INFORME IMPLEMENTACIÓN CONTROL DEL ROBOT UR3E EN PYTHON.

FECHA: 15/07/2023

ESCRITO POR: MIGUEL MONTES LORENZO

1. CONTENIDO DEL DOCUMENTO

Este documento contiene una explicación del trabajo que he realizado durante el mes de julio de 2023 terminando los programas Python necesarios para el control del robot UR3E para el proyecto de investigación de Lucia Güitta sobre entrenamiento de brazos robóticos en un entorno mixto (virtual + real). El objetivo de este documento es que cualquier persona que necesite entender y/o modificar alguno de los scripts empleados pueda conocer su utilidad y su funcionamiento.

2. MI ROL EN EL PROYECTO

Soy la tercera persona que trabaja en el control del brazo robótico UR3E. Previamente han trabajado en el también Lorenzo Rodríguez Pérez y Félix García Fernández en ese orden. A continuación adjunto la dirección de los repositorios de cada una de las personas que hemos colaborado en el proyecto.

Lorenzo Rodríguez Pérez:

https://github.com/lorenzo-rod/ciclab_cumillas.git

Félix García Fernández:

https://github.com/Felix-Garci/UR3e_com.git

Miguel Montes Lorenzo:

https://github.com/MiguelMontesLorenzo/UR3E_software.git

Al incorporarme Lucía me explicó que las personas que habían trabajado previamente en el proyecto, y en especial Félix, habían terminado la programación de un script de Python con el que abrir una comunicación y controlar los movimientos del brazo robótico. Me pidió que me encargara de, en primer lugar, revisar la eficiencia y la estructura de dicho programa, y en segundo lugar, construir una clase de control equivalente a la que ella estaba empleando para el control del brazo robótico industrial IRB 120 de ABB, de forma que se pudiera manejar utilizando la misma interfaz.

Una cosa que es importante mencionar es que, como se puede ver tanto en el programa de control en Python como en los informes de los otros colaboradores, esta parte del proyecto en

un inicio contemplaba 2 posibles tipos de control del brazo, el control dada la posición angular de las articulaciones y el control dada la posición cardinal del extremo del brazo. Sin embargo, al discutirlo con Lucía, me comentó que ella solo estaba empleando el control dada la posición angular y que me centrara en cerciorarme de que éste funciona correctamente. Es por esto por lo que, aunque en el código haya fragmentos que hacen referencia al segundo tipo de control, este no está testeado. Además, tal y como se menciona en el informe de Lorenzo, su funcionamiento requiere de la ejecución de un programa distinto en el "polyscope".

3. INTRODUCCIÓN

En primer lugar, quiero aclarar que debido a que soy la tercera persona trabajando en esta parte del proyecto, he estado trabajando con un cierto nivel de abstracción, en especial sobre la parte de Lorenzo. Por tanto, hay algunas decisiones sobre el diseño y la estructura que no he tomado yo, y que, si se quieren entender en más profundidad, deberían revisarse en su documentación. Consecuente, en este documento habrá apartados concretos sobre cuyo funcionamiento no entraré en detalle a bajo nivel y simplemente me limitaré a explicar como los he entendido yo al trabajar con ellos.

El UR3E es un brazo robótico de 6 articulaciones que cuenta con un controlador propio (la caja gris que hay debajo de la mesa). Además, el controlador cuenta con a una especie de monitor Tablet llamado "**polyscope**", que permite, encender y apagar el brazo, desarrollar y ejecutar programas (utilizando un lenguaje propio de la empresa Universal Robots) para ejecutar movimientos y modificar registros, activar el movimiento libre (permite mover las articulaciones a mano), detener el brazo usando el botón de emergencia, etc

El UR3E está conectado por ethernet (no sé en profundidad como ha sido configurada esta conexión) al ordenador de escritorio con sistema operativo Linux (Ubuntu) ubicado en el Laboratorio. Por tanto todos los programas de Python que se mencionen en este documento deben ejecutarse en dicho equipo, ya que de lo contrario no podrá establecerse la conexión con el brazo robótico.

La comunicación entre el robot y Python se realiza a través de la modificación de una serie de registros en ambas direcciones. Es decir, el controlador del brazo envía información al equipo modificando registros "input", y el equipo envía información al controlador modificando registros "output". De esta forma, para mover el brazo, es necesario emplear un programa en el "polyscope" que lea los registros "input" que contienen la información sobre las posiciones y las velocidades a las que debe mover cada articulación, y que, a continuación, llame a las sentencias necesarias para llevar a cabo el movimiento con dichos valores. Recíprocamente, el programa de control en Python debe leer en los registros "output" con las nuevas posiciones registradas por el controlador del brazo en cada momento, realizar los cálculos pertinentes, y escribir de nuevo en los registros "input" la información necesaria para que el brazo pueda llevar a cabo su siguiente movimiento.

