

Developers Note for ADAMR2 Project

一関高専 専攻科 生産工学専攻 2年 藤野航汰

2020年12月2日

目次

第 1 章	ハードウェア組立に関する Tips	2
1.1	被覆付圧着端子の付け方	2
第 2 章	URDF を記述する	5
2.1	URDF とは？	5
2.2	何故 URDF を書くのか？	6
2.3	ADAMR2 の実機用 URDF モデリング	6
2.4	URDF モデリング用パッケージの作成	6
2.5	base_link の作成と追加	10
2.6	caster_*_link の作成と追加	16
2.7	lidar_link の作成と追加	20
2.8	wheel_link の作成と追加	23
第 3 章	デバイス設定	27
3.1	F310/F710 ゲームパッドを ROS で利用する	27
3.2	RPLiDAR A2 を利用する	30
第 4 章	コントローラの実装	32
4.1	ros_control とは	32
4.2	コントローラの構築に必要なもの	33
4.3	adamr2_control パッケージの準備	33
4.4	adamr2_driver の作成	36
参考文献		44

第1章

ハードウェア組立に関する Tips

1.1 被覆付圧着端子の付け方

ADAMR2 では、スイッチや端子台との接続に被覆付圧着端子を使用しています。被覆付圧着端子を配線ケーブルに取り付ける際は、裸圧着端子よりも多くのことに気を付けなければなりません。ここでは被覆付圧着端子を取り付ける際の手順と注意点を説明します。

1.1.1 使用する端子とケーブルについて

ADAMR2 で使用する配線部品を表 1.1 に示します。

Name	Model Number
ニチフ 銅線用絶縁被覆付圧着端子丸型	TMEV1.25-3
差込み型接続端子 187 シリーズ (バリュー品) メス (嵌合部絶縁型)	MTR-480809-FA
絶縁付圧着端子 Y 型	F1.25-5
通信機器用ビニル電線 KV シリーズ	KV 1.25SQ カ-200
通信機器用ビニル電線 KV シリーズ	KV 1.25SQ カ-200

Table1.1 List of Wiring Components

使用する配線ケーブルは太さ AWG16(sq1.25) の撲線ビニル電線です。被覆付圧着端子は、内径 $\phi 3$ mm の丸型圧着端子、187 サイズの差込型接続端子、内径 $\phi 5$ mm の Y 型圧着端子の 3 種類です。いずれも AWG16 サイズの電線に対応しているものを使用します。

1.1.2 必要な工具

配線には以下の工具が必要となります。

- 圧着工具
- ワイヤーストリッパー
- ニッパー
- 電工ペンチ

被覆付圧着端子をケーブルに取り付けるには、専用の工具が必要となります。ここではホーザン株式会社の絶縁被覆付圧着端子用圧着工具 P-743 を使用しました。

ワイヤーストリッパーはホーザン株式会社の P-90-A を使用しました。

また、先端に被覆付圧着端子のためのカシメ部がついた電工ペンチがあると圧着を確実に行うことができるでの、あった方が望ましいです。

1.1.3 圧着の手順

圧着端子の種類が違っても、作業手順はおおよそ同じです。まずはケーブルの被覆を剥ぎます。P-90-A の 1.6 mm のスロットを使うと綺麗に電線を剥ぐことができます。およそ 6 mm だけ被覆を剥ぐと、圧着をス



Fig.1.1 HOZAN P-743



Fig.1.2 HOZAN P-90-A



Fig.1.3 A Crimp Plier

ムーズに行うことができます。

そして、ここが重要なのですが、圧着端子を付ける前に図 1.1.3 のように電線をねじって撲っておきます。電線をねじっておくことで、圧着した際に端子との摩擦が増し、端子が外れにくくなります。^{*1}

電線の準備ができたら、圧着端子を挿入し、圧着工具のダイスにセットします。今回使用する圧着端子のサイズはすべて 1.25 なので、P-743 の赤色のマークが付いたダイスにセットします。この時、圧着工具に端子をセットする際の向きに注意してください。正しい向きにセットしないと圧着が上手くいきません。図 1.1.3 を参考に、正しい向きで端子をセットしてください。

端子をダイスにセットしたら、ずれないように気を付けながらハンドルを握って圧着します。

丸型端子及び Y 型端子は、P-743 による圧着だけではしっかりと圧着することができます。しかし、差込型接続端子の場合はこれだけでは端子がすぐ抜けてしまう可能性があります。^{*2}そこで、P-743 による圧着の後、電

^{*1} 電線をねじらずに圧着すると、特に差込型接続端子は簡単にすっぽ抜けます。

^{*2} 筆者が購入した差込型接続端子の品質に問題があったからかもしれません。



Fig.1.4 Twist the Wire



Fig.1.5 Direction for Setting the Crimped Terminal

工ペンチを使ってさらに圧力をかけ、圧着を強くする必要があります。かなり強めに力を込めて追い圧着すれば、すっぽ抜けてしまうことはなくなるはずです。

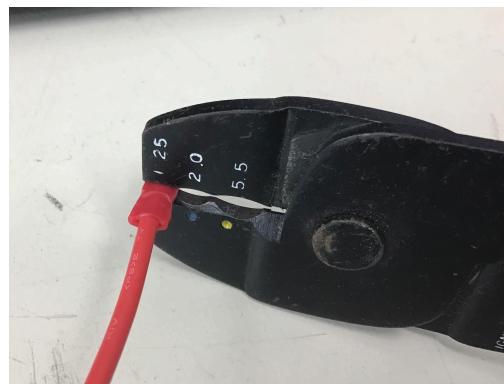


Fig.1.6 Crimp Further with Crimp Plier

以上で圧着端子のケーブルへの取り付け作業は完了です。圧着端子の取り付け方法に関しては、本ドキュメントだけでなく工具メーカーのページ等を参考に、正しい方法で作業を行うことを心掛けてください。

第2章

URDFを記述する

本章では、ROSにおいてロボットの構造をモデリングするためのフォーマットである URDF(Unified Robot Description Format) と、URDF をプログラマブルに記述することのできるマクロパッケージである `xacro`^{*1}について解説します。

2.1 URDFとは？

URDFについての説明を MathWorks の Web サイトから引用します。[1]

URDF (Unified Robotics Description Format) は、製造業の組み立てライン用ロボット マニピュレーター アームや遊園地用のアニメトロニクス ロボットなどのマルチボディ システムをモデル化するために、学術界や産業界で使用される XML 仕様です。

URDFでは、ロボットのモデルをリンク (Link) とジョイント (Joint) からなるツリー構造で表現します。ボディやホイール、センサ等、駆動しないブロックをリンクとして扱い、リンクとリンクとの接続(固定、回転、直動、etc)をジョイントとして扱います。URDFによってモデリングされたロボットは、最終的に図 2.1 のような構造になります。

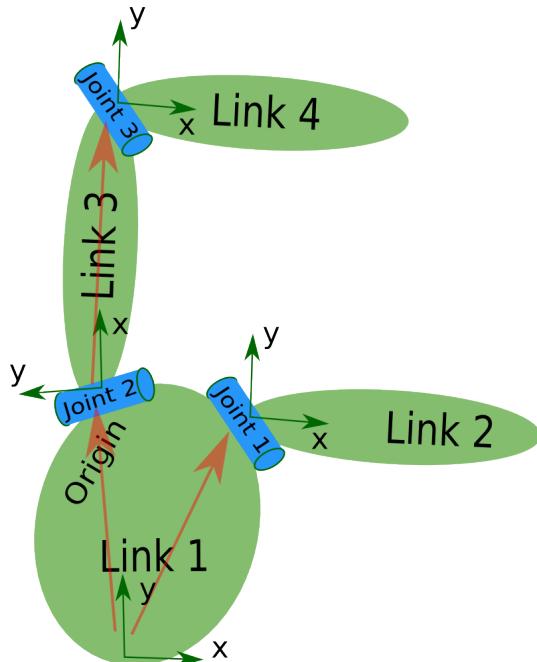


Fig.2.1 Tree Structure of URDF[2]

URDFを記述することで、ロボットを構成する各リンクの位置関係やジョイントの属性を表すことができます。また、URDFの各リンク/ジョイントに詳細なオプションを追加することで、シミュレーション用モデル

*1 <http://wiki.ros.org/xacro>

を作成することもできます。

2.2 何故 URDF を書くのか？

ROS の教本やチュートリアルでよく紹介されている URDF ですが、実のところ URDF を書かなくてもロボットの実機を動かすシステムを構築することができます。URDF が提供するのは、ロボット座標系におけるセンサやアクチュエータ等の位置関係を示す座標変換であり、これは TF パッケージを用いることでも実現できるものです。言い換えれば、TF を直接発行してロボットの各コンポーネントの座標変換を提供することができれば URDF を記述する必要は無いという事です。

では、URDF を記述する理由はあるのでしょうか？これはあくまで筆者の考えですが、以下のような利点があると考えます。

1. ロボットの各コンポーネントの位置関係を一括で管理することができる
2. ロボットの詳細な構造を可視化することができる
3. 実機と同じモデルでシミュレーションを行うことができる

URDF を記述することの大きな利点の 1 つとして、ロボットのモデルの可視化を行うことができるという点があると考えます。リンクやジョイントの座標変換を提供するだけなら TF パッケージを使うことで実現できるのですが、その場合モデルの可視化を行うことができません。そのため、入力した数値が正しいのかどうかを検証するのが困難になります。例えば、ロボットのハードウェアの設計を変更して、センサの位置が代わってしまったという場合を考えます。TF パッケージと URDF のどちらを使う場合でも、新しいセンサの位置を 3D CAD の設計データから計算して数値を導出するのは同じですが、TF パッケージを使う場合は可視化の手段が乏しいため、その数値が正しいかどうかを検証することが難しくなります。一方で URDF を使う場合は、URDF ファイルを編集^{*2}し、更新したモデルを `rviz` で可視化することで、センサの位置が正しい位置にいるのかどうかを目で確かめることができます。

また、副産物的な考え方ですが、実機用の URDF ファイルにシミュレーションのためのオプションを追記することで、ロボットのシミュレーション環境を簡単に整えることが可能になります。`ros_control`^{*3} のフレームワークと合わせることで実機とシミュレーションで同一のコントローラを使ってロボットを動かすことができるため、SLAM や Navigation 等のアプリケーションの開発を効率よく進められるようになります。

以上の理由から、ここでは URDF を記述してロボットのモデリングを行うことを強く推奨します。次の小節から、ADAMR2 で実際に使用している `xacro` ファイルをもとに、URDF によるロボットの実践的なモデリングについて解説します。

2.3 ADAMR2 の実機用 URDF モデリング

早速 `xacro` を用いて実際に ADAMR2 の実機用 URDF モデリングを行います。`xacro` を用いるため、URDF を直接記述することはありませんが、URDF の記述に関する基礎知識が必要になります。このセクションでは URDF 記述のチュートリアルについては取り扱わず、より実践的な内容を説明します。URDF モデリングのチュートリアルは Web 上に大量に存在します。この小節を読む前に、そちらを参照して基礎知識を身に付けてください。

これからモデリングするロボットのスケッチを図 2.2 に、リンクとジョイントのグラフを図 2.3 に示します。

トップレベルのリンクとして `base_footprint`(空のリンク) があり、その下にロボットのボディである `base_link` があります。ホイール、センサ、キャスターのリンクはすべて `base_link` の子リンクです。

2.4 URDF モデリング用パッケージの作成

2.4.1 ディレクトリ構成

ここから実際に URDF モデリングを行っていきます。まずは、URDF モデルを置くための ROS パッケージを作成します。慣習的に、ロボットの URDF モデルは `*_description`(*はロボットもしくはプロジェクトの名前) という名前の ROS パッケージに配置します。ADAMR2 プロジェクトならば「`adamr2_description`」という名前になります。URDF モデリングを始める前にパッケージを作っておきましょう。ビルド依存パッケージは無いので、パッケージ作成コマンドにオプションは必要ありません。

^{*2} 実際は `xacro` で記述した後に URDF をエクスポートすることが多いので、直接 URDF ファイルを編集することはありません。

^{*3} http://wiki.ros.org/ros_control

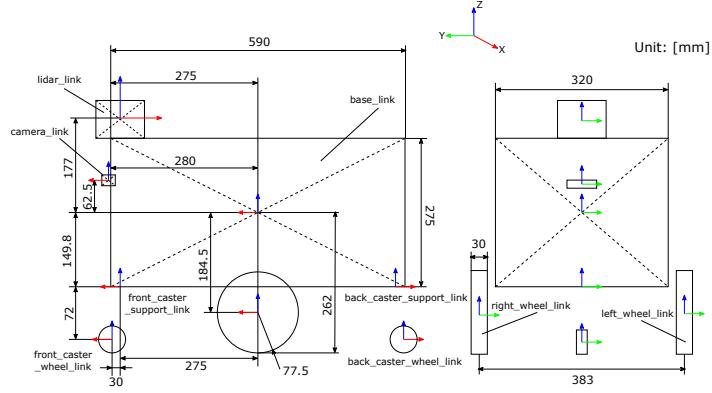


Fig.2.2 Link Structure of Diff-Drive Mobile Robot

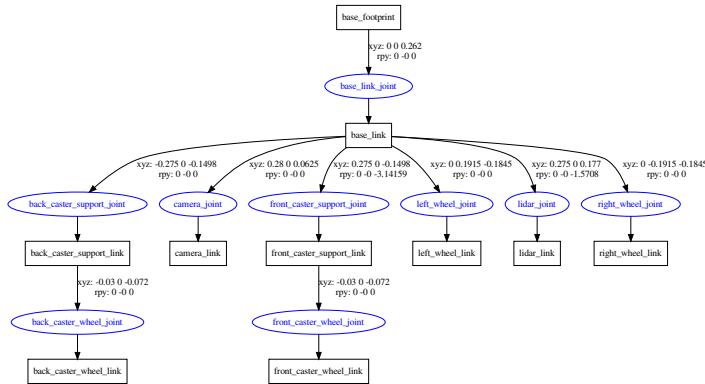


Fig.2.3 URDF Tree of ADAMR2

Code 2.1 Create a Package to put the URDF Model in

```
catkin create pkg adamr2_description
```

パッケージが作成できたら、中にいくつかのディレクトリを作成します。adamr2_description パッケージのディレクトリ構成はコード 2.2 のようにします。

