Future Robotics Technology Center

千葉工業大学

ROSを用いた自律移動ロボットのシステム構築

千葉工業大学 未来ﾛﾎﾞｯﾄ技術研究ｾﾝﾀｰ

原 祥尭（HARA, Yoshitaka）

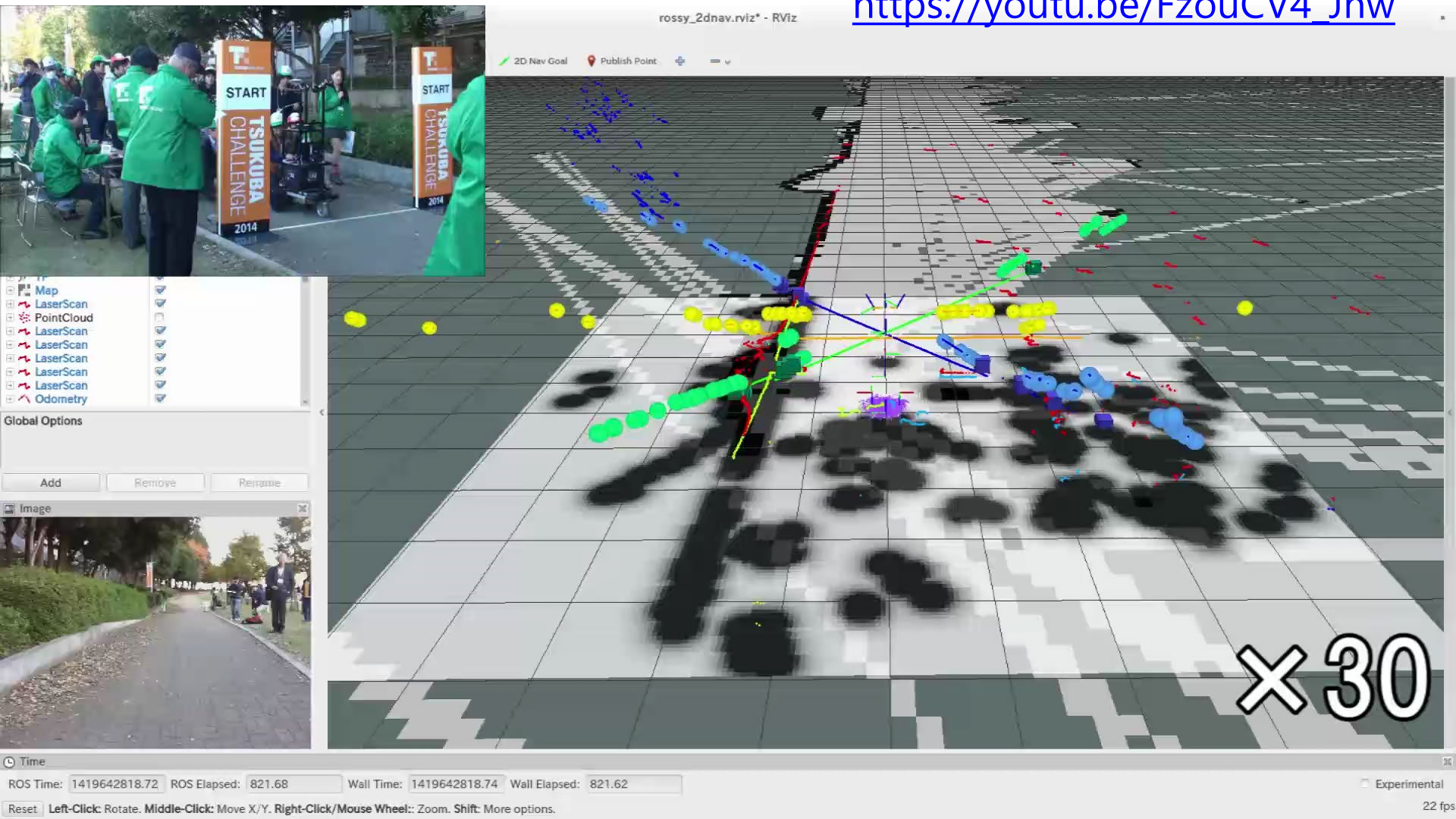
@ystk_hara

https://www.slideshare.net/hara-y/ros-nav-rsj-seminar

RSJ 第131回ロボット工学セミナー 2021-01-23

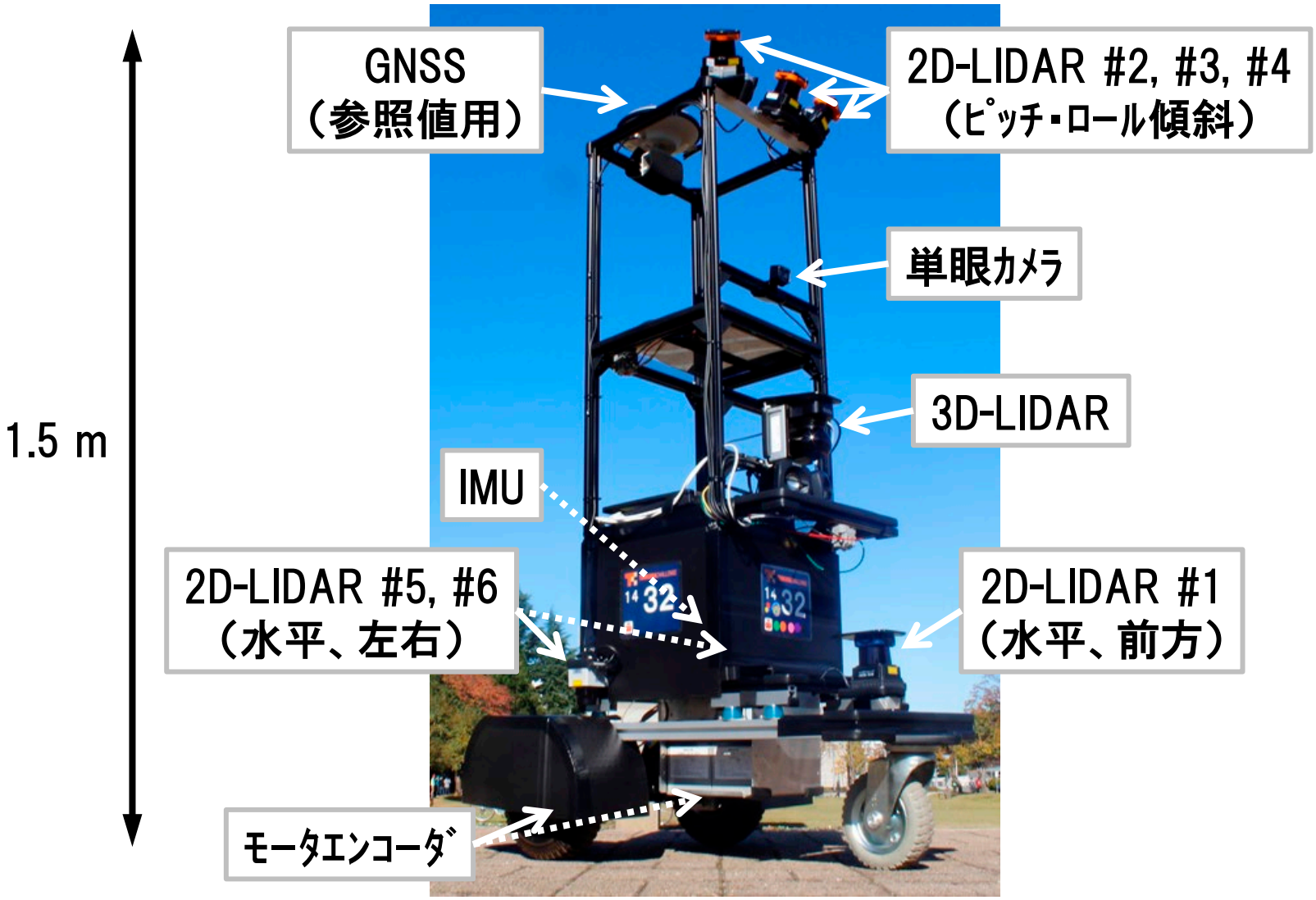
つくばチャレンジでの自律走行の様子

https://youtu.be/FzouCV4_Jnw

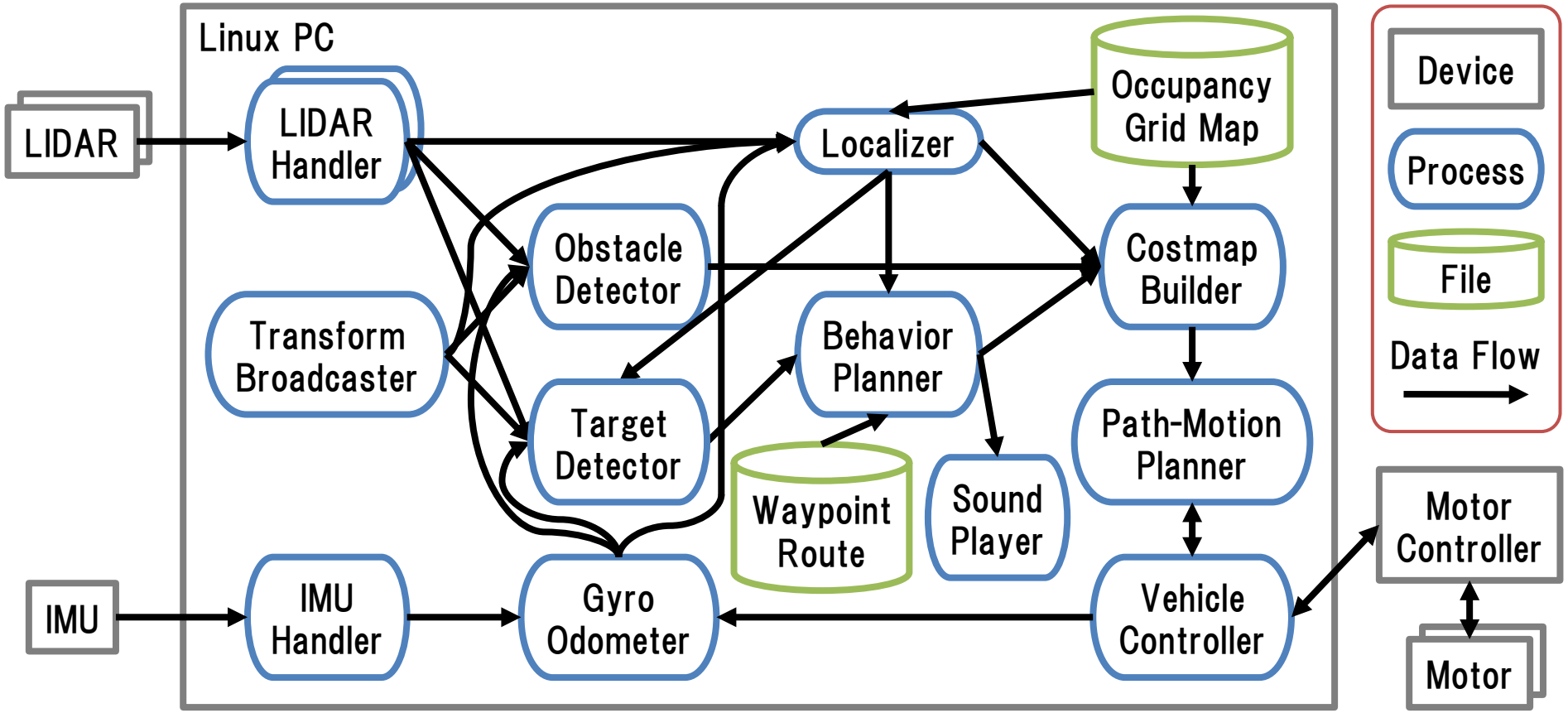


千葉工業大学 未来ロボット技術研究センター
Future Robotics Technology Center

開発したロボットの外観と搭載センサ



自律走行システムの構成図



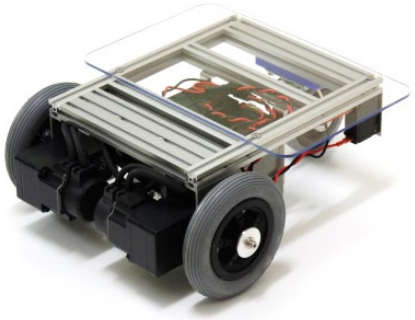
千葉工業大学 未来ロボット技術研究センター
Future Robotics Technology Center

Contents

1. 自律走行を実現するための ROS ﾊﾟｯｹｰｼﾞ
2. 自己位置推定と SLAM の基本
3. 各ﾊﾟｯｹｰｼﾞのｱﾙｺﾞﾘｽﾞﾑ
   a. 自己位置推定（amcl）
   b. SLAM、地図生成（gmapping）
   c. 大域的経路計画、局所的動作計画（move_base）
4. ROS に関する情報の調べ方
5. ROS での開発に関する知見
6. まとめ

自律走行に使用できるﾃﾞﾊﾞｲｽとﾊﾟｯｹｰｼﾞ

- **ｾﾝｻ** http://wiki.ros.org/Sensors
- **移動ﾛﾎﾞｯﾄ台車** http://wiki.ros.org/Robots
  - **T-frog i-Cart mini**（T-frog ﾓｰﾀｺﾝﾄﾛｰﾗ）
    http://t-frog.com/
  - **T-frog i-Cart edu**（T-frog ﾓｰﾀｺﾝﾄﾛｰﾗ）
    http://t-frog.com/



- **T-frog ﾓｰﾀｺﾝﾄﾛｰﾗ ROS 対応ﾊﾟｯｹｰｼﾞ**
  - **ypspur_ros**：公式ﾊﾟｯｹｰｼﾞ（SEQSENSE 渡辺先生）
    https://github.com/openspur/ypspur_ros
  - **ypspur_ros_bridge**：旧ﾊﾟｯｹｰｼﾞ（筑波大 ﾛﾎﾞ研）
    http://www.roboken.iit.tsukuba.ac.jp/platform/gitweb/
  - **open-rdc icart_mini**：Gazebo 対応（千葉工大 林原研）
    https://github.com/open-rdc/icart

ROS で使える3次元ｼﾐｭﾚｰﾀ

自律走行に関するﾊﾟｯｹｰｼﾞとその機能

- **slam_gmapping**
  - SLAM、地図生成
- **navigation**
  - 自己位置推定
  - 占有格子地図生成、障害物情報の管理
  - 大域的経路計画
  - 局所的動作計画（障害物回避）

自律走行に必要な機能をひととおり提供、C++で実装されている

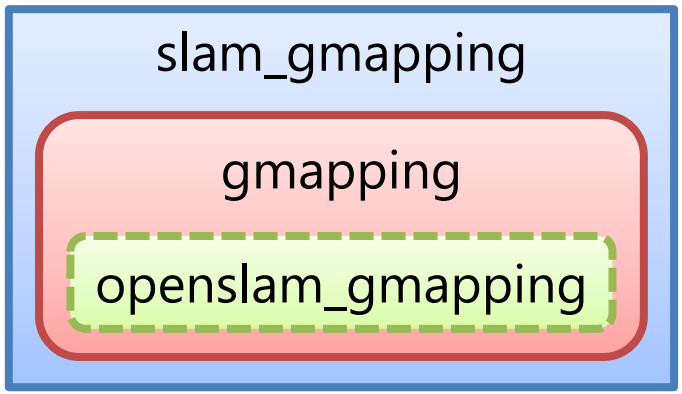
自律走行を行う際のワークフロー

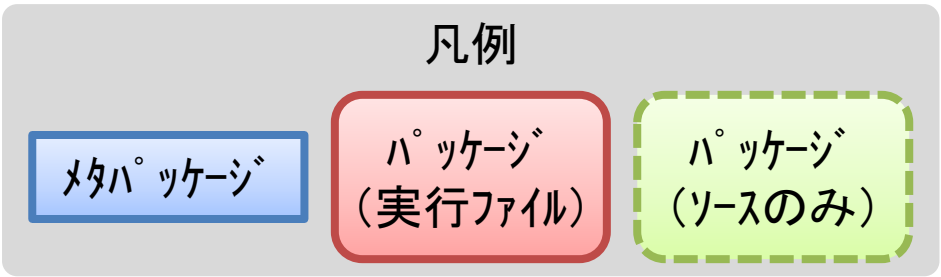
1. ﾛﾎﾞｯﾄを手動走行させて環境のｾﾝｻﾃﾞｰﾀを取得、bag ﾌｧｲﾙとして保存

