Python module integration guide BPS 1.9 / LS 1.2

Introduction

The Python module serves as a tool to send **Requests** and receive **Responses** from the **Vision controller** to your PC. You can use it to get more familiar with the concept of how Robot modules work, or to create communication with unsupported Robotic brands. A big advantage is that using this module doesn't require a robot.
For now this module is compatible only with **ABB_IRB/1.9.0**, only ABB robots can be simulated.

1. Prerequisites

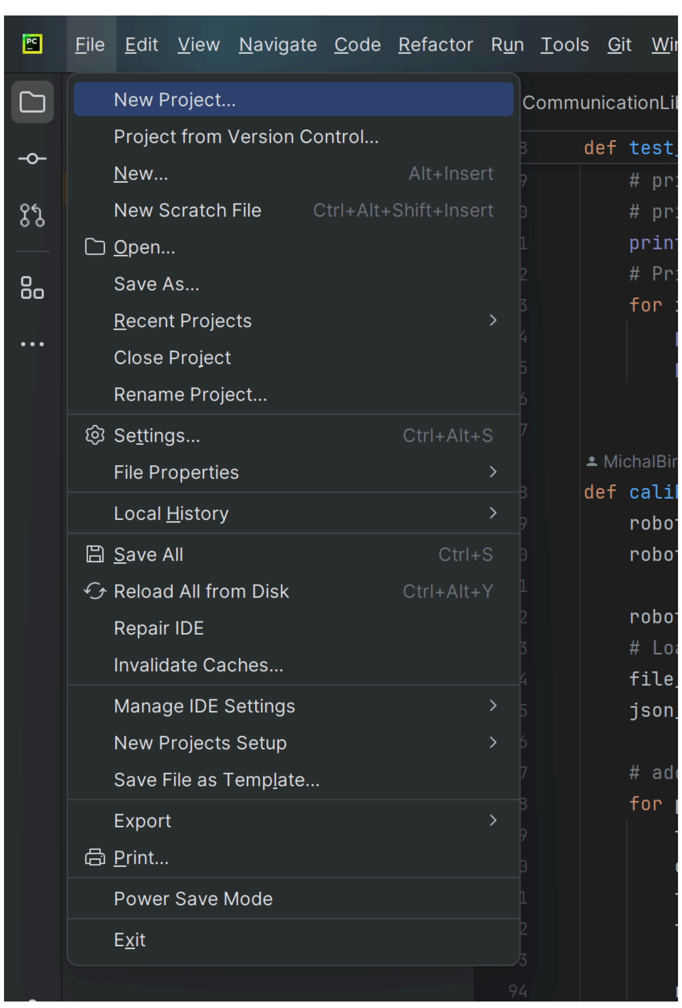
- Python 3.12 (tested on)
- Programming environment - Pycharm / VS code
- Vision controller (Nuvo 5xxx, 9xxx - tested on)
- Locator Studio 1.2.0
- Binpincking Studio 1.9.0

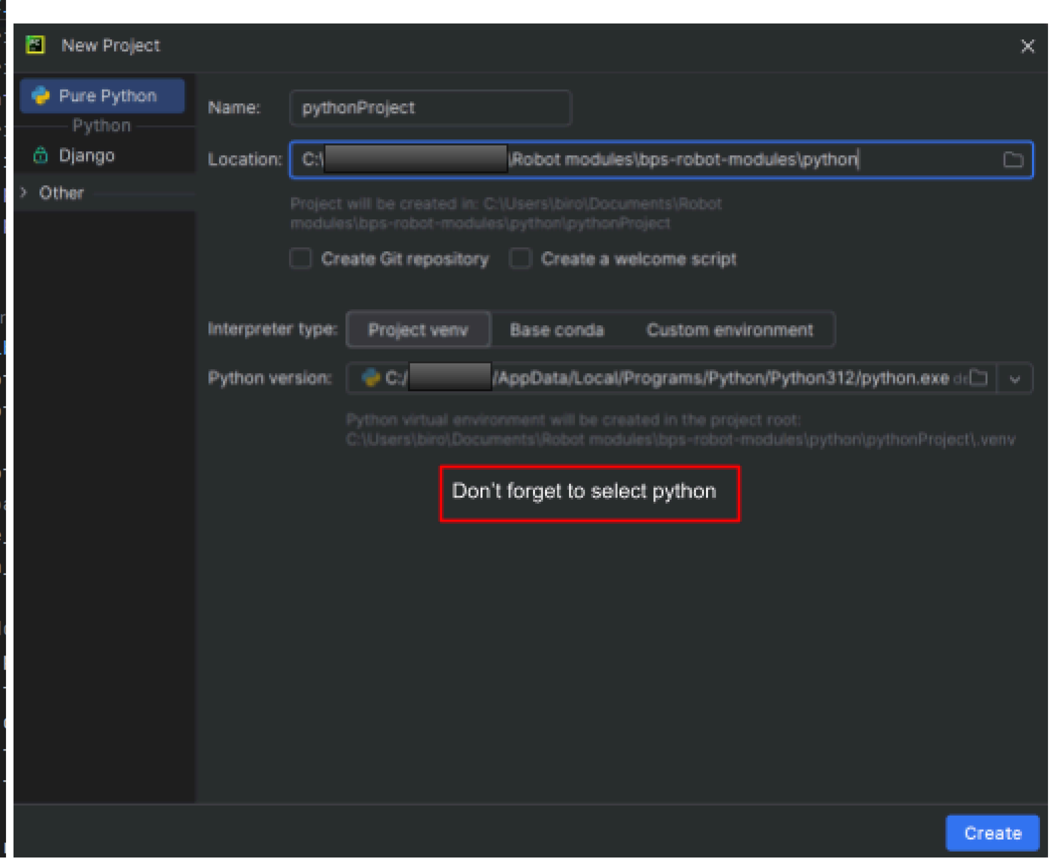
2. Python environment setup

To use the python module we will need an environment in which we can run the program and Vision controller with which we will communicate.

