Welcome to Duckiesky

Robots are the decathlon of computer science: to make a robot work, you need to understand robotics, which we define as a program that includes a sensor and an actuator. Additionally though, you typically need to understand systems, because your robot will use multiple programs running on a computer to make its decisions; you need to understand networking to make the computers talk; you need to worry about algorithms to make efficient use of the computing resources and prove bounds on your robot's behavior; and you need to understand hardware, because hardware limits affect all aspects of the robot behavior, and if your CPU overheats, your robot isn't going anywhere.

This textbook contains assignments, projects, and technical material related to the Duckiedrone, a small autonomous drone. After taking this course, students will be able to:

- Explain the space of designs for robotic communications, safety, state estimation, and control.
- Apply that knowledge to construct programs for communications, safety, state estimation, and control.
- Build, program, and operate an autonomous robot drone.

Assignment 1: Introduction

This assignment gives an introduction to our course and reviews some basic material you will need. *Hand ins will be noted in italics. Create an answers.txt file in your GitHub repository (see handin instructions at the bottom of this page) in which to write your answers.*

Collaboration Policy

Please read and sign the collaboration_policy_for_CS1951R. *Submit the signed pdf with filename collaboration_policy.pdf.*

Safety Policy

Please read and sign the safety_policy_for_CS1951R. *Submit the signed pdf with filename safety_policy.pdf*

Motivations (20 points)

Submit the answers to these questions in answers.txt

Before you start putting a lot of time into this course, it is important to figure out what you will get out of the course. Think about what you expect to learn from this course and why it is worth investing a lot of time.

1. What is a robot?
2. If I can fly a drone by remote, what can I get out of programming it?

Matrices and Transformations (20 points)

Write the answers to these questions in the corresponding section of answers.txt.

Transformation matrices are fundamental to reasoning about sensors and actuators. For example, the robot might detect a landmark with its camera, and we might want to know the location of the landmark relative to the robot's base. Or we might want to know where we can expect the landmark to be located after the robot has moved forward. We will cover this in detail but for know are asking you to do a warmup on these topics.

For this problem we strongly recommend you do these calculations by hand, because they are warmup questions designed to remind you of some of the prerequisite material for the class.

1. Multiply the matrix by the following vector:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

2. Multiply the matrix by the following vector:

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 4 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{bmatrix}$$

3. Imagine a robot is located in the (x, y) coordinate plane at the origin $(0, 0)$. It uses a sensor to detect an obstacle at a distance of $6m$ and a heading of 30°. The positive y-axis represents 0°, and the degrees increase when turning clockwise. What are the (x, y) coordinates of the obstacle? Give the coordinates in *answers.txt*. Then draw your answer on a map and add it to your repo as *map.png*.

Law of Leaky Abstractions (20 points)

Write your answers in the corresponding section of answers.txt

Read The Law of Leaky Abstractions. How might this be especially relevant to robotics? Make sure you address:

1. Give an example of a system that you have worked with that had an abstraction that "leaked." Describe the abstraction, what it was hiding, and what went wrong so that the abstraction broke down.
2. How can we use abstractions in light of these challenges?

Handin

If you do not have a GitHub account, please create one at this link. We will be using git repos throughout the course for versioning, moving code around, and submitting assignments.

Once you have a GitHub account, click on this link to join our GitHub classroom. This should ask you to select your name from the list and create a repository for you. Clone the directory to your computer

```
git clone https://github.com/h2r/assignment-introduction-yourGithubName.git
```

This will create a new folder. Before you submit your assignment, your folder should contain

- `collaboration_policy.pdf`
- `safety_policy.pdf`
- `answers.txt`
- `map.png`

Commit and push your changes before the assignment is due. This will allow us to access the files you pushed to GitHub and grade them accordingly. If you commit and push after the assignment deadline, we will use your latest commit as your final submission, and you will be marked late.

```
cd assignment-introduction-yourGitHubName
git add collaboration_policy.pdf safety_policy.pdf answers.txt map.png
git commit -a -m 'some commit message. maybe handin, maybe update'
git push
```

Note that assignments will be graded anonymously, so don't put your name or any other identifying information on the files you hand in.

Project 1: Building Your Drone

The build instructions for the drone can be found here.

Assignment 2: Safety

This unit asks you to think about safety considerations with robotics. For some of these questions, there is no one correct answer; however we will publish our answers, and our grading rubric will allow for multiple answers.

Safety is one of the most important considerations in robotics. Imagine that someone throws the drone at a person as hard as they can. This sort of motion is what the robot is capable of doing using its motors and accelerating at top speed. It is extremely important that whenever you fly your drone or operate any robot, that you keep yourself and the people around you safe. Safety is everyone's responsibility!

You are responsible for operating your drone in a safe manner. The most important safety advice we can give is that each person is responsible for the safe operation. This includes speaking up if you see an unsafe situation, acquiring information if you do not know if something is safe, and taking care of yourself. (For example, don't operate your drone when you haven't slept enough.)

Assignment

The goal of this assignment is to ask you to think critically about how to ensure robots are operated safely, and to devise guidelines for operating your robot safely.

OSHA Safety Analysis (50 points)

Write your answers in `answers.txt`

Read the OSHA Technical Manual on Industrial Robots and Robot System Safety.

Perform a hazard analysis for the drone, based on the OSHA guidelines. Make sure you answer each of the following subquestions in a few sentences (in your own words).

1. What tasks will the robot be programmed to perform?
2. What are the startup, command, or programming procedures?
3. What environmental conditions are relevant?
4. What are location/installation requirements to fly the drone?
5. What are possible human errors?
6. What maintenance is necessary?
7. What are possible robot and system malfunctions?
8. What is the normal mode of operation?

FAA Rules (20 points)

Write the answers to these questions in the corresponding sections in `answers.txt`

In the United States, the Federal Aviation Administration regulates outdoor flight. (It does not regulate flight indoors.) Read the FAA website on Unmanned Aircraft Systems. Take the TRUST test from any of the providers listed on the website. If you wish to fly your drone outside, you may also register your drone with the FAA for $5 (not covered by us). You do not need to fly outside for this class; if you only fly inside, you do not need to register your drone.

Provide short answers to the following questions.

1. What procedures should you follow when flying your drone outside the CIT? (You might find it easiest to use the B4UFLY Smartphone App).
2. What is the closest airport to the CIT? Hint: Make sure to check for heliports as well.
3. What are some risks of drone flight? How could people get hurt with the robot?
4. When do you need to report an accident to the FAA?

Brown Rules/Community Guidelines (5 points)

The FAA requires you to follow all community guidelines for flying drones. Brown University has an Unmanned Aircraft System (UAS) Policy that we must follow to fly on campus. Flying is already approved in our lab space, but must be approved to fly in your dorm room or other places on campus.

Provide short answers to the following question:

1. If you want to fly a drone outside the CIT in the quad, what procedures should you follow, with respect to Brown's community guidelines?

Flying at Home (5 points)

Write your answers in `answers.txt`.

Answer the following questions about flying at home over the duration of this semester.

We understand that students may be subject to different laws pertaining to how they fly their drone, depending on where they live. We would like you to look into the local laws so you can fly your drone legally.

1. Are there any region-specific rules in your area of residence that differs from the FAA rules? Can you fly indoors in your place of residence? Is there an area nearby that you may be able to fly outside? If so, does it need pre-approval?
2. What are the risks?
3. What should you do to plan?
4. What safety precautions should you take before you fly?

Soldering Station (10 points)

We added this question when running the class remotely. Even though we are now providing soldering space, it is still important to know the rules for soldering safely. Submit a photo of the lab if you will solder there, or the BDW, or wherever you will be building your drone.

Submit a photo with the filename `solder_station.jpg`.

It is important for you to have a safe, well-ventilated area where you will be able to solder your electronics.

Look for a designated area in your place of residence, or in the lab for you to soldier, and take a picture of that area clearly showing the following:

1. Nearby a window or a ventilation fan that can suck away the smoke from the soldering iron.
2. A fire extinguisher in the vicinity; safety first!
3. A wall plug or an extension cord with surge protection, to plug in the soldering iron and fan.
4. A table to solder on! If it has a cover, make sure it is non-flammable.

Flight Area (10 points)

Submit a photo with the filename `fly_area.jpg`.

In order to fly your drone (and you definitely will in this course!), you will to think about where to fly it safely. You may lose control of the drone and it might hit the ceiling or the wall, so it is best to plan for those possible failiure modes and get a large open space if possible.

Look for a designated area in your place of residence, and take a picture of that area clearly showing the following:

1. A ceiling that does not have a lot of attachments (like dangling lights, chandeliers etc that may fall and hurt you in cases of ceiling strikes).
2. A spacious area, at least 5ft by 5ft (or 1.5m by 1.5m).
3. An area where you can instruct people to keep a safe distance while flying your drone. For example, an area right in the middle of a pedestrian street may not be a good fit since young kids from your neighborhood may come running in at any time.

Handin

Use this link to access the assignment on Github classroom. Commit the files to hand in, as you did in the Introduction assignment.

Your handin should contain the following files:

- `answers.txt`
- `solder_station.jpg`
- `fly_area.jpg`

If you are uncomfortable with submitting photos of your place of residence, please reach out to the TAs so that we can confirm that you will be working in a safe, non-hazardous environment for the duration of our course.

Assignment 3: Linux and Networking

The networking component of this assignment will help you understand how to communicate with your drone. Fundamentally, robots are computers that are linked through networks. In robotics, accounting for networking allows both more robust and more efficient design. The networking part of this assignment describes how to use basic networking with a focus on concepts most useful to robotics.

Networking may not seem like a topic in robotics, but it is one of the most common reasons robots fail to work. If you cannot connect your base station to the robot, you cannot see the robot's status; you cannot see sensor output; you cannot send actuation commands. Moreover networks in the wild can be set up in a variety of diverse ways that may or may not allow your base station to connect to your robot. For example, Brown's default guest network does not allow peer-to-peer connections, so even if you get your base station and the robot connected on that network, you still cannot talk to the robot.

As a result, it is essential to be familiar with basic networking concepts in order to make your drone, or any robot, work. This unit asks you to think and learn about some networking concepts. We also cover helpful linux commands.

This assignment is comprised of two parts: Introduction to Linux (Part 1), Networking (Part 2). Please complete all parts of this assignment. You can do this assignment in the vscode shell on your drone, or on any machine with Python and the requisite shell commands.

Assignment

Part 1: Introduction to Linux

The learning objectives of this assignment are to familiarize you with basic Linux shell commands, standard input, standard output, standard error, and pipes. You will use these ideas when interacting with the Linux shell to operate your drone. Additionally you will use these ideas in the next section when working on the networking exercises.

Background Information

When you enter a command in a shell, it executes a program. These programs read from a stream, known as "standard input" and write to two output streams, "standard output" and "standard error". When you `print` in python, it writes its output to standard output. In another language, such as C, you use other functions, such as `printf` to write to standard output.

In addition to writing to standard output, a program can read from standard input. The program `cat`, short for concatenate, reads from standard input and writes the result to standard output.

Standard Output (10 points)

1. Write a python program that prints "Hello world" to standard output. *Save the program as `hello1.py` and submit it.*
2. Write a python program that prints "Hello world" to standard output using `sys.stdout`. *Save the program as `hello2.py` and submit it.*

3. Write a bash script that prints “Hello World” to standard output. *Save the script as `hello.sh` and submit it.*

Standard Input (10 points)

Write answers to questions 1-2 in `shell.txt`. Submit this file.

1. Run `cat` with no arguments. Why does `cat` seem like it is hanging?
2. When you run `cat`, type a message into your terminal, and press `Control-D`. Describe what `cat` does. Make sure to include which streams are being used, and for what purpose.
3. Write a python program `my_cat.py` that reads a message from standard input and prints to standard output, just as `cat` does. You only need to reproduce the behavior of `cat` when run with no arguments. In addition, you do not need to handle `Control-D`. *Submit this file.*

Pipes (20 points)

Pipes are used to redirect standard input, standard output, and standard error. First, `>` is used to redirect standard output to a file. For example, `echo "Hello World" > test.txt` will write the string `Hello World` to `test.txt`. *Write answers to questions 1-4 in `shell.txt`. Submit this file.*

1. Create files `one.txt`, `two.txt` and `three.txt` that contain the strings `1`, `2`, and `3`, respectively using `echo` and output redirect.
2. By convention, almost all shell programs read input from standard input, and write their output to standard output. Any error messages are printed to standard error. You can chain shell programs together by using `|`. For example, the program `ls` writes the contents of a directory to standard output. The program `sort` reads from standard input, sorts what it reads, and writes the sorted content to standard output. So you can use `ls | sort` to print out a sorted directory list. Read the man page for sort (`man sort`) to learn how to sort in reverse order. What is the bash script (using `|`) that prints the contents of a directory in reverse alphabetical order?
3. Use `cat`, `|` and `echo` to print `hello world.` Do not write to any files and use both commands one time.
4. This is not the simplest way to print hello world. Can you suggest a simpler way? (We asked you to do it the more complicated way to practice with pipes.)
5. Write a python script that reads from standard input, sorts lines in reverse alphabetical order, and prints the result. It should behave like `sort -r`. It does not need to process any command line arguments. *Submit your script in a file called `my_reverse_sort.py`. Do not submit this script in `shell.txt`*

Standard Error (10 points)

In addition to standard input and standard output, there is a third stream, standard error. If there is an error in a chain of pipes, it will be printed to the terminal rather than buried in the input to the next program.

1. Recall that `ls -a | sort > sorted.txt` puts all the names of files in a directory sorted in alphabetical order into the file `sorted.txt`. If you modify the command to be `ls -a -hippo | sort > sorted.txt`, what text is in `sorted.txt`, what is outputted as standard error, and why? *Answer this question in `shell.txt`. Submit this file.*
2. Create a python script that prints reversed sorted output to standard error. Use it to sort `ls -a` instead of `sort`. *Submit the file containing the script as `my_sort_status.py`.*

Part 2: Networking

The learning objectives of this section are to familiarize you with how a TCP/IP server works and how to explore a network to find what computers (and robots!) are around, and then how to connect to them. We will use tools at a lower level than the robot programming interface you will use in the rest of the course, in order to focus on the general networking ideas.

Netcat (20 points)

The command `nc` is short for “netcat” and is similar to `cat` but works over network connections. It reads from standard input and writes its contents not to standard output, but to a specified server. *Write your answers in the corresponding sections of `networking.txt`.*

1. Point `nc` to google.com as follows: `nc www.google.com 80` When you first connect, it will be silent. Then type any arbitrary text and press enter. What is the error number?
2. Now type some valid http into nc: `GET / HTTP/1.1`. What is the output?
3. Now use `nc` to make a server. In one window, type `nc -l 12345`. This will cause `nc` to listen on port 12345. In another terminal on the same machine, type `nc localhost 12345`. You can type a message in one window and it will appear in the other window (and vice versa). This trick can be very useful to test basic internet connectivity - if the client and server can send packets at all. *No answer is required for this question.*
4. By convention, `roscore` listens on port 11311. Try using `nc` to connect to port 11311 on a machine where `roscore` is running, such as the Raspberry Pi on your drone. What protocol is roscore using to communicate (think application layer)?

Talking to Your Robot (10 points)

So far, this assignment has required access to `localhost`, the local machine you are connected to, and `google.com`.

Most commonly, the base station and robot are connected over TCP/IP to the same local network. Then you can look up your machine’s IP address (`ifconfig` in Unix; other ways in other OSes), and your robot’s IP address, and connect them. How can you find your robot’s IP address? Well it’s a chicken-and-egg problem. If you knew the IP address, you can connect to the robot and run `ifconfig` and find the IP address, but you don’t know the IP address.

What to do? There are several solutions. *Write the answers to the following questions in `networking.txt`.*

1. Brainstorm how you can solve the chicken-and-egg program to connect to your robot. List three different solutions.

Look Ma, No Internet! (10 points)

But what about if there *is* no public internet connection? What if you want to fly your drone in the wilderness? Well, there does exist cellular modems and satellite connections, but you can also tell your drone to act as a Wifi Hotspot. It can create a network and run a DHCP server. You can configure this on your drone using the file `/etc/hostapd/hostapd.conf`. Then you can connect your laptop’s base station using the SSID and passphrase specified in that file, and connect to the drone.

Alternatively you can set up your laptop as the Wifi base station and configure the drone to connect to its network. The details will vary depending on your laptop OS and settings.

Your Raspberry Pi is configured to be a Wireless AP Master by default. Connect to it with your base station. *Write the answers to the following questions in `networking.txt`.*

1. Which machine is acting as the DHCP server?
2. What is the Raspberry Pi’s IP address? What is yours?
3. Describe another network configuration for the wifi, other than the Raspberry Pi being a Wireless AP Master.
4. Describe three network configurations for a network allowing a basestation and Duckiedrone to communicate with each other. Feel free to add additional devices, such as a cell phone performing internet connection sharing.

Handin

When you are done, use this link to create your assignment Github Repo.

Repo should include:

- `hello1.py`, `hello2.py`, `hello.sh`, `my_cat.py`, `my_reverse_sort.py`, `my_sort_status.py`

- `shell.txt`, `networking.txt`

Assignment 4: Middleware

This unit asks you to think and learn about middleware. For our drone, we use ROS (Robot Operating System).

This assignment will help you understand how the different components of your drone talk with each other. ROS is a framework (known as 'middleware') for robot software development that is widely used on both industrial and commercial settings and is currently the industry standard in research. You will go through a few tutorials to gain exposure to the core concepts of ROS.

Before you begin the ROS component of this assignment, read through the ROS section of the Software Architecture portion of the Operations Manual. This document provides a general overview of ROS. Do not worry about understanding everything in this section; we are asking you to read it only to expose you to the material you will be covering in the assignment and throughout the course.

Assignment

First set up the docker image following these instructions.

Creating a Publisher and Subscriber (50 points)

Answer these questions in `ros.txt` and submit the ROS package you create.

1. Read understanding nodes.
2. Start the `screen` session we use to fly the drone. (Look here to refresh about the screen environment.) Use `rosnode list` to display what nodes are running when you start the screen. If you wish, take a look at the software architecture diagram and look at all of the blue ROS topics to gain a visual understanding of all of the nodes that are running. Once again, do not worry about understanding everything now, or knowing what each topic is used for- you will learn this through experience as the course progresses. *No answer is required for this question*
3. Use `rosnode info` to find out more about as many nodes as you'd like. What is the name of the time of flight (tof) node and what topics does it publish?
4. Do the ROS tutorial to create a package. Name your package `ros_assignment_pkg`. Create it in the `catkin_ws/src` directory on your drone.
5. Do the building packages tutorial.
6. Follow the ROS publisher/subscriber tutorial using the workspace and package you created above. *Hand in the entire package.*
7. Start the `screen` session we use to fly the drone. Use `rostopic echo` and `rostopic hz` to examine the results of various topics. What is the rate at which we are publishing the range reading?

Messages (5 points)

Make all modifications in your ROS package from Problem 1 and hand in the package

1. Read Creating a ROS msg. You do not need to read the section on services.
2. In your package from question 1, create a ROS message called `MyMessage` with a field for a `string`, called `name`, and a field for an array of `float64`, called `contents`. Edit files such as `CMakeLists.txt` to ensure your message is compiled and available for use. *Make these modifications in the package from problem 1 and hand it in.*

Reading the TOF Sensor (15 points)

1. Write a ROS subscriber on your drone to read the values from the time of flight sensor topic and print them to `stdout`. *Name the file `my_echo.py` and submit it.*
2. Write a second ROS subscriber that listens to the time of flight sensor topic and calculates the mean and variance over a ten second window using NumPy. Print these values to `stdout`. *Name the file `mean_and_variance.py` and submit it.*

Handin

When you are done, use this link to create your assignment Github Repo.

- `my_echo.py`, `mean_and_variance.py`
- `ros.txt`
- `ros_assignment_pkg`

Assignment 5: Your Robot's Sensors

Your drone is equipped with three sensors:

1. An inertial measurement unit (IMU)
2. A time-of-flight (ToF) sensor
3. A downward facing camera.

Thanks to these sensors, the drone is equipped with enough understanding of its environment to control its flight and fly autonomously. Each sensor is described below. By interfacing with each of these sensors, you will gain exposure to core robotics concepts including frame conversions, interpreting digital signals, and computer vision.

Time-of-flight Sensor

A range sensor is any sensor that measures the distance to an object. There are three main types that are used on quadcopters: ultrasonic, infrared, and time-of-flight. For ultrasonic and infrared, a wave is emitted from one element of the sensor and received by the other. The time taken for the wave to be emitted, reflected, and be absorbed by the second sensor allows the range to be calculated. Infrared is more accurate, less noisy, and has a better range than the ultrasonic range sensor. The time-of-flight sensor shines infrared light at the world and measures how long it takes to bounce back. Your drone uses the time-of-flight (TOF) sensor because it accurately measures range and does not require an extra analog to digital converter board as does the infrared sensor.

Inertial Measurement Unit (IMU)

An IMU is a device that uses accelerometers and gyroscopes to measure forces (via accelerations) and angular rates acting on a body. The IMU on the Duckiedrone is a built-in component of the flight controller. Data provided by the IMU are used by the state estimator, which you will be implementing in the next project, to better understand its motion in flight. In addition, the flight controller uses the IMU data to stabilize the drone from small perturbations.

The IMU can be used to measure global orientation of roll and pitch, but **not** yaw. This is because it measures acceleration due to gravity, so it can measure the downward pointing gravity vector. However, this information does not give a global yaw measurement. Many drones additionally include a magnetometer to measure global yaw according to the Earth's magnetic field, but the Duckiedrone does not have this sensor.

Note that IMUs do NOT measure position or linear velocity. The acceleration measurements can be integrated (added up over time) to measure linear velocity, and these velocity estimates can be integrated again to measure position. However, without some absolute measurement of position or velocity, these estimates will quickly diverge. To measures these properties of the drone, we need to use the camera as described below.

Camera

Each drone is equipped with a single Arducam 5 megapixel camera. The camera is used to measure motion in the planar directions. This camera points down towards the ground to measure `x`, `y`, and yaw velocities of the drone using optical flow vectors that are extracted from the camera images. This is a lightweight task, meaning that it does not require a lot of computational effort, because these vectors

are already calculated by the Raspberry Pi's image processor for h264 video encoding. We also use the camera to estimate the relative position of the drone by estimating the rigid transformations between two images.

Assignment 5: Sensors Theory

This assignment comprises several parts:

1. ToF Theory
2. Affine Transforms
3. Rotation Representations
4. Optical Flow

Please complete all parts of this assignment. In Part 4, you will find a link to GitHub classroom that will contain a `solutions.tex` template to submit your answers.

Part 1: Estimating Height with the Time of Flight Sensor

The time-of-flight (TOF) sensor on the drone outputs a distance estimate. ROS requires all distances to be in meters.

Questions

1. Look on the data sheets for the TOF sensor on your drone. You will be using the recommended Python library to read data from the TOF sensor. In what units does it output distances?
2. This Adafruit product uses the same TOF sensor as your drone. What are the minimum and maximum distances that the sensor can measure?

Part 2: Affine Transformations

Background Information

In order to estimate the Duckiedrone's position (a 2-dimensional column vector $v = [x\ y]^T$) using the camera, you will need to use affine transformations. An affine transformation $f : \mathbb{R}^n \to \mathbb{R}^m$ is any transformation of the form $v \to Av + b$, where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. The affine transformations we are interested in are *rotation*, *scale*, and *translation* in two dimensions. So, the affine transformations we will look at will map vectors in $\mathbb{R}^2$ to other vectors in $\mathbb{R}^2$.

Let's first look at rotation. We can rotate a column vector $v \in \mathbb{R}^2$ about the origin by the angle θ by premultiplying it by the following matrix:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Let's look at an example. Below we have the vector $[1, 2]^T$. To rotate the vector by $\theta = \frac{2\pi}{3}$, we pre-multiply the vector by the rotation matrix:

$$\begin{bmatrix} \cos\frac{2\pi}{3} & -\sin\frac{2\pi}{3} \\ \sin\frac{2\pi}{3} & \cos\frac{2\pi}{3} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -2.232 \\ -0.134 \end{bmatrix}$$

A graphical representation of the transformation is shown below. The vector $[1, 2]^T$ is rotated $\frac{2\pi}{3}$ about the origin to get the vector $[-2.232, -0.134]^T$

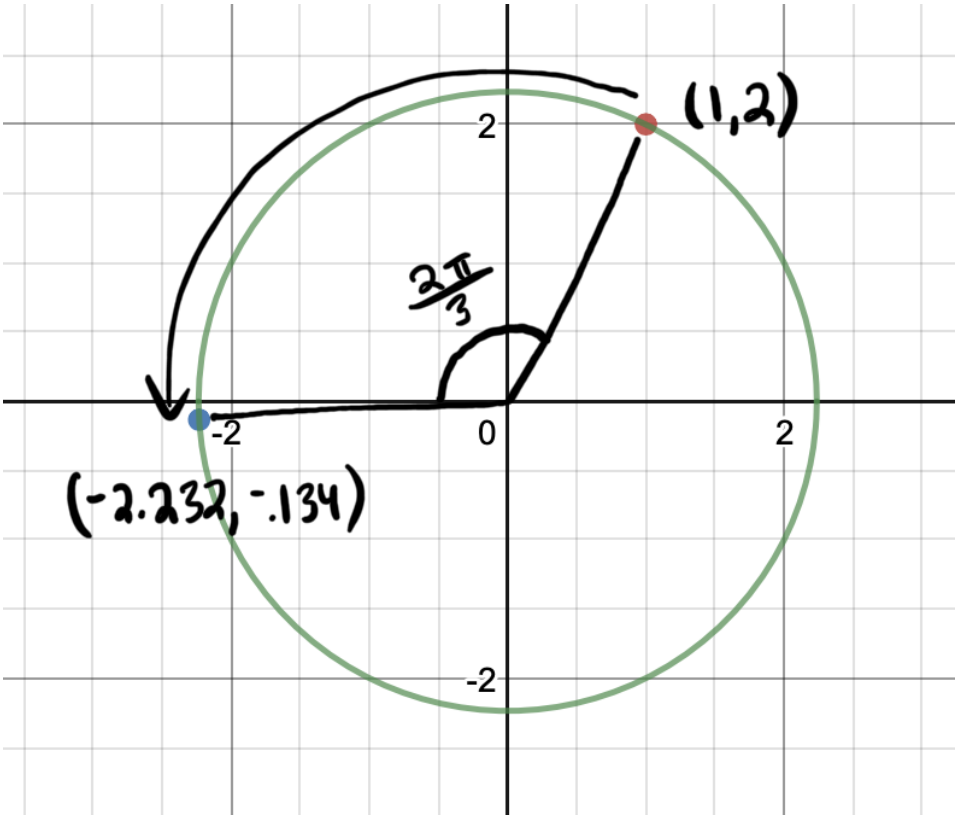


Fig. 1 Rotating one point about the origin

Next, let's look at how scale is represented. We can scale a vector $v \in \mathbb{R}^2$ by a scale factor s by pre-multiplying it by the following matrix:

$$\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}$$

We can scale a single point $[1, 2]^T$ by a factor of `.5` as shown below:

$$\begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} .5 \\ 1 \end{bmatrix}$$

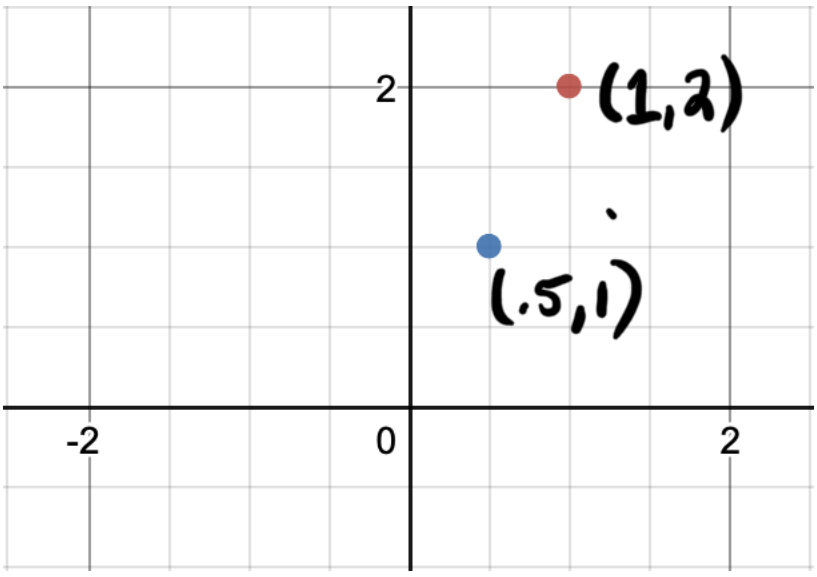


Fig. 2 Scaling one point

When discussing scaling, it is helpful to consider multiple vectors, rather than a single vector. Let's look at all the points on a rectangle and multiply each of them by the scale matrix individually to see the effect of scaling by a factor of `.5`:

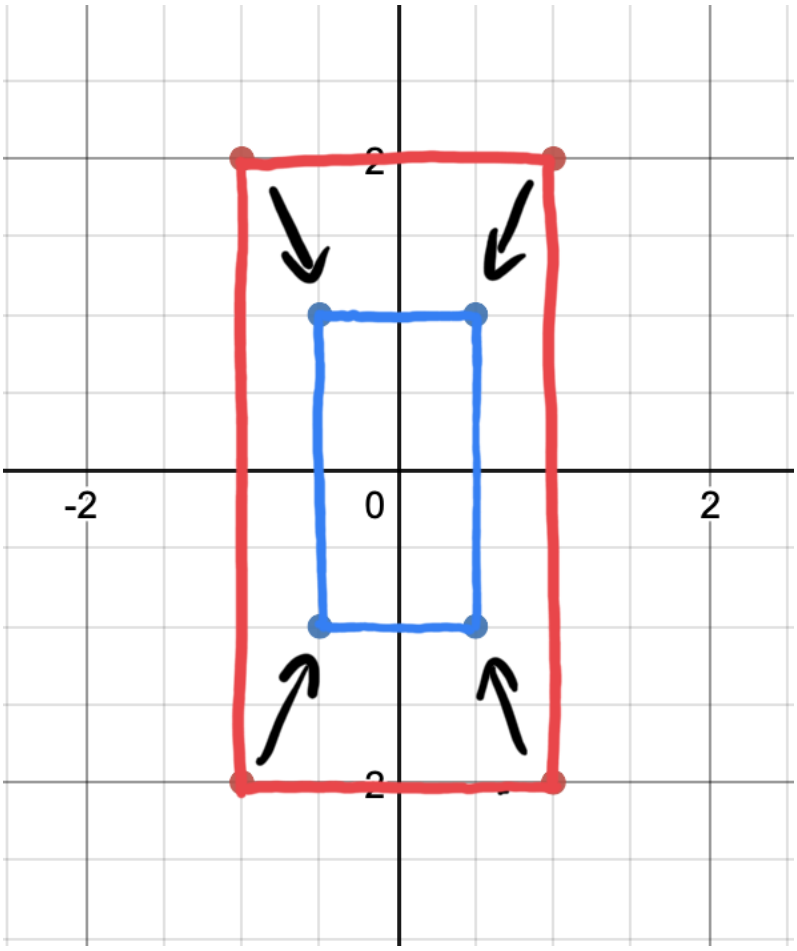


Fig. 3 Scaling multiple points

Now we can see that the rectangle was scaled by a factor of `.5`.

