Técnicas de Simulación

06. Sistemas dinámicos

Miguel A. Castellanos

Contents

---

1 Sistemas dinámicos

Como dijimos en un tema anterior los sistemas estacionarios son aquellos en los que el procesamiento se produce independientemente del tiempo, es decir, todos los procesos son cohetáneos, se producen simultáneamente. Por el contrario los sistemas dinámicos son aquellos en los que alguna parte del sistema (o todo él) cambia al fluir el tiempo.

En los sistemas temporales sencillos el tiempo es solo una variable de entrada más, es lo que se llama un tiempo paramétrico y su influencia es igual que la de otra variable, por ejemplo: La probabilidad de detectar un estímulo visual se puede caracterizar como una función logística del tiempo:

$$P(detección) = logística(t)$$

En este caso el tiempo funciona de forma paramétrica y sería equivalente a utilizar cualquier otra variable x, la edad, la inteligencia, etc. Un modelo así, aunque es un modelo temporal, no es estrictamente un modelo dinámico. Para que un sistema sea dinámico el estado del sistema en un momento dato t ($S(t)$) tiene que depender de alguna manera del estado anterior t-1, $S(t) = f(S(t-1))$.

La programación del tiempo suele ser compleja y difícil, por eso, salvo que sea estrictamente necesario, es preferible no introducirlo como variable. Cuando simulamos sistemas de visión humana es evidente que los procesos se suceden en el tiempo, pero como no es relevante calcular o simular ese tiempo simplemente se tiene en cuenta que un proceso antecede o precede a otro, como en cualquier sistema estacionario. Introducir el tiempo haría innecesariamente complejo el modelo y no aportaría valor.

Los modelos dinámicos son realmente complejos, y son muy utilizados en muchas ciencias como la economía, la ecología, la epidemiología, etc. En el siguiente gráfico se expone un modelo dinámico sencillo de crecimiento utilizado en economía:

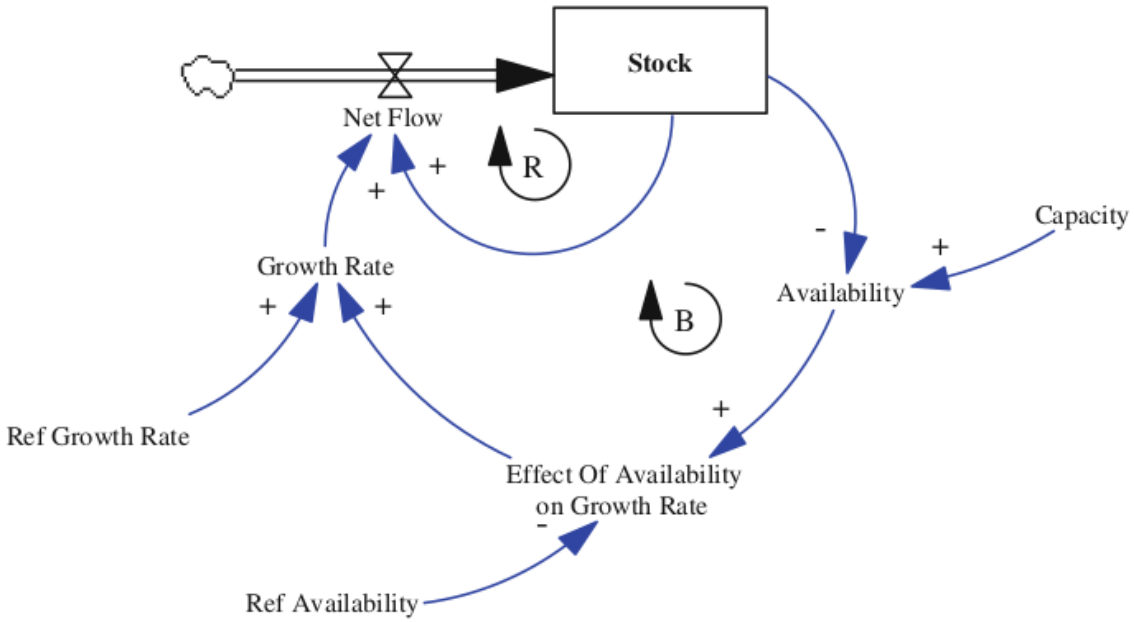


Figure 1: Modelo económico

Estos modelos pueden ser difíciles de implementar y en general, para trabajar con ellos se utilizan soluciones numéricas como veremos más adelante. En esta lección veremos modelos dinámicos que resolveremos calculándolos por etapas.

