intro R

Juan Manuel Morales

2015-09-07

R de reproducibilidad

Hoy en día, los científicos deben estar capacitados para "sacarle el jugo" a distintas fuentes de información. Parte del trabajo consiste en procesar bases de datos, resumir, graficar, y ajustar modelos a datos. Trabajar con herramientas tipo Excel no solo es limitante (y poco elegante) sino que además atenta contra la "reproducibilidad" de la investigación ya que normalmente no nos queda una documentación detallada de cómo se llega a un resultado o un gráfico en particular (hay mucho movimiento y "click" de mouse de por medio). La reproducibilidad es (o debería ser) un estándar mínimo para el trabajo científico. Si bien, el hecho de que algo sea reproducible no quiere decir que sea correcto, al menos lo hace más fácil de evaluar, de revisar y de re-utilizar. Un paso importante para lograr la reproducibilidad es hacer todo con instrucciones escritas (*scripts*) en lugar de con el mouse.

En resumen, para trabajar en ciencia hay que saber un mínimo de programación. Si tenemos que elegir en qué lenguaje programar, `R` o `Python` parecen las mejores opciones a la fecha, pero `R` es actualmente el más popular entre los ecólogos y es el que vamos a usar en este curso. `R` es una implementación de dominio público del lenguaje estadístico `S` que ha ganado popularidad por varios motivos. Se puede descargar de https://cran.r-project.org/ y funciona en PC, Mac y Linux. Además de usarse para análisis estadísticos, `R` se puede programar para hacer casi cualquier cosa. Tiene programación basada en objetos y programación funcional. Hay una creciente comunidad de usuarios, incluyendo a algunos grosos...

`R` se distribuye con una interfase gráfica rudimentaria, pero hay varios motivos para trabajar con `R` através de RStudio. Entre otras cosas, RStudio hace posible conectarse con repositorios online como GitHUb o Bitbucket para control de versiones y crear documentos (como éste que fue escrito en RStudio) que inlcuyen código de `R`, figuras, ecuaciones, etc. Aprendiendo a usar `R` y a documentar nuestro trabajo con las herramientas disponibles através de RStudio podremos trabajar mejor y de manera más eficiente. Al principio no parece, pero es **mucho mejor** trabajar con registros de comandos (scripts) y lograr que nuestra investigación sea reproducible. De lo contrario y parafraseando a Walter Sobchak (1998), seremos "*fuckin' amateurs*"...

Cosas Básicas de R

R funciona como una calculadora:

```
2 + 2
```

```
## [1] 4
```

```
2 * 3
```

```
## [1] 6
```

```
3^2
```

```
## [1] 9
```

```
2/3
```

```
## [1] 0.6666667
```

```
exp(1)
```

```
## [1] 2.718282
```

```
log(10)
```

```
## [1] 2.302585
```

```
log10(10)
```

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

R resuelve operaciones lógicas:

```
2 > 1
```

```
## [1] TRUE
```

```
2 == 1
```

```
## [1] FALSE
```

```
2 <= 2
```

```
## [1] TRUE
```

Un aspecto básico que tenemos que dominar es que R trabaja con vectores, matrices, listas y "dataframes". Empecemos generando un vector:

```
n <- c(2, 6, 18, 54, 162, 486)
```

El comando c concatena los valores entre comas. La flecha <- asigna lo que está a la derecha al 'objeto' de la izquierda. En vez de <- podemos usar = pero los puristas de R prefieren <-.

Para ver los valores del vector en la ventana de comando escribimos su nombre el la ventana de comando y hacemos *enter*:

```
n
```

```
## [1]   2   6  18  54 162 486
```

Un tipo especial de vector son las secuencias. Para generar una secuencia de valores podemos hacer por ejemplo:

```
yr <- 2009:2014
```

Hagamos de cuenta que los vectores n y yr corresponden a censos de una especie invasora. Si queremos ver el tamaño poblacional en el año 2010 hacemos:

```
n[2]
```

```
## [1] 6
```

Una alternativa es hacer:

```
n[yr == 2010]
```

```
## [1] 6
```

O usar la función `which`:

```
idx <- which(yr == 2010)
n[idx]
```

```
## [1] 6
```

