Neurapy

Release v4.17.0-alpha.55

Neura Robotics GmbH

Jun 04, 2024

NEURAPY

CHAPTER

ONE

NEURAPY

NeuraPy is an easy-to-use, extendable, robot-agnostic Python API for Neura's control software, designed for application developers.

1.1 Robot Class

NeuraPy provides a *Robot* class that allows access to all the functionalities and static parameters of the robot. These can be accessed as methods and attributes of an instantiation of the *Robot* class.

1.2 Parameters

- *robot_name*
- *dof*
- *payload*
- *controller_ip*
- *version*

1.3 Functionalities

S.No.	Functionality	Action
1.	compute_forward_kinematics	Compute forward kinematics(end-effector's cartesian pose) for the given joint configuration.
2.	compute_inverse_kinematics	Compute inverse kinematics(joint configuration) for the given end-effector's cartesian pose.
3.	create_tool	Creates a new tool in the robot's database using the provided tool data.
4.	disable_collision_detection	Disable collision detection on the robot.

continues on next page

Table 1 – continued from previous page

S.No.	Functionality	Action
5.	disable_reflex	Disable reflex after collision.
6.	enable_collision_detection	Enable collision detection on the robot.
7.	enable_reflex	Enable reflex after collision.
8.	encoder2rad	Convert encoder impulses to joint angles in radians.
9.	execute_program	To run a program created from teach pendant's tree program creator.
10.	executor	Execute already planned motion IDs.
11.	get_analog_input	Get the value of an analog input.
12.	get_analog_output	Get the value of an analog output.
13.	get_current_cartesian_pose	Get the current Cartesian pose of the robot's TCP.
14.	get_current_cartesian_pose_with_timestamp	Get the current Cartesian pose of the robot's TCP along with UTC timestamp.
15.	get_current_joint_angles	Get current joint angles of the robot.
16.	get_current_joint_angles_with_timestamp	Get the current joint angles of the robot along with UTC timestamp.
17.	get_current_joint_torques	Get the current joint torques of the robot.
18.	get_current_joint_torques_with_timestamp	Get the current joint torques of the robot along with UTC timestamp.
19.	get_current_joint_velocities	Get current joint velocities of the robot.
20.	get_current_joint_velocities_with_timestamp	Get the current joint velocities of the robot along with UTC timestamp.

continues on next page

Table 1 – continued from previous page

S.No.	Functionality	Action
21.	get_current_load_side_encoder_values	Get the current load side encoder ticks of the robot.
22.	get_current_load_side_encoder_values_with_timestamp	Get the current load side encoder ticks of the robot along with UTC timestamp.
23.	get_current_motor_side_encoder_values	Get the current motor side encoder ticks of the robot.
24.	get_current_motor_side_encoder_values_with_timestamp	Get the current motor side encoder ticks of the robot along with UTC timestamp.
25.	get_current_tool_cogs	Get the current tool center of gravity (COG) in XYZ directions (measured from the robot's flange frame/tool mounting point).
26.	get_current_tool_inertias	Get the current tool inertias Ixx, Iyy, Izz, Ixy, Ixz, and Iyz.
27.	get_current_tool_mass	Get the current tool mass.
28.	get_current_tool_properties	Get the current tool properties.
29.	get_current_tool_rpy_offsets	Get the current tool roll, pitch, and yaw (RPY) offsets in radians.
30.	get_current_tool_translation_offsets	Get the current tool translation offsets in XYZ directions.
31.	get_diagnostics	Method to query current robot diagnostics.
32.	get_digital_input	Get the value of a digital input. Please use the get_io_configuration function to get the number of available IOs.
33.	get_digital_output	Get the value of a digital output. Please use the get_io_configuration function to get the number of available IOs.
34.	get_doc	Get the description and sample usage of the specified function.
35.	get_encoder_offsets	To retrieve the encoder offsets of a robot.
36.	get_errors	Query the list of errors present on the robot.

continues on next page

Table 1 – continued from previous page

S.No.	Functionality	Action
37.	get_gravity_vector	To retrieve the set gravity vector value from the database.
38.	get_io_configuration	Get the number of available IOs.
39.	get_joint_acceleration	Get the current acceleration value for joint motion.
40.	get_joint_speed	Get the current speed value for joint motion.
41.	get_linear_acceleration	Get the current acceleration value for linear motion.
42.	get_linear_speed	Get the current speed value for linear motion.
43.	get_mode	Query the current robot mode (Teach/Automatic/SemiAutomatic).
44.	get_override	Get the current override value.
45.	get_point	Retrieves a point from the database based on the given name.
46.	get_reference_frame	Retrieve the coordinates and orientations of a user-defined reference frame.
47.	get_reference_frame_with_offset	Retrieve a reference frame with an applied offset to its position.
48.	get_sim_or_real	To check whethre robot is in real or simulation mode.
49.	get_tcp_pose	To get the current TCP pose in XYZRPY.
50.	get_tcp_pose_with_timestamp	Get the current robot's TCP pose along with UTC timestamp.
51.	get_tool_analog_input	Get the value of a tool analog input. Please use the get_io_configuration function to get the number of available IOs.
52.	get_tool_digital_input	Get the value of a tool digital input. Please use the get_io_configuration function to get the number of available IOs.

continues on next page

Table 1 – continued from previous page

S.No.	Functionality	Action
53.	get_tool_digital_output	Get the value of a tool digital output. Please use the get_io_configuration function to get the number of available IOs.
54.	get_tool_flange_pose	Get the current tool flange pose in XYZRPY format.
55.	get_tools	Retrieves a list of tools from the robot's database.
56.	get_warnings	Query the list of warnings present on the robot.
57.	get_zerog_status	Method to check the status of the free drive mode.
58.	grasp	To grasp the object with the attached gripper.
59.	gripper	Method to control the gripper attached to the robot.
60.	ik_fk	Compute forward/inverse kinematics for a given configuration.
61.	io	Access and manipulate (output) IO values on the robot.
62.	is_collision_enabled	Check if reflex after collision is enabled for the robot.
63.	is_free_drive_mode_enabled	Get the status of the robot free drive mode.
64.	is_reflex_enabled	Check if reflex after collision is enabled for the robot.
65.	is_robot_in_collision	Check if the robot is currently in collision.
66.	is_robot_in_automatic_mode	Method to check whether the robot is in automatic mode or not.
67.	is_robot_in_semi_automatic_mode	Method to check whether the robot is in semi-automatic mode or not.
68.	is_robot_in_teach_mode	Method to check whether the robot is in teach mode or not.

continues on next page

Table 1 – continued from previous page

S.No.	Functionality	Action
69.	is_robot_in_simulation	Method to check whether the robot is currently in simulation mode.
70.	jog	Method to jog programmatically. This needs to be used in conjunction with turn_on_jog, turn_off_jog methods.
71.	list_methods	To list the available functions in the API.
72.	motion_status	Query the motion status of the robot.
73.	move_circular	To move the robot in a circular path across the given poses.
74.	move_composite	To move the robot in given linear and circular path combinations.
75.	move_joint	To move the robot to the specified joint configuration in joint space.
76.	move_linear	To move the robot in a linear path across the given poses.
77.	move_linear_from_current_position	To move the robot in a linear path across the given poses from the current position.
78.	move_trajectory	Move the robot with a given joint trajectory.
79.	notify_error	To notify an error message to the teach pendant.
80.	notify_warning	To notify a warning message to the teach pendant.
81.	override	To adjust and query the override value.
82.	pause	To pause the robot's motion.
83.	plan_joint_trajectory	To plan the given joint trajectory.
84.	plan_move_circular	To plan a circular path across the given poses.

continues on next page

Table 1 – continued from previous page

S.No.	Functionality	Action
85.	plan_move_composite	To plan move composite across the given poses.
86.	plan_move_joint	To plan a joint space motion across the given configurations.
87.	plan_move_linear	To plan a move linear path across the given poses.
88.	plan_move_recorded_path	To plan the given pre-recorded path.
89.	power	To power on/off the robot.
90.	power_off	To power off the robot.
91.	power_on	To power on the robot.
92.	program_status	Query the program status of the robot.
93.	quaternion_to_rpy	Convert a quaternion representation to roll-pitch-yaw (RPY) angles.
94.	read_safeio	Method to read the values of configurable/safe Input/Output.
95.	record_path	To move the robot in a pre-recorded path.
96.	release	To release the grasped object.
97.	reset_collision	Reset the collision state of the robot.
98.	reset_control	To restart the control software running on the robot. Equivalent to the reset control option from the teach pendant.
99.	reset_errors	Method to reset the errors on the robot.
100.	reset_warnings	Method to reset/clear the warnings on the robot.

continues on next page

Table 1 – continued from previous page

S.No.	Functionality	Action
101.	robot_status	To query the robot status.
102.	rpy_to_quaternion	Convert roll-pitch-yaw (RPY) angles to a quaternion representation.
103.	set_analog_output	Set analog outputs of the control box.
104.	set_digital_output	Set digital outputs of the control box.
105.	set_encoder_offsets	To set the joint encoder offsets of the robot.
106.	set_gravity_vector	To set the gravity vector of the robot.
107.	set_joint_acceleration	Set the acceleration value for joint motion.
108.	set_joint_speed	Set the speed value for joint motion.
109.	set_linear_acceleration	Set the acceleration value for linear motion.
110.	set_linear_speed	Set the speed value for linear motion.
111.	set_mode	To change the mode of the robot (Teach/SemiAutomatic/Automatic).
112.	set_opcua_msg	Send a message to the OPC UA server running on the control box.
113.	set_override	Set the override value.
114.	stop	To stop the robot's motion and to terminate the script execution.
115.	switch_to_automatic_mode	Switch the robot to Automatic mode.
116.	switch_to_real	Switch the robot's running context to real mode.

continues on next page

Table 1 – continued from previous page

S.No.	Functionality	Action
117.	switch_to_semi_automatic_mode	Switch the robot to Semi-Automatic mode.
118.	switch_to_simulation	Switch the robot's running context to simulation mode.
119.	switch_to_teach_mode	Switch the robot to Teach mode.
120.	turn_off_free_drive_mode	To turn off free drive mode.
121.	turn_off_jog	Method to disable jogging programmatically. This needs to be used in conjunction with jog, turn_on_jog methods.
122.	turn_on_free_drive_mode	To turn on free drive mode.
123.	turn_on_jog	Method to enable jogging programmatically. This needs to be used in conjunction with jog, turn_off_jog methods.
124.	unpause	To unpause the robot's motion.
125.	wait	Wait for a specified signal.
126.	wait_for_analog_input	Wait for an analog input signal to match the expected value.
127.	wait_for_digital_input	Wait for a digital input signal to match the expected value.
128.	wait_for_digital_input_timer_off_delay	Wait for a given delay and returns after the given digital input signal reaches low.
129.	wait_for_digital_input_timer_on_delay	Wait for a given delay and returns after the given digital input signal reaches high.
130.	wait_for_tool_analog_input	Wait for a tool-specific analog input signal to match the expected value.
131.	wait_for_tool_digital_input	Wait for a tool-specific digital input signal to match the expected value.
132.	wait_for_tool_digital_input_timer_off_delay	Wait for a given delay and returns after the given tool digital input signal reaches low.

continues on next page

Table 1 – continued from previous page

S.No.	Functionality	Action
133.	wait_for_tool_digital_input_timer_on_delay	Wait for a given delay and returns after the given tool digital input signal reaches high.
134.	zero_g	Toggle the freedrive/Gravity compensation mode.

CHAPTER

TWO

OFFLINE SIMULATION SETUP

Offline simulation setup is used for offline programming and simulation of robot programs.

The offline simulation software is provided as a virtual machine image and has been tested exclusively on VirtualBox 7.0 and above.

Please note that using this virtual image on different virtualization software may lead to unexpected issues.

2.1 System Requirements

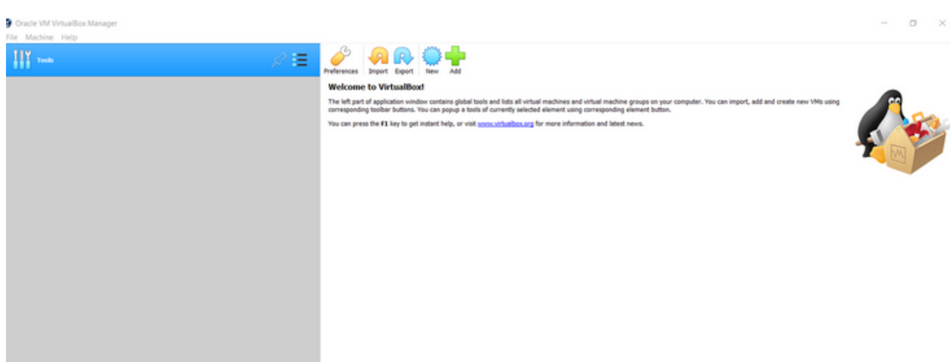
To ensure optimal performance, the following system requirements are recommended:

- **Hard Drive Space:** A minimum of 30GB of free space on your hard drive is required.
- **RAM:** You should have at least 10GB of RAM to ensure smooth performance.
- **Processor:** An 8-12 core processor is recommended to handle the virtual machine efficiently.

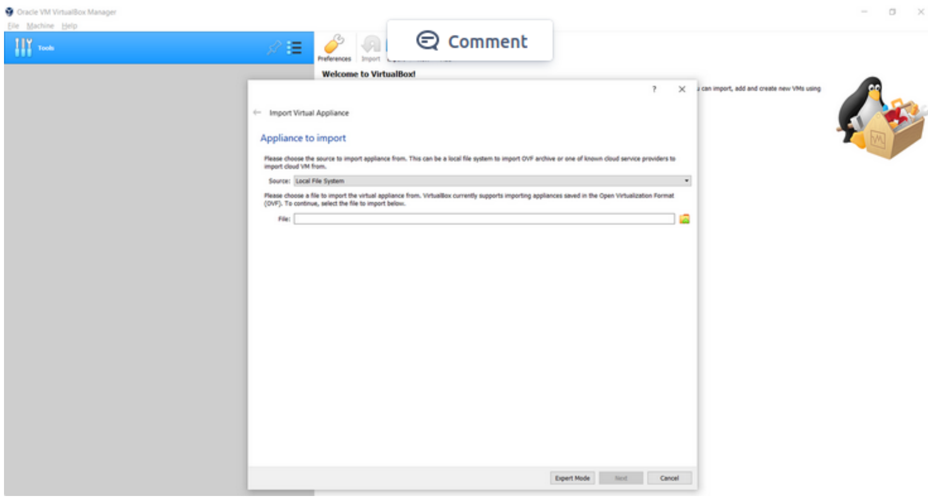
Meeting these system requirements will help ensure that your virtual machine runs smoothly without performance issues.

2.2 Installation Guidelines for Windows

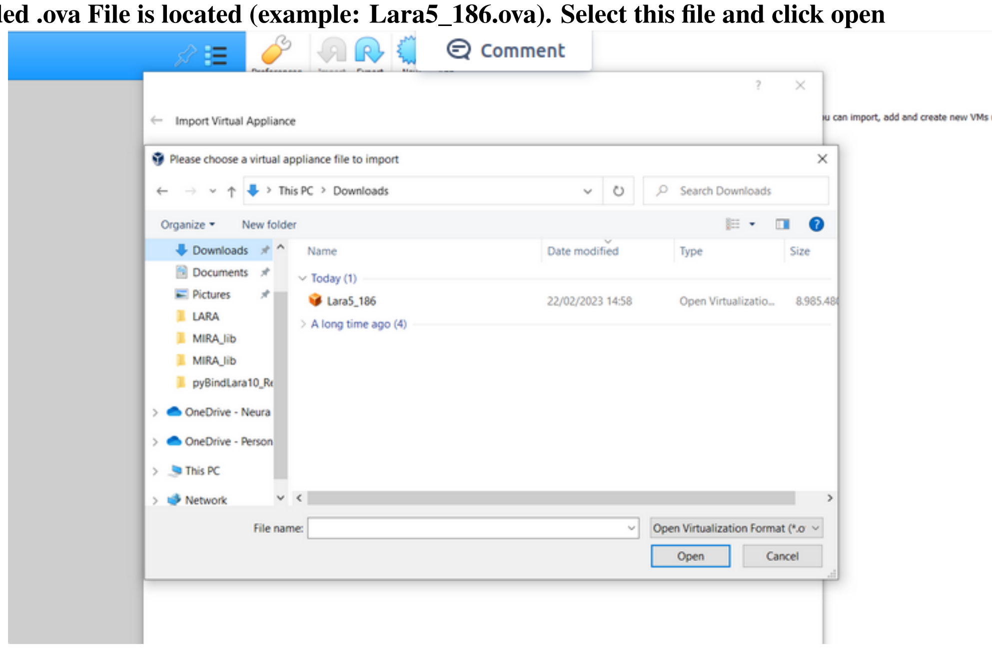
1. Download the Windows OS compatible VirtualBox from virtualbox.org and install it.
2. Download the offline simulation software VM image (LARA5).
3. Open the VirtualBox GUI by navigating to the start menu. You should see a window similar to the one below



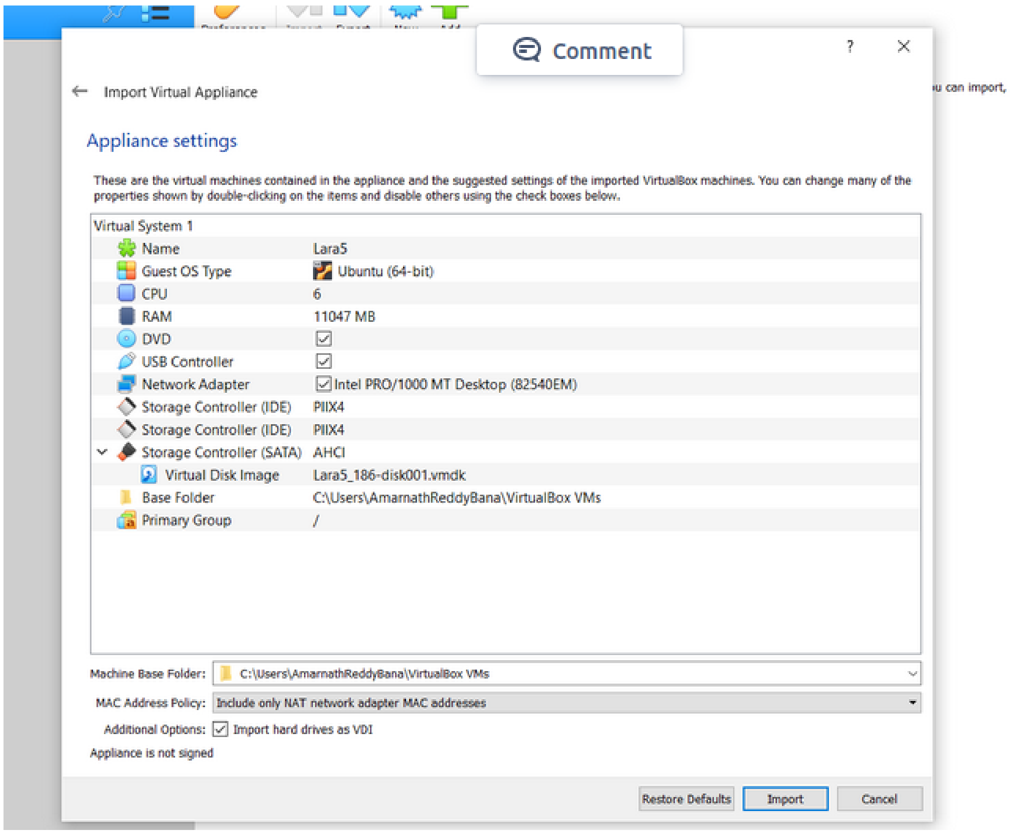
4. Click on the Import Icon. This action will open a second window for importing the Virtual Appliance



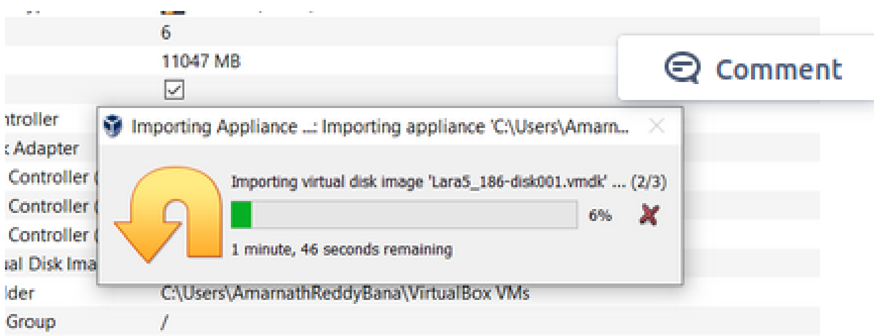
5. Please click on the folder icon beside the File. It will pop up a browser window. Specify where the downloaded .ova File is located (example: Lara5_186.ova). Select this file and click open



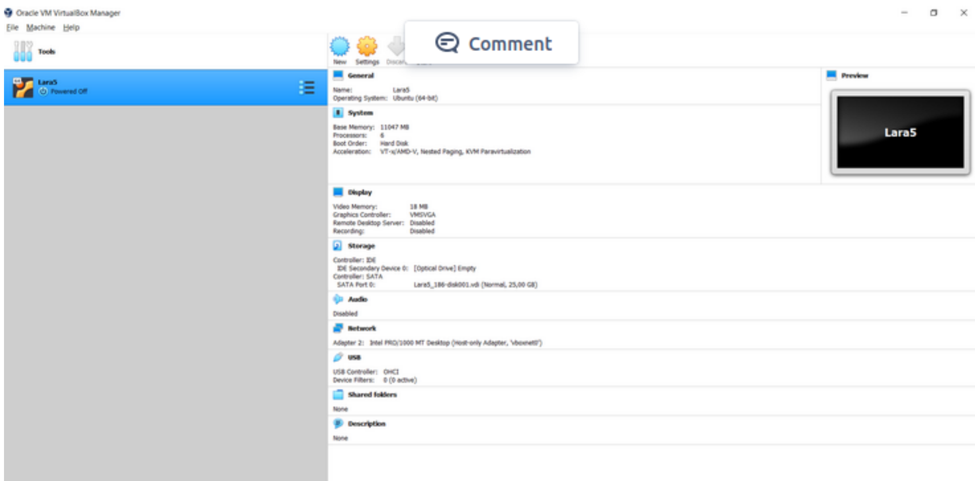
6. Please click next, and the configuration settings will pop up. It is recommended to keep the same configuration settings. Please press Restore Defaults before Import



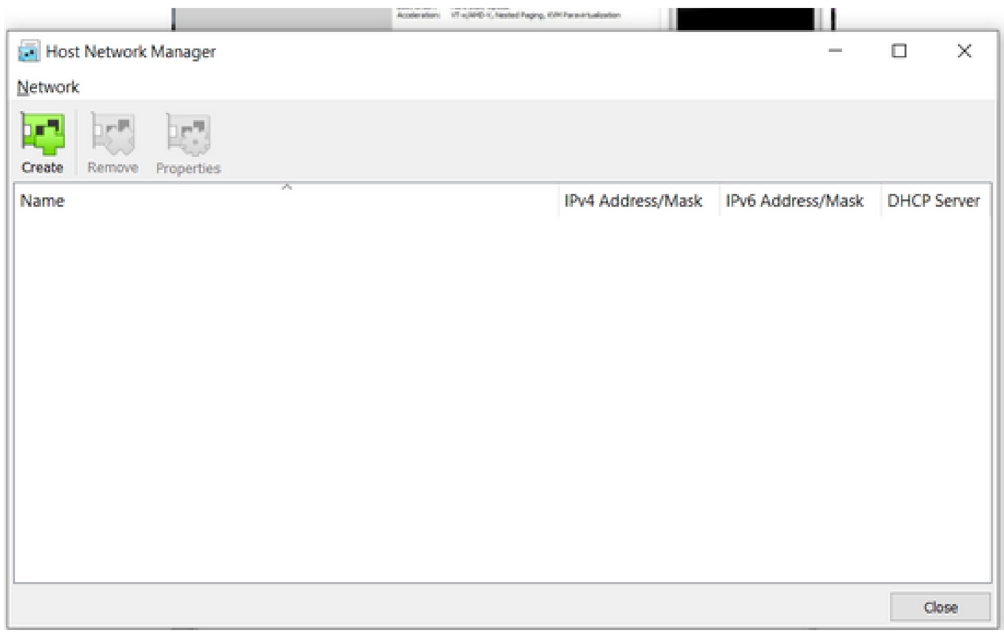
7. Click on Import, and an acknowledgment window with elapsed time will pop up. Please let it be finished



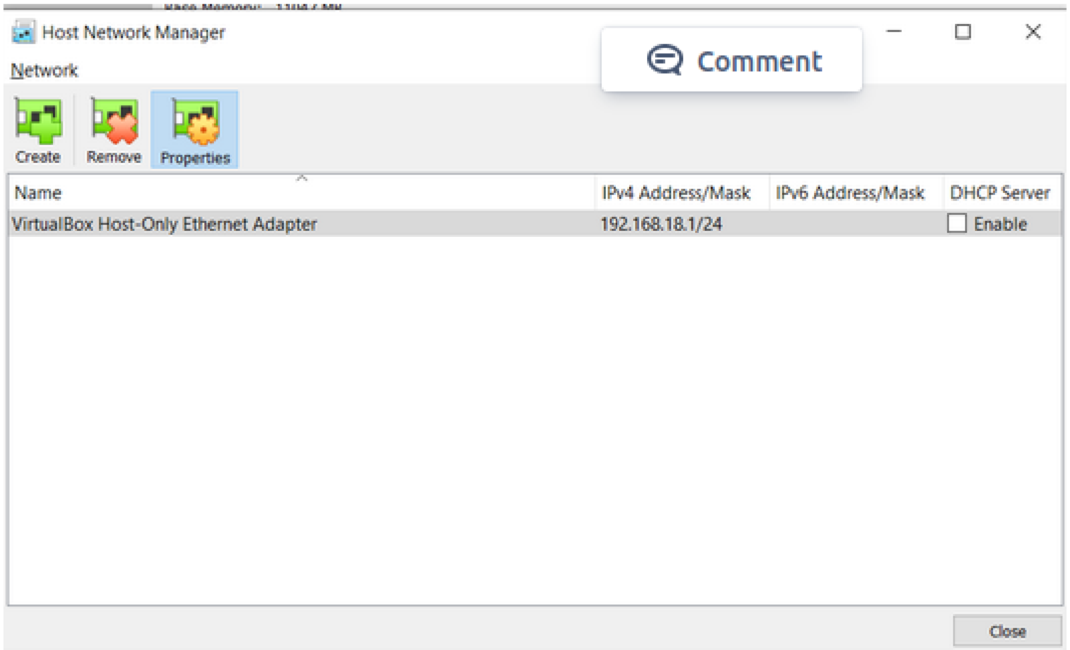
8. After this process is finished, the VirtualBox GUI should look like as below



9. **To configure the network adapter go to File > Host Network Manager. A window will pop up. Please click on Create**

