

Algorithms for Robotics

or: An Unofficial Companion Guide to the Georgia
Institute of Technology's **CS7638**: *Robotics: AI Techniques*



George Kudrayvtsev

george.k@gatech.edu

Last Updated: April 24, 2020

(Draft)

0	Preface	3
1	Introduction to Probability	5
1.1	Localization	5
1.2	Probability	7
2	Sensing	8
3	Planning	9
3.1	An Algorithm: Greedy Search	11
3.2	Heuristic-Based Search	13
3.3	An Algorithm: The All-Seeing Eye	14
4	Moving	17
4.1	Smoothing	17
4.2	PID Controller	18
4.2.1	P is for Proportional	18
4.2.2	D is for Derivative	20
4.2.3	I is for Integration	21
	Integral Windup	21
4.2.4	Parameter Search: TWIDDLE	22
5	Simultaneous Localization and Mapping	25
5.1	Constraints	26
5.1.1	Motion Constraints	27
5.1.2	Landmark Constraints	28
5.2	Generalization	29

5.2.1	Introducing Noise	30
5.3	Summary	31
Index of Terms		33

PREFACE

I read that Teddy Roosevelt once said, “Do what you can with what you have where you are.” Of course, I doubt he was in the tub when he said that.

— Bill Watterson, *Calvin and Hobbes*

Before we begin to dive into all things robotics, here are a few things I do in this notebook to elaborate on concepts:

- An item that is **highlighted like this** is a “term;” this is some vocabulary or identifying word/phrase that will be used and repeated regularly in subsequent sections. I try to cross-reference these any time they come up again to link back to its first defined usage; most mentions are available in the [Index](#).
- The presence of a **TODO** means that I still need to expand that section or possibly to mark something that should link to a future (unwritten) section or chapter.
- An item in a **maroon box**, like...

BOXES: A Rigorous Approach

... this example, often represents fun and interesting asides or examples that pertain to the material being discussed. They are largely optional, but should be interesting to read and have value, even if it’s not immediately rewarding.

- An item in a **blue box**, like...

QUICK MAFFS: Proving That the Box Exists

... this example, is a mathematical aside; I only write these if I need to dive deeper into a concept that’s mentioned in lecture. This could be proofs, examples, or just a more thorough explanation of something

that might've been “assumed knowledge” in the text.

INTRODUCTION TO PROBABILITY

73.6% of all statistics are made up and 90% of quotes are mis-attributed.

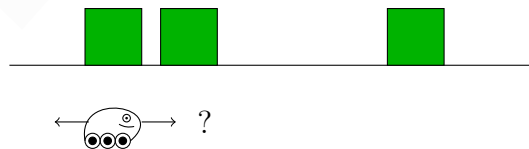
— Abraham Lincoln, *The Internet*

WE’LL open our discussion of algorithms for robotics applications by discussing probability. However, we’ll take an example-based approach, working up to a method for **localization** using basic probability theory.

Localization is the need to determine where you are in the world. This is an important skill for humans and robots alike and can be achieved to varying degrees of success; some of us just have a “bad sense of direction.” Given a layout of some area, a robot should be able to determine where it is based on sensor readings or other measurements.

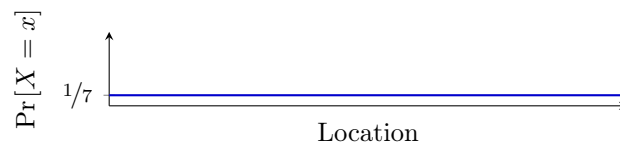
1.1 Localization

Suppose a robot is in a simple hallway with three doors:



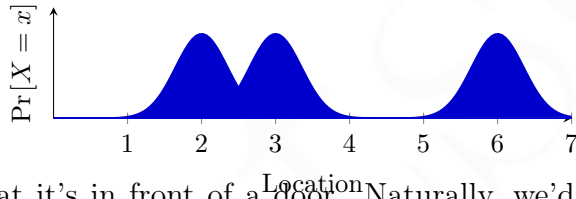
We see that it’s in front of the first door, but the robot doesn’t know that. Instead, it has a **probability distribution** representing its confidence at each point in the hallway. For the sake of simplicity, suppose we discretize the hallway into 7 spots; then, we can say that X represents the distribution of probabilities for each position in the hallway. Since the robot knows nothing about its location yet, this is a **uniform distribution**:

$$X = \{1/7, 1/7, 1/7, 1/7, 1/7, 1/7, 1/7\}$$



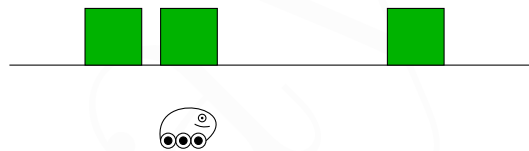
For the robot to increase its certainty about the situation, it needs to perform a *measurement*. The distribution before the measurement is called the **prior** or **belief** distribution; afterwards, it's called a **posterior**. Suppose

the robot measures, with some doubt, that it's in front of a door. Naturally, we'd expect its location probability distribution to now peak at doors.

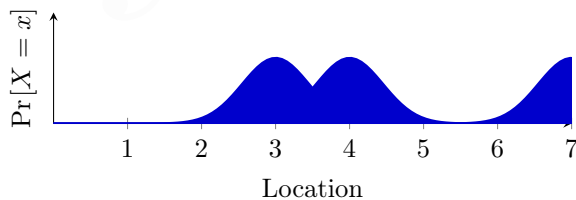


The reason why we don't have hard spikes is because of uncertainty in our measurement: sensors are never perfect, so we need to take into account the fact that it's possible we measured "door" when there in fact wasn't one.

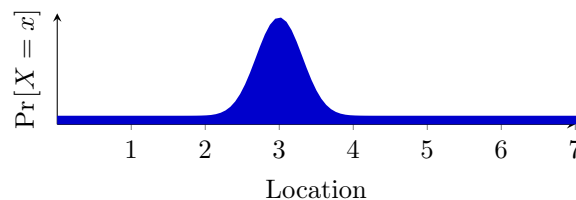
We continue iterating on the localization process by doing this again after a movement. Suppose we move one step to the right:



Our prior (pre-measurement) belief distribution should shift accordingly, but it also flattens out a bit because again, movements are not perfect, so we need to take into account the possibility of not moving exactly 1 unit.



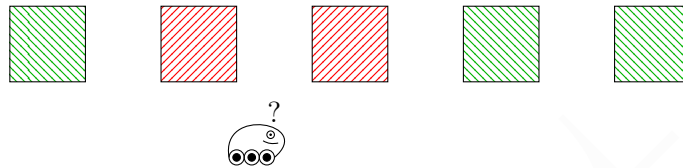
Our robot takes a measurement and determines it's in front of a door again. Well, there's only one place in our map in which you can measure a door, move right, then *still* measure a door, and our posterior distribution reflects this:



If you followed this logic, you understand probability and localization!

