Sven Svensson
Anders Andersson
Martina Martinsson

Technical report

Change size of frame:

The window is locked to a certain width and height because of the problem to make the GUI dynamic. So the window is 900x500 by default but if you want to increase the size that is no problem. In the GUI function under """ setup of the window""" you can change the size in the root.minsize(x,y). This will make the windows larger and the increase the space between the buttons in the GUI

Make a certain frame take up more space in the GUI than another:

All the frames in the GUI are locked to take up a certain amount of the total space in the GUI. Lets say you want to add some button to a certain frame and the space is not enough. And a frame next to it have place to spare. Under """creating all the frames in the window """ all the frame are created and you can see for example:
button_frame=Frame(root,bg="white",width=(window_width/4),height=window_height)
That this frame spans 1/4 of the windows width and the whole height of the windows_height.
By changing these values you can make the frame larger.

Adding a button to be executed directly :

This tutorial also makes it possible to add the button to a sequence and save to a file as long as it does not need parameters.

Under #a list containing all the buttons in the upper_button_frame you can see a list containing all the buttons in that frame.
For example: ("forward",lambda:button_pressed("forward",listbox),0.05,0.5).
The first element is the text that will be printed on the button. The next element is a function that sends the buttons text and the listbox in the listbox_frame to a function called button_pressed when the button is pressed. The last two elements are the buttons position in the frame. Just add the text for your button here instead of "forward" and place it somewhere in the frame.

To make something happen when the button is pressed you have to add an element to the dict "lookup" in"get_lookup" function. For example:
"forward":lambda: move_and_paint(move_straight_move_forever,ask_for_frame(*"distance"*))
The first element is the button text, it is important that this is exactly the same as entered in the "lookup" list. The seconds element is the function so be run hidden in a lambda function.

Add a button that can be saved to a sequence and saved to file that needs a parameter:

In the button_pressed() you need to add an "elif" statement. For example:

```
elif string ==*"your button text"*:
    temp_var=ask_for_frame(*"for example distance"*)
        add_program_to_run(string,lambda:your_function(temp_var),listbox)
```

The ask_for_frame() function is used when you need a number and the ask_for_text() is used when you need a str.

In the load_in_command_from_file() you need to add another "elif" statement. For example:

```
elif string =="your button text":
        add_program_to_run(string,lambda:your_function(tuple[1]),listbox)
```

At this moments we only support one parameter but if it requested we can add support for that.

Add your own state to the lower_button_frame:

In the list_of_lower_button on the GUI() function you need to add a new element. For example:

```
("your state",lambda:change_state("your state",lower_button_frame),0.65,"red")
```

The first element is the name of your state, the second is a functions that takes your name as a parameter. The third element is where on the y-axis in the frame the button should be placed. The fourth element should always be "red".

In the end of the function change_state() you need to add an "if" statement. For example:

```
if mode=="your mode":
        your_mode_function(move_on_canvas)
        change_state("run state",frame)
```

you need to have a function that will run when the state is changed. If you want the robot to move on the canvas while running your_mode_function() you need the function move_on_canvas() but this is not mandatory. When your function is done the state will be changed to run state.

Add some information that should be updated continuous in the canvas:

Somewhere in the GUI() function create an text object to be placed in the canvas as such:

```
global canvas_text
canvas_text=canvas.create_text(80,10,text="your text",anchor="ne")
```

the two first elements determined where the text will be placed in the canvas. The text element is the the text that will be shown when the GUI is started.

Create a function that will update that value:

```
def your_function():
    global connected,canvas_text,canvas
    if connected:
        try:
            text= "text: " + str(#get information from a sensor here)
            canvas.itemconfig(canvas_text, text=text)
            root.update()
            root.after(1000,your_function)
        except AttributeError:
            pass
```

This will update the value in the canvas once every second if you want to change the refresh rate change the first parameter in "root.after(1000,your_function)" to a value in ms that suits your needs.

