Introduction to practical disease modeling: Exercise 1

Carsten Kirkeby

Exercises with herd dynamics

---

1. Introducing herd dynamics

We start by constructing a simple simulation model for a dairy farm, and use an age counter (days) for the cows.

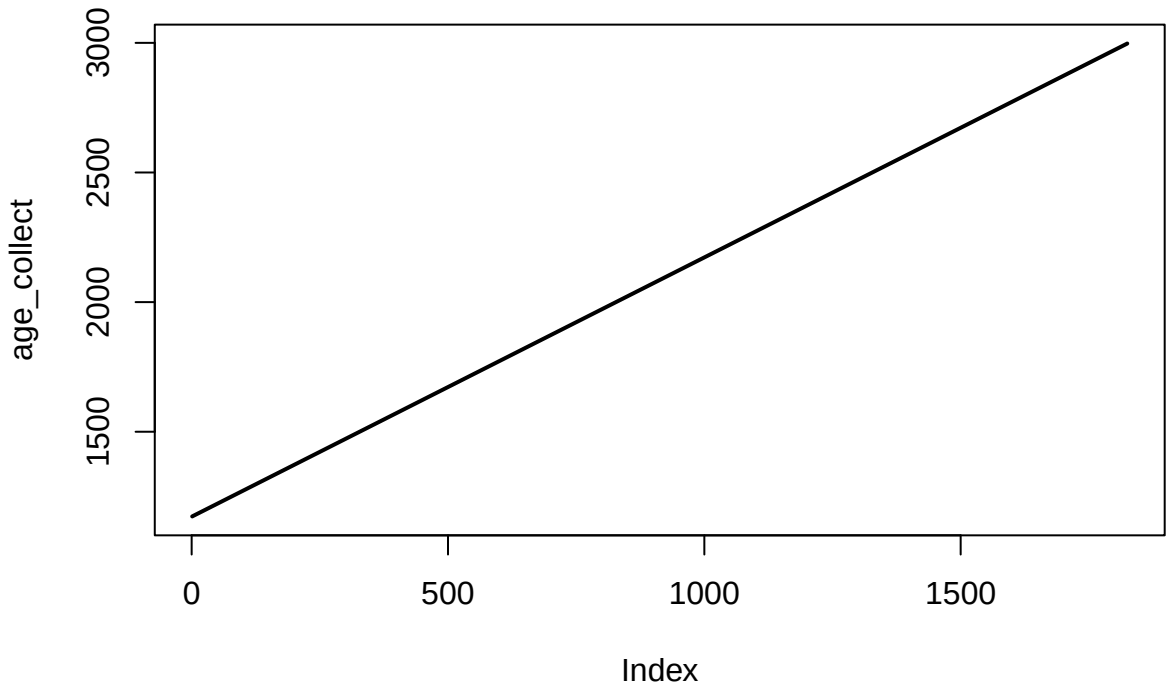
Please look at the code and make sure you understand what each line does.

First we construct a herd with cows that age over time:

```
set.seed(250)

n_cows <- 100

# Create the farm:
farm <- data.frame(id =1:n_cows,
                   age=round(runif(n_cows,730,1642)))

# We want to simulate 5 years:
end_time <- 5 * 365

# Collect the mean age of the cows in herd over time:
age_collect <- numeric(end_time)

for (t in seq_len(end_time))
{

   # Add one day to the age of all the animals, for each simulated:
   farm$age <- farm$age + 1

   # Save the daily mean age of all cows:
   age_collect[t] <- mean(farm$age)
}
```

We can then plot the age of the cows over time:

```r
plot(age_collect, type="l", lwd=2)
```

