

テクニカルドキュメント

こんぶ畑

2024年3月1日

1 チーム基本情報

1.1 チームコード・チーム名

WRS002 こんぶ畑

1.2 リーグ

Rescue Simulation (Erebus)

1.3 国・地域

岡山

2 チーム情報

2.1 チームについての説明

チーム名：こんぶ畑（こんぶばたけ）

メンバーの1人が所属していた部活の愛称と顧問への敬意を込めて名付けた。

リーダーは高1、高2とレスキューラインに参加してきた。今年度は受験の年で参加は半ば諦めていたが、幸運にも年内に志望校合格を果たし、もう1人を連れてくることでチームを結成することができた。

結成は2024年になってからだ。わずか3ヶ月で初めての競技に挑む。

2.1.1 情報公開

情報公開はロボカップジュニアの国はのようなもだ。

しかし、シミュレーションリーグでコードをオープンにすることは、実機リーグのそれとは持つ意味合いが少し違うと考える。良くも悪くも、たった1つのexeファイルが文字通り「全て」なのだから。だが、RCJJのコミュニティに何か少しでも貢献できることがあればと思い、今年もソースコードを大会終了後にGitHubで公開することにした。

リンク

実際にこのソースコードを公開することで直接誰かの参考になるとはあまり思っていない。それよりも、自分を含めより多くの人が情報公開をすることで、よりコミュニティが活発になることに期待している。

2.2 チームの所属団体

個人参加

2.3 チームメンバー

○○○○ (●●●●)

リーダーとしてプログラムの作成をした。

今年こそ最後の大会。悔いのないように頑張りたい。

□□□□ (■■■■)