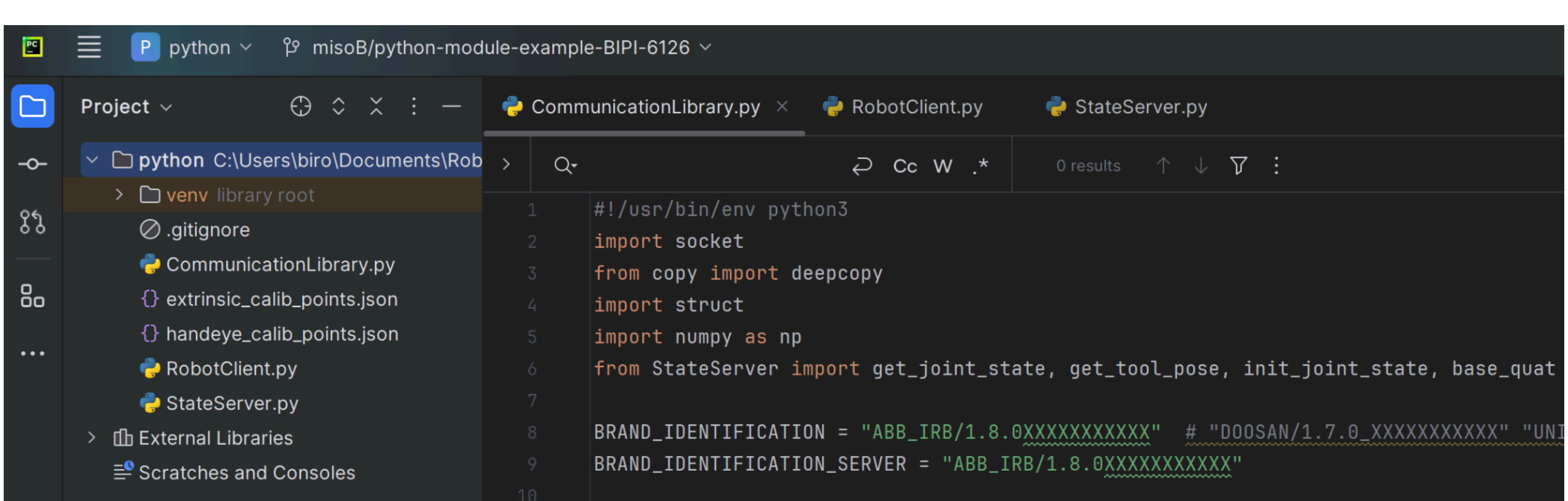
Open programming environment in which you can run python programs. In this documentation we will use PyCharm IDE. If you don't have one please install it - you can download the free version at this link (*look for* ***PyCharm Community Edition*** *- it is an open source software which can be used for commercial use*).

Once you open the environment, download the module and import it to the interface:
File -> New project -> insert location of the folder with module / python location -> CREATE



