Zip Sim Coding Challenge

Background

Zipline has an exciting new contract to start delivering medical supplies in the small but prosperous land of Zipago. Zipago is an unusual place, existing on the surface of a toroid in a parallel universe. It has a pleasant, temperate climate. Unsurprisingly, the company's existing flight control stack is completely unsuitable for operating a vehicle within this environment. You have been brought on to implement the new algorithms needed to fly the vehicle and deliver supplies.

Challenge

Write an "autopilot" program to fly a zipline vehicle and successfully deliver as many packages as possible while avoiding crashing. You may use whatever programming language you wish, as long as it is buildable and runnable on a modern linux, OS-X or windows 10 system via freely available tools.

This challenge is intended to be fun for you to implement and for us at Zipline to evaluate. There's no time limit, and the intent is for it to take only an hour or so to code up something basic, and no more than a few hours to come up with something unique to your style. Even if your solution isn't perfect, don't sweat it. Just send us whatever you came up with if it starts to feel like a chore.

External Requirements

These requirements must be met by your implementation in order for it to be considered ready for production.

- Double-delivery rate less than 1 in 1000 flights (ZIPAA requirement to avoid liability)
- Crash rate less than 1 in 100 flights (ZAA safety requirement for certification)
- Parachute rate less than 1 in 10 flights (Budget for replacement hardware in operation)
- Deliver at least 10% of packages on average (Contractual obligation with customer)

Internal Requirements

Zipline's company culture values individual and team ownership, transparency, continuous improvement, servant leadership, and an obsession with pleasing our customers. As the engineer doing the work, that basically means it's ultimately up to you to figure out what needs to be done and decide how best to do it!

Speaking specifically about customer obsession, that not only includes paying customers, but also all the people between you and those paying customers: healthcare workers, flight operators, manufacturing technicians, software release managers, code reviewers, and even yourself six months from now when you need to understand the terrible mess of code you wrote. Try to consider the needs of all of those customers when deciding how to best tackle this engineering problem.

It's fair to say though that the more reliable package delivery is, the more successful the company will be. The market is fiercely competitive, and while we don't have a good sense of our secretive competitors' capabilities, they're on our heels and ready to corner the market if we slip up.

ZAA Certification

ZAA is the Zipago Aviation Authority, the regulatory body responsible for keeping the airspace safe. Your completed source code must be submitted at the end of the activity for independent review by the ZAA. Please take into consideration the fact that readable code is more likely to be approved than unreadable code.

Alongside the source code, you must submit directions on how to build and run your program, and a short document (1-2 paragraphs) explaining why you believe your implementation meets or exceeds the less than 1 in 100 crash rate requirement as specified by the ZAA.

ZIPAA Liability

ZIPAA is the Zipago Insurance Portability and Accountability Act, a law passed to protect the sensitive and private health information of the citizens of Zipago. If your autopilot leads to a potential ZIPAA violation, you'll be required to retroactively defend your engineering processes in a hearing, or face immediate dismissal and potential criminal legal action should you ever personally set foot in Zipago.

Independent Validation

The submitted solution will be run against the lesser of 1,000 simulated flights, or as many as can be run in 10 minutes.

Zip Sim Documentation

The python3 script zip_sim.py is provided to you as a tool for use during development and testing. When invoked without any arguments, it runs in a human-interactive mode with a randomly generated map. Human-interactive mode is controlled via the left and right arrow keys and the spacebar (to drop a package). The simulation may also be passed a command to run in

order to instantiate the autopilot. The autopilot program communicates with the simulation via stdin and stdout.

The simulation's random number generator may be seeded with --seed=<number>, allowing the same map to be re-simulated.

The simulation may be run faster than real-time without visualization via the --headless option.

When you have the visualization up and running, you can start and stop time by pressing the 'p' key. While paused, you can single-step the simulation by pressing 's'. You can also control how fast the simulation runs by pressing the ',' and '.' keys.

There are other potentially useful visualization options. Use the --help option to list them.

Example Shell Commands

These commands assume that a modern version of python 3 is installed and invokable via the command "python". For more information about virtual environments, visit https://docs.python.org/3/library/venv.html
The commands also assume a bash-like shell. For tips on how to work in a windows cmd shell, check out https://mothergeo-py.readthedocs.io/en/latest/development/how-to/venv-win.html

Value	Description
`python -m venv test_environment`	Make a python virtual environment to work out of
`source test_environment/bin/activate`	Activate the virtual environment
`pip install -r requirements.txt`	Install zip sim's package dependencies
`python zip_sim.py`	Play a random level
`python zip_sim.py a.out` `python zip_sim.py python my_pilot.py`	Run various autopilots

Simulation Characteristics

The simulated world of Zipago is 2000 meters long by 50 meters wide. It can be thought of as the 2-dimensional projected surface of a toroid; the world wraps around on itself, rather than terminating at boundaries. The long dimension by convention is the X-axis, and the narrow

dimension is the Y-axis. The vehicle always faces forward and moves in the positive X-axis, and the "port side" of the vehicle always points in the direction of the positive Y-axis.

