6.01: Introduction to EECS I

Optimal Search Algorithms

Week 13 *November 30, 2010*

Reading: 9.5 – 9.6

The story so far

- **Search domain** — characterized by successors function, legal actions, start state, goal function.
- **Search tree** — an explicit representation for the search space.
- **Depth-first search** — explore search tree by expanding deepest node. stack
- **Breadth-first search** — explore search tree by expanding shallowest node. queue
- **Dynamic programming** — do not revisit nodes.

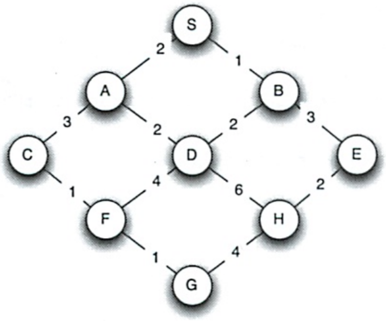
Reasoning w/ uncertanity
Move plant inside controller
-robot trys at paths

Pruning - don't go back up path came down
-no ∞ loops

Adding

Cost

In many applications, actions have different costs, for example, distance between cities can vary.



Our algorithms thus far ignore this.

Want to find cheapest path (smallest cost)

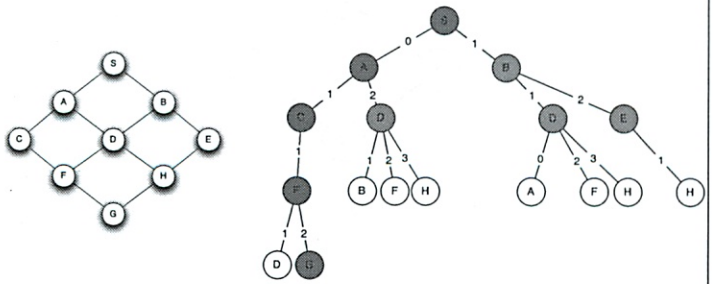
Cost

```
map1dist = {'S' : [('A', 2), ('B', 1)],
            'A' : [('S', 2), ('C', 3), ('D', 2)],
            'B' : [('S', 1), ('D', 2), ('E', 3)],
            'C' : [('A', 3), ('F', 1)],
            'D' : [('A', 2), ('B', 2), ('F', 4), ('H', 6)],
            'E' : [('B', 3), ('H', 2)],
            'F' : [('C', 1), ('D', 4), ('G', 1)],
            'H' : [('D', 6), ('E', 2), ('G', 4)],
            'G' : [('F', 1), ('H', 4)]}
```

Path cost is the sum of the action costs along a path.

Breadth-First Search

Enumerates all 1-hop paths, then 2-hop paths, then 3-hop paths, etc.



Uniform-Cost Search

Enumerate paths in order of their total **path cost**.

Like breadth-first search, but:

- The agenda is a priority queue (returns least cost entry).
- Instead of testing for a goal state when we put an element *into* the agenda, we test for a goal state when we take an element *out of* the agenda.

Guaranteed to find a shortest path.

Priority Queue

A priority queue is a data structure with the same basic operations as stacks and queues, with two differences:

- Items are pushed into a priority queue with a numeric score, called a *cost*.
- When it is time to pop an item, the item in the priority queue with the least *cost* is returned and removed from the priority queue.

(oh so don't have to visit every one ?)

Can sort the items by cost when you insert them - then just pop 1st

can do at time of pop as well

or something else more complicated

Priority Queue

```
>>> pq = PQ()
>>> pq.push('a', 3)
>>> pq.push('b', 6)
>>> pq.push('c', 1)
>>> pq.pop()
'c'
>>> p1.pop()
'a'
```

Priority Queue

Simple implementation using lists

```
class PQ:
    def __init__(self):
        self.data = []
    def push(self, item, cost):
        self.data.append((cost, item))
    def pop(self):
        (index, cost) = util.argmaxIndex(self.data, lambda (c, x): -c)
        return self.data.pop(index)[1] # just return the data item
    def isEmpty(self):
        return self.data is []
```

here lowest-cost during pop

can't min, so just -cost

The pop operation in this implementation can take time proportional to the number of nodes (in the worst case).

Better algorithms (using trees) reduce run time to be proportional to the log of the number of nodes (in the worst case).

Search Node

```
class SearchNode:
    def __init__(self, action, state, parent, actionCost):
        self.state = state
        self.action = action
        self.parent = parent
        if self.parent:
            self.cost = self.parent.cost + actionCost
        else:
            self.cost = actionCost
```

keep track of cost

ucSearch

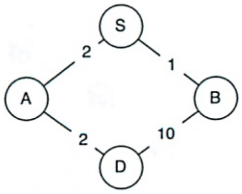
```
def ucSearch(initialState, goalTest, actions, successor):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    while not agenda.isEmpty():
        n = agenda.pop()
        if goalTest(n.state):
            return n.path()
        for a in actions:
            (newS, cost) = successor(n.state, a)
            if not n.inPath(newS):
                newN = SearchNode(a, newS, n, cost)
                agenda.push(newN, newN.cost)
    return None
```

← priority queue

← just test state

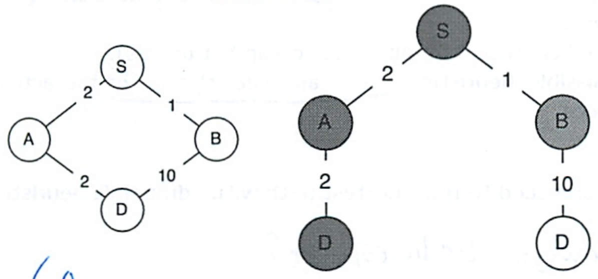
very similar to previous

Example



ucSearch: From S to D

Numbers on links are distances not action indices



S,0
↳ agenda (SA,2),(SB,1)
take

SB,1
↳ agenda (SA,2)(SBD,11) ← don't goal test until expand
Sum of cost

SA,2
↳ (SBD,11)(SBD,4) — take lowest item – check – is goal – win!

← Must be shortest path

ucSearch with DP

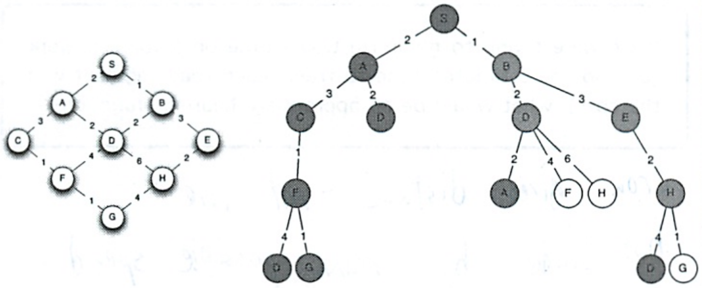
– Only need to remember shortest path from start state to any other state

```
def ucSearch(initialState, goalTest, actions, successor):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    expanded = { }
    while not agenda.isEmpty():
        n = agenda.pop()
        if not expanded.has_key(n.state):
            expanded[n.state] = True
            if goalTest(n.state):
                return n.path()
            for a in actions:
                (newS, cost) = successor(n.state, a)
                if not expanded.has_key(newS):
                    newN = SearchNode(a, newS, n, cost)
                    agenda.push(newN, newN.cost)
    return None
```

← same as visited

} have I been here before?

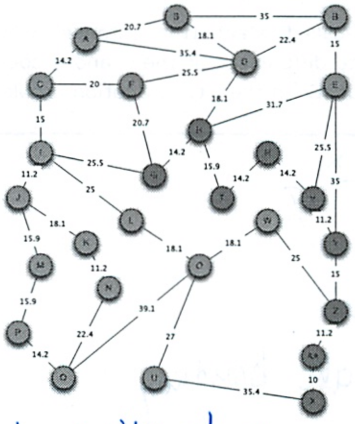
ucSearch with DP: From S to G



Agenda: [(SBDH, 9), (SBEHG, 7), (SACFG, 7)] Expanded: [S, B, A, D, E, C, H, F]

Found goal! On seperate paper

Search in Big Spaces: Getting to X



did not write down

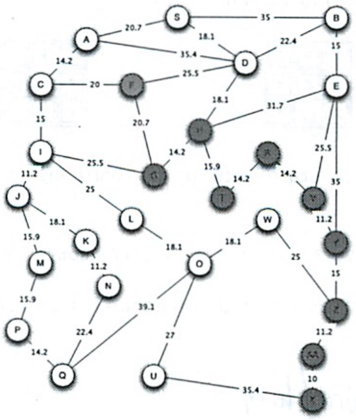
Search with heuristics

A **heuristic function** takes a state as an argument and returns a numeric estimate of the total cost that it will take to reach the goal from there.

- Used to focus the search in relevant direction.
- Actual cost + heuristic is a better estimate of total cost.
- For map-like problems, Euclidean distance from node to goal is good heuristic.

example map: straight line distance

Search in Big Spaces: Getting to X



~~[illegible]~~ (cost to next one + guess dist to goal)
as before new

A* = ucSearch with heuristics

```
def ucSearch(initialState, goalTest, actions, successor, heuristic):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    expanded = { }
    while not agenda.isEmpty():
        n = agenda.pop()
        if not expanded.has_key(n.state):
            expanded[n.state] = True
            if goalTest(n.state):
                return n.path()
            for a in actions:
                (newS, cost) = successor(n.state, a)
                if not expanded.has_key(newS):
                    newN = SearchNode(a, newS, n, cost)
                    agenda.push(newN, newN.cost + heuristic(newS))
    return None
```

↑

- takes state
- returns estimate

state → goal

Good and Bad Heuristics

We want heuristic close to actual distances but cheap to compute.

- The perfect heuristic: solve the problem and use the answer (too expensive).
- Trivial heuristic: 0 for all nodes (cheap but useless).
- **Admissible heuristic:** always an underestimate of the actual distance.

A* is guaranteed to find shortest path with admissible heuristic

Why always underestimate?

- guarenteed that can't be anything further out that costs less

Check Yourself

Would the so-called 'Manhattan distance', which is the sum of the absolute differences of the x and y coordinates be an admissible heuristic in the city navigation problem, in general?

No! Take Broadway

Check Yourself

If we were trying to minimize travel time on a road network (and so the estimated time to travel each road segment was the cost), what would be an appropriate heuristic function?

Crow gives distance - not time

but divide by max possible speed

remember want cheapest, not just a

Eight Puzzle

Think of it as moving the blank space.

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

Represent starting state with both board layout and location of empty space.

`startState = (((2, 8, 3), (1, 6, 4), (7, None, 5)), (2, 1))`

location of space

362,880 states! Only half are reachable from any given starting state.

Search problem

think about moving the blank

Formulation as a state machine

```
class EightPuzzleSM(sm.SM):
    legalInputs = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    def __init__(self, goal):
        self.goal = goal
    def nextState(self, state, action):
        (board, (x, y)) = state
        (dx, dy) = action
        newSpaceLoc = (util.clip(x + dx, 0, 2),
                       util.clip(y + dy, 0, 2))
        newBoard = swap(board, (x, y), newSpaceLoc)
        return (newBoard, newSpaceLoc)
    def getNextValues(self, state, action):
        return (self.nextState(state, action), 1)
    def done(self, state):
        return state == self.goal
```

ooh they like clip

↑defined

Running it

Heuristic search examples

Heuristics:

- `h0`: always 0
- `h1`: number of tiles out of place
- `h2`: total Manhattan distance of tiles out of place

Heuristic	Visited	Expanded	PathCost
h0	66	37	5
h1	14	7	5
h2	12	6	5

Different start state

Heuristics:

- `h0`: always 0
- `h1`: number of tiles out of place
- `h2`: total Manhattan distance of tiles out of place

Heuristic	Visited	Expanded	PathCost
h0	177,877	121,475	23
h1	26,471	16,115	23
h2	3,048	1,859	23

A numeric example

- States: integers
- Start state: 1
- Legal actions (and successors) in state n:
  $\{2n, n+1, n-1, n^2, -n\}$
- Goal test: $x = 10$

What are possible heursitics? Are they admissible?

i take log of distance
- ~~not~~ admissible?
  - no due to square
- Some problems naturally fit this
  - map
- hard to say structure of space
- its not easy to find in all cases
  - just do ~~Depth First~~ Breath First w/ DP
- 10-n
  - not admissible since not underestimate

This Week

Software lab: Path planning

Design lab: Map-making and planning

Nanoquiz Make-up: Wednesday, December 1: 4PM - 9PM in 34-501

You should have filled in the tutor problem to select which NQs you are going to make up:

- Everyone can make up NQ 1
- Everyone can chose any two additional two NQs to make up
- If you have excuses from S^3 for missed NQs, you can make those up as well.

If you choose to make up a NQ, the new score will replace the old score, even if it's lower.

S, 0 Expanded/visited

↳ (SA, 2), (SB, 1) S

(SB, 1)

↳ (SA, 2) (SBD, 3) (SBE, 4) SB

(SA, 2)

↳ (SBD, 3) (SBE, 4) (SAC, 5) (SAD, 4) SBA

↑ sum of cost

(SBD, 3)

↳ (SBE, 4) (SAC, 5) (SAD, 4) (SBDF, 7) (SBDH, 9) SBAD

↑ no reason to go back to A
we know path we aready found
was the best
– can not be better

(SBE, 4)

↳ (SAC, 5) ~~(SAD, 4)~~ (SBDF, 7) (SBDH, 9) (SBEH, 6) SBADE

↑ already found
Shortest to D – don't expand

(SAC, 5)

↳ (SBDF, 7) (SBDH, 9) (SBEH, 6) (SACF, 6) SBADEC

(SBEH, 6)

↳ (SBDF, 7) (SBDH, 9) (SACF, 6) (SBEHG, 7) SBADECH

↑ no D, already visited

②

(SACF, 6)

↳ ~~(SBDF,~~ 7) (SBDH, 9) (SBEHG, 7) (SACFG, 7) SBADECHF

↑ already at F w/lower cost

↑? why not crossed out?

↑ ? pick one at random?

SBEHG, 7

↳ (SBDH, 9) (SBEHG, 7) (SACFG, 7) SBADECHFG

at goal

↑ best!

↑ or this one

Software Lab 13: Plan 13 From Outer Space

You can do the lab on any computer. Do `athrun 6.01 update` to get the files for this lab, which will be in `Desktop/6.01/lab13/swLab/`, or get the software distribution from the course web page.

