Week 11 Assignments: Multitasking in a Single-Threaded Environment

General Submission Guidelines:

1. Code Quality:
   - Your `.mlapp` files should be well-organized, with clear and meaningful variable names.
   - Include comments to explain key sections of your code.
   - Follow good programming practices for readability and maintainability.
2. Testing:
   - Ensure your applications run without errors.
   - Test all functionalities thoroughly before submission.
   - Verify that the user interface remains responsive during operation.
3. For all solutions, use the provided `Template.mlapp` file. In the code section you will see two custom functions `setup()` and `loop()`. This week's homework is designed around single threaded programming. This is a general structure for single-threaded programming such as with microcontrollers.
   - The `setup()` function will run once, you can use this to set up variables, etc
   - The `loop()` function will run endlessly once `setup()` has run once.
   - All programming shall use these two functions (along with any other custom functions you develop)
   - No callbacks will be used with the exception of one, which is allowed for the PID simulator as that will be necessary in order to get instant feedback from the line position slider.
   - See image below for clarification, new code is allowed in the green are with the two pre-defined functions and any of your own you wish to create. You may also use the properties as you see fit. No code is allowed in the red area where callbacks are created *except* for the PID simulator.

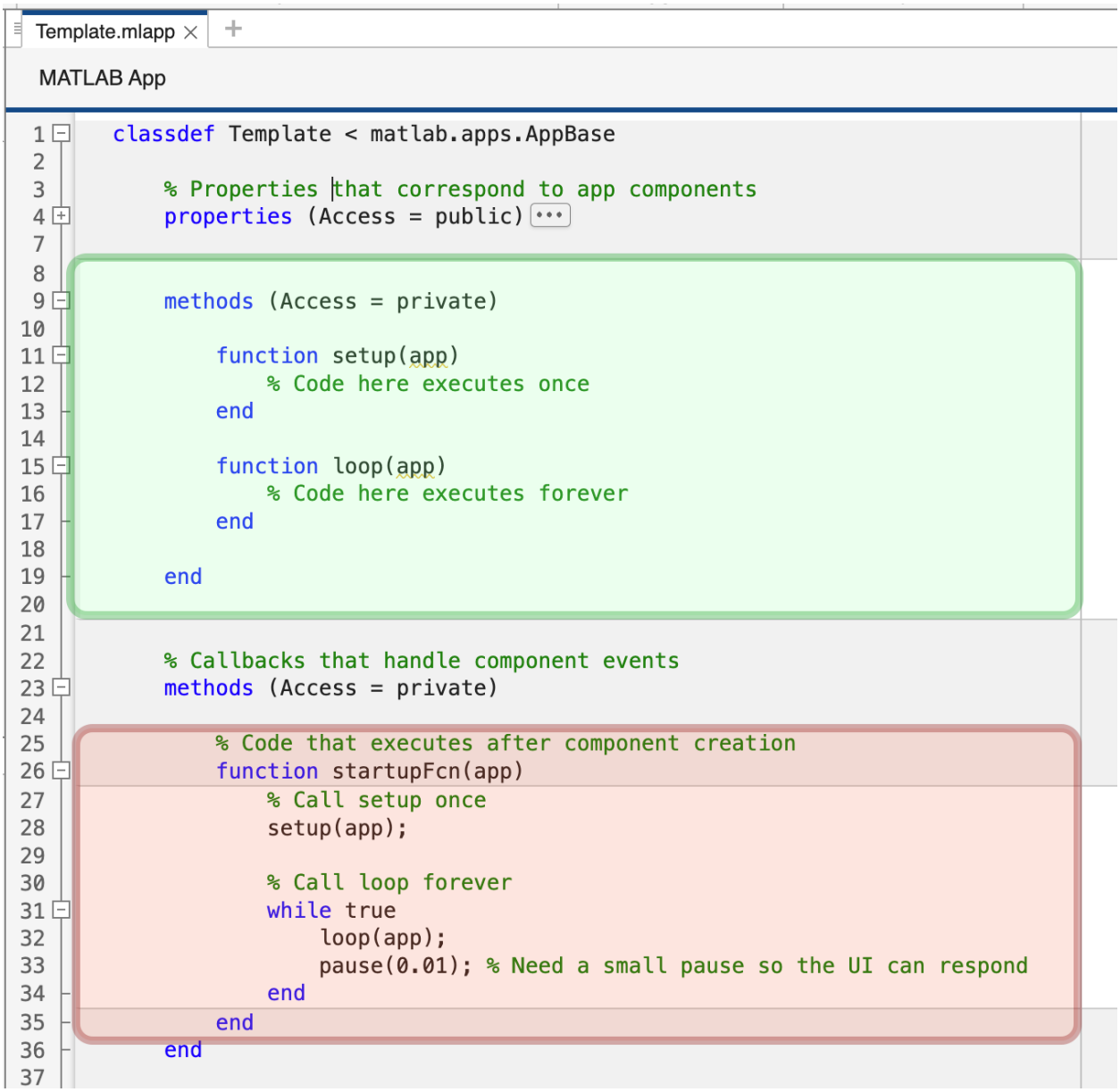

Figure 1: do code and do not code

Examples

As always, the examples are merely used to illustrate what is expected. You have creative freedom to design your applications as you see fit, as long as they meet the requirements.

Assignment 1: Blinking an LED with pause

Objective:

Understand the limitations of using blocking functions like pause in a single-threaded environment by creating a simple application that blinks an LED and updates a timer label.

Requirements:

1. Create a MATLAB App Designer application that includes:
   - An LED (lamp) component that can be turned on and off.
   - An element or elements to start and stop the blinking process.
   - A label to display a timer showing the elapsed time since blinking started.
2. Implement the blinking functionality using the pause function:
   - The LED should turn on for 1 second, then off for 1 second, repeatedly.
   - Use `pause(1)` to create the delay between toggling the LED state.
3. Observe the behavior:
   - Note how the use of pause affects the updating of the timer label.
   - Observe any issues with the application's responsiveness during blinking.
4. Submission:
   - Commit and push your MATLAB `blinkWithDelay.mlapp` file to your repository.

Tips:

- Remember that `pause` halts the entire execution of the program, which may prevent other tasks (like updating the timer label) from running.
- Pay attention to how the user interface behaves when pause is in use.