What about translation? Remember that an affine transformation is of the form $v \rightarrow Av + b$. You may have noticed that rotation and scale are represented by only a matrix A, with the vector b effectively equal to 0. We could represent translation by simply adding a vector $b = [dx\ dy]^T$ to our vector v. However, it would be convenient if we could represent all of our transformations as matrices, and then obtain a single transformation matrix that scales, rotates, and translates a vector all at once. We could not achieve such a representation if we represent translation by adding a vector.

So how do we represent translation (moving dx in the x direction and dy in the y direction) with a matrix? First, we append a 1 to the end of v to get $v' = [x, y, 1]^T$. Then, we premultiply v' by the following matrix:

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$$

Even though we are representing our x and y positions with a 3-dimensional vector, we are only ever interested in the first two elements, which represent our x and y positions. The third element of v' is *always* equal to 1. Notice how pre-multiplying v' by this matrix adds dx to x and dy to y. $$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ 1 \end{bmatrix}$$

So this matrix is exactly what we want!

As a final note, we need to modify our scale and rotation matrices slightly in order to use them with v' rather than v. A summary of the relevant affine transforms is below with these changes to the scale and rotation matrices.

$$\text{Rotation: } \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{Scale: } \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{Translation: } \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$$

Estimating Position on the Duckiedrone

Now that we know how rotation, scale, and translation are represented as matrices, let's look at how you will be using these matrices in the sensors project.

To estimate your drone's position, you will be using a function from OpenCV called `estimateRigidTransform`. This function takes in two images I_1 and I_2 and a boolean B. The function returns a matrix estimating the affine transform that would turn the first image into the second image. The boolean B indicates whether you want to estimate the affect of shearing on the image, which is another affine transform. We don't want this, so we set B to `False`.

`estimateRigidTransform` returns a matrix in the form of:

$$E = \begin{bmatrix} s\cos\theta & -s\sin\theta & dx \\ s\sin\theta & s\cos\theta & dy \end{bmatrix}$$

This matrix should look familiar, but it is slightly different from the matrices we have seen in this section. Let R, S, and T be the rotation, scale, and translation matrices from the above summary box. Then, E is the same as TRS, where the bottom row of TRS is removed. You can think of E as a matrix that first scales a vector $u = [x, y, 1]^T$ by a factor of s, then rotates it by θ, then translates it by dx in the x direction and dy in the y direction, and then removes the 1 appended to the end of the vector to output $u' = [x', y']$.

Wow that was a lot of reading! Now on to the questions…

Questions

1. Your Duckiedrone is flying over a highly textured planar surface. The Duckiedrone's current x position is x_0, its current y position is y_0, and its current yaw is ϕ_0. Using the Raspberry Pi Camera, you take a picture of the highly textured planar surface with the Duckiedrone in this state. You move the Duckiedrone to a different state (x_1 is your x position, y_1 is your y position, and ϕ_1 is your yaw) and then take a picture of the highly textured planar surface using the Raspberry Pi Camera. You give these pictures to `esimateRigidTransform` and it returns a matrix E in the form shown above.
   Write expressions for x_1, y_1, and ϕ_1. Your answers should be in terms of x_0, y_0, ϕ_0, and the elements of E. Assume that the Duckiedrone is initially is located at the origin and aligned with the axes of the global coordinate system.

- Hint 1: Your solution does not have to involve matrix multiplication or other matrix operations. Feel free to pick out specific elements of the matrix using normal 0-indexing, i.e. $E[0][2]$.
- Hint 2: Use the function arctan2 in some way to compute the yaw.

Part 3: Gimbal Lock

The orientation of an object in 3D space can be described by a set of three values: (α, β, γ), where α is roll, β is pitch, and γ is yaw.

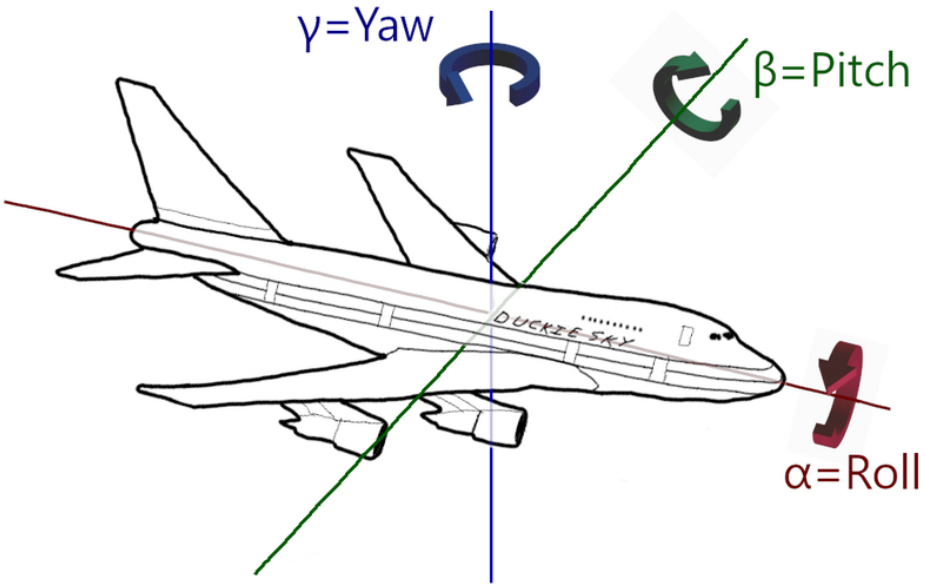


Fig. 4 Roll, pitch, and yaw

Mathematically, any point $\mathbf{p}$ on an object that undergoes rotation (α, β, γ) will have a new coordinate $\mathbf{p}'$ calculated as follows:

$$\mathbf{p}' = R\mathbf{p}$$

Where:

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

$$R = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Gimbal Lock

Ideally, we would hope that the parameters (α, β, γ) are enough to rotate any point $\mathbf{p}$ (distance d from the origin) to any other point $\mathbf{p}'$ (also distance d from the origin, since rotations do not change distance). Upon closer thought, it would seem as if we have more than enough parameters to do this, since it only takes two parameters (θ, ϕ) to describe all points on the 3D unit sphere

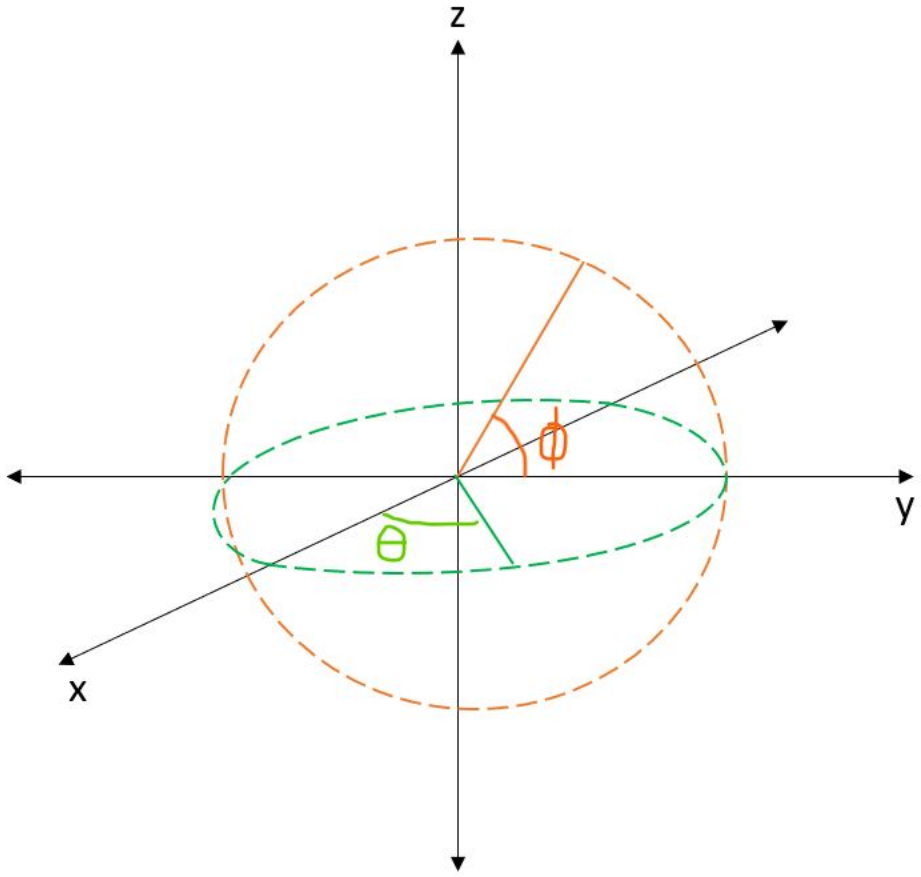


Fig. 5 Two parameters sweeping out a sphere

However, this intuition is a bit off. If any one parameter is held fixed, it may be impossible for $\mathbf{p}$ to be rotated to some other $\mathbf{p}'$ by varying the remaining two parameters. Moreover, if a certain parameter is set to a certain problematic value, then varying the remaining two parameters will either sweep out a circle (not a sphere!), or not affect $\mathbf{p}$ at all, depending on what $\mathbf{p}$ is. This result is way different from what we expected! The name for this degenerate case is gimbal lock.

Questions

1. Suppose an airplane pitches the nose up to a pitch of $-\pi/2$ ($90\backslash\mathrm{degree}$ up):

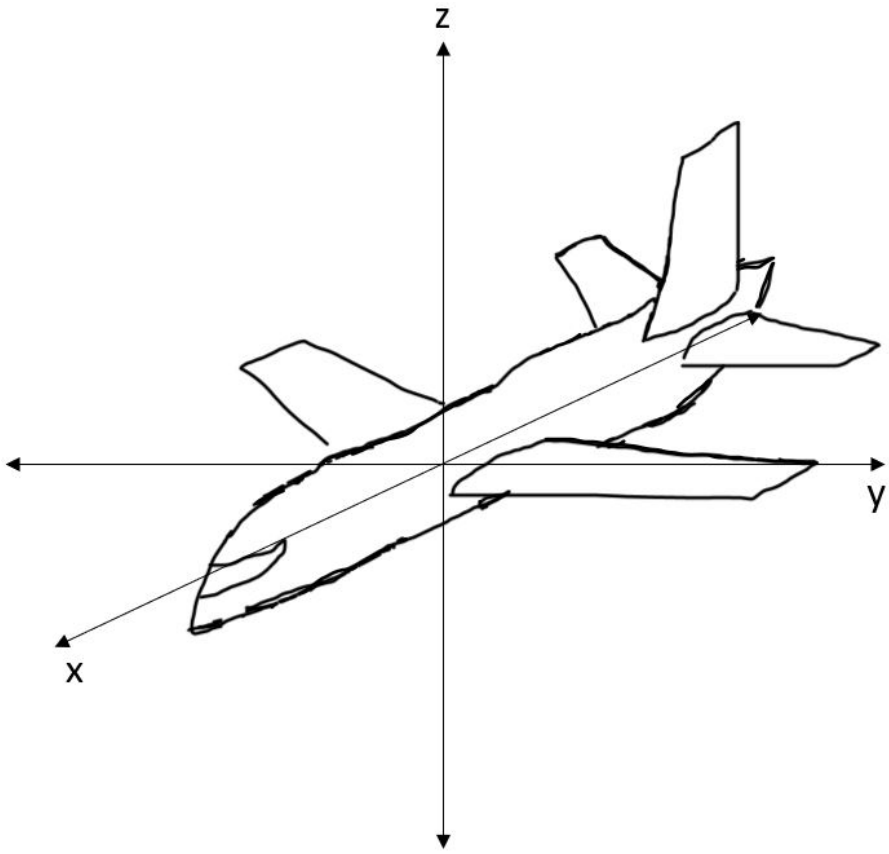


Fig. 6 Airplane

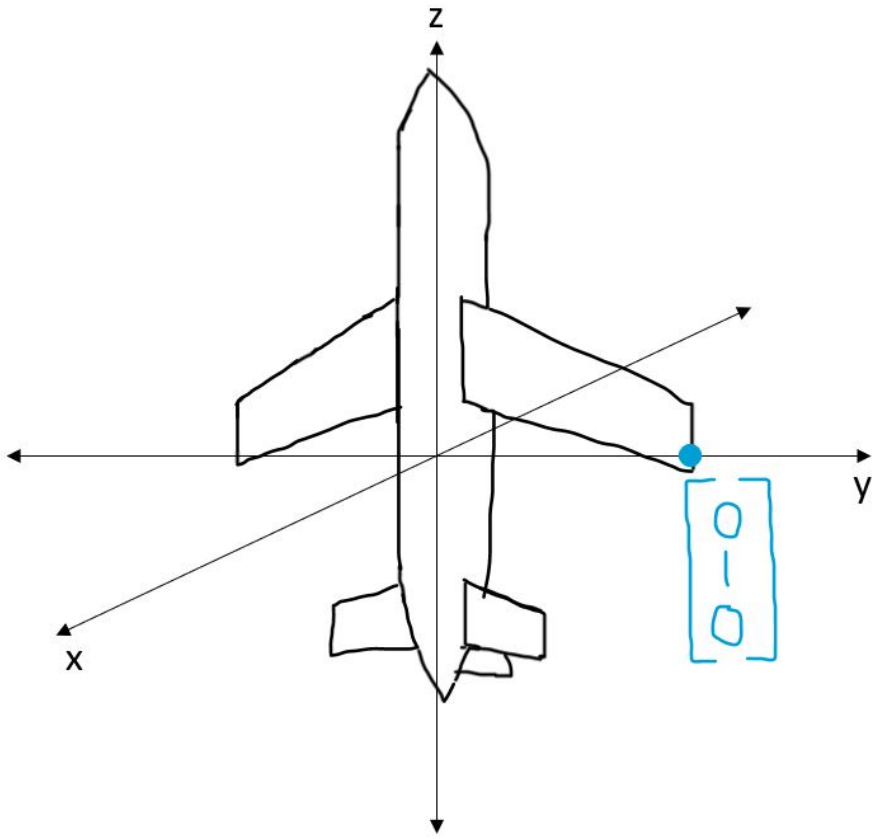


Fig. 7 Airplane with pitch at 90 degrees

Let $R_{\text{gim}\beta}$ denote the rotation matrix R for $\beta = -\pi/2$. Prove that

$$R_{\text{gim}\beta} = \begin{bmatrix} 0 & -\sin(\alpha+\gamma) & -\cos(\alpha+\gamma) \\ 0 & \cos(\alpha+\gamma) & -\sin(\alpha+\gamma) \\ 1 & 0 & 0 \end{bmatrix}$$

2. Consider the point $\mathbf{p} = [0 \quad 1 \quad 0]^T$ on the pitched airplane, i.e. the tip of the wing. Does there exist any α, γ such that $\mathbf{p}' = R_{\text{gim}\beta}\mathbf{p}$ for $\mathbf{p}' = [1 \quad 0 \quad 0]^T$?
   Show your work and briefly explain your reasoning (1-2 sentences).
3. Consider the point $\mathbf{p} = [0 \quad 1 \quad 0]^T$ on the pitched airplane, i.e. the tip of the wing: Can we set α, γ such that:

   $$\mathbf{p}' = R_{\text{gim}\beta}\mathbf{p}$$

   for some $\mathbf{p}'$ on the XY unit circle (e.g. $[\frac{\sqrt{2}}{2} \quad \frac{\sqrt{2}}{2} \quad 0]^T$)?
   You do not have to show any work, but briefly explain your reasoning (1-2 sentences).
4. Consider the point $\mathbf{p} = [0 \quad 1 \quad 0]^T$ on the pitched airplane, i.e. the tip of the wing. Can we set α, γ such that:

   $$\mathbf{p}' = R_{\text{gim}\beta}\mathbf{p}$$

   for some $\mathbf{p}'$ on the YZ unit circle (e.g. $[0 \quad \frac{\sqrt{2}}{2} \quad \frac{\sqrt{2}}{2}]^T$)?
   You do not have to show any work, but briefly explain your reasoning (1-2 sentences).

Reflect on your answers to the previous 4 questions. What are the questions trying to portray? Why are the answers different? Why is $-\pi/2$ (i.e. pitched up $90\backslash\text{degree}$) a "certain problematic value"? What would happen to an airplane that pitched that much? Could a pilot recover from such a situation? Are 3 parameters enough to allow for rotations in all situations?

Part 4: Estimating Velocity by Summing Optical Flow Vectors

We want to estimate our x and y velocity using the Duckiedrone's camera. Thankfully, optical flow from the Raspberry Pi Camera is calculated on board the Raspberry Pi itself. All we have to do is process the optical flow vectors that have already been calculated for us!

To calculate the x velocity, we have to sum the x components of all the optical flow vectors and multiply the sum by some normalization constant. We calculate the y velocity in the same way. Let c be the normalization constant that allows us to convert the sum of components of optical flow vectors into a velocity.

How do we calculate c? Well, it must have something to do with the current height of the drone. Things that are far away move more slowly across your field of view. If a drone is at a height of `.6` and a feature passes through its camera's field of view in 1 second, then that drone is moving faster than another drone at a height of `.1` whose camera also passes over the same feature in 1 second. If we let a be the altitude of the drone, then the drone's normalization constant must be $c = ab$, where b is some number that accounts for the conversion of optical flow vectors multiplied by an altitude to a velocity. You do not have to worry about calculating b (the *flow coefficient*), as it is taken care of for you.

In summary, to calculate the x velocity, we have to sum the x components of the optical flow vectors and then multiply the sum by ab. The y velocity is calculated in the exact same way.

Questions

1. The Pi calculates that the optical flow vectors are [5 4], [1, 2], and [3, 2]. The flow vectors are in the form [x-component, y-component]. What are your x and y velocities $\dot{x}$ and $\dot{y}$? Your solution will be in terms of a, the altitude, and b, the flow coefficient.

Handin

Use this link to access the assignment on GitHub classroom. Commit the files to hand in, as you did in the Introduction assignment.

Your handin should contain the following files:

- `solutions.tex`
- `solutions.pdf`

Project 2: Sensor Interfacing

Overview

In this project, you will be interfacing with your drone's sensors to extract data, parse it into useful values, and publish the data on ROS topics. First, you will interface with the infrared range sensor, thus providing the drone with knowledge of its height relative to the ground. Then, you will interface with the IMU through the flight controller to extract the attitude of the drone (roll, pitch, and yaw), linear accelerations, and calculate the angular rates. Finally, you will interface with the camera to extract velocities using optical flow, and positions using rigid transforms. Woah, that's a lot of data! This is because you are in fact obtaining all the information from each sensor that you will need for the drone to fly autonomously. In the next project, you will write a state estimator which fuses all of this sensor data to estimate the state of the drone.

How this project fits into software stack

Take a look at the software architecture diagram and notice the hardware components: *Flight Controller*, *Infrared Sensor*, and *Camera*. This is the hardware you'll be interfacing with in this project. Also notice the corresponding ROS nodes in the diagram. These are the ROS nodes you'll be creating to extract and publish sensor values.

A note about how to approach this project

These docs give a high-level overview of the project. You will find more detailed directions in the stencil code. If you are unsure about what you have to do after reading these docs, the stencil code should give you a clearer idea.

Handin

Use this link to generate a GitHub repo for this project. Clone the directory to your computer with `git clone https://github.com/h2r/project-sensors-implementation-yourGithubName.git`. This will create a new folder.

When you submit your assignment, your folder should contain modified versions of the following files in addition to the other files that came with your repo:

- `student_infrared_pub.py`
- `student_analyze_flow.py`
- `student_analyze_phase.py`
- `student_flight_controller_node.py`

Commit and push your changes before the assignment is due. This will allow us to access the files you pushed to GitHub and grade them accordingly. If you commit and push after the assignment deadline, we will use your latest commit as your final submission, and you will be marked late.

```
cd project-sensors-implementation-yourGithubName
git add -A
git commit -a -m 'some commit message. maybe hand-in, maybe update'
git push
```

Note that assignments will be graded anonymously, so please don't put your name or any other identifying information on the files you hand in.

Using your Time-of-flight Sensor

In this part of the project, you will learn how to estimate the drone's height using its time-of-flight sensor. The drone is equipped with a VL53L0X distance sensor, which is used for estimating the distance from the drone to the ground. The sensor outputs a digital signal containing the distance from

the sensor which is read in by the Raspberry Pi via I2C communication using the associated Python library.

Setup

Change to `~/catkin_ws/src` on your drone, and then run

```
git clone https://github.com/h2r/project-sensors-implementation-yourGithubName
```

You should create a GitHub personal access token for your drone to make this possible. It only needs permissions to read and write to repositories.

Change directories into `~/catkin_ws/src/project-sensors-implementation-yourGithubName`. You can run

```
rosrun project-sensors-yourGithubName student_tof_pub.py
```

You may stop `student_tof_pub.py` with `ctrl-c`, edit it within that tab, and then re-run

```
rosrun project-sensors-yourGithubName student_tof_pub.py
```

to test your changes.

Problem 1: Publish your TOF Reading

In `student_tof_pub.py`, fill in the minimum range, maximum range, and current range read from the sensor into the ROS message. When you run this node, you will be publishing a ROS Range message which is a standard message included with ROS.

Checkoff:

Using `rostopic echo /pidrone/range` or the height graph on the web interface, verify that:

- The TOF node is publishing a message with all the fields you want
- The range field of the message is a roughly accurate measure of the drone's altitude

You can now fly your drone with your own range node!

Interfacing with the IMU

Your drone is equipped with a Skyline32 Flight Controller which has a built in IMU. In this part of the project, you will learn how to interface with the flight controller board to extract the attitude, accelerations, angular rates of the drone from the built-in IMU. In addition, you will extract the battery levels from the flight controller so that you'll be able to tell when you're battery is too low.

Setup Change directories into `~/ws/src/pidrone_pkg` and modify `pi.screenrc` to start up with your flight controller node by changing `python flight_controller_node.py\n` to `rosrun project-sensors-yourGithubName student_flight_controller_node.py\n` (or, alternatively, `python \path\to\student_flight_controller_node.py\n`).

Problem 1: Extracting the Battery Data

The flight controller is capable of reading the voltage and current of the power source plugged into the drone. This is possible because of the red and brown wire pair (i.e. *battery monitor* wire pair) plugged into the FC. The power information is useful because it allows us to programmatically shut down the drone if the voltage is too low (e.g. LiPo batteries are quickly ruined if discharged too low).

TODO:

1. Take a look at Battery.msg in the ~/ws/src/pidrone_pkg/msg directory on your drone. This is a custom message we've created to communicate the battery values.
2. In `student_flight_controller_node.py`, do the following:
    - Fill in each `TODO` regarding the `battery_message` in the `__init__` method.

- Fill in each `TODO` in the `update_battery_message` method.

Problem 2: Extracting IMU data

Linear accelerations and attitude (i.e. roll, pitch, yaw) can also be extracted from the FC, thanks to the accelerometer and gyroscope. In addition, the angular rates (e.g. change in roll over change in time) can be calculated by using the attitude measurements.

TODO:

1. Take a look at the Imu ROS message type to get an understanding of the data you'll be collecting.
2. In `student_flight_controller_node.py`, do the following:
    - Fill in each `TODO` regarding the `imu_message` in the `__init__` method.
    - Fill in each `TODO` in the `update_imu_message` method.

Velocity Estimation via Optical Flow

In this part of the project you will create a class that interfaces with the Arducam to extract planar velocities from optical flow vectors.

Code Structure

To interface with the camera, you will be using the `raspicam_node` library. This library publishes both images and optical flow vectors to ROS topics. You will estimate velocity using the flow vectors, and estimate small changes in position by extracting features from pairs of frames. In the sensors project repo, we've included a script called `student_optical_flow.py` which you will edit, so it publishes the estimated velocity from the flow vectors.

Similarly a second script is `student_rigid_transform.py` which you will edit, so it subscribes to the image topic and publishes position estimates.

Analyze and Publish the Sensor Data

On your drones, the chip on the Raspberry Pi dedicated to video processing from the camera calculates motion vectors (optical flow) automatically for H.264 video encoding. Click here to learn more. You will be analyzing these motion vectors in order to estimate the velocity of your drone.

Exercises

You will now implement your velocity estimation using optical flow by completing all of the `TODO`'s in student_optical_flow.py. There are two methods you will be implementing.

The first method is `setup`, which will be called to initialize the instance variables.

1. Create a ROS publisher to publish the velocity values.

The perspicacious roboticist may have noticed that magnitude of the velocity in global coordinates is dependent on the height of the drone. Add a subscriber to the topic /pidrone/state to your AnalyzeFlow class and save the z position value to a class variable in the callback. Use this variable to scale the velocity measurements by the height of the drone (the distance the camera is from what it is perceiving).

