OMRON

PyARCL User’s Guide

V 1.0

Table of Contents

Revision History

Revision	Date	Author	Changelog
1.0	30/05/2023	Marco Zangrandi	Document Published

What is PyARCL

PyARCL is a Python package developed by OMRON Italy with the aim of providing a collection of functions and classes to communicate with OMRON Autonomous Mobile Robots.

PyARCL is essentially a Python-written wrapper of the ARCL programming language, and the User should therefore have deep knowledge of the existing Robot APIs to understand the inner workings of this package.

The main aim of **PyARCL** was to provide simple-to-use, fast-to-implement Python code to be used in a safe environment. Its syntax encourages scripting and throwaway coding, but this doesn't mean that the User might not find of interest building a performant, real-time monitoring system or other traceability-related applications.

PyARCL is delivered with absolutely no warranty of any kind.

PyARCL was not built to be run in a production environment and therefore the User should not expect production-level performance from the presented functions and classes.

Aim of this document

This document aims at giving a complete explanation and overview of the main classes and functions present in the **PyARCL** package and understanding their typical usage in the numerous '**Example**' sections.

At the end the User may find a small **Cookbook** of typical applications of this package.

Related Manuals

Related Products	Main Topic	Manual Name
LD60/90	Product Guide	LD-60/90 User's Manual
LD250	Product Guide	LD-250 User's Manual
HD1500	Product Guide	HD1500 User's Manual
Fleet Manager	Product Guide	Enterprise Manager 2100 User's Guide
All Robots	Robot API	Advanced Robotics Command Language Reference Guide
Fleet Manager	Fleet Manager API	Advanced Robotics Command Language Fleet Manager Integration Guide
Fleet Manager	Simulator Software	Fleet Simulator User's Manual

1 Package Overview

The **PyARCL** package is composed of three different modules, different by dependencies and scope.

1.1 AMRBase Module

The **PyARCL.AMRBase** module mainly exists to provide the **Robot** and **FleetManager** classes, as well as the main functions used by said classes to communicate with the OMRON AMRs.

The module does not have dependencies outside of the core Python libraries (that come with an ordinary Python installation) and should therefore seamlessly integrate with other Python packages.

AMRBase		
Provides	Classes	**Robot**, **FleetManager**
	Functions	None
Dependencies	Core	**telnetlib, datetime, threading, logging, re, inspect, functools, os**
	Extra	None

1.2 AMRSimTools Module

The **PyARCL.AMRSimTools** module provides functions designed to help the User to run AMR simulations, mostly expected to be used in conjunction with the *OMRON Enterprise Manager* running in *Simulator Mode*.

The module does not have dependencies outside of the core Python libraries (that come with an ordinary Python installation) and should therefore seamlessly integrate with other Python packages.

AMRSimTools		
Provides	Classes	None
	Functions	**program, wait, waitForJob, waitForJobState, waitForInput, randomNumber**
Dependencies	Core	**threading, time, random**
	Extra	None

1.3 AMRFrontEnds Module

The **PyARCL.AMRFrontEnds** module provides a single class, **FleetStatsGUI**, which displays a simplistic dashboard representing the performances of a fleet (simulated or not) when on object instantiation.

The module does have dependencies outside of the core Python libraries, that will need to be installed for the components to work. Be especially aware of the tkinter package dependency, which is a core dependency from Python 3.7 onwards only and will need to be installed manually on an older Python installation.

AMRSimTools		
Provides	Classes	**FleetStatsGUI**
	Functions	None
Dependencies	Core	**tkinter, threading, time, datetime**
	Extra	**matplotlib**

The AMRFrontEnds module and the FleetStatsGUI class were tested functioning with matplotlib v. 5.7.1.
However, a more recent or older version of the matplotlib package would almost certainly work perfectly fine as well.

2 PyARCL Installation

At the moment, **PyARCL** is not distributed through PyPI, even though the eventual distribution through the official Python Package Index channel will be made in due time.
When **PyARCL** will be released through PyPI, the User may install the package simply by running the command:

```
pip install PyARCL
```

Another way to use **PyARCL** without installing it in the User's virtual environment or current Python installation is to simply copy the 'PyARCL' folder containing the **PyARCL** modules in the current work directory.
If the User's Python files will be placed in the same directory where the 'PyARCL' folder is located, they will be able to import the **PyARCL** package without fail.

3 Robot and FleetManager Properties/Methods

The **PyARCL.AMRBase** module mainly provides the two main classes of the PyARCL package that handle communication with the OMRON AMRs and Enterprise Manager appliances. Since the ARCL command language has a lot of common commands between the Robot and Fleet Manager communication (e.g. **queueDropoff**, **getDateTime**, **getGoals**, …), there are a lot of common components between the **Robot** and **FleetManager** classes.

For simplicity, the **Robot** and **FleetManager** components will be presented under this structure:

1. Section 4.1 'Robot Class Unique Components' will contain the methods and properties that are exclusively related to the **Robot** class.
2. Section 5.1 'FleetManager Class Unique Components' will contain the methods and properties exclusively related to the **FleetManager** class.
3. Extra section 6 'Robot and FleetManager Classes Common Components' will contain all properties and methods that are callable by both a **Robot** and a **FleetManager** object.

4 The Robot Class

The **Robot** class is used to represent a single OMRON AMR. The User does not need to specify the robot model (i.e. LD-60, LD-250, HD-1500, etc…) to instance an object, since all OMRON AMRs use the same structure for the ARCL API.

Since the **Robot** class will be available as a top-level import from PyARCL, import the **Robot** class by simply declaring:

```
from PyARCL import Robot
```

To instantiate a **Robot** object the User may use the class constructor, which requires in order:

Attribute	Type	Default Value	Description
IP	*string*		The IP address of the AMR
password	*string*	adept	The ARCL server password, as specified in the robot configuration.
port	*int*	7171	The ARCL server TCP socket number, as specified in the robot configuration.

The only needed input parameter is the AMR IP address. This means that if the ARCL port is set as the default (7171) and the password is "adept", then the Robot object can be instantiated simply by declaring:

```
from PyARCL import Robot
AMR = Robot('192.168.1.120')
```

Assumed 192.168.1.120 to be the AMR IP address.
If the AMR has a custom ARCL server configuration, using a different TCP socket number or a different password, then the password and/or port needs to be specified:

```
from PyARCL import Robot
AMR = Robot('192.168.1.120', password='omron', port='8000')
```

Assumed the ARCL password and port to be relatively "omron" and 8000.
Note that since input parameters are ordered, the same object can be instantiated also by declaring:

```
from PyARCL import Robot
AMR = Robot('192.168.1.120', 'omron', '8000')
```

May the User wish to do so.

4.1 Robot Class Unique Components

In this section there are described the methods and properties that describe a **Robot** class and are exclusive to a **Robot** object. Also check section 6 'Robot and FleetManager Classes Common Components' to get the full picture of the methods and properties callable by a **Robot** object.
Instantiate a **Robot** object as described in section 4 to start using them.

The Robot properties and method employ a queue-like system to achieve multithreading safety. The User may create one instance of a Robot object and access its properties and methods concurrently without fear of race conditions happening while parsing the ARCL responses.

4.1.1 Robot.status Property

The **status** property allows to retrieve information about the robot status as described by ARAM. Accessing the property returns a **Status** object containing the following attributes:

Attribute	Type	Description
aramStatus	*string*	The full ARAM status as printed by the 'status' ARCL command.
stateOfCharge	*float*	The state of charge of the robot. Value is 0-1 for 0%-100%.
location	*list*	List of three floats, representing x, y, θ in order.
loc_score	*float*	Localization score of the robot
temperature	*float*	Internal (CPU) temperature of the robot.

Python allows to concatenate seamlessly subfield access, meaning that the User may access (for example) the localization score of the AMR by simply declaring:

```
localization_score = AMR.status.loc_score
```

Instead of splitting the object retrieval and the attribute inspection:

```
AMR_status = AMR.status
localization_score = AMR_status.loc_score
```

The status property returns a custom Status object containing information updated to the moment the status property was called. Every time the status property is requested by the User, PyARCL automatically sends the 'status' ARCL command and parses the response to build an updated Status object.

Examples

In this section there are presented two simple examples of the typical usage of the status property:

1. The first example just shows how the Status object represents itself to the User when printed.
2. The second example shows a loop that continuously checks the robot location and prints whether it's headed left or right.

Example 1. Status representation

```
>>> AMR = Robot("10.67.18.121")
>>> print(AMR.status)

Status:
    Aram Status:            Going to Mag12_P
    State of Charge:        84.0

    Location:               X: 163157 [mm], Y: 24327 [mm], Θ: 177 [deg]
    Localization Score:     1.0

    CPU Temperature:        37.9

>>> print(AMR.status.temperature)

37.9
```

Example 2. Status check loop

```
>>> from PyARCL.AMRSimTools import wait
>>> while True:
>>>     robot_theta = AMR.status.location[2] # Retrieve theta
>>>     print(f'Robot theta value is {robot_theta}')
>>>     if -90 < robot_theta < 90:
>>>         print('Robot going right')
>>>     else:
>>>         print('Robot going left')
>>>     wait(1) # Waiting one second between each check

Robot theta value is 15
Robot going right
Robot theta value is 53
Robot going right
Robot theta value is 91
Robot going left
Robot theta value is 129
Robot going left
```

4.1.2 Robot.inputList Property

The **inputList** property returns a list containing the names of all the inputs in the AMR that have been configured to be 'custom'.

Example

```
>>> AMR.inputList

['i1', 'i10', 'i2', 'i3', 'i4', 'i5', 'i6', 'i7', 'i8', 'i9']
```

4.1.3 Robot.inputs Property

The **inputs** property returns a container functionally identical to a dictionary that contains the values of the AMR's custom inputs. The input value is represented by a Boolean, 'True' for 'on' and 'False' for 'off'.

The inputs property returns a custom InputDict container, dictionary-like object containing information updated to the moment the inputs property was called. Every time the inputs property is requested by the User, PyARCL automatically sends the 'inputList' and 'inputQuery' ARCL commands accordingly and parses the answer to provide an updated response.

To access an input, provide its name to the **inputs** property as the lookup key:

```
>>> AMR.inputs['i1']

True
```

Examples

In this section there are presented four simple examples of the typical usage of the **inputs** property:

1. The first example just shows how the **InputDict** container represents itself to the User when printed.
2. The second example shows how inputs values can be used as Booleans.
3. The third example shows a loop that continuously checks the robot input and displays a message when it turns on.
4. The fourth example shows iteration between inputs to print which are the ones that are on.

