Mapping and Navigation

Principles and Shortcuts

January 6th, 2011

Slides from Edwin Olson’s 2008 presentation
Presented by Eric Timmons etimmons@mit.edu
Goals for this talk

- Why should I build a map?

- Three mapping algorithms
  - Forgetful local map
    - Really easy, very useful over short time scales (seconds to a minute)
  - Topological roadmap
    - Also really easy, moderately useful over arbitrary time scales
  - World’s simplest—but powerful—SLAM algorithm
    - A taste of the “real thing”.
Attack Plan

- *Motivation and Advice*
- Algorithms:
  - Forgetful Map
  - Topological Map
  - SLAM
- Sensor Comments
Why build a map?

• Playing field is big, robot is slow

• Driving around perimeter takes a minute!

• Scoring takes time… often ~20 seconds to “line up” to a mouse hole.
Maslab Mapping Goals

- Be able to efficiently move to specific locations that we have previously seen
  - I’ve got a bunch of balls, where’s the nearest goal?

- Be able to efficiently explore unseen areas
  - Don’t re-explore the same areas over and over

- Build a map for its own sake
  - No better way to wow your competition/friends/audience.
A little advice

- Mapping is hard! And it’s not required to do okay.

- Concentrate on basic robot competencies first
- Design your algorithms so that map information is helpful, but not required
- Pick your mapping algorithm judiciously
  - Pick something you’ll have time to implement and test
  - Lots of newbie gotchas, like 2pi wrap-around
Visualization

- Visualization is critical

  - *Impossible* to debug your code unless you can see what’s happening

  - Write code to view your maps and publish them!

  - Nobody will appreciate your map if they can’t see it.
Attack Plan

• Motivation and Advice
• Algorithms:
  – *Forgetful Map*
  – Topological Map
  – SLAM
• Sensor Comments
Forgetful Local Map

- It’s as good as your dead-reckoning

- Estimate your dead-reckoning error, don’t use data that’s useless.
  - Don’t throw it away though—log it.

- Easy to implement
Dead-Reckoning

Compute robot’s position in an arbitrary coordinate system

\[ x = \sum d_i \cdot \cos(\theta_i) \]
\[ y = \sum d_i \cdot \sin(\theta_i) \]
\[ \theta_i = \sum \Delta \theta_i \]

Easy to compute:

- Get \( d_i \) from wheel encoders (or back EMF-derived velocity?)
- Get \( \Delta \theta_i \) from gyro
  - Actually, integration done for you
The problem with dead-reckoning

- Error accumulates over time
  - Really fast—errors in $\theta_i$ cause *super-linear* increases in error
  - Use zero-velocity update

- Distance error proportional to measured distance
  - Anywhere from 10-50% depending on sensors

- Gyro error mostly a function of *time*.
  - About 1-5 degrees per minute.
World’s simplest (metrical) map

- Every time you see something, record it in a list

- Looking for something?
  - Search *backwards* in the list

- Don’t use old data
  - Estimate distance/theta error by subtracting cumulative error estimates
  - If theta error > 30 degrees or so → bearing is bad
  - If distance error > 30% of distance to object → bearing is bad
  - (These constants made up– you’ll need to experiment!)

<table>
<thead>
<tr>
<th>Cumulative Distance/Orientation error</th>
<th>What</th>
<th>Location (x,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.2, 0.1)</td>
<td>Goal</td>
<td>(2.3, 1.1)</td>
</tr>
<tr>
<td>(0.4, 0.15)</td>
<td>Robot Pose</td>
<td>(2.0, 1.0)</td>
</tr>
<tr>
<td>(1.0, 0.2)</td>
<td>Barcode</td>
<td>2.4, 1.2)</td>
</tr>
<tr>
<td>(2.0, 0.22)</td>
<td>Barcode</td>
<td>(3.5, .3)</td>
</tr>
<tr>
<td>(2.5, 0.3)</td>
<td>Robot Pose</td>
<td>(3.0, 1.0)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Zero-velocity updates

