Capstone Python Project – ***Features***

CSSE 120, Introduction to Software Development

General instructions:

The following assumes a 3-person team. If you are a 2-person team, see your instructor for how to deal with that.

All features must be implemented in a nice ***Graphical User Interface*** (***GUI***) to which all team members must contribute.

- All input and output must be from your GUI (plus any external devices you might use, like Roomba remotes). There must be ***no input or output from/to a Console window***.
- The more ***different kinds*** of GUI widgets, the better.
- The more your follow good GUI design principles[1] (and you can ***explain how your GUI does so***), the better.

There are ***green***, ***blue***, ***yellow*** ***(sorry, no yellow this term),*** and ***uncolored*** features. Greens and blues are simpler robot ideas. Yellows are more sophisticated robot ideas (that are more challenging to implement). Uncolored features are open-ended opportunities that vary wildly in difficulty.

1. **The team should work collaboratively to accomplish the *green* tasks at the beginning of the project.** Help each other as needed.
2. **Each student must thereafter implement *one blue feature*.** Coordinate to determine who does which. Two-person teams can choose any two of the three.
3. **For a good score on the project, each student** should ***also*** implement some (not all!) of the uncolored features. There are a wide variety to choice from. Talk with your instructor to help decide which to try!

The best projects will take care to re-use each other's GUI, functions and data wherever practical.

Grading and demos:

The grading is based on:

- the quality of your individual contributions to the project;
- your attention to detail to the Trello board, SVN, and your code;
- your effectiveness as a member of your team.

Due date:

The final project code is due on Monday, August 24. How to demo the project is TBD.

[1] Ben Schneiderman, *Eight Golden Rules of Interface Design.* https://www.cs.umd.edu/users/ben/goldenrules.html.

Features (brief version – see long version for full details):

1. The user can ***connect and disconnect*** to the robot, after specifying whether or not to use the simulator and if not, what port to use for connecting. The program should behave reasonably if the user errs (e.g. by choosing a wrong port, or connecting to an already-connected robot).
2. The ***GUI indicates,*** for ***each Sprint*** and ***each team member***, ***the total hours that the team member worked on that Sprint.*** The data for this item should be ***read from the files*** for *hours-worked* that each team member must maintain, in the *process* folder of the project.
3. ***Move autonomously***, by going a specified ***distance*** in a specified ***direction*** at a specified ***speed***. That is, the user can set the direction (forward, backward, spin left or spin right) and the distance and speed (each in some reasonable units). Then, the user can make the robot go (e.g. by pressing a *Go* button) and the robot should move the specified direction for the specified distance at the specified speed, with some reasonable accuracy.

   An important by-product of this feature is to provide a good set of functions that teammates will use for most of the movements that they ask of the robot. The ***re-usability*** of those functions will play a large role in determining your grade on this feature.
4. Be ***tele-operated*** (i.e., remote-controlled, like a remote-control car) with a **Roomba remote**. The user can use the d-pad to make the robot move in any direction (forward/backward, spin left/right) at three different speeds. See the next page for **implementation requirements**.
5. ***Follow a curvy black line*** using PID control.
6. Move through a sequence of user-specified ***waypoints***.
7. Connect an IR "hat" (or use the remote control) to the robot and **tele-operate other robots**.
8. ***Move autonomously***, by going until a specified ***sensor*** reaches a specified ***threshold***. Sensors should include the bump sensors and the 4 cliff sensors, at the least.
9. Play ***N random notes***, where the user specifies N. The notes must not be "clipped".
10. ***Follow another robot***.
11. Robot composes new music.
12. Do sophisticated movements, e.g. trace a regular polygon or parallel park as in the video at https://www.youtube.com/watch?v=N4F0-MXK5jM.
13. Do interesting things with its ***internal*** sensors.
14. Do interesting things with ***computer vision*** (using the camera), e.g. finding objects, using semaphores to communicate, or …
15. Do interesting things with ***external motors and/or servos.***
16. Use ***swarm techniques*** and/or distributed algorithms to accomplish interesting things.
17. Use ***parallel algorithms*** (in processes and/or threads, in a single processor or across cores) to accomplish interesting things.
18. Use ***internet communication*** and/or ***files*** to do interesting things.
19. ***Compose a fictitious bio*** for itself and/or for you.
20. Use a ***Leap Motion device*** (and accompanying Python software) to control the robot with hand movements.
21. ***Interact with a different kind of robot***, e.g. a quadcopter or BERO robot.