Example 1. Inputs representation

```
>>> robot_inputs = AMR.inputs
>>> print(robot_inputs)

Inputs:
    i1: on
    i10: off
    i2: off
    i3: on
    i4: off
    i5: on
    i6: off
    i7: off
    i8: on
    i9: on
```

Example 2. Inputs as Booleans

```
>>> def is_it_on(input_name):
>>>     if AMR.inputs[input_name]:
>>>         print(f'{input_name} is on')
>>>     else:
>>>         print(f'{input_name} is off')
>>>
>>> is_it_on('i1')
>>> is_it_on('i2')

i1 is on
i2 is off
```

Example 3. Checking the inputs

```
>>> import time
>>> while True:
>>>     if AMR.inputs['i2']:
>>>         print('input has been turned on!')
>>>         break
>>>     else:
>>>         print('still nothing...')
>>>     time.sleep(1)

still nothing...
still nothing...
still nothing...
input has been turned on!
```

Example 4. Looping the inputs

```
>>> open_inputs = []
>>> for input_name in AMR.inputs:
>>>     if AMR.inputs[input_name]:
>>>         open_inputs.append(input_name)
>>> total_message = 'The inputs that are on are: '
>>> for input_name in open_inputs:
>>>     total_message += f'{input_name}, '
>>> total_message = total_message[:-2] # Removing last ', '
>>> print(total_message)

The inputs that are on are: i1, i2, i3, i5, i8, i9
```

4.1.4 Robot.outputList Property

The **outputList** property returns a list containing the names of all the outputs in the AMR that have been configured to be 'custom' (e.g. 'custom', 'customAlsoInput', …)

Example

```
>>> AMR.outputList

['o1', 'o10', 'o2', 'o3', 'o4', 'o5', 'o6', 'o7', 'o8', 'o9']
```

4.1.5 Robot.outputs Property

The **outputs** property returns a container functionally identical to a dictionary that contains the values of the AMR's custom outputs. The output value is represented by a Boolean, 'True' for 'on' and 'False' for 'off'.

The inputs property returns a custom OutputDict container, dictionary-like object containing information updated to the moment the inputs property was called. Every time the inputs property is requested by the User, PyARCL automatically sends the 'outputList' and 'outputQuery' ARCL commands accordingly and parses the answer to provide an updated response.

The **outputs** property allows the User to both retrieve the outputs value and to set their state at will. In typical Pythonic fashion this is done by just setting the appropriate output to the desired 'True' or 'False' counterpart for 'on' and 'off' respectively.

To access an output, provide its name to the **outputs** property as the lookup key:

```
>>> AMR.outputs['o1']
False
```

To set its value, access and set it to the desired state:

```
>>> AMR.outputs['o1']
False
>>> AMR.outputs['o1'] = True
>>> AMR.outputs['o1']
True
```

Examples

In this section there are presented three simple examples of the typical usage of the **outputs** property:

1. The first example just shows how the **OutputDict** container represents itself to the User when printed.
2. The second example shows how the **outputs** property allows interaction between the User program and the AMRs: The example script sends a command to the AMR based on the output value, effectively making the output act as a task / macro trigger.
3. The third example shows a more complex case of User program – AMR interaction. The program sets up a TCP server where the AMR connects and sends a single string containing an integer between 0 and 15 (through the *arbitraryServerSend* task) and the program modifies the Robot IO to match the binary representation of the integer sent.

Example 1. Outputs representation

```
>>> AMR.outputs
Outputs:
    o1: off
    o10: on
    o2: off
    o3: on
    o4: off
    o5: on
    o6: on
    o7: on
    o8: off
    o9: off
```

Example 2. Running instant tasks based on output value

```python
import time
from PyARCL.AMRBase import sendCommand
old_output_value = None
while True:
    current_output_value = AMR.outputs['o1'] # store the output value
    if current_output_value != old_output_value: # If output value has changed
        if current_output_value: # Change max speed to 200 mm/s
            sendCommand(AMR, 'doTaskInstant movementParameters Normal 200')
        else: # Change max speed back to default
            sendCommand(AMR, 'doTaskInstant movementParameters Normal 0')

    old_output_value = current_output_value # update the previous loop's value
    time.sleep(1) # Wait one second before restarting the loop
```

Example 3.

```python
def binarize_outputs(number): # Maximum allowed number: 15, as it uses 4 bits
# Take decimal number and get a binary representation as string
bin_number = bin(int(number)).replace('0b', '')
    bin_str = bin_number.rjust(4, '0')

    # Open sequence of outputs accordingly: 'o1', 'o2', 'o3', 'o4'
    for i, value in enumerate(bin_str):
        if value == '1':
            AMR.outputs[f'o{i+1}'] = True # i+1 because i starts at 0
        elif value == '0':
            AMR.outputs[f'o{i+1}'] = False

import socket
PORT = 65431  # Port to listen on (non-privileged ports are > 1023)

while True:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind(('', PORT))
        s.listen()
        conn, addr = s.accept()
        with conn:
            while True:
                data = conn.recv(1024)
                try:
                    dec_number = data[:data.index(b'\x00')] # Sanitize mes-sage
                    binarize_outputs(dec_number) # Open the outputs accord-ingly
                except:
                    pass

                if not data:
                    break
```

4.1.6 Robot.queueDropoff Method

The **queueDropoff** method is the only job-dispatching method that can be exclusively used by a **Robot** object, whether the relative AMR may be connected to a Fleet Manager or not.
This method is also the only way to force the assignment of the job to a single robot connected to a Fleet Manager without employing Custom Tasks.

The **queueDropoff** method generates a single-segment job containing only a dropoff segment. The method uses the following arguments:

Arguments	Type	Default Value	Description
missionGoal	*string*		The goal name to send to the dropoff mission to.
priority	*int*	**If in Fleet**: 20 **If Standalone**: config. def.	The job segment priority.
customJobId	*string*	None	If specified, the string will be set as the custom job ID.

The queueDropoff method returns a Job object, which can be used to check the real-time state of the job and its job segments, as well as modifying its segments goals and priorities at will. Check section 7 'The Job Class' for more information.

Example

```
>>> import time
>>> import datetime
>>> now = datetime.datetime.now()
>>> print(f'Job launched at {now}')
>>> job = AMR.queueDropoff('Goal1', priority=50, customJobId='PyARCL_Test')
>>> while True:
>>>     if job.iscomplete:
>>>         break
>>>     time.sleep(1)
>>> now = datetime.datetime.now()
>>> print(f'Job finished at {now}')

Job launched at 2023-04-18 17:30:40.632921
Job finished at 2023-04-18 17:31:14.214158
```

```
>>> job

Job: PyARCL_Test
    Completition: True
    Job Segments:
        [1]: DROPOFF1

>>> job['DROPOFF1']

JobSegment: DROPOFF1
    JobID:          PyARCL_Test
    JobType:        Dropoff

    Priority:       50
    State:          Completed
    Substate:       None
    Goal:           Goal1
    AssignedRobot:  AMR1
    QueuedTime:     2023-04-18 17:30:40
    FinishedTime:   2023-04-18 17:31:14
    FailNumber:     0
```

4.1.7 Robot.goto Method

The goto method is used to send the robot to a goal, as if it was sent there by a simple 'goto' ARCL command.

Arguments	Type	Default Value	Description
goal	*string*		The goal name to send to the robot to.

4.1.8 Robot.dock Method

The dock method is used to send the robot to dock, as if it was sent there by a simple 'dock' ARCL command.
The method has no attributes, simply call it by declaring:

```
AMR.dock()
```

4.1.9 Robot.undock Method

The undock method is used to command the robot to undock, as if it was sent a simple 'undock' ARCL command.
The method has no attributes, simply call it by declaring:

```
AMR.undock()
```

4.1.10 Robot.doTask Method

The **doTask** method is used to command the robot to perform a task, as if it was sent a simple 'dotask <task> <attributes>' command.

Arguments	Type	Default Value	Description
task	*string*		The task name to command the robot with.
attributes	*string*		The task attributes to send with the task.

4.1.11 Robot.executeMacro Method

The **executeMacro** is used to command the robot to run a macro, as if it was sent a simple 'executemacro <macro>' command.

Arguments	Type	Default Value	Description
macro	*string*		The macro name the robot needs to run.

4.1.12 Robot.stop Method

The **stop** method is used to stop, as if it was sent there by a simple 'stop' ARCL command.
The method has no attributes, simply call it by declaring:

```
AMR.stop()
```

5 The FleetManager Class

The **FleetManager** class is used to represent a Fleet Manager, most commonly being an OMRON Enterprise Manager appliance.

Since the **FleetManager** class will be available as a top-level import from PyARCL, import the **FleetManager** class by simply declaring:

```
from PyARCL import FleetManager
```

To instantiate a **FleetManager** object the User may use the class constructor, which requires in order:

Attribute	Type	Default Value	Description
IP	*string*		The IP address of the AMR
password	*string*	"adept"	The ARCL server password, as specified in the robot configuration.
port	*int*	7171	The ARCL server TCP socket number, as specified in the robot configuration.

The only needed input parameter is the EM IP address. This means that if the ARCL port is set as the default (7171) and the password is "adept", then the **FleetManager** object can be instantiated simply by declaring:

```
from PyARCL import FleetManager
EM = FleetManager('192.168.1.250')
```

Assumed 192.168.1.250 to be the EM IP address.
If the EM has a custom ARCL server configuration, using a different TCP socket number or a different password, then the password and/or port needs to be specified:

```
from PyARCL import FleetManager
FM = FleetManager('192.168.1.250', password='omron', port='8000')
```

Assumed the ARCL password and port to be relatively "omron" and 8000.
Note that since input parameters are ordered, the same object can be instantiated also by declaring:

```
from PyARCL import FleetManager
FM = FleetManager('192.168.1.250', 'omron', '8000')
```

May the user wish to do so.

5.1 FleetManager Class Unique Components

In this section there are described the methods and properties that describe a **FleetManager** class and are exclusive to a **FleetManager** object. Also check section 6 'Robot and FleetManager Classes Common Components' to get the full picture of the methods and properties callable by a **FleetManager** object.
Instantiate a **FleetManager** object as described in section 5 to start using them.

The FleetManager properties and method employ a queue-like system to achieve multithreading safety. The User may create one instance of a FleetManager object and access its properties and methods concurrently without fear of race conditions happening while parsing the ARCL responses.

5.1.1 FleetManager.robots attribute

The **robots** attribute is a dictionary used to store the **Robot** objects representing the AMRs connected to the Enterprise Manager. The dictionary uses the AMR identifier as key value and returns the full **Robot** object.

The robots dictionary is not auto-filled. The User may need to add manually the Robot objects to the robots dictionary by using the attachRobot method of the related FleetManager object.

Why is this useful?

The **robots** dictionary allows to quickly access a **Robot** object by using its identifier as a key, which is a value that is usually very easy to find attached to jobs and missions.

```
AMR = EM.robots[job.assignedRobot]
```

This allows the User to easily do things like the one shown in the example, where a job is sent to the Fleet Manager, only to be checked for progression and, once an AMR is assigned to it, point to the relative **Robot** object to get information about the AMR status.

