Tutorial 4

システム創造プロジェクト TA: 田所祐一

2018 年 10 月 16 日（火）

1　概要

今回の Tutorial の目標は，PSD センサを使用して前方の壁に衝突しないようにしながら，位置を制御することである．前回は PD 制御を取り扱ったが，今回は PID 制御を使用する．また，より複雑なプログラムを実装する際の便利な考え方や，ファイルの分割，変数の命名規則などのテクニックについても解説する．

今回の課題は以下のとおりである．

1. PSD センサによる距離の計測 (第 2 節)
2. PID 制御の実装とチューニング (第 3 節)
3. ステートマシンの実装・ファイル分割の理解 (第 4 節)

2　PSD センサによる距離の計測

システム創造プロジェクトでは，Fig. 1 に示す PSD センサ (GP2Y0A21YK0F) が配布されている．PSD センサを利用することで対象物体との距離を計測することができる．

2.1　PSD センサの概要

PSD はセンサ自体が赤外線を発し，反射光の戻って来る位置を測定することで距離情報を計測する測距センサである．PSD は反射光の強度ではなく位置を測定に用いるため，反射物の色，反射率の影響を受けにくいという特徴がある．また，物体の接近だけでなく，物体への距離を数値的に計測することができる．GP2Y0A21YK0F のピン配置は，Fig. 2 のようになっている．VCC, GND にそれぞれ +5V, 0V を供給すると，距離に応じた電圧が VO から出力される．GP2Y0A21YK0F の駆動電源は，Tutorial 1 で製作したケーブルを使用してシールド基板から直接接続・給電する．

> **警告**　Tutorial 1 で製作した配線用ケーブルの順番が正しいことを，接続する前に確認しておくこと．

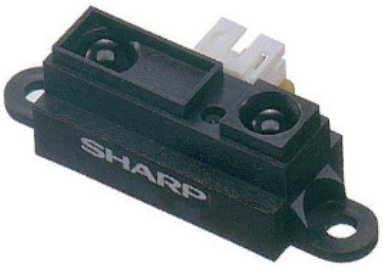
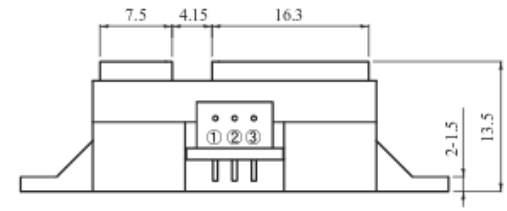


Fig. 1:　GP2Y0A21YK0F（PSD センサ）



Connector signal

	signal name
①	Vo
②	GND
③	Vcc

Connector :
J.S.T.TRADING COMPANY,LTD,
S3B-PH

Fig. 2:　GP2Y0A21YK0F (PSD センサ) のピン配置

2.2　PSD センサの出力特性

今回配布する PSD センサは距離に対応した電圧をアナログ出力する．この電圧を AD 変換してマイコンボードに取り込む．Fig. 3 にデータシートから引用した距離に対する電圧の特性グラフを示す．

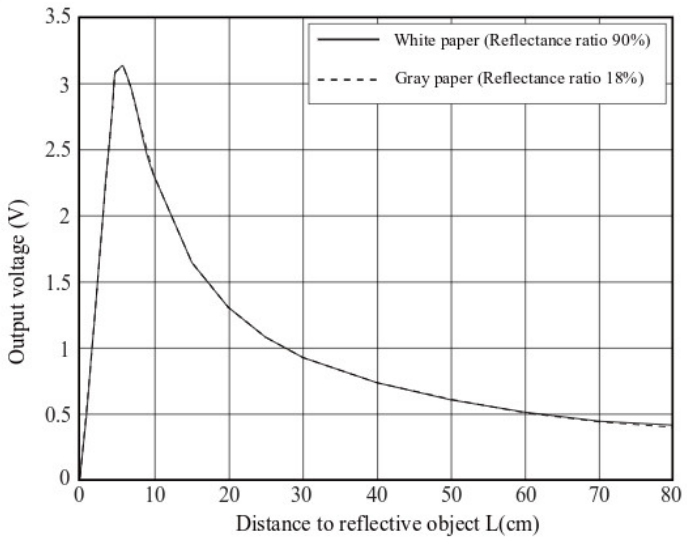


Fig. 3: 距離に対する電圧の特性グラフ（GP2Y0A21YK0F データシートより引用）

警告 PSD センサは個体によっても特性が異なるため, 精密な測定をする場合は実験をして個々の特性線図を考える必要がある.

