

# Research Track II

## Software Documentation

Carmine Tommaso Recchiuto

# Course Structure

Research Track II will ideally continue the work performed in the first semester course.

With Research Track I, you should be able to deal with simple, but effective, software architectures for robotics, and you should well know the principles of ROS.

In Research Track II, we will also deal with general problems related to robotics (but not only) research: data visualization, code documentation, Jupyter notebooks, statistics, report writing.

# Course Structure

- Software Documentation
- Communicating Evidence Visually
- Notebook for Robotics
- Statistics in Robotics
- Elements of Research Methodology (Research Lines)

# Course Structure

Two main activities:

1)

- Continuous evaluation: During classes you will work starting of the assignments of RT1. An assignment is to be delivered around 24<sup>th</sup> May

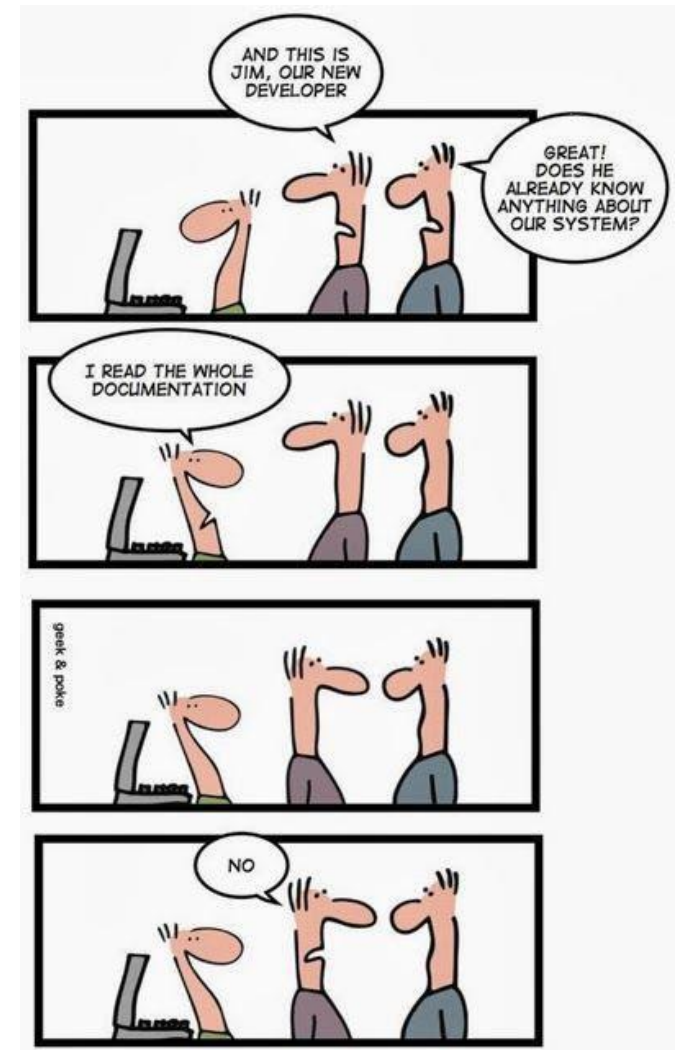
2)

- Research Line choice: based on your preferences, you will be allocated to a specific research line (in groups)

You will prepare a report and discuss your work

# Documentation?

- ✓ API (functions, methods) documentation – Reference documentation regarding making calls and classes
- ✓ README – A high-level overview of the software, usually alongside the source code
- ✓ Release notes - Information describing the latest software or feature releases, and bug fixes
- ✓ System documentation – Documents describing the software system, including technical design documents, software requirements, and UML diagrams



# Documentation Tools

Documenting your code, especially large projects, can be daunting. Thankfully there are some tools out and references to get you started:

Sphinx: <https://www.sphinx-doc.org/en/master/> A collections of tools to auto-generate documentation in multiple formats

Doxygen: <https://www.doxygen.nl/manual/docblocks.html> A tool for generating documentation that supports Python as well as multiple other languages



BE AWARE!!!



SOMEBODY MAY ACTUALLY READ IT!

