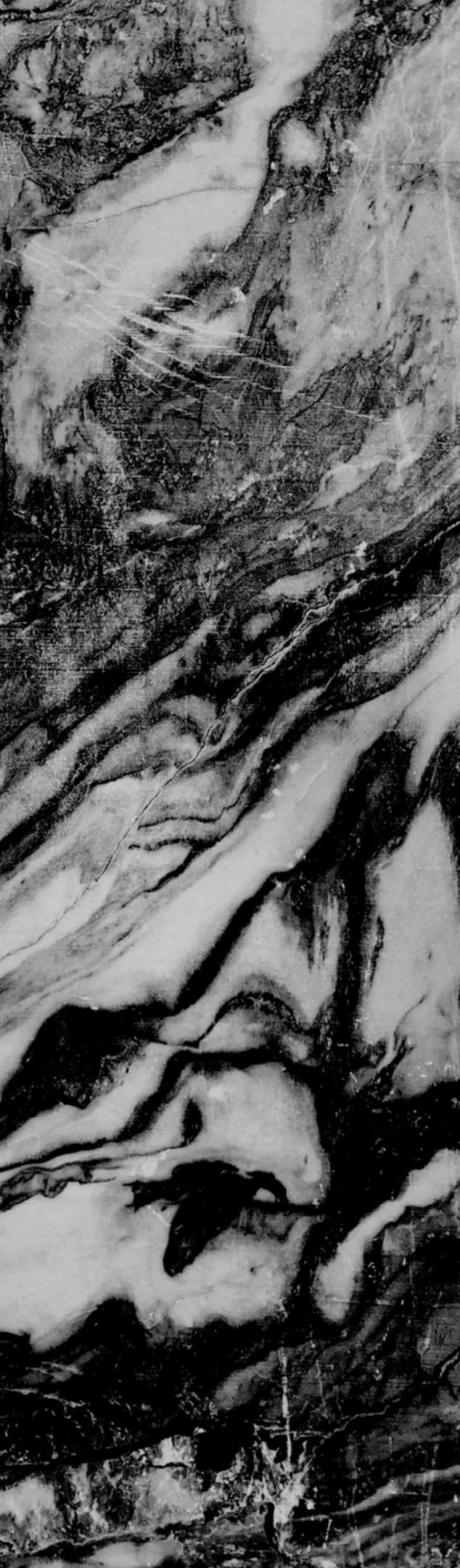

x86_64 ロボット開発環境下 での技術紹介&Tips的な何か

@yanötzvvv



制御概観

基本構造

センサ (*Transducer*)

マイコン
MCU

制御器 (*Controller*)

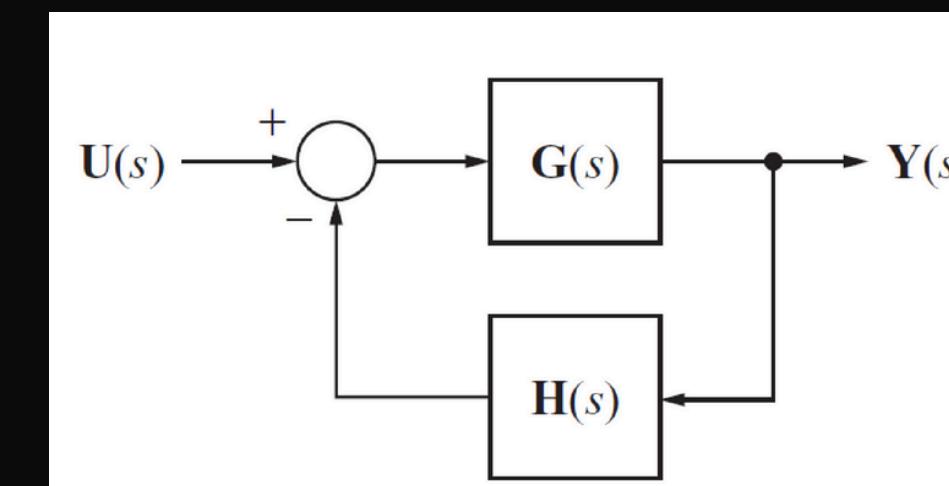
モータ (*Actuator*)

図示

Q. 制御屋は何をするか

- A. センサの情報を読み取り(状態取得)、
制御計算をして(PIDなど)、
アクチュエータへ出力する(指令)