La forma más sencilla de introducir el tiempo es siempre forma discreta, la variable de entrada es un conjunto de valores discretos (t: t0, t1, t2, ...), cada una de esas unidades temporales discretas se suelen llamar **etapas o épocas**. Una forma más compleja implica que el tiempo fluya de manera continua en el sistema, pero eso no lo vamos a explicar porque supone un incremento sustancial de complejidad y no se verá en este curso. En el resto de la lección trabajaremos cono modelos dinámicos de tiempo discreto.

2 Iteratividad y Recursividad

Cuando el sistema es dinámico, pero sin variables que influyan entre las etapas, y el tiempo transcurre de forma discreta (cada ciclo es una nueva ejecución), suelen llamarse sistemas **iterativos**. Un proceso iterativo es aquel en el que la salida de un ciclo es la entrada del ciclo siguiente. Todo proceso iterativo (y todo proceso dinámico) tiene que ser inicializado con algún valor o semilla que inicie la secuencia.

$$f_0(semilla) \rightarrow f_1(f_0(semilla)) \rightarrow f_2(f_1(f_0(semilla))) \cdots$$

Una forma muy hábil de programar la iteratividad es con recursividad. Se dice que un proceso es recursivo cuando este se llama a sí mismo, por ejemplo: una función que en su código se llama a sí misma. Es más claro con un ejemplo: vamos a sumar los x primeros números naturales de forma recursiva, fíjate cómo dentro de la función *suma* se llama a sí misma x + *suma* (x-1).

```
suma <- function(x){
  x+suma(x-1)
}

suma(4) # 4+3+2+1
```

Si has ejecutado esta sintaxis habrás visto cómo la función te ha dado un error de *stack*, este error se refiere a la pila de memoria del sistema operativo y nos indica que la función ha empezado a llamarse recursivamente a sí misma hasta que ha saturado la memoria disponible. Eso es muy habitual en recursividad. La recursividad es muy eficiente pero muy fácil de cometer errores graves al usarla. Sin duda alguna la recursividad ha matado a disgustos a estudiantes de informática durante décadas y destruido más de un ordenador. Por suerte en R, al ser un lenguaje interpretado, no se tiene acceso directo al hardware del ordenador por lo que el máximo perjuicio que vamos a crear es que aparezca ese mensaje, pero en lenguajes que permiten el control a bajo nivel como C hay que tener más cuidado al programar recursividad.

La solución al problema anterior es incorporar algún tipo de control dentro de la función, vamos a reescribirla:

```
suma <- function(x){
  if (x <= 1) return(x)
  return(x+suma(x-1))
}
```

Ahora el *if* va a controlar que no continúe sumando más allá del 1 y que se detenga la función a su debido tiempo.

```
suma(3)
```

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

```
suma(10)
```

```
## [1] 55
```

Ahora la función nos devuelve los valores correctamente. No es imprescindible programar funciones recursivas, ya que la mayoría de las veces se puede resolver el mismo problema usando bucles, pero en general, las soluciones recursivas se consideran "más elegantes" que las de los bucles.

2.1 Tarea 1

Crea una función recursiva para calcular el factorial de un número, recuerda que $n! = n * n-1 * n-2 * ... * 1$ y que $0! = 1$ y $1! = 1$.

Otra utilidad típica de la recursividad es el trabajo con series numéricas, por ejemplo con la famosa serie de Fibonacci. Mira el siguiente código:

```
# fibonacci recursivo con control
fib <- function(x,y, conta, stop=10){
  if (conta > stop) return("terminado")
  z = x+y
  print(z)
  fib(y,z, conta+1, stop)
}
```

```
fib(1,1)
```