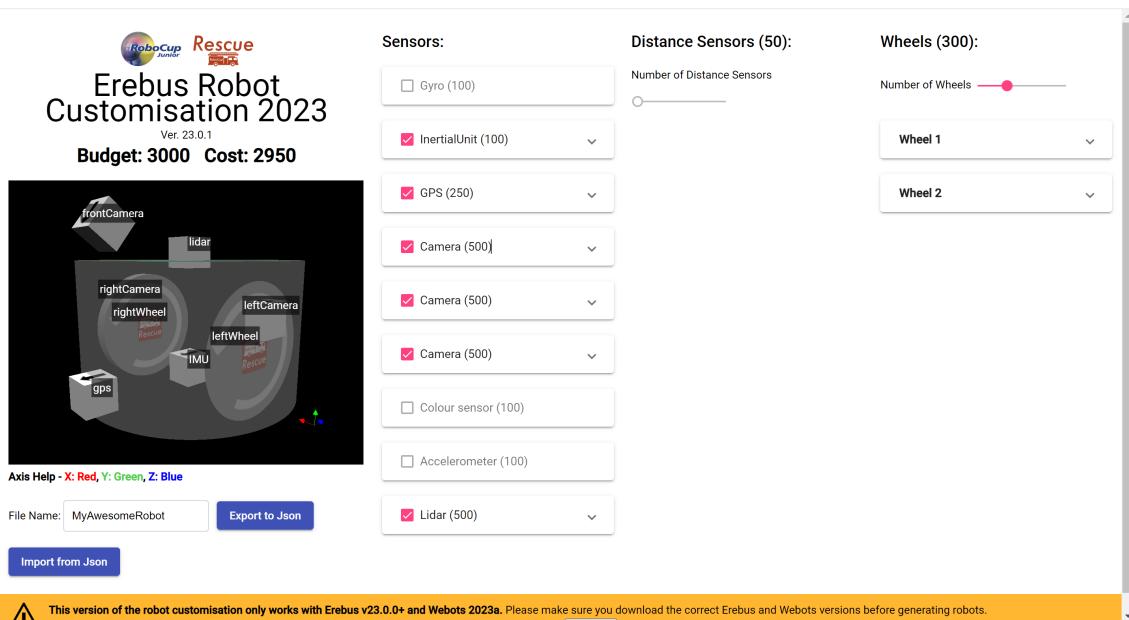
突然誘われた一般人

プログラム関連は無知

フィールド作成、ポスターのデザイン提案、修正を行った。

3 口ボットの構成

3.1 口ボットの写真



3.2 口ボット全般の説明

タイヤ・モータの数はそれぞれ2つ。Robot Customization の標準設定をそのまま使っている。

3.3 使用しているセンサーと使用目的

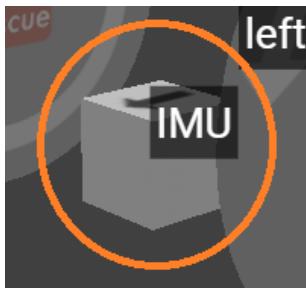
- 左右モータ + エンコーダ

標準のものをそのまま使用している。



- IMU

角度を取得するために使用している。



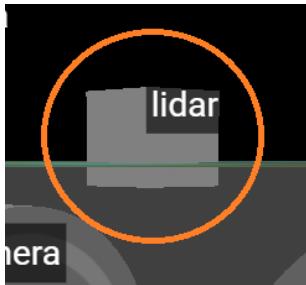
- GPS

位置情報を取得するために使用している。



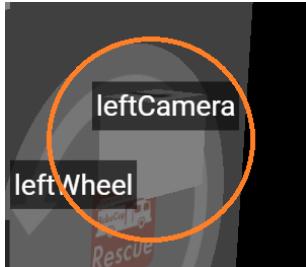
- LiDAR

障害物の検知、壁の認識に使用している。



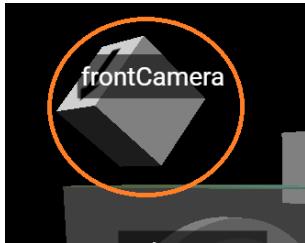
- 左右カメラ

被災者の検知に使用している。



- 正面カメラ

障害物を発見と床の色の認識に使用している。



3.4 被災者の探索

正面と左右に 1 つずつカメラを搭載している。

正面カメラ

斜め上から目の前のタイル 1 枚を俯瞰するように設置している。

Fov(視野角) を標準の 1rad から 1.5rad に広げて使用している。

側面カメラ

被災者に対して水平な高さから、被災者がちょうど収まるくらいの位置に置いている。

4 ロボットのソフトウェア

4.1 使用しているプログラミング言語・ソフトウェア

主な開発環境

- Webots 2023.a
- Erebus v23.0.5
- C++ 14 (メイン言語)
- Python 3.12.1
- Visual Studio 2022

使用した言語は C++ である。レスキューシミュレーションにおいては Python の方がポピュラーであることは何となしに知っていたが、C++ の方が好みなので C++ を採用した。

もちろん、画像認識の選択肢が狭まるることは承知の上である。

Visual Studio の環境構築には苦労した。なにせ、レスキューシミュレーション公式サイトにも Discord にも情報がないのである。結果的には、Webots の公式サイトの情報を基に作成した。このままではよくないと思い、簡単にだが環境構築方法をまとめたページを作成した。リンク

ソースコードの管理には GitHub を使用した。draw.io で作成したポスターと合わせて管理している。

ブランチ戦略には GitHub Flow を使用している。かつては git-flow を使用していたが、今回は小規模開発のためシンプルで細やかなリリースを重ねていける開発スタイルを選択した。