- Gyros accumulate error as a function of integration time
  - Even if you’re not moving

- Idea: if robot is stationary, stop gyro integration → stop error accumulation
Attack Plan

• Motivation and Advice
• Algorithms:
  – Forgetful Map
  – *Topological Map*
  – SLAM
• Sensor Comments
Topological Maps

- Learn and remember invariant properties in the world:
  - “I can see barcodes 3 and 7 when I’m sitting next to barcode 12”

- De-emphasize *metrical* data
  - Maybe remember “when I drove directly from barcode 2 to barcode 7, it was about 3.5 meters”

- Very easy!
  - But you can probably only put barcodes (maybe goals) into the map
Topological Maps

- Nodes in graph are easily identifiable features
  - E.g., barcodes

- Each node lists things “near” or visible to it
  - Other bar codes
  - Goals, maybe balls

- Implicitly encode obstacles
  - Walls obstruct visibility!

- Want to get somewhere?
  - Drive to the nearest barcode, then follow the graph.
Topological Maps - Challenges

- Building map takes time
  - Repeated 360 degree sensor sweeps

- Solutions sub-optimal
  - (But better than random walk!)

- You may have to resort to random walking when your graph is incomplete

- Hard to visualize since you can’t recover the actual positions of positions
Attack Plan

• Motivation and Advice
• Algorithms:
  – Forgetful Map
  – Topological Map
  – **SLAM**
• Sensor Comments
Brute-Force SLAM

- Simultaneous Localization and Mapping (SLAM)

- The following approach is exact, complete
  - (Is used in the “real world”)
  - I’ll show a version that works, but isn’t particularly scalable.

- Break out the 18.06!
  - Weren’t paying attention? Quick refresher coming…
Quick math review

- **Linear** approximation to arbitrary functions
  - \( f(x) = x^2 \)
    - near \( x = 3 \), \( f(x) \approx 9 + 6 \,(x-3) \)
    - \( f(3) + \frac{df}{dx} \cdot (x-3) \)

- \( f(x,y,z) = (some\ mess) \)
  - near \((x_0, y_0, z_0)\): \( f(x) \approx F_0 + \begin{bmatrix} \frac{df}{dx} & \frac{df}{dy} & \frac{df}{dz} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} \)
Quick math review

From previous slide:

\[ f(x) = f_0 + \left[ \begin{array}{ccc} \frac{df}{dx} & \frac{df}{dy} & \frac{df}{dz} \\ \Delta x & \Delta y & \Delta z \end{array} \right] \]

Re-arrange:

\[ \begin{bmatrix} \frac{df}{dx} & \frac{df}{dy} & \frac{df}{dz} \\ \Delta x & \Delta y & \Delta z \end{bmatrix} = f(x) - f_0 \]

Linear Algebra notation:

\[ \mathbf{J} \mathbf{d} = \mathbf{r} \]
Example

- We observe range $z_d$ and heading $z_{\theta}$ to a feature.
  - We express our observables in terms of the state variables ($x^* \ y^* \ \text{theta}^*$) and noise variables ($v^*$)

\[
h = \begin{cases} 
  z_d = [(x_f - x_r)^2 + (y_f - y_r)^2]^{1/2} + v_d \\
  z_{\theta} = \arctan 2(y_f - y_r \ , x_f - x_r) - x_{\theta} + v_{\theta}
\end{cases}
\]
Example

Compute a linear approximation of these constraints:

- Differentiate these constraints with respect to the state variables
- End up with something of the form $Jd = r$
Example

\[ h = \begin{pmatrix} 
  z_d = [(x_f - x_r)^2 + (y_f - y_r)^2]^{1/2} + v_d \\
  z_\theta = \arctan 2(y_f - y_r, x_f - x_r) - \theta_r + v_\theta 
\end{pmatrix} \]

