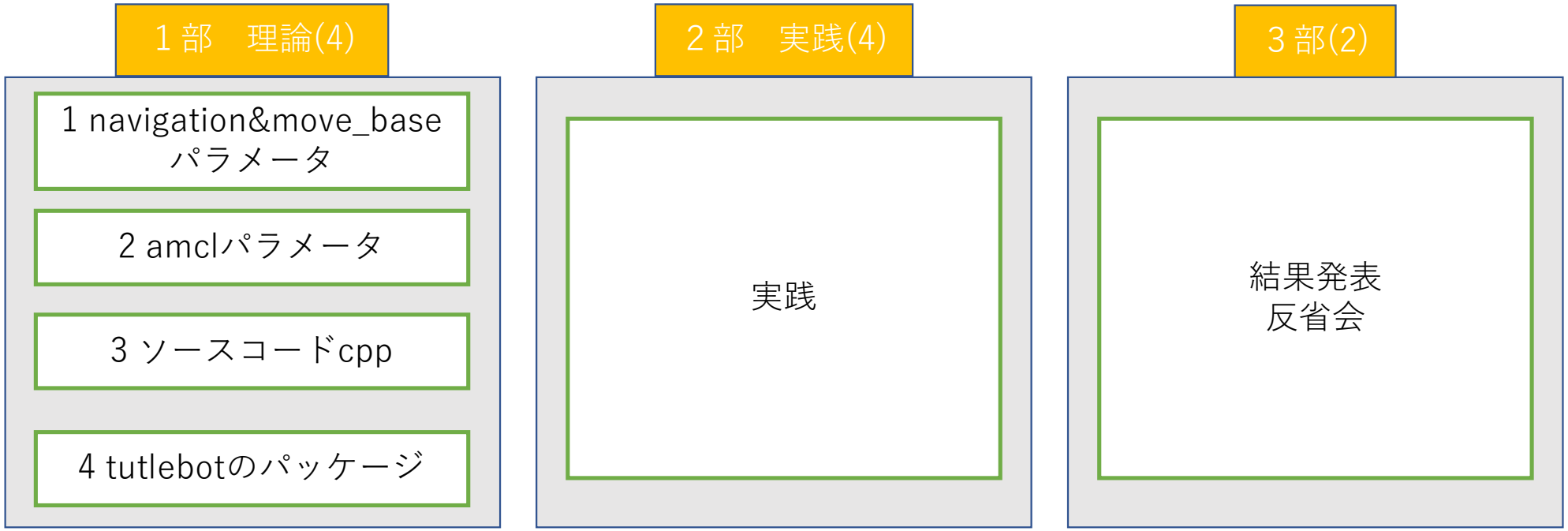
CIT自律移動勉強会(navigation)
第3回
move_baseの中身

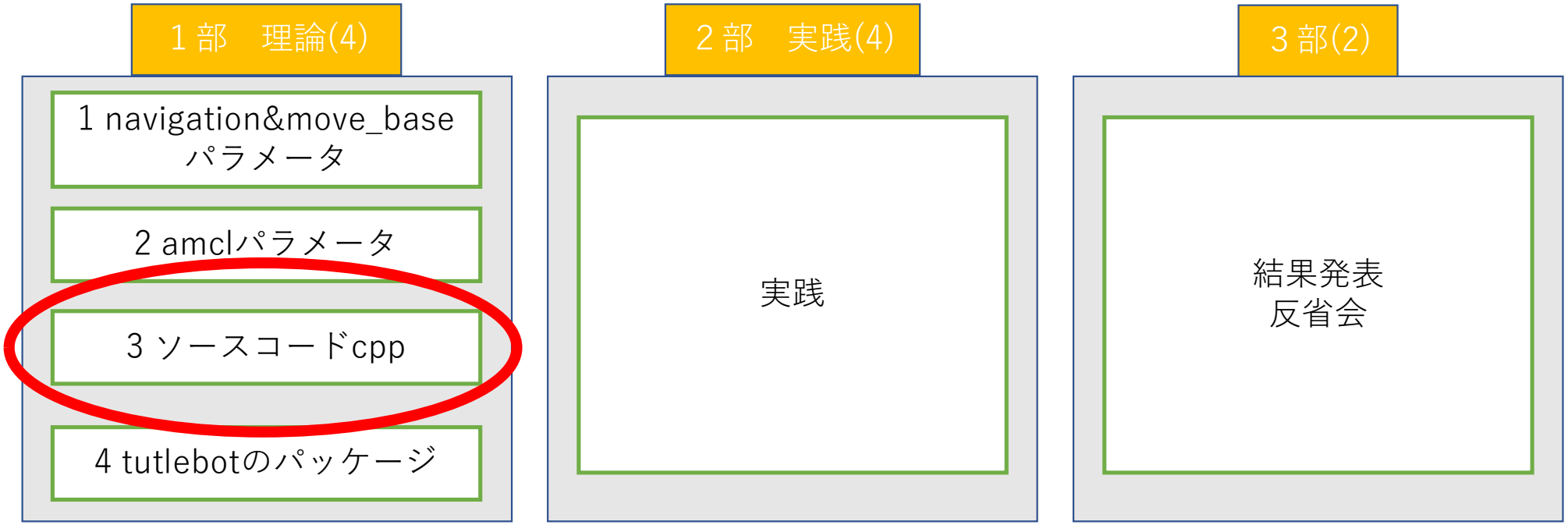
目次

- 勉強会スケジュール
- (復習)Movebaseについて
- ソースコード(cpp)

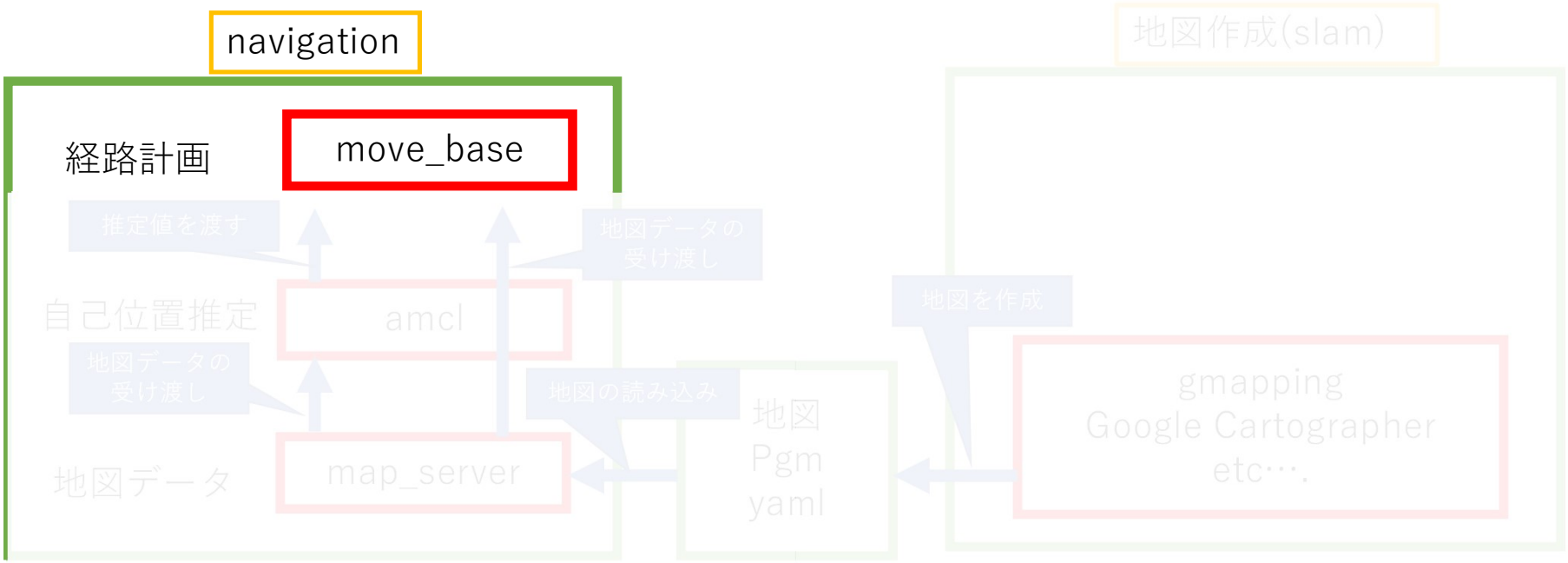
スケジュール


1部　理論(4)
1 navigation&move_base
パラメータ
2 amclパラメータ
3 ソースコードcpp
4 tutlebotのパッケージ
2部　実践(4)
実践
3部(2)
結果発表
反省会

スケジュール


1部　理論(4)
1 navigation&move_base
パラメータ
2 amclパラメータ
3 ソースコードcpp
4 tutlebotのパッケージ
2部　実践(4)
実践
3部(2)
結果発表
反省会

もっとわかりやすく//////



参考　Navigation Stack を理解する@MoriKen

move_base

- Navigationの中心部（親玉）
  様々な情報を加味して、最終的に動作を決める
  後述するlaunchのみでも動作可能
  （自己位置推定なし！？）
- コストマップ→通れるか？障害物？を表したmap
  - Localcostmap(局所的狭い範囲)
  - Globalcostmap(大域的全域)
- 経路計画→現在地からゴールまでや障害物の回避経路
  - Localplanner(局所的、障害物回避など)
  - globalplanner(大域的、最終目的地までの全体)

costmap

- 占有格子地図で表す通れそうな領域、障害物のある領域を数値で表現した地図(Costmap2D 2 次元に限定)