22. Do something interesting… ***[You suggest what!]***

Features – with details:

1. The user can ***connect and disconnect*** to the robot, after specifying whether or not to use the simulator and if not, what port to use for connecting.

   The program should behave reasonably if the user errs (e.g. by choosing a wrong port, or connecting to an already-connected robot).

2. The ***GUI indicates,*** for ***each Sprint*** and ***each team member***, ***the total hours that the team member worked on that Sprint.*** The data for this item should be ***read from the files*** for *hours-worked* that each team member must maintain, in the *process* folder of the project.

   Whoever implements this feature determines the format for the *hours-worked* files, and conveys that format to her teammates. Each team member maintains her own file per the format.

3. ***Move autonomously***, by going a specified ***distance*** in a specified ***direction*** at a specified ***speed***. That is, the user can set the direction (forward, backward, spin left or spin right) and the distance and speed (each in some reasonable units). Then, the user can make the robot go (e.g. by pressing a *Go* button) and the robot should move the specified direction for the specified distance at the specified speed, with some reasonable accuracy.

   An important by-product of this feature is to provide a good set of functions that teammates will use for most of the movements that they ask of the robot.

   Advanced options include:

   - There are multiple implementations (any of which can be chosen by the user), with demonstrated understanding of when and why one is better/worse than another. For example, one implementation is the "time" approach, another is the "distance sensor" approach (which itself is really a collection of approaches parameterized by the time to wait between sensor readings), and a third is the "send a script" approach.
   - There is high accuracy for the best implementations.
   - Can move linearly and angularly (hence along a curve) at the same time, with some reasonable understanding of "distance" and "speed" in that case.
   - The motion can be interrupted by the user.

4. Be ***tele-operated*** (i.e., remote-controlled, like a remote-control car) with a **Roomba remote**. The user uses the d-pad to make the robot move in any direction (forward/backward, spin left/right) at three different speeds:

   a. **Slow**: 20 cm/s

   b. **Medium**: 30 cm/s

   c. **Fast**: 40 cm/s

   Assign the Roomba remote buttons to the following actions:

   - Spot: set to Slow speed
   - Clean: set to Medium speed
   - Max: set to Fast speed
   - *Hold* D-pad up: move forward at the set speed
   - *Hold* D-pad left: spin left in place at the set speed
   - *Hold* D-pad right: spin right in place at the set speed
   - *Hold* D-pad down: move backward at the set speed

   If the user lets go of a movement button, the movement stops.

   Advanced options include:

   - Make the **P** button at the top of the remote change the robot into **sound mode**. While in this mode, pressing the remote buttons does not move the robot, but instead causes

the robot to play different sound tones. Pressing the P button *again* goes back to movement mode.

- Make the **red pause button** on the remote toggle tele-operation. With this option, the robot should always listen for the IR signal, and when the user presses the red button, the robot will begin tele-operation. When the user presses the red button again, the robot should stop tele-operation and allow other features to execute normally.

5. ***Follow a curvy black line*** using bang-bang and PID control.

   See the PID video for an explanation of bang-bang and PID control.

   Assume a curvy black line about 2 inches wide, with reasonably gentle curves, using the left front signal (for the left wheel speed) and the right front signal (for the right wheel speed). (You can also use other sensors if you wish.)

   First implement bang-bang control. Then implement ***P (proportional) control***, with the P constants tuned reasonably.

   Warning: The simulator is WAY different from real robots for this feature. Start with the simulator, but realize that real robots require SUBSTANTIAL tuning. (Their IR light sensors vary wildly from robot to robot, and even within a robot!) You **must** provide an interface to calibrate the darkness of the lines under current lighting conditions. The human may place the robot in positions as desired. You must not hard-code the darkness of the lines into the program.

   Advanced options include:

   - Use I and D (the rest of PID). Make a line where they help.
   - The user can set all the parameters at run-time, ideally even while the robot is doing line-following.
   - The line-following can be interrupted by the user.
   - Uses additional sensors to enable following more challenging lines.
   - Can follow a curvy wall, using a “bump and bounce” algorithm that is akin to bang-bang control.
   - Can follow a curvy wall, using the wall sensor but (ideally) the same PID code as for line following.

6. Move through a sequence of user-specified ***waypoints***.

   That is, the user can enter a sequence of (x, y) coordinates and tells the robot to go. Then, the robot moves to each, one after the other. (The origin of the coordinate system is where the robot began the sequence of moves.)