2. bag ﾌｧｲﾙを再生し、**slam_gmapping** ﾊﾟｯｹｰｼﾞを使用してｵﾌﾗｲﾝで地図生成（ｵﾝﾗｲﾝも可能）

3. 生成した地図に基づき、**navigation** ﾊﾟｯｹｰｼﾞを使用して自律走行

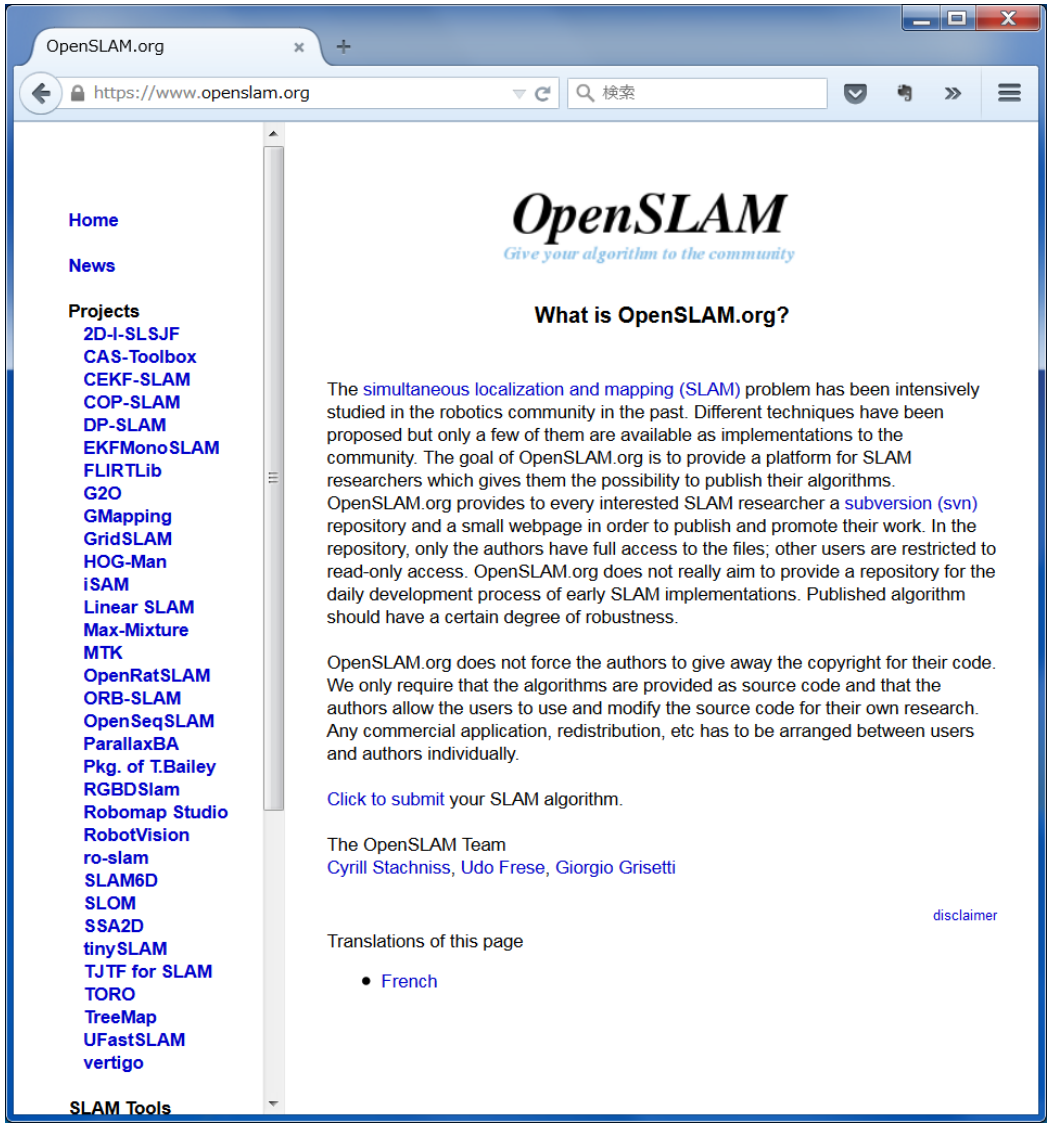
slam_gmapping ﾊﾟｯｹｰｼﾞの構成





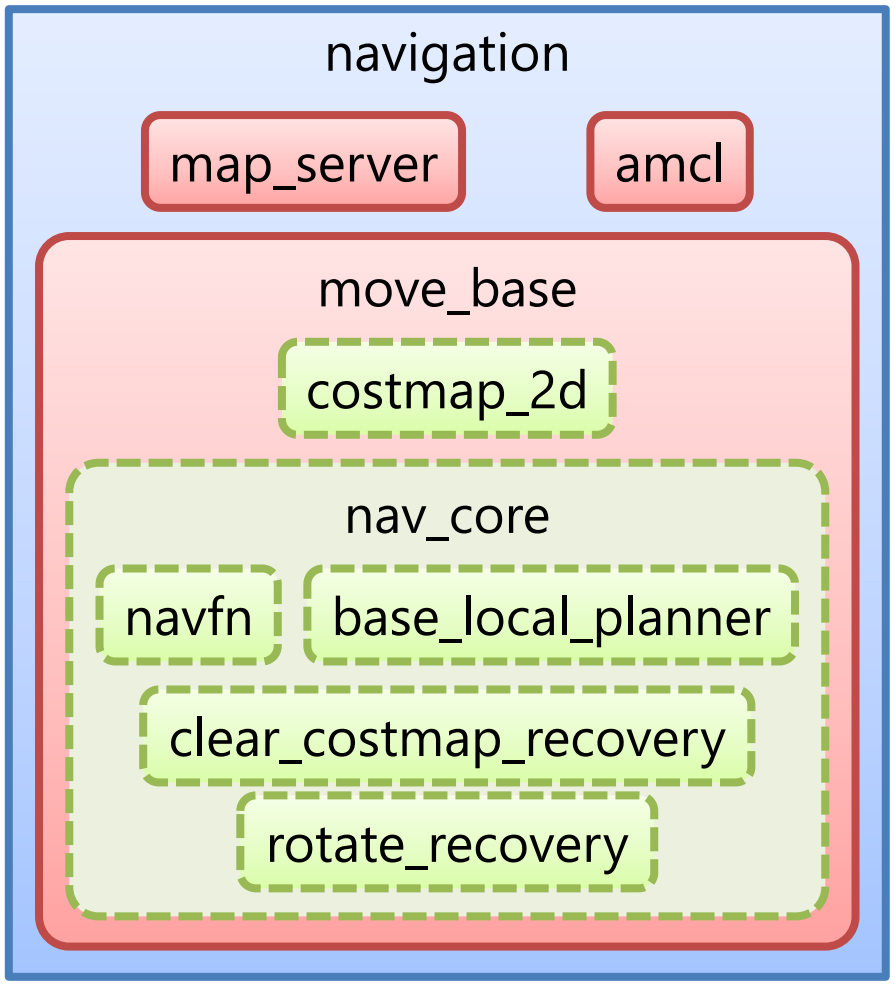
- **gmapping**：SLAM、地図生成の実行（ROSﾗｯﾊﾟｰ）
  - **openslam_gmapping**：
    OpenSLAM で公開されている Rao-Blackwellized Particle Filter による Grid-based SLAM
    （FastSLAM 2.0 での Grid Mapping）

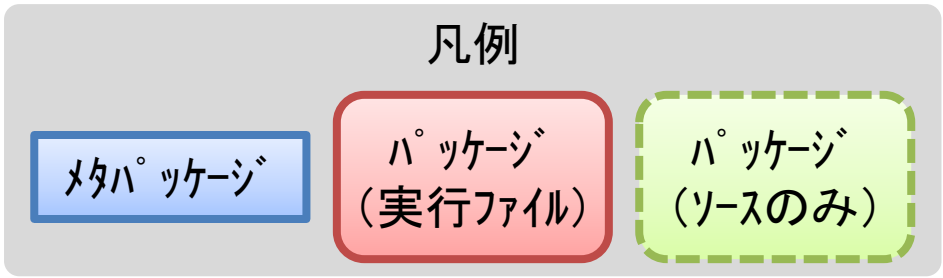
OpenSLAM とは



- SLAM の各種ｱﾙｺﾞﾘｽﾞﾑをｵｰﾌﾟﾝｿｰｽで公開する Web ｻｲﾄ
- EKF, RBPF, Graph-based SLAM, ICP ﾏｯﾁﾝｸﾞなど複数の手法の実装あり
- 研究者が各手法を比較するために公開
- ﾗｲﾌﾞﾗﾘではない

navigation ﾊﾟｯｹｰｼﾞの構成





- **map_server**：既存地図の配信
- **amcl**：自己位置推定
- **move_base**：移動ﾀｽｸの実行
  - **costmap_2d**：ｺｽﾄﾏｯﾌﾟの生成
  - **nav_core**：行動ｲﾝﾀﾌｪｰｽ定義
    - **navfn**：大域的経路計画
    - **base_local_planner**：局所的動作計画
    - ***_recovery**：ｽﾀｯｸした場合の復帰動作

nav_core によるﾊﾟｯｹｰｼﾞの入れ替え

- ﾍｯﾀﾞﾌｧｲﾙでのｲﾝﾀﾌｪｰｽ定義のみ（純粋仮想関数）
  - 大域的経路計画：BaseGlobalPlanner
  - 局所的動作計画：BaseLocalPlanner
  - ｽﾀｯｸ復帰動作：RecoveryBehavior

- 実装は move_base のﾊﾟﾗﾒｰﾀで実行時に変更可能

- ROS の pluginlib という機能で実現されている

入れ替え可能なﾊﾟｯｹｰｼﾞとそのｱﾙｺﾞﾘｽﾞﾑ

- **大域的経路計画（BaseGlobalPlanner）**
  - navfn：ﾀﾞｲｸｽﾄﾗ法
  - global_planner：A*／ﾀﾞｲｸｽﾄﾗ法
  - carrot_planner：ｺﾞｰﾙに向かって直進
- **局所的動作計画（BaseLocalPlanner）**
  - base_local_planner：Dynamic Window Approach
  - dwa_local_planner：Dynamic Window Approach の別実装
- **ｽﾀｯｸ復帰動作（RecoveryBehavior）**
  - clear_costmap_recovery：ｺｽﾄﾏｯﾌﾟをｸﾘｱ
  - rotate_recovery：ﾛﾎﾞｯﾄが旋回して障害物を測定
  - move_slow_and_clear：ｺｽﾄﾏｯﾌﾟをｸﾘｱ、移動速度を制限

標準で使用されるｱﾙｺﾞﾘｽﾞﾑのまとめ

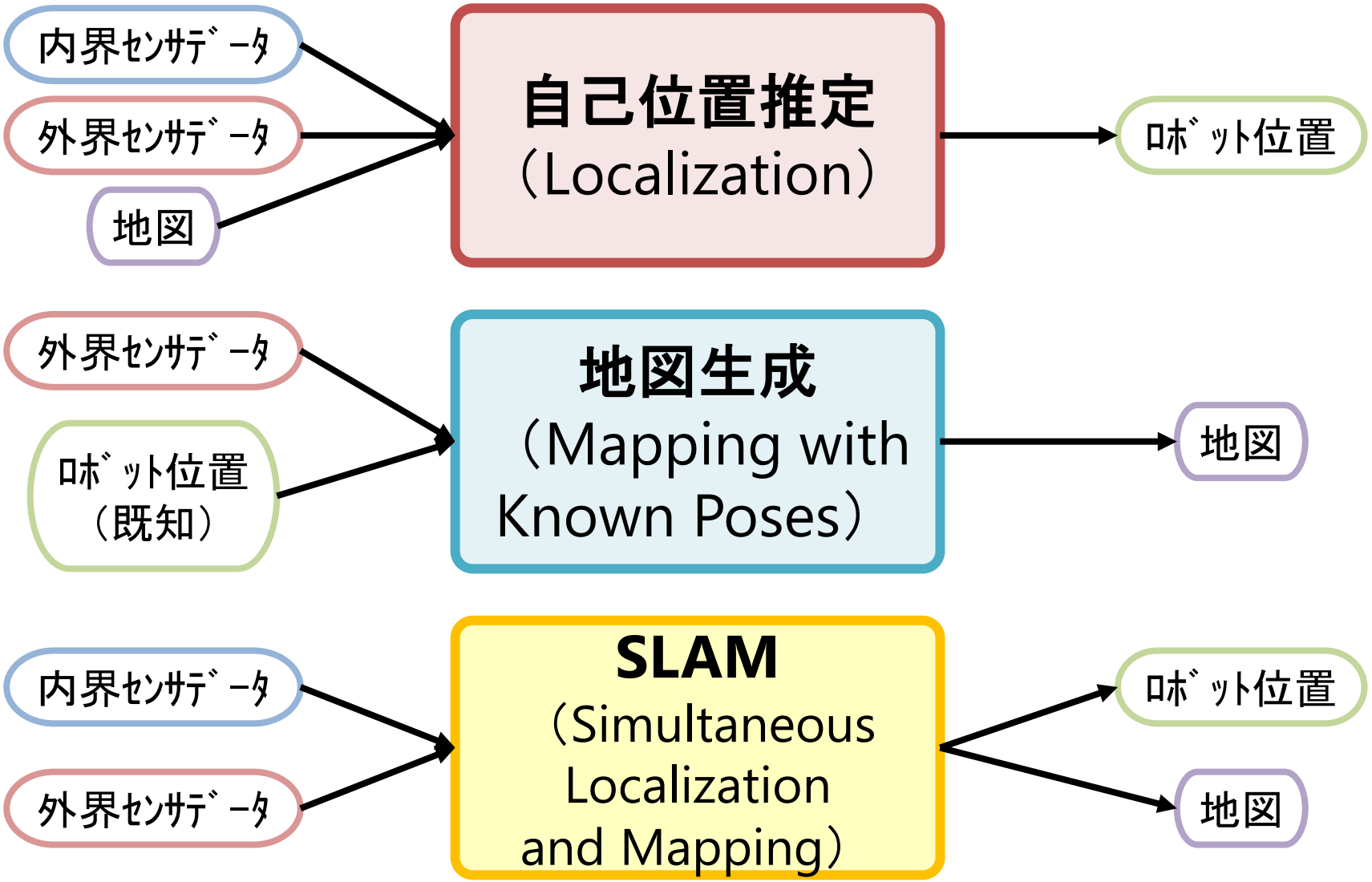
- **SLAM、地図生成（gmapping）**
  Rao-Blackwellized Particle Filter による Grid-based SLAM（FastSLAM 2.0 での Grid Mapping）
- **自己位置推定（amcl）**
  Adaptive/Augmented Monte Carlo Localization
- **経路計画（move_base）**
  Global Dynamic Window Approach
  - **大域的経路計画（navfn）**
    Navigation Function に基づくﾀﾞｲｸｽﾄﾗ法
  - **局所的動作計画（base_local_planner）**
    Dynamic Window Approach

Contents

1. 自律走行を実現するための ROS ﾊﾟｯｹｰｼﾞ
2. 自己位置推定と SLAM の基本
3. 各ﾊﾟｯｹｰｼﾞのｱﾙｺﾞﾘｽﾞﾑ
   a. 自己位置推定（amcl）
   b. SLAM、地図生成（gmapping）
   c. 大域的経路計画、局所的動作計画（move_base）
4. ROS に関する情報の調べ方
5. ROS での開発に関する知見
6. まとめ

千葉工業大学 未来ロボット技術研究センター
Future Robotics Technology Center

自己位置推定、地図生成、SLAM とは



自己位置推定と SLAM の手法

	スキャンマッチング系		Bayes Filter 系	Graph-based SLAM 系
性質	逐次最適化、オンライン		フィルタリング、オンライン	全体最適化、オフライン
手法の概要	点群を最適化計算で位置合わせ		事前確率と尤度を確率的に融合	ﾛﾎﾞｯﾄ位置やﾗﾝﾄﾞﾏｰｸ位置（地図）を表すｸﾞﾗﾌを最適化
	詳細ﾏｯﾁﾝｸﾞ	大域ﾏｯﾁﾝｸﾞ		
	初期位置あり、ﾕｰｸﾘｯﾄﾞ空間で対応付け	初期位置なし、特徴量空間で対応付け		
手法の例	ICP、NDT など	Spin Image、FPFH、PPF、SHOT など	EKF、HF、**PF**、**RBPF** など	ﾎﾟｰｽﾞ調整、ﾊﾞﾝﾄﾞﾙ調整、完全 SLAM

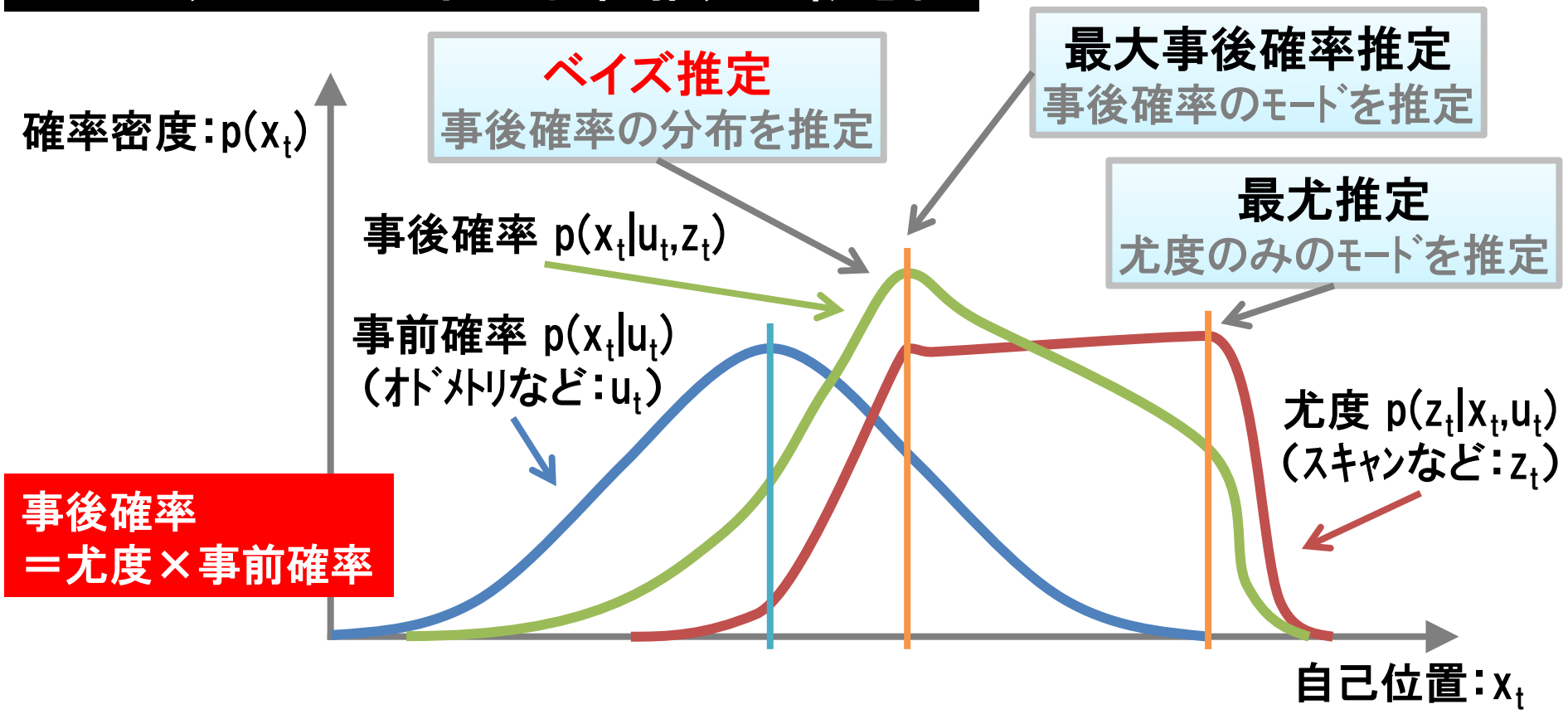
以下のﾊﾟｯｹｰｼﾞは一部の手法を提供しているに過ぎない

- amcl：Particle Filter (PF) による Monte Carlo Localization
- gmapping：Rao-Blackwellized Particle Filter (RBPF)

