

Введение

В этой теме мы проверим работоспособность установленного ROS и чуточку поуправляем черепашкой.

Содержание

- [Содержание](#)
- [Проверяем, активировано ли пространство](#)
- [Hello ROS](#)

С этого момента мы считаем, что установлен пакет `ros-noetic-desktop-full` и активировано пространство ROS в системе.

Информацию по установке можете посмотреть в [разделе FAQ](#)

Проверяем, активировано ли пространство

Первым делом проверим установленную версию ROS:

```
echo $ROS_DISTRO
```

На экране должно появиться: `noetic`.

Теперь проверим все переменные, которые относятся к ROS:

```
env | grep "^\$ROS_"
```

Вот и все переменные, которые начинаются с "ROS_":

```
ROS_DISTRO=noetic
ROS_ETC_DIR=/opt/ros/noetic/etc/ros
ROS_PACKAGE_PATH=/opt/ros/noetic/share
ROS_PYTHON_VERSION=3
ROS_VERSION=1
ROS_ROOT=/opt/ros/noetic/share/ros
ROS_MASTER_URI=http://localhost:11311
```

Замечательно, если видишь подобный вывод, значит все готово к началу! Поехали тестить!

Hello ROS

Теперь попробуем запустить какой-нибудь пакет. Для этого установили Turtlesim.

Для его запуска понадобится 3 окна терминала:

- В первом окне запускаем ядро ROS командой

```
roscore
```

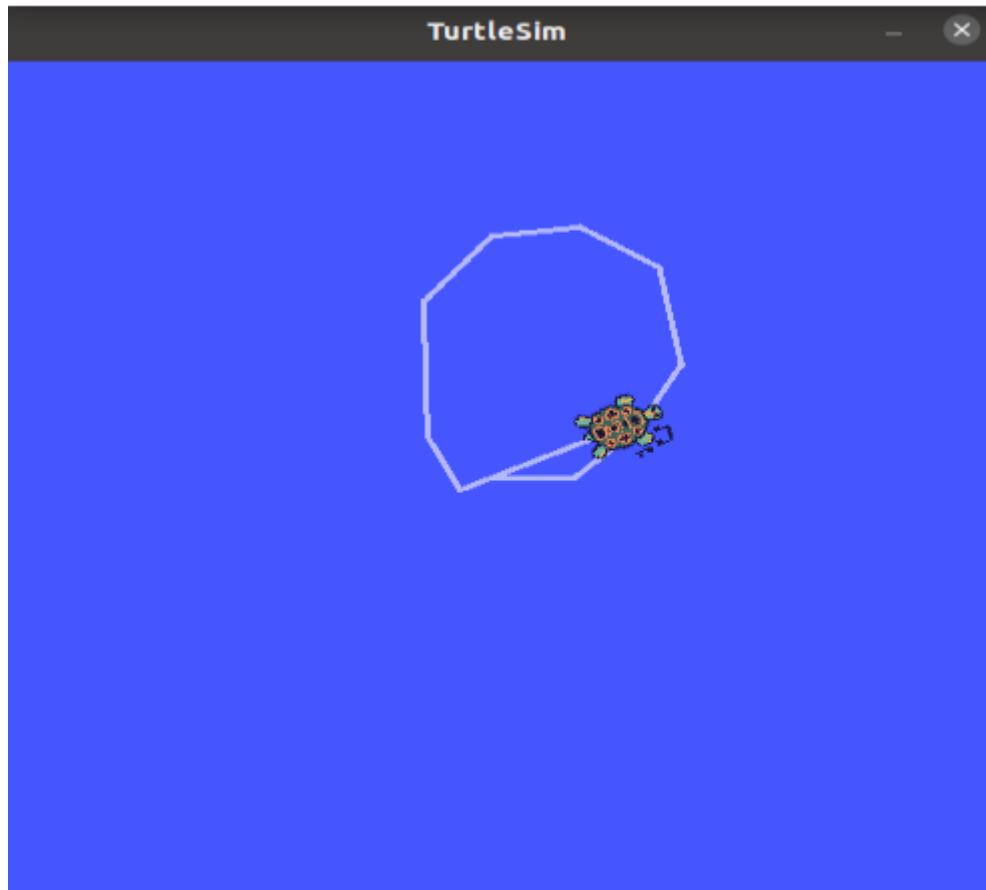
- В втором окне запустим программу с черепашкой

```
rosrun turtlesim turtlesim_node
```

- В третьем окне запускаем программу для управления ею с клавиатуры

```
rosrun turtlesim turtle_teleop_key
```

Мы должны увидеть черепашку на синем фоне:



Передвинь терминал так, чтобы фокус был на терминале с teleop (управление) программой и было видно окно с черепашкой. Нажимай кнопки стрелок и управляемте черепашкой.

Если все получилось, то УРА. Все установилось и работает 🎉. Пора переходить к самому вкусному!

Наверное, хорошо будет, если тебе еще подскажут, как выключать программы =) Говорим:
Сначала жмите Ctrl, а затем букву С - такие комбинации будут показываться как Ctrl+C.

Основы работы в Shell

Содержание

- Содержание
- Начнём, пожалуй
- Чуть поговорим о файловой системе в Linux
- Основные команды работы с файлами
 - ls
 - mkdir
 - cd
 - Больше о перемещении - relative vs. absolute
 - А как двигаться назад (наверх)?
 - mv
 - nano
 - Vim/NeoVim
 - rm
- Скрипты shell
 - Переменные
 - Исполняемость
- Переменные окружения
 - Сессия терминала
 - Импорт переменных из скрипта
 - Точка входа терминала
- Команды для обработки вывода
- Чему научились?
- Ресурсы

Начнём, пожалуй

Привет! В этой теме пройдёмся по основным моментам, которые нужно знать, чтобы безболезненно пользоваться [командной оболочкой](#) Shell (Bourne Shell).

Вообще, в Ubuntu оболочка по-умолчанию - Bash (Bourne again shell). Существует и куча других оболочек, например, [zsh](#), [fish](#), [xsh](#) и другие. Главная суть в том, что все они реализуют набор команд, с помощью которых можно управлять операционной системой.

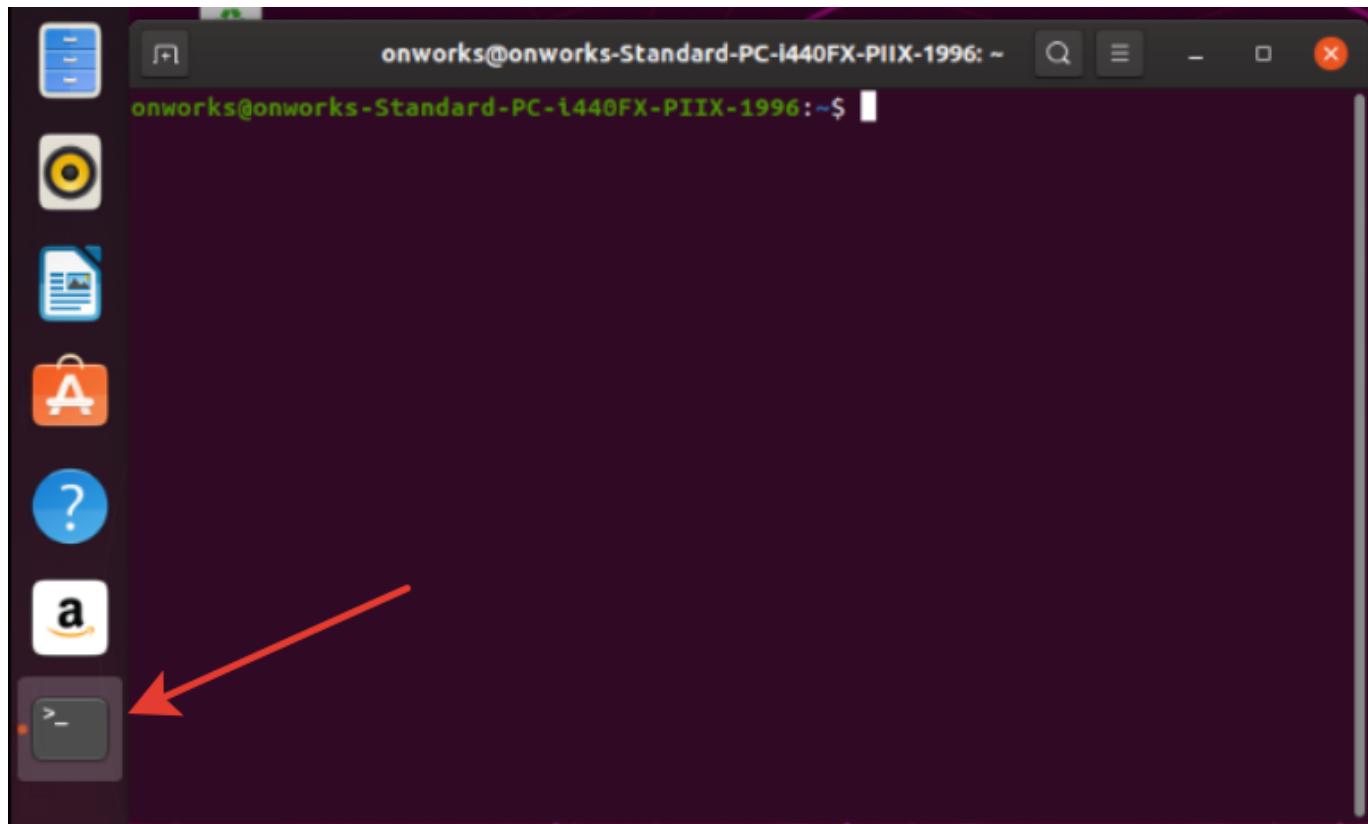
Важно, [POSIX-совместимых](#) команд - это стандарт такой =)

На самом деле, когда в файловом менеджере мы перетаскиваем мышкой папку или файлы, удаляем что-то, делаем какие-то действия с помощью графического интерфейса (GUI - Graphical User Interface), под капотом (внутри системы) исполняются те или иные команды CLI (Command Line Interface).

Не всегда напрямую те, что мы разберём, но логика такова, что вся графика - это надстройка над рядом базовых действий, которые можно сделать командами в Shell.

Поэтому, сегодня наша **цель** - разобраться как работать с командной строкой!

Поехали, открываем терминал (находим в меню приложений **Terminal**) и видим это:



У тебя может быть другое название зелёными буквами и название терминала =)

Это и есть терминал, в котором мы будем вводить наши команды. Давайте начнём наше знакомство с первой команды:

```
pwd
```

pwd - команда вывода директории, в которой находимся сейчас

В результате увидите **/home/user** вывод.

Отлично! Простая первая команда удалась, но мы отклонились от традиций, обычно же знакомство начинается с "Hello World!", не будем упускать возможность и познакомимся с командой вывода **echo**:

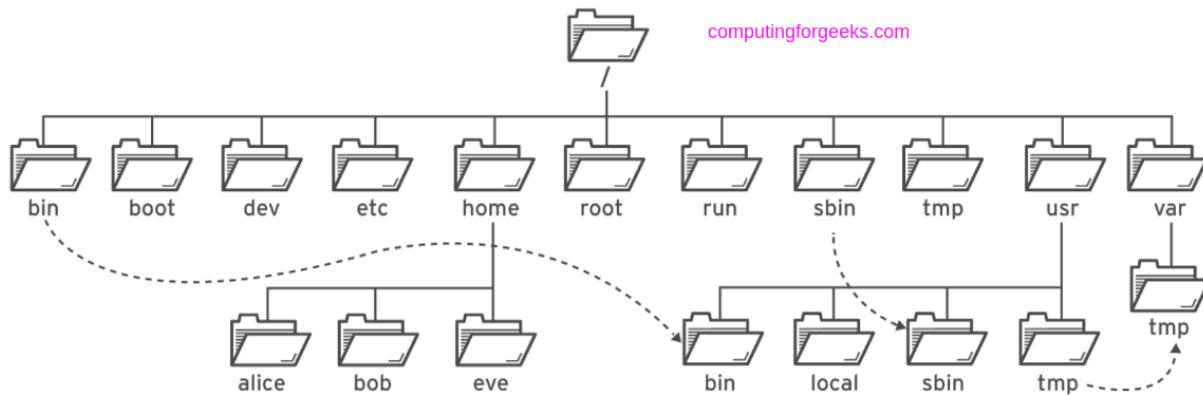
```
echo "Hello World!"
```

Ю-xy! Наш первый Hello-World! Поздравляю!

Чуть поговорим о файловой системе в Linux

Давай подробнее рассмотрим, почему путь выглядит именно так и что это за символы '**/?**'

Вся файловая система в Linux устроена в виде дерева. Вот, взгляните:



Подробнее про каждую директорию кратко расписано [здесь](#)

По сути, путь в виде строки '/' - это путь до корня файловой системы. В корне лежат все остальные папки, а в них лежат другие подпапки.

Так как нам интерпретировать путь `/home/user`? В корне лежит директория `home`, внутри которой есть директория `user`, внутри которой мы и находимся.

На самом деле в Windows используются такие же пути, но там вместо корня диски (`C:\`, `D:\` и т.д.), а папки разделяет символ `\`.

Основные команды работы с файлами

Теперь, когда мы знаем, как вывести директорию, в которой мы находимся и понимаем, как устроены папки и файлы в Linux, пора научиться ими управлять!

ls

Самая первая команда, которой стоит научиться - это команда вывода файлов и директорий в конкретной папке. Для этого используем `ls`:

```
ls
# Вывод: Downloads Pictures и т.д.
```

В зависимости от языка названия папок могут меняться в системе.

Отлично, но что, если мне хочется больше информации о папках и директориях? Вот тут мы встречаемся с **опциями команд**! Опции - это расширения команд, которые позволяют задать желаемое поведение (изменить под себя). Ведь удобно вывести просто список директорий и папок, но что если нужно больше информации? Используем опцию `-l`!

```
ls -l
```

и видим

```
total 40
drwxr-xr-x 15 user user 4096 Aug  6 23:56 Downloads
...
```

Воуу, сколько информации. Тут и дата последнего редактирования, и пользователь, владеющий файлами и папками и много другого, но мы пока сконцентрируемся на том, что это крутая опция!

У большинства команд есть опция `-h` (или длинная опция `--help`), которая выводит информацию по всем возможным настройкам утилиты.

💪 Выведи помощь к команде `ls` и прочитай, какие ещё есть опции у неё.

mkdir

Хорошо, выводить информацию полезно, но ещё полезнее научиться управлять файлами и папками. Следующая команда `mkdir` ("make directory") - позволит создавать директории.

💪 Создай папку, введя `mkdir super_dir` или другое имя вместо "super_dir" в качестве **аргумента** команды. Проверь в `ls` и файловом менеджере, что папка создана.

Аргумент - дополнительная информация, которая передаётся после команды. Опция отличается наличием символа "-".

Вы смогли сами создать папку? Крууутооо! Погнали внутрь!

cd

Так, мы папку создали и даже смогли зайти в неё в файловом менеджере, но как нам передвигаться в терминале? Всё просто! Для этого есть команда `cd` ("change directory").

💪 Перейди в директорию командой `cd super_dir`. Выведи с помощью `pwd` нынешний путь и проверь, что переход удачен.

💪 А теперь вопрос на подумать, почему, после первого выполнения перехода в "super_dir" второй уже не работает? А если открыть терминал заново, то снова, один раз работает, а другой раз нет?

Так, мы освоили способ перемещения, но ведь всё подкрепляется практикой, пора создать целую структуру.

💪 Создай следующую структуру папок и проверь себя в файловом менеджере. Убедитесь, что вы можете перемещаться по всем этим директориям без проблем.

- `super_dir`
 - `fruits`
 - `apple`
 - `orange`
 - `pear`
 - `vegetables`
 - `lettuce`

- onion
- cucumber
- spinach

Больше о перемещении - relative vs. absolute

Обратим внимание, что два раза команду не выполнить, так как при переходе в папку "super_dir" мы уже находимся в ней. То есть, команда `cd super_dir` - это переход в директорию из нынешней директории. Если внутри `super_dir` нет второй с тем же именем, то и команда несколько раз работать не будет.

Думаю понятно, что как в случае `mkdir`, так и в случае `cd` требуется аргумент, которым является имя для создания/перехода. Этим аргументом у нас являлась директория "super_dir" - работа с директорией из нынешней папки.

Так мы познакомились с относительным путем.

Относительный (Relative) путь - путь в файловой системе от нынешнего положения. Может быть `super_dir` или `./super_dir`. `.` - это символ, который при использовании в пути можно заменить на "отсюда" ("current directory").

Короче, если хотим создавать и делать что-то из нынешней директории, то пользуемся **относительными** путями.

А что если теперь выполнить команду `cd /home/user/super_dir` (обрати внимание, у тебя вместо `user` в `pwd` могло быть другое имя пользователя)? Попробуй! Несколько раз подряд? 🤪

Интересно, а такая команда выполняется, почему?

Всё просто, в этом случае мы задаём **абсолютный** путь в файловой системе:

Абсолютный (Absolute) путь - путь в файловой системе от "/" (корня).

Абсолютным путём не всегда удобно пользоваться, но в некоторых ситуациях он полезен, поэтому мотаем на ус.

А как двигаться назад (наверх)?

Если так подумать, то после создания нашей базы знаний фруктов и овощей (задачка про папки с `fruits/vegetables`) мы умеем только двигаться вперёд с помощью `cd` и относительного пути.

После изучения абсолютного пути, мы можем вернуться в корень нашей базы (`cd /home/user/super_dir`) и снова двигаться вглубь. Но что если я хочу из папки `orange` вернуться в папку `fruits`?

Да можно сделать `cd /home/user/super_dir/fruits`, но смотри дальше, там будет фокус с точками!

Что, если я скажу, что `.` - это нынешняя директория (можешь даже попробовать `cd . =)`), а директория выше по дереву - это, погоди-погоди, **две точки**!

Да, именно "... определяет путь до папки выше по дереву. (В некоторых интерпретаторах "... - это подъём на два уровня выше).

Итого, если тебе нужно передвинуться из `apple` папки в папку `fruits` - просто делаешь `cd ..` и вот, цель достигнута!

💪 Ну всё, теперь ты умеешь просто профессионально двигаться в командной строке! Погнали, зайди в каждую папку, выйди из неё и переместись в следующую! Время путешествий!

mv

Так, мы подошли к моменту, когда в нашей базе мы не хотим иметь лук (onion), не нравится он нам, а поменяем его на тыкву!

Синтаксис команды для переименования папок и файлов:

```
mv <оригинальное имя> <новое имя>
```

Например, `mv super_dir new_super_dir` переименует нашу базовую директорию. Делать это не обязательно, пойдём сразу практиковаться!

💪 Переименуй лук в тыкву (rumpkin). Также, переименуй один любой фрукт в любой другой фрукт.

Такс, такс, базу мы подредактировали, молодцы!

nano

Папки это хорошо, но мы не за тем учимся терминалу, чтобы просто создавать папки! Сейчас мы очень быстро научимся создавать и редактировать текстовые файлы!

А с учётом того, что исходники - это текстовые файлы, то это очень полезный навык!

Переходим в директорию `super_dir` (это вы уже умеете, так что сами) и создаём там текстовый файл с описанием нашей базы фруктов и овощей:

```
nano description.txt
```

Откроется редактор нового файла:

The screenshot shows a terminal window with the title "GNU nano 4.8" and the file name "description.txt". The editor interface is dark-themed. At the top, there's a menu bar with options like "File", "Edit", "Search", "Text", "Help", and "About". Below the menu is a toolbar with icons for "Get Help", "Write Out", "Where Is", "Cut Text", "Justify", "Cur Pos", "Undo", "Exit", "Read File", "Replace", "Paste Text", "To Spell", "Go To Line", "Redo", "Mark Text", and "Copy Text". The main area of the editor is currently empty.

Работать в нём просто, двигаете курсор с помощью стрелок указателей и набираете текст.

Чтобы выйти, нам помогает подсказка "^X Exit". Символ "^" означает клавишу "Ctrl", так что для выхода нажимаете "Ctrl+X".

Но перед этим надо сохранить файл, "^O Write Out", то есть, "Ctrl+O".

Если не сохранить, но выйти - нам напомнят о том, что файл не сохранен! Удобно!

В nano ещё куча разных hotkey комбинаций, например, вырезать всю строку, поиск в файле и т.д.
Больше инфы найдете [здесь!](#)

Обычно в практике nano не используется постоянно, но это удобно, если нужно быстро подредактировать/посмотреть какой-то файл, а в редакторе типа VSCode отдельно открывать долго.

Если хотите просто создать пустой файл и не редактировать его, то можно воспользоваться командой `touch another_description.txt` (имя файла своё), но на практике такое нужно редко, так что nano рулит.

💪 Самое время в каждой папке фрукта и овоща написать краткое описание с помощью nano. Например, в `description.txt` внутри "apple" папки пишем "Красное и круглое". Попробуйте создавать и редактировать файлы с помощью относительных и абсолютных путей. Так же, помимо nano есть много других популярных текстовых редакторов, например Vim или Emacs, не бойтесь экспериментировать и выбирать инструменты себе по душе.

Vim/NeoVim

Еще один популярный текстовый редактор, менее дружелюбен к пользователю чем nano, но обладает большим количеством плагинов, позволяющих собрать из него почти полноценную IDE. Обычно имеет в комплекте с пакетом самого редактора идет полноценный учебник по использованию - Vimtutor. Там описываются все основные горячие клавиши, лайфхаки по общему принципу использования и т.п. Если сказать коротко то Vim имеет три режима работы

- Командный

- Режим вставки
- Визуальный При открытии файла вы первоначально находитесь в командном режиме

```
vim file.txt
```

Чтобы перейти в режим вставки достаточно нажать `i`, после чего вы сможете печатать. После внесения изменений нажимаем "**ESC**" для выхода в "**командный режим**", вводим ":" (двоеточие), вводим "**wq**" (write-quit) - для сохранения изменений и выхода; "**q!**" - для выхода без сохранения, и нажимаем "**Enter**". Горячие клавиши и особенности работы с текстом сможете узнать в Vimtutor, если вам приглянется этот текстовый редактор 😊

`rm`

Стоит упомянуть и о возможности удаления файлов и папок с помощью команды `rm`.

Чтобы удалить файл, достаточно написать `rm description.txt`, то есть, просто указав путь до файла в качестве аргумента.

Для удаления папок используем опцию `-r`, которая проходит внутрь папок и удаляет файлы и папки внутри (рекурсивно).

Скрипты shell

Смотри, мы сделали уже кучу действий и каждую команду набирали вручную. А что, если нам потребуется выполнить эти действия на другом компьютере повторно? Например, снова создать структуру файлов, которую мы делали ранее.

Вспоминать и по памяти (или с листочка) снова их последовательно делать - не вариант! Погнали знакомиться со скриптами!

Скрипты - это просто набор команд, который выполняется последовательно. Есть ещё широкоизвестное название "макросы", но тут им не пользуются.

Чтобы сделать Bash скрипт, достаточно создать файл с расширением `.sh`.

💪 Перейди в папку "super_dir" и внутри создай файл `my_first_script.sh`. Внутри скрипта напиши всего одну команду - `echo "Hello World!"`.

Отлично, у нас есть скрипт, который приветствует, но как его вызвать, чтобы он выполнил команды? Всё очень просто, явно вызываем интерпретатор:

```
bash my_first_script.sh
```

Великолепно! Вот мы и научились создавать скрипты! Но мы на этом не останавливаемся - дальше интереснее!

Переменные

Внутри bash существует возможность использовать переменные. Определяются они максимально просто, пишем имя переменной, знак "=", значение.

 Важно, что пробелов между именем, "=" и значением быть не должно!!!

Например, чтобы вывести "World" через переменную делаем так:

```
WORLD_STRING="world"  
echo "Hello $WORLD_STRING!"
```

Заметили? В первой строке определили переменную, а в третьей к ней обратились. Обращение к переменной делается через символ "\$".

Но что, если мне надо вывести прямо символ "\$"? В таком случае ведь всё после этого символа будет использовано как имя переменной? Агась, но для этого можно использовать символ экранирования "\". Например echo "Million \\$s i have"

 В своём скрипте напиши код, который будет выводить строку "Hello World! My name is User!", где вместо "User" будет подставляться твоё имя.

Исполняемость

В ряде случаев мы не можем явно указать вызов интерпретатора `bash script.sh`.

Как быть? Как нам тогда запускать скрипты, ведь это так удобно!

Что если я скажу тебе, что можно прописать интерпретатор прямо в файле! Как это сделать? Сейчас покажу, прописывай это в начало своего скрипта самой первой строкой:

```
#!/usr/bin/bash
```

Такая строка называется **shebang** и такую штуку можно встретить не только в Bash скриптах. Главное, что это только одна часть исполняемости. Мы пока только прописали интерпретатор, а в Linux нужно сделать ещё один момент, дать файлу права на исполнение.

Shebang имеет синтаксис `#!interpreter [optional-arg]`. **interpreter** должен быть абсолютным путём до интерпретатора.

```
chmod +x my_first_script.sh
```

Команда `chmod` позволяет менять "права" у файла, например, можно сделать файл "read-only", или (как в нашем случае) сделать скрипт исполняемым.

После этого выведи `ls -l my_first_script.sh` и в строке разрешений будет видна буква "x" - это значит, что файл стал исполняемым!

Теперь, после этих незамысловатых магических действий (ещё раз, в файл прописали интерпретатор и сделали файл исполняемым) мы можем вызывать скрипт всего лишь вот так:

```
./my_first_script.sh  
# или по абсолютному пути: /home/user/super_dir/my_first_script.sh, но это обычно  
неудобно =)
```

В результате мы должны увидеть всё то же самое, что видели при вызове `bash my_first_script.sh`!

Если получилось - круто! Таким образом, можно сделать любой файл исполняемым, главное прописать, каким интерпретатором исполняемым.

Если прочитаете статью про shebang, то увидите, что лучшей практикой является использовать `#!/usr/bin/env bash`.

Переменные окружения

Помнишь, буквально недавно мы разобрались, что в bash есть переменные. Но те переменные, что есть в скрипте, существуют только при вызове скрипта.

💪 Если сомневаетесь, попробуйте определить переменную `MY_SCRIPT_VAR`, вызвать скрипт с ней и после вызова проверить, есть ли она в терминале командой `echo "$MY_SCRIPT_VAR"`.

Думаю, ты удивишься, если мы представим, что весь терминал - это один большой долго исполняющийся скрипт =)

Это всё к чему? В нашем терминале определена куча разных переменных, которые используются системой. Чтобы получить этот список, достаточно вызвать команду `env`. Попробуйте!

Этой командой мы выводим весь список **переменных окружения**, которые определены в **сессии терминала**. Boy, целых два новых термина ...

Да, начнём с переменных окружения - это переменные, которые немного отличаются от обычных переменных.

Давай представим, что у нас есть скрипт, который вызывает другой скрипт. Так вот, переменные из первого скрипта не будут доступны во втором, так как это обычные переменные.

А вот если её определить через `export VARIABLE=value` (прописав `export` перед переменной), то она станет переменной окружения, станет круче и будет видна всем дочерним скриптам (тем, который создаются из основного нашего скрипта).

А зачем нам сейчас это знать? Да всё просто, весь запуск системы - это запуск кучи скриптов одного в другом. А в системе есть пачка полезных переменных окружения, о которых нам надо знать:

- `HOME` - путь до домашней директории пользователя;
- `PATH` - пути до директорий, где искать утилиты для исполнения;
- `PWD` - путь до нынешней директории;
- `USER` - имя пользователя

💪 Попробуй каждую из переменных вывести с помощью `echo`.

Итого, переменные окружения определяются где-то в запуске и поэтому, будь они обычными, мы бы до них не добрались. А так, просто знайте, что переменные окружения очень полезны.

Сессия терминала

Такс, а что с сессиями? Что это такое?

Давай проведём простой эксперимент: открой два терминала, в одном определи переменную окружения `export MY_VAR=10` и проверь, что в этом терминале она есть в `env` и выводится через `echo`.

Отлично, а теперь сделай `env` и `echo` (без её определения!) во втором терминале на эту переменную и найди её.

Её нету? Как? Проверь ещё раз!

Окей, если так и не получилось найти, то ты теперь знаешь, что такие сессии - это отдельные пространства, в которых есть свои переменные.

То есть, если создать два терминала, то создаётся две сессии, которые между собой не делятся переменными и остальным. Скажу больше, вызов скрипта - это отдельная сессия, но если определить переменную окружения в терминале и в этом же терминале вызвать скрипт, то эта переменная окружения будет доступна скрипту. Но она не будет доступна, если вызвать тот же скрипт в другом терминале.

Запутал? Не переживай! Просто помни, если переменная не определена, то может неправильно сделана настройка установки переменных. Они наследуются по принципу, если раньше не было определено, то и не будет.

Импорт переменных из скрипта

Мы уже убедились, что команды в скрипте - это удобно. Раз, и все команды вызваны. Но что, если я хочу сделать также с переменными окружения? Написать скрипт, который в сессии определит все нужные переменные?

Есть в `bash` такая возможность, называется "env sourcing".

По сути, мы в скрипте пишем экспорты переменных, которые хотим определить в сессии и хитро вызываем. Давай напишем скрипт `env_pack.sh`:

```
#!/usr/bin/env bash

export VAR_ONE="one"
export SUPER_VAR="puper"
export BIBA="boba"
```

Сохраним, делаем исполняемым.

Теперь, попробуй вызвать скрипт и после проверить эти переменные через `echo`:

```
./env_pack.sh  
echo $VAR_ONE  
echo $SUPER_VAR  
echo $BIBA
```

Хм, все строки пустые? Правильно, потому что при вызове скрипта создалась внутренняя сессия, там создались переменные окружения, но наверх их никто не может поднять!

Тогда, давай сделаем sourcing через следующую команду:

```
source ./env_pack.sh
```

После этой команды проверь три наших переменные.

Они на месте! Но что же произошло? Тут все оч просто, по-умолчанию сессия создаётся на каждый вызов скрипта, но `source` команда заставляет вызвать скрипт в той же сессии, что мы сейчас находимся (сессия терминала).

Итого, вот так можно определять скрипты, которые хранят в себе настройки переменных и подкидывать пачки переменных прямо в сессии.

Команда `source ./env_pack.sh` аналогична команде `. ./env_pack.sh`. Да, точка - это команда `source =)`

Точка входа терминала

Теперь ещё больше упростим себе жизнь! Мы всё говорим об автоматизации всяких штук, что сложные структуры папок можно создать одним скриптом, что кучу настроек переменных можно сделать одной командой.

Это уже крутые возможности!

Но что, если есть какие-то команды и переменные, которые надо делать/настраивать при запуске терминала? Согласись, было бы удобно иметь такое место, куда можно прописать команды, которые будут делаться каждое создание терминала?

Есть такой вкуснятины у меня для тебя, называется "rc-файл"!

По сути, каждый раз, когда открывается терминал, он делает `source $HOME/.bashrc`. То есть мы можем отредактировать `.bashrc` файл, чтобы настроить его так, как нам нужно!

Давайте сделаем это!

💪 По-умолчанию, в `.bashrc` уже есть команды, поэтому правильным подходом будет написать в конце файла все команды. Пропиши определение переменных "VAR_ONE", "SUPER_VAR" и "BIBA" из предыдущего раздела в `bashrc` и создай новый терминал. Убедись, что в терминале переменные определены безо всяких действий сразу с запуска.

Хоть `.bashrc` и находится в домашней директории, но его не видно в файловом менеджере и в команде `ls`. Привильно, потому что в Linux файлы/папки, имя которых начинается с точки - скрытые. Для них у `ls` есть опция `-a`.

Обрати внимание, что мы подставили `$HOME` в путь при выполнении команды `source`. Любые переменные можно так использовать! Для `$HOME` в Linux есть сокращение `~`, то есть абсолютный путь до `.bashrc` можно сформировать как `~/bashrc`.

Вот так, теперь, когда ты умеешь настраивать запуск терминала под себя, тебе любая проблема по плечу!

Команды для обработки вывода

Последнее, о чём сегодня поговорим - обработка вывода команд.

Помнишь, команда `env` выводит огромный список из переменных окружения. Можно глазами искать нужные данные в списке, но мы же с тобой знаем, что всегда найдётся способ сделать это лучше и удобнее!

Для этого в Linux есть специальный подход под названием `pipe`. Это способ передачи результата одной команды в другую для его обработки.

Если проще, что если я хочу из всего вывода `env` найти только те строки, которые содержат строку "HOME"?

Тогда мы берем `env`, символ "|" (pipe) и новую для нас команду `grep`, которая ищет строки, содержащие интересующий шаблон!

Попробуем:

```
env | grep "HOME"
```

💪 Попробуй вывести те строки, которые содержат строку "PY".

Вау, круто! И я скажу больше, pipes можно складывать из любого количества команд!

💪 Выведи, сколько строк отображается командой `env`, воспользовавшись через pipe командой `wc`. У `wc` есть полезные опции для этого, посмотри справку.

Чему научились?

- Краткий экскурс в пару команд управления файлами
- Как писать скрипты и делать их исполняемыми
- Как работать с переменными окружения и сессиями
- Как импортировать переменные из скрипта в сессию
- Как настраивать запуск терминала под свои нужды
- Как обрабатывать вывод от одной команды другими

В этой теме прошли только основы-основ, так то и в скриптах есть логические конструкции, циклы и т.д., и команд управлени в разы больше, так что не останавливайтесь на этом, читайте статьи в интернете и пробуйте узнавать новое самостоятельно!

Ресурсы

- Хороший справочник по Bash
- Курс на Stepik: Введение в Linux
- Книги по Linux
- Еще больше книг
- Премудрости Bash
- Плагины для NeoVim
- Полезный софт на каждый день

Git - ОСНОВЫ

В этой теме мы рассмотрим основные моменты, как работать с репозиториями системы контроля версий (VCS) Git и как синхронизироваться с GitHub (веб-сервисом для хранения кода).

Содержание

- Содержание
- Начнем с простого - что такое `git` и репозиторий?
 - Что делать с новыми файлами?
 - Что приходится переживать файлу в репозитории
 - Мой первый коммит!
 - Что по истории?
 - Пора развивать историю
 - А что если я в stage добавил лишний файл или часть изменений файла?
 - Шагаем во времени
 - Задание раз
- Тогда что такое GitHub и зачем он нужен?
 - Так что, создаем удаленный репо?
 - Репозиторий нас связал
 - СИ-И-И-ИНХРОНИЗАЦИЯ!
 - Задание два
- Конфликты!
 - Задание три
- Вместо выводов
- Небольшие рекомендации

Начнем с простого - что такое `git` и репозиторий?

В основе работы с Git лежит понятие **репозитория**. Репозиторием может быть любая директория в файловой системе.