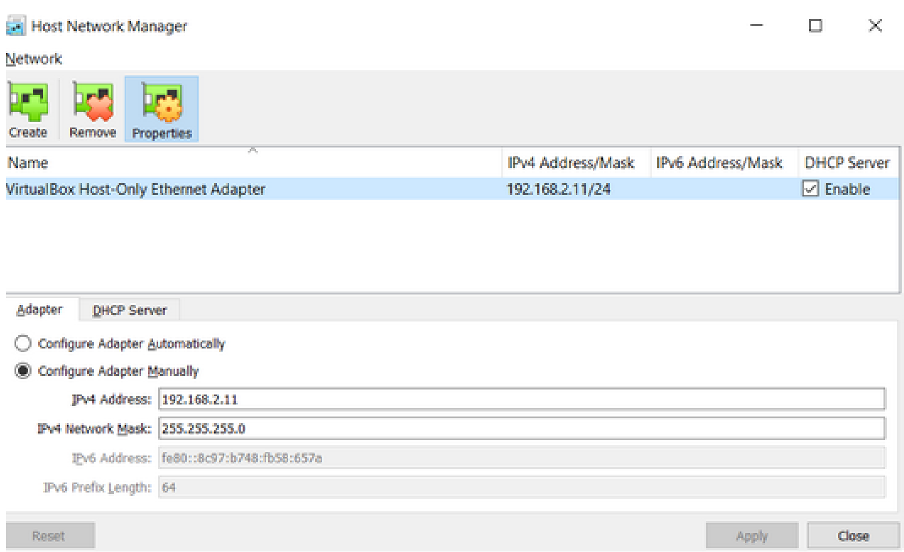


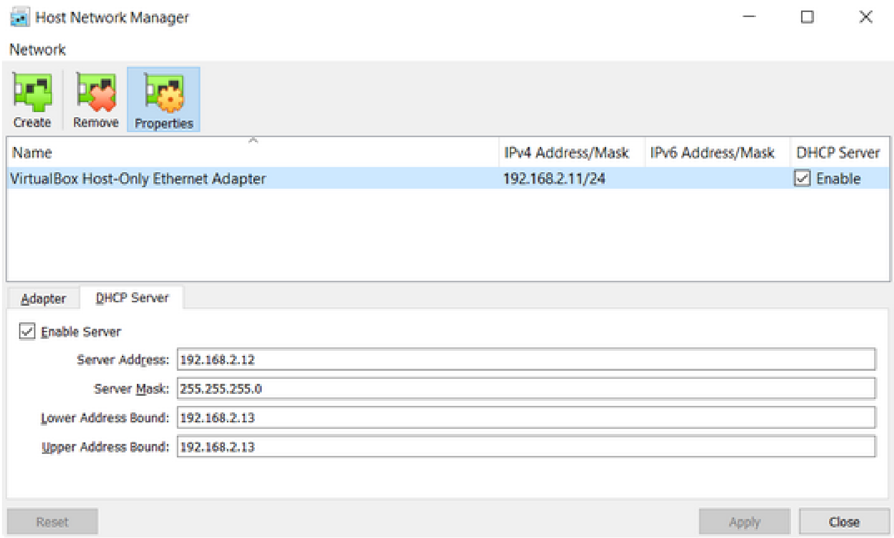
10. Once the Adapter is created, it should show VirtualBox Host-Only Ethernet Adapter. Please enable the DHCP Server by clicking on Enable checkbox under DHCP Server at the top right corner. Next click on the Properties please



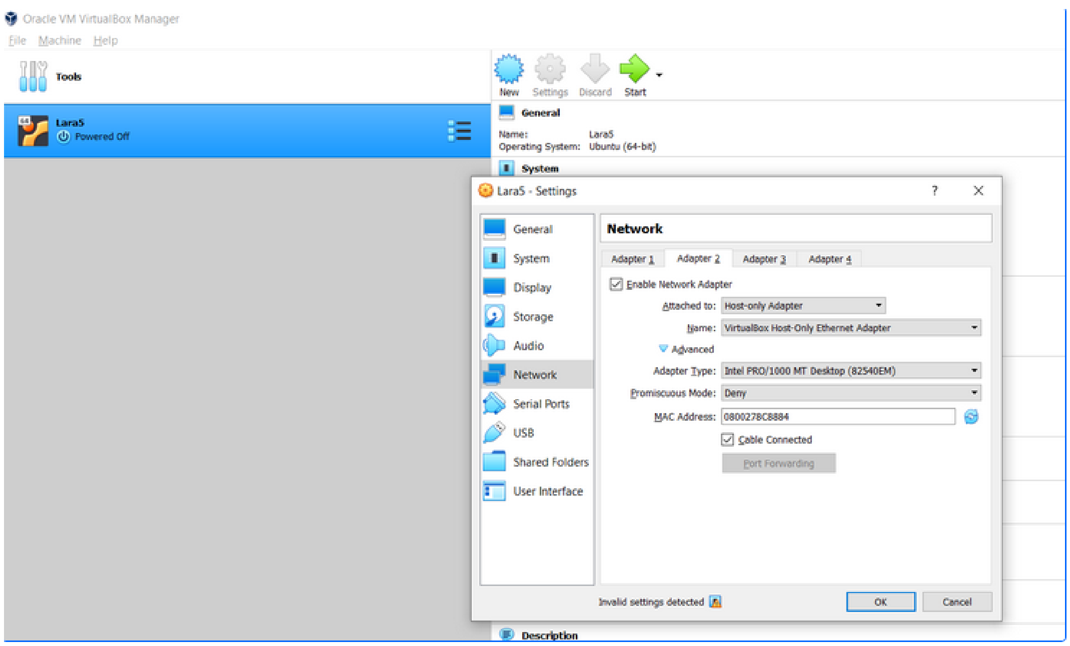
11. Choose "Configure Adapter Manually" option and adjust the fields as follows:
    - **IPv4 Address:** 192.168.2.11
    - **IPv4 Network Mask:** 255.255.255.0

    In the DHCP Server settings:
    - Click on "Enable Server".
    - Set the server address to 192.168.2.12.
    - Set the server mask to 255.255.255.0.
    - Set the lower address bound to 192.168.2.13.
    - Set the upper address bound to 192.168.2.13.

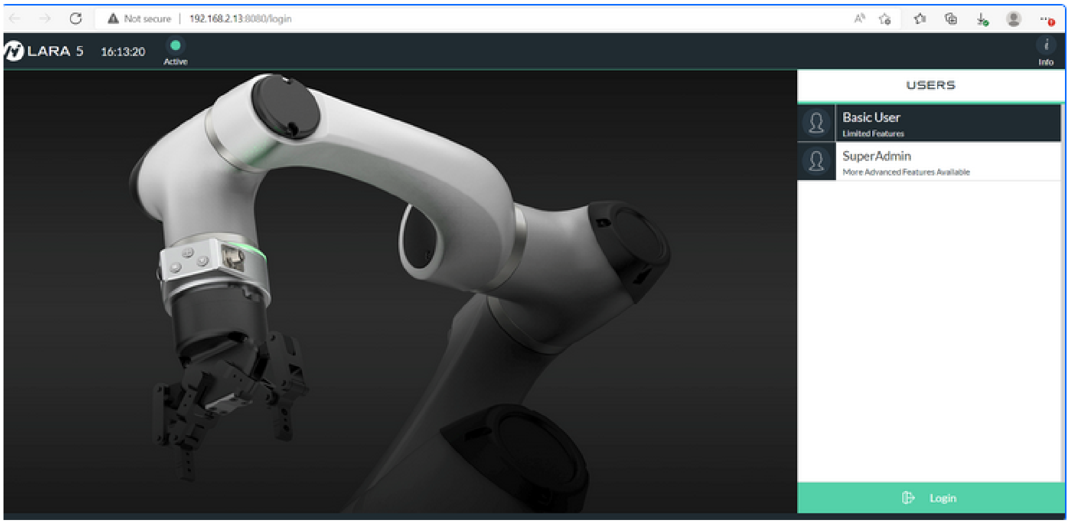




12. Click on *Settings* -> *Network* -> *Adapter 1*. If *Enable Network Adapter* is checked, uncheck it first.Click on *Adapter 2* and check *Enable Network Adapter*. From the drop-down menu of *Attached to*, select *Host-only Adapter*. In the *Name* field, choose *VirtualBox Host-Only Ethernet Adapter*.The settings should appear as follows

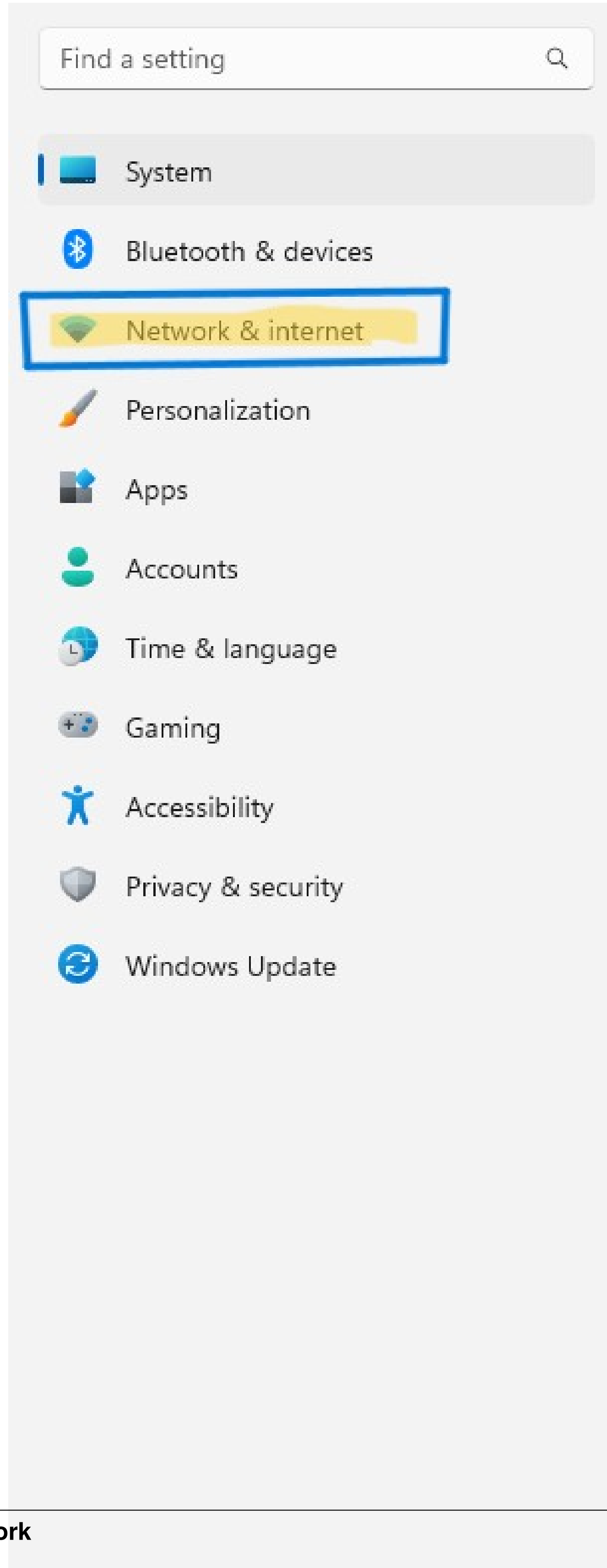


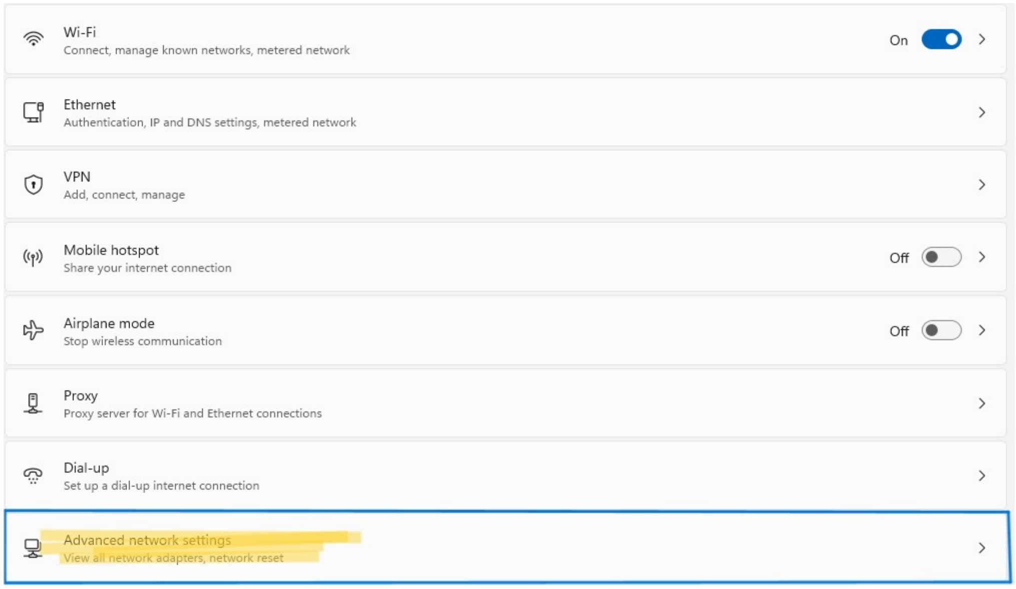
13. The configuration of the offline simulation VM image is finished. Please start Lara5 and for 30 seconds. To access the GUI, please open chrome and enter http://192.168.2.13:8080/login. If every process was successful, the address should load the GUI interface


Not secure | 192.168.2.13:8080/login
LARA 5
16:13:20
Active
Info
USERS
Basic User
Limited Features
SuperAdmin
More Advanced Features Available
Login

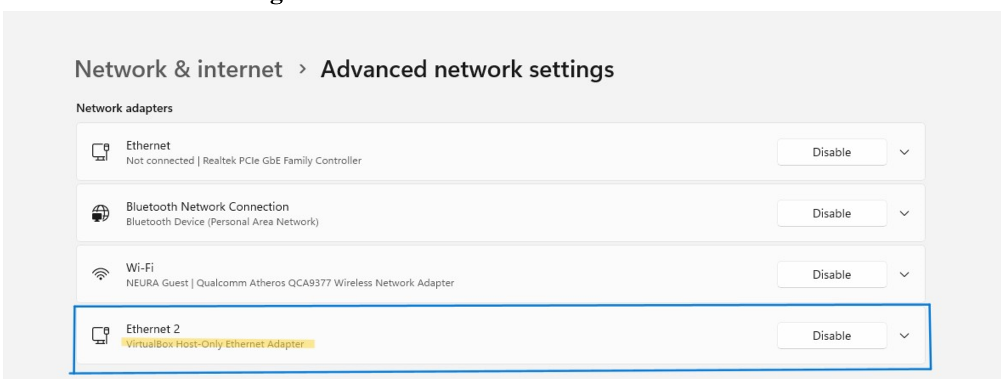
2.3 Troubleshooting Network

1. **On your Windows PC Click on Settings → Network & Internet → Advanced network settings**





2. **Click on Advanced network settings**



3. Please verify the status of the Ethernet Adapter for VirtualBox in the Advanced network settings. If it is currently enabled, simply click the "Enable" button to confirm. If it's already enabled by default, click the "Enable" button, wait for it to disable, pause for 3 seconds, and then re-enable it.

2.4 Installation Guidelines for Ubuntu

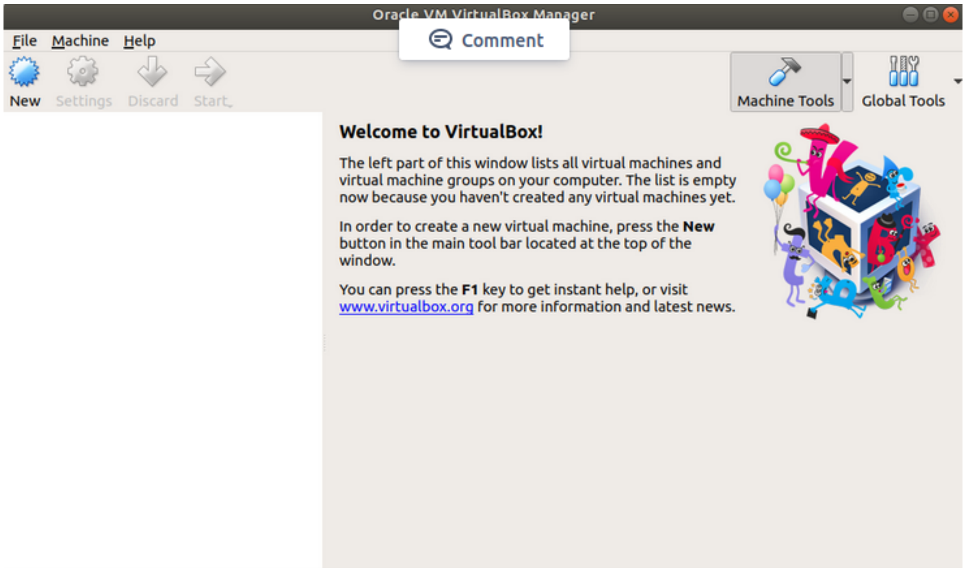
1. Install VirtualBox on Ubuntu using the following commands from the apt repositories:

```
sudo dpkg -i ~/"path"/virtualbox*deb
```

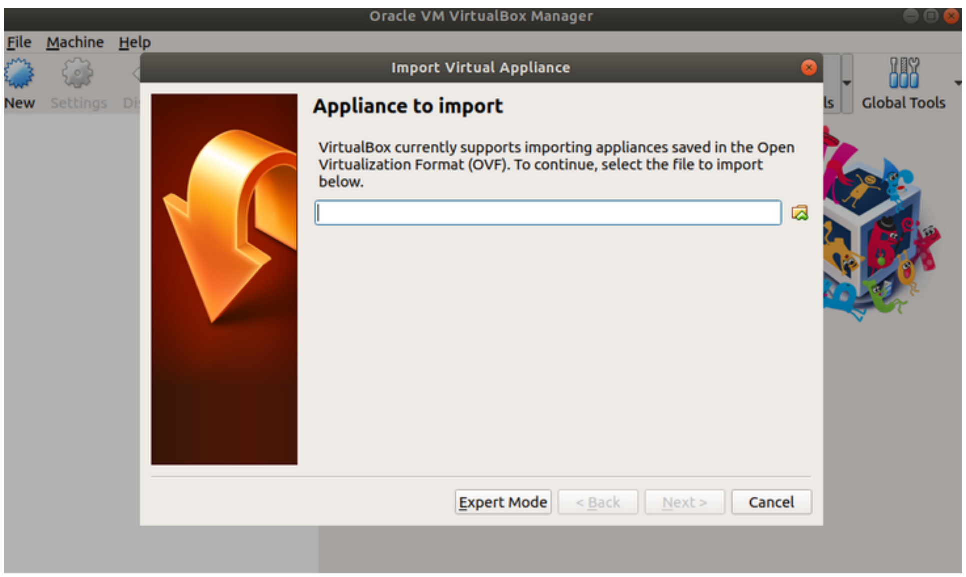
2. Download the offline simulation software VM image (LARA5).
3. To enable static IP allotment to VirtualBox in a custom subnet, execute the following commands

```
sudo mkdir /etc/vbox
sudo touch /etc/vbox/networks.conf
echo "* 10.0.0.0/8 192.168.0.0/16" | sudo tee -a /etc/vbox/networks.conf
```

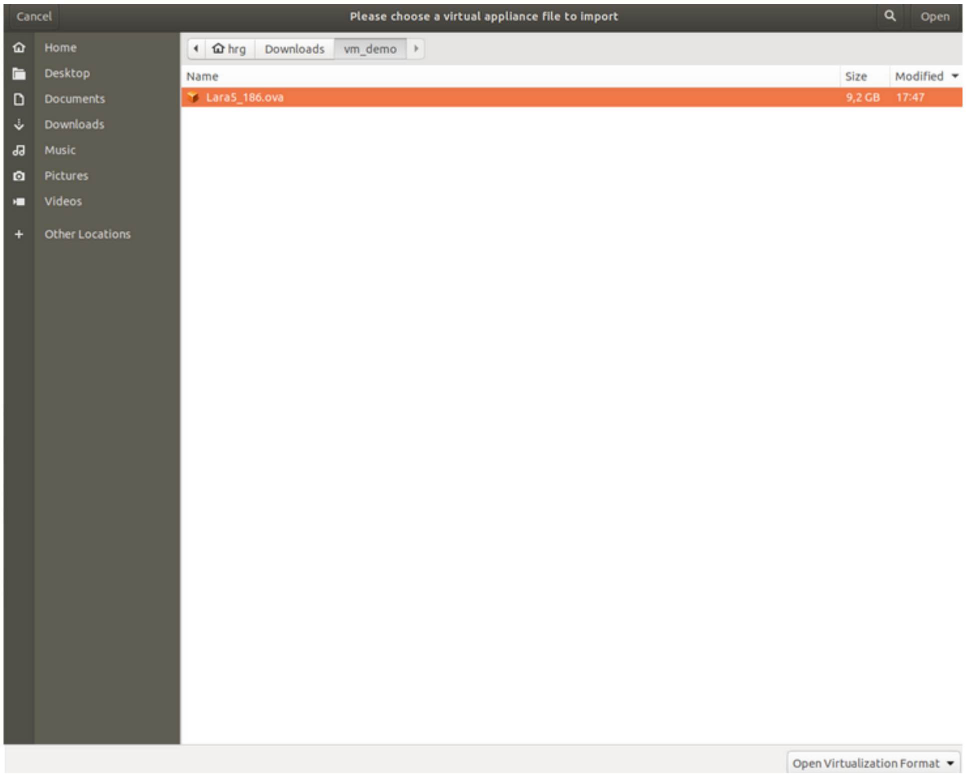
4. Please open virtualbox from the terminal. A window as below should pop up.



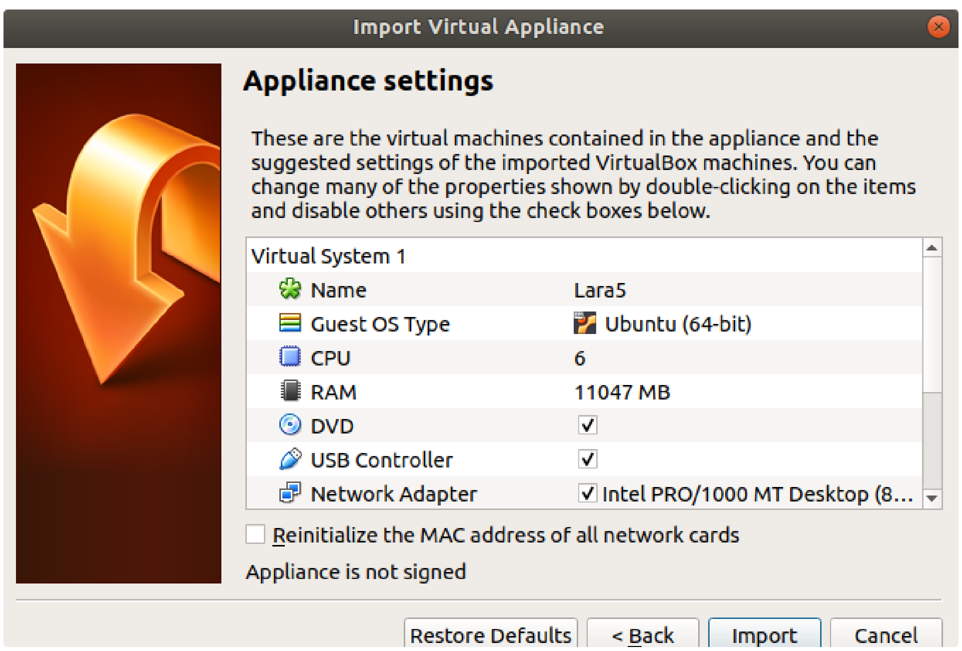
5. Click on the Import virtual appliance from file tab and, a second window to import Virtual Appliance will appear



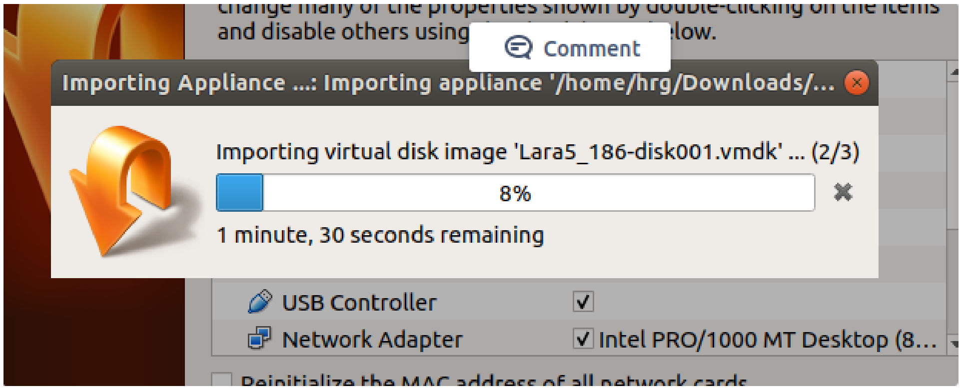
6. Please click on the folder icon beside the File. It will pop up a browser window. Specify where the downloaded .ova File is located (example: Lara5_186.ova). Select this file and click open



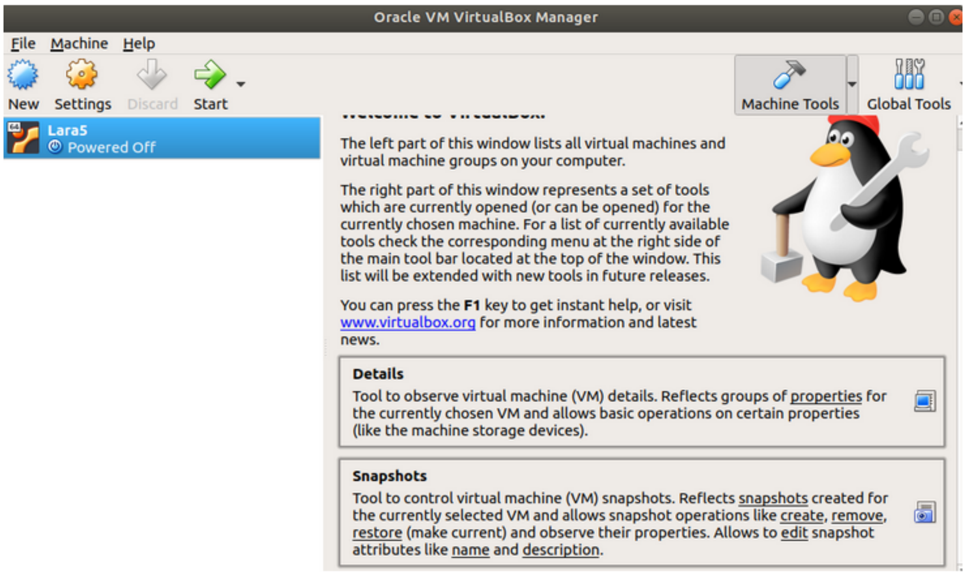
7. Please click next, and the configuration settings will pop up. It is recommended to keep the same configuration settings. Please press Restore Defaults before Import.



