CS 4/6545 Autonomous Robotics: Feedback control

Read all instructions carefully. Your submission will consist of all code and plots. If you are using your own `Ubuntu Linux` installation, you will need to install the following packages:

- `python-matplotlib`
- `libopenscenegraph-dev`
- `libqhull-dev`
- `liblapack-dev`
- `libboost-all-dev`
- `libav-tools`
- `Ravelin` (PPA)
- `Moby` (PPA)

1. **Build the controller plugins for the simulated robot.** The controller plugins use error feedback control to "control" the robot.

   You do this by building the simulator's "control plugin"—*without actually implementing a controller*—and then running the simulation:

   ```
   drum:~/motion-control-manipulator$ # this is a comment
   drum:~/motion-control-manipulator$ mkdir debug # CMake does not like to build in
       the main directory
   drum:~/motion-control-manipulator$ cd debug
   drum:~/motion-control-manipulator/debug$ cd debug
   drum:~/motion-control-manipulator/debug$ cmake .. # run CMake to build the
       Makefile
   drum:~/motion-control-manipulator/debug$ make # build
   drum:~/motion-control-manipulator/debug$ cd ..
   drum:~/motion-control-manipulator$ # run Moby
   drum:~/motion-control-manipulator$ moby-driver -r -mt=13.0 -p=./debug/
       libsinusoidal-controller.so ur10.xml
   ```

   A few items of note:

   - The simulator simulates the robot's movement over 13.0 seconds of virtual time.
   - The simulation does not run at real time speed.

2. **Set some tentative gains for a PID controller** in `gains.dat`. The first column of each line of the file corresponds to the joints that you are setting the gains for. The second through fourth columns correspond to the proportional, integrative, and derivative gains, respectively.

3. **Implement a PD controller for driving the robot through a sinusoidal motion** by editing the controller in `sinusoidal-controller.cpp`. I have taken care of one tedious part for you already: determining the joint angles for the sinusoidal motion. You need to:

   (a) Lookup the gains for each joint (using the STL `map` data structure)
   (b) Get the current joint position (this is done for you already)
   (c) Determine the desired joint position (stored in the `q_des map`)
   (d) Compute the *positional error*
   (e) Get the desired joint velocity (this is done for you already)
   (f) Determine the desired joint velocity (stored in the `qd_des map`)
   (g) Compute the *velocity error*
   (h) Compute the torque to apply to the joint via the *PD control law*.

4. **Re-build the controller and simulate the robot** again using Step 1.

5. **Make a movie of the robot executing the sinusoidal motion**. Once you are satisfied that your robot is controlled well, we will make a movie of it executing the sinusoidal motion:

   ```
   drum:~/motion-control-manipulator$ moby-driver -r -mt=10 -p=./debug/libsinusoidal-
       controller.so -v=10 -y=osg ur10.xml
   ```

   and following the step labeled "Render the OpenSceneGraph files" on this page: `https://github.com/PositronicsLab/Moby/wiki/Rendering-videos`. *You'll need to futz with the camera position and camera target to make the video look good*, though you can start with the settings in `ur10.xml`.

6. We now want to have the robot perform a different task: maintaining its initial configuration. **Implement a PD controller** for keeping the robot stationary in `resting-controller.cpp`. The procedure will be almost identical to that described in Step (3).

7. **Re-build and simulate the robot with the resting controller**:

   ```
   drum:~/motion-control-manipulation$ moby-driver -r -mt=2 -p=./debug/libresting-
       controller.so ur10.xml
   ```

8. **Plot the control performance**. The plugin generates data of the joint angles over time. We now use `matplotlib` to generate a plot of the shoulder "lift" joint data like so:

   ```
   drum:~/motion-control-manipulator$ python plot_PD_PID.py shoulder_lift_joint
   ```

   This program with this particular argument generates the file `shoulder_lift_joint.png`. Performance for different joints (those listed in `gains.dat` can be plotted by replacing "shoulder_lift_joint" with the desired joint.

