Tutorial 3

システム創造プロジェクト TA: 田所祐一

2018 年 10 月 15 日（月）

1 概要

今回の Tutorial の目標は，前回までで製作した機体を使って，地面に引かれた線に沿って走行させるライントレースプログラムを作成することである．ライントレースは，今後の競技の基礎となる最も重要な要素のひとつなので，確実に理解することが望ましい．

今回扱う内容は以下のとおりである．

- AD 変換 (`analogRead`)
- フォトリフレクタによるライン検出
- ライントレースプログラムの作成
- 中間試技会に向けたアルゴリズムの改良

2 AD 変換

Arduino に搭載された AVR マイコンには AD 変換器が搭載されており，ピン A0〜A5 の合計 6 チャンネルの入力が可能となっている．分解能は 10bit であり，0V から 5V までの値を 10bit に離散化した 0〜1023 までの値が得られる．Arduino では，`analogRead` 関数を使用することで，AD 変換を行い，値を取得することができる．

- `analogRead(pin)`
  指定したピンについて AD 変換を行い，アナログ電圧に対応する値を取得する．取得できる値は 0–1023 の範囲であり，この値は 0–5[V] に対応している．

AD 変換で得られた値 k から電圧 V を計算するには，

$$V \approx \frac{5 \cdot k}{1024}$$

という関係式を用いればよい．また，$1024 \approx 1000$ と考えれば，

$$1000V \approx 5 \cdot k$$

として，計測した電圧のミリボルト単位での値を整数で扱うことができる．

⚠ 注意	AD 変換器の分解能が 10bit であるということは，10bit の精度で計測できるということを意味しないということに注意する．AD 変換値の精度は，電源回路の性能やノイズに大きく左右される．

⚠ 警告	浮動小数点演算により AD 変換の結果を電圧値に変換してもよいが，今回利用する Arduino に搭載されている AVR マイコンには，FPU（浮動小数点演算装置）がないことに注意されたい．その影響で，プログラムを書き込む際にはコンパイル可能であるが，整数で演算するときに比べてメモリの消費量は多くなり，計算にかかる時間も長くなる．

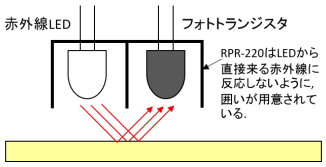

Fig. 1:　フォトリフレクタ概略図

Fig. 2:　RPR-220

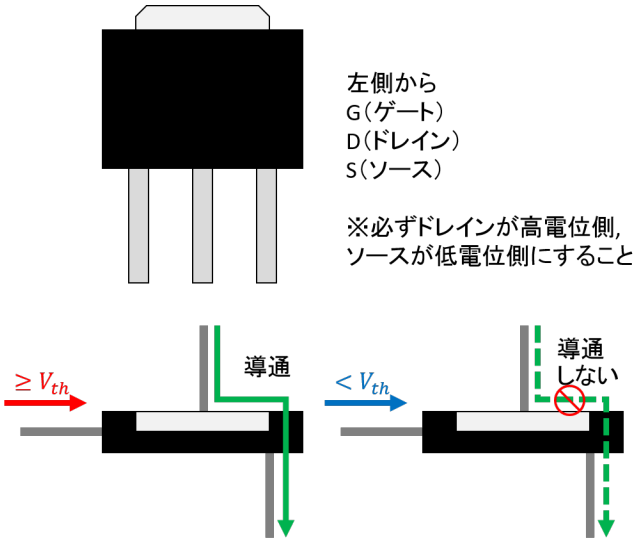

Fig. 3:　Nch FET の動作概略図

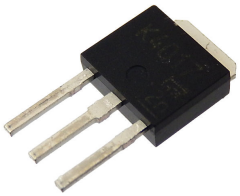

Fig. 4:　2SK4017

3　フォトリフレクタによるライン検出

3.1　フォトリフレクタ（RPR-220）

フォトリフレクタとは，Fig. 1 のように赤外線 LED（発光部）とフォトトランジスタ（受光部）を組み合わせたセンサである．本チュートリアルではフォトリフレクタとして RPR-220 を使用する．実際の部品の外観を Fig. 2 に示す．フォトリフレクタの出力 (電流) は，対象物の光の反射率、および物体との距離によって変化する．マイコンのアナログ入力は一般に電流ではなく電圧を測る．したがって，マイコンでフォトリフレクタ回路の出力を測るためには電流を電圧に変換する必要がある．今回はフォトリフレクタのエミッタ-GND 間に抵抗を入れることにより，電圧を測定する．出力電圧をマイコンの AD 変換機能を利用することで，反射光が弱いか強いか，すなわちフォトリフレクタが線上にあるかないかを判断する．なお，フォトリフレクタは対象物と近くなければ機能しない．データシートによれば，RPR-220 は対象物（すなわち地面）と 6〜8 mm 離れている環境が望ましい．