他にも、GitHub の Project 機能を使ったタスク管理を行うこともできた。

Backlog | Team capacity | Current iteration | Roadmap | My items | New view

Filter by keyword or by field

Todo 8 / 10 Estimate: 0 ...
This item hasn't been started

Kyomuri #11 Improve DFS L

Kyomuri #13 When To Stop? M

Kyomuri #12 Return To Start Tile L

Kyomuri #14 Optimize Map For Submission L

Kyomuri #15 + Add item

In Progress 1 / 5 Estimate: 0 ...
This is actively being worked on

Kyomuri #10 Improve Wall Recognition L

Done 0 Estimate: 0 ...
This has been completed

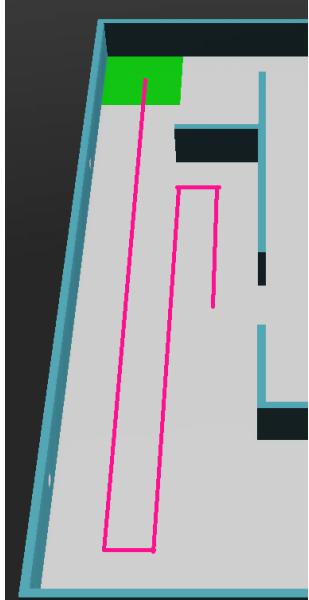
+ Add item

4.2 迷路探索

4.3 探索アルゴリズム

深さ優先探索を使用する。探索は 6cm 単位で行う。

オリジナルの工夫点も加えた。



上図のマップでは、ピンク線で示したように探索済みのマスを半分跨ぎながら進むという効率の悪い探索を行う可能性がある。これは機体のサイズは $12 \times 12\text{cm}$ を基準とするのに対して、 6cm 単位で探索を行うことが原因である。

そこで深さ優先探索における「未探索」「探索済み」という2つのパラメーターに「部分的に探索済み」を加え、進行方向の候補が複数あった場合に「部分的に探索済み」の優先度を下げることで、上図のような探索を防いでいる。

4.4 マッピング

vector 配列で隨時拡張しながら、提出するマップと同じ形式で記録する。

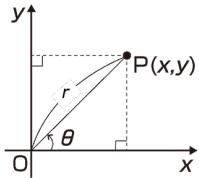
このマップは深さ優先探索や幅優先探索で最短経路を求めることにも使う。

4.5 LiDAR の活用

LiDAR ってなんか難しそう。でも使えればこれ以上の味方はいない。

4.5.1 値の補正

搭載されている LiDAR から取得できるデータは全体を 4 周できる分だけあり、それぞれ 512 個ずつ存在する。それぞれ Y 軸方向に対する向き（もしくは位置）の違いがあり、4 周のうち 3 周目が最も適切に距離関係を示していたためこれを使用している。



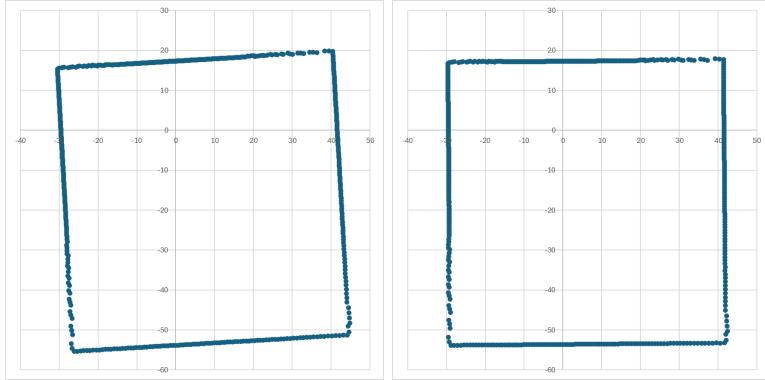
LiDAR のデータは極座標だと考えることができる。それを XZ の直交座標に変換して扱う。