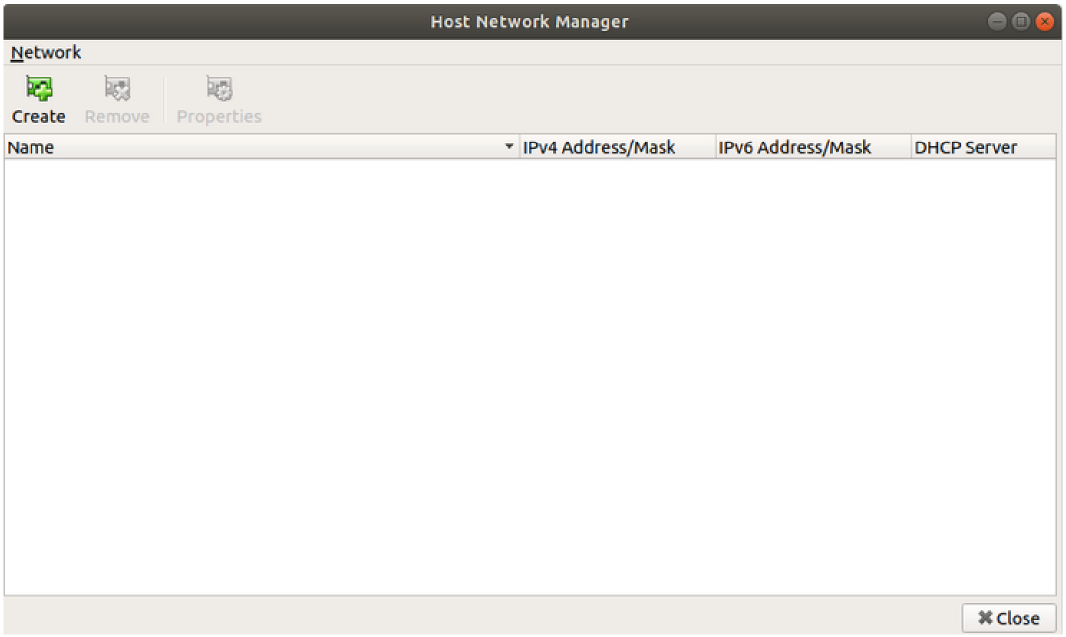
8. Click on Import, and an acknowledgment window with elapsed time will pop up. Please let it be finished



9. After this process is finished, the VirtualBox GUI should look like as below.



10. To configure the network adapter go to File > Host Network Manager. A window will pop up. Please click on Create

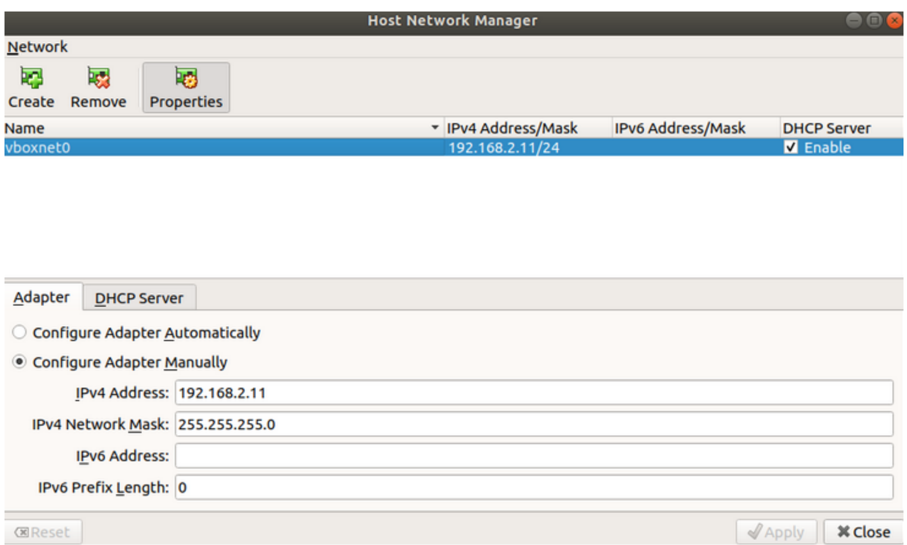


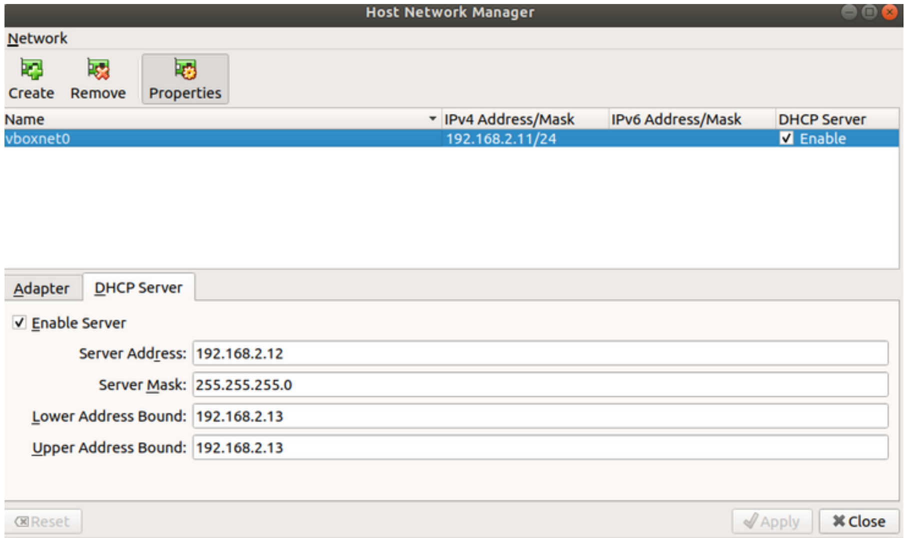
11. Once the Adapter is created, it should show vboxnet0. Please enable the DHCP Server by clicking on Enable checkbox under DHCP Server at the top right corner. Next click on the Properties please.

12. Choose "Configure Adapter Manually" option and adjust the fields as follows:

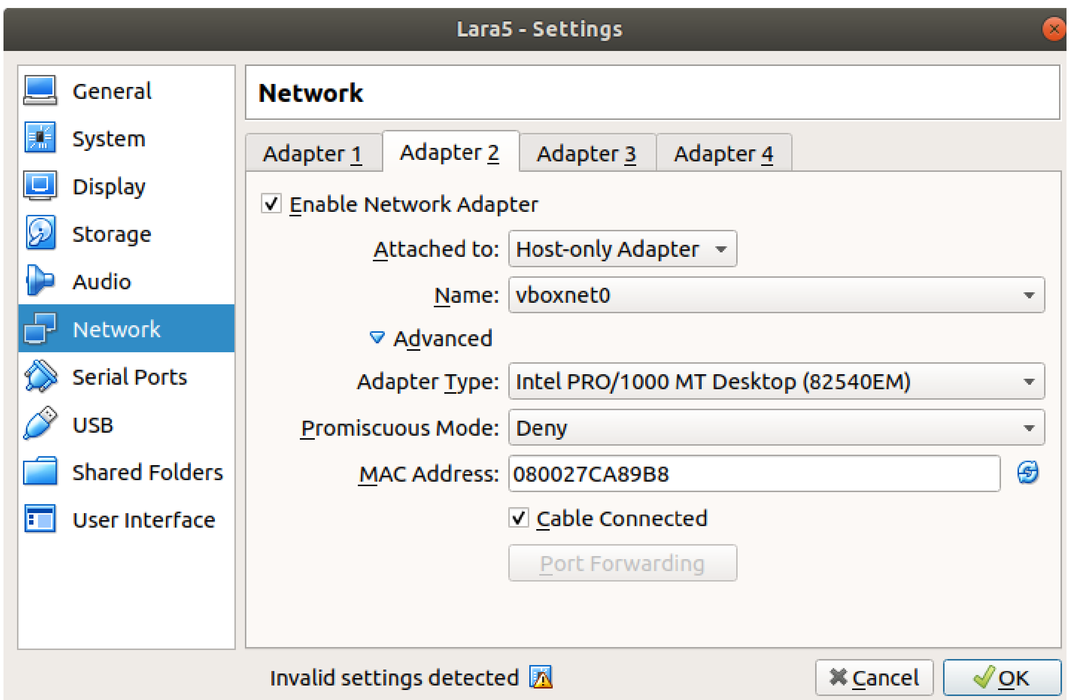
    - **IPv4 Address:** 192.168.2.11
    - **IPv4 Network Mask:** 255.255.255.0

    In the DHCP Server settings: - Set the server address to 192.168.2.12. - Set the server mask to 255.255.255.0. - Set the lower address bound to 192.168.2.13. - Set the upper address bound to 192.168.2.13.

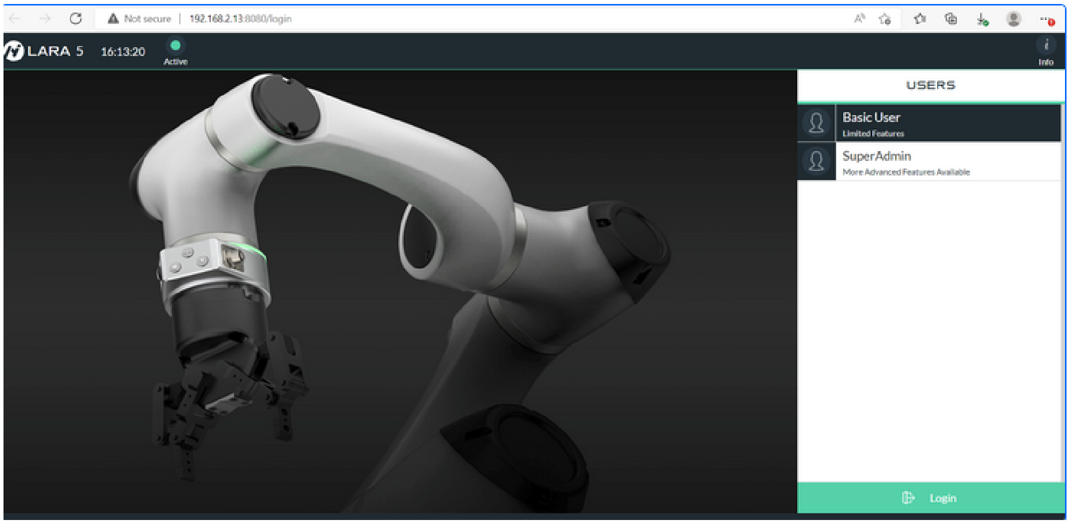




13. Click on Settings -> Network -> Adapter 1 , If Enable Network Adapter is checked, uncheck it first. Click on Adapter 2 check Enable Network Adapter. From the drop-down menu of Attached to: select Host-only Adapter. On the Name filed please select, VirtualBox Host-Only Ethernet Adapter. It should look like



14. The configuration of the Lara5 is finished. Please start Ubuntu Lara5 and for 30 seconds. To access the GUI, please open chrome and enter http://192.168.2.13:8080/login. If every process was successful, the address should load the GUI interface.


Not secure | 192.168.2.13:8080/login
LARA 5
16:13:20
Active
Info
USERS
Basic User
Limited Features
SuperAdmin
More Advanced Features Available
Login

CHAPTER

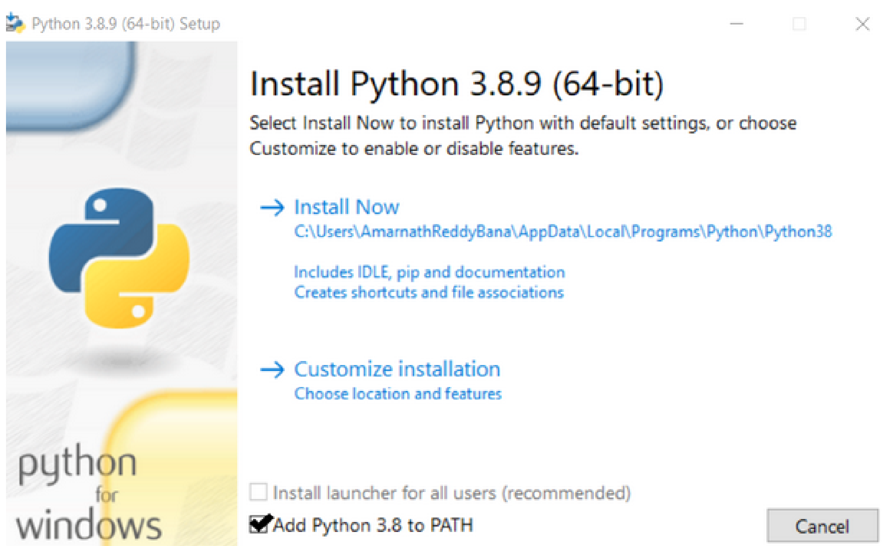
THREE

API INSTALLATION

3.1 Installation Instructions for Windows OS

3.1.1 Python 3.8 Installation

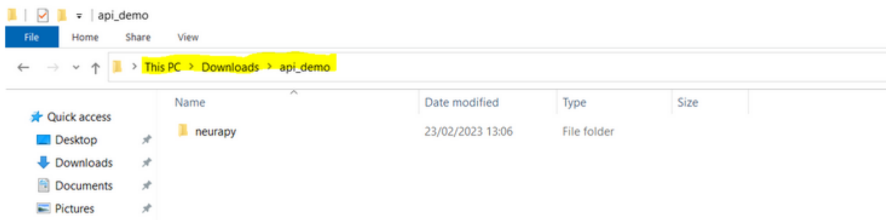
1. Download the Python 3.8 executable (*.exe*) file from the official Python site.
2. Run the downloaded Python 3.8 executable to initiate the installation process.
3. During installation, ensure to check the "Add Python 3.8 to PATH" box.
4. Proceed with the installation by selecting the "Install Now" option.



5. Close the setup, once the installation is complete

3.1.2 NeuraPy Installation

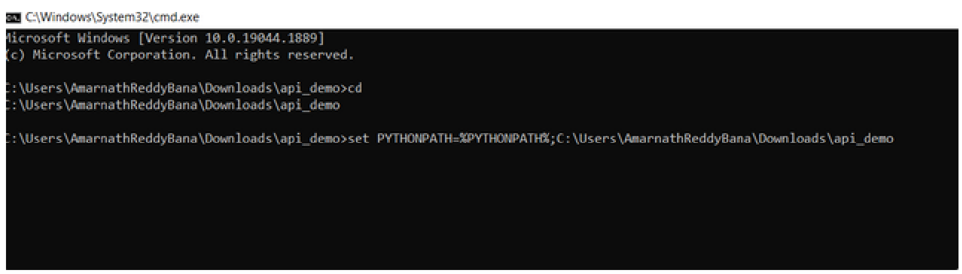
1. Download NeuraPy for Windows and navigate to the location where NeuraPy is downloaded.
2. Open the command prompt from the location where NeuraPy is downloaded. You can do this by typing "cmd" in the address bar or pressing Alt+D, then typing "cmd", and pressing Enter.



3. Install the required pywin32 and prettytable dependencies using pip (Internet connectivity is needed for this step)

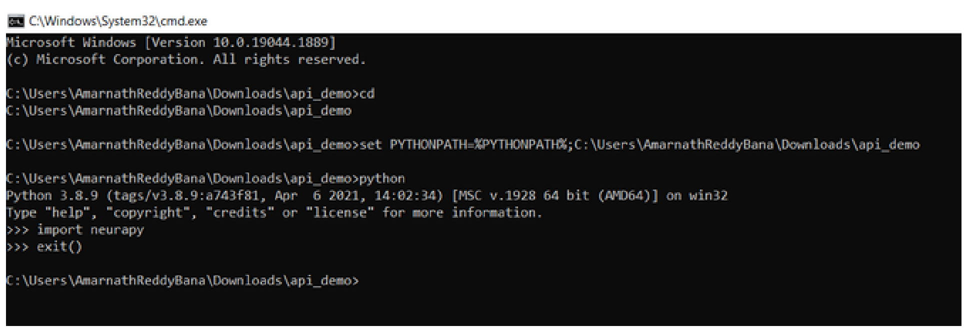
```
pip install pywin32 prettytable
```

4. Get the current location using "cd" command in command prompt and append the current location to python path




```
# open cmd from address bar
cd # to get the current location
set PYTHONPATH=%PYTHONPATH%;<location obtained from previous command>
```

5. Open python terminal from command prompt after setting the python path. API can now be imported in python console

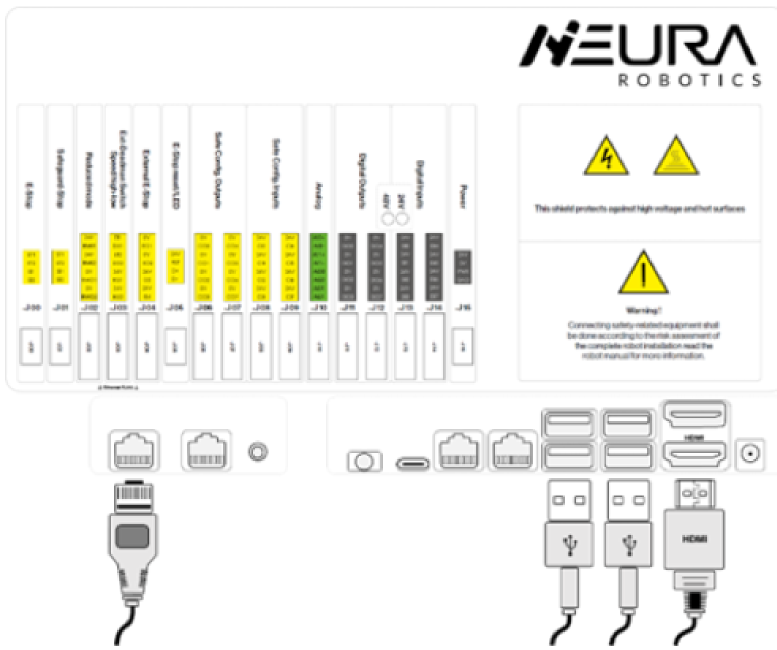


3.1.3 Network Configuration

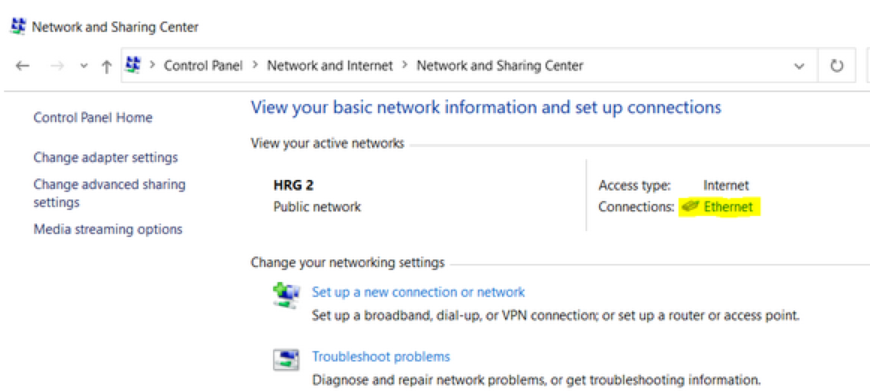
To configure the network for controlling the robot via an external machine connected through Ethernet, follow these steps:

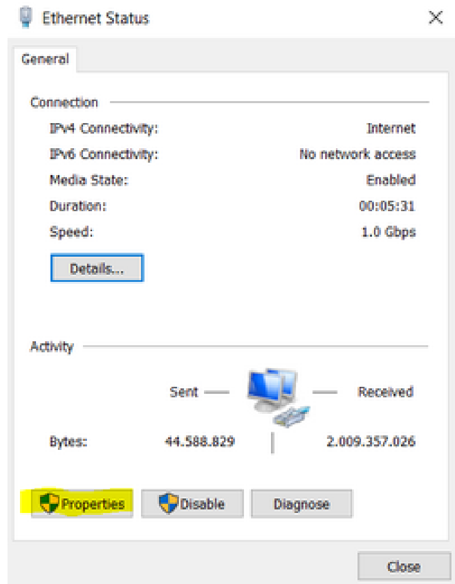
1. Connect Ethernet to the Control Box:

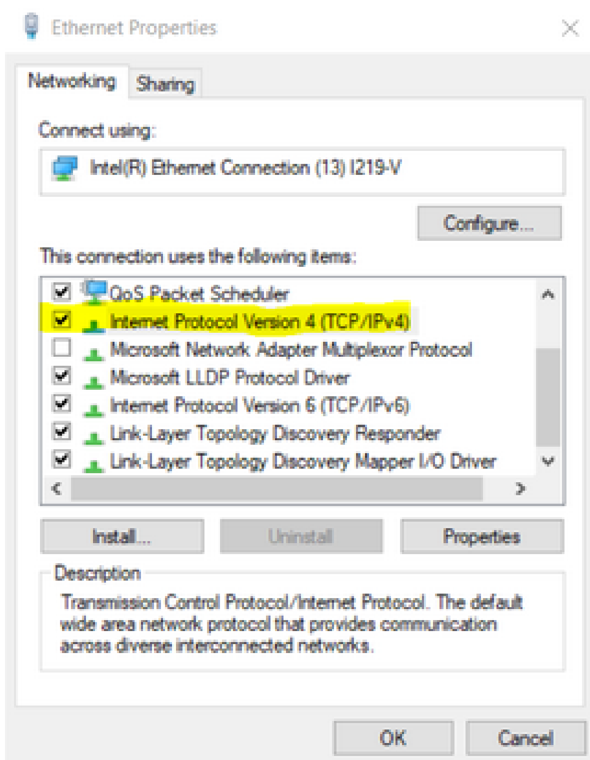
   - Open the control box using the provided key.
   - Locate the Ethernet port on the lower left side of the control box. It will be labeled as Ethernet RJ45.
   - Connect one end of the Ethernet cable to the Ethernet port on the control box, and the other end to the Ethernet port on the external PC.

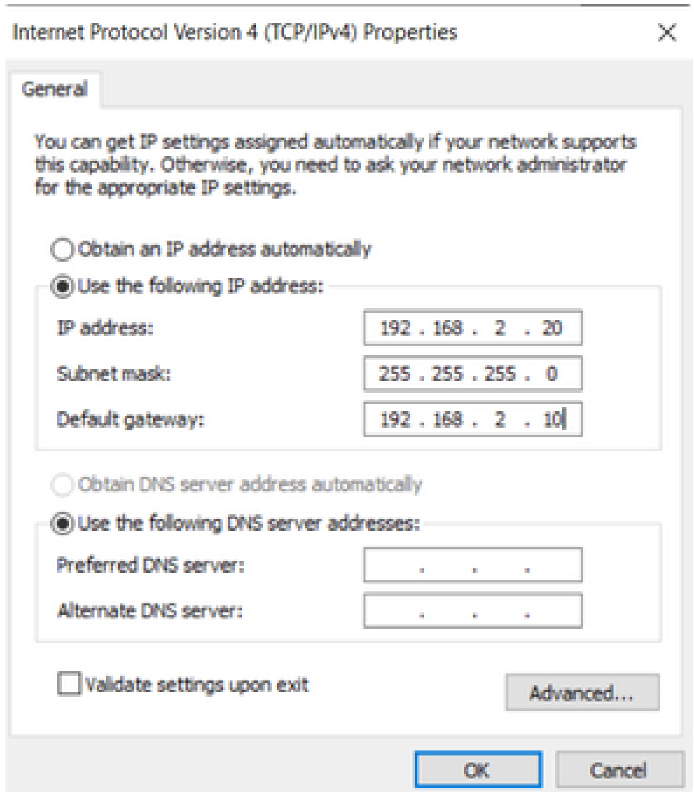


2. Control Panel → Network status and tasks → Ethernet → properties → ip4 setting




Ethernet Status
General
Connection
IPv4 Connectivity: Internet
IPv6 Connectivity: No network access
Media State: Enabled
Duration: 00:05:31
Speed: 1.0 Gbps
Details...
Activity
Sent — — Received
Bytes: 44.588.829 2.009.357.026
Properties
Disable
Diagnose
Close


Ethernet Properties
Networking
Sharing
Connect using:
Intel(R) Ethernet Connection (13) I219-V
Configure...
This connection uses the following items:
QoS Packet Scheduler
Internet Protocol Version 4 (TCP/IPv4)
Microsoft Network Adapter Multiplexor Protocol
Microsoft LLDP Protocol Driver
Internet Protocol Version 6 (TCP/IPv6)
Link-Layer Topology Discovery Responder
Link-Layer Topology Discovery Mapper I/O Driver
Install...
Uninstall
Properties
Description
Transmission Control Protocol/Internet Protocol. The default wide area network protocol that provides communication across diverse interconnected networks.
OK
Cancel



3.1.4 Configuring client Ip-address

This section is only import for those that run their communication on a non-default Ip-address (anything but 192.168.2.13). If this is the case and the communication is set up on another Ip-address e.g. 192.168.2.99 we need to configure it on the client side.

1. Set an environment variable called SOCKET_ADDRESS to 192.168.2.99
    - Open a console by pressing the Windows-key and typing cmd
    - Choose run as administrator
    - type in the console: *setx SOCKET_ADDRESS="192.168.2.99"*

3.2 Installation Instructions for Ubuntu

Note: NeuraPy API is currently supported only for Ubuntu 18 and Ubuntu 20.

1. Check the Version of Ubuntu:

   Open the terminal and type the following command:

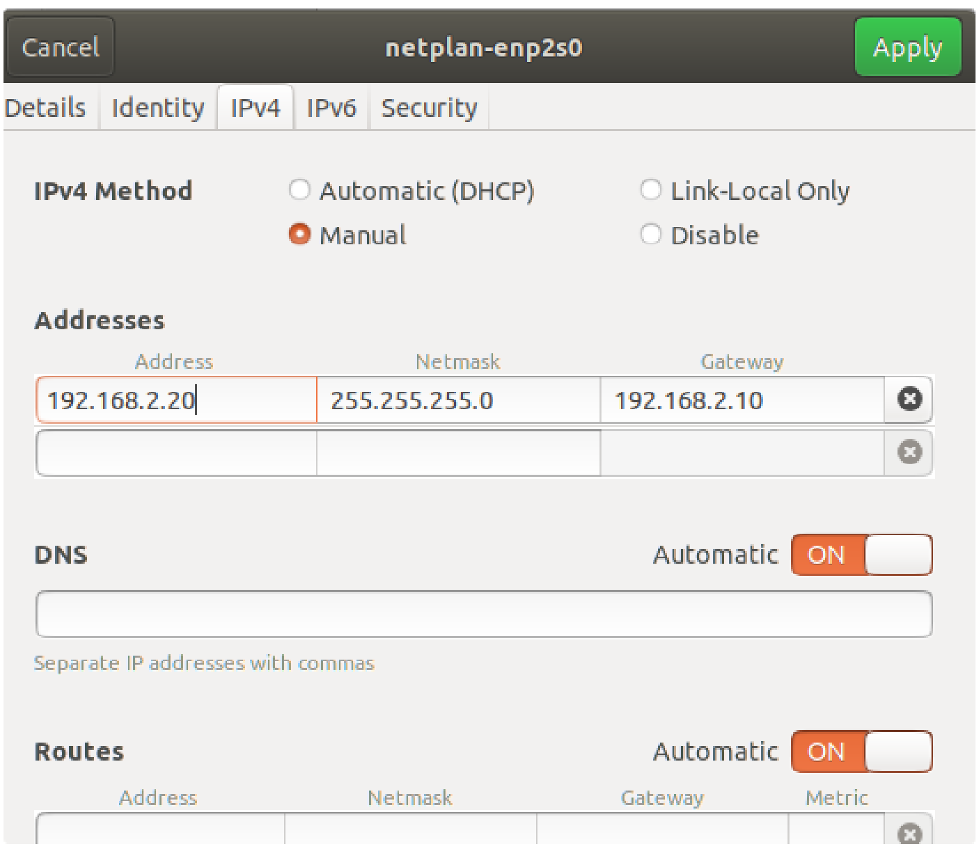
   ```
   lsb_release -a
   ```

2. Depending on the version of the ubuntu, download and navigate to DEP_Deployment folder of theneurapy and execute deploy_neurapy.shscript located inside the downloaded folder with LARA5(respective robot name) argument like below

   ```
   cd neurapy_ubuntu18 or neurapy_ubuntu20
   . deploy_neurapy.sh LARA5 #respective robot name
   ```

3.2.1 Network Configuration

When the robot is controlled via an external machine connected through Ethernet, you need to set up the network on the external machine with the following configuration



3.2.2 Installing an IDE Visual Studio Code (VSCode)

This step-by-step guide will walk you through the process of installing VSCode on your system, regardless of your technical expertise. 1. Pre-installation Preparation

Before we begin, ensure that your system meets the following requirements:

> Operating System: Windows, macOS, or Linux. Internet Connection: A stable internet connection is required for downloading the installation files. Storage Space: Make sure you have enough free space on your hard drive for the installation.

