詳解 Conidae 技術資料

種子島ロケットコンテスト2023

はじめに

この資料は、第19回種子島ロケットコンテストのCansat部門ミッション種目にて

優勝したチームConidaeのプログラミング、電装、構造すべての技術についてできる限り詳しく解説した資料です。

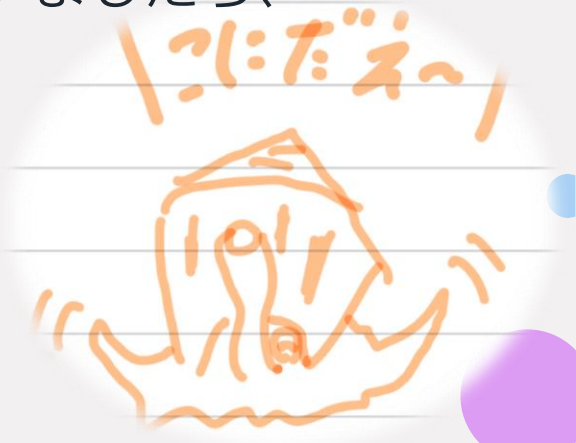
私たちConidaeは、この資料を種子島ロケットコンテストに参加するライバルの皆さんに公開することで、大会をより高度に、より面白くし、宇宙を目指す学生全体が、新しい視点から宇宙開発に挑戦することの助けになればと思っています。

聞きたいことがある、連絡を取りたい、何かを一緒にやりたいなどございましたら、

https://twitter.com/Conidae_52

↑こちらのtwitterアカウントのDMまでご連絡下さい。

できる範囲でお答え致します。



はじめに

今回Conidaeが作成したプログラムは、すべてこちらのGitHubにて公開しています。

アプリとして使用するには、Android Studioをインストールして、リポジトリをクローンもしくはzipファイルとしてダウンロードした後、この資料の最後にある手順に従って使えるようにしていただく必要があります。

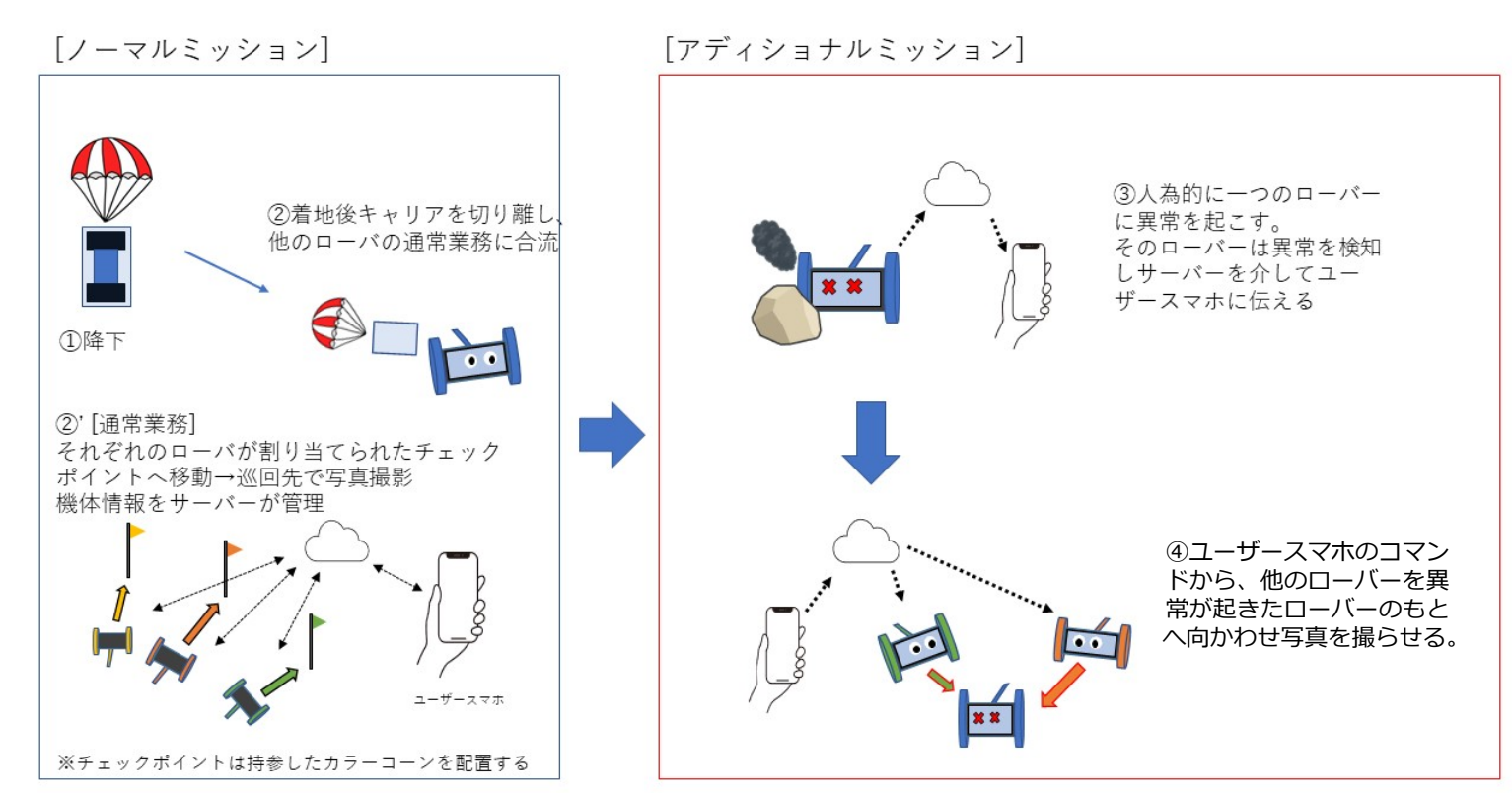
なお、このプログラムを書いたのはプロでもなんでもなく、半年前にはAndroidもJavaもGoogleAppScriptも、すべて何も知らない0の状態から勉強してきたAndroid超初心者の学生たちです。そのため、大変申し訳ないことに、バグや動作不良、動作環境の違いなどによる不具合は必ずと言っていいほど起こりますし、それらにより生じた損害等については一切の責任を負いかねます。

もしアプリを使用する場合は、自己責任の下で使用していただきますよう、何卒宜しくお願い致します。また、プログラムの流用や改変などについてはご自由に行っていただいて構いません。