また、以降において軸の表記がないグラフは縦軸を上側正の Z 軸、横軸を右側正の X 軸とする。

また、ロボットが毎回理想的な位置でデータを取得できるとは限らない。少なからず「角度」「位置」のズレが生じるはずだ。

ロボットから見た点群とフィールドの地形を同列に扱うには点群の補正が不可欠である。

まず、回転方向のズレを修正する。左の図が修正前、右の図が修正後だ。



この修正には回転行列を用いて計算する。

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \end{pmatrix}$$

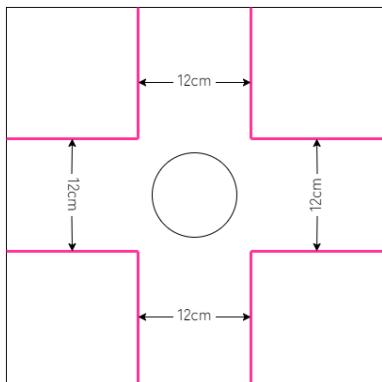
次に位置の補正をする。

理想的な位置と GPS から得られた実際の位置の差を計算し、LiDAR の点群の原点を理想的な位置に合わせる。

4.5.2 壁の認識

前後左右の隣のマスの壁の種類を判別する。

1. 調べる方向 12cm のデータを点群から切り取る



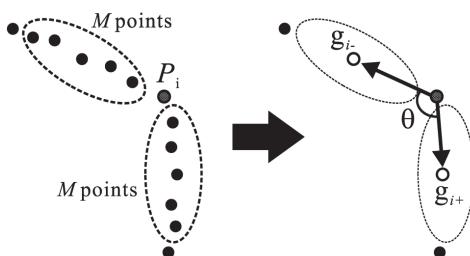
2. 点群を回転させる

このままでは切り取った点群は東西南北各々の方向を向いているので、それを機体から見て前の方向に回転させる。

こうすることでこの後の処理において条件分岐を少なくすることができる。

3. ベクトルトレーサー法を用いて特徴点を抽出する。

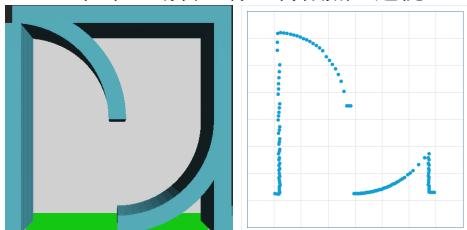
使用しているベクトルトレーサー法はこの文献を参考に作成した。



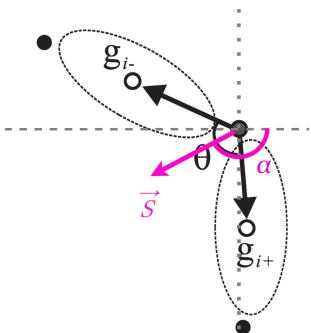
ある点 P_i の前後 2 点の重心 g_{i+}, g_{i-} を求め、これら 3 点のなす角 θ を内積を使って求める。

このままでは 1 つの特徴点に対して複数が抽出される場合がある。そこで連続して抽出された特徴点は θ が最大のものを代表値として扱う。

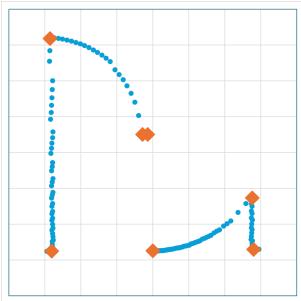
しかし、下の場合の様に特徴点が連続して抽出されることが正しい場合もある。



そこで P_i を原点とした g_{i+} ベクトルと g_{i-} ベクトルの和である s ベクトルが X 軸となす角 a を定義する。 a_n の値が a_{n-1} と大きく違う場合は別の特徴点とみなす。

