pyRobots Documentation

Release 2.0

Séverin Lemaignan et al.

February 17, 2017

Contents

As you may well know if you ever tried to use them to implement under-specified tasks, state machines are not the best tool to code robot controllers.

`pyRobots` provides a set of Python decorators to easily turn standard functions into background tasks which can be cancelled at anytime and to make your controller *resource aware* (no, a robot can not turn left AND right at the same time).

It also provides a event-based mechanism to monitor specific conditions and asynchronously trigger actions.

It finally provides a library of convenient tools to manage poses in a uniform way (quaternions, Euler angles and 4D matrices, I look at you) and to interface with existing middlewares (ROS, naoqi, aseba...).

CHAPTER 1

Main features

- Turns any Python function into a background *action* with the decorator `@action`.
- Robot actions are non-blocking by default: they are instanciated as futures (lightweight threads),
- Actions can be cancelled at any time via signals (the *`ActionCancelled`* signal is raised):

```
@action
def safe_walk(robot):
try:
    robot.walk()
except ActionCancelled:
    robot.go_back_to_rest_pose()

action = robot.safe_walk()
time.sleep(1)
action.cancel()
```

- Lock specific resources with a simple `@lock(...)` in front of the actions. When starting, actions will wait for resources to be available if needed:

```
L_ARM = Resource()
R_ARM = Resource()
ARMS = CompoundResource(L_ARM, R_ARM)

@action
@lock(ARMS)
def lift_box(robot):
    #...

@action
@lock(L_ARM)
def wave_hand(robot):
    #...

@action
@lock(L_ARM, wait=False)
def scratch_head(robot):
    #...

robot.lift_box()
robot.wave_hand() # waits until lift_box is over
robot.scratch_head() # skipped if lift_box or
                     # wave_hand are still running
```

- Supports *compound resources* (like `WHEELS` == `LEFTWHEEL` + `RIGHTWHEEL`)
- Create event with `robot.whenever(<condition>).do(<action>)`
- Poses are managed explicitely and can easily be transformed from one reference frame to another one (integrates with ROS TF when available).
- Extensive logging support to debug and replay experiments.

Support for a particular robot only require to subclass *GenericRobot* for this robot (and, obviously, to code the actions you want your robot to perform).

CHAPTER 2

Code Documentation

The documentation is currently sparse. Please fill bug reports everytime you can not figure out a specific bit.

Main entry points

- *robots.robot.GenericRobot*

Full package documentation

robots package

Subpackages

robots.concurrency package

robots.concurrency.action module

robots.concurrency.action.**action**(*fn*)

When applied to a function, this decorator turns it into a asynchronous task, starts it in a different thread, and returns a 'future' object that can be used to query the result/cancel it/etc.

The main methods available on these 'future' object include *RobotAction.wait()* to wait until the action completes, and *RobotAction.cancel()* to request the action to stop (ie, it raises an *ActionCancelled* signal within the action thread). See *RobotAction* for the full list of available methods.

Action implementation may want to handle the *ActionCancelled* signal to properly process cancellation requests.

Usage example:

```
    @action
    def safe_walk(robot):
    try:
        robot.walk()
    except ActionCancelled:
        robot.go_back_to_rest_pose()

    action = robot.safe_walk()
    time.sleep(1)
    action.cancel()
```

In this example, after one second, the `safe_walk` action is cancelled. This sends the signal *ActionCancelled* to the action, that can appropriately terminate.

robots.concurrency.concurrency module Concurrency support for pyRobot.

This module provides:

- an implementation of *SignalingThread* (threads that explicitely handle signals like cancelation)
- heavily modified Python futures to support robot action management.
- A future executor that simply spawn one thread per future (action) instead of a thread pool.

These objects should not be directly used. Users should instead rely on the *action()* decorator.

Helpful debugging commands:

```
>>> sys._current_frames()
>>> inspect.getouterframes(sys._current_frames()[<id>])[0][0].f_locals
```

class robots.concurrency.concurrency.**FakeFuture**(*result*)
Used in the 'immediate' mode.

result()

wait()

class robots.concurrency.concurrency.**RobotAction**(*actionname*)
Bases: concurrent.futures._base.Future

add_subaction(*action*)

cancel()

childof(*action*)
Returns true if this action is a child of the given action, ie, has been spawned from the given action or any of its descendants.

result()

set_parent(*action*)

set_thread(*thread*)

wait()
alias for result()

class robots.concurrency.concurrency.**RobotActionExecutor**

actioninfo(*future_id*)

cancel_all()
Blocks until all the currently running actions are actually stopped.

