KOSTAT-UNFPA Summer Seminar on Population

Workshop 1. Demography in R

Day 3: Writing functions

Instructor: Tim Riffe
tim.riffe@gmail.com

Assistant: Rustam Tursun-Zade
rustam.tursunzade@gmail.com

28 July 2021

Contents

1 Summary

Today we will learn how to make simple functions, and how to combine them into more complex functions. We will use the example of the life table, making each column transformation into a separate function. We will then combine these small functions into a life table function and use it to calculate many life tables.

2 Motivation

Everything in `R` is either a function or can be construed as one, I claim. You can do many things without needing to know how to write functions, but function-writing can enhance your work in many ways. Writing functions means you don't have to type out and re-derive the same code so often. Indeed, once you have a good function that gets a task done, you may never need to think about it much again! Repeated use of a self-written function may even lead to you perfecting it little by little. Code that makes use of custom functions can be shorter and easier to understand, especially if you follow certain standards and rules. This all leads to your code being more robust (sooner or later). Writing functions help you think and design your analyses in a modular way, which helps you deconstruct complex tasks into series of simple tasks. Small functions that do simple things can end up being used in many different places in your code. Further, we can learn to make small functions and how to combine them in a single day. Let's do this!

3 Anatomy of a function

Here's an oldy but goody:

```
hello <- function(your_name){
    paste0("Hello ", your_name, ". Have a nice day!")
}
hello("Tim")
```

```
## [1] "Hello Tim. Have a nice day!"
```

1. `hello` is the name of the new function, which we know because
2. `<- function(){}` is being assigned to it, which creates functions
3. The thing in the parentheses `(your_name)` is the name of an argument to be used
4. The part inside `{ }` is the body of the function, presumably making use of `your_name`.
5. In this particular case we're using the function `paste0()`, which concatenates text, including our argument `your_name`.

Those are the parts of a function. The function can have more arguments, which could be any kind of data of any size and shape, and the stuff that happens inside the function (inside `{ }`) should know what to do with the arguments. It should produce a result, and return it. For our little function above, it returns a character string (the last thing evaluated). In our case it printed to console because we didn't assign it to anything. We can be more explicit about that by telling it what to return:

```r
# Identical in this case
hello <- function(your_name){
    out <- paste0("Hello ", your_name, ". Have a nice day!")
    return(out)
}
hello("Tim")
```

```
## [1] "Hello Tim. Have a nice day!"
```

So here's an outline:

```r
function_name <- function(arguments){
    # do stuff with the arguments to calculate a result
    return(result)
}
```

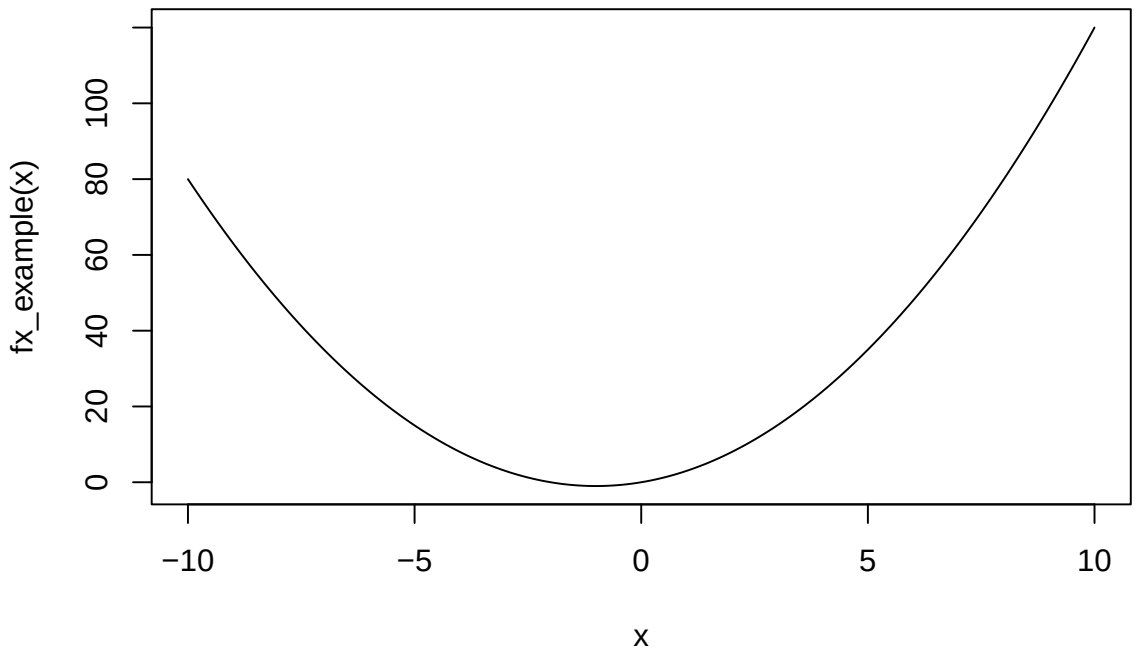
Functions can be very large. Let's look at some:

```r
lm # type into console with no parentheses to see the guts of a function
```