Example

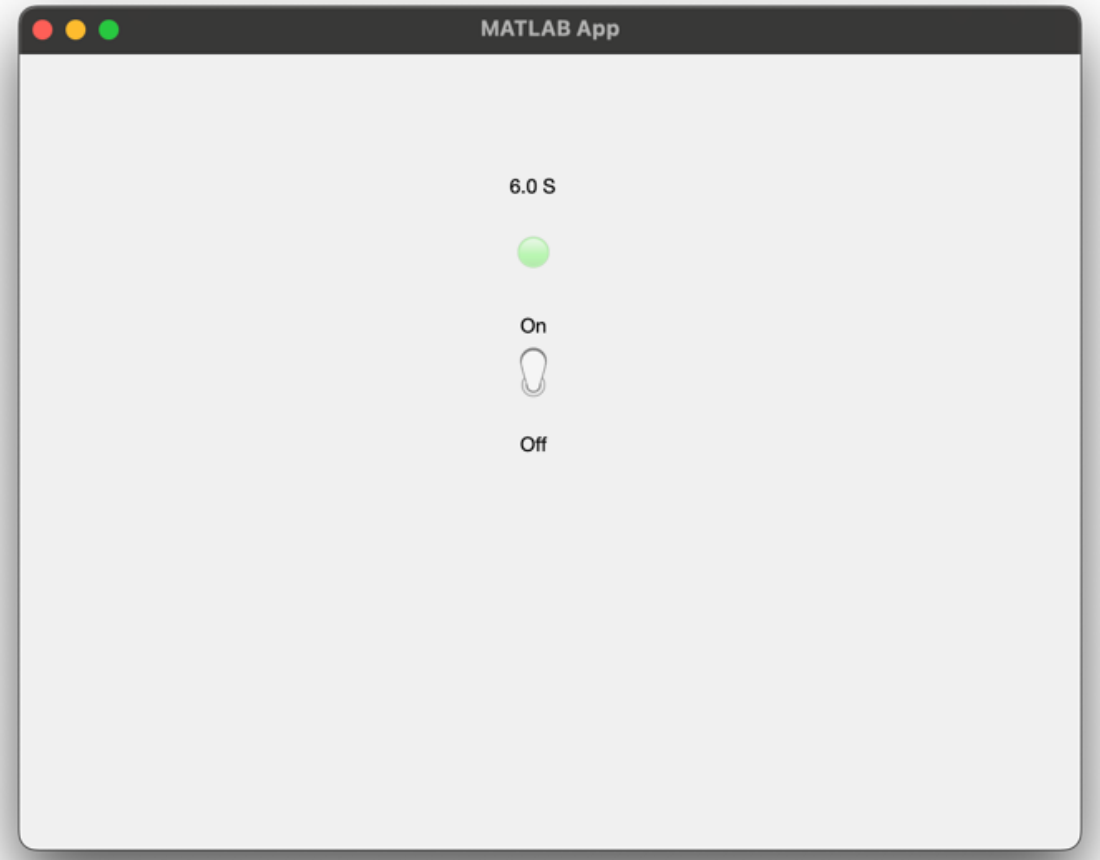

Figure 2: blink with delay

Assignment 2: Blinking an LED without pause

Objective:

Learn how to implement non-blocking delays using elapsed time checks to create a responsive application that blinks an LED and updates a timer label simultaneously.

Requirements:

1. Modify your application from Assignment 1 to:
   - Replace the use of pause with non-blocking timing methods.
   - Ensure the LED continues to blink on and off every 1 second.
   - Keep the timer label updating continuously, showing the elapsed time.
2. Implement non-blocking blinking:
   - Use tic and toc to track elapsed time.

- Use a persistent variable or a property to keep track of when the LED was last toggled.
- In your event loop, check if enough time has passed to toggle the LED without stopping the entire program.

3. Ensure application responsiveness:
   - The user interface should remain responsive during blinking.
   - The timer label should update smoothly without delays.
4. Submission:
   - Submit your updated MATLAB `blinkNoDelay.mlapp` file.

Tips:

- Use conditional statements based on elapsed time to decide when to toggle the LED.
- Keep your main loop running continuously, checking for tasks to perform without using pause.

Example:

Figure 3: blink no delay

Assignment 3: Creating a Dynamic GUI with Multiple Components

Objective:

Design a graphical application that demonstrates multitasking in a single-threaded environment by managing multiple GUI components that update independently at different intervals.

Requirements:

1. Create a MATLAB App Designer application that includes at least five different GUI components (e.g., Lamps, switches, sliders, gauges, text labels, etc.).
2. Implement independent updates for each component:
   - Each component should have at least one property (e.g., color, text, value) that updates at its own specific interval.
   - The update intervals for the components should be different (e.g., one updates every 0.5 seconds, another every 1.2 seconds, etc.).
3. Demonstrate multitasking:
   - Use an event loop to manage the updates without blocking the main execution.
   - Ensure the user interface remains responsive, and all components update as expected.
4. Visual Effect:
   - When running the application, you should see the components changing colors, text, values, etc., independently and at different frequencies.
5. Submission:
   - Submit your MATLAB `crasyDashboard.mlapp` file.
   - Include a short write-up (1-2 paragraphs) explaining:
   - How you implemented the independent updates.
   - The challenges faced in managing multiple tasks.

Tips:

- Assign each component a `lastRunTime` and an `updateInterval` to manage their updates.
- Use each component's `UserData` property to store timing information.