3.2 Nch FET (2SK4017)

今回はデバッグがしやすいように，フォトリフレクタの出力をある程度可視化できる LED をつけている．この LED を ON/OFF するスイッチングの目的で N チャネルの FET を使用している.[*1]FET は基本的にはゲート，ドレイン，ソースの 3 本のピンによって構成されている．Fig. 3 に示すように，Nch FET はゲートとソース間の電圧がスレッショルド電圧（しきい値電圧）V_{th} を超えるとドレイン-ソース間が導通するスイッチング素子と考えることができる．逆にスレッショルド電圧以下である場合にはドレイン-ソース間は遮断される．フォトリフレクタ回路の出力電圧が低いとき，すなわちフォトリフレクタが線上にあるとき，ドレイン-ソース間が遮断され，LED は点灯しない．逆に出力電圧が高いとき，すなわちフォトリフレクタが線外にあるとき，ドレイン-ソース間が導通し，LED は点灯する．

今回使用する FET は 2SK4017 である．実際の部品の外観図を Fig. 4 に示す．2SK4017 のスレッショルド電圧は 1.3〜2.5[V][*2]である．

[*1] 他にも P チャネルの FET というものがあり，挙動が全く異なるので”FET”という部品が一概に同じ動作をすると思わないこと．

[*2] FET の温度によって変化する．2SK4017 の場合 25 °C で 1.95[V] 程度である．

3.3 フォトリフレクタのキャリブレーション

危険	電源を入れる前に，フォトリフレクタ基板とシールド基板が正しく配線されているかどうかを確認すること．特に，電源の逆接には注意すること．

フォトリフレクタには個体差があり，同じだけの光量を与えても流れる電流が個体によって異なる．前回の Tutorial でも確認したように，製作したフォトリフレクタ基板には可変抵抗が搭載されており，これを回して適切に調整することで，個体差を吸収することができる．

まずは，フォトリフレクタ基板が出力する電圧を Arduino で読み取り，シリアルモニタからセンサの計測値を確認できるようなスケッチを作成する．Serial およびシリアルモニタの使い方については，Tutorial 2A の資料を参照すること．

```
#define LINE_R A0
#define LINE_M A1
#define LINE_L A2

void setup() {
  Serial.begin(1000000);
}

void loop() {
  int valR = analogRead(LINE_R);
  int valM = analogRead(LINE_M);
  int valL = analogRead(LINE_L);

  Serial.print(valL); Serial.print(',');
  Serial.print(valM); Serial.print(',');
  Serial.print(valR);
  Serial.println("");

  delay(100);
}
```

プログラム内の LINE_R, LINE_M, LINE_L は，それぞれ右，中央，左のセンサが接続されているアナログピンを表している．機体のフォトリフレクタ基板の部分を白い紙の上などに置いて，上記のスケッチを実行してみよう．シリアルモニタを開くと，左・中央・右のセンサ値が 0.1 秒間隔で表示される．調整ドライバーを使用して可変抵抗を回し，すべてのセンサの計測値がなるべく等しくなるように調整する．同様にして，黒い面についても計測値がなるべく等しくなるように調整する．

危険	可変抵抗を回す際は，非導電性の調整ドライバーを使用すること．導電性の高い金属製のドライバーを使うと，誤って基板上のパターンに接触してしまうおそれがある．

注意	計測値は 0〜1023 の範囲しか出力できず，範囲外の電圧が入力された場合，上下に飽和する．

4　ライントレースプログラムの作成

4.1　ライントレースの原理

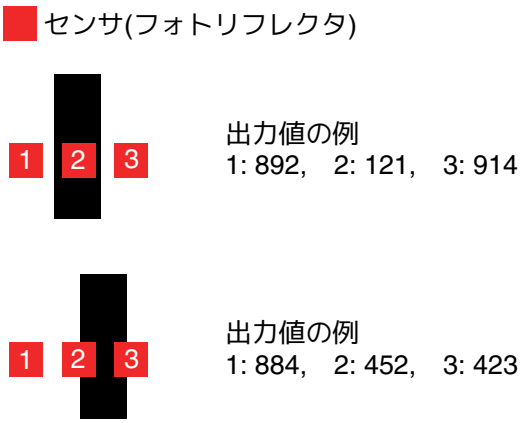

Fig. 5:　フォトリフレクタの計測値の例

実際のコースのラインの上に置き，キャリブレーションで使用したスケッチを実行して，シリアルモニタで数値を確認してみよう．Fig. 5 のように，黒線の上のセンサからは低い電圧が，白い板の上のセンサからは高い電圧が出力される．また，白と黒の部分をまたぐようにセンサが位置している場合，白・黒の状態の中間の値を示すことが確認できるだろう．

