



# Robotics Engineering II Report

Sanjeevan Kennedy , Carlos Saucedo

Professor Ralph Seulin

May 15,2017

# Overview

In this report we explore the multiple dynamics of ROS by setting the task to pick up an object with the PhantomX Pincher. Our plan is simple, we travel through Arduino and MoveIt! , promising programs that will allow us to control the arm through ROS. Synonymously we will manipulate the Turtlebot to move through our created map from Lidar by setting manual goals, and we troubleshoot and find paths to complete our objective. Ultimately we will notate in this report where we finished.

## Dynamixel Programming

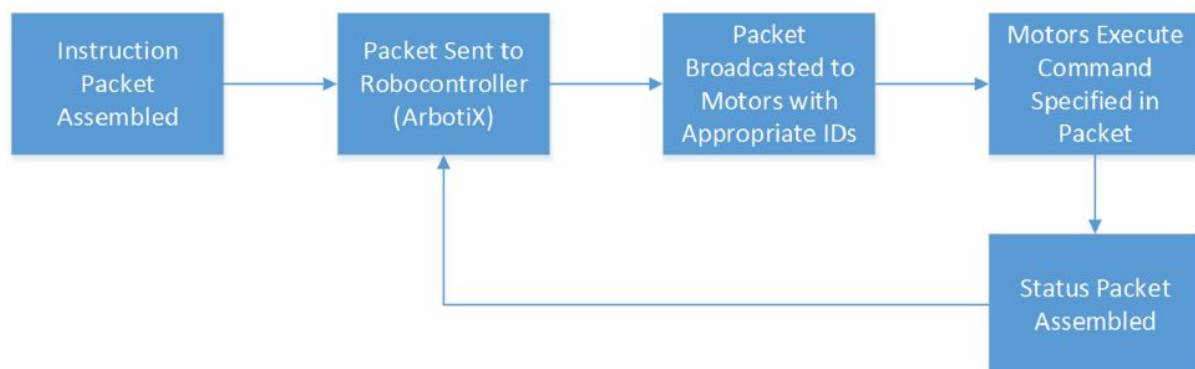
### Binary Data and Decimal Conversion

Dynamixel servos are commanded by receiving packets of binary instructions. The user sends instructional packets, and the motors will then send status packets.

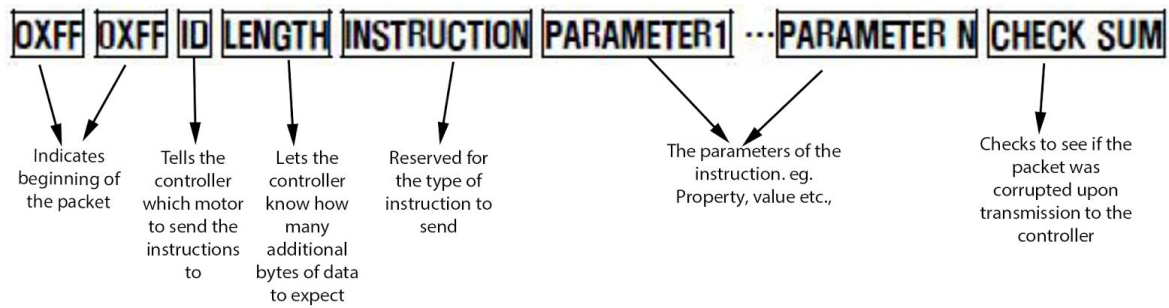
### Hexadecimal Representation

- The advantages of hexadecimal are that it is much simpler to convert binary to hexadecimal, and one byte of data can always be represented by two hexadecimal characters.

### Dynamixel Communication Overview



- An instruction packet is command data that is sent to the main controller, which then sends it to Dynamixel motors. the 0XFF 0XFF does is notifies the controller that it is the beginning of the packet.



- The format of the status packet is nearly identical to that of the instruction packet; the only difference is in place of the “Instruction” byte, the status packet sends back an “Error” byte. The error byte is capable of identifying any of these potential causes of error using only one byte of data. For eg. 0010 0000 signifies an overheating error as all bits are set to 0, except bit 2 (starting from bit 0)

Byte Value	Name	Function	No.of Parameters
0x01	PING	No execution. It is used when controller is ready to receive Status Packet	0
0x02	READ_DATA	This command reads data from Dynamixel	2
0x03	WRITE_DATA	This command writes data to Dynamixel	2 or more
0x04	REG WRITE	It is similar to WRITE_DATA, but it remains in the standby state without being executed until the ACTION command arrives.	2 or more
0x05	ACTION	This command initiates motions registered with REG WRITE	0
0x06	RESET	This command restores the state of Dynamixel to the factory default setting.	0
0x83	SYNC WRITE	This command is used to control several Dynamixels simultaneously at a time.	4 or more

The above is instruction table with the Byte values for and the corresponding functions.

### Dynamixel Packet Examples

Data is sent to the serial buffer of the robocontroller which will ultimately be sent directly to the motors

#### Example 1: Write Data

How to change the ID of an AX-12A motor from 1 to 5?

##### Step 1: Beginning of packet

- Each packet will begin with the two bytes 0xFF 0xFF to signify that it is indeed the beginning

of the packet.

Packet so far: 0XFF 0XFF

#### Step 2: ID

- To send data to the motor with ID #1, the value of this byte should just be 1 (0X01)

Packet so far: 0XFF 0XFF 0X01

#### Step 3: Length

- The next byte of data will tell the motor with ID #1 how many more bytes of data to expect with 2 parameters needed (one to indicate the motor about the ID change and the other carrying the value to which it's changed.) Thus the length byte will be 4 (0X04)

Packet so far: 0XFF 0XFF 0X01 0X04

#### Step 4: Instruction

- Possible instructions are writing data, reading data, simply pinging the system to inform the controller of system status, etc...Eg. WRITE\_DATA is signified by the value 3 (0X03)

Packet so far: 0XFF 0XFF 0X01 0X04 0X03

#### Step 5: Parameters

- Two parameters are needed. As mentioned in step 3 the parameters would map to the address of the ID value in the EEPROM. The instruction table is referred to find the value for the AX-12A motor. If the goal is to change the ID of the motor to 5, the second parameter value will be 0X05 and the first parameter byte will be 0X03 as 3 is the byte value for write.

Packet so far: 0XFF 0XFF 0X01 0X04 0X03 0X03 0X05

#### Step 6: Checksum

- The last byte is the checksum value. The checksum is equal to the opposite value of the low byte of the sum of all of the previous bytes.

#### Step 6a: Add up all data bytes (not the 0XFF 0XFF):

$$0X01 + 0X04 + 0X03 + 0X03 + 0X05 = 1 + 4 + 3 + 3 + 5 = 16$$

#### Step 6b: Convert to binary

$$16 = 2^4 = 0001\ 0000.$$

#### Step 6c: Take the opposite value

- The binary sum value is taken and the not bit operator is applied to it.

$$\sim(0001\ 0000) = 1110\ 1111.$$

#### Step 6d: Convert to hexadecimal and compile the final byte

- 1110 in decimal is 14, in hexadecimal that maps to E. 1111 in decimal is 15, in hexadecimal that maps to F. So 1110 1111 = 0XEF.

Final Instruction Packet: 0XFF 0XFF 0X01 0X04 0X03 0X03 0X05 0XEF

- The checksum byte can be recalculated by the controller and compared the value that is sent. If they are mismatched, that means some data was lost in communication and the data may be corrupted

### **Example 2: Read Data**

The following example is to read the present voltage that the motor with ID 1 is receiving.

#### **Step 1: Beginning of packet**

- It is still just 0XFF 0XFF; all packets will start this way when working with Dynamixel motors.

Packet so far: 0XFF 0XFF

#### **Step 2: ID**

- The ID of the motor in which the data is read is just 1, so our ID byte is 0X01

Packet so far: 0XFF 0XFF 0X01

#### **Step 3: Length**

- Two parameters exist, plus the instruction and the checksum. The length will be 0X04. The two parameters are the address of the voltage, and the length of the data that is being read (1 byte).

Packet so far: 0XFF 0XFF 0X01 0X04

#### **Step 4: Instruction**

- The byte value for reading data is found. It is designated by 0X02, so the byte for this instruction is 0X02.

Packet so far: 0XFF 0XFF 0X01 0X04 0X02

#### **Step 5: Parameters**

- The first parameter is the address of data to be read. If the goal is to read the voltage (value = 1) being supplied to the motor whose address is 42, the first parameter will be 42 (0X2A) and the second parameter value will be the number of bytes to read (which is just 1). If it is needed to read from multiple addresses, an additional byte for each address is required.

Packet so far: 0XFF 0XFF 0X01 0X04 0X02 0X2A 0X01

#### **Step 6: Checksum**

- The checksum value is to be calculated and attached to the end of our packet so that the controller can verify that no data was lost upon sending the packet. We will take the same steps that we did in example 1.