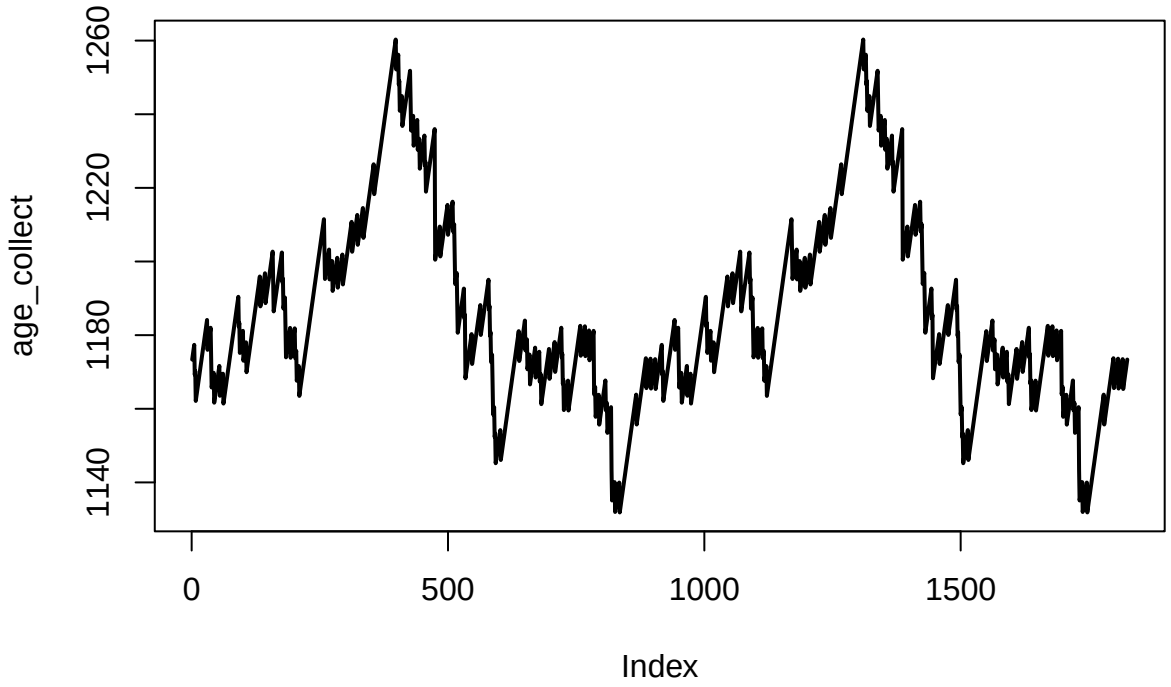


We see that the cows age over time as the mean age in the herd increases. However, this does not reflect a real herd. We want to model a cattle farm where the old cows are replaced with new cows over time. When the cows reach 1642 days, they will be replaced with a new cow that is two years old:

```r
set.seed(250)

n_cows <- 100

# Create the farm:
farm <- data.frame(id =1:n_cows,
                   age=round(runif(n_cows,730,1642)))

# We want to simulate 5 years:
end_time <- 5 * 365

# Collect the mean age of the cows in herd over time:
age_collect <- numeric(end_time)

for (t in seq_len(end_time))
{
   # If cows reach the age of 1642, they are replaced with a new cow that is 2 years old in DIM = 1:
   farm$age[farm$age>=1642] <- 730

   # Add one day to the age of all the animals, for each simulated:
   farm$age <- farm$age + 1

   # Save the daily mean age of all cows:
   age_collect[t] <- mean(farm$age)
}
```

We can then plot the age of the cows over time:

```
plot(age_collect, type="l", lwd=2)
```



The mean age in the herd oscillates over time, as a natural cause of the dynamics in the herd when cows age and are replaced with new cows.

We can look at the variation in the age over time:

```
quantile(age_collect)
```

```
##       0%      25%      50%      75%     100%
## 1131.85 1169.13 1179.41 1201.37 1260.29
```

2. Exercises

Hint: Modify the above code to solve the exercises.

A.

What happens with the variation if you increase the number of cows to 500? And 5000?

B.

Try to change the age before replacement up and down. What happens with the variation in the results? Look at the mean age of the cows over time. Can you identify a repeating pattern (periodicity) in the results? Where does that come from?

C.

Now introduce dynamics in the replacement age: Make a new column with an individual replacement age for each cow (normal distribution with mean = 1642 and sd=50). When the cows reach their own predefined replacement age, then replace them. Remember that the new cows should have their own replacement age. What happens? Look at the periodicity now. What happened?

D.

What modification would you like to introduce to this model, to explore the dynamics in the herd?

E.

Try to run the initial model twice with the same seed and compare results. Then change the seed and compare again. What happens?

---

3. Bonus exercise

If you have finished the exercises above, then consider the following bonus programming exercise (but don't worry if you don't have time to do this - it is optional!).