- 空間をグリッド(四角）で区切り、グリッドにおける
  障害物っぽさを0~255で表現

経路計画(local_planner)

動的ウィンドウアプローチ（DWA）アルゴリズム
①ロボットの制御空間（dx、dy、dtheta）で個別にサンプリング
②サンプリングされた速度ごとに、ロボットの現在の状態から前方シミュレーションを実行して、サンプリングされた速度が一定の時間適用された場合に何が起こるかを予測
③障害物への近接性、ゴールへの近接性、グローバルパスへの近接性、速度などを盛り込んだ前方シミュレーションから得られた各軌道を下記の式で 評価 （スコアリング)、障害物と衝突する軌道を破棄。

$$G(v,\omega)=\sigma(\alpha\times heading(v,\omega)+\beta\times dist(v,\omega)+\gamma\times velocity(v,\omega))$$

heading(v,ω):制御入力の時のロボットの方位とゴール方向の差の角度を180度から引いた値
dist(v,ω):制御入力の時の、最近傍の障害物までの距離
velocity(v,ω):制御入力の速度
④最高スコアの軌道を選択し、関連する速度を出力。
⑤繰り返し

Move_base.cpp

- 注意
  - 私自身がプログラムに明るい人ではないため、訂正、ご意見募集

Move_base.cpp

- 初期化処理後、主となる動作はgoalトピックを受信するまでは何もしない。
  - 受信後、設定されたプランナーに従ってゴールまで移動する。
- PLANNING,CONTROLLING, CLEARINGの3つの状態を持つ。
  - PLANNING
    - グローバルプラン生成中の状態
  - CONTROLLING
    - ローカルプランを生成しゴールまで移動している状態
  - CLEARING
    - 障害物クリアモード

主な流れ

- 初期化処理
  - MoveBaseインスタンスを生成(実体って呼ばれる)
  - アクションサーバー起動
  - goalトピック受信時の設定
  - グローバルプランナー用スレッド起動
  - グローバルプランナーの設定
  - ローカルプランナーの設定
  - その他
- goalトピック受信後
  - goalトピックをアクションサーバーに配信
  - プランナーに従ってゴールまで移動する

MoveBaseインスタンスを生成

- move_baseノードを実行するとmove_base_node.cppのメイン関数が呼び出され、MoveBaseクラスのインスタンスが生成される。
- 以降の初期化処理はMoveBaseクラスのコンストラクタでの処理になる

アクションサーバー起動

- as_ = new MoveBaseActionServer(ros::NodeHandle(), "move_base", boost::bind(&MoveBase::executeCb, this, _1), false);

- 最初にアクションサーバーを起動させる。
- コールバック関数にはアクションの目標をサーバーが受信した時に起動するexecuteCb関数を登録する。

goalトピック受信時の設定

- ros::NodeHandle simple_nh("move_base_simple");
- goal_sub_ = simple_nh.subscribe<geometry_msgs::PoseStamped>("goal", 1, boost::bind(&MoveBase::goalCB, this, _1));

- goalトピックを受信したときに処理を行うコールバック関数を登録

グローバルプランナー用スレッドを起動

- planner_thread_ = new boost::thread(boost::bind(&MoveBase::planThread, this));

- グローバルプランナー用スレッドを生成する。
- スレッドで動作させる関数にはplanThread関数を登録

グローバルプランナー用スレッドを起動

- while (wait_for_wake || !runPlanner_)
- {
- // if we should not be running the planner then suspend this thread
- ROS_DEBUG_NAMED("move_base_plan_thread", "Planner thread is suspending");
- planner_cond_.wait(lock);
- wait_for_wake = false;
- }
- スレッド生成後は、プランナー生成の準備が整うまでスレッドをサスペンドする。
- planThread関数内で以下の様なサスペンド処理が実行される。

グローバルプランナーの設定

```
planner_costmap_ros_ = new costmap_2d::Costmap2DROS("global_costmap", tf_);
 planner_costmap_ros_->pause();

 // initialize the global planner
 try
 {
  planner_ = bgp_loader_.createInstance(global_planner);
  planner_->initialize(bgp_loader_.getName(global_planner), planner_costmap_ros_);
 }
 catch (const pluginlib::PluginlibException &ex)
 {
  ROS_FATAL("Failed to create the %s planner, are you sure it is properly "
            "registered and that the containing library is built? Exception: %s",
            global_planner.c_str(), ex.what());
  exit(1);
 }
```

グローバルプランナーの設定

- グローバルプランナー用コストマップとしてCostmap2DROSクラスのインスタンスを生成する。
- 生成後、コストマップは一時停止。
- インスタンス生成後、ClassLoaderを用いてグローバルプランナーのインスタンスを生成する。
- プランナーのインスタンスを初期化する。