2. Downloading VSCode

   Open your web browser (e.g., Chrome, Firefox). In the address bar, type https://code.visualstudio.com/ and press Enter. You will be directed to the official Visual Studio Code website. Look for the download button and click on it. The website should automatically detect your operating system and provide the appropriate download link.

3. **Installing VSCode**

   For Windows:

   > Once the download is complete, locate the downloaded file (usually in your Downloads folder). Double-click on the downloaded file (ending with .exe) to start the installation process. Follow the on-screen instructions in the installation wizard. You may be prompted to choose installation options. You can usually leave these at their default settings unless you have specific preferences. Click on "Install" or "Finish" to complete the installation process. VSCode is now installed on your Windows system!

   For macOS:

   > Locate the downloaded file (usually in your Downloads folder). Double-click on the downloaded file (ending with .dmg) to mount the disk image. Drag the Visual Studio Code icon to the Applications folder to install it. Once the copying process is complete, eject the disk image. VSCode is now installed on your macOS system!

   For Linux:

   > Depending on your Linux distribution, installation methods may vary. The following example demonstrates installation on Ubuntu using the Terminal. Open a Terminal window by pressing Ctrl + Alt + T. Navigate to the directory where the downloaded file is located. Use the appropriate package manager command to install the .deb package. For example:
   >
   > ` sudo dpkg -i <file-name>.deb `
   >
   > ` sudo apt-get install -f `
   >
   > Replace <file-name> with the actual name of the downloaded file. Once the installation is complete, you can launch VSCode from the applications menu or by typing code in the Terminal. VSCode is now installed on your Linux system!

4. Getting Started

   Launch Visual Studio Code by searching for it in your applications menu or by double-clicking its icon. Congratulations! You have successfully installed VSCode. Familiarize yourself with the interface and explore the features to get started with your coding projects.

That's it! You're now ready to start coding with Visual Studio Code. Happy coding!

CHAPTER

FOUR

API REFERENCE

4.1 Robot Class

class `neurapy.robot.`**`Robot`**(**args*, ***kwargs*)

Bases: `object`

`activate_ethercat_interface`()

Activate the Ethercat/el6695 interface.

Args: To be added in the future if required (IP setting)

Returns
True if the ethercat interface is activated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.activate_ethercat_interface()
```

`activate_ethernetIP_interface`(**args*)

Activate the Ethernet/IP interface.

Parameters
path (*str*) – The path of yaml file to activate the Ethernet/IP interface.

Returns
True if the Ethernet/IP interface is activated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.activate_ethernetIP_interface('/path/to/yaml')
```

`activate_opcua_interface`()

Activate the OPCUA interface.

Args: To be added in the future if required (IP setting)

Returns
True if the OPCUA interface is activated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.activate_opcua_interface()
```

activate_profinet_interface()

Activate the Profinet interface.

Args: To be added in the future if required (IP setting)

Returns
True if the Profinet interface is activated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.activate_profinet_interface()
```

activate_ros_interface(**args*)

Activate the ROS interface.

Parameters
mode (*str*) – The mode to activate the ROS interface. - 'position': Activate the ROS interface in position mode. - 'torque': Activate the ROS interface in torque mode.

Returns
True if the ROS interface is activated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.activate_ros_interface('position')
```

activate_servo_interface(**args*)

Activate the Servo interface.

Parameters
mode (*str*) – The mode to activate the Servo interface. - 'position': Activate the Servo interface in position mode. - 'torque': Activate the Servo interface in torque mode.

Returns
True if the Servo interface is activated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.activate_servo_interface('position')
```

compute_forward_kinematics(*joint_angles*)

Compute forward kinematics(end-effector's cartesian pose) for the given joint configuration.

Parameters
: **joint_angles** (*List*) – List of joint angles in radians.

Returns
: List representing the tcp pose in XYZRPY format.

Return type
: List

Raises
: **ValueError** – If the provided joint_angles length doesn't match with robot's degrees of freedom.

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
target_joint_angles = [0.2]*r.dof
tcp_pose = r.compute_forward_kinematics(joint_angles)
```

compute_inverse_kinematics(*target_pose*, *reference_joint*)

Compute inverse kinematics(joint configuration) for the given end-effector's cartesian pose.

Parameters

- **target_pose** (*List*) – List representing the target end-effector pose in XYZRPY format.
- **reference_joint** (*List/numpy.array*) – List of reference joint angles(of robot configuration) in radians.

Returns
: List of joint angles in radians that achieve the target end-effector pose.

Return type
: List

Raises
: **IKNotFound** – If failed to get a solution for the given pose

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
target_end_effector_pose = [0.140448, -0.134195, 1.197456, 3.1396, -0.
↪589, -1.025]
reference_joint_angles = [-1.55, -0.69, 0.06, 1.67, -0.02, -1.57, 0.11]
joint_angle_solution = r.compute_inverse_kinematics(target_end_effector_
↪pose, reference_joint_angles)
```

create_tool(*tool_data*)

Creates a new tool in the robot's database using the provided tool data.

Parameters
`tool_data` (*dict*) – A dictionary containing the data for the new tool.

Returns
True if the tool was added successfully

Return type
bool

Raises

- `ConnectionError` – If a connection to the remote database cannot be established.
- `ValueError` – If tool creation is failed with the given input data

Tool Description

Key/Value	Default Value	Description	Required or Not
_controlOA	False	True, if the tool is controlled by controlbox analog outputs	Not Requried
_controlOD	False	True, if the tool is controlled by controlbox digital outputs	Not Required
_toolOA	False	True, if the tool is controlled by analog outputs of port present on robot tool flange	Not Required
_toolOD	False	True, if the tool is controlled by digital outputs of port present on robot tool flange	Not Required
autoM	0	Mass of the tool	Required
autoMeasureX	0	tool center of gravity (COG) in X direction,measured from robot's flange frame	Not Required
autoMeasureY	0	tool center of gravity (COG) in Y direction,measured from robot's flange frame	Not Required
autoMeasureZ	0	tool center of gravity (COG) in Z direction,measured from robot's flange frame	Not Required
closeInput	0	percentage of gripper width for close action	Not Required
cmdID	16	Input specific for certain kind of grippers	Not Required
force	0	Gripper closing force	Not Required
gripper		Gripper type	Not Required
grippertype	Standard Gripper	Type of gripper Modbus/Standard	Not Required
inertiaXX	0	Ixx of tool	Not Required
inertiaXY	0	Ixy of tool	Not Required
inertiaXZ	0	Ixz of tool	Not Required
inertiaYY	0	Iyy of tool	Not Required
inertiaYZ	0	Iyz of tool	Not Required
inertiaZZ	0	Izz of tool	Not Required
name	N/A	Name of the tool	Required
offCOA	[0, 0, 0, 0, 0, 0, 0, 0]		Not Required

continues on next page

Table 1 – continued from previous page

Key/Value	Default Value	Description	Required or Not
offCOD1	0	if the tool is controlled via control box digital outputs, offCOD1 is the pin mapped to turn off (close) the tool	Not Required
offCOD2	0		Not Required
offTOA	[0, 0]		Not Required
offTOD	0	if the tool is controlled via tool digital outputs, offTOD is the pin mapped to turn off (close) the tool	Not Required
offsetA	0	TCP roll offset	Not Required
offsetB	0	TCP pitch offset	Not Required
offsetC	0	TCP yaw offset	Not Required
offsetX	0	TCP offset in X	Not Required
offsetY	0	TCP offset in Y	Not Required
offsetZ	0	TCP offset in Z	Not Required
onCOA	[0, 0, 0, 0, 0, 0, 0, 0]		Not Required
onCOD1	0	if the tool is controlled via control box digital outputs, onCOD1 is the pin mapped to turn on (open) the tool	Not Required
onCOD2	0		Not Required
onTOA	[0, 0]		Not Required
onTOD	0	if the tool is controlled via tool digital outputs, onTOD is the pin mapped to turn on (open) the tool	Not Required
openInput	0		Not Required
portID	0		Not Required
protocol	0		Not Required
robot_type	Tool		Not Required
slaveID	0		Not Required
speed	0		Not Required

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
tool_data = { '_controlOA': False,
 '_controlOD': False,
 '_toolOA': False,
 '_toolOD': False,
 'autoM': 0,
 'autoMeasureX': 0,
 'autoMeasureY': 0,
 'autoMeasureZ': 0,
 'closeInput': 0,
 'cmdID': 16,
```

(continues on next page)

(continued from previous page)

```
 'description': 'Tool Description',
 'force': 0,
 'gripper': '',
 'grippertype': 'Standard Gripper',
 'inertiaXX': 0,
 'inertiaXY': 0,
 'inertiaXZ': 0,
 'inertiaYY': 0,
 'inertiaYZ': 0,
 'inertiaZZ': 0,
 'name': 'somerandomtool',
 'offCOA': [0, 0, 0, 0, 0, 0, 0, 0],
 'offCOD1': 0,
 'offCOD2': 0,
 'offTOA': [0, 0],
 'offTOD': 0,
 'offsetA': 0,
 'offsetB': 0,
 'offsetC': 0,
 'offsetX': 0,
 'offsetY': 0,
 'offsetZ': 0,
 'onCOA': [0, 0, 0, 0, 0, 0, 0, 0],
 'onCOD1': 0,
 'onCOD2': 0,
 'onTOA': [0, 0],
 'onTOD': 0,
 'openInput': 0,
 'portID': '',
 'protocol': 0,
 'robot_type': 'Tool',
 'slaveID': 0,
 'speed': 0}

tools_data = r.create_tool(tool_data)
```

deactivate_ethercat_interface()

Deactivate the Ethercat interface.

Returns
True if the Ethercat interface is deactivated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.deactivate_opcua_interface()
```

deactivate_ethernetIP_interface()

Deactivate the Ethernet/IP interface.

Returns
True if the Ethernet/IP interface is deactivated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.deactivate_ethernetIP_interface()
```

deactivate_opcua_interface()

Deactivate the OPCUA interface.

Returns
True if the OPCUA interface is deactivated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.deactivate_ethernetIP_interface()
```

deactivate_profinet_interface()

Deactivate the Profinet interface.

Returns
True if the Profinet interface is deactivated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.deactivate_profinet_interface()
```

deactivate_ros_interface()

Deactivate the ROS interface.

Returns
True if the ROS interface is deactivated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.deactivate_ros_interface()
```

deactivate_servo_interface()

Deactivate the Servo interface.

Returns
True if the Servo interface is deactivated successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.deactivate_servo_interface()
```

disable_collision_detection()

Disable collision detection on the robot.

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.disable_collision_detection()
```

disable_reflex()

Disable reflex after collision.

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.disable_reflex()
```

enable_collision_detection()

Enable collision detection on the robot.

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.enable_collision_detection()
```

enable_reflex()

Enable reflex after collision.

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.enable_reflex()
```

encoder2rad(*impulses*)

Convert encoder impulses to joint angles in radians.

Parameters
impulses (*list*) – A list of encoder impulses for each joint.

Returns
A list of joint angles in radians corresponding to the provided encoder impulses.

Return type
list

Sample Usage:

```
import time
from neurapy.robot import Robot
r = Robot()
encoder_ticks = r.robot_status('loadSideEncValue')
joint_angles = r.encoder2rad(encoder_ticks)
```

execute_program(*name*)

To run a program created from teach pendant's tree program creator

Parameters
`name` (*str*) – The name of the saved program to execute.

Returns
True if the program executed successfully, raises an exception if execution encountered an error.

Return type
bool

Raises
`Exception` – If an exception is raised during program execution, this function raises it with the exception message.

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
try:
    result = r.execute_program("Program_001")
    if result:
        print("Program executed successfully.")
except Exception as e:
    print("Error: " + str(e))
```

executor(*list_of_planned_motion_ids*)

Execute already planned motion IDs.

This method allows you to execute a list of planned motion IDs, which represent pre-defined motion sequences or plans. This needs to be used in conjuction with plan_move_ciruclar/joint/linear/composite/record_path functions.

Parameters
`Int` (*List of*) – List of planned motion IDs to be executed.

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
planned_motion_ids = r.plan_move_joint([0]*r.dof)  # Replace with your␣
↪planned motion IDs
r.executor([planned_motion_ids])
```

finish()

get_analog_input(*io_name*)

Get the value of an analog input.Please use get_io_configuration function to get the number of available IOs.

Parameters

io_name (*int*) – The name of the analog input.

Returns

The value of the analog input.

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_get = r.get_analog_input(1)
print(io_get)
```

get_analog_output(*io_name*)

Get the value of an analog output.Please use get_io_configuration function to get the number of available IOs.

Parameters

io_name (*int*) – The name of the analog output.

Returns

The value of the analog output.

Return type

float

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_get = r.get_analog_output(1)
print(io_get)
```

get_current_cartesian_pose()

Get the current Cartesian pose of the robot's TCP.

Returns

Cartesian pose of the tcp in the form of [X,Y,Z,w,x,y,z]

X,Y,Z - position in meters w,x,y,z - rotation representation in quaternion

Return type

List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
cartesian_pose = r.get_current_cartesian_pose()
```

Deprecated since version 4.13: This function is deprecated and will be removed in future versions. Use *get_tcp_pose_quaternion* instead.

get_current_cartesian_pose_with_timestamp()

Get the current Cartesian pose of the robot's TCP along with UTC timestamp.

Returns
Tuple of a list of robot Cartesian position values and UTC timestamp

Return type
Tuple

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
cartesian_pose, timestamp = r.get_current_cartesian_pose_with_timestamp()
```

Deprecated since version 4.13: This function is deprecated and will be removed in future versions. Use *get_tcp_pose_quaternion_with_timestamp* instead.

get_current_joint_angles()

Get current joint angles of the robot.

Returns
List of robot joint angles in radians

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
joint_angles = r.get_current_joint_angles()
```

get_current_joint_angles_with_timestamp()

Get the current joint angles of the robot along with UTC timestamp.

Returns
Tuple of a list of robot joint angles in radians and UTC timestamp

Return type
Tuple

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
joint_angles, timestamp = r.get_current_joint_angles_with_timestamp()
```

get_current_joint_torques()

Get the current joint torques of the robot.

Returns
List of joint torque values

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
joint_torques = r.get_current_joint_torques()
```

get_current_joint_torques_with_timestamp()

Get the current joint torques of the robot along with UTC timestamp.

Returns
Tuple of a list of joint torque values and UTC timestamp

Return type
Tuple

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
joint_torques, timestamp = r.get_current_joint_torques_with_timestamp()
```

get_current_joint_velocities()

Get current joint velocites of the robot.

Returns
List of robot joint velocites in radians/sec

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
joint_velocites = r.get_current_joint_velocities()
```

get_current_joint_velocities_with_timestamp()

Get the current joint velocities of the robot along with UTC timestamp.

Returns
Tuple of a list of robot joint velocities in radians/sec and UTC timestamp

Return type
Tuple

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
joint_velocities, timestamp = r.get_current_joint_velocities_with_
↪timestamp()
```

get_current_load_side_encoder_values()

Get the current load side encoder ticks of the robot.

Returns
List of encoder tick values

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
encoder_ticks = r.get_current_load_side_encoder_values()
```

get_current_load_side_encoder_values_with_timestamp()

Get the current load side encoder ticks of the robot along with UTC timestamp.

Returns
Tuple of a list of encoder tick values and UTC timestamp

Return type
Tuple

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
encoder_ticks, timestamp = r.get_current_load_side_encoder_values_with_
↪timestamp()
```

get_current_motor_side_encoder_values()

Get the current motor side encoder ticks of the robot .

Returns
List of encoder tick values

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
encoder_ticks = r.get_current_motor_side_encoder_values()
```

get_current_motor_side_encoder_values_with_timestamp()

Get the current motor side encoder ticks of the robot along with UTC timestamp.

Returns
Tuple of a list of encoder tick values and UTC timestamp

Return type
Tuple

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
encoder_ticks, timestamp = r.get_current_motor_side_encoder_values_with_
↪timestamp()
```

get_current_tool_cogs()

Get the current tool center of gravity (COG) in XYZ directions(measured from robot's flange frame/tool mounting point).

Returns
List representing the tool COG in XYZ directions.

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
tool_cog = r.get_current_tool_cogs()
```

`get_current_tool_inertias()`

Get the current tool inertias Ixx, Iyy, Izz, Ixy, Ixz, and Iyz.

Returns
List representing the tool inertias with these values.

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
tool_inertias = r.get_current_tool_inertias()
```

`get_current_tool_mass()`

Get the current tool mass.

Returns
The mass of the tool.

Return type
float

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
tool_mass = r.get_current_tool_mass()
```

`get_current_tool_properties()`

Get the current tool properties.

Returns
List containing the tool properties. i.e . [tool_mass,roll_offset,pitch_offset,yaw_offset,x_offset,y_offset,z_offset,COG
Izz, Ixy, Ixz, and Iyz]

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
tool_properties = r.get_current_tool_properties()
```

`get_current_tool_rpy_offsets()`

Get the current tool roll, pitch, and yaw (RPY) offsets in radians.

Returns
List representing the tool RPY offsets in radians.

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
tool_rpy_offsets = r.get_current_tool_rpy_offsets()
```

get_current_tool_translation_offsets()

Get the current tool translation offsets in XYZ directions.

Returns
List representing the tool translation offsets in XYZ directions..

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
tool_offsets = r.get_current_tool_translation_offsets()
```

get_diagnostics()

Method to query current robot diagnostics

Args : N/A

Return Values:

dictionary : If critical dictionary contains diagnostics values i.e. critical, powered_off, collision_status else returns empty dictionary

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
print(r.get_diagnostics())
```

get_digital_input(*io_name*)

Get the value of a digital input.Please use get_io_configuration function to get the number of available IOs.

Parameters
io_name (*int*) – The name of the digital input.

Returns
The value of the digital input.

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_get = r.get_digital_input(1)
print(io_get)
```

get_digital_output(*io_name*)

Get the value of a digital output.Please use get_io_configuration function to get the number of available IOs.

Parameters
io_name (*int*) – The name of the digital output.

Returns
The value of the digital output 0 or 1.

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_get = r.get_digital_output(1)
print(io_get)
```

get_doc(*function_name*)

Get the description and sample usage of the specified function.

Parameters
function_name (*str*) – The name of the function for which description is needed.

Returns
short description and usage of the function.

Return type
description(str)

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
description = r.get_doc("move_joint")
print(description)
```

get_encoder_offsets()

To retrieve the encoder offsets of a robot.

Returns
A list containing the encoder offsets for the robot's encoders.

Return type
List[float]

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
encoder_offsets = r.get_encoder_offsets()
```

get_errors(**args*, ***kwargs*)

Query the list of errors present on the robot.

Returns
A list of errors present on the robot.

Return type
List[str]

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
print(r.get_errors())
```

get_flange_pose(*timestamp=False*, *representation='rpy'*)

Get the current flange pose in XYZRPY format.

Returns
List representing the flange pose with XYZRPY values.

Return type
List

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
flange_pose = r.get_flange_pose()
```

get_flange_pose_quaternion()

Get the current flange pose in XYZwxyz format.

Returns

List representing the flange pose with [X,Y,Z,w,x,y,z]

X,Y,Z - position in meters w,x,y,z - rotation representation in quaternion

Return type
List

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
flange_pose_quaternion = r.get_flange_pose_quaternion()
```

get_flange_pose_quaternion_with_timestamp()

Get the current flange pose in XYZwxyz format with UTC timestamp.

Returns

List representing the flange pose with [X,Y,Z,w,x,y,z] and UTC timestamp

X,Y,Z - position in meters w,x,y,z - rotation representation in quaternion

Return type
List

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
flange_pose_quaternion, timestamp = r.get_flange_pose_quaternion_with_
↪timestamp()
```

get_flange_pose_with_timestamp()

Get the current tool flange pose in XYZRPY format with UTC timestamp.

Returns
Tuple of the flange pose with XYZRPY values and UTC timestamp

Return type
Tuple

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
flange_pose, timestamp = get_flange_pose_with_timestamp
```

get_gravity_vector()

To retrieve the set gravity vector value from the database.

Returns
A dict containing the gravity vector values

Return type
dict

Raises
ConnectionError – If there is a failure to connect to the robot's database.

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
vector = r.get_gravity_vector()
print(vector)
```

get_io_configuration()

Get the number of available IOs.

Returns
A dictionary containing the type and number of the IOs.

Return type
dict

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_configuration = r.get_io_configuration()
print(io_configuration)
```

get_joint_acceleration()

Get the current acceleration value for joint motion.

Returns: float: The current acceleration value for joint motion.(Percentage of maximum angular acceleration)

get_joint_speed()

Get the current speed value for joint motion.

Returns: float: The current speed value for joint motion.(Percentage of maximum angular speed)

get_linear_acceleration()

Get the current acceleration value for linear motion.

Returns: float: The current acceleration value for linear motion.(m/s2)

get_linear_speed()

Get the current speed value for linear motion.

Returns: float: The current speed value for linear motion.(m/s)

get_mode(**args*, ***kwargs*)

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following functions instead:
>
> - *is_robot_in_teach_mode()*
> - *is_robot_in_automatic_mode()*
> - *is_robot_in_semi_automatic_mode()*

Query the current robot mode (Teach/Automatic/SemiAutomatic).

Returns

The current robot mode.

- 'Teach' if the robot is in Teach mode.
- 'Automatic' if the robot is in Automatic mode.
- 'SemiAutomatic' if the robot is in semi-automatic mode.

Return type

str

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
mode = r.get_mode()
print(mode)
```

get_override()

Get the current override value.

Returns

The current override value.

Return type

float

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
override_value = r.get_override()
print(f"Current override value: {override_value}")
```

get_point(*name*, *representation='Joint'*)

Retrieves a point from the database based on the given name.

Parameters

- **name** (*str*) – The name of the point to retrieve from the database.
- **representation** (*str*) – To specify the point representation type.

  Joint - Default - returns Joint values of the stored point

  Cartesian - returns Cartesian pose of the stored point

Returns

A list containing the details of the point.

for Joint representation - List contains the values of joint angles in radian

for Cartesian representation - List contains X,Y,Z,R,P,Y values

X,Y,Z - 3D position - in meters

R,P,Y - 3D rotation represented in Euler angles in radians - rotation order 'ZYX'

Return type

List

Raises

- **ConnectionError** – If a connection to the remote database cannot be established.
- **ValueError** – If no point is saved in the database with the given name or the given representation type is invalid

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
point = r.get_point("P1",representation="Joint")
print(point)
point = r.get_point("P1",representation="Cartesian")
print(point)
```

get_program_names()

Retrieves list of program names from the robot's database

Returns

A list of names of the programs available in the robot's database.

Return type

list

Raises

ConnectionError – If connection to the remote database cannot be established.

Sample Usage

from neurapy.robot import Robot r = Robot() program_names = r.get_program_names() print(program_names)

get_reference_frame(*name*)

Retrieve the coordinates and orientations of a user-defined reference frame.

Parameters
name (*str*) – The name of the reference frame to retrieve.

Returns

A list containing the values of the reference frame.

frame - [X,Y,Z,R,P,Y]

X,Y,Z - 3D position - in meters

R,P,Y - 3D rotation represented in Euler angles in radians - rotation order 'ZYX'

Return type
List

Raises
ConnectionError – If there is a failure to connect to the robot's database.

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
frame = r.get_reference_frame("tool_frame")
print(frame)
```

get_reference_frame_with_offset(*name*, *offset*)

Retrieve a reference frame with an applied offset to its position.

Parameters

- **name** (*str or list*) – The name of the reference frame or the reference frame as a list.
- **offset** (*list*) – A list containing the offset values [X, Y, Z] to apply to the frame's position.

Returns

A list containing the [X, Y, Z, A, B, C] values (coordinates and angles)
of the translated reference frame.

Return type
list

Raises
ValueError – If the name argument is neither a valid string nor a list. If the offset argument is not a list of length 3.

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
x_offset = 0.02 # in meters
y_offset = 0.1 # in meters
z_offset = 0.0 # in meters
frame = r.get_reference_frame_with_offset("world",[x_offset,y_offset,z_
↪offset])
print(frame)
```

get_selected_tool_data()

Retrieves the selected tool from the robot's database.

Returns
The selected tool data from database in the robot's database.

Return type
list

Raises
ConnectionError – If connection to the remote database cannot be established.

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
tool = r.get_selected_tool_data()
print(tool)
```

get_servo_trajectory_scaling_factor(**args*, ***kwargs*)

Method to get the trajectory scaling factor from ServoJ

Returns: - [double] scaling factor

Sample Usage

```
from neurapy.robot import Robot
import time
from ruckig import InputParameter, OutputParameter, Result, Ruckig

r = Robot()

#Switch to external servo mode
r.activate_servo_interface('position')

dof = 6

otg = Ruckig(dof, 0.001)  # DoFs, control cycle
inp = InputParameter(dof)
out = OutputParameter(dof)

inp.current_position = r.get_current_joint_angles()
inp.current_velocity = [0.]*dof
inp.current_acceleration = [0.]*dof

inp.target_position = [0., 0., 0., 0., 0., 0.]
inp.target_velocity = [0.]*dof
inp.target_acceleration = [0.]*dof

inp.max_velocity = [0.5]*dof
inp.max_acceleration = [3]*dof
inp.max_jerk = [10.]*dof
res = Result.Working
```

(continues on next page)

(continued from previous page)

```
while res == Result.Working:
    '''
    Error code is returned through Servo.
    '''
    error_code = 0
    if(error_code < 3):

        res = otg.update(inp, out)

        position = out.new_position
        velocity = out.new_velocity
        acceleration = out.new_acceleration

        error_code = r.servo_j(position, velocity, acceleration)
        scaling_factor = r.get_servo_trajectory_scaling_factor()
        out.pass_to_input(inp)
        time.sleep(0.001)
    else:
        print("Servo in error, error code, ", error_code)
        break
r.deactivate_servo_interface()

r.stop()
```

get_sim_or_real(**args*, ***kwargs*)

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following function instead:
>
> - *is_robot_in_simulation()*

Method to get the mode of the robot sim/real

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
context = r.get_sim_or_real()
print(context) #Real/Simulation
```

get_tcp_pose(*timestamp=False*, *representation='rpy'*)

to get the current tcp pose in XYZRPY format

Returns
TCP pose as [X,Y,Z,R,P,Y]

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
```

(continues on next page)

(continued from previous page)

```
tcp_pose = r.get_tcp_pose()
print(tcp_pose)
```

get_tcp_pose_quaternion()

to get the current tcp pose in XYZwxyz format

Returns
TCP pose as [X,Y,Z,w,x,y,z]

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
tcp_pose_quaternion = r.get_tcp_pose_quaternion()
```

get_tcp_pose_quaternion_with_timestamp()

to get the current tcp pose in XYZwxyz format along with UTC timestamp

Returns
TCP pose as [X,Y,Z,w,x,y,z] and UTC timestamp

Return type
Tuple

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
tcp_pose_quaternion = r.get_tcp_pose_quaternion()
```

get_tcp_pose_with_timestamp()

Get the current robot's TCP pose in XYZRPY format along with UTC timestamp.

Returns
Tuple of the robot's TCP pose and UTC timestamp.

Return type
Tuple

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
tcp_pose, timestamp = r.get_tcp_pose_with_timestamp()
```

get_tool_analog_input(*io_name*)

Get the value of a tool analog input.Please use get_io_configuration function to get the number of available IOs.

