



## 15. EXCISES

### 15.1 There is any ROS Command?

**Exercise 15.1 — ROS Commands List.** ROSで使用する基本コマンドについて調べ、一覧表にまとめよ。下記の一覧表(15.1)は一例に過ぎない。他にも必要なコマンドがあるので、必ず追加せよ。

Table 15.1: Typical ROS Commands

	Command(使い方)	意味	注意点
1	roscore	ROS コミュニケーションの中核	必ず起動 ( roslaunch 参照 )
2	rosrun package executable		IPやMASTER_URIも変更可
3	rostopic		
4	rosnode		
5	rospack		
6	catkin_create_package		
7	catkin_make		
8	rospack		
9	catkin_init_workspace		groovy以前はrosws
10			
11			

### 15.2 演習Node

ここでは、以下の内容を確認する。

Table 15.2: Nodeの理解の為に、学習する内容

概念	内容
Nodes	他のノードと通信する実行単位
Topics	ノードが発行/購読メッセージ
Messages	ノード間のやりとりをする際のデータ形式

Table 15.2: Nodeの理解の為に、学習する内容(続き)

概念	内容
Master	ノード間の名前解決
rosout	C言語のstdout/stderrに対応
roscore	Master/rosout/parameter server

**Exercise 15.2** 具体的に以下の処理を行い、レポートせよ。

**step 1** roscore

terminal 1: \$ roscore

**step 2** rosnode

terminal 2: \$ rosnode list

**step 3** rosnode info

terminal 2: \$ rosnode info /rosout

**step 4** turtlesimの起動

terminal 3: \$ rosrun turtlesim turtlesim\_node

**step 5** Step 2及び3を再度実行

**step 6** terminal 3上のプログラムをCNTL+Cで停止後、次のコマンド実行

terminal 3: \$ rosrun turtlesim turtlesim\_node \_\_name:=ER

### 15.3 演習Topic

ここでは、topicつまりnodeが公開する情報を観察する手法を習得する。<http://www.ros.org/wiki/ROS/Tutorials/UnderstandingTopics>を参考にしましょう。ここでは、以下の内容を確認する。

Table 15.3: Topicの理解の為に、学習する内容

概念	内容
Nodes	他のノードと通信する実行単位
Topics	ノードが発行/購読メッセージ
rostopic	topic情報の表示
rqt_graph	roscoreに集約される情報の可視化
rosmsg	message情報の可視化
rqt_plot	時間変化の可視化
Messages	ノード間のやりとりをする際のデータ形式
Master	ノード間の名前解決
rosout	C言語のstdout/stderrに対応
roscore	Master/rosout/parameter server

**Exercise 15.3** 具体的に以下の処理を行い、レポートせよ。なお、rqt\_graph等の可視化統合環境である rqt をつかうとリアルタイムでの動きを監査しやすい。

**step 1** roscoreの起動

terminal 1: \$ roscore

**step 2** rosnode list

terminal 2: \$ rosnode list

**step 3** rosnode info

terminal 2: \$ rosnode info /rosout

**step 4** turtlesim

terminal 3: \$ rosrun turtlesim turtlesim\_node

**step 5** Step 2及び3を再度実行

**step 6** teleop

terminal 4: \$ rosrun turtlesim turtle\_teleop\_key

**step 7** Step 2及び3を再度実行

**step 8** rqtによる可視化

terminal 5: \$ rqt

**step 9** rostopic

terminal 6: \$ rostopic echo /turtle1/cmd\_vel

rqt\_graphが随時変化していることを確認する。

**step 10** rostopic

terminal 6: \$ rostopic type /turtle1/cmd\_vel

**step 11** rosmsg

terminal 6: \$ rosmsg show geometry\_msgs/Twist

**step 12** コマンドラインから亀を操作（1回だけ実行する -1 の部分を, -r 10 にすると 10 Hzで連続動作）。なお、“x:”の後ろに空白が入るので注意せよ。

terminal 6: \$ rostopic pub -1 /turtle1/cmd\_vel geometry\_msgs/Twist '{linear:{x: 2.0,y:  
↳ 0.0,z: 0.0},angular:{x: 0.0,y: 0.0,z: 0.0}}'

**step 13** コマンドラインから亀を操作(連続)

terminal 6: \$ rostopic pub -1 /turtle1/cmd\_vel geometry\_msgs/Twist '{linear:{x: 0.2,y:  
↳ 0.0,z: 0.0},angular:{x: 0.0,y: 0.0,z: 0.5}}'

**step 14** トピックの更新間隔の確認

terminal 6: \$ rostopic hz /turtle1/pose

**step 15** rqt\_plot

terminal 6: \$ rqt\_plot /turtle1/pose/x,/turtle1/pose/y /turtle1/pose/theta

**Exercise 15.4** 前述のturtlesimを実行している状態で、以下のシェル・ファイル(plan.sh)を作り、連続でコマンドライン操作する手法を実行せよ

```
1 #!/bin/bash
2 rosservice call /clear
3 #
4 for i in {1..100} ;\
```

```

5 do \
6   echo $i ; \
7   rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist '{linear:{x: 2.0,y: 0.0,z: 0.0}, \
8   ↪ angular:{x: 0.0,y: 0.0,z: 0.0}}' && \
9   rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist '{linear:{x: 0.2,y: 0.0,z: 0.0}, \
9   ↪ angular:{x: 0.0,y: 0.0,z: 0.5}}' ; \
9 done;

```

**step 1** shell file作成

```
$ vi plan.sh
```

**step 2** plan.sh に実行権(executeモード)を設定

```
$ chmod u+x ./plan.sh
```

**step 3** rqtなどにより監視状態の設定

```
$ rqt
```

**step 4** シェルの実行

```
$ ./plan.sh
```

## 15.4 演習Topic2:TalkerとListener

[https://raw.githubusercontent.com/ros/ros\\_tutorials/hydro-devel/roscpp\\_tutorials/talker/talker.cpp](https://raw.githubusercontent.com/ros/ros_tutorials/hydro-devel/roscpp_tutorials/talker/talker.cpp)および、`listener.cpp`を参考にして、以下の内容を確認する。