cancel_all_others()
Blocks until all the currently running actions *except the calling one* are actually stopped.

get_current_action()
Returns the RobotAction linked to the current thread.

submit(*fn*, **args*, ***kwargs*)

class robots.concurrency.concurrency.**RobotActionThread**(*future*, *initialized*, *fn*, *args*, *kwargs*)
Bases: *robots.concurrency.concurrency.SignalingThread*

run()

class robots.concurrency.concurrency.**SignalingThread**(**args*, ***kwargs*)
Bases: threading.Thread

cancel()

pause()

robots.concurrency.signals module

exception robots.concurrency.signals.**ActionCancelled**
Bases: exceptions.UserWarning

exception robots.concurrency.signals.**ActionPaused**
Bases: exceptions.UserWarning

robots.events package

robots.events.events module pyRobots' events implementation

class robots.events.events.**EventMonitor**(*robot*, *var*, *value=None*, *becomes=None*, *above=None*, *below=None*, *increase=None*, *decrease=None*, *oneshot=False*, *max_firing_freq=10*, *blocking=True*)

ABOVE = '>'

BECOMES = 'becomes'

BELOW = '<'

DECREASE = '-='

INCREASE = '+='

VALUE = '='

close()

do(*cb*)

stop_monitoring()

wait()
Blocks until an event occurs.

class robots.events.events.**Events**(*robot*)
Exposes high-level primitives to create and cancel event monitors.

robots.robot.GenericRobot creates and holds an instance of *Events()* that you can use: you should not need to instanciate yourself this class.

cancel_all()
Cancels all event monitors and interrupt running event callbacks (if any).

close()

every(*var*, *max_firing_freq=10*, *blocking=True*, ***kwargs*)
Alias for *whenever()*.

on(*var*, ***kwargs*)
Creates a new *EventMonitor* to watch a given event model (one shot).

On the first time the event is fired, the monitor is removed.

Returns a new instance of *EventMonitor* for this event.

stop_all_monitoring()
Stops all event monitoring, but do not interrupt event callbacks, if any is running.

You may want to use *stop_all_monitoring()* instead of *cancel_all()* when you need to prevent new events of being raised *from* an event callback (*cancel_all()* would interrupt this callback as well).

whenever(*var*, *max_firing_freq=10*, *blocking=True*, ***kwargs*)
Creates a new *EventMonitor* to continuously watch a given event.

`var` can either be a predicate or the name of an entry in the robot's state container (`robot.state`). In the later case, a supplementary keyword argument amongst `value=`, `become=`, `above=`, `below=`, `increase=`, `decrease=` must be provided to define the behaviour of the monitor.

Example:

```
# using the robot state:
robot.whenever("touch_sensor", value = True).do(on_touched)
robot.whenever("sonar", below = 0.4).do(on_obstacle_near)
robot.whenever("bumper", becomes = True).do(on_hit_obstacle)

# using a predicate:
def is_tired(robot):
    # do any computation you want...
    now = datetime.datetime.now()
    evening = now.replace(hour=20, minute=0, second=0, microsecond=0)
    return robot.state["speed"] > 1.0 and now > evening

robot.whenever(is_tired).do(go_to_sleep)
```

Parameters

- **var** – either a predicate (callable) or one of the key of `robot.state`.
- **max_firing_freq** – set how many times pe second this event may be triggered (default to 10Hz. 0 means as many as possible).
- **blocking** – if `True`, the event callback is blocking, preventing new event to be triggered until the callback has completed (defaults to `True`).
- **kwargs** – the monitor behaviour (cf above)

Returns a new instance of *EventMonitor* for this event.

robots.poses package

robots.poses.position module

class `robots.poses.position.`**FrameProvider**
Bases: `object`

get_transform(*frame*)
Returns the transformation between this frame and the map.

If the frame is unknown, raises UnknownFrameError.

exception `robots.poses.position.`**InvalidFrameError**
Bases: `exceptions.RuntimeError`

class robots.poses.position.**PoseManager**(*robot*)
Bases: object

A pose is for us a dict {'x':x, 'y':y, 'z':z, 'qx':qx, 'qy':qy, 'qz':qz, 'qw':qw, 'frame':frame}, ie a (x, y, z) cartesian pose in meter interpreted in a specific reference frame, and a quaternion describing the orientation of the object in radians.