# Doxygen

Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL

- It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
- You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically

# Doxygen - Example

On Ubuntu (and on the Docker image, but without sudo):

```
sudo apt-get install -y doxygen  
sudo apt-get install doxygen-gui
```





# Doxygen - Tips

But how should you comment your code for having a complete documentation in Doxygen?

A special comment block is a C or C++ style comment block with some additional markings, so doxygen knows it is a piece of structured text that needs to end up in the generated documentation. For Python and cpp there are different comment conventions.

For each entity in the code there are two (or in some cases three) types of descriptions, which together form the documentation for that entity; a *brief* description and *detailed* description, both are optional. For methods and functions there is also a third type of description, the so called *in body* description, which consists of the concatenation of all comment blocks found within the body of the method or function.

Having more than one brief or detailed description is allowed (but not recommended, as the order in which the descriptions will appear is not specified).

As the name suggest, a brief description is a short one-liner, whereas the detailed description provides longer, more detailed documentation.

# Doxygen - Tips

So, let's try to add Doxygen documentation to one of the package that we have used in RT1.

At first, you need to add a description of the file:

```
/**  
 * \file exercise1.cpp  
 * \brief Controller for the turtlesim  
 * \author Carmine Recchiuto  
 * \version 0.1  
 * \date 27/02/2024  
 **/
```

**\file** is needed to document a file! (if you don't have this command, you will not see the documentation related to that file, unless you select the option All-Entities in your doxygen-gui configuration).

**\brief** for the brief description can be removed in case the option JAVADOC\_AUTOBRIEF is set to yes.

For special commands, you may use @ instead of \

# Doxygen - Tips

The general description is not ended yet.

```
* \param [in] world_width Define the width of the discretized world.
*
* \details
*
* Subscribes to: <BR>
*   ° /robot_behavior_state_machine/smach/container_status
*
* Publishes to: <BR>
*   ° /PlayWithRobot
*
* Services : <BR>
*   ° /GiveGesture
*
* Description :
*
* This node simulates a person behavior. The person is assumed to move randomly in the environment.
* The person calls the robot to play in an interval indicated with the use of the parameters:
* minum_time_btw_calls and maximum_time_btw_calls.
*
*/
```

Commenting a ros node, we could add:

- ROS parameters used, specifying if they are input or output parameters
- Subscribed Topics
- Published Topics
- Advertised Services
- Clients
- Action Clients / Services
- A more detailed description of the node

# Doxygen - Tips

Please consider that in the example we are using the so-called JAVADOC style, which consist of a C-style comment block starting with two `'s`, like this:

```
/**
 * ... text ...
 */
```

There exists other possible ways for commenting c++ code:

- you can use the Qt style and add an exclamation mark (!) after the opening of a C-style comment block, as shown in this example:

```
/*!  
 * ... text ...  
 */
```

- a block of at least two C++ comment lines, where each line starts with an additional slash or an exclamation mark. Here are examples of the two cases:

```
///          //!  
/// ... text ...  ///!... text ...  
///          //!
```

# Doxygen - Tips

After the initial general file description, you need to add comments for functions and classes that you have :

```
static int width; ///< World discretized dimension in width.  
static int height; ///< World discretized dimension in height.
```

A possible alternative is `/**< .... */`

To document global variable, you can use the above syntax, which allow for placing the comment after the variable, in a more compact way. For a function, we need to specify a brief and a more detailed description, the input arguments(if any) and the return value (if any)

```
/**  
 * \brief Brief description of the function.  
 * \param min defines the lower bound of the range [default 300 s]  
 * \param max defines the upper bound of the range [default 600 s]  
  
 * \return always true as this method cannot fail.  
 *  
 * This function creates a geometry_msgs/Pose message. It fills up the response message  
 * with random value for x and y (ranging between 0 and the width, or height, respectively)  
 */
```