Table 15.4: 学習する内容

概念	内容
Nodes	他のノードと通信する実行単位
Topics	ノードが発行/購読メッセージ
Messages	ノード間のやりとりをする際のデータ形式
Master	ノード間の名前解決
rosout	C言語のstdout/stderrに対応
roscore	Master/rosout/parameter server

**Exercise 15.5** 具体的に以下の処理を行い、レポートせよ。上述の演習問題を参考にして、詳細に解析すること。

**step 1** rocore

```
terminal 1: $ roscore
```

**step 2** publisher

```
terminal 2: $ rosrun roscpp_tutorials talker
```

**step 3** subscriber

```
terminal 3: $ rosrun roscpp_tutorials listener
```

**step 4** rqt\_graph

## 15.5 To investigate the topics those nodes use to communicate with one another

terminal 4: \$ rqt\_graph

### 15.5 To investigate the topics those nodes use to communicate with one another

**Exercise 15.6** 具体的に以下の処理を行い、レポートせよ。上述の演習問題を参考にして、詳細に解析すること。

```
terminal 1: $ roscore
terminal 2: $ rosrun turtlesim turtlesim_node __name:=A
terminal 3: $ rosrun turtlesim turtlesim_node __name:=B
terminal 4: $ rosrun turtlesim turtle_teleop_key __name:=C
terminal 5: $ rosrun turtlesim turtle_teleop_key __name:=D
```

なお、トラブルが発生したときは、rosrwtコマンドを使うと便利である。

### 15.6 Subscriberをつくれ

**Exercise 15.7** 以下のpublisherに対応するsubscriberプログラムを作成せよ

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3 #include <iostream>
4
5 int main(int argc, char **argv){
6     ros::init(argc, argv, "sample_publisher1");
7     ros::NodeHandle n;
8     ros::Publisher pub = n.advertise<std_msgs::String>("message", 1000);
9     ros::Rate loop_rate(10);
10    while ( ros::ok() ) {
11        std_msgs::String msg;
12        std::stringstream ss;
13        ss << "I am ROBOT!";
14        msg.data = ss.str();
15        pub.publish(msg);
16        ros::spinOnce();
17        loop_rate.sleep();
18    }
19    return 0;
20 }
```

Hint: 次のプログラムの①～③にいれるべきものを考えよう

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3 void messageCallback(const std_msgs::String::ConstPtr& msg){
4     ROS_INFO("I heard: [%s]", msg->data.c_str());
5 }
6 int main(int argc, char **argv){
7     ros::init(argc, argv, "sample_subscriber1");
8     ros::NodeHandle n;
9     ros::Subscriber sub = n.subscribe(①, ②, ③ );
10    ros::spin();
11    return 0;
12 }
```

### 15.7 CallBack関数の理解

ここでは、turtle\_sim\_nodeを操作するプログラムを3種類作成し、CallBack関数の利  
用法を学ぶ。なお、(1)rqtコマンド群を使ってプログラムの動作確認をすること,(2)ヒン  
トを書いておくが、名前は適宜変えること,(3)UNIXのdiffコマンドを使うとプログラム  
の違いを発見しやすい。

#### Exercise 15.8 my\_kameパッケージを作成せよ

```
hint:$catkin_create_pkg my_turtle std_msgs rospy roscpp
```

#### Exercise 15.9 プログラムmove\_kame\_without\_callback.cppを作成し、その動作を確 認せよ。

```

1 #include "ros/ros.h"
2 #include "geometry_msgs/Twist.h"
3 int main(int argc, char **argv) {
4     const double FORWARD_SPEED_MPS = 0.5;
5     // Initialize the node
6     ros::init(argc, argv, "move_turtle");
7     ros::NodeHandle node;
8
9     // A publisher for the movement data
10    ros::Publisher pub = node.advertise<geometry_msgs::Twist>("turtle1/cmd_vel", 10);
11
12    // Drive forward at a given speed. The robot points up the x-axis.
13    // The default constructor will set all commands to 0
14    geometry_msgs::Twist msg;
15    msg.linear.x = FORWARD_SPEED_MPS;
16
17    // Loop at 10Hz, publishing movement commands until we shut down
18    ros::Rate rate(10);
19    ROS_INFO("Starting to move forward");
20    while (ros::ok()) {
21        pub.publish(msg);
22        rate.sleep();
23    }
24    return (0);
25 }
```

CMakeLists.txtファイルの最終行に以下のような行を追加するの忘れない。(名前を変更  
しないと失敗する)

```
add_executable(move_turtle src/move_turtle.cpp)
target_link_libraries(move_turtle ${catkin_LIBRARIES})
```

#### Exercise 15.10 launchファイルmove\_kame.launchを作成し、その動作を確認せよ。ヒ ントとなるlaunchファイルを記す。

```
<launch>
```

```

2 <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node" />
3   <node name="move_turtle" pkg="my_turtle" type="move_turtle" output="screen"/>
4 </launch>
```

**Exercise 15.11** プログラムmove\_kame\_with\_callback.cppを作成し、その動作を確認せよ。

```

1 #include "ros/ros.h"
2 #include "geometry_msgs/Twist.h"
3 #include "turtlesim/Pose.h"
4
5 // Topic messages callback
6 void poseCallback(const turtlesim::PoseConstPtr& msg) {
7     ROS_INFO("x: %.2f, y: %.2f", msg->x, msg->y);
8 }
9
10 int main(int argc, char **argv) {
11     const double FORWARD_SPEED_MPS = 0.5;
12
13     // Initialize the node
14     ros::init(argc, argv, "move_turtle");
15     ros::NodeHandle node;
16
17     // A publisher for the movement data
18     ros::Publisher pub = node.advertise<geometry_msgs::Twist>("turtle1/cmd_vel", 10);
19
20     // A listener for pose
21     ros::Subscriber sub = node.subscribe("turtle1/pose", 10, poseCallback);
22
23     // Drive forward at a given speed. The robot points up the x-axis.
24     // The default constructor will set all commands to 0
25     geometry_msgs::Twist msg;
26     msg.linear.x = FORWARD_SPEED_MPS;
27
28     // Loop at 10Hz, publishing movement commands until we shut down
29     ros::Rate rate(10);
30     ROS_INFO("Starting to move forward");
31     while (ros::ok()) {
32         pub.publish(msg);
33         ros::spinOnce(); // Allow processing of incoming messages
34         rate.sleep();
35     }
36     return(0);
37 }
```