#### **Step 6a: Add up all the data bytes (excluding the 0XFF 0XFF beginning of packet)**

$$0X01 + 0X04 + 0X02 + 0X2A + 0X01 = 1 + 4 + 2 + 42 + 1 = 50$$

#### **Step 6b: Convert to Binary**

$$50 = 0011\ 0010$$

#### **Step 6c: Take the opposite value**

$$\sim(0011\ 0010) = 1100\ 1101$$

#### **Step 6d: Convert to hexadecimal and compile the final byte**

1100 = 12 in decimal which maps to C in hexadecimal

1101 = 13 in decimal which maps to D in hexadecimal

Checksum byte = 0XCD

Final Packet: 0XFF 0XFF 0X01 0X04 0X02 0X2A 0X01 0XCD

### **Programming Dynamixel Motors Using the Arduino IDE and ax12 Library**

- Once the controller is blinking properly, the Dynamixel motor can be connected using the 3 pin Dynamixel compatible cable. All ports are electronically equivalent; they can be used for organizing separate chains of motors but they will all behave the same. Then, A new sketch is created and the “Bioloid” library is imported. Each “Sketch” for an Arduino controller will have at least two required functions: the setup() function and the loop() function. Any code that is placed within the setup() function will execute immediately upon uploading the sketch to the board whereas within the loop() function will execute immediately upon completion of the setup() code; this code will continue to run over and over again until power is removed from the board, or another sketch is uploaded.



```
Dyna_PositionSet
#include <ax12.h>
#include <BioloidController.h>

void setup() {}
void loop() {}
```

#### Changing the Position of a Dynamixel Motor:

- The sketch is uploaded and the motion of a Dynamixel motor is ready to be controlled

##### Step 1: Center the Motor

```
SetPosition(1,512);
```

```
delay(3000);
```

- The range of values for the position of an AX- 12A motor are 0-1023, so setting it to 512 should center it. The delay(x) function will impose a pause on the execution of the code for x number of milliseconds, and it is essential for many Dynamixel operations.

##### Step 2: Move the Motor Back and Forth

```
SetPosition(1, 0);
```

```
delay(3000);
```

```
SetPosition(1, 1023);
```

```
delay(3000);
```

- These four lines of code will have the motor move to its two extreme positions with a three second delay in-between each motion

### Step 3: Uploading the Sketch

- A sketch can be uploaded by clicking the (Upload) button in the top left of the IDE. The (Verify) button compiles to check for errors. A status bar in the bottom right of the screen that will show when it is compiling, when it is uploading, and when it is done. The compiling errors, will notify the user, and the code will not upload until the errors are fixed.

### **Using ax12SetRegister and ax12SetRegister2:**

- These are two more versatile ax12 functions. The parameters are: the ID of the motor, the starting register (or address) of the property to modify, and the value to place in that register(s). The ax12SetRegister function will modify one value mapped to an address in the motor whereas the ax12SetRegister2 function will modify two values with one function call.

ax12SetRegister example:

```

#include <ax12.h>
#include <BioloidController.h>

void setup()
{
    //Initializes the serial monitor
    Serial.begin(9600);
    //Gives you three seconds to open the serial monitor
    delay(3000);
    Serial.println("Hello World!");
    delay(10);
    //Initialized to -1 by convention
    int newID = -1;
    //Change ID of motor 1 to 2
    ax12SetRegister(1, 3, 2);
    //Checks the value in address 3 (ID) of the Dynamixel
    newID = ax12GetRegister(2, 3, 1);
    //Prints the data to the Serial Monitor
    Serial.print("The new ID of the Dynamixel motor is: ");
    Serial.println(newID);
}

void loop()
{
}

```

- This register is used when the ID values are represented with only one byte of data. The three parameters are : the id (which is 1) the regstart (which is 3) and the replacement ID (which will be 2)
- Uploading the first sketch created and seeing if the position moves is the most obvious way to verify if the ID has actually changed or not. Also, the ax12GetRegister(id,regstart, length) function can be used to verify. The three parameters are similar to that of ax12SetRegister, with the exception of the length parameter. The length represents the number of bytes of data that are being read. For reading properties that take up two bytes in the register, the length may be two.
- A variable called newID is set to -1 which just a convention that can make troubleshooting easier later on. Since all values relating to a Dynamixel motor are unsigned (they are positive), the variable can be initialized to -1 and it will never interfere with the values being set to later. The newID value can be set to



the value returned by the `ax12GetRegister` function. The line of code which does this is **`newID=ax12GetRegister(2,3,1);`**

- **`Serial.begin(9600);`** initializes the serial object. (where 9600 is the baud rate). This will usually be one of the first commands in the code, so it is placed before our **`int newID = -1`** line. One other useful convention is to put a delay in-between any `Serial.write` lines, as the data can only transfer so quickly. “Hello World” is written to the Serial monitor in the **`Serial.println(“Hello world!”);`** and then the `newID` value is written. The user has 3 seconds to open the serial monitor upon uploading to see the “Hello world!” message as 3000 is delay given.

Writing the data to the serial monitor is done by the code

**`Serial.print(“The new ID of the Dynamixel motor is: “);`**  
**`Serial.println(newID);`**

`ax12SetRegister2` example:

```
Dyna_PositionSet2
#include <ax12.h>
#include <BioloidController.h>

void setup()
{
    //Sets the position of motor with ID#1 to 512
    ax12SetRegister2(1, 30, 512);
    delay(3000);
}

void loop()
{
    ax12SetRegister2(1, 30, 0);
    delay(3000);
    ax12SetRegister2(1, 30, 1023);
    delay(3000);
}
```

- The `ax12SetRegister2` function will modify two bytes of data in two adjacent addresses in the memory of the motor, and it will also convert integers greater than one byte (255) to the lower bytes of data. For example the value of 1023 in binary is 0000 0011 1111 1111; the function will do that conversion, and send the high byte (0000 0011) to

the address requiring the high (H) byte, and the low byte (1111 1111) to the address requiring the low (L) byte.

- Most properties require both bytes, so this function proves useful. After setting up the sketch, the motor is centered using the code **ax12SetRegister2(1,30,512);** where the address represented by 30 is Goal Position(L). When using this function, the low byte address will always be referred to (which will be the lower number), not the high byte address.
- The function will write to both addresses (in this case 30 and 31), but putting 31 as the parameter will cause an error. This function is used to set the position. The middle parameter (the address) 30 is repeated for every function call, and it can be included in the library so it is not necessary to keep entering it.
- There exists a list of all defined variables in the ax12 library. For instance, the defined value for address 30 is `AX_GOAL_POSITION_L`. In other words, if `AX_GOAL_POSITION_L` is written in the code, the `#define` value in the library would have the compiler replace every instance of this line with the value 30.

### **Controlling Several Motors Simultaneously:**

```

straight_position
int pos2 = adjustPosition(3,2);
}

void loop()

//function is for rest position

{

SetPosition (1, 500);//direct
SetPosition (2, 200);
SetPosition (3, 500);
SetPosition (4, 670);
SetPosition (5, 400);
delay(4000);
}

//Set position of motor b based on position of a motor a
//return the new value of the position of motor b

int adjustPosition(int a, int b)

{
    //gets two bytes of data starting at address 36 of motor A
    int aPos = ax12GetRegister(a, 36, 2);

    if(aPos < 512)

    {
        SetPosition(b, aPos);
        return aPos;
    }

    else

    {
        SetPosition(b, 1023);
        return 1023;
    }
}

```

- The robot has a rest position when motor 1 has position 500, motor 2 has position 200, and so on. The delay at the end of the function gives the motor three seconds to return to this position. This is also the appropriate way to control more than one Dynamixel motor from one sketch. The only thing the user had to do was change the ID parameter of the **SetPosition** function. Now, anytime when the function “**setRestPosition();**” is called, these six lines will execute.
- The **adjustPosition()** function will measure the position of motor A, and adjust the position of motor B depending on the position of motor A and the value of the new position of motor B must be returned back to the code that

called it. The “int” before the name of the function tells that the function will send an integer back to wherever it is called.

- The “int a, int b” in-between the parenthesis means that there are two integer parameters that the function expects while being called. It is also possible to overload the function, meaning multiple function definitions can exist with the same name but different parameter signatures.
- **int aPos = ax12GetRegister(a,36,2);** is itself a function which would get both bytes of the present position of motor with id “a” and store it in “aPos”. If the position of motor a is less than 512, it will set the position of motor b to whatever it currently is. It then returns the value that b will get to. If it is greater than or equal to 512, it sets b to 1023 and returns 1023. Only one of these logic paths can be utilized in a function call, so only one integer will ever be returned. Thus, new value of the position of motor 2 is stored in the integer variable “pos2”.

# Using the ROS MoveIt! and Navigation Stack

MoveIt! Is set of packages for motion planning with RViz, manipulation, 3D perception, kinematics, collision checking, control, and navigation.

