情報メディア実験 B レポート
ライントレーサロボットのフルスクラッチ実装

Kazushi Nakamura

2024 年 2 月 5 日

目次

第 1 章

はじめに

1.1　目的

本実験では、ライントレーサロボットの制作を通して、電気回路解析、制御工学に関する理論、および実際のソフトウェア、ハードウェアへの応用を学ぶことを目的とする。ライントレーサロボットは、周囲と明るさの異なるコース線（一般的には黒色または白色）を追従する非常に単純なロボットであり、最もシンプルな制御方法とキットを用いればロボット製作が初めての小学生でも容易に制作が可能なレベルである。しかし、フルスクラッチで設計し、高速で追従させようとすると必要な知識は広範にわたり、難易度は高くなる。本レポートでは、ロボット製作の過程での学習成果、および、実際のロボットの設計、製作方法、評価についてまとめる。

1.2　本レポートの構成

2 章では、電気回路解析の基本についての学習成果として、直流回路および交流回路の解析手法、過渡現象、よく用いられる素子系についてをまとめる。3 章は制御工学の基本についての学習成果をまとめる。2, 3 章の内容は一般的なものであるから、読む時間がない場合は飛ばして次の 4 章から読むことを推奨する。4 章では、実際に走行体に使用したハードウェア設計について部分に分割して示す。5 章では、ソフトウェア設計について、システム構成、制御手法および、部分ごとの実装について示す。これら設計に関する章ではこれから作ろうとする人に向けた解説という体で記述している。6 章では、走行体の評価、振り返り、改善点、感想等について示す。

第 2 章

電気回路解析の基本

2.1 概要

本章では電気回路解析の基本的な方法について学習したことを解説する。ロボットを作るときには、回路設計が必要不可欠である。正しい知識のもとに回路設計を行えば初歩的なミスによる部品の破壊が減らせるほか、トラブルが発生したときの原因究明にも回路解析の知識が役に立つ。もちろん、ここで扱う知識がすべて今回のロボット製作に直ちに役に立つ訳では無い。たとえば、電気回路解析のメインテーマは直流回路ではなく交流回路であるが、今回のロボット製作で交流回路を利用している人はいないだろう。しかし、直流回路であっても DC-DC コンバーダのように交流に似た理論で動く部品は存在するし、発振器を使用した回路も決して珍しくない。コンデンサはどうだろうか。直流回路にコンデンサは登場しないが、実際に扱う回路は理想的ではないから過渡現象が存在することを忘れてはいけない。そして、電源基板のような部分には過渡現象を抑えるためにコンデンサが使われているが、それを解析するためには交流回路の知識があったほうが便利である。一通りの知識を抑えておくことで直接的には関係ないように思われる場面で役に立つことは非常に多い。

本章では高校で学習するキルヒホッフの法則を紹介したあと、キルヒホッフの法則よりも楽に早く回路を解析するための手法を紹介し、キルヒホッフの法則からの脱却を図るというコンセプトで電気回路解析の基礎を解説する。

2.2 電気回路の導入

2.2.1 導入

それでは早速始めよう。まず、電気回路にはどのような構成要素があるだろうか。非常にシンプルである。基本的には、電源、配線、抵抗という 3 つの要素しかないと考えて問題ない。これは直流回路に限らず、交流回路の世界ではコンデンサ、インダクタといった素子が登場するものの、驚くことにこれらの素子は一定の手続きを踏めば抵抗と同じように考えて解くことができるのである。[*1]

次に、電気回路が解けるとはどのような状態であるだろうか。多くの場合、既知の素子を接続した回路上で任意の点の電流と、2 点間の電圧が求められている状態を指す。よって、我々は「電流」と「電圧」について

[*1] 注意点としては、電気回路の世界ではほとんどの場合「線形の回路」のみを扱う。線形な回路は線形な素子から構成される。少し難しく感じられるが、線形な素子であるかどうか大雑把に判定するな方法としてはオームの法則に従うかどうかを考えれば良い。豆電球のような非オーム抵抗は線形な素子ではない。その他にも、半導体素子であるトランジスタやダイオードなども線形な素子ではない。これらは電子回路の範囲で扱うものである。

正しい理解をする必要がある。

2.2.2 電流と電圧

まずは電圧から考えよう。電磁気学を学んだ人に説明するなら、「単位電荷に換算した静電気力のポテンシャルエネルギーである電位の 2 点間の差」「電場を積分して求められる電位の 2 点間の差」などと言えば十分であるが、実は、電磁気学を学んでいない人に電圧の概念を正しく理解してもらうのは意外と難しい。たとえば、中学理科を説明しているトライさんのサイト [?] では、「電圧のイメージは、流れる電子 1 粒のいきおいのこと」「電流 のイメージは、1 秒間に流れる電子の数のこと」とされている。この文章から電圧と電流の違いがわかるだろうか？ 理解している人にとってはこの文章は大きく外してはいないことがわかる。しかし、勢いが強いと言われれば、電子 (電荷) が高速で流れている状態をイメージしやすい。そして電荷のスピードというのはどちらかというと電圧ではなく電流の定義である。やはりよくわからなくなってしまった。

