Intelligent Robotics MSc Summer Technical Challenge

Mobile Robot Control in V-REP with Python

Mark A Post (mark.post@york.ac.uk)

1. Aims and Objectives

This is the first step in the Intelligent Robotics MSc Summer Technical Challenge.

This challenge can be done from home, either by starting a PC or VM from a bootable Ubuntu image or by installing the needed software on your own computer. It is intended to set you up with a simulation and programming environment that you can use anywhere to complete robotics laboratory tasks.

2. Learning outcomes

By the end of this challenge session, you should be able to:

- Create a robot simulation in V-REP and control a robot with Python
- Test algorithms on virtual robots in place of real ones in the laboratory

3. Software and hardware

- V-REP (CoppeliaSim) Software (https://www.coppeliarobotics.com/)
- Example simulation environment (v-rep-lab.ttt) and control scripts (v-rep-lab.py and v-rep-maze.py) provided in the Appendix.
- Python or iPython development environment (Spyder3 is recommended)

4. Pre-Lab Preparation

- Download/Install Python 3 and at minimum, the python3-numpy and python3-opencv libraries plus any others you want to use for your platform.
    - Note that if you are using Windows you need to make sure that you are running the 64-bit version and also download the 64-bit version of Python as the V-REP .dll libraries are also 64-bit, the link to download Python as of the time of this writing (please use whatever the latest version is) is https://www.python.org/ftp/python/3.8.2/python-3.8.2-amd64.exe

5. Set Up the Remote Python API

V-REP provides several modes of programmatically controlling elements within the simulation via an API framework. The many API options are detailed here:

https://www.coppeliarobotics.com/helpFiles/en/apisOverview.htm

The ROS and BlueZero interfaces may be of the most use for larger robotic projects and you are welcome to use them any time. For the moment, so as to make it as easy as possible to get working quickly with V-REP on any computing platform (without having to install new middlewares or languages), we are providing examples for the Python Remote API.

Remote API Initialization

The Server-side (V-REP) set-up for the Remote API is described here:

https://www.coppeliarobotics.com/helpFiles/en/remoteApiServerSide.htm

To enable the Remote API for your robot, you will need to add a *non-threaded child script* to the base element of your robot that, at minimum, includes in the `function sysCall_init()` part of the script the function that starts the API `simRemoteApi.start(<portnum>)`, where <portnum> is the network port number that you will use to contact your robot in the simulation. This has been done for you already in the "v-rep-lab.ttt" scene provided using network port 19999 – look at the script attached to "Robot" for details. When you run the simulation, the scripts attached to each element run as well, and the remote API will open on the specified port. This allows you to open a different port for each robot or element in your simulation and connect to them through different programs.

Python Remote API

The client-side Remote API set-up is described for all supported languages here:

https://www.coppeliarobotics.com/helpFiles/en/remoteApiClientSide.htm

The standard Remote API for Python does not require additional middleware or messaging, but it communicates with V-REP using a binary shared library and a Python library that you can import. First, find the appropriate shared library for your platform in: `<your V-REP directory>/programming/remoteApiBindings/lib/lib/`

If you are using Linux, you will find a shared library in `<Ubuntu Version>/remoteApi.so` from this directory. If you are using Mac you will find `remoteApi.dylib`, and for Windows you will find `remoteApi.dll`. **Copy this file into the directory you will use for your Python script.**

Next, find the Python files `sim.py` and `simconst.py` as well as some other useful scripts in: `<your V-REP directory>/programming/remoteApiBindings/python/python/.`

Copy all these script files (just to be sure) into the directory you will use for your Python script.

Lastly, place the example scripts `v-rep-lab.py` and `v-rep-maze.py` into this directory as well. To access the V-REP Python API functions in a Python script, import them with `import sim`.

6. Connect to the Robot using the Remote Python API

The `v-rep-lab.py` script provides a basic example of the API functions that you can use to control your mobile robot. It assumes that your robot starts up the API on port 19999. You *must start the simulation in V-REP before you run Python, because the API on this port is started by the Lua script*.

