StampFly 勉強会 2026

ドローン制御工学ハンズオン ワークショップ

伊藤 恒平

2026/3/2 – 3/6

資料ダウンロード / Slide Download

Docswell

スライド資料はこちらからダウンロードできます

https://www.docswell.com/s/Kouhei_Ito/KLV77G-2026-03-02-010945

GitHub リポジトリ / GitHub Repository

stampfly_ecosystem

ファームウェア・ツール・資料の全ソースコード

`https://github.com/M5Fly-kanazawa/stampfly_ecosystem`

工場出荷状態に戻す / Restore Factory Firmware

注意

ワークショップを進めると、機体・送信機のファームウェアは上書きされます。
工場出荷状態に戻したい場合は、以下のコマンドを実行してください。

機体（Vehicle）を工場出荷状態に戻す

```
1 sf flash vehicle --legacy
```

送信機（Controller）を工場出荷状態に戻す

```
1 sf flash controller --legacy
```

注意: 事前に `source setup_env.sh` で開発環境をセットアップしてください。

デバイスを USB 接続した状態で実行してください。

リポジトリの更新 / Sync Repository

通常の更新

`git pull` でリモートの最新変更を取り込む

```
cd ~/stampfly_ecosystem
git pull
```

`git pull` でエラーが出た場合

ローカルの変更がリモートと競合すると pull が失敗する。その場合はローカルを破棄してリモートに強制一致させる:

```
git fetch origin
git reset --hard origin/main
git clean -fd
```

注意: ローカルの変更・追加ファイルはすべて失われます。

Lesson 0

環境セットアップ

Environment Setup

StampFly Workshop

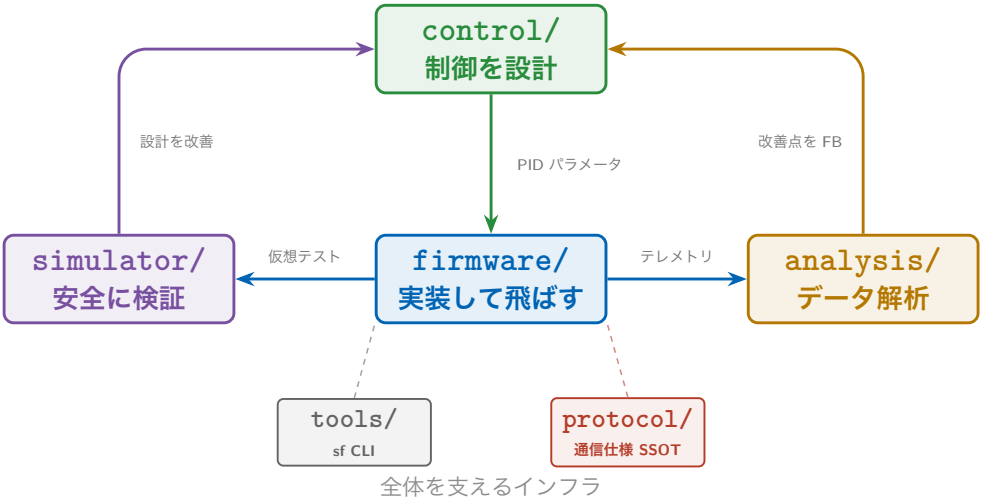
ワークショップカリキュラム / Workshop Curriculum

Day	Lesson	テーマ
Day 1	0–2	環境セットアップ + モータ + コントローラ
Day 2	3–5	LED + IMU + 初フライト
Day 3	6–8	モデリング + システム同定 + PID 制御
Day 4	9–12	姿勢推定 + API + 独自 FW + Python SDK
Day 5 AM	13	精密着陸競技会

StampFly ハードウェア / Hardware

項目	仕様
MCU	ESP32-S3 (M5Stamp S3)
IMU	BMI270 (6 軸 400 Hz)
気圧	BMP280
ToF	VL53L3CX
質量	37 g（バッテリ含む）
通信	ESP-NOW + WiFi
バッテリ	LiPo 1S 3.7 V

エコシステム全体像 / Ecosystem Overview



開発ツール sf CLI / Development Tool

カテゴリ	コマンド例	説明
ビルド	`sf build` / `sf flash`	ファームウェア開発
診断	`sf doctor` / `sf monitor`	デバッグ
ログ	`sf log wifi` / `sf log viz`	テレメトリ取得・可視化
シミュレータ	`sf sim run`	仮想環境で練習
キャリブレーション	`sf cal gyro/accel/mag`	センサ校正

ワークショップの基本ワークフロー（各レッスン共通）

`sf lesson switch N` → `sf lesson build` → `sf lesson flash`

レッスン切替 → ビルド → 書き込みの 3 ステップで、

コードを書いてすぐに実機で動作確認できます。

Windows 環境構築 (1/2) / Windows Setup

確認コマンド (CMD)	期待される結果	なければ
`git --version`	git version 2.x	`winget install Git.Git`
`python --version`	Python 3.8 以上	`winget install Python.Python.3.12`

インストール手順（CMD で実行）

1. リポジトリをクローン
2. `install.bat` を実行（ESP-IDF + sfcli 自動インストール）
3. `setup_env.bat` で開発環境をアクティベート

```
git clone https://github.com/M5Fly-kanazawa/stampfly_ecosystem
cd stampfly_ecosystem
install.bat
setup_env.bat
```

Windows 環境構築 (2/2) / Driver & Verification

USB シリアルドライバ

CH9102F ドライバをインストール（M5Stack 製品用）

`https://docs.m5stack.com/en/download`

動作確認

`sf doctor` で環境診断 — すべて OK になれば完了

macOS / Linux ユーザー

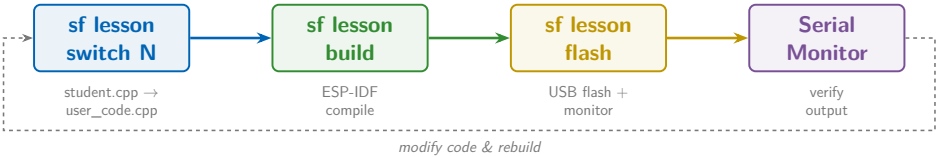
`install.sh` + `setup_env.sh` を使用してください

今日のゴール / Today's Goal

目標

ワークショップファームウェアをビルド・書き込み・動作確認する

1. `sf lesson build` でビルド
2. `sf lesson flash` で書き込み
3. シリアルモニタで `"Hello StampFly!"` を確認



sf CLI とは / What is sf CLI?

- StampFly 開発専用のコマンドラインツール
- ESP-IDF のコマンドをシンプルに実行できる

コマンド	説明
`sf doctor`	環境診断
`sf lesson build`	ワークショップビルド
`sf lesson flash`	書き込み＋モニタ
`sf lesson switch N`	レッスン切り替え
`sf monitor`	シリアルモニタ

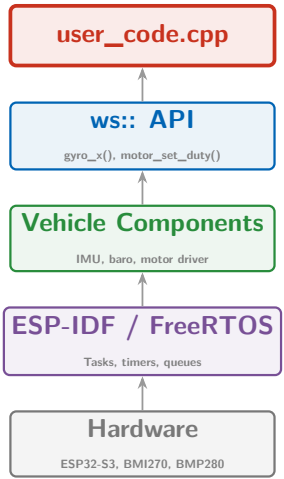
ワークショップの構造 / Workshop Structure

学生が実装する関数は 2 つだけ！

- `setup()` — 起動時に 1 回呼ばれる
- `loop_400Hz(dt)` — 400Hz（2.5ms 毎）で呼ばれる

- ハードウェアの複雑さは `ws::` 名前空間で隠蔽
- FreeRTOS, SPI/I2C, センサーフュージョンを意識する必要なし

ファームウェアの全体像 / Firmware Architecture



← You are here!

setup() / loop_400Hz()

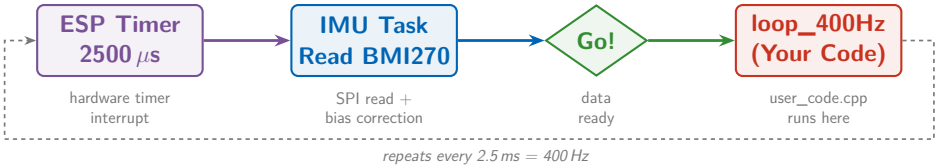
sensor read, filter, motor

xTaskCreate, xQueueSend

SPI, I2C, GPIO

- 皆さんのコードはここ！
  `user_code.cpp` を編集するだけ
- `ws::gyro_x()` と書くだけで
  SPI 通信→フィルタ→バイアス補正を
  全部やってくれる
- 下のレイヤーはワークショップ中
  意識する必要なし

400Hz ループの裏側 / Behind the 400Hz Loop



ポイント

`loop_400Hz(dt)` は常に IMU データ更新後に呼ばれる— 学生はタイマー管理不要、ただ関数を書くだけ！

実習: Hello StampFly / Hands-on

user_code.cpp

```cpp
#include "workshop_api.hpp"
void setup() {
    ws::print("Hello StampFly!");
}
void loop_400Hz(float dt) {
    // Nothing to do in Lesson 0
}
```

手順

1. `sf lesson switch 0`
2. 上のコードを `user_code.cpp` に書く
3. `sf lesson build && sf lesson flash`

作業の進め方 / Workflow

ファイル構造

```
firmware/workshop/
  ├── lessons/
  │   ├── lesson_00/student.cpp
  │   ├── lesson_01/student.cpp
  │   └── ...
  └── main/
      └── user_code.cpp ← 編集！
```

1. `sf lesson switch N`
   テンプレートを `user_code.cpp` にコピー
2. `user_code.cpp` を編集
3. `sf lesson build`
4. `sf lesson flash`

注意

`student.cpp` はテンプレートなので直接編集しない！
`switch` で何度でもリセットできる

コントローラのセットアップ / Controller Setup

コントローラのビルドと書き込み

1. コントローラを USB 接続
2. `sf build controller`
3. `sf flash controller`

確認

LCD に起動画面が表示されれば OK
Lesson 2 でペアリングして使用する

シミュレータで遊ぶ / Try the Simulator

USB HID モードに切替

1. 画面タッチでメニューを開く
2. `Comm:` を選択
3. `USB HID` に切替 → 自動再起動

操作	説明
スロットル	上昇
ロール / ピッチ	傾き
ヨー	回転

アクロモード（角速度制御）で飛行

コマンド

```
sf sim run
sf sim run -w ringworld
```

注意

終了後は `Comm:` を `ESP-NOW` に戻すこと（実機飛行用）

チェックポイント / Checkpoint

確認事項

- ☐ `sf doctor` がエラーなしで通る
- ☐ ビルドが成功し `"Hello StampFly!"` が表示される
- ☐ コントローラのビルド・書き込みが完了した
- ☐ シミュレータが起動し操縦を試した

次のレッスン

Lesson 1: モータ制御 — モータを回してプロペラの動きを確認

Lesson 1

モータ制御

Motor Control

StampFly Workshop

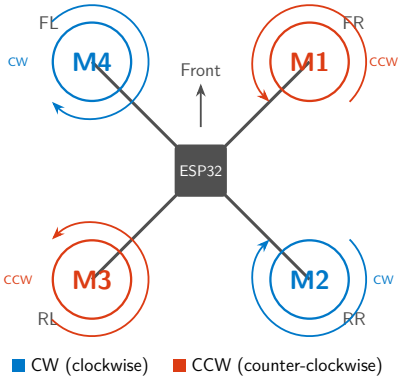
今日のゴール / Today's Goal

目標

モータの番号と配置を理解し、PWM で個別制御する

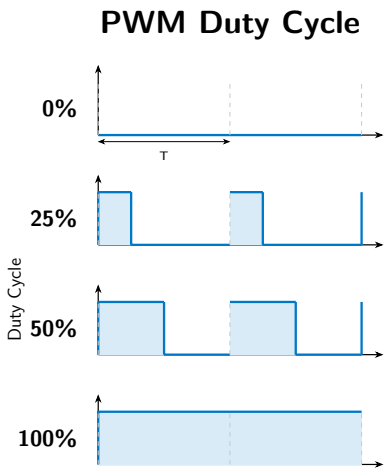
- モータ番号 M1–M4 の位置を覚える
- ARM / DISARM の仕組みを理解する
- PWM デューティ比でモータ回転数を制御

モータ配置 / Motor Layout



- 対角のモータが同じ方向に回転（トルクバランス）
- M1(FR), M3(RL) = CCW / M2(RR), M4(FL) = CW

PWM とは / What is PWM?



