2015 年度 卒業論文

自律ロボット実現に向けた ROS による環境情報の取得

奈良女子大学 理学部 情報科学科 4 回生 新出研究室
12251591 柿原怜奈

平成 28 年 2 月

目 次

概 要

近年、動的に変化する環境下においても、自律的に目的を達成するロボットの実現が望まれている。これを実現するにあたり、ロボットに視覚情報を取り入れることが求められる。我々は、ロボット用のソフトウェアフレームワークである ROS を使用して、RGB-D カメラからの画像取得を行い、これを用いたロボットの行動制御の実装を行った。本論文ではそのうち ROS によるカメラからの画像取得の部分について述べる。

1 はじめに

1.1 研究背景

近年、自律的に目的を達成するロボットの実現が求められている。このようなロボットは目標達成の手段の 1 つとして「目的の場所に移動する」という能力を要することが多いと考えられる。これを実現するには、ロボットが周りの環境を把握するために視覚情報を取り入れる必要がある。また、一般に目標達成のためには、ロボットの多数の機能を制御する必要がある。そのためには、ロボット用のソフトウェアフレームワークを使用するのが適しており、その一つに Robot Operating System(以下 ROS と記載)[1] というものがある。よって本研究では、ROS を使ったカメラからの画像取得とそれを用いた目的の場所に移動するロボットの開発を目指した。なお本研究は、奈良女子大学理学部 4 回生兼松との共同研究である。本論文では ROS を使ったカメラからの画像取得について述べる。目的の場所に移動するロボットの開発については [2] で述べられている。

1.2 Xtion

本研究では、画像取得にあたり、カメラとして Xtion PRO LIVE[3](以下 Xtion と記載) を使用した。これは、 RGB センサーと深度センサーを搭載した物体の動きをデータ化する PC 用モーションキャプチャデバイスである。Xtion を使用するために特別な機器を用意する必要はなく、PC に USB 2.0 ケーブルで接続すれば使い始めることができる。付属の DVD には Linux 用ドライバが入っており、Linux 環境で使うことができる。さらに、バスパワーに対応しているため、電源ケーブルは必要がなく手軽にモーションキャプチャを行うことができる。

図 1: Xtion

2 ROS

2.1 概要

ROS とは世界中で広く使用されているロボット用ソフトウェアフレームワークである。ロボット作成のソフトウェア開発を支援するライブラリとツールをオープンソースで提供している。本研究では ROS を動作させるプラットフォームとしては公式サポートされている Ubuntu を選択した。よって、商用の環境に依存せずに、ROS を使用した開発が可能である。ROS を使う利点としては、複数のノードを組み合わせてひとつの大きなプログラムを作るため、ロボットの機能同士を結合させやすく、さらにプログラムの再利用性が高い。また、OpenCV などの画像処理ライブラリと連携しやすい。そして、ロボット開発において最も重要なソフトウェアの状態を可視化する強力なツールを備えていることなどが挙げられる。プログラミング言語はプログラムの実装やメモリ管理が容易な Python を選択した。

2.2 ROS の基本

- パッケージ

  ROS でプログラムを動かす上での最小単位。ROS のプログラムを動かすには、基本的には「rosrun パッケージ ファイル名」とする。ROS のプログラムを起動するとノードができる。

- ノード (Node)

  ROS パッケージ内の実行ファイル (プロセス)。また、ROS におけるソフトウェアの 1 つの単位であり、ノードを組み合わせてシステムをつくることで、プログラムの再利用性が高まり、ノード単位でのデバックが可能になる。

- launch ファイル

  複数のノードをまとめて実行するには、launch ファイル (ファイル名.launch) という、実行するノードをまとめたファイルを作成する。roslaunch コマンドで「roslaunch パッケージ ファイル名.launch」を実行する。

2.2.1 通信方法

ROS の通信手段の一つとしてトピックによってノード間の連携を行う方法 (図 2) がある。この通信では一方向のメッセージの配信または購読を行う。非同期通信であるため、必要に応じてデータの送受信が可能である。また、一度の接続で継続的にメッセージの送受信が可能である。ここで通信を行うノードには、トピックを配信する Publisher ノードと、トピックを購読する Subscriber ノードの 2 種類がある。