4. MANEJO DEL POLYSCOPE

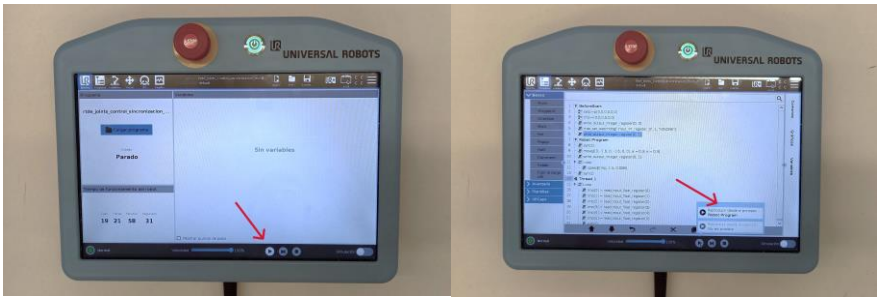
En este apartado trataré por encima el cómo utilizar el “polyscope” para cada una de las tareas que he mencionado previamente. En realidad, el “polyscope” tiene muchas más opciones de las que voy a mencionar a continuación (y que yo no conozco en profundidad), pero estas son las que yo he encontrado necesarias para hacer todas las pruebas pertinentes durante el desarrollo de los programas de Python:

En primer lugar, para encender el “polyscope” basta con pulsar el botón de encendido.

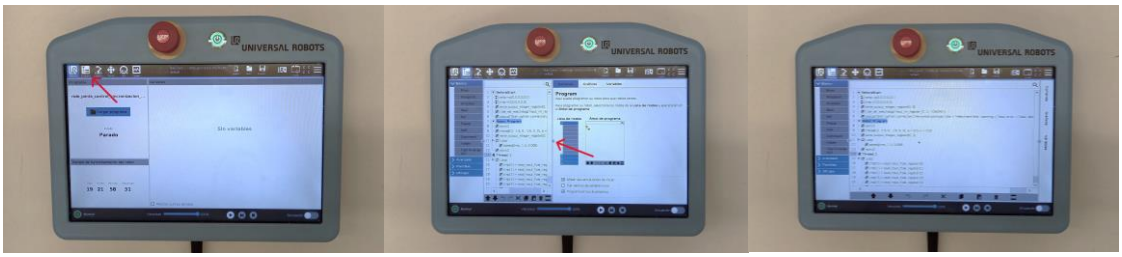
Una vez encendido el “polyscope” se abrirá la siguiente pestaña, que ofrece una opción para encender el brazo, si la seleccionamos, el controlador comprobará que el brazo está en su posición, quitará los seguros, y completará su inicialización.

Para cargar un programa se seleccionará la opción de “cargar un programa” la opción de “abrir” situada en la barra superior, lo que abrirá a un explorador de archivos que permitirá, abrir, eliminar, copiar y pegar programas y directorios.

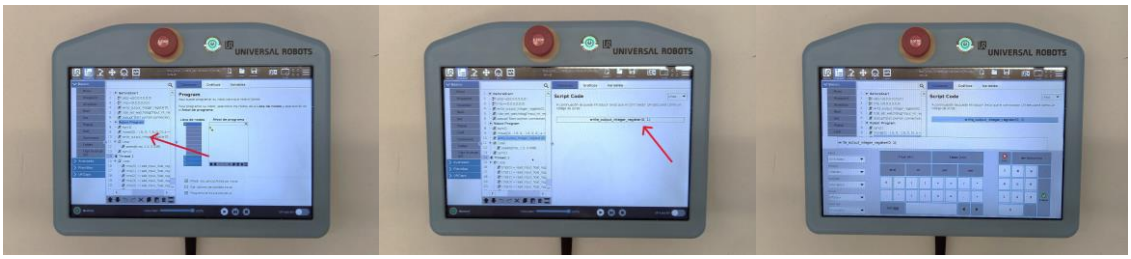
Una vez abierto un programa, para ejecutarlo basta con seleccionar la opción de “play” que aparece abajo a la derecha.



Una vez abierto un programa, para ver el código que lo compone se selecciona la pestaña de “programa” que aparece arriba a la izquierda.



Una vez en la pestaña de “programa” para editar una línea de código basta con pulsar sobre ella, lo que hará que aparezca en el lado derecho de la pantalla donde se puede editar.

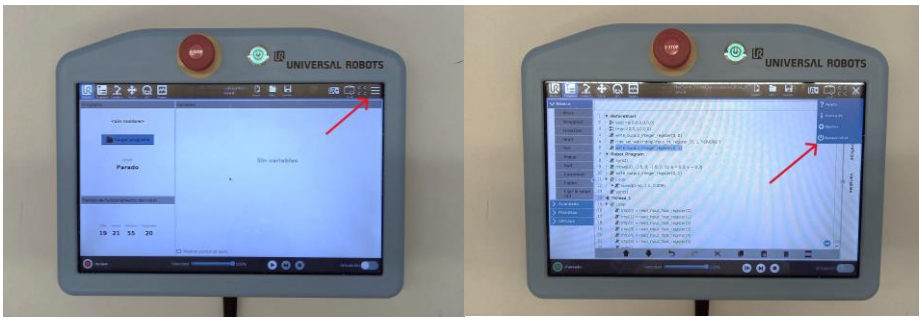


Durante la ejecución de un programa que mueve el brazo, para activar el bloqueo de emergencia (que bloquea el movimiento del brazo al instante) basta con pulsar el botón rojo situado en el margen superior del “polyscope”. Es importante estar siempre atento al brazo durante una ejecución de dudoso resultado para evitar que el brazo golpee algún objeto o a la mesa.