The relevant files in the distribution are:

- `plannerStandaloneSkeleton.py`: file to write your code in
- `worlds/mapTestWorld.py`, `worlds/bigPlanWorld.py`: files describing world configurations that you can read in as specifications of planning problems.

Read section 9.5 of the readings, if you haven't already.

In this week's software and design labs, we will build up to a system that can run on the robot, allowing it to make a map of the obstacles around it and plan a path to a desired destination. We'll start by formulating the basic planning problem as a search in a two-dimensional grid of states, build a machine that makes a map using sonar data, and make the robot dynamically replan new paths through the world as its map changes.

In this software lab, we will get our basic planning infrastructure up and running. We will work on making plans for a robot to move among states in a discretized map of the world. The states in the plan will be locations in the world that the robot can move among.

1 Grid Map Representation

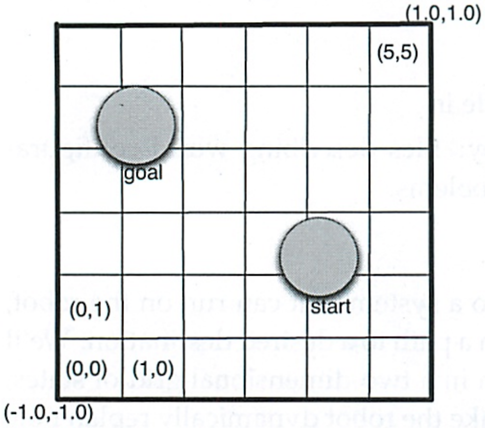
Consider a circular robot. We will be making plans for this robot moving around a simulated world containing obstacles. The state of the robot can be described by its pose: x in meters, y in meters, and θ in radians: (x, y) is the location of the robot's center and θ is the angle that the front of the robot makes relative to the x axis. We will use instances of the `util.Pose` class to represent the robot's poses in the real world. We can obtain the `util.Point` representing the x, y position of a pose using `pose.point()`.

Planning paths as continuous curves in x, y, θ is very hard, so we will instead model the robot's state space somewhat more coarsely. We will ignore orientation altogether and we will discretize the x, y positions of the robot into a grid, with indices ranging from 0 to `xN - 1` for x and from 0 to `yN - 1` for y.

So, a state for the robot, in the space that we will search for plans, is described by the coordinates of a grid cell – a tuple of two indices, `(ix, iy)`. It is important to be clear about when you are working with:

- a real world pose (an instance of `util.Pose` with coordinates in meters and radians), or
- a real world point (an instance of `util.Point` with coordinates in meters), or
- indices describing a grid cell (a tuple of two indices in a grid).

The figure below shows the robot in a world where the x and y coordinates vary from -1m to $+1\text{m}$, which is discretized into 6 intervals in each dimension, resulting in 36 grid cells. The lower-left cell always has indices `(0, 0)`, with x increasing to the right and y increasing going up.



We will use the `BasicGridMap` class (defined in `lib601.basicGridMap.py`) to represent the grid and indicate which grid cells contain obstacles. We can create a grid with obstacles from a world file of the kind that we use in soar. These world files specify the min and max values for x and y as well as boundary lines for the walls and obstacles.

These are the most important methods and attributes of the class `BasicGridMap` (more detailed documentation is available in the software documentation on the reference tab of the course home page):

- `pointToIndices(self, point)`: takes a `util.Point` representing coordinates of the robot in the real world map and returns a tuple of integer indices, representing the grid cell of the robot.
- `indicesToPoint(self, indices)`: takes a tuple of integer indices, representing the grid cell of the robot and returns a `util.Point` representing coordinates of the center of that cell in the real world map.
- `xN, yN`: number of cells in the x and y dimensions
- `robotCanOccupy(self, indices)`: returns `True` if the robot can be positioned with its center in this cell and not cause a collision with an obstacle in the world, and `False` otherwise.
- `xStep`: attribute representing the length, in meters, of a side of a grid cell; we assume the cells are square (so `yStep` equals `xStep`).

Step 1.

Wk.13.1.1 Solve this tutor problem to develop an understanding of the grid map representation.

1.1 Grid Dynamics

Now we will think about how to design a state machine class that represents the dynamics of the robot on a grid map. The `GridDynamics` class will be a state machine, whose inputs are actions the robot can take to move on the grid, and whose states are pairs of grid indices indicating the robot's position. We will use *uniform cost search*(UCS) to find shortest paths through the world defined by the grid dynamics: UCS requires each action to be annotated by its cost, so we will use the output of the state machine to encode the cost of each action.

The class needs to provide a `legalInputs` attribute and a `getNextValues` method. It does not need to supply a `done` method or a starting state: we will want to specify the starting state and the goal when we call the search procedure.

The state should be a pair `(ix, iy)` of indices representing the robot's position in the grid.

The state machine should allow 8 possible actions, moving to the four directly adjacent and the four diagonally adjacent grid cells. The elements of `legalInputs` will be the names of each of these actions. It doesn't matter what names you give the actions; in fact, the names can be tuples that describe the actions.

Remember that the input of the state machine will be one of the elements of the list of legal inputs; and the output of the `getNextValues` method should be a pair `(nextState, cost)`, where `nextState` is an `(ix, iy)` pair, and `cost` is a positive number representing the cost of taking that action. The cost of each move should be the distance the robot will travel, measured in meters (the length of a grid-cell side, in meters, is stored in the `xStep` attribute of instances of `basicGridMap.BasicGridMap`). Remember that a diagonal motion is longer than a horizontal or vertical one.

The `__init__` method of your `GridDynamics` class should take as input an instance of the `BasicGridMap` class, as described in section 1.

When implementing `getNextValues` be sure to consider the following:

- If the robot attempts to move into a square that it cannot occupy, it should stay where it was, but the cost should be the same as if the move had been legal.
- You do not need to worry about moving off the boundary of the map, because the boundary squares will already be marked as not occupiable.
- You do, however, have to be extra careful about moving diagonally: when your current and target squares are free, but one of the other two squares that are adjacent to both the current

and target squares is occupied, it is possible that the robot will have a collision. Such a move should be treated in the same way as attempting to move into a square that is occupied.

- *When we connect up with the map maker it may occasionally happen that the grid cell the robot is currently in is suddenly marked as not occupiable; your dynamics should allow the robot to move* ***out*** *of a cell that is not occupiable, as long as the cell it is moving into is occupiable.*

Step 2. Implement the `GridDynamics` class in the file `plannerStandaloneSkeleton.py`.

Check Yourself 1. This procedure is defined in `plannerStandaloneSkeleton.py`:

```
def testGridDynamics():
    gm = TestGridMap(0.15)
    r = GridDynamics(gm)
    print util.prettyString(r.legalInputs)
    ans1 = [r.getNextValues((1,1), a) for a in r.legalInputs]
    print util.prettyString(ans1)
    ans2 = [r.getNextValues((2,3), a) for a in r.legalInputs]
    print util.prettyString(ans2)
    ans3 = [r.getNextValues((3, 2), a) for a in r.legalInputs]
    print util.prettyString(ans3)

    gm2 = TestGridMap(0.4)
    r2 = GridDynamics(gm2)
    ans4 = [r2.getNextValues((2,3), a) for a in r2.legalInputs]
    print util.prettyString(ans4)
```

It creates two different instances of `GridDynamics`, tests them, and prints out the results. Be sure you understand what the results should be. Each time we create an instance of `TestGridMap`, a window will pop up showing the map (it's basically the same in both cases, except for the world is bigger in the second (the same number of cells, but they are larger)).

Step 3. Test your code by running `testGridDynamics()` or other test cases you find helpful, and be sure it is correct.

Step 4. Note that three of the test cases in the tutor problem are the same as the first three test cases in our `testGridDynamics`; but the answers may look different because we are iterating over your `legalInputs` attribute in one case, and our `legalInputs` attribute in the other, and they may be in a different order. The tutor takes that into account when checking.

Wk.13.1.2 Paste your `GridDynamics` class definition and any helper procedures it needs into this tutor problem; check it and submit.

1.2 Making a plan and sticking to it

Now that we have a state machine that represents the dynamics of our domain, we can run search algorithms on that state machine to find good paths through the space.

We want to construct a procedure

```
planner(initialPose, goalPoint, worldPath, gridSquareSize)
```

that plans a path using `ucSearch.smSearch` on the grid dynamics of a grid map. It should draw the resulting path in the map, and return it.

Step 5. Implement the `planner` procedure in `plannerStandaloneSkeleton.py`. It should return the plan found by the search.

You need to pass a grid map as an argument to your `GridDynamics` class initializer. You can create a grid map corresponding to a soar world with

```
basicGridMap.BasicGridMap(worldPath, gridSquareSize)
```

where `worldPath` is a string representing the name of a file containing a soar world definition, and `gridSquareSize` is the size, in meters, of a side of each grid cell. Don't worry about what `worldPath` needs to be; just take the argument you're given and pass it through to initialize the `BasicGridMap` instance.

Then, you use `ucSearch.smSearch` to search that machine for a path from the initial pose to the goal point. You will need to convert both the initial pose and the goal point into grid indices for planning.

When a `BasicGridMap` is created, it will create a new window, displaying the obstacles in the world. Your planner should draw the path it finds in that window. The `BasicGridMap` class provides the method `drawPath(self, listOfIndices)`, where `listOfIndices` is of the form `[(ix1, iy1), (ix2, iy2), ...]`, specifying a list of grid-index pairs. It draws the starting cell in purple, the ending cell in green, and the rest in blue. Remember that the plan returned by the search is a list of `(action, state)` tuples, so you cannot pass that in directly.

To get the algorithm to display the states it is visiting, define your goal test function (inside the planner procedure) to have this form:

```
def g(s):
    gm.drawSquare(s, 'gray')
    return yourGoalTestHere
```

where `gm` is the name of your instance of `BasicGridMap`, `s` is a pair of grid indices, and `yourGoalTestHere` is the actual expression that you are testing to see whether `s` is a goal state.

Step 6. At the top of `plannerStandaloneSkeleton.py`, there are definitions of two worlds you can test in, each with a reasonable start and goal point and grid square size specified. Test your procedure in a world by running `plannerStandaloneSkeleton.py` in Idle (be sure to start Idle with -n, so you can see the graphics), and then evaluating, for example,

```
testPlanner(mapTestWorld)
```

You should see a map window pop up, first showing the obstacles, the states as they are visited, and then, when the planner has completed, the path you drew.

Check Yourself 2. In `mapTestWorld.py` with the discretization, start, and goal as defined in `plannerStandaloneSkeleton.py`, you should find a solution with cost about 6.7 (that is, a path about 6.7 m long.) The search should visit about 800 nodes and expand about 270 states. Be sure you understand why some squares are being colored gray. Does this search seem efficient?

Step 7. Read section 9.6 of the readings, if you haven't already.

Think of a suitable admissible heuristic, implement it, and see how it affects the planning process. Be sure that your heuristic is expressed in the same units as the cost function.

Check Yourself 3. Test your heuristic search, first, in `mapTestWorld`. You should find that it doesn't make any difference in the length of the solution, though it may choose a slightly different path. You should find, however, that the number of nodes visited goes down to about 500 and the number of states expanded down to about 150.

Step 8. Now, test in `bigPlanWorld` with and without the heuristic. You should see a very noticable difference in the number of nodes expanded. Keep screenshots showing the paths and visited grid cells with and without the heuristic.

Checkoff 1. Demonstrate your search running in `bigPlanWorld.py` with the heuristic function. Compare its behavior to the search without a heuristic function. Explain the difference in visited states.

swLab 13

11/30

- plannerStandaloneSkeleton.py
- worlds/mapTestWorld.py , worlds/bigPlanWorld.py
- will build system that can run on robot that lets it make a map of obstifacles around it
- + plan a path to destination
- 2D grid of states
- today: get basic planning infrastuture up + running

Part 1 Grid Map Representation

- circular robot
- position (x, y, θ) w/ util.Pose
  ↑ angle from →
- planning path as curve hard, so model state space more coarsly
  - discretize into grid
    x → 0 → xN-1
    y → 0 → yN-1
- (ix, iy) - state of robot

②

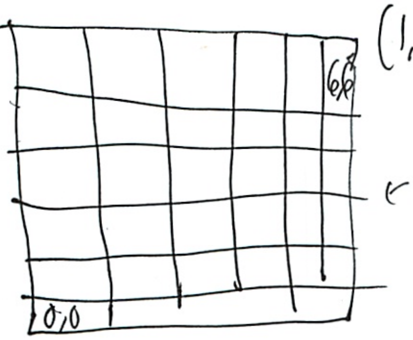
- w/ realworld Pose
- w/ " point
- w/ indices describing grid cell

(don't really get)

Use BasicGridMap class to represent grid + obstacles

- methods:
  - pointToIndices(self, point) → turns point into grid indicies
  - indicies toPoint(self, indicies) → reverse
  - XN, YN - # cells in each dimension
  - robotCanOccupy(self, indices) - ~~True~~ False if obstacle
  - xStep - lenght in meters of grid cell
    = yStep since square

13.1.1 Tutor Modeling The World



(1,0,1.0) ∈ ~~met~~ real/meter coords

← suppose to be squares

- robot has width of (2

③

a) What is starting pose of robot in meters

robot in (4, 2)

each grid $\left(\frac{1}{6}\right)$ *also need to add 1 to 1-index it

So $\left(\frac{5}{6}, \frac{3}{6}\right)$

* $-\frac{1}{12}$ to put at center of grid

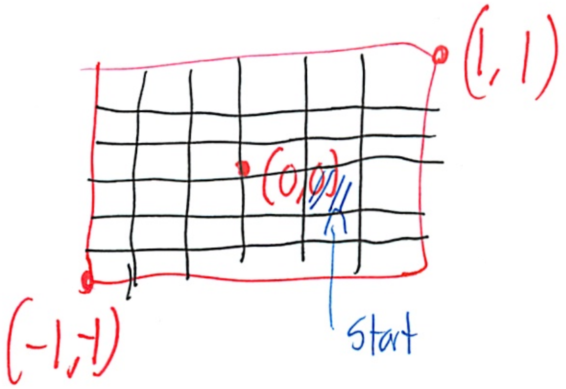
But robot not really at center of grid

Window is wide enough

Try $\left(\frac{5}{6}, \frac{3}{6}\right)$ ✗

*do they need ints?