Для того, чтобы сделать из обычной директории репозиторий, достаточно вызвать в этой директории команду инициализации:

```
git init
```

Эта команда создаст в директории скрытую директорию `.git`, которая будет содержать в себе настройки этого репозитория.

⚠ Рекомендуется вручную не редактировать файлы в этой директории, если не понимаешь, зачем это делать. Команды утилиты `git` будут сами редактировать нужные файлы в ней.

С этого момента в репозитории (инициализированной директории) будут работать команды утилиты `git`!

💡 Попробуй вызвать команду `git status` в неинициализированной директории. Ожидаемый вывод?

Что делать с новыми файлами?

Отлично, репозиторий в результате получился, но что толку, если мы еще ничего не применили и не управляем версиями? Здесь начнем с небольшой теории!

Чтобы начать управлять версиями, нам нужно создавать файлы, так? Всё верно! Но простое создание файлов ничего не даст, так как репозиторий **не будет знать, что файл появился в репозитории**. Да-да, создать файл недостаточно, нужно сообщить репозиторию о том, что тут есть новый файл и что за ним надо следить!

❓ Тут можно подумать "Почему бы создателям git не сделать автоматическое добавление файлов в репозиторий"? Всё просто! Управляемость важнее, чем удобство автоматизации при работе с утилитами! Это важно в функционале базовых утилит, но у git есть куча надстроек, которые позволяют сделать автодобавление только что созданных файлов и другие плюшки, но **базовая утилита должна в первую очередь давать делать базовые операции**.

💡 Создай файл `shopping_list.md` (список покупок) в репозитории и запиши в него три вещи, которые хочется (или надо) купить в магазине

Вот вроде и файл есть, но как понять, что сейчас происходит в репозитории? Есть очень простая и полезная команда, которой рекомендуем пользоваться постоянно:

```
git status
```

И что мы видим в выводе?

```
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
    shopping_list.md
```

Отлично, уже с первых шагов git подсказывает, что есть файлы, которые вообще в репозитории никак не учитываются (Untracked) и даже команду, как это исправить! Освоим ее!

Для добавления файлов к учету git есть хорошая команда:

```
git add <path>
```

Вместо `<path>` указывается путь (или несколько путей) до файла(-ов) и она добавляется к учету в репозитории.

💡 Добавь список покупок к учету в репозитории

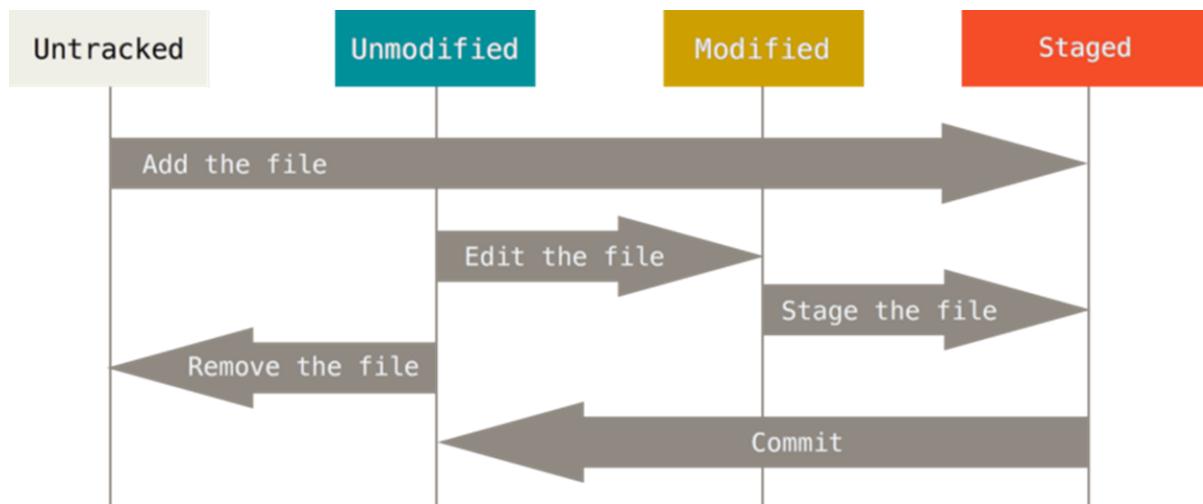
Если добавить получилось, то в статусе увидим новую картинку:

```
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
  new file: shopping_list.md
```

Хм, отлично, теперь он не **untracked**, а **to be committed**. Но что это все значит? Окей, вроде теперь система видит файл, но что значит **commit**?

Давай перейдем к теории на чуть-чуть.

Что приходится перекивать файлу в репозитории



Любой файл в репозитории имеет одно из четырех состояний:

- Untracked - создан, но не в учете репозитория
- Unmodified - учитывается репозиторием, но с последнего коммита не было изменений
- Modified - с последнего коммита что-то поменялось
- Staged - изменения с последнего коммита готовы коммититься

Тут мы приходит к понятию **коммита** в git:

Коммит (как действие) - сохранение внесённых изменений. Разработчик коммитит последние изменения. **Коммит** (как состояние) - состояние репозитория в определенный момент. По сути, делая коммит, создает точка фиксации изменений кода/файлов.

Основная идея работы с коммитами в фиксации изменений, которые затрагивают кусочек работы. Например, нужно написать код чтения данных с датчика - не факт, что изменения затронут один файл, поэтому с последнего коммита проводятся наработки и после изменения коммитятся в репозиторий, чтобы зафиксировать состояние. Но что говорить, давай сделаем свой первый коммит!

Мой первый коммит!

Когда в статусе виднеется блок **to be committed**, значит уже что-то добавлено в Stage и это готово в коммиту.

Но мы ведь люди разбирающиеся? Просто так коммитить что-то может быть небезопасно, вдруг в Stage попало что-то, чего не должно там быть, как узнать, что я сейчас буду коммитить?

Конечно для этого есть команда 

```
git diff --cached
```

И вот мы видим, что действительно добавленный список попал в Stage:

```
diff --git a/shopping_list.md b/shopping_list.md
new file mode 100644
index 0000000..fdc4696
--- /dev/null
+++ b/shopping_list.md
@@ -0,0 +1,3 @@
+* Молоко
+* Сахар
+* Сосиски
```

 **git diff** - это одна из утилит, которая позволяет посмотреть изменения в diff формате. Это такой формат, который показывает изменения в виде строк, которым предшествуют символы "+" или "-", означающие, что какая-то строка была добавлена, а какая-то удалена. Таким образом можно понимать, как изменились файлы и что планируется коммитить.

 Чтобы выйти из **git diff**, нажмите клавишу **q**

Отлично, убедились, что в Stage попало только то, что нужно! Поехали делать первый коммит! Команда простая:

```
git commit -m "My first super commit with super shopping list to remember what I
want to buy! Cool!"
```

Опция **-m** позволяет задать комментарий в коммите. Это очень важно, чтобы в истории коммитов ориентироваться, что и зачем было сделано.

 Если гит не хочет делать коммит и пишет просьбу указать "Ты кто такой?" (почта и имя), глянь в [FAQ раздел](#), он не любит работать с незнакомцами 

 Не ленись писать понятные комментарии! Будущий ты скажет спасибо, поверь 

 Сделай проверку своего Stage, проверь еще раз статус, если все ок - делай коммит! Только сообщение напиши своё!

Так-с, вот git подсказал, что создал новый файл и даже сколько строк добавил, круто:

```
[master (root-commit) 781c410] My first super commit with super shopping list to
remember what I want to buy! Cool
1 file changed, 3 insertions(+)
create mode 100644 shopping_list.md
```

Что по истории?

Немного резюмируем, пока процесс прост: репо (сокращение от репозиторий) создано, файл создан и добавлен к Stage, закоммичен. Еще бы понимать, а как посмотреть коммиты, а то утилита консольная, никакой графики.

Есть у меня для тебя команда, которая позволяет посмотреть историю коммитов. Название ее как нельзя логично 😊

```
git log
```

И вот видно, что в истории всего один коммит:

```
commit 781c410d0d8fd... (HEAD -> master)
Author: ***
Date:   ***

My first super commit with super shopping list to remember what I want to buy!
Cool
```

Отлично, теперь еще и посмотреть историю можно, вроде и команды простые и столько всего сделать можно!

Да, а еще у команды куча опций, что отображение истории можно прямо настраивать под себя, например:

```
git log --graph --abbrev-commit --decorate --format=format:'%C(bold
blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim
white)- %an%C(reset)%C(auto)%d%C(reset)' --all
```

Выглядит вот так:

```
* 781c410 - (6 minutes ago) My first super commit with super shopping list to
remember what I want to buy! Cool - *** (HEAD -> master)
```

Можно почитать `git log -h` или найти в интернете разные команды, которыми пользуются люди 😊.

Пора развивать историю

С добавлением нового файла с нуля вроде разобрались, но самое вкусное, когда начинаются модификации!

Тут важно помнить, git работает от коммита до коммита, так что по сути, если файл уже есть в системе, то у него есть три состояния:

- Unmodified - ничего не сделано с файлом
- Modified - есть изменения, но еще не в Stage
- Staged - изменения добавили в Stage, **вот они только и попадут в коммит**

❓ Тут ведь понятно, что Modified, но не Staged изменения не попадут в коммит?

⌚ Теперь добавь в список покупок еще два пункта, а один из предыдущих удали.

Вот и обновления списка подъехали, что нам git показывает?

```
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git restore <file>..." to discard changes in working directory)  
modified:   shopping_list.md
```

Хорошо, исправления он увидел, но давай посмотрим их! Тут `git diff --cached` не поможет, так как он показывает только Staged изменения, а у нас пока только Modified. Что делать? Воспользоваться просто `git diff`!

```
diff --git a/shopping_list.md b/shopping_list.md  
index fdc4696..0ca6f54 100644  
--- a/shopping_list.md  
+++ b/shopping_list.md  
@@ -1,3 +1,4 @@  
 * Молоко  
 * Сахар  
 -* Сосиски  
 +* Корнишоны  
 +* Вода
```

Вот и diff строки удаления подъехали 🎉. Если тут вопросов не возникает, то давай добавим в Stage изменения!

✓ Git уже и так подсказывает, что для добавления в Stage нужна команда `git add`, но мы на всякий обратим внимание, `add` команда используется как для добавления новых файлов (кстати тоже сразу в Stage), так и для добавления изменений в файле(-ах).

⌚ Ну вот и все, добавляй изменения в Stage, проверяй статус, убедись, что все идет, как ожидается. Делай коммит и проверь историю - у тебя теперь целых два коммита!

А что если я в stage добавил лишний файл или часть изменений файла?

Тут есть несколько вариантов:

- поправить файл и добавить в Stage обновленную версию (обрати внимание, добавление в Stage не равно коммит, можно постоянно менять Staged состояние до коммита);
- откатить все Staged изменения командой `git restore --staged <file>` (из подсказки самого git в статусе 😊).

Здесь хочется обратить внимание на неочевидный момент, после добавления в Stage, но до коммита, можно дальше редактировать файл. Может получиться такая картинка:

```
Changes to be committed:  
(use "git restore --staged <file>..." to unstage)  
modified: shopping_list.md  
  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git restore <file>..." to discard changes in working directory)  
modified: shopping_list.md
```

Один и тот же файл и в Stage и Modified. Как такое возможно? Очень просто, те изменения, что поймал `git add` попали в Stage, а все модификации, что были дальше - все еще Modified, так что их тоже надо добавлять.

⚠ Если данный вопрос вызывает сложности - рекомендуем отдельно запросить у преподавателя разъяснение, так как лично можно подобрать больше примеров и объяснений для улучшения понимания.

⚠ Таким образом, откатывать изменения из Stage несложно, главное не путать с откатыванием изменений Modified файлов, так как по сути это стирание изменений и откат файла до состояния последнего коммита. Аккуратнее!

Шагаем во времени

Отлично, на этом моменте ты уже знаешь, как накидывать новые коммиты, управлять файлом во всех четырех состояниях! Это уже очень круто!

Теперь узнаем про главную особенность VCS систем - переходы между коммитами. Ведь коммит - это зафиксированное состояние файлов и должна быть возможность перейти на сохраненное состояние, чтобы как минимум посмотреть, что было там!

Тут все просто, смотрим историю:

```
commit 39598a4181eb392c78e0cce9328c5a78732b1525 (HEAD -> master)  
Author: ***  
Date:   ***  
  
        List update, I forgot some items, sorry =(
```

```
commit 781c410d0d8fd8e6900babe53dcc8548f5c22  
Author: ***  
Date:   ***
```

```
My first super commit with super shopping list to remember what I want to buy!
Cool
```

или компактный вывод

```
* 39598a4 - (11 minutes ago) List update, I forgot some items, sorry =( - Alex
Home WSL (HEAD -> master)
* 781c410 - (30 minutes ago) My first super commit with super shopping list to
remember what I want to buy! Cool - Alex Home WSLb
```

Обрати внимание на эти два идентификатора в компактном виде и длинные строки в полной версии `git log`:

```
[39598a4]181eb392c78e0cce9328c5a78732b1525
[781c410]d0d8fdAA29e6900babe53dcc8548f5c22
```

Эти строки - хэш-суммы коммитов ([Git SHA, другой источник](#)), уникальные идентификаторы, которые однозначно уникальны и не повторяются среди коммитов. Укороченная версия просто показывает первые 7 символов, это как раз то минимальное количество хэша, который нужно указать, чтобы была возможность перейти на него.

Для перехода на состояние конкретного коммита есть команда:

```
git checkout 781c410
```

⚠️ У каждого репозитория свои хэш-суммы, посмотри свою историю и для команды бери оттуда!

В результате репозиторий перейдет в состояние файлов на момент коммита с хэшом `781c410`. Чтобы вернуть обратно достаточно набрать `git checkout master` или указать хэш конкретного коммита.

❓ `git checkout master` - это команда перехода не по хэшу коммита, а на последнее состояние ветки. В этой теме понятие ветки не рассматривается, но просто обращаем внимание, что у git по-умолчанию создается ветка `master`, а в некоторых репозиториях веткой по-умолчанию может быть `main`.

👉 Перейди на первый коммит в истории, проверь файлы, действительно ли они вернулись к состоянию на момент первого коммита?

Задание раз

Отлично, базовые команды работы с git изучены, настало время создавать репозиторий с кодом!

Создай свежий репозиторий, проведи работу с ним так, чтобы в нем был Python скрипт (например `main.py`), в котором вначале будет только вывод `Hello Git`. Закоммить изменения.

После этого добавь 2-3 строки с выводом в консоль и тоже сделай коммит, так создай историю на 4 коммита (или больше) за счет добавления и удаления строк вывода. Убедись, что при переходе по истории отображаются именно то состояние, которое коммитилось.

Ну и самое вкусное, разберись, как с помощью команд `git` просмотреть изменения между вторым и последним коммитом в истории (чтобы между ними было не менее 2x коммитов). В `diff` отображении ты должен увидеть ожидаемый вывод.

Тогда что такое GitHub и зачем он нужен?

The screenshot shows a GitHub repository page for 'github / codespaces-demo'. The 'Code' tab is selected. The commit history on the left shows several commits from 'mona' over the last two days, each with a commit message like 'Update README.md', 'commit message', etc. The code editor on the right contains a single file named 'README.md' with the content 'Hello world.'.

About

A package from Mona, for GitHub.

- Readme
- MIT license
- Code of conduct
- 1 stars
- 1 watching
- 1 forks

Packages 1

- octopackage/octopackage

Contributors 1

- mona Octocat

Достоверные источники сообщают, что

"GitHub — крупнейший [веб-сервис](#) для хостинга IT-проектов и их совместной разработки."

Так, ну ок, хостинг IT-проектов и совместная разработка, сложные слова...

Разберем следующий случай, вот как мы научились работать:

1. Создаем репозиторий на компе
2. Пишем код
3. Коммитим изменения

Теперь мы радуемся, что можем управлять и переходить на разные состояния кода

Отлично, а теперь чуть расширим процесс, чтобы получить еще больше профита!

1. Создаем репозиторий на компе **и на GitHub**
2. **Связываем репозиторий локальный (на компе) и удаленный (на GitHub)**
3. Пишем код

4. Коммитим изменения 🤝

5. Пушим в удаленный репозиторий 🚀

А теперь мы еще больше радуемся, потому что можем не только переходить на разные состояния кода, но еще и:

- Пошарить код с друзьями и коллегами 💬 💬
- Быть уверенным, что в случае проблем с диском код не потеряется 📁
- Иметь доступ к коду с любого компа и из любой точки с интернетом 🌐
- И много разных плюшек типа "трекера задач", "удобный просмотр коммитов" (а не вот этот ваш в терминале), "просмотр и редактирование кода в браузере" и многое другое! 🎉

Так вот по этой разнице видно, что сам по себе Git - это утилита для управления репозиторием, работы с ним локальной, а вот GitHub - это сервис, с которым можно синхронизироваться (скидывать обновления) и получать удобную работу в команде + бесплатный бэкап кода.

Шикарно! Но мы же пока не научились синхронизироваться с репозиторием?

Не беда, там все просто, главное, чтобы тебе было понятно, что с началом работы с GitHub (или аналогами типа BitBucket, GitLab и др.) появляется определение локального и удаленного репозитория.

Локальный репозиторий - репозиторий на локальном компьютере разработчика.

Удаленный репозиторий - репозиторий на веб-сервисе.

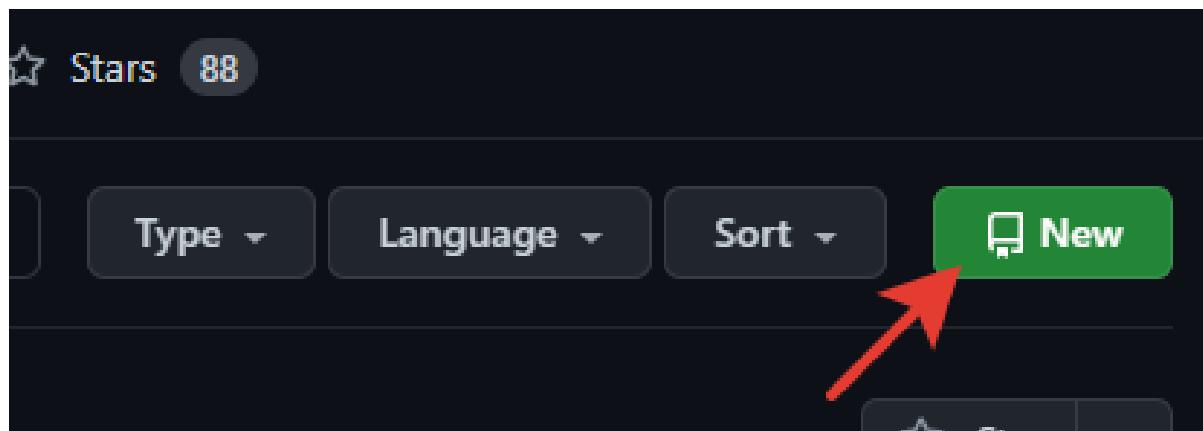
Чаще всего мы синхронизируем локальный с удаленным, чтобы доставить обновления на сервис, но в командной разработке, чтобы передать обновления от одного разработчика другому, надо синхронизировать и удаленный с локальным.

❓ Если в ходе работы с Git - GitHub появляются ошибки или проблемы - это нормально. На начальных этапах понимания всей этой системы совершаются ошибки и их надо спокойно разбирать. В интернете очень много разборов ситуаций и ответов на многие вопросы! Не пугайся!

Так что, создаем удаленный репо?

Ну поехали, если еще нет аккаунта на [GitHub](#), то вперед его создавать! Делается это очень просто через меню [Sign Up](#).

Создав аккаунт, поехали делать репозиторий! Либо в окне своих репозиториев выбираешь кнопочку [New](#), либо сразу идем по [ссылке на создание репо](#).



Там обязательно ввести **название репозитория**, остальное неважно (ну или потом разберись с настройками).

A screenshot of a newly created GitHub repository named 'KaiL4eK / my-new-super-repo'. The page shows a 'Quick setup' section with instructions for setting up the repository on desktop or via command line. It also includes sections for pushing an existing repository and importing code from another repository.

```

Quick setup — if you've done this kind of thing before
Set up in Desktop or HTTPS SSH https://github.com/KaiL4eK/my-new-super-repo.git
Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line
echo "# my-new-super-repo" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/KaiL4eK/my-new-super-repo.git
git push -u origin main

...or push an existing repository from the command line
git remote add origin https://github.com/KaiL4eK/my-new-super-repo.git
git branch -M main
git push -u origin main

...or import code from another repository
You can initialize this repository with code from a Subversion, Mercurial, or TFS project.
Import code

```

Отлично, после того как репозиторий создан, можно перейти к связыванию их и синхронизации!

Репозиторий нас связал

В окне нового репозитория уже есть подсказки, что и как делать! Но рассмотрим два случая!

[1] Случай один, первым создается репозиторий удаленный, а локального еще нет и код не создавался.

Очень простой и хороший кейс, так как по сути все, что надо сделать - это клонировать репозиторий, тем самым создав локальный из удаленного! Команда простейшая:

```
git clone https://github.com/KaiL4eK/my-new-super-repo.git
```

Такая команда создаст директорию с названием репозитория и вот они уже связаны. Прекрасно!

[2] Случай два, код уже есть в локальном репозитории и нужно синхронизировать его с удаленным репо.

Посложнее, но тоже реализуемо!

Суть в том, что нужно указать в локальном репозитории, что он связан с удаленным (remote). Не поверишь, как выглядит команда 

```
git remote add origin https://github.com/KaiL4eK/my-new-super-repo.git
```

После этой команды локальный понимает, что он связан с удаленным, но синхронизации еще не было. пока мы настроили связь!

СИ-И-И-ИНХРОНИЗАЦИЯ!

На самом деле термина “синхронизация” в git терминологии нет, просто термин очень красивый, вот и пользуюсь 

Есть два более простых: **push** и **pull**.

Да-да, это не то, чтобы мы нажали кнопку и пошел тяжелый процесс синхронизации файлов. Мы либо отправляем изменения на удаленный (push), либо стягиваем изменения с удаленного (pull).

Так вот в случае, когда мы клонировали репозиторий, они уже синхронизированы, но для случая ручной связи двух репозиториев надо явно вызвать команду пуша.

 В списке команд фигурирует команда `git branch -M main` – это команда переименования ветки в ветку `main`. Это важно сделать, если локальный репо создавался командой `git init`, так как git основной веткой создает `master`, а GitHub не так давно перешел на именование основной ветки `main`. Но загвоздка в том, что без переименования будут сложности с отправкой коммитов, так что не забывайте это сделать. Делается это один раз после создания репо.

Тут сверху есть предупреждение, короче, один раз поле создания репозитория переименуй основную ветку на `main`, либо через `git status` убедись, что она уже названа `main`:

```
On branch **main**
```

Команда переименования, если еще не переименовано:

```
git branch -M main
```

Так-с, коммиты есть, связь есть, ветка правильно названа, что делаем? Правильно, пушим коммиты на удаленку!

```
git push
```

⚠ В этот момент Git может спросить креды (username и пароль). О том, как сделать токен есть в разделе [FAQ](#).

Получаем

```
fatal: The current branch main has no upstream branch.  
To push the current branch and set the remote as upstream, use  
  
git push --set-upstream origin main
```

Сообщение говорит нам о том, что у нынешней ветки `main` нет `upstream` ветки. Хм, что же это значит?

Да все просто, локальная ветка создана локально, а удаленная - удаленно. "Спасибо, капитан", скажешь ты, но в этой фразе действительно есть логика 😊

Так как ветки созданы по отдельности, то первый такой `push` надо делать с указанием, в какую удаленную ветку пушить. Так сказать, связать ветки.

```
git push -u origin main
```

После использования опции `-u` дальше можно ее не использовать, так как связать их надо один раз!

⚠ Если созданный удаленный репозиторий уже содержит `README.md` или другие файлы, то могут быть ошибки, решение которым можно найти в [FAQ](#).

✓ Кстати, в случае клонирования репо ветки уже связаны. А вот если создаешь новую локальную ветку, то надо заново с помощью `-u` указывать, в какую ветку на удаленном репо лить 😊

Задание два

Ну что смотришь? Пора создать репозиторий на GitHub и залить свое творчество! Давай скорее, дальше очень вкусная и частая тема ждет нас!

Как зальешь на GitHub репо свои изменения, пройдись по интерфейсу, убедись, что видишь коммиты. Посмотри, что еще есть в репо.

Конфликты!

Вот работаешь один с репозиторием, никому не мешаешь, и тут твой друг/подруга предлагает замутить стартап или просто помочь с разработкой. Вроде хорошие новости, но давай рассмотрим ситуацию.

Ты пишешь в скрипте `main.py` код распознавания котиков на улице, а сосед по коду пишет в том же скрипте код распознавания собачек в доме.

Вроде пока все хорошо, но тут в чате прилетает "Код готов, уже на GitHub", а твой код еще в разработке.

Так в чем проблема, закончу и залью свою часть, но не тут то было 😬

Код готов и проверен - коммит - пуш - пу-пу-пу...

```
To github.com:KaiL4eK/my-new-super-repo.git
! [rejected]      main -> main (fetch first)
error: failed to push some refs to 'git@github.com:KaiL4eK/my-new-super-repo.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Что случилось, почему так??

Все просто, давай вспомним, что коммит - это состояние кода, а между коммитами хранятся изменения кода.

Получается так, что в твоей ветке из коммита, например,

```
1111111 - Script created
```

Создался следующий коммит

```
2221112 - Added code to detect cats in the street
1111111 - Script created
```

А у соседа по коду

```
2121212 - Added code to detect dogs in da house
1111111 - Script created
```

У вас два локальных репозитория и у каждого своя история коммитов, но на репо успела прилететь вторая, а что делать тебе?

Для начала понимаем, что для принятия каких-либо действия надо все стянуть на локальный репозиторий, потому что на удаленном пока ничего сделать нельзя, твою историю коммитов туда не доставить. Пулим с удаленного:

```
git pull
```

И получаем

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 695 bytes | 695.00 KiB/s, done.
From github.com:KaiL4eK/my-new-super-repo
  b769e62..17d490f main      -> origin/main
Auto-merging shopping_list.md
CONFLICT (content): Merge conflict in shopping_list.md
Automatic merge failed; fix conflicts and then commit the result.
```

Что? Что за конфликты? Мы же даже не ругались?

Правильно, конфликты бывают разные, но многие из них можно и нужно решить.

Если откроем общий файл, который редактировался (код детектирования не покажу, но вот даже на списке покупок может быть конфликт), то увидим интересную картину:

```
* Молоко
* Сахар
* Корнишоны
* Вода
<<<<< HEAD
* Сосиски
=====
* Сгуха
>>>>> 17d490f23fcaceb3ce5eb3980bad82216bf4ebe2
```

Идея в том, что git подсвечивает конфликтующие части символами <<<<<, ===== и >>>>>.

Можно увидеть, что в коде образуются блоки. Рядом с символами пишет, откуда взято.

💡 **HEAD** - обозначение локальной ветки, **17d490f** - коммит, в котором обозначенные изменения были сделаны.

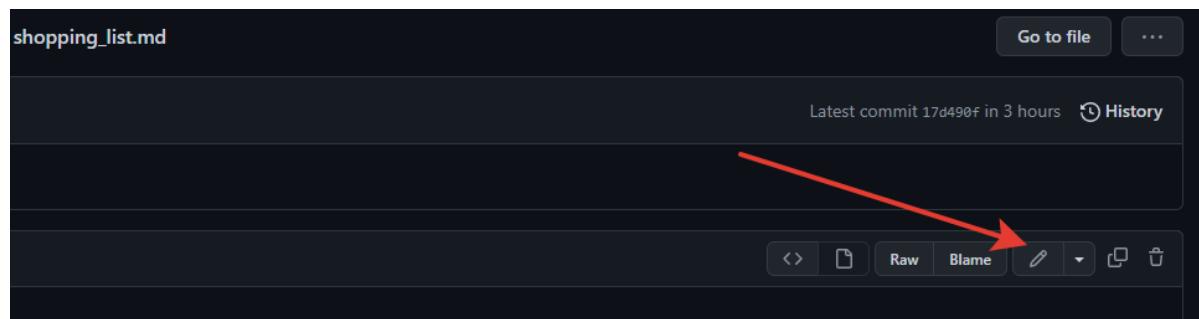
Так вот таким образом git предлагает отредактировать код, чтобы не осталось этих символов и оставить одну часть, другую или как-то хитро совместить. Основная идея в том, что git тебе говорит "Ну тут конфликт, делай что хочешь с этим, на этом мои полномочия все".

Так вот редактируешь файлик руками и после этого **делаешь еще коммит**, чтобы создать такой коммит в истории, который совмещает предыдущие два и исправляет пересечения редактирования.

Вот так просто мы решили конфликт разработки, хотя сами разработчики не ругались 😅

Задание три

Для этой темы отдельная задачка. Сначала создадим конфликт, а потом реши его 🤓



Отредактируй список покупок на GitHub (добавь пункт в список). Сделай коммит на GitHub.

После этого сделай изменения в локальном репозитории в списке (тоже добавь элемент, но другой, а то фокус не удастся 😊) и сделай коммит.

Теперь попробуй сделать push с локального в удаленный. Ура, есть конфликт! Реши его и посмотри, как сформировалась история коммитов.

Вместо выводов

Тут мы рассмотрели основы-основ, многое в git осталось за кадром (ветки, тэги, типы слияний и т.д.). Как правило, помнить все возможности не требуется, есть стандартный flow, который повторяется чаще всего, а уже какие-то специфичные вещи гуглятся и разбираются отдельно.

Вот еще страничка с ресурсами по Git, чтобы можно было получше прокачаться в нем: [Git](#) Так же можете пройти небольшой интерактивный туториал по Git, чтобы закрепить знания на практике [Gitgud](#)

Небольшие рекомендации

- Не инициализируй новые репозитории внутри уже инициализированного репозитория, если не знаешь, что такое [git submodules](#).
- Активно пользуйся [.gitignore](#) файлом, чтобы не перегружать репозиторий - репозиторий должен хранить только исходный код и исходные документы.
- Чаще используй команду [git status](#), чтобы быть уверенным, что планируешь закоммитить.
- Не бойтесь делать частые коммиты, лучше сделать больше частых фиксаций (особенно в конце рабочего дня) и синхронизировать с GitHub, чем долго не коммитить, а потом влить все разом и получить жирный коммит с кучей изменений. Слить несколько коммитов в один всегда можно, а вот разделять сложнее. Да и на утро может оказаться, что диск отвалился, а ты две недели писал код, но не коммитил 🤦.
- Не пользуйся force push, просто не используй, оно тебе не надо 😊

Hello, TurtleBot!

В этой теме мы улучшим запуск черепашки до трёхмерной модели! Представь себе, робот с кучей крутых датчиков в симуляторе на нашем компьютере! Bay!

Содержание

- [Содержание](#)
- [Подготовка](#)
- [Робот, покажись!](#)
- [Пора двигаться!](#)
- [Научим робота видеть!](#)
- [Робот, который смог \(двигаться к цели\)!](#)
- [Чему научились?](#)
- [Задачки](#)
- [Вопросики](#)

Подготовка

Так давайте начнём, первым делом установим все необходимые пакеты:

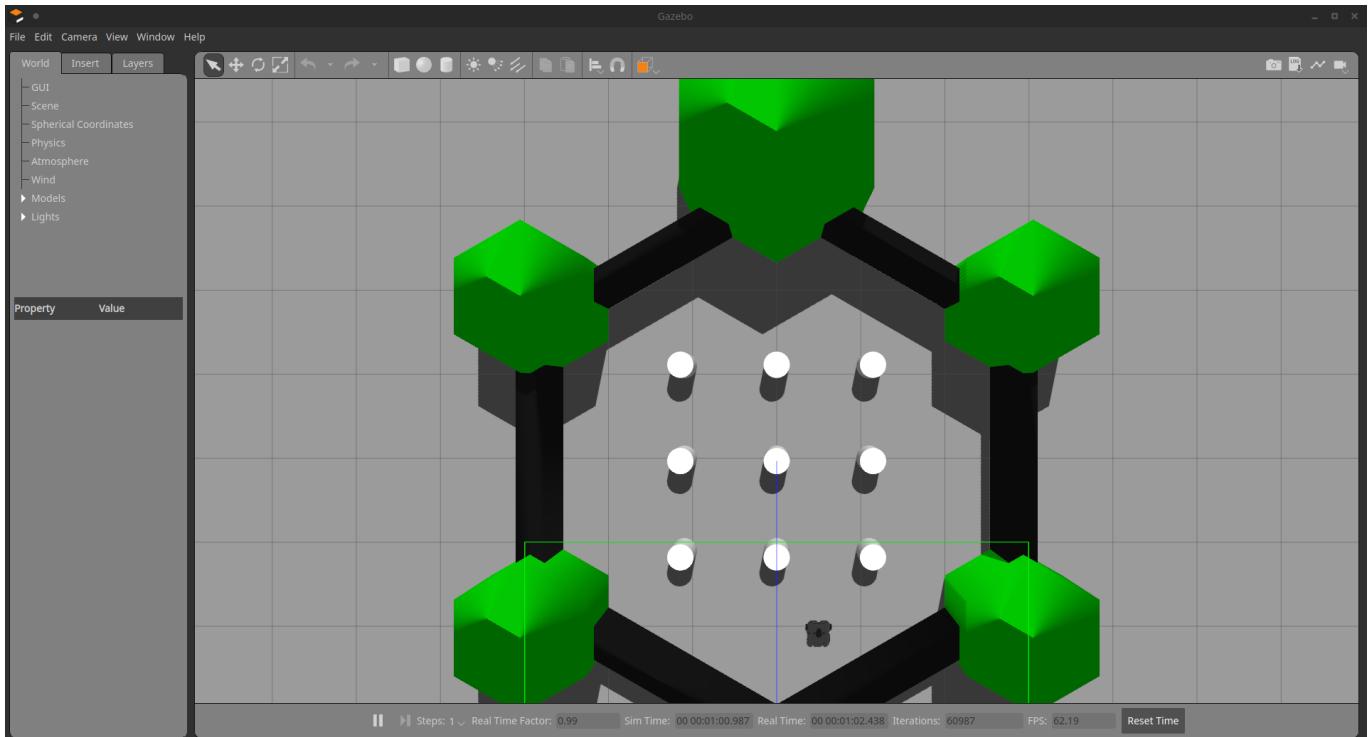
```
sudo apt install \
    ros-noetic-gmapping \
    ros-noetic-dwa-local-planner \
    ros-noetic-turtlebot3-gazebo \
    ros-noetic-turtlebot3-teleop \
    ros-noetic-turtlebot3-slam \
    ros-noetic-turtlebot3-navigation
```

Робот, покажись!