SLAM の性能は屋外環境に対して不充分な場合も多い

最尤推定、最大事後確率推定、ﾍﾞｲｽﾞ推定

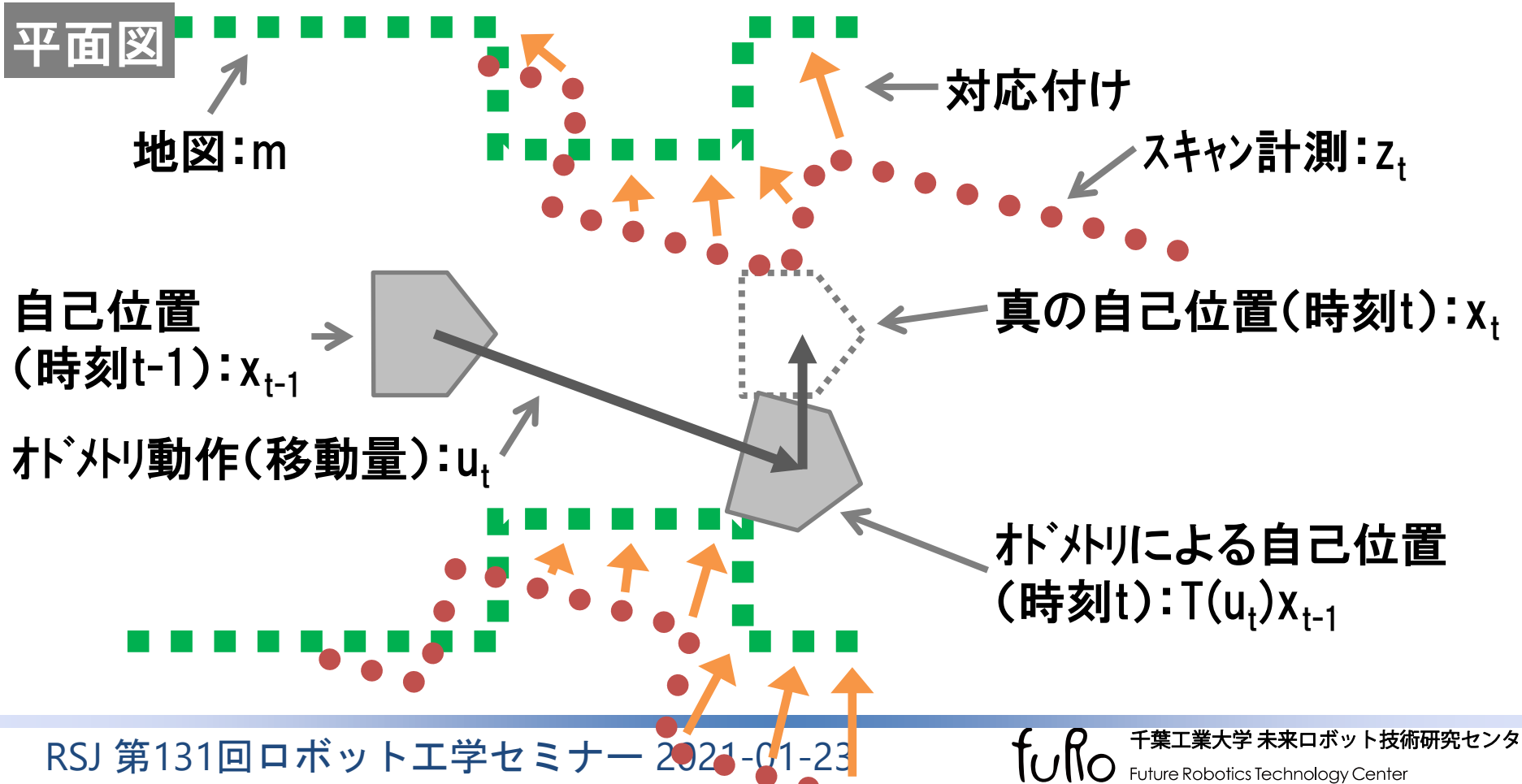
ﾍﾞｲｽﾞの定理による自己位置推定の概念図



Bayes Filter は、ﾍﾞｲｽﾞの定理に基づくﾍﾞｲｽﾞ推定によって自己位置の事後確率分布を求める

Bayes Filter による自己位置推定

- ロボットの自己位置の確率分布を求める
- 地図は事前に与えられている



確率的自己位置推定の隠れﾏﾙｺﾌﾓﾃﾞﾙ

自己位置の時系列を1次ﾏﾙｺﾌ過程と仮定

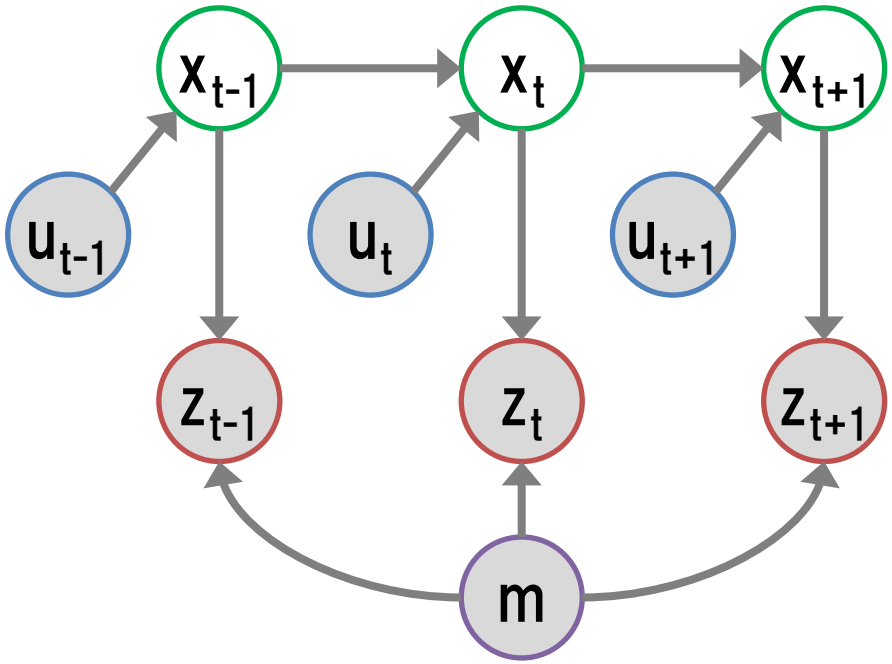
自己位置推定のｸﾞﾗﾌｨｶﾙﾓﾃﾞﾙ

robot pose: x
(hidden variable)

motion: u
(e.g. odometry)

measurement: z
(e.g. laser scan)

map: m



Bayes Filter の定式化

- 漸化式で時系列の確率過程を表現
- Extended Kalman Filter、Histogram Filter などはすべて Bayes Filter の理論に基づいた実装

$$p(\boldsymbol{x}_t \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{m})$$ ← 時刻tの自己位置x_tの事後確率

$$= \eta\, p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m})$$ ← 尤度(計測モデル)

$$\cdot \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)\; p(\boldsymbol{x}_{t-1} \mid \boldsymbol{u}_{1:t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m}) d\boldsymbol{x}_{t-1}$$

動作モデル　　事前確率　　時刻t-1の自己位置x_{t-1}

x：ﾛﾎﾞｯﾄ位置
u：制御動作(ｵﾄﾞﾒﾄﾘなど)
m：地図
z：環境計測(ﾚｰｻﾞｽｷｬﾝなど)

Bayes Filter の数学的導出

$p(\boldsymbol{x}_t \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{m})$ ← 時刻tの自己位置x_tの事後確率

↓ ベイズの定理

$= \eta\, p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})\; p(\boldsymbol{x}_t \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})$

条件付き独立性（過去には非依存）　全確率の定理

$= \eta\, p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m}) \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})\; p(\boldsymbol{x}_{t-1} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m}) d\boldsymbol{x}_{t-1}$

条件付き独立性（1次マルコフ性）　条件付き独立性

$= \eta\, p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m}) \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)\; p(\boldsymbol{x}_{t-1} \mid \boldsymbol{u}_{1:t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m}) d\boldsymbol{x}_{t-1}$

尤度（計測モデル）　動作モデル　事前確率　時刻t-1の自己位置x_{t-1}

漸化式として定式化

ベイズの定理の適用

ベイズの定理

$$p(A \mid B) = \frac{p(B \mid A)\, p(A)}{p(B)}$$

条件付きベイズの定理

$$p(A \mid B, C) = \frac{p(B \mid A, C)\, p(A \mid C)}{p(B \mid C)}$$

自己位置推定に適用

$$p(x_t \mid u_{1:t}, z_{1:t}, m)$$

$$= p(x_t \mid z_t, u_{1:t}, z_{1:t-1}, m)$$

$$= \frac{p(z_t \mid x_t, u_{1:t}, z_{1:t-1}, m)\; p(x_t \mid u_{1:t}, z_{1:t-1}, m)}{p(z_t \mid u_{1:t}, z_{1:t-1}, m)}$$

$$= \eta\; p(z_t \mid x_t, u_{1:t}, z_{1:t-1}, m)\; p(x_t \mid u_{1:t}, z_{1:t-1}, m)$$

x：ﾛﾎﾞｯﾄ位置
m：地図
u：制御動作（ｵﾄﾞﾒﾄﾘ）
z：環境計測（ﾚｰｻﾞｽｷｬﾝ）

ﾏﾙｺﾌ性の仮定と条件付き独立性の適用

1次マルコフ性を仮定

- 現在の状態は、1時刻前の状態にだけ依存
- それ以前の状態や他のデータとは独立

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m}) = p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m})$$

現在の環境計測z_tは、現在の状態である自己位置x_tと地図mにのみ依存

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m}) = p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)$$

現在の状態である自己位置x_tは、1時刻前の状態x_{t-1}とそれ以後の制御動作u_tにのみ依存

全確率の定理の適用

全確率の定理

$$p(\boldsymbol{A}) = \int p(\boldsymbol{A} \mid \boldsymbol{B})\, p(\boldsymbol{B}) d\boldsymbol{B}$$

条件付き全確率の定理

$$p(\boldsymbol{A} \mid \boldsymbol{C}) = \int p(\boldsymbol{A} \mid \boldsymbol{B}, \boldsymbol{C})\, p(\boldsymbol{B} \mid \boldsymbol{C}) d\boldsymbol{B}$$

自己位置推定に適用

$$p(\boldsymbol{x}_t \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})$$

隠れ変数x_{t-1}を導入

$$= \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})\, p(\boldsymbol{x}_{t-1} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m}) d\boldsymbol{x}_{t-1}$$

x：ﾛﾎﾞｯﾄ位置
m：地図
u：制御動作（ｵﾄﾞﾒﾄﾘ）
z：環境計測（ﾚｰｻﾞｽｷｬﾝ）

Contents

1. 自律走行を実現するための ROS ﾊﾟｯｹｰｼﾞ
2. 自己位置推定と SLAM の基本
3. 各ﾊﾟｯｹｰｼﾞのｱﾙｺﾞﾘｽﾞﾑ
   a. 自己位置推定（amcl）
   b. SLAM、地図生成（gmapping）
   c. 大域的経路計画、局所的動作計画（move_base）
4. ROS に関する情報の調べ方
5. ROS での開発に関する知見
6. まとめ

Particle Filter の定式化

- Bayes Filter の一種だが、定式化は若干異なる
- 各ﾊﾟｰﾃｨｸﾙは、時刻tの自己位置だけでなく、時刻1から時刻tの走行軌跡を保持する
- 動作ﾓﾃﾞﾙによる事前確率の計算に積分なし

$$p(\boldsymbol{x}_{1:t} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{m})$$ ← 時刻1からtの走行軌跡$x_{t:t}$の事後確率

$$= \eta\, p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m})$$ ← 尤度(計測モデル)

$$\cdot\, p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)\; p(\boldsymbol{x}_{1:t-1} \mid \boldsymbol{u}_{1:t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})$$

動作モデル　　事前確率　　時刻1からt-1の走行軌跡$x_{1:t-1}$

Particle Filter の数学的導出

$p(\boldsymbol{x}_{1:t} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{m})$ ← 時刻1からtの走行軌跡$x_{t:t}$の事後確率

↓ ベイズの定理

$= \eta\, p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})\, p(\boldsymbol{x}_{1:t} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})$

↓ 条件付き独立性（過去には非依存）　↓ 乗法定理

$= \eta\, p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m})\, p(\boldsymbol{x}_t \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})\, p(\boldsymbol{x}_{1:t-1} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})$

↓ 条件付き独立性（1次マルコフ性）　↓ 条件付き独立性

$= \eta\, p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m})\, p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)\, p(\boldsymbol{x}_{1:t-1} \mid \boldsymbol{u}_{1:t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})$

尤度（計測モデル）　動作モデル　事前確率

時刻1からt-1の走行軌跡$x_{1:t-1}$

漸化式として定式化

乗法定理の適用（Particle Filter の場合）

- 乗法定理（条件付き確率）

$$p(A, B) = p(A \mid B)\, p(B)$$

- 条件付き乗法定理（3変数での条件付き確率）

$$p(A, B \mid C) = p(A \mid B, C)\, p(B \mid C)$$

- 自己位置推定に適用

$$p(\boldsymbol{x}_{1:t} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})$$

$$= p(\boldsymbol{x}_t \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})\; p(\boldsymbol{x}_{1:t-1} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{m})$$

x：ﾛﾎﾞｯﾄ位置
m：地図
u：制御動作（ｵﾄﾞﾒﾄﾘ）
z：環境計測（ﾚｰｻﾞｽｷｬﾝ）

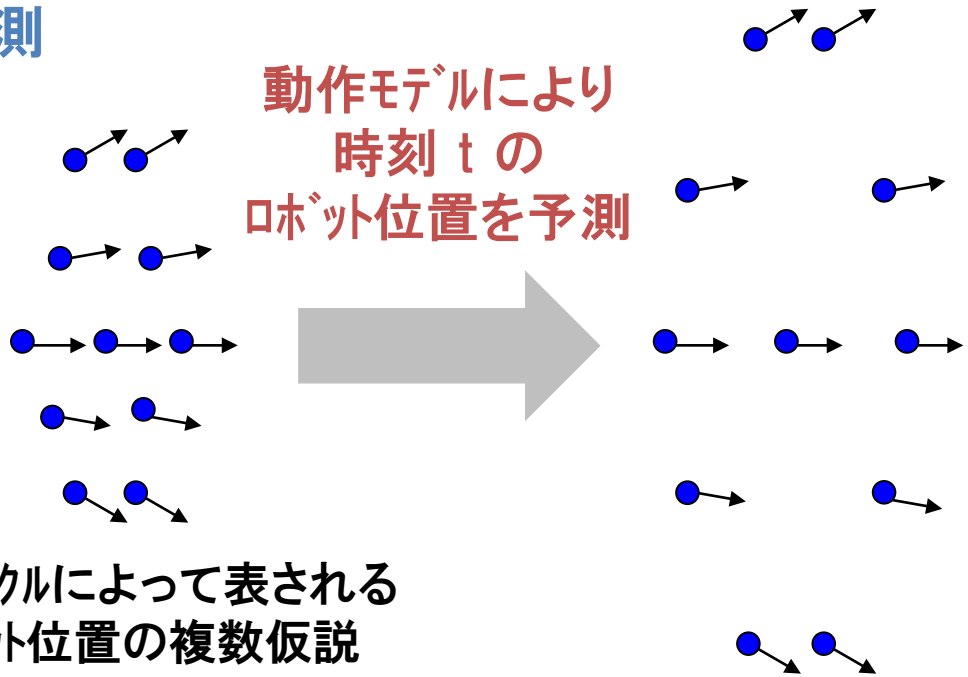
Adaptive Monte Carlo Localization

- **Particle Filter による2次元3自由度での確率的自己位置推定（Monte Carlo Localization）**
  - ｵﾄﾞﾒﾄﾘ動作ﾓﾃﾞﾙ
  - ﾋﾞｰﾑ計測ﾓﾃﾞﾙ／尤度場計測ﾓﾃﾞﾙ
- **KLD-Sampling による Adaptive MCL**
  推定位置の不確かさに応じて、ﾊﾟｰﾃｨｸﾙ数を適応的に調整
- **Augmented MCL**
  真値周辺のﾊﾟｰﾃｨｸﾙの喪失に対処するために、ﾗﾝﾀﾞﾑﾊﾟｰﾃｨｸﾙを挿入

Adaptive Monte Carlo Localization

処理ｲﾒｰｼﾞ（平面図）

1. 予測



ﾊﾟｰﾃｨｸﾙによって表される
ﾛﾎﾞｯﾄ位置の複数仮説

時刻：t-1

時刻：t

：ﾛﾎﾞｯﾄ位置

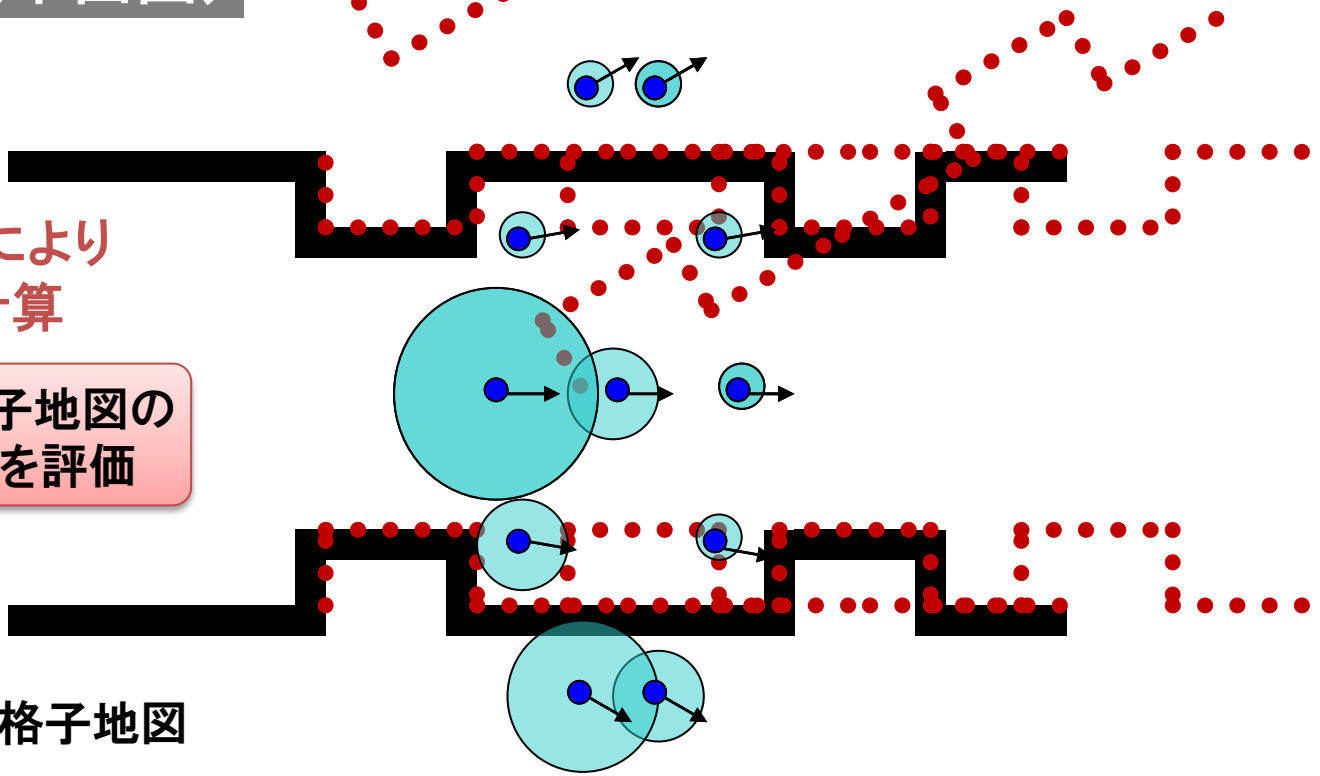
Adaptive Monte Carlo Localization

処理ｲﾒｰｼﾞ（平面図）

2. 尤度計算

計測ﾓﾃﾞﾙにより
尤度を計算

ｽｷｬﾝと占有格子地図の
重なり度合いを評価



：占有格子地図

：現時刻のｽｷｬﾝ計測

：尤度

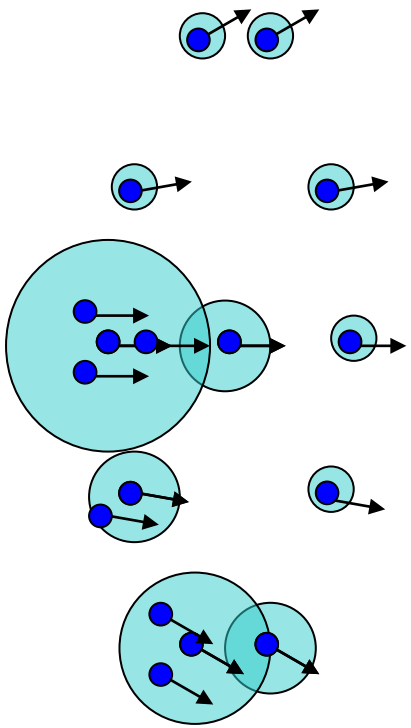
事前に与えられた1つの地図に対して、各ﾊﾟｰﾃｨｸﾙのｽｷｬﾝを評価

Adaptive Monte Carlo Localization

処理ｲﾒｰｼﾞ（平面図）

3. ﾘｻﾝﾌﾟﾘﾝｸﾞ

- 尤度の高いﾊﾟｰﾃｨｸﾙは自身のｺﾋﾟｰを多く残す
- 尤度の低いﾊﾟｰﾃｨｸﾙは消滅
- KLD-Sampling により、推定位置の不確かさに応じてﾊﾟｰﾃｨｸﾙ数を調整



：尤度

Contents

1. 自律走行を実現するための ROS ﾊﾟｯｹｰｼﾞ
2. 自己位置推定と SLAM の基本
3. 各ﾊﾟｯｹｰｼﾞのｱﾙｺﾞﾘｽﾞﾑ
   a. 自己位置推定（amcl）
   b. SLAM、地図生成（gmapping）
   c. 大域的経路計画、局所的動作計画（move_base）
4. ROS に関する情報の調べ方
5. ROS での開発に関する知見
6. まとめ

Rao-Blackwellized Particle Filter の定式化

ﾛﾎﾞｯﾄ走行軌跡と地図を求める完全 SLAM、独立性の仮定により因子分解して解く

$$p(\boldsymbol{x}_{1:t}, \boldsymbol{m} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t})$$

← ﾛﾎﾞｯﾄ走行軌跡と地図の同時確率

$$= p(\boldsymbol{x}_{1:t} \mid \boldsymbol{u}_{1:t}, \boldsymbol{z}_{1:t}) \prod_{n=1}^{N} p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t})$$

