

**Note:** This tutorial assumes that you have completed the previous tutorials: Comprendre les nodes ROS (/fr/ROS/Tutorials/UnderstandingNodes).

💡 Please ask about problems and questions regarding this tutorial on [answers.ros.org](https://answers.ros.org) (<http://answers.ros.org>). Don't forget to include in your question the link to this page, the versions of your OS & ROS, and also add appropriate tags.

# Comprendre les Topics ROS

**Description:** Ce tutoriel introduit les concepts de Topics sous ROS ainsi que l'utilisation des outils en ligne de commande rostopic (/rostopic) et rqt\_plot (/rqt\_plot).

**Tutorial Level:** BEGINNER

**Next Tutorial:** Comprendre les Services & paramètres ROS (/fr/ROS/Tutorials/UnderstandingServicesParams)

## Sommaire

1. Setup
  1. roscore
  2. turtlesim
  3. turtle keyboard teleoperation
2. Les Topics ROS
  1. Utilisation de rqt\_graph
  2. Introduction à l'utilitaire rostopic
  3. Utilisation de rostopic echo
  4. Utilisation de rostopic list
3. ROS Messages
  1. Utilisation de la commande rostopic type
4. rostopic la suite
  1. Utilisation de rostopic pub
  2. Using rostopic hz
5. Using rqt\_plot
2. Video Tutorial

## 1. Setup

### 1.0.1 roscore

Commençons par vérifier que Roscore fonctionne, **dans un nouveau terminal:**

```
$ roscore
```

Si vous aviez laissé roscore en fonction à la fin du tutoriel précédent, vous devriez obtenir le message d'erreur suivant:

```
roscore cannot run as another roscore/master is already running.  
Please kill other roscore/master processes before relaunching
```

C'est tout à fait normal. Nous n'avons besoin que d'une seule instance de roscore en service.

### 1.0.1 turtlesim

Nous allons une nouvelle fois utiliser Turtlesim dans ce tutoriel. **Dans un nouveau terminal, entrez:**

```
$ rosrun turtlesim turtlesim_node
```