FB Block Diagram



I. 環境

まずはNeofetch (自宅PC)

```
tatsv@tatsv
-----
OS: Arch Linux x86_64
Host: Thirdwave PC
Kernel: 6.17.8-zen1-1-zen
Uptime: 19 days, 12 hours, 12 mins
Packages: 2427 (pacman), 7 (flatpak)
Shell: fish 4.2.1
Resolution: 3840x2160
DE: GNOME 49.2
WM: Mutter
WM Theme: Orchis-Purple-Dark
Theme: Breeze [GTK2/3]
Icons: Win10Sur-blue [GTK2/3]
Terminal: konsole
CPU: 11th Gen Intel i7-11700 (16) @ 4.800GHz
GPU: Intel RocketLake-S GT1 [UHD Graphics 750]
Memory: 8457MiB / 15746MiB
```

いつもの作業環境はArchか
Ubuntu24.04.
ROS2(基本Ubuntu上で動く)
使うのでDockerを常用。

BTW, Fishシェルはおすすめ。

Docker/Podman Composeの導入

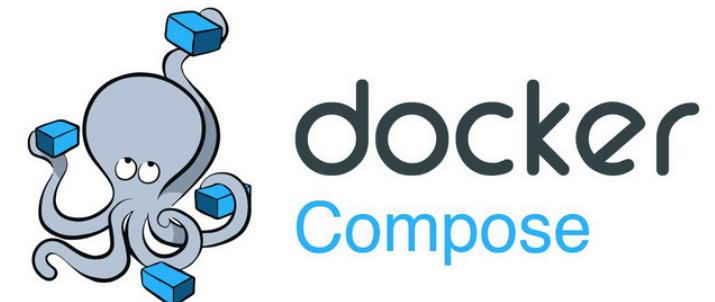
「どこでも同じ環境でプログラムを動かせる箱」

pros:

- 環境再現性、安定性の確保
- ホストOSに依存しない
- チームで環境の統一が可能
- 依存関係の記述が可能
- CI統合が容易になる

cons:

- GUI表示設定が若干分かりづらい(XII Forwarding)
- Device/Network周りの設定が地雷
- CPUアーキテクチャにImageが依存する(x86_64 vs aarch64)
- 巨大イメージ、長時間ビルド...



CI(Continuous Integration)の導入

「コードの自動健康診断」

pros:

- GitHub Actionsなど導入 자체は比較的楽(特にDocker使ってれば)
- PRやPush時にビルドを自動で回せる
- 自動化してるのでビルド忘れ等を防げる
- ReStructuredTextなどで書いたDocsをデプロイ
- コードの品質を担保

cons:

- 過信してしまう罠
- 待ち時間がストレス



CodeLangs

- C++: ロボットのコードを書くメインの言語
→ 足回りのモータ制御、腕制御、いろいろ
- C: マイコン(STM32, Esp32等)弄るとき
→ メインロジックはC++で書くので意外と出番は少ない
- Python: ロボットや通信のテスト用の簡単なコード記述、データ可視化
→ ファイル出力、json加工、websocket、何でも楽。httpサーバはコマンド一撃で立つ。
- Dart: Flutterで使う。CとJSを足して2で割った感じ。全体的にJavaっぽいかも。
→ C++のQtが嫌いな為、ロボットのUI用に使っていたが、つい最近廃止を進め中
- Rust: 安全、速い。強力なCrate(library)も多い
→ FlutterからTauriにUIを乗り換え中。(rclrsでロボットと通信をws無しで直通できる為)
- TypeScript: 現在はReactでTauriのフロント用に使っているのみ。



ROS₂の導入(Robot Operating System)

「ロボット開発の為のSDKや通信規約をまとめた統合的FW」

feat:

- DDSベースの分散型の通信システムを提供
- ロボットの各機能をNodeに分ける→Gitを使ったチーム開発でも相性O
- Moveit!(腕制御)やnav2(経路生成)など高級制御用の強力なツールと連携可能
- STM32等にもRTOSを載せて上に搭載可能
- Debug用CLIコマンドが便利
- データ可視化、物理シミュレーションにも有利
- BridgeでWebSocketに変換し、外部UI等とjson投げて通信も可能
- pkgの共有でコードを使いまわすことができる
- コンテナ上での運用も余裕



ArchLinuxの導入

「Keep It Simple, Stupid!」



pros:

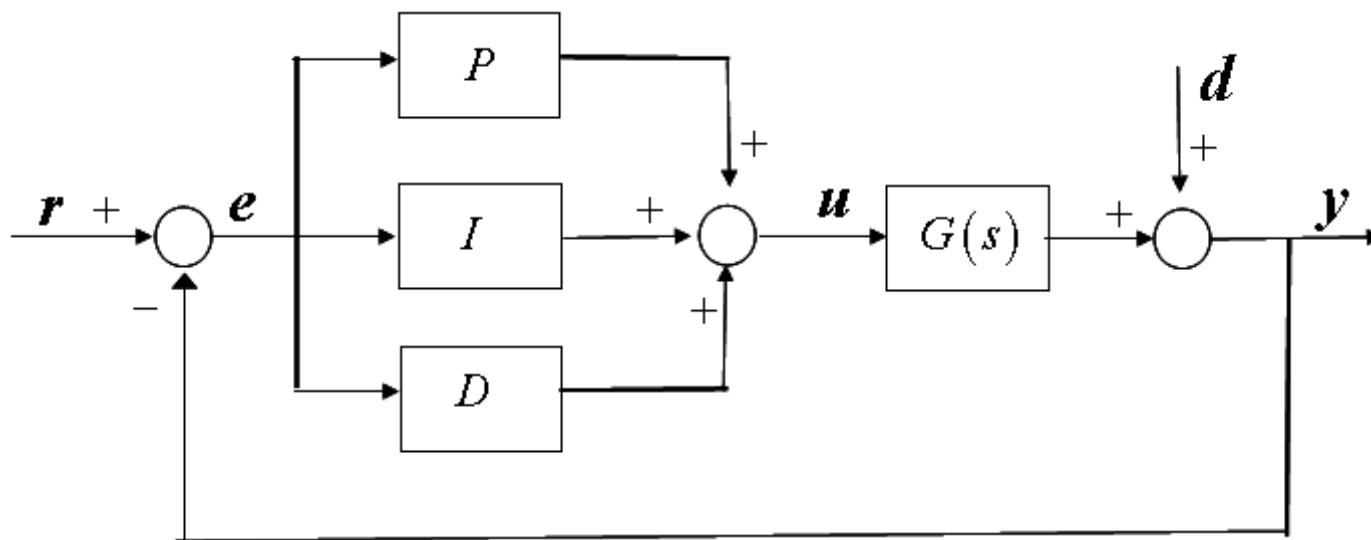
- AUR(Arch User Repository)をyay/paruで探せば大体なんもある→開発でも普段でも便利
- ArchWikiが最強
- とりあえず軽い
- 最新pkgがすぐに手に入る
- Native LinuxなのでWSL2固有の問題(Network/Device)など気にする必要がない

cons:

- インストールが若干ハードル高いかも
- たまーに壊れる
- AUR乱用すると依存地獄
- Debian系ではないのでサポートされていないソフトが存在

II. 少し制御の話

PID Controller (Proportional-integral-derivative Controller)



$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t)$$

ここで、
e(偏差) = r(目標値) - y(観測値)
e(目標までの差) → u(制御値)