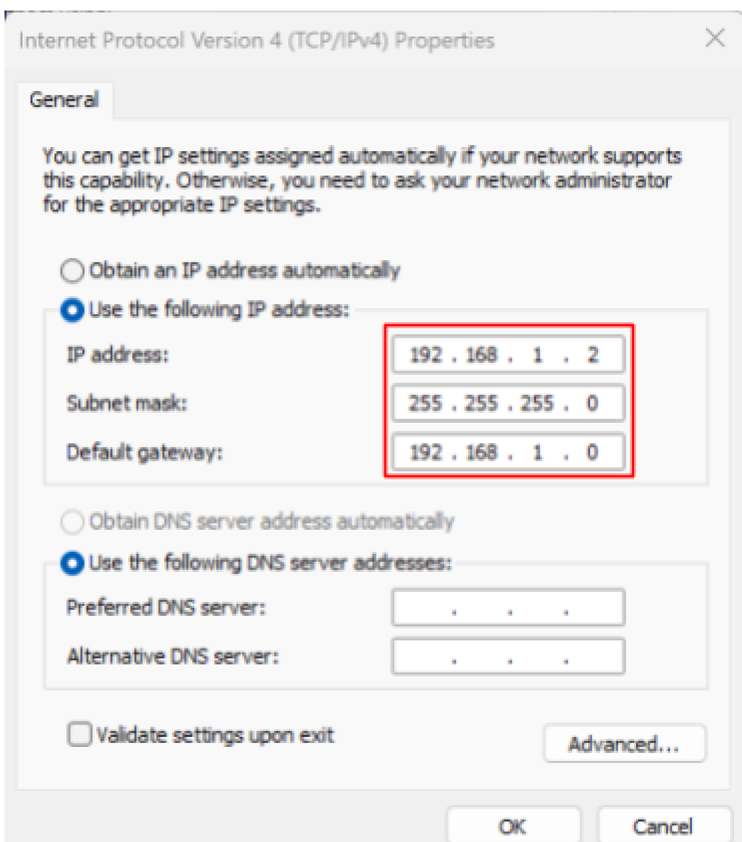


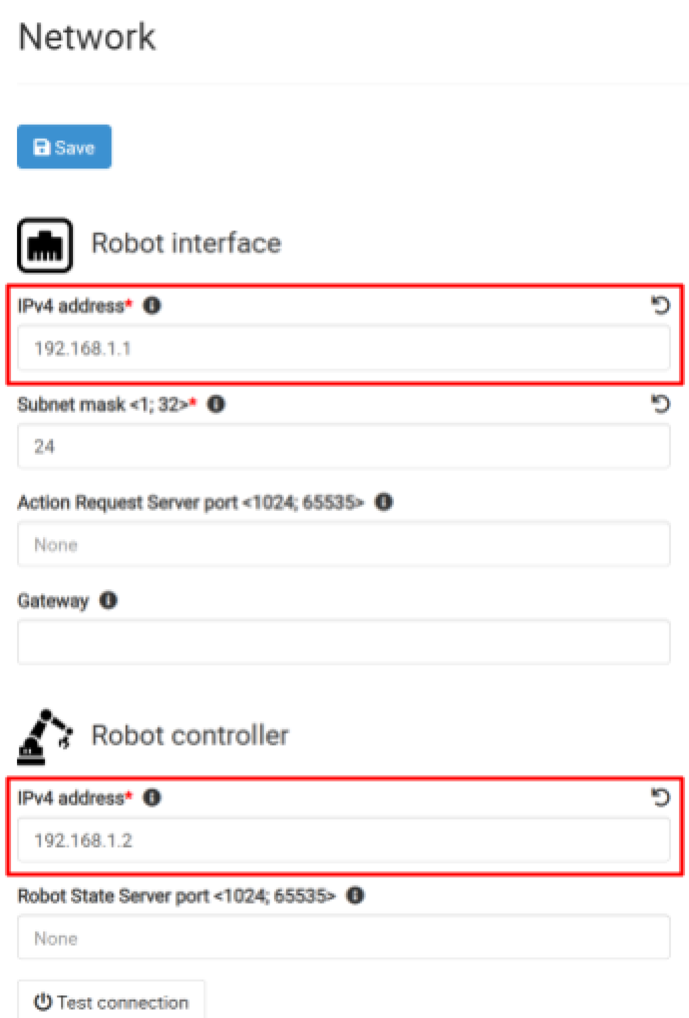
Now you can see python module in the interface:



2.1 Setup IPv4 communication

"Robot controller" ip address in Vision controller Locator/Binpicking Studio (*Network page*) must be the same ip address as your computer ipv4 ethernet.
To set IP address of your ethernet open - ***Control panel -> View network status and tasks (under Network and Internet) -> Ethernet -> Properties***





Don't forget to set the IP address in RobotClient.py to the IP address of "Robot interface" (Vision controller) set in the Locator/Binpicking Studio.

CommunicationLibrary.py | RobotClient.py | StateServer.py

```python
#!/usr/bin/env python3

import CommunicationLibrary
import time
import json

CONTROLLER_IP = "192.168.1.2"
PORT = 11003


def test_ls():
    robot = CommunicationLibrary.RobotRequestResponseCommunication()  #
    robot.connect_to_server(CONTROLLER_IP, PORT)  # communication betwe
```

In case of using *StateServer.py,* set variable ROBOT_CONTROLLER_IP to the address set in the Locator/ Binpicking Studio *Network page.*

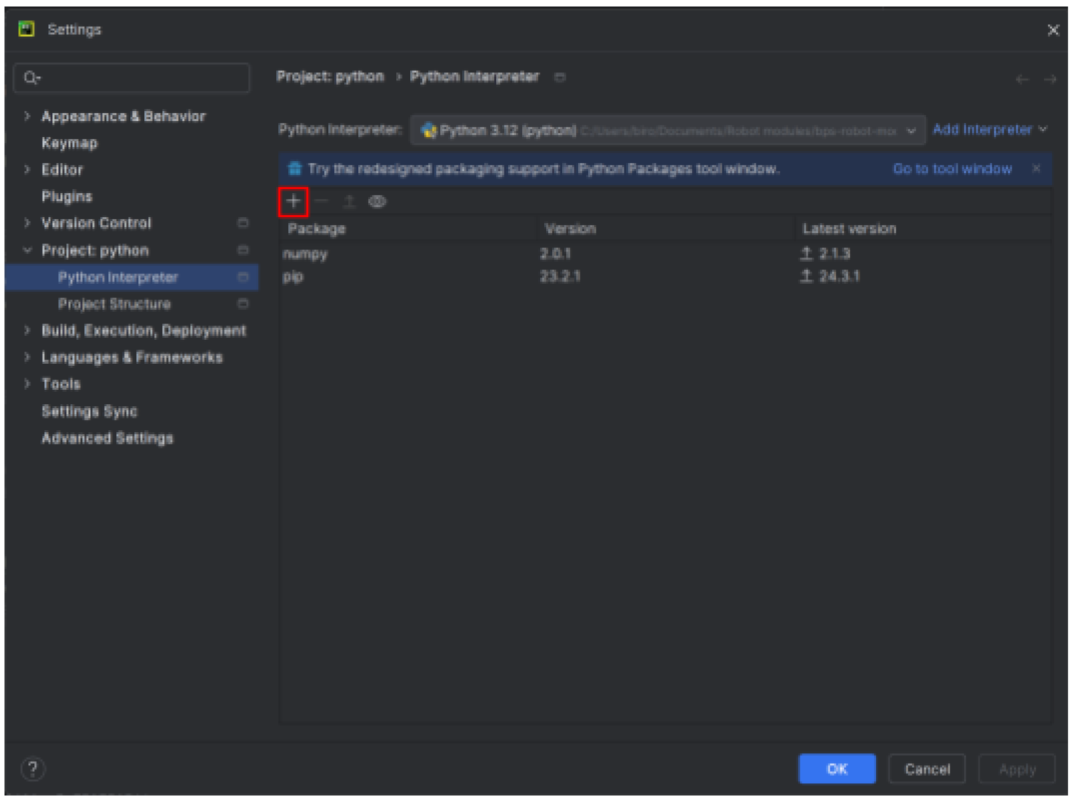
CommunicationLibrary.py | RobotClient.py | StateServer.py

