Massachusetts Institute of Technology

Robotics: Science and Systems I

Lab 5: Local Navigation and Error Analysis

Distributed: 2/25/2015, 3pm

Wiki Materials and Briefings Due: 3/4/2015, 3pm

Objectives and Lab Overview

In Lab 3/4 you explored using camera data in order to sense your environment and react appropriately to it. In lecture, we have started to learn about navigation by considering localization and navigation-oriented sensing.Your objective in this lab is to preliminarily integrate these different capabilities. This lab will give you the technical skills to incorporate bump and sonar sensors. You will learn to process sensor measurements with simple filters. You will learn how to make your robot follow a wall. Finally, you will learn how to acquire a model of the environment and use it to allow your robot to tour an obstacle.

Your objectives in this lab are to:

- Integrate bump and sonar sensing into your robot.
- Program the robot to detect and react to collisions.
- Program the robot to reliably detect an obstacle wall and drive along it.
- Program the robot to acquire a geometric model for a convex obstacle from the sensor data.

We advise you to read the lab hand-out quickly at the start of the lab and determine what work can be done by splitting up responsibilities among group members.

Time Accounting and Self-Assessment:

Make a dated entry called "Start of Local Navigation Lab" on your Wiki's Self-Assessment page. Before doing any of the lab parts below, assign a number to describe your proficiency: 1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert

- **Electronics**: How proficient are you at working with sonar and bump sensors?
- **Data Analysis**: How proficient are you at working with measurement data and error analysis?
- **Motion Control**: How proficient are you at crafting robot motion algorithms using sensor feedback?

To start the lab, you should have:

- Your robot from the previous lab.
- Sonar sensor hardware: 2 sonar sensors on mounts plus "nuts and bolts" to attach them to your robot.
- 2 bump switches (1 clockwise and 1 counter-clockwise) and 2 whiskers, plus wiring, header, heat shrink, connection plates and screws,

Physical Units

We remind you to use MKS units (meters, kilograms, seconds, radians, watts, etc.) throughout the course and this lab. In particular, this means that *whenever you state a physical quantity, you must state its units*. Also show units in your intermediate calculations.

1 Software and Development Environment Preliminaries

Follow the usual procedure to populate your working area with the provided lab code. On the Sun workstation, one person should first update the source directory `~/RSS-I-pub/` by running the following commands:

```
cd ~/RSS-I-pub
svn up
```

Now, as in previous labs, export the new `LocalNavigation` directory into your group's working copy and add it (and all subdirectories) to the list of files under version control:

```
cd ~/RSS-I-group/
svn export ~/RSS-I-pub/labs/lab5/
svn export ~/RSS-I-pub/labs/lab5_msgs/
svn add lab5*
svn commit -m "added new source for LocalNavigation lab"
```

We will again be using ROS. Recall from Lab 3/4, that there are several options for running ROS nodes (e.g., SSH forwarding, or distributing the ROS nodes on the workstation and netbook). You may find it more convenient to run your code on the workstation as you develop and debug it, but you should also ensure that it runs correctly on the laptop for autonomous operation.

2 Incorporate the Bump and Sonar Sensor Hardware

Recall from the lecture on sensors that a bump sensor is a digital on/off sensor, and a sonar is an analog sensor.

1. Create wiring harnesses for each of your bump sensors. The sensors have three connections: COM, NO (normally open) and NC (normally closed). When the bump switch is not actuated, the COM connection is shorted to the NC connection. When the switch is activated as a result of the robot hitting a wall, the COM connection is shorted to the NO terminal and the COM to NC connection is broken.

   Design your wiring harnesses such that the COM terminals are connected to ground and the NO terminals are connected to the signal line of the digital inputs on the μOrcboard. We use pull-up resistors on the μOrcboard such that when the bump switch is not activated, the μOrcboard reads a high logic level from the digital I/O pin. When the the robot hits a wall, the μOrcboard will read a low logic level.

2. Mount your bump sensors onto the front of your robot. You should have two types of bump sensor: one rotates clockwise and the other rotates counter-clockwise. You should place the clockwise sensor on the right side of the front of your robot so that the whisker points toward the outside of the robot.

   Once you have mounted the sensors and installed the whiskers, connect the bump sensors to the slow digital I/O ports of μOrcboard: the left bump sensor to port 0 and right to port 1. The COM header holes should connect to the GND pin. Route your wires neatly, and bend the bump sensor whiskers to minimize contact distance (i.e. distance from the robot to the obstacle when the bump sensor activates) and avoid hitting any mounting screws.