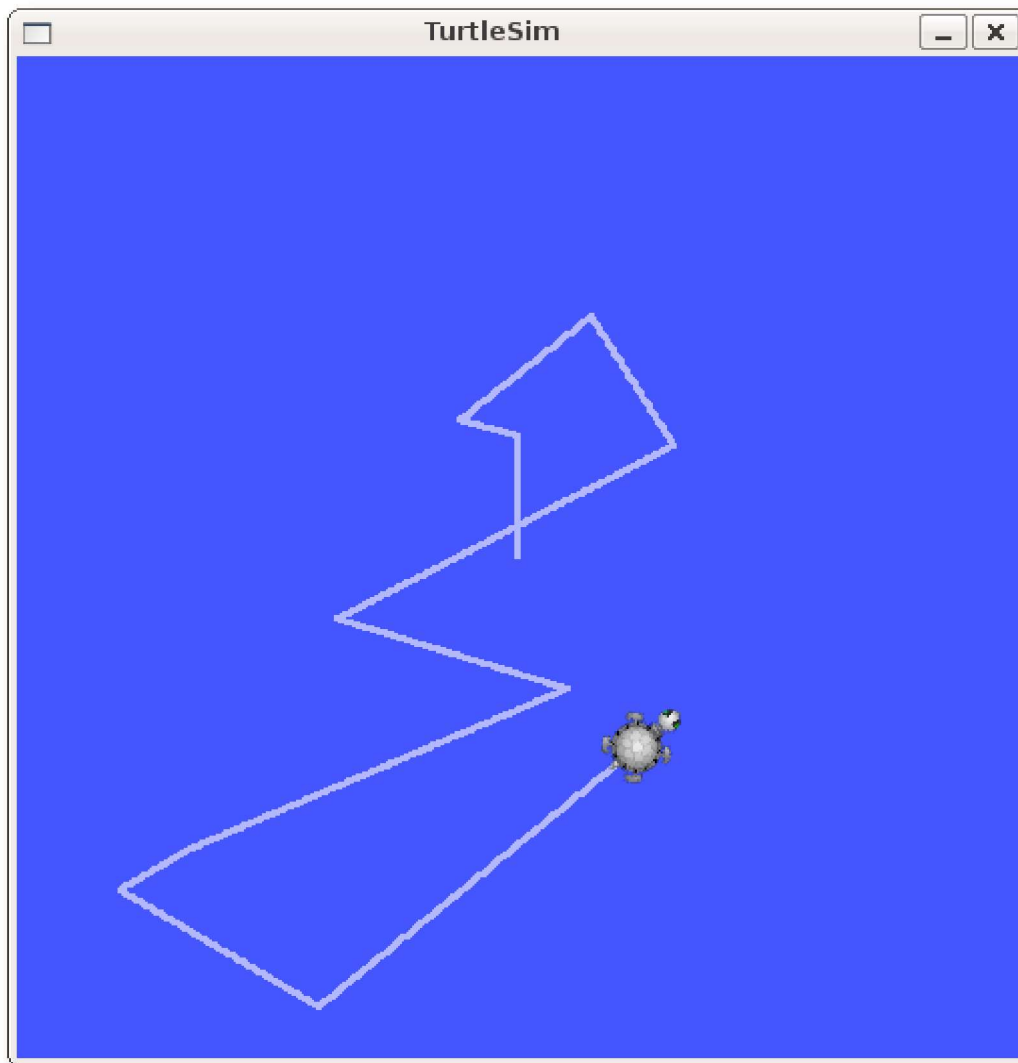
### 1.0.1 turtle keyboard teleoperation

Nous allons également avoir besoin de quelque chose pour diriger notre tortue. **Dans un nouveau terminal:**

```
$ rosrun turtlesim turtle_teleop_key
```

```
[ INFO] 1254264546.878445000: Started node [/teleop_turtle], pid [5528], bound on [aqy], xmlrpc port [43918], tcpport port [55936], logging to [~/ros/ros/log/teleop_turtle_5528.log], using [real] time  
Reading from keyboard  
-----  
Use arrow keys to move the turtle.
```

Désormais, vous pouvez utiliser les touches fléchées du clavier pour diriger la tortue. Si ce n'est pas le cas, n'oubliez pas de **sélectionner la fenêtre de terminal où le node turtle\_teleop\_key s'exécute** afin que les appuis sur les touches soient effectivement reçus.



Maintenant que vous êtes capables de diriger votre tortue, voyons ce qu'il se passe en coulisses.

## 1.1 Les Topics ROS

Les nodes `turtlesim_node` et `turtle_teleop_key` communiquent entre eux à travers un **Topic** ROS. `turtle_teleop_key` **publie** les actions sur les touches du clavier via ce topic, tandis que le node `turtlesim` **souscrit** à ce même topic afin de recevoir ces actions. Utilisons `rqt_graph` (`/rqt_graph`) qui montrera les nodes et topics actuellement actifs.

Note: Si vous utilisez (encore) la distribution `electric` ou antérieure, `rqt` n'est pas disponible. Utilisez `rxgraph` à la place.

### 1.1.1 Utilisation de `rqt_graph`

`rqt_graph` va créer un graphique dynamique représentant l'état du système. `rqt_graph` est inclus dans le package `rqt`. Si vous ne l'avez pas encore installé, lancez les commandes suivantes:

```
$ sudo apt-get install ros-<distro>-rqt
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

En remplaçant évidemment <distro> par le nom de la distribution que vous avez installé (hydro, indigo, jade, etc.)

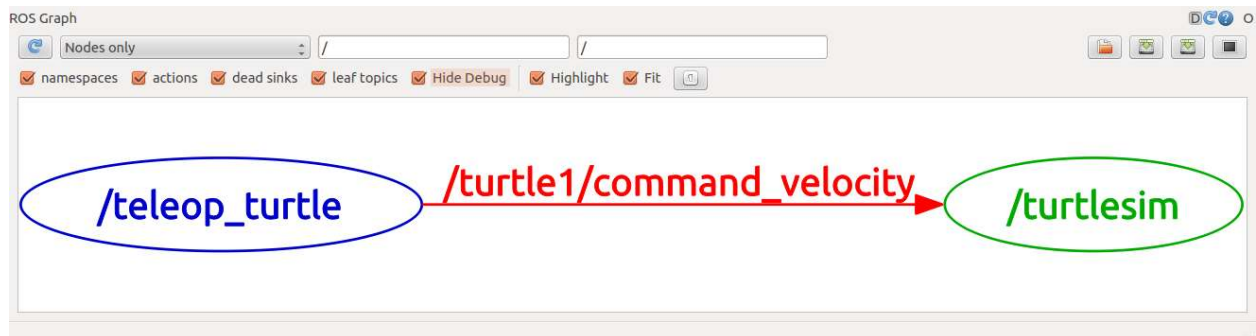
**Dans un nouveau terminal:**

```
$ rosrun rqt_graph rqt_graph
```

Vous devriez obtenir l'ouverture de la fenêtre suivante:



Si vous déplacez votre souris au dessus de `/turtle1/command_velocity`, cela devrait mettre en relief les nodes ROS (ici en vert & bleu), et les topics (en rouge). Comme vous pouvez le voir, les nodes `turtlesim_node` et `turtle_teleop_key` communiquent à travers le topic nommé `turtle1/command_velocity`.



### 1.1.2 Introduction à l'utilitaire rostopic

L'utilitaire `rostopic` permet d'obtenir des informations concernant les **topics** ROS.

Utilisez l'option `help` pour lister les sous-commandes de `rostopic`:

```
$ rostopic -h
```

```
rostopic bw      display bandwidth used by topic
rostopic echo    print messages to screen
rostopic hz      display publishing rate of topic
rostopic list    print information about active topics
rostopic pub     publish data to topic
rostopic type    print topic type
```

Utilisons quelques-unes de ces sous-commandes pour inspecter le fonctionnement de turtlesim.

### 1.1.3 Utilisation de rostopic echo

`rostopic echo` permet d'afficher les données publiées par un Topic.

Utilisation:

```
rostopic echo [topic]
```

Voyons les données du topic `command velocity` publiées par le node `turtle_teleop_key`.

*Pour ROS Hydro et ultérieurs, cette donnée est publiée via le topic `/turtle1/cmd_vel`. Dans un nouveau terminal, lancer la commande:*

```
$ rostopic echo /turtle1/cmd_vel
```

*Pour ROS Groovy et précédents, cette donnée est publiée via le topic `/turtle1/command_velocity`. Dans un nouveau terminal, lancer la commande:*

```
$ rostopic echo /turtle1/command_velocity
```

Vous ne verrez probablement rien parce qu'aucune donnée n'est actuellement publiée dans le topic. Faisons publier quelques données par `turtle_teleop_key` en appuyant sur les touches fléchées.

**Souvenez vous: si la tortue ne bouge pas à l'écran, sélectionnez le terminal dans lequel 'tourne' `turtle_teleop_key`.**

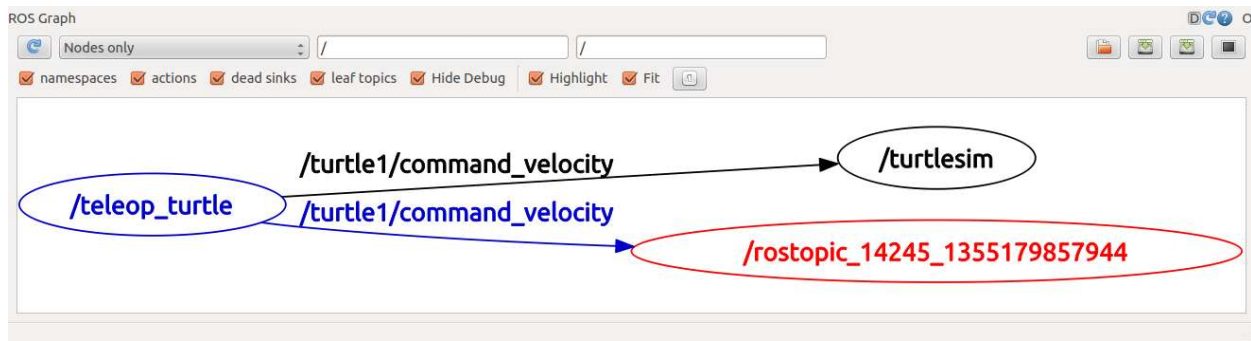
*Pour ROS Hydro et ultérieurs, vous devriez désormais voir quelque chose de similaire à ci-dessous, lorsque vous appuyez sur la touche 'flèche haut' du clavier:*

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

*Pour ROS Groovy et précédents, vous devriez voir quelque chose de similaire à ci dessous:*

```
---
linear: 2.0
angular: 0.0
---
linear: 2.0
angular: 0.0
---
linear: 2.0
angular: 0.0
---
linear: 2.0
angular: 0.0
---
linear: 2.0
angular: 0.0
```

Maintenant, retournons jeter un oeil sous `rqt_graph`. cliquer sur le bouton 'refresh' (coin supérieur gauche de la fenêtre) pour visualisez le nouveau node. Comme vous pouvez le voir, `rostopic echo`, ici en rouge, a également souscrit au topic `turtle1/command_velocity` (ou pour les distribs `hydro` & ultérieures, `turtle1/cmd_vel`)



### 1.1.4 Utilisation de rostopic list

la commande `rostopic list` renvoie une list de tous les topics actuellement souscrits & publiés.

Voyons maintenant ce que la sous commande `list` prend en arguments. Dans un nouveau terminal, lancer:

```
$ rostopic list -h
```

```
Usage: rostopic list [/topic]

Options:
  -h, --help            show this help message and exit
  -b BAGFILE, --bag=BAGFILE
                        list topics in .bag file
  -v, --verbose          list full details about each topic
  -p                    list only publishers
  -s                    list only subscribers
```

Si l'on utilise `rostopic list` avec l'option **verbose**:

```
$ rostopic list -v
```

Cela va afficher une liste 'bavarde' des topics publiant et souscrivants ainsi que leurs types.

```
Published topics:
* /turtle1/color_sensor [turtlesim/Color] 1 publisher
* /turtle1/command_velocity [turtlesim/Velocity] 1 publisher
* /rosout [roslib/Log] 2 publishers
* /rosout_agg [roslib/Log] 1 publisher
* /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:
* /turtle1/command_velocity [turtlesim/Velocity] 1 subscriber
* /rosout [roslib/Log] 1 subscriber
```

## 1.2 ROS Messages

La communication entre topics s'effectue par l'envoi de **messages** ROS entre les différents nodes.

Si l'on considère la communication entre le **publisher** (`turtle_teleop_key`) et le **subscriber** (`turtlesim_node`), ceux-ci doivent envoyer et recevoir le même **type** de message. Cela signifie en fait que le **type** de topic est défini par le **type** de messages qu'il utilise. Le **type** de message à envoyer vers un topic particulier peut être déterminé en utilisant la commande `rostopic type`.

### 1.2.1 Utilisation de la commande `rostopic type`

`rostopic type` renvoie le type de message de chaque topic publié.

Utilisation:

```
rostopic type [topic]
```

*Pour ROS Hydro et suivants,*

essayez:

```
$ rostopic type /turtle1/cmd_vel
```

Vous devriez voir apparaître le message suivant:

```
geometry_msgs/Twist
```

On peut également visualiser les détails de messages en utilisant la commande `rosmmsg`:

```
$ rosmmsg show geometry_msgs/Twist
```

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

*Pour ROS Groovy et précédents,*

essayez:

```
$ rostopic type /turtle1/command_velocity
```

Vous devriez obtenir:

```
turtlesim/Velocity
```



Encore une fois, vous pouvez obtenir plus de détails sur le message en utilisant la commande `rosmmsg`:

```
$ rosmmsg show turtlesim/Velocity
```

```
float32 linear  
float32 angular
```

Maintenant que nous savons quel type de messages `turtlesim` attend, nous pouvons commencer à publier (envoyer) des ordres à notre tortue.

## 1.3 rostopic la suite

Nous avons vu la composition des **messages ROS**, voyons maintenant l'utilisation des messages avec `rostopic`.

### 1.3.1 Utilisation de rostopic pub

`rostopic pub` publie les données vers un topic actif.

Fonctionnement:

```
rostopic pub [topic] [msg_type] [args]
```

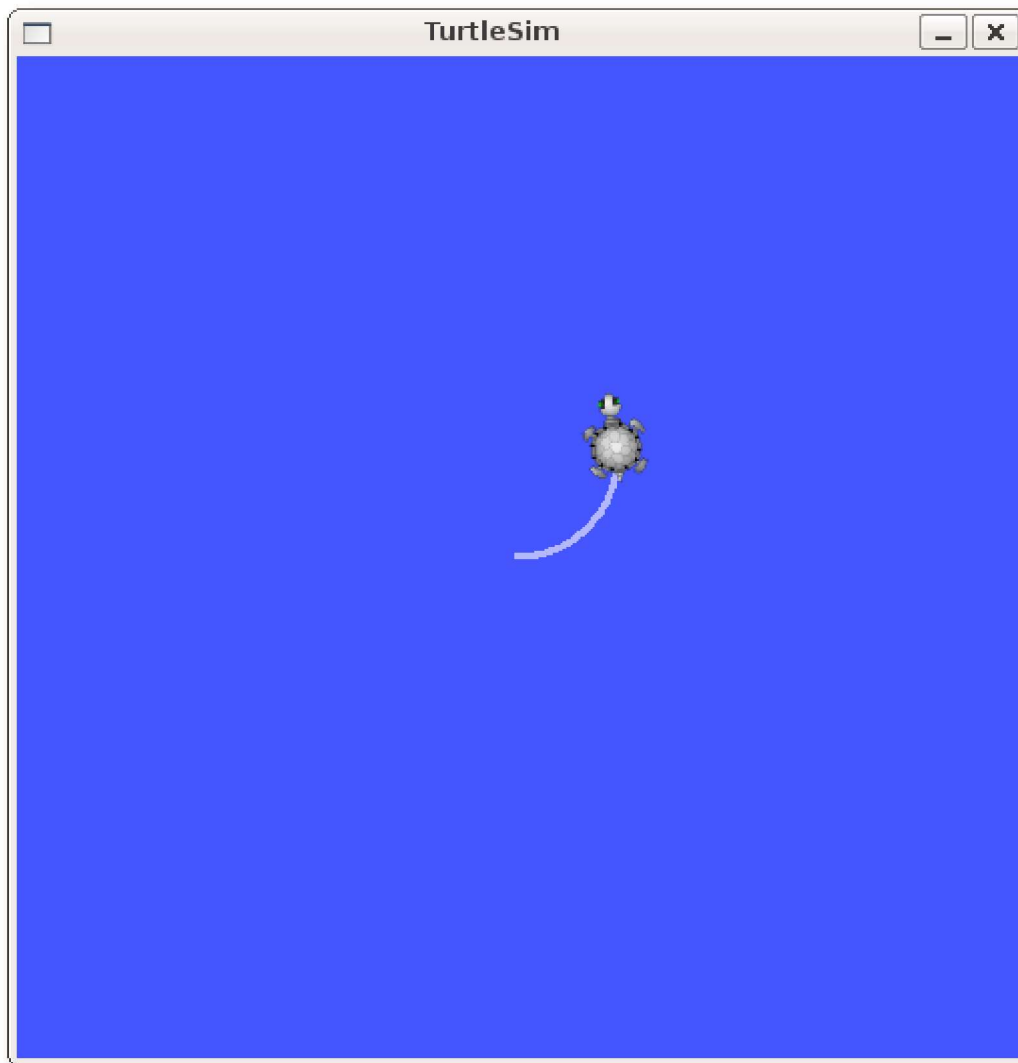
*Pour ROS Hydro et ultérieurs, exemple:*

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]'  
'[0.0, 0.0, 1.8]'
```

*Pour ROS Groovy et antérieurs, exemple:*

```
$ rostopic pub -1 /turtle1/command_velocity turtlesim/Velocity -- 2.0 1.8
```

Cette commande va envoyer un message unique vers `turtlesim`, lui ordonnant de se déplacer selon une vitesse linéaire de 2.0 et une vitesse angulaire de 1.8.