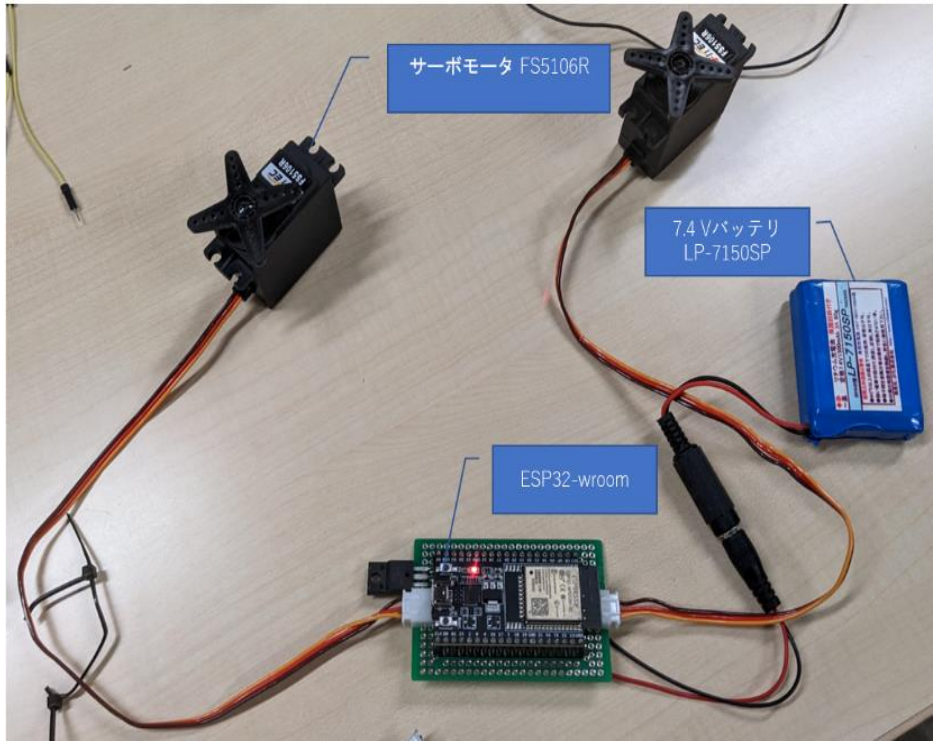
目次

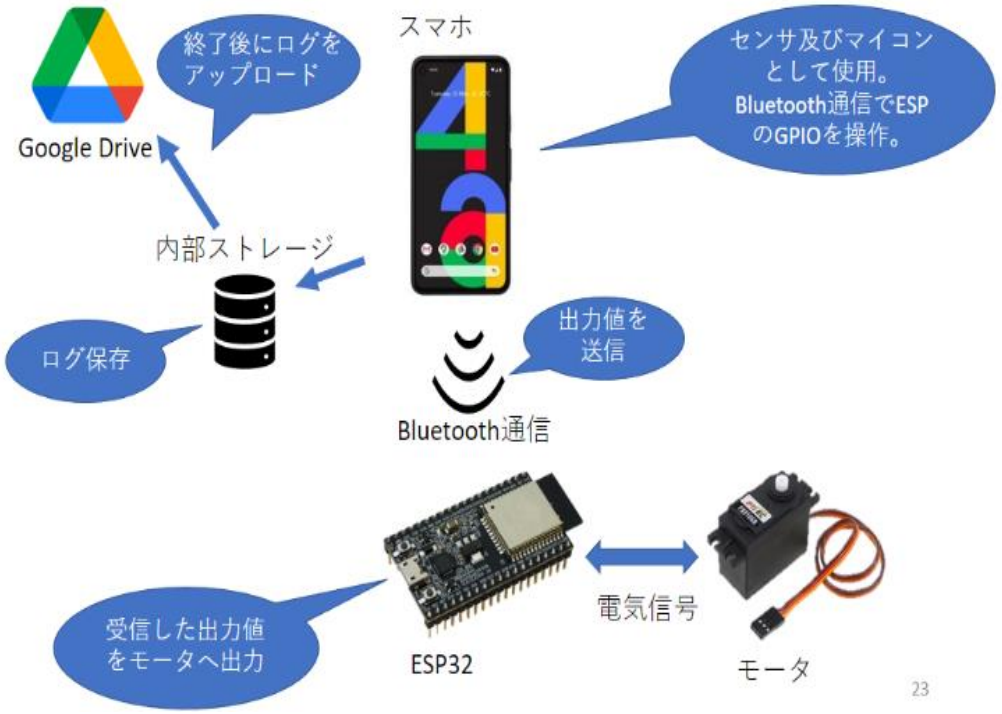
概要

今回私たちは、次に示すようなミッションを行うこととし、
・スマホを用いたローバーを作成する
・複数機体をサーバーによって連携させる
という手段によってこれを達成しました。



概要

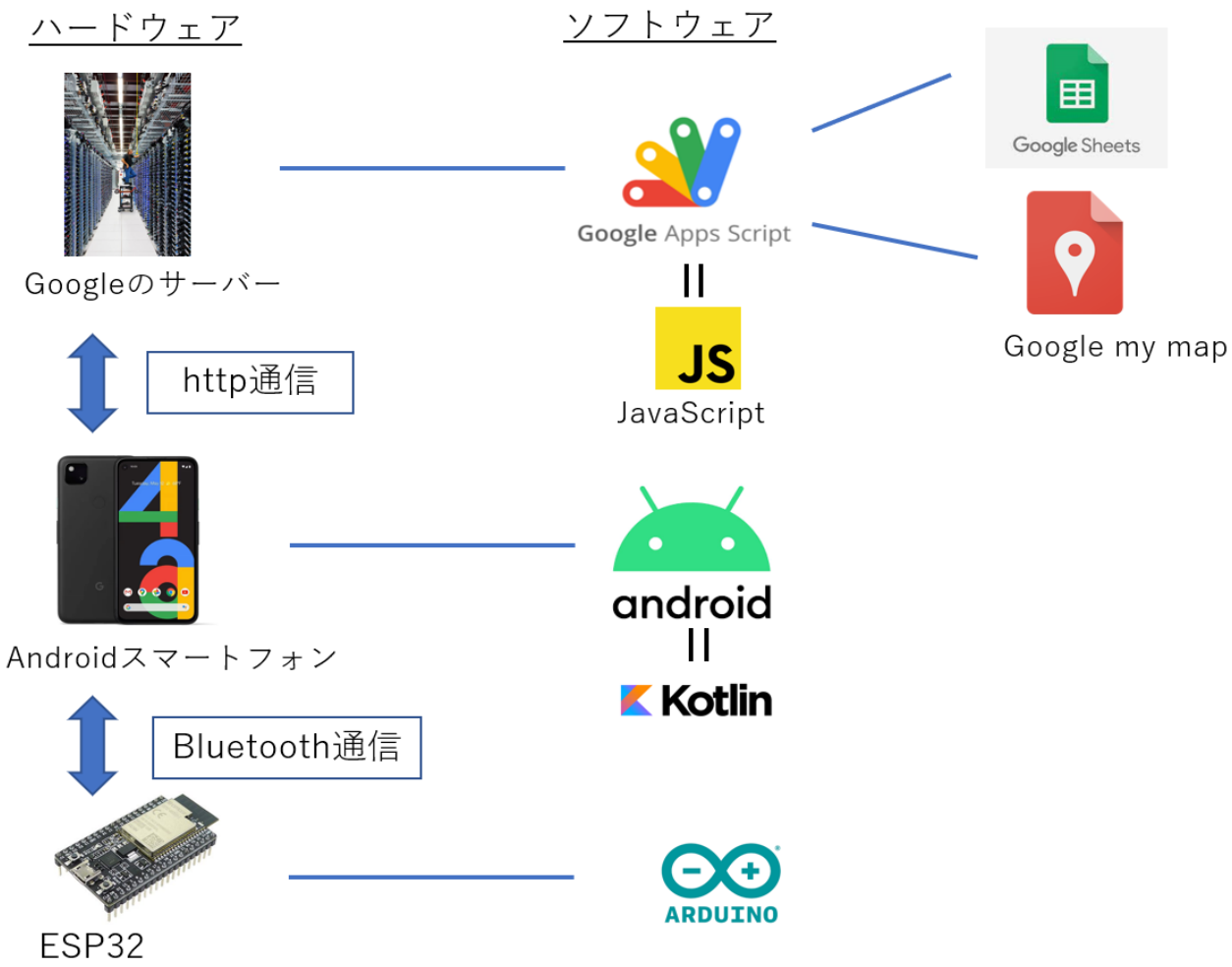




今回の探査機はこのような形で動作しています。

概要

機体の連携に使用したサーバー及びプラットフォームの構成は右のような形になります。

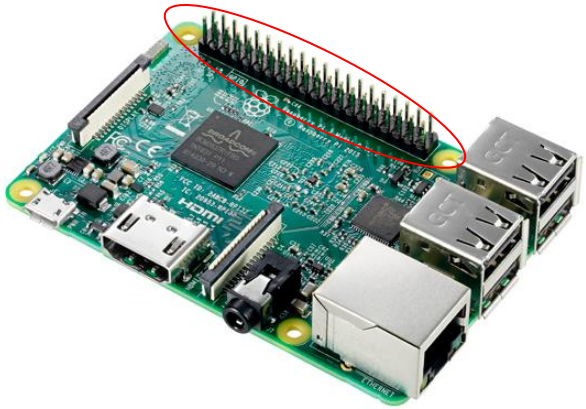


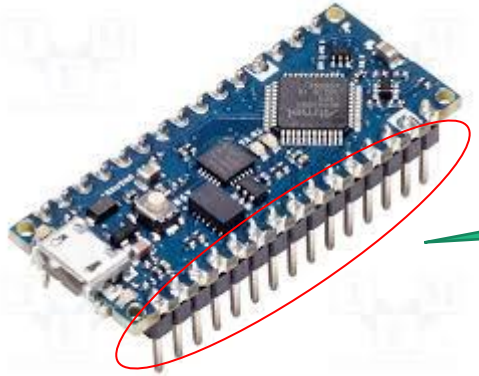
誘導

Bluetooth接続

スマホにはもとより多数のセンサと、探査機にしては高性能なコンピューターが搭載されているため、ロボットの制御用コンピューターに流用できれば回路の簡略化と探査機の高性能化につながります。