1. Create a ROS subscriber to obtain the altitude (z-position) of the drone for scaling the motion vectors.

The second method is `motion_cb`, which is called every time that the camera gets a set of flow vectors, and is used to analyze the flow vectors to estimate the x and y velocities of your drone.

1. Estimate the velocities, using the `TODO`'s as a guide.
2. Publish the velocities.

Check your Measurements

You'll want to make sure that the values you're publishing make sense. To do this, you'll be echoing the values that you're publishing and empirically verifying that they are reasonable.

Exercises

Verify your velocity measurements

1. Start up your drone and launch a screen
2. Navigate to `4 and quit the node that is running
3. Run `rosrun project-sensors-yourGithubName student_analyze_flow.py`
4. Enter `rostopic echo /pidrone/picamera/twist`
5. Move the drone by hand to the left and right and forward and backward to verify that the measurements make sense

Checkoff

1. Verify that the velocity values are reasonable (roughly in the range of -1m/s to 1m/s) and have the correct sign (positive when the drone is moving to the right or up, and negative to the left or down).

Position Estimation via OpenCV's estimateRigidTransform

In this part of the project you will create a class that interfaces with the picamera to extract planar positions of the drone relative to the first image taken using OpenCV's estimateRigidTransform function.

Ensure images are being passed into the analyzer

Before attempting to analyze the images, we should first check that the images are being properly passed into the analyze method

Exercises

1. Open `student_rigid_transform_node.py` and print the `data` argument in the method `image_callback`. Verify you are receiving images from the camera.

Analyze and Publish the Sensor Data

To estimate our position we will make use of OpenCV's *estimateAffinePartial2D* function. This will return an affine transformation between two images if the two images have enough in common to be matched, otherwise, it will return None.

Exercises

Complete the TODOs in `image_callback`, which is called every time that the camera gets an image, and is used to analyze two images to estimate the x and y translations of your drone.

1. Save the first image and then compare subsequent images to it using cv2.estimateAffinePartial2D. (Note that the fullAffine argument should be set to False.)
2. If you print the output from `estimateAffinePartial2D`, you'll see a 2x3 matrix when the camera sees what it saw in the first frame, and a None when it fails to match. This 2x3 matrix is an affine transform which maps pixel coordinates in the first image to pixel coordinates in the second image.
3. Implement the method `translation_and_yaw`, which takes an affine transform and returns the x and y translations of the camera and the yaw.
4. As with velocity measurements, the magnitude of this translation in global coordinates is dependent on the height of the drone. Add a subscriber to the topic /pidrone/state and save the value to `self.altitude` in the callback. Use this variable to compensate for the height of the camera in your method from step 4 which interprets your affineTransform.

Account for the case in which the first frame is not found

Simply matching against the first frame is not quite sufficient for estimating position because as soon as the drone stops seeing the first frame it will be lost. Fortunately we have a fairly simple fix for this: compare the current frame with the previous frame to get the displacement, and add the displacement to the position the drone was in in the previous frame. The framerate is high enough and the drone moves slow enough that the we will almost never fail to match on the previous frame.

Exercises

Modify your `RigidTransformNode` class to add the functionality described above.

1. Store the previous frame. When estimateAffinePartial2D fails to match on the first frame, run estimateAffinePartial2D on the previous frame and the current frame.
2. When you fail to match on the first frame, add the displacement to the position in the previous frame. You should use `self.x_position_from_state` and `self.y_position_from_state` (the position taken from the `pidrone/state` topic) as the previous coordinates.

Note The naive implementation simply sets the position of the drone when we see the first frame, and integrates it when we don't. What happens when we haven't seen the first frame in a while so we've been integrating, and then we see the first frame again? There may be some disagreement between our integrated position and the one we find from matching with our first frame due to accumulated error in the integral, so simply setting the position would cause a jump in our position estimate. The drone itself didn't actually jump, just our estimate, so this will wreak havoc on whatever control algorithm we write based on our position estimate. To mitigate these jumps, you should use a filter to blend your integrated estimate and your new first-frame estimate. Since this project is only focused on publishing the measurements, worrying about these discrepancies is unnecessary. In the UKF project, you will address this problem.

Connect to the JavaScript Interface

Now that we've got a position estimate, let's begin hooking our code up to the rest of the flight stack.

To connect to the JavaScript interface, clone `pidrone_pkg` on your base station machine. Point any web browser at the web/index.html directory. This will open up the web interface that we will be using the rest of the semester.

1. Create a subscriber (in the setup function) to the topic `/pidrone/reset_transform` and a callback owned by the class to handle messages. ROS Empty messages are published on this topic when the user presses r for reset on the JavaScript interface. When you receive a reset message, you should take a new first frame, and set your position estimate to the origin again.
2. Create a subscriber to the topic `/pidrone/position_control`. ROS Bool messages are published on this topic when the user presses `p` or `v` on the JavaScript interface. When we're not doing position hold we don't need to be running this resource-intensive computer vision, so when you receive a message you should enable or disable your position estimation code.

Measurement Visualization

Debugging position measurements can also be made easier through the use of a visualizer. A few things to look for are sign of the position, magnitude of the position, and the position staying steady when the drone isn't moving. Note again that these measurements are unfiltered and will thus be noisy; don't be alarmed if the position jumps when it goes from not seeing the first frame to seeing it again.

Exercises

Use the web interface to visualize your position estimates

1. Connect to your drone and start a new screen
2. Run `rosrun project-sensors-yourGithubName student_rigid_transform_node.py` in `4.
3. Hold your drone up about 0.25 m with your hand
4. In the web interface, press `r` and the `p` to engage position hold.
5. Use `rostopic echo /pidrone/picamera/pose` to view the output of your *student_analyze_phase* class

6. Move your drone around by hand to verify that the values make sense.
7. Look at the web interface and see if it tracks your drone. Pressing `r` should set the drone drone visualizer back to the origin.

Checkoff

1. Verify that the position values are reasonable (roughly in the range of -1m to 1m) and have the correct sign (positive when the drone is moving to the right or up, and negative to the left or down).

Project Checkoff

Functionality Check

1. Run `student_tof_pub.py` and open up the web interface. Move the drone up and down and ensure that the height readings are reasonable.
2. Run `student_optical_flow_node.py` and `student_rigid_transform_node.py` and open up the web interface. Turn on velocity control (enabled by default). Slowly move the drone around over a highly textured planar surface and ensure that the raw velocity readings are reasonable.

Questions

You will be asked to answer one of the following questions:

1. What types of measurements does the flight controller report in order to describe the orientation of the drone? What do we do to these measurements and why?
2. How does optical flow allow us to estimate the planar velocity of the drone? Why do we need to fly over a textured surface?
3. Why do we have a `state_callback` in `student_analyze_phase.py`? What do we do with the state information?

Assignment 6: PID Controller Theory

PID Controllers generalities

Introduction

A PID (proportional, integral, derivative) controller is a control algorithm extensively used in industrial control systems to generate a control signal based on error. The error is calculated by the difference between a desired setpoint value, and a measured process variable. The goal of the controller is to minimize this error by applying a correction to the system through adjustment of a control variable. The value of the control variable is determined by three control terms: a proportional term, integral term, and derivative term.

Characteristics of the Controller

Key Terms and Definitions

- **Process Variable**: The parameter of the system that is being monitored and controlled.
- **Setpoint**: The desired value of the process variable.
- **Control Variable/Manipulated Variable**: The output of the controller that serves as input to the system in order to minimize error between the setpoint and the process variable.
- **Steady-State Value**: The final value of the process variable as time goes to infinity.
- **Steady-State Error**: The difference between the setpoint and the steady-state value.
- **Rise Time**: The time required for the process variable to rise from 10 percent to 90 percent of the steady-state value.
- **Settling Time**: The time required for the process variable to settle within a certain percentage of the steady-state value.
- **Overshoot**: The amount the process variable exceeds the setpoint (expressed as a percentage).

General Algorithm

The error of the system $e(t)$, is calculated as the difference between the setpoint $r(t)$ and the process variable $y(t)$. That is:

$$e(t) = r(t) - y(t)$$

The controller aims to minimize the rise time and settling time of the system, while eliminating steady-state error and maximizing stability (no unbounded oscillations in the process variable). It does so by changing the control variable $u(t)$ based on three control terms.

Proportional Term

The first control term is the proportional term, which produces an output that is proportional to the calculated error:

$$P = K_p e(t)$$

The magnitude of the proportional response is dependent upon K_p, which is the proportional gain constant. A higher proportional gain constant indicates a greater change in the controller's output in response to the system's error.

Integral Term

The second control term is the integral term, which accounts for the accumulated error of the system over time. The output produced is comprised of the sum of the instantaneous error over time multiplied by the integral gain constant K_i:

$$I = K_i \int_0^t e(\tau)\,\mathrm{d}\tau$$

Derivative Term

The final control term is the derivative term, which is determined by the rate of change of the system's error over time multiplied by the derivative gain constant K_d:

$$D = K_d \frac{de(t)}{dt}$$

Overall Control Function

The overall control function can be expressed as the sum of the proportional, integral, and derivative terms:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{de(t)}{dt}$$

In practice, the discretized form of the control function may be more suitable for implementation:

$$u(t) = K_p e(t_k) + K_i \sum_{i=0}^{k} e(t_i)\Delta t + K_d \frac{e(t_k) - e(t_{k-1})}{\Delta t}$$

The figure below summarizes the inclusion of a PID controller within a basic control loop.

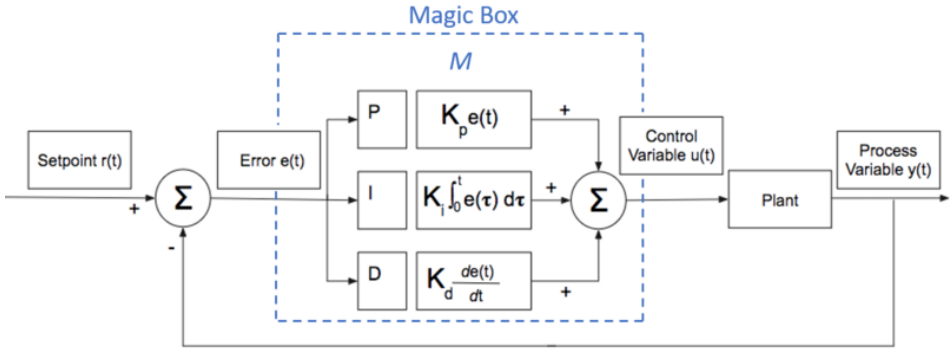


Fig. 8 PID Controller Block Diagram

Tuning

Tuning a PID controller refers to setting the control parameters K_p, K_i, and K_d to optimal values in order to obtain the ideal control response. After understanding the general effects of each control term on the control response, tuning can be accomplished through trial-and-error or by other specialized tuning schemes, such as the Ziegler-Nichols tuning method. A graph of the process variable or system error can display the effects of the controller terms on the system; the control parameters can then be modified appropriately to optimize the control response. Although the independent effects of each parameter are explained below, the three control terms may be correlated and so changing one parameter may impact the influence of another. The general effects of each term are therefore useful as reference, but the actual effects will vary depending on the specific control system.

Effects of K_p

For a given level of error, increasing K_p will proportionally increase the control output. This causes the system to react more quickly (thereby decreasing the rise time and the settling time by a small amount). Even so, setting the proportional gain too high could cause massive overshoot, which in turn could destabilize the system. Increasing K_p also has the effect of decreasing the steady-state error. However, as the value of the process variable approaches the setpoint and the error decreases, the proportional term will also decrease. As a result, with a P-controller (a controller with only the proportional term), the process variable will asymptotically approach the setpoint, but will never quite reach it. Thus, a P-controller cannot be used to completely eliminate steady-state error.

Effects of K_i

The integral term takes into account past error, as well as the duration of the error. If error persists for a long time, the integral term will continue to accumulate and will eventually drive the error down. This has the effect of reducing and eliminating steady-state error. However, the build-up of error can cause the value of the process variable to overshoot, which can increase the settling time of the system, though it decreases the rise time.

Effects of K_d

By calculating the instantaneous rate of change of the system's error and using this slope for linear extrapolation, the derivative term anticipates future error. While the proportional and integral terms both act to move the process variable to the setpoint, the derivative term seeks to dampen their efforts and decrease the amount the system overshoots in response to a large change in error (which would greatly affect the magnitude of the proportional and integral contributions to the overall control output). If set at an appropriate level, the derivative term reduces oscillations, which decreases the settling time and improves the stability of the system. The derivative term has negligible effects on steady-state error and only decreases the rise time by a minor amount.

Summary of Control Terms

Term	Rise Time	Overshoot	Settling Time	Steady-State Error
K_p	Decrease	Increase	Minor Change	Decrease
K_i	Decrease	Increase	Increase	Eliminate
K_d	Minor Change	Decrease	Decrease	No Change

Table 1: General Effects of Each Control Term on the System

Fig. 9 General Effects of Control Terms

Ziegler-Nichols Closed-Loop Tuning Method

Ziegler and Nichols [] developed two techniques for tuning PID controllers, a closed-loop tuning method and an open-loop tuning method. With the closed-loop tuning method, the PID controller is initially turned into a P controller with K_p set to zero. K_p is slowly increased until the system exhibits stable

oscillatory behavior, at which point it is denoted K_u, the ultimate or critical gain. As such, K_u should be the smallest K_p value that causes the control loop to have regular oscillations. The ultimate or critical period T_u of the oscillations needs to be measured. Then, using the constants determined experimentally by Ziegler and Nichols, the controller gain values can be computed as follows:

$$K_p = 0.6K_u$$

$$K_i = 2K_p/(T_u)$$

$$K_d = K_p(T_u)/8$$

Although the Ziegler-Nichols method may yield initial tuning values that work relatively well, the system's control loop can be tuned further by adjusting the controller gain values based on the general effects of each control term as explained above.

Potential Problems

In real-world applications, the PID controller exhibits issues that require modifications to the general algorithm. In certain situations, one may find that a P-Controller, PD-Controller (eliminating the integral term), or a PI-Controller (eliminating the derivative term) are more advantageous controllers for the system. Alternatively, different techniques can be employed to counteract the problems that may affect the usability of a control term.

Integral Windup

Integral wind-up occurs when, due to a large change in setpoint, the control output causes the system's actuator to become saturated. At this point, the integrated error between the process variable and the setpoint will continue to grow (because the actuator is at its limit and cannot drive the process variable any closer to the setpoint). In turn, the control output will continue to grow and will no longer have any effect on the system. When the setpoint finally changes and the error changes sign (meaning the new setpoint is now below the value of the process variable), the integral term will take a while to "unwind" all of the error that it has accumulated before producing a reverse control action that will move the process variable in the correct direction towards the setpoint.

There exist numerous ways to address integral wind-up. One way is to keep the integral term within predefined upper and lower bounds. Another way is to set the integral term to zero if the control output will cause the system's actuator to saturate. Yet another way is to reduce the integral term by a constant multiplied by the difference between the actual output and the commanded output. If the actuator is not saturated, then the difference between the actual and commanded output will be zero and will not affect the integral term. If the actuator is saturated, then the additional feedback in the control loop will drive the commanded output closer to the saturation limit. If the setpoint changes and causes the error to change sign, then the integral term will not need to unwind in order to produce an appropriate control action. Setpoint ramping — in which the setpoint is increased or decreased incrementally to reach the desired value — may also help prevent integral wind-up.

Derivative Noise

Since the derivative term is proportional to the change in error, it is consequently highly sensitive to noise (which would produce drastic changes in error). Using a low-pass filter on the derivative term, or finding the derivative of the process variable (as opposed to the error), or taking a weighted mean of previous derivative terms could help ensure that high-frequency noise does not cause the derivative term to adversely affect the control output.

Cascaded Controllers

When multiple measurements can be used to control a single process variable, these measurements can be combined using a cascaded PID controller. In cascaded PID control, two PID controllers are used conjointly to yield a better control response. The output of the PID controller for the outer control loop determines the setpoint for the PID controller of the inner control loop. The outer loop controller controls the primary process variable of the system, while the inner loop controller controls a system

parameter that tends to change more rapidly in order to minimize the error of the outer control loop. The two controllers have separate tuning values, which can be optimized for the part of the system that they control. This enables an overall better control response for the system as a whole.

High-Level Description of the Duckiedrone PID Stack

The drone platform utilizes a number of PID controllers to autonomously control its motion. The standard PID class implements the discrete version of the ideal PID control function. The control output returned is the sum of the proportional, integral, and derivative terms, as well as an offset constant term, which is the base control output before being corrected in response to the calculated error. A specified control range keeps the control output within predefined bounds.

Cascaded Position and Velocity Controllers

The flight command for the drone consists of four pulse-width modulation (pwm) values that are sent to the flight controller and translated into motor speeds to set the drone's roll, pitch, yaw, and throttle, respectively. When the drone attempts to hover with zero velocity, or when a velocity command is sent from the web interface, the error between the commanded velocity and the actual velocity of the drone (determined by optical flow) is calculated. The x-velocity error serves as input to the roll PID controllers and the y-velocity error serves as input to the pitch PID controllers. For the throttle PID controllers, the z-position error is used as input. The z-velocity error is not used because the actual z-position of the drone is directly measured by the infrared sensor, and is thus easier to control, while the camera estimation of the z-velocity is not as accurate. The output of each controller is then used to set the roll, pitch, and throttle commands to achieve the desired velocity.

To accomplish position hold on the drone, cascaded PID controllers are utilized. The outer control loop is concerned with the position of the drone, and the inner control loop changes the velocity of the drone in order to attain the desired position. The two position PID controllers (one for front-back planar motion and the other for left-right planar motion) each calculate a setpoint velocity based on position error, which serves as input to the velocity PID controller. The roll, pitch, and throttle PID controllers then compute the appropriate flight commands based on the difference between the current velocity and this setpoint velocity.

Low and High Integral Terms

The drone requires two PID controllers to control each of its roll, pitch, and throttle. One controller has a fast-changing integral term with a high K_i value, while the other controller has a slow-changing integral term with a low K_i value. The inclusion of the low integral controller is intended to adjust for systemic sources of error, such as poor weight distribution or a damaged propeller. If the magnitude of the calculated velocity error is below a certain threshold, the flight command is set to the control output of its low integral controller and the integral term of its high integral controller is reset to zero. This helps to prevent integral wind-up for the high integral controller (the throttle PID controllers also use an integral term control range to bound the the value of the integral term and prevent wind-up). If the calculated velocity error is above the specified threshold, it is constrained within a preset range. The flight command is calculated by adding the integral term of the low integral PID controller to the overall control output of the high integral PID controller.

Derivative Smoothing

In order to address the derivative term's sensitivity to high-frequency noise, the derivative term is smoothed over by taking a weighted mean of the past three derivative terms. A derivative term control range is also used to constrict the values of the derivative term for the throttle PID controllers.

Assignment 6: PID Controller Theory

The True Value and Error Curves

The figure below shows a true value curve for a PID controller. Draw the corresponding error curve for this graph. You can draw by hand and upload the picture. (Hint: refer to the error definition equation from before)

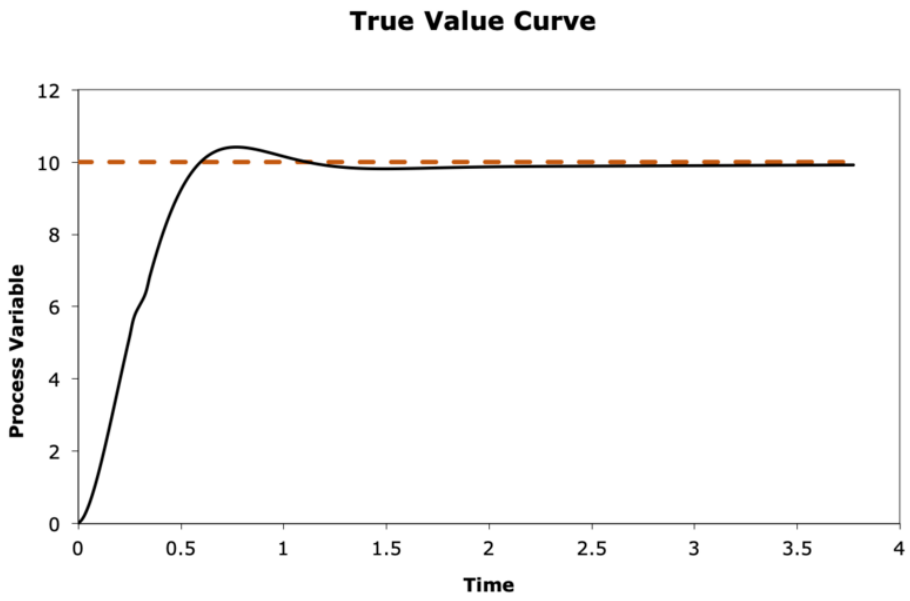


Fig. 10 True Value Curve for A PID Controller. The orange dot line indicates the setpoint and the black line is the true value curve.

Explain an Effect

Answer the following questions (3-5 sentences each):

- What will happen when the absolute value of K_p is very large? What will happen when the absolute value of K_p is very small?
- Can K_p be tuned such that the P term stops oscillations? Why or why not?
- Can the process variable stabilize at the setpoint (i.e. zero steady-state error) with only the P term and the D term? Why or why not?

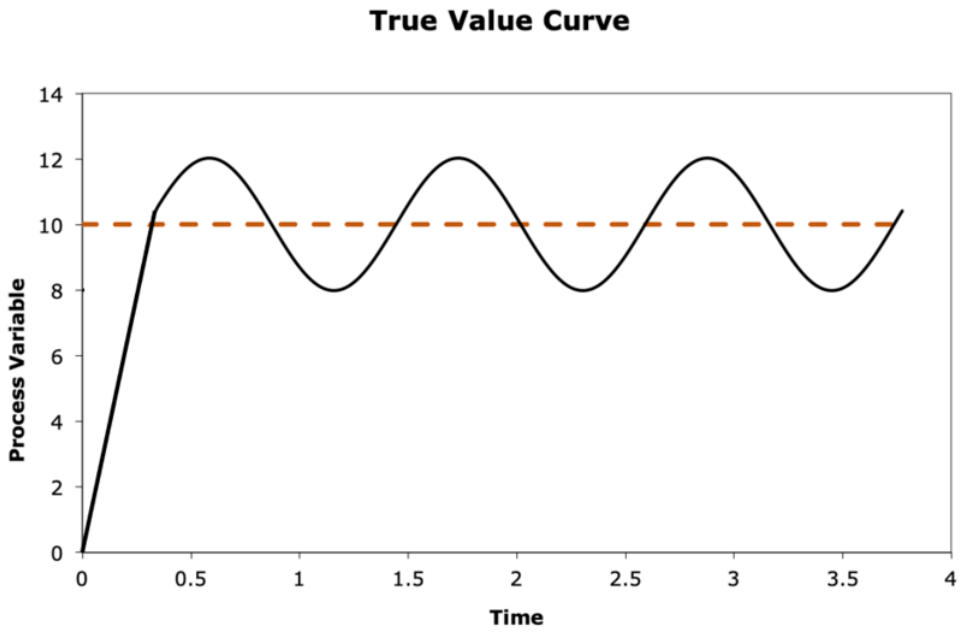
Explain the following effects caused by K_p, K_i and K_d (3-5 sentences each). For example, here is a sample answer (though you do not need to follow the pattern):

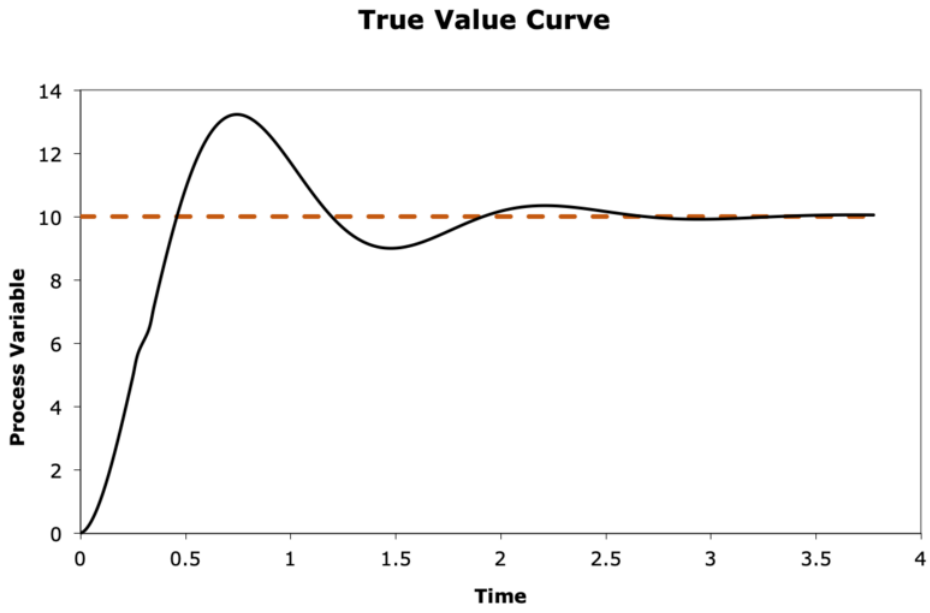
- [Q:] *The rise time decreases when K_d increases.
- [A:] *When K_d increases, the error at time step $t+1$ decreases. This is because larger and larger K_d results in larger and larger control signals at time step t. This drives the system to achieve a lower error at time step $t+1$. As the error at time step $t+1$ decreases, the slope of the true value curve increases. Since the slope increases, the rising time towards the setpoint should decrease (slightly).

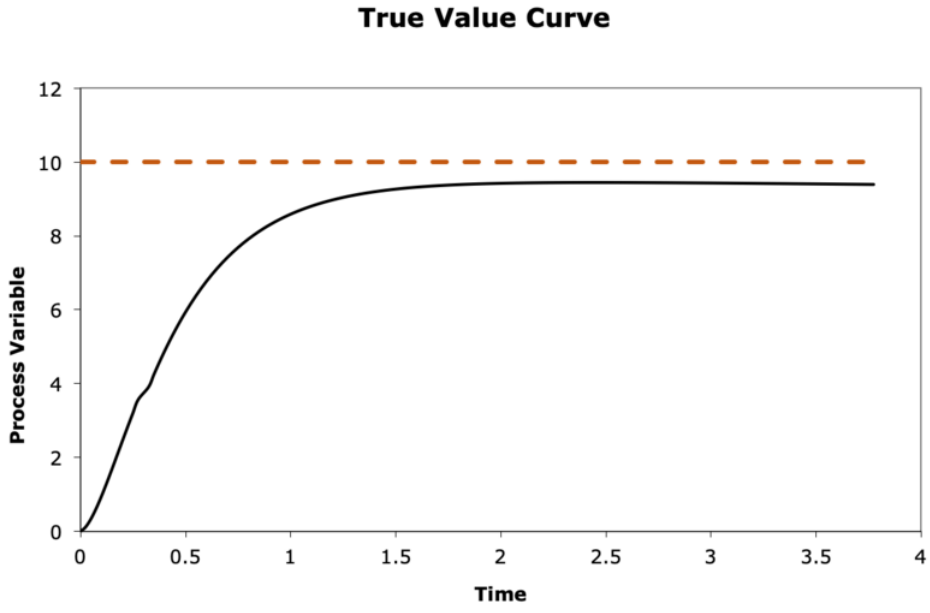
Start Tuning

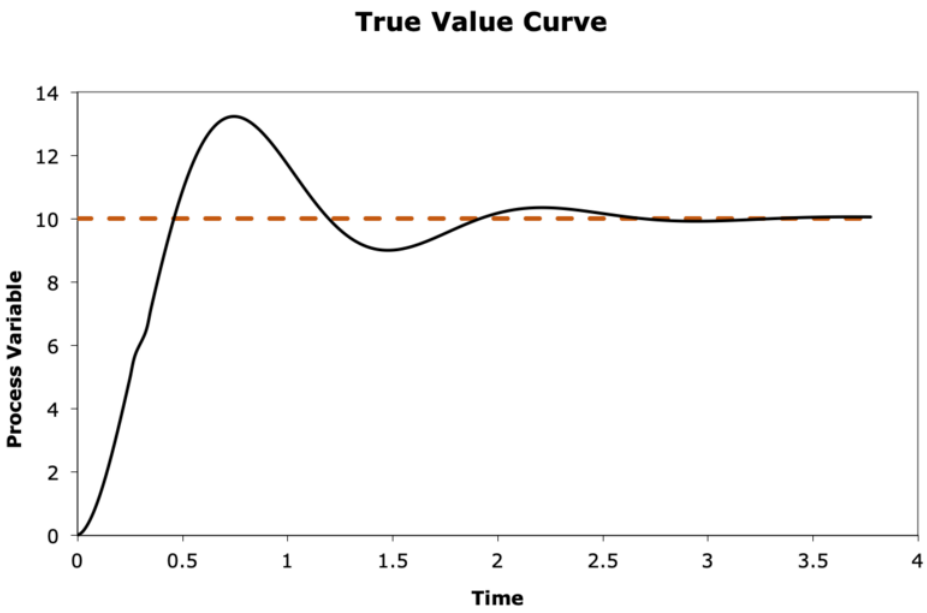
When designing a PID controller, it is important to choose a good set of K_p, K_i, and K_d; poor choices can result in undesirable behavior. The graphs in the figure below illustrate behavior resulting from unknown sets of K_p, K_i, and K_d. In each graph, the orange dot line indicates the setpoint and the black line is the true value curve. For each graph, answer the following (1-2 sentences each):

1. Which term(s) went wrong, if any? In other words, which term(s) are too high or too low?
2. How can you correct the behavior?