Code 2.2 Directory Structure of adamr2_description

```
adamr2_description/
├─ launch/
├─ meshes/
└─ urdf/
  ├─ package.xml
  └─ CMakeLists.txt
```

launch/ディレクトリには、URDF モデルの可視化を行うための launch ファイルを置きます。meshes/ディレクトリには、STL 形式のロボットの 3D モデルを置きます。rviz/ディレクトリには、URDF モデル可視化時の rviz の設定ファイルを置きます。そして、urdf/ディレクトリに xacro ファイルを格納していきます。

urdf/ディレクトリの中身のディレクトリ構造はコードのようになります。

Code 2.3 Directory Structure of urdf/

```
urdf/
└─ base/
  └─ base.xacro
```

```

└── caster/
    └── caster.xacro
└── lidar/
    └── lidar.xacro
└── wheel/
    ├── transmission.xacro
    └── wheel.xacro
└── robot.xacro

```

`robot.xacro` がルートファイルです。ルートファイルから各コンポーネントの `xacro` ファイルをインクルードし、ロボットのモデルを作成します。このファイルは最終的に `xacro` パッケージのノードを用いて URDF ファイルに変換されます。

ロボットの主要コンポーネントであるボディ、キャスター、LiDAR、ホイールのそれぞれに対してディレクトリを作り、その中に各々の `xacro` ファイルを格納します。ここで、キャスターはロボットの前後に 1 つずつ、ホイールはロボットの左右に 1 つずつ存在しますが、それぞれに対応する `xacro` ファイルを複数作成する必要はありません。共通のモジュールとして `xacro` ファイルを作成しておき、ルートファイルから読み込む際に名前や位置を付けることで各リンクを定義します。また、`wheel` には `transmission.xacro` というファイルがありますが、これは `ros_control` を用いたコントローラを作る際に必要となるオプションを設定するためのファイルです。

2.4.2 ルートファイルの作成

では早速、最初の `xacro` ファイルを作成してみましょう。ルートファイルである `robot.xacro` を記述してみます。urdf/ディレクトリの中に `robot.xacro` という名前の空のファイルを作成し、まずコード 2.4 のように記述します。

Code 2.4 `robot.xacro`

```

<?xml version="1.0"?>
<robot name="adamr2" xmlns:xacro="http://ros.org/wiki/xacro">
</robot>

```

URDF のルートファイルは、`robot` という要素のタグで始まる必要があります。他の全ての要素はこの `robot` タグの中に入っているなければなりません。^[3] `robot` タグの `name` 属性にはロボットの名前を指定します。ここで指定した名前は後々使用することになるので、一意な名前をつけておきましょう。また、`xmlns:xacro` 属性にも値を設定しておきます。この値は ROS 固有のものなので、コピペして構いません。1 行目の文は XML のバージョン指定です。1.0 を指定しておきます。

これだけでは何も起きてないので、`robot` タグ内に要素を追加してみましょう。コード 2.5 のように追記してみます。

Code 2.5 `robot.xacro`

```

<?xml version="1.0"?>
<robot name="adamr2" xmlns:xacro="http://ros.org/wiki/xacro">
    <link name="base_footprint"/>
</robot>

```

`link` タグを記述することにより、ロボットにリンクを追加することができます。本来、`link` タグは `inertial`、`visual`、`collision` の要素を持つのですが^[4]、`base_footprint` リンクは実体を持たない空のリンクなので、空要素タグで記述します。

`link` 空要素タグの中で `name` 属性に値を代入しています。この `name` 属性をつけた名前がリンクの名前となり、他のリンクから参照できるようになります。

2.4.3 URDFへの変換と構文チェック

まだ 1 つのリンク（しかも空のリンク）しか追加していませんが、ひとまず URDF の構文チェックを行ってみましょう。ROS のツールに `check_urdf` というものがあり、これを使用すると URDF が正しい構造になっているかどうかをチェックすることができます。

しかし、xacro ファイルのままではチェックを行うことができません。⁴そのため、まず xacro ファイルから URDF ファイルに変換し、変換したファイルをチェックにかけることになります。

xacro ファイルを URDF ファイルに変換するには、xacro パッケージのノードを使用します。roscore を起動した状態で、コードのコマンドを実行します。

Code 2.6 Conversion from xacro to urdf

```
rosrun xacro xacro robot.xacro > robot.urdf
```

このコマンドによって、robot.xacro と同じディレクトリに robot.urdf が生成され、コード 2.7 に示すような内容が書き込まれます。

Code 2.7 Generated URDF file

```
<?xml version="1.0" encoding="utf-8"?>
<!-- =====>
<!-- | This document was autogenerated by xacro from robot.xacro | -->
<!-- | EDITING THIS FILE BY HAND IS NOT RECOMMENDED | -->
<!-- =====>
<robot name="adamr2">
  <link name="base_footprint"/>
</robot>
```

生成された URDF ファイルの先頭に書かれている通り、このファイルを直接編集することは推奨されません。生成元となった xacro ファイルを編集して修正作業等を行います。

現時点での robot.xacro は空のリンクが 1 つだけのシンプルなファイルなので、URDF に変換しても大した違いはありません。とりあえずこの URDF ファイルをチェックにかけてみましょう。コード 2.8 を実行します。

Code 2.8 Check URDF

```
check_urdf robot.urdf
```

このコマンドは ROS ノードではないため、roscore が起動していないなくても使うことができます。正しく記述できている場合、コード 2.9 に示すようなメッセージがターミナルに表示されます。

Code 2.9 Check Result

```
robot name is: adamr2
----- Successfully Parsed XML -----
root Link: base_footprint has 0 child(ren)
```

ここまで作業で、xacro ファイルの記述と URDF への変換、及び構文チェックまでを行うことができました。URDF を xacro によって記述するときは、このように URDF に変換してチェック作業を行いながら書くことになります。

2.4.4 URDF の可視化

urdf_to_graphviz コマンドを使えば、作成した URDF をグラフにして可視化することができます。コード 2.10 に示すコマンドを実行することで、URDF のツリー構造を可視化した PDF 画像ファイルを得ることができます。

Code 2.10 Visualize URDF Tree

```
urdf_to_graphviz robot.urdf
```

出力結果は図 2.4 のようになります。現時点では base_footprint リンクしか定義していないので、グラフのノードは 1 つだけです。

⁴ チェックツールを実行すること自体は可能ですが、URDF に含まれるリンク等が展開されないのでチェックにはなりません。

base_footprint

Fig.2.4 URDF Tree (base_footprint Only)

2.5 base_link の作成と追加

節 2.4 までの作業で、ルートファイル robot.xacro の作成と base_footprint リンクの定義を行いました。この節ではロボットのボディのリンクである base_link の定義とルートファイルへの追加の作業を行います。

2.5.1 base.xacro の作成

まず、base_link の要素や属性の定義を記述するファイルを作成します。urdf/ディレクトリ以下に base/ディレクトリを作成し、base.xacro という名前のファイルを作成します。そして、まずはコード 2.11 のような内容を記述します。

Code 2.11 base.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:macro name="base" params="parent *joint_origin">
    </xacro:macro>
  </robot>
```

ルートファイルと同じく、robot タグをトップレベルに記述し、その中に他の全ての要素を記述していきます。ただし、コンポーネントファイルには name 属性を書く必要はありません。名前空間だけ指定しておきましょう。

robot タグの中には、xacro:macro タグが書かれています。これが xacro におけるマクロの定義であり、ルートファイルから呼び出されるものです。マクロの中に link や joint 等のタグを記述しておけば、ルートファイルから呼び出されたときにそれらが展開される、という仕組みです。xacro:macro タグには name 属性と params 属性があります。name 属性にはマクロの名前を指定します。params 属性には、そのマクロが取る引数を指定することができます。引数は複数設定することができ、半角スペースで区切って記述します。コード 2.11 では、引数として parent(親リンク名の指定) と *joint_origin^{*5}(座標原点の指定) の 2 つを取るように設定しています。

base.xacro ファイルでは、このマクロを読み込んだら「base_link の定義」と「base_footprint と base_link を繋ぐジョイント base_link_joint の定義」が行われるようにマクロを定義します。

2.5.2 joint の設定

この小節では base_link_joint の定義を行います。コード 2.12 のようにマクロタグ内に joint タグを追記します。

Code 2.12 Configurate base_link_joint

```
<?xml version="1.0"?>
```

^{*5} アスタリスクが先頭に付くパラメータは「ブロックパラメータ」と呼ばれるパラメータで、任意の要素を展開することができます。

```

<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:macro name="base" params="parent *joint_origin">
    <joint name="base_link_joint" type="fixed">
      <xacro:insert_block name="joint_origin"/>
      <parent link="${parent}"/>
      <child link="base_link"/>
    </joint>
  </xacro:macro>
</robot>

```

joint タグは name と type の 2 つの属性を持ち、また以下の要素を持ちます。[5]

- origin : 親リンクから見たジョイントの原点座標
- parent : 親リンク名
- child : 子リンク名
- axis : ジョイントの軸。回転ジョイントの場合は回転軸を表す
- calibration : ジョイントの絶対位置を校正するために使用される、ジョイントの基準位置
- dynamics : ジョイントの物理的特性を指定するための要素
- limit : ジョイントの角度、力、速度のリミットを指定するための要素
- mimic : 他のジョイントの動きを真似させるときに使用する要素
- safety_controller : ジョイントが安全に動作するために設定するソフトなリミット

このうち、ジョイントの定義に必須なのは parent と child のみで、他の要素は指定しなくてもジョイントを定義することができます。

ここでは、ジョイント名を「base_link_joint」、ジョイントタイプを「fixed」とし、要素には parent と child、そして origin を設定します。

child 要素は base_link です。base_link はまだ存在せず、2.5.3 で定義します。parent 要素にはマクロが引数として受け取った値をそのまま使用します。引数をマクロ内で利用するには、\${PARAMETER_NAME} の書式を使用します。

joint タグの origin 要素で指定するのは、親リンクから見たジョイントの原点座標です。図 2.5.2 のように、ジョイントが子リンクの原点になります。ここでは base_link が子リンクなので、base_link の座標系原点は base_link_joint の位置になるという事になります。

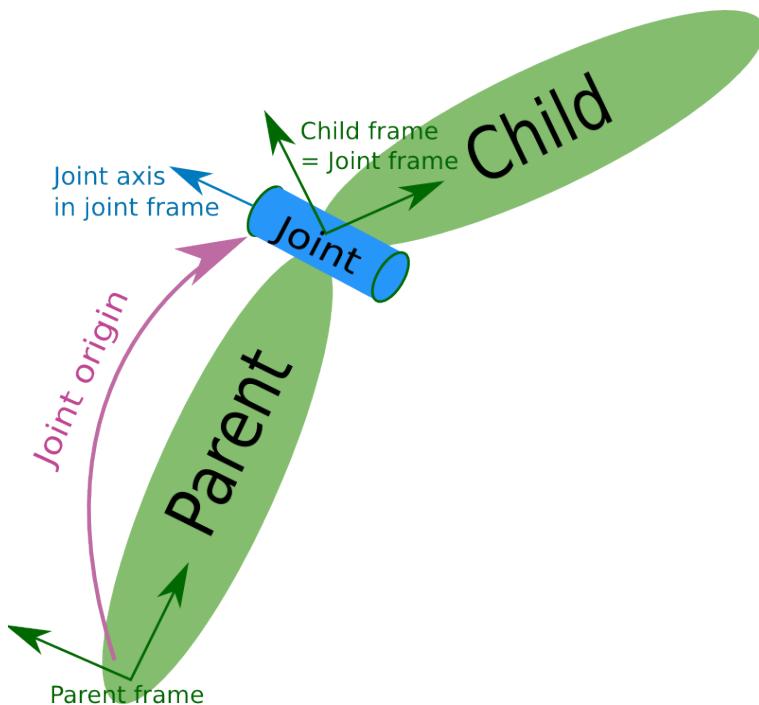


Fig.2.5 Coordinate Transformation Relationship Between Link and Joint[5]

origin 要素もマクロが引数として受け取ったものを使用します。ただし、origin 要素は単なる値ではないた

め, `parent` 要素のようにそのまま使うことはできません。そこで使用するのがブロックパラメータと呼ばれる引数です。これを使用することで、ルートファイルからジョイントの `origin` 要素を定義できるようになります。コード 2.12 では`*joint_origin` という名前でブロックパラメータを定義しています。ルートファイルから読み込む際の詳細はで説明します。

ジョイントの設定は以上で完了です。次は `link` タグの設定を行います。

2.5.3 link タグの設定

ここから `link` タグを定義して、`base_link` の要素を記述していきます。`link` タグは `inertial`, `visual`, `collision` の3つの要素を持ちます。`link` タグ及び各要素の詳細について知りたい方は、ROS Wiki のページ⁶を参照してください。

Gazebo シミュレーションを行う際は全ての要素について定義する必要がありますが、`rviz` で可視化するだけなら `visual` 要素を設定するだけで事足ります。本章では実機用 URDF の記述を目的としているため、ここではシミュレーション用の設定については説明しません。シミュレーション用の設定については後の章で解説します。

`visual` 要素はモデルの見た目の要素を定義するタグです。コード 2.13 のようにマクロタグ内に `link` タグを追記し、更にその中に `visual` 要素を記述します。

Code 2.13 link Element

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:macro name="base" params="parent *joint_origin">
    <joint name="base_link_joint" type="fixed">
      <xacro:insert_block name="joint_origin"/>
      <parent link="${parent}"/>
      <child link="base_link"/>
    </joint>

    <link name="base_link">
      <visual>
        <origin xyz="0.0 0.0 0.0"/>
        <geometry>
          <mesh filename="package://adaml2_description/meshes/base_link.STL"/>
        </geometry>
        <material name="red">
          <color rgba="1.0 0.0 0.0 1.0"/>
        </material>
      </visual>
    </link>
  </xacro:macro>
</robot>
```

`visual` 要素は更に `origin`, `geometry`, `material` 要素を持ちます。`geometry` 要素ではリンクの見た目を表す形状を定義します。長方形、球、円柱等のプリミティブな図形を設定できる他、STL ファイルを読み込んで設定することができます。実際、コード 2.13 では `mesh` 要素に STL ファイルを指定して形状を設定しています。STL ファイルは 2.5.4 で作成します。