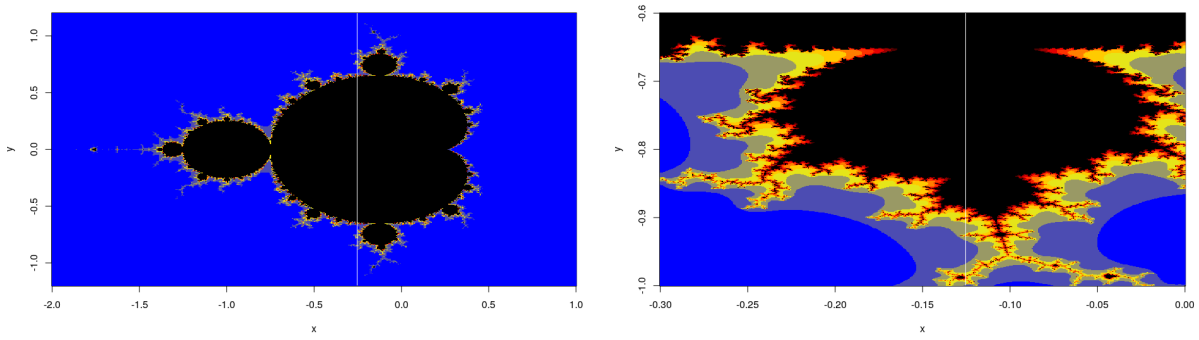
Definimos la función y la inicializamos con los dos primeros números de las serie *fib(1,1)*. Esta función acaba dando un error porque volvemos a saturar la memoria, esto se produce porque no hemos programado un mecanismo de control.

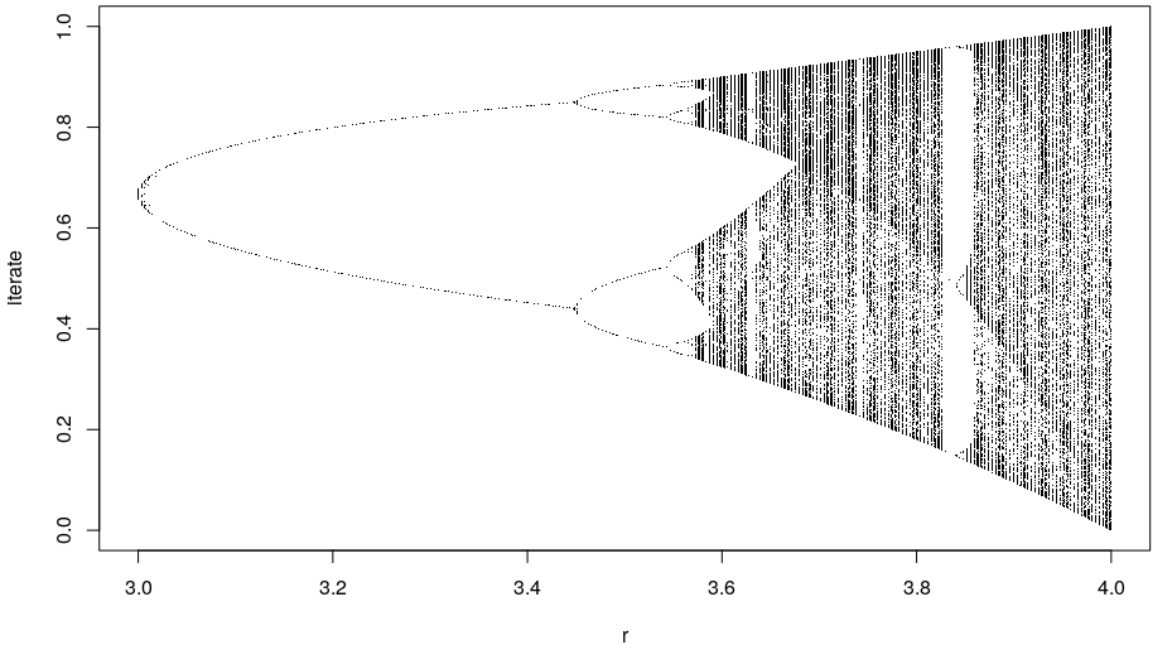
2.2 Tarea 2

Construye una función recursiva de fibonacci que tenga control, es decir, que se le indique el número de elementos que tiene que calcular y se detenga al llegar a él.

3 Caos

Aunque los sistemas iterativos sean sencillos no tienen por qué producir resultados simples. Para hacerse una idea, la mayoría de las matemáticas del Caos y de los sistemas complejos se han estudiado con sistemas iterativos de ecuaciones muy sencillas, como el atractor de Lorenz o los fractales. En cuanto tenemos dos variables interactuando entre sí ya podríamos acabar con un sistema no lineal casi imposible de trabajar con él. En la siguiente figura os pongo dos ejemplo de cómo dos funciones dinámicas muy sencillas producen resultados aparentemente caóticos. Uno de ellos es el fractal de mandelbrot y la otra un sistema de bifurcación.