True Value Curve
Process Variable
Time
0
2
4
6
8
10
12
14
0
0.5
1
1.5
2
2.5
3
3.5
4


True Value Curve
Process Variable
Time
0
2
4
6
8
10
12
14
0
0.5
1
1.5
2
2.5
3
3.5
4


True Value Curve
Process Variable
Time
0
2
4
6
8
10
12
0
0.5
1
1.5
2
2.5
3
3.5
4



PID on the Duckiedrone

Sometimes a PID controller will have an extra offset/bias term K in the control function (see the equation below). For the drone, this K is the base throttle needed to get the drone off the ground.

$$u\left(t_k\right) = K_p e\left(t_k\right) + K_i \sum_{i=0}^{k} e\left(t_i\right)\Delta t + K_d \frac{e\left(t_k\right) - e\left(t_{k-1}\right)}{\Delta t} + K$$

Altitude Control

Suppose you are implementing an altitude PID controller for your drone (i.e. up/down movement).

1. If the *setpoint* is the desired height of the drone, then what is the *process variable*, the *error* and the *control variable* for the altitude PID controller?
2. What could happen if K is set too high?

Note: We are looking only for a higher level description to demonstrate understanding of the PID controllers.

Velocity Control

Suppose you are implementing a velocity PID controller for your drone. In this case, the drone only moves forward/backward and left/right. Your (hypothetical) controller is implemented so that when 'L' is pressed, the drone moves left at a constant velocity, and when 'L' is released, the drone stops moving.

1. What is the *setpoint*, *process variable*, *error* and *control variable* for the velocity PID controller?
2. How do these key terms change to cause the drone to move when you press 'L'?

Note: We are looking only for a higher level description to demonstrate understanding of the PID controllers.

Handin

Use this link to access the assignment on Github classroom. Commit the files to hand in, as you did in the Introduction assignment. You'll find a template `answers.tex` files for your answers.

Your handin should contain the following files:

- `answers.tex`
- `answers.pdf`

Project 3: Implementing an Altitude PID Controller

Overview

For this project, you will be implementing a one-dimensional PID controller to control the drone's altitude (movement along the drone's z-axis). In part one, you will write your PID class and test it using a drone simulator. In part two, you will answer questions about control.

There is one section that is not required for this project. In Appendix A, we explain how to transfer the altitude PID controller you wrote in part 1 to your drone and tune it to achieve stable flight.

Handin

Use this link to generate a Github repo for this project. Clone the directory to your computer

```
git clone https://github.com/h2r/project-pid-implementation-yourGithubName.git
```

This will create a new folder. The *README.md* in your repo provides short descriptions of each project file.

When you submit your assignment, your folder should contain the following files (that you modified) in addition to all of the other files that came with your repo:

- `answers_pid.md`
- `student_pid_class.py`
- `z_pid.yaml`

Commit and push your changes before the assignment is due. This will allow us to access the files you pushed to GitHub and grade them accordingly. If you commit and push after the assignment deadline, we will use your latest commit as your final submission, and you will be marked late.

```
cd project-pid-implementation-yourGitHubName
git add -A
git commit -a -m 'some commit message. maybe handin, maybe update'
git push
```

Note that assignments will be graded anonymously, so please don't put your name or any other identifying information on the files you hand in.

Part 1: Altitude PID in Simulation

In this part of the project, you will be implementing a PID controller for a simulated drone that can only move in one dimension, the vertical dimension. You can control the speed the motors spin on the drone, which sets the thrust being generated by the propellers. In this system, the process variable is the drone's altitude, the setpoint is the desired altitude, and the error is the distance in meters between the setpoint and the drone's altitude. The output of the control function is a PWM (pulse-width modulation) value between 1100 and 1900, which is sent to the flight controller to set the drone's throttle.

You should implement the discretized version of the PID control function in *student_pid_class.py*:

$$u(t) = K_p e(t_k) + K_i \sum_{i=0}^{k} e(t_i)\Delta t + K_d \frac{e(t_k) - e(t_{k-1})}{\Delta t} + K$$

$$K_p, K_i, K_d, K = Constants\ and\ Offset\ Term$$

$$e(t_k) = Error\ at\ Time\ t_k$$

$$\Delta t = Time\ Elapsed\ from\ Previous\ Iteration$$

Notice that there is an extra offset term K added to the control function. This is the base PWM value/throttle command before the three control terms are applied to correct the error in the system.

To tune your PID, set the parameters (K_p, K_i, K_d, K) in *z_pid.yaml*.

To test your PID, run `python sim.py` on your base station or a department computer but not on your drone, since it requires a graphical user interface to visualize the output. The PID class in *student_pid_class.py* will automatically be used to control the simulated drone. The *up* and *down* arrow keys will change the setpoint, and *r* resets the simulation.

You will need *numpy*, *matplotlib*, and *yaml* to run the simulation. To install these dependencies, run `pip install numpy matplotlib pyyaml`.

Write brief answers to all exercises in `answers_pid.md`.

Problem 1: Implement an Idealized PID

Exercises

1. Implement the `step` method to return the constant K. At what value of K does the drone takeoff? Set K to 1300 for the remainder of the questions.
2. Implement the P term. What happens when K_p is 50? 500? 5000?
3. Implement the D term. Set K_p to zero. What happens when K_d is 50? 500? 5000?
4. Now tune K_p and K_d so that the drone comes to a steady hover. Describe the trade-off as you change the ratio of K_p to K_d.
5. Implement the I term and observe the difference between PD and PID control. What role does the I term play in this system? What happens when K_p and K_d are set to zero?
6. Implement the `reset` method and test its behavior. If implemented incorrectly, what problems can you anticipate reset causing?
7. Finally, tune the constants in your PID controller to the best of your abilities. When the setpoint is moving, the drone should chase the setpoint very closely. When the setpoint is still, the drone should converge exactly at the setpoint and not oscillate. Report your tuning values.

Problem 2: Tuning a PID with Latency

Now, we introduce latency! Run the simulation as `python sim.py -l 6` to introduce 24 milliseconds of latency (six steps of latency running at 25 hz).

Exercises

1. Tune the constants in your PID controller to the best of your abilities. The drone should chase the setpoint very closely, but will converge more slowly when the setpoint is still. Report your tuning values.
2. Compare your tuning values to the values you obtained in problem 1.
3. Explain the effect of latency on each control term.

Problem 3: Tuning a PID with Latency, Noise, and Drag

In the most realistic mode, you will tune a controller with latency, noise, and a drag coefficient. You can do this with the command line arguments `python sim.py -l 3 -n 0.5 -d 0.02` to be most realistic to real-world flight.

Exercises

1. Tune with these arguments to be as good as possible. Report your tuning values.
2. Compare your tuning values to the values from problems 1 and 2.

Run `python sim.py -h` to see the other simulator parameters. We encourage you to experiment with those and observe their effects on your controller.

After you finish this part of the project, make sure that you push the final versions of the files that you modified to your Github repo.

Part 2: Tuning

Write brief answers to all exercises in `answers_pid.md`.

Problem 1: Ziegler-Nichols Method

Imagine you are flying your drone and observing its flight for tuning it. Ideally, you would tune the K_p by slowly increasing its value between flights until you can see the drone moving up and down with uniform oscillations. The final K_p value that causes uniform oscillations is termed as K_u, the ultimate gain. While, the time difference between these two peaks during oscillations is termed as T_u, the ultimate period.

Exercises

1. Given K_u = 500, T_u = 10. Use your K_u and T_u values to compute K_p, K_i, and K_d using Ziegler-Nichols Method.

Problem 2: Flying with Velocity Control

In velocity control, we use planar velocity measure from the camera as the process variable. The keyboard keys are used to set the setpoints.

Exercises

1. Now suppose you are flying in velocity mode over a blank white poster board. How do you expect the drone to behave, and why will it behave this way? **hint**: Think about why we fly over a highly textured planar surface.

Project Checkoff

Functionality Check

1. The output of `step()` function in `student_pid_class.py` should be between 1100 and 1900.
2. The simulated drone should converge exactly at the setpoint and not oscillate for:

- Idealized PID, `python sim.py`,
- PID with latency, `python sim.py -l 6`,
- PID with latency, noise and drag, `python sim.py -l 3 -n 0.5 -d 0.02`

Questions

You will be asked to answer one of the following questions:

1. In `step()` function in `student_pid_class.py`, which lines of your code relate to P/I/D term and how do you calculate u(t)?
2. In `reset()` function in `student_pid_class.py`, which variables you updated and why?

Appendix A: Altitude Tuning

In this part, you will be transferring the altitude PID you created in part 1 onto your drone. You will then tune the PID gains on your drone as you did in the simulator.

Flying with Our PID

When you first flew the drone, in the Duckietown stack, you were directly setting the PWM “stick” levels with the keyboard and mouse. This translates to moving the throttle, roll, and pitch sticks on an RC receiver to fly the drone. There is a PID running on the flight controller that interprets these stick commands as desired rolls, pitches, and yaws, and then controls the motor spinning to achieve this.

Now, you will instead fly with a higher level of autonomy and use our PID controller to fly with velocity and height control. You will use the camera to estimate velocity with Optical Flow, and the range sensor to estimate height. Then the PID controller will set the PWMs to achieve these targets: either a fixed velocity of 0 m/s in X and Y, or a fixed velocity based on what keys are pressed on the keyboard. We are doing position control for the height, so it adjusts the throttle to maintain a target height above the ground using an altitude sensor.

Bring Down the Containers

First, bring down all the containers. Run `docker ps -a` and the `docker stop` on each container. Then reboot, and run `docker update --restart=no my-container` for each container that automatically restarts. Reboot again. When you are done, `docker ps -a` should show no containers running.

Update `pidrone_pkg`

Now update and rebuild the `pidrone_pkg` container. Run `git pull` in `pidrone_pkg` and then `rake build`, `rake create` and `rake start`. The build this time will be much faster, because it will not have to download everything from scratch.

Fly!

Follow the instructions in the operations manual in order to fly here.

Make sure this works before trying to tune your own PID.

STOP HERE

Please stop here for now. These instructions have not been tested with the new version of the drone.

You will now be using your altitude PID to control the height of the drone. To tune your altitude PID, you will first use the Ziegler-Nichols tuning method to generate an initial set of tuning parameters. You will then fine tune these parameters similar to how you tuned the drone in simulation.

To use your PID, you'll be running `student_pid_controller.py` instead of `pid_controller.py`. This will allow your PID to run alongside our planar PIDs; your PID will be responsible for keeping the drone flying steady vertically.

Setup

Change directories to `~/catkin_ws/src`. Run `git clone https://github.com/h2r/project-pid-implementation-yourGithubName.git`. In your repo, change "pidrone_project3_pid" to "project-pid-implementation-yourGithubName" in *package.xml* and "project(pidrone_project3_pid)" to "project(project-pid-implementation-yourGithubName)" in *CMakeLists.txt*. Also remove the msg folder, and comment out "add_message_files" in *CMakeLists.txt*. Then change directories back to `~/ws/` and run `catkin_make --pkg project-pid-implementation-yourGitHubName`.

OR

Use the `scp` command to transfer *student_pid_class.py*, *student_pid_controller.py*, and *z_pid.yaml* from the repo on your base station to the scripts folder of your drone (`~/catkin_ws/src/pidrone_pkg/scripts/`). In the instructions below, instead of using `rosrun`, you may use `python` to execute your scripts.

Change directories into `~/catkin_ws/src/pidrone_pkg` and modify *pi.screenrc* to start up with your altitude pid by changing `python pid_controller.py\n` to `rosrun project-pid-implementation-yourGitHubName student_pid_controller.py\n`. Prepare your drone to fly and then navigate to `4 of the screen. Press ctrl-c to quit student_pid_controller.

In this screen (`4), modify `~/catkin_ws/src/project-pid-implementation-yourGitHubName/z_pid.yaml` by setting $K$ to 1250 and the rest of the gain constants to 0. Now run `rosrun project-pid-implementation-yourGitHubName student_pid_controller.py` to fly with your altitude PID.

Exercises

Fly your drone and observe its flight. Tune K_p by slowly increasing its value between flights until you can see the drone moving up and down with uniform oscillations. Each time you will need to quit the controller, edit `~/catkin_ws/src/project-pid-implementation-yourGitHubName/z_pid.yaml`, and then run `rosrun project-pid-implementation-yourGitHubName student_pid_controller.py` again to use the new PID gains.

1. Record your final K_p value that causes uniform oscillations as K_u, the ultimate gain.
2. Fly your drone and pause the altitude graph on the web interface when you see two peaks. Find the time difference between these two peaks and record this value as T_u, the ultimate period.
3. Use your K_u and T_u values to compute K_p, K_i, and K_d. Refer to the equations in the Ziegler-Nichols section in the introduction to this project. Record these values and change *z_pid.yaml* accordingly.
4. Fly your drone with the set of tuning values generated by the Ziegler-Nichols method. Note that the Ziegler-Nichols method should enable safe flight, but will probably not control your drone's altitude very well! Empirically tune the gain constants in *z_pid.yaml* on your drone as you did in the simulator portion of this project. [2] Record your final tuning values.

Footnotes

2 Use the graph on the web interface to observe the drone's behavior as it oscillates around the 0.3m setpoint the drone's ability to hover at the setpoint. When observing the drone itself, try to get eye-level with the drone to just focus on the the altitude and ignore the planar motion; it is easier to focus on one axis at a time when tuning the PIDs. The planar axes can be re-tuned after you tune your altitude pid if need be.

Unscented Kalman Filter

The fundamental issue of state estimation impacts widespread robotics applications. Sensors give us an imperfect view of the world, as they tend to contain noise in their readings. Similarly, when a robot moves in the physical world, its motion has some amount of uncertainty, deviating randomly from the ideal model that we predict it might follow. Rather than ignoring these uncertainties and forming a naive state estimate, we will be harnessing the power of probability to yield mathematically optimal estimates using the Unscented Kalman Filter (UKF) algorithm. The Kalman Filter (KF) algorithm and its variants such as the UKF comprise part of the field of probabilistic robotics, which aims to account for uncertainties that the robot will inherently face as it interacts with the world with imperfect information. An entire course could be taught only on the topics of filtering and state estimation.

In this project, we give a high-level overview of the necessary foundations to understand the UKF algorithm. Then, you will implement a UKF with a simple one-dimensional model of the drone's motion to estimate its position and velocity along the vertical axis. Later on in the project, we will expand the model to three spatial dimensions.

Background

Motivation

Recall that control systems abstract away lower-level control and can be leveraged to build autonomous systems, in which $y(t)$ is the process variable, i.e. the measurement of behavior we care about controlling. For example, $y(t)$ would be the altitude of a drone in an altitude controller. Fig. 11 shows an example feedback control system of the form.

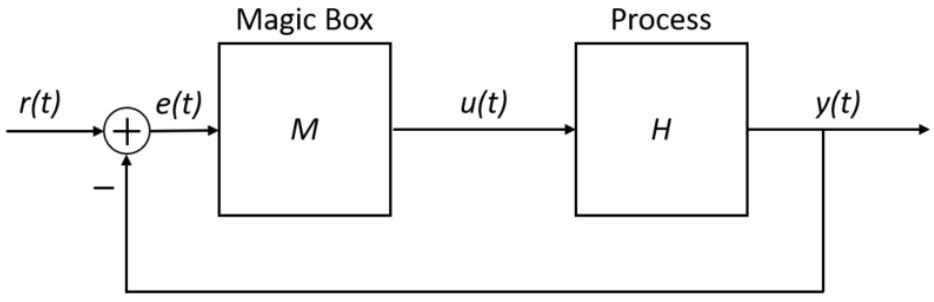


Fig. 11 A feedback control system

So far, we have naively been using raw sensor data as our $y(t)$ measurement. More specifically, we've been using the range reported by the drone's downward facing IR sensor as the "ground truth" measurement of altitude.

However, there are a **few problems** with this:

- In the real world, actual sensor hardware is not perfect – there's noise in sensor readings. For example, a $10 TOF sensor might report a range of 0.3m in altitude, when in reality the drone is at 0.25m. While a more expensive sensor would be less susceptible to noise, it would still not be perfect.
- The sensor readings may not really represent the behavior we wish to control. For example, it might seem like a downward facing TOF would be a good representation of a drone's altitude, but suppose the drone rolls a non-trivial amount $\theta or flies over a reflective surface$ (See Fig. 12).
- No one sensor may be enough to measure the $y(t)$ we really care about. For example, two 2D cameras would be needed to measure depth for a depth controller.

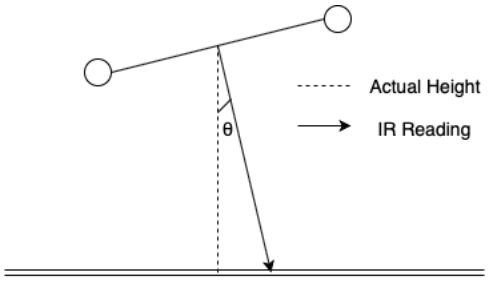


Fig. 12 An example of inaccurate sensor reading

These problems imply that we need a higher-level abstraction for our $y(t)$, namely one that accounts for: noise, robot motion, and sensor data. Let $\mathbf{x}_t$ be such an abstraction called state. For example, $\mathbf{x}_t = [altitude\ at\ time\ t]$ for the purposes of altitude control. To account for noise, we should consider the distribution of possible states at time t:

$$P(\mathbf{x}_t)$$

Furthermore, let $\mathbf{z}_{1:t}$ represent readings from all sensors from time 1 to t. Likewise, let $\mathbf{u}_{1:t}$ represent all robot motions from time 1 to t. Then we can account for robot motion and sensor data on our distribution via conditioning:

$$P(\mathbf{x}_t|\mathbf{z}_{1:t}, \mathbf{u}_{1:t})$$

Let this distribution be known as $bel(\mathbf{x}_t)$, i.e. the belief of the state of our dynamic system at time t. Suppose, hypothetically, that we knew the distribution $bel(\mathbf{x}_t)$ (though we haven't discussed how to determine it yet). Then we could change $y(t)$ in our control system from a naïve sensor reading to:

$$\mathbb{E}_{x_t \sim P(x_t|\mathbf{z}_{1:t}, \mathbf{u}_{1:t})}[x_t]$$

For a specific x_t in the state vector $\mathbf{x}_t$ (e.g. the altitude element in $\mathbf{x}_t$). Consequently, this would make $e(t)$ in a feedback control system:

$$e(t) = r(t) - y(t) = r(t) - \mathbb{E}_P[x_t] = desired - \mathbf{expected}\ actual$$

Understand $bel(\mathbf{x}_t)$

Although $bel(\mathbf{x}_t)$ seems to solve all our problems, it is unclear how to determine it explicitly. One way to do so is to decompose it into quantities that are easier to determine. Furthermore, it may prove helpful to make a few reasonable assumptions that will simplify the decomposition. In this vain, consider instead the distribution:

$$P(\mathbf{z}_t|\mathbf{x}_{0:t}, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})$$

which is called the **measurement model**. It is probably reasonable to assume that knowing the state at time t is enough to determine the distribution of our sensor data; knowing all previous states, motions, and sensor data probably won't add any new info. So our first simplification is a *Markov* assumption about the measurement model:

$$P(\mathbf{z}_t|\mathbf{x}_{0:t}, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1}) := P(\mathbf{z}_t|\mathbf{x}_t)$$

Likewise, consider the distribution:

$$P(\mathbf{x}_t|\mathbf{x}_{0:t-1}, \mathbf{u}_{1:t}, \mathbf{z}_{1:t})$$

Which is called the **motion model**. It is probably reasonable to assume that the state at time t only depends on the previous state and the motion that happened since. So another simplification is the *Markov* assumption about the motion model:

$$P(\mathbf{x}_t|\mathbf{x}_{0:t-1}, \mathbf{u}_{1:t}, \mathbf{z}_{1:t}) = P(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$$

Why did we bother with an aside about these distributions and assumptions? Because they allow us to decompose $bel(\mathbf{x}_t)$ as follows:

$$bel(\mathbf{x}_t) = P(\mathbf{x}_t|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) = \eta P(\mathbf{z}_t|\mathbf{x}_t) \int P(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) bel(\mathbf{x}_{t-1}) d\mathbf{x}_{t-1}$$

where

$$bel(\mathbf{x}_{t-1}) = P(\mathbf{x}_{t-1}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1})$$

and η is a constant. This decomposition gives rise to the **Bayes Filter algorithm** (see sections below). So effectively, it has reduced the challenge of determining the unknown (and difficult to figure out) distribution $P(\mathbf{x}_t|\mathbf{z}_{1:t}, \mathbf{u}_{1:t})$ into figuring out the distributions $P(z_t|x_t)$ and $P(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$, i.e. measurement and motion models respectively.

Fortunately, these distributions are easier to figure out than $bel(\mathbf{x}_t)$; we can either determine these distributions experimentally for a robot or we can make assumptions about the PDF class of these functions (see next section).

Bayes Filter Extension

So far we've reduced the challenge of determining $bel(\mathbf{x}_t)$ explicitly to instead determining $P(z_t|x_t)$ and $P(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t)$. One way to "determine" these is to purposefully assume they are distributions from well known parameterized PDF classes. A popular choice would the class of Gaussians, i.e. $\mathcal{N}(\mu, \sigma^2)$, which is a class parameterized by mean μ and variance σ^2; each (μ, σ^2) pair gives a different-shaped bell curve.

So we assume:

$$\begin{aligned} P(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) &= \mathcal{N}(\mathbf{x}_t|\mu = f(\mathbf{x}_{t-1}, \mathbf{u}_t), \sigma^2 = k_1) \\ P(\mathbf{z}_t|\mathbf{x}_t) &= \mathcal{N}(\mathbf{z}_t|\mu = g(\mathbf{x}_t), \sigma^2 = k_2) \\ bel(\mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_0|\mu = \mu_0, \sigma^2 = \sigma_0^2) \end{aligned}$$

where f and g are linear functions, k_1 and k_2 are some pre-determined variances. Together, these assumptions lead to $bel(\mathbf{x}_t)$ being a Gaussian as well:

$$bel(\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_t|\mu = \mu_t, \sigma^2 = \sigma_t^2)$$

Which gives rise to the Kalman Filter algorithm (see sections below). Note that the KF algorithm has the same form as the Baye's Filter algorithm (since the base derivation is the same), but the KF algorithm only needs to find μ_t and σ_t^2 at each time step t.

Practically, using a Kalman Filter means providing linear functions f and g as input. What are f and g?

- f is a function that captures the motion dynamics of a system. Simply put, f can be thought of as calculating the "predicted" $\mathbf{x}_t$ after motion. For example, suppose we have a drone that moves only horizontally. Let state $\mathbf{x}_t$ be the horizontal position x_pos at time t, $\mathbf{u}_t$ be the horizontal

velocity v_t (i.e. the control signal we send the drone), and $\widehat{\mathbf{x}}_t$ is the predicted state due to motion. Then:

$$\widehat{\mathbf{x}}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_t) = x\_pos_{t-1} + v_t \Delta t = [1]\mathbf{x}_{t-1} + [\Delta t]\mathbf{u}_t$$

- g is a function that transforms the state into something that can be compared to the sensor data $\mathbf{z}_t$. Simply put, $g(\mathbf{x}_t) = \widehat{\mathbf{z}}_t$, i.e. a "predicted" sensor reading based on the current state. For example: suppose a drone is rolled by θ, $\mathbf{z}_t = [r]$ for a range r reported by an IR sensor, and $\mathbf{x}_t$ is $[d, \theta]$ for an altitude of d and roll of θ:

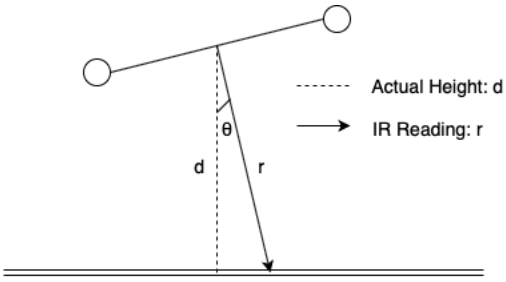


Fig. 13 An example of ir measurement calculation

Then a **non-linear** g woould be:

$$g(\mathbf{x}_t) = \widehat{\mathbf{z}}_t = \left[\frac{d}{\cos\theta}\right]$$

A key requirement of the Kalman Filter algorithm is that f and g need to be linear functions. This is necessary in order for the various Gaussians to multiply such that $bel(\mathbf{x}_t)$ is still a Gaussian. Under the assumption of linearity, the Kalman Filter is provably optimal.

However, most systems which have non-linearities. For example, the drone's update function involves nonlinear trigonometric functions like sin and cosine. Many people use the Extended Kalman Filter(EKF) algorithm instead. The EKF handles non-linear functions by basically doing a first-order Taylor expansion (to create a linear approximation) on f and g, then passing them to the Kalman Filter algorithm. To perform this approximation, the Jacobian matrix of f and g must be computed (e.g., the deriviative with respect to every input variable.) This Jacobian matrix must be derived and then implemented in code, an non-trivial effort.

Fortunately, another alternative is the Unscented Kalman Filter(UKF) algorithm, which is a sampling-based variant of the Kalman Filter. Like the EKF, the UKF can handle non-linear f and g. The UKF is not only simpler to implement than the EKF, it also performs better, although it is not provably optimal.

Background before UKF

Before we dive into the UKF, there are some foundations that we should build up:

- Estimating by averaging
- The Bayes Filter
- Gaussians
- The Kalman Filter

The basis for the Kalman Filter lies in probability; as such, if you want to better understand some of these probabilistic algorithms, you may find it helpful to brush up on probability. A useful reference on probability and uncertainty is [Jay03].

Since the UKF is an adaptation of the standard Kalman Filter, a lot of our discussion will apply to Kalman Filters in general.

Estimating by Averaging

Imagine a simple one-dimensional system in which your drone moves along the z-axis by adjusting its thrust. The drone has a downward-pointing infrared (IR) range sensor that offers you a sense of the drone's altitude, albeit with noise, as the sensor is not perfect. You are aware of this noise and want to eliminate it: after all, you know by empirical observation that your drone does not oscillate as much as the noisy IR range readings suggest, and that the average value of a lot of IR readings gets you a close estimate of the actual height. What can you do? A simple solution is to average recent range readings so as to average out the noise and smooth your estimate. One way to implement this is with a **moving average** that computes a weighted sum of the previous average estimate and the newest measurement.

$$\hat{\mathbf{x}}_t = \alpha\hat{\mathbf{x}}_{t-\Delta t} + (1-\alpha)\mathbf{z}_t$$

where $\hat{\mathbf{x}}_t$ is the weighted average of the drone's state (here, just its height) at time t computed by weighting the previous average $\hat{\mathbf{x}}_{t-\Delta t}$ by a scalar α between 0 and 1 and the new measurement $\mathbf{z}_t$ (here, just the raw IR reading) by $(1-\alpha)$. A higher α will result in a smoother estimate that gives more importance to the previous average than the new measurement; as such, a smoother estimate results in increased latency.