- **Pulse Width Modulation**（パルス幅変調）
- **Duty 0%** = 停止
- **Duty 100%** = 最大回転
- API: `motor_set_duty(id, duty)`
  id=1–4, duty=0.0–1.0

ARM / DISARM とは / What is ARM?

安全スイッチ / Safety Switch

- **ARM** = モーター出力を有効化（回転可能になる）
- **DISARM** = モーター出力を無効化（強制停止）
- コードから `arm()` で ARM できる

重要 / Important

`arm()` を呼ばないとモーターは回らない！
`setup()` 内で最初に呼ぶこと

モータ制御 API

関数	説明	引数
`arm()`	モーター出力を有効化	—
`disarm()`	モーター出力を無効化	—
`motor_set_duty(id, duty)`	個別モータ設定	id=1–4, duty=0.0–1.0
`motor_set_all(duty)`	全モーター一括	duty=0.0–1.0
`motor_stop_all()`	全モータ停止	—

安全注意 / Safety

duty は 0.15 以下で！ 機体を手で押さえてテスト！

実習: モータを順番に回す / Hands-on

user_code.cpp

```cpp
#include "workshop_api.hpp"
static uint32_t timer = 0;

void setup() {
    ws::print("Lesson 1: Motor Control");
    ws::arm();  // Enable motor output
}
void loop_400Hz(float dt) {
    timer++;
    // 800 ticks = 2 seconds at 400Hz
    int motor_id = (timer / 800) % 4 + 1;
    ws::motor_stop_all();
    ws::motor_set_duty(motor_id, 0.1f);
}
```

チェックポイント / Checkpoint

確認事項

- ☐ `arm()` なしではモータが回らないことを確認した
- ☐ M1 が FR（右前）のプロペラを回す
- ☐ M1→M2→M3→M4 の順に回転する

次のレッスン

Lesson 2: コントローラ入力 — スティックでモータを操作

Lesson 2

コントローラ入力

Controller Input

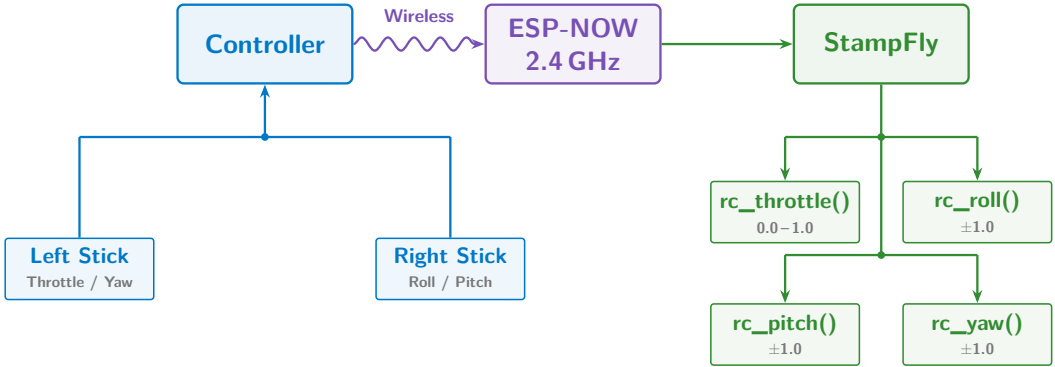
StampFly Workshop

今日のゴール / Today's Goal

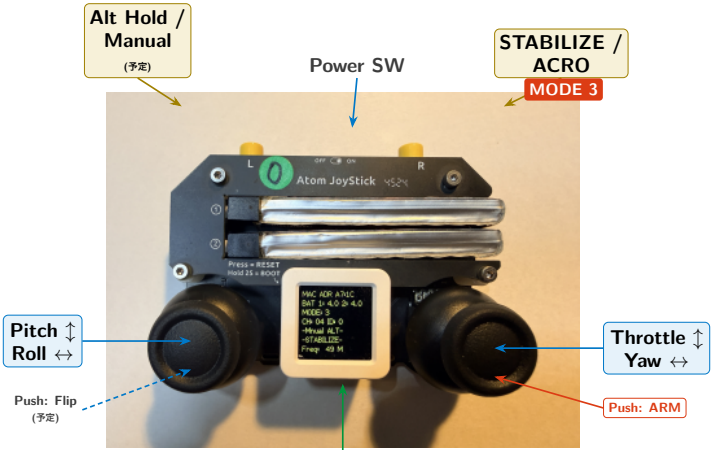
目標: スティック値を変数に読み取り、演算でモータを個別制御する

- ESP-NOW 無線通信の仕組みを理解
- チャンネル設定で他の受講生との混信を回避
- 変数と四則演算でスティック → モータ Duty を計算
- オープンループ手動操縦の限界を体感

ESP-NOW 通信フロー / Communication Flow



コントローラ各部 / Controller Layout (MODE 3)



コントローラ API

関数	説明	値域
`set_channel(ch)`	WiFi チャンネル設定	1, 6, 11
`rc_throttle()`	スロットル	0.0 – 1.0
`rc_roll()`	ロール	-1.0 – +1.0
`rc_pitch()`	ピッチ	-1.0 – +1.0
`rc_yaw()`	ヨー	-1.0 – +1.0
`is_armed()`	ARM 状態	true / false
`motor_set_duty(id, v)`	モータ個別制御	id=1–4, v=0.0–1.0

ペアリング手順 / Pairing

Step	操作
1	コントローラの **M5 ボタンを押しながら電源 ON** → LCD に “Pairing mode...” 表示
2	StampFly のボタンを **3 秒長押し** → 青 LED 高速点滅 + ビープ音
3	ペアリング完了 → ビープ音が鳴り、次回から自動接続

注意

- 教室では**一組ずつ**ペアリングする（ブロードキャスト通信のため近くの機体と干渉する可能性）
- うまくいかない場合: 両方を再起動して Step 1 からやり直す

チャンネル設定 / Channel Setting

教室で複数の StampFly を同時に飛ばすと、同じチャンネル同士で**混信**が起きる。チャンネルを分けて回避する。

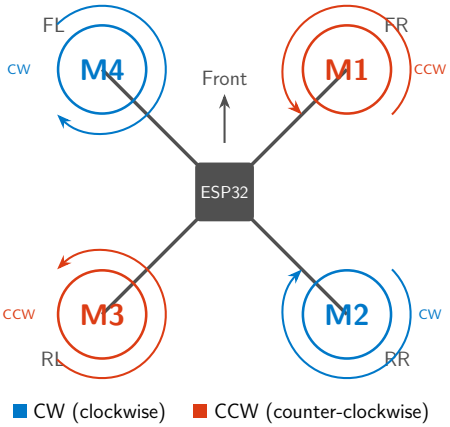
Ch	割り当て例
1	グループ A（緑）
6	グループ B（黄）
11	グループ C（赤）

設定方法

`set_channel(ch)` を `setup()` 内で呼ぶ

機体とコントローラで同じ番号にすること！

モータ配置とミキシング / Motor Layout & Mixing

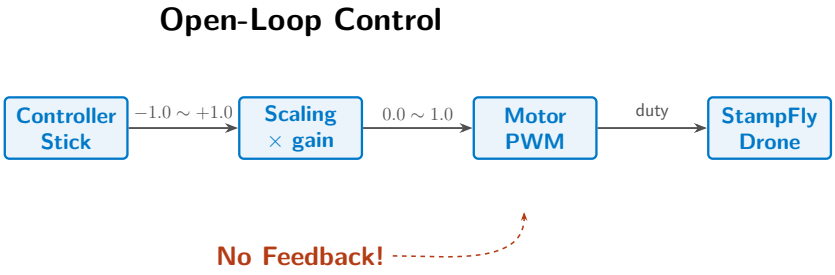


Motor	T	Roll	Pitch	Yaw
M1 FR	+	−	+	+
M2 RR	+	−	−	−
M3 RL	+	+	−	+
M4 FL	+	+	+	−

直感的に理解する

Pitch+ → 前傾 → 前 (M1,M4) 強 + 後 (M2,M3) 弱
Roll+ → 右傾 → 右 (M1,M2) 弱 + 左 (M3,M4) 強

オープンループ制御 / Open-Loop Control



問題点

フィードバックがないため、外乱（風・重心ずれ）に対応できない！
→ Lesson 5/8 で PID 制御を導入して解決

実習: 手動ミキシング (1/2) / Hands-on: Setup

user_code.cpp — setup

```cpp
#include "workshop_api.hpp"
static uint32_t tick = 0;
void setup() {
    ws::print("L2: Open-Loop Control");
    ws::arm();              // Enable motor output
    ws::set_channel(1);     // 1, 6, or 11
}
```

チャンネル番号

講師が指定した番号（1 / 6 / 11）に書き換えること！
コントローラも同じチャンネルに設定する

実習: 手動ミキシング (2/2) / Hands-on: Loop

user_code.cpp — loop

```
void loop_400Hz(float dt) {
    tick++;
    float t = ws::rc_throttle();
    float r = ws::rc_roll()  * 0.3f;
    float p = ws::rc_pitch() * 0.3f;
    float y = ws::rc_yaw()   * 0.3f;
    ws::motor_set_duty(1, t - r + p + y); // FR
    ws::motor_set_duty(2, t - r - p - y); // RR
    ws::motor_set_duty(3, t + r - p + y); // RL
    ws::motor_set_duty(4, t + r + p - y); // FL
    if (tick % 80 == 0)
        ws::print("T=%.2f R=%.2f P=%.2f Y=%.2f",
            t, r, p, y);
}
```

チェックポイント / Checkpoint

確認事項

- □ 指定チャンネルで混信なく通信できた
- □ コントローラとペアリングできた
- □ スロットルで全モータが均等に回る
- □ ピッチスティックで前後モータの回転差が出る