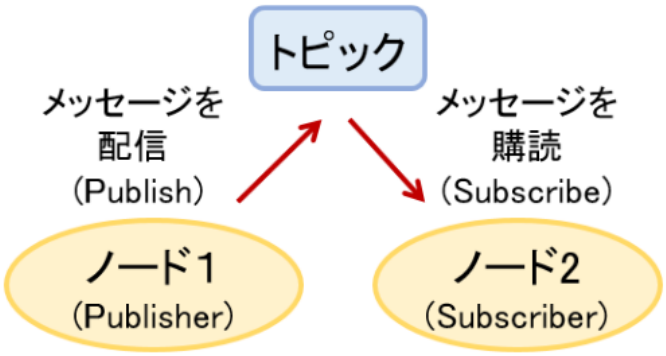

図 2: トピックによる通信モデル

- メッセージ (Message)

  トピックへの配信や購読の際に用いられる ROS データ型。自作のメッセージ型を定義することもできる。

- Publisher

  メッセージを特定のトピックへ配信 (Publish) をする配信者

- Subscriber

  特定のトピックを購読対象として登録しメッセージを購読 (Subscribe) する購読者

- トピック (Topic)

  メッセージをやりとりする回線。ノードはトピックに対してメッセージを配信する。また、メッセージを受信するためにトピックを購読する。

- サービス (Service)

  ノードと通信するもう 1 つの方法。サービスを介して、ノードに要求 (コマンド) を送ったり、要求に対する返答を受け取ったりできる。トピックが単なる通信路であるのに対し、サービスは受け取った情報に対して何らかの処理を行って結果を返すなどができる。

- トピック・ノードに関するコマンド

  rostopic list：現在購読・配信されている全てのトピックのリストを表示する。

  rosnode list：実行中の ROS ノードのリストを表示する。

  rosnode info /[node 名]：特定のノードに関する情報を表示する。ノードが配信・購読するトピックがわかる。

ROS の通信方法としては、上述のようにトピックとサービスがあるが、本研究ではトピックによる通信を使用してカメラ画像の送受信を行った [4]。

2.3 可視化ツール

ROS はソフトウェアの状態を可視化する様々なツールを備えている。その中には、複雑なノード構成を可視化できるツールがある。

2.3.1 rqt_graph

rqt_graph を使用すると、現在起動しているノードとトピックのやり取りの状況を可視化することができる。本研究では、カメラからの画像データ配信ノードが画像データ購読ノードと実際に通信を行えていることの確認およびデバッグにこの rqt_graph を使った。

ノードが起動している状態で、新しい端末を開き以下のコマンドを実行すると図 3 のように Publisher、Subscriber、Topic が表示され、ノード間の関係を可視化することができる。

```
$ rosrun rqt_graph rqt_graph
```

