Industrial Training

May 10, 2023

Contents

CHAPTER 1

Setup PC

1.1 PC Setup

There are two options for utilizing the ROS-Industrial training materials. The first **recommended** option is to utilize a pre-configured virtual machine. The second option is to install a native Ubuntu machine with the required software. The virtual machine approach is by far the easiest option and ensures the fewest build errors during training but is limited in its ability to connect to certain hardware, particularly over USB (i.e. kinect-like devices). For the perception training a .bag file is provided so that USB connection is not required for this training course.

1.1.1 Virtual Machine Configuration (Recommended)

The VM method is the most convenient method of utilizing the training materials:

1. Download virtual box
2. Download ROS Melodic training VM
3. Import image into virtual box
4. Start virtual machine
    1. *Note: If possible, assign two cores in Settings>>System>>Processor to your virtual machine before starting your virtual machine. This setting can be adjusted when the virtual machine is closed and shut down.
5. Log into virtual machine, user: `ros-industrial`, pass: `rosindustrial` (no spaces or hyphens)
6. Get the latest changes (Open Terminal).

```
cd ~/industrial_training
git fetch origin
git checkout melodic
git pull
./.check_training_config.bash
```

Limitations of Virtual Box

The Virtual Box is limited both in hardware capability(due to VM limitations) and package installs (to save space). Kinect-based demos aren't possible due to USB limitations.

Common VM Issues

On most new systems, Virtual Box and VMs work out of the box. The following is a list of issues others have encountered and solutions:

- Virtualization must be enabled - Older systems do not have virtualization enabled (by default). Virtualization must be enabled in the BIOS. See http://www.sysprobs.com/disable-enable-virtualization-technology-bios for more information.

1.1.2 Direct Linux PC Configuration

An installation shell script is provided to run in Ubuntu Linux 18.04 LTS (Bionic). This script installs ROS and any other packages needed for the environment used for this training.

After this step (or if you already have a working ROS environment), clone the training material repository into your home directory:

```
git clone -b melodic https://github.com/ros-industrial/industrial_training.git
~/industrial_training
```

1.1.3 Configuration Check

The following is a quick check to ensure that the appropriate packages have been installed and the the `industrial_training` git repository is current. Enter the following into the terminal:

```
~/industrial_training/.check_training_config.bash
```

CHAPTER 2

Prerequisites

2.1 C++

2.2 Linux Fundamentals

`Slides`

2.2.1 Navigating the Ubuntu GUI

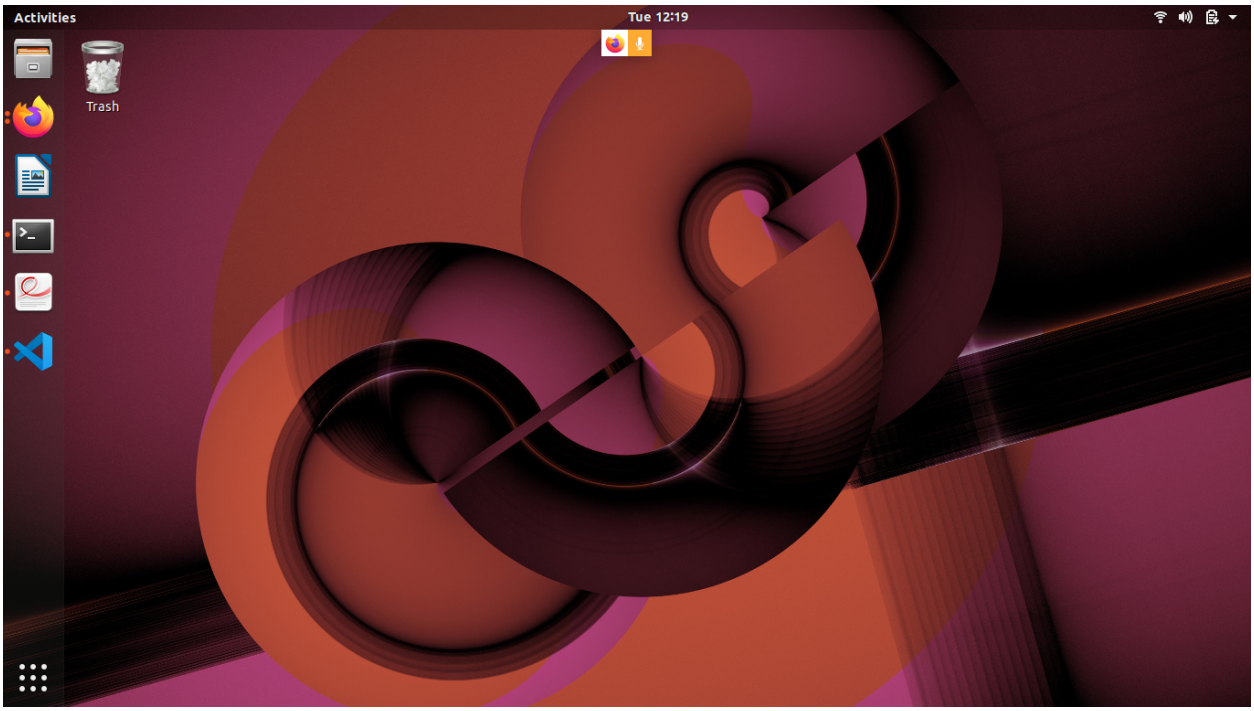
In this exercise, we will familiarize ourselves with the graphical user interface (GUI) of the Ubuntu operating system.

Task 0: Presentation Slides

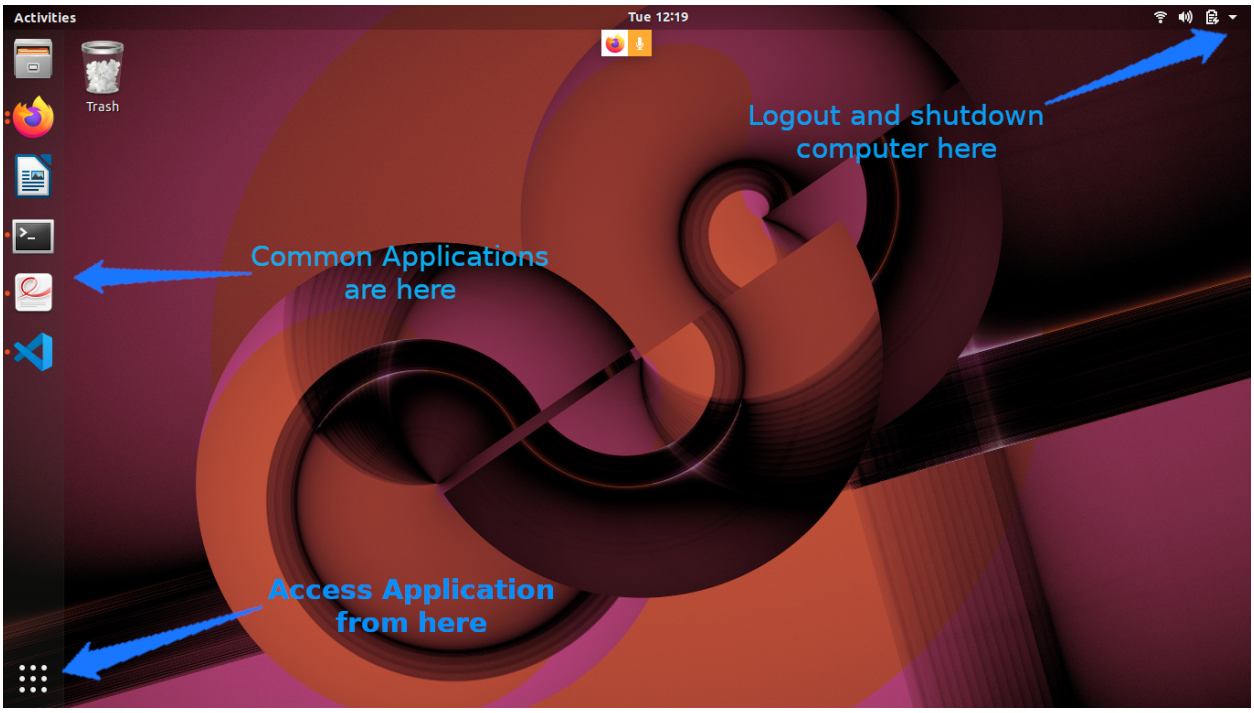
Don't forget about the presentation slides that accompany this Lesson!

Task 1: Familiarize Yourself with the Ubuntu Desktop

At the log-in screen, click in the password input box, enter `rosindustrial` for the password, and hit enter. The screen should look like the image below when you log in:



There are several things you will notice on the desktop:



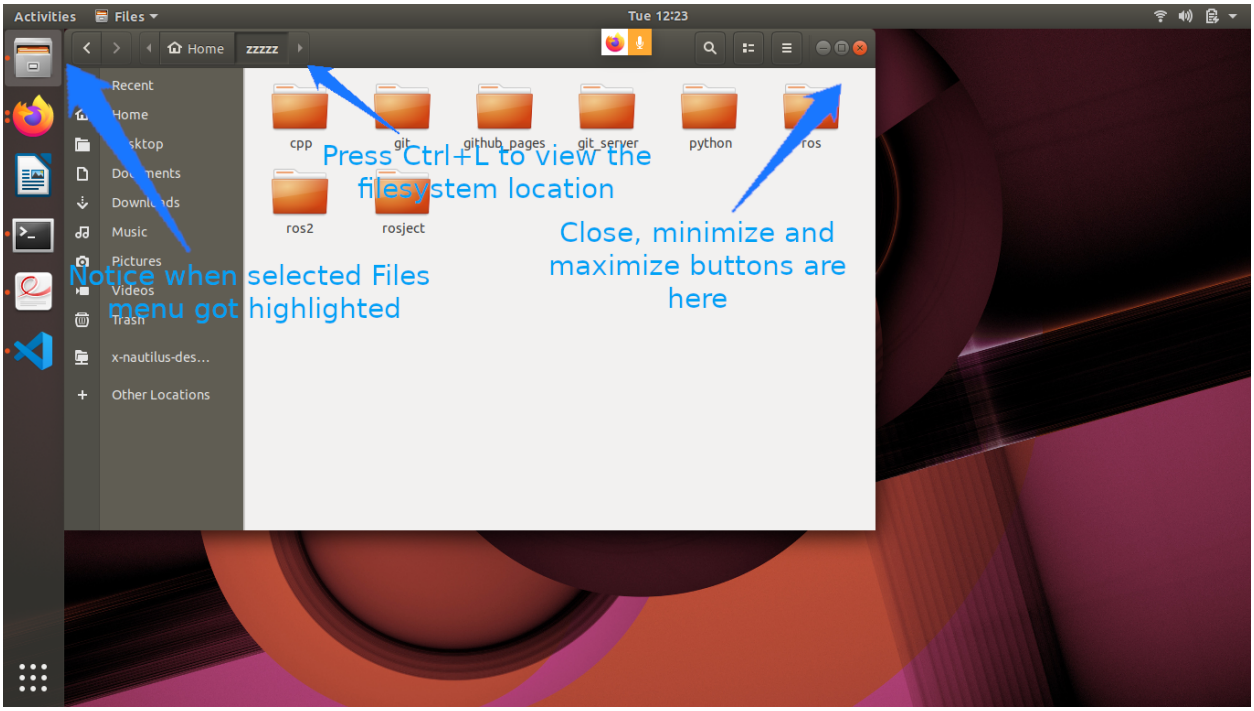
1. The gear icon on the top right of the screen brings up a menu which allows the user to log out, shut down the computer, access system settings, etc…
2. The bar on the left side shows running and "favorite" applications, connected thumb drives, etc.
3. The top icon is used to access all applications and files. We will look at this in more detail later.
4. The next icons are either applications which are currently running or have been "pinned" (again, more on pinning

later)

5. Any removable drives, like thumb drives, are found after the application icons.
6. If the launcher bar gets "too full", clicking and dragging up/down allows you to see the applications that are hidden.
7. To reorganize the icons on the launcher, click and hold the icon until it "pops out", then move it to the desired location.

Task 2: Open and Inspect an Application

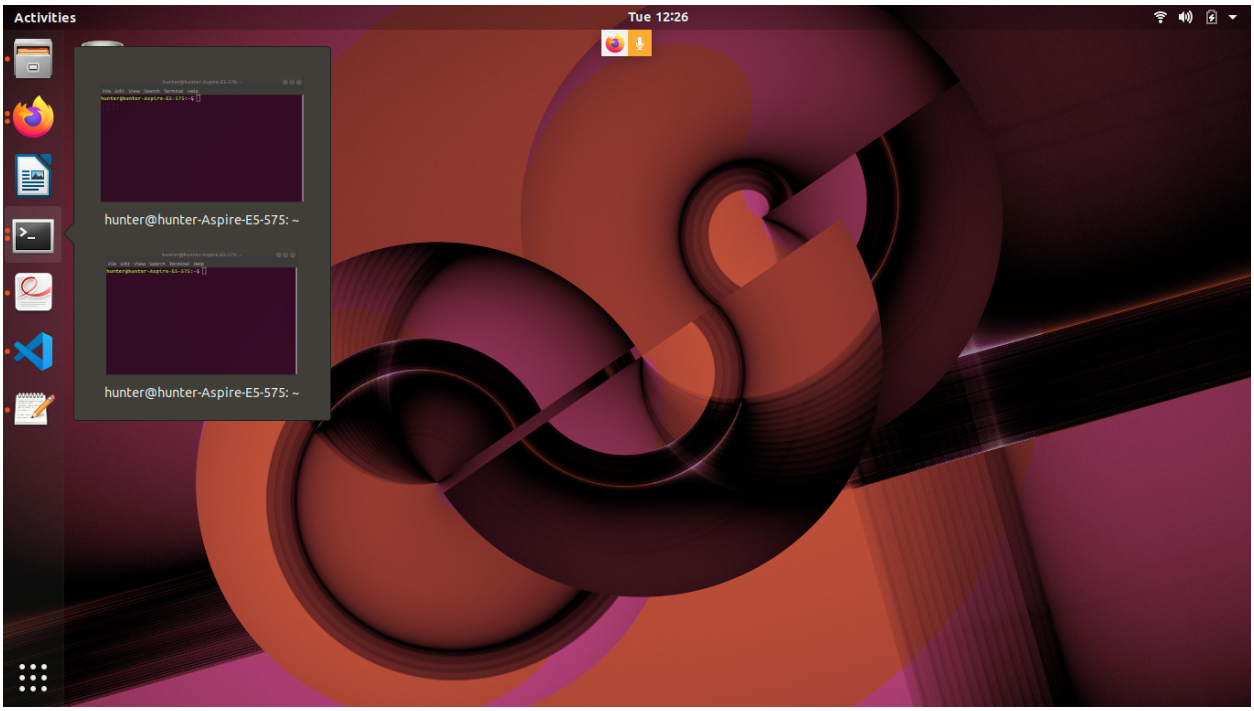
Click on the filing-cabinet icon in the launcher. A window should show up, and your desktop should look like something below:



Things to notice:

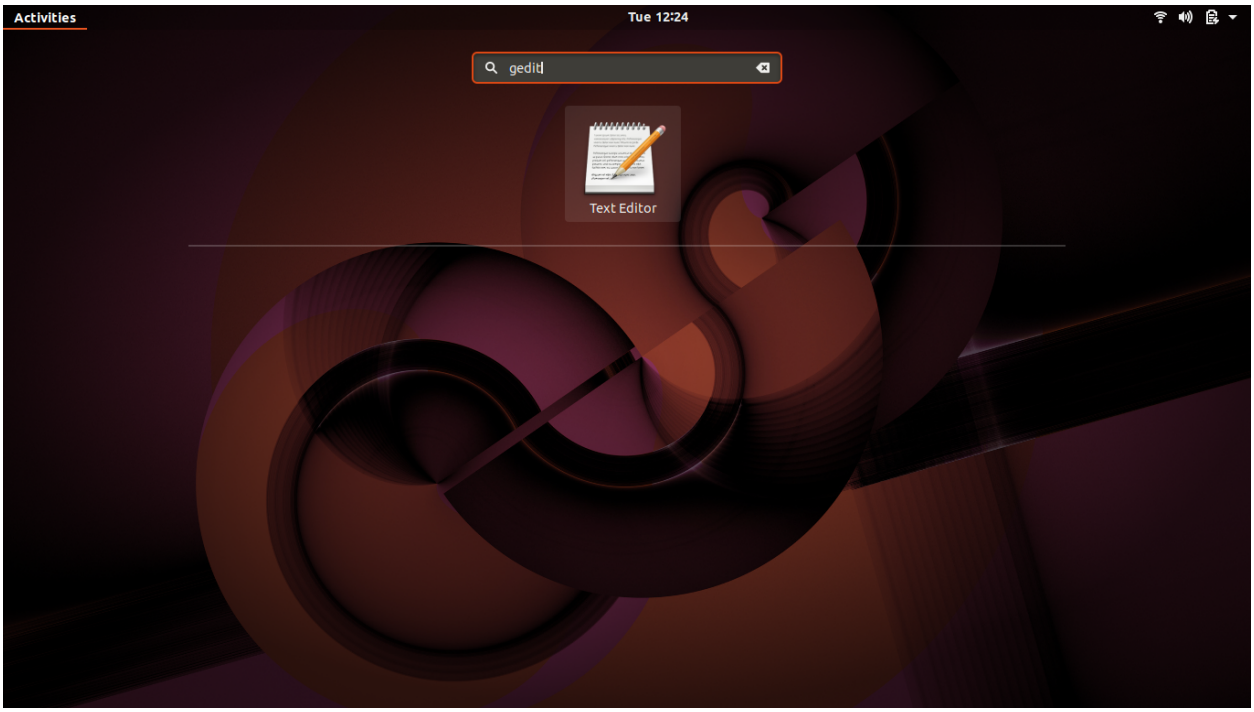
1. The close, minimize, and maximize buttons typically found on the right-hand side of the window title bar are found on the left-hand side.
2. The menu for windows are found on the menu bar at the top of the screen, much in the same way Macs do. The menus, however, only show up when you hover the mouse over the menu bar.
3. Notice that there are menu highlights of the folder icon. The dots on the left show how many windows of this application are open. Clicking on these icons when the applications are open does one of two things:

- If there is only one window open, this window gets focus.
- If more than one are open, clicking a second time causes all of the windows to show up in the foreground, so that you can choose which window to go to (see below):

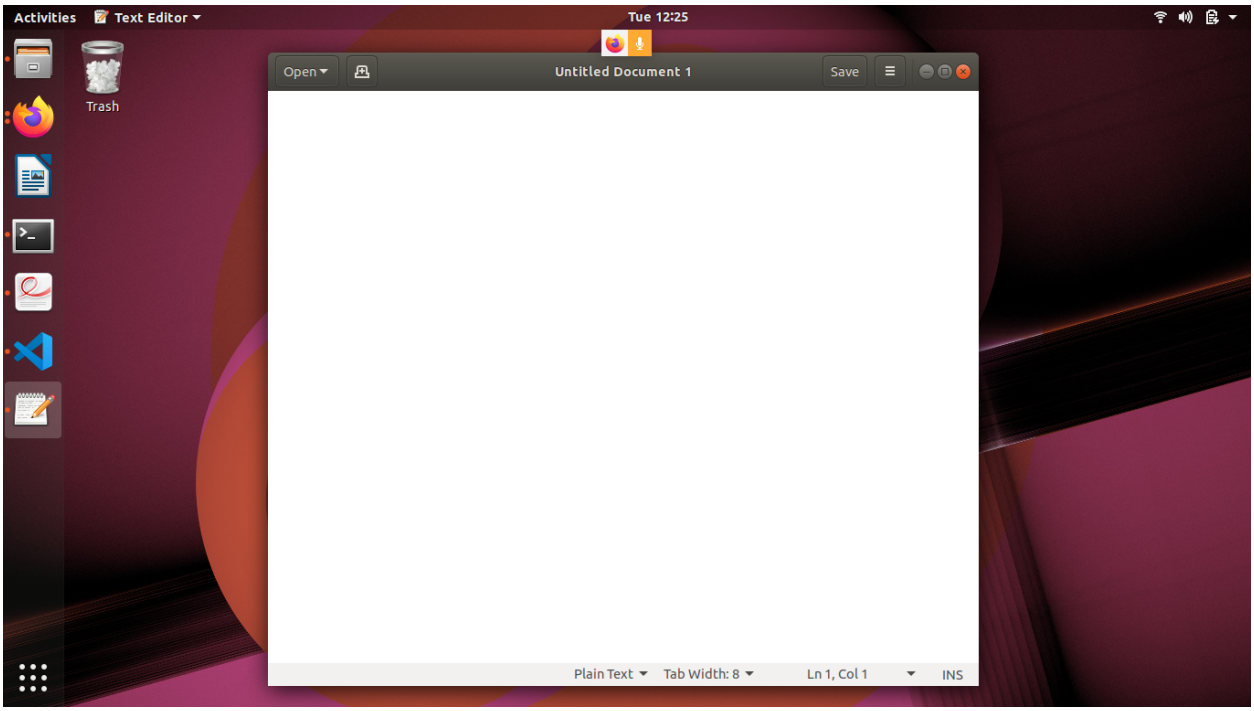


Task 3: Start an Application & Pin it to the Launcher Bar

Click on the launcher button (top left) and type gedit in the search box. The “Text Editor” application (this is actually gedit) should show up (see below):



Click on the application. The text editor window should show up on the screen, and the text editor icon should show up on the launcher bar on the left-hand side (see below):



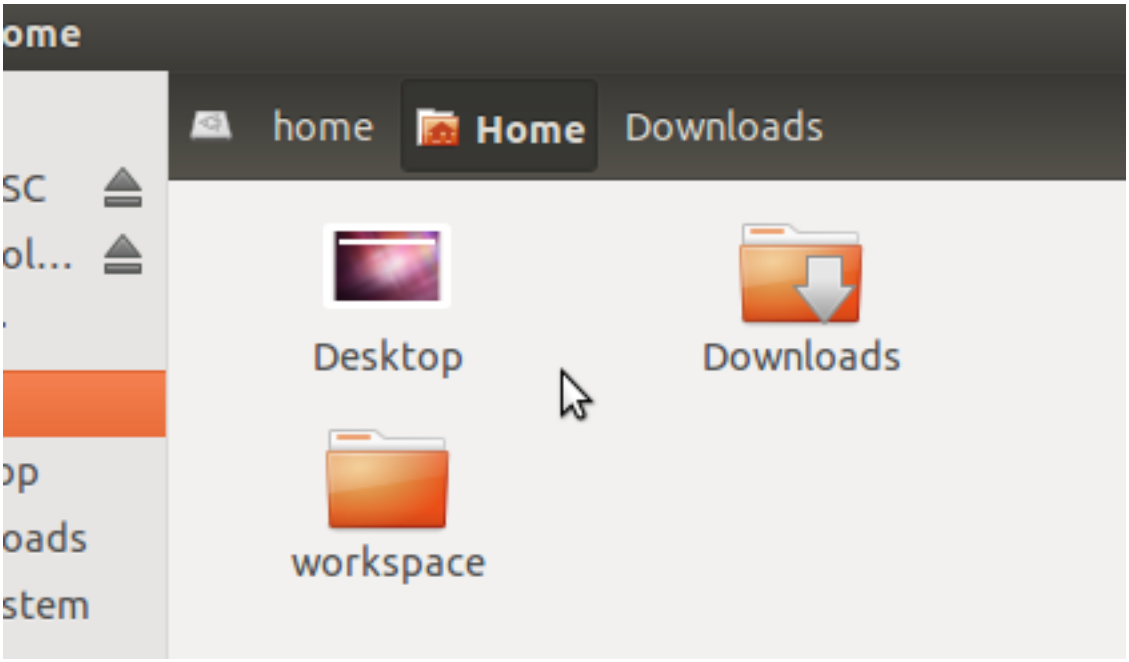
1. Right-click on the text editor launch icon, and select “Lock to Launcher”.
2. Close the gedit window. The launcher icon should remain after the window closes.
3. Click on the gedit launcher icon. You should see a new gedit window appear.

2.2.2 Exploring the Linux File System

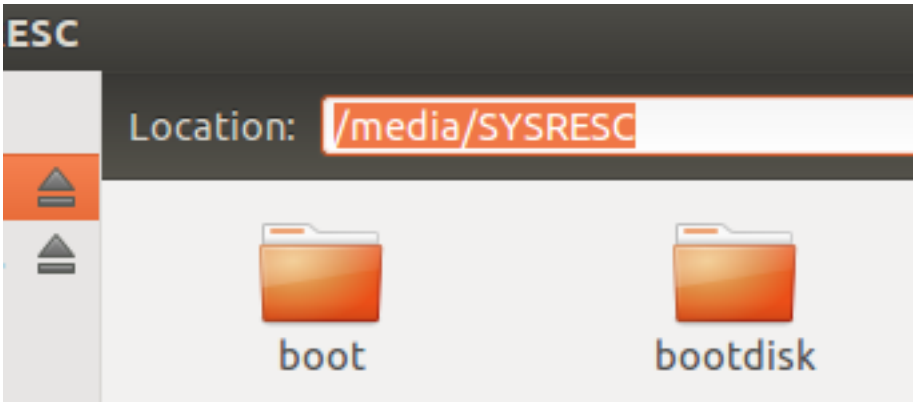
In this exercise, we will look at how to navigate and move files in the Linux file system using the Ubuntu GUI, and learn some of the basics of Linux file attributes.

Using the File Browser to Navigate

1. Open the folder browser application we opened in the previous exercise. You should see an window like the one below. The icons and text above the main window show the current location of the window in the file system.



1. The icons at the top constitute the "location bar" of the file browser. While the location bar is very useful for navigating in the GUI, it hides the exact location of the window. You can show the location by pressing *Ctrl+L*. You should see the location bar turn into something like the image below:



1. The folder browser opens up in the user's home folder by default. This folder is typically */home/*, which in the ROS-Industrial training computer is */home/ros-industrial*. This folder is the only one which the user has full access to. This is by design for security's sake.
2. By default, the file browser doesn't show hidden files (files which begin with a . character) or "backup" files (which end with a ~ character). To show these files, click on the "View" menu, and select "Show Hidden Files" (or press Ctrl+H). This will show all of the hidden files. Uncheck the option to re-hide those files.
3. Two hidden directories are *never* shown: The . folder, which is a special folder that represents the current folder, and .., which represents the folder which contains the current folder. These will become important in the next exercise.
4. On the left hand side of the window are some quick links to removable devices, other hard drives, bookmarks, etc. Click on the "Computer" shortcut link. This will take you to the "root" of the file system, the / folder. All of the files on the computer are in sub-folders under this folder.
5. Double click on the *opt* folder, then the *ros* folder. This is where all of the ROS software resides. Each version is stored in its own folder; we should see a "melodic" folder there. Double-click on that folder. The *setup.bash*

file will be used in the terminal exercise to configure the terminal for ROS. The programs, data, etc. are in the *bin* and *share* folders. You generally do not need to modify any of these files directly, but it is good to know where they reside.

Making Changes

Copying, Moving, and Removing Files

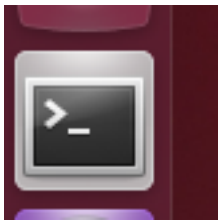
1. Create a directory and file
    1. Make a directory *<Home>/ex0.3*. We will be working within this folder.
        - Inside the file browser, click on the "Home" shortcut in the left sidebar.
        - Right click in the file browser's main panel and select "New Folder".
        - Name the folder "ex0.3" and press "return".
    2. Make a file *test.txt* inside the newly-created *ex0.3* folder.
        - Double-click on the *ex0.3* folder. Note how the File Browser header changes to show the current folder.
        - Right click in the file browser's main panel and select "New Document", then "Empty Document".
        - Name the file "test.txt" and press "return".
2. Copying Files
    1. Copy the file using one of the following methods:
        - Click and hold on the *test.txt* file, hold down on the control key, drag somewhere else on the folder, and release.
        - Click on the file, go to the "Copy" from the "Edit" menu, and then "Paste" from the "Edit" menu. *Remember: to see the Menu, hover your mouse above the bar at the top of the screen*
    2. Rename the copied file to *copy.txt* using one of the following methods:
        - Right-click on the copied file, select "Rename…" and enter *copy.txt*.
        - Click on the file, press the F2 key, and enter *copy.txt*.
    3. Create a folder *new* using one of the following methods:
        - Right-click on an open area of the file browser window, select "New Folder", and naming it *new*
        - Select "New Folder" from the "File" menu, and naming it *new*
    4. Move the file *copy.txt* into the *new* folder by dragging the file into the *new* folder.
    5. Copy the file *test.txt* by holding down the Control key while dragging the file into the new folder.
    6. Navigate into the *new* folder, and delete the *test.txt* folder by clicking on the file, and pressing the delete key.

2.2.3 The Linux Terminal

In this exercise, we will familiarize ourselves with the Linux terminal.

Starting the Terminal

1. To open the terminal, click on the terminal icon:



2. Create a second terminal window, either by:
   - Right-clicking on the terminal and selecting the "Open Terminal" or
   - Selecting "Open Terminal" from the "File" menu
3. Create a second terminal within the same window by pressing "Ctrl+Shift+T" while the terminal window is selected.
4. Close the 2nd terminal tab, either by:
   - clicking the small 'x' in the terminal tab (not the main terminal window)
   - typing `exit` and hitting enter.
5. The window will have a single line, which looks like this:

   ```
   ros-industrial@ros-i-melodic-vm:~$
   ```

6. This is called the prompt, where you enter commands. The prompt, by default, provides three pieces of information:
   1. *ros-industrial* is the login name of the user you are running as.
   2. *ros-i-melodic-vm* is the host name of the computer.
   3. ~ is the directory in which the terminal is currently in. (More on this later).
7. Close the terminal window by typing `exit` or clicking on the red 'x' in the window's titlebar.

Navigating Directories and Listing Files

Prepare your environment

1. Open your home folder in the file browser.
2. Double-click on the `ex0.3` folder we created in the previous step.
   - *We'll use this to illustrate various file operations in the terminal.*
3. Right click in the main file-browser window and select "Open in Terminal" to create a terminal window at that location.
4. In the terminal window, type the following command to create some sample files that we can study later:
   - `cp -a ~/industrial_training/exercises/0.3/. .`

ls Command

1. Enter `ls` into the terminal.
   - You should see `test.txt`, and `new` listed. (If you don't see 'new', go back and complete the previous exercise).
   - Directories, like `new`, are colored in blue.
   - The file `sample_job` is in green; this indicates it has its "execute" bit set, which means it can be executed as a command.
2. Type `ls *.txt`. Only the file `test.txt` will be displayed.
3. Enter `ls -l` into the terminal.
   - Adding the `-l` option shows one entry per line, with additional information about each entry in the directory.
   - The first 10 characters indicate the file type and permissions
   - The first character is `d` if the entry is a directory.
   - The next 9 characters are the permissions bits for the file
   - The third and fourth fields are the owning user and group, respectively.
   - The second-to-last field is the time the file was last modified.
   - If the file is a symbolic link, the link's target file is listed after the link's file name.
4. Enter `ls -a` in the terminal.
   - You will now see one additional file, which is hidden.
5. Enter `ls -a -l` (or `ls -al`) in the command.
   - You'll now see that the file `hidden_link.txt` points to `.hidden_text_file.txt`.

`pwd` and `cd` Commands

1. Enter `pwd` into the terminal.
   - This will show you the full path of the directory you are working in.
2. Enter `cd new` into the terminal.
   - The prompt should change to `ros-industrial@ros-i-melodic-vm:~/ex0.3/new$`.
   - Typing `pwd` will show you now in the directory `/home/ros-industrial/ex0.3/new`.
3. Enter `cd ..` into the terminal. * In the previous exercise, we noted that `..` is the parent folder. * The prompt should therefore indicate that the current working directory is `/home/ros-industrial/ex0.3`.
4. Enter `cd /bin`, followed by `ls`.
   - This folder contains a list of the most basic Linux commands. *Note that `pwd` and `ls` are both in this folder.*
5. Enter `cd ~/ex0.3` to return to our working directory.
   - Linux uses the ~ character as a shorthand representation for your home directory.
   - It's a convenient way to reference files and paths in command-line commands.
   - You'll be typing it a lot in this class... remember it!

If you want a full list of options available for any of the commands given in this section, type `man <command>` *(where* `<command>` *is the command you want information on) in the command line. This will provide you with built-in documentation for the command. Use the arrow and page up/down keys to scroll, and* `q` *to exit.*

Altering Files

mv Command

1. Type `mv test.txt test2.txt`, followed by `ls`.
   - You will notice that the file has been renamed to `test2.txt`. *This step shows how* `mv` *can rename files.*
2. Type `mv test2.txt new`, then `ls`.
   - The file will no longer be present in the folder.
3. Type `cd new`, then `ls`.
   - You will see `test2.txt` in the folder. *These steps show how* `mv` *can move files.*
4. Type `mv test2.txt ../test.txt`, then `ls`.
   - `test2.txt` will no longer be there.
5. Type `cd ..`, then `ls`.
   - You will notice that `test.txt` is present again. *This shows how* `mv` *can move and rename files in one step.*

cp Command

1. Type `cp test.txt new/test2.txt`, then `ls new`.
   - You will see `test2.txt` is now in the `new` folder.
2. Type `cp test.txt "test copy.txt"`, then `ls -l`.
   - You will see that `test.txt` has been copied to `test copy.txt`. *Note that the quotation marks are necessary when spaces or other special characters are included in the file name.*

rm Command

1. Type `rm "test copy.txt"`, then `ls -l`.
   - You will notice that `test copy.txt` is no longer there.

mkdir Command

1. Type `mkdir new2`, then `ls`.
   - You will see there is a new folder `new2`.

touch Command

1. Type `touch ~/Templates/"Untitled Document"`.
    - This will create a new Document named **"Untitled Document"**

You can use the `-i` flag with `cp`, `mv`, and `rm` commands to prompt you when a file will be overwritten or removed.

Job management

Stopping Jobs

1. Type `./sample_job`.
    - The program will start running.
2. Press Control+C.
    - The program should exit.
3. Type `./sample_job sigterm`.
    - The program will start running.
4. Press Control+C.
    - This time the program will not die.

Stopping "Out of Control" Jobs

1. Open a new terminal window.
2. Type `ps ax`.
3. Scroll up until you find `python ./sample_job sigterm`.
    - This is the job that is running in the first window.
    - The first field in the table is the ID of the process (use `man ps` to learn more about the other fields).
4. Type `ps ax | grep sample`.
    - You will notice that only a few lines are returned.
    - This is useful if you want to find a particular process
    - *Note: this is an advanced technique called "piping", where the output of one program is passed into the input of the next. This is beyond the scope of this class, but is useful to learn if you intend to use the terminal extensively.*
5. Type `kill <id>`, where `<id>` is the job number you found with the `ps ax`.
6. In the first window, type `./sample_job sigterm sigkill`.
    - The program will start running.
7. In the second window, type `ps ax | grep sample` to get the id of the process.
8. Type `kill <id>`.
    - This time, the process will not die.
9. Type `kill -SIGKILL <id>`.

- This time the process will exit.

Showing Process and Memory usage

1. In a terminal, type `top`.
   - A table will be shown, updated once per second, showing all of the processes on the system, as well as the overall CPU and memory usage.
2. Press the Shift+P key.
   - This will sort processes by CPU utilization. *This can be used to determine which processes are using too much CPU time.*
3. Press the Shift+M key.
   - This will sort processes by memory utilization *This can be used to determine which processes are using too much memory.*
4. Press q or Ctrl+C to exit the program.

Editing Text (and Other GUI Commands)

1. Type `gedit test.txt`.
   - You will notice that a new text editor window will open, and `test.txt` will be loaded.
   - The terminal will not come back with a prompt until the window is closed.
2. There are two ways around this limitation. Try both...
3. **Starting the program and immediately returning a prompt:**
   1. Type `gedit test.txt &`.
      - The `&` character tells the terminal to run this command in "the background", meaning the prompt will return immediately.
   2. Close the window, then type `ls`.
      - In addition to showing the files, the terminal will notify you that `gedit` has finished.
4. **Moving an already running program into the background:**
   1. Type `gedit test.txt`.
      - The window should open, and the terminal should not have a prompt waiting.
   2. In the terminal window, press Ctrl+Z.
      - The terminal will indicate that `gedit` has stopped, and a prompt will appear.
   3. Try to use the `gedit` window.
      - Because it is paused, the window will not run.
   4. Type `bg` in the terminal.
      - The `gedit` window can now run.
   5. Close the `gedit` window, and type `ls` in the terminal window.
      - As before, the terminal window will indicate that `gedit` is finished.

Running Commands as Root

1. In a terminal, type `ls -a /root`.
   - The terminal will indicate that you cannot read the folder `/root`.
   - Many times you will need to run a command that cannot be done as an ordinary user, and must be done as the "super user"
2. To run the previous command as root, add `sudo` to the beginning of the command.
   - In this instance, type `sudo ls -a /root` instead.
   - The terminal will request your password (in this case, `rosindustrial`) in order to proceed.
   - Once you enter the password, you should see the contents of the `/root` directory.

***Warning**: `sudo` is a powerful tool which doesn't provide any sanity checks on what you ask it to do, so be **VERY** careful in using it.*

CHAPTER 3

Basic Topics

3.1 Session 1 - ROS Concepts and Fundamentals

`Slides`

3.1.1 ROS-Setup

In this exercise, we will setup ROS to be used from the terminal, and start roscore

---

Motivation

In order to start programming in ROS, you should know how to install ROS on a new machine as well and check that the installation worked properly. This module will walk you through a few simple checks of your installed ROS system. Assuming you are working from the VM, you can skip any installation instructions as ROS is already installed.

Reference Example

Configuring ROS

Further Information and Resources

Installation Instructions

Navigating ROS

Scan-N-Plan Application: Problem Statement

We believe we have a good installation of ROS but let's test it to make sure.

Scan-N-Plan Application: Guidance

Setup ~/.bashrc

1. If you are ever having problems finding or using your ROS packages make sure that you have your environment properly setup. A good way to check is to ensure that environment variables like ROS_ROOT and ROS_PACKAGE_PATH are set:

```
printenv | grep ROS
```

2. If they are not then you might need to 'source' some setup.*sh files.

```
source /opt/ros/melodic/setup.bash
```

3. In a "bare" ROS install, you will need to run this command on every new shell you open to have access to the ROS commands. One of the setup steps in a *typical* ROS install is to add that command to the end of your `~/.bashrc` file, which is run automatically in every new terminal window. Check that your `.bashrc` file has already been configured to source the ROS-melodic `setup.bash` script:

```
tail ~/.bashrc
```

This process allows you to install several ROS distributions (e.g. indigo, kinetic, melodic) on the same computer and switch between them by sourcing the distribution-specific `setup.bash` file.

Starting roscore

1. *roscore* is a collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate. It is launched using the *roscore* command.

```
roscore
```

   roscore will start up:

   - a ROS Master
   - a ROS Parameter Server
   - a rosout logging node

   You will see ending with `started core service [/rosout]`. If you see `roscore: command not found` then you have not sourced your environment, please refer to section 5.1. .bashrc Setup.

2. To view the logging node, open a new terminal and enter:

```
rosnode list
```

   The logging node is named */rosout*

3. Press *Ctrl+C* in the first terminal window to stop roscore. Ctrl-C is the typical method used to stop most ROS commands.

3.1.2 Create Catkin Workspace

In this exercise, we will create a ROS catkin workspace.

Motivation

Any ROS project begins with making a workspace. In this workspace, you will put all the things related to this particular project. In this module we will create the workspace where we will build the components of our Scan-N-Plan application.

Reference Example

Steps to creating a workspace: Creating a Catkin Workspace

Note: Many current examples on ros.org use the older-style `catkin_init_workspace` commands. These are similar, but not directly interchangeable with the `catkin_tools` commands used in this course.

Further Information and Resources

Using a Catkin Workspace: Using a Workspace

Scan-N-Plan Application: Problem Statement

We have a good installation of ROS, and we need to take the first step to setting up our particular application. Your goal is to create a workspace - a catkin workspace - for your application and its supplements.

Scan-N-Plan Application: Guidance

Create a Catkin Workspace

1. Create the root workspace directory (we'll use `catkin_ws`)

```
cd ~/
mkdir --parents catkin_ws/src
cd catkin_ws
```

2. Initialize the catkin workspace

```
catkin init
```

   - *Look for the statement "Workspace configuration appears valid", showing that your catkin workspace was created successfully. If you forgot to create the `src` directory, or did not run `catkin init` from the workspace root (both common mistakes), you'll get an error message like "WARNING: Source space does not yet exist".*

3. Build the workspace. This command may be issued anywhere under the workspace root-directory (i.e. `catkin_ws`).

```
catkin build
ls
```

   - *See that the `catkin_ws` directory now contains additional directories (build, devel, logs).*

4. These new directories can be safely deleted at any time (either manually, or using `catkin clean`). Note that catkin never changes any files in the `src` directory. Re-run `catkin build` to re-create the build/devel/logs directories.

```
catkin clean
ls
catkin build
ls
```

5. Make the workspace visible to ROS. Source the setup file in the devel directory.

```
source devel/setup.bash
```

- *This file MUST be sourced for every new terminal.*
- To save typing, add this to your `~/.bashrc` file, so it is automatically sourced for each new terminal:
  1. `gedit ~/.bashrc`
  2. add to the end: `source ~/catkin_ws/devel/setup.bash`
  3. save and close the editor

3.1.3 Installing Packages

Motivation

Many of the coolest and most useful capabilities of ROS already exist somewhere in its community. Often, stable resources exist as easily downloadable debian packages. Alternately, some resources are less tested or more "cutting edge" and have not reached a stable release state; you can still access many of these resources by downloading them from their repository (usually housed on Github). Getting these git packages takes a few more steps than the debian packages. In this module we will access both types of packages and install them on our system.

Reference Example

apt-get usage

Further Information and Resources

Ubuntu apt-get How To

Git Get Repo

Git Clone Documentation

Scan-N-Plan Application: Problem Statement

We have a good installation of ROS, and we have an idea of some packages that exist in ROS that we would like to use within our program. We have found a package which is stable and has a debian package we can download. We've also found a less stable git package that we are interested in. Go out into the ROS world and download these packages!

1. A certain message type exists which you want to use. The stable ROS package is called: calibration_msgs
2. You are using an AR tag, but for testing purposes you would like a node to publish similar info : fake_ar_publisher

Your goal is to have access to both of these packages' resources within your package/workspace:

1. calibration_msgs (using apt-get)

2. fake_ar_publisher (from git)

Scan-N-Plan Application: Guidance

Install Package from apt Repository

1. Open a terminal window. Type roscd calibration_msgs.

```
roscd calibration_msgs
```

   - This command changes the working directory to the directory of the ROS *calibration_msgs* package.
   - You should see an error message '**No such package/stack 'calibration_msgs**' .
   - *This package is not installed on the system, so we will install it.*

2. Type *apt install ros-melodic-calibration-msgs*.

```
apt install ros-melodic-calibration-msgs
```

   - The program will say it cannot install the package, and suggests that we must run the program as root.
   - Try pressing the *TAB* key while typing the package name.
     - The system will try to automatically complete the package name, if possible.
     - Frequent use of the TAB key will help speed up entry of many typed commands.

3. Type *sudo apt install ros-melodic-calibration-msgs*.

```
sudo apt install ros-melodic-calibration-msgs
```

   - Note the use of the *sudo* command to run a command with "root" (administrator) privileges.
   - Enter your password, and (if asked) confirm you wish to install the program.

4. Type *roscd calibration_msgs* again.

```
roscd calibration_msgs
```

   - This time, you will see the working directory change to */opt/ros/melodic/share/calibration_msgs*.