A convenient substitution: \( \lambda = 1/(1 + (d_y / d_x))^2 \)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>( x_r )</th>
<th>( y_r )</th>
<th>( \theta_r )</th>
<th>( x_f )</th>
<th>( y_f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z_d )</td>
<td>(- d_x / d )</td>
<td>(- d_y / d )</td>
<td>0</td>
<td>( d_x / d )</td>
<td>( d_y / d )</td>
</tr>
<tr>
<td>( z_\theta )</td>
<td>( \lambda d_y / d_x^2 )</td>
<td>(- \lambda / d_x )</td>
<td>-1</td>
<td>(- \lambda d_y / d_x^2 )</td>
<td>( \lambda / d_x )</td>
</tr>
</tbody>
</table>

\[ H = \text{Jacobian of } h \text{ with respect to } x \]
Metrical Map example

By convention, this pose is (0,0,0)

Unknown variables (x,y,theta) per pose

Constraints (arising from odometry)

number unknowns == number of equations, solution is critically determined.

\[ d = J^{-1}r \]
The feature gives us more unknowns
Observations give us more equations

number unknowns < number of equations, solution is over determined.
Least-squares solution is:

\[ d = (J^TJ)^{-1}J^Tr \]

More equations = better pose estimate
Computational Cost

- The least-squares solution to the mapping problem:

\[ d = (J^T W J)^{-1} J^T W b \]
\[ x_{i+1} = x_i + d \]

- Must invert* a matrix of size 3Nx3N (N = number of poses.) Inverting this matrix costs O(N^3)!
  - N is pretty small for maslab
  - How big can N get before this is a problem?

- JAMA, Java Matrix library

* We’d never actually invert it; it’s better to use a Cholesky Decomposition or something similar. But it has the same computational complexity. JAMA will do the right thing.
State of the Art

- Simple! Just solve

\[ d = (J^T W J)^{-1} J^T W b \]

faster, using less memory.

(many a PhD Thesis. Hopefully good for at least one more)
Metrical Map - Weighting

- Some sensors (and constraints) better than others
- Put *weights* in block-diagonal matrix $W$

$$W = \text{weight of eqn 1}$$
$$W = \text{weight of eqn 2}$$

$$d = (J^T W J)^{-1} J^T Wr$$

- What is the interpretation of $J^T W J$?
What does all this math get us?

- Okay, so why bother?
Odometry Trajectory

- Integrating odometry data yields a trajectory.

- Uncertainty of pose increases at every step.
Metrical Map example

1. Original Trajectory with odometry constraints

2. Observe external feature
   Initial feature uncertainty = pose uncertainty + observation uncertainty

3. Reobserving feature helps subsequent pose estimates
Attack Plan

• Motivation and Advice
• Algorithms:
  – Forgetful Map
  – Topological Map
  – SLAM
• Sensor Comments
Getting Data - Odometry

- Roboticists bread-and-butter
  - You should use odometry in some form, if only to detect if your robot is moving as intended

- “Dead-reckoning” : estimate motion by counting wheel rotations
  - Encoders (binary or quadrature phase)
  - Maslab-style encoders are very poor

- Motor modeling
  - Model the motors, measure voltage and current across them to infer the motor angular velocity
  - Angular velocity can be used for dead-reckoning
  - Pretty lousy method, but possibly better than low-resolution flaky encoders
Getting Data - Camera

- Useful features can be extracted!
  - Lines from white/blue boundaries
  - Balls (great point features! Just delete them after you’ve moved them.)
  - “Accidental features”

- You can estimate bearing \textit{and} distance.
  - Camera mounting angle has effect on distance precision

- Triangulation
  - Make bearing measurement
  - Move robot a bit (keeping odometry error small)
  - Make another bearing measurement

More features = better navigation performance
Range finders

- Range finders are most direct way of locating walls/obstacles.