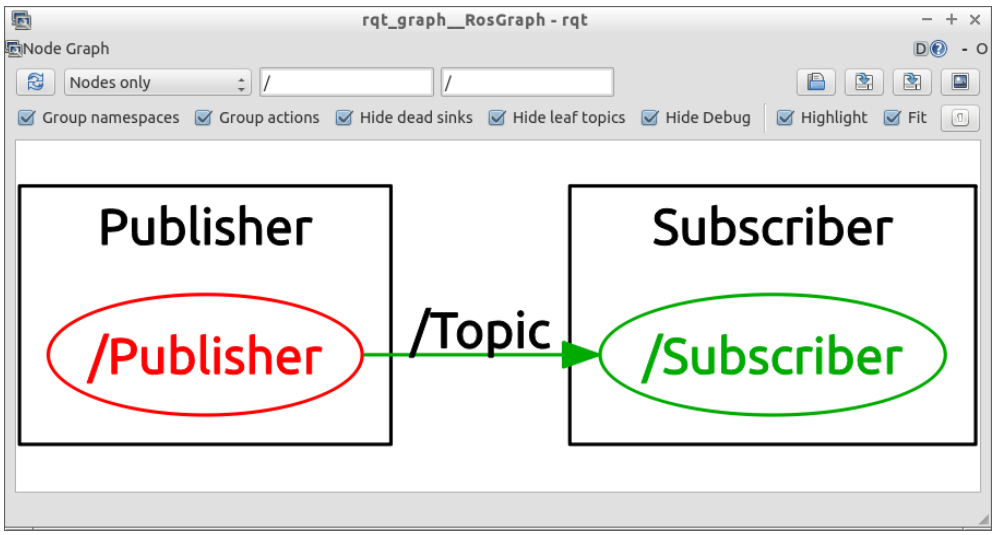

図 3: rqt_graph

3　実装

3.1　実装方針

実装方針としては、ROS のもとでカメラからの画像取得とロボットが目標地点へ到達する行動を実現するために、新たに 2 つの tracking ノードと control_nxt ノード [2] を作成する。tracking ノードではカメラからの画像取得、および目標物上に追跡点を指定して物体追跡 [2] を行い、cp トピックと camera トピックに配信を行う。cp トピックには追跡点に関するメッセージを配信し、camera トピックには画像データをイメージメッセージとして配信する。cp トピックに配信したメッセージは control_nxt ノードで購読し、利用する。tracking ノードが camera トピックにイメージメッセージを配信できているかを確認するために、イメージメッセージを購読するノードを実行する。ここでは、例として ROS の既存の、画像表示のための image_view ノードを用いる。

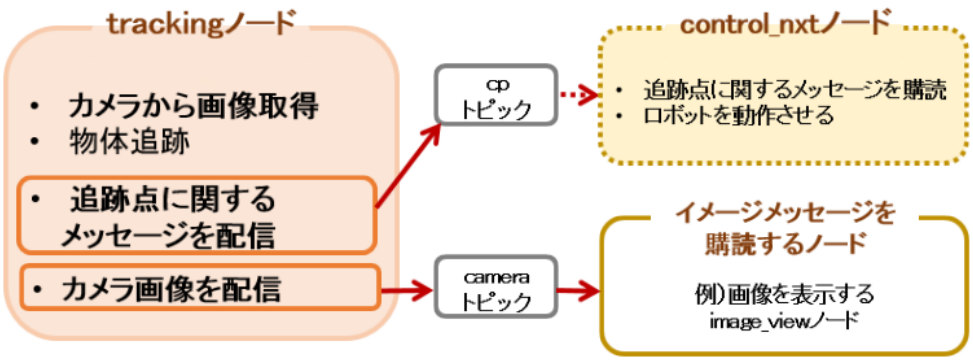

図 4: 実装方針

3.2 実装

作成した tracking ノードでは以下のことを行う。まず Xtion を起動する。Xtion の開発環境として OpenNI2 を使用した。OpenNI2 とはモーションキャプチャデバイスの開発環境である。本研究では、基本的に Ubuntu および ROS が提供するパッケージで OpenNI2 を利用し、一部 Xtion の DVD で提供される nonfree の Linux 用ドライバを用いた。OpenNI2 を初期化して、Xtion の RGB カメラを起動する。

Xtion 起動

```
# OpenNI2 をイニシャライズし、Xtion の RGB カメラを起動する
openni2.initialize()
dev = openni2.Device.open_any()
print dev.get_sensor_info(openni2.SENSOR_COLOR)
color_stream = dev.create_color_stream()
```

次に RGB 画像取得を行う。OpenNI2 でカメラフレームをキャプチャするために、ビデオモードの設定をする。そしてフレームレートや解像度などを決め、RGB 画像の取得を行う。

RGB 画像取得

```
# Video モードを設定する
video_mode = openni2.VideoMode()
video_mode.fps = 30
video_mode.resolutionX = 640
video_mode.resolutionY = 480
video_mode.pixelFormat = openni2.PIXEL_FORMAT_RGB888

color_stream.set_video_mode(video_mode)

color_stream.start()
```

次に目標物上に追跡点を指定し、物体追跡を行う [2]。このとき、追跡点に関する情報を control_nxt ノードで使用するために、追跡点に関する情報を持つ CharacterPoint という自作のメッセージ型の定義をした。そして、cp トピックに CharacterPoint 型のメッセージを配信する。このメッセージを control_nxt ノードで使用するためには、cp トピックを CharacterPoint 型で購読する。

CharacterPoint 型

```
int32 x
int32 y
int32 width
int32 height
```

cp トピックに配信

```
#自作のメッセージ型を使用するためには下記の 2 行がいる
import roslib
roslib.load_manifest('my_nxt_ros') #引数はパッケージ名
#メッセージ型の定義をインポート
from my_nxt_ros.msg import CharacterPoint

#cp トピックに配信する Publisher を作る
pub_cp = rospy.Publisher('cp', CharacterPoint,
       queue_size=1000)

#CharacterPoint 型メッセージ
msg_cp = CharacterPoint()
#ウィンドウサイズ
msg_cp.width = video_mode.resolutionX
msg_cp.height = video_mode.resolutionY
#追跡点の座標
msg_cp.x = self.points_now[0][0][0]
msg_cp.y = self.points_now[0][0][1]

#cp トピックに配信
pub_cp.publish( msg_cp )
```

cp トピックから購読

```
#自作のメッセージ型を使用するためには下記の 2 行がいる
import roslib
roslib.load_manifest('my_nxt_ros') #引数はパッケージ名
#メッセージ型の定義をインポート
from my_nxt_ros.msg import CharacterPoint

#cp トピックから CharacterPoint 型のメッセージを購読
rospy.Subscriber('cp', CharacterPoint, cpCallback)
```