**Exercise 15.12** プログラムmove\_kame\_with\_args.cppを作成し、その動作を確認せよ。

```

1 #include <iostream>
2 #include <string>
3 #include "ros/ros.h"
4 #include "geometry_msgs/Twist.h"
5 #include "turtlesim/Pose.h"
```

```

6 // Topic messages callback
7 void poseCallback(const turtlesim::PoseConstPtr& msg) {
8     ROS_INFO("x: %.2f, y: %.2f", msg->x, msg->y);
9 }
10 }
11 int main(int argc, char **argv) {
12     const double FORWARD_SPEED_MPS = 0.5;
13     std::string robot_name = std::string(argv[1]);
14
15     // Initialize the node
16     ros::init(argc, argv, "move_turtle");
17     ros::NodeHandle node;
18
19     // A publisher for the movement data
20     ros::Publisher pub = node.advertise<geometry_msgs::Twist>(robot_name + "/cmd_vel",
21         ↪ 10);
22
23     // A listener for pose
24     ros::Subscriber sub = node.subscribe(robot_name + "/pose", 10, poseCallback);
25
26     geometry_msgs::Twist msg;
27     msg.linear.x = FORWARD_SPEED_MPS;
28
29     ros::Rate rate(10);
30     ROS_INFO("Starting to move forward");
31     while (ros::ok()) {
32         pub.publish(msg);
33         ros::spinOnce(); // Allow processing of incoming messages
34         rate.sleep();
35     }
36     return(0);
37 }
```

**Exercise 15.13** launchファイルにargsパラメータを付けて、プログラムmove\_kame\_with\_args.cppを起動せよ。ヒントのlaunchファイルを下記に記す。

```

1 <launch>
2     <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node" />
3     <node name="move_turtle" pkg="my_turtle" type="move_turtle" args="turtle1" output=""
4         ↪ screen"/>
5 </launch>
```

**Exercise 15.14** turtleを1m直進させ、45°反時計回りに回転させ、停止するプログラムを作成せよ。なお、初期位置と終了位置を標準出力に出力せよ

## 複数robot制御

ROSの複数ロボット・ノード間制御技術を学ぶ。具体的には、以下の3つの手法について考える。

手法1. 同じdriverを使った複数シミュレータの制御

手法2. 2つのドライバ実体による複数シミュレータ制御

手法3. 遠隔操作network構築

## 15.8 同じdriverを用いた2つのシミュレータの同時制御

ここでは、同じdriverを用いた2つのシミュレータの同時制御法を学ぶ。具体的には、次の3ステップで、2つのシミュレータを立ち上げ、第4ステップで制御命令を送る手順を学ぶ。

**Step 1:** まず、1つ目のターミナルで、roscoreを立ち上げる。具体的なコマンド：

```
terminal 1:$ roscore
```

**Step 2:** 次に、2つ目のターミナルで、最初のturtle simulatorノードを立ち上げる。

```
terminal 2:$ rosrun turtlesim turtle_sim
```

**Step 3:** 最後に、3つ目のターミナルで、“\_\_name”パラメータを付けて、デフォルト・ノード名とは別の名前で、2番目のturtle simulatorノードを立ち上げる。ただし、(1) アンダースコア(\_)は、2つであること、(2) PASCAL言語風に“:=“記号を使って名前を代入していることに注意する。実行例：

```
terminal 3:$ rosrun turtlesim turtlesim_node __name:=turtlesim2
```

更に、「同じdriverを使って、2つのシミュレータを制御する」手法を学ぶ。

**Step 4:** 4つ目のターミナル上で、次の制御ノードを立ち上げ、矢印キーで2つのシミュレータが同時に制御する。具体的なコマンド：

```
terminal 4:$ rosrun turtlesim turtle_teleop_key
```

**Exercise 15.15** Step 3で、“\_\_name”パラメータを付けずに、rosrunコマンド実行するとどうなるか?出力メッセージだけでなく、理由も明確にせよ。

ヒント1：roscoreの動きに注意する。

ヒント2：“.log”ディレクトリに細かい記録が残っている。

**Exercise 15.16** 上記の3つのステップの前後で、rostopicコマンドを用いて、topicの状態を確認し、各topicの意味を調べよ。

ヒント：5つめのターミナル上で、コマンドを実行しましょう。

```
terminal 5:$ rostopic list
```

この他にも、調査に必要なコマンドを探してみよう。

**Exercise 15.17** rqt-graphを使って、ノード間コミュニケーションがどのようにになっているのかを調べよ。

ヒント：どのステップの状態を確認しているのか注意し、グラフの変化を読み取る。

**Exercise 15.18 — 発展課題.** 後述するlaunchファイルを使って、本節の内容を実行するにはどうしたらよいかを具体的に述べよ。

**Exercise 15.19 — Report課題.** 以上を踏まえ、本節の実習を行い、画像付きでレポートせよ。なお、レポートのタイトルは、“Two Turtle Simulators via Same Driver Instance”とせよ。

### 15.9 2つのドライバ実体を用いた2つのシミュレータ制御

本節では、[http://docs.ros.org/indigo/api/turtlesim/html/draw\\_square\\_cpp\\_source.html](http://docs.ros.org/indigo/api/turtlesim/html/draw_square_cpp_source.html)にある“draw\_square.cpp”を利用して、“draw\_square”ドライバの2実体によって動かされる2つのturtleシミュレータがもつnamespace概念を身に着けることを学ぶ。

2つの異なるnamespaceにおいて、“turtle\_sim”的2実体を走らせるlaunchファイルを使うことを考える。つまり、それぞれのturtle\_simシミュレータにおいて、ノードは異なる名前と異なるtopic名を持っていることを確認しておく必要がある。イメージ図を図15.9に示しておく。