ただし、Cansatローバーによく使用されているraspberryPiやArduinoなどとの決定的な違いとして、GPIOが存在しません。





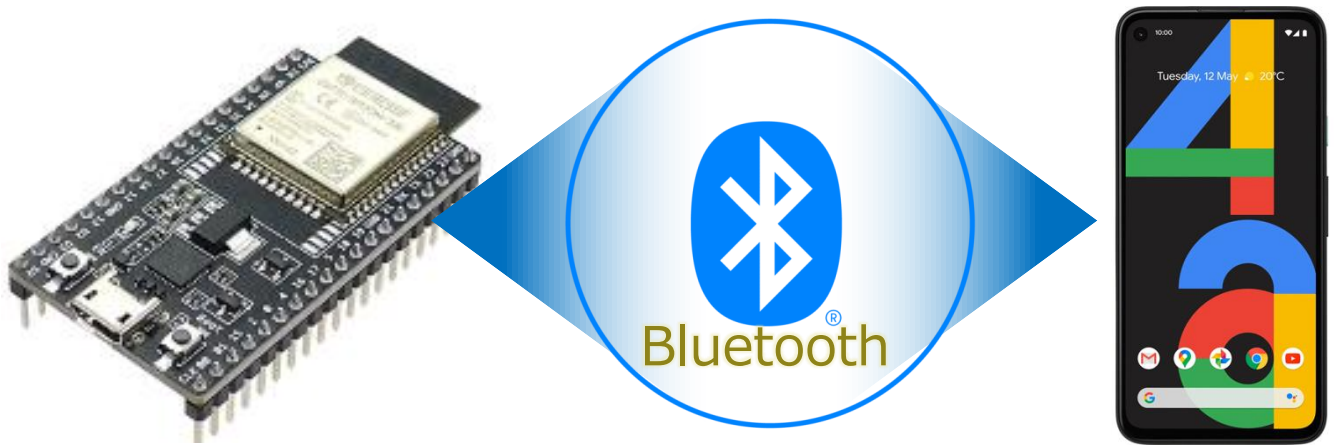
スマホには無い！

GPIOが無いと、算出した制御量をモーターに伝えることができないため、スマホを使用するために、Bluetooth接続によってESP32のGPIOを操作することにしました。

Bluetooth接続

Bluetoothはその通信の目的によって適切なプロファイルを選択する必要がありますが、今回は、Bluetooth ClassicのSPPというプロファイルを用いてAndroid-ESP間の通信を行っています。

SPPというプロファイルは、Bluetoothを使ってシリアル通信をすることができるプロファイルで、ArduinoやraspberryPiで行っているシリアル通信と同じように通信を行えます。



Bluetooth接続

Bluetoothでの通信では、

“左モーターの出力,右モーターの出力;”

という形でデータを送っており、例えば右の出力を30,左を60にしたい場合は、

60,30;

という文字列をBluetoothを介して送ることになります。

詳しくはGithubに載せたConidae_ESP32のプログラムを見ていただきますが、

ESP側に書き込んだArduino言語では、文字列

操作用の便利なライブラリが見当たらなかったため

Char型の配列と区切り文字のポインタを使うことで

制御値の文字列の分割を行いました。

```
String val_ipt = SerialBT.readStringUntil(';');
//文字列解析のためにchar型に変換
char charBuf[50];
char *value = NULL;
val_ipt.toCharArray(charBuf,50);
value = strtok(charBuf,",");
String rightValue = value;
int right= rightValue.toInt();
value = strtok(NULL,",");
String leftValue = value;
int left= leftValue.toInt();
```

Bluetooth接続

Android側では、何らかの理由でBluetooth接続が切れたときのために、データを送る際にBluetooth接続が切れていたら繋ぎなおすようなプログラムにしました。

これは、振動などでESPへの給電が一瞬ストップしたときなどに、プログラムが異常終了することを防ぐものです。

ただし、これはBluetooth接続をする関数内で同じ関数を呼び出すことになっているため、無限ループになってしまうという欠点もあります。

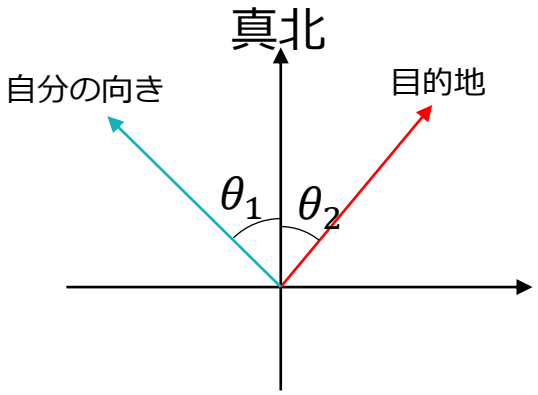
```
try{
    mBluetoothSocket.connect()
    mOutputStream = mBluetoothSocket.outputStream
    mInputStream = mBluetoothSocket.inputStream
    println("接続ができたんだえ")
    isConnectionOK = true
}catch (e:Exception){
    println("何らかの問題が発生したんだえ")
    isConnectionOK = false
    try{
        mBluetoothSocket.close()
        println("もう一度接続を試みるんだえ")
        this.connectDevice()
    }catch (ex:IOException){
        println("閉じれなかったんだえ")
        ex.printStackTrace()
    }
}
```

誘導

誘導に関しては、現在位置と目的地から算出した目標方位角と、現在の目標方位角との偏差を算出し、その偏差が無くなるようにモーターを駆動させることで目的地の方を向き、更に前進して目的地までの距離を縮めるという動作をさせています。

目標方位の算出はAndroidのLocationライブラリを使って算出し、

偏差は目標方位-現在方位の値を条件分岐させて算出しています。



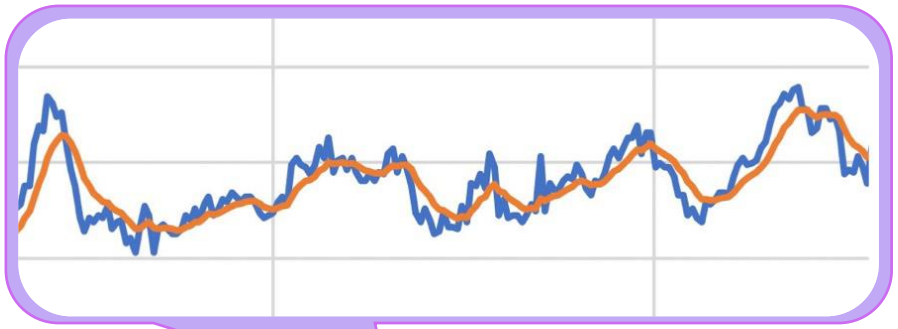
θ_1：自分の向き $(-180 < \theta_1 < 180)$
θ_2：目標角 $(-180 < \theta_2 < 180)$
$\Delta = \theta_2 - \theta_1$ ：目標角と自分の向きの差$(-360 < \Delta < 360)$
φ：偏差 $(-180 < \varphi < 180)$

$$\varphi = \begin{cases} \Delta + 360\ (-360 < \Delta < -180) \\ \Delta\ (-180 < \Delta < 180) \\ \Delta - 360\ (180 < \Delta < 360) \end{cases}$$

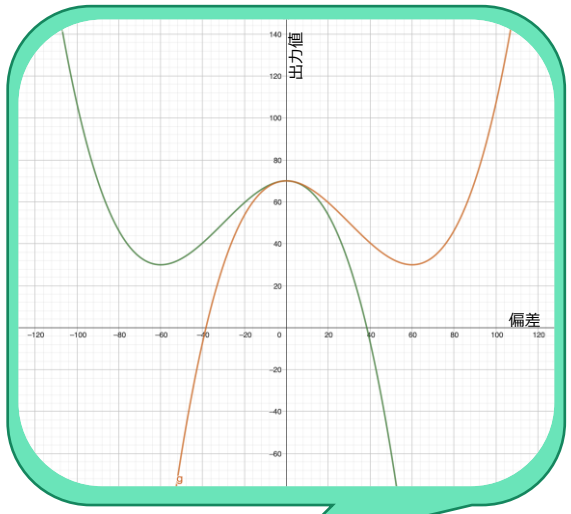
駆動

左右のモーター出力は、モーターが急激に動くのを防ぐため、偏差に対して出力値が三次関数を描くように関数を組みました。

その上で移動平均を使って、時間的に滑らかに変化するようにし、スムーズな動作を可能にしています。



移動平均



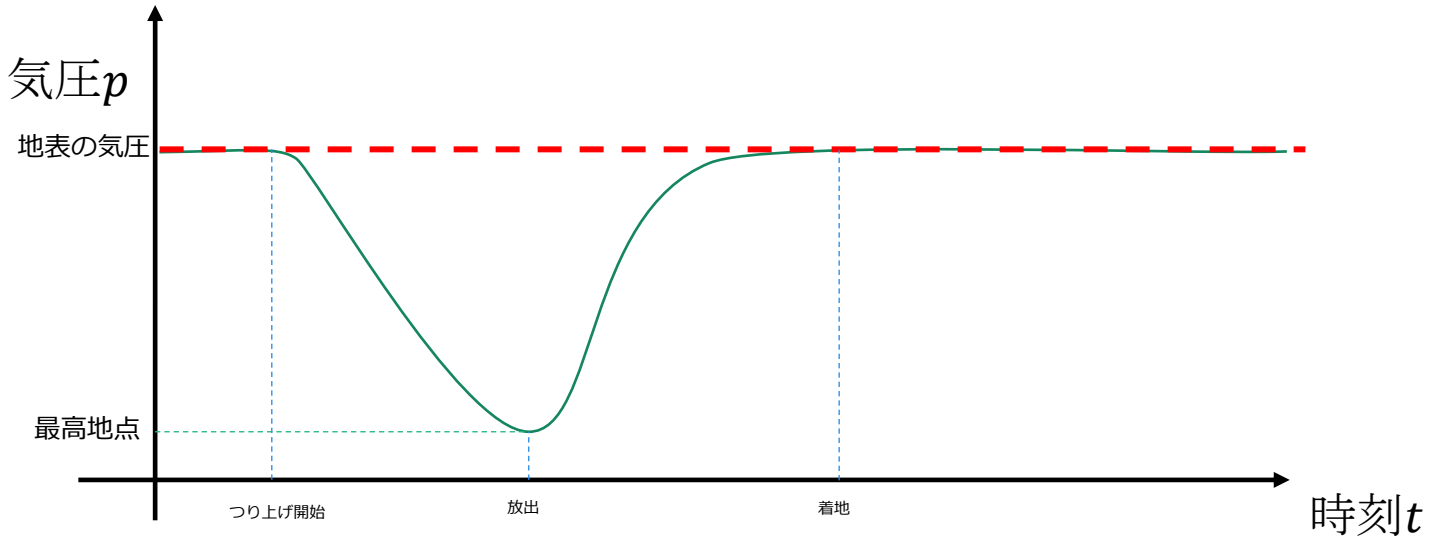
三次関数

```
212             phi /= 2
213             val preRight = right
214             val preLeft = left
215             right = (0.5*preRight+0.5*(-0.00296296*phi*phi*phi-0.1333333*phi*phi+70.0)).toInt()
216             left = (0.5*preLeft+0.5*(0.00296296*phi*phi*phi-0.1333333*phi*phi+70.0)).toInt()
```