Cet exemple est un peu complexe, voyons chaque argument de la commande en détail.

*Pour ROS Hydro et ultérieurs,*

- Cette commande va publier des messages vers le topic donné:

```
rostopic pub
```

- Cette option (tiret-un) va générer un unique message puis sortir:

```
-1
```

- Le nom du topic vers lequel publier le message:

```
/turtle1/cmd_vel
```

- Le type de message utilisé lors de la publication vers le topic:

```
geometry_msgs/Twist
```

- Cette option (double-tiret) indique au parser d'options qu'aucune des données qui suivent ne sont à considérer comme des options: nécessaire lorsque les arguments passés au programme contiennent un ou des tirets, tels des nombres négatifs.

```
--
```

- Comme indiqué précédemment, un message du type `geometry_msgs/Twist` possède deux vecteurs composés chacun de trois éléments en virgule flottante: `linear` et `angular`. Dans le cas présent, les valeurs `'[2.0, 0.0, 0.0]'` correspondent aux valeurs du vecteur `linear` `x=2.0`, `y=0.0`, et `z=0.0`, et les valeurs `'[0.0, 0.0, 1.8]'` correspondent aux valeurs du vecteur `angular` `x=0.0`, `y=0.0`, et `z=1.8`. Ces valeurs sont transmises selon la syntaxe YAML, décrite plus en détails ici: [YAML command line documentation](#) (`/ROS/YAMLCommandLine`).

```
'[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

*Pour ROS Groovy et antérieurs,*

- Cette commande va publier les messages d'un topic donné:

```
rostopic pub
```

- L'option ( tiret-un) provoquera la publication d'un seul message avant arrêt:

```
-1
```

- Voici le nom du topic sous lequel on va publier les messages:

```
/turtle1/command_velocity
```

- This is the message type to use when publishing to the topic:

```
turtlesim/Velocity
```

- This option (double-dash) tells the option parser that none of the following arguments is an option. This is required in cases where your arguments have a leading dash -, like negative numbers.

```
--
```

- As noted before, a `turtlesim/Velocity` msg has two floating point elements : `linear` and `angular`. In this case, `2.0` becomes the linear value, and `1.8` is the angular value. These arguments are actually in YAML syntax, which is described more in the [YAML command line documentation](#) (`/ROS/YAMLCommandLine`).