1.2 Probability

With this intuition under our belt, let's make things a little more concrete. We'll work with a simpler example, keeping our little robot friend around. We have a map of five tiles, each of which is either **red** or **green**.



Our initial distribution is uniform with each position x having $\Pr[X = x] = 1/5 = 0.2$. Suppose our robot claims to measure that he's on a **red** tile. As before, we expect our distribution to be biased towards red tiles. Mathematically, this is just a product. We can choose an arbitrary large number of red tiles and a smaller one for green tiles, say, **red** = 0.6 and **green** = 0.2. Then we multiply our distribution accordingly:

	$1/5$	$1/5$	$1/5$	$1/5$	$1/5$
\times	0.2	0.6	0.6	0.2	0.2
$=$	0.04	0.12	0.12	0.04	0.04

This becomes our belief! The last step is to turn it into a valid probability distribution; it doesn't exactly make sense for only 36% of the cases to be covered. We need to **normalize** our distribution by dividing it by its sum:

	0.04	0.12	0.12	0.04	0.04
\div	0.36	0.36	0.36	0.36	0.36
$=$	$1/9$	$1/3$	$1/3$	$1/9$	$1/9$

Each of these values can formally be described in the form $\Pr[X = x_i | Z]$, the **conditional probability** of being at cell x_i (where x_i is in the set of all possibilities, X) after measuring (or *observing*) Z .

This is called the **measurement update** step and is an integral part of localization.

SENSING

All our knowledge begins with the senses, proceeds then to the understanding, and ends with reason. There is nothing higher than reason.

— Immanuel Kant, *Critique of Pure Reason*

T **HOUGH** this chapter may be expanded in the future with supplemental content, I've already covered this content at length during *Computer Vision*. I highly recommend just heading over to [Chapter 12: Tracking](#) for the overview on both **Kalman filters** and **particle filters**.

PLANNING

Life is what happens to us while we are making other plans.

— Allen Saunders

THE various tracking methods we covered in the previous chapter gives us information about the world. Now that we are “localized,” how can we use that information to actually take action and achieve some goals? Enter **planning**.

There are many situations in which we have to find the optimal path from a starting location to some destination. It could come from a literal use-case, like how a mapping application tries to find you the optimal way to stop at In-n-Out on your trip across California, or it could come from a more figurative use-case, like deciding the optimal chess move by searching the “game tree.” As far as the algorithms are concerned, these are equivalent: you are searching through a set of possible states and transitions between them to find an optimal path.

A TALE OF TWO APPS: Google vs. Waze

Fascinatingly-enough, these two routing applications (which ultimately belong to the same company, Alphabet) often offer completely different routes from one destination to another. This can largely be attributed to a philosophical difference that drives the two applications: Google Maps is more concerned with a simple, familiar, and easy-to-follow route from one place to another that takes big, recognizable streets and sometimes offers slight alternatives that may save time but typically don’t introduce extra complexity; on the other hand, Waze is community-driven, meaning you can use it to aggressively find convoluted shortcuts that will shave milliseconds off of your trip time at your tires’ expense (given all of the extra turning you’ll be doing).

When we get into the idea of a [cost function](#) shortly, we will see how different choices for this can greatly affect the plan.

Check out [this discussion](#) for a little more on Waze and Google if you're interested.

When discussing these planning, path-finding, or search algorithms (these terms are often interchangeable) we typically use an example of a maze. Given that we are working with robots and have real contexts that we can apply this to, we'll be considering how an autonomous vehicle can navigate a handful of intersections without maiming its passengers.

Consider the following situation, in which the **blue** car wants to get to the specified destination and has two plans:

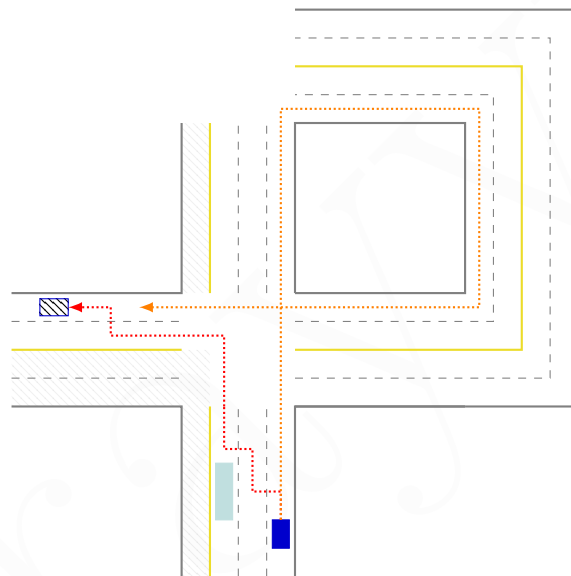


Figure 3.1: A complicated, ever-evolving situation we may commonly face when driving.

The **red** plan has a high risk: merging across two lanes in such a short span of time (with the optional theoretical **teal truck** in the way) and will also likely require waiting at a red light, but it's also the shortest path. On the other hand, the **orange** path is simpler and doesn't run the same risks, however it traverses a further distance. Which path do you take?

FUN FACT: UPS Driving Patterns

If you follow the matra of UPS drivers, the **orange** path is always the better choice: UPS drivers almost [never turn left](#).

More importantly, what path does an autonomous vehicle take, and how can we

encode this algorithmically?

In this chapter, we'll cover robot motion planning methods, starting with *discrete* methods in which the world is “chopped up” into small bins and moving on to *continuous* motion planning methods. Our problem can be defined more concretely. Given:

- a map
- a starting location
- a goal location
- a cost function

We want to find **the path from start \rightarrow goal with the lowest cost.**

Once we have the algorithms in place, we'll see that the only thing that really varies is the cost function. Let's define it a little more. A **cost function** defines an association between an action with a cost. For example, “just go straight” might cost 1, whereas going diagonally might cost $\sqrt{2} \approx 1.4$ (thank u Pythagoras). In such a world, going diagonally would always be better than going straight, turning, and going straight again!

3.1 An Algorithm: Greedy Search

As is tradition, let's imagine a simple grid-based maze:

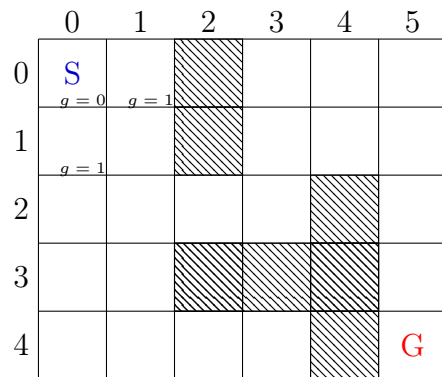
	0	1	2	3	4	5
0	S					
1						
2						
3						
4						G

Where we wish to reach the **goal** from the **starting** point. Suppose our set of possible actions is movement in any of the 4 directions (no diagonals) and they all cost the same.