There's wind that moves in the XY plane with varying speed and direction. The vehicle maintains a constant forward airspeed of 30 m/s, and it is able to precisely control its lateral (side-to-side) airspeed up to +/- 30 m/s. This means that the vehicle's speed relative to the ground varies with the wind.

The vehicle carries 10 packages for delivery. When a package is dropped, it moves along a ballistic trajectory corresponding to the vehicle's ground velocity at the time that the package was dropped. It takes the package 0.5 seconds to fall and hit the ground. The visualizer draws a reticle estimating where a package would land if dropped.

The vehicle launches and recovers at a distribution center. Launch occurs instantaneously at the start of the simulation. Recovery occurs by flying the vehicle through the designated recovery zone. If the vehicle misses the recovery zone, then the simulation still ends with the assumption that the vehicle had to land via backup parachute. Parachute landings are bad because they slow down operations and often cause damage that's expensive to repair.

The vehicle carries an onboard forward facing lidar sensor that can measure distances to objects within 255 meters of the vehicle. The lidar takes 31 samples, corresponding to a +/-15 degree sweep in 1 degree increments. (Note that the lidar can be visualized by passing in the --show-lidar option.)

There are 10 delivery sites, consisting of circles 10 meters in diameter. A package is successfully delivered if it lands within the radius of the circle. The delivery sites have 1 meter wide circular retro-reflectors in their center, making them highly visible to the lidar sensor. It doesn't matter what specific package gets delivered to each delivery site (the vehicle is smart enough to release the correct package), but it's bad to deliver multiple packages to the same delivery site because it leaks customer's personal health information in a way that violates ZIPAA.

There are some number of trees that are tall enough for the vehicle to collide with. The trees are periodically manicured to be a uniform size, and they are visible as 6 meter wide circles via lidar. The vehicle is robust enough to survive some light branch scraping, but getting too close will destroy it. Crashing into a tree is bad because, in addition to totally ruining the vehicle, it opens up a can of worms with the ZAA.

Simulation Results

At the end of the simulation, the number of successful deliveries and ZIPAA violations is displayed via stdout. The simulation process's exit code reflects the reason why the simulation ended.

Exit Code	Reason
0	The vehicle recovered successfully
1	The vehicle deployed its parachute
2	The vehicle crashed, either due to colliding with a tree or because the autopilot errored out
3	The simulation's visualization window was explicitly closed

Note that the exit code is arbitrary if the simulation was killed via an external signal (such as by pressing ctrl-C).

Autopilot API

The zip sim tool internally executes the specified command as a subprocess, and uses stdin and stdout to communicate with it via a binary protocol. The simulation sends a single TELEMETRY packet to the autopilot, and then waits for a single COMMAND packet in response. Communication continues back-and-forth in a ping-pong manner until the simulation ends or the autopilot unexpectedly exits. The simulation advances 1/60th of a second between rounds of communication. The autopilot process is expected to politely exit when the sim closes its pipe to the autopilot's stdin.

All values in the packets are in big-endian byte order.

TELEMETRY Packet (44 bytes)

Value	Description
uint16_t timestamp	The current simulation time in milliseconds
int16_t recovery_x_error	The distance away from the recovery system in the x axis, in meters.
float wind_vector_x	The x component of the current wind velocity vector, in meters per second.
float wind_vector_y	The y component of the current wind velocity vector, in meters per second.
int8_t recovery_y_error	The distance away from the recovery system in the y axis, in meters.

uint8_t lidar_samples[31]	The distance of lidar returns in a -15 degree to +15 degree sweep in 1 degree increments, in meters. 0 means the lidar sample was out of range.

COMMAND Packet (8 bytes)

Value	Description
float lateral_airspeed	The desired lateral airspeed, in meters per second. Limited to +/- 30m/s
uint8_t drop_package	Set to 1 to command a package to drop. The value must be commanded back to 0 before commanding a package to drop again.
uint8_t padding[3]	Three bytes of padding to make the packet a multiple of 4 bytes in size.