```
2.0 1.8
```

You may have noticed that the turtle has stopped moving; this is because the turtle requires a steady stream of commands at 1 Hz to keep moving. We can publish a steady stream of commands using `rostopic pub -r` command:

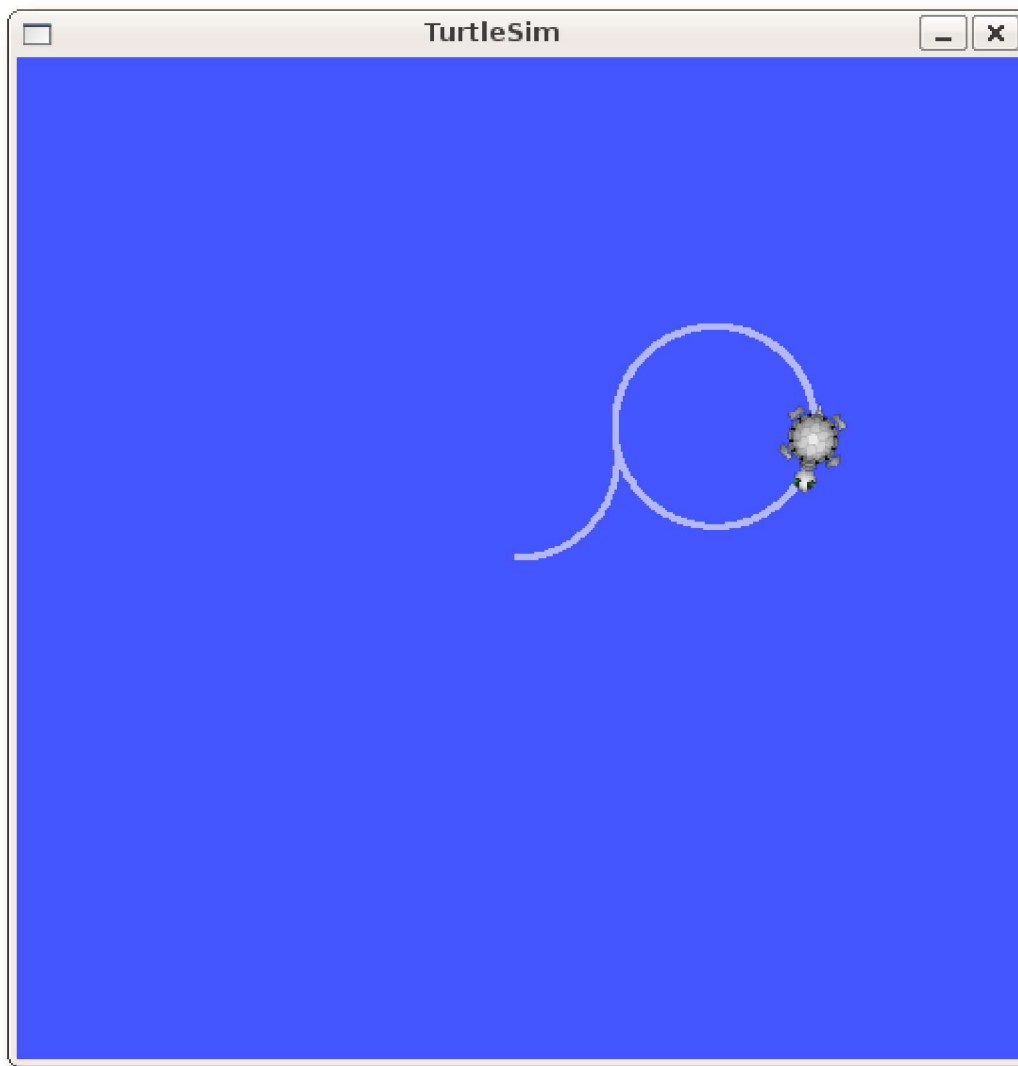
*For ROS Hydro and later,*

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

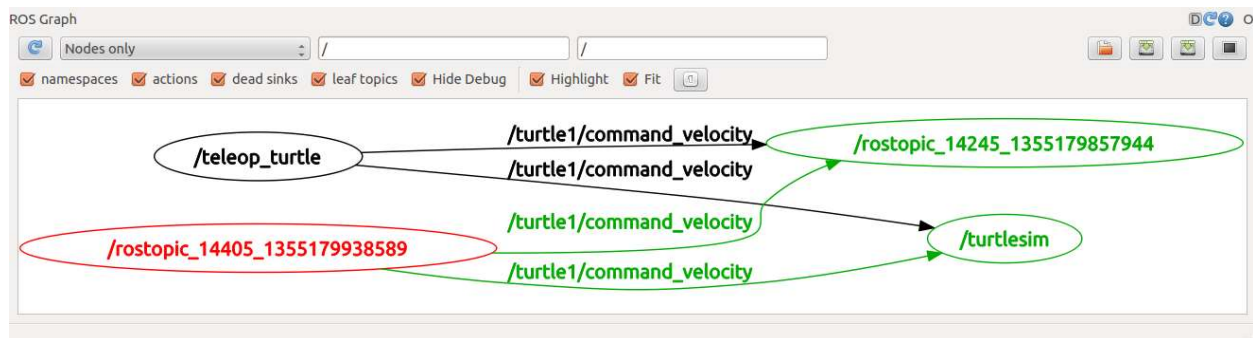
*For ROS Groovy and earlier,*

```
$ rostopic pub /turtle1/command_velocity turtlesim/Velocity -r 1 -- 2.0 -1.8
```

This publishes the velocity commands at a rate of 1 Hz on the velocity topic.



We can also look at what is happening in `rqt_graph`. The `rostopic pub` node (here in red) is communicating with the `rostopic echo` node (here in green):



As you can see the turtle is running in a continuous circle. In a **new terminal**, we can use `rostopic echo` to see the data published by our `turtlesim`:

### 1.3.2 Using `rostopic hz`

`rostopic hz` reports the rate at which data is published.

Usage:

```
rostopic hz [topic]
```

Let's see how fast the `turtlesim_node` is publishing `/turtle1/pose`:

```
$ rostopic hz /turtle1/pose
```

You will see:

```
subscribed to [/turtle1/pose]
average rate: 59.354
  min: 0.005s max: 0.027s std dev: 0.00284s window: 58
average rate: 59.459
  min: 0.005s max: 0.027s std dev: 0.00271s window: 118
average rate: 59.539
  min: 0.004s max: 0.030s std dev: 0.00339s window: 177
average rate: 59.492
  min: 0.004s max: 0.030s std dev: 0.00380s window: 237
average rate: 59.463
  min: 0.004s max: 0.030s std dev: 0.00380s window: 290
```

Now we can tell that the `turtlesim` is publishing data about our turtle at the rate of 60 Hz. We can also use `rostopic type` in conjunction with `rosmmsg show` to get in depth information about a topic:

For ROS Hydro and later,

```
$ rostopic type /turtle1/cmd_vel | rosmmsg show
```

For ROS Groovy and earlier,