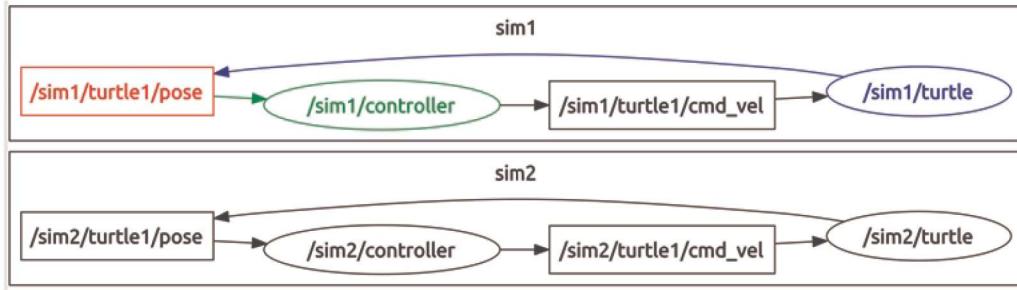


Figure 15.1: rqt-graphによるノード間接続表示例

まず、下準備として以下の処理を行う。新しい実習プロジェクトの作成は記載していない。

1. draw\_square.cppを準備する。

```
terminal 1:$ vi draw_square.cpp
```

で空のファイルを準備する。次に、ブラウザを使って[http://docs.ros.org/indigo/api/turtlesim/html/draw\\_square\\_8cpp\\_source.html](http://docs.ros.org/indigo/api/turtlesim/html/draw_square_8cpp_source.html)よりソースコードを打ち込むのが、正しいやり方だがあえて、vimエディタの練習として、マウスでソースコードをクリップする。最後に、viで開いたファイルにペーストする。

2. この状態では、文頭に5ヶタの連番と1つの空白が邪魔である。viの編集機能を使って、すべての行から文頭6文字を削除する。具体的には以下の処理をする。

**vi 1:** 第1行目1桁目にカーソルを移動

**vi 2:** cntl + V

**vi 3:** shift+g (大文字のG)で文末に移動

**vi 4:** 右矢印キーを押して6桁分右に移動

**vi 5:** d キーを押す

**註 1:** 失敗しても落ち着いてu すると操作1つ分前の状態に戻る

**vi 6:** 確認したら:w :q でファイルを保存終了する

<http://code.ros.org/>は、結構接続に時間がかかり、接続失敗も多いので、[http://code.metager.de/source/xref/ros/tutorials/turtlesim/tutorials/draw\\_square.cpp](http://code.metager.de/source/xref/ros/tutorials/turtlesim/tutorials/draw_square.cpp)などを利用するのもよい。

3. 今回はBoostライブラリを利用するので、CMakelist.txtにboostの設定

```
terminal 1:$ vi CMakelist.txt
```

によりファイルを開き、Boostパッケージを追加する。

```
find_package(Boost REQUIRED COMPONENTS thread)
```

viを保存終了する。

今回は、`lanuch`ファイルを使って操作する。ファイル名を`multi2.launch`とする。

#### 4. vi `multi2.launch`

```

1 <launch>
2   <group ns="sim1">
3     <node name="turtle" pkg="turtlesim" type="turtlesim_node"/>
4     <node name="controller" pkg="turtlesim" type="draw_square" />
5   </group>
6   <group ns="sim2">
7     <node name="turtle" pkg="turtlesim" type="turtlesim_node"/>
8     <node name="controller" pkg="turtlesim" type="draw_square" />
9   </group>
10 </launch>

```

2行目及び4行目の`group`タグのパラメータ`ns`がnamespace用の変数である。この値が異なっていれば、別々のnamespaceでシミュレータが動く。

次に、`remap`について考えよう。`remap`とは、コードを書く際に一旦決定した`name`を実行時に如何様にでも変えることのできる機能である。複数人で開発を進めていくロボットの世界では、似たような名前を付けて開発することは多々あり、詳細な名前付けまでを開発の最初から決めておくことはできない。そこで、オブジェクト指向もしくはメタモルフェイックなプログラミング設計思想をうまく利用している。サブ・パラメータ`from`や`to`から容易に想像できる様に情報の受け渡しを行える。これと同様の設計思想は、1970年代のシミュレーションの世界ではごく普通に行われていたし、通信制御の世界でも行われていた俗にいう「枯れた技術」である。

実感してもらうために、`turtle_teleop_key`を使って、操作する`launch`ファイルを書いてみよう

#### 5. vi `MultiTeleop.launch`

```

1 <launch>
2   <group ns="sim1">
3     <node name="turtle" pkg="turtlesim" type="turtlesim_node"/>
4   </group>
5   <group ns="sim2">
6     <node name="turtle" pkg="turtlesim" type="turtlesim_node"/>
7   </group>
8   <node pkg="turtlesim" name="teleop" type="turtle_teleop_key" output="screen">
9     <remap from="/turtle1/cmd_vel" to="/sim1/turtle1/cmd_vel"/>
10    </node>
11 </launch>

```

この`remap`操作をコマンドライン上で行うには、以下のようにすればよい。

```
$ rosrun turtlesim turtle_teleop_key /turtle1/cmd_vel:=/sim1/turtle1/cmd_vel
```

**Exercise 15.20 — `rostopic`でのチェック。** `rostopic`コマンドを使って動作を確かめよ。 ■

**Exercise 15.21 — `rosnode`でのチェック。** `rosnode`コマンドを使って動作を確かめよ。 ■

**Exercise 15.22 — `rqt-graph`でのチェック。** `rqt-graph`コマンドを使って動作を確かめよ。 ■

**Exercise 15.23** 前節と同様に`launch`ファイルを使わずに実習せよ。 ■

**Exercise 15.24 — Report**課題。以上を踏まえ、本節の実習を行い、画像付きでレポートせよ。なお、レポートのタイトルは、“Two Turtle Simulators with Two Driver Instances”とせよ。

## 15.10 遠隔操作networkの構築

通信の世界でよく知られたMaster-Slaveモデルを考える。仮に、Master側にturtleBotのようなロボットを配置し、Slave側に操作する人間側のコンピュータがあると考えれば、図15.10に示すような処理の流れを作ることでロボットを遠隔操作するシステムの構築ができる。

