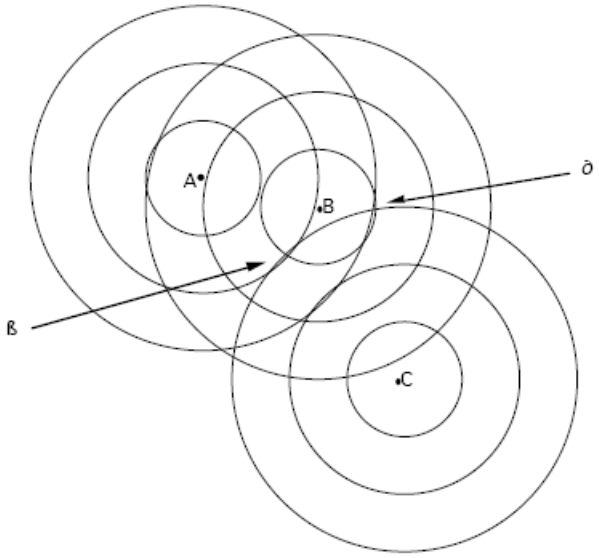
Assignment 2b: Where Am I?*

Due December 7, 11:59 PM

Introduction

Oh no! You are lost in two-dimensional space! But fear not – you have a fairly accurate map that indicates the locations of all the nearby stars. To sort things out, you measure the approximate distance to several of the stars around you. Using the map and the distance measurements, figure out where you are.

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 observed stars, so 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, so 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.



In this assignment, we'll detail one possible algorithm for determining your location, then ask you to implement it using C++ functional programming constructs.

* This is a classic CS107 assignment written by Nick Parlante and Jerry Cain. I've ported the assignment from Scheme into C++, but otherwise the credit for this assignment belongs to them. Sadly, with the curriculum revision under way, CS107 will no longer offer the classic Where Am I? assignment.

Our Strategy

You are given a star map and several distance measurements. Since we have a list of distances to stars but don't know which stars the distances correspond to, we'll simply try each possible combination of distances and stars, effectively "guessing" which stars we are looking at. 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 *-or-* 6 from A and 7 from C *-or-* 7 from A and 6 from B *-or-*
7 from A and 6 from C *-or-* 6 from B and 7 from C *-or-* 7 from B and 6 from C

If you'll notice, if you pair a star (a point in space) with a distance, you end up with a circle in space of all the points equidistant from that star. One of the guesses should be correct, since if we made the measurements reasonably well, there is some combination of stars and distances that represents what we actually measured. What separates the correct guess from all the incorrect ones is that if we've correctly guesses which distances go with which stars, all the circles defined by the guesses should intersect more or less in a single point (if we measured perfectly, the circles would have exactly one common point, but let's assume our instruments have a margin of error). The circles should all come together around the correct location as at location ß on the previous page. The challenge is to determine *which* of these guesses is actually the correct answer. For a false guess, the circles will intersect at points that are spread around the map, rather than clustered about a single point.

Our strategy can be broken down into a few steps. First, come up with a list of all the guesses. Second, write functions to determine whether a single guess is "clumped." Finally, run these functions over all the guesses, then pick out the correct one.

The Starter Code

Unlike the earlier two assignments in CS106L, this assignment has starter files available from the CS106L website. I've taken care of some of the tedious and difficult functions for you, and have set up a nice user input system so that you can concentrate your energy in writing the algorithm rather than doing input error-checking.

The first few steps of the assignment are handled for you in the starter code. The basic program will prompt the user to enter a set of distances, followed by a set of star locations. It then generates all possible guesses based on those inputs and returns them as a `vector<vector<circleT> >`, where `circleT` is simply a point and a radius. After that, the program is yours to create!

Coding Conventions

Unlike the other assignments in this class, when working on this assignment, you must use the functional programming techniques we introduced this week in class. This assignment is designed to get you out of your imperative syntax comfort zone and to immerse you in the wonderful world of functional constructs.

In most of the tasks detailed below, I have explicitly stated that you are not to use loops in your solution. Most of your grade on this assignment will be determined by the extent to which you use the functional programming libraries. If you submit a working solution that does not use functional programming, you will not receive credit on the assignment. Admittedly, this is a rather harsh policy, but hopefully once you've gotten a feel for functional programming, you'll begin using it in some of your future software endeavors.

STL Algorithms Revisited

Now that you're armed with the full power of C++ functors, let's revisit some of the STL algorithms we covered earlier in the quarter and discuss how to maximize their firepower.