This approach works for many applications, but for our drone, we want to be able to know right away when it makes a sudden movement. Averaging the newest sensor reading with past readings suffers from latency, as it takes time for the moving average to approach the new reading. Ideally we would be able to cut through the noise of the IR sensor and experience no latency. We will strive for this kind of state estimation.

As a related thought experiment, imagine that you do not have control of the drone but are merely observing it from an outsider's perspective. In this scenario, there is one crucial bit of information that we lack: the **control input** that drives the drone. If we are controlling the drone, then presumably we are the ones sending it commands to, for example, accelerate up or down. This bit of information is useful (but not necessary) for the Bayes and Kalman Filters, as we will discuss shortly. Without this information, however, employing a moving average to estimate altitude is not a bad approach.

The Bayes Filter

To be able to get noise-reduced estimates with less latency than a via an averaging scheme, we can look to a probabilistic method known as the Bayes Filter, which forms the basis for Kalman filtering and a number of other probabilistic robotics algorithms. The idea with a Bayes Filter is to employ **Bayes' Theorem** and the corresponding idea of conditional probability to form probability distributions representing our belief in the robot's state (in our one-dimensional example, its altitude), given additional information such as the robot's previous state, a control input, and the robot's predicted state.

Say we know the drone's state $\mathbf{x}_{t-\Delta t}$ at the previous time step as well as the most recent control input $\mathbf{u}_t$, which, for example, could be a command to the motors to increase thrust. Then, we would like to find the probability of the drone being at a new state $\mathbf{x}_t$ given the previous state and the control input. We can express this with conditional probability as:

$$p(\mathbf{x}_t \mid \mathbf{u}_t, \mathbf{x}_{t-\Delta t})$$

This expression represents a **prediction** of the drone's state, also termed the **prior**, as it is our estimate of the drone's state before incorporating a measurement. Next, when we receive a measurement from our range sensor, we can perform an **update**, which looks at the measurement and the prior to form a **posterior** state estimate. In this step, we consider the probability of observing the measurement $\mathbf{z}_t$ given the state estimate $\mathbf{x}_t$:

$$p(\mathbf{z}_t \mid \mathbf{x}_t)$$

By Bayes' Theorem, we can then derive an equation for the probability of the drone being in its current state given information from the measurement:

$$p(\mathbf{x}_t \mid \mathbf{z}_t) = \frac{p(\mathbf{z}_t \mid \mathbf{x}_t)p(\mathbf{x}_t)}{p(\mathbf{z}_t)}$$

After a little more manipulation and combining of the predict and update steps, we can arrive at the Bayes Filter algorithm [TBF05]:

$$
\begin{aligned}
&\text{Bayes\_Filter}(bel(\mathbf{x}_{t-\Delta t}), \mathbf{u}_t, \mathbf{z}_t) : \\
&\quad \text{for all } \mathbf{x}_t \text{ do :} \\
&\quad\quad \bar{bel}(\mathbf{x}_t) = \int p(\mathbf{x}_t \mid \mathbf{u}_t, \mathbf{x}_{t-\Delta t}) bel(\mathbf{x}_{t-\Delta t}) \mathrm{d}\mathbf{x} \\
&\quad\quad bel(\mathbf{x}_t) = \eta p(\mathbf{z}_t \mid \mathbf{x}_t) \bar{bel}(\mathbf{x}_t) \\
&\quad \text{endfor} \\
&\quad \text{return } bel(\mathbf{x}_t)
\end{aligned}
$$

The filter calculates the probability of the robot being in each possible state $\mathbf{x}_t$ (hence the for loop). The prediction is represented as $\bar{bel}(\mathbf{x}_t)$ and embodies the prior belief of the robot's state after undergoing some motion, before incorporating our most recent sensor measurement. In the measurement update step, we compute the posterior $bel(\mathbf{x}_t)$. The normalizer η is equal to the reciprocal of $p(\mathbf{z}_t)$; alternatively, it can be computed by summing up $p(\mathbf{z}_t \mid \mathbf{x}_t)\bar{bel}(\mathbf{x}_t)$ over all states $\mathbf{x}_t$. This normalization ensures that the new belief $bel(\mathbf{x}_t)$ integrates to 1.

Gaussians

Bayes Filter is a useful concept, but often it is too difficult to compute the beliefs, particularly with potentially infinite state spaces. We want to then find a useful way to represent these probability distributions in a manner that accurately represents the real world while also making computation feasible. To do this, we exploit **Gaussian** functions.

We can represent the beliefs as Gaussian functions with a mean and a covariance matrix. Why? The state variables and measurements are random variables in that they can take on values in their respective sample spaces of all possible states and measurements. By the Central Limit Theorem, these random variables will be distributed normally (i.e., will form a Gaussian probability distribution) when you take a lot of samples. The Gaussian assumption is a strong one: think of a sensor whose reading fluctuates due to noise. If you take a lot of readings, most of the values should generally be concentrated in the center (the mean), with more distant readings occurring less frequently.

We use Gaussians because they are a good representation of how noise is distributed and because of their favorable mathematical properties. For one, Gaussians can be described by a mean and a covariance, which require less bookkeeping. Furthermore, Gaussian probability density functions added together result in another Gaussian, and products of two Gaussians (i.e., a joint probability distribution of two Gaussian distributions) are proportional to Gaussians [Jr18], which makes for less computation than if we were to use many samples from an arbitrary probability distribution. The consequence of these properties is that we can pass a Gaussian through a linear function and recover a Gaussian on the other side. Similarly, we can compute Bayes' Theorem with Gaussians as the probability distributions, and we find that the resulting probability distribution will be Gaussian [Jr18].

In the Bayes Filter, we talked about the predict and update steps. The prediction uses a **state transition function**, also known as a **motion model**, to propagate the state estimate (which we can represent as a Gaussian) forward in time to the next time step. If this function is linear, then the prior state estimate will also be Gaussian. Similarly, in the measurement update, we compute a new distribution using a measurement function to be able to compare the measurement and the state. If this function is linear, then we can get a Gaussian distribution for the resulting belief. We will elaborate on this constraint of linearity when we discuss the usefulness of the Unscented Kalman Filter, but for now you should be comfortable with the idea that using Gaussians to represent the drone's belief in its state is a helpful and important modeling assumption.

Multivariate Gaussians

Most of the time when we implement a Kalman Filter, we track more than one state variable in the state vector (we will go over what these terms mean and some intuition for why we do this in the next section). We also often receive more than one control and measurement input. What this means is that, as you may have noticed in the above equations which contain boldface vectors, we want to represent state estimates in more than one dimension in state space. As a result, our Gaussian representations of these state estimates will be **multivariate**. We won't go into much detail about this notion except to

point out, for example, that tracking multiple state variables with a multivariate Gaussian (represented as a vector of means and a covariance matrix) allows us to think about how different state variables are correlated. Again, we will cover this in greater detail as we talk about Kalman Filters in the following section—now you know, however, that there is motivation for using multi-dimensional Gaussians. If you want to learn more about this topic, we recommend [Labbe's textbook][Jr18], which also contains helpful graphics to understand what is going on with these Gaussians intuitively.

The Kalman Filter

High-Level Description of the Kalman Filter Algorithm

Recall from the Bayes Filter the procedure of carrying out predictions and measurement updates. The Kalman Filter, an extension of Bayes Filter with Gaussian assumptions on the belief distributions, aims to fuse measurement readings with predicted states. For example, if we know (with some degree of uncertainty) that the drone is moving upward, then this knowledge can inform us about the drone's position at the next time step. We can form a **prediction** of the drone's state at the next time step given the drone's current state and any control inputs that we give the drone. Then, at the next time step when we receive a new measurement, we can perform an **update** of our state estimate. In this update step, we look at the difference between the new measurement and the predicted state. This difference is known as the *residual*. Our new state estimate (referred to in literature as the *posterior*) lies somewhere between the predicted state, or *prior*, and the measurement; the scaling factor that accomplishes this is known as the *Kalman gain* [Jr18]. Fig. 14 depicts this process of prediction and measurement update.

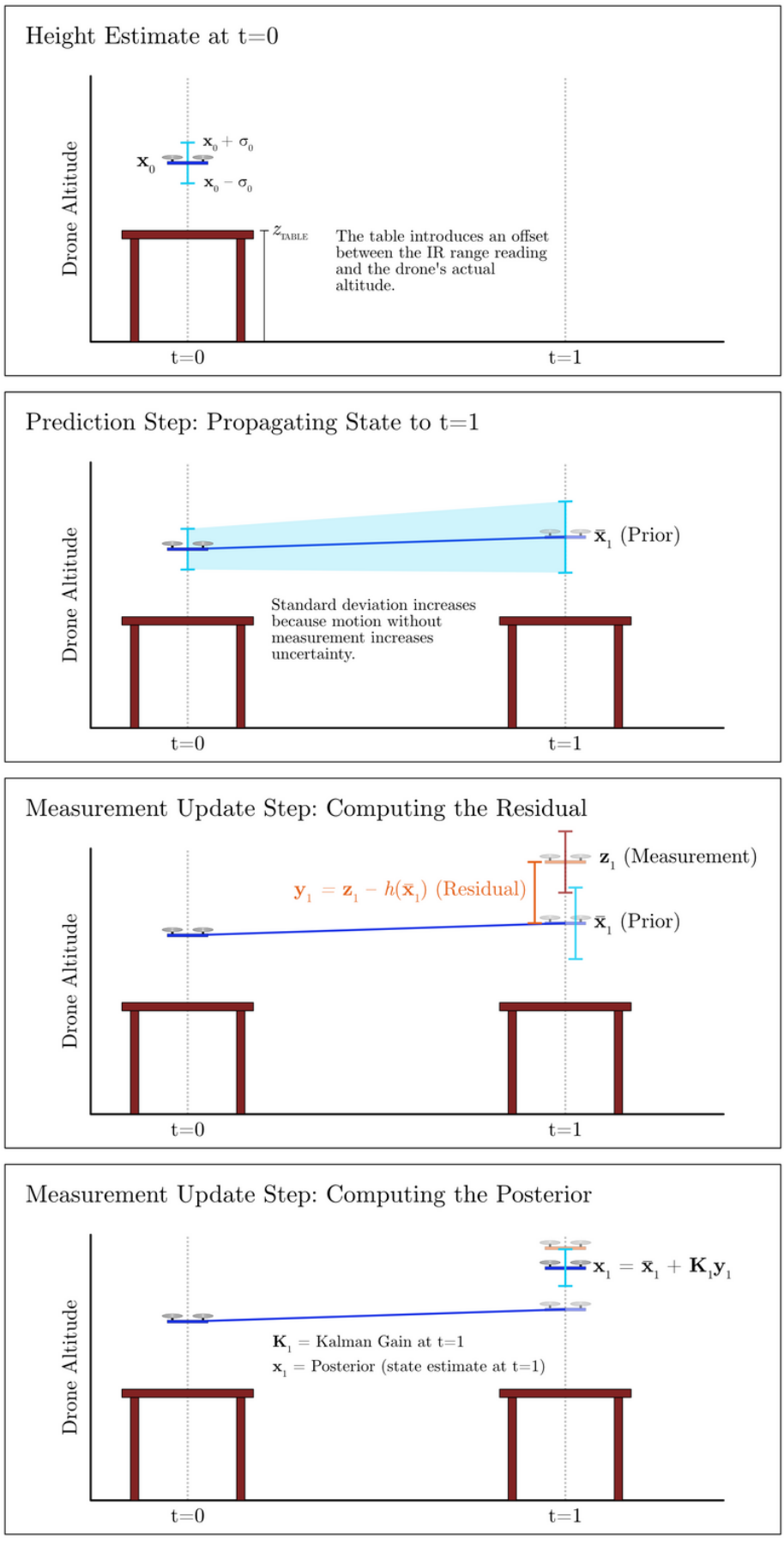


Fig. 14 Predict-Update Cycle for a Kalman Filter Tracking a Drone's Motion in One Dimension

State Vector and Covariance Matrix

The KF accomplishes its state estimate by tracking certain **state variables** in a state vector, such as position and velocity along an axis, and the **covariance matrix** corresponding to the state vector. In the first part of this project, your UKF's state vector will track the drone's position and velocity along the z-axis and will look like:

$$\mathbf{x}_t = \begin{bmatrix} z \\ \dot{z} \end{bmatrix}$$

where z and $\dot{z}$ are the position and velocity of the drone along the z-axis, respectively. Fig. 14 depicts a contrived example in which the drone is hovering above a table. We will describe why we show this contrived example shortly when we discuss the measurement function, but for now, you should focus on the fact that we want to know the drone's position along the z-axis (where the ground is 0 meters) and its velocity.

Why did we decide to track not only position but also velocity? What if we were only interested in the drone's position along the z-axis? Well, while in some cases we might only be immediately interested in, say, knowing the robot's position, the addition of $\dot{z}$ is also important. $\dot{z}$ is known as a *hidden variable*: we do not have a sensor that allows us to measure this quantity directly. That said, keeping track of this variable allows us to form better estimates about z. Why? Since position and velocity are correlated quantities, information about one quantity can inform the other. If the drone has a high positive velocity, for instance, we can be fairly confident that its position at the next time step will be somewhere in the positive direction relative to its previous position. The covariances between position and velocity allow for a reasonable estimate of velocity to be formed—as does any information about acceleration, for example. Chapter 5.7 of Labbe's textbook [Jr18] describes the importance of this correlation between state variables such as position and velocity. In addition, as you will see when we discuss the state transition model of a Kalman Filter, the control input impacts the prior state estimate. In this particularly instance, the control input is a linear acceleration value, which can be integrated to provide information about velocity.

The state vector tracks the mean μ_t of each state variable, which—as we noted in the section on Gaussians—we assume is normally distributed about μ_t. To characterize the uncertainty in this state estimate, we use an $n \times n$ covariance matrix where n is the size of the state vector. For this state vector, then, we define the covariance matrix as:

$$\mathbf{P}_t = \begin{bmatrix} \sigma_z^2 & \sigma_{z,\dot{z}} \\ \sigma_{\dot{z},z} & \sigma_{\dot{z}}^2 \end{bmatrix}$$

where $\sigma_z^2 = \mathrm{Var}\,(z)$, for example, denotes the variance in the position estimates and $\sigma_{\dot{z},z} = \sigma_{z,\dot{z}} = \mathrm{Cov}\,(z, \dot{z})$ denotes the covariance between the position and velocity estimates. As mentioned above, position and velocity are typically positively correlated, as a positive velocity indicates that the drone will likely be at a more positive position at the next time step.

The first frame of Fig. 14 illustrates a state estimate and the standard deviation of that height estimate.

State Transition Model for the Prediction Step

The part of the KF that computes a predicted state $\bar{\mathbf{x}}_t$ is known as the state transition function. The prediction step of the UKF uses the state transition function to propagate the current state at time $t - \Delta t$ to a prediction of the state at the next time step, at time t. In standard Kalman Filter literature for linear systems, this transition function can be expressed with two matrices: a state transition matrix $\mathbf{A}_t$ and a control function $\mathbf{B}_t$ that, when multiplied with the current state vector $\mathbf{x}_{t-\Delta t}$ and with the control input vector $\mathbf{u}_t$, respectively, sum together to output the prediction of the next state.

$$\bar{\mathbf{x}}_t = \mathbf{A}_t \mathbf{x}_{t-\Delta t} + \mathbf{B}_t \mathbf{u}_t$$

We give $\mathbf{A}_t$ and $\mathbf{B}_t$ each a subscript t to indicate that these matrices can vary with time. Often, these matrices will include one or more Δt terms in order to properly propagate the state estimate forward in time by that amount. If our control input, for example, comes in at a varying frequency, then the time step Δt will change.

More generally, in nonlinear systems—where the UKF is useful, which we will describe later—a single transition function $g(\mathbf{x}_{t-\Delta t}, \mathbf{u}_t, \Delta t)$ can express the prediction of what the next state will be given the current state estimate $\mathbf{x}_{t-\Delta t}$, the control input $\mathbf{u}_t$, and the time step Δt [TellexBrownLupashin18]. For robotic systems such as the Duckiedrone, the state transition function often involves using kinematic equations to form numerical approximations of the robot's motion in space.

$$\bar{\mathbf{x}}_t = g(\mathbf{x}_{t-\Delta t}, \mathbf{u}_t, \Delta t)$$

The control input that you will use for this project is the linear acceleration along the z-axis $\ddot{z}$ being output by the IMU. While the distinction between this control input and other measurements might seem vague, we can think of these acceleration values as being commands that we set when we control the drone. Indeed, since we control the drone's throttle and thus the downward force of the propellers, we do control the drone's acceleration by Newton's Second Law of Motion:

$$\mathbf{F} = m\mathbf{a}$$

That said, even though in practice people do successfully use IMU accelerations as control inputs, research [EE15] indicates that in certain cases it may be better to use IMU data in the measurement update step; this is an example of a design decision whose performance may depend on the system you are modeling. We choose to use the IMU's acceleration as the control input $\mathbf{u}_t$:

$$\mathbf{u}_t = [\ddot{z}]$$

Expressing Newton's Second Law in terms of our control input, we have:

$$\mathbf{F} = m\ddot{z}\hat{z}$$

which denotes that the net force $\mathbf{F}$ acting on the drone is equal to its mass m (assumed to be constant) multiplied by the acceleration $\ddot{z}$ in the $\hat{z}$ direction (i.e., along the z-axis).

The second frame of Fig. 14 shows the result of the state transition function: the drone's state estimate has been propagated forward in time, and in doing so, the uncertainty in its state has increased, since its motion model has some degree of uncertainty and the new measurement has not yet been incorporated.

Measurement Function

After the prediction step of the KF comes the measurement update step. When the drone gets a new measurement from one of its sensors, it should compute a new state estimate based on the prediction and the measurement. In the z-axis motion model for the first part of this project, the sensor we consider is the infrared (IR) range sensor. We assume that the drone has no roll and pitch, which means that the IR reading directly corresponds to the drone's altitude. The measurement vector, then, is defined as:

$$\mathbf{z}_t = [r]$$

where r is the IR range reading.

In our contrived example shown in Fig. 14, however, the IR range reading does not directly correspond to the drone's altitude: there is an offset due to the height of the table z_{TABLE}.

As depicted in the third frame of Fig. 14, part of the measurement update step is the computation of the residual $\mathbf{y}_t$. This value is the difference between the measurement and the predicted state. However, the measurement value lives in *measurement space*, while the predicted state lives in *state space*. For your drone's particular 1D example (without the table beneath the drone), the measurement and the position estimate represent the same quantity; however, in more complicated systems such as the later part of this project in which you will be implementing a UKF to track multiple spatial dimensions, you will find that the correspondence between measurement and state may require trigonometry. Also, since the sensor measurement often only provides information pertaining to part of the state vector, we cannot always transform a measurement into state space. Chapter 6.6.1 of Labbe's textbook [Jr18] describes the distinction between measurement space and state space.

Consequently, we must define a measurement function $h(\bar{\mathbf{x}}_t)$ that transforms the prior state estimate into measurement space. (For the linear Kalman Filter, this measurement function can be expressed as a matrix $\mathbf{H}_t$ that gets multiplied with $\bar{\mathbf{x}}_t$.) As an example, our diagrammed scenario with the table requires the measurement function to account for the height offset. In particular, $h(\bar{\mathbf{x}}_t)$ would return a 1×1 matrix whose singular element is the measurement you would expect to get with the drone positioned at a height given by the z value in the prior $\bar{\mathbf{x}}_{t,z}$:

$$h(\bar{\mathbf{x}}_t) = \left[\bar{\mathbf{x}}_{t,z} - z_{\text{TABLE}}\right]$$

This transformation allows us to compute the residual in measurement space with the following equation:

$$\mathbf{y}_t = \mathbf{z}_t - h(\bar{\mathbf{x}}_t)$$

Once the residual is computed, the posterior state estimate is computed via the following equation:

$$\mathbf{x}_t = \bar{\mathbf{x}}_t + \mathbf{K}_t \mathbf{y}_t$$

where $\mathbf{K}_t$ is the Kalman gain that scales how much we "trust" the measurement versus the prediction. Once this measurement-updated state estimate $\mathbf{x}_t$ is calculated, the filter continues onto the next predict-update cycle.

The fourth frame of Fig. 14 illustrates this fusion of prediction and measurement in which a point along the residual is selected for the new state estimate by the Kalman gain. The Kalman gain is determined mathematically by taking into account the covariance matrices of the motion model and of the measurement vector. While we do not expect you to know exactly how to compute the Kalman gain, intuitively it is representative of a ratio between the uncertainty in the prior and the uncertainty in the newly measured value.

At a high level, that's the Kalman Filter algorithm! Below is the general linear Kalman Filter algorithm [TellexBrownLupashin18] [TBF05] [Jr18] written out in pseudocode. We include Δt as an argument to the $\text{predict}()$ function since it so often is used there. We use boldface vectors and matrices to describe this algorithm for the more general multivariate case in which we are tracking more than one state variable, we have more than one measurement variable, et cetera. We also have not yet introduced you to the $\mathbf{Q}_t$ and $\mathbf{R}_t$ matrices; you will learn about them later in this project when you implement your first filter. Also, we did not previously mention some of the equations written out in this algorithm (e.g., the computation of the Kalman gain); fret not, however, as you are not responsible for understanding all of the mathematical details. Nonetheless, we give you this algorithm for reference and for completeness. As an exercise, you might also find it helpful to compare the KF algorithm to the Bayes Filter algorithm written above.

function $\text{predict}(\mathbf{x}_{t-\Delta t}, \mathbf{P}_{t-\Delta t}, \mathbf{u}_t, \Delta t)$
  `// Compute predicted mean`
  $\bar{\mathbf{x}}_t = \mathbf{A}_t \mathbf{x}_{t-\Delta t} + \mathbf{B}_t \mathbf{u}_t$
  `// Compute predicted covariance matrix`
  $\bar{\mathbf{P}}_t = \mathbf{A}_t \mathbf{P}_{t-\Delta t} \mathbf{A}_t^\top + \mathbf{Q}_t$
  return $\bar{\mathbf{x}}_t, \bar{\mathbf{P}}_t$

function $\text{update}(\bar{\mathbf{x}}_t, \bar{\mathbf{P}}_t, \mathbf{z}_t)$
  `Compute the residual in measurement space`
  $\mathbf{y}_t = \mathbf{z}_t - \mathbf{H}_t \bar{\mathbf{x}}_t$
  `Compute the Kalman gain`
  $\mathbf{K}_t = \bar{\mathbf{P}}_t \mathbf{H}_t^\top \left(\mathbf{H}_t \bar{\mathbf{P}}_t \mathbf{H}_t^\top + \mathbf{R}_t\right)^{-1}$
  `Compute the mean of the posterior state estimate`
  $\mathbf{x}_t = \bar{\mathbf{x}}_t + \mathbf{K}_t \mathbf{y}_t$

$$
\begin{aligned}
&\text{Compute the covariance of the posterior state estimate} \\
&\mathbf{P}_t = (\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\bar{\mathbf{P}}_t \\
&\text{return } \mathbf{x}_t, \mathbf{P}_t \\
&\\
&\textbf{function } \text{kalman\_filter}(\mathbf{x}_{t-\Delta t}, \mathbf{P}_{t-\Delta t}) \\
&\quad \mathbf{u}_t = \text{get\_control\_input}() \\
&\quad \Delta t = \text{compute\_time\_step}() \\
&\quad \bar{\mathbf{x}}_t, \bar{\mathbf{P}}_t = \text{predict}(\mathbf{x}_{t-\Delta t}, \mathbf{P}_{t-\Delta t}, \mathbf{u}_t, \Delta t) \\
&\quad \mathbf{z}_t = \text{get\_sensor\_data}() \\
&\quad \mathbf{x}_t, \mathbf{P}_t = \text{update}(\bar{\mathbf{x}}_t, \bar{\mathbf{P}}_t, \mathbf{z}_t) \\
&\quad \text{return } \mathbf{x}_t, \mathbf{P}_t
\end{aligned}
$$

[EE15] A. T. Erdem and A. Ö. Ercan. Fusing inertial sensor data in an extended kalman filter for 3d camera tracking. *IEEE Transactions on Image Processing*, 24(2):538–548, Feb 2015. doi:10.1109/TIP.2014.2380176.

[Jay03] Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.

[Jr18](1,2,3,4,5,6,7) Roger R Labbe Jr. Kalman and bayesian filters in python. Published as a Jupyter Notebook hosted on GitHub at https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python and in PDF form at https://drive.google.com/file/d/0By_SW19c1BfhSVFzNHc0SjduNzg/view?usp=sharing (accessed August 29, 2018), May 2018.

[TBF05](1,2) Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN 0262201623.

[TellexBrownLupashin18](1,2) S. Tellex, A. Brown, and S. Lupashin. Estimation for Quadrotors. *ArXiv e-prints*, August 2018. arXiv:1809.00037.

The Unscented Kalman Filter: Nonlinear State Estimation

Limitations of the Standard (Linear) Kalman Filter

So far, we have discussed the standard Kalman Filter algorithm. However, we have not mentioned its limitations. The standard Kalman Filter assumes that the system is both *linear* and *Gaussian*. In other words, the uncertainties in the motion and measurement models are assumed to be normally distributed about a mean in order to produce optimal estimates, which allows us to represent the state estimate as a Gaussian with mean and variance. For many systems, the Gaussian assumption is a good one. Intuitively, one can imagine that a sensor's noise, for example, varies more or less symmetrically about a true mean value, with larger deviations occurring less frequently.

The greater constraint, however, is the assumption that the system is linear. What we mean by this is that the state transition function and measurement function are linear functions, and as a result, when we pass Gaussian distributions through these functions, the output remains Gaussian or proportional to a Gaussian. An arbitrary nonlinear function, on the other hand, will not output another Gaussian or scaled Gaussian, which is a problem since so much of the Kalman Filter math depends on the state estimate being Gaussian. The Unscented Kalman Filter was expressly designed to robustly handle this issue of nonlinearity.

In this project's z-axis UKF, the functions are linear, so indeed a standard Kalman Filter would suffice. However, for the second UKF that you will be implementing, there are nonlinearities due to the drone's orientation in space. To make the transition easier from the first part to the second part of this project, we are asking you to implement a UKF even for a linear system. The UKF estimates will be the same as a KF; the only downsides might be code complexity and computation time. That said, you will be using a Python library called FilterPy (written by Labbe, author of *Kalman and Bayesian Filters in Python* [Jr18] that handles and hides most of the filtering math anyway.

You might also be wondering what the term “unscented” has to do with a Kalman Filter that applies to nonlinear systems. There is no greater technical meaning to the word; the inventor claims it is an arbitrary choice that resulted from his catching a glimpse of a coworker’s deodorant while trying to come up with a name for his filter [uhl].

Underlying Principle of the UKF

To handle the nonlinearities, the UKF uses a sampling scheme. An alternative to the UKF known as the Extended Kalman Filter (EKF) uses Jacobians to linearize the nonlinear equations, but the UKF takes a deterministic sampling approach that in many cases results in more accurate estimates and is a simpler algorithm to implement [TellexBrownLupashin18].