Let's make another simple example, for some practice turning formulas into functions:

$$f(x) = 2x + x^2$$

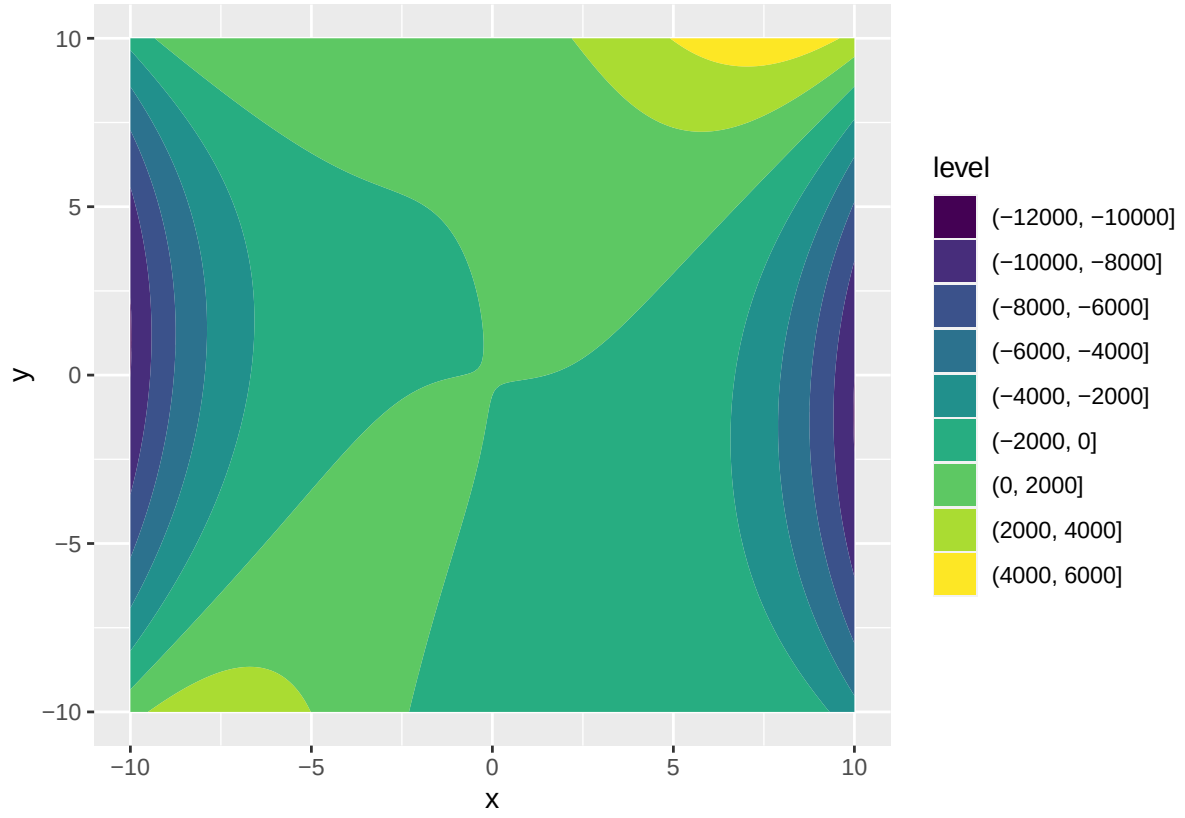
```r
fx_example <- function(x){
  2 * x + x^2
}
# take a look
x <- seq(-10,10,by=.1)
plot(x,fx_example(x),type="l")
```



What about a function of *two* parameters? It really doesn't matter what the operations look like, if you see formulas like these, the functions can be directly coded:

$$f(x, y) = 2 + 2x + 3y + 23xy + y^3 - e^{xy}$$

```
library(tidyverse)
fx_example2 <- function(x,y){
  2 + 2*x + 3*y + 23*x*y + y^3 - x^4 + (x*y)^2
}
y      <- seq(-10, 10, by = .1)

combos <- expand.grid(x = x, y = y)

combos %>%
  mutate(z = fx_example2(x, y)) %>%
  ggplot(aes(x = x, y = y, z = z)) +
  geom_contour_filled()
```



`expand.grid()` creates a `data.frame` consisting in all unique combinations of whatever vectors you give it. `mutate()` creates a column by applying our new function to the data, in our case based on `x` and `y` columns. We then pipe the resulting data to `ggplot()` to view as a filled contour plot, or *surface*.