以下に，PSD センサの計測値を `Serial` で PC に送信するサンプルスケッチを示す．`analogRead` や `Serial` の使用方法については，Tutorial 2A, 3 の資料を参照すること．

```
// PSD センサのアナログピン
#define PSD_F A3

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

void loop() {
  int valF = analogRead(PSD_F);
  Serial.println(valF);
  delay(100);
}
```

PSD センサをピン A3 に接続する．このスケッチを実行し，センサの前方約 15cm の位置に障害物を置いて計測する．そのときのおおよその AD 変換値をメモしておく．この値は，つぎの節の PID 制御の目標値として使用する．

3　PID 制御の実装

PSD センサで計測した距離をフィードバックし，前の壁との距離を目標値に収束させる PID 制御を実装する．

使用しているモータとギアボックスは，回転の始動や維持にかかるトルクが比較的大きいため，パルス幅が 0 でなくとも回転を停止してしまうことがある．そのため，PD 制御のみではこのトルクと制御入力が釣り合ってしまい，偏差が定常的に残ってしまう．定常偏差を 0 に近づけるには，積分項を追加することが効果的であることを 2 年次に学んだ．本節では，このことを実践的に確認するとともに，前回の Tutorial で学んだ制御周期と制御則の実装方法について，理解を深める．

今回の制御目標は，機体を前後に移動させることによって，PSD センサの計測値を予め取得しておいた目標値に収束させることである．すなわち，目標値と計測値の偏差を 0 に収束させれば良い．Tutorial 3 で実装した PD 制御をベースにして，偏差の積分値を表わす変数を追加し，制御入力の計算式に取り入れる．

サンプルスケッチの概略をつぎのページに示す．目標値の変数 `refF` には，前節で計測した PSD センサの AD 変換値を代入する．プログラムの基本的な構造は前回の PD 制御と同様である．このプログラムを通して，基本的な実装の方法を再確認してほしい．

⚠ 注意	つぎのページのスケッチでは，前回までに実装した `setup` 関数，および `setMotorPulse` 関数やモータドライバ用の IO ピンの定義を省略している．前回までの資料を参照して各自実装するか，前回作成したスケッチからコピーすること．

⚠ 注意	プログラムでは，$K_P = 0.25, K_I = 0, K_D = 0.01$ としているが，このパラメータのままでは定常偏差が残る上，即応性にも乏しい．どのように値を変更すれば改善するか復習し，考えよ．

```
// PSD センサのアナログピン
#define PSD_F A3

// KP: P ゲイン, KI: I 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 4
#define KI_NUM 0
#define KI_DEN 1
#define KD_NUM 1
#define KD_DEN 100

int refF = 300; // 目標値. ここを各自計測した値に設定する.
int e = 0;       // 今の偏差
int ePrev = 0;  // 前の偏差
int eInt = 0;   // 偏差の積分値
int eDiff = 0;  // 偏差の微分値

void frontDistanceControl() {
  int valF = analogRead(PSD_F);

  // 目標値との偏差を計算する
  e = refF - valF;
  eInt += e / 50;
  eDiff = (e - ePrev) * 50;
  ePrev = e;

  // 前進指令値 (PID)
  int v = e * KP_NUM / KP_DEN + eInt * KI_NUM / KI_DEN + eDiff * KD_NUM / KD_DEN;
  // 回転指令値 (左回り正)
  int w = 0;

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

void loop() {
  frontDistanceControl();

  // 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();
}
```

4　複雑なプログラムを作成するにあたって

複雑な動作をロボットにさせようとすると，動作の基本となるモータ駆動・センサ利用のための関数や制御周期を処理する関数のほかに，より上位のロジックが必要となってくる．本節では，そのロジックのおおまかな実装方法と，規模の大きいプログラムを整理するためのテクニックについて解説する．