The UKF uses a function to compute so-called **sigma points**, which are the sample points to pass through the state transition and measurement functions. Each sigma point also has corresponding **weights** for the sample’s mean and covariance. The sigma points are generated so that there are $2n + 1$ of them, where n is the size of the state vector. Imagine a one-dimensional state vector, for example, which we represent as a single-variable Gaussian. In this instance, $2(1) + 1 = 3$ sigma points are chosen. One of these points is the mean of the Gaussian, and the two other points are symmetric about the mean on either side. The exact distance of these points from the mean sigma point will vary depending on parameters passed into the sigma point function, but we do not expect you to worry about these parameters. The idea, though, is that these $2(1) + 1 = 3$ sigma points and their weights are sufficiently representative of the Gaussian distribution.

Next, these points that represent the Gaussian state estimate are passed through a nonlinear function (i.e., the state transition or measurement functions), which can scatter the points arbitrarily. We then want to recover a Gaussian from these scattered points, and we do so by using the **unscented transform**, which computes a new mean and covariance matrix. To compute the new mean, the unscented transform calculates a weighted sum of each sigma point with its associated sample mean weight.

UKF in the Prediction Step

The UKF formulates the prior state estimate by specifying a set of sigma points $\boldsymbol{\mathcal{X}}_{t-\Delta t}$ according to the current state estimate and then propagating these points through the state transition function to yield a new set of sigma points $\boldsymbol{\mathcal{Y}}_t$, which are passed through the unscented transform to produce the prior state estimate.

UKF in the Update Step

Below is the algorithm for the Unscented Kalman Filter [TellexBrownLupashin18], [Jr18]. Note that the sigma point weights denoted by $W_i^{(m)}$ and $W_i^{(c)}$ can be computed as part of a number of sigma point algorithms. We will use Van der Merwe’s scaled sigma point algorithm to compute the sigma points and weights [MW03], [Jr18]. The sigma points get computed at each prediction, whereas the weights can be computed just once upon filter initialization.

$$
\begin{aligned}
&\textbf{function } \text{predict}(\mathbf{x}_{t-\Delta t}, \mathbf{P}_{t-\Delta t}, \mathbf{u}_t, \Delta t) \\
&\quad \texttt{// Compute 2n+1 sigma points given the most recent state estimate} \\
&\quad \boldsymbol{\mathcal{X}}_{t-\Delta t} = \text{compute\_sigma\_points}(\mathbf{x}_{t-\Delta t}, \mathbf{P}_{t-\Delta t}) \\
&\quad \texttt{// Propagate each sigma point through the state transition function} \\
&\quad \text{for } (i = 0; i \leq 2n; i\text{++}): \\
&\quad\quad \boldsymbol{\mathcal{Y}}_{i,t} = g(\boldsymbol{\mathcal{X}}_{i,t-\Delta t}, \mathbf{u}_t, \Delta t) \\
&\quad \texttt{// Compute the prior mean and covariance by passing the sigma} \\
&\quad \texttt{// points through the unscented transform (the next two lines)} \\
&\quad \bar{\mathbf{x}}_t = \textstyle\sum_{i=0}^{2n} W_i^{(m)} \boldsymbol{\mathcal{Y}}_{i,t} \\
&\quad \bar{\mathbf{P}}_t = \textstyle\sum_{i=0}^{2n} W_i^{(c)} \left(\boldsymbol{\mathcal{Y}}_{i,t} - \bar{\mathbf{x}}_t\right)\left(\boldsymbol{\mathcal{Y}}_{i,t} - \bar{\mathbf{x}}_t\right)^\mathsf{T} + \mathbf{Q}_t \\
&\quad \text{return } \bar{\mathbf{x}}_t, \bar{\mathbf{P}}_t
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{function } \text{update}(\bar{\mathbf{x}}_t, \bar{\mathbf{P}}_t, \mathbf{z}_t) \\
&\quad \texttt{// Compute the measurement sigma points} \\
&\quad \text{for } (i = 0; i \leq 2n; i\text{++}) : \\
&\quad\quad \boldsymbol{\mathcal{Z}}_{i,t} = h(\boldsymbol{\mathcal{Y}}_{i,t}) \\
&\quad \texttt{// Compute the mean and covariance of the measurement} \\
&\quad \texttt{// sigma points by passing them through the unscented} \\
&\quad \texttt{// transform (the next two lines)} \\
&\quad \boldsymbol{\mu}_z = \sum_{i=0}^{2n} W_i^{(m)} \boldsymbol{\mathcal{Z}}_{i,t} \\
&\quad \mathbf{P}_z = \sum_{i=0}^{2n} W_i^{(c)} \left(\boldsymbol{\mathcal{Z}}_{i,t} - \boldsymbol{\mu}_z\right)\left(\boldsymbol{\mathcal{Z}}_{i,t} - \boldsymbol{\mu}_z\right)^\mathsf{T} + \mathbf{R}_t \\
&\quad \mathbf{y}_t = \mathbf{z}_t - \boldsymbol{\mu}_z \\
&\quad \texttt{// Compute the cross covariance between state and measurements} \\
&\quad \mathbf{P}_{xz} = \sum_{i=0}^{2n} W_i^{(c)} \left(\boldsymbol{\mathcal{Y}}_{i,t} - \bar{\mathbf{x}}_t\right)\left(\boldsymbol{\mathcal{Z}}_{i,t} - \boldsymbol{\mu}_z\right)^\mathsf{T} \\
&\quad \texttt{// Compute the Kalman gain} \\
&\quad \mathbf{K}_t = \mathbf{P}_{xz}\mathbf{P}_z^{-1} \\
&\quad \texttt{// Compute the mean of the posterior state estimate} \\
&\quad \mathbf{x}_t = \bar{\mathbf{x}}_t + \mathbf{K}_t\mathbf{y}_t \\
&\quad \texttt{// Compute the covariance of the posterior state estimate} \\
&\quad \mathbf{P}_t = \bar{\mathbf{P}}_t - \mathbf{K}_t\mathbf{P}_z\mathbf{K}_t^\mathsf{T} \\
&\quad \text{return } \mathbf{x}_t, \mathbf{P}_t
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{function } \text{unscented\_kalman\_filter}(\mathbf{x}_{t-\Delta t}, \mathbf{P}_{t-\Delta t}) \\
&\quad \mathbf{u}_t = \text{get\_control\_input}() \\
&\quad \Delta t = \text{compute\_time\_step}() \\
&\quad \bar{\mathbf{x}}_t, \bar{\mathbf{P}}_t = \text{predict}(\mathbf{x}_{t-\Delta t}, \mathbf{P}_{t-\Delta t}, \mathbf{u}_t, \Delta t) \\
&\quad \mathbf{z}_t = \text{get\_sensor\_data}() \\
&\quad \mathbf{x}_t, \mathbf{P}_t = \text{update}(\bar{\mathbf{x}}_t, \bar{\mathbf{P}}_t, \mathbf{z}_t) \\
&\quad \text{return } \mathbf{x}_t, \mathbf{P}_t
\end{aligned}
$$

[uhl] First-hand: the unscented transform. Engineering and Technology History Wiki. Contributions from Uhlmann, Jeffrey. Accessed August 31, 2018. URL: https://ethw.org/First-Hand:The_Unscented_Transform.

[Jr18](1,2,3) Roger R Labbe Jr. Kalman and bayesian filters in python. Published as a Jupyter Notebook hosted on GitHub at https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python and in PDF form at https://drive.google.com/file/d/0By_SW19c1BfhSVFzNHc0SjduNzg/view?usp=sharing (accessed August 29, 2018), May 2018.

[MW03] Rudolph Van Der Merwe and Eric Wan. Sigma-point kalman filters for probabilistic inference in dynamic state-space models. In *In Proceedings of the Workshop on Advances in Machine Learning*. 2003.

[TellexBrownLupashin18](1,2) S. Tellex, A. Brown, and S. Lupashin. Estimation for Quadrotors. *ArXiv e-prints*, August 2018. arXiv:1809.00037.

Steps to Design and Implement a Kalman Filter on a Robot

To apply a Kalman Filter (linear KF or UKF) to a specific robot, there are certain parts of the algorithm that we need to define.

1. **State Vector:** The first aspect of the KF design process specific to the robot application is the selection of state variables to track in the state vector.
2. **Motion Model:** The motion model of the robot demands careful thought when designing a KF, as it determines the state transition function.
3. **Measurement Model:** The robot's sensor suite plays a significant role in how the robot forms state estimates. Its sensors determine the measurement function.
4. **Process Noise and Measurement Covariance Matrices:** The process noise and measurement covariance matrices must be determined from the motion model and sensor suite, respectively.
5. **Initialization of the Filter:** The filter must have initial values on which to perform the predictions and measurement updates.

6. **Asynchronous Inputs:** Sometimes, the KF has to be adapted to handle asynchronous inputs from real-world sensors, whose data rates are not strictly fixed.
7. **Tuning and Testing:** Finally, once a filter is implemented, it is a good idea to tune and test it in simulation and then on the real robot, quantifying its performance if possible.

We will be going over these design decisions and implementation details step-by-step as you implement your filters in one and three spatial dimensions on the drone.

Project 4: UKF

2D UKF Design and Implementation

Fig. 15 2D UKF Height Estimates Visualized in the JavaScript Web Interface

It is time for you to design and implement a 2D UKF specific to the Duckiedrone! For the implementation, we will have you use the Python library FilterPy, which abstracts away most of the nitty-gritty math. If needed, you can refer to the FilterPy documentation and source code here.

Handin

Use this link to generate a Github repo for this project. Clone the directory to your computer `git clone https://github.com/h2r/project-ukf-2020-yourGithubName.git` This will create a new folder.

Commit and push your changes before the assignment is due. This will allow us to access the files you pushed to Github and grade them accordingly. If you commit and push after the assignment deadline, we will use your latest commit as your final submission, and you will be marked late.

```
cd project-ukf-2020-yourGithubName
git add -A
git commit -a -m 'some commit message. maybe handin, maybe update'
git push
```

Note that assignments will be graded anonymously, so please don't put your name or any other identifying information on the files you hand in.

Design and Implement the 2D Filter

This part of the project has **two deliverables** in your repository, which are to be accessed and submitted via GitHub Classroom:

1. A LaTeX PDF document `ukf2d_written_solutions.pdf`, generated from `ukf2d_written_solutions.tex`, with the answers to the UKF design and implementation questions.
2. Your implementation of the UKF written in the `state_estimators/student_state_estimator_ukf_2d.py` stencil code. In this stencil code file, we have placed `TODO` tags describing where you should write your solution code to the relevant problems.

In addition to implementing the UKF in code, we want you to learn about the design process, much of which occurs outside of the code that will run the UKF. Plus, we have some questions we want you to answer in writing to demonstrate your understanding of the UKF. Hence, you will be writing up some of your solutions in $\LaTeX$. We are having you write solutions in $\LaTeX$ because it will in particular enable you to write out some of the UKF math in a clear (and visually appealing!) format. In these documents, please try to follow our math notation wherever applicable.

Task: From your repository, open up the `ukf2d_written_solutions.tex` file in your favorite $\LaTeX$ editor. This could be in Overleaf, your Brown CS department account, or locally on your own computer. *Before submitting your document, please make sure it is compiled into a PDF. If you are having trouble with $\LaTeX$, please seek out the help of a TA.*

State Vector

For this part of the project, we are going to track the drone's position and velocity along the z-axis:

$$\mathbf{x}_t = \begin{bmatrix} z \\ \dot{z} \end{bmatrix}$$

State Transition Function

For this UKF along the z-axis, your state transition function will take into account a control input $\mathbf{u}$ defined as follows:

$$\mathbf{u}_t = [\ddot{z}]$$

$\ddot{z}$ is the linear acceleration reading along the z-axis provided by the IMU.

Task (Written Section 1.2.2): Implement the state transition function $g(\mathbf{x}_{t-\Delta t}, \mathbf{u}_t, \Delta t)$ by filling in the template given in Section 1.2.2 of `ukf2d_written_solutions.tex` with the correct values to propagate the current state estimate forward in time. Remember that for the drone, this involves kinematics (hint: use the constant acceleration kinematics equations). Since there is a notion of transitioning the state from the previous time step, this function will involve the variable Δt.

Task: Translate the state transition function into Python by filling in the `state_transition_function()` method in `state_estimators/student_state_estimator_ukf_2d.py`. Follow the "TODO"s there. Note the function's type signature for the inputs and outputs.

Measurement Function

At this stage, we are only considering the range reading from the IR sensor for the measurement update step of the UKF, so your measurement vector $\mathbf{z}_t$ will be the following:

$$\mathbf{z}_t = [r]$$

Task (Written Section 1.3.2): In `ukf2d_written_solutions.tex`, implement the measurement function $h(\bar{\mathbf{x}}_t)$ to transform the prior state estimate into measurement space. For this model's state vector and measurement vector, $h(\bar{\mathbf{x}}_t)$ can be implemented as a 1×2 matrix that is multiplied with the 2×1 state vector, outputting a 1×1 matrix: the same dimension as the measurement vector $\mathbf{z}_t$, which allows for the computation of the residual.

Task: As before, translate the measurement function into code, this time by filling in the `measurement_function()` method. Follow the "TODO"s there. Note the function's type signature for the inputs and outputs.

Process Noise and Measurement Covariance Matrices

The process noise covariance matrix $\mathbf{Q}_t$ represents how uncertain we are about our motion model. It needs to be determined for the prediction step, but you do not need to determine this yourself, as this matrix can be notoriously difficult to ascertain. Feasible values for the elements of $\mathbf{Q}_t$ are provided in the code.

On the other hand, the measurement noise covariance matrix $\mathbf{R}_t$ has a more tangible meaning: it represents the variance in our sensor readings, along with covariances if sensor readings are correlated. For our 1D measurement vector, this matrix just contains the variance of the IR sensor.

The interplay between $\mathbf{Q}_t$ and $\mathbf{R}_t$ dictates the value of the Kalman gain $\mathbf{K}_t$, which scales our estimate between the prediction and the measurement.

Task: Characterize the noise in the IR sensor by experimentally collecting data from your drone in a stationary setup and computing its variance. To do so, prop the drone up so that it is stationary and its IR sensor is about 0.3 m from the ground, pointing down, unobstructed. To collect the range data, execute the following commands on your drone:

Navigate to pidrone_pkg and start the code:

```
$ roscd pidrone_pkg
$ ./start.sh
```

After navigating to a free screen, echo the infrared ROS topic and extract just the range value. To automatically log a lot of IR range readings, you must redirect standard output to a file like so:

```
$ rostopic echo /pidrone/infrared/range > ir_data.txt
```

We have provided a script `ir_sample_variance_calculation.py` that reads in the range readings from the file (so make sure this file is named `ir_data.txt` and is in the same directory as `ir_sample_variance_calculation.py`), computes the sample variance, and plots the distribution of readings using `matplotlib`. If you want to run this on your drone, then you will have to ensure that your `ssh` client has the capability to view pop-up GUI windows in order to view the plot. If you have XQuartz installed on your base station, for example, then this should allow you to run `ssh -Y pi@192.168.42.1`. Otherwise, you can run this script on a computer that has Python, `matplotlib`, and `numpy` installed.

Your plot should look somewhat Gaussian, as in Fig. 16.

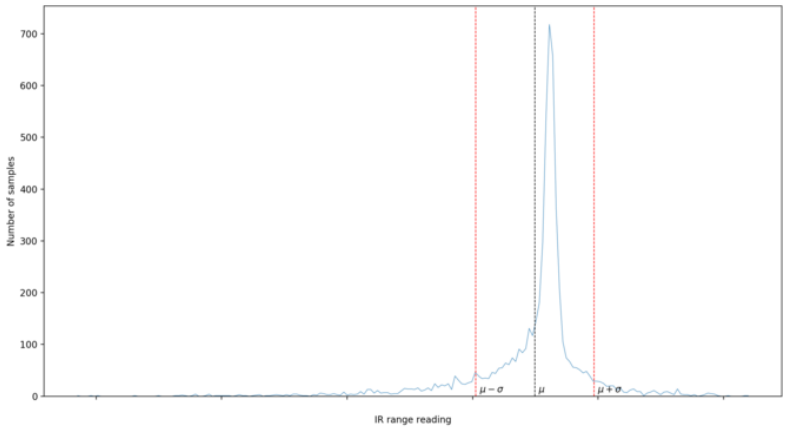


Fig. 16 Sample Distribution of Infrared Range Data

When running `ir_sample_variance_calculation.py`, you can pass in command-line arguments of `-l` to plot a line chart instead of a bar chart and `-n` followed by a positive integer to indicate the number of intervals to use for the histogram (defaults to 100 intervals).

Task (Written Section 1.3.3): Record the resulting sample variance value in `ukf2d_written_solutions.tex`. Also include an image of your histogram in `ukf2d_written_solutions.tex`.

Task: Enter this sample variance value into the code for `self.ukf.R` in the `initialize_ukf_matrices()` method.

Initialize the Filter

Before the UKF can begin its routine of predicting and updating state estimates, it must be initialized with values for the state estimate $\mathbf{x}_t$ and state covariance matrix $\mathbf{P}_t$, as the first prediction call will rely on propagating these estimates forward in time. There is no set way to initialize the filter, but one common approach is to simply take the first measurements that the system receives and treat them as the best estimate of the state until we have estimates for each variable.

Task: For your drone, you want to wait until the first IR reading comes in and then set the corresponding z position value equal to this measurement. This only accounts for one of the two state variables. For now, initialize $\dot{z} = 0 \text{ m}/\text{s}$. Go ahead and implement this state estimate initialization in code in the `ir_data_callback()` method, which gets called each time this ROS node receives a message published by the IR sensor.

Task: In addition to initializing the state estimate, you must initialize the time value corresponding to the state estimate. We provide a method `initialize_input_time()` that accomplishes this, but you must call it in the appropriate location.

Another aspect of the filter that can be initialized upon the first receipt of a measurement is the state covariance matrix $\mathbf{P}_t$. How do we know what values to use for this initialization? Again, this is a design decision that can vary by application. We can directly use the variance of the IR sensor to estimate an initial variance for the height estimate. We won't worry about initializing the velocity variance or the covariances. If we always knew that we were going to start the filter while the drone is at rest, then we could confidently initialize velocity to 0 and assign a low variance to this estimate.

Task: Initialize the $\mathbf{P}_t$ matrix in the `ir_data_callback()` method with the variance of the IR sensor for the variance of the z position estimate. FilterPy initializes instance variables for you, but you should assign these variables initial values. You can refer to the FilterPy documentation to figure out what variable names to use.

Task (Written Section 2.1): How else could you initialize the estimate for $\dot{z}$ given the raw range readings from the IR sensor? Recall that the range readings are an estimate for z, and you can differentiate z to get $\dot{z}$. Describe in `ukf2d_written_solutions.tex` what you would do and the potential pros and cons of your approach. Do not implement this in code.

It is unlikely that the filter initialization will be perfect. Fret not—the Kalman Filter can handle poor initial conditions and eventually still converge to an accurate state estimate. Once your predict-update loop is written, we will be testing out the impact of filter initialization.

Asynchronous Inputs

The traditional Kalman Filter is described as a loop alternating between predictions and measurement updates. In the real world, however, we might receive control inputs more frequently than we receive measurement updates; as such, instead of throwing away information, we would prefer to perform multiple consecutive predictions. Additionally, our inputs (i.e., control inputs and sensor data) generally arrive asynchronously, yet the traditional Kalman Filter algorithm has the prediction and update steps happen at the same point in time. Furthermore, the sample rates of our inputs are typically not constant, and so we cannot design our filter to be time invariant. These are all problems that should be considered when transitioning from the theoretical algorithm to the practical application.

Task (Written Section 2.2): Describe why, in a real-world Kalman Filter implementation, it generally makes sense to be able to perform multiple consecutive predictions before performing a new measurement update, whereas it does not make sense algorithmically to perform multiple consecutive measurement updates before forming a new prediction. It might be helpful to think about the differences between what happens to the state estimate in the prediction versus the update step. Write your answer in `ukf2d_written_solutions.tex`.

Task: Implement the predicting and updating of your UKF, keeping in mind the issue of asynchronous inputs. These steps will occur in two ROS subscriber callbacks: 1) `imu_data_callback` when an IMU control input is received and 2) `ir_data_callback` when an IR measurement is received. Remember that we want to perform a prediction not only when we receive a new control input but also when we receive a new measurement in order to propagate the state estimate forward to the time of the measurement. One way to do this prediction without a new control input is to assume that the control input remains the same as last time (which is what we suggest); another potential approach might be to not include a control input in those instances (i.e., set it to zeros). The method for our FilterPy UKF object that you want to use to perform the prediction is `self.ukf.predict()`, which takes in a keyword argument `dt` that is the time step since the last state estimate and a keyword argument `u`, corresponding to the argument `u` of `state_transition_function()`, that is a NumPy array with the control input(s). The method to do a measurement update is `self.ukf.update()`, which requires a positional argument consisting of a measurement vector as a NumPy array. Call `self.publish_current_state()` at the end of each callback to publish the new state estimate to a ROS topic.

Note that these callbacks get called in new threads; therefore, there is the potential for collisions when, say, both IMU and IR data come in almost at the same time and one thread has not had the opportunity to finish its UKF computations. We don't want both threads trying to simultaneously alter the values of certain variables, such as the $\mathbf{P}_t$ matrix when doing a prediction, as this can cause the filter to output nonsensical results and break. Therefore, we have implemented a simple callback blocking scheme—using the `self.in_callback` variable—that ignores a new callback if another callback is going on, essentially dropping packets if there are collisions. While this may not be the optimal or most stable way to handle the issue (one could imagine the IMU callback, for example, always blocking the IR callback and hence preventing measurement updates), it at least gets rid of the errors that would occur with collisions. If you so desire, feel free to implement your own callback management system that perhaps balances the time allocated to different callbacks.

Tune and Test the Filter

In this problem, you will be testing your UKF that you have implemented thus far. You will start by testing on simulated drone data. We have set up the simulation to publish its data on ROS topics so that your UKF program interfaces with the drone's ROS environment and will be able to be applied directly to real, live data coming from the drone during flight. The output from the UKF can be evaluated in the JavaScript web interface (see `pidrone_pkg/web/index.html`).

In Simulation

To run your UKF with simulated drone data, you first have to make sure that your package is in the `~/ws/src` directory on your drone. Your package has a unique name, so you will need to modify some files. There are two places near the top of `package.xml` where you should replace `pidrone_project2_ukf` with your repo name, and similarly there is one place near the top of the `CMakeLists.txt` file where you should do the same. Then, in `~/ws`, run `catkin_make` to build your package. By running this command, you will be able to run ROS and access nodes from your package. If you experience issues with `catkin`, please do not hesitate to reach out to the TAs.

In order to test your UKF within our software stack, navigate to the file in `pidrone_pkg/scripts` called `state_estimator.py` and edit the line that assigns a value to `student_project_pkg_dir`, instead inserting your project repo name `project-ukf-2020-yourGithubName`.

Next, run ROS as usual with the `./staart_pidrone_code.sh` file in the `pidrone_pkg`. Upon start-up, go ahead and terminate the IR and flight controller nodes, as these would conflict with the drone simulator's simulated sensors. In the state estimator screen, terminate the current process and then run the following command:

```
python state_estimator.py --student --primary ukf2d --others simulator --ir_var
IR_VARIANCE_ESTIMATE
```

with your computed estimate of your IR sensor's variance that you used to determine the $\mathbf{R}_t$ matrix in your UKF in place of `IR_VARIANCE_ESTIMATE`. This command will automatically run your 2D UKF as the primary state estimator, along with the drone simulator. The EMA filter will also be run automatically with the 2D UKF, since the 2D UKF does not provide a very complete state vector in three-dimensional flight scenarios. This will also by default allow you to compare the output of your UKF to the EMA filter on altitude. Note the `--student` flag, which ensures that your UKF script is run.

Now in the web interface, once you connect to your drone, you should see four curves in the **Standard View** of the Height Readings chart as in Fig. 17.

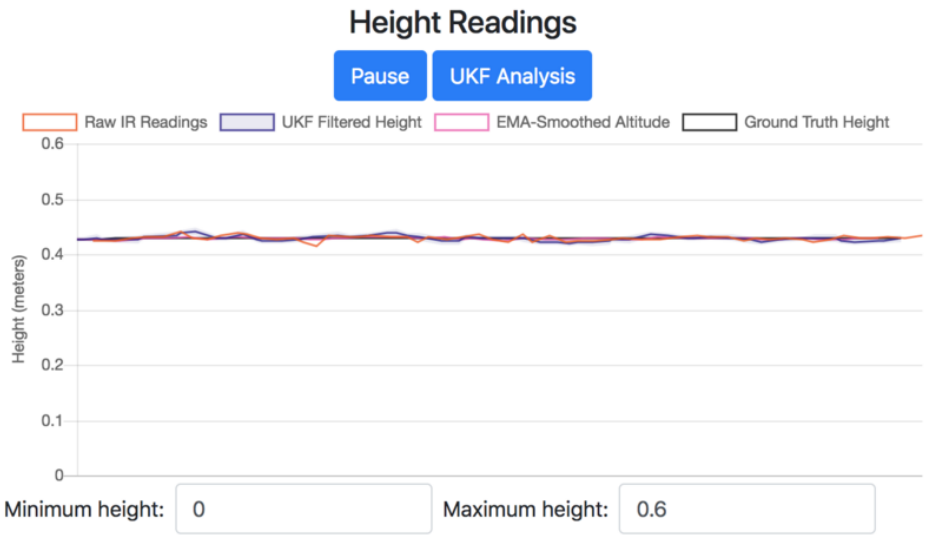


Fig. 17 Standard View of the Height Readings Chart with Drone Simulated Data

1. Raw IR Readings: the orange curve that shows the drone simulator's simulated noisy IR readings
2. UKF Filtered Height: the blue curve that shows your UKF's height estimates, along with a shaded region indicating plus and minus one standard deviation, which is derived from the z position variance in the covariance matrix
3. EMA-Smoothed Altitude: the pink curve that shows the EMA filter's estimates
4. Ground Truth Height: the black curve that is the simulated drone's actual height that we are trying to track with the UKF

If you click on the **UKF Analysis** button, the chart will change over to reveal different datasets, shown in Fig. 18.

Fig. 18 UKF Analysis View of the Height Readings Chart with Drone Simulated Data

With this chart, we can analyze the performance of the UKF. The orange curve represents the error between the UKF and ground truth from the simulation; the closer to zero this value, the better the UKF estimates are tracking the actual altitude of the simulated drone. The blue shaded region indicates plus and minus one standard deviation of the UKF's z position estimates. If the system is indeed behaving in a nice Gaussian manner and the UKF is well tuned, then we expect to see about 68% of the points in the orange dataset lying in the blue shaded region. Also note that on the left side of Fig. 18, the standard deviation and error start off relatively high; this is because the filter is starting out, improving its estimates from initial values.

If you are seeing that your UKF altitude estimates are lagging significantly behind the simulated data in the Height Readings chart, then this is likely due to computation overhead. The UKF takes time to compute, and if it tries to compute a prediction and/or update for each sensor value that it receives, it can sometimes fall behind real time. In this case, you should run the state estimator with the IR and IMU data streams throttled:

```
python state_estimator.py --student --primary ukf2d --others simulator --ir_var
IR_VARIANCE_ESTIMATE --ir_throttled --imu_throttled
```

Make sure your UKF is producing reasonable outputs that seem to track ground truth pretty well. In the UKF Analytics view of the chart, you should see about two-thirds of the points in the error dataset lying within one standard deviation, based on your UKF's state covariance, relative to ground truth.

To test out your UKF's robustness in the face of poor initialization, you can compare how long it takes the state estimates to converge to accurate values with good initial conditions and with poor initial conditions. You do not have to report or hand in anything for this task; it is just for your understanding of the capabilities of the UKF.