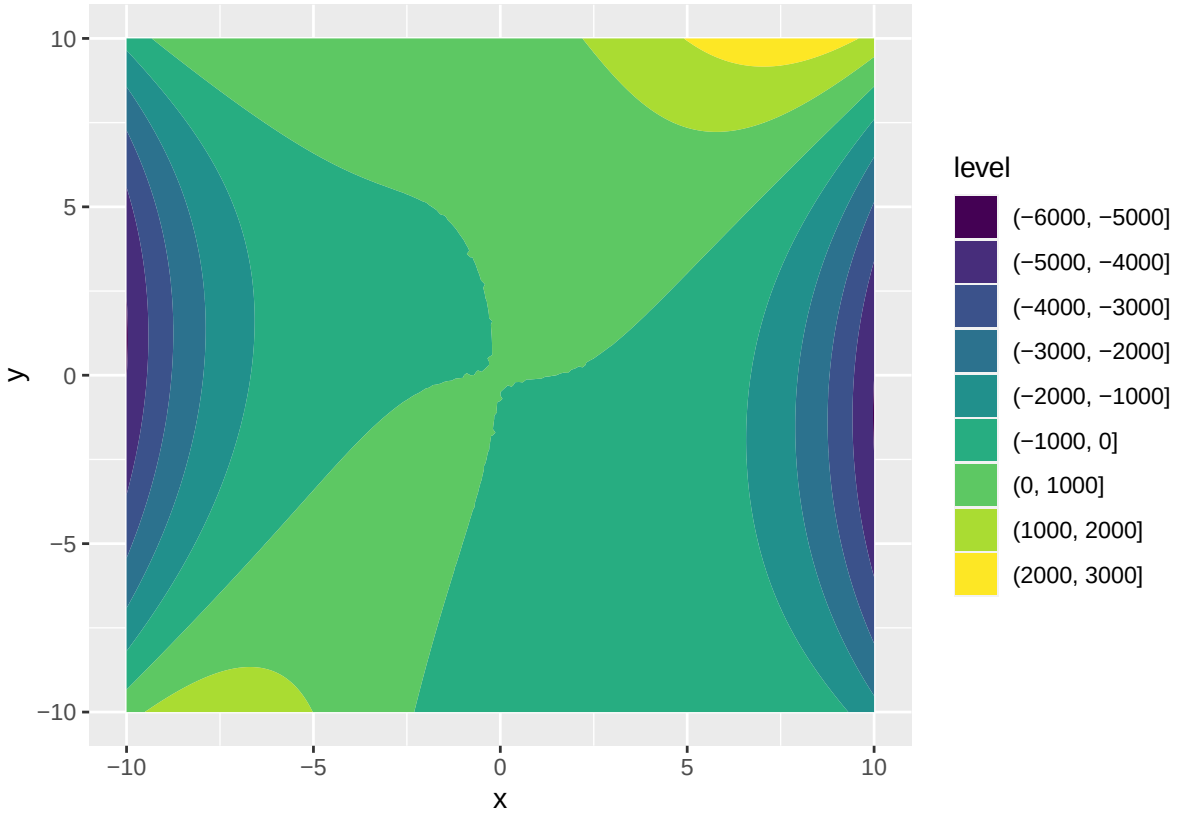
4 *Wrapping* a function

As you can see in our first cheap example, functions can use other functions (`paste0()`), as long they can be found. This makes it easy to recycle your code and be able to do ever more complex

tasks. Function `a` can get used by function `b` can get used by function `c` and so on, many layers deep. This allows complex tasks to be simplified.

For example, we can use these two functions together somehow (totally contrived):

```
fx_example3 <- function(x,y){
  z <- fx_example2(x = x, y = y)
  1 / fx_example(1 / z)
}

combos %>%
  mutate(z = fx_example3(x, y)) %>%
  ggplot(aes(x = x, y = y, z = z)) +
  geom_contour_filled()
```



A couple rules for using functions inside of other functions are that

1. you *should* be explicit about argument names (e.g. `x = x`)
2. `R` should know where to find the functions you're using
3. leave it in a good commented (or better, *documented*) state for your colleagues and for *future-you*.

For (2) the easiest is to just have it loaded in memory, either by having defined it earlier in your script (or a script your `source()`), or by calling `library()` or `require()` to get the requisite package loaded. There are more robust ways of doing this that would require us to learn how to *make* packages, but there's no time for that in our schedule.

Knowing the basics for creating and wrapping functions is all we need to be able to make lifetable functions.

5 Lifetables

5.1 Reading in data from WHO

We will take death rates ${}_nM_x$ and a quantity we write as ${}_nA_x$ (which is interpreted as the mean time spent in an age interval by those that die in an interval) as givens for calculating lifetables. I read in data from the WHO Global Health Observatory (GHO) using an API package called `rgho` (Filipovic-Pierucci (2020)). You can examine my (expedient, not necessarily as efficient as could be) data wrangling code in a separate markdown script that we don't have time to cover in class in a file called `03_data_prep.Rmd` (or `.pdf`). The data is called `LT_inputs.csv`, and we can read it straight into `R` from `github` without needing to manually download anything.