次のレッスン

Lesson 3: LED 制御 — システム状態を LED で可視化

Lesson 3

LED 制御

LED Control

StampFly Workshop

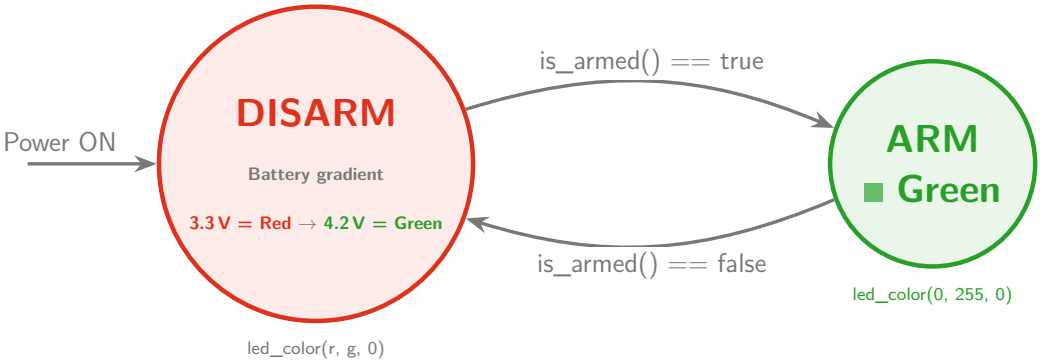
今日のゴール / Today's Goal

目標

LED でシステムの状態（ARM / DISARM）とバッテリー残量を可視化する

- WS2812 アドレサブル LED の制御
- 状態遷移の考え方（ステートマシン）
- バッテリー電圧のモニタリング

LED 状態遷移 / LED State Machine



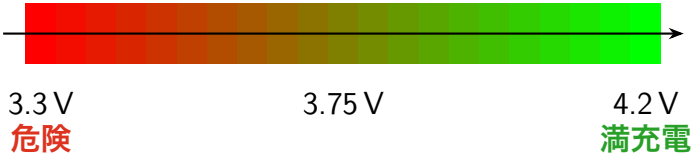
LED 制御 API

関数	説明	引数
`disable_led_task()`	システム LED 更新を無効化	—
`enable_led_task()`	システム LED 更新を再有効化	—
`led_color(r, g, b)`	LED 色設定	各 0–255
`is_armed()`	ARM 状態確認	true / false
`battery_voltage()`	バッテリー電圧	3.0–4.2 V

WS2812 LED

- RGB フルカラー（各チャンネル 8bit = 1677 万色）
- 1 本のデータ線で制御（ESP32 RMT ペリフェラル使用）
- `led_color()` は裏表両方の LED を同じ色に設定

バッテリー電圧と色 / Battery Voltage & Color



- LiPo バッテリー: 3.3V（空）〜 4.2V（満充電）
- **3.3V 以下で使用禁止！** バッテリーが損傷します

実習: ARM 状態で LED 色を変える / Hands-on

user_code.cpp

```cpp
#include "workshop_api.hpp"
void setup() {
    ws::print("Lesson 3: LED Control");
    ws::disable_led_task();
    ws::set_channel(1);
}
void loop_400Hz(float dt) {
    if (ws::is_armed()) {
        ws::led_color(0, 255, 0);  // Armed: green
    } else {
        float v = ws::battery_voltage();
        float lv = (v - 3.3f) / 0.9f; // 3.3-4.2V -> 0-1
        lv = (lv < 0) ? 0 : (lv > 1) ? 1 : lv;
        ws::led_color(255*(1-lv), 255*lv, 0);
    }
}
```

チェックポイント / Checkpoint

確認事項

- □ DISARM 時に赤〜緑のグラデーション表示
- □ ARM すると緑に変わる
- □ バッテリー残量に応じて色が変化する

次のレッスン

Lesson 4: IMU センサー — 加速度・ジャイロで姿勢を知る

Lesson 4

IMU センサー

IMU Sensor

StampFly Workshop

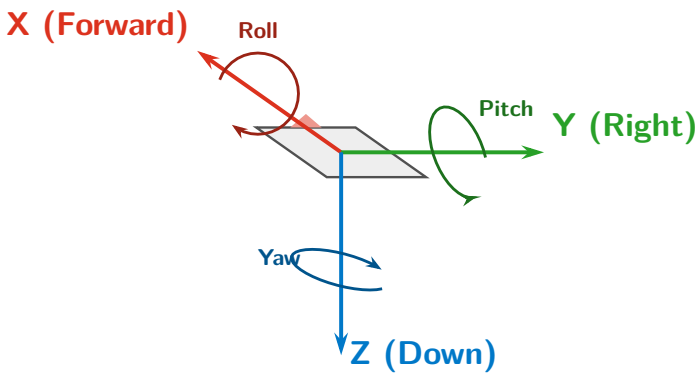
今日のゴール / Today's Goal

目標

IMU（ジャイロ+加速度）データを読み取り、シリアルモニタと WiFi テレメトリで確認する

- NED 座標系（北-東-下）の理解
- ジャイロスコープで角速度を取得
- 加速度センサで並進加速度を取得
- WiFi テレメトリでデータ取得・可視化

NED 座標系 / NED Coordinate System



BMI270 IMU — 6 軸（3 軸ジャイロ + 3 軸加速度）400 Hz　|　回転矢印 = ジャイロ正方向（右手の法則）

IMU API

関数	説明	単位
`gyro_x/y/z()`	角速度 (Roll/Pitch/Yaw)	rad/s
`accel_x/y/z()`	加速度 (X/Y/Z)	m/s^2

静止時の値

ジャイロ ≈ 0、加速度 Z $\approx -9.81\,m/s^2$（重力の反力、上向き）

実習: 6 軸読み取り + シリアル表示 / Hands-on

user_code.cpp

```cpp
#include "workshop_api.hpp"
static uint32_t tick = 0;
void setup() { ws::print("Lesson 4: IMU Sensor"); }
void loop_400Hz(float dt) {
    tick++;
    float gx=ws::gyro_x(), gy=ws::gyro_y(), gz=ws::gyro_z();
    float ax=ws::accel_x(),ay=ws::accel_y(),az=ws::accel_z();
    if (tick % 40 == 0)  // Print every 100ms
        ws::print("G=(%.3f,%.3f,%.3f) A=(%.2f,%.2f,%.2f)",
                  gx, gy, gz, ax, ay, az);
}
```

WiFi テレメトリ受信 / Receiving WiFi Telemetry

システムが IMU・姿勢・センサデータを自動的に 400 Hz で送信しています

手順

1. シリアルモニタで SSID を確認（`StampFly_XXXX`）
2. PC の WiFi で StampFly に接続（IP: `192.168.4.1`）
3. `sf log wifi -d 30` で 30 秒キャプチャ → CSV 自動保存

出力例

`logs/stampfly_wifi_20260303T143000.csv` に保存されました

`-o name.csv` でファイル名指定、`--no-save` で保存なし（統計のみ）

データの可視化 / Data Visualization

保存した CSV を可視化

```
sf log viz logs/stampfly_wifi_*.csv
```

モード	表示内容
`--mode all`	全パネル（デフォルト）
`--mode sensors`	IMU + 高度センサ
`--mode attitude`	姿勢推定（Roll/Pitch/Yaw）

- `--save plot.png` でファイル保存可能
- StampFly を手で揺らしながらキャプチャし、ジャイロの変化をグラフで確認しよう

チェックポイント / Checkpoint

確認事項

- □ シリアルモニタでジャイロ・加速度の値を確認できる
- □ 静止時に accel_z ≈ -9.81 を確認（反力）
- □ `sf log wifi` でセンサデータを受信できる
- □ `sf log viz` でデータをグラフ表示できる

次のレッスン

Lesson 5: レート P 制御 — ジャイロフィードバックで初フライト

Lesson 5

レート P 制御 ＋ 初フライト

Rate P-Control + First Flight

StampFly Workshop

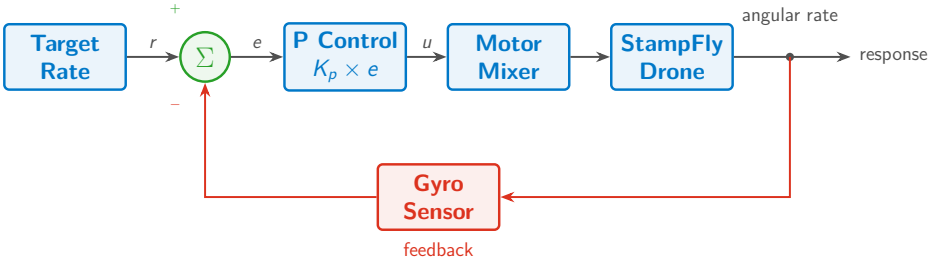
今日のゴール / Today's Goal

目標

比例フィードバック制御を実装し、初の制御飛行を行う

- 閉ループ制御の考え方
- P 制御で角速度を安定化
- ARM/DISARM による安全管理

閉ループ制御 / Closed-Loop Control



制御 API

関数	説明	値域
`gyro_x/y/z()`	ジャイロ角速度	rad/s
`rc_roll()`	ロールスティック	$-1.0 \sim +1.0$
`rc_pitch()`	ピッチスティック	$-1.0 \sim +1.0$
`rc_yaw()`	ヨースティック	$-1.0 \sim +1.0$
`rc_throttle()`	スロットル	$0.0 \sim 1.0$
`motor_mixer(T,R,P,Y)`	モーターミキサー	—
`is_armed()`	ARM 状態確認	true/false

実習: 3軸 P 制御 / Hands-on

user_code.cpp

```cpp
#include "workshop_api.hpp"
void setup() { ws::print("Lesson 5: Rate P-Control"); }
void loop_400Hz(float dt) {
    if (!ws::is_armed()) {
        ws::motor_stop_all(); ws::led_color(50,0,0); return;
    }
    ws::led_color(0, 50, 0);  // Green = ARM
    float Kp_rp=0.5f, Kp_yaw=2.0f, rmax=1.0f, ymax=5.0f;
    float re = ws::rc_roll()*rmax  - ws::gyro_x();
    float pe = ws::rc_pitch()*rmax - ws::gyro_y();
    float ye = ws::rc_yaw()*ymax   - ws::gyro_z();
    ws::motor_mixer(ws::rc_throttle(),
        Kp_rp*re, Kp_rp*pe, Kp_yaw*ye);
}
```

安全注意 / Safety

フライト前チェック

- □ **保護メガネを着用**
- □ **低スロットルから徐々に上げる**（最初は 30% 以下）
- □ プロペラの回転方向を確認（M1=CCW, M2=CW, M3=CCW, M4=CW）
- □ 異常時はすぐにスロットルを下げて DISARM

チェックポイント / Checkpoint

確認事項

- □ ARM 後スロットルを上げて安定ホバリング
- □ スティック操作で機体が応答する
- □ DISARM で即座にモーターが停止する

次のレッスン

Lesson 6: システムモデリング — 伝達関数からゲインを設計

Lesson 6

システムモデリング

System Modeling

StampFly Workshop

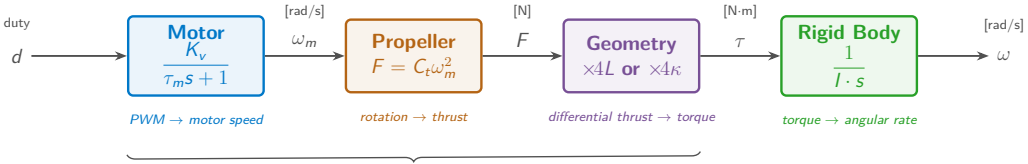
今日のゴール / Today's Goal

目標

プラント伝達関数を導出し、モデルに基づいて P ゲインを設計する

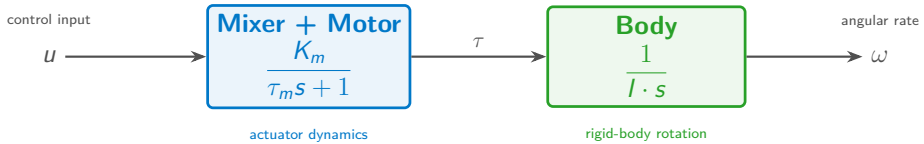
1. モータ+機体のプラント伝達関数 $G_p(s)$ を理解
2. P 制御の閉ループ特性（ω_n, ζ）を導出
3. ζ から K_p を設計 — L05 の初フライト経験をモデルで裏付け

