FPGAを用いた小型自動走行車の開発におけるこれまでの取り組みと今後の課題

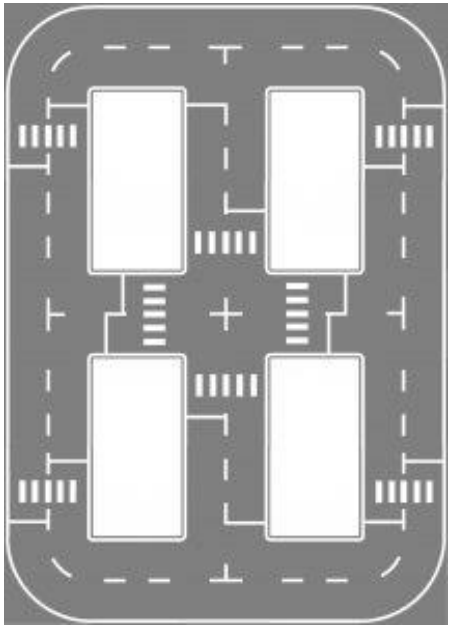
立命館大学大学院 理工学研究科

工藤 裕也, 高田 厚志

目次

1. コンテストに関して

- 概要
  - テーマは小型ロボットの自動走行
  - 開発条件
    - FPGAを使うこと
    - 100Wh未満のバッテリーで動作すること
    - 外部と通信しないこと
    - CCD/CMOSカメラ以外のセンサを使用すると減点
  - ルール
    - 車線遵守
    - 指示通りの道順で道路を走行する



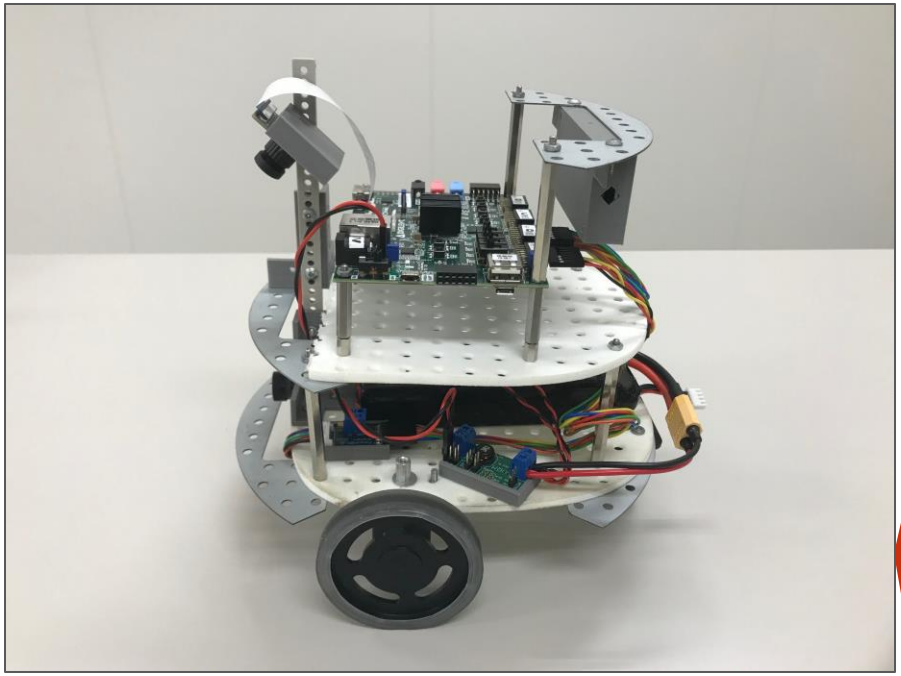
自動走行を行うコース

2. これまでの取り組み

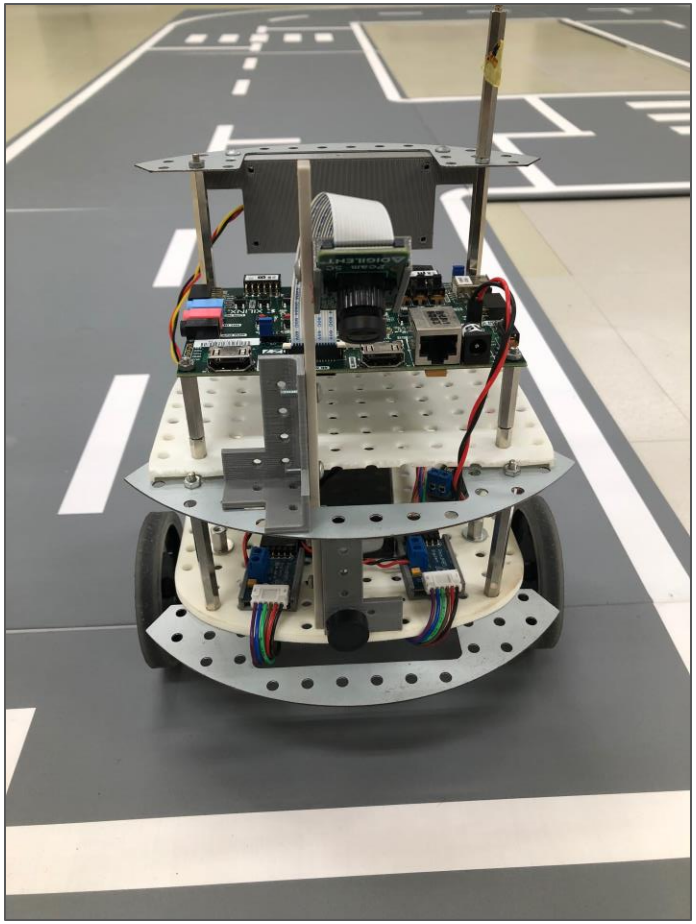
2.1 第8回 相磯秀夫杯 (2018-09-08) Ver.

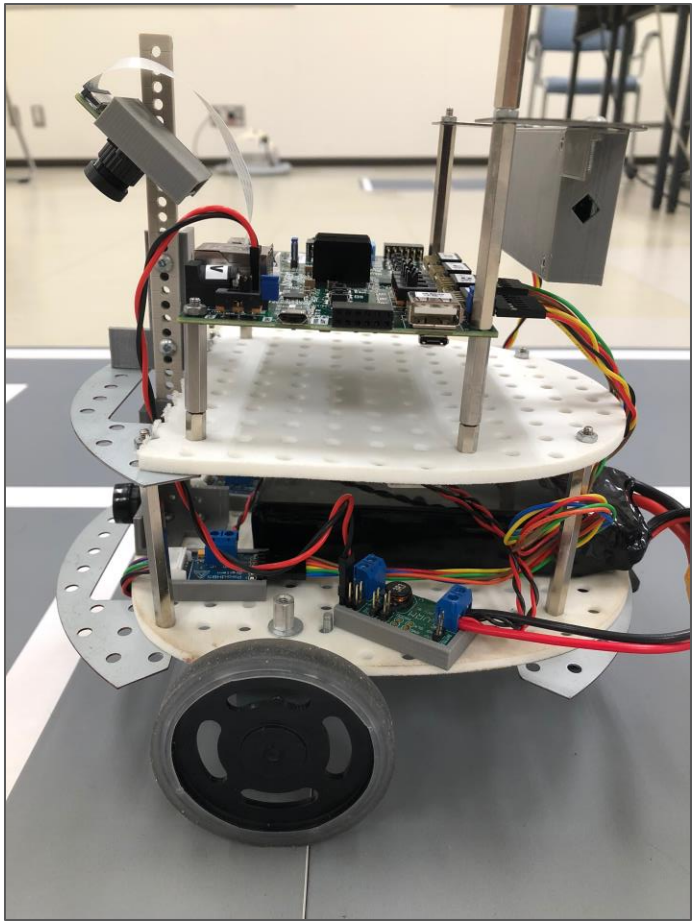
2.1.1 車体構成

- ZYBO Z7-20 (Xilinx Zynq XC7Z020)
- Pcam-5C (OV5640)
- VRM Rev. B
- RB-Sta-15 (Li-Poバッテリー)
- Pmod HB5 (Hブリッジ回路)
- IG220053X0085R (DCモーター)



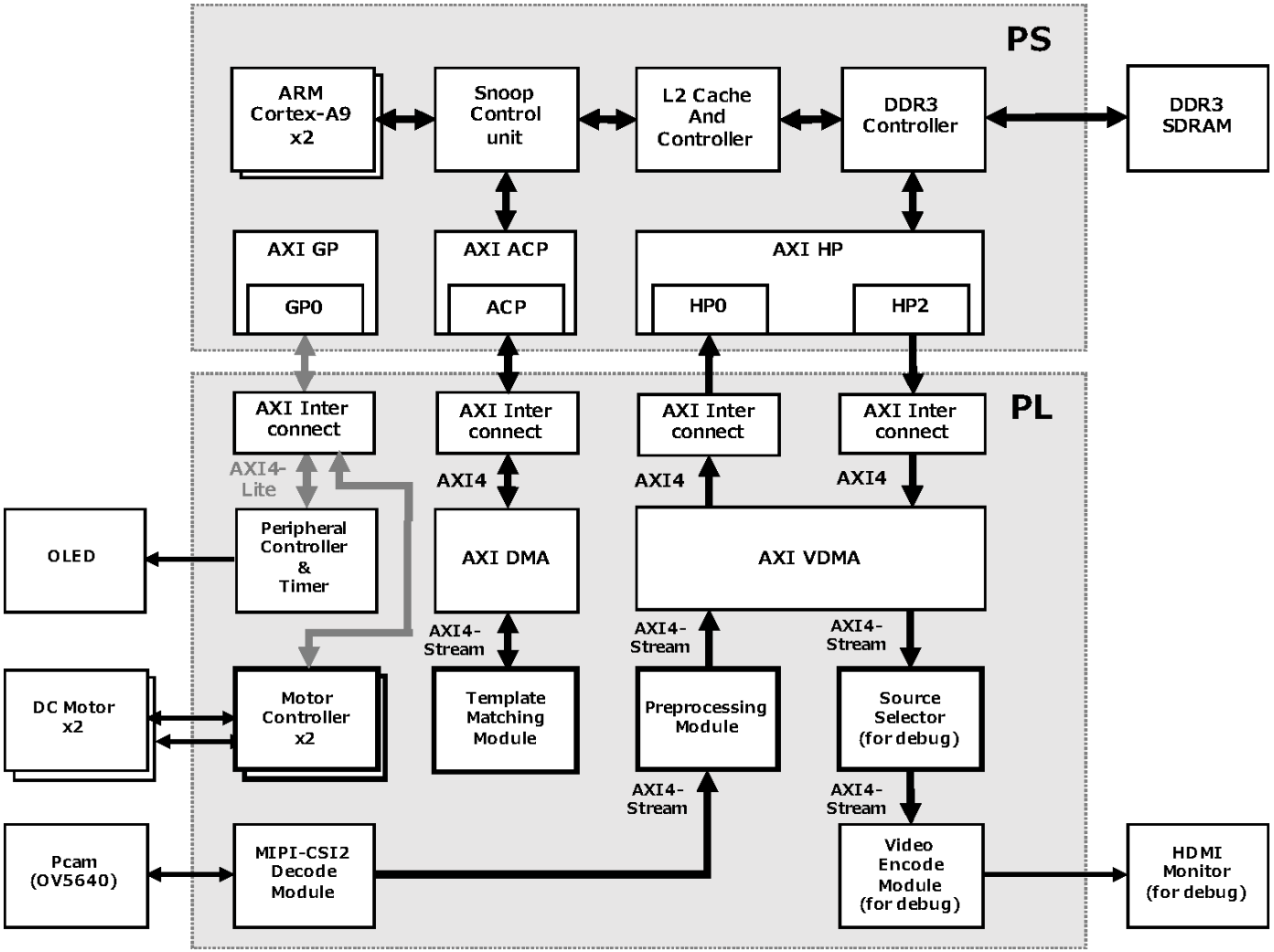
2.1.1 車体構成





2.1 第8回 相磯秀夫杯 (2018-09-08) Ver.

2.1.2 システムアーキテクチャ



2.1 第8回 相磯秀夫杯 (2018-09-08) Ver.