Open the `v-rep-lab.py` script. This is a summary of the various program elements that you may find useful for control of your mobile robot:

Initialization

Import the `sim,time,cv2` (OpenCV),and `numpy` libraries. You can add any more libraries from the vast number available for Python that you might need.

You don't have to use global variables, but it is convenient to define them first. We use the following global variables that are hopefully self-descriptive:

```
leftMotorSpeed = 0.0
rightMotorSpeed = 0.0
leftWheelOdom = 0.0
rightWheelOdom = 0.0
lastLeftWheelPosition = 0.0
lastRightWheelPosition = 0.0
```

To avoid potential confusion, it is a good idea to make sure that all open connections to your robot are closed when starting your control program. You can do this with:

```
sim.simxFinish(-1)
```

Now, connecting to the simulator is a matter of specifying an IP address and port number that the simulation has opened for control of your robot. Note that this makes it quite easy to run the simulation on one computer and run the control program on another computer (even a Raspberry Pi) on your local network. The API function is:

```
clientID = sim.simxStart('127.0.0.1', 19999, True, True, 5000, 5)
```

Check to make sure that you are connected to the simulation. If `ClientID` is -1 it means that the connection is not available. Any other number is a valid ID (usually it starts at 0).

We also record the start time of the simulation with:

```
thistime = time.time()
```

With a valid ClientID, you can get the "handle" ID numbers of all objects that are available from the API instance by searching for their objectName. The objectName is simply the string identifier of each object as it is listed in the Scene Hierarchy. For example, the sphere that makes up the main body of our robot is called "Robot", and the camera used for identifying the maze is called "Camera".

Handles to Objects

After checking whether the simulation is connected with `if clientID != -1:`, a handle to the camera on the robot is obtained with:

```
res,camera = sim.simxGetObjectHandle(clientID, 'Camera',
sim.simx_opmode_oneshot_wait)
```

Several "operation modes" for communicating with the API are available, but for the moment, `sim.simx_opmode_oneshot_wait` will work for most of the functions we need. The variable `camera` is the handle we use to control the camera, and the return value `res` indicates whether the camera was successfully found (the return value `sim.simx_return_ok` resolves to 0).

Other scene objects include the self-explanatory "collisionSensor", "leftMotor", "rightMotor", "leftWheel", "rightWheel", "Robot" (the robot body), and "Floor" (as a relative point of reference).

You can also add a vision sensor named "monitorCamera" (just copy the existing Camera) to work as a "dummy" camera to which you can send images so they can be displayed within the simulation).

Main Program Loop

The main loop of the program should check the connection ID for the client periodically as it will become -1 when the simulation is stopped. You can make the program end when the simulation is stopped by starting your main loop with:

```
while (sim.simxGetConnectionId(clientID) != -1):
```

When you exit the program, you should always close the simulation (even if the simulation ended) with:

```
sim.simxFinish(clientID)
```

If you are using OpenCV or another API, you should make sure to close it as well as needed, for example closing any remaining OpenCV windows with:

```
cv2.destroyAllWindows()
```

Camera Image Processing

Using the handle to the camera, we can obtain a camera image using the following API function:

```
res, resolution, image = sim.simxGetVisionSensorImage(clientID,
camera, 0, sim.simx_opmode_streaming)
```

We capture one frame before the main loop using the operation mode `sim.simx_opmode_streaming` to initialize the streaming of images, and then we capture a still image each iteration of the program loop with operation mode `sim.simx_opmode_buffer` for processing in OpenCV or other API functions. The images are not streamed immediately – some initial time is required until the video stream starts up. While images are not available, the return value `res` will contain `sim.simx_return_novalue_flag`. When an image has been captured successfully, `res` will contain `sim.simx_return_ok`, and you can proceed in processing the image.