Parameters
io_name (*str*) – The name of the tool analog input.

Returns
The value of the tool analog input.

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_get = r.get_tool_analog_input(1)
print(io_get)
```

get_tool_digital_input(*io_name*)

Get the value of a tool digital input.Please use get_io_configuration function to get the number of available IOs.

Parameters
: **io_name** (*str*) – The name of the tool digital input.

Returns
: The value of the tool digital input.

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_get = r.get_tool_digital_input(1)
print(io_get)
```

get_tool_digital_output(*io_name*)

Get the value of a tool digital output.Please use get_io_configuration function to get the number of available IOs.

Parameters
: **io_name** (*str*) – The name of the tool digital output.

Returns
: The value of the tool digital output.

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_get = r.get_tool_digital_output(1)
print(io_get)
```

get_tool_flange_pose()

Get the current flange pose in XYZRPY format.

Returns
: List representing the flange pose with XYZRPY values.

Return type
: List

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
flange_pose = r.get_tool_flange_pose()
```

Deprecated since version 4.13: This function is deprecated and will be removed in future versions. Use *get_flange_pose* instead.

get_tools()

Retrieves list of tools from the robot's database.

Returns
A list of tools available in the robot's database.

Return type
list

Raises
ConnectionError – If connection to the remote database cannot be established.

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
tools = r.get_tools()
print(tools)
```

get_warnings(**args*, ***kwargs*)

Query the list of warnings present on the robot.

Returns
A list of warnings present on the robot.

Return type
List[str]

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
print(r.get_warnings())
```

get_zerog_status()

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following function instead:
>
> - *is_free_drive_mode_enabled()*

Get the status of the robot free drive mode.

Returns

The status of the free drive mode.

- 'Turned On' if the free drive mode is turned on.
- 'Turned Off' if the free drive mode is turned off.

Return type
str

Sample Usage:

```
import time
from neurapy.robot import Robot
r = Robot()
status = r.get_zerog_status()
```

grasp()

To grasp the object with the attached gripper

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.grasp()
```

gripper(**args*, ***kwargs*)

Method to control the gripper attached to the robot

Sample Usage:

```
from time import sleep
from neurapy.robot import Robot
r = Robot()
r.gripper('on')
sleep(2)
r.gripper('off')
```

ik_fk(**args*, ***kwargs*)

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following functions instead:
>
> - *compute_forward_kinematics()*
> - *compute_inverse_kinematics()*

Compute forward/inverse kinematics for a given configuration.

Parameters

- **operation_type** (*str*) – The type of operation. - 'ik': Joint configuration calculation for the given cartesian position. - 'fk': Cartesian position calculation for the given joint configuration.
- **target_pose** (*List[float], optional*) – The target pose in XYZRPY format. Required only for 'ik' calculation.
- **target_angle** (*List[float], optional*) – The list of joint angles. Required only for 'fk' calculation.
- **current_joint** (*List[float], optional*) – The list of current joint angles. Required only for 'fk' calculation.

Returns

The target position in XYZRPY format if operation_type is 'ik'.
The target angle if operation_type is 'fk'.

Return type
Union[List[float], List[float]]

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
target_pose = r.ik_fk("fk", target_angle = [0.2,0.2,0.2,0.2,0.2,0.2])
target_angle = r.ik_fk("ik", target_pose = [0.140448, -0.134195, 1.
↪197456, 3.1396, -0.589, -1.025],current_joint = [-1.55, -0.69, 0.06, 1.
↪67, -0.02, -1.57, 0.11])
```

`initialize_servo()`

`io`(**args*, ***kwargs*)

Access and manipulate (output) IO values on the robot.

Parameters

- **operation_type** (*str*) – The type of operation. - 'get': Get the IO value. - 'set': Set the output IO value.
- **io_name** (*str*) – The name of the IO. (List of available IO names can be found here)
- **target_value** (*Any, optional*) – The target value of the IO. Required only for the 'set' operation.

Returns

The value of the queried 'io_name' for the 'get' operation.
True or False if the 'set' operation has succeeded or not.

Return type
Union[Any, bool]

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_get = r.io("get", io_name = "DO_1")
io_set = r.io("set", io_name = "DO_2", target_value = True)
print(io_get,io_set)
```

`is_collision_enabled()`

Check if reflex after collision is enabled for the robot.

Returns
True if reflex collision avoidance is enabled, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.is_collision_enabled():
    print("Collision detection is enabled.")
```

(continues on next page)

(continued from previous page)

```
else:
    print("Collision detction is not enabled.")
```

is_free_drive_mode_enabled()

Get the status of the robot free drive mode.

Returns

The status of the free drive mode.

- True if the free drive mode is turned on.
- False if the free drive mode is turned off.

Return type
bool

Sample Usage:

```
import time
from neurapy.robot import Robot
r = Robot()
status = r.is_free_drive_mode_enabled()
```

is_reflex_enabled()

Check if reflex after collision is enabled for the robot.

Returns
True if reflex collision avoidance is enabled, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.is_reflex_enabled():
    print("Reflex collision avoidance is enabled.")
else:
    print("Reflex collision avoidance is not enabled.")
```

is_robot_in_automatic_mode()

Method to check whether the robot is in automatic mode or not

Returns
True, if the robot is in automatic mode, False if it is not in automatic mode

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.is_robot_in_automatic_mode():
    print("Robot is in automatic mode")
```

(continues on next page)

(continued from previous page)

```
else:
    print("Robot is not in automatic mode")
```

is_robot_in_collision()

Check if the robot is currently in collision.

Returns
True if the robot is in collision, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.is_robot_in_collision():
    print("Robot is in collision.")
else:
    print("Robot is not in collision.")
```

is_robot_in_semi_automatic_mode()

Method to check whether the robot is in semi automatic mode or not

Returns
True, if the robot is in semi automatic mode, False if it is not in semi automatic mode

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.is_robot_in_semi_automatic_mode():
    print("Robot is in semi automatic mode")
else:
    print("Robot is not in semi automatic mode")
```

is_robot_in_simulation()

Method to check whether the robot is in simulation or not

Returns
True, if the robot is in simulation, False if it is in real mode

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.is_robot_in_simulation():
    print("Robot is in simulation")
else:
    print("Robot is in real")
```

is_robot_in_teach_mode()

Method to check whether the robot is in teach mode or not

Returns
True, if the robot is in teach mode, False if it is not in teach mode

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.is_robot_in_teach_mode():
    print("Robot is in teach mode")
else:
    print("Robot is not in teach mode")
```

jog(**args, **kwargs*)

Method to jog programatically. This needs to used in conjuction with turn_on_jog, turn_off_jog methods.

Args: - set_jogging_external_flag (int): A flag to enable external jogging mode

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()

'''
jog_velocity - velocity ranging from [-1,1] for all joints. Signifies␣
↪percentage of total speed used (1 = 100%)
jog_type - can be either cartesian or joint jogging
turn_on_jog sets the parameters of external jogging, jogging velocity,␣
↪jogging type and sets the mode for jogging to external mode.
'''
r.turn_on_jog(jog_velocity=[0.2, 0.2, 0.2, 0.2, 0.2, 0.2], jog_type=
↪'Joint')
r.jog(set_jogging_external_flag=1)
i = 0
'''
Requires minimum number of cycles in the loop for performing jogging.
Depends upon jogging velocity, override.
'''
while i < 500:
    #command to set flag to start jogging in external mode. This command␣
↪has to be used each time external jog command has to be sent
    r.jog(set_jogging_external_flag=1)
    i += 1

#Command to switch back to internal jogging mode (GUI)
r.turn_off_jog()
```

list_methods()

To list the available functions in the API

Returns
Table listing the available functions and short description of the functions.

Return type
table(str)

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.list_methods()
```

motion_status()

Query the motion status of the robot.

Returns

The motion status of the robot.

- 'NOT_RUNNING': If the robot is not running.
- 'RUNNING' : If the robot is running.
- 'PAUSED' : If the robot is in a paused state.

Return type
str

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
print(r.motion_status())
```

move_circular(**args*, ***kwargs*)

To move the robot in a circular path across the given poses. This method takes the following arguments/keyword arguments:

Parameters

- **target_pose** – List of 3 pose configurations (starting, middle, and end points). (type: Pose configuration - [X,Y,Z,R,P,Y], float, units: Position values in meters and rotation values in radians, required: Yes) Alternative: List of Strings (Type: [String], names of existing points)
- **speed** – Translation Speed. (type: float, units: m/sec, default_value: 0.25, required: No)
- **acceleration** – Translation Acceleration. (type: float, units: m/sec2, default_value: 0.25, required: No)
- **jerk** – Translation Jerk (type: float, units: m/sec3, default_value: 500.0, required: No)
- **rotation_speed** – Rotational Speed. (type: float, units: rad/sec, default_value: 0.5, required: No)
- **rotation_acceleration** – Rotational Acceleration. (type: float, units: rad/sec2, default_value: 1.57, required: No)
- **rotation_jerk** – Rotational Jerk. (type: float, units: rad/sec3, default_value: 500.0, required: No)

- **blending_mode** – The blending type that is selected to blend between points. (type: enum, units: N/A, default_value: DYNAMIC_BLENDING, required: No)

  0 - NO_BLENDING, if selected goes to the default blending mode. 1 - DYNAMIC_BLENDING, blending based on velocity and acceleration. 2 - STATIC_BLENDING, blending based on the given blend_radius.

- **blend_radius** – Blend Radius, if static blending is selected. (type: float, units: m, default_value: 0.01, required: No)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed

Return Values:

Returns

True if motion is executed successfully, False if motion is not executed successfully

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
circular_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "jerk": 100,
    "rotation_speed": 1.57,
    "rotation_acceleration": 5.0,
    "rotation_jerk": 100,
    "blending_mode": 1,
    "target_pose": [
        [
            0.3744609827431085,
            -0.3391784988266481,
            0.23276604279256016,
            3.14119553565979,
            -0.00017731254047248513,
            -0.48800110816955566
        ],
        [
            0.37116786741831503,
            -0.19686307684994242,
            0.23300456855796453,
```

(continues on next page)

(continued from previous page)

```
            3.141423225402832,
            -0.00020668463548645377,
            -0.48725831508636475
        ],
        [
            0.5190337951593321,
            -0.1969996948428492,
            0.23267853691809767,
            3.1414194107055664,
            -0.00017726201622281224,
            -0.48750609159469604
        ]
    ],
    "current_joint_angles": r.get_current_joint_angles()
}
r.move_circular(**circular_property)
r.stop() # if there are multiple motions than,this needs to be called␣
↪only once at the end of the script
```

Example 2: .. code-block:: python

from neurapy.robot import Robot

r = Robot() r.move_circular([“P1”, “P2”, “P3”]) r.stop()

move_composite(**args*, ***kwargs*)

To move the robot in the specified linear and circular motion combinations.

Parameters

commands – List of linear and circular command combinations.

linear command: - targets: list of target poses

circular command: - targets: list of target poses

(type: Pose configuration - [X,Y,Z,R,P,Y], float, units: Position values in meters and rotation values in radians, required: Yes)

Parameters

- **speed** – Translation Speed. (type: float, units: m/sec, default_value: 0.25, required: No)
- **acceleration** – Translation Acceleration. (type: float, units: m/sec2, default_value: 0.25, required: No)
- **jerk** – Translation Jerk (type: float, units: m/sec3, default_value: 500.0, required: No)
- **rotation_speed** – Rotational Speed. (type: float, units: rad/sec, default_value: 0.5, required: No)
- **rotation_acceleration** – Rotational Acceleration. (type: float, units: rad/sec2, default_value: 1.57, required: No)
- **rotation_jerk** – Rotational Jerk. (type: float, units: rad/sec3, default_value: 500.0, required: No)
- **blending_mode** – The blending type that is selected to blend between points. (type: enum, units: N/A, default_value: STATIC_BLENDING, required: No)

0 - NO_BLENDING, if selected goes to the default blending mode. 1 - DYNAMIC_BLENDING, blending based on velocity and acceleration. 2 - STATIC_BLENDING, blending based on the given blend_radius.

- **blend_radius** – Blend Radius, if static blending is selected. (type: float, units: m, default_value: 0.01, required: No)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed

Returns

True if motion is executed successfully, False if motion is not executed successfully

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
composite_motion_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "jerk": 100,
    "rotation_speed": 1.57,
    "rotation_acceleration": 5.0,
    "rotation_jerk": 100,
    "blending_mode": 2,
    "blend_radius": 0.005,
    "current_joint_angles": r.get_current_joint_angles(),
    "commands": [
        {
            "linear": {
                "targets": [
                    [
                        -0.000259845199876027,
                        -0.5211437049195536,
                        0.4429382717719519,
                        3.14123272895813,
                        -0.0007908568368293345,
                        -1.570908784866333
                    ],
                    [
                        -0.16633498440272945,
                        -0.5201452059140722,
```

(continues on next page)

(continued from previous page)

```
                        0.4427486025872017,
                        3.140937089920044,
                        -0.0005319403717294335,
                        -1.571555495262146
                    ]
                ]
            }
        },
        {
            "circular": {
                "targets": [
                    [
                        -0.16633498440272945,
                        -0.5201452059140722,
                        0.4427486025872017,
                        3.140937089920044,
                        -0.0005319403717294335,
                        -1.571555495262146
                    ],
                    [
                        -0.16540090985202305,
                        -0.3983552679378624,
                        0.44267608017426174,
                        3.1407113075256348,
                        -0.00036628879024647176,
                        -1.5714884996414185
                    ],
                    [
                        -0.33446498807559716,
                        -0.3989652352814891,
                        0.4421152856242009,
                        3.1402060985565186,
                        0.00030071483342908323,
                        -1.572899580001831
                    ]
                ]
            }
        }
    ]
}
r.move_composite(**composite_motion_property)
r.stop() # if there are multiple motions than,this needs to be called␣
↪only once at the end of the script
```

move_joint(**args*, ***kwargs*)

To move the robot to specified joint configuration in joint space.

Parameters

- **target_joint** – List of joint configurations (type: List of Joint Values - float or int, units: radians, required: Yes) Alternative: (List of) String (type: String, name of existing point)
- **speed** – Angular Speed. (type: float, units: % of maximum angular speed, default_value: 0.25, required: No)

- **acceleration** – Angular Acceleration. (type: float, units: % of maximum angular acceleration, default_value: 0.25, required: No)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed
- **enable_blending** – Allows to switch between blending and non blending mode. (type: Bool - True/False, units: N/A, default_value: False, required: No)

  If set to True - Joint Motion does not stop at every point If set to False - Joint Motion stops at every point.

Returns

True if motion is executed successfully, False if motion is not executed successfully

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

Example 1:

```
from neurapy.robot import Robot
r = Robot()
joint_property = {
    "speed": 50.0,
    "acceleration": 50.0,
    "safety_toggle": True,
    "target_joint": [
        [
            2.5995838308821924,
            0.24962416292345468,
            -1.8654403327490414,
            0.04503286318691005,
            -1.1740563715454926,
            0.10337461241185522
        ],
        [
            2.1372059994827075,
            0.24939733788589463,
            -1.8651270179353125,
            0.044771940725327274,
            -1.173860821592129,
            0.10315646291502645
        ],
        [
            1.9180047887810003,
```

(continues on next page)

(continued from previous page)

```
            -0.24855170101601043,
            -1.3680228668892351,
            0.12404421791100637,
            -1.1914147150222498,
            -0.13255713717112075
        ]
    ],
    "current_joint_angles":  r.get_current_joint_angles()
}
r.move_joint(**joint_property)
r.stop() # if there are multiple motions than,this needs to be called␣
↪only once at the end of the script
```

Example 2:

```
from neurapy.robot import Robot
r = Robot()
r.move_joint("Home")
r.move_joint(["P1", "P2"])
r.stop()
```

move_linear(**args*, ***kwargs*)

To move the robot to specified poses in Cartesian/Task space.

Parameters

- **target_pose** – List of pose configurations. (type: Pose configuration - [X,Y,Z,R,P,Y], float, units: Position values in meters and rotation values in radians, required: Yes) Alternative: List of Strings (type: [String], names of existing points)
- **speed** – Translation Speed. (type: float, units: m/sec, default_value: 0.25, required: No)
- **acceleration** – Translation Acceleration. (type: float, units: m/sec2, default_value: 0.25, required: No)
- **jerk** – Translation Jerk (type: float, units: m/sec3, default_value: 500.0, required: No)
- **rotation_speed** – Rotational Speed. (type: float, units: rad/sec, default_value: 0.5, required: No)
- **rotation_acceleration** – Rotational Acceleration. (type: float, units: rad/sec2, default_value: 1.57, required: No)
- **rotation_jerk** – Rotational Jerk. (type: float, units: rad/sec3, default_value: 500.0, required: No)
- **blending** – Blending. (type: Bool - True/False, units: N/A, default_value: False, required: No)

  True - Blending is turned on, motions inside are executed with given blending mode. False - Blending is turned off, motions stop at each point.
- **blending_mode** – The blending type that is selected to blend between points. (type: enum, units: N/A, default_value: NO_BLENDING, required: No)

  0 - NO_BLENDING, if selected goes to the default blending mode. 1 - DYNAMIC_BLENDING, blending based on velocity and acceleration. 2 - STATIC_BLENDING, blending based on the given blend_radius.

- **blend_radius** – Blend Radius, if static blending is selected. (type: float, units: m, default_value: 0.01, required: No)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed

Returns
True if motion is executed successfully, False if motion is not executed successfully

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

Example 1:

```
from neurapy.robot import Robot

r = Robot()
linear_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "rotation_speed": 0.5,
    "blending": True,
    "blending_mode": 2.
    "blend_radius": 0.005,
    "target_pose": [
        [
            0.3287228886,
            -0.1903355329,
            0.4220780352,
            0.08535207028439847,
            -2.797181496822229,
            2.4713321627410485
        ],
        [
            0.2093363791501374,
            -0.31711250784165884,
            0.422149168855134,
            -3.0565555095672607,
            -0.3447442352771759,
            -1.1323236227035522
        ],
        [
            0.2090521916195534,
            -0.5246753336643587,
```

(continues on next page)

(continued from previous page)

```
            0.4218773613553828,
            -3.0569007396698,
            -0.3448921740055084,
            -1.1323626041412354
        ],
        [
            0.3287228886,
            -0.1903355329,
            0.4220780352,
            0.08535207028439847,
            -2.797181496822229,
            2.4713321627410485
        ]
    ],
    "current_joint_angles":  r.get_current_joint_angles()
}
r.move_linear(**linear_property)
r.stop() # if there are multiple motions than,this needs to be called␣
↪only once at the end of the script
```

Example 2:

```
from neurapy.robot import Robot

r = Robot()
r.move_linear(["P1", "P2"])
r.stop()
```

move_linear_from_current_position(**args*, ***kwargs*)

To move the robot from current position to the specified target pose/s.Unlike move_linear, current position is added as the first target in this function.

Parameters

- **target_pose** – List of pose configurations. (type: Pose configuration - [X,Y,Z,R,P,Y], float, units: Position values in meters and rotation values in radians, required: Yes)
- **speed** – Translation Speed. (type: float, units: m/sec, default_value: 0.25, required: No)
- **acceleration** – Translation Acceleration. (type: float, units: m/sec2, default_value: 0.25, required: No)
- **jerk** – Translation Jerk (type: float, units: m/sec3, default_value: 500.0, required: No)
- **rotation_speed** – Rotational Speed. (type: float, units: rad/sec, default_value: 0.5, required: No)
- **rotation_acceleration** – Rotational Acceleration. (type: float, units: rad/sec2, default_value: 1.57, required: No)
- **rotation_jerk** – Rotational Jerk. (type: float, units: rad/sec3, default_value: 500.0, required: No)
- **blending** – Blending. (type: Bool - True/False, units: N/A, default_value: False, required: No)

  True - Blending is turned on, motions inside are executed with given blending mode. False - Blending is turned off, motions stop at each point.

- **blending_mode** – The blending type that is selected to blend between points. (type: enum, units: N/A, default_value: NO_BLENDING, required: No)

  0 - NO_BLENDING, if selected goes to the default blending mode. 1 - DYNAMIC_BLENDING, blending based on velocity and acceleration. 2 - STATIC_BLENDING, blending based on the given blend_radius.

- **blend_radius** – Blend Radius, if static blending is selected. (type: float, units: m, default_value: 0.01, required: No)

- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)

- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed

Returns

True if motion is executed successfully, False if motion is not executed successfully

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
linear_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "jerk": 100,
    "rotation_speed": 1.57,
    "rotation_acceleration": 5.0,
    "rotation_jerk": 100,
    "blending": True,
    "blending_mode": 1,
    "blend_radius": 0.005,
    "target_pose": [
        [
            0.3287228886,
            -0.1903355329,
            0.4220780352,
            0.08535207028439847,
            -2.797181496822229,
            2.4713321627410485
        ]
    ],
    "current_joint_angles":  r.get_current_joint_angles()
}
```

(continues on next page)

(continued from previous page)

```
r.move_linear_from_current_position(**linear_property)
r.stop() # if there are multiple motions than,this needs to be called␣
↪only once at the end of the script
```

move_trajectory(**args*, ***kwargs*)

Move the robot with a given joint trajectory.

Parameters:

timestamps
[list of float] List of times at which the joint configurations should be reached (units: sec).

target_joint
[list of float] List of joint configurations (units: radians).

current_joint_angles
[list of float, optional] Current Robot Joint Configuration. If not provided, it will be obtained from Robot Status method.

Returns
True if the motion is executed successfully, False if not.

Return type
bool

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
trajectory_motion_property = {
  "current_joint_angles": r.get_current_joint_angles(),
  "timestamps": [
    0.01, 0.02, 0.03
  ],
  "target_joint": [
    [0.0531381, 0.157485, -0.100272, 1.29569, -5.99211e-05, 0.700957,-0.
↪000371511],
    [-0.208897,0.461728,-0.433937,1.66485,-5.99211e-05,0.700382,-0.
↪000383495],
    [0.0120801,0.621298,0.0149563,1.67381,5.99211e-05,0.687403,-0.
↪000359527]
  ]
}
r.move_trajectory(**trajectory_motion_property)
r.stop() # if there are multiple motions than,this needs to be called␣
↪only once at the end of the script
```

notify_error(*msg*)

To notify an error message to teach pendant .

Parameters
msg (*str*) – The error message to be sent.

Returns
True if successful, else False

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.notify_error("Something went wrong!"):
    print("Error message sent successfully.")
else:
    print("Failed to send error message.")
```

notify_info(*msg*)

To notify a info message to teach pendant.

Parameters
msg (*str*) – The error message to be sent.

Returns
True if successful, else False

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.notify_info("This is an info!"):
    print("Info message sent successfully.")
else:
    print("Failed to send info message.")
```

notify_warning(*msg*)

To notify a warning message to teach pendant.

Parameters
msg (*str*) – The error message to be sent.

Returns
True if successful, else False

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.notify_warning("This is a warning!"):
    print("Warning message sent successfully.")
else:
    print("Failed to send warning message.")
```

override(**args*, ***kwargs*)

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following functions instead:
>
> - *get_override()*
> - *set_override()*

Set or get the override value on the robot.

Parameters

- **operation_type** (*str*) – The type of operation. - 'get': Get the current override value. - 'set': Set the override value.
- **target_value** (*float, optional*) – The override value. Required only for the 'set' operation.

Returns

True or False if the 'set' operation succeeds or fails.
The override value if the 'get' operation succeeds.

Return type
Union[bool, float]

Sample Usage:

```
import time
from neurapy.robot import Robot
r = Robot()
override_value = r.override("get")
print("override_value : " + str(override_value))
time.sleep(1)

override_value = r.override("set", target_value = 0.4)
print("Setting Override to 0.4")
time.sleep(1)

override_value = r.override("get")
print("override_value : " + str(override_value))
```

pause(**args*, ***kwargs*)

To pause the robot's motion.