3. Using the small bus PCB, create a wiring harness for your sonars. Both sonars communicate with the μOrcboard using the shared I^2C communication bus. This is a two-wire, bidirectional bus where the μOrcboard is the master and the sonars are the slave devices. In addition to the data (SDA) and clock (SCL) lines used for communication, the sonars also require +5V and a ground connection.

   The μOrcboard has two I^2C ports, one that uses 3.3V and another that uses 5V. Because the sonars run on 5V, you'll need to use the 5V I2C port. Both ports are side-by-side just above channel 0 of the fast digital I/O bank. Starting from the I^2C pin closest to the fast I/O bank, the I^2C connections on the four pin header are Ground, SCL, +5V, SDA, the SDA connection being closest to the center of the μOrcboard. On the sonars, the connector pin order, starting from the pin closest to the corner and running inward is 5V, SDA, SCL, Mode (which you should leave unconnected) and Ground. For more information, consult the SRF02 technical specification: `http://www.robot-electronics.co.uk/htm/srf02techI2C.htm`

4. Before mounting the sonar sensors, test them with the robot up on blocks. You should have two sonar sensors with different IDs. As you plug in the sonars to the I^2C, you will notice the LED on the sonar make a long flash, followed by either no short flash (ID: 0) or 2 short flashes (ID: 4). The sonar with ID 0 connects the front of the robot, and ID 4 connects to the rear of the robot. Consult with the TA/LA regarding the ID of the sonar you are using.

   We will use `rxplot` to examine the output of the sonars:

   (a) Start `roscore` on the netbook with:
   ```
   ssh -X rss-student@netbook
   export ROS_MASTER_URI=http://netbook:11311
   export ROS_HOSTNAME=netbook
   roscore
   ```
   (b) Start the ROS nodes and `rxplot` from the workstation:
   ```
   export ROS_MASTER_URI=http://netbook:11311
   export ROS_HOSTNAME=workstation
   roslaunch lab5 sonar_test.launch
   ```

5. Move a large object in front of each sensor individually to identify the front and rear sensor. Then, mount the sonars on the left side of your robot using the supplied mounting plates, standoffs, and L-brackets.

Deliverables: Your report on the Wiki, named "LocalNavigation Lab Report Group N",Please include a picture of sensors and robot and let us know what was difficult, if anything. Also include a screenshot of the rxplot output.

3 Characterize your Sensors

In the field of observation, chance favors the prepared mind.
— Louis Pasteur

3.1 Getting More Sensor Data

In the previous lab, you saw how to get camera data from the robot by subscribing to `/rss/video` and potentially odometry by subscribing to `/rss/odometry`. Now you'll implement some new interfaces to process data from your bumper and sonar sensors. Use `rostopic` to identify the proper messages and examine `uorc_publisher` to determine the proper message types.

- Write a new java class called `LocalNavigation` that subscribes to the sonar and bumper data. Your event handlers should (initially) do nothing more than print out the sensor state.
- `LocalNavigation` should implement `NodeMain` and implement the `onStart()`, `onShutdown()`, `onShutdownComplete()`, and `getDefaultNodeName()` methods.

When everything seems to be working, drive your robot around manually using the `SonarGUI` program we have provided, which extends `VisionGUI` from the last lab with some new capabilities. Verify that the sensor readings are reasonable.

3.2 Collision Detection Using Bumpers

As you progress through the lab you will implement a Finite State Machine (FSM) which will ultimately be able to explore a polygonal obstacle. You will begin the development of this FSM in the next few sections by writing some of its sub-routines.

- Create a variable in your `LocalNavigation` class named `state`, and create a constant called `STOP_ON_BUMP`.

- Whenever your state is updated, publish a descriptive string on the topic `/rss/state` for debugging purposes. You could do this by creating a method which you call to set the state, or by creating a class to represent the state. A string can be published by using code similar to:

  ```
  Publisher<org.ros.message.std_msgs.String> statePub;
  ...
  statePub = node.newPublisher("/rss/state", "std_msgs/String");
  ...
  org.ros.message.std_msgs.String str = new org.ros.message.std_msgs.String();
  ...
  statePub.publish(str);
  ```

- Modify your bumper handler to stop the robot when `state == STOP_ON_BUMP` and any bumper is depressed.