`origin` 要素では `geometry` 要素で指定した図形の原点の座標を設定することができます。リンクの座標系から見た位置、即ちこのリンクが所属するジョイントの原点からみた位置で指定することになります。このドキュメントでは、基本的にリンクの位置は `joint` タグの位置を設定することで決定します。そのため、`link` タグの `origin` 要素の値はほとんどの場合でゼロになります。

`material` 要素ではリンクの色を設定することができます。ただし、プリミティブ図形または STL ファイルを使用する場合は、単色でしか設定できません。リンクの見た目の色を詳細に決定したい場合は、COLLADA ファイル(拡張子は*.dae)を用意する必要があります。ロボットの開発にとってあまり本質的ではないと考えたので、このドキュメントでは COLLADA ファイルの作成については説明しません。

⁶ <http://wiki.ros.org/urdf/XML/link>

2.5.4 メッシュファイル(STL形式)の作成と割り当て

visual要素の mesh に指定する STL ファイルを、3D-CADソフトであるSolidWorks2019を用いて作成します。SolidWorksにはパーツファイルをSTLにエクスポートする機能があるので、これを使います。

ROS REP:103^{*7}にあるように、ROSでは長さの単位を全てメートルで扱います。しかし、3D-CADで設計を行うときはミリメートル単位でモデリングするのが一般的です。従って、そのままSTLエクスポートすると、ROSから読み込んだときにモデルの大きさが1000倍になってしまいます。また、ROSではロボットの座標軸の向きは「X軸がロボットの前進方向、Z軸は鉛直上向き方向、Y軸は右手座標系となるような向き」と決められています。3D-CADのデフォルト座標軸がこのような向きになっていることは少ないため、STLファイルをエクスポートした後にBlender等の3D-CGファイルでスケールや向きを調整しなければならない場合が殆どです。

しかし、SolidWorksにはSTLファイルをエクスポートする際に、単位系や座標系を指定することができる機能があります。そのため、適切な設定を行いさえすれば3D-CGソフトで後処理を行うことは必要ありません。

図2.5.4に、base_linkに対応付けるロボットのボディの3D-CADモデルを示します。これはアセンブリファイルをパーツファイルに変換したもので、筐体の他、アクチュエータ(ホイールは含まない)も内包しています。3D-CADの座標軸が図の左下に表示されていますが、X軸はロボットの前方を向いているものの、Z軸が鉛直上向きになっていません。また、ミリメートル単位で設計を行ったため、単位変換を行う必要があります。

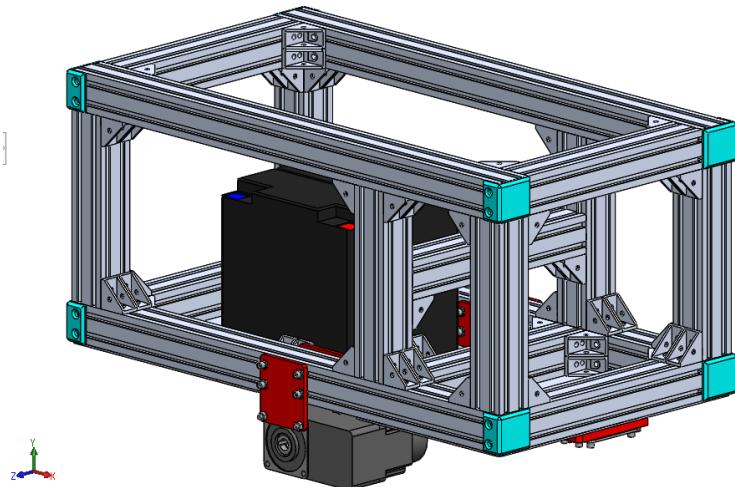


Fig.2.6 3D Model of base_link

STLファイルをエクスポートする前の準備として、まず座標系を定義しましょう。今回のモデルでは、原点はアルミフレームが描く直方体の中心に位置しているので、STLファイルの原点にしたい場所に新たに点を打つ必要はありません。まずは座標系のもとになる軸を定義します。フィーチャー→参照ジオメトリ→軸を選択します。そして、図2.5.4のように2つの平面を選択して、X軸、Y軸、Z軸の3本を新たに定義します。軸を追加したら、座標系を定義します。先ほど追加した軸を使って、ROSが推奨する座標軸の向きになるよう設定します。少し見づらいですが、図2.5.4ではX軸に軸1を、Z軸に軸3を対応付けた結果、望ましい座標軸の向きになっていることがわかります。

座標系の定義ができたので、STLファイルをエクスポートしましょう。指定保存を開き、ファイルの種類(拡張子)に「STL」を選択します。STLを選択すると、「オプション」ボタンが表示されるので、それをクリックします。

オプションを開くと、図2.5.4のような画面が表示されます。ここで単位と出力座標系を設定します(2.5.4で赤線を引いたところ)。単位をメートルに、出力座標系に先ほど作成した座標系(ここでは「座標系1」という名前)を設定し、「モデルの座標系で出力」というチェックボックスをオンにします。お好みで解像度の設定もできます。「粗表示」を選ぶとモデルが粗くなってしまいますが、ファイルの容量を減らすことができます。

この設定で「OK」をクリックし、「保存」を選択することで、ROSでそのまま使用できるSTLファイルを

*7 <https://www.ros.org/reps/rep-0103.html>

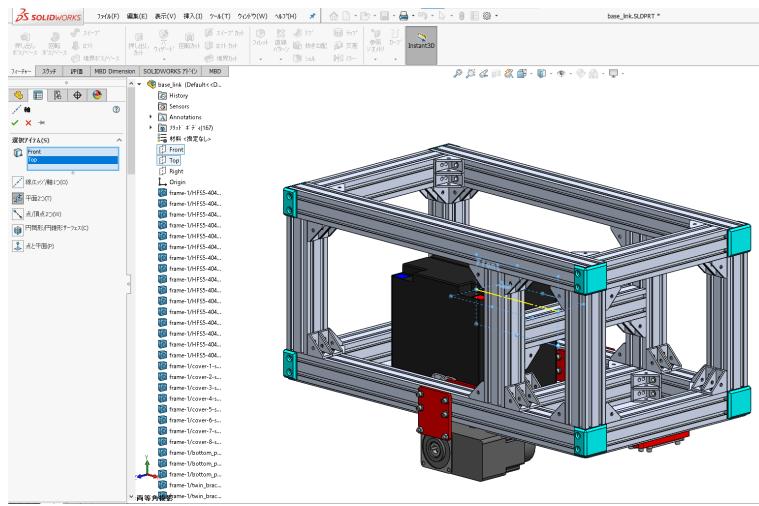


Fig.2.7 Add Axes for STL Export

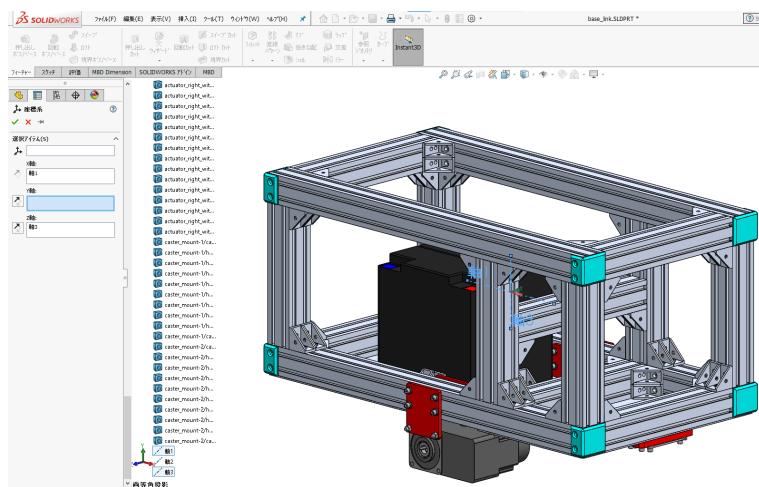


Fig.2.8 Add Coordinate System for STL Export

作成することができます。このSTLファイルを、`adamlr2_description/meshes/`ディレクトリに `base_link.STL` という名前で保存します。コード 2.13 が正しく記述されていれば、STLファイルを読み込むことができるようになります。

2.5.5 ルートファイルからインクルードする

ここまででの作業で `base_link` の定義を完了することができたので、これをルートファイルから読み込んでロボットモデルに組み込みましょう。コード 2.14 に、`robot.xacro` にいくつかの文を追記します。

Code 2.14 Add `base_link` to Robot Model

```
<?xml version="1.0"?>
<robot name="adamlr2" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:include filename="$(find adamlr2_description)/urdf/base/base.xacro"/>

  <link name="base_footprint"/>

  <xacro:base parent="base_footprint">
    <origin xyz="0.0 0.0 0.262"/>
  </xacro:base>

</robot>
```

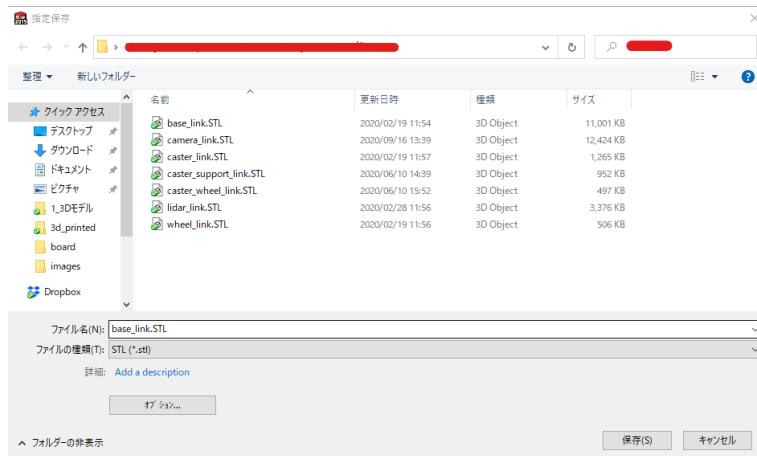


Fig.2.9 Save as STL File

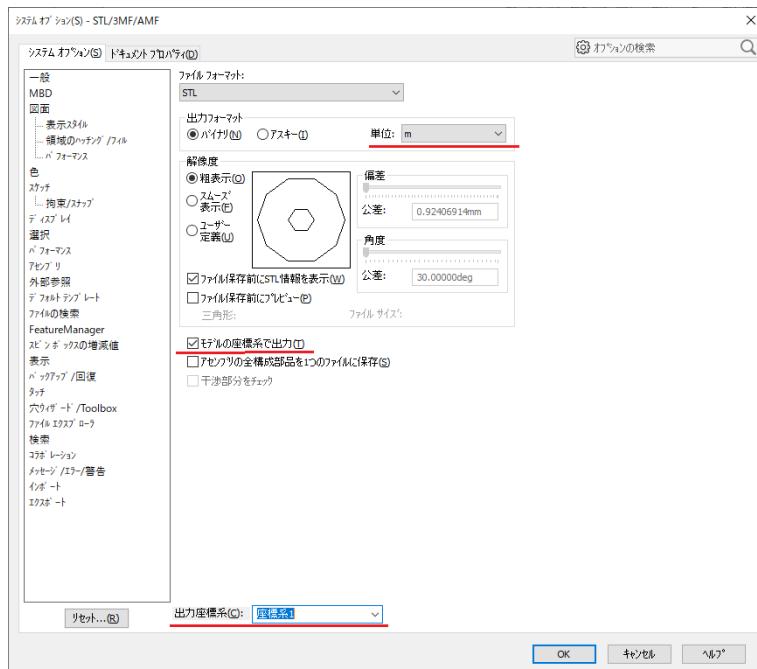


Fig.2.10 STL Export Options

3行目に追記したコマンド `xacro:include` で, `base.xacro` ファイルを読み込んでいます。これにより, `base.xacro` で定義したマクロを使用できるようになります。そして, 7行目から9行目で `xacro:base` マクロを使って `base_link` と `base_link_joint` を定義しています。マクロタグ内で `origin` 要素の記述を行っているのは、マクロがブロックパラメータとして `origin` 要素を引数としているからです。こうすることにより、設計変更が発生してリンクの場所を変更しなければならなくなっても、`robot.xacro` を編集するだけで対応することが出来るようになります。

`robot.xacro` を編集したら、念の為 URDF チェックをしておきましょう。URDF の構文チェックの手順は、2.4.3 の手順を参照してください。URDF を正しく記述できていれば、コード 2.15 のように表示されるはずです。

Code 2.15 Result of Check URDF

```
robot name is: adamr2
----- Successfully Parsed XML -----
root Link: base_footprint has 1 child(ren)
    child(1): base_link
```

2.5.6 rviz による可視化

せっかくリンクを追加したので、rviz で可視化してみましょう。launch/ディレクトリを作成し、その中に rviz を起動するための launch ファイル display.launch を作成し、コード 2.16 の内容を記述します。

Code 2.16 display.launch

```
<launch>
  <arg name="model" default="$(find adamr2_description)/urdf/robot.xacro"/>
  <param name="robot_description" command="$(find xacro)/xacro $(arg model)"/>
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz"/>
</launch>
```

param タグで robot.xacro を展開し、robot_description という名前のパラメータに登録します。ROS ではロボットのリンクやジョイントの状態を、robot_description という名前のパラメータで一元管理しています。URDF の情報をこのパラメータに登録することで、URDF で定義したロボットのモデルを ROS に伝えます。

また、この launch ファイルでは 2 つのノードを起動しています。1 つは robot_state_publisher ノードです。このノードは robot_description パラメータの情報を解釈して TF を投げてくれるノードです。2 つ目は rviz です。

launch ファイルを作成したら、一旦パッケージをビルドしておきましょう。パッケージをビルドすることで、そのパッケージに launch ファイルが存在することをシステムに登録できます。^{*8}ROS のワークスペースに移動して、コード 2.17 のコマンドを実行します。

Code 2.17 Build adamr2_description Package

```
cd ~/catkin_ws/src
catkin build adamr2_description
source ~/catkin_ws/devel/setup.bash
```

パッケージをビルドしたら、launch ファイルを実行してみましょう。

Code 2.18 Launch display.launch

```
roslaunch adamr2_description display.launch
```

launch ファイルを実行すると、rviz が起動し、図 2.5.6 のような画面が表示されます。

起動した直後はオプションを何も設定していないので、何も表示されません。ロボットのモデルを表示するためには、左下の「Add」ボタンを押し、「RobotModel」を追加します(図 2.5.6)。