Теперь начнём знакомиться, давайте запустим нашего робота в симуляторе и увидим, что запустилось всё правильно!

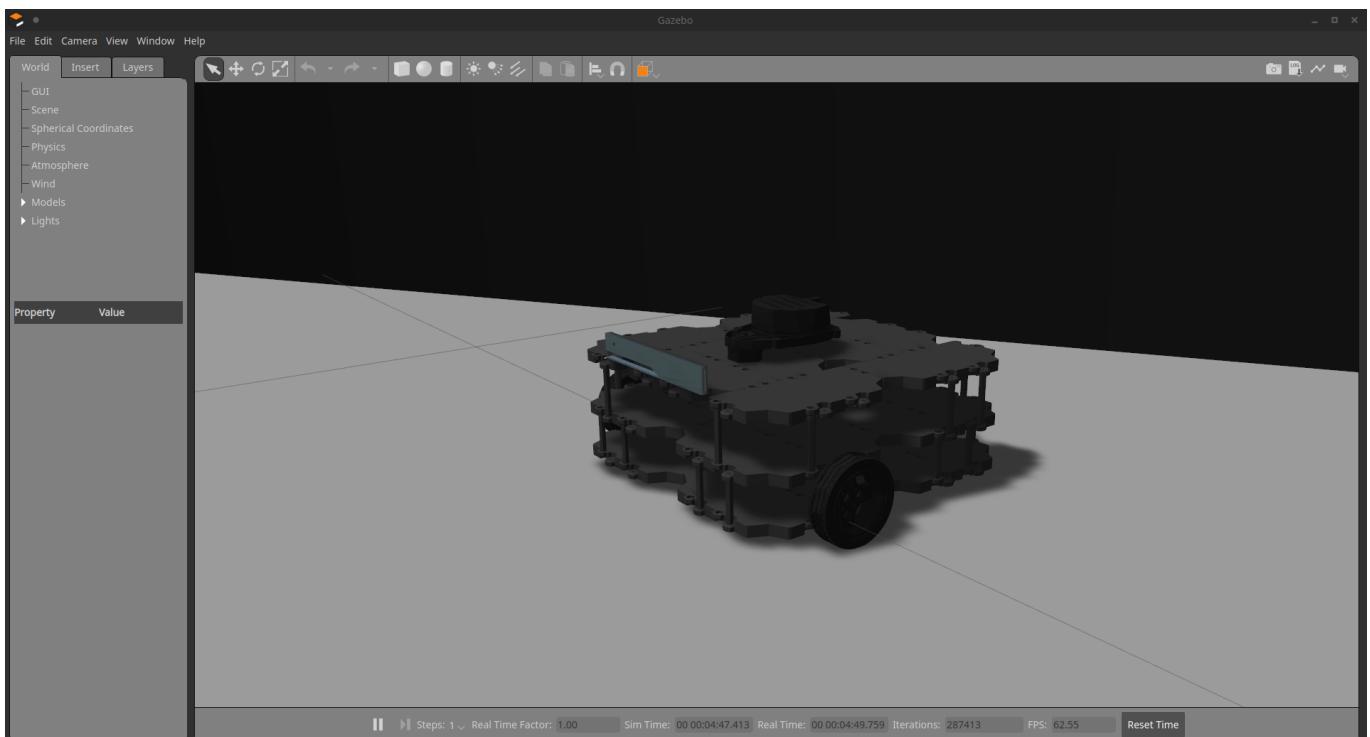
```
# TURTLEBOT3_MODEL=waffle - означает установку переменной окружения для запуска
данный команды (мы указываем желаемую модель)
TURTLEBOT3_MODEL=waffle roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

После выполнения команды должен запуститься симулятор **Gazebo**, в котором будет наш робот:



Вон там снизу справа видишь робота? Это наша вафелька! Если быть точнее, то Turtlebot модели Waffle =)

Попробуй приблизиться с помощью левой, правой клавиш мыши и колёсика (на колёсико ещё можно нажимать) и разглядеть поближе:



Какая хорошая детализация, не так ли? Красивый робот =)

Полюбуйся ещё минутку на робота, осмотри карту и двигаем дальше!

Пора двигаться!

Робот в симуляторе - это безопасная зверюшка, с которой можно делать что угодно, даже если и сломается, то перезапуск всё починет!

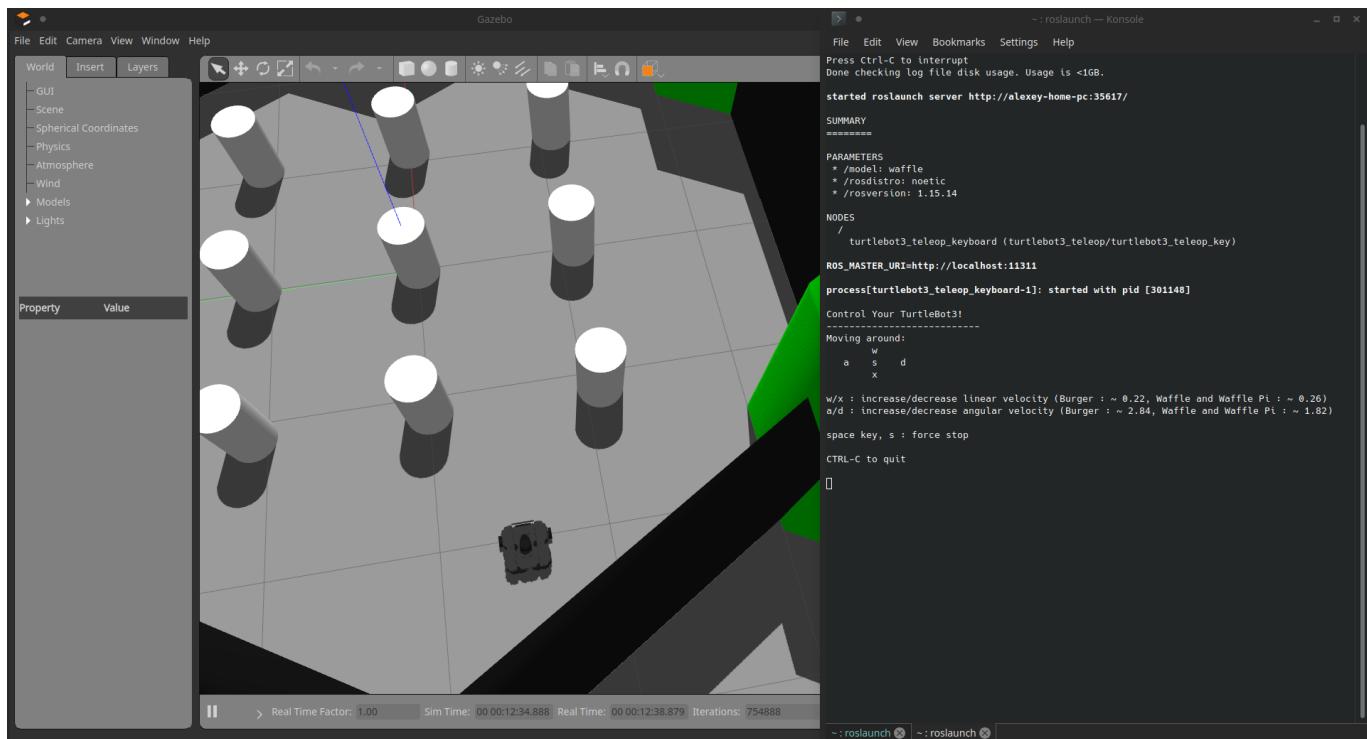
Чтобы заставить робота двигаться, надо передать роботу команды управления. Сейчас мы его просто запустили, даже немного осмотрели, но ничего не передаём.

Давай запустим команду, которая включит возможность управления роботом с клавиатуры, как мы делали с 2D черепашкой:

Симулятор не выключайте! Делаем это в отдельном окне!

```
# Обратите внимание, тут тоже просят модельку задать
TURTLEBOT3_MODEL=waffle roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

 Таакс, настроили окна, положили пальцы на WASD (ещё X для хода назад) и погнали!



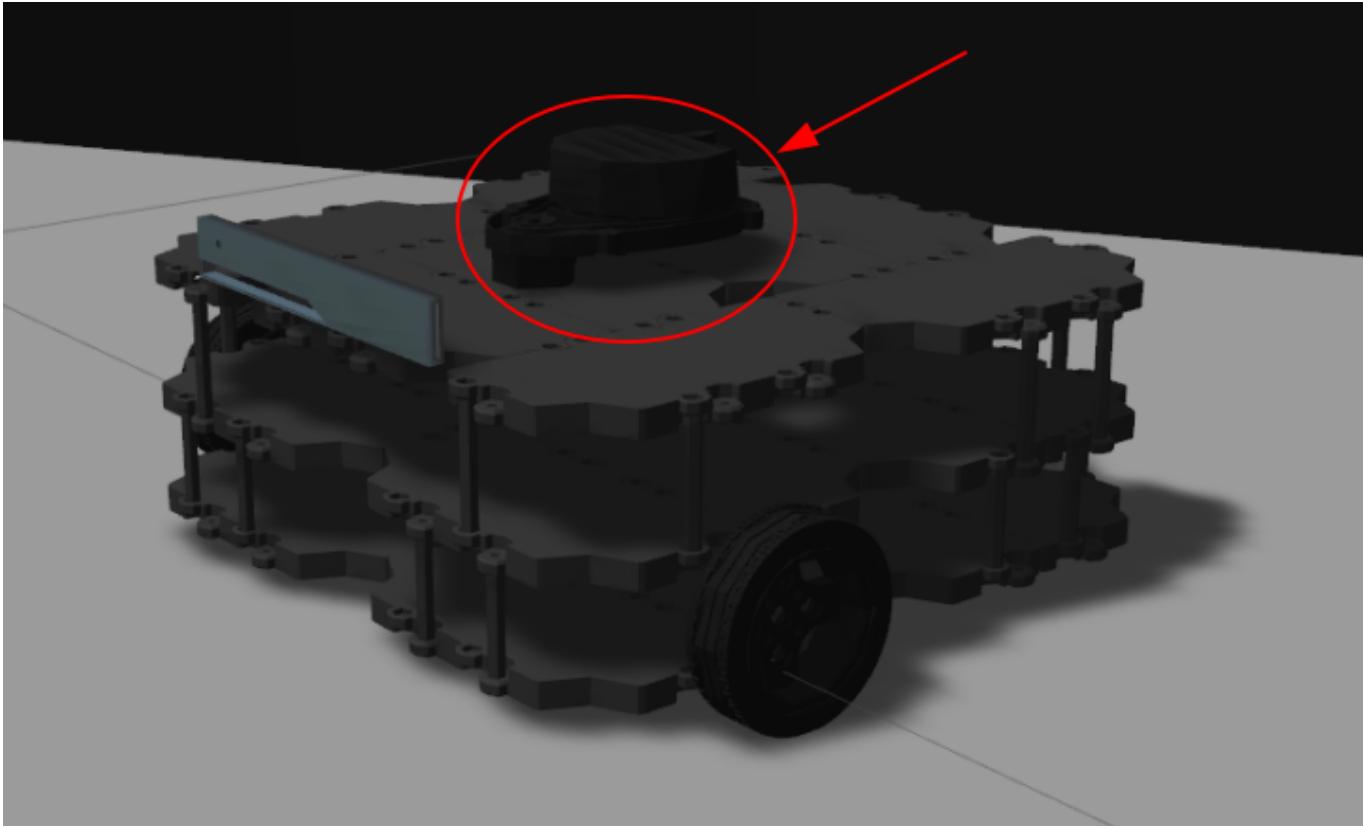
Отлично! Но подумай, чтобы управлять роботом, нам нужно смотреть сверху, где он находится. Ты в симуляторе смотришь с птичьего вида, а что если нам нужно работать с роботом, которому надо залазить в очень тесные и непролазные места? Нужно научить робота видеть самого!

Научим робота видеть!

Чтобы дать роботу глаза потребуется две составляющие:

- Датчик, который передаёт информацию об окружающем мире;
- Программа, которая обрабатывает данные с датчика и позволяет воспринимать информацию о дистанции до объектов.

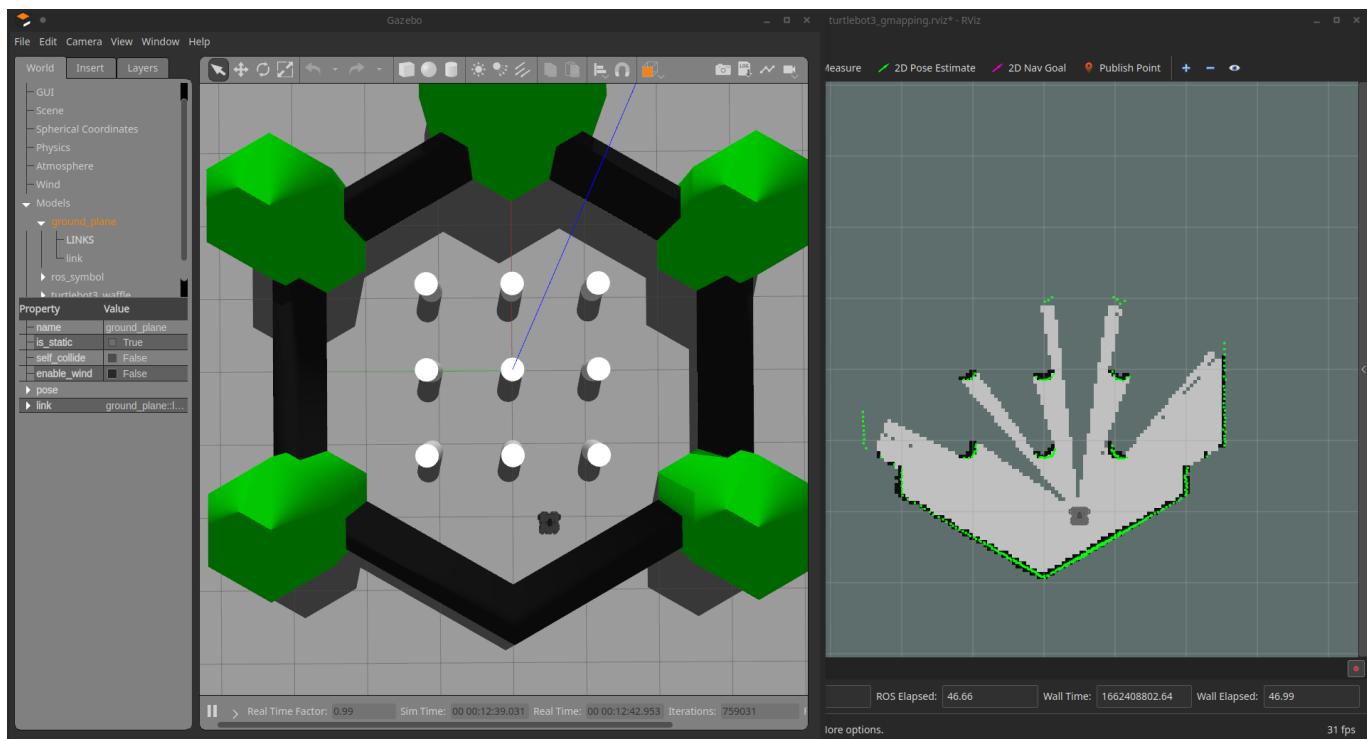
В качестве датчика у нас будет выступать **лидар**! Смотри, где он установлен:



Подробности о лидаре узнаешь на лекциях, а пока считаем, что это датчик, который сканирует плоскость и получает информацию о расстояниях до препятствий вокруг (360 градусов).

Отлично! Датчик у нас есть, а теперь давай в третьем терминале запустим программу, которая позволит роботу получать информацию с датчиков и обрабатывать её:

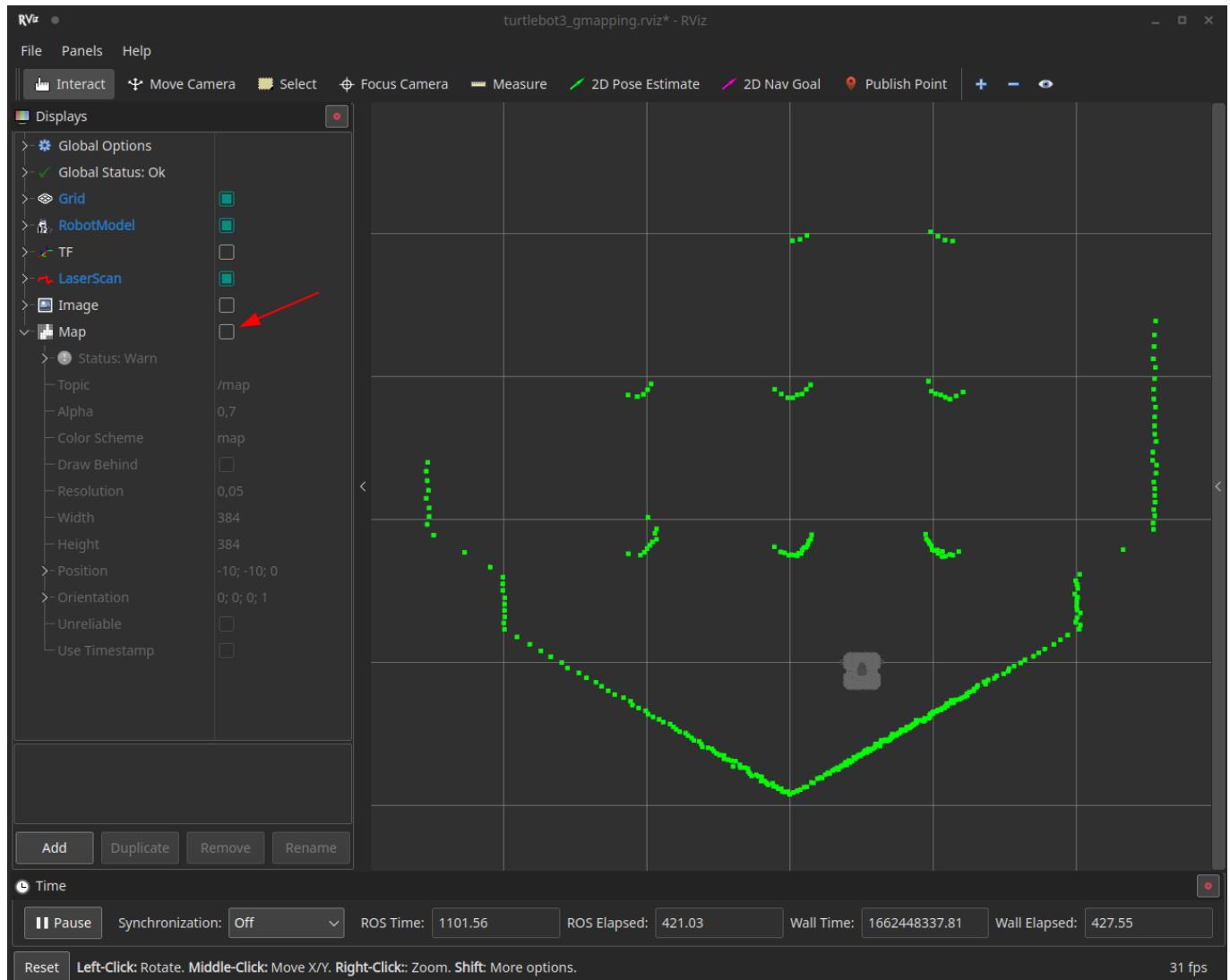
```
TURTLEBOT3_MODEL=waffle roslaunch turtlebot3_slam turtlebot3_slam.launch
```



Если всё запустилось правильно, то запустились программы для получения и обработки данных с лидара. Ты можешь их видеть в интерфейсе программы **rviz**. Rviz - это интерфейс отображения информации в системе робота.

Тут немного разберёмся, в Gazebo виден виртуальный мир симулятора, будто ты смотришь своими глазами, а Rviz - это интерфейс для отображения разной информации, которая есть в роботе! По сути, это взгляд глазами робота. Подумай, своими глазами сверху видишь всю карту, а робот видит только часть, пока никуда не двигался.

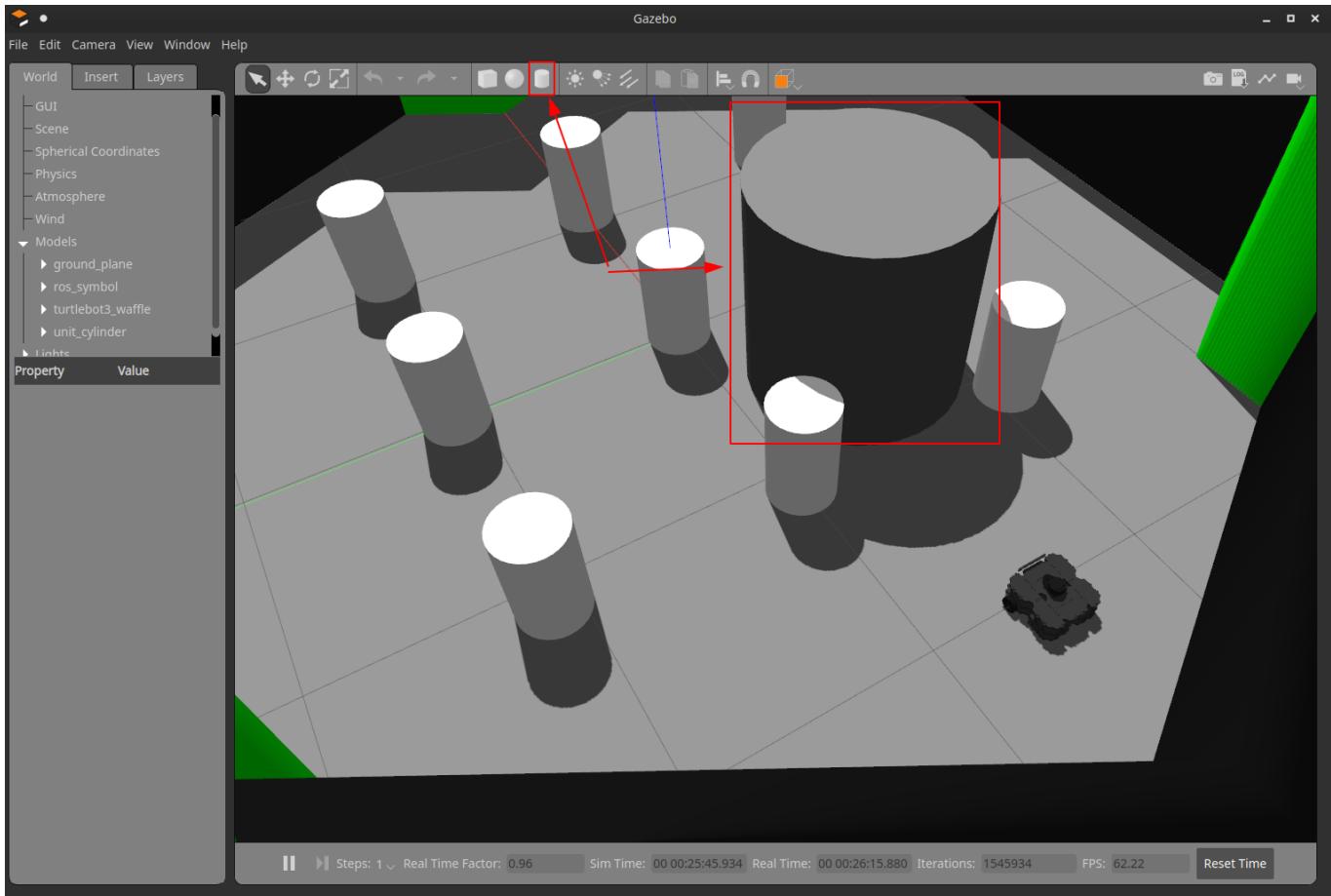
Давай посмотрим на информацию пока только с лидара? Отключаем отображение карты, чтобы были видны только робот и данные с лидара:



Отлично, теперь видно только то, что приходит с лидара без обработки!

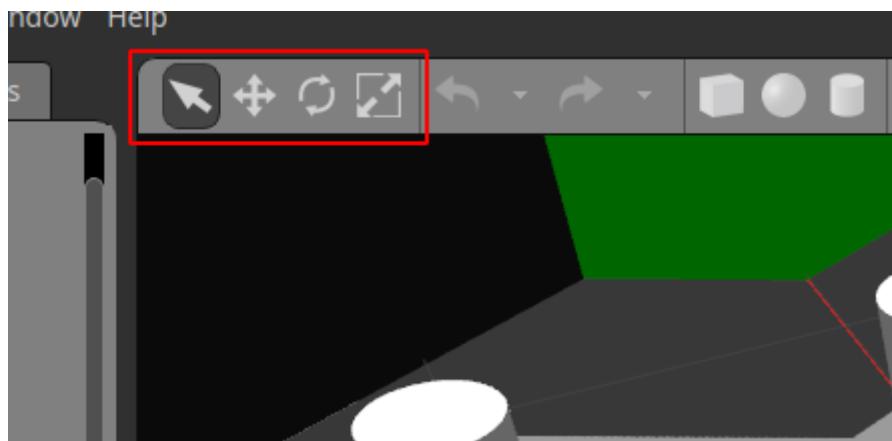
Давай попробуем немного поменять наш виртуальный мир и убедиться, что информация с лидара меняется!

Поставь цилиндр на свою карту с помощью кнопки цилиндра и проверь, как поменялись ваши данные с лидара:



Поменялись? Понимаешь, как они меняются, если ставите цилиндр в разных местах?

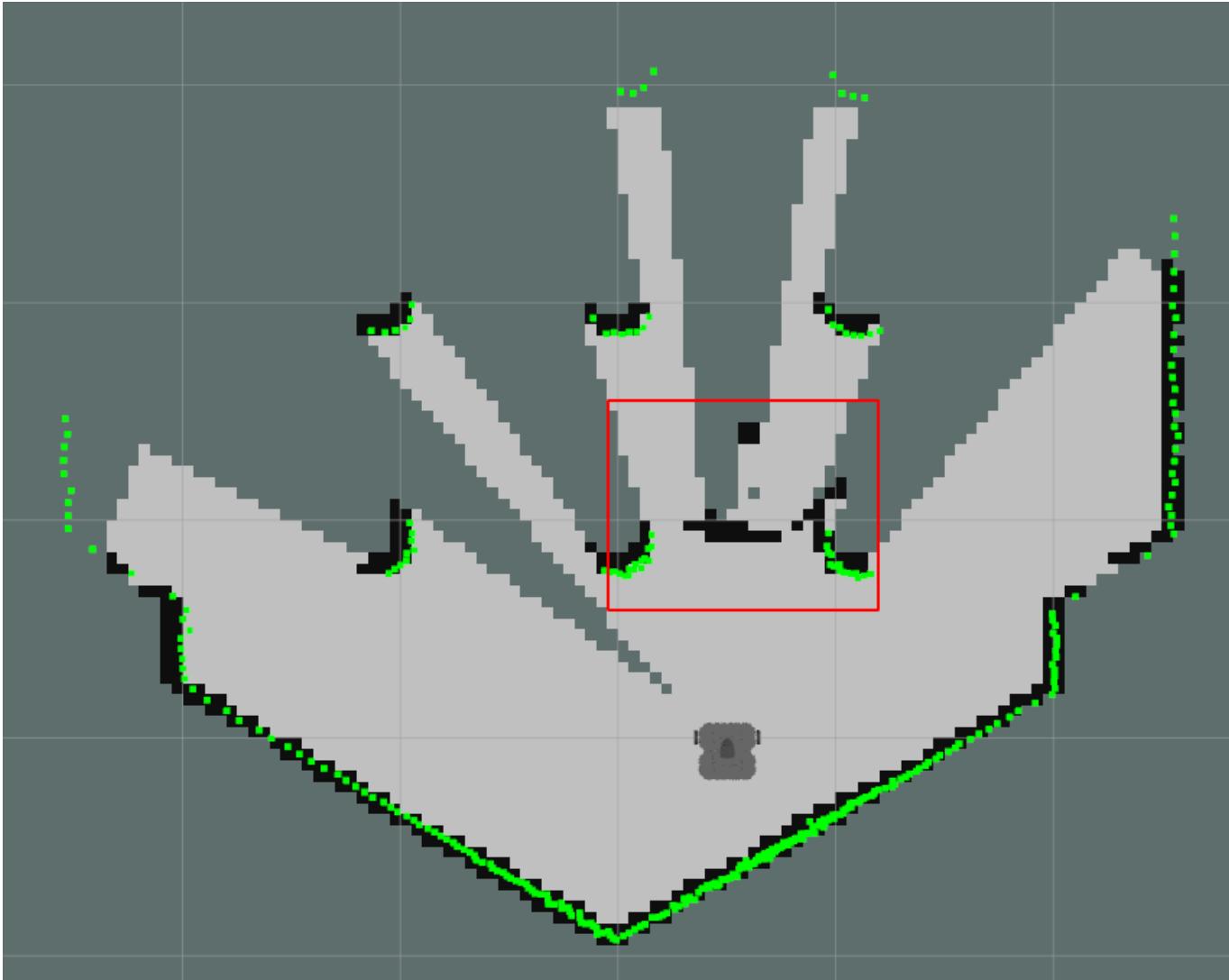
Попробуй попередвигать/поразворачивать/изменять размер цилиндра с помощью этого меню:



Убедись, что вы понимаете, как меняются данные с лидара. Если нет - двигай ещё =)

Небольшое задание, попробуй поменять размер (диаметр) цилиндра так, чтобы он пропал с лидара. Зависит ли диаметр пропадания от расстояния?

Отлично, после небольшого исследования возвращаем карту на место (включаем в rviz). Если мы активно двигали цилиндр, то на карте можно увидеть такие участки:



Это остатки рабочей системы построения карты в нашей среде. Ведь в Rviz мы только отключили отображение карты, но построение карты работало в фоне всё это время!

Но не беда, сейчас мы будем катать по карте и в ходе покатушек карта обновится и ложные препятствия пропадут.

💪 Упс, я уже немного раскрыл следующее задание.. Ну ничего, поехали, переключаем фокус на терминал с управлением с клавиатуры (если закрыли, не беда, просто включите управление teleop) и твоя задача - построить карту нашей небольшой местности. А если чувствуешь в себе дух гонщика, то попробуй сделать это на скорости!

Представляешь, ты всего лишь с помощью пары команд смог запустить робота в симуляторе и уже катаешь его с помощью своей собственной клавиатуры и строишь карту! Круто, правда?

Но согласись, сегодня дистанционным управлением по кнопкам никого не удивить, давай заставим робота двигаться к цели, учитывая информацию с карты?

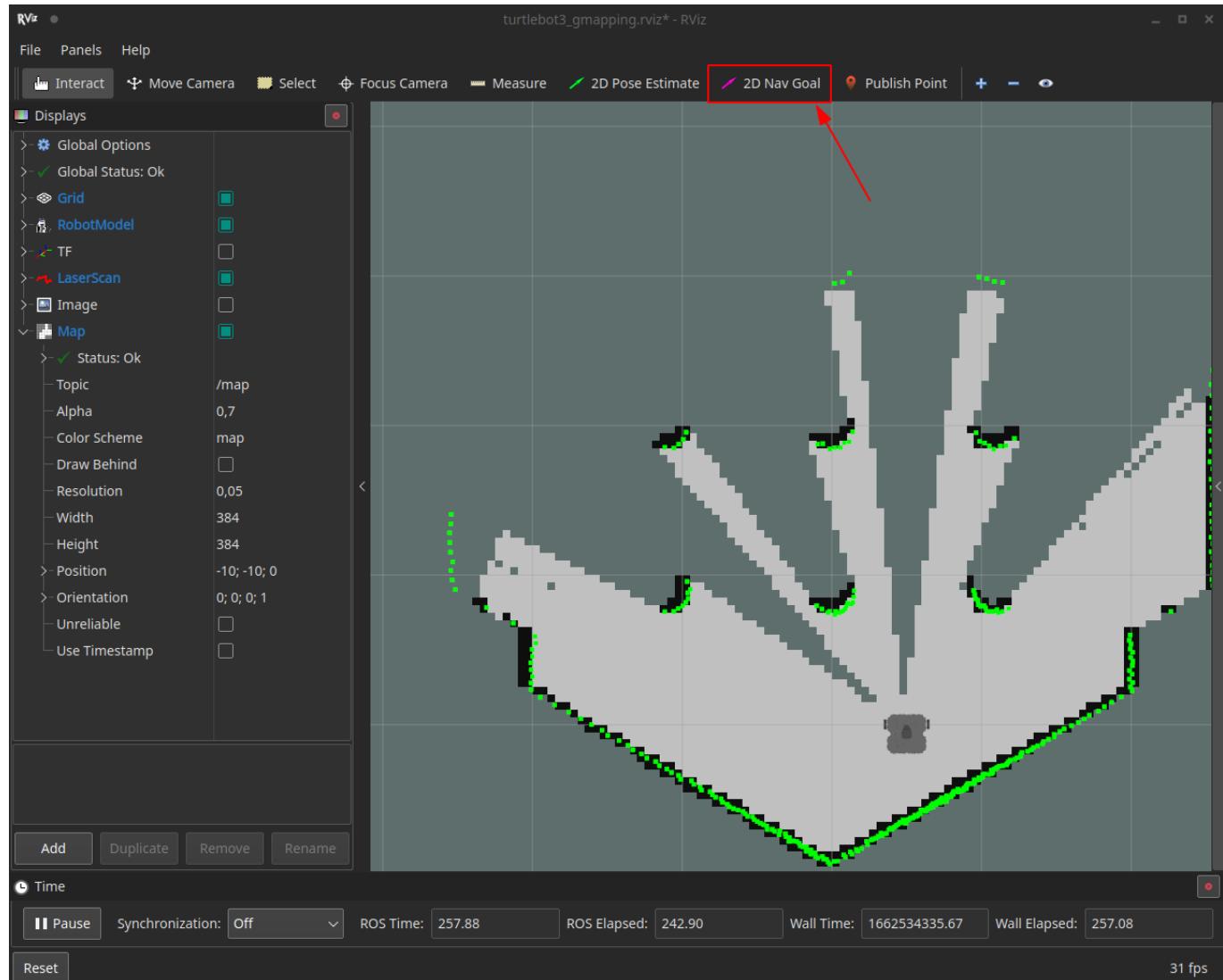
Робот, который смог (двигаться к цели)!

Давай выключим управление teleop и запустим программу, которая будет учитывать карту с данными с лидара и строить

```
TURTLEBOT3_MODEL=waffle roslaunch turtlebot3_navigation move_base.launch
```

Вроде запустили move_base вместо teleop, но ничего не поменялось. Правильно, ведь при управлении с клавиатуры тоже ничего не было видно - робот просто стоял и ждал команды, как ему двигаться.

