Research Track II: Jupyter Notebook – Widgets and ROS

Carmine Tommaso Recchiuto

Exercise

- Given the simple turtlesim example:
    - Create a notebook that sets an angular and linear velocity for the turtle
    - Try to implement the same controller implemented in the RT1 course
    - Plots the position of the turtle*

*You can try to implement two different modalities to plot the position of the turtle. You can plot it «live» (more complex) or you can record the position for a certain number of seconds, and then plot the resulting graph.

Exercise

Solution 1:

Cell [1]

```
import rospy
from turtlesim.msg import Pose
from geometry_msgs.msg import Twist
import time
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

x_plot = []
y_plot = []
pub = rospy.Publisher("turtle1/cmd_vel", Twist, queue_size=1)
```

Exercise

Cell [2]

```
def turtleCallback(msg):
    global x_plot, y_plot
    x_plot.append(msg.x)
    y_plot.append(msg.y)

    vel = Twist()
    if (msg.x > 9.0):
        vel.linear.x = 1.0
        vel.angular.z = 1.0
    elif (msg.x < 1.5):
        vel.linear.x = 1.0
        vel.angular.z = -1.0
    else:
        vel.linear.x = 1.0
        vel.angular.z = 0.0
    pub.publish(vel)
```

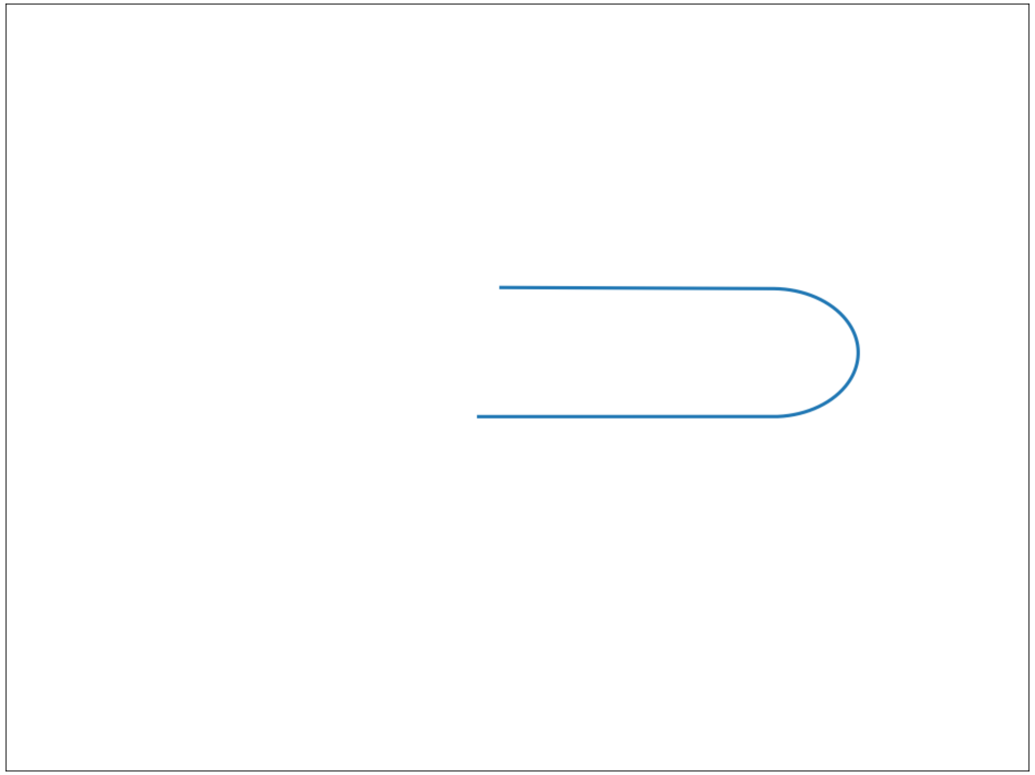
Cell [3]

```
%matplotlib widget
rospy.init_node("controller")
sub = rospy.Subscriber("turtle1/pose", Pose,
turtleCallback)
time.sleep(10)
sub.unregister()
vel = Twist()
vel.linear.x = 0.0
vel.angular.z = 0.0
pub.publish(vel)
```