そこで、状況を説明するのに昔から用いられているのが水流のモデルである。これも正しく意味を吟味できていないと大きな誤解を招く諸刃の剣であるが、この考え方では次のように置き換えることができる[*2]。電源↔水を上に汲み上げるポンプ、配線↔水平な水路、抵抗↔水路中につけた水車、として閉じた水路を形成する。このようにおけば、ポンプで水を持ち上げる高さ h に比例して単位時間あたりに水路中のある位置を通過する水の量は増加する。もし、$h = 0$ であれば水が流れることはない。同じことが電気回路にも言える。電気回路では、高さにあたるのが電圧、単位時間に水路の断面を通過した水の量が電流、となる。注意点としては、「高さ」という量は常に 2 点間の差で決定されていて、1 点で絶対的に決定されるわけではないということである。たとえばポンプで水を 5 m 汲み上げたとき、その基準となっているのは持ち上げる前の位置である。逆に、ポンプで水を汲み上げたあとの高さを基準の高さ 0 m とすれば、持ち上げる前の高さは-5 m である。おなじことが電圧にも言えて、電圧が 1.5 V の電池は負極を基準とすれば正極は 1.5 V であるが、正極を基準とすれば、負極は-1.5 V である[*3]。電圧と聞くと水圧のように 1 点で決まるものであると誤解しやすい。そういった場合は、電磁気学では電圧と同じ意味で「電位差」という言葉がよく使われることを覚えておくとよいだろう。

次に電流について考えよう。水流モデルを考えれば、水流は (水流) = (通過水量)/(経過時間) で表現できる。同様に電流を表現すれば、

$$I(\text{電流}) = Q(\text{電荷の通過量})/t(\text{経過時間})$$

となる。これを拡張して微小時間でも同様の性質が成り立つと考えれば、

$$i(t) = \frac{dq(t)}{dt}$$

が成り立つ。

では抵抗はどのような装置であると考えられるか。抵抗はエネルギーを別の種類のエネルギーへと変換して取り出す装置である。水車は水の位置エネルギーを主に運動エネルギーへと変換する。同様に、電気回路中の抵抗は電気エネルギーを主に熱エネルギーへと変換する装置である。

[*2] 水流モデルには圧力に基づくモデルと高さの差に基づくモデルの二種があり、あまり明確に区別されずに用いられてきたことが文献 [?] で考察されている。

[*3] 回路図中では電圧の基準点は GND(グラウンド) と書かれることが多い。

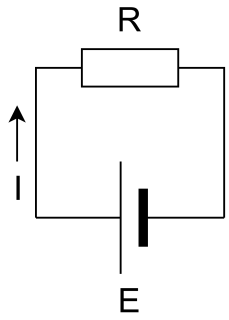

図 2.1

2.2.3 オームの法則

図 2.1 ような単純な回路を考える。水流モデルと同様に、電気回路でも与えた電圧 E に比例して電流 I は増加する。このとき、以下の式が成り立つ。

$$E = RI$$

この係数 R が抵抗である。同じ電圧ならば抵抗が大きいと電流は小さくなるため、抵抗は電流の流れにくさを表す。通常、R は時間によらず一定であるとみなして解析するが、実際の抵抗は温度変化による抵抗値の変動があるので注意する必要がある。R が一定であるとみなせるとき電圧と電流は比例し、このような関係は線形性とよばれる。電気回路の諸定理の中には線形性を満たすものにのみ適用できるものも多いから適用範囲に注意しなければならない。

また、抵抗の逆数 $G = 1/R$ を用いると、オームの法則は以下のように変形できる。

$$I = GE$$

この G をコンダクタンスとよび、電流の流れやすさを表す。

2.2.4 理想電源

理想電圧源

回路の状態によらず、常に負極と正極の間の電圧が一定であり、一切の内部抵抗がない電源装置を理想電圧源と呼ぶ。

実際の電源装置は理想電源ではない。たとえば、生活の中で身近な直流電源装置として、電池があげられる。電池は利用しているとだんだんと消耗し、両端の電圧が下がっていく。また、瞬間的に大きい電流が流れたときに両端の電圧が下がることがあり、やはり一定とは見なせない。そして、電池は内部抵抗が存在するため純粋に電源の機能だけとは見なせないのである。

理想電流源

回路の状態によらず、常に接続部分に一定の電流を流す電源装置を理想電流源と呼ぶ。

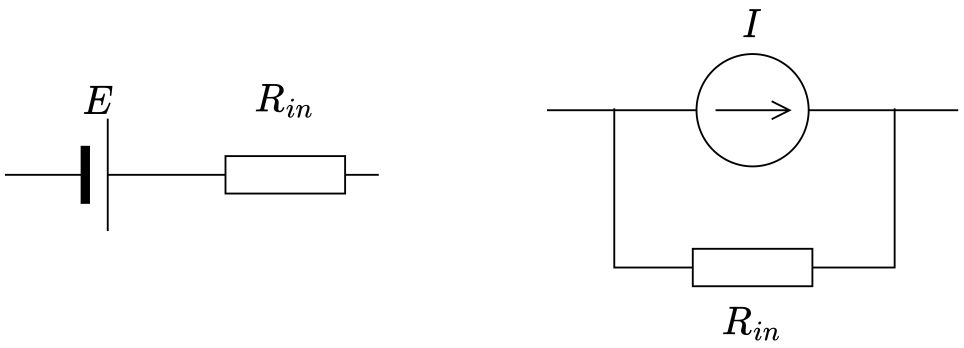

図 2.2　現実の電源の等価電圧源と等価電流源

電圧と電流の因果律を考えれば、電圧があることが原因で電流が生じるから、電流源はより現実の電源装置からは離れた概念である。しかし、後述するように電源の等価変換を考えることにより回路解析では有用な場面がある。