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.pause()
```

plan_joint_trajectory(**args*, ***kwargs*)

To plan the given joint trajectory.

Parameters:

timestamps
[list of float] List of times at which the joint configurations should be reached (units: sec).

target_joint
[list of float] List of joint configurations (units: radians).

current_joint_angles
[list of float, optional] Current Robot Joint Configuration. If not provided, it will be obtained from Robot Status method.

Parameters
store_id – identifier to the stored plan. (type: int , if not provided a random will be assigned, which will be returned after the function execution)

Returns
plan_id (equal to store_id if provided in inputs, else an identifier function has generated to store the plan)

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
trajectory_motion_property = {
  "current_joint_angles": r.get_current_joint_angles(),
  "timestamps": [
    0.01, 0.02, 0.03
  ],
  "target_joint": [
    [0.0531381, 0.157485, -0.100272, 1.29569, -5.99211e-05, 0.700957,-0.
↪000371511],
    [-0.208897,0.461728,-0.433937,1.66485,-5.99211e-05,0.700382,-0.
↪000383495],
    [0.0120801,0.621298,0.0149563,1.67381,5.99211e-05,0.687403,-0.
↪000359527]
  ]
}
plan_id = r.plan_move_trajectory(**trajectory_motion_property)
execute = r.executor([plan_id])
```

plan_move_circular(**args*, ***kwargs*)

To plan a circular path across the given poses. This method takes the following arguments/keyword arguments:

Parameters

- **target_pose** – List of 3 pose configurations (starting, middle, and end points). (type: Pose configuration - [X,Y,Z,R,P,Y], float, units: Position values in meters and rotation values in radians, required: Yes)
- **speed** – Translation Speed. (type: float, units: m/sec, default_value: 0.25, required: No)

- **acceleration** – Translation Acceleration. (type: float, units: m/sec2, default_value: 0.25, required: No)
- **jerk** – Translation Jerk (type: float, units: m/sec3, default_value: 500.0, required: No)
- **rotation_speed** – Rotational Speed. (type: float, units: rad/sec, default_value: 0.5, required: No)
- **rotation_acceleration** – Rotational Acceleration. (type: float, units: rad/sec2, default_value: 1.57, required: No)
- **rotation_jerk** – Rotational Jerk. (type: float, units: rad/sec3, default_value: 500.0, required: No)
- **blending_mode** – The blending type that is selected to blend between points. (type: enum, units: N/A, default_value: DYNAMIC_BLENDING, required: No)

  0 - NO_BLENDING, if selected goes to the default blending mode. 1 - DYNAMIC_BLENDING, blending based on velocity and acceleration. 2 - STATIC_BLENDING, blending based on the given blend_radius.
- **blend_radius** – Blend Radius, if static blending is selected. (type: float, units: m, default_value: 0.01, required: No)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed
- **store_id** – identifier to the stored plan. (type: int , if not provided a random will be assigned, which will be returned after the function execution)

Return Values:

Returns

plan_id (equal to store_id if provided in inputs, else an identifier function has generated to store the plan)

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
circular_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "jerk": 100,
    "rotation_speed": 1.57,
    "rotation_acceleration": 5.0,
    "rotation_jerk": 100,
```

(continues on next page)

(continued from previous page)

```
    "blending_mode": 2,
    "blend_radius": 0.05,
    "target_pose": [
        [
            0.3744609827431085,
            -0.3391784988266481,
            0.23276604279256016,
            3.14119553565979,
            -0.00017731254047248513,
            -0.48800110816955566
        ],
        [
            0.37116786741831503,
            -0.19686307684994242,
            0.23300456855796453,
            3.141423225402832,
            -0.00020668463548645377,
            -0.48725831508636475
        ],
        [
            0.5190337951593321,
            -0.1969996948428492,
            0.23267853691809767,
            3.1414194107055664,
            -0.00017726201622281224,
            -0.48750609159469604
        ]
    ],
    "store_id":234,
    "current_joint_angles": r.get_current_joint_angles()
}
plan_id = r.plan_move_circular(**circular_property)
execute = r.executor([plan_id])
```

plan_move_composite(**args*, ***kwargs*)

To plan move composite across the given poses. This method takes the following arguments/keyword arguments:

To move the robot in the specified linear and circular motion combinations.

Parameters

commands – List of linear and circular command combinations.

linear command: - targets: list of target poses

circular command: - targets: list of target poses

(type: Pose configuration - [X,Y,Z,R,P,Y], float, units: Position values in meters and rotation values in radians, required: Yes)

Parameters

- **speed** – Translation Speed. (type: float, units: m/sec, default_value: 0.25, required: No)
- **acceleration** – Translation Acceleration. (type: float, units: m/sec2, default_value: 0.25, required: No)

- **jerk** – Translation Jerk (type: float, units: m/sec3, default_value: 500.0, required: No)
- **rotation_speed** – Rotational Speed. (type: float, units: rad/sec, default_value: 0.5, required: No)
- **rotation_acceleration** – Rotational Acceleration. (type: float, units: rad/sec2, default_value: 1.57, required: No)
- **rotation_jerk** – Rotational Jerk. (type: float, units: rad/sec3, default_value: 500.0, required: No)
- **blending_mode** – The blending type that is selected to blend between points. (type: enum, units: N/A, default_value: STATIC_BLENDING, required: No)

  0 - NO_BLENDING, if selected goes to the default blending mode. 1 - DYNAMIC_BLENDING, blending based on velocity and acceleration. 2 - STATIC_BLENDING, blending based on the given blend_radius.
- **blend_radius** – Blend Radius, if static blending is selected. (type: float, units: m, default_value: 0.01, required: No)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed
- **store_id** – identifier to the stored plan. (type: int , if not provided a random will be assigned, which will be returned after the function execution)

Returns

plan_id (equal to store_id if provided in inputs, else an identifier function has generated to store the plan)

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
composite_motion_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "jerk": 100,
    "rotation_speed": 1.57,
    "rotation_acceleration": 5.0,
    "rotation_jerk": 100,
    "blending_mode": 1,
    "blend_radius": 0.01,
    "current_joint_angles": r.get_current_joint_angles(),
```

(continues on next page)

(continued from previous page)

```
    "commands": [
        {
            "linear": {
                "blend_radius": 0.005,
                "targets": [
                    [
                        -0.000259845199876027,
                        -0.5211437049195536,
                        0.4429382717719519,
                        3.14123272895813,
                        -0.0007908568368293345,
                        -1.570908784866333
                    ],
                    [
                        -0.16633498440272945,
                        -0.5201452059140722,
                        0.4427486025872017,
                        3.140937089920044,
                        -0.0005319403717294335,
                        -1.571555495262146
                    ]
                ]
            }
        },
        {
            "circular": {
                "targets": [
                    [
                        -0.16633498440272945,
                        -0.5201452059140722,
                        0.4427486025872017,
                        3.140937089920044,
                        -0.0005319403717294335,
                        -1.571555495262146
                    ],
                    [
                        -0.16540090985202305,
                        -0.3983552679378624,
                        0.44267608017426174,
                        3.1407113075256348,
                        -0.00036628879024647176,
                        -1.5714884996414185
                    ],
                    [
                        -0.3344649880755971 6,
                        -0.3989652352814891,
                        0.4421152856242009,
                        3.1402060985565186,
                        0.00030071483342908323,
                        -1.572899580001831
                    ]
                ]
```

(continues on next page)

(continued from previous page)

```
            }
        }
    ],
    "store_id":234,
    "current_joint_angles": r.get_current_joint_angles()
}
plan_id= r.plan_move_composite(**composite_motion_property)
execute = r.executor([plan_id])
```

plan_move_joint(**args*, ***kwargs*)

To plan a joint space motion across the given configurations. This method takes the following arguments/keyword arguments:

To move the robot to specified joint configuration in joint space.

Parameters

- **target_joint** – List of joint configurations. (type: List of Joint Values - float, units: radians, required: Yes)
- **speed** – Angular Speed. (type: float, units: % of maximum angular speed, default_value: 0.25, required: No)
- **acceleration** – Angular Acceleration. (type: float, units: % of maximum angular acceleration, default_value: 0.25, required: No)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed
- **enable_blending** – Allows to switch between blending and non blending mode. (type: Bool - True/False, units: N/A, default_value: False, required: No)

  If set to True - Joint Motion does not stop at every point If set to False - Joint Motion stops at every point.
- **store_id** – identifier to the stored plan. (type: int , if not provided a random will be assigned, which will be returned after the function execution)

Returns

plan_id (equal to store_id if provided in inputs, else an identifier function has generated to store the plan)

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
joint_property = {
    "speed": 50.0,
    "acceleration": 50.0,
    "safety_toggle": True,
    "enable_blending": True,
    "target_joint": [
        [
            2.5995838308821924,
            0.24962416292345468,
            -1.8654403327490414,
            0.04503286318691005,
            -1.1740563715454926,
            0.10337461241185522
        ],
        [
            2.1372059994827075,
            0.24939733788589463,
            -1.8651270179353125,
            0.044771940725327274,
            -1.173860821592129,
            0.10315646291502645
        ],
        [
            1.9180047887810003,
            -0.24855170101601043,
            -1.3680228668892351,
            0.12404421791100637,
            -1.1914147150222498,
            -0.13255713717112075
        ]
    ],
    "store_id":234,
    "current_joint_angles":  r.get_current_joint_angles()
}
plan_id =r.plan_move_joint(**joint_property)
execute = r.executor([plan_id])
```

plan_move_linear(**args*, ***kwargs*)

To plan a move linear path across the given poses. This method takes the following arguments/keyword arguments:

To move the robot to specified poses in Cartesian/Task space.

Parameters

- **target_pose** – List of pose configurations. (type: Pose configuration - [X,Y,Z,R,P,Y], float, units: Position values in meters and rotation values in radians, required: Yes)
- **speed** – Translation Speed. (type: float, units: m/sec, default_value: 0.25, required: No)
- **acceleration** – Translation Acceleration. (type: float, units: m/sec2, default_value: 0.25, required: No)

- **jerk** – Translation Jerk (type: float, units: m/sec3, default_value: 500.0, required: No)
- **rotation_speed** – Rotational Speed. (type: float, units: rad/sec, default_value: 0.5, required: No)
- **rotation_acceleration** – Rotational Acceleration. (type: float, units: rad/sec2, default_value: 1.57, required: No)
- **rotation_jerk** – Rotational Jerk. (type: float, units: rad/sec3, default_value: 500.0, required: No)
- **blending** – Blending. (type: Bool - True/False, units: N/A, default_value: False, required: No)

  True - Blending is turned on, motions inside are executed with given blending mode. False - Blending is turned off, motions stop at each point.
- **blending_mode** – The blending type that is selected to blend between points. (type: enum, units: N/A, default_value: NO_BLENDING, required: No)

  0 - NO_BLENDING, if selected goes to the default blending mode. 1 - DYNAMIC_BLENDING, blending based on velocity and acceleration. 2 - STATIC_BLENDING, blending based on the given blend_radius.
- **blend_radius** – Value of the blend radius, if blending is needed between two segments of motion. (type: float, units: meters, default_value: 0, required: No)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed
- **store_id** – identifier to store the plan. (type: int , if not provided a random will be assigned, which will be returned after the function execution)

Returns

plan_id (equal to store_id if provided in inputs, else an identifier function has generated to store the plan)

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
linear_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "jerk": 100,
    "rotation_speed": 1.57,
```

(continues on next page)

(continued from previous page)

```
    "rotation_acceleration": 5.0,
    "rotation_jerk": 100,
    "blending": True,
    "blending_mode": 2,
    "blend_radius": 0.01,
    "target_pose": [
        [
            0.3287228886,
            -0.1903355329,
            0.4220780352,
            0.08535207028439847,
            -2.797181496822229,
            2.4713321627410485
        ],
        [
            0.2093363791501374,
            -0.31711250784165884,
            0.422149168855134,
            -3.0565555095672607,
            -0.3447442352771759,
            -1.1323236227035522
        ],
        [
            0.2090521916195534,
            -0.5246753336643587,
            0.4218773613553828,
            -3.0569007396698,
            -0.3448921740055084,
            -1.1323626041412354
        ],
        [
            0.3287228886,
            -0.1903355329,
            0.4220780352,
            0.08535207028439847,
            -2.797181496822229,
            2.4713321627410485
        ]
    ],
    "store_id":234,
    "current_joint_angles":  r.get_current_joint_angles()
}
plan_id = r.plan_move_linear(**linear_property)
execute_motion = r.executor([plan_id]) #To execute the planned id
```

plan_move_recorded_path(**args*, ***kwargs*)

To plan the given pre-recorded path. This method takes the following arguments/keyword arguments:

Parameters

- **is_motion** – True if constant velocity is needed during the motion. (type: bool, required: Yes)
- **file_location** – Location of the recorded path file. (type: str, required: Yes)

- **speed** – Linear Speed. (type: float, units: m/sec, required: Yes)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed
- **store_id** – identifier to the stored plan. (type: int , if not provided a random will be assigned, which will be returned after the function execution)

Returns
plan_id (equal to store_id if provided in inputs, else an identifier function has generated to store the plan)

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.
- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

power(**args*, ***kwargs*)

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following functions instead:
>
> - *power_on()*
> - *power_off()*

Turn on/off the power to the robot.

Parameters
value (*str*) – Specifies the power action. - 'on' : Power on the robot. - 'off' : Power off the robot.

Returns
True if the operation is executed successfully, False otherwise.

Return type
bool

Sample Usage:

```
from time import sleep
from neurapy.robot import Robot

r = Robot()
r.power('on')
sleep(2)
r.power('off')
```

power_off()

To power off the robot.

Returns
True if the robot's power is successfully turned off, False otherwise.

Return type
bool

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
if r.power_off():
    print("Robot powered off successfully.")
else:
    print("Failed to power off the robot.")
```

power_on()

To power on the robot.

Returns
True if the robot's power is successfully turned on, False otherwise.

Return type
bool

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
if r.power_on():
    print("Robot powered on successfully.")
else:
    print("Failed to power on the robot.")
```

program_status()

Query the program status of the robot.

Returns

The program status of the robot.

- 'NOT_RUNNING': If the program is not running from the Teach pendant.
- 'RUNNING' : If the program is running from the Teach pendant.
- 'PAUSED' : If the program running from the Teach pendant is in a pause state.

Return type
str

Sample Usage:

```
from neurapy.robot import Robot

r = Robot()
print(r.program_status())
```

quaternion_to_rpy(*w*, *x*, *y*, *z*)

Convert a quaternion representation to roll-pitch-yaw (RPY) angles.

Parameters

- **w** (*float*) – Scalar component (real) of the quaternion.
- **x** (*float*) – First vector component of the quaternion.
- **y** (*float*) – Second vector component of the quaternion.
- **z** (*float*) – Third vector component of the quaternion.

Returns

A list containing [roll, pitch, yaw] angles in radians.

Return type

list

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
quaternion = r.quaternion_to_rpy(0.85,0,0.52,0)
```

read_safeio(**args*, ***kwargs*)

Method to read the values of configurable/safe Input/Outputs

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
safe_io = r.read_safeio(1)
print(safe_io) #True/False
```

record_path(**args*, ***kwargs*)

To move the robot in a pre-recorded path. This method takes the following arguments/keyword arguments:

Parameters

- **is_motion** – True if constant velocity is needed during the motion. (type: bool, required: Yes)
- **file_location** – Location of the recorded path file. (type: str, required: Yes)
- **speed** – Linear Speed. (type: float, units: m/sec, required: Yes)
- **current_joint_angles** – Current Robot Joint Configuration. (type: List of Joint Values - float, units: radians, default_value: Joint Configuration obtained from Robot Status method, required: No)
- **safety_toggle** – Safety toggle. (type: Bool - True/False, units: N/A, default_value: value of the safety toggle in Program screen if not set, required: No)

  If set to True, Max speed is slashed to 25% False - No reduction in already set max speed

Returns

True if motion is executed successfully, False if motion is not executed successfully

Raises

- **WrongMode** – If the robot is in Teach mode, while executing this function
- **ConnectionError** – If there is a failure to connect to the robot.

- **UnfeasibleMotion** – If motion is not possible with the given input parameters
- **InterruptedError** – If there is any interruption during the execution of the motion

release()

To release the grasped object

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.release()
```

remove_load()

Remove the load which was set in set_load and set load back to the tool parameters.

Returns
List containing the updated tool properties. i.e . [tool_mass,roll_offset,pitch_offset,yaw_offset,x_offset,y_offset,z_offset,COG_x,COG_y,COG_z,Ixx,Iyy, Izz, Ixy, Ixz, and Iyz]

Return type
List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.remove_load()
```

reset_collision()

Reset the collision state of the robot.

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
r.reset_collision()
```

reset_control()

To restart the control software running on the robot. Equivalent to reset control option from Teach pendant

Returns
True if the control reset was successful, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.reset_control():
    print("Control reset successful.")
else:
    print("Control reset failed.")
```

reset_errors(**args*, ***kwargs*)

Method to reset the errors on the robot

Sample Usage:

```
import time
from neurapy.robot import Robot
r = Robot()
r.reset_error()
```

reset_warnings()

Method to reset/clear the warnings on the robot

Sample Usage:

```
import time
from neurapy.robot import Robot
r = Robot()
r.reset_warnings()
```

robot_status(**args*, ***kwargs*)

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following functions instead:
>
> - *get_current_joint_angles()*
> - *get_current_joint_angles_with_timestamp()*
> - *get_current_cartesian_pose()*
> - *get_current_cartesian_pose_with_timestamp()*
> - *get_current_joint_torques()*
> - *get_current_joint_torques_with_timestamp()*
> - *get_current_joint_torques_with_timestamp()*
> - *get_current_joint_velocities_with_timestamp()*
> - *get_current_load_side_encoder_values()*
> - *get_current_load_side_encoder_values_with_timestamp()*
> - *get_current_motor_side_encoder_values()*
> - *get_current_motor_side_encoder_values_with_timestamp()*

Query the current status of the robot.

Returns

- cartesianPosition (Dict[str, Any]): Robot pose.
- jointAngles (Dict[str, Any]): Joint positions.
- jointTorques (Dict[str, Any]): Joint torque.
- commandedjointAngle (Dict[str, Any]): Last commanded joint angle.
- taskStateTwist (Dict[str, Any]): Task state twist linear.

- loadSideEncValue (Dict[str, Any]): Primary encoder value.
- motorSideEncValue (Dict[str, Any]): Secondary encoder value.

Return type
Tuple[Dict[str, Any], Dict[str, Any], Dict[str, Any], Dict[str, Any], Dict[str, Any], Dict[str, Any]]

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
c = r.robot_status('jointAngles')
```

rpy_to_quaternion(*r*, *p*, *y*)

Convert roll-pitch-yaw (RPY) angles to a quaternion representation.

Parameters

- **r** (*float*) – Roll angle in radians.
- **p** (*float*) – Pitch angle in radians.
- **y** (*float*) – Yaw angle in radians.

Returns
A list representing the quaternion [w,x, y, z].

Return type
list

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
rpy = r.rpy_to_quaternion(0,1.1,0)
```

save_point(*point_data*)

Stores a point in the database.

Parameters
point_data (*dict*) – A dictionary containing the data for the new point.

Returns
True if the point was added successfully

Return type
bool

Raises

- **ConnectionError** – If a connection to the remote database cannot be established.
- **ValueError** – If point creation failed with the given input data

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
point_data = {
    "type": "Point",
    "visibility": True,
```

(continues on next page)

(continued from previous page)

```
    "description": "Description of your point",
    "originX": 0, # X-Coordinate of your point
    "originY": -0.347247, # Y-Coordinate of your point
    "originZ": 0.139187, # Z-Coordinate of your point
    "originA": 3.1416, # R of your point
    "originB": -0.6458, # P of your point
    "originC": -1.5708, # Y of your point
    "offsetX": 0,
    "offsetY": 0,
    "offsetZ": 0,
    "offsetA": 0,
    "offsetB": 0,
    "offsetC": 0,
    "a1": 1.5708001839725434, # joint 1 of your point
    "a2": 0, # joint 2 of your point
    "a3": -2.49582083, # joint 3 of your point
    "a4": 0, # joint 4 of your point
    "a5": 0, # joint 5 of your point
    "a6": 0, # joint 6 of your point
    "name": "name of your point"
    "__v": 0
}
result = r.save_point(point_data)
print(result)
```

servo_j(**args*, ***kwargs*)

Method to do Servo in Joint Space

Args: - servo_target_position ([int]*dof) : List of joint target position in rad. Length of list should be equal to degree of freedom - servo_target_velocity ([int]*dof) : List of joint target velocity in rad/s. Length of list should be equal to degree of freedom - servo_target_acceleration ([int]*dof) : List of joint target acceleration in rad/s^2. Length of list should be equal to degree of freedom

Returns: - Servo warning and error codes

Sample Usage

```
from neurapy.robot import Robot
import time
from ruckig import InputParameter, OutputParameter, Result, Ruckig

r = Robot()

#Switch to external servo mode
r.activate_servo_interface('position')

dof = 6

otg = Ruckig(dof, 0.001)  # DoFs, control cycle
inp = InputParameter(dof)
out = OutputParameter(dof)

inp.current_position = r.get_current_joint_angles()
```

(continues on next page)

(continued from previous page)

```
inp.current_velocity = [0.]*dof
inp.current_acceleration = [0.]*dof

inp.target_position = [0., 0., 0., 0., 0., 0.]
inp.target_velocity = [0.]*dof
inp.target_acceleration = [0.]*dof

inp.max_velocity = [0.5]*dof
inp.max_acceleration = [3]*dof
inp.max_jerk = [10.]*dof
res = Result.Working



while res == Result.Working:
    '''
    Error code is returned through Servo.
    '''
    error_code = 0
    if(error_code < 3):

        res = otg.update(inp, out)

        position = out.new_position
        velocity = out.new_velocity
        acceleration = out.new_acceleration

        error_code = r.servo_j(position, velocity, acceleration)
        scaling_factor = r.get_servo_trajectory_scaling_factor()
        out.pass_to_input(inp)
        time.sleep(0.001)
    else:
        print("Servo in error, error code, ", error_code)
        break
r.deactivate_servo_interface()

r.stop()
```

servo_x(**args*, ***kwargs*)

Method to do Servo in Cartesian Space

Args: - servo_target_position ([int]*7) : List of cartesian target position in m. Length of list should be equal to 7 (X, Y, Z, qw, qx, qz, qz) - servo_target_velocity ([int]*7) : List of cartesian target velocity in m/s. Length of list should be equal to 7 - servo_target_acceleration ([int]*7) : List of cartesian target acceleration in m/s^2. Length of list should be equal 7. Not used in Current Version - gain parameter (double) : ServoX gain parameter, used for Propotional controller (should be between [0.2, 100])

Returns: - Servo warning and error codes

Sample Usage

```
from neurapy.robot import Robot
import time
```

(continues on next page)

(continued from previous page)

```
from ruckig import InputParameter, OutputParameter, Result, Ruckig
import copy

r = Robot()

#Switch to external servo mode
r.activate_servo_interface('position')

cart_pose_length = 7 #X,Y,Z,qw,qx,qy,qz

otg = Ruckig(cart_pose_length, 0.001)  # control cycle
inp = InputParameter(cart_pose_length)
out = OutputParameter(cart_pose_length)

inp.current_position = r.get_current_cartesian_pose()
inp.current_velocity = [0.]*cart_pose_length
inp.current_acceleration = [0.]*cart_pose_length

target = copy.deepcopy(inp.current_position)
target[0] += 0.2 # Move 200mm in X direction
inp.target_position = target
inp.target_velocity = [0.]*cart_pose_length
inp.target_acceleration = [0.]*cart_pose_length

inp.max_velocity = [0.5]*cart_pose_length
inp.max_acceleration = [3]*cart_pose_length
inp.max_jerk = [10.]*cart_pose_length
res = Result.Working

servox_proportional_gain = 25

while res == Result.Working:
    '''
    Error code is returned through Servo.
    '''
    error_code = 0
    if(error_code < 3):

        res = otg.update(inp, out)

        position = out.new_position
        velocity = out.new_velocity
        acceleration = out.new_acceleration

        error_code = r.servo_x(position, velocity, acceleration, servox_
↪proportional_gain)
        scaling_factor = r.get_servo_trajectory_scaling_factor()
        out.pass_to_input(inp)
        time.sleep(0.001)
    else:
        print("Servo in error, error code, ", error_code)
        break
```

(continues on next page)

(continued from previous page)

```
r.deactivate_servo_interface()

r.stop()
```

set_analog_output(*io_name*, *target_value*)

Set analog outputs of the control box.Please use get_io_configuration function to get the number of available IOs.

Parameters

- **io_name** (*int*) – The name of the analog output.
- **target_value** – The target value to set for the analog output.

Returns

The result of the IO operation.

Return type

bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_set = r.set_analog_output(1,3.2)
print(io_set)
```

set_digital_output(*io_name*, *target_value*)

Set digital outputs of the control box.Please use get_io_configuration function to get the number of available IOs.

Parameters

- **io_name** (*int*) – The number of the digital output.
- **target_value** – The target value to set for the digital output.

Raises: - TypeError: If io_name is not an int value or target_value is not a boolean value

Returns

The result of the IO operation.

Return type

bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_set = r.set_digital_output(1,True)
print(io_set)
```

set_encoder_offsets(*list_of_encoder_offsets*)

> **Warning:** This function sets the encoder values of joints. Incorrect usage of this function, such as setting incorrect encoder values, may result in serious unexpected behavior of the robot.

Sets the encoder offsets for the robot's joints.

Parameters
`list_of_encoder_offsets` (*list*) – A list of encoder offset values(int) for each joint. The length of the list must be equal to the number of joints on the robot.

Returns
Returns True if the encoder offsets were successfully updated.

Return type
bool

Raises

- `ValueError` – If the length of the provided encoder offsets list is not equal to the number of joints on the robot.
- `ConnectionError` – If there is a failure to connect to the robot or the database, or if there is an issue updating the encoder offsets.

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
r.set_encoder_offsets([#encoder offsets from the provided robot manual/
↪data sheet])
```

set_gravity_vector(*gravity_vector*)

> **Warning:** This function sets the gravity vector of the robot. Incorrect usage of this function, such as setting incorrect gravity vector, may result in unexpected behavior of the robot.

To set a new gravity vector values on the robot.

Parameters
`List` – A list containing the gravity vector values along x,y,z directions

Raises

- `ConnectionError` – If there is a failure to connect to the robot's database.
- `ValueError` – If the provided input is not correct(length and magnitude wise)

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
r.set_gravity_vector([0,0,9.8])
```

set_joint_acceleration(*acceleration*)

Set the acceleration value for joint motion.

Parameters: acceleration (float): The acceleration value for joint motion, ranging from 0 to 100.(Percentage of maximum angular acceleration)

Raises: - TypeError: If acceleration is not a numeric value (int or float). - ValueError: If acceleration is outside the range of 0 to 100.

Returns: None

set_joint_speed(*speed*)

Set the speed value for joint motion.

Parameters: speed (float): The speed value for joint motion, ranging from 0 to 100.(Percentage of maximum angular speed)

Raises: - TypeError: If speed is not a numeric value (int or float). - ValueError: If speed is outside the range of 0 to 100. Returns: None

set_linear_acceleration(*acceleration*)

Set the acceleration value for linear motion.

Parameters: acceleration (float): The acceleration value for linear motion, ranging from 0.0 to 1.0.(m/s2)

Raises: - TypeError: If acceleration is not a numeric value (int or float). - ValueError: If acceleration is outside the range of 0.0 to 1.0.

Returns: None

set_linear_speed(*speed*)

Set the speed value for linear motion.

Parameters: speed (float): The speed value for linear motion, ranging from 0.0 to 1.0.(m/s)

Raises: - TypeError: If speed is not a numeric value (int or float). - ValueError: If speed is outside the range of 0.0 to 1.0.

Returns: None

set_load(*load_mass*, *load_cog_x*, *load_cog_y*, *load_cog_z*)

Set a load, e.g. an object which is grasped. The tool parameters (mass, center of gravity) are updated.

Parameters

- **load_mass** (*float*) – mass of load
- **load_cog_x** (*float*) – center of gravity of load with respect to the tcp frame in x direction
- **load_cog_y** (*float*) – center of gravity of load with respect to the tcp frame in y direction
- **load_cog_z** (*float*) – center of gravity of load with respect to the tcp frame in z direction

Returns

List containing the updated tool properties. i.e . [tool_mass,roll_offset,pitch_offset,yaw_offset,x_offset,y_offset,z_offset,COG_x,COG_y,COG_z,Ixx,Iyy, Izz, Ixy, Ixz, and Iyz]

Return type

List

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.set_load(load_mass=1.1, load_cog_x=0.02, load_cog_y=0.03, load_cog_z=0.
↪04)
```

set_mode(**args*, ***kwargs*)

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following functions instead:

- *switch_to_teach_mode()*
- *switch_to_automatic_mode()*
- *switch_to_semi_automatic_mode()*

Toggle the robot modes (Teach/Automatic/SemiAutomatic).

Parameters
`value` (*str*) – The mode to set. - 'Teach': Change to teach mode. - 'Automatic': Change to Automatic mode. - 'SemiAutomatic': Change to semi-automatic mode.

Returns
True if the operation is executed successfully, False otherwise.

Return type
bool

Sample Usage:

```
import time
from neurapy.robot import Robot
r = Robot()
r.set_mode("Teach")
time.sleep(1)
r.set_mode("Automatic")
time.sleep(1)
r.set_mode("SemiAutomatic")
```

set_opcua_msg(**args*, ***kwargs*)

Send a message to the OPC UA server running on the control box.

Parameters

- **message** (*str*) – The message to be sent to the OPC UA server.
- **opcua_register_index** (*int*) – The index of the OPC UA register where the message will be stored.

Returns
True if the message is successfully sent and stored in the specified register, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
message = "Hello, OPC UA Server!"
register_index = 1  # Replace with the index of the target OPC UA␣
↪register
if r.set_opcua_msg(message, register_index):
    print("Message sent and stored in OPC UA register successfully.")
else:
    print("Failed to send the message or store it in the OPC UA register.
↪")
```

set_override(*value*)

Set the override value.

Parameters

value (*int*) – The new override value to set.

Returns

True if the override value is successfully set, False otherwise.

Return type

bool

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
new_override_value = 0.5  # it should be from 0 to 1
if r.set_override(new_override_value):
    print(f"Override value set to {new_override_value} successfully.")
else:
    print("Failed to set the override value.")
```

set_sim_real(**args*, ***kwargs*)

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following functions instead:
>
> - *switch_to_real()*
> - *switch_to_simulation()*

Method to toggle the sim/real context of the control software

Sample Usage:

```
import time
from neurapy.robot import Robot
r = Robot()
r.set_sim_real(True)
```

set_tool(**args*, ***kwargs*)

Notify the gripper/tool change to software components after the physical change.

Parameters

tool_name (*str*) – The name of the tool defined from the GUI.

Returns

True if the operation succeeds, False if the operation fails.

