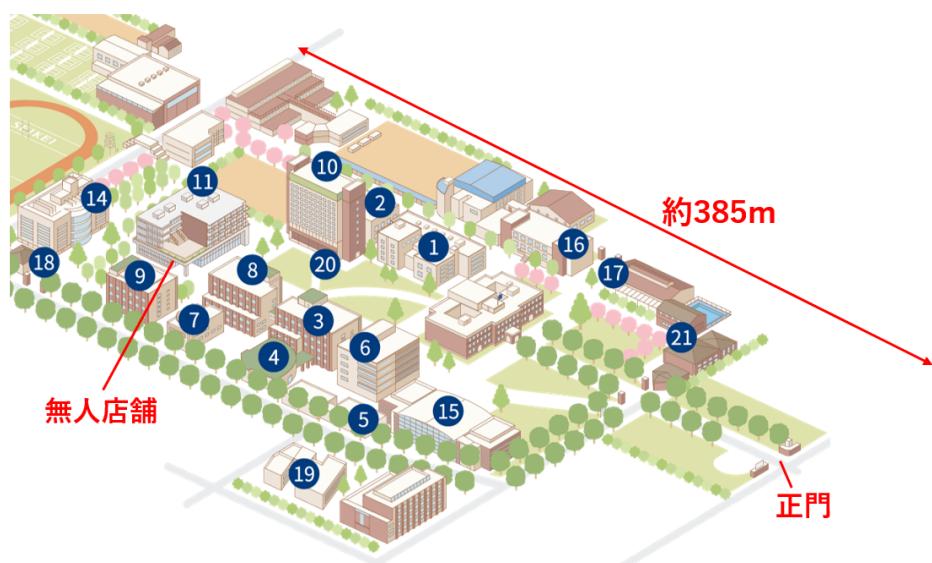


図 1.2: 成蹊大学キャンパスマップ

## 1.2 研究目的

本研究では、成蹊大学キャンパス内に設置された無人店舗を対象とし、OpenCV を用いた画像処理、超音波センサを用いたセンシング、デプスカメラを用いたセンシングという 3 つの手法から在庫状況に関するデータの収集を行う。さらに、各手法の利点と欠点を整理し、データ収集の精度および設置の容易性の観点から実現性の最も高い手法を検討する。加えて、収集したデータを LINE Bot を用いて利用者に共有することで、遠隔から在庫状況を把握可能なシステムの構築を目的とする。

# 第2章 画像処理による在庫検出

無人店舗の在庫状況に関するデータの収集を実現するため、複数の技術的アプローチについて検討を行い、最適な在庫検出手法を明確にする。本章では、画像処理技術を用いた在庫検出について、OpenCV を用いた検出方法、輪郭抽出、座標による商品識別、検出した結果と考察を説明する。

## 2.1 画像取得に用いた機器

画像処理を行うにあたり、無人店舗の商品棚の画像を取得するため、スマートフォンのカメラ機能を利用した。本章では、画像処理技術を用いた在庫検出の実現性について検討することを主な目的としているため、コスト削減や画像取得の容易さの観点からこの選択をした。実際の画像取得に用いたスマートフォンを図 2.1 に、取得した商品棚の画像を図 2.2、図 2.3 に示す。



図 2.1: 画像取得用スマートフォン<sup>[4]</sup>



図2.2: おにぎりを中心とする商品棚



図2.3: 様々な色の商品が並ぶ棚

画像を一定の品質で取得するため、カメラは等倍かつ品棚から約3mの位置で撮影を行う。図2.4に示す一部結果を可視化した画像において、商品の周りに検出を表現する矩形（バウンディングボックス）が表示されていることから、本章で用いたスマートフォンのカメラ機能は、画像処理による商品検出に十分な性能を満たしていると判断した。スマートフォンのカメラ性能を表2.1に示す。



図2.4: 一部商品棚の検出結果

表2.1: スマートフォンのカメラ性能

使用機種	Sony Xperia 10 VI 5G
使用カメラ	メインカメラ
画像解像度	1920×1080
総画素数	約207万画素

表 2.1 に示した使用するカメラの画像解像度が  $1920 \times 1080$  であることから、本章は、解像度  $1920 \times 1080$  と同様、もしくはそれ以上の性能を有するカメラを用いたときの画像処理技術による在庫検出について検討する。

## 2.2 OpenCV

OpenCV (Open Source Computer Vision Library) は、画像および動画に関する処理機能・検出機能をまとめたオープンソースのライブラリである<sup>[5]</sup>。Windows や Linux, iOS, Android などさまざまな OS に対応しており、Raspberry Pi などの端末上で利用することもできる。さらに、豊富な画像処理機能を搭載しており、高度な画像処理を比較的容易に実装できるという特徴を有しているため、本研究の目的達成に重要な、様々な種類の商品の高精度検出およびリアルタイム性のあるデータ収集に最適だと考え、今回 OpenCV を使用した。

### 2.2.1 グレースケール変換

OpenCV の機能にグレースケール変換がある。これは、色の情報を省き、明るさの度合いという情報のみで表現するための画像処理であり<sup>[6]</sup>、ソースコード A.1 に示すグレースケール変換前後の画像比較から分かる通り、グレースケール変換によって、画像内の明暗を明確にすることができる。

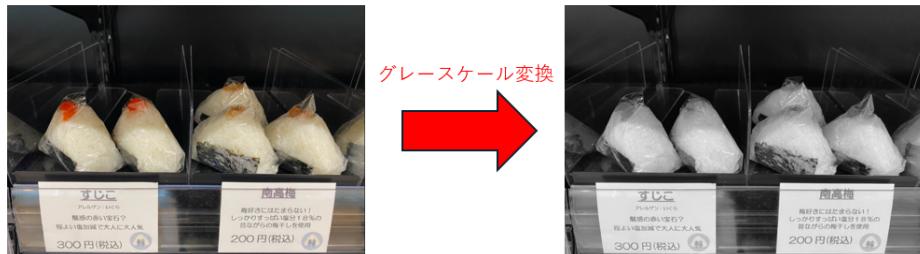


図 2.5: グレースケール変換を適用した画像

通常、処理を行う前のカラー画像（以下 RGB 画像とする）は、R (Red), G (Green), B (Blue) の 3 つの値（以下 RGB 値とする）で表現され、それぞれ 0～255 の数値をとる。一方、グレースケール変換を行った画像（以下グレースケール画像とする）は、輝度という 1 つの値で表現され<sup>[7]</sup>、0～255 の数値をとる。図 2.6 に示す輝度の値による明るさの違いから分かる通り、値が小さいと暗く、大きいと明るく表現される。



図2.6: 輝度の値による明るさ

グレースケール変換の仕組みについて説明する。変換方法は複数存在するが、OpenCVでは、加重平均法が用いられている<sup>[8]</sup>。グレースケール画像を表現する輝度という値は、RGB画像のRGB値から計算された値であり、加重平均法はR, G, Bの値それぞれに重み付けをして輝度を計算する方法である。OpenCVにおいて輝度は

$$\text{輝度} = 0.2989 \times R + 0.5870 \times G + 0.1140 \times B \quad (2.1)$$

である。このRGBそれぞれに対する重みは、緑の光が最も検知しやすく、青の光は比較的検知しにくいという人間の視覚特性をもとに設定されている<sup>[9]</sup>。

グレースケール変換の活用が有効な場面を検討する。RGBはデジタル機器同様、人間の視覚が光の総量に強く依存する性質を利用して構造化されており<sup>[10]</sup>、光の三原色として表現される。そのため、RGB値をすべて255にして表現される色は白であり、RGB値をすべて0にして表現される色は黒である。(2.1)より、RGB値をすべて255にしたとき、輝度の値は最大である255をとり、RGB値をすべて0にしたとき、輝度の値は最小である0をとる。したがって、画像にグレースケール変換を適用した際、白に近い色を持つ物体は明るく表現され、黒に近い色を持つ物体は暗く表現される。

本研究において、図2.7に示す実際の商品棚の一部から分かる通り、研究対象である無人店舗の一部の商品棚は、おにぎりのような白に近い色が特徴的な商品を中心に扱っている。また、図2.8に示す実際の商品棚の全体から分かる通り、無人店舗の商品棚全体が黒いデザインとなっている。したがって、おにぎりを中心に扱う商品棚に関しては、グレースケール変換を活用することで、商品と背景の差が明確になり、精度の高い検出が期待できる。加えて、グレースケール画像が輝度という1つの値で表現できる特徴から、情報量の削減による処理の高速化が期待できる。



図2.7: 白に近い色を持つ商品が多い棚



図2.8: 無人店舗の商品棚全体

## 2.2.2 HSV 変換

OpenCV の機能に HSV 変換がある。これは、RGB 画像を色相 (Hue), 彩度 (Saturation), 明度 (Value) の 3 つの要素で表現された画像（以下 HSV 画像とする）に変換する機能である<sup>[11]</sup>。

色相 (Hue) について説明する。これは、具体的な色の種類を表す要素で、図 2.9 に示す実際のカラー ホイールから分かる通り、0～359 の値を用いて様々な色を表現する。しかし、OpenCV では効率的な処理を目的として 8 bit 画像を主な対象としているため、扱う数値を 0～255 の範囲で表現することが望ましい。そのため、図 2.10 に示す OpenCV のカラー ホイールから分かる通り、OpenCV は色相の値を 1/2 にスケーリングし、0～179 の値を用いて色を表現する<sup>[12]</sup>。

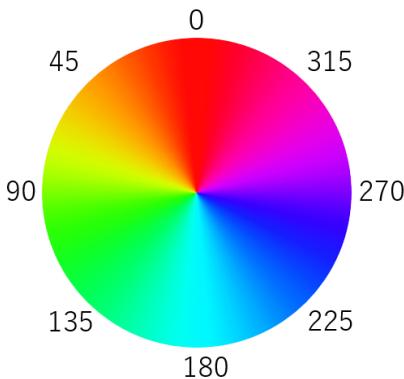


図2.9: 一般的な色相範囲

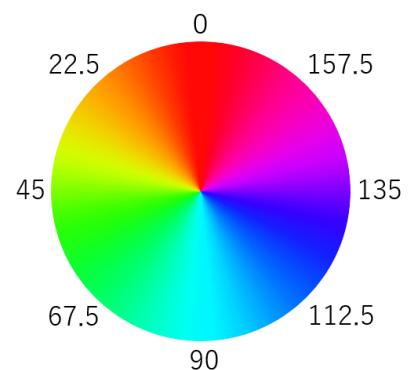


図2.10: OpenCV の色相範囲

彩度 (Saturation) について説明する。これは、色の鮮やかさや濃さを表す要素で<sup>[11]</sup>、色相と異なり、数値に明確な範囲が存在しないため、OpenCV は 0～255 の範囲に値をスケーリングして

表現する。図2.11に示す数値ごとの彩度表現から分かる通り、彩度の値が255をとると色は最も鮮やかになり、0の値をとると最も鈍くなる。

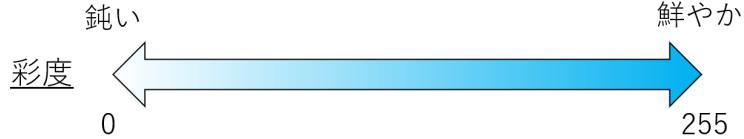


図2.11: 彩度の値による変化

明度 (Value) について説明する。これは、色の明るさを表す要素で<sup>[11]</sup>、彩度と同様に、数値に明確な範囲が存在しないため、OpenCVは0~255の範囲に値をスケーリングして表現する。図2.12に示す数値ごとの明度表現から分かる通り、明度の値が255をとると色は最も明るくなり、0の値をとると最も暗くなる。



図2.12: 明度の値による変化

HSV変換の仕組みについて説明する。グレースケール変換同様、HSV変換においてもRGB値を基に計算されている。RGB値をそれぞれ  $R$ ,  $G$ ,  $B$  とし、これらを0~1の範囲で表現したものをそれぞれ  $R'$ ,  $G'$ ,  $B'$  ( $R' = \frac{R}{255}$ ,  $G' = \frac{G}{255}$ ,  $B' = \frac{B}{255}$ ) とする。また、RGBの最大値 ( $\max(R, G, B)$ ) を  $C_{max}$ 、最小値 ( $\min(R, G, B)$ ) を  $C_{min}$  とし、その差 ( $C_{max} - C_{min}$ ) を  $\Delta$  とする。色相 ( $H$ ) は、 $C_{min} = R$  のとき

$$H = \frac{1}{2}(60 \times (\frac{B' - G'}{\Delta}) + 180) \quad (2.2)$$

であり、 $C_{min} = G$  のとき

$$H = \frac{1}{2}(60 \times (\frac{R' - B'}{\Delta}) + 300) \quad (2.3)$$

であり、 $C_{min} = B$  のとき

$$H = \frac{1}{2}(60 \times (\frac{G' - R'}{\Delta}) + 60) \quad (2.4)$$

である<sup>[13]</sup>. また, OpenCVでは $\Delta=0$ のとき,  $H=0$ として定義される.