Las matemáticas del caos son muy bonitas e interesantes pero exceden los objetivos del curso. Os dejo en los anexos 1 y 2 el código en R para crear, de forma un tanto rudimentaria, el hombre de mandelbrot y un sistema dinámico de bifurcación. El alumno si está interesado puede investigar por su cuenta con el código y en los siguientes enlaces:

- Mandelbrot
- Árbol de bifurcaciones

4 Un modelo epidemiológico sencillo (SIR)

Una forma perfecta para ejemplificar un sistema dinámico son los modelos epidemiológicos, en los que el estado de expansión de una epidemia en un momento t depende del estado t-1. Los modelos epidemiológicos pueden ser realmente complejos, introduciendo cientos de variables en los cálculos y requiriendo de supercomputadores para resolver las predicciones; aquí vamos a trabajar con el más sencillo de todos, y aun así potente y predictivo, como es el modelo SIR.

SIR son la siglas de *Susceptible, Infected y Removed*. Los individuos en este modelo solo pueden ser de un de estos tres tipos:

- *Susceptible* (S): Es la población que es susceptible de ser contagiada, al principio de la epidemia es toda la población.
- *Infected* (I): individuos infectados, aquellos que están contagiados en un momento t.
- *Removed* (R): Individuos eliminados de la epidemia, son aquellos infectados que ya no pueden contagiarse, bien porque han muerto o porque han desarrollado inmunidad.

EL modelo es controlado por tres parámetros

- N: El tamaño de la población
- α: La probabilidad de infección, a veces se expresa como ratios, por ejemplo, que se infecte uno de cada 10.000 supone una probabilidad de $1/10000 = 0.0001$.

- β: La probabilidad de que un infectado pase a ser Removed (R).

El modelo es de tiempo discreto, en el que cada unidad t de tiempo es definida arbitrariamente, en este ejemplo cada unidad de tiempo va a ser un día.

Con estos elementos se puede definir el modelo de la siguiente manera:

Situación inicial: partimos de t=0, con una población de N individuos de los cuales uno está contagiado, las probabilidades de contagio y eliminación son α y β respectivamente. En el inicio tenemos N sujetos susceptibles, 1 elemento infectado y 0 eliminados (Removed):

$$S(0) = N$$
$$I(0) = 1$$
$$R(0) = 0$$

Es evidente que N=S(t)+I(t)+R(t)

Para un momento t determinado los valores de S,I y R son los siguientes:

$$S(t+1) \sim binom(S(t), (1-\alpha)^{I(t)})$$

$$R(t+1) \sim R(t) + binom(I(t), \beta)$$
$$I(t+1) = N + 1 - R(t+1) - S(t+1)$$

El modelo SIR es un modelo claramente dinámico porque para conocer el estado de los valores en t necesitas haber calculado todos los valores anteriores.

Fíjate que el modelo SIR no distingue entre personas que mueren y supervivientes, para él todos son *Removed*, además no tiene en cuenta ciertas variables como la topología, cercanía de unos individuos a otros, movilidad, aparición de vacunas, etc. Como he dicho el modelo SIR es muy sencillo, pero nos sirve para iniciarnos en los modelos epidemiológicos.