Right before mainloop() in the GUI() add: root.after(10000,your_function) so the function will be called upon efter the mainloop is initiated.

Bind a function to a certain key:

In the key_start_fn() function add an "elif" statement. For example:

```
elif event.keysym==any_key:
        your_function()
```

change any_key to the key you want to bind and change your_function() to that function that want to be run when the button is pressed.

If you want a function to start when you press the key and stop when you release the key you do exactly as above but the function makes something happen, for example the robot starts moving forward. In the key_stop_fn() you add an "elif" statement. For example:

```
elif event.keysym == "any_key":
    stop_your_function()
```

Modules

The following modules are python files used by the GUI or through user interaction in a terminal to make the robot perform certain actions. The files are motor_control.py, sensors.py, weapon_control.py and main.py respectively. Here comes an explanation of what these modules are meant to do.

Motor control

Motor control converts more user-friendly units, such as distance in meters to the robots own rotational degrees. Functions involving measuring distances and following and finding walls are also here. The first functions you will find involve turning the robot using certain motors. For example, the function turn_right_right_wheel() indicates that the robot is to turn his right wheel a certain degree, making a right turn. Other functions include turn_right_two() which turns the robot by activating two motors at once. There is also a function turn_two(degree) which allows the user to turn the robot a set direction.

The other functions define how the robot should move straight forward. There is move_straight_move(distance) which allows the user to input a distance with which the robot should travel. Move_straight_move_forever(distance) does the same thing, however it uses the built in function move_forever as well as a sleep() to determine for how long the robot should keep moving in order to go the distance it should. The sleep is calculated by the speed with which the robot moves.

The last functions, beside the ones related to following a wall, take in distance, a condition and an action. These functions allow the robot to keep moving a distance until the condition is fulfilled. When that happens, the robot stops moving, executes the action (func1), and then goes on to keep going the rest of the distance. Move_straight_condition() also takes in a mode, so that the robot can do different things after performing the action. It also uses a function to_GUI, so that it can draw on the canvas.

Follow and find wall are two functions which makes the robot follow and find a wall respectively.

Follow_wall() makes the robot calculate how oblique its path is compared to the wall beside it. After a while it makes adjustments and continues going on its new path. Find_wall() enables the robot to find a wall and turn so that it is standing parallel to it.

Sensors

The sensors module contains functions which utilizes the sensors on the robot, for example the ultrasonic sensor and the color sensor. It makes detecting a room or resetting the ultrasonic sensor much easier. Set_ultra_zero() resets the ultrasonic sensor to its original state and set_ultra_degree(degree) turns in in a degree determined by the user. The room_layout() functions turn the ultrasonic sensor in certain degrees, enabling it to return readings of how the robot is standing in relation to the room.

Weapon control

In weapon control there are 2 functions which control how balls can be fired when a ball shooter is attached to the robot. If you call the function shoot_one_ball(), on ball is fired from the cannon and if you call upon shoot_several_balls(n) an n number of balls will be fired.

Main

In main is where the highest level functions are (besides from ones located in the GUI). The states peaceful, guard_room, guard_mode and hide_mode are the different modes the robot can be in. In peaceful mode, the robot moves around the room randomly, playing sound files when near an object.

In guard room, the robot measures the premises and then starts guarding it. It does so by driving around the room randomly and shooting balls towards intruders. If the ball count reaches 0, the robot enters hide mode in which it locates a corner and hides in it. Guard_mode is a variation of this, only the robot guards a 1x1 area by following a pattern.

Further down are functions that are generally good to have, functions which you can put into other functions or use in order to get the robot to a certain point. It can, for instance, find the closest point in front of it by turning 360 degrees, check if it is in a corner by doing the same thing, and also measure distance by calculating how far it has traveled in a certain time. It can also use functions in this part of the file to dance, attack by shooting balls, measure distance of a room (which data can be sent to the guard_room() function) and also navigate through a room.