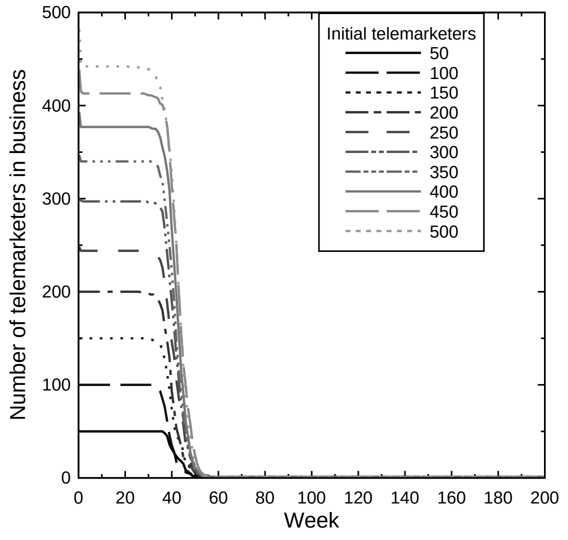
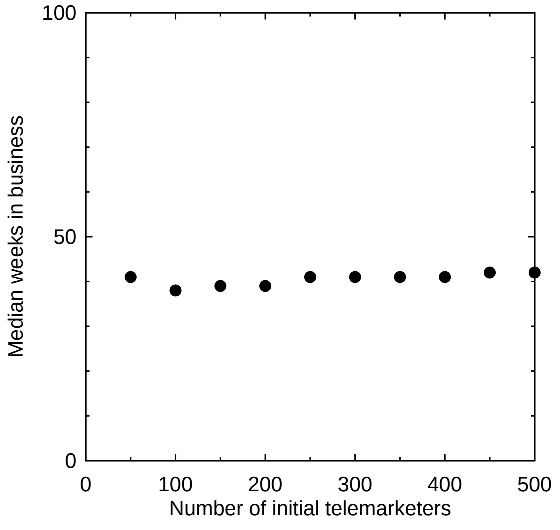
Chapter 14 Exercise Hints and Solutions

Agent-based and Individual-Based Modeling: *A Practical Introduction, 2nd Edition*

Exercise 1

Here are example results with telemarketers doing sales in descending size order, using the code at the top of page 187. These results are qualitatively different than with randomized order: no matter how many telemarketers start, only one is left by week 59. There is no longer any relationship between the initial number of telemarketers and how many remain in business over time.





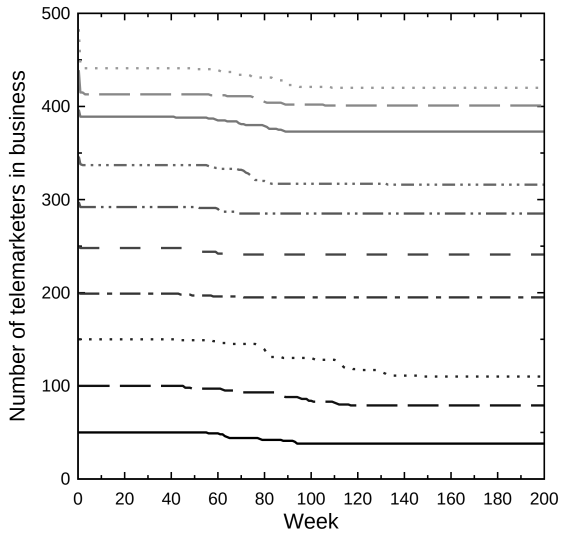
Exercise 2

The results provided as instructor materials for exercise 5 of Chapter 13 used asynchronous updating: as soon as a telemarketer determines that it would be out of money it tries to merge with another. (Some students might have programmed that exercise using synchronous updating. In any case, the two approaches are illustrated here and in the solutions for Chapter 13.)

The synchronous updating approach has all the telemarketers do their weekly accounting, but not form mergers until all have finished their accounting. Then, in a separate action scheduled in the `go` procedure, all the telemarketers that (a) have negative bank balances, and (b) have not already merged, attempt to merge with a larger telemarketer.