There are three approaches to computer programming:

1. Procedural programming. This is the standard way in which we tend to code disease models, as it is easiest to get used to for simple models, and is the approach used in the exercises above. However, it is not very expandable so is not necessarily a good approach for more complex models.
2. Functional programming. This lends itself very well to certain tasks such as data analysis using dplyr/tidyverse style code. However, it is not very well suited to writing disease models.
3. Object-oriented programming. This is a more modern style of programming that is used for most commercial programming applications. The central idea of object-oriented programming is that we encapsulate the implementation of a (potentially large and complex) system within an object so that the inner workings of this object are hidden. Higher-level usage of the object occurs via methods that are also defined within the object. The A big advantage of this approach is that changes can be made to the implementation of the object without breaking other parts of the code that use the object. This is particularly relevant when dealing with larger projects with multiple complex parts, but is also useful for smaller collaborative projects. A second advantage is that it becomes much easier to replace small parts of your R code with faster code written in C++ that has the same methods. The downside is that the code takes a little bit of getting used to, as the idea of an object with a persistent state is quite different to what we typically do with procedural programming.

The best way to illustrate this is with an example. The code below implements a "farm" object that encapsulates the same model as we had before. We use the R6 package to set up an object that has a persistent state (i.e. remembers things about itself), accessor methods (so that we can interact with the system externally), and implementation methods that update the internal state.

```
library("R6")

## Define the farm class:
Farm <- R6Class("Farm",

  # Private objects that we can only see within the class:
  private = list(

    # The number of cows:
    n_cows = 0L,

    # The data frame of cows:
    cows = data.frame(id = numeric(0),
                      age = numeric(0)),
```

```
    # The current time:
    time = 0L

  ),

  # The public methods:
    public = list(

      # An initialize method is mandatory:
        initialize = function(n_cows) {

          # Set the number of cows:
          private$n_cows <- n_cows

          # Set the data frame of cows:
          private$cows <- data.frame(id = 1:n_cows,
                                     age = round(runif(n_cows,730,1642)))

          # Set the time:
          private$time <- 1L

        },

        # An update method for a single time step:
        update = function() {

          private$cows$age <- private$cows$age + 1L
      private$time <- private$time + 1L

        },

        # A method to get the current cows:
        get_cows = function() {

          return(private$cows)

        },

        # A method to get the current average age:
        get_mean_age = function() {

          mean_age <- mean(private$cows$age)
          return(mean_age)

        }

    )
)
```

Running this code doesn't actually do anything other than define the class. To run it we need to instantiate an object of this class, then do things with it:

```
# This calls the initialize method with the arguments supplied:
farm <- Farm$new(n_cows = 100L)
```

```r
# Now we can call the other methods like so:
farm$get_cows()
```