物理メカニズム / Physical Mechanism



duty 信号がどうやって機体の角速度になるか — 4 つのステージ

プラントモデル（簡略化） / Plant Model (Simplified)



$$G_p(s) = \frac{K}{s(\tau_m s + 1)}, \qquad K = \frac{K_m}{I}$$

ホバー近傍で線形化 → 2 ブロックに集約

- **Mixer + Motor**: duty → トルク— 1 次遅れ $K_m/(\tau_m s + 1)$
- **Body**: トルク → 角速度— 積分器 $1/(I \cdot s)$

τ_m と K_m の導出 / Deriving τ_m and K_m

モータ時定数 τ_m

DC モータの電気・機械モデルから:

$$\tau_m = \frac{J_{mp} \cdot R_m}{K_e^2 + D_m R_m}$$

J_{mp}	2.01e-8 kg·m^2	回転子慣性
R_m	0.34 Ω	巻線抵抗
K_e	6.13e-4 V/(rad/s)	逆起電力定数
D_m	3.69e-8	粘性摩擦
τ_m	**0.018 s** ≈ **0.02 s**	

実効トルクゲイン K_m

K_f: ホバー点で duty を Δd 変えたときの 1 モータ推力変化量

$$K_f = \left.\frac{dF}{dd}\right|_{hover}, \quad K_m = \begin{cases} 4L \cdot K_f & \text{Roll/Pitch} \\ 4\kappa \cdot K_f & \text{Yaw} \end{cases}$$

K_f	0.010 N/duty	同定値
L	0.023 m	アーム長
κ	9.71e-3 m	トルク推力比
$K_{m,rp}$	**9.3e-4** N·m	Roll/Pitch
$K_{m,yaw}$	**3.9e-4** N·m	Yaw

実効プラントゲイン: $K = K_m / I \quad (K_{roll} = 102,\ K_{pitch} = 70,\ K_{yaw} = 19\ \text{rad/s}^2)$

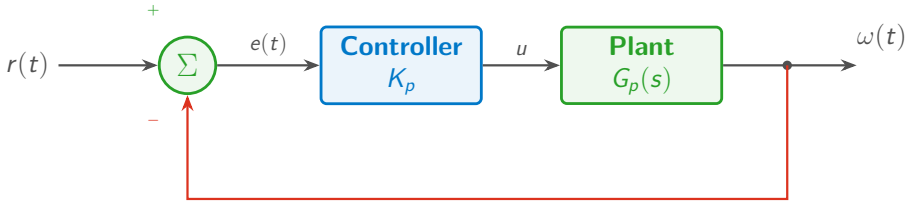
パラメーター一覧 / Parameter Summary

パラメータ	記号	Roll	Pitch	Yaw
慣性モーメント	I	9.16e-6	13.3e-6	20.4e-6 $\mathrm{kg \cdot m^2}$
トルクゲイン	K_m	9.3e-4	9.3e-4	3.9e-4 N·m
モータ時定数	τ_m		0.02 s（共通）	
プラントゲイン	$\boldsymbol{K = K_m / I}$	**102**	**70**	**19** $\mathrm{rad/s^2}$

なぜ軸ごとに K が違う？

- Roll/Pitch: 同じ K_m だが $I_{yy} > I_{xx}$ → Pitch の方が遅い
- Yaw: $\kappa \ll L$ なので K_m 自体が小さい → さらに遅い

P 制御の閉ループ / P Control Closed Loop



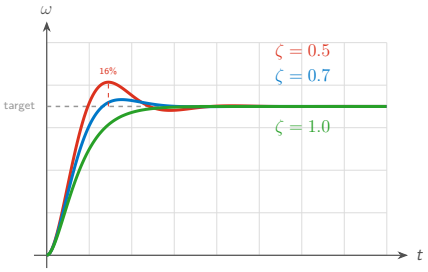
$$G_{cl}(s) = \frac{K_p K}{\tau_m s^2 + s + K_p K}$$

閉ループ伝達関数

$$G_{cl}(s) = \frac{K_p K}{\tau_m s^2 + s + K_p K}$$

⟵ **2 次系！**

2次系の特性 / Second-Order Characteristics



固有振動数と減衰比

$$\omega_n = \sqrt{\frac{K_p K}{\tau_m}}, \quad \zeta = \frac{1}{2\sqrt{K_p K \tau_m}}$$

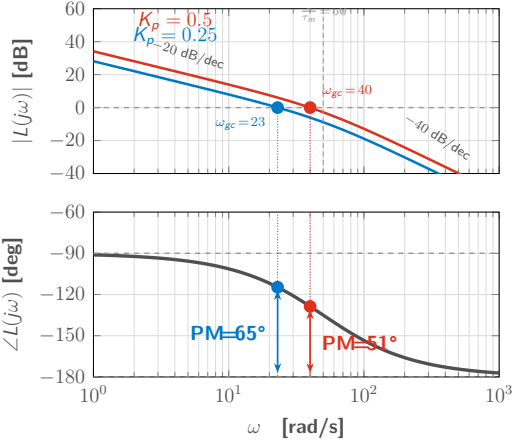
設計式: ζ から K_p を逆算

$$K_p = \frac{1}{4\zeta^2 K \tau_m}$$

オーバーシュート量 $\approx e^{-\pi\zeta/\sqrt{1-\zeta^2}}$

($\zeta = 0.5 \to 16\%$, $\zeta = 0.7 \to 5\%$)

開ループボード線図 / Open-Loop Bode Plot



開ループ伝達関数

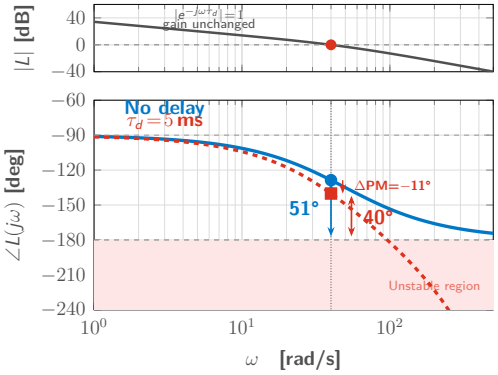
$$L(s) = K_p \cdot G_p(s) = \frac{K_p K}{s(\tau_m s + 1)}$$

位相余裕 PM

- ω_{gc}: $|L(j\omega_{gc})| = 1$ となる周波数
- $\mathrm{PM} = 90° - \arctan(\tau_m \omega_{gc})$

K_p を上げると $\omega_{gc} \uparrow \rightarrow$ PM $\downarrow \rightarrow$ 振動的に

無駄時間の影響 / Dead Time Effect



制御ループの無駄時間

$\tau_d \approx 5\,\mathrm{ms}$（センサ処理 + 制御演算 + PWM 更新）

$$L_{delay}(s) = L(s) \cdot e^{-\tau_d s}$$

	モデル	実機	
$K_p = 0.5$	PM=51°	PM≈40°	60° 未達!
$K_p = 0.25$	PM=65°	PM≈59°	ギリギリ

実用上 PM $\geq 60°$ が必要

$\zeta = 0.7$ 設計でも遅延込みでボーダーライン→「モデル上は安定」でも実機で振動する理由

ゲイン設計表 / Gain Design Table

	K	$K_p = 0.5$ の ζ	$\zeta = 0.7$ の K_p	$\zeta = 1.0$ の K_p
Roll	102	0.50	0.25	0.12
Pitch	70	0.60	0.36	0.18
Yaw	19	1.15	1.34	0.66

読み取り

- L05 の $K_p = 0.5$ は Roll で $\zeta = 0.50$（やや振動的）
- Yaw は $\zeta > 1$（過減衰 = 応答が遅い）— 軸ごとに K_p を変えるべき理由！

実習: モデルベースゲイン設計 / Hands-on

user_code.cpp (excerpt)

```
// Model-based Kp design (target zeta = 0.7)
const float tau_m = 0.02f;
const float K_roll  = 102.0f;
const float K_pitch =  70.0f;
const float K_yaw   =  19.0f;
float zeta = 0.7f;

float Kp_roll  = 1.0f/(4*zeta*zeta*K_roll *tau_m);
float Kp_pitch = 1.0f/(4*zeta*zeta*K_pitch*tau_m);
float Kp_yaw   = 1.0f/(4*zeta*zeta*K_yaw  *tau_m);

// Apply per-axis Kp in control loop
float roll_cmd  = Kp_roll  * (target_roll  - ws::gyro_x());
float pitch_cmd = Kp_pitch * (target_pitch - ws::gyro_y());
float yaw_cmd   = Kp_yaw   * (target_yaw   - ws::gyro_z());
```

チェックポイント / Checkpoint

確認事項

- □ $G_p(s) = K/(s(\tau_m s + 1))$ を説明できる
- □ モデルベース K_p と L05 の $K_p = 0.5$ を比較飛行
- □ Roll/Pitch/Yaw の ζ 差を体感で理解

次のレッスン

Lesson 7: システム同定 — フライトデータでモデルを検証

Lesson 7

システム同定

System Identification

StampFly Workshop

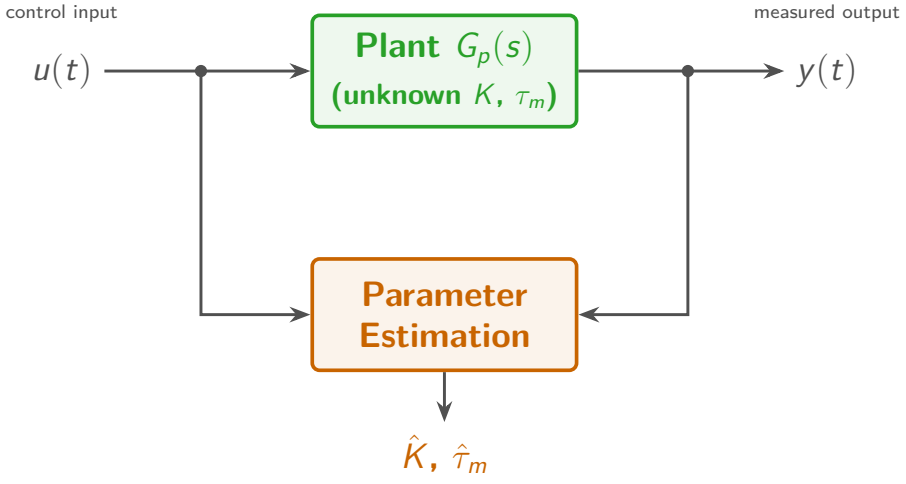
今日のゴール / Today's Goal

目標

フライトデータからプラントモデル $G_p(s) = \dfrac{K}{s\,(\tau_m s + 1)}$ のパラメータ K, τ_m を同定する

- システム同定（SysID）の考え方
- WiFi テレメトリでデータ取得
- `sf sysid fit` でモデルフィッティング
- 同定値と L6 理論値を比較

システム同定とは / What is System Identification?