2.1.3 自動走行システム

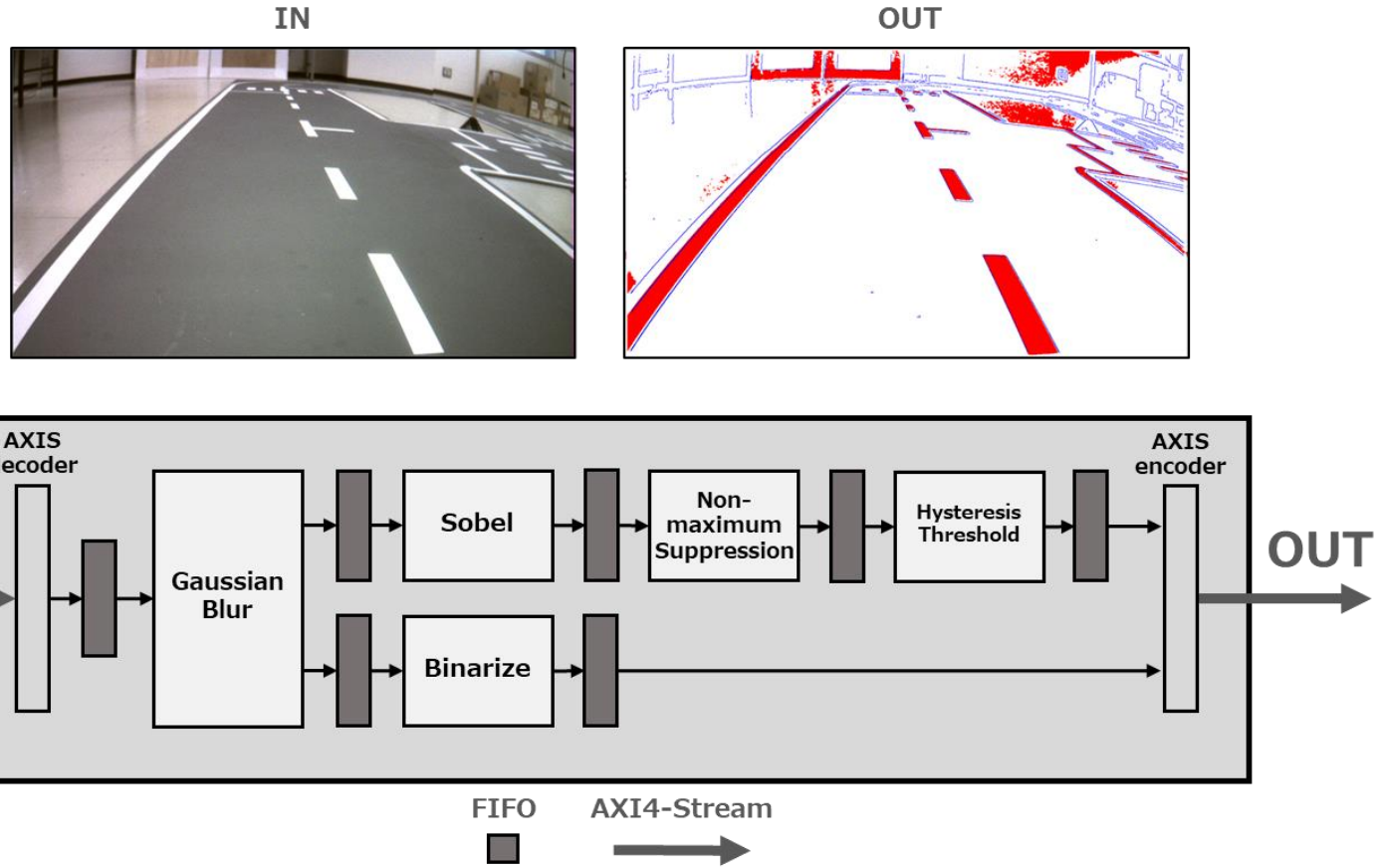
- SWはベアメタル実装
  - OSなし
  - C/C++で記述
- 画像処理の一部・モータ制御にFPGAを使用
  - Cannyエッジ検出
  - 二値化
  - テンプレートマッチング
  - モータのPID制御
- 路面認識のみ実装
  - 障害物や歩行者、信号は検出しない
  - 単純なアルゴリズムで完走することを目標とした

2.1.3 自動走行システム

- 前処理 (HW/SW)
  - Cannyエッジ検出 (HW)
  - 二値化 (HW)
  - 歪曲収差補正 (SW)
  - 射影変換 (SW)
- 路面標識認識 (HW)
  - テンプレートマッチング (HW)
- 車道外側線推定 (SW)
  - 直線ハフ変換 (SW)
- ライントレース (SW)

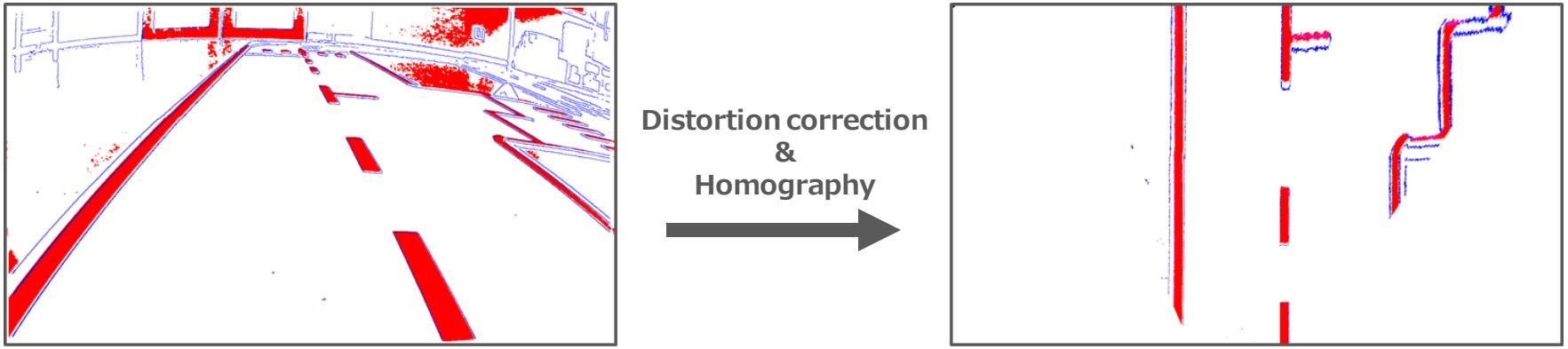
2.1.3 自動走行システム

- 前処理 (HW)
  - Cannyエッジ検出と二値化を並列実行



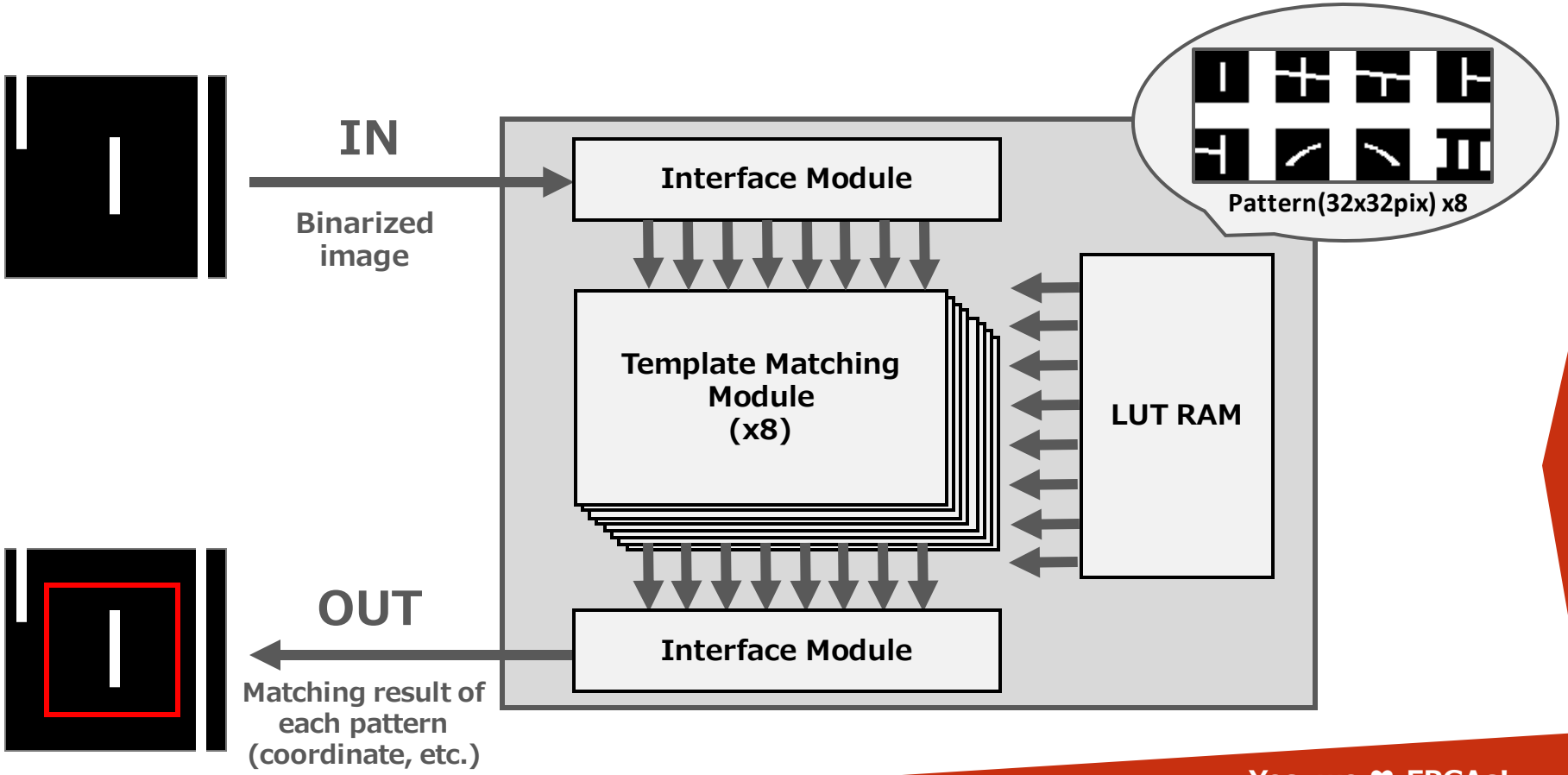
2.1.3 自動走行システム

- 前処理 (SW)
  - 歪曲収差補正・射影変換をスクラッチ実装
  - OpenCVの実装を参考



2.1.3 自動走行システム

- 路面標識認識(HW)
  - SADによるテンプレートマッチングを並列実行
  - **8**(pattern)x**32**(pix)x**32**(pix)回のXOR演算を**1**cycleで処理



2.1.3 自動走行システム

- 車道外側線推定 (SW)
  - 直線ハフ変換によって得られる$\rho\theta$空間から、左右の外側線として尤もらしい点の組を求める

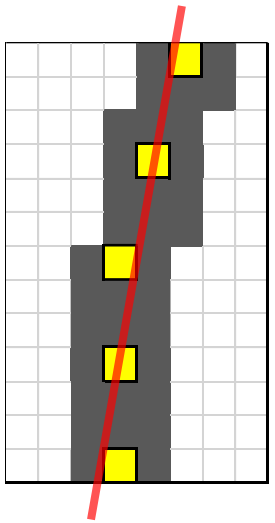
1. エッジ画像に対して直線ハフ変換を実行
2. ノイズを除去（$\rho\theta$空間でローパスフィルタを適用）
3. 以下の２つの条件を満たす点(p_1, p_2)の組の内、もっとも値の和が大きいものを全探索で求める
   1. $|p_{1\rho} - p_{2\rho}| = Road\ width$
   2. $p_{1\theta} = p_{2\theta}$

＊ $\rho\theta$空間のθ方向は誤差が許容できる程度で量子化

2.1.3 自動走行システム

- ライントレース (SW)
  - 白線検出を行い、車体が白線から一定距離、かつ、平行となるように車体を制御する

1. 二値画像に対して4連結ラベリング処理を適用
2. 得られた連結成分の内、外側線として尤もらしいものを推定する
3. 推定した連結成分に対し、下図のようなサンプリングを行う
4. 最小二乗法を用いて直線を求める
5. 直線の角度と直線から車体までの距離を用いたフィードバック制御により車体の操舵角を決定する



2.1 第8回 相磯秀夫杯 (2018-09-08) Ver.

2.1.4 性能評価

表. リソース使用数

Module name	LUT	FF	BRAM	DSP
Preprocessing Module	2042(3.84%)	3012(2.83%)	12(8.57%)	11(5.00%)
Template Matching Module	16231(31.0%)	20867(19.6%)	4(2.86%)	0(0.00%)
Motor Controller x2	3756(7.06%)	3884(3.65%)	0(0.00%)	56(25.5%)
Other	10938(20.6%)	14830(13.9%)	15.5(11.1%)	0(0.00%)
Total	32967(62.0%)	42593(40.0%)	31.5(22.5%)	67(30.5%)

表. 平均処理時間

Name	SW	HW	SW/HW
Preprocessing (Canny edge detection, Binarization)	-	6.367ms	-
Preprocessing (Coordinate transformation)	25.73ms	-	-
Template matching (Eight times)	246.4ms	0.1244ms	1981
Outside line estimation	241.2ms	-	-
Line trace	0.6378ms	-	-

2.1 第8回 相磯秀夫杯 (2018-09-08) Ver.