Manually Moving the Drone Up and Down

Next, you will step out of the realm of simulation and test your UKF on your drone, manually moving it along the vertical axis to test out the response you get with your IR sensor. For this step, the command you want to use is:

```
python state_estimator.py --student --primary ukf2d
```

with the `--ir_throttled` and `--imu_throttled` flags as needed. You want to make sure your IR sensor and flight controller nodes are actually running. First, quit any existing screens, then calibrate your accelerometer with:

```
roscd pidrone_pkg
python scripts/calibrateAcc.py
```

Debugging Task: Test out your UKF by moving your drone up and down and examining the Height Readings chart. Does it behave as you expect? Does the estimated height seem to have more or less noise than the raw IR sensor reading? If there are unexpected deviations or spikes from the measurements, consider why this might be, especially in comparison to the results you saw when running the UKF in simulation. A possible cause is that the prediction step without a measurement update is not being modeled well or is getting poor/noisy control inputs to the point where the process noise that we assigned was too low. Try tuning the scalar that multiplies the values of the $\mathbf{Q}_t$ matrix `self.ukf.Q` in the `initialize_ukf_matrices()` method to better reflect the variance of the process. You should see a greater standard deviation as well as smaller spikes in the estimates.

Another aspect that you should consider is the prediction that occurs in your IMU callback. Note that the unthrottled sample rate of the IR sensor is around 80 Hz, while the IMU only comes in at about 30 Hz. Therefore, the control input is being changed less frequently than the predictions and measurement updates occur in the IR callback. While in the Asynchronous Inputs section we indicated that you should do a prediction whenever you get a new control input, in this application, it might make

sense to save computation and only do predictions right before measurement updates. Plus, the accelerometers are noisy, and it can be difficult in a discrete domain to integrate these accelerations and expect accurate position estimates reported before including the measurement update. To keep our estimates reasonable, we can wait for the measurement update step to fuse the noisy prior prediction with the new measurement—and since this step can actually occur more frequently than the control input, we can maintain good measurement-informed estimates while saving on prediction computation. To simplify the problem, we can move the prediction and update out of a sensor callback and in its own loop in the main thread of the program.

Task: Modify your UKF to only do predictions and updates in a loop in the main thread of your program, using `rospy.Rate(self.loop_hz)` to regulate the rate at which the UKF tries running (feel free to look up documentation on how to use `rospy.Rate()`). You will want to store the data that come in from the IMU and IR sensor in instance variables that you can use in your main loop. Note that you should now use the `-hz` flag followed by a number (defaults to 30), rather than the various sensor throttle flags, to alter the rate of your UKF. Visually compare your UKF output to the EMA output.

Task: Visually compare the UKF output with and without the IMU running. You should notice a difference in how well/quickly the UKF tracks the measurements when there is no control input to better inform the prediction step.

Task (Written Section 2.3): In `ukf2d_written_solutions.tex`, describe how a (well-tuned) Kalman Filter outperforms an exponential moving average (EMA) filter applied to raw sensor data. Test this out by moving your drone up and down and comparing the UKF and EMA estimates. Once your UKF seems to outperform the EMA, attach an image of the Height Readings graph to your `ukf2d_written_solutions.tex` document showing this difference between your UKF and the EMA, and briefly describe the different features.

In Flight

It's time to fly your drone with the UKF providing it with real-time filtered estimates of its position and velocity along the z-axis.

Task: Fly your drone while running:

```
python state_estimator.py --student --primary ukf2d
```

with the `-hz` flag as needed. Evaluate its performance using the web interface as you did for the manual motion testing.

7D UKF Design and Implementation

While tracking the drone's z position and velocity is helpful, it is a simplified model of the drone and does not encapsulate as many of the degrees of freedom of the drone as we might like. For this reason, you are now going to develop a UKF that tracks the drone in three spatial dimensions with a 7D state vector. Your code from the 2D UKF will be a useful reference, and many parts will be reusable for the 7D UKF.

This part of the project has **two deliverables** in your `project-ukf-2020-yourGithubName` repository, which are to be accessed and submitted via GitHub Classroom:

1. A $\LaTeX$ PDF document `ukf7d_written_solutions.pdf`, generated from `ukf7d_written_solutions.tex`, with the answers to the UKF design and implementation questions.
2. Your implementation of the UKF written in the `state_estimators/student_state_estimator_ukf_7d.py` stencil code. In this stencil code file, we have placed "TODO" tags describing where you should write your solution code to the relevant problems.

State Vector

Just as you tracked position and velocity along one axis in the 2D UKF, now you will track position and velocity along three global-frame axes. You will also track the drone's yaw value ψ. Changes to the drone's orientation will cause nonlinearities that the UKF was designed to address.

$$\mathbf{x}_t = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \psi \end{bmatrix}$$

We don't ask you to track the drone's attitude (roll ϕ and pitch θ), as that makes for an even larger state vector and adds complexity. Also, the IMU incorporates its own filter to produce its estimates of roll and pitch, so there may not be much benefit to adding these variables to our UKF. As such, you will use these roll and pitch values as strong estimates to inform the state transition and measurement functions.

State Transition Function

We define a control input $\mathbf{u}_t$ populated by linear accelerations from the IMU:

$$\mathbf{u}_t = \begin{bmatrix} \ddot{x}^b \\ \ddot{y}^b \\ \ddot{z}^b \end{bmatrix}$$

As noted in the background section, one could treat these acceleration values as measurements instead of control inputs; for relative ease of implementation, we have chosen to use accelerations as control inputs. The linear accelerations are in the drone's body frame, denoted by the superscript b, so we need to rotate these vectors into the global frame based on the yaw variable that we are tracking and the IMU's roll and pitch values. This transformation will occur in the state transition function.

To accomplish this rotation, you will use quaternion-vector multiplication (to be implemented in the stencil code in the `apply_quaternion_vector_rotation()` method). What does this operation look like, and why use this instead of Euler angles? For one, Euler angles are prone to gimbal lock, which is an issue we want to avoid in robotics. Therefore, many people in robotics and other fields such as computer graphics make use of the quaternion to avoid gimbal lock and (arguably) more elegantly encode an object's orientation or rotation. Even though your drone probably will not encounter gimbal lock in its relatively constrained envelope of operation (i.e., we are not doing flips—yet!), we want to introduce you to using quaternions in a practical calculation. Here is a visualization that might help you better grasp the admittedly unintuitive idea of the quaternion.

In particular, we are interested in rotating a vector described relative to the drone's body frame into the global frame. For example, as the drone yaws, its body-frame x-axis will rotate relative to the global frame, so a linear acceleration value sensed by the IMU along the drone's body-frame x-axis will not always correspond to the same direction in the global frame. You can imagine that roll and pitch only complicate this mapping between body and global frame.

In the state transition function, you will be rotating the body-frame linear acceleration vector from the IMU into the global frame. The computation to do this with a quaternion is as follows:

$$\mathbf{u}_t^g = \mathbf{q} \cdot \mathbf{u}_t \cdot \mathbf{q}^*$$

where $\mathbf{u}_t^g$ is the linear acceleration control input in the global frame, $\mathbf{q}$ is the quaternion that rotates a vector from body to global frame, $\mathbf{u}_t$ is your body-frame control input that you get from the IMU, and $\mathbf{q}^*$ is the conjugate of $\mathbf{q}$. Note that, for correct dimensionality, $\mathbf{u}_t^g$ and $\mathbf{u}_t$ should be 4-element vectors to match the quaternion's $[x, y, z, w]$ components and should have the real component w equal to 0, making these vectors "pure" quaternions.

The steps to implement this rotation in the `apply_quaternion_vector_rotation()` method, then, looks something like this:

1. Create a quaternion from Euler angles using `tf.transformations.quaternion_from_euler(roll, pitch, yaw)`, with the appropriate values for roll, pitch, and yaw (radians). The output of this function is a quaternion expressed as an array of $[x, y, z, w]$. It represents the drone's orientation and can be thought of as the quaternion to rotate a vector or frame from the global frame to the body frame. We want a quaternion that does the opposite rotation.
2. Invert the quaternion to get a quaternion that rotates a vector from the body frame to the global frame. To do this, simply negate the w component (i.e., the fourth element of the first quaternion).
3. Express the vector to be rotated as a "pure" quaternion, which means appending a zero to the vector.
4. Carry out $\mathbf{q} \cdot \mathbf{u}_t \cdot \mathbf{q}^*$ by applying the following functions appropriately: `tf.transformations.quaternion_multiply` and `tf.transformations.quaternion_conjugate`.
5. Drop the fourth element of the result of this computation, and return this 3-element array.

Task (Written Section 1.2.2): Implement the state transition function $g(\mathbf{x}, \mathbf{u}, \Delta t)$ in `ukf7d_written_solutions.tex`. Remember that for the drone, this involves kinematics, and since we are now tracking yaw and additionally considering the roll and pitch from the IMU, a rotation will be necessary so that we track state variables in the global frame. Your implementation will use quaternion-vector multiplication as described above to accomplish this rotation. We do not expect you to write out the details of the transformation, but in your notation, you should be clear about the frame in which the control input is described (e.g., you could indicate global frame by notating the control input as $\mathbf{u}_t^g$).

Task: Translate the state transition function into Python by filling in the `state_transition_function()` method in `state_estimators/student_state_estimator_ukf_7d.py`. Follow the "TODO"s there. Be sure to implement `apply_quaternion_vector_rotation()` as well. As usual, note the functions' type signatures for the inputs and outputs.

Measurement Function

The measurements $\mathbf{z}_t$ that are considered are the IR slant range reading r, x and y planar position estimates and yaw estimates ψ_{camera} from the camera, and the velocities along the x- and y-axes provided by optical flow, which you learned about and implemented in the sensors project. Note that in the 2D UKF, we took the IR reading to be a direct measure of altitude; here, in three spatial dimensions, you will use the roll and pitch values directly from the IMU (i.e., not estimated in our UKF) to convert between the slant range, which is what the IR sensor actually provides, and altitude in the measurement function.

$$\mathbf{z}_t = \begin{bmatrix} r \\ x \\ y \\ \dot{x} \\ \dot{y} \\ \psi_{\text{camera}} \end{bmatrix}$$

At the start of your 2D UKF implementation, we asked you to take into account the notion of asynchronous inputs and to do predictions and updates when these values came in. As you later found out, this approach might not yield the best results in our particular application, due to computation limitations and also poor estimates when doing dead reckoning (i.e., predicting based on the current state estimate and motion of the drone) alone in a time step. In this 7D UKF, a similar issue can arise if trying to do a prediction and update cycle in each callback. The sporadic updates, although theoretically feasible, impose the added burden of CPU load inherent in the UKF predict and update steps. A possible solution to this issue is to drop packets of data by throttling down the sensor inputs to the UKF, which will degrade our estimates across the board. Also, by implementing callbacks that block one another, there is the potential that important updates are not being executed as often as they should be, and the system can become unreliable.

The alternative solution to this issue that we have found works better and that you will implement is to reduce the amount of computation done with each sensor input. Instead of throttling the data as it comes in, you will essentially be throttling the predict-update loop—as you ended up doing in the 2D

UKF—using the `-hz` flag. When new data come in, you should store these values as the most recent values for the relevant measurement variables. Then, in a single thread, a predict-update loop will be running and using these measurements. This approach suffers from the fact that the measurements will not be incorporated into the state estimate at the *exact* time at which the inputs were received, but the predict-update loop will be running at a fast rate anyway as it will only run in one thread, so the latency should be negligible. In addition, this approach should make the algorithm simpler to implement, as you will be following the standard predict-update loop model using a single measurement function and measurement noise covariance matrix. An asynchronous approach requires that specific versions of the measurement function and covariance matrix be used for each specific sensor update, as stated by Labbe in chapter 8.10.1 of [Jr18].

Task (Written Section 1.3.2): In `ukf7d_written_solutions.tex`, implement the measurement function $h(\bar{\mathbf{x}}_t)$ to transform the prior state estimate into measurement space for the given measurement vector. Be sure to convert altitude to IR slant range based on the drone's orientation in space. This requires some trigonometry with the roll and pitch angles.

Task: Translate the measurement function into code by filling in the `measurement_function()` method. Follow the "TODO"s there. Note the function's type signature for the inputs and outputs.

Process Noise and Measurement Covariance Matrices

As in the 2D UKF, we do not expect you to derive reasonable values for the process noise.

Task (Written Section 1.3.3): In `ukf7d_written_solutions.tex`, define the measurement noise covariance matrix with reasonable estimates for the variances of each sensor input. You already have an estimate for the IR sensor variance that you experimentally determined in the previous part of the project; for the other sensor readings, you can provide intuitive estimates and potentially attempt to later derive experimental values for these variances if your filter is not performing well.

Task: Enter these sample variance values into the code for `self.ukf.R` in the `initialize_ukf_matrices()` method.

Initialize the Filter

As with the 2D UKF, we must initialize our filter before it can engage in its predicting and updating.

Task: Initialize the state estimate $\mathbf{x}_t$ and the state covariance matrix $\mathbf{P}_t$ with values as sensor data come in.

Asynchronous Inputs

We touched upon this in the **Measurement Function** section. To handle asynchronous inputs, you should update instance variables with the most recent data collected and run a loop to form predictions and updates with these data.

Task: Implement the predict-update loop. It might be useful to refer to the `rospy` documentation on setting loop rates and sleeping.

Task: Complete any remaining "TODO"s in the 7D UKF source code.

Tune and Test the Filter

It is now time to put your 7D UKF to the test.

In Simulation

To run your 7D UKF with simulated data, you need to run ROS on your Raspberry Pi and terminate certain nodes upon running the screen:

- `flight_controller_node.py`
- `vision_flow_and_phase.py`

The simulation is only in two dimensions in the xy-plane, so to also test z position estimates, you should keep the `infrared_pub.py` node running to see your filter work on real IR data.

Next, in the state estimator screen, terminate the current process and then run the following command:

```
python state_estimator.py --student -p ukf7d -o simulator ema --sdim 2
```

If performance is clearly sub-optimal, consider using the `-hz` flag.

This command will run your 7D UKF as the primary state estimator, along with the 2D drone simulator and the EMA filter for comparison. If you do not want to run the EMA filter, simply omit the `ema` argument when running the `state_estimator.py` script.

Task: Make sure your UKF is producing reasonable outputs, especially in the **Top View** chart in which the simulation and its nonlinear behavior are occurring. You should qualitatively feel confident that your UKF marker (the blue marker) is more closely tracking the Ground Truth marker (black) with less noise than the Raw Pose Measurement marker (orange).

Manually Moving the Drone

In this part of the project, you will move your drone around with your hand, holding it above a highly-textured planar surface so that the downward-facing camera can use its optical flow and position estimation to provide information about the drone's pose and twist in space. You should ensure that the following nodes are running:

- `flight_controller_node.py`
- `infrared_pub.py`
- `vision_flow_and_phase.py`

Then, you should run your UKF with this command:

```
python state_estimator.py --student -p ukf7d -o ema
```

using the `-hz` flag as needed.

Task: Use the web interface to verify visually that the height estimates and the x, y, and yaw estimates appear to have less noise than the sensor readings, and that these estimates appear to track your drone's actual pose in space. Compare your UKF to the EMA estimates for altitude and the raw camera pose data in the **Top View** chart.

In Flight

Now you are going to fly with your 7D UKF, using both velocity control and position hold.

Task: Test your drone's stability in position hold and velocity control 1) while running just the EMA filter for state estimation and 2) while running your 7D UKF. You can use the web interface to move your drone around and send it other commands.

Final Hand-In

Before the project deadline, you must ensure that final versions of your solution files and code are handed in via GitHub Classroom. These files are:

From the 2D UKF section:

- `ukf2d_written_solutions.pdf` (compiled from `ukf2d_written_solutions.tex`)
- `student_state_estimator_ukf_2d.py` in the `state_estimators` directory

From the 7D UKF section:

- `ukf7d_written_solutions.pdf` (compiled from `ukf7d_written_solutions.tex`)
- `student_state_estimator_ukf_7d.py` in the `state_estimators` directory

Then come to TA hours to show us your working UKF code. Note that we will ask you to explain one random TODO section that you filled out.

Project 5: Localization and SLAM

Having a good estimate of position is necessary for most tasks in autonomous mobile robotics. A self driving car, a delivery drone, or even a Roomba is not very useful without knowledge of its own location. The task of determining the location of a robot is known as *localization.* In this project, we will implement two algorithms for localization on the Duckiedrone: **Monte Carlo localization** and **FastSLAM.**

Note that the Bayes filter setup, with observation model and measurement model is exactly the same for the Kalman filter and the localization/SLAM particle filter. The particle filter is another way to handle non-Gaussian density functions. In many ways it is simpler to implement, because each particle is transitioned/updated assuming its state values are ground truth, and covariance is maintained over the entire set of particles rather than updating a covariance matrix. However it requires more computation as performance/accuracy scales with the number of particles. Notably particle filters are capable of representing multimodal distributions, making them a good fit for localization, where you may have multiple peaks (for example as the robot goes down a corridor and sees identical observations), that then resolve as it reaches a disambiguation location.

These algorithms cover two important cases: one in which the robot has a map of its environment available beforehand, and a second in which it does not. In this second case, the robot must use its sensors to simultaneously develop a map of its surroundings and localize itself relative to that map. Not surprisingly, this is referred to as the *simultaneous localization and mapping problem*, hereafter referred to as SLAM.

Please use this link to generate your Github classroom repository and pull the stencil code. Use the Github repo created to handin your assignment and backup any changes you make.

Localization Background

Bayes Filter

Monte Carlo Localization is a type of Bayes filter. You'll remember the general Bayes Filter algorithm from the UKF project earlier in the course. This material is also covered in the EdX lectures for the class, which also contains mathematical derivations for the filter.

The Bayes Filter incorporates information available to the robot at each point in time to produce an accurate estimate of the robot's position. Its core idea is to take advantage of data from two sources: the controls given to the robot and the measurements detected by the robot's sensors.

At each new time step, the Bayes filter recursively produces a state estimate, represented as a probability density function called the *belief.* The belief assigns to every possible pose in the state space of the robot the probability that it is the robot's true location. This probability is found in two steps called **prediction** and **update**.

The prediction step incorporates controls given to the robot between the previous state and the current one. It finds the probability of reaching a new state given the previous state and the control (hence recursion). The model used to find this probability is known as a *state transition model* and is specific to the robot in question.

The state transition model:
$p(x_t|u_t, x_{t-1})$
ie. the probability that the most recent control u_t will transition the previous state x_{t-1} to the current state x_t

It is possible to estimate the state of the robot using only the prediction step and not incorporating the measurements taken by the robot's sensors. This is known as *dead reckoning*. The dead reckoning estimate may be made more accurate by incorporating measurements from the robot's sensors.

The Bayes filter does this in the update step by finding the probability that the current measurements are observed in the current state. The model used for this is known as a *measurement model* and is specific to the robot in question.

The measurement model:
$p(z_t|x_t)$
ie. the probability that the current measurement z_t is observed given the state x_t

You may have noticed that each of the above steps required computing a probability stated like "the probability of x given y." Such a probability is denoted $p(X|Y)$ and may be calculated by the famous Bayes Theorem for conditional probabilities, hence the name of the algorithm.

Now, let's take a look at the Bayes Filter:

$$\begin{aligned}&\text{Bayes\_Filter}(bel(x_{t-1}), u_t, z_t):\\&\quad\text{for all } x_t \text{ do}:\\&\quad\quad \bar{bel}(x_t) = \int p(x_t|u_t, x_{t-1}) bel(x_{t-1}) \mathrm{d}x\\&\quad\quad bel(x_t) = \eta p(z_t|x_t) \bar{bel}(x_t)\\&\quad\text{endfor}\\&\quad\text{return } bel(x_t)\end{aligned}$$

ie compute a belief by finding the probability of each possible new state. For each state, incorporate both the probability that the control transitions the previous state to this one and that the current measurements are observed in this state.

The first step $\bar{bel}(x_t) = \int p(x_t|u_t, x_{t-1}) bel(x_{t-1}) \mathrm{d}x$ is the *motion prediction*. $\bar{bel}(x_t)$ represents the belief *before* the measurement is incorporated. The integral is computed discretely and becomes: $\sum_x p(x_t|u_t, x_{t-1}) bel(x_{t-1})$

The second step $bel(x_t) = \eta p(z_t|x_t) \bar{bel}(x_t)$ is the *measurement update*. This computation is straightforward, the normalizer η is the reciprocal of the sum of $p(z_t|x_t) \bar{bel}(x_t)$ over all x_t. This factor will normalize the sum.

Monte-Carlo Localization

The phrase "Monte Carlo" refers to the principle of using random sampling to model a complicated deterministic process. Rather than represent the belief as a probability distribution over the entire state space, MC localization randomly samples from the belief to save computational time. Because it represents the belief as samples, it is capable of representing multimodal distributions. For example when localizing, if the robot is teleported (i.e., turned off, and moved, and then turned on), its initial believe is uniform over the entire map. The Gaussian distribution has a hard time representing this, because its weight is centered on the mean (although you could approximate it with a very very large covariance). Another example is if the robot is experiencing aliasing. For example, if it is going down a long corridor, its observations at different points down the corridor will be exactly the same, until it reaches a distinguishing point, such as an intersection or the end of the corridor. Particle filters can represent this by having particles all along the corridor; whereas a Gaussian distribution will struggle because of the need to pick one place to center the distribution with the mean.

MC localization is a *particle filter* algorithm. In our implementation, we will use several **particles** which each represent a possible position of the drone. In each time step (for us defined as a new frame captured by the drone's camera) we will apply a *motion prediction* to adjust the poses of the particles, as well as a *measurement update* to assign a probability or *weight* to each particle. This process is analogous to Bayes Filtering.

Finally, at each time step we *resample* the particles. Each particle has a probability of being resampled that is proportional to its weight. Over time, particles with less accurate positions are weeded out, and the particles should converge on the true location of the drone!

To retrieve a position estimate of the drone at any time, we can take a simple idea from probability and compute the *expectation* of the belief distribution: the sum over each particle in the filter of its pose times its weight.

The expectation of a random variable X:
$E[X] = \sum_x xp(X = x)$

$p(X = x)$ is the probability that the true pose of the drone is equal to a particle's estimate of the pose, ie, the weight of the particle. For example, if we wanted to retrieve the pose estimate for the drone along the x axis, we would take the weighted mean of each particle's x value, where the weight is the weight of each particle.

The following diagram shows the operation of MC Localization. In the diagram, our friendly H2R robot is trying to localize himself relative to a long table with some nuts and bolts, which are very useful to a robot!

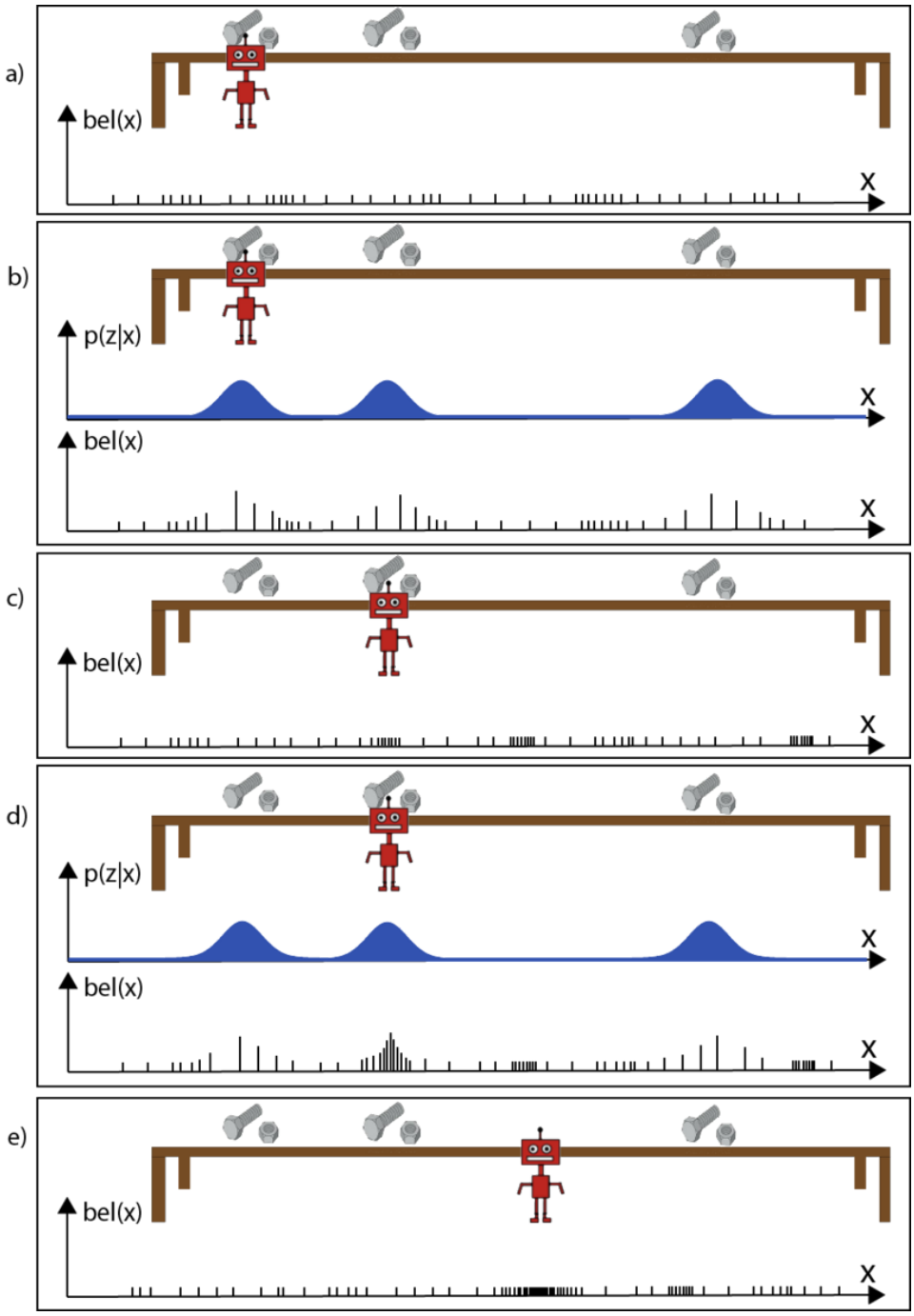


Fig. 19 Monte Carlo Localization. Vertical lines represent particles whose height represents the weight of the particle. p(z|x) is the measurement function. Figure inspired by Probabilistic Robotics.

- a. The robot starts in front of the first bolt. A set of particles are initialized in random positions throughout the state space. Notice that the particles have uniform initial weights.
- b. We weight the set of particles based on their nearness to the bolts using the measurement function.
- c. The robots moves from the first bolt to the second one, the motion model causes all particles to shift to the right. In this step, we also resample a new set of particles around the most likely positions from step b.
- d. Again, we weight the particles based on their nearness to the bolts, we can now see a significant concentration of the probability mass around the second bolt, where the robot actually is.
- e. The robot moves again and we resample particles around those highest weighted from part d. We can now see that the belief distribution is heavily concentrated around the true pose of the robot.

If you are feeling shaky about the MC localization algorithm, we recommend studying the diagram above until things start to make sense!

In `localization_answers.md` provide answers to the following questions:

Problem 1 - Localization Theory Questions

Q1- What is the advantage of particle filters relative to the Gaussian representation used by the Kalman filter?

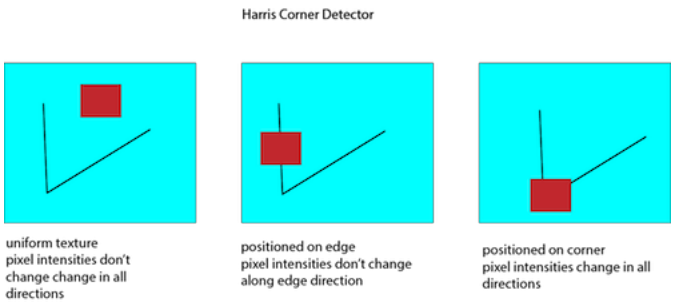
Q2- Can Monte Carlo Localization approximate any distribution? If no, explain why? If yes, describe what controls the nature of approximation?

Our Localization Implementation

To complete our understanding of how we will implement a particle filter on the drone for localization, we need to address specific state transition and measurement models.

