EECS 106B Lab 3: Nonholonmic control with Turtlebots *

Due date: Monday, April 8th at 11:59pm

Goal

Constrained Control with Turtlebots

You will develop controllers for a second order nonholonomic systems (eg cars) using two methods. First you'll perform control directly on its its Lie algebra to gain a deeper understanding of Lie brackets. Then you'll utilize the steering with sinusoids method to do more efficient open loop control.

Contents

*Developed by Valmik Prabhu and Chris Correa, Spring 2019

1 Theory

For this lab, you'll be running a script that makes the turtlebot (normally a unicycle-model robot) behave like a bicycle model.

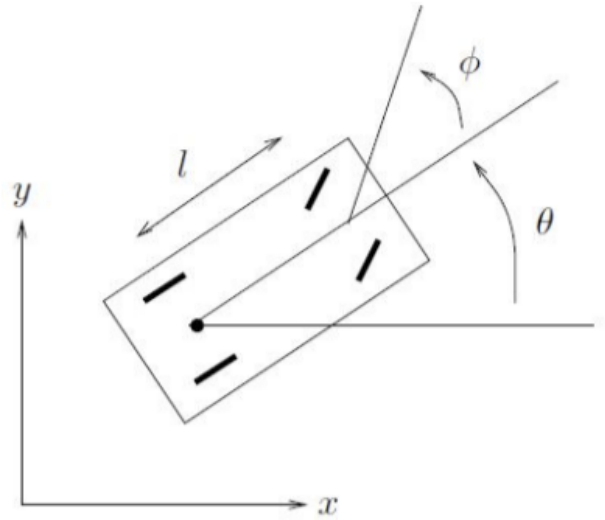

Figure 1: Dynamics of the Bicycle Model

As you've seen in the homework, the bicycle model has the following dynamics:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ \frac{1}{l}\tan(\phi) \\ 0 \end{bmatrix} u_1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} u_2 \tag{1}$$

where u_1 is the car velocity, and u_2 the steering rate. As you can see the two input vector fields g_1 and g_2, where

$$g_1 = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ \frac{1}{l}\tan(\phi) \\ 0 \end{bmatrix} \tag{2}$$

and

$$g_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3}$$

do not span the whole state space $\mathbb{R}^4$. In order to examine the steerability of the system, we examine the Lie algebra, which is found by taking successive Lie Brackets of the input vector fields. The Lie bracket represents the second order terms that arise from alternating forward and reverse control of two inputs, and can be calulated through

$$g_3 = [g_1, g_2] = \frac{\partial g_2}{\partial x} g_1 - \frac{\partial g_1}{\partial x} g_2 \tag{4}$$

$$g_4 = [g_1, g_3] \tag{5}$$

$$g_5 = [g_2, g_3] \tag{6}$$

and so on. See homework four for more information. Working out the Lie Brackets shows that g_3 controls θ, and g_4 controls x and y perpendicular to the direction the car is facing. Both of these directions are constrained by velocity constraints, but by switching between steering left and right and driving forwards and backwards, you can generate motion in these directions.

1.1 Bang-Bang Control

Bang-bang control is a control strategy wherein you alternate between maximizing and minimizing controller input. It's often implemented because it's very easy and surprisingly optimal for several classes of systems. We'll be doing something similar, but not the same as true bang-bang control. Rather than saturating every input at once, we'll be alternating between running u_1 and u_2, but freely setting the magnitude rather than simply saturating the actuators. Using `bangbang.py` for inspiration, you'll be designing a full state feedback controller using the Lie Algebra of the system as your inputs. As the bicycle model is drift-free, a proportional controller should be sufficent, but you may need to add additional terms as you see fit. Given some error term $e = q - \hat{q}$, you'll be responsible for picking which order to control the Lie brackets and how to implement the proportional term (by increasing the magnitude of the inputs, the number of motions, and/or the speed of each maneuver). Note that you can theoretically control a number of modes simultaneously by superposition. However, I'm not sure how well this would work in practice, and this would be an interesting experiment for you to try.

