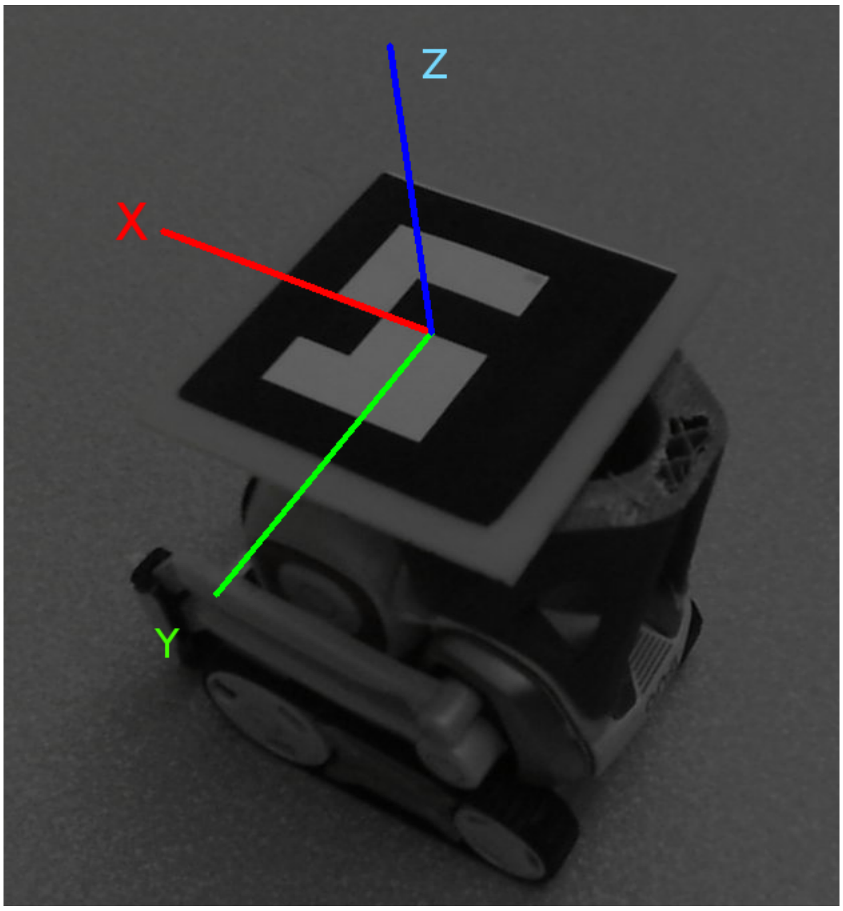
COZMO_FSM MANUAL

George Joseph Nov, 2017

1 Perched Camera SLAM

Perched cameras can now be used as landmarks for the particle filter. As long as the camera can see the aruco marker on the robot, the landmarks are updated using an extended Kalman filter.



To let the camera's see the robots, each robot has to be fit with an aruco marker as shown above. The axis of the marker should fit exactly with the axis of the robot. It was also be parallel to the work plane. Deviations from this will cause errors to accumate over time and build a bad sharedmap.

To start using perched cams, the camera numbers are required. In Linux, camera numbers are allotted starting from 0 and increment with each new camera. Thus typically, the default webcam has camera number 0. To verify the number use **robot.world.perched.check_camera(camera)** command in simple_cli. This will briefly open a window showing the view of the camera. An example is shown below.

```
C> robot.world.perched.check_camera(1)
```

If the camera does not exist, the following error message will be show, and the default camera is used.

```
C> robot.world.perched.check_camera(3)
VIDEOIO ERROR: V4L: index 3 is not correct!
```

After the correct cameras have verified, the perched_camera thread can be started by running **robot.world.perched.start_perched_camera_thread(list_of_camera_numbers)**. This will start the perched camera thread. If robot.aruco_id (id of the marker on the robot) has not been set, a prompt will request it. After entering the value, the perched_camera thread will start running.

```
C> robot.world.perched.start_perched_camera_thread([1,2])
Please enter the aruco id of the robot:90
Particle filter now using perched cameras
```

If the camera can see the given aruco_id, a camera landmark will be added. This can be viewed in the **particle_viewer** (zoom out if not initially visible)

```
C> show particle_viewer
launching opengl event loop
request creation of 1
Type 'h' in the particle viewer window for help.
```

These landmarks are updated as the robot moves. The estimation of the camera position becomes more accurate as the robot moves around.

2 Shared Map

Landmarks and objects can now be shared between robots using the sharedmap system. This requires one server (makes the sharedmap) and multiple clients. Firstly, the server_thread should be started on the server by running **robot.world.server.start_server_thread()**. Again a prompt will request the robot's aruco_id if it has not been set. This will successfully start the server as shown below.

```
C> robot.world.server.start_server_thread()
Please enter the aruco id of the robot:90
Server started
```

Both the client and server should be on the same network.The client_thread is started by running **robot.world.client.start_client_thread("server_ipaddress")**

```
C> robot.world.client.start_client_thread("128.237.138.149")
Please enter the aruco id of the robot:91
Attempting to connect to 128.237.138.149 at port 1800
No server found, make sure the address is correct, retrying in 10 seconds
Connected.
```

Which shows the following message on the server's console.

```
Got connection from ("128.237.136.53", 33298)
Started thread for Client-91
```

Once the server_thread has started, it will keep connecting to new clients automatically. For now, there is a limit of 100 clients. To fully start the shared map, the perched_camera thread must be running on all machines. The landmarks and poses are sent automatically and the transforms are calculated. Please note that to calculate the transform from one robot's frame to another, both robots must be visible in at least one camera at the same time.

The following is the sequence of commands to start the shared map between two robots. Each has one perched camera connected to it.