ﾛﾎﾞｯﾄ走行軌跡 $x_{1:t}$ を Particle Filter で推定

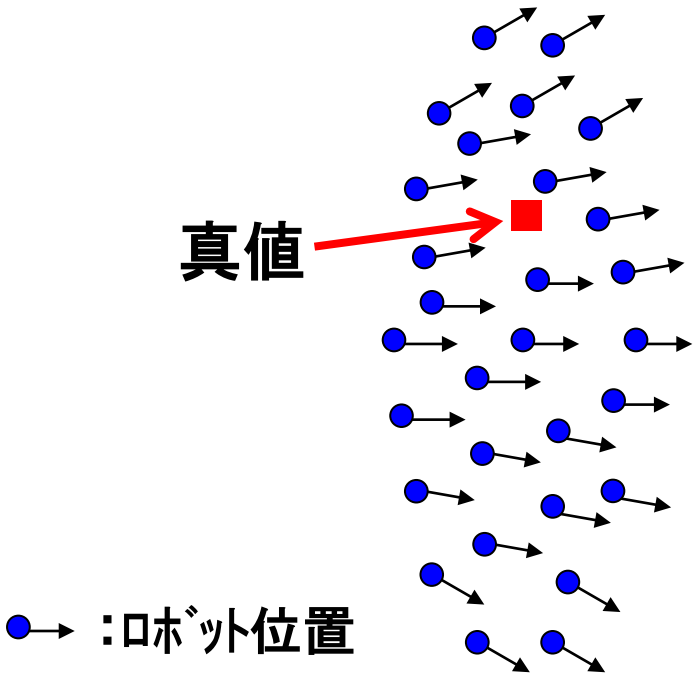
地図 m を Binary Bayes Filter で推定

x：ﾛﾎﾞｯﾄ位置
u：制御動作（ｵﾄﾞﾒﾄﾘなど）
m：地図
z：環境計測（ﾚｰｻﾞｽｷｬﾝなど）

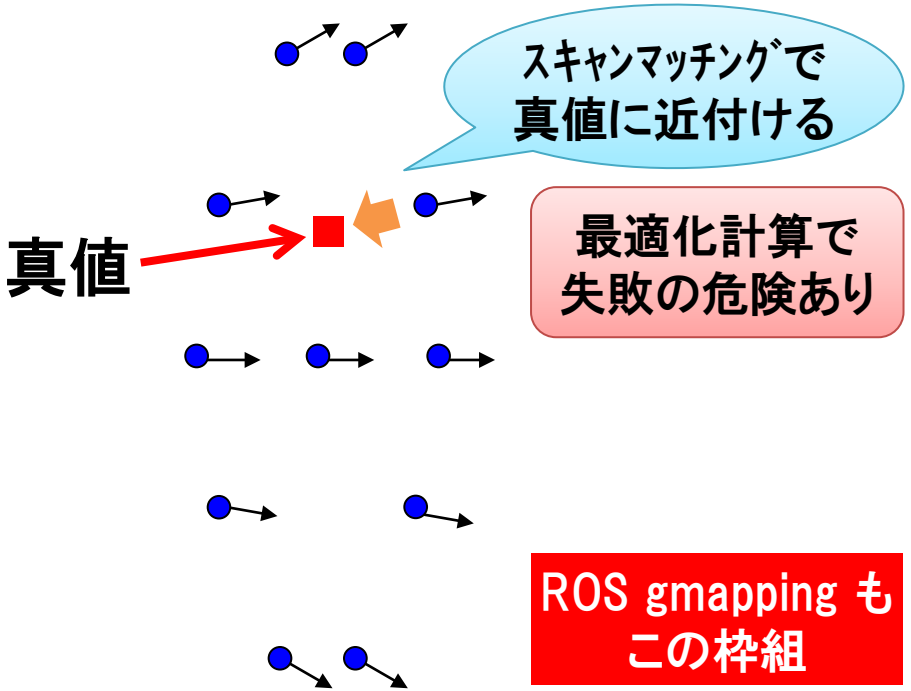
提案分布における最適化計算の利用

計算ｺｽﾄはﾊﾟｰﾃｨｸﾙ数に依存するので、少数のﾊﾟｰﾃｨｸﾙそれぞれで最適化計算を併用

多数のﾊﾟｰﾃｨｸﾙによる提案分布（高精度だが計算ｺｽﾄ大）



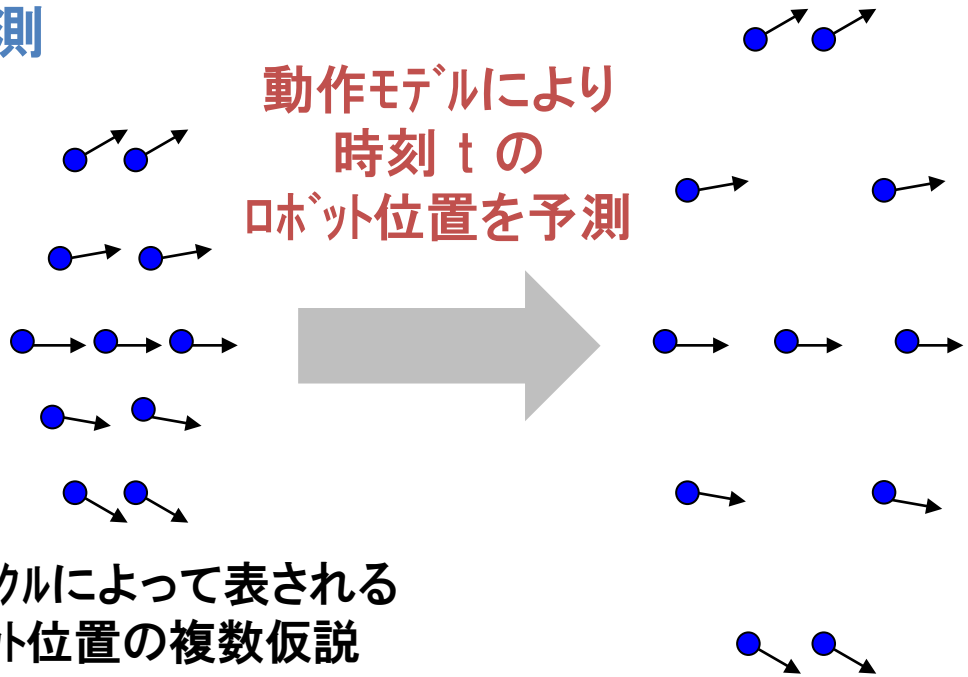
少数のﾊﾟｰﾃｨｸﾙと最適化計算による提案分布（FastSLAM 2.0）



Rao-Blackwellized Particle Filter SLAM

処理ｲﾒｰｼﾞ（平面図）

1. 予測



ﾊﾟｰﾃｨｸﾙによって表される
ﾛﾎﾞｯﾄ位置の複数仮説

時刻：t-1

時刻：t

：ﾛﾎﾞｯﾄ位置

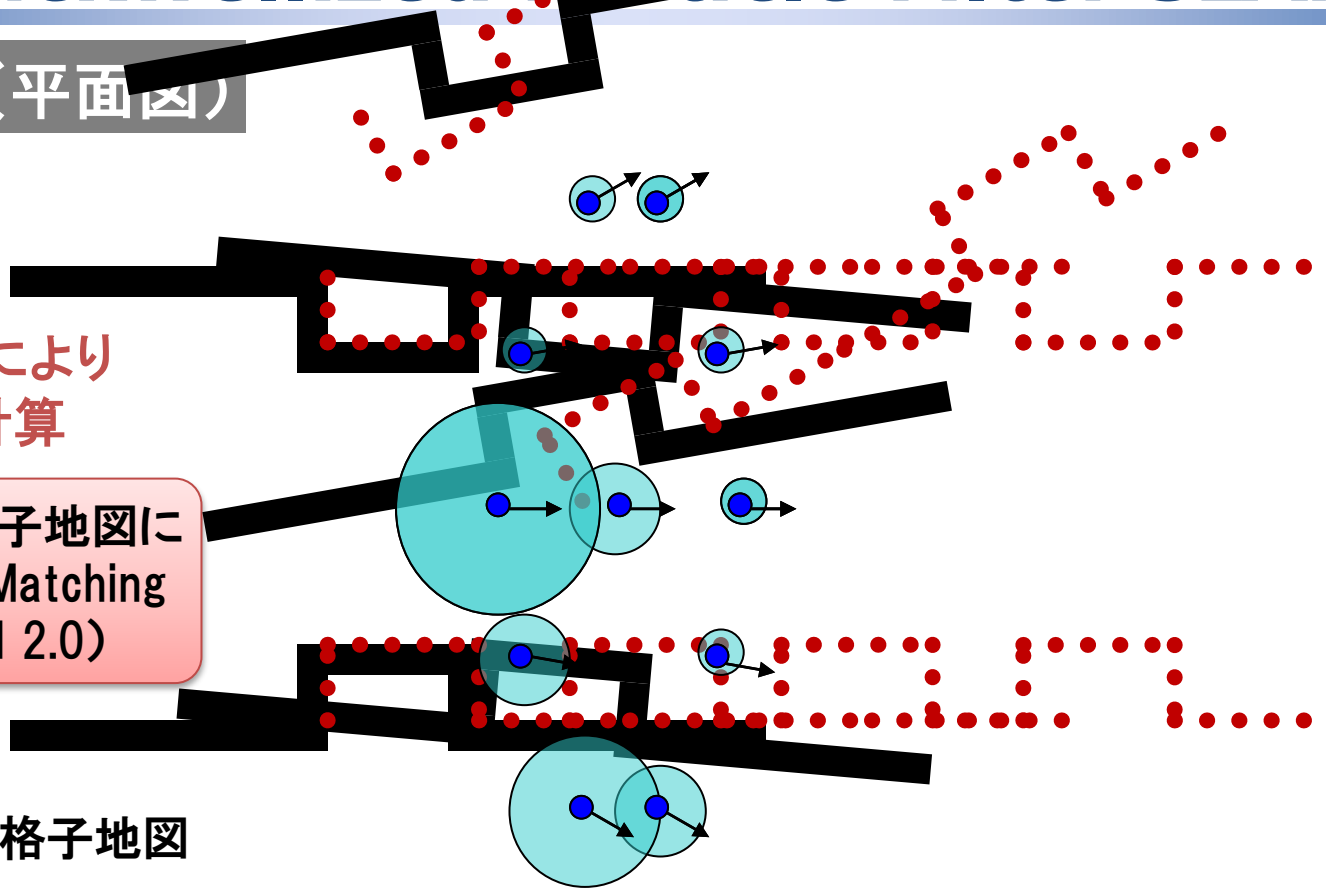
Rao-Blackwellized Particle Filter SLAM

処理ｲﾒｰｼﾞ(平面図)

2. 尤度計算

計測ﾓﾃﾞﾙにより
尤度を計算

ｽｷｬﾝを占有格子地図に
Greedy Scan Matching
(FastSLAM 2.0)



：占有格子地図

：現時刻のｽｷｬﾝ計測

：尤度

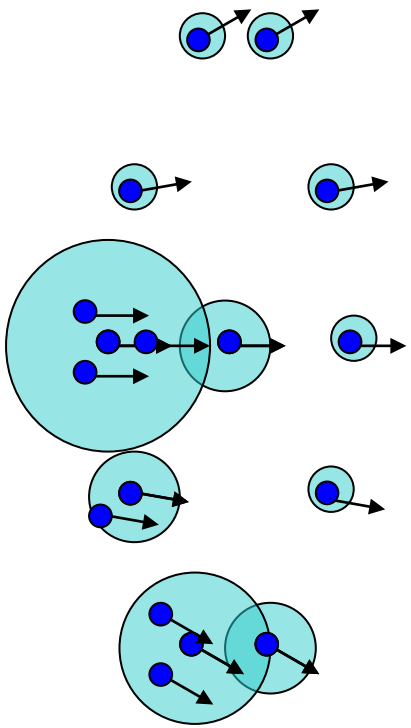
Rao-Blackwellization により各ﾊﾟｰﾃｨｸﾙごとに地図を持っている

Rao-Blackwellized Particle Filter SLAM

処理ｲﾒｰｼﾞ（平面図）

3. ﾘｻﾝﾌﾟﾘﾝｸﾞ

- 尤度の高いﾊﾟｰﾃｨｸﾙは自身のｺﾋﾟｰを多く残す
- 尤度の低いﾊﾟｰﾃｨｸﾙは消滅

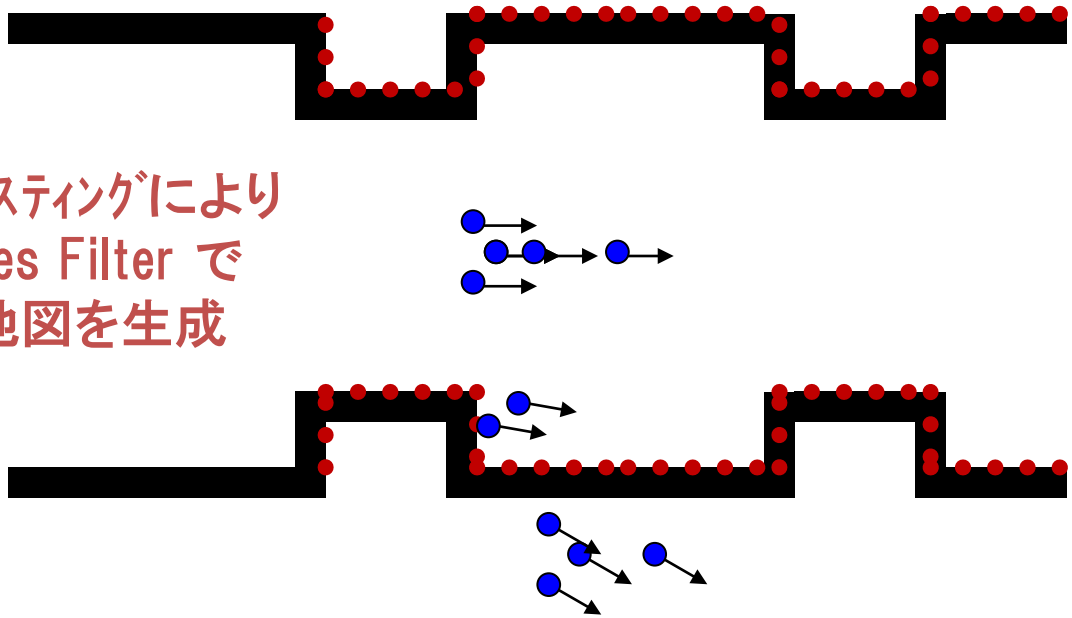


：尤度

Rao-Blackwellized Particle Filter SLAM

処理ｲﾒｰｼﾞ（平面図）

4. 地図生成



ﾚｰｻﾞのﾚｲｷｬｽﾃｨﾝｸﾞにより
Binary Bayes Filter で
占有格子地図を生成

：占有格子地図

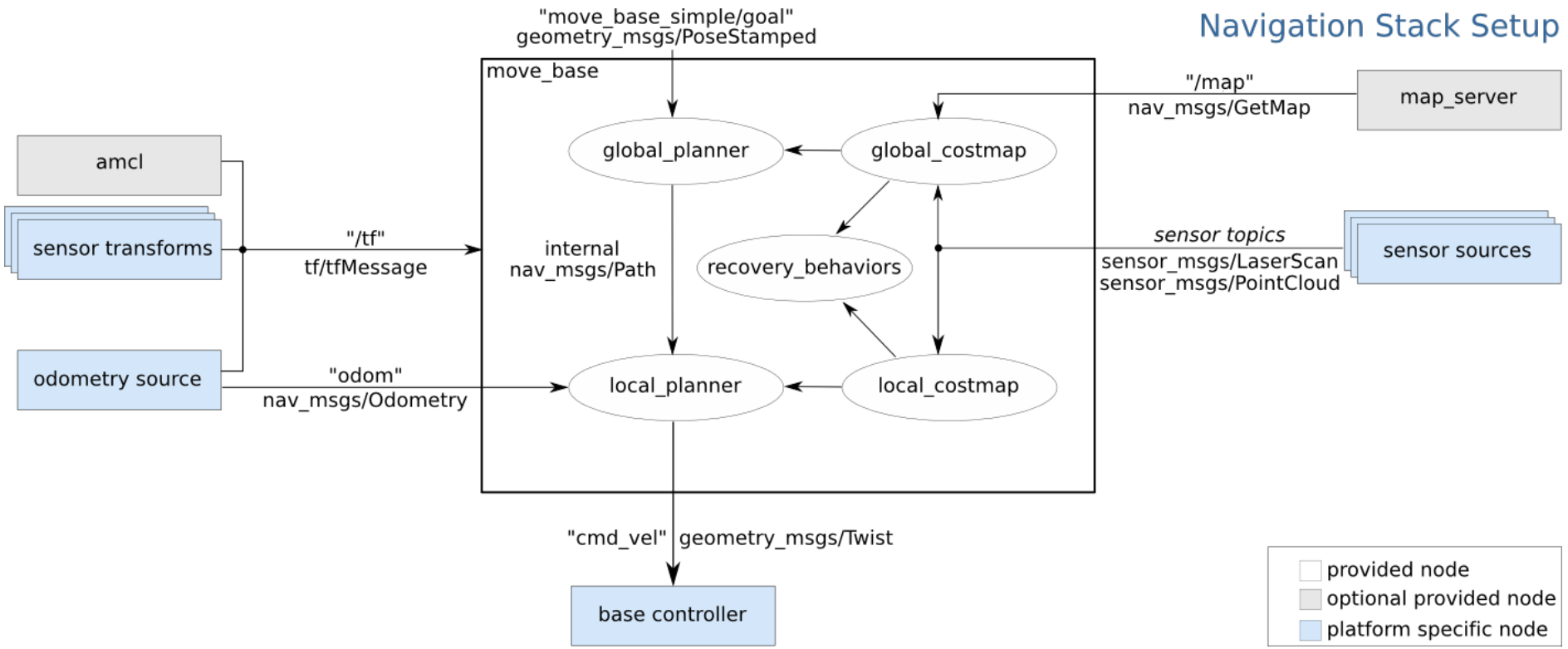
：現時刻のｽｷｬﾝ計測

Rao-Blackwellization により各ﾊﾟｰﾃｨｸﾙごとにそれぞれの地図を生成する

Contents

1. 自律走行を実現するための ROS ﾊﾟｯｹｰｼﾞ
2. 自己位置推定と SLAM の基本
3. 各ﾊﾟｯｹｰｼﾞのｱﾙｺﾞﾘｽﾞﾑ
   a. 自己位置推定（amcl）
   b. SLAM、地図生成（gmapping）
   c. 大域的経路計画、局所的動作計画（move_base）
4. ROS に関する情報の調べ方
5. ROS での開発に関する知見
6. まとめ