The image is stored as a flat list of 196.608 RGB values. The camera is set to 256x256 pixel resolution (this can be changed by selecting the camera and clicking “scene object properties”) and separate values for red, green and blue are stored for each pixel (256*256*3=196608). To use OpenCV for machine vision, you will need to convert this to a Numpy N-dimensional array as follows:

```
cvimg = np.array(image, dtype=np.uint8)
```

The image can then be processed using conventional OpenCV Python functions, for example Canny edge detection (this example works with line mazes in OpenCV, make sure to tune the parameters yourself for other uses):

```
cvedges = cv2.Canny(cvimg, 32, 64)
```

After processing the image, you may have to resize it to ensure it is the same size as the original image, or to a different size if needed. Maintaining the original camera image size can be done by using the `resolution` returned by the image capture function, with:

```
cvedges.resize([resolution[1],resolution[0],3])
```

While developing your algorithms, it is useful to display the image in real time in a separate OpenCV window or several windows. You can display the image in a persistent window with:

```
cv2.imshow('image', cvedges)
```

Finally, you may want to be able to see the processed image back in the simulation rather than having it in separate windows (convenient if screen space is limited). For this, you can use a “dummy” camera such as a vision sensor named “monitorCamera”. It doesn’t matter where you place the camera, but you can externally set the image it returns to the simulator from the Python API so that it acts as a video monitor. First, convert the OpenCV image back into a flat array with:

```
outimage = cvedges.flatten()
```

Then you can set the image that the “camera” is returning using the API function:

```
res = sim.simxSetVisionSensorImage(clientID, monitor, outimage, 0, sim.simx_opmode_oneshot)
```

Note that depending on the format of the image you are using, colours and other properties may be different after you flatten it. The complete optional code for returning a camera to V-REP is:

```
#Convert from OpenCV representation to V-REP
outimage = cvedges.flatten()
#Set the image returned by the "dummy" orthographic camera
res = sim.simxSetVisionSensorImage(clientID, monitor, outimage, 0, sim.simx_opmode_oneshot)
```

Wheel Odometry

Rotary wheel encoders are a fundamental sensor for mobile robots. V-REP does not provide a stock rotary encoder sensor for continuous rotation, such as pulse or quadrature encoders. However, you can read the joint position of the motor shafts directly from the leftMotor and rightMotor using their handles. Joint positions are read as tuples with the position in the second index, so you can use:

```
leftWheelPosition = sim.simxGetJointPosition(clientID, leftMotor, sim.simx_opmode_oneshot)[1]

rightWheelPosition = sim.simxGetJointPosition(clientID, rightMotor, sim.simx_opmode_oneshot)[1]
```

You will also need to measure the radius of your robot's wheels in order to determine how far you will move for each radian of rotation (quite simply, one radian of rotation will move your robot forward the distance of the radius of your wheels). While you can hard-code this based on the design of your robot (the mobile robot provided has a wheel radius of 0.04), you can also calculate this directly from the dimensions of the *bounding boxes* of your robot's wheels (the boxes that enclose the physical extents of objects, used for collision detection among other things.) You can find these parameters in the list of V-REP parameter IDs at https://www.coppeliarobotics.com/helpFiles/en/objectParameterIDs.htm and obtain the diameter of the wheels by subtracting the minimum *X* coordinate of the bounding box from the maximum *X* coordinate in the robot's frame of reference (which has the *X* axis pointing straight forward). The API functions to obtain these (using '\' to continue lines) are:

```
leftWheelDiameter = \
 sim.simxGetObjectFloatParameter(clientID, leftWheel, 18, sim.simx_opmode_oneshot)[1] \
 - sim.simxGetObjectFloatParameter(clientID, leftWheel, 15, sim.simx_opmode_oneshot)[1]
rightWheelDiameter = \
 sim.simxGetObjectFloatParameter(clientID, rightWheel, 18, sim.simx_opmode_oneshot)[1] \
 - sim.simxGetObjectFloatParameter(clientID, rightWheel, 15, sim.simx_opmode_oneshot)[1]
```