Example

```
>>> import time
>>>
>>> Sim123 = Robot('192.168.1.123')
>>> Sim126 = Robot('192.168.1.126')
>>> FM116 = FleetManager('192.168.1.116')
>>>
>>> FM116.attachRobot(Sim123)
>>> FM116.attachRobot(Sim126)
>>>
>>> job = FM116.queuePickupDropoff('Goal107', 'Goal108')
>>>
>>> while True:
>>>     assigned_robot = job.assignedRobot
>>>     if assigned_robot != 'None': # Do nothing if job is currently unassigned
>>>         print(f'Robot assigned is {assigned_robot}')
>>>         assigned_robot_status = FM116.robots[assigned_robot].status
>>>         break
>>>     time.sleep(1)
>>>
>>> print(assigned_robot_status)

Robot assigned is Sim123
Status:
    Aram Status:           Going to Goal107
    State of Charge:       97.0

    Location:              X: 39831 [mm], Y: -17608 [mm], θ: 95 [deg]
    Localization Score:    1.0

    CPU Temperature:       -127.0
```

5.1.2 FleetManager.attachRobot Method

The **attachRobot** method is used to add a **Robot** object to the **FleetManager** object's **robots** attribute.

Arguments	Type	Default Value	Description
AMR	*Robot*		The Robot object to add to the robots dictionary.

Example

```
>>> Sim123 = Robot('192.168.1.123')
>>> FM116 = FleetManager('192.168.1.116')
>>>
>>> FM116.attachRobot(Sim123)
>>>
>>> Sim123_id = Sim123.config['Enterprise Manager Connection']['Identifier']
>>> print(FM116.robots[Sim123_id])

Robot: 192.168.1.123
    ID:    Sim123

    Status:
        Aram Status:           Parked
        State of Charge:       96.0

        Location:              X: 70335 [mm], Y: -13847 [mm], Θ: 0 [deg]
        Localization Score:    1.0

        CPU Temperature:       -127.0

    SetNetGo Ver.:   Robot-7.0.1
    ARAM Ver.:       7.0.4
    MARC Ver.:       3.0.0
```

6 Robot and FleetManager Classes Common Components

As explained back in section 3, in this section will be presented all methods, attributes and properties that are accessible by either a **Robot** or **FleetManager** object.
To get the full picture of all capabilities of the **Robot** and **FleetManager** classes, also check sections 4.1 and 5.1 respectively.

6.1 Robot/FleetManager.configSectionList Property

The **configSectionList** property returns a list of the names of all ARAM configuration sections of the relative AMR or EM the object is referring to.

Example

```
>>> EM.configSectionList

['ARCL server setup',
 'Connection timeouts',
 'Debug Features',
 'Enterprise Data Log Settings',
 'Enterprise Debug Features',
 'Enterprise Debug log',
 'Enterprise Features',
 'Enterprise Manager',
 'Extra Logging',
 'Files',
 'Fleet Docking',
 'Fleet Testing Features',
 'Internal Extra Logging',
 'Language/Location',
 'Log Config',
 'Map Features',
 'MultiRobot',
 'Outgoing Enterprise ARCL Commands',
 'Outgoing Enterprise ARCL Connection',
 'Queuing Manager',
 'Standby',
 'Testing Features']
```

6.2 Robot/FleetManager.config Property

The **config** property is used to access the AMR or EM ARAM configuration. ARAM configuration is accessed primarily through the MobilePlanner application but it can also be accessed through ARCL, which commands have been taken and wrapped in this property for the User convenience.

The config property returns a custom ConfigSectionDict container, dictionary-like object containing information updated to the moment the config property was called. Every time the config property is requested by the User, PyARCL automatically sends the 'getConfigSectionList' and 'getConfigSectionValues' ARCL commands accordingly and parses the answer to provide an updated response.

To access the ARAM configuration, simply declare:

```
>>> EM.config

Section: ARCL server setup
   ArclConfig: false
   OpenTextServer: true
   PortNumber: 7171
   Password: adept
   LogReceived: true
   LogSent: false
   SessionTimeoutMins: -1
   ArclScan: false
   ArclDisableMotors: false
   ArclFreeMotors: false
Section: Connection timeouts
   CentralServerToClientTimeoutInMins: 2
   CentralServerToRobotTimeoutInMins: 2
...
   NegativeClearances: false
   FollowGuideWithoutAcquire: false
   UseMovePath: false
   TestDockBusy: true
   DoNotEnsureProperSafetyCommissioning: false
```

The returned object is a **ConfigSectionDict** container, acting like a "dictionary of dictionaries". Indeed by requesting the config property to return a specific section configuration, it returns:

```
>>> EM.config['ARCL server setup']

{'ArclConfig': False,
 'OpenTextServer': True,
 'PortNumber': 7171,
 'Password': 'adept',
 'LogReceived': True,
 'LogSent': False,
 'SessionTimeoutMins': -1,
 'ArclScan': False,
 'ArclDisableMotors': False,
 'ArclFreeMotors': False}
```

It returns a simple dictionary item where the parameter names are the keys and the values are the ARAM configuration values as returned by ARCL.
This practically means that the User might access the value of a specific configuration parameter by double-nesting the "square brackets" (immediately accessing the returned dictionary):

```
>>> EM.config['ARCL server setup']['PortNumber']

7171
```

For **Robot** objects only, representing an AMR, configuration parameters might be set directly by assigning them a new value:

Configuration modification is only possible if the 'ArclConfig' parameter has been checked in the ARAM configuration. ARAM will return an error otherwise.

```
>>> AMR.config['ARCL server setup']['ArclConfig']

True

>>> AMR.config['Path Planning Settings']['MaxSpeed']

1500

>>> AMR.config['Path Planning Settings']['MaxSpeed'] = 900
>>> AMR.config['Path Planning Settings']['MaxSpeed']

900
```

Configuration is edited through the use of 'configStart', 'configAdd' and 'configParse' ARCL commands, which are used automatically by the config property to make it a seamless User experience.

This is an edge case of the use of PyARCL and the Robot class altogether. In general and for general purposes, configuration should be edited through the MobilePlanner application.

Example

In this example, the Python script checks whether it’s day or night and issues a map change accordingly.

```
>>> from datetime import datetime, time
>>>
>>> oldtiming = ''
>>> while True:
>>>     now = datetime.now().time()
>>>     if now >= time(23,00) or now <= time(8,00):
>>>         timing = 'night'
>>>     else:
>>>         timing = 'day'
>>>
>>>     if timing != oldtiming:
>>>         if timing == 'night':
>>>             print('Detecting day to night change, changing map')
>>>             FM.config['Files']['Map'] = 'Night_Map.map'
>>>        elif timing == 'day':
>>>             print('Detecting night to day change, changing map')
>>>             FM.config['Files']['Map'] = 'Day_Map.map'
>>>
>>>     time.sleep(60) # Check every minute

Detecting day to night change, changing map
```

6.3 Robot/FleetManager.datastoreGroupList property

The datastoreGroupList property returns a list of the names of all AMR or FM datastore groups of the relative AMR or EM the object is referring to.

Example

```
>>> AMR.datastoreGroupList

['Accelerometer',
 'ARCLStatus',
 'BatteryVoltage',
 'ChargingInfo',
 'CPUUse',
 'DateAndTime',
 'DebugLog',
 'Docking',
 'DockingInfo',
 'EncoderPose',
 'EncoderPoseInterpolated',
 'EstopStat',
 'JobCounts',
 'Laser',
 'LaserLocalization',
 'Memory',
 'ModeInfo',
 'PacketCount',
 'Path_Planning',
 'PathPlanningType',
 'QueueManagement',
 'RobotHeading',
 'RobotPose',
 'RobotPoseInterpolated',
 'RobotTemperature',
...
 'Uptime',
 'Velocities',
 'Versions',
 'WheelInfo',
 'WirelessStats']
```

6.4 Robot/FleetManager.datastore Property

The **datastore** property is used to access the AMR or EM datastore, which contains all sort of data regarding the AMR or EM operations.

The datastore property returns a custom DatastoreGroupDict container, dictionary-like object containing information updated to the moment the datastore property was called. Every time the config property is requested by the User, PyARCL automatically sends the 'getDataStoreGroupList' and 'getDataStoreGroupValues' ARCL commands accordingly and parses the answer to provide an updated response.

To access the AMR or EM datastore, simply declare:

```
>>> AMR.datastore

DataStore:
    Group: Accelerometer
        TipAngle: 0.000
        IsTipped: 0
...
    Group: ChargingInfo
        RobotChargeState: Not
        RobotChargeStateNumber: 0
    Group: CPUUse
        CPUUse: 34
    Group: DateAndTime
        DateAndTime: Tue Apr 25 14:00:04 2023
    Group: DebugLog
        DebugLogState: Not
        DebugLogSeconds: -1
...
        LeftEncoder: 0
        RightEncoder: 0
    Group: WirelessStats
        WirelessLink: 0
        WirelessQuality: 0
```

The returned object is a **DatastoreGroupDict** container, acting like a "dictionary of dictionaries". Indeed by requesting the datastore property to return the representation of a specific section, it returns:

```
>>> AMR.datastore['Path_Planning']

{'Distance_to_goal': 365.43,
 'Plan_Time_Ave': 5.2,
 'DWA_Time_Ave': 3.2,
 'Plan_Time_Instant': 20.0,
 'DWA_Time_Instant': 0.0,
 'Num_New_Obstacle_Points': 0,
 'Closest_Obstacle_Distance': -1,
 'Plan_Time_Limited_Lin_Vel': 1500,
 'Local_Plan_Area_Size_X': 5390,
 'Local_Plan_Area_Size_Y': 1750,
 'Desired_Lin_Vel_': 0,
 'Desired_Rot_Vel_': 0,
 'Desired_Heading_': 720}
```

It returns a simple dictionary item where the parameter names are the keys and the values are the datastore group values as returned by ARCL.
This practically means that the User might access the value of a specific datastore parameter by double-nesting the "square brackets" (immediately accessing the returned dictionary):

```
>>> AMR.datastore['Path_Planning']['Distance_to_goal']

365.43
```

Example

In this example, a simple script saves in a CSV file the data for robot pose and localization score every second for ten seconds. This basic example can be expanded to make a full-fledged custom datalogger for the User's AMR.