現実の電源の等価電源

先に述べたように電池のような現実の電源は理想電源ではない。しかし、内部抵抗を考慮することで、より現実に近い電源モデルを考えることができる。図 2.2 のように、電圧源の場合は理想電圧源 E と直列に抵抗 R_{in} を挿入し、電流源の場合は理想電流源 I と並列に抵抗 R_{in} を挿入することで、内部抵抗を考慮した電源モデルとなる。

2.2.5　電圧上昇と電圧降下

2.2.6　電力

2.3　直流回路の解析

2.3.1　キルヒホッフの法則

オームの法則により、ループが 1 つしかない回路については解くことができる。より一般的に、ループが複数の回路でも適用できる法則がキルヒホッフの法則である。

キルヒホッフの電圧則 (KVL)

回路中にある閉ループをとる。どのような経路を通ってきてもループを 1 周して戻ってきたときの電位は最初と同じである。

キルヒホッフの電流則 (KCL)

2.3.2 オイラーの公式

2.3.3 閉路解析法

2.3.4 節点解析法

2.4 正弦波交流回路の解析

2.5 過渡現象

2.6 回路素子の性質

第 3 章

古典制御の基本

3.1 概要

3.2 制御系の導入

3.3 ラプラス変換と伝達関数

3.4 PID 制御

3.5 安定性解析

第 4 章

ハードウェア設計

4.1　ハードウェア概観

4.2　部品表

4.3　ベース

4.4　センサ部

4.5　メイン基板

第 5 章

ソフトウェア設計

5.1 ソフトウェア概観

5.1.1 全体構成

本章ではソフトウェアの構成に必要な理論について説明する。今回構成した走行体には、光センサとモータのエンコーダという 2 つのセンサが搭載されていおり、それらの値をモータの制御に反映したフィードバック制御アルゴリズムを構築する必要がある。そこで本機ではモータの回転速度を制御量、PWM 制御の入力量を操作量とした PID 制御を用いることとした。ただし、一般的に PID 制御は 1 入力 1 出力の系であるから、光センサとエンコーダという 2 つのセンサ値のある系ではそのまま適用することができず、少し工夫が必要となる。ライントレーサの制御には例えば ET ロボコンについての記事 [?] が詳しい。これによれば、光センサの PID 制御値とエンコーダの PID 制御値をそれぞれ計算し、和を取ったものを直接操作量として PWM 制御に入力すればよいとされている。この手法は両方の制御がダイレクトに操作量に反映されるのでよいパラメータを設定できれば非常に応答性が高いプログラムになると思われる。しかし、6 つのパラメータを同時に調節する必要があり、走行の様子からどのパラメータをどの程度調節すればよいのかを見極めるのは初学者には難しい。そのため、今回はセンサ値の PID から各モータの目標速度を変更し、その速度に向かってモータ側の PID 制御を行うという 2 段の PID 制御を行う方式を最終的に採用した。この方式は、センサ値から直接操作量を変更しないで間にモータ側の PID 制御を挟むため、必ずしも最適な速度制御とはならない可能性、およびディレイがある可能性があることが欠点である。しかし、モータ側の PID パラメータの調整とセンサ側の調整を切り離し、1 度に 3 つのパラメータのみ対象として調整することができることがメリットである。この構成をブロック線図的に示すと、以下のようになる。