Ohhh



thats why $\frac{1}{3}$

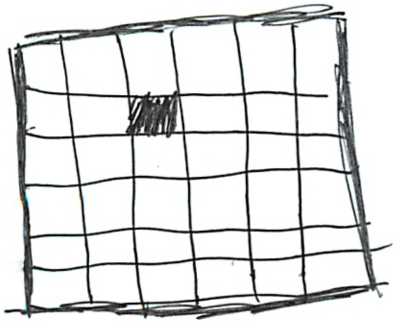
$\left(\frac{1}{3}, -\frac{1}{3}\right)$ ✓

Indicies (4, 2) ✓

$\left(\frac{1}{3}, -\frac{1}{3}\right)$ ✓

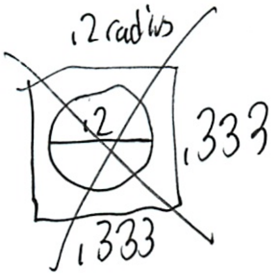
⑨

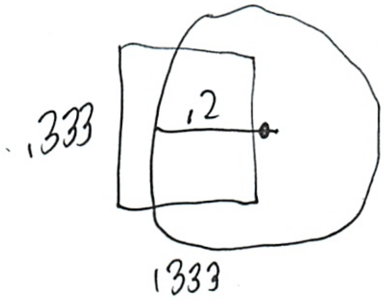
2.



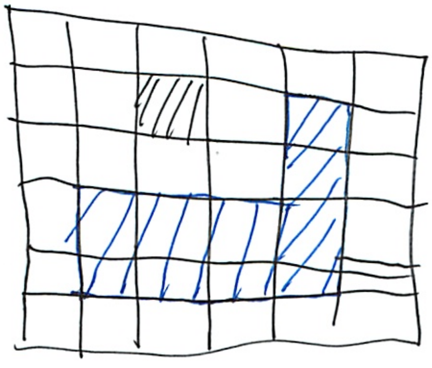
What are the grid indicies (ix, iy) that are Safe (no collisions)

- with robot radius .2
- don't know where robot is in within cell
  - be conservative - assume could be anwhere





- So first thought is all cells, except that one
- But if they want us to be conservative like that then nothing that touches a bad space



TA said was correct, now need to enter ✓ done

1.1 Grid Dynamics

Think about how to design a SM that represents
our GridDynamics

Will use Uniform Cost search
- each transition must be matched w/ a cost

need legalInputs
getNextValues methods

state = (ix, iy)

8 possible actions
- diagonal allowed

Cost = distance robot travels (measured in meters)
- pay attention to diagnal

init → takes BasicGridMap class

if robot tries illegal move - stays where it is
but cost same as if move legal!

don't worry about moving off map - bandry squares
already marked

be careful diagonal

won't work
- but still cost remember

⑥ If cell currently in becomes unoccupiable - allow to move out

Implement in plannerStandalone Skeleton.py
- has test procedure w/ map

---

do legal moves like before

init db

Now get Next Value much like before
- but w/ map.robotCanOccupy( )

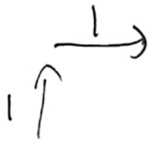
Where does it get cost?
- if not in state
- or is it?

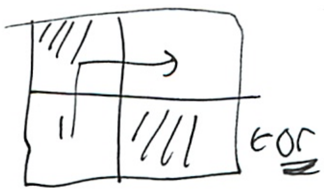
Or just return new cost

That seemed pretty easy

ⓧ answers off by a bit
- why?

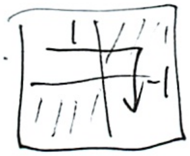
Perhaps its that diagnoal thing





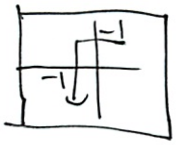
So for (1,1)
if (1,0) or (0,1)
bad - no go

⑦ being very conservative I think



(1, -1)

(1, 0) (0, -1)



(-1, -1) split

(-1, 0) (0, -1)

But 0 is current state!

ⓧ still no go

~~Absolut~~ TA: Make costs absolute

And if diagonal cost

- use $\sqrt{x^2 + y^2}$

Bad operand for type list

- oh typo

ⓥ works now

Note: They had a dist() procedure built somewhere

⑧ 1.2 Making a Plan and Stick to It

- run a search algorithm on SM to find good path through the space

  planner(initalPose, goalPoint, worldPath, gridSqSize)

  ~~that~~ uses ucSearch.smSearch

- Implement planner proc in plannerStandaloneSkeleton.py
  - return plan found by search
  - pass the grid map to GridDynamics

    basicGridMap.BasicGridMap(worldPath, gridSquareSize)

    ↑ string w/ soar world def file name

    just pass through

- will create new window w/ obstacles in world
  - draw paths w/ drawPath(self, listOfIndicies) method

- have a goal

  def g(s)

  gm.drawSquare(s, 'gray')

  return goalTest

⑨ Implement and Test (w/ numpy)

- So this is a wrapper around ucSearch.smSearch?
- need to convert pose to indicies for start state
  - does work even though I put in Pose, not Point
    - guess it ignores θ
      - which we don't care about here
- need goal test function
  - what is goal Point
    - point
- I see - turns gray if visited
- Oh s is a coord and GoalPoint is a Point()
  - Can see it filling in gray squares
  - Oh point also in meters?
    - is fractional

✓ Seems to work now

- Yeah solution cost 6.7

⑩

CY2 - Does not seem efficent b/c visiting every one

Step 7: Think about heuristic

- will ↓ states visited
- test in bigPlanWorld as well
- heuristics should make a big difference

Think as crow flyes would be good huristic

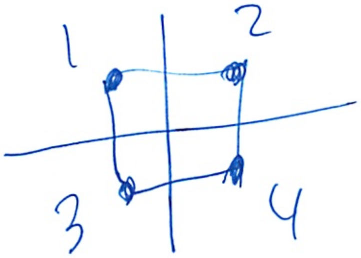
- yeah b/c distance is time
- TA: Good
  - Must be $\leq$ actual distance
  - Closest point in the cell

- Do first try ignoring this
- Actually worked pretty good
- Visited about half of the states
- But how closest point in cell?

✓ Checkoff 1

⑪

Would do



Got checkoff w/o this

-might not supposed to have

-[redacted]

nanoQuiz 2 Makeup

12/1

- Should be easy
- missed b/c got up late
- Hammock class
  - empty, anyone can sit
  - full - refused
  - next from same person grants
- 3 operations
  - `__init__`
  - Sit down, takes name
  - leave
  - multiple people can be on it
  - 3/4 checks work

✓ Checked

- easy

Nano Quiz 13

12/2

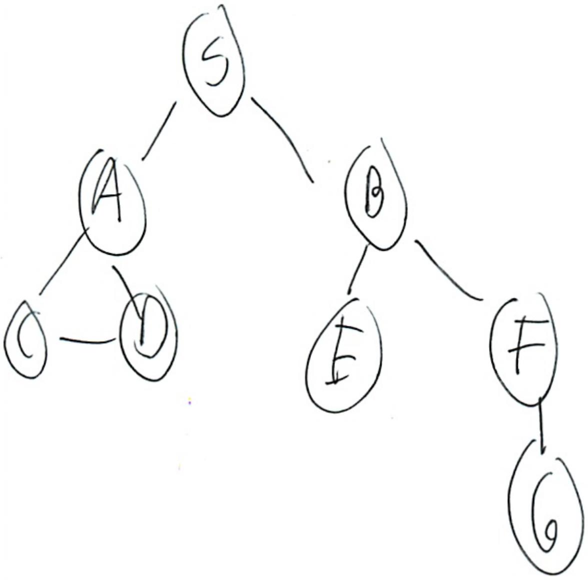
start S

goal G

alpha

no revisiting

BF no DR



S ↳ SA SB

$\overset{S}{A}$ ↳ $\overset{S}{B}$ $\overset{SA}{C}$ $\overset{SA}{D}$

$\overset{S}{B}$ ↳ $\overset{SA}{C}$ $\overset{SA}{D}$ $\overset{SB}{E}$ $\overset{SB}{F}$

②

$\overset{SA}{C} \hookrightarrow \overset{SA}{D} \quad \overset{SB}{\bar{E}} \quad \overset{SB}{F} \quad \overset{SAC}{D}$

Yeah right for breth

$\overset{SA}{D} \hookrightarrow \overset{SB\cancel{A}}{\bar{E}} \quad \overset{SB}{\hat{F}} \quad \overset{SAC}{D} \quad \overset{SAD}{C}$

$\overset{SB}{E} \hookrightarrow \overset{SB}{F} \quad \overset{SAC}{D} \quad \overset{SAD}{C}$

$\overset{SB}{F} \hookrightarrow \overset{SB}{\hat{F}} \quad \overset{SAC}{D} \quad \boxed{\overset{SBF}{G}}$

I think I missed one
-can write none

Now w/ DP

S

SA

SB

SAC

SAD

③

SBE

SBF

SBFG

✓ all right 1st try!

Design Lab 13: I Walk the Line

You can do the lab on any computer with soar. Do `athrun 6.01 update` to get the files for this lab, which will be in `Desktop/6.01/lab13/designLab/`, or get the software distribution from the course web page.

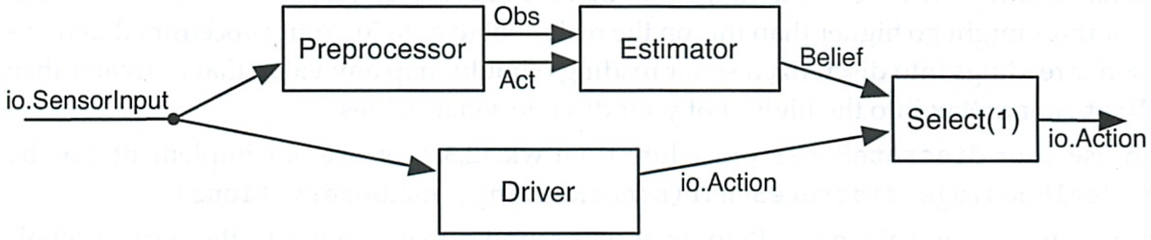
The relevant files in the distribution are:

- `lineLocalizeSkeleton.py`: file to write your code in
- `lineLocalizeBrain.py`: brain file to run, to test robot localization
- `worlds/oneDreal.py, worlds/oneDdiff.py, worlds/oneDslope.py`: world files for soar simulation

1 Overview

In this lab, we will implement and ultimately test in soar a robot localizer, as outlined in tutor problems **wk.12.3.2** and **wk.12.3.3**.

Here is the architecture of the system we will construct.



The `Driver` and `Select` state machine classes are already implemented. The `Driver` machine will generate instances of `io.Action` that make the robot move forward; the `Select(1)` machine takes tuples (or lists) of values as input and always returns the second element of the tuple as output. The robot knows the ideal readings for each of the possible discrete locations it might be in, but doesn't know where it is initially; the goal of the state estimation process is to determine the robot's location. The effect of this behavior is that the robot always drives forward, but the state estimation process is running in parallel, and as a side effect, the current belief state estimate of where the robot is in the world will be displayed in a window.

2 Preprocessor

Step 1. Tutor problem **wk.12.3.3** describes the preprocessor module in detail. Develop a strategy for implementing a state machine class that will behave as a preprocessor.

Check Yourself 1. Be sure your implementation plan is clear. What will the internal state of the preprocessor machine be?

What will the starting state be?

Talk to a staff member if this isn't clear to you.

Step 2. Implement the preprocessor by filling in the body of the state machine class `PreProcess` in `lineLocalizeSkeleton.py`. It should have a method `__init__(self, numObservations, stateWidth)`, where `numObservations` is the discrete number of observations and `stateWidth` is the width, in meters, of a discrete robot location. Here are some useful things to remember:

- Good sonar readings will be in the range 0 to `sonarDist.sonarMax`, which is set to 1.5, but actual readings might go higher than this on the real robot (e.g. to 5). Your procedure that maps actual sonar readings into discretized sonar readings should map any value that is greater than `sonarDist.sonarMax` into the highest of your discrete sonar values.
- You can use your `discreteSonar` procedure from **wk.12.3.2**, or use our implementation by calling `idealReadings.discreteSonar(sonarReading, numObservations)`.
- Be sure you understand `round` in Python: it will round a real number to the nearest whole number but, strangely, it keeps the value in floating point. So, to turn the result into an integer, you need to do `int(round(2.8))`, which will give you 3. Note that `int` truncates instead of rounding, so `int(2.8)` is 2, which is probably not what you want.

Make sure that the preprocessor generates a single value of `None` as output on the first step (problem wk.12.3.3 may have led you to think the output should be `(None, None)`).

Step 3. Test your preprocessor on the example from tutor problem **wk.12.3.3** as follows:

- Run your `lineLocalizeSkeleton.py` file in Idle.
- Make an instance of your preprocessor machine, called pp1, using parameters that match the tutor problem: 10 discrete observation values, 10 discrete location values, `xMin = 0.0` and `xMax = 10.0` (this means that the state width is 1.0 in this example).
- Do `pp1.transduce(preProcessTestData)`.
- Make sure the outputs match the ones from the tutor problem.
- Now make another instance, called pp2, using 20 discrete observation values, 12 discrete location values, `xMin = 0.0` and `xMax = 8.0`.
- Do `pp2.transduce(preProcessTestData)`.
- The outputs should be `[None, (10, 2), (3, 7)]`.

It will be useful, for later debugging, to make the `PreProcess` machine print its output on each step.

Checkoff 1. Show your `PreProcess` output for both cases to a staff member. Be sure the output is a single `None` on the first step.

3 State Estimator

The estimator module in our architecture will be an instance of `seGraphics.StateEstimator`, which we have already written; it's just like the state estimator you wrote last week, but it displays the current belief state and observation probabilities in a pair of windows. Whenever we make an instance of a state estimator, we have to pass in an instance of `ssm.StochasticSM`, which describes what we know about the system whose hidden state we are trying to estimate. Our job, in this section of the lab, is to create the appropriate `ssm.StochasticSM`, with an initial belief distribution, an observation model, and a transition model, for the robot localization problem. The state that we are trying to estimate is the *discretized* `x` coordinate of the robot's location, which can be in the range `0` to `numStates - 1`.