## Package, Installation and Architecture:

- The following command to install moveit!:

```
sudo apt-get install ros-indigo-moveit
```

- Architecture:

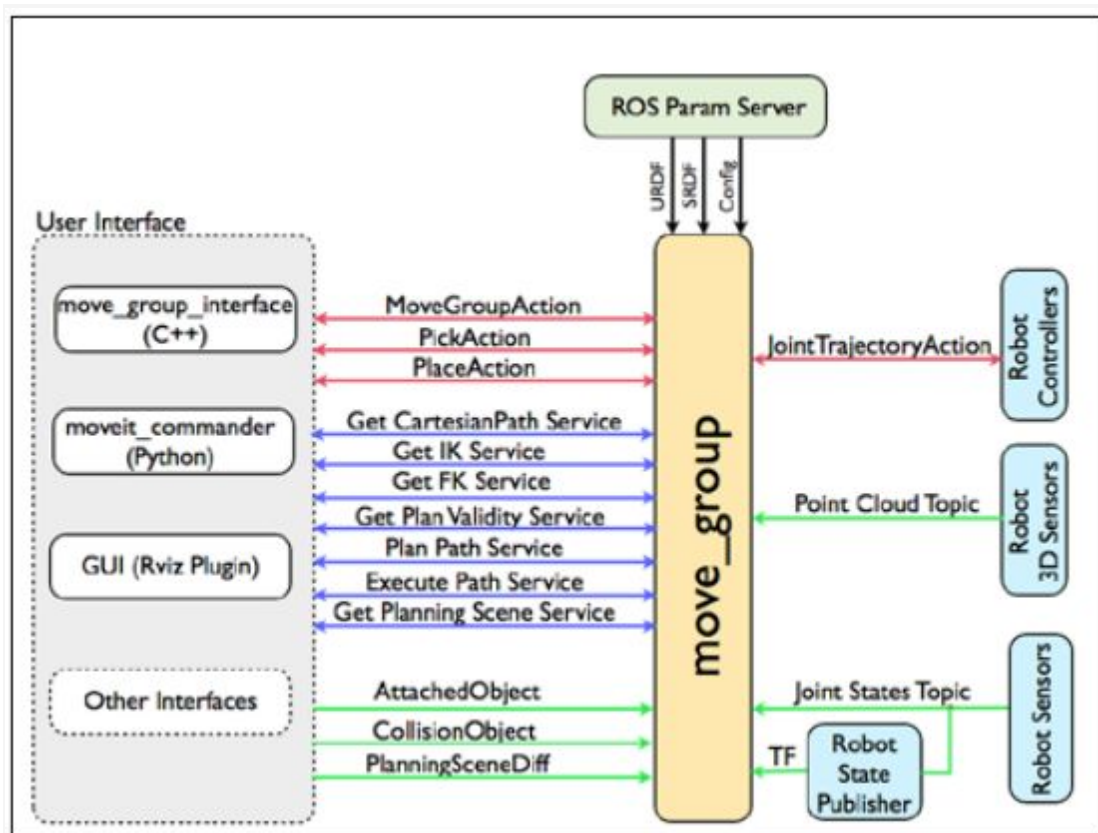


Fig 1. MoveIt! Architecture diagram

The **move\_group** node is an integrator. It does not run any kind of motion planning algorithms but instead connects all the functionalities (point cloud, joint state of the robot, and transform (T.F) of the robot in the form of topics and services) as plugins.

From the parameter server, it collects the robot kinematics data, such as robot\_description (URDF), SRDF (Semantic Robot Description Format), and the configuration files.

C++ or Python MoveIt! APIs and RViz motion planning plugin can be used to command the **move\_group** node to perform actions such as pick/place, IK, FK, etc.

### **Motion planning:**

Motion planning is the technique to find an optimum path with the knowledge of the starting pose of the robot, a desired goal pose of the robot, the geometrical description of the robot (URDF file), and geometrical description of the world, that moves the robot gradually from the start pose to the goal pose, while never touching any obstacles in the world and without colliding with the robot links

- MoveIt! can talk to the motion planners through the plugin interface. Any motion planner can be used by simply changing the plugin. The default planner for the **move\_group** node is OMPL.
- To start motion planning, we should send a motion planning request to the motion planner which specified our planning requirements. The planning requirement may be setting a new goal pose of the end-effector, for example, for a pick and place operation.
- We can set additional kinematic constraints for the motion planners. Given next are some inbuilt constraints in MoveIt!:
  - ❖ Position constraints: These restrict the position of a link
  - ❖ Orientation constraints: These restrict the orientation of a link
  - ❖ Visibility constraints: These restrict a point on the link to be visible in a particular area (view of a sensor)
  - ❖ Joint constraints: These restrict a joint within its joint limits
  - ❖ User-specified constraints: Using these constraints, the user can define his own constraints using the callback functions

### **Motion planning request adapters:**

- The planning request adapters help to pre-process the motion planning request and post process the motion planning response. One of the uses of pre-processing requests is that it helps to correct if there is a violation in the joints states and, for the post processing, it can convert the path generated by the planner to a time parameterized

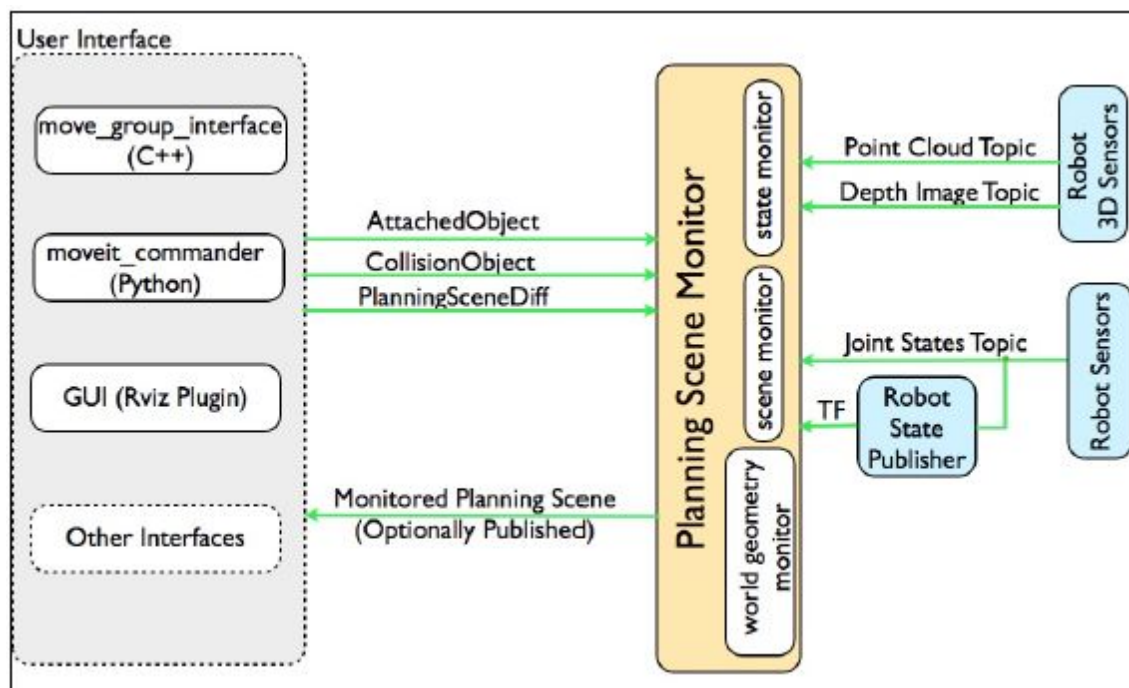
trajectory. Following are some of the default planning request adapters in MoveIt!:

- **FixStartStateBounds**: If a joint state is slightly outside the joint limits, then this adapter can fix the initial joint limits within the limits.
- **FixWorkspaceBounds**: This specifies a workspace for planning with a cube size of 10m x 10m x 10m.

- **FixStartStateCollision:** This adapter samples a new collision free configuration if the existing joint configuration is in collision. It makes a new configuration by changing the current configuration by a small factor called `jiggle_factor`.
- **FixStartStatePathConstraints:** This adapter is used when the initial pose of the robot does not obey the path constraints. In this, it finds a near pose which satisfies the path constraints and uses that pose as the initial state.
- **AddTimeParameterization:** This adapter parameterizes the motion plan by applying the velocity and acceleration constraints.

### MoveIt! planning scene:

The term planning scene is used to represent the world around the robot and also stores the state of the robot itself



### MoveIt! Kinematics handling:

- MoveIt! provides a great flexibility to switch the inverse kinematics algorithms using the robot plugins. Users can write their own IK solver as a MoveIt! plugin and switch from the default solver plugin whenever required.
- The package called IKFast can be used to generate a C++ code for solving IK using analytical methods,
- Forward kinematics and finding jacobians are already integrated to the MoveIt! `RobotState` class.

### MoveIt! collision checking:

- The `CollisionWorld` object inside MoveIt! is used to find collisions (for different types of objects, such as meshes, primitive shapes such as boxes,

cylinders etc.) inside a planning scene which is using the FCL (Flexible Collision Library) package as a backend

- To reduce this computationally expensive tasks, MoveIt! provides a matrix called ACM (Allowed Collision Matrix) which contains a binary value (as 1 where the bodies are always so far that they would never collide with each other) corresponding to the need to check for collision between two pairs of bodies.