Si queremos ver el tamaño poblacional desde el 2011 al 2014

```
n[3:6]
```

```
## [1]  18  54 162 486
```

o bien

```
n[yr >= 2011]
```

```
## [1]  18  54 162 486
```

o con `which`:

```
idx <- which(yr > 2010)
n[idx]
```

```
## [1]  18  54 162 486
```

Algo interesante para ver en un set de datos como este es el "factor reproductivo" o "tasa finita de incremento la población", que no es otra cosa que $n(t+1)/n(t)$. Por ejemplo, para el 2010 la tasa de crecimiento fue de: `n[2]/n[1]` $= 3$.

Bucles (*loops*)

Para no repetir las cuentas año por año podemos hacer un "bucle" (*for loop*) pero antes necesitamos generar un vector donde vamos a guardar los resultados:

```
tasa <- numeric(length(n) - 1)
```

Aquí combinamos dos funciones de R: `numeric` y `length`. Para ver qué hace `numeric` podemos hacer `?numeric`. Poniendo un signo de pregunta y luego el nombre de una función de R accedemos a la documentación de esa función (que a veces no es de mucha ayuda...). Una mejor forma de encontrar ayuda e inlcuso información acerca de funciones que no conocemos es visitando http://rseek.org/.

Volviendo a nuestro *loop*, lo que queremos es repetir una operación (o varias) e ir guardando los resultados:

```
for (i in 2:length(n)) {
    tasa[i - 1] <- n[i]/n[i - 1]
}
```

El comando `for` en este caso repite en valores de `i` la cuenta $n(t+1)/n(t)$ que en lenguaje de R es `n[i]/[i-1]` porque `i` está definido desde 2 hasta el largo del vector `n`. Usamos el valor de `i` para cada iteración para ver qué valores de `n` usar en la cuenta y para indicar en qué posición del vector `tasa` guardamos el resultado.

En este caso podríamos obtener los mismos valores haciendo:

```
for (i in 1:(length(n) - 1)) {
    tasa[i] <- n[i + 1]/n[i]
}
```

Es importante notar que `1:(length(n)-1)` no es lo mismo que `1:length(n)-1`. Compruébenlo...

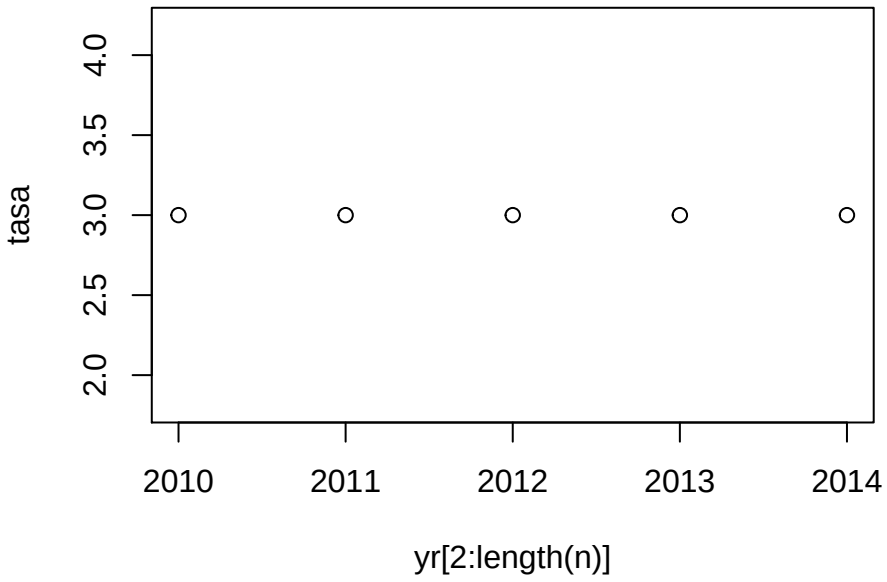
Para ver qué salió de estas cuentas escribimos

```
tasa
```

```
## [1] 3 3 3 3 3
```