2.1.5 所感・考察

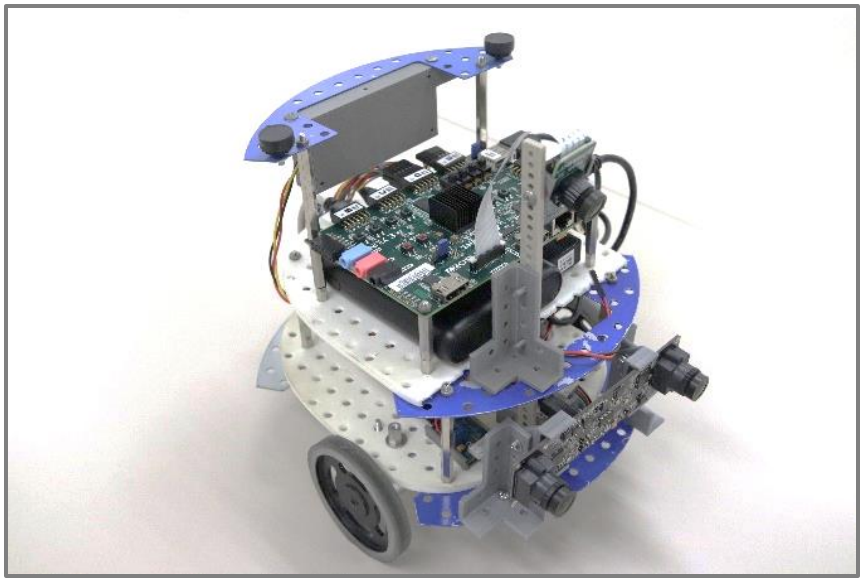
- コースを完走することが可能な必要最低限の実装を行った
  - 少ない労力（？）で結果を残せた
  - 悪く言うと拡張性・汎用性に乏しい

- ベアメタル実装を行ったため、OpenCV等の便利なライブラリを扱うことが出来ず、一部の実装に手間がかかった
  - FPGA実装箇所のデバッグは快適だった

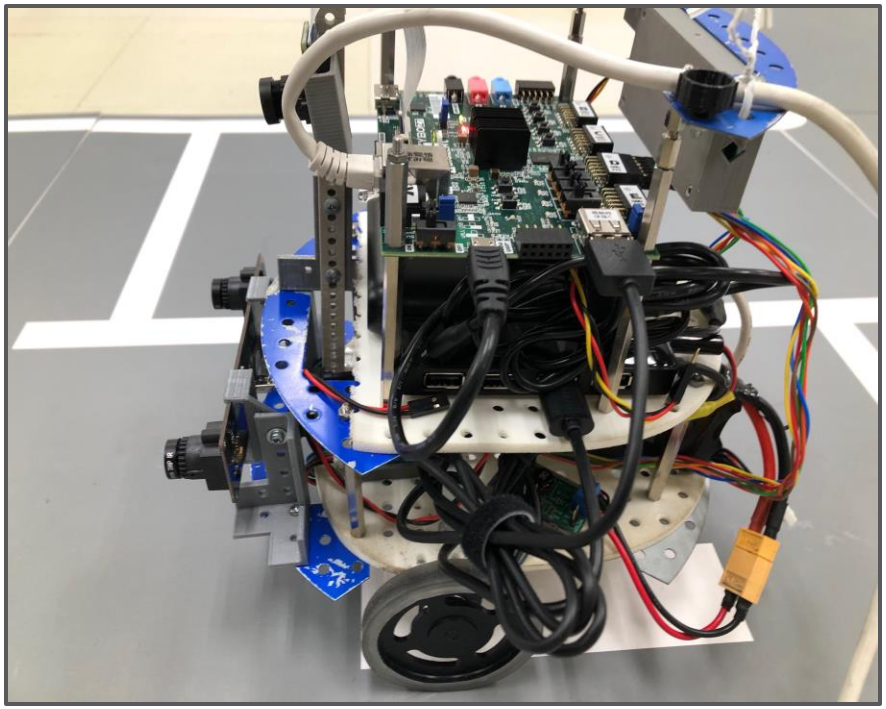
2.2. FPT2018 (2018-12-10) Ver.

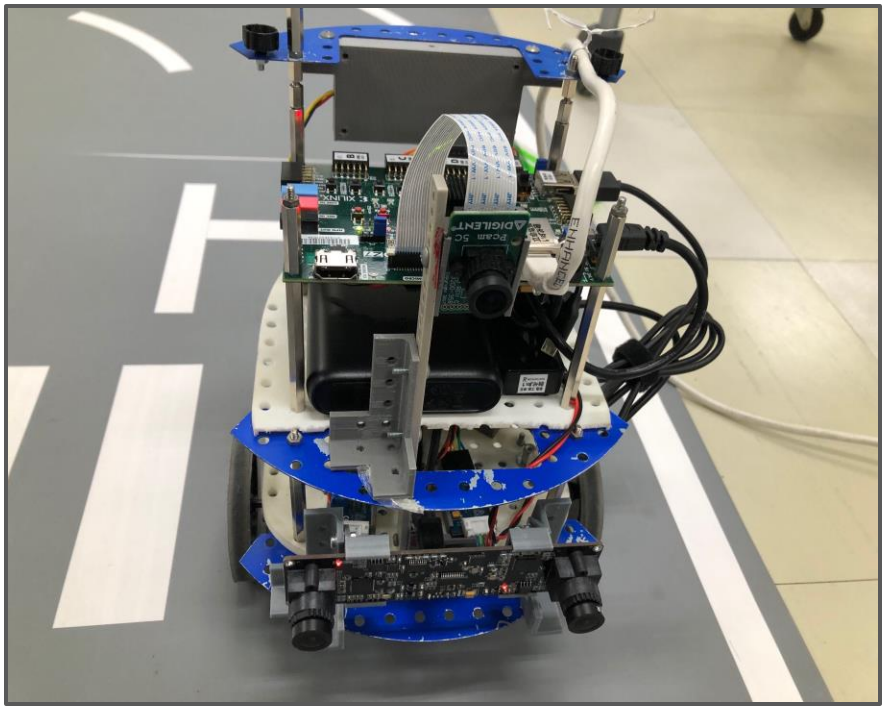
2.2.1 車体構成

- ZYBO Z7-20 (Xilinx Zynq XC7Z020)
- Pcam-5C (OV5640)
- Optor Visual-Inertial Camera
- VRM Rev. B
- PowerCore 13000 USB-C
- RB-Sta-15 (Li-Poバッテリー)
- Pmod HB5 (Hブリッジ回路)
- IG220053X0085R (DCモーター)



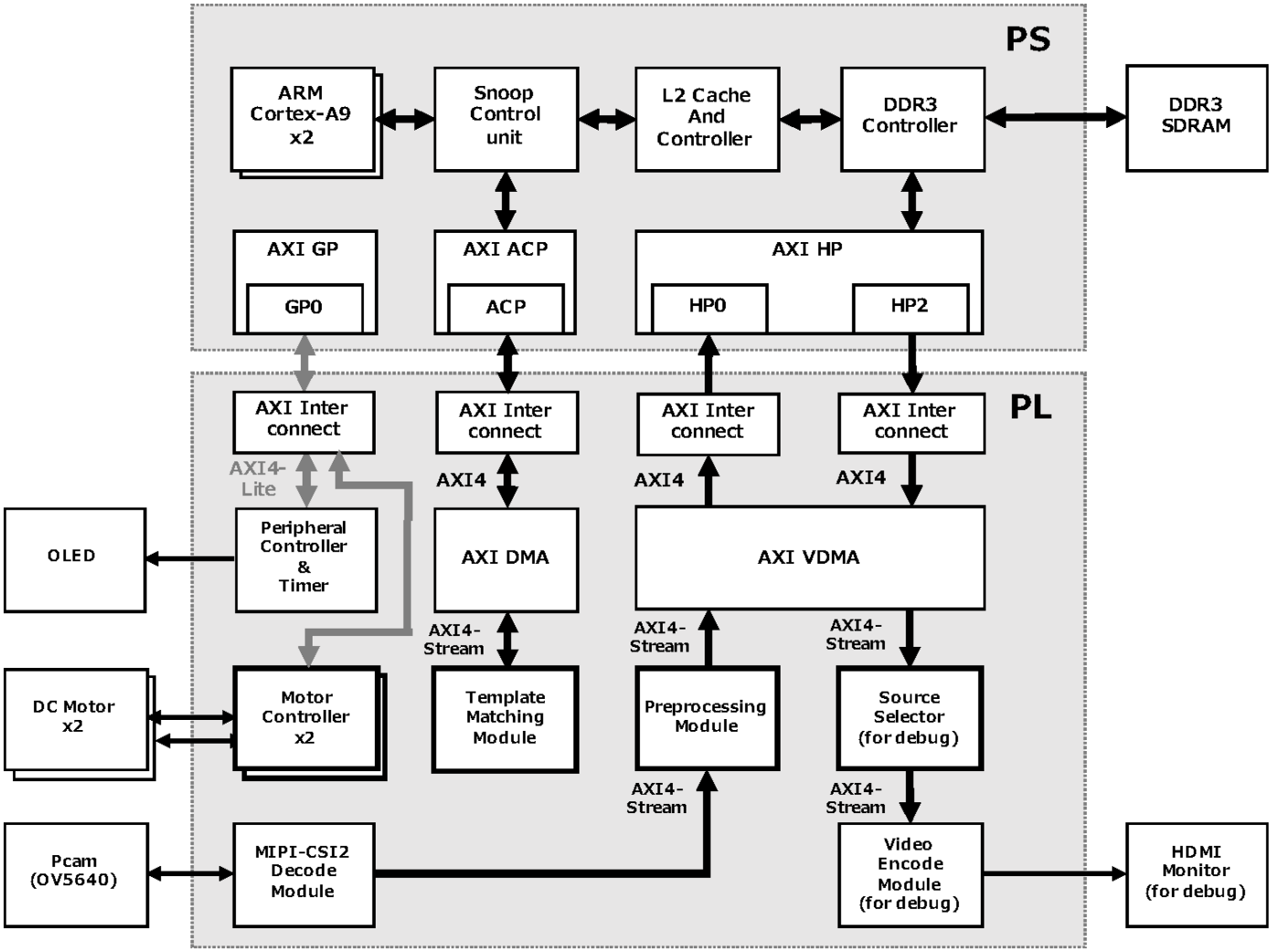
2.2.1 車体構成





2.2. FPT2018 (2018-12-10) Ver.

2.2.2 システムアーキテクチャ



2.2. FPT2018 (2018-12-10) Ver.

2.2.3 自動走行システム

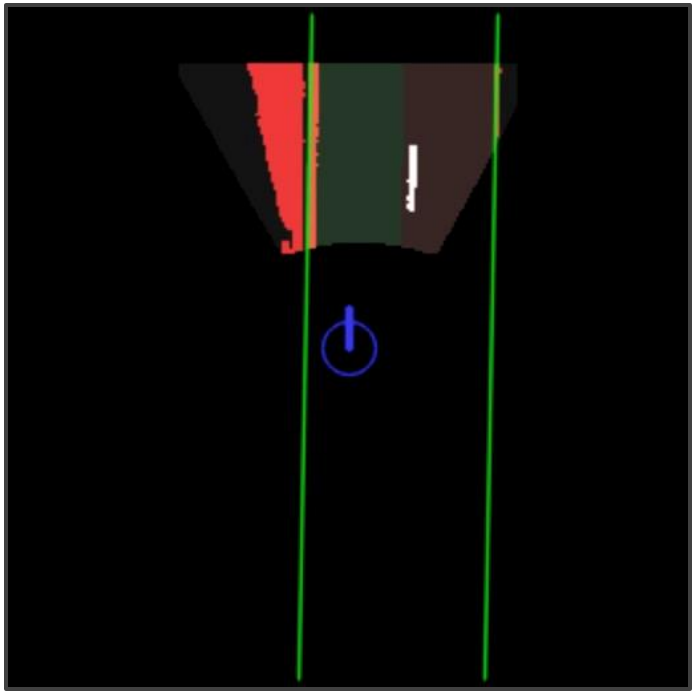
- CPU上でLinuxを動作させる
  - Ubuntu16.04
- 画像処理の一部・モータ制御にFPGAを使用
  - Cannyエッジ検出
  - 二値化
  - テンプレートマッチング
  - モータのPID制御
- より柔軟な実装を目指して
  - ライントレースに頼りすぎない（or 頼らない）
  - 経路計画アルゴリズムを導入

2.2.3 自動走行システム

- 前処理 (HW/SW)
  - Cannyエッジ検出 (HW)
  - 二値化 (HW)
  - 歪曲収差補正 (SW)
  - 射影変換 (SW)
- 路面標識認識 (HW)
  - テンプレートマッチング (HW)
- 車道外側線推定 (SW)
  - 直線ハフ変換 (SW)
- 地図生成(SW) <- new!
- 自己位置推定(HW/SW) <- new!
  - Wheel odometry
- 経路計画(SW) <- new!
  - RRT*
- 障害物認識(SW) <- new!
  - ステレオマッチング

2.2.3 自動走行システム

- 地図生成(SW)
  - 経路計画に必要となる地図を生成する
  - 以下の３つの情報を用いて生成
    - 路面の俯瞰画像
    - 外側線推定の結果
    - テンプレートマッチングの結果
  - 進入可能領域と進入禁止領域を決定
    - 経路計画時の制約として利用する
  - 局所的な地図だけでよい
    - 経路計画の適用範囲も局所的であるため



2.2.3 自動走行システム

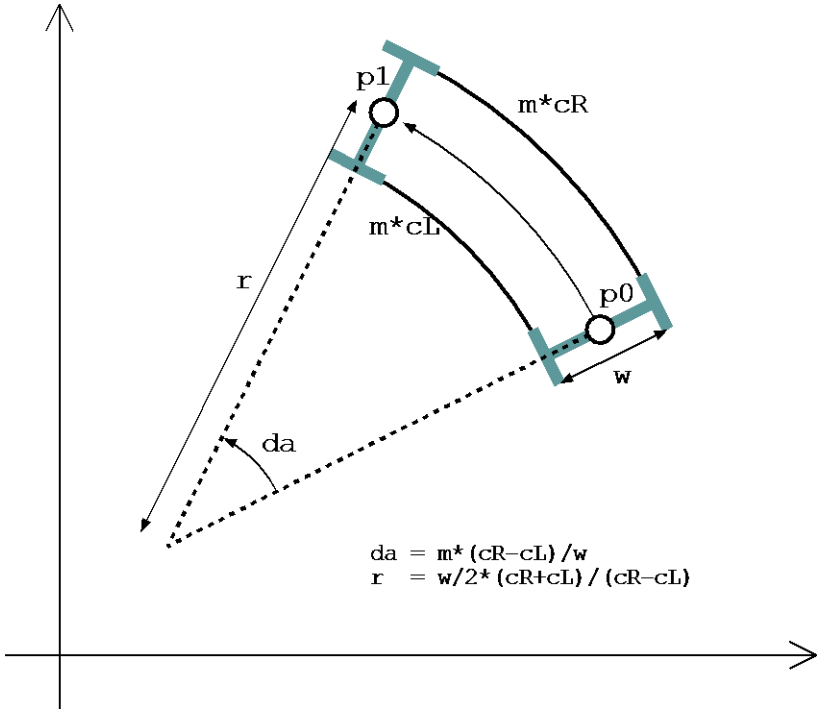
- 自己位置推定(HW/SW)
  - 左右の車輪のモータに付随しているロータリエンコーダのカウンタ値を読み取り、その値の変化から車体の移動量（Odometry・Dead Reckoning）を算出する

$$\Delta a = \frac{m}{w}(c_R - c_L)$$

$$r = \frac{w}{2} \cdot \frac{c_R + c_L}{c_R - c_L}$$

$$m \cdot C_L = \left(r - \frac{w}{2}\right) \cdot \Delta a$$

$$m \cdot C_R = \left(r + \frac{w}{2}\right) \cdot \Delta a$$



2.2.3 自動走行システム

- 経路計画(SW)
  - ある状態空間において、状態Aから状態Bまで経路を求めたい
    - ここで、状態空間はユークリッド空間とする
  - 最適に近い経路をより高速に求めたい
    - 最適な経路とは、制約を満たし、経路のユークリッド距離が理論的に最短となるもの
  - この問題に適用可能なアルゴリズムは大きく2つに分類される
    - Graph-search method
    - Sampling-based method
  - 今回は**RRT***(Optimal Rapidly-exploring Random Tree)を採用
    - Sampling-based methodに分類される
    - **RRT**(Rapidly-exploring Random Tree)の拡張
    - 与えられた制約に対する解が存在するなら、必ず解を導出することが出来る
    - 確率的な完全性を持つ(probabilistically completeness)
    - 最適解の近似解を導出することが出来る(asymptotically optimal)