1.2 Steering with Sinusoids

You'll also be implementing the steering with sinusoids algorithm from this paper by Sastry and Murray. They prove that a good way to control nonholonomic systems is, rather than using a bang-bang "square-wave" approach, to control the inputs with out-of-phase sinusoids. To do this, they require that the system dynamics approximate a cannonical model

$$\begin{aligned}\dot{x}_1 &= u_1\\ \dot{x}_2 &= u_2\\ \dot{x}_3 &= x_2 u_1\\ \dot{x}_4 &= x_3 u_1\end{aligned}$$

Our car model does not match this cannonical form, but can be made to approximate it by applying a nonlinear transformation

Cannonical Car Model

$$\begin{aligned}\dot{x} &= v_1, & v_1 &= \cos(\theta)u_1\\ \dot{\phi} &= v_2, & v_2 &= u_2\\ \dot{\alpha} &= \frac{1}{l}\tan(\phi)v_1, & \alpha &= \sin(\theta)\\ \dot{y} &= \frac{\alpha}{\sqrt{1-\alpha^2}}v_1\end{aligned}$$

As you can see, this model approximately matches the cannonical form, with

$$\begin{aligned}\dot{x}_1 &= u_1\\ \dot{x}_2 &= u_2\\ \dot{x}_3 &= f(x_2)u_1\\ \dot{x}_4 &= g(x_3)u_1\end{aligned}$$

Even though $\dot{x}_3$ and $\dot{x}_4$ have nonlinear terms, we can use Fourier analysis to get rid of them. Another problem is not so easy to get rid of; this model contains a singularity at $\theta \in \{90°, -90°\}$. When the robot is turned 90°, we know that all the input velocity u_1 goes in the y direction, and everything works normally. However, when that occurs in this model, u_1 goes to zero, $\dot{\alpha}$ goes to zero, and and $\dot{y}$ goes to $\frac{0}{0}$. While this last one could potentially be resolved through L'Hopital's rule, this is still not ideal. One potential solution is to match the cannonical model in a different way, by replacing x with y as the first cannonical state. The model would look something like this:

Alternate Model

$$\begin{aligned}\dot{y} &= v_1, & v_1 &= \sin(\theta)u_1\\ \dot{\phi} &= v_2, & v_2 &= u_2\\ \dot{\alpha} &= -\frac{1}{l}\tan(\phi)v_1, & \alpha &= \cos(\theta)\\ \dot{x} &= \frac{\alpha}{\sqrt{1-\alpha^2}}v_1\end{aligned}$$

(*Note: if you choose to use this model, please rederive it and double check for sign errors*)
This is also an approximation of the cannonical system, except this system is undefined at $\theta \in \{0°, 180°\}$. You could simply switch models depending on where θ happens to be, perhaps even within one plan. Another thing you could do is ignore the cannonical form and use $\dot{\theta} = \frac{1}{l}\tan(\phi)u_1$ for planning changes to θ (but not y), because as you'll see, x is unnecessary in computing an open loop control law for θ.

1.2.1 The Algorithm

The algorithm for developing an open-loop path for steering is deceptively simple, but contains a couple very tricky bits. It's conducted in three stages:

1. First, steer the x_i's, which in this case are x and ϕ to their desired states. This is reasonably simple, and requires choosing $v_i = (x_{i_d} - x_i)/\Delta t_d$, where Δt_d is the desired time to complete the maneuver.

2. Second, steer x_{ij}, which in this case is θ or α to its desired state. To do this, set

$$v_1 = a_1 \sin(\omega t)$$
$$v_2 = a_2 \cos(\omega t)$$

The maneuver will run for $\frac{2\pi}{\omega}$ seconds (one period), and its magnitude will be determined by a_1 and a_2. The tricky bit comes when trying to select a_1 and a_2 to move some desired magnitude in θ. If the system followed the standard cannonical form, the difference would simply be $x_{ij}(\frac{2\pi}{\omega}) = x_{ij}(0) - \frac{a_1 a_2}{\omega}$. However, since our model is nonlinear, this doesn't work. To find the change in α, we first look at the change in ϕ. Since $\dot{\phi} = a_2 \cos(\omega t)$ we know

$$\phi(t) = \phi(0) + \frac{a_2}{\omega}\sin(\omega t) \tag{7}$$

Note that $\phi(\frac{2\pi}{\omega}) = \phi_0$, which demonstrates that these sinusoids don't change x_i, only x_{ij}. Thus,

$$\dot{\alpha}(t) = f(\phi(t))v_1(t) = f\left(\frac{a_2}{\omega}\sin(\omega t)\right) a_1 \sin(\omega t) \tag{8}$$

We could find $\alpha(\frac{2\pi}{\omega})$ by integrating, but this integral is very difficult to compute. One way to (slightly) simplify it is to use a Fourier expansion.