5. Type *sudo apt remove ros-melodic-calibration-msgs* to remove the package.

```
sudo apt remove ros-melodic-calibration-msgs
```

   - *Don't worry. We won't be needing this package for any future exercises, so it's safe to remove.*

6. Type *cd ~* to return to your home directory.

```
cd ~
```

Download and Build a Package from Source

1. Identify the source repository for the desired package:
   1. Go to github.
   2. Search for fake_ar_publisher.

3. Click on this repository, and look to the right for the *Clone or Download*, then copy to clipboard.

2. Clone the *fake_ar_publisher* repository into the catkin workspace's *src* directory.

```
cd ~/catkin_ws/src
git clone https://github.com/jmeyer1292/fake_ar_publisher.git
```

   - *Use Ctrl-Shift-V to paste within the terminal, or use your mouse to right-click and select paste*
   - *Git commands are outside of the scope of this class, but there are good tutorials available here*

3. Build the new package using `catkin build` *The build command can be issued from anywhere inside the catkin workspace*

4. Once the build completes, notice the instruction to *"re-source setup files to use them"*.

   - In the previous exercise, we added a line to our `~/.bashrc` file to automatically re-source the catkin setup files in each new terminal.
   - This is sufficient for most development activities, but you may sometimes need to re-execute the `source` command in your current terminal (e.g. when adding new packages):

```
source ~/catkin_ws/devel/setup.bash
```

5. Once the build completes, explore the *build* and *devel* directories to see what files were created.

6. Run *rospack find fake_ar_publisher* to verify the new packages are visible to ROS.

```
rospack find fake_ar_publisher
```

   - This is a helpful command to troubleshoot problems with a ROS workspace.
   - If ROS can't find your package, try re-building the workspace and then re-sourcing the workspace's `setup.bash` file.

3.1.4 Creating Packages and Nodes

In this exercise, we will create our own ROS package and node.

Motivation

The basis of ROS communication is that multiple executables called nodes are running in an environment and communicating with each other in various ways. These nodes exist within a structure called a package. In this module we will create a node inside a newly created package.

Reference Example

Create a Package

Further Information and Resources

Building Packages

Understanding Nodes

Scan-N-Plan Application: Problem Statement

We've installed ROS, created a workspace, and even built a few times. Now we want to create our own package and our own node to do what we want to do.

Your goal is to create your first ROS node:

1. First you need to create a package inside your catkin workspace.
2. Then you can write your own node

Scan-N-Plan Application: Guidance

Create a Package

1. cd into the catkin workspace src directory *Note: Remember that all packages should be created inside a workspace src directory.*

```
cd ~/catkin_ws/src
```

2. Use the ROS command to create a package called *myworkcell_core* with a dependency on *roscpp*

```
catkin create pkg myworkcell_core --catkin-deps roscpp
```

See the catkin_tools documentation for more details on this command.

- *This command creates a directory and required files for a new ROS package.*
- *The first argument is the name of the new ROS package.*
- *Use `--catkin-deps` to specify packages which the newly created package depends on.*

3. There will now be a folder named *myworkcell_core*. Change into that folder and edit the *package.xml* file. Edit the file and change the description, author, etc., as desired.

```
cd myworkcell_core
gedit package.xml
```

If you forget to add a dependency when creating a package, you can add additional dependencies in the package.xml *file.*

STOP! We'll go through a few more lecture slides before continuing this exercise.

Create a Node

1. In the package folder, create the file *src/vision_node.cpp* (using *gedit*).
2. Add the ros header (include ros.h).

```
/**
**  Simple ROS Node
**/
#include <ros/ros.h>
```

3. Add a main function (typical in c++ programs).

```
/**
**  Simple ROS Node
**/
#include <ros/ros.h>

int main(int argc, char* argv[])
{

}
```

4. Initialize your ROS node (within the main).

```
/**
**  Simple ROS Node
**/
#include <ros/ros.h>

int main(int argc, char* argv[])
{
  // This must be called before anything else ROS-related
  ros::init(argc, argv, "vision_node");
}
```

5. Create a ROS node handle.

```
/**
**  Simple ROS Node
**/
#include <ros/ros.h>

int main(int argc, char* argv[])
{
  // This must be called before anything else ROS-related
  ros::init(argc, argv, "vision_node");

  // Create a ROS node handle
  ros::NodeHandle nh;
}
```

6. Print a "Hello World" message using ROS print tools.

```
/**
**  Simple ROS Node
**/
#include <ros/ros.h>

int main(int argc, char* argv[])
{
  // This must be called before anything else ROS-related
  ros::init(argc, argv, "vision_node");

  // Create a ROS node handle
  ros::NodeHandle nh;

  ROS_INFO("Hello, World!");
}
```

7. Do not exit the program automatically - keep the node alive.

```
/**
**  Simple ROS Node
**/
#include <ros/ros.h>

int main(int argc, char* argv[])
{
  // This must be called before anything else ROS-related
  ros::init(argc, argv, "vision_node");

  // Create a ROS node handle
  ros::NodeHandle nh;

  ROS_INFO("Hello, World!");

  // Don't exit the program.
  ros::spin();
}
```

ROS_INFO is one of the many logging methods.

- It will print the message to the terminal output, and send it to the */rosout* topic for other nodes to monitor.
- There are 5 levels of logging: *DEBUG, INFO, WARNING, ERROR, & FATAL.*
- To use a different logging level, replace INFO in *ROS_INFO* or *ROS_INFO_STREAM* with the appropriate level.
- Use *ROS_INFO* for printf-style logging, and *ROS_INFO_STREAM* for cout-style logging.

8. Now that you have created the source code for the node, we need to add instructions for building it into an executable program. In the package folder, edit the `CMakeLists.txt` file. Browse through the example rules, and add an executable (`add_executable`) named `vision_node` with the source file named `src/vision_node.cpp`. Also within the `CMakeLists.txt`, make sure your new `vision_node` executable gets linked (`target_link_libraries`) to the catkin libraries.

```
project(myworkcell_core)
+ add_compile_options(-std=c++11)

...

############
## BUILD ##
############

include_directories(
  ${catkin_INCLUDE_DIRS}
)

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

These lines should be added to the `CMakeLists.txt` in the order and location specified above. The commands already exist in the template `CMakeLists.txt` file and can be uncommented and slightly modified

- Uncomment existing template examples for `add_compile_options` near the top (just below `project()`)
- Uncomment and edit existing template examples for `add_executable` and `target_link_libraries` near the bottom in the *BUILD* section

- This helps make sure these rules are defined in the correct order, and makes it easy to remember the proper syntax.

Note: You're also allowed to spread most of the CMakeLists rules across multiple lines, as shown in the `target_link_libraries` *template code*

9. Build your program (node), by running `catkin build` in a terminal window
   - *Remember that you must run* `catkin build` *from within your* `catkin_ws` *(or any subdirectory)*
   - This will build all of the programs, libraries, etc. in *myworkcell_core*
   - In this case, it's just a single ROS node *vision_node*

Run a Node

1. Open a terminal and start the ROS master.

```
roscore
```

   The ROS Master must be running before any ROS nodes can function.

2. Open a second terminal to run your node.
   - In a previous exercise, we added a line to our `.bashrc` to automatically source `devel/setup.bash` in new terminal windows
   - This will automatically export the results of the build into your new terminal session.
   - If you're reusing an existing terminal, you'll need to manually source the setup files (since we added a new node):

```
source ~/catkin_ws/devel/setup.bash
```

3. Run your node.

```
rosrun myworkcell_core vision_node
```

   This runs the program we just created. Remember to use TAB to help speed-up typing and reduce errors.

4. In a third terminal, check what nodes are running.

```
rosnode list
```

   In addition to the /rosout node, you should now see a new /vision_node listed.

5. Enter *rosnode kill /vision_node*. This will stop the node.

   Note: It is more common to use Ctrl+C to stop a running node in the current terminal window.

Challenge

Goal: Modify the node so that it prints your name. This will require you to run through the build process again.

3.1.5 Topics and Messages

In this exercise, we will explore the concept of ROS messages and topics.

Motivation

The first type of ROS communication that we will explore is a one-way communication called messages which are sent over channels called topics. Typically one node publishes messages on a topic and another node subscribes to messages on that same topic. In this module we will create a subscriber node which subscribes to an existing publisher (topic/message).

Reference Example

Create a Subscriber

Further Information and Resources

Understanding Topics

Examining Publisher & Subscriber

Creating Messages and Services

Scan-N-Plan Application: Problem Statement

We now have a base ROS node and we want to build on this node. Now we want to create a subscriber within our node.

Your goal is to create your first ROS subscriber:

1. First you will want to find out the message structure.
2. You also want to determine the topic name.
3. Last you can write the c++ code which serves as the subscriber.

Scan-N-Plan Application: Guidance

Add the fake_ar_publisher Package as a Dependency

1. Edit your package's `CMakeLists.txt` file (`~/catkin_ws/src/myworkcell_core/CMakeLists.txt`). Make the following changes in the matching sections of the existing template file, by uncommenting and/or editing existing rules.
    1. Tell cmake to find the fake_ar_publisher package:

```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  fake_ar_publisher
)
```

    2. Add The catkin runtime dependency for publisher.

```
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## LIBRARIES: libraries you create in this project that dependent projects␣
↪also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also␣
↪need
catkin_package(
 #  INCLUDE_DIRS include
 #  LIBRARIES myworkcell_core
   CATKIN_DEPENDS
     roscpp
     fake_ar_publisher
#  DEPENDS system_lib
)
```

3. Uncomment/edit the `add_dependencies` line **below** your `add_executable` rule:

```
add_dependencies(vision_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_
↪EXPORTED_TARGETS})
```

2. add dependencies into your package's `package.xml`:

```
<depend>fake_ar_publisher</depend>
```

3. `cd` into your catkin workspace

```
cd ~/catkin_ws
```

4. Build your package and source the setup file to activate the changes in the current terminal.

```
catkin build
source ~/catkin_ws/devel/setup.bash
```

5. In a terminal, enter `rosmsg list`. You will notice that, included in the list, is `fake_ar_publisher/ARMarker`. If you want to see only the messages in a package, type `rosmsg package <package_name>`

6. Type `rosmsg show fake_ar_publisher/ARMarker`. The terminal will return the types and names of the fields in the message.

   Note that three fields under the `header` field are indented, indicating that these are members of the `std_msgs/Header` message type

Run a Publisher Node

1. In a terminal, type `rosrun fake_ar_publisher fake_ar_publisher_node`. You should see the program start up and begin publishing messages.
2. In another terminal, enter `rostopic list`. You should see `/ar_pose_marker` among the topics listed. Entering `rostopic type /ar_pose_marker` will return the type of the message.
3. Enter `rostopic echo /ar_pose_marker`. The terminal will show the fields for each message as they come in, separated by a `---` line. Press Ctrl+C to exit.
4. Enter `rqt_plot`.
   1. Once the window opens, type `/ar_pose_marker/pose/pose/position/x` in the "Topic:" field and click the "+" button. You should see the X value be plotted.

2. Type `/ar_pose_marker/pose/pose/position/y` in the topic field, and click on the add button. You will now see both the x and y values being graphed.
3. Close the window

5. Leave the publisher node running for the next task.

Create a Subscriber Node

1. Edit the `vision_node.cpp` file.
2. Include the message type as a header

```
#include <fake_ar_publisher/ARMarker.h>
```

3. Add the code that will be run when a message is received from the topic (the callback).

```
class Localizer
{
public:
  Localizer(ros::NodeHandle& nh)
  {
      ar_sub_ = nh.subscribe<fake_ar_publisher::ARMarker>("ar_pose_marker", 1,
      &Localizer::visionCallback, this);
  }

  void visionCallback(const fake_ar_publisher::ARMarkerConstPtr& msg)
  {
      last_msg_ = msg;
      ROS_INFO_STREAM(last_msg_->pose.pose);
  }

  ros::Subscriber ar_sub_;
  fake_ar_publisher::ARMarkerConstPtr last_msg_;
};
```

4. Add the code that will connect the callback to the topic (within `main()`)

```
int main(int argc, char** argv)
{
  ...
  // The Localizer class provides this node's ROS interfaces
  Localizer localizer(nh);

  ROS_INFO("Vision node starting");
  ...
}
```

- You can replace or leave the "Hello World" print... your choice!
- These new lines must go below the `NodeHandle` declaration, so `nh` is actually defined.
- Make sure to retain the `ros::spin()` call. It will typically be the last line in your `main` routine. Code after `ros::spin()` won't run until the node is shutting down.

5. Run `catkin build`, then `rosrun myworkcell_core vision_node`.
6. You should see the positions display from the publisher.
7. Press Ctrl+C on the publisher node. The subscriber will stop displaying information.

8. Start the publisher node again. The subscriber will continue to print messages as the new program runs.
    - This is a key capability of ROS, to be able to restart individual nodes without affecting the overall system.
9. In a new terminal, type `rqt_graph`. You should see a window similar to the one below:

- The rectangles in the the window show the topics currently available on the system.
- The ovals are ROS nodes.
- Arrows leaving the node indicate the topics the node publishes, and arrows entering the node indicate the topics the node subscribes to.

3.2 Session 2 - Basic ROS Applications

`Slides`

3.2.1 Services

In this exercise, we will create a custom service by defining a .srv file. Then we will write server and client nodes to utilize this service.

Motivation

The first type of ROS communication that we explored was a one-way interaction called messages which are sent over channels called topics. Now we are going to explore a different communication type, which is a two-way interaction via a request from one node to another and a response from that node to the first. In this module we will create a service server (waits for request and comes up with response) and client (makes request for info then waits for response).

Reference Example

Create a Service Server/Client

Further Information and Resources

- Creating Messages & Services
- Understanding Services & Params
- Examining Service Client

Scan-N-Plan Application: Problem Statement

We now have a base ROS node which is subscribing to some information and we want to build on this node. In addition we want this node to serve as a sub-function to another "main" node. The original vision node will now be responsible for subscribing to the AR information and responding to requests from the main workcell node.

Your goal is to create a more intricate system of nodes:

1. Update the vision node to include a service server
2. Create a new node which will eventually run the Scan-N-Plan App
    - First, we'll create the new node (myworkcell_core) as a service client. Later, we will expand from there

Scan-N-Plan Application: Guidance

Create Service Definition

1. Similar to the message file located in the fake_ar_publisher package, we need to create a service file. The following is a generic structure of a service file:

```
#request
---
#response
```

2. Create a folder called `srv` inside your `myworkcell_core` package (at same level as the package's `src` folder)

```
cd ~/catkin_ws/src/myworkcell_core
mkdir srv
```

3. Create a file (gedit or QT) called `LocalizePart.srv` inside the `srv` folder.

4. Inside the file, define the service as outlined above with a request of type `string` named `base_frame` and a response of type `geometry_msgs/Pose` named `pose`:

```
#request
string base_frame
---
#response
geometry_msgs/Pose pose
```

5. Edit the package's `CMakeLists.txt` and `package.xml` to add dependencies on key packages:

   - `message_generation` is required to build C++ code from the .srv file created in the previous step
   - `message_runtime` provides runtime dependencies for new messages
   - `geometry_msgs` provides the `Pose` message type used in our service definition

   1. Edit the package's `CMakeLists.txt` file to add the new **build-time** dependencies to the existing `find_package` rule:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  fake_ar_publisher
  geometry_msgs
  message_generation
)
```

   2. Also in `CMakeLists.txt`, add the new **run-time** dependencies to the existing `catkin_package` rule:

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES myworkcell_node
  CATKIN_DEPENDS
    roscpp
    fake_ar_publisher
    message_runtime
    geometry_msgs
#  DEPENDS system_lib
)
```

3. Edit the `package.xml` file to add the appropriate build/run dependencies:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
<depend>geometry_msgs</depend>
```

6. Edit the package's `CMakeLists.txt` to add rules to generate the new service files:

   1. Uncomment/edit the following `CMakeLists.txt` rule to reference the `LocalizePart` service we defined earlier:

```
## Generate services in the 'srv' folder
add_service_files(
   FILES
   LocalizePart.srv
)
```

   2. Uncomment/edit the following `CMakeLists.txt` rule to enable generation of messages and services:

```
## Generate added messages and services with any dependencies listed here
generate_messages(
   DEPENDENCIES
   geometry_msgs
)
```

7. NOW! you have a service defined in you package and you can attempt to *Build* the code to generate the service:

```
catkin build
```

Note: (or use Qt!)

Service Server

1. Edit `vision_node.cpp`; remember that the ros wiki is a resource.
2. Add the header for the service we just created

```
#include <myworkcell_core/LocalizePart.h>
```

3. Add a member variable (type: `ServiceServer`, name: `server_`), near the other `Localizer` class member variables:

```
ros::ServiceServer server_;
```

4. In the `Localizer` class constructor, advertise your service to the ROS master:

```
server_ = nh.advertiseService("localize_part", &Localizer::localizePart, this);
```

5. The `advertiseService` command above referenced a service callback named `localizePart`. Create an empty boolean function with this name in the `Localizer` class. Remember that your request and response types were defined in the `LocalizePart.srv` file. The arguments to the boolean function are the request and response type, with the general structure of `Package::ServiceName::Request` or `Package::ServiceName::Response`.

```
bool localizePart(myworkcell_core::LocalizePart::Request& req,
                  myworkcell_core::LocalizePart::Response& res)
```

(continues on next page)

(continued from previous page)

```
{

}
```

6. Now add code to the `localizePart` callback function to fill in the Service Response. Eventually, this callback will transform the pose received from the `fake_ar_publisher` (in `visionCallback`) into the frame specifed in the Service Request. For now, we will skip the frame-transform, and just pass through the data received from `fake_ar_publisher`. Copy the pose measurement received from `fake_ar_publisher` (saved to `last_msg_`) directly to the Service Response.

```
bool localizePart(myworkcell_core::LocalizePart::Request& req,
                  myworkcell_core::LocalizePart::Response& res)
{
  // Read last message
  fake_ar_publisher::ARMarkerConstPtr p = last_msg_;
  if (!p) return false;

  res.pose = p->pose.pose;
  return true;
}
```

7. You should comment out the `ROS_INFO_STREAM` call in your `visionCallback` function, to avoid cluttering the screen with useless info.

8. Build the updated `vision_node`, to make sure there are no compile errors.

Service Client

1. Create a new node (inside the same `myworkcell_core` package), named `myworkcell_node.cpp`. This will eventually be our main "application node", that controls the sequence of actions in our Scan & Plan task. The first action we'll implement is to request the position of the AR target from the Vision Node's `LocalizePart` service we created above.

2. Be sure to include the standard ros header as well as the header for the `LocalizePart` service:

```
#include <ros/ros.h>
#include <myworkcell_core/LocalizePart.h>
```

3. Create a standard C++ main function, with typical ROS node initialization:

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "myworkcell_node");
  ros::NodeHandle nh;

  ROS_INFO("ScanNPlan node has been initialized");

  ros::spin();
}
```

4. We will be using a cpp class "ScanNPlan" to contain most functionality of the myworkcell_node. Create a skeleton structure of this class, with an empty constructor and a private area for some internal/private variables.

```
class ScanNPlan
{
```

(continues on next page)

(continued from previous page)

```
public:
  ScanNPlan(ros::NodeHandle& nh)
  {

  }

private:
  // Planning components

};
```

5. Within your new ScanNPlan class, define a ROS ServiceClient as a private member variable of the class. Initialize the ServiceClient in the ScanNPlan constructor, using the same service name as defined earlier ("localize_part"). Create a void function within the ScanNPlan class named `start`, with no arguments. This will contain most of our application workflow. For now, this function will call the `LocalizePart` service and print the response.

```
class ScanNPlan
{
public:
  ScanNPlan(ros::NodeHandle& nh)
  {
    vision_client_ = nh.serviceClient<myworkcell_core::LocalizePart>("localize_
↪part");
  }

  void start()
  {
    ROS_INFO("Attempting to localize part");
    // Localize the part
    myworkcell_core::LocalizePart srv;
    if (!vision_client_.call(srv))
    {
      ROS_ERROR("Could not localize part");
      return;
    }
    ROS_INFO_STREAM("part localized: " << srv.response);
  }

private:
  // Planning components
  ros::ServiceClient vision_client_;
};
```

6. Now back in `myworkcell_node`'s `main` function, instantiate an object of the `ScanNPlan` class and call the object's `start` function.

```
ScanNPlan app(nh);

ros::Duration(.5).sleep();  // wait for the class to initialize
app.start();
```

7. Edit the package's `CMakeLists.txt` to build the new node (executable), with its associated dependencies. Add the following rules to the appropriate sections, directly under the matching rules for `vision_node`:

```
add_executable(myworkcell_node src/myworkcell_node.cpp)

add_dependencies(myworkcell_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_
↪EXPORTED_TARGETS})

target_link_libraries(myworkcell_node ${catkin_LIBRARIES})
```

8. Build the nodes to check for any compile-time errors:

```
catkin build
```

Note: (or use Qt!)

Use New Service

1. Enter each of these commands in their own terminal:

```
roscore
rosrun fake_ar_publisher fake_ar_publisher_node
rosrun myworkcell_core vision_node
rosrun myworkcell_core myworkcell_node
```

3.2.2 Actions

This Exercise is not part of the standard ROS-I Training Class workflow. Follow the standard ROS tutorials (linked below), for practice using ROS **Actions**.

ROS Tutorials for C++ Action Client/Server usage

- SimpleActionServer
- SimpleActionClient

3.2.3 Launch Files

In this exercise, we will explore starting groups of nodes at once with launch files.

Motivation

The ROS architecture encourages engineers to use ''nodes" as a fundamental unit of organization in their systems, and applications can quickly grow to require many nodes to operate. Opening a new terminal and running each node individually quickly becomes unfeasible. It'd be nice to have a tool to bring up groups of nodes at once. ROS ''launch" files are one such tool. It even handles bringing `roscore` up and down for you.

Reference Example

Roslaunch Examples

Further Information and Resources

Roslaunch XML Specification

Debugging and Launch Files

Scan-N-Plan Application: Problem Statement

In this exercise, you will:

1. Create a new package, `myworkcell_support`.
2. Create a directory in this package called `launch`.
3. Create a file inside this directory called `workcell.launch` that:
    1. Launches `fake_ar_publisher`
    2. Launches `vision_node`

You may also choose to launch `myworkcell_core` node with the others or keep it separate. We often configure systems with two main launch files. In this example, `fake_ar_publisher` and `vision_node` are "environment nodes", while `myworkcell_node` is an "application" node.

1. "Environment" Launch File - driver/planning nodes, config data, etc.
2. "Application" Launch File - executes a sequence of actions for a particular application.

Scan-N-Plan Application: Guidance

1. In your workspace, create the new package `myworkcell_support` with a dependency on `myworkcell_core`. Rebuild and source the workspace so that ROS can find the new package:

```
cd ~/catkin_ws/src
catkin create pkg myworkcell_support --catkin-deps myworkcell_core
catkin build
source ~/catkin_ws/devel/setup.bash
```

2. Create a directory for launch files (inside the new `myworkcell_support` package):

```
roscd myworkcell_support
mkdir launch
```

3. Create a new file, `workcell.launch` (inside the `launch` directory) with the following XML skeleton:

```
<launch>

</launch>
```

4. Insert lines to bring up the nodes outlined in the problem statement. See the reference documentation for more information:

```
<node name="fake_ar_publisher" pkg="fake_ar_publisher" type="fake_ar_publisher_
↪node" />
<node name="vision_node" pkg="myworkcell_core" type="vision_node" />
```

    - *Remember: All launch-file content must be **between** the `<launch> ... </launch>` tag pair.*

5. Test the launch file:

```
roslaunch myworkcell_support workcell.launch
```

Note: roscore and both nodes were automatically started. Press Ctrl+C *to close all nodes started by the launch file. If no nodes are left running, roscore is also stopped.*

6. Notice that none of the usual messages were printed to the console window. Launch files will suppress console output below the **ERROR** severity level by default. To restore normal text output, add the `output="screen"` attribute to each of the nodes in your launch files:

```
<node name="fake_ar_publisher" pkg="fake_ar_publisher" type="fake_ar_publisher_
↪node" output="screen"/>
<node name="vision_node" pkg="myworkcell_core" type="vision_node" output="screen"␣
↪/>
```

3.2.4 Parameters

In this exercise, we will look at ROS Parameters for configuring nodes||

Motivation

By this point in these tutorials (or your career), you've probably typed the words `int main(int argc, char** argv)` more times than you can count. The arguments to `main` are the means by which a system outside scope and understanding of your program can configure your program to do a particular task. These are *command line parameters*.

The ROS ecosystem has an analogous system for configuring entire groups of nodes. It's a fancy key-value storage program that gets brought up as part of `roscore`. It's best used to pass configuration parameters to nodes individually (e.g. to identify which camera a node should subscribe to), but it can be used for much more complicated items.

Reference Example

Understanding Parameters

Further Information and Resource

Roscpp tutorial

Private Parameters

Parameter Server

Scan-N-Plan Application: Problem Statement

In previous exercises, we added a service with the following definition:

```
# request
string base_frame
---
# response
geometry_msgs/Pose pose
```

So far we haven't used the request field, `base_frame`, for anything. In this exercise we'll use ROS parameters to set this field. You will need to:

1. Add a private node handle to the main method of the `myworkcell_node` in addition to the normal one.
2. Use the private node handle to load the parameter `base_frame` and store it in a local string object.
   - If no parameter is provided, default to the parameter to `"world"`.
3. When making the service call to the `vision_node`, use this parameter to fill out the `request::base_frame` field.
4. Add a `<param>` tag to your launch file to initialize the new value.

Scan-N-Plan Application: Guidance

1. Open up `myworkcell_node.cpp` for editing.
2. Add a new `ros::NodeHandle` object to the `main` function, and make it private through its parameters. For more guidance, see the ros wiki on this subject.

```
ros::NodeHandle private_node_handle ("~");
```

3. Create a temporary string object, `std::string base_frame;`, and then use the private node handle's API to load the parameter `"base_frame"`.

```
std::string base_frame;
private_node_handle.param<std::string>("base_frame", base_frame, "world"); //␣
↪parameter name, string object reference, default value
```

   - *`base_frame` parameter should be read after the `private_node_handle` is declared, but before `app.start()` is called*

4. Add a parameter to your `myworkcell_node` "start" function that accepts the base_frame argument, and assign the value from the parameter into the service request. Make sure to update the `app.start` call in your `main()` routine to pass through the `base_frame` value you read from the parameter server:

```
void start(const std::string& base_frame)
{
  ...
  srv.request.base_frame = base_frame;
  ROS_INFO_STREAM("Requesting pose in base frame: " << base_frame);
  ...
}

int main(...)
{
  ...
  app.start(base_frame);
  ...
}
```

   - *`srv.request` should be set **before** passing it into the service call (`vision_client.call(srv)`)*

5. Now we'll add `myworkcell_node` to the existing `workcell.launch` file, so we can set the `base_frame` parameter from a launch file. We'd like the `vision_node` to return the position of the target relative to the world frame, for motion-planning purposes. Even though that's the default value, we'll specify it in the launch-file anyway:

```
<node name="myworkcell_node" pkg="myworkcell_core" type="myworkcell_node" output=
↪"screen">
  <param name="base_frame" value="world"/>
</node>
```

6. Try it out by running the system.

```
catkin build
roslaunch myworkcell_support workcell.launch
```

- Press *Ctrl+C* to kill the running nodes
- Edit the launch file to change the base_frame parameter value (e.g. to "test2")
- Re-launch workcell.launch, and observe that the "request frame" has changed
  - The response frame doesn't change, because we haven't updated vision_node (yet) to handle the request frame. Vision_node always returns the same frame (for now).
- Set the base_frame back to "world"

3.3 Session 3 - Motion Control of Manipulators

`Slides`

3.3.1 Introduction to URDF

In this exercise, we will explore how to describe a robot in the URDF format.

Motivation

Many of the coolest and most useful capabilities of ROS and its community involve things like collision checking and dynamic path planning. It's frequently useful to have a code-independent, human-readable way to describe the geometry of robots and their cells. Think of it like a textual CAD description: "part-one is 1 meter left of part-two and has the following triangle-mesh for display purposes." The Unified Robot Description Format (URDF) is the most popular of these formats today. This module will walk you through creating a simple robot cell that we'll expand upon and use for practical purposes later.

Reference Example

Building a Visual Robot Model with URDF from Scratch

Further Information and Resources

- XML Specification
- ROS Tutorials
- XACRO Extensions
- SolidWorks to URDF Exporter

Scan-N-Plan Application: Problem Statement

We have the software skeleton of our Scan-N-Plan application, so let's take the next step and add some physical context. The geometry we describe in this exercise will be used to:

1. Perform collision checking
2. Understand robot kinematics
3. Perform transformation math Your goal is to describe a workcell that features:
4. An origin frame called `world`
5. A separate frame with "table" geometry (a flat rectangular prism)
6. A frame (geometry optional) called `camera_frame` that is oriented such that its Z axis is flipped relative to the Z axis of `world`

Scan-N-Plan Application: Guidance

*Note: If you have not completed the previous tutorials, copy myworkcell_core and myworkcell_support packages from ~/industrial-training/exercises/2.3/src and git clone https://github.com/jmeyer1292/fake_ar_publisher.git into your current workspace src folder.

1. It's customary to put describing files that aren't code into their own "support" package. URDFs typically go into their own subfolder ''urdf/". See the abb_irb2400_support package. Add a `urdf` sub-folder to your application support package.
2. Create a new `workcell.urdf` file inside the `myworkcell_support/urdf/` folder and insert the following XML skeleton:

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

3. Add the required links. See the irb2400_macro.xacro example from an ABB2400. Remember that all URDF tags must be placed **between** the `<robot> ... </robot>` tags.
    1. Add the `world` frame as a "virtual link" (no geometry).

    ```
    <link name="world"/>
    ```

    2. Add the `table` frame, and be sure to specify both collision & visual geometry tags. See the `box` type in the XML specification.

    ```
    <link name="table">
      <visual>
        <geometry>
          <box size="1.0 1.0 0.05"/>
        </geometry>
      </visual>
      <collision>
        <geometry>
          <box size="1.0 1.0 0.05"/>
        </geometry>
      </collision>
    </link>
    ```

    3. Add the `camera_frame` frame as another virtual link (no geometry).

```
<link name="camera_frame"/>
```

4. Connect your links with a pair of fixed joints Use an `rpy` tag in the `world_to_camera` joint to set its orientation as described in the introduction.

```
<joint name="world_to_table" type="fixed">
  <parent link="world"/>
  <child link="table"/>
  <origin xyz="0 0 0.5" rpy="0 0 0"/>
</joint>

<joint name="world_to_camera" type="fixed">
  <parent link="world"/>
  <child link="camera_frame"/>
  <origin xyz="-0.25 -0.5 1.25" rpy="0 3.14159 0"/>
</joint>
```

5. It helps to visualize your URDF as you add links, to verify things look as expected:

```
roslaunch urdf_tutorial display.launch model:=`rospack find myworkcell_
↪support`/urdf/workcell.urdf
```

If nothing shows up in Rviz, you may need to change the base frame in RVIZ (left panel at top) to the name of one of the links in your model.

3.3.2 Workcell XACRO

In this exercise, we will create an XACRO file representing a simple robot workcell. This will demonstrate both URDF and XACRO elements.

Motivation

Writing URDFs that involve more than just a few elements can quickly become a pain. Your file gets huge and duplicate items in your workspace means copy-pasting a bunch of links and joints while having to change their names just slightly. It's really easy to make a mistake that may (or may not) be caught at startup. It'd be nice if we could take some guidance from programming languages themselves: define a component once, then re-use it anywhere without excessive duplication. Functions and classes do that for programs, and XACRO macros do that for URDFs. XACRO has other cool features too, like a file include system (think #include), constant variables, math expression evaluation (e.g., say PI/2.0 instead of 1.57), and more.

Reference Example

Cleaning Up URDF with XACRO Tutorial

Further Information and Resources

Xacro Extension Documentation

Creating a URDF for an Industrial Robot

Scan-N-Plan Application: Problem Statement

In the previous exercise we created a workcell consisting of only static geometry. In this exercise, we'll add a UR5 robot *assembly* using XACRO tools.

Specifically, you will need to:

1. Convert the `*.urdf` file you created in the previous sample into a XACRO file with the `xacro` extension.
2. Include a file containing the xacro-macro definition of a `UR5`
3. Instantiate a `UR5` in your workspace and connect it to the *table* link.

Scan-N-Plan Application: Guidance

1. Rename the `workcell.urdf` file from the previous exercise to `workcell.xacro`
2. Bring in the `ur_description` package into your ROS environment. You have a few options:
    1. You can install the debian packages.

    ```
    sudo apt install ros-melodic-ur-description ros-melodic-ur-kinematics
    ```

    Note: these may or may not exist for the current distribution of ROS. If this is the case, you will need to download them from source (see next step)

    1. You can clone it from GitHub to your catkin workspace:

    ```
    cd ~/catkin_ws/src
    git clone https://github.com/ros-industrial/universal_robot.git
    catkin build
    source ~/catkin_ws/devel/setup.bash
    ```

    It's not uncommon for description packages to put each "module", "part", or "assembly" into its own file. In many cases, a package will also define extra files that define a complete cell with the given part so that we can easily visually inspect the result. The UR package defines such a file for the UR5 (ur5_robot.urdf.xacro): It's a great example for this module.

3. Locate the xacro file that implements the UR5 macro and include it in your newly renamed `workcell.xacro` file. Add this include line near the top of your `workcell.xacro` file, beneath the `<robot>` tag:

```
<xacro:include filename="$(find ur_description)/urdf/ur5.urdf.xacro" />
```

    If you explore the UR5 definition file, or just about any other file that defines a Xacro macro, you'll find a lot of uses of `${prefix}` in element names. Xacro evaluates anything inside a "${}" at run-time. It can do basic math, and it can look up variables that come to it via properties (ala-global variables) or macro parameters. Most macros will take a "prefix" parameter to allow a user to create multiple instances of said macro. It's the mechanism by which we can make the eventual URDF element names unique, otherwise we'd get duplicate link names and URDF would complain.

4. Including the `ur5.urdf.xacro` file does not actually create a UR5 robot in our URDF model. It defines a macro, but we still need to call the macro to create the robot links and joints. *Note the use of the `prefix` tag, as discussed above.*

```
<xacro:ur5_robot prefix="" joint_limited="true"/>
```

    Macros in Xacro are just fancy wrappers around copy-paste. You make a macro and it gets turned into a chunk of links and joints. You still have to connect the rest of your world to that macro's results. This means you have to look at the macro and see what the base link is and what the end link

is. Hopefully your macro follows a standard, like the ROS-Industrial one, that says that base links are named "base_link" and the last link is called "tool0".

5. Connect the UR5 `base_link` to your existing static geometry with a fixed link.

```
<joint name="table_to_robot" type="fixed">
  <parent link="table"/>
  <child link="base_link"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
</joint>
```

6. Create a new `urdf.launch` file (in the `myworkcell_support` package) to load the URDF model and (optionally) display it in rviz:

```
<launch>
  <arg name="gui" default="true"/>
  <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find␣
↪myworkcell_support)/urdf/workcell.xacro'" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_
↪state_publisher"/>
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_
↪state_publisher">
    <param name="use_gui" value="$(arg gui)"/>
  </node>
  <node name="rviz" pkg="rviz" type="rviz" if="$(arg gui)"/>
</launch>
```

7. Check the updated URDF in RViz, using the launch file you just created:

   `roslaunch myworkcell_support urdf.launch`

   - Set the 'Fixed Frame' to 'world' and add the RobotModel and TF displays to the tree view on the left, to show the robot and some transforms.
   - Try moving the joint sliders to see the UR5 robot move.

3.3.3 Coordinate Tranforms using TF

In this exercise, we will explore the terminal and C++ commands used with TF, the transform library.

Motivation

It's hard to imagine a useful, physical "robot" that doesn't move itself or watch something else move. A useful application in ROS will inevitably have some component that needs to monitor the position of a part, robot link, or tool. In ROS, the "eco-system" and library that facilitates this is called TF. TF is a fundamental tool that allows for the lookup the transformation between any connected frames, even back through time. It allows you to ask questions like: "What was the transform between A and B 10 seconds ago." That's useful stuff.

Reference Example

ROS TF2 Listener Tutorial

Further Information and Resources

- Wiki Documentation

- TF2 Tutorials
- TF2 Buffer API

Scan-N-Plan Application: Problem Statement

The part pose information returned by our (simulated) camera is given in the optical reference frame of the camera itself. For the robot to do something with this data, we need to transform the data into the robot's reference frame.

Specifically, edit the service callback inside the vision_node to transform the last known part pose from `camera_frame` to the service call's `base_frame` request field.

Scan-N-Plan Application: Guidance

1. Specify `tf2_ros` and `tf2_geometry_msgs` as dependencies of your core package.
   - Edit `package.xml` (+2 lines) and `CMakeLists.txt` (+4 lines) as in previous exercises
2. Add `tf2_ros::Buffer` and `tf2_ros::TransformListener` objects to the vision node (as class members variables).

```
#include <tf2/buffer.h>
#include <tf2/transform_listener.h>
...
tf2_ros::Buffer buffer_;
tf2_ros::TransformListener listener_;
```

3. The transform listener must be constructed using the buffer. Initialize it in the class constructor:

```
class Localizer
{
public:
  Localizer(ros::NodeHandle& nh) : listener_(buffer_)
```

4. Add code to the existing `localizePart` method to convert the reported target pose from its reference frame ("camera_frame") to the service-request frame:
   1. Remove the placeholder line from a previous exercise that sets the resulting pose equal to the pose from the last message

```
- res.pose = p->pose.pose;
```

   2. To perform a coordinate transformation, we will use a *stamped pose* object which bundles a 3D pose with metadata about what coordinate system the pose is in, which is known as a *header*. These pieces of data both come from the message received by the marker publisher:

```
geometry_msgs::PoseStamped target_pose_from_cam;
target_pose_from_cam.header = p->header;
target_pose_from_cam.pose = p->pose.pose;
```

   3. Use the buffer object to transform this `PoseStamped` object to another coordinate frame, specified by the frame in the service request:

```
geometry_msgs::PoseStamped target_pose_from_req = buffer_.transform(
    target_pose_from_cam, req.base_frame);
```

- Note: The buffer looks up the transformation between the camera frame and the base frame at the specific time when the message was first generated, which is also recorded in the header of the message.
- There are many other *Stamped* versions messages besides `PoseStamped`. Most of them can also be transformed to different coordinate system using the same method.

1. Return the transformed pose in the service response.

```
res.pose = target_pose_from_req.pose;
```

5. Run the nodes to test the transforms:

```
catkin build
roslaunch myworkcell_support urdf.launch
roslaunch myworkcell_support workcell.launch
```

6. Change the "base_frame" parameter in `workcell.launch` (e.g. to "table"), relaunch the `workcell.launch` file, and note the different pose result. Change the "base_frame" parameter back to "world" when you're done.

3.3.4 Build a MoveIt! Package

In this exercise, we will create a MoveIt! package for an industrial robot. This package creates the configuration and launch files required to use a robot with the MoveIt! Motion-Control nodes. In general, the MoveIt! package does not contain any C++ code.

Motivation

MoveIt! is a free-space motion planning framework for ROS. It's an incredibly useful and easy-to-use tool for planning motions between two points in space without colliding with anything. Under the hood MoveIt is quite complicated, but unlike most ROS libraries, it has a really nice GUI Wizard to get you going.

Reference Example

Using MoveIt with ROS-I

Further Information and Resources

MoveIt's Standard Wizard Guide

Scan-N-Plan Application: Problem Statement

In this exercise, you will generate a MoveIt package for the UR5 workcell you built in a previous step. This process will mostly involve running the MoveIt! Setup Assistant. At the end of the exercise you should have the following:

1. A new package called `myworkcell_moveit_config`
2. A moveit configuration with one group ("manipulator"), that consists of the kinematic chain between the UR5's `base_link` and `tool0`.

Scan-N-Plan Application: Guidance

1. Start the MoveIt! Setup Assistant (don't forget auto-complete with tab):

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

2. Select "Create New MoveIt Configuration Package", select the `workcell.xacro` you created previously, then "Load File".
3. Work your way through the tabs on the left from the top down.
    1. Generate a self-collision matrix.
    2. Add a fixed virtual base joint. e.g.