We'll use `read_csv()`, which has nice default settings for reading in delimited files like this.

```
library(tidyverse)
library(readr)
# this just so it fits on the page!
main_repo  <- "https://raw.githubusercontent.com/timriffe/KOSTAT_Workshop1"
extra_path <- "/master/Data/LT_inputs.csv"
path       <- paste0(main_repo, extra_path)
LT         <- read_csv(path)
glimpse(LT)
```

```
## Rows: 13,395
## Columns: 7
## $ Country <chr> "Algeria", "Algeria", "Algeria", "Algeria", "Algeria", "Algeri~
## $ ISO3    <chr> "DZA", "DZA", "DZA", "DZA", "DZA", "DZA", "DZA", "DZA", "DZA",~
## $ Year    <dbl> 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 20~
## $ Sex     <chr> "f", "f", "f", "f", "f", "f", "f", "f", "f", "f", "f", "f", "f~
## $ Age     <dbl> 0, 1, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 7~
## $ nMx     <dbl> 0.03443, 0.00145, 0.00070, 0.00046, 0.00065, 0.00078, 0.00094,~
## $ nAx     <dbl> 0.09535144, 2.80888571, 2.50000000, 2.50000000, 2.50000000, 2.~
```

5.2 Lifetable transformations as functions

5.3 Death probabilities between age x and $x+n$ ${}_nq_x$

The first and key step is to transform a set of age-specific death rates into a set of age-specific probabilities of dying (${}_nq_x$). The relationship between ${}_nM_x$ and ${}_nq_x$ has been established based on analyses of actual cohorts (for mathematical proof, see Preston, Heuveline, and Guillot (2000), p. 42-43).

$$ {}_nq_x = \frac{n * {}_nM_x}{1 + (n - {}_nA_x){}_nM_x} $$

where ${}_nA_x$ is the average number of person-years lived in the interval by those dying in the interval and n is the width of the age-interval.

For single ages or when we're pragmatic, we define ${}_nA_x = n/2$ with the exceptions of the first and the last age group. Other approximations are also available, but these only matter when age groups are wider than a year. In our case, we're working with abridged lifetables, some of which represent high mortality settings, and the ${}_nA_x$ assumptions are consequential. In our case, I provide this value so that we don't need to work so hard at deriving it in class. You could find several popular ${}_nA_x$ approximations in the `DemoTools` package Riffe et al. (2021).

Cutting to the chase, we can rather directly convert the ${}_nq_x$ formula to an `R` function:

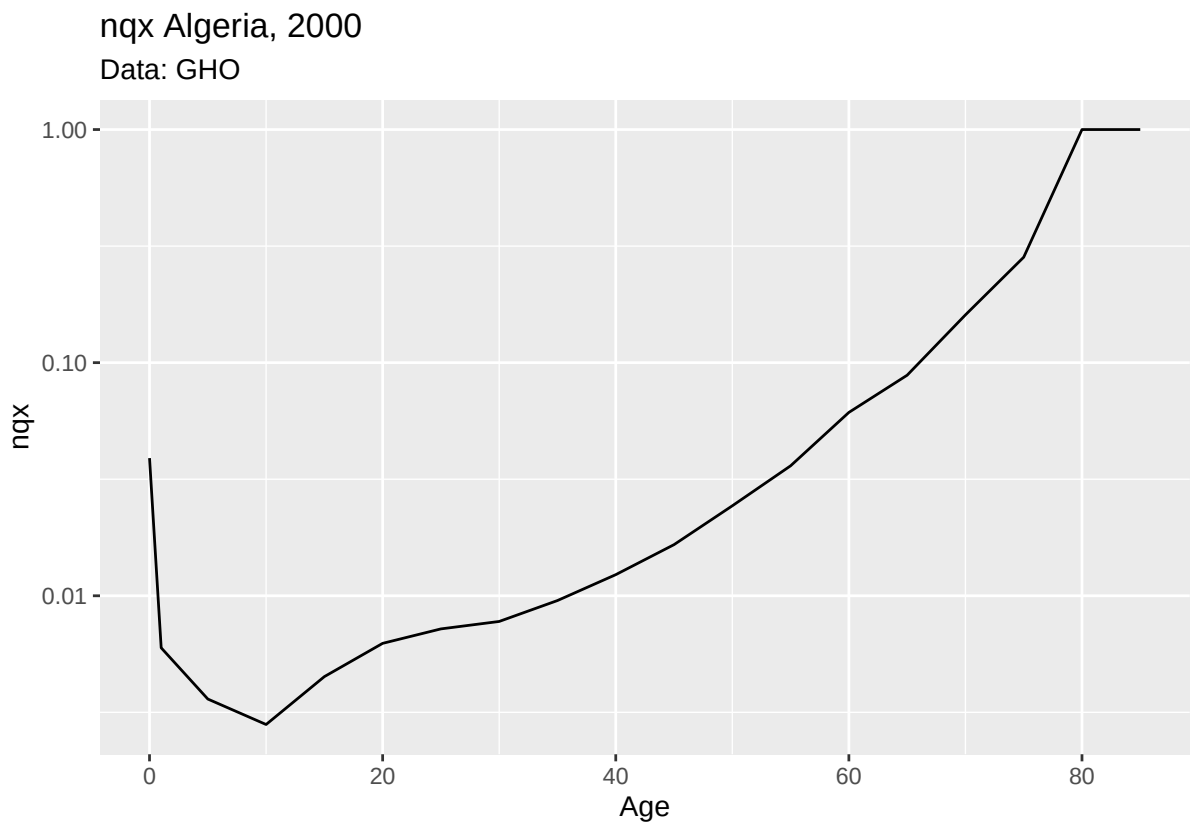
```r
calc_nqx <- function(nMx, nAx, n){
  qx         <- (n * nMx) / (1 - (n - nAx) * nMx)

  # these are kludges, necessary to ensure nqx results as a probability
  qx[qx > 1] <- 1
  qx[qx < 0] <- 0
  qx
}
# example
calc_nqx(.01, .5, 1)
```

```
## [1] 0.01005025
```

```r
calc_nqx(.01, .5, 5)
```

```
## [1] 0.05235602
```

```r
# Example of what nqx looks like over age
LT %>%
  filter(Country == "Algeria",
         Year == 2000,
         Sex == "t") %>%
  mutate(n = c(1,4,rep(5,17)),
         nqx = calc_nqx(nMx, nAx, n)) %>%
  ggplot(aes(x = Age, y = nqx)) +
  geom_line() +
  scale_y_log10() +
  labs(title = "nqx Algeria, 2000",
       subtitle = "Data: GHO")
```