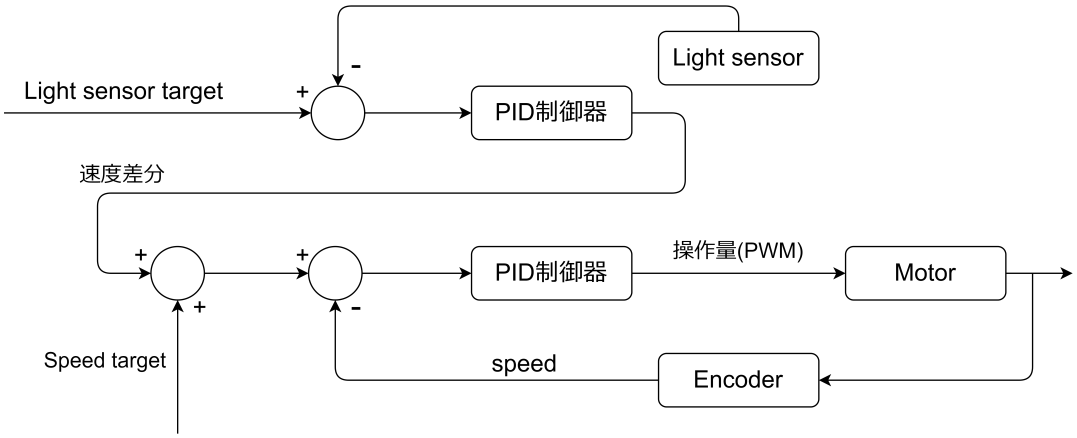

図 5.1　採用したシステムのブロック線図

PID 制御器が 2 つ直列に接続されていること、および速度の PID 制御の前で 3 つの値の加減算が行われていることが特徴的である。

5.2　光センサ値の検出

5.2.1　コース線の中心の推定

先に述べたように、本機は 8 つのセンサを積んでいるが、PID 制御に回すためには 1 つのスカラ値に変換する必要がある。センサ値の統合には様々な方法があるが、線が機体の外側に行くほど絶対値の大きい値を返すような単調な関数となっていることが望まれる。そこで、本機では 2 つのモータを結んだ線の中心から、コース線の中心と推定される位置の角度をセンサ値として利用する。ここで、コース線が左側にあるとき、負の値を返し、右側にあるときに正の値を返すとする。位置関係を示すと、図 5.2 のようになる。

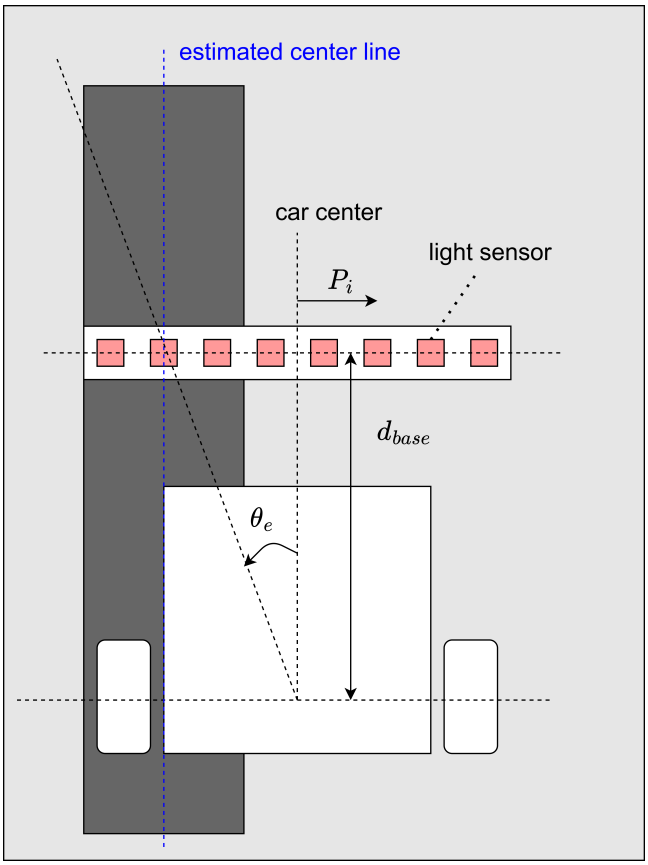

図 5.2　位置関係

まず、コース線の中心位置を推定する方法を考えよう。図の左側のセンサから順に 0 番から 7 番であるとする。車体の中心線 car_censor を基準として右側を正、左側を負とする座標系を考えると、各センサの位置は 0 番から順に-7, -5, -3, -1, 1, 3, 5, 7 [cm] にある。つまり、i 番目のセンサ位置 P_i は

$$P_i = -7 + 2i$$

である。次に、センサ i がコース線に乗っているか判定する関数を $b[i]$ とすれば、

$$b[i] = \begin{cases} 1 & (\text{センサ}i\text{がコース線上にある}) \\ 0 & (\text{センサ}i\text{がコース線上にない}) \end{cases}$$

$b[i]$ はセンサの実測値に対してしきい値を付けることで容易に実装することができる。

よって、これらを足し合わせ、“コース線上に乗っている” センサ数 N で割り、平均を取ることでコース線の推定位置 P_e は

$$P_e = \frac{1}{N} \sum_{n=0}^{7} b[i] P_i$$

5.2.2 角度の推定

次に、車軸から見込んだ推定角度 θ_e を求めよう。本機において、車軸とセンサのある位置までの距離 $d_{base} = 13$[cm] である。また、$\tan\theta_e = \frac{P_e}{d_{base}}$ である。ゆえに、

$$\theta_e = \tan^{-1}\frac{P_e}{d_{base}}$$

である。

三角関数は計算量が他の演算に対して大きいのでボトルネックとなることがある。その際には、$\tan^{-1}$ をテイラー展開して、適当な次数で切ることで冪関数として扱えば良い[*1]。今回の配置であれば、5 次近似程度あれば十分な精度である。しかし、実際に実験した結果、今回は特にボトルネックにならなかったので、検討はしたものの、特に必要はなかった。