# Doxygen - Tips

To add documentation for a class, we need to use the special commands `\class` to specify that we are actually commenting a class:

```
/**
 * \class Test
 * \brief A test class.
 *
 * A more detailed class description.
 */

class Test
{
    public:

    /**
     * \brief An enum.
     * More detailed enum description.
     */
    enum TEnum {
        TVal1, /**< enum value TVal1. */
        TVal2, /**< enum value TVal2. */
    }

    *enumPtr, /**< enum pointer. Details. */
    enumVar; /**< enum variable. Details. */
}
```

```
/**
 * \brief A constructor.
 *
 * A more elaborate description of the constructor.
 */
    Test();

/**
 * \brief A normal member taking two arguments and returning an integer value.
 *
 * \param a an integer argument.
 * \param s a constant character pointer.
 * \return The test results
 */
    int testMe(int a, const char *s);

/**
 * \brief A public variable.
 *
 * Details.
 */
    int publicVar;
}
```

# Doxygen - Tips

Additional info for writing Doxygen documentation can be found here: [Doxygen Manual: Documenting the code](#)

What about python? Although Doxygen has been mainly conceived for Java and C++, it may also be used for documenting python code.

You can still use a formalism which makes use of docstrings:

```
""" \package my_package
    \brief Documentation for this module.
```

```

    More details.
    """
```

```
def func():
    """
    \brief Documentation for a function.
```

```

    More details.
    """
    pass
```

```
class PyClass:
    """
    \brief Documentation for a class.
```

```

    More details.
    """
```

```
    def __init__(self):
        """The constructor."""
        self._memVar = 0;
```

```
    def PyMethod(self):
        """Documentation for a method."""
        pass
```

# Doxygen - Tips

However, when using "" none of doxygen's special commands are supported.

To exploit all the potentialities of doxygen in python, it's better to comment the code using comments that start with "##". These type of comment blocks are more in line with the way documentation blocks work for the other languages supported by doxygen and this also allows the use of special commands.

```
## @package my package
# \file explore_state.py
# \brief This file contains stuff..
# \author Carmine Recchiuto
# \version 0.1
# \date 27/02/2024
#
# \details
#
# Subscribes to: <BR>
#   [None]
#
# Publishes to: <BR>
#   [None]
#
# Service : <BR>
#   [None]
#

##
# \brief Brief function description
# \param userdata is the structure containing the data shared among states.
# \return a string consisting of the state outcome
#
# This function is....
#

if we want to write the comments for a function, we should write it before that of if we want to type it in the
function, we should follow the tab rule.

def turtleCallback(msg):
    """
    my comments
    """
    rospy.logonfo()
    .
    .
```



# Doxygen - Example

To run graphic interface:  
video 1, 53:00

Once documented all your code, you can use the doxygen-gui (Doxywizard) for configuring doxygen, to finally get the output.

The configuration is quite easy. We just need to:

- Specify the working directory (the package)
- Write the project synopsis
- Specify the source code directory (again, the package, with the scan recursively option, so that the code in src and scripts will be considered)
- Specify the destination directory (e.g. docs)

We can now run doxygen to check the output.

Example: turtlesim\_controller. Notice that not all scripts and sources have been added to the documentation (this may be changed by selecting the option All-Entities in the doxygui)



# What about Python?

Docstring conventions are described at <https://www.python.org/dev/peps/pep-0257/> . Their purpose is to provide your users with a brief overview of the object. They should be kept concise enough to be easy to maintain but still be elaborate enough for new users to understand their purpose and how to use the documented object.

In all cases, the docstrings should use the triple-double quote (""" ) string format. This should be done whether the docstring is multi-lined or not. At a bare minimum, a docstring should be a quick summary of whatever is it you're describing and should be contained within a single line

*"""This is a quick summary line used as a description of the object."""*

Multi-lined docstrings are used to further elaborate on the object beyond the summary. All multi-lined docstrings have the following parts:

- A one-line summary line
- A blank line preceding the summary
- Any further elaboration for the docstring
- Another blank line

# Examples – Class Docstrings

Class docstrings should contain the following information:

- A brief summary of its purpose and behavior
- Any public methods, along with a brief description
- Any class properties (attributes)

The class constructor parameters should be documented within the `__init__` class method docstring. Individual methods should be documented using their individual docstrings. Class method docstrings should contain the following:

- A brief description of what the method is and what it's used for
- Any arguments (both required and optional) that are passed including keyword arguments
- Label any arguments that are considered optional or have a default value
- Any exceptions that are raised
- Any restrictions on when the method can be called