WiFi テレメトリ / WiFi Telemetry

WiFi テレメトリ（自動送信）

システムが IMU・姿勢・センサデータを 400 Hz で自動送信
`sf log wifi` で受信 → CSV 保存 → `sf log viz` で可視化

コマンド	説明
`sf log wifi`	WiFi テレメトリ取得（CSV 保存）
`sf log wifi -o data.csv`	ファイル名指定で保存
`sf log analyze data.csv`	フライトデータ分析
`sf log viz data.csv`	波形可視化

プラント入出力の復元 / Reconstructing Plant I/O

テレメトリの記録内容

`ctrl_roll`	スティック入力
`gyro_x`	角速度（計測値）

復元手順（K_p 既知）

1. target $=$ ctrl $\times$ rate_max
2. $u(t) = K_p \times (\text{target} - \text{gyro})$
3. $y(t) = \text{gyro}$

なぜ開ループ同定が可能?

K_p が既知なら、プラントへの入力 $u(t)$ を計算できる。閉ループデータでも、開ループモデルを直接フィッティングできる。

パラメータ	値
K_p	L5 の値（例: 0.5）
`rate_max`	1.0 rad/s

フィッティングの仕組み / How Fitting Works

最小二乗法によるパラメータ推定

候補の K, τ_m でモデル出力 $\hat{y}(t)$ をシミュレーションし、実測 $y(t)$ との誤差が最小になる K, τ_m を探す。

1. 復元した $u(t)$ をモデルに入力: $u(t) \xrightarrow{G_p(s)} \hat{y}(t)$
2. 実測 $y(t)$ との二乗誤差を計算: $J(K, \tau_m) = \sum \left(\hat{y}(t) - y(t)\right)^2$
3. J を最小化する K, τ_m を数値最適化で求める

`sf sysid fit` が自動で行うこと

- ホバリング区間の自動検出
- データを短いセグメントに分割してフィッティング
- 結果の統計処理（中央値・不確かさ）

実習: データ取得 / Hands-on: Data Acquisition

L5 の P 制御で飛行し、テレメトリデータを取得する

同定には K_p と `rate_max` の値を記録しておくこと

1. `sf lesson switch 7` でテンプレートをコピー
2. `user_code.cpp` に K_p（例: 0.5）をセット
3. ビルド & 書き込み: `sf lesson build` → `sf lesson flash`
4. 離陸し、スティック操作でロール・ピッチ入力を入れる
5. PC でデータ受信:

ターミナル

```
1 sf log wifi -o flight.csv
```

sf sysid fit / Model Fitting Tool

コマンド

```
sf sysid fit flight.csv --kp 0.5 --plot
```

出力例

```
=== Plant Model Identification ===
  Model: G_p(s) = K / (s * (tau_m * s + 1))
  Roll  K= 98.5 (ref:102.0, err:3.4%) tau_m=0.019 R2=0.94
  Pitch K= 72.1 (ref: 70.0, err:3.0%) tau_m=0.021 R2=0.91
```

`--kp 0.5`	飛行時の K_p（**必須**）
`--axis roll`	特定軸のみ分析
`--rate-max 1.0`	`rate_max`（既定=1.0, yaw=5.0）
`-o result.yaml`	結果をファイルに保存

モデル検証 / Model Verification

軸	K (同定)	K (L6 理論)	τ_m (同定)	τ_m (理論)	R^2
Roll	?	102.0	?	0.020	?
Pitch	?	70.0	?	0.020	?
Yaw	?	19.0	?	0.020	?

設計 K_p の計算（$\zeta = 0.7$）

$$K_p = \frac{1}{4\zeta^2 K \tau_m}$$

考察

- 同定値と理論値のずれ → モデル誤差・無駄時間が原因
- 同定 K, τ_m で設計した K_p は L6 の理論値と近いか?

チェックポイント / Checkpoint

確認事項

- ☐ `sf log wifi` でフライトデータを取得できた
- ☐ `sf sysid fit` で K, τ_m を同定した
- ☐ 同定値と L6 理論値を比較した

次のレッスン

Lesson 8: PID 制御 — モデルに基づく I/D 項の追加で制御を改善

Lesson 8

PID 制御

PID Control

StampFly Workshop

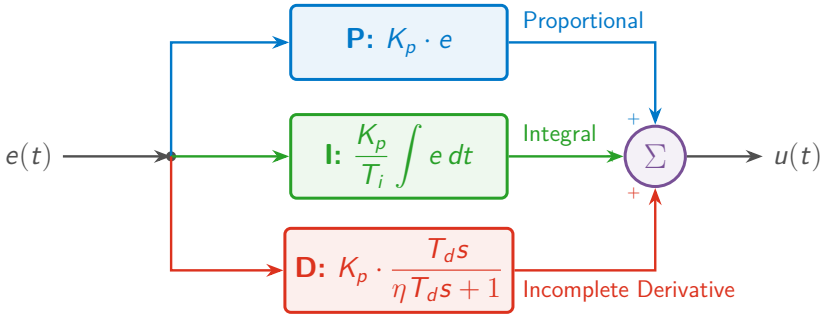
今日のゴール / Today's Goal

目標

モデルベースの K_p を起点に、I 項・D 項を追加して制御を改善する

- P 制御の限界: モデル不確かさ・外乱に弱い
- 工学形式 $K_p/T_i/T_d$ とは
- 不完全微分: ノイズに強い微分項
- ログから調整する方法

PID 制御器 / PID Controller



$$C(s) = K_p \left(1 + \frac{1}{T_i s} + \frac{T_d s}{\eta T_d s + 1} \right)$$

なぜ不完全微分か / Why Incomplete Derivative?

問題

- ジャイロにはノイズがある
- 理想微分 $\frac{de}{dt}$ はノイズを増幅
- 高周波振動 $\rightarrow$ モータに悪影響

解決策

- 不完全微分フィルタで高周波カット
- $\eta = 0.1 \sim 0.2$
- 小 $\rightarrow$ フィルタ弱い
- 大 $\rightarrow$ フィルタ強い

トレードオフ

微分の効き（応答速度）$\longleftrightarrow$ ノイズ除去（安定性）

工学形式パラメータ / Engineering Form

パラメータ	意味	効果
K_p	比例ゲイン	大 $\rightarrow$ 応答速い、大きすぎ $\rightarrow$ 振動
T_i	積分時間 [s]	小 $\rightarrow$ 積分強い（偏差早く消える）、小さすぎ $\rightarrow$ ワインドアップ
T_d	微分時間 [s]	大 $\rightarrow$ 微分強い（振動抑制）、大きすぎ $\rightarrow$ ノイズ増幅
η	フィルタ係数	$0.1 \sim 0.2$、大 $\rightarrow$ ノイズに強い

教科書形式との対応

$$K_i = K_p / T_i\,, \quad K_d = K_p \cdot T_d$$

ログから調整する / Log-Based Tuning

ログで見える症状	原因	調整
振動が収まらない	K_p 過大	$K_p \downarrow$
応答が遅い	K_p 過小	$K_p \uparrow$
定常偏差が残る	I 項不足	$T_i \downarrow$（積分強化）
オーバーシュート大	D 項不足	$T_d \uparrow$（微分強化）
高周波ノイズ	η 過小	$\eta \uparrow$（$0.1 \to 0.2$）
ゆっくり発散	ワインドアップ	$T_i \uparrow$

ヒント

L7 で取得したテレメトリデータを見ながら調整する

推奨ゲイン / Recommended Gains

軸	K_p	T_i [s]	T_d [s]	η
Roll	0.25	1.67	0.01	0.125
Pitch	0.36	1.67	0.01	0.125
Yaw	1.34	4.0	0.005	0.125

K_p は L6 のモデルベース設計値（$\zeta = 0.7$）を使用

チューニング手順

1. T_i=0, T_d=0 にして K_p を調整（振動しない最大値）
2. T_i を大きい値 (5.0) から徐々に下げる
3. T_d を小さい値 (0.001) から徐々に上げる

実習 Step 1: 理想 PID / Ideal PID

user_code.cpp (roll axis excerpt)

```
float Kp=0.25f, Ti=1.67f, Td=0.01f;
float integral=0, prev_err=0;
// inside loop_400Hz(dt)
float err = target - ws::gyro_x();
float P = Kp * err;
if (Ti > 0) integral += (dt/(2*Ti))*(err+prev_err);
float I = Kp * integral;
float D = Kp*Td*(err - prev_err)/dt; // ideal D
prev_err = err;
float roll_out = P + I + D;
```

やってみよう

飛ばしてみて → 高周波振動に注目

実習 Step 2: 不完全微分 / Incomplete Derivative

user_code.cpp (D term replacement)

```
// Replace ideal D with incomplete derivative filter
float eta = 0.125f;
float d_filt = 0;  // persistent state
// inside loop_400Hz(dt)
float alpha = 2*eta*Td / dt;
float a = (alpha - 1) / (alpha + 1);
float b = 2*Td / ((alpha + 1) * dt);
d_filt = a * d_filt + b * (error - prev_err);
float D = Kp * d_filt;  // Filtered!
```

やってみよう

再び飛ばして → 高周波振動が減ったことを確認

チェックポイント / Checkpoint

確認事項

- □ 理想微分 vs 不完全微分で高周波振動の違いを確認
- □ 定常偏差がなくなった（P のみと比較）
- □ ゲイン調整表を使って応答を改善できた

次のレッスン

Lesson 9: 姿勢推定 — センサフュージョンで角度を推定

Lesson 9

姿勢推定

Attitude Estimation

StampFly Workshop

今日のゴール / Today's Goal

目標

相補フィルタを実装し、ESKF の推定値と比較する

- ジャイロ積分のドリフト問題を理解
- 相補フィルタで加速度センサとジャイロを融合
- Teleplot で CF vs ESKF をリアルタイム比較

姿勢の表現 — ZYX オイラー角

オイラー角とは

3 次元の姿勢（回転）を 3 つの角度で表現する方法の一つ

記号	名称	回転軸	意味
ϕ	Roll（ロール）	X'' 軸	左右の傾き
θ	Pitch（ピッチ）	Y' 軸	前後の傾き
ψ	Yaw（ヨー）	Z 軸	機首方向

StampFly の規約: ZYX オイラー角

回転順序: Z 軸 $(\psi) \rightarrow Y'$ 軸 $(\theta) \rightarrow X''$ 軸 (ϕ)

航空宇宙分野で標準的な **Tait–Bryan 角**。ジンバルロック $(\theta = \pm 90^\circ)$ に注意。

オイラー角微分とジャイロ角速度の関係

ジャイロの測定値 $\boldsymbol{\omega}$ からオイラー角の微分を求める

ジャイロはボディ座標系の角速度 $\boldsymbol{\omega} = [\omega_x,\ \omega_y,\ \omega_z]^T$ を測定する。オイラー角の微分とは**一致しない**。現在の姿勢に依存する変換が必要:

$$\begin{aligned}
\dot{\phi} &= \omega_x + \omega_y \sin\phi \tan\theta + \omega_z \cos\phi \tan\theta \\
\dot{\theta} &= \omega_y \cos\phi - \omega_z \sin\phi \\
\dot{\psi} &= \omega_y \left(\sin\phi \,/\, \cos\theta\right) + \omega_z \left(\cos\phi \,/\, \cos\theta\right)
\end{aligned}$$