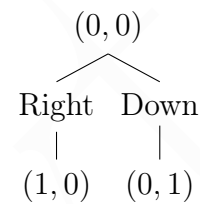
We begin exploration from $(0,0)$ by *expanding* to neighboring nodes, meaning they get added to what we call the “open list.” Furthermore, we've expanded on $(0,0)$ and returning to it wouldn't make sense, so it gets added to the “closed list”:

$$\text{open} = [(1,0), (0,1)] \quad \text{closed} = [(0,0)]$$

Moving to both of these from our root costs 1 unit which we accumulate into what's called the g -score, so our state is something like this:

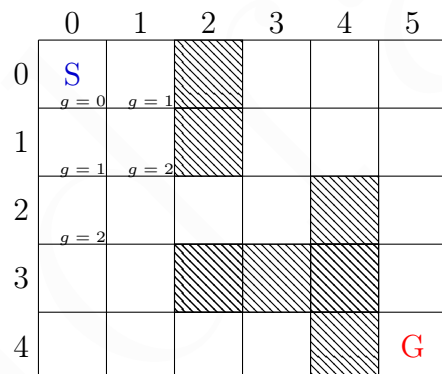


(a) The partially-explored map.

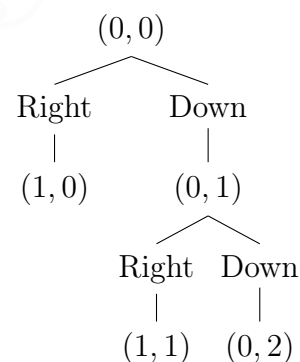


(b) The exploration of the game tree.

From there, we expand on the lowest-cost node in the open list. In this case they have equal scores so the choice is arbitrary; let's go down.

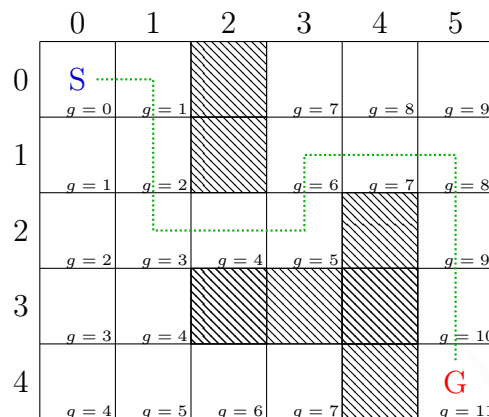


(a) The partially-explored map.



(b) The exploration of the game tree.

We continue this process recursively, continually expanding the node with the lowest g -score, adding its neighbors to the open list while ignoring nodes on the closed list. The whole algorithm is described in [algorithm 3.1](#). Finally, we end up with a path to the goal with the lowest possible cost (note that the exact path depends on tie-breaking strategies):



We can (but won't) prove the fact that this is the optimal result because we always (that is, greedily) choose to explore towards the lowest g -score which represents the accumulated cost. This method works but has a massive downside: we had to explore a large chunk of the map that led us nowhere to find this path.

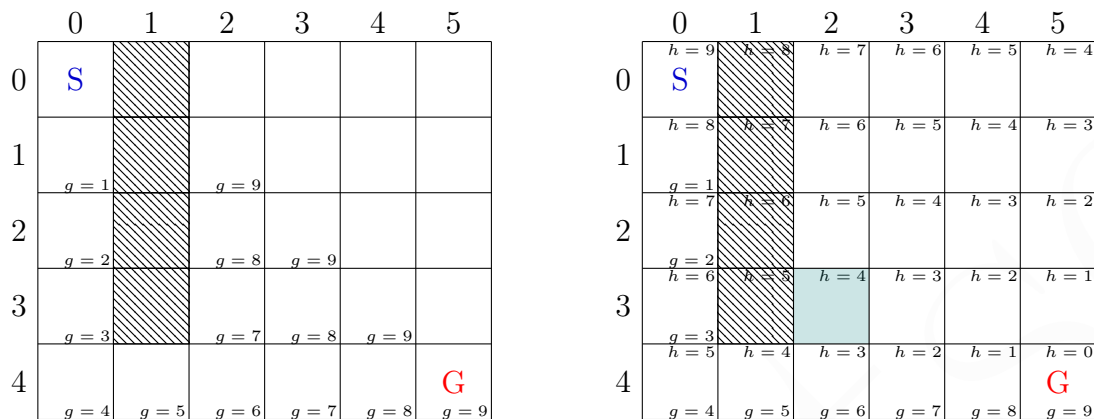
3.2 Heuristic-Based Search

We can avoid searching too much of the map by introducing the concept of a **heuristic** to our search. A heuristic is a rough approximation about how far away we are from the goal at a given cell; it can help break ties and actually help us avoid exploring large swaths of the map. While the g -score is our total cost *from the start*, the heuristic is an approximate cost *to the goal*. Their sum typically provides a good metric for the “optimality” of a cell and is the basis for the **A* search algorithm**.

For grid-based search, for example, a common heuristic is the **Manhattan distance**, which is a cell's $\Delta x + \Delta y$ from the goal.¹ For example, the Manhattan distance at (1,2) in the grid above is $(5 - 1) + (4 - 2) = 6$. If you wanted to allow diagonal movements, you could just use the Euclidean distance as your heuristic. The most important part about the heuristic is that it **must** be optimistic—it should **never overestimate**. More specifically, $\forall x, y : h(x, y) \leq \text{truth}$, where the *truth* is how long it really would take to reach the goal from (x, y) . Following this rule will guarantee that A* will find the optimal solution while doing the minimal amount of work.

To illustrate A*, we will use the same grid as before but with different obstacles. On the left is the exploration after a greedy search, while on the right is the exploration using A* with a Manhattan distance heuristic:

¹ The term “Manhattan” distance comes from the fact that it's a metric of the amount of blocks you'd need to walk through Manhattan's grid to get to your destination.



Notice that A* only explores the exact cells it needs to. The incorporation of the heuristic lets it automatically prefer cells that get closer to the goal. There is no sense in exploring (2,3), for example (marked in teal above) when it takes us further from G than (3,4).

The only difference between greedy search and A* is that we now track the f -value, the sum of the accumulated cost and the heuristic:








$$f = g + h(x, y)$$

We now choose the node with the lowest f value rather than g value. That's it! That's A*. I highly recommend watching [this video](#) on A* in action in a self-driving car simulation that combines localization and search to plan paths that traverse complicated mazes and elaborate scenarios.

3.3 An Algorithm: The All-Seeing Eye👁️

Recall our situation in Figure 3.1: we were unsure if we would have the time to get ahead of (or behind) the truck to make the left turn. As we act, the world around us changes. There is a chance that we won't be able to make the turn, so it's useful to know what the optimal strategy is after we instead cross the intersection.