This example is not meant to hint to the User that they may prefer using PyARCL to build a custom datalogger instead of the implemented ARAM datalogger feature.
For production-level needs, always use the ARAM built-in datalogger.

```
>>> AMR = Robot('192.168.1.123')
>>> import csv
>>> import time
>>> from datetime import datetime
>>>
>>> with open('localization_log.csv', 'w', newline='') as file:
>>>     writer = csv.writer(file)
>>>     fields = ['DateTime', 'X', 'Y', 'Theta', 'LocScore']
>>>
>>>     writer.writerow(fields) # Writing headers
>>>
>>>     start_time = datetime.now()
>>>     while True:
>>>         AMR_pose = AMR.datastore['RobotPose']
>>>         AMR_loc = AMR.datastore['LaserLocalization']
>>>
>>>         AMR_x = AMR_pose['RobotX']
>>>         AMR_y = AMR_pose['RobotY']
>>>         AMR_th = AMR_pose['RobotTh']
>>>         AMR_locscore = AMR_loc['LaserLocalizationScore']
>>>         timestamp = str(AMR.dateTime)
>>>
>>>         writer.writerow([timestamp, AMR_x, AMR_y, AMR_th, AMR_locscore])
>>>
>>>         elapsed_secs = (datetime.now() - start_time).seconds
>>>         if elapsed_secs >= 10:
>>>             break
>>>
>>> with open('localization_log.csv', newline='') as file:
>>>     csv_contents = file.read()
>>>     print(csv_contents)

DateTime,X,Y,Theta,LocScore
2023-04-25 17:06:01,27203,-20590,-44.7,100
2023-04-25 17:06:02,27590,-20906,-34.0,99
2023-04-25 17:06:03,28028,-21137,-18.4,98
2023-04-25 17:06:04,28605,-21269,-4.2,97
2023-04-25 17:06:05,29336,-21267,3.1,99
2023-04-25 17:06:06,30387,-21215,0.6,100
2023-04-25 17:06:07,31406,-21208,-0.7,98
2023-04-25 17:06:08,32814,-21203,-1.3,100
2023-04-25 17:06:09,33853,-21201,0.2,95
2023-04-25 17:06:10,35494,-21179,1.1,96
2023-04-25 17:06:11,36886,-21100,10.8,97
```

6.5 Robot/FleetManager.infoList Property

The **infoList** property returns a list of the names of all AMR or FM “informations” of the relative AMR or EM the object is referring to. The “informations” are pieces of data the AMR or EM stores that is freely open to use by the user. There are default “informations” that hold system rundata but the real scope of the info system is for the user to store data directly inside the AMR or EM.

Example

```
>>> AMR.infoList

['RobotIP',
 'WirelessLink',
 'WirelessQuality',
 'Local Plan Time',
 'Local DWA Time',
 'Num New Obstacle Points',
 'Closest Obs Distance',
 'PlanTime Limited Lin Vel',
 'Local Plan Area Size X',
 'Local Plan Area Size Y',
 'Odometer(KM)',
 'HourMeter',
 'Temperature(C)',
 'LaserScore',
 'LaserLock',
 'LaserNumSamples',
 'LaserNumPeaks',
 'MPacs',
 'Laser_1 Pacs',
 'Laser_1 Filtered_v2',
 'SBC Uptime',
 'ARAM Uptime',
 'Idle',
 'TipAngle',
 'IsTipped',
 'Queue ID',
 'Queue Job ID',
 'DebugLogState',
 'DebugLogSeconds']
```

6.6 Robot/FleetManager.createInfo Method

The **createInfo** method generates an "information" stored inside the memory of the AMR or EM. These "informations" can be checked and manipulated through the **info** property.

Arguments	Type	Default Value	Description
newInfoName	*string*		The newly created info name.
maxLength	*int*	20	The info maximum character length.
startingValue	*string*	'None'	The starting value for the new info.

Note that the new "information" name cannot contain any spaces, despite standard (i.e. already present at bootup) "informations" being able to have names containing spaces. The User may use the underscore character "_" instead to separate words.

Due to the ARCL command limitation, the initial value of the "information" cannot contain any spaces, but the stored string value can be updated after info generation to contain spaces without problems.
In general, the recommendation is to handle "information" creation with a generation first and then a value set through the **info** property manipulation:

```
>>> AMR.createInfo('MyInfo')
>>> AMR.info['MyInfo'] = 'This is a nice info'
>>> print(AMR.info['MyInfo'])

'This is a nice info'
```

6.7 Robot/FleetManager.info Property

The **info** property is used to access the AMR or EM "informations", which contains all sort of data regarding the AMR or EM operations. The User might use and check "informations" at will, and modify the "informations" that they may create as they wish.

The info property returns a custom InfoDict container, dictionary-like object containing information updated to the moment the info property was called. Every time the info property is requested by the User, PyARCL automatically sends the 'getInfoList' and 'getInfo' ARCL commands accordingly and parses the answer to provide an updated response.

To access the AMR or EM information database, simply declare:

```
>>> AMR.info
Info:
    RobotIP: 192.168.1.123
    WirelessLink: false
    WirelessQuality: 100%
    Local Plan Time: 0
    Local DWA Time: 3
    Num New Obstacle Points: 7
    Closest Obs Distance: -1
    PlanTime Limited Lin Vel: 920
    Local Plan Area Size X: 3290
    Local Plan Area Size Y: 1330
    Odometer(KM): 7
...
    SBC Uptime: 456798
    ARAM Uptime: 99790
    Idle: 28132
    TipAngle: 0.000 deg
    IsTipped: false
    Queue ID:
    Queue Job ID:
    DebugLogState: Not
    DebugLogSeconds: -1
```

The returned object is an **InfoDict** container, acting like a simple dictionary.

Request the "information" value by simply using its name as the lookup key:

```
>>> AMR.info['ARAM Uptime']

100385
```

For custom "informations", either created by the User through the 'createInfo' ARCL command or the **createinfo** method, their value may be modified by simply setting their relative **info** property value:

```
>>> AMR.info['mycoolinfo']

'my info is very cool'

>>> AMR.info['mycoolinfo'] = 'and it can be edited'
>>> AMR.info['mycoolinfo']

'and it can be edited'
```

Example

The example script shows a custom program that is used easily to log the number of times a robot has gone to a goal in a fleet; using "informations" to make robots save goal reach data through after goal tasks and handle it through PyARCL.
This is how it works:

a. When a robot arrives at any goal, it triggers the 'After every goal' condition set in the 'Special' tasks tab, and it sets an "information" that has the same name of the reached goal to "1". This is done through the 'arbitraryServerSend' task that sends the 'updateInfo $g 1' ARCL command to the robot localhost IP.
b. PyARCL checks for the "informations" present inside the robot, and if an "information" that has the same name of a goal has been set to "1", then it updates the reached goal count database and sets the "information" back to "0".

```
>>> from PyARCL import Robot, FleetManager
>>> from datetime import datetime
>>> import time
>>>
>>> FM116 = FleetManager('192.168.1.116')
>>> Sim123 = Robot('192.168.1.123')
>>> Sim126 = Robot('192.168.1.126')
>>>
>>> FM116.attachRobot(Sim123)
>>> FM116.attachRobot(Sim126)
>>>
>>> # Init and set database
>>> goal_list = FM116.goalList
>>> goal_reach_dict = {}
>>> for robotname in FM116.robots:
>>>     goal_reach_dict[robotname] = {}
>>>     for goalname in goal_list:
>>>         goal_reach_dict[robotname][goalname] = 0
>>>
>>> # Create infos
>>> for robotname in FM116.robots:
>>>     robot = FM116.robots[robotname]
>>>     for goalname in goal_list:
>>>         robot.createInfo(goalname, maxLength=1, startingValue=0)
>>>
>>> # loop
>>> start_time = datetime.now()
>>> while True:
>>>     for robotname in FM116.robots:
>>>         robot = FM116.robots[robotname]
>>>         for goalname in goal_list:
>>>             if robot.info[goalname] == 1: # If info has been set to 1
>>>                 goal_reach_dict[robotname][goalname] += 1 # Update database
>>>                 robot.info[goalname] = 0 # Reset info back to zero
>>>
>>>     if (datetime.now() - start_time).seconds > 120: # Exit after two minutes
>>>         break
>>>     else:
>>>         time.sleep(5) # Wait 5 seconds before restarting loop
>>>
>>> print(goal_reach_dict)

{'Sim123': {'Goal1': 3, 'Goal3': 2, 'Goal2': 0}, 'Sim126': {'Goal1': 0, 'Goal3': 2,
'Goal2': 2}}
```

6.8 Robot/FleetManager.goalList property

The **goalList** property returns a list of the names of all the goals present in the map currently in use.

Example

```
>>> AMR.goalList
['Goal116',
 'Goal101',
 'Goal102',
 'Goal117',
 'Goal103',
 'Goal104',
 'Goal131',
 'Goal105',
 'Goal106',
 'Goal118',
 'Goal114',
 'Goal119',
 'Goal108',
 'Goal107',
 'Goal113',
 'Goal127',
 'Goal110',
 'Goal109',
 'Goal120',
 'Goal129',
 'Goal130',
 'Goal121',
 'Goal112',
 'Goal111',
 'Goal115',
...
 'Goal125',
 'Goal123',
 'Goal126',
 'Goal124',
 'Goal122']
```

6.9 Robot/FleetManager.queuePickup Method

The **queuePickup** method generates a single-segment job containing only a pickup segment. The method uses the following arguments:

Arguments	Type	Default Value	Description
missionGoal	*string*		The goal name to send to the pickup mission to.
priority	*int*	configuration default	The job segment priority.
customJobId	*string*		If specified, the string will be set as the custom job ID.

Note that the **queuePickup** method is only available for Fleet Managers and Standalone AMRs that have the 'Queuing Manager' option active in the ARAM configuration.

The queuePickup method returns a Job object, which can be used to check the real-time state of the job and its job segments, as well as modifying its segments goals and priorities at will. Check section 7 'The Job Class' for more information.

Example

```
>>> import time
>>> import datetime
>>> now = datetime.datetime.now()
>>> print(f'Job launched at {now}')
>>> job = AMR.queuePickup('Goal1', priority=50, customJobId='PyARCL_Test')
>>> while True:
>>>     if job.iscomplete:
>>>         break
>>>     time.sleep(1)
>>> now = datetime.datetime.now()
>>> print(f'Job finished at {now}')

Job launched at 2023-05-08 10:45:02.145368
Job finished at 2023-05-08 10:47:44.645287
```

6.10 Robot/FleetManager.queuePickupDropoff Method

The **queuePickupDropoff** method generates a simple job containing a single pickup and a single dropoff segment. The method uses the following arguments:

Arguments	Type	Default Value	Description
missionGoal1	*string*		The goal name to send to the pickup mission to.
missionGoal2	*string*		The goal name to send to the dropoff mission to.
priority1	*int*	configuration default	The pickup job segment priority.
priority2	*int*	configuration default	The dropoff job segment priority.
customJobId	*string*		If specified, the string will be set as the custom job ID.

Note that the **queuePickupDropoff** method is only available for Fleet Managers and Standalone AMRs that have the 'Queuing Manager' option active in the ARAM configuration.

The queuePickupDropoff method returns a Job object, which can be used to check the real-time state of the job and its job segments, as well as modifying its segments goals and priorities at will. Check section 7 'The Job Class' for more information.

Example

```
>>> import time
>>> import datetime
>>>
>>> now = datetime.datetime.now()
>>> print(f'Job launched at {now}')
>>>
>>> job = FM.queuePickupDropoff('Goal1', 'Goal2',
>>>                             priority1=50,
>>>                             priority2=100,
>>>                             customJobId='PyARCL_Test'
>>>                             )
>>>
>>> pickup_id = job.idList[0]
>>> dropoff_id = job.idList[1]
>>>
>>> while True:
>>>    if job[pickup_id].state == 'Completed':
>>>        break
>>>    time.sleep(1)
>>> now = datetime.datetime.now()
>>> print(f'Pickup segment finished at {now}')
>>>
>>> while True:
>>>    if job[dropoff_id].state == 'Completed':
>>>        break
>>>    time.sleep(1)
>>> now = datetime.datetime.now()
>>> print(f'Dropoff segment finished at {now}')

Job launched at 2023-05-08 17:35:21.558206
Pickup segment finished at 2023-05-08 17:36:01.417490
Dropoff segment finished at 2023-05-08 17:36:29.243926
```

6.11 Robot/FleetManager.queueMulti Method

The **queueMulti** is the go-to method to generates custom job containing any kind of segment type combination and length.
The method uses the following arguments:

Arguments	Type	Default Value	Description
missionGoalList	*list of str*		List containing the goal names of the entire mission.
typeList	*list of str*	Pickup-Dropoff alternation for all job segments.	List containing the type ('Pickup' or 'Dropoff') of each job segment.
priorityList	*list of int*	configuration default for both Pickup and Dropoff segment types.	List containing the priority of each job segment.
customJobId	*string*		If specified, the string will be set as the custom job ID.