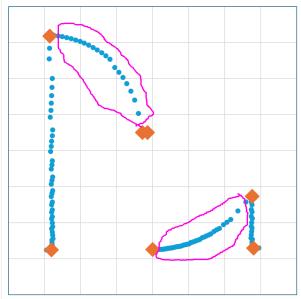


これまでのプロセスを経て抽出された特徴点が下の画像中のオレンジ色の点である。

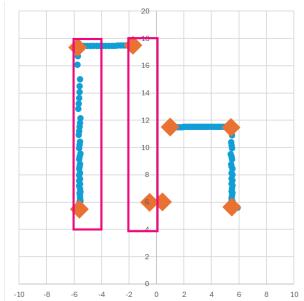


4. 判定する範囲の決定

この点群において壁の種類を判定するのに必要なのは円で囲った範囲である。これの左右の範囲の判定には先ほどどの特徴点を使う。



左側について、下の図の四角形で囲んだ領域内に特徴点があった場合、その中で最も端から離れたものを始点・終点として範囲を決定する。右側についても同様のことを行う。



5. 判定対象の有無

先ほど決定した範囲の最大値を求め、これが 18cm よりも大きい場合は判定対象外とする。

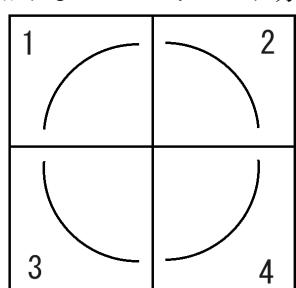
6. まっすぐな壁であるか

図 10 のようなまっすぐな壁であるかどうかを判定する。

そのために 4. で決定した範囲内の点の Z 座標の分散を計算する。まっすぐな壁であれば Z 軸の値はほぼ一様であり、分散は小さくなる。

7. 曲線の決定

6. において直線の壁でないと判定された場合は、壁は曲線であると考えられる。その曲線がどこを中心とするかによって 4 パターンに分けられる。



まず、右端と左端の Z 座標を比較すれば、”1,4”と”2,3”を分けることができる。

ここからさらに分類するために微分する。

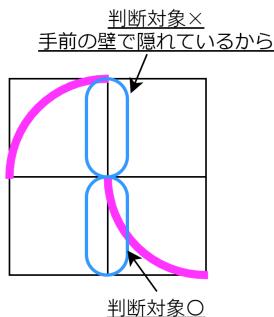
例えば、”1 or 4”と判定された対象の場合、壁の測定結果を微分すると”1”ならば減少関数、”4”ならば増加関数となる。これを右端と左端の傾きを比較することで実現する。



赤色→傾きの絶対値が大きい
青色→傾きの絶対値が小さい

8. 中央の壁

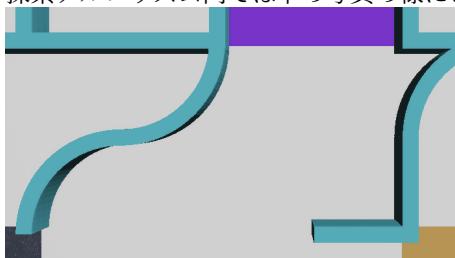
これまでの認識で、中央の壁に覆いかぶさるものがないと判断した場合にのみ中央の壁の認識をする。