2.2.3 自動走行システム

- RRT(Rapidly-exploring Random Tree)
  - n次元の状態空間を$X \in \mathbb{R}^n$、暗黙的に生成されるグラフを$G(V,E)$とする
  - $V \in X, e = (u \in V, v \in V) \in E, x_{start} \in X, x_{goal} \in X$
  - $x_{goal} \in V$を満たすまで以下の操作を繰り返す

1. 一様分布に従い、$x_{rand} \in X$を生成する
   このとき、ある確率でx_{rand}を目標状態x_{goal}とする
2. Vの要素の内、x_{rand}とのユークリッド距離が最も近いものをx_{near}とする
3. x_{new}を以下のように導出する

$$x_{new} = \Delta t \left(\frac{x_{rand} - x_{near}}{|x_{rand} - x_{near}|} \right)$$

4. x_{near}とx_{new}の間に存在する全ての状態が制約を満たしていれば以下の操作を行う

$$V \xleftarrow{+} \{x_{new}\}, \; E \xleftarrow{+} \{x_{near}, x_{new}\}$$

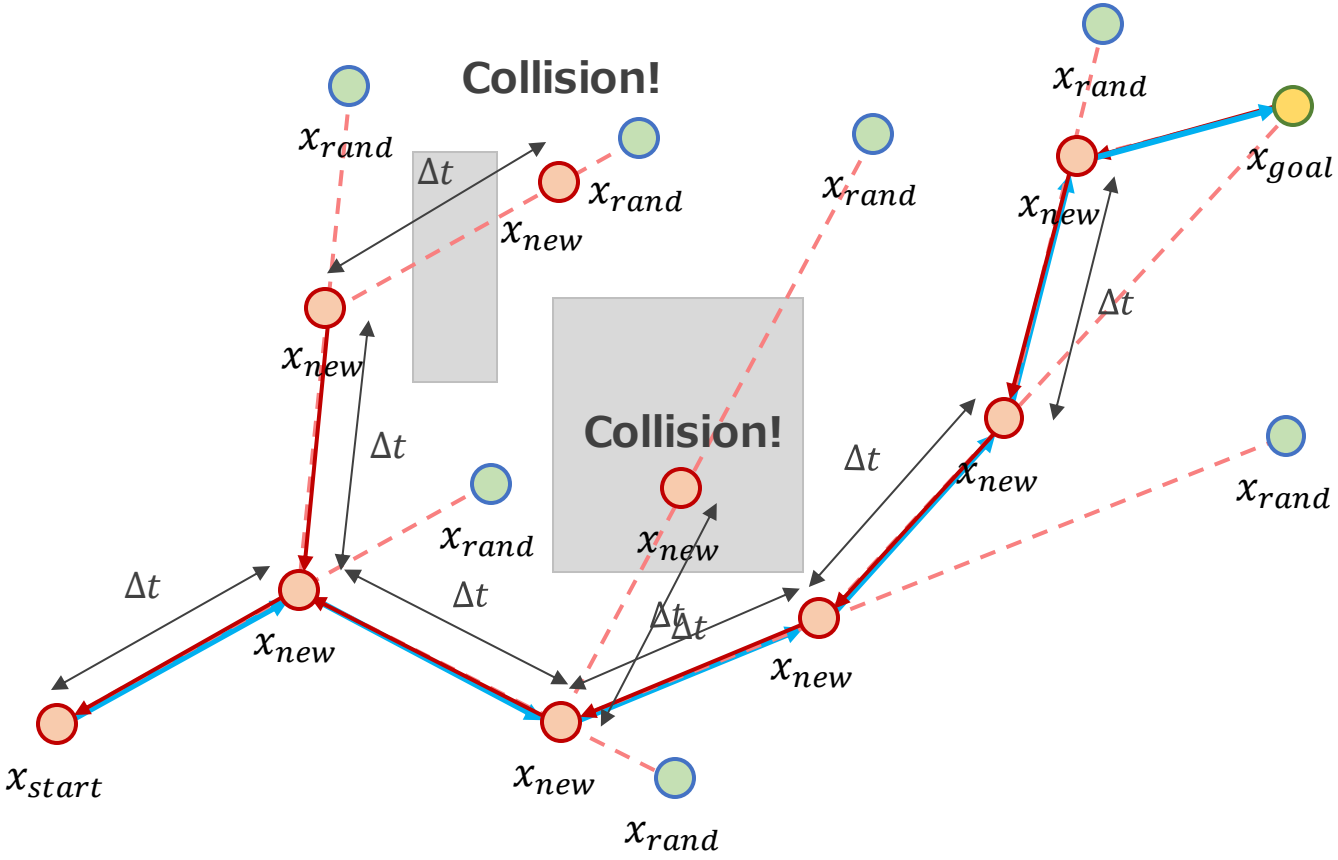
```
GENERATE_RRT(x_init, K, Δt)
1  T.init(x_init);
2  for k = 1 to K do
3      x_rand ← RANDOM_STATE();
4      x_near ← NEAREST_NEIGHBOR(x_rand, T);
5      u ← SELECT_INPUT(x_rand, x_near);
6      x_new ← NEW_STATE(x_near, u, Δt);
7      T.add_vertex(x_new);
8      T.add_edge(x_near, x_new, u);
9  Return T
```

擬似コード[*]

[*] LaValle, Steven M. (October 1998). "Rapidly-exploring random trees: A new tool for path planning," Technical Report. Computer Science Department, Iowa State University (TR 98–11).

2.2.3 自動走行システム

- RRT(Rapidly-exploring Random Tree)

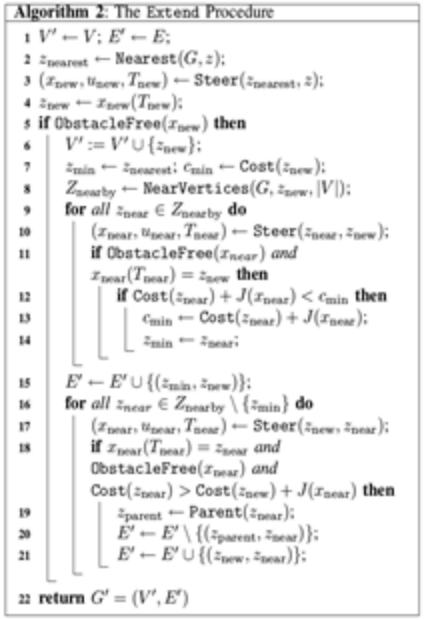


2.2.3 自動走行システム

- RRT*(Optimal Rapidly-exploring Random Tree)
  - 各頂点に開始状態からの頂点経由の距離を持たせ、これを用いて辺の情報を随時更新することで、最適解の近似解を得ることが出来る
  - n回目の操作において、辺の情報を更新する頂点集合V_{update}を以下のように定義する

$$V_{update} := \left\{ v \in V \middle| \left(\frac{\gamma}{\zeta_d} \frac{\log n}{n} \right)^{\frac{1}{d}} < |v - x_{new}| \right\}$$

  - V_{update}に含まれる頂点の内、開始状態からの頂点経由の距離とx_{new}までのユークリッド距離の和が最短のものをx_{new}の親ノードとする
  - V_{update}に含まれる頂点それぞれについて、x_{new}を親ノードとした方が開始状態からの頂点経由の距離が短くなる場合、x_{new}を親ノードとして繋ぎ変える（Rewire処理）

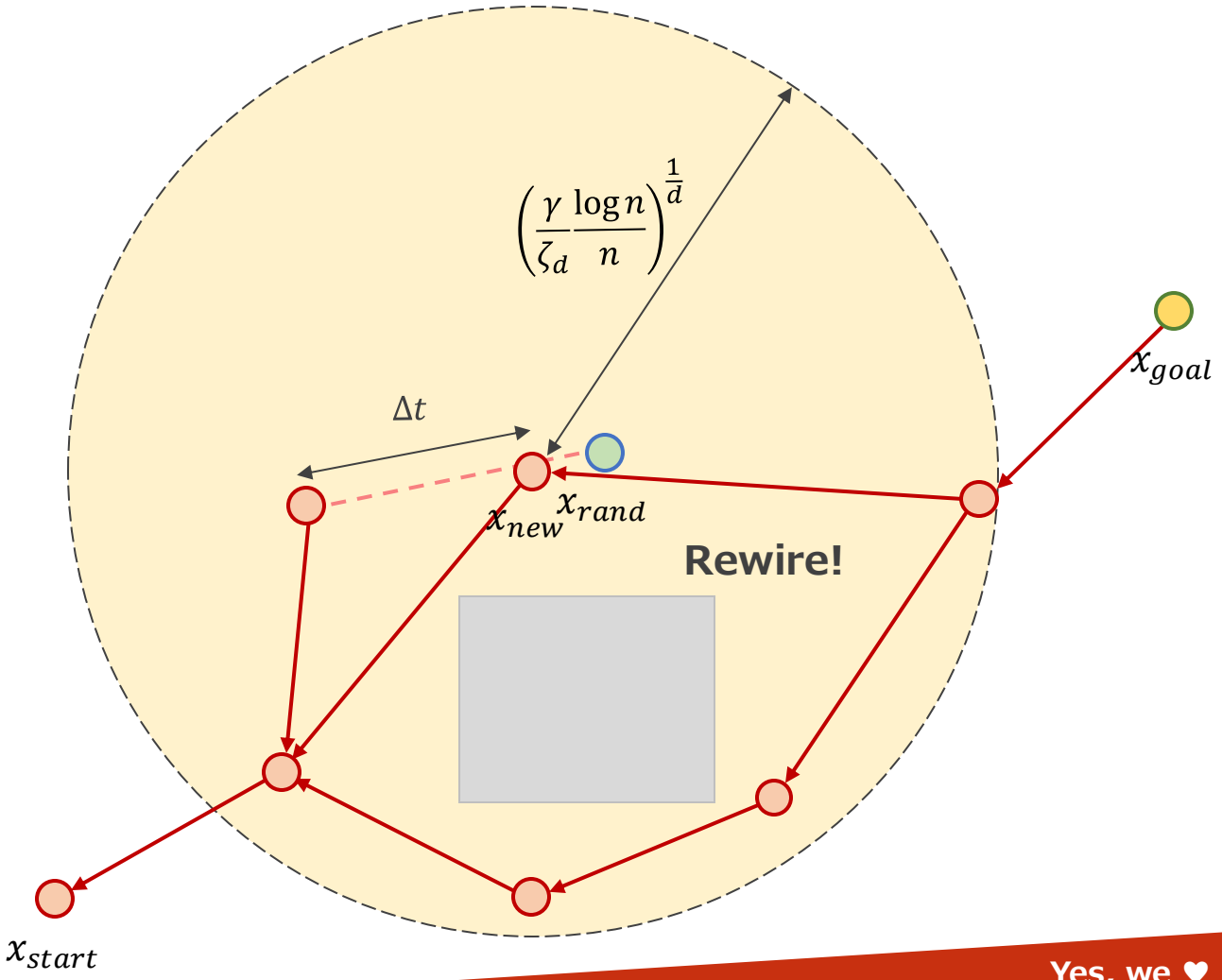


擬似コード[*]

[*] S. Karaman and E. Frazzoli,“ Optimal kinodynamic motion planning using incremental sampling-based methods, ”49th IEEE Conference on Decision and Control (CDC), 2010, pp. 7681-7687.