Note that the **queueMulti** method is only available for Fleet Managers and Standalone AMRs that have the 'Queuing Manager' option active in the ARAM configuration.

The User may use the **queueMulti** method to handle any kind of job generation needs and may very well be the go-to method to use to generate easily any kind of AMR mission.
Essentially, the **queueMulti** method can be used to replace the previously shown **queuePickup** and **queuePickupDropoff** methods; indeed:

```
FM.queueMulti(['Goal1', 'Goal2'], ['Pickup', 'Dropoff'],[10, 20])
```

Generates the exact same job as:

```
FM.queuePickupDropoff('Goal1', 'Goal2', 10, 20)
```

And:

```
FM.queueMulti('Goal1', 'Pickup', 50)
```

Generates the exact same job as:

```
FM.queuePickup('Goal1', 50)
```

In this last example, the queueMulti method was passed strings and integers as arguments. When this happens, the method automatically generates and assigns lists accordingly with the User instruction. This should only be done in edge cases for readability.

The queueMulti method returns a Job object, which can be used to check the real-time state of the job and its job segments, as well as modifying its segments goals and priorities at will. Check 'The Job Class' chapter for more information.

Example

```
>>> import time
>>> import datetime
>>>
>>> now = datetime.datetime.now()
>>> print(f'Job launched at {now}')
>>>
>>> job = FM.queueMulti(['Goal105', 'Goal119', 'Goal125', 'Goal131', 'Goal110'],
>>>                     ['Pickup', 'Pickup', 'Dropoff', 'Dropoff', 'Dropoff'],
>>>                     [10, 20, 30, 40, 50],
>>>                     'PyARCL_test'
>>>                     )
>>>
>>> for segment_id in job.idList:
>>>     while True:
>>>         if job[segment_id].state == 'Completed':
>>>             break
>>>         time.sleep(1)
>>>     now = datetime.datetime.now()
>>>     segment_type = job[segment_id].type
>>>
>>>     print(f'{segment_type} segment finished at {now}')

Job launched at 2023-05-08 22:13:34.757766
Pickup segment finished at 2023-05-08 22:14:48.955204
Pickup segment finished at 2023-05-08 22:15:30.538700
Dropoff segment finished at 2023-05-08 22:16:29.416137
Dropoff segment finished at 2023-05-08 22:18:02.439478
Dropoff segment finished at 2023-05-08 22:19:07.201671
```

6.12 Robot/FleetManager.queueList property

The **goalList** property returns a list of the names of all the jobs currently present in the AMR or Fleet Manager memory.

To define how and how many jobs the Enterprise Manager or AMR should store inside its own memory, refer to the 'Queuing Manager' section of the ARAM configuration.

Example

```
>>> FM.queueList
['JOB18',
 'JOB16',
 'JOB34',
 'JOB39',
 'JOB44',
 'JOB49',
 'JOB54',
 'JOB59',
 'JOB64',
 'JOB69',
 'JOB74',
 'JOB79',
 'JOB84',
 'JOB89',
 'JOB94',
 'JOB99',
 'JOB104',
 'JOB109',
 'JOB114',
 'JOB119',
 'JOB24',
 'JOB29']
```

6.13 Robot/FleetManager.queue Property

The **queue** property is used to access the AMR or EM stored job memory. The job memory contains both completed, queued and in progress jobs, and can be accessed to monitor the amount of missions the robots are doing, have done and will do.

The queue property returns a custom JobDict container, dictionary-like object containing information updated to the moment the queue property was called. Every time the queue property is requested by the User, PyARCL automatically sends the 'queueList' and 'queueQuery' ARCL commands accordingly and parses the answer to provide an updated response.

To view a preview the robot job memory, simply declare:

```
>>> EM.queue

Jobs:
    [1] JOB18
        [1.1] PICKUP18
    [2] JOB16
        [2.1] PICKUP16
        [2.2] DROPOFF17
    [4] JOB34
        [4.1] PICKUP34
        [4.2] PICKUP35
        [4.3] DROPOFF36
        [4.4] DROPOFF37
        [4.5] DROPOFF38
    [5] JOB39
        [5.1] PICKUP39
        [5.2] PICKUP40
        [5.3] DROPOFF41
        [5.4] DROPOFF42
        [5.5] DROPOFF43
...
    [23] JOB56
        [23.1] PICKUP29
        [23.2] PICKUP30
        [23.3] DROPOFF31
        [23.4] DROPOFF32
        [23.5] DROPOFF33
```

The returned object is a **JobDict** container, acting like a dictionary. The User may access a single job stored in the job memory simply by calling it:

```
>>> FM.queue['JOB16']

Job: JOB16
    Completition: True

    Job Segments:
        [1]: PICKUP16
        [2]: DROPOFF17
```

It returns a **Job** object related to the job that was requested to the User. Refer to section 7 'The Job Class' for more information regarding **Job** objects and how to use them to check the data of the single missions.

Note that thanks to the nature of **Job** objects redirecting key access (table-lookup) to the inner 'jobSegments' container, the User may access a single segment of a specific job by double-nesting the "square brackets":

```
>>> FM.queue['JOB16']['PICKUP16']

JobSegment: PICKUP16
    JobID:          JOB16
    JobType:        Pickup

    Priority:       50
    State:          Completed
    Substate:       None
    Goal:           Goal115
    AssignedRobot:  Sim126
    QueuedTime:     2023-05-08 16:02:06
    FinishedTime:   2023-05-08 16:03:13
    FailNumber:     0
```

Which returns a **JobSegment** object that can then be accessed by the User as normal. Refer to section 8 'The JobSegment Class' for more information regarding **JobSegment** objects.

Example

In this section it will be shown a script that when run will get all data regarding the job memory stored inside a Fleet Manager and saves it in a CSV file.

```
>>> import csv
>>> with open('job_queue.csv', 'w', newline='') as file:
>>>     writer = csv.writer(file)
>>>     fields = ['Job_ID', 'JobSegment_ID', 'JobType', 'Priority', 'State',
>>>               'Substate', 'Goal', 'Assigned_Robot', 'Queued_Time',
>>>               'Finished_Time', 'Fail_Number']
>>>     writer.writerow(fields) # Writing headers
>>>
>>>     for job_name in EM.queueList:
>>>         for jobsegment_name in EM.queue[job_name].idList:
>>>             data = []
>>>
>>>             jobsegment = EM.queue[job_name][jobsegment_name]
>>>             data.append(job_name)
>>>             data.append(jobsegment_name)
>>>             data.append(jobsegment.type)
>>>             data.append(jobsegment.priority)
>>>             data.append(jobsegment.state)
>>>             data.append(jobsegment.substate)
>>>             data.append(jobsegment.goal)
>>>             data.append(jobsegment.assignedRobot)
>>>             data.append(jobsegment.queuedTime)
>>>             data.append(jobsegment.finishedTime)
>>>             data.append(jobsegment.failNumber)
>>>
>>>             writer.writerow(data)
>>>
>>> with open('job_queue.csv', newline='') as file:
>>>     csv_contents = file.read()
>>>     print(csv_contents)

Job_ID,JobSegment_ID,JobType,Priority,State,Substate,Goal,Assigned_Ro-
bot,Queued_Time,Finished_Time,Fail_Number
JOB124,PICKUP124,Pickup,10,Completed,None,Goal102,Sim123,2023-05-09 04:31:42,2023-
05-09 04:32:23,0
JOB124,DROPOFF125,Dropoff,20,Completed,None,Goal115,Sim123,2023-05-09
04:31:42,2023-05-09 04:33:04,0
JOB124,PICKUP126,Pickup,10,Completed,None,Goal130,Sim123,2023-05-09 04:31:42,2023-
05-09 04:33:43,0
...
JOB200,DROPOFF201,Dropoff,20,Completed,None,Goal115,Sim123,2023-05-09
04:31:52,2023-05-09 05:04:36,0
JOB200,PICKUP202,Pickup,10,Completed,None,Goal130,Sim123,2023-05-09 04:31:52,2023-
05-09 05:05:15,0
JOB200,DROPOFF203,Dropoff,20,Completed,None,Goal102,Sim123,2023-05-09
04:31:52,2023-05-09 05:06:21,0
```

6.14 Robot/FleetManager.dateTime property

The **DateTime** property returns a **datetime** object relative to the AMR or EM onboard RTC date and time.

Example

```
>>> AMR.dateTime

datetime.datetime(2023, 5, 6, 10, 28, 4)
```

6.15 Robot/FleetManager.tripReset method

The **tripReset** method is used to command the AMR or EM to reset their database values specific to the current trip.
The method has no attributes, simply call it by declaring:

```
EM.tripReset()
```

7 The Job Class

The **Job** class is used to represent an AMR job, containing various segments either of pickup or dropoff type.
This class is not meant to be used directly by the User to describe missions, but it's used by the "queue-like" methods and properties of the **Robot** and **FleetManager** objects to return fully instantiated **Job** objects.

7.1 Job.jobid Attribute

The **jobid** attribute is a constant which holds the relative job unique ID.
Access the **jobid** attribute simply by declaring:

```
>>> job.jobid
'JOB210'
```

7.2 Job.idList Attribute

The **idList** attribute is a list which holds the IDs of all the job segments related to the accessed job.
Access the **idList** attribute simply by declaring:

```
>>> job.idList
['PICKUP210', 'DROPOFF211', 'PICKUP212', 'DROPOFF213']
```

7.3 Job.jobSegments Container

The **jobSegments** container is functionally identical to a dictionary, requesting as keys the job segments names (as provided by the **idList** attribute) and returning the relative **JobSegment** objects.
Check section 8 'The JobSegment Class' to get more information on how to extract data from **JobSegment** objects.

The User may access the **jobSegments** container either by simply declaring it:

```
>>> job.jobSegments

Job Segments:
    [1]: PICKUP210
    [2]: DROPOFF211
    [3]: PICKUP212
    [4]: DROPOFF213

>>> job.jobSegments['DROPOFF211']

JobSegment: DROPOFF211
    JobID:          JOB210
    JobType:        Dropoff

    Priority:       20
    State:          Completed
    Substate:       None
    Goal:           Goal109
    AssignedRobot:  Sim123
    QueuedTime:     2023-05-09 06:20:31
    FinishedTime:   2023-05-09 06:21:51
    FailNumber:     0
```

Or by directly accessing the parent **Job** object through key-access:

```
>>> job['DROPOFF211']

JobSegment: DROPOFF211
    JobID:          JOB210
    JobType:        Dropoff

    Priority:       20
    State:          Completed
    Substate:       None
    Goal:           Goal109
    AssignedRobot:  Sim123
    QueuedTime:     2023-05-09 06:20:31
    FinishedTime:   2023-05-09 06:21:51
    FailNumber:     0
```

Examples

In this section there are presented two simple examples of the typical usage of the **jobSegments** container:

1. The first example just shows how the **jobSegments** property can be used to print in-depth information about the job at hand.
2. The second example shows an application where a job is sent only after the first segment of another job is finished.