9. **Tune control gains as necessary** by repeating the last two steps until performance is satisfactory *across all eight joints*.

10. **Label rise time, peak time, settling time, overshoot, and the error band** by editing `plot_PD_PID.py` and regenerating the eight plots.

11. Now, **implement a <u>PID</u> controller** for keeping the robot stationary by editing `resting-controller-PID.cpp`. This means that—in addition to the PD controls above—you want to accumulate the positional error on every controller iteration and use that to compute the motor torque. I recommend using a *static variable* inside your control function for the accumulator. See this thread (`http://stackoverflow.com/questions/6223355/static-variables-in-class-methods`) for more information.

12. **Re-build the controller and simulate the robot again**, this time using the command:

    ```
    drum:~/motion-control-manipulator$ moby-driver -r -p=./debug/resting-controller-
        PID.so ur10.xml
    ```

13. **Plot the control performance again**. Generate the eight plots as before; I expect performance will be better when the "integral" portion of the controller is included.

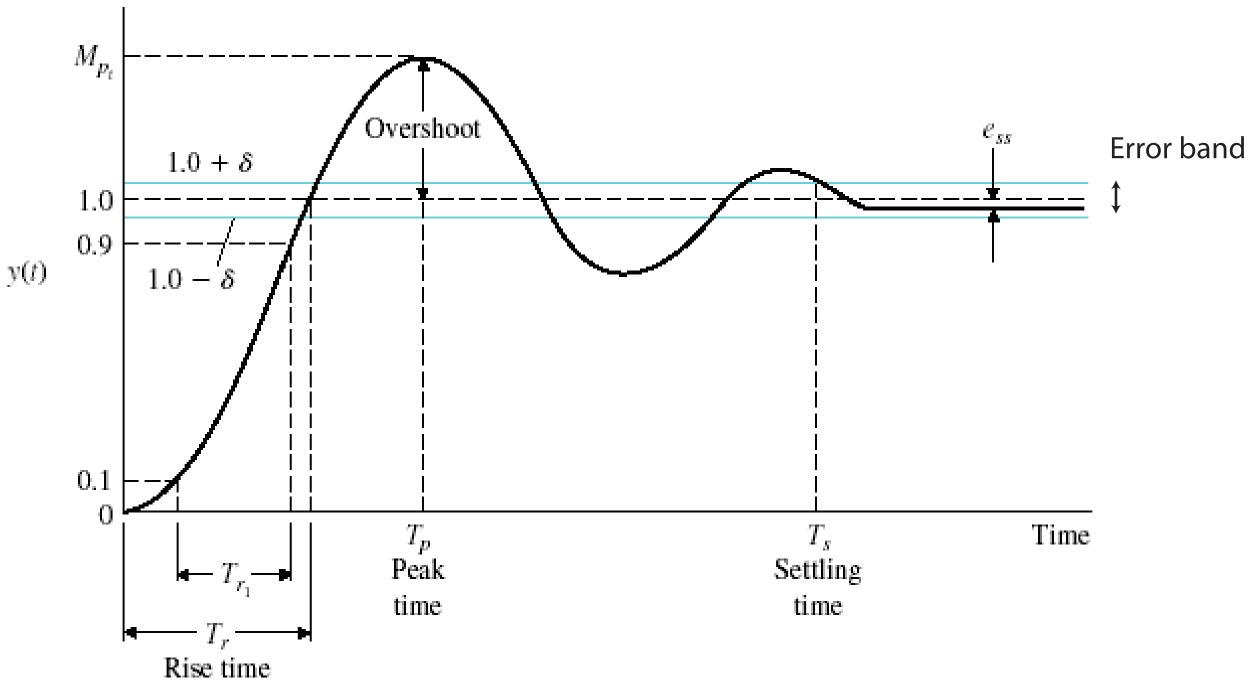

Figure 1: Depiction of rise time, peak time, overshoot, settling time, and error band for a feedback controller attempting to maintain a set point of 1.0.