- Build a “LIDAR” by putting a range finder on a servo
  - High quality data! Great for mapping!
  - Terribly slow.
    - At least a second per scan.
      - With range of > 1 meter, you don’t have to scan very often.
    - Two range-finders = twice as fast
      - Or alternatively, 360° coverage
    - Hack servo to read analog pot directly
      - Then slew the servo in one command at maximum speed instead of stepping.
    - Add gearbox to get 360° coverage with only one range finder.
Parting Words

- Many issues we didn’t cover
  - Data Association

- Good reference:
Questions?
Extended Kalman Filter

- $x$: vector of all the state you care about (same as before)
- $P$: covariance matrix (same as $(J^TWJ)^{-1}$ before)

- **Time update:**
  - $x' = f(x,u,0)$  \(\leftarrow\) integrate odometry
  - $P = APA^T + BQB^T$  \(\leftarrow\) adding noise to covariance
    - $A = $ Jacobian of $f$ wrt $x$
    - $B = $ Jacobian of noise wrt $x$
    - $Q = $ covariance of odometry
Metrical Map - Weighting

- Some sensors (and constraints) better than others
- Put *weights* in block-diagonal matrix $W$

$$W = \begin{bmatrix}
\text{weight of eqn 1} & 0 \\
0 & \text{weight of eqn 2}
\end{bmatrix}$$

$$d = (J^T W J)^{-1} J^T W r$$

- What is the interpretation of $J^T W J$?
Correlation/Covariance

- In multidimensional Gaussian problems, equal-probability contours are ellipsoids.

- Shoe size doesn’t affect grades: 
  \[ P(\text{grade,shoesize}) = P(\text{grade})P(\text{shoesize}) \]

- Studying helps grades: 
  \[ P(\text{grade,studytime}) \neq P(\text{grade})P(\text{studytime}) \]
  - We must consider \( P(x,y) \) jointly, respecting the correlation!
  - If I tell you the grade, you learn something about study time.
Why is covariance useful?

- Loop Closing (and Data Association)
- Suppose you observe a goal (with some uncertainty)
  - Which previously-known goal is it?
  - Or is it a new one?
- Covariance information helps you decide
  - If you can tell the difference between goals, you can use them as navigational landmarks!
Extended Kalman Filter

- **Observation**
  - $K = PH^T(HPH^T + VRV^T)^{-1}$ ← Kalman “gain”
  - $x' = x + K(z - h(x,0))$
  - $P = (I-KH)P$

- $P$ is your covariance matrix
  - Just like $(J^TWJ)^{-1}$

$H$ = Jacobian of *constraint* wrt $x$
$B$ = Jacobian of noise wrt $x$
$R$ = covariance of *constraint*
Kalman Filter: Properties

- You incorporate sensor observations one at a time.
- Each successive observation is the same amount of work (in terms of CPU).
- *Yet, the final estimate is the global optimal solution.*
  - The same solution we would have gotten using least-squares. Almost.

The Kalman Filter is an *optimal*, recursive estimator.
Kalman Filter: Properties

- In the limit, features become highly correlated
  - Because observing one feature gives information about other features

- Kalman filter computes the *posterior pose*, but **not** the posterior *trajectory*.
  - If you want to know the path that the robot traveled, you have to make an extra “backwards” pass.
Kalman Filter: Shortcomings

- With N features, update time is still large: $O(N^2)$!
- For Maslab, N is small. Who cares?
- In the “real world”, N can be $>>10^6$.
- Linearization Error
- Current research: lower-cost mapping methods
Old Slides
Kalman Filter

- Example: Estimating where Jill is standing:
  - Alice says: $x=2$
    - We think $\sigma^2 = 2$; she wears thick glasses
  - Bob says: $x=0$
    - We think $\sigma^2 = 1$; he’s pretty reliable

- How do we combine these measurements?
Simple Kalman Filter

• Answer: algebra (and a little calculus)!
  – Compute mean by finding maxima of the log probability of the product \( P_A P_B \).
  – Variance is messy; consider case when \( P_A = P_B = \text{N}(0,1) \)