As you likely learned in your lower division math classes, periodic functions can be represented as a summation of sinusoids called the Fourier Sine Series. By taking this expansion, we approximate

$$f\left(\frac{a_2}{\omega}\sin(\omega t)\right) = \beta_1 \sin(\omega t) + \beta_2 \sin(2\omega t) \cdots \tag{9}$$

Which means that our integral becomes

$$\alpha(\frac{2\pi}{\omega}) = \alpha(0) + \int_0^{\frac{2\pi}{\omega}} \beta_1 \sin(\omega t) a_1 \sin(\omega t) + \beta_2 \sin(2\omega t) a_1 \sin(\omega t) \cdots dt \tag{10}$$

Now, we use the property that sinusoids of integrally related frequencies are mutually orthogonal over their period. Two functions $f(x)$ and $g(x)$ are orthogonal on $x \in [a, b]$ if $\int_a^b f(x)g(x)dx = 0$. The fact that the terms of the Fourier sine series are orthogonal make sense since the sinusoids essentially form a basis with weights β. By using this property, all but the first term in the integral disappears, leaving us with

$$\alpha(\frac{2\pi}{\omega}) = \alpha(0) + \int_0^{\frac{2\pi}{\omega}} \beta_1 a_1 \sin(\omega t)^2 dt = \alpha(0) + \frac{\pi a_1 \beta_1}{\omega} \tag{11}$$

This is nice and easy. Of course, β_1 still requires an integral to compute:

$$\beta_1 = \frac{\omega}{\pi}\int_0^{\frac{2\pi}{\omega}} f\left(\frac{a_2}{\omega}\sin(\omega t)\right)\sin(\omega t)dt \tag{12}$$

You'll have to compute this integral numerically and find an a_1 and a_2 that result in a desired translation of x_{ij}. You may have to implement binary search for this, or some other method to find these values. Also note that it may be easier to numerically integrate α directly, rather than numerically finding β_1. However, this may make it much more computationally intensive to find a_1 and a_2.

3. The third step is to use integrally related sinusoids to control x_{ijk}, in this case y, without changing any of the other variables. Here, we choose

$$v_1 = a_1 \sin(\omega t)$$
$$v_2 = a_2 \cos(2\omega t)$$

Once again, we have to deal with a very hairy integral in order to figure out $y(\frac{2\pi}{\omega})$. We want to find

$$y(\frac{2\pi}{\omega}) = y(0) + \int_0^{2\pi} \omega g(\alpha(t)) a_1 \sin(\omega t) dt \tag{13}$$

Where

$$\alpha(t) = \int_0^t f(\frac{a_2}{\omega} \sin(2\omega\tau)) a_1 \sin(\omega\tau) d\tau \tag{14}$$

We use the Fourier Sine Series to get

$$y(\frac{2\pi}{\omega}) = y(0) + \int_0^{2\pi} \omega\beta_1 \sin(\omega t) a_1 \sin(\omega t) + \beta_2 \sin(2\omega t) a_1 \sin(\omega t) \cdots dt \tag{15}$$

Which becomes

$$y(\frac{2\pi}{\omega}) = y(0) + \frac{\pi a_1 \beta_1}{\omega} \tag{16}$$

where

$$\beta_1 = \frac{\omega}{\pi} \int_0^{\frac{2\pi}{\omega}} g\left(\alpha(t)\right) \sin(\omega t) dt = \frac{\omega}{\pi} \int_0^{\frac{2\pi}{\omega}} g\left(\int_0^t f(\frac{a_2}{2\omega} \sin(2\omega\tau)) a_1 \sin(\omega\tau) d\tau \right) \sin(\omega t) dt \tag{17}$$

You can expand this out, but it won't be very pretty. You'll still need to do quite a lot of numerical integration. In fact, you'll assuredly have to do binary search here, since β_1 depends on a_1 as well as a_2. Previously, you could set an arbitrary a_2, find β_1, and then set a_1 to produce the desired displacement. Now you'll need to do binary search on a_1 to find the correct combination of a_1 and β_1 in order to generate the desired displacement.