   Advanced options include:

   - There is a nice way to enter coordinates (e.g. by clicking on a map displayed in a window).
   - The path of the robot is shown on a window as the robot moves.
   - The movement can be interrupted by the user.
   - Coordinates can come from a file.
   - The robot can move around obstacles as it moves from waypoint to waypoint.
   - User can control speeds as well (perhaps via pre-specification, perhaps via tele-operation, perhaps both).
   - The robot remembers paths on which it is tele-operated and then can reproduce the paths autonomously.
   - The robot keeps track of its position through ALL its movements (even those produced by teammate's code) and can reproduce them from any point the user specifies.

7. Connect an IR "hat" to the robot and **tele-operate other robots**.

   Note: This description uses IR, but other sensors might be able to be used in a similar way. *Check with your instructor before beginning to implement this feature!*

   The robot can wear an IR transmitter ("hat") accessory that broadcasts in all directions. Program the robot to broadcast the same IR signals as the Roomba remote. Then gather several robots and run them in tele-operated mode, described in #6. The robot wearing the IR hat should transmit signals to control the other robots and lead a **line dance***.*

   Advanced options include:

   - Have the robot read dance moves from a **file**.
   - Have the robot lead a dancing ***and singing*** party. The robot should transmit same signal as the **P** button on the Roomba remote during the dance to make the robots start singing.

   IMPORTANT: You get NO credit for time spent typing in long sequences of dance moves, notes, or long periods of time transliterating a song to the Create's MIDI note system. Focus on the interesting things you can do via the programming.

8. ***Move autonomously***, by going until a specified ***sensor*** reaches a specified ***threshold***. Sensors should include the bump sensors and the 4 cliff sensors, at the least

   In particular, the user can set the speed and which bumpers to use (both, just-left or just-right). Then, the user tells the robot to start, at which point the robot moves until the relevant bumper(s) are pressed.

   Likewise, the user can set the speed, which of the four cliff sensors to use, and a "darkness level." Then, the user tells the robot to start, at which point the robot moves until the specified sensor sees sufficient "darkness".

   Advanced options include:

   - The user can choose from sensors beyond the bump and cliff sensors.
   - The user can choose one or more sensors to be active in determining when to stop. For example, the user might choose the left bump sensor, the right bump sensor, or both.

- The user can choose from different kinds of sensors (combining bump and cliff sensors, for example), if a variety of ways.
- The "stopping condition" can be more than just a threshold whose value is exceeded.
- The best implementations might require multiple sensors mixed in interesting ways, e.g. the infrared hears 100 followed a second later by 200.
- Perhaps the coolest implementation would allow the user to supply a function definition (written "on the fly") for the stopping condition.
    - Play ***N random notes***, where the user specifies *N.* The notes must not be "clipped". Hint: Use the *song_playing* sensor appropriately to avoid clipping.

The user specifies the length of time each note should be played -- either a fixed length of time, or a range from which the time should be chosen at random.

9. Follow another robot.

Uses the camera, or uses IR emitters with black tape over the IR sensor for directionality, or the other robot sends codes to indicate directionality.

10. Compose music. The **robot** composes the music, **not** you, according to the principles of music theory, and plays the music. Option: do likewise for dances and/or light shows.
11. Do sophisticated movements, e.g. trace a regular polygon or parallel park as in the video at https://www.youtube.com/watch?v=N4F0-MXK5jM.
12. Do interesting things with its ***internal*** sensors. One simple example: there is a sensor that, perhaps surprisingly, can tell when a robot is "stuck" even when the robot is attempting to move BACKWARDS.
13. Do interesting things with ***computer vision*** (using the camera), e.g. finding objects, using semaphores to communicate, or …
14. Do interesting things with ***external motors and/or servos.***
15. Offer ***Rogerian psychotherapy***, ala Eliza (http://en.wikipedia.org/wiki/ELIZA).
16. Use ***swarm techniques*** and/or distributed algorithms to accomplish interesting things.
17. Use ***parallel algorithms*** (in processes and/or threads, in a single processor or across cores) to accomplish interesting things.
18. Use ***internet communication*** and/or ***files*** to do interesting things.
19. ***Compose a fictitious bio*** for itself and/or for you.
20. Use a ***Leap Motion device*** (and accompanying Python software) to control the robot with hand movements.
21. ***Interact with a different kind of robot***, e.g. a quadcopter or BERO robot.

22. Do something interesting… ***[You suggest what!]***