着地判定

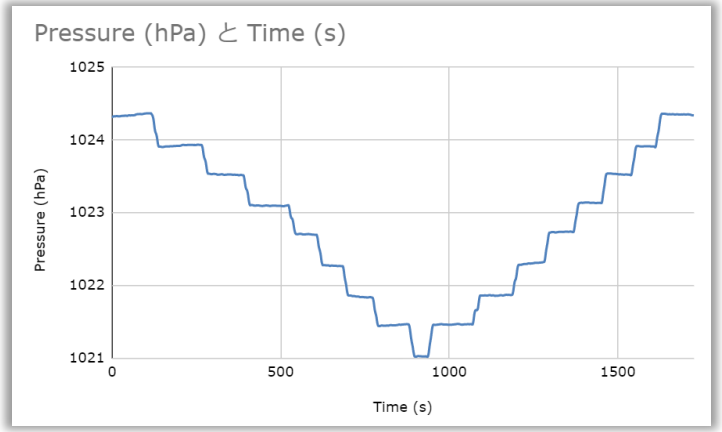
着地判定は、上空でパラシュートからの切り離しが行われてしまわないよう気圧を用いた着地判定を行いました。

どう考えたかというと、気圧が高度に対して線形的に変化していくことはわかっているため、大会のように、クレーンで釣り上げて落とす場合は以下のようなグラフを描くことが予想されます。

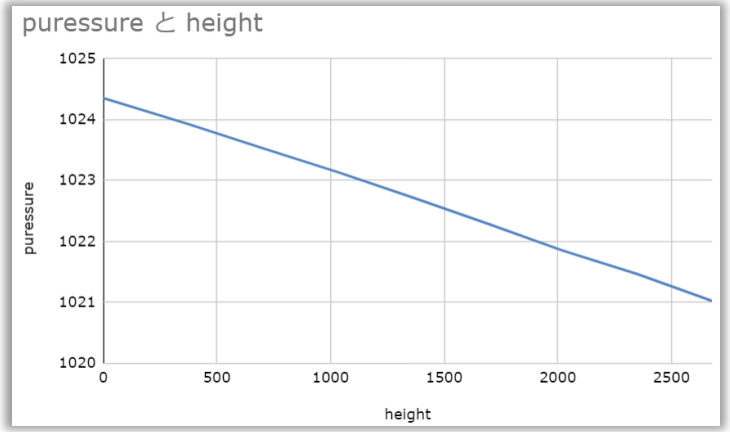


着地判定

そのため、気圧と高度の値を実際に計測し、気圧-高度の勾配から比例係数を算出することで落下距離を算出するプログラムを書き、大会で言われていた30mのうち、15m程度落下したことが確認出来たらタイマーでの計測に移り、溶断回路を作動させることにしました。



気圧の測定は階段を使い、一階昇るごとに一時停止するという測定方法をとりました。



高度の算出は一階分の高さをメジャーにて測り、高度と気圧の関係を出しました。最終的な係数は最小二乗法を用いて求めます。

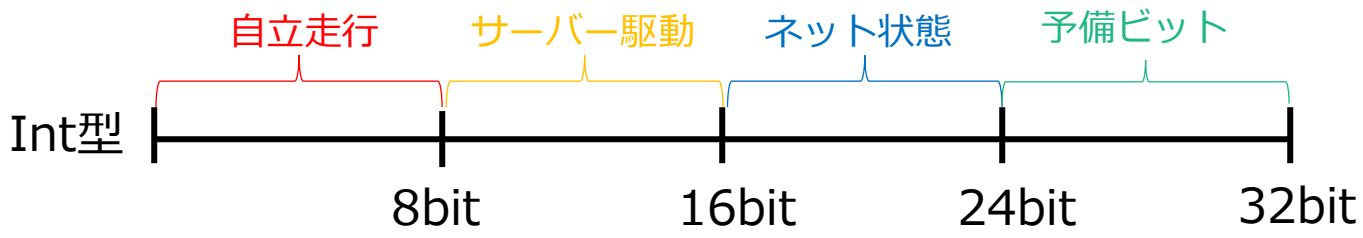
なお、気圧そのものではなく気圧の勾配から算出するようにした理由は、日や天候によって気圧の値が変化してしまうからです。

ステータス管理

スタックや反転、Bluetoothやhttp通信のエラーなどの、機体の状態は32ビットのInt型の変数に値を代入し、それぞれのビットに一つの状態を割り当てることで表現しています。

これとビット論理積を使用することで、一つのint型変数に代入された数字が、32個の状態を同時に表現することができ、サーバーと通信する際に自身の状態をたった一つのint型の数を送信するだけで済むようになります。

具体的には以下のように割り当てて使用しています。



自立走行関連：
スタック = 2
Bluetoothエラー = 4
反転 = 8
気圧着地判定 = 16
サーバー駆動関連：
ユーザー = 256
ローバー = 512
実行中 = 1024
ネット状態関連：
オンライン = 65536
タイムアウト = 131072

ログ

ログに関しては、動作ログとセンサログの二つのファイルを走行開始時に作り、それぞれログが取られた時間と記録する内容を一行にまとめてログを取るようにしています。

また、ログのファイル名が被ってしまうのを防ぐため、ファイル名にはファイルが作られた時刻を使用するようにしています。

動作ログ

```
2023-3-5-15-20-31-674,運転開始だえ
2023-3-5-15-20-31-681,ブルルルル...
2023-3-5-15-23-18-612,30.3742685461636,130.95983415842 に向けて運転するんだえ
2023-3-5-15-23-18-613,ゴールから逆算するんだえ
2023-3-5-15-23-18-615,ゴールまでの距離：{77.06229}目標方位{165.2248077392578}なんだえ
2023-3-5-15-23-18-616,ゴールから逆算するんだえ
2023-3-5-15-23-18-617,ゴールまでの距離：{77.06229}目標方位{165.2248077392578}なんだえ
2023-3-5-15-23-18-617,出力するんだえ
2023-3-5-15-23-18-619,0,0
2023-3-5-15-23-18-645,ゴールから逆算するんだえ
2023-3-5-15-23-18-646,ゴールまでの距離：{77.06229}目標方位{165.2248077392578}なんだえ
2023-3-5-15-23-18-647,出力するんだえ
2023-3-5-15-23-18-647,32,31
2023-3-5-15-23-18-669,ゴールから逆算するんだえ
2023-3-5-15-23-18-670,ゴールまでの距離：{77.06229}目標方位{165.2248077392578}なんだえ
2023-3-5-15-23-18-671,出力するんだえ
2023-3-5-15-23-18-671,48,47
2023-3-5-15-23-18-693,ゴールから逆算するんだえ
2023-3-5-15-23-18-695,ゴールまでの距離：{77.06229}目標方位{165.2248077392578}なんだえ
2023-3-5-15-23-18-697,出力するんだえ
2023-3-5-15-23-18-699,56,55
```