ローカルプランナーの設定

```
controller_costmap_ros_ = new costmap_2d::Costmap2DROS("local_costmap", tf_);
 controller_costmap_ros_->pause();

 // create a local planner
 try
 {
  tc_ = blp_loader_.createInstance(local_planner);
  tc_->initialize(blp_loader_.getName(local_planner), &tf_, controller_costmap_ros_);
 }
 catch (const pluginlib::PluginlibException &ex)
 {
  ROS_FATAL("Failed to create the %s planner, are you sure it is properly "
            "registered and that the containing library is built? Exception: %s",
            local_planner.c_str(), ex.what());
  exit(1);
 }
```

ローカルプランナーの設定

- ローカルプランナー用コストマップとしてCostmap2DROSクラスのインスタンスを生成する。
- 生成後、コストマップは一時停止
- インスタンス生成後、ClassLoaderを用いてローカルプランナーのインスタンスを生成する。
- プランナーのインスタンスを初期化する。

その他

- planner_costmap_ros_->start();
- controller_costmap_ros_->start();

- state_ = PLANNING;

- as_->start();

その他

- 各コストマップを一時停止から開始へ遷移させる。
- move_baseの状態をPLANNINGへ遷移
- アクションサーバーの開始

goalトピック受信後
goalトピックをアクションサーバーに配信

- void MoveBase::goalCB(const geometry_msgs::PoseStamped::ConstPtr &goal)
- {
- ROS_DEBUG_NAMED("move_base", "In ROS goal callback, wrapping the PoseStamped in the "
- "action message and re-sending to the server.");
- move_base_msgs::MoveBaseActionGoal action_goal;
- action_goal.header.stamp = ros::Time::now();
- action_goal.goal.target_pose = *goal;

- action_goal_pub_.publish(action_goal);
- }

goalトピックをアクションサーバーに配信

- goalCBが起動し、アクションサーバーに目標を配信する処理を行う。

プランナーに従って移動する処理

```
// ゴール到着判定
if (tc_->isGoalReached())
{
  ROS_DEBUG_NAMED("move_base", "Goal reached!");
  resetState();

  // disable the planner thread
  boost::unique_lock<boost::recursive_mutex> lock(planner_mutex_);
  runPlanner_ = false; // プランナースレッドのサスペンド指示
  lock.unlock();
  // アクションサーバーに目標到達を送信
  as_->setSucceeded(move_base_msgs::MoveBaseResult(), "Goal reached.");
  return true;
}
```

プランナーに従って移動する処理

- // 選択したプランナーに応じたcomputeVelocityCommands関数が実行され、ローカルプランの生成と速度コマンドの算出を行う。
- if (tc_->computeVelocityCommands(cmd_vel))
- {
- ROS_DEBUG_NAMED("move_base", "Got a valid command from the local planner: %.3lf, %.3lf, %.3lf",cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z);
- last_valid_control_ = ros::Time::now();
- // make sure that we send the velocity command to the base
- vel_pub_.publish(cmd_vel);
- if (recovery_trigger_ == CONTROLLING_R)
- recovery_index_ = 0;
- }

プランナーに従って移動する処理

- アクションサーバーに目標が配信されるとexecuteCb関数が起動する。
- この関数では、予め設定してた周期で無限ループを繰り返しながらmove_baseの状態に応じて以下のような処理を行う。
- ローカルプランナーを生成する処理
- ロボットへ速度コマンドを送信する処理
- ゴール到着判定処理。
- その他
- リカバリー処理など
- グローバルプランがまだ生成されていない場合、グローバルプランナー用スレッドを起動させてグローバルプランが生成されるまで待つ。
- グローバルプラン生成
- サスペンドされていたスレッドがexecuteCb関数内で起動状態に遷移される。
- 選択したグローバルプランナーに応じてmakePlan関数が実行されてる。

プランナーに従って移動する処理

- グローバルプラン生成後
- move_baseの状態をCONTROLLINGに遷移
- グローバルプランナー用スレッドはサスペンド状態に戻す。
- executeCb関数での処理
- ローカルプランを生成する。
- ローカルプランから最適な経路を選択し速度コマンドに変換してロボットへ送信する。
- ゴールに到着したか判定を行う。
- ゴールに到着していない場合はローカルプラン生成からやり直す。
- ゴールに到着した場合、アクションサーバーにゴール到着成功を送信する。

まとめ

- Navigationとは

  →地図ベースで自律移動を行うためのパッケージ群

- move_base

  →navigationパッケージの中心(経路計画)

  →パラメータはロボットや目的に合わせて後述の参考文献を参考に要調整(◡ ‿ ◡)

来週の内容

第四回自律移動勉強会
turtlebotのパッケージ
(実践編の準備)