Теперь, робот не ждёт команды с клавиатуры, а ждёт задания цели, куда ему двигаться. Давайте ему зададим эту цель с помощью кнопки **2D Nav Goal**:



Для задания включаете кнопку, левой кнопкой зажимаете в том месте, где хочется остановиться, а затем с зажатой кнопкой управляете направлением постановки.

Покатайтесь, посмотрите вид окна gazebo и rviz. Постарайтесь построить полную карту.

Но есть очень важное неудобство, мы не видим маршрут, по которому поедет робот, и цели, куда ему надо ехать. Давайте отобразим эту информацию в rviz!

Почему именно в rviz? Gazebo - это наш вид из глаз, поэтому мы на нём отобразить информацию не можем. А вот Rviz может отображать информацию, о которой знает робот, поэтому, это самое подходящее место для визуализации!

Здесь будет без картинки - начинаем привыкать искать подходящие кнопки

В меню Displays (слева в окне) нажимаем **Add**, затем выбираем вкладку **By topic** и ищем путь **/move_base/current_goal**. Выбираем под этим путём строку Pose (с красной стрелкой) и дважды кликаем по ней или снизу нажимаем **OK**.

Отлично, если сейчас задать цель движения, то будет видно, куда он должен двигаться! Круто и очень удобно.

Но ёщё, чтобы понимать, как робот поедет, давай отобразим путь робота!

Аналогично переходим в меню выбора отображения (**Add->By topic**), выбираем путь **move_base/NavfnROS/plan** и там выбираем Path (зеленая линия). Перед тем, как окончательно выбрать, внизу в Display Name наберите "Global Path".

Отлично, теперь мы видим путь до места, как "планирует" двигаться робот, но зеленый цвет - это очень ярко, поэтому давайте настроим отображение!

В меню Displays найдите Global Path (с зеленой линией), раскройте его, и давайте настраивать:

- Line Style ставим Billboards
- В меню Color выбираем любимый цвет
- В Line Width ставим желаемую толщину линии

Иии, вот так нехитро мы настроили то, как мы хотим видеть информацию внутри робота. Отлично!

Вот так мы научились запускать весь необходимый стек для управления роботом по заданию цели! Поздравляю!

Осталось дело за малым, в следующих темах подробно разобрать, что под этим лежит, чтобы в будущем ты смог сделать своего робота и запустить все необходимое!

Чему научились?

- Запускать симулятор с роботом из готовых launch-файлов
- Чуток научились работать с Gazebo
- Запускать и управлять (задавать роботу команды) с клавиатуры
- Запускать построение карты по лидару
- Запускать управление через задание цели на карте робота (которую построил робот)
- Работать с интерфейсом отображения информации Rviz

Молодец!

Задачки

Настала время первых экспериментов! Они очень полезны, так как пытаясь поменять что-то в системе и понять, почему так происходит, Вы приобретаете кучу опыта!

- Давайте кататься на карте **turtlebot3_house.launch** и **turtlebot3_autorace.launch**.
- В данных Image поменяйте ImageTopic на **/camera/rgb/image_raw** и покатайтесь. Как думаете, что это за информация?
- Попробуйте отключить RobotModel, включить TF и покататься по карте. Подсказка, TF - это разные системы координат. Как Вы думаете, почему часть TF стоит на месте, а часть двигается?

Какие TF стоят на месте? Отключить часть TF можно в меню TF->Frames.

- Попробуйте поменять модель робота на другую: [waffle_pi](#), [burger](#).

Вопросики

- Из каких частей состоит управление роботом по заданию цели? Что нужно роботу, чтобы реализовать такое?
- В чем отличие управление по заданию цели и управление с клавиатуры?
- Что нужно, чтобы обеспечить роботу возможность видеть? А если слышать или чувствовать запахи?
- Какие проблемы имеет лидар?
- Какая программа позволяет отобразить данные, которые есть внутри робота?
- Какая программа позволяет смотреть на симулятор глазами внешнего наблюдателя?

My First Package

Содержание

- [Содержание](#)
- [Привет](#)
- [Почему пакеты - это важно?](#)
- [Как мне создать локальный workspace?](#)
- [Готовим тарелку](#)
- [Чему научились?](#)
- [Задачки](#)
- [Вопросики](#)

Привет

До этого момента мы запускали уже готовые программы в разных терминалах, чтобы всё работало. Давай теперь начнём организовывать своё рабочее место, чтобы творить свои интересные штуки!

Например, мы достаточно накатались с черепашкой и хотим сделать своего робота в симуляторе. Робота, который приносит холодные напитки из холодильника! Это же надо не только уметь двигаться к цели, но и управлять манипуляторами (руками робота), а на черепашке такого нету =(Остаётся только творить самим!

Или нам подходит черепашка, но хочется для построения карты использовать другой датчик/другую программу для картографирования.

Или, у нас уже есть робот в реальном мире, но с ним опасно/долго постоянно работать (заряжать приходится, возвращать на начальную точку и т.д.) и мы хотим сделать в симуляторе, чтобы настраивать программы и тестировать работу программ на роботе!

Или...

В практике можно встретить очень много ситуаций, когда нам нужно сделать что-то своё, используя готовые решения (модифицировать их) или совсем с нуля создать. Для этого нам надо познакомиться с понятиями **пакета и рабочего пространства (workspace)** в ROS.

Почему пакеты - это важно?

Представь себе огромный шведский стол! На нём есть разные блюда разной кухни - роллы, пицца, хинкали, борщ и т.д.

Обычно, мы привыкли приходить к столу с подносом, накладывать на тарелки блюда (не смешивая их!) и потом с тарелками на подносе идти за стол. Отлично, на минутку представим вкусный стол, как мы отведаем еду и, если голод напомнил о себе - сходи покушай =)



Мы можем продолжать? =)

Так вот давай теперь представим легкую альтернативу:

- Блюдо - это полезные материалы, модели в симуляторе, программы, чего там только нету.
- Тарелка - это **пакет**, мы на него накладываем конкретное блюдо, но не смешиваем с другими, потому что будет невкусно!
- Твой стол - это рабочее пространство, на котором размещаются тарелки и ведётся работа (кушаем-с).

По сути, главная суть аналогии в том, что по пакетам мы раскладываем материалы (коды программ, конфигурации, описания, модели в симуляторе) по разным темам, чтобы не скидывать всё в кучу. Если сложить в кучу, то потом замучаемся разбираться, что и где находится!

То есть, если мы хотим сделать своего робота, то мы может создать отдельный пакет для робота, при этом материалы по другим роботам будут лежать в отдельных пакетах!

Если в нашем роботе есть: телеуправление, навигация, построение карты, то можно сложить это в отдельные пакеты (как, например, у черепашки turtlebot3_teleop, turtlebot3_navigation, turtlebot3_slam).

Рабочее пространство в свою очередь просто объединяет пакеты в себе в системе (на диске).

Первое время мы будем работать только с двумя рабочими пространствами. С первым уже знакомы - системное, в него устанавливаются все пакеты, которые мы ставим через `sudo apt install ros-noetic-....`. Второе, с которым будем работать - локальное. Там хранятся пакеты, которые мы пишем сами или собираем из исходников.

Надеюсь, важность организации кода и материалов по пакетам понятна? Есть и альтернативный путь - можете создать для всех своих работ единственный пакет и все складывать в него, но потом на своем опыте поймёте, почему такой подход сложен в работе по мере роста проекта =)

Как мне создать локальный workspace?

Чуть подготовимся, поставим `catkin` инструмент сборки командой `sudo apt install python3-catkin-tools`

Так, ну если в системном пространстве мы уже запускали пакеты, то где находится локальное?

А его ещё нету =) Его надо создать!

Сделаем это, по общей практике это делается в домашней директории пользователя:

```
mkdir $HOME/catkin_ws
```

Если вы не понимаете, что делает команда `mkdir` или что значит `$HOME` - прочтайте [Shell топик](#) в нулевом разделе.

Так мы создадим директорию `catkin_ws` - это и будет нашим локальным (ещё можно назвать пользовательским, так как у каждого пользователя в системе будет своё) пространством.

Но пока это не пространство, а просто папка, давайте сделаем всё, чтобы её инициализировать!

```
# Go внутрь
cd ~/catkin_ws
# Стартуем инициализацию
catkin init
```

И должны увидеть что-то наподобии:

```
Initializing catkin workspace in `/home/user/catkin_ws`.

-----
Profile:           default
Extending:        [env] /opt/ros/noetic
Workspace:        /home/user/catkin_ws

-----
Build Space:      [missing] /home/user/catkin_ws/build
Devel Space:      [missing] /home/user/catkin_ws/devel
Install Space:    [unused] /home/user/catkin_ws/install
Log Space:        [missing] /home/user/catkin_ws/logs
Source Space:     [missing] /home/user/catkin_ws/src
DESTDIR:          [unused] None

-----
Devel Space Layout:   linked
Install Space Layout: None

-----
Additional CMake Args: None
Additional Make Args:  None
Additional catkin Make Args: None
Internal Make Job Server: True
```

```
Cache Job Environments:      False
-----
Buildlisted Packages:        None
Skiplisted Packages:        None
-----
Workspace configuration appears valid.
-----
-----
WARNING: Source space `/home/user/catkin_ws/src` does
not yet exist.
```

Шикарно! У нас есть рабочее пространство!

Ай-да теперь создавать наш первый пакет в этом пространстве!

Готовим тарелку

Отлично, пространство мы создали, теперь сделаем свой пакет.

Первое, что нужно сделать - это создать директорию `src` внутри пространства:

```
# Обязательно внутри catkin_ws
mkdir src
# И заходим внутрь
cd src
```

Теперь создаём пакет с названием `super_robot_package`:

Если хочешь, можешь создать пакет со своим названием, но дальше тогда придется во всех командах заменять имя.

```
catkin create pkg super_robot_package
```

В выводе должен получиться подобный вывод:

```
Creating package "super_robot_package" in "/home/user/catkin_ws/src"...
Created file super_robot_package/package.xml
Created file super_robot_package/CMakeLists.txt
Successfully created package files in
/home/user/catkin_ws/src/super_robot_package.
```

После создания пакета очень важно его собрать! Для этого используем команду для сборки:

```
catkin build super_robot_package
```

В выводе увидите что-то такое:

```
NOTE: Forcing CMake to run for each package.  
-----  
[build] Found 1 packages in 0.0 seconds.  
[build] Updating package.  
Starting >>> catkin_tools_prebuild  
Finished <<< catkin_tools_prebuild [ 2.1 seconds ]  
Starting >>> super_robot_package  
Finished <<< super_robot_package [ 1.9 seconds ]  
[build] Summary: All 2 packages succeeded!
```

Отлично! Как нам теперь проверить, что система ROS видит наш пакет? Познакомимся с утилитой, которая работает с пакетами системы ROS - **rospack**!

```
# Посмотрим список пакетов в системе  
rospack list
```

В выводе видим:

```
...  
turtlebot3_gazebo /opt/ros/noetic/share/turtlebot3_gazebo  
turtlebot3_msgs /opt/ros/noetic/share/turtlebot3_msgs  
turtlebot3_navigation /opt/ros/noetic/share/turtlebot3_navigation  
turtlebot3_slam /opt/ros/noetic/share/turtlebot3_slam  
turtlebot3_teleop /opt/ros/noetic/share/turtlebot3_teleop  
turtlesim /opt/ros/noetic/share/turtlesim  
urdf /opt/ros/noetic/share/urdf  
urdf_parser_plugin /opt/ros/noetic/share/urdf_parser_plugin  
urdf_sim_tutorial /opt/ros/noetic/share/urdf_sim_tutorial  
urdf_tutorial /opt/ros/noetic/share/urdf_tutorial  
visualization_marker_tutorials  
/opt/ros/noetic/share/visualization_marker_tutorials  
visualization_msgs /opt/ros/noetic/share/visualization_msgs  
voxel_grid /opt/ros/noetic/share/voxel_grid  
webkit_dependency /opt/ros/noetic/share/webkit_dependency  
xacro /opt/ros/noetic/share/xacro  
xmlrpcpp /opt/ros/noetic/share/xmlrpcpp
```

Но в этом выводе тяжело искать, давайте отфильтруем этот список, поискав наш пакет:

```
rospack list | grep super_robot_package
```

Хм.. Ничего? То есть, наш пакет не видно? Давайте посмотрим, где ROS ищет пакеты с помощью переменной окружения `ROS_PACKAGE_PATH`:

```
echo $ROS_PACKAGE_PATH  
# /opt/ros/noetic/share
```

Ага, такс, а как заставить ROS искать пакеты в нашем WS (workspace)? Нужно сделать похожее с тем, что мы делали, когда устанавливали ROS.

Для системного пространства мы использовали команду `source /opt/ros/noetic/setup.bash`

Для локального WS мы используем команду:

```
source $HOME/catkin_ws/devel/setup.bash
```

После её выполнения проверь наличие пакета!

```
rospack list | grep super_robot_package  
# super_robot_package /home/user/catkin_ws/src/super_robot_package
```

Оп, вот и пакет нашёлся, значит он теперь учитывается в системе ROS.

 Убедись с помощью проверки переменной `ROS_PACKAGE_PATH`, где ищутся теперь пакеты.

Но ведь команда `source` работает только для нынешней сессии shell, чтобы она вызывалась в каждом терминале, как обычно, прописывай в `~/.bashrc` руками или командой `echo "source \$HOME/catkin_ws/devel/setup.bash" >> ~/.bashrc`.

Чему научились?

- Теперь мы можем создавать локальное рабочее пространство (WS)
- Подключать WS в систему ROS, чтобы учитывались пакеты внутри
- А ещё создать в нем пакеты
- Собирать пакеты
- И с помощью `rospack` выводить список пакетов и искать по ним!

Молодцы, круто!

Задачки

- Как с помощью утилиты `rospack` определять путь до конкретного пакета, не используя `grep`?
Подсказка, все команды `rospack` можно посмотреть с помощью `rospack -h`.
- Определите расположение пакетов `turtlebot3_gazebo` и других, которые мы ранее использовали.

- Мы знаем, что WS находится по пути `~/catkin_ws/src` и можем перейти в пакеты внутри, но что если пакеты где-то в другом месте? Воспользуйтесь утилитой `roscd` и разберитесь, как перейти в директорию пакета с помощью этой команды.

Вопросики

- Зачем нужно рабочие пространства и пакеты в ROS?
- Как нужно организовывать исходные коды программ и материалы по пакетам?
- Какими командами создаётся рабочее пространство? Что нужно сделать после его создания?
- Какие этапы создания пакета в рабочем пространстве?
- В какой переменной находятся пути, по которым ROS ищет пакеты?

Ещё немного поговорим о пакетах

Содержание

- [Содержание](#)
- [Структура WS](#)
- [Что делать с Git?](#)
- [Чему научились?](#)
- [Задание](#)

В этой теме мы обсудим остальные вопросы касательно пакетов!

Структура WS

В результате создания WS и сборки пакета появляются следующие директории:

```
build    # Файлы сборки, все, что генерируется в ходе сборки пакетов
devel    # Файлы результатов сборки
logs     # Логи сборки пакетов
src      # Исходные коды пакетов (создается пользователем)
```

Как видно, мы создаём директорию `src`, а после этого создаются доп.директории, которые зависят от платформы, на которой собираются пакеты.

Вот так немного обсудили, что это за директории.

Что делать с Git?

Теперь речь о Git, всем известно, что в системе контроля версий (VCS) хранятся только исходники, так как исходники не зависят от платформы и из них можно собрать конечные программы.

Тогда, получается, что в Git мы не можем хранить всё рабочее пространство, так как в рабочем пространстве находятся результаты и файлы процесса сборки.

Но что тогда нам хранить в Git? Правильно, только сами пакеты. В таком случае мы получаем возможность управлять каждым пакетом и собирать рабочее пространство из разных пакетов.

Итого, мы формулируем правило:

В Git храним только исходные коды и файлы самих пакетов, файлы результатов сборки не кладем в Git!

Чему научились?

- Важно, что в VCS хранить надо только исходники. В ROS это удобно, так как исходники хранятся в пакете, а сборка делается в других папках.
- Таким образом, в VSC (например, Git) храним сами пакеты и наборы пакетов!

Задание

Сделай репозиторий на GitHub и загрузи созданный пакет в репозиторий на GH.

Если не знаете, что делать с Git или вообще, что это такое - го читать [тотик из раздела 0 про Git!](#)

Launch everything!

Содержание

- [Содержание](#)
- [Привет!](#)
- [Подготовка](#)
- [Мой первый launch](#)
- [Залезем чуть глубже](#)
- [Аргументы launch](#)
- [Задание](#)
- [Чему научились?](#)

Привет!

Привет! Сегодня мы с тобой узнаем, что такое launch-файлы и утилита `roslaunch`.

Нам довелось пользоваться этой программой, чтобы запускать разные части нашей системы на черепашке. Как мы уже обсуждали ранее, много готового существует в Open Source мире и в ROS пакетах, но что если нам хочется сделать что-то своё? Или как-нибудь организовать то, что уже создано?

Например, нам в моменте запуска системы с черепашкой и управлением надо было запускать три терминала! А с более сложными системами может потребоваться стартовать и под десяток, если посчитать все необходимые части. Давай разберёмся, чем нам поможет `roslaunch`?

Подготовка

Уже должно быть создано и подключено локальное рабочее пространство, а также создан собственный пакет как репозиторий на GitHub.

Мой первый launch

Думаю, я порадую вас, если скажу, что вместо запуска двух терминалов (с симулятором и построением карты) можно записать всё в один запуск!

Для начала нам нужно создать папку `launch` и в ней сделать файл `turtlebot3_for_fun.launch` с начинкой:

```
<launch>
    <include file="$(find turtlebot3_gazebo)/launch/turtlebot3_world.launch" />
    <include file="$(find turtlebot3_slam)/launch/turtlebot3_slam.launch" />
</launch>
```

Что-то новенькое! Если ты ни разу не встречался с языком разметки XML, то это действительно всё в новинку!

Тут ничего сложного, конструкции в XML построены по принципу открыть-закрыть. В этом тексте `<launch>` - это открывающий тэг, `</launch>` - закрывающий. У тэгов могут быть атрибуты, например `file` у тэга `<include>`. Подробности можно найти в описании [формата launch и roslaunch](#) и конкретно об `include`.

Ну, что говорить, сохрани и давай запустим!

```
TURTLEBOT3_MODEL=waffle rosrun super_robot_package turtlebot3_for_fun.launch
```

Без пересборки ROS сразу видит, что у нас в пакете есть launch файл.

Работает! Появился интерфейс симулятора и Rviz с картой! Убедил? Это работает =)

Итак, теперь подробнее про сам launch формат, он всегда имеет единственный тэг `<launch>`, который обрамляет весь файл!

По сути, launch-файл - это перечисление тех программ, которые надо вместе запустить!

! Но очень важно понимать, что в отличии от скриптов, launch не гарантирует последовательный запуск! Это просто список программ и других launch-файлов, которые надо вместе запустить.

Конкретно в нашем файле мы включаем в него (`include`) два других launch файла, которые запускают симулятор и программы для картографирования по лидару!

Но что это за формат такой? Почему мы как будто пишем путь до файла, а не указываем пакет и т.д.?

Да, у него такой формат, но главное, что нам не надо искать реальный путь до пакета в системе! В этом помогает инструкция `find`, которая, облаченная в `$(...)` позволяет выполнить команду `$(find pkg)` и вместо неё подставится путь до файла в системе. Это ещё и очень переносимый способ, ведь у другого человека расположение пакета `turtlebot3_gazebo` может отличаться!

А дальше мы пишем как и у нас, папка `launch` слэш название файла!

Кстати, список файлов внутри пакета/внутри папки `launch` можно посмотреть с помощью утилиты `rosls`:

```
rosls turtlebot3_gazebo/launch
```

💪 Для закрепления попробуй добавить в наш launch ещё один `include`, который запустит движение по заданию цели и покатайтесь.

Почему мы не добавляем `teleop` в `launch`? Этот узел очень специфичный, он требует работы с фокусом терминала, а у нас от `slam.launch` много сыпется в него, поэтому фокус не получить - надо запускать отдельным терминалом.

Залезем чуть глубже

Получается, если я могу написать launch-файл, то и другие launch-файлы - это просто текстовые файлы с включениями?

Да! И чтобы посмотреть их начинку воспользуемся утилитой `roscat`! Например, залезем в `turtlebot3_world.launch`:

```
roscat turtlebot3_gazebo turtlebot3_world.launch
```

В результате получим:

```
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger,
waffle, waffle_pi]"/>
  <arg name="x_pos" default="-2.0"/>
  <arg name="y_pos" default="-0.5"/>
  <arg name="z_pos" default="0.0"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find
turtlebot3_gazebo)/worlds/turtlebot3_world.world"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>

  <param name="robot_description" command="$(find xacro)/xacro --inorder $(find
turtlebot3_description)/urdf/turtlebot3_$(arg model).urdf.xacro" />

  <node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf" args="-urdf -model
turtlebot3_$(arg model) -x $(arg x_pos) -y $(arg y_pos) -z $(arg z_pos) -param
robot_description" />
</launch>
```

Хмм, а тут немало всего, но новое для нас - это тэги `arg`, `node` и `param`!

А ещё, мы нашли, почему он всё время требует переменную окружения `TURTLEBOT3_MODEL`! Вот и первый момент, который хочется исправить в этих скриптах, чтобы его не задавать!

Так что же это за новые тэги? Разберём кратко:

`arg` - тэг, который позволяет задавать входные аргументы launch-файлу. То есть, можно запустить launch с параметром `x_pos` и это как-то повлияет на запуск. По-умолчанию этот параметр стоит -2.0, поэтому задавать не обязательно. Если не указать `default` атрибут, то надо будет каждый раз при запуске launch указывать!

Хорошим тоном при написании скрипта является написание атрибута `doc`, который объясняет, что это за параметр. Не переживай, это мы тоже поправим.

node - тэг, указывающий запуск программы. Ведь launch - это текстовый файл, который включает инструкции к запуску, а вот **node** (**узел**) в ROS - это сама программа или скрипт, которые содержит исходный код. То есть, сколько не включай один launch в другой, кто-то из них должен стартовать какие-то программы (узлы). Подробнее разберёмся позже!

param - подобно переменным окружения в терминале, в ROS есть **параметры**, которые можно задать и тем самым настроить узлы (программы). Это тоже на сладенько оставим =)

Так, ну теперь мы тут нашли кучу точек для улучшения, значит пора исправлять всё то, что нам не нравится!

👉 Создай свой лаунч `turtlebot3_world.launch` и скопирай туда содержание исходного launch. И теперь пора вносить исправления!

- Замени обращение `$(env TURTLEBOT3_MODEL)` на свою любимую модельку, чтобы она стартовала по-умолчанию.
- Напиши `doc` атрибуты параметрам `x_pos`, `y_pos`, `z_pos`. Попробуй представить, что это за аргументы, за что они отвечают. Может и не угадаешь, зато попытаешься, а обновить всегда сможем после проверки позже =)

После этого проверь, что запуск симулятора из своего пакета работает без задания `TURTLEBOT3_MODEL`.

Отлично, вот так, мы взяли существующий скрипт и улучшили его для себя!

Аргументы launch

Теперь важно разобраться с аргументами, что это и в чём особенности, потому что таким образом мы можем настраивать запуск без изменения содержимого файла. Это очень удобно!

Во-первых, чтобы проверить, какие аргументы есть в launch-файле, у `roslaunch` есть удобная опция `--ros-args`.

:muscle: Посмотри, как выглядит описание аргументов в новом launch с помощью этой опции.

Прекрасно, так можно получить информацию о файле без его открытия. Теперь, как нам задавать их? Ну, тут два пути =)

Первое, мы можем задать значение аргументу при вызове `roslaunch`, делается это так:

В примере мы используем исходный launch, но не забывай, что у тебя ещё более новый и удобный launch!

```
TURTLEBOT3_MODEL=waffle roslaunch turtlebot3_gazebo turtlebot3_world.launch  
z_pos:=4.0
```

Разбираем, запускаем как обычно, но после пишем имя аргумента (name), символ ":" (привет из паскаля) и значение.

! Тут как в bash пробелы низя (нельзя)

:muscle: Попробуй запустить world launch из своего пакета, задавая `x_pos` и `y_pos`. Ожидаем ли меняется положение?

А теперь попробуй задать тип модели. А что если задать тип `tobasko`? Что происходит?

Такс, ручной метод задания аргументов успешно освоен, но теперь надо научиться заданию аргументов при `include` из другого launch!

Тут тоже несложно! Когда определяем `include` внутри launch просто внутри пишем ещё тэги:

```
<include file="$(find turtlebot3_gazebo)/launch/turtlebot3_world.launch" >
  <arg name="z_pos" value="4.0">
</include>
```

Это аналогично команде `TURTLEBOT3_MODEL=waffle roslaunch turtlebot3_gazebo turtlebot3_world.launch z_pos:=4.0`. Удобно, согласись?

Пора применять на практике!

💪 Обнови файл `turtlebot3_for_fun.launch`, чтобы он использовал `turtlebot3_world.launch` из твоего пакета и пропиши новое положение (а может и модель) через аргументы.

Обязательно проверь, что при запуске всё ок.

Вот подлянка, для `turtlebot3_slam/turtlebot3_slam.launch` и `turtlebot3_navigation/move_base.launch` всё ещё нужен `TURTLEBOT3_MODEL`.

💪 А давай перепишем под себя `turtlebot3_slam/turtlebot3_slam.launch`, `turtlebot3_slam/turtlebot3_gmapping.launch`, `turtlebot3_navigation/move_base.launch`? Тогда launch-файлы из нашего пакета не будут требовать `TURTLEBOT3_MODEL` вообще!

Можно и `turtlebot3_teleop/turtlebot3_teleop_key.launch` захватить (переписать для себя), если хочется с клавиатуры управлять и не задавать переменную окружения (тут её вообще можно убрать).

💪 Теперь задание серьёзное, не каждый справляется, но я верю - ты сможешь! Разберись в запуске `turtlebot3_world.launch` и найди, где и что управляет запуском GUI Gazebo (интерфейс). Теперь, в launch добавь аргумент, чтобы управлять этим (назови, например, аргумент `gazebo_gui`). По-умолчанию сделай `false`.

Получилось? А теперь сделай в `turtlebot3_for_fun.launch` тоже аргумент, чтобы можно было управлять этим при запуске `turtlebot3_for_fun.launch`, например:

```
roslaunch super_robot_package turtlebot3_for_fun.launch gazebo_gui:=true
```

Обязательно проверь результат своих трудов! (Подсказка) Чтобы подставить значение из одного аргумента в другой надо использовать конструкцию `value="$(arg foo)"`, где `foo` - это имя аргумента для подстановки.

Воу, если у тебя получилось сделать задание, то ты точно стал гуру аргументов в `launch`!

Штош, вот мы и познакомились с `launch`-файлами, что это и как с этим работать. Дальше нас ждёт много интересного, а пока можно наслаждаться тем, что мы уже смогли поправить и сделать удобнее готовые скрипты - это прекрасный результат!

Задание

Здесь задания пока нет, проверь себя, что в пакете лежат подправленные `launch`-файлы для `turtlebot3`, они работают и не требуют `TURTLEBOT3_MODEL`.

Залей обновления в репозиторий, чтобы не потерять!

Если при ходе по топику что-то непонятно - задай вопрос преподавателям!

Чему научились?

- Файлы `launch` хранятся в папке `launch` (спасибо, кэп!)
- Как писать простейшие `launch`
- Задание аргументов при включении и вручную

Переходим к лидару

Содержание

- Содержание
- Всё начинается с лидара
- Попытка отобразить исходную информацию из лидара - не пытка
- TF 101
- TF - всё относительно
- Болоольше инструментов
 - rqt_tf_tree
 - tf_echo
- TF из Gazebo
- Настройка Rviz
- Чему научились?
- Задание
- Вопросики
- Ресурсы
 - Стандарты

Всё начинается с лидара

Мы с тобой уже научились запускать симулятор и организовывать запуск необходимых узлов и других launch как нам надо.

А ещё и аргументы добавлять, чтобы делать удобное управление запуском!

Это уже хорошие навыки, но давай теперь начнём осваивать каждую часть набора программ (узлов), которые позволяют роботу двигаться к цели, учитывая препятствия!

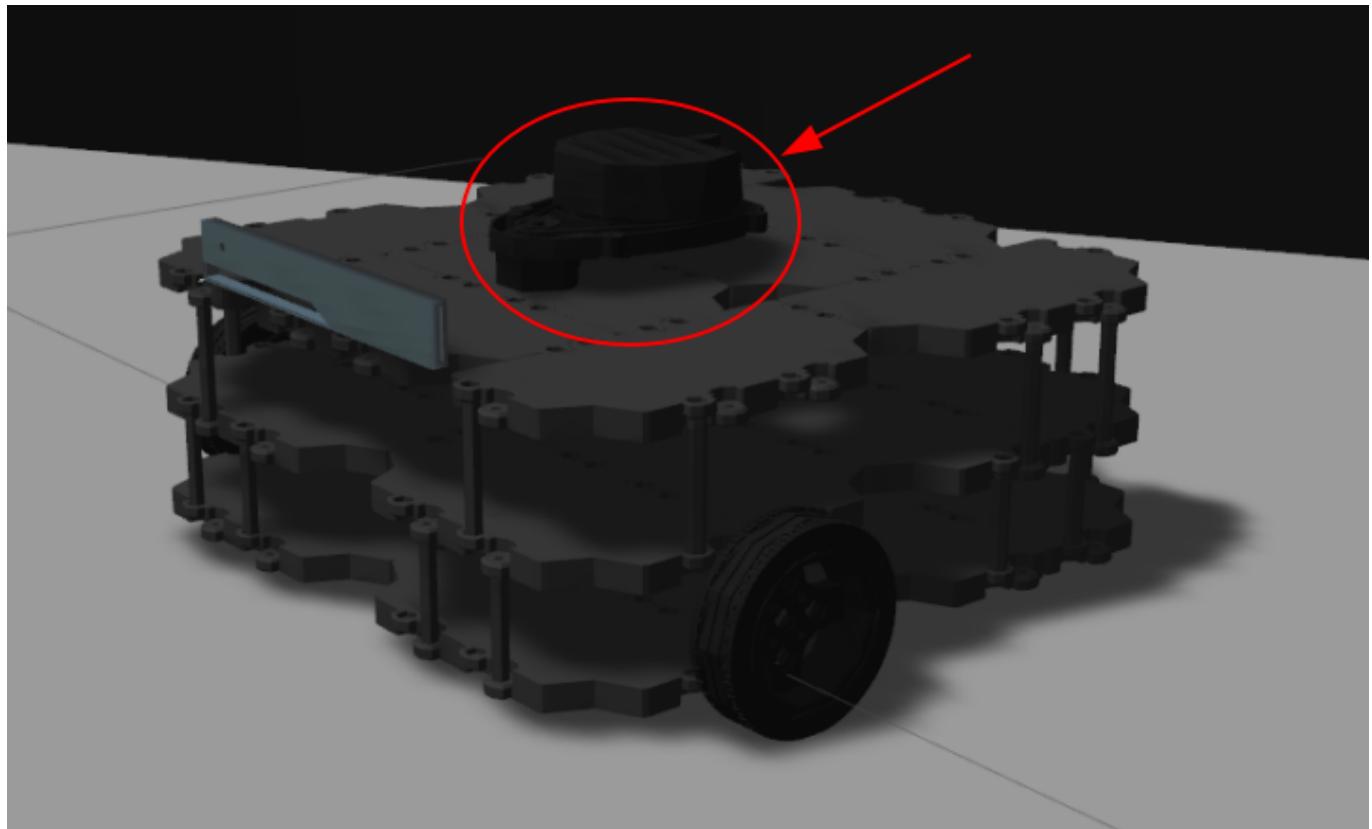
Начнём мы с лидара. На лекции тебе подробнее расскажу о том, что это и как работает, а мы пока будем смотреть, что нам делать, если у нас на роботе уже есть лидар.

Попытка отобразить исходную информацию из лидара - не пытка

Давай запустим наш launch, который стартует симулятор:

```
roslaunch super_robot_package turtlebot3_world.launch
```

В симуляторе мы уже видели, что лидар у нас на вафельке стоит:

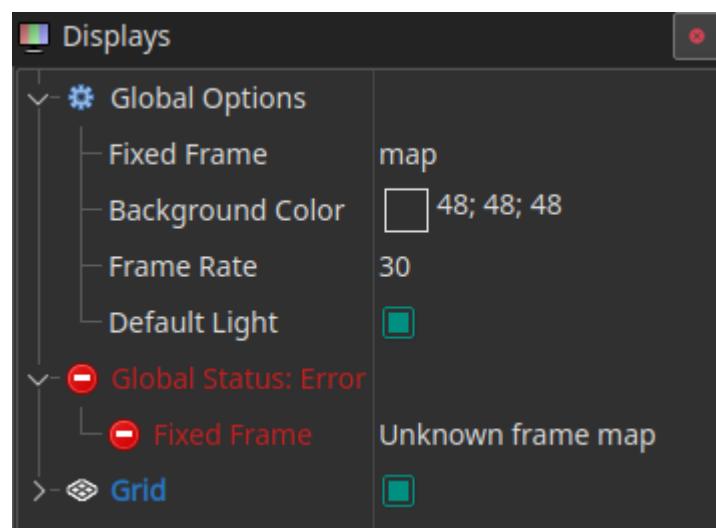


Этот лидар выдаёт информацию в виде точек - расстояний до препятствий вокруг на плоскости сканирования. Как же нам её увидеть в сыром виде?