```
name = 'FixedBase' (arbitrary)
child = 'world' (should match the URDF root link)
parent = 'world' (reference frame used for motion planning)
type = 'fixed'
```

    3. Add a planning group called `manipulator` that names the kinematic chain between `base_link` and `tool0`. Note: Follow ROS naming guidelines/requirements and don't use any whitespace, anywhere.

        a. Set the kinematics solver to `KDLKinematicsPlugin`

    4. Create a few named positions (e.g. "home", "allZeros", etc.) to test with motion-planning.
    5. Don't worry about adding end effectors/grippers or passive joints for this exercise.
    6. Enter author / maintainer info.

        Yes, it's required, but doesn't have to be valid

    7. Generate a new package and name it `myworkcell_moveit_config`.
        - make sure to create the package inside your `catkin_ws/src` directory

The outcome of these steps will be a new package that contains a large number of launch and configuration files. At this point, it's possible to do motion planning, but not to execute the plan on any robot. To try out your new configuration:

```
catkin build
source ~/catkin_ws/devel/setup.bash
roslaunch myworkcell_moveit_config demo.launch
```

Don't worry about learning how to use RViz to move the robot; that's what we'll cover in the next session!

Using MoveIt! with Physical Hardware

MoveIt!'s setup assistant generates a suite of files that, upon launch:

- Loads your workspace description to the parameter server.
- Starts a node `move_group` that offers a suite of ROS services & actions for doing kinematics, motion planning, and more.
- An internal simulator that publishes the last planned path on a loop for other tools (like RViz) to visualize.

Essentially, MoveIt can publish a ROS message that defines a trajectory (joint positions over time), but it doesn't know how to pass that trajectory to your hardware.

To do this, we need to define a few extra files.

1. Create a `controllers.yaml` file (`myworkcell_moveit_config/config/controllers.yaml`) with the following contents:

```
controller_list:
  - name: ""
    action_ns: joint_trajectory_action
    type: FollowJointTrajectory
    joints: [shoulder_pan_joint, shoulder_lift_joint, elbow_joint, wrist_1_joint,␣
↪wrist_2_joint, wrist_3_joint]
```

2. Create the `joint_names.yaml` file (`myworkcell_moveit_config/config/joint_names.yaml`):

```
controller_joint_names: [shoulder_pan_joint, shoulder_lift_joint, elbow_joint,␣
↪wrist_1_joint, wrist_2_joint, wrist_3_joint]
```

3. Fill in the existing, but blank, controller_manager launch file (`myworkcell_moveit_config/launch/myworkcell_moveit_controller_manager.launch.xml`):

```
<launch>
  <arg name="moveit_controller_manager"
       default="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
  <param name="moveit_controller_manager"
         value="$(arg moveit_controller_manager)"/>

  <rosparam file="$(find myworkcell_moveit_config)/config/controllers.yaml"/>
</launch>
```

4. Create a new `myworkcell_planning_execution.launch` (in `myworkcell_moveit_config/launch`):

```
<launch>
  <!-- The planning and execution components of MoveIt! configured to run -->
  <!-- using the ROS-Industrial interface. -->

  <!-- Non-standard joint names:
       - Create a file [robot_moveit_config]/config/joint_names.yaml
           controller_joint_names: [joint_1, joint_2, ... joint_N]
       - Update with joint names for your robot (in order expected by rbt␣
↪controller)
       - and uncomment the following line: -->
  <rosparam command="load" file="$(find myworkcell_moveit_config)/config/joint_
↪names.yaml"/>

  <!-- the "sim" argument controls whether we connect to a Simulated or Real␣
↪robot -->
  <!--  - if sim=false, a robot_ip argument is required -->
  <arg name="sim" default="true" />
  <arg name="robot_ip" unless="$(arg sim)" />

  <!-- load the robot_description parameter before launching ROS-I nodes -->
  <include file="$(find myworkcell_moveit_config)/launch/planning_context.launch"␣
↪>
    <arg name="load_robot_description" value="true" />
  </include>

  <!-- run the robot simulator and action interface nodes -->
  <group if="$(arg sim)">
```

(continues on next page)

(continued from previous page)

```
    <include file="$(find industrial_robot_simulator)/launch/robot_interface_
↪simulator.launch" />

    <!-- publish the robot state (tf transforms) -->
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_
↪state_publisher" />
  </group>

  <!-- run the "real robot" interface nodes -->
  <!--   - this typically includes: robot_state, motion_interface, and joint_
↪trajectory_action nodes -->
  <!--   - replace these calls with appropriate robot-specific calls or launch␣
↪files -->
  <group unless="$(arg sim)">
    <remap from="follow_joint_trajectory" to="joint_trajectory_action"/>
    <include file="$(find ur_modern_driver)/launch/ur_common.launch" >
      <arg name="robot_ip" value="$(arg robot_ip)"/>
      <arg name="min_payload" value="0.0"/>
      <arg name="max_payload" value="5.0"/>
    </include>
  </group>

  <include file="$(find myworkcell_moveit_config)/launch/move_group.launch">
    <arg name="publish_monitored_planning_scene" value="true" />
  </include>

  <include file="$(find myworkcell_moveit_config)/launch/moveit_rviz.launch">
    <arg name="config" value="true"/>
  </include>

</launch>
```

5. Now let's test the new launch files we created:

```
roslaunch myworkcell_moveit_config myworkcell_planning_execution.launch
```

3.3.5 Motion Planning using RViz

In this exercise, we will (finally) learn how to use the MoveIt! RViz plugin to plan and execute motion on a simulated robot. We will explore the different options and constraints associated with both MoveIt! and the RViz plugin.

Launch the Planning Environment

1. Source your catkin workspace.
2. Bring up the planning environment, connected to a ROS-I Simulator node:

```
roslaunch myworkcell_moveit_config myworkcell_planning_execution.launch
```

Plugin Display Options

1. Find and test the following display options in the Displays panel, *Motion Planning* display

- *Scene Robot -> Show Robot Visual*
- *Scene Robot -> Show Robot Collision*
- *Planning Request -> Query Start State*
- *Planning Request -> Query Goal State*

2. For now, enable display of the *Show Robot Visual* and *Query Goal State*, leaving *Show Robot Collision* and *Query Start State* disabled
3. Select the `Panel -> Motion Planning - Trajectory Slider` menu option to display a trajectory-preview slider.
    - *this slider allows for detailed review of the last planned trajectory*

Basic Motion

1. In the *Motion Planning* panel, select the *Planning* tab
2. Under the *Query* section, expand the *Select Goal State* section
    - *select <random valid> and press Update*
    - *observe the goal position in the graphics window*
3. Click *Plan* to see the robot motion generated by the MoveIt! planning libraries
    - *deselect Displays -> Motion Planning -> Planned Path -> Loop Animation to stop the cyclic display*
    - *select Displays -> Motion Planning -> Planned Path -> Show Trail to show the swept path*
4. Click *Execute* to run the motion on the Industrial Robot Simulator
    - *observe that the multi-colored scene robot display updates to show that the robot has "moved" to the goal position*
5. Repeat steps 2-5 a few more times
    - *try using the interactive marker to manually move the robot to a desired position*
    - *try using a named pose (e.g. "straight up")*

Beyond the Basics

1. Experiment with different Planning Algorithms
    - *select Context tab, choose a Planning Algorithm (drop-down box next to "OMPL")*
    - *the RRTkConfigDefault algorithm is often much faster*
2. Environment Obstacles
    - Adjust the Goal State to move the robot into collision with an obstacle (e.g. the table)
        - note the colliding links are colored red
        - since the position is unreachable, you can see the robot search through different positions as it tries to find a solution
        - try disabling the Use Collision-Aware IK setting on the Planning tab
        - see that the collisions are still detected, but the solver no longer searches for a collision-free solution
    - Try to plan a path through the obstacle

  - It may help to have "Collision-Aware IK" disabled when moving the Goal State
  - If the robot fails to plan, check the error log and try repeating the plan request
  - Because the default planners are sampling-based, they may produce different results on each execution
  - You can also try increasing the planning time to allow a successful plan to be created
  - Try different planning algorithms in this, more complex, planning task
- Try adding a new obstacle to the scene:
  - Under the Scene Objects tab, add the `I-Beam.dae` CAD model
    - This file is located in the *industrial_training* repo: `~/industrial_training/exercises/3.4/I-Beam.dae`
  - Move the I-Beam into an interesting position, using the manipulation handles
  - Press Publish Scene, to push the updated position to MoveIt
  - Try to plan around the obstacle

3.4 Session 4 - Descartes and Perception

`Slides`

3.4.1 Motion Planning using C++

In this exercise, we'll explore MoveIt's C++ interface to programatically move a robot.

Motivation

Now that we've got a working MoveIt! configuration for your workcell and we've played a bit in RViz with the planning tools, let's perform planning and motion in code. This exercise will introduce you to the basic C++ interface for interacting with the MoveIt! node in your own program. There are lots of ways to use MoveIt!, but for simple applications this is the most straight forward.

Reference Example

Move Group Interface tutorial

3. Further Information and Resources

- MoveIt! Tutorials
- MoveIt! home-page

Scan-N-Plan Application: Problem Statement

In this exercise, your goal is to modify the `myworkcell_core` node to:

1. Move the robot's tool frame to the center of the part location as reported by the service call to your vision node.

Scan-N-Plan Application: Guidance

1. Edit your `myworkcell_node.cpp` file.
    1. Add `#include <tf/tf.h>` to allow access to the tf library (for frame transforms/utilities).
        - Remember that we already added a dependency on the `tf` package in a previous exercise.
    2. In the `ScanNPlan` class's `start` method, use the response from the `LocalizePart` service to initialize a new `move_target` variable:

```
geometry_msgs::Pose move_target = srv.response.pose;
```

        - make sure to place this code *after* the call to the vision_node's service.
2. Use the `MoveGroupInterface` to plan/execute a move to the `move_target` position:
    1. In order to use the `MoveGroupInterface` class it is necessary to add the `moveit_ros_planning_interface` package as a dependency of your `myworkcell_core` package. Add the `moveit_ros_planning_interface` dependency by modifying your package's `CMakeLists.txt` (2 lines) and `package.xml` (1 line) as in previous exercises.
    2. Add the appropriate "include" reference to allow use of the `MoveGroupInterface`:

```
#include <moveit/move_group_interface/move_group_interface.h>
```

    3. Create a `moveit::planning_interface::MoveGroupInterface` object in the `ScanNPlan` class's `start()` method. It has a single constructor that takes the name of the planning group you defined when creating the workcell moveit package ("manipulator").

```
moveit::planning_interface::MoveGroupInterface move_group("manipulator");
```

    4. Set the desired cartesian target position using the `move_group` object's `setPoseTarget` function. Call the object's `move()` function to plan and execute a move to the target position.

```
// Plan for robot to move to part
move_group.setPoseReferenceFrame(base_frame);
move_group.setPoseTarget(move_target);
move_group.move();
```

    5. As described here, the `move_group.move()` command requires use of an "asynchronous" spinner, to allow processing of ROS messages during the blocking `move()` command. Initialize the spinner near the start of the `main()` routine after `ros::init(argc, argv, "myworkcell_node")`, and **replace** the existing `ros::spin()` command with `ros::waitForShutdown()`, as shown:

```
ros::AsyncSpinner async_spinner(1);
async_spinner.start();
...
ros::waitForShutdown();
```

3. Test the system!

```
catkin build
roslaunch myworkcell_moveit_config myworkcell_planning_execution.launch
roslaunch myworkcell_support workcell.launch
```

4. More to explore...
    - In RViz, add a "Marker" display of topic "/ar_pose_visual" to confirm that the final robot position matches the position published by `fake_ar_publisher`

- Try repeating the motion planning sequence:
  1. Use the MoveIt rviz interface to move the arm back to the "allZeros" position
  2. Ctrl+C the `workcell.launch` file, then rerun
- Try updating the `workcell_node`'s `start` method to automatically move back to the `allZeros` position after moving to the AR_target position. See here for a list of `move_group`'s available methods.
- Try moving to an "approach position" located a few inches away from the target position, prior to the final move-to-target.

3.4.2 Introduction to Descartes Path Planning

In this exercise, we will use what was learned in the previous exercises by creating a Descartes planner to create a robot path.

Motivation

MoveIt! is a framework meant primarily for performing "free-space" motion where the objective is to move a robot from point A to point B and you don't particularly care about how that gets done. These types of problems are only a subset of frequently performed tasks. Imagine any manufacturing ''process" like welding or painting. You very much care about where that tool is pointing the entire time the robot is at work.

This tutorial introduces you to Descartes, a ''Cartesian" motion planner meant for moving a robot along some process path. It's only one of a number of ways to solve this kind of problem, but it's got some neat properties:

- It's deterministic and globally optimum (to a certain search resolution).
- It can search redundant degrees of freedom in your problem (say you have 7 robot joints or you have a process where the tool's Z-axis rotation doesn't matter).

Reference Example

Descartes Tutorial

Further Information and Resources

Descartes Wiki

APIs:

- descartes_core::PathPlannerBase
- descartes_planner::DensePlanner
- descartes_planner::SparsePlanner

Scan-N-Plan Application: Problem Statement

In this exercise, you will add a new node to your Scan-N-Plan application, based on a reference template, that:

1. Takes the nominal pose of the marker as input through a ROS service.
2. Produces a joint trajectory that commands the robot to trace the perimeter of the marker (as if it is dispensing adhesive).

Scan-N-Plan Application: Guidance

In the interest of time, we've included a file, `descartes_node.cpp`, that:

1. Defines a new node & accompanying class for our Cartesian path planning.
2. Defines the actual service and initializes the Descartes library.
3. Provides the high level work flow (see planPath function).

Left to you are the details of:

1. Defining a series of Cartesian poses that comprise a robot "path".
2. Translating those paths into something Descartes can understand.

Setup workspace

1. Clone the Descartes repository into your workspace src/ directory.

```
cd ~/catkin_ws/src
git clone -b melodic-devel https://github.com/ros-industrial-consortium/descartes.
↪git
```

2. Copy over the `ur5_demo_descartes` package into your workspace src/ directory.

```
cp -r ~/industrial_training/exercises/4.1/src/ur5_demo_descartes .
```

3. Copy over the `descartes_node_unfinished.cpp` into your core package's src/ folder and rename it `descartes_node.cpp`.

```
cp ~/industrial_training/exercises/4.1/src/descartes_node_unfinished.cpp␣
↪myworkcell_core/src/descartes_node.cpp
```

4. Add dependencies for the following packages in the `CMakeLists.txt` & `package.xml` files, as in previous exercises.
   - `ur5_demo_descartes`
   - `descartes_trajectory`
   - `descartes_planner`
   - `descartes_utilities`
   - `eigen_conversions`
5. Create rules in the `myworkcell_core` package's `CMakeLists.txt` to build a new node called `descartes_node`. As in previous exercises, add these lines near similar lines in the template file (not as a block as shown below).

```
add_executable(descartes_node src/descartes_node.cpp)
add_dependencies(descartes_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_
↪EXPORTED_TARGETS})
target_link_libraries(descartes_node ${catkin_LIBRARIES})
```

Complete Descartes Node

We will create a Service interface to execute the Descartes planning algorithm.

1. Define a new service named `PlanCartesianPath.srv` in the `myworkcell_core` package's `srv/` directory. This service takes the central target position and computes a joint trajectory to trace the target edges.

```
# request
geometry_msgs/Pose pose

---

# response
trajectory_msgs/JointTrajectory trajectory
```

2. Add the newly-created service file to the `add_service_file()` rule in the package's `CMakeLists.txt`.
3. Since our new service references a message type from another package, we'll need to add that other package (`trajectory_msgs`) as a dependency in the `myworkcell_core` `CMakeLists.txt` (3 lines) and `package.xml` (1 line) files.
4. Review `descartes_node.cpp` to understand the code structure. In particular, the `planPath` method outlines the main sequence of steps.
5. Search for the TODO commands in the Descartes node file and expand on those areas:
    1. In `makeToolPoses`, generate the remaining 3 sides of a path tracing the outside of our "AR Marker" target part.
    2. In `makeDescartesTrajectory`, convert the path you created into a Descartes Trajectory, one point at a time.
        - *Don't forget to transform each nominal point by the specified reference pose:* `ref * point`
    3. In `makeToleranced CartesianPoint`, create a `new AxialSymmetricPt` from the given pose.
        - See here for more documentation on this point type
        - Allow the point to be symmetric about the Z-axis (`AxialSymmetricPt::Z_AXIS`), with an increment of 90 degrees (PI/2 radians)
6. Build the project, to make sure there are no errors in the new `descartes_node`

Update Workcell Node

With the Descartes node completed, we now want to invoke its logic by adding a new `ServiceClient` to the primary workcell node. The result of this service is a joint trajectory that we must then execute on the robot. This can be accomplished in many ways; here we will call the `JointTrajectoryAction` directly.

1. In `myworkcell_node.cpp`, add include statements for the following headers:

```
#include <actionlib/client/simple_action_client.h>
#include <control_msgs/FollowJointTrajectoryAction.h>
#include <myworkcell_core/PlanCartesianPath.h>
```

   You do not need to add new dependenies for these libraries/messages, because they are pulled in transitively from moveit.

2. In your ScanNPlan class, add new private member variables: a ServiceClient for the `PlanCartesianPath` service and an action client for `FollowJointTrajectoryAction`:

```
ros::ServiceClient cartesian_client_;
actionlib::SimpleActionClient<control_msgs::FollowJointTrajectoryAction> ac_;
```

3. Initialize these new objects in your constructor. Note that the action client has to be initialized in what is called the `initializer list`.

```
ScanNPlan(ros::NodeHandle& nh) : ac_("joint_trajectory_action", true)
{
  // ... code
  cartesian_client_ = nh.serviceClient<myworkcell_core::PlanCartesianPath>("plan_
↪path");
}
```

4. At the end of the `start()` function, create a new Cartesian service and make the service request:

```
// Plan cartesian path
myworkcell_core::PlanCartesianPath cartesian_srv;
cartesian_srv.request.pose = move_target;
if (!cartesian_client_.call(cartesian_srv))
{
  ROS_ERROR("Could not plan for path");
  return;
}
```

5. Continue adding the following lines, to execute that path by sending an action directly to the action server (bypassing MoveIt):

```
// Execute descartes-planned path directly (bypassing MoveIt)
ROS_INFO("Got cart path, executing");
control_msgs::FollowJointTrajectoryGoal goal;
goal.trajectory = cartesian_srv.response.trajectory;
ac_.sendGoal(goal);
ac_.waitForResult();
ROS_INFO("Done");
```

6. Build the project, to make sure there are no errors in the new `descartes_node`

Test Full Application

1. Create a new `setup.launch` file (in `workcell_support` package) that brings up everything except your workcell_node:

```
<include file="$(find myworkcell_moveit_config)/launch/myworkcell_planning_
↪execution.launch"/>
<node name="fake_ar_publisher" pkg="fake_ar_publisher" type="fake_ar_publisher_
↪node" />
<node name="vision_node" type="vision_node" pkg="myworkcell_core" output="screen"/
↪>
<node name="descartes_node" type="descartes_node" pkg="myworkcell_core" output=
↪"screen"/>
```

2. Run the new setup file, then your main workcell node:

```
roslaunch myworkcell_support setup.launch
rosrun myworkcell_core myworkcell_node
```

It's difficult to see what's happening with the rviz planning-loop always running. Disable this loop animation in rviz (Displays -> Planned Path -> Loop Animation), then rerun `myworkcell_node`.

Hints and Help

Hints:

- The path we define in `makeToolPoses()` is relative to some known reference point on the part you are working with. So a tool pose of (0, 0, 0) would be exactly at the reference point, and not at the origin of the world coordinate system.
- In `makeDescartesTrajectorty(...)` we need to convert the relative tool poses into world coordinates using the "ref" pose.
- In `makeTolerancedCartesianPoint(...)` consider the following documentation for specific implementations of common joint trajectory points:
  - http://docs.ros.org/indigo/api/descartes_trajectory/html/
- For additional help, review the completed reference code at `~/industrial_training/exercises/4.1/src`

3.4.3 Introduction to Perception

In this exercise, we will experiment with data generated from the Asus Xtion Pro (or Microsoft Kinect) sensor in order to become more familiar with processing 3D data. We will view its data stream and visualize the data in various ways under Rviz.

Point Cloud Data File

The start of most perception processing is ROS message data from a sensor. In this exercise, we'll be using 3D point cloud data from a common Kinect-style sensor.

1. First, publish the point cloud data as a ROS message to allow display in rviz.
   1. Start `roscore` running in a terminal.
   2. Create a new directory for this exercise:

```
mkdir ~/ex4.2
cd ~/ex4.2
cp ~/industrial_training/exercises/4.2/table.pcd .
```

   3. Publish pointcloud messages from the pre-recorded `table.pcd` point cloud data file:

```
cd ~/ex4.2
rosrun pcl_ros pcd_to_pointcloud table.pcd 0.1 _frame_id:=map cloud_pcd:=orig_
↪cloud_pcd
```

   4. Verify that the `orig_cloud_pcd` topic is being published: `rostopic list`

Display the point cloud in RViz

1. Start an RViz window, to display the results of point-cloud processing

```
rosrun rviz rviz
```

2. Add a **PointCloud2** display item and set the desired topic.
   1. Select **Add** at the bottom of the Displays panel

2. Select **PointCloud2**
3. Expand **PointCloud2** in the display tree, and select a topic from topic drop down.
    - *Hint: If you are using the point cloud file, the desired topic is* `/orig_cloud_pcd`.

Experiment with PCL

Next, we will experiment with various command line tool provided by PCL for processing point cloud data. There are over 140 command line tools available, but only a few will be used as part of this exercise. The intent is to get you familiar with the capabilities of PCL without writing any code, but these command line tools are a great place to start when writing your own. Although command line tools are helpful for testing various processing methods, most applications typically use the C++ libraries directly for "real" processing pipelines. The ROS-I Advanced training course explores these C++ PCL methods in more detail.

Each of the PCL commands below generates a new point cloud file (`.pcd`) with the result of the PCL processing command. Use either the `pcl_viewer` to view the results directly or the `pcd_to_pointcloud` command to publish the point cloud data as a ROS message for display in rviz. Feel free to stop the `pcd_to_pointcloud` command after reviewing the results in rviz.

Downsample the point cloud using the pcl_voxel_grid.

1. Downsample the original point cloud using a voxel grid with a grid size of (0.05,0.05,0.05). In a voxel grid, all points in a single grid cube are replaced with a single point at the center of the voxel. This is a common method to simplify overly complex/detailed sensor data, to speed up processing steps.

```
pcl_voxel_grid table.pcd table_downsampled.pcd -leaf 0.05,0.05,0.05
pcl_viewer table_downsampled.pcd
```

1. View the new point cloud in rviz.(optional)

```
rosrun pcl_ros pcd_to_pointcloud table_downsampled.pcd 0.1 _frame_id:=map cloud_
↪pcd:=table_downsampled
```

Note: For the PointCloud2 in rviz change the topic to */table_downsampled* to show the new data.

Extracting the table surface from point cloud using the pcl_sac_segmentation_plane.

1. Find the largest plane and extract points that belong to that plane (within a given threshold).

```
pcl_sac_segmentation_plane table_downsampled.pcd only_table.pcd -thresh 0.01
pcl_viewer only_table.pcd
```

View the new point cloud in rviz.(optional)

```
rosrun pcl_ros pcd_to_pointcloud only_table.pcd 0.1 _frame_id:=map cloud_pcd:=only_
↪table
```

Note: For the PointCloud2 in rviz change the topic to */only_table* to show the new data.

Extracting the largest cluster on the table from point cloud using the pcl_sac_segmentation_plane.

1. Extract the largest point-cluster not belonging to the table.

```
pcl_sac_segmentation_plane table.pcd object_on_table.pcd -thresh 0.01 -neg 1
pcl_viewer object_on_table.pcd
```

View the new point cloud in rviz.(optional)

```
rosrun pcl_ros pcd_to_pointcloud object_on_table.pcd 0.1 _frame_id:=map cloud_
↪pcd:=object_on_table
```

Note: For the PointCloud2 in rviz change the topic to */object_on_table* to show the new data.

Remove outliers from the cloud using the pcl_outlier_removal.

1. For this example, a statistical method will be used for removing outliers. This is useful to clean up noisy sensor data, removing false artifacts before further processing.

```
pcl_outlier_removal table.pcd table_outlier_removal.pcd -method statistical
pcl_viewer table_outlier_removal.pcd
```

1. View the new point cloud in rviz. (optional)

```
rosrun pcl_ros pcd_to_pointcloud table_outlier_removal.pcd 0.1 _frame_id:=map cloud_
↪pcd:=table_outlier_removal
```

Note: For the PointCloud2 in rviz change the topic to */table_outlier_removal* to show the new data.

Compute the normals for each point in the point cloud using the pcl_normal_estimation.

1. This example estimates the local surface normal (perpendicular) vectors at each point. For each point, the algorithm uses nearby points (within the specified radius) to fit a plane and calculate the normal vector. Zoom in to view the normal vectors in more detail.

```
pcl_normal_estimation only_table.pcd table_normals.pcd -radius 0.1
pcl_viewer table_normals.pcd -normals 10
```

Mesh a point cloud using the marching cubes reconstruction.

Point cloud data is often unstructured, but sometimes processing algorithms need to operate on a more structured surface mesh. This example uses the "marching cubes" algorithm to construct a surface mesh that approximates the point cloud data.

```
pcl_marching_cubes_reconstruction table_normals.pcd table_mesh.vtk -grid_res 20
pcl_viewer table_mesh.vtk
```

3.5 Application Demo 1 - Perception-Driven Manipulation

3.5.1 Application Demo 1 - Perception-Driven Manipulation

Perception-Driven Manipulation Introduction

Goal

The purpose of these exercises is to implement a ROS node that drives a robot through a series of moves and actions in order to complete a pick and place task. In addition, they will serve as an example of how to integrate a variety of software capabilities (perception, controller drivers, I/O, inverse kinematics, path planning, collision avoidance, etc) into a ROS-based industrial application.

Objectives

- Understand the components and structure of a real or simulated robot application.
- Learn how to command robot moves using Moveit!.
- Learn how to move the arm to a joint or Cartesian position.
- Leverage perception capabilities including AR tag recognition and PCL.
- Plan collision-free paths for a pick and place task.
- Control robot peripherals such as a gripper.

Inspect the `pick_and_place_exercise` Package

In this exercise, we will get familiar with all the files that you'll be interacting with throughout these exercises.

Acquire and initialize the Workspace

```
cp -r ~/industrial_training/exercises/Perception-Driven_Manipulation/\
template_ws ~/perception_driven_ws
cd ~/perception_driven_ws
source /opt/ros/melodic/setup.bash
catkin init
wstool update -t src
```

Download debian dependencies

Note: Make sure you have installed and configured the rosdep tool.

Then, run the following command from the `src` directory of your workspace:

```
rosdep install --from-paths . --ignore-src -y
```

Build your workspace

```
catkin build
```

Note: If the build fails then revisit the previous two steps to make sure all the dependencies were downloaded.

Source the workspace

Run the following command from your workspace parent directory

```
source devel/setup.bash
```

Locate and navigate into the package

```
cd ~/perception_driven_ws/src/collision_avoidance_pick_and_place/
```

Look into the `launch` directory

- `ur5_setup.launch`
  - Brings up the entire ROS system (MoveIt!, Rviz, perception nodes, ROS-I drivers, robot I/O peripherals)
- `ur5_pick_and_place.launch`
  - Runs your pick and place node.

Look into the `config` directory

- `ur5/`
  - `pick_and_place_parameters.yaml`
    - List of parameters read by the pick and place node.
  - `rviz_config.rviz`
    - Rviz configuration file for display properties.
  - `target_recognition_parameters.yaml`
    - Parameters used by the target recognition service for detecting the box from the sensor data.
  - `test_cloud_obstacle_descriptions.yaml`
    - Parameters used to generate simulated sensor data (simulated sensor mode only).
  - `collision_obstacles.txt`
    - Description of each obstacle blob added to the simulated sensor data (simulated sensor mode only)

Look into the `src` directory

- `nodes/`
  - `pick_and_place_node.cpp`
    - Main application thread. Contains all necessary headers and function calls.
- `tasks/`
  - `create_motion_plan.cpp`
  - `create_pick_moves.cpp`

  - `create_place_moves.cpp`
  - `detect_box_pick.cpp`
  - `pickup_box.cpp`
  - `place_box.cpp`
  - `move_to_wait_position.cpp`
  - `set_attached_object.cpp`
  - `set_gripper.cpp`

Note: The `tasks` directory contains source files with incomplete function definitions. You will fill with code where needed in order to complete the exercise.

- `utilities/`
  - `pick_and_place_utilities.cpp`
    - Contains support functions that will help you complete the exercise.

Package Setup

In this exercise, we'll build our package dependencies and configure the package for the Qt Creator IDE.

Build Package Dependencies

In a terminal enter:

```
cd ~/perception_driven_ws
catkin build
source devel/setup.bash
```

Import Package into QTCreator

In QTCreator select the following menu item: *File* → *New File or Project*.

In the dialog that appears, on the left select *Other Project* and in the middle section select *ROS Workspace*. Confirm your selection with the *Choose* button.

Open the Main Thread Source File

In the *Edit* tab, open the file `pick_and_place_node.cpp` in the directory `[workspace source directory]/collision_avoidance_pick_and_place/src/nodes`

Start in Simulation Mode

In this exercise, we will start a ROS system that is ready to move the robot in simulation mode.

Run setup launch file in simulation mode (simulated robot and sensor)

In a terminal enter:

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch
```

Rviz will display all the workcell components including the robot in its default position; at this point your system is ready. However, no motion will take place until we run the pick and place node.

Setup for real sensor and simulated robot

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch \
    sim_sensor:=false
```

Setup for real robot and simulated sensor data

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch \
    sim_robot:=false robot_ip:=[robot ip]
```

Setup for real robot and real sensor

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch \
    sim_robot:=false robot_ip:=[robot ip] sim_sensor:=false \
    sim_gripper:=false
```

Initialization and Global Variables

In this exercise, we will take a first look at the main application `pick_and_place_node.cpp`, its public variables, and how to properly initialize it as a ROS node.

Application Variables

Open the file `pick_and_place_utilities.h` in the following directory:

The C++ class `pick_and_place_config` defines public global variables used in various parts of the program. These variables are listed below:

```
ARM_GROUP_NAME  = "manipulator";
TCP_LINK_NAME   = "tcp_frame";
MARKER_TOPIC = "pick_and_place_marker";
PLANNING_SCENE_TOPIC = "planning_scene";
TARGET_RECOGNITION_SERVICE = "target_recognition";
MOTION_PLAN_SERVICE = "plan_kinematic_path";
WRIST_LINK_NAME = "ee_link";
ATTACHED_OBJECT_LINK_NAME = "attached_object_link";
WORLD_FRAME_ID  = "world_frame";
HOME_POSE_NAME  = "home";
WAIT_POSE_NAME  = "wait";
```

(continues on next page)

(continued from previous page)

```
AR_TAG_FRAME_ID    = "ar_frame";
GRASP_ACTION_NAME = "grasp_execution_action";
BOX_SIZE          = tf::Vector3(0.1f, 0.1f, 0.1f);
BOX_PLACE_TF      = tf::Transform(tf::Quaternion::getIdentity(), tf::Vector3(-0.8f,-0.
↪2f,BOX_SIZE.getZ()));
TOUCH_LINKS = std::vector<std::string>();
RETREAT_DISTANCE  = 0.05f;
APPROACH_DISTANCE = 0.05f;
```

In the main program `pick_and_place_node.cpp`, the global `application` object provides access to the program variables through its `cfg` member.

For instance, in order to use the `WORLD_FRAME_ID` global variable we would do the following:

```
ROS_INFO_STREAM("world frame: " << application.cfg.WORLD_FRAME_ID)
```

Node Initialization

In the `pick_and_place_node.cpp` file, the following block of code in the `main` function initializes the `PickAndPlace` application class and its main ROS and MoveIt! components.

```
int main(int argc,char** argv)
{
  geometry_msgs::Pose box_pose;
  std::vector<geometry_msgs::Pose> pick_poses, place_poses;

  /*␣
↪==========================================================================================*/
↪
  /*        INITIALIZING ROS NODE
      Goal:
      - Observe all steps needed to properly initialize a ros node.
      - Look into the 'cfg' member of PickAndPlace to take notice of the parameters␣
↪that
        are available for the rest of the program. */
  /*␣
↪==========================================================================================*/
↪

  // ros initialization
  ros::init(argc,argv,"pick_and_place_node");
  ros::NodeHandle nh;
  ros::AsyncSpinner spinner(2);
  spinner.start();

  // creating pick and place application instance
  PickAndPlace application;

  // reading parameters
  if(application.cfg.init())
  {
    ROS_INFO_STREAM("Parameters successfully read");
  }
  else
  {
```

(continues on next page)

(continued from previous page)

```
    ROS_ERROR_STREAM("Parameters not found");
    return 0;
  }

  // marker publisher
  application.marker_publisher = nh.advertise<visualization_msgs::Marker>(
                  application.cfg.MARKER_TOPIC,1);

  // planning scene publisher
  application.planning_scene_publisher = nh.advertise<moveit_msgs::PlanningScene>(
                  application.cfg.PLANNING_SCENE_TOPIC,1);

  // moveit interface
  application.move_group_ptr = MoveGroupPtr(
      new moveit::planning_interface::MoveGroupInterface(application.cfg.ARM_GROUP_
↪NAME));
  application.move_group_ptr->setPlannerId("RRTConnectkConfigDefault");

  // motion plan client
  application.motion_plan_client = nh.serviceClient<moveit_msgs::GetMotionPlan>
↪(application.cfg.MOTION_PLAN_SERVICE);

  // transform listener
  application.transform_listener_ptr = TransformListenerPtr(new␣
↪tf::TransformListener());

  // marker publisher (rviz visualization)
  application.marker_publisher = nh.advertise<visualization_msgs::Marker>(
                  application.cfg.MARKER_TOPIC,1);

  // target recognition client (perception)
  application.target_recognition_client = nh.serviceClient<collision_avoidance_pick_
↪and_place::GetTargetPose>(
                  application.cfg.TARGET_RECOGNITION_SERVICE);

  // grasp action client (vacuum gripper)
  application.grasp_action_client_ptr = GraspActionClientPtr(
                  new GraspActionClient(application.cfg.GRASP_ACTION_NAME,true));
```

Move Arm to Wait Position

The `MoveGroup` class in MoveIt! allows us to move the robot in various ways. With `MoveGroup` it is possible to move to a desired joint position, cartesian goal or a predefined pose created with the Setup Assistant. In this exercise, we will move the robot to a predefined joint pose.

Locate Function

- In the main program, locate the method call to `application.move_to_wait_position()`.
- Go to the source file of that function by clicking in any part of the function and pressing `F2` in QtCreator.
- Alternatively, browse to the file at

```
[workspace source directory]/collision_avoidance_pick_and_place/src/tasks/move_to_
↪wait_position.cpp
```

- Remove the first line containing the following `ROS_ERROR_STREAM ...` so that the program runs.

Complete Code

- Find every line that begins with the comment `Fill Code:` and read the description. Then, replace every instance of the comment `ENTER CODE HERE` with the appropriate line of code.

```
/* Fill Code:
    .
    .
    .
*/
/* ========  ENTER CODE HERE ======== */
```

- The name of the predefined "wait" pose was saved in the global variable `cfg.WAIT_POSE_NAME` during initialization.

Build Code and Run

- Compile the pick and place node:
  - In QTCreator: *Build* → *Build Project*
  - Alternatively, in a terminal:

```
catkin build collision_avoidance_pick_and_place
source ./devel/setup.bash
```

- Run the supporting nodes with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch
```

- In another terminal, run your node with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_pick_and_place.launch
```

- If the robot is not already in the wait position, it should move to the wait position. In the terminal, you will see something like the following message:

```
[ INFO] [1400553673.460328538]: Move wait Succeeded
[ERROR] [1400553673.460434627]: set_gripper is not implemented yet. Aborting.
```

API References

- setNamedTarget()
- move()

Open Gripper

In this exercise, the objective is to use a "grasp action client" to send a grasp goal that will open the gripper.

Locate Function

- In the main program, locate the function call to `application.set_gripper()`.
- Go to the source file of that function by clicking in any part of the function and pressing `F2` in QtCreator.
- Remove the first line containing the following `ROS_ERROR_STREAM ...` so that the program runs.

Complete Code

- Find every line that begins with the comment `Fill Code:` and read the description. Then, replace every instance of the comment `ENTER CODE HERE` with the appropriate line of code.

```
/* Fill Code:
    .
    .
    .
*/
/* ========  ENTER CODE HERE ======== */
```

- The `grasp_goal.goal` property can take on three possible values:

```
grasp_goal.goal = object_manipulation_msgs::GraspHandPostureExecutionGoal::GRASP;
grasp_goal.goal = object_manipulation_
↪msgs::GraspHandPostureExecutionGoal::RELEASE;
grasp_goal.goal = object_manipulation_msgs::GraspHandPostureExecutionGoal::PRE_
↪GRASP;
```

- Once the grasp flag has been set you can send the goal through the grasp action client

Build Code and Run

- Compile the pick and place node:
  - In QTCreator: *Build* → *Build Project*
  - Alternatively, in a terminal:

```
catkin build collision_avoidance_pick_and_place
```

- Run the supporting nodes with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch
```

- In another terminal, run your node with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_pick_and_place.launch
```

- If the task succeeds you will see something like the following in the terminal (below). The robot will not move, only gripper I/O is triggered:

```
[ INFO] [1400553290.464877904]: Move wait Succeeded
[ INFO] [1400553290.720864559]: Gripper opened
[ERROR] [1400553290.720985315]: detect_box_pick is not implemented yet.  Aborting.
```

API References

- sendGoal()

Detect Box Pick Point

The coordinate frame of the box's pick can be requested from a ROS service that detects it by processing the sensor data. In this exercise, we will learn how to use a service client to call that ROS service for the box pick pose.

Locate Function

- In the main program, locate the function call to `application.detect_box_pick()`.
- Go to the source file of that function by clicking in any part of the function and pressing `F2` in QtCreator.
- Remove the first line containing the following `ROS_ERROR_STREAM ...` so that the program runs.

Complete Code

- Find every line that begins with the comment `Fill Code:` and read the description. Then, replace every instance of the comment `ENTER CODE HERE` with the appropriate line of code.

```
/* Fill Code:
    .
    .
    .
*/
/* ========  ENTER CODE HERE ======== */
```

- The `target_recognition_client` object in your programs can use the `call()` method to send a request to a ROS service.
- The ROS service that receives the call will process the sensor data and return the pose for the box pick in the service structure member `srv.response.target_pose`.

Build Code and Run

- Compile the pick and place node:
  - In QTCreator: *Build* → *Build Project*
  - Alternatively, in a terminal:

```
catkin build collision_avoidance_pick_and_place
```

- Run the supporting nodes with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch
```

- In another terminal, run your node with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_pick_and_place.launch
```

- A blue box and voxel grid obstacles will be displayed in rviz. In the terminal you should see a message like the following:

```
[ INFO] [1400554224.057842127]: Move wait Succeeded
[ INFO] [1400554224.311158465]: Gripper opened
[ INFO] [1400554224.648747043]: target recognition succeeded
[ERROR] [1400554224.649055043]: create_pick_moves is not implemented yet. ␣
↪Aborting.
```

API References

- call()

Create Pick Moves

The gripper moves through three poses in order to do a pick: approach, target and retreat. In this exercise, we will use the box pick transform to create the pick poses for the TCP (Tool Center Point) coordinate frame and then transform them to the arm's wrist coordinate frame.

Locate Function

- In the main program, locate the function call to `application.create_pick_moves()`.
- Go to the source file of that function by clicking in any part of the function and pressing `F2` in QtCreator.
- Remove the first line containing the following `ROS_ERROR_STREAM ...` so that the program runs.

Complete Code

- Find every line that begins with the comment `Fill Code:` and read the description. Then, replace every instance of the comment `ENTER CODE HERE` with the appropriate line of code.

```
/* Fill Code:
    .
    .
    .
*/
/* ========  ENTER CODE HERE ======== */
```

- The `create_manipulation_poses()` uses the values of the approach and retreat distances in order to create the corresponding poses at the desired target.
- Since MoveIt! plans the robot path for the arm's wrist, it is necessary to convert all the pick poses to the wrist coordinate frame.
- The lookupTransform() method can provide the pose of a target relative to another pose.

Build Code and Run

- Compile the pick and place node:
  - In QTCreator: *Build* → *Build Project*

  - Alternatively, in a terminal:

```
catkin build collision_avoidance_pick_and_place
```

- Run the supporting nodes with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch
```

- In another terminal, run your node with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_pick_and_place.launch
```

- The tcp and wrist position at the pick will be printed in the terminal. You should see something like this:

```
[ INFO] [1400555434.918332145]: Move wait Succeeded
[ INFO] [1400555435.172714267]: Gripper opened
[ INFO] [1400555435.424279410]: target recognition succeeded
[ INFO] [1400555435.424848964]: tcp position at pick: [-0.8, 0.2, 0.17]
[ INFO] [1400555435.424912520]: tcp z direction at pick: [8.65611e-17, -8.66301e-
↪17, -1]
[ INFO] [1400555435.424993675]: wrist position at pick: x: -0.81555
y: 0.215563
z: 0.3

[ERROR] [1400555435.425051853]: pickup_box is not implemented yet.  Aborting.
```

API References

- lookupTransform()
- TF Transforms and other useful data types

Pick Up Box

In this exercise, we will move the robot through the pick motion while avoiding obstacles in the environment. This will be accomplished by planning for each pose and closing or opening the vacuum gripper when apropriate. Also, we will demonstrate how to create a motion plan that MoveIt! can understand and solve.

Locate Function

- In the main program, locate the function call to `application.pickup_box()`.
- Go to the source file of that function by clicking in any part of the function and pressing `F2` in QtCreator.
- Remove the first line containing the following `ROS_ERROR_STREAM ...` so that the program runs.

Complete Code

- Find every line that begins with the comment `Fill Code:` and read the description. Then, replace every instance of the comment `ENTER CODE HERE` with the appropriate line of code.

```
/* Fill Code:
    .
    .
    .
*/
/* ========  ENTER CODE HERE ======== */
```

- Inspect the `set_attached_object` method to understand how to manipulate a `robot_state` object which will then be used to construct a motion plan.
- Inspect the `create_motion_plan` method to see how an entire motion plan request is defined and sent.
- The execute() method sends a motion plan to the robot.

Build Code and Run

- Compile the pick and place node:
  - In QTCreator: *Build* → *Build Project*
  - Alternatively, in a terminal:

```
catkin build collision_avoidance_pick_and_place
```

- Run the supporting nodes with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch
```

- In another terminal, run your node with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_pick_and_place.launch
```

- The robot should go through the pick moves (approach, pick and retreat) in addition to the moves from the previous exercises. In the terminal you will see something like:

```
[ INFO] [1400555978.404435764]: Execution completed: SUCCEEDED
[ INFO] [1400555978.404919764]: Pick Move 2 Succeeded
[ERROR] [1400555978.405061541]: create_place_moves is not implemented yet. ␣
↪Aborting.
```

API References

- execute()
- MoveGroupInterface class

Create Place Moves

The gripper moves through three poses in order to place the box: Approach, place and retreat. In this exercise, we will create these place poses for the TCP coordinate frame and then transform them to the arm's wrist coordinate frame.

Locate Function

- In the main program , locate the function call to `application.create_place_moves()`.
- Go to the source file of that function by clicking in any part of the function and pressing `F2` in QtCreator.
- Remove the first line containing the following `ROS_ERROR_STREAM ...` so that the program runs.

Complete Code

- Find every line that begins with the comment `Fill Code:` and read the description. Then, replace every instance of the comment `ENTER CODE HERE` with the appropriate line of code.

```
/* Fill Code:
    .
    .
    .
*/
/* ========  ENTER CODE HERE ======== */
```

- The box's position at the place location is saved in the global variable `cfg.BOX_PLACE_TF`.
- The `create_manipulation_poses()` uses the values of the approach and retreat distances in order to create the corresponding poses at the desired target.
- Since MoveIt! plans the robot path for the arm's wrist, it is necessary to convert all the place poses to the wrist coordinate frame.
- The lookupTransform() method can provide the pose of a target relative to another pose.

Build Code and Run

- Compile the pick and place node:
  - In QTCreator: *Build* → *Build Project*
  - Alternatively, in a terminal:

```
catkin build collision_avoidance_pick_and_place
```

- Run the supporting nodes with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch
```

- In another terminal, run your node with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_pick_and_place.launch
```

- The tcp and wrist position at the place location will be printed on the terminal. You should see something like:

```
[ INFO] [1400556479.404133995]: Execution completed: SUCCEEDED
[ INFO] [1400556479.404574973]: Pick Move 2 Succeeded
[ INFO] [1400556479.404866351]: tcp position at place: [-0.4, 0.6, 0.17]
[ INFO] [1400556479.404934796]: wrist position at place: x: -0.422
y: 0.6
z: 0.3
```

(continues on next page)

(continued from previous page)

```
[ERROR] [1400556479.404981729]: place_box is not implemented yet.  Aborting.
```

API References

- lookupTransform()
- TF Transforms and other useful data types

Place Box

In this exercise, we will move the robot through the place motions while avoiding obstacles with an attached payload. In addition, the gripper must be opened or close at the appropriate time in order to complete the task.

Locate Function

- In the main program, locate the function call to `application.place_box()`.
- Go to the source file of that function by clicking in any part of the function and pressing `F2` in QtCreator.
- Remove the first line containing the following `ROS_ERROR_STREAM ...` so that the program runs.

Complete Code

- Find every line that begins with the comment `Fill Code:` and read the description. Then, replace every instance of the comment `ENTER CODE HERE` with the appropriate line of code.

```
/* Fill Code:
    .
    .
    .
*/
/* ========  ENTER CODE HERE ======== */
```

- The execute() method sends a motion plan to the robot.

Build Code and Run

- Compile the pick and place node:
  - In QTCreator: *Build* → *Build Project*
  - Alternatively, in a terminal:

```
catkin build collision_avoidance_pick_and_place
```

- Run the supporting nodes with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_setup.launch
```

- In another terminal, run your node with the launch file:

```
roslaunch collision_avoidance_pick_and_place ur5_pick_and_place.launch
```

- At this point your exercise is complete and the robot should move through the pick and place motions and then back to the wait pose. Congratulations!

API References

- execute()
- MoveGroupInterface class

3.6 Application Demo 2 - Descartes Planning and Execution

3.6.1 Application Demo 2 - Descartes Planning and Execution

Introduction

Goal

- This application will demonstrate how to use the various components in the Descartes library for planning and executing a robot path from a semi-constrained trajectory of points.

Objectives

- Become familiar with the Descartes workflow.
- Learn how to load a custom Descartes RobotModel.
- Learn how to create a semi-constrained trajectory from 6DOF tool poses.
- Plan a robot path with a Descartes Planner.
- Convert a Descartes Path into a MoveIt! message for execution.
- Executing the path on the robot.

Application Structure

In this exercise, we'll take a look at all the packages and files that will be used during the completion of these exercises.

Acquire and initialize the Workspace

```
cd ~/industrial_training/exercises/Descartes_Planning_and_Execution
cp -r template_ws ~/descartes_ws
cd ~/descartes_ws
source /opt/ros/melodic/setup.bash
catkin init
```

Download source dependencies

Use the wstool command to download the repositories listed in the `src/.rosinstall` file

```
cd ~/descartes_ws/src/
wstool update
```

Download debian dependencies

Make sure you have installed and configured the rosdep tool. Then, run the following command from the `src` directory of your workspace.

```
rosdep install --from-paths . --ignore-src -y
```

Build your workspace

```
catkin build --cmake-args -G 'CodeBlocks - Unix Makefiles'
```

If the build fails, then revisit the previous two steps to make sure all the dependencies were downloaded.

Source the workspace

Run the following command from your workspace parent directory

```
source devel/setup.bash
```

List All the Packages in the Application

```
cd ~/descartes_ws/src
ls -la
```

- `plan_and_run` : Contains the source code for the `plan_and_run` application. You'll be completing the exercises by editing source files in this package
- `ur5_demo_moveit_config` : Contains support files for planning and execution robot motions with Moveit. This package was generated with the Moveit Setup Assistant
- `ur5_demo_support` : Provides the robot definition as a URDF file. This URDF is loaded at run time by our `plan_and_run` application.
- `ur5_demo_descartes` : Provides a custom Descartes Robot Model for the UR5 arm. It uses a Inverse-Kinematics closed form solution; which is significantly faster than the numerical approach used by the **Moveit-StateAdapter**.

The `plan_and_run` package

```
roscd plan_and_run
ls -la
```

- `src` : Application source files.
- `src/demo_application.cpp` : A class source file that contains the application implementation code.
- `src/plan_and_run.cpp` : The application main access point. It invokes all the tasks in the application and wraps them inside the "`main`" routine.
- `src/tasks` : A directory that contains all of the source files that you'll be editing or completing as you make progress through the exercises.
- `include` : Header files
- `include/plan_and_run/demo_application.h` : Defines the application skeleton and provides a number of global variables for passing data at various points in the exercises.
- `launch`: Launch files needed to run the application
- `launch/demo_setup.launch` : Loads `roscore`, `moveit` and the runtime resources needed by our application.
- `launch/demo_run.launch` : Starts our application main executable as a ROS node.
- `config`: Directory that contains non-critical configuration files.