Example

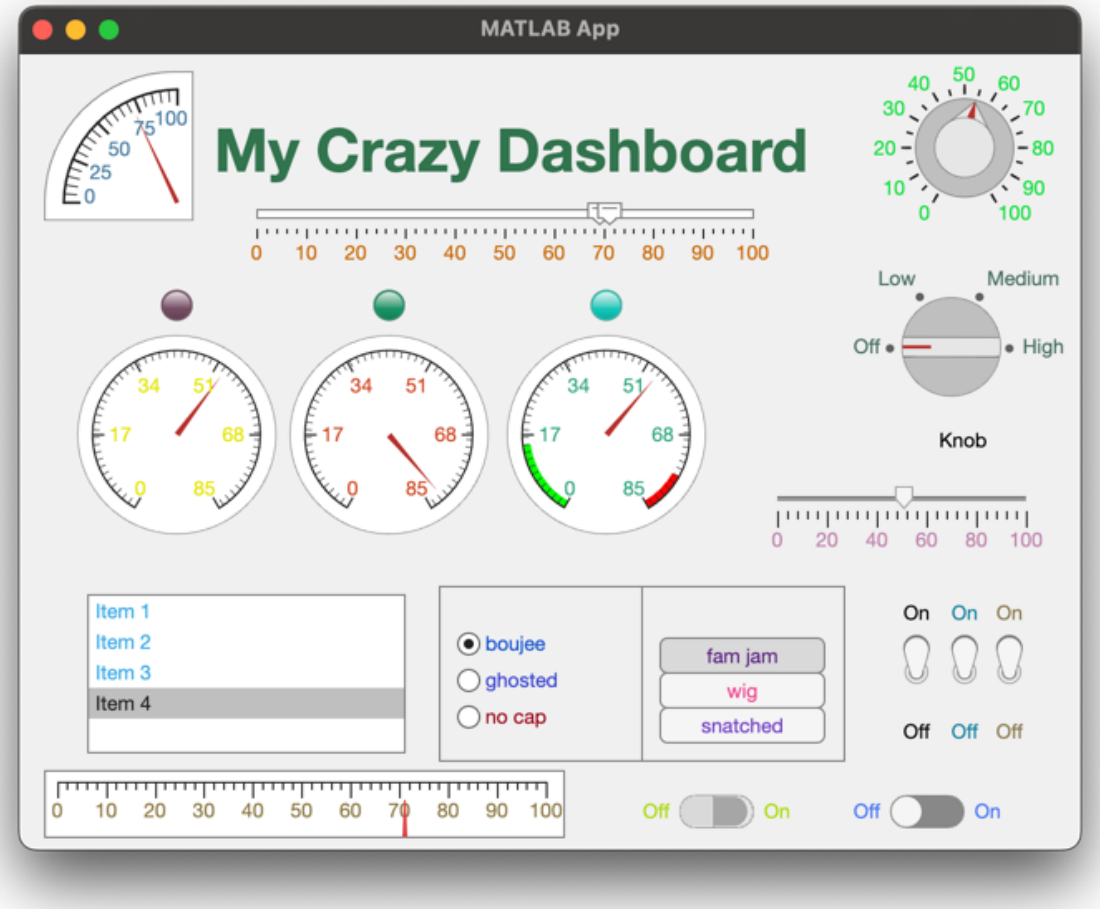

Figure 4: crazy dashboard

Assignment 4: PID Line-Following Simulator

Objective:

Develop a graphical application that simulates a line-following robot controlled by a PID (Proportional-Integral-Derivative) algorithm using only the P (Proportional) and D (Derivative) components. The application should allow the user to adjust parameters and visualize how the robot adjusts its path.

Requirements:

1. Create a MATLAB App Designer application that includes:
   - Input fields or controls for:
   - Goal setpoint (desired line position).
   - Set motor speed.
   - PID gains: kP (proportional gain) and kD (derivative gain).

- A slider representing the line position sensor input.
- Gauges or displays for the left and right motor speeds.
- A switch or button to start and stop the simulation.

2. Implement the PID control algorithm:
   - Use the P and D components to calculate adjustments to motor speeds based on the error between the line position and the setpoint.
   - Update the motor speed gauges in real-time as the simulation runs.
     - **Important** This should also update as the line position slider is changed (instantaneously)
   - Store previous error values for the derivative calculation.
3. Simulation behavior:
   - When the simulation is running, adjusting the line position slider should cause the motor speeds to adjust accordingly, simulating how a robot would correct its path to follow a line.
   - The application should visually reflect the changes in motor speeds, showing how the robot responds to the line position.
   - Adjust the numbers and see how a high kP is useless beyond a certain point, but too low is not significant enough, etc.
4. Submission:
   - Submit your MATLAB `pidSimulator.mlapp` file.