Para reactivar el brazo tras el bloqueo de emergencia se deberá tirar del botón rojo hacia arriba y volver a realizar el proceso de inicialización.

Una vez inicializado el brazo, para habilitar el movimiento libre (es decir, manipular su posición con las manos sin encontrar resistencia) basta con pulsar el botón gris situado en la parte superior trasera del “polyscope”.

Para apagar el “polyscope” (y consecuentemente desactivar el brazo) bastará con seleccionar la opción de “apagar” en el menú desplegable arriba a la derecha.



5. PROGRAMA DEL POLYSCOPE A EJECUTAR

La versión final del programa que debe ejecutarse en el "polyscope" es una versión muy similar a la descrita por Lorenzo en su informe para el control por posición angular. En este punto es importante mencionar que la programación en el "polyscope" permite usar funciones para efectuar movimientos (indicando una posición angular de cada articulación y dejando que el propio controlador del brazo se encargue de moverlo a esa posición gestionando la velocidad y la aceleración en cada momento) y funciones para establecer velocidades (que permiten al propio usuario gestionar la velocidad de cada articulación en cada instante de tiempo).

Pese a que a priori el uso de funciones de ejecución de movimientos puede parecer más recomendable (ya que no obliga al usuario a mantener un hilo de ejecución paralela monitorizando en todo momento la velocidad angular en cada articulación) Lorenzo tomó la decisión de llevar a cabo el control utilizando las funciones de velocidad. Estoy seguro de que él, que es el que ha estudiado con más profundidad la construcción de programas en el "polyscope", encontró sus motivos para tomar esta decisión, y por tanto he mantenido el uso de las funciones de velocidad.

En el momento en el que yo he dejado el proyecto la ruta en el "polyscope" en la que se encuentra el programa es la siguiente:

home/control_rtde_MM/rtde_joints_control_sincronization_modificada.urp

Sin embargo, por si acaso en el futuro alguien modificara la estructura de archivos, dejo adjunta una imagen del código del programa:

```
BeforeStart:
tmp := [0,0,0,0,0,0]
write_output_integer_register(0,0)
rtde_set_watchdog("input_int_register_0", 1, "IGNORE")
popup("Start python connection, then press continue", title="Hello there fella", warning=False, error=False, blocking=True)

RobotProgram:
movej([0,-1.6,0,-1.6,0,0])
write_output_integer_register(0,1)
sync()
Loop:
    speed;(tmp, 1.5, 0.008)
sync()

Thread_1:
Loop:
    tmp[0] = read_input_float_register(0)
    tmp[1] = read_input_float_register(1)
    tmp[2] = read_input_float_register(2)
    tmp[3] = read_input_float_register(3)
    tmp[4] = read_input_float_register(4)
    tmp[5] = read_input_float_register(5)
    sync()
```

En primer lugar el código está estructurado en 3 hilos distintos, el "BeforeStart", el "RobotProgram", y el "Thread_1". El "BeforeStart" es siempre el primero en ejecutarse. El "RobotProgram" y el "Thread_1" se ejecutan en paralelo tras la finalización del "BeforeStart".

A continuación procedo a explicar cual es la utilidad de cada sentencia del programa:

BeforeStart:

$tmp := [0,0,0,0,0,0]$ $\#$ *inicialización de un vector de dimensión* 6 *del propio programa a* $0's$

$write\_output\_integer\_register(0,0)$

$\#\ asignar\ un\ valor\ de\ 0\ al\ primero\ de\ los\ registros\ output\ de\ tipo\ entero$

$rtde\_set\_watchdog("input\_int\_register\_0", 1, "IGNORE")$

$\#\ inicializa\ el\ watchdog\ empleando\ el\ registro\ indicado$

$popup(\dots)$

$\#\ Lanza\ un\ mensaje\ en\ el\ polyscope\ que\ detiene\ la\ ejecución\ del\ programa\ hasta\ una\ confirmación\ del\ usuario$

RobotProgram:

$movej([0, -1.6, 0, -1.6, 0, 0, 0])$

$\#\ mueve\ cada\ articulación\ a\ las\ posiciones\ angulares\ indicadas\ (pone\ el\ brazo\ en\ vertical)$

$write\_output\_integer\_register(0,0)\quad \#\ asignar\ un\ valor\ de\ 0\ al\ primero\ de\ los\ registros\ output\ de\ tipo\ entero$

$sync()\quad \#\ función\ que\ soncroniza\ los\ hilos\ que\ se\ ejecutan\ en\ paralelo$

$Loop:\quad \#\ inicializa\ un\ bucle\ infinito$

$\quad speedj(tmp, 1.5, 0{,}008)\quad \#\ establece\ en\ cada\ articulación\ (i)\ una\ velocidad\ de\ tmp[i]$

$sync()$

Thread_1:

$Loop:\quad \#\ inicializa\ un\ bucle\ infinito$

$\quad tmp[i]\ =\ read\_input\_float\_register(i)$

$\quad \#\ establece\ en\ la\ posición\ i-\text{ésima}\ del\ vector\ tmp\ el\ valor\ del\ i-\text{ésimo}\ registro\ de\ coma\ flotante$

$\quad sync()$

El funcionamiento del programa por tanto es el siguiente:

En el hilo **“BeforeStart”**:

Primero se crea un vector “tmp” que va a servir para almacenar la velocidad de cada una de las 6 articulaciones en cada momento.