Main Application Source File

In the "`plan_and_run/src/plan_and_run_node.cpp`" you'll find the following code:

```
int main(int argc,char** argv)
{
  ros::init(argc,argv,"plan_and_run");
  ros::AsyncSpinner spinner(2);
  spinner.start();

  // creating application
  plan_and_run::DemoApplication application;

  // loading parameters
  application.loadParameters();

  // initializing ros components
  application.initRos();

  // initializing descartes
  application.initDescartes();

  // moving to home position
  application.moveHome();

  // generating trajectory
  plan_and_run::DescartesTrajectory traj;
  application.generateTrajectory(traj);


  // planning robot path
  plan_and_run::DescartesTrajectory output_path;
  application.planPath(traj,output_path);

  // running robot path
```

(continues on next page)

(continued from previous page)

```
  application.runPath(output_path);

  // exiting ros node
  spinner.stop();

  return 0;
}
```

In short, this program will run through each exercise by calling the corresponding function from the `application` object. For instance, in order to initialize Descartes the program calls `application.initDescartes()`. Thus each exercise consists of editing the source file where that exercise is implemented, so for `application.initDescartes()` you'll be editing the `plan_and_run/src/tasks/init_descartes.src` source file.

The DemoApplication Class

In the header file "`plan_and_run/include/plan_and_run/demo_application.h`" you'll find the definition for the application's main class along with several support constructs. Some of the important components to take notice of are as follows:

- Program Variables: Contain hard coded values that are used at various points in the application.

```
const std::string ROBOT_DESCRIPTION_PARAM = "robot_description";
const std::string EXECUTE_TRAJECTORY_ACTION = "execute_trajectory";
const std::string VISUALIZE_TRAJECTORY_TOPIC = "visualize_trajectory_curve";
const double SERVER_TIMEOUT = 5.0f; // seconds
const double ORIENTATION_INCREMENT = 0.5f;
const double EPSILON = 0.0001f;
const double AXIS_LINE_LENGHT = 0.01;
const double AXIS_LINE_WIDTH = 0.001;
const std::string PLANNER_ID = "RRTConnectkConfigDefault";
const std::string HOME_POSITION_NAME = "home";
```

- Trajectory Type: Convenience type that represents an array of Descartes Trajectory Points.

```
typedef std::vector<descartes_core::TrajectoryPtPtr> DescartesTrajectory;
```

- **DemoConfiguration Data Structure**: Provides variables whose values are initialize at run-time from corresponding ros parameters.

```
struct DemoConfiguration
{
  std::string group_name;                 /* Name of the manipulation group␣
↪containing the relevant links in the robot */
  std::string tip_link;                   /* Usually the last link in the kinematic␣
↪chain of the robot */
  std::string base_link;                  /* The name of the base link of the robot */
  std::string world_frame;                /* The name of the world link in the URDF␣
↪file */
  std::vector<std::string> joint_names;   /* A list with the names of the mobile␣
↪joints in the robot */


  /* Trajectory Generation Members:
   *  Used to control the attributes (points, shape, size, etc) of the robot␣
↪trajectory.
```

(continues on next page)

(continued from previous page)

```
   *  */
  double time_delay;               /* Time step between consecutive points in the␣
↪robot path */
  double foci_distance;            /* Controls the size of the curve */
  double radius;                   /* Controls the radius of the sphere on which the␣
↪curve is projected */
  int num_points;                  /* Number of points per curve */
  int num_lemniscates;             /* Number of curves*/
  std::vector<double> center;      /* Location of the center of all the lemniscate␣
↪curves */
  std::vector<double> seed_pose;   /* Joint values close to the desired start of the␣
↪robot path */

  /*
   * Visualization Members
   * Used to control the attributes of the visualization artifacts
   */
  double min_point_distance;       /* Minimum distance between consecutive trajectory␣
↪points. */
};
```

- **DemoApplication Class**: Main component of the application which provides functions for each step in our program. It also contains several constructs that turn this application into a ROS node.

```
class DemoApplication
{
public:
  /*  Constructor
   *    Creates an instance of the application class
   */
  DemoApplication();
  virtual ~DemoApplication();

  /* Main Application Functions
   *  Functions that allow carrying out the various steps needed to run a
   *  plan and run application.  All these functions will be invoked from within
   *  the main routine.
   */

  void loadParameters();
  void initRos();
  void initDescartes();
  void moveHome();
  void generateTrajectory(DescartesTrajectory& traj);
  void planPath(DescartesTrajectory& input_traj,DescartesTrajectory& output_path);
  void runPath(const DescartesTrajectory& path);

protected:

  /* Support methods
   *  Called from within the main application functions in order to perform␣
↪convenient tasks.
   */

  static bool createLemniscateCurve(double foci_distance, double sphere_radius,
                                    int num_points, int num_lemniscates,
```

(continues on next page)

(continued from previous page)

```
                                    const Eigen::Vector3d& sphere_center,
                                    EigenSTL::vector_Affine3d& poses);

  void fromDescartesToMoveitTrajectory(const DescartesTrajectory& in_traj,
                                       trajectory_msgs::JointTrajectory& out_
↪traj);

  void publishPosesMarkers(const EigenSTL::vector_Affine3d& poses);


protected:

  /* Application Data
   *  Holds the data used by the various functions in the application.
   */
  DemoConfiguration config_;



  /* Application ROS Constructs
   *  Components needed to successfully run a ros-node and perform other important
   *  ros-related tasks
   */
  ros::NodeHandle nh_;                          /* Object used for creating and␣
↪managing ros application resources*/
  ros::Publisher marker_publisher_;             /* Publishes visualization message to␣
↪Rviz */
  std::shared_ptr<actionlib::SimpleActionClient<moveit_msgs::ExecuteTrajectoryAction>>
↪   moveit_run_path_client_ptr_; /* Sends a robot trajectory to moveit for execution␣
↪*/



  /* Application Descartes Constructs
   *  Components accessing the path planning capabilities in the Descartes library
   */
  descartes_core::RobotModelPtr robot_model_ptr_; /* Performs tasks specific to the␣
↪Robot
                                                     such IK, FK and collision␣
↪detection*/
  descartes_planner::SparsePlanner planner_;      /* Plans a smooth robot path given␣
↪a trajectory of points */

};
```

Application Launch File

This file starts our application as a ROS node and loads up the necessary parameters into the ROS parameter server. Observe how this is done by opening the "`plan_and_run/launch/demo_run.launch`" file:

```
<launch>
  <node name="plan_and_run_node" type="plan_and_run_node" pkg="plan_and_run" output=
↪"screen">
    <param name="group_name" value="manipulator"/>
    <param name="tip_link" value="tool"/>
```

(continues on next page)

(continued from previous page)

```
    <param name="base_link" value="base_link"/>
    <param name="world_frame" value="world"/>
    <param name="trajectory/time_delay" value="0.1"/>
    <param name="trajectory/foci_distance" value="0.07"/>
    <param name="trajectory/radius" value="0.08"/>
    <param name="trajectory/num_points" value="200"/>
    <param name="trajectory/num_lemniscates" value="4"/>
    <rosparam param="trajectory/center">[0.36, 0.2, 0.1]</rosparam>
    <rosparam param="trajectory/seed_pose">[0.0, -1.03, 1.57 , -0.21, 0.0, 0.0]</
↪rosparam>
    <param name="visualization/min_point_distance" value="0.02"/>
  </node>
</launch>
```

- Some of the important parameters are explained as follows:
- group_name: A namespace that points to the list of links in the robot that are included in the arm's kinematic chain (base to end-of-tooling links). This list is defined in the `ur5_demo_moveit_config` package.
- tip_link: Name of the last link in the kinematic chain, usually the tool link.
- base_link: Name for the base link of the robot.
- world_frame: The absolute coordinate frame of reference for all the objects defined in the planning environment.
- The parameters under the "`trajectory`" namespace are used to generate the trajectory that is fed into the Descartes planner.
- trajectory/seed_pose: This is of particular importance because it is used to indicate preferred start and end joint configurations of the robot when planning the path. If a ''seed_pose" wasn't specified then planning would take longer since multiple start and end joint configurations would have to be taken into account, leading to multiple path solutions that result from combining several start and end poses.

General Instructions

In this exercise, we'll demonstrate how to run the demo as you make progress through the exercises. Also, it will be shown how to run the system in simulation mode and on the real robot.

Main Objective

In general, you'll be implementing a `plan_and_run` node incrementally. This means that in each exercise you'll be adding individual pieces that are needed to complete the full application demo. Thus, when an exercise is completed run the demo in simulation mode in order to verify your results. Only when all of the exercises are finished should you run it on the real robot.

Complete Exercises

1. To complete an exercise, open the corresponding source file under the `src/plan_and_run/src/tasks/` directory. For instance, in Exercise 1 you'll open `load_parameters.cpp`.
2. Take a minute to read the header comments for specific instructions for how to complete this particular exercise. For instance, the `load_parameters.cpp` file contains the following instructions and hints:

```
/* LOAD PARAMETERS
  Goal:
    - Load missing application parameters into the node from the ros parameter server.
    - Use a private NodeHandle in order to load parameters defined in the node's␣
↪namespace.
  Hints:
    - Look at how the 'config_' structure is used to save the parameters.
    - A private NodeHandle can be created by passing the "~" string to its␣
↪constructor.
*/
```

1. Don't forget to comment out the line:

```
ROS_ERROR_STREAM("Task '"<<__FUNCTION__ <<"' is incomplete. Exiting"); exit(-1);
```

This line is usually located at the beginning of each function. Omitting this step will cause the program to exit immediately when it reaches this point.

1. When you run into a comment block that starts with `/* Fill Code:` this means that the line(s) of code that follow are incorrect, commented out or incomplete at best. Read the instructions following `Fill Code` and complete that task as described. An example of instructions comment block is the following:

```
  /*  Fill Code:
   * Goal:
   * - Create a private handle by passing the "~" string to its constructor
   * Hint:
   * - Replace the string in ph below with "~" to make it a private node.
   */
```

1. The `[ COMPLETE HERE ]` entries are meant to be replaced by the appropriate `code entry`. The right code entries could either be program variables, strings or numeric constants. One example is shown below:

```
ros::NodeHandle ph("[ COMPLETE HERE ]: ?? ");
```

In this case the correct replacement would be the string `"~"`, so this line would look like this:

```
ros::NodeHandle ph("~");
```

1. As you are completing each task in this exercise, you can run the demo (see following sections) to verify that it was completed properly.

Run Demo in Simulation Mode

1. In a terminal, run the setup launch file as follows:

```
roslaunch plan_and_run demo_setup.launch
```

- When the virtual robot is ready , Rviz should be up and running with a UR5 arm in the home position and you'll see the following messages in the terminal:

```
      .
      .
      .
*********************************************************
* MoveGroup using:
```

(continues on next page)

(continued from previous page)

```
*      - CartesianPathService
*      - ExecutePathService
*      - KinematicsService
*      - MoveAction
*      - PickPlaceAction
*      - MotionPlanService
*      - QueryPlannersService
*      - StateValidationService
*      - GetPlanningSceneService
*      - ExecutePathService
*********************************************************

[ INFO] [1430359645.694537917]: MoveGroup context using planning plugin ompl_
↪interface/OMPLPlanner
[ INFO] [1430359645.694700640]: MoveGroup context initialization complete

All is well! Everyone is happy! You can start planning now!
```

- This launch file only needs to be run once.

1. In a separate terminal, run the application launch file:

```
roslaunch plan_and_run demo_run.launch
```

- Look in the Rviz window and the arm should start moving.

Run Demo on the Real Robot

Notes

- Make sure that you can `ping` the robot and that there aren't any obstacles near it.

1. In a terminal, run the setup launch file as follows:

```
roslaunch plan_and_run demo_setup.launch sim:=false robot_ip:=000.000.0.00
```

Notes:

- Enter the robot's actual IP address into the `robot_ip` argument. The robot model in Rviz should be in about the same position as the real robot.

1. In a separate terminal, run the application launch file:

```
roslaunch plan_and_run demo_run.launch
```

- This time the real robot should start moving.

Load Parameters

In this exercise, we'll load some ROS parameters to initialize important variables within our program.

Locate Exercise Source File

- Go to the main application source file located in `plan_and_run/src/plan_and_run_node.cpp`.

- In the main program, locate the function call to `application.loadParameters()`.
- Go to the source file for that function located in the `plan_and_run/src/tasks/load_parameters.cpp`. Alternatively, in Eclipse you can click in any part of the function and press "F2" to bring up that file.
- Comment out the first line containing the `ROS_ERROR_STREAM ...` entry so that the function doesn't quit immediately.

Complete Code

- Find comment block that starts with `/* Fill Code:` and complete as described.
- Replace every instance of `[ COMPLETE HERE ]` accordingly.

Build Code and Run

- `cd` into your catkin workspace and run:

```
catkin build --cmake-args -G 'CodeBlocks - Unix Makefiles'
source ./devel/setup.bash
```

- Then run the application launch file:

```
roslaunch plan_and_run demo_setup.launch
roslaunch plan_and_run demo_run.launch
```

API References

ros::NodeHandle

NodeHandle::getParam()

Initialize ROS

In this exercise, we'll initialize the ros components that our application needs in order to communicate to MoveIt! and other parts of the system.

Locate Exercise Source File

- Go to the main application source file located in `plan_and_run/src/plan_and_run_node.cpp`.
- In the main program, locate the function call to `application.initRos()`.
- Go to the source file for that function located in the `plan_and_run/src/tasks/init_ros.cpp`.
  - *Alternatively, in QTCreator, click on any part of the function and press "F2" to bring up that file.*
- Comment out the first line containing the `ROS_ERROR_STREAM ...` entry so that the function doesn't quit immediately.

Complete Code

- Observe how the ros Publisher `marker_publisher_` variable is initialized. The node uses it to publish a `visualization_msgs::!MarkerArray` message for visualizing the trajectory in RViz.
- Initialize the `moveit_run_path_client_ptr_` action client with the `ExecuteTrajectoryAction` type.
- Find comment block that starts with `/* Fill Code:` and complete as described.
- Replace every instance of `[ COMPLETE HERE ]` accordingly.

Build Code and Run

- `cd` into your catkin workspace and run `catkin build`
- Then run the application launch file:

```
roslaunch plan_and_run demo_run.launch
```

API References

visualization_msgs::MarkerArray

NodeHandle::serviceClient()

Initialize Descartes

This exercise consists of setting up the Descartes Robot Model and Path Planner that our node will use to plan a path from a semi-constrained trajectory of the tool.

Locate Exercise Source File

- Go to the main application source file located in `plan_and_run/src/plan_and_run_node.cpp`.
- In the main program, locate the function call to `application.initDescartes()`.
- Go to the source file for that function located in the `plan_and_run/src/tasks/init_descartes.cpp`.
  - Alternatively, in QTCreator, you can click in any part of the function and press "F2" to bring up that file.
- Comment out the first line containing the `ROS_ERROR_STREAM( ...` entry so that the function doesn't quit immediately.

Complete Code

- Invoke the descartes_core::RobotModel::initialize() method in order to properly initialize the robot.
- Similarly, initialize the Descartes planner by passing the `robot_model_` variable into the `descartes_core::!DensePlanner::initialize()` method.
- Find comment block that starts with `/* Fill Code:` and complete as described.
- Replace every instance of `[ COMPLETE HERE ]` accordingly.

Build Code and Run

- `cd` into your catkin workspace and run `catkin build`
- Then run the application launch file:

```
roslaunch plan_and_run demo_run.launch
```

API References

descartes_core::RobotModel descartes_planner::DensePlanner

Move Home

In this exercise, we'll be using MoveIt! in order to move the arm.

Locate Exercise Source File

- Go to the main application source file located in `plan_and_run/src/plan_and_run_node.cpp`.
- In the main program , locate the function call to `application.moveHome()`.
- Go to the source file for that function located in the `plan_and_run/src/tasks/move_home.cpp`.
  - Alternatively, in QTCreator, click on any part of the function and press "F2" to bring up that file.
- Comment out the first line containing the `ROS_ERROR_STREAM( ...` entry so that the function doesn't quit immediately.

Complete Code

- Use the MoveGroupInterface::move() method in order to move the robot to a target.
- The moveit_msgs::MoveItErrorCodes structure contains constants that you can use to check the result after calling the `move()` function.
- Find comment block that starts with `/* Fill Code:` and complete as described.
- Replace every instance of `[ COMPLETE HERE ]` accordingly.

Build Code and Run

- `cd` into your catkin workspace and run `catkin build`
- Then run the application launch file:

```
roslaunch plan_and_run demo_run.launch
```

API References

setNamedTarget()

MoveGroupInterface class

Generate a Semi-Constrained Trajectory

In this exercise, we'll be creating a Descartes trajectory from an array of cartesian poses. Each point will have rotational freedom about the z axis of the tool.

Locate exercise source file

- Go to the main application source file located in `plan_and_run/src/plan_and_run_node.cpp`.
- In the main program, locate the function call to `application.generateTrajectory()`.
- Go to the source file for that function located in the `plan_and_run/src/tasks/generate_trajectory.cpp`.
  - Alternatively, in QTCreator, click on any part of the function and press "F2" to bring up that file.
- Comment out the first line containing the `ROS_ERROR_STREAM( ...` entry so that the function doesn't quit immediately.

Complete Code

- Observe how the 'createLemniscate()' is invoked in order to generate all the poses of the tool for the trajectory. The poses from it are then used to create the Descartes Trajectory.
- Use the AxialSymmetric constructor to specify a point with rotational freedom about the z-axis.
- The AxialSymmetricPt::FreeAxis::Z_AXIS enumeration constant allows you to specify the **Z** as the free rotational axis
- Find comment block that starts with `/* Fill Code:` and complete as described .
- Replace every instance of `[ COMPLETE HERE ]` accordingly.

Build Code and Run

- CD into your catkin workspace and run `catkin build`
- The run the application launch file

```
roslaunch plan_and_run demo_run.launch
```

API References

descartes_trajectory::AxialSymmetricPt

Plan a Robot Path

In this exercise, we'll pass our trajectory to the Descartes planner in order to plan a robot path.

Locate Exercise Source File

- Go to the main application source file located in `plan_and_run/src/plan_and_run_node.cpp`.
- In the main program, locate the function call to `application.planPath()`.
- Go to the source file for that function located in the `plan_and_run/src/tasks/plan_path.cpp`. Alternatively, in Eclipse you can click in any part of the function and press "F2" to bring up that file.
- Comment out the first line containing the `ROS_ERROR_STREAM( ...` entry so that the function doesn't quit immediately.

Complete Code

- Observe the use of the AxialSymmetricPt::getClosestJointPose() in order to get the joint values of the robot that is closest to an arbitrary joint pose. Furthermore, this step allows us to select a single joint pose for the start and end rather than multiple valid joint configurations.
- Call the DensePlanner::planPath() method in order to compute a motion plan.
- When planning succeeds, use the DensePlanner::getPath() method in order to retrieve the path from the planner and save it into the `output_path` variable.
- Find comment block that starts with `/* Fill Code:` and complete as described.
- Replace every instance of `[ COMPLETE HERE ]` accordingly.

Build Code and Run

- `cd` into your catkin workspace and run `catkin build`
- Then run the application launch file:

```
roslaunch plan_and_run demo_run.launch
```

API References

descartes_planner::DensePlanner

Run a Robot Path

In this exercise, we'll convert our Descartes path into a MoveIt! trajectory and then send it to the robot.

Locate Exercise Source File

- Go to the main application source file located in `plan_and_run/src/plan_and_run_node.cpp`.
- In the main program, locate the function call to `application.runPath()`.
- Go to the source file for that function located in the `plan_and_run/src/tasks/run_path.cpp`.
  - Alternatively, in QTCreator, click on any part of the function and press "F2" to bring up that file.
- Comment out the first line containing the `ROS_ERROR_STREAM( ...` entry so that the function doesn't quit immediately.

Complete Code

- Find comment block that starts with `/* Fill Code:` and complete as described.
- Replace every instance of `[ COMPLETE HERE ]` accordingly.

Build Code and Run

- `cd` into your catkin workspace and run `catkin build`
- Then run the application launch file:

```
roslaunch plan_and_run demo_run.launch
```

API References

MoveGroupInterface::move()

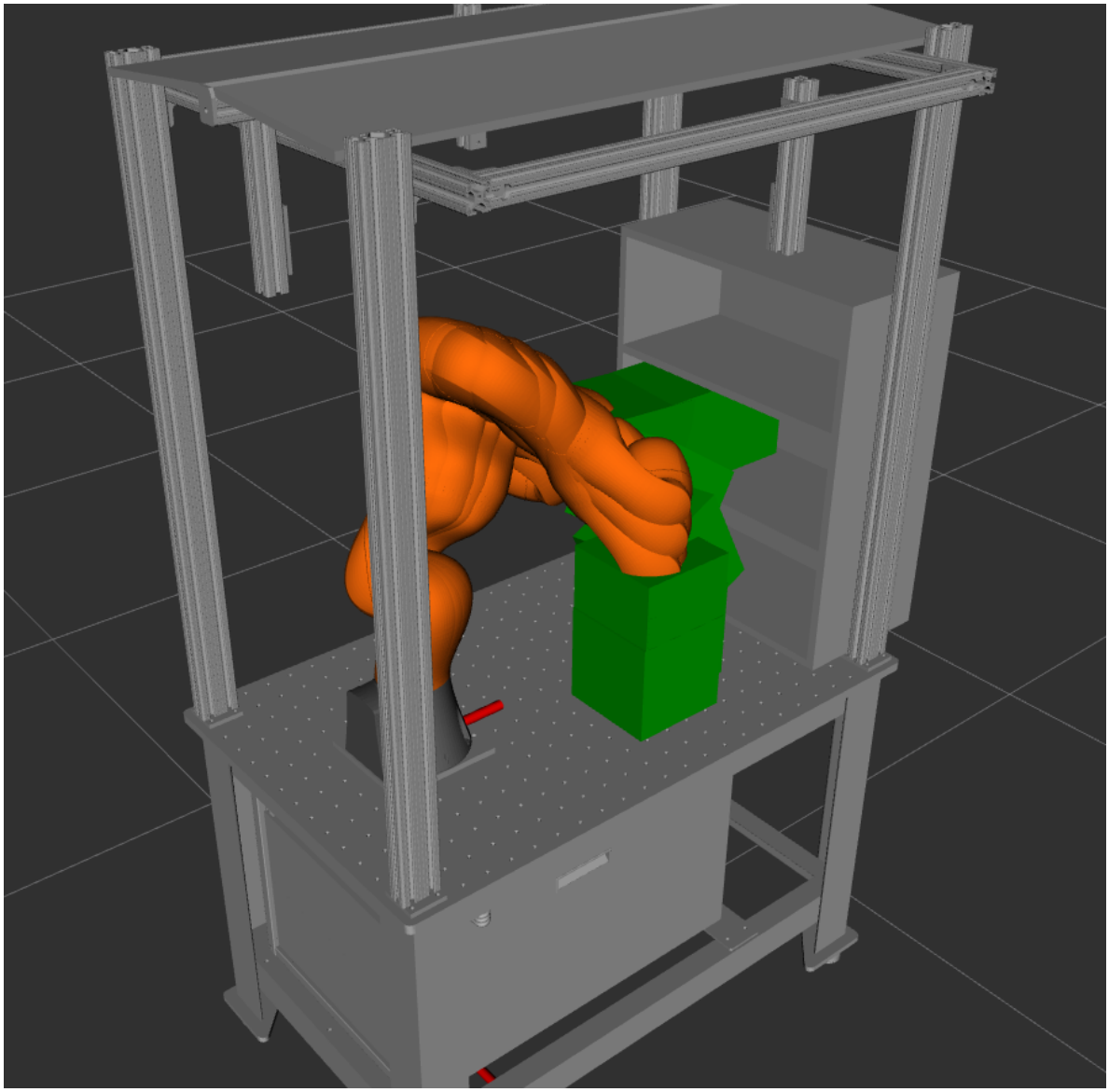
3.7 Application Demo 3 - Optimization Based Path Planning

3.7.1 Application Demo 3 - Optimization Based Planning

Optimization Based Planning Introduction

Goal

- The purpose of these exercises is to implement a ROS node that drives a robot through a series of moves and actions in order to complete a pick and place task. In addition, they will serve as an example of how to utilize more specilized tools such as the optimization based path planner, TrajOpt, and the control interfaces of the KUKA iiwa robot, while also integrating a variety of software capabilities (perception, controller drivers, I/O, inverse kinematics, path planning, collision avoidance, etc) into a ROS-based industrial application.



Objectives

- Understand the components and structure of a real or simulated robot application.
- Leverage perception capabilities using PCL
- Learn how to setup a TrajOpt path planning problem
- Learn how to use costs and constraints in TrajOpt
- Learn how to move the arm to a joint or Cartesian position
- Plan collision-free paths for a pick and place task
- Send these trajectories to real robot hardware

Outline

- First we will explore the given template code and import it into QT Creator
- We will then use the data from a simulated 3D sensor and PCL to find the top of the box as the pick point
- Next, we will give a conceptual introduction to TrajOpt exploring the procedure for building a problem, adding costs, and solving for a trajectory
- We will then build a series of helper functions for TrajOpt to perform the different parts of a pick and place operation and test in simulation
- With simulation working we will move to a real 3D sensor, learning to calibrate it and testing it in conjunction with a simulated robot.
- Finally, we will move to a physical robot with a real 3D sensor.

Inspect the "pick_and_place" Package

In this exercise, we will get familiar with all the files that you'll be interacting with throughout these exercises.

Acquire and initialize the workspace

```
cp -r ~/industrial_training/exercises/Optimization_Based_Planning/template_ws ~/
↪optimized_planning_ws
cd ~/optimized_planning_ws
source /opt/ros/melodic/setup.bash
catkin init
```

Download source dependencies

Use the wstool command to download the repositories listed in the **src/.rosinstall** file

```
cd ~/optimized_planning_ws/src/
wstool update
```

Download debian dependencies

Make sure you have installed and configured the rosdep tool. Then, run the following command from the **/src** directory of your workspace.

```
rosdep install --from-paths . --ignore-src -y -r
```

We also need to install glfw.

```
sudo apt install libglfw3-dev libglfw3
```

Build your workspace

```
catkin build
```

If the build fails then revisit the previous two steps to make sure all the dependencies were downloaded.

Source the workspace

Run the following command from your workspace parent directory

```
source devel/setup.bash
```

Explore your workspace

Your workspace contains 7 packages

- ***pick_and_place*** - This is the main pick and place package. It contains the main pick and place node and the launch file to bring up the system.
- ***pick_and_place_perception*** - This package contains the perception pipeline. We will develop a PCL algorithm to use a 3d camera to detect a pick object and pass it's location to the pick and place node.
- ***pick_and_place_support*** - This package contains the support files for the system. This is where files such as the robot model are stored. It also contains code for camera calibration.
- ***gl_depth_sim*** - This package simulates a 3d camera by using OpenGL to convert a mesh and a camera pose into a point cloud. It was installed from github by wstool.
- ***tesseract*** - This package runs the planning environment. It contains tools for robot path planning, collision checking, and visualization. It was installed from github by wstool.
- ***trajopt*** - This package contains the trajopt motion planner. We will use this to path plan - optimizing for avoiding collisions, following a set of desired waypoints, and meeting physical robot constraints. It was installed from github by wstool.
- ***iiwa_stack*** - This package contains the hardware driver for controlling a KUKA iiwa collaborative robot. It also contains the robot description that is used for simulation and path planning. It was installed from github by wstool.

Look into the src directory

Looking in the src directory of the main pick_and_place package, you see the following.

Nodes:

- **test_bed_core_node.cpp** : Main application thread. Contains all necessary headers and function calls to perform a scripted pick and place operation.
- **sensor_simulator_3d.cpp** : Simulates data from a 3d camera sensor. Only used in simulation.

Utilities

- **trajopt_pick_and_place_constructor.cpp** : Contains source code for trajopt pick and place helper functions. This is where the majority of the trajopt implementation will be.

Package Setup

In this exercise, we'll build our package dependencies and configure the package for the Qt Creator IDE.

Import Workspace into QT

1. In QT do the following:

```
File -> New -> Other Project -> ROS Workspace
```

1. Fill the Project Name and Location information

- Name: `pick_and_place`
- Distribution: `melodic`
- Build System: `CatkinTools`
- Workspace Path `~/optimized_planning_ws`

1. Click next to go to the summary and leave version control to defaults
2. With the project open, type ctrl+b to build the project. By building the project through QT Creator, it allows the IDE to enable features like autocomplete effectively.

Start in Simulation Mode

In this exercise, we will start a ROS system that is ready to move the robot in simulation mode.

Run setup launch file in simulation mode (simulated robot and sensor)

In a terminal

```
roslaunch pick_and_place pick_and_place.launch
```

Rviz will display all the workcell components including the robot in its default position.

Look in `test_bed_core_node.cpp`. This code is already complete. A service to find the pick target is run, and a TrajOpt problem is setup to perform a pick and place operation. While the service to find the pick target will run, it will not find a pick and the pathplanning will not run. These will be the next tasks.

Detect Box Pick Point

The first step in a pick and place operation is that the pick location must be found. To do that, we will leverage a 3D camera sensor and the Point Cloud Library (PCL).

The coordinate frame of the box's pick can be requested from a ros service that detects it by processing the sensor data. In this exercise, we will learn to write a ROS service and apply the neccessary filters to locate the pick location.

Overview of the process

The perception node is launched. This registers a new service with `roscore` and pulls necessary parameters from the ROS parameter server. It then waits until the service is called. When the ROS perception service is called:

1. Service pulls latest point cloud from the 3D sensor

2. Point cloud is cropped to exclude areas outside of the work cell
3. RANSAC plane segmentation is used to remove the work table from the point cloud
4. Euclidean clster extraction is used to cluster the remaining points
5. Outliers are removed
6. The largest cluster is taken as the pick object (additional logic could be added here to support multiple pick objects in view)
7. RANSAC plane segmentations is used to find the top of the box
8. The centroid of these points is calcuated
9. The service returns this pose

Explore processing_node.launch

Open processing_node.launch in demo3_perception/launch. This file launches the perception node. Note the standalone flag which can be set to true for testing the perception without the rest of the system.

Additionally, it defines adds some associated rosparams to the parameter server. Explore these parameters. While the given values should work in simulation, it is likely that some of the PCL filter parameters will need to be changed when moving to real hardware. Here are some of them associated with the ROS setup.

- cloud_debug: true means intermediate point clouds will be published for debugging
- cloud_topic: Topic from which the service pulls the point cloud
- world_frame: Frame into which the point cloud is placed
- camera_frame: Frame associated with the camera location

Define ROS Service

We first review the ROS service definition. This file will define the service inputs and outputs.

- In the pick_and_place_perception/srv directory, open GetTargetPose.srv
- Ensure that the following code is in this file. Note that the service request is empty. It returns a boolean flag, a ROS message of the geometry_msgs/Pose type for the location of the center of the top of the box, and two geometry_msgs/Pose messages that define a bounding box around the top of the box.

```
# Request - empty
---
# Response - service returns a bool, a geometry_msgs/Pose, and 2 geometry_msgs/Points
bool succeeded
geometry_msgs/Pose target_pose
geometry_msgs/Point min_pt
geometry_msgs/Point max_pt
```

Explore perception_node architecture

In order for a RIS service to be registered on roscore, it must be advertised similarly to a node. This is done with `nh.advertiseService(args)`. While this is often done the main(), for our use case this is done in the constructor of the PerceptionPipeline class. The service function is then a member of that class. Creating a pipeline class has several advantages.

- It allows us to create ROS publishers as class members that are also maintained between service calls. We are using this for debugging in our case by publishing the intermediate point clouds.
- It allows us to read parameters and store them in class members that are maintained between service calls, reducing processing time.
- It allows us to instatiate multiple instances of the same pipeline.

Complete Code

- A template had been provided with some of the setup complete
- Find every line that begins with the comment “”Fill Code: ‘”” and read the description. Then, replace every instance of the comment “”ENTER CODE HERE”” with the appropriate line of code

```
/* Fill Code:
     .
     .
     .
*/
//ENTER CODE HERE: Brief Description
```

- The ‘”find_pick_client”’ object in your programs can use the ‘”call()”’ method to send a request to a ros service.
- The ros service that receives the call will process the sensor data and return the pose for the box pick in the service structure member ‘”srv.response.target_pose”’.

Build Code and Run

- Compile the pick and place node in QT

```
Build -> Build Project
```

- Alternatively, in a terminal cd into your workspace directory and do the following

```
catkin build
```

- Run your node with the launch file used before which calls the processing_node.launch. While you could call that launch file directly, the environment would not be loaded, and there would be no point cloud to process.

```
roslaunch pick_and_place pick_and_place.launch
```

- test_bed_core_node will call your service automatically. However, you can also test your code from the command line by calling `rosservice call /find_pick` when a point cloud is is being published on the cloud_topic (/cloud by default).
- Some errors and warnings will be present when the template code is run. As the correct code is filled, these will disappear until a fully defined Pose message is returned from the service.

Additional exploration (optional)

You may notice that this service does not run terribly quickly. Aside from publishing the clouds for debugging, intermediate copies of the point cloud are made that simplify the demonstration but hurt performance. Additionally, PCL has a wide variety of filters available and the solution given here is just one of many possible. Use the included timing functions to explore changes that can speed processing.

API References

call()

Introduction to Trajopt

What is TrajOpt?

Trajopt is an optimization based path planner that utilizes Sequential Quadratic Programming. It models the path planning problem as an optimization problem of the costs and constraints provided by the user, then iteratively attempts to converge to a global minimum (though with no guarantee).

Why use TrajOpt?

1. ***Speed***: Trajopt solves many problems very quickly.
2. ***Reliability***: Trajopt can reliably solve a large number of planning problems.
3. ***Flexibility***: Trajopt provides the ability for a user to define their own costs and constraints. This means that you can define a unique optimization objection specific to any application.

Basic Usage

The typical workflow of using Trajopt for pathplanning is as follows.

1. Create a ProblemConstructionInfo (pci). This can either be loaded from a json file or generated directly in C++. This contains information about the costs/constraints desired, the robot, and the planning environment.
2. Convert ProblemConstructionInfo to a TrajOptProb using ConstructProblem. This constructs the optimization into a standardized format used by the optimizers.
3. Optimize. The optimizer will optimize a `num_timesteps x num_joints + 1` matrix to minimize the costs and constraints. Each column represents a joint with the exception of the last one which is time.

The next sections will cover some of the important parts of setting up a TrajOpt problem.

1. Basic Info

The basic_info section of the pci contains (as you might expect) basic information for the planner.

- ***[int] n_steps***: The number of trajectory states (timesteps) to be used in the planning of the trajectory.
- ***[string] manip***: The name of the manipulator of the robot to plan for.
- ***[string] robot (optional)***: TODO
- ***[bool] start_fixed***: Whether to force the first trajectory state to exactly equal the first state given in the init info. If true, the associated joint values will be set to the initial conditions with an equality constraint
- ***[vector/list of ints] dofs_fixed (optional)***: Indices corresponding to any degrees of freedom that you want to have remain in the same position the entire trajectory. These will be set to the initial conditions with an equality constraint
- ***[sco::ModelType] convex_solver (optional)***: This specifies the solver to use. If unspecified, the included BPMPD solver will be used.

- ***[bool] use_time ***: If true, TrajOpt adds a column to represent (1/dt). This must be set to true if any of the costs/constraints are set to use time.
- ***[double] dt_upper_limit (optional)***: This is the upper limit of 1/dt values allowed. Note this value is 1/dt ***NOT*** dt.
- ***[double] dt_lower_limit (optional)***: This is the lower limit of 1/dt values allowed. Note this value is 1/dt ***NOT*** dt.

2. Init Info

The init_info section of the pci contains information detailing the initial trajectory Trajopt should start from.

- ***[InitInfo::Type] type***: The type of initialization. Valid values are:
  - ***STATIONARY***: Initializes the entire trajectory to the current joint states of the robot. No data is needed.
  - ***GIVEN_TRAJ***: You provide the entire initial trajectory in the **data** member
- ***JOINT_INTERPOLATED***: You provide an **endpoint** member for the initial trajectory. The trajectory is the joint interpolated between the current state and the endpoint.
- ***[TrajArray] data***: Array containing the initialization information. - If doing C++, must contain a trajectory with all joints over all timesteps. - If using Json, you should only provide what is needed given the type you chose.

3. Optimization Info (optional)

This section of the pci can typically be set to defaults. In fact, when using a Tesseract planner as we will in this demo, the selections here get overwritten. These are the parameters that govern the SQP optimization routine. More details can be found by exploring the TrajOpt documentation.

4. Costs

These are functions that you desire to be minimized, such as joint accelerations or cartesian velocities. The optimizer will seek to optimize the sum of the weighted costs subject to the constraints below. These correspond to the objective function. These will be discussed in more detail in 3.6.

5. Constraints

These are the conditions that must be satisfied by the optimizer. Failure to satisfy these constraints will cause the weighting of these terms to increase until they are satisfied or the maximum number of iterations is reached. These directly correspond to optimizer constraints. Section 3.6 will cover the costs and contraints available.

Performance

There are several things that you can do to help speed the trajopt optimization.

1. Be sure to build your project in Release mode. Using the command line this is `catkin build --cmake-args -DCMAKE_BUILD_TYPE=Release"` In Qt Creator, the build type option is under the projects tab.
2. Set the Trajopt logging level higher than Info before optimization. Do this with `#include <trajopt_utils/logging.hpp> trajopt::gLogLevel = util::LevelWarn;`

3. Change optimization parameters. The parameters in BasicTrustRegionSQPParameters can effect both the speed with which the optimization converges and the point at which the optimization triggers as having converged.
4. Change your problem. This may seem obvious, but sometimes constraints can be mutually exlusive. You can also have two very similar competing costs (this is one of TrajOpt's advantages). For example, a cartesian pose constraint that is out of the robot workspace would cause a failure. A cartesian pose cost that is in collision could slow down optimization. Be judicious about which terms are really necessary to be constraints and which ones could be costs.

Notes

At its core, trajopt does nothing more than take some cost functions and minimize them, but there are some details that the user should be aware of.

In updating the trajectory, trajopt treats both costs and constraints as costs. If it fails to satisfy constraints, then it will increase the weight (penalty) of the constraints in an effort to have them overcome the weight of the costs to move the trajectory toward satisfying the constraints. It will increase the penalty applied to constraint violations a finite number of times, so if the weights of your costs are too high, it may fail produce a result that satisfies constraints.

Furthermore, it should be noted that the result will be organized into a m x n matrix, where m is the number of timesteps and n is the number of joints + a column for time. The ith row then represents the joint state at the ith timestep of the trajectory.

Costs and Constraints

This section provides an overview of the common features of costs and constraints, the different types and their meaning, and a list of the currently avaible costs and constraints. Additional information can be found in the TrajOpt documentation as well as in the trajopt_examples package.

Common Ground

Costs and constraints are specific types of *TermInfos* in Trajopt. Many of the same types of terms can be used as either a cost or a constraint. This is set by setting the term_type member to either TT_COST or TT_CNT. When using the json interface, this is handled for you. Additionally, there is a TT_USE_TIME option that allows some terms to use time parameterization.

All terms also have an optional `name` parameter that you can set to specify a name for that specific term. When using Json, this defaults to the term type. In C++, name is empty by default.

Adding Terms with Json

When constructing the problem using Json, it will go in a list of costs or constraints. You then need to specificy the type of term (Joint Velocity, etc) and a member :code:`params` that contains all of the parameters for that type of term.

```
{
"costs" :
[
OR
{
"constraints" :
[
```

(continues on next page)

(continued from previous page)

```
...

    {
        "type" : "this_term_type",
        "name" : "optional_term_name",
        "params" :
        {
            "string_a" : "string",
            "int_b" : 1,
            "float_c" : 0.5
        }
    }
]
}
```

Adding Terms with C++

When constructing the problem in C++, each term has a respective TermInfo object. Create a `std::shared_ptr` to the type of term you wish to add, update the parameter members of that object, and `push_back` that term to the end of `cost_infos` or `cnt_infos` of the `ProblemConstructionInfo` object. If the term can be either a cost or a constraint, do not forget to set the `term_type` member to the proper enumeration to reflect whether you intend the term to be a cost or a constraint.

```
// make the term
std::shared_ptr<TypeTermInfo> term(new TypeTermInfo);

// set the parameters
term->string_a = "string";
term->int_b = 1;
term->float_c = 0.5;

// optionally set its name
term->name = "optional_term_name";

// push_back onto the correct infos vector
// pci = ProblemConstructionInfo

// if cost
term->term_type = trajopt::TT_COST
pci.cost_infos.push_back(term);

// or

// if constraint
term->term_type = trajopt::TT_CNT
pci.cnt_infos.push_back(term)
```

Joint Level Terms

There are a number of joint level costs available in TrajOpt. These are

- Joint Position

- Joint Velocity
- Joint Acceleration
- Joint Jerk

Some important notes:

- Each of these is calculated from the joint position values using numerical differentiation. If the trajectory is too finely discretized, these values may be somewhat meaningless
- When time parameterization is disabled, these costs assume unit time steps. This means velocity for instance is just `x1-x0`. This is a useful cost, but the user should be aware of what it means.
- Each of these costs have a target (required), an upper_tol (optional), and a lower_tol (optional).
  - The target sets the desired value (usually 0) for all joints. Thus it should be a vector the length of the DOF.
  - If `upper_tol==lower_tol==0` then an equality costs/cnt will be set with a squared error.
  - If `upper_tol!=lower_tol` then a hinge cost/cnt will be set, centered about target and hinged at the tolerances.
    - Note: This "dead band" of 0 cost can cause numerical issues sometimes. It is often advantageous to set a small equality cost in addition to a larger hinge cost if one is needed.

Cartesian Terms

Similarly, there are a number of cartesian terms available in TrajOpt. They are:

- Dynamic Cartesian Pose (DynamicCartPoseTermInfo)
- Cartesian Pose (CartPoseTermInfo)
- Cartesian Velocity (CartVelTermInfo)

Note: These make use of the FK and numeric gradient calculation and come with a higher computation cost than joint level costs. However, this is not significant for most cases, and it is often possible to solve even hundreds of these at once (see the puzzle_piece example).

Miscellaneous Terms

Miscellaneous terms include:

- Collision

The collision cost is one of TrajOpt's most notable features. Using features in Tesseract, it is able to use convex-convex collision checking to very quickly calculate collisions. In addition to discrete point collision checking. It also has the ability to do collision checking of a swept volume between two joint states. When an object is found in collision, it is able to provide a vector to get out of collision - allowing for more intelligent state updates between iterations.

Create Pick and Place Helpers

With knowledge of the way that TrajOpt works, we will now use it to build our application. Consider the motion steps in a pick and place operation.

Pick

- Free space move to approach pose
- Linear move to target pose

Place

- Linear move to retreat pose
- Free space move to approach pose
- Linear move to target pose

In order to simplify the scripting of the pick and place operation, helper functions are defined. These are then used in `test_bed_core_node` to perform the operation.

Explore function definitions

- In pick_and_place/include open trajopt_pick_and_place_constructor.h
- Look at each of public member functions. These are the functions you will complete. Understand each the inputs and purpose of each of them.

Complete Code

- Open trajopt_pick_and_place_constructor.cpp
- Find every line that begins with the comment “”Fill Code: ‘”” and read the description. Then, replace every instance of the comment “”ENTER CODE HERE”” with the appropriate line of code

```
/* Fill Code:
     .
     .
     .
*/
// ENTER CODE HERE:
```

You may find Sections 3.5 and 3.6 helpful in completing this exercise. Additionally, QT Creator’s autocomplete functionality can aid in finding the correct methods described in the comments.

Build Code and Run

The code can be run by using the same launch file as before. This time the robot should pathplan when the perception service completes

```
roslaunch pick_and_place pick_and_place.launch
```

Setting up a 3D sensor

In this exercise, we will setup an Intel RealSense camera to work with our robotic workcell. In this demo we will use the Intel RealSense D415 though a wide range of 3D sensors could be used to publish a ROS point cloud and allow for seamless integration with the rest of the robotic system. The RealSense ROS package can be found on the wiki.