In fact, it'd be useful to know what the optimal strategy to reach a goal is at *any* point on the map. Of course, we could perform A* from every cell, but that would be computational overkill. Instead, we will explore an algorithm that uses dynamic programming to determine the best action at every cell; this will be our “policy” map. For example:

	0	1	2	3	4	5
0	↓	↓		→	↓	↓
1	↓	↓		→	→	↓
2	→	→	→	↑		↓
3	↑	↑				↓
4	↑	↑	←	←		G

To see how we can compute this policy map efficiently, let's introduce a new concept: value. The *value* of a cell is how much it costs to get to the goal from that cell. Naturally, the value at the goal is 0. The value at its immediate neighbors is 1. At their neighbors, 2, and so on... We can compute the value by recursively iterating like this from the goal:

$$f(x, y) = \min_{x', y'} [f(x', y') + 1]$$

where our initial values are:

$$f(x, y) = \begin{cases} 0 & \text{if } (x, y) \text{ is the goal} \\ \infty & \text{otherwise} \end{cases}$$

ALGORITHM 3.1: A greedy search algorithm.

Input: A grid-like map of the world, M .

Input: A mapping of actions and their respective costs, $F(a) \mapsto c$.

Input: The start and goal locations, S and G .

Result: An optimal path, P^* .

```
/* A way to track the costs of closed nodes (0=open) in the world.
```

```
*/
```

```
closed := ZEROES( $M.shape$ )
```

```
closed[ $S$ ] := 1
```

```
/* Start the open list with our initial location */
```

```
 $p := (S_x, S_y)$ 
```

```
 $g := 0$ 
```

```
 $open := [(g, p)]$ 
```

```
while true do
```

```
  if  $open = \emptyset$  then
```

```
    | return failure
```

```
  end
```

```
   $n_g, n_x, n_y :=$  node in  $open$  with lowest  $g$ -score
```

```
   $g = n_g$ 
```

```
  if  $(n_x, n_y) = G$  then
```

```
    | break
```

```
  end
```

```
  /* Store each unexplored action result in the open list. */
```

```
  foreach valid action  $a := (dx, dy)$  do
```

```
    |  $p_a := (n_x + dx, n_y + dy)$ 
```

```
    if  $closed[p_a] = 0$  then
```

```
      |  $g_a := g + F(a)$ 
```

```
      |  $closed[p_a] := g_a$ 
```

```
      | PUSH( $open, (g_a, p_a)$ )
```

```
    end
```

```
  end
```

```
end
```

```
/* Now that we know we have a path with the lowest cost, simply
   traverse the map from  $G \rightarrow S$  while choosing the cheapest node to
   find it. */
```

```
 $P^* := \emptyset$ 
```

```
 $p := G$ 
```

```
while  $p \neq S$  do
```

```
  |  $p := \min_{n \in \text{NEIGHBORS}(M[p])} closed[p]$ 
```

```
  | PUSH( $P^*, p$ )
```

```
end
```

```
return  $P^*$ 
```


MOVING

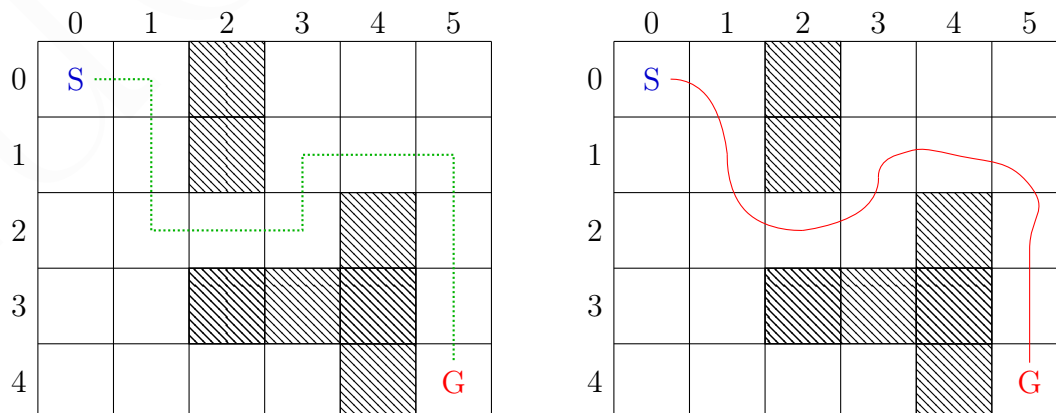
And the danger is that in this move toward new horizons and far directions, that I may lose what I have now, and not find anything except loneliness.

— Sylvia Plath, *The Unabridged Journals of Sylvia Plath*

WITH a solid understanding of how we can sense the world around us, localize our position within it, and create a plan to reach some goal location based on our perception of the world. In this chapter, we'll work through turning that plan into actual actions our robot is capable of doing in order to step through the plan. Specifically, we'll be looking at how to generate smooth paths and how to control our robot through a method called **PID control**.

4.1 Smoothing

Recall that in **Planning**, we worked with a 2 dimensional, discrete grid of the world, and our plan had very rigid actions in mind. For example, we would need to *fully stop* at (1, 0) and make a 90° turn before moving again. Ideally, we would be able to keep moving the entire time and make smooth turns to traverse our path:



Let's formalize this idea. Our n -length path previously was a set of points denoting each grid $\{x_0, x_1, \dots, x_{n-1}\}$. To smooth it out, we'll start with a new variable y_i , where $y_i = x_i$. Then, we optimize two things at once: the distance between each y_i and its corresponding x_i , and the distance between each y_i and its neighbor y_{i+1} :

$$\min \begin{cases} (x_i - y_i)^2 \\ (y_i - y_{i+1})^2 \end{cases}$$

If we optimized either of these individually, we would get either the original path or no path at all, respectively. Thus by finding the perfect balance between the two, we can generate a smooth path. We can also introduce a scalar, α , to vary the level of "smoothness" we want. Thus, we try to minimize this cumulative distance:

$$\varepsilon(x_{0..n-1}) = \min_{y_{0..n-1}} [(x_i - y_i)^2 + \alpha(y_i - y_{i+1})^2]$$

Now note that we still actually want to get to our destination... so we operate under the additional constraint that $y_0 = x_0$ and $y_{n-1} = x_{n-1}$.

We solve this via **gradient descent**. We rearrange our optimization process a bit to iteratively reassign to y_i . Each time, we add a term that is proportional (based on α) to the deviation of our current y_i from x_i . Similarly, we add a term that is proportional (based on β) to the deviation of our current y_i from *both* of its neighbors, y_{i-1} and y_{i+1} :

$$\begin{aligned} y_i &= y_i + \alpha(x_i - y_i) \\ y_i &= y_i + \beta(y_{i-1} + y_{i+1} - 2y_i) \end{aligned}$$

Empirically, values of $\alpha = 0.5, \beta = 0.1$ apparently work well. The process of gradient descent involves continually making this adjustment until the change in y_i stops being significant (indicating convergence). We formalize this approach in [algorithm 4.1](#).

4.2 PID Controller

In this section, we'll build up the blocks towards the idea of a **PID controller**.