Keeping with convention for robot joints, the joint positions are read as angles in radian units and in the range of $[-\pi, \pi]$, causing a *singularity* at $\pm\pi$ radians or ± 180 *degrees*. While it would be straightforward just to track the joint position at each loop iteration and subtract it from the next joint position read, the singularity at $\pm\pi$ means that whenever the joint position crosses ± 3.14, then $\pm 2\pi$ will be erroneously added to the rotation distance. To correct for this, the simplest solution is to detect when the $\pm\pi$ singularity is crossed between loop iterations, and then subtract out the resulting 2π before converting to a distance measurement using the radius (*diameter/2*). This can be done with:

```
dTheta = leftWheelPosition - lastLeftWheelPosition
if dTheta > np.pi:
      dTheta -= 2*np.pi
elif dTheta < -np.pi:
      dTheta += 2*np.pi
leftWheelOdom += dTheta * leftWheelDiameter / 2
lastLeftWheelPosition = leftWheelPosition
dTheta = rightWheelPosition - lastRightWheelPosition
if dTheta > np.pi:
      dTheta -= 2*np.pi
elif dTheta < -np.pi:
      dTheta += 2*np.pi
rightWheelOdom += dTheta * rightWheelDiameter / 2
lastRightWheelPosition = rightWheelPosition
```

Important note: This means that if you run the motors fast enough that they rotate more than π radians (3.14 units) in one cycle of the program, your encoders will become highly inaccurate. This is a key limitation of the current simulation setup (*though if you can develop a better algorithm for unambiguously monitoring wheel speed you will benefit by having a much faster robot!*)

Control

V-REP allows you to set both the position and velocity of actuators from the remote API. To control the mobile robot, generally only control of the velocity is needed (though position control might be

useful for fine manoeuvers by moving the wheels by a precise amount). The velocities of the leftMotor and rightMotor can be set to `leftMotorSpeed` and `rightMotorSpeed` respectively with:

```
sim.simxSetJointTargetVelocity(clientID, leftMotor, leftMotorSpeed, sim.simx_opmode_oneshot)
sim.simxSetJointTargetVelocity(clientID, rightMotor, rightMotorSpeed, sim.simx_opmode_oneshot)
```

You are eventually expected to develop the algorithms for autonomous control of your robot yourself. However, it is very helpful to have some direct control of the robot and its functions while developing your code. There are many ways of setting up interaction with Python, but if you are using OpenCV functions anyway then it is convenient to use the `cv2.waitKey()` function to capture input with OpenCV display windows. A bitwise & with *0x255* is used to ensure that an 8-bit character is returned from the `waitKey()` function, after which it returns a value of 255 if no key is pressed. Any other value will be the keyboard character pressed prior to the function call. A simple keyboard control mapping is included as an example so that you can move the robot by well-known 'w'-'a'-'s'-'d' keyboard commands ('w'=forward; 's'=backward; 'a'=turn left; 'd'=turn right).

```
keypress = cv2.waitKey(1) & 0xFF
if keypress == ord(' '):
      leftMotorSpeed = 0.0
      rightMotorSpeed = 0.0
elif keypress == ord('w'):
      leftMotorSpeed += 0.1
      rightMotorSpeed += 0.1
elif keypress == ord('s'):
      leftMotorSpeed -= 0.1
      rightMotorSpeed -= 0.1
elif keypress == ord('a'):
      leftMotorSpeed -= 0.1
      rightMotorSpeed += 0.1
elif keypress == ord('d'):
      leftMotorSpeed += 0.1
      rightMotorSpeed -= 0.1
elif keypress == ord('q'):
      break
```

The if-then structure is used for clarity. If you are familiar with Pythonic methods, you could re-implement this much more cleanly using a list comprehension or dictionary.

Important note: Since you are using OpenCV keypress functions, the keyboard events are associated with the OpenCV window that is opened with `imshow()`. The Python program will **not** detect your keypresses unless the OpenCV window is raised/selected/highlighted in your window manager. It may be useful to keep this window on top of other windows while using manual control in the simulator.

Tracking and Ground Truth

One of the greatest advantages of a simulation environment is that obtaining accurate ground truth is extremely easy, compared to the complexities of accurately estimating the position of a robot in complex real environments. Although you are generally still expected to guide your robots with sensor feedback and odometry, you may find it valuable to check the state estimations of your robot against the actual positions and velocities that are simulated in V-REP. To do this, you can read the position

and orientation of the Robot body's coordinate system with respect to a stationary reference, for example the coordinate system of the Floor. The API functions to do this are:

```
position = sim.simxGetObjectPosition(clientID, body, floor, sim.simx_opmode_oneshot)[1]
orientation = sim.simxGetObjectOrientation(clientID, body, floor, sim.simx_opmode_oneshot)[1]
```