Tips:

- Use an event loop to continuously read the line position and update motor speeds without blocking the UI.
- Ensure the application remains responsive during the simulation.
- A template GUI will be provided; you can use it as a starting point for your application.

Example

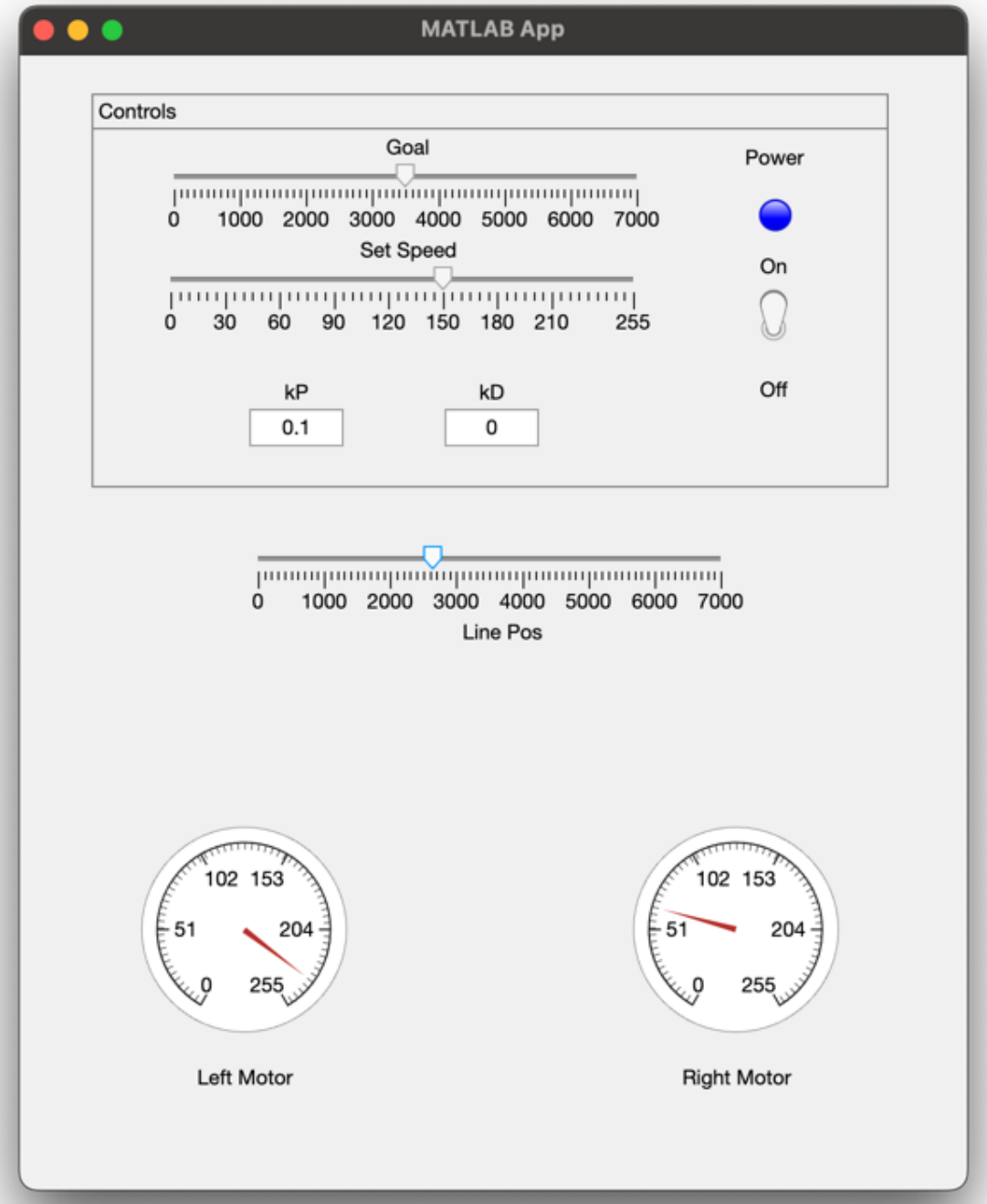

Figure 5: pid simulator