センサログ

```
2023-3-5-15-20-31-710,30.3749468,130.9596203
2023-3-5-15-20-33-29,30.374957,130.9596175
2023-3-5-15-20-34-56,30.3749576,130.9596197
2023-3-5-15-20-35-56,30.374958,130.9596225
2023-3-5-15-20-36-78,30.3749566,130.9596258
2023-3-5-15-20-37-67,30.3749542,130.9596275
2023-3-5-15-20-38-66,30.3749519,130.9596287
2023-3-5-15-20-39-98,30.3749486,130.9596307
2023-3-5-15-20-40-90,30.3749466,130.9596326
2023-3-5-15-20-41-88,30.374946,130.959632
2023-3-5-15-20-42-91,30.3749475,130.9596297
2023-3-5-15-20-42-977,30.3749482,130.9596291
2023-3-5-15-20-44-92,30.3749486,130.959629
2023-3-5-15-20-45-241,30.3749491,130.9596296
2023-3-5-15-20-47-69,30.3749495,130.9596307
2023-3-5-15-20-48-58,30.37495,130.9596317
2023-3-5-15-20-49-103,30.3749508,130.9596317
2023-3-5-15-20-50-44,30.374951,130.9596313
2023-3-5-15-20-51-55,30.374951,130.959631
2023-3-5-15-20-52-868,30.3749509,130.95963
2023-3-5-15-20-54-78,30.3749507,130.9596293
2023-3-5-15-20-55-70,30.3749508,130.9596287
2023-3-5-15-20-56-49,30.3749508,130.9596283
2023-3-5-15-20-57-56,30.3749507,130.9596276
```

機体の連携

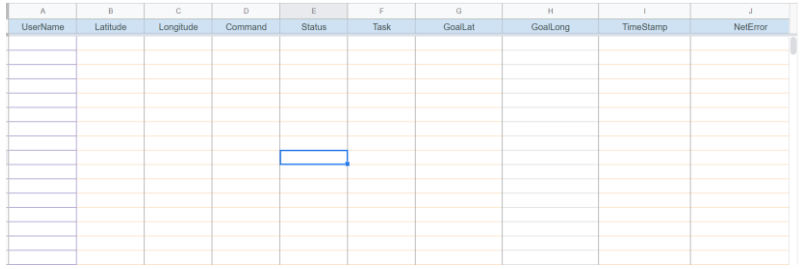
サーバー

今回使用したサーバー機能は、GoogleAppsScript(以下、GAS)というプラットフォームと、スプレッドシートを用いて実装しています。

参考： https://ja.wikipedia.org/wiki/Google_Apps_Script

GASによって書かれたプログラムは、ウェブアプリケーションとしてデプロイすることができ、これにスプレッドシートをデータベースとして組み込むことで、機体の情報を保持・加工して、機体連携のために動作する機能を実装できます。

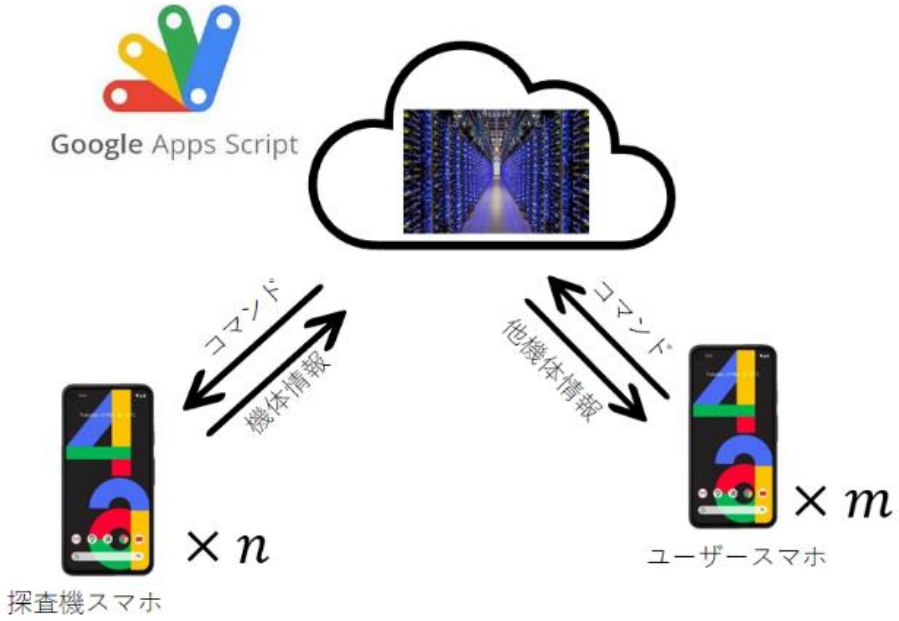
```
function doGet(query) {
  const acts = new ActionSheet("ここにシートのIDを入力します")
  if(query.parameter.comm==2){
    acts.adddata[acts.UserName]=query.parameter.name
    acts.adddata[acts.Latitude]=query.parameter.lat
    acts.adddata[acts.Longitude]=query.parameter.long
    acts.adddata[acts.Command]=query.parameter.comm
    acts.adddata[acts.Status]=query.parameter.state
    acts.register()
    let txt = acts.createReturn(query.parameter.comm)
    return ContentService.createTextOutput(txt)
  }
```



デプロイしたアプリは、GETメソッドで渡されるクエリのコマンドに応じて動作を変更するようになっています。

Http通信

複数機体の連携には、Http通信を使ってサーバーとして機能するwebアプリケーションに、GETメソッドでコマンドと、それを実行するのに必要な情報を載せたクエリを送り、そのレスポンスに、自信が行うべき動作とその実行に必要な情報を受け取るようにしています。動作としては、以下の図のようになります。




```
?comm=4&name=NAMEK&lat=35.719899&long=139.9104054&state=512
```

GETメソッドを使う時のクエリは、
主にコマンドと自分の機体情報が載っています。

4#{"UserName":{"UserName":"UserName","Latitude":"Latitude","Longitude":"Longitude","Command":"Command","Status":"Status","Task":"Task","GoalLat":"GoalLat","GoalLong":"GoalLong","TimeStamp":"TimeStamp","NetError":"NetError","Online":"Online","GoalUser":"GoalUser"},"conites":{"UserName":"conites","Latitude":32.7198956,"Longitude":137.9103972,"Command":64,"Status":1280,"Task":2,"GoalLat":32.719382568119,"GoalLong":132.911107942461,"TimeStamp":"2023-03-08T07:51:52.413Z","NetError":0,"Online":0,"GoalUser":""},"NAMEK":{"UserName":"NAMEK","Latitude":"33.719899","Longitude":"135.9104054","Command":"4","Status":51200000,"Task":1,"GoalLat":52120,"GoalLong":52120,"TimeStamp":"2023-03-08T09:32:25.976Z","NetError":0,"Online":1,"GoalUser":""}}