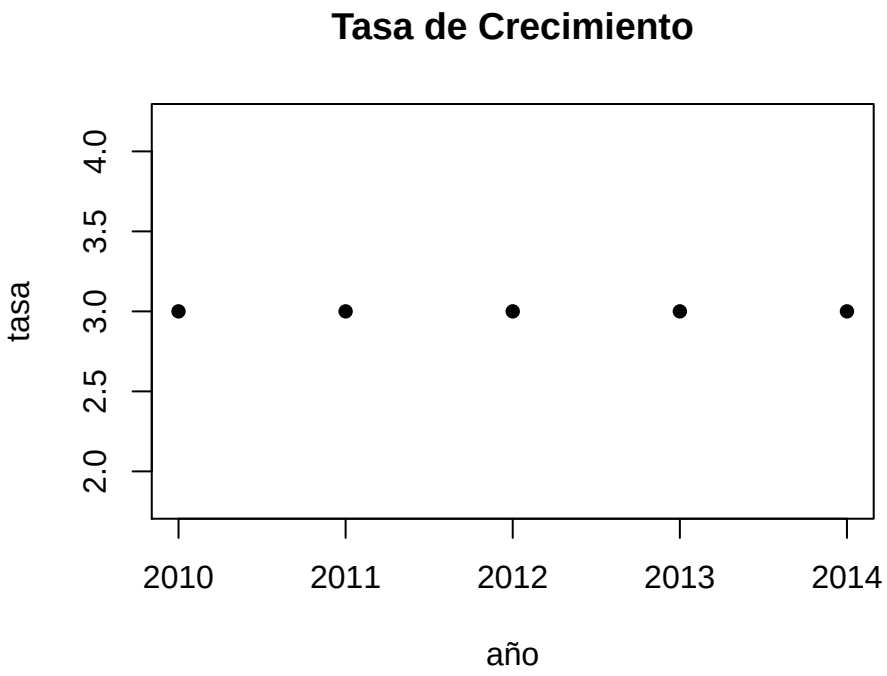
Para graficar la tasa de incremento en cada año podemos usar

```
plot(yr[2:length(n)], tasa)
```



Para "embellecer" un poco el gráfico podemos usar `main`, `xlab`, y `pch`

```
plot(yr[2:length(n)], tasa, main = "Tasa de Crecimiento", xlab = "año", pch = 16)
```

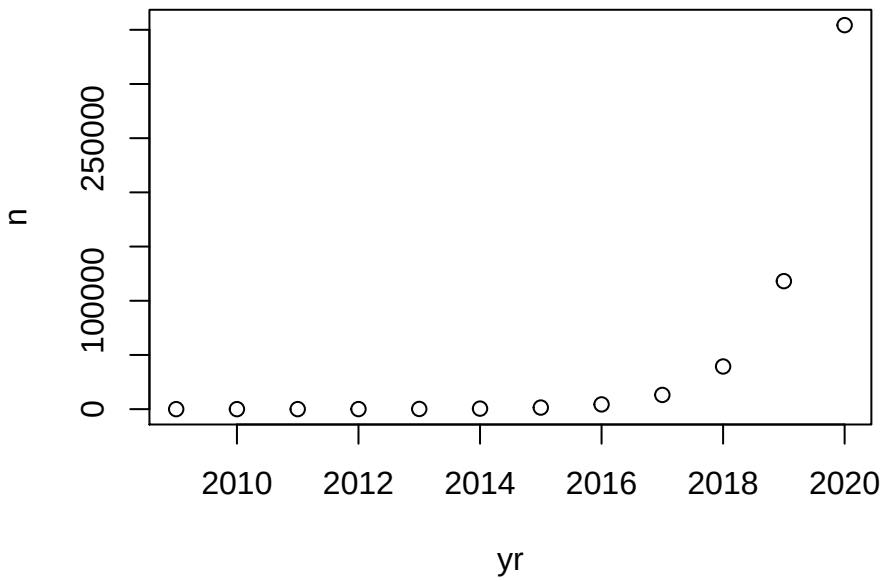


Ahora que aprendimos a usar un *for loop*, podemos ver que en este caso no lo necesitábamos! De hecho, una muy buena característica de `R` es que permite hacer operaciones sobre vectores, listas, etc. como veremos oportunamente. De todas formas, es importante aprender a manejar los *loops* ya que son una herramienta básica de programación.

```
tasa <- n[2:6]/n[1:5]
```