5.2.3 読み込みのボトルネック

以上で 8 つのセンサを 1 つのスカラ値に変換することができた。この処理では 8 つのセンサを順次読みこむことから、ボトルネックとなり得ることに注意する必要がある。実は当初設計では採用したマイコンではピンが不足していたのでマルチプレクサを順次切り替えて処理する形式を取っていた。しかし、実際に試したところ、回路設計の関係なのか、1 度切り替えてから電圧が安定するまで 110 μs 程度要することがわかった[*2]。このディレイは毎回挟むと後述の速度検出のサンプリング周波数が不足するため解消する必要があった。そのため、本機ではマイコンを 2 台利用するように設計を変更し、光センサ値の検出に関係する部分をすべて移動した。遅延への懸念から、光センサ用マイコンから本マイコンへの通信には通信プロトコルを使用せず、適切なレンジにマッピングしたアナログ値 (電圧) を出力し、受け取り側で扱いやすい値に再マッピングすることで問題なく利用することができる。今回は受け取り側で [-10, 10] の範囲に再マッピングした。

5.3 速度の検出

5.3.1 エンコーダ

速度の検出にはモータに付いているエンコーダを活用する。エンコーダにはホールセンサが付いており、車輪を回転させると、1 回転で決まった回数 P だけデジタルでパルスが送られる。本機においては $P = 11$ である。つまり、信号を $f(t)$ とし、モータを定速で回転させたならば $f(t) = 1$ (HIGH) となる時間と、$f(t) = 0$ (LOW) となる時間を周期的に繰り返す。通常モータは正転と逆転を判別するために位相の異なるホールセンサが 2 つ搭載されている。しかし、今回はモータは逆転しないものとし、1 つのエンコーダにのみ着目する。

[*1] $\arctan x$ のテイラー展開は 2 次近似まで x であるから、大きさの正規化さえ気をつければ P_e を直接角度の推定値として扱ってもあまり問題はない。今回は精度を優先した。

[*2] マルチプレクサ側の問題ではない。モーターのエンコーダも同様にマルチプレクサを使っていたが、3 μs 程度のディレイで問題なく動作した。

5.3.2 センサ値のサンプリング

デジタル値が流れている生データから速さを取得するためには、信号の値の変化を捉えてその時間間隔を調べればよい。1 つのエンコーダで捉えられる値の変化は HIGH to LOW または LOW to HIGH の 2 種類である。仮に両方の変化を捉えるとすれば、1 回転で $2P$ 回の変化を捉えることとなる。一般に、より多くの変化を捉えるほうが精度のよい速さが求められるが、サンプリング周波数に注意する必要がある。プログラム中で速度を毎ループ計算せずに非同期で処理する場合、1 ループにかかる時間の逆数がサンプリング周波数である。そして、1 ループで 1 回の変化が起こる状態 (毎回のループで HIGH -> LOW -> HIGH と変化している状態) が取得しうる最高速度であり、それ以上の速度は測定できない。しかし、そもそもナイキスト定理を考えれば、この状態はすでにエイリアシングが発生して精度が低下しているから元の信号の周波数は更に低くなければいけない。さらに言えば、ナイキスト定理では 2 倍のサンプリング周波数があれば良いことになっているが、波形を復元する特別な工夫がないなら 10 倍以上の周波数を使うべきである。フーリエ変換を考えて厳密に言えば、デジタル信号は矩形波であるから、無限大の周波数がなければ修復できない。そのため、どのようなサンプリング周波数で計測しても、たまに飛び値が出る。しかし、モータは瞬間的に追従することはできないうえ、次の瞬間には正しい値に戻っていることを考えれば飛び値が出ることは問題にならない。そして、LOW->HIGH->LOW をあたかも 1 つの波であるかのように捉えてその周波数を考えてみればこの計測法でも比較的正しい速度が計算できることがわかる。こちらで考えても周波数が高すぎるときは、変化の読み落としが発生しているので全く信用にならない。本機では LOW to HIGH のみを捉えることとする。

ここまでは、非同期で処理する場合を考えた。では同期処理ではどのような問題があるだろうか。まず、同期処理では毎回の処理でそのときの速度を定める。つまり、毎回速度が確定するまでこの処理のみを実行するので、読み落としは発生せず、エイリアシングの問題等は一切考える必要がない。ここだけ聞くと、同期処理のほうがよく聞こえるかもしれないが、問題はこの処理が終わる (= エンコーダの変化を 2 点で捉える) まで次の処理に進めないことである。高速走行時には短い時間で変化を捉えられるので大きな問題にはならないかもしれないが、低速時には待ち時間が増えるし、停止時には停止しているかどうかを明確に判別できるだけの最大待ち時間待たなければならなくなる。その時間はモータに依存するが本機においては最低でも 20ms であった。つまり、1 ループに最大 20ms、高速走行時でも非同期処理よりは当然遅く、数 ms の待ち時間が発生する。待ち時間とはつまり制御が入らずに空走する時間である。例えば、50cm/s で走行中に 5ms の待ち時間があったとすると、空走距離は 50cm/s $\times$5ms $=$ 0.25cm である。僅かな量に感じるかもしれないが、これだけの空走距離があると急なカーブ等には対応することが困難である。つまり、同期処理で速度を取得すると非常に大きなボトルネックになるので、速度の計算は非同期処理にする必要がある。

