CS107 Handout 56
Autumn 2000 December 1, 2000

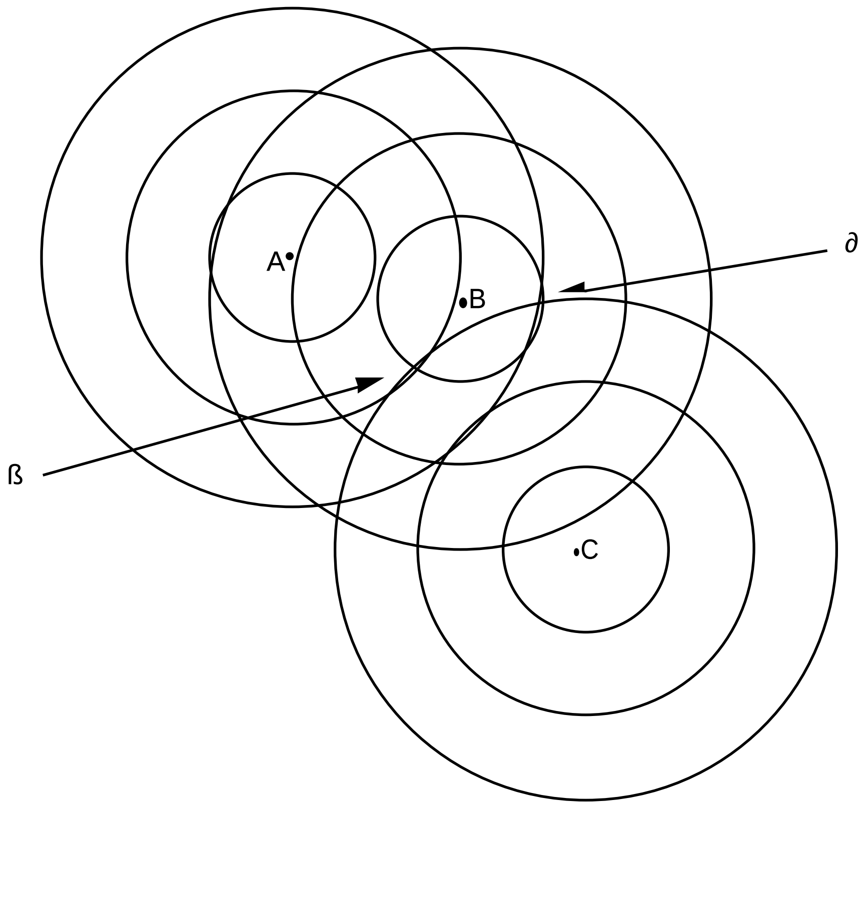
Assignment 5: Where am I?

Assignment written by Nick Parlante and polished off by Jerry.

Here is the problem: You are in a two-dimensional space, and you are lost. Although you don't know your location, you do have a fairly accurate map which indicates where all the stars are in the space. To sort things out, you measure the approximate distance to several of the stars around you. The problem is, using the map and the distance measurements, figure out where you are.

Due: Friday, December 8th at midnight.

Consider the three-star system below. Suppose your observe a star which is one unit away, another star which is two units away, and a third which is three units away. You do not know the identities of the stars observed. Knowing that you are X units away from a star is equivalent to knowing that you are somewhere on a circle of radius X around that star. Below are circles of radius one, two, and three plotted around stars A, B, and C. Your location should be somewhere where three circles intersect. From that point, you must have looked out and saw the stars at their respective distances. ß and ∂ are potential points. However, ∂ cannot be your location— although it is one away from B, unfortunately it is three away from both A and C. ß must be your location. The star that was one away was B, the star that was two away was A, and the star that was three away was C.



This problem demands a sophisticated algorithm. The expressiveness of the Lisp language allows you to concentrate on the ample challenges of the problem. The problem is presented in the form a problem set— you are asked to write many small functions. For the last step you will combine all of the small functions to solve the whole problem. This helps you to acclimate to Lisp style decomposition— many short but conceptually dense functions. For each function, I have given some sample output. You should take advantage of this to test each function when it is written. Lisp is well suited to this sort of piecemeal approach. Lisp code is so dense, that debugging more than a few lines at a time becomes impossible, so you need to deal with things in little steps. I have also provided code to support the types 2-d "point" and "circle". There are routines to compute the distance between two points (easy) and to find the points of intersection of two circles (a pain).

Strategy Overview

What follows is one strategy for solving the problem. It's probably not the best, it's just the first solution I worked out. Feel free to experiment and try something different. Just document what you're doing. You should make extensive use of `mapcar` and `lambda`. As raw material, you have a star map and several distance measurements. Each distance corresponds to some star, but you don't yet know which one. Pairing each distance with a particular star yields a "guess" which may or not be right. Matching a distance with a star defines a circle, so you can think of a guess as a set of circles. So for example given the distances 6 and 7 and stars A, B, and C the 6 possible guesses are:

```
6 from A and 7 from B
6 from A and 7 from C
7 from A and 6 from B
7 from A and 6 from C
6 from B and 7 from C
7 from B and 6 from C
```

One of the guesses should correspond to reality in that it matches the correct distance with the correct star. The correct guess should be distinguished in that all of the circles will more-or-less intersect at a point. "More-or-less" because the distance measurements are approximate, so the circles will not intersect exactly. The circles should all come together around the correct location as at location ß on the previous page. The challenge is to pick the correct guess out from among the false ones. For a false guess, the circles will intersect at points which are spread around the map, rather than clustered about a single point.