自律走行時のﾃﾞｰﾀの流れ

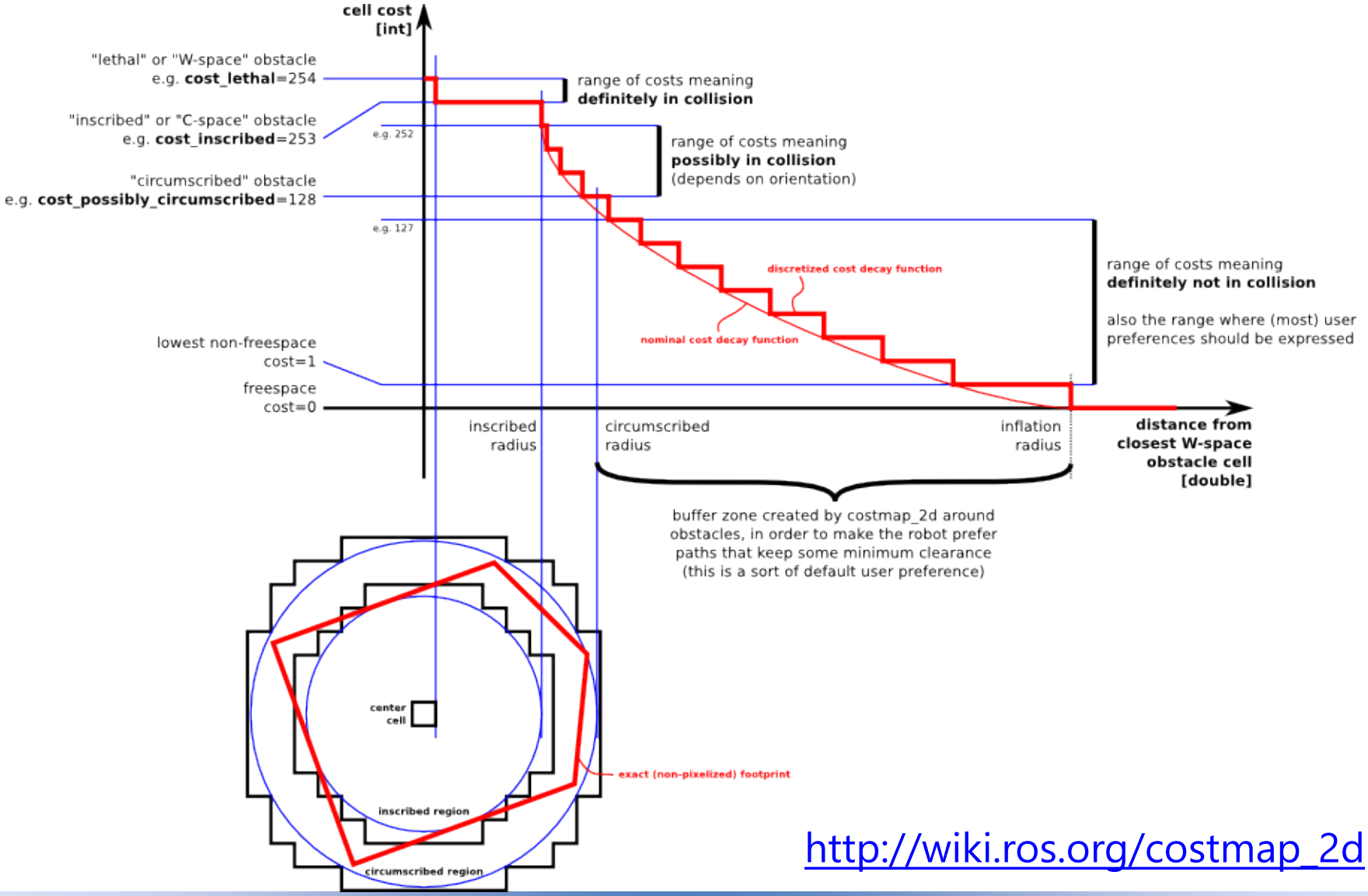


http://wiki.ros.org/move_base

costmap_2d によるｺｽﾄﾏｯﾌﾟの生成

- ﾚｰｻﾞｽｷｬﾝまたは3次元点群のﾃﾞｰﾀを用いて、2次元／3次元の占有格子地図を生成
- ﾚｲｷｬｽﾃｨﾝｸﾞにより、占有／ﾌﾘｰ／未知の3状態で障害物情報を管理
- 占有格子地図の障害物をﾛﾎﾞｯﾄの内接円半径の大きさで膨張させ、x, y の2自由度ｺﾝﾌｨｸﾞﾚｰｼｮﾝ空間として2次元ｺｽﾄﾏｯﾌﾟを生成
- ｸﾞﾛｰﾊﾞﾙｺｽﾄﾏｯﾌﾟとﾛｰｶﾙｺｽﾄﾏｯﾌﾟの2種類を生成

costmap_2d によるｺｽﾄﾏｯﾌﾟの生成



http://wiki.ros.org/costmap_2d

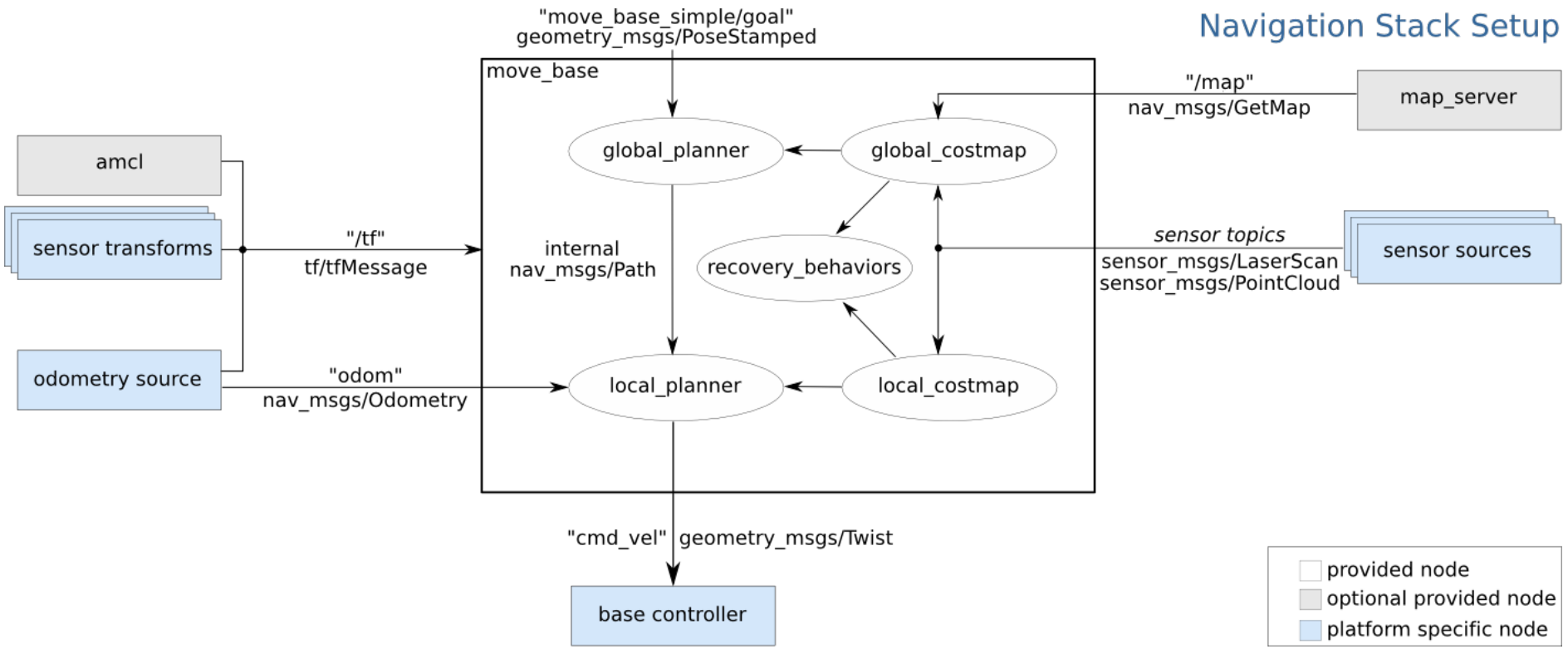
ｺｽﾄﾏｯﾌﾟを生成する枠組み

- ROS Hydro 以降、costmap_2d の構成が変更
  http://wiki.ros.org/costmap_2d
  http://wiki.ros.org/costmap_2d/hydro
  http://wiki.ros.org/hydro/Migration#Navigation_-_Costmap2D

- LayeredCostmap という仕組みが導入された
  - Static Map Layer：SLAM で生成した静的地図
  - Obstacle Map Layer：検出した障害物の蓄積と消去
    （ObstacleCostmapPlugin / VoxelCostmapPlugin）
  - Inflation Layer：障害物を膨張したｺﾝﾌｨｸﾞﾚｰｼｮﾝ空間

- ROS の pluginlib という機能で実現されている

自律走行時のﾃﾞｰﾀの流れ



http://wiki.ros.org/move_base

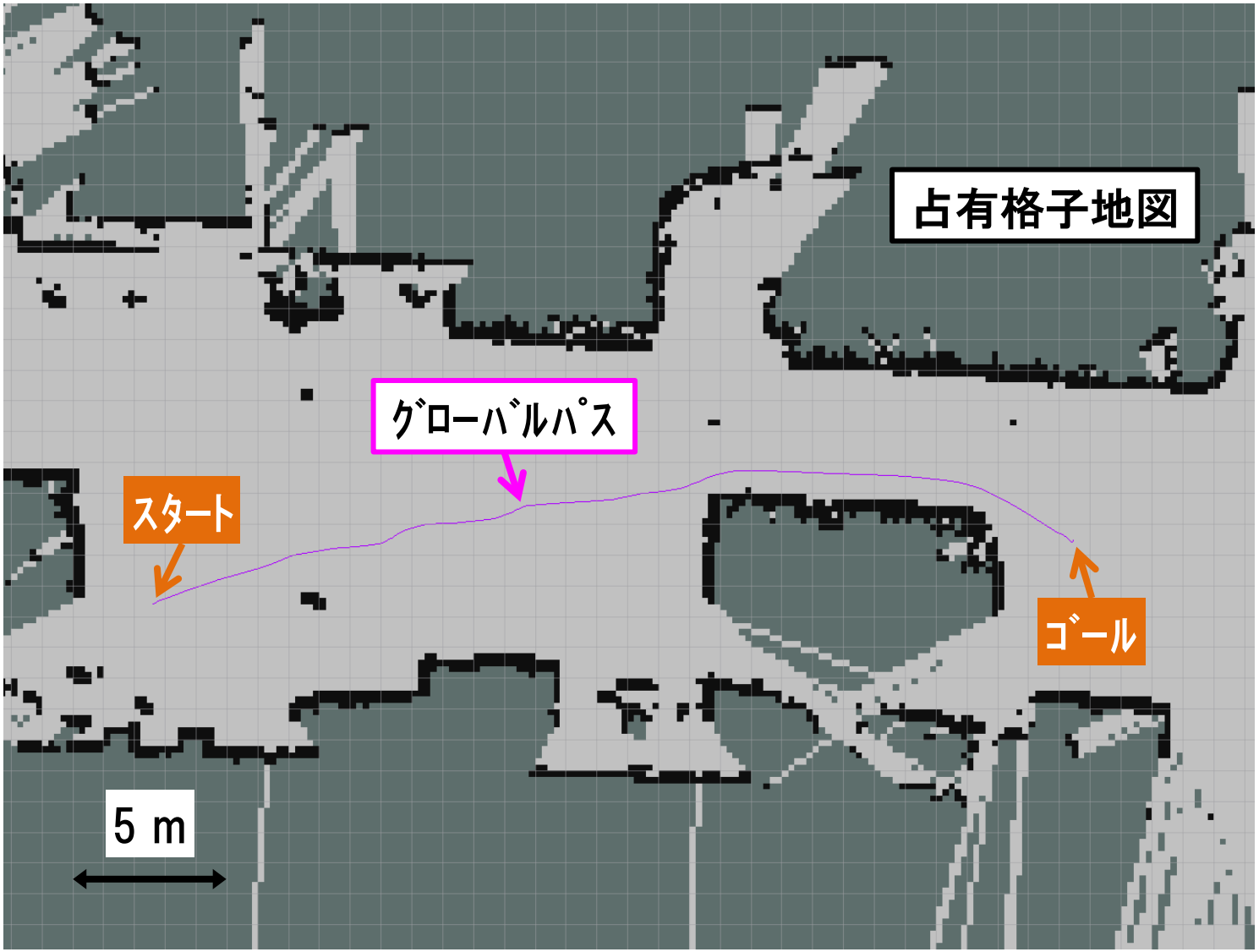
navfn の大域的経路計画

- **Navigation Function** と呼ばれるﾎﾟﾃﾝｼｬﾙ場で評価
  - ｺﾞｰﾙまでの距離と障害物からの距離をｺｽﾄとする
  - ｸﾞﾛｰﾊﾞﾙｺｽﾄﾏｯﾌﾟに基づいて処理

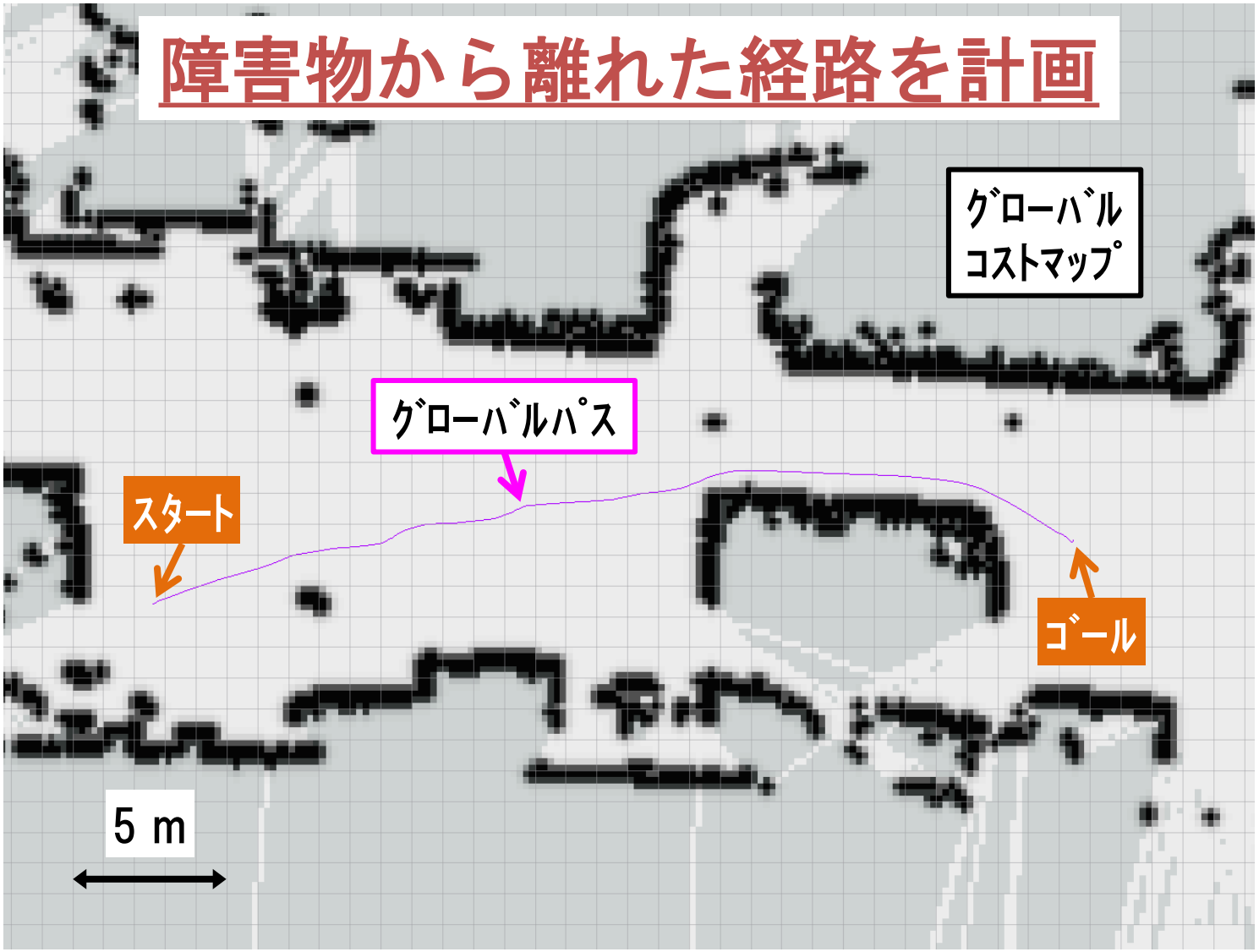
- 格子地図に対して、**ﾀﾞｲｸｽﾄﾗ法**でﾎﾟﾃﾝｼｬﾙ場の最小ｺｽﾄとなる経路を探索
  - 各ｾﾙのﾎﾟﾃﾝｼｬﾙ場のｺｽﾄを計算しながらﾀﾞｲｸｽﾄﾗ法で探索し、ｺﾞｰﾙまでのｸﾞﾛｰﾊﾞﾙﾊﾟｽを計画

- ﾛﾎﾞｯﾄの方向は考慮しない（円形と仮定）
  - 方向によっては障害物と衝突する経路が計画される

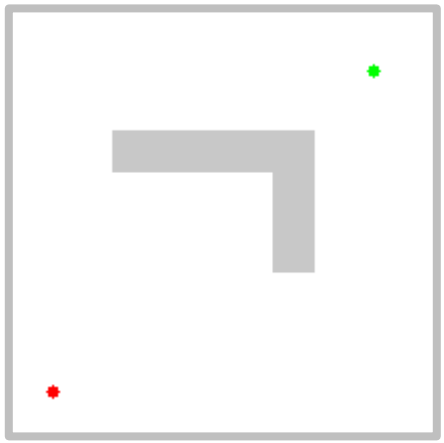
navfn による大域的経路計画の例



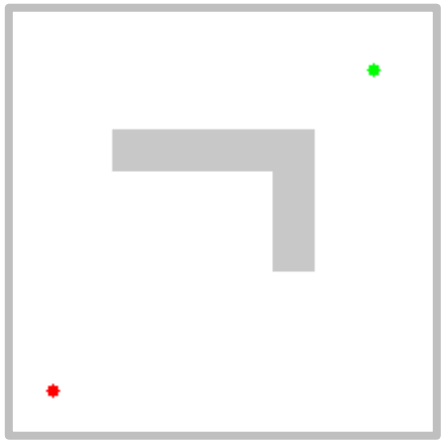
navfn による大域的経路計画の例



ﾀﾞｲｸｽﾄﾗ法とA*の違い



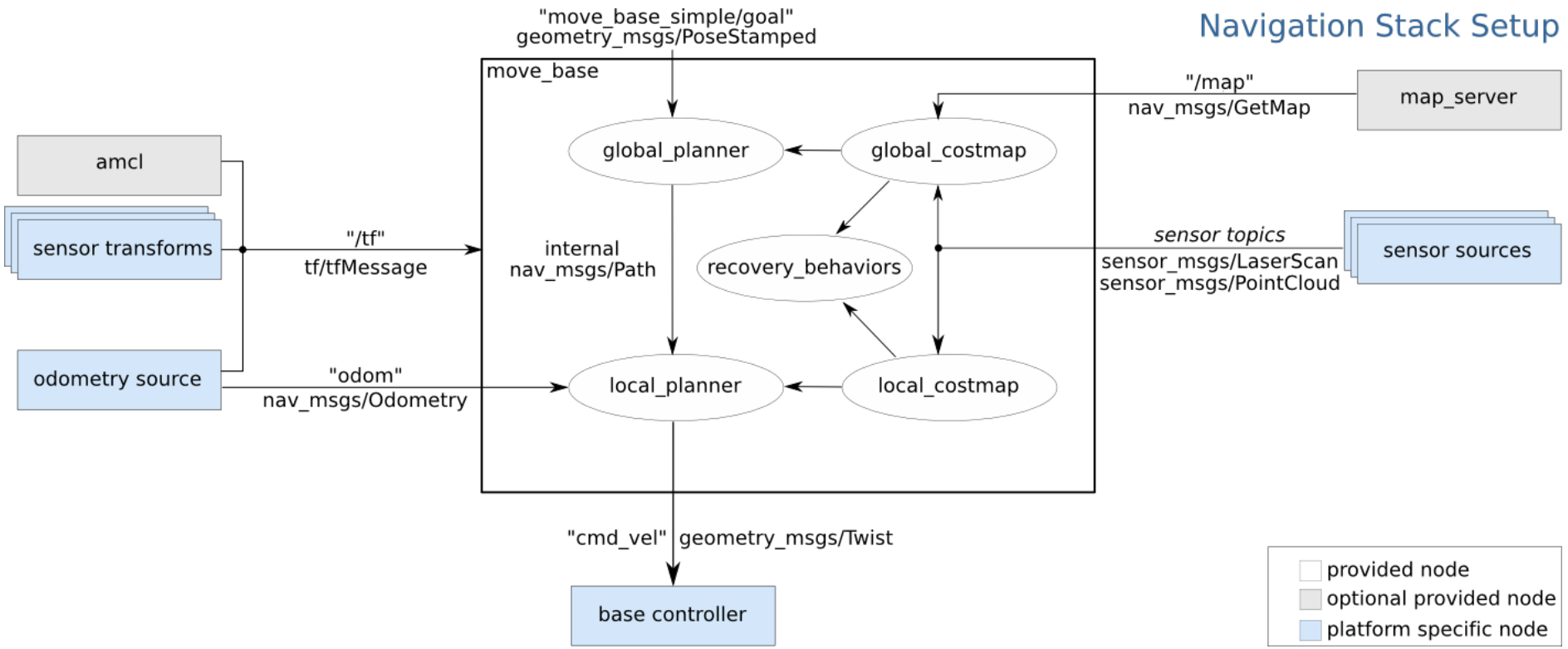
- ﾀﾞｲｸｽﾄﾗ法
  幅優先探索をｺｽﾄを評価するように拡張した、最良優先探索と呼ばれる手法の一種