Installing the RealSense SDK

Check your Ubutnu kernel and ensure that it is 4.4, 4.10, 4.13, or 4.15

```
uname -r
```

Register the server's public key

```
sudo apt-key adv --keyserver keys.gnupg.net --recv-key C8B3A55A6F3EFCDE || sudo apt-
↪key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-key C8B3A55A6F3EFCDE
```

Add the server to the list of repositories

```
sudo add-apt-repository "deb http://realsense-hw-public.s3.amazonaws.com/Debian/apt-
↪repo xenial main" -u
```

Install RealSense demos and utilities

```
sudo apt-get install librealsense2-dkms
sudo apt-get install librealsense2-utils
sudo apt-get install librealsense2-dev
sudo apt-get install librealsense2-dbg
```

Test the install by connecting the RealSense camera and running the viewer utility in the terminal.

```
realsense-viewer
```

Insure the the camera is being recognized as a USB 3.0+ device. If that is not the case, verify that the USB cable used is a 3.0 cable. It has also been found that for long usb cables (>3 m), using an external USB card helps reliability. Regardless, occasional power cycling (unplugging and replugging the USB) may be required. If problems arise during this process, see the librealsense documentation.

Installing ROS package

The ROS package to interface with the RealSense camera now needs to be cloned into your workspace. To do this, navigate into your ***/src*** directory and run the following.

```
git clone https://github.com/intel-ros/realsense.git -b development
```

Next, build your workspace.

```
catkin build
```

If errors occur, make sure that the RealSense SDK was correctly installed.

The calibration makes use of one other package. Install that now.

```
sudo apt install ros-kinetic-rgdb-launch
```

Run the demo launch file to verify that the ROS interface is working. You should see an RGBD image displayed in RVIZ. As usual, you may need to source your workspace again if you have added new packages. Do this with `source devel/setup.bash` from the root of your workspace.

```
roslaunch realsense2_camera demo_pointcloud.launch
```

Note: The Realsense ROS package only supports certain versions of the Realsense SDK. When in realsense-viewer make note of the release number, and make sure that the branch of the ROS package you are using is compatible. To pull a specific release (in this case 2.1.2), use

```
git clone https://github.com/intel-ros/realsense.git -b 2.1.2
```

Further, you may need to update the camera firmware version. To do that, see Intel's documentation. It is recommended that you use the latest production release.

Calibration

The General Idea

Calibrating a camera in your workspace typically happens in two steps:

1. Calibrate the "intrinsics", the camera sensor & lens, using something like ROS' camera calibration package.
2. Armed with the intrinsics, calibrate the "extrinsics", or the pose of the camera in your workcell.

In our case, the RealSense intrinsics have been calibrated at the factory. This leaves only the second step.

Terminology

- **Extrinsic Parameters**: "the extrinsic parameters define the position of the camera center and the camera's heading in world coordinates" [ref]. An extrinsic calibration thus tries to find WHERE your camera is relative to some frame of reference, usually the base of a robot or the wrist of a robot.
- **Intrinsic Parameters**: When talking about cameras, these parameters define *how* points in 3D space are projected into a camera image. They encompass internal properties of the camera sensor and lens such as focal length, image sensor format, and principal point. An intrinsic calibration tries to solve these
- **Rectified Image**: Real world cameras and their lenses are NOT perfectly described by the commonly used pinhole model of projection. The deviations from that model, called *distortions*, are estimated as part of *intrinsic calibration* and are used in software to produce an "undistorted" image called the *rectified image*. In our case, the RealSense driver will do this for us.

Setting up Calibration

The calibration for this exercise is based off of an AR Tag. The AR Tag can be found in ***pick_and_place_support/config***. Print it off then measure and make note of the actual, printed size of the black square.

Next, we need to place the AR tag at a known location relative to the robot. Since we are still simulating the robot, we will do this by aligning the AR tag with the hole pattern on the bottom of the workcell. Note that the holes are on a 2" hole spacing, and the measurements are taken from the center of the iiwa base. The default values are for the center of the AR tag to be placed 12" in front of the robot with positive x in the long dimension of the workcell.

Performing the Extrinsic Calibration

With the hardware set up, it's time to perform the calibration. The code has been written for you in pick_and_place_support/src/extrinsic_calibration.cpp.

Open `calibration.launch` in the pick_and_place_support package. Notice the first 5 arguments. `sim_robot` is a flag to set whether or not we will use the robot to locate the target or measure manually. `targ_x` is location of the target when performing the calibration. `marker_size` is the size of the black square measured previously in cm.

- Measure the location of the center of the AR tag in meters. Note that x is in the direction of the long dimension of the workcell and z is up with y defined to follow the right hand rule.
- If more than one camera is connected, the serial number of the camera to be calibrated must be provided. The serial numbers of all connected cameras can be found by using `rs-enumerate-devices`.
- Launch the calibration script filling the values for the target's location and size

```
roslaunch pick_and_place_support calibration.launch sim_robot:=true targ_x:=[fill_
↪value] targ_y:=[fill_value] targ_z:=[fill_value] marker_size:=[fill_value] serial_
↪no_camera:=[fill_value]
```

- Record the output. The script should show the camera as a TF floating in space. After 60 iterations, the program will pause and display the calibration result in the format `x y z roll pitch yaw`. When you are satisfied that the camera location appears to match reality, copy this pose.
- Update the camera location in pick_and_place/launch/pick_and_place.launch. Lines 38-43 publish the locations of the cameras for the system. Update these numbers with the values from calibration (they are in the same `x y z yaw pitch roll` format.

Running with real camera and simulated robot

With the camera calibrated, it is time to run the system with a simulated robot but real depth camera. Since our code has already been tested in simulation, this is quite easy. First, update the camera serial numbers in ***pick_and_place/launch/bringup_realsense.launch***. Then simply launch the same file as before but set the sim_sensor flag to false

```
roslaunch pick_and_place pick_and_place.launch sim_sensor:=false
```

Setting-up-a-robot.md

Next, we will move to using a real robot. For this tutorial, we will be using the KUKA iiwa 7kg collaborative robot. These instructions are primarily taken from the iiwa_stack wiki, and it is recommended that the user read this wiki before following these instructions. Further, as with any hardware integration task, it is likely that there will be aspects specific to your setup that may not be addressed here. As always, this is where the large ROS community knowledge base is here to help. Commonly, solutions to such problems can be found in ROS Answers or in the driver repo's issue tracker.

Introduction

The **KUKA LBR IIWA** redundant manipulator is programmed using the KUKA's Sunrise Workbench platform and its Java APIs.A **Sunrise project**, containing one or more **Robotic Application** that can be synchronized to the robot cabinet and executed from the **SmartPad**.

Within this software package, you will find a **Robotic Application** that can be used with the robot.It establishes a connection with an additional machine connected via Ethernet to the robot cabinet via ROS.The additional machine, having ROS installed, will be able to send and receive ROS messages to and from the aforementioned **Robotic Application**.We make use of ROS messages already available in a mint ROS distribution as well as custom ones, made to work with the KUKA arm (*iiwa_msgs*).A user is then able to manipulate the messages received from the robot and to send new ones as commands to it, simply within C++ or python code, taking leverage of all the ROS functionalities.

Note on computers

In this wiki, we will potentially deal with 3 computers. The first is the **Workbench** PC. It is used to load settings on the KUKA. It has a version Sunrise Workbench that is compabitle with your robot's firmware installed, and is running a Windows OS.The second computer is the **ROSCORE** computer. This will refer to the additional machine equipped with ROS. This is the computer (or partition) containing your ROS workspace. Finally there is the **CABINET** PC that is used to refer to anything included in robot cabinet.

Setup Guide

IMPORTANT : A version of **Sunrise Workbench** with the **Sunrise.Connectivity** module is required to use this stack. If you have something referring to "Smart Servoing", that is the correct module. Connectivity has been tested on Sunrise versions: 1.5, 1.7, 1.9, 1.11, 1.13, 1.14, and 1.15. If you don't have the Connectivity module, you might want to ask KUKA for it.

Whatever version you use, the Sunrise OS, Sunrise workbench, and Connectivity module must all be compatible.

Cabinet setup

Cabinet setup

A good part of this page is taken from Khansari's wiki. In case that one will go down one day, here it is.

Before we start, this is a good time to make a factory image of your cabinet. If you purchased the KUKA recovery stick, this will allow you to restore the cabinet in case something goes wrong. Unfortunately the recovery stick does not come with a factory image, so make one before you get started. To do that, connect a monitor and keyboard to the cabinet then simply turn off the cabinet, insert the usb, and reboot the cabinet. It will boot to the recovery menu. You will likely need to change the language using the Sprachauswahl button. Select create image and give it a name. When you're done shut down the cabinet, remove the stick, and reboot.

The Cabinet of an IIWA robot has multiple ethernet ports, unless otherwise specified, we are going to make use of the KONI interface.

NOTE: The KONI Ethernet port is on the left side of the front panel of the cabinet. There should be a label on top of it.

The KONI interface by default belongs to the Real Time OS on the Cabinet, and available basically only for FRI.To use it, we need to change the configuration of the Cabinet itself.

WARNING: Follow these instructions at your own risk. KUKA doesn't provide these instructions, but it's rather a home-brew solution. It may cause problems to your hardware (even if so far none is known).

Change the configuration of the KONI interface

1. **Log in into the Cabinet.**There are different methods to have access to the cabinet. The best for this job is to connect the cabinet to external monitor, mouse, and keyboard (USB and DVI ports on the bottom left) and start the cabinet without the SmartPad plugged in.The Windows7 Embedded should start up with the *KukaUser* already logged in, in case not you could search for the credentials of that user, google is your friend.
2. **Shutdown KRC**Right-click on the green icon in the system tray (bottom right, next to the clock) and click Stop KRC.
3. **Check the Network Interfaces**Open the list of the network interfaces: Start -> View network connections.At this point you should only have one interface: **Realtime OS Virtual Network Adapter**.Just remember it exists and **DO NOT EVER CHANGE ITS SETTINGS**.
4. **Open a terminal and type**`C:\KUKA\Hardware\Manager\KUKAHardwareManager.exe -assign OptionNIC -os WIN`This will assign the KONI interface to Windows.

Note: If you are not familiar with the German keyboard, you might want to change it by going to start->Region and Language -> Keyboards and language and selecting your favorite keyboard layout.

A COMMON ISSUE: It is been reported by several users that the above command does not work in some new versions of Sunrise OS, and gives the following error: "Assignment not possible while KS is running". To handle this issue

you need to remove KS from the windows auto startup. Do this by going to start -> msconfig.exe and disabling BoardPackage and KR C. Restart the PC and now KUKA software should not be loaded. Now apply the command. After that, put back the KS in the Windows auto startup. Restart Windows and you should be good to continue with the remaining steps.

1. **Install the network adapter driver (often not needed)**Open the Device Manager, if you don't have any network adapter marked in yellow then just skip this point.Else, You can find the driver for it in C:\KUKA\Hardware\Drivers.Sometimes you will find there two folders for different Ethernet controllers, check within the Device Manager which one you need.
2. **Reboot and open Start -> View network connections again**Change the settings (IP address) of the **new** Ethernet adapter to the one you want to use - within the Network adapter settings in Windows. Choose a **different** subnet than the one used by Sunrise Workbench, for example, we use IP: *160.69.69.69* and subnet mask: 255.255.0.0.

Alternatively, you could open `C:\Windows\System32\drivers\etc\hosts` as Administrator and add the hostname and IP of the ROS machine you are using. This will allow to get an IP in the subnet via DHCP.

Connect an Ethernet cable between the **KONI** interface and the Ethernet adapter on the **ROSCORE** side.

Further comments from users can be found in some issues: e.g. #65

NOTE: Due to this alteration, other applications using the KONI Ethernet Port of the Cabinet won't work anymore.For example, any application using the native KUKA's FRI library won't be functioning, as it uses the same Ethernet Port for communication.To revert the change execute:`C:\KUKA\Hardware\Manager\KUKAHardwareManager.exe -assign OptionNIC -os RTOS`this will assign back the KONI interface to the Real Time OS.

Roscore PC setup

The first 3 steps should already have been completed if you are following this tutorial. However, they are kept here for completeness.

1. **Install ROS KINETIC or INDIGO** (if not already there) as described at http://wiki.ros.org/kinetic/Installation/Ubuntu. (till section 1.7)It's also a good idea to install the python catkin tools`sudo apt-get install python-catkin-tools`
2. **Clone this repository to your workspace** (you can omit the first 3 commands if you already have one) :`mkdir ros_ws && cd ros_ws && mkdir srccatkin_init_workspacegit clone https://github.com/SalvoVirga/iiwa_stack.git src/iiwa_stack`
3. **Download dependences** :`rosdep install --from-paths src --ignore-src -r -y`
4. **Build the workspace** :`catkin build`
5. **Source workspace** :`source devel/setup.bash`
6. **Setup network** :Connect the KONI port of the cabinet to the ROS PC. Setup the network such that the ROS PC has a static IP address on the same subnet as the one used by the cabinet. For this example we are using subnet: 255.255.0.0 and IP: 160.69.69.100.

Open`gedit ~/.bashrc &`and append these two lines at the end`export ROS_IP=xxx.xxx.xxx.xxxexport ROS_MASTER_URI=http://$ROS_IP:11311`With *xxx.xxx.xxx.xxx* an IP address on the same subnet of the one you used in the **SUNRISE** setup. Also, configure the network interface - within your Operating System - you are going to use with the same IP address. For example, we use *160.69.69.100*.

1. **Open a terminal and ping the KONI port of the cabinet**.In **our case** : `ping 160.69.69.69`if you get an answer, everything is ready.

Sunrise OS Setup (on Workbench PC)

Next we will do some setup on the robot's Sunrise OS using the workbench PC (the Windows PC with sunrise workbench installed). The iiwa is somewhat unique in that the ROS node actually runs on the controller. Here we are going to use the Sunrise Workbench utility to load the necessary files on the controller. It is worth noting that this utility is also the only way to change the majority of the iiwa settings, so if you need to make changes to the safety config or other settings, this will be the way to do it.

Connect the Workbench PC

The workbench PC should be connected to the cabinet using the x66 ethernet port. Depending on your computer's setup, you may have to define a static IP as well.

Set up your Sunrise Project

1. Create a Sunrise project, or load an existing one from the controller (default IP: 172.31.1.147).
2. Open the StationSetup.cat file
3. Select the *Software* tab
4. Enable the Servo Motion packages, this is how that page should look like : ![station][stationsetup]
5. Save and apply changes

Now you will need to add `iiwa_stack` (more precisely the content of `iiwa_ros_java`) to the Sunrise project.

You could either:

- Copy iiwa_ros_java into the Sunrise Project: (changes will not be seen by git)
    1. Copy the content of *iiwa_stack/iiwa_ros_java/src* inside the *src* folder.
    2. Copy the folder *iiwa_stack/iiwa_ros_java/ROSJavaLib* into the root of the project.
    3. Inside Sunrise Workbench select all the files inside *ROSJavaLib*, right click and choose *Build Path* -> Add to Build Path... (you may have to refresh to see them)

or

- Create symlinks to iiwa_ros_java: (changes will be tracked by git!)
    1. Run the command prompt as an administrator (press windows key->type cmd->right click->run as an administrator).
    2. type `cd \path\to\sunrise\project`
    3. type `mklink /D ROSJavaLib ..\relative\path\to\iiwa_stack\iiwa_ros_java\ROSJavaLib`
    4. type `cd src`
    5. type `mklink /D de ..\relative\path\to\iiwa_stack\iiwa_ros_java\src\de`
    6. Inside Sunrise Workbench select all the files inside *ROSJavaLib*, right click and choose *Build Path* -> Add to Build Path...

IMPORTANT 6. Have you already set your **Safety Configuration**? If not, you should! **HERE** we show how we set up ours.7. Be sure the right **master_ip** is set in the *config.txt* file. See below.

Configuration File

From release 1.7.x, we introduced a **configuration file** that allows to easily set some parameters.In the **Java** project you just created, you should have a **config.txt** file.

There you can set :

- robot_name: name of your robot, by default *iiwa*. Read more about this HERE;
- master_ip: IP of the **ROSCORE** machine you just set HERE, for example *160.69.69.100*;
- master_port: port for ROS communication, also set HERE, by default is *11311*;
- ntp_with_host: turns on/off the ntptimeprovider from ROSJava to sync the two machines. The default value is false, note that you need an NTP provider on the external machine if you want to set it to true.

Once you properly fill this file, **everything is ready**. There should be no red sign or problem found by the IDE. If this is a new project? Then you first need to *Install* it (*Station Setup -> Installation*) and then synchronize it.

Execute trajectory on robot

With setup complete, you are ready to execute on the robot. First, switch the robot into auto mode by turning the key, selecting auto, and turning the key back.

Then select the ROSSmartServo program on the smart pad and press the run button.

Finally, simply run the same code from before, this time with sim_robot=false.

```
roslaunch pick_and_place pick_and_place.launch sim_sensor:=false
sim_robot:=false
```

The system should use a 3D scan of the environment to find the pick location, path plan the pick and place moves, then ask you if you would like to execute. Simply type y+enter to execute on the robot.

One thing to note is that a ROS node is actually running on the KUKA cabinet PC. This means that pausing the program on the smart pad (e.g. by manually jogging it) or restarting the launch file can cause problems. In order to get the program to reconnect to ROS master, you may need to restart the program by unselecting it and reselecting it in the program menu then rerunning it.

Setting up a Schunk Gripper

For this tutorial, we will use a Schunk WSG050 gripper. We will communicate with this gripper via Ethernet TCP/IP. However, the ROS driver hides most of the details of this interface, allowing us to control it without indepth knowledge of TCP/IP. Further, it is worth noting that there are at least 2 community supported ROS drivers that provide different interfaces. We will demonstrate only the first one.

- https://github.com/ipa320/ipa325_wsg50.git
- https://github.com/nalt/wsg50-ros-pkgS

Note that this tutorial is more advanced than some of the previous tutorials. While examples and guidelines will be given, the “solution” is not provided. Be sure to refer to the prior tutorials as well as the ROS wiki for more information. Additionally, searching any error messages is always a good way to utilize the vast ROS community knowledgebase.

Setup Network

- Connect to the gripper using a static IP address such as 192.168.1.2
- Open a browser and enter 192.168.1.20
- Settings → Network → change IP address and subnet mask to
  - IP: 160.69.69.20
  - subnet: 255.255.0.0

Clone the driver package into your workspace

Like many community ROS package, there is not a debian release for this driver. Therefore you will need to clone the source of the package into the src space of your workspace.

```
git clone https://github.com/ipa320/ipa325_wsg50.git
```

Create Example node

Edit Build Information

Next we will create a C++ example. First we need to add an executable to the CMakeLists.txt

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

Additionally, you will need to add a dependency on the ipa325_wsg50 by adding it to the appropriate places in the CMakeLists.txt

- find_package(catkin REQUIRED COMPONENTS [all_of your other packages] ipa325_wsg50)
- catkin_package(CATKIN_DEPENDS [all of your other packages] ipa325_wsg50)

Finally, you need to add the following line to your package.xml file

```
 <depend>ipa325_wsg50</depend>
```

Add gripper_interface_node.cpp

Now create the gripper_interface_node.cpp file in the location specified in the CMakeLists.txt

```
#include <ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include <ipa325_wsg50/WSG50HomingAction.h>
#include <ipa325_wsg50/WSG50GraspPartAction.h>
#include <ipa325_wsg50/WSG50ReleasePartAction.h>
#include <ipa325_wsg50/ackFastStop.h>

int main (int argc, char **argv)
{
  ros::init(argc, argv, "test_gripper");
```

(continues on next page)

(continued from previous page)

```
  ros::NodeHandle nh;

  // create the action client
  // true causes the client to spin its own thread
  actionlib::SimpleActionClient<ipa325_wsg50::WSG50HomingAction> home_ac(
↪"WSG50Gripper_Homing", true);
  actionlib::SimpleActionClient<ipa325_wsg50::WSG50GraspPartAction> grasp_ac(
↪"WSG50Gripper_GraspPartAction", true);
  actionlib::SimpleActionClient<ipa325_wsg50::WSG50ReleasePartAction> release_ac(
↪"WSG50Gripper_ReleasePartAction", true);
  ros::ServiceClient srv = nh.serviceClient<ipa325_wsg50::ackFastStop>("/
↪AcknowledgeFastStop");

  ROS_INFO("Waiting for action servers to start.");
  home_ac.waitForServer(); //will wait for infinite time
  grasp_ac.waitForServer();
  release_ac.waitForServer();
  ipa325_wsg50::ackFastStop ack;
  srv.call(ack);

  ROS_INFO("Homing gripper...");
  ipa325_wsg50::WSG50HomingGoal home_goal;
  home_goal.direction = true;   // True is in the out direction
  home_ac.sendGoal(home_goal);
  bool finished_before_timeout = home_ac.waitForResult(ros::Duration(10.0));

  if (finished_before_timeout)
  {
    actionlib::SimpleClientGoalState state = home_ac.getState();
    ROS_INFO("Homing finished: %s",state.toString().c_str());
  }
  else
    ROS_INFO("Homing did not finish before the time out.");

  ROS_INFO("Grasping Part...");
  ipa325_wsg50::WSG50GraspPartGoal grasp_goal;
  grasp_goal.width = 80;  // Part width in mm
  grasp_goal.speed = 20;  // Speed in mm/s
  grasp_ac.sendGoal(grasp_goal);
  finished_before_timeout = grasp_ac.waitForResult(ros::Duration(15.0));

  if (finished_before_timeout)
  {
    actionlib::SimpleClientGoalState state = grasp_ac.getState();
    ROS_INFO("Grasp finished: %s",state.toString().c_str());
  }
  else
    ROS_INFO("Grasp did not finish before the time out.");

  ROS_INFO("Releasing Part...");
```

Test Example

To test this example, first launch the launch file that was included with the ROS driver (`wsg50.launch`). Then start the C++ example. Verifiy that it is working - opening and closing the Schunk gripper.

You can now take this code and add it to `test_bed_core_node.cpp` at the appropriate times in order to pick and place the box. While some headers should go at the top, the majority of the code should go in the execution section at the bottom. Look for

```
// Put gripper code here
// End gripper code here
```

Setting up a Gripper - Suction Cup

For this tutorial, we will use a suction cup gripper. In order to control it, we need to toggle a solenoid with a GPIO module. We will use the Acromag 951ELN-4012, but the approach would be similar for a wide variety of Modbus controlled GPIO modules.

Note that this tutorial is more advanced than some of the previous tutorials. While examples and guidelines will be given, the "solution" is not provided. Be sure to refer to the prior tutorials as well as the ROS wiki for more information. Additionally, searching any error messages is always a good way to utilize the vast ROS community knowledgebase.

Setup Network

- Change the IP address
- Connect to the module using a static IP address such as 128.1.1.2
- Open a browser and enter 128.1.1.100
  - Username: User
  - Password: password00
- Go to network configuration and change the following
  - Static IP: 160.69.69.40
  - Subnet mask: 255.255.0.0
  - DNS Server: blank

Hardware setup

Important Note: These instructions are for a specific setup. Use them only as guidelines. Consult the documentation for your hardware before attempting

Wire the solenoid valve

The solenoid input needs to be wired to one of the digital outputs of the gpio module.

Wire the GPIO module.

This gpio module needs needs external power. Wire in a DC source between 15-36V to the DC+ and DC- ports of the gpio module.

Additionally, the digital outputs need a dc source wired in as well. Wire in a DC source between 5.5-35V to the EXC+ and RTN ports of gpio module

Make Modbus interface

The gpio module being used for this tutorial communicates using Modbus. While this tutorial will not cover the details of Modbus, we can use ROS to communicate with it without an indepth knowledge of Modbus.

Clone the Modbus package into your workspace

```
git clone https://github.com/HumaRobotics/modbus.git
```

Create Modbus Interface node

Next we need to create the node that will send the commands to the gpio module. This is simply modified from the example given in the package cloned above. To do this, create a python node in the pick_and_place/src directory (`touch my_node.py`). Make sure that it is executable (either using `chmod +x filename` or by right clicking on the file)

Next copy in the code below. Note that the registers are from the documentation for this module.

```
#!/usr/bin/env python
###############################################################################
# This software is graciously provided by HumaRobotics
# under the Simplified BSD License on
# github: git@www.humarobotics.com:baxter_tasker
# HumaRobotics is a trademark of Generation Robots.
# www.humarobotics.com

# Copyright (c) 2013, Generation Robots.
# All rights reserved.
# www.generationrobots.com
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:
#
# 1. Redistributions of source code must retain the above copyright notice,
#  this list of conditions and the following disclaimer.
#
# 2. Redistributions in binary form must reproduce the above copyright notice,
#  this list of conditions and the following disclaimer in the documentation
#  and/or other materials provided with the distribution.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
# THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
# PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
# BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
# CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
# INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
# CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
# ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
# THE POSSIBILITY OF SUCH DAMAGE.
#
# The views and conclusions contained in the software and documentation are
# those of the authors and should not be interpreted as representing official
```

(continues on next page)

(continued from previous page)

```
# policies, either expressed or implied, of the FreeBSD Project.

import rospy
from modbus.modbus_wrapper_client import ModbusWrapperClient
from std_msgs.msg import Int32MultiArray as HoldingRegister

# This is the register to control the 951EN-4012 Digital Out
NUM_REGISTERS = 1
ADDRESS_WRITE_START = 102

if __name__=="__main__":
    rospy.init_node("modbus_client")
    rospy.loginfo("""
    This file shows the usage of the Modbus Wrapper Client to write to a register.
    To see the read registers of the modbus server use: rostopic echo /modbus_wrapper/
↪input
    To see sent something to the modbus use a publisher on the topic /modbus_wrapper/
↪output
    """)
    host = "160.69.69.40"
    port = 502

    # setup modbus client
    modclient = ModbusWrapperClient(host,port=port,rate=50,reset_registers=False,sub_
↪topic="modbus_wrapper/output",pub_topic="modbus_wrapper/input")
    modclient.setWritingRegisters(ADDRESS_WRITE_START,NUM_REGISTERS)
    rospy.loginfo("Setup complete")

    # start listening to modbus and publish changes to the rostopic
    modclient.startListening()
    rospy.loginfo("Modbus listener started. Publish binary mask on modbus_wrapper/
↪output to set I/O")

    # We are now listening. If anything is sent to the topic, it will be sent via␣
↪Modbus to the register above.
    while not rospy.is_shutdown():
        rospy.sleep(0.1)

    # Stops the listener on the modbus when ROS shuts down
    modclient.stopListening()
```

Create C++ Example

Next we will create a C++ example. First we need to add an executable to the CMakeLists.txt

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

Now create the gpio_interface_node.cpp file in the location specified in the CMakeLists.txt

```
#include <ros/ros.h>
#include <std_msgs/Int32MultiArray.h>

int main (int argc, char **argv)
```

(continues on next page)

(continued from previous page)

```
{
  ros::init(argc, argv, "test_gpio");
  ros::NodeHandle nh;

  ros::Publisher modbus_pub = nh.advertise<std_msgs::Int32MultiArray>("modbus_wrapper/
↪output", 1000);

  std_msgs::Int32MultiArray msg;
  while( ros::ok())
  {
    // Set all IO to active (high) 0b111111
    std::vector<int> all_active = {63};
    msg.data = all_active;
    modbus_pub.publish(msg);
    ros::Duration(1.0).sleep();

    // Set all IO to inactive (low) 0b000000
    std::vector<int> all_inactive = {0};
    msg.data = all_inactive;
    modbus_pub.publish(msg);
    ros::Duration(1.0).sleep();

    // Set some IO active - some inactive 0b000111
    std::vector<int> half_active = {7};
    msg.data = half_active;
    modbus_pub.publish(msg);
    ros::Duration(1.0).sleep();
  }

  //exit
  return 0;
}
```

Test Example

To test this example, first start a roscore and the Python modbus interface node. Then start the C++ example. Verifiy that it is working - switching the IO on and off every second. If the solenoid is wired correctly, you will likely be able to hear it triggering.

You can now take this code and add it to `test_bed_core_node.cpp` at the appropriate times in order to pick and place the box. While some headers should go at the top, the majority of the code should go in the execution section at the bottom. Look for

```
// Put gripper code here
// End gripper code here
```

CHAPTER 4

Advanced Topics

4.1 Session 5 - Path Planning and Building a Perception Pipeline

`Slides`

4.1.1 Advanced Descartes Path Planning

> In this exercise, we will use advanced features with Descartes to solve a complex path for part being held by the robot which gets processed by a stationary tool.

Motivation

MoveIt! is a framework meant primarily for performing "free-space" motion where the objective is to move a robot from point A to point B and you don't particularly care about how that gets done. These types of problems are only a subset of frequently performed tasks. Imagine any manufacturing ''process" like welding or painting. You very much care about where that tool is pointing the entire time the robot is at work.

This tutorial introduces you to Descartes, a ''Cartesian" motion planner meant for moving a robot along some process path. It's only one of a number of ways to solve this kind of problem, but it's got some neat properties:

- It's deterministic and globally optimum (to a certain search resolution).
- It can search redundant degrees of freedom in your problem (say you have 7 robot joints or you have a process where the tool's Z-axis rotation doesn't matter).

Reference Example

Descartes Tutorial

Further Information and Resources

Descartes Wiki

APIs:

- descartes_core::PathPlannerBase
- descartes_planner::DensePlanner
- descartes_planner::SparsePlanner

Scan-N-Plan Application: Problem Statement

In this exercise, you will add two new nodes, two xacro, and config file to your Scan-N-Plan application, that:

1. Takes the config file `puzzle_bent.csv` and creates a descartes trajectory where the part is held by the robot and manipulated around a stationary tool.
2. Produces a joint trajectory that commands the robot to trace the perimeter of the marker (as if it is dispensing adhesive).

Scan-N-Plan Application: Guidance

In the interest of time, we've included several files:

1. The first is a template node `adv_descartes_node.cpp` where most of the exercise is spent creating the complicated trajectory for deburring a complicated part.
2. The second node, `adv_myworkcell_node.cpp`, is a duplicate of the `myworkcell_node.cpp` that has been updated to call the `adv_plan_path` service provided by `adv_descartes_node.cpp`.
3. The config file `puzzle_bent.csv` which contains the path relative to the part coordinate system.
4. The two xacro files `puzzle_mount.xacro` and `grinder.xacro` which are used to update the urdf/xacro `workcell.xacro` file.

Left to you are the details of:

1. Updating the workcell.xacro file to include the two new xacro files.
2. Updating the moveit_config package to define a new Planning Group for this exercise, including the new end-effector links.
3. Defining a series of Cartesian poses that comprise a robot "path".
4. Translating those paths into something Descartes can understand.

Setup workspace

1. This exercise uses the same workspace from the Basic Training course. If you don't have this workspace (completed through Exercise 4.1), copy the completed reference code and pull in other required dependencies as shown below. Otherwise move to the next step.

```
mkdir ~/catkin_ws
cd ~/catkin_ws
cp -r ~/industrial_training/exercises/4.1/src .
cd src
git clone https://github.com/jmeyer1292/fake_ar_publisher.git
```

(continues on next page)

(continued from previous page)

```
git clone -b melodic-devel https://github.com/ros-industrial-consortium/descartes.
↪git
git clone -b kinetic-devel https://github.com/ros-industrial/universal_robot.git
```

2. Copy over the `adv_descartes_node_unfinished.cpp` into your core package's src/ folder and rename it `adv_descartes_node.cpp`.

```
cp ~/industrial_training/exercises/5.0/src/adv_descartes_node_unfinished.cpp␣
↪myworkcell_core/src/adv_descartes_node.cpp
```

3. Create rules in the `myworkcell_core` package's `CMakeLists.txt` to build a new node called `adv_descartes_node`. As in previous exercises, add these lines near similar lines in the template file (not as a block as shown below).

```
add_executable(adv_descartes_node src/adv_descartes_node.cpp)
add_dependencies(adv_descartes_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_
↪EXPORTED_TARGETS})
target_link_libraries(adv_descartes_node ${catkin_LIBRARIES})
```

4. Copy over the `adv_myworkcell_node.cpp` into your core package's src/ folder.

```
cp ~/industrial_training/exercises/5.0/src/myworkcell_core/src/adv_myworkcell_
↪node.cpp myworkcell_core/src/
```

5. Create rules in the `myworkcell_core` package's `CMakeLists.txt` to build a new node called `adv_myworkcell_node`. As in previous exercises, add these lines near similar lines in the template file (not as a block as shown below).

```
add_executable(adv_myworkcell_node src/adv_myworkcell_node.cpp)
add_dependencies(adv_myworkcell_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_
↪EXPORTED_TARGETS})
target_link_libraries(adv_myworkcell_node ${catkin_LIBRARIES})
```

6. Copy over the necessesary config file:

```
mkdir ~/catkin_ws/src/myworkcell_core/config
cp ~/industrial_training/exercises/5.0/src/myworkcell_core/config/puzzle_bent.csv␣
↪myworkcell_core/config/
cp ~/industrial_training/exercises/5.0/src/myworkcell_support/urdf/grinder.xacro␣
↪myworkcell_support/urdf/
cp ~/industrial_training/exercises/5.0/src/myworkcell_support/urdf/puzzle_mount.
↪xacro myworkcell_support/urdf/
mkdir ~/catkin_ws/src/myworkcell_support/meshes
cp ~/industrial_training/exercises/5.0/src/myworkcell_support/meshes/* myworkcell_
↪support/meshes/
```

7. Add new package dependencies:
   - Add `tf_conversions` to `CMakeLists.txt` (2 places) and `package.xml` (1 place)

Update your workcell.xacro file.

1. Add two `<include>` tags for the new `grinder.xacro` and `puzzle_mount.xacro` files.
2. Attach the grinder to the **world** link with the following offset:

```
<origin xyz="0.0 -0.4 0.6" rpy="0 3.14159 0"/>
```

- Look in the `grinder.xacro` file to locate the appropriate `<child_link>` name.
- Copy one of the other `<joint>` tag definitions and modify as appropriate.

3. Attach the puzzle mount to the robot's **tool0** frame with the following offset:

```
<origin xyz="0 0 0" rpy="1.5708 0 0"/>
```

- Look in the `puzzle_mount.xacro` file to locate the appropriate `<child_link>` name. You may need to study its various `<link>` and `<joint>` definitions to find the root link of this part.
- The `tool0` frame is standardized across most ROS-I URDFs to be the robot's end-effector mounting flange.

4. Launch the demo.launch file within your moveit_config package to verify the workcell. There should be a grinder sticking out of the table and a puzzle-shaped part attached to the robot.

```
roslaunch myworkcell_moveit_config demo.launch
```

Add new planning group to your moveit_config package.

1. Re-run the MoveIt! Setup Assistant and create a new Planning Group named **puzzle**. Define the kinematic chain to extend from the **base_link** to the new **part** link.

```
roslaunch myworkcell_moveit_config setup_assistant.launch
```

- *Note: Since you added geometry, you should also regenerate the allowed collision matrix.*

Complete Advanced Descartes Node

1. First, the function `makePuzzleToolPoses()` needs to be completed. The file path for **puzzle_bent.csv** is needed. For portability, don't hardcode the full path. Please use the ROS tool `ros::package::getPath()` to retrieve the root path of the relevant package.
   - reference getPath() API
2. Next, the function `makeDescartesTrajectory()` needs to be completed. The transform between **world** and **grinder_frame** needs to be found. Also Each point needs to have the orientation tolerance set for the z-axis to +/- PI;
   - reference lookupTransform() API
   - reference tf::conversions namespace
   - reference TolerancedFrame definition
   - reference OrientationTolerance definition

Update the setup.launch file.

1. Update the file to take a boolean argument named **adv** so that either the basic or advanced descartes node can be launched. Use `<if>` and `<unless>` modifiers to control which node is launched.
   - reference roslaunch XML wiki

Test Full Application

1. Run the new setup file, then your main advanced workcell node:

```
roslaunch myworkcell_support setup.launch adv:=true
rosrun myworkcell_core adv_myworkcell_node
```

   - Descartes can take **several minutes** to plan this complex path, so be patient.
   - It's difficult to see what's happening with the rviz planning-loop animation always running. Disable this loop animation in rviz (Displays -> Planned Path -> Loop Animation) before running `adv_myworkcell_node`.

4.1.2 Building a Perception Pipeline

In this exercise, we will fill in the appropriate pieces of code to build a perception pipeline. The end goal will be to broadcast a transform with the pose information of the object of interest.

Prepare New Workspace:

We will create a new catkin workspace, since this exercise does not overlap with the previous exercises.

1. Disable automatic sourcing of your previous catkin workspace:
   1. `gedit ~/.bashrc`
   2. comment out (#) the last line, sourcing your `~/catkin_ws/devel/setup.bash`

   Note: This means you'll need to manually source the setup file from your new catkin workspace in each new terminal window.

   1. Close gedit and source ROS into your environment

```
source /opt/ros/melodic/setup.bash
```

2. Copy the template workspace layout and files:

```
cp -r ~/industrial_training/exercises/5.1/template_ws ~/perception_ws
cd ~/perception_ws/
```

1. Initialize and Build this new workspace

```
catkin init
catkin build
```

2. Source the workspace

```
source ~/perception_ws/devel/setup.bash
```

1. Copy the PointCloud file from prior Exercise 4.2 to your home directory (~):

```
cp ~/industrial_training/exercises/4.2/table.pcd ~
```

2. Import the new workspace into your QTCreator IDE:

   - In QTCreator: *File -> New File or Project -> Other Project -> ROS Workspace -> ~/perception_ws*

Intro (Review Existing Code)

Most of the infrastructure for a ros node has already been completed for you; the focus of this exercise is the perception algorithms/pipleline. The *CMakelists.txt* and *package.xml* are complete and an executable has been provided. You could run the executable as is, but you would get errors. At this time we will explore the source code that has been provided - browse the provided *perception_node.cpp* file. The following are highlights of what is included.

1. Headers:
    - You will have to uncomment the PCL related headers as you go
2. int main():
    - The `main` function has been provided along with a while loop within the main function
3. ROS initialization:
    - Both `ros::init` and `ros::NodeHandle` have been called/initialized. Additionally there is a private nodehandle to use if you need to get parameters from a launch file within the node's namespace.
4. Set up parameters:
    - Currently there are three string parameters included in the example: the world frame, the camera frame and the topic being published by the camera. It would be easy to write up a few `nh.getParam` lines which would read these parameters in from a launch file. If you have the time, you should set this up because there will be many parameters for the pcl methods that would be better adjusted via a launch file than hardcoded.
5. Set up publishers:
    - Two publishers have been set up to publish ros messages for point clouds. It is often useful to visualize your results when working with image or point cloud processing.
6. Listen for PointCloud2 (within while loop):
    - Typically one would listen for a ros message using the ros subscribe method with a callback function, as done here. However it is often useful to do this outside of a callback function, so we show an example of listening for a message using `ros::topic::waitForMessage`.
7. Transform PointCloud2 (within while loop):
    - While we could work in the camera frame, things are more understandable/useful if we are looking at the points of a point cloud in an xyz space that makes more sense with our environment. In this case we are transforming the points from the camera frame to a world frame.
8. Convert PointCloud2 (ROS to PCL) (within while loop)
9. Convert PointCloud2 (PCL to ROS) and publish (within while loop):
    - This step is not necessary, but visualizing point cloud processing results is often useful, so conversion back into a ROS type and creating the ROS message for publishing is done for you.

So it seems that a lot has been done! Should be easy to finish up. All you need to do is fill in the middle section.

Primary Task: Filling in the blanks

The task of filling in the middle section containing the perception algorithms is an iterative process, so each step has been broken up into its own sub-task.

Implement Voxel Filter

1. Uncomment the *voxel_grid* include header, near the top of the file.
2. Change code:

   The first step in most point cloud processing pipelines is the voxel filter. This filter not only helps to downsample your points, but also eliminates any NAN values so that any further filtering or processing is done on real values. See PCL Voxel Filter Tutorial for hints, otherwise you can copy the below code snippet.

   Within `perception_node.cpp`, find section

```
/* ========================================
 * Fill Code: VOXEL GRID
 * ========================================*/
```

   Copy and paste the following beneath that banner.

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_ptr (new pcl::PointCloud<pcl::PointXYZ>␣
↪(cloud));
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_voxel_filtered (new pcl::PointCloud
↪<pcl::PointXYZ> ());
pcl::VoxelGrid<pcl::PointXYZ> voxel_filter;
voxel_filter.setInputCloud (cloud_ptr);
voxel_filter.setLeafSize (float(0.002), float(0.002), float(0.002));
voxel_filter.filter (*cloud_voxel_filtered);
```

3. Update Publisher Within `perception_node.cpp`, find section

```
/* ========================================
 * CONVERT POINTCLOUD PCL->ROS
 * PUBLISH CLOUD
 * Fill Code: UPDATE AS NECESSARY
 * ========================================*/
```

   Uncomment `pcl::toROSMsg`, and replace `*cloud_ptr` with `*cloud_voxel_filtered`

   After each new update, we'll be swapping out which point-cloud is published for rviz viewing

   Note: If you have the time/patience, I would suggest creating a ros publisher for each type of filter. It is often useful to view the results of multiple filters at once in Rviz and just toggle different clouds.

4. Compile

```
catkin build
```

Viewing Results

1. Run the (currently small) perception pipeline. Note: In rviz change the global frame to **kinect_link**.

```
cd ~
roscore
rosrun tf2_ros static_transform_publisher 0 0 0 0 0 0 world_frame kinect_link
rosrun pcl_ros pcd_to_pointcloud table.pcd 0.1 _frame_id:=kinect_link cloud_
↪pcd:=kinect/depth_registered/points
```

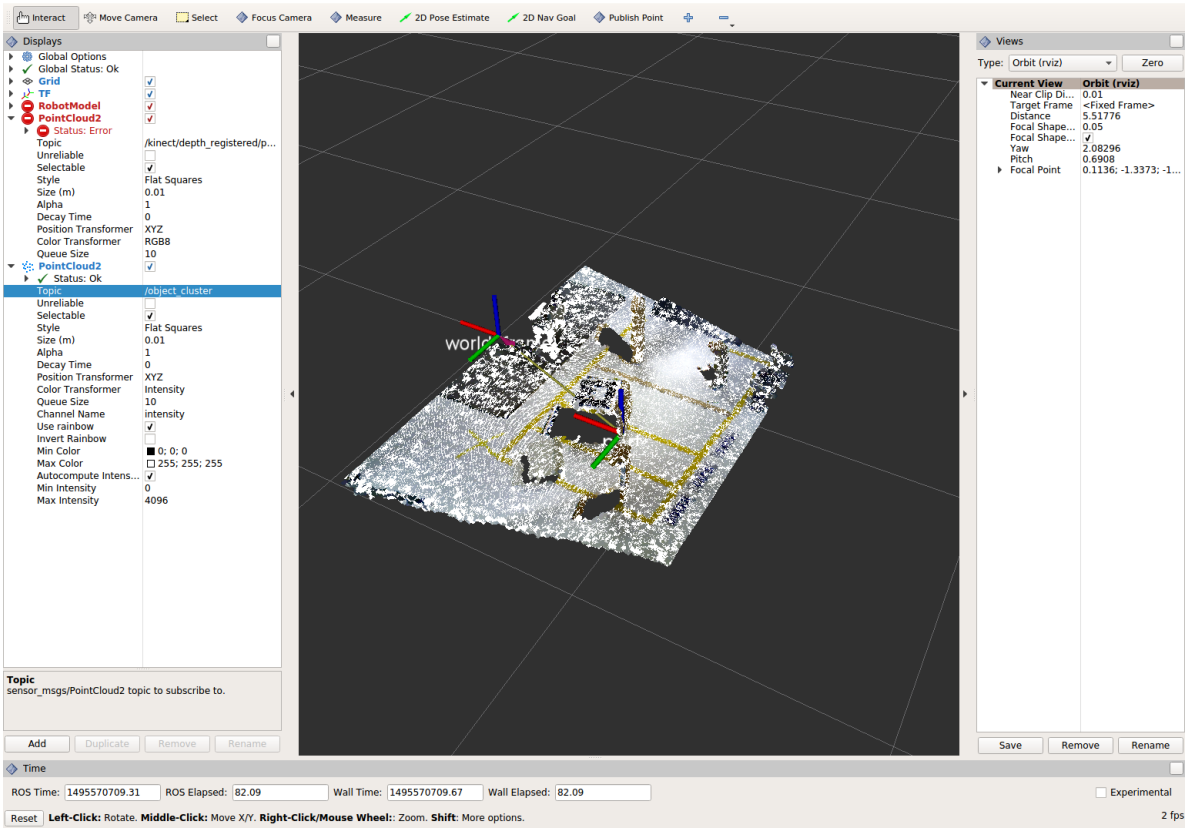
(continues on next page)

(continued from previous page)

```
rosrun rviz rviz
rosrun lesson_perception perception_node
```

2. View results

   Within Rviz, add a *PointCloud2* Display subscribed to the topic "object_cluster". What you see will be the results of the voxel filter overlaid on the original point cloud (assuming you have completed exercise 4.2 and saved a new default config or saved a config for that exercise).



3. When you are done viewing the results, try changing the voxel filter size from 0.002 to 0.100 and view the results again. Reset the filter to 0.002 when done.

   - To see the results of this change, use Ctrl+C to kill the perception node, re-build, and re-run the perception node.

Note: You do not need to stop any of the other nodes (rviz, ros, etc).

---

1. When you are satisfied with the voxel filter, use Ctrl+C to stop the perception node.

Implement Pass-through Filters

1. As before, uncomment the PassThrough filter include-header near the top of the file.
2. Change code:

   The next set of useful filtering to get the region of interest, is a series of pass-through filters. These filters crop your point cloud down to a volume of space (if you use x y and z filter). At this point you should apply a series

of pass-through filters, one for each the x, y, and z directions. See PCL Pass-Through Filter Tutorial for hints, or use code below.

Within perception_node.cpp, find section

```
/* ========================================
 * Fill Code: PASSTHROUGH FILTER(S)
 * ========================================*/
```

Copy and paste the following beneath that banner.