While finding a_1 and a_2 for general constraints is particularly difficult, there are ways to simplify the model you use for certain conditions. Furthermore, I suggest that you use a heuristic method to get things somewhat working before implementing all the integrals.

1.3 Tracking Nonholonomic Open Loop Trajectories

Creating these open-loop trajectories is all well and good, but as you've seen, executing trajectories in the real world never works perfectly. Thus, in order to properly execute trajectories provided by the above planner, we need to add feedback control. Grad students (and interested undergrads) will need to implement a feedback controller based on this paper. The control design is reasonably simple to implement but requires a significant amount of computation. When given an open-loop trajectory (a series of waypoints in time containing the predicted state $x^0(t)$ and the desired input $u^0(t)$ at each time t), the controller uses the linearized system at each waypoint. We define

$$A(t) = \frac{\partial}{\partial x} f(x^0(t), u^0(t))$$
$$B(t) = \frac{\partial}{\partial u} f(x^0(t), u^0(t))$$

We've now created a linear time varying system along the open-loop trajectory.
To be continued

2 Logistics and Lab Safety

We expect that all students in this class are mature and experienced at working with hardware, so we give you much more leeway than in EECS 106A. Please live up to our expectations.

2.1 Groups, Robots and Accounts

Remember that groups must be composed of 2-3 students, at least one of whom must have completed EECS 106A. Robots should be reserved on the robot calendar here. The rules are

1. **Groups with a reservation have priority.** You can go in and use (or continue using) the hardware whenever you like if no one has a reservation. However, you must pass along the hardware once the next reservation starts. Please be respectful and try to plan ahead; it's not cool to take the first twenty minutes of another group's section winding down.
2. **Do not reserve more than two hours at a time.** This is shared hardware and we want to ensure that all groups have enough time to complete the lab.
3. **One computer per group (if needed)** We only have ten lab computers, and have around thirty students. If more than ten students want to use the computers, please try to limit yourself to one computer per group. In addition, groups with robot reservations have priority on the computers next to their respective robots. Groups can accomplish working in parallel by using personal computers, file-sharing software like git, remoting into the lab computers, and/or using simulators like Gazebo or RViz.
4. **Please remember to charge the robots after use**. That way others will be able to use the bot for all of their time slot.

2.2 Safety

Remember to follow the lab safety rules:

1. **Never work on hardware alone.** Robots are potentially dangerous and very expensive, so you should always have someone on hand to help if something goes wrong. This person should be another 106B student, though in cases when group schedules are highly incompatible, you may talk to a TA to arrange an alternate solution, such as bringing a friend (note that you would be entirely responsibe for this person's conduct).
2. **Do not leave the room while the robot is running.** Running robots must be under constant observation.
3. **Always operate the robot with the E-Stop within reach.** If Baxter is going to hit a person, the wall, or a table, press the E-Stop.
4. **Power off the robot when leaving.** Unless you're trading off with another group, the robots should be powered down for safety. To power on/off the robot, press the white button on the back of the robot. For turtlebots, cycle power (to kill any remaining processes), then power on and plug into a charger. Turtlebots cannot charge when off.
5. **Terminate all processes before logging off.** Leaving processes running can disrupt other students, is highly inconsiderate, and is difficult to debug. Instead of logging off, you should type

   ```
   killall -u <username>
   ```

   into your terminal, where `<username>` is your instructional account username (`ee106b-xyz`). You can also use `ps -ef | grep ros` to check your currently running processes.
6. **Do not modify robots without consulting lab staff.** Last semester we had problems with students losing gripper pieces and messing with turtlebots. This is inconsiderate and makes the TAs' lives much more difficult.
7. **Tell the course staff if you break something.** This is an instructional lab, and we expect a certain amount of wear and tear to occur. However, if we don't know something is broken, we cannot fix it.

2.3 Questions and Help

If you experience software-related errors, please perform the following:

1. Document exactly what you did leading up to the error (commands, terminal output, etc.)
2. Post on Piazza with a description of the error, the above materials, and a description of what you did to try and fix it.

Chris and Valmik **Will Not** diagnose any programming errors without the above documentation. However, feel free to ask us theoretical questions on piazza, during office hours, or after lecture or discussion.