4.2.1 P is for Proportional

Suppose we want to control a self-driving car. If we wanted to follow a straight path, in a perfect world the obvious ideal action would be to not adjust the steering whatsoever. That, of course, assumes we're already on that path... ideally, we'd find a way to get there, instead. A reasonable way to ensure that a car meets and then follows a straight trajectory is by minimizing what's called the **crosstrack error**, which is a measure of how far we are from the goal path:

ALGORITHM 4.1: Applying gradient descent to smoothing a path.

Input: An $(n + 1)$ -element path, $\mathbf{p} = \{p_0, p_1, \dots, p_n\}$, where each p_i is actually a 2-dimensional point (x_i, y_i) .

Input: The proportionality factors, α and β .

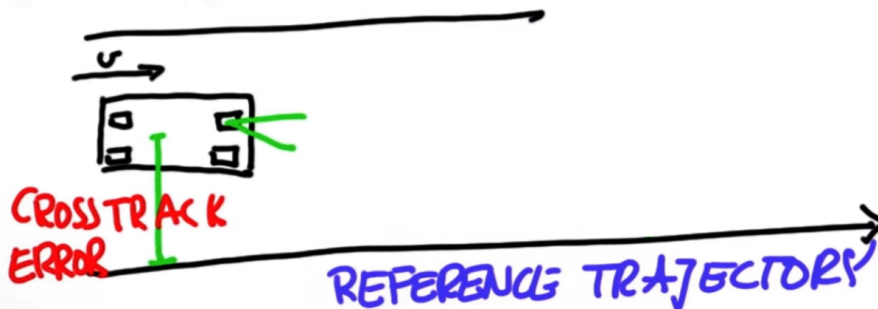
Input: Optionally, a threshold that indicates convergence, $\tau = 1 \times 10^{-6}$.

Result: A smoothed path, $\mathbf{q} = \{q_0, q_1, \dots, q_n\}$, fixed at the endpoints, so $q_0 = x_0, q_n = x_n$.

```

 $\mathbf{q} := \mathbf{p}$ 
do
   $\Delta := 0$                                      // track the total change to the path
  foreach  $i \in \{0, \dots, n\}$  do
     $\delta := \alpha \cdot (q_i - p_i)$ 
    /* If we have neighbors, adjust accordingly... If you want a
       cyclical path, do these absolutely, just operate with  $i$ 
       under  $(\text{mod } n)$ . */
    if  $i < n$  then
       $\delta += \beta \cdot (q_{i+1} - q_i)$ 
    end
    if  $i > 0$  then
       $\delta += \beta \cdot (q_{i-1} - q_i)$ 
    end
     $q_i += \delta$                                      // adjust smoothed point accordingly
     $\Delta += |\delta_x| + |\delta_y|$                  // track absolute change
  end
while  $\Delta > \tau$ 
return  $\mathbf{q}$ 

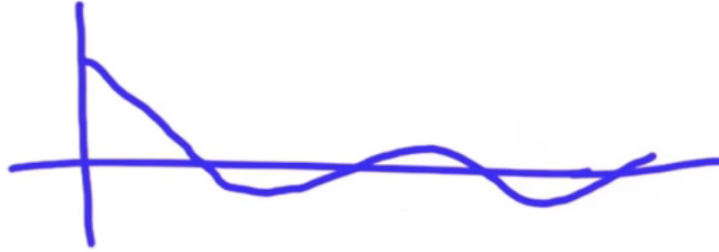
```



The idea is that there is a metric—the crosstrack error—that tracks the vector between the current trajectory and the reference trajectory; by minimizing this error (staying proportional to the path), you follow intended trajectory. Mathematically,

we model this using τ , a scalar that controls how “aggressive” we are in minimizing the error. Given a crosstrack error ε , we’d say that $\alpha = \tau\varepsilon$, where α is the resulting steering angle.

Unfortunately, a P-controller has a tendency to overshoot: intuitively, we can imagine that on the last “step” in which the crosstrack error goes to zero, the car’s wheels will still be oriented in previous direction.



It may eventually always be *very* close to the reference trajectory, but will never converge perfectly. We would call this controller **marginally stable**. Larger values of τ cause larger oscillations around the reference trajectory.

4.2.2 D is for Derivative

Is there a way eliminate this oscillation? We don’t really want to waver back-and-forth around the ideal trajectory—that sounds like a recipe for motion sickness. The trick is to incorporate the reduction of error based on our existing trajectory by using the temporal derivative. Since we typically work over discrete chunks of time, we calculate our derivative the same way, tracking some Δt change in time between each adjustment:

$$\alpha = -\tau_p \varepsilon - \tau_d \cdot \frac{d}{dt} \varepsilon \quad \text{where} \quad \frac{d}{dt} \varepsilon = \frac{\varepsilon_t - \varepsilon_{t-1}}{\Delta t}$$

We now have two factors: the proportional gain τ_p and the differential gain τ_d . As we get closer to our target trajectory, the rate of change in our error (that is, $\frac{d}{dt} \varepsilon$) decreases, causing a smoother approach:



4.2.3 I is for Integration

Unfortunately, the world is imperfect, and if you've ever let your steering wheel go on the highway you know how your car can veer off to one side ever-so-slightly. In our scenario, this means our crosstrack error ε is offset by some **systematic bias** introduced by mechanical errors, and our convergence to the target trajectory will forever include that bias:



How can we take this bias into account? As humans, we notice drift over time and compensate accordingly. Similarly here, we can accumulate the crosscheck error over time and incorporate that into our steering angle:

$$\alpha = -\tau_p \varepsilon - \tau_d \cdot \frac{d}{dt} \varepsilon - \tau_i \sum_t \varepsilon_t \quad (4.1)$$

Imagine we've "converged" α without accounting for the bias. Now, the last term dominates, growing larger and larger because of the accumulated bias and correcting the steering angle accordingly to reduce it!

This brings about our final **PID controller**, incorporating the **p**roportional, **i**ntegral, and **d**ifferential terms:

$$\alpha = \underbrace{-\tau_p \varepsilon}_{\text{proportional}} \underbrace{-\tau_d \cdot \frac{d}{dt} \varepsilon}_{\text{differential}} \underbrace{-\tau_i \sum_t \varepsilon_t}_{\text{integral}}$$

Integral Windup

Over time, our reference trajectory will change. With each change, we will have a new, large error (since we were *just* converging on a different reference). This means that our unchecked integral term will grow *very* large:

$$\alpha = -\tau_p \varepsilon - \tau_d \cdot \frac{d}{dt} \varepsilon - \underbrace{\tau_i \sum_t \varepsilon_t}_{\text{biiiiiig}}$$

Eventually, the α recommended by the PID controller is physically meaningless to the system. If we're manipulating the velocity of a car, for example, the controller might

want us to accelerate to $\pm 120\text{mph}$ to optimally achieve a new goal velocity. However, this is incredibly unsafe and physically impossible! Furthermore, our accumulated error causes responses to reference changes to lag behind (see [Figure 4.1](#), because the τ_i term is still adjusting towards the previous reference (and all reference trajectories before it!).