そして、Fixed Frame の値を base_footprint に変更します。デフォルトの値は map というフレームが割り当てられていますが、ロボットのモデルには map というフレームは存在しません。ロボットモデルのルートとなるフレーム名は base_footprint に設定しているので、プルダウンメニューを開き、base_footprint を選択します。

ここまで設定すれば、図 2.5.6 のようにロボットのモデルが正しく表示されるはずです。base_link がホイールの大きさを考慮した高さに置かれていることが分かると思います。

本節では、コンポーネントとしての xacro ファイルの記述の仕方、SolidWorks を利用した STL ファイルの作成方法、ルートファイルからのインクルードの仕方、及び rviz を使った URDF モデルの可視化について説明しました。センサやホイール等のコンポーネントも、本節で説明した手順の通りに追加することができます。次節以降の作業を行う際にづまづいたときは、本節に戻って手順を確認してください。

2.6 caster_*_link の作成と追加

ロボットのモデルに自在キャスターを追加しましょう。実機用のモデルには必要のないリンクですが、後々シミュレーションを行うときに必要になるので、今のうちに追加しておきます。

^{*8} ROS1 では、ROS2 のように launch ファイルを変更する度にビルドする必要があります。ここでの作業は launch ファイルの存在をシステムに知らせるためのものです。

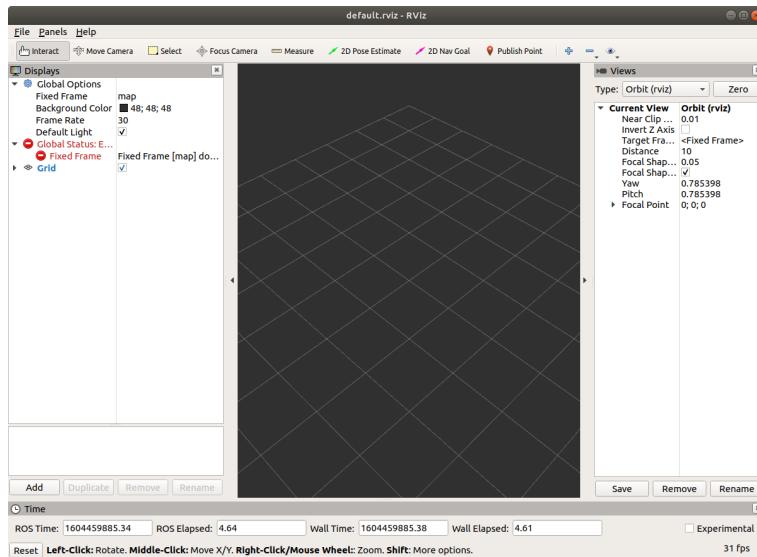


Fig.2.11 Screenshot of rviz Right After It Started

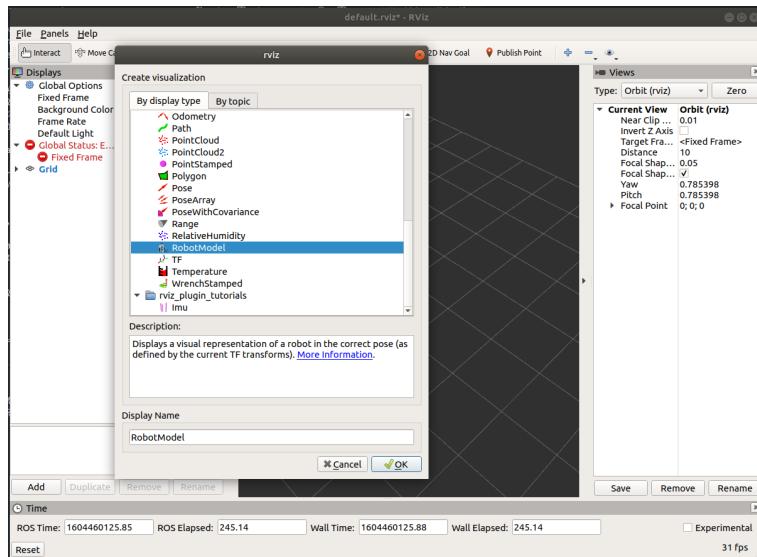


Fig.2.12 Add RobotModel to rviz

自在キャスターは2つのパーツから構成されます。1つ目はキャスターの車輪パーツ、2つ目は車輪を支えるサポートパーツです。シミュレーションを行うことを考えて、これらのパーツを別々のリンクとして定義することにします。

2.6.1 caster.xacro の作成

urdf/ディレクトリ以下にcaster/ディレクトリを作り、その中にcaster.xacroという名前のファイルを作成します。そして、コード2.19の内容を記述します。

Code 2.19 caster.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:macro name="caster" params="prefix parent *joint_origin">
    <joint name="${prefix}_caster_support_joint" type="fixed">
      <xacro:insert_block name="joint_origin"/>
      <parent link="${parent}"/>
      <child link="${prefix}_caster_support_link"/>
```

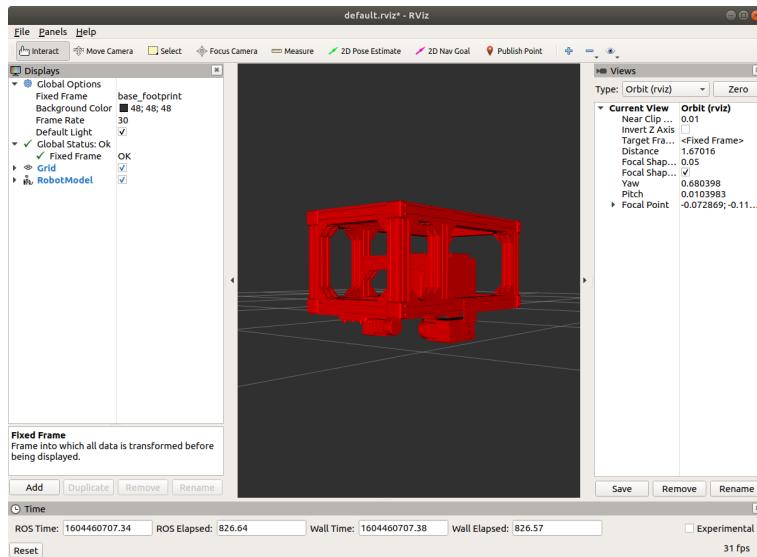


Fig.2.13 base_link is Correctly Displayed

```

</joint>

<link name="${prefix}_caster_support_link">
  <visual>
    <origin rpy="0.0 0.0 0.0"/>
    <geometry>
      <mesh filename="package://adamlr2_description/meshes/caster_support_link.STL"/>
    </geometry>
    <material name="red">
      <color rgba="0.0 0.0 0.0 1.0"/>
    </material>
  </visual>
</link>

<joint name="${prefix}_caster_wheel_joint" type="fixed">
  <origin xyz="-0.030 0.0 -0.072" rpy="0.0 0.0 0.0"/>
  <parent link="${prefix}_caster_support_link"/>
  <child link="${prefix}_caster_wheel_link"/>
</joint>

<link name="${prefix}_caster_wheel_link">
  <visual>
    <origin rpy="0.0 0.0 0.0"/>
    <geometry>
      <mesh filename="package://adamlr2_description/meshes/caster_wheel_link.STL"/>
    </geometry>
    <material name="red">
      <color rgba="0.0 0.0 0.0 1.0"/>
    </material>
  </visual>
</link>
</xacro:macro>
</robot>

```

caster.xacro では、\${prefix}_caster_support_link と \${prefix}_caster_wheel_link の 2 つのリンクと、それらを繋ぐ \${prefix}_caster_support_joint と \${prefix}_caster_wheel_joint の 2 つのジョイントを定義するマクロ caster を定義しています。caster マクロは 3 つの引数を取ります。このうち、prefix はリンクやジョイントの名前の接頭辞です。自在キャスターはロボットの前後に 1 つずつ存在しますが、それらは全く同じパーツなので、xacro ファイルを複数定義するのは無駄になります。そのため、接頭辞を引数にとって区別することで、1 つの xacro ファイルで 2 つの部品を定義できるようにしています。

base_link と \${prefix}_caster_support_link は \${prefix}_caster_support_joint によって接続されます。ジョイン

トタイプは fixed です。本来ならばこのジョイントは、Z 軸を回転中心とする連続回転ジョイントのタイプにする必要があります。しかし、実機用の URDF ファイルでこのような受動回転するコンポーネントのジョイントタイプを連続回転にすると、リンクの位置関係を取得できないために TF のエラーが発生します。そのため、実機用 URDF ではジョイントのタイプを fixed にしています。\${prefix}_caster_support_link と \${prefix}_caster_wheel_link を繋ぐジョイント \${prefix}_caster_wheel_joint も同様に固定タイプのジョイントとしています。

サポートパーツと車輪パーツの位置関係は一定なので、xacro ファイル内でローカルに \${prefix}_caster_wheel_joint の原点を設定してしまっています。一方で、自在キャスターがロボットボディのどこに取り付けられるかはユーザー次第なので、\${prefix}_caster_support_joint の原点はルートファイルから決定できるように、マクロのブロックパラメータとして設定しています。

2.6.2 STL ファイルの作成と割り当て

2.5.4 を参考に、3D-CAD ソフトから STL ファイルをエクスポートして meshes/ ディレクトリに配置してください。車輪パーツとサポートパーツは別ファイルとして書き出します。出力座標系は図 2.6.2 を参照してください。

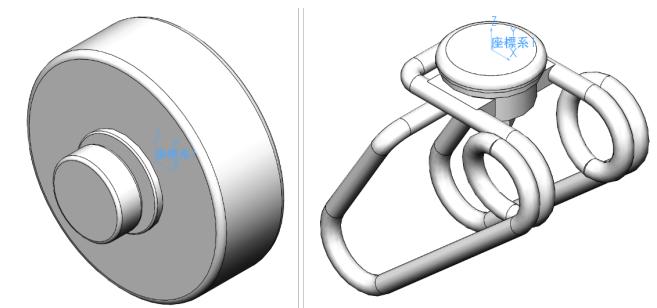


Fig.2.14 Coordinate System of Caster Parts

2.6.3 ルートファイルからインクルードする

作成したリンクをロボットモデルに組み込みましょう。robot.xacro にコード 2.20 のように追記します。

Code 2.20 Add Caster-Link to Robot Model

```
<?xml version="1.0"?>
<robot name="adamr2" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:include filename="$(find adamr2_description)/urdf/base/base.xacro"/>
  <xacro:include filename="$(find adamr2_description)/urdf/caster/caster.xacro"/>

  <link name="base_footprint"/>

  <xacro:base parent="base_footprint">
    <origin xyz="0.0 0.0 0.262"/>
  </xacro:base>

  <xacro:caster prefix="back" parent="base_link">
    <origin xyz="-0.275 0.0 -0.1498"/>
  </xacro:caster>

  <xacro:caster prefix="front" parent="base_link">
    <origin xyz="0.275 0.0 -0.1498" rpy="0 0 ${radians(180)}"/>
  </xacro:caster>
</robot>
```

caster.xacro ファイルをインクルードし、xacro:caster マクロを使って前後の自在キャスターを定義しています。ロボットの前方に付けるキャスターは、向きを 180 度変えるために origin 要素の rpy 属性でヨー角を設定しています。

2.7 lidar_link の作成と追加

RPLiDAR のリンクをロボットモデルに追加します。ロボットの部品の中でも、センサは特に座標変換が重要なコンポーネントです。センサのデータシートやマニュアルをよく読んで、適切な座標変換ができるようにならう。

2.7.1 RPLiDAR A2 の光学窓

RPLiDAR A2M6 のデータシートを見ると、RPLiDAR A2 の光学窓は図 2.7.1 のようになっており、センサの底面から 30.8 mm の高さに位置していることがわかります。URDF を記述するときは、lidar_link のリンク座標系の原点がこの位置になるようにしなければなりません。

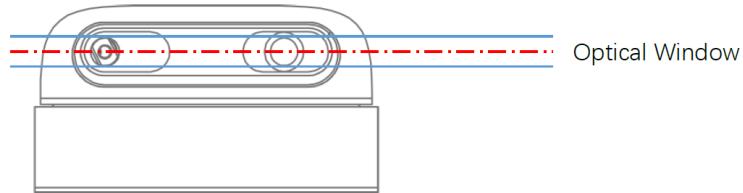


Fig.2.15 RPLiDAR A2 Optical Window

2.7.2 RPLiDAR A2 の座標系

RPLiDAR A2 は少々特殊な座標系を持っており、その向きは図 2.7.2 に示す通りになっています。

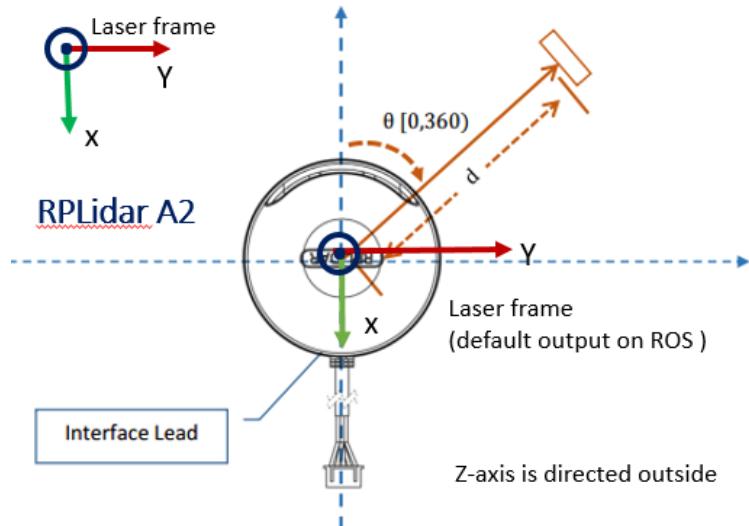


Fig.2.16 Coordinate System of RPLiDAR A2

図を見るとわかるのですが、ROS REP:103 で推奨されている座標系と一致していません (Z 軸周りに 180 度回転させた向きになっている)。そのため、モデルの見た目とセンサデータの座標変換とを両立させるために、ジョイントの向きを図 2.7.2 の通りにし、モデルの見た目を 180 度回転させる、という処理が必要になります。

2.7.3 STL ファイルの作成

STL ファイルの出力座標系は図 2.7.3 のようにします。座標系の原点はセンサ底面から 30.8 mm の高さに位置しています。

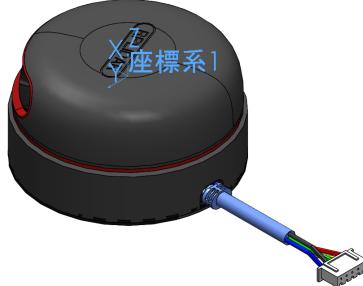


Fig.2.17 Output Coordinate System of RPLiDAR Model

2.7.4 lidar.xacro の記述

以上を踏まえて、lidar.xacro を記述します。urdf/ディレクトリ以下に lidar/ディレクトリを作り、その中にコード 2.21 の内容を記述したファイル lidar.xacro を作成します。

Code 2.21 lidar.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:macro name="lidar" params="parent visual_yaw_orientation *joint_origin">
    <joint name="lidar_joint" type="fixed">
      <xacro:insert_block name="joint_origin"/>
      <parent link="${parent}"/>
      <child link="lidar_link"/>
    </joint>

    <link name="lidar_link">
      <visual>
        <origin xyz="0.0 0.0 0.0" rpy="0 0 ${visual_yaw_orientation}"/>
        <geometry>
          <mesh filename="package://adaml2_description/meshes/lidar_link.STL"/>
        </geometry>
        <material name="blue">
          <color rgba="0.0 0.0 1.0 1.0"/>
        </material>
      </visual>
    </link>
  </xacro:macro>
</robot>
```

lidar マクロは引数として親リンクの名前、モデルの見た目の向き、そしてジョイントの原点座標のブロックを取ります。ジョイント原点座標とともにモデルの向きを取ることによって、RPLiDAR 特有の座標系に起因する問題を解決しています。