ライントレースの基本的な原理は，センサの計測値から機体の左右方向のずれを検出し，左にずれていれば右回りに，右にずれていれば左回りに回転方向の入力を与えるというシンプルなものである．通常，ライントレースを扱う資料では，計測値をあるしきい値で白・黒の 2 状態に分け，そのパターンに応じて定数を制御入力として与えるという方針で制御を行っていることが多い．しかし，この方法はしきい値や制御入力の値のような調整するべきパラメータが増えたり，センサの状態の切り替わりで入力が不連続に変化するため，機体の振動を誘発したりしてしまう．そこで，この Tutorial では，このような ON/OFF の制御ではなく，白・黒の中間の連続的な値を利用し，PD 補償を用いて制御入力を計算することにする．

制御目標は，Fig. 5 の上側の図のような状態である．また，横方向の機体位置の偏差は，左右のセンサ値の差として観測できることがわかる．したがって，この Tutorial では左右のセンサ値の差を 0 にするような PD 制御を考える．

⚠ 注意	フォトリフレクタのキャリブレーションを行わないと，正確に線に沿って走ることが難しくなる．

4.2　ライントレースの実装

2 年次に学習した PD 補償は，連続時間システムに対するものであった．しかし，コンピュータは離散的な時間ステップ (クロック) を基準として動作しているため，連続時間の制御器をそのまま実装することはできない．マイコンに制御則を実装する際には，一定の時間間隔 (**制御周期**と呼ぶ) ごとに以下の処理を行う必要がある．

1. センサからデータを取得する
2. それに応じて制御入力を計算する
3. 制御入力をアクチュエータの指令値に反映させる

本 Tutorial で扱うライントレースの場合，`analogRead` でフォトリフレクタ基板の信号の電圧を取得し，それに応じて PD 補償を用いて制御入力を計算し，最後に DC モータを駆動するために PWM のパルス幅を設定する，という一連の処理を行うことになる．

さて，Tutorial 2A で解説したように，Arduino では繰り返し実行する処理を `loop` 関数に記述するのであった．また，`loop` 関数の実行間隔を調整するために `delay` 関数を使用することができるが，`loop` 関数全体を一定の時間間隔で実行するためには不十分であるということも説明した．制御周期を `loop` 関数を使用して実装するためには，この問題を解決する必要があることに注意する．以下に示すコードは，これらに注意して実装した，ライントレースの実装例である．

```
// フォトリフレクタのアナログピン
#define LINE_R A0
#define LINE_M A1
#define LINE_L A2

// KP: P ゲイン, KD: D ゲイン
// **_NUM: 分子, **_DEN: 分母 を指定する
#define KP_NUM 1
#define KP_DEN 20
#define KD_NUM 0
#define KD_DEN 1

// モータ駆動用の PWM ピン
#define MOTOR_L_IN1 5
#define MOTOR_L_IN2 6
#define MOTOR_R_IN1 9
#define MOTOR_R_IN2 10

// モータの PWM を設定する関数
void setMotorPulse(int left, int right) {
  if(left > 0) {
    analogWrite(MOTOR_L_IN1, min(left, 255)); analogWrite(MOTOR_L_IN2, 0);
  } else {
    analogWrite(MOTOR_L_IN1, 0); analogWrite(MOTOR_L_IN2, min(-left, 255));
  }
  if(right > 0) {
    analogWrite(MOTOR_R_IN1, min(right, 255)); analogWrite(MOTOR_R_IN2, 0);
  } else {
    analogWrite(MOTOR_R_IN1, 0); analogWrite(MOTOR_R_IN2, min(-right, 255));
  }
}

unsigned long tPrev; // 前の時刻

void setup() {
  Serial.begin(1000000);
  tPrev = millis();
}

int x = 0;      // 今の状態
int xPrev = 0; // 前の状態
int xDiff = 0; // 状態の微分値
```