An implementation of synchronous updating is provided: `Telemarketer-Mergers-Synch_Ch14-Ex2_2ndEd.nlogo`. This change in scheduling produced quite different results, with far fewer telemarketers going out of business (compare the following graph with the results of Exercise 5 of Chapter 13). The median time in business is the full 200 weeks for all initial telemarketer number scenarios.

The difference is because, in the original asynchronous version, a telemarketer could be merged with (made a “parent” company) before it did its weekly accounting and, hence, before its bank balance reflected any losses from the current week. With synchronous updating, telemarketers with merge only with others that are in good shape *after* the current week’s gains or losses are accounted for. Hence, telemarketers are no longer likely to merge with others that are actually losing money.



Exercise 3

It should be quite clear just from the interface that `ask-concurrent` has very strong and strange effects on model results, though the results can depend on exactly how the sales procedure is programmed. In the version provided as instructor materials for Chapter 13, replacing `ask` with `ask-concurrent` in the `do-sales` procedure causes all the telemarketers grow quite rapidly, with none going out of business for many ticks. However, execution becomes extremely slow. Most noticeable is that the number of sales increases exponentially over time (for a while, then linearly), to levels far exceeding the mathematically feasible amount (which the number of patches in the model × the number of ticks executed × 2, because each sale produces two units of income). This result is presumably because more than one telemarketer can call the same customer and claim a sale before the customer can change its color to indicate that it no longer accepts calls.

Exercise 4

An implementation of the Mousetrap model of Section 14.2.5 is provided (`Mousetrap_Ch14-Ex4.nlogo`). It includes the plot requested in this exercise.

The time each ball is in the air is modeled using a random floating-point number to make sure that each trap has a *unique* trigger time. If trigger times were discrete (e.g., only on ticks representing 0.1 second) then the code would have to be capable of dealing with multiple traps that trigger at once. (The code actually could be made to deal with this situation quite easily.)

Exercise 5

Using if `pcolor = yellow` (the initial color of untriggered mousetraps) does not work because traps remain yellow in the time between when their snap has been scheduled and when the snap is actually executed. This change would re-set a trap's trigger time to a later time each time a ball lands on it, until its snap is finally executed. The effects are not obvious just from watching the View, but if students analyze the mean trigger time of the triggered mousetraps they should see differences that would be significant with standard statistical tests.

Exercise 6

Students might think of using links to connect a patch and the patch it triggers, but links only connect turtles, not patches. Instead, turtles can be used to represent the balls: create two turtles on each patch, and when their patch is triggered, have them move to the patch that is triggered next. Use `pen-down` to draw their path.

Exercise 7

The NetLogo library's Mousetrap model is under the Mathematics section of the Sample Models category. Key ways that it differs from our model include:

- Time is discrete: the time it takes a ball to move to the next trap is always 1 tick instead of depending on the distance. This is the main difference.
- Ball movement is slightly different: instead of picking a patch from those within the maximum travel distance, balls are represented as turtles that actually move; and balls stop when they hit the edge of the space. The different ball movement may seem minor, but it means that in the NetLogo library model balls are more likely to land nearer their starting point. (Our version randomly draws a patch from those within a radius; most of those patches are farther away than half the radius. If you enter `show mean [distance myself] of patches in-radius 10` in a patch Agent Monitor, the answer is 6.7.)
- (The library model assumes there is only one ball per trap, which seems like a major difference. But it assumes that the ball landing on the trap also flies when the trap is triggered; so both models assume that a trap throws two balls.)

Students may notice such differences in results as the NetLogo library version taking more ticks to finish, but overall results are qualitatively similar. The most prominent difference is that the library version with discrete time steps executes far more quickly. This is in fact one reason that discrete time steps are so popular: they simplify the model by assuming things happen only at a few discrete times, and simpler models usually execute faster.

Exercise 8

Modifying the Telemarketer model so it better represents how real telemarketers would all call customers at the same time is not completely trivial. First, astute students should understand that we cannot program the model so multiple calls are underway at the same time; Exercise 3 attempts to do this and fails because we cannot keep multiple telemarketers from selling to the same customer at the same time.

We must still schedule one call at a time, but we can make the telemarketers take turns making individual calls. This will require changing the `go` and `do-sales` procedures so that: (a) telemarketers keep track of how many calls they have made and when they can make no more, (b) each telemarketer makes only one call at a time instead of processing all their calls at once, and (c) there is some algorithm for deciding which telemarketer makes the next call.