```
$ rostopic type /turtle1/command_velocity | rosmmsg show
```

Now that we've examined the topics using `rostopic` let's use another tool to look at the data published by our `turtlesim`:

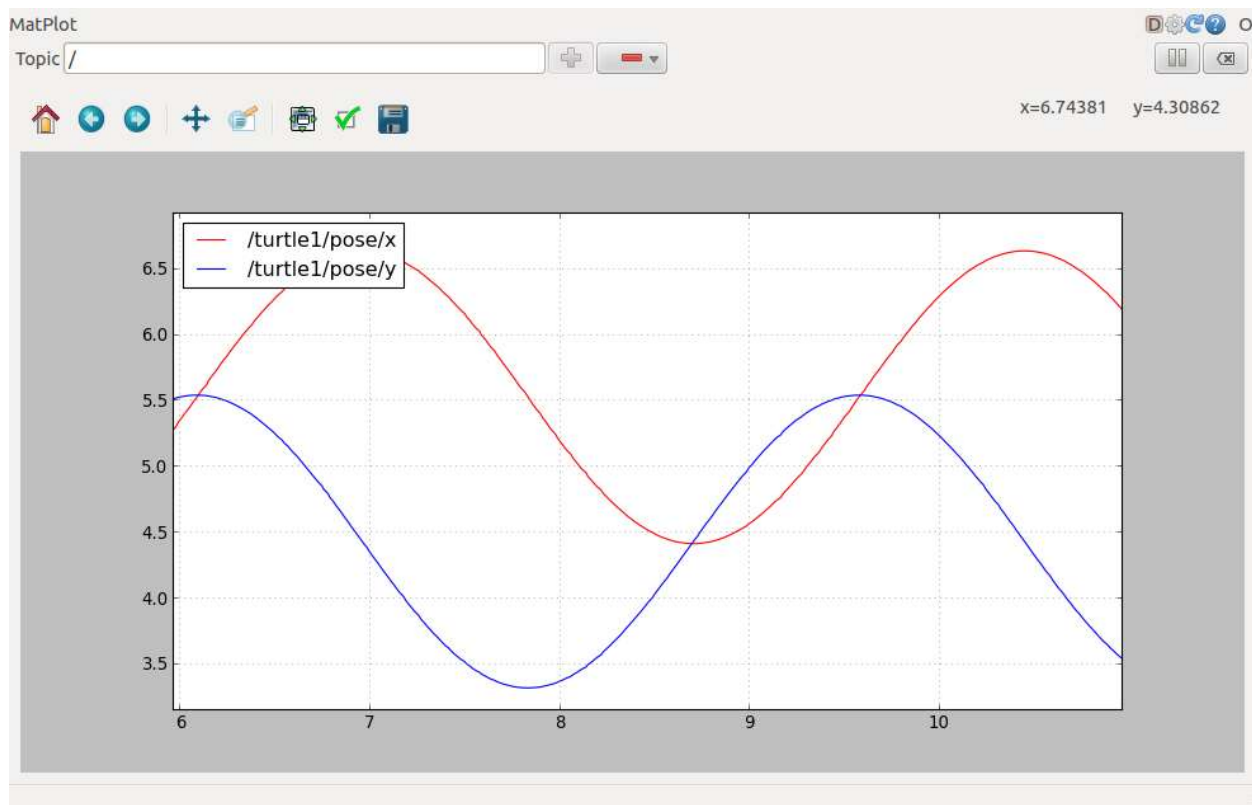
## 1.4 Using `rqt_plot`

Note: If you're using `electric` or earlier, `rqt` is not available. Use `rxplot` instead.

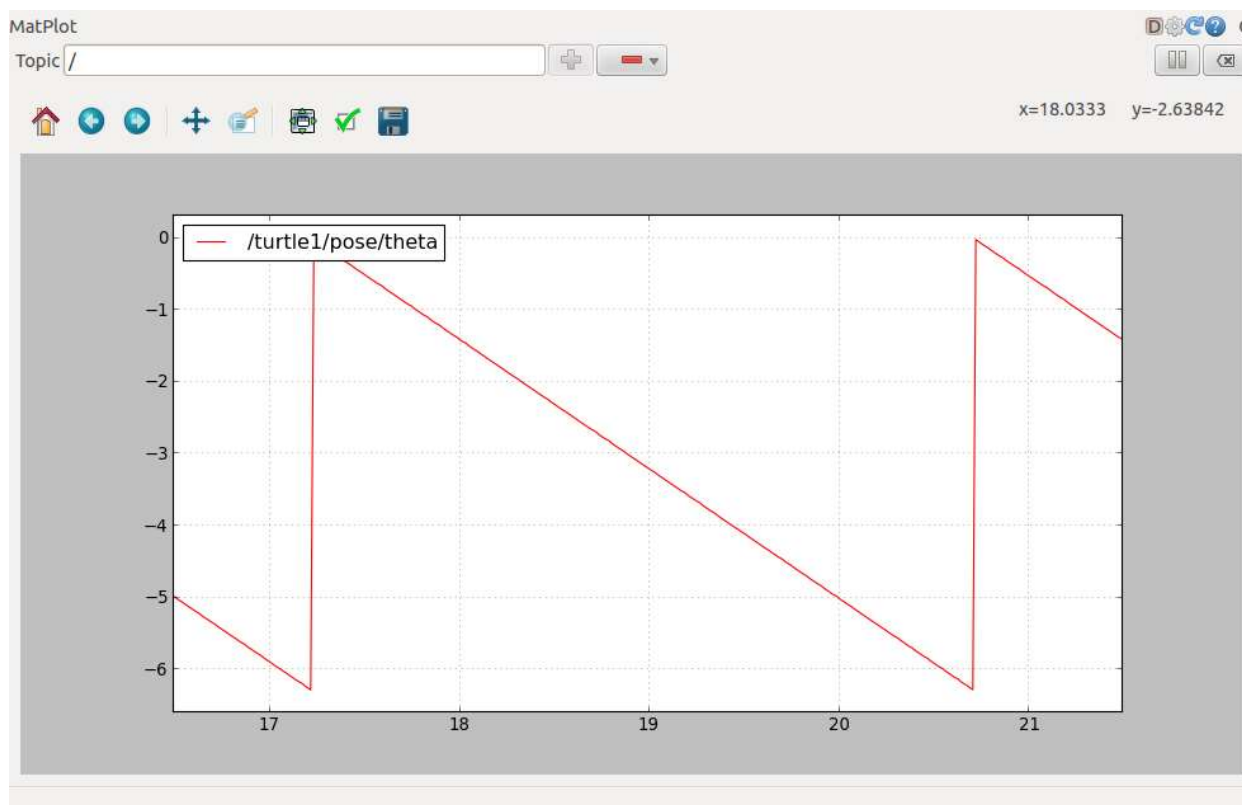
`rqt_plot` displays a scrolling time plot of the data published on topics. Here we'll use `rqt_plot` to plot the data being published on the `/turtle1/pose` topic. First, start `rqt_plot` by typing