## **Generating MoveIt! Configuration package using Setup Assistant tool**

### **Step 1 – Launching the Setup Assistant tool:**

```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

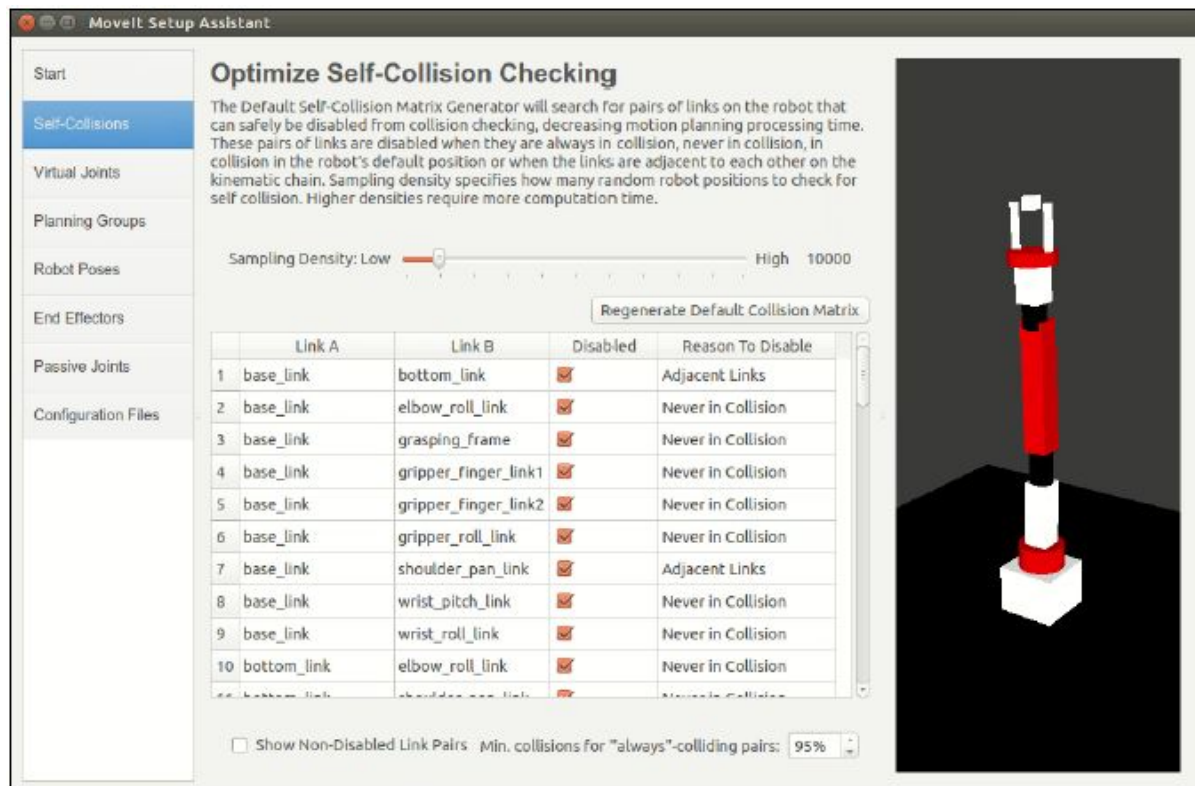


- A new MoveIt! Config package is created or an existing one is loaded

### **Step 2 – Generating the Self-Collision matrix:**

- In this step, MoveIt! searches for a pair of links on the robot which can be safely disabled (after analysis) from the collision checking.





- The sampling density is the number of random positions to check for selfcollision.

### Step 3 – Adding virtual joints:

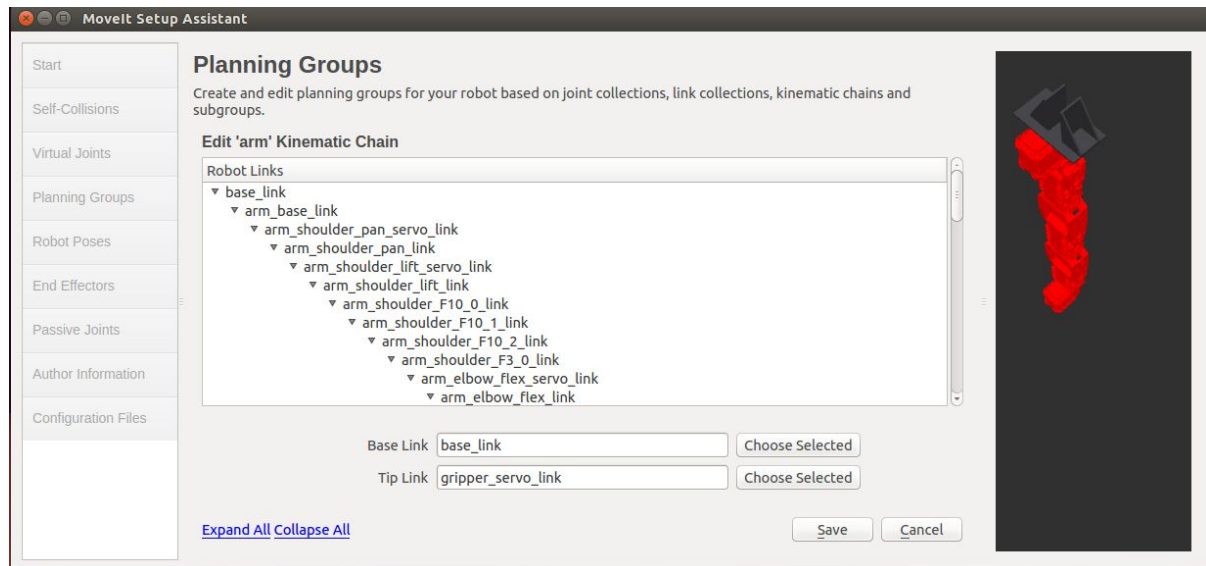
- virtual joints are defined with respect to the odom frame

### Step4 - Planning groups:

- A group of joints/links in a robotic arm which plans together in order to achieve a goal position of a link or the end effector.

two planning groups, one for the arm and one for the gripper.

The groups, joints and links are added.



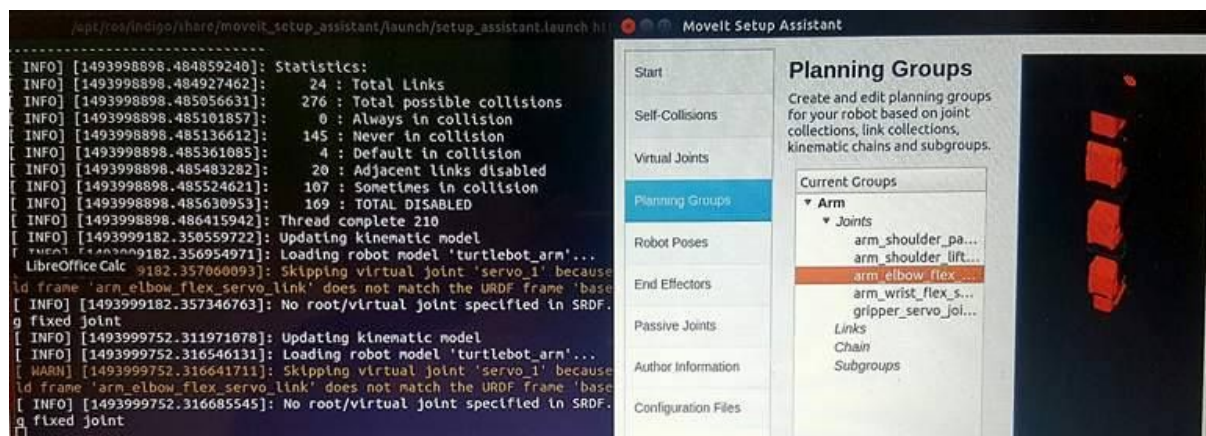
(Actual PhantomX Pincher in MoveIt!)

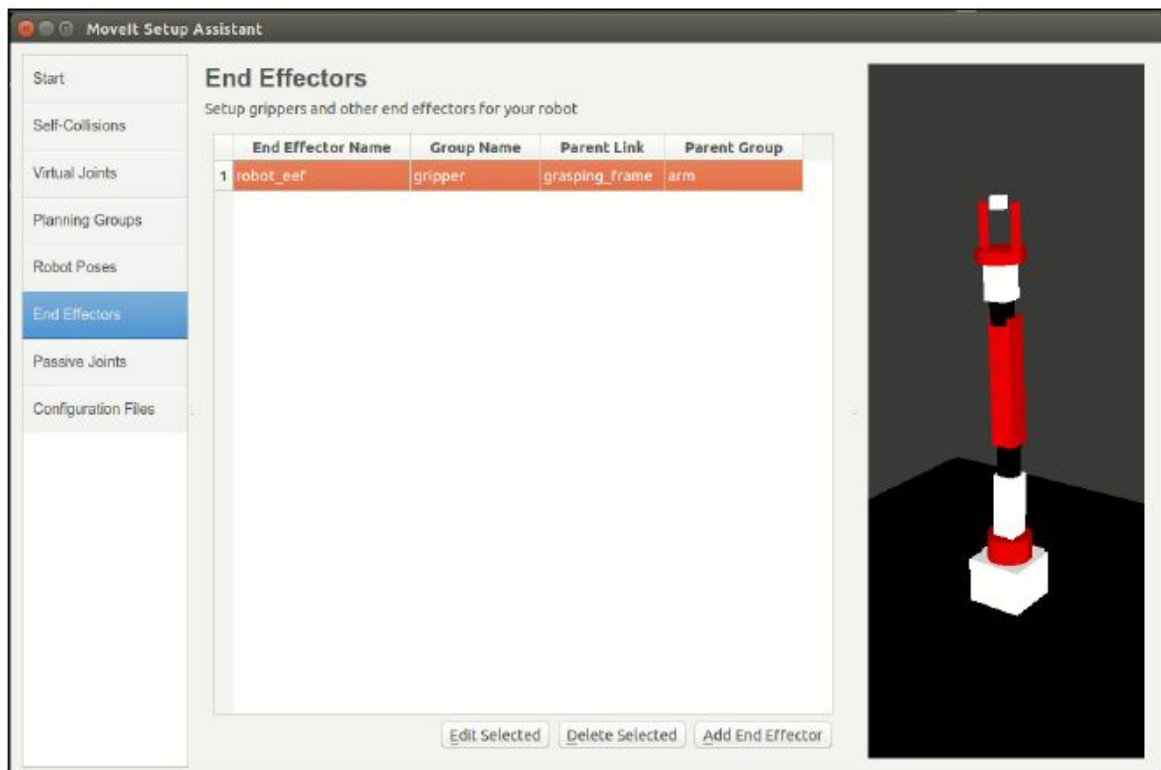
### Step 5 – Adding the robot poses:

- Certain fixed poses in the robot configuration are added and they are called while programming with the API's

### Step 6 – Setup the robot end effector:

- The robot end effector, the end effector group, the parent link, and the parent group are added and assigned

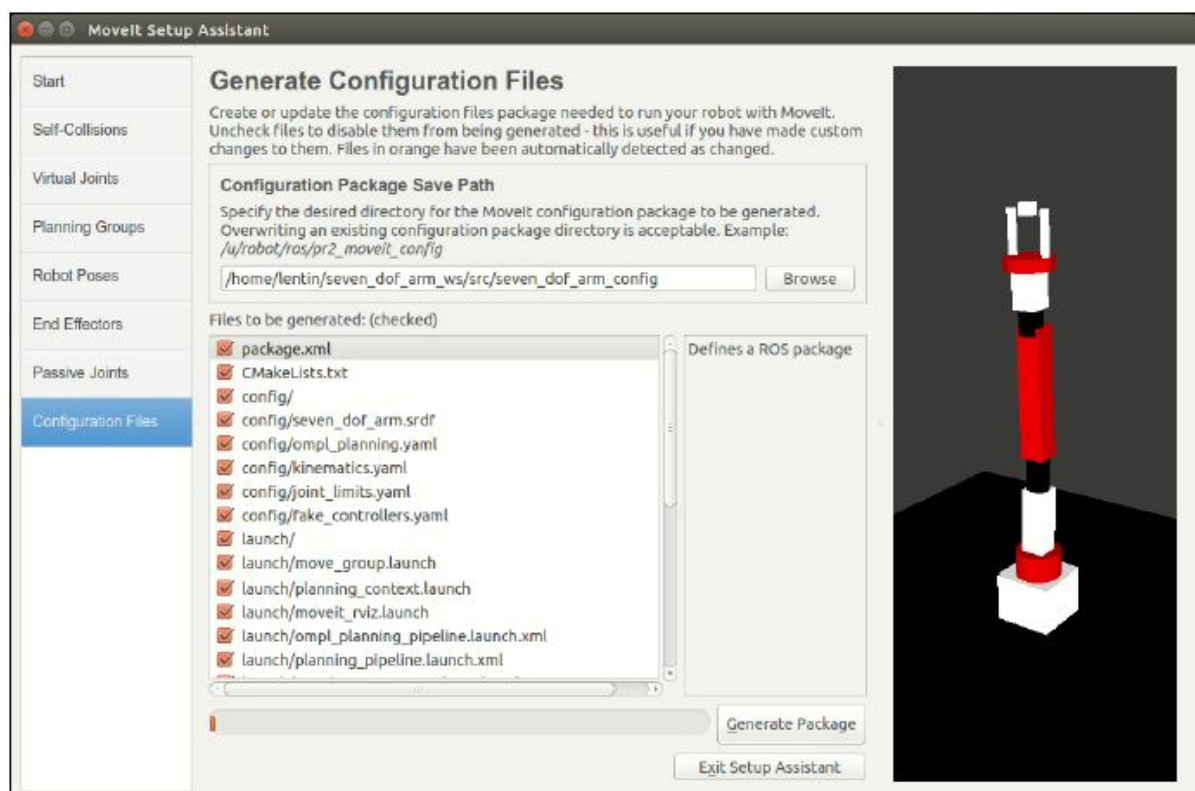




## Step 7 – Adding passive joints

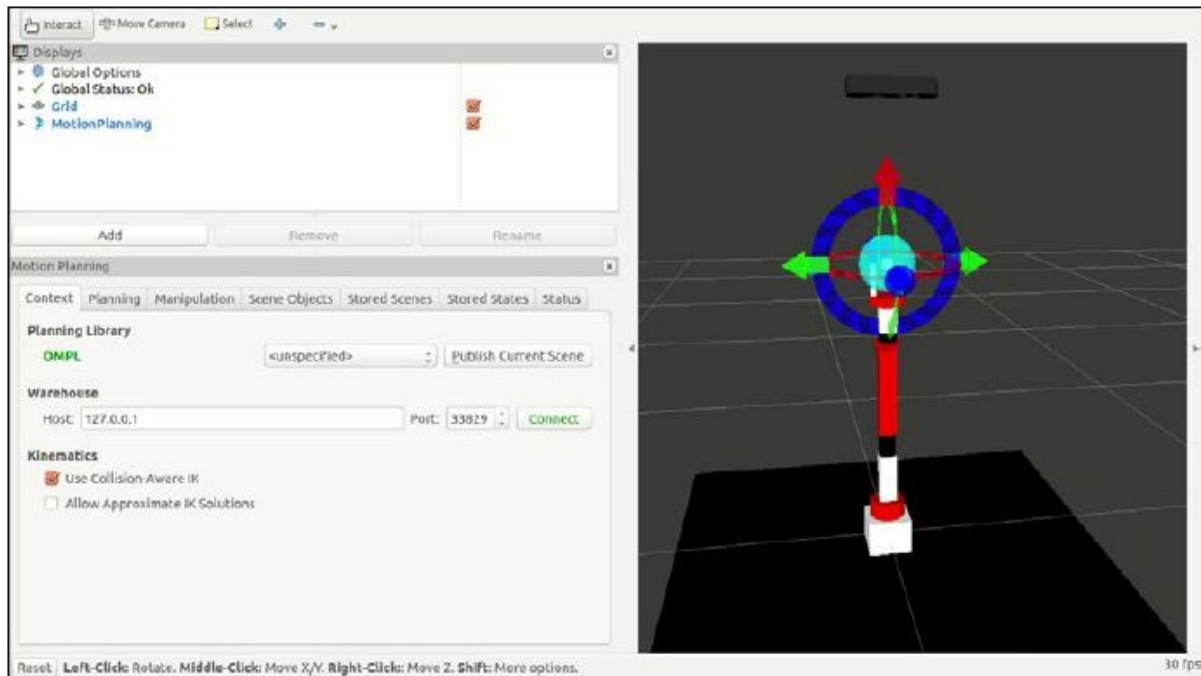
- The passive joints (eg, Caster wheels) in the robot are specified. The planner will ignore these kind of joints during motion planning.

## Step 8 – Generating configuration files



- The tool will generate a configuration package which contains the file needed to interface MoveIt!.