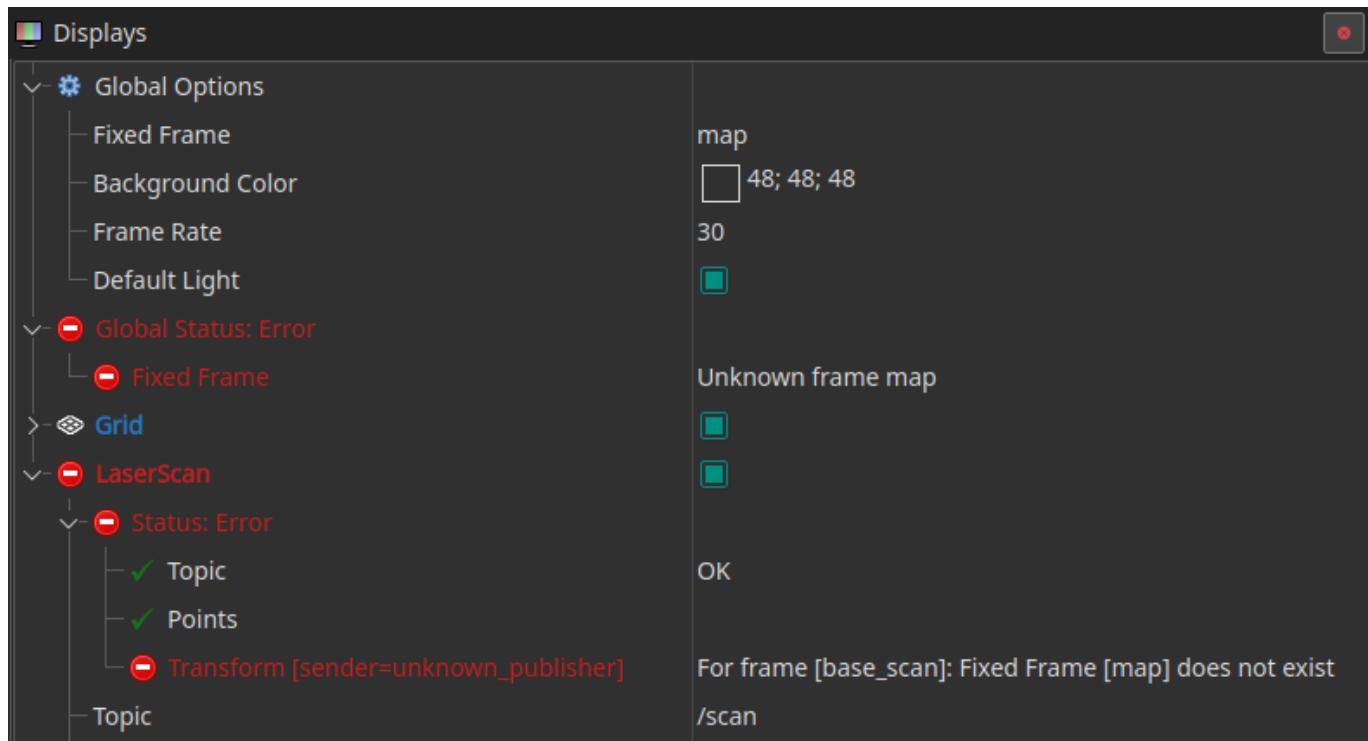
Для этого нам, как обычно, поможет узел, который отображает информацию из робота (как робот видит окружение и какая в нём есть информация) - Rviz.

```
rviz  
# или rosrun rviz rviz
```

Запускаем её и уже видим странную проблему:



Как мы тогда до этого работали с Rviz? Всё же было хорошо. Странно очень, но давай попробуем добавить информацию от лидара для отображения: [Add -> By topic -> /scan -> LaserScan](#).



Ещё какая-то проблема! Ну всё, дальше двигаться нельзя - ничего не показывается и есть ошибки.

Но давай посмотрим на первую ошибку, в "Global Status" ошибка "Unknown frame map". Это, обычно, происходит, если в "Global Options" -> "Fixed Frame" выбран несуществующий **фрейм**.

Что такое **фрейм** - чуть позже разберём, а пока выбери в Fixed Frame, например, `base_footprint`.

О, в "Global Status" теперь "OK"! Это уже результат! Но в LaserScan новая проблема:



Такс, ну тут уже без новых знаний не обойдёмся...

Ну, поехали, узнаем, про какие **фреймы** идёт речь и почему они так нужны для Rviz!

TF 101

TF - это очень удобная система в ROS, которая позволяет работать с системами координат (СК) в пространстве. Принято считать, что TF - это сокращение от "transform". То есть, эта система организует трансформацию между СК.

Например, в руке есть плечо, локоть и кисть, у каждой части есть своё положение в пространстве. Мы знаем, что расстояние от плеча до локтя не меняется, как и расстояние от локтя до кисти. Получается, у нас есть СК в плече, локте и кисти. Между плечом и локтём 30 см и между кистью и локтём 30 см. Вот так мы описали относительное положение между СК!

Сначала прямую руку прижмём к туловищу, рука идёт вниз. Допустим, ось Z идет наверх, X - прямо, Y - влево. Если у плеча $X=0$, $Y=0$, $Z=0$, то у локтя будет порядка -0.3 м по Z, а у кисти около -0.6 м. Всё остальное по нулям ($X=0$, $Y=0$).

Теперь поднимем прямую руку ровно наверх, Z локтя и кисти будут иметь те же значения, но положительные. Тут всё просто, так?

А теперь фокус, плечо сгибаем около 65 градусов, локоть на 35 градусов, какие координаты будут у локтя и кисти?

Если ты очень хорошо помнишь тригонометрию и принципы поворотов систем координат, то тебе не составит труда посчитать.

Дать время подумать? =)

Эта математика расчёта уже давно реализована и как раз TF позволяет нам не беспокоиться о том, что и как надо делать в плане расчётов!

Мы описываем отношения между системами координат (СК) и после этого, построив цепочку из СК, можем относительно одной получать информацию об остальных!

Это очень удобно! Но к чему слова? Нам же надо разобраться с проблемой отображения данных в Rviz, что он хочет?

Во-первых, термин **фрейм** в Rviz - это просто система координат. Тут всё просто.

Во-вторых, что за фрейм **base_scan** он хочет? Тут сложнее, но интереснее!

Лидар выдает нам 360 точек (по точке на каждый градус окружности вокруг) - это просто расстояния от лидара до препятствия, числа.

Но чтобы их отобразить, Rviz надо знать, относительно чего рисовать. Ведь лидар где-то на нашем роботе располагается, а значит надо сообщить rviz, где находится лидар на роботе, чтобы относительно этой точки нарисовать точки препятствий вокруг.

Для этого мы сообщим системе, как располагается **base_link** относительно **base_footprint**. **base_footprint** в данном случае выступает центром робота, спроектированном на пол (высота = 0).

Чтобы сообщить о том, насколько высоко располагается лидар относительно пола, добавим запуск узла, который создаёт **статический TF** между фреймами. Создадим новый launch с названием **turtlebot3_tf.launch**:

```
<node pkg="tf" type="static_transform_publisher" name="base_footprint_2_base_link"  
args="0 0 0.3 0 0 0 base_footprint base_scan 100" />
```

Не забудь, что `<launch>` тэг должен обрамлять весь файл, то есть добавлять надо внутрь него!

Тут мы немного цепляем запуск узлов в launch-файла. Чтобы опубликовать статический TF (сообщить расположение `base_scan` относительно `base_footprint`), нам нужно запустить узел `static_transform_publisher` из пакета `tf`.

Чтобы это сделать мы прописываем в атрибутах тэга `<node>`:

- `pkg` - название пакета, из которого запускаем,
- `type` - название узла, который в пакете надо запустить,
- `name` - как узел будет называться в системе,
- `args` - аргументы узла.

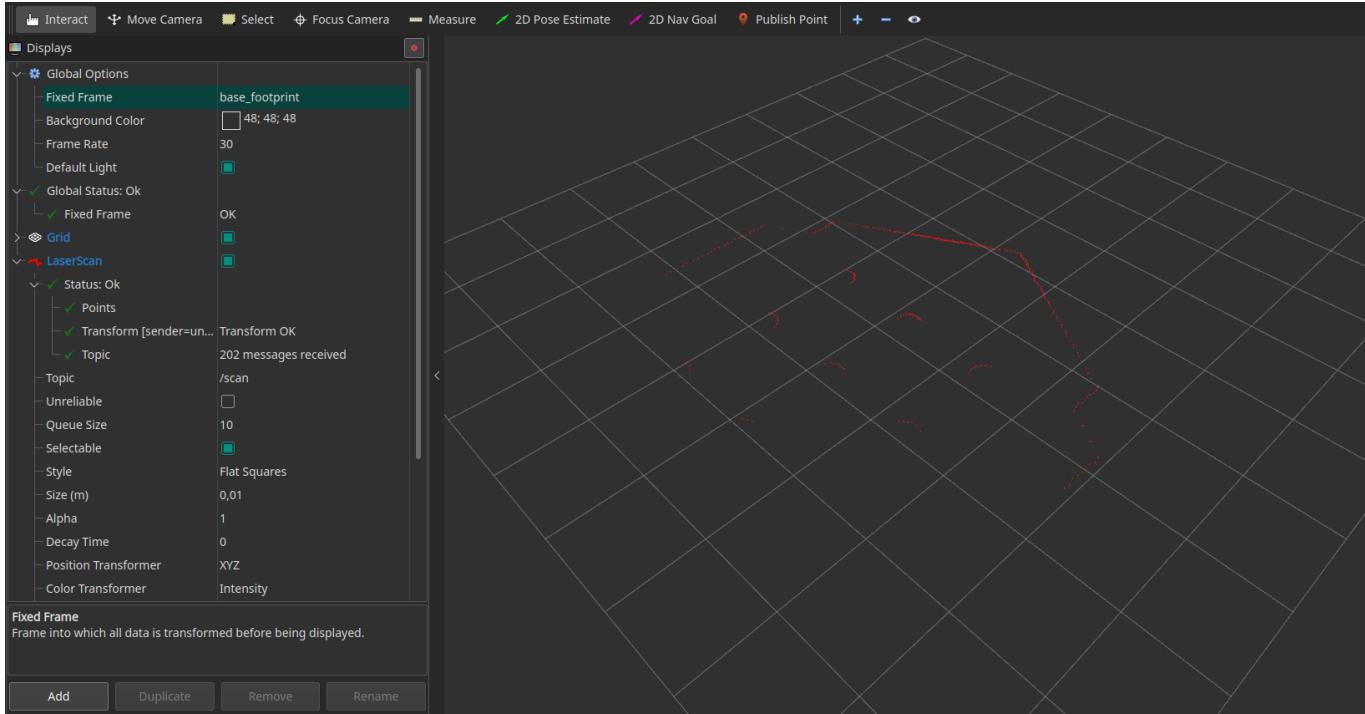
Аргументы у `static_transform_publisher` узла следующие `x y z yaw pitch roll frame_id child_frame_id period_in_ms`. Разберём:

- `x, y, z` - линейное расположение `child_frame_id` относительно `frame_id`,
- `yaw pitch roll` - угловое расположение (поворот) `child_frame_id` относительно `frame_id`,
- `frame_id child_frame_id` - имена фрейма, между которыми создаём TF,
- `period_in_ms` - частота публикации, обычно ставится 100 или 1000, и норм.

Такс, ну мы много тут всего написали, но главное, что нам теперь надо запустить новый launch и посмотреть, помогло ли это отобразить данные из лидара:

```
roslaunch super_robot_package turtlebot3_tf.launch
```

Посмотрим Rviz и увидим, что данные отобразились! Отлично!



Для лучшего отображения точек в LaserScan->"Size (m)" можно поменять размер на 0.03, например

Мы смогли починить недостающую информацию для отображения данных с лидара! В системе не хватало информации о том, как расположен `base_scan` фрейм относительно `base_footprint`!

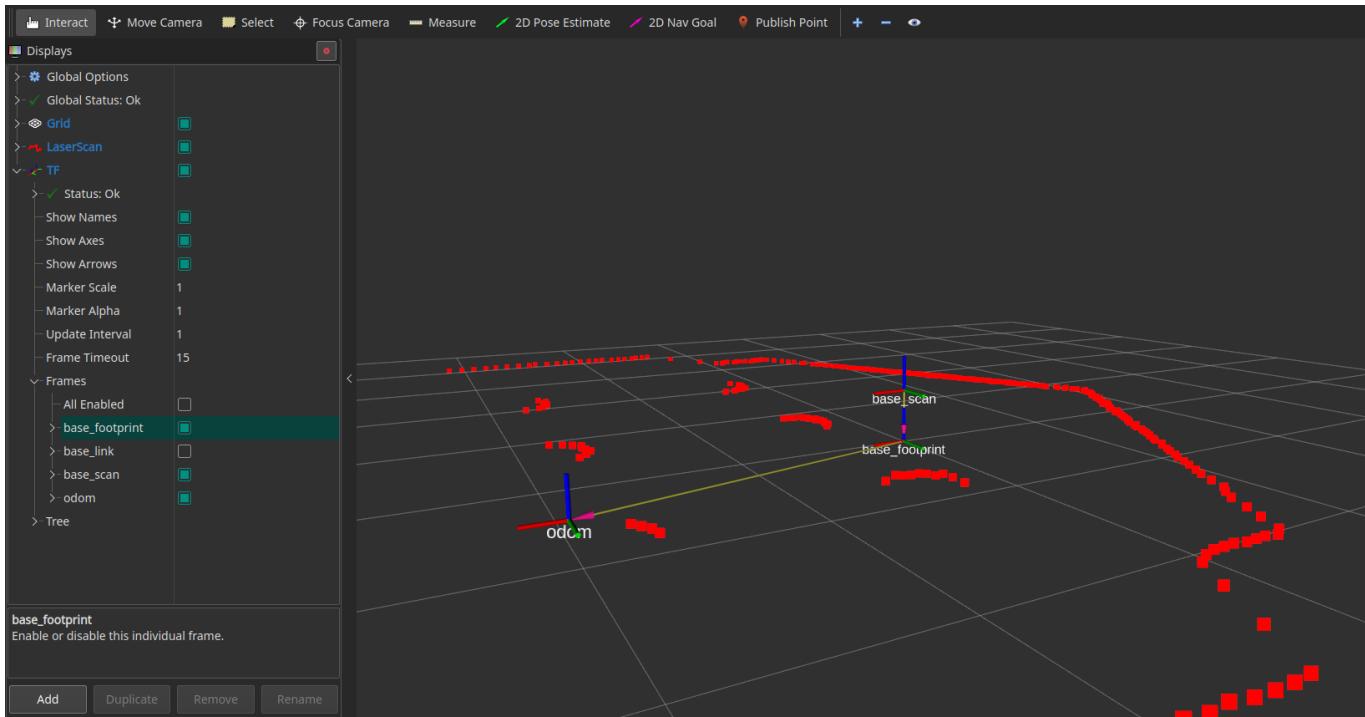
 Попробуй поменять значения `x`, `y`, `z`, `yaw` `pitch` `roll` в аргументах TF узла и перезапускать после изменений launch с этим узлом. Посмотри, как это влияет на отображение.

TF - всё относительно

Результат есть, круто, но не складывается ли ощущение, что мы просто угадали и всё сделали правильно? Ведь, если разбираешься с каким-нибудь инструментом, всегда нужно чётко осознавать, что, как и почему делается!

Во-первых, важно сейчас понять, что СК не существует сама по себе! Именно поэтому мы описываем не расположения фреймов самих по себе, а именно относительное расположение между ними. Так, чтобы отобразить, мы задаём базовый фрейм (Fixed Frame), относительно которого всё и рисуется.

Во-вторых, давай отобразим фреймы в Rviz, чтобы увидеть, как они расположены! `Add -> By display type -> TF:`



Так ведь удобнее, правда?

А теперь попробуй покататься с помощью teleop launch в пакете и скажи, какие из TF (TF - это соотношение между фреймами) являются **статическими**, а каким **динамическими**?

Мы пока не объясняли эти термины, но думаю, тут уже понятно, в чём разница =)

Статический TF (`base_footprint -> base_scan`) – положение между фреймами не меняется во времени, никогда! **Динамический TF** (`odom -> base_footprint`) – положение между фреймами может меняться во времени.

Почему не бывает динамического фрейма? Всё просто, например, фрейм `base_footprint` относится как к динамическому TF, так и к статическому TF. Нет однозначности в этой характеристики.

Так, прекрасно, мы освоили ещё немногих терминов и смогли отрисовать TF в Rviz, чтобы проще было ориентироваться! Отличный результат!

Бооольше инструментов

Сейчас всё идет по плану, но часто в ходе разработки или отладки системы происходят проблемы и всплывают баги настройки, ой как часто...

Более того, мы смогли отобразить TF в Rviz, но в нём не понятно, как увидеть соотношение между фреймами и более того, численных значений преобразований (TF) между фреймами!

Значит, сейчас пока мы не можем в полной мере понять, что происходит внутри системы - погнали осваивать инструменты!

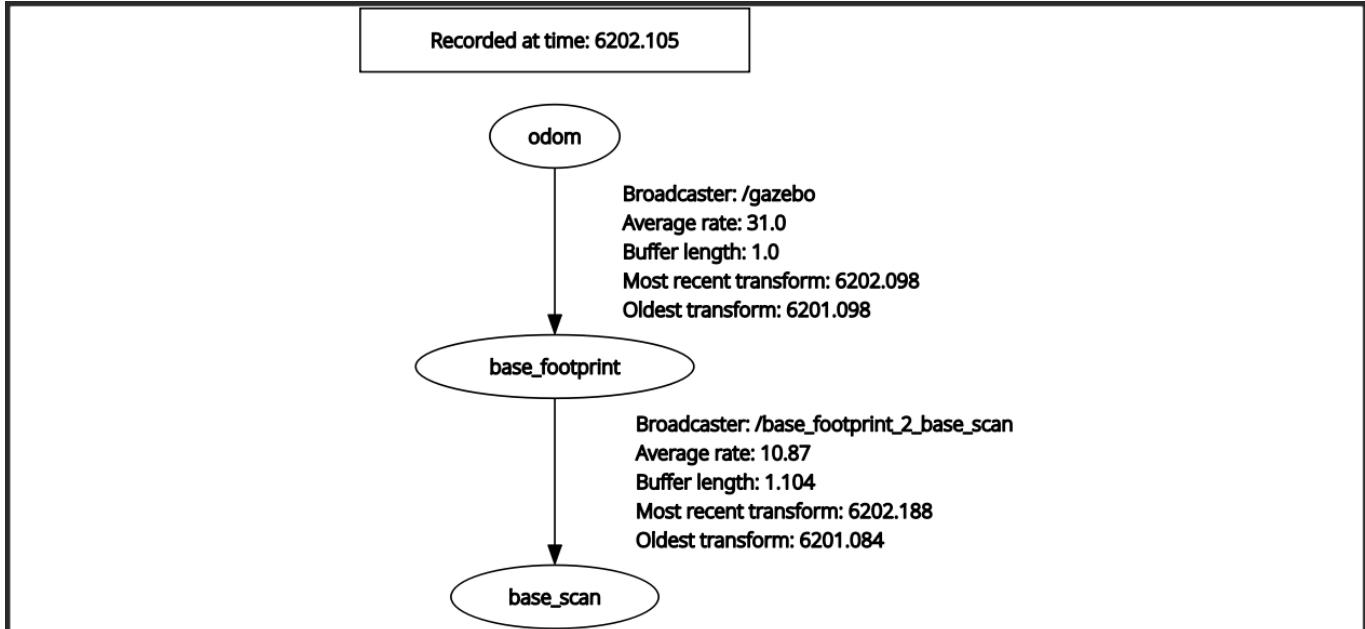
rqt_tf_tree

Первый инструмент поможет нам понять, как соотносятся между собой фреймы и кто является источником информации о TF - `rqt_tf_tree`!

Просто стартуем:

```
rosrun rqt_tf_tree rqt_tf_tree
```

И вот мы видим простой и приятный интерфейс:



В этом интерфейсе мы видим, как фреймы зависят!

Важно, правильное дерево имеет формат **дерева**, то есть у нижестоящих фреймов не может быть два родителя или стрелочки не могут идти наверх!

💪 Попробуй заменить в `turtlebot3_tf.launch` фрейм `base_footprint` на `base_link` и перезапусти. Отобрази `rqt_tf_tree`, видишь проблему? Ошибка в названии ведёт к созданию двух деревьев, такого тоже быть не должно! Верни обратно, как было =)

С помощью этого инструмента можно понять, как взаимосвязаны фреймы и есть ли ошибки в построении дерева TF, так как на этом достаточно часто происходят ошибки.

tf_echo

Другой инструмент позволит нам понять численные характеристики той или иной TF - `tf_echo`.

```
rosrun tf tf_echo base_footprint base_scan
```

И вот результат:

```

At time 6673.459
- Translation: [0.000, 0.000, 0.300]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
              in RPY (radian) [0.000, -0.000, 0.000]
  
```

```
        in RPY (degree) [0.000, -0.000, 0.000]
At time 6674.462
- Translation: [0.000, 0.000, 0.300]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
    in RPY (radian) [0.000, -0.000, 0.000]
    in RPY (degree) [0.000, -0.000, 0.000]
At time 6675.463
- Translation: [0.000, 0.000, 0.300]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
    in RPY (radian) [0.000, -0.000, 0.000]
    in RPY (degree) [0.000, -0.000, 0.000]
```

Как видно, статическая TF не меняется - логично! =)

 Отобрази TF "odom -> base_footprint". Посмотри значения между ними.

 Разберись, как в tf_echo задать частоту отображения данных

Отлично! Вот у нас и получилось понять соотношение между фреймами, а также получить информацию о конкретном TF в численном виде, а не "на глаз"!

TF из Gazebo

Важной хитростью работы с симулятором является то, что модель в симуляторе уже описана с учётом расположения датчиков и деталей между собой. Поэтому, в Gazebo есть узел, который может опубликовать все статические TF, описывающие робота...

Мы не скрывали это от вас, просто важно с некоторыми вещами разобраться поподробнее!

Давай отключим все launch и создадим файл, в который добавим:

- Запуск world.launch
- Запуск публикации информации о расположении частей робота (TF)

```
<node pkg="robot_state_publisher" type="robot_state_publisher"
      name="robot_state_publisher">
    <param name="publish_frequency" type="double" value="50.0" />
</node>
```

- Запуск rviz

```
<node pkg="rviz" type="rviz" name="rviz" />
```

► Подсказка

```

<launch>
  <include file="$(find super_robot_package)/launch/turtlebot3_world.launch" />

  <node pkg="robot_state_publisher" type="robot_state_publisher"
    name="robot_state_publisher">
    <param name="publish_frequency" type="double" value="50.0" />
  </node>

  <node pkg="rviz" type="rviz" name="rviz" />
</launch>

```

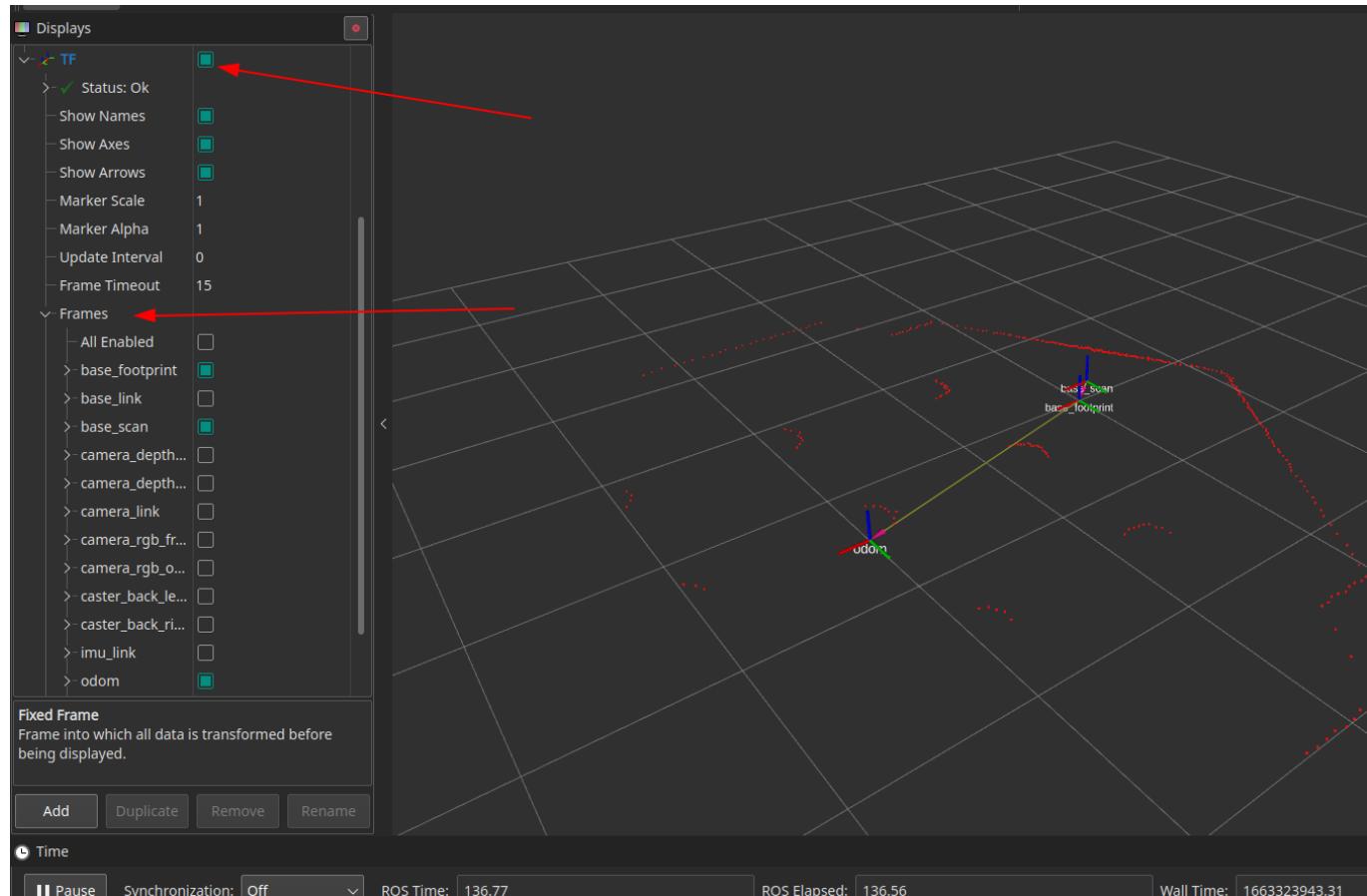
А теперь стартуем его!

```
roslaunch super_robot_package turtlebot3_sim_start.launch
```

Ах да, Rviz же после перезапуска сбивается...

Ну, настрой снова TF, LaserScan и Fixed Frame.

Хм, а фреймов стало гораздо больше, как же увидеть всё в этой мешанине? Не сложно, просто открой параметры TF и настрой те TF, который хочется видеть ([odom](#), [base_footprint](#), [base_scan](#)).



Отлично, так значит, можно сразу получить описание робота без необходимости прописывать каждый TF руками?

Не совсем, в этом случае расположения фреймов описаны в симуляторе, а значит в нём есть вся необходимая информация!

Если у тебя робот без модели в симуляторе, то придётся описывать ручками, но это не так страшно, как кажется! Берём рулетку и в бой =)

На деле у нас появился launch-файл, который стартует все необходимое в симуляторе, теперь можно идти дальше и изучать другие аспекты работы системы ROS на роботе!

Но погоди отключать настроенный Rviz!

Настройка Rviz

Такс, не спешим, у нас осталась одна нерешённая проблемка...

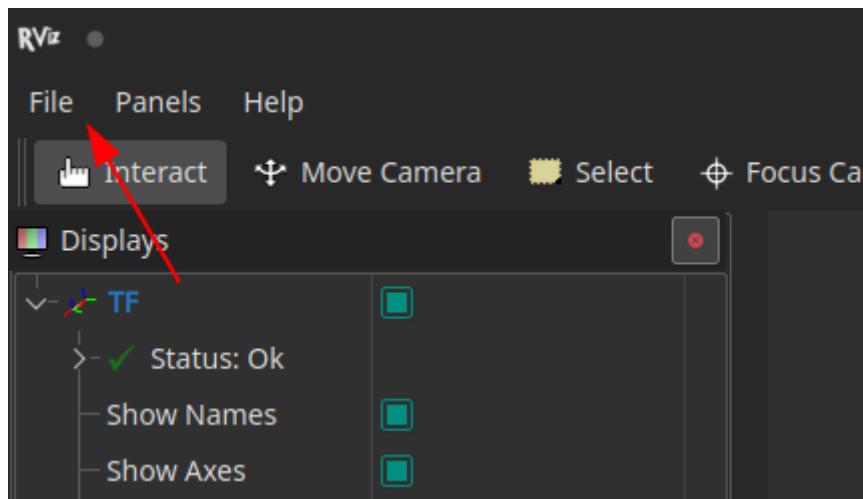
Мы тут перезапустили Rviz и приходится каждый раз настраивать! Это же уйма времени будет потрачена! Не годится.

Давай посмотрим `rviz -h` и найдём очень полезную строку:

```
-d [ --display-config ] arg A display config file (.rviz) to load
```

То есть, в rviz есть опция, которая позволяет подгрузить конфигурацию отображения из файла? Хм, это может быть полезно!

Закончи настройку отображения информации как тебе хочется (может точки больше сделаешь?) и жмём кнопку **File->"Save Config As"** (сверху слева):



Не нажимай "Save Config" или сочетание "Ctrl+S", так как это сохранит нынешнее отображение как настройки по-умолчанию!

В окне сохранения пройди в папку пакета и там создай папку `rviz` и туда сохрани в файл под названием `sim_initial.rviz`.

А тепеерь, магия, погнали в `turtlebot3_sim_start.launch` и там к запуску Rviz добавь `args="-d $(find super_robot_package)/rviz/sim_initial.rviz"`. Это по сути путь до нашей конфигурации в

пакете.

Добавил? Отлично, перезапускай!

Всё на месте! Теперь тебе не придется каждый раз настраивать Rviz! А ещё тут есть приятный сюрприз, если поменять что-либо в отображении и нажать "Ctrl+S", то это сохранится и в следующем запуске будет тут же! Шикарно!

А всего-то поняли, что есть проблема и в два счёта решили ее, проверив `help`!

! Практика показывает, что создание rviz конфигураций под каждую задачу является удобным способом хранения. Не пытайтесь держать одну конфигурацию под все задачи, например, "проверка работы модели в симуляторе", "отображение данных с лидара", "отображения инфы от планировщика" и т.д.

Чему научились?

- Настройка Rviz
- Ознакомились с понятием TF и её удобствами
- Смогли сделать статическую публикацию TF в launch
- Научились отображать TF в Rviz
- Освоили инструменты `rqt_tf_tree` и `tf_echo`
- Узел `robot_state_publisher` для публикации информации о TF робота из описания в симуляторе

Задание

Не забудь залить все свои обновления в Git и на сервер!

- Попробуй отобразить другую информацию в Rviz - теперь ты это умеешь! Посмотри, что происходит с остальными фреймами при движении.
- Посмотри дерево TF после перехода на `robot_state_publisher`.
- Посмотри значения TF между разными фреймами, все ли значения понятны?

Вопросики

- Почему мы "объясняли" системе, как расположен `base_link` относительно именно `base_footprint`? Почему не относительно `odom`?
- В чём разница статических и динамических TF?
- Почему фрейм нельзя характеризовать статическим/динамическим?
- Как сохранить настройку Rviz? Как её восстановить?
- * Что такое кватернион?

Ресурсы

- [Хорошая статья с примером описания основных фреймов робота](#)

Стандарты

Вообще, это просто два основных стандарта касательно фреймов и TF, но мы очень рекомендуем их почитать:

- [REP-105](#) - описание основных фреймов;
- [REP-103](#) - описание основных единиц измерения и координатных преобразований.

Узлы и топики

Содержание

- Содержание
- Привет
- Запустим всё для управления
- Программы - узлы
- Каналы передачи данных - топики
- RQT Graph
- Запуск
- Master
- Чему научились?
- Вопросики

Привет

Итак, мы уже касались вопроса программ в ROS. Не раз запускали Gazebo и Rviz, но теперь пара раз и навсегда определись с понятиями **узел** и **топик**.

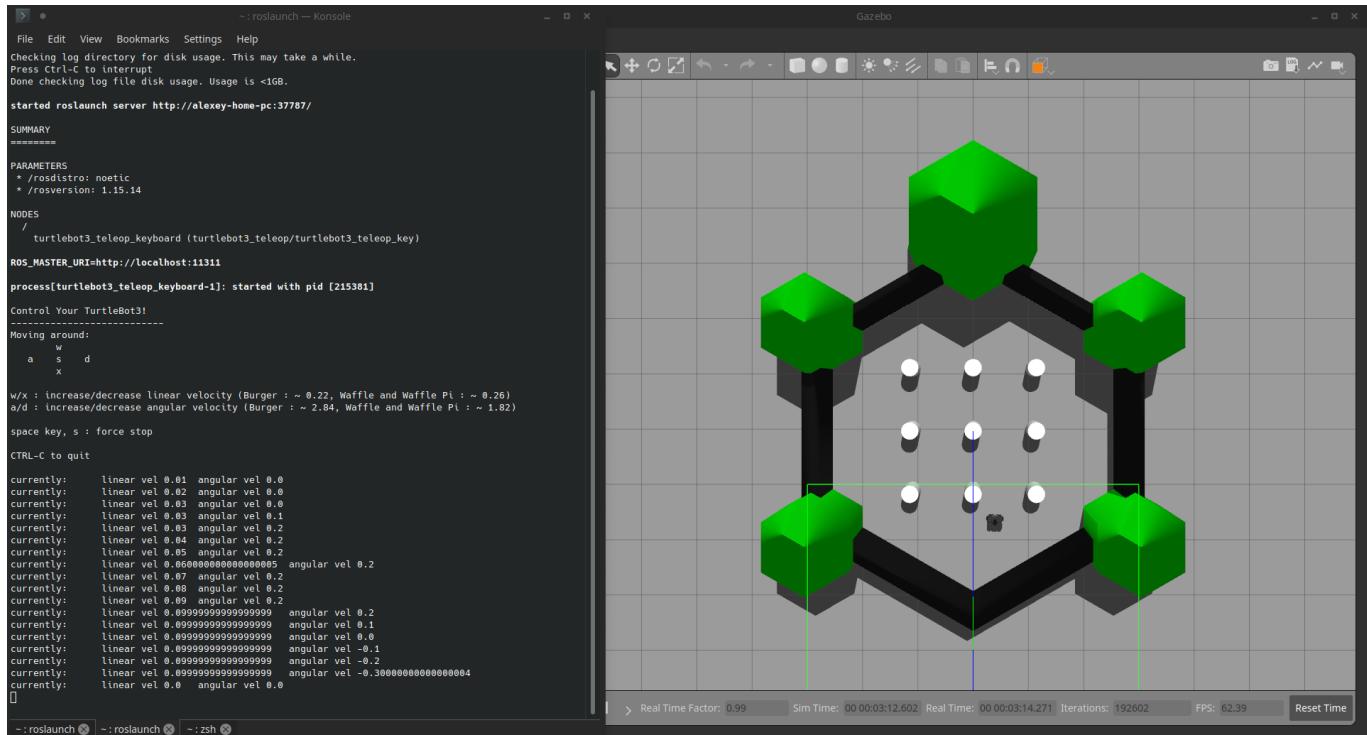
А делать мы это будем путём выяснения, как работает teleop управление и как задаются команды управления!

Запустим всё для управления

Давай запустим симулятор и включим телеуправление с клавиатуры. Конечно, когда запустили, надо немного покататься! Естественно, чтобы убедиться в работоспособности, а не только, чтобы развлечься =)

```
roslaunch super_robot_package turtlebot3_world.launch
# Следующая команда в отдельном терминале
roslaunch super_robot_package turtlebot3_teleop_key.launch
```

Итого, мы имеем симулятор (Gazebo) и управление с клавиатуры:



Шикарно, начнём!