2.7.5 ルートファイルからインクルードする

lidar_link をロボットモデルに取り込みましょう。コード 2.22 のように robot.xacro を編集します。lidar マクロの引数 visual_yaw_orientation に、radians マクロを使って弧度法で角度を渡しています。

Code 2.22 Add lidar_link to Robot Model

```
<?xml version="1.0"?>
```

```

<robot name="adamlr2" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:include filename="$(find adamlr2_description)/urdf/base/base.xacro"/>
  <xacro:include filename="$(find adamlr2_description)/urdf/caster/caster.xacro"/>
  <xacro:include filename="$(find adamlr2_description)/urdf/lidar/lidar.xacro"/>

  <!-- base_footprint -->
  <link name="base_footprint"/>

  <!-- base_link -->
  <xacro:base parent="base_footprint">
    <origin xyz="0.0 0.0 0.262"/>
  </xacro:base>

  <!-- front caster -->
  <xacro:caster prefix="back" parent="base_link">
    <origin xyz="-0.275 0.0 -0.1498"/>
  </xacro:caster>

  <!-- back caster -->
  <xacro:caster prefix="front" parent="base_link">
    <origin xyz="0.275 0.0 -0.1498" rpy="0 0 ${radians(180)}"/>
  </xacro:caster>

  <!-- lidar -->
  <xacro:lidar parent="base_link" visual_yaw_orientation="${radians(180)}">
    <origin xyz="0.275 0 0.177" rpy="0 0 ${radians(180)}"/>
  </xacro:lidar>
</robot>

```

2.7.6 rviz による可視化

センサを取り付けたので、rviz で可視化して、正しい位置に取り付けられているかどうか確認します。2.5.6 を参考に、URDF モデルを rviz で可視化します。

とはいっても、可視化の度に RobotModel を追加して、基準フレームを base_footprint にして… という作業を繰り返すのは面倒です。そこで、rviz の設定ファイルを保存しておいて、可視化の度に読み込むようにしてみましょう。rviz を起動し、ロボットモデルの可視化が正しくできている状態 (RobotModel を追加して base_footprint を基準フレームに設定している状態) にします。図 2.7.6 のように、画面左上のツールバーから「File」→「Save Config As」を選択します。

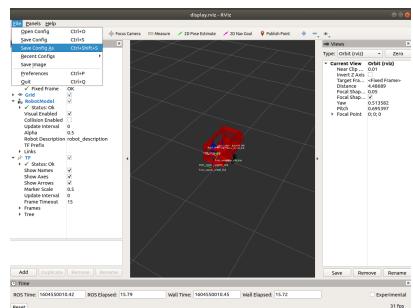


Fig.2.18 Save rviz Config File

adamlr2_description パッケージの中に設定ファイルを保存します。保存場所はどこでもいいですが、わかりやすいように rviz/ という名前のディレクトリを作り、display.rviz という名前で保存しましょう。

この設定ファイルを rviz 起動時に読み込ませることで、保存しておいた設定を復元することができます。rviz に設定ファイルを渡して起動するには、launch ファイルを少し編集する必要があります。コード 2.23 のように display.launch ファイルを編集します。

Code 2.23 rviz Load Config File at Startup

```

<launch>
  <arg name="model" default="$(find adamlr2_description)/urdf/robot.xacro"/>

```

```

<arg name="rvizconfig" default="$(find adamr2_description)/rviz/display.rviz"/>

<param name="robot_description" command="$(find xacro)/xacro $(arg model)" />

<node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"
      />
<node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)"/>
</launch>

```

launch ファイルの arg タグを追加して、新しい変数 rvizconfig を定義しています。これは先程保存した rviz の設定ファイルへのパスです。そして、rviz を起動する node タグの arg 属性に設定ファイルを読み込ませるオプションを追加しています。

この launch ファイルを実行することで、適切な設定がなされた状態でモデルが表示されます。このように rviz の設定ファイルを保存しておき、実行時に読み込ませることで、作業を効率よく進めることができます。テクニックとして覚えておいて損はないでしょう。

lidar_link を追加した段階でのロボットの見た目は図 2.7.6 のようになります。RPLiDAR が正しい位置・向きに取り付けられていることがわかります。リンクの座標軸が表示されていますが、これは TF を表示させているからです。「Add」ボタンから追加することができます。

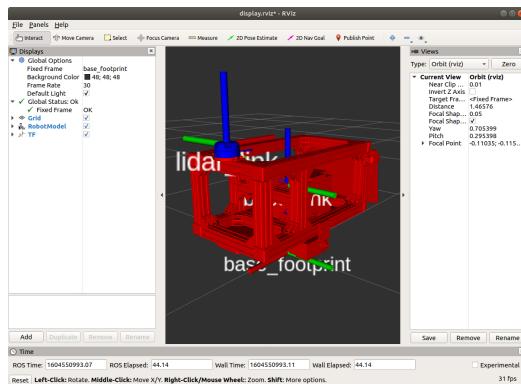


Fig.2.19 Screenshot of rviz

2.8 wheel_link の作成と追加

最後に wheel_link とそのジョイントを追加していきます。wheel_link を定義するために 2 つのファイルを準備します。wheel.xacro と transmission.xacro の 2 つです。

2.8.1 wheel.xacro の作成

まずは wheel.xacro を記述していきます。urdf/ディレクトリ以下に wheel/ディレクトリを作成し、wheel.xacro という名前のファイルを作成します。そして、コード 2.24 のように記述します。

Code 2.24 wheel.xacro

```

<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:macro name="wheel" params="prefix parent *joint_origin *joint_axis">
    <joint name="${prefix}_wheel_joint" type="continuous">
      <xacro:insert_block name="joint_origin"/>
      <parent link="${parent}"/>
      <child link="${prefix}_wheel_link"/>
      <xacro:insert_block name="joint_axis"/>
    </joint>

    <link name="${prefix}_wheel_link">
      <visual>
        <origin rpy="0.0 0.0 0.0"/>
        <geometry>

```

```

<mesh filename="package://adamr2_description/meshes/wheel_link.STL"/>
</geometry>
<material name="black">
    <color rgba="0.0 0.0 0.0 1.0"/>
</material>
</visual>
</link>
</xacro:macro>
</robot>

```

wheel.xacro の内容は他のリンクとほとんど変わり映えしません。ジョイントとリンクを定義しているだけです。ホイールを繋ぐジョイントはタイプを continuous にします。

2.8.2 transmission.xacro の作成

ホイールジョイントに対する ros_control の設定を行うマクロは transmission.xacro で定義します。wheel.xacro と同じディレクトリに transmission.xacro という名前のファイルを作成し、コード 2.25 のような内容を記述します。

Code 2.25 transmission.xacro

```

<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
    <xacro:macro name="wheel_trans" params="prefix">
        <transmission name="${prefix}_wheel_trans">
            <type>transmission_interface/SimpleTransmission</type>
            <joint name="${prefix}_wheel_joint">
                <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
            </joint>

            <actuator name="${prefix}_wheel_motor">
                <mechanicalReduction>1</mechanicalReduction>
            </actuator>
        </transmission>
    </xacro:macro>
</robot>

```

このマクロではホイールのジョイントに対して transmission 要素を追加しています。transmission 要素には type, joint, actuator の 3 つの要素を持っています。この URDF は ros_control のコントローラの 1 つである diff_drive_controller から利用されることを想定しているので、type は transmission_interface/SimpleTransmission を、joint の hardwareInterface 要素には hardware_interface/VelocityJointInterface を設定しています。actuator 要素の mechanicalReduction には、ロボットのアクチュエータの減速比を設定します。ADAMR2 ではギヤード BLDC モータを使用していますが、ギア比の計算は YP-Spur が行ってくれるため、ここで設定する必要はありません。ギヤ比は 1 に設定しておきましょう。

2.8.3 ルートファイルからインクルードする

ホイールをロボットに組み込みます。robot.xacro を更に編集し、コード 2.26 のようにします。xacro:include タグを使って wheel.xacro と transmission.xacro を読み込むことを忘れないようにしてください。

Code 2.26 Add wheel_link to Robot Model

```

<?xml version="1.0"?>
<robot name="adamr2" xmlns:xacro="http://ros.org/wiki/xacro">
    <xacro:include filename="$(find adamr2_description)/urdf/base/base.xacro"/>
    <xacro:include filename="$(find adamr2_description)/urdf/caster/caster.xacro"/>
    <xacro:include filename="$(find adamr2_description)/urdf/lidar/lidar.xacro"/>
    <xacro:include filename="$(find adamr2_description)/urdf/wheel/wheel.xacro"/>
    <xacro:include filename="$(find adamr2_description)/urdf/wheel/transmission.xacro"/>

    <!-- base_footprint -->
    <link name="base_footprint"/>

```

```

<!-- base_link -->
<xacro:base parent="base_footprint">
  <origin xyz="0.0 0.0 0.262"/>
</xacro:base>

<!-- front caster -->
<xacro:caster prefix="front" parent="base_link">
  <origin xyz="0.275 0.0 -0.1498" rpy="0 0 ${radians(180)}"/>
</xacro:caster>

<!-- back caster-->
<xacro:caster prefix="back" parent="base_link">
  <origin xyz="-0.275 0.0 -0.1498"/>
</xacro:caster>

<!-- lidar -->
<xacro:lidar parent="base_link" visual_yaw_orientation="${radians(180)}">
  <origin xyz="0.275 0 0.177" rpy="0 0 ${radians(270)}"/>
</xacro:lidar>

<!-- left wheel -->
<xacro:wheel prefix="left" parent="base_link">
  <origin xyz="0.0 0.1915 -0.1845"/>
  <axis xyz="0 1 0"/>
</xacro:wheel>
<xacro:wheel_trans prefix="left"/>

<!-- right wheel -->
<xacro:wheel prefix="right" parent="base_link">
  <origin xyz="0.0 -0.1915 -0.1845"/>
  <axis xyz="0 1 0"/>
</xacro:wheel>
<xacro:wheel_trans prefix="right"/>
</robot>

```

`xacro:wheel` を呼び出してホイールを定義しています。ここで重要なのが、ホイールの回転軸は「ロボットが直進する方向を正とする」ようにしなければならないということです。コード 2.26 を見るとわかるように、左右どちらの車輪も、Y 軸正方向を回転軸に設定しています。これは `diff_drive_controller` を使用する都合で発生するものです。^{*9}

2.8.4 rviz による可視化

駆動パーツであるホイールを追加したので、`rviz` で可視化して、ついでにジョイントを動かしてみましょう。`joint_state_publisher_gui` パッケージを使用すれば、スライダーを使ってジョイントを動かすことができます。`joint_state_publisher_gui` パッケージはデフォルトで入っていない場合があるので、追加でインストールします。

Code 2.27 Install `joint_state_publisher_gui`

```

sudo apt update
sudo apt install ros-melodic-joint-state-publisher-gui

```

`display.launch` に `joint_state_publisher` と `joint_state_publisher_gui` を起動する文を追加します。

Code 2.28 `display.launch`

```

<launch>
  <arg name="model" default="$(find adamr2_description)/urdf/robot.xacro"/>
  <arg name="rvizconfig" default="$(find adamr2_description)/rviz/display.rviz" />

  <param name="robot_description" command="$(find xacro)/xacro $(arg model)" />

```

^{*9} 後述しますが³、YP-Spur における車輪の軸はこのようになっていないため、結局ドライバノード側で回転速度を反転しなければなりません。にも関わらずこのようにしているのは、できるだけシミュレーション環境と互換性を持たせるためです。

```

<node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"
      "/>
<node name="joint_state_publisher_gui" pkg="joint_state_publisher_gui" type="joint_state_publisher_gui"/>
<node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true"/>
</launch>

```

これを実行すると、図 2.8.4 のように、rviz にロボットモデルが表示され、また joint_state_publisher_gui のウィンドウが現れます。joint_state_publisher_gui のウィンドウのスライダーをマウスで動かすことによって、rviz 上のロボットのホイールも動きます。ジョイントの角度を正の数値にすると、ロボットが前進する方向へホイールが動くことを確認してください。