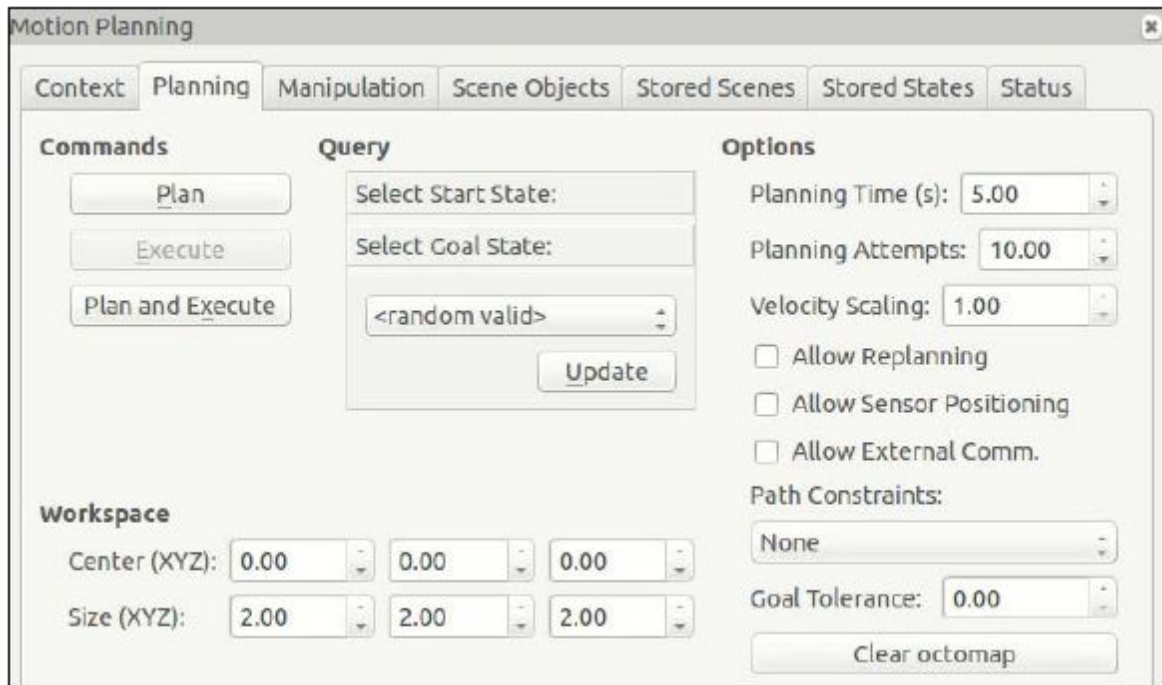
## Motion planning of robot in RViz using MoveIt! configuration package



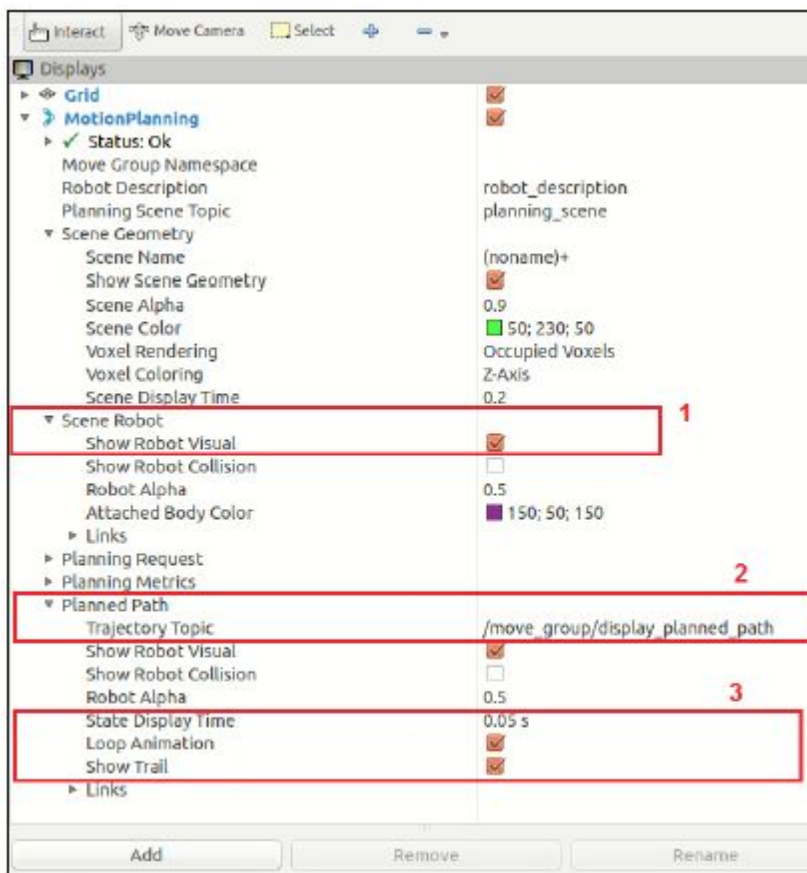
- `$ roslaunch seven_dof_arm_config demo.launch` command is run to launch the file to obtain the RViz. OMPL displayed in green proves the successful load of the library.

### Using the RViz MotionPlanning plugin

- The start and goal poses can be planned in the query panel of the planning tab.



- The interactive marker attached to the arm gripper helps in the assignment. Orange color depicts an existence of a planning solution and red depicts a self collided state. Random assignment can also be done.



The 'Show Robot Visual' option is checked to show the robot model.





- Similarly, the other options when checked, help with visualization.

## Interfacing the MoveIt! configuration package to Gazebo

### Step 1 – Writing the controller configuration file for MoveIt!

- The configuration **.yaml** file is created inside the config folder of the `seven_dof_arm_config` package.

### Step 2 – Creating the controller launch files

- The controller configuration file contains the definition of the two controller interfaces; viz., the arm and the gripper. The type of action used in the controllers is **FollowJointTrajectory**, and the action namespace is **follow\_joint\_trajectory**. The joints under each group must be listed.
- The **default: true** indicates that it will use the default controller, which is the primary controller in MoveIt! for communicating with the set of joints.

### Step 3 – Creating the controller configuration file for Gazebo

- A new file called **trajectory\_control.yaml** which contains the list of the Gazebo ROS controllers that need to be loaded along with Gazebo is created