Программы - узлы

Уже говорили, что программы и скрипты в ROS называются узлами (nodes). Это всё, что надо знать про определение =)

Что интереснее, как с ними взаимодействовать?

Для начала, надо понять, что при запуске робота запускается сразу несколько узлов, а значит нам нужен способ посмотреть весь список запущенных узлов.

В любой работе относительно узлов нам поможет утилита `rosnode`:

```
rosnode list
```

В выводе видим:

```
/gazebo
/gazebo_gui
/rosout
/turtlebot3_teleop_keyboard
```

Прямо сейчас запущено всего 4 узла, пройдёмся, что это за узлы:

- `gazebo` - сервер симуляции, по сути, делает все вычисления физики симулятора;
- `gazebo_gui` - графический интерфейс (GUI) для симулятора, если помнишь, его можно отключить;

- `rosout` - это узел, который стартует всегда в системе ROS, он обрабатывает вывод из других узлов (консольный вывод);
- `turtlebot3_teleop_keyboard` - наш узел управления с клавиатуры.

Вроде несложно, так?

Тогда, чтобы разобраться с тем, как работает управление с клавиатуры, нам надо глубже копнуть в информацию об узле управления, для этого есть команда `info`:

```
rosnode info /turtlebot3_teleop_keyboard
```

Вывод

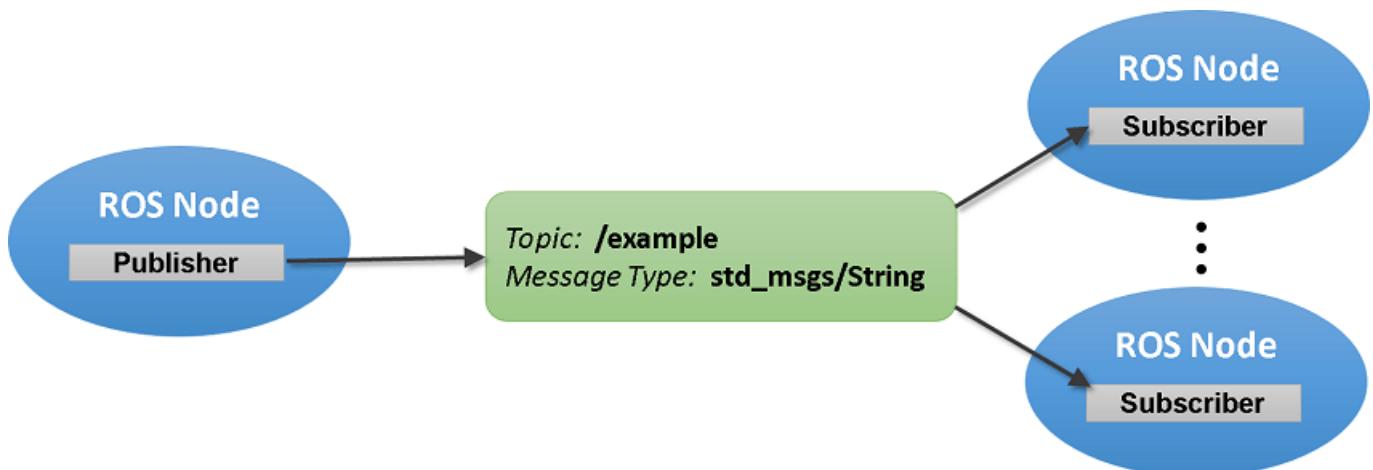
```
-----
Node [/turtlebot3_teleop_keyboard]
Publications:
* /cmd_vel [geometry_msgs/Twist]
* /rosout [rosgraph_msgs/Log]

Subscriptions:
* /clock [rosgraph_msgs/Clock]
```

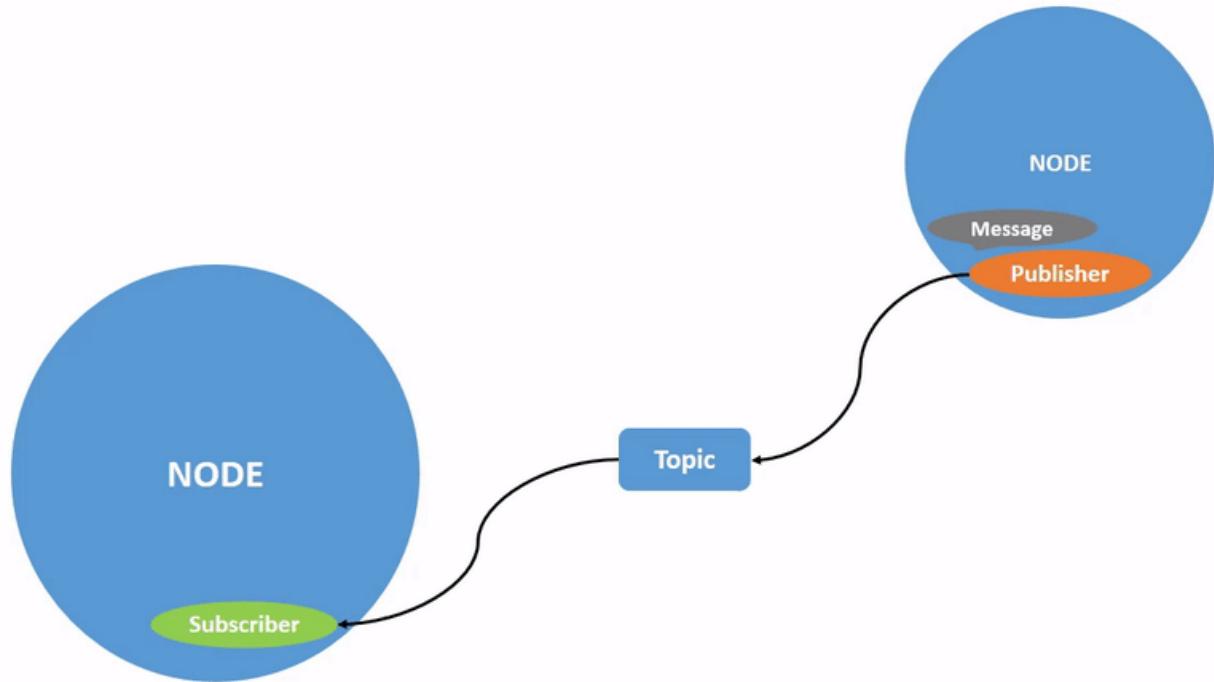
Разберём-с, узел управления с клавиатуры имеет "Publications" и "Subscriptions". Что-то публикуется и на что-то он подписывается. Вот тут мы начинаем пересекаться с понятием **топика**. Пойдём разберёмся с ними!

Каналы передачи данных - ТОПИКИ

Давай взглянем на то, как организовано общение между узлами в ROS:



Есть ещё такая gif-ка:



В ROS каждый узел выполняет свою задачу, но они не могут работать без коммуникации между собой. Так вот топики - это один из способов общения между узлами. Как видно на картинке, узлы могут публиковать в топики (Publication - отправлять через него данные) и подписываться на топики (Subscription - получать через него данные).

Ну, а в нашем примере, `teleop` подписывается на `/clock`, а публикует в `/cmd_vel` и `/rosout`.

Если `/clock` и `/rosout` - это сервисные топики, то `/cmd_vel`, полагаем, отправляет интересную информацию!

Давай проанализируем топики в системе с помощью утилиты `rostopic` и команды `list`:

```
rostopic list
```

Вот видим в выводе много разных топиков и в частности:

```
...
/clock
/cmd_vel
...
```

Ну, просто получить список полезно, но мы также можем и получить информацию о конкретном топике командой `info`:

```
rostopic info /cmd_vel
```

Вывод:

```
Type: geometry_msgs/Twist  
  
Publishers:  
* /turtlebot3_teleop_keyboard (http://alexey-home-pc:34797/)  
  
Subscribers:  
* /gazebo (http://alexey-home-pc:34455/)
```

Вот так мы можем видеть, что узел публикации подтвердился, [/turtlebot3_teleop_keyboard](#), а ещё мы увидели, кто подписался на топик, [/gazebo](#) - симулятор!

Давай теперь сами подпишемся на топик и посмотрим, какая информация ходит через него?

```
rostopic echo /cmd_vel
```

И у нас непрерывно покатился вывод:

```
...  
linear:  
  x: 0.0  
  y: 0.0  
  z: 0.0  
angular:  
  x: 0.0  
  y: 0.0  
  z: 0.0  
...
```

А вот это уже интересно, давай попробуем поуправлять в терминале телеуправления и параллельно смотреть на вывод из подписки [/cmd_vel](#):



Смотри, значения меняются! Значит, через этот топик передаётся информация о команде на движение. Если более конкретно, то через [linear/x](#) поле передаётся желаемая линейная скорость, а через [angular/z](#) - команда на скорость поворота.

! Таким образом, топик - это канал передачи потоковой информации (так как передаем непрерывно). Данные внутри передаются с определённой структурой, которая регламентирована **тиปом сообщения у топика**. Это видно и на картинке, а в нашем случае [/cmd_vel](#) имеет тип "geometry_msgs/Twist". О каждом типе можно почитать в [справке](#) или через команду [rosmsg show geometry_msgs/Twist](#).

Для аналогии можно привести конвейерную ленту, через которые передаются коробки конкретной формы и размера. информацию о коробке позволяет получить утилита `rosmsg`.

 Посмотри, из каких подтипов состоит сообщение `geometry_msgs/Twist` (`geometry_msgs/Vector3`). Найди информацию об этом типе в справке и в `rosmsg`.

А каким образом определить частоту передачи этого потока? Ведь в поток публикуется с каким-то периодом, мы же все знаем про дискретные системы. Попробуй разобраться самостоятельно:

 Определи команду у `rostopic`, которая позволяет вывести частоту публикации в топик.

Вот так мы в общих чертах познакомились с узлами и топиками, но это были цветочки, а теперь будут ягодки!

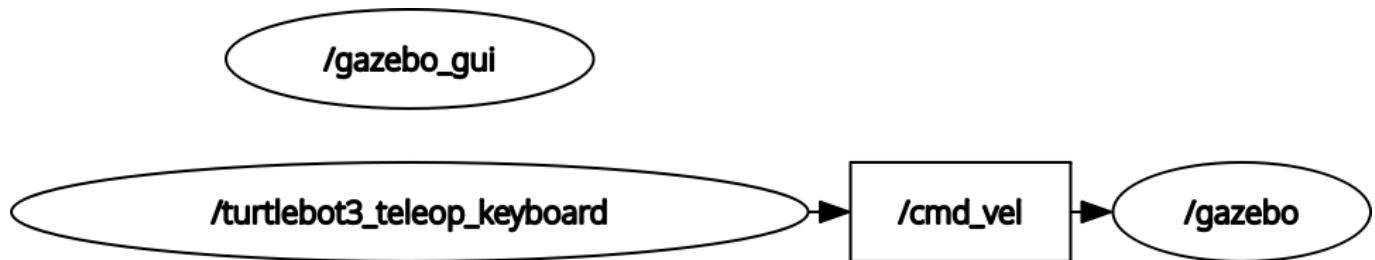
RQT Graph

Это всё хорошо, но погодите, нам каждый раз, когда хотим посмотреть как и куда направляется информация, надо по каждому топику и узлу выводить информацию?

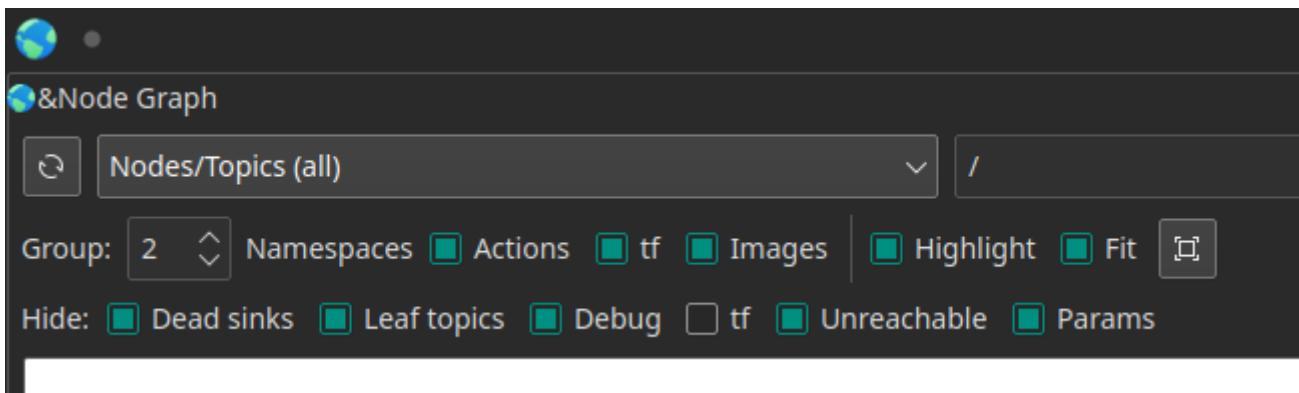
Отличный вопрос, конечно же нет! Для этого сделана удобная утилита `rqt_graph`, которая показывает, как и через что связаны узлы, проверим:

```
rqt_graph
```

И видим:



Убедись, что настроено меню сверху вот так, иначе много лишних топиков показывается:



Сейчас у нас немного информации между узлами передается, но в сложных системах такие диаграммы позволяют найти ситуации очепяток и других проблем, которые не дают системе нормально работать.

Вот такой простой и удобный инструмент!

Запуск

Мы уже увидели, что узел teleop передает информацию через топик `/cmd_vel`, даже посмотрели эти сообщения, определили тип, описание.

Да мы даже запускали уже сколько узлов, но пора разобраться раз и навсегда, как делается запуск узлов!

Итак, запуск в консоли делается командой `rosrun` с аргументами **имя пакета** и **имя узла**. Например, для запуска `turtlebot3_teleop_key` из пакета `turtlebot3_teleop` делается командой:

```
rosrun turtlebot3_teleop turtlebot3_teleop_key
```

Аналогично, запуск из launch той же команды делается определением:

```
<node pkg="turtlebot3_teleop" type="turtlebot3_teleop_key"
      name="turtlebot3_teleop_keyboard">
</node>
```

Ещё раз разберём семантику атрибутов тэга `<node>`:

- **pkg** - название пакета, из которого запускаем,
- **type** - название узла, который в пакете надо запустить,
- **name** - как узел будет называться в системе,
- **args** - аргументы узла (здесь не представлены).

Отлично, как запускать какие-то узлы мы узнали, но давай подумаем с практической точки зрения. Если у сообщений через топики есть конкретный тип, а значит и формат, то мы можем использовать любой узел, который сможет публиковать сообщения с тем же типом в топик.

В таком случае мы можем заменить узел управления на другой, более удобный.

Посмотрим на этот [узел телеуправления](#).

Попробуем установить его через apt:

```
sudo apt install ros-noetic-teleop-twist-keyboard
```

А теперь отключим наш исходный узел телеуправления и запустим новый!

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

И, поехали! Обрати внимание, кнопки для управления другие.

Отлично, мы опробовали запуск через `rosrun`, а теперь попробуем сделать запуск с указанием предельных значений через `launch`. Создадим файл `turtlebot3_teleop_new.launch` и в нём пропишем запуск нового узла с параметризацией, как задано на странице GitHub:

```
<launch>
  <node pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py"
name="turtlebot3_teleop_keyboard" output="screen">
    <param name="speed" value="0.4" />
    <param name="turn" value="0.7" />
  </node>
</launch>
```

Про **параметры** узла мы ещё не говорили, но уже сейчас хочется настроить управление для более удобного контроля робота. Поэтому, настраивай конфигурацию под себя и поехали дальше!

Master

Последнее, что нам нужно узнать сегодня про узлы - это из мааленькая специфика работы. Зовётся эта маленькая, но очень важная специфика - **мастер**.

Закрой все узлы и попробуй запустить новый узел управления:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

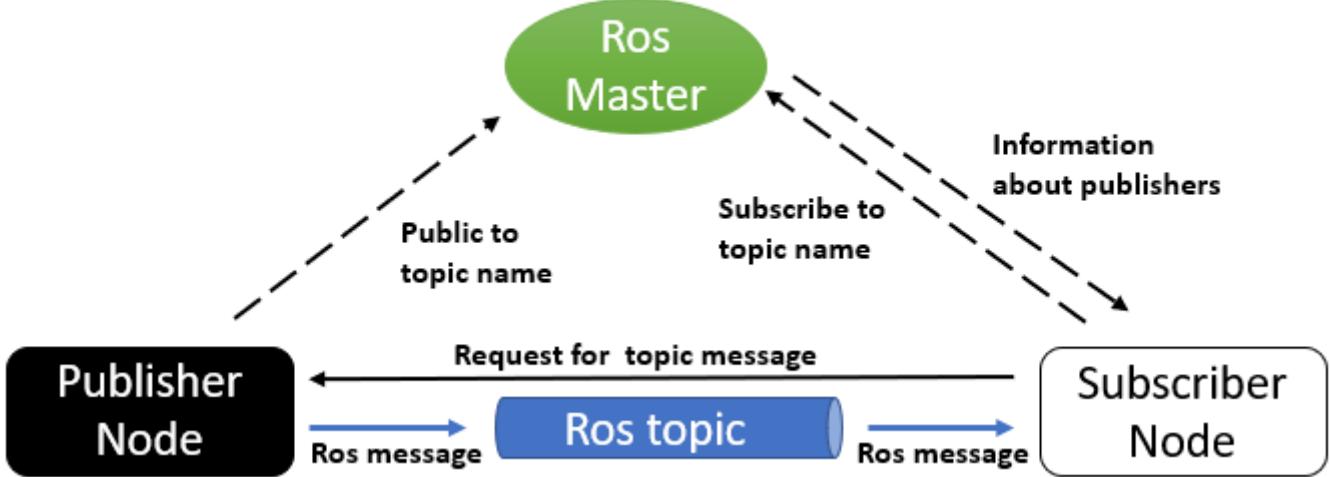
Или `rviz`, или `rosnode list`.

Все они говорят о следующих ошибках:

- `Unable to register with master node [http://localhost:11311]: master may not be running yet.`
- `ERROR: Unable to communicate with master!`
- `Could not contact ROS master at [http://localhost:11311], retrying...`

Все сообщают о том, что мастер не запущен! Да кто же этот ваш *мастер*?

Всё просто, посмотри на эту схему:



В части топиков и узлов, мастер выступает как хранитель информации. Когда узел публикует информацию, он сообщает, какой тип сообщений топика и как зовется топик для публикации мастеру. Аналогично при подписке узел запрашивает у мастера информацию по топику, есть ли топик с таким именем, какой у него тип (проверяет соответствие) и т.д.

Вот так мастер стоит между подпиской и публикацией, чтобы организовать правильно все каналы связи!

Почему тогда всё работает, если сначала запустить launch через `roslaunch`? Roslaunch - это хитрая утилита, которая запускает мастера, если он ещё не запущен. Вот так, без каких-либо вопросов =)

Можно ещё в логе старта увидеть строку "auto-starting new master"

А что же делать, если надо запустить один только узел без `roslaunch`? Тут тоже всё просто, надо сначала запустить `roscore`! Попробуй сначала стартануть `roscore`, а потом запускать узлы, что запускали ранее! Видишь, всё работает!

! Итого, мастер существует, пока запущен `roscore` или пока работает первый `roslaunch`, который его запускает.

Чему научились?

- Получилось запустить новый узел телекоманд
- Освоен новый инструмент визуализации `rqt_graph`

Вопросики

- Что такое rosmaster? Зачем он нужен? Почему без него не стартуют узлы?
- Почему при запуске launch не нужно стартовать roscore?
- Какая команда `rostopic` отвечает за определение частоты публикации в топик?

Одометрия, что это и с чем едят?

Содержание

- [Содержание](#)
- [Привет!](#)
- [Зачем нужна одометрия?](#)
- [Пора буксовать](#)
- [Чему научились?](#)
- [Задание](#)
- [Вопросики](#)

Привет!

На данный момент ты неплохо разбираешься во внутренней кухне программной части робота: узлы (ноды), топики, TF, конфигурация Rviz, написание своих launch.

Всё это позволяет уверенно идти дальше и разбираться в вопросе, как организовано движение робота во внешней среде!

Сегодня мы затронем немного теоретическую, но очень важную тему - одометрия в роботе, так как в ROS для неё выделен отдельный фрейм в REP-105 и вообще куча материала!

Зачем нужна одометрия?

Если посмотреть [инфу о фреймах в стандарте](#), то можно заметить три основные фрейма, которые выделяют относительно движущихся роботов: `base_link`, `odom`, `map`.

- `base_link` часто ставят вместе с `base_footprint` за единственной разницей, что `base_link` находится где-то в центре робота, а `base_footprint` - на уровне Z=0 (на уровне земли). То есть, `base_{link/footprint}` - это фрейм, привязанный к движущемуся роботу.
- `odom` - интересующий нас фрейм, он показывает положение в пространстве, относительно которого робот движется.
- `map` - фрейм, который привязан к фиксированной точке в пространстве, относительно которой движется робот.

Все определения фреймов даны в собственной интерпретации и не претендуют на полноту.

Так, а не кажется тебе, что `odom` и `map` уж очень похожи? По сути так, но у них есть одно маленькое различие. `odom` фрейм - это определённая точка в пространстве, относительно которой показывает движение наша **система одометрии в роботе**.

Отличительной особенностью `map` является то, что этот фрейм более стабилен и относительно него TF формируется не только системой одометрии, но и системам позиционирования (типа картографирование, коррекция по карте и т.д.).

Так, кажется, это сложно, простой пример!

Вот есть вафелька и у неё есть два колеса. Мы начинаем движение и говорим ей ехать 20 секунд. Теперь, вафелька проехала какое-то расстояние, так? Но как узнать, какое?

У нас есть три источника данных: рулетка, оценка перемещения по построенной карте и колёсная одометрия.

Ну, думаю, понятно, что мы делаем автономных/телеуправляемых роботов, а значит с **рулеткой** не побегать - это вариант отпадает, так как не применим в ходе работы робота.

Колёсная одометрия - это неплохой метод оценить перемещение робота. У нас же есть формула длины окружности? Есть информация о диаметре. По-любому в микроконтроллере подключён энкодер с колёс, который знает, сколько оборотов робот сделал за 20 секунд. Значит можно вычислить пройденный прямо путь?

Например, 3 оборота, длина окружности колеса ~ 0.32 м. Значит, робот проехал 0.96 м. Это показания колёсной одометрии!

Карта помещения строится, например, с помощью лидара - это мы видели и пробовали. Значит, Робот (система картографирования) может соотнести новый скан от лидара и существующую карту и понять, что робот переместился.

Положим, что система карты говорит нам, что робот переместился на 1 м.

Вот так получается, что одометрия говорит о перемещении на 0.96 м, а карта говорит о перемещении на 1 м!

Тогда TF `odom->base_link` будет X=0.96, а что будет с `map->base_link`?

Система картографирования сместит TF `map->odom` на 0.04 м, чтобы TF `map->base_link` получился X=1.0.

О том, как работает система карт с фреймом `map` мы поговорим в другой теме, но главное, что `odom` фрейм и TF `odom->base_link` показывает, как смещается робот в мире, судя по показаниям одометрии. Во многих системах есть ошибки расчёта математики одометрии (не любое движение можно по длине окружности посчитать), а также, робот может попросту пробуксовывать, а система внутри него будет думать, что робот всё ещё движется!

Давай проверим это на практике!

Пора буксовать

Нам надо убедиться, что колёсная одометрия не всегда отражает корректные показания и подвержена влиянию разных факторов. Сейчас мы проведем эксперимент, в котором упремся в столб и будем дальше двигаться в него. Во время этого движения посмотрим, как отрабатывает система одометрии по колёсам.

В нашем симуляторе вафельки уже включена колёсная одометрия, но разработчики пакета черепашки сделали хитро. В качестве источника информации используется информация из симулятора, что не соответствует работе в реальному мире. Нужно в качестве источника информации сделать энкодер, а не информацию из симулятора.

Можно не делать следующие действия, если переживаете, результаты мы показали в виде gif в конце раздела!

Для приведения модели к поведению как в реальном мире надо выполнить следующую команду, которая находится в файле описания модели источник одометрии и заменяет на `encoder` источник.

Всегда аккуратно относитесь к командам, которые делаются с `sudo`. Лучше лишний раз переспросить, если не уверены в том, что команда делает.

```
sudo sed -i -E "s/(<odometrySource>).*(</odometrySource>)/\1encoder\2/"  
"$(rospack find turtlebot3_description)/urdf/turtlebot3_waffle.gazebo.xacro"
```

`sed` - это очень мощная утилита для поиска и замены строк текста в файле.

- ▶ Чтобы вернуть обратно на то, как было [клик]

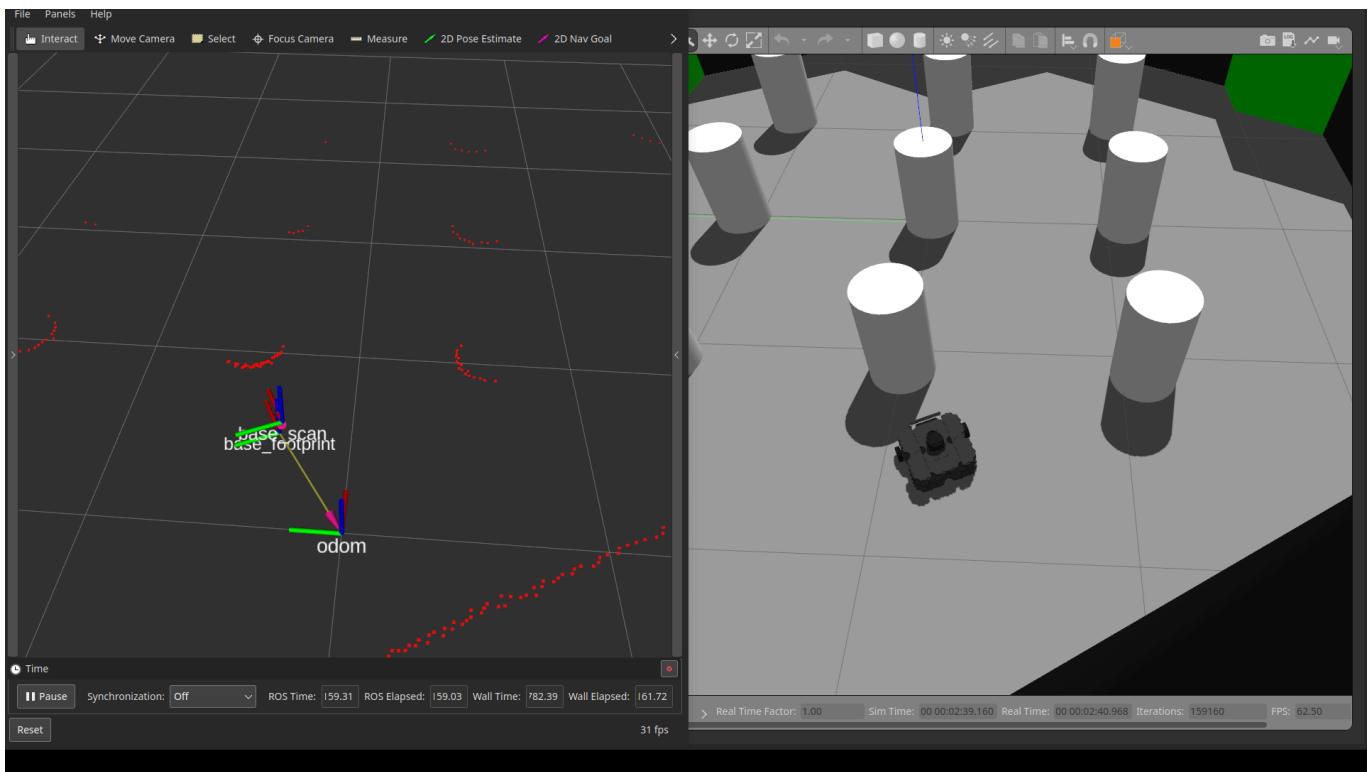
```
```bash  
sudo sed -i -E "s/(<odometrySource>).*(</odometrySource>)/\1world\2/" "$(rospack
find turtlebot3_description)/urdf/turtlebot3_waffle.gazebo.xacro"
```
```

После этого запускай наш симулятор (`turtlebot3_sim_start.launch`), управление на клавиатуре (на твой выбор). После этого настрой в Rviz отображение TF и данных с лидара, а в качестве Fixed Frame поставь `odom`.

Иии, поехали в столб! Как упрёшься, не останавливайся, смотри, как ведёт себя TF! Ему кажется, что колёса всё ещё крутятся, а значит TF `odom->base_footprint` продолжает смещать, но на деле ведь не так?

Вот результат в виде GIF:

В конце записи мы немного понажимали влево и вправо, но видишь, робот упёрся и не может откликнуться.



Укатились, думает робот, а на деле всё там же =)

Чему научились?

Главное, что надо запомнить, на колёсных роботах колёсная одометрия - это один из наиболее часто используемых источников информации. В большинстве случаев он полезен и хорошо работает, но надо понимать, что малые промахи и через час катаний робот, судя по колесной одометрии, уже будет не там, где реально находится. Это называется **дрифт** - он проявляется как смещения между истинным значением и показаниями измерений. Ему подвержены относительные источники измерения. Абсолютные (например, ориентир по маячкам или по карте) как правило не страдают от такого эффекта.

Таким образом, уровень фрейма **odom** - это относительные показания от разных видов одометрии (колесная, визуальная, инерциальная), которые позволяют локализовать робота, но могут иметь дрифт.

Задание

Не забывай закидывать наработки в Git!

Если ты провёл замену на **encoder** источник информации одометрии в модели, то можешь попробовать включить **turtlebot3_slam.launch** из своего пакета и посмотреть, что происходит, когда с учётом **gmapping** робот едет в стену? Попробуй поменять Fixed Frame на **odom**, потом обратно на **map**.

Вопросики

- Какие источники информации подвержены дрифту?
- Какие виды одометрии вы знаете/узнали?
- В каких ситуациях кроме езды в стену/столб одометрия может врать?

Gmapping, параметры, конфиги и тар

Содержание

- Содержание
- Подготовимся
- Gmapping
- Первые запуски и настройка
- Параметры в ROS
- Никуда без тар фрейма
- Карта - это матрица
- Цветастая карта
- Больше удобства - конфиги
- Чему научились?
- Задание

Ух, ну и большая сегодня будет тема! Нам надо разобраться, как запускать картографирование, настраивать и работать с ним! Поэтому, готовимся узнать много нового!

Подготовимся

У нас есть launch, который запускает симулятор и дополнительно запускает `robot_state_publisher` для публикации TF и rviz для отображения данных. В этой теме мы будем делать новый launch, который будет не только запускать симулятор, но и включать узел `gmapping`.

Давай задумаемся, как правильно организовать launch для запусков? С одной стороны, мы могли бы скопировать код из `turtlebot3_sim_start.launch` и вставить в новый launch, пусть он будет зваться `turtlebot3_sim_mapping.launch`. Также добавить всё необходимое для картографирования - запуск узла gmapping и новая конфигурация для rviz.

Так как мы теперь хотим посмотреть, как строится карта, а не посмотреть, как работает робот в симуляторе - это другая задача, а значит нам нужно создать новую конфигурацию rviz.

Но получится, что в случае обновления кода запуска симулятора нам тогда надо менять код в двух местах? Это не следует принципу DRY, по которому нужно стараться избегать дублирования, где это можно легко избежать.

Тогда мы можем сделать include launch `turtlebot3_sim_start.launch`, а внутри сделать запуск rviz управляемым (с помощью аргумента и атрибута `if` у `<node>` тэга), а в `turtlebot3_sim_start.launch` сделать запуск rviz со своей конфигурацией! Тогда при запуске `turtlebot3_sim_start.launch` будет запущен один rviz, а при запуске `turtlebot3_sim_mapping.launch` - другой (под другую задачу).

Если хотите попрактиковаться - сделайте самостоятельно оговоренное, а потом проверьте себя с описанным результатом =)

Значит, в `turtlebot3_sim_start.launch` добавляем следующие конструкции:

```

<launch>
  <arg name="rviz" default="false" />
  ...
  <node if="$(arg rviz)" pkg="rviz" type="rviz" name="rviz" args="-d $(find
super_robot_package)/rviz/sim_initial.rviz" />
</launch>

```

Проверь, что запуск `turtlebot3_sim_start.launch` и `rviz` в нём стал управляемым.

Gmapping

Отлично, теперь после проведенной подготовки самое время переходить к основному блюду - [gmapping](#)!

По сути `gmapping` - это узел, который занимается построением плоской карты препятствий. Есть и другие реализации данной задачи (`hector_mapping`, `cartographer` и др.), но сегодня мы будем пробовать работать именно с этим алгоритмом, так как он представляется проще всего в настройке и запуске.

Давай начнем знакомство сразу с практики! Создай `launch` с названием `turtlebot3_my_gmapping.launch` (название специально отражает, что мы сами разбираемся в нём) и начинкой в виде запуска узла `gmapping`. Должно получиться что-то такое:

```

<launch>
  <node pkg="gmapping" type="slam_gmapping" name="gmapping">
  </node>
</launch>

```

Если тебе непонятно, почему узел запускаем `slam_gmapping`, то ты скорее всего не читал доки по [gmapping](#) - посмотри их ещё раз.

И давай сразу для запуска симулятора с новым настраиваемым конфигом `rviz` сделаем два действия:

1. Скопируем `rviz/sim_initial.rviz` в `rviz/sim_mapping.rviz` - это будет наш конфиг отображения для задачи построения карты
2. Сделаем `launch` запуска всего этого добра - `turtlebot3_sim_mapping.launch`:

```

<launch>
  <arg name="rviz" default="false" />

  <include file="$(find
super_robot_package)/launch/turtlebot3_sim_start.launch" />
  <include file="$(find
super_robot_package)/launch/turtlebot3_my_gmapping.launch" />

  <node if="$(arg rviz)" pkg="rviz" type="rviz" name="rviz" args="-d $(find
super_robot_package)/rviz/sim_mapping.rviz" />
</launch>

```

```
super_robot_package)/rviz/sim_mapping.rviz" />
</launch>
```

Вот так мы сделали launch, который запускает всё это добро с правильной конфигурацией, а также выделили запуск gmapping в отдельный файл, что позволит далее редактировать только его, ведь мы планируем работать только с ним!