This class helps with:

- converting from other convention to our convention,
- converting back to other conventions.

add_frame_provider(*provider*)

angular_distance(*angle1*, *angle2*)
Returns the (minimal, oriented) angular distance between two angles *after normalization on the unit circle*.

Angles are assumed to be radians.

The result is oriented (from angle1 to angle2) and guaranteed to be in range]-pi, pi].

distance(*pose1*, *pose2='base_link'*)
Returns the euclidian distance between two pyRobots poses.

If the second pose is omitted, "base_link" is assumed (ie, distance between a pose and the robot itself).

euler(*pose*)

get(*raw*)
takes a loosly defined 'pose' as input and returns a properly formatted and normalized pose.

Input may be:

- a frame
- an incomplete pose dictionary
- a list or tuple (x,y,z), (x,y,z,frame) or (z,y,z,rx,ry,rz) or (x,y,z,qx,qy,qz,qw)

inframe(*pose*, *frame*)
Transform a pose from one frame to another one.

Uses transformation matrices. Could be refactored to use directly quaternions.

static isin(*point*, *polygon*)
Determines if a 2D point is inside a given 2D polygon or not.

Parameters

- **point** – a (x,y) pair
- **polygon** – a list of (x,y) pairs.

Copied from: http://www.ariel.com.au/a/python-point-int-poly.html

myself()
Returns the current robot's pose, ie the pose of the ROS TF 'base_link' frame.

normalize(*pose*)

static normalize_angle(*angle*)
Returns equivalent angle such as -pi < angle <= pi

static normalizedict(*pose*)

normalizelist(*pose*)

pantilt(*pose*, *ref='/base_link'*)
Convert a xyz target to pan and tilt angles from a given viewpoint.

Parameters

- **pose** – the target pose
- **ref** – the reference frame (default to base_link)

Returns (pan, tilt) in radians, in]-pi, pi]

static quaternion_from_euler(*rx*, *ry*, *rz*)

test_angular_distance()
Small regression test for the computation of angular distances

exception robots.poses.position.**UnknownFrameError**
Bases: exceptions.RuntimeError

robots.poses.ros_positions module

class robots.poses.ros_positions.**ROSFrames**
Bases: *robots.poses.position.FrameProvider*

asROSpose(*pose*)
Returns a ROS PoseStamped from a pyRobots pose.

Parameters **pose** – a standard pyRobots pose (SPARK id, TF frame, [x,y,z], [x,y,z,rx,ry,rz], [x,y,z,qx,qy,qw,qz], {'x':..., 'y':...,...})

Returns the corresponding ROS PoseStamped

get_transform(*frame*)

inframe(*pose*, *frame*)
Transforms a given pose in the given frame.

publish_transform(*name*, *pose*)
Publishes a new TF frame called 'name' based on the pyRobots transform 'pose'.

Note that this function *does not* normalize the input pose, which must already be a dictionary with the keys [x,y,z,qx,qy,qz,qw,frame].

robots.resources package

robots.resources.lock module

robots.resources.lock.**lock**(*res*, *wait=True*)
Used to define which resources are acquired (and locked) by the action.

This decorator may be used as many times as required on the same function to lock several resources.

Usage example:

```
L_ARM = Resource()
R_ARM = Resource()
ARMS = CompoundResource(L_ARM, R_ARM)

@action
@lock(ARMS)
def lift_box(robot):
    #...
```

```
@action
@lock(L_ARM)
def wave_hand(robot):
    #...

@action
@lock(L_ARM, wait=False)
def scratch_head(robot):
    #...

robot.lift_box()
robot.wave_hand() # waits until lift_box is over
robot.scratch_head() # skipped if lift_box or
                     # wave_hand are still running
```

Parameters

- **res** – an instance of Resource or CompoundResource
- **wait** – (default: true) if `true`, the action will wait until the resource is available, if `false`, the action is skipped if the resource is not available.

robots.resources.resources module

class `robots.resources.resources.`**CompoundResource**(**args*, ***kwargs*)

acquire(*wait=True*, *acquirer='unknown'*)

release()

class `robots.resources.resources.`**Resource**(*name=''*)

acquire(*wait=True*, *acquirer='unknown'*)

release()

robots.mw package

robots.helpers package

robots.helpers.ansistrm module An ANSI-based colored console log handler, based on https://gist.github.com/758430, and with a few special features to make sure it works well in pyRobots' concurrent environment.

class `robots.helpers.ansistrm.`**ConcurrentColorizingStreamHandler**(*scheme=None*)