OpenCV and Features

The drone's primary sensor is its downward-facing camera. To process information from the camera, we will use a popular open source computer vision library called *OpenCV.* We can use opencv to extract *features* from an image. In computer vision, features are points in an image where we suspect there is something interesting going on. For a human, it is easy to identify corners, dots, textures, or whatever else might be interesting in an image. But a computer requires a precise definition of thing-ness in the image. A large body of literature in computer vision is dedicated to detecting and characterizing features, but in general, we define features as areas in an image where the pixel intensities change rapidly. In the following image, features are most likely to be extracted at the sharp corner in the line. Imagine looking at the scene through the red box as it moved around slightly in several directions starting in each of the three points shown below. Through which box would you see the scene change the most?



When we extract features from the drone's camera feed, OpenCV will give us a **keypoint** and **descriptor** for each feature. The keypoint holds the (x,y) coordinate of the feature in the image frame. The descriptor holds information about the feature which can be used to uniquely identify it, commonly stored as a binary string.

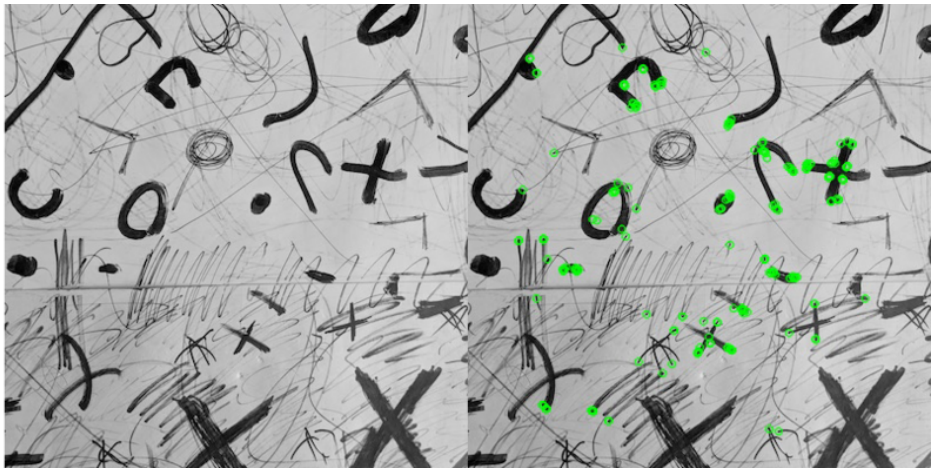


Fig. 20 An image taken on the drone's camera, shown with and without plotting the coordinates of 200 features detected by ORB

The specific feature detector we will be using is called ORB.

> **See also**
>
> If you want to learn more about ORB, you may read more about it here .

Using OpenCV, we are able to perform some powerful manipulations on features. For example, panoramic image stitching can be achieved by matching feature descriptors from many overlapping images, and using their corresponding keypoints to precisely line up the images and produce a single contiguous scene.

We will use OpenCV features to implement both the motion (state transition) and measurement models for localization. We can find the movement of the drone for the motion update by measuring how far it moves between consecutive camera frames. This is done by matching the descriptors between two frames, then using their keypoint positions to compute a transformation between the frames. This transformation will give us an x, y, and yaw displacement between two frames. Note that this requires some overlap between two image frames.

To find the probability for each particle, we would like to measure the accuracy of the particle's pose. We will do this by comparing the camera's current image to the map of the drone's environment. Remember, this is a localization algorithm, meaning that we have a map available beforehand. In our case, the map is an image of the area over which the drone will fly. We can match the descriptors from the current image to the descriptors in the map image, and compute the transformation between the

sets of corresponding keypoints to obtain a global pose estimate. The probability that a given particle is the correct pose of the drone is proportional to the error between the global pose estimate and the particle's pose.

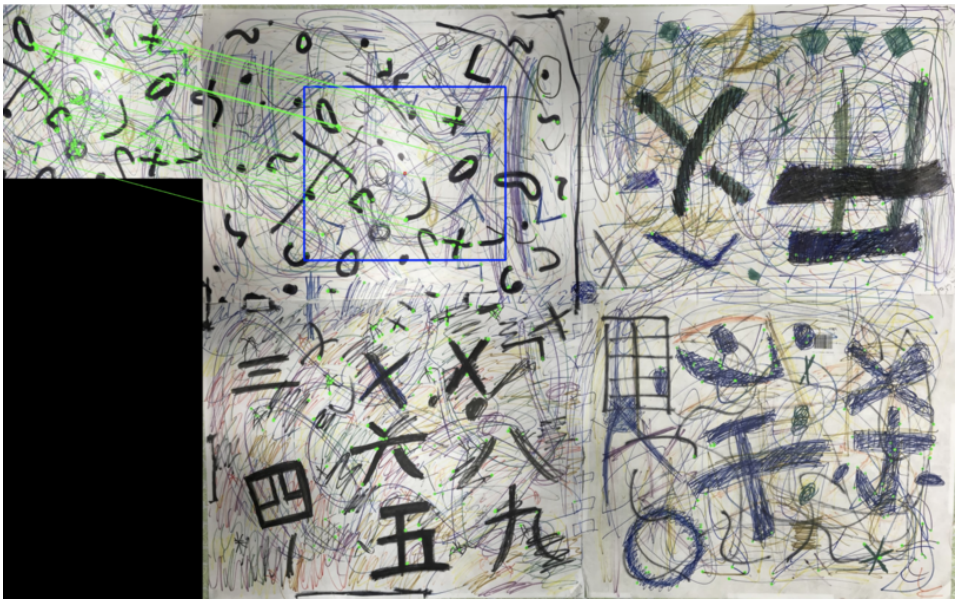


Fig. 21 Computing the transformation from the drone's current view to the map in order to determine global pose

The following algorithm formulates precisely how we will use the ability visualized above to compute the global pose to weight the particles in MC localization. We observe features and attempt to match them to the map. If there enough matches, we compute the global pose of the drone, and compute q, the probability of the particle. q is equal to the product of the probabilities of the error between the particle's pose and the computed global pose in x, y, and yaw.

$$
\begin{array}{ll}
\textbf{Landmark Model Known Correspondence} & \\
\textbf{Measure}(\mathbf{c_t}, \mathbf{x_t}, \mathbf{m}) & \mathbf{c} \leftarrow \textbf{match}(\mathbf{c_t}, \mathbf{m}) \\
\quad \textbf{If} |\mathbf{c}| \geq \mathbf{n} & \mathbf{x}, \mathbf{y}, \mathbf{yaw} = \textbf{compute}(\mathbf{c}) \\
\qquad \mathbf{q} = \textbf{prob}(\mathbf{x} - \mathbf{x_t}[0], \sigma_{\mathbf{x}}^2) \cdot \textbf{prob}(\mathbf{y} - \mathbf{x_t}[1], \sigma_{\mathbf{y}}^2) \cdot \textbf{prob}(\mathbf{yaw} - \mathbf{x_t}[3], \sigma_{\mathbf{yaw}}^2) & \\
\quad \textbf{Else} & \mathbf{q} = \mathbf{0} \\
\quad \textbf{Endif} & \textbf{return } \mathbf{q}
\end{array}
$$

One final consideration for our implementation of MC Localization is how often to perform the motion and measurement updates. We ought to predict motion as often as possible to preserve the tracking of the drone as it flies. But the measurement update is more expensive than motion prediction, and doesn't need to happen quite so often.

A naive solution is to perform updates after every set number of camera frames. But since we are already computing the distance between each frame, it is straightforward to implement a system which waits for the drone to move a certain distance before updating. This idea is known as a **keyframe** scheme and is useful in many scenarios when computations on every camera frame are not feasible. It will be useful later on in SLAM to have the threshold for distance between two keyframes equal to the length of the camera's field of view, so we will implement such a system here.

Localization Assignment

Getting Set Up

You should have cloned the GitHub Classroom link to receive the deliverables for this project.

You should receive a directory named “project-localization-yourGithubName”. The only part of this assignment which you must run on the drone is localization (the last assignment on this page). To do this, place your directory in the `/ws/src` folder on your drone. You should find the `package.xml` and `CMakeLists.txt` files which you will need to modify to build the package. On line 3 of “package.xml” you should replace `yourGithubName` with your GitHub name so it matches the name of your directory. Do the same on line 2 of `CMakeLists.txt` Finally, you should navigate to the `/ws` folder and run

```
catkin_make --pkg project-localization-slam-2019-yourGithubName
```

to build your package so it is ros-runnable from the pidrone_pkg. You should only need to do this step one time.

Dependencies

In order to complete this project, we will make use of the following libraries: Numpy for computations, OpenCV for computer vision, and MatPlotLib for creating plots and animations. You are welcome to run on your drone the parts which do not require visualization, ie the OpenCV assignment. However, the particle filter assignment will require you to view a MatPlotLib animation. To accommodate this, you may either install the required dependencies on your own computer (optional!) or work on a department machine which already has them. If you install OpenCV yourself, make sure the version is 2.4.9. The easiest way to work on this project is to work over ssh on your computer and use XQuartz (what the -Y is for when you type ssh -Y) which will allow you to view animations over ssh. As a reminder, to access your account on the department machines, open a terminal and run `ssh -Y your_login@ssh.cs.brown.edu`. You may use cyberduck or your preferred method to transfer files from your computer to the department machines.

Particle Filter

First, you will complete a series of quick exercises which will guide you through implementing a simplified particle filter. This part of the assignment must be completed on a computer with matplotlib and numpy installed. You will be given two files:

```
student_particle_filter.py
animate_particle_filter.py
```

In student_particle_filter you will implement a particle filter which causes a set of randomly generated points on a 2d plane to converge on a specific point. student_particle_filter will write the particles’ poses to a text file, which animate_particle_filter will read and use to generate an animation.

Note that there are more detailed instructions for each step in the comments of student_particle_filter.

Problem 1: Setup Define a Particle class to represent the particles in the filter. Each particle should store its position (x,y) and its weight.

Define a ParticleFilter class to store the set of particles, the desired pose, and the methods which operate on the particle set. Create an **init** method which takes the number of particles as input and creates a set of particles at random positions.

Problem 2: Motion Implement a method for the ParticleFilter class which adds some random Gaussian noise to the x and y positions of each particle in the filter. Be sure that the noise is different for each particle. *Hint:* try numpy.random.normal.

Problem 3: Measurement Update Implement a method for the ParticleFilter class which sets the weight of each particle inversely proportional to the particle’s distance from the desired pose.

Problem 4: Test Try running your code! If it works properly, the particle poses should be written to a file called “particle_filter_data.txt.” You can then run the file “animate_particle_filter” to view an animation of your particle filter converging on the desired pose which you set.

Problem 5: Optimization OPTIONAL STEP Now that your filter is running, let’s consider how we can optimize this process so that the localization particle filter will run quickly in real time on your drones.

Python data structures and their operations are relatively slow compared to their Numpy counterparts because Numpy is written in C. You will use Numpy arrays to avoid storing the set of particle poses and their weights as lists of Python objects. You may comment out the Particle class entirely and replace the list of particle objects with two Numpy arrays for poses and weights stored in the ParticleSet class. Adjust the rest of the code accordingly. This step is meant to help you understand the optimizations (which are done in the same way) in the localization code.

OpenCV

This part of the assignment may be completed on your drones, or any computer with OpenCv and NumPy.

Now we that know the basics of how a particle filter uses weights and resampling to converge on a target, we need to address how to use OpenCV to estimate the motion and global position of the flying drone. To do this, you will complete a short assignment using OpenCV functions to compute the translation in the plane between two drone poses, represented by two overlapping images taken on a real drone. You will be provided with the following files:

```
image_A.jpg
image_B.jpg
student_compute_displacement.py
```

`student_compute_displacement.py` will indicate the infrared reading taken by the drone at the time images A and B were taken. This is important because the real-world dimensions of a pixel in the image will vary based on the height of the drone. Why is this?

Your job is to write code in student_compute_displacement.py that will extract features from both images and compute a transformation between them. Use this transformation to compute the x,y, and yaw displacement in *meters* between the two images. This is exactly how you will implement the motion model for localization: we consider the meter displacement between two drone images to be the motion of the drone between the poses at which the images were taken.

Appendix: Implement Localization on the Duckiedrone

Feel free to give this a try; we haven't worked through the bugs and this part of the assignment is optional.

We are now ready to implement localization on the drone.

You will be given two files:

```
student_run_localization.py
student_localization_helper.py
```

`student_run_localization` runs localization on the drone and is complete, you will not need to implement any code in that file. However, you may adjust the NUM_PARTICLE and NUM_FEATURE values to experiment with the speed/accuracy tradeoff concerning the number of particles in the filter and the number of features extracted by OpenCV. You may also edit this file if you need to change the map over which you want to localize.

`student_localization_helper` contains the particle filter class and its methods. Many of the methods are not implemented. The docstrings describe the intended functionality of each function, and the TODOs indicate tasks to be completed. Your assignment is to follow the TODOs and implement the missing functionality of the particle filter. Much of the code you just wrote can be used here!

Tip: we recommend that you read through the parts of the code which we are not asking you to implement, as this will help you to understand what is going on with the code and will likely save you debugging time. For example, we are not asking you to implement "resample_particles" or "initialize_particles" for localization, but it might help you to understand how they work! The same goes for the SLAM project.

Note that for both this part of the assignment and for SLAM, there is not any “correct” universal implementation of the code as long as your solutions work.

Testing

To test the functionality of your localization code, you may fly the drone while running

```
rosrun project_localization_slam_2019_yourGithubName student_run_localization.py
```

in the vision window. Follow the Mapping and Localization instructions in the operations manual to see how to change the map. You should see poses printed out which correspond to the drone’s position over the map.

You may also use animate_particle_filter.py to view the animation of your particle filter. Print the (x,y) pose of each particle on separate lines in a text file to be read by animate_particle_filter, put x and y pose coordinates on separate lines. Make sure you adjust animate_particle_filter.py to reflect the number of particles you are using! (using the visualizer here is optional)

Checkoff

We will verify that your code has the following functionality:

1. You can run student_run_localization.py and take off with your drone.
2. While flying, you can hit ‘r’ and the poses will begin printing to the terminal. You can hit ‘r’ again and localization will restart.
3. While flying, you can hit ‘p’ to toggle position hold on and off.
4. Run student_run_localization.py while holding the drone over a mapped area. Do not arm the drone. As you move the drone around, verify that the poses reflect the movement. Verify visually that the poses are close to the actual position of the drone in the map. For example, if you are holding the drone above the bottom left corner of the mapped area, the pose should be close to (0,0).

SLAM Background

Congratulations! You have implemented a real-time localization algorithm for a flying drone.

While this code tends to work pretty well, consider the limitations of a localization algorithm which only works when a map of the environment is available beforehand. In the future, we will see autonomous robots operating in our schools and homes. Such robots will have no access to maps of their environments beforehand; they will need to map their environments in real time!

To provide this functionality to the drone, we will extend the localization particle filter such that each particle will not just estimate the path of the drone, but a map of the drone’s environment.

The algorithm that accomplishes this is called FastSLAM. A map in FastSLAM is represented by a set of **landmarks.** A Gaussian approximates the pose of the landmark. For us, this is a pose (x,y) and a 2x2 covariance matrix. In our implementation, a single landmark in the map corresponds to a single feature extracted by OpenCV.

Most SLAM algorithms seek to approximate the following probability distribution:

$$p(\Theta, x^t | z^t, u^t)$$

where:

- Θ is the map consisting of N landmarks $\Theta = \theta_1, \dots \theta_N$
- x^t is the path of the robot $x^t = x_1, \dots, x_t$
- z^t is the sequence of measurements $z^t = z_1, \dots, z_t$
- u^t is the sequence of controls, $u^t = u_1, \dots, u_t$

ie approximate the path of the drone and the map of its environment given all past measurements and controls.

The main mathematical insight of FastSLAM is the ability to factor the above belief distribution by landmark:

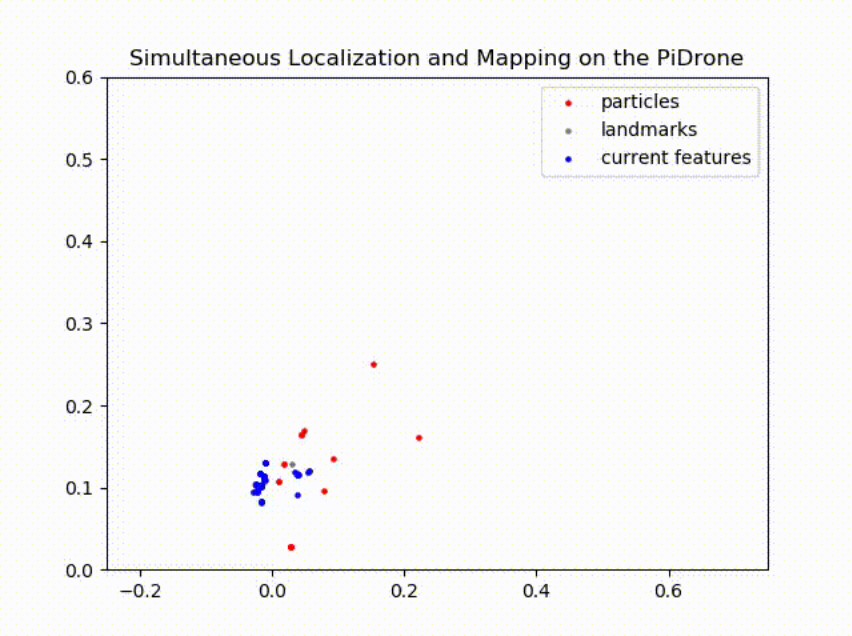
$$p(\Theta, x^t | z^t, u^t) = p(x^t | z^t, u^t) \Pi_n p(\theta_n | x^t, z^t, u^t)$$

This factorization asserts the fact that landmark positions in the map are *conditionally independent* of one another if the path of the robot is known. Hence the product over n for each landmark θ_n.

With this insight, we can represent the map with many 2-dimensional Gaussians, one for the pose of each landmark. Otherwise, as in the popular EKF SLAM algorithm, we would have to store and update a 2N-dimensional Gaussian, with `N` the number of landmarks in the map. As you will see, our maps will grow to include hundreds of landmarks. Updating a covariance matrix with $(2N)^2$ entries for `N=500` landmarks would not be so fun!

The basic steps of FastSLAM will closely resemble those of MC Localization: generate a set of particles and in each time step: update their positions with motion data, weight them based on their accuracy, and resample.

The following animation shows FastSLAM running on the Duckiedrone:



In grey are all of the landmarks in the map, in blue are the features being observed by the drone during each moment in time, and in red are the poses of the FastSLAM particles (our belief about the location of the **drone**).

Notice that as the drone moves throughout the plane, newly observed features, marked in blue, are added to the map as grey particles. As areas of the map are revisited by the drone, the algorithm updates those areas with the new information, and you can see the landmarks shift. Remember that the pose of each landmark is filtered with an EKF, so as we revisit a landmark more times, we incorporate more information about it and our certainty about the map increases.

Please provide answers to the following questions in `answers.md`

Problem 2 - FastSLAM questions

Q1- Why is the property of the landmark positions being conditionally independent important for FastSLAM?

Q2- Does FastSLAM include EKF's? If yes, how are they part of the algorithm?

Our SLAM Implementation

As mentioned before, each particle holds a complete estimate of the map. Altogether, our FastSLAM particles will consist of a pose (x,y), a list of landmark objects, and a weight. The landmark objects will each store a pose (x,y), a 2x2 covariance matrix, a descriptor, and a counter. We will discuss the descriptor and counter shortly.

The motion prediction, resampling, and pose estimation steps in FastSLAM will not be different than in MC Localization. In fact, you can reuse much of your code from MC Localization for these steps!

The measurement update, however, is a little more involved. Since we no longer have a map of the environment available to compare with the current camera frame to weight each particle, we will need some way to judge the confidence of a particle based on the set of landmarks.

To accomplish this, we will attempt to match the current features extracted from the camera feed with the landmarks stored in each particle. This is why we store a feature descriptor with each landmark object.

When updating each particle, we will attempt to match each newly observed feature *with only those landmarks which lie in a close range around the particle's pose.* This step is very important as it will ensure that the map stored in each particle is conditioned on the robot pose represented by that particle. Otherwise, all particles will have the same weights and having a filter with many particles would be pointless!

Each observed feature will either match with a landmark in the particle or it will not. If the feature matches a landmark that was already observed, you should "reward" that particle by increasing its weight. This is because *we are more confident that a particle's pose is correct if the landmarks in the map around that particle are matching with the features we currently observe.* At this time, you must also update the pose and covariance of the landmark using data from the newly observed feature.

If the observed feature does not match an existing landmark, you should add it to this landmark's map and "punish" the particle's weight because extending the map with new landmarks decreases our confidence in its correctness.

We would also like to have some scheme to ensure that dubious landmarks in the map get discarded. Otherwise, the map will grow too large to store. To implement this, each landmark will have a counter. Increment the counter each time the landmark gets revisited, and decrement it if the landmark was in a close range to the pose of the particle, yet not matched to. If the counter goes below a certain threshold, remove that landmark from the map. This ensures that landmarks which *should* have been seen, yet were not, are removed. Removing a landmark from the map is also a good time to punish the weight of that particle.

The exact weighting scheme for the measurement update will be left up to you.

> **Tip**
>
> When you punish a particle, rather than reduce its weight, you should increase it by a positive value that is relatively smaller than the "reward" value. This ensures that the weights of the particles all have the same sign.

The last part of the implementation we have not covered is the process of setting the pose and covariance of new landmarks, and updating them when a landmark is revisited. To do this, FastSLAM uses the EKF or Extended Kalman Filter. The EKF is very similar to the UKF which you implemented in project 3, but with a different scheme for linearizing the motion and measurement functions. Since you have already implemented similar code in that project, you will be provided with two functions:

- `add_landmark`
- `update_landmark`

which will take care of all of the EKF linear algebra for you.

More Formally:

The state is the drone's position and yaw, assuming we are mostly horizontal:

$$x_t = \begin{bmatrix} lx \\ y \\ z \\ \psi \end{bmatrix}$$

We assume velocity control, so we move in the plane, up and down, and yaw. This is set with the `Mode` messages in `pidrone_pkg`.

> **Note**
>
> The z in the `Mode` message is a position and not a velocity. However I want to change this, and propose we ignore z for now anyway - just keep a constant height.

$$u_t = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\psi} \end{bmatrix}$$

Then the transition function is:

$$g(x_t, u_t, \Delta t) = \begin{bmatrix} x_{t,x} + u_{t,\dot{x}\Delta t} \\ x_{t,y} + u_{t,\dot{y}\Delta t} \\ x_{t,z} + u_{t,\dot{z}\Delta t} \\ x_{t,\psi} + u_{t,\dot{\psi}\Delta t} \end{bmatrix}$$

Following Thrun 2005, we assume that we can process a camera image and localize each feature in the image, if it is present. We will then obtain a range, r and bearing, ϕ for the features in the image. We assume access to a map, $m = \{m_1, m_2, \ldots, m_n\}$ the set of all landmarks. Each m_i is the location x, y, z consisting of the location of the ith landmark.

$$f(z_t) = f_t^1, f_t^2, \ldots,$$

We assume each feature is an independent measurement:

$$p(f(z_t)|x_t, m) = \prod_i p(r_t^i, \phi_t^i, s_t^i | x_t, m)$$

Then the measurement model for each feature i is:

$$h(i, x_t, m) = \begin{bmatrix} \sqrt{(m_{i,x} - x)^2 + (m_{i,y} - y)^2} \\ \text{atan2}(m_{i,y} - y, m_{i,x} - x) - \psi) \\ s_i \end{bmatrix}$$

Number of features, the feature detection and computation method (currently `ORB`, can be `sift` or `surf`), number of particles, could be changed for better performance.

Algorithm 1 landmark model known correspondence (x, y, yaw)

procedure MEASURE(c_t, x_t, m) ▷ c_t is all features we currently observed
  $c \leftarrow match(c_t, m)$ ▷ find matched features in global pose
  if $|c| \geq$ n **then** ▷ n is a constant, $n = 10$
    $x, y, yaw = compute(c)$ ▷ compute current observed pose
    $q = prob(x - x_t[0], \sigma_x^2) \cdot prob(y - x_t[1], \sigma_y^2) \cdot prob(yaw - x_t[3], \sigma_{yaw}^2)$
  else
    $q = 0$ ▷ cannot find n matched features
  return q

Fig. 22 Algorithm 1: Landmark model known correspondence (x, y, yaw)

This math follows references:

- Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping:part i. *IEEE robotics and automation magazine*, 13(2):99110, 2006.
- Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics.* MITpress, 2005.

SLAM Assignment

The assignment is slightly different than the localization project due to some computational constraints. Unfortunately, SLAM is not fast enough to run in real time on board the raspberry pi. Instead, you will implement SLAM to run on some pre-recorded flight data (lists of keypoints, descriptors, and IR data). Your SLAM will run offline on the data, and you can judge the correctness of your implementation using animate_slam.py to view the animation of the flight. We will provide you with the animation produced by our solution code for comparison.

If you try to run your SLAM program offline on the drone, you will find that the program takes up to 15 minutes to run! Instead, we recommend that you use the department machines to develop your solution, as they have opencv, numpy, and matplotlib pre-installed and only take a few moments to run SLAM on sample data. Alternatively, you are welcome to install these dependencies on your own computers.

After you have implemented your solutions offline, you may optionally use your solution code to generate maps onboard the drone, then fly the drone localizing over the map. To do this, you may use a modified version of the localization code, called MATL (mapping and then localization) which will be provided for you. See the more detailed instructions for this below.

We will provide you with the following files:

```
slam.py
student_slam_helper.py
utils.py
map_data.txt
animate_slam.py
animation.mp4
```

- `slam.py` and `utils.py` contain fully implemented helper code that you are welcome to read but do not need to edit. You may, however, edit the number of particles in `slam.py` to experiment with the speed/accuracy tradeoff for FastSLAM. You should find that 15 particles are plenty.
- `utils.py` contains add_landmark and update_landmark as promised, as well as some other helper code.
- `map_data.txt` contains data from a previous flight. It stores keypoints, descriptors, and IR data from each camera frame. The helper code included in `slam.py` will read the data from map_data, create a FastSLAM object imported from `student_slam_helper.py` (your code), and run the data through your SLAM implementation.
- `slam_data.txt` will hold data generated by your SLAM implementation including particle poses, landmark poses, and currently observed feature poses. slam_helper.py will write these data as it runs through the saved flight data with your SLAM implementation. You can use animate_slam.py to view the animation from your SLAM.
- `animation.mp4` is the animation generated by our solution code with which you can compare your own animations.

The only thing left for you to do is implement the missing parts of the slam_helper file, which are indicated with TODOs. The intended functionality of each missing method is indicated in the docstrings. You will find that you have already implemented much of the functionality of SLAM in your localization code.

Dependencies

Similar to the particle filter assignment, developing SLAM offboard will require the libraries NumPy, OpenCV, and MatPlotLib. Again, you are welcome to install these dependencies on your own machines or use the department machines to implement this program. The easiest way to work on this project is probably to work over ssh on your laptops and use XQuartz (what the -Y is for when you type ssh -Y) which will allow you to view animations over ssh!

Checkoff

You should develop your SLAM implementation, on the department machines or your own computer, using the provided helper code. When you want to test your implementation, you should first run slam.py to run through your implementation in slam_helper.py with the sample data (takes 1-2 minutes to run) and then run animate_slam.py to read from slam_data.txt and create your animation. If your animation closely follows that from the solution code, you've done a great job!

The checkoff for this project is simple, run your slam_helper.py implementation on the sample data for a TA. Show the TA the corresponding animation. An easy way to do this is to login to your account on the department machines with "ssh -Y" and if you have xquartz installed on your computer, you can run your animation from the terminal over ssh.

On-Board Offline on the Drone

SLAM can run but it is quite slow on the Raspberry Pi. We have gotten it to work on board the Raspberry Pi, but offline. So you fly, record data, land, and then make a map and localize. Details are in the operation manual, and there are a lot of rough edges.

Handin

Please be sure to push your finished project directory to GitHub classroom to handin. For your final handin, you should have edited all of the following files:

```
student_slam_helper.py
student_localization_helper.py
student_compute_displacement.py
student_particle_filter.py
```