You can also print out all the relevant parameters for your robot as in the following example:

```
print("Pos:[", round(position[0], 2), round(position[1], 2), round(position[2], 2), "]", \
 "Rot:[", round(orientation[0], 2), round(orientation[1], 2), round(orientation[2], 2), "]\n", \
 "LWheelOdom:", "{:03.2f}".format(leftWheelOdom), \
 "RWheelOdom:", "{:03.2f}".format(rightWheelOdom))
```

Note that this only makes use of a small fraction of the API functions and capabilities described in https://www.coppeliarobotics.com/helpFiles/en/apisOverview.htm. For your projects, you will want to examine the many options in much more detail to decide how to build and control your robot.

TASK 5: MODIFY THE SIMULATION WITH THE REMOTE PYTHON API

It is also possible to use the Remote API to modify the simulation itself, either before or after starting the simulation. When you start V-REP, it opens network port 19997 that serves as a "system management" connection to the API. You can use this port for changing elements of the simulation before starting the simulation and controlling your robot. We will use it for dynamically creating mazes in V-REP that your robot can traverse (as it is tedious to create many mazes by hand).

If you are making your own scene, you will need to create the initial maze elements by yourself using planes. Start by resizing the floor to be at least 4 of the original floor panels in size. Then create a square plane that is 1 unit by 1 unit in size just above the floor in the *Z* direction (it defaults to 0.001 units above the floor, but change it to 0.002 if you want to hide the maze lines underneath). Change it's colour to blue (by making ambient/diffuse red and green 0 and all other components 0) with the Scene Object Properties window, place it near the edge of your floor, and rename it to "Start". Now, copy and paste it with Ctrl-C and Ctrl-V so you have another square plane. Change it's colour to Red (by making ambient/diffuse blue and green 0 and all other components 0) and place it anywhere on the floor. Rename it to "End". Finally, create a single maze segment as a plane that is 0.6 units long (*X*) by 0.1 units wide (*Y*) and the default 0.001 units off the floor (*Z*). Change its colour to green (by making ambient/diffuse red and blue 0 and all other components 0), and place it anywhere on the floor. Rename it to "Line". The script "v-rep-maze.py" will rearrange these components to generate a maze.

To open the V-REP system management port, use the following API function:

```
clientID = sim.simxStart('127.0.0.1', 19997, True, True, 5000, 5)
```

For creating a maze for your robot, there are many limitations to the remote API. Draw objects cannot be wider than about 8 pixels, so the maze is created from non-collidable planes just above the floor. However, the remote API does not appear to have a function to create planes. Therefore, the maze can be created by copying and pasting plane segments, and then moving the start and end points appropriately. The API commands to do this are:

```
res,line = sim.simxGetObjectHandle(clientID, 'Line', sim.simx_opmode_oneshot_wait)
if res != sim.simx_return_ok: print('Could not get handle to Line')
offset = (2.0, 2.0, 0.0)
```

```
rotation = (0.0, 0.0, np.pi/2.0)
lastline = [line]
newline = sim.simxCopyPasteObjects(clientID, lastline, sim.simx_opmode_oneshot_wait)[1]
sim.simxSetObjectPosition(clientID, newline[0], lastline[0], offset, sim.simx_opmode_oneshot_wait)
sim.simxSetObjectOrientation(clientID, newline[0], lastline[0], rotation, sim.simx_opmode_oneshot_wait)
```

The script “v-rep-maze.py” does this to generate mazes that your robot can traverse within V-REP. You can also modify the Python script to change the kind of maze that is produced if you want to further test the capabilities and robustness of your algorithms.

Important note: Changes made to the scene while the simulation is *running* will be removed when the simulation is stopped, while changes made to the scene while the simulation is *stopped* will remain during running and afterwards, as well as being saved when you save the scene. It may be useful to use this feature to either save a maze that you want to run repeatedly under a specific filename, or create a variety of mazes in real time to test your algorithms.

Run the “v-rep-maze.py” script with the simulation open and look at the maze that results. If you want to create a new maze, you can re-run the script and it will replace the maze. When one is generated that you are happy with, save the simulation scene under a new environment for your navigation programming.