Example 2. Job representation

```
>>> job = EM.queueMulti(['Goal105', 'Goal125', 'Goal103'])
>>> print(job.jobSegments)
>>>
>>> for job_segment_name in job.jobSegments:
>>>     print(job.jobSegments[job_segment_name])

Job Segments:
    [1]: PICKUP223
    [2]: DROPOFF224
    [3]: PICKUP225
JobSegment: PICKUP223
    JobID:          JOB223
    JobType:        Pickup

    Priority:       10
    State:          Pending
    Substate:       None
    Goal:           Goal105
    AssignedRobot:  None
    QueuedTime:     2023-05-09 11:16:19
    FinishedTime:   None
    FailNumber:     0
...
```

Example 2. Job dispatch handling

```
>>> import time
>>>
>>> job = EM.queueMulti(['Goal1', 'Goal2', 'Goal3'])
>>>
>>> while True:
>>>     # Exit the loop if the first job segment is complete
>>>     if job[job.idList[0]].state == 'Completed':
>>>         break
>>>     time.sleep(1)
>>>
>>> # Send the other job after the first segment of the previous mission is done
>>> job2 = EM.queueMulti(['Goal1', 'Goal4', 'Goal5'])
```

7.4 Job.assignedRobot Property

The **assignedRobot** property returns the name of the robot currently assigned to the accessed job.
Access the **assignedRobot** value by simply declaring:

```
>>> job.assignedRobot
'Sim123'
```

The assignedRobot property returns a value updated to the moment the assignedRobot property was called. Every time the assignedRobot property is requested by the User, PyARCL automatically sends the relative 'queueQuery' ARCL command and parses the answer to build an updated response.

7.5 Job.iscomplete Property

The **iscomplete** property returns the value of competition of the job. If the last job segment is completed by an AMR, then the job is considered complete.
Access the **iscomplete** value by simply declaring:

```
>>> job.iscomplete
True
```

The iscomplete property returns a value updated to the moment the iscomplete property was called. Every time the iscomplete property is requested by the User, PyARCL automatically sends the relative 'queueQuery'ARCL command and parses the answer to build an updated response.

7.6 Job.cancel Method

The **cancel** method is used to cancel the accessed job, as if it were sent the 'queueCancel' ARCL command.
The method has no attributes, simply call it by declaring:

```
job.cancel()
```

8 The JobSegment Class

The **JobSegment** class is used to represent a job segment, either of pickup or dropoff type.
This class is not meant to be used directly by the User to describe sub-missions, but it's used by the **Job** objects to return fully instantiated **JobSegment** objects.

8.1 JobSegment.id Attribute

The id attribute is a constant which holds the relative job segment unique ID.
Access the id attribute simply by declaring:

```
>>> job.jobSegments[job_segment_name].id

'PICKUP225'
```

8.2 JobSegment.jobid Attribute

The **jobid** attribute is a constant which holds the parent job unique ID, which the job segment belongs to.
Access the **jobid** attribute simply by declaring:

```
>>> job.jobSegments[job_segment_name].jobid

'JOB223'
```

8.3 JobSegment.assignedRobot Property

The **assignedRobot** property returns the name of the robot currently assigned to the accessed job segment.
Access the **assignedRobot** value by simply declaring:

```
>>> job.jobSegments[job_segment_name].assignedRobot

'Sim123'
```

The assignedRobot property returns a value updated to the moment the assignedRobot property was called. Every time the assignedRobot property is requested by the User, PyARCL automatically sends the relative 'queueQuery' ARCL command and parses the answer to build an updated response.

8.4 JobSegment.state Property

The **state** property returns the state of the accessed job segment.
Access the **state** value by simply declaring:

```
>>> job.jobSegments[job_segment_name].state

'InProgress'
```

Note that the job segment most common states are:

- Pending
- InProgress
- Completed
- Cancelled

The state property returns a value updated to the moment the state property was called. Every time the state property is requested by the User, PyARCL automatically sends the relative 'queueQuery' ARCL command and parses the answer to build an updated response.

8.5 JobSegment.substate Property

The **substate** property returns the substate of the accessed job segment.
Access the **substate** value by simply declaring:

```
>>> job.jobSegments[job_segment_name].substate

'Driving'
```

The substate property returns a value updated to the moment the substate property was called. Every time the substate property is requested by the User, PyARCL automatically sends the relative 'queueQuery' ARCL command and parses the answer to build an updated response.

8.6 JobSegment.type Property

The **type** property returns the job-type of the accessed job segment. Access the **type** value by simply declaring:

```
>>> job.jobSegments[job_segment_name].type

'Dropoff'
```

Note that the job segment types can either be:

- Pickup
- Dropoff

8.7 JobSegment.queuedTime Property

The **queuedTime** property returns the **datetime** object representing the time at which the accessed job segment was queued. Access the **queuedTime** value by simply declaring:

```
>>> job.jobSegments[job_segment_name].queuedTime

datetime.datetime(2023, 5, 9, 11, 36, 25)
```

Which can be represented as:

```
>>> print(job.jobSegments[job_segment_name].queuedTime)

2023-05-09 11:36:25
```

8.8 JobSegment.finishedTime Property

The **finishedTime** property returns the **datetime** object representing the time at which the accessed job segment was finished. Access the **finishedTime** value by simply declaring:

```
>>> job.jobSegments[job_segment_name].finishedTime
datetime.datetime(2023, 5, 9, 11, 37, 53)
```

Which can be represented as:

```
>>> print(job.jobSegments[job_segment_name].finishedTime)
2023-05-09 11:37:53
```

The substate property returns a value updated to the moment the substate property was called. Every time the substate property is requested by the User, PyARCL automatically sends the relative 'queueQuery' ARCL command and parses the answer to build an updated response.

8.9 JobSegment.failNumber Property

The **failNumber** property returns the number of times the job was failed by an AMR whilst assigned to it.
Access the **failNumber** value by simply declaring:

```
>>> job.jobSegments[job_segment_name].failNumber
9
```

The failNumber property returns a value updated to the moment the failNumber property was called. Every time the failNumber property is requested by the User, PyARCL automatically sends the relative 'queueQuery' ARCL command and parses the answer to build an updated response.

8.10 JobSegment.goal Property

The **goal** property returns the goal name associated to the accessed job segment.
Access the **goal** value by simply declaring:

```
>>> job.jobSegments[job_segment_name].goal
'Goal1'
```

Note that the **goal** property can also be set to modify in real time the job segment goal.

```
>>> job.jobSegments[job_segment_name].goal
'Goal1'
>>> job.jobSegments[job_segment_name].goal = 'Goal2'
>>> job.jobSegments[job_segment_name].goal
'Goal2'
```

The goal associated to the job segment can only be modified if the job segment state is either 'InProgress' or 'Pending'. If the job segment has already been completed or cancelled, it can no longer be modified.

The goal property returns a value updated to the moment the goal property was called. Every time the goal property is requested or set by the User, PyARCL automatically sends the relative 'queueQuery' or 'queueModify' ARCL command respectively and parses the answer to build an updated response.

8.11 JobSegment.priority Property

The **priority** property returns the priority value (as an integer) associated to the accessed job segment.
Access the **priority** value by simply declaring:

```
>>> job.jobSegments[job_segment_name].priority

10
```

Note that the **priority** property can also be set to modify in real time the job segment priority.

```
>>> job.jobSegments[job_segment_name].priority

10

>>> job.jobSegments[job_segment_name].priority = 50
>>> job.jobSegments[job_segment_name].priority

50
```

The priority associated to the job segment can only be modified if the job segment state is either 'InProgress' or 'Pending'. If the job segment has already been completed or cancelled, it can no longer be modified.

The priority property returns a value updated to the moment the priority property was called. Every time the priority property is requested or set by the User, PyARCL automatically sends the relative 'queueQuery' or 'queueModify' ARCL command respectively and parses the answer to build an updated response.

9 AMRSimTools Components

The **AMRSimTools** module provides a range of functions to aid feasibility study-aimed simulations through the use of *OMRON Enterprise Manager* running in *Simulator Mode*.

In this section, there will be presented all of the functions meant to be used by the User, as well as some examples of real "job dispatching scripts" that can be used to simulate the continue calls of a pull request system.

Check also the **Cookbook** at the end for more examples regarding typical job dispatching scripts of feasibility studies and fleet dimensioning-purpose simulations.

9.1 AMRSimTools.program

The **program** function is designed to help the User instantiate a multi-threading system for AMR fleet feasibility/dimensioning simulation purposes.
The main way the **program** function can be used is to mimic a pull manufacturing system, continuously requesting missions to be done by the AMR in the fleet.

Arguments	Type	Default Value	Description
main	*function*		Main function that will be executed in the new thread.
repeatTime	*Int*	None	If specified, the program function will generate a new thread identical to the first (still targeting main function) after the amount of seconds entered.
arguments	*tuple*	Empty tuple	Tuple containing the arguments of the main function in case a function requiring arguments was passed as the main function.

Note that it's best to pass a zero-arguments function to the program function for multithreading purposes. There are tricks to get around this limitation and they are shown in the following paragraphs.

Why is This Useful?

To create a thread whose scope will be to only send a mission to the fleet manager every *n* seconds, the following template may be used:

```
from PyARCL import FleetManager
from PyARCL.AMRSimTools import program, wait

EM = FleetManager("192.168.1.115")

def loop():
    n = 60 # seconds
    while True:
        EM.queueMulti(['Goal1', 'Goal2'])
        wait(n)

program(loop)
```

Where in this example *n* was 60 seconds.
Note that if the simulation were to need two concurrent pull requests, one every *n* seconds and the other every *m* seconds, and requiring different goals for their missions, then the User would need to create one loop for each.

To avoid creating several functions that differ only by a little bit, while retaining the use of zero-arguments functions as the main **program** function, the following, more general template may be used:

```
from PyARCL import FleetManager
from PyARCL.AMRSimTools import program, wait

EM = FleetManager("192.168.1.115")

def machine(name, n):
    def missiongen():
        while True:
            goals = ['Goal1', name]
            EM.queueMulti(goals)
            wait(n)
    return missiongen

program(machine("Goal2", 60))
program(machine("Goal3", 30))
program(machine("Goal4", 45))
program(machine("Goal5", 120))
```

In this case, calling the **machine** function inside the **program** declaration, returns immediately the customized, corresponding **missiongen** function, which is a zero-parameters function generated automatically by **machine**.

The User may take this template for almost all very simple mission generators that do not need to check the sent job state or do something particular with the assigned robot.