離散積分（毎ステップ更新）

$$\phi_{k+1} = \phi_k + \dot{\phi}_k\,\Delta t, \quad \theta_{k+1} = \theta_k + \dot{\theta}_k\,\Delta t, \quad \psi_{k+1} = \psi_k + \dot{\psi}_k\,\Delta t$$

注意: 現在の (ϕ, θ) に依存 → 毎ステップ再計算。$\theta = \pm 90°$ で $\cos\theta = 0$（ジンバルロック）

加速度からのオイラー角算出 (1) — 座標変換

静止時、加速度センサは重力の**反力**を測定する

NED 座標系の重力 $[0,\, 0,\, g]^T$ に対し、センサ出力は $\mathbf{R}^T\,[0,\, 0,\, -g]^T$:

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \mathbf{R}^T \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} = \begin{bmatrix} g\sin\theta \\ -g\sin\phi\cos\theta \\ -g\cos\phi\cos\theta \end{bmatrix}$$

- a_x は**ピッチ角 θ のみ**に依存（ϕ を含まない）
- a_y と a_z は**ロール角 ϕ とピッチ角 θ の両方**に依存
- **ヨー角 ψ** はどの成分にも現れない（重力は鉛直 → ψ の情報なし）

前提: 機体が静止または等速運動中であること。加速中は重力以外の成分が混入しノイズとなる。

加速度からのオイラー角算出 (2) — Roll と Pitch の導出

Roll ϕ の算出: $(-a_y)$ と $(-a_z)$ の比

$(-a_y)/(-a_z)$ をとると $g\cos\theta$ が約分され θ が消える:

$$\frac{-a_y}{-a_z} = \frac{g\sin\phi\cos\theta}{g\cos\phi\cos\theta} = \frac{\sin\phi}{\cos\phi} = \tan\phi \qquad \Longrightarrow \qquad \boxed{\phi = \operatorname{atan2}(-a_y,\ -a_z)}$$

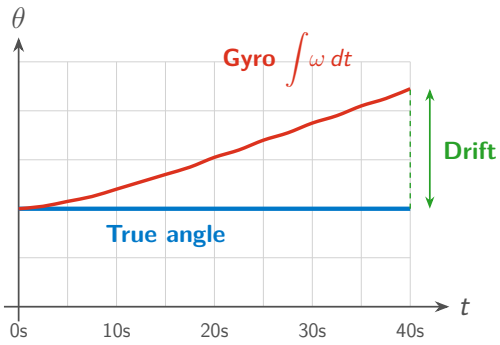
Pitch θ の算出: a_x と $\sqrt{a_y^2 + a_z^2}$

$a_y^2 + a_z^2 = g^2\cos^2\theta\,(\sin^2\phi + \cos^2\phi) = g^2\cos^2\theta$ より:

$$\frac{a_x}{\sqrt{a_y^2 + a_z^2}} = \frac{g\sin\theta}{g\cos\theta} = \tan\theta \qquad \Longrightarrow \qquad \boxed{\theta = \operatorname{atan2}\left(a_x,\ \sqrt{a_y^2 + a_z^2}\right)}$$

Yaw ψ: 加速度のみでは算出不可（磁気センサが必要） **長所:** ドリフトなし **短所:** 振動ノイズ大

ジャイロドリフト問題 / Gyro Drift Problem

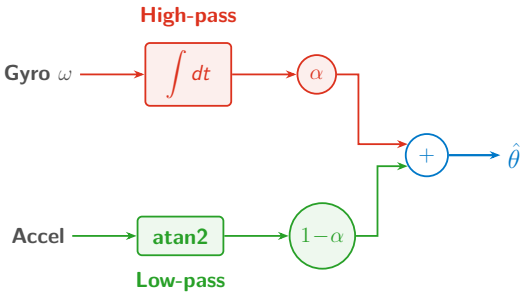


- ジャイロは**角速度** ω を測定
- 角度 $\theta = \int \omega \, dt$（積分が必要）
- バイアスが蓄積 → ドリフト増大

問題

ジャイロ単体では長時間の姿勢推定が不可能

相補フィルタ / Complementary Filter



数式

$$\hat{\theta}_k = \alpha\,(\hat{\theta}_{k-1} + \omega\,\Delta t) + (1-\alpha)\,\theta_{\text{accel}}$$

- $\alpha = 0.98$（ジャイロ 98% + 加速度 2%）
- ジャイロ: 短期精度◎ / 長期ドリフト☒
- 加速度: 長期安定◎ / 振動ノイズ☒

姿勢推定 API

関数	説明	単位
`gyro_x/y/z()`	角速度（Roll/Pitch/Yaw）	rad/s
`accel_x/y/z()`	加速度（X/Y/Z）	m/s^2
`estimated_roll/pitch/yaw()`	ESKF 推定姿勢角	rad
`print(fmt, ...)`	シリアル出力（Teleplot 対応）	—

実習で使う組み合わせ

ジャイロ + 加速度 → 相補フィルタで姿勢を自作

`estimated_roll()` → ESKF の推定値と比較

実習: 相補フィルタ + Teleplot / Hands-on

user_code.cpp (excerpt)

```cpp
#include "workshop_api.hpp"
#include <cmath>
static float cf_roll = 0.0f, cf_pitch = 0.0f;
void setup() { ws::print("Lesson 9: Attitude Estimation"); }
void loop_400Hz(float dt) {
    float gx = ws::gyro_x(), gy = ws::gyro_y();
    float ax = ws::accel_x(), ay = ws::accel_y(),
          az = ws::accel_z();
    float accel_roll  = atan2f(ay, az);
    float accel_pitch = atan2f(-ax, az);
    constexpr float alpha = 0.98f;
    cf_roll  = alpha*(cf_roll + gx*dt) + (1-alpha)*accel_roll;
    cf_pitch = alpha*(cf_pitch + gy*dt) + (1-alpha)*accel_pitch;
    // Teleplot output (VSCode Teleplot extension)
    ws::print(">cf_roll:%.1f", cf_roll*57.3f);
    ws::print(">eskf_roll:%.1f", ws::estimated_roll()*57.3f);
}
```

Teleplot によるリアルタイム可視化

Teleplot フォーマット

>変数名:値 をシリアルに出力するだけ

```
// Teleplot format: >name:value
ws::print(">cf_roll:%.2f", cf_roll * 57.3f);
ws::print(">cf_pitch:%.2f", cf_pitch * 57.3f);
ws::print(">eskf_roll:%.2f",
          ws::estimated_roll() * 57.3f);
```

セットアップ

1. VSCode 拡張: `alexnesnes.teleplot` をインストール
2. Teleplot パネルでシリアルポートを選択
3. 自動でグラフ化される

注意: 400 Hz 全 tick で出力するとシリアル帯域に負荷。4 tick 毎（100 Hz）にデシメーション推奨

Madgwick フィルタ / Madgwick Filter

勾配降下法によるクォータニオン推定

チューニングパラメータが β の **1つだけ** — シンプルかつ高精度

- 状態: $\mathbf{q} = [q_0,\, q_1,\, q_2,\, q_3]$
- 予測: $\dot{\mathbf{q}} = \frac{1}{2}\,\mathbf{q} \otimes \boldsymbol{\omega}$
- 補正勾配:
  $\nabla f = \mathbf{J}^T(\hat{\mathbf{q}},\, \hat{\mathbf{d}})\; \mathbf{f}(\hat{\mathbf{q}},\, \hat{\mathbf{d}})$

更新則

$$\mathbf{q}_{t+1} = \mathbf{q}_t + \Delta t \left(\frac{1}{2}\,\mathbf{q}_t \otimes \boldsymbol{\omega} - \beta\, \frac{\nabla f}{|\nabla f|} \right)$$

$\mathbf{f}$: 加速度/磁気の目的関数、$\mathbf{J}$: そのヤコビアン、β: 補正ゲイン（≈ 0.04）

EKF（拡張カルマンフィルタ）

非線形カルマンフィルタの5ステップ

状態ベクトル例: $\mathbf{x} = [\phi,\ \theta,\ \psi,\ b_{gx},\ b_{gy},\ b_{gz}]^T$

ステップ	数式	意味
状態予測	$\hat{\mathbf{x}}^- = f(\hat{\mathbf{x}}, \mathbf{u})$	ジャイロで姿勢を積分
共分散予測	$\mathbf{P}^- = \mathbf{F}\mathbf{P}\mathbf{F}^T + \mathbf{Q}$	不確かさの伝播
カルマンゲイン	$\mathbf{K} = \mathbf{P}^-\mathbf{H}^T(\mathbf{H}\mathbf{P}^-\mathbf{H}^T + \mathbf{R})^{-1}$	予測と観測の信頼度配分
状態更新	$\hat{\mathbf{x}} = \hat{\mathbf{x}}^- + \mathbf{K}(\mathbf{z} - h(\hat{\mathbf{x}}^-))$	観測で補正
共分散更新	$\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}^-$	不確かさを縮小

ESKF（誤差状態カルマンフィルタ）

StampFly で実際に使われている 15 状態 ESKF

名目状態 + 誤差状態の分離アーキテクチャ

誤差状態ベクトル（15 states）:

$$\delta\mathbf{x} = [\delta\mathbf{p},\ \delta\mathbf{v},\ \delta\boldsymbol{\theta},\ \delta\mathbf{b}_g,\ \delta\mathbf{b}_a]^T$$

- 予測: 名目状態はジャイロ/加速度で直接積分
- 誤差共分散のみ伝播

利点: 誤差状態は常に小さい → 線形化が正確

観測更新:

- 気圧 → 高度補正
- ToF → 対地高度補正
- 磁気 → ヨー補正
- 光学フロー → 速度補正

推定手法の比較 / Estimator Comparison

	相補フィルタ	Madgwick	EKF	ESKF
計算量	極小	小	中	中〜大
パラメータ数	1（α）	1（β）	$\mathbf{Q}, \mathbf{R}$ 行列	$\mathbf{Q}, \mathbf{R}$ 行列
推定対象	Roll/Pitch	R/P/Yaw	姿勢+バイアス	位置/速度/姿勢/バイアス
使用センサ	Gyro+Accel	Gyro+Accel+Mag	Gyro+Accel+Mag	全センサ（6 種）
精度	○	◎	◎	◎◎

StampFly のデフォルト: ESKF（15 状態）を使用。このレッスンでは相補フィルタを自作し、ESKF と比較して理解を深める。

チェックポイント / Checkpoint

確認事項

- □ 手で傾けると CF のロール・ピッチが変化する
- □ CF と ESKF の値が概ね一致する（Teleplot で確認）
- □ α を変えて応答の違いを観察
- □ Madgwick / EKF / ESKF の特徴を説明できる