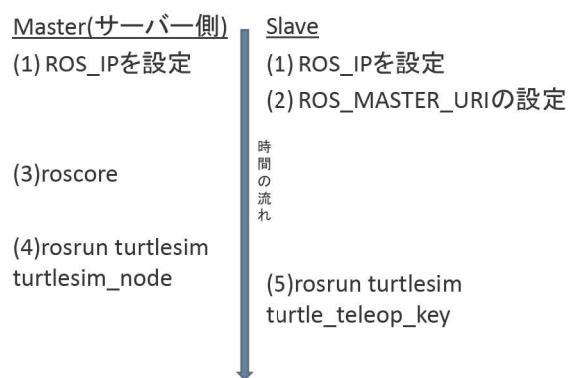


Figure 15.2: Master & Slave Modelを模した遠隔ロボット操作系の処理フロー

具体的な処理フローを考える。ロボットを遠隔操作するためには、通信関係の情報取得が必要不可欠である。

### network 1: ifconfig によるネットワーク情報の取得

```
$ ifconfig -a
```

図15.10は、その実行例である。eth0は、ethernet通信装置0番であり、wlan0はwireless LAN通信装置0番である。今、下線部が自分のコンピュータのIPアドレスである。

**network 2:ROS\_IP:** IPアドレスを~.bashrcにROSのネットワーク情報として書き込んでおく。

```
$ echo "export ROS_IP=10.0.1.6" >> ~/.bashrc
```

**network 3:ROS\_MASTER\_URI** TurtleBot役のコンピュータのIPアドレスを~.bashrcにROSのネットワーク情報として書き込んでおく。

```
$ echo "export _ROS_MASTER_URI=http://10.0.1.7:11311" >> ~/.bashrc  
$ source ~/.bashrc
```

これ以降、実習中に次々と開くすべてのterminalにROSのネットワーク情報を伝えておくことが出来る。ただし、実習が終わり、一旦ネットワークから離脱すると、別のIPアドレスとなるので注意する。

**network 4:ping:** ping コマンドを使ってネットワークを介してお互いに接続できるかを確認する。

```
slave machine: $ ping $ROS_MASTER_URI
```

```
eth0 Link encap:イーサネットハードウェアアドレス 28:d2:44:48:de:74
      UP BROADCAST MULTICAST MTU:1500 メトリック:1
      RXパケット:0 エラー:0 損失:0 オーバラン:0 フレーム:0
                  TXパケット:0 エラー:0 損失:0 オーバラン:0 キャリア:0
                  衝突(Collisions):0 TXキュー長:1000
      RXバイト:0 (0.0 B) TXバイト:0 (0.0 B)
      割り込み:20 メモリ:f1600000-f1620000

lo Link encap:ローカルループバック
      inetアドレス:127.0.0.1 マスク:255.0.0.0
      inet6アドレス: ::1/128 範囲:ホスト
      UP LOOPBACK RUNNING MTU:65536 メトリック:1
      RXパケット:21096996 エラー:0 損失:0 オーバラン:0 フレーム:0
      TXパケット:21096996 エラー:0 損失:0 オーバラン:0 キャリア:0
      衝突(Collisions):0 TXキュー長:0
      RXバイト:419553401459 (419.5 GB) TXバイト:419553401459 (419.5 GB)

wlan0 Link encap:イーサネットハードウェアアドレス dc:fb:02:58:d5:34
      inetアドレス:10.0.1.6 ブロードキャスト:10.0.1.255 マスク:255.255.255.0
      inet6アドレス: fe80::defb:2ff:fe58:d534/64 範囲:リンク
      UP BROADCAST RUNNING MULTICAST MTU:1500 メトリック:1
      RXパケット:53488 エラー:0 損失:0 オーバラン:0 フレーム:0
      TXパケット:46834 エラー:0 損失:0 オーバラン:0 キャリア:0
      衝突(Collisions):0 TXキュー長:1000
      RXバイト:39587301 (39.5 MB) TXバイト:6987130 (6.9 MB)
```

Figure 15.3: ifconfigの出力例：wlan0の下線部に注目

```
master machine: $ ping [IP address on slave machine]
```

途中ルーターなどを介してマシン同士が通信しようとするとルーター設定により禁止されていることがあるので注意しなければならない。

### 15.11 ROS Stage Simulator

ここでは、[http://wiki.ros.org/simulator\\_stage](http://wiki.ros.org/simulator_stage)を参考にして、Stage Simulatorの使い方を学ぶ。そして、レーザーセンサーの取り扱いを学ぶ。

注意事項 `ROS_MASTER_IP`や`roscore`の起動など基本的なことは詳述していない。

**Step 1** `stage_ros`パッケージの起動確認（バッククオート記号:`SHIFT`+`@`に注意、シングルクオート:`SHIFT`+`7`ではありません）

```
$ rosrun stage_ros stageros 'rospack find stage_ros' /world/willow-erratic.world
```

**Step 2** Stage Simulator画面が出たのちに、`R`キーを押すと俯瞰モードが変化する。マウスを使って、赤と青の立方体を探す。

**Step 3** Stage window内で`D`(メニューの`View`→`Data`)でレーザースキャナーの範囲を確認できる

**Step 4** 別のターミナルで、実際のworldファイルを確認する

```
$ rosed stage_ros/world
$ pwd
$ ls -alF
```

**Exercise 15.25** worldファイルの記述規則(syntactic)を確認する。 ■

**Step 5** teleop用パッケージのインストール（Indigoのひとは、hydroの部分をindigoに変更）

```
$ sudo apt-get install ros-hydro-teleop-twist-keyboard
```

**Step 6** 別のターミナルでteleop起動

```
$rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

**Exercise 15.26** rqtやrostopic等を用いて、動作を確認せよ。 ■

```
$rostopic echo -n 1
$rosmsg show sensor_msgs/LaserScan
```

#### 15.11.1 Stopper ノードの作成実行

障害物(obstacle)を発見すると停止するStopperノードを作成する。具体的には、`Stopper.h`, `Stopper.cpp`, `runStopper.cpp`の3つを作成し、実行する。

**Step 1** 新しく`my_stage`パッケージを作成