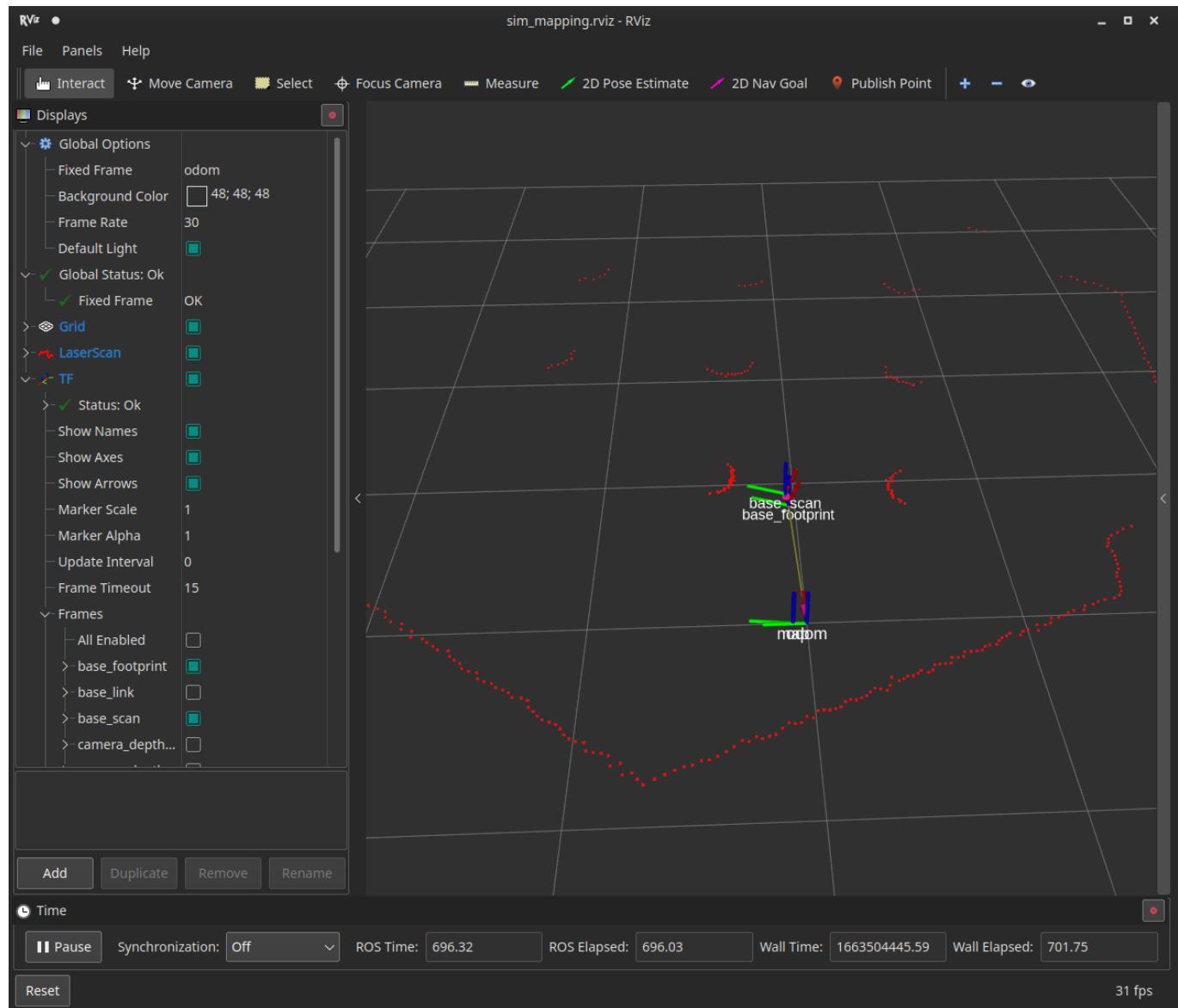
Ну что, поехали? Давай запустим нашего робота вместе с картографированием!

Первые запуски и настройка

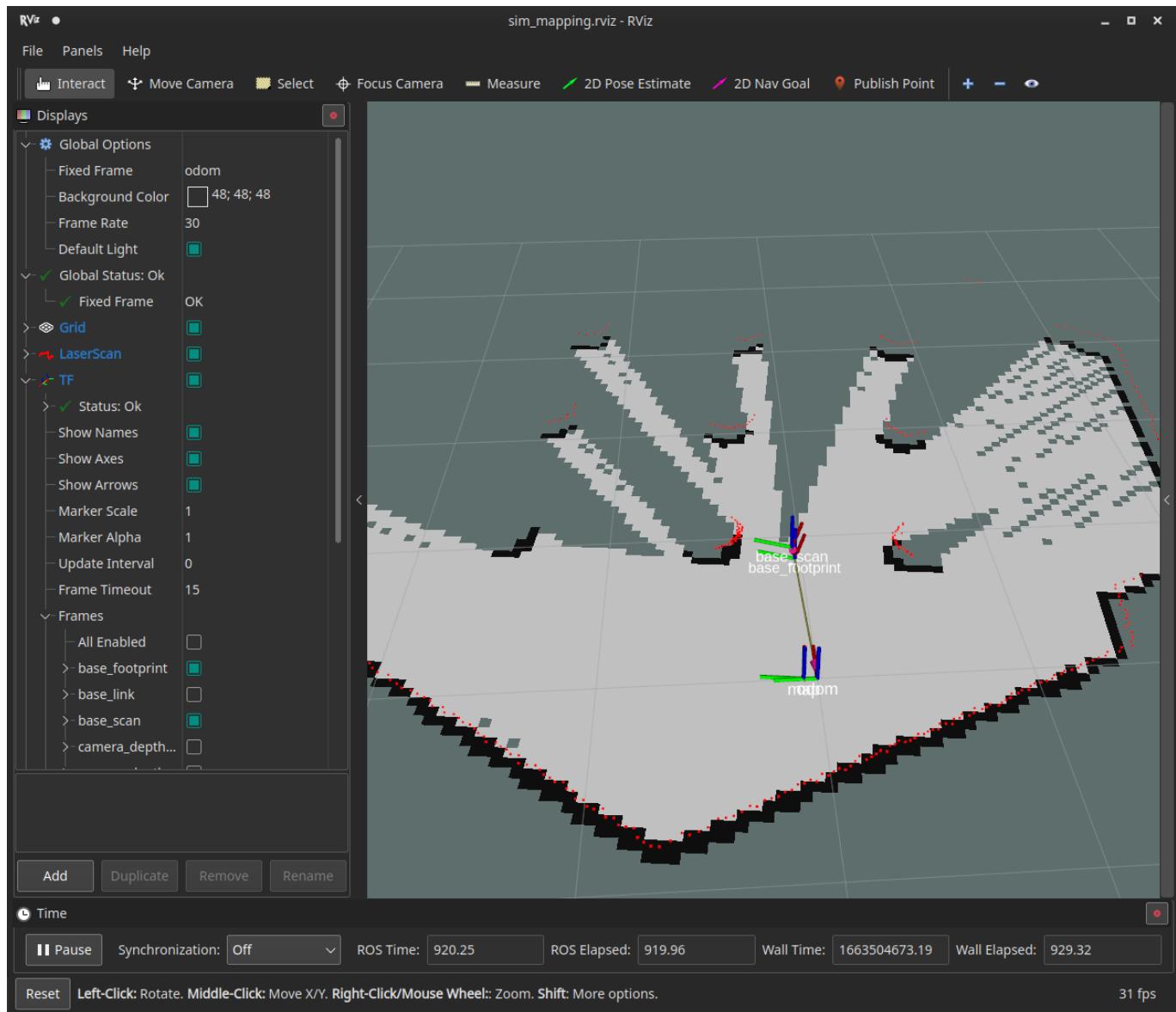
Если кто забыл =)

```
# Помнишь, что мы rviz по-умолчанию отключаем? Если тебе неудобно - можешь для
# себя включить запуск rviz по-умолчанию.
# Главное, чтобы тебе было удобно работать со своими инструментами!
roslaunch super_robot_package turtlebot3_sim_mapping.launch rviz:=true
```

Вот мы запустили и видим всё то же, что и видели при обычном запуске симулятора...



Естественно, Ватсон! Мы же не настроили Rviz! Главное, что в шапке виднеется `sim_mapping.rviz`, а значит конфиг правильный. Давай добавим информацию из топика `/map` и в качестве Fixed Frame поставим `map`.



Другое дело! Вот и наша построенная карта.

Ну что, готово, запустили, всё заработало - на этом закончим?

Не так быстро =) Если посмотришь [раздел параметров](#), то можно увидеть кучу параметров для настройки узла. Это одна из важных тем при работе с узлами, так как именно через параметры они настраиваются чаще всего и уметь работать с ними - это неотъемлемый инструмент при работе в ROS.

Параметры в ROS

Вообще, параметрам посвящен [раздел в ROS официальной документации](#). Мы из него возьмём только часть про узлы и их настройку.

! Главное, что надо запомнить касательно параметров - значения параметров хранятся, пока жив мастер. Если мастер перезапускается (`roscore` или первый запущенный `launch`), то все установленные параметры сбрасываются. Именно поэтому обычно руками параметры задаются только для пробы чего-либо, а в долгосрочной перспективе они должны быть записаны в файлах.

Мы не будем рассматривать, как задавать параметры через `rosrun`, так как в практике это редко используется, но ты можешь использовать утилиту `rosparam` для установки параметров вручную.

В launch файлах параметры узлам устанавливаются максимально просто!

Внутри тэга `<node>` надо прописать тэг `<param>`, а в нём атрибуты `name` и `value`, которые работают также, как и установка `<arg>` тэга при include другого launch. Вот [доки по тэгу param](#).

Так что давай сразу поменяем параметр, который отвечает за то, какой фрейм считать базовым, ведь по-умолчанию у параметра `base_frame` стоит `base_link`, а мы хотим работать с `base_footprint`, чтобы всё было в плоскости Z=0. Так что давай поставим правильное значение этому параметру!

В документации символ `~` означает, что параметр "приватный", то есть относится именно в узлу. Нас это не будет никак беспокоить, просто есть такой термин.

```
<node pkg="gmapping" type="slam_gmapping" name="gmapping">
    <param name="base_frame" value="base_footprint" />
</node>
```

Вот так просто! Не забудь сохранить Rviz, перезапусти и убедись, что всё ок!

Никуда без map фрейма

Если обратить внимание, то только с началом работы с gmapping у на появляется фрейм `map` из стандарта REP-105.

Это так, потому что только система картографирования может предоставлять информацию о перемещении с учетом коррекции по карте. Ведь мы помним, что `odom` хороший, но имеет дрейф, а карта - это источник абсолютных позиций робота, по которым можно этот дрейф корректировать!

Таким образом система картографирования предоставляет наиболее достоверную информацию, поэтому положение робота "на карте" смотрят именно относительно фрейма `map` по стандарту.

В стандарте может фигурировать ещё один более глобальный уровень `world`, но он чаще используется при использовании GPS и с двумя и более роботами.

Но как же это делает система картографирования? Тут алгоритм внутри сложный, но логика простая: gmapping подписывается на топик `scan` и, используя карту, которую сам же и строит, а также приходящие сканы, производит коррекцию так, чтобы сканы ложились на точки препятствия на карте (чёрные точки).

💪 С помощью параметра `map_frame` поменяй название фрейма, который будет отвечать за карту на `my_map`. Перезапусти и через `rqt_tf_tree` и `rviz` убедись, что изменения применились. Верни после эксперимента обратно на `map`, чтобы соответствовать стандарту =)

Карта - это матрица

Если попробовать вывести информацию с топика через `rostopic echo /map`, то можно получить кучу нечитаемого вида. Именно поэтому важно уметь представлять информацию различными способами, так как что-то можно посмотреть в числах, а что-то воспринимается визуально (или в другой интерпретации). Мы ведь не смотрим картинки попиксельно?

Так и здесь, карта - это на самом деле прямоугольная матрица из чисел. Если ты знаешь Python, MATLAB или другой язык, касающийся математики (или саму математику), то ты уже знаешь, что матрица - это двумерный массив с некоторым количеством строк и столбцов (высота и ширина карты).

Чтобы получить эти значения касательно карты, можно вывести информацию из топика [/map_metadata](#):

```
resolution: 0.05000000074505806
width: 4000
height: 4000
```

Ширина и высота заданы в точках (элементах матрицы), а `resolution` показывает разрешение - сколько метров покрывает каждый элемент матрицы [метров на блок].

Можно считать, что вся карта - это картинка, каждый элемент матрицы - это пиксель, а физический размер пикселя - это разрешение.

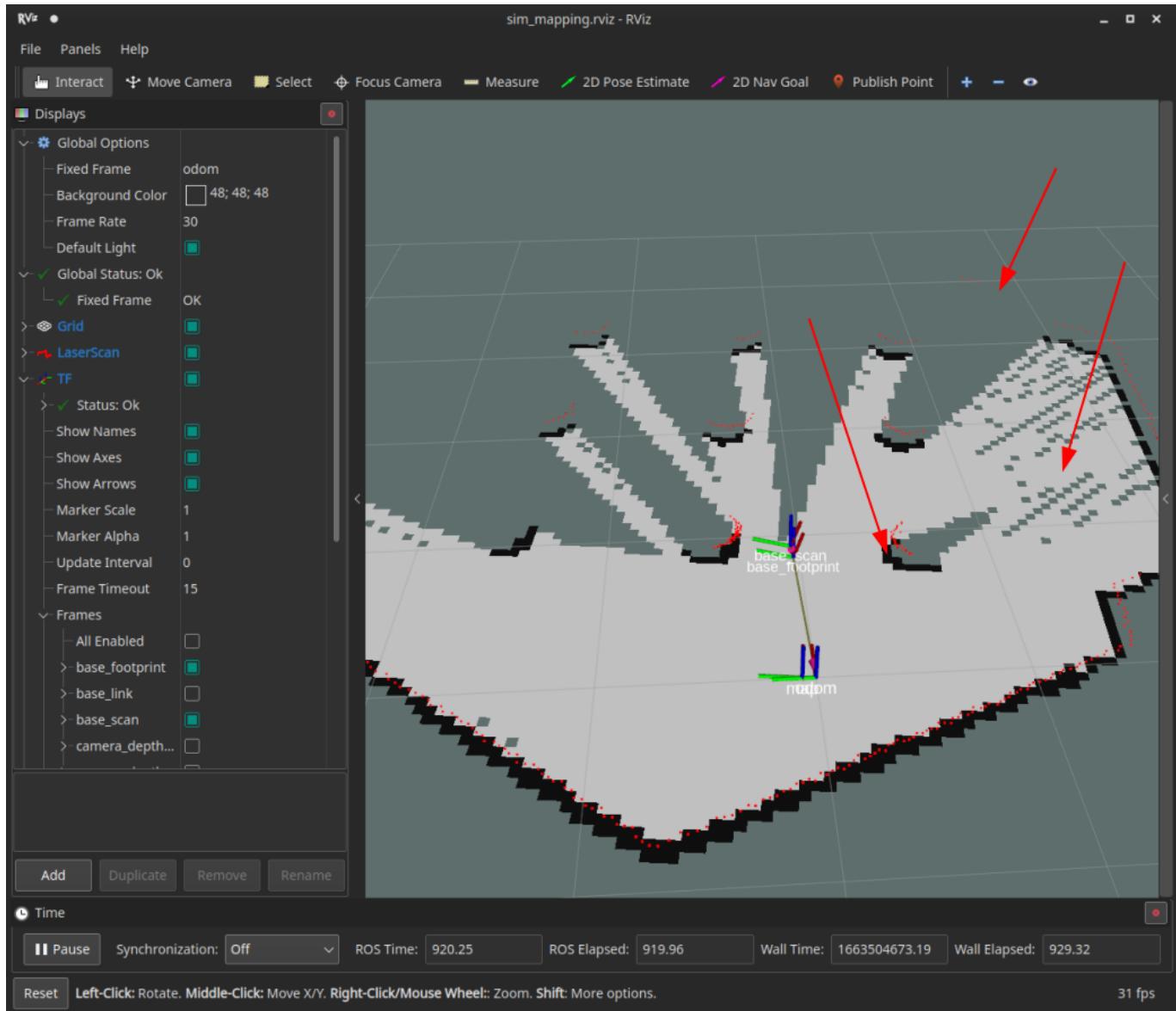
Это важно понимать, так как, например, ширину и высоту можно настраивать под свои нужды, чтобы потреблять меньше оперативной памяти, а разрешение часто влияет на скорость обработки и загрузку процессора.

 Поменяй параметр `delta` на значения (0.1, 0.01, 0.3, 0.001) и посмотри, как влияет на скорость обработки/детализацию карты.

Цветастая карта

Отлично, с форматом передачи разобрались, но что значат черные/белые/серые клетки на карте в rviz?

Если посмотреть в [документацию типа топика /map](#), то можно увидеть, что в `data` поле хранятся значения "-1" и от 0 до 100. Вот и на карте у нас они отображаются:



В отображении карты от gmapping используется три основных цвета:

- черный (значение 100) - точки в матрице, на которых находится препятствие
- белый (значение 0) - точки, где точно нет препятствия
- серый (значений -1) - неизвестно, так как туда не попадал лазер.

Тут можно сразу задать вопрос, а почему между столбами и в некоторых зонах белого появляются серые точки? В первом случае лидар не достаёт до стены на другой стороне карты, поэтому нельзя точно сказать, есть ли там препятствие. Алгоритм gmapping работает по принципу "скан достал до препятствия, значит на пути от робота до этой точки нет препятствий".

! Обрати внимание, что в системе карт ROS есть понятие "неизвестная" местность и она отличается от состояния "свободно". Это будет важно понимать при дальнейшей работе.

Больше удобства - конфиги

В ходе изучения работы gmapping мы пробовали устанавливать разные параметры, но представь, что тебе надо поменять 10, 20 или больше параметров. Это приводит launch к разрастанию (на каждый параметр своя строка).

А что если в launch ещё и не один узел надо так настраивать?

В таком случае было бы удобно вынести конфигурацию параметров в отдельное место. Этим местом будет "конфигурационный файл". А если проще - конфиг.

Можно сделать файл в формате YAML и удобно в нём настраивать под себя узел, а launch будет заниматься своей работой - запускать узлы!

Давай сделаем свой первый конфиг в пакете для узла `gmapping`. Создаём папку `config` внутри пакета. Внутри этой папки создаём файл `gmapping_params.yaml`.

Ну и переносим в него все наши параметры из тэгов `<param>`. Но как перенести, не просто же скопировать строки?

Правильно! YAML работает по принципу "ключ: значение", где ключ - это имя параметра, а значение - это значение =)

Должно получиться что-то наподобие:

```
base_frame: base_footprint
map_frame: map
delta: 0.05
```

Окей, в файл перенесли, но как нам это всё передать в узел? В этом нам поможет тэг `<rosparam>`. Мы просто использует атрибуты `command` и `file`. Первый указывает, что надо сделать и из всех представленных в доках нам нужна команда "load". А в `file` как обычно пишем путь до файла конфига (там даже примерчик с `$(file pkg)` есть =)

Получается, запуск узла будет выглядеть так:

```
<node pkg="gmapping" type="slam_gmapping" name="gmapping">
  <rosparam command="load" file="$(find
super_robot_package)/config/gmapping_params.yaml" />
</node>
```

Так мы перенесли конфигурацию узла из launch в отдельный конфиг, что разделяет по файлам задачи и делает работу с пакетом удобнее!

Чему научились?

Вот так незатейливо мы смогли взять узел для картографирования и подключить самостоятельно к нашему роботу. Круто! Молодцы!

Как уже ранее сообщалось, в ROS действительно уже сделано много готового и удобного, поэтому нам не нужно писать самостоятельно софт, но уметь правильно использовать и настраивать - это не менее важный и полезный навык. Главное, это понимать, для чего всё это делается и держать в голове фактор удобства использования!

Задание

- С помощью `rqt_graph` или утилит `rosnode`, `rostopic` определи, на что подписывается узел `/gmapping`. Проверь, соответствует ли это документации.
- Определи, на что влияют параметры `linearUpdate` и `angularUpdate`. Настрой их в конфиге под себя и посмотри, как это влияет на коррекцию TF `map->odom`.
- Посмотри конфигурацию из пакета `turtlebot3_slam` для gmapping и попробуй взять оттуда разные настройки. Стало ли лучше/быстрее/точнее работать (оценка на глаз)?
- * Попробуй самостоятельно сделать launch-файлы и запустить `hector_slam` в качестве картографической системы. Не забудь установить пакеты с помощью `apt`.

Навигация роботов

- [Что такое навигация?](#)
- [Из чего состоит навигация?](#)
- [Состав move_base](#)
- [Карты стоимостей](#)
- [Планнер](#)
- [Переходим к практике](#)
- [Чему научились?](#)
- [Задание](#)
- [Вопросики](#)
- [Ресурсы](#)

Что такое навигация?

Во-первых, следует определить, что здесь будем понимать под навигацией. Навигация - это область исследования, которая фокусируется на процессе мониторинга и управления перемещением корабля или транспортного средства из одного места в другое (с)Wiki. Из такого определения можно сразу же сделать два вывода. Навигация занимается лишь вопросом перемещения из одной точки в другую, включая поиск пути. При этом она игнорирует вопросы, связанные с определением положения и картографирования. Из этого следует, что для того, чтобы запустить навигацию, первоначально необходимо поднять SLAM и затем предоставить карту и текущее положение в навигационный стек.

Из чего состоит навигация?

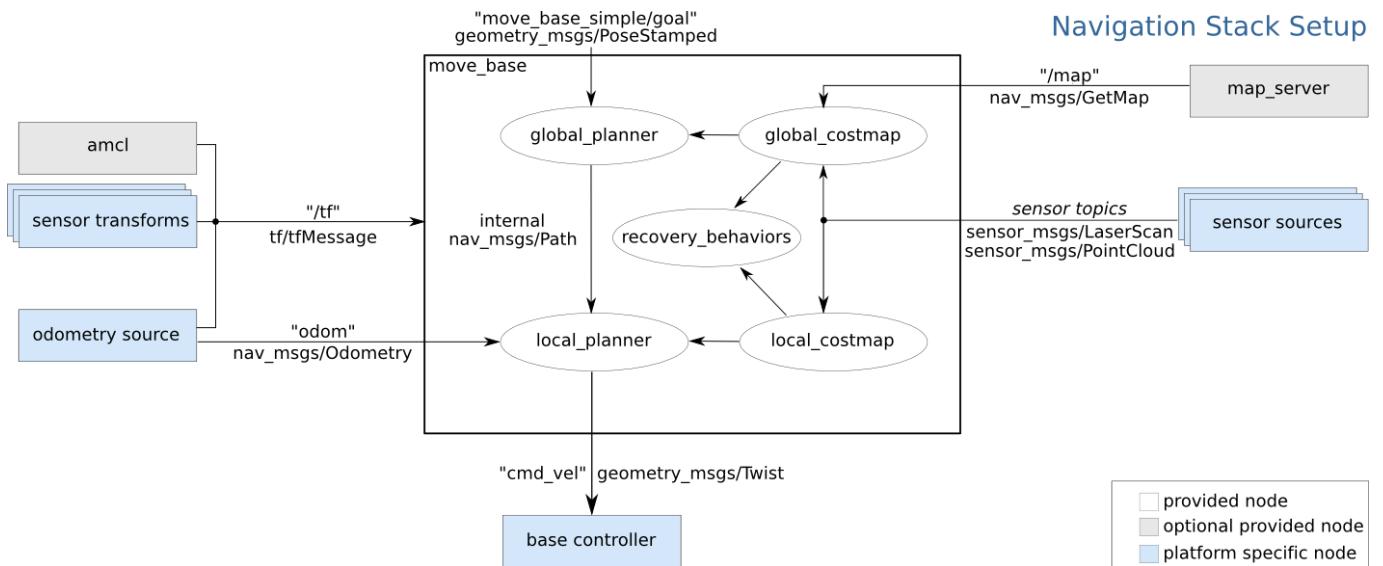
Основной составляющей навигации являются алгоритмы поиска пути. Они отвечают за то, чтобы навигационный стек непосредственно выполнял свою прямую функцию. В уже устоявшемся представлении поиск пути осуществляется в два этапа: глобальное планирование и локальное планирование.

Глобальный планировщик отвечает за генерацию плана движения от стартовой точки до конечной. И теоретически может вообще не обновляться в процессе движения, либо обновляться, но очень редко. Перед локальным планировщиком же стоит цель безопасно провести робота через небольшие локальные участки местности, избегая периодически возникающие препятствия. То есть по сути в определенный момент времени выделяется некоторая область на карте, внутри которой локальный планировщик строит путь от текущего положения робота к точке глобального плана на границе локальной карты. И этот процесс повторяется довольно часто (обычно 5-10 Гц).

Состав move_base

Стек навигации включает в себя множество пакетов для решения целей навигации: планировщики, работа с картами, локализация. Каждый узел существует независимо и может быть написан вручную или найден на просторах GitHub. Для объединения всего этого добра в ROS существует узел [move_base](#). На картинке ниже показана схема работы узла move_base и его взаимодействия с другими компонентами. Синие узлы различаются в зависимости от платформы робота, серые являются необязательными, а белые узлы являются обязательными. На первый взгляд схема может показаться

запутанной, но по мере изучения отдельных узлов навигации рекомендуется к ней возвращаться, и рано или поздно всё встанет на свои места =)



Сам узел `move_base` находится в черном квадрате посередине и включает две большие задачи: построение карты стоимостей и планирование маршрутов.

Во многих пакетах ROS встречается разделение на `global` и `local`. `Local` обычно привязывается к системе координат движущегося робота, а `global` к неподвижной системе координат карты.

Начнем разбираться по порядку.

Карты стоимостей

Для того, чтобы начать какое-либо движение, необходимо понимать вид окружающей среды: где можно ездить, а где нельзя. `move_base` преобразует данные бортовых датчиков (камеры, лидары и т.п.) и предзаписанных карт в формат `costmap`. Основной формат внутреннего представления карт в ROS носит название `OccupancyGrid`, где пространство вокруг робота разбивается на клетки заданного размера. У каждой клетки есть значение "occ", которое отражает вероятность занятости клетки в процентах. В простейшем случае есть 3 вида клеток:

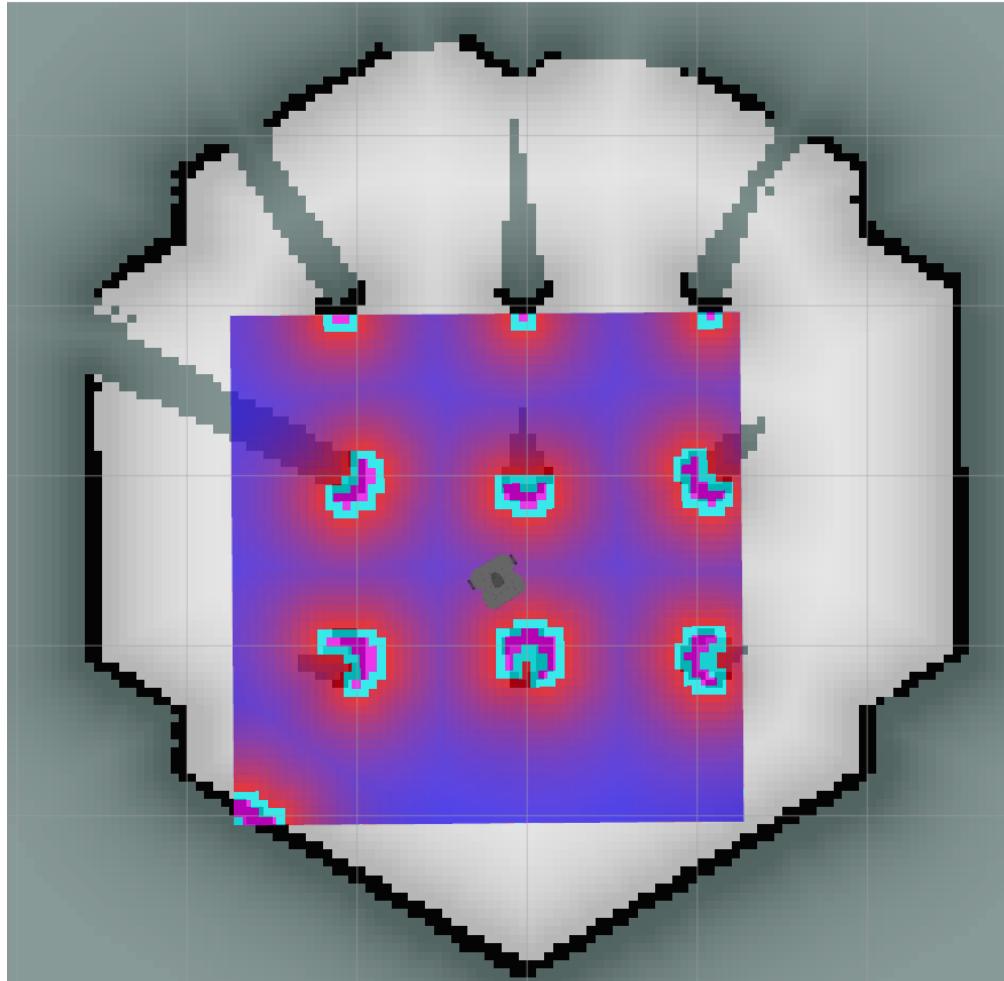
- свободная область: `occ = 0`
- занятая область: `occ = 100`
- неизвестная область: `occ = -1`

Обычно используется следующий подход: в `global costmap` отправляется карта всей местности (план этажа, квартиры, завода), а в `local costmap` отправляются данные лазерных лидаров и камер, которые помогут учитывать препятствия, возникающие перед роботом.

Для того, чтобы увидеть отличие на примере - запустим нашего тартлбота вместе со стеком навигации:

```
roslaunch super_robot_package turtlebot3_sim_start.launch
TURTLEBOT3_MODEL=waffle roslaunch turtlebot3_slam turtlebot3_slam.launch
TURTLEBOT3_MODEL=waffle roslaunch turtlebot3_navigation move_base.launch
```

Переходим в меню выбора отображения (`Add->By topic`), выбираем путь `move_base/global_costmap/costmap` и там выбираем Мар. Перед тем, как окончательно выбрать, внизу в Display Name наберите "Global Map". Аналогично добавляем `local_costmap`. Должно получиться что-то вроде рисунка ниже, где цветным показана локальная карта стоимостей, а в оттенках серого - глобальная.



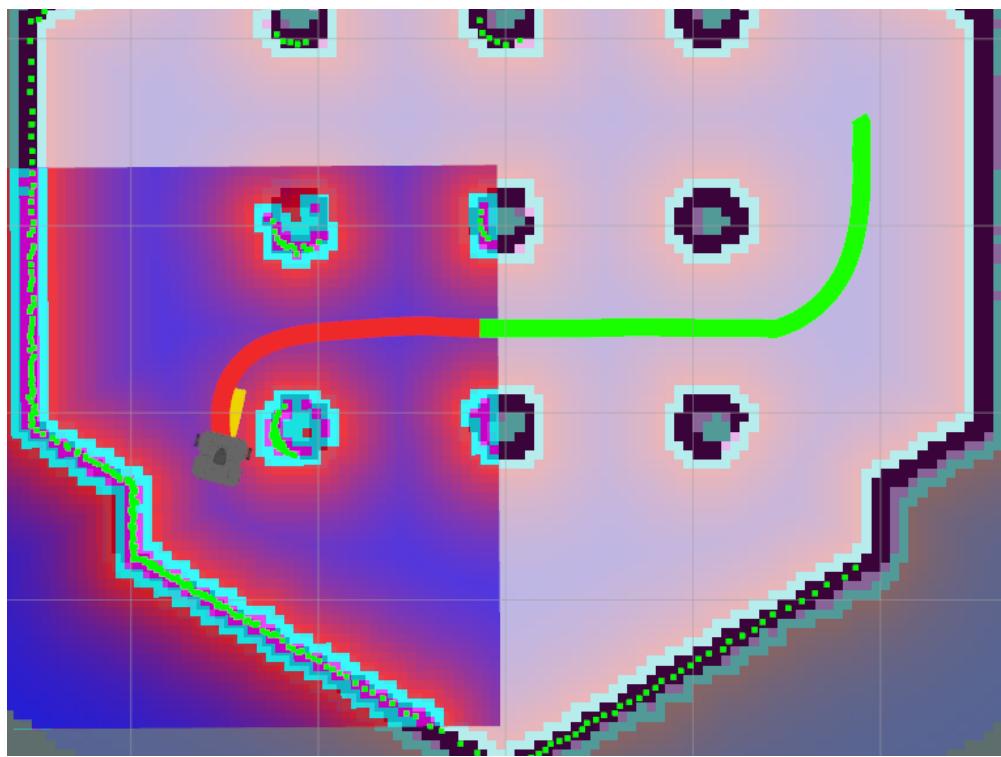
Для смены режима отображения карты, в меню Displays нажимаем на локальную карту и в выпадающем меню в разделе Color Scheme выбираем "costmap"

Планнер

После создания карты, переходим к планированию движения. Если для карт стоимостей подход создания локальной и глобальной карты идентичен, то у планировщиков всё немного сложнее. Глобальный и локальный планнеры выполняют разные задачи и реализуются разными пакетами. Алгоритм следующий:

1. Global Planner, используя global costmap, строит кратчайший путь к цели.
2. Local Planner следует глобальному пути, но учитывает возникающие препятствия и габариты робота.

Работу планнеров так же можно увидеть в Rviz, для этого переходим в меню выбора отображения (`Add->By topic`), выбираем путь `move_base/NavfnROS/plan` и там выбираем Path (зеленая линия). Внизу в Display Name наберите "Global Path". Аналогично добавьте топики локального планнера, которые находятся в топиках `/move_base/DWAPlannerROS/global_plan` и `/move_base/DWAPlannerROS/local_plan`. В результате должно получиться 3 траектории, вроде такого:



💡 Попробуйте несколько раз задать цель в Rviz через стрелку 2D Nav Goal и посмотреть на работу локального и глобального планинера. За что отвечает каждая траектория? На сколько далеко строятся маршруты? По какому принципу строится локальная и глобальная траектория?

Переходим к практике

Сейчас давайте уже подробнее разберемся, как же заставить робота самостоятельно ехать по заданной траектории, а именно посмотрим на пакет `move_base`, располагающийся в стеке навигации, который мы с вами рассмотрели чуть раньше.

Для того, чтобы наш робот поехал, а именно заработала автономная навигация - нужно подготовить дополнительные файлы. Давайте скопируем файлы из `turtlebot3_navigation/param` к себе в папочку `config`.