彩度 ( $S$ ) は

$$S = \frac{\Delta}{C_{max}} \quad (2.5)$$

である. また, OpenCVでは $C_{max}=0$ のとき,  $S=0$ として定義される.

明度 ( $V$ ) は

$$V = C_{max} \quad (2.6)$$

である[13].

HSVは, 色を鮮やかさや明るさという直感的に分かりやすい要素で表現するため, 原色の組み合わせで表現するRGBに比べ, 各要素を変動させた場合の色の変化がイメージしやすく, 細かな色の調整が簡単にできる.

本研究において, 図2.3に示した実際の商品棚の一部から分かる通り, 無人店舗の一部の商品棚は, 様々な色の商品を扱っている. そのため, 輝度の値が商品によって異なり, 図2.13に示す様々な色の商品が並ぶ棚のグレースケール画像を用いた検出結果から分かる通り, 検出精度が低くなる. そこでHSV画像を活用することで, 安定した精度の検出が期待できる.



図2.13: 様々な色の商品が並ぶ棚のグレースケール検出

## 2.3 輪郭抽出

商品の位置を特定するにあたり、グレースケール画像およびHSV画像から商品の輪郭抽出を行う。具体的には、商品と背景を判別するための閾値を設定し、これを満たす画素集合を輪郭として抽出する。閾値とは、判断の境目となる値のこと<sup>[14]</sup>で、OpenCVでは、利用者が任意に設定できる。

### 2.3.1 グレースケール画像の輪郭抽出

グレースケール画像に適用する閾値を考える。2.2.1節で述べたように、グレースケール画像では、輝度という0～255の範囲をとる1つの値で表現されるため、閾値はこの値を用いて適用する。

図2.14に示す照明が商品棚へ及ぼす影響から分かる通り、無人店舗の商品棚は4段ある棚の段数のうち、最上段のみ照明が強く当たる構造になっている。そのため、輝度の値で表現するグレースケール画像では、全体に同一の閾値を設定すると、最上段とその他の段で検出精度に差が生じる。この問題に対処するため、画像内座標を利用し、最上段とそれ以外の範囲を切り分け、最上段の閾値を180～255、それ以外の範囲を100～255にそれぞれ設定する。この閾値は、図2.7に示したグレースケール変換の適用する商品棚において、様々な閾値を試した結果、最も検出精度が高くなると判断した値である。照明の当たり方を考慮した閾値設定のイメージを図2.15に示す。



図2.14: 照明の商品棚への影響

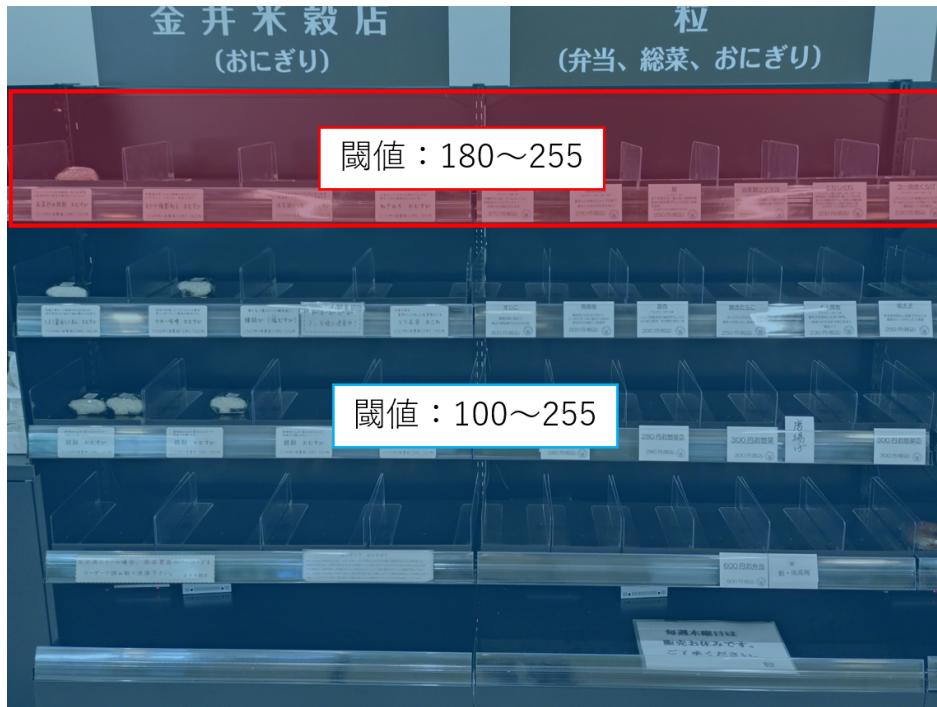


図 2.15: グレースケール画像の閾値設定のイメージ

### 2.3.2 HSV 画像の輪郭抽出

HSV 画像に適用する閾値を考える。2.2.2 節で述べたように、HSV 画像では、0~179 の範囲をとる色相、0~255 の範囲をとる彩度、明度という 3 つの値で表現されるため、閾値はこれらの値を用いて適用する。

図 2.3 に示した HSV 変換を適用する商品棚から分かる通り、様々な色の商品が存在するため、画像全体に同一の閾値を設定すると商品ごとの検出精度に差が生じる。この問題に対処するため、画像内座標を利用し、商品ごとに閾値を設定する。図 2.13 に示したグレースケール変換では検出精度が低くなる商品棚において、各商品に設定した閾値を表 2.2 に示す。この閾値は、図 2.13 に示したグレースケール変換を適用する商品棚において、存在する商品に対し様々な閾値を試した結果、最も検出精度が高くなると判断した値である。商品ごとの閾値設定のイメージを図 2.16 に示す。また、表 2.2 に示す閾値のうち、商品名が none となっているのは、図 2.13 において商品が存在しない棚で誤検出が起こらないか確かめるためのものである。

表2.2: HSV画像における閾値設定

商品名	色相 (H)	彩度 (S)	明度 (V)
白玉粒あんベーグル	20~35	100~255	100~255
クランベリー&クリームチーズベーグル	20~35	100~255	100~255
イチジク&クリームチーズベーグル	20~35	100~255	100~255
明太ポテトベーグル	10~25	100~255	20~200
ビーフカレーベーグル	10~25	100~255	20~200
ベーコンペッパーべーグル	20~35	100~255	100~255
アップルシナモンベーグル	20~35	100~255	100~255
チーズベーグル	10~25	100~255	20~200
none	10~70	100~255	100~255



図2.16: HSV画像の閾値設定イメージ

### 2.3.3 輪郭抽出の手法

輪郭抽出を行うにあたり、RETR\_EXTERNAL と CHAIN\_APPROX\_SIMPLE という 2 つの定数を用いる。

閾値を設定した画像は、各画素において輝度もしくはHSVの値が閾値を満たすか満たさないかのどちらかに区別される。RETR EXTERNALは、閾値を満たす画素の集合のうち、その最外周の輪郭のみを抽出する<sup>[15]</sup>。閾値を満たす画素を白、満たさない画素を黒で表現した商品棚を図2.17に示す。また、RETR EXTERNALによる輪郭抽出のイメージを図2.18に示す。

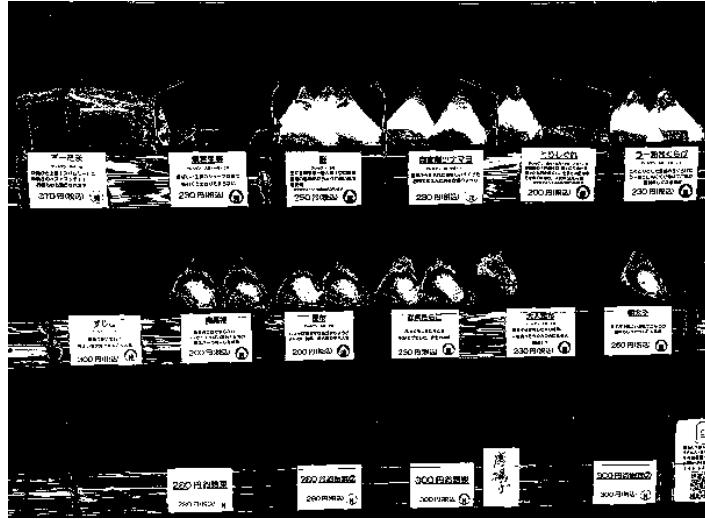


図2.17: 商品と背景を白と黒で表現した画像

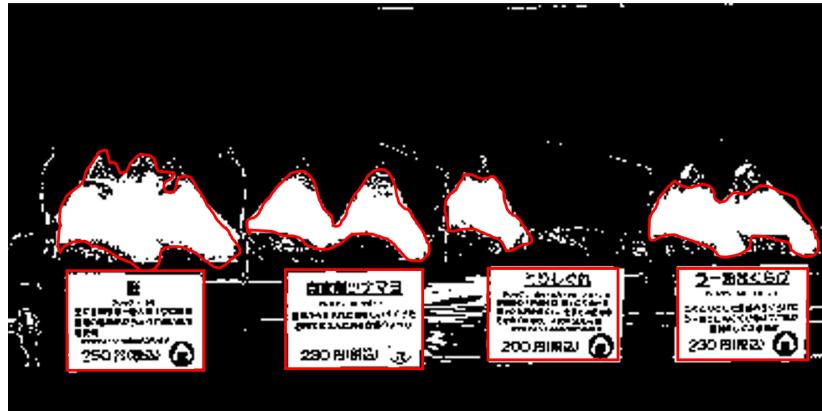


図2.18: RETR EXTERNALの輪郭抽出イメージ

輪郭は無数の点によって表現される。CHAIN APPROX SIMPLEは、輪郭が完全に直線になる部分の点を省いて表現する<sup>[15]</sup>。これによって、リアルタイム性が求められる本研究において、特に重要な処理の高速化を実現できる。CHAIN APPROX SIMPLEによる輪郭表現のイメージを図2.19に示す。

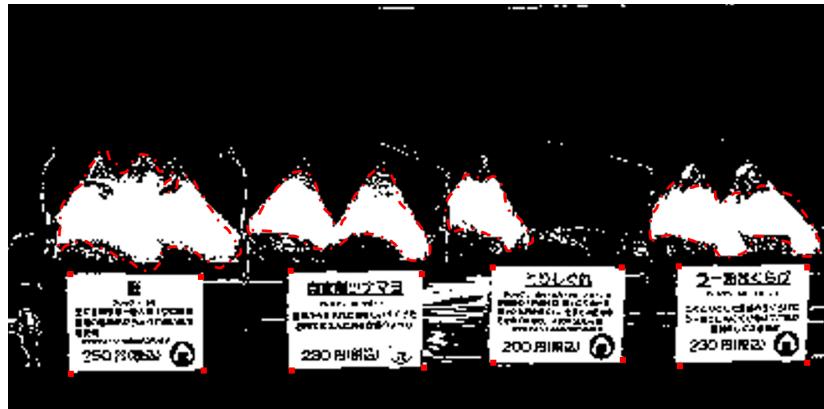


図 2.19: CHAIN APPROX SIMPLE の輪郭抽出イメージ

## 2.4 類似商品の識別

図 2.20 に示す見た目近い異なる商品から分かる通り、無人店舗には見た目から種類の識別が困難な商品が存在する。そこで、抽出した輪郭を基に、画像内座標を用いて商品の種類を識別する。



図 2.20: 見た目近い商品

### 2.4.1 輪郭の座標表現

抽出した商品の輪郭を、1点の座標で表現する。手法としては、輪郭をバウンディングボックスと呼ばれる矩形で表し、その中心点の座標を算出する。図 2.21 に示すバウンディングボックスおよび中心点のイメージから分かる通り、バウンディングボックスは、抽出した輪郭を完全に含み、かつ最小になる矩形であり、バウンディングボックス左上の画像内座標  $(x, y)$ 、バウンディング

ボックスの幅  $w$ , バウンディングボックスの高さ  $h$  という 4 つの情報と, これらの情報から算出したバウンディングボックスの中心点の画像内座標で表現する. 中心点の画像内  $x$  座標  $C_x$  は

$$C_x = x + \frac{w}{2} \quad (2.7)$$

であり, 画像内  $y$  座標  $C_y$  は

$$C_y = y + \frac{h}{2} \quad (2.8)$$

である. バウンディングボックスで商品の輪郭を表現し, 画像内の商品の位置を 1 つの座標  $(C_x, C_y)$  で定義することによって, 商品の画像内の位置情報を利用する処理の高速化が期待できる.