```
$ catkin_create_pkg my_stage std_msgs roscpp rospy
```

**Step 2** `Stopper.h`, `Stopper.cpp`, `runStopper.cpp`の3つを作成(ソース例を示すが、`include`の追加、デストラクタ`~`付きなどコンパイルに必要な事項は各自で確認すること)

```

1 #include "ros/ros.h"
2 #include "sensor_msgs/LaserScan.h"
3
4 class Stopper {
5 public:
6     // Tunable parameters
7     const static double FORWARD_SPEED_MPS = 0.5;
8     const static double MIN_SCAN_ANGLE_RAD = -30.0/180*M_PI;
9     const static double MAX_SCAN_ANGLE_RAD = +30.0/180*M_PI;
10    const static float MIN_PROXIMITY_RANGE_M = 0.5; // Should be smaller than
11        ↪ sensor_msgs::LaserScan::range_max
12
13    Stopper();
14    void startMoving();
15
16 private:
17    ros::NodeHandle node;
18    ros::Publisher commandPub; // Publisher to the robot's velocity command topic
19    ros::Subscriber laserSub; // Subscriber to the robot's laser scan topic
20    bool keepMoving; // Indicates whether the robot should continue moving
21
22    void moveForward();
23    void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan);
24};

```

Program 15.1: Stopper.hの例

```

1 #include "Stopper.h"
2 #include "geometry_msgs/Twist.h"
3
4 Stopper::Stopper()
5 {
6     keepMoving = true;
7
8     // Advertise a new publisher for the simulated robot's velocity command topic
9     commandPub = node.advertise<geometry_msgs::Twist>("cmd_vel", 10);
10
11    // Subscribe to the simulated robot's laser scan topic
12    laserSub = node.subscribe("base_scan", 1, &Stopper::scanCallback, this);
13}
14
15 // Send a velocity command
16 void Stopper::moveForward() {
17     geometry_msgs::Twist msg; // The default constructor will set all commands to 0
18     msg.linear.x = FORWARD_SPEED_MPS;
19     commandPub.publish(msg);
20}
21 // Process the incoming laser scan message
22 void Stopper::scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan)
23 {
24     // Find the closest range between the defined minimum and maximum angles
25     int minIndex = ceil((MIN_SCAN_ANGLE_RAD - scan->angle_min) / scan->
26         ↪ angle_increment);
26     int maxIndex = floor((MAX_SCAN_ANGLE_RAD - scan->angle_min) / scan->
27         ↪ angle_increment);
28
29     float closestRange = scan->ranges[minIndex];
30     for (int currIndex = minIndex + 1; currIndex <= maxIndex; currIndex++) {
31         if (scan->ranges[currIndex] < closestRange) {
32             closestRange = scan->ranges[currIndex];
33         }
34     }
35 }

```

```

32     }
33 }
34
35 ROS_INFO_STREAM("Closest_range:_" << closestRange);
36
37 if (closestRange < MIN_PROXIMITY_RANGE_M) {
38     ROS_INFO("Stop!");
39     keepMoving = false;
40 }
41
42 void Stopper::startMoving()
43 {
44     ros::Rate rate(10);
45     ROS_INFO("Start_moving");
46
47 // Keep spinning loop until user presses Ctrl+C or the robot got too close to an obstacle
48 while (ros::ok() && keepMoving) {
49     moveForward();
50     ros::spinOnce(); // Need to call this function often to allow ROS to process incoming
51     // messages
52     rate.sleep();
53 }

```

Program 15.2: Stopper.cppの例

```

1 #include "Stopper.h"
2
3 int main(int argc, char **argv) {
4     // Initiate new ROS node named "stopper"
5     ros::init(argc, argv, "stopper");
6
7     // Create new stopper object
8     Stopper stopper;
9
10    // Start the movement
11    stopper.startMoving();
12
13    return 0;
14 };

```

Program 15.3: runStopper.cppの例

**Step 3** CMakelists.txtをPrg. 15.4を参考に修正せよ

```

1 cmake_minimum_required(VERSION 2.8.3)
2 project(my_stage)途中省略
3
4
5
6 ## Declare a cpp executable
7 add_executable(stopper src/Stopper.cpp src/run_stopper.cpp)
8
9 ## Specify libraries to link a library or executable target against
10 target_link_libraries(stopper ${catkin_LIBRARIES})

```

Program 15.4: CMakelists.txtの改造例

**Exercise 15.27** launchファイルを作成し、実行分析せよ

```

1 <launch>
2   <node name="stage" pkg="stage_ros" type="stageros" args="$(find_stage_ros)/world/
  ↪ willow-erratic.world"/>
3   <node name="stopper" pkg="my_stage" type="stopper" output="screen"/>
4 </launch>
```