The file `lineLocalizeSkeleton.py` contains the following skeleton of a procedure that should construct and return the appropriate `ssm.StochasticSM` model. The parameters are:

- `ideal`: a list of ideal sonar readings, of length `numStates`
- `xMin, xMax`: the minimum and maximum `x` coordinates the robot can travel between
- `numStates`: the number of discrete states into which the `x` range is divided
- `numObservations`: the number of discrete observations

```
def makeRobotNavModel(ideal, xMin, xMax, numStates, numObservations):
    startDistribution = None
    def observationModel(ix):
        pass
    def transitionModel(a):
```

```
        pass
    return ssm.StochasticSM(startDistribution, transitionModel, observationModel)
```

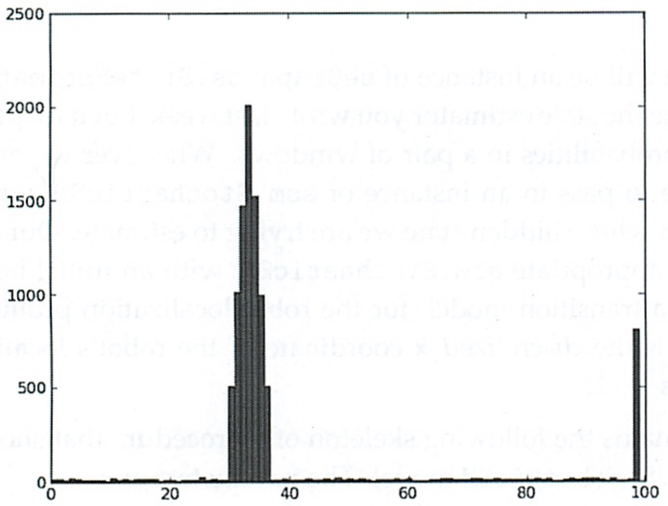
3.1 Initial distribution

Step 4. Define `startDistribution`, which should be uniform over all possible discrete robot locations. You can create a uniform distribution with `dist.UniformDist`.

3.2 Observation model

The observation model is a conditional probability distribution, represented as a procedure that takes a state (discrete robot location) as input and returns a distribution over possible observations (discrete sonar readings). Our job is to create an observation model that characterizes the distribution of sonar readings that are likely to occur when the robot is in a particular location.

This figure shows a histogram of 10,000 sonar readings generated in a situation in which there were 100 possible discrete sonar values over the range 0 to 1.5 m and where the ideal sonar reading was 0.5 m. The x axis is the discrete sonar reading and the y axis is the number of readings (out of 10,000) that fell into that interval.



It has the following features:

- There is always a non-trivial likelihood of getting an observation at the maximum range (due to reflections, etc). The maximum value is `sonarDist.sonarMax`.
- It is most likely to get an observation at the ideal distance, but there might be small relative errors in the observation (that is, we might see an object at 0.88 meters when it's really at 0.9 meters).
- There is some small chance of making any observation (due to someone walking by, etc.).

Pay particular attention to the 'width' of the noise distribution. It is important to write your mixture models so they are sensitive to the discretization granularity of the sonar readings: with the same amount of noise in the real world, the width in terms of the number of bins will be different for different granularities.

You can use `dist.MixtureDist`, `dist.triangleDist`, `dist.UniformDist`, and `dist.DeltaDist` to construct a distribution that describes well the data shown in histogram.

Check Yourself 2. Sketch out your plan for the observation model. Be sure you understand the type of the model and the mixture distributions you want to create. Ask a staff member if you're unsure on any of these points.

Step 5. Implement the observation model and test it to be sure it's reasonable. It doesn't need to match the histogram in the figure exactly.

For debugging, you can create a model, and then get the observation conditional probability distribution like this:

```
model = makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10, 10)
model.observationDistribution (1) ← some #
```

Here, `testIdealReadings` is the same set of ideal readings from tutor problem **wk.12.3.3**. Recall that these readings are already discretized.

Debug your distributions by plotting them, being sure that you have started Idle with `-n`. If `d` is a distribution you've created, you can plot it with `distPlot.plot(d)`.

If `observationModel` is your observation model, using the readings in `testIdealReadings`, write down the 4 highest-probability entries in `observationModel(7)` (this is an instance of `DDist`). What does the 7 stand for here?

at position x=7

Step 6. Now, make a model for the case with 100 observation bins, instead of 10.

```
model100 = makeRobotNavModel(testIdealReadings100, 0.0, 10.0, 10, 100)
```

Plot the observation distribution for robot location 7 in `model` and `model100`. Be sure they are consistent and correct.

3.3 Transition model

The transition model is a conditional probability distribution, represented as a procedure that takes an action as input and returns a procedure; that procedure takes a starting state (discrete robot location) as input, and returns a distribution over resulting states (discrete robot locations). You can compute the next location that would result if there were no error in odometry, and then return a distribution that takes into account the fact that there might be errors in the robot's reported motion.

For now, the only error in the transitions is due to discretization of the reported actions. Think about what discrete locations the robot could possibly have moved to, given a reported action of having moved k discrete locations. Use a triangle distribution to model the discretization error.

Check Yourself 3. Sketch out your plan for the transition model. Be sure you understand the type of the models and the distributions you want to create. Ask a staff member if you're unsure on any of these points.

Step 7. Implement the transition model and test it to be sure it's reasonable. Create a `ssm.StochasticSM`, and then get the transition model (which is a procedure that returns a conditional probability distribution) like this:

```
model = makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10, 10)
model.transitionDistribution
```

If `transitionModel` is your transition model, write down `transitionModel(2)(5)` (this is an instance of `DDist`). What do the 2 and 5 stand for here? Be sure the result makes sense to you.

3.4 Combined preprocessing and estimation

Step 8. Now we'll put the two modules we just made together and be sure they work correctly. Use `sm.Cascade` to combine

- an instance of your `PreProcess` class, and
- an instance of the `seGraphics.StateEstimator` class.

The `seGraphics.StateEstimator` instance is given your `ssm.StochasticSM` model, using 10 discrete observation values, 10 discrete location values, xMin = 0.0, and xMax = 10.0. Call this machine `ppEst`.

Check Yourself 4. Do `ppEst.transduce(preProcessTestData)`. Compare the result to the belief states in `wk.12.3.3`. Remember that you are now assuming noisy observations and noisy actions. Are your results consistent with the ones you found in the tutor?

Checkoff 2.	Show your answers to the questions above and your plots of the observation distributions to a staff member. Explain what they mean.

4 Putting it All Together

Now, we'll put all the machines together to make a behavior that can control the robot. The file `lineLocalizeBrain.py` contains all the scaffolding necessary. It makes one call that you need to think about:

```
robot.behavior = \
    lineLocalize.makeLineLocalizer(numObservations, numStates, ideal, xMin, xMax, y)
```

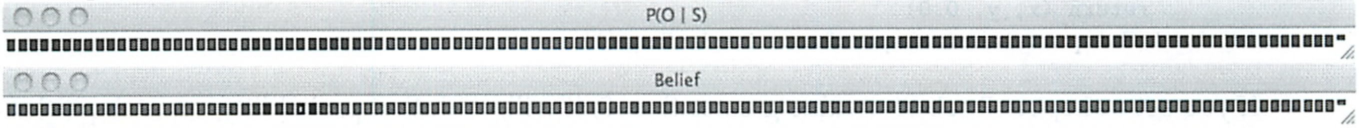
Step 9. In your `lineLocalizeSkeleton.py` file, implement the procedure `makeLineLocalizer` with the arguments shown above; it should construct a complete robot behavior, as outlined in the architecture diagram, whose inputs are `io.SensorInput` instances and whose outputs are `io.Action` instances. Read about the `sm.Select` state machine in the software documentation.

You will need instances of the preprocessor and estimator machines like those you made in the previous section, together with the driver state machine. The driver is a state machine whose input is an instance of `io.SensorInput` and whose output is an instance of `io.Action`. You can create it with

```
move.MoveToFixedPose(util.Pose(xMax, robotY, 0.0), maxVel = 0.5)
```

assuming that the robot starts at some location with y coordinate `robotY`, and will move to the right until its x coordinate is `xMax`.

Step 10. Start soar and run your behavior using `lineLocalizeBrain.py` in the world `worlds/oneDdiff.py`. It will pop up windows like these (to see the colors, look at it online):



The first window shows, for each state (possible discrete location of the robot), how likely the current observation is in that state. In this example, the robot's current observation is one that is likely to be observed when it is in any of the locations that is colored blue, and unlikely to be observed in the locations colored red. The second window shows the current belief state, using colors to indicate probabilities. Black is the uniform probability, brighter blue is more likely, brighter red is less likely. The actual location of the robot is shown with a small gold square in the belief state window.

You can use the **step** button in soar to move the robot step by step and look at and understand the displays. It is necessary to move the robot two steps before the displays become interesting.

Step 11. Now run your behavior in the world `oneDreal.py` (you will need to edit the line in `lineLocalizeBrain.py` that selects the world file, as well as select a new simulated world in soar). What is the essential difference between this world and `oneDdiff.py`?

Step 12. Now run your behavior in the world `oneDslope.py`, **without changing the world file selected in the brain**. This will mean that the robot **thinks** it is in the world `oneDreal.py`, and has observation models that are appropriate for that world, but it is, instead, in an entirely different world. What happens when you run it? What do the displays mean?

> *Checkoff 3.* Demonstrate your running localization system to a staff member. Explain the meanings of the colors in the display windows and argue that what your system is doing is reasonable. Explain why the behavior differs between `oneDreal` and `oneDdiff`. Explain what happens when there is a mismatch between the world and the model.

5 If you're interested in doing more...

Here are some possible extensions to this lab.

5.1 Handle Teleportation

Add this code to your brain file:

```
teleportProb = 0.0
import random
class RandomPose:
    def draw(self):
        x = random.random()*(xMax - xMin) + xMin
        return (x, y, 0.0)
io.enableTeleportation(teleportProb, RandomPose())
```

If you set `teleportProb` to a value greater than 0, it will, with that probability, on each motion step, 'teleport' the robot to an x coordinate chosen uniformly at random from the robot's x range. (maintaining the same heading and y coordinate). This is a good way to test your localization.

If necessary, modify your transition distribution so that it can cope with a world in which the robot might teleport. Think about what parameter in your model should match `teleportProb`.

Turn up the teleportation probability and see if your robot can cope.

5.2 Real robot

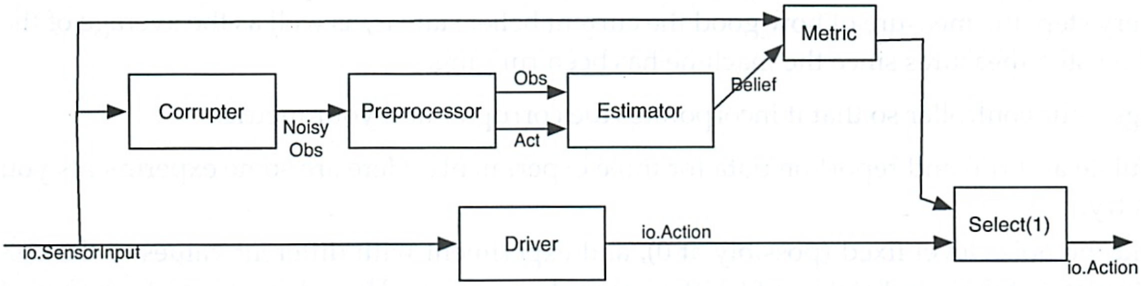
Try your localizer on the real robot. You'll need to:

- Set `maxVel` to 0.1
- Take out the `discreteStepLength` call from the brain.
- Change `cheatPose` to `False`
- Change the y value of the target pose for the driver to 0.0
- Use boxes covered with bubble wrap to set up a world that corresponds to `oneDreal.py`.
- (Possibly) adjust the amount of noise in your model of the sonar and the motion error.

5.3 Metric and simulation experiments

By looking at the belief-state windows during simulation, we can get a reasonably good idea of whether our estimator is working, but it is hard to tell how well. This is especially a problem, because there is no noise in the sonar readings generated by the simulator. In this upload problem, we will add noise to the simulated sonar and odometry readings, and we will devise a 'metric' for, or way of measuring, the effectiveness of the estimator.

To do this, we will need to add two new modules to our system, resulting in the architecture shown here:



These are the new modules:

- **Corrupter**: This machine takes in the true sonar and odometry readings and corrupts them with noise.
  - **Input:** Instance of `io.SensorInput`.
  - **Output:** Instance of `corruptInput.CorruptedSensorInput`.

  You can treat instances of the `corruptInput.CorruptedSensorInput` class as if they were instances of `io.SensorInput`: they have exactly the same attributes. The values in the output class are just slightly corrupted versions of the sensor and odometry values in the input. We have implemented this class for you. To make a new instance of this machine, do:

```
corruptInput.SensorCorrupter(sonarStDev, odoStDev)
```

  where `sonarStDev` is the standard deviation of the noise added to the sonar measurements and `odoStDev` is the standard deviation of the noise added to the x *component only* of the odometry. Start with very little corruption (e.g., standard deviations of 0.01). **A triangle distribution with a half-width of 3σ is a reasonable discrete approximation to a Gaussian with standard deviation σ.**

- **Metric**: The `Metric` state machine has these types:
  - **Input:** Pair (inp, belief) where `inp` is an instance of `io.SensorInput`, containing the true robot odometry, and `belief` is a distribution over the possible discrete `x` locations of the robot.
  - **Output:** Real number representing average estimation quality over the life of the machine.

  The metric state machine outputs a measure of how 'correct' the belief state is: that is, how much probability it assigns to some range of locations near the robot's true location. It also prints the metric value on each time step.

Knowing the true `x` location of the robot and the estimator's belief state, what is a good measure of how well the estimator is performing? Be sure that your measure is as insensitive as possible to the *size* of the discretization. Think, for example, whether the metric you have come up with will be as appropriate when the `x` range is divided into 30 bins as when it is divided into 300.

1. Implement a metric state machine. It doesn't really matter what it outputs, but it should print, on every step, the measure of how good the current belief state is, as well as the average of the per-time-step measures since the machine has been running.
2. Change your controller so that it incorporates the corrupter and your metric.
3. Formulate and run and report on data for three experiments. Here are some experiments you might try:
   - Hold the noise level fixed (possibly at 0), and experiment with different values of the discretization of the robot state and/or the sonar observations. How does the performance of the estimator vary?
   - Hold the discretization levels fixed, and vary the amount of noise in the world. You could hold your model constant (thus experimenting with the degree of match/mismatch between your model and the world). Or, you could make your model as accurate a reflection of the noise in the world as possible (thus experimenting with the limits of estimation as the sensor data becomes noisier).
   - Hold noise and discretization levels fixed, and experiment with different worlds. You can also try these worlds: `oneDreal.py` and `oneDslope.py`.