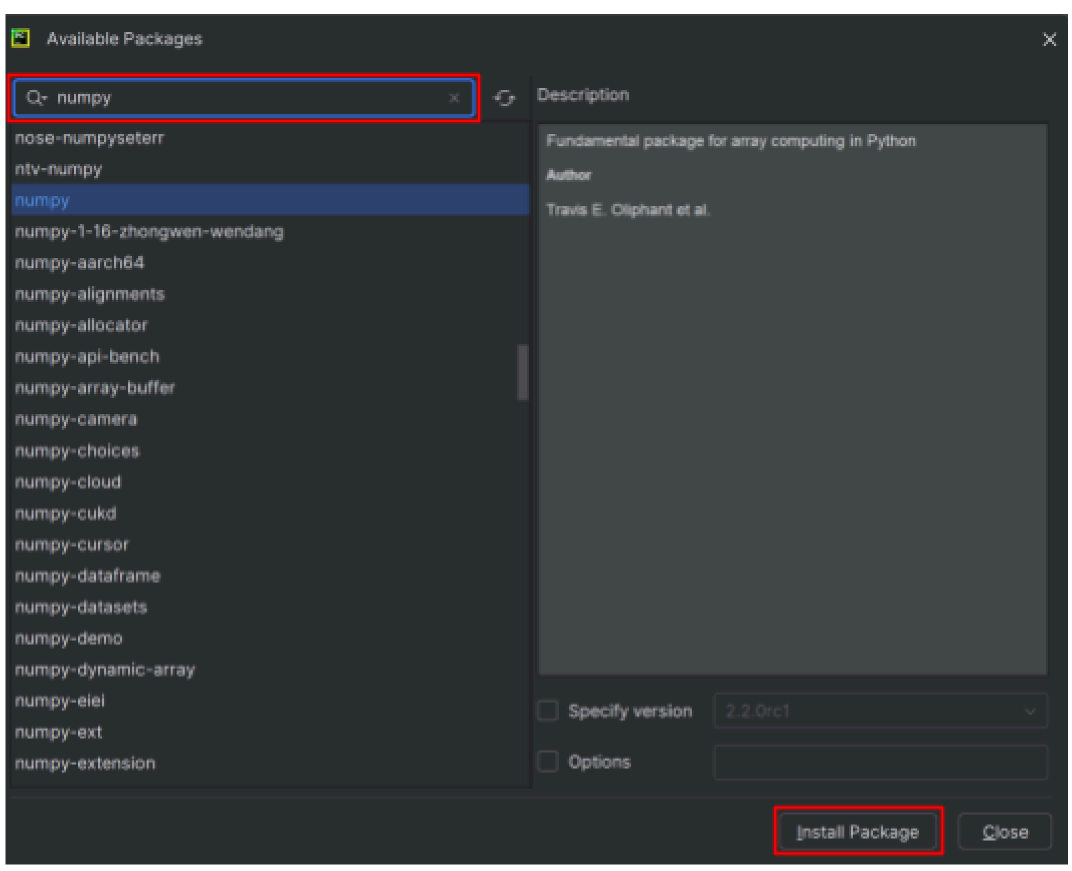
```python
#!/usr/bin/env python3
import socket
import time
import CommunicationLibrary
import random
import math
import numpy as np

SOCKET_RECV_TIMEOUT = 5
ROBOT_CONTROLLER_IP = "192.168.1.2"
PORT = 11004
```

2.2 Install libraries

Before running the program check if all libraries you need are installed. In the case any of the libraries are missing, install them.
File -> Settings (Ctrl + Alt + S)





List of libraries necessary for running the program depends on the files you want to run - Examples, calibration, State server, etc..