```
1 $ roslaunch my_stage my_stage.launch
```

**Exercise 15.28** Extra credit: Add a parameter in the roslaunch file that specifies how close your robot gets to obstacles before it starts rotating.**15.12 Mapping**

地図(Map)を学習することは、移動体ロボットを理解する上で大変重要な要素である。Mapを利用することにより、localization, path planning, activity planningといったタスクを効率よくロボットに実行させることができる。移動体空間表現の代表例として、Grid maps, Geometric maps, Voronoi graphs, Quadtreesなどある。これらの詳細なアルゴリズムについては2年生秋のロボットプログラミングIIで学修する。

一例として、Occupancy Grid Maps(占有格子地図法)について考える。実体環境を格子(Cell)間隔5cm~50cmの等間隔に分割して、その中の平均値で観測量を表す。平均観測量を0%~100%の確率値に置き換えておく。この確率値の空間分布を確率密度関数と呼ぶ。一番簡単な実装として、障害物がある格子を1、ロボットと障害物を結ぶ線上的格子には0を割り振る。なお障害物の奥側のようにセンサーが届かなかない場所の確率密度は評価のしようがない。プログラミングの都合上、未知量を数値化する際に-1を割り当てておくことが多い。

この手法の利点は、簡単に表現でき、処理速度も速いことである。一方、欠点として、不正確になりやすく、無駄にメモリを使いがちになる。例えば、1つの格子枠内にちょっとでも物体が侵入するとその格子範囲内全体に確率密度が割り振られることになる。

最近は、Simultaneous localization and mapping (SLAM, 同時自己位置推定)が脚光を浴びている。未知の環境において、ロボット自体によって環境計測を行い障害物と自己位置を同時に推定する手法である。いわば「鶏が先か卵が先か」問題を引き起こすので、研究テーマとしても大変興味深い。物理学の非平衡統計力学の知見である確率密度分布関数の工学応用として、例えば、particle filterと呼ばれる処理をこのSLAMに適用する研究(fastSLAM)などがある。

この手法は、まず、環境を格子状に分けた状態空間とみなし、各状態空間を占有する粒子集団を確率密度関数として表現する。先ほどは一つ一つの格子内にちょっとでも物体が入れば1、なければ0の極端な処理をしていたが、ここでは、何%その格子内に物体が居るのかを示す。この意味で、確率密度と呼ぶ。後は、(1)異なる速度(向きと速さ)を持つ粒子をちょっとずつ移動させるシミュレーションを行い、(2)そのシミュレーション結果と実際の計測結果とを比較し、(3)もっともらしくその格子に存在する確率を推定し、(4)初期の分布からシミュレーションにより得られた尤もらしい期待値を新たな粒子分布と定めることを繰り返す。

ROSには、GMapping(<http://wiki.ros.org/gmapping>)と呼ばれるlaser-based SLAMが準備されている。そのノード名は、slam\_gmappingである。例として<https://www.youtube.com/watch?v=khSrWtBOXik>をみるとよい。

**Step 1** gmappingパッケージのインストール（Indigoのひとは、hydroの部分をindigoに変更）

```
$ sudo apt-get install ros-hydro-slam-gmapping
```

**Step 2** roscore起動

**Step 3** Stage simulatorおよびteleopを起動( 15.11節参照)

**Step 4** 新しいターミナルでgmappingパッケージを起動

```
$rosrun gmapping slam_gmapping scan:=base_scan
```

**Step 5** topic /mapにpublishされており、メッセージはnav\_msgs/OccupancyGridにある。 dum出力する（画面に出し続ける）には

```
$rostopic echo /map -n1
```

とする。

つづけて、map\_serverを使って、地図(map)の保存や読み込みをする。

**Step 1** map serverのインストール

```
$ sudo apt-get install ros-hydro-map-server
```

**Step 2** 動的に生成されるmapの保存( saver と serverをまちがえないように)

```
$ rosrun map_server map_saver [-f mapname]
```

なお、コマンド実行したディレクトリにmap.pgmとmap.yamlが生成される。中身を確認する。

**Exercise 15.29** Ubuntuでは、eogを使うとpgmファイルを画面表示できる。実行せよ。

```
$ eog map.pgm
```

YAML(Yet Another Markup LanguageまたはYAML Ain't Markup Language<sup>1</sup>)ファイルは、XMLファイルと違い、データフォーマット仕様を定める記述方式である。インデントにより階層的なデータ構造を示すことができる。ただしTABではなくスペース文字を使ってインデントするので留意する。

**Exercise 15.30** map.yamlファイルの中身を確認せよ。

```
$ vi map.yaml
```

YAMLファイルの例として、以下のような内容が記述されている。

```
image: map.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

**Exercise 15.31** rqtなどのツールを使って、node graph等の状態を確認せよ

<sup>1</sup>ain'tとは、am not,are notなどの非標準短縮形で、歌詞などに現れることが多い

rvizパッケージを使って、動的変化（teleopしながらどのようなMAPを生成していくのか）をとらえる3D可視化作業をおこなう。

**step 1** rvizの起動(初期起動画面(図15.12参照)

```
$ rosrun rviz rviz
```

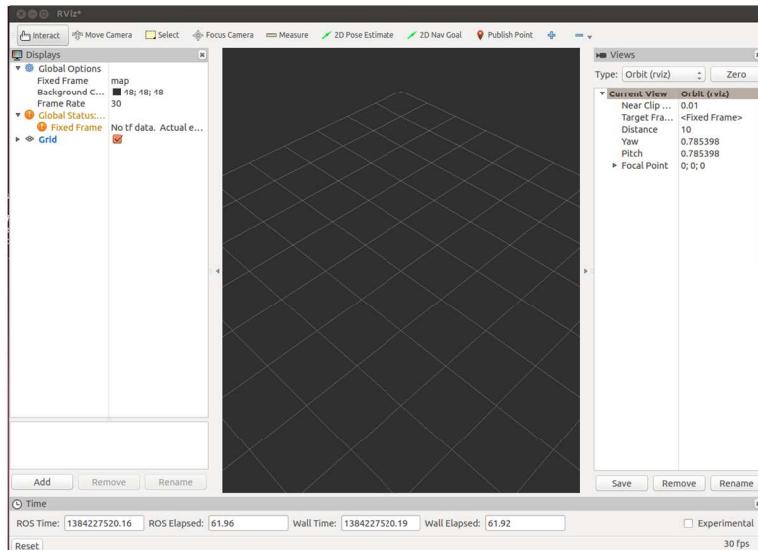


Figure 15.4: rviz初期起動画面

なお、マウス操作として、右クリックしたままで動かすことでzoom in/outができる、左クリックしたままで動かすことでpanningできる。

rvizの初期起動画面では、左からDisplay, 3D画像, Viewsの3画面になっている。Displayパネルに注目すると、Global Options, Global Status, Gridsの項目がある。Displayパネル下にあるAddボタンを使って、LaserScan, TF, Map項目の追加とパラメータ調整を行う。