5.3.3 生データから速さへの変換

前節でモータ 1 回転に捉える変化数を決めた。改めて P とおく。つまり、本機においては $P = 11$ である。次に、値の変化にかかった時間を Δt [μs] とする。モータのギア比 (減速比) を R とする。タイヤの直径を D [cm] とすれば、タイヤ外周部の速度 v [cm/s] は次のように表される。

$$v = \frac{\pi D \times 10^6}{RP\Delta t}$$

これは実質的に時間の 1 変数関数であり、残りは機体に固有の定数であるから、切り分けて前処理に回すことで処理を高速化できることに留意したい。また、車体が停止しているとき、Δt は永遠に求まらないため、同

期処理と同様に一定時間経過したら速度を 0 と判定するしきい値時間を用意する必要があることに注意する。

5.3.4 非同期処理の実装

本機は Arduino 互換ボードを使用しており、ソフトウェアの実装言語は Arduino 言語である。そのため、Python 等高機能言語に実装されている async/await 等の高級な非同期処理インターフェースはない。そのような状況でシンプルに非同期処理を実装するためには、時間を確認することが有効である。Aruduino 言語には、経過ミリ秒を表示する millis() やマイクロ秒を表示する micros() が存在するため、それを用いることで素朴な非同期処理を実装することができる。

その他の選択肢としては、今回は採用していないが割り込み処理にする方法が考えられる。ボード依存であるが、Arduino は割り込み処理をサポートしている。ホールセンサの信号の変化をトリガとした割り込み処理をすればより無駄のない速度計測ができる可能性がある。

5.3.5 ノイズ対策

エンコーダでは、ホールセンサの信号を HIGH と LOW で送信する。回転の途中で値の境界付近の状況にあるとき、短い周期で信号が震えることがある。これはスイッチにおけるチャタリングに近い現象であり、同様に対策が必要となる。本機では、信号値を取得する際に毎ループで続けて 3 回取得し、すべての信号値が揃わなければ信用できないとして信号を取得し直す、というロジックによりこの問題へ対処した。

5.4 モータ制御

5.4.1 センサの PID 制御

これまででそれぞれのセンサ値を取ることができた。次にこれをモータの動作に落とし込む必要がある。改めて、PID 制御を時間の式で示せば、

$$y = K_p e + K_i \int e\, dt + K_d \frac{\mathrm{d}e}{\mathrm{d}t}$$

ただし e は目標値から現在値を引いた偏差である。これはアナログ値であるから、デジタル値による記法に変換する必要がある。これにはいくつかの流儀があるが、今回は最も基本的な方式として、微小時間は無視する、積分要素は面積を取らずに現在差分を単純に足す、微分要素は差分要素に置き換えるという方法を取った。

今一度 PID 制御器の結果がどのように使われるか確認すると、光センサについての PID の結果は曲がる方向に応じてモータの目標速度を変化させる役目をもつ。モータについての PID の結果はもちろん所望の目標速度に近づくようにモータへ送る操作量を調整する。図 5.1 にブロック線図を示してあるから、あと必要なことはそのまま実装して、パラメータの調節を行うことのみ (これが一番大変かもしれない！) である。

5.4.2 光センサにおける PID 調整前のパラメータ推定

PID 制御のパラメータ調節は地道な作業であるが、論理的に考えられる部分については、ある程度あたりをつけてから始めると楽である。特に、PID 制御の調節は P 制御から始め、K_p の大きさによって他の値のレンジの決定されるので、K_p の正当性が最も重要である。ここでは、光センサの P 制御値について考察を行うことでパラメータにあたりをつけてみよう。

ライントレースを行ううえで一番走行が難しいのは急カーブである。逆にいえば、急カーブを走行してもコースアウトせずに即座に車体の安定を取り戻せる状態なら他の場所は問題にならない。では K_p をいったいどのくらいの大きさにすれば最も急なカーブを曲がり切れるだろうか？

最も急なカーブの曲率半径を r[cm] とする。車体の中心線からタイヤの中心までの距離を d[cm] とする。与えた走行速度目標を v_t[cm/s] とする。ここで、走行速度目標とはカーブに差し掛かって光センサ値による補正がかかった値ではなく、光センサ値によらない (つまり、直進中の値とも言える) である。ある瞬間、走行体は角速度 $\dot{\theta}$[rad/s] で円運動をしているとしよう。このとき、機体の制御により、外側の車輪の目標速度は $v_t + \Delta v$、内側の車輪の速度は $v_t - \Delta v$ に変更される。ただし、Δv は K_p に依存する値である。このとき、速度に関する次の 2 つの式が成り立つ。

$$
\begin{aligned}
v_t + \Delta v &= (r + d)\dot{\theta} \\
v_t - \Delta v &= (r - d)\dot{\theta}
\end{aligned}
$$

ここから $\dot{\theta}$ を削除することにより、

$$
\begin{aligned}
\frac{v_t + \Delta v}{v_t - \Delta v} &= \frac{r + d}{r - d} \\
&= k
\end{aligned}
$$