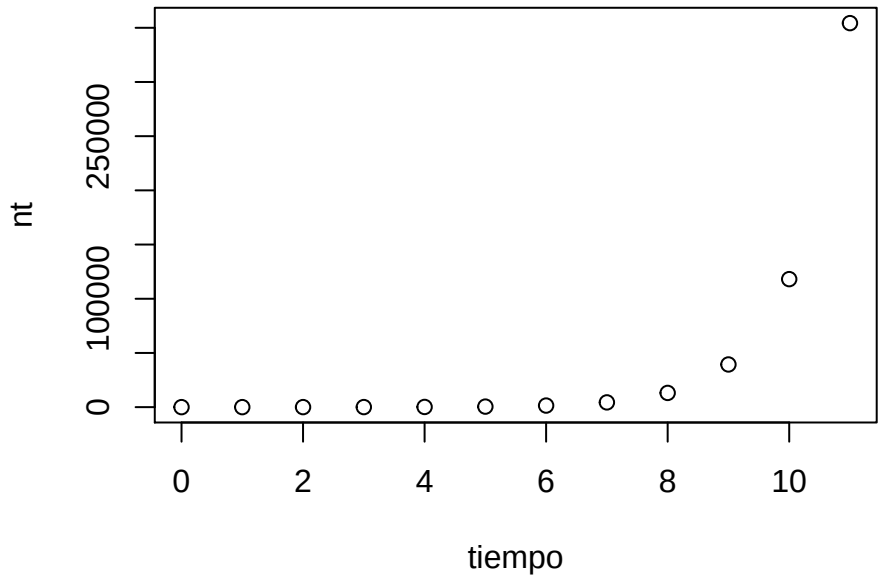
En cualquier caso, vemos que la tasa de incremento poblacional es constante e igual a 3. Podemos usar ese valor para proyectar el tamño poblacional en el futuro:

```
lambda <- 3
yr <- c(yr, 2015:2020)
n <- c(n, numeric(6))

for (i in 7:length(n)) {
    n[i] <- n[i - 1] * lambda
}

plot(yr, n)
```

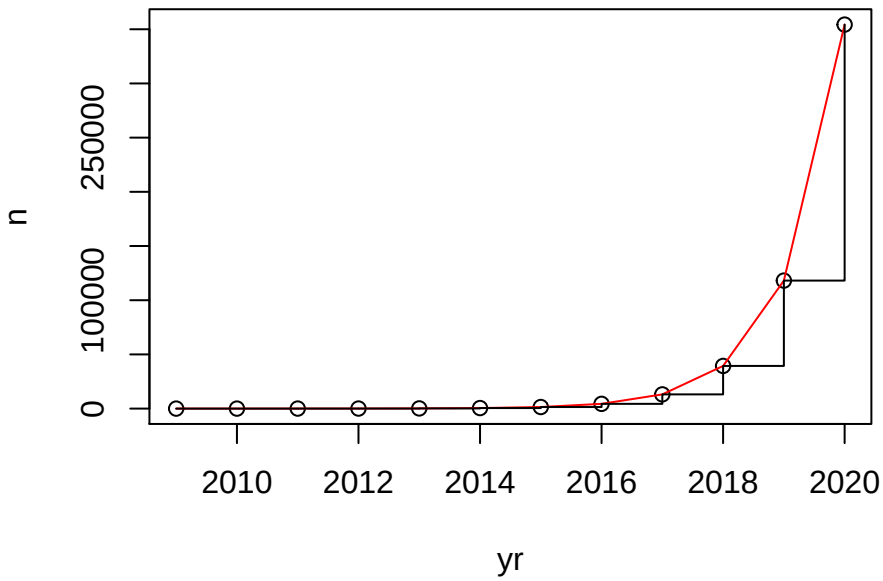


Lo mismo podríamos hacer definiendo condiciones iniciales y usando la fórmula $n(t) = n(0)\lambda^t$.

```
n0 <- 2
tiempo <- 0:11
nt <- n0 * lambda^tiempo

plot(tiempo, nt)
```