# Examples – Class Docstrings

```
class Animal:
    """
    A class used to represent an Animal

    ...

    Attributes
    -----
    says_str : str
        a formatted string to print out what the animal says
    name : str
        the name of the animal
    sound : str
        the sound that the animal makes
    num_legs : int
        the number of legs the animal has (default 4)

    Methods
    -----
    says(sound=None)
        Prints the animals name and what sound it makes
    """

    says_str = "A {name} says {sound}"


def __init__(self, name, sound, num_legs=4):
    """
    Parameters
    -----
    name : str
        The name of the animal
    sound : str
        The sound the animal makes
    num_legs : int, optional
        The number of legs the animal (default is 4)
    """

    self.name = name
    self.sound = sound
    self.num_legs = num_legs


def says(self, sound=None):
    """Prints what the animals name is and what sound it makes.

    If the argument `sound` isn't passed in, the default Animal
    sound is used.

    Parameters
    -----
    sound : str, optional
        The sound the animal makes (default is None)

    Raises
    -----
    NotImplementedError
        If no sound is set for the animal or passed in as a
        parameter.
    """

    if self.sound is None and sound is None:
        raise NotImplementedError("Silent Animals are not supported!")

    out_sound = self.sound if sound is None else sound
    print(self.says_str.format(name=self.name, sound=out_sound))
```

# Examples – Docstring Formats

You may have noticed that there has been specific formatting with common elements: **Arguments, Returns, and Attributes**. There are specific docstrings formats that can be used to help docstring parsers and users have a familiar and known format. The formatting used within the example before is in the NumPy/SciPy-style docstrings. Some of the most common formats are the following:

## Google Docstrings

```
"""Gets and prints the spreadsheet's header columns
```

### Args:

```
    file_loc (str): The file location of the spreadsheet  
    print_cols (bool): A flag used to print the columns to the  
    console (default is False)
```

### Returns:

```
    list: a list of strings representing the header columns  
"""
```

## reStructured Text

```
"""Gets and prints the spreadsheet's header columns
```

```
:param file_loc: The file location of the spreadsheet  
:type file_loc: str  
:param print_cols: A flag used to print the columns to the  
console  
    (default is False)  
:type print_cols: bool  
:returns: a list of strings representing the header columns  
:rtype: list  
"""
```

# Examples – Docstring Formats

## NumPy/SciPy Docstrings

```
"""Gets and prints the spreadsheet's header columns
```

### Parameters

```
-----
```

```
file_loc : str
```

```
    The file location of the spreadsheet
```

```
print_cols : bool, optional
```

```
    A flag used to print the columns to the console (default is
False)
```

### Returns

```
-----
```

```
list
```

```
    a list of strings representing the header columns
```

```
"""
```

## Epytext Example

```
"""Gets and prints the spreadsheet's header columns
```

```
@type file_loc: str
```

```
@param file_loc: The file location of the spreadsheet
```

```
@type print_cols: bool
```

```
@param print_cols: A flag used to print the columns to the
console
```

```
    (default is False)
```

```
@rtype: list
```

```
@returns: a list of strings representing the header columns
```

```
"""
```

# Sphinx

Sphinx was originally created for Python, but it has now facilities for the documentation of software projects in a range of languages.

Sphinx uses [reStructuredText](#) as its markup language

- **Output formats:** HTML (including Windows HTML Help), LaTeX (for printable PDF versions), ePub, Texinfo, manual pages, plain text
- **Extensive cross-references:** semantic markup and automatic links for functions, classes, citations, glossary terms and similar pieces of information
- **Hierarchical structure:** easy definition of a document tree, with automatic links to siblings, parents and children
- **Automatic indices:** general index as well as a language-specific module indices
- **Code handling:** automatic highlighting using the [Pygments](#) highlighter
- **Extensions:** automatic testing of code snippets, inclusion of docstrings from Python modules (API docs)
- **Contributed extensions:** more than 50 extensions [contributed by users](#) in a second repository

# Sphinx

```
$ (sudo) apt-get install python3-sphinx  
$ pip3 install breathe for using C++  
$ pip3 install sphinx-rtd-theme
```

With these three commands, you are installing sphinx, plus breathe, which is a tool for integrating doxygen documentation in sphinx, and thus comment also cpp code, and the ReadTheDocs theme, one of the most used theme for creating documentation.