Exercise

Cell [4]:

```
np_x_plot = np.array(x_plot)
np_y_plot = np.array(y_plot)
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.set_xlim(0, 12)
ax.set_ylim(0, 12)
ax.plot(np_x_plot,np_y_plot)
ax.set_title("my_turtle")
ax.set_xlabel("x")
ax.set_ylabel("y")
```



Exercise

Solution 2:

Let’s add the new Cell:

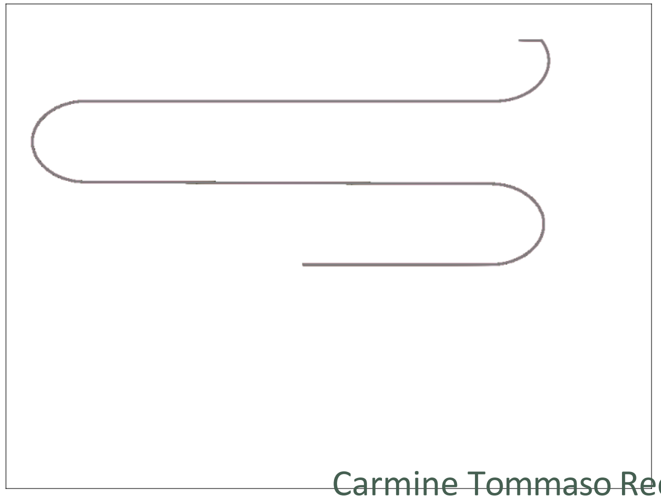
Cell [2]:

```
def plot(x_plot,y_plot):
    np_x_plot = np.array(x_plot)
    np_y_plot = np.array(y_plot)
    ax.plot(np_x_plot,np_y_plot)
```

And in the turtleCallback function, we call now the plot function:

Cell [3]:

```
[…]
    plot(x_plot,y_plot)
```



Finally, we remove from Cell [3] (which is now Cell [4]) the following lines

```
time.sleep(10)
sub.unregister()
```

iPyWidgets

- We can add some widgets (es. Buttons, Sliders) to our Jupyter Notebook.
- **ipywidgets**, also known as jupyter-widgets or simply widgets, are interactive HTML widgets for Jupyter notebooks and the IPython kernel.
- **To use the widget framework, you need to import ipywidgets:**

```
import ipywidgets as widgets
```

Constructing and returning an **IntSlider** automatically displays the widget. Widgets are displayed inside the output area below the code cell. Clearing cell output will also remove the widget.

```
w = widgets.IntSlider()
display(w)
```

All of the IPython widgets share a similar naming scheme. To read the value of a widget, you can query its value property:

```
w.value
```

iPyWidgets

- To see the entire list of the properties of a widget you can query the keys attribute:

```
w.keys
```

- While creating a widget, you can set some or all of the initial values of that widget by defining them as keyword arguments in the widget's constructor (as seen below).

```
widgets.Text(value='Hello World!', disabled=True)
```

- You can of course at any moment change the properties of the widget:

```
a = widgets.Text(value='Hello World!', disabled=True)
display (a)

a.disabled = False
```

iPyWidgets

- If you need to display the same value in two different ways, you'll have to use two different widgets. Instead of attempting to manually synchronize the values of the two widgets, you can use the jslink function to link two properties together. Below, the values of two widgets are linked together.

```
a = widgets.FloatText()
b = widgets.FloatSlider()
display(a,b)

mylink = widgets.jslink((a, 'value'), (b, 'value'))
```

iPyWidgets

- A lot of different widgets may be used:

- Numeric widgets (slider, progress, text)
- Boolean widgets (button, checkbox)
- Selection widgets (dropdown, radiobuttons, selectionslider, buttons)
- String widgets (textarea, password)
- Image widgets (an image that can be added to the document)
- Play (animation) widget -> it may perform animations by iterating on a sequence of integers with a certain speed
- Date picker
- Color picker
- File upload -> allows to upload any type of file(s) into the memory of the kernel