Se establece el valor del primer registro output de tipo entero a 0. Este registro es el que se emplea para comunicarle al hilo del programa de Python que se encarga de la monitorización del brazo cuando tiene el control sobre este. Un valor de 0 en este registro indica que el que tiene el control es el controlador del UR3E.

Se establece el “watchdog” en el primer registro input de tipo entero. No me he parado a buscar con total seguridad que es el “watchdog”, pero por lo que he podido entender creo que es un registro que se emplea para indicarle al controlador del brazo que el valor del resto de registros ha sido actualizado. Creo que el argumento con valor 1 indica que la frecuencia de revisión del

“watchdog” es de 1 Hz. En todo caso, si en algún momento se necesita manipular esta parte del programa debería revisarse esto con más profundidad.

Se lanza un “popup” (un pequeño mensaje que detiene la ejecución del programa hasta la confirmación del usuario) en la pantalla del “polyscope” indicando que se ha alcanzado el punto del programa en el que se debe iniciar el comunicación desde la clase desarrollada en Python (la elección de este punto está relacionada con la estructura del programa desarrollado en Python como se verá más adelante).

En el hilo **“RobotProgram”**:

Primero se mueve el brazo a la posición inicial indicada. En concreto, se pone el brazo en vertical

Se establece el valor del primer registro output de tipo entero a 1. De esta forma, se le indica al programa de Python que el brazo ya está listo para recibir órdenes.

Se llama a la función “sync” para llevar a cabo la sincronización entre hilos. No estoy seguro si la función “sync” es una función nativa del propio lenguaje de Universal Robots o una función desarrollada por algún otro usuario del “polyscope” previamente (ya en la documentación de Universal Robots no he encontrado ninguna referencia a ella). La forma en la que creo que funciona es bloqueando la ejecución de cualquier hilo que llegue al punto donde esta se ubica hasta el momento en el que todos los hilos con una función “sync” en su interior lleguen a ella.

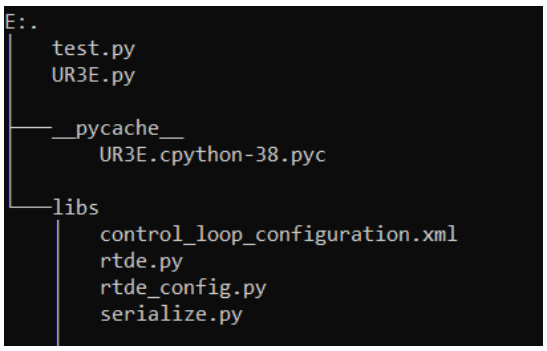
Se lanza un bucle infinito durante el cual controlador está de forma constante asignando a cada una de las articulaciones del brazo el valor de velocidad presente en la posición correspondiente del vector “tmp”.

En el hilo **“Thread_1”**:

Se lanza un bucle infinito que se encarga en cada iteración de actualizar el valor de la posición i-ésima del vector “tmp” de acuerdo al correspondiente registro input de tipo coma flotante.

6. ESTRUCTURA DE ARCHIVOS DE LOS SCRIPTS DE PYTHON

La estructura de archivos que se puede encontrar en mi repositorio del proyecto tiene la siguiente forma:



El fichero **UR3E.py** es el archivo que contiene la clase principal “RobotControl” (clase de más alto nivel que admite directamente las instrucciones para mover el UR3E a una posición dada) y la clase auxiliar

“UR3EConnection” que es la que se encarga de la gestión del hilo paralelo de monitorización y de la comunicación con el controlador a través de los módulos de Universal Robots.

El fichero **test.py** es una fichero que contiene un código de pruebas desde el que se instancia la clase “RobotControl” y se ejecutan algunos movimientos y peticiones de información del brazo.

Dentro del directorio **libs** se encuentran los módulos desarrollados por Universal Robots

Los ficheros **rtde.py** y **serialize.py** incorporan directamente todo el proceso de comunicación con el controlador del brazo mediante la librería de “sockets”, evitando que el usuario tenga que implementarlo desde cero.

El fichero **control_loop_configurantion.xml** define todos los registros tanto de “input” como de “output” que se van a emplear para el proceso de comunicación entre el equipo y el controlador del UR3E. Este el único fichero por el que el usuario debería modificar en caso de querer modificar la metodología de la comunicación.

El fichero **rtde_config.py** se encarga de llevar a cabo la lectura de la información relativa a los registros presente en el fichero de control_loop_configuration.xml mediante la librería “xml.etree.ElementTree”.