Change your code so that the initial value of `state` is `STOP_ON_BUMP`, and run it to verify the behavior is as desired (drive the robot around manually with `SonarGUI`; it should automatically stop now when either bump sensor encounters an obstacle).

3.3 Robot Alignment Using Bumpers

Now you will write an FSM sub-routine which aligns the front of the robot to be roughly parallel to a planar obstacle facing the robot.

- Create new constants `ALIGN_ON_BUMP`, `ALIGNING`, and `ALIGNED`, each with a value unique relative to all other state constants.
- Modify your bumper handler so that
  - if `state` is `ALIGN_ON_BUMP` and either of the bumpers is depressed, `state` changes to `ALIGNING`
  - if `state` is `ALIGNING` (including if it was just changed),
    - if no bumper is depressed, the robot moves forward slowly
    - if only one bumper is depressed, the robot rotates slowly in the appropriate direction to try to activate the other bumper
    - if both bumpers are depressed, the robot stops and `state` changes to `ALIGNED`

Change your code so that the initial value of `state` is `ALIGN_ON_BUMP`, and run it to verify the behavior is as desired, using one of the provided obstacle surfaces (initiate the align behavior by manually driving one bump sensor into an obstacle using `SonarGUI`, or just start the robot with the sensor already activated).

Test the behavior for at least 10 different starting positions of the robot with respect to the obstacle. In each case, use the ruler and protractor provided in your lab kit to measure the angle of the robot face relative to the obstacle after the robot enters the `ALIGNED` state.

Compute the minimum, maximum, and average error angles and include these values in your Wiki.

3.4 Checkpoint 1

- Sonars and bump sensors mounted on robot.
- Demonstrate functioning sensors using `sonar_test.launch`.
- Demonstrate `ALIGN_ON_BUMP`.

3.5 Sonar Calibration and Data Segmentation

In this part of the lab you will develop code which uses the side-facing sonars to detect the presence of an obstacle wall and to incrementally estimate its start, end, and pose (location and orientation) in the world.

First, establish a global coordinate frame for your robot. For example, you may choose to use the coordinate systems marked on the floor from prior labs, or you can simply define the world frame to be identical to the robot frame at the start of each run. In either case, your robot should be at the origin and aligned to the x axis (i.e. $\theta = 0$) at the start of each run (in the latter arrangement, these hold by definition). *Clearly describe your coordinate system in your Wiki.*

- Make your `LocalNavigation` subscribe to the robot odometry, and ensure that when you start your code your robot correctly reports that it is at $(x, y, \theta) = (0, 0, 0)$, even if it has driven around in a prior run. Note that if you do not want to restart all the ROS nodes and the Orcboard between runs, you may need to write code to reset the odometry in software (`Robot.resetRobotBase()` and `Robot.setVelocity(0.0, 0.0)`) at the start of each run.
- Choose a rectangular obstacle with one face roughly 1m long, and place it about 0.5m ahead and 0.5m to the left (measured from robot frame origin) of the robot, with the selected face parallel to and facing the left side of the robot.

We have provided new ROS messages which cause `SonarGUI` to graphically display robot poses, sonar points, lines, and line segments. Create and publish new instances of these messages in your `LocalNavigation` code to display all graphical data. When you are asked to include such data in your Wiki, just take a screenshot of `SonarGUI` window showing the desired data.

You should not need to modify `SonarGUI` for any of the tasks we ask you to perform.

- Add code to your sonar handler to plot the locations of each sonar ping, in the world frame. You may assume that the reported `range` for each ping gives the perpendicular distance from the mounting location of the corresponding sonar on the left side of the robot to the detected object. Plot the front and rear sonar data with **different symbols**. Note that you will have to combine the range values with the most recent robot odometry data to produce the coordinates of sonar hit-points in world frame.

  Take a screenshots of the GUI showing both front and rear sonar pings and post it on your wiki.

Using `SonarGUI`, drive your robot slowly past the obstacle several times, starting about 0.5m before the obstacle, and ending about 0.5m after the obstacle each time. Estimate a threshold which *segments* the data into obstacle points and non-obstacle points (you may want to temporarily add print statements to display the numeric values of each ping).

Take screen shoots showing of the different runs and post the on the wiki.

- Pick two colors, one to represent non-obstacle points, and one to represent obstacle points. In your code, use the threshold to set the colors of the plotted points appropriately. Be aware that some sonars will display zero for infinite distance.