The very first algorithm we covered in CS106L was `accumulate`, defined in the `<numeric>` header. If you recall, `accumulate` sums up the elements in a range and returns the result. For example, given a `vector<int>`, the following code returns the sum of all of the `vector`'s elements:

```
accumulate(myVector.begin(), myVector.end(), 0);
```

The first two parameters should be self-explanatory, and the third parameter (zero) represents the initial value of the sum.

However, this view of `accumulate` is surprisingly limited, and to treat `accumulate` as simply a way to sum container elements would be an error. Rather, *`accumulate` is a general-purpose function for transforming a collection of elements into a single value.*

There is a second version of the `accumulate` algorithm that takes a binary function as a fourth parameter. This version of `accumulate` is implemented like this:

```
template<typename InputIterator, typename Type, typename BinaryFn> inline
Type accumulate(InputIterator start, InputIterator stop, Type initial,
                BinaryFn fn)
{
   while(start != stop)
   {
      initial = fn(initial, *start);
      ++start;
   }
   return initial;
}
```

This `accumulate` iterates over the elements of a container, calling the provided binary function on the current value of the variable `initial` (called the *accumulator*) and the next element of the container, then storing the result back in `initial`. In other words, `accumulate` continuously updates the value of the accumulator based on its initial value and the values contained in the input range. Finally, `accumulate` returns the value stored in `initial`. Note that the version of `accumulate` we encountered earlier in the quarter is actually a special case of the above version where the provided callback function computes the sum of its parameters.

To see an example of `accumulate` in action, let's consider an example, Recall that the STL algorithm `lower_bound` returns an iterator to the first element in a range that compares greater than or equal to some value. However, `lower_bound` requires the elements in the iterator range to be in sorted order, so if you have an unsorted `vector`, you cannot use `lower_bound`. Let's write a function `UnsortedLowerBound` that accepts a range of iterators and a lower bound, then returns the value of the lowest element in the range greater than or equal to the lower bound. For simplicity, let's assume we're working with a `vector<int>` so that we don't get bogged down in template syntax, though this approach can easily be generalized.

Although this function can be implemented using loops, we can leverage off of `accumulate` to come up with a considerably more concise solution. Thus, we'll define a functor class to pass to `accumulate`, then write `UnsortedLowerBound` as a wrapper call to `accumulate` with the proper parameters. Consider the following functor:

```
struct LowerBoundHelper
{
    const int lowestValue;
    explicit LowerBoundHelper(int lower) : lowestValue(lower) {}
    int operator() (int bestSoFar, int current)
    {
        if(current >= lowestValue && current < bestSoFar) return current;
        return bestSoFar;
    }
};
```

This functor's constructor accepts the value that we want to lower-bound. Its `operator ()` function accepts two `int`s, the first representing the current lowest value and the second the lowest value greater than `lowestValue` we've encountered so far. If the value of the current element is greater than or equal to the lower bound and also less than the best value so far, `operator ()` returns the value of the current element. Otherwise, it simply returns the best value we've found so far. Thus if we call this functor on every element in the `vector` and keep track of the return value, we should end up with the lowest value in the `vector` greater than or equal to the lower bound. We can now write the `UnsortedLowerBound` function like this:

```
int UnsortedLowerBound(const vector<int> &input, int lowerBound)
{
    return accumulate(input.begin(), input.end(), INT_MAX,
                      LowerBoundHelper(lowerBound));
}
```

Our entire function is simply a wrapped call to `accumulate`, passing a specially-constructed `LowerBoundHelper` object as a parameter. Note that we've specified the starting parameter as `INT_MAX`, a constant defined in the `<climits>` header file that contains the largest possible value of an `int`. That way, if there is a value in the container greater than the lower bound, on some iteration of the `LowerBoundHelper` call, that value will replace `INT_MAX` as the maximum value. Otherwise, if none of the elements are greater than the lower bound, the function will return `INT_MAX` as a sentinel.

If you need to transform a range of values into a single result (of any type you wish), use `accumulate`. To transform a range of values into another range of values, use `transform`. We discussed `transform` briefly earlier in the quarter in the context of `ConvertToUpperCase` and `ConvertToLowerCase`, but such examples are just the tip of the iceberg. `transform` is nothing short of a miracle function, and it arises a whole host of circumstances. You will get some practice with `transform` in this assignment.

Task Breakdown

We've broken the assignment down into eight small steps, each with several tests you can run to make sure your code is working properly. Each individual step should be manageable, and hopefully you can motor through several (or all) of them in a single sitting.

Task One. Initially, we're given a guess as a `vector<circleT>`, but all of our earlier discussion focused more on their points of intersection. Write a function `ComputeAllIntersections` that takes in a `vector<circleT>` and returns a `vector<pointT>` of all the points where the circles intersect. Chances are that each circle in the `vector` will intersect with each other. You can be sure of checking all of the possibilities by intersecting the first circle in the `vector` with all of the ones to the right, then intersecting the second circle with all of the ones to its right, and so on. *n* circles should yield at most (n^2 - *n*) points of intersection, which may contain duplicates. Duplicates are good, actually—they're affirmation that you may be about to find yourself when all is said and done.