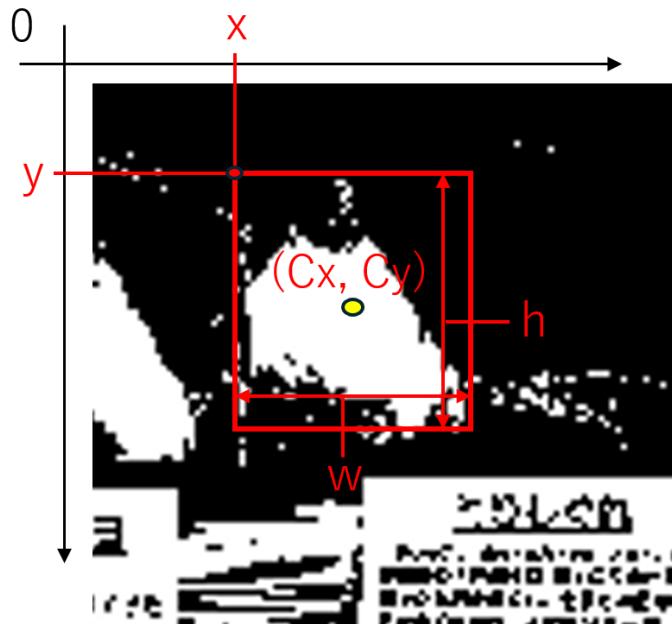


図 2.21: バウンディングボックスおよび中心点イメージ

図 2.17 に示した抽出した輪郭を白で表現した画像から分かる通り, 商品以外に対しても輪郭抽出は行われる. そのため, 商品が存在する範囲のみにバウンディングボックスの表現を適用し, 商品以外の輪郭の座標データを取得しないよう設定する. これによって, 処理の高速化が期待できる. 商品が存在する範囲のみでバウンディングボックスを表現するイメージを図 2.22 に示す.



図 2.22: 商品のみ輪郭抽出のイメージ

図 2.23 に示す輪郭のサイズ指定無しのバウンディングボックス表現から分かる通り，バウンディングボックスは極端に小さい輪郭や大きい輪郭も表現してしまう。そのため，座標データを取得するバウンディングボックスをサイズでフィルタリングする。具体的には，14500~75000 画素数（画素面積）のバウンディングボックスのみ座標データを取得するように設定しており，これは様々な条件でフィルタリングを行った結果，極端なサイズの輪郭を排除し，全商品の輪郭を十分に表現できると判断した閾値である。輪郭のサイズによるフィルタリングを行うことで，ノイズを除去し，余計なデータの取得，計算を避け，処理を高速化できる。輪郭のサイズによるフィルタリングを行ったバウンディングボックス表現を図 2.24 に示す。



図 2.23: サイズ指定なしの検出



図 2.24: サイズ指定ありの検出

## 2.4.2 座標による商品の識別

画像処理による検出を実用化する際、特定の距離と画角からの撮影を想定している。また、無人店舗の商品棚は特定の位置に商品が配置されている。これらを踏まえ、各商品の棚の範囲情報と、抽出した商品の輪郭を表現するバウンディングボックスの中心点の座標情報から商品の種類を特定する。

図 2.25 に示す商品の範囲情報のイメージから分かる通り、各商品の棚の範囲を矩形（以下棚範囲矩形とする）で表現する。手法としては、棚範囲矩形の左上と右下の画像内座標を事前に取得し、商品ごとに定義する。これは、画像内座標が商品によって異なるためである。また、棚範囲矩形と商品の名称を対応させ、各商品の存在する範囲を定義する。

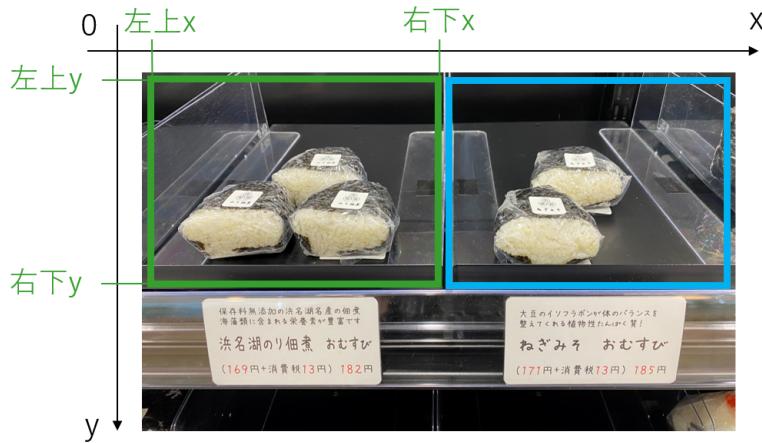


図 2.25: 棚範囲矩形のイメージ

商品名を対応させた棚範囲矩形と、2.4.1 節で述べた商品の輪郭を表すバウンディングボックスの中心点の座標情報を用いて、商品の種類を特定する。具体的には、バウンディングボックスの中心点が特定の棚範囲矩形内に含まれているか、座標情報を用いることで検出し、含まれていた場合、その棚範囲矩形に対応する商品名を割り当てることで、バウンディングボックスが表す商品を特定する。バウンディングボックスの中心点と棚範囲矩形を用いた商品の特定イメージを図 2.26 に示す。

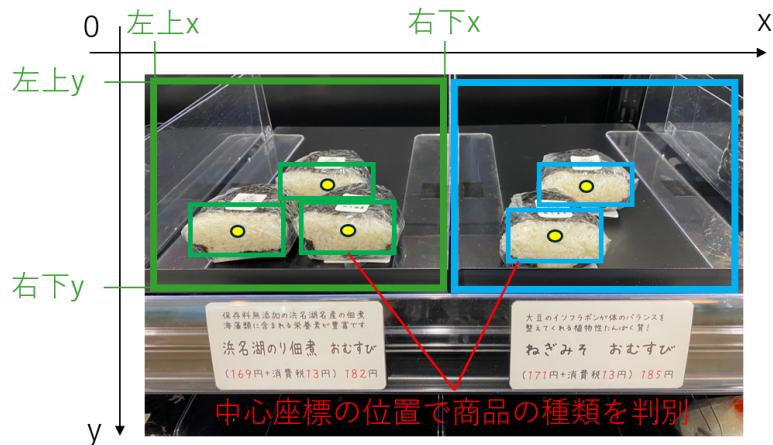


図 2.26: 棚範囲矩形を用いた商品検出のイメージ

## 2.5 検出結果と考察

画像処理による商品検出では、図 2.2 に示したおにぎりなどの白に近い色が特徴的な商品が並ぶ棚に対しては、グレースケール変換を利用し、図 2.3 に示した様々な色の商品が並ぶ棚に対しては、HSV 変換を利用する。

### 2.5.1 グレースケール画像の検出結果

グレースケール画像の検出結果を図 2.27 に示す。この図では、商品ごとの棚範囲矩形、輪郭のバウンディングボックスとその中心点を表現しており、バウンディングボックスの誤検出を確かめるため、商品が存在しない一部の棚にも閾値 100~255 の棚範囲矩形を設定している。また、棚範囲矩形をそれぞれ異なる色で表現し、輪郭のバウンディングボックスを対応する棚範囲矩形と同じ色で示している。



図 2.27: グレースケール画像の検出結果

図 2.27 の検出結果から分かる通り、輪郭の検出と商品の識別が可能であり、商品が存在しない場所で輪郭のバウンディングボックスの誤検出も発生しないことが分かる。一方で、複数の商品をまとめて1つと検出してしまう現象が確認できる。

### 2.5.2 HSV 画像の検出結果

HSV 画像の検出結果を図 2.28 に示す。2.5.1 節で述べたグレースケール画像の検出と同様、棚範囲矩形と輪郭のバウンディングボックス、その中心点、商品が存在しない棚の棚範囲矩形の設定、輪郭のバウンディングボックスと棚範囲矩形の対応の同色表示をそれぞれ表現している。ここで、商品が存在しない棚における棚範囲矩形の閾値は表 2.2 に示す none の値を用いる。

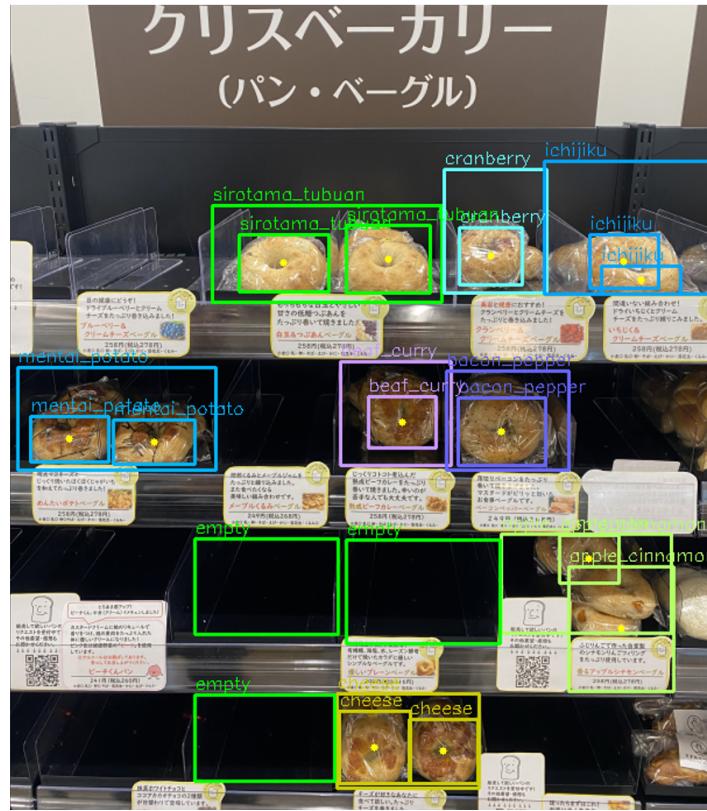


図 2.28: HSV 画像の検出結果

図 2.28 の検出結果から分かる通り、輪郭の検出と商品の識別が可能であり、商品が存在しない場所で輪郭のバウンディングボックスの誤検出も発生しないことが分かる。また、図 2.13 に示した様々な色の商品が並ぶ棚のグレースケール画像から分かる通り、輝度の値がそれぞれ異なる商品が並ぶ棚は、グレースケール変換を用いた場合、検出精度が低くなるが、HSV 変換を用いることで、検出精度の向上が見込める。一方で、グレースケール画像と同様に、複数の商品をまとめて 1 つと検出してしまう現象が確認できる。

### 2.5.3 考察

2.5.1 節および 2.5.2 節で述べた検出結果から、白に近い色が特徴的な商品が並ぶ棚に関してはグレースケール変換を行い、様々な色の商品が並ぶ棚に関しては HSV 変換を用いるのが最適だと考える。図 2.30 に示す無人店舗の商品棚の分類から分かる通り、グレースケール変換を適用すべき商品棚と HSV 変換を適用すべき商品棚は、それぞれ全体の半分の棚範囲を有している。そのため、実用化の際には、グレースケール画像を取得するカメラ、HSV 画像を取得するカメラの計 2 台が必要だと考える。



図 2.29: 画像処理を適用する棚分類

図 2.30 に示す無人店舗の設置場所から分かる通り、無人店舗は屋内に設置されている。そのため、天気をはじめとする外部環境の変化の影響を受けづらく、適切な閾値を設定することで、安定した検出が可能であると考える。しかし、2.5.1 節および 2.5.2 節で述べた検出結果から、グレースケール画像を用いた検出および HSV 画像を用いた検出どちらにおいても、正確な個数の検出は困難だと考える。



図 2.30: 画像処理を適用する棚分類

# 第3章 超音波センサによる在庫の把握

本章では、本研究で行った超音波センサによる在庫把握の手段について述べる。

## 3.1 超音波センサによる在庫把握の方法

本研究では、無人店舗内の商品棚における在庫の把握を超音波センサを用いることによって行った。商品棚の奥側に超音波センサを設置し、センサから発信された超音波が商品に反射して戻ってくるまでの時間を距離に計算し直すことで行う。距離は以下の式で求められる。

$$\text{距離} = \frac{\text{音速} \times \text{時間}}{2} \quad (3.1)$$

ここで、音速は約 343m/s である。計測した距離の長短で在庫の状況を判断する。使用した超音波センサは図 3.1 に示す Rainbow E-Technology 社の HC-SR04 である。このセンサは単体では動作しないため、マイコンボードと組み合わせて使用する必要がある。超音波センサで取得したデータを USB 経由でパソコンに送信し、そのデータをパソコンで評価する。



図 3.1: Rainbow E-Technology 社 HC-SR04

## 3.2 研究に使用した装置

### 3.2.1 超音波センサの性能

HC-SR04 の性能を表 3.1 に示す。測定可能距離は 0.02m～4.5m であり、商品棚での在庫把握は可能であると考えられる。

表 3.1: HC-SR04 の精度

動作電圧	3～5.5V
測定可能距離	0.02～4.5m
測定方式	超音波
動作温度	-10～70°C

### 3.2.2 マイコンボードを含めた装置全体

前述の通り、この超音波センサは単体では動作しないためマイコンボードと組み合わせる必要がある。マイコンボードには Raspberry Pi Pico 2 WH を使用した。装置の全体図を図 3.2 に示す。