Often we're sure to *close out* the lifetable by making the *final* ${}_nq_x$ value equal to 1. You could optionally modify the function to impute 1 like so `nqx[length(nqx)] <- 1`.

5.4 Survival probabilities between age x and $x+n$, ${}_np_x$

The survival probabilities between age x and $x+n$ (${}_np_x$) is simply one minus ${}_nq_x$. It is interpreted as the chance of surviving from age x to age $x+n$.

$$ {}_np_x = 1 - {}_nq_x $$

Really there's no need to program a function for this column, as we can just use ${}_nq_x$ as the function argument and take its complement as needed.

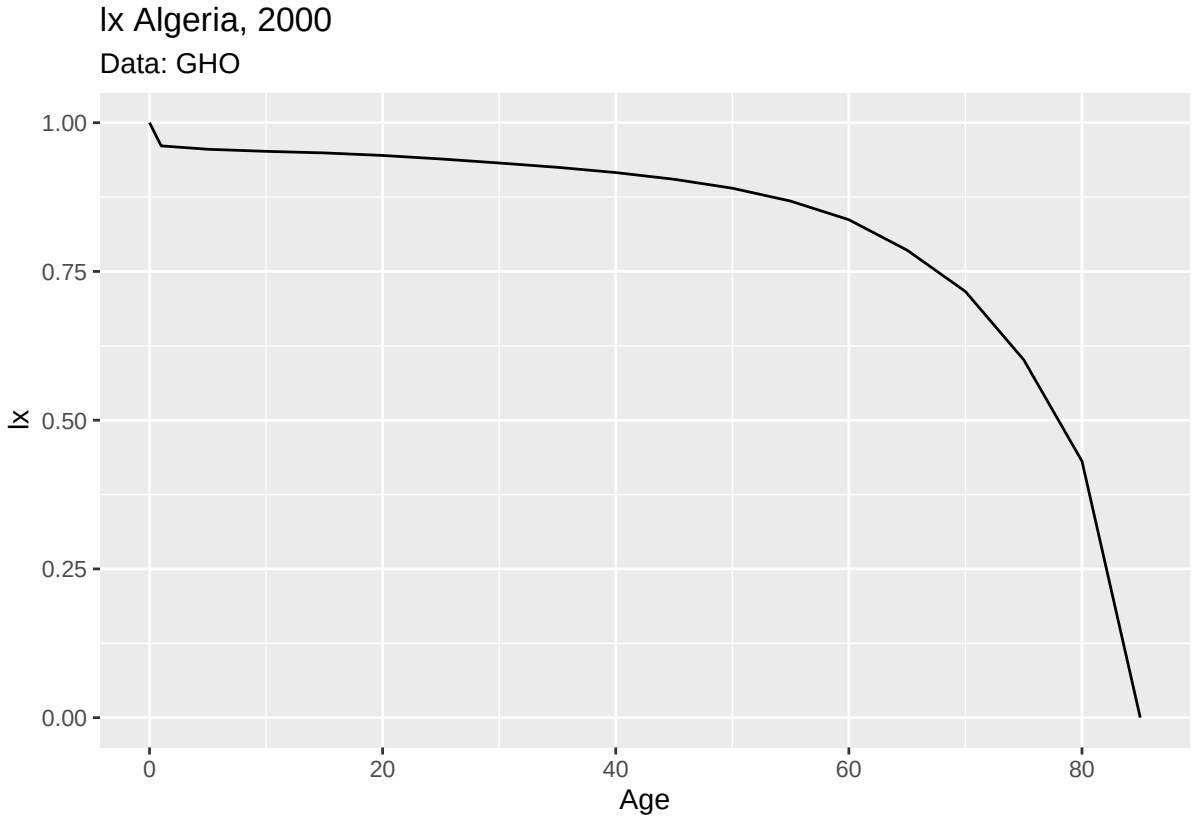
5.5 Survival probabilities to age x, l_x