Server:

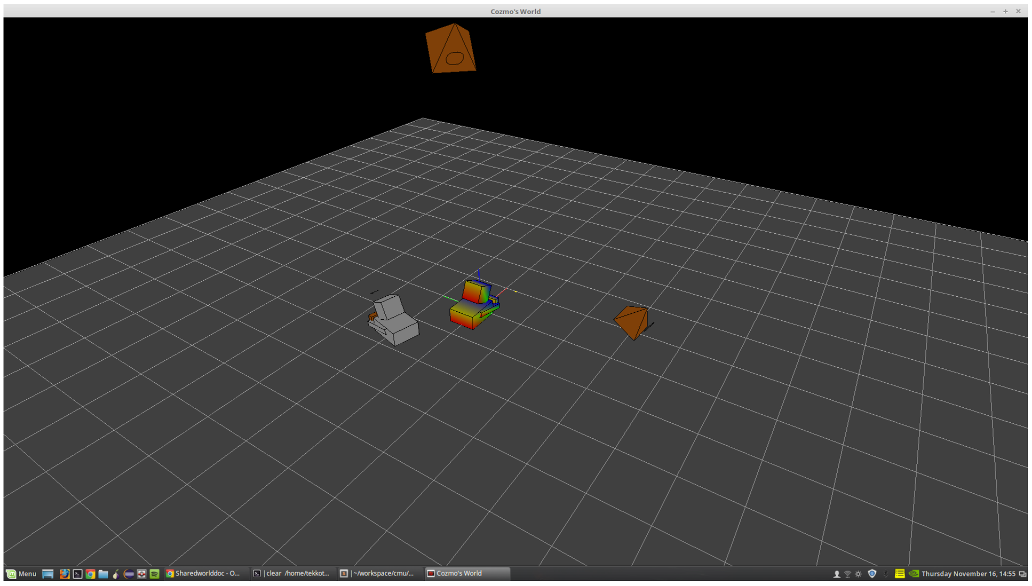
```
C> robot.world.perched.check_camera(1)
C> robot.world.perched.start_perched_camera_thread(1)
Please enter the aruco id of the robot:90
Particle filter now using perched cameras
C> robot.world.server.start_server_thread()
Server started
Got connection from ("128.237.136.53", 53088)
Started thread for Client-91
```

Client:

```
C> robot.world.perched.start_perched_camera_thread(1)
Please enter the aruco id of the robot:91
Particle filter now using perched cameras
C> robot.world.client.start_client_thread("128.237.138.149")
Attempting to connect to 128.237.138.149 at port 1800
Connected.
```

Note that now all the perched camera data is shared among the robots. Thus even if a camera is not directly connected to a robot, it can act as a landmark via the server. Finally run **show worldmap_viewer** on the server to view the shared map. Again, this map become more accurate as the robot moves. Run **WarmUp().now()** on both the server and clients to get an accurate estimation.

```
C> show worldmap_viewer
C> WarmUp().now()
```



A special case is when a robot is not connected to any perched cameras, in that case, it can be part of the sharedmap by running **robot.world.perched.start_perched_camera_thread()**. The following is the special case with the server connected to two cameras and the client to no cameras.

Server:

```
C> robot.world.perched.check_camera(1)
C> robot.world.perched.check_camera(2)
C> robot.world.perched.start_perched_camera_thread([1,2])
Please enter the aruco id of the robot:90
Particle filter now using perched cameras
C> robot.world.server.start_server_thread()
Server started
Got connection from ("128.237.136.53", 43696)
Started thread for Client-91
C> show worldmap_viewer
```

Client:

```
C> robot.world.perched.start_perched_camera_thread()
Please enter the aruco id of the robot:91
Particle filter now using perched cameras
C> robot.world.client.start_client_thread("128.237.138.149")
Attempting to connect to 128.237.138.149 at port 1800
Connected.
```

To safely stop using the perched cameras, run the following command.

```
C> robot.world.perched.stop_perched_camera_thread()
```

Note: While the sharedmap is running all functions that require the pilot fails due to high cpu usage.

3 Doorpass

Doorpass.fsm includes a few new functions as described below,

3.1 GoToWall()

GoToWall(wall_number) instructs the robot to go through the closest door of the specified wall.

```
C> GoToWall(25).now()
```

GoToWall(wall_number, door_number) instructs the robot to go through the specified door of the specified wall.

```
C> GoToWall(25,26).now()
```

Note that the wall must be present in **robot.world.world_map.objects** or a foreign wall sent over the network must be present. The robot adds a wall when it sees atleast two markers of the wall at the same time.

```
C> robot.world.world_map.objects
{"Wall-25": <WallObj 25: (252.7,-23.8) @ 24 deg. for 610.0>}
```

3.2 Explore()

This is just to test the **GoToWall** function. The robot looks around, goes through the first wall it sees and repeats the process until it sees no new doors.

```
C> Explore().now()
```

3.3 WarmUp()

The robot moves forward 10cm and backward 10cm. Useful to initialize the sharedmap for accurate estimation.

```
C> WarmUp().now()
```

3.4 GoToRobot()

Similar to **GoToWall()**, **GoToRobot(aruco_id)** instructs the robot to come face to face with the specified robot. This function requires the sharedmap to be running. Currently, it is quite buggy as the pilot fails while the shared map is running.

```
C> GoToRobot(91).now()
```

3.5 WallPilotToPose()

Similar to **PilotToPose()**, **WallPilotToPose(pose)** instructs the robot to go to the required pose. However, this uses the **GoToWall()** method automatically to use doors if walls are present in the workplane. Currently, it is quite buggy and unpredictable. This code assumes that a straight global path is the optimal path and that there are no obstacles near the doors.

```
C> WallPilotToPose(Pose(1200,0,0,angle_z=Angle(0))).now()
```