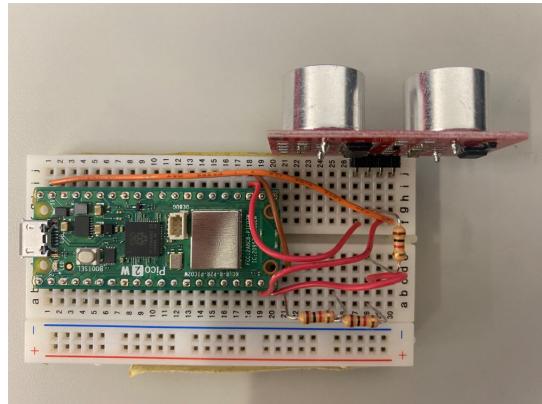


図 3.2: 装置全体図

## 3.3 超音波センサの配置

超音波センサを図 3.3 に示す商品棚の奥側のスペースに配置し、商品との距離を計測する。奥側に設置したのは、配線やセンサ本体が利用客の邪魔にならないようにするためである。



図 3.3: 配置場所

### 3.4 超音波センサの距離測定に使用するプログラム

超音波センサで距離を測定するにあたり、ソースコード A.3 に示す Python のプログラムをマイコンボードに書き込んだ。

ソースコード 3.1: マイコンボードに書き込んだ Python プログラム

```

1 from machine import Pin
2 import machine
3 import utime
4
5 hasshin = Pin(14, Pin.OUT) #14番を出力（発信）と定義
6 jyushin = Pin(15, Pin.IN) #15番を入力（受信）と定義
7
8 def read_distance():
9     hasshin.low() #9行から13行までのコードで超音波を一瞬だけ発信
10    utime.sleep_us(2)
11    hasshin.high()
12    utime.sleep_us(10)
13    hasshin.low()
14    while jyushin.value() == 0: #超音波が受信されていないとき
15        start = utime.ticks_us() #現時点での稼働時間をマイクロ秒単位で返す
16    while jyushin.value() == 1: #超音波が受信されているとき
17        goal = utime.ticks_us() #現時点での稼働時間をマイクロ秒単位で返す
18    #超音波は一瞬なので、jyushin.value()==0となり、無限ループにはならない
19    passed = goal - start
20    distance = float((passed * 340 * 0.0001) / 2) #cmで出力
21    print(distance)
22
23 while True:
24     read_distance()
25     utime.sleep(1)

```

センサで取得した距離から在庫を評価するためのプログラムをソースコード 3.2 に示す。パソコンの OS は Windows を想定している。

### ソースコード 3.2: 在庫評価用のPython プログラム

```
1 from datetime import datetime
2 import serial
3
4 ser = serial.Serial('COM3', 9600) #Windowsの場合
5
6 while(1):
7     dt_now = datetime.now()
8     distance = ser.readline().decode('utf-8').strip()
9     if float(distance) > float(21.0):
10         print(f'{dt_now}在庫なし{distance}')
11     elif float(distance) > float(10.0):
12         print(f'{dt_now}在庫あり{distance}')
13     else:
14         print(f'{dt_now}在庫大量{distance}')
```

評価の対象はベーグルとした。商品によって大きさが違うため、評価対象を変更する場合にはプログラム内の数値を変更する必要がある。

## 3.5 考察

超音波センサで商品とセンサ間の距離を把握することで、実際に陳列されている商品の在庫状況を把握することが可能であると考えるが、3つの課題がある。

1つ目は、商品ごとに評価の基準を設定する必要があることである。前述したソースコード 3.2 のプログラムでは在庫の状況を3段階で評価しているが、評価の基準はセンサから物体までの長さであり、同じ評価基準の場合、商品の大きさによっては正しい結果にならないことがある。そのため、評価の対象を変える場合はプログラム上の数字も変える必要がある。また、商品の一部分が他の商品と重なっている場合、評価の基準を正しく設定したとしても正しい結果にはならないことがある。

2つ目は、無人店舗で取り扱っている商品は60種類とかなり多いため、超音波センサを1商品に対し1台設置すると多額の費用が発生し電源供給の課題も発生するという課題である。その課題を解決するには、1台のマイコンボードに対し2つ以上の超音波センサを設置するなどの工夫が必要となる。

3つ目は、無人店舗において店舗利用者が商品を手に取ったあと、商品棚の奥側にある在庫が前出されないという課題である。商品の前出しがされないと実際は在庫数が少ないが在庫数に余裕ありと判断されるため、在庫把握に超音波センサを用いる場合はこの問題を対処する必要がある。この問題を解決するために一部の小売店では図3.4のような自動で前出しされるフェイシングスタンド<sup>[16]</sup>を用いている。フェイシングスタンドを使用すれば商品が自動で前出しされるという利点があるが、無人店舗は商品の種類が多く、大量のフェイシングスタンドが必要なため導入のハードルが高い。また、無人店舗にはおにぎりのように潰れてしまいやすい商品があるため、導入は厳しい。



図3.4: フェイシングスタンド

これらの課題から、本研究では在庫把握の手段として超音波センサを用いるのは困難と判断した。

# 第4章 デプスカメラによる在庫把握

本章では、本研究で行ったデプスカメラによる在庫把握の手段について述べる。

## 4.1 デプスカメラによる在庫把握の方法

デプスカメラで無人店舗内の商品棚を撮影し、カメラと物体の距離を測定することでの在庫把握を行う。使用したデプスカメラは図4.1に示す Arducam 社の B0410 である。このデプスカメラは 3D-ToF (Time of Flight) 方式のカメラで、赤外光を用いて距離を計測している。赤外光を用いることで、光が入らない暗い場所でも使用できる。



図4.1: Arducam 社 B0410

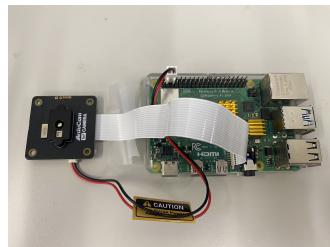


図4.2: B0410 を接続した Raspberry Pi

### 4.1.1 デプスカメラの精度

B0410 の性能を表4.1に示す。

表4.1: デプスカメラの性能

型番	B0410
解像度	240x180
コマ数	30fps
計測範囲	2m または 4m
画角	対角 70°
接続方法	Raspberry Pi 本体に接続

## 4.2 デプスカメラの配置

デプスカメラの配置方法は、防犯カメラのように店舗の壁面に設置する方法または超音波センサと同じように商品棚の奥側に設置する方法の 2 通りがある。前者の設置方法では 1 台のカメラで複数の商品の在庫を把握できるという利点があるが、今回使用したデプスカメラは解像度が非常に低いえ、商品棚全体を認識できるようにデプスカメラを設置することが困難であった。そのため、本研究では後者の商品棚の奥側に設置する方法を採用した。

## 4.3 デプスカメラでの距離測定に使用するプログラム

デプスカメラで距離を測定するにあたり、Arducam 社の SDK をダウンロードし、Raspberry Pi で実行する Python のプログラム (preview\_depth.py) を使用した。このプログラムを実行すると、デプスカメラで撮影された映像が図 4.3 のように Raspberry Pi 上に表示される。ここで、画像左側の「preview\_confidence」は赤外線画像で、画面右側の「preview」は、計測した距離を並べて画像にしたもの OpenCV のカラーマップである cv2.COLORMAP\_RAINBOW で擬似的にカラー化した画像である。

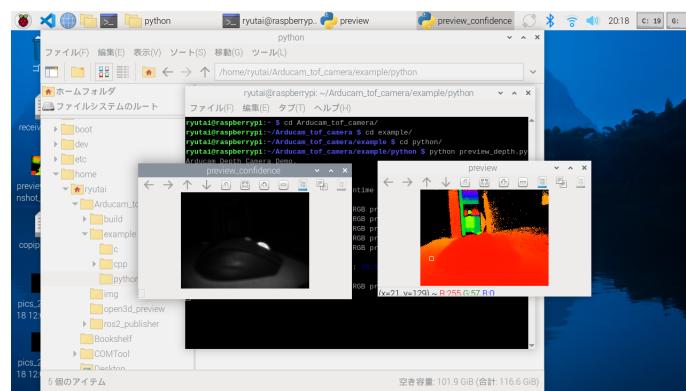


図 4.3: Raspberry Pi のスクリーンショット

## 4.4 考察

デプスカメラを用いて物体とカメラ間の距離を計測することで、超音波センサでの課題を解決しながら商品の在庫把握が可能であると考えたが、本研究で採用したデプスカメラの設置方法では、デプスカメラを大量に用意する必要がありそのうえ電源供給の課題が発生する。B0410は1台あたり9000円でありカメラ1台につき Raspberry Pi が1台必要なため、商品1種類につきB0410を1台用意する場合には多額の費用が発生する。

商品棚全体を認識できるようにデプスカメラを配置できれば、カメラの台数を少なくしつつ第2章で用いたOpenCVによる画像処理で在庫把握が行える。だが、今回使用したB0410はカラー画像の出力には対応していないため、第2章で用いたHSV変換による物体検出の手法が使えない。

高解像度かつカラー表示ができるデプスカメラを壁面に設置することで、これらの問題が解決するが、依然として商品棚に陳列されている在庫の具体的な個数の把握は困難である。

# 第5章 LINE Botを用いた遠隔の在庫把握

第4章で述べたように、無人店舗の在庫状況に関するデータの収集には、デプスカメラを用いる手法が最適と判断した。本章では、収集したデータとLINE Botを用いて、利用者が場所の制約なく無人店舗の在庫状況を閲覧できるシステムの構築について、LINE Bot, ngrok, システムの動作結果と考察を説明する。

## 5.1 LINE Bot

LINE Botは、LINE上で動作する自動応答プログラムであり、利用者から送られる特定の文字や数字に対し、あらかじめ設定したメッセージを自動で返信する機能を持つ<sup>[20]</sup>。また、LINEはLINEヤフー株式会社が提供する日本最大級のコミュニケーションアプリであり、総務省による全年代を対象としたソーシャルメディアの利用率調査では、91.1%の人々がLINEを利用していると回答しているため<sup>[21]</sup>、無人店舗の利用者に限らず多くの人が使用経験のあるアプリケーションだと考える。実際に無人店舗の在庫状況を遠隔から確認する際、慣れ親しんだ画面で操作でき、手軽かつ分かりやすく在庫状況を把握できると考えため、本研究ではLINE Botを用いる。

LINE Botは、主に公式アカウント、Messaging API、Webhookという3つの要素で構成されている。

### 5.1.1 公式アカウント

LINEは、コミュニケーションをとりたいアカウントに対し、友達追加を行うことで、文字や画像、スタンプのやり取りができる。そのため、LINE Botを利用するにあたり、紐づけるアカウントが必要である。これを公式アカウントと呼び、本研究では、無人店舗の在庫状況を返答する公式アカウントを作成し、利用者が友達追加することで、遠隔の在庫把握を実現する。LINE上のやり取りの様子を図5.1に示す。



図5.1: LINE 上のやり取りの様子

### 5.1.2 Messaging API

Messaging APIは、LINE公式アカウントを通じて、利用者と双方向のコミュニケーションを実現するAPIであり、利用者から送られるメッセージに応答する機能を備えている<sup>[22]</sup>.

API (Application Programming Interface) は、ソフトウェアやWebサービスの間をつなぐインターフェースであり、これを利用することで、外部サービスからのデータ取得を実現できる<sup>[23]</sup>.