3 Project Tasks

You'll be using the Turtlebots in Cory 111.
For this lab, you'll write two types of controllers to execute nonholonomic motions

1. **Direct control of the Lie algebra** Develop your own velocity controller based on the Lie algebra to execute the following maneuvers. For each maneuver, the robot should start at the state $[x, y, \theta, \phi] = [0, 0, 0, 0]$.
   (a) Translation in x: Your final displacement should be $[1, 0, 0, 0]$.
   (b) Translation in y (parallel parking): Your final displacement should be $[0, 0.5, 0, 0]$.
   (c) Translation in y (parallel parking): Your final displacement should be $[0, 1, 0, 0]$.
   (d) Yaw rotation (u turn): Your final displacement should be $[0, 0, 180°, 0]$
   (e) Translating in x and y (s turn): Your final displacement should be $[0.5, 0.5, 0, 0]$
2. **Steering with Sinusoids** Implement the steering with sinusoids algorithm from this paper to generate a path (set of waypoints parameterized in time, containing both inputs and expected positions) which accomplishes each of the five maneuvers above. Then execute them on the robot. *Extra for grad students:* Grad students will additionally implement a closed-loop feedback controller when executing this trajectory, based on this paper. Watch out! There are state and input bounds on the turtlebot, so ensure that your oscillations don't get too big.

4 Deliverables

To demonstrate that your implementation works, deliver the following:

1. Videos of your implementation working. Provide a link to your video in your report. You don't need a video of each run of each path, but we should see from your videos that your implementation works on hardware. Also, please combine all videos into one video file on upload.
2. Provide the code by adding "berkeley-ee106b" as a collaborator to a *private* github repo.
3. Submit a detailed report with the following:
   (a) Describe your approach and methods in detail, including a formal description of the controllers you used.
   (b) Let us know what of any difficulties and how you overcame them.
   (c) Summarize your results in both words and with figures. In particular, show:
   - For each maneuver, plot the true Turtlebot state (x, y, θ, ϕ) reached with your controller versus the desired state for each control task, as a function of time. For the Lie algebra control, your desired state will be a constant (step function), and your true state will be your measured state deuring execution. For the steering with sinusoids, the desired state is the predicted position at each step of the path $x^0(t)$, while your actual state is your measured state during execution.
   - For each maneuver, plot the desired and true (x, y) states of the turtlebot against one another on the same plot. These plots should look like the upper right plot of Figure 7 of the Sastry Murray paper.
   - Analyze and compare the performance of both control methods. How accurate is each method? How fast? Is one method more optimal than the others? Can you think of something more optimal that you'd do in the real world?

   (d) Go to google scholar and look up *Nonholonomic Motion Planning: Steering with Sinusoids*. As you can see, it's been cited thousands of times, and is still fairly heavily cited today. Read and summarize 2 papers written in the last 3 years which cite *Steering with Sinusoids*. How has *Steering with Sinusoids* contributed to these papers?
4. **BONUS:** In addition to your report, write a couple paragraphs detailing how the lab documentation and starter code could be improved. This course is evolving quickly, and we're always looking for feedback. This should be a separate section in the report, and only well-considered, thoughtful feedback will merit full credit.

5 Getting Started

We assume that you've already completed all the configuration steps from labs 0, 1, and 2. There are some different steps for lab 3.

5.1 Configuration and Workspace Setup

Note: *we expect all students to already be familiar with configuration procedures, either through EECS 106A, Lab 0, or some other place. Therefore, we won't be providing any explanations in this section. Please consult Lab 0 if you require a refresher*
Type the following commands:

```
cd ~/ros_workspaces
mkdir ~/ros_workspaces/lab3_ws
mkdir ~/ros_workspaces/lab3_ws/src
cd ~/ros_workspaces/lab3_ws/src
catkin_init_workspace
cd src
git clone https://github.com/chriscorrea14/lab3_pkg
cd ..
catkin_make
source devel/setup.bash
cd src/lab3_pkg
python setup.py develop --user
```

Note, you may not have to run the last line, as we think we setup `catkin_make` to automatically run the `setup.py`. Please let us know if we succeeded.