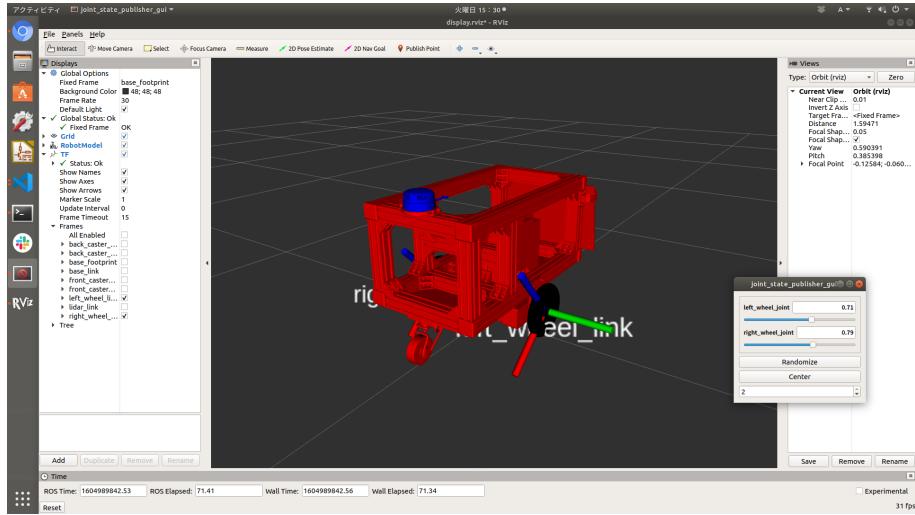


Fig.2.20 joint_state_publisher_gui

第3章

デバイス設定

3.1 F310/F710 ゲームパッドを ROS で利用する

3.1.1 製品の概要

Logicool Gamepad F310/F710 は、Logicool から発売されている USB 接続の PC 用ゲームパッドです。



Fig.3.1 Logicool F710 Gamepad

有線接続の F310 は 2310 円、無線接続の F710 は 4950 円と安価に入手することができ、ロボットの操縦用コントローラとして使用することもできます。

3.1.2 ゲームパッドの入力モード

310/F710 ゲームパッドは 2 つの入力モードを持っており、物理スイッチによってモードを切り替えることができます。

- DirectInput モード：他のゲームパッドと同じ挙動を示すモード
- XInput モード：Xbox360 コントローラを PC に繋いだときと同じ挙動を示すモード

モード切替スイッチは、F310 には本体背面に、F710 は本体側面上部に付いています。



Fig.3.2 Mode Selector Switch for F710

ROS パッケージの joy^{*1} から F310/F710 ゲームパッドを使用するときは、DirectInput モードを使用する必要があります。

3.1.3 DirectInput モードのキーマッピング

DirectInput モードで ROS に接続した際の F310/F710 ゲームパッドのキーマッピングを表 3.1 及び表 3.2 に示します。

sensor_msgs/Joy^{*2}型メッセージでは、ゲームパッドの各軸・各ボタンの信号の値が axes と buttons の 2 つのリストに格納されます。各リストに対してインデックスを指定することで、対応するボタン・軸のデータを得ることができます。

Table3.1 Button Mapping of F310/F710 Gamepad in ROS

Buttons	X	A	B	Y	LB	RB	LT	RT	BACK	START	LeftStick	RightStick
index	0	1	2	3	4	5	6	7	8	9	10	11

Table3.2 Axis Mapping of F310/F710 Gamepad in ROS

Axis	Left Horiz.	Left Vert.	Right Horiz.	Right Vert.	Arrow Horiz.	Arrow Vert.
index	0	1	2	3	4	5

3.1.4 ゲームパッド起動用ファイルを準備する

F310/F710 ゲームパッドを ROS に接続して、ゲームパッドからロボットへ速度指令を出せるようにしてみましょう。ゲームパッドから速度指令を出せるようにするには、以下の 2 つのパッケージに含まれるノードが必要になります。

- joy
- teleop_twist_joy

joy パッケージは汎用ゲームパッドのための ROS ドライバです。teleop_twist_joy^{*3} は、sensor_msgs/Joy メッセージを geometry_msgs/Twist^{*4} メッセージに変換するためのパッケージです。Twist 型メッセージで速度指令を受け取るタイプのロボットをゲームパッドから動かす際によく使われているパッケージで、ADAMR2 でもこのパッケージを使用しています。

この 2 つのパッケージのノードを同時に起動する launch ファイルを joy.launch とし、adaml2 Bringup というパッケージに保存することにします。teleop_twist_joy パッケージを利用するには適切なコンフィグファイルが必要となります。それも adaml2 Bringup パッケージに置いておくことにします。

まずはパッケージを作ります。コード 3.1 のコマンドを実行して、ROS ワークスペースに adaml2 Bringup パッケージを作ります。

Code 3.1 Create adaml2 Bringup Package

```
cd ~/catkin_ws/src
catkin create pkg adaml2 Bringup
```

adaml2 Bringup パッケージの中には 2 つのディレクトリを作ります。launch/ディレクトリと config/ディレクトリです。まずは launch ファイルを準備します。launch/ディレクトリを作成し、joy.launch という名前の launch ファイルを作成します。そして、コード 3.2 のような内容を記述します。

*1 <http://wiki.ros.org/joy>

*2 http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/Joy.html

*3 http://wiki.ros.org/teleop_twist_joy

*4 http://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/Twist.html

Code 3.2 joy.launch

```

<launch>
  <arg name="joy_dev" default="/dev/input/js0"/>
  <arg name="config_file_path"
        default="$(find adamr2_bringup)/config/f310.config.yml"/>

  <group ns="/adaml2/diff_drive_controller">
    <!-- joy node -->
    <node pkg="joy" type="joy_node" name="joy_node">
      <param name="dev" value="$(arg joy_dev)"/>
      <param name="deadzone" value="0.3" />
      <param name="autorepeat_rate" value="20"/>
    </node>

    <!-- teleop_twist_joy node -->
    <node pkg="teleop_twist_joy" name="teleop_twist_joy" type="teleop_node">
      <rosparam command="load" file="$(arg config_file_path)"/>
    </node>
  </group>
</launch>

```

launch ファイルのパラメータとして、joy_dev と config_file_path を宣言しています。joy_dev はゲームパッドのデバイスファイル名で、ゲームパッドデバイスを 1 つしか接続していない場合は/dev/input/js0 になります。config_file_path は teleop_twist_joy パッケージのノードに与えるパラメータを記述したコンフィグファイルのパスです。まだ作成していないので、このままでは launch ファイルを実行することはできません。

この launch ファイルでは、group ns タグで名前空間を設定しています。joy_node と teleop_node は名前空間/adaml2/diff_drive_controller の下に置かれ、トピックやパラメータの名前が変化します。名前空間の設定を行う理由は、対向 2 輪ロボット用のコントローラである diff_drive_controller が、名前空間の下に置かれた速度指令トピックを受信するようになっているためです。

次に、teleop_node のためのコンフィグファイルを記述します。ROS では、ノードに与えるパラメータを 1 つのファイルにまとめて記述する際に、YAML 形式のファイルを使用します。adaml2_bringup パッケージディレクトリ直下に config/ディレクトリを作り、f310.config.yml^{*5} という名前のファイルを作成します。そして、コード 3.3 のような内容を記述します。

Code 3.3 f310.config.yml

```

axis_linear: 1
scale_linear: 0.3
scale_linear_turbo: 0.5

axis_angular: 0
scale_angular: 0.94
scale_angular_turbo: 1.57

enable_button: 1
enable_turbo_button: 2

```

この設定では、左アナログスティックの縦軸が直進速度、横軸が旋回速度に対応するようになっています。また、押しているボタンによって速度のスケールが変わるようにになっており、A ボタンは通常のスケール、B ボタンでターボスケールになるようになっています。スケールの具体的な値は scale_linear と scale_angular で指定しています。

より詳細なパラメータの設定方法は、teleop_twist_joy の ROS Wiki のページを参照してください。

3.1.5 ゲームパッドを使ってみる

launch ファイルおよび設定ファイルの準備が終わったので、実際にゲームパッドを ROS 上で使ってみます。

^{*5} YAML 形式のファイルの拡張子は.yaml と.yml が使え(てしまい)ますが、ロボットのシステムを組む際はどちらかに統一した方が良いです。ここでは.yaml で統一していますが、ROS で SLAM をしたときに保存されるマップデータの YAML ファイル拡張子が.yaml のなので、.yaml で統一した方が良いのかもしれません。

PC にゲームパッドを接続し、正しく認識されているかどうかを確認してください。デバイスファイルが存在するかどうかのチェックは、コード 3.4 のコマンドで行うことができます。

Code 3.4 Check Device File of Gamepad

```
ls /dev/input/js*
```

「/dev/input/js0」のような表示がターミナルに表示されれば問題ありません。また、Ubuntu でゲームパッドが正しく使えるかどうかをテストするソフトウェアである「jstest-gtk」があります。ROS から利用する前に、このソフトウェアでチェックをするとよいでしょう。jstest-gtk は apt コマンドでインストールすることができます。インストールした後、ターミナルで「jstest-gtk」と実行すればソフトウェアが起動します。

Code 3.5 Install jstest-gtk

```
sudo apt update  
sudo apt install jstest-gtk
```

ゲームパッドのチェックが終わったら、ROS からゲームパッドを利用してみましょう。パッケージをビルドして環境変数を読み込んだら、launch ファイルを実行してみます。

Code 3.6 launch joy.launch

```
cd ~/catkin_ws  
catkin build  
source /opt/ros/melodic/setup.bash  
source ~/catkin_ws/devel/setup.bash  
roslaunch adamr2_bringup joy.launch
```

launch ファイルが正しく実行された場合、2つのノードが起動し、/adamr2/diff_drive_controller/joy トピックや/adamr2/diff_drive_controller/cmd_vel トピックなどの配信が開始されるはずです。

3.2 RPLiDAR A2 を利用する

3.2.1 製品の概要

RPLiDAR A2 は、SLAMTEC 社から発売されている安価な 360° 計測可能な LiDAR センサです。



Fig.3.3 SLAMTEC RPLiDAR A2

RPLiDAR シリーズの ROS ドライバは既に用意されているため^{*6}、購入後すぐに利用することができます。

3.2.2 launch ファイルの用意

RPLiDAR A2 を起動するための launch ファイルを作成します。adamr2_bringup パッケージの launch/ディレクトリに、rplidar_a2.launch という名前のファイルを作成し、コード 3.7 のように記述します。

^{*6} <http://wiki.ros.org/rplidar>

Code 3.7 rplidar_a2.launch

```
<launch>
  <arg name="device"    default="/dev/ttyUSB0"/>
  <arg name="baudrate"  default="115200"/>
  <arg name="frame_id"  default="lidar_link"/>
  <arg name="inverted"  default="false"/>
  <arg name="angle_compensate" default="true"/>

  <group ns="/adamr2">
    <node name="rplidarNode" pkg="rplidar_ros" type="rplidarNode" output="screen">
      <param name="serial_port"      type="string" value="$(arg device)"/>
      <param name="serial_baudrate"  type="int"   value="$(arg baudrate)"/>
      <param name="frame_id"        type="string" value="$(arg frame_id)"/>
      <param name="inverted"        type="bool"  value="$(arg inverted)"/>
      <param name="angle_compensate" type="bool"  value="$(arg angle_compensate)"/>
    </node>
  </group>
</launch>
```

RPLiDAR の ROS ドライバノードである rplidarNode は、以下の 6 つのパラメータを取ります。

1. serial_port
2. serial_baudrate
3. frame_id
4. inverted
5. angle_compensate
6. scam_mode

serial_port は RPLiDAR のデバイスファイルを指定します。RPLiDAR A2 のデバイスファイルは、他に USB デバイスを繋いでいなければ「/dev/ttyUSB0」になります。serial_baudrate はシリアル通信のボーレートです。RPLiDAR A2 ならばボーレートは 115200 固定です。frame_id は LiDAR の TF フレーム名を指定します。[2.7](#) で URDF を記述した際に、「lidar_link」という名前を付けていたので、ここではそれを指定します。

また、ノードに名前空間を設定しています。名前空間の設定は必須ではありませんが、[*7](#) ここでは scan メッセージにも名前空間を指定することにします。

3.2.3 RPLiDAR を使ってみる

早速作成した launch ファイルを実行して、RPLiDAR を使ってみましょう。RPLiDAR を PC に接続して、launch ファイルを実行します。

Code 3.8 Launch rplidar_a2.launch

```
roslaunch adamr2Bringup rplidar_a2.launch
```

無事にノードが起動したら、rviz を起動してセンサデータを確認してみましょう。固定フレームに「lidar_link」を指定します。TF が配信されていないのでプルダウンメニューには表示されません。名前を直接入力しましょう。次に左下の「Add」ボタンから「LaserScan」を追加し、トピック名に「adamr2/scan」を指定します。センサデータが見えるようになれば OK です。

*7 複数台のロボットを制御したりしないのであれば、むしろデフォルトのままの方が混乱が起きにくいです。

第4章

コントローラの実装

本章ではロボットの動作の制御を行うためのコントローラを作成します。ロボットのコントローラの構築には、ROSのコントローラフレームワークである `ros_control` を使用します。`ros_control` を使うためにはいくつかの準備が必要となるため、結構面倒だったり難しかったりしますが、完成した時の便利さはかなりのものになります。

4.1 `ros_control` とは

`ros_control`^{*1} とは、ロボットのジョイントアクチュエータを制御するためのコントローラインターフェース、コントローラマネージャ、トランスマッショング、ハードウェアインターフェース、制御ツールボックスなどを含むパッケージ群のことです。もともと Willow Garage 社の PR2 で使用されていたコントローラを一般的なロボットに使用できるように書き直したもので、ロボットに依存しない方法でリアルタイム性能を引き出すコントローラとして開発されました。`ros_control` の枠組みを使用することで、ハードウェアを抽象化してプログラミングを簡単にしたり、共通のコントローラを使って実機のロボットとシミュレーション上のロボットを動かしたりするできるようになります。

`ros_control` のシステムの概要図を ROS Wiki から引用します。

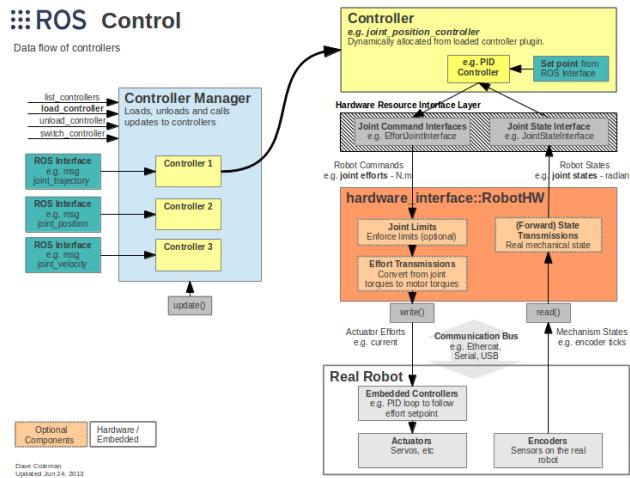


Fig.4.1 Diagram of `ros_control`

ADAMR2 では、`ros_control` に準拠したコントローラである `diff_drive_controller`^{*2} を使用して、ロボットを制御するシステムを構築しています。次節以降でシステムの構築の手順を説明していきます。