How reliably does your threshold work? Does your code ever produce a false-positive (i.e., classify a sonar point as an obstacle reflection when the sonar was unarguably not actually pointed at the obstacle)? or a false negative (a sonar point classified as a non-obstacle point when the sonar was actually pointed at the obstacle)? Take a screenshot of your GUI after a test run and post it on the Wiki.

In either case, you may wish to implement a simple low-pass filter on the sonar data (actually two separate filters, one for the front sonar and one for the rear) as it is received. For example, you could implement a sliding-window moving average filter, or an Infinite Impulse Response (IIR) filter[1] (research those terms if you are not already familiar with them). Ask a TA for help if you think you need a filter but are still unsure how to implement it.

[1]An IIR filter can be computed as $r_f = (1.0 - \alpha)r_f + \alpha r_n$ where r_f is an instance field holding the current value of the range filter, α is a constant weighting factor between 0.0 and 1.0 (e.g. 0.7), and r_n is the newly acquired range value.

3.6 Modeling the Obstacle

Now you will formulate a model of the obstacle wall as a line (and later, as a line *segment*) in world frame. The overall idea is that your robot will decide when an obstacle wall has begun, incrementally estimate the parameters of a line which fits the obstacle data points as they are acquired, and then decide when the obstacle has ended.

You'll develop this code in two stages. First, you will focus on the model acquisition and visualization. Then, in the next section, you will integrate this code into the FSM you have been developing.

You will write code to compute the parameters (a, b, c) of a line

$$ax + by + c = 0$$

which bests fits all existing sonar points (x_i, y_i) corresponding to the obstacle in the *least-squares* sense. **Appendix A gives the details of this computation. Read it carefully before you start writing the code.**

- Add structured code to your `LocalNavigation` class, or create a new class, to incrementally estimate the fit-line parameters (a, b, c). This means that your code should maintain a current estimate of of the parameters and should be able to update those estimates accordingly as new obstacle points are detected. As you design your code, pay particular attention to minimizing the amount of computation done for each newly acquired sonar point. You should also have a clean way to reset the whole process. An incremental estimator like this constitutes a simple *linear filter*.
- Adjust your sonar handler to call the line estimation code for each obstacle point (according to the threshold you identified above) detected by *either* sonar. (Re)plot a line (not a line segment) in `SonarGUI` each time any of the line parameters are updated (note that, unlike other objects, `SonarGUI` only draws one line at a time, so the last line you drew will be automatically erased).

Use `SonarGUI` to manually drive your robot slowly past the obstacle several times, with `SonarGUI` incrementally displaying your segmented (i.e. colored) sonar data, the fit-line as it is estimated, and the robot pose as it progresses past the wall. Start each run about 0.5m before the front corner of the obstacle, and end about 0.5m after the end of the obstacle. *Take a screenshot of these runs and post them on the Wiki.*

Deliverables: Your report on the Wiki should include a brief discussion and answers to any questions above. Be sure to include

- the error data you measured for your bumper alignment behavior
- a description of how you defined your world coordinate frame
- the numeric value of the threshold you used for the sonar and a description of what it means and how your code uses it
- a snapshot from the end of a sonar data run showing the collected robot poses, appropriately colored sonar data points, and final obstacle fit line.

Briefly describe the structure and operation of your linear filter code, and point out any particular challenges, bugs, or other difficulties you encountered.

4 Wall Following

In this section, you will add behaviors to your FSM which find and follow an obstacle wall in the environment. These behaviors are examples of the `Bug` algorithm described by Lumelsky and Stepanov (1987).

Select an obstacle with a planar wall about 1m long, and place it somewhere in the world frame you defined above. You will begin each test run with your robot at $(0, 0, 0)$ in world frame, and you'll use `SonarGUI` to manually point the robot towards the selected obstacle face. Once it is aligned on a (slow!) collision course, let go of the controls. Your FSM code will take over as soon as it detects the robot has encountered the obstacle with either bumper.

Temporarily disable the code in your sonar handler, set up the robot and obstacle, and verify that your existing bumper alignment routine is working correctly. The robot should reliably approach the obstacle, align to face it, and stop.