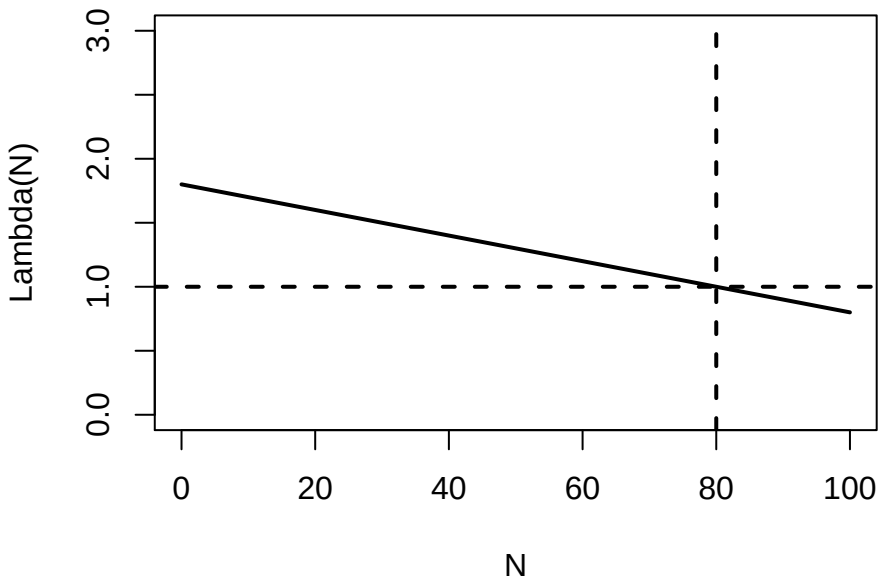

```
# o podemos graficar las dos trayectorias poblacionales
plot(yr, n)
lines(tiempo + yr[1], nt, col = 2)
lines(tiempo + yr[1], nt, type = "s")
```



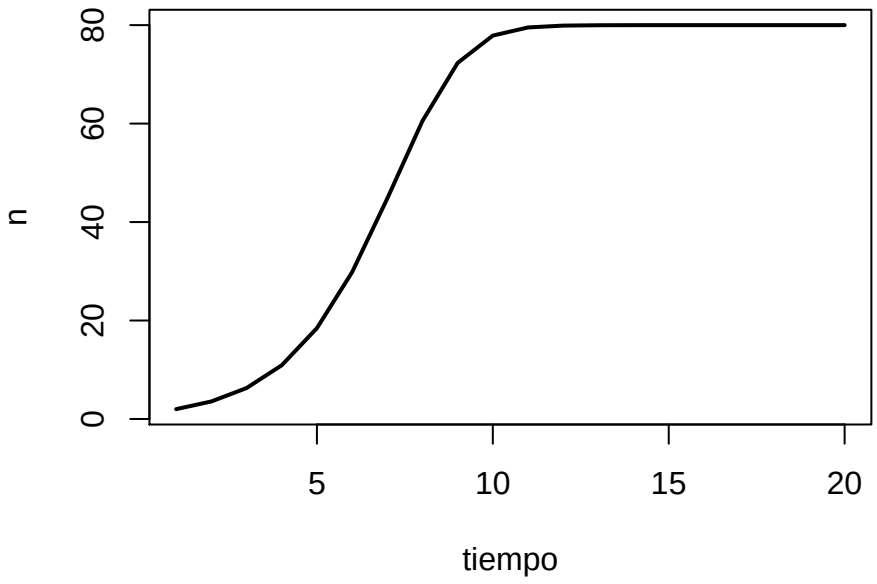

```
# 'lines' agrega lineas a un gráfico existente 'col' es una opción de
# gráficos para definir colores
```

Es poco creible que la población siga y siga creciendo como $n(t) = n(0)\lambda^t$. Una opción es pensar que la tasa de incremento poblacional decrece con el tamaño poblacional (modelo logístico).

```r
lambda <- 1.8
N <- 0:100
K <- 80  # capacidad de carga
rd <- lambda - 1  # rd es la tasa de crecimiento poblacional per cápita
plot(N, lambda - rd/K * N, type = "l", ylab = "Lambda(N)", ylim = c(0, 3), lwd = 2)
abline(h = 1, lty = 2, lwd = 2)
abline(v = K, lty = 2, lwd = 2)
```




```r
# simulemos una trayectoria
n0 <- 2
n <- numeric(20)
n[1] <- n0
for (t in 1:(length(n) - 1)) {
    n[t + 1] <- n[t] + rd * n[t] * (1 - n[t]/K)
}

plot(1:length(n), n, type = "l", lwd = 2, xlab = "tiempo")
```