• Try deriving these equations at home!

\[
\sigma^2 = \frac{1}{\sigma_A^2} + \frac{1}{\sigma_B^2}
\]

\[
\mu = \frac{\mu_A \sigma_B^2 + \mu_B \sigma_A^2}{\sigma_A^2 + \sigma_B^2}
\]
We now think Jill is at:

- \( x = 0.66 \)
- \( \sigma^2 = 0.66 \)

Note: Observations always reduce uncertainty
- Even in the face of conflicting information, EKF never becomes less certain.
Kalman Filter

- Now Jill steps forward one step

- We think one of Jill’s steps is about 1 meter, $\sigma^2 = 0.5$

- We estimate her position:
  - $X = X_{\text{before}} + X_{\text{change}}$
  - $\sigma^2 = \sigma_{\text{before}}^2 + \sigma_{\text{change}}^2$

- Uncertainty *increases*
Data Association

Data association: The problem of recognizing that an object you see now is the same one you saw before

- Hard for simple features (points, lines)
- Easy for “high-fidelity” features (barcodes, bunker hill monuments)

With perfect data association, most mapping problems become “easy”
Data Association

If we can’t tell when we’re reobserving a feature, we don’t learn anything!

- We need to observe the same feature twice to generate a constraint.
Data Association: Bar Codes

- Trivial!

- The Bar Codes have unique IDs; read the ID.
Data Association: Nearest Neighbor

- Nearest Neighbor
  - Simplest data association “algorithm”
  - Only tricky part is determining when you’re seeing a brand-new feature.
The blue tick marks can be used as features too.
- Probably hard to tell that a particular tick mark is the one you saw 4 minutes ago…
- You only need to reobserve the same feature *twice* to benefit!
- If you can track them over short intervals, you can use them to improve your dead-reckoning.
  - Use nearest-neighbor. Your frame-to-frame uncertainty should only be a few pixels.
Data Association: Tick Marks

- Ideal situation:
  - Lots of tick marks, randomly arranged
  - Good position estimates on all tick marks

- Then we search for a *rigid-body-transformation* that best aligns the points.
Data Association: Tick Marks

- Find a rotation that aligns the most tick marks…
  - Gives you data association for matched ticks
  - Gives you rigid body transform for the robot!

Rotation+Translation
Metrical Map: Cost Function

- Cost function *could* be arbitrarily complicated
  - Optimization of these is intractable
- We can make a local approximation around *the current pose estimates*
  - Resembles the arbitrary cost function in that neighborhood
  - Typically Gaussian
Metrical Map: Real World Cost Function

Cost function arising from aligning two LADAR scans
Consider each pose/feature:
- Fix all others features/poses
- Solve for the position of the unknown pose

Repeat many times
- Will converge to minimum
- Works well on small maps
Nonlinear Map Optimization

LogP/|C|: -1.443730e+08
time: 0.071
Occupancy Grids

- Divide the world into a grid
  - Each grid records whether there’s something there or not
    - Usually as a probability
  - Use current robot position estimate to fill in squares according to sensor observations
Occupancy Grids

• Easy to generate, hard to maintain accuracy
  – Basically impossible to “undo” mistakes

• Convenient for high-quality path planning

• Relatively easy to tell how well you’re doing
  – Do your sensor observations agree with your map?
FastSLAM (Gridmap variant)

- Suppose you maintain a whole bunch of occupancy maps
  - Each assuming a slightly different robot trajectory

- When a map becomes inconsistent, throw it away.

- If you have enough occupancy maps, you’ll get a good map at the end.
Gridmap, a la MASLab

- Number of maps you need increases \textit{exponentially} with distance travelled. (Rate constant related to odometry error)

- Build grid maps until odometry error becomes too large, then start a new map.