- A*（A-Star）
  ﾋｭｰﾘｽﾃｨｯｸ関数でｺｽﾄの予測値を追加した、ﾀﾞｲｸｽﾄﾗ法の改良版

https://en.wikipedia.org/wiki/Dijkstra's_algorithm
https://en.wikipedia.org/wiki/A*_search_algorithm

自律走行時のﾃﾞｰﾀの流れ



http://wiki.ros.org/move_base

base_local_planner の局所的動作計画

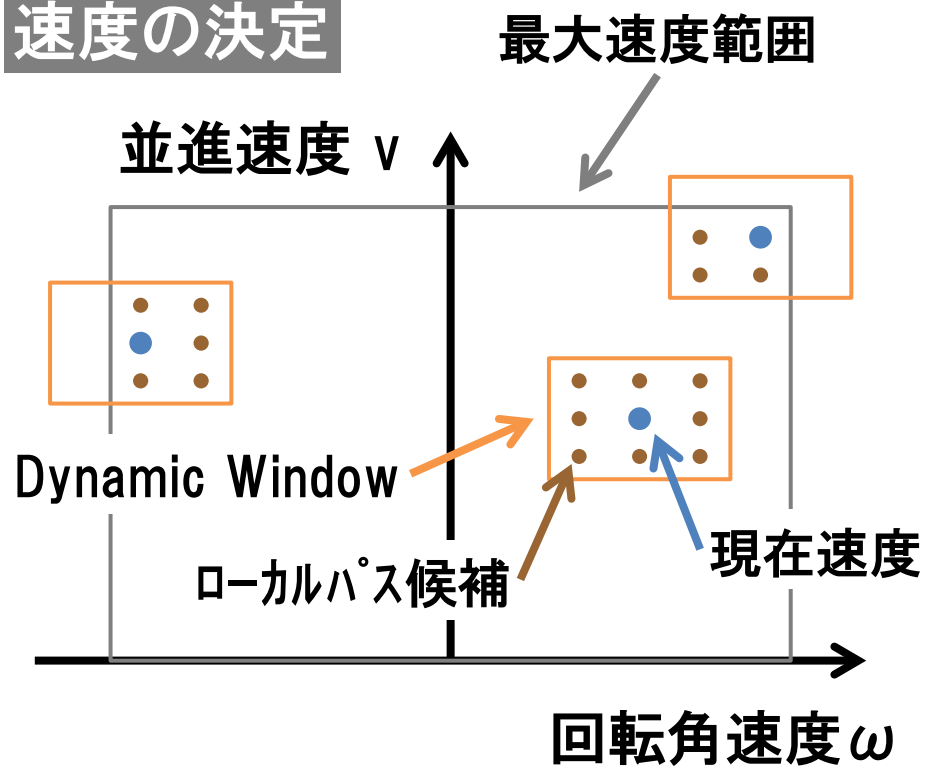
- 大域的経路計画によるｸﾞﾛｰﾊﾞﾙﾊﾟｽに追従しつつ、障害物を回避する動作（速度）を計画

- **Dynamic Window Approach** を使用
  - ﾛﾎﾞｯﾄのｷﾈﾏﾃｨｸｽ（運動ﾓﾃﾞﾙ）を考慮した軌跡
  - 現在の速度に基づき、ﾀﾞｲﾅﾐｸｽを考慮して実行可能な複数のﾛｰｶﾙﾊﾟｽ候補を生成
  - 評価関数により、ｸﾞﾛｰﾊﾞﾙﾊﾟｽに近く、かつ障害物から離れたﾛｰｶﾙﾊﾟｽの動作を選択
  - ﾛｰｶﾙｺｽﾄﾏｯﾌﾟに基づいて処理

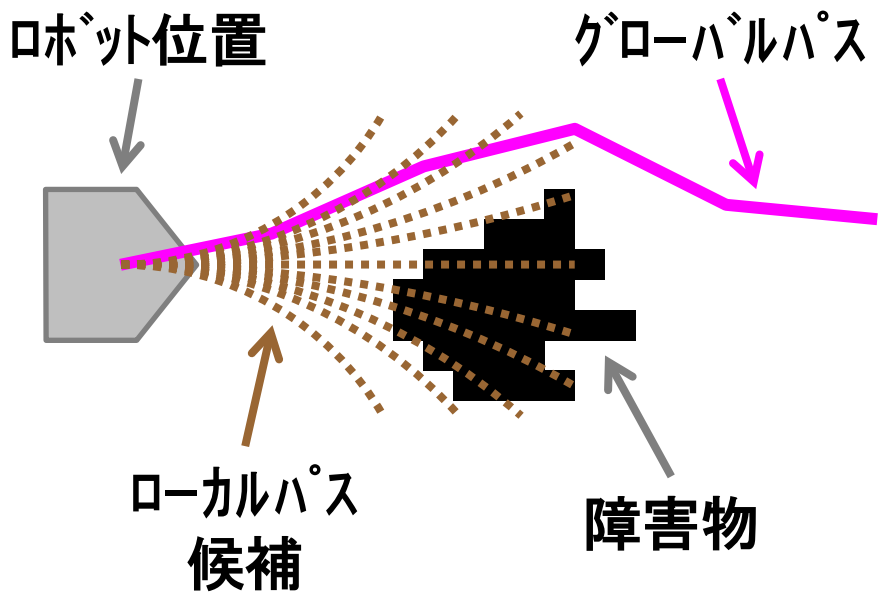
Dynamic Window Approach

キネマティクスとダイナミクスを考慮してローカルパス候補の軌跡を生成、離散化してコストを評価

速度の決定



平面図



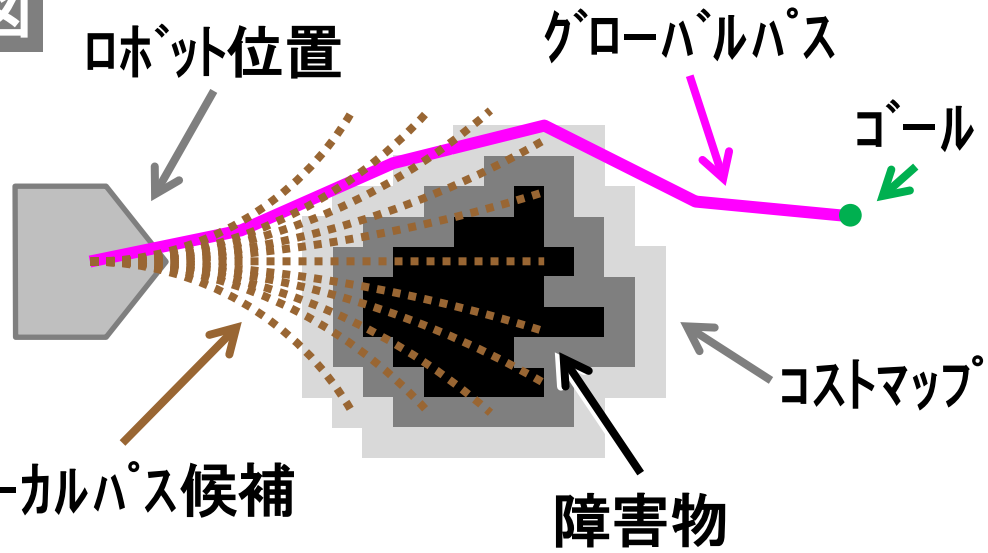
Dynamic Window Approach のｺｽﾄ評価関数

軌跡ｺｽﾄ ＝

経路距離ｽｹｰﾙ × ﾛｰｶﾙﾊﾟｽ端点とｸﾞﾛｰﾊﾞﾙﾊﾟｽの距離
＋ ｺﾞｰﾙ距離ｽｹｰﾙ × ﾛｰｶﾙﾊﾟｽ端点とｺﾞｰﾙの距離
＋ 障害物ｽｹｰﾙ × ﾛｰｶﾙﾊﾟｽ上での最大障害物ｺｽﾄ

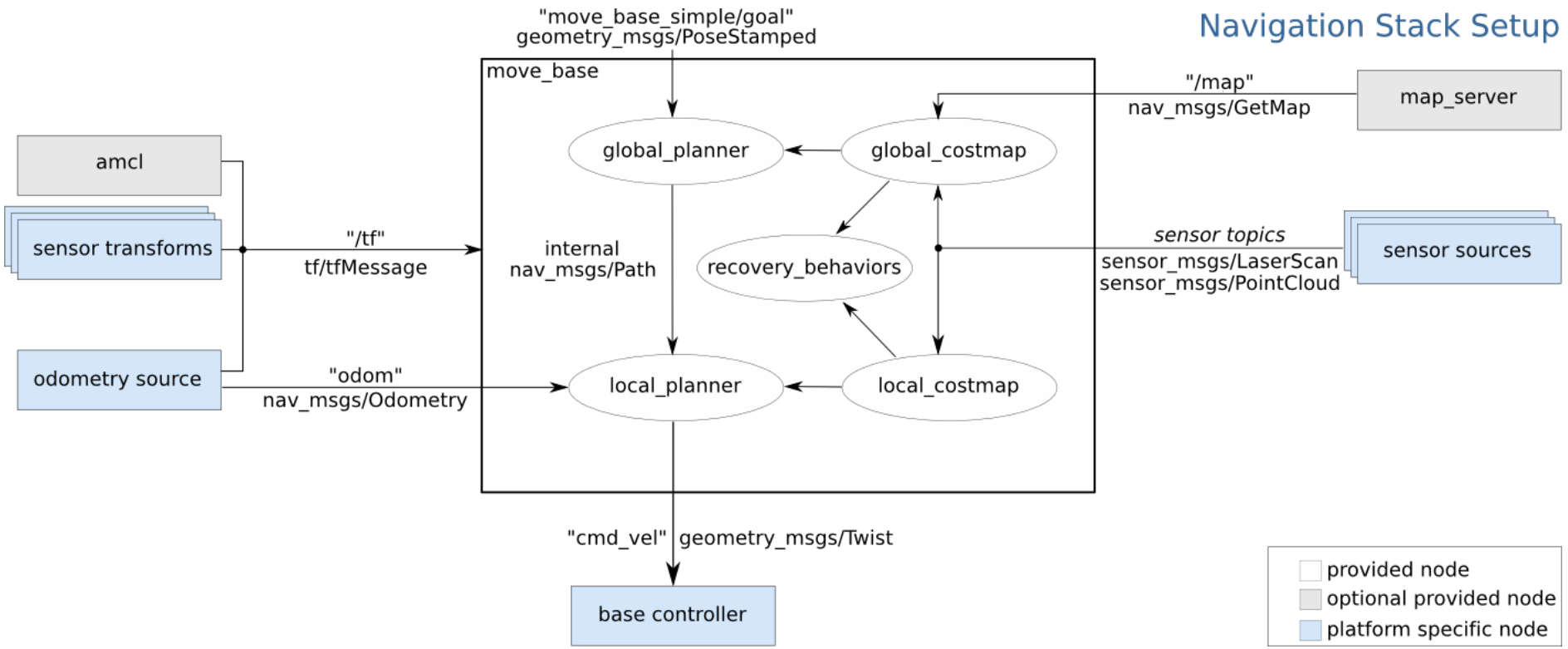
上記の ROS の実装は、ｵﾘｼﾞﾅﾙ論文の評価関数とは異なる

平面図



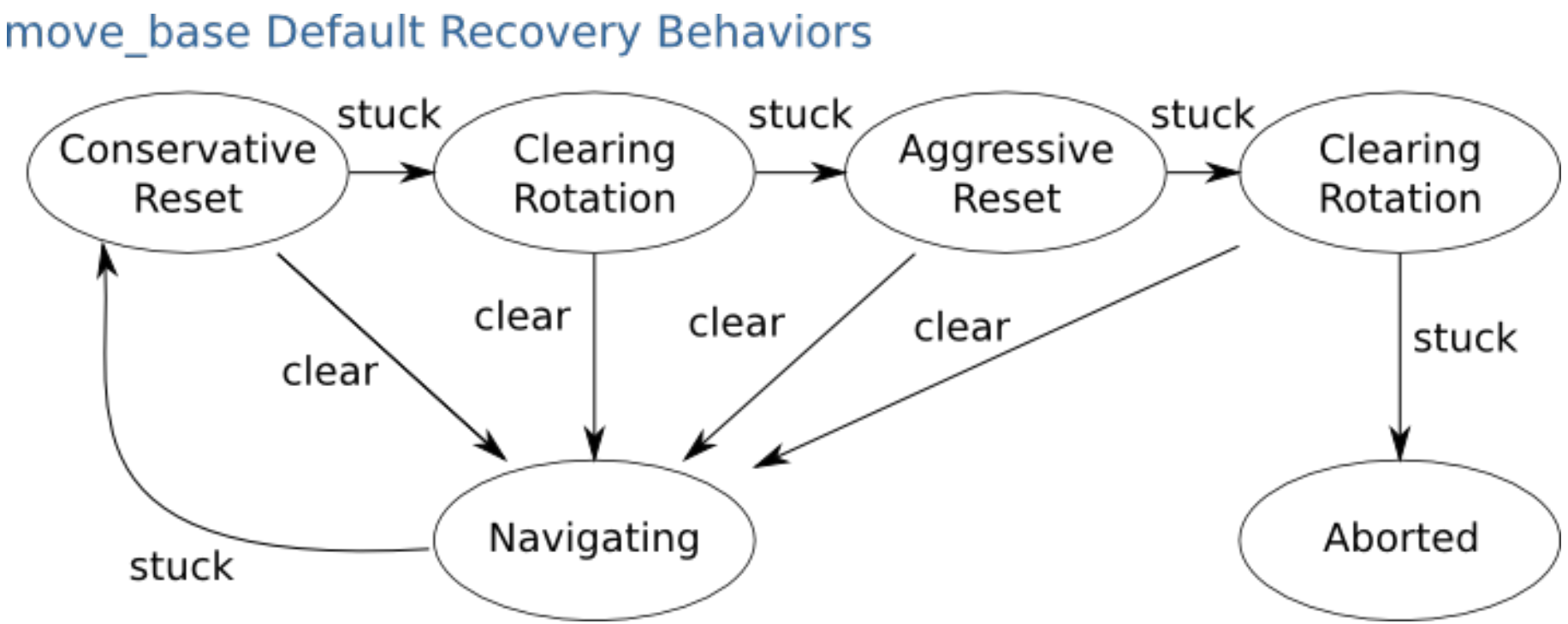
- 距離は各ｾﾙごとに事前計算
- ﾛｰｶﾙﾊﾟｽの振動を抑制するため、一定距離は逆符号の速度を無効化

自律走行時のﾃﾞｰﾀの流れ



http://wiki.ros.org/move_base

ｽﾀｯｸした場合の復帰動作の状態遷移



http://wiki.ros.org/move_base

Contents

1. 自律走行を実現するための ROS ﾊﾟｯｹｰｼﾞ
2. 自己位置推定と SLAM の基本
3. 各ﾊﾟｯｹｰｼﾞのｱﾙｺﾞﾘｽﾞﾑ
   a. 自己位置推定（amcl）
   b. SLAM、地図生成（gmapping）
   c. 大域的経路計画、局所的動作計画（move_base）
4. **ROS に関する情報の調べ方**
5. ROS での開発に関する知見
6. まとめ

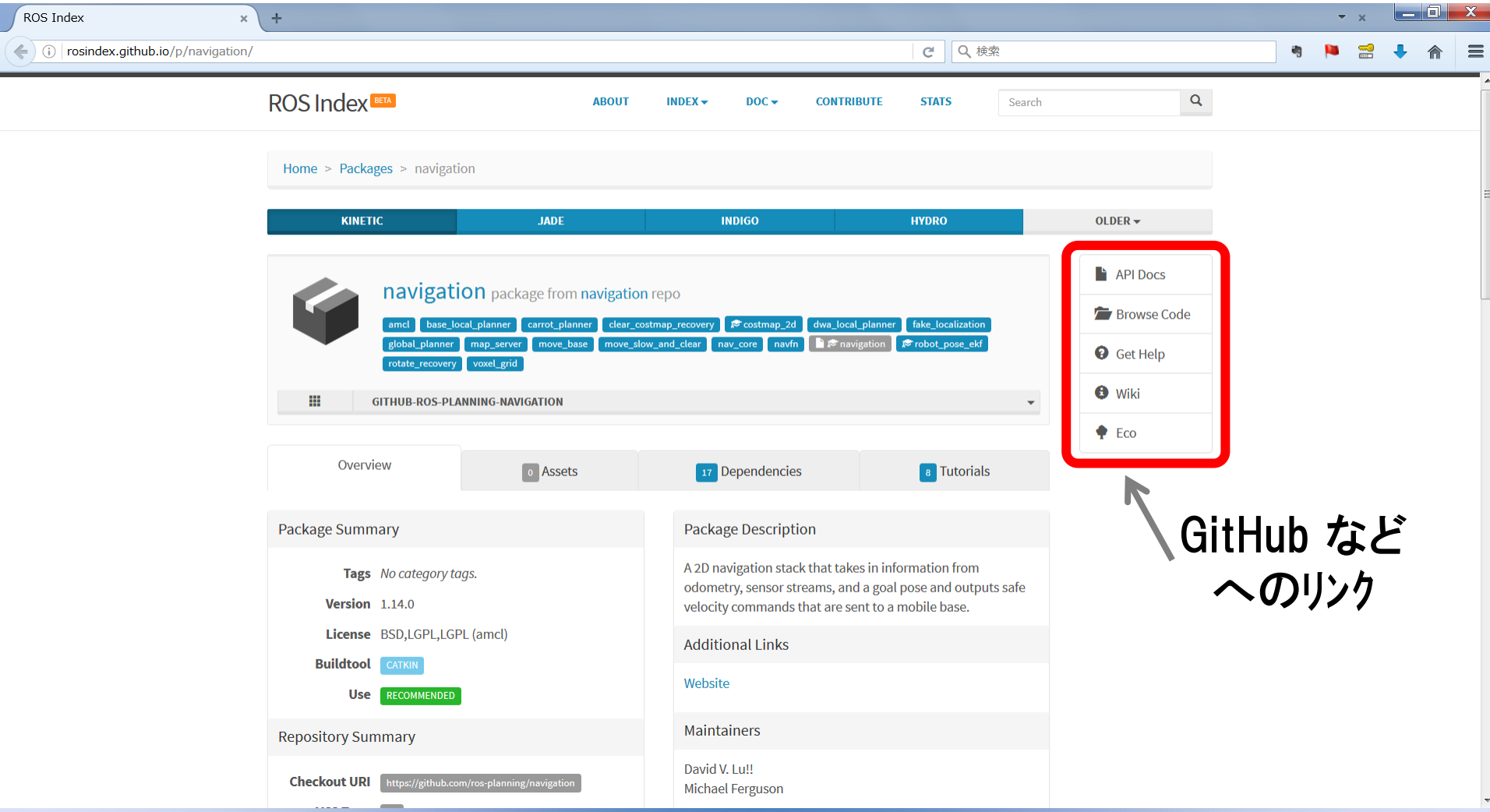
ROS に関する情報源

- **ROS.org**（ﾊﾟｯｹｰｼﾞﾘｽﾄ、ﾆｭｰｽなど）
  https://www.ros.org/
- **ROS Wiki**（共同編集文章）
  https://wiki.ros.org/
- **ROS Answers**（質問掲示板）
  https://answers.ros.org/
- **ROS Index**（ﾊﾟｯｹｰｼﾞ情報、2015年ﾛｰﾝﾁ）
  https://index.ros.org/
- **GitHub Organizations**（ｿｰｽｺｰﾄﾞ、Issue Trackers）
  https://github.com/ros-planning
  https://github.com/ros-perception など
- **C++ / Python API**（Doxygen / Sphinx 生成文章）
  https://docs.ros.org/en/api/ 以下の各ﾃﾞｨﾚｸﾄﾘ
- **ROS Discourse**
  https://discourse.ros.org/