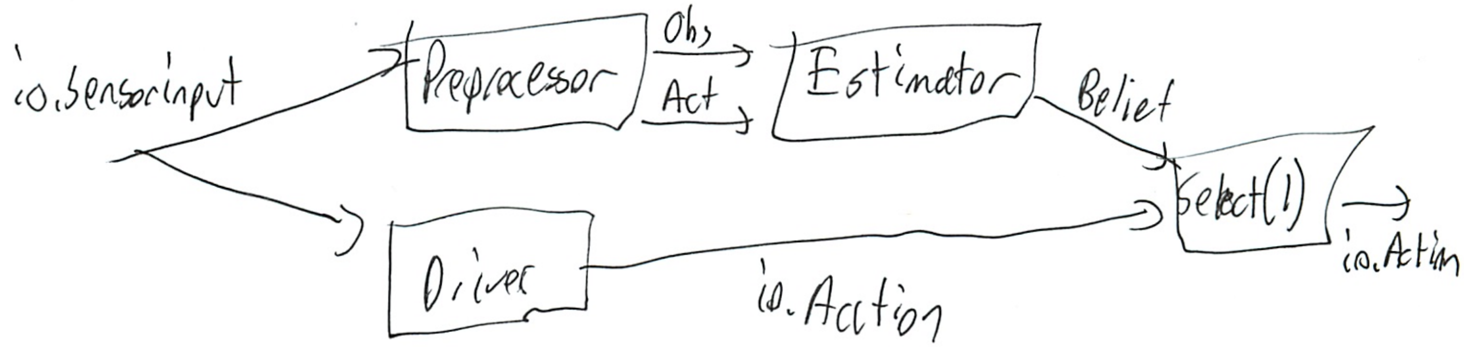
desLab 13

12/2

lineLocalizeSkeleton.py
lineLocalizeBrain.py – for soar
word files

---

implement localizer as in tutor 12.3.2 12.3.3



Driver + Select also implemented
↳ io.Actions that move robot forward
→ just returns 2nd value in tuble

- knows ideal reading for each location
- but does not know where it is initally
- so as it runs forward it refines its belief of where it is

②

Preprocessor

- i just copy from the tutor problem
- what is internal + starting state?
- remember it descitizes the reading
- but how does preprocessor have a state?
- oh diff w/ odometry
- return none

- start is none
- after Obs_{t-1} = state

Implement in PreProcess class in lineLocalizeSkeleton.py

- `__init__` (self, numObs, stateWidth)
  - readings range from 0 to sonarDist.sonarMax
    - can use their descitizing function
  - Use int(round())

- their descitizing returns 1 #
  - just robot discrete #
  - needs to report displacement descritizing

But that should not go in init

But also need like ~~SM~~ gNV methods

- its a SM

Just outputs None in 1st state

What is its input

- ~~tuple~~ io.input
- oh I see - pre process Test Data

SensorInput([.8, 1], util.Pose(1, .5, 0))

↑ Class instance

↑ Sonars
- just want 1st one

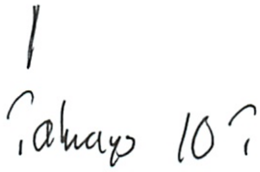
↑ xyT Tuple

Other cws

- discretize sonar input
- do they have a discritize displacement?
  - ↳ $x_t - x_{t-1}$
  - no - copy code from last time
  - don't think we ever did position

④

Just do it again



¿always 10 discrete positions?

- no numstates

TA: simple formula int(rand(($x_t - x_{t-1}$)/w)

I was going to do something more complex w/ loops
This is so much eaiser!
- its b/c the round function

Returns tuple

⊗ lots of coding errors to fix!

- getNextState - not getNextValue?

Can implement getNextValues instead

✓ running new

Procedure for testing

⊗ Does not match tutor
[None, (1,0), (1,0)]

~~Sonar0 input seems wrong!~~

No is right

Remember obs should be from t-1

-I was doing t

So need to change state (old obs, old pos)

Changed

Look at state after 1st run

✓ Obs right now

displacement right

Their math formula must be wrong

Or I implemented it wrong

Oh! I am passing wrong state interval

-not 10 but 1 =state width

✓ Works

New test 2

- confusing what it is

✓ Checkoff 1

⑥

State Estimator

- be an instance of seGraphics. StateEstimator we already wrote
  - which has quite complicated
  - ~~Bayes rule~~
    - broke down Bayes evidence ← converting ; view
      Total Prob ← all of the possibilities ; move
- displays current belief state + obs prob in windows
- instance of ssm. StochasticSM
- ? want to create this
  - inital belief
  - obs model
  - trans model
  - w/ discetized x coords 0 → num states -1
- lineLocalizeSkeleton.py
  - ideal for each state
  - xMin, xMax
  - num States
  - num Obs

  (think we worked w/ this before)
  (well entered #s)

⑦

```
def  makeRobotNavModel(ideal, xMin, xMax, numStates, numObs):
    Start dist ~~=None~~ _ _ _
    def obsModel(x):
        ...
    def transModel(a):
        ...
    return ssm.StocasticSM(start dist, transModel, obsModel)
```

Inital Dist

- Uniform
- w/ dist.UniformDist
- (for any # states)
- must make elements
  - range(start, stop, width)

    width = $\frac{Max - Min}{W}$

✓ Nice works out

⑧

Obs Model

- Cond probability
- procedure
  - state as input
  - output : d dist
- Oh right, sonar reading not always right
- So make a bunch
- and estimate from them
  (like 6.04)
- Use sonarDist . sonarMax
  - I guess that is MAP
- Can still be an error
- Can be totally wrong
- pay attention to the "width"
  - is that the var ?
- Can use dists to condstruct dists that descibe data shown in histogram
- TA: do something like it in code w/ a triangle dist
  - not just 1 # (MAP)
  - like what we did before

Prof: triangle b/c don't want it to spread

Build a mix w/ all 3 possibilities

- triangle
- low uniform anywhere
- delta at sonar Max
  - ↳ single point

C42

Implement

- ix is current position
- where triangle should be centered around
- don't say what it is - just do
- triangle will take care of
- must guess half width
  - very steep $\frac{\cancel{3}4}{100}$ = ←peak as well
- we cant have fixed # - must be % of # of states
  - 4% of the # states
- implement low, high as well
- delta
  - wait $\underbrace{\text{is this over}}_{\text{X positions}}$

- No! this is readings!
  - Min is <u>not</u> xMin its 0
  - Max = sonar max

Diff states as well
↳ for obs

not ix - but the ideal for that ix

- right do we know that?
  - ideal[ix]
- ix is <u>known</u> current location?
  - yeah location as input
- remember will test all of them as doing Bayes rule

- Now the mixture
- can you bias particular dists??
  - yes w/ p

$$p D_1(x) + (1-p) D_2(x)$$

- now must estimate p
  - baseline vs triangle → .05 ← guess
  - above vs delta → .85

Now test w/ model = make Robot Nav Model

⊗ Not a class
  - opps lot of self. to remove

⊗ step is 0
  - oh range can't do non ints
  - need rangeD
  - find online

⊗ triangle Dist is lower case - while the others
  - really !?!

I should be doing this over the discretized steps
  - <u>not</u> actual width
  - Check above on this
  - yeah not xMin, xMax, width = real
    but 0, numstates, 1
  - still not working

⊗ Opps was other problem - ¿ could have done ¿?
    w/ real values?

-¿ how plot dists again

⊗ Oh - does not look good

Break it down

Basline ✓

Triangle ⊗ -all 0s

- ideal works
- 0h since numObs = ~~[illegible]~~ 10
  - ~~must be at least 1~~
  - it rounds to 0
  - must be at least 1
- Max is not sonarMax
  - its numObs -1
- and so should delta
- ~~but no ideal~~
- ideals need to be descritized too! ??
  - is it value in m
  - ~~Yes~~ + no are discrete already
  - but dont use idealReadings 100 yet

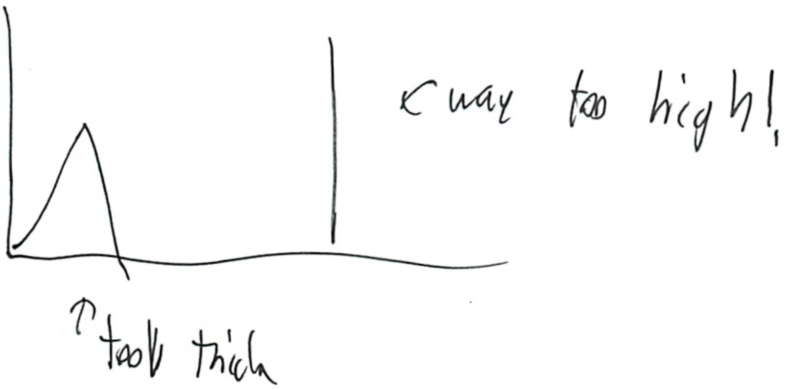
✓ Much better picture

~~Othe~~ baseline	.004
ideal	.764
delta	.204

⑬

Try w/ log if more detail

(✓) Got it to show



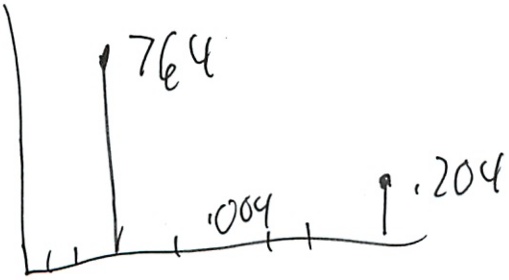
b/c set to .1 ~~th~~ ~~[illegible]~~ b/c when 10

– need to always round up to at least 1 obs stepwidth

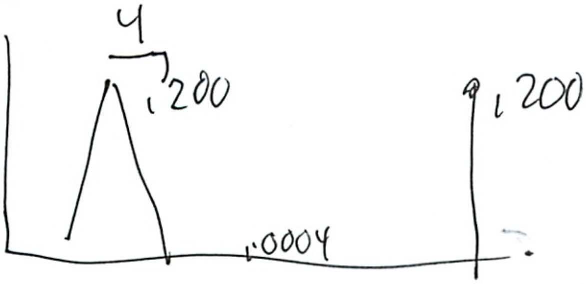
(✓) much thinner

– nice solution

width 10



width 100



⑭

base line higher

was .05

now .10

mix 1 vs delta

was .8

try .5

– oh got higher – wrong way

try .9

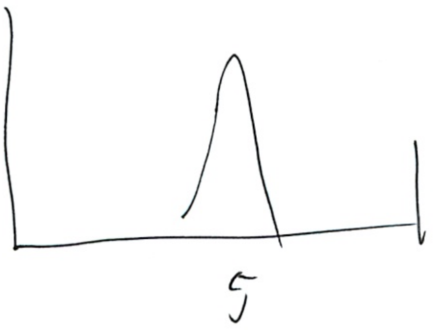
✓ much better – should be a little less than half

try .92

base line perhaps too low

TA: looks good!

Try w/ where should be 5



obsModel(7)

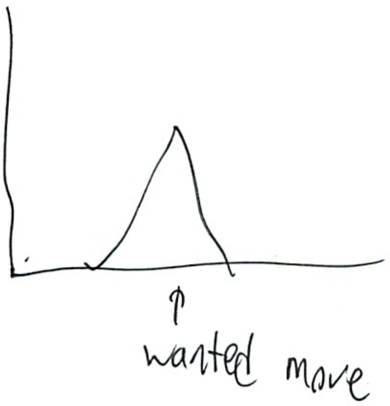
↑ means at pos 7

Sonar should be 5

Transition model

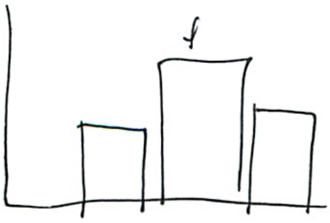
- Conditional prob dist
- inp → action
- out → proc → inp → start state → ~~the~~ robot location
  - Out → dist over resulting possible states

- no error in odometry for now - rethink this
- now just discretizing errors - descritizing actions not very exact



12/2
On own

or



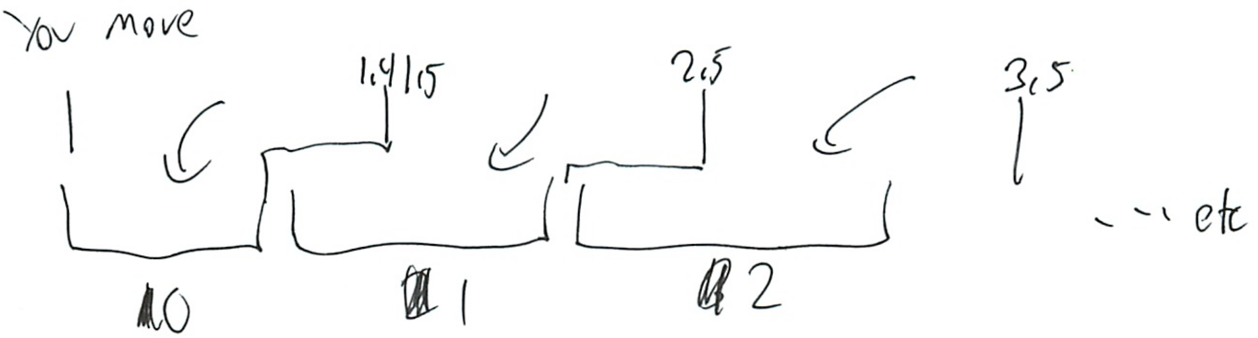
← well is triangle

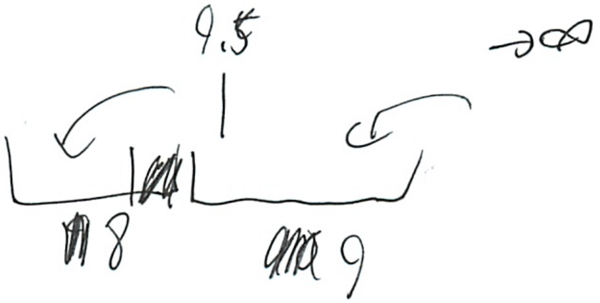
⑯

Well what is it?