7. CONFIGURACIÓN DE LOS REGISTROS

Como ya se menciona previamente, toda la comunicación entre el equipo que instancia la clase definida en Python y el controlador del UR3E se lleva a cabo mediante tal la modificación de una serie de registros en ambas direcciones. El nombre y el tipo de estos registros están definidos en el fichero **control_loop_configurantion.xml** que se muestra a continuación:

```
<?xml version="1.0"?>
<rtde_config>
    <recipe key="state"> <!--allows to define variables that are sent from the robot to the computer -->
        <field name="runtime_state" type="UINT32"/>
        <field name="actual_q" type="VECTOR6D"/>
        <field name="actual_qd" type="VECTOR6D"/>
        <field name="actual_TCP_speed" type="VECTOR6D"/>
        <field name="actual_TCP_pose" type="VECTOR6D"/>
        <field name="output_bit_registers0_to_31" type="UINT32"/>
        <field name="output_int_register_0" type="INT32"/>
        <field name="output_double_register_0" type="DOUBLE"/>
        <field name="output_double_register_1" type="DOUBLE"/>
        <field name="output_double_register_2" type="DOUBLE"/>
        <field name="output_double_register_3" type="DOUBLE"/>
        <field name="output_double_register_4" type="DOUBLE"/>
        <field name="output_double_register_5" type="DOUBLE"/>
    </recipe>

    <recipe key="setp"> <!--allows you to define variables that are sent from the computer to the robot-->
        <field name="input_double_register_0" type="DOUBLE"/>
        <field name="input_double_register_1" type="DOUBLE"/>
        <field name="input_double_register_2" type="DOUBLE"/>
        <field name="input_double_register_3" type="DOUBLE"/>
        <field name="input_double_register_4" type="DOUBLE"/>
        <field name="input_double_register_5" type="DOUBLE"/>
        <field name="input_double_register_6" type="DOUBLE"/>
        <field name="input_double_register_7" type="DOUBLE"/>   <!-- prueba -->
        <field name="input_double_register_8" type="DOUBLE"/>   <!-- prueba -->
        <field name="input_double_register_9" type="DOUBLE"/>   <!-- prueba -->
        <field name="input_double_register_10" type="DOUBLE"/>  <!-- prueba -->
        <field name="input_double_register_11" type="DOUBLE"/>  <!-- prueba -->
    </recipe>

    <recipe key="watchdog">
        <field name="input_int_register_0" type="INT32"/>
    </recipe>
</rtde_config>
```

Los registros definidos dentro de la etiqueta “recipe” con la clave “**state**” son los que permiten al equipo conocer el estado en el que se encuentra el UR3E en cada momento. Creo que de los registros definidos dentro de esta etiqueta los 5 últimos estaban destinados a la monitorización por control de la posición terminal del brazo, por lo que en la práctica no se usan. Los registros más importantes dentro del apartado de estado son los siguientes:

- **actual_q**: es un vector de 6 dimensiones que contiene en todo momento las posiciones angulares de cada una de las articulaciones del brazo.
- **actual_TCP_pose**: es un vector de 6 dimensiones donde las 3 primeras posiciones describen en todo momento las coordenadas espaciales donde se ubica el extremo del brazo tomando como punto de referencia en la base del mismo.
- **output_int_register_0**: es un registro que indica cuando es el controlador del brazo el que tiene el control sobre la monitorización del UR3E (cuando el registro toma el valor 0) y cuando lo tiene el equipo (cuando el registro toma el valor 1).

Los registros definidos dentro de la etiqueta “recipe” con la clave “**setp**” son los que permiten al equipo trasmitir la información necesaria para controlar el movimiento del UR3E a través del programa ejecutado en el “polyscope”. En este caso los registros del 7 al 11 también estaban destinados a la monitorización por control de la posición terminal del brazo, por lo que en la práctica no se usan. Los 7 registros restantes tienen la función de:

- **Los registros del 0 al 5**: trasmitir la velocidad angular que cada una de las articulaciones del UR3E tiene que adoptar en cada momento.
- **El registro 6**: trasmitir el modo de ejecución (se explicarán los valores que este puede tomar más adelante). En realidad, el programa ejecutado en el “polyscope” no utiliza para nada esta información, ya que el modo de ejecución es siempre el de control por velocidad angular en cada articulación.

El registro definido dentro de la etiqueta “recipe” con la clave “**watchdog**” es el que aloja, tal y como su propio nombre indica, el registro del “watchdog”.

8. ESTRUCTURA DE LA CLASE: UR3EConnection

La clase **UR3Connection** es la primera de las 2 clases definidas en el fichero **UR3E.py**. Para entender las tareas de las que se hace cargo esta clase lo primero es conocer su estructura:

El método constructor inicializa las siguientes variables:

self.t: variable destinada a almacenar la referencia al hilo de ejecución paralela que monitoriza el control del UR3E.

self.MODE: variable destinada a almacenar en todo momento el modo de ejecución del UR3E. Los posibles estados que puede tomar son:

- **-1** : para dar la orden de detener la ejecución del hilo de monitorización y por tanto de detener el brazo.
- **0**: estado en el que el hilo de monitorización tiene el control sobre el brazo pero no se está ejecutando ningún movimiento.
- **1**: estado en el que el hilo está monitorizando un movimiento dado por el método de control de por posición angular final de las articulaciones.
- **2**: estado en el que el hilo está monitorizando un movimiento dado por el método de control de movimiento por posición cartesiana final de las articulaciones.