情報が実装と一致しない場合も多い
（特に日本語ﾍﾟｰｼﾞは古い）

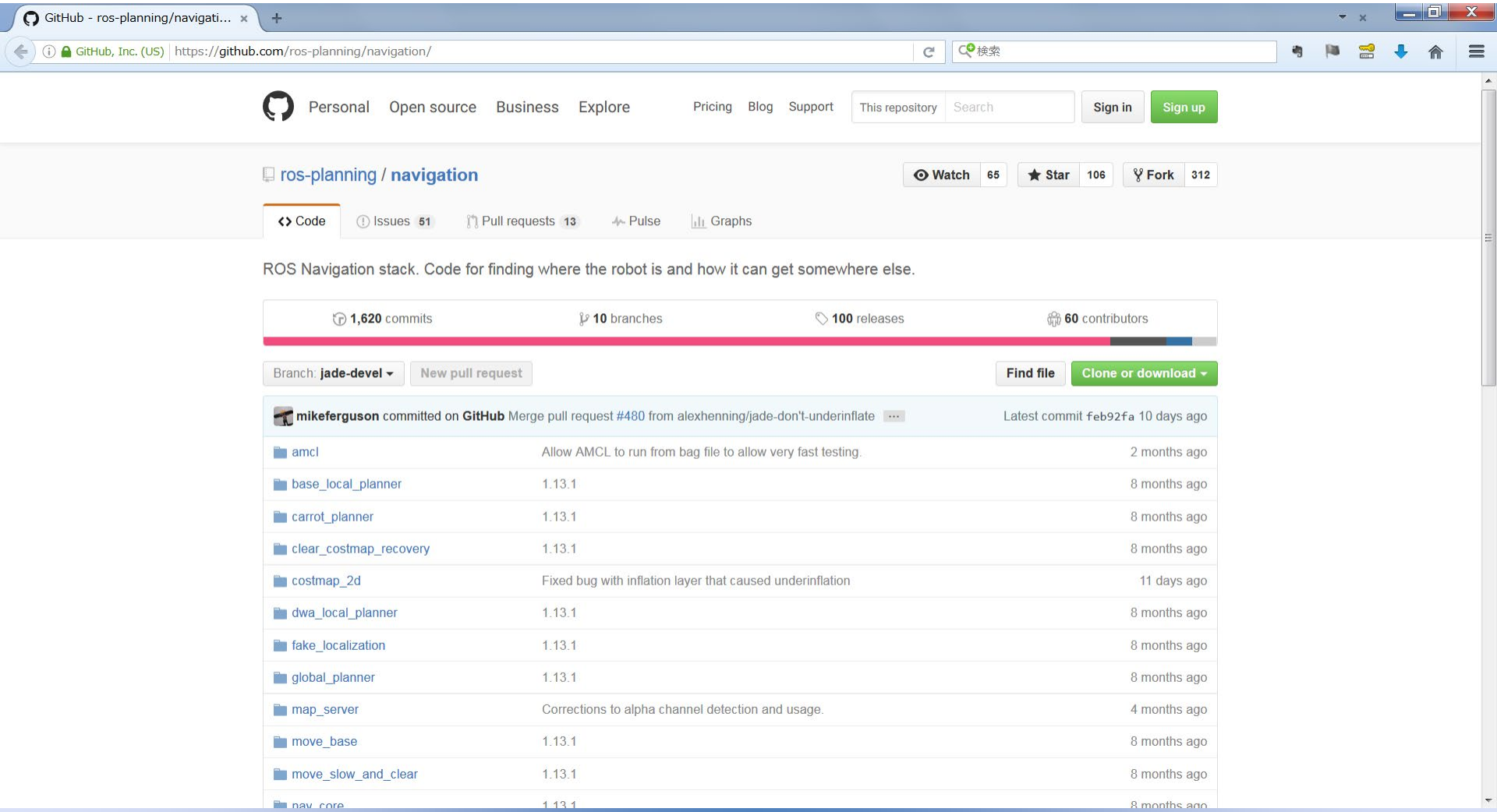
ROS Index（ﾊﾟｯｹｰｼﾞ情報を集約）

README や package.xml からの自動生成文章



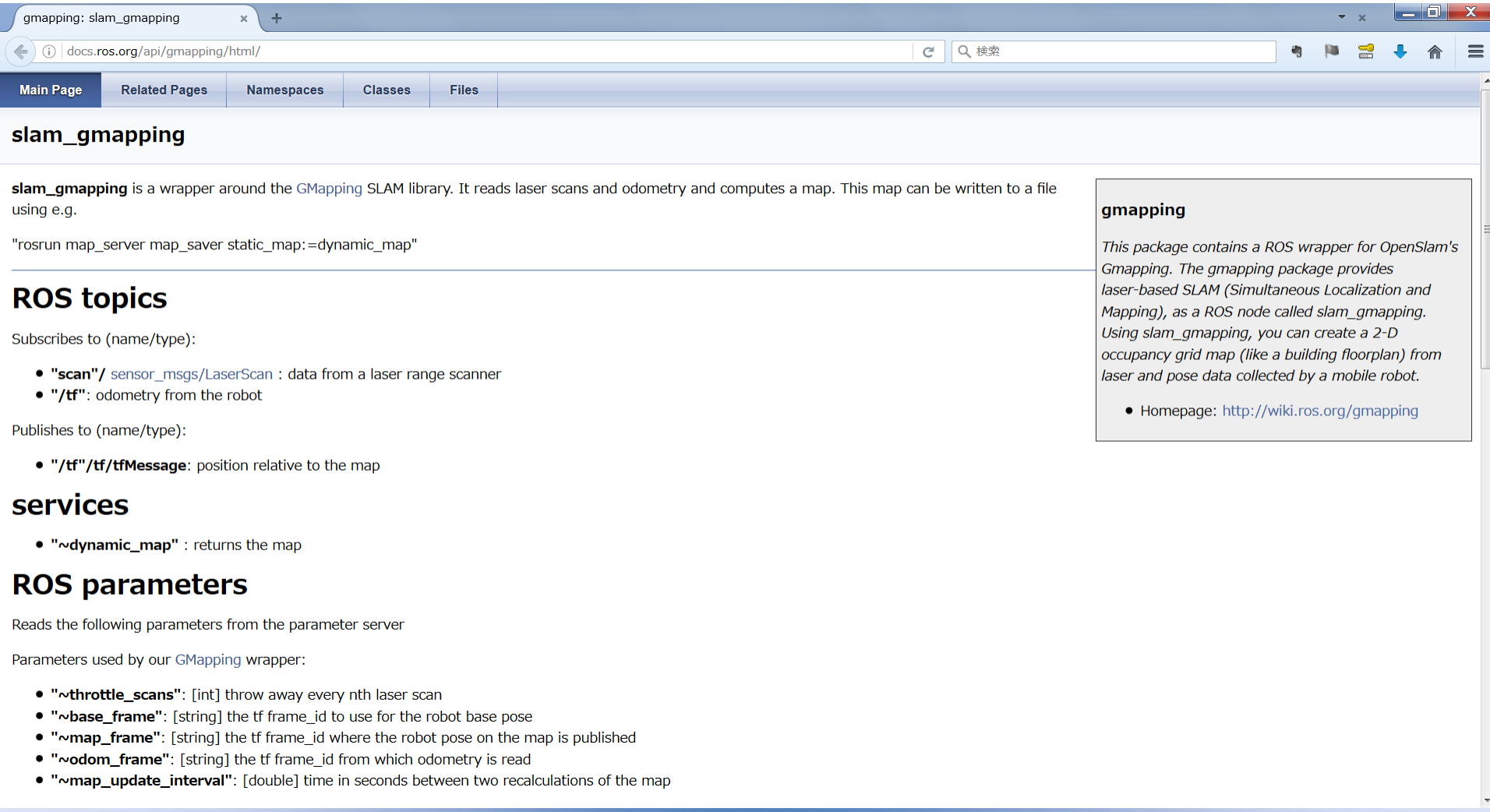
GitHub Organizations（ｺｰﾄﾞ、Issues）

ﾊﾟｯｹｰｼﾞの種類ごとに組織されている



ROS C++/Python API (Doxygen/Sphinx)

ｿｰｽｺｰﾄﾞのｺﾒﾝﾄからの自動生成文章



自律走行の実現に役立つﾁｭｰﾄﾘｱﾙﾍﾟｰｼﾞ

- **ROS 概要**
  https://wiki.ros.org/ROS/StartGuide
- **ROS 基本ﾁｭｰﾄﾘｱﾙ**（ﾌﾟﾛｾｽ間通信、開発ﾂｰﾙなど）
  https://wiki.ros.org/ROS/Tutorials
  https://wiki.ros.org/roscpp_tutorials/Tutorials
- **tf ﾁｭｰﾄﾘｱﾙ**（座標変換）
  https://wiki.ros.org/tf/Tutorials
  https://wiki.ros.org/tf2/Tutorials
- **navigation ﾁｭｰﾄﾘｱﾙ**（自律走行）
  https://wiki.ros.org/navigation/Tutorials/RobotSetup
  https://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide
- **slam_gmapping ﾁｭｰﾄﾘｱﾙ**（SLAM、地図生成）
  https://wiki.ros.org/slam_gmapping/Tutorials/MappingFromLoggedData
- **rviz ﾁｭｰﾄﾘｱﾙ**（ﾋﾞｭｰｱ）
  https://wiki.ros.org/rviz/Tutorials
  https://wiki.ros.org/navigation/Tutorials/Using%20rviz%20with%20the%20Navigation%20Stack

Contents

1. 自律走行を実現するための ROS ﾊﾟｯｹｰｼﾞ
2. 自己位置推定と SLAM の基本
3. 各ﾊﾟｯｹｰｼﾞのｱﾙｺﾞﾘｽﾞﾑ
   a. 自己位置推定（amcl）
   b. SLAM、地図生成（gmapping）
   c. 大域的経路計画、局所的動作計画（move_base）
4. ROS に関する情報の調べ方
5. **ROS での開発に関する知見**
6. まとめ

千葉工業大学 未来ロボット技術研究センター
Future Robotics Technology Center

ROS ﾊﾟｯｹｰｼﾞの様々な使い方

- **ROS API で既存ﾊﾟｯｹｰｼﾞを launch するだけ**
  - ROS が想定している基本的な使い方
  - ﾒｯｾｰｼﾞ型は比較的良く整理されている
- **C++/Python API で使用（ﾗｲﾌﾞﾗﾘとして使用）**
  - ｱﾙｺﾞﾘｽﾞﾑの中身を個別に使える
  - ﾗｲﾌﾞﾗﾘとしては整理が不充分なﾊﾟｯｹｰｼﾞも多い
- **既存ﾊﾟｯｹｰｼﾞの中身を改造**
  - ｿｰｽｺｰﾄﾞが整理されておらず、苦労する場合もある
- **新規ﾊﾟｯｹｰｼﾞを作成**
  - 標準的なﾒｯｾｰｼﾞ型に従えば、既存ﾊﾟｯｹｰｼﾞと連携可

ﾊﾟﾗﾒｰﾀ調整には
ｱﾙｺﾞﾘｽﾞﾑの
理解が必要

slam_gmapping, navigation にない機能

- 3次元空間の SLAM、地図生成
- Graph-based SLAM
- PF による3次元6自由度での自己位置推定
  （EKF は robot_pose_ekf ﾊﾟｯｹｰｼﾞで可能）
- 3次元点群を用いた SLAM／自己位置推定
- x, y, yaw の3自由度ｺﾝﾌｨｸﾞﾚｰｼｮﾝ空間
- 3次元6自由度での経路計画
- 3次元点群を用いた段差などの障害物の検出
- 移動障害物の検出と追跡

ROS のﾋﾞﾙﾄﾞｼｽﾃﾑ

- **rosbuild**（2012年以前の古いｼｽﾃﾑ）
- **catkin（catkin_make ｺﾏﾝﾄﾞ、全ﾊﾟｯｹｰｼﾞ同時ﾋﾞﾙﾄﾞ）**
  https://wiki.ros.org/catkin
  ﾜｰｸｽﾍﾟｰｽ src/ 直下にﾄｯﾌﾟﾚﾍﾞﾙ CMakeLists.txt あり
  WS初期化　$ catkin_init_workspace
  ﾊﾟｯｹｰｼﾞ生成　$ catkin_create_pkg <pkg_name>
  ﾋﾞﾙﾄﾞ　$ catkin_make
- **catkin_tools（catkin ｺﾏﾝﾄﾞ、各ﾊﾟｯｹｰｼﾞ個別ﾋﾞﾙﾄﾞ）**
  catkin_make_isolated の後継、2015年ﾛｰﾝﾁ、ﾍﾞｰﾀ版
  https://catkin-tools.readthedocs.io/
  ﾜｰｸｽﾍﾟｰｽ src/ 直下にﾄｯﾌﾟﾚﾍﾞﾙ CMakeLists.txt なし
  WS初期化　$ catkin init
  ﾊﾟｯｹｰｼﾞ生成　$ catkin create <pkg_name>
  ﾋﾞﾙﾄﾞ　$ catkin build

catkin ﾜｰｸｽﾍﾟｰｽの構成に関する Tips

- **ﾌｧｲﾙﾂﾘｰは階層的なﾃﾞｨﾚｸﾄﾘでも良い**

```
catkin_ws/                          (catkin ﾜｰｸｽﾍﾟｰｽ)
    src/                            (src ﾃﾞｨﾚｸﾄﾘ)
        stack_dir/                  (複数ﾊﾟｯｹｰｼﾞを集約)
            pkg_a/                  (ﾊﾟｯｹｰｼﾞ a)
            pkg_b/                  (ﾊﾟｯｹｰｼﾞ b)
```

- **複数のﾜｰｸｽﾍﾟｰｽをｵｰﾊﾞｰﾚｲできる**
  https://wiki.ros.org/catkin/Tutorials/workspace_overlaying
  - 同名のﾊﾟｯｹｰｼﾞをｵｰﾊﾞｰﾚｲ可能
  - 初回の catkin_make ｺﾏﾝﾄﾞ実行時に読み込んでいた設定が新しく生成される setup.bash に引き継がれる
  - setup.bash は最後に読み込んだもので上書きされる

既存ﾊﾟｯｹｰｼﾞの改造に便利

Subscriber に登録できるｺｰﾙﾊﾞｯｸの型

boost::function でｻﾎﾟｰﾄされた型を登録可能

https://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers

- **通常の関数**
- **ﾒﾝﾊﾞ関数（ｸﾗｽのﾒｿｯﾄﾞ）**
  https://wiki.ros.org/roscpp_tutorials/Tutorials/UsingClassMethodsAsCallbacks
- 関数ﾎﾟｲﾝﾀ
- 関数ｵﾌﾞｼﾞｪｸﾄ（ﾌｧﾝｸﾀ）
- **boost::bind**
  - ｺｰﾙﾊﾞｯｸ関数に引数を渡す場合に使用
  - C++11 の bind には ROS が未対応

Subscriber へのｺｰﾙﾊﾞｯｸ関数の登録方法

- 通常の関数、引数なし
- 通常の関数、引数あり← boost::bind 使用
- ﾒﾝﾊﾞ関数、引数なし
  （ｸﾗｽのｵﾌﾞｼﾞｪｸﾄ／ﾎﾟｲﾝﾀ経由で登録）
- ﾒﾝﾊﾞ関数、引数あり← boost::bind 使用
  （ｸﾗｽのｵﾌﾞｼﾞｪｸﾄ／ﾎﾟｲﾝﾀ経由で登録）
- ﾒﾝﾊﾞ関数、ｸﾗｽ内 Subscriber、引数なし
- ﾒﾝﾊﾞ関数、ｸﾗｽ内 Subscriber、引数あり
  ← boost::bind 使用

推奨

通常の関数、引数なし

```
void cbFunc(const std_msgs::String::ConstPtr &msg) {
  // コールバック処理
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "subscriber_example");
  ros::NodeHandle nh;
  ros::Subscriber sub
    = nh.subscribe("topic_name", 100, cbFunc);
  ros::spin();
  return 0;
}
```

Subscriber のﾃﾝﾌﾟﾚｰﾄ型は
基本的には省略可能

通常の関数、引数あり（値渡し）

```
void cbFuncCopy(const std_msgs::String::ConstPtr &msg, int num) {
  // コールバック処理
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "subscriber_example");
  ros::NodeHandle nh;
  int num = 1234;
  ros::Subscriber sub
    = nh.subscribe<std_msgs::String>("topic_name", 100,
        boost::bind(cbFuncCopy, _1, num));
  ros::spin();
  return 0;
}
```

bind の場合は Subscriber の ﾃﾝﾌﾟﾚｰﾄ型を省略できない

C++11 の bind は未対応

通常の関数、引数あり（参照渡し）

```
void cbFuncRef(const std_msgs::String::ConstPtr &msg,
  std::vector<int> &obj) {
  // コールバック処理
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "subscriber_example");
  ros::NodeHandle nh;
  std::vector<int> obj;
  ros::Subscriber sub
    = nh.subscribe<std_msgs::String>("topic_name", 100,
        boost::bind(cbFuncRef, _1, std::ref(obj)));
  ros::spin();
  return 0;
}
```

参照渡しは ref() / cref() が必要
（C++11 と boost のどちらも OK）

ﾒﾝﾊﾞ関数、引数なし（ｸﾗｽのｵﾌﾞｼﾞｪｸﾄ）

```
class Foo {
public:
  void cbMethod(const std_msgs::String::ConstPtr &msg) {
    // コールバック処理
  }
};
int main(int argc, char **argv) {
  ros::init(argc, argv, "subscriber_example");
  ros::NodeHandle nh;
  Foo foo_obj;
  ros::Subscriber sub
    = nh.subscribe("topic_name", 100, &Foo::cbMethod, &foo_obj);
  ros::spin();
  return 0;
}
```

ﾒﾝﾊﾞ関数、引数なし（ｸﾗｽのﾎﾟｲﾝﾀ経由）

```
class Foo {
public:
  void cbMethod(const std_msgs::String::ConstPtr &msg) {
    // コールバック処理
  }
};
int main(int argc, char **argv) {
  ros::init(argc, argv, "subscriber_example");
  ros::NodeHandle nh;
  auto foo_ptr = boost::make_shared<Foo>();
  ros::Subscriber sub
    = nh.subscribe("topic_name", 100, &Foo::cbMethod, foo_ptr);
  ros::spin();
  return 0;
}
```