Bases: `logging.StreamHandler`

A log handler that:

- (tries to) guarantee strong thread-safety: the threads generating log message can be interrupted at *any* time without causing dead-locks (which is not the case with a regular `StreamHandler`: the calling thread may be interrupted while it owns a lock on stdout)
- propagate pyRobots signals (ActionCancelled, ActionPaused)
- colors the output (nice!)

bright_scheme = {40: (None, 'red', False, False), 10: (None, 'blue', False, False), 20: (None, 'white', False, False), 50: (

color_map = {'blue': 4, 'black': 0, 'yellow': 3, 'cyan': 6, 'green': 2, 'magenta': 5, 'white': 7, 'red': 1}

colorize(*message*, *record*)

csi = '\x1b['

dark_scheme = {40: (None, 'red', False, False), 10: (None, 'blue', False, False), 20: (None, 'black', False, False), 50: ('re

emit(*record*)

format(*record*)

handle(*record*)

Override the default handle method to *remove locking*, because Python logging, while thread-safe according to the doc, does not play well with us raising signals (ie exception) at anytime (including while the logging system is locking the output stream).

is_tty

mono_scheme = {40: (None, None, False, False), 10: (None, None, False, False), 20: (None, None, False, False), 50: (None

output_colorized(*message*)

reset = '\x1b[0m'

run()

xmas_scheme = {40: ('red', 'yellow', False, True), 10: ('red', 'yellow', False, True), 20: ('red', 'white', False, True), 50: (

robots.helpers.ansistrm.**main**()

robots.helpers.misc module

robots.helpers.misc.**enable_logger_print**()

robots.helpers.misc.**enum**(**sequential*, ***named*)

class robots.helpers.misc.**valuefilter**(*maxlen=10*)

MAX_LENGTH = 10

append(*val*)

get()

robots.robot module

class robots.robot.**GenericRobot**(*actions=None*, *supports=0*, *dummy=False*, *immediate=False*, *configure_logging=True*)

Bases: object

This class manages functionalities that are shared across every robot 'backends' (ROS, Aseba,...)

You are expected to derive your own robot implementation from this class, and it is advised to use instances of *GenericRobot* within a context manager (ie `with MyRobot as robot: ...` construct).

Its role comprises of:

- automatic addition of proxy methods for the robot actions
- actions execution (spawning threads for actions via `self.executor`
- pose management through the `robot.poses` instance variable
- event monitoring through the `robot.on(...).do(...)` interface

GenericRobot defines several important instance variables, documented below.

Variables

- **state** – the state vector of the robot. By default, a simple dictionary. You can overwrite it with a custom object, but it is expected to provide a dictionary-like interface.
- **poses** – an instance of *PoseManager*.
- **executor** – instance of *RobotActionExecutor* responsible for spawning and starting threads for the robot actions. You should not need to access it directly.

Example of a custom robot:

```
from robots import GenericRobot

class MyRobot(GenericRobot):

    def __init__(self):
        super(MyRobot, self).__init__()

        # create (and set) one element in the robot's state. Here a bumper.
        # (by default, self.state is a dictionary. You can safely
        # overwrite it with any dict-like object.
        self.state["my_bumper"] = False

        # do whatever other initialization you need for your robot

    # Implement here all the accessors you need to talk to the robot
    # low-level, like:

    def send_goal(self, pose):
        # move your robot using your favorite middleware
        print("Starting to move towards %s" % pose)

    def stop(self):
        # stop your robot using your favorite middleware
        print("Motion stopped")

    def whatever_other_lowlevel_method_you_need(self):
        #...
        pass

# create actions
@action
def move_forward(robot):
    #...
    pass

with MyRobot() as robot:

    # Turn on DEBUG logging.
    # Shortcut for logging.getLogger("robots").setLevel(logging.DEBUG)
    robot.debug()

    # subscribe to events...
    robot.whenever("my_bumper", value = True).do(move_forward)

    try:
        while True:
```

```
            time.sleep(0.5)
    except KeyboardInterrupt:
        pass
```

Note: A note on debugging

Several methods are there to help with debugging:

- *loglevel()*: default to INFO. logging.DEBUG can be useful.
  - *silent()*: alias for loglevel(logging.WARNING)
  - *info()*: alias for loglevel(logging.INFO)
  - *debug()*: alias for loglevel(logging.DEBUG)
- *running()*: prints the list of running tasks (with their IDs)
- *actioninfo()*: give details on a given action, including the exact line being currently executed