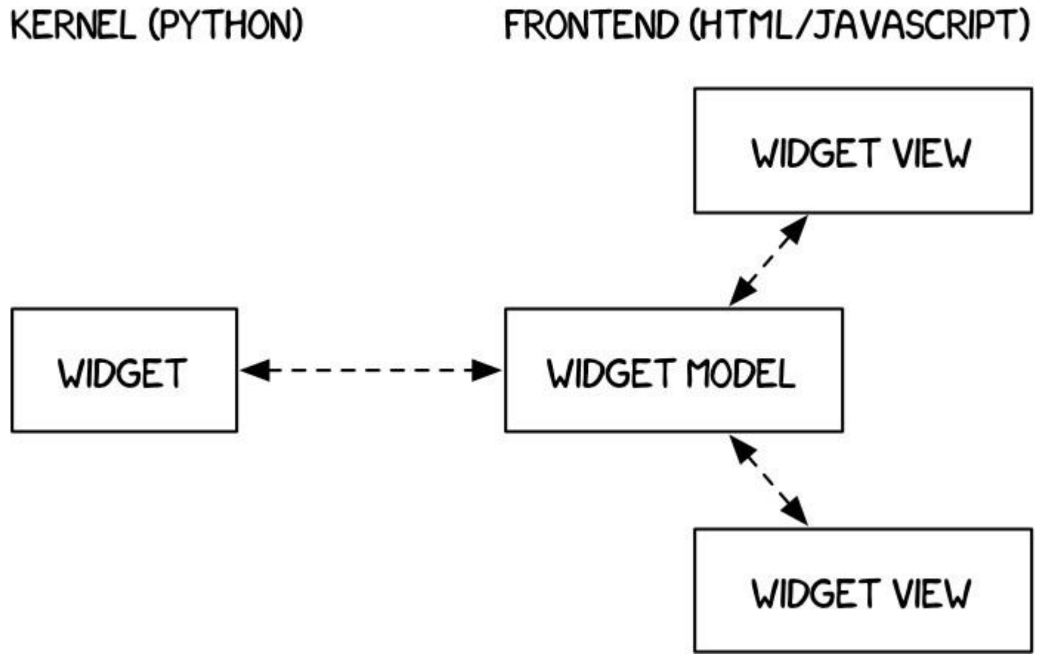
Additional information and some template examples can be found at: Widget List — Jupyter Widgets 8.0.0a4 documentation (ipywidgets.readthedocs.io)

Multiple Display calls

If you display the same widget twice, the displayed instances in the front-end will remain in sync with each other. Try dragging the slider below and watch the slider above.

```
display(a)
```

Widgets are represented in the back-end by a single object. Each time a widget is displayed, a new representation of that same object is created in the front-end. These representations are called views



Numeric Widgets

Widgets for displaying integers and floats, both bounded and unbounded. The integer widgets share a similar naming scheme to their floating point counterparts.

Es.

```
c = widgets.IntSlider(
    value=7,
    min=0,
    max=10,
    step=1,
    description='Test:',
    disabled=False,
    continuous_update=False,
    orientation='horizontal',
    readout=True,
    readout_format='d'
)
```

continuous_update is set to update the value of the widget only when the user explictly submits the value (in the case of the slider, when the user finishes dragging the slider

```
d = widgets.IntSlider()
display (c,d)
mylink2 = widgets.jslink((c,'value'),(d,'value'))
```

Numeric Widgets

Different types of numeric widgets exist:

FloatSlider

```
widgets.FloatSlider(orientation=‘vertical’)
```

FloatLogSlider

```
widgets.FloatLogSlider(value=10, base=10, min=-10, max=10, step=1.0, description='Log Slider’)
```

Float(Int)RangeSlider

```
widgets.FloatRangeSlider(value=[5, 7.5], min=0, max=10.0)
```

Float(Int)Progress

```
widgets.FloatProgress(value=7.5, min=0, max=10.0, bar_style='info’)
```

BoundedFloat(Int)Text

```
widgets.BoundedFloatText(value=7.5, min=0, max=10.0)
```

Float(Int)Text

```
widgets.FloatText(value=7.5)
```

Boolean Widgets

Three widgets designed to display a boolean value:

ToggleButton

```
widgets.ToggleButton(
    value=False,
    description='Click me',
    disabled=False,
    button_style='', # 'success', 'info', 'warning', 'danger' or ‘’
    icon = ‘check’
)
```

Checkbox

```
widgets.Checkbox(
    value=False,
    description='Check me',
    disabled=False
)
```

Valid (read only)

```
widgets.Valid(
    value=False,
    description='Valid!',
)
```

A boolean widget can still be linked to numeric widget as seen before.

Selection Widgets

Several widgets can be used to display selection lists, and two that can be used to select multiple values
The value, which is a string, can be expressed in terms of (label, value) pairs.