C++11 の shared_ptr や unique_ptr は未対応

ﾒﾝﾊﾞ関数、引数あり

```
class Foo {
public:
  void cbMethodArg(const std_msgs::String::ConstPtr &msg,
    std::vector<int> &obj) {
    // コールバック処理
  }
};
int main(int argc, char **argv) {
  ros::init(argc, argv, "subscriber_example");
  ros::NodeHandle nh;
  Foo foo_obj;
  std::vector<int> obj;
  ros::Subscriber sub
    = nh.subscribe<std_msgs::String>("topic_name", 100,
        boost::bind(&Foo::cbMethodArg, &foo_obj, _1, std::ref(obj)));
  ros::spin();
  return 0;
}
```

ﾒﾝﾊﾞ関数、ｸﾗｽ内 Subscriber、引数なし

ROS ﾉｰﾄﾞ ｸﾗｽ

推奨

```
class BarNode {
public:
  BarNode() {
    sub_ = nh_.subscribe("topic_name", 100, &BarNode::cbMethod, this);
  }
private:
  void cbMethod(const std_msgs::String::ConstPtr &msg) {
    // コールバック処理
  }
  ros::NodeHandle nh_;
  ros::Subscriber sub_;
};
int main(int argc, char **argv) {
  ros::init(argc, argv, "subscriber_example");
  BarNode bar_obj;
  ros::spin();
  return 0;
}
```

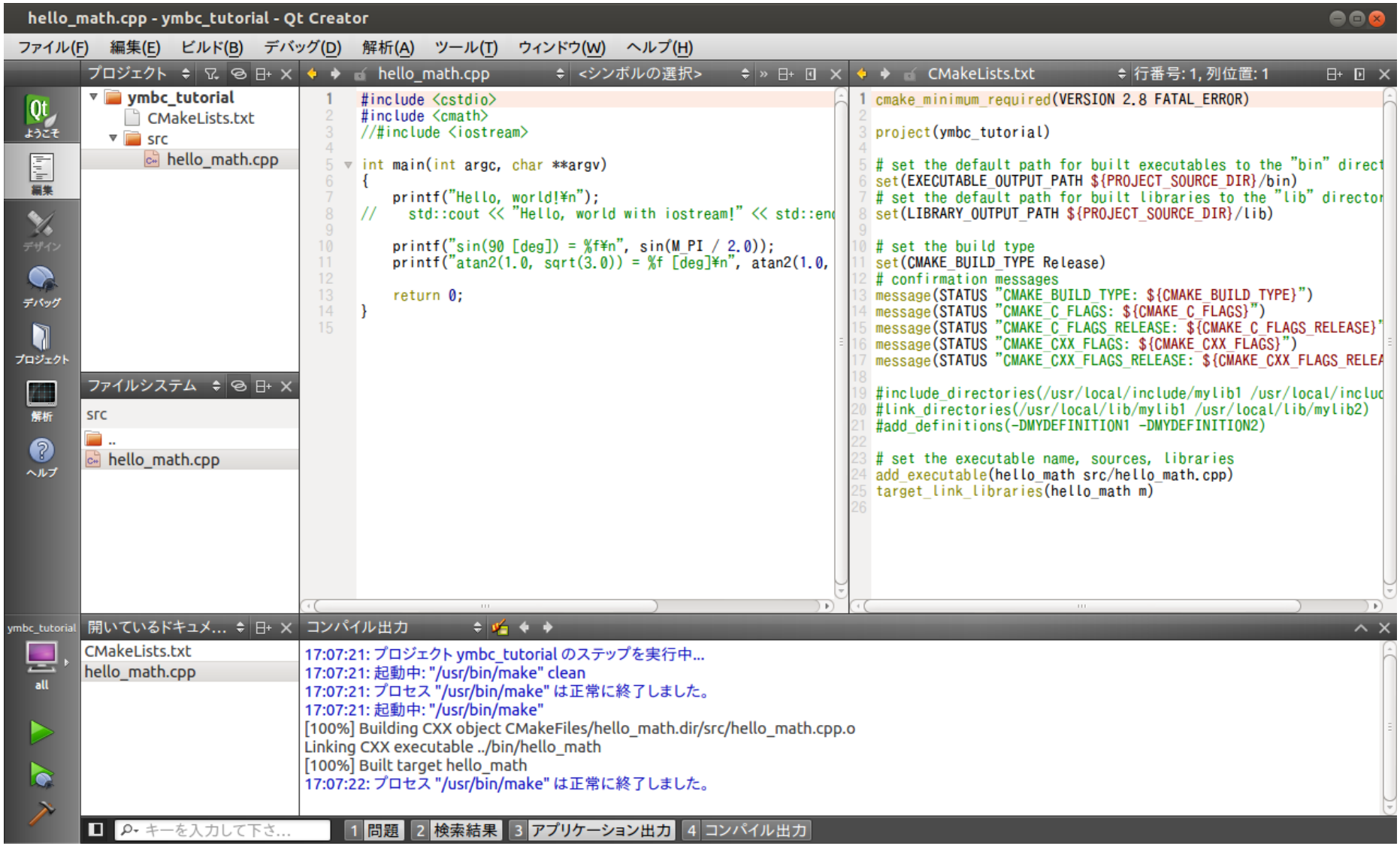
ﾒﾝﾊﾞ関数、ｸﾗｽ内 Subscriber、引数あり

```
class BarNode {
public:
  BarNode() {
    std::vector<int> obj;
    sub_ = nh_.subscribe<std_msgs::String>("topic_name", 100,
      boost::bind(&BarNode::cbMethodArg, this, _1, obj));
  }
private:
  void cbMethodArg(const std_msgs::String::ConstPtr &msg,
    std::vector<int> obj) {
    // コールバック処理
  }
  ros::NodeHandle nh_;
  ros::Subscriber sub_;
};
int main(int argc, char **argv) {
  ros::init(argc, argv, "subscriber_example");
  BarNode bar_obj;
  ros::spin();
  return 0;
}
```

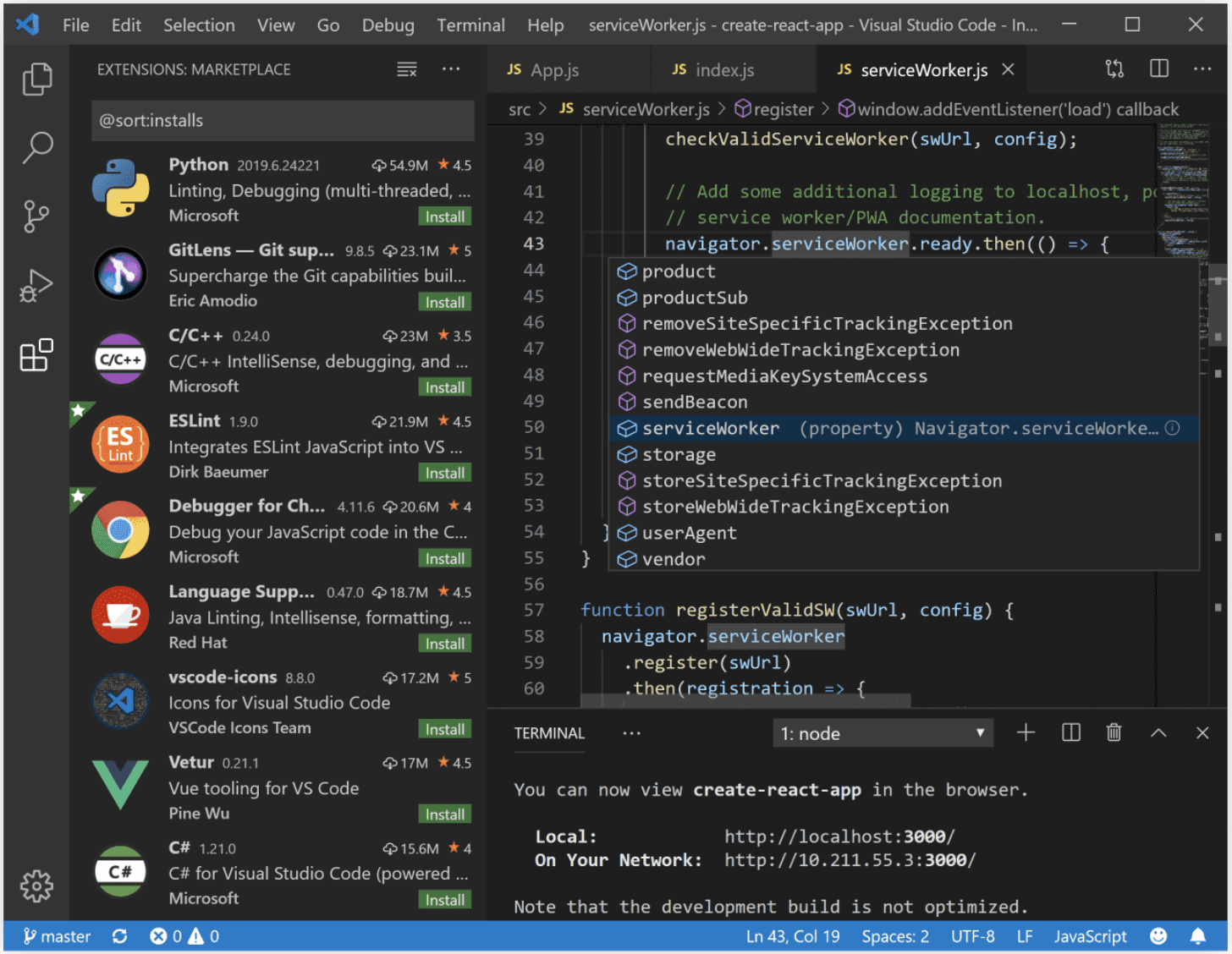
ROS ﾉｰﾄﾞ ｸﾗｽ

参照渡しの場合は
obj が既に無効なので注意

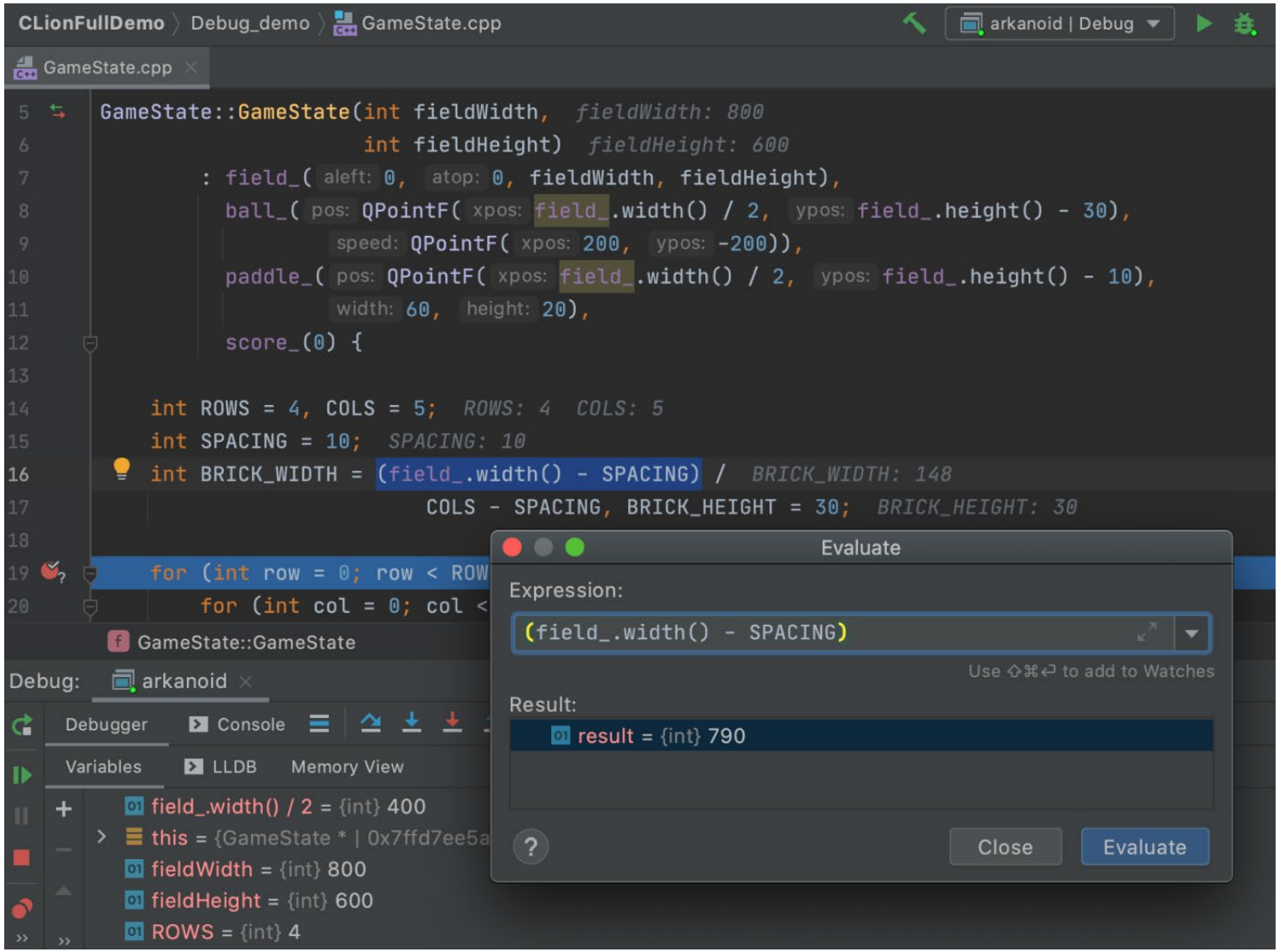
統合開発環境 Qt Creator



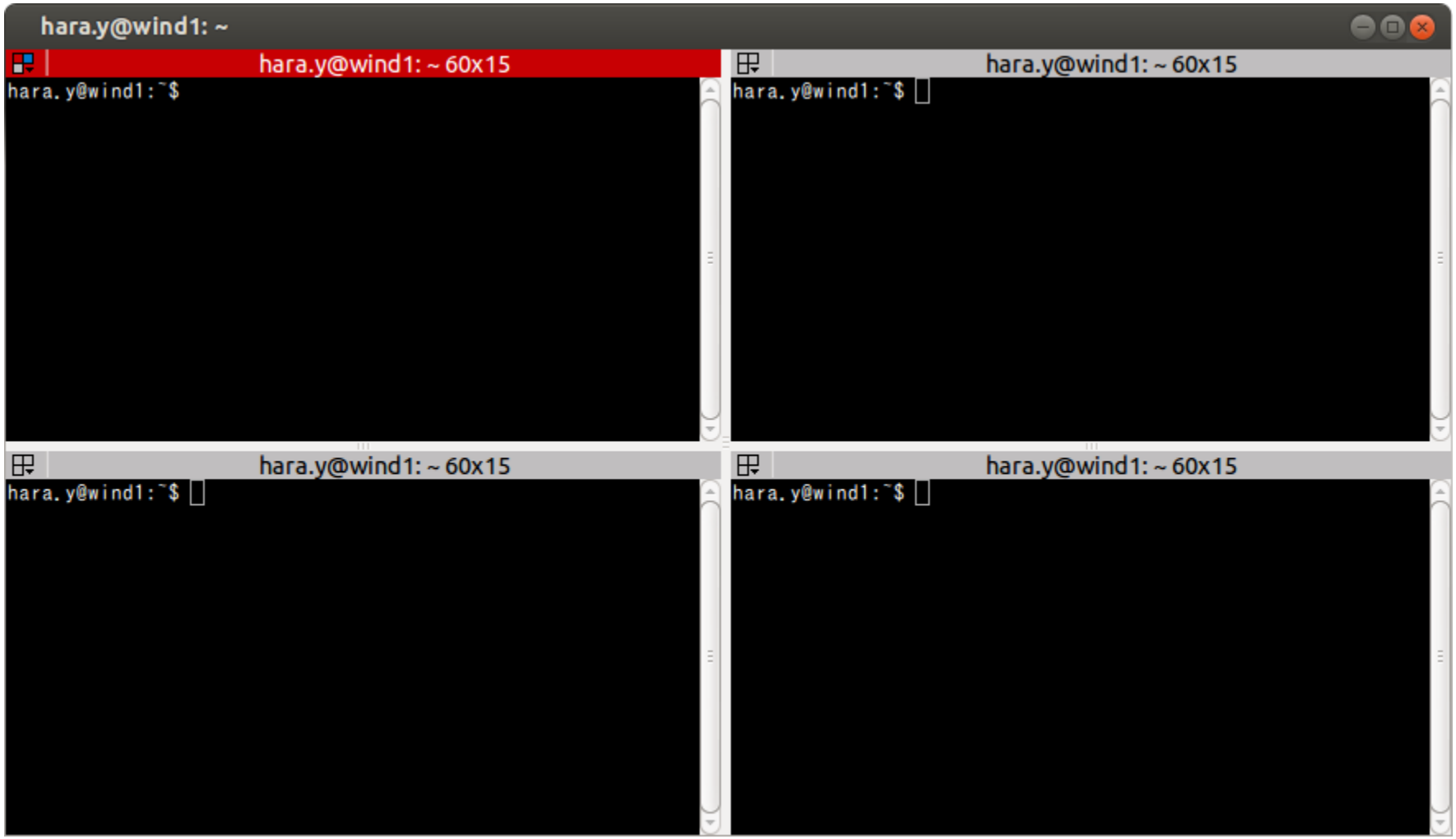
統合開発環境 Visual Studio Code



統合開発環境 CLion



画面分割できるﾀｰﾐﾅﾙ Terminator



screen / tmux のような画面分割を簡単に利用可能

Contents

1. 自律走行を実現するための ROS ﾊﾟｯｹｰｼﾞ
2. 自己位置推定と SLAM の基本
3. 各ﾊﾟｯｹｰｼﾞのｱﾙｺﾞﾘｽﾞﾑ
   a. 自己位置推定（amcl）
   b. SLAM、地図生成（gmapping）
   c. 大域的経路計画、局所的動作計画（move_base）
4. ROS に関する情報の調べ方
5. ROS での開発に関する知見
6. まとめ

まとめ

- **slam_gmapping, navigation の構成、ｱﾙｺﾞﾘｽﾞﾑ**
  - 自己位置推定：Adaptive Monte Carlo Localization
  - SLAM、地図生成：Rao-Blackwellized Particle Filter SLAM（FastSLAM 2.0 での Grid Mapping）
  - 大域的経路計画：Navigation Function とﾀﾞｲｸｽﾄﾗ法
  - 局所的動作計画：Dynamic Window Approach
- **ROS に関する情報の調べ方**（ROS Index を活用）
- **ROS での開発に関する知見**
  - catkin_make ｺﾏﾝﾄﾞ／catkin ｺﾏﾝﾄﾞ（新しいﾋﾞﾙﾄﾞｼｽﾃﾑ）
  - Subscriber へのｺｰﾙﾊﾞｯｸ関数の登録方法
  - Qt Creator：CMake を開ける統合開発環境
  - Terminator：画面分割できるﾀｰﾐﾅﾙ

参考文献

- **ROS Wiki**
  https://wiki.ros.org/
- 友納 正裕: **“移動ﾛﾎﾞｯﾄのための確率的な自己位置推定と地図構築”**, 日本ﾛﾎﾞｯﾄ学会誌, vol. 29, no. 5, pp. 423-426, 2011.
- 友納 正裕: **“移動ﾛﾎﾞｯﾄの自己位置推定と地図構築の基礎”**, 第47回ﾛﾎﾞｯﾄ工学ｾﾐﾅｰ, 2008.
- Sebastian Thrun, Wolfram Burgard, and Dieter Fox: **“Probabilistic Robotics”**, The MIT Press, 2005.
  （邦訳）上田 隆一: **“確率ﾛﾎﾞﾃｨｸｽ”**, ﾏｲﾅﾋﾞ, 2007.
- 坪内 孝司: **“移動体の位置認識”**, 計測自動制御学会 編, ﾋﾞｰｸﾙ, ｺﾛﾅ社, 2003.
- Dieter Fox, Wolfram Burgard, and Sebastian Thrun: **“The Dynamic Window Approach to Collision Avoidance”**, IEEE Robotics & Automation Mag., vol. 4, no. 1, pp. 23-33, 1997.