Once done that, you can go in the directory where you have the code to comment, and run:

```
$ sphinx-quickstart To run:  
                     video 1, 1:10:50
```

This will start an user interface for configuring the environment.





# Sphinx conf.py

in the source directory

This procedure creates a conf.py file, which still needs to be updated. Things to be added:

Modify the code:  
video1, 1:13:00

*for C++*

```
import os
import subprocess
import sys
sys.path.insert(0, os.path.abspath('../'))
one directory up → ../ : if it doesn't run
subprocess.call('doxygen Doxyfile.in', shell=True)
show_authors=True
```

```
_# -- Project information -----
# https://www.sphinx-doc.org/en/master/usage/configuration.html#project-information
```

```
project = 'turtlebot_controller'
copyright = '2024, Carmine Recchiuto'
author = 'Carmine Recchiuto'
release = '0.1'
```

This is needed to run doxygen, generating an xml file which will be used by sphinx for creating its own documentation, and to be able to find all source code.

# Sphinx conf.py

```
extensions = [  
    'sphinx.ext.autodoc',  
    'sphinx.ext.doctest',  
    'sphinx.ext.intersphinx',  
    'sphinx.ext.todo',  
    'sphinx.ext.coverage',  
    'sphinx.ext.mathjax',  
    'sphinx.ext.ifconfig',  
    'sphinx.ext.viewcode',  
    'sphinx.ext.githubpages',  
    "sphinx.ext.napoleon",  
    'sphinx.ext.inheritance_diagram',  
    'breathe'  
]
```



Although all of them are not needed, we can just copy and paste all of them.

This is a list of the most common extensions used in sphinx. Some of them are needed (breathe, for doxygen; or autodoc for adding documentation from docstrings), others are optional (intersphinx, for adding links to other python package, or viewcode, to link the code in the documentation).

```
highlight_language = 'c++'  
source_suffix = '.rst'  
master_doc = 'index'  
html_theme = 'sphinx_rtd_theme'
```

This will set the ReadTheDocs theme, and let sphinx find the source code

*just copy and paste*

*but I don't know why*

# Sphinx conf.py

```
# -- Extension configuration -----  
  
# -- Options for intersphinx extension -----  
  
# Example configuration for intersphinx: refer to the Python standard  
library.  
  
intersphinx_mapping = {'python': ('https://docs.python.org/3', None)}  
  
# -- Options for todo extension -----  
  
# If true, `todo` and `todoList` produce output, else they produce nothing.  
todo_include_todos = True  
  
# -- Options for breathe  
  
breathe_projects = {  
    "turtlebot_controller": "../build/xml/"  
}  
breathe_default_project = "turtlebot_controller"  
breathe_default_members = ('members', 'undoc-members')
```

Finally, we set the configuration of some of the extensions used (intersphinx, todo, and breathe)

if my file is combination of python and C++ code we should use extra part in the "conf.py" file.  
the two lines in the page 25  
and the 3 last part of this page

in this case after writing the Doxyision comments, we type the doxywizard in the terminal and follow the previous steps but at the last step, we save it as the "Doxyfile.in" file instead of run that.

# Sphinx

Now, it's time to add some documentation to our code. Concerning the cpp code, we can keep using the same documentation that we have used for doxygen (indeed, sphinx uses doxygen for generating documentation for cpp code).

However, we still need to generate a configuration file.

Let's open the doxywizard, we set the same options as before, except:

- The destination directory is now `_build`
- We select XML among the output format to generate

Finally, we do not run doxygen, but we save the file (in our package folder) as `Doxyfile.in`

Pay attention to the indentation! [video, 1:21:00](#)

# Sphinx

For the python code, we insert documentation by using docstrings, and the sphinx syntax. As mentioned before, sphinx mainly uses restructured text, but we can also use Google Docstrings.