### Step 4 – Creating the launch file for Gazebo trajectory controllers

- A launch file which launches Gazebo, the trajectory controllers, and the MoveIt! interface in a single command is created. This launch file spawns the robot model in Gazebo, publishes the joint states, attaches the position controller, attaches the trajectory controller, and at last launches **moveit\_planning\_execution.launch** inside the MoveIt! package for starting the MoveIt! nodes along with RViz.

**\$ roslaunch seven\_dof\_arm\_config**

**seven\_dof\_arm\_bringup\_moveit.launch**

- The above command is run to start motion planning inside RViz and execute in Gazebo. After planning, the execute button is clicked to send the trajectory to the gazebo controllers.

### Step 5 – Debugging the Gazebo- MoveIt! Interface

- If the trajectory is not executing on Gazebo, the topics are listed

**\$ rostopic list**

- If the controllers are not ready, the trajectory will not execute in Gazebo, and the error messages are displayed in the terminal

Some issues we ran into was making the bridge between MoveIt!, and the Arm. After creating the URDF file and making sure all of the launch files and folders were organized. The error we ran into kept pointing towards syntax written in the URDF files from the packets downloaded from the user rst-tu-dortmund for the packet named pxpincher. We could troubleshoot the issue and change variable names or syntax to match description but ultimately catkin\_make was running into many issues and we had limited time so we decided to study controlling each servo manually through Arduino to understand the basics. Depicted below is some of the errors we stopped troubleshooting because of time limitations and project deadlines.