Xtion から取得した画像の色空間は RGB であるが、ROS では色空間として標準的に BGR が用いられるため、このままでは利用できない。よって、次に OpenCV で色空間を RGB から BGR に変換する。

色空間変換

```
# RGB カメラで捉えた画像を OpenCV2 で使えるようにする
frame = color_stream.read_frame()
frame_data = frame.get_buffer_as_uint8()
color_array = numpy.frombuffer(frame_data,
    dtype=numpy.uint8).reshape((480,640,3))

# RGB から BGR に変換
bgr = cv2.cvtColor(color_array, cv2.COLOR_RGB2BGR)
```

最後に画像データの配信を行う。まず、tracking というノードを作成する。次にトピックに配信を行う Publisher を作成する。ここで、配信するトピック名は camera とする。画像データをトピックに配信するには、ROS のデータ型に変換する必要があり、ROS の Python のバインディングを用いて OpenCV の画像データ型を ROS イメージメッセージに変換する。これでイメージメッセージとして配信する準備ができたため、tracking ノードからトピックへ、カメラフレームをイメージメッセージとして配信する。

画像データ配信

```
#tracking というノードを作成
rospy.init_node( 'tracking' )

# camera1 トピックに配信する Publisher を作る
pub_image = rospy.Publisher('camera', Image, queue_size=100)

# OpenCV の IplImage を ROS イメージメッセージに変換するために
CvBridge を使う
bridge = CvBridge()

# camera トピックに配信する
pub_image.publish(bridge.cv2_to_imgmsg(bgr, 'bgr8'))
```

また、tracking ノードと control_nxt ノード間においてトピックによる通信を行っているため、この 2 つのノードをまとめて実行するための launch ファイルを作成した。

launch ファイル

```
<?xml version="1.0" encoding="UTF-8"?>

<launch>
 <!-- tracking.py & controle_nxt -->
 <node pkg="my_nxt_ros" type="tracking" name="tracking"
       output="screen"/>
 <node pkg="my_nxt_ros" type="control_nxt" name="control_nxt"
       output="screen"/>
</launch>
```

3.3 実行結果

tracking ノードが画像の配信をしているかの確認のために、tracking ノードと image_view ノードを同時に実行した。これによりカメラで取得した画像を image_view ノードで表示することができた。image_view ノードの実行画面を図 5 に示す。