You move 1.5 m

it says 2





So how represent w/ a dist?

If you move 1-1.49 its wrong

2 → 2.49

but if only 10 lines, can't represent

def transitionModel(a)

-a is discrete action

(? like → 1)

but then why are there two param in example? (2)(5)

-oh (2) is in first function the action

⑰

Then (5) is the second function's input

- robots location

Returns dist over resulting possible states

- ↳ So thinking back - shift the DDist by that # in a
  - ↳ shift w/ a for loop
  - ↳ or is there a bilt in function
- ↳ But first how to pull variables in
  - how only 1 parameter

Or this should likely be different

my drawing I did

- want to ask, lab opens in 30 min

Meanwhile check past hw

DDist { nominal loc^{+a} : .9 , ~~nominal loc +~~ a+1 = .9

↳ no perfect

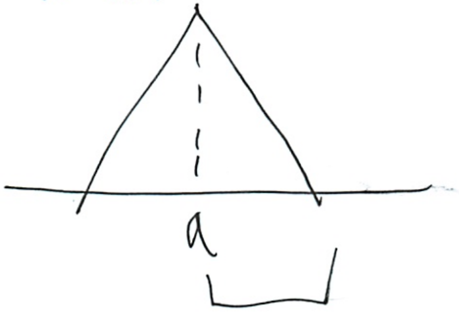
its describing

if

TA: Exact same one we used

- they just tried to explain it "better"

↳ aka worse



experimental variable = size of bin "step"

Could use old # [10%, 80%, 10%]

↳ but worse

return Triangle(a, step, xMin, xMax)

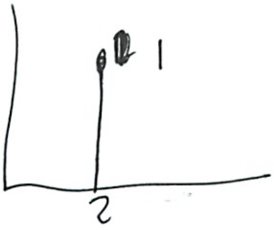
↳ here x move

⊗ has no __call__ method

This is that 2nd parameter

- well if did .prob(5) it should work

① Also dist is



Is that right?

Should be better w/ 100 - test

- no that is still 10 states

Or is this expected behavior?

3.4 Combined preprocessing + estimation

- just sm.Cascade
- what is the StateEstimator instance
  - ↳ says "given ssm.StocasticSM" model
  - ↳ which is makeRobotNavModel( )

⊗

↳ Make sure to pass instances!

Test it out - (confusing to know what is what)

⊗ Again DDist has no call

- So the call thing was wrong

So it should return procedure
1st is action
2nd arg is where started

12/2
OH

⑳

```
def transModel(a):
    def something(state):
        return dist.DDist({state+a: .8, ...
        or just do a triangle
         return triangleDist(state+a, 2, 0, numstates-))
    return something
```

↑ for now

↑ actual (not xmax) remember!

↑ really stupid distinction in my mind

ⓧ still big error
TA tried for like 15 min

~~pp.smCascade~~

```
ppEst. = sm.Cascade(ppl, seGraphics.StateEstimator(model))
```

forgot this
- need to fit into the state estimator

Wrapper
- draws things
- takes care of None
  ↳ skipping it

Ok Back on track

- Mine was not noisy transitions
- But otherwise close

✓ Checkoff 2

Part 4 Putting it All Together

in lineLocalizeBrain.py

- w/ lots scaffolding
- important call robot.behavior =
  lineLocalize.makeLineLocalizer(# obs, # states, ideal, xMin, xMax, y)

back in line localize skeleton, implement makeLineLocalizer

Should construct complete robot behavior

↳ input io.SensorInput
  output io.Action

Read about sm.Select

└ machine whose input is list and whose output is kth element of the list

Need instances of preprossor + estimator + driver

driver = move.MoveToFixedPose(util.Pose(xMax, robotY, 0.0), maxVel = .5)

↑ move slowly to the end of the line

Start soar w/ worlds/oneDdiff.py

Need to implement MakeLineLocalizer

- But what should I be doing?

TA: The sm on front of page
Combine w/ cascades + parallels

ⓧ need to change the hardcoded # in make RobotNavModel to variables

Ok now running - tied up my PC - + crashed

ⓧ Now crashing every time

TA: Hit step

- Actually working - but very slow!

TA: try lab laptop

But it does not seem very accurate

Runs better on lab laptop
↳ but still wrong

What am I doing wrong?

Prof Need at least triangle width 2 in obs

Prof biggest problem world in brain must match what you pick

⊗ Now nothing is updating

⊗ Now back to same problem

Comment out my test code

⊗

TA: Another hard code!

✓ Nice working!

And reloads correctly!

Having wrong belief file of course bad

✓ Checkoff M3

6.01: Introduction to EECS I

PCAP Recap

Week 14 *December 7, 2010*

Putting the pieces together

- Software design and implementation
- Circuits for sensing and motor control
- Linear systems controllers for trajectory following
- Probabilistic state estimation
- Trajectory planning

PCAP

To design and analyze complex systems, we have to find organizing structures that are *compositional*:

- primitives
- means of composition
- means of abstraction
- abstract entities can do anything a primitive can

> Infinite use of finite means. – von Humboldt

∞ sets possible

PCAP systems in 6.01, large and small

- **Procedures**: function composition and definition
- **Data**: lists, dictionaries, objects
- **Polynomials**: add, mul
- **State machines**: cascade, parallel
- **Terminating SMs**: sequence, repeat
- **Signals**: add, scale, delay, transduce
- **Systems**: cascade, feedback
- **Circuits**: resistor, voltage/current source, wiring abstraction via equivalents, isolation via op-amps - needed these
- **Plans**: individual actions, sequencing
- **Probability distributions**: joint, condition, marginalize
- **Probability distributions**: square, triangle, mixture

Classes + instances

Can build ∞ deep data structure

- mix + match

Follow-On Courses

- **6.041** – Probabilistic Systems Analysis
  Prereq: 18.02
- **6.042** – Mathematics for Computer Science - Discrete math
  Prereq: 18.01
- **6.02** – Introduction to EECS II - Communication
  Prereq: 18.03 or 18.06; 6.01
- **6.002** – Circuits and Electronics
  Prereq: 18.03; 6.01
- **6.005** – Elements of Software Construction - ramped back the work
  Coreq: 6.042; 6.01
- **6.006** – Introduction to Algorithms
  Prereq: 6.042; 6.01
- **6.007** – Electromagnetic Energy: From Motors to Lasers
  Prereq: 18.03; 6.01
- **6.034** – Artificial Intelligence
  Prereq: 6.01

SB in Computer Science and Molecular Biology

- Proposed new degree jointly administered by EECS and Biology
- Prepares students for graduate study in biology, in CS, and in emerging programs at the interface
- Prepares students for careers that leverage computational biology, e.g., pharmaceuticals, bioinformatics, medicine, ...

2/3 6-3
2/3 7

SB in Computer Science and Molecular Biology

Requirements:

- Mathematics and introductory subjects (3) – (18.03 or 18.06), 6.01, Math for CS
- Chemistry (2) – Organic Chemistry and Thermodynamics
- Introductory Lab (1,5) – Intro to Experimental Biology
- Foundational CS (3) – Software Engineering, Introductory and Advanced Algorithms
- Foundational Biological Science (3) – Genetics, Biochemistry, Cell Biology
- Restricted Elective in Computational Biology (1)
- Restricted Elective in Molecular/Cellular Biology (1)
- Advanced Undergraduate Project

IAP activities

- Maslab
  - IAP Robotics competition, listed as 6.186
- 6.270 – Autonomous Robot Design Competition
- 6.370 – BattleCode AI programming competition
- 6.470 (officially listed as 6.188)
  - Learn how to build a website, engage in an exciting competition "for glory, honor and money"
  - Lectures include HTML5, CSS, JavaScript, AJAX, PHP, MySQL, Ruby on Rails, Silverlight, and Flash
  - See web.mit.edu/6.470

Little Brother – Cory Doctorow

If you've never programmed a computer, you should. There's nothing like it in the whole world. When you program a computer, it does **exactly** what you tell it to do. It's like designing a machine—any machine, like a car, like a faucet, like a gas-hinge for a door—using math and instructions. It's awesome in the truest sense: it can fill you with awe.

A computer is the most complicated machine you'll ever use. It's made of billions of micro-miniaturized transistors that can be configured to run any program you can imagine. But when you sit down at the keyboard and write a line of code, those transistors do what you tell them to.

Most of us will never build a car. Pretty much none of us will ever create an aviation system. Design a building. Lay out a city.

Those are complicated machines, those things, and they're off-limits to the likes of you and me. But a computer is like, ten times more complicated, and it will dance to any tune you play. You can learn to write simple code in an afternoon.

Start with a language like Python, which was written to give non-programmers an easier way to make the machine dance to their tune. Even if you only write code for one day, one afternoon, you have to do it. Computers can control you or they can lighten your work – if you want to be in charge of your machines, you have to learn to write code.

Putting PCAP ideas to work

- PR2 Humanoid robot
- Little Dog

DL14: I'm the Map!

- **Checkoffs**: The checkoffs are due during software and design labs this week. You are expected to work during both labs, but **are not expected** to do any work outside of lab time.
- **Windows**: The graphics software that we are using seems to crash fairly reliably under Windows Vista and Windows 7. Please use a lab laptop instead.
- **Steps**: This lab is written with many small steps to help debugging. If you have done checkoff 1 and feel confident in your debugging skills, you can skip straight to implementing a system that passes checkoff 4, building a map that works reliably in medium noise conditions, using state estimation to aggregate multiple sonar readings over time.

You can do the lab on any computer that can run `soar` (modulo the comment above about crashes on Windows 7 and Windows Vista). Do `athrun 6.01 update` to get the files for this lab, which will be in `Desktop/6.01/lab14/designLab/`, or get the software distribution from the course web page.

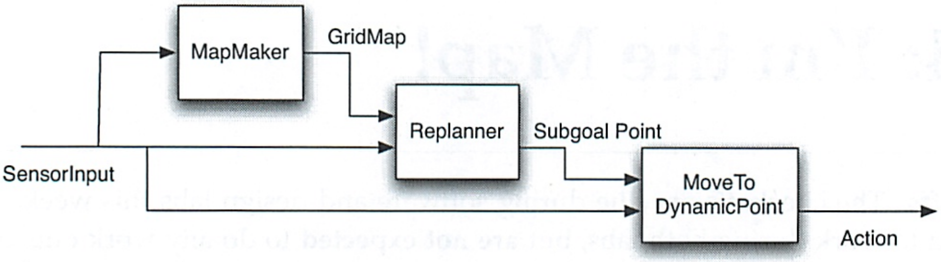
The relevant files in the distribution are:

- `mapMakerSkeleton.py`: file in which to write your map maker code
- `mapAndReplanBrain.py`: brain file to run the map maker
- `bayesMapSkeleton.py`: file in which to write your Bayesian map representation
- `robotRaceBrain.py`: brain file for running on the real robot
- `mapAndRaceBrain.py`: brain file for running in simulation; prints out timing information

1 Introduction

In this lab, we will connect the planner from Software Lab 13 with a state machine that dynamically builds a map as the robot moves through the world. The robot will, optimistically, start out by assuming that all of the locations it does not know about are free of obstacles, it will make a plan on that basis, and then begin executing the plan. But, as it moves, it will see obstacles with its sonars, and add them to its map. If it comes to believe that its current plan is no longer achievable, it will plan again. Thus, starting with no knowledge of the environment, the robot will be able to build a map. We'll start by building a simple map maker, then see what happens as the sensor data becomes less reliable, then adapt the map maker to handle unreliable sensor data.

Here is a diagram of the architecture of the system we will build.



Our architecture has three modules. We will give you our implementations of the replanner and the module that moves to a given point; you will concentrate on the mapmaker.

The `move.MoveToDynamicPoint` class of state machines takes instances of `util.Point` as input, and generates instances of `io.Action` as output. That means that the point at which the robot is 'aiming' can be changed dynamically over time. (Remember that you wrote a machine like this in lab 3!).

The `replanner.ReplannerWithDynamicMap` state machine takes a `goalPoint` as a parameter at initialization time. The `goalPoint` is a `util.Point`, specifying a goal for the robot's location in the world, which will remain fixed. The robot's sensor input (which contains information about the robot's current location) as well as the `DynamicGridMap` instance that is output by the `MapMaker` will be the inputs to this machine. The replanner makes a new plan on the first step, draws it into the map, and outputs the first 'subgoal' (that is, the center of the grid square to which the robot is supposed to move next), which is input to the driving state machine. On subsequent steps the replanner does two things:

1. It checks to see if the first or second subgoal locations on the current plan are blocked in the world map. If so, it calls the planner to make a new plan.
2. It checks to see if it has reached its current subgoal; if so, it removes that subgoal from the front of its stored plan and starts generating the next subgoal in the list as output.

2 Mapmaker, mapmaker, make me a map

Your job is to write a state-machine class, MapMaker, in the file `mapMakerSkeleton.py`. It will take as input an instance of `io.SensorInput`. Its state will be the map we are creating, which can be represented using an instance of `dynamicGridMap.DynamicGridMap`, which is like `basicGridMap.BasicGridMap`, but instead of creating the map from a file, it allows the map to be constructed dynamically. The grid map will be both the state and the output of this machine. The starting state of the mapmaker can just be the initial `dynamicGridMap.DynamicGridMap` instance.

For efficiency reasons, we are going to violate our state machine protocol and say that your `getNextValues` method should return *the same instance* of `dynamicGridMap.DynamicGridMap` that was passed in as the old state, returning this instance as the next state and the output. It should make changes to that map using the `setCell` and `clearCell` methods. If we were to copy it every time, the program would be painfully slow.

inherited by

& also GridMap – such as Point to indices

The `dynamicGridMap.DynamicGridMap` class provides these methods:

- `__init__(self, xMin, xMax, yMin, yMax, gridSquareSize)`: initializes a grid with minimum and maximum real-world coordinate ranges as specified by the parameters, and with grid square size as specified. The grid is stored in the attribute `grid`. Initially, all values are set to `False`, indicating that they are **not** occupied.
- `setCell(self, (ix, iy))`: sets the grid cell with indices `(ix, iy)` to be occupied (that is, sets the value stored in the cell to be `True`).
- `clearCell(self, (ix, iy))`: sets the grid cell with indices `(ix, iy)` to be **not** occupied (that is, sets the value stored in the cell to be `False`).
- `occupied(self, (ix, iy))`: returns `True` if the cell with indices `(ix, iy)` is occupied by an obstacle.
- `robotCanOccupy(self, (ix, iy))`: returns `True` if it is safe for the robot to have its center point anywhere in this cell.
- `squareColor(self, (ix, iy))`: returns the color that the grid cell at `(ix, iy)` should be drawn in; in this case, it draws a square in black if it is marked occupied by an obstacle. It draws squares in gray that are not occupied by obstacles but are not occupiable by the robot because they are too close to an obstacle square; the gray cells are computed by `robotCanOccupy`.