Согласно умной [страничке в ROS wiki](#) для навигации нашего робота нужно скопировать к себе следующие файлы:

```
base_local_planner_params.yaml          # The parameter of the speed command  
to the robot  
costmap_common_params_burger.yaml       # The parameter of costmap  
configuration consists  
costmap_common_params_waffle.yaml       # The parameter of costmap  
configuration consists  
costmap_common_params_waffle_pi.yaml    # The parameter of costmap  
configuration consists  
dwa_local_planner_params.yaml          # The parameter of the speed command  
to the robot  
global_costmap_params.yaml             # The parameter of the global area  
motion planning  
local_costmap_params.yaml              # The parameter of the local area  
motion planning
```

```
move_base_params.yaml          # parameter setting file of move_base
that supervises the motion planning.
```

Сделаем это через `roscp`: Для нашего случая будет такой ряд команд, если вы ранее в качестве робота выбрали `waffle`:

```
# Путь к нашей папке config'ов
COPY_TARGET=`rospack find super_robot_package`/config
# Копируем файлы
roscp turtlebot3_navigation costmap_common_params_waffle.yaml $COPY_TARGET
roscp turtlebot3_navigation local_costmap_params.yaml $COPY_TARGET
roscp turtlebot3_navigation global_costmap_params.yaml $COPY_TARGET
roscp turtlebot3_navigation move_base_params.yaml $COPY_TARGET
roscp turtlebot3_navigation dwa_local_planner_params_waffle.yaml $COPY_TARGET
```

Осталось совсем немного... скачаем локальный планировщик [dwa_local_planner](#)

```
sudo apt install ros-noetic-dwa-local-planner
```

Так, теперь нам осталось создать launch-файл для запуска всего этого добра. Давайте посмотрим, что у нас уже имеется и возьмем к себе на вооружение. Итак, нам нужно посмотреть `turtlebot3_navigation/move_base.launch`. Если пакет `move_base` еще не установлен, то вы знаете, что делать 😊

```
rosedit turtlebot3_navigation move_base.launch
```

Ну-с, тут вроде бы все достаточно понятно... вначале у нас идут аргументы, потом подгружаются конфиги, которые мы с вами уже скопировали к себе.

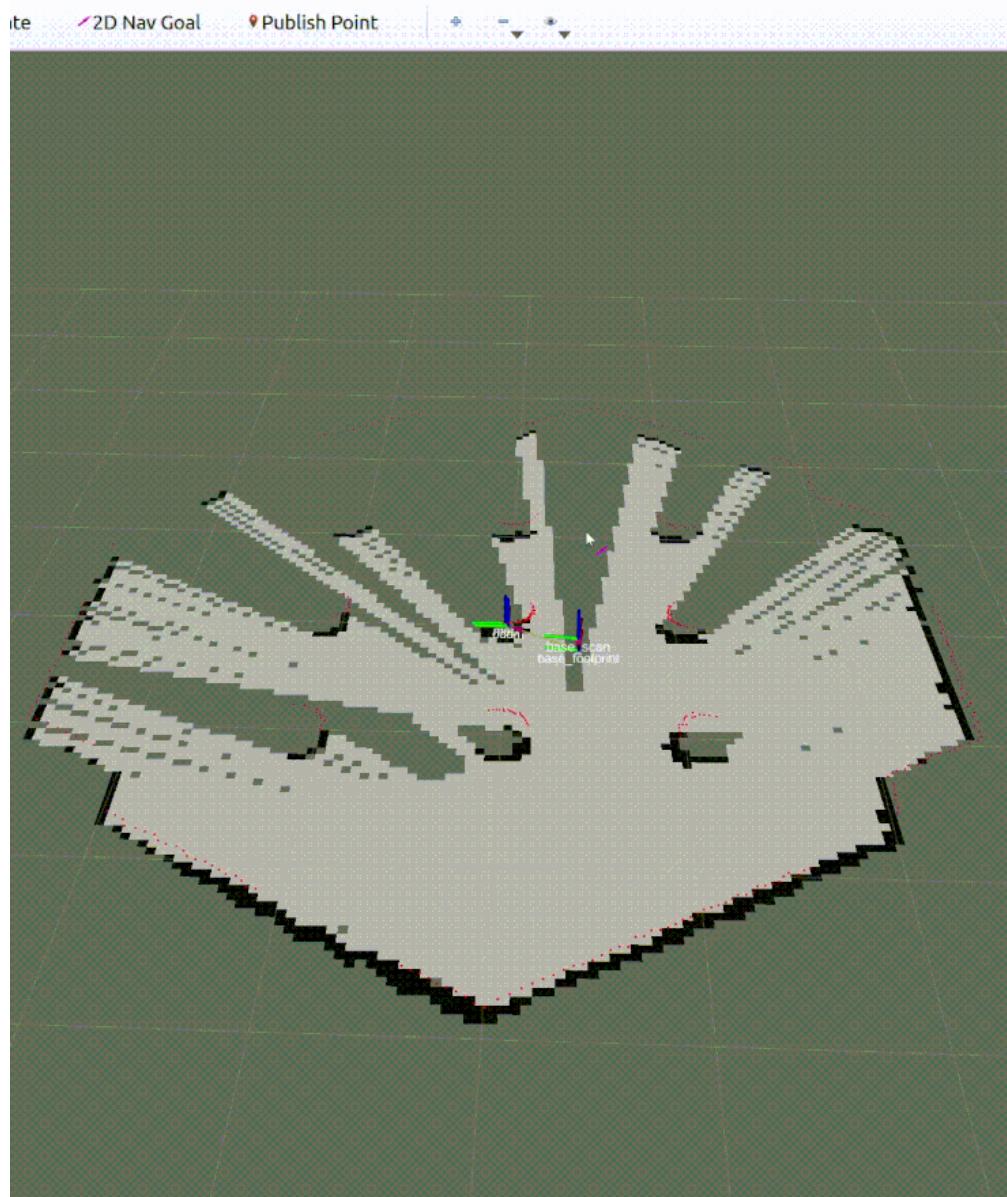
```
<launch>
  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger,
waffle, waffle_pi]"/>
  <arg name="cmd_vel_topic" default="/cmd_vel" />
  <arg name="odom_topic" default="odom" />
  <arg name="move_forward_only" default="false"/>

  <!-- move_base -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS" />
    <rosparam file="$(find super_robot_package)/config/costmap_common_params_$(<arg
model>.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find super_robot_package)/config/costmap_common_params_$(<arg
```

```
model).yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find super_robot_package)/config/local_costmap_params.yaml"
command="load" />
    <rosparam file="$(find super_robot_package)/config/global_costmap_params.yaml"
command="load" />
    <rosparam file="$(find super_robot_package)/config/move_base_params.yaml"
command="load" />
    <rosparam file="$(find
super_robot_package)/config/dwa_local_planner_params_${arg model}.yaml"
command="load" />
    <remap from="cmd_vel" to="$(arg cmd_vel_topic)"/>
    <remap from="odom" to="$(arg odom_topic)"/>
    <param name="DWAPlannerROS/min_vel_x" value="0.0" if="$(arg
move_forward_only)" />
</node>
</launch>
```

На основании этого создадим себе `turtlebot3_move_base.launch`. В принципе, чтобы наш с вами робот поехал автономно осталось уже совсем немного. Осталось написать launch-файл для запуска симулятора, навигации и построения карты. Наверно, я вас сейчас обрадую:). У нас уже с вами все есть.

💡 Попробуй написать launch-файл `turtlebot3_gazebo_slam.launch`, в котором будет запуск `turtlebot3_sim_start.launch`, `turtlebot3_my_gmapping.launch`, `turtlebot3_move_base.launch` и не забудь добавить запуск `rviz`



Чему научились?

В этот раз мы с вами неплохо потрудились - начали разбираться в одном из основных моментов автономных мобильных платформ, а именно в стеке навигации. Разобрались, как он устроен и что за какую часть отвечает. Научились, как управлять роботом с помощью задания целей, и написали launch файл для автономного движения нашего робота и построения карты.

Задание

Основное задание для вас - это внимательно пройти этот топик, выполнив каждое задание по ходу.

Вопросики

1. За что отвечает глобальный и локальный планировщик? В чем их отличия?
2. По какому принципу строится локальная и глобальная траектория?
3. Какие задачи должен выполнять move_base?

Ресурсы

Более детально с [стеком навигации](#) вы можете познакомиться на страничке ROS Wiki

Настройка MoveBase

- DWA Local Planner
- Global Planner
- Costmap Common Params
 - Local and Global Costmap
- Основные параметры. Move Base params
- Чему научились?
- Задание
- Вопросики
- Ресурсы

Навигационный стек имеет множество параметров, которые необходимо настраивать в зависимости от конфигурации робота. Как думаете, подойдут ли параметры двухкилограммовой черепашки к двухтонному грузовику с кинематикой Аккермана? Очень хотелось бы, но, к сожалению, к каждому объекту требуется свой подход и настройка. Давайте рассмотрим на примере черепашки, как изменение параметров влияет на характер её движения.

DWA Local Planner

В прошлом топике мы уже начали знакомиться с локальным планировщиком [DWA local planner](#), однако все параметры в нем были выставлены по default. Для лучшего понимания смысла параметров, рекомендуем немного изучить принцип работы данного планировщика на странице ROS Wiki, либо ещё где-то. Давайте посмотрим, что в нем вообще имеется, откроем наш [dwa_local_planner_params_waffle.yaml](#). Из названия уже понятно, что там должны быть какие-то параметры 🐢

DWAPlannerROS:

```
# Robot Configuration Parameters
max_vel_x: 0.26
min_vel_x: -0.26

max_vel_y: 0.0
min_vel_y: 0.0

# The velocity when robot is moving in a straight line
max_vel_trans: 0.26
min_vel_trans: 0.13

max_vel_theta: 1.82
min_vel_theta: 0.9

acc_lim_x: 2.5
acc_lim_y: 0.0
acc_lim_theta: 3.2
```

```

# Goal Tolerance Parameters
xy_goal_tolerance: 0.05
yaw_goal_tolerance: 0.17
latch_xy_goal_tolerance: false

# Forward Simulation Parameters
sim_time: 2.0
vx_samples: 20
vy_samples: 0
vth_samples: 40
controller_frequency: 10.0

# Trajectory Scoring Parameters
path_distance_bias: 32.0
goal_distance_bias: 20.0
occdist_scale: 0.02
forward_point_distance: 0.325
stop_time_buffer: 0.2
scaling_speed: 0.25
max_scaling_factor: 0.2

# Oscillation Prevention Parameters
oscillation_reset_dist: 0.05

# Debugging
publish_traj_pc : true
publish_cost_grid_pc: true

```

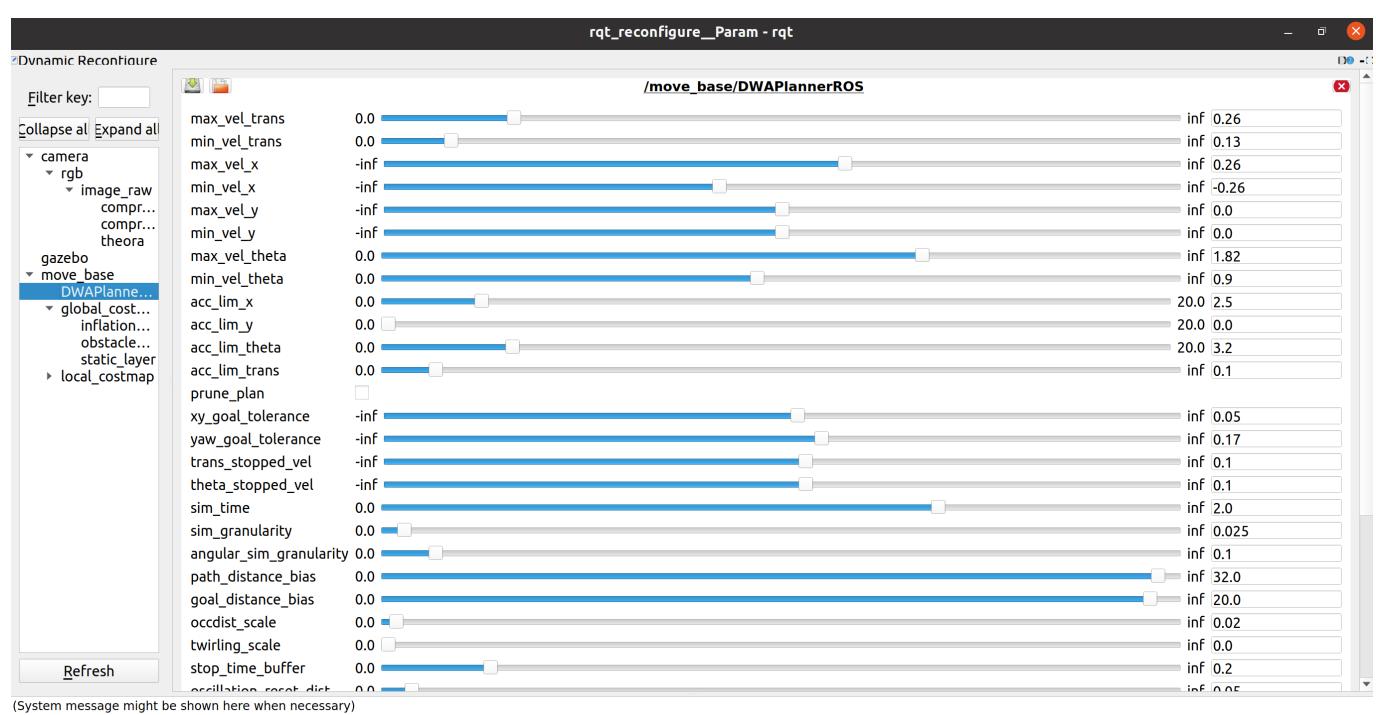
Так, ну мы вас не обманули и вы, действительно, можете увидеть тут кучу каких-то параметров, которые пока что особо не понятно, за что отвечают. Настало время разобраться с этим. Все параметры сгруппированы в несколько категорий, а именно:

- **Robot Configuration Parameters** - параметры конфигурации робота
- **The velocity when robot is moving in a straight line** - параметры при движении по прямой линии
- **Goal Tolerance Parameters** - параметры допустимого отклонения от цели
- **Forward Simulation Parameters** - параметры моделирования. (При планировании маршрута к цели, алгоритм проводит моделирование движения из текущего местоположения, для предсказания осуществимости траектории и выбора наилучшей)
- **Trajectory Scoring Parameters** - параметры оценки траектории
- **Oscillation Prevention Parameters** - параметры предотвращения колебаний
- **Debugging**- отладка

Запускаем наш созданный в прошлом топике launch `turtlebot3_gazebo_slam.launch`

Для того, чтобы проще было настраивать параметры конфигурации в запущенной системе (и не перезапускать по 20 раз) воспользуемся `rqt_reconfigure`

```
rosrun rqt_reconfigure rqt_reconfigure
```

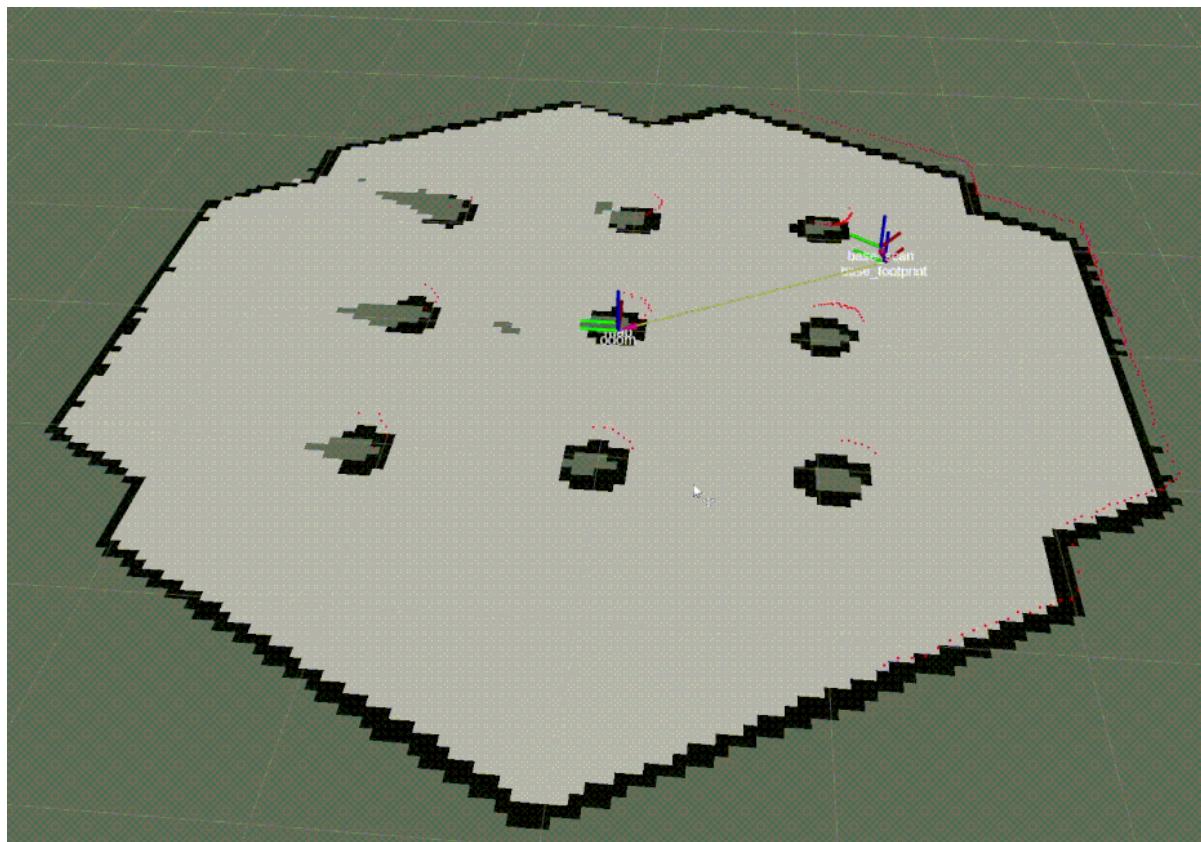


Например, давайте посмотрим на параметр `max_vel_theta`. Для `waffle` этот параметр задан, как `max_vel_theta: 1.82`

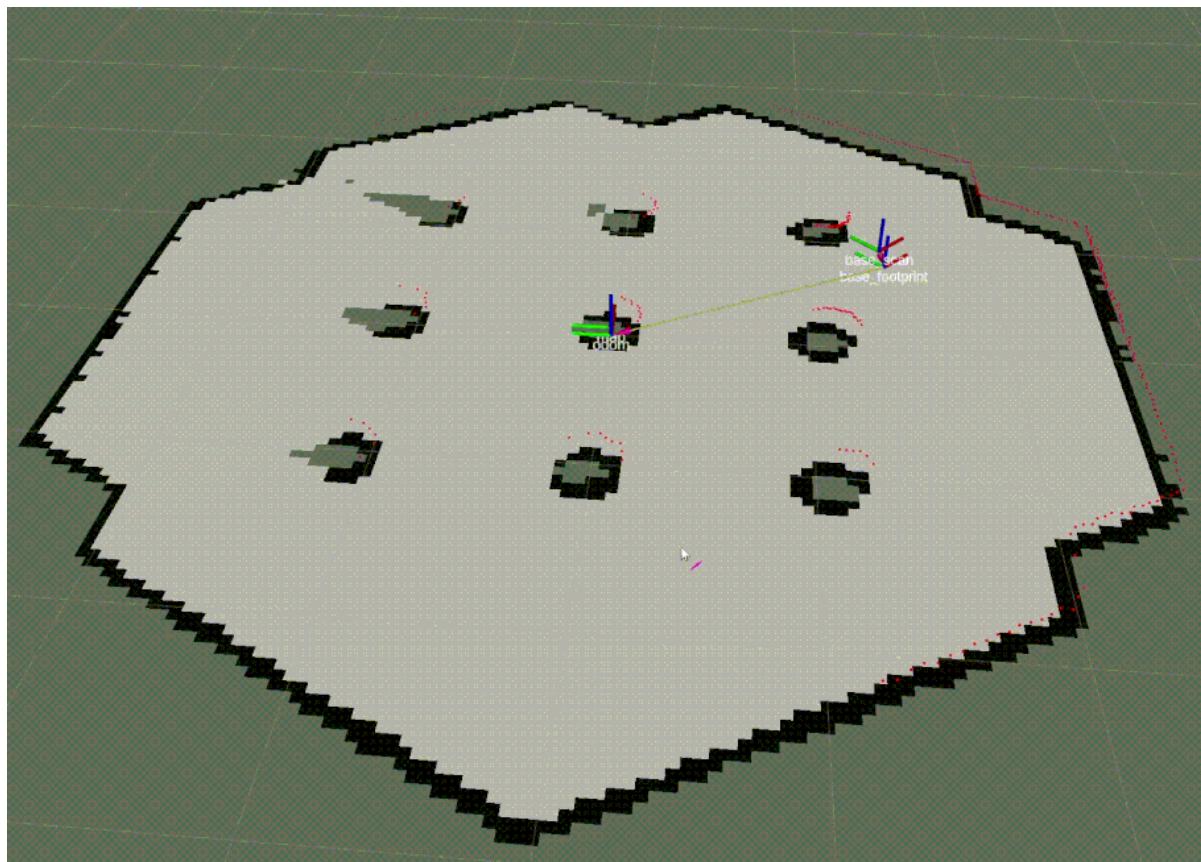


Окей, а что будет, если этот параметр понизить до 0.5? А потом до 0? Пробуем:

```
max_vel_theta = 0.5
```



```
max_vel_theta = 0.0
```



💪 Время сделать вывод... каким образом изменение параметра `max_vel_theta` сказывается на планировании маршрута робота?



Теперь попробуйте самостоятельно поменять параметры `sim_time` и `occdist_scale`.

Сделайте вывод о том, как уменьшение/увеличение этих значений влияет на планирование маршрута робота.

Получилось? Здорово! Тогда двигаемся дальше

Global Planner

Про глобальный планнер было сказано много слов, но пока что нигде в файлах не встречались упоминания о нем (особо). Дело в том, что `move_base` и в частности `turtlebot` в качестве глобального планировщика по умолчанию использует `navfn`. Он довольно простенький и особо не имеет параметров. Давайте попробуем изменить стандартный глобальный планнер на `global_planner` путем добавления в наш `move_base.launch` следующего параметра:

```
<param name="base_global_planner" value="global_planner/GlobalPlanner" />
```

Если когда-нибудь подумаете, что у вас плохо с фантазией, вспомните, что разработчик дал своему глобальному планнеру имя "GlobalPlanner" 😊

Теперь при запуске будет использоваться данный планнер, со стандартными параметрами. Смену плannера можно увидеть по новым названиям топиков для глобального пути:

`/move_base/GlobalPlanner/plan` Список параметров планировщика можно глянуть на ROS wiki, но как и с локальным параметрами можно выгружать из конфига. Для этого в своей папке с конфигами создаем файл `global_planner_params.yaml` со следующим содержанием:

```
GlobalPlanner:                                     # Also see:
  http://wiki.ros.org/global_planner
    old_navfn_behavior: false                      # Exactly mirror behavior of
    navfn, use defaults for other boolean parameters, default false
    use_quadratic: true                           # Use the quadratic approximation
    of the potential. Otherwise, use a simpler calculation, default true
    use_dijkstra: true                            # Use dijkstra's algorithm.
    Otherwise, A*, default true
    use_grid_path: false                          # Create a path that follows the
    grid boundaries. Otherwise, use a gradient descent method, default false

    allow_unknown: true                           # Allow planner to plan through
    unknown space, default true
                                                #Needs to have
    track_unknown_space: true in the obstacle / voxel layer (in costmap_commons_param)
    to work
    planner_window_x: 0.0                         # default 0.0
    planner_window_y: 0.0                         # default 0.0
    default_tolerance: 0.0                        # If goal in obstacle, plan to the
    closest point in radius default_tolerance, default 0.0

    publish_scale: 100                           # Scale by which the published
    potential gets multiplied, default 100
```

```

planner_costmap_publish_frequency: 0.0          # default 0.0

lethal_cost: 253                            # default 253
neutral_cost: 50                           # default 50
cost_factor: 3.0                          # Factor to multiply each cost
from costmap by, default 3.0
publish_potential: true           # Publish Potential Costmap (this
is not like the navfn pointcloud2 potential), default true

```

Далее добавляем строчку в наш move_base.launch

```
<rosparam file="$(find super_robot_package)/config/global_planner_params.yaml"
command="load" />
```

 Попробуйте изменить параметры use_dijkstra, use_grid_path и понять как меняется вид глобальной траектории. В каких случаях следует выставлять параметр allow_unknown в значение false?

Costmap Common Params

В ROS, как правило, в стеке навигации существуют две карты, хранящие информацию о препятствиях в окружающей среде. Это local and global costmap. Из названия понятно, что одна из них глобальная - та, которая строит протяженный маршрут до конечной цели, а локальная - способна перестраивать маршруты непосредственно в те моменты, когда на свободном пространстве по какой-то причине появилось препятствие, которое следует объехать.

В `costmap_common_params` устанавливается, как правило, что будет касаться, как и локального, так и глобального планировщика. Тут применяется карта стоимости, предоставляющая сведения о препятствиях в мире. Для того, чтобы обеспечить корректную связь между ними - нужно связать карты затрат с теми устройствами, которые будут предоставлять им информацию об окружающей обстановке. В общем виде конфигурация параметров выглядит так 

```

obstacle_range: 2.5
raytrace_range: 3.0
footprint: [[x0, y0], [x1, y1], ... [xn, yn]]
#robot_radius: ir_of_robot
inflation_radius: 0.55

observation_sources: laser_scan_sensor point_cloud_sensor

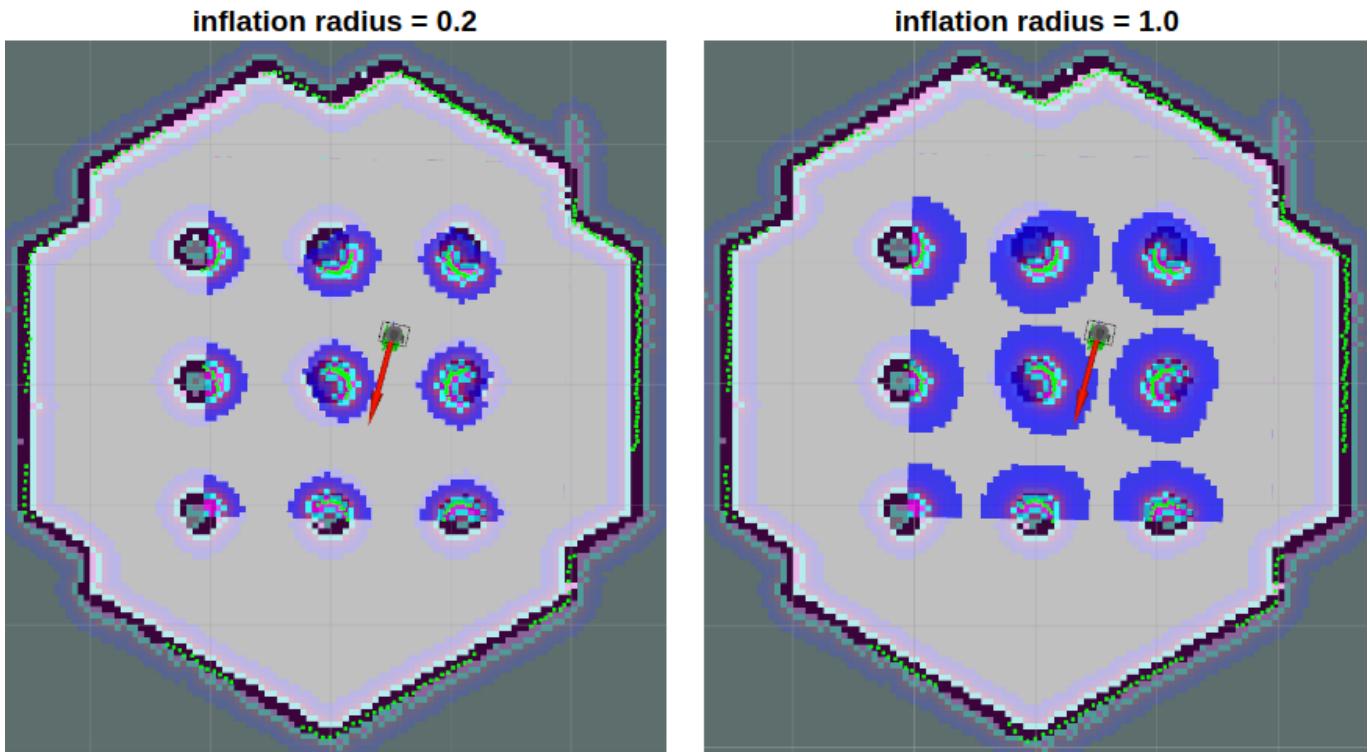
laser_scan_sensor: {sensor_frame: frame_name, data_type: LaserScan, topic:
topic_name, marking: true, clearing: true}

point_cloud_sensor: {sensor_frame: frame_name, data_type: PointCloud, topic:
topic_name, marking: true, clearing: true}

```

Давайте тут уже остановимся чуть-чуть поподробнее. Итак:

- **obstacle_range** - максимальная дальность, при которой робот будет учитывать препятствия и наносить их на свою карту. В нашем случае - 2.5 метра.
- **raytrace_range** - максимальность дальность, при которой будет происходить чистка карты от динамически возникающих объектов.
- **footprint** - размеры робота, предполагая, что его центр находится в точке (0;0),(0;0)
- **robot_radius** - размеры робота, если он круглый
- **inflation_radius** - увеличение ширины стен и препятствий. Например, если этот параметр задан, как 0.55, то тогда робот будет прокладывать маршрут таким образом, чтобы не притираться к увеличенным стенам/препятствиям на 0.55 метра. Посмотрите рисунок ниже, чтобы лучше понять это.



- **observation_sources** - список из датчиков, которые будут публиковать данные в карту затрат
- **laser_scan_sensor** или **point_cloud_sensor**, где в качестве **sensor_frame** устанавливается система координат датчика, **data_type** - тип датчика, **topic** - имя топика, **marking** и **clearing** - определение разрешения на добавление и очистку соответственно препятствий с карты затрат. Стандартным типом данных для лидара в ROS является LaserScan, а для камер глубины или 3D лидаров - PointCloud.

Надеемся, что теперь стало еще более понятно. Поэтому давайте откроем наш **costmap_common_params_[model_name].yaml**

```
obstacle_range: 3.0
raytrace_range: 3.5

footprint: [[-0.205, -0.155], [-0.205, 0.155], [0.077, 0.155], [0.077, -0.155]]
#robot_radius: 0.17

inflation_radius: 1.0
cost_scaling_factor: 3.0
```

```
map_type: costmap
observation_sources: scan
scan: {sensor_frame: base_scan, data_type: LaserScan, topic: scan, marking: true,
clearing: true}
```

💡 Поменяйте `inflation_radius` нашего робота. Верно ли то, что при каком-то значении он не сможет проехать между препятствиями? Если да, то при каком?

Local and Global Costmap

Разбираемся с глобальной картой стоимости, открываем соответствующий `.yaml` файл.

- `global_frame` - параметр, определяющий в какой системе координат будет карта стоимости.
- `robot_base_frame` - фрейм системы координат робота,
- `update_frequency` - частота обновления карты
- `publish_frequency` - частота публикации карты
- `transform_tolerance` - длительность допустимой задержки в ожидании преобразования
- `static_map` - принимает значения либо `True` либо `False` и определяет должен ли робот локализовать себя на уже существующей, подгруженной карте
- `footprint` - габариты робота на плоскости

Таким образом, мы разобрали параметры нашего `global_costmap_params.yaml`

```
global_costmap:
  global_frame: map
  robot_base_frame: base_footprint

  update_frequency: 10.0
  publish_frequency: 10.0
  transform_tolerance: 0.5

  static_map: true
```

В `local_costmap_params.yaml` должно быть все относительно понятно из предыдущего разбора, за исключением параметра `rolling_window`, принимающего либо `True`, либо `False`. Если параметр будет принимать значение `True`, то в таком случае карта стоимости будет оставаться центрированной вокруг робота, когда робот перемещается по миру. Параметры `width`, `height` и `resolution` задают ширину, высоту и разрешение в метрах карты затрат. Посмотрим на него:

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_footprint

  update_frequency: 10.0
  publish_frequency: 10.0
  transform_tolerance: 0.5
```

```
static_map: false
rolling_window: true
width: 3
height: 3
resolution: 0.05
```

Основные параметры. Move Base params

Осталось совсем немного, давайте напоследок взглянем на наш `move_base_params.yaml`

```
shutdown_costmaps: false
controller_frequency: 10.0
planner_patience: 5.0
controller_patience: 15.0
conservative_reset_dist: 3.0
planner_frequency: 5.0
oscillation_timeout: 10.0
oscillation_distance: 0.2
```

К основным параметрам `move_base` относятся:

- `base_global_planner` - тип глобального планировщика
- `base_local_planner` - тип локального планировщика
- `controller_frequency` - частота работы навигации и отправки команд роботу
- `controller_patience` - время ожидания адекватных данных управления до очистки карты
- `planner_frequency` - частота обновления глобального плана
- `planner_patience` - время ожидания поиска пути до очистки карты
- `oscillation_timeout` - время восстановления данных (карты)
- `oscillation_distance` - расстояние, которое нужно пройти для восстановления

Чему научились?

В этот раз мы с вами еще подробнее погрузились в стек навигации роботов и разобрали, как конфигурировать стек `move_base`. Как ранее говорилось, все элементы стека можно менять будто кубики LEGO 🎲 и подбирать планировщики и плагины под свои задачи. Стоит запомнить, что выбор плагинов и соответствующих конфигов осуществляется в файле `move_base.launch`.

Задание

Основное задание для вас - это пройти внимательно этот топик, почитайте внимательно про параметры, посмотрите на что они влияют. И убедитесь, что даже небольшое изменение одного из них может существенно сказаться на всем планировщике в целом. С этим нужно быть внимательным.

Когда ты сделаешь все это, посмотри еще раз выше, как менять в запуске тип глобального планировщика. В связи с этим, наше дополнительное (но супер желательное 💯) задание для тебя: проделать все тоже самое, но для локального планировщика. Предлагаем поработать с `teb_local_planner`.

💡 Установите командой на компьютер `teb_local_planner` аналогично тому, как вы устанавливали `dwa_local_planner`

Удачи! 🚗 🚗 🚗

Вопросики

1. Имеет ли смысл в глобальном планировщике использовать параметр `rolling_window`? Почему?
2. На какую карту помещаются резко возникающие препятствия?
3. Возможно ли очистить карту во время движения робота от динамически возникающих объектов (людей, кошечек, машинок)? Или они так и останутся на ней.

Ресурсы

Более подробную информацию о настройке стека навигации вы можете найти:

- [Basic Navigation Tuning Guide](#)
- [ROS Navigation Tuning Guide by Kaiyu Zheng](#)
- 11 глава [ROS Robot Programming book](#)