```
$ rosrun rqt_plot rqt_plot
```

in a new terminal. In the new window that should pop up, a text box in the upper left corner gives you the ability to add any topic to the plot. Typing `/turtle1/pose/x` will highlight the plus button, previously disabled. Press it and repeat the same procedure with the topic `/turtle1/pose/y`. You will now see the turtle's x-y location plotted in the graph.



Pressing the minus button shows a menu that allows you to hide the specified topic from the plot. Hiding both the topics you just added and adding `/turtle1/pose/theta` will result in the plot shown in the next figure.

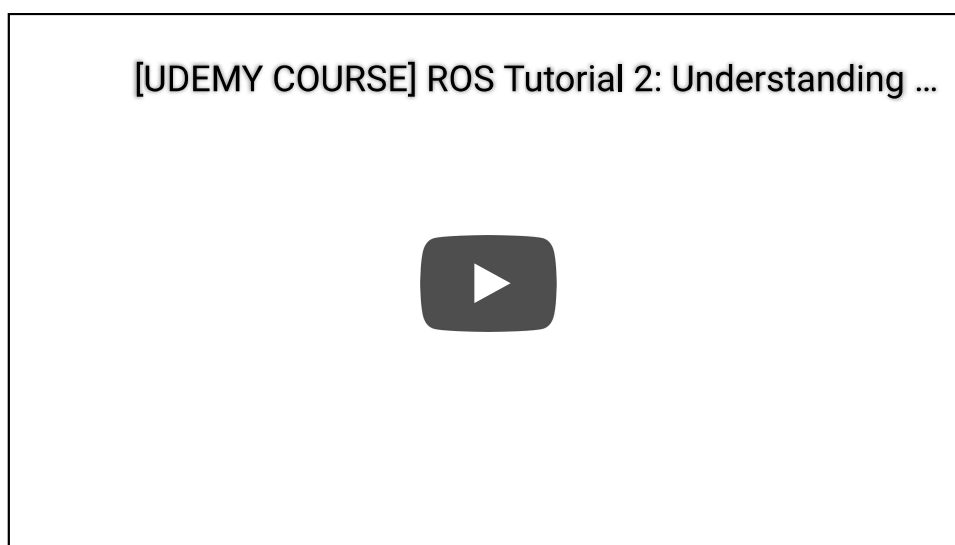


That's it for this section, use Ctrl-C to kill the rostopic terminals but keep your turtlesim running.


Now that you understand how ROS topics work, let's look at how services and parameters work (/ROS/Tutorials/UnderstandingServicesParams).

## 2. Video Tutorial

The following video presents a small tutorial using turtlesim on ROS nodes and ROS topics.



Except where otherwise noted, the ROS wiki is licensed under the

Wiki: fr/ROS/Tutorials/UnderstandingTopics (dernière édition le 2018-09-16 17:25:16 par  ArnaudMarot (mailto:rikuo.net@gmail.com))

Creative Commons Attribution 3.0 (<http://creativecommons.org/licenses/by/3.0/>)

Brought to you by:  Open Robotics

(<https://www.openrobotics.org/>)