とおくと、k は r と d によって決まるからコースと車体のプラットフォームが変わらない限り定数である。この k を用いると、

$$
\Delta v = \frac{k - 1}{k + 1} v_t
$$

と表される。つまり、光センサの PID 制御全体で Δv より大きい速度の変化分を出せないと曲がりきれないことが実験しなくてもわかる。

本機体について考えてみよう。本機体において、$d = 7.5$[cm] 速度 $v_t = 100$[cm/s] で、曲率半径 $r = 10$[cm] の円を曲がるとする。このとき、$k = 7$ であるから、$\Delta v = \frac{6}{8} v_t = \frac{3}{4} \times 100 = 75$ である。いま、光センサ値が [-10, 10] の範囲でマッピングされている。現在の光センサの値を l とする。光センサの目標値は中心である 0 とすれば $e = l$。ここでの偏差は定義とは反対であるが、その後のデータの与え方を反転させれば問題ない。絶対値で扱っていると考えても良い。P 制御しか行わない状態を仮定すると、$\Delta v = K_p |l| > 75$ となる。光センサが最大値のときにぎりぎり曲がりきれると仮定すると、$l = \pm 10$ で、$K_p > 7.5$ となる。実際にはもっと余裕を持ちたいのでより大きな値となると推定される。たとえば、$l = 8$ で曲がりきりたいとすると、$K_p > 9.38$ 程度となる。これと D 制御を組み合わせることで、急なカーブでもより安定して走行することができる。

実際に採用した値が $K_p = 10.0, K_d = 12.5, K_i = 0.0002$ であることを考えるとこの考察は非常に有益であることがわかる。

5.4.3 アドホックな調整

ここまでで、カーブを含めたほとんどの場合において安定した走行をするための議論は完了した。しかし、まだ特殊な場合で安定しない可能性が残されている。本節ではそのような特定の状況に対応するアドホックな処理について議論する。

安定しない可能性があるのは速度 0 からの立ち上がりの状況である。理由は 2 つある。

1 つ目は、目標速度を大きくすればするほど、初めの P 制御が強く効くからである。理想的状況であれば、速やかに目標速度に到達するため問題ない。しかし、左右のモータの回りやすさに差がある場合、初めの制御量が大きく入ると左右差が大きく出やすく、場合によっては修正する間もなくコースアウトしてしまう。

これに対応するためには立ち上がりの偏差に制限をかければよい。つまり、両輪が低速時に限り、現在速度から逆算したターゲット速度を与えることで実現される。これは、数式モデルとしては異なるが、一次遅れ要素に似た性質をもつ。つまり、以下のようにすればよい。

```
DelayAccelerationSpeed(targetSpeed, maxError, leftCurrentSpeed, RightCurrentSpeed)
  if leftCurrentSpeed < threshold and rightCurrentSpeed < threshold
    newTargetL = min(leftCurrentSpeed + maxError, targetSpeed)
    newTargetR = min(RightCurrentSpeed + maxError, targetSpeed)
    return newTargetL, newTargetR
  else
    return TargetSpeed, TargetSpeed
```

副次的効果として、偏差の最大値が部分的ではあるが制限されることになるため、モータ側の PID 制御の K_p をより大きくしても問題が起こりにくくなる。すべての場合で偏差の最大値を制限すれば偏差が maxError 以下であることが保証されるのでそれを念頭に置いた設計ができるが、急カーブを曲がりにくくなるという難点があるため推奨しない。

安定しないもう 1 つの理由は、センサの PID 制御値の速度依存性である。5.4.2 節で速度に応じて与えるべきセンサの K_p が変化することを述べた。もちろん K_p は目標速度に合わせて作られているから、立ち上がりの速度ではいささか強く効きすぎる。そして、先に述べたように、モータには個体差があるから同じように操作量を与えてもまっすぐに進み出すわけではない。その結果、低速でラインを跨ごうとし、一方の車輪が完全に止まろうとする、逆向きに曲がろうとしてもう一方の車輪が完全に止まろうとする、という動作を繰り返して、振動的な挙動をしながら加速する。このような挙動ではスムーズな立ち上がりとは言えないから、対処したほうがよい。

これに対応する解決策も単純であり、両輪が低速であるときにはセンサの PID の係数を書き換えれば良い。もちろんなめらかに変化させることもできるが、低速用の係数を用意して、書き換えるか、書き換えないかという 2 段階があれば十分であると思われる。本機の場合は、50cm/s にチューニングした PID 値を用意して二段階にしておけば十分安定した走り出しとなった。

5.4.4 Xbee との通信

低速で走行実験を行う場合、万が一コースアウトしても手で止めに行けば問題にならない。しかし、100cm/s といった高速で走行する場合、手で止めると機体にダメージが入ったり、手で止める前に壁に衝突する危険性がある。走行会の前に事故によって機体が破損するといった事態は絶対に避けなければならない。これを避けるため、遠隔で発進と緊急停止を行うことができる機能があるとよい。やり方は様々あると考えられるが、本機では Xbee を搭載して遠隔シリアル通信による発進と停止の機能を実装した。

5.5 ソースコード

長くなるので、GitHub のリポジトリに上げてある。

リンクは `https://github.com/RiceCake1/LineFollow/tree/main`.