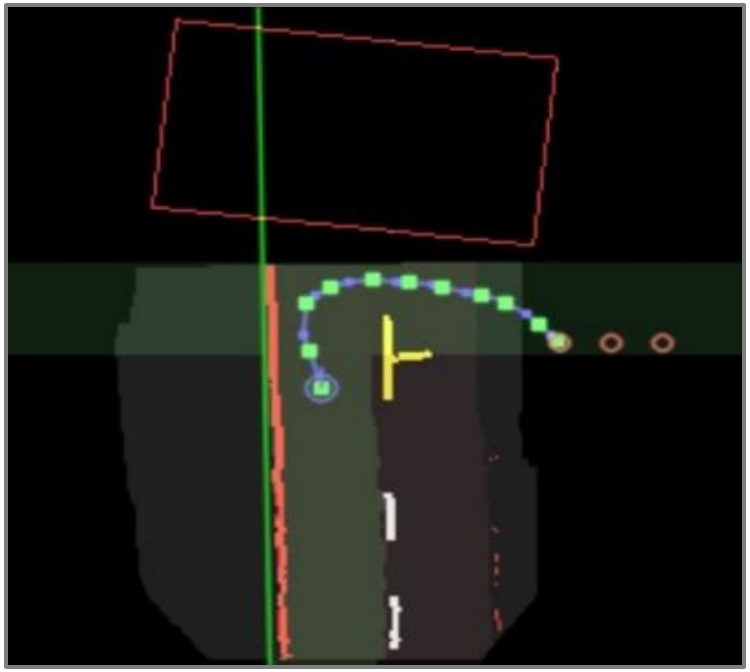
2.2.3 自動走行システム

- RRT*(Optimal Rapidly-exploring Random Tree)



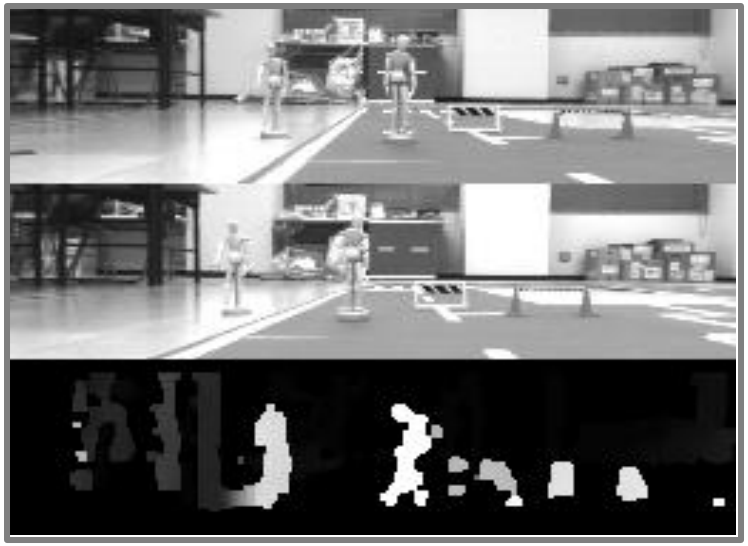
2.2.3 自動走行システム

- 経路計画(SW)
  - 地図、つまり、２次元平面を状態空間とみなし、RRT*を用いて経路計画を行う
  - 車体の現在座標を開始状態とし、テンプレートマッチングの結果から目標状態を決定



2.2.3 自動走行システム

- 障害物認識(SW)
  - ステレオカメラを用いて障害物を検出する
  - BM法により視差画像を生成し、
    得られた画像をラベリング、
    連結成分の形状・大きさから
    障害物の種類を判定する
  - 搭載したステレオカメラ（Optor）で、
    車体前方0.9～1.4mの範囲の
    物体の検出が可能



上段：右画像
中段：左画像
下段：視差画像

2.2. FPT2018 (2018-12-10) Ver.

2.2.4 性能評価

- ここでのデータは残っていない

2.2. FPT2018 (2018-12-10) Ver.

2.2.5 所感・考察

- システムの破綻、電源周りのトラブルが発生したため
  コンテストを棄権してしまった

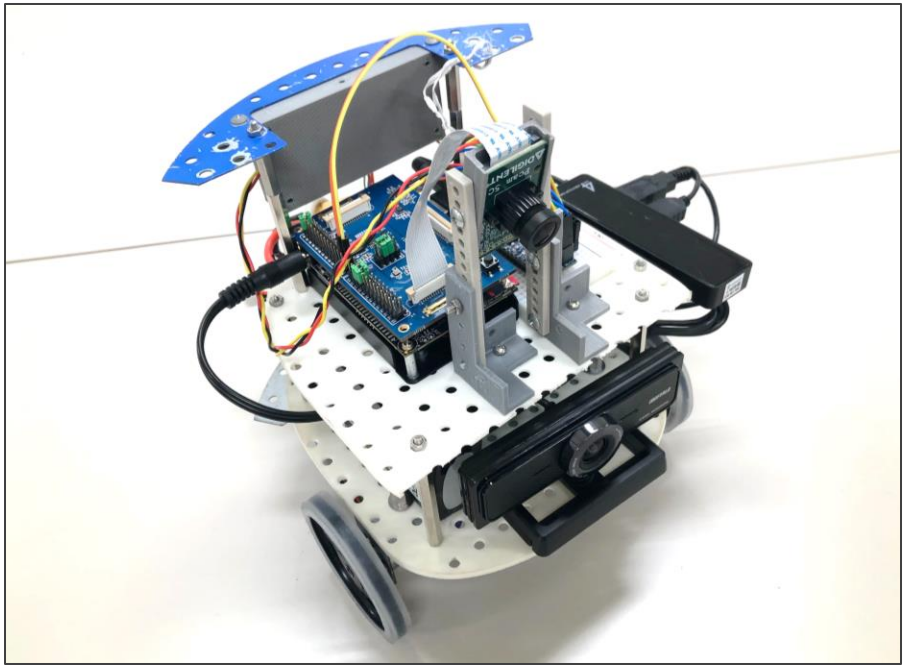
- システムの破綻について、経路計画を行うための制約を前情報なしで生成するのが困難だった
  - これを無理に行おうとすると、結局コンテストに特化した実装になってしまい、汎用性・拡張性に乏しくなってしまう

- 実装するアルゴリズムの種類が大幅に増え、CPUリソースの不足が起こった
  - Zynq-7000のARM Cortex-A9では厳しい
  - より高性能なCPUを持つFPGA SoCを採用したくなった

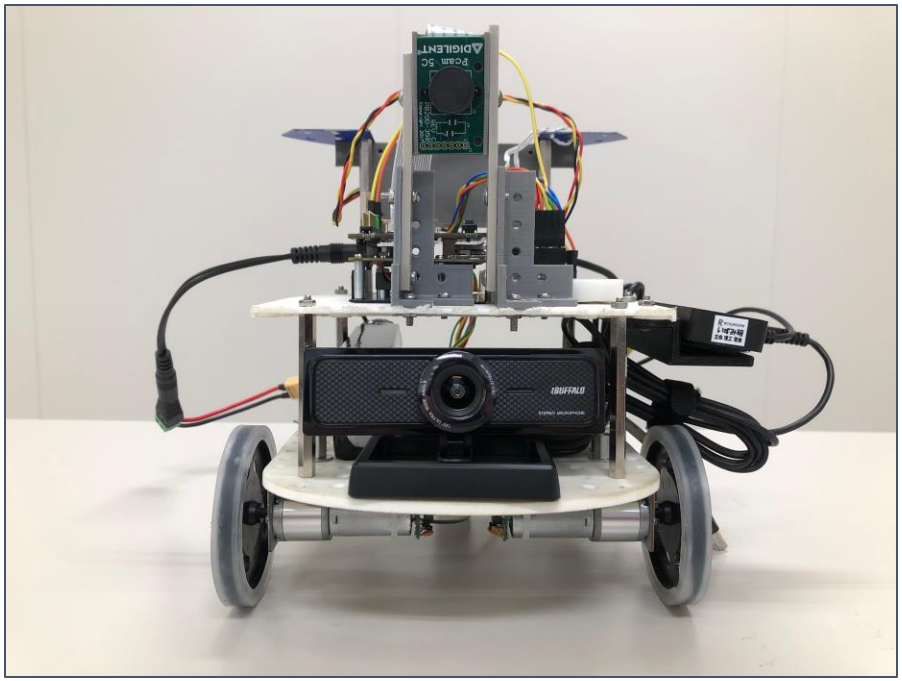
2.3. HEART2019 (2019-06-06) Ver.

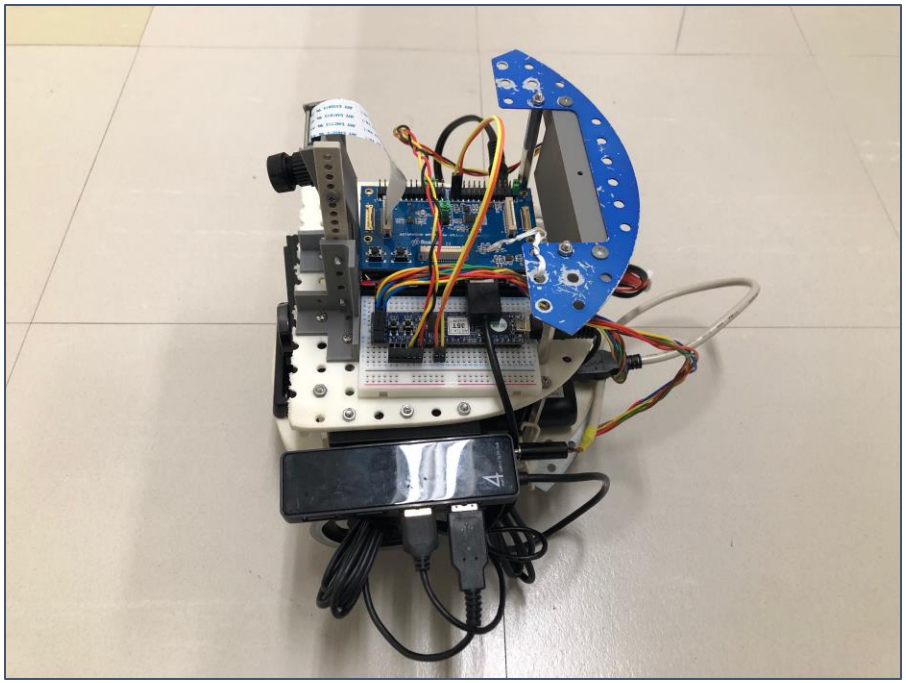
2.3.1 車体構成

- Ultra96 (Xilinx Zynq UltraScale + MPSoC ZU3EG)
- Cmod A7-35T (Xilinx Artix-7 XC7A35T)
- Pcam-5C (OV5640)
- USB Camera(BSW20KM11BK)
- VRM Rev. B
- PowerCore 13000 USB-C
- RB-Sta-15 (Li-Poバッテリー)
- Pmod HB5 (Hブリッジ回路)
- IG220053X0085R (DCモーター)