**Exercise 15.32** Addボタンを押すと、By Display TypeとBy Topicの項目がある。By Display Type内の各小項目について、何を意味するのか一覧表を作成せよ。 ■

**step 2** Display→Addの後、By Display Type内の小項目LaserScanを選び、By Topics内の小項目/base\_scanを選択し、追加する。

**step 3** Global OptionsのFixed Frameの値をbase\_linkに変更

**step 4** Display→Addの後、By Display Type内の小項目TFを選択し、追加する。

**step 5** Display→Addの後、By Display Type内の小項目Mapを選び、By Topics内の小項目/mapを選択し、追加する。

**Exercise 15.33** teleopをつかってturtleを動かすと、rviz画面がどのようになるか確認せよ ■

**Exercise 15.34** 既に準備されているconfigurationファイルを読み込んでrvizを起動。なお、バッククオート記号: [SHIFT] + [@] に注意、シングルクオート: [SHIFT] + [7]ではありません)

```
$ rosrun rviz rviz -d 'rospack find stage_ros'/rviz/stage.rviz
```

**Exercise 15.35** launchファイルを作成し実行せよ

```

1 <launch>
2   <node name="stage" pkg="stage_ros" type="stageros" args="$(find_stage_ros)/world/
3     ↪ willow-erratic.world"/>
4   <node name="gmapping" pkg="gmapping" type="slam_gmapping">
5     <param name="scan" value="base_scan"/>
6   </node>
7   <node name="rviz" pkg="rviz" type="rviz" args="$(find_stage_ros)/rviz/stage.rviz"/>

```

**Exercise 15.36** 地図ファイル(.pgm)をあなたの/mapディレクトリにコピーしたのちに、Map serverノードを起動せよ

```

1 <launch>
2   <arg name="map_file" default="$(find_my_package)/maps/willow-full-0.05.pgm"/>
3
4   <!-- Run the map server -->
5   <node name="map_server" pkg="map_server" type="map_server" args="$(arg_map_file)
6     ↪ _0.05" />

```

### 15.13 ROS Service

ここでは、あなたのROSノードに地図を読み込む手法について学ぶ。`map_server`パッケージから`static_map`と呼ばれるROSサービスを使う。これまで学んできた`publish` / `subscribe`モデルの代わりに、`request` / `reply`パラダイムについて学ぶ。

ROS サービスは、要求メッセージ(`request`)と返答メッセージ(`reply`)を含む`[srv]`ファイルによって定義される。roscppは、これら`[srv]`ファイルを C++のソースコードに変換し、3つのクラスを生成する。例えば、`my_package/srv/Foo.srv`は、表15.5に示すように変換される。

Table 15.5: srvファイルの変換

型	意味
<code>my_package::Foo</code>	service definition
<code>my_package::Foo::Request</code>	request message
<code>my_package::Foo::Response</code>	response message

**Exercise 15.37** Staic Mapによるデータ処理サンプルを以下に示す。launchファイルなど起動分析にかかる一連の処理を行え

```

1 #include <ros/ros.h>
2 #include <nav_msgs/GetMap.h>
3 #include <vector>
```

```
4
5 using namespace std;
6
7 // grid map
8 int rows;
9 int cols;
10 double mapResolution;
11 vector<vector<bool>> grid;
12
13 bool requestMap(ros::NodeHandle &nh);
14 void readMap(const nav_msgs::OccupancyGrid& msg);
15 void printGrid();
16
17 int main(int argc, char** argv){
18     ros::init(argc, argv, "load_ogm");
19     ros::NodeHandle nh;
20
21     if ( !requestMap(nh) ) exit(-1);
22
23     printGrid();
24     return 0;
25 }
26
27 bool requestMap(ros::NodeHandle &nh){
28     nav_msgs::GetMap::Request req;
29     nav_msgs::GetMap::Response res;
30
31     while ( !ros::service::waitForService("static_map", ros::Duration(3.0)) ) {
32         ROS_INFO("Waiting for service static_map to become available");
33     }
34
35     ROS_INFO("Requesting the map...");
36     ros::ServiceClient mapClient = nh.serviceClient<nav_msgs::GetMap>("static_map");
37
38     if (mapClient.call(req, res)) {
39         readMap(res.map);
40         return true;
41     } else {
42         ROS_ERROR("Failed to call map service");
43         return false;
44     }
45 }
46 void readMap(const nav_msgs::OccupancyGrid& map){
47     ROS_INFO("Received a %d X %d map @ %.3f m/px\n"
48             ,map.info.width, map.info.height, map.info.resolution);
49     rows = map.info.height;
50     cols = map.info.width;
51     mapResolution = map.info.resolution;
52
53     // Dynamically resize the grid
54     grid.resize(rows);
55     for (int i = 0; i < rows; i++) {
56         grid[i].resize(cols);
57     }
58     int currCell = 0;
59     for (int i = 0; i < rows; i++) {
60         for (int j = 0; j < cols; j++) {
61             if (map.data[currCell] == 0) // unoccupied cell
62                 grid[i][j] = false;
```

```
63     else
64         grid[i][j] = true; // occupied (100) or unknown cell (-1)
65         currCell++;
66     }
67 }
68
69 void printGrid(){
70     printf("Grid_map:\n");
71     int freeCells = 0;
72     for (int i = 0; i < rows; i++) {
73         printf("Row_no.%d\n", i);
74         for (int j = 0; j < cols; j++) {
75             printf("%d ", grid[i][j] ? 1 : 0);
76         }
77         printf("\n");
78     }
79 }
```

Program 15.5: StaicMap.cppの例



## 15.14 Navigation Stack

<http://wiki.ros.org/navigation>を参考にして、Navigation Stackを体験する。Navigation Stackとは、クラッシュしたり行方不明にならずに、ある位置から目的地までロボットを移動させる技術である。odometry(走行位置)とセンサから得られた情報をつかって、安全な移動速度を割出し、ゴールまで導く。動画として<http://www.youtube.com/watch?v=qziUJcUDfBc>をみるとよくわかるだろう。

ここでは、以下の5項目について学ぶ。

- 1 ROS navigation stack
- 2 Navigation planners
- 3 Costmaps
- 4 Running ROS navigation with Stage and rviz
- 5 Sending goals to robots

具体的には、

- 1 map\_server
- 2 gmapping
- 3 amcl
- 4 global\_planner
- 5 local\_planner
- 6 move\_base

のパッケージを使う。

Navigation Stackの基本ステップは、(1)移動先の決定(2)AMCL処理(3)経路策定(4)move\_base(5)cmd\_velとodomトピック経由で(6)Base controllerを動かす6項目からなる。