Now you'll implement a new subroutine which aligns the robot so its left side is parallel to and facing the obstacle:

- Add code to your bumper handler which is triggered when `state` is `ALIGNED` (including if it has just entered that state). Once aligned, the robot should
  - back up a small amount
  - stop
  - rotate clockwise $\pi/2$ radians
  - stop and enter a new state `ALIGNED_AND_ROTATED`

  The robot should end up with its left side parallel to the obstacle, separated by a distance d of about 0.5m (from the wall to robot frame origin). You may wish to implement one or more intermediate states, and you may adjust the actual distance d as you see fit (d will also be important for the tasks you will perform in the next sections, so you may need to adjust it as you go along).

4.1 Finding the Start of the Wall

Your robot is now almost ready to follow the obstacle wall. Because we will ultimately be interested in *touring* the whole obstacle perimeter, we will first back the robot up to the start of the wall (observe that, so far, the robot may currently be at any point along the obstacle wall):

- add code to an appropriate event handler so that when `state` is `ALIGNED_AND_ROTATED` (including if it was just changed) the state is immediately changed to a new state, `BACKING_UP`.
- re-enable your sonar handler, and adjust it so that
  - if `state` is `BACKING_UP` and an obstacle is detected with either sonar (using your threshold code) the robot moves slowly backwards, tracking the wall, as described below.
  - it uses your linear filter to maintain continuous estimates of the obstacle fit-line parameters, and it (re)plots the fit line in `SonarGUI` whenever its parameters change. Also, re-enable (if necessary) your code which plots the colored sonar data points.
  - if `state` is `BACKING_UP` and an obstacle is *not* detected with either sonar, the robot stops and enters a new state `FINDING_WALL`.

To track the wall, implement a feedback controller (you may find P or PD control most appropriate) which computes robot velocity commands to keep the robot approximately parallel to the wall at distance d.

Hint: You may wish to remind yourself of the formula for the perpendicular distance from a point—i.e. the robot center point as reported most recently to your odometry handler—to a line (see appendix A), from which you can derive the translational error term.

Hint: You can derive the orientation error term either from the robot orientation reported by odometry *or* from the sonar data when both sonars have detected the object, or from a combination of the two sources.

Write your controller code in a structured way and trigger it to compute new robot velocity commands either periodically (e.g. every 50ms) or whenever new odometry or sonar data is supplied.

Hint: If you add a Java `Timer` to call your control code periodically, be aware that it will be executing in a different thread than your ROS event handlers. Any state variables which can be read in one thread and written in another require synchronization in both places.

4.2 Following the Wall

Now we're ready to find and follow the wall. Add code to your sonar handler so that:

- the robot moves slowly forward whenever `state` is `FINDING_WALL` and neither sonar detects an obstacle
- whenever `state` is `FINDING_WALL` and either sonar detects an obstacle, `state` changes to a new state `TRACKING_WALL`, the linear filter is reset, and the current robot pose (x, y, θ) and sonar readings are stored in instance fields for later use.
- as long as `state` is `TRACKING_WALL`, the robot moves slowly forward, tracking the wall with your feedback controller, and updating the linear filter. Again, `SonarGUI` should show a continuously updating display of the robot pose, the current obstacle fit line, and the colored sonar data points.
- whenever `state` is `TRACKING_WALL` and *neither* sonar detects an obstacle, the robot stops, and `state` changes to a new state `WALL_ENDED`. The wall fit line should be erased from `SonarGUI`; you will now compute a line *segment* which more completely represents the wall. Use the current fit line parameters, the data you stored above when the robot first found the wall, and the current robot pose and sonar readings to estimate the endpoints of a line segment representing the wall, and add this segment to the display in `SonarGUI`.

Hint: You have more data than is necessary to define a line segment, so you will need to develop a method to reduce the data to compute the segment. Estimate the relative *certainties* of the different kinds of data you have, and try to rely primarily on the most certain values. For example, you may decide to make the line segment coincident with the obstacle fit line, and use the other data only to find the positions of the two endpoints on this line.

4.3 Measuring Performance

To quantify the performance of your code, you will now collect some data and plot it.

- add a new boolean instance field to your `LocalNavigation` class called `saveErrors`
- add code to your feedback controller to write a line to an ASCII data file with the format

  ```
  timestamp translation_error rotation_error
  ```

  at each control update whenever `saveErrors` is true. The errors should reflect the difference between the goal (distance and orientation from the wall) and the actual.
- *use gnuplot, matlab or a spreadsheet program of your choice to plot each error vs. time for a single complete run of your wall following behavior. At the end of the run, save a screenshot of the final state of the* `SonarGUI` *window. Post both the plot and screenshot on the Wiki.*

Deliverables: Your Wiki should include your error plots, screenshots, a brief description of your procedure, a discussion of the architecture and operation of your controller code, and descriptions of any tasks you found particularly challenging. Be sure to state the d *value you have used. Be prepared to demonstrate your entire wall following behavior at the start of the next lab.*

5 Checkpoint 2

1. Drive the robot alongside a wall. The SonarGUI should be displaying the sonar data and the best-fit line estimated from that data.
2. Demonstrate your robot finding and following a wall.