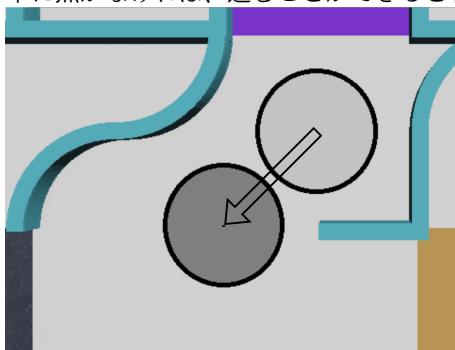
最も X 座標が 0 に近い点の Z 座標を元にその有無を判断する。

9. 斜めに進むという選択肢

探索アルゴリズム内では下の写真の様にどうしても斜め前に進まなければいけない場合がある。



斜めに進めるかどうかを判断するために、直径 7cm の円を仮想的に斜め前のタイルの中心に置く。そして円の中に点があれば、進むことができると判断する。

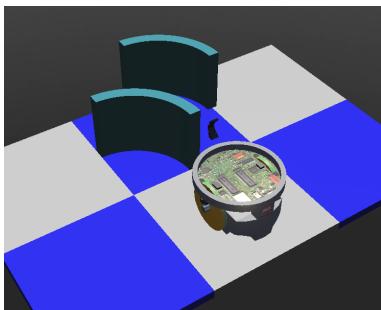


4.5.3 障害物の検知 with カメラ

LiDAR だけで障害物を探すのは無謀だ。サイズ、位置、あまりにもパターンが多すぎる。

しかし、カメラに視点を移せば障害物は明確な特徴を持っている。壁は青に寄った色をしているが、障害物は彩度 0 だ。これを利用して障害物を探す。

このマップを例に取ってみよう。

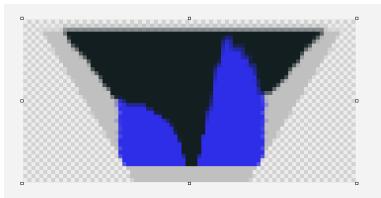


正面カメラを使って撮影されたのが下の写真だ。



しかしこれは鳥瞰画像ではないため扱いづらい。

そこで射影変換を用いて修正する。

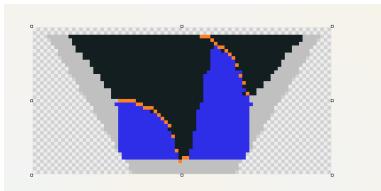


正面のタイルが正方形として映し出されるように修正した。

さらにどこまでが壁(障害物)なのかを判断するために LiDAR のデータを画像上に適用する。

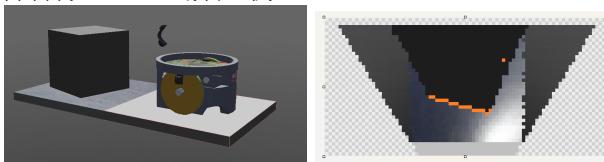
このような処理を挟むのは、壁と床の境界を判別することでチェックポイントタイルなどの「灰色っぽい床」と「灰色っぽい障害物」を混同しないためである。

下の画像のオレンジ色の点が LiADR の点である。



すると、この点よりも上にある部分は壁もしくは障害物であることが分かる。後は彩度が低ければ障害物、青よりの色であれば壁であると判断できる。

障害物があった場合の例：

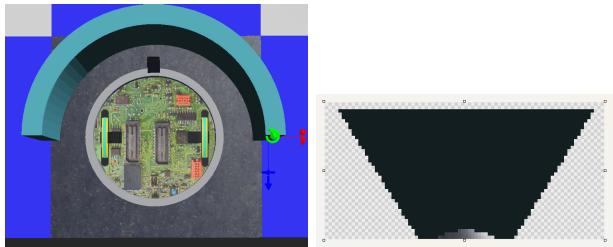


4.6 床の色の識別

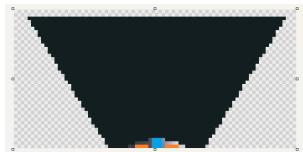
正面カメラの下側には自分のいるタイルが映っている。カメラを使うことで、機体が2枚のタイルを跨いでいるときでも左右それぞれの場合について判断することができる。

また、カメラに手前のマスも写っていることで、落とし穴に差し掛かることなくこれを判断することができる。

最も判断しづらいのは下のパターンだ。チェックポイントタイルの反射で床の色が一様でなく、さらに壁で見える床の面積も狭い。一番下でも 17 ピクセルしか判断材料がない。



この場合、水色で示した範囲はタイルの境界の可能性があるため使えない。そこでオレンジ色で示したそれぞれ 4 ピクセルを用いる。



多くのタイルは画角のどこに映っても色が変化することはない。よって HSV を用いて狭い閾値でも判断できる。

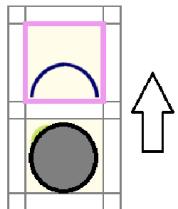
問題のチェックポイントタイルは彩度 0 かつ、明度の分散が大きいという条件で定義する。

4.7 落とし穴

「差し掛かる前に避ける」が基本方針である。

LiDAR によって進路を決定し、まずその進路を向く。この時障害物を検知を行い、それと同時に自分のいるタイルの色の判定、進む先のタイルの色の判定を行う。ここで壁の判定と一致しない障害物を発見した際には進路の修正を行う。

進む先のタイルの色の判定については「落とし穴 or それ以外」のみのスケールで行う。



進行方向として選択された以上、最低でも上図のスペースは担保されているはずである。これを利用して自分のいるタイルの時よりも余裕をもって隣のタイルの色を判定することができる。

左右について色の判定を行い、どちらか片方でも落とし穴と判定された場合はマップにそれを記録し、進路の訂正を行う。

5 被災者の発見

Python を手放したことによって便利な Yolo や SSD などの機械学習ライブラリが使えなくなってしまった。だが、一般的にはメイズ、シミュレーションを見渡しても機械学習が一般的なようだ。ここまでくると機械学習への反骨心が芽生えてきた。

5.1 事前処理

毎回毎回、被災者に関するすべての処理をしていると、制御周期が長くなりすぎてしまう。
そこで、「どんな種類かわからないけど、何らかの被災者 or ハザードマップ」を判断してから、本格的な分類に入る。

