Installation

Run:

```
pip install -e .
```

Assignment 1.5 - PID, iLQR, and MPC

In this task, you have to implement three types of controllers: PID, iLQR, and iLQR+MPC.

First, try to run the program:

```
python balloon.py --human
```

You will see a game in which you can control a drone with WASD keys (W/S - thrust increase/decrease, A/D - pitch left/right). Your task is to pop as many balloons as you can in the limited time.

If you want to run an experiment faster, just add `--fast` flag to the command line. Note that in `fast` mode you can't control the drone yourself.

PID

To add PID player, add a `--pid` flag:

```
python balloon.py --pid
```

Propotional Integral Derivative (PID) controller is a generic error-based model-free controller, which works by producing controlling signal as a function of the error. This function consists of three blocks: proportional (`P * error_t`), integral (`I * sum_i=0^t(error_i)`), and derivative (`D * (error_t - error_[t-1})`). Constants `P` , `I` , and `D` are tuned separately (e.g. manually), look up PID tuning for details.

In this assignment, you have to implement PID controller in `pid_player.py:(class PID)` and a cascade PID controller in `pid_player.py:(class PIDPlayer)` . The first part is fairly straightforward, while the second needs some explanation.

The drone control signal has two components - `T_left` and `T_right` (T stands for thrust). These are basically the currents applied to left and right motors. The errors are formulated in terms of the drone's pose - position and orientation. Since our drone is 2-D, the position has `x` and `y` coordinates and the orientation is encoded by the angle `\theta` .

And now you see the problem. There are three error components, and controls are both dependant on multiple error components. How do we formulate the filters?

First, we reformulate the controls from `T_left` and `T_right` to `T_left - T_right` and `T_left + T_right`. This makes sense since the thrust difference affects the angle, while the sum affects the uplifting force.

Now, we can formulate the simple PID cascade.

```
Cascade 1:
(y - y_setpoint) -> PID -> y_speed_setpoint
(y_speed - y_speed_setpoint) -> PID -> T_sum
```

In the first cascade, we first compute a Y error, feed it to PID controller which outputs a setpoint for Y speed. Next, this setpoint is used to compute a Y speed error which is fed to the second PID controller that sets the sum of thrusts.

```
Cascade 2:
(x - x_setpoint) -> PID -> angle_setpoint
(angle - angle_setpoint) -> PID -> T_diff
```

In the second cascade, we first compute a X error, feed it to PID controller which outputs a setpoint for the angle. Next, this setpoint is used to compute an angle error which is fed to the second PID controller that sets the difference of thrusts.

iLQR

To add iLQR player, add a `--ilqr` flag:

```
python balloon.py --ilqr
```

Iterative Linear Quadratic Regulator is a model-based controller, which works with locally smooth state-action spaces. The linear version had been discussed in lectures. For iLQR see this article for reference and pseudocode. Also see this and this.

The task is to implement three pieces of logic in `ilqr_player.py:rollout, backward_pass, and forward_pass`. See `ilqr_player:run_ilqr` for details.

iLQR + MPC

To add MPC player, add a `--mpc` flag:

```
python balloon.py --mpc
```

Model Predictive Control (MPC) is a framework of applying a short-horizon controller in the following way. Let's say we have a controller `A` which output at the moment `t` computes a plan (sequence of actions) `U=(U_t, ... U_{t+T})` of length `T`. In case of MPC, only the first action (`U_t`) is then applied, after which the entire plan is recomputed. While simple, this strategy allows to adapt to modelling errors.

This part is the easiest for you. All you have to do is to implement `self.use_mpc` flag logic in `ilqr_player.py:iLQRPlayer.act`. If `self.use_mpc` is true, execute only the first action.

In order to evaluate the advantages of MPC, try increasing drone's mass. In this case, the dynamics model is not entirely correct:

```
python balloon.py --mpc --ilqr --fast --mass 2.0
```

Expect iLQR jumping around without making any progress, while iLQR+MPC performing well enough.

High score

Lastly, compare the high scores of each method. For instance, run:

```
python balloon.py --mpc --ilqr --pid --fast
```

Here's my score report:

```
PID collected : 32
iLQR collected : 75
iLQR_MPC collected : 92

Winner is : iLQR_MPC !
```

Try tuning parameters, modifying methods, etc. If your high score is better than mine on any of these methods - make sure to mention that in your submission!