*1 http://wiki.ros.org/ros_control

*2 http://wiki.ros.org/diff_drive_controller

4.2 コントローラの構築に必要なもの

`diff_drive_controller` を使用してロボットの制御システムを作るには、以下のものが必要になります。

1. ロボットのアクチュエータの情報を記述した URDF ファイル
2. コントローラのコンフィグファイル
3. RobotHW クラスを継承して作ったハードウェアドライバノード (C++ で記述)
4. コントローラやドライバノードを起動する launch ファイル

ロボットの URDF ファイルは既に作成しました。これから作る必要があるのは、コンフィグファイルや launch ファイル、そして C++ で記述したノードです。コンフィグファイルや launch ファイルは `adamlr2_control` というパッケージを作り、その中に配置します。また、ドライバノードは `adamlr2_driver` というパッケージを作り、その中で開発します。

4.3 adamlr2_control パッケージの準備

4.3.1 パッケージ作成

ロボットのコントローラに関連するファイルを格納しておくパッケージ `adamlr2_control` を作成します。このパッケージはビルド時依存パッケージを持たないので^{*3}、オプションは必要ありません。

Code 4.1 Create `adamlr2_control` Package

```
catkin create pkg adamlr2_control
```

`adamlr2_control` パッケージのディレクトリ構成はコード 4.1 のようになります。

Code 4.2 Directory Structure of `adamlr2_control`

```
adamlr2_control/
├── config/
├── launch/
├── scripts/
└── package.xml
    └── CMakeLists.txt
```

`config`/ディレクトリには、`diff_drive_controller` や YP-Spur に与えるパラメータを記述したファイルを配置します。`launch`/ディレクトリにはコントローラを起動するための launch ファイルを配置します。`scripts`/ディレクトリには YP-Spur を実行するためのシェルスクリプトを格納します。

4.3.2 package.xml の編集

`package.xml` に依存パッケージを記述しておきます。`adamlr2_control` パッケージにはビルド時依存パッケージが無いので `CMakeLists.txt` は編集しなくてもよいのですが、他のパッケージのノードを呼び出したりする都合上、実行時依存パッケージは書いておく必要があります。^{*4} `package.xml` をコードのように編集します。`maintainer` タグや `license` タグは適切な情報を記述してください。

Code 4.3 `package.xml` in `adamlr2_control`

```
<?xml version="1.0"?>
<package format="2">
  <name>adamlr2_control</name>
  <version>0.0.0</version>
  <description>The adamlr2_control package</description>

  <maintainer email="hoge@hogehoge.com">hoge</maintainer>
```

^{*3} 実行時依存パッケージはあるので `package.xml` を編集しなければなりません。

^{*4} 実際には実行時依存パッケージを書かなくても動きます。依存パッケージを書くのは、アプリケーション実行時にパッケージが不足してシステムが起動できない等の問題を極力回避するためです。

```

<license>BSD</license>

<buildtool_depend>catkin</buildtool_depend>

<exec_depend>xacro</exec_depend>
<exec_depend>diff_drive_controller</exec_depend>
<exec_depend>joint_state_controller</exec_depend>
<exec_depend>controller_manager</exec_depend>
<exec_depend>robot_state_publisher</exec_depend>
<exec_depend>adamlr2_description</exec_depend>
<exec_depend>adamlr2_driver</exec_depend>
<exec_depend>ypspur</exec_depend>
</package>

```

これでパッケージの用意は完了です。次は launch ファイルの作成を行います。

4.3.3 コントローラを起動するための launch ファイル

ロボットのコントローラを起動する launch ファイルを作成します。「この launch ファイルを実行することで、ロボットの駆動系が立ち上がり、操縦可能になる」ような launch ファイルを作成します。

adamlr2_control パッケージ以下に、launch/ディレクトリを作成し、adamlr2_control.launch という名前のファイルを作成します。そして、コード 4.4 のように記述します。

Code 4.4 adamlr2_control.launch

```

<launch>
  <arg name="model" default="$(find adamlr2_description)/urdf/robot.xacro"/>
  <arg name="ypspur_params" default="$(find adamlr2_control)/config/adamlr2.param"/>
  <arg name="ypspur_dev" default="/dev/ttyACM0"/>

  <param name="robot_description" command="$(find xacro)/xacro $(arg model)"/>

  <rosparam file="$(find adamlr2_control)/config/controller.yaml" command="load"/>

  <group ns="adamlr2">
    <node name="controller_spawner" type="spawner" pkg="controller_manager"
      respawn="false" output="screen"
      args="joint_state_controller diff_drive_controller"/>

    <node name="robot_state_publisher" type="robot_state_publisher"
      pkg="robot_state_publisher" respawn="false"
      output="screen"/>

    <node name="ypspur_launcher" type="ypspur_launcher.sh"
      pkg="adamlr2_control" output="screen"
      args="$(arg ypspur_params) $(arg ypspur_dev)"/>

    <node name="adamlr2_driver_node" type="adamlr2_driver_node" pkg="adamlr2_driver"
      output="screen"/>
  </group>
</launch>

```

この launch ファイルは 3 つの ROS ノードと 1 つのシェルスクリプトを実行します。1 つ目は controller_manager パッケージの spawner ノードです。このノードを使って、joint_state_controller と diff_drive_controller の 2 つのコントローラをロード・実行しています。

2 つ目は robot_state_publisher です。robot_description というパラメータを読んでロボットのモデルを把握し、更に joint_state というトピックを受け取ってロボットの TF を更新します。robot_description パラメータは launch ファイル上部の`<param>`タグで設定しており、ここでは xacro ファイルを URDF に展開して robot_description パラメータに登録しています。joint_state トピックは joint_state_controller によって配信されます。

3 つ目は adamlr2_driver パッケージの adamlr2_driver_node です。diff_drive_controller によって計算された各車輪の回転速度を YP-Spur に伝える役目を持つノードです。このノードはこれから作るので、現時点ではこ

の launch ファイルを実行することはできません。

最後に, `ypspur_launcher.sh` というシェルスクリプトを実行します。これは, C++ プログラムが YP-Spur とやり取りするのに必要なアプリケーション `ypspur-coordinator` を実行するためのスクリプトです。 `ypspur-coordinator` の実行には, YP-Spur に与えるパラメータファイルとモータードライバのデバイスファイルを引数として指定する必要があるので, シェルスクリプトに引数としてそれらを与えています。 launch ファイル上部の `arg` タグで宣言している `ypspur_params` と `ypspur_dev` がそれに当たります。

4.3.4 各種コントローラのためのコンフィグファイル

`diff_drive_controller` は, `geometry_msgs/Twist` 型トピックを受け取って各車輪に与える速度を計算します。そのためには車輪の半径やトレッド長さ等の情報が必要になります。それらの情報は ROS パラメータとして与えることになるのですが、多くのパラメータを記述することになるので、1つのファイルにまとめて記述できた方が好ましいです。

`adamr2_control` パッケージ以下に config/ディレクトリを作成し、`controller.yml` という名前のファイルを作成します。そして、コード 4.5 のように記述します。

Code 4.5 controller.yml

```
admr2:
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  diff_drive_controller:
    type: "diff_drive_controller/DiffDriveController"
    left_wheel: 'left_wheel_joint'
    right_wheel: 'right_wheel_joint'
    publish_rate: 50

  pose_covariance_diagonal: [0.001, 0.001, 1000000.0, 1000000.0, 1000000.0, 10.0]
  twist_covariance_diagonal: [0.001, 0.001, 1000000.0, 1000000.0, 1000000.0, 10.0]

  wheel_separation: 0.383
  wheel_separation_multiplier: 1.0
  wheel_radius: 0.0775
  left_wheel_radius_multiplier: 1.0
  right_wheel_radius_multiplier: 1.0

  cmd_vel_timeout: 0.5

  base_frame_id: base_footprint
  odom_frame_id: odom

  linear:
    x:
      has_velocity_limits: true
      max_velocity: 0.9
      min_velocity: -0.9
      has_acceleration_limits: true
      max_acceleration: 1.5
      min_acceleration: -1.5

  angular:
    z:
      has_velocity_limits: true
      max_velocity: 3.14
      min_velocity: -3.14
      has_acceleration_limits: true
      max_acceleration: 6.28
      min_acceleration: -6.28
```

このコンフィグファイルでは、`joint_state_controller` と `diff_drive_controller` の2つのノードに対するパラ

メータの設定を行っています。2つのノードは adamr2 という名前空間の下に置かれます。^{*5}
left_wheel および right_wheel パラメータには、URDFで設定した車輪のジョイント名を指定します。[4.4.2](#) で名付けた通りの名前を指定してください。

wheel_radius で車輪の半径を、wheel_separation でトレッド長さを指定しています。対象のロボットのハードウェア仕様に合わせて設定してください。ADAMR2 ではトレッド長さ 383 mm、ホイール半径 77.5 mm なので、メートル単位にしてそれぞれ 0.383, 0.0775 と設定しています。

wheel_separation_multiplier, left_wheel_radius_multiplier および right_wheel_radius_multiplier は、車輪に関するパラメータの補正係数です。ホイール半径やトレッド長さの実際の値が、設計値よりもズレている場合に、これらの係数を使って調整を行います。これらの補正係数は ROS の Dynamic Reconfigure ^{*6} という仕組みによってノード実行時に動的に数値を変更することができるため、実機を調整するのにうってつけです。wheel_separation 等の数値を直接変更するのではなく、補正係数を使ってオドメトリの調整を行うことをお勧めします。

base_frame_id にはオドメトリの基準となるロボットのフレーム名を指定します。ROS REP:105 では base_link にするよう推奨していますが、ADAMR2 では base_footprint を使っているため、そのように設定します。odom_frame_id にはオドメトリ座標系を表す TF フレームの名前を指定します。ROS REP:105 に従って、odom という名前を付けておきます。

ファイルの末尾にあるのはロボットのソフトウェア的な速度・加速度制限です。直進速度と旋回速度に対してそれぞれ速度制限と加速度制限、更には躍度制限を付けることができます。ADAMR2 ではハードウェアの仕様に合わせて、速度制限と加速度制限を適用しています。

diff_drive_controller が取る ROS パラメータについてのより詳細な情報は、ROS Wiki を参照してください。

4.3.5 ypspur_launcher.sh の作成

ypspur_launcher.sh は、ROS のプログラムとモータードライバとの通信に必要な YP-Spur のソフトウェア ypspur-coordinator を起動するためのシェルスクリプトです。adamr2_control ディレクトリ以下に scripts/ ディレクトリを作成し、ypspur_launcher.sh という名前のファイルを作成します。そして、コード [4.6](#) のように記述します。

Code 4.6 ypspur_launcher.sh

```
#!/bin/bash

# Execute yp-spur in background
# Argument 1 is the path of parameter file
# Argument 2 is the device file of the motor driver
ypspur-coordinator -p $1 -d $2

echo "Robot parameters [$1] loaded."
```

中身は単純で、コマンド引数として YP-Spur のパラメータファイルとモータードライバのデバイスファイルを受け取り、ypspur-coordinator を実行するだけです。コメント文に書かれている通り、\$1 はパラメータファイル、\$2 はデバイスファイルを表します。このシェルスクリプトは adamr2_control.launch ファイルから呼び出されます。

4.4 adamr2_driver の作成

いよいよモータードライバと通信するソフトウェアドライバノード adamr2_driver_node を作成します。ノードの作成には C++ 言語を使用するので、ある程度の C++ の知識が必要になります。

4.4.1 パッケージの作成

adamr2_driver_node を開発するためのパッケージを新たに作成します。パッケージ名は adamr2_driver とします。コード [4.7](#) を ROS ワークスペースディレクトリで実行し、パッケージを生成します。

^{*5} 必ずしも adamr2 のような名前空間の下に置く必要はありませんが、やはり名前空間を設定しておいた方が好ましいと考えたためこのようにしています。

^{*6} http://wiki.ros.org/dynamic_reconfigure

Code 4.7 Create adamr2_driver Package

```
catkin create pkg adamr2_driver \
--catkin-deps roscpp hardware_interface transmission_interface controller_manager ypspur
```

依存パッケージとして roscpp, hardware_interface, transmission_interface, controller_manager, ypspur を指定しています。

パッケージのディレクトリ構成はコード 4.8 のようになります。roscpp を依存パッケージに指定したので、生成されたパッケージのディレクトリにはソースコードを置くためのディレクトリが自動生成されています。このパッケージには launch/ や config/ ディレクトリ等は作りません。

Code 4.8 Directory Structure of adamr2_driver

```
adamlr2_control/
├ include/adamlr2_driver/
├ src/
└ package.xml
  CMakeLists.txt
```

4.4.2 C++ ノードの作成

ここから C++ ノードを作成していきます。ここでは以下の 3 つのソースファイルから 1 つのノードを作成することにします。

- adamr2_driver.h : クラス宣言のためのヘッダファイル
- adamr2_driver.cpp : クラス定義のためのソースファイル
- adamr2_driver_node.cpp : ノードの本体

adamlr2_driver.h は include/adamlr2_driver/ ディレクトリに、adamlr2_driver.cpp と adamlr2_driver_node.cpp は src/ ディレクトリに置きます。

adamlr2_driver.h の記述

まずはヘッダファイルから記述していきます。include/adamlr2_driver/ ディレクトリに adamlr2_driver.h という名前のファイルを作成し、コード 4.9 のように記述します。

Code 4.9 adamlr2_driver.h

```
#ifndef ADAMR2_ADAMR2_DRIVER_H_
#define ADAMR2_ADAMR2_DRIVER_H_

#include <ros/ros.h>
#include <hardware_interface/joint_command_interface.h>
#include <hardware_interface/joint_state_interface.h>
#include <hardware_interface/robot_hw.h>

namespace adamlr2 {
    class Adamr2Driver : public hardware_interface::RobotHW {
        public:
            Adamr2Driver();
            ~Adamr2Driver();

            ros::Time getTime() const {
                return ros::Time::now();
            }

            ros::Duration getPeriod() const {
                return ros::Duration(0.01);
            }

            int open() const;
            void stop() const;
            void read(ros::Time, ros::Duration);
            void write(ros::Time, ros::Duration);
    };
}
```

```

protected:
    hardware_interface::JointStateInterface joint_state_interface;
    hardware_interface::VelocityJointInterface joint_vel_interface;
    double cmd_[2];
    double pos_[2];
    double vel_[2];
    double eff_[2];
};

} // namespace adamr2

#endif