Return type

bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.set_tool(tool_name="name given during tool creation in GUI")
```

set_tool_digital_output(*io_number*, *target_value*)

Set the target value on the specified tool digital output.Please use get_io_configuration function to get the number of available IOs.

Parameters

- **io_number** (*int*) – Number of the tool digital output to be triggered
- **target_value** (*bool*) – The target value to set for the tool digital output.

Returns

The result of the IO operation.

Return type

bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_set = r.set_tool_digital_output(1,True) # To set tool digital output␣
↪1 to high
print(io_set)
```

set_tool_digital_outputs(*target_value*)

Set digital outputs of the tool.Please use get_io_configuration function to get the number of available IOs.

Parameters

target_value (*List of floats*) – The target value to set for the tool digital outputs.

Returns

The result of the IO operation.

Return type

bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_set = r.set_tool_digital_outputs([0.0,1.0,0.0]) # To set tool digital␣
↪output to high
print(io_set)
```

start_external_interface(**args*, ***kwargs*)

Starts the external interface.

Parameters

- **interface_name** (*str*) – The name of the interface to start.
- ***args** – Additional arguments to be passed to the interface.
- **args[0]** – supported interface types: ethercat, servo, ethernet_ip, ros, opcua
- **args[1]** – supported interface modes: position, torque (for ros, servo only)
- ****kwargs** – Additional keyword arguments to be passed to the interface.

Returns
True if the interface is started successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
# start the servo interface in position mode
r.start_external_interface('servo', 'position')
```

stop(**args*, ***kwargs*)

To stop the robot's motion and to terminate the script execution. This needs to be added at the end of the script(only once) for a proper script termination

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.stop()
```

stop_external_interface(**args*, ***kwargs*)

Stops the external interface.

Parameters

- **interface_name** (*str*) – The name of the interface to stop.
- ***args** – Additional arguments to be passed to the interface.
- **args[0]** – supported interface types: ethercat, servo, ethernet_ip, ros, opcua
- ****kwargs** – Additional keyword arguments to be passed to the interface.

Returns
True if the interface is stopped successfully, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
# stop the servo interface which is currently in activation
r.stop_external_interface('servo')
```

switch_to_automatic_mode()

Switch the robot to Automatic mode.

Returns
True if the robot successfully switches to Automatic mode, False otherwise.

Return type
bool

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
if r.switch_to_automatic_mode():
    print("Robot switched to Automatic mode.")
else:
    print("Failed to switch to Automatic mode.")
```

switch_to_real()

Switches the robot's running context to real mode.

Returns
True if the switch was successful, False otherwise.

Return type
bool

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
r.switch_to_real()
```

switch_to_semi_automatic_mode()

Switch the robot to Semi Automatic mode.

Returns
True if the robot successfully switches to Semi Automatic mode, False otherwise.

Return type
bool

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
if r.switch_to_semi_automatic_mode():
    print("Robot switched to Semi Automatic mode.")
else:
    print("Failed to switch to Semi Automatic mode.")
```

switch_to_simulation()

Switches the robot's running context to simulation mode.

Returns
True if the switch was successful, False otherwise.

Return type
bool

Sample Usage

```
from neurapy.robot import Robot
r = Robot()
r.switch_to_simulation()
```

switch_to_teach_mode()

Switch the robot to Teach mode.

Returns
True if the robot successfully switches to Teach mode, False otherwise.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
if r.switch_to_teach_mode():
    print("Robot switched to Teach mode.")
else:
    print("Failed to switch to Teach mode.")
```

turn_off_free_drive_mode()

To turn off free drive mode :returns: True if the operation is executed successfully, False otherwise. :rtype: bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
r.turn_off_free_drive_mode()
```

turn_off_jog(**args*, ***kwargs*)

Method to disable jogging programatically. This needs to used in conjuction with jog, turn_on_jog methods.

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
r.turn_on_jog(jog_velocity=[0.2, 0.2, 0.2, 0.2, 0.2, 0.2], jog_type=
↪'Joint')
r.jog(set_jogging_external_flag=1)
i = 0
while i < 500:
    r.jog(set_jogging_external_flag=1)
    i += 1
r.turn_off_jog()
```

turn_on_free_drive_mode()

To turn on free drive mode :returns: True if the operation is executed successfully, False otherwise. :rtype: bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
r.turn_on_free_drive_mode()
```

turn_on_jog(**args*, ***kwargs*)

Method to enable jogging programatically. This needs to used in conjuction with jog, turn_off_jog methods.

Args: - jog_velocity (list of float): A list of joint velocities ranging from -1 to 1 for all joints. - jog_type (str): Specifies the type of jogging, which can be 'Cartesian' or 'Joint'.

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
r.turn_on_jog(jog_velocity=[0.2, 0.2, 0.2, 0.2, 0.2, 0.2], jog_type=
↪'Joint')
r.jog(set_jogging_external_flag=1)
i = 0
while i < 500:
    r.jog(set_jogging_external_flag=1)
    i += 1
r.turn_off_jog()
```

unpause(**args*, ***kwargs*)

To unpause the robot's motion.

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.unpause()
```

update_current_tool_parameters(**args*, ***kwargs*)

Update the current gripper/tool parameters in the tool list and the selected tool properties.

Parameters

- **_controlOA;False;True** –
- **Requried** (*if the tool is controlled by controlbox analog outputs;Not*) –
- **_controlOD;False;True** –
- **Required** (*speed;0;;Not*) –
- **_toolOA;False;True** –
- **Required** –
- **_toolOD;False;True** –
- **Required** –
- **tool;Required** (*name;N/A;Name of the*) –
- **gravity** (*autoMeasureZ;0;tool center of*) –
- **gravity** –
- **gravity** –
- **Required** –
- **Required** –

- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **tool;Required** –
- **offCOA;[0** –
- **0** –
- **0** –
- **0** –
- **0** –
- **0** –
- **0** –
- **Required** –
- **outputs** (*onTOD;0;if the tool is controlled via tool digital*) –
- **off** (*offTOD is the pin mapped to turn*) –
- **Required** –
- **offTOA;[0** –
- **Required** –
- **outputs** –
- **off** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **onCOA;[0** –
- **0** –
- **0** –
- **0** –
- **0** –

- **0** –
- **0** –
- **Required** –
- **outputs** –
- **on** (*onTOD is the pin mapped to turn*) –
- **Required** –
- **onTOA;[0** –
- **Required** –
- **outputs** –
- **on** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –

Returns
True if the operation succeeds, False if the operation fails.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.update_current_tool_parameters(offsetZ=0.1, autoM=0.5)
```

update_tool_parameters(**args*, ***kwargs*)

Update the gripper/tool parameters in the tool list with a given tool_name.

Parameters

- **tool_name** (*str*) – The name of the tool defined from the GUI.; Required
- **_controlOA;False;True** –
- **Requried** (*if the tool is controlled by controlbox analog outputs;Not*) –
- **_controlOD;False;True** –
- **Required** (*speed;0;;Not*) –
- **_toolOA;False;True** –
- **Required** –
- **_toolOD;False;True** –
- **Required** –

- **tool;Required** (*name;N/A;Name of the*) –
- **gravity** (*autoMeasureZ;0;tool center of*) –
- **gravity** –
- **gravity** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **tool;Required** –
- **offCOA;[0** –
- **0** –
- **0** –
- **0** –
- **0** –
- **0** –
- **0** –
- **Required** –
- **outputs** (*onTOD;0;if the tool is controlled via tool digital*) –
- **off** (*offTOD is the pin mapped to turn*) –
- **Required** –
- **offTOA;[0** –
- **Required** –
- **outputs** –
- **off** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –

- **Required** –
- **onCOA;[0** –
- **0** –
- **0** –
- **0** –
- **0** –
- **0** –
- **0** –
- **Required** –
- **outputs** –
- **on** (*onTOD is the pin mapped to turn*) –
- **Required** –
- **onTOA;[0** –
- **Required** –
- **outputs** –
- **on** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –
- **Required** –

Returns
True if the operation succeeds, False if the operation fails.

Return type
bool

Sample Usage:

```
from neurapy.robot import Robot
r = Robot()
r.update_tool_parameters(tool_name="name given during tool creation in␣
↪GUI", offsetZ=0.1, autoM=0.5)
```

wait(**args*, ***kwargs*)

Wait for a specified signal.

Parameters

- **port_name** (*str*) – The name or identifier of the tool-specific analog input.
- **expected_value** (*float*) – The expected value of the analog input. If None, the function waits for any change in the input value

Returns
True if the specified condition or event is met, False otherwise.

Return type
bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_wait = r.wait(port_name="DI_2", expected_value=1.0, wait_for_
↪signal=True, wait_time=0.0) #waits till digital input 2 is True
print(io_wait)
```

wait_for_analog_input(*io_name*, *expected_value*, *wait_time=0.0*)

Wait for an analog input signal to match the expected value.

Parameters

- **io_name** (*str*) – The name or identifier of the analog input.
- **expected_value** (*float*) – The expected value of the analog input.
- **wait_time** (*float*) – waits for this amount of time(in secs), before returning. default value is 0

Returns
True if the analog input matches the expected value, False otherwise.

Return type
bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_wait = r.wait_for_analog_input(1,2.3) #waits till analog input 1 is 2.
↪3
print(io_wait)
```

wait_for_digital_input(*io_name*, *expected_value*, *wait_time=0.0*)

Wait for a digital input signal to match the expected value.

Parameters

- **io_name** (*str*) – The name or identifier of the digital input.
- **expected_value** (*bool*) – The expected value of the digital input
- **wait_time** (*float*) – waits for this amount of time(in secs), before returning. default value is 0

Returns
True if the digital input matches the expected value, False otherwise.

Return type
bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_wait = r.wait_for_digital_input(1,True) #waits till digital input 1␣
↪is True
print(io_wait)
```

wait_for_digital_input_timer_off_delay(*io_name*, *delay=0.0*)

Wait for a given delay and returns after the given digital input signal reaches low .

Parameters

- **io_name** (*str*) – The name or identifier of the digital input.
- **wait_time** (*float*) – waits for this amount of time(in secs), before returning. default value is 0

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_wait = r.wait_for_digital_input_timer_off_delay(1,10) #waits for␣
↪10sec and returns after digital input 1 reaches low
print(io_wait)
```

wait_for_digital_input_timer_on_delay(*io_name*, *delay=0.0*)

Wait for a given delay and returns after the given digital input signal reaches high .

Parameters

- **io_name** (*str*) – The name or identifier of the digital input.
- **wait_time** (*float*) – waits for this amount of time(in secs), before returning. default value is 0

Returns

True if the digital input matches the expected value, False otherwise.

Return type

bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_wait = r.wait_for_digital_input_timer_on_delay(1,10) #waits for 10sec␣
↪and returns after digital input 1 reaches high
print(io_wait)
```

wait_for_tool_analog_input(*io_name*, *expected_value*)

Wait for a tool-specific analog input signal to match the expected value.

Parameters

- **io_name** (*str*) – The name or identifier of the tool-specific analog input.
- **expected_value** (*float*) – The expected value of the analog input. If None, the function waits for any change in the input value.

Returns
True if the tool-specific analog input matches the expected value, False otherwise.

Return type
bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_wait = r.wait_for_tool_analog_input(1,2.3v) #waits till tool analog␣
↪input 1 is 2.3v
print(io_wait)
```

wait_for_tool_digital_input(*io_name*, *expected_value*, *wait_time=0.0*)

Wait for a tool-specific digital input signal to match the expected value.

Parameters

- **io_name** (*str*) – The name or identifier of the tool-specific digital input.
- **expected_value** (*bool*) – The expected value of the digital input.
- **wait_time** (*float*) – waits for this amount of time(in secs), before returning. default value is 0

Returns
True if the tool-specific digital input matches the expected value, False otherwise.

Return type
bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_wait = r.wait_for_tool_digital_input(1,True) #waits till tool digital␣
↪input 1 is True
print(io_wait)
```

wait_for_tool_digital_input_timer_off_delay(*io_name*, *delay=0.0*)

Wait for a given delay and returns after the given tool digital input signal reaches low .

Parameters

- **io_name** (*str*) – The name or identifier of the tool digital input.
- **wait_time** (*float*) – waits for this amount of time(in secs), before returning. default value is 0

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_wait = r.wait_for_tool_digital_input_timer_off_delay(1,10) #waits for␣
↪10sec and returns after tool digital input 1 reaches low
print(io_wait)
```

wait_for_tool_digital_input_timer_on_delay(*io_name*, *delay=0.0*)

Wait for a given delay and returns after the given tool digital input signal reaches high .

Parameters

- **io_name** (*str*) – The name or identifier of the tool digital input.
- **wait_time** (*float*) – waits for this amount of time(in secs), before returning. default value is 0

Returns
True if the digital input matches the expected value, False otherwise.

Return type
bool

Sample Usage

```
import time
from neurapy.robot import Robot
r = Robot()
io_wait = r.wait_for_tool_digital_input_timer_on_delay(1,10) #waits for␣
↪10sec and returns after tool digital input 1 reaches high
print(io_wait)
```

zero_g(**args*, ***kwargs*)

> **Warning:** This function is deprecated and will be removed in the next version. Please use the following functions instead:
>
> - *turn_on_free_drive_mode()*
> - *turn_off_free_drive_mode()*

Toggle the freedrive/Gravity compensation mode.

Parameters
value (*str*) – Specifies the mode action. - 'on' : Turn on the freedrive mode. - 'off' : Turn off the freedrive mode.

Returns
True if the operation is executed successfully, False otherwise.

Return type
bool

Sample Usage:

```
import time
from neurapy.robot import Robot
r = Robot()
r.zero_g("on")
time.sleep(1)
r.zero_g("off")
```

CHAPTER

FIVE

EXAMPLES FOR LARA 5

1. **Robot Object Initialization**

```
from neurapy.robot import Robot

r = Robot()
print(r.robot_name)
print(r.dof)
print(r.platform)
print(r.payload)
print(r.kURL)
print(r.robot_urdf_path)
print(r.current_tool)
print(r.connection)
print(r.version)
```

2. Move joint

```
from neurapy.robot import Robot

r = Robot()
joint_property = {
    "speed": 50.0,
    "acceleration": 50.0,
    "safety_toggle": True,
    "target_joint": [
        [
            2.5995838308821924,
            0.24962416292345468,
            -1.8654403327490414,
            0.04503286318691005,
            -1.1740563715454926,
            0.10337461241185522
        ],
        [
            2.1372059994827075,
            0.24939733788589463,
            -1.8651270179353125,
            0.044771940725327274,
            -1.173860821592129,
            0.10315646291502645
        ],
```

(continues on next page)

(continued from previous page)

```
        [
            1.9180047887810003,
            -0.24855170101601043,
            -1.3680228668892351,
            0.12404421791100637,
            -1.1914147150222498,
            -0.13255713717112075
        ]
    ],
    "current_joint_angles": r.robot_status("jointAngles")
}
r.move_joint(**joint_property)
```

3. Move linear

```
from neurapy.robot import Robot

r = Robot()
linear_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "blend_radius": 0.005,
    "target_pose": [
        [
            0.3287228886,
            -0.1903355329,
            0.4220780352,
            0.08535207028439847,
            -2.797181496822229,
            2.4713321627410485
        ],
        [
            0.2093363791501374,
            -0.31711250784165884,
            0.422149168855134,
            -3.0565555095672607,
            -0.3447442352771759,
            -1.1323236227035522
        ],
        [
            0.2090521916195534,
            -0.5246753336643587,
            0.4218773613553828,
            -3.0569007396698,
            -0.3448921740055084,
            -1.1323626041412354
        ],
        [
            0.3287228886,
            -0.1903355329,
            0.4220780352,
            0.08535207028439847,
```

(continues on next page)

(continued from previous page)

```
            -2.797181496822229,
            2.4713321627410485
        ]
    ],
    "current_joint_angles": r.robot_status("jointAngles")
}
r.move_linear(**linear_property)
```

4. Move circular

```
from neurapy.robot import Robot

r = Robot()
circular_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "target_pose": [
        [
            0.3744609827431085,
            -0.3391784988266481,
            0.23276604279256016,
            3.14119553565979,
            -0.00017731254047248513,
            -0.48800110816955566
        ],
        [
            0.37116786741831503,
            -0.19686307684994242,
            0.23300456855796453,
            3.141423225402832,
            -0.00020668463548645377,
            -0.48725831508636475
        ],
        [
            0.5190337951593321,
            -0.1969996948428492,
            0.23267853691809767,
            3.1414194107055664,
            -0.00017726201622281224,
            -0.48750609159469604
        ]
    ],
    "current_joint_angles": r.robot_status("jointAngles")
}
r.move_circular(**circular_property)
```

5. Move composite

```
from neurapy.robot import Robot

r = Robot()
composite_motion_property = {
```

(continues on next page)

(continued from previous page)

```
    "speed": 0.432,
    "acceleration": 0.2,
    "current_joint_angles": r.robot_status('jointAngles'),
    "commands": [
        {
            "linear": {
                "blend_radius": 0.005,
                "targets": [
                    [
                        -0.000259845199876027,
                        -0.5211437049195536,
                        0.4429382717719519,
                        3.14123272895813,
                        -0.0007908568368293345,
                        -1.570908784866333
                    ],
                    [
                        -0.16633498440272945,
                        -0.5201452059140722,
                        0.4427486025872017,
                        3.140937089920044,
                        -0.0005319403717294335,
                        -1.571555495262146
                    ]
                ]
            }
        },
        {
            "circular": {
                "targets": [
                    [
                        -0.16633498440272945,
                        -0.5201452059140722,
                        0.4427486025872017,
                        3.140937089920044,
                        -0.0005319403717294335,
                        -1.571555495262146
                    ],
                    [
                        -0.16540090985202305,
                        -0.3983552679378624,
                        0.44267608017426174,
                        3.1407113075256348,
                        -0.00036628879024647176,
                        -1.5714884996414185
                    ],
                    [
                        -0.33446498807559716,
                        -0.3989652352814891,
                        0.4421152856242009,
                        3.1402060985565186,
                        0.00030071483342908323,
```

(continues on next page)

(continued from previous page)

```
                        -1.572899580001831
                    ]
                ]
            }
        }
    ]
}
r.move_composite(**composite_motion_property)
```

6. Move trajectory

```
from neurapy.robot import Robot

r = Robot()
trajectory_motion_property = {
    "current_joint_angles": r.robot_status('jointAngles'),
    "timestamps": [
        0.01, 0.02, 0.03
    ],
    "target_joint": [
        [0.0531381, 0.157485, -0.100272, 1.29569, -5.99211e-05, 0.700957, -
↪0.000371511],
        [-0.208897, 0.461728, -0.433937, 1.66485, -5.99211e-05, 0.700382, -
↪0.000383495],
        [0.0120801, 0.621298, 0.0149563, 1.67381, 5.99211e-05, 0.687403, -0.
↪000359527]
    ]
}
r.move_trajectory(**trajectory_motion_property)
```

7. Move recorded path

```
from neurapy.robot import Robot

r = Robot()
recorded_path_property = {
    "is_motion": False,
    "file_location": "<path to file location>",
    "speed": 0.25,
    "use_recordings_filter": False,
    "current_joint_angles": r.robot_status('jointAngles'),
}
r.record_path(**recorded_path_property)
```

8. Power

```
from time import sleep
from neurapy.robot import Robot

r = Robot()
r.power('on')
sleep(2)
r.power('off')
```

9. Motion status

```
from neurapy.robot import Robot

r = Robot()
print(r.motion_status())
```

10. Program status

```
from neurapy.robot import Robot

r = Robot()
print(r.program_status())
```

11. Warnings

```
from neurapy.robot import Robot

r = Robot()
print(r.get_warnings())
```

12. Errors

```
from neurapy.robot import Robot

r = Robot()
print(r.get_errors())
```

12. Get point

```
from neurapy.robot import Robot
r = Robot()
point = r.get_point("P1")
print(point)
```

13. IOs

```
import time
from neurapy.robot import Robot
r = Robot()
io_get = r.io("get", io_name = "DO_1")
io_set = r.io("set", io_name = "DO_2", target_value = True)
print(io_get,io_set)
```

14. Gripper

```
from time import sleep
from neurapy.robot import Robot
r = Robot()
r.gripper('on')
sleep(2)
r.gripper('off')
```

15. Set tool

```
from neurapy.robot import Robot
r = Robot()
r.set_tool(tool_name="name given during tool creation in GUI")
```

16. Robot Status

```
from neurapy.robot import Robot
r = Robot()
c = r.robot_status('jointAngles')
"""
            "cartesianPosition" -  returns Robot Pose in xyzwxyz format(rotation in␣
↪quaternion)
            "jointAngles" -  returns Joint Positions
            "jointTorques" -  returns Joint Torque
            "commandedjointAngle" - returns last commanded joint angle
            "taskStateTwist" - returns taskStateTwist linear
            "loadSideEncValue" - returns Primary Encoder Value
            "motorSideEncValue" - reutrns Secondary Encoder Value
"""
```

17. Zero G

```
import time
from neurapy.robot import Robot
r = Robot()
r.zero_g("on")
time.sleep(1)
r.zero_g("off")
```

18. Forward/Inverse kinematics

```
from neurapy.robot import Robot
r = Robot()
target_pose = r.ik_fk("fk", target_angle = [0.2,0.2,0.2,0.2,0.2,0.2])
target_angle = r.ik_fk("ik", target_pose = [0.140448, -0.134195, 1.197456, 3.1396, -
↪0.589, -1.025],current_joint = [-1.55, -0.69, 0.06, 1.67, -0.02, -1.57, 0.11])
```

19. Override

```
import time
from neurapy.robot import Robot
r = Robot()
override_value = r.override("get")
print("override_value : " + str(override_value))
time.sleep(1)

override_value = r.override("set", target_value = 0.4)
print("Setting Override to 0.4")
time.sleep(1)

override_value = r.override("get")
print("override_value : " + str(override_value))
```

20. Set mode

```
import time
from neurapy.robot import Robot
r = Robot()
r.set_mode("Teach")
time.sleep(1)
r.set_mode("Automatic")
time.sleep(1)
r.set_mode("SemiAutomatic")
```

21. Get mode

```
from neurapy.robot import Robot
r = Robot()
mode = r.get_mode()
print(mode)
```

22. Reset Error

```
import time
from neurapy.robot import Robot
r = Robot()
r.reset_error()
```

23. Get tools

```
import time
from neurapy.robot import Robot
r = Robot()
tools_data = r.get_tools()
```

24. Create tool

```
from neurapy.robot import Robot
r = Robot()
tool_data = { '_control0A': False,
 '_control0D': False,
 '_tool0A': False,
 '_tool0D': False,
 'autoM': 0,
 'autoMeasureX': 0,
 'autoMeasureY': 0,
 'autoMeasureZ': 0,
 'closeInput': 0,
 'cmdID': 16,
 'description': 'Tool Description',
 'force': 0,
 'gripper': '',
 'grippertype': 'Standard Gripper',
 'inertiaXX': 0,
 'inertiaXY': 0,
 'inertiaXZ': 0,
 'inertiaYY': 0,
 'inertiaYZ': 0,
```

(continues on next page)

(continued from previous page)

```
 'inertiaZZ': 0,
 'name': 'NoTool',
 'offCOA': [0, 0, 0, 0, 0, 0, 0, 0],
 'offCOD1': 0,
 'offCOD2': 0,
 'offTOA': [0, 0],
 'offTOD': 0,
 'offsetA': 0,
 'offsetB': 0,
 'offsetC': 0,
 'offsetX': 0,
 'offsetY': 0,
 'offsetZ': 0,
 'onCOA': [0, 0, 0, 0, 0, 0, 0, 0],
 'onCOD1': 0,
 'onCOD2': 0,
 'onTOA': [0, 0],
 'onTOD': 0,
 'openInput': 0,
 'portID': '',
 'protocol': 0,
 'robot_type': 'Tool',
 'slaveID': 0,
 'speed': 0}

tools_data = r.create_tool(tool_data)
```

25. Get encode offsets

```
from neurapy.robot import Robot
r = Robot()
encoder_offsets = r.get_encoder_offsets()
```

26. Encoder to radian values

```
from neurapy.robot import Robot
r = Robot()
encoder_ticks = r.robot_status('loadSideEncValue')
joint_angles = r.encoder2rad(encoder_ticks)
```

27. Quaternion to roll pitch yaw

```
from neurapy.robot import Robot
r = Robot()
rpy = r.quaternion_to_rpy(0.85,0,0.52,0)
```

28. Roll pitch yaw to quaternion

```
from neurapy.robot import Robot
r = Robot()
quaternion = r.rpy_to_quaternion(0,1.1,0)
```

29. Get zerog status

```
import time
from neurapy.robot import Robot
r = Robot()
status = r.get_zerog_status()
```

30. Get reference frame

```
from neurapy.robot import Robot
r = Robot()
frame = r.get_reference_frame("tool_frame")
print(frame)
```

31. Jogging

```
from neurapy.robot import Robot

robot = Robot()
"""
jog_velocity - velocity ranging from [-1,1] for all joints
jog_type - can be either cartesian or joint jogging
turn_on_jog changes the state from interal jogging (from GUI) to external jogging
"""
robot.turn_on_jog(jog_velocity=[0.2, 0.2, 0.2, 0.2, 0.2, 0.2], jog_type='Joint')

# command to set flag for jogging in external mode.
robot.jog(set_jogging_external_flag = 1)
i = 0

"""
Requires minimum number of cycles in the loop for performing jogging.
Depends upon jogging velocity, override.
"""
while(i < 500):
    """
    command to set flag for jogging in external mode. This command has to be used␣
↪each time
    external jog command has to be sent
    """
    robot.jog(set_jogging_external_flag = 1)
    i+=1

"""
Change the state from external jog to internal jog  (GUI) and sets all other
external parameters to false
"""
robot.turn_off_jog()
```

32. Get reference frame with offset

```
from neurapy.robot import Robot
r = Robot()
x_offset = 0.02 # in meters
y_offset = 0.1 # in meters
```

(continues on next page)

(continued from previous page)

```
z_offset = 0.0 # in meters
frame = r.get_reference_frame_with_offset("world",[x_offset,y_offset,z_offset])
print(frame)
```

33. Get TCP pose

```
from neurapy.robot import Robot
r = Robot()
tcp_pose = r.get_tcp_pose()
print(tcp_pose)
```

34. Get sim or real

```
from neurapy.robot import Robot
r = Robot()
context = r.get_sim_or_real()
print(context) #Real/Simulation
```

35. Read safeio

```
from neurapy.robot import Robot
r = Robot()
safe_io = r.read_safeio(1)
print(safe_io) #True/False
```

36. Servo J

```
from neurapy.robot import Robot
import time
from ruckig import InputParameter, OutputParameter, Result, Ruckig

r = Robot()

#Switch to external servo mode
r.activate_servo_interface('position')

dof = 6

otg = Ruckig(dof, 0.001)  # DoFs, control cycle
inp = InputParameter(dof)
out = OutputParameter(dof)

inp.current_position = r.get_current_joint_angles()
inp.current_velocity = [0.]*dof
inp.current_acceleration = [0.]*dof

inp.target_position = [0., 0., 0., 0., 0., 0.]
inp.target_velocity = [0.]*dof
inp.target_acceleration = [0.]*dof

inp.max_velocity = [0.5]*dof
inp.max_acceleration = [3]*dof
inp.max_jerk = [10.]*dof
```

(continues on next page)