まず、前処理では LiDAR を使ってカメラの向いている方向に壁があるかを判断する。

次に、エッジ検出をして直線に近似し、そこから四角形、ひし形を探す。

これで四角形が見つかった場合はロボットの移動を停止して個別の判定に移る。

ここで一時停止することによって、被災者救出の条件である 1 秒停止を始めることができる。その間に判別してしまえば処理にかかる時間は気にしなくともよいということだ。

5.2 ハザードマップの F・O

これらは赤色、黄色と他にない色を持っている。

そのため黄色があれば確実に O、赤が有って且つ黄色があれば確実に F であると判断できる。

5.3 その他の被災者、ハザードマップ

SIFT という方法を使って特徴点を抽出する。そしてその抽出した特徴点を、事前に撮影したものの特徴点とマッチングして、つながった特徴点の数が最も多いものを報告する。

これは、まだまだ未完である。もし上手くいきそうになかったら OpenCV の機械学習 → PyTorch の順で試すつもりだ。

6 工夫した点・苦労した点・アピールポイント

6.1 読みやすいコードを

前回大会のコードの反省点、それは各マイコンのプログラムを 1 つのファイルに書いていたことだ。無造作に並ぶ千行近いコード。さよなら可読性。読みづらいったらありやしない。

この反省を基に、今回はファイルを複数に分割して作成した。

具体的には、各センサ・モータのクラスをそれぞれ含んだファイル、マッピングのファイル、DFS のファイル... といった具合でモジュール単位に分けている。

こうすることでカプセル化されたセンサやモータを DFS などで簡単に使うことができ、可読性・保守性も向上する。
また、書いたコードを「循環的複雑度」という指標を用いて定期的に評価した。

6.2 GPS トレース

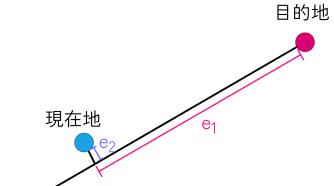
ライントレースではラインを基準に走り、メイズでは距離センサを基準に走ることが多いだろう。だがこのプログラムでは GPS を基準に走る。私はそれを GPS トレースと呼んでいる。

関数の主な仕様は

- 目的地の座標と、速度を引数として関数に渡す
- 目的地の方向を自動で向く
- 目的地までの距離と経路からのズレを偏差として PID 制御

目的地と現在地の座標を比較し、Z 軸方向との偏角が小さい場合は目的地の Z 座標を経路として走り、X 軸方向との偏角が小さい場合は目的地の X 座標を経路として走る。

斜め方向に目的地まで走るオプションもある。始点と目的地を結んだ直線をトレースするものである。経路からのズレ（偏差）を求めるには点と直線の距離公式から絶対値を除いた式を利用した。あえて絶対値を付けないことで現在地が直線の上下どちらにあるかを判別でき、そのままベクトル量の偏差として扱うことができる。



$$u_1(t) = K_{p1}e_1(t) + K_{i1} \int e_1(t)dt - K_{d1} \frac{de_1(t)}{dt}$$

$$u_2(t) = K_{p2}e_2(t) + K_{i2} \int e_2(t)dt - K_{d2} \frac{de_2(t)}{dt}$$

$$LeftSpeed = u_1(t)(speed + u_2(t))$$

$$RightSpeed = u_1(t)(speed - u_2(t))$$

6.3 壁マッチング

今回のプログラム内には採用されていないが、点群同士のマッチングをベースにした壁認識も開発した。

ここでは約 200 パターンの壁を予め測定、モデルとしてプログラムに組み込む。この際、特徴量の 1 つとして中心から見た左右の点の数も記録する。

この点の数は壁のパターンによって左右それぞれ 5 種類ほどに分類でき、これを基にフィルターをかけることでマッチングの対象となるモデルを元の約 200 個から 10 個程度に抑えることができる。

マッチングは中央から見た左右に分けて行う。左右それぞれでマッチングする点を 3 回ずらしながら累積 2 乗誤差を記録し、それが最も小さいものをそのモデルの「偏差」として記録する。そしてその偏差が最も小さかったモデルを採用する。