6 Model Acquisition

It's a walk-off!
— Billy Zane, Zoolander

In this final part of the lab you will complete the FSM to acquire a geometric model for an obstacle in the environment. You can assume that the obstacle is a simple closed convex polygon, and that each side of the obstacle is at least 1.5 times as wide as the front of your robot. You should not assume that your robot knows anything else about the obstacle, including the number of sides.

The FSM you have developed up to this point can serve as the initial part of the model acquisition algorithm. Extend it so that:

- at the end of each wall `state` is reset to `ALIGN_ON_BUMP`, and the robot is commanded with velocities that drive it slowly counter-clockwise along a circle of radius d tangent to its current heading.
- for each wall, the obstacle points, fit line, and fit segment are all plotted in a color c which is distinct from all the other wall colors (you may just randomly pick c for each wall, a random color generator is provided in `SonarGUI`).
- the robot stops and enters a new terminal state, `DONE`, when it detects that it has toured the entire obstacle.

The main challenge here is to figure out how to decide when the tour is complete (and to get all the details right). There is more than one way to determine completion. Consider different possibilities, describe them in your writeup, and think about and document the relative strengths and weaknesses you expect for each. Specifically consider the relative effects of sensing and command uncertainty that you predict, and think about and document how you could make your termination procedure more robust to them.

Hint: what is the sum of the *external* angles of a convex polygon?

Test your behavior for at least two trials on at least two obstacles of different shapes (at least four trials total). *Take screenshots of the final* `SonarGUI` *window at the end of a good run on each obstacle, and include corresponding error plots (i.e. at least two screenshots and four error plots). Capture these runs on video. (be sure your videos include identifying information at the start).*

Does your robot always complete the skills you programmed? If not, what are the most common sources of error? How long does it take for the robot to complete the model acquisition task for each of the two objects you used? Can these running times be improved? If so, how, and what might the trade-offs be, if any?

Deliverables: Your report on the Wiki should include a brief description of your procedure, the dimensions of the obstacles you used, answers to the questions above, your error plots and screenshots, the velocities you used to drive the robot in a circle, and a detailed discussion of the alternative termination methods you considered, the one you implemented, and why. Be prepared to demonstrate your entire model acquisition behavior at the start of the next lab. Also, somewhere on your Wiki you should include your FSM with all transitions and states.

A Appendix: Lines in the Plane

We can represent any line in the plane as the locus of points (x, y) which satisfy the equation

$$ax + by + c = 0 \tag{1}$$

where (a, b, c) are three parameters (constants) which define the position and orientation of the line. We require that not all three of the parameters are simultaneously zero, since if that were true then all (x, y) would trivially satisfy (1). This implies that it can not even be the case that $a = b = 0$, since then c would have to be non-zero, and *no* point (x, y) could satisfy (1).

As a geometric object, a line in the plane actually has only *two* degrees of freedom (DoF). There are many ways to define these, for example

- slope and y intercept (what happens when line is vertical?)
- slope and x intercept (what happens when line is horizontal?)
- x intercept and y intercept (what happens when line is either vertical or horizontal?)
- angle with respect to x-axis (which angle, precisely?) and perpendicular distance from origin
- etc.

As you can see, many of the possible ways to define them have issues in certain cases. We'd of course like to avoid such situations as much as possible, since we would like to be able to represent *any* line without failure.

Moreover, we started by defining the line according to equation (1), which has three parameters, not two. There must be some redundancy among those three parameters. It turns out that we can add an *algebraic constraint* among the parameters which reduces their effective DOF to two:

$$a^2 + b^2 = 1. \tag{2}$$

So now (a, b, c) define a line according to (1) iff (2) holds. Or, said in a more useful way, if you are given *any* triple of parameters (a', b', c'), not all zero, as described above, but not necessarily satisfying $(a')^2 + (b')^2 = 1$, you can simply compute a new set (a, b, c) of parameters *for the same line*:

$$l_n := \sqrt{a^2 + b^2} \qquad \text{(:= means “defined as”)}$$

and then

$$a = a'/l_n \qquad b = b'/l_n \qquad c = c'/l_n.$$

This will always be possible because l_n can only be zero if both a and b are zero. To see that this procedure does not change the line, think of it as dividing the entire equation of the line by l_n.