この節では、実装を関数別に簡単に解説する。なお、Arduino ではデフォルトで setup 関数と loop 関数が必要である。setup 関数は起動時に 1 度だけ呼ばれる関数であり、ピンの設定や、初期化等を書く場所である。loop 関数は setup 関数のあとに常に呼ばれる関数であり、メインループとなっている。

5.5.1 光センサ関係 (光センサ用ボード側)

float ReadSensorBinary(int sensorNum)

センサがコース線に乗っているか判定する関数。

float ReadSensor(int sensorNum)

アナログ値を読み [0,1] にマップして返す関数。デバッグ用。

int getLinePos()

以前使われていた、センサ値の統合アルゴリズム v1。

float getLinePosThetaWeighted()

以前使われていた、センサ値の統合アルゴリズム v2。角度を考慮し始めた。試行錯誤の痕跡。

float getThetaWithArctan()

最終的に採用された、センサ値の統合アルゴリズム v3。解説した通り単調な関数となっていて、非常に安定性が高い。これに変更してからカーブ後の振動の収束が有意に早くなった。

5.5.2 光センサ関係 (メインボード側)

float getLinePos()

メインボード側で光センサ処理用マイコンから送られてきた値を読み、[-10.0, 10.0] の範囲に再マップする関数。

float getPIDsensorValue(int linePos)

光センサの PID 制御値を計算し、返す関数。低速時に係数を変更するアドホック処理も含まれている。

5.5.3 エンコーダ関係

int _smoothEncoderSignal(int encoderNum)

ノイズ対策として 3 回エンコーダの信号を読み、すべて一致するまで繰り返す内部関数。

bool _isChanged(int encoderNum)
信号が変化したかを判定する内部関数。

float _getSpeedWithTime(int motorNum)
1 つのモータについて非同期処理で速度を取得する内部関数。

void getSpeedWithTime(float *spd)
上の関数を用いて両輪の処理をまとめた関数。

5.5.4 モータの制御関係

void controlMotor(float Rspeed, float Lspeed)
[-1,1] の値の 2 引数を取り、両輪の PWM 制御を簡単に行う関数。負の値ならば後退、正の値ならば前進。本番環境では内部関数だが、デバッグ用途ではそのまま使える。

void delayAccelerationSpeed(float targetSpeed, float *newTargetSpeed, float maxError = 10)
アドホック処理のための関数。立ち上がりの速度偏差に制限をかける。

void controlEachMotorWithPID(float rightTargetSpeed, float leftTargetSpeed)
左右モータの目標速度をそれぞれ引数に取り、PID 制御値を計算して、controlMotor でモータを制御する。

float limitSpeedOnCorner(float targetSpeed, float deacceleratedSpeed, int linePositon)
終盤に作られた失敗作。コーナーでは両輪とも減速すれば平常時の目標速度が速くても曲がり切れるのではないだろうか、という仮説に基づいて作られたアドホック制御関数。実際には、この関数が働き始めるときにはすでにブレーキには遅すぎる地点であることがわかった。消されたものも多いが、このプロジェクトにはこのような不採用になった関数が山程存在し、その数だけ試行錯誤があった。参考までに、最終形である現在ではメインボードのコードは 280 行程度であるが、一時期は 500 行以上あった (1 月 20 日のコミット)。

void controlMotorFromSensors(float targetSpeed, float deacceleratedSpeed=10)
最終的に採用されたすべてが集約された関数。センサ値の取得、光センサの PID 制御値の計算、モータの PID 値の計算、モータへの操作量の入力までがすべてこの関数で繋がった。

5.5.5 その他

float mapFloat(float InputValue, float InputLower, float InputUpper, float OutputLower, float OutputUpper)
ある範囲内の実数を別の範囲にマッピングするための関数。

show...()
show 関数系は名前が示すものをシリアルで表示するデバッグ用関数。

initAll()

名前の通り、変数の初期化等をまとめている関数。setup で実行するほか、Xbee で停止させていた走行体を再び走行させるときに実行される。

Xbee()

無線通信を行い、ソフトウェア的に停止、発進を行うための関数。高速でのデバッグでは壁に衝突するなど事故が乱発していたため、必ず機体を緊急停止できる人員を配置し、その人の許可[*3]なく発進させてはいけないという管制塔システムを導入した。その結果安全性が向上し、機体を破壊することなく無事本番を迎えられた。

[*3] 電源を入れると発進する状態のときに “Cleared for run” の合図。後に電源を入れても自動で発進せず、管制管が発進信号を送るように変更された。

第 6 章

評価

6.1 ループ速度

6.2 実測記録

6.3 感想

付録 A

応用的な電気回路解析

A.1　4 端子回路

A.2　ひずみ波交流の解析

A.3　三相交流

A.4　分布定数回路

付録 B

設計の変遷

本付録では、最終成果物に至るまでの設計の変遷を示す。

B.1 初号機

B.2 センサ値の統合

付録 C

インシデント、問題解決集