- socket
- sys
- copy
- struct
- numpy
- json

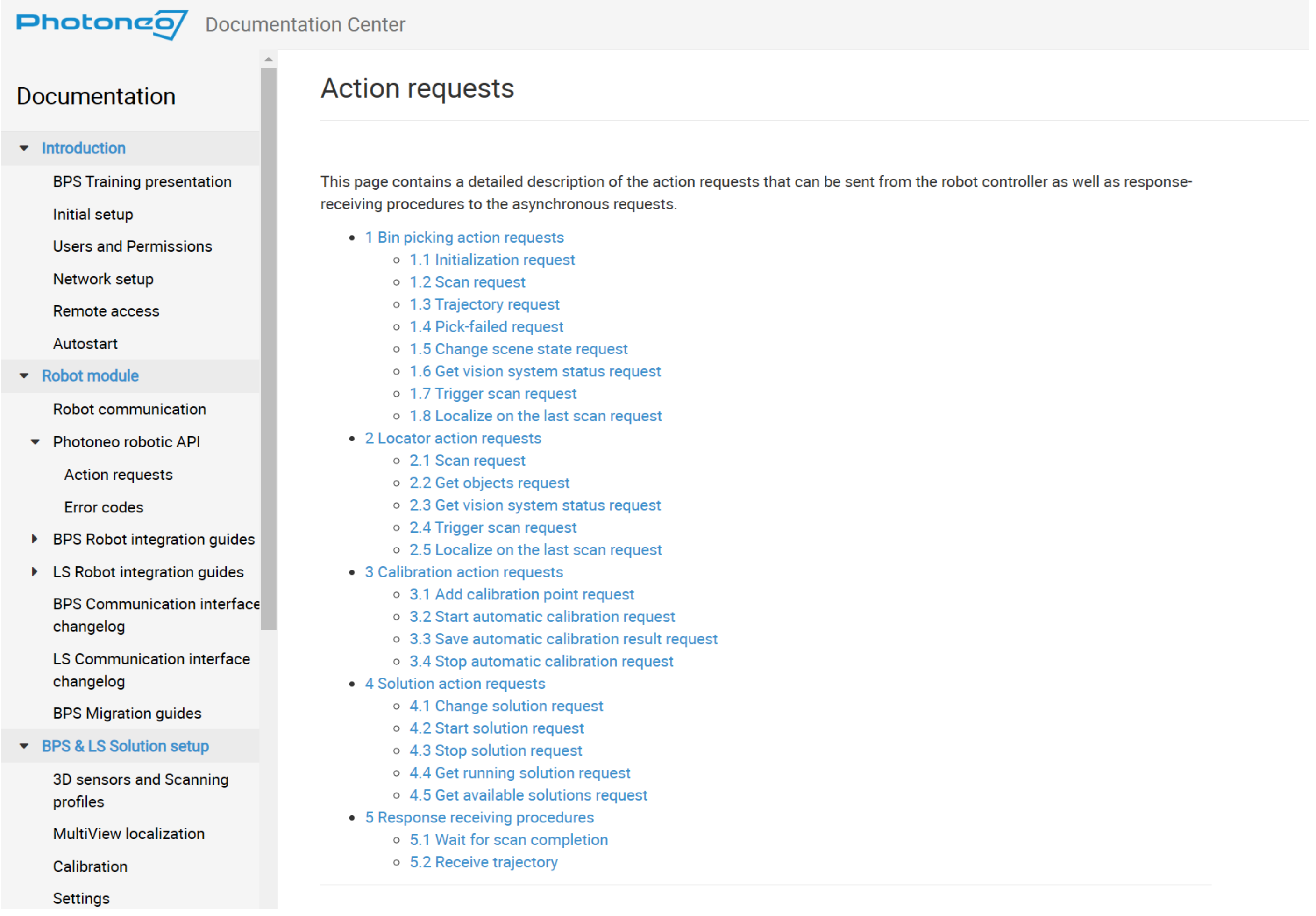
3. Python module

The module contains 2 main python files - *CommunicationLibrary.py, StateServer.py* and example programs.

3.1 Robotic API

This section describes available API calls provided by the Robot module. These procedures are intended for high-level control of the bin picking application.

Note: Please read Action requests for detailed documentation of these procedures(***user login:*** *customer* ***password:*** *Ready2learnhow2pick*).



These procedures are contained in the *CommunicationLibrary.py* and must not be edited! This file contains every **Action request function**, ***pho_send_request*** function and ***pho_receive_response*** function, which is universal for every request.
Action request functions are divided into 4 groups - *Binpicking, Locator, Calibration, Solution* requests.

Example of action request function:

```python
def pho_request_binpicking_init(self, vs_id, start, end):
    payload = struct.pack("i", vs_id)  # payload - vision system ID
    payload = payload + floatArray2bytes(start) # payload - start
    payload = payload + floatArray2bytes(end)  # payload - end
    self.pho_send_request(ActionRequest.PHO_BINPICKING_INIT, payload)
    self.pho_receive_response(ActionRequest.PHO_BINPICKING_INIT)
```

- *There are 3 input parameters -* ***vision system ID***, ***start position*** *and* ***end position***.
- *In variable* ***payload*** *we put together message which contains bytes that we send to the Vision controller*
- *function pho_send_request() send message to Controller*
- *function pho_receive_response() wait for response from Controller, receive it and parse it based on the message type.*

3.1.1 Connection procedures

Warning: These procedures are contained in the **pho_common** API section and must not be edited!