Dropdown

```
widgets.Dropdown(
    options=[('One', 1), ('Two', 2), ('Three', 3)],
    value=2,
    description='Number:',
)
```

RadioButtons

```
widgets.RadioButtons(
    options=[('One', 1), ('Two', 2), ('Three', 3)],
    description='Number:',
    disabled=False
)
```

Selection Widgets

ToggleButtons

```
widgets.ToggleButtons(
    options=[('One', 1), ('Two', 2), ('Three', 3)],
    description='Number:',
    button_style='', # 'success', 'info', 'warning', 'danger' or ‘’
)
```

SelectMultiple

```
widgets.SelectMultiple(
    options=[('One', 1), ('Two', 2), ('Three', 3)],
    value=[2], # this should be a tuple
    description='Number:'
)
```

String Widgets

Different widgets can be used to display a string value. The Text, Textarea, and Combobox widgets accept input.

```
widgets.Textarea(
   value='Hello World',
   placeholder='Type something',
   description='String:'
)
```

```
widgets.Combobox(
    placeholder='Choose Someone',
    options=['Paul', 'John', 'George', 'Ringo'],
    description='Combobox:'
)
```

The HTML box may be used to render mathematical equations:

```
widgets.HTMLMath(
   value=r"Some math and <i>HTML</i>: \(x^2\) and $$\frac{x+1}{x-1}$$",
   placeholder='Some HTML',
   description='Some HTML',
)
```

Widgets Layout

- There are also different possibilities for defining the position and the dimensions of the widgets in the Notebook.

```
from ipywidgets import Button, Layout

b = Button(description='(50% width, 80px height) button',
       layout=Layout(width='50%', height='80px'))
b
```

- Moreover, container widgets can be used. Containers may hold other widgets. They can be horizontal, or vertical.

```
item1 = widgets.IntSlider()
item2 = widgets.Text(value='Hello World')
widgets.Box([item1, item2])
```

Widgets Layout

- Also, the Vbox and Hbox helpers may be used to arrange child widgets in vertical and horizontal boxes

```
from ipywidgets import Button, Layout, ButtonStyle, GridBox, VBox, HBox
import ipywidgets as widgets


b1  = Button(description='Button1',
        layout=Layout(width='auto', align="center",
        grid_area='b1'),
        style=ButtonStyle(button_color='lightblue'))
b2   = Button(description='Button2',
        layout=Layout(width='auto', grid_area='b2'),
        style=ButtonStyle(button_color='moccasin'))
b3 = Button(description='Button3',
        layout=Layout(width='auto', grid_area='b3'),
        style=ButtonStyle(button_color='salmon'))

HBox([b3,VBox([b1, b2])])
```

Alternatively, a Grid Box can be used:

```
GridBox(children=[b1, b2, b3], layout=Layout(
      width='40%', grid_template_rows='auto auto',
      grid_template_columns='33% 33% 33%',
      grid_template_areas='''
      " . b1 . "
      "b2 . b3 "
      ''')
      )
```

Additional info can be found here
https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20Styling.html#

Output Widgets

- Output widgets are particularly interesting, since they are able to capture and display stdout, stderror and rich output generated by IPython.

```
out = widgets.Output(layout={'border': '1px solid black'})
out
```

- After the widget is created, direct output to it using a context manager. You can print text to the output area. But also rich output can also be directed to the output area. Anything which displays nicely in a Jupyter notebook will also display well in the Output widget.

```
with out:
    for i in range(10):
        print(i, 'Hello world!')

from IPython.display import YouTubeVideo
with out:
    display(YouTubeVideo('eWzY2nGfkXk'))
```

Output Widgets

The output widget forms the basis of how interact and related methods are implemented. It can also be used by itself to create rich layouts with widgets and code output. One simple way to customize how an interact UI looks is to use the ***interactive_output function*** to hook controls up to a function whose output is captured in the returned output widget.

```
a = widgets.IntSlider(description='a')
b = widgets.IntSlider(description='b')
c = widgets.IntSlider(description='c')
def f(a, b, c):
    print('{}*{}*{}={}'.format(a, b, c, a*b*c))

out = widgets.interactive_output(f, {'a': a, 'b': b, 'c': c})

widgets.HBox([widgets.VBox([a, b, c]), out])
```