```

オリジナルのロボットで ros_control を使用する場合は、hardware_interface::RobotHW を継承したクラスを作成し、ロボットに合わせた設定や関数定義をする必要があります。[6]

ロボットのドライバと通信を行うのが read() メンバ関数と write() メンバ関数です。read() 関数でモータードライバからホイールエンコーダの情報を読み取り、write() 関数でモータードライバに車輪の速度を与えます。^{*7}

open() メンバ関数と stop() メンバ関数はそれぞれ YP-Spur の初期化関数と停止関数です。メンバ関数の中身の実装は adamr2_driver.cpp で行います。

メンバ変数の joint_state_interface 及び joint_vel_interface は、URDF で定義したホイールジョイントの情報を登録するための変数です。これらの変数の初期化はクラスコンストラクタ内で行います。

cmd_[] メンバ変数は、コントローラが計算した車輪への速度指令値が格納される配列です。対向 2 輪ロボットなので配列の要素数は 2 つです。pos_[], vel_[], eff_[] メンバ変数は、ジョイントの状態(位置、速度、トルク)を保持するための配列です。モータードライバから報告された位置、速度をこの変数に書き込み、それらの値をもとにコントローラがオドメトリ等を計算します。尚、トルクに関しては本ロボットでは扱わないので eff_[] 変数が更新されることはありません。

admr2_driver.cpp の記述

次にクラスの実装を行うソースファイルを記述します。src/ディレクトリに adamr2_driver.cpp という名前のファイルを作成し、コード 4.10 のように記述します。

Code 4.10 adamr2_driver.cpp

```

#include "admr2_driver/admr2_driver.h"
#include <ros/ros.h>

#include <ypspur.h>

namespace adamr2 {
    Adamr2Driver::Adamr2Driver() {
        // YP-Spur initialization.
        if (this->open() < 0) {
            ROS_WARN_STREAM("Error: Couldn't open spur.\n");
        }

        pos_[0] = 0.0;
        pos_[1] = 0.0;
        vel_[0] = 0.0;
        vel_[1] = 0.0;
        eff_[0] = 0.0;
        eff_[1] = 0.0;
        cmd_[0] = 0.0;
        cmd_[1] = 0.0;

        // Joint state setting for right-wheel-joint
        hardware_interface::JointStateHandle state_handle_1("right_wheel_joint", &pos_[0], &vel_[0], &eff_[0]);
        joint_state_interface.registerHandle(state_handle_1);
    }
}

```

^{*7} 文献によつては read() で指令送信、write() でデータ読み取りを行う、と書かれていますが、感覚的に見て write() で書き込み、read() で読み取りを行うのが道理でしょう。

```

// Joint state setting for left-wheel-joint
hardware_interface::JointStateHandle state_handle_2("left_wheel_joint", &pos_[1], &vel_[0], &eff_[0]);
joint_state_interface.registerHandle(state_handle_2);

registerInterface(&joint_state_interface);

// Joint handle setting for right-wheel-joint
hardware_interface::JointHandle vel_handle_1(joint_state_interface.getHandle("right_wheel_joint"), &cmd_[0]);
joint_vel_interface.registerHandle(vel_handle_1);
// Joint handle setting for left-wheel-joint
hardware_interface::JointHandle vel_handle_2(joint_state_interface.getHandle("left_wheel_joint"), &cmd_[1]);
joint_vel_interface.registerHandle(vel_handle_2);

registerInterface(&joint_vel_interface);
}

Adamr2Driver::~Adamr2Driver() {
    this->stop();
}

int Adamr2Driver::open() const {
    int ret = Spur_init();

    // Set the maximum angular velocity and acceleration.
    // unit is [rad/s] and [rad/s^2] in tire axis.
    YP_set_wheel_vel(11.6, 11.6);
    YP_set_wheel_accel(19.35, 19.35);

    return ret;
}

void Adamr2Driver::stop() const {
    YP_wheel_vel(0, 0);
    Spur_stop();
    ros::Duration(1).sleep();
    Spur_free();
}

void Adamr2Driver::read(ros::Time time, ros::Duration period) {
    // yp_vel[0] is right wheel velocity, yp_vel[1] is left wheel velocity.
    double yp_vel[2] = {0.0, 0.0};
    YP_get_wheel_vel(&yp_vel[0], &yp_vel[1]);

    // Reverse the velocity of the right wheel.
    // This is due to the coordinate system of the right wheel.
    yp_vel[0] = -yp_vel[0];

    for (unsigned int i = 0; i < 2; i++) {
        pos_[i] += yp_vel[i] * period.toSec();
        vel_[i] = yp_vel[i];
    }
}

void Adamr2Driver::write(ros::Time time, ros::Duration period) {
    YP_wheel_vel(-cmd_[0], cmd_[1]);
}

} // namespace adamr2

```

このソースファイルでは、`adamr2_driver.h`で宣言したクラスのメンバ関数の実装を行っています。以降、関数の詳細な解説を行っていきます。まずはクラスコンストラクタから解説します。コード 4.11 にクラスコンストラクタを抜粋します。

Code 4.11 Class Constructor in adamr2_driver.cpp

```
Adamr2Driver::Adamr2Driver() {
    // YP-Spur initialization.
    if (this->open() < 0) {
        ROS_WARN_STREAM("Error: Couldn't open spur.\n");
    }

    pos_[0] = 0.0;
    pos_[1] = 0.0;
    vel_[0] = 0.0;
    vel_[1] = 0.0;
    eff_[0] = 0.0;
    eff_[1] = 0.0;
    cmd_[0] = 0.0;
    cmd_[1] = 0.0;

    // Joint state setting for right-wheel-joint
    hardware_interface::JointStateHandle state_handle_1("right_wheel_joint", &pos_[0], &vel_
        [0], &eff_[0]);
    joint_state_interface.registerHandle(state_handle_1);
    // Joint state setting for left-wheel-joint
    hardware_interface::JointStateHandle state_handle_2("left_wheel_joint", &pos_[1], &vel_
        [0], &eff_[0]);
    joint_state_interface.registerHandle(state_handle_2);

    registerInterface(&joint_state_interface);

    // Joint handle setting for right-wheel-joint
    hardware_interface::JointHandle vel_handle_1(joint_state_interface.getHandle("right_wheel_jo
        int"), &cmd_[0]);
    joint_vel_interface.registerHandle(vel_handle_1);
    // Joint handle setting for left-wheel-joint
    hardware_interface::JointHandle vel_handle_2(joint_state_interface.getHandle("left_wheel_jo
        int"), &cmd_[1]);
    joint_vel_interface.registerHandle(vel_handle_2);

    registerInterface(&joint_vel_interface);
}
```

クラスコンストラクタでは、YP-Spur、メンバ変数、ジョイントインターフェースの初期化を行っています。重要となるのが後半のジョイントインターフェースの初期化です。URDFで定義したホイールジョイントの情報をhardware_interfaceに登録しています。hardware_interface::JointStateHandleクラスに与える第一引数は、URDFで定義したホイールジョイントの名前です。で定義した通り、right_wheel_jointとleft_wheel_jointという名前で登録しています。

次に、open()関数とstop()関数です。コード4.12に抜粋します。

Code 4.12 open() and stop() Member Function in adamr2_driver.cpp

```
int Adamr2Driver::open() const {
    int ret = Spur_init();

    // Set the maximum angular velocity and acceleration.
    // unit is [rad/s] and [rad/s^2] in tire axis.
    YP_set_wheel_vel(11.6, 11.6);
    YP_set_wheel_accel(19.35, 19.35);

    return ret;
}

void Adamr2Driver::stop() const {
    YP_wheel_vel(0, 0);
    Spur_stop();
    ros::Duration(1).sleep();
    Spur_free();
}
```

`open()` 関数では YP-Spur の初期化を行っています。`ypspur` ライブラリの `Spur_init()` 関数を使って初期化を行っています。また、`YP_set_wheel_vel` 関数及び `YP_set_wheel_accel` 関数を使って、各車輪の最大速度と最大加速度を設定しています。第一引数が右車輪、第二引数が左車輪に対応する値です。どちらも同じ数値を設定します。ここで設定している値は、直径 155 mm のホイールを付けた際にロボットの直進移動速度が 0.9 m/s を超えないように、また直進加速度が 1.5 m/s² を超えないような値に設定しています。

次に `read()` 関数と `write()` 関数です。コード 4.13 に抜粋します。

Code 4.13 `read()` and `write()` Member Function in `adamr2_driver.cpp`

```
void Adamr2Driver::read(ros::Time time, ros::Duration period) {
    // yp_vel[0] is right wheel velocity, yp_vel[1] is left wheel velocity.
    double yp_vel[2] = {0.0, 0.0};
    YP_get_wheel_vel(&yp_vel[0], &yp_vel[1]);

    // Reverse the velocity of the right wheel.
    // This is due to the coordinate system of the right wheel.
    yp_vel[0] = -yp_vel[0];

    for (unsigned int i = 0; i < 2; i++) {
        pos_[i] += yp_vel[i] * period.toSec();
        vel_[i] = yp_vel[i];
    }
}

void Adamr2Driver::write(ros::Time time, ros::Duration period) {
    YP_wheel_vel(-cmd_[0], cmd_[1]);
}
```

`read()` 関数では `YP_get_wheel_vel()` 関数を使って、モータードライバから各車輪の速度を求めています。受け取った結果を一時変数に保持し、座標系を解決してから `pos_[]` 変数と `vel_[]` 変数に格納しています。

でも述べたように、本ロボットのホイールの座標系は、「ロボットが直進する方向を正とする」と定めています。しかし YP-Spur が返す車輪速度値はそのようにはなっていません。具体的にいうと、右車輪の速度の正負が逆さまになって返ってきます。そのため、右車輪の速度の値の正負を反転させてメンバ変数に記録しています。

YP-Spur から得られる情報は車輪速度だけですが、時間がわかっているので位置も求めることができます（あくまで推定値ですが）。`read()` 関数の最後の `for` 文で、ジョイントの位置と速度をメンバ変数に記録しています。

`write()` 関数では車輪の速度指令値を YP-Spur に与えているだけです。ここでも右車輪の値の正負は反転させなければなりません。

adamr2_driver_node.cpp の記述

最後に ROS ノードの本体となるソースファイルを記述します。`src/` ディレクトリに `adamr2_driver_node.cpp` という名前のファイルを作成し、コード 4.14 のように記述します。

Code 4.14 `adamr2_driver_node.cpp`

```
#include "adamr2_driver/adamr2_driver.h"
#include <ros/ros.h>
#include <controller_manager/controller_manager.h>

//extern "C" {
#include <ypspur.h>
//}

int main(int argc, char **argv[]) {
    ros::init(argc, argv, "adamr2_driver_node");
    ros::NodeHandle nh;

    adamr2::Adamr2Driver driver;
    controller_manager::ControllerManager cm(&driver);
```

```

ros::AsyncSpinner spinner(1);
spinner.start();

while(ros::ok()) {
    ros::Time now = driver.getTime();
    ros::Duration dt = driver.getPeriod();

    if (YP_get_error_state() == 0) {
        driver.write(now, dt);
        cm.update(now, dt);

        driver.read(now, dt);
    } else {
        ROS_WARN("T-Frog driver disconnected.");
        driver.stop();

        while (driver.open() < 0) {
            ROS_WARN("Try to connect T-Frog driver...");
            ros::Duration(1).sleep();
        }
    }

    ROS_INFO("T-Frog driver connected.");
}

dt.sleep();
}

spinner.stop();

return 0;
}

```

adamr2_driver.h で宣言したクラスのオブジェクト (driver) を作り、ros_control の枠組みに従って while 文で制御ループを回しています。

while ループの中で、driver.write() 関数を呼び出して速度指令値をモータードライバに送り、cm.update() 関数を呼び出して状態を更新、driver.read() 関数を呼び出して車輪の現在速度を取得しています。

4.4.3 CMakeLists.txt の編集

ノードをビルドするために、CMakeLists.txt を編集します。コード

Code 4.15 CMakeLists.txt

```

cmake_minimum_required(VERSION 2.8.3)
project(adamr2_driver)

find_package(catkin REQUIRED COMPONENTS
    roscpp
    hardware_interface
    transmission_interface
    controller_manager
)
catkin_package()

find_library(ypspur_LIBRARIES ypspur)

include_directories(
    include
    ${catkin_INCLUDE_DIRS}
    ${ypspur_INCLUDE_DIRS}
)
add_executable(adamr2_driver_node

```

```
src/adamr2_driver_node.cpp  
src/adamr2_driver.cpp  
)  
  
target_link_libraries(adamr2_driver_node  
${catkin_LIBRARIES}  
${ypspur_LIBRARIES}  
)
```

4.4.4 パッケージのビルド

ROS ノードの実行可能ファイルを生成するために、ビルドを実行します。コードを実行して、ワークスペースをビルドしましょう。

Code 4.16 Build Workspace

```
catkin build
```

ビルドが通ったら、無事にドライバノードを作成できることになります。

参考文献

- [1] MathWorks. URDF 入門. <https://jp.mathworks.com/help/physmod/sm/ug/urdf-model-import.html>.
- [2] Open Source Robotics Foundation. urdf/XML/model - ROS wiki. <http://wiki.ros.org/urdf/XML/model>.
- [3] Open Source Robotics Foundation. urdf/XML/robot - ROS wiki. <http://wiki.ros.org/urdf/XML/robot>.
- [4] Open Source Robotics Foundation. urdf/XML/link - ROS wiki. <http://wiki.ros.org/urdf/XML/link>.
- [5] Open Source Robotics Foundation. urdf/XML/joint - ROS wiki. <http://wiki.ros.org/urdf/XML/joint>.
- [6] 西田健, 森田賢, 岡田浩之, 原祥堯, 山崎公俊, 田向権, 垣内洋平, 大川一也, 斎藤功, 田中良道, 有田裕太, 石田裕太郎. 実用ロボット開発のための ROS プログラミング. 森北出版, 2018.