レスポンスはJSON形式で帰ってきます。

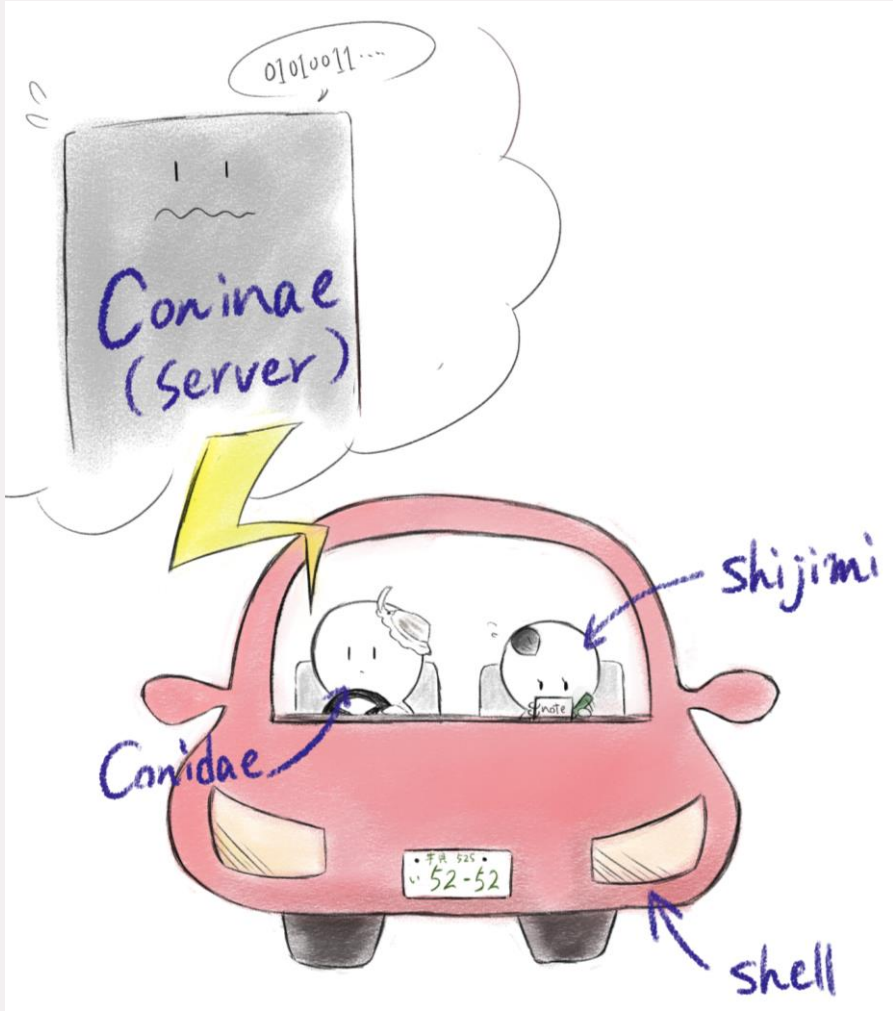
その他

オブジェクト指向

Androidアプリは、主にJavaまたはKotlinという、オブジェクト指向言語と呼ばれている言語を使用して開発します。

非常にざっくりとした説明をすると、オブジェクト指向でプログラミングするというのは、現実に存在する“物”をそのままプログラムとして書くことを意味します。

そのため、今回私たちは右のようなシステムをプログラムとして書き起こすことにしました。



クラス

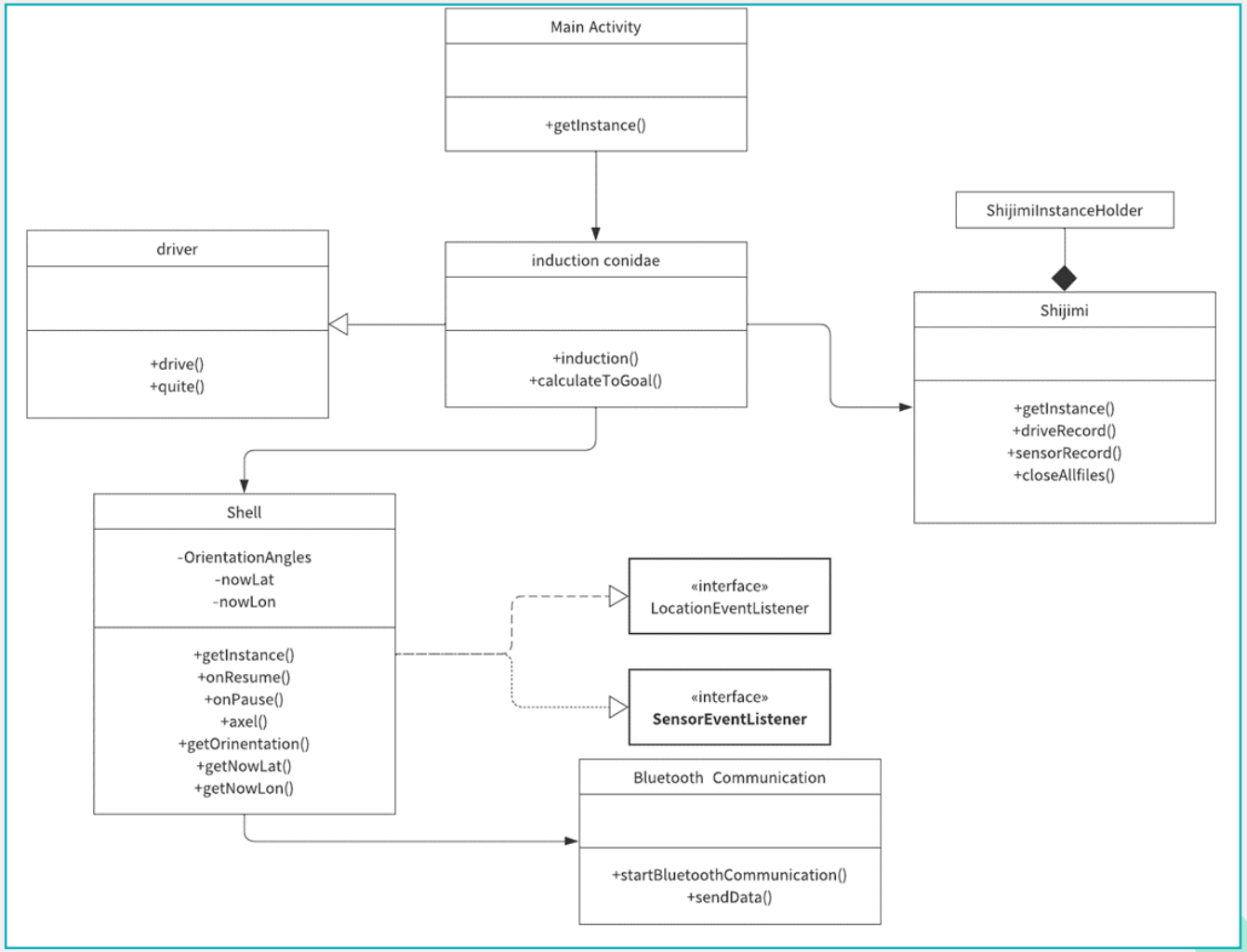
右に初期のクラスの構成を示します。（最終的な物とは少々形が変わっております。）

主なクラス

Shell:機体の状態を保持、Bluetooth接続機器への信号の送信

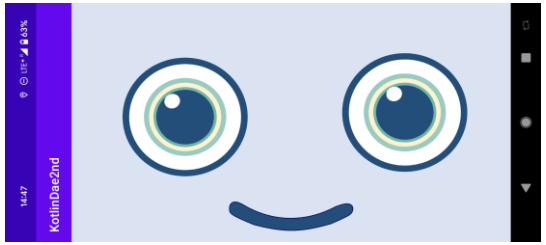
InductionConidae:機体の誘導

Shijimi:ログの書き込み

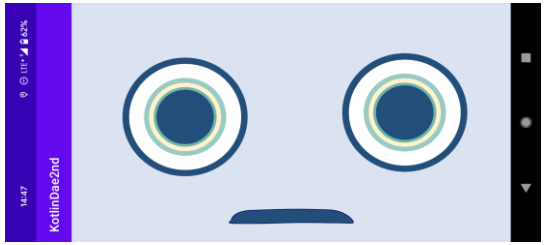


UI

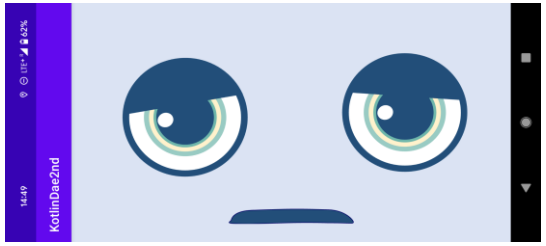
この探査機はAndroid端末のアプリケーションとして開発したため、GUIから探査機の状態を知ったり、目的地の設定などができるようにしました。



通常時



スタック、反転時



接続エラー

探査機として動作する際には、先述したステータス管理を用いて、ビット論理積で0が出るかそれ以外かで表情を変化させることができるようになっています。

```
if(state.and(InductionKonidae.STATE_PRESSUREOK)==0){//高度が十分下がっていない時
    fightFace()
}
else if(state.and(InductionKonidae.STATE_REVERSE)!=0){//反転しているとき
    nothingFace()
}
else if(state.and(InductionKonidae.STATE_CONNECTION_ERR)!=0){//Bluetoothエラーのとき
    troubleFace()
}
else{
    normalFace()
}
```

UI

ビット論理積を使った状態の把握方法について具体例を挙げると、

例えば、機体の状態を保持している変数statusが

status == 66568

だったとすると、これをビットで表せば、

00000000000000010000010000001000

となり、

反転時の値は8,すなわち

00000000000000000000000000001000なため、これとstatusでビット論理積をとると

00000000000000000000000000001000となり、

status & 8 != 0 がtrueなため、反転として判断されます。

同様の判定方法で、ほかの状態についても把握しています。

UI

地上局として動作させるときには、フィールド上の状態を地図に表示するため、サーバーが現在保持しているデータを受け取り、解析したうえで地図にプロットする必要があります。


```
UPDATE_MAP -> {//帰ってきたコマンドがマップ更新のとき
    val retJson = JSONObject(payload)
    val keys: Iterator<String> = retJson.keys()
    removeAllOverlays()//マップ上のオーバーレイを全消去
    while(keys.hasNext()){
        val key = keys.next()
        val json = retJson.getJSONObject(key)
        try{...}catch (e:java.lang.Exception){
            println("It does not Double!")
        }
    }
```

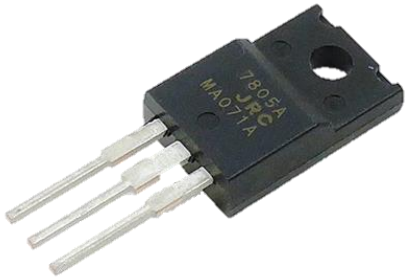
サーバーからは、下のようなJSON形式の文字列が返ってくるため、要素を配列に格納し、繰り返し文の中で地図上にプロットしています。

{"UserName":{"UserName":"UserName","Latitude":"Latitude","Longitude":"Longitude","Command":"Command","Status":"Status","Task":"Task","GoalLat":"GoalLat","GoalLong":"GoalLong","TimeStamp":"TimeStamp","NetError":"NetError","Online":"Online","GoalUser":"GoalUser"},"conites":{"UserName":"conites","Latitude":32.7198956,"Longitude":137.9103972,"Command":64,"Status":1280,"Task":2,"GoalLat":32.719382568119,"GoalLong":132.911107942461,"TimeStamp":"2023-03-08T07:51:52.413Z","NetError":0,"Online":0,"GoalUser":""},"NAMEK":{"UserName":"NAMEK","Latitude":"33.719899","Longitude":"135.9104054","Command":"4","Status":51200000,"Task":1,"GoalLat":52120,"GoalLong":52120,"TimeStamp":"2023-03-08T09:32:25.976Z","NetError":0,"Online":1,"GoalUser":""}}