This indicator indicates the chance of surviving from birth to age x (l_x) OR the number of survivors at age x relative to the radix of the life table. The l_0 is interpreted as the initial size (radix) of the population, generally set to 1 or 100,000. Here's one of several ways to calculate it given what we have so far:

$$ l_{x+n} = r \prod_{y=0}^{x} {}_np_y $$

where $r = {}_nl_0$ is the radix. To program this, our arguments should be ${}_nq_x$ (`nqx`) or ${}_np_x$ (`npx`) and `radix`. In this case, we can assign a default value for the radix when defining the function using `radix = 1`. Whenever the argument isn't specified by the user, 1 will be assumed.

```
# nqx is a full vector over age.
calc_lx <- function(nqx, radix = 1){
  npx <- 1 - nqx
  n   <- length(nqx)
  lx  <- cumprod(npx)
  # shift it back 1, as we start with 100%!
  # also ensure outgoing vector is the same length.
  lx  <- radix * c(1, lx[-n])
  lx
}

LT %>%
  filter(Country == "Algeria",
         Year == 2000,
         Sex == "t") %>%
  mutate(n = c(1,4,rep(5,17)),
         nqx = calc_nqx(nMx, nAx, n),
         lx = calc_lx(nqx = nqx)) %>%
  ggplot(aes(x = Age, y = lx)) +
  geom_line() +
  labs(title = "lx Algeria, 2000",
       subtitle = "Data: GHO")
```



5.6 Death distribution, ${}_nd_x$

The life table deaths (${}_nd_x$) is the number of (synthetic) persons dying between age x and $x+n$, relative to the radix, and represents the distribution of deaths over age. There are two ways of calculating ${}_nd_x$. When programming, this is the most pragmatic way of calculating it:

$$ {}_nd_x = {}_nq_x * l_x $$

One could ask, do we really need this function?

```
calc_ndx <- function(nqx, lx){
   nqx * lx
}
```

5.7 Person-years lived between age x and $x+n$, ${}_nL_x$

The number of person-years between age x and $x+n$ (${}_nL_x$) is calculated as:

$$ \begin{aligned} {}_nL_x &= n(l_x - {}_nd_x) + {}_na_x * {}_nd_x \\ &= n * l_x - (n - {}_n a_x){}_nd_x \end{aligned} $$