Messaging APIを利用するにあたり、無料で利用できるコミュニケーションプラン、月額5000円のライトプラン、月額15000円のスタンダードプランという、それぞれ返信できるメッセージ数が異なる3つのプランが存在するが、本研究ではLINE Botの有効性を調査するのが目的であり、大量のメッセージを返信する実用段階には満たないため、コミュニケーションプランを用いている。

図5.2に示すLINE Botを用いて在庫状況を把握するシステムの構造から分かる通り、Messaging APIは、Flaskサーバに格納された在庫データを、LINEプラットフォームを通じて利用者に送信するために用いる。ここで、LINEプラットフォームは、LINEヤフー株式会社が提供する通信および制御を行うための基盤であり、Messaging APIはこの基盤上で提供されるサービスの1つである。また、Flaskサーバは、Pythonプログラミング言語を用いたWebアプリケーションフレームワークであり、シンプルかつ軽量、さらに柔軟性や拡張性が高いという特徴がある<sup>[24]</sup>。本研究では、LINE Botの有効性を調査するのが目的のため、検出した在庫データを保持するサーバとして十分な機能を持ち、無料で利用できるFlaskサーバを用いた。

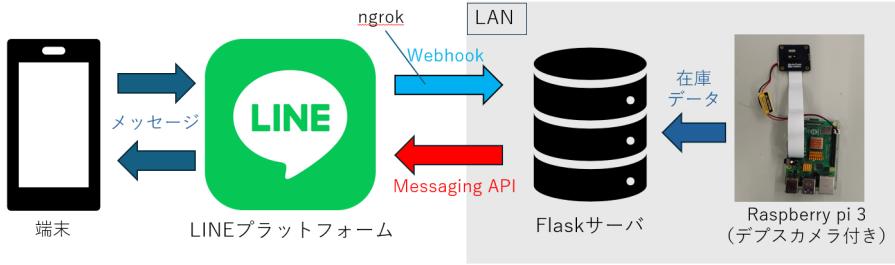


図 5.2: 遠隔在庫把握システムの構造

Messaging API を利用するためには、図 5.3 に示す LINE Bot の Messaging API 設定画面（以下 LINE Developers 画面とする）のように、チャネルアクセストークンの発行が必要である。これは、Messaging API を安全に利用するための認証キーであり、本研究では Flask サーバから LINE プラットフォームへメッセージ送信を行うための身分証としての役割を果たす。図 5.3 は、セキュリティ上の理由から、チャネルアクセストークンを隠して示している。



図 5.3: チャネルアクセストークン発行画面

### 5.1.3 Webhook と ngrok

Webhook は、特定のイベントが発生した際、あらかじめ設定したアプリケーションに通知を自動で送信する仕組みであり<sup>[25]</sup>、図 5.2 に示した遠隔在庫把握システムの構造から分かる通り、本研究では、在庫データを管理する Flask サーバに対し、イベントの発生を通知する際に用いる。遠隔在庫把握システムにおけるイベントとは、利用者からのメッセージを指し、Webhook を利用することで、利用者のメッセージに対して応答する仕組みを実現できる。

Webhook を利用し、イベントの発生を通知するにあたり、ローカル環境で動作するサーバから外部ネットワークへの通信はツールを用いず実現できるが、外部ネットワークからローカル環境への通信は特定のツールが必要という問題が存在する。この問題を解決するツールの1つに ngrok があり<sup>[26]</sup>、図5.2に示した遠隔在庫把握システムの構造から分かる通り、本研究ではインターネット上のLINE プラットフォームから学内 LAN 内にある Flask サーバに Webhook を送信するため ngrok を用いる。このツールの特徴として、無料かつ簡単に利用できる点が挙げられる。また、ngrok は、より便利な機能を備えた有償版もあるが、本研究では LINE Bot の有効性調査に支障のない無料版を用いる。

LINE Bot における ngrok を用いた Webhook の具体的な設定方法について説明する。テキストベースのプログラム実行ツール（以下ターミナルとする）において、ソースコード5.1に示す Flask サーバのポート公開コードのように、ポート番号を 5000 として ngrok に公開 URL を生成させる。ここで、ポート番号を 5000 としているのは、Flask サーバのデフォルト設定がこの番号のためであり<sup>[27]</sup>、任意に変更することができるが、本研究ではデフォルトの番号を用いている。生成される公開 URL を図5.4に示すが、セキュリティ上の理由から、URL を隠して示している。

ソースコード 5.1: Flask サーバのポート公開用コード

```
1 ngrok http 5000
```

Version	3.24.0-msix
Region	Japan (jp)
Latency	20ms
Web Interface	http://[REDACTED]
Forwarding	https://[REDACTED] -> http://localhost:5000

図5.4: ngrok で生成される公開 URL

図5.4に示したngrokによって生成される公開URLに「/callback」を付与したものを、図5.5に示すLINE DevelopersのWebhook設定画面のように、Webhook URLとして設定し、Webhookの利用をONにする。ここで、「/callback」は、本研究においてFlaskサーバがWebhookを受け取るための窓口を示しており、利用者のメッセージに沿った応答をするために必要なコードである。また、Webhook利用時に自動で発行されるチャネルシークレットという文字列を用いてWebhookの認証を行う。チャネルシークレットはLINE Developersから確認できる。これらの作業によって、公式アカウントに利用者がメッセージを送信した際、インターネット上のLINE プラットフォームを通じて、学内 LAN 内の Flask サーバへイベントの発生を通知できる。「/callback」を用いた Webhook の窓口設定に関するコードをソースコード5.2に示す。このコードにおいて、signature は Webhook の通知が LINE プラットフォームから送られた正規のものであるか、改ざんされていないかを認証するための署名であり、body はメッセージ内容、ユーザ ID などのイベント情報を文字列で取得するためのコードである。また、図5.5はセキュリティ上の理由から、Webhook URLを隠して示している。



図 5.5: Webhook 設定画面

ソースコード 5.2: Webhook の窓口設定用コード

```

1 from flask import Flask, request, abort
2 from linebot import WebhookHandler
3
4 # Webhook受信
5 @app.route("/callback", methods=['POST'])
6 def callback():
7
8     # LINEからの改ざん防止用署名を取得
9     signature = request.headers['X-Line-Signature']
10
11    # イベント情報 (JSON形式) を取得
12    body = request.get_data(as_text=True)
13    print("Received body:", body)
14
15    # 処理の受け渡し
16    try:
17        handler.handle(body, signature)
18
19    # エラー発生時の処理
20    except Exception as e:
21        print("[ERROR] LINE handler:", e)
22        abort(400)
23
24    return 'OK'

```

## 5.2 遠隔在庫把握システムの流れ

まず、Flask サーバに在庫データを格納する仕組みについて説明する。図 5.2 に示した遠隔在庫把握システムの構造から分かる通り、デプスカメラで検出した商品在庫の個数情報を、Raspberry pi3 を用いて JSON 形式で Flask サーバに送信する。この際、Flask サーバに保持するデータは各商品 1つであり、10 秒間隔で最新のデータに更新する。ここで、最新のデータのみを格納している理由としては、Flask サーバの処理を軽くするためである。また、素早く返信を行うために、利用者からのメッセージを検知して Flask サーバがデータ取得を始めるのではなく、Flask サーバにあらかじめ格納されている最新の在庫データを用いて返信を行う。Raspberry pi3 に実装する Flask サーバへデータを送信するためのコードをソースコード 5.3 に示し、Flask サーバ側に実装する Raspberry pi3 から送信されるデータ受信用のコードをソースコード 5.4 に示す。本研究では、LINE Bot の有効性調査のため、実際に検出された値ではなく、1つの商品の検出値に見立てた実験用のダミー値を利用している。実用化の際には無人店舗の全商品の在庫を検出、データとして格納することを想定している。注意点として、Flask サーバを立てる PC と Raspberry pi3 は同一 LAN 内で通信を行う必要がある。本研究では学内 LAN を用いている。また、セキュリティ上の理由から、ソースコード 5.3において Flask サーバを立てた PC の IP アドレスは、実際のものではなく例を示している。

ソースコード 5.3: Raspberry pi3 から Flask サーバへのデータ送信用コード

```
1 import requests
2 import json
3 import time
4
5 # サーバのURL指定（IP アドレスは伏せている）
6 FLASK_SERVER_URL = "http://[サーバを立てたPCのIPアドレス(IPv4)]:5000/upload"
7
8 # データ送信準備
9 def send_data_to_server(value):
10     data = {"value": value}
11     try:
12         res = requests.post(FLASK_SERVER_URL, json=data)
13         print(f"[SEND] Sent {value}, Status: {res.status_code}")
14
15     # エラー処理
16     except Exception as e:
17         print(f"[ERROR] {e}")
18
19 # データ定期送信
20 if __name__ == "__main__":
21     while True:
22
23         # センサ値のダミーデータ（実用化の際は検出したセンサデータに切り替える）
24         fake_value = 5
25         send_data_to_server(fake_value)
```

```
time.sleep(10)
```

ソースコード 5.4: Flask サーバ側に実装する Raspberry からのデータ受信用コード

```

1 from flask import Flask, request, jsonify
2 from datetime import datetime
3
4 # Raspberry pi3からデータ受信
5 @app.route("/upload", methods=['POST'])
6 def upload():
7
8     # JSON形式でデータ受信
9     try:
10         data = request.get_json()
11
12     # データの妥当性確認
13     if not data or "value" not in data:
14         return jsonify({"error": "Invalid JSON"}), 400
15
16     # データ取得
17     value = data["value"]
18
19     # 最新データに更新
20     latest_data["timestamp"] = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
21     latest_data["value"] = value
22
23     print(f"[UPLOAD] Received new data from device: {value}")
24
25     return jsonify({"status": "success", "received_value": value}), 200
26 except Exception as e:
27     print("[ERROR] Upload failed:", e)
28     return jsonify({"error": str(e)}), 500

```

ソースコード 5.5 に示す Web ブラウザを用いたデータ閲覧のためのコードを用いて、Flask サーバに格納される最新の在庫データを可視化する。図 5.6 示す可視化したデータから分かる通り、商品名、更新時間、数量を可視化している。

ソースコード 5.5: ブラウザ上でのデータ閲覧用コード

```

1 from flask import Flask
2
3 # 最新データを保持する変数
4 latest_data = {"ブルーベリー & クリームチーズベーグル": {"timestamp": None, "value": None}}
5
6 @app.route("/status")
7
8 # HTMLで記述
9 def status():
10     html = """
11     <!DOCTYPE html>
12     <html lang="ja">
13         <head>

```

```
14 |         <meta charset="UTF-8">
15 |         <title>最新データ</title>
16 |         <style>
17 |             body{
18 |                 font-family: "Segoe UI", sans-serif;
19 |                 background-color: #ffffff;
20 |                 color: #000000;
21 |                 padding: 50px;
22 |             }
23 |             h1{
24 |                 font-size: 28px;
25 |                 font-weight: 700; /*太め*/
26 |                 margin-bottom: 20px;
27 |             }
28 |             ul{
29 |                 list-style-type: none;
30 |                 padding-left: 10px;
31 |                 font-size: 18px;
32 |                 font-weight: 400; /*通常の太さ*/
33 |                 line-height: 1.8;
34 |             }
35 |         </style>
36 |     </head>
37 |     <body>
38 |         <h1>最新データ</h1>
39 |         <ul>
40 |             <li>商品名：ブルーベリー & クリームチーズベーグル</li>
41 |             <li>時間：{latest_data['timestamp']}
```

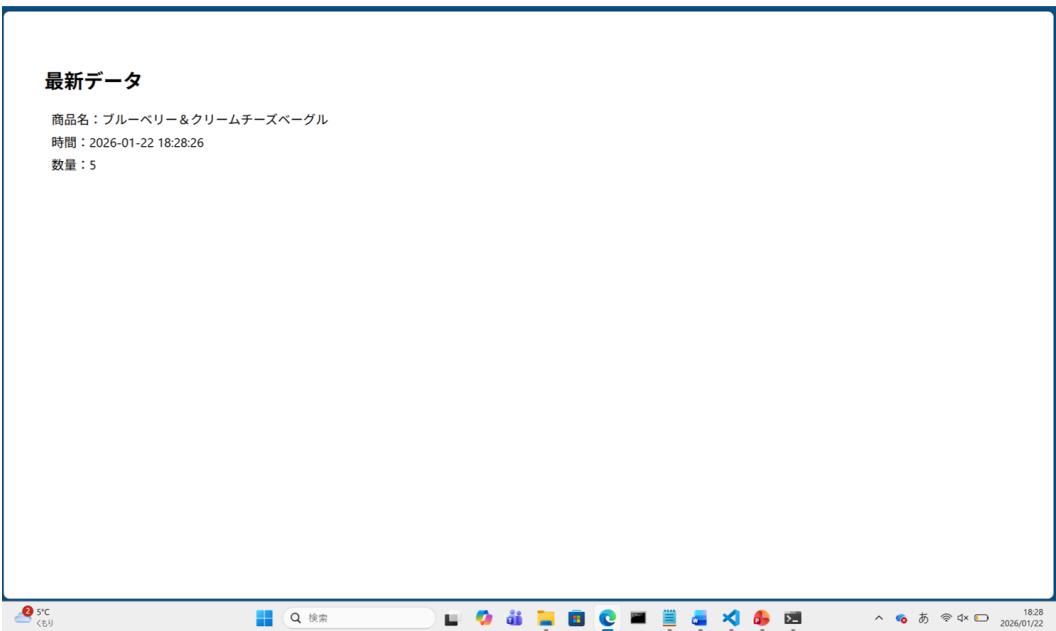


図 5.6: ブラウザで可視化した在庫データ

Flask サーバに最新の在庫データが格納されることを前提に、LINE 公式アカウントに利用者がメッセージを送信してから返信されるまでの、LINE プラットフォームと FLASK サーバ間の Messaging API と Webhook の通信の流れについて説明する。図 5.2 に示した遠隔在庫把握システムの構造から分かる通り、利用者がメッセージを送信した際、そのメッセージは LINE プラットフォームが受け取る。その後、6.1.3 節で述べたように、ngrok による Webhook の利用で、Flask サーバへイベント発生の通知を行い、利用者からのメッセージを認識、その内容があらかじめ定めたテキストと一致した場合、自動で返信を行う。この返信には 6.1.2 節で述べたように、Messaging API が用いられ、本研究では商品名を「ブルーベリー＆クリームチーズベーグル」とし、「「ブルーベリー＆クリームチーズベーグル」の在庫」というメッセージを受信した際にその個数を返答するよう設定、検証を行う。この返答は、LINE プラットフォームを通じて、利用者へ送信される。利用者からのメッセージテキストを認識し、Messaging API を用いて返答するためのコードをソースコード 5.6 に示す。ここで、LINE 公式アカウントの認証に用いるチャネルアクセストークンとチャネルシークレットは、セキュリティ上の理由から伏せる。