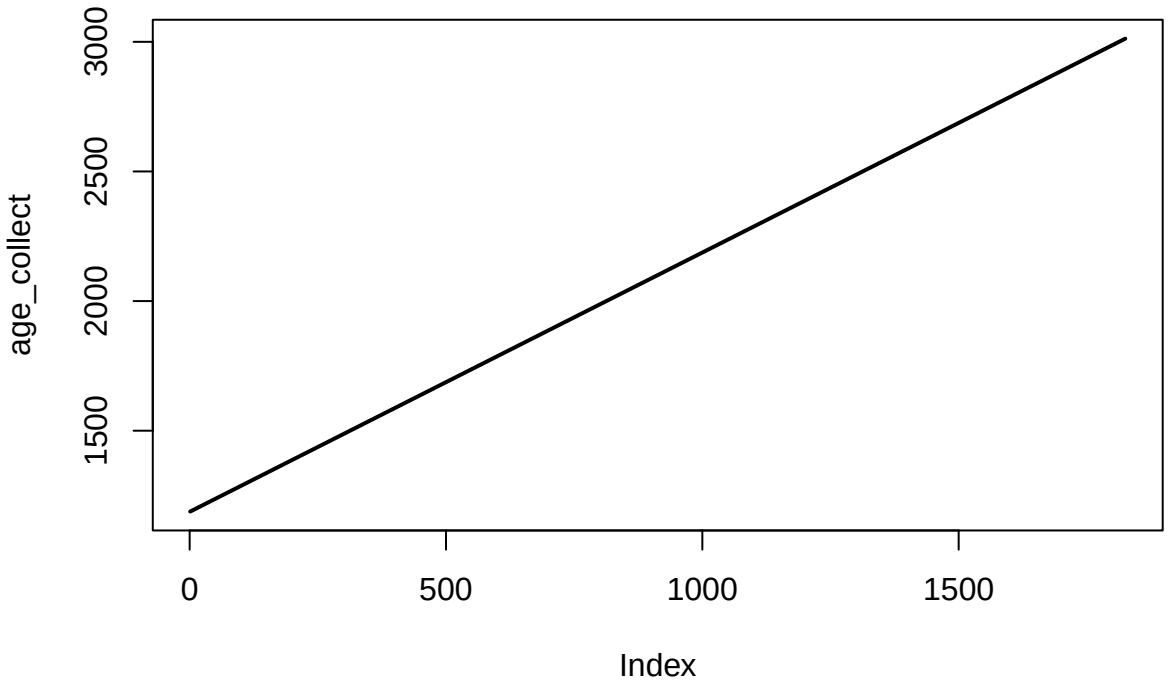
```
##       id  age
## 1      1 1171
## 2      2 1429
## 3      3  892
## 4      4 1021
## 5      5 1437
## 6      6 1047
## 7      7  796
## 8      8 1036
## 9      9 1163
## 10    10  749
## 11    11 1069
## 12    12  913
## 13    13 1415
## 14    14 1458
## 15    15 1058
## 16    16  817
## 17    17 1377
## 18    18  765
## 19    19  847
## 20    20 1225
## 21    21  843
## 22    22 1325
## 23    23 1233
## 24    24  836
## 25    25 1042
## 26    26 1061
## 27    27 1584
## 28    28 1408
## 29    29 1366
## 30    30 1581
## 31    31 1394
## 32    32 1624
## 33    33 1498
## 34    34  965
## 35    35 1034
## 36    36 1048
## 37    37  965
## 38    38 1309
## 39    39 1488
## 40    40 1322
## 41    41 1262
## 42    42 1013
## 43    43  745
## 44    44 1348
## 45    45 1313
## 46    46  858
## 47    47 1267
## 48    48 1591
## 49    49 1559
```

```
## 50   50 1469
## 51   51 1416
## 52   52  835
## 53   53 1443
## 54   54 1280
## 55   55 1613
## 56   56  815
## 57   57 1540
## 58   58 1438
## 59   59  797
## 60   60 1382
## 61   61 1076
## 62   62 1013
## 63   63 1452
## 64   64  780
## 65   65 1116
## 66   66 1202
## 67   67  820
## 68   68 1387
## 69   69 1385
## 70   70  966
## 71   71 1119
## 72   72  753
## 73   73  779
## 74   74 1216
## 75   75  779
## 76   76 1195
## 77   77 1571
## 78   78  984
## 79   79  833
## 80   80 1232
## 81   81  897
## 82   82 1617
## 83   83 1413
## 84   84 1101
## 85   85 1313
## 86   86 1477
## 87   87  778
## 88   88 1228
## 89   89 1448
## 90   90 1069
## 91   91 1312
## 92   92  857
## 93   93 1171
## 94   94 1634
## 95   95 1584
## 96   96 1192
## 97   97 1377
## 98   98 1262
## 99   99 1512
## 100 100 1222
```

```
farm$get_mean_age()
```

```
## [1] 1187.17
```

```r
# To run it we do something similar to before except that the implementation is now hidden:

end_time <- 5 * 365
age_collect <- numeric(end_time)

for (t in seq_len(end_time)) {

   # Update the object:
   farm$update()

   # Save the daily mean age of all cows:
   age_collect[t] <- farm$get_mean_age()
}

plot(age_collect, type="l", lwd=2)
```



Bonus question A:

Copy/paste the R6 model above, and add the same change to the replacement age as we did before.

Bonus question B:

What changes do you need to make to the code that runs the model?

Bonus question C:

Are there any further modifications that you could make to avoid hard-coding the replacement age?

For more information on using R6 classes, see: https://r6.r-lib.org/articles/Introduction.html

You should also bear in mind that R6 is not the only way to implement this kind of object-oriented programming. In fact, you could replace the R6 model in this example with an equivalent model coded either as a base R "reference class" or a class coded in C++ and interfaced to R using an Rcpp module. The cool

thing is that the code to run the model would be completely the same - so other people using your model in this way need never know that they are actually running C++ code!