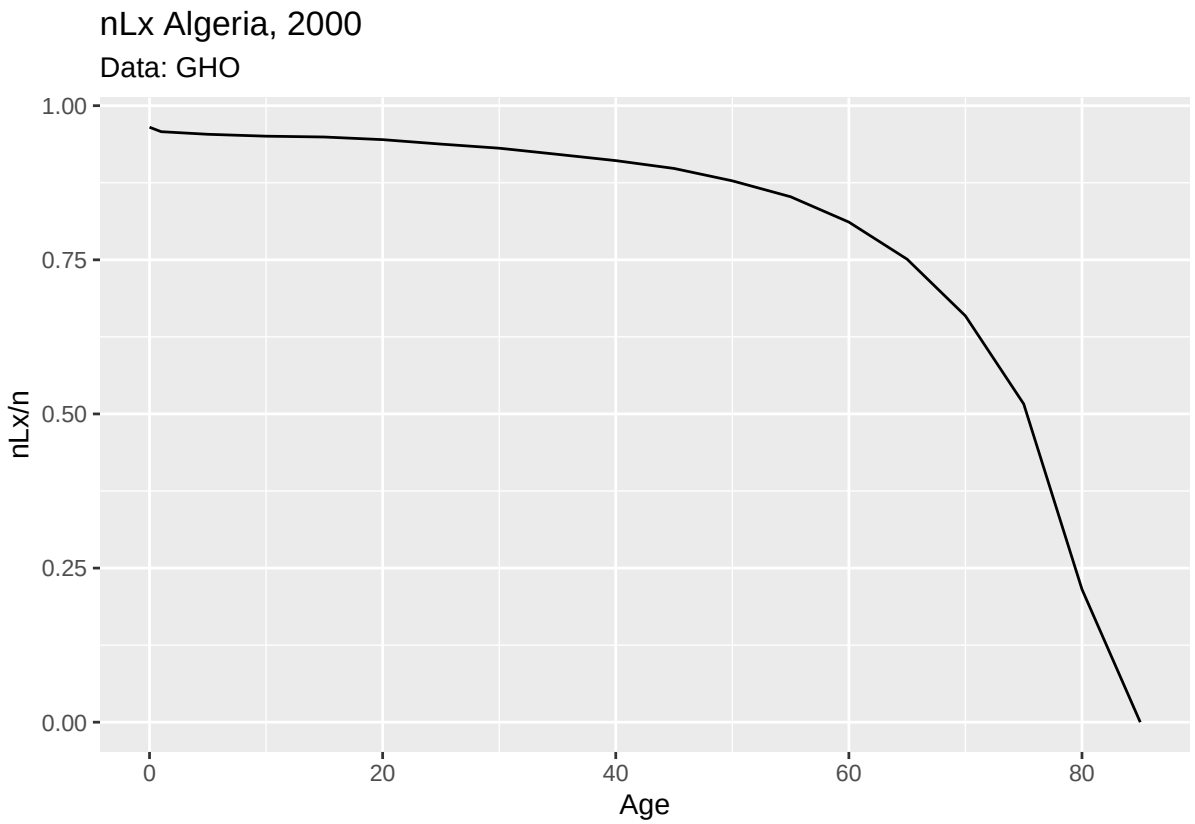
Note

${}_nm_x =_n d_x/{}_nL_x$

and

${}_nq_x =_n d_x/l_x$

```r
calc_nLx <- function(lx, ndx, nAx, n){
  N        <- length(lx)
  nLx      <- n[-N] * lx[-1] + nAx[-N] * ndx[-N]
  # special treatment for open age
  nLx[N]     <- lx[N] * nAx[N]
  nLx
}

LT %>%
  filter(Country == "Algeria",
         Year == 2000,
         Sex == "t") %>%
  mutate(n = c(1,4,rep(5,17)),
         nqx = calc_nqx(nMx, nAx, n),
         lx = calc_lx(nqx = nqx, radix = 1),
         ndx = calc_ndx(nqx = nqx, lx = lx),
         nLx = calc_nLx(lx = lx, ndx = ndx, nAx = nAx, n = n)) %>%
  ggplot(aes(x = Age, y = nLx / n)) +
  geom_line() +
  labs(title = "nLx Algeria, 2000",
       subtitle = "Data: GHO")
```



5.8 Person-years lived above age x T_x

Calculating the number person-years lived above age x (T_x) is a key step to calculate life expectancy. It consists in finding the sum of ${}_nL_x$ from age x:

$$T_x = \sum_{y=x}^{\infty} {}_nL_y$$

```
calc_Tx <- function(nLx){
  # to understand this, look at the nLx curve,
  # then imagine integrating from the right
  # to the left. Then compare with the formula!
  nLx %>% rev() %>% cumsum() %>% rev()
}
```

5.9 Life expectancy e_x

The last indicator in the life table is probably one of the most used in demographic analysis. The life expectancy is the average number of years lived by a (synthetic) cohort reaching age x. It consists in dividing the number of person-years lived above age x by the number of people alive at age x:

$$e_x = \frac{T_x}{l_x}$$

Since `mutate()` let's you make columns in a sequentially dependent way, we can actually do this whole lifetable inside a single `mutate()` statement. However, each combination of `Year` and `Sex` is an independent lifetable, so we need to declare groups beforehand using `group_by()`:

```
calc_ex <- function(Tx, lx){
  Tx / lx
}

LT %>%
  filter(Country == "Algeria",
         Year == 2000,
         Sex == "t") %>%
  mutate(n = c(1,4,rep(5,17)),
         nqx = calc_nqx(nMx, nAx, n),
         lx = calc_lx(nqx = nqx, radix = 1),
         ndx = calc_ndx(nqx = nqx, lx = lx),
         nLx = calc_nLx(lx = lx, ndx = ndx, nAx = nAx, n = n),
         Tx = calc_Tx(nLx = nLx),
         ex = calc_ex(Tx = Tx, lx = lx)) %>%
  filter(Age == 0) %>%
  select(ex)
```

```
## # A tibble: 1 x 1
##      ex
##   <dbl>
## 1  70.2
```