Let's say the User may need to use a more complex job generator script; for example one that mimics a job-shop process: The mission generator will need to send each *n* seconds a mission that picks up the semi-finished goods at "Goal1", brings them to the correct work machine "WorkGoalX", and when the work is finished then it brings them to the warehouse "Goal2".
The difficult part here is that there is no way to know how much time it will take for the first mission to complete beforehand, and therefore it may be difficult to reuse the previously shown scripts that simply make use of a **wait** command to restart the loop. The User may indeed think about a similar solution such as this one:

```python
from PyARCL import FleetManager
from PyARCL.AMRSimTools import program, wait, waitForJob
from datetime import datetime

EM = FleetManager("192.168.1.115")

def machine(name, n):
    def missiongen():
        while True:
            tic = datetime.now()

            goals = ['Goal1', name]
            job = EM.queueMulti(goals)
            waitForJob(job)
            wait(60) # Wait for machine to finish
            newgoals = [name, 'Goal2']
            EM.queueMulti(newgoals)

            toc = datetime.now()
            delta_t = (toc-tic).seconds
            wait_n = n - delta_t
            if wait_n > 0:
                wait(wait_n)

    return missiongen

program(machine("WorkGoal1", 60))
program(machine("WorkGoal2", 30))
program(machine("WorkGoal3", 45))
program(machine("WorkGoal4", 120))
```

Where a tic-toc count is used to measure the in-between time and adjust the wait time accordingly, but this only works if the adjusting time is way less than the original time interval *n*.

Fortunately, the **program** function allows to natively handle these cases in a cleaner and more functional way, by using the **repeatTime** parameter:

```
from PyARCL import FleetManager
from PyARCL.AMRSimTools import program, wait, waitForJob

EM = FleetManager("192.168.1.115")

def machine(name):
    def missiongen():
        goals = ['Goal1', name]
        job = EM.queueMulti(goals)
        waitForJob(job)
        wait(60) # Wait for machine to finish
        newgoals = [name, 'Goal2']
        EM.queueMulti(newgoals)
    return missiongen

program(machine("WorkGoal1"), repeatTime=60)
program(machine("WorkGoal2"), repeatTime=30)
program(machine("WorkGoal3"), repeatTime=45)
program(machine("WorkGoal4"), repeatTime=120)
```

By using the **repeatTime** argument the **missiongen** returned function is instantiated every *n* seconds, each one in a different thread that will complete all the associated actions independently one from another.

For more complex examples on mission generators, check the **Cookbook** section at the end.

9.2 AMRSimTools.randomNumber

The **randomNumber** function is only present for readability of code. It wraps around the **random.randomint** function.
It's a simple function of two arguments:

Arguments	Type	Default Value	Description
n1	*Int*		Lower limit for random integer generation
n2	*int*		Upper limit for random integer generation

The function simply returns a random integer between n1 and n2.

Example

```
>>> from PyARCL.AMRSimTools import randomNumber
>>> for i in range(10):
>>>     print(randomNumber(1, 10))

10
4
5
10
6
10
7
1
10
8
```

9.3 AMRSimTools.wait

The **wait** function is only present for readability of code. It wraps around the **time.sleep** function.
It's a simple function of a single argument:

Arguments	Type	Default Value	Description
secs	*Int*		Number of seconds to wait

The function simply makes the interpreter wait the number of seconds specified before continuing.

9.4 AMRSimTools.waitForInput

The **waitForInput** function waits for a robot to raise a specific input before continuing.

Arguments	Type	Default Value	Description
robot	*Robot*		Robot object to which check the input.
inputName	*string*		Input name to check
interval	*int*	1	Time interval in which the internal loop checks for the input to raise. The value is to be intended in seconds.

Example

```
>>> AMR = Robot('192.168.1.123')
>>> print('Waiting for o1 to raise')
>>> waitForInput(AMR, 'o1')
>>> print('o1 has been raised!')

Waiting for o1 to raise
o1 has been raised!
```

9.5 AMRSimTools.waitForJob

The **waitForJob** function waits for a specific mission to be completed before continuing.

Arguments	Type	Default Value	Description
job	*Job*		Job object to check completion of.
interval	*int*	1	Time interval in which the internal loop checks for the Job object's iscomplete property to become true. The value is to be intended in seconds.

This is done by continuously checking the internal **Job** object's **iscomplete** property to become 'True'.

Example

```
>>> from PyARCL.AMRSimTools import waitForJob
>>> import datetime
>>> now = datetime.datetime.now()
>>> print(f'Job launched at {now}')
>>> job = AMR.queueDropoff('Goal1', priority=50, customJobId='PyARCL_Test')
>>> waitForJob(job)
>>> now = datetime.datetime.now()
>>> print(f'Job finished at {now}')

Job launched at 2023-05-16 18:27:40.821223
Job finished at 2023-05-16 18:28:22.274812
```

9.6 AMRSimTools.waitForJobState

The **waitForJobState** function waits for a specific job segment's state to become a specific value before continuing.

Arguments	Type	Default Value	Description
job	*Job*		Job object to check the JobSegment state of.
desiredState	*String*		The desired state of the job segment
jobSegmentNumber	*int*	-1	The number of the job segment to check as the order of execution, which incidentally is the same of the idList property.
interval	*int*	1	Time interval in which the internal loop checks for the JobSegment object's state property to become equal to desiredState. The value is to be intended in seconds.

This is done by continuously checking the referenced **JobSegment** object's **state** attribute to become equal to the desiredState argument.
The **JobSegment** object is extracted from the passed **Job** object through the jobSegmentNumber argument; by default, the function checks for the last job segment's state.

Example

```
>>> import datetime
>>> from PyARCL.AMRSimTools import waitForJobState
>>>
>>> now = datetime.datetime.now()
>>> print(f'Job launched at {now}')
>>>
>>> job = FM.queueMulti(['Goal1', 'Goal2', 'Goal3', 'Goal4', 'Goal5'],
>>>                     ['Pickup', 'Pickup', 'Dropoff', 'Dropoff', 'Dropoff'],
>>>                     [10, 20, 30, 40, 50],
>>>                     'PyARCL_test'
>>>                     )
>>>
>>> for i, segment_id in enumerate(job.idList):
>>>     waitForJobState(job, 'Completed', i)
>>>     now = datetime.datetime.now()
>>>     segment_type = job[segment_id].type
>>>
>>>     print(f'{segment_type} segment finished at {now}')

Job launched at 2023-05-16 18:46:09.179853
Pickup segment finished at 2023-05-16 18:47:03.303729
Pickup segment finished at 2023-05-16 18:47:54.911139
Dropoff segment finished at 2023-05-16 18:48:35.833732
Dropoff segment finished at 2023-05-16 18:49:25.902260
Dropoff segment finished at 2023-05-16 18:50:15.671865
```

10 AMRFrontEnds Components

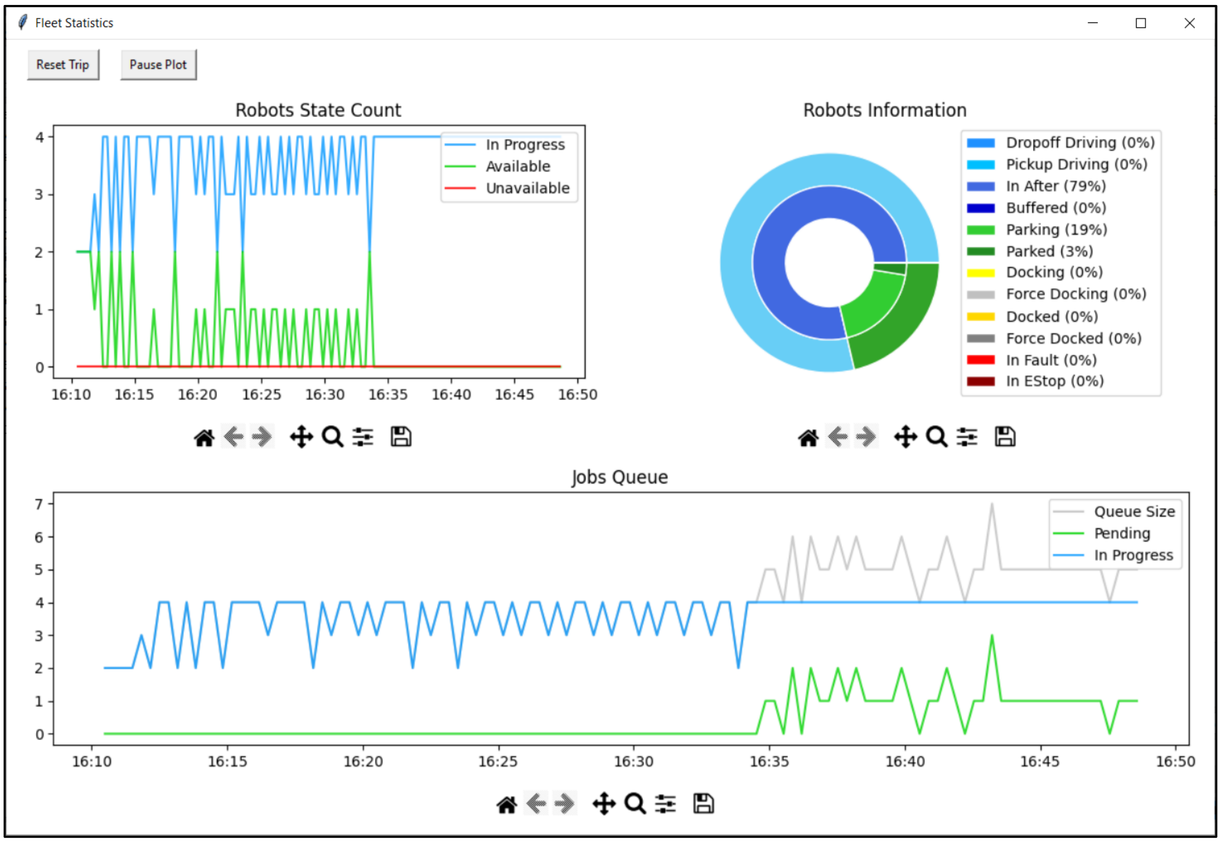
The **AMRSimTools** module provides dashboard components to use in combination with the User's AMR and Fleet Managers.

Note that this module currently needs the matplotlib package to be installed. Install matplotlib by running the pip install command in the command window:

```
pip install matplotlib
```

10.1 AMRFrontEnds.FleetStatsGUI Class

The **FleetStatsGUI** class, when used to instantiate an object, it generates a window with useful stats regarding the ongoings of the application:



Note that the interpreter holds when instantiating the window, until the generated window is closed. If the User puts some code after the FleetStatsGUI class is used to instantiate an object, it will be run after the generated window is closed.

The **FleetStatsGUI** class constructor requires the following arguments:

Attribute	Type	Default Value	Description
FM	*FleetManager*		The FleetManager object to plot statistics for.
delta_t	*int*	20	The refresh rate at which data is sampled and printed.
reset_trip	*bool*	True	Boolean that controls whether or not to run the tripReset method of the FleetManager object before starting saving statistics.