Widgets Events

Let’s see now how we can practically use widgets in our jupyter notebook example. Dealing with Buttons (Boolean widgets) we can use the on_click method, associating a callback to the event. To capture prints (or any other kind of output) and ensure it is displayed, be sure to send it to an Output widget.

```
from IPython.display import display
button = widgets.Button(description="Click Me!")
output = widgets.Output()

display(button, output)
#handle changes, the observe method of the widget can be
used to register a callback.
def on_button_clicked(b):
  with output:
    print("Button clicked.")

button.on_click(on_button_clicked)
```

- ✓ The widget can be used to modify the plot built during the previous week using matplotlib. Consider that you can clean a plot by using the plt.cla() function

Widgets Events

```
import matplotlib as mpl
import matplotlib.pyplot as plt

import numpy as np
import math
%matplotlib widget
x = np.arange(0, math.pi*2, 0.05)
y = np.sin(x)
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.plot(x,y)
ax.set_title("sine_wave")
ax.set_xlabel("angle")
ax.set_ylabel("sin")
```

```
button = widgets.Button(description="Click Me!")
display(button)
def on_button_clicked(b):
        global x, y
        x = x *2
        y = y = np.sin(x)
        plt.cla()
        ax.plot(x,y)

button.on_click(on_button_clicked)
```

Widgets Events

✓ To handle general changes, the observe method of the widget can be used to register a callback.

```
int_range = widgets.IntSlider()
output2 = widgets.Output()

display(int_range, output2)

def on_value_change(change):
  with output2:
            print(change['new'])

int_range.observe(on_value_change, names='value')
```

```
int_range = widgets.IntSlider()

display(int_range)

def on_value_change(change):
        x_new = x*change['new']
        y_new = np.sin(x_new)
        plt.cla()
        ax.plot(x_new,y_new)

int_range.observe(on_value_change, names='value')
```

Widgets Events

- A similar way to deal with widgets events is to use the *interact* function, which may simplify things in some cases.

- *interact* autogenerates UI controls for function arguments, and then calls the function with those arguments when you manipulate the controls interactively. To use interact, you need to define a function that you want to explore:

```
from ipywidgets import interact, interactive, fixed,
interact_manual
import ipywidgets as widgets

def f(x):
        return x
```

When you pass this function as the first argument to interact along with an integer keyword argument (x=10), a slider is generated and bound to the function parameter.

```
interact(f, x=10);
```

When you move the slider, the function is called, and its return value is printed.

- If you pass True or False, interact will generate a checkbox, if you pass a string, interact will generate a text box

Jupyter and ROS

- Luckily the Jupyter Notebooks and Lab may be used and integrated with ROS. As for the other libraries, we need to install some extensions:

  - pip3 install jupyros

Moreover, you can install a specific library for jupyterlab. You can install it directly by using the Extension Manager (or by using the command **jupyter labextension install jupyter-ros** and **jupyter labextension install @jupyter-widgets/jupyterlab-manager**)

As a first example, the jupyter-ros tools may help in publishing and receiving messages in the Jupyter Notebook.

For receiving, the jupyter-ros package contains a helper that redirects output to a specific output widget (instead of spamming the entire notebook).

```
import jupyros as jr
import rospy
from std_msgs.msg import String

rospy.init_node('jupyter_node')
jr.subscribe('/sometopic', String, lambda msg: print(msg))
```

Jupyter and ROS

This creates a output widget, and buttons to toggle (stop or start) receiving messages. Internally, jupyter-ros will save a handle to the subscriber thread. Note that we did not use the rospy-way of creating a subscriber but delegated this to the jupyter-ros package.

For the publishing, the package contains tools to automatically generate widgets from message definitions.

```
import jupyros as jr
import rospy
from std_msgs.msg import String

rospy.init_node('jupyter_node')
jr.publish('/sometopic', String)
```

This results in a jupyter widget where one can insert the desired message in the text field. The form fields (jupyter widgets) are generated automatically from the message definition. If we use a a different message type, we will get different fields. For example, a Vector3 message type contains three float fields (x,y, and z) for which we will get 3 FloatTextField instances – these can only hold float values (and not text).

Jupyter and ROS

Jupyter-ros offer also some options for visualizing data. As an alternative to rqt_plot, we can use in Jupyter:

def live_plot(plot_string, topic_type, history=100, title=None).