En la práctica, lo que controla el valor de esta variable es la sección del bucle principal del hilo de monitorización que se va a estar ejecutando.

self.ROBOT_HOST, **self.ROBOT_PORT**: variables que contienen la información sobre la ip y el puerto necesaria para llevar a cabo la conexión.

self.config_filename: variable que contiene la ruta donde se ubica el fichero de configuración de los registros.

self.con: variable destinada a almacenar la referencia al objeto de la conexión abierta a través de la clase RTDE (clase principal definida en el módulo rtde.py), que ya incorpora todo el proceso de comunicación a través de la librería "soket".

self.gain: variable que almacena un coeficiente para controlar la magnitud de la velocidad angular de las articulaciones.

self.setp: objeto en el que se almacenan los valores a ser establecidos en cada uno de los registros en la próxima actualización de los mismos.

self.watchdog: objeto que se encarga de asegurar que la comunicación entre el equipo y el controlador del UR3E sigue siendo segura. Un "watchdog" en un contexto de programación es un temporizador que cada vez que alcanza el 0.

self.angles: variable que contiene el estado presente de las posiciones angulares en las que se encuentra cada una de las articulaciones del robot.

self.target_angles: variable que contiene las posiciones objetivo de cada una de las articulaciones del robot.

self.check: variable binaria destinada a definir cuando el error entre las posiciones del UR3E y las posiciones objetivo es lo suficientemente pequeño como para detener el movimiento.

self.error: variable que contiene la tolerancia con la que se puede cambiar el estado de "self.check". Actualmente está establecido en un 1% del movimiento en curso.

Dadas las anteriores variables internas, la estructura de la clase viene condicionada por su funcionamiento, que es el siguiente:

Para empezar, se inicializa la comunicación a través de una llamada al método **connect2robot**, que lleva a cabo la ejecución de las siguientes operaciones:

- Lectura de la información contenida en el fichero **control_loop_configurantion.xml** a través del módulo **rtde_config.py**. En concreto obtiene el nombre y el tipo de las variables que se van a estar compartiendo mediante la modificación de los registros definidos en el fichero anterior.

```
conf = rtde_config.ConfigFile(self.config_filename)
state_names, state_types = conf.get_recipe('state')
setp_names, setp_types = conf.get_recipe('setp')
watchdog_names, watchdog_types = conf.get_recipe('watchdog')
```

- Se inicia la conexión con el UR3E a través del módulo rtde.py (que utiliza la librería de sockets para ello).

```
# connection to the robot
self.con = rtde.RTDE(self.ROBOT_HOST, self.ROBOT_PORT)
self.con.connect()
connection_state = self.con.connect() # returns 0 when connection is active

# check connection
while connection_state != 0:
    time.sleep(5)
    connection_state = self.con.connect()
```

- Se definen todos los registros que se van a emplear durante el proceso de comunicación. Es decir, los registros de "input" (del equipo al controlador), los de "output" (del controlador al equipo), y el "watchdog" (que se encarga de monitorizar la estabilidad de la comunicación).

```
self.con.send_output_setup(state_names, state_types)
self.setp = self.con.send_input_setup(setp_names, setp_types)
self.watchdog = self.con.send_input_setup(watchdog_names, watchdog_types)

# Initialize 6 registers which will hold the target angle values
self.setp.input_double_register_0 = 0.0
self.setp.input_double_register_1 = 0.0
self.setp.input_double_register_2 = 0.0
self.setp.input_double_register_3 = 0.0
self.setp.input_double_register_4 = 0.0
self.setp.input_double_register_5 = 0.0
self.setp.input_double_register_6 = 0.0
    ....
# function "rtde_set_watchdog" in "rtde_control_loop.urp" creates 1 Hz watchdog
self.watchdog.input_int_register_0 = 0
print("connected")
```

- Se inicializa el hilo de monitorización de velocidad de las articulaciones del robot.

```
self.t.start()
    return 0
```

Una vez termina la ejecución del método **connect2robot**, ya hay un hilo paralelo encargándose de los cálculos de velocidades y de la lectura o modificación de registros para la comunicación, por lo que el hilo principal queda libre para atender a peticiones de, modificación de la posición objetivo o devolución de información sobre el estado del UR3E.

Cada vez que el usuario (o una clase intermediaria) desee mover el robot a una nueva posición lo hará a través de una llamada al método **go_ang**, que se encargará de poner la variable "self.check" a 0, modificar la posición objetivo, y establecer el modo de ejecución a 1. La adquisición y liberación del semáforo **mutex** se lleva a cabo para evitar que halla más de un método desbloqueando la ejecución del bucle principal del hilo de monitorización. El "release" del semáforo **mode_bloq** desbloquea la del bucle principal (que en ese momento estará bloqueado en la sección de modo de ejecución 0. El "acquire" del semáforo **finished_movement_confirmation** bloquea la finalización de este método hasta el momento en el que el

UR3E alcance la posición objetivo (véase el código de la sección de ejecución en modo 0 en el bucle principal el hilo de monitorización para entender esto completamente).

```
def go_ang(self, angles):
    self.check = 0
    mutex.acquire()
    self.target_ang = angles
    self.MODE = 1
    mode_bloq.release()
    finished_movement_confirmation.acquire()
    mutex.release()
```

El hilo de monitorización de velocidad (definido con el método **monitor**), a su vez, se encarga de las siguientes operaciones:

- Comprobación de que la conexión ha sido abierta correctamente.

```
print("monitoring...")
# check the connection is open
if not self.con.send_start():
    print("Error conenction")
    sys.exit()
```

- Ejecución del bucle principal mientras que el modo de ejecución sea distinto de -1. Dentro del bucle, gracias a una serie de condicionales, se ejecuta una sección del programa u otra en función del valor de la variable **self.MODE** (que controla el modo de ejecución). La única instrucción que se ejecuta en cualquier modo de ejecución es un condicional que termina la repetición del bucle en caso de fallo en la conexión.