(つぎのページへ続く)

```
void loop() {
  int valR = analogRead(LINE_R);
  int valM = analogRead(LINE_M);
  int valL = analogRead(LINE_L);

  // 左右のセンサ値の差を状態として扱う．正のとき右寄り，負のとき左寄り．
  x = valR - valL;
  // 状態の時間微分．50 倍 は 0.02(秒) 分の 1 の意味．
  xDiff = (x - xPrev) * 50;
  xPrev = x;

  // 前進指令値
  int v = 200;
  // 回転指令値 (左回り正) PD 制御を実装している
  int w = x * KP_NUM / KP_DEN + xDiff * KD_NUM / KD_DEN;

  // モータの PWM パルスを設定
  setMotorPulse(v - w, v + w);

/*
  // デバッグ用の出力
  Serial.print(valL); Serial.print(',');
  Serial.print(valM); Serial.print(',');
  Serial.print(valR);
  Serial.println("");

  Serial.print("x = "); Serial.print(x);
  Serial.print(", ");
  Serial.print("dx = "); Serial.print(xDiff);
  Serial.print(", ");
  Serial.print("v = "); Serial.print(v);
  Serial.print(", ");
  Serial.print("w = "); Serial.print(w);
  Serial.println("");
*/

  // 処理時間を計算
  unsigned int tProc = millis() - tPrev;
  if(tProc < 20) {
    // 処理時間と合わせて 20ms になるように delay を入れる
    delay(20 - tProc);
  }
  tPrev = millis();
}
```

このスケッチは，これまでの Tutorial で学習してきた内容を使用したものであり，新しい点は以下の 2 点だけである．

1. `loop` 関数の一定間隔実行 (処理時間に応じた `delay`)
2. PD 制御の実装

これらの要素について解説する．

■ `loop` 関数の一定間隔実行

- `millis()`
  Arduino が起動してからの時間をミリ秒単位で取得する関数．

`loop` 関数の最後に記述されているのは，`loop` 関数の実行時間を一定に保つための処理である．前回のループの時刻と現在の時刻との差 `tProc` は，`loop` 関数の前半に記述した処理の所要時間に相当する．したがって，`20 - tProc` [ms] だけ待機することで，`loop` 関数全体としての実行時間は 20ms となり，結果として 20ms ごとに `loop` 関数が実行されることになる．また，`setup` 関数では時刻の変数 `tPrev` を初期化している．

■ PD 制御の実装

AD 変換の節でも述べたように，マイコンで実数の演算をさせることは場合によっては可能であるが，少なくとも Arduino に搭載されている AVR マイコンでは浮動小数点数の高速な演算ができない．この理由から，PD ゲインの定数 K_P, K_D を分母と分子に分けてプログラムの冒頭で定義し，整数同士の演算に簡略化している．

前節で説明したように，状態は左右のセンサの計測値の差である．

```
x = valR - valL;
```

状態の微分を離散時間で表現する方法はいくつかあるが，ここでは 1 次の後退差分で近似している．

```
int xDiff = (x - xPrev) * 50;
```

これを用いて PD 補償の計算を行っているのは，以下の行である．

```
int w = x * KP_NUM / KP_DEN + xDiff * KD_NUM / KD_DEN;
```

これを機体の回転に関する指令値に入力することで，左右のセンサの計測値の差が 0 になるように制御できる．

⚠ 注意	本資料の PD ゲインは，サンプルとして $K_P = 0.05, K_D = 0$ としている．この値のままでは収束速度や安定性が不十分である．コースを走らせながら考えてチューニングしてみよう．

5　中間試技会に向けたアルゴリズムの改良

この節では，中間試技会に向けたアルゴリズムの改良に役立つアイディアを示す．どのように実装するかについては，各自考えて取り組んでもらいたい．「正常に走行している」「脱線した」などの状態を使ったプログラミングについては，次回の Tutorial で扱う．

5.1　脱線を検出する

ラインから外れたときに，停止したりラインの上に復帰するための処理を加えたりすることを考える．脱線したときには，すべてのセンサがフィールドの白色部分を観測しているはずである．したがって，すべてのセンサの計測値があるしきい値を超えているかどうかを判定すれば，脱線しているかしていないかを簡易的に判定することができる．

ただし，センサの計測値には様々なノイズが加わるため，正常に走行している際に脱線したと誤判定してしまう可能性がある．これを防止するためには，脱線の条件を満たす計測値を連続して観測した場合にのみ，脱線したと判定する方法が考えられる．

5.2　交差点や横線マーカーを読む

競技フィールドに引かれたラインには，機体の走行に使用する線と，線の終端であることを示す横線がある．また，ポイントを獲得するためにはフィールド上の交差点を判定して旋回し，目的の場所へ到達する必要がある．交差点や横線の上ではセンサの計測値はすべて低い値を示していることが期待できる．したがって，脱線の検出と同様のアプローチを用いることができる．