4.1　ファイルの分割

POINT	ファイルを分割したサンプルを資料の `Software/PSDPIDMain` フォルダに用意してあるので，参考にしてもらいたい．

Tutorial のような簡単な内容でも徐々にプログラムが長くなっていることからわかるように，1 つのファイルに全ての処理を記述すると可読性が損なわれ，コードの再利用や，不具合が起きた際の原因箇所の特定が困難になる．したがって，ファイルを分割して整理しておくことが重要である．通常の C 言語では，ヘッダファイル (*.h) を利用したり関数のプロトタイプ宣言をしたりすることで，別の c ファイルにある関数を利用できるようになるが，C 言語の仕様を適切に理解していないと不具合の温床となってしまうことが懸念される．

Arduino IDE では，そのような深い理解なしにファイルを分割できるように，「タブ」と呼ばれる簡易的な仕組みが用意されている．ここでは今回の Tutorial のスケッチを，`PSDPIDMain`, `Control`, `Motor` の 3 つのファイルに分割してみる．まず，Arduino IDE の右側にある [▼] ボタンをクリックし，「新規タブ」をクリック，ファイル名には「`Control`」と入力し，新しいタブを追加する．作成されたファイルの中に，制御に関係するコード (アナログピン番号，ゲインの定数，偏差の変数の定義と，`frontDistanceControl` 関数) を移動する．同様に `Motor` タブを作成して，モータの駆動に関係するコード (ピン番号の定義と `setMotorPulse` 関数) を移動する．

このようにファイルを分割することで，それぞれの要素 (制御，モータ駆動など) がそれぞれのファイルにまとまる．また，メインのスケッチが `setup` 関数と `loop` 関数のみの最低限の内容になり，プログラム全体の見通しが良くなる．

4.2　ステートマシン

POINT	実装例を資料の `Software/StateMachineMain` フォルダに用意してあるので，参考にしてもらいたい．

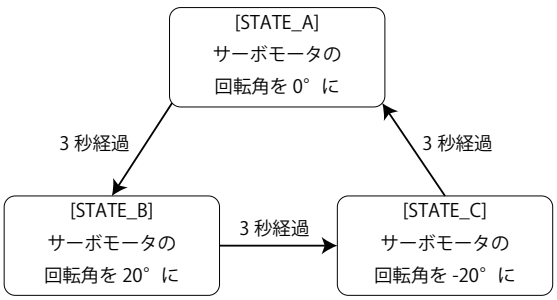


Fig. 4:　状態遷移図の例

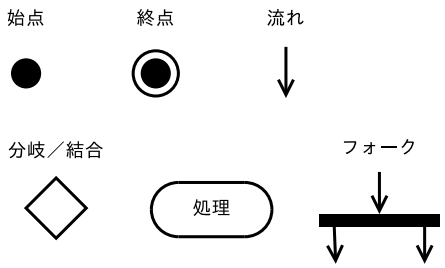


Fig. 5:　アクティビティ図で用いられる主な部品

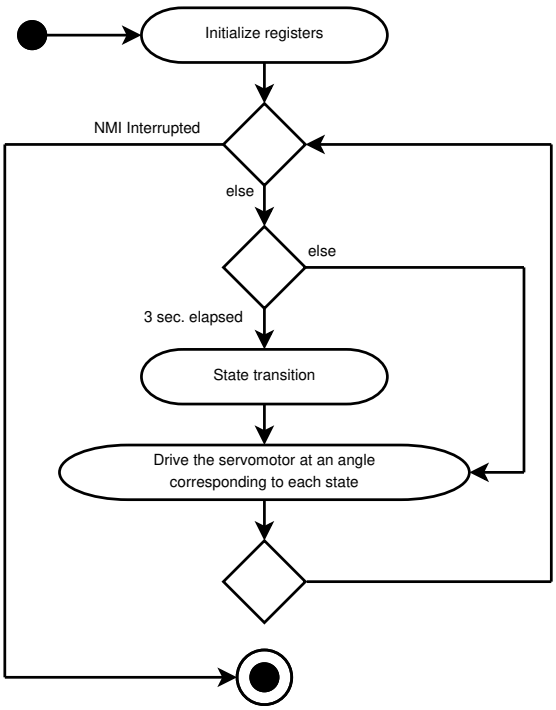


Fig. 6:　アクティビティ図の例

複雑なプログラムを作成するひとつの方法として，プログラムに「状態」を導入するという手法がある．これは，プログラムに状態を与え，内部あるいは外部のイベントによって状態が遷移していくことによりプログラムの挙動を表現しようというものである．その状態の遷移とイベントとの関係を表したものが状態遷移図である（Fig. 4 参照）．