The algorithm for deciding which telemarketer makes the next call is the challenge. A simple approach is to select a telemarketer randomly (e.g., `ask one-of turtles [make-a-call]`). However, this approach assumes all telemarketers make calls at the same rate (because they have equal probability of making the next call), whereas larger telemarketer companies likely have more callers and phone lines and can make calls at a higher rate. This simple approach would therefore be biased against the larger telemarketers: the smaller ones would finish all their calls earlier, and leave the larger telemarketers to finish their calls late in the time step when many potential customers have already been called.

This bias could be removed by weighting the random selection of the next telemarketer call by telemarketer size, so bigger telemarketers are more likely to make the next call. There is no simple way to do this weighting in NetLogo (although it is a frequent topic on the user forum).

However, there is at least one relatively easy way to make the telemarketers take turns making calls while giving larger telemarketers more turns:

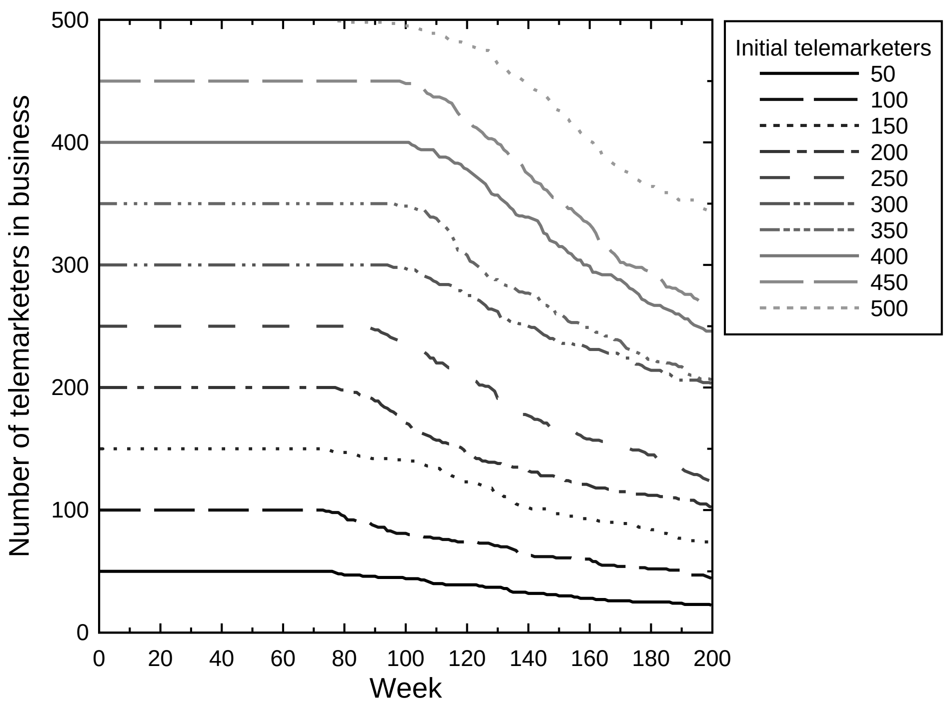
- Create a list (the “call list”) that will contain, each time step, one item for each call that will be made that day.
- At the start of the time step, have each telemarketer calculate the number of calls it can make (its size × 100). The telemarketer puts one copy of itself on the call list for each call it can make: if the telemarketer can make 400 calls, it adds itself to the list 400 times.
- Randomly shuffle the list (see the list primitive `shuffle`).
- Have the go procedure go through the call list and ask each turtle on the list to make a call:

```
foreach call-list
  [
    next-caller -> ask next-caller [ make-a-call ]
  ]
```

(The `foreach` primitive was introduced in Chapter 14.)

An implementation of this approach is provided with the instructor materials: Telemarketer_Ch14-Ex8_2ndEd.nlogo. One noticeable (and predictable) difference is that this version of the model executes very slowly. The results, shown below for comparison to Figure

13.3, indicate that telemarketers stay in business much longer than in the original version of the model. This difference suggests that the telemarketer failures in the original version could often result from a single unlucky time step in which the telemarketer executed its calls after most of the other telemarketers.


Number of telemarketers in business
Week
Initial telemarketers