```
pcl::PointCloud<pcl::PointXYZ> xf_cloud, yf_cloud, zf_cloud;
pcl::PassThrough<pcl::PointXYZ> pass_x;
pass_x.setInputCloud(cloud_voxel_filtered);
pass_x.setFilterFieldName("x");
pass_x.setFilterLimits(-1.0,1.0);
pass_x.filter(xf_cloud);

pcl::PointCloud<pcl::PointXYZ>::Ptr xf_cloud_ptr(new pcl::PointCloud
↪<pcl::PointXYZ>(xf_cloud));
pcl::PassThrough<pcl::PointXYZ> pass_y;
pass_y.setInputCloud(xf_cloud_ptr);
pass_y.setFilterFieldName("y");
pass_y.setFilterLimits(-1.0, 1.0);
pass_y.filter(yf_cloud);

pcl::PointCloud<pcl::PointXYZ>::Ptr yf_cloud_ptr(new pcl::PointCloud
↪<pcl::PointXYZ>(yf_cloud));
pcl::PassThrough<pcl::PointXYZ> pass_z;
pass_z.setInputCloud(yf_cloud_ptr);
pass_z.setFilterFieldName("z");
pass_z.setFilterLimits(-1.0, 1.0);
pass_z.filter(zf_cloud);
```

You can change the filter limit values to see different results.

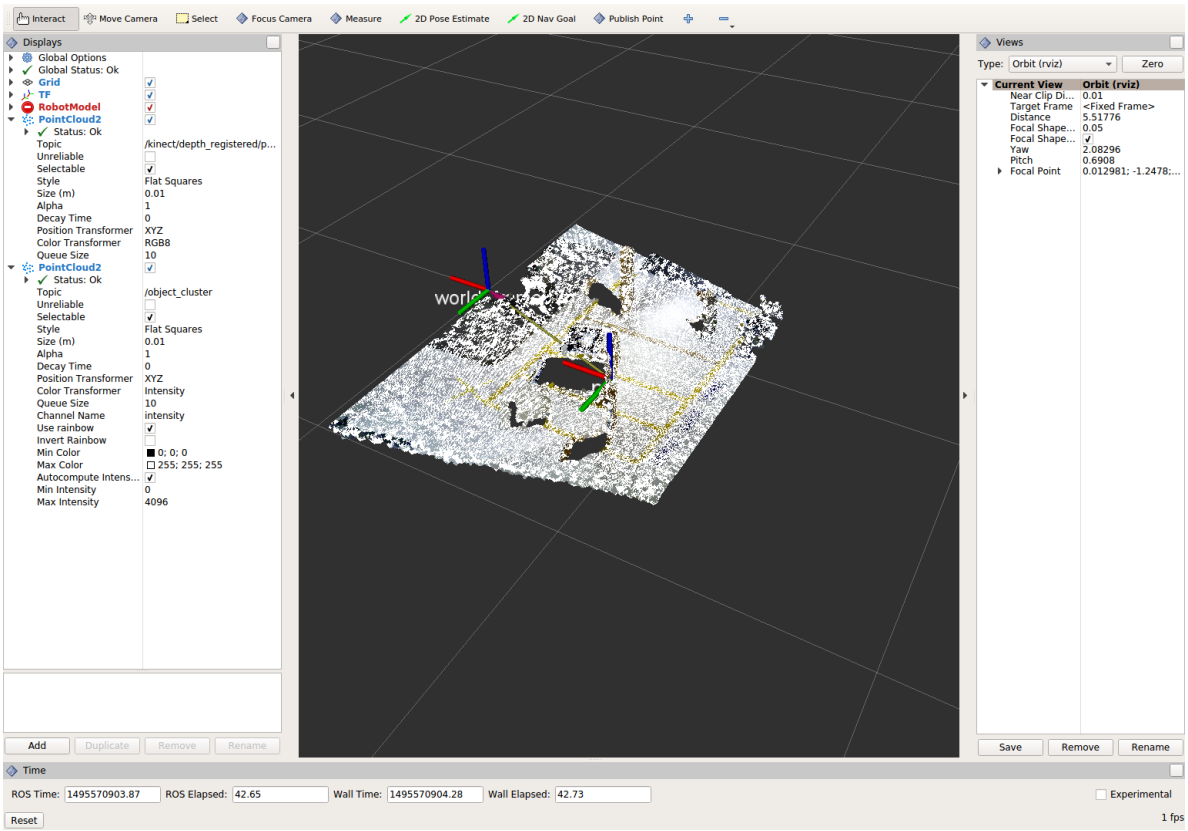
3. Find the `pcl::toROSMsg` call where the `pc2_cloud` is populated. This is the point cloud that is published to RViz display. Replace the current cloud (`*cloud_voxel_filter`) with the final Passthrough Filter result (`zf_cloud`).

4. Compile and run

```
catkin build
rosrun lesson_perception perception_node
```

5. View results

   Within Rviz, compare PointCloud2 displays based on the `/kinect/depth_registered/points` (original camera data) and `object_cluster` (latest processing step) topics. Part of the original point cloud has been "clipped" out of the latest processing result.



Note: Try modifying the X/Y/Z FilterLimits (e.g. +/- 0.5), re-build, and re-run. Observe the effects in rviz. When complete, reset the limite to +/- 1.0.

1. When you are satisfied with the pass-through filter results, press Ctrl+C to kill the node. There is no need to close or kill the other terminals/nodes.

Plane Segmentation

1. Change code

   This method is one of the most useful for any application where the object is on a flat surface. In order to isolate the objects on a table, you perform a plane fit to the points, which finds the points which comprise the table, and then subtract those points so that you are left with only points corresponding to the object(s) above the table. This is the most complicated PCL method we will be using and it is actually a combination of two: the RANSAC segmentation model, and the extract indices tool. An in depth example can be found on the PCL Plane Model Segmentation Tutorial; otherwise you can copy the below code snippet.

   Within perception_node.cpp, find section:

```
/* ========================================
 * Fill Code: PLANE SEGEMENTATION
 * ========================================*/
```

   Copy and paste the following beneath that banner.

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cropped_cloud(new pcl::PointCloud
↪<pcl::PointXYZ>(zf_cloud));
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_f (new pcl::PointCloud<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud
↪<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new pcl::PointCloud
↪<pcl::PointXYZ> ());
// Create the segmentation object for the planar model and set all the parameters
pcl::SACSegmentation<pcl::PointXYZ> seg;
pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
seg.setOptimizeCoefficients (true);
seg.setModelType (pcl::SACMODEL_PLANE);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setMaxIterations (200);
seg.setDistanceThreshold (0.004);
// Segment the largest planar component from the cropped cloud
seg.setInputCloud (cropped_cloud);
seg.segment (*inliers, *coefficients);
if (inliers->indices.size () == 0)
{
  ROS_WARN_STREAM ("Could not estimate a planar model for the given dataset.") ;
  //break;
}
```

Once you have the inliers (points which fit the plane model), then you can extract the indices within the point-cloud data structure of the points which make up the plane.

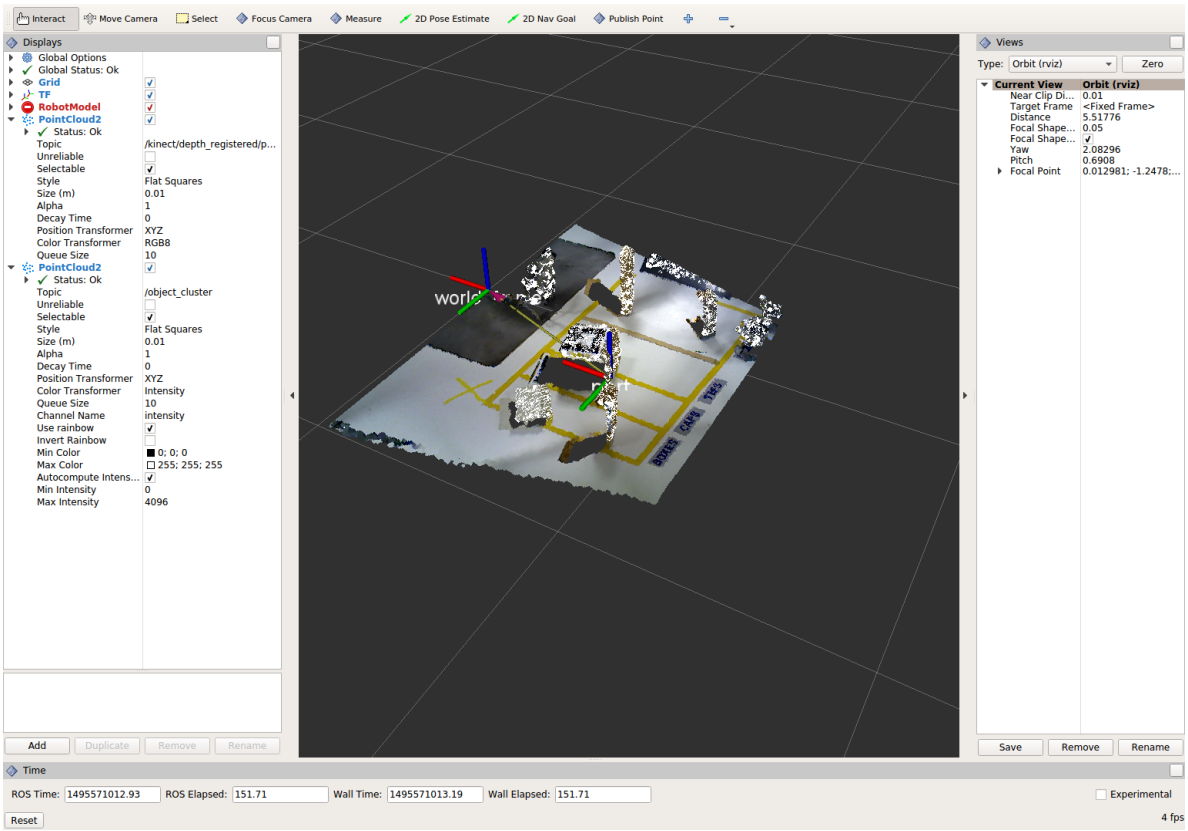
```
// Extract the planar inliers from the input cloud
pcl::ExtractIndices<pcl::PointXYZ> extract;
extract.setInputCloud (cropped_cloud);
extract.setIndices(inliers);
extract.setNegative (false);

// Get the points associated with the planar surface
extract.filter (*cloud_plane);
ROS_INFO_STREAM("PointCloud representing the planar component: " << cloud_plane->
↪points.size () << " data points." );
```

Then of course you can subtract or filter out these points from the cloud to get only points above the plane.

```
// Remove the planar inliers, extract the rest
extract.setNegative (true);
extract.filter (*cloud_f);
```

2. Find the `pcl::toROSMsg` call where the `pc2_cloud` is populated. This is the point cloud that is published to RViz display. Replace the current cloud (`zf_cloud`) with the plane-fit outliers result (`*cloud_f`).
3. **Compile and run, as in previous steps.** Did you forget to uncomment the new headers used in this step?
4. Evaluate Results

   Within Rviz, compare PointCloud2 displays based on the `/kinect/depth_registered/points` (original camera data) and `object_cluster` (latest processing step) topics. Only points lying above the table plane remain in the latest processing result.



5. When you are done viewing the results you can go back and change the"setMaxIterations" and "setDistanceThreshold" values to control how tightly the plane-fit classifies data as inliers/outliers, and view the results again. Try using values of `MaxIterations=100` and `DistanceThreshold=0.010`

6. When you are satisfied with the plane segmentation results, use Ctrl+C to kill the node. There is no need to close or kill the other terminals/nodes.

Euclidean Cluster Extraction (optional, but recommended)

1. Change code

   This method is useful for any application where there are multiple objects. This is also a complicated PCL method. An in depth example can be found on the PCL Euclidean Cluster Extration Tutorial.

   Within perception_node.cpp, find section

```
/* ========================================
 * Fill Code: EUCLIDEAN CLUSTER EXTRACTION (OPTIONAL/RECOMMENDED)
 * ========================================*/
```

   Follow along with the PCL tutorial, insert code in this section.

   Copy and paste the following beneath the banner.

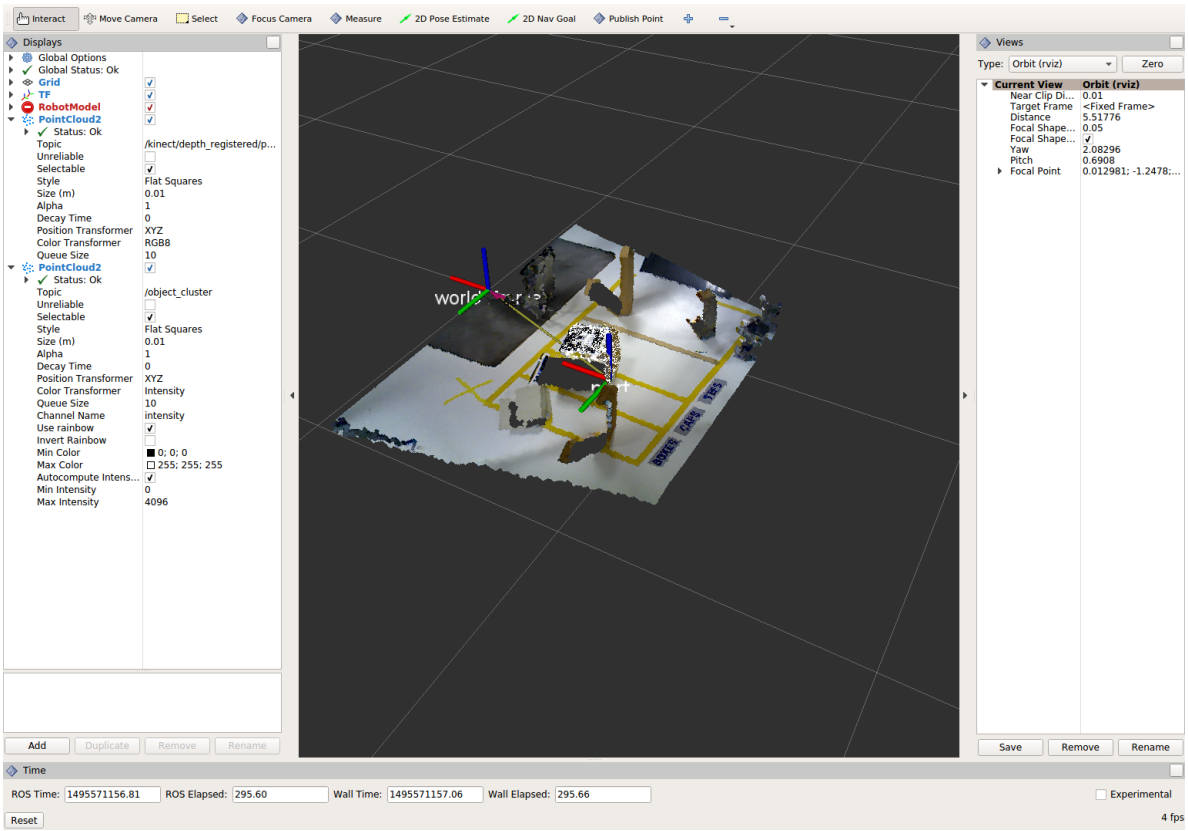
```
// Creating the KdTree object for the search method of the extraction
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree
↪<pcl::PointXYZ>);
*cloud_filtered = *cloud_f;
tree->setInputCloud (cloud_filtered);
```

(continues on next page)

(continued from previous page)

```
std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
ec.setClusterTolerance (0.01); // 2cm
ec.setMinClusterSize (300);
ec.setMaxClusterSize (10000);
ec.setSearchMethod (tree);
ec.setInputCloud (cloud_filtered);
ec.extract (cluster_indices);

std::vector<sensor_msgs::PointCloud2::Ptr> pc2_clusters;
std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr > clusters;
for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin ();
↪ it != cluster_indices.end (); ++it)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new pcl::PointCloud
↪<pcl::PointXYZ>);
  for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it->
↪indices.end (); pit++)
    cloud_cluster->points.push_back(cloud_filtered->points[*pit]);
  cloud_cluster->width = cloud_cluster->points.size ();
  cloud_cluster->height = 1;
  cloud_cluster->is_dense = true;
  std::cout << "Cluster has " << cloud_cluster->points.size() << " points.\n";
  clusters.push_back(cloud_cluster);
  sensor_msgs::PointCloud2::Ptr tempROSMsg(new sensor_msgs::PointCloud2);
  pcl::toROSMsg(*cloud_cluster, *tempROSMsg);
  pc2_clusters.push_back(tempROSMsg);
}
```

2. Find the `pcl::toROSMsg` call where the `pc2_cloud` is populated. This is the point cloud that is published to RViz display. Replace the current cloud (`*cloud_f`) with the largest cluster (`*(clusters.at(0))`).
3. Compile and run, as in previous steps.
4. View results in rviz. Experiment with `setClusterTolerance`, `setMinClusterSize`, and `setMaxClusterSize` parameters, observing their effects in rviz.



5. When you are satisfied with the cluster extraction results, use Ctrl+C to kill the node. There is no need to close or kill the other terminals/nodes.

Create a CropBox Filter

1. Change code

   This method is similar to the pass-through filter from Sub-Task 2, but instead of using three pass-through filters in series, you can use one CropBox filter. Documentation on the CropBox filter and necessary header file can be found here.

   Within perception_node.cpp, find section

```
/* ========================================
 * Fill Code: CROPBOX (OPTIONAL)
 * ========================================*/
```

   This CropBox filter should replace your passthrough filters, you may delete or comment the passthrough filters. There is no PCL tutorial to guide you, only the PCL documentation at the link above. The general setup will be the same (set the output, declare instance of filter, set input, set parameters, and filter).

   Set the output cloud:

```
pcl::PointCloud<pcl::PointXYZ> xyz_filtered_cloud;
```

   Declare instance of filter:

```
pcl::CropBox<pcl::PointXYZ> crop;
```

Set input:

```
crop.setInputCloud(cloud_voxel_filtered);
```

Set parameters - looking at documentation, CropBox takes an Eigen Vector4f as inputs for max and min values:

```
Eigen::Vector4f min_point = Eigen::Vector4f(-1.0, -1.0, -1.0, 0);
Eigen::Vector4f max_point = Eigen::Vector4f(1.0, 1.0, 1.0, 0);
crop.setMin(min_point);
crop.setMax(max_point);
```

Filter:

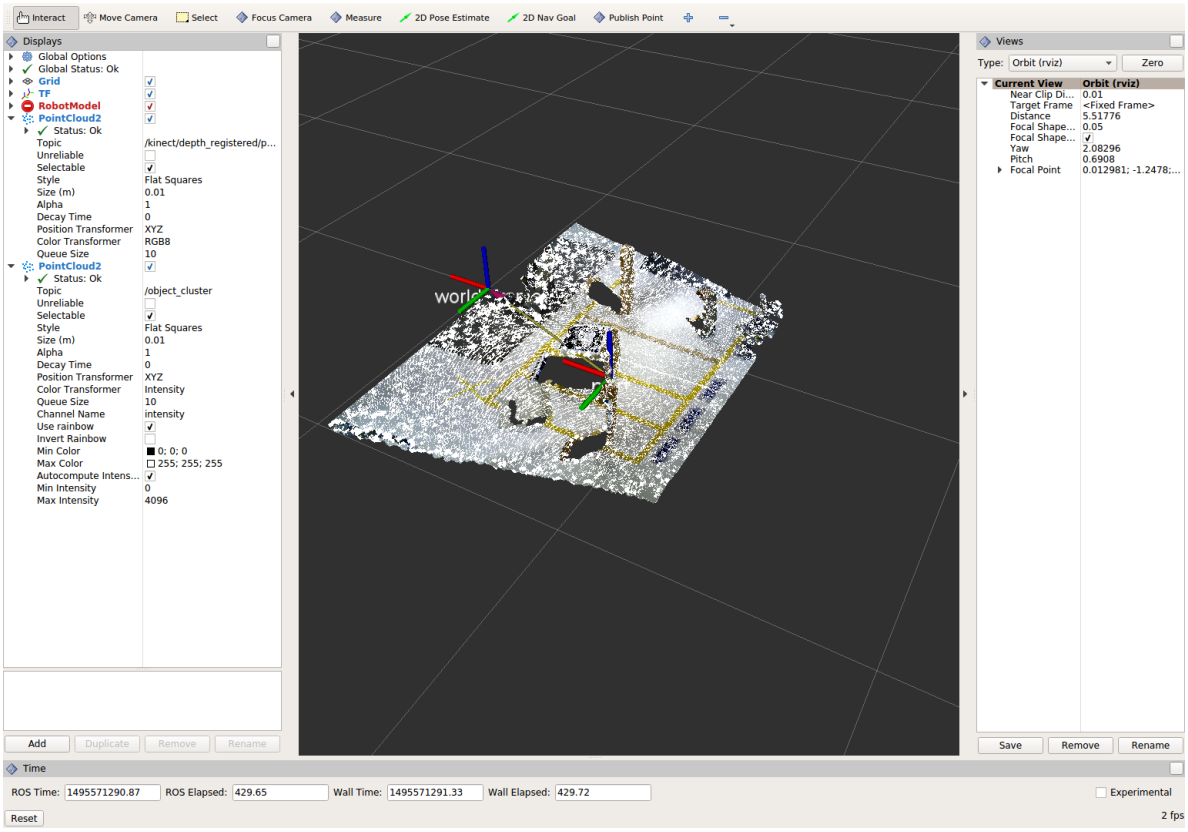
```
crop.filter(xyz_filtered_cloud);
```

If you delete or comment the passthrough filters and have already written the plane segmentation code, then make sure you update the name of the cloud you are passing into the plane segmentation. Replace zf_cloud with xyz_filtered_cloud:

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cropped_cloud(new pcl::PointCloud
↪<pcl::PointXYZ>(xyz_filtered_cloud));
```

2. Find the `pcl::toROSMsg` call where the `pc2_cloud` is populated. This is the point cloud that is published to RViz display. Replace the current cloud with the new filtered results (`xyz_filtered_cloud`).
3. Compile and run, as in previous steps

   The following image of the CropBox filter in use will closely resemble the Plane Segmentation filter image.



Create a Statistical Outlier Removal

1. Change code

   This method does not necessarily add complexity or information to our end result, but it is often useful. A tutorial can be found here.

   Within perception_node.cpp, find section

```
/* ========================================
 * Fill Code: STATISTICAL OUTLIER REMOVAL (OPTIONAL)
 * ========================================*/
```

   The general setup will be the same (set the output, declare instance of filter, set input, set parameters, and filter).

   Set the output cloud:

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cluster_cloud_ptr= clusters.at(0);
pcl::PointCloud<pcl::PointXYZ>::Ptr sor_cloud_filtered(new pcl::PointCloud
↪<pcl::PointXYZ>);
```

   Declare instance of filter:

```
pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;
```

   Set input:

```
sor.setInputCloud (cluster_cloud_ptr);
```

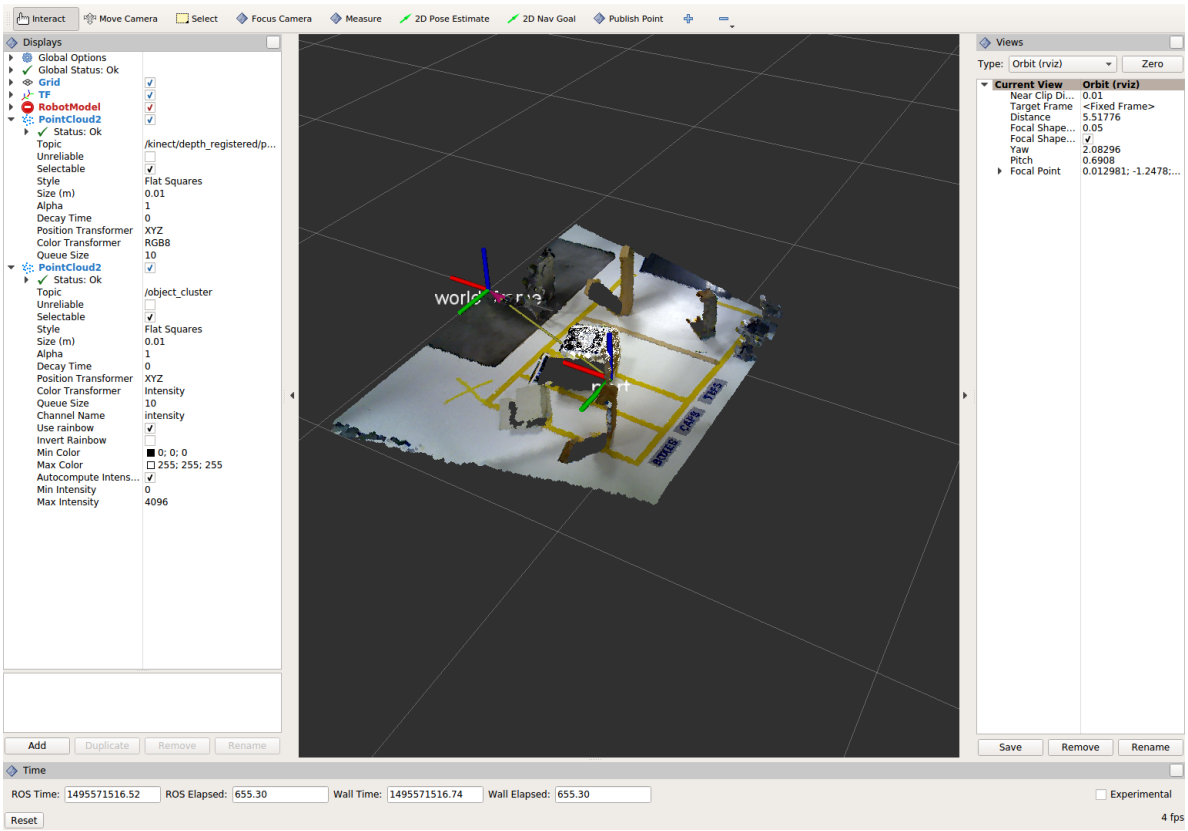
   Set parameters - looking at documentation, S.O.R. uses the number of neighbors to inspect and the standard-deviation threshold to use for outlier rejection:

```
sor.setMeanK (50);
sor.setStddevMulThresh (1.0);
```

   Filter:

```
sor.filter (*sor_cloud_filtered);
```

2. Find the `pcl::toROSMsg` call where the `pc2_cloud` is populated. Replace the current cloud with the new filtered results (`*sor_cloud_filtered`).
3. Compile and run, as in previous steps



Create a Broadcast Transform

While this is not a filter method, it demonstrates how to publish the results of a processing pipeline for other nodes to use. Often, the goal of a processing pipeline is to generate a measurement, location, or some other message for other nodes to use. This sub-task broadcasts a TF transform to define the location of the largest box on the table. This transform could be used by other nodes to identify the position/orientation of the box for grasping.

1. Change/Insert code

   Within perception_node.cpp, find section

```
/* ========================================
 * BROADCAST TRANSFORM (OPTIONAL)
 * ========================================*/
```

   Follow along with the ROS tutorial. The important modifications to make are within the setting of the position and orientation information (setOrigin(tf::Vector3(msg->x, msg->y, 0.0)), and setRotation(q)). Create a transform:

```
static tf::TransformBroadcaster br;
tf::Transform part_transform;

//Here in the tf::Vector3(x,y,z) x,y, and z should be calculated based on the␣
↪pointcloud filtering results
part_transform.setOrigin( tf::Vector3(sor_cloud_filtered->at(1).x, sor_cloud_
↪filtered->at(1).y, sor_cloud_filtered->at(1).z) );
tf::Quaternion q;
```

(continues on next page)

(continued from previous page)

```
q.setRPY(0, 0, 0);
part_transform.setRotation(q);
```

Remember that when you set the origin or set the rpy, this is where you should use the results from all the filters you've applied. At this point the origin is set arbitrarily to the first point within. Broadcast that transform:

```
br.sendTransform(tf::StampedTransform(part_transform, ros::Time::now(), world_
↪frame, "part"));
```

2. Compile and Run as usual. In this case, add a TF display to Rviz and observe the new "part" transform located at the top of the box.

Create a Polygonal Segmentation

When using sensor data for collision detection, it is sometimes necessary to exclude "known" objects from the scene to avoid interference from these objects. MoveIt! contains methods for masking out a robot's own geometry as a "Self Collision" filtering feature. This example shows how to do something similar using PCL's Polygonal Segmentation filtering.

1. Change code

   This method is similar to the plane segmentation from Sub-Task 3, but instead of segmenting out a plane, you can segment and remove a prism. Documentation on the PCL Polygonal Segmentation can be found here and here. The goal in this sub-task is to remove the points that correspond to a known object (e.g. the box we detected earlier). This particular filter is applied to the entire point cloud (original sensor data), but only after we've already completed the processing steps to identify the position/orientation of the box.

   Within perception_node.cpp, add `#include <tf_conversions/tf_eigen.h>` and find section

```
/* ========================================
 * Fill Code: POLYGONAL SEGMENTATION (OPTIONAL)
 * ========================================*/
```

   Set the input cloud:

```
pcl::PointCloud<pcl::PointXYZ>::Ptr sensor_cloud_ptr (new pcl::PointCloud
↪<pcl::PointXYZ>(cloud));
pcl::PointCloud<pcl::PointXYZ>::Ptr prism_filtered_cloud (new pcl::PointCloud
↪<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr pick_surface_cloud_ptr(new pcl::PointCloud
↪<pcl::PointXYZ>);
```

   Declare instance of filter:

```
pcl::ExtractPolygonalPrismData<pcl::PointXYZ> prism;
```

   Set extraction indices:

```
pcl::ExtractIndices<pcl::PointXYZ> extract_ind;
```

   Set input and output:

```
prism.setInputCloud(sensor_cloud_ptr);
pcl::PointIndices::Ptr pt_inliers (new pcl::PointIndices());
```

Set parameters - looking at documentation, ExtractPolygonalPrismData uses a pointcloud defining the polygon vertices as its input.

```
// create prism surface
double box_length=0.25;
double box_width=0.25;
pick_surface_cloud_ptr->width = 5;
pick_surface_cloud_ptr->height = 1;
pick_surface_cloud_ptr->points.resize(5);

pick_surface_cloud_ptr->points[0].x = 0.5f*box_length;
pick_surface_cloud_ptr->points[0].y = 0.5f*box_width;
pick_surface_cloud_ptr->points[0].z = 0;

pick_surface_cloud_ptr->points[1].x = -0.5f*box_length;
pick_surface_cloud_ptr->points[1].y = 0.5f*box_width;
pick_surface_cloud_ptr->points[1].z = 0;

pick_surface_cloud_ptr->points[2].x = -0.5f*box_length;
pick_surface_cloud_ptr->points[2].y = -0.5f*box_width;
pick_surface_cloud_ptr->points[2].z = 0;

pick_surface_cloud_ptr->points[3].x = 0.5f*box_length;
pick_surface_cloud_ptr->points[3].y = -0.5f*box_width;
pick_surface_cloud_ptr->points[3].z = 0;

pick_surface_cloud_ptr->points[4].x = 0.5f*box_length;
pick_surface_cloud_ptr->points[4].y = 0.5f*box_width;
pick_surface_cloud_ptr->points[4].z = 0;

Eigen::Affine3d eigen3d;
tf::transformTFToEigen(part_transform,eigen3d);
pcl::transformPointCloud(*pick_surface_cloud_ptr,*pick_surface_cloud_ptr,
↪Eigen::Affine3f(eigen3d));

prism.setInputPlanarHull( pick_surface_cloud_ptr);
prism.setHeightLimits(-10,10);
```

Segment:

```
prism.segment(*pt_inliers);
```

Remember that after you use the segmentation algorithme that you either want to include or exclude the segmented points using an index extraction.

Set input:

```
extract_ind.setInputCloud(sensor_cloud_ptr);
extract_ind.setIndices(pt_inliers);
```

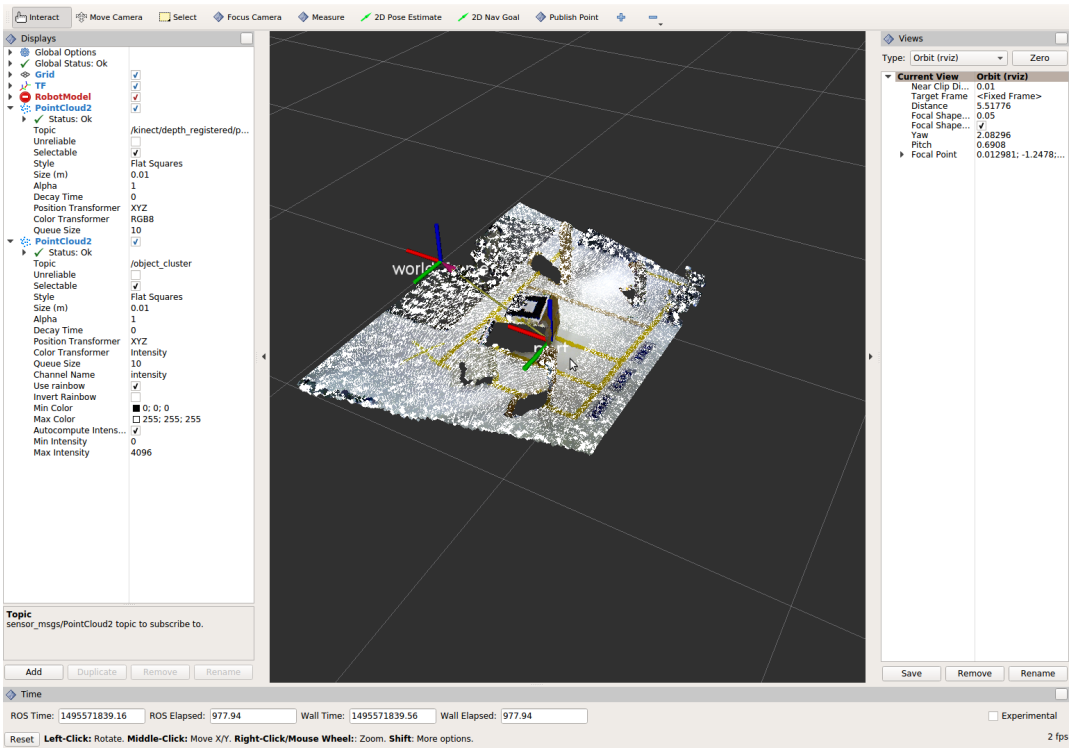
This time, we invert the index extraction, so that we remove points inside the filter and keep points outside the filter.

```
extract_ind.setNegative(true);
```

Filter:

```
extract_ind.filter(*prism_filtered_cloud);
```

2. Find the `pcl::toROSMsg` call where the `pc2_cloud` is populated. This is the point cloud that is published to RViz display. Replace the current cloud with the new filtered results (`*prism_filtered_cloud`).
3. Compile and run as before.



Note: Notice that the target box has been removed from the point cloud display.

Write a launch file

While this is not a filter method, it is useful when using PCL or other perception methods because of the number of parameters used in the different methods.

1. Change/Insert code

   If you are really awesome and read the Task 1 write-up thoroughly, you will note that it was suggested that you put your parameters in one place.

   Within perception_node.cpp, find section

```
/*
 * SET UP PARAMETERS (COULD TO BE INPUT FROM LAUNCH FILE/TERMINAL)
 */
```

   Ideally, as the given parameter examples showed, you would *declare* a parameter of a certain type (std::string frame;), then assign a value for that parameter (frame="some_name";). Below is an example of some of the parameters you could have set.

```
world_frame="kinect_link";
camera_frame="kinect_link";
```

(continues on next page)

(continued from previous page)

```
cloud_topic="kinect/depth_registered/points";
voxel_leaf_size=0.002f;
x_filter_min=-2.5;
x_filter_max=2.5;
y_filter_min=-2.5;
y_filter_max=2.5;
z_filter_min=-2.5;
z_filter_max=1.0;
plane_max_iter=50;
plane_dist_thresh=0.05;
cluster_tol=0.01;
cluster_min_size=100;
cluster_max_size=50000;
```

If you took this step, you will be in great shape to convert what you have into something that can be input from a launch file, or yaml file. You could use the "getParam" method as described in this tutorial. But a better choice might be to use the param method, which returns a default value if the parameter is not found on the parameter server. Get params from ros parameter server/launch file, replacing your previous hardcoded values (but leave the variable declarations!)

```
cloud_topic = priv_nh_.param<std::string>("cloud_topic", "kinect/depth_registered/
↪points");
world_frame = priv_nh_.param<std::string>("world_frame", "kinect_link");
camera_frame = priv_nh_.param<std::string>("camera_frame", "kinect_link");
voxel_leaf_size = priv_nh_.param<float>("voxel_leaf_size", 0.002);
x_filter_min = priv_nh_.param<float>("x_filter_min", -2.5);
x_filter_max = priv_nh_.param<float>("x_filter_max",  2.5);
y_filter_min = priv_nh_.param<float>("y_filter_min", -2.5);
y_filter_max = priv_nh_.param<float>("y_filter_max",  2.5);
z_filter_min = priv_nh_.param<float>("z_filter_min", -2.5);
z_filter_max = priv_nh_.param<float>("z_filter_max",  2.5);
plane_max_iter = priv_nh_.param<int>("plane_max_iterations", 50);
plane_dist_thresh = priv_nh_.param<float>("plane_distance_threshold", 0.05);
cluster_tol = priv_nh_.param<float>("cluster_tolerance", 0.01);
cluster_min_size = priv_nh_.param<int>("cluster_min_size", 100);
cluster_max_size = priv_nh_.param<int>("cluster_max_size", 50000);
```

2. Write launch file.

   Using gedit or some other text editor, make a new file (''lesson_perception/launch/processing_node.launch") and put the following in it.

```
<launch>
  <node name="processing_node" pkg="lesson_perception" type="perception_node"␣
↪output="screen">
    <rosparam>
      cloud_topic: "kinect/depth_registered/points"
      world_frame: "world_frame"
      camera_frame: "kinect_link"
      voxel_leaf_size: 0.001 <!-- mm -->
      x_filter_min: -2.5 <!-- m -->
      x_filter_max: 2.5 <!-- m -->
      y_filter_min: -2.5 <!-- m -->
      y_filter_max: 2.5 <!-- m -->
      z_filter_min: -2.5 <!-- m -->
      z_filter_max: 2.5 <!-- m -->
```

(continues on next page)

(continued from previous page)

```
      plane_max_iterations: 100
      plane_distance_threshold: 0.03
      cluster_tolerance: 0.01
      cluster_min_size: 250
      cluster_max_size: 500000
    </rosparam>
  </node>
</launch>
```

3. Compile as usual...

But this time, run the new launch file that was created instead of using rosrun to start the processing node.

The results should look similar to previous runs. However, now you can edit these configuration parameters much easier! No recompile step is required; just edit the launch-file values and relaunch the node. In a real application, you could take this approach one step further and implement dynamic_reconfigure support in your node. That would allow you to see the results of parameter changes in RViz in real-time!

When you are satisfied with the results, go to each terminal and *CTRL-C*.

We're all done! So it's best to make sure everything is wrapped up and closed.

4.1.3 Introduction to STOMP

Motivation

- Learn how to plan with STOMP through !MoveIt!.

Information and Resources

- STOMP for MoveIt!
- Plugins for MoveIt!

Objectives

- Integrate STOMP into !MoveIt! by changing and adding files to a **moveit_config** package.
- We'll then generate STOMP plans from the Rviz Motion Planning Plugin

Setup

- Create a workspace

```
mkdir --parent ~/catkin_ws/src
cd ~/catkin_ws
catkin init
catkin build
source devel/setup.bash
```

- Copy over existing exercise

```
cd ~/catkin_ws/src
cp -r ~/industrial_training/exercises/4.1 .
```

- Clone industrial_moveit repository into your workspace

```
cd ~/catkin_ws/src
git clone https://github.com/ros-industrial/industrial_moveit.git
git checkout melodic-devel
```

- Install Missing Dependencies

```
cd ~/catkin_ws/src/4.1
rosinstall . .rosinstall
catkin build
```

- Create a **moveit_config** package created with the MoveIt! Setup Assistant

Add STOMP

1. Create a "*stomp_planning_pipeline.launch.xml*" file in the **launch** directory of your **moveit_config** package. The file should contain the following:

```
<launch>

  <!-- Stomp Plugin for MoveIt! -->
  <arg name="planning_plugin" value="stomp_moveit/StompPlannerManager" />

  <!-- The request adapters (plugins) ORDER MATTERS -->
  <arg name="planning_adapters" value="default_planner_request_adapters/
↪FixWorkspaceBounds
                                       default_planner_request_adapters/
↪FixStartStateBounds
                                       default_planner_request_adapters/
↪FixStartStateCollision
                                       default_planner_request_adapters/
↪FixStartStatePathConstraints" />

  <arg name="start_state_max_bounds_error" value="0.1" />

  <param name="planning_plugin" value="$(arg planning_plugin)" />
  <param name="request_adapters" value="$(arg planning_adapters)" />
  <param name="start_state_max_bounds_error" value="$(arg start_state_max_bounds_
↪error)" />
  <rosparam command="load" file="$(find myworkcell_moveit_config)/config/stomp_
↪planning.yaml"/>

</launch>
```

!!! Take notice of the **stomp_planning.yaml** configuration file, this file must exists in moveit_config package.