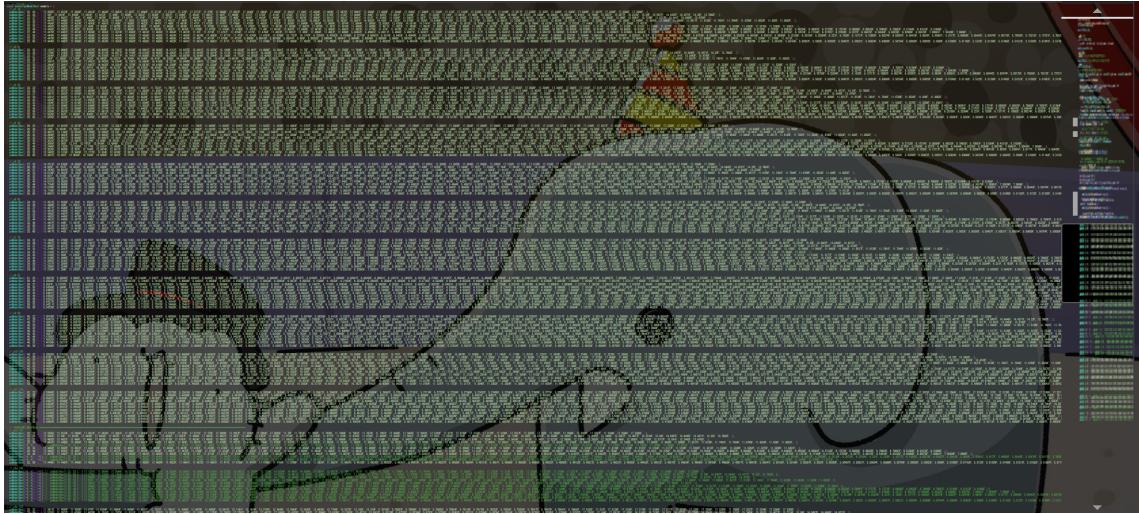
この手法の問題点として以下のものが挙げられた。

1. 機体の位置ずれによって点の数が増減した場合に対処できない。
2. 向きによって壁の生成パターンが違う。
3. タイルの中心でない (1/2 タイル進んだ状態) でも生成規則が違う。
4. 測定誤差による累積 2 乗偏差が壁の違いによるそれを上回る。

1. は ICP などで使用される最近傍探索の技術で解決できる。2.3. はそのパターンをコードに組み込んでしまえば解決できる上に、その微妙な違いによって本来見えない裏側の壁まで判定できる。

しかし、4. を解決できなかったため今回の採用は断念した。SLAM などで使用される点群と点群同士のマッチングは測定対象が一部一致する 2 つの比較であったのに対して、今回は多数の不正解から正解を探す比較である。前提条件が違ったのだ。

ちなみにモデルはこのように保存されている。



6.4 空間把握

これも現時点ではプログラム内には採用されていないものである。フィールド内で広い空間 ($2*3$ とか) に出てきた時、それを普通に回るのでは効率が悪いと考え、その空間サイズを把握できるような関数を書いた。

これまでの関数は「点」を基準に書いてきたが、それでは下の画像の青い線で示されたところを判断できない。



そこで「線」を基準に考える。ある意味正しい空間把握はできないが、最低限自分の周りの空間サイズまでは把握できる。

判定方法は「壁認識」の「9. 斜めに進むという選択肢」でも説明した直径 7cm の円を線にぶつかるかで動かすという方法。そして指定した範囲の線を切り取ってそこから距離を判断する方法がある。

ただ、そのサイズはロボットが動くラビに日々変化してしまう。それに対応するのが難しく使用できていない。

6.5 数学的視点

今年の大会において痛感したのが、数学の偉大さだ。分散・微分・三角関数などを使うという発想は高校数学をちゃんと履修した賜物だと考えている。

普通に生きている分にはその存在を感じさせない彼だが、現代社会を根底で支えていることを深く感じさせられた。今後も数学を学び続けるモチベーションになったように思う。