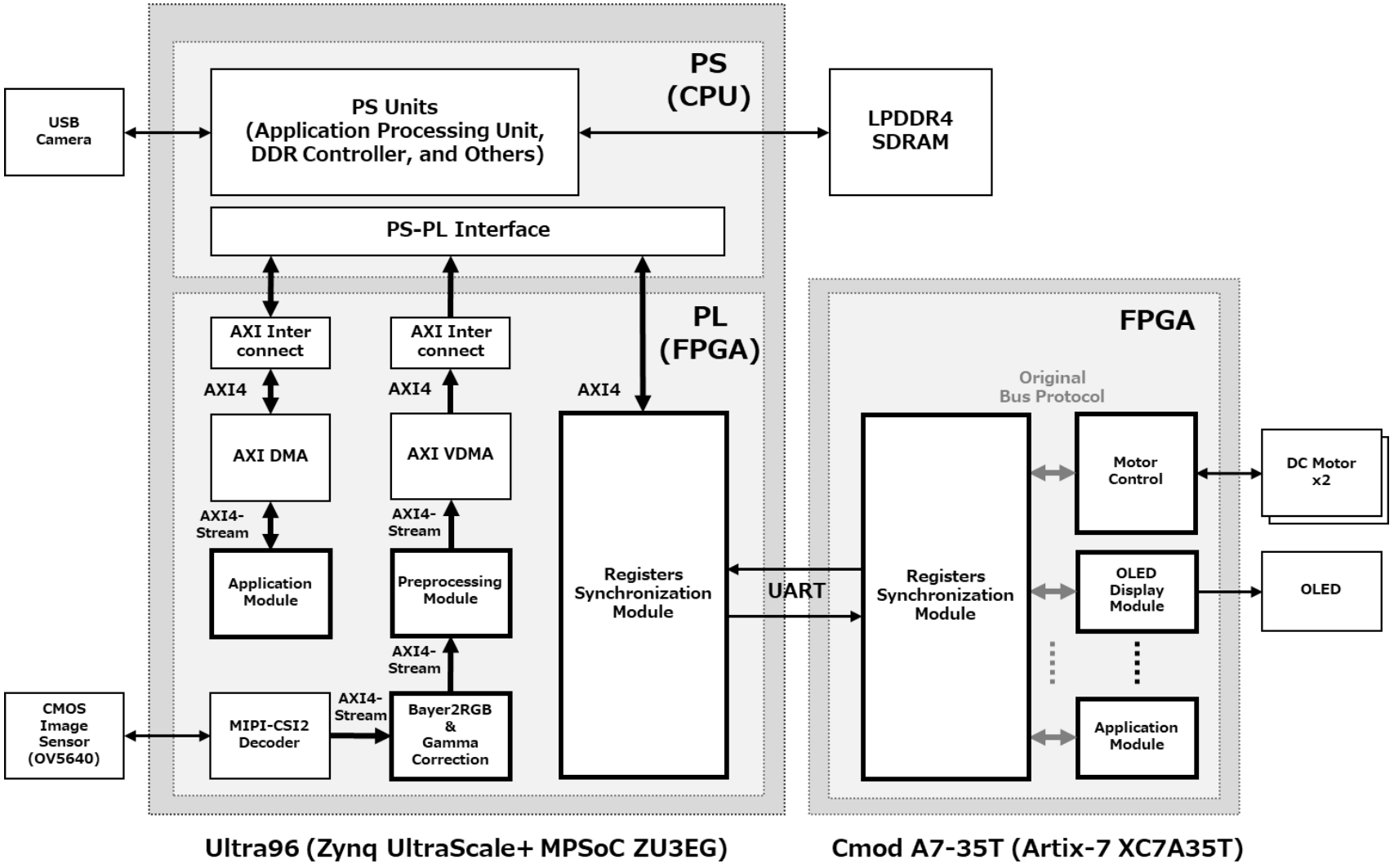
2.3.1 車体構成



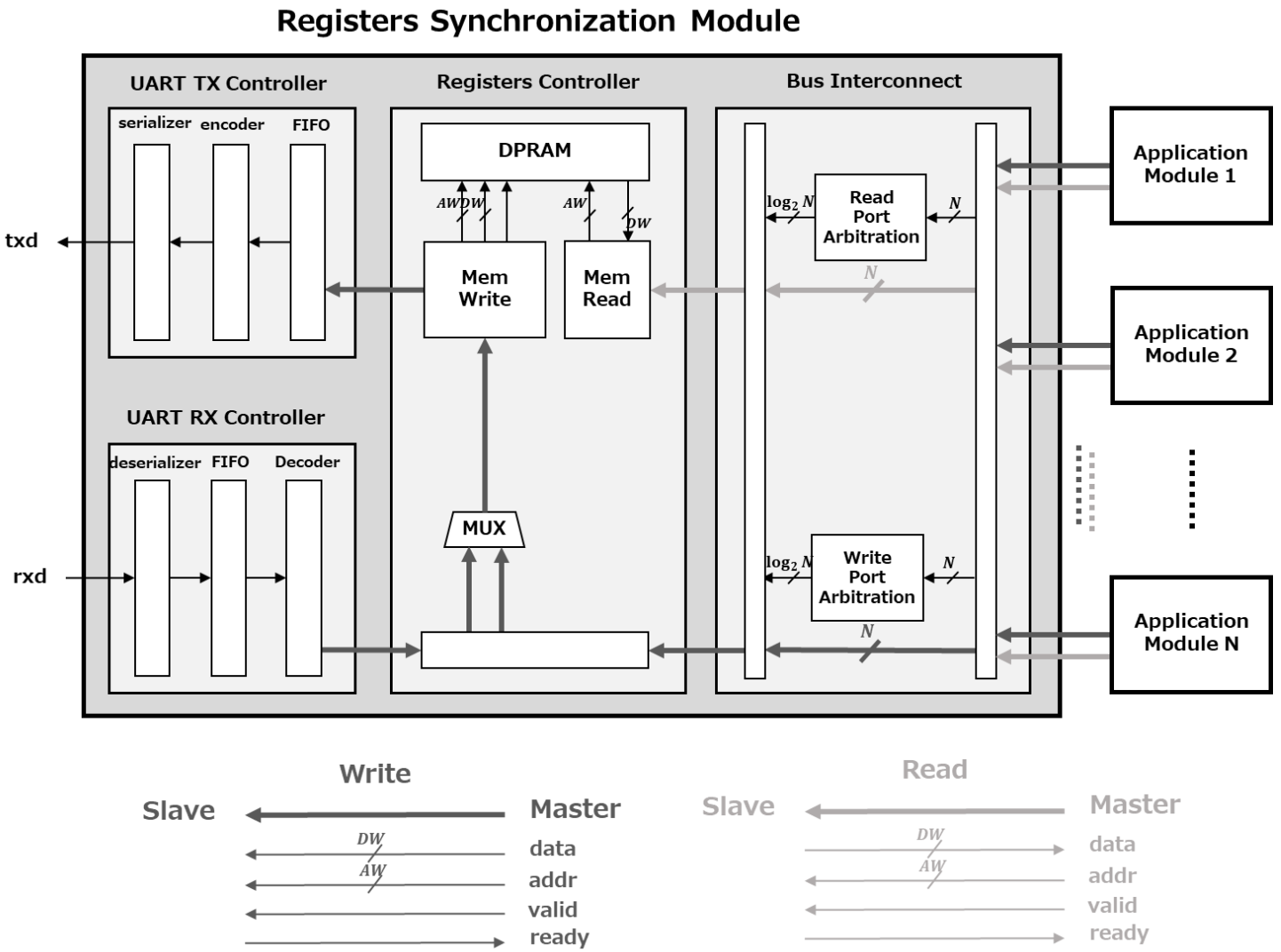


2.3. HEART2019 (2019-06-06) Ver.

2.2.2 システムアーキテクチャ



2.2.2 システムアーキテクチャ



2.3. HEART2019 (2019-06-06) Ver.

2.3.3 自動走行システム

- システムを抜本的に見直した
  - 機能の抽象度を意識したレイヤ構造を採用
  - 環境情報を事前に指定する設計に変更
    - 地図・制約・目標状態等を事前に用意しておき、システムの起動時に読み込む
    - コンテスト以外の環境でも、上記の情報を与えることで自動走行が可能に
    - 自己位置推定と経路計画、経路追従の実装に焦点を絞ることが出来る

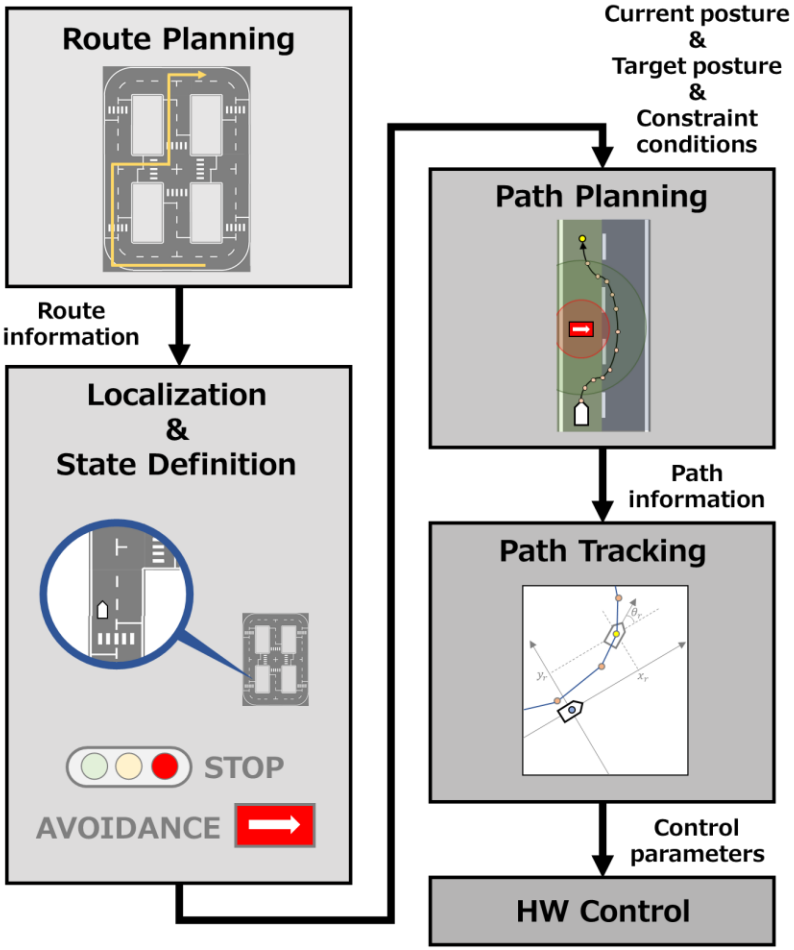
- 自己位置推定(HW/SW)
  - Wheel odometry
  - 特徴量マッチング　　<- new!

- 経路計画(SW)
  - RRT*

- 経路追従(SW)
  - PID + Pure Pursuit　　<- new!

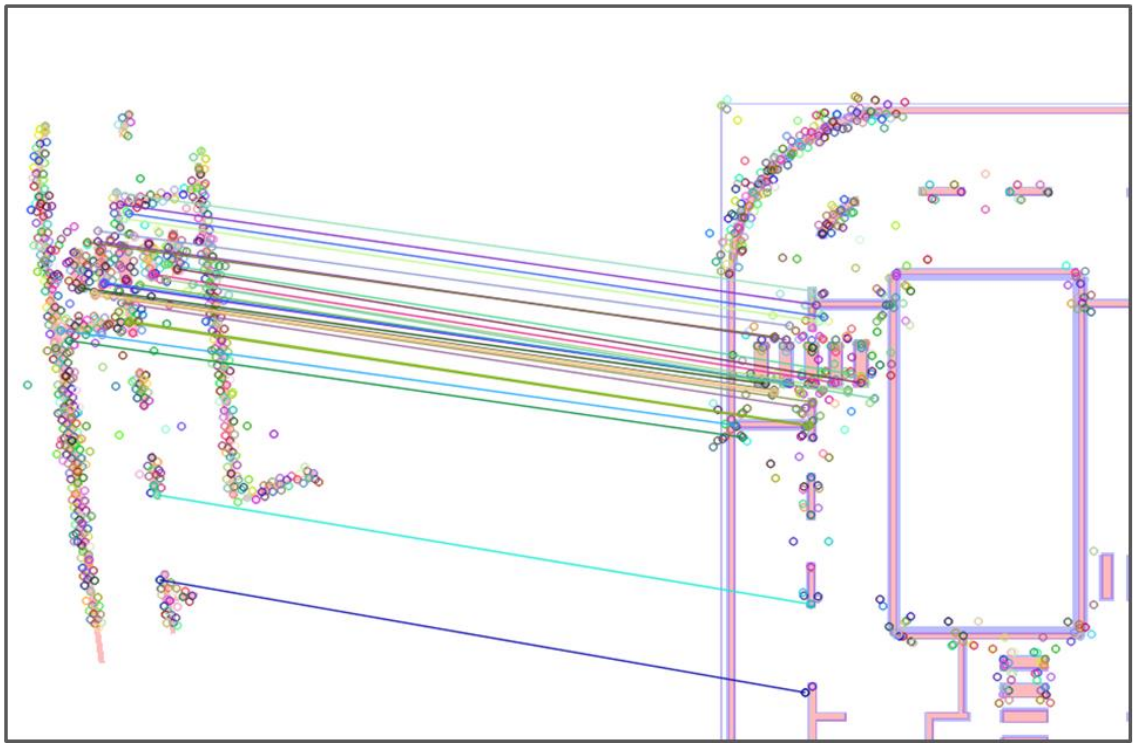
2.3.3 自動走行システム

- システムのレイヤ構造



2.3.3 自動走行システム

- 自己位置推定(HW/SW)
  - Wheel odometryに加えて、特徴量マッチングによる自己位置の補正を導入
  - 事前に用意していた地図とオンラインで取得した俯瞰画像を比較
  - 現在保持している自己位置に対する $\Delta x, \Delta y, \Delta\theta$を導出する



2.3.3 自動走行システム

- 経路追従(SW)
  - 生成した経路を追従するための車体制御を行う
  - 速度制御にPID、操舵角の制御にPure Pursuitを適用

$$\alpha = \tan^{-1}\left(\frac{y_{ref} - y_{current}}{x_{ref} - x_{current}}\right) - \theta_{current}$$

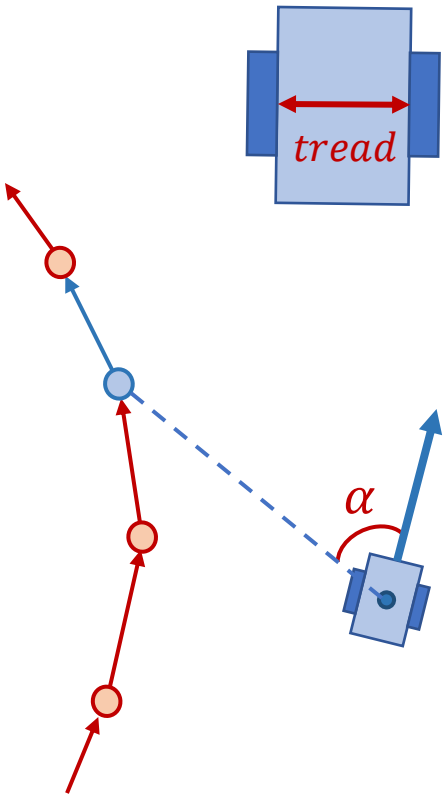
$$L = \sqrt{\left(x_{ref} - x_{current}\right)^2 + \left(y_{ref} - y_{current}\right)^2}$$

$$R = \frac{L}{2\sin\alpha}$$

$$w = \frac{v_{PID\_out}\sin\alpha}{L}$$

$$\omega_l = w(R - tread)$$

$$\omega_r = w(R + tread)$$



2.3. HEART2019 (2019-06-06) Ver.

2.3.4 性能評価

表. リソース使用数(Ultra96)

Module Name	LUT	FF	BRAM	LUT RAM	DSP
MIPI-CSI2 Decoder	2695(3.83%)	3777(2.68%)	4(1.85%)	0(0.00%)	0(0.00%)
Bayer2RGB	104(0.15%)	178(0.13%)	1(0.46%)	0(0.00%)	0(0.00%)
Gamma Correction	1138(1.61%)	36(0.03%)	0(0.00%)	0(0.00%)	0(0.00%)
AXI VDMA	1956(2.77%)	2478(1.76%)	2.5(1.16%)	165(0.57%)	0(0.00%)
RAM Synchronization Module	252(0.36%)	498(0.35%)	2(0.93%)	0(0.00%)	0(0.00%)
Preprocessing Module	2223(3.15%)	2049(1.45%)	9(4.17%)	47(0.16%)	9(2.50%)

2.3. HEART2019 (2019-06-06) Ver.

2.3.5 所感・考察

- システムレベルでの見直しを行った結果、実装の見通しが良くなり、また、汎用性・拡張性が向上した
- 一方で、自己位置推定の精度があまり高くないため、コンテストでは完走することが出来なかった
  - Wheel odometryは車体旋回時の精度が低い
- 課題として、自己位置推定の精度向上、経路追従の安定性の向上が挙げられる