- Try to find old maps which contain data about your current position
  - Relocalization is usually hard, but you have unambiguous features to help.
Occupancy Grid: Path planning

- Use A* search
  - Finds optimal path (subject to grid resolution)
  - Large search space, but optimum answer is easy to find

- `search(start, end)`
  - Initialize `paths` = set of all paths leading out of cell “start”
  - Loop:
    - let `p` be the best path in `paths`
      - Metric = distance of the path + straight-line distance from last cell in path to goal
    - if `p` reaches `end`, return `p`
    - Extend path `p` in all possible directions, adding those paths to `paths`
Occupancy Grid: Path planning

- How do we do path planning with EKFs?
- Easiest way is to rasterize an occupancy grid on demand
  - Either all walls/obstacles must be features themselves, or
  - Remember a local occupancy grid of where walls were at each pose.
Attack Plan

• Motivation and Terminology
• Mapping Methods
  – Topological
  – Metrical
• Data Association
• Sensor Ideas and Tips
Finding a rigid-body transformation

- Method 1 (silly)
  - Search over all possible rigid-body transformations until you find one that works
    - Compare transformations using some “goodness” metric.

- Method 2 (smarter)
  - Pick two tick marks in both scene A and scene B
  - Compute the implied rigid body transformation, compute some “goodness” metric.
  - Repeat.
    - If there are N tick marks, M of which are in both scenes, how many trials do you need? Minimum: \((M/N)^2\)
  - This method is called “RANSAC”, RANdom SAmple Consenus
Attack Plan

• Motivation and Terminology
• Mapping Methods
  – Topological
  – Metrical
• Data Association
• Sensor Ideas and Tips
Debugging map-building algorithms

- You can’t debug what you can’t see.

- Produce a visualization of the map!
  - Metrical map: easy to draw
  - Topological map: draw the graph (using graphviz/dot?)
  - Display the graph via BotClient

- Write movement/sensor observations to a file to test mapping independently (and off-line)
Today’s Lab Activities
Bayesian Estimation

• Represent unknowns with probability densities
  – Often, we assume the densities are Gaussian
    \[ P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \]
  – Or we represent arbitrary densities with particles
    • We won’t cover this today
Some constraints are better than others.

Incorporate constraint “weights”

- Weights are closely related to covariance:
  \[ W = \Sigma^{-1} \]
- Covariance of poses is:
  \[ A^TWA \]

In principle, equations might not represent independent constraints. But usually they are, so these terms are zero.

\[ x = (A^TWA)^{-1}A^TWb \]

* Of course, “covariance” only makes good sense if we make a Gaussian assumption.
Map representations

Occupancy Grid

Pose/Feature Graph
Graph representations

- **Occupancy Grids:**
  - Useful when you have dense range information (LIDAR)
  - Hard to undo mistakes

- I don’t recommend this…
Graph representations

- Pose/Feature graphs
  - **Metrical**
    - Edges contain relative position information
  - **Topological**
    - Edges imply “connectivity”
    - Sometimes contain “costs” too (maybe even distance)

- If you store ranging measurements at each pose, you can generate an occupancy grid *from* a pose graph
Metrical Maps

Advantages:
- Optimal paths
- Easier to visualize
- Possible to distinguish different goals, use them as navigational features
- Way cooler

Disadvantages:
- There’s going to be some math.
  *gasp* Partial derivatives!
State Correlation/Covariance

- We observe features relative to the robot’s current position
  - Therefore, feature location estimates *covary* (or correlate) with robot pose.

- Why do we care?
  - We get the wrong answer if we don’t consider correlations
  - Covariance is useful!
Once we’ve solved for the position of each pose, we can re-project the observations of obstacles made at each pose into a coherent map

That’s why we kept track of the old poses, and why $N$ grows!
What if we only want to estimate:
- Positions of each goal
- Positions of each barcode
- Current position of the robot?

The Kalman filter is our best choice now.
- Almost the same math!
- Not enough time to go into it: but slides are on wiki