5.10 A Lifetable function

You probably noticed that a whole lifetable operation can fit in a single `mutate()` call! Well, that may be so, but there's still value in creating a wrapper function that does the whole thing:

```r
calc_LT <- function(nMx, nAx, n, radix){
  N <- length(nMx)
  nqx  <- calc_nqx(nMx, nAx, n)
  lx   <- calc_lx(nqx = nqx, radix = 1)
  ndx  <- calc_ndx(nqx = nqx, lx = lx)
  nLx  <- calc_nLx(lx = lx, ndx = ndx, nAx = nAx, n = n)
  Tx   <- calc_Tx(nLx = nLx)
  ex   <- calc_ex(Tx = Tx, lx = lx)
  Age  <- cumsum(c(0,n))[1:N]


  tibble(Age = Age,
         nMx = nMx,
         nAx = nAx,
         nqx = nqx,
         lx = lx,
         ndx = ndx,
         nLx = nLx,
         Tx = Tx,
         ex = ex)
}

DZA <-
  LT %>%
  filter(Country == "Algeria",
         Sex == "t",
         Year == 2000)

calc_LT(DZA$nMx, DZA$nAx, n= c(1,4,rep(5,17)), 1)
```

```
## # A tibble: 19 x 9
##      Age     nMx   nAx     nqx    lx     ndx   nLx    Tx     ex
##    <dbl>   <dbl> <dbl>   <dbl> <dbl>   <dbl> <dbl> <dbl>  <dbl>
##  1     0 0.0376  0.102 0.0390  1     0.0390  0.965 70.2   70.2
##  2     1 0.00149 1.74  0.00598 0.961 0.00575 3.83  69.2   72.0
##  3     5 0.00072 2.5   0.00361 0.955 0.00345 4.77  65.4   68.5
##  4    10 0.00056 2.5   0.00280 0.952 0.00267 4.75  60.6   63.7
##  5    15 0.0009  5     0.0045  0.949 0.00427 4.75  55.9   58.9
##  6    20 0.00125 5     0.00625 0.945 0.00591 4.72  51.1   54.1
##  7    25 0.00144 4.03  0.00721 0.939 0.00677 4.69  46.4   49.4
##  8    30 0.00155 4.17  0.00776 0.932 0.00723 4.66  41.7   44.8
##  9    35 0.0019  2.77  0.00954 0.925 0.00882 4.61  37.1   40.1
## 10    40 0.00245 2.65  0.0123  0.916 0.0113  4.55  32.5   35.4
## 11    45 0.00329 2.77  0.0166  0.905 0.0150  4.49  27.9   30.8
## 12    50 0.0048  2.27  0.0243  0.890 0.0216  4.39  23.4   26.3
## 13    55 0.00709 2.45  0.0361  0.868 0.0313  4.26  19.0   21.9
## 14    60 0.0119  2.49  0.0612  0.837 0.0512  4.06  14.8   17.6
## 15    65 0.0169  2.49  0.0883  0.786 0.0694  3.75  10.7   13.6
## 16    70 0.0297  2.51  0.160   0.716 0.115   3.30   6.95   9.71
## 17    75 0.0496  2.50  0.283   0.601 0.170   2.58   3.66   6.08
## 18    80 0.152   2.50  1       0.431 0.431   1.08   1.08   2.50
## 19    85 0.263   3.73  1       0     0       0      0    NaN
```

5.11 reformulate function to use in a tidy pipeline

To use easily in a tidy pipeline, we'll just want to be sure that the input consists in a whole group (or chunk) of data:

```
# data.frame in, data.frame out!
calc_LT_tidy <- function(data, radix){
  # this is hacky, but works.
  # just pick out the needed vectors from the group of data
  calc_LT(nMx = data$nMx,
          nAx = data$nAx,
          n = data$n,
          radix = radix)
}
```

5.12 Calculate all the lifetables

Here, out lifetable function is a bit different in that it returns a result with several columns, rather than just a single vector. We need a new tidy verb to deal with this situation, since `mutate()` can't handle this. When operating on grouped data, we can do a `data.frame`-in `data.frame`-out version of `mutate()` with `group_modify()`:

```
AFR_LT <-
  LT %>%
  mutate(n = case_when(Age == 0 ~ 1,
                       Age == 1 ~ 4,
                       TRUE ~ 5)) %>%
  # same grouping approach as before
  group_by(Country, Year, Sex) %>%
  # the ~ is important! .x is the code name for the data object
  # passing through the pipeline. data is *our* argument name
  group_modify(~ calc_LT_tidy(data = .x), radix = 1) %>%
  ungroup()
```

Let's sit back for a moment and be proud of having calculated so many lifetables at once. How many are there anyway?

```
LT %>%
  select(Country, Year, Sex) %>%
  # picks out unique combos
  distinct() %>%
  nrow()
```

```
## [1] 705
```

Not bad!

6 Exercises

1. Make your own lifetable function. Keep it somewhere you won't forget. Celebrate with a nice cup of tea. Or some other beverage. (remember this one from class isn't rigorous, you may need to go back and modify it from time to time as issues arise)
2. Pick a country / year combination used here, and compare the life expectancy estimate with some others you can find. World Population Prospects, for example, or perhaps an

official or otherwise published estimate. If you find a discrepancy, do you know what's behind it?

3. What is the range of life expectancy in our estimates?

References

Filipovic-Pierucci, Antoine. 2020. *Rgho: Access WHO Global Health Observatory Data from r*. https://CRAN.R-project.org/package=rgho.

Preston, S, Patrick Heuveline, and Michael Guillot. 2000. "Demography: Measuring and Modeling Population Processes. 2001." *Malden, MA: Blackwell Publishers*.

Riffe, Tim, José Manuel Aburto, Ilya Kashnitsky, Monica Alexander, Marius D. Pascariu, Sara Hertog, and Sean Fennell. 2021. *DemoTools: Standardize, Evaluate, and Adjust Demographic Data*.