```
e 256, in _param_tag
    vals = self.opt_attrs(tag, context, ('value', 'textfile', 'binfile', 'command'))
File "/opt/ros/indigo/lib/python2.7/dist-packages/roslaunch/xmlloader.py", line 202, in opt_attrs
    return [self.resolve_args(tag_value(tag,a), context) for a in attrs]
File "/opt/ros/indigo/lib/python2.7/dist-packages/roslaunch/xmlloader.py", line 183, in resolve_args
    return substitution_args.resolve_args(args, context=context.resolve_dict, resolve_anon=self.resolve_anon)
File "/opt/ros/indigo/lib/python2.7/dist-packages/roslaunch/substitution_args.py", line 312, in resolve_args
    resolved = _resolve_args(resolved, context, resolve_anon, commands)
File "/opt/ros/indigo/lib/python2.7/dist-packages/roslaunch/substitution_args.py", line 325, in _resolve_args
    resolved = commands[command](resolved, a, args, context)
File "/opt/ros/indigo/lib/python2.7/dist-packages/roslaunch/substitution_args.py", line 141, in _find
    source_path_to_packages=source_path_to_packages)
File "/opt/ros/indigo/lib/python2.7/dist-packages/roslaunch/substitution_args.py", line 184, in _find_executable
    full_path = _get_executable_path(rp.get_path(args[0]), path)
File "/usr/lib/python2.7/dist-packages/rospkg/rospack.py", line 200, in get_path
    raise ResourceNotFound(name, ros_paths=self._ros_paths)
ResourceNotFound: pxpincher_description
ROS path [0]=/opt/ros/indigo/share/ros
ROS path [1]=/home/turtlebot/indigo/src
ROS path [2]=/home/turtlebot/ros/indigo/catkin_ws/src
ROS path [3]=/opt/ros/indigo/share
ROS path [4]=/opt/ros/indigo/stacks
turtlebot@CO-P-ROBOT16:~$
```

Ultimately interfacing Gazebo and MoveIt with your robot you can expect to run into many issues if you are new to ROS that will take time to troubleshoot. Once the URDF package is completed we can try to run Roslaunch and interface ROS with our equipment in this case the PhantomX Pincher.

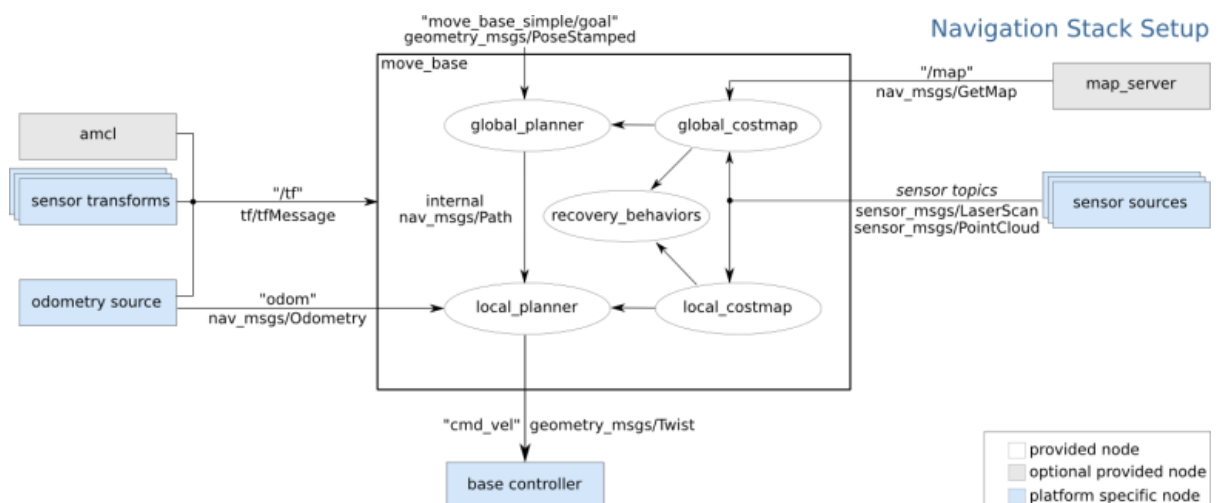
## Understanding ROS Navigation stack

The main aim of the ROS navigation package is to move a robot from the start position to the goal position, without making any collision with the environment.

The Navigation stack contains implementation of the standard algorithms, such as SLAM, A \*(star), Dijkstra, AMCL, and so on, which makes the robot navigate autonomously and thus requiring the user to only feed the goal position and robot odometry from various sensors.

### ROS Navigation hardware requirements

- The Navigation package will work better in differential drive and holonomic and the mobile robot should be controlled by sending velocity commands
- The robot should mount a planar laser somewhere around the robot. It is used to build the map of the environment
- The Navigation stack will perform better for square and circular shaped mobile bases.



The explanations of all the blocks which are provided as input to the Navigational stack

- **Odometry source:** Odometry data of a robot gives the robot position with respect to its starting position and also holds the velocity data and is a mandatory input. The message type is **nav\_msgs/Odometry**.
- **Sensor source:** The laser scan data or point cloud data to the Navigation stack for mapping the robot environment is provided. This data, along with odometry, combines to build the global and local cost map of the robot. The



main sensors used here are Laser Range finders or Kinect 3D sensors. The data should be of type **sensor\_msgs/LaserScan** or **sensor\_msgs/PointCloud**.

- **sensor transforms/tf**: The robot should publish the relationship between the robot coordinate frame using ROS tf.
- **base\_controller**: The main function of the base controller is to convert the output of the Navigation stack, which is a twist (**geometry\_msgs/Twist**) message, and convert it into corresponding motor velocities of the robot. The optional nodes of the Navigation stack are **amcl** and **map server**,

## Working with Navigation packages

### Understanding the **move\_base** node

The **move\_base** node is from a package called **move\_base**. The main function of this package is to move a robot from its current position to a goal position with the help of other Navigation nodes. The **move\_base** node inside this package links the **global-planner** and the **local-planner** for the path planning, connecting to the **rotate-recovery** package if the robot is stuck in some obstacle and connecting **global costmap** and **local costmap** for getting the map.

Information about the node:

- The node basically is an implementation of **SimpleActionServer** which takes a goal pose with message type (**geometry\_msgs/PoseStamped**). A goal position can be sent to this node using a **SimpleActionClient** node. The **move\_base** node subscribes the goal from a topic called **move\_base\_simple/goal**, which is the input of the Navigation stack.
- When this node receives a goal pose, it links to components such as **global\_planner**, **local\_planner**, **recovery\_behavior**, **global\_costmap**, and **local\_costmap**, generates the output which is the command velocity (**geometry\_msgs/Twist**), and sends to the base controller for moving the robot for achieving the goal pose.

The packages linked by the **move\_base** node:

- **global-planner**: This package provides libraries and nodes for planning the optimum path from the current position of the robot to the goal position, with respect to the robot map.

- local-planner: The main function of this package is to navigate the robot in a section of the global path planned using the global planner.
- rotate-recovery: This package helps the robot to recover from a local obstacle by performing a 360 degree rotation
- clear-costmap-recovery: This package is also for recovering from a local obstacle by clearing the costmap by reverting the current costmap used by the Navigation stack to the static map.
- costmap-2D: The main use of this package is to map the robot environment. Robot can only plan a path with respect to a map.
- map-server: Map server package allows us to save and load the map generated by the costmap-2D package.
- AMCL: AMCL is a method to localize the robot in map. This approach uses particle filter to track the pose of the robot with respect to the map, with the help of probability theory. In the ROS system, AMCL can only work with maps which were built using laser scans.
- gmapping: The gmapping package is an implementation of an algorithm called Fast SLAM which takes the laser scan data and odometry to build a 2D occupancy grid map.

Of course, these components come together when we want to use `move_base`, but there are certain ideas we need to understand before compiling all these tools together.

### **Running `move_base` on a Real Robot**

The `move_base` configuration is included in the `rbx1_nav/config/turtlebot` directory

#### **Working of Navigation stack**

- The required data to be published are, proper odometry value, tf information, and sensor data from the laser, and have a base controller and map of the surrounding.

#### **Localizing on the map**

- The first step the robot is going to perform is localizing itself on the map. The AMCL package will help to localize the robot on the map.

#### **Sending a goal and path planning**

- The `move_base` node will send this goal position to a global planner which will plan a path from the current robot position to the goal position. This plan is with respect to the global costmap which is feeding from the map server. The global planner will send this path to the local planner, which executes each segment of the global plan. The local planner gets the odometry and the sensor value from the `move_base` node and finds a collision free local plan for the robot. The local planner is associated with the local costmap, which can monitor the obstacle(s) around the robot.

### Collision recovery behavior

- The global and local costmap are tied with the laser scan data. If the robot is stuck somewhere, the Navigation package will trigger the recovery behavior nodes, such as the clear costmap recovery or rotate recovery nodes.

### Sending the command velocity

- The local planner generates command velocity in the form of a twist message which contains linear and angular velocity (geometry\_msgs/Twist), to the robot base controller. The robot base controller converts the twist message to the equivalent motor speed.

### Running move\_base on a Real Robot:

**`$roslaunch rbx1_bringup turtlebot_minimal_create.launch`**

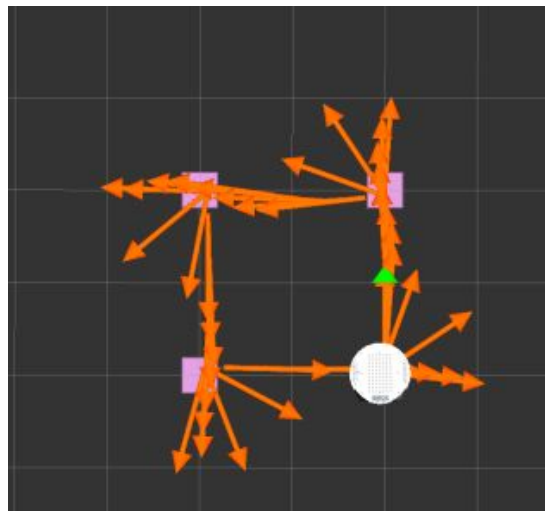
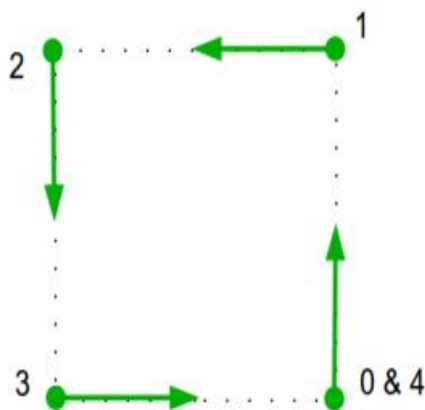
- The above command is the launch file for an original turtlebot.

**`$roslaunch rbx1_nav tb_move_base_my_map.launch`**

- This particular launch file loads a set of navigation parameters that should work fairly well with the original TurtleBot

**`$roslaunch rviz rviz -d `rospack find rbx1_nav`/nav.rviz`**

- The RViz is brought up with the above command.



### Building a map using SLAM

The ROS Gmapping package is a wrapper of open source implementation of SLAM called OpenSLAM. The package contains a node called `slam_gmapping`, which is the implementation of SLAM which helps to create a 2D occupancy grid map from the laser scan data and the mobile robot pose. The basic hardware requirement for doing SLAM is a laser scanner which is horizontally mounted on the top of the

robot, and the robot odometry data. The 2D map of the environment using the gmapping package through the following procedure.

### Creating a launch file for gmapping

- The main task while creating a launch file for the gmapping process is to set the parameters for the **slam\_gmapping** node and the **move\_base** node. The **slam\_gmapping** node subscribes the laser data (**sensor\_msgs/LaserScan**) and the **tf** data, and publishes the occupancy grid map data as output (**nav\_msgs/OccupancyGrid**). This node is highly configurable.
- The main parameters needed to configure the **move\_base** node are the global and local costmap parameters, the local planner, and the **move\_base** parameters.
- The launch file is placed in the `diff_wheeled_robot_gazebo/launch` folder.

### Running SLAM on the differential drive robot

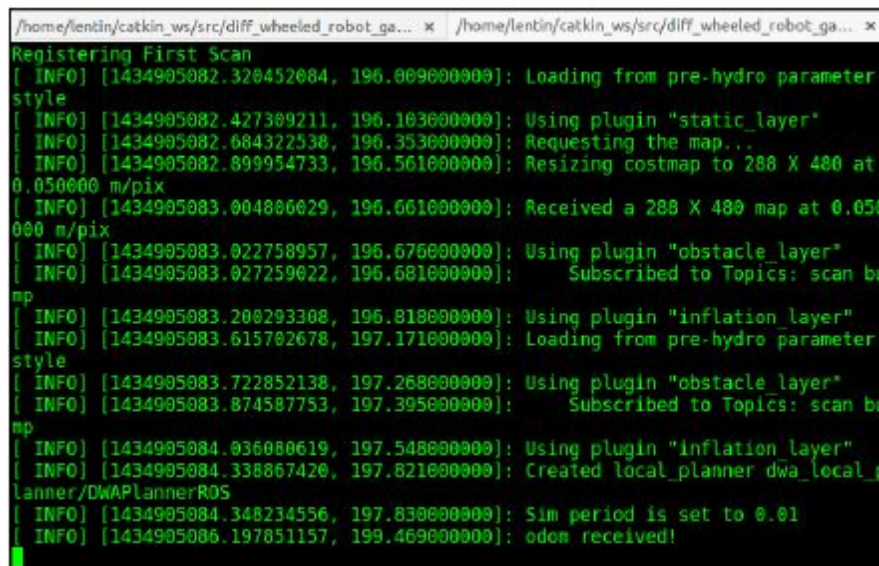
Start the robot simulation by using Willow Garage world:

```
$roslaunch diff_wheeled_robot_gazebo diff_wheeled_gazebo_full.launch
```

Start the gmapping launch file by using the following command:

```
$ roslaunch diff_wheeled_robot_gazebo gmapping.launch
```

- If the gmapping launch file is working fine, the following kind of output will appear on the terminal:

A terminal window screenshot showing the output of the gmapping launch file. The output includes several INFO messages from the ROS log, indicating the loading of pre-hydro parameters, the use of various plugins (static\_layer, obstacle\_layer, inflation\_layer), and the creation of the local planner (DWAPlannerROS). The terminal text is as follows:

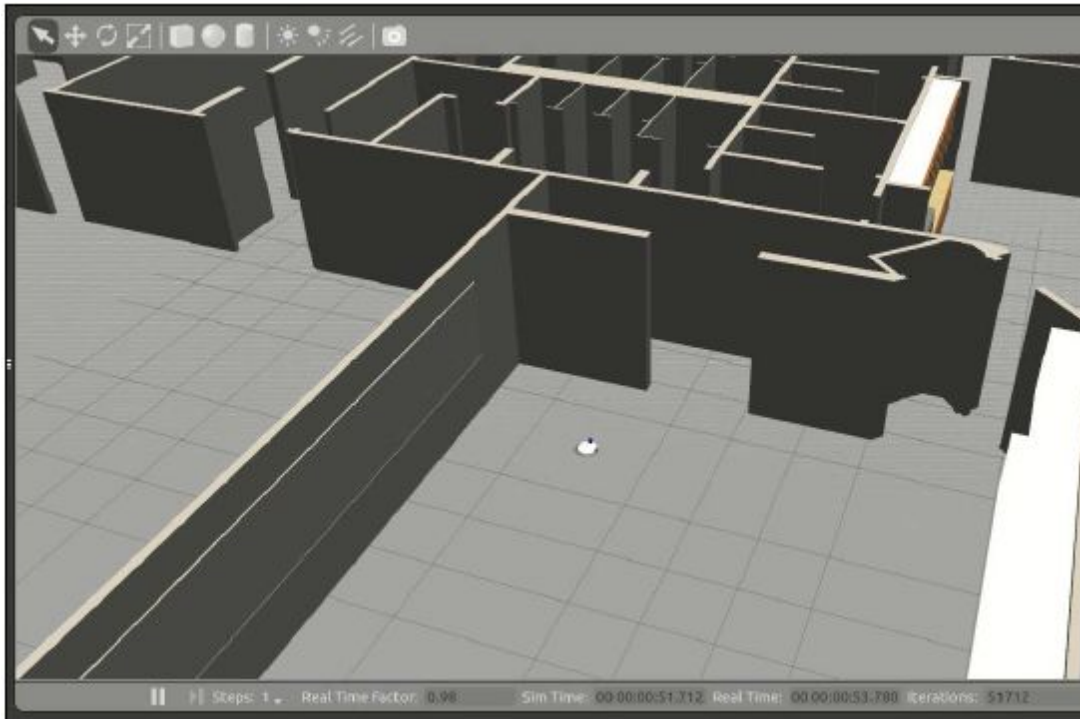
```
/home/lentin/catkin_ws/src/diff_wheeled_robot_ga... x /home/lentin/catkin_ws/src/diff_wheeled_robot_ga... x
Registering First Scan
[ INFO] [1434905082.320452084, 196.009000000]: Loading from pre-hydro parameter
style
[ INFO] [1434905082.427309211, 196.103000000]: Using plugin "static_layer"
[ INFO] [1434905082.684322538, 196.353000000]: Requesting the map...
[ INFO] [1434905082.899954733, 196.561000000]: Resizing costmap to 288 X 480 at
0.050000 m/pix
[ INFO] [1434905083.004806029, 196.661000000]: Received a 288 X 480 map at 0.050
000 m/pix
[ INFO] [1434905083.022758957, 196.676000000]: Using plugin "obstacle_layer"
[ INFO] [1434905083.027259022, 196.681000000]: Subscribed to Topics: scan bu
mp
[ INFO] [1434905083.200293308, 196.818000000]: Using plugin "inflation_layer"
[ INFO] [1434905083.615702678, 197.171000000]: Loading from pre-hydro parameter
style
[ INFO] [1434905083.722852138, 197.268000000]: Using plugin "obstacle_layer"
[ INFO] [1434905083.874587753, 197.395000000]: Subscribed to Topics: scan bu
mp
[ INFO] [1434905084.036080619, 197.548000000]: Using plugin "inflation_layer"
[ INFO] [1434905084.338867420, 197.821000000]: Created local_planner dwa_local_p
lanner/DWAPlannerROS
[ INFO] [1434905084.348234556, 197.830000000]: Sim period is set to 0.01
[ INFO] [1434905086.197851157, 199.469000000]: odom received!
```

```
$ roslaunch diff_wheeled_robot_control keyboard_teleop.launch
```

- The above command starts the keyboard teleoperation for manually navigating the robot around the environment.

```
$ rosrn map_server map_saver -f willo
```

The above command built map using the following command. This command will listen to the map topic and save into the image. The map server package does this operation.



- The current Gazebo view of the robot and the robot environment appear
- ```
$ rosrn map_server map_saver -f willo
```

I think gmapping is a useful aspect in robotics, we can use endless amounts of applications for example in China a shipping warehouse. Given enough time, we could manipulate the Arm to pick up an object and have the Turtlebot run with set goals and timers to move the object placed on the Turtlebot.



## Conclusion

This report has summarized information about Dynamixel programming and communication and how the dynamixel packets are written and read. It also talks about the uses of the Arduino IDE and ax12 library in programming dynamixel motors. Moreover, it also talks about the use of ax12SetRegister and ax12SetRegister2 in controlling several motors simultaneously. Dynamixel motors can be controlled in real-time using a serial connection and MATLAB as well as C++ as MATLAB just happens to be one of the easiest to implement and C++ is one of the most widely used. All of the basics regarding serial communication and serial objects will hold true for most languages. Many libraries exist out there to help the user read data from and write data to serial connections.

The second part of the report includes a brief overview of MoveIt! and the Navigation stack of ROS and demonstrates its capabilities using Gazebo simulation of a robotic arm mobile base. The chapter starts with a MoveIt! overview and discusses detailed concepts about MoveIt!. After discussing MoveIt!, Gazebo was interfaced. After interfacing, the trajectory from MoveIt! on Gazebo was executed. The next section was about the ROS Navigation with the move\_base packet. After trial and error we were able to upload a map and allow the turtlebot to navigate through the saved mapped environment from the Ladar. We did not successfully complete the integration of ROS and MoveIt! , but we were able to write a code in arduino that allows us to manipulate the arm. Ultimately, we navigated through ROS and its applications successfully by applying the knowledge learned through the introductory and have a good understanding on how to navigate and troubleshoot ROS errors given enough time.

## References:

- ❖ *ROS By Example A Do-It-Yourself Guide to the Robot Operating System VOLUME 1* by R. PATRICK GOEBEL
- ❖ *Mastering ROS for Robotics programming* by Lentil Joseph
- ❖ *Introduction to Dynamixel Motor Control Using the ArbotiX-M Robocontroller* by Daniel Jacobson
- ❖ [https://www.youtube.com/watch?v=\\_QndP\\_PCRSw](https://www.youtube.com/watch?v=_QndP_PCRSw) - Chinese delivery firm moving to embrace automation
- ❖