For simplicity, we've provided you a `ComputeIntersectionPoints` function which, given two circles, returns a `vector<pointT>` of all the points where those circles intersect. If the two circles do not intersect, `ComputeIntersectionPoints` returns an empty `vector`. `ComputeIntersectionPoints` will cause a runtime error if the two circles are identical.

Although this assignment is designed to let you practice functional programming skills, in this task you are free to use a simple double-for loop. Think of this task as a way to settle into the code base and to get a feel for what we've provided.

To check and see if your code is working, try these tests:

`Input:` [Center: (0, 0), radius 1] and [Center: (1, 0), radius 1]
`Output`[*]**:** **(**0.5, 0.86602), **(**0.5, -0.86602)

`Input:` [Center: (0 0), radius 1], [Center: (1 0), radius 1], and [Center: (1, 1), radius 1]
`Output:` (0.5 0.86602), (0.5 -0.86602), (2.77555E-16 1.0), (1.0 2.7755E-16), (0.13397 0.5), (1.866020.5)

We now know where the circles generated by this guess will intersect, but we have no way of knowing whether or not this guess is on the right track. If the guess is correct, then about half of the points will be clustered together and the other half 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.

Task Two. Write a function `GenerateDistanceProduct` that takes a `pointT` and a `vector<pointT>` and returns the product of the distances between that point and all the points in the `vector`. The point itself may be in the list. If so, you should not include the point in the product or you will end up with a distance product of zero. We've provided a `ComputeDistance` function that you can use to calculate the distance between two points, but you'll still need to account for the aforementioned edge case.

Do not use any loops when writing `GenerateDistanceProduct`. This is a great place to practice functional programming with the STL. There are several interesting ways to make this work and you're free to try any one you choose (I recommend `accumulate` and a binary functor – see earlier in the handout for details about how to do this) provided that you leverage off of the STL algorithms and let functional programming do the work for you. Here are some tests you can run to verify that your function works correctly:

---

* Because of rounding errors and precision quality, you might get different output – that's totally fine. If you're within a reasonable margin, you should be okay.

Input: Single point (2, 0) and the points (0, 0), (2, 0), (6, 0)
Output: 8.0

Input: Single point (3, 3) and the points (2, 5), (7, 8), (10, 1), (3, 2)
Output: 104.2353

Task Three. The next step is to rate how far each intersection point is from all the other intersection points. Write a function `RatePoints` which takes a `vector<pointT>` and returns a `vector<pair<double, pointT> >` where the `first` field of the `pair` represents the distance product of the point stored in the `second` field to other points. Throughout the rest of this project, we'll refer to a `pair<double, pointT>` as a "rated point," since it's simply a way of tracking a point and its associated rating. As with `GenerateDistanceProduct`, do not use any loops when writing this function. Using functional programming and algorithms (I recommend `transform`), you can make this function fit in a small space.

To make sure your code is correct, try these tests:

Input: (0, 0), (2, 0), (6, 0)
Output: (0, 0) has rating 12.0, (2, 0) has rating 8.0, (6, 0) has rating 24.0.

Input: (2, 5), (7, 8), (10, 1), (3, 2)
Output: (2, 5) has rating 164.9242, (7, 8) has rating 320.2249, (10, 1) has rating 481.6637, (3, 2) has rating 161.2451.

Task Four. Write a function `SortRatedPoints` which takes a list of rated points, and sorts them in ascending order of rating. Use the `sort` algorithm for this one. Once you're done, confirm that your code works with these tests:

Input: (0, 0) has rating 12.0, (2, 0) has rating 8.0, (6, 0) has rating 24.0.
Output: (2, 0) has rating 8.0, (0, 0) has rating 12.0, (6, 0) has rating 24.0.

Input: (2, 5) has rating 164.9242, (7, 8) has rating 320.2249, (10, 1) has rating 481.6637, (3, 2) has rating 161.2451.
Output: (3, 2) has rating 161.2451, (2, 5) has rating 164.9242, (7, 8) has rating 320.2249, (10, 1) has rating 481.6637.

Task Five. The points with small distance ratings tend to clump, while the points with large distance ratings tend to be outliers. For a correct guess, about half of the points will be clumped and the other half will be distributed randomly. To isolate the points in the clump, use the above functions to rate and sort the points, and then take the front half of the point list, rounded down. Write a function `GetClumpedPoints` which takes a `vector<pointT>` of points, rates them, sorts them, and then returns the half of the points with the smallest ratings. `GetClumpedPoints` should strip the ratings off of the points before returning them. Do not use loops.