ソースコード 5.6: LINE プラットフォームを通して利用者へ返信するためのコード

```

1 from linebot import LineBotApi, WebhookHandler
2 from linebot.models import MessageEvent, TextMessage, TextSendMessage
3
4 # 認証情報の設定
5 LINE_CHANNEL_ACCESS_TOKEN = ''
6 LINE_CHANNEL_SECRET = ''
7 line_bot_api = LineBotApi(LINE_CHANNEL_ACCESS_TOKEN)
8 handler = WebhookHandler(LINE_CHANNEL_SECRET)

```

```

9
10 # 最新データを保持する変数
11 latest_data = {"ブルーベリー＆クリームチーズベーグル": {"timestamp": None, "value": None}}
12
13 @handler.add(MessageEvent, message=TextMessage)
14 def handle_message(event):
15     text = event.message.text
16     if text == "「ブルーベリー＆クリームチーズベーグル」の在庫":
17         reply = f"現在の「ブルーベリー＆クリームチーズベーグル」は、{latest_data['value']}個で
18             す。"
19         line_bot_api.reply_message(event.reply_token, TextSendMessage(text=reply))

```

### 5.3 公式アカウント上のメッセージ送信手法

本研究では、LINE 公式アカウント上で利用者が在庫状況を確認する際、商品の選択を行うためのメッセージ送信手法として、リッチメニューおよびリッチメッセージを用いる。これは、画像およびテキストによって表現されたバナーを利用者が選択することにより、バナーごとに設定された固定メッセージが自動で送信されるものである。リッチメニューおよびリッチメッセージの特徴として、視覚的に情報を表現できること、テキストを入力する必要がなく、公式アカウント管理者が設定したメッセージを利用者に送信させられることの2つが挙げられる。また、リッチメニューは図 5.7 に示すように、トーク画面を開いたときに表示される選択式バナーであり、リッチメッセージはソースコード A.6 に示すように、リッチメニューを選択したとき、公式アカウントの返信として表示される選択式バナーである。

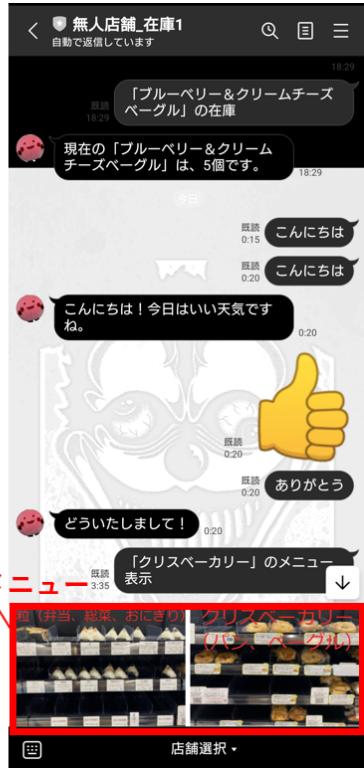


図5.7: リッチメニュー表示



図5.8: リッチメッセージ表示

無人店舗の商品は数が多いため、商品名とその見た目を記憶するのは困難である。そのため、図5.7およびソースコードA.6に示す実際のLINE公式アカウントのトーク画面から分かる通り、リッチメニューおよびリッチメッセージを用いて商品棚や商品名の見た目と名称を同時に表示することで、利用者の想定する商品を選択することができる。

6.2節で述べたように、在庫データを返信するには、利用者の送信するメッセージテキストが、あらかじめ定めたテキストに完全に一致している必要がある。図5.9および図5.10に示すLINE公式アカウント上のやり取りから分かる通り、リッチメニューおよびリッチメッセージはバナーをタップすると必ず固定のメッセージが送信され、そのメッセージは公式アカウント管理者が設定できる。そのため、これらを利用することで、迅速かつ正確な在庫データの提供が可能になる。

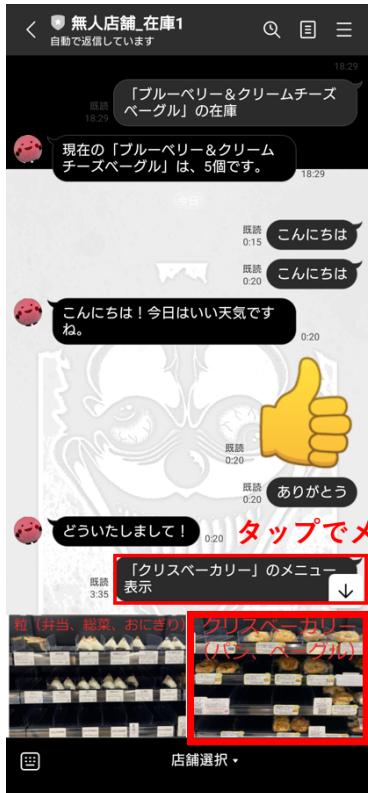


図5.9: リッチメニューのタップ操作



図5.10: リッチメッセージのタップ操作

リッチメニューおよびリッチメッセージの設定方法について説明する。図5.11に示すLINE公式アカウントのリッチメニュー設定画面から、リッチメニューの表現テンプレート、適用する画像およびテキスト、タップ操作時のアクション、メニューバーをそれぞれ設定する。これに表示期間を設定し、公開することで図5.7に示したように、トーク画面を開いた際にリッチメニューが表示される。



図5.11: リッチメニュー設定画面

図5.12に示すLINE公式アカウントのリッチメッセージ設定画面から、リッチメッセージの表現テンプレート、適用する画像およびテキスト、タップ操作時のアクションをそれぞれ設定する。その後、図5.13に示すLINE公式アカウントの応答メッセージ設定画面から、リッチメッセージが返信として表示されるためのキーワードを設定することで、図6.8に示したように、利用者がキーワードを公式アカウントに送信した際、対応するリッチメッセージが表示される。利用者のキーワード送信を、リッチメニューおよび別のリッチメッセージでタップ操作した際のアクションに設定することで、タップ操作のみで特定の商品に辿り着くことができる。リッチメニューおよびリッチメッセージを利用した商品検索の流れを図5.14に示す。



図5.12: リッチメッセージ設定画面



図5.13: 応答メッセージ設定画面



図5.14: 商品検索の流れ

## 5.4 システムの動作結果と考察, 課題

図5.6に示したFlaskサーバの格納データ可視化画面から分かる通り, Raspberry pi3からFlaskサーバへのデータ送信および格納については, 正確に行うことができた.

6.2節で述べたように, Messaging API, Webhook, ngrokを用いて, 利用者が「ブルーベリー & クリームチーズベーグル」の在庫を把握するため, LINE公式アカウントにメッセージを送信した際の結果を図5.15に示す. この結果から, LINE Botを用いた在庫データの把握が可能であると判断した.

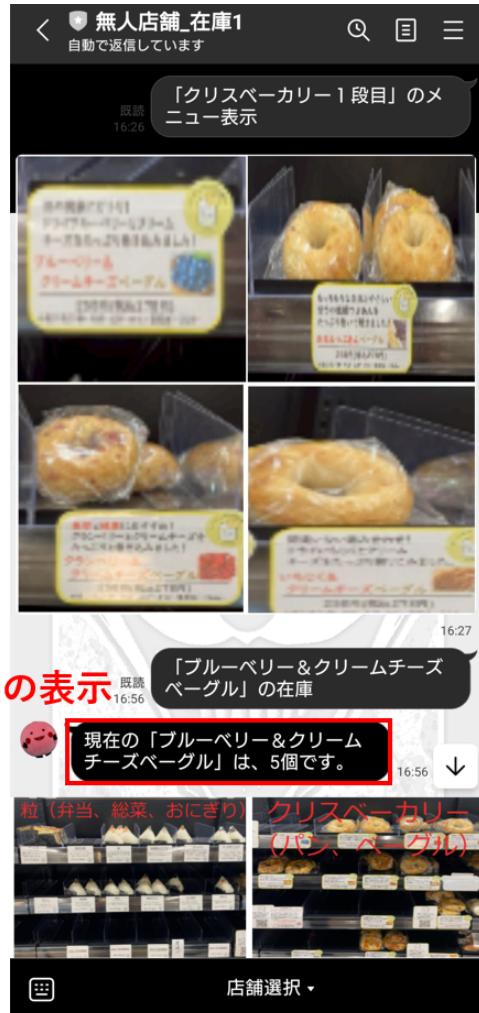


図 5.15: 遠隔在庫把握の結果

6.2 節で述べたように、本研究では、処理の軽量化のため Flask サーバに最新のデータのみを格納している。しかし、過去の在庫データの履歴を時刻情報などと併せて保持することで、遠隔の在庫把握以外、例えば、商品の時刻ごとの売れ行きや天気、季節による売れ行きの変化など、様々な目的に応用できると考える。

本研究では、利用者がメッセージを送信したときに在庫を返信するシステムに関して、LINE Bot による実現性を確認するため在庫データの格納および中継方法に Flask サーバを用いた。しかし、実用化の際は、當時、もしくは長時間の運用が考えられるため、本研究と同様に PC 上の Flask サーバを用いると、1台の PC を Raspberry pi3 と同じ LAN 内で起動させ続ける必要がある。そこで、Raspberry pi 上で Flask サーバを起動する方法が考えられる。これは、検出を行う Raspberry pi3 自体を用いても可能だが、負荷分散のため検出用とは別の Raspberry pi が適切であると考える。また、より安定的な稼働のため、AWS、Microsoft Azure、Google Cloud といったクラウドサービスの利用も考えられる。

このシステムの実用化にあたり、2つの課題がある。

1つ目は、Messaging APIの使用に関する通数制限である。6.1.2節で述べたように Messaging APIには、3つのプランがあり、それぞれで送信できるメッセージの数量が変わる。具体的には、無料のコミュニケーションプランで月200通、月5000円のライトプランで月5000通、月15000円のスタンダードプランで月30000通となっており、この通数を超過すると、コミュニケーションプランおよびスタンダードプランではLINE公式アカウントからメッセージを送信ができなくなり、スタンダードプランでは利用できるが、1通の超過に対し3円の追加料金が発生する<sup>[28]</sup>。そのため、無人店舗の遠隔在庫把握システムを多くの人が利用する場合、LINE Botを用いると費用が大きくなると考えられる。ここで使用している通数という単位は  
通数 = 送信メッセージ数 × 送信先アカウント数で計算できる。

2つ目は、PCでLINEを利用する場合、リッチメニューが表示されない点である。図5.16に示すPC版LINEで表示した公式アカウントのトーク画面から分かる通り、リッチメニューはモバイル専用機能であり、PC版およびWeb版のLINEでは表示できない<sup>[29]</sup>。そのため、実用化の際、モバイル機器以外を用いる利用者に向けて遠隔在庫把握システムを構築する場合は、本章で説明したリッチメニューを利用した商品検索とは別の手法を用いる必要がある。

## 第6章 結論

本研究では、無人店舗の在庫検出において実現性の高い手法を検討するため、OpenCV を用いた画像処理、超音波センサを用いたセンシング、デプスカメラを用いたセンシングという 3 つの手法を検討した。さらに、遠隔から在庫状況を把握可能なシステムの構築を目的として、LINE Bot を用いた在庫データ共有の実用性を検証した。

OpenCV を用いた画像処理では、白に近い色が特徴的な商品の並ぶ棚に関してはグレースケール変換を、様々な色の商品が並ぶ棚に HSV 変換をそれぞれ適用することで、一定の精度の検出が可能となった。しかし、画像上において重なっている商品および密接している商品の検出は困難であった。また、検出には画像内座標を用いるため、実用化の際、画像データ取得用カメラの位置と画角を固定する必要があり、設置が困難という物理的課題が明らかになった。

超音波センサを用いたセンシングでは、商品とセンサとの距離を在庫の個数へと変換し、在庫状況の把握をする手法の確立を行った。これにより、商品の在庫が「大量にある」、「ある」、「ない」の 3 段階での状況把握が可能となったが、商品の前出しがなされない問題や電力供給の問題が明らかになった。そのため、超音波センサでは在庫把握は困難であるとした。