5.2 Working With Robots

For this lab, you'll be working with Turtlebots. There are six turtlebots: Red, Blue, and Yellow, Green, Pink, and Black. To start, you must power the turtlebot base. Once the robot is powered on, check connectivity by running

```
ping <color>.local
```

Where `color` is either `red`, `blue`, `yellow`, `green`, `pink`, or `black`. Now reconfigure your ROS network settings to set the Turtlebot as your ROS master, rather than your computer. Open your `~\.bashrc` file and add the line

```
export ROS_MASTER_URI=http://<color>.local:11311
```

Once you've made this change, you'll need to re-source your `.bashrc` file in any open terminal windows that you plan to use this lab. (Any terminal windows you open after making this change will automatically incorporate it, since they run `.bashrc` on launch.)
When you're done working with the turtlebots, you should remove this command from your `.bashrc` file. If you don't you won't be able to run a `roscore` on your home computer properly. Note that reverting this change will only take effect after logging out of your computer. To work on other things immediately, type

```
export ROS_MASTER_URI=http://localhost:11311
```

in all terminal windows you plan to use.
Now, ssh into the Turtlebot from any lab workstation:

```
ssh turtlebot@<color>.local
```

The password is `EE106A18`. Then run

```
roslaunch turtlebot_bringup minimal.launch --screen
```

The TurtleBot is now launched, along with all of its sensors, and it is ready to receive motion commands. When you're done using a turtlebot, you can close the ssh connection by typing `exit` in the command line.
TurtleBot commands are sent over the topic `cmd_vel`. Later, you'll be using a conversion function to make the turtlebot behave like a car, and therefore you'll be sending commands to a different topic, but for now, just use the built-in keyboard teleoperation node. Open a new terminal window and `ssh` into the TurtleBot (keep the minimal.launch running). Then run the following:

```
roslaunch turtlebot_teleop keyboard_teleop.launch --screen
```

Then you can control the Turtlebot using the keyboard (the controls will be shown on the screen).
We have provided a launch file that will open all the necessary things to turn the turtlebot / turtlesim from a unicycle model to a bicycle model. To do this run one of the following to run on the turtlesim or turtlebot (respectively):

```
roslaunch lab3_pkg bicycle_converter.launch
roslaunch lab3_pkg bicycle_converter.launch turtlesim:=false
```

Launch files are very useful ways to combine lots of ROS commands together without opening a ton of terminal files. You should look inside to see how it works.

5.3 Starter Code

- `src/lab3/converter/bicycle_converter.py` You should look through this file to make sure you know how it works, but hopefully you won't have to modify it at all. It subscribes to the `/bicycle/cmd_vel` topic which listens for `BicycleCommandMsg` messages. It publishes the bicycle state to the `/bicycle/state` topic as a `BicycleStateMsg` message. If you're confused at what is in these messages look at `lab3.msg`.
- `scripts/main.py` This first calls a service in BicycleConverter to reset the turtlebot / turtlesim position, and update the internal state representation to 0 (note that turtlesim starts at $(x, y) = (5.5, 5.5)$ instead of $(0, 0)$. Make sure your code is robust to this). You should then create a planner which creates trajectories and an executor which executes trajectories.
- `src/lab3/planners/sinusoid_planner.py` The bulk of your efforts will go into this file. You should spend lots of time reading the steering with sinusoids paper. The high level goal of this class is to take in a goal pose, and then use the algorithm described in the paper to determine a trajectory of `BicycleCommandMsg`'s to steer the bicycle to the goal pose. You should create individual trajetories to steer x, ϕ, α, and y separately, then combine the trajectories together.

6 Scoring

Table 1: Point Allocation for Lab 3

Section	Points
Video	3
Code	3
Methods	10
Results: Control on the Lie Algebra	10
Results: Sinusodal Steering	10
Results: Closed Loop Path Following	10*
Discussion of Results	10
Paper Summaries	4
Bonus Difficulties section	5*

Summing all this up, this mini-project will be out of 50 points, with an additional 15 points possible. For grad students, the mini-project will be out of 60 points, with an additional 5 points possible.

7 Submission

You'll submit your writeup(s) on Gradescope and your code by adding "berkeley-ee106b" as a collaborator to a github repo. Your code will be checked for plagiarism, so please submit your own work. Only one submission is needed per group. Please add your groupmates as collaborators on both Github and Gradescope.

8 Improvements

If you notice any typos or things in this document or the starter code which you think we should change, please let us know here. Next year's students will thank you.