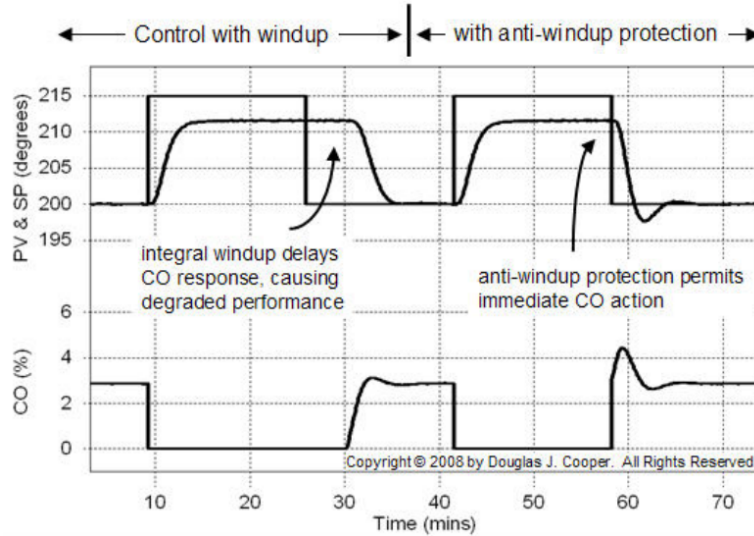


Figure 4.1: A temperature controller’s response without (left) and with (right) protection against integral windup. Notice how the system on the left lags in response to the change in the reference term. The *CO* term here (controller output) is equivalent to α in our discussions (the image comes from [here](#)).

When the integral term grows too large to have a meaningful impact on the system, we say the system suffers from **integral windup**. The easiest way to alleviate this is simply to stop integrating when the controller output, α , reaches the minimal or maximal value in the system.

4.2.4 Parameter Search: TWIDDLE

We have a PID controller equation now ([4.1](#)), but now we have three “magic numbers” that represent the gain of each component: τ_p, τ_d, τ_i . How can we find the best values for these?

The answer is basically to let the computer find them for you using the **twiddle** algorithm, sometimes also called **coordinate ascent**. In essence, we “twiddle” each value by a small amount with the aim of maximizing the overall “goodness” of the parameter set. This idea of “goodness” comes from an input function that essentially measures how well the parameters perform. For our purposes, we might choose a function that returns the average **crosstrack error** over a period of simulation time.

A generic twiddling algorithm catered is described in [algorithm 4.2](#); it should be trivial to cater towards our three τ s. Armed with such an algorithm, we can find the

best set of τ s for (4.1) in our specific system, and use that to follow a smooth path (that we could have planned using A^*) that we created earlier in [section 4.1](#)!

ALGORITHM 4.2: The Twiddle algorithm for finding an arbitrary vector of parameters.

Input: A function measuring how good a parameter set is, $G(\mathbf{p})$.

Result: The ideal set of (independent!) parameters, \mathbf{p}^* .

```

/* A vector of proposed parameters and some potential changes to
   each; obviously these vectors are the same size as the number of
   expected arguments to  $G$ . */
 $\mathbf{p} := [0 \ 0 \ \dots]$ 
 $\Delta\mathbf{p} := [1 \ 1 \ \dots]$ 
 $\varepsilon^* := G(\mathbf{p})$ 

/* While we still have meaningful adjustments... */
while  $\sum_i (\Delta p)_i \geq 0.00001$  do
    /* Adjust each parameter independently. */
    foreach  $p_i \in \mathbf{p}$  and  $d_i \in \Delta\mathbf{p}$  do
         $p_i += d_i$ 
         $\varepsilon := G(\mathbf{p})$ 
        if  $\varepsilon < \varepsilon^*$  then           // if it's better, twiddle larger next time
             $\varepsilon^* := \varepsilon$ 
             $d_i := 1.1d_i$ 
        else                           // try twiddling the other way
             $p_i -= 2d_i$                // 2x to undo the previous adjustment, too
             $\varepsilon := G(\mathbf{p})$ 
            if  $\varepsilon < \varepsilon^*$  then // if it's better, twiddle larger next time
                 $\varepsilon^* := \varepsilon$ 
                 $d_i := 1.1d_i$ 
            else                       // bad twiddle, so restore and twiddle less
                 $p_i += d_i$ 
                 $d_i := 0.9d_i$ 
            end
        end
    end
end
end
return  $\mathbf{p}$ 

```

SIMULTANEOUS LOCALIZATION AND MAPPING

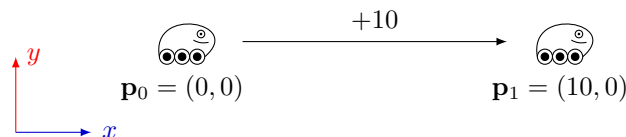
Come on and slam, and welcome to the jam!
Come on and slam, if you wanna jam!

— Space Jam, *Quad City DJ's*

UNTIL now, we have been tackling the automated navigation problem one step at a time. We discussed how to sense the world around us, how to plan actions based on those senses, and how to execute that plan. An important caveat of our sensing algorithm was that we needed a model of the world—to find out which door we were in front of when we did [Localization](#), we needed to know which locations *had* doors! In this final chapter, we will synthesize everything that we’ve learned into the SLAM algorithm, which will allow us to *simultaneously* localize ourselves within the world *and* map out the world itself. We will be learning specifically about the [Graph SLAM](#) method. There are many methods to SLAM, but this one is by far the easiest to explain and digest.

The key principle to keep in mind is that a robot that is building a map of an unknown area may lose track of where it is by the natural imperfections of its movement and the overall uncertainty in its motion.

Let’s craft what will be our running example. Suppose our robot lives in the xy coordinate plane and wishes to move from the origin to the right 10 units; for simplicity, let’s assume we don’t want any movement in the y direction:



In a perfect world, our position \mathbf{p}_1 after movement is $(10, 0)$ as above, but we have

learned that because of the imperfections of the real world, we need to incorporate a model of uncertainty into our position. Specifically, we have been modeling our posterior position as a Gaussian centered around our ideal, expected destination: $\mathbf{p}_1 = \mathbf{p}_0 + [\mathcal{N}(10, \sigma_x) \quad \mathcal{N}(0, \sigma_y)]$.