Now we can always be sure that (2) holds, so the DoF of the parameter space (three parameters - one constraint = two DoF) matches the DoF we expect to see for a line. Further, this particular constraint is convenient because it confers a straightforward geometric meaning to the parameters (a, b, c). To see it, rewrite equation (1) in vector form:

$$\begin{aligned} \mathbf{n} &:= (a, b) \\ \mathbf{p} &:= (x, y) \\ \mathbf{n} \cdot \mathbf{p} + c &= 0. \end{aligned} \tag{3}$$

Constraint (2) ensures that $\mathbf{n}$ is actually a unit vector. So, recalling that the dot product of a vector with a unit vector returns the length of the component of the first vector in the direction of the second, we can interpret (3) to mean that $\mathbf{p}$ is on the line iff its component in the direction of $\mathbf{n}$ has length $-c$. Since c is fixed for all $\mathbf{p}$, it turns out that this geometrically means that $\mathbf{n}$ is the *unit normal* to the line, and c is the (signed) distance from the origin to the line in the *opposite* direction of $\mathbf{n}$. The unit normal $\mathbf{n}$ may either point towards the line from the origin, in which case c will be negative, or $\mathbf{n}$ will point away from the line, in which case c will be positive–the math holds either way.

A.1 Perpendicular Distance from a Point to a Line

Our parametrization, combined with constraint (2), is even more convenient because it gives a very simple formula for the perpendicular (i.e. shortest) distance from any arbitrary point $\mathbf{q}$ in the plane to the line. Again we use the dot product to find the length of the component of $\mathbf{q}$ in the direction of $\mathbf{n}$. The difference between $-c$ and this product will be the signed perpendicular distance d from $\mathbf{q}$ to the line (a, b, c), in the direction of $\mathbf{n}$:

$$d = -c - \mathbf{n} \cdot \mathbf{q} \tag{4}$$

Note that as long as (2) holds the absolute value of d is the same as the absolute value of the expression to the left of the equal sign in (1), i.e.

$$|d| = |ax + by + c| \tag{5}$$

where $\mathbf{q} = (x, y)$, which makes sense—points on the line have zero distance from the line.

A.2 Fitting a Line to a set of Points with Least-Squares

If you are given a set of points in the plane (x_i, y_i), at least two of them distinct, it may make sense to try to find a line (a, b, c) which somehow best *fits* the points. There are different ways to define the definition of "fit". It turns out that one particularly convenient way is to define an error term for each point which increases in absolute value in proportion to the perpendicular distance from that point to the fit line, and then find the line which minimizes the sum of the squares of these error terms.

We will not provide a derivation or justification for this process here[2], but we will give you the bottom line (ahem...): a set of equations which you can use to compute (a, b, c) parameters of the best-fit line.

It's unfortunately a bit more complex to directly find the best-fit (a, b, c) which also satisfy the above constraint (2), so instead we'll use a simpler constraint

$$c = -1. \tag{6}$$

for this computation. After the best-fit parameters are computed, you may of course use l_n as above to compute a new triple of parameters for the best-fit line which satisfy (2).

(6) has already given us the value of one of the three parameters of the best-fit line. The other two may be computed like this:

$$X := \sum_i x_i \qquad Y := \sum_i y_i \qquad X_2 := \sum_i x_i^2 \qquad Y_2 := \sum_i y_i^2 \qquad Z := \sum_i x_i y_i$$

$$D := X_2 Y_2 - Z^2 \tag{7}$$

$$a = \frac{XY_2 - YZ}{D} \tag{8}$$

$$b = \frac{YX_2 - XZ}{D} \tag{9}$$

As you implement this, note that the computation will be numerically ill-conditioned if D is near zero, which will occur if you have fewer than two distinct points, and also potentially in other situations. One way to handle this is to compute D first, check whether it is very close to zero, and if so, to skip this fit-line update (i.e. to continue using the previously-computed fit line parameters, if any).

[2]If you are really interested (and we hope you are!), read about the *Moore-Penrose Pseudoinverse*, which is the general least-squares linear system solution method upon which the following equations are based.