次のレッスン

Lesson 10: ws:: API リファレンス

Lesson 10

ws:: API リファレンス

ws:: API Reference

StampFly Workshop

今日のゴール / Today's Goal

目標

ws:: API 全 43 関数をカテゴリ別に理解し、各関数の使い方を把握する

- ws:: API 全 43 関数をカテゴリ別に理解
- 各 API の使い方をコード例で確認
- 環境・距離センサ API で全センサに直接アクセス

ws:: API 概要 / API Overview

カテゴリ	概要	関数数
Motor Control	モータ制御・Arm/Disarm	7
RC Input	コントローラスティック入力	4
RC Buttons	ボタン・飛行モード	5
IMU Sensor	ジャイロ・加速度センサ	6
Env / Distance	気圧・磁気・ToF・光学フロー	10
Estimation	ESKF 姿勢・高度推定	4
LED	LED 色制御	3
Utility	時刻・電圧・通信・出力	4
合計		**43**

`#include "workshop_api.hpp"` で全関数を使用可能。全関数は `ws::` 名前空間に属する。

より深いアクセスには **StampFlyState** を使用（後半で解説）。

Motor Control API — Duty & Mixer

関数	引数	説明
`motor_set_duty(id, duty)`	id: 1–4, duty: 0–1	個別モータ duty 設定
`motor_set_all(duty)`	duty: 0–1	全モータ同一 duty
`motor_stop_all()`	—	全モータ即時停止
`motor_mixer(t, r, p, y)`	各 float	スラスト+姿勢ミキシング

```
1 // Individual motor control
2 ws::motor_set_duty(1, 0.3f);  // Motor 1 (FR) = 30%
3 ws::motor_stop_all();          // Emergency stop
4
5 // Mixer: thrust + attitude corrections
6 ws::motor_mixer(0.5f, 0.0f, 0.0f, 0.0f);  // Hover
```

Motor Control API — Arm / Disarm

関数	戻り値	説明
`arm()`	void	モータ出力を有効化（Arm）
`disarm()`	void	モータ出力を無効化（Disarm）
`is_armed()`	bool	Arm 状態を返す（`true` = Armed）

```
if (ws::rc_throttle() < 0.05f) ws::arm();

if (ws::is_armed()) {
    ws::motor_mixer(ws::rc_throttle(),
        ws::rc_roll(), ws::rc_pitch(), ws::rc_yaw());
} else {
    ws::motor_stop_all();
}
```

注意: `arm()` 前はモータコマンドを送っても回転しない

RC Input API

関数	範囲	説明
`rc_throttle()`	$[0.0, 1.0]$	スロットル（上がプラス）
`rc_roll()`	$[-1.0, 1.0]$	ロール（右がプラス）
`rc_pitch()`	$[-1.0, 1.0]$	ピッチ（前がプラス）
`rc_yaw()`	$[-1.0, 1.0]$	ヨー（右回転がプラス）

```
1 void loop_400Hz(float dt) {
2     float thr = ws::rc_throttle();  // 0.0 - 1.0
3     float rol = ws::rc_roll();      // -1.0 - 1.0
4     float pit = ws::rc_pitch();
5     float yaw = ws::rc_yaw();
6     ws::motor_mixer(thr, rol, pit, yaw);  // ACRO mode
7 }
```

RC Buttons / Mode API

関数	戻り値	説明
`rc_throttle_yaw_button()`	bool	左スティック押し込みボタン
`rc_roll_pitch_button()`	bool	右スティック押し込みボタン
`rc_stabilize_acro_mode()`	bool	ACRO モード判定（true = ACRO）
`rc_alt_mode()`	bool	高度保持モード判定
`rc_pos_mode()`	bool	位置保持モード判定

```
1 void loop_400Hz(float dt) {
2     if (ws::rc_stabilize_acro_mode()) {
3         acro_control(dt);       // ACRO: rate control
4     } else {
5         stabilize_control(dt);  // Stabilize: angle control
6     }
7 }
```

IMU Sensor API

関数	単位	説明
gyro_x/y/z()	rad/s	R/P/Y 角速度（BMI270）
accel_x/y/z()	m/s^2	X/Y/Z 加速度（NED 座標系）

```
void loop_400Hz(float dt) {
    float gx = ws::gyro_x();     // [rad/s] roll rate
    float gy = ws::gyro_y();     // [rad/s] pitch rate
    float gz = ws::gyro_z();     // [rad/s] yaw rate
    float az = ws::accel_z();    // [m/s^2] ~9.81 at rest

    ws::print(">gyro_x:%.3f", gx);
    ws::print(">accel_z:%.2f", az);
}
```

静止時: accel_z() ≈ 9.81（NED 座標系、下向き正）

Estimation API

関数	単位	説明
`estimated_roll()`	rad	ESKF ロール推定角
`estimated_pitch()`	rad	ESKF ピッチ推定角
`estimated_yaw()`	rad	ESKF ヨー推定角
`estimated_altitude()`	m	ESKF 推定高度（正 = 上）

```
void loop_400Hz(float dt) {
    float roll  = ws::estimated_roll();     // [rad]
    float pitch = ws::estimated_pitch();
    float alt   = ws::estimated_altitude(); // [m]
    // P control: level the drone
    ws::motor_mixer(ws::rc_throttle(),
        (0.0f-roll)*0.5f, (0.0f-pitch)*0.5f, ws::rc_yaw());
}
```

ESKF はジャイロ+加速度+気圧+ToF+磁気+光学フローを統合（L09 参照）

環境センサ API — 気圧・磁気

関数	単位	説明
`baro_altitude()`	m	気圧高度（BMP280）
`baro_pressure()`	Pa	気圧値（BMP280）
`mag_x()`	μT	磁気 X 成分（BMM150）
`mag_y()`	μT	磁気 Y 成分（BMM150）
`mag_z()`	μT	磁気 Z 成分（BMM150）

```
1 float alt = ws::baro_altitude();  // [m]
2 float p   = ws::baro_pressure();  // [Pa]
3 float mx  = ws::mag_x();          // [uT]
4 ws::print(">baro_alt:%.2f", alt);
```

環境センサ API — ToF・光学フロー

関数	単位	説明
`tof_bottom()`	m	下方 ToF 距離（VL53L3CX, 0–2 m）
`tof_front()`	m	前方 ToF 距離（-1 = 未接続）
`flow_vx()`	m/s	光学フロー速度 X（PMW3901）
`flow_vy()`	m/s	光学フロー速度 Y（PMW3901）
`flow_quality()`	0–255	光学フロー品質（高い = 良好）

```
float dist = ws::tof_bottom();     // [m] ground distance
float vx   = ws::flow_vx();        // [m/s]
uint8_t sq = ws::flow_quality();   // 0-255
ws::print(">tof_bottom:%.3f", dist);
```

LED Control API

関数	引数/戻り値	説明
`led_color(r, g, b)`	r,g,b: 0–255	LED 色を RGB 設定
`disable_led_task()`	void	システム LED タスク停止
`enable_led_task()`	void	システム LED タスク再開
`is_led_task_disabled()`	bool	LED タスク停止中か確認

```
void setup() {
    ws::disable_led_task();  // Take over LED control
}
void loop_400Hz(float dt) {
    float v = ws::battery_voltage();
    if (v > 3.8f)      ws::led_color(0, 255, 0);   // Green
    else if (v > 3.5f) ws::led_color(255, 165, 0); // Orange
    else               ws::led_color(255, 0, 0);   // Red
}
```

注意: `disable_led_task()` を先に呼ぶこと（LED 上書き防止）

Utility API

関数	戻り値	説明
`millis()`	uint32_t (ms)	起動からの経過時間
`battery_voltage()`	float (V)	バッテリー電圧
`print(fmt, ...)`	void	printf 形式でデバッグ出力
`set_channel(ch)`	void	WiFi チャネル設定（1, 6, 11）

```
void setup() { ws::set_channel(6); }
void loop_400Hz(float dt) {
    static uint32_t last = 0;
    uint32_t now = ws::millis();
    if (now - last >= 20) {  // 50 Hz decimation
        last = now;
        ws::print(">battery:%.2f", ws::battery_voltage());
    }
}
```

ハードウェアセンサ仕様 / Hardware Sensors

センサ	型番	サンプルレート	測定量
IMU	BMI270	400 Hz	加速度 + ジャイロ
気圧	BMP280	50 Hz	気圧 → 高度
磁気	BMM150	10–30 Hz	磁気ベクトル
ToF（下方）	VL53L3CX	30 Hz	対地距離（0–2 m）
ToF（前方）	VL53L3CX	30 Hz	前方距離（0–2 m）
光学フロー	PMW3901	100 Hz	対地速度

ws:: API で全センサに直接アクセス可能（`baro_altitude()`, `tof_bottom()` 等）。
より低レベルなアクセスには **StampFlyState** を使用。

ESKF は全センサを統合して姿勢・位置・速度を推定（Lesson 9 参照）。

Teleplot

Teleplot 形式 (>name:value) でシリアル出力すると、
VS Code 拡張 Teleplot がリアルタイムグラフを描画する。

```
void loop_400Hz(float dt) {
    static uint32_t tick = 0; tick++;
    if (tick % 8 != 0) return;  // 50 Hz decimation

    // Teleplot output (>name:value format)
    ws::print(">baro_alt:%.2f", ws::baro_altitude());
    ws::print(">tof_bottom:%.3f", ws::tof_bottom());
    ws::print(">eskf_alt:%.2f", ws::estimated_altitude());
}
```

セットアップ: VSCode 拡張 alexnesnes.teleplot をインストール→ Teleplot パネルでシリアルポートを選択 → 自動グラフ化

実習: 全センサ Teleplot 出力 / Hands-on

```
#include "workshop_api.hpp"
void setup() { ws::print("Lesson 10: Sensor API"); }
void loop_400Hz(float dt) {
    static uint32_t tick = 0; tick++;
    if (tick % 8 != 0) return;  // 50 Hz decimation
    // Environmental sensors (ws:: API)
    ws::print(">baro_alt:%.2f", ws::baro_altitude());
    ws::print(">mag_x:%.1f", ws::mag_x());
    ws::print(">tof_bottom:%.3f", ws::tof_bottom());
    ws::print(">flow_vx:%.3f", ws::flow_vx());
    // ESKF estimation
    ws::print(">eskf_alt:%.2f", ws::estimated_altitude());
    ws::print(">eskf_roll:%.1f", ws::estimated_roll()*57.3f);
}
```

Teleplot パネルでシリアルポートを選択し確認（前ページ参照）

チェックポイント / Checkpoint

確認事項

- ☐ ws:: API 全 43 関数のカテゴリと役割を把握した
- ☐ Motor / RC / IMU / Sensor / Estimation の使い方を理解した
- ☐ 気圧・磁気・ToF・光学フロー API で値を取得できた
- ☐ Teleplot で複数センサのグラフを同時表示した

次のレッスン

Lesson 11: 独自ファームウェア開発

Lesson 11

独自ファームウェア開発

Custom Firmware Development

StampFly Workshop

今日のゴール / Today's Goal

目標

`sf app new` で独自プロジェクトを作り、ネイティブ API で全センサにアクセスする

- `sf app new` で独自ファームウェアプロジェクトを作成
- テンプレートの構造（`app_main`, `ControlTask`）を理解
- `StampFlyState` で全センサ・推定値に直接アクセス
- Teleplot でセンサデータをリアルタイム可視化

開発フロー / Development Flow

注意: これまでのレッスンとやり方が変わります

ワークショップ（`firmware/workshop/`）の外に、独立したプロジェクトを作ります。

	L0–L10（これまで）	**L11（このレッスン）**
開始	`sf lesson switch N`	`sf app new my_drone`
編集先	`firmware/workshop/main/`	`firmware/my_drone/main/`
ファイル	`user_code.cpp`	`main.cpp`
ビルド	`sf lesson build`	`sf build my_drone`
書き込み	`sf lesson flash`	`sf flash my_drone -m`
API	ws:: ラッパー	ネイティブ（StampFlyState）