As an example, we could plot the x and y position of our beloved turtlesim.

```
from turtlesim.msg import Pose
jr.live_plot("/turtle1/pose/:x:y", Pose, 10000)
```

However this has some limitation: that's why the jupyter-ros tool also allows for visualizing robots by means of the ros3d widgets. For jupyter-ros, the powerful [ROS3D.js]() library has been used to create widgets for the jupyter notebook frontend: it allows for creating some robot viewers inside the notebook!

```
from jupyros import ros3d

v = ros3d.Viewer()
v.objects = [ros3d.GridModel()]
v
```

Jupyter and ROS

We could add additional elements to the visualization:

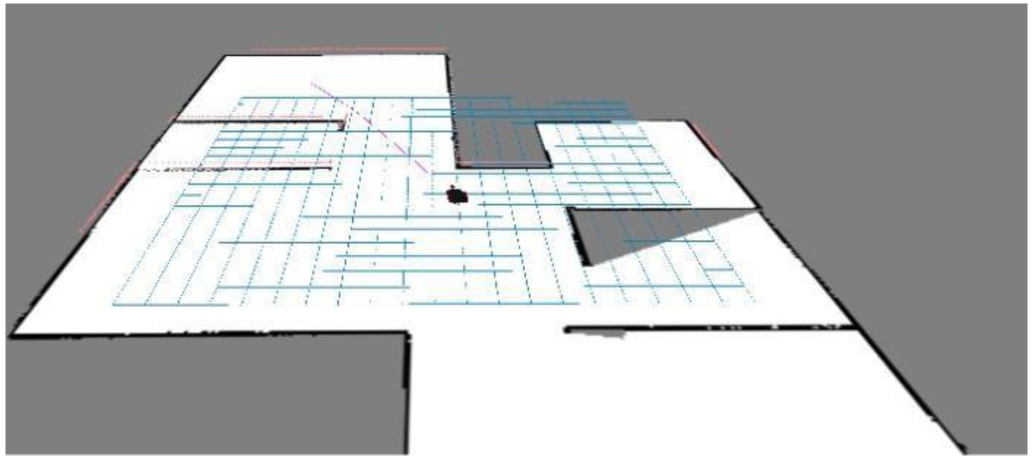
```
from jupyros import ros3d
import os

v = ros3d.Viewer()
rc = ros3d.ROSConnection(url="ws://localhost:9090")
tf_client = ros3d.TFClient(ros=rc, fixed_frame='odom')

laser_view = ros3d.LaserScan(topic="/scan", ros=rc, tf_client=tf_client)
urdf = ros3d.URDFModel(ros=rc, tf_client=tf_client,
g = ros3d.GridModel()

v.objects = [g, laser_view, urdf]

v
```



Jupyter, Matplotlib and ROS

Of course we can also use Matplotlib function for plotting ROS data in a Jupyter Notebook (we will see how to avoid the need for manually updating the graph).

```
%matplotlib widget
import numpy as np
import matplotlib.pyplot as plt
import jupyros as jr
import rospy
from nav_msgs.msg import Odometry
import ipywidgets as widgets

x_data=[]
y_data=[]
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
line, = ax.plot([], [], 'ro')
ax.set_xlim(( -20, 20))
ax.set_ylim((-20, 20))
ax.set_title("my_robot")
ax.set_xlabel("x")
ax.set_ylabel("y")
```

```
def odom_callback(msg):
    global x_data, y_data
    y_data.append(msg.pose.pose.position.y)
    x_data.append(msg.pose.pose.position.x)


rospy.init_node('odom_visualizer_node')
jr.subscribe('/odom', Odometry, odom_callback)

from IPython.display import display
button = widgets.Button(description="Click Me!")
output = widgets.Output()
```

```
display(button, output)
#handle changes, the observe method of
the widget can be used to register a
callback.
def on_button_clicked(b):
    np_x_plot = np.array(x_data)
    np_y_plot = np.array(y_data)
    fig = plt.cla()
    ax.set_xlim(( -20, 20))
    ax.set_ylim((-20, 20))
    ax.plot(np_x_plot,np_y_plot)

button.on_click(on_button_clicked)
```

Assignment II-A

- Starting from the second assignment of the course, create a jupyter notebook to replace the user interface (the node «A»)

- Try using widgets to let the user know the position of the robot and all targets that have been set and cancelled in the environment