  Si el modo de ejecución es 0, que es el modo por defecto en el que se inicializa la clase, el UR3E está inmóvil, es decir, su posición presente coincide con la posición objetivo definida por el usuario. Cuando la ejecución del bucle principal llega a la sección del modo de ejecución 0, actualiza los ángulos de las articulaciones y la posición terminal presente del UR3E, actualiza los registros pertinentes para establecer la velocidad a 0 en todas las articulaciones, y entra en el juego de semáforos. El "release" del semáforo **finished_movement_confirmation** desbloquea que la ejecución del método **go_ang**, permitiendo su finalización. El "acquire" del semáforo **mode_bloq** impide que continue la ejecución hasta el momento en el que se produzca la siguiente llamada a **go_ang**.

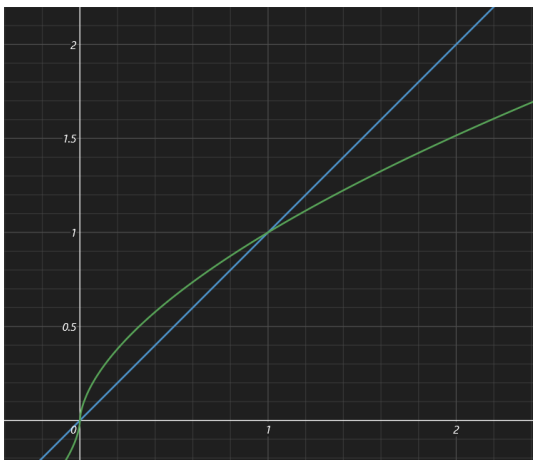
```
if self.MODE == 0:
    self.angles = state.actual_q
    self.position = state.actual_TCP_pose
    self.list_to_setp(self.setp, [0,0,0,0,0,0])
    self.con.send(self.setp)
    finished_movement_confirmation.release()
    mode_bloq.acquire()
```

  Si el modo de ejecución es 1 el UR3E estará en movimiento, es decir, este es el modo de ejecución que adopta el programa cuando la diferencia entre la posición objetivo y la posición real del robot supera el margen de error establecido. En cada paso del programa por la sección del modo de

ejecución 1, se actualizan los ángulos de las articulaciones y la posición terminal presente del UR3E, a través de los métodos **compute_speed** y **compute_control_effort** se calcula la nueva velocidad que debe tomar cada articulación, se actualizan los registros necesarios para establecer las velocidades calculadas, y se comprueba si el UR3E se encuentra suficientemente cerca de la posición objetivo como para finalizar el movimiento y volver al modo de ejecución 0.

```
if self.MODE == 1:
    # get UR3 position
    self.angles = state.actual_q
    self.position = state.actual_TCP_pose
    # compute control effort
    speeds = self.compute_speed(self.target_ang, state.actual_q)
    control_effort = self.compute_control_effort(speeds, self.gain)
    # reformat control effort list into setpoint
    control_effort.append(self.MODE)
    self.list_to_setp(self.setp, control_effort)
    self.con.send(self.setp)
    # check if distance to target pos is enough small
    self.check = self.check_dif(self.target_ang, self.angles)
    if self.check == 1:
        self.MODE = 0
```

El cálculo de la velocidad dentro del método **compute_speed** es una de las cosas que he modificado respecto al desarrollo de Félix. En versiones anteriores del programa, la velocidad en cada momento se establecía simplemente mediante la diferencia entre la posición objetivo y la posición presente del UR3E. Los inconvenientes de este método son 2. En primer lugar, cuando las diferencias entre las posiciones presente y objetivo de una articulación eran muy grandes (ej. -2π y 2π) se obtenían velocidades iniciales en el movimiento excesivamente altas. En segundo lugar, cuando las diferencias entre las posiciones presente y objetivo de una articulación eran muy pequeñas o medianas, el robot se movía muy despacio, empleando un tiempo innecesario en completar el movimiento. Para mitigar en cierta medida ambos problemas, simplemente he agregado un exponente sobre el resultado de la diferencia normalizada.



$$speed = gain \times (target - actual) \quad \rightarrow \quad speed = gain \times (target - actual)^{0.6}$$

9. ESTRUCTURA DE LA CLASE: RobotOrders

La clase **RobotOrders** es mucho más sencilla que la clase **UR3EConnection**, ya que en realidad solo es una intermediaria que adapta todo el desarrollo lógico explicado previamente a una estructura equivalente a la clase que Lucía ha estado empleando para gestionar el control del **IRB 120**. Los métodos de la clase RobotOrders son mucho más auto explicativos, cada uno de ellos se encarga de lo siguiente:

- **__init__ (constructor)**: A parte de llevar a cabo la instanciación del propio objeto, se encarga de instanciar la clase **UR3EConnection** y de llamar a su método **connect2Robot** para iniciar la conexión con el controlador.

```
def __init__(self):
    self.control_info = [0, 0, 0, 0, 0, 0, 0]
    # set the communication
    self.Communication = UR3EConnection()
    self.confirmation = self.Communication.connect2Robot()
    return
```

- **moveJoints**: Recibe un vector de 6 dimensiones conteniendo los ángulos objetivo a los que se desean mover las articulaciones del UR3E a través de una llamada al método **go_ang** de **UR3EConnection**. Antes de finalizar, el método devuelve un valor binario indicando si la nueva posición del brazo es similar a la posición pedida bajo un determinado umbral de error.

```
def moveJoints(self, angles):
    # send the order
    self.Communication.go_ang(angles)
    # recive the new orientation
    joint_values = self.Communication.get_ang()
    jv_high = [item + 0.5 for item in angles]
    jv_low = [item - 0.5 for item in angles]
    # comprobation of joints
    if joint_values > jv_low and joint_values < jv_high:
        check = 1   # right joints' orientation
    else:
        check = 0   # wrong joints' orientation
    return check
```

- **ask4RobotJoints**: Devuelve las posiciones angulares en las que se encuentran cada una de las articulaciones del brazo.
- **ask4AbsPosition**: Devuelve la posición terminal del brazo en coordenadas cartesianas tomando como punto de referencia la base del mismo.

10. CORRECCIÓN DE POSIBLES PROBLEMAS DE CONEXIÓN

Por la forma en la que se ha llevado a cabo el diseño y la colocación de semáforos en el hilo de monitorización en la clase **UR3EConnection**, si tiene lugar un periodo de más de 5 segundos sin producirse una petición de movimiento, el bucle de dicho hilo no pasará por la instrucción **self.con.send(self.watchdog)** (que es la que mantiene abierto el "watchdog") y se interrumpirá la conexión. En principio esto no debería ser un problema de cara al uso que Lucía pretende darle al UR3E, ya que el programa de entrenamiento del modelo de aprendizaje automático estaría moviendo el robot de forma constante. Sin embargo, simplemente por la posibilidad de que en el futuro surja la necesidad de utilizar el UR3E de otra forma, bastaría con modificar la sección del modo de ejecución 0 en el hilo de monitorización de la forma que se muestra en la siguiente imagen.

```
if self.MODE == 0:
    self.angles = state.actual_q
    self.position = state.actual_TCP_pose
    self.list_to_setp(self.setp, [0,0,0,0,0,0])
    self.con.send(self.setp)
    if not mutex.acquire(blocking=False):
        if not finished_movement_confirmation.acquire(blocking=False):
            if not mode_bloq.acquire(blocking=False):
                finished_movement_confirmation.release()
            else:
                mode_bloq.acquire()
```

La idea de la modificación es evitar que el flujo de ejecución se detenga al pasar por esta sección debido al semáforo **mode_bloq**. Sin embargo, sí se mantienen las labores de los semáforos **finished_movement_confirmation** (que evita que el método **go_ang** para ejecutar un movimiento termine antes de la finalización del movimiento) y **mutex** (que impide la realización de un movimiento mientras que la inicialización de la clase no haya terminado).

11. RECAPITULACIÓN DE CÓMO LLEVAR A CABO LA CONEXIÓN

Este apartado está destinado a dar unas indicaciones rápidas para iniciar correctamente la comunicación entre el script de Python desde el que se vaya a realizar el control del UR3E y el controlador del propio UR3E sin necesidad de haber leído detenidamente el resto del documento.

En primer lugar se debe descargar el conjunto de ficheros del repositorio: https://github.com/MiguelMontesLorenzo/UR3E_software.git

y colocarlos en el mismo directorio donde se ubica el script desde el que el usuario pretenda controlar el robot (si lo desea puede eliminar el fichero de test, en este se encuentra un ejemplo de cómo iniciar este proceso de comunicación, pero no es un script necesario para el funcionamiento del mismo).

Para llevar a cabo correctamente la instanciación de la clase de control desde Python primero debe asegurarse de que su ordenador está conectado por ethernet al controlador del UR3E. Dentro del script de su programa debe importar el módulo **UR3E.py**. Por último, debe diseñar el

resto del programa para que la instanciación de la clase de control **RobotOrders** no se produzca antes de tener el programa del "polyscope" listo en el "popup" de entrada. Cuando el programa del "polyscope" alcance este punto es cuando el flujo de ejecución debe ejecutar la sentencia de instanciación:

robot_control = UR3E.RobotOrders()

Una vez completada esta instrucción, entonces puede aceptar la confirmación en el "popup" del "polyscope". Cuando lo haga, el controlador moverá el "polyscope" a la posición inicial (poniendo el brazo en vertical) y entonces se le cederá el control a su objeto de **robot_control**.

Para llegar al punto de inicialización del programa del "polyscope" deberá hacer lo siguiente.

- Encender el "polyscope"
- Iniciar el UR3E
- Abrir el programa ubicado en la ruta:
  home/control_rtde_MM/rtde_joints_control_sincronization_modificada.urp.
- Ejecutar dicho programa
- Esperar a que aparezca en la pantalla un "popup" con el siguiente mensaje:
  "start python connection, then press continue"