4.1 Tarea 3

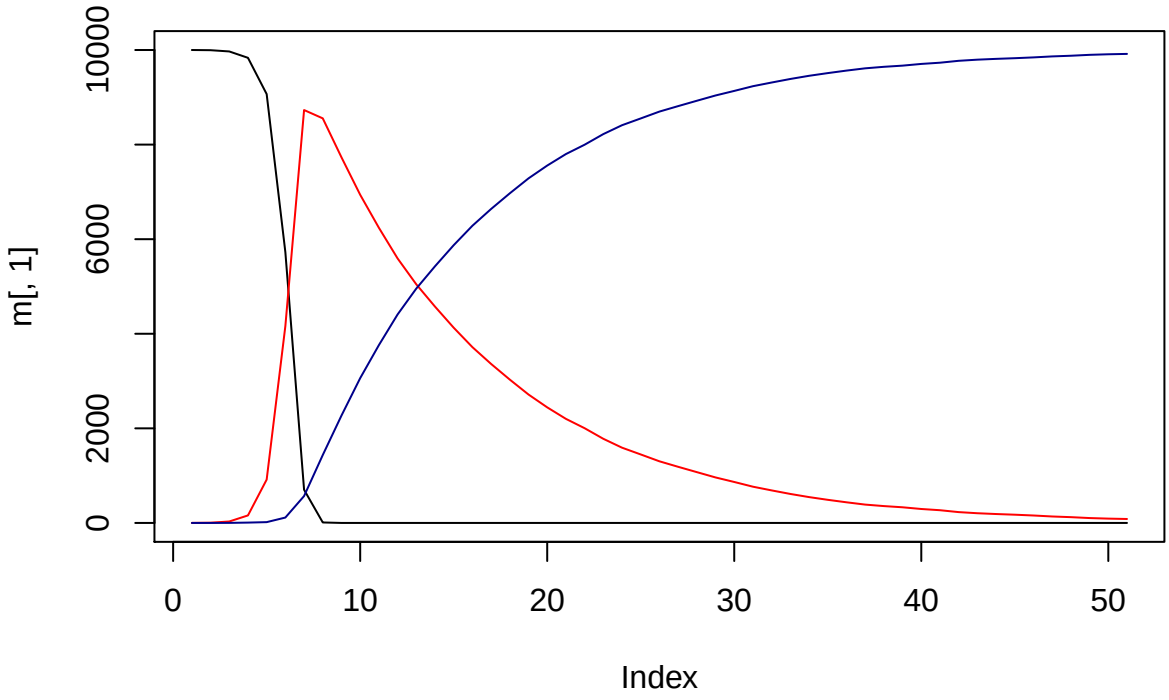
Construye un función que implemente el modelo SIR, tiene que ser un función que acepte los siguientes parámetros: alfa, Beta, N y T, donde T es el número de etapas (días) que va a ser calculado.

El modelo tiene que ser ejecutado para T épocas o iteraciones, y debe devolver los valores de S,R e I para cada una de las épocas (en una matriz o data.frame). No se pueden utilizar soluciones numéricas como *desolve* y similares, debe programarse directamente el cálculo de cada etapa.

Si lo has programado adecuadamente e inicializas la semilla aleatoria con set.seed(1), deberías obtener los siguientes resultados:

```
set.seed(1)
m <- SIRsim(a = 0.0005, b = 0.1, N=10000, T=50)

plot(m[,1], type="l", col="black")
lines(m[,2], type="l", col="red")
lines(m[,3], type="l", col="darkblue")
```




```
head(m) #S, I, R
```

```
##       [,1] [,2] [,3]
## [1,] 10000    1    0
## [2,]  9996    5    0
## [3,]  9970   30    1
## [4,]  9835  159    7
## [5,]  9068  916   17
## [6,]  5725 4162  114
```

Como he dicho antes el modelo SIR es una simplificación, en la realidad los individuos Removed se dividen en aquellos que han muerto (*Death*) y los que han sobrevivido y muestran inmunidad (*Recovered*). Para tener estas dos categorías tenemos que dividir β en dos parámetros distintos: β_D y β_R, de tal forma que ahora:

$$D(t+1) \sim D(t) + binom(I(t), \beta_D)$$

$$R(t+1) \sim R(t) + binom(I(t), \beta_R)$$

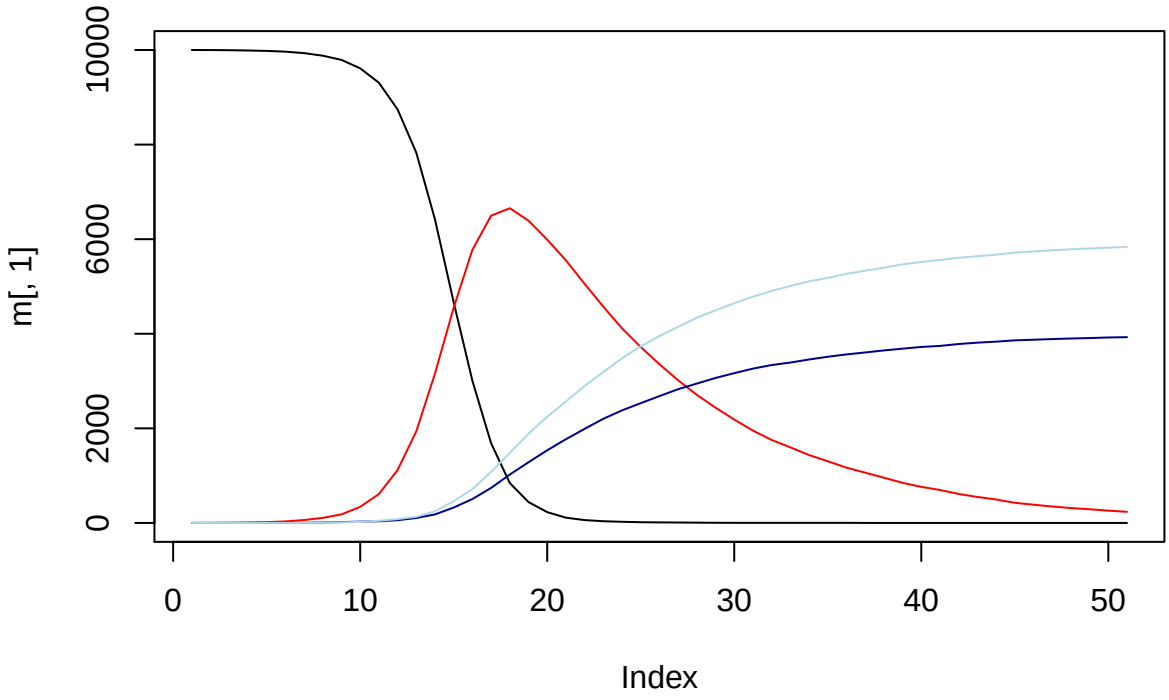
Donde D representa a *Death* y R ahora no representa a los *Removed*, sino a los *Recovered*.