For example, we can add at the beginning of turtlebot\_controller.py:

```
"""
.. module:: turtlebot_controller
   space
   :platform: Unix
   :synopsis: Python module for the turtlebot_controller

   .. moduleauthor:: Carmine Recchiuto carmine.recchiuto@dibris.unige.it

```

This node implements a controller for the turtlesim

Subscriber:  
/turtle1/pose

```
"""
```

- \* is for bold text
- ``rospy`` [<http://wiki.ros.org/rospy/>](http://wiki.ros.org/rospy/) add a link
- ``geometry_msgs::Twist`` to add inline code
- `:mod:`scripts.exercise2`` links to other modules

And for the main function (after the def controller ()): )

```
"""
    This function initializes the ROS node and
    waits for the robot's
    *pose*, controlling the robot publishing a
    *cmd_vel*, by relying on
    the `rospy <http://wiki.ros.org/rospy/>`_
    module.

    The velocity is passed as a
    ``geometry_msgs::Twist`` message; you can find
    an updated version of this module in
    :mod:`scripts.exercise2`.
"""
```

# Sphinx

If you have some functions or variables (e.g. in the `go_to_point.py` script):

```
"""  
.. module: exercise2  
:platform unix  
:synopsis: Python module for controlling the turtlesim
```

```
.. moduleauthor:: Carmine Recchiuto
```

ROS node for controlling the robot

Subscribes to:  
  `/my_turtle/pose`

Publishes to:  
  `/my_turtle/cmd_vel`

Clients:  
  `/kill`

```
"""
```

```
pub = rospy.Publisher("my_turtle/cmd_vel", Twist)  
""" Publisher for the robot's velocity  
"""
```

```
def turtleCallback(msg):  
    """
```

Callback function to set the actual robot's velocity

Args:  
  `msg(Pose)`: the robot's position

```
    """
```

[Documenting Your Project Using Sphinx —  
an example pypi project v0.0.5 documentation  
\(pythonhosted.org\)](#)

# Sphinx

We are almost done. We just need to modify the index.rst script, which will be used by sphinx (together with conf.py and (indirectly) with Doxygen.in, to build our documentation).

The index.rst should look like this:

```
.. toctree::
    :maxdepth: 2
    :caption: Contents:
```

```
Indices
*****
```

```
* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

```
Turtlebot_controller documentation!
*****
```

This is the documentation of the turtlebot\_controller package!

```
Turtlebot_controller Module
=====
```

```
.. automodule:: scripts.turtlebot_controller
    :members:
```

```
Exercise2 Module
=====
```

```
.. automodule:: scripts.exercise2
    :members:
```

```
Turtlebot_controller Module in cpp
=====
```

```
.. doxygenfile:: turtlebot_controller.cpp
    :project: turtlebot_controller
```