デプスカメラを用いたセンシングでは、商品の前出しがなされない問題が解決される。だが、カメラを壁面に設置できないと大量のデプスカメラを用意する必要がある。デプスカメラを壁面に設置できれば、カメラの台数を少なくしつつ OpenCV を用いた画像処理の手法が使用できる。よって、カメラを壁面に設置したうえでデプスカメラによる在庫把握が最適だと考えたが、依然として在庫の具体的な個数の検出は困難である。

LINE Bot を用いた在庫データ共有の実用性の検証では、Raspberry pi3 から Flask サーバへデータを定期的に送信し、格納する手法の確立、加えて Messaging API や Webhook などの仕組みを利用し、LINE 上で利用者のメッセージに反応して返信する手法の確立を行った。これらを組み合わせることで、無人店舗の LINE 公式アカウントから、商品の在庫を利用者の任意のタイミングで確認できる遠隔在庫把握システムの基盤を構築した。しかし、実用化を考えると長時間の安定した稼働が必要であり、Flask サーバではそれが実現しづらい。また、Messaging API の通数制限により、実用化によって無人店舗の公式アカウントの利用者が増えるにつれ、必要なコストが増加するという課題が明らかになった。

今後の課題としては、在庫検出手法と遠隔把握手法を組み合わせた遠隔在庫把握システムの構築、運用であり、そのためには本研究で示した検出精度、実現性、コストの大きさといった観点からの課題に取り組み、解消していく必要がある。また、これらの課題に取り組み、遠隔在庫把握システムの運用を実現することで、無人店舗の利便性向上による利用者の増加が期待できる。

# 参考文献

- [1] Digital Intelligence チャンネル, 「センシングとは? 基礎から注意点を分かりやすく解説し活用例を紹介」, <https://www.cloud-for-all.com/blog/what-is-sensing>, 最終更新 2024.12.11
- [2] Schoo for Business, 「画像処理とは? その特徴や活用される事例について解説する」, <https://schoo.jp/biz/column/885>, 最終更新 2025.09.19
- [3] TTG, 「無人店舗の仕組みとは? メリットやデメリットと万引き対策まで徹底解説」, <https://ttg.co.jp/media/unmanned-store-system>, 最終更新 2025.05.08
- [4] Device HD, 「Sony Xperia 10 VI 5G」, <https://devicehd.com/smartphones/en/product/66c3181893f2fb776375368c/>
- [5] SAMURAIENGINEER Blog, 「OpenCV とは? できることや特徴をわかりやすく解説」, <https://www.sejuku.net/blog/113292>, 最終更新 2025.12.26
- [6] Qiita, 「グレースケール画像の うんちく」, <https://qiita.com/yoya/items/96c36b069e74398796f3>, 最終更新 2025.04.20
- [7] MUSASHI AI, 「グレースケール変換」, <https://musashi-ai.com/glossary/2023/06/132b15f51be239030f34b40a152b0c724506964c>
- [8] Mustafa Murat ARAT, 「RGB to Grayscale Conversion」, [https://mmuratarat.github.io/2020-05-13/rgb\\_to\\_grayscale\\_formulas](https://mmuratarat.github.io/2020-05-13/rgb_to_grayscale_formulas), 最終更新 2020.05.13
- [9] Gigahertz-Optik, 「1.6 Spectral Sensitivity of the Human Eye」, <https://www.gigahertz-optik.com/en-us/service-and-support/knowledge-base/basics-light-measurement/light-color/spectr-sens-eye/>
- [10] Britannica, 「RGB color model」, <https://www.britannica.com/science/RGB-color-model>, 最終更新 2026.01.23
- [11] PEKO STEP, 「HSV 色空間」, <https://www.peko-step.com/html/hsv.html>
- [12] OpenCV, 「Color spaces in OpenCV」, <https://opencv.org/blog/color-spaces-in-opencv/#h-hsv-hue-saturation-value-color-space>, 最終更新 2025.04.29
- [13] 数理超入門部, 「HSV 色空間とは? RGB から変換するときの計算式」, <https://algorithm.hatenablog.com/entry/2023/07/10/183000>

joho.info/image-processing/hsv-color-space/, 最終更新 2017.07.04

- [14] IT 用語辞典 e-Words, 「閾値【threshold】しきい値」, <https://e-words.jp/w/%E9%96%BE%E5%80%A4.html>
- [15] OpenCV オープンソースのすすめ, 「OpenCV の findContours 関数を使った画像の輪郭検出」, [https://www.argocorp.com/OpenCV/imageprocessing/opencv\\_find\\_contours.html](https://www.argocorp.com/OpenCV/imageprocessing/opencv_find_contours.html)
- [16] フェイシングスタンド 河淳 スタンド/仕切板/仕切ワイヤー 【通販モノタロウ】 ,<https://www.monotaro.com/g/02794558/>, 2026/01/29閲覧
- [17] ArduCAM/Arducam\_tof\_camera, [https://github.com/ArduCAM/Arducam\\_tof\\_camera](https://github.com/ArduCAM/Arducam_tof_camera), 2026/01/29閲覧
- [18] ToF 特集 ToF カメラとは? ToF カメラを使ってできること | inrevium <https://www.inrevium.com/pickup/tofcamera/>, 2026/01/29閲覧
- [19] Arducam ToF Camera SDK – for Raspberry Pi - Arducam Wiki <https://docs.arducam.com/Raspberry-Pi-Camera/Tof-camera/Arducam-ToF-Camera-SDK/>
- [20] note, 「LINE Bot とは何か?初心者向けにわかりやすく解説」, [https://note.com/bonjour\\_maman/n/n2a2ce62cde15](https://note.com/bonjour_maman/n/n2a2ce62cde15), 最終更新 2025.09.16
- [21] 総務省情報通信政策研究所, 「令和6年度情報通信メディアの利用時間と情報行動に関する調査報告書（概要）」p.12, [https://www.soumu.go.jp/main\\_content/001017240.pdf](https://www.soumu.go.jp/main_content/001017240.pdf), 最終更新 2025.06
- [22] FirstContact, 「Messaging API とは?～意味やできることを解説！～」, <https://first-contact.jp/blog/article/messaging-api/>, 最終更新 2025.03.05
- [23] Qiita, 「初心者向け解説：API とは？その仕組みと活用法を分かりやすく解説」, [https://qiita.com/UKI\\_datascience/items/18605ce56c7d9a4e4ca0](https://qiita.com/UKI_datascience/items/18605ce56c7d9a4e4ca0), 最終更新 2025.01.16
- [24] ミライサーバー, 「Flask とは？基本知識からインストール手順まで詳しく解説！」, [https://www.miraiserver.ne.jp/column/about\\_flask/](https://www.miraiserver.ne.jp/column/about_flask/), 最終更新 2025.08.18
- [25] blastengine, 「Webhook とは？仕組みやメリット、APIとの違い、利用方法について分かりやすく解説」, [https://blastengine.jp/blog\\_content/webhook/](https://blastengine.jp/blog_content/webhook/), 最終更新 2024.10.31
- [26] Qiita, 「Ngrok の使い方・実際に私が使っている事例を紹介」, <https://qiita.com/halapolo/items/a9d2345836b0302a264d>, 最終更新 2025.05.14
- [27] iifx.dev, 「Flask 開発サーバーをネットワーク公開する完全ガイド」, <https://iifx.dev/ja/articles/32624714/flask%E9%96%8B%E7%99%BA%E3%82%B5%E3%83%BC%E3%83%90%E3%83%BC%E3%82%92%E3%83%8D%E3%83%83%E3%83%88%E3%83%>

AF%E3%83%BC%E3%82%AF%E5%85%AC%E9%96%8B%E3%81%99%E3%82%8B%E5%AE%8C%E5%85%  
A8%E3%82%AC%E3%82%A4%E3%83%89, 最終更新 2025.07.19

- [28] LINE Developers, 「Messaging APIの料金」, <https://developers.line.biz/ja/docs/messaging-api/pricing/>
- [29] LINE Developers, 「リッチメニューの概要」, <https://developers.line.biz/ja/docs/messaging-api/rich-menus-overview/>

## 謝辞

本研究にご協力ならびにご助言をいただいた全ての皆様に深く感謝いたします。特に、小川隆申教授ならびに謝文昂助教には、終始熱心なご指導をいただきました。併せて、流体力学研究室の皆様にはご助言とご支援をいただきましたことに深く感謝いたします。また、本研究の実施に際し、センサやカメラの設置および電源の供給ラインの検討等ご協力いただきました成蹊大学財務部管財課の皆様に、厚く御礼申し上げます。ここに、本研究に関わってくださったすべての方々に感謝の意を表します。

# 付録A プログラム

## A.1 画像処理のソースコード

以下はソースコードである。OpenCVを用いた画像処理の在庫検出において、グレースケール変換を利用した検出プログラムをA.1に示す。

ソースコード A.1: グレースケール画像の検出プログラム

```
1 import cv2
2 import numpy as np
3 import csv
4
5 target_zones = []
6
7 # プログラム利用の際, "csv_data"は画像内商品のデータを入力したcsvファイルを用いる
8 with open("csv_data", newline='') as csvfile:
9     reader = csv.DictReader(csvfile)
10    for row in reader:
11        label = row['label']
12        color_B = int(row['color_B'])
13        color_G = int(row['color_G'])
14        color_R = int(row['color_R'])
15        x1 = int(row['x1'])
16        y1 = int(row['y1'])
17        x2 = int(row['x2'])
18        y2 = int(row['y2'])
19        target_zones.append({"label": label, "rect": (x1,y1,x2,y2), "color": (color_B,color_G,
19          color_R)})
20
21 # プログラム利用の際, "image"は実際の画像を入れる
22 img = cv2.imread('image')
23
24 # サイズ調整
25 scale_percent = 20
26 w = int(img.shape[1]*scale_percent/100)
27 h = int(img.shape[0]*scale_percent/100)
28 r = cv2.resize(img, (w, h), interpolation=cv2.INTER_AREA)
29
30
31 r_gray = cv2.cvtColor(r, cv2.COLOR_BGR2GRAY) # グレースケール変換
32
33 # 1番上の段とそれ以外の段でしきい値分ける
34 maskA = np.zeros_like(r_gray)
35 maskA[199:273, 49:602] = 255 # [y1:y2, x1,x2] 1番上の段
36 maskB = cv2.bitwise_not(maskA) # それ以外の段
```

```

37 binaryA = cv2.inRange(r_gray, 180, 255) # 1番上の段のしきい値
38 binaryB = cv2.inRange(r_gray, 100, 255) # それ以外の段のしきい値
39 mask_1 = cv2.bitwise_and(binaryA, maskA)
40 mask_2 = cv2.bitwise_and(binaryB, maskB)
41 mask = cv2.bitwise_or(mask_1, mask_2)
42
43 contours, _ =cv2.findContours(mask,
44                             cv2.RETR_EXTERNAL, # 外側の輪郭
45                             cv2.CHAIN_APPROX_SIMPLE) # 直線近似
46 r_color = r.copy()
47
48 for zone in target_zones: # 商品判定に用いる範囲の可視化
49     x1, y1, x2, y2 = zone['rect']
50     cv2.rectangle(r_color, (x1,y1), (x2,y2), zone['color'], 2)
51     cv2.putText(r_color, zone['label'], (x1,y1-5), cv2.FONT_HERSHEY_SIMPLEX, 0.5, zone['color'],
52                 1)
53 min_area = 40 # 面積が小さい輪郭指定
54 max_area = 3000 # 面積が大きい輪郭指定
55 target_counts = {zone['label']: 0 for zone in target_zones}
56
57 for i, contour in enumerate(contours): # 明度が高い部分（白い部分）の座標抽出
58     area = cv2.contourArea(contour) # 面積小さい輪郭削除
59     if area < min_area:
60         continue
61     if area > max_area:
62         continue
63     x, y, w, h = cv2.boundingRect(contour)
64     cx = x+(w/2)
65     cy = y+(h/2)
66
67 # 判定
68 judged_label = None
69 judged_color = (255,0,0)
70 for zone in target_zones:
71     x1, y1, x2, y2 = zone['rect']
72     if x1 <= cx <= x2 and y1 <= cy <= y2:
73         judged_label = zone['label']
74         judged_color = zone['color']
75         target_counts[judged_label] += 1
76         break
77
78 if judged_label is not None:
79     # 判定に用いたバウンディングボックス表示
80     cv2.rectangle(r_color, (x,y), (x+w, y+h),
81                   judged_color, # 描画の色(Blue, Green, Red)
82                   2 # 描画の線の太さ
83                   )
84
85     # バウンディングボックスの中心座標表示
86     cv2.circle(r_color, (int(cx),int(cy)),
87                 3, # 円の半径
88                 (0,255,255),
89                 -1 # 線の太さ(-1は塗りつぶし指定)