Tests for Task Five:

Input: (0, 0), (2, 0), (6, 0)
Output: (2, 0)

Input: (0, 0), (2, 0), (6, 0), (1, 0)
Output: (1, 0), (2, 0)

Task Six. The clumped points are a good indicator of where we actually are, but there's so many of them that they're difficult to manage. In this next step, we'll average all of the points together into a single point that should be roughly in the center. Write a function `GetAveragePoint` that accepts a `vector<pointT>` of points and averages them all down to a single point. To obtain the average value, average together the *x* values to get an *x* value and all the *y* values to get a *y* value. `GetAveragePoint` should return a `pair<double, pointT>` including the distance product of average point to the source points. If you've written your `RatePoints` function well, this step might already have been taken care of. Again, do not use loops. This is an excellent spot to show off your skills with `accumulate`.

Test your solution with:

Input: (0, 0), (2, 0), (6, 0)
Output: (2.66667, 0) has rating 5.9259

Input: (0, 0), (2, 0), (6, 0), (1, 0), (5, 4), (4, 5)
Output: (3, 1.5) has rating 590.8865

Task Seven. Based on all of the functions so far, write a function `GetBestEstimate` that takes a `vector<circleT>` representing a single guess, then computes all the points of intersection, winnows those points down to those which are most clumped, and returns their average point (a `pair<double, pointT>`) Do not use loops.

Test your code with:

Input: [Center: (0 0), radius 1], [Center: (1 0), radius 1], and [Center: (1, 1), radius 0.1]
Output: (1, 0.995) has rating 0.009975.

Task Eight. Finally, given a `vector<double>` of distances and a `vector<pointT>` of star locations, write a function `WhereAmI` that accepts a `vector<vector<circleT> >` of all possible guesses, uses `GetBestEstimate` to get an answer out of each one, sorts the estimates in increasing order of distance rating, then prints out the list of rated points. The first point is where you are, the rest are your other possible locations, in decreasing order of likelihood. If you like, you can modify `WhereAmI` to only print the top 5 or so guesses. Do not use loops. If your program works, it should pass the following test:

Input: Distances: 2.5, 11.65, 7.75. Stars: (0, 0), (4, 4), (10, 0)
Output: (11.4814, 2.0012) has value 5.1641E-6.
(-1.8429, -1.2165) has value 0.3394
(2.30937, 0) has value 0.916787
(7.69063, 0) has value 0.916787
(5.36328, 5.36328) has value 2.53293
(3.89543, 4.06971) has value 6.23421

Advice, Tips, and Tricks

This programming project will be very different from the ones you're used to. Functional programming is a completely different way of thinking and can take time to adjust to. Here are some general tips and tricks to be aware of when working with functional programming:

- *Understand* `transform` *and* `accumulate`. `transform` and `accumulate` are two of the most powerful STL algorithms when combined with functional programming, but they can take some time to adjust to. Use `transform` to transform a range of elements into a new range of elements – for example, when marking each point with its associated distance product. `accumulate`, on the other hand, is best suited for computing a single value based on the contents of a container. Remember that you can provide a functor to `accumulate` to do things beyond regular addition.
- *Don't panic when you see compiler errors*. You will be pushing the STL to its limits and will almost certainly run into some absolutely ferocious compiler errors, most of which stem from simple type mismatches. If you hit a compiler error, be sure to fix the topmost error before moving downward, or you might end up trying to correct totally valid code. Reread your code slowly to make sure all of your parameters are correct and that your types match. As always, if you ever get stuck, send me an email and I'll be glad help out.
- *Keep a C++ reference handy*. Know where to look to get more information about the STL algorithms and container classes. You'll be using them an awful lot this assignment, and if there's an algorithm that perfectly solves your problem, it helps to know where to go to find it. Handout #12 on STL algorithms will be quite valuable, as will Handouts #20 and #21 on functors and functional programming. Also, don't hesitate to use online resources for help, and as always, feel free to email me with any questions you might have.
- *Don't worry about edge cases*. In this assignment, there are several edge cases to consider – for example, the user might enter distances and star configurations that cannot possibly work. For this assignment, don't worry about such inputs. It should be enough work learning functional programming!
- *Ask for help when you need it!* This is probably the most conceptually difficult assignment of the quarter, so if you get stuck or just need more time to finish, please send me an email and I'll be glad to help out.

Deliverables

Please email any files you have updated or modified to htiek@stanford.edu. Be sure to include your name at the top of any files you submit. Then pat yourself on the back – you're now a veteran of C++ functional programming and have completed the last assignment of the quarter!