Let's break this down really quickly, though this should generally be a review of how a [Kalman filter](#) models motion. We want to minimize the error in our change in x :

$$\varepsilon_x = (x_1 - (x_0 + 10))^2$$

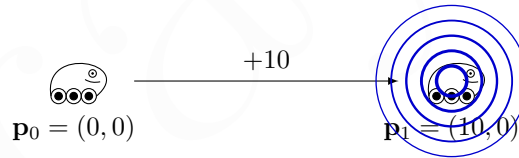
where x_1 is our “true” position, and $x_0 + 10$ is what we expected our position to be in a perfect world. This becomes the mean of our Gaussian distribution:¹²

$$\mathcal{N}(\varepsilon_x, \sigma) \sim \exp\left(\frac{(x_1 - x_0 - 10)^2}{-2\sigma^2}\right) \quad (5.1)$$

We want a similar thing for y , except since we want no change in y , we want our error (and thus Gaussian mean) to be as close to 0 as possible: $\varepsilon_y = (y_1 - y_0)^2$. This gives us:

$$\mathcal{N}(\varepsilon_y, \sigma) \sim \exp\left(\frac{(y_1 - y_0)^2}{-2\sigma^2}\right)$$

We've essentially modeled a bunch of circles around $(10, 0)$ that represent the possible locations for our robot after movement (with circles closer to $(10, 0)$ being exponentially likelier):



5.1 Constraints

The product of these x and y probabilities is now our constraint: we want to maximize the likelihood of our position \mathbf{p}_1 given our starting position \mathbf{p}_0 . What Graph SLAM does is define our probability using a sequence of these products. We model an **initial location constraint**, often just $\mathbf{p}_0 = \mathbf{0}$ for simplicity, a series of **relative motion constraints** (drawn below in **red**), and a series of **relative measurement constraints** (drawn below in **green**) to some common observable landmark. These relative constraints tie together each of our expected positions like a series of rubber bands, as shown in [Figure 5.1](#)

¹ Remember, σ is our variance, which measures how confident we are in our hardware; it affects how far or close we expect x_1 to be to our expected truth. For a very nice, expensive motor, σ will be low.

² We use \sim instead of $=$ here for brevity, avoiding the normalization coefficient, $\frac{1}{\sqrt{2\pi\sigma}}$, in front of the exponential.

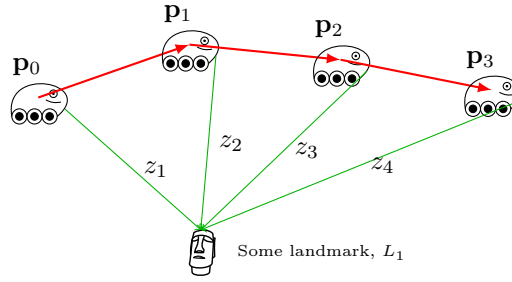


Figure 5.1: A visualization of Graph SLAM: the **red** motion constraints and **green** measurement constraints form a graph whose edges are the motion and measurement values.

The “solution” to our positions is would be the one that results in the smallest error among *all* of these constraints. That sounds like a job for... a system of equations!

Suppose we have three positions (we stick to one dimension for simplicity), (x_0, x_1, x_2) , and two universal landmarks L_1, L_2 observed at each of these positions. We can craft a matrix that describes their relationship.

5.1.1 Motion Constraints

We start with our first movement, $x_0 \rightarrow x_1$. Suppose we told our robot to move 5 units (in all dimensions, for simplicity), so $x_1 = x_0 + 5$ in a perfect world, or $x_1 - x_0 = 5$. We likewise know that $x_0 - x_1 = -5$. These become the first entries in the matrix, whose rows and columns just correspond to all of our possible values:

$$\begin{array}{c}
 x_0 \\
 x_1 \\
 x_2 \\
 L_1 \\
 L_2
 \end{array}
 \begin{array}{c|ccccc|c}
 & x_1 & x_2 & x_3 & L_1 & L_2 & \\
 \hline
 1 & -1 & \cdot & \cdot & \cdot & -5 \\
 -1 & 1 & \cdot & \cdot & \cdot & 5 \\
 \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
 \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
 \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
 \end{array}
 \begin{array}{c}
 x_0 \\
 x_1 \\
 x_2 \\
 L_1 \\
 L_2
 \end{array}$$

Suppose now that the motion between $x_1 \rightarrow x_2 = -4$. How would this affect our matrix? Naturally, it will affect all cells relating x_1 and x_2 . We know that:

$$x_1 - x_2 = 4 \quad (5.2)$$

$$x_2 - x_1 = -4 \quad (5.3)$$

All we do is add this new information into our above matrix!

$$\begin{bmatrix}
 1 & -1 & \cdot & \cdot & \cdot \\
 -1 & 2 & -1 & \cdot & \cdot \\
 \cdot & -1 & 1 & \cdot & \cdot \\
 \cdot & \cdot & \cdot & \cdot & \cdot \\
 \cdot & \cdot & \cdot & \cdot & \cdot
 \end{bmatrix}
 \begin{bmatrix}
 x_0 \\
 x_1 \\
 x_2 \\
 L_1 \\
 L_2
 \end{bmatrix}
 =
 \begin{bmatrix}
 -5 \\
 9 \\
 -4 \\
 \cdot \\
 \cdot
 \end{bmatrix}$$

We can repeat this process for any number of motion constraints. How do we our measurement constraints?

QUICK MAFFS: Sanity Check

The second row of our matrix now represents the following equation:

$$-x_0 + 2x_1 - x_2 = 9 \quad (5.4)$$

We haven't directly related all three of these terms yet; is this relation even valid? Let's make sure. We know:

$$\begin{cases} x_1 - x_0 = 5 \\ x_1 - x_2 = 4 \end{cases}$$

Adding these together gives us (5.4), so it looks like the math checks out!

What's interesting is that we can do a simple rearrangement of the above to isolate and remove x_1 and get a new relationship:

$$\begin{aligned} 4 + x_2 &= 5 + x_0 \\ x_2 - x_0 &= 1 \end{aligned}$$

We could theoretically introduce this additional row constraint into our matrix:

$$\begin{bmatrix} -1 & \cdot & 1 & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix}$$

Neat!

5.1.2 Landmark Constraints

Suppose we measure that the distance from x_1 to some landmark L_0 is 9 units. Just as before, this creates a new pair of constraints in the matrix, but this time from a measurement:

$$\begin{aligned} L_0 - x_1 &= 9 \\ x_1 - L_0 &= -9 \end{aligned}$$

And again, as before, we just add these into our constraint matrix:

$$\begin{bmatrix} 1 & -1 & \cdot & \cdot & \cdot \\ -1 & 3 & -1 & -1 & \cdot \\ \cdot & -1 & 1 & \cdot & \cdot \\ \cdot & -1 & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ L_1 \\ L_2 \end{bmatrix} = \begin{bmatrix} -5 \\ 0 \\ -4 \\ 9 \\ \cdot \end{bmatrix}$$

Just like we did for the motion constraints, we can repeat this process for all of the landmark measurements we make at each point. The more observations we have, the more constrained our matrix is, and the more certain we can be of all of our positions.

5.2 Generalization

The method we've described above generalized to a linear system: $\boldsymbol{\xi} = \boldsymbol{\Omega}\boldsymbol{\mu}$ (thank u Greece, very cool), where $\boldsymbol{\Omega}$ describes our constraint scalars, $\boldsymbol{\mu}$ describes our (unknown) robot positions and landmark observations, and $\boldsymbol{\xi}$ contains the result constants (the column vector in our running example above).