What should the mapmaker do? The most fundamental thing it knows about the world is that the grid cell at the very end of a sonar ray is occupied by an obstacle. So, on each step, for each sonar sensor, if its value is less than `sonarDist.sonarMax`, you should mark the grid cell containing the point at the end of the sonar ray in the map as containing an obstacle. The `sonarHit` procedure you wrote in tutor problem **Wk.12.2.1** is available as

```
sonarDist.sonarHit(dist, sonarPose, robotPose)
```

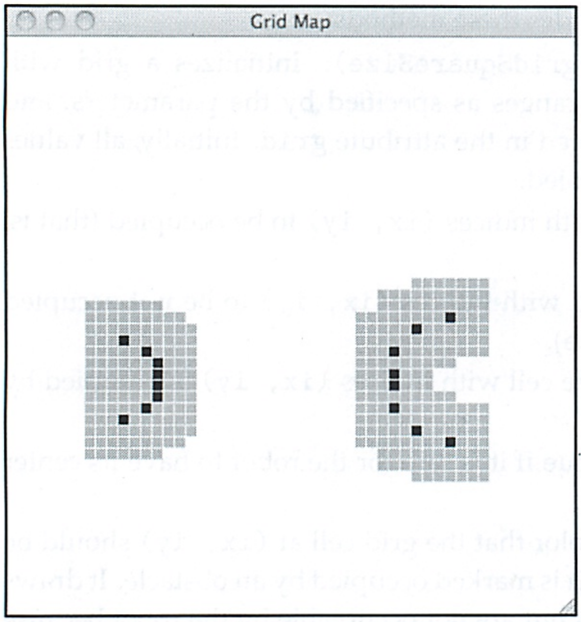
A list of the poses of all the sonar sensors is available in `sonarDist.sonarPoses`.

Step 1.

Check Yourself 1. Consider these two possible sensor input instances (each has a list of 8 real-valued sonar readings and a pose).

```
testData = [SensorInput([0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2],
                        util.Pose(1.0, 2.0, 0.0)),
            SensorInput([0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4],
                        util.Pose(4.0, 2.0, -math.pi))]
```

Be sure you understand why they give rise to the map shown below. Remember that the black squares are the only ones that are marked as occupied as a result of the sonar readings; the gray squares are the places that the robot cannot occupy (because it would collide with one of the black locations).



Step 2. Implement the `MapMaker` class. It will be called as follows:

```
MapMaker(xMin, xMax, yMin, yMax, gridSquareSize)
```

Remember to initialize the `startState` attribute and to define a `getNextValues` method that marks the cells at the end of the sonars rays in the input.

Step 3. Test your map maker inside idle (be sure to start with the `-n` flag). by doing this:

```
testMapMaker(testData)
```

It will make an instance of your `MapMaker` class, and call `transduce` on it with the `testData` from the Check Yourself question. Verify that your results match those in the figure.

Step 4. Now, test your code in soar. The file `mapAndReplanBrain.py` contains the necessary state machine combinations to connect all the parts of the system together into a brain that can run in soar. You can work in any of the worlds described in the top of the brain file; select the appropriate simulated world in soar, and then be sure that you have a line like `useWorld(dl14World)` that selects the appropriate dimensions for the world you're working in. Be sure to use the simulated world corresponding to the world file you have selected when you test your code.

A window will pop up that shows the current state of the map and plan. Black squares are those the map maker has marked as occupied. Sometimes squares will be drawn in gray: that means that, although they are not occupied by obstacles, they are not occupiable by the robot. Not all such non-occupiable squares will be drawn in gray (we don't want to redraw the whole screen too often), however.

Checkoff 1. Show the map that your mapmaker builds to a staff member. If it does anything surprising, explain why. How does the dynamically updated map interact with the planning and replanning process? Is the total path that the robot takes optimal with respect to the true map?

3 A noisy noise annoys an oyster

Step 5. By default, the sonar readings in soar are perfect. But the sonar readings in a real robot are nothing like perfect. Find the line in `mapAndReplanBrain.py` that says:

```
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: noNoise
```

and change it to one of

```
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: smallNoise
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: mediumNoise
```

This increases the default variance (width of gaussian distribution) for the sonar noise model to a non-zero value.

Check Yourself 2. Run the brain again in these noisier worlds. Why doesn't it work? How does the noise in the sensor readings affect its performance?

In fact, we get more information from the sonar sensors than just the fact that end of the ray is occupied. We also know that the grid cells along the sonar ray, between the sensor and the very last cell, are clear. Even when the sonar reading is greater than the maximum good value, you might consider marking the cells along the first part of the ray as being clear.

Step 6. Improve your `MapMaker` class to take advantage of this information. You will probably find the procedure `util.lineIndices(start, end)` useful: `start` and `end` should each be a pair of (x, y) integer grid cell indices; the return value is a list of (x, y) integer grid cell index pairs that constitute the line segment between `start` and `end`. You can think of these cells as the set of grid locations that could reasonably be marked as being clear, based on a sonar measurement. *Be sure not to clear the very last point, which is the one that you are already marking as occupied; although it might work now, if you clear and then mark that cell each time, it will cause problems in Section 4, when we use a state estimator to aggregate the evidence we get about each grid cell over time.*

Step 7. Test your new `MapMaker` in Idle by doing

```
testMapMakerClear(testClearData)
```

Note that this is `testMapMakerClear`, which is a different procedure from `testMapMaker`. It will create an instance of your map maker and set all of the grid squares to be occupied, initially. Then it will call `transduce` with this input:

```
testClearData = [SensorInput([1.0, 5.0, 5.0, 1.0, 1.0, 5.0, 5.0, 1.0],
                             util.Pose(1.0, 2.0, 0.0)),
                 SensorInput([1.0, 5.0, 5.0, 1.0, 1.0, 5.0, 5.0, 1.0],
                             util.Pose(4.0, 2.0, -math.pi))]
```

Check Yourself 3. Predict what the resulting map should look like, and make sure your code produces the right thing.

Step 8. Run `mapAndReplanBrain` in soar again and make sure you understand what happens with both no noise and medium noise.

Checkoff 2. Show your new map maker running, first with no noise and then with medium noise. We don't necessarily expect it to work reliably: but you should explain what it's doing and why.

4 Bayes Map

One way to make the mapping more reliable in the presence of noise is to treat the problem as one of *state estimation*: we have unreliable observations of the underlying state of the grid squares, and we can aggregate that information over time.

Our space of possible hypotheses for state estimation should be the space of all possible maps. But if our map is a 20 by 20 grid, then the number of possible map-grids is 2^{400} (each cell can either be occupied or not, and so this is like the number of 400-digit binary numbers), which is *much* too large a space in which to do estimation. In order to make the problem computationally tractable, we will make a very strong **independence assumption: the state of each square of the map is independent of the states of the other squares.** If we do this, then, instead of having one state estimation problem with 2^{400} states, we have 400 state estimation problems, each of which has 2 states (the grid cell can either be occupied or not).

Luckily, we have already built a nice state-machine class for state estimation, and we can use it to build a new subclass of `dynamicGridMap.DynamicGridMap`, where each cell in the grid contains an instance of `seFast.StateEstimator` (which you implemented in **Wk.11.2.3**).

Your job is to write the definition for the `BayesGridMap` class, in the file `bayesMapSkeleton.py`. Before doing this, you'll need to think through how to use state estimators as elements of the grid.

Recall that the argument to the `__init__` method of `seFast.StateEstimator` is an instance of `ssm.StochasticSM`, which specifies the dynamics of the environment. Here are some points to think about when specifying the world dynamics of a single map grid cell:

- There are two possible states of the cell: occupied or not.
- There are two possible observations we may make of this cell: it is free, or it was the location of a sonar hit.
- You can assume that the environment is completely static: that is, that the actual state of a grid cell never changes, even though your belief about it changes as you gather observations. But, if you want to, you can also consider the situation where the environment changes, perhaps because furniture is moved.

> *Check Yourself 4.* Remember that the sonar beams can sometimes bounce off of obstacles and not return to the sensor, and that when we say a square is clear, we say that it has nothing anywhere in it. What do you think the likelihood is that we observe a cell to be free when it is really occupied? That we observe it as a hit when it is really not occupied? What should the prior (starting) probabilities be that any particular cell is occupied?
>
> Decide on possible values for the state of the cell. Assume that the observation can be either `'hit'`, if there is a sonar hit in the cell or `'free'` if the sonar passes through the cell. **To forestall confusion, pick names for the internal states that are neither `'hit'` nor `'free'`.**
>
> If you are having trouble formulating the starting distribution, observation and transition models for the state estimator, talk to a staff member.

Step 9. Write code in `bayesMapSkeleton.py` to create an instance of `ssm.StochasticSM` that models the behavior **of a single grid cell.**

Step 10. Test your grid cell model by doing

```
testCellDynamics(cellSSM, yourTestInput)
```

where `cellSSM` is an instance of `ssm.StochasticSM` and `yourTestInput` is one of the lists below. It will create an instance of a state estimator for a single grid cell and feed it a stream of observations. Then it will call `transduce` with the data input.

What is its final degree of belief that the cell is occupied if you give it this input data? (Why are the `None`s here?)

```
mostlyHits = [('hit', None), ('hit', None), ('hit', None), ('free', None)]
```

How about if you give it this input data?

```
mostlyFree = [('free', None), ('free', None), ('free', None), ('hit', None)]
```

Now it is time to think through a strategy for implementing the `BayesGridMap` class. You will have to manage the initialization and state update of the state estimator machines in each cell yourself. You should be sure to call the `start` method on each of the state-estimator state machines just after you create this grid. You will also, whenever you get evidence about the state of a cell, have to call the `step` method of the estimator, with the input `(o, a)`, where `o` is an observation and `a` is an action; we will be, effectively, ignoring the action parameter in this model, so you can simply pass in `None` for `a`.

You can remind yourself of the appropriate methods for creating a state estimator and for starting and stepping a state machine by looking at the online software documentation.

Your `BayesGridMap` will be a subclass of `DynamicGridMap` and can be modeled directly on the following aspects of `DynamicGridMap.py`:

```
class DynamicGridMap(gridMap.GridMap):
    def makeStartingGrid(self):
        return util.make2DArray(self.xN, self.yN, False)
    def squareColor(self, (xIndex, yIndex)):
        if self.occupied((xIndex, yIndex)): return 'black'
        else: return 'white'
    def setCell(self, (xIndex, yIndex)):
        self.grid[xIndex][yIndex] = True
        self.drawSquare((xIndex, yIndex))
    def clearCell(self, (xIndex, yIndex)):
        self.grid[xIndex][yIndex] = False
        self.drawSquare((xIndex, yIndex))
    def occupied(self, (xIndex, yIndex)):
        return self.grid[xIndex][yIndex]
```

Here is some further description of the methods you'll need to write. Remember that the grid of values in a `DynamicGridMap` is stored in the attribute `grid`. We don't need to write the `__init__` method, because it will be inherited from `DynamicGridMap`.

- `makeStartingGrid(self)`: Construct and return two-dimensional array (list of lists) of instances of `seFast.StateEstimator`. You can use the attributes `xN` and `yN` of `self` to know how big to make the array. You should use `util.make2DArrayFill` for this (be sure you understand why `make2DArray` is not appropriate).
- `setCell(self, (xIndex, yIndex))`: This method should do a state-machine update on the state machine in this cell, for the observation that there is a sonar hit in this cell. And it should redraw the square in the map in case its color has changed.
- `clearCell(self, (xIndex, yIndex))`: This method should do a state-machine update on the state machine in this cell, for the observation that this cell is free. And it should redraw the square in the map in case its color has changed.

- `occProb(self, (xIndex, yIndex))`: This method returns a floating point number between 0 and 1, representing the probability with which we believe that the specified cell is occupied. This is used for display purposes by the `squareColor` method, which has already been written.
- `occupied(self, (xIndex, yIndex))`: This method returns `True` if the cell should be considered to be occupied for the purposes of planning and `False` if not. You may have to experiment with this a bit in order to find a good threshold on the probability that the square is occupied. Use the `occProb` method specified above.

Step 11.

Wk.14.2.3	Solve this tutor problem on making collections of object instances.

Step 12. Now, implement the `BayesGridMap` class in `bayesMapSkeleton.py`. It already has the `squareColor` method defined.

Step 13. Test your code in Idle by:

- Changing your `MapMaker` to use `bayesMap.BayesGridMap` instead of `dynamicGridMap.DynamicGridMap`. No further change to that class should be necessary.
- Running `mapMakerSkeleton.py` in Idle, and then typing in the shell:

```
testMapMakerN(1, testData)
```

It will do an update with the same data as we used in with the dynamic grid map. Now, the window that pops up uses a different color scheme: white means likely to be clear and bright green means likely to be blocked, with continuous variation between the colors. If a cell is considered to be blocked, it is colored black; if a cell is not blocked, but is also not occupiable by the robot, it is colored red.

If you type

```
testMapMakerN(2, testData)
```

then it will update the map 2 times with the given data.

Check Yourself 5. Try it with two updates. Try it with testClearData. Be sure it all makes sense.

Step 14. Now, test your mapper in soar, by running `mapAndReplanBrain` as before. You might find it particularly useful to use the step button.

Checkoff 3. Demonstrate your mapper in `mapAndReplanBrain` using your `BayesMap` module with medium and high noise. If it doesn't work with high noise, explain what the issues are.

5 Real robot

Now, let's see how well this works in the real world! Take your laptop to one of the real-world playpens, connect it to a robot, and run `robotRaceBrain.py`. You may have to adjust the parameters in your state estimator (typically, the false-positive rate, or the threshold for considering a square to be blocked) in order for it to work reliably.