3. 現在の取り組み

3.1. 自己位置推定

3.1.1. Monocular Visual Odometry

3.1.2. Particle Filter

3.2. 経路計画

3.2.1. Informed-RRT*

3. 現在の取り組み

3.1. 自己位置推定

3.1.1. Monocular Visual Odometry

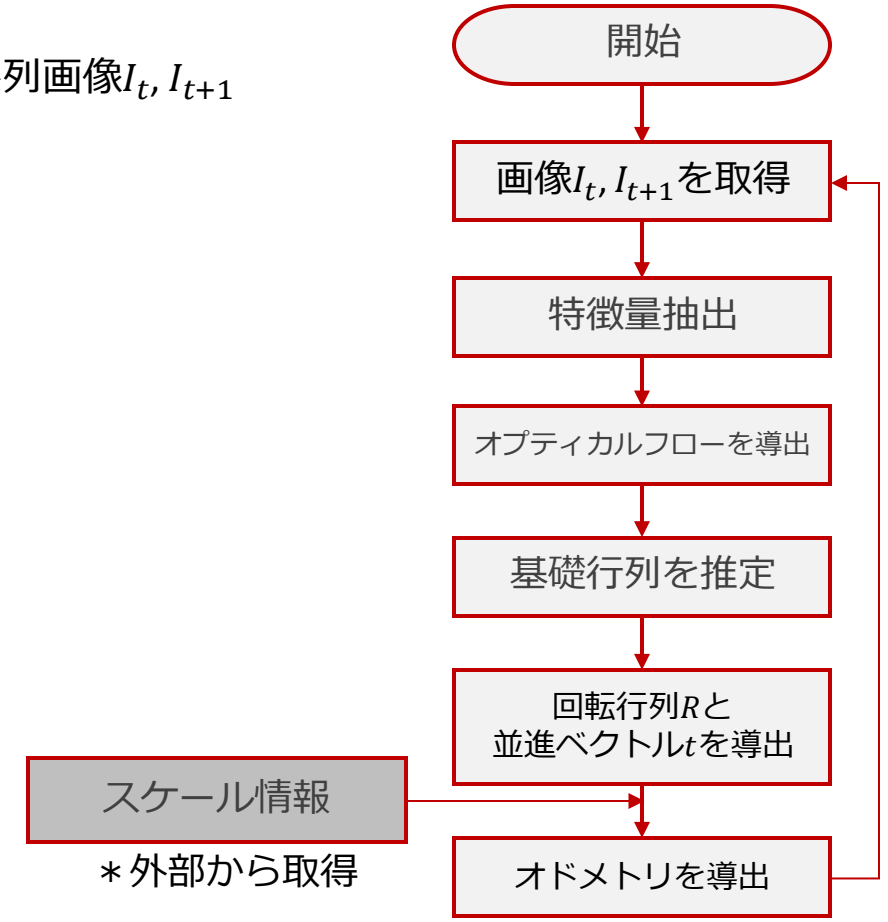
3.1.2. Particle Filter

3.2. 経路計画

3.2.1. Informed-RRT*

3.1.2. Monocular Visual Odometry

- 単眼カメラからオドメトリを導出
  - 入力
    - キャリブレーションされた時系列画像I_t, I_{t+1}
    - 移動量の絶対値
  - 出力
    - 回転行列R, 並進ベクトルt



3. 現在の取り組み

3.1. 自己位置推定

3.1.1. Monocular Visual Odometry

3.1.2. Particle Filter

3.2. 経路計画

3.2.1. Informed-RRT*

3.1.2. Particle Filter

- 動機

ロータリーエンコーダによって求められるWheel Odometry
のみを用いた状態の推定を行っていた

↓

車体のドリフトなどによる推定誤差が大きい

↓

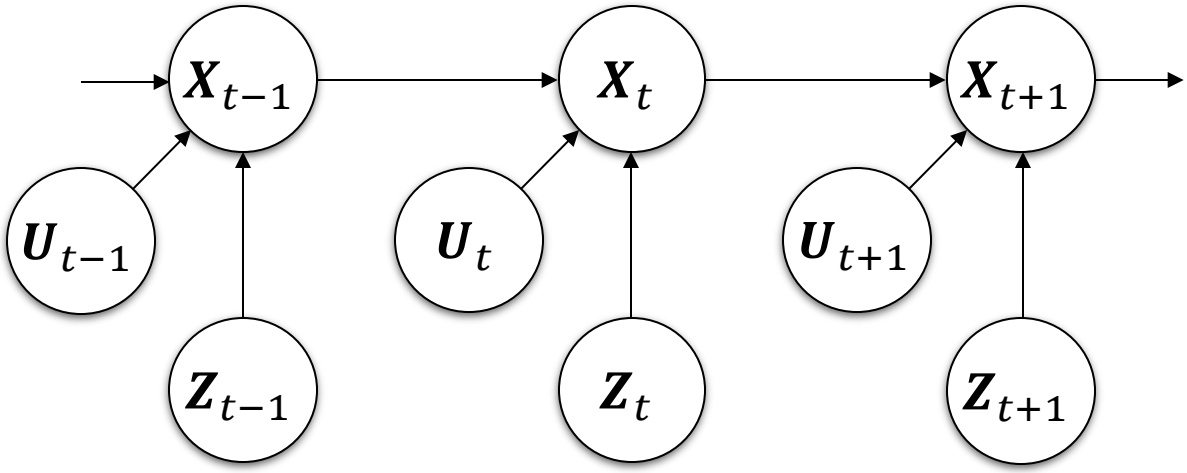
ロータリーエンコーダ以外のセンサも使用して、
状態の推定の精度を向上させたい

↓

ベイズフィルタを用いたセンサフュージョン

3.1.2. Particle Filter

- Particle Filterとは
  - ある未確定な状態の推測を行う逐次ベイズフィルタの一種
  - モンテカルロ法を用いる
  - 状態の推定に複数の観測モデルを考慮することが出来る



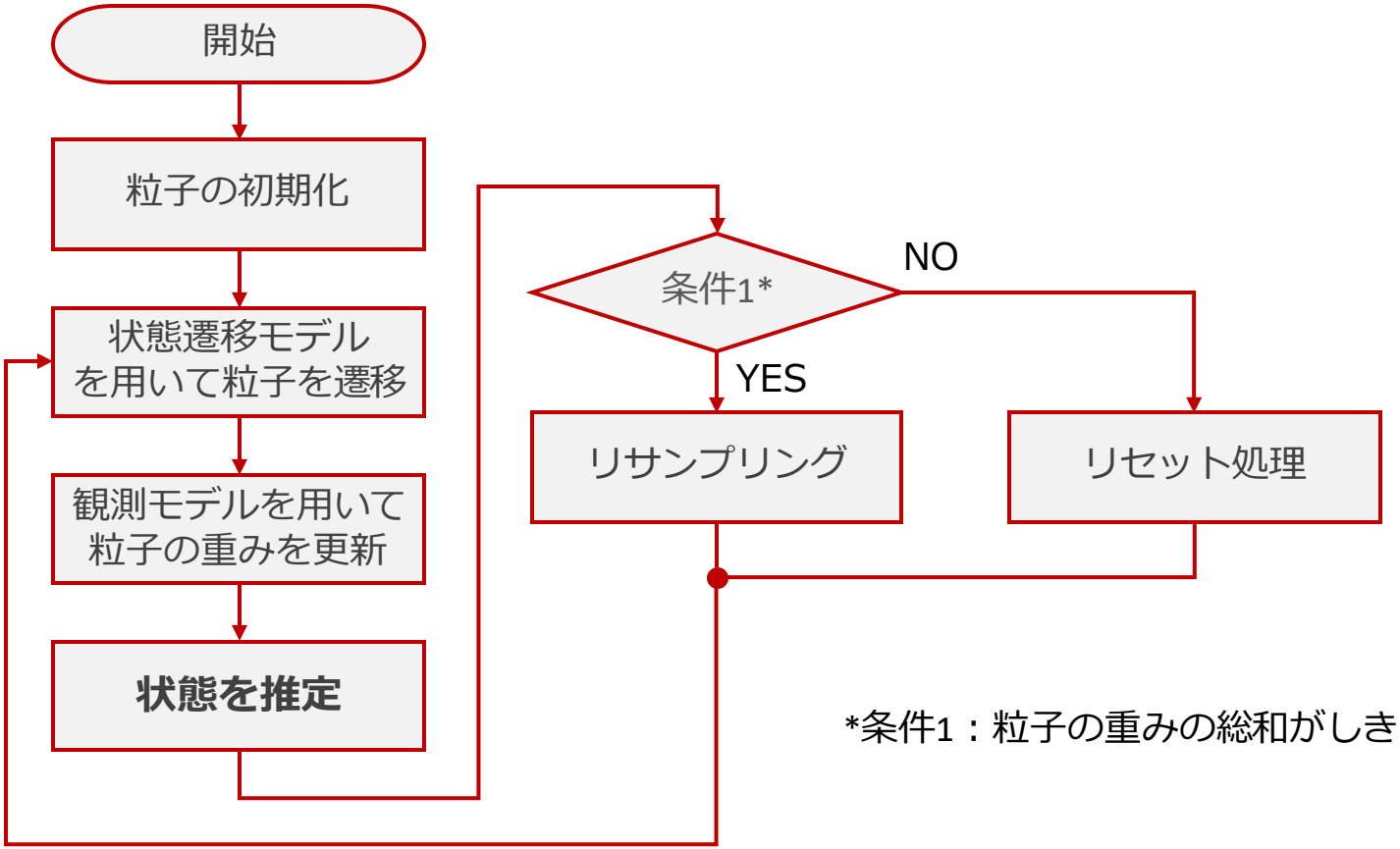
X_t：推測される状態
U_t：状態遷移モデル
Z_t：観測モデル

3.1.2. Particle Filter

- Particle Filterとは
  - 内部に多数の状態を持つ
    - 状態のそれぞれを粒子と呼ぶ
  - それぞれの粒子は重みを持つ
    - **次状態として尤もらしい粒子ほど重みは大きくなる**
  - 全ての粒子は状態遷移モデルによって遷移する
    - このとき、遷移にはガウスノイズを付与する
    - 一般的に、車輪ロボットの自己位置推定において、状態遷移モデルにはWheel Odometryを適用する
  - 全ての粒子と観測モデルを用いて状態を推定する
    - 観測モデルは複数適用することが出来る
    - 一般的に、観測モデルには、Visual odometry、GPSによって得られる変位、ジャイロセンサや加速度センサによって得られる変位などを適用する

3.1.2. Particle Filter

- 今回実装したParticle Filterの処理の流れ



*条件1：粒子の重みの総和がしきい値以上

3.1.2. Particle Filter

- 定義
  - ある時点tにおける車体の状態X_tを以下のように表す

$$X_t = (x_t, y_t, \theta_t)$$

  - ある時点tにおける粒子の状態P_tを以下のように表す

$$P_t = (x_t, y_t, \theta_t, w_t)$$

  - ある時点tにおけるWheel Odometryを以下のように表す

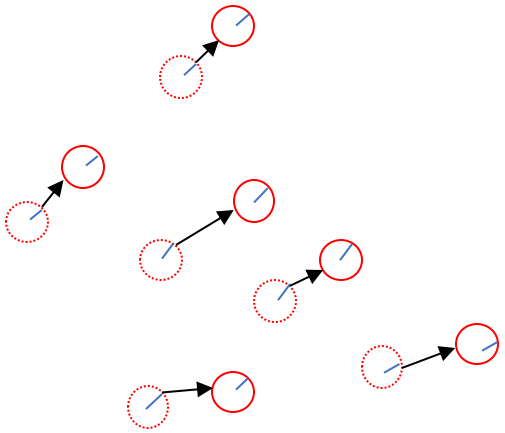
$$wo_t = (dx_t, dy_t, d\theta_t)$$

  - ある時点tにおけるMonocular Visual Odometryを以下のように表す

$$vo_t = (dx_t, dy_t, d\theta_t)$$

3.1.2. Particle Filter

- 状態遷移モデルを用いた予測
  - 各パーティクルを状態遷移関数に基づいて移動させる
    - この時、システムノイズを付与する
    - 一般的にはガウスノイズを用いる



wo ：Wheel Odometry

f ：状態遷移関数

$v \sim N(m, \sigma^2)$ ：ガウス分布(平均m, 分散 σ^2)

$$P_t = f(P_{t-1}, wo_{t-1}) + v$$

$$= \begin{bmatrix} x_{t-1} & 0 & 0 \\ 0 & y_{t-1} & 0 \\ 0 & 0 & \theta_{t-1} \end{bmatrix} + \begin{bmatrix} dx_{t-1} & 0 & 0 \\ 0 & dy_{t-1} & 0 \\ 0 & 0 & d\theta_{t-1} \end{bmatrix} + v$$

3.1.2. Particle Filter

- 観測モデルを用いて粒子の重みを更新
  - 尤度関数の設計
    - 尤もらしい解付近で尤度が最大化されること
    - 現在の観測に対してロバストであること
  - 今回設計する尤度関数
    - 確率密度関数$f(x)$は下式である

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

    - 正規分布に従うと仮定すると尤度関数Lは下式で表される

$$L(\text{類似度}, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(\text{類似度}-\mu)^2}{2\sigma^2})$$