Connection procedure	Description / Usage

Connect to Action Request Server `connect_to_server(self, CONTROLLER_IP, PORT)`	**Description** Function to establish a new connection to the Action Request Server. **Note:** The Port and IP of the Action Request Server (vision controller) is configured in **CommuniationLibrary.py** **Usage** The procedure should be called only once at the beginning of the program. Only after the connection has been established is it possible to send requests.
`close_connection(self)`	**Usage** The procedure should be called only once at the end of the program to clove connection with Server.

3.1.2 Communication procedures

Bin picking requests

Request	Input variables	Output
Initialization request `pho_request_binpicking_init (self, vs_id, start, end)`	`vs_id` - vision system ID `start` - start joint pose `end` - end joint pose	`error` - response code
Scan request `pho_request_binpicking_scan (self, vs_id, tool_pose=None)`	`vs_id` - vision system ID `Tool_pose` - optional parameter - tool pose which consist of 7 number - translation(3) and quaternion(4)	`error` - response code
Trajectory request `pho_request_binpicking_trajectory (self, vs_id)`	`vs_id` - vision system ID	`error` - response code `trajectory_data` - trajectory joint positions in array `gripper_commands` - gripper operations `gripping_info` - tool invariances, gripping point ID, gripping point invariances `dimensions(AI only)` - X, Y, Z dimensions of found object
Pick-failed request `pho_request_binpicking_pick_failed (self, vs_id)`	`vs_id` - vision system ID	`error` - response code
Get vision system status request `pho_request_binpicking_get_vision_system_status(self, vs_id)`	`vs_id` - vision system ID	`error` - response code `Status_data` - number of localized objects, number of ready objects, pipeline status
Change scene state request `pho_request_binpicking_change_scene_status(self, scene_status_id)`	`scene_status_id` - scene state ID	`error` - response code

Trigger scan request `pho_request_binpicking_trigger_scan(self, vs_id, tool_pose=None)`	`vs_id` - vision system ID `tool_pose` - Robot pose(cartezian) optional argument, used in Handeye Multiview	`error` - response code
Localize on the last scan request `pho_request_binpicking_reuse_scan(self, vs_id, tool_pose=None)`	`vs_id` - vision system ID `tool_pose` - Robot pose(cartezian) optional argument, used in Handeye Multiview	`error` - response code

Locator request

Request	Input variables	Output
Scan request `pho_request_locator_scan(self, vs_id, tool_pose=None)`	`vs_id` - vision system ID `Tool_pose` - Robot pose(cartezian) optional argument, used in Handeye Singleview/Multiview)	`error` - response code
Get objects request `pho_request_locator_get_objects(self, vs_id, number_of_objects)`	`vs_id` - vision system ID `number_of_objects` - number of requested objects	`error` - response code `object_pose` - list of cartesian robot pose (x,y,z + quaternion) `dimensions (AI only)` - X, Y, Z dimensions of found object
Get vision system status request `pho_request_locator_get_vision_system_status(self, vs_id)`	`vs_id` - vision system ID	`error` - response code `Status_data` - number of localized objects, number of ready objects, pipeline status
Trigger scan request `pho_request_locator_trigger_scan(self, vs_id, tool_pose=None)`	`vs_id` - vision system ID `tool_pose` - Robot pose(cartezian) optional argument, used in Handeye Multiview	`error` - response code
Localize on the last scan request `pho_request_locator_reuse_scan(self, vs_id, tool_pose=None)`	`vs_id` - vision system ID `tool_pose` - Robot pose(cartezian) optional argument, used in Handeye Singleview/Multiview	`error` - response code

Calibration requests

Request	Input variables	Output
Add calibration point request `pho_request_calibration_add_point(self, tool_pose=None)`	`tool_pose` - Robot pose(cartezian) must be used with Locator	`error` - response code
Start automatic calibration request `pho_request_calibration_start(self, sol_id, vs_id)`	`sol_id` - solution system ID `vs_id` - vision system ID	`error` - response code
Save automatic calibration request `pho_request_calibration_save(self)`	-	`error` - response code `calib_data` - accuracy `camera_pose` - cartesian pose of camera

Stop automatic calibration request `pho_request_calibration_stop(self)`	-	`error` - response code

Solution requests

Request	Input variables	Output
Change solution request `pho_request_solution_change(self, sol_id)`	`sol_id` - solution ID	`error` - response code
Start solution request `pho_request_solution_start(self, sol_id)`	`sol_id` - solution ID	`error` - response code
Stop solution request `pho_request_solution_stop(self)`	---	`error` - response code
Get running solution request `pho_request_solution_get_running(self)`	---	`error` - response code `running_solution` - ID of running solution
Get available solutions request `pho_request_solution_get_available(self)`		`error` - response code `available_solution` - list of IDs of available solution

Response receiving procedures

Response receiving procedures	Input variables
Wait for scan completion - Binpicking `pho_binpicking_wait_for_scan(self)`	---
Wait for scan completion - Locator `pho_locator_wait_for_scan(self)`	---

3.1.3 Output variables

Output values from requests are stored in class ***ResponseData*** which consists of output variables mentioned in tables above. To read variables it is important to create object in your program which will contain all data from responses.
In the code example below, is created object - **data.** Through this object we are able to reach data from responses in the variables like *error* or *data_pose*.

```python
data = CommunicationLibrary.ResponseData()

print(data.error)
print(data.object_pose)
```

3.2 StateServer.py

Simulates State server of a robot. It generates the joint and tool pose of an ABB robot. Joint pose is generated in a specific range and joints are incremented and decremented by random values in the infinite loop. Tool pose moves on a circle with center in the z axis of the robot.