2. Create the "*stomp_planning.yaml*" configuration file

   This file contains the parameters required by STOMP. The parameters are specific to each ''planning group" defined in the SRDF file. So if there are three planning groups "manipulator", "manipulator_tool", and "manipulator_rail" then the configuration file defines a specific set of parameters for each planning group:

```
stomp/manipulator_rail:
  group_name: manipulator_rail
  optimization:
    num_timesteps: 60
```

(continues on next page)

(continued from previous page)

```
    num_iterations: 40
    num_iterations_after_valid: 0
    num_rollouts: 30
    max_rollouts: 30
    initialization_method: 1 #[1 : LINEAR_INTERPOLATION, 2 : CUBIC_POLYNOMIAL, 3␣
↪: MININUM_CONTROL_COST
    control_cost_weight: 0.0
  task:
    noise_generator:
      - class: stomp_moveit/NormalDistributionSampling
        stddev: [0.05, 0.8, 1.0, 0.8, 0.4, 0.4, 0.4]
    cost_functions:
      - class: stomp_moveit/CollisionCheck
        collision_penalty: 1.0
        cost_weight: 1.0
        kernel_window_percentage: 0.2
        longest_valid_joint_move: 0.05
    noisy_filters:
      - class: stomp_moveit/JointLimits
        lock_start: True
        lock_goal: True
      - class: stomp_moveit/MultiTrajectoryVisualization
        line_width: 0.02
        rgb: [255, 255, 0]
        marker_array_topic: stomp_trajectories
        marker_namespace: noisy
    update_filters:
      - class: stomp_moveit/PolynomialSmoother
        poly_order: 6
      - class: stomp_moveit/TrajectoryVisualization
        line_width: 0.05
        rgb: [0, 191, 255]
        error_rgb: [255, 0, 0]
        publish_intermediate: True
        marker_topic: stomp_trajectory
        marker_namespace: optimized
stomp/manipulator:
  group_name: manipulator
  optimization:
    num_timesteps: 40
    num_iterations: 40
    num_iterations_after_valid: 0
    num_rollouts: 10
    max_rollouts: 10
    initialization_method: 1 #[1 : LINEAR_INTERPOLATION, 2 : CUBIC_POLYNOMIAL, 3␣
↪: MININUM_CONTROL_COST
    control_cost_weight: 0.0
  task:
    noise_generator:
      - class: stomp_moveit/NormalDistributionSampling
        stddev: [0.05, 0.4, 1.2, 0.4, 0.4, 0.1]
    cost_functions:
      - class: stomp_moveit/CollisionCheck
        kernel_window_percentage: 0.2
        collision_penalty: 1.0
        cost_weight: 1.0
        longest_valid_joint_move: 0.05
```

(continues on next page)

(continued from previous page)

```
    noisy_filters:
      - class: stomp_moveit/JointLimits
        lock_start: True
        lock_goal: True
      - class: stomp_moveit/MultiTrajectoryVisualization
        line_width: 0.04
        rgb: [255, 255, 0]
        marker_array_topic: stomp_trajectories
        marker_namespace: noisy
    update_filters:
      - class: stomp_moveit/PolynomialSmoother
        poly_order: 5
      - class: stomp_moveit/TrajectoryVisualization
        line_width: 0.02
        rgb: [0, 191, 255]
        error_rgb: [255, 0, 0]
        publish_intermediate: True
        marker_topic: stomp_trajectory
        marker_namespace: optimized
```

!!! *Save this file in the* **config** *directory of the moveit_config package*

3. Modify the **move_group.launch** file: Open the **move_group.launch** in the launch directory and change the `pipeline` parameter value to `stomp` as shown below:

```
    .
    .
    .
<!-- move_group settings -->
<arg name="allow_trajectory_execution" default="true"/>
<arg name="fake_execution" default="false"/>
<arg name="max_safe_path_cost" default="1"/>
<arg name="jiggle_fraction" default="0.05" />
<arg name="publish_monitored_planning_scene" default="true"/>

<!-- Planning Functionality -->
<include ns="move_group" file="$(find myworkcell_moveit_config)/launch/planning_
↪pipeline.launch.xml">
  <arg name="pipeline" value="stomp" />
</include>

    .
    .
    .
```

Run MoveIt! with STOMP

1. In a sourced terminal, run the **demo.launch** file:

```
roslaunch myworkcell_moveit_config demo.launch
```

2. In Rviz, select robot start and goal positions and plan:

- In the "Motion Planning" panel, go to the "Planning" tab.
- Click the "Select Start State" drop-down, select "allZeros" and click "Update"

- Click the "Select Goal State" drop-down, select "home" and click "Update"
- Click the "Plan" button and watch the arm move past obstacles to reach the goal position. The blue line shows the tool path.

Explore STOMP

1. In Rviz, select other "Start" and "Goal" positions and then hit plan and see the robot move.
2. Display the *Noisy Trajectories* by clicking on the "Marker Array" checkbox in the "Displays" Rviz panel. Hit the "Plan" button again and you'll see the noisy trajectory markers as yellow lines.

   Note: STOMP explores the workspace by generating a number of noisy trajectories as a result of applying noise onto the current trajectory. The degree of noise applied can be be changed by adjusting the "stddev" parameters in the "*stomp_config.yaml*" file. Larger "stddev" values correspond to larger motions of the joints.

Configure STOMP

We'll now change the parameters in the **stomp_config.yaml** and see what effect those changes have on the planning.

1. Ctrl-C in the terminal where you ran the **demo.launch** file earlier to stop the **move_group** planning node.
2. Locate and open up the **stomp_config.yaml** with your preferred editor.
3. Under the "manipulator_rail" group, take notice of the values assigned to "stddev" parameter. Each value is the amplitude of the noise applied to the joint at that position in the array. For instance, the leftmost value in the array will be the value used to set the noise of the first joint "rail_to_base"; which moves the rail along the x direction. Since the "rail_to_base" is a prismatic joint then its units are in meters; for revolute joints the units are radians.
4. Change the "stddev" values (preferably one entry at a time), save the file and rerun the "demo.launch" file in the terminal.
5. Go back to the Rviz window and select arbitrary "Start" and "Goal" positions to see what effect your changes have had on the planning performance.

More info on the STOMP parameters

The STOMP wiki explains these parameter in more detail.

4.1.4 Building a Simple PCL Interface for Python

In this exercise, we will fill in the appropriate pieces of code to build a perception pipeline. The end goal will be to create point cloud filtering operations to demonstrate functionality between ROS and python.

Prepare New Workspace:

We will create a new catkin workspace, since this exercise does not overlap with the previous ScanNPlan exercises.

1. Disable automatic sourcing of your previous catkin workspace:
   1. `gedit ~/.bashrc`
   2. Comment out the line of your `.bashrc` file which sources the previous workspace

```
source /opt/ros/melodic/setup.bash
# source ~/catkin_ws/devel/setup.bash
```

2. Copy the template workspace layout and files:

```
cp -r ~/industrial_training/exercises/python-pcl_ws ~
cd ~/python-pcl_ws/
```

3. Initialize and Build this new workspace

```
catkin init
catkin build
```

4. Source the workspace

```
source ~/python-pcl_ws/devel/setup.bash
```

5. Download the PointCloud file and place the file in your workspace's **src** directory :

```
cp -r ~/industrial_training/exercises/4.2/table.pcd src/
```

Intro (Review Existing Code)

Most of the infrastructure for a ROS node has already been completed for you; the focus of this exercise is the perception algorithms/pipeline. The `CMakelists.txt` and `package.xml` are complete and a source file has been provided. At this time we will explore the source code that has been provided in the `py_perception_node.cpp` file. This tutorial has been modified from training Exercise 5.1 Building a Perception Pipeline and as such the C++ code has already been set up. Open the `perception_node.cpp` file and review the filtering functions.

Create a Python Package

Now that we have converted several filters to C++ functions, we are ready to call it from a Python node.

1. In the terminal, change the directory to your src folder. Create a new package inside your python-pcl_ws:

```
cd ~/python-pcl_ws/src/
catkin create pkg test_pkg_python --catkin-deps rospy
```

2. Check that your package was created:

```
ls
```

We will not be including 'perception_msgs' as a dependency as we will not be creating custom messages in this course. If you wish for a more in depth explanation including how to implement customer messages, here is a good MIT resource on the steps taken.

1. Open `CMakeLists.txt`. Uncomment line 19 or wherever you find **# catkin_python_setup()** and save.

```
catkin_python_setup()
```

Creating setup.py

The `setup.py` file makes your python module available to the entire workspace and subsequent packages. By default, this isn't created by the `catkin_create_pkg` command.

1. In your terminal type

```
gedit filter_call/setup.py
```

2. Copy and paste the following to the `setup.py` file

```
## ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD
from distutils.core import setup
from catkin_pkg.python_setup import generate_distutils_setup
# fetch values from package.xml
setup_args = generate_distutils_setup(
packages=[''],
package_dir={'': 'include'},
)
setup(**setup_args)
```

   Change `packages = [ . . . ],` to your list of strings of the name of the folders inside your *include* folder. By convention, this will be the same name as the package, or `filter_call` . The configures `filter_call/include/filter_call` as a python module available to the whole workspace.

3. Save and close the file.

   In order for this folder to be accessed by any other python script, the `__init__.py` file must exist.

4. Create one in the terminal by typing:

```
touch filter_call/include/filter_call/__init__.py
```

Publishing the Point Cloud

As iterated before, we are creating a ROS C++ node to filter the point cloud when requested by a Python node running a service request for each filtering operation, resulting in a new, aggregated point cloud. Let's start with modifying our C++ code to publish in a manner supportive to python. Remember, the C++ code is already done so all you need to do is write your python script and view the results in Rviz.

Implement a Voxel Filter

1. In `py_perception_node.cpp`, take notice of the function called `filterCallBack` (around line 170). This function will be the entry point for all service calls made by the Python client in order to run point cloud filtering operations.

```
bool filterCallback(lesson_perception::FilterCloud::Request& request,
                    lesson_perception::FilterCloud::Response& response)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr filtered_cloud (new pcl::PointCloud
↪<pcl::PointXYZ>);

  if (request.pcdfilename.empty())
  {
    pcl::fromROSMsg(request.input_cloud, *cloud);
    ROS_INFO_STREAM("cloud size: " << cloud->size());
  }
  else
  {
```

(continues on next page)

(continued from previous page)

```
    pcl::io::loadPCDFile(request.pcdfilename, *cloud);
  }

  if (cloud->empty())
  {
    ROS_ERROR("input cloud empty");
    response.success = false;
    return false;
  }

  switch (request.operation)
  {

    case lesson_perception::FilterCloud::Request::VOXELGRID :
    {
      filtered_cloud = voxelGrid(cloud, 0.01);
      break;
    }
    default :
    {
      ROS_ERROR("No valid request found");
      return false;
    }

   }

/*
 * SETUP RESPONSE
 */
  pcl::toROSMsg(*filtered_cloud, response.output_cloud);
  response.output_cloud.header=request.input_cloud.header;
  response.output_cloud.header.frame_id="kinect_link";
  response.success = true;
  return true;
}
```

2. Within `main`, take notice of the lines starting at 244, this is where we load the parameters used by the various filters.

```
priv_nh_.param<double>("leaf_size", leaf_size_, 0.0f);
```

Build the package and go into the **filter_call** package now

3. Now that we have the framework for the filtering, open your terminal. Make sure you are in the `filter_call` directory. Create a `scripts` folder.

```
mkdir scripts
```

4. Copy and paste the following code at the top of `filter_call.py` to import necessary libraries:

```
#!/usr/bin/env python

import rospy
import lesson_perception.srv
from sensor_msgs.msg import PointCloud2
```

5. We will create an `if` statement that contains the `main` function that is called when the node is run from the

command line. Paste the following after the import statements:

```
if __name__ == '__main__':
    try:
       rospy.init_node('filter_cloud', anonymous=True)
       rospy.wait_for_service('filter_cloud')

       rospy.spin()
    except Exception as e:
        print("Service call failed: %s" % str(e))
```

1. The `rospy.init_node` function initializes the node and gives it a name
2. The `rospy.wait_for_service` waits for the `filter_cloud` service.
3. The `rospy.spin` is the Python counterpart of the `roscpp::spin()` function in C++.

6. Call the service to apply a Voxel Grid filter. Copy and paste the following inside the `try` block in the line following the `rospy.wait_for_service` function:

```
# =======================
# VOXEL GRID FILTER
# =======================

srvp = rospy.ServiceProxy('filter_cloud', lesson_perception.srv.FilterCloud)
req = lesson_perception.srv.FilterCloudRequest()
req.pcdfilename = rospy.get_param('~pcdfilename', '')
req.operation = lesson_perception.srv.FilterCloudRequest.VOXELGRID

# FROM THE SERVICE, ASSIGN POINTS
req.input_cloud = PointCloud2()

# ERROR HANDLING
if req.pcdfilename == '':
    raise Exception('No file parameter found')

# PACKAGE THE FILTERED POINTCLOUD2 TO BE PUBLISHED
res_voxel = srvp(req)
print('response received')
if not res_voxel.success:
    raise Exception('Unsuccessful voxel grid filter operation')

# PUBLISH VOXEL FILTERED POINTCLOUD2
pub = rospy.Publisher('/perception_voxelGrid', PointCloud2, queue_size=1,␣
↪latch=True)
pub.publish(res_voxel.output_cloud)
print("published: voxel grid filter response")
```

7. We need to make the Python file executable. In your terminal:

```
sudo chmod +x filter_call/scripts/filter_call.py
```

Viewing Results

1. In your terminal, run

```
roscore
```

2. Source a new terminal and run the C++ filter service node

```
rosrun lesson_perception py_perception_node
```

3. Source a new terminal and run the Python service client node. Note your file path may be different.

```
rosrun filter_call filter_call.py _pcdfilename:=`d`/src/table.pcd
```

4. Source a new terminal and run the `tf2_ros` package to publish a static coordinate transform from the child frame to the world frame

```
rosrun tf2_ros static_transform_publisher 0 0 0 0 0 0 world_frame kinect_link
```

5. Source a new terminal and run Rviz

```
rosrun rviz rviz
```

6. Add a new PointCloud2 in Rviz
7. In global options, change the fixed frame to **kinect_link** or **world_frame**, and in the PointCloud 2, select your topic to be '/perception_voxelGrid'

Note: You may need to uncheck and recheck the PointCloud2.

Implement Pass-Through Filters

1. In `py_perception_node.cpp` in the `lesson_perception` package, update the switch to look as shown below:

```
switch (request.operation)
{

  case lesson_perception::FilterCloud::Request::VOXELGRID :
  {
    filtered_cloud = voxelGrid(cloud, 0.01);
    break;
  }
  case lesson_perception::FilterCloud::Request::PASSTHROUGH :
  {
    filtered_cloud = passThrough(cloud);
    break;
  }
  default :
  {
    ROS_ERROR("No valid request found");
    return false;
  }

}
```

2. Save and build

Edit the Python Code

3. Open the python node and copy paste the following code after the voxel grid, before the `rospy.spin()`. Keep care to maintain indents:

```
# =======================
# PASSTHROUGH FILTER
# =======================

srvp = rospy.ServiceProxy('filter_cloud', lesson_perception.srv.FilterCloud)
req = lesson_perception.srv.FilterCloudRequest()
req.pcdfilename = ''
req.operation = lesson_perception.srv.FilterCloudRequest.PASSTHROUGH
# FROM THE SERVICE, ASSIGN POINTS
req.input_cloud = res_voxel.output_cloud

# PACKAGE THE FILTERED POINTCLOUD2 TO BE PUBLISHED
res_pass = srvp(req)
print('response received')
if not res_voxel.success:
    raise Exception('Unsuccessful pass through filter operation')

# PUBLISH PASSTHROUGH FILTERED POINTCLOUD2
pub = rospy.Publisher('/perception_passThrough', PointCloud2, queue_size=1,␣
↪latch=True)
pub.publish(res_pass.output_cloud)
print("published: pass through filter response")
```

4. Save and run from the terminal, repeating steps outlined for the voxel filter.

   Within Rviz, compare PointCloud2 displays based on the `/kinect/depth_registered/points` (original camera data) and `perception_passThrough` (latest processing step) topics. Part of the original point cloud has been "clipped" out of the latest processing result.

   When you are satisfied with the pass-through filter results, press Ctrl+C to kill the node. There is no need to close or kill the other terminals/nodes.

Plane Segmentation

This method is one of the most useful for any application where the object is on a flat surface. In order to isolate the objects on a table, you perform a plane fit to the points, which finds the points which comprise the table, and then subtract those points so that you are left with only points corresponding to the object(s) above the table. This is the most complicated PCL method we will be using and it is actually a combination of two: the RANSAC segmentation model, and the extract indices tool. An in depth example can be found on the PCL Plane Model Segmentation Tutorial; otherwise you can copy the below code snippet.

1. In `py_perception_node.cpp`, update the switch statement in `filterCallback` to look as shown below:

```
switch (request.operation)
{

  case lesson_perception::FilterCloud::Request::VOXELGRID :
  {
    filtered_cloud = voxelGrid(cloud, 0.01);
    break;
  }
  case lesson_perception::FilterCloud::Request::PASSTHROUGH :
  {
    filtered_cloud = passThrough(cloud);
    break;
```

(continues on next page)

(continued from previous page)

```
  }
  case lesson_perception::FilterCloud::Request::PLANESEGMENTATION :
  {
    filtered_cloud = planeSegmentation(cloud);
    break;
  }
  default :
  {
    ROS_ERROR("No valid request found");
    return false;
  }

}
```

2. Save and build

Edit the Python Code

3. Copy paste the following code in `filter_call.py`, after the passthrough filter section. Keep care to maintain indents:

```
# =======================
# PLANE SEGMENTATION
# =======================

srvp = rospy.ServiceProxy('filter_cloud', lesson_perception.srv.FilterCloud)
req = lesson_perception.srv.FilterCloudRequest()
req.pcdfilename = ''
req.operation = lesson_perception.srv.FilterCloudRequest.PLANESEGMENTATION
# FROM THE SERVICE, ASSIGN POINTS
req.input_cloud = res_pass.output_cloud

# PACKAGE THE FILTERED POINTCLOUD2 TO BE PUBLISHED
res_seg = srvp(req)
print('response received')
if not res_voxel.success:
    raise Exception('Unsuccessful plane segmentation operation')

# PUBLISH PLANESEGMENTATION FILTERED POINTCLOUD2
pub = rospy.Publisher('/perception_planeSegmentation', PointCloud2, queue_size=1,␣
↪latch=True)
pub.publish(res_seg.output_cloud)
print("published: plane segmentation filter response")
```

4. Save and run from the terminal, repeating steps outlined for the voxel filter.

   Within Rviz, compare PointCloud2 displays based on the `/kinect/depth_registered/points` (original camera data) and `perception_planeSegmentation` (latest processing step) topics. Only points lying above the table plane remain in the latest processing result.

   1. When you are done viewing the results you can go back and change the `setMaxIterations` and `setDistanceThreshold` parameter values to control how tightly the plane-fit classifies data as inliers/outliers, and view the results again. Try using values of `maxIterations=100` and `distThreshold=0.010`
   2. When you are satisfied with the plane segmentation results, use Ctrl+C to kill the node. There is no need to close or kill the other terminals/nodes.

Euclidian Cluster Extraction

This method is useful for any application where there are multiple objects. This is also a complicated PCL method. An in depth example can be found on the PCL Euclidean Cluster Extration Tutorial.

1. In `py_perception_node.cpp`, update the switch statement in `filterCallback` to look as shown below:

```
switch (request.operation)
{

  case lesson_perception::FilterCloud::Request::VOXELGRID :
  {
    filtered_cloud = voxelGrid(cloud, 0.01);
    break;
  }
  case lesson_perception::FilterCloud::Request::PASSTHROUGH :
  {
    filtered_cloud = passThrough(cloud);
    break;
  }
  case lesson_perception::FilterCloud::Request::PLANESEGMENTATION :
  {
    filtered_cloud = planeSegmentation(cloud);
    break;
  }
  case lesson_perception::FilterCloud::Request::CLUSTEREXTRACTION :
  {
    std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> temp␣
↪=clusterExtraction(cloud);
    if (temp.size()>0)
    {
      filtered_cloud = temp[0];
    }
    break;
  }
  default :
  {
    ROS_ERROR("No valid request found");
    return false;
  }

}
```

2. Save and build

Edit the Python Code

3. Copy paste the following code in `filter_call.py` after the plane segmentation section. Keep care to maintain indents:

```
# ========================
# CLUSTER EXTRACTION
# ========================

srvp = rospy.ServiceProxy('filter_cloud', lesson_perception.srv.FilterCloud)
req = lesson_perception.srv.FilterCloudRequest()
req.pcdfilename = ''
```

(continues on next page)

(continued from previous page)

```
req.operation = lesson_perception.srv.FilterCloudRequest.CLUSTEREXTRACTION
# FROM THE SERVICE, ASSIGN POINTS
req.input_cloud = res_seg.output_cloud

# PACKAGE THE FILTERED POINTCLOUD2 TO BE PUBLISHED
res_cluster = srvp(req)
print('response received')
if not res_voxel.success:
    raise Exception('Unsuccessful cluster extraction operation')

# PUBLISH CLUSTEREXTRACTION FILTERED POINTCLOUD2
pub = rospy.Publisher('/perception_clusterExtraction', PointCloud2, queue_size=1,␣
↪latch=True)
pub.publish(res_cluster.output_cloud)
print("published: cluster extraction filter response")
```

4. Save and run from the terminal, repeating steps outlined for the voxel filter.
    1. When you are satisfied with the cluster extraction results, use Ctrl+C to kill the node. If you are done experimenting with this tutorial, you can kill the nodes running in the other terminals.

Future Study

The student is encouraged to convert Exercise 5.1 into callable functions and further refine the filtering operations.

Furthermore, for simplicity, the python code was repeated for each filtering instance. The student is encouraged to create a loop to handle the publishing instead of repeating large chunks of code. The student can also leverage the full functionality of the parameter handling instead of just using defaults, can set those from python. There are several more filtering operations not outlined here, if the student wants practice creating those function calls.

4.1.5 OpenCV Image Processing (Python)

In this exercise, we will gain familiarity with both OpenCV and Python, through a simple 2D image-processing application.

Motivation

OpenCV is a mature, stable library for 2D image processing, used in a wide variety of applications. Much of ROS makes use of 3D sensors and point-cloud data, but there are still many applications that use traditional 2D cameras and image processing.

This tutorial uses python to build the image-processing pipeline. Python is a good choice for this application, due to its ease of rapid prototyping and existing bindings to the OpenCV library.

Further Information and Resources

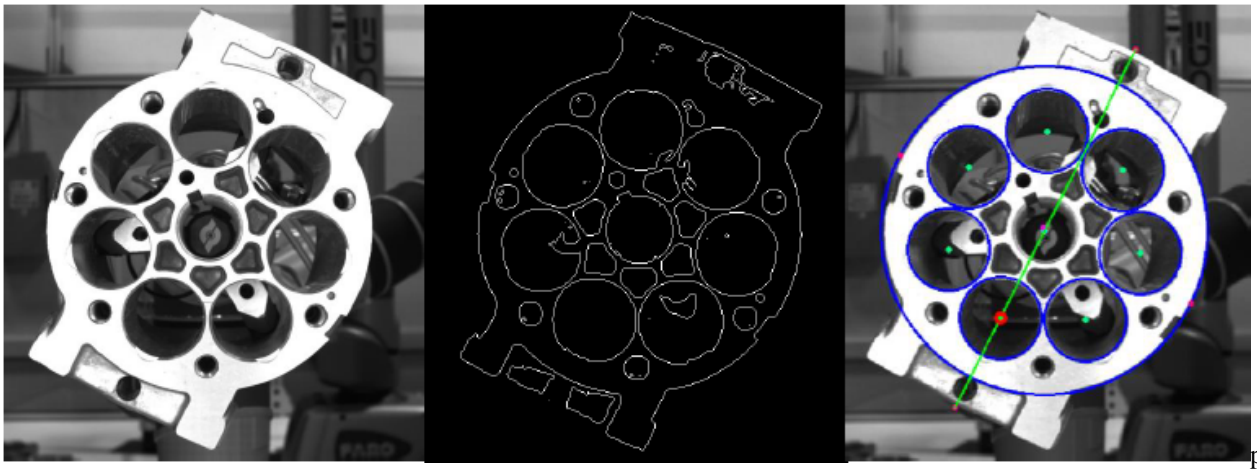
- OpenCV Website
- OpenCV API
- OpenCV Python Tutorials
- ROS cv_bridge package (Python)

- Writing a Publisher and Subscriber (Python)
- sensor_msgs/Image

Problem Statement

In this exercise, you will create a new node to determine the angular pose of a pump housing using the OpenCV image processing library. The pump's orientation is computed using a series of processing steps to extract and compare geometry features:

1. Resize the image (to speed up processing)
2. Threshold the image (convert to black & white)
3. Locate the pump's outer housing (circle-finding)
4. Locate the piston sleeve locations (blob detection)
5. Estimate primary axis using bounding box
6. Determine orientation using piston sleeve locations
7. Calculate the axis orientation relative to a reference (horizontal) axis


pump images

Implementation

Create package

This exercise uses a single package that can be placed in any catkin workspace. The examples below will use the `~/catkin_ws` workspace from earlier exercises.

1. Create a new `detect_pump` package to contain the new python nodes we'll be making:

```
cd ~/catkin_ws/src
catkin create pkg detect_pump --catkin-deps rospy cv_bridge
```

   - all ROS packages depend on `rospy`
   - we'll use `cv_bridge` to convert between ROS's standard Image message and OpenCV's Image object
   - `cv_bridge` also automatically brings in dependencies on the relevant OpenCV modules

2. Create a python module for this package:

```
cd detect_pump
mkdir nodes
```

   - For a simple package such as this, the Python Style Guide recommends this simplified package structure.
   - More complex packages (e.g. with exportable modules, msg/srv defintions, etc.) should us a more complex package structure, with an `__init__.py` and `setup.py`.
     - reference Installing Python Scripts
     - reference Handling setup.py

Create an Image Publisher

The first node will read in an image from a file and publish it as a ROS Image message on the `image` topic.

- Note: ROS already contains an `image_publisher` package/node that performs this function, but we will duplicate it here to learn about ROS Publishers in Python.

1. Create a new python script for our image-publisher node (`nodes/image_pub.py`). Fill in the following template for a skeleton ROS python node:

```
#!/usr/bin/env python
import rospy

def start_node():
    rospy.init_node('image_pub')
    rospy.loginfo('image_pub node started')

if __name__ == '__main__':
    try:
        start_node()
    except rospy.ROSInterruptException:
        pass
```

2. Allow execution of the new script file:

```
chmod u+x nodes/image_pub.py
```

3. Build the package and run the image publisher:

```
catkin build
roscore
rosrun detect_pump image_pub.py
```

   - You should see the "node started" message

4. Read the image file to publish, using the filename provided on the command line
   1. Import the `sys` and `cv2` (OpenCV) modules:

```
import sys
import cv2
```

   2. Pass the command-line argument into the `start_node` function:

```
def start_node(filename):
...
start_node( rospy.myargv(argv=sys.argv)[1] )
```

- Note the use of `rospy.myargv()` to strip out any ROS-specific command-line arguments.

3. In the `start_node` function, call the OpenCV imread function to read the image. Then use imshow to display it:

```
img = cv2.imread(filename)
cv2.imshow("image", img)
cv2.waitKey(2000)
```

4. Run the node, with the specified image file:

```
rosrun detect_pump image_pub.py ~/industrial_training/exercises/5.4/pump.jpg
```

- You should see the image displayed
- Comment out the `imshow`/`waitKey` lines, as we won't need those any more
- Note that you don't need to run `catkin build` after editing the python file, since no compile step is needed.

5. Convert the image from OpenCV Image object to ROS Image message:
   1. Import the `CvBridge` and `Image` (ROS message) modules:

```
from cv_bridge import CvBridge
from sensor_msgs.msg import Image
```

   2. Add a call to the CvBridge cv2_to_imgmsg method:

```
bridge = CvBridge()
imgMsg = bridge.cv2_to_imgmsg(img, "bgr8")
```

6. Create a ROS publisher to continually publish the Image message on the `image` topic. Use a loop with a 1 Hz throttle to publish the message.

```
pub = rospy.Publisher('image', Image, queue_size=10)
while not rospy.is_shutdown():
    pub.publish(imgMsg)
    rospy.Rate(1.0).sleep()  # 1 Hz
```

7. Run the node and inspect the newly-published image message
   1. Run the node (as before):

```
rosrun detect_pump image_pub.py ~/industrial_training/exercises/5.4/pump.jpg
```

   2. Inspect the message topic using command-line tools:

```
rostopic list
rostopic hz /image
rosnode info /image_pub
```

   3. Inspect the published image using the standalone image_view node

```
rosrun image_view image_view
```

Create the Detect_Pump Image-Processing Node

The next node will subscribe to the `image` topic and execute a series of processing steps to identify the pump's orientation relative to the horizontal image axis.

1. As before, create a basic ROS python node (`detect_pump.py`) and set its executable permissions:

```
#!/usr/bin/env python
import rospy

# known pump geometry
#  - units are pixels (of half-size image)
PUMP_DIAMETER = 360
PISTON_DIAMETER = 90
PISTON_COUNT = 7

def start_node():
    rospy.init_node('detect_pump')
    rospy.loginfo('detect_pump node started')

if __name__ == '__main__':
    try:
        start_node()
    except rospy.ROSInterruptException:
        pass
```

```
chmod u+x nodes/detect_pump.py
```

   - Note that we don't have to edit `CMakeLists` to create new build rules for each script, since python does not need to be compiled.

2. Add a ROS subscriber to the `image` topic, to provide the source for images to process.
   1. Import the `Image` message header

```
from sensor_msgs.msg import Image
```

   2. Above the `start_node` function, create an empty callback (`process_image`) that will be called when a new Image message is received:

```
def process_image(msg):
    try:
       pass
    except Exception as err:
        print err
```

      - The try/except error handling will allow our code to continue running, even if there are errors during the processing pipeline.

   3. In the `start_node` function, create a ROS Subscriber object:
      - subscribe to the `image` topic, monitoring messages of type `Image`
      - register the callback function we defined above

```
rospy.Subscriber("image", Image, process_image)
rospy.spin()
```

- reference: rospy.Subscriber
- reference: rospy.spin

4. Run the new node and verify that it is subscribing to the topic as expected:

```
rosrun detect_pump detect_pump.py
rosnode info /detect_pump
rqt_graph
```

3. Convert the incoming `Image` message to an OpenCV `Image` object and display it As before, we'll use the `CvBridge` module to do the conversion.
   1. Import the `CvBridge` modules:

```
from cv_bridge import CvBridge
```

   2. In the `process_image` callback, add a call to the CvBridge imgmsg_to_cv2 method:

```
# convert sensor_msgs/Image to OpenCV Image
bridge = CvBridge()
orig = bridge.imgmsg_to_cv2(msg, "bgr8")
```

      - This code (and all other image-processing code) should go inside the `try` block, to ensure that processing errors don't crash the node.
      - This should replace the placeholder `pass` command placed in the `try` block earlier

   3. Use the OpenCV `imshow` method to display the images received. We'll create a pattern that can be re-used to show the result of each image-processing step.
      1. Import the OpenCV `cv2` module:

```
import cv2
```

      2. Add a display helper function above the `process_image` callback:

```
def showImage(img):
    cv2.imshow('image', img)
    cv2.waitKey(1)
```

      3. Copy the received image to a new "drawImg" variable:

```
drawImg = orig
```

      4. **Below** the `except` block (outside its scope; at `process_image` scope, display the `drawImg` variable:

```
# show results
showImage(drawImg)
```

   4. Run the node and see the received image displayed.

4. The first step in the image-processing pipeline is to resize the image, to speed up future processing steps. Add the following code inside the `try` block, then rerun the node.

```
# resize image (half-size) for easier processing
resized = cv2.resize(orig, None, fx=0.5, fy=0.5)
drawImg = resized
```

- you should see a smaller image being displayed
- reference: resize()

5. Next, convert the image from color to grayscale. Run the node to check for errors, but the image will still look the same as previously.

```
# convert to single-channel image
gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
drawImg = cv2.cvtColor(gray, cv2.COLOR_GRAY2BGR)
```

- Even though the original image looks gray, the JPG file, Image message, and `orig` OpenCV image are all 3-channel color images.
- Many OpenCV functions operate on individual image channels. Converting an image that appears gray to a "true" 1-channel grayscale image can help avoid confusion further on.
- We convert back to a color image for `drawImg` so that we can draw colored overlays on top of the image to display the results of later processing steps.
- reference: cvtColor()

6. Apply a thresholding operation to turn the grayscale image into a binary image. Run the node and see the thresholded image.

```
# threshold grayscale to binary (black & white) image
threshVal = 75
ret,thresh = cv2.threshold(gray, threshVal, 255, cv2.THRESH_BINARY)
drawImg = cv2.cvtColor(thresh, cv2.COLOR_GRAY2BGR)
```

You should experiment with the `threshVal` paramter to find a value that works best for this image. Valid values for this parameter lie between [0-255], to match the grayscale pixel intensity range. Find a value that clearly highlights the pump face geometry. I found that a value of `150` seemed good to me.

- reference threshold

7. Detect the outer pump-housing circle.

This is not actually used to detect the pump angle, but serves as a good example of feature detection. In a more complex scene, you could use OpenCV's Region Of Interest (ROI) feature to limit further processing to only features inside this pump housing circle.

1. Use the `HoughCircles` method to detect a pump housing of known size:

```
# detect outer pump circle
pumpRadiusRange = ( PUMP_DIAMETER/2-2, PUMP_DIAMETER/2+2)
pumpCircles = cv2.HoughCircles(thresh, cv2.HOUGH_GRADIENT, 1, PUMP_DIAMETER,␣
↪param2=2, minRadius=pumpRadiusRange[0], maxRadius=pumpRadiusRange[1])
```

- reference: HoughCircles

2. Add a function to display all detected circles (above the `process_image` callback):

```
def plotCircles(img, circles, color):
    if circles is None: return
```

(continues on next page)

(continued from previous page)

```
    for (x,y,r) in circles[0]:
        cv2.circle(img, (int(x),int(y)), int(r), color, 2)
```

3. Below the circle-detect, call the display function and check for the expected # of circles (1)

```
plotCircles(drawImg, pumpCircles, (255,0,0))
if (pumpCircles is None):
    raise Exception("No pump circles found!")
elif len(pumpCircles[0])<>1:
    raise Exception("Wrong # of pump circles: found {} expected {}".
↪format(len(pumpCircles[0]),1))
else:
    pumpCircle = pumpCircles[0][0]
```

4. Run the node and see the detected circles.

   - Experiment with adjusting the `param2` input to `HoughCircles` to find a value that seems to work well. This parameter represents the sensitivity of the detector; lower values detect more circles, but also will return more false-positives.
   - Tru removing the `min/maxRadius` parameters or reducing the minimum distance between circles (4th parameter) to see what other circles are detected.
   - I found that a value of `param2=7` seemed to work well

8. Detect the piston sleeves, using blob detection.

   Blob detection analyses the image to identify connected regions (blobs) of similar color. Filtering of the resulting blob features on size, shape, or other characteristics can help identify features of interest. We will be using OpenCV's SimpleBlobDetector.

   1. Add the following code to run blob detection on the binary image:

```
# detect blobs inside pump body
pistonArea = 3.14159 * PISTON_DIAMETER**2 / 4
blobParams = cv2.SimpleBlobDetector_Params()
blobParams.filterByArea = True
blobParams.minArea = 0.80 * pistonArea
blobParams.maxArea = 1.20 * pistonArea
blobDetector = cv2.SimpleBlobDetector_create(blobParams)
blobs = blobDetector.detect(thresh)
```

      - Note the use of an Area filter to select blobs within 20% of the expected piston-sleeve area.
      - By default, the blob detector is configured to detect black blobs on a white background. so no additional color filtering is required.

   2. Below the blob detection, call the OpenCV blob display function and check for the expected # of piston sleeves (7):

```
drawImg = cv2.drawKeypoints(drawImg, blobs, (), (0,255,0), cv2.DRAW_MATCHES_
↪FLAGS_DRAW_RICH_KEYPOINTS)
if len(blobs) <> PISTON_COUNT:
    raise Exception("Wring # of pistons: found {} expected {}".
↪format(len(blobs), PISTON_COUNT))
pistonCenters = [(int(b.pt[0]),int(b.pt[1])) for b in blobs]
```

   3. Run the node and see if all piston sleeves were properly identified

9. Detect the primary axis of the pump body.

   This axis is used to identify the key piston sleeve feature. We'll reduce the image to contours (outlines), then find the largest one, fit a rectangular box (rotated for best-fit), and identify the major axis of that box.

   1. Calculate image contours and select the one with the largest area:

```
# determine primary axis, using largest contour
im2, contours, h = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_
↪SIMPLE)
maxC = max(contours, key=lambda c: cv2.contourArea(c))
```

   2. Fit a bounding box to the largest contour:

```
boundRect = cv2.minAreaRect(maxC)
```

   3. Copy these 3 helper functions to calculate the endpoints of the rectangle's major axis (above the `process_image` callback):

```
import math
...

def ptDist(p1, p2):
    dx=p2[0]-p1[0]; dy=p2[1]-p1[1]
    return math.sqrt( dx*dx + dy*dy )

def ptMean(p1, p2):
    return ((int(p1[0]+p2[0])/2, int(p1[1]+p2[1])/2))

def rect2centerline(rect):
    p0=rect[0]; p1=rect[1]; p2=rect[2]; p3=rect[3];
    width=ptDist(p0,p1); height=ptDist(p1,p2);

    # centerline lies along longest median
    if (height > width):
        cl = ( ptMean(p0,p1), ptMean(p2,p3) )
    else:
        cl = ( ptMean(p1,p2), ptMean(p3,p0) )

    return cl
```

   4. Call the `rect2centerline` function from above, with the bounding rectangle calculated earlier. Draw the centerline on top of our display image.

```
centerline = rect2centerline(cv2.boxPoints(boundRect))
cv2.line(drawImg, centerline[0], centerline[1], (0,0,255))
```

10. The final step is to identify the key piston sleeve (closest to centerline) and use position to calculate the pump angle.

    1. Add a helper function to calculate the distance between a point and the centerline:

```
def ptLineDist(pt, line):
    x0=pt[0]; x1=line[0][0]; x2=line[1][0];
    y0=pt[1]; y1=line[0][1]; y2=line[1][1];
    return abs((x2-x1)*(y1-y0)-(x1-x0)*(y2-y1))/(math.sqrt((x2-x1)*(x2-
↪x1)+(y2-y1)*(y2-y1)))
```

2. Call the `ptLineDist` function to find which piston blob is closest to the centerline. Update the drawImg to show which blob was identified.

```
# find closest piston to primary axis
closestPiston = min( pistonCenters, key=lambda ctr: ptLineDist(ctr,␣
↪centerline))
cv2.circle(drawImg, closestPiston, 5, (255,255,0), -1)
```

3. Calculate the angle between the 3 key points: piston sleeve centerpoint, pump center, and an arbitrary point along the horizontal axis (our reference "zero" position).

   1. Add a helper function `findAngle` to calculate the angle between 3 points:

```
import numpy as np

def findAngle(p1, p2, p3):
    p1=np.array(p1); p2=np.array(p2); p3=np.array(p3);
    v1=p1-p2; v2=p3-p2;
    return math.atan2(-v1[0]*v2[1]+v1[1]*v2[0],v1[0]*v2[0]+v1[1]*v2[1]) *␣
↪180/3.14159
```

   2. Call the `findAngle` function with the appropriate 3 keypoints:

```
# calculate pump angle
p1 = (orig.shape[1], pumpCircle[1])
p2 = (pumpCircle[0], pumpCircle[1])
p3 = (closestPiston[0], closestPiston[1])
angle = findAngle(p1, p2, p3)
print "Found pump angle: {}".format(angle)
```

11. You're done! Run the node as before. The reported pump angle should be near 24 degrees.

Challenge Exercises

For a greater challenge, try the following suggestions to modify the operation of this image-processing example:

1. Modify the `image_pub` node to rotate the image by 10 degrees between each publishing step. The following code can be used to rotate an image:

```
def rotateImg(img, angle):
    rows,cols,ch = img.shape
    M = cv2.getRotationMatrix2D((cols/2,rows/2),angle,1)
    return cv2.warpAffine(img,M,(cols,rows))
```

2. Change the `detect_pump` node to provide a **service** that performs the image detection. Define a custom service type that takes an input image and outputs the pump angle. Create a new application node that subscribes to the `image` topic and calls the `detect_pump` service.

3. Try using `HoughCircles` instead of `BlobDetector` to locate the piston sleeves.

4.2 Session 6 - Documentation, Unit Tests, ROS Utilities and Debugging ROS

`Slides`

4.2.1 Documentation Generation

Motivation

The ROS Scan-N-Plan application is complete and tested. It is important to thoroughly document the code so that other developers may easily understand this program.

Information and Resources

`doxygen` generates documentation from annotated source code

`rosdoc_lite` is a ROS wrapper for doxygen

Scan-N-Plan Application: Problem Statement

We have completed and tested our Scan-N-Plan program from Exercise 4.0 and we need to release the code to the public. Your goal is to make documentation viewable in a browser. You may accomplish this by annotated the `myworkcell_core` package with `doxygen` syntax and generating documentation with `rosdoc_lite`.

Scan-N-Plan Application: Guidance

Annotate the Source Code

1. Open the myworkcell_node.cpp file from the previous example.
    1. Annotate above the `ScanNPlan` class:

```
/**
* @brief The ScanNPlan class is a client of the vision and path plan servers.␣
↪ The ScanNPLan class takes
* these services, computes transforms and published commands to the robot.
*/
class ScanNPlan
```

    2. Annotate above the `start` method

```
/**
 * @brief start performs the motion planning and execution functions of the␣
↪ScanNPlan of
 * the node. The start method makes a service request for a transform that
 * localizes the part.  The start method moves the "manipulator"
 * move group to the localization target.  The start method requests
 * a cartesian path based on the localization target.  The start method
 * sends the cartesian path to the actionlib client for execution, bypassig
 * MoveIt!
 * @param base_frame is a string that specifies the reference frame
 * coordinate system.
 */
void start(const std::string& base_frame)
```

    3. Annotate above the `main` function

```
/**
 * @brief main is the ros interface for the ScanNPlan Class
 * @param argc ROS uses this to parse remapping arguments from the command␣
↪line.
 * @param argv ROS uses this to parse remapping arguments from the command␣
↪line.
 * @return ROS provides typical return codes, 0 or -1, depending on the
 * execution.
 */
int main(int argc, char** argv)
```

4. Additional annotations may be placed above private variables or other important code elements.

Generate documentation

1. Install `rosdoc_lite`:

```
sudo apt install ros-melodic-rosdoc-lite
```

2. Build the workspace so we can produce documentation for its packages later:

```
catkin build
```

3. Source the workspace

```
source ./devel/setup.bash
```

4. Run `rosdoc_lite` to generate the documentation

```
roscd myworkcell_core
rosdoc_lite .
```

View the Documentation

1. Open the documentation in a browser:

```
firefox doc/html/index.html
```

2. Navigate to Classes -> ScanNPlan and view the documentation.

4.2.2 Unit Testing

In this exercise we will write a unit tests in the *myworkcell_core* package.

Motivation

The ROS Scan-N-Plan application from Exercise 4.0 is complete and documented. Now we want to test the program to make sure it behaves as expected.

Information and Resources

Google Test: C++ XUnit test framework

rostest: ROS wrapper for XUnit test framework

catkin testing: Building and running tests with catkin

Problem Statement

We have completed and and documented our Scan-N-Plan program. We need to create a test framework so we can be sure our program runs as intended after it is built. In addition to ensuring the code runs as intended, unit tests allow you to easily check if new code changes functionality in unexpected ways. Your goal is to create the unit test frame work and write a few tests.