Instantiate a **FleetStatsGUI** object by declaring:

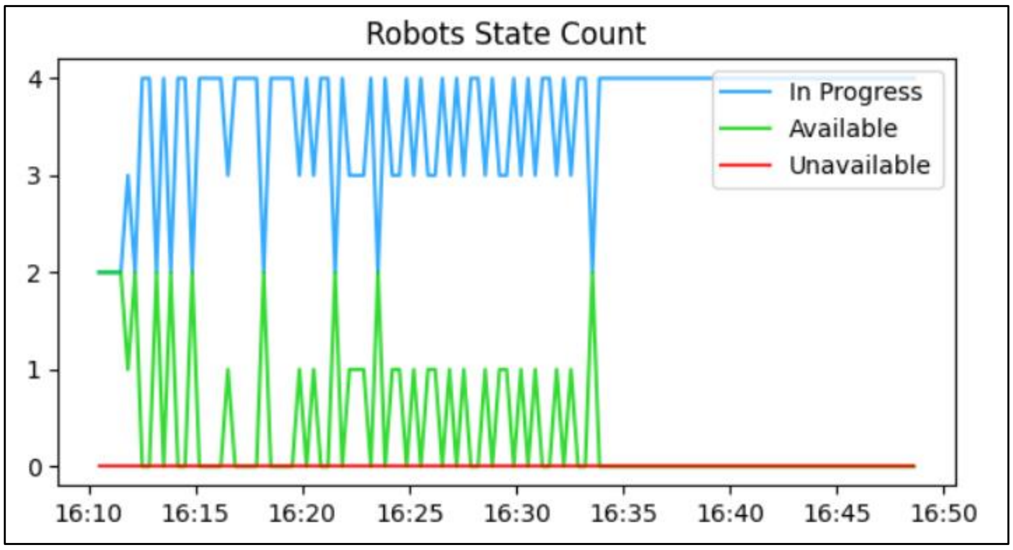
```
from PyARCL import FleetManager
from PyARCL.AMRFrontEnds import FleetStatsGUI

FM = FleetManager('192.168.1.115')
FleetStatsGUI(FM, delta_t=2, reset_trip=False)
```

10.1.1 Window Components

In this section are explained the various buttons and plots present in the window generated by a **FleetStatsGUI** object.

Robot State Counts Plot

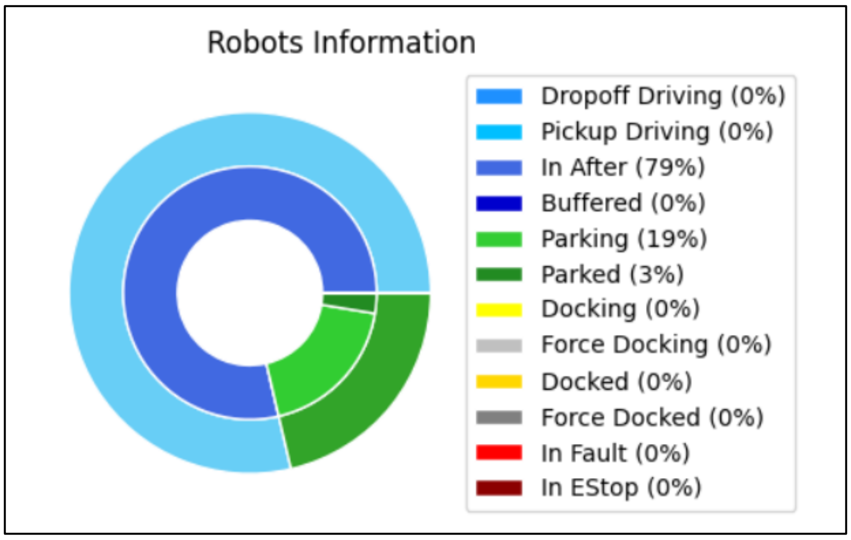


The Robot State Counts plot represents the state history of the AMRs connected to the fleet.

The plot is composed of three lines:

Color	Description
🟩 **Green**	Available Robots, in Idle state.
🟦 **Blue**	Robot at work, either executing tasks or going at a specific goal.
🟥 **Red**	Robot in fault, either lost or with an active ARAM Fault.

Robots Information Pie Chart

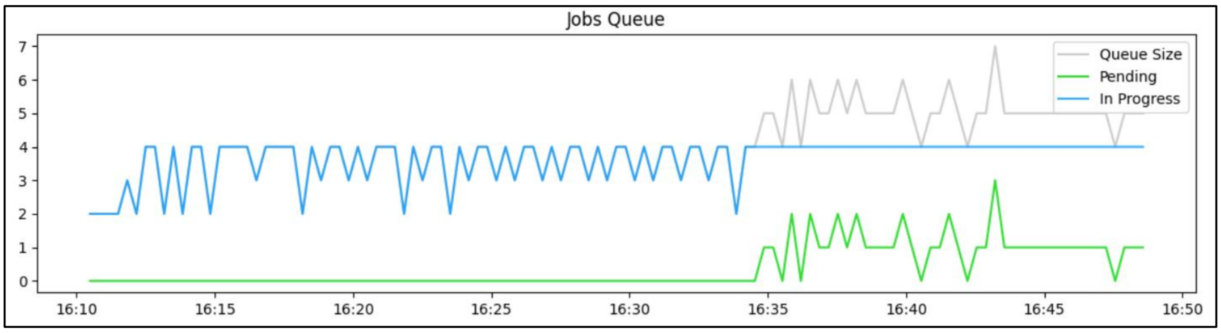


The Robots Information pie chart represents the statistics on the states of the AMRs connected to the fleet.

There are two main circles:

- The outer circle shows the main states of the AMRs, those being:
    - 🟩 Green: Idle
    - 🟨 Yellow: Charging
    - 🟦 Blue: Working
    - 🟥 Red: In Fault
- The inner circle shows the in-depth substates of the main states described in the outer circle.

Jobs Queue Plot

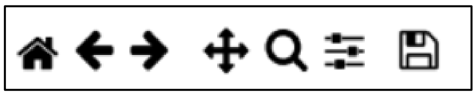


The Jobs Queue plot represents the history of the missions in the fleet queue.

The plot is composed of three lines:

Color	Description
🟩 **Green**	Missions pending in the queue. No robots are assigned to these jobs.
🟦 **Blue**	Missions that are being worked on, a robot is currently assigned to each one.
⬛ **Grey**	The total number of missions in the queue, being the sum of the blue and green line.

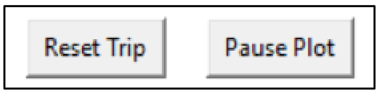
Plot Navigation Bar



The Navigation bar is present for each of the previously presented plots, and is used to interact with the graphs:

Button	Command	Description
🏠	Home View	Reset the view to the original one
←	Previous View	Move to the previous view, undo graph modification
→	Next View	Move to the next view, redo graph modification
✥	Pan	Move the graph view around by panning
🔍	Zoom	Select an area to zoom the plot
☰	Configure	Configure values regarding how the plot is displayed
💾	Save	Save a .PNG image file of the current graph view

Action Buttons



The action buttons at the top of the window allow the user to interact with the graph plotting methods:

Button	Description
Reset Trip	Reset graph data, and issue a resetTrip command to the relative FleetManager object used to create the FleetStatsGUI object.
Pause Plot	Pause the plot refresh of the graphs. Press again to resume.

It is advised to pause the plot of the graphs when attempting to save a graph PNG image file through the use of the Navigation bar.
This is done to prevent the plot refreshing and modifying eventual zooms/pans the User might have done to better show the data of interest.

11 The CookBook

The aim of this section is to give the User a jump start into PyARCL, getting into knowing real possibilities and applications that this package can offer.

The idea in which this CookBook is built is to give the User a number of copy-pastable examples to start quickly using PyARCL, allowing they to study and to learn practically the design patterns to follow.

As of PyARCL v.1, the CookBook section is still under construction. Please understand.

11.1 Simulation Scripts

In this section there will be presented scripts strictly related to the world of AMR simulations, with aim of fleet dimensioning and feasibility assessment.

11.1.1 General Template for Simulation Scripts

This script is the typical example that is used to instance a job-dispatcher script for AMR simulations, that instances a **FleetStatsGUI** window and stops the mission generation once the window is closed.

```
from PyARCL import FleetManager
from PyARCL.AMRSimTools import program, wait
from PyARCL.AMRFrontEnds import FleetStatsGUI

EM = FleetManager("192.168.1.115")

stop = False
pick_n = 1

def machine(drop_n, t):
    global stop
    def missiongen():
        while not stop:
            EM.queueMulti([f'Pick{pick_n}', f'Drop{drop_n}'])
            wait(t)
    return missiongen

program(machine(1, 1*60))
program(machine(2, 2*60))
program(machine(3, 3*60))
program(machine(4, 4*60))
program(machine(5, 5*60))
program(machine(6, 6*60))
program(machine(7, 7*60))
program(machine(8, 8*60))

FleetStatsGUI(EM)

stop = True
```

The 'stop' global variable indeed triggers the end of the job-generating threads, but it's only reached once the '**FleetStatsGUI**' window is closed, as explained in section 10.1.

11.2 Miscellaneous Scripts

In this section will be presented more miscellaneous scripts, for all purposes and intentions.

11.2.1 Custom Docking Program

This is an example of a custom program that defines logic to send a fleet to dock based on these instructions:

a. If a robot is idle, send it to dock.
b. If a robot is doing a job, check whether it has already completed the first pickup segment:
    i. If the first segment is completed, wait for the entire job to finish, then send the robot to dock.
    ii. If the first segment is not completed, send the robot to dock immediately.

Note: for this program to work correctly, the 'TestDockBusy' configuration parameter needs to be checked.

```python
from PyARCL import Robot, FleetManager
from threading import Thread
import time

FM116 = FleetManager('192.168.1.116')
Sim123 = Robot('192.168.1.123')
Sim126 = Robot('192.168.1.126')

FM116.attachRobot(Sim123)
FM116.attachRobot(Sim126)

def dock_after_complete(robot, job):
    def handler():
        while True:
            if job.iscomplete:
                break
            else:
                time.sleep(1)
        robot.dock()
    return handler

def fleet_dock():
    for robotname in FM116.robots:
        robot = FM116.robots[robotname]
        current_job = robot.info['Queue Job ID']
        if current_job == '':
            robot.dock()
        else:
            cur_job_obj = FM116.queue[current_job]
            first_cur_job_seg = cur_job_obj.idList[0]
            if cur_job_obj.jobSegments[first_cur_job_seg].state == 'Complet-ed':
                t = Thread(target=dock_after_complete(robot, cur_job_obj))
                t.start()
            else:
                robot.dock()

def fleet_undock():
    for robotname in FM116.robots:
        robot = FM116.robots[robotname]
        robot.undock()

def dock_fleet_during_lunch():
    fleet_dock()
    time.sleep(30*60) # Dock the fleet for 30 minutes
    fleet_undock()

if __name__ == "__main__":
    dock_fleet_during_lunch()
```