Here is the strategy in a nutshell: create a list with all the guesses in it. Each guess is a list of circles. In the correct guess, the intersection points will clump; all the other guesses should have non-clumping intersections. Write functions which operate on a single guess to determine if it is clumped or not. Run these functions over the list of all guesses to pick out the correct one.

The first step is generating the list of all guesses. This is actually an extremely difficult function to write, so I have provided it. Given a list of distances and a list of stars all-guesses returns a list of all the possible guesses. Each guess pairs a distance with one of the stars. Here is an example with the distances `(2 3)` and the stars `(a b)`.

```
? (all-guesses '(2 3) '(a b))
(((2 A) (3 B)) ((2 B) (3 A)))
```

In this care there are two possible guesses: 1) pair 2 with and 3 with b, or 2) pair 2 with b and 3 with a. Each pairing is a list length 2 and each guess is a list of pairings. Here is the example from above with the distances `(6 7)` and the stars `(A B C)`.

```
? (all-guesses '(6 7) '(a b c))
(((6 A) (7 B)) ((6 A) (7 C)) ((6 B) (7 A)) ((6 B) (7 C)) ((6 C) (7 A)) ((6
C) (7 B)))
```

Each star is represented by its coordinate. Suppose the star `a` is at location `x = 3 y = 5` and the star `b` is at location `x = 8.2`, `y = 0`. Then each star would be represented by its coordinate rather than use letters. So instead of `a` use `(3 5)` and instead of `b` use `(8.2 0)`. A circle is defined by a center point and a radius. So each guess is like a list of circles.

```
? (all-guesses '(2 3) '((3 5) (8.2 0)))
(((2 (3 5)) (3 (8.2 0))) ((2 (8.2 0)) (3 (3 5))))
```

The following functions deal with a single guess. Each guess is a list of circles, and the challenge is to determine if the circles all intersect in a clump or not. Consider a particular guess...

1. Write a function `intersection-points` which takes a list of circles and returns a list of all the points where the circles intersect. Use the provided circle function `intersect` which takes two circles and returns a list of the points where the circles intersect. Each circle in the list may intersect with each of the other circles in the list. You can be sure of checking all of the possibilities by intersecting the first circle in the list with all of the ones to the right. Then intersect the second circle with all of the ones to its right and so on. N circles should yield (n^2 - n) points of intersection. The list of points may contain duplicates. Duplicates are good— they will be in effect corroboration about your location.

   ```
   ? (intersection-points '((1 (0 0)) (1 (1 0))))
   ((0.5 0.8660254037844386) (0.5 -0.8660254037844386))
   ? (intersection-points '((1 (0 0)) (1 (1 0)) (1 (1 1))))
   ((0.5 0.8660254037844386) (0.5 -0.8660254037844386)
   (2.7755575615628914E-16 1.0) (1.0 2.7755575615628914E-16)
   (0.1339745962155614 0.5) (1.8660254037844386 0.5))
   ```

   Investigate two LISP built-ins: `append`, and `mapcan`. You should actually try to write this function using two-levels of `car-cdr` recursion, and your first pass will probably rely on that, but it's also good lisp karma to be using the iterator built-ins like `mapcar` and `mapcan`, and to usethe on-the-fly `lambda` expressions whenever possible.

Having gotten all the points of intersection, the problem is— are the points clumped together around a single location, or are the more spread out. If the guess is correct, then the points will have the following property: about half of the points will be clustered together. The other half of the points will be much more spread out. If the guess is incorrect, then there will be no particular clumping of the points. The following functions will determine if the points are clumped or not.

2. Write a function `distance-product` which takes a point and a list of points and returns the product of the distances between that point and all the points in the list. The point itself may be in the list. In that case, it should be removed so that it doesn't force the product to be zero. Use the `dist` function provided as part of the point type. The result is eight in the following example since `(2 0)` is two away from `(0 0)` and four away from `(6 0)`.

   ```
   ? (distance-product '(2 0) '((0 0) (2 0) (6 0)))
   ```

   Rather than using `car-cdr` recursion, use `mapcar` and a `lambda` expressions wrapping around the `dist` function you've been given to compute a list of distances, and the apply the `#'*` object to the resulting list. You'll need to cope with the problem where the first circle may be in the circle list, leaving a `0` where you'd rather have a `1`. You can overcome this problem in a number of ways, and any one of them is more than acceptable.