Escribiendo Funciones

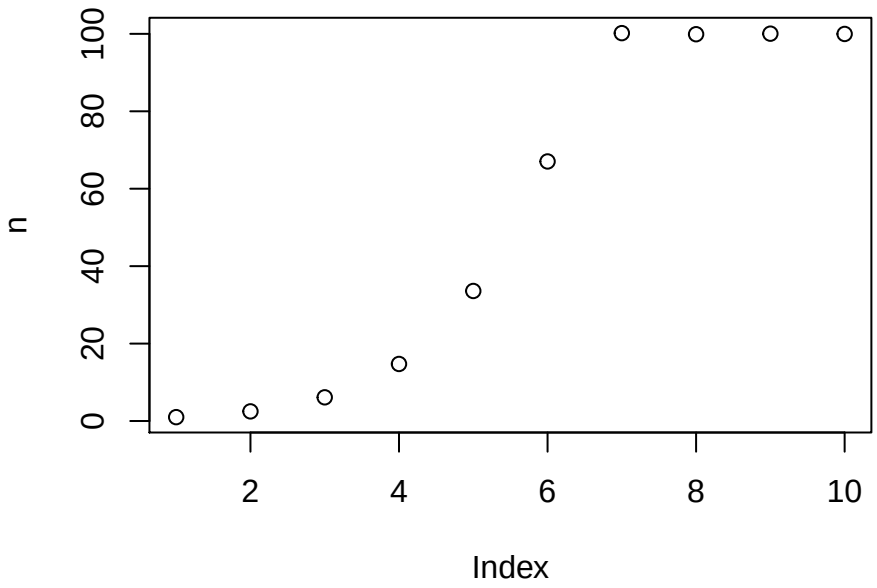
Un aspecto clave de la programación es escribir funciones. Una función es un conjunto de instrucciones que toma "inputs" o "argumentos", usa esos inputs para calcular otros valores que devuelve como resultado. Por ejemplo, podemos escribir una función para el modelo logístico:

```
logistic <- function(n0, rd, K, nyr = 10) {
    n <- numeric(nyr)
    n[1] <- n0
    for (t in 1:(length(n) - 1)) {
        n[t + 1] <- n[t] + rd * n[t] * (1 - n[t]/K)
    }
    return(n)
}
```

Esta función tiene como argumentos a `n0`, `rd`, `K` y el número de años (`nyr`) que queremos simular.

A menos que le digamos otra cosa, `R` respeta el orden en que definimos los inputs de la función. Entonces, si queremos simular la trayectoria de una población que sigue un modelo logístico con `n0=1`, `rd=1.5`, y `K=100` por `10` generaciones, podemos hacer:

```
n <- logistic(1, 1.5, 100)
plot(n)
```



En este caso, no le dijimos a la función que queríamos `nyr=10` porque está definido por defecto. En realidad no es muy buena idea pasarle valores de esta manera a las funciones. Nos conviene ser más explícitos:

```
n <- logistic(n0 = 1, rd = 1.5, K = 100, nyr = 20)
```

Incluso podemos cambiar el orden de los argumentos si es que los llamamos por su nombre:

```
n <- logistic(nyr = 20, rd = 1.5, n0 = 2, K = 100)
```

Mejor aún es definir los valores de los inputs aparte. Por ejemplo:

```
n0 <- 2
rd <- 1.5
K <- 100
nyears <- 20  # no necesariamente usamos el mismo nombre
n <- logistic(nyr = nyears, rd = rd, n0 = n0, K = K)
```

Ejercicios:

Ricker (1954), cambió la relación lineal entre λ y n por $exp(-a * n)$. Escribir una función para simular trayectorias poblacionales con el modelo de Ricker y compararlo con el logístico.

```
con el logístico
```