```

```

90         )
91
92     # ラベル表示
93     cv2.putText(r_color, judged_label, (x,y-5), # テキストの座標
94                 cv2.FONT_HERSHEY_SIMPLEX, # フォント
95                 0.5, # フォントサイズ(倍率)
96                 judged_color,
97                 1 # 線の太さ
98             )
99
100    # 商品の個数表示(ターミナル)
101   for label, count in target_counts.items():
102       number = f"{label}:{count}"
103       print(f"{label}:{count}個")
104
105   cv2.imshow('Original', r_color)
106   cv2.waitKey(0)
107   cv2.destroyAllWindows()

```

OpenCV を用いた画像処理の在庫検出において、HSV 変換を利用した検出プログラムを A.2 に示す。

ソースコード A.2: HSV 画像の検出プログラム

```

1 import cv2
2 import numpy as np
3 import csv
4
5 target_zones = []
6 # プログラム利用の際、"csv_data"は画像内商品のデータを入力したcsvファイルを用いる
7 with open("csv_data", newline='') as csvfile:
8     reader = csv.DictReader(csvfile)
9     for row in reader:
10         label = row['label']
11         color_B = int(row['color_B'])
12         color_G = int(row['color_G'])
13         color_R = int(row['color_R'])
14         x1 = int(row['x1'])
15         y1 = int(row['y1'])
16         x2 = int(row['x2'])
17         y2 = int(row['y2'])
18         h_min = int(row['h_min'])
19         h_max = int(row['h_max'])
20         s_min = int(row['s_min'])
21         s_max = int(row['s_max'])
22         v_min = int(row['v_min'])
23         v_max = int(row['v_max'])
24         target_zones.append({"label": label, "rect": (x1,y1,x2,y2), "color": (color_B,color_G,
25                                         color_R), "hsv":((h_min, s_min, v_min), (h_max, s_max, v_max))})
26
27 # プログラム利用の際、"image"は実際の画像を入れる
28 img = cv2.imread('image/kurisu_bakery.jpg')
29 # サイズ調整

```

```

30 | scale_percent = 20
31 | w = int(img.shape[1]*scale_percent/100)
32 | h = int(img.shape[0]*scale_percent/100)
33 | r = cv2.resize(img, (w, h), interpolation=cv2.INTER_AREA)
34 |
35 | r_color = cv2.cvtColor(r, cv2.COLOR_BGR2HSV) # カラー変換
36 |
37 | for zone in target_zones: # 商品判定に用いる範囲の可視化
38 |     x1, y1, x2, y2 = zone['rect']
39 |     # 各商品ごとにしきい値設定
40 |     lower, upper = np.array(zone['hsv'])
41 |     cv2.rectangle(r_color, (x1,y1), (x2,y2), zone['color'], 2)
42 |     cv2.putText(r_color, zone['label'], (x1,y1-5), cv2.FONT_HERSHEY_SIMPLEX, 0.5, zone['color'],
43 |                 1)
44 |
45 | mask = cv2.inRange(r_color, lower, upper) # 単体mask(閾値のトレードオフが起こらないとき)
46 | result = cv2.bitwise_and(r_color, r_color, mask=mask)
47 |
48 | contours, _ =cv2.findContours(mask,
49 |                                 cv2.RETR_EXTERNAL, # 外側の輪郭
50 |                                 cv2.CHAIN_APPROX_SIMPLE) # 直線近似
51 |
52 | min_area = 580 # 面積が小さい輪郭指定
53 | max_area = 3000 # 面積が大きい輪郭指定
54 | target_counts = {zone['label']: 0 for zone in target_zones}
55 |
56 | for i, contour in enumerate(contours): # 明度が高い部分(白い部分)の座標抽出
57 |     area = cv2.contourArea(contour) # 面積小さい輪郭削除
58 |     if area < min_area:
59 |         continue
60 |     if area > max_area:
61 |         continue
62 |     x, y, w, h = cv2.boundingRect(contour)
63 |     cx = x+(w/2)
64 |     cy = y+(h/2)
65 |     # print(f"領域{i}: center=({cx},{cy}), width={w}, height={h}")
66 |
67 |     # 判定
68 |     judged_label = None
69 |     judged_color = (255,0,0)
70 |     for zone in target_zones:
71 |         x1, y1, x2, y2 = zone['rect']
72 |         if x1 <= cx <= x2 and y1 <= cy <= y2:
73 |             judged_label = zone['label']
74 |             judged_color = zone['color']
75 |             target_counts[judged_label] += 1
76 |             break
77 |
78 |     if judged_label is not None:
79 |         # 判定に用いたバウンディングボックス表示
80 |         cv2.rectangle(r_color, (x,y), (x+w, y+h),
81 |                         judged_color, # 描画の色(Blue, Green, Red)
82 |                         2 # 描画の線の太さ
83 |                         )

```

```

83
84     # バウンディングボックスの中心座標表示
85     cv2.circle(r_color, (int(cx),int(cy)),
86                 3, # 円の半径
87                 (0,255,255),
88                 -1 # 線の太さ (-1は塗りつぶし指定)
89             )
90
91     # ラベル表示
92     cv2.putText(r_color, judged_label, (x,y-5), # テキストの座標
93                 cv2.FONT_HERSHEY_SIMPLEX, # フォント
94                 0.5, # フォントサイズ(倍率)
95                 judged_color,
96                 1 # 線の太さ
97             )
98
99 # 商品の個数表示(ターミナル)
100 for label, count in target_counts.items():
101     number = f"{label}:{count}"
102     print(f"{label}:{count}個")
103
104 cv2.imshow('Original', r_color)
105 cv2.waitKey(0)
106 cv2.destroyAllWindows()

```

## A.2 超音波センサのソースコード

超音波センサでの在庫把握において、マイコンボードで実行したプログラムがA.3である。

ソースコード A.3: マイコンボードで実行したプログラム

```

1 from machine import Pin
2 import machine
3 import utime
4
5 hasshin = Pin(14, Pin.OUT) #14番を出力(発信)と定義
6 jyushin = Pin(15, Pin.IN) #15番を入力(受信)と定義
7
8 def read_distance():
9     hasshin.low() #9行から13行までのコードで超音波を一瞬だけ発信
10    utime.sleep_us(2)
11    hasshin.high()
12    utime.sleep_us(10)
13    hasshin.low()
14    while jyushin.value() == 0: #超音波が受信されていないとき
15        start = utime.ticks_us() #現時点での稼働時間をマイクロ秒単位で返す
16    while jyushin.value() == 1: #超音波が受信されているとき
17        goal = utime.ticks_us() #現時点での稼働時間をマイクロ秒単位で返す
18    #超音波は一瞬なので、jyushin.value()==0となり、無限ループにはならない
19    passed = goal - start
20    distance = float((passed * 340 * 0.0001) / 2) #cmで出力
21    print(distance)

```

```

22
23     while True:
24         read_distance()
25         utime.sleep(1)

```

超音波センサから得られた数値を元に、パソコンで在庫を評価するプログラムがA.4である。

ソースコード A.4: パソコンで実行したプログラム

```

1 from datetime import datetime
2 import serial
3
4 ser = serial.Serial('COM3', 9600) #Windowsの場合
5
6 while(1):
7     dt_now = datetime.now()
8     distance = ser.readline().decode('utf-8').strip()
9     if float(distance) > float(21.0):
10         print(f"{dt_now}在庫なし{distance}")
11     elif float(distance) > float(10.0):
12         print(f"{dt_now}在庫あり{distance}")
13     else:
14         print(f"{dt_now}在庫大量{distance}")

```

### A.3 LINE Botのソースコード

LINE Botを用いた在庫データの共有について、Raspberry pi3からFlaskサーバへデータを送信するためのプログラムをA.5に示す。

ソースコード A.5: Raspberry pi3とFlaskサーバ間のデータ送信プログラム

```

1 import requests
2 import json
3 import time
4
5 # サーバのURL指定(IPアドレスは伏せている)
6 FLASK_SERVER_URL = "http://[サーバを立てたPCのIPアドレス(IPv4)]:5000/upload"
7
8 # データ送信準備
9 def send_data_to_server(value):
10     data = {"value": value}
11     try:
12         res = requests.post(FLASK_SERVER_URL, json=data)
13         print(f"[SEND] Sent {value}, Status: {res.status_code}")
14
15     # エラー処理
16     except Exception as e:
17         print(f"[ERROR] {e}")
18
19 # データ定期送信
20 if __name__ == "__main__":

```

```

21     while True:
22
23         # センサ値のダミーデータ（実用化の際は検出したセンサデータに切り替える）
24         fake_value = 5
25         send_data_to_server(fake_value)
26         time.sleep(10)

```

LINE Bot を用いた在庫データの共有について、公式アカウント利用者からの在庫に関するメッセージに対し、Flask サーバから LINE プラットフォームを通して返信を行うプログラムを A.6 に示す。

ソースコード A.6: 利用者への在庫通知プログラム

```

1  from flask import Flask, request, jsonify, abort
2  from linebot import LineBotApi, WebhookHandler
3  from linebot.models import MessageEvent, TextMessage, TextSendMessage
4  from datetime import datetime
5
6  app = Flask(__name__)
7
8  # チャネルアクセストークンおよびチャネルシークレットは伏せている
9  LINE_CHANNEL_ACCESS_TOKEN = ''
10 LINE_CHANNEL_SECRET = ''
11 line_bot_api = LineBotApi(LINE_CHANNEL_ACCESS_TOKEN)
12 handler = WebhookHandler(LINE_CHANNEL_SECRET)
13
14 # 最新データを保持する変数
15 latest_data = {"ブルーベリー＆クリームチーズベーグル": {"timestamp": None, "value": None}}
16
17 # Webhook受信
18 @app.route("/callback", methods=['POST'])
19 def callback():
20     # LINEからの改ざん防止用署名を取得
21     signature = request.headers['X-Line-Signature']
22
23     # イベント情報(JSON形式)を取得
24     body = request.get_data(as_text=True)
25     print("Received body:", body)
26
27     # 処理の受け渡し
28     try:
29         handler.handle(body, signature)
30
31     # エラー発生時の処理
32     except Exception as e:
33         print("[ERROR] LINE handler:", e)
34         abort(400)
35
36     return 'OK'
37
38 @handler.add(MessageEvent, message=TextMessage)
39 def handle_message(event):
40     text = event.message.text

```

```

41     if text == "「ブルーベリー＆クリームチーズベーグル」の在庫":
42         reply = f"現在の「ブルーベリー＆クリームチーズベーグル」は、{latest_data['value']}個で
43             す。"
44         line_bot_api.reply_message(event.reply_token, TextSendMessage(text=reply))
45     else:
46         return
47
48 # Raspberry pi3からデータ受信
49 @app.route("/upload", methods=['POST'])
50 def upload():
51
52     # JSON形式でデータ受信
53     try:
54         data = request.get_json()
55
56     # データの妥当性確認
57     if not data or "value" not in data:
58         return jsonify({"error": "InvalidJSON"}), 400
59
60     # データ取得
61     value = data["value"]
62
63     # 最新データに更新
64     latest_data["timestamp"] = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
65     latest_data["value"] = value
66
67     print(f"[UPLOAD] Received new data from device: {value}")
68
69     return jsonify({"status": "success", "received_value": value}), 200
70 except Exception as e:
71     print("[ERROR] Upload failed:", e)
72     return jsonify({"error": str(e)}), 500
73
74 @app.route("/status")
75 def status():
76     html = """
77     <!DOCTYPE html>
78     <html lang="ja">
79     <head>
80         <meta charset="UTF-8">
81         <title>最新データ</title>
82         <style>
83             body{
84                 font-family: "Segoe UI", sans-serif;
85                 background-color: #ffffff;
86                 color: #000000;
87                 padding: 50px;
88             }
89             h1{
90                 font-size: 28px;
91                 font-weight: 700; /* 太め */
92                 margin-bottom: 20px;
93             }

```

```
94 |         ul{{  
95 |             list-style-type:none;  
96 |             padding-left:10px;  
97 |             font-size:18px;  
98 |             font-weight:400; /*通常の太さ*/  
99 |             line-height:1.8;  
100|         }}  
101|     </style>  
102| </head>  
103| <body>  
104|     <h1>最新データ</h1>  
105|     <ul>  
106|         <li>商品名：ブルーベリー & クリームチーズベーグル</li>  
107|         <li>時間：{latest_data['timestamp']}108|         <li>数量：{latest_data['value']}109|     </ul>  
110| </body>  
111| </html>  
112| """  
113|     return html  
114|  
115| if __name__ == "__main__":  
116|     print("[SYSTEM] Flask server is running on port 5000...")  
117|     app.run(host="0.0.0.0", port=5000)
```

## **付録B 原稿非掲載データ**

### **B.1 実験結果**

例えばここに本文中に載せられなかった実験結果などを載せる。