Parameters

- **actions** (*list*) – a list of packages that contains modules with actions (ie, modules with functions decorated with @action). Proxies to these actions are appended to the instance of GenericRobot upon construction.
- **supports** – (default: 0) a mask of middlewares the robot supports. Supported middlewares are listed in robots.mw.__init__.py. For example supports = ROS|POCOLIBS means that both ROS and Pocolibs are supported. This requires the corresponding Python bindings to be available.
- **dummy** (*boolean*) – if True (defults to False), the robot is in 'dummy' mode: no actual actions are performed. The exact meaning of 'dummy' is left to the subclasses of GenericRobot.
- **immediate** (*boolean*) – if True (defaults to False), actions are executed in the main thread instead of their own separate threads. Useful for some specific debugging scenarios.
- **configure_logging** (*boolean*) – if True (default), configures a default colorized console logging handler.

actioninfo(*id*)
Print details on a running action (including the current line number).

cancel_all()
Sends a 'cancel' signal (ie, the *ActionCancelled* exception is raised) to all running actions.

Note that, if called within a running action, this action *is cancelled as well*. If this is not what you want, use *cancel_all_others()* instead.

Actions that are not yet started (eg, actions waiting on a resource availability) are simply removed for the run queue.

cancel_all_others()
Sends a 'cancel' signal (ie, the *ActionCancelled* exception is raised) to all running actions, *except for the action that call :meth:'cancel_all_others'* (note that its currently running subactions *will be cancelled*).

Actions that are not yet started (eg, actions waiting on a resource availability) are simply removed for the run queue.

close()

static **configure_console_logging**()

debug()

filtered(*name*, *val*)
Helper to easily filter values (uses an accumulator to average a given 'name' quantity)

info()

load_actions(*actions*)

loglevel(*level=20*)

running()
Print the list of running actions.

silent()

static **sleep**(*duration*)
Active sleep. Must used by actions to make sure they can be quickly cancelled.

supports(*middleware*)

wait(*var*, ***kwargs*)
Alias to wait on a given condition. Cf `robots.events.Events` for details on the acceptable conditions.

wait_for_state_update(*timeout=None*)
Blocks until the robot state has been updated.

This is highly dependent on the low-level mechanisms of your robot, and should almost certainly be overriden in your implementation of a GenericRobot subclass.

The default implementation simply waits `ACTIVE_SLEEP_RESOLUTION` seconds.

class `robots.robot.`**State**
Bases: `dict`

robots.introspection module

robots.roslogger module

class `robots.roslogger.`**RXConsoleHandler**(*topic='/rosout'*)
Bases: `logging.Handler`

emit(*record*)

CHAPTER 3

Minimum Working Example

...that includes the creation of a specific robot

```
import time
from robots import GenericRobot
from robots.decorators import action, lock
from robots.resources import Resource
from robots.signals import ActionCancelled

# create a 'lockable' resource for our robot
WHEELS = Resource("wheels")

class MyRobot(GenericRobot):

    def __init__(self):
        super(MyRobot, self).__init__()

        # create (and set) one element in the robot's state. Here a bumper.
        self.state.my_bumper = False

        # do whatever other initialization you need :-)

    def send_goal(self, pose):
        # move your robot using your favorite middleware
        print("Starting to move towards %s" % pose)

    def stop(self):
        # stop your robot using your favorite middleware
        print("Motion stopped")

    def whatever_lowlevel_method_you_need(self):
        pass

@lock(WHEELS)
@action
def move_forward(robot):
    """ We write action in a simple imperative, blocking way.
    """

    # the target pose: simply x += 1.0m in the robot's frame. pyRobots
    # will handle the frames transformations as needed.
    target = [1.0, 0., 0., "base_link"]

    try:
```

```
        robot.send_goal(target)

        while(robot.pose.distance(robot.pose.myself(), target) > 0.1):
            # robot.sleep is exactly like time.sleep, except it lets the pyrobots
            # signals pass through.
            robot.sleep(0.5)

        print("Motion succeeded")

    except ActionCancelled:
        # if the action is cancelled, clean up your state
        robot.stop()


with MyRobot() as robot:

    # Turn on DEBUG logging.
    # Shortcut for logging.getLogger("robots").setLevel(logging.DEBUG)
    robot.debug()

    robot.whenever("my_bumper", value = True).do(move_forward)

    try:
        while True:
            time.sleep(0.5)
    except KeyboardInterrupt:
        pass
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

r

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X