3.1.2. Particle Filter

- 観測モデルを用いて粒子の重みを更新
  - 観測モデルz_tに観測ノイズを付与
    - 観測モデルz_tは**Monocular Visual Odometry**を使用する

$$z_t = X_{t-1} + vo_{t-1} + w \qquad w \sim N(m, \sigma^2)：観測ノイズ$$

$$= \begin{bmatrix} x_{t-1} & 0 & 0 \\ 0 & y_{t-1} & 0 \\ 0 & 0 & \theta_{t-1} \end{bmatrix} + \begin{bmatrix} dx_{t-1} & 0 & 0 \\ 0 & dy_{t-1} & 0 \\ 0 & 0 & d\theta_{t-1} \end{bmatrix} + w$$

3.1.2. Particle Filter

- 観測モデルを用いて粒子の重みを更新
  - 観測モデルと予測パーティクルの距離（誤差）を算出
    - 位置に関して

$$\mathrm{d} = \sqrt{{z_t}^2 - {P_t}^2}$$

    - 向きに関して

$$\mathrm{m} = |z_t - P_t| * \mathrm{k} \qquad \mathrm{k}: \text{定数}$$

  - 位置に関して確率密度を算出

$$\mathrm{L}(d, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(d-\mu)^2}{2\sigma^2})$$

  - 向きに関して確率密度を算出

$$\mathrm{L}(m, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(m-\mu)^2}{2\sigma^2})$$

3.1.2. Particle Filter

- 観測モデルを用いて粒子の重みを更新
  - 各パーティクルを更新

$$w_t = w_{t-1} * \mathrm{L}(d, \mu, \sigma^2) * \mathrm{L}(m, \mu, \sigma^2)$$

  - 重みの正規化

$$w_t^m = \frac{w_t^m}{\sum_{m=1}^{M} w_t^m} \quad , for\ m = 1, \ldots, M$$

3.1.2. Particle Filter

- 状態推定
  - 重み付き平均で推定する

$$X_t = \sum_{m=1}^{M} w_t^m * P_t^m$$

  - 最大重みで推定する

$$X_t = P_t^{\,\mathrm{argmax}\,(w_t^m)}$$

＊予測よりも観測の方が十分に信用できる場合に，

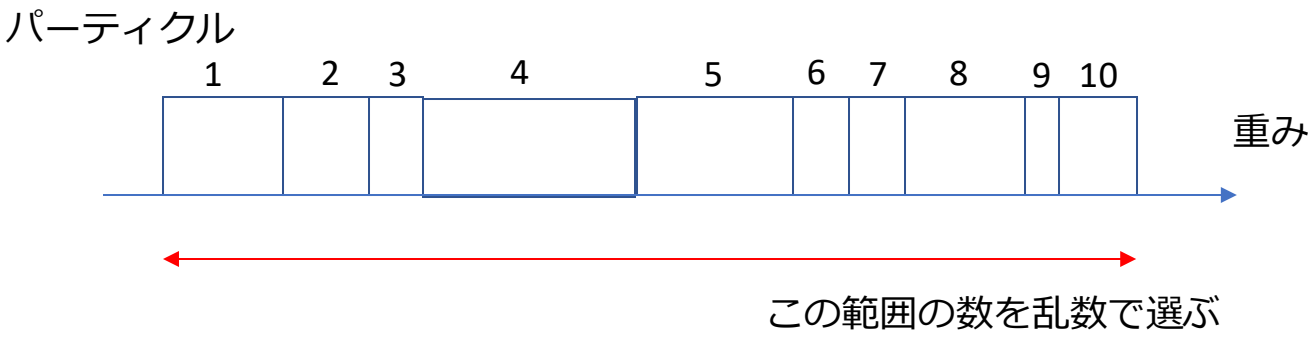
近似的に最大重み推定が利用できる

3.1.2. Particle Filter

- リサンプリング
  - 予測・尤度計算を単純に繰り返ししまうていくと,
    重みがほぼ0の無意味なパーティクルだらけになる
  - パーティクルを再度サンプリングして,
    一様な重みのサンプルの集合で表現し直す
  - リサンプリングの手法
    - 単純ランダムサンプリング
    - 層別サンプリング
    - 等間隔サンプリング ← 今回使用

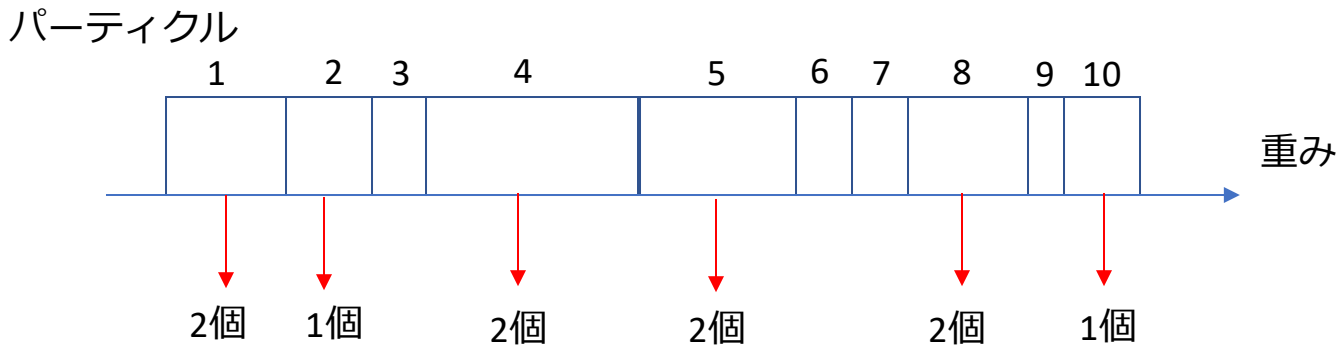
3.1.2. Particle Filter

- リサンプリング
  - 単純ランダムサンプリング
    - 1. 重みの和を求める
    - 2. 区間[0, 重みの和]で乱数生成
    - 3. 元のパーティクルの重みの累積度と
      乱数を比べて釣り合う順番のパーティクルを抽出
    - 4. 選んだパーティクルの場所に重み1/Nのパーティクルを置く



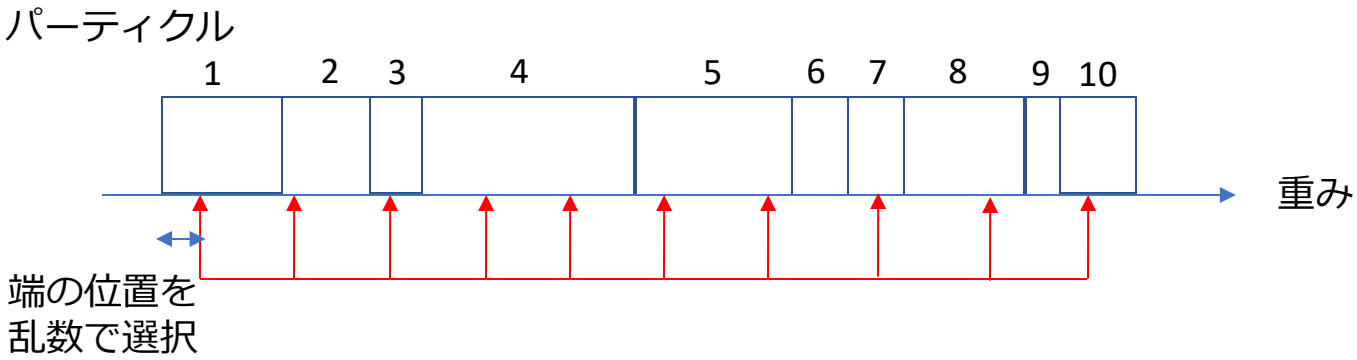
3.1.2. Particle Filter

- リサンプリング
  - 層別サンプリング
    - 更新前のパーティクルの重みに比例して新たなパーティクルを生成



3.1.2. Particle Filter

- リサンプリング
  - 等間隔サンプリング
    - 幅w：1/重みの合計
    - 最初のサンプリング点：区間[0, 重みの合計]でランダムに選択



3.1.2. Particle Filter

- リセット
  - パーティクルとロボットの位置の乖離が疑われた時に使用
    - パーティクルの**重みの総和**が任意のしきい値以下
  - 単純リセット
    - 過去の推定、センサの情報をすべて消去して
      推定しなおす。発動条件を厳しくする
  - センサリセット
    - 過去の推定よりもセンサの情報を重視（センサがGPSなら有効）
  - 膨張リセット ←今回使用
    - パーティクルの分布を膨張させる

3. 現在の取り組み

3.1. 自己位置推定

3.1.1. Monocular Visual Odometry

3.1.2. Particle Filter

3.2. 経路計画

3.2.1. Informed-RRT*

3.2.1 Informed-RRT*

- Informed-RRT*(Informed Optimal Rapidly-exploring Random Tree)
  - RRT*にヒューリスティックを適用したもの
  - A*, LPA*(Lifelong Planning A*)の考え方を応用
  - 状態$x \in X$を引数とする以下のヒューリスティック関数を定義

$$\hat{f}(x) \coloneqq \hat{g}(x) + \hat{h}(x)$$

$$\hat{g}(x) \coloneqq |x_{start} - x|$$

$$\hat{h}(x) \coloneqq |x - x_{goal}|$$

  - 状態空間$X_{\hat{f}}$を以下のように定義する

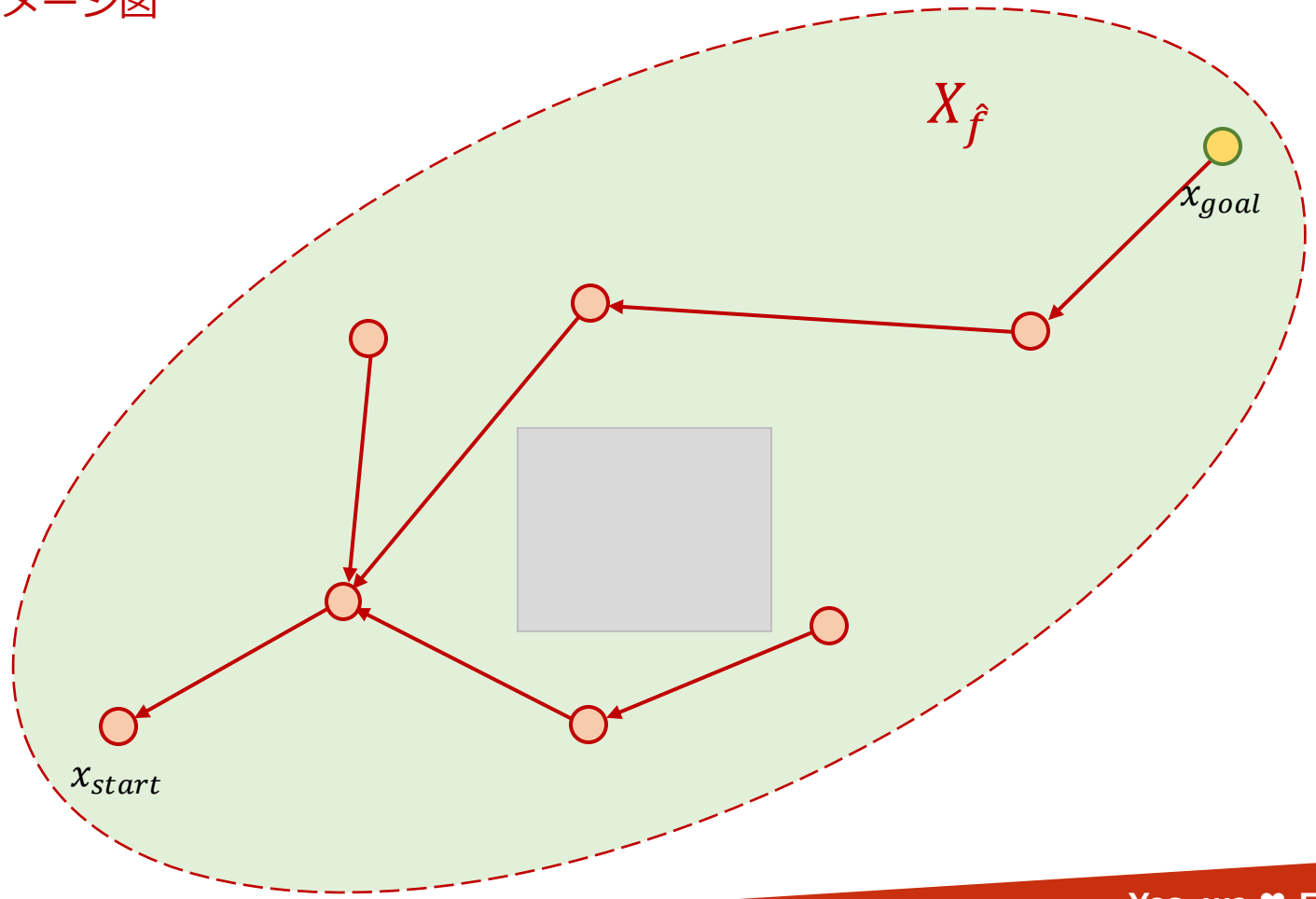
$$X_{\hat{f}} \coloneqq \{x \in X | \hat{f}(x) < c_{best}\}$$

  - c_{best}は、その時点における開始状態から目標状態までの頂点経由のユークリッド距離（目標状態まで到達できない場合は∞とする）

  - **$\overline{X_{\hat{f}}}$をサンプリングしても、目標状態までの最短距離が更新されることはない**

3.2.1 Informed-RRT*

- Informed-RRT*(Informed Optimal Rapidly-exploring Random Tree)
  - RRT*にヒューリスティックを適用したアルゴリズム
  - イメージ図



4. 今後の課題

4.1. 自己位置推定の精度の改善

- センサーフュージョンの観測モデルに適用する移動量を追加・改善する
  - IMU
  - Stereo VO
- 尤度関数の見直し
  - 複数のセンサ情報を上手く組み合わせる

4.2. 経路計画・経路追従の高速化

- Informed-RRT*のFPGA実装