You need to run it separately from the RobotClient.py.

If you want to check if the State server is working go to ***Environment -> Robot -> Motion mode: Robot module*** and you should be able to see the robot moving.

Attention!

This serves only to test and show functionality of the module. It generates random values, therefore it cannot be used for calibration or other requests that use the TCP of the robot.

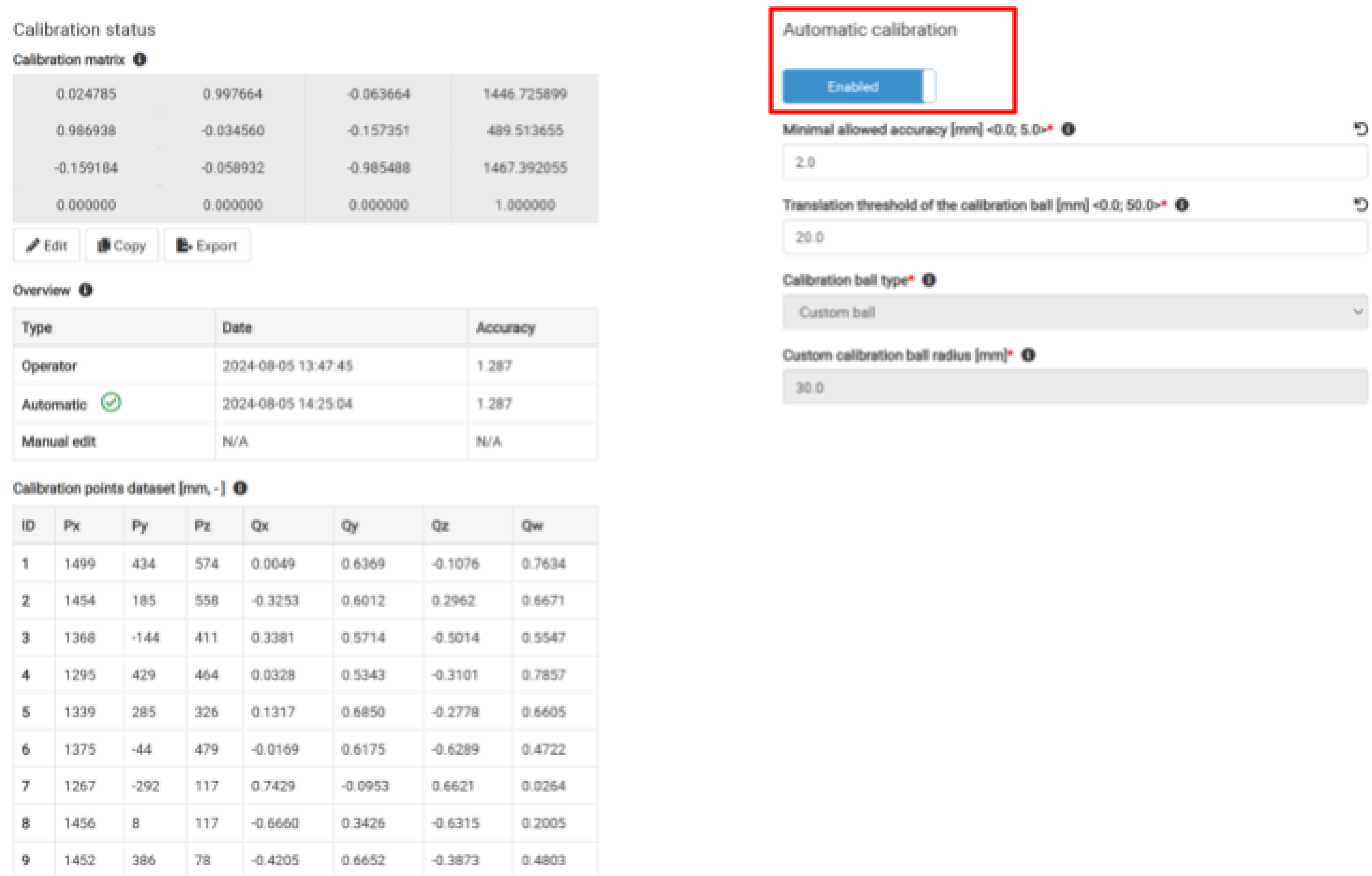
4. Examples

4.1 Automatic Calibration

The python module's goal is to simulate the robot. If we want to simulate the calibration process, we need to have scans from the process of calibration and pair them with the TCP (*translation, quaternion*) that belongs to the position of the robot in the particular scan. In the module there is an example for understanding the process.

It's important to mention that we are talking about **automatic calibration**, you need a calibrated system to enable the automatic calibration in the Binpicking studio Calibration settings.

In attachments you can find solutions with already calibrated Vision systems to test and run the automatic calibration from the example program.



4.1.1 Extrinsic

First you need to open a File camera - list of scans on which we will simulate the calibration. You can find the scans in the attachments folder - .json. After that you will load a TCP pose from the *extrinsic_calib_points.json* file. They are in the order with the scans. After that you can run the robot program for calibration.