ここでは非常によく用いられる制御手法であるPID制御のみ紹介。

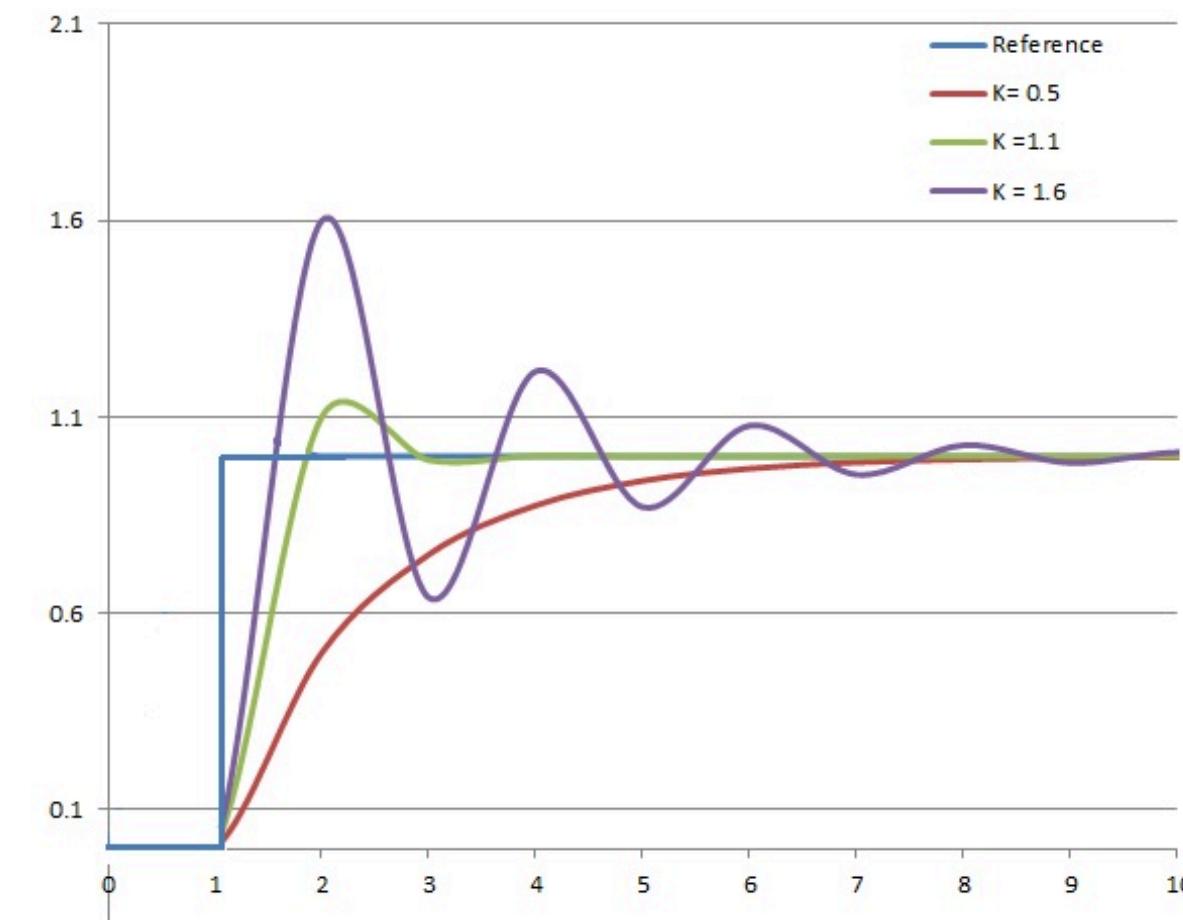
シンプルで強力。複雑な数理モデル化が基本必要ない。

eを小さくする、つまり目標値に近づけるようにゲインの値を調整し制御する

K_p : 今の誤差に反応

K_i : 過去の誤差を補正

K_d : 誤差の変化を予測



// Rust

離散PID簡易実装

e、eの積分、微分値を計算

PID制御式を実装

3つのゲインは調整する必要がある

微分はオイラー前進法、積分は矩形
積分で簡単に実装

基本的なロジックはこれだけ。

```
struct PID {
```

```
    kp: f64,
```

```
    ki: f64,
```

```
    kd: f64,
```

```
    prev_error: f64,
```

```
    integral: f64,
```

```
}
```

```
impl PID {
```

```
    fn update(&mut self, setpoint: f64, measured: f64, dt: f64) -> f64 {
```

```
        // 目標値から観測値を引く
```

```
        let error = setpoint - measured;
```

```
        self.integral += error * dt;
```

```
        let derivative = (error - self.prev_error) / dt;
```

```
        self.prev_error = error;
```

```
        // さっきの式
```

```
        self.kp * error + self.ki * self.integral + self.kd * derivative
```

```
}
```

```
}
```

モデリング

URDF

- ロボットの構造をXMLのフォーマットで記述
- ロボットの状態可視化
- SRDFと組み合わせてアーム制御に使ったりもする

数理モデル

- LQR, MPC等現代制御ベースの設計をするための基盤
- ロボットの動きを微分方程式で記述
- FK/IK(運動学)を用いて、関節角度<--->EndEffector位置の変換(行列演算)
- 動力学を用いてトルクと加速度の関係を記述

他にも...

I₂C, SPI, CANなどの通信まわり、
ネットワーク、プログラミング、組み込み、
Linux, 回路、機械要素、AI, 画像処理... etc.

場合によってはUIを作るためのWebやアプリ作成もやる

単に“制御”や“ロボットプログラミング”と言っても範囲は膨大...

一人でやるのは大変だが、
其々で個人の強みを生かせる場面は多いかもしれない。

Fin.