電装

探査機の駆動回路は以下の素子を使用しています。

ESP32
Bluetoothの受信



NJM7805
5Vへの降圧

NJM7806
6Vへの降圧

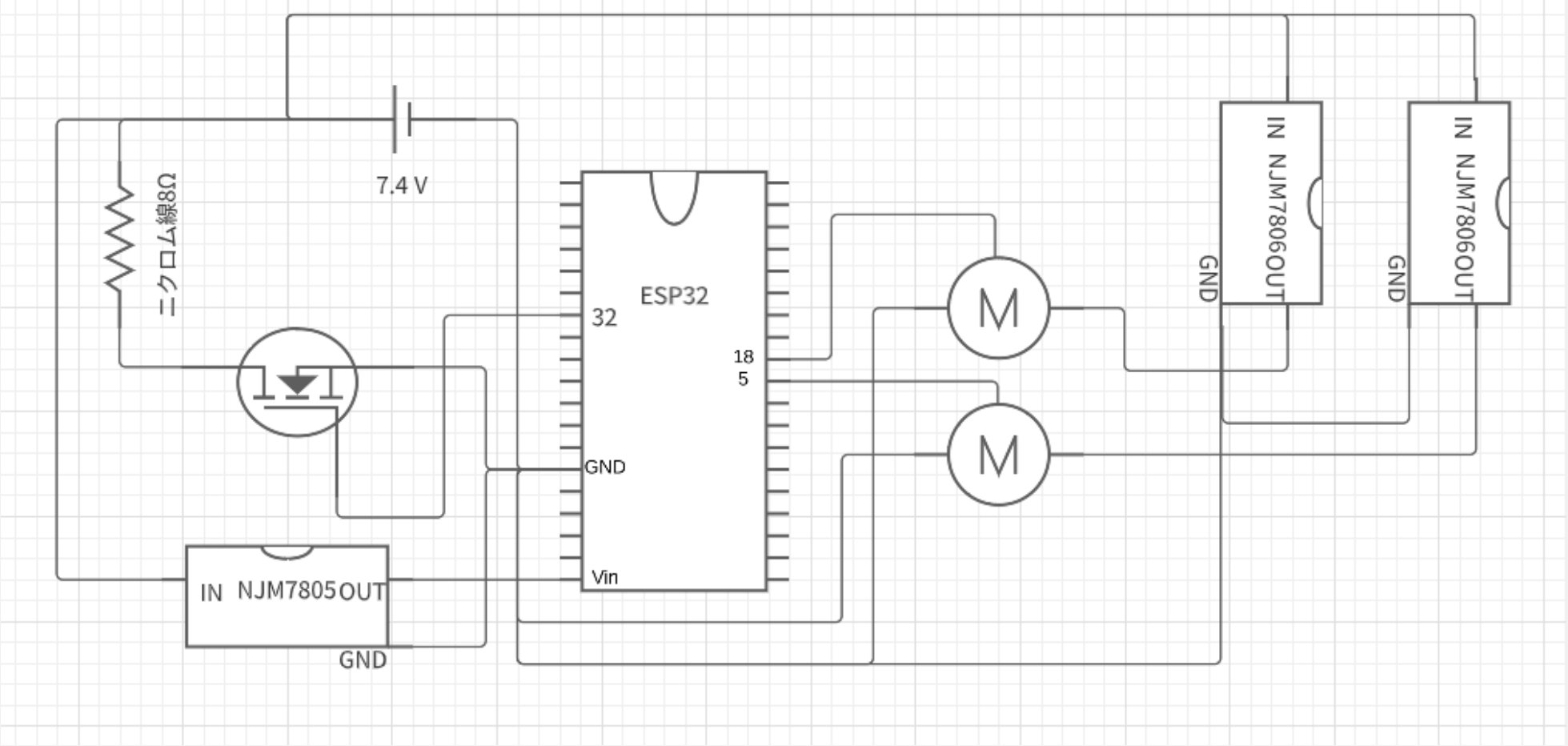
FS5106R
連続回転サーボ

2SK4017
溶断回路用Mosfet

DTP603048-2S
7.4V,860mAh

電装

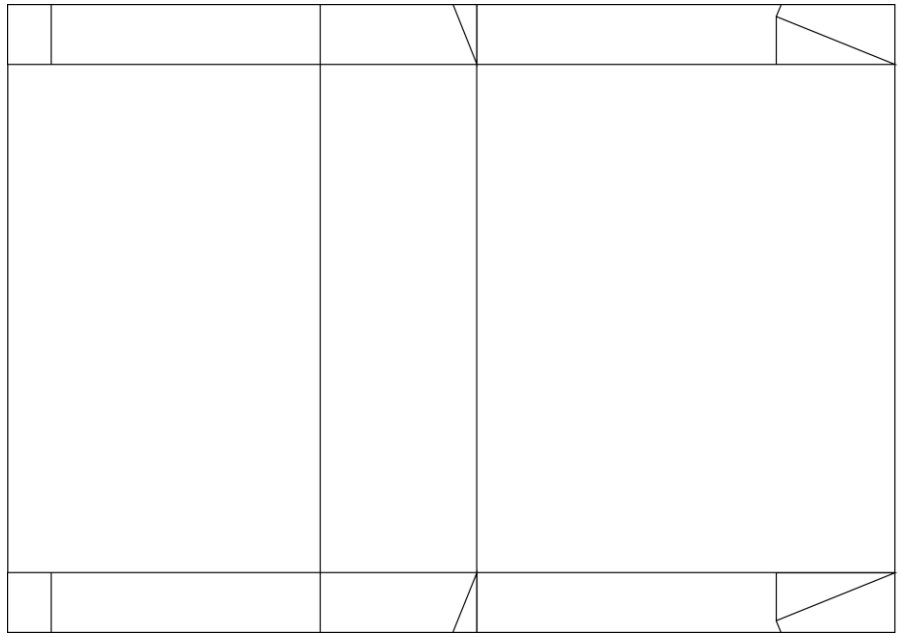
回路図は以下の通りです。(見づらくて申し訳ありません)



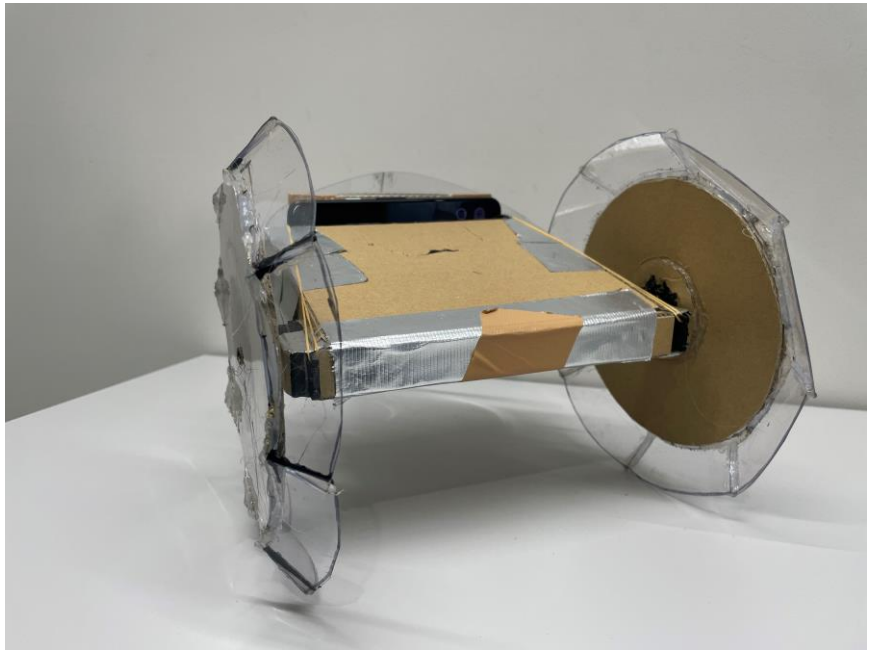
ESP以外の電子部品は3端子レギュレータとmosfetで、バッテリー電圧の降圧に使用しています。