Note: File camera and .json files are included only to show and help customers to better understand how the program works. For real applications it is necessary to send TCP from a robot and scans from a real camera.

```python
import CommunicationLibrary
import json

CONTROLLER_IP = "192.168.1.1"
PORT = 11003

tool_pose = []

robot = CommunicationLibrary.RobotRequestResponseCommunication()  # object is created
robot.connect_to_server(CONTROLLER_IP, PORT)  # communication between VC and robot is created

robot.pho_request_calibration_start(6,1)
# Load the JSON data
file_path = 'extrinsic_calib_points.json'
#  load JSON data from a file
with open(file_path, 'r') as file:
    json_data = json.load(file)

# add 9 calibration point
for point in json_data:
    translation_m = point["translation"]
    quaternion = point["quaternion"]
    translation_mm = [x * 1000 for x in translation_m] # mm to m
    tool_pose = translation_mm + quaternion

    robot.pho_request_calibration_add_point(tool_pose)


robot. pho_request_calibration_save()
robot.pho_request_calibration_stop()
```

	type	name
Program	.python file	extrinsic_calibration.py

Scans	folder - .praw files	Extrinsic_calibration
Tool poses (TCP)	.json file	extrinsic_calib_points.json

4.1.2 Handeye

Automatic calibration for Handeye works the same as for Extrinsic. Change the file camera and load the right .json file with calibration poses according to the table.

	type	name
Program	.python file	handeye_calibration.py
Scans	folder - .praw files	Handeye_calibration
Tool poses (TCP)	.json file	handeye_calib_points.json

4.2 Locator

4.2.1 Basic_example.py

This example shows a basic application, where we request scan and then locate objects. You need to import solution ***example-solution-layer-localization-253.pbcf*** to BPS from folder - ***solutions.***

```python
import CommunicationLibrary

CONTROLLER_IP = "192.168.1.1"
PORT = 11003

robot = CommunicationLibrary.RobotRequestResponseCommunication()  # object is created
robot.connect_to_server(CONTROLLER_IP, PORT)  # communication between VC and robot is created

robot.pho_request_solution_start(253)

# request scan
robot.pho_request_locator_scan(1)
robot.pho_locator_wait_for_scan()
# request position of 5 located objects
robot.pho_request_locator_get_objects(1, 5)

robot.close_connection()  # communication needs to be closed
```

4.2.2 Change_solution

This example shows how to change the solution on the Vision controller. You need to import solutions ***example-solution-ai-localization-252.pbcf*** and ***example-solution-layer-localization-253.pbcf*** to BPS from folder - ***solutions.***

```python
import CommunicationLibrary

CONTROLLER_IP = "192.168.1.1"
PORT = 11003

robot = CommunicationLibrary.RobotRequestResponseCommunication()  # object is created
robot.connect_to_server(CONTROLLER_IP, PORT)  # communication between VC and robot is created

robot.pho_request_solution_start(252)
robot.pho_request_locator_scan(1)
robot.pho_locator_wait_for_scan()
robot.pho_request_locator_get_objects(1, 5)

robot.pho_request_solution_change(253)
robot.pho_request_locator_scan(1)
robot.pho_locator_wait_for_scan()
robot.pho_request_locator_get_objects(1, 5)

robot.close_connection()  # communication needs to be closed
```

4.2.3 Multiview_handeye

This example shows how to use Multiview and localize objects on the scene captured by multiple scans. You need to import solution ***python-multiview_loca-1.pbcf*** to BPS from folder - ***solutions.***

```python
import CommunicationLibrary
import json

CONTROLLER_IP = "192.168.1.1"
PORT = 11003

tool_pose = []

robot = CommunicationLibrary.RobotRequestResponseCommunication()  # object is created
robot.connect_to_server(CONTROLLER_IP, PORT)  # communication between VC and robot is created

robot.pho_request_solution_start(1)

file_path = 'multiview_pose.json'
#  load JSON data from a file
with open(file_path, 'r') as file:
    json_data = json.load(file)

for point in json_data:
    translation_mm = point["translation"]
    quaternion = point["quaternion"]
    tool_pose = translation_mm + quaternion # put TCP together

    robot.pho_request_locator_trigger_scan(1,tool_pose) # trigger scan to capture scan, repeat this request
    robot.pho_locator_wait_for_scan()

robot.pho_request_locator_scan(1,tool_pose) # start meshing of scans and localization on the scene
robot.pho_locator_wait_for_scan()
```

4.3 Binpicking

4.3.1 Basic_example

This example shows the basic application of Binpicking studio. You need to import solution ***example-solution-cad-picking-254.pbcf*** to BPS from folder - ***solutions.***

```python
import CommunicationLibrary

CONTROLLER_IP = "192.168.1.1"
PORT = 11003

start_pose = [3.14, 0.6, 1.13, 3.14, 0.6, 3.14]
end_pose = [3.14, 0.6, 1.13, 3.14, 0.6, 3.14]


robot = CommunicationLibrary.RobotRequestResponseCommunication()  # object is created
robot.connect_to_server(CONTROLLER_IP, PORT)  # communication between VC and robot is created

robot.pho_request_solution_start(254)
robot.pho_request_binpicking_init(1, start_pose, end_pose)

# request scan
robot.pho_request_binpicking_scan(1)
robot.pho_binpicking_wait_for_scan()

# request trajectory
robot.pho_request_binpicking_trajectory(1)

robot.close_connection()  # communication needs to be closed
```