今回の Tutorial で作成したプログラムに，収束を判定して停止するプログラムを追加することを考える．偏差が収束しているかどうかを判定するには，Control タブのコードにある偏差の変数 e の値を取得する必要がある．タブをまたいで変数を利用することは通常できないため，e の値を取得するための関数を Control に追記する．

```
// 偏差を返す関数
int getFrontDistanceError() {
  return e;
}
```

(Control.ino に追記)

ステートマシンの実装は，状態を表わす変数と，その値によって実行する処理を分岐させる switch 文を使用すると簡単である．また，処理内容と状態遷移を個別の switch 文に記述することで，状態の数が増えても読みやすさを維持できる．つぎに示すスケッチは，収束を判定して停止する動作を行うステートマシンの実装例である．

```
#define CONTROL_ACTIVE 0
#define CONTROL_INACTIVE 1
int state = CONTROL_ACTIVE;

#define EPSILON 30        // 偏差の絶対値が EPSILON より小さければ収束と判定する
int convCount = 0; // 連続で収束と判定された回数

void loop() {
  // 状態ごとの処理
  switch(state) {
    case CONTROL_ACTIVE:
      frontDistanceControl();
      // 収束判定 収束したと連続で判定された回数を記録する
      if(abs(getFrontDistanceError()) < EPSILON) {
        convCount++;
      } else {
        convCount = 0;
      }
      break;
    case CONTROL_INACTIVE:
      endControl();
      break;
    default:
      break;
  }

  // 状態遷移
  switch(state) {
    case CONTROL_ACTIVE:
      // 50 回以上 (1 秒以上) 連続で収束と判定されたとき，CONTROL_INACTIVE へ遷移する
      if(convCount >= 50) {
        state = CONTROL_INACTIVE;
      }
      break;
    case CONTROL_INACTIVE:
      break;
    default:
      break;
  }

  // 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();
}
```

(StateMachineMain.ino)

今回の Tutorial 程度の単純なプログラムならともかく，本番でロボットを動かすくらいの複雑さになると，コードから動作を読み解くことが困難になる．したがって，状態遷移図やプログラム全体のアクティビティ図（流れ図，Fig. 6 参照）を作成することを推奨する．これらの図は自分の理解を深めるだけでなく，プログラムの構造を他者と共有するのにも役立つため，積極的に活用してほしい．その際は Fig. 5 に示すアクティビティ図でよく用いられる部品を参考にしてもらいたい．

4.3 変数・関数・定数の命名規則

変数名や関数名の書き方にある一定の規則を設けておくと，プログラムの可読性の向上が期待できる．この規則のことを，命名規則と呼ぶ．以下に代表的な命名規則を示す．

- Camel Case
  語頭を大文字にする記法.
  例: `frontDistanceControl()`, `getFrontDistanceError()`, `convCount`
- Snake Case
  語句どうしをアンダースコア (_) でつなぐ記法.
  例: `CONTROL_ACTIVE`, `PSD_F`
- Hungarian
  変数名の頭に型や役割を表わす文字列をつける記法 (Tutorial では使用していない).
  例: `uiNumberOfPeople` (`unsigned int` 型), `cntNumberOfPeople` (カウントを表わす)

命名規則には，このほかにも動詞・名詞の利用や順番に関する規則なども含まれる．コード内で統一されていれば，基本的にはどれを使用しても構わない．Tutorial 資料のプログラムは，Arduino 標準の定数や関数との統一性を考えて，つぎのように使い分けている．

- 定数: 大文字 Snake Case
- 変数: 小文字スタートの Camel Case
- 関数: 小文字スタートの Camel Case
- ファイル名: 大文字スタートの Camel Case