Checkoff 4. Demonstrate your mapper on a real robot. You can move the obstacles around in the playpen for added fun, but be sure that you don't make it impossible to go from the start to the goal.

6 Go, speed racer, go!

Thus far, we worked on speeding up planning time by using a heuristic. And our robots can avoid obstacles by building a map and planning paths to avoid them. Now, we're going to work on making our robots move more quickly through the world. Your job during this lab is to speed up your robot as much as possible; at the end, we'll have a race.

In this section we will concentrate on the `Replanner` and `MoveToDynamicPoint` modules.

The `mapAndRaceBrain.py` is currently set up to work in `raceWorld.py`: select that as the simulated world, and run the brain. To change the world you're working in, change the `useWorld` line in the brain (and remember to change the simulated world, as well).

When you run using `mapAndRaceBrain.py`, you'll notice that, when the robot reaches its goal, it stops and prints out something like

```
Total steps: 320
Elapsed time in seconds: 209.554840088
```

That's the number of soar primitive steps it took to execute your plan, and the amount of elapsed time it took. These numbers will be your 'score'. Note that we are aiming for low scores!

You can debug on your own laptop or an athena machine, but scores will only be considered official if they are run on a lab laptop.

Step 15.

Check Yourself 6. Run your robot through `raceWorld` and see what score you get. Write this down, because it's your baseline for improvement.

You will notice that there are several things that slow your robot down as it executes its plans:

- Each individual step, from grid cell to grid cell, is controlled by a proportional controller in `move.MoveToFixedPoint`. The controller has to slow down to carefully hit each subgoal.
- Rotations take a long time.

Below are some possible strategies for addressing these problems. You don't need to do any or all of these. If you pick one of your own (which we encourage!), talk to a staff member.

You can speed up the robot by producing a plan that requires less stopping and/or less turning. Implement these by editing your `GridDynamics` class or the `ReplannerWithDynamicMap` class in `replannerRace.py` (read that code carefully).

1. Plan with the original set of actions, but then post-process the plan to make it more efficient. If the plan asks the robot to make several moves along a straight line, you can safely remove the intermediate subgoals, until the location where the robot finally has to turn.
2. Augment the space in which you are planning to include the robot's heading. Add an additional penalty for actions that cause the robot to rotate. Experiment with the penalty to improve your score. (This is pretty hard to get right; only do it if you have lots of spare time).
3. Increase the set of actions, to include moves that are more than one square away. You can use the procedure `util.lineIndicesConservative((ix1, iy1), (ix2, iy2))` to get a list of the grid cells that the robot would have to traverse if it starts at `(ix1, iy1)` and ends at `(ix2, iy2)`. This list of grid cells is conservative because it doesn't cut any corners.

You can also speed up the execution of the paths by changing the gains and tolerances in the `move.MoveToDynamicPoint` behavior (in `move.py`). Read the code in the file to understand what these parameters mean, and then consider adjusting them to improve the robot's behavior. But be sure you do not cause crashes into obstacles! You can edit these lines of code in `mapAndRaceBrain.py`.

```
move.MoveToFixedPoint.forwardGain = 1.0
move.MoveToFixedPoint.rotationGain = 1.0
move.MoveToFixedPoint.angleEps = 0.05
```

You are not allowed to change the `maxVel` parameter.

Step 16. Implement some improvements to make the robot go faster.

> *Check Yourself 7.* Post your best scores in simulation on `raceWorld` and `lizWorld` on the board.

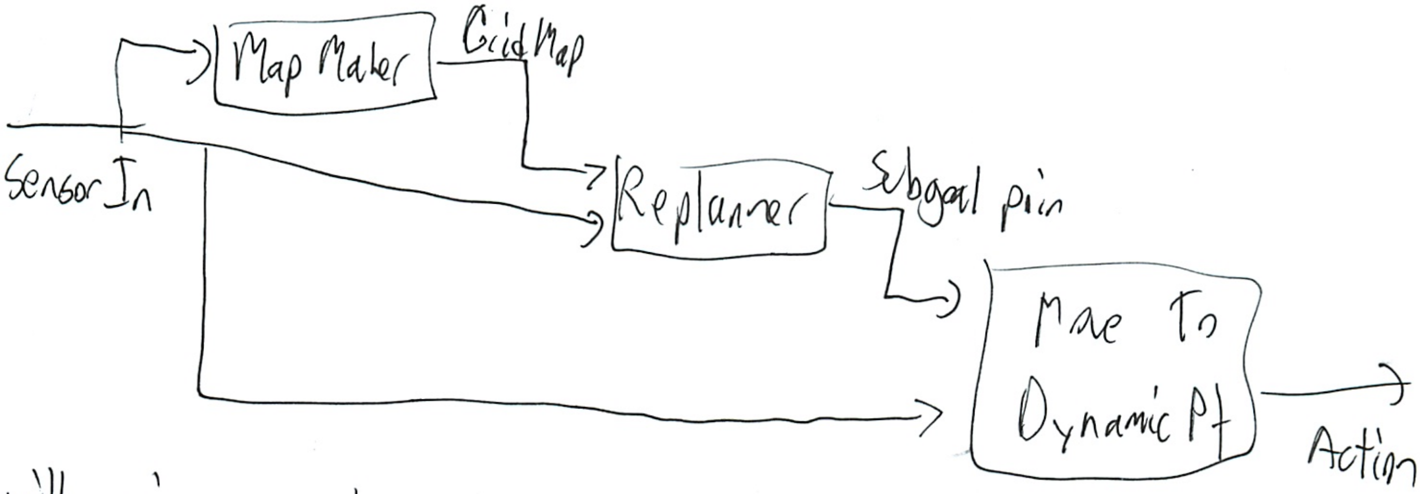
Step 17. Run on a real robot, using `robotRaceBrain.py`. It has a good pair of start-goal values and boundaries for the size of the big world in the front of the room. It will only work on the robot. If you want to test in simulation, you can switch back to using `mapAndRaceBrain.py`.

> *Checkoff 5.* Post your best score on the robot on the blackboard. Special prizes to the winners!

DL 14

12/7

- Work on lab laptop
- have skeleton + lab files
- use planner from Sw Lab 13 w/ SM
  - build map as robot moves through world
- at first → thinks there are no obsticales
  - when sees - adds to map
  - eventually use unreliable sensor data



- will give replanner + move module
  ↳ (last lab ::)
- move.MoveToDynamicPoint
  ↳ in util.Point
  ↳ out io.Action

Replanner. ReplannerWithDynamicMap
  ↳ init goal Point → util.Point
  ↳ in Sensor info
    Dynamic Grid Map
  ↳ out subgoal to move two

- Checks if 1st or second subgoals are blocked
  ↳ if so → replan
- checks if current pos = subgoal
  ↳ removes it from plan

2. Mapmaker make me a map

- write MapMaker class
  ↳ in io.SensorInput
- map is instance of dynamicGridMap
  ↳ similar to BasicGridMap
- actually change it w/ setCell(), clearCell()
  - violates SM principles for speed

③

Provides a bunch of methods

Fill in w/ sonar
  - if SonarHit - know obstical at that distance

Implement mapMaker
- this is the SM?
- yeah
- transduce data over

State = map

Then input are sensor values
  - measure
  - set cells
  - ? return display

Oh it shows pic for you

Look at desLab 13 - just the start

But for all sensors

- ~~[illegible]~~ -

Or no ideal readings?
- ? descitizing differently?
- Oh just use SonarDist. SonarHit from wk 12

④

Returns point hit in global frame
- ah remember in 3D frame

Only if value less than sonar max

dist = distance, not distribution!
- aka sonar value

- Need poses for sensors

sonarDist.sonarPoses
↑plural

ⓧ sonarHit takes an x,y point → not a util.Point!

ⓧ Must be indicies
↳ oh must do PointtoIndicies

✓ Cool works!

Now try in soar

ⓧ Return (state, state)
^ I had None

Really very cool!
?How is it getting that blue goal path ?
- our ~~state~~ planner we wrote earlier
plans it out quickly in sw time - so

⑤

Would be faster if knew the map

- could ~~do it in ha~~ plan out fully w/ no exploring

Exploring in SW - fast
HW - slow

Checkoff 1 ✓

3 Noisy Noise

Sonar Readings not perfect in real life

turn on some noise

↳ ie increase variance

Little noise - pretty much same

except gets stuck when draws gray into a lot of points

- search fails since everything is blocked in

Happens earlier in Medium noise

So mark all the cells up to sensor end point as clear

- Use util.lineIndicies(start, end)
- don't clear last point! ↑ indicies!

returns a list

↳ lots of debugging!

ⓧ Still a lot of grayness

Don't think it should test yet

Do other test w/ clear

- seems to do something strange
- guess that is what they want

(✓) Now see it working

Now it can find its way out by clearing gray spots by rechecking

; More like a second opinion than clearing what is in front of it

; I guess that is 2nd opinion

(✓) Checkoff 2

Part 4 Bayes Map

One way to make mapping more reliable is to use state estimation - aggregating information over time

Each square is independent - treat independently

⑦

Write a BayesGridMap class definition in BayesMapSkeleton.py

- before think about state estimation

__init__ is instance of a ssm

- S {free, occupied}
- O {empty, hit}
-

Environment is static (obstacles don't move)

Estimate P(cell free | occupied) due to sonar mistakes

↑ observed ↑ actual

12/9 Class

P(occupied) prior?

O = {hit, free}

S should be someting else (~~obs~~ obstacle, empty)

Write code to model a single cell

(will ~~write~~ run multiple times - once for each cell)

(much more managable - ~~think of how to d~~ should have thought of this!)

Test with testCellDynamics(cellSSM, yourTestInput)

⑧ obs model

P(obs | state)

```
if s = obstacle
    return DDist free .1 hit .9
else
    return DDist free .9 hit .1
```

↑ guessing now

trans model

- what form was this again
- (all a blur)
- return dist.triangleDist(state + a, 2, 0, numstates - 1)
- but this is transition from obstacle to empty

or other way around

- prob actually pay attention
- ? nominal value ?

Prof: can't transition from one state to another
So just tran model s:1
But need weird function thing

```
def vGivenAS(a):
    def something(s):
        return dist.DDist({s: 1})
    return something
```

Prof: We do this weirdness to allow conditional probabilities in certain cases

Now need to define model

- start dist
- obs dist
- trans dist

So that is ssm.StochasticSM(startDist, vGivenAS, oGivenS)

↑ P(0) ↑ trans ↑ obs

p(0) = {obstacle .2 empty .8}

↑ again guessing

Getting output data now

Does not look right - fixed a value

✓ makes sense now

⑩

Now think through strategy for implementing BayesGridMap class

- manage initalization + state update for each cell
  + start ^ step() w/ (0,a)
  ^ None since we are ignoring

- BayesGridMap like a DynamicGridMap
  ~~use much of same~~ code just a sub class of

makeStartingGrid(self) : construct the 2D array
w/ util.make2DArrayFill

setCell(self, (xIndex, yIndex)) : do state update for o = hit
+ redraw map

ClearCell(self, (xIndex, yIndex)) o = free

occProb(self, (xIndex, yIndex)) : returns prob floating pt

occupied(self, (xIndex, yIndex)) : returns True if occProb
above a certain ~~thresh~~ threshhold

⑪

But first tutor 14.2.3 Aliasing Instances

```
Class MyClass:
    def __init__(self, v):
        self.v = v
```

Part 1

```
def lotsofClasses(n, v):
    One= MyClass(v)
    result = []
    for i in range(n)
        result.append(one)
    return result

Class10 = lotsofClasses(10, 'oh')
Class10[0].v = 'no'
```

- well just one instance of the class
- v = no for all of them ✓

⑫

Part 2: Try 2

Define a new version of lots of Class each w/ its ~~an~~ own instance

```
for i in range(n)
    result.append(MyClass(v))
```

✓ 1st try

Part 3: Try 3

Define another version of lotsOfClass that has seperate instance of the objects in each location of list - but this time using util.makeVectorFill

```
return util.makeVectorFill(n, MyClass(v))
```

ⓧ myCall has no call method

recall function takes "initFun"

```
return [initFun(i) for i in range(dim)]
```

TA: should be a function lambda x: MyClass(v) ✓

Now back to problem

- where is height, etc?
- should be ~~[illegible]~~ built in
- well called from MapMaker
- opps paper says self. xN

Function eval each time - like recipe

Not ssm as the fill

But a state estimator seFast

- takes a model, ~~a~~ the ssm

? need to implement getnext value?

Where did we use this before?

(all these pieces don't fit together in my head)

in desLab 13 - just called it - no

ⓧ

? Get rid of x so lambda: estimator

ⓧ - still tries to give 2 params

Prof lamda x, y: estimator

⑭

Need to give even though don't care about

Fun = function, not "Fun"

✓ Now its making our array

Need to do update functions now

Now need to feed data in

what is state?

- Isn't it internal
- it should be a ddist

ⓧ Now having colors error

Call step not getNextValue

- -and don't include state
- - duh it tracks

Can't start light

Needed to implement occProb lst

✓ Now it works!

- -Cells turn white as it sees they are free

Is bright green likly to be blocked light?

- - depends on start prob?

Yeah - it makes it less green

⑮ Why does it blank white at the end?

Oh well ignore for now

✓ works w/ testClearData

now try in Soar

half worked Is green - but it lost its willingness to turn

Its plan is just to go straight

It never marks things as hit

ⓧ Never turns red

Had same instance - just always re calling

don't have Variable estimator - call it in directly

✓ Now cobot is working right

Confused prof Freeman

Step 6 (skip ahead)

↓

So far added a heuristic to speed up
now want to speed up more
w/ Replanner + MoveToDynamicPoint modules
w/ MapAndRaceBrain.py + raceWorld

ⓧ Need to copy gridDynamicsSkeleton.py from 13.1.2

✓ Now it works well at basic level
234 41.84 baseline

Some things to improve
- Slows down to hit point exactly
- rotations slow

Can speed up by producing plan w/ less stopping + turning
(I was thinking not having it stop exactly)

Plan w/ original set of actions, but post-process
1. Remove intermediate subgoals
2. Add penalty to turning (hard to get right)
3. Increase set of actions to include moves more than just the one next to

Can also change gains + tolerances in move. Move To Dynamic Point

But don't let it crash!

Can't change ~~the~~ max Vel though!

(Don't have to actually do since class is over)