API 比較 / API Comparison

ws:: ワークショップ

- `ws::gyro_x()`, `ws::alt()`
- `ws::print(">tag:%.2f", v)`
- `setup()` / `loop_400Hz(dt)`
- 1 関数 = 1 センサ値

ネイティブ開発

- `state.getIMUData(a, g)`
- `printf(">tag:%.2f\n", v)`
- `app_main()` / `ControlTask()`
- 構造体で複数値を一括取得

ネイティブ = vehicle ファームウェアと同じ構成。あらゆるアルゴリズムを実装可能。

プロジェクト作成 / Create Project

```
sf app new my_drone     # firmware/my_drone/ が生成
sf build my_drone       # ビルド
sf flash my_drone -m    # 書き込み + モニタ
```

テンプレートの特徴

- vehicle と同じセンサタスク・初期化コードを再利用
- `ControlTask()` のみを独自実装
- IMU, 気圧, ToF, 光学フロー, モーター等すべて利用可能

プロジェクト構成 / Project Structure

```
firmware/my_drone/
  CMakeLists.txt          # ビルド設定
  main/
    CMakeLists.txt        # コンパイル対象の定義
    main.cpp              # ← ここを編集
```

ファイル	役割
CMakeLists.txt	vehicle/components と common を参照（全コンポーネント利用可能）
main/CMakeLists.txt	vehicle のタスク（IMU, Baro, ToF 等）と init.cpp を再利用
main/main.cpp	ユーザーが編集する唯一のファイル（ControlTask + コールバック）

テンプレート構造 / Template Structure

app_main() — ブートシーケンス

NVS → センサ初期化 → ESKF 起動 → タスク起動（自動生成済み）

ControlTask() — 400 Hz ユーザーコード ← ここを編集

センサ読み取り・制御計算・モーター出力を記述するメインループ

コールバック関数

- `onButtonEvent()` — ボタン押下で ARM / DISARM
- `handleControlInput()` — コントローラからの操縦入力を処理

ControlTask / 制御ループ

```
void ControlTask(void* pvParameters) {
    while (true) {
        xSemaphoreTake(g_control_semaphore, ...);
        // --- 1. センサ読み取り ---
        state.getIMUData(accel, gyro);     // 400 Hz
        state.getAttitudeEuler(r, p, y);   // ESKF 推定値
        state.getBaroData(alt, p);         // 50 Hz
        state.getToFData(bot, fnt);        // 30 Hz
        // --- 2. 制御計算（ユーザー実装） ---
        // PID, モーターミキシング等
        // --- 3. モーター出力 ---
        // g_motor.setThrust(0, thrust);
        // --- 4. Teleplot 出力（50 Hz） ---
        if (tick % 8 == 0) printf(...);
    }
}
```

セマフォで IMU タイマー（2500 μs）と同期。各センサは独立タスクで取得済み。

StampFlyState API

メソッド	ソース	取得データ
`getIMUData(a, g)`	BMI270	加速度 + ジャイロ
`getBaroData(alt, p)`	BMP280	高度 [m], 気圧 [Pa]
`getMagData(mag)`	BMM150	磁気 x,y,z [μT]
`getToFData(b, f)`	VL53L3CX	下方/前方距離 [m]
`getFlowData(vx, vy)`	PMW3901	速度 vx,vy [m/s]
`getAttitudeEuler(r, p, y)`	ESKF	roll, pitch, yaw [rad]
`getPowerData(v, i)`	INA3221	電圧 [V], 電流 [A]

`StampFlyState::getInstance()` でシングルトンを取得し、各メソッドを呼び出す。

実習: 全センサ Teleplot 可視化 / Hands-on

```cpp
auto& state = stampfly::StampFlyState::getInstance();
stampfly::Vec3 accel, gyro;
state.getIMUData(accel, gyro);
float alt, p;
state.getBaroData(alt, p);
float roll, pitch, yaw;
state.getAttitudeEuler(roll, pitch, yaw);

static uint32_t tick = 0; tick++;
if (tick % 8 == 0) {  // 50 Hz
    printf(">baro_alt:%.2f\n", alt);
    printf(">roll_deg:%.1f\n", roll * 57.3f);
    printf(">gyro_x:%.3f\n", gyro.x);
}
```

手順: `sf app new my_viz` → `firmware/my_viz/main/main.cpp` の ControlTask を編集 → `sf build my_viz` → `sf flash my_viz -m` → Teleplot

チェックポイント / Checkpoint

確認事項

- ☐ `sf app new` でプロジェクトを作成しビルドできた
- ☐ プロジェクト構成（3 ファイルの役割）を理解した
- ☐ `app_main()` と `ControlTask()` の役割を理解した
- ☐ StampFlyState で全センサ値を直接取得できた
- ☐ Teleplot でリアルタイムグラフを確認した

次のレッスン

Lesson 12: Python SDK プログラム飛行

Lesson 12

Python SDK プログラム飛行

Python SDK Programmatic Flight

StampFly Workshop

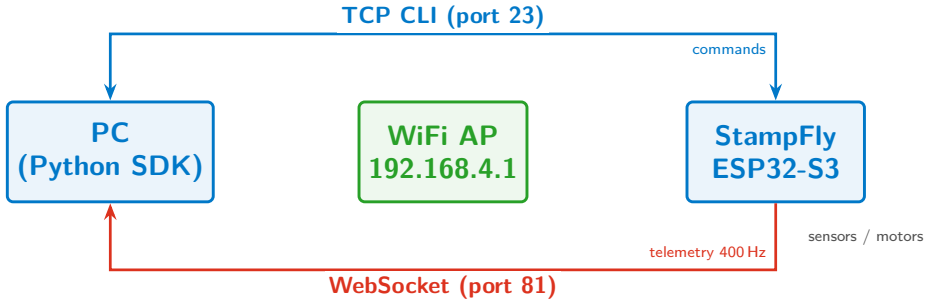
今日のゴール / Today's Goal

目標

Python SDK の設計思想と将来の自律飛行の姿を理解する

- SDK アーキテクチャ（TCP CLI + WebSocket）を理解
- Python API で離陸・移動・着陸をプログラム化
- Tello SDK 互換設計の利点を知る

SDK アーキテクチャ / SDK Architecture



TCP CLI (port 23)

コマンド送受信（takeoff, land...）

WebSocket (port 81)

テレメトリ受信（400 Hz RT）

Python SDK API

関数	説明	備考
`StampFly(host)`	インスタンス生成	デフォルト 192.168.4.1
`connect()`	WiFi 接続	TCP CLI + WS
`takeoff()`	離陸	ブロッキング
`land()`	着陸	ブロッキング
`move_forward(cm)`	前進	20–200 cm
`rotate_clockwise(deg)`	時計回り回転	1–360°
`send_rc_control(lr,fb,ud,yaw)`	RC 制御値送信	-100~$+100$
`get_telemetry()`	テレメトリ取得	400 Hz dict
`end()`	切断	リソース解放

Tello SDK との互換性 / Tello Compatibility

djitellopy 互換

Tello のコードをほぼそのまま StampFly に移植可能

- 同じ API 名（`takeoff`, `land`, `move_forward`...）
- `connect_or_simulate()` でオフライン開発も可能
- 大学の Tello 教材を流用可能

StampFly の利点

- 内部の PID ゲインを自由に変更可能
- テレメトリを 400 Hz で取得
- 制御理論の実験プラットフォーム
- ESP32-S3 のファームウェアも公開

コード例: 基本フライト / Basic Flight

flight_basic.py

```python
from stampfly import StampFly

drone = StampFly()
drone.connect()

drone.takeoff()
drone.move_forward(50)     # 50 cm forward
drone.rotate_clockwise(90)
drone.move_forward(50)
drone.land()

drone.end()
```

コード例: テレメトリ取得 / Telemetry

flight_telemetry.py

```python
import time
from stampfly import StampFly

drone = StampFly()
drone.connect()
drone.takeoff()

# Hover + read telemetry for 5 seconds
for _ in range(50):
    drone.send_rc_control(0, 0, 0, 0)
    t = drone.get_telemetry()
    print(f"alt={t.get('alt',0):.1f}")
    time.sleep(0.1)

drone.land()
drone.end()
```

開発ロードマップ / Development Roadmap

状態	機能
✓	TCP CLI 通信（port 23）
✓	WebSocket テレメトリ（port 81）
✓	ws:: ワークショップ API
✓	sf CLI ツール群
	Python SDK パッケージ
	connect_or_simulate()
	Jupyter Notebook 連携
	自律ミッション（ウェイポイント）

将来の姿

- Python でミッション記述
- Jupyter でデータ解析
- シミュレータ連携
- ROS2 ブリッジ

チェックポイント / Checkpoint

確認事項

- ☐ SDK アーキテクチャ（TCP + WebSocket）を理解した
- ☐ Python API の基本関数を把握した
- ☐ Tello 互換の設計意図を理解した
- ☐ 開発ロードマップを確認した

次のレッスン

Lesson 13: 精密着陸競技会 ルール説明

Lesson 13

精密着陸競技会

Precision Landing Competition

StampFly Workshop

競技会概要 / Competition Overview

精密着陸競技

パイロットは定位置から操縦し、3 m 先のヘリポートに精密着陸する

- **種目**: 精密着陸（パイロット定位置、3 m 先のヘリポートに着陸）
- **ヘリポート**: 約 40 cm × 40 cm
- **評価**: 着陸までのタイム（ベストタイム採用）

競技ルール詳細 / Competition Rules

項目	内容
種目	精密着陸（パイロット定位置操縦）
距離	パイロットからヘリポートまで 3 m
ヘリポート	40 cm × 40 cm
制限時間	60 秒
試行回数	3 回（ベストタイム採用）
判定	ARM → 離陸 → ヘリポート着陸までの時間
パイロット	定位置から動かない
失格条件	安全エリア外飛行、手による介入、パイロットの移動

注意: 安全第一！ 周囲に人がいないことを確認してから飛行

タイムスケジュール / Schedule

時間	内容
Day 4（本日午後）	
13:00–13:30	ルール説明（このスライド）
13:30–16:00	準備・自由練習・予選
Day 5（最終日）	
9:00– 9:15	ルール確認・機体チェック
9:15– 9:45	最終チューニング
9:45–10:45	競技本番
10:45–11:30	結果発表・表彰・振り返り

攻略のヒント / Tips for Success

PID ゲイン調整

- 安定性重視（急旋回よりも穏やかな応答）
- ヨー制御で方向を合わせる
- スロットル微調整で高度を安定させる

練習のポイント

- まず安定ホバリングを確立
- 前進 → 停止 → 降下の手順を練習
- バッテリ・プロペラを毎回チェック

テレメトリ活用: Teleplot でリアルタイムデータを確認し、ゲインを追い込む

チェックポイント / Checkpoint

準備確認

- □ 競技ルールと失格条件を理解した
- □ バッテリがフル充電されている
- □ PID ゲインの調整方法を把握している
- □ 安全チェックリストを確認した

さあ、練習を始めよう！

13:30 から自由練習 → 最高の精密着陸を目指そう