We can solve this exactly with a bit of linear algebra—inverting the matrix $\boldsymbol{\Omega}$ via something like [numpy.linalg.inv](https://numpy.org/doc/stable/reference/generated/numpy.linalg.pinv.html)—to find the best positions $\boldsymbol{\mu} = [x_0 \ x_1 \ \dots]$ that fulfill our constraints:

$$\boldsymbol{\mu} = \boldsymbol{\Omega}^{-1}\boldsymbol{\xi}$$

EXAMPLE 5.1: Graph SLAM

Let's run through an example to solidify our understanding. Suppose our robot undergoes the following (ideal) motions:

$$\begin{cases} x_0 = -3 \\ x_1 = x_0 + 5 \\ x_2 = x_1 + 3 \end{cases}$$

That is, it starts at $x = -3$, moves 5 units to the right, then 3 more units to the right. For simplicity, suppose we have no observations. Filling out our constraint matrix for the initial location constraint gives us:

$$\begin{bmatrix} 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -3 \\ \cdot \\ \cdot \end{bmatrix}$$

Introducing the first motion constraint adds up to:

$$\begin{bmatrix} 2 & -1 & \cdot \\ -1 & 1 & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -8 \\ 5 \\ \cdot \end{bmatrix}$$

Then, introducing our final motion constraint gives us:

$$\underbrace{\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}}_{\Omega} \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}}_{\mu} = \underbrace{\begin{bmatrix} -8 \\ 2 \\ 3 \end{bmatrix}}_{\xi}$$

Finding the [inverse](#), Ω^{-1} , gives us:

$$\Omega^{-1} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix}$$

Finally, solving $\mu = \Omega^{-1}\xi$ gives us the exact result we expected:

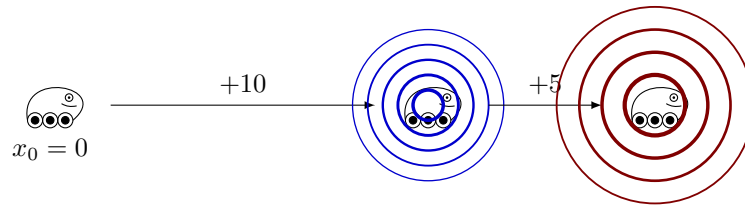
$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} -8 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -8 + 2 + 3 = -3 \\ -8 + 2 \cdot 2 + 2 \cdot 3 = 2 \\ -8 + 2 \cdot 2 + 3 \cdot 3 = 5 \end{bmatrix}$$

5.2.1 Introducing Noise

Now in [example](#) and our matrix constructions above, we had perfect measurements. But in our initial construction of the problem, we recognized that there was actually some Gaussian noise perturbing our measurements!

Mathematically, the introduction of noise means that a “perfect” solution to $\Omega\mu = \xi$ no longer exists. However, the solution vector μ that we find is *still the best*! It’s the solution that minimizes the overall error in our system, meaning our “rubber bands” (see [Figure 5.1](#) again to refresh the analogy) had to stretch or shrink the least overall amount.

Why is this true? We’ll have to refer back to the noise model of (5.1) earlier. When we moved our robot a noisy 10 units, the uncertainty of our position was modeled by a Gaussian. If we then told it to move *another* 5 units, we would be a little *more* uncertain, right?



Mathematically, we model this by multiplying our Gaussians:

$$x_2 \sim (x_0 + \mathcal{N}(10, \sigma)) \cdot \mathcal{N}(5, \sigma)$$

If we expand this expression out (refer to (5.1)) and squint hard enough, we can see the rationale for our “add each constraint” approach. Note that this isn’t meant to be mathematically rigorous, but rather demonstrate a bit of intuition behind why μ is still the best possible positions given our constraints. To find the value that maximizes the total probability of our measurements, we isolate our total errors:

$$\begin{aligned}
 x_2 &\sim \exp\left(\frac{(x_1 - x_0 - 10)^2}{-2\sigma^2}\right) \cdot \exp\left(\frac{(x_2 - x_1 - 5)^2}{-2\sigma^2}\right) \\
 &\sim \frac{(x_1 - x_0 - 10)^2}{-2\sigma^2} + \frac{(x_2 - x_1 - 5)^2}{-2\sigma^2} \\
 &\sim \underbrace{\frac{(x_1 - x_0 - 10)^2}{\sigma^2}}_{1^{\text{st}} \text{ motion constraint}} + \underbrace{\frac{(x_2 - x_1 - 5)^2}{\sigma^2}}_{2^{\text{nd}} \text{ motion constraint}}
 \end{aligned}$$

Recall: $e^x \cdot e^y = e^{x+y}$
so we can drop the exp and
maintain proportionality

drop the constants since they have
no bearing on the probability
maximization

Notice how we broke things down to essentially be a sum of constraints, and how they’re both scaled by $1/\sigma^2$, which encodes our “certainty” in each measurement! If we had different certainties at each point in time (perhaps we measure our position with sonar at one point, and do a depth estimation with stereo cameras at another point), we can incorporate this into the probability distribution of x_2 as well by using a separate σ_1 and σ_2 in each term.

We can essentially scale each constraint depending on our certainty in the accuracy of said constraint (for both position and landmark measurements)! To loop back to the matrix of constraints, this means that when we add in a new constraint we can actually scale its impact by $\frac{1}{\sigma}$ depending on our confidence. Remember, a smaller σ indicates higher confidence.

For example, suppose we were *very* confident of our position after our first move in our earlier example, with $\sigma = 0.2$. When we introduced our first motion constraint in (5.2), then, we can scale it up by, say $1/0.2 = 5$:

$$\begin{aligned}
 5x_1 - 5x_2 &= 20 \\
 5x_2 - 5x_1 &= -20
 \end{aligned}$$

This means that constraint would be more impactful on the overall “equalization” of the rubber bands of our various constraints, preserving this edge in the graph more truthfully!

5.3 Summary

Let’s reflect on the beauty of what we’ve derived here. Given a series of measurements about the world (which essentially *map* it out) and a series of approximations about

our current position in the world (which *localizes* us within the map), we can solve for these values *simultaneously* because they all impose constraints on one another.

This is a pretty magical thing, and actually was not that complicated to create! Armed with this algorithmic knowledge, a robot would be able to map an unknown environment while also understanding where it is in that environment.

INDEX OF TERMS

A

*A** 13, 23

B

belief 6

C

conditional probability 7

coordinate ascent 22

cost function 9, 11

crosstrack error 18, 22

G

gradient descent 18

Graph SLAM 25

H

heuristic 13

I

integral windup 22

K

Kalman filter 26

Kalman filters 8

L

localization 5

M

Manhattan distance 13

marginally stable 20

measurement update 7

N

normalize 7

P

particle filters 8

PID control 17

PID controller 18, 21

posterior 6

prior 6

probability distribution 5

S

systematic bias 21

T

twiddle 22

U

uniform distribution 5