量産機体

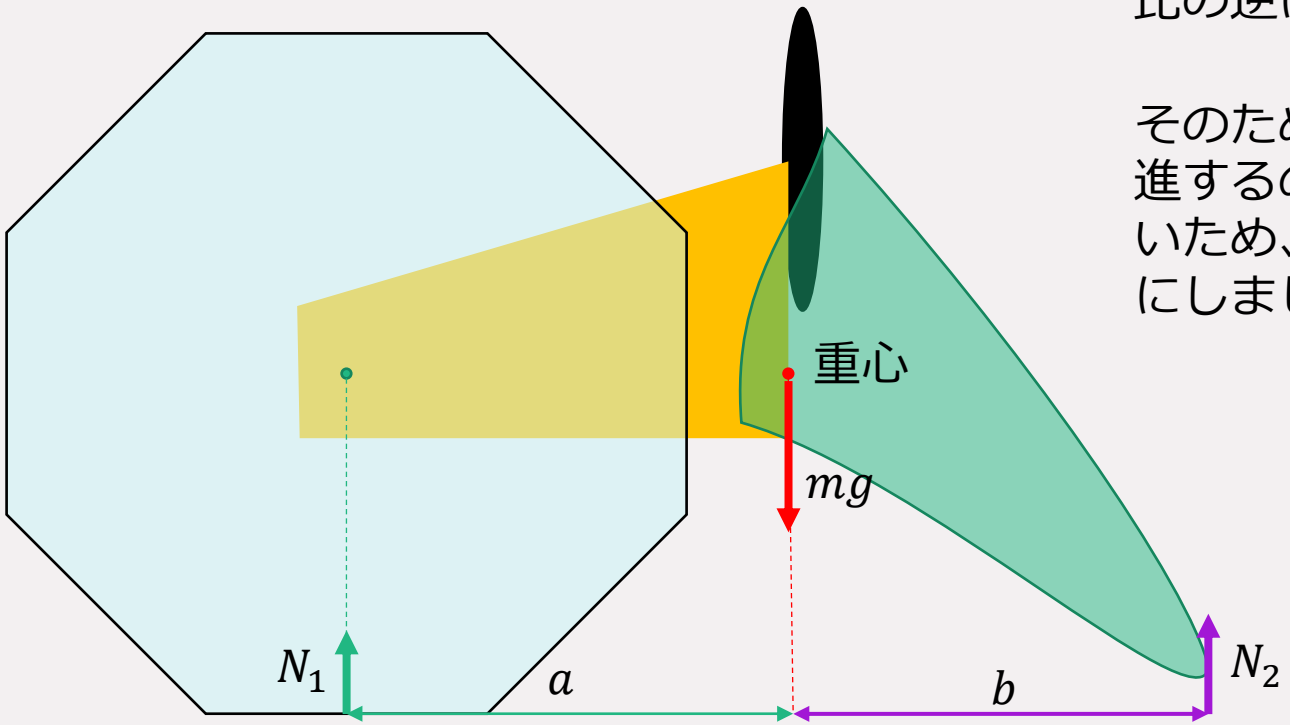
量産機体は、量産するにあたって製作的にも資金的にもとにかくコストのかからないボディとする必要がありました。そこでボディには、メルカリなどへの出品用に売られている段ボールと、下敷きを使用して製作しました。



ボディの展開図。プリントアウトして段ボールに張り付けて使える。



量産機体



左の図のように、量産機は車輪とスタビライザーで自重を支えていて、垂直抗力の比は、重心からの距離の比の逆になります。($N_1 = mg \cdot \frac{b}{a+b}$)

そのため、図のbの長さを十分に大きくとらないと前進するのに十分な摩擦力を車輪側で得ることができないため、今回はスタビライザーを後ろに長くする構造にしました。

よくある質問

技術交流会でよく聞かれた質問をまとめました！

・なんでサーボなのか？

→モータードライバーが要らないからです、サーボでも駆動力があれば走ります！

夏の能代でも使用してたので実績もあります

・量産機はなぜダンボールで作ったのか？

→ 大きな理由はコストが抑えられることと、加工性が高く、手に入りやすいことです。

・アプリは1から作ったんですか？

→はい、誰も何も知らない状態から半年間、みんなで勉強と実装を幾度となく繰り返して作りました!

よくある質問

・スタビライザーがなんで大きいんですか？/なんでこの形なのか？

→スマホが重いので、スタビライザーが小さかったりすると後ろに荷重がかかりすぎてしまい安定しないからです。

・車輪がなぜ内側に折れ曲がってるのか？

→モーターの軸に剪断応力だけが加わっている形にしたかったからです

・何を勉強したんですか？

→僕たちの場合は、右のようなことを新しく学びました。

情報系の人が全くいなかったため、必然的に学ぶことは多くなってしまった印象です。

新しく学んだこと…

Java
Kotlin
Google App sprict
Bluetooth
Http
Jason
ポインタ
Arduino
Google Map API
Google Cloud platform
オブジェクト指向
ビット論理積
Git
Github
Xml
ローパスフィルター
Gradle
Python
Eventlistener
デザインパターン
CAD
LT Spice

Etc…

アプリの導入・使用方法

アプリを使えるようにするには、

1.GoogleAppScriptのプロジェクトを作る。このテキストを作ったプロジェクトのエディタに張り付け。

アプリの導入・使用方法

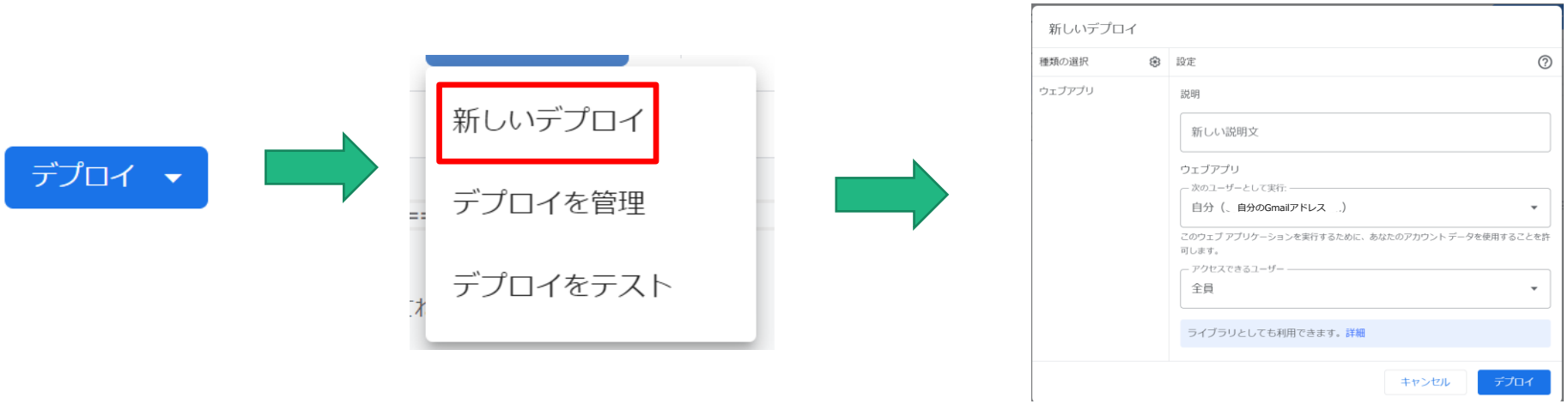
アプリを使えるようにするには、

2.データベースにするスプレッドシートを作る。

データベースのテンプレ を自分のドライブにコピー。

3.2.で作ったシートのIDを1.のプロジェクトのプログラム内に書き込む

4.プロジェクトをデプロイ。このとき、種類をウェブアプリにして、公開範囲を全員にする。

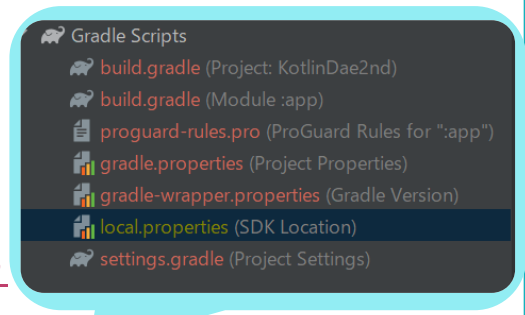


アプリの導入・使用方法

4.KotlinDae2nd内のCommanderMapActivityとServerConidaeFaceActivity内のurlの、プロジェクトIDを書き換える。

```
private fun communicateServer(sendCommand:Int){//サーバー通信をする関数。渡されたコマンドによって動作内容を変える。
    val queue= Volley.newRequestQueue( context: this)
    var url="https://script.google.com/macros/s/ここにIDをいれるよ！！！！/exec?"
    url += when (sendCommand) {
        UPDATE_MAP -> {//マップ更新
```

```
private fun serverDrive(sendCommand:Int){
    val queue=Volley.newRequestQueue( context: this)
    var url="https://script.google.com/macros/s/ここにIDをいれるよ！！！！/exec?"
    url += when (sendCommand) {
        REGISTER_AND_REQUEST_GOAL -> {
            "comm=2"
        }
```



5.GoogleMapのAPIキーを取得する。APIキーの取得方法はこちらとこちら

6.local.properties の最後にMAPS_API_KEY=“取得したキー”の形で書き込む

7.あとはアプリをインストールして開始。


おしまい。