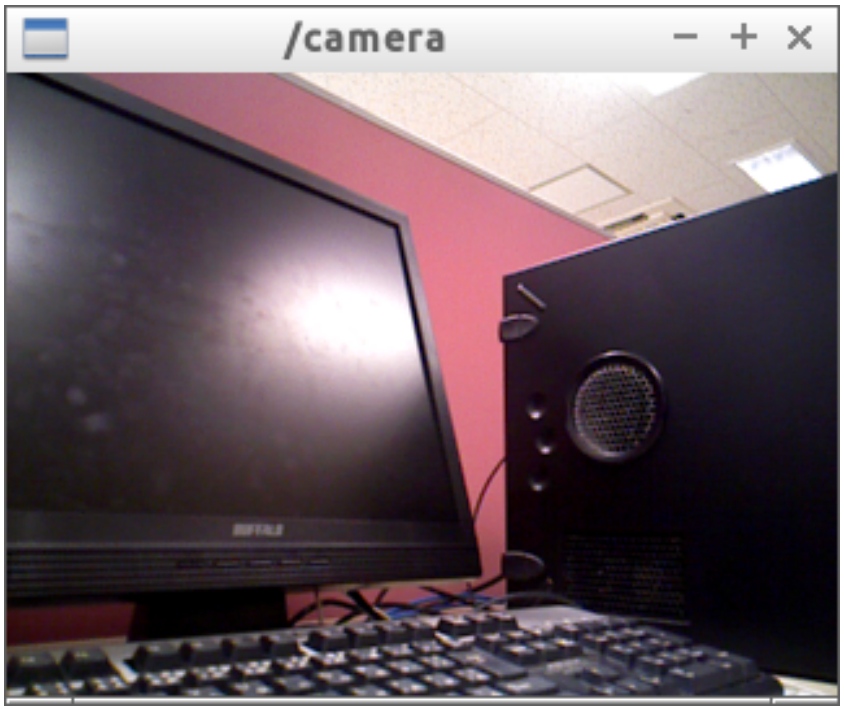

図 5: image_view による画像表示

また、rqt_graph で tracking ノードと image_view ノードと control_nxt ノードの関係を可視化すると図 6 のようになり、tracking ノードが camera トピックと cp トピックに配信を行っていることが確認できる。

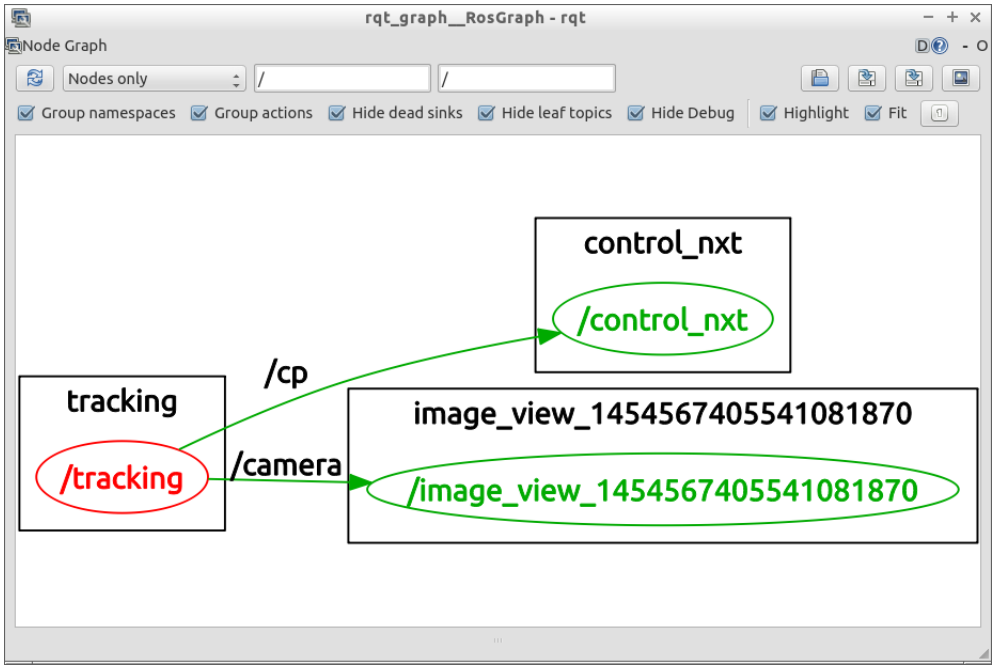

図 6: ノード間の関係

4　まとめ

本研究によって、RGB-D カメラから画像の取得を行い、取得した画像やそれに関する情報を ROS を使用して配信できるようになった。これにより、ROS プラットフォームでのロボットの開発にカメラ画像を用いることが可能となった。今後の課題として、カメラ画像を使用した目的の場所に移動するロボットの開発には高精度の物体認識を行うことが必要となってくる。また、物体認識に距離情報を用いることがあるので、RGB センサーと深度センサーから同時に情報を取得することが望まれる。さらに、今回 ROS のもとでカメラ画像の取得が可能となったため、今後目的を達成するロボットの開発を ROS のもとで統合的に行うことが望まれる。

5 謝辞

本研究の遂行および本論文の執筆にあたり、丁寧にご指導して下さった新出尚之准教授に深く感謝し、厚く御礼申し上げます。また、新出研究室の皆様に感謝の意を表します。ありがとうございました。

参考文献

[1] ROS.org. http://www.ros.org/.

[2] 兼松明未. ROS によるロボットの目標地点への到達行動の実現について. 2015 年度卒業論文, 奈良女子大学理学部情報科学科, 2016.

[3] Xtion PRO LIVE. https://www.asus.com/jp/3D-Sensor/Xtion_PRO_LIVE/.

[4] Aaron Martinez and Enrique Fernandez. *Learning ROS for Robotics Programming*. Packt Publishing, 2013.