```
:imported-members:
:members:
:undoc-members:
:show-inheritance:
```

may be added as well

We do not need to  
specify all functions  
of the modules

After modifying the "index.rst" file, use the "make html" order in the terminal to create the html file or save changes.

add to  
"index.rst"

Doxygen  
documentation

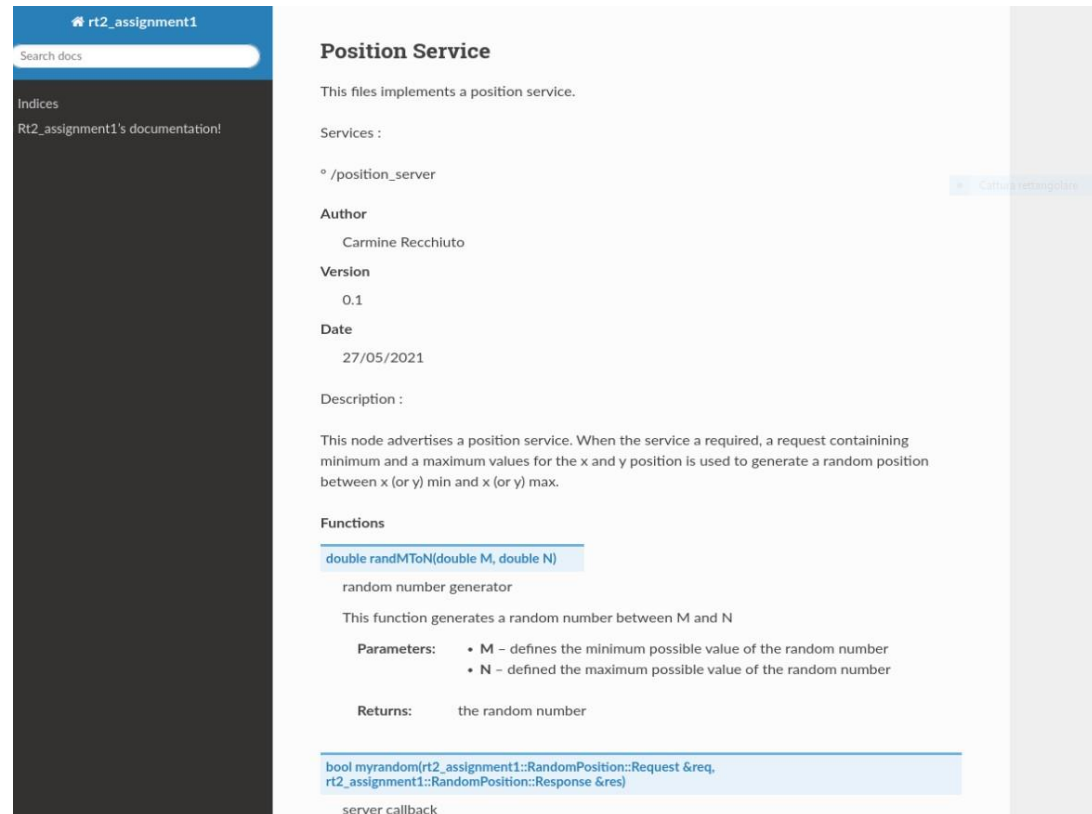
→ when our code is combination of  
python and C++ code

Carmine Tommaso Recchiuto

# Sphinx

Great! We can run *make html* in the terminal, to see (hopefully) our beautiful documentation in sphinx. In the terminal, you will see at first output related to doxygen, and then to sphinx.

The html pages will be in `_build/html`



The screenshot displays a Sphinx-generated HTML page for a project named 'rt2\_assignment1'. The left sidebar contains a search bar and a link to 'Indices'. The main content area is titled 'Position Service' and includes a description, a list of services, author information, version, date, and a detailed description of the service. It also lists functions, with 'double randMTon(double M, double N)' highlighted, showing its parameters and return value. At the bottom, a code snippet for a server callback is shown.

**Position Service**

This files implements a position service.

Services :

- ° /position\_server

**Author**

Carmin Recchiuto

**Version**

0.1

**Date**

27/05/2021

**Description :**

This node advertises a position service. When the service a required, a request containing minimum and a maximum values for the x and y position is used to generate a random position between x (or y) min and x (or y) max.

**Functions**

**double randMTon(double M, double N)**

random number generator

This function generates a random number between M and N

**Parameters:**

- **M** – defines the minimum possible value of the random number
- **N** – defined the maximum possible value of the random number

**Returns:** the random number

**bool myrandom(rt2\_assignment1::RandomPosition::Request &req, rt2\_assignment1::RandomPosition::Response &res)**

server callback



# GitHub

Finally, it could be nice to put our documentation online, so that it could be visualized by people using our repository.

You can still use github for that.

- ✓ Create a folder docs, containing your documentation
- ✓ In case of sphinx documentation, add an empty file, in the docs folder, named .nojekyll (this is needed for using the sphinx layout)
- ✓ Finally, go to Settings -> Pages, and activate an url which may be used to visualize the documentation.

In the case of the package rt2\_assignment1, I have created two branches (doxygen and sphinx), adding the related documentation in the docs folder. Then, I have associated my gh-page to the doxygen documentation:

[Indices — rt2\\_assignment1 0.1 documentation \(carmined8.github.io\)](https://carmined8.github.io/rt2_assignment1/0.1/)

# Assignment

Add the documentation to the github repository of your **second** Research Track I assignment.

Depending on the programming language used for the assignment, you can choose if using Sphinx or Doxygen.