4.2 Tarea 4

Reconstruye el modelo anterior a uno nuevo llamado SIDR que incluya β_D y β_R. Inicializando a set.seed(1) deberías obtener lo siguiente:

```
m <- SIDRsim(a = 0.0001, bd = 0.04, br=0.06, N=10000, T=50)

plot(m[,1], type="l", col="black")
lines(m[,2], type="l", col="red")
lines(m[,3], type="l", col="darkblue")
lines(m[,4], type="l", col="lightblue")
```




```
head(m) #S, I, D, R
```

```
##       [,1] [,2] [,3] [,4]
## [1,] 10000    1    0    0
## [2,]  9999    2    0    0
## [3,]  9996    5    0    0
## [4,]  9990    9    2    0
## [5,]  9981   17    2    1
## [6,]  9965   32    3    1
```

4.3 Tarea 5

Una vez construido tu modelo, y comprobado que funciona correctamente, podemos pasar a su utilización. Supongamos una población de N=100000 individuos y una enfermedad cuyo α puede variar entre 0.5 y 9.0, el β_d entre 0.01 y 0.05 y el β_r entre 0.05 y 0.15. Con estos datos, calcule, por fuerza bruta, el peor y el mejor escenario posible, es decir, aquella combinación de parámetros que dan la mayor y menor cantidad de muertes al final de la simulación. Dibuje las gráficas de ambos escenarios. Si es posible, paralelízelo.

5 Expansión de un fuego

Existen modelo topológicos que tratan de simular el comportamiento de un proceso en un espacio n-dimensional. Uno de los más sencillos es el de expansión de un incendio, que es también un sistema dinámico ya que el avance del fuego en un momento t+1 depende de la situación del incendio en t. Vamos a verlo paso a paso.

Primero tenemos que presentar una función que nos será muy útil: *write.gif* del paquete "*caTools*". Esta función nos crea un gif animado a partir de un conjunto de matrices, mira el ejemplo:

```
require(caTools)
```

```
## Loading required package: caTools
```

```
lado = 25
steps = 10


# creamos un array 3D para almacenar todas las matrices que creemos
storage <- array(0, c(lado, lado, steps))

for (i in 1:steps) {
  # creamos una matriz aleatoria
  M <- matrix(rbinom(lado**2,1,0.4), nrow=lado, ncol=lado)

  # la almacenamos en storage
  storage[,,i] <- M
}

storage <- storage/max(storage)
write.gif(storage, filename="ejemploGif.gif", col="jet", delay=100)
```

Este código habrá producido un gif que al abrirlo (puedes directamente tirarlo en un navegador con Chrome y se abrirá) parece un grupo de puntos aleatorios. Como es solo de 25 x 25 se verá muy pequeño así que hazlo lo más grande que puedas para verlo mejor. Una imagen de una de esas matrices es la siguiente:

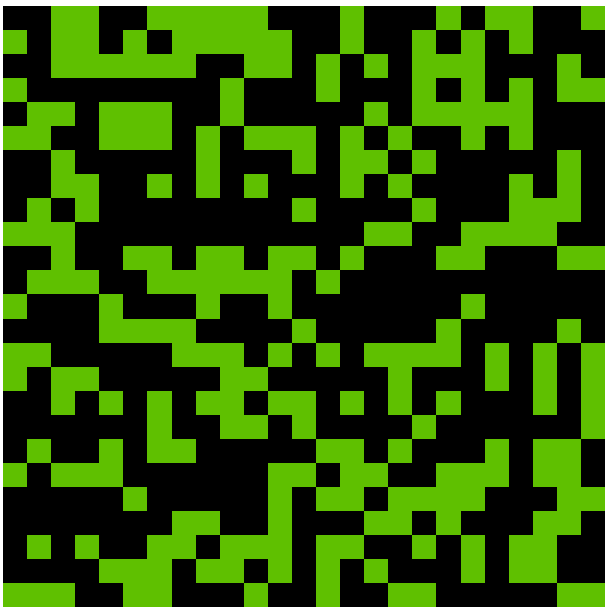


Figure 2: Un gif aleatorio

El segundo elemento que vamos a necesitar es una función (sumVecinos) que, dada una matriz, hará la suma de los valores adyacentes a un punto dado. Esta función es la clave de todo:

```
# Creamos una matriz aleatoria de 0 y 1 de lado = 10
set.seed(1)

# creamos una matriz cuadrada compuesta por 0 y 1 aleatorios
lado= 10
M <- matrix(rbinom(lado**2,1,0.4), nrow=lado, ncol=lado)
M
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    0    1    0    1    0    1    0    0     0
##  [2,]    0    0    0    0    1    1    0    1    1     0
##  [3,]    0    1    1    0    1    0    0    0    0     1
##  [4,]    1    0    0    0    0    0    0    0    0     1
##  [5,]    0    1    0    1    0    0    1    0    1     1
##  [6,]    1    0    0    1    1    0    0    1    0     1
##  [7,]    1    1    0    1    0    0    0    1    1     0
##  [8,]    1    1    0    0    0    0    1    0    0     0
##  [9,]    1    0    1    1    1    1    0    1    0     1
## [10,]    0    1    0    0    1    0    1    1    0     1
```

```
# definimos la función sumaVecinos
sumVecinos <- function(M,i,j, valor){
  f_min <- max(1, (i - 1))
  f_max <- min(nrow(M), (i + 1))
  c_min <- max(1, (j - 1))
  c_max <- min(ncol(M), (j + 1))

  sum(M[f_min:f_max, c_min:c_max]==valor)
}


# probamos que funciona bien
sumVecinos(M,2,2, 1) # correcto, la suma de los vecinos del punto M[2,2] es 3
```

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

```
sumVecinos(M,6,8, 1) # correcto
```

```
## [1] 5
```

```
# no produce errores cuando los vecinos estan fuera de la matriz
sumVecinos(M,1,1, 1)
```

```
## [1] 0
```

```
sumVecinos(M,10,10, 1)
```

```
## [1] 2
```

Fíjate que la función solo suma los números que coinciden con *valor*, así le podemos indicar que sume los adyacentes que son iguales a 1, a 2 o al valor que deseemos.

Ahora ya podemos simular el modelo de expansión de un fuego. Según este modelo tendremos una matriz bidimensional que simula el bosque, a la que llamaremos Bosque.

Dentro del bosque puede haber los siguientes elementos:

- Un hueco: una posición del Bosque que no está ocupada por un árbol, se representa con un 0
- Un árbol: un posición del bosque ocupada por un árbol y representada por un 1
- Un árbol ardiendo: se representa con un 2
- un árbol apagado: un árbol que o bien se ha apagado o se ha calcinado completamente y que ya no puede volver a arder, se representa con un 3.

El fuego está controlado por un parámetro alfa que es la probabilidad de que un árbol empiece a arder y una probabilidad beta que es la probabilidad de que se apague el fuego en un árbol.

La probabilidad de que un árbol salga ardiendo depende de cuántos árboles a su alrededor están ardiendo, lógicamente si está rodeado de árboles ardiendo será más fácil que arda, la forma de parametrizarlo es con la

siguiente sintaxis:

if(M[i,j]==1) M2[i,j] <- if(runif(1) > (1-a)**sumVecinos(M, i, j)) 2 else 1

Para un árbol cualquiera (M[i,j]==1]) generamos un número aleatorio entre 0 y 1 (runif(1)), si este número es mayor que la probabilidad de arder ($(1-\alpha)^{sumVecinos(M,i,j)}$) el árbol arde (pasa a un valor de 2), en otro caso permanece como estaba.