3. The next step is to rate how far each intersection point is from all the other intersection points. To do this, write a function `rate-points` which takes a list of points and returns a list where each point is annotated to show its distance-product from the other points. So the point `(2 0)` in the above example gets replaced by `(8 (2 0))`. Use `mapcar`, instead of car-cdr recursion, and annotate the point using a lambda expression. This function is way short provided you understand how the list function works.

   ```
   ? (rate-points '((0 0) (2 0) (6 0)))
   ((12 (0 0)) (8 (2 0)) (24 (6 0)))
   ```

4. Write a function `sort-points` which takes a list of rated points, and sorts them in ascending order of rating.

```
? (sort-points '((12 (0 0)) (8 (2 0)) (24 (6 0))))
((8 (2 0)) (12 (0 0)) (24 (6 0)))
? (sort-points (rate-points '((0 0) (2 0) (6 0))))
((8 (2 0)) (12 (0 0)) (24 (6 0)))
```

Lisp has a built in sort function which takes a list and a comparator function. It sorts the list to be consistent with the comparator, so `(sort '(3 2 1 4) #'<)` returns `(1 2 3 4)` while `(sort '(3 2 1 4) #'>)` returns `(4 3 2 1)`. The comparator can be any function of two arguments, including a `lambda`.

5. The points with the small distance ratings tend to be in a clump, while the points with large distance ratings tend to be out by themselves. For a correct guess, half of the points will be clumped and the other half will be spread all around. To isolate the points in the clump, use the above functions to rate and sort the points, and then just take the front half of the list. If the list is of odd length, the extra element should be discarded. Write a function `clumped-points` which takes a list of points, rates them, sorts them, and then returns the half of the points with the smallest ratings. `clumped-points` should return the points without the ratings.

```
? (clumped-points '((0 0) (2 0) (6 0)))
((2 0))
? (clumped-points '((0 0) (2 0) (6 0) (1 0)))
((1 0) (2 0))
```

The unfortunately named function `butlast` takes a list and a number, and returns the front of the list but without the given number of elements at its tail.
So `(butlast '(3 2 1 4) 2)` returns `(3 2)`. `(length list)` returns the length of a list. `(ceiling num)` rounds `num` up to the next integer.

6. The next step is to take the clumped points and average them together. The function `average-point` should take a list of points and averages them all down to a single point. The average point is obtained by averaging all the x values to get an x value and all the y values to get a y value. `average-point` should also include the distance rating indicating how far the average point was from all the points. When the distance rating is small, it will mean that the points were in a clump. You should use `let` to avoid computing the average point twice.

```
? (average-point '((0 0) (2 0) (6 0)))
(160/27 (8/3 0))
? (average-point '((0 0) (2 0) (6 0) (1 0)))
(675/256 (9/4 0))
```

7. Built on all of the functions so far, the function `best-estimate` takes a guess (a list of circles), computes all the points of intersection, winnows those points down to those which are most clumped, and returns their average point.

   ```
   ? (best-estimate '((1 (0 0)) (1 (2 0)) (0.1 (1 0))))
   (5.942527670663184E-4 (1.0016666666666667 0.033291640592396886))
   ```

   The implication: the three circles essentially intersect around the point (1 0)

8. Finally, given a list of distances and a list of star locations, the function `where-am-i` should compute all the possible guesses, use best-estimate to get an answer out of each one, and sort the estimates in increasing order of distance rating. The result is a list of rated points. The first point is where you are, the rest are your other possible locations, in decreasing order of likelihood.

   ```
   ? (where-am-i '(2.5 11.65 7.75) '((0 0) (4 4) (10 0)))
   ((5.164102748844367E-6 (11.481441859657613 2.001220110464802))
   (0.3394092159986836 (-1.8429290506186957 -1.216560811506545))
   (0.6676116235553851 (7.76704142138513 0.4622501635210244))
   (0.7871787994250546 (2.128838984322123 0.5892556509887895))
   (4.398427430402362 (3.9811875000000003 6.126803974552016))
   (45.38616651704703 (4.326820849071189 4.1540809322972985)))
   ?
   ```

   `(11.5 2.0)` is the solution. (Data came from scattering stars randomly on a piece of paper, and then measuring the distances with a ruler to get realistically noisy data) The other solutions have distance terms a several orders of magnitude larger, so they are the wrong guesses. Sometimes there will be more than one reasonable solution given the observed, such as in the following case.

   ```
   ? (where-am-i '(1 2) '((0 0) (3 0)))
   ((1 (1 0)) (1 (2 0)))
   ```

   In this case it's useful that the function returns both the guesses since the best we can deduce is that the solution is either `(1 0)` or equally likely `(2 0)`. Both match the observed with equal perfection. If you like, you can modify `where-am-i` to only return the top 5 or so guesses, as it gets a bit tiresome seeing the scores of wrong guesses that you never care about.

   For you final test, determine your location for the following problem:

   ```
   distances: 2.65 5.55 5.25
   stars: (0 0) (4 6) (10 0) (7 4) (12 5).
   ```

   You can verify your answer with some paper and a ruler. It never hurts to back up your digital computer with an analog one.