(continued from previous page)

```
res = Result.Working

while res == Result.Working:
    '''
    Error code is returned through Servo.
    '''
    error_code = 0
    if(error_code < 3):

        res = otg.update(inp, out)

        position = out.new_position
        velocity = out.new_velocity
        acceleration = out.new_acceleration

        error_code = r.servo_j(position, velocity, acceleration)
        scaling_factor = r.get_servo_trajectory_scaling_factor()
        out.pass_to_input(inp)
        time.sleep(0.001)
    else:
        print("Servo in error, error code, ", error_code)
        break
r.deactivate_servo_interface()

r.stop()
```

37. Servo X

```
from neurapy.robot import Robot
import time
from ruckig import InputParameter, OutputParameter, Result, Ruckig
import copy

r = Robot()

#Switch to external servo mode
r.activate_servo_interface('position')

cart_pose_length = 7 #X,Y,Z,qw,qx,qy,qz

otg = Ruckig(cart_pose_length, 0.001)  # control cycle
inp = InputParameter(cart_pose_length)
out = OutputParameter(cart_pose_length)

inp.current_position = r.get_current_cartesian_pose()
inp.current_velocity = [0.]*cart_pose_length
inp.current_acceleration = [0.]*cart_pose_length

target = copy.deepcopy(inp.current_position)
target[0] += 0.2 # Move 200mm in positive X direction
inp.target_position = target
inp.target_velocity = [0.]*cart_pose_length
```

(continues on next page)

(continued from previous page)

```
inp.target_acceleration = [0.]*cart_pose_length

inp.max_velocity = [0.5]*cart_pose_length
inp.max_acceleration = [3]*cart_pose_length
inp.max_jerk = [10.]*cart_pose_length
res = Result.Working

servox_proportional_gain = 25

while res == Result.Working:
    '''
    Error code is returned through Servo.
    '''
    error_code = 0
    if(error_code < 3):

        res = otg.update(inp, out)

        position = out.new_position
        velocity = out.new_velocity
        acceleration = out.new_acceleration

        error_code = r.servo_x(position, velocity, acceleration, servox_
↪proportional_gain)
        scaling_factor = r.get_servo_trajectory_scaling_factor()
        out.pass_to_input(inp)
        time.sleep(0.001)
    else:
        print("Servo in error, error code, ", error_code)
        break
r.deactivate_servo_interface()

r.stop()
```

CHAPTER

SIX

EXAMPLES FOR LARA 8

1. **Robot Object Initialization**

```
from neurapy.robot import Robot

r = Robot()
print(r.robot_name)
print(r.dof)
print(r.platform)
print(r.payload)
print(r.kURL)
print(r.robot_urdf_path)
print(r.current_tool)
print(r.connection)
print(r.version)
```

2. Move joint

```
from neurapy.robot import Robot

r = Robot()
joint_property = {
    "speed": 50.0,
    "acceleration": 50.0,
    "target_joint": [
        previous_joint_angles,
        [
            0.7740276502469322,
            0.13599234975308788,
            1.2223307505101257,
            -0.31901765024691225,
            0.3242476502469122,
            -0.31901765024691225
        ],
        [
            0.30333999999999633,
            -0.5334353004938217,
            1.4419907505101492,
            -0.31901765024691225,
            0.3242476502469122,
            -0.31901765024691225
        ],
```

(continues on next page)

(continued from previous page)

```
        [
            0.059259210493824405,
            0.08133676975309524,
            1.4419907505101492,
            -1.0084715804938613,
            -0.7008323497530976,
            -0.31901765024691225
        ]
    ]
}
r.move_joint(**joint_property)
```

3. Move linear

```
from neurapy.robot import Robot

r = Robot()
r.set_mode("Automatic")
time.sleep(1)

# moving the robot to initial position
joint_property = {
        "speed": 50.0,
        "acceleration": 50.0,
        "safety_toggle": True,
        "target_joint": [
            [
                -0.0007546,
                0.2713685,
                1.2664022,
                -0.0007550,
                1.6046191,
                -0.0007501,
            ]
        ],
    "current_joint_angles":r.robot_status("jointAngles")
    }
r.move_joint(**joint_property)
time.sleep(1)

linear_property = {
        "speed": 0.9,
        "acceleration": 0.2,
        "target_pose": [
            [
                    0.521504613338733,
                    -0.0005120121635832758,
                    0.4429621801014548,
                    3.140838623046875,
                    -0.0007978816283866763,
                    3.1415627002716064
                ],
```

(continues on next page)

(continued from previous page)

```
                [
                    0.5216067833501538,
                    -0.21429644443731097,
                    0.44311804388931564,
                    3.140573024749756,
                    -0.0008633044781163335,
                    3.1414897441864014
                ],
                [
                    0.30336607796720333,
                    -0.2139397968890111,
                    0.44282379715713066,
                    3.140761613845825,
                    -0.0011759602930396795,
                    3.1410810947418213
                ],
                [
                    0.3033097863631349,
                    0.19367482201823044,
                    0.44268805519921206,
                    -3.1399381160736084,
                    -0.0015319086378440264,
                    3.141218662261963
                ],
                [
                    0.5376223865405412,
                    0.19093033055550618,
                    0.44330028363322316,
                    -3.1380574703216553,
                    -0.0019246124429628253,
                    3.1399550437927246
                ],
                [
                    0.521504613338733,
                    -0.0005120121635832758,
                    0.4429621801014548,
                    3.140838623046875,
                    -0.0007978816283866763,
                    3.1415627002716064
                ],
        ],
    "current_joint_angles":r.robot_status("jointAngles")
    }
r.move_linear(**linear_property)
r.stop()
```

4. Move circular

```
from neurapy.robot import Robot

r = Robot()
r.set_mode("Automatic")
```

(continues on next page)

(continued from previous page)

```
time.sleep(1)

# Move to initial position
joint_property = {
    "speed": 50.0,
    "acceleration": 50.0,
    "safety_toggle": True,
    "target_joint": [
        [0.3223498, 0.170158, 1.3894271, -0.0039065, 1.5862455, -0.0013291]
    ],
    "current_joint_angles": r.robot_status("jointAngles")
}
r.move_joint(**joint_property)
time.sleep(1)
circular_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "target_pose": [
        [
            0.3744609827431085,
            -0.3391784988266481,
            0.23276604279256016,
            3.14119553565979,
            -0.00017731254047248513,
            -0.48800110816955566
        ],
        [
            0.37116786741831503,
            -0.19686307684994242,
            0.23300456855796453,
            3.141423225402832,
            -0.00020668463548645377,
            -0.48725831508636475
        ],
        [
            0.5190337951593321,
            -0.1969996948428492,
            0.23267853691809767,
            3.1414194107055664,
            -0.00017726201622281224,
            -0.48750609159469604
        ]
    ],
    "current_joint_angles": r.robot_status("jointAngles")
}
r.move_circular(**circular_property)
r.stop()
```

5. Move composite

```
from neurapy.robot import Robot
```

(continues on next page)

(continued from previous page)

```
r = Robot()
r.set_mode("Automatic")
time.sleep(1)

# Move to initial  position
joint_property = {
    "speed": 50.0,
    "acceleration": 50.0,
    "safety_toggle": True,
    "target_joint": [
        [0.9855635563580816, 0.3476807153487805, 1.1687323223466928, -0.
↪0007014516639916519, 1.6256822043262882, -0.0006834753266336617]
    ],
    "current_joint_angles": r.robot_status("jointAngles")
}
r.move_joint(**joint_property)
time.sleep(1)

# Move Composite
composite_motion_property = {
        "speed": 0.25,
        "acceleration": 0.1,
        "current_joint_angles": r.robot_status('jointAngles'),
        "commands": [ {
                "linear":       {
                    "blend_radius": 0.005,

                    "targets":  [
                            [
                                0.30323030824121955,
                                0.4573874594177961,
                                0.44309265860894687,
                                3.140892505645752,
                                -0.0005030535394325852,
                                -2.155384063720703
                            ],
                            [
                                0.5452084494942819,
                                0.058104230123491495,
                                0.44311138849670073,
                                3.1414873600006104,
                                -0.0014394369209185243,
                                -3.0334575176239014
                            ]

                                ]
                            }
                    },
                    {
                "circular": {
                    "targets":  [
                        [
```

(continues on next page)

(continued from previous page)

```
                            0.5452084494942819,
                            0.05810423012349l495,
                            0.44311138849670073,
                            3.1414873600006104,
                            -0.001439436920918524З,
                            -3.0334575176239014
                        ],
                        [
                            0.4708356936660009,
                            0.280180550729363,
                            0.4431937440044086,
                            3.1407277584075928,
                            -0.0022785856854170 56,
                            -2.601893424987793
                        ],
                        [
                            0.30323030824121955,
                            0.4573874594177961,
                            0.44309265860894687,
                            3.140892505645752,
                            -0.0005030535394325852,
                            -2.155384063720703
                        ]

                            ]
                        }
                }
            ]
        }

r.move_composite(**composite_motion_property)
r.stop()
```

6. Move trajectory

```
from neurapy.robot import Robot

r = Robot()
trajectory_motion_property = {
            "current_joint_angles": r.robot_status('jointAngles'),
            "timestamps": [
                0.01,0.02,0.03
            ],
            "target_joint": [
                [0.21,0.2,1.34,0,1.61,0],
                [0.22,0.21,1.35,0,1.61,0],
                [0.23, 0.22, 1.36,0,1.61,0]
            ]
        }
r.move_trajectory(**trajectory_motion_property)
r.stop()
```

7. Move recorded path

```python
from neurapy.robot import Robot

r = Robot()
recorded_path_property = {
    "is_motion": False,
    "file_location": "<path to file location>",
    "speed": 0.25,
    "use_recordings_filter": False,
    "current_joint_angles": r.robot_status('jointAngles'),
}
r.record_path(**recorded_path_property)
```

CHAPTER

SEVEN

EXAMPLES FOR LARA 10

1. **Robot Object Initialization**

```
from neurapy.robot import Robot

r = Robot()
print(r.robot_name)
print(r.dof)
print(r.platform)
print(r.payload)
print(r.kURL)
print(r.robot_urdf_path)
print(r.current_tool)
print(r.connection)
print(r.version)
```

2. Move joint

```
from neurapy.robot import Robot

r = Robot()
joint_property = {
    "speed": 50.0,
    "acceleration": 50.0,
    "target_joint": [
        previous_joint_angles,
        [
            0.7740276502469322,
            0.13599234975308788,
            1.2223307505101257,
            -0.31901765024691225,
            0.3242476502469122,
            -0.31901765024691225
        ],
        [
            0.30333999999999633,
            -0.5334353004938217,
            1.4419907505101492,
            -0.31901765024691225,
            0.3242476502469122,
            -0.31901765024691225
        ],
```

(continues on next page)

(continued from previous page)

```
        [
            0.059259210493824405,
            0.08133676975309524,
            1.4419907505101492,
            -1.0084715804938613,
            -0.7008323497530976,
            -0.31901765024691225
        ]
    ]
}
r.move_joint(**joint_property)
```

3. Move linear

```
from neurapy.robot import Robot

r = Robot()
r.set_mode("Automatic")
time.sleep(1)

# moving the robot to initial position
joint_property = {
        "speed": 50.0,
        "acceleration": 50.0,
        "safety_toggle": True,
        "target_joint": [
            [
                -0.0007546,
                0.2713685,
                1.2664022,
                -0.0007550,
                1.6046191,
                -0.0007501,
            ]
        ],
    "current_joint_angles":r.robot_status("jointAngles")
    }
r.move_joint(**joint_property)
time.sleep(1)

linear_property = {
        "speed": 0.9,
        "acceleration": 0.2,
        "target_pose": [
            [
                    0.521504613338733,
                    -0.0005120121635832758,
                    0.4429621801014548,
                    3.140838623046875,
                    -0.0007978816283866763,
                    3.1415627002716064
                ],
```

(continues on next page)

(continued from previous page)

```
                [
                    0.5216067833501538,
                    -0.21429644443731097,
                    0.44311804388931564,
                    3.140573024749756,
                    -0.0008633044781163335,
                    3.1414897441864014
                ],
                [
                    0.30336607796720333,
                    -0.2139397968890111,
                    0.44282379715713066,
                    3.140761613845825,
                    -0.0011759602930396795,
                    3.1410810947418213
                ],
                [
                    0.3033097863631349,
                    0.19367482201823044,
                    0.44268805519921206,
                    -3.1399381160736084,
                    -0.00153190863784402 6,
                    3.141218662261963
                ],
                [
                    0.5376223865405412,
                    0.19093033055550618,
                    0.44330028363322316,
                    -3.1380574703216553,
                    -0.0019246124429628253,
                    3.1399550437927246
                ],
                [
                    0.521504613338733,
                    -0.0005120121635832758,
                    0.4429621801014548,
                    3.140838623046875,
                    -0.0007978816283866763,
                    3.1415627002716064
                ],
        ],
    "current_joint_angles":r.robot_status("jointAngles")
    }
r.move_linear(**linear_property)
r.stop()
```

4. Move circular

```
from neurapy.robot import Robot

r = Robot()
r.set_mode("Automatic")
```

(continues on next page)

(continued from previous page)

```
time.sleep(1)

# Move to initial position
joint_property = {
    "speed": 50.0,
    "acceleration": 50.0,
    "safety_toggle": True,
    "target_joint": [
        [0.3223498, 0.170158, 1.3894271, -0.0039065, 1.5862455, -0.0013291]
    ],
    "current_joint_angles": r.robot_status("jointAngles")
}
r.move_joint(**joint_property)
time.sleep(1)
circular_property = {
    "speed": 0.25,
    "acceleration": 0.1,
    "target_pose": [
        [
            0.3744609827431085,
            -0.3391784988266481,
            0.23276604279256016,
            3.14119553565979,
            -0.00017731254047248513,
            -0.488001108169555566
        ],
        [
            0.37116786741831503,
            -0.19686307684994242,
            0.23300456855796453,
            3.141423225402832,
            -0.00020668463548645377,
            -0.48725831508636475
        ],
        [
            0.5190337951593321,
            -0.1969996948428492,
            0.23267853691809767,
            3.1414194107055664,
            -0.00017726201622281224,
            -0.48750609159469604
        ]
    ],
    "current_joint_angles": r.robot_status("jointAngles")
}
r.move_circular(**circular_property)
r.stop()
```

5. Move composite

```
from neurapy.robot import Robot
```

(continues on next page)

(continued from previous page)

```
r = Robot()
r.set_mode("Automatic")
time.sleep(1)

# Move to initial  position
joint_property = {
    "speed": 50.0,
    "acceleration": 50.0,
    "safety_toggle": True,
    "target_joint": [
        [0.9855635563580816, 0.3476807153487805, 1.1687323223466928, -0.
↪0007014516639916519, 1.6256822043262882, -0.0006834753266336171]
    ],
    "current_joint_angles": r.robot_status("jointAngles")
}
r.move_joint(**joint_property)
time.sleep(1)

# Move Composite
composite_motion_property = {
        "speed": 0.25,
        "acceleration": 0.1,
        "current_joint_angles": r.robot_status('jointAngles'),
        "commands": [ {
                "linear":       {
                    "blend_radius": 0.005,

                    "targets":  [
                            [
                                0.30323030824121955,
                                0.4573874594177961,
                                0.44309265860894687,
                                3.140892505645752,
                                -0.0005030535394325852,
                                -2.155384063720703
                            ],
                            [
                                0.5452084494942819,
                                0.05810423012349149 5,
                                0.44311138849670073,
                                3.1414873600006104,
                                -0.0014394369209185243,
                                -3.0334575176239014
                            ]

                                ]
                            }
                    },
                    {
                "circular": {
                    "targets":  [
                        [
```

(continues on next page)

(continued from previous page)

```
                        0.5452084494942819,
                        0.058104230123491495,
                        0.44311138849670073,
                        3.1414873600006104,
                        -0.0014394369209185243,
                        -3.0334575176239014
                    ],
                    [
                        0.4708356936660009,
                        0.280180550729363,
                        0.4431937440044086,
                        3.1407277584075928,
                        -0.002278585685417056,
                        -2.601893424987793
                    ],
                    [
                        0.30323030824121955,
                        0.4573874594177961,
                        0.44309265860894687,
                        3.140892505645752,
                        -0.0005030535394325852,
                        -2.155384063720703
                    ]

                        ]
                    }
            }
        ]
    }

r.move_composite(**composite_motion_property)
r.stop()
```

6. Move trajectory

```
from neurapy.robot import Robot

r = Robot()
trajectory_motion_property = {
            "current_joint_angles": r.robot_status('jointAngles'),
            "timestamps": [
                0.01,0.02,0.03
            ],
            "target_joint": [
                [0.21,0.2,1.34,0,1.61,0],
                [0.22,0.21,1.35,0,1.61,0],
                [0.23, 0.22, 1.36,0,1.61,0]
            ]
        }
r.move_trajectory(**trajectory_motion_property)
r.stop()
```

7. Move recorded path

```
from neurapy.robot import Robot

r = Robot()
recorded_path_property = {
    "is_motion": False,
    "file_location": "<path to file location>",
    "speed": 0.25,
    "use_recordings_filter": False,
    "current_joint_angles": r.robot_status('jointAngles'),
}
r.record_path(**recorded_path_property)
```

CHAPTER

EIGHT

EXAMPLES FOR MAIRA 7

1. **Move linear**

```
from neurapy.robot import Robot
import time

# Initializing the robot and setting it to Automatic mode
r = Robot()
r.set_mode("Automatic")
time.sleep(1)

# Moving the robot to Initial position
joint_property = {
        "speed": 50.0,
        "acceleration": 50.0,
        "safety_toggle": True,
        "target_joint": [
            [
                1.406,
                -0.8005,
                0.07339,
                -0.7950,
                -0.00769,
                -1.51563,
                -0.65509
            ]
        ],
    "current_joint_angles":r.robot_status("jointAngles")
    }
r.move_joint(**joint_property)

#Linear motion
linear_property = {
        "speed": 0.9,
        "acceleration": 0.2,
        "target_pose": [
            [
                    -0.14602202042158488,
                    -1.2039712892654788,
                    0.5542529402524848,
                    -3.1250412464141846,
```

(continues on next page)

(continued from previous page)

```
                    -0.05224483087658882,
                    -1.030046820640564
                ],
                [
                    0.32102659790371435,
                    -1.2039027313771722,
                    0.5543345835272986,
                    -3.125161409378051 8,
                    -0.05228979140520096,
                    -1.030221700668335
                ],
                [
                    0.3214140596556481,
                    -0.8018749844104675,
                    0.55438618778188,
                    -3.1251306533813477,
                    -0.052478402853012085,
                    -1.0297577381134033
                ],
                [
                    -0.2883707227945903,
                    -0.8016352570186388,
                    0.5542651616343833,
                    -3.125182628631592,
                    -0.052266936749219894,
                    -1.0304018259048462
                ],
                [
                    -0.14602202042158488,
                    -1.2039712892654788,
                    0.5542529402524848,
                    -3.1250412464141846,
                    -0.05224483087658882,
                    -1.030046820640564
                ]
        ],
    "current_joint_angles":r.robot_status("jointAngles")
    }
r.move_linear(**linear_property)
r.stop()
```

2. Move circular

```
from neurapy.robot import Robot
import time

# Initializing the robot and setting it to Automatic mode
r = Robot()
r.set_mode("Automatic")
time.sleep(1)

#move robot to initial position
```

(continues on next page)

(continued from previous page)

```
joint_property = {
        "speed": 50.0,
        "acceleration": 50.0,
        "safety_toggle": True,
        "target_joint": [
            [
            2.687,
            -0.4671,
            -0.1182,
            -1.2874,
            0.11341,
            -1.3622,
            -1.23634
            ]
        ],
    "current_joint_angles":r.robot_status("jointAngles")
    }
r.move_joint(**joint_property)
time.sleep(1)

# Move Circular
circular_property = {
        "speed": 1.0,
        "acceleration": 0.1,
        "target_pose": [
                [
                    0.8827123230328554,
                    -0.4975248049417001,
                    0.5826623081385038,
                    3.135085105895996,
                    -0.06406070291996002,
                    0.655619740486145
                ],
                [
                    -0.03171114438156373,
                    -1.0127658173803937,
                    0.58266951895531,
                    3.1350698471069336,
                    -0.064027339220047,
                    -0.43322843313217163
                ],
                [
                    -0.7324668416327569,
                    -0.7001335665738433,
                    0.5826737224363503,
                    3.1350419521331787,
                    -0.06396733969449997,
                    -1.209897518157959
                ]
        ],
    "current_joint_angles":r.robot_status("jointAngles")
    }
```

(continues on next page)

(continued from previous page)

```
r.move_circular(**circular_property)
r.stop()
```

3. **Move composite**

```
from neurapy.robot import Robot
import time

# Initializing the robot and setting it to Automatic mode
r = Robot()
r.set_mode("Automatic")
time.sleep(1)
#Move robot to initial position
joint_property = {
        "speed": 50.0,
        "acceleration": 50.0,
        "safety_toggle": True,
        "target_joint": [
            [
                2.687546699107577,
                -0.46715931807787425,
                -0.11823711193030452,
                -1.2874158466547219,
                0.11341945351835118,
                -1.3622644468222112,
                -1.2363431996854033
            ]
        ],
    "current_joint_angles":r.robot_status("jointAngles")
    }
r.move_joint(**joint_property)
time.sleep(1)

#Move Composite
composite_motion_property = {
        "speed": 0.432,
        "acceleration": 0.1,
        "current_joint_angles": r.robot_status('jointAngles'),
        "commands": [ {
                "linear": {
                    "blend_radius": 0.005,

                    "targets": [
                            [
                                0.8827123230328554,
                                -0.4975248049417001,
                                0.5826623081385038,
                                3.135085105895996,
                                -0.06406070291996002,
                                0.655619740486145
                            ],
                            [
```

(continues on next page)

(continued from previous page)

```
                                -0.7324668416327569,
                                -0.7001335665738433,
                                0.5826737224363503,
                                3.1350419521331787,
                                -0.06396733969449997,
                                -1.209897518157959
                            ]
                        ]
                    }
                },
                {
                    "circular": {
                        "targets": [
                            [
                                -0.7324668416327569,
                                -0.7001335665738433,
                                0.5826737224363503,
                                3.1350419521331787,
                                -0.06396733969449997,
                                -1.209897518157959
                            ],
                            [
                                -0.03171114438156373,
                                -1.0127658173803937,
                                0.58266951895531,
                                3.1350698471069336,
                                -0.064027339220047,
                                -0.43322843313217163
                            ],
                            [
                                0.8827123230328554,
                                -0.4975248049417001,
                                0.5826623081385038,
                                3.135085105895996,
                                -0.06406070291996002,
                                0.655619740486145
                            ]
                        ]
                    }
                },

            ]
        }
r.move_composite(**composite_motion_property)
r.stop()
```

CHAPTER

NINE

TCP-FLANGE CONNECTION PIN OUTS FOR LARA

NEURA ROBOTICS

6.3.4.1. TCP Flange connection pin-out

External flange connections pin-out (12 pole M12 connector / IEC 61076-2-101)

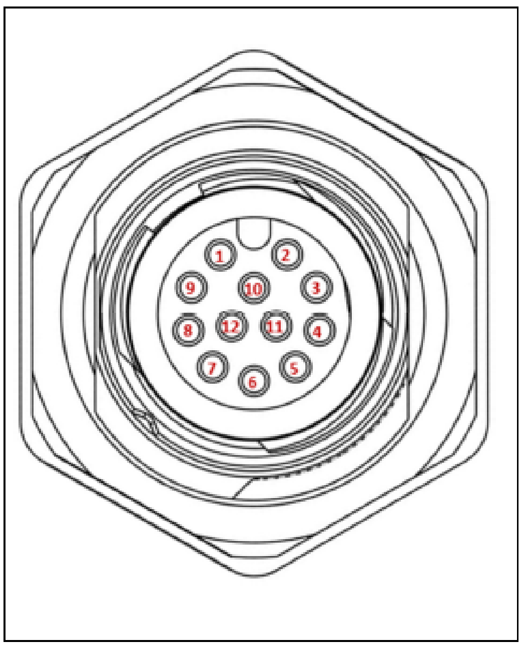


Fig. 25: IO ports, pin assignment of M12 connector (IEC 61076-2-101)

Pin-out of M12 connector for external connection is defined as follows:

Pin	Line	Cable color	Definition
1	Input_0	Brown	Digital input 0
2	Input_1	Blue	Digital input 1
3	Input_2	White	Digital input 2
4	Output_0	Green	Digital output 0
5	Output _1	Powder	Digital output 1
6	Output _2	Yellow	Digital output 2
7	RS485_A (TX+)	Black	RS484+ communication (Modbus RTU)
8	RS485_B (TX-)	Gray	RS484- communication (Modbus RTU)
9	AI 0	Red	Analog input 0
10	AI 1	Purple	Analog input 1
11	24 V	Ash powder (Grey/ Pink)	Power supply (24V)
12	GND	Red / Blue	Common Ground (0V)

CHAPTER TEN

TCP-FLANGE CONNECTION PIN OUTS FOR MAIRA

External flange connections

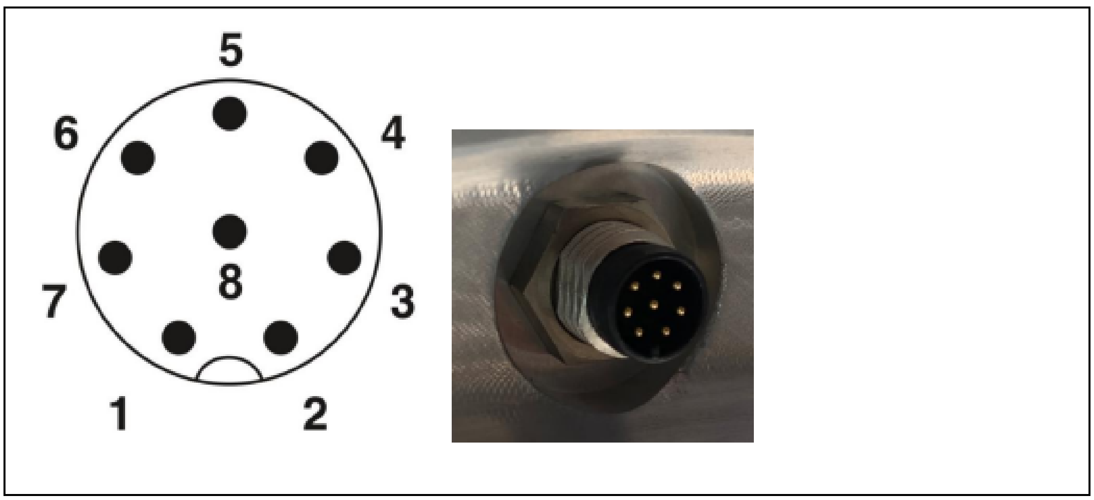


Fig. 29: M8 connectors, pin assignment (SACC-DSI-M8MS-8CON-M8/0,5)

IO external flange connections are defined as follows:

Pin	Line color	Definition	Notes
Port1			
1	White	TX+	
2	Brown	RX-	
3	Green	TX-	
4	Yellow	RX+	
5	Gray	+24V	
6	Pink		Not used
7	Blue		Not used
8	Red	GND	
Port 2			
1	White	DI_0	
2	Brown	DI_1	
3	Green	DO_0	
4	Yellow	DO_1	
5	Gray	+24V	
6	Pink	AI_0	
7	Blue	AI_1	
8	Red	GND	
Port 3			
1	White	RS485_A	
2	Brown	RS485_B	
3	Green		Not used
4	Yellow		Not used
5	Gray	+24V	
6	Pink		Not used
7	Blue		Not used
8	Red	GND	

PYTHON MODULE INDEX

n

INDEX

I

J

L

M

W

Z