Guidance

Create the unit test frame work

1. Create a *test* folder in the *myworkcell_core/src* folder. In the workspace directory:

```
catkin build
source devel/setup.bash
roscd myworkcell_core
mkdir src/test
```

2. Create utest.cpp file in the *myworkcell_core/src/test* folder:

```
touch src/test/utest.cpp
```

3. Open utest.cpp in QT and include ros & gtest:

```
#include <ros/ros.h>
#include <gtest/gtest.h>
```

4. Write a dummy test that will return true if executed. This will test our framework and we will replace it later with more useful tests:

```
TEST(TestSuite, myworkcell_core_framework)
{
  ASSERT_TRUE(true);
}
```

5. Next include the general main function, which will execute the unit tests we write later:

```
int main(int argc, char **argv)
{
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

6. Edit *myworkcell_core/CMakeLists.txt* to build the u_test.cpp file. Append CMakeLists.txt:

```
if(CATKIN_ENABLE_TESTING)
  find_package(rostest REQUIRED)
  add_rostest_gtest(utest_node test/utest_launch.test src/test/utest.cpp)
  target_link_libraries(utest_node ${catkin_LIBRARIES})
endif()
```

7. Create a test folder under *myworkcell_core*

```
roscd myworkcell_core
mkdir test
```

8. Create a test launch file:

```
touch test/utest_launch.test
```

9. Open the *utest_launch.test* file in QT and populate the file:

```
<?xml version="1.0"?>
<launch>
    <node pkg="fake_ar_publisher" type="fake_ar_publisher_node" name="fake_ar_
↪publisher"/>
    <test test-name="unit_test_node" pkg="myworkcell_core" type="utest_node"/>
</launch>
```

10. Build and test the framework

```
catkin run_tests myworkcell_core
```

The console output should show (buried in the midst of many build messages):

```
[ROSTEST]-----------------------------------------------------------------------

[myworkcell_core.rosunit-unit_test_node/myworkcell_core_framework][passed]

SUMMARY
 * RESULT: SUCCESS
 * TESTS: 1
 * ERRORS: 0
 * FAILURES: 0
```

This means our framework is functional and now we can add usefull unit tests.

Note: You can also run tests directly from the command line, using the launch file we made above: *rostest myworkcell_core utest_launch.test*. Note that test files are not built using the regular *catkin build* command, so use *catkin run_tests myworkcell_core* instead.

Add stock publisher tests

1. The rostest package provides several tools for inspecting basic topic characteristics hztest, paramtest, publishtest. We'll add some basic tests to verify that the *fake_ar_publisher* node is outputting the expected topics.
2. Add the test description to the *utest_launch.test* file:

```
<test name="publishtest" test-name="publishtest" pkg="rostest" type="publishtest">
    <rosparam>
      topics:
        - name: "/ar_pose_marker"
          timeout: 10
          negative: False
        - name: "/ar_pose_visual"
          timeout: 10
          negative: False
    </rosparam>
</test>
```

3. Run the test:

```
catkin run_tests myworkcell_core
```

You should see:

Summary: 2 tests, 0 errors, 0 failures

Write specific unit tests

1. Since we will be testing the messages we get from the fake_ar_publisher package, include the relevant header file (in *utest.cpp*):

```
#include <fake_ar_publisher/ARMarker.h>
```

2. Declare a global variable:

```
fake_ar_publisher::ARMarkerConstPtr test_msg_;
```

3. Add a subscriber callback to copy incoming messages to the global variable:

```
void testCallback(const fake_ar_publisher::ARMarkerConstPtr &msg)
{
  test_msg_ = msg;
}
```

4. Write a unit test to check the reference frame of the ar_pose_marker:

```
TEST(TestSuite, myworkcell_core_fake_ar_pub_ref_frame)
{
    ros::NodeHandle nh;
    ros::Subscriber sub = nh.subscribe("/ar_pose_marker", 1, &testCallback);

    EXPECT_NE(ros::topic::waitForMessage<fake_ar_publisher::ARMarker>("/ar_pose_
↪marker", ros::Duration(10)), nullptr);
    EXPECT_EQ(1, sub.getNumPublishers());
    EXPECT_EQ(test_msg_->header.frame_id, "camera_frame");
}
```

5. Add some node-initialization boilerplate to the main() function, since our unit tests interact with a running ROS system. Replace the current main() function with the new code below:

```
int main(int argc, char **argv)
{
```

(continues on next page)

(continued from previous page)

```
  testing::InitGoogleTest(&argc, argv);
  ros::init(argc, argv, "MyWorkcellCoreTest");

  ros::AsyncSpinner spinner(1);
  spinner.start();
  int ret = RUN_ALL_TESTS();
  spinner.stop();
  ros::shutdown();
  return ret;
}
```

6. Run the test:

```
catkin run_tests myworkcell_core
```

7. view the results of the test:

```
catkin_test_results build/myworkcell_core
```

You should see:

```
Summary: 3 tests, 0 errors, 0 failures
```

4.2.3 Using rqt Tools for Analysis

In this exercise we will use rqt_console, rqt_graph and urdf_to_graphviz to understand behavior of the ROS system.

Motivation

When complicated multi-node ros systems are running it can be important to understand the interactions of nodes.

Information and Resources

Using a catkin workspace

Problem Statement

The Scan-N-Plan application is complete. We would like to further inspect the application using the various ROS rqt tools.

Guidance

`rqt_graph`: view node interaction

In complex applications, it may be helpful to get a visual representation of the ROS node interactions.

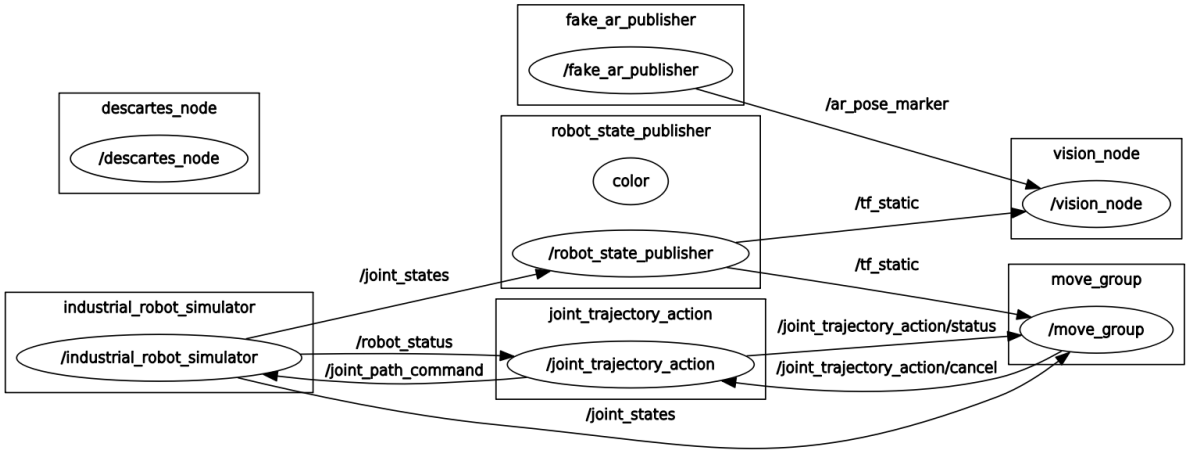
1. Launch the Scan-N-Plan workcell:

```
roslaunch myworkcell_support setup.launch
```

1. In a 2nd terminal, launch the rqt_graph:

```
rqt_graph
```

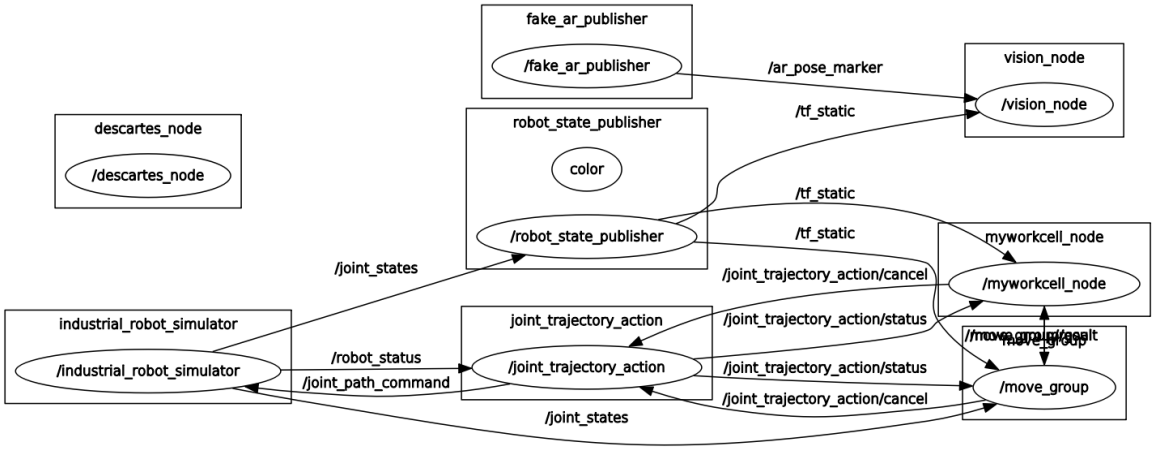
1. Here we can see the basic layout of our Scan-N-Plan application:



2. In a 3rd terminal, launch the descartes path planner.:

```
rosrun myworkcell_core myworkcell_node
```

1. You must update the graph while the node is running because the graph will not update automatically. After the update, we see our updated ROS network contains out myworkcell_node. Also, The myworkcell_node is publishing a new topic `/move_group/goal` which is subscribed by the move_group node.



`rqt_console`: view messages:

Now, we would like to see the output of the path planner. rqt_console is a great gui for viewing ROS topics.

1. Kill the rqt_graph application in the 2nd terminal and run rqt_console:

```
rqt_console
```

1. Run the path planner:

```
rosrun myworkcell_core myworkcell_node
```

1. The rqt_console automatically updates, showing the logic behind the path planner:

#	Message	Severity	Node	Stamp	Topics
#26	Done	Info	/myworkcell_node	07:16:40.4246286...	/attached_collision_obj...
#25	Inside goal constraints, stopped moving, return success for action	Info	/joint_trajectory_action	07:16:40.4239990...	/joint_path_command, /...
#24	Publishing trajectory	Info	/joint_trajectory_action	07:16:37.6905980...	/joint_path_command, /...
#23	Received new goal	Info	/joint_trajectory_action	07:16:37.6905615...	/joint_path_command, /...
#22	Got cart path, executing	Info	/myworkcell_node	07:16:37.6900420...	/attached_collision_obj...
#21	Traj size: 48 List size: 48	Info	/descartes_node	07:16:37.6881907...	/rosout
#20	Descartes found path with total cost: 6.95549	Info	/descartes_node	07:16:37.6881444...	/rosout
#19	Recieved cartesian planning request	Info	/descartes_node	07:16:36.0367572...	/rosout
#18	Inside goal constraints, stopped moving, return success for action	Info	/joint_trajectory_action	07:16:35.6227504...	/joint_path_command, /...
#17	Publishing trajectory	Info	/joint_trajectory_action	07:16:31.5230577...	/joint_path_command, /...
#16	Received new goal	Info	/joint_trajectory_action	07:16:31.5229876...	/joint_path_command, /...
#15	Filtered joint names to 6 joints	Info	/joint_trajectory_action	07:14:56.9458895...	/rosout
#14	SimpleSetup: Path simplification took 0.022507 seconds and changed from 4 to 21 states	Info	/move_group	07:16:31.4307904...	/execute_trajectory/fee...
#13	Solution found in 0.279170 seconds	Info	/move_group	07:16:31.4081734...	/execute_trajectory/fee...
#12	RRTConnect: Created 5 states (3 start + 2 goal)	Info	/move_group	07:16:31.4081367...	/execute_trajectory/fee...
#11	RRTConnect: Starting planning with 1 states already in datastructure	Info	/move_group	07:16:31.1293635...	/execute_trajectory/fee...
#10	Planner configuration 'manipulator' will use planner 'geometric::RRTConnect'. Additional configu...	Info	/move_group	07:16:31.1283151...	/execute_trajectory/fee...
#9	Planning attempt 1 of at most 1	Info	/move_group	07:16:31.1247921...	/execute_trajectory/fee...
#8	Combined planning and execution request received for MoveGroup action. Forwarding to plannin...	Info	/move_group	07:16:31.1242277...	/execute_trajectory/fee...
#7	Ready to take commands for planning group manipulator.	Info	/myworkcell_node	07:16:31.1184091...	/attached_collision_obj...
#6	Loading robot model 'myworkcell'...	Info	/myworkcell_node	07:16:29.9111193...	/joint_trajectory_action...
#5	Loading robot model 'myworkcell'...	Info	/myworkcell_node	07:16:29.8255205...	/joint_trajectory_action...
#4	part localized: pose: position:	Info	/myworkcell_node	07:16:29.7890571...	/joint_trajectory_action...
#3	Requesting pose in base frame: world	Info	/myworkcell_node	07:16:29.7855257...	/joint_trajectory_action...
#2	Attempting to localize part	Info	/myworkcell_node	07:16:29.7854464...	/joint_trajectory_action...
#1	ScanNPlan node has been initialized	Info	/myworkcell_node	07:16:29.2753531...	/rosout

`rqt_plot`: view data plots

rqt_plot is an easy way to plot ROS data in real time. In this example, we will plot robot joint velocities from our path plan.

1. Kill the rqt_console application in the 2nd terminal and run rqt_plot:

```
rqt_plot
```

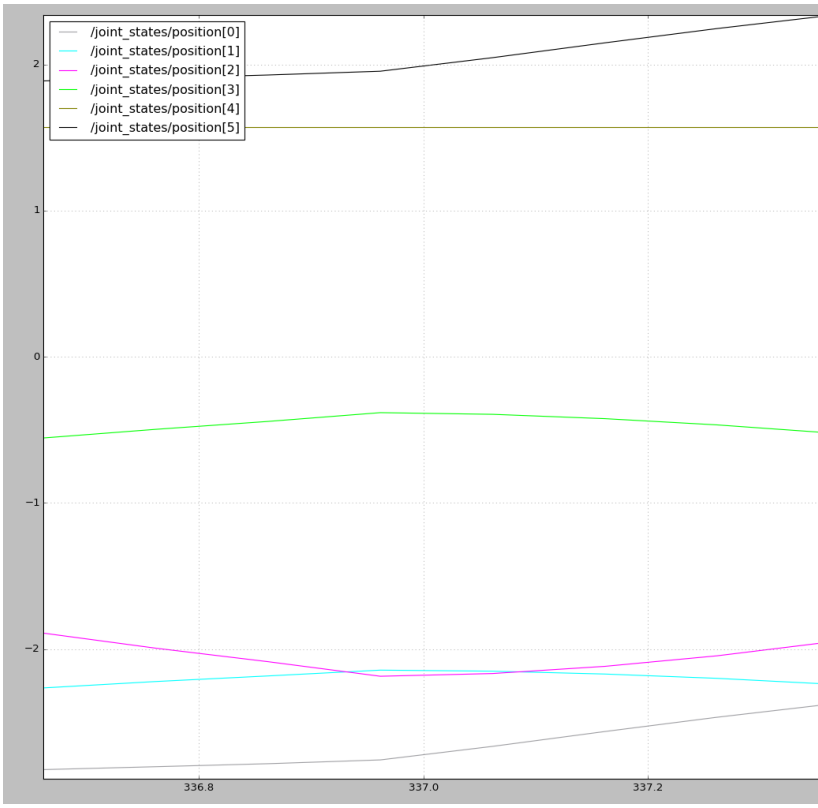
1. In the `Topic` field add the following topics:

```
/joint_states/position[0]
/joint_states/position[1]
/joint_states/position[2]
/joint_states/position[3]
/joint_states/position[4]
/joint_states/position[5]
```

1. Then run the path planner:

```
rosrun myworkcell_core myworkcell_node
```

1. We can see the joint positions streaming in real-time:



4.2.4 ROS Style Guide and ros_lint

Motivation

The ROS Scan-N-Plan application from Exercise 4.0 is complete, tested and documented. Now we want to clean up the code according to the style guide so other developers can easily understand our work.

Information and Resources

The Official ROS C++ Style Guide

Automated Style Guide Enforcement

Scan-N-Plan Application: Problem Statement

We have completed and tested our Scan-N-Plan program from Exercise 4.0 and we need to release the code to the public. Your goal is to ensure the code we have created conforms to the ROS C++ Style Guide.

Scan-N-Plan Application: Guidance

Configure Package

1. Add a build dependency on `roslint` to your `myworkcell_core` package's `package.xml`:

```
<build_depend>roslint</build_depend>
```

2. Add `roslint` to the `find_package(...)` command in the `CMakeLists.txt` file:

```
find_package(catkin REQUIRED COMPONENTS
 ...
 roslint
)
```

3. Invoke `roslint` from the `CMakeLists.txt` file

```
roslint_cpp()
```

Run roslint

1. To run `roslint`:

```
catkin build myworkcell_core --make-args roslint
```

2. The output of `roslint` for the ScanNPlan class from Exercise 4.0 indicates that there are several issues regarding formatting

```
/home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/myworkcell_node.cpp:0:␣
↪ No copyright message found.  You should have a line: "Copyright [year]
↪<Copyright Owner>"  [legal/copyright] [5]
/home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/myworkcell_node.cpp:9:␣
↪ Single-parameter constructors should be marked explicit.  [runtime/explicit]␣
↪[5]
/home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/myworkcell_node.
↪cpp:54:  Lines should be <= 120 characters long  [whitespace/line_length] [2]
/home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/myworkcell_node.
↪cpp:54:  At least two spaces is best between code and comments  [whitespace/
↪comments] [2]
/home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/myworkcell_node.
↪cpp:54:  Add #include <string> for string  [build/include_what_you_use] [4]
Done processing /home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/
↪myworkcell_node.cpp
/home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/vision_node.cpp:0:  No␣
↪copyright message found.  You should have a line: "Copyright [year] <Copyright␣
↪Owner>"  [legal/copyright] [5]
/home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/vision_node.cpp:12:  ␣
↪Single-parameter constructors should be marked explicit.  [runtime/explicit] [5]
/home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/vision_node.cpp:23:  ␣
↪Should have a space between // and comment  [whitespace/comments] [4]
Done processing /home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/vision_
↪node.cpp
Total errors found: 8
```

3. Once these issues are corrected, the output of `roslint` should look like this:

```
Done processing /home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/
↪myworkcell_node.cpp
Done processing /home/ros-industrial/ros/catkin_ws/src/myworkcell_core/src/vision_
↪node.cpp
Total errors found: 0
```

4.2.5 Docker AWS

Demo #1 - Run front-end Gazebo host and back-end in Docker

Setup workspace

Front-end (run on host and only contains gui)

in terminal 1

```
mkdir -p ./training/front_ws/src
cd ./training/front_ws/src
gazebo -v
git clone -b gazebo7 https://github.com/fetchrobotics/fetch_gazebo.git
git clone https://github.com/fetchrobotics/robot_controllers.git
git clone https://github.com/fetchrobotics/fetch_ros.git
cd ..
catkin build fetch_gazebo fetch_description
```

Back-end (run in container)

In this step, we will create a docker image that has the executables we need:

- run /bin/bash in the rosindustrial/core:indigo image then apt-get the package, commiting the result.
- run /bin/bash in the rosindustrial/core:indigo image then build the package from source, commiting the result.
- create a docker container using the fetch Dockerfile, which we will perform. https://gist.github.com/AustinDeric/242c1edf1c934406f59dfd078a0ce7fa

```
cd ../fetch-Dockerfile/
docker build --network=host -t rosindustrial/fetch:indigo .
```

Running the Demo

Run the front-end

Run the front end in terminal 1:

```
source devel/setup.bash
roslaunch fetch_gazebo playground.launch
```

Run the backend

There are multiple ways to perform this:

- run /bin/bash in the fetch container and manually run the demo node.
- run the demo node directly in the container, which is the method we will perform

Run the back end in terminal 2:

```
docker run --network=host rosindustrial/fetch:indigo roslaunch fetch_gazebo_demo demo.
↪launch
```

Demo #2 - Run front-end on a web-server and back-end in docker

start the environment

```
docker run --network=host rosindustrial/fetch:indigo roslaunch fetch_gazebo␣
↪playground.launch headless:=true gui:=false
```

run the gazebo web server:

```
docker run -v "/home/ros-industrial/roscloud/training/front_ws/src/fetch_gazebo/fetch_
↪gazebo/models/test_zone/meshes/:/root/gzweb/http/client/assets/test_zone/meshes/" -
↪v "/home/ros-industrial/roscloud/training/front_ws/src/fetch_ros/fetch_description/
↪meshes:/root/gzweb/http/client/assets/fetch_description/meshes" -it --network=host␣
↪giodegas/gzweb /bin/bash
```

then run the server:

```
/root/gzweb/start_gzweb.sh && gzserver
```

run the demo in terminal 3:

```
docker run --network=host fetch roslaunch fetch_gazebo_demo demo.launch
```

Demo #3 Robot Web Tools

In this demo we will run an industrial robot URDF viewable in a browser In terminal 1 we will load a robot to the parameter server

```
mkdir -p abb_ws/src
git clone -b kinetic-devel https://github.com/ros-industrial/abb.git
docker run -v "/home/ros-industrial/roscloud/training/abb_ws:/abb_ws" --network=host -
↪it rosindustrial/core:melodic /bin/bash
cd abb_ws
catkin build
source devel/setup.bash
roslaunch abb_irb5400_support load_irb5400.launch
```

in terminal 2 we will start the robot web tools:

```
docker run --network=host rosindustrial/viz:kinetic roslaunch viz.launch
```

in terminal 3 we will launch the webserver first we need to start a www folder

```
cp -r abb_ws/src/abb/abb_irb5400_support/ www/
```

```
docker run -v "/home/ros-industrial/roscloud/training/www:/data/www" -v "/home/ros-
↪industrial/roscloud/training/nginx_conf/:/etc/nginx/local/" -it --network=host␣
↪rosindustrial/nginx:latest /bin/bash
nginx -c /etc/nginx/local/nginx.conf
```

4.3 Session 7 - ROS2

4.3.1 ROS2 Basics Exercise

Motivation

Our goal for this exercise is to go through the basic ROS2 commands to understand how they work. We will not be writing any code during this exercise, instead we will just run through a few commands in the shell.

Workspaces and packages

1. Start by creating a new ROS2 workspace. As with catkin in ROS1, a workspace consists of a folder with a `src/` subdirectory.

```
mkdir -p ~/ros2_ws/src
```

2. Clone the ROS2 demos examples repositories which we will use to run some examples. When building existing packages from source, it is important to verify the code version as the most recent version may have changes incompatible with the current release. Here we checkout the repositories to the `dashing` release branch when cloned.

```
cd ~/ros2_ws/src
git clone -b dashing git@github.com:ros2/demos.git
git clone -b dashing git@github.com:ros2/examples.git
```

3. Install any required dependencies for the repositories using rosdep. This step is the same as in ROS1 since rosdep is installed as a system tool.

```
cd ~/ros2_ws
rosdep install --ignore-src --from-paths src/
```

Building packages

ROS2 uses `colcon` as the build tool for ROS packages. Colcon is installed separately from ROS distributions as a pure Python package. It is always available in the user's PATH without sourcing a workspace setup file.

1. Run `colcon -h` to see a short help summary and a list of verbs that can be used. Run `colcon build -h` to see the help description for the `build` verb.
2. Run `colcon list` and `colcon info` to see infomation about what packages are currently in the workspace. Colcon can be directed to ignore packages if a directory contains an empty file named `COLCON_IGNORE`.
3. Build the workspace. This will require a setup script to be sourced so Ament can find the required dependent ROS packages. Check your `.bashrc` file and either change or remove any line that sources a ROS1 setup file at the end of the file. Remember if you change your `.bashrc` to start a terminal for it to take effect.

```
source /opt/ros/dashing/setup.bash
colcon build
```

   - **Important**: Unlike catkin tools, colcon looks for packages and will create its output folders wherever you run it. Be sure you are in the workspace root before running it.
   - Colcon does not currently have a `clean` verb. To rebuild a workspace from scratch, you must remove the generated outputs manually: `rm -r build/ install/ log/`

4. Source the newly-built workspace. Inside the new `install/` folder will be both `setup.bash` and `local_setup.bash` files. The `setup.bash` will configure a terminal environment to see the workspace packages as well as the environment the workspace was built in. `local_setup.bash` will add the workspace packages to the current environment.
    - If you keep your `.bashrc` to source `/opt/ros/dashing/setup.bash` then you should only need to run `source install/local_setup.bash`
    - Otherwise, run `source install/setup.bash`

The `ros2` command

ROS2 replaces the set of `ros*` command line tools with subcommands of a single `ros2` command. Briefly we'll run through some of ones that may be useful for a developer.

`ros2 pkg`

Shows information about ROS packages visible in the current environment

- Use `ros2 pkg list` to see a list of all visible packages
- Use `ros2 pkg executables` to see a list of executables made available by a package. E.g., `ros2 pkg executables tf2_ros`
- Use `ros2 pkg prefix` to get the installation location of package

`ros2 run`

Runs executables provided by a package (`ros2 run <package_name> <executable_file>`)

- Open two terminals sourced to the `ros2_ws` workspace.
- In the first one, run `ros2 run demo_nodes_cpp listener`. Tab completion should be available so you don't have to type the full package and executable names.
- In the second, run `ros2 run demo_nodes_cpp talker`. You should now start to see a string message start to be printed repeatedly on both terminals. Leave both nodes running.

Note: You did not need to start a ROS master process before running nodes. The two nodes discovered each other on the network in a decentralized way as they started up.

`ros2 node`

Shows information about currently running ROS nodes.

- Open a third terminal sourced to the `ros_ws` workspace.
- Run `ros2 node list` to see a list of running nodes
- Run `ros2 node info /talker` to see the topics and services the talker node is using

ros2 topic

Shows information about topics currently used by one or more nodes.

- Run `ros2 topic list` to see currently published topics
- Run `ros2 topic info /chatter` to see information about the topic the talker node is publishing
- Run `ros2 topic echo /chatter` to locally subscribe to the `/chatter` topic and print it on the terminal
- Run `ros2 topic -h` for a list of more available subcommands

ros2 service/srv

Shows information about ROS services.

- Either stop the talker and listener nodes with `Ctrl-C` or open two more terminals.
- In the first terminal, run `ros2 run demo_nodes_cpp add_two_ints_server`
- In the second terminal, run `ros2 service list -t` to show currently available services and what type they are.
- Run `ros2 srv show example_interfaces/srv/AddTwoInts` to see the service definition of the `/add_two_ints` service. Notice the names and types of the fields used in the service request.
- Run `ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts '{a: 1, b: 2}'` to manually call the service. The fields of the service request must be set using a YAML dictionary that specifies the field names and values.

ros2 launch

Start an installed launch file (`ros2 launch <package_name> <launch_file>`)

- Close any currently running nodes
- Run `ros2 launch dummy_robot_bringup dummy_robot_bringup.launch.py`. Again, tab completion should be able to help here. Notice the `.py` suffix on the launch file, it's just a python script.

This starts five nodes that can inspected with the above command line tools. You might notice a `/joint_states` topic publishing changing values and a `/scan` topic publishing simulated 2D laser scanner data. Let's visualize them.

RViz

RViz has been renamed to `rviz2` and substantially rewritten for ROS2 but appears more-or-less the same from the user's point-of-view.

- In a free terminal, run `rviz2` while the dummy robot launch file is still running.
- Change the 'Fixed Frame' to 'world'
- Add a `RobotModel` display type. Expand the display options and set 'Description Source' to 'Topic' and 'Description Topic' to '/robot_description'. You should now see a robot moving on your screen.
- Add a `LaserScan` display and have it display the '/scan' topic. You should start to see 2D laser scan data appear as if the scanner were attached to the end of the robot. Getting intermittent transform errors is normal.

4.3.2 ROS1 to ROS2 porting

Introduction

Our goal for this exercise is to have you fully port a small ROS1 application into ROS2. We'll be using the basic training material from sessions 1 and 2 as the initial ROS1 application. This will provide some exposure to the differences in the basic tools of ROS development including publishers, subscribers, services, and parameters. We will be using exercise 2.3 as the starting point.

Workspace setup

1. Clone the ROS2 version of `fake_ar_publisher` into your ROS2 workspace

```
cd ~/ros2_ws/src/
git clone -b ros2 https://github.com/ros-industrial/fake_ar_publisher.git
```

2. Create empty ROS2 packages.

```
cd ~/ros2_ws/src/
ros2 pkg create myworkcell_core
ros2 pkg create myworkcell_support
```

3. Verify the workspace builds with `colcon build`
4. Set up a new project in Qt Creator with 'Build System' set to 'Colcon' and 'Distribution' set to '/opt/ros/dashing'. This is optional but highly recommended since you will get code autocompletion when editing C++ in the Qt Creator IDE. Verify that the workspace builds successfully from inside the IDE.
5. Copy the ROS1 code from the packages at `~/industrial_training/exercises/2.3/src/` into the empty ROS2 packages. We'll want everything copied over except `CMakeLists.txt` and `package.xml`, where we will be modifying the empty ROS2 versions instead. For more complex porting efforts, it will likely be more efficient to directly modify the ROS1 versions of these files.

Porting

We will more or less follow the same sequence that the ROS1 basic training takes in switching to ROS2.

Topic subscription

1. Open and start editing the `CMakeLists.txt` file in `myworkcell_core`
2. Package dependencies: Ament packages you depend on are now found directly using the CMake `find_package` command. Almost always the first invocation will be `find_package(ament_cmake)` which is required to make this package an ament package.

   Add additional dependencies for this application below this first `find_package` command:

```
find_package(rclcpp REQUIRED)
find_package(fake_ar_publisher_msgs REQUIRED)
```

3. Build the vision node: An executable is added in the usual CMake way. Rather than use `include_directories` and `target_link_libraries` to depend on other packages, a convenience macro is provided to depend on other ament packages.

```
add_executable(vision_node src/vision_node.cpp)
ament_target_dependencies(vision_node rclcpp fake_ar_publisher)
```

4. Install the vision node: Everything in ROS2 must have an install rule in order to be used at runtime.

```
install(TARGETS vision_node
    ARCHIVE DESTINATION lib/${PROJECT_NAME}
    LIBRARY DESTINATION lib/${PROJECT_NAME}
    RUNTIME DESTINATION lib/${PROJECT_NAME}
)
```

5. Above the final call to `ament_package()` add a line which indicates which ament packages this package depends on at runtime:

```
ament_export_dependencies(rclcpp fake_ar_publisher)
```

6. You may try to build the workspace again now and you should get a compile error about `ros/ros.h` not being found. We're now ready to port the C++ code. Open `vision_node.cpp` for editing.

7. Change the headers: ROS2 requires headers to end in .hpp instead of .h and includes for msgs and srv require the file name to file/msg/ and file/srv/ respectively in their names.

```
- #include <ros/ros.h>
+ #include <rclcpp/rclcpp.hpp>
- #include <fake_ar_publisher/ARMarker.h>
+ #include <fake_ar_publisher/msg/ar_marker.hpp>
```

8. Comment out the callbacks, the member variables, and the contents of the constructor. Change the `Localizer` class to inherit from `rclcpp::Node`.

```
class Localizer : public rclcpp::Node
{
public:
    Localizer()
    : Node("vision_node")
    {
       //...
    }
    //...
};
```

9. The main function can now be replaced with ROS2 equivalents of creating a node and spinning to wait for callbacks to be invoked:

```
int main(int argc, char* argv[])
{
    // This must be called before anything else ROS-related
    rclcpp::init(argc, argv);

    // Create the node
    auto node = std::make_shared<Localizer>();

    // Don't exit the program.
    rclcpp::spin(node);
}
```

10. Now we'll port the subscriber Replace the `ar_sub_` and `last_msg_` member variables with:

```
rclcpp::Subscription<fake_ar_publisher::msg::ARMarker>::SharedPtr ar_sub_;
fake_ar_publisher::msg::ARMarker::SharedPtr last_msg_;
```

11. Add the subscription creation to the constructor:

```
using namespace std::placeholders;

ar_sub_ = this->create_subscription<fake_ar_publisher::msg::ARMarker>("ar_pose_
↪marker", rclcpp::QoS(1),
  std::bind(&Localizer::visionCallback, this, _1));
```

and the callback that stores the message:

```
void visionCallback(fake_ar_publisher::msg::ARMarker::SharedPtr msg)
{
  last_msg_ = msg;
  RCLCPP_INFO(this->get_logger(), "Received ARMarker message");
}
```

12. Verify the workspace builds now and test the topic subscription. Run the publisher node and vision node in separate terminals:

```
ros2 run fake_ar_publisher fake_ar_publisher
ros2 run myworkcell_core vision_node
```

Interface generation

We'll now build the `LocalizePart` service type to use in the `myworkcell_core` package.

1. The `.srv` file syntax is basically unchanged, but we do have to modify the definition. Since ROS2 does not allow you to return a boolean from a service callback, we must track success and failure ourselves. Add a `success` member to the response in `LocalizePart.srv`:

```
#request
string base_frame
---
#response
bool success
geometry_msgs/Pose pose
```

2. Modify CMake and `package.xml` to build the service. In `CMakeLists.txt` add additional dependencies and a call to generate custom interfaces:

```
find_package(rosidl_default_generators REQUIRED)
find_package(geometry_msgs REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/LocalizePart.srv"
  DEPENDENCIES geometry_msgs
)
```

To use generated interfaces in the same package, another dependency must be added on built targets:

```
rosidl_target_interfaces(vision_node ${PROJECT_NAME} "rosidl_typesupport_cpp")
```

In `package.xml` make sure the `format` version is 3 and add the following tags:

```
<member_of_group>rosidl_interface_packages</member_of_group>
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
```

3. Make sure the workspace builds and you can now see the `myworkcell_core/srv/LocalizePart` service when running `ros2 srv list`.

Services

1. At the top of `vision_node.cpp` add/uncomment:

```
#include <myworkcell_core/srv/localize_part.hpp>
```

Then add the service server to the vision node, replacing the `ServiceServer` member variable with:

```
rclcpp::Service<myworkcell_core::srv::LocalizePart>::SharedPtr server_;
```

Add the service creation to the constructor:

```
server_ = this->create_service<myworkcell_core::srv::LocalizePart>("localize_part
↪",
  std::bind(&Localizer::localizePart, this, _1, _2));
```

and the service callback:

```
void localizePart(myworkcell_core::srv::LocalizePart::Request::SharedPtr req,
                  myworkcell_core::srv::LocalizePart::Response::SharedPtr res)
{
  // Read last message
  fake_ar_publisher::msg::ARMarker::SharedPtr p = last_msg_;

  if (!p){
    RCLCPP_ERROR(this->get_logger(), "no data");
    res->success = false;
    return;
  }

  res->success = true;
  res->pose = p->pose.pose;
}
```

2. Run both the `fake_ar_publisher` node and the `vision_node` and try manually calling the service.
3. Now we'll set up the application node to call the service. Add a new target to `CMakeLists.txt`:

```
add_executable(myworkcell_node src/myworkcell_node.cpp)
ament_target_dependencies(myworkcell_node rclcpp fake_ar_publisher_msgs)
rosidl_target_interfaces(myworkcell_node ${PROJECT_NAME} "rosidl_typesupport_cpp")
```

and add `myworkcell_node` to the existing `install` function call, after `vision_node`.

4. Open `myworkcell_node.cpp` to start porting it over. As done with the vision node, convert the `ScanNPlan` class to inherit from `rclcpp::Node` and rewrite the `main` function in the same way. For now, comment out the code that loads the 'base_frame' parameter and call the `start` function with a hardcoded `"world"`.

```
class ScanNPlan : public rclcpp::Node
{
public:
    ScanNPlan() : Node("scan_n_plan")
    {
        //...
    }
    //...
}

int main(int argc, char* argv[])
{
    //...
}
```

5. Replace the `vision_client` member variable with:

```
rclcpp::Client<myworkcell_core::srv::LocalizePart>::SharedPtr vision_client_;
```

   and create it in the constuctor with:

```
vision_client_ = this->create_client<myworkcell_core::srv::LocalizePart>(
↪"localize_part");
```

6. The `start` function will be fully replaced since it is quite a bit more complex. Since ROS2 services are now asynchronous, the client returns a `future` object when called and the developer is responsible for spinning until a response returns from the vision node. Read through the new `start` function here before replacing to make sure you understand everything that is happening.

```
void start(const std::string& base_frame)
{
    using namespace std::chrono_literals;

    // Need to wait until vision node has data
    rclcpp::sleep_for(1s);

    RCLCPP_INFO(this->get_logger(), "Attempting to localize part in frame: %s",␣
↪base_frame.c_str());

    // The vision client needs to wait until the service appears
    while (!vision_client_->wait_for_service(500ms)) {
        if (!rclcpp::ok()) {
            RCLCPP_ERROR(this->get_logger(), "client interrupted while waiting␣
↪for service to appear.");
            return;
        }
        RCLCPP_INFO(this->get_logger(), "waiting for service to appear...");
    }

    // Create a request for the LocalizePart service call
    auto request = std::make_shared<myworkcell_core::srv::LocalizePart::Request>
↪();
    // The base_frame that is passed in is used to fill the request
    request->base_frame = base_frame;

    auto future = vision_client_->async_send_request(request);
    if (rclcpp::spin_until_future_complete(this->get_node_base_interface(),␣
↪future) != rclcpp::executor::FutureReturnCode::SUCCESS)
```

(continues on next page)

(continued from previous page)

```
    {
        RCLCPP_ERROR(this->get_logger(), "Failed to receive LocalizePart service␣
↪response");
        return;
    }

    auto result = future.get();
    if (! result->success)
    {
        RCLCPP_ERROR(this->get_logger(), "LocalizePart service failed");
        return;
    }

    RCLCPP_INFO(this->get_logger(), "Part Localized:  w: %f, x: %f, y: %f, z: %f",
            result->pose.orientation.w,
            result->pose.position.x,
            result->pose.position.y,
            result->pose.position.z);
}
```

7. Verify everything builds and works

Parameters

1. Parameters work slightly differently than in ROS1. The most significant change is the parameter declaration mechanism, which, when given a default value, ensures that a parameter will exist under the given name which matches the type of the default. Modify the `main` function to declare and get the `base_frame` parameter:

```
auto node = std::make_shared<ScanNPlan>();

// String to store the base_frame parameter after getting it from the Node's␣
↪parameter client
std::string base_frame;
node->declare_parameter("base_frame", "");
node->get_parameter("base_frame", base_frame);

if (base_frame.empty())
{
    RCLCPP_ERROR(node->get_logger(), "No 'base_frame' parameter provided");
    return -1;
}

// Call the vision client's LocalizePart service using base_frame as a parameter
node->start(base_frame);
```

2. Unfortunately, parameters can not currently be set on the command line when using `ros2 run`. To test that the parameter is set, you must create and save a YAML file:

```
/**:
  ros__parameters:
    base_frame: world
```

Then load this file when running the node:

```
ros2 run myworkcell_core myworkcell_node __params:=my_params_file.yaml
```

Launch file

1. Currently, the `myworkcell_support` package only contains a launch file for starting the three required nodes. We will port it to a ROS2 python launch script.
2. Start by creating a new file `workcell.launch.py` under the `launch/` directory.
3. Modify the contents to add the required python imports and the required function which will return the launch configuration description:

```
from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
    print('Workcell is launching...')
```

4. Each node in the XML launch file will correspond to a `Node` launch 'action' in the new launch description. The named arguments used in the action constructor should have clear meaning. You'll need one for the `fake_ar_publisher`:

```
fake_ar_pub_node = launch_ros.actions.Node(
    node_name='fake_ar_publisher_node',
    package='fake_ar_publisher',
    node_executable='fake_ar_publisher_node',
    output='screen',
    parameters=[{'x': -0.6},
                {'y': 0.4},
                {'z': 0.1},
                {'camera_frame': 'camera_frame'}],
)
```

one for the `vision_node`:

```
vision_node = launch_ros.actions.Node(
    node_name='vision_node',
    package='myworkcell_core',
    node_executable='vision_node',
    output='screen',
)
```

and one for `myworkcell_node`:

```
myworkcell_node = launch_ros.actions.Node(
    node_name='myworkcell_node',
    package='myworkcell_core',
    node_executable='myworkcell_node',
    output='screen',
    parameters=[{'base_frame': 'world'}],
)
```

Notice the `parameters` argument must be set with a list of dictionaries. Another common argument not used here is `arguments`, set with a list of command line arguments that the executable will start with.

5. Now bundle the three nodes in a `LaunchDescription` and return it from the function:

```
return LaunchDescription([fake_ar_pub_node, vision_node, myworkcell_node])
```

6. Add an `install` call to `CMakesList.txt` to install the launch file:

```
install(DIRECTORY launch
  DESTINATION share/${PROJECT_NAME}/
)
```

7. Modify `package.xml` to show the runtime dependency on the packages

```
<exec_depend>fake_ar_publisher</exec_depend>
<exec_depend>myworkcell_core</exec_depend>
```

8. After building the workspace test the launch file with:

```
ros2 launch myworkcell_support workcell.launch.py
```

   Note that the file loaded by `ros2 launch` resides in the `install/` directory and will not update if you edit the version in the `src/` space. To avoid having to rebuild the workspace to see changes in the launch script, you can run `colcon build` with the `--symlink-install` option so the file in the install space is just a link back to the source space. This build option is also useful when developing ROS python code in general.

4.3.3 ROS1-ROS2 Bridge Demo

Introduction

This is a system integration exercise to demonstrate operation of the ROS1-ROS2 topic and service bridge. Using the bridge does not require anything different when developing either ROS1 or ROS2 software and so we will not worry about writing code for this exercise.

This demo is an example of a system where a ROS2 application needs to call a planner that only exists as a ROS1 package. Specifically, this demo is calling the Descartes motion planner, as seen in exercise 4.1. On the ROS1 side, services are provided to generate motion plans and to execute them. In order to use these custom services, the bridge needs to be recompiled with the definitions in the build environment, which is the bulk of the complexity in this exercise.

Building the Demo

This exercise uses the ROS1 bridge to call ROS nodes from ROS2 nodes and therefore the build procedure is somewhat involved.

Create a ROS workspace for exercise 4.1

1. Create a catkin workspace for the exercise 4.1 ROS packages and dependencies

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
```

2. Create a symlink to exercise 4.1 in the training repo

```
cd ~/catkin_ws/src
ln -s ~/industrial_training/exercises/4.1/src demo
```

3. Clone additional dependencies

```
git clone https://github.com/ros-industrial-consortium/descartes.git
git clone https://github.com/ros-industrial/universal_robot.git
git clone https://github.com/ros-industrial/fake_ar_publisher.git
```

4. Source and build exercise

```
. /opt/ros/melodic/setup.bash
catkin build
```

Create the ROS2 workspace

1. Create a colcon workspace for the exercise 7.2 ROS packages and dependencies

```
mkdir -p ~/colcon_ws/src
cd ~/colcon_ws/src
```

2. Create a symlink to exercise 7.2

```
ln -s ~/industrial_training/exercises/7.2/src demo
```

3. Source and build

```
cd ~/colcon_ws/src
. /opt/ros/dashing/setup.bash
colcon build
```

Build the ROS1 Bridge

1. Create the ros1_bridge workspace. We build the bridge in a separate workspace because it needs to see both ROS1 and ROS2 packages in its environment and we want to make sure our application workspaces only see the packages from the distribution they are in.

```
mkdir -p ~/ros1_bridge_ws/src
cd ~/ros1_bridge_ws/src
```

2. Clone the ROS1 bridge into your ROS2 workspace, selecting the branch that matches your ROS release

```
git clone -b dashing https://github.com/ros2/ros1_bridge.git
```

3. Source ROS1 and ROS2 setup bash files. This is one of the only times you'll want to mix setup files from different ROS distributions.

```
. ~/catkin_ws/devel/setup.bash
. ~/colcon_ws/install/setup.bash
```

   You should see a warning about the `ROS_DISTRO` variable being previously set to a different value. Now the environment contains references to both distributions which can be verified by observing the CMake path:

```
echo $CMAKE_PREFIX_PATH | tr ':' '\n'
```

4. Build the bridge. This may take a while since it is creating mappings between all known message and service types.

```
colcon build --packages-select ros1_bridge --cmake-force-configure --cmake-args -
↪DBUILD_TESTING=FALSE
```

5. List the mapped message and service pairs

```
source install/local_setup.bash
ros2 run ros1_bridge dynamic_bridge --print-pairs
```

This should list the custom `myworkcell_msgs(ROS2) <-> myworkcell_core` mapped services

Run the Demo

Run the ROS nodes

1. In terminal sourced to your ROS catkin_ws workspace start the roscore

```
roscore
```

2. In another sourced terminal, run the following launch file:

```
roslaunch myworkcell_support ros2_setup.launch
```

This will launch the motion planning and motion execution nodes is ROS. Rviz will come up as well.

Run the ROS1 bridge

1. In your ros1_bridge_ws ROS2 workspace, source the workspace if you haven't

```
cd ~/ros1_bridge_ws
source install/setup.bash
```

2. Export the *ROS_MASTER_URI* environment variable

```
export ROS_MASTER_URI=http://localhost:11311
```

3. Run the bridge

```
ros2 run ros1_bridge dynamic_bridge
```

Run the ROS2 nodes

1. Open a new terminal and source your ROS2 workspace

```
cd ~/colcon_ws
source install/setup.bash
```

2. Run the python launch file that starts the ROS2 nodes for this application

```
ros2 launch myworkcell_core workcell.launch.py
```

If the program succeeds you should see the following output:

[myworkcell_node-2] Got base_frame parameter world[myworkcell_node-2] Waiting for client /localize_part[myworkcell_node-2] Waiting for client /plan_path[myworkcell_node-2] Waiting for client /move_to_pose[myworkcell_node-2] Waiting for client /execute_trajectory[myworkcell_node-2] Found all services[myworkcell_node-2] Requesting pose in base frame: world[myworkcell_node-2] Part localized[myworkcell_node-2] Planning trajectory[myworkcell_node-2] Executing trajectory[myworkcell_node-2] Trajectory execution complete

You should also see the robot moving accordingly in the Rviz window.

Issues:

- **Problem**: The application only runs successfully once, the next run will fail due to a tf lookup operation in the vision_node. This likely happens due to the bridge removing the bridging for all `/tf` topics after the application ends. Solution:

  Solution: Restart the `ros1_bridge` and then the `workcell.launch.py` application.