La probabilidad de que se apague un árbol es simplemente beta, así que la parametrizamos como:

if(M[i,j]==2) M2[i,j] <- if(runif(1) < b) 3 else 2

Si el generador aleatorio genera un número inferior a beta entonces el árbol pasa a estar apagado (pasa a valer un 3)

Estas normas sencillas se implementan en la siguiente función:

```
incendio <- function(M, a, b, ini, steps, filename){
#  M <- matrix(0, nrow=lado, ncol=lado) # el bosque
  lado=nrow(M)

  for (i in 1:nrow(ini)) M[ini[i,1], ini[i,2]] <- 2
  # M[lado/2,lado/2] <- 2   # iniciamos el incendio en la mitad del tablero

  M2 <- matrix(0, nrow=lado, ncol=lado) # segunda matriz para no pisarse en los calculo de los vecinos

  storage <- array(0, c(lado, lado, steps))
  for(k in 1:steps){
    # print(k)
    for(i in 1:nrow(M)){
      for(j in 1:ncol(M)){
        if(M[i,j]==0) M2[i,j] <- 0
        if(M[i,j]==1) M2[i,j] <- if(runif(1) > (1-a)**sumVecinos(M, i, j, 2)) 2 else 1
        if(M[i,j]==2) M2[i,j] <- if(runif(1) < b) 3 else 2
        if(M[i,j]==3) M2[i,j] <- 3
      }
    }
    M=M2
    storage[,,k] <- M

  }

  storage <- (storage)/max(storage)
  write.gif(storage, filename=filename, col="jet", delay=100)
}
```

Y ahora podemos correr la simulación

```
# construimos un bosque todo repleto de arboles sin huecos vacios
lado = 50
alfa = 0.2
beta = 0.2

Bosque <- matrix(1, nrow=lado, ncol=lado) # el bosque

# prendemos fuego al arbol de la posicion 25, 25
ini <- matrix(c( 25,  25), ncol=2, byrow=T)
```

```
incendio(Bosque, a=alfa, b=beta, ini, 100, "incendio1.gif")
```

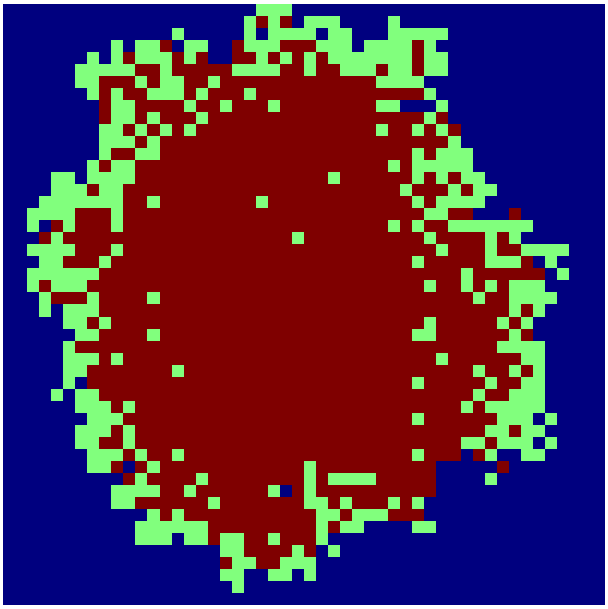


Figure 3: Un bosque ardiendo

En el fichero *incendio.gif* se puede ver la animación de la expansión del incendio hasta que todo el bosque está calcinado.

5.1 Tarea 6

Podemos sacar estadísticas del incendio en cada una de las etapas (steps), podemos calcular el número de árboles susceptibles de arder, los arboles ardiendo y los arboles calcinados. La función *incendio* puede devolver una matriz similar a la que devolvía SIDRsim y hacer un plot con esa información. De hecho el modelo de incendio es una versión bidimensional de SIR, podemos conceptualizarlo como árboles quemados o como individuos contagiados, ya que los modelos son completamente equivalentes.

Modifica la función *incendio* para que al terminar te devuelva una matriz similar a la de SIR, en el que para cada etapa te diga el número de árboles susceptibles de arder (S), el número de árboles ardiendo (I) y el número de árboles calcinados (R). Dibuja la gráfica.

5.2 Tarea 7

Como puedes construir distintas topologias del bosque estudia el efecto de la densidad de los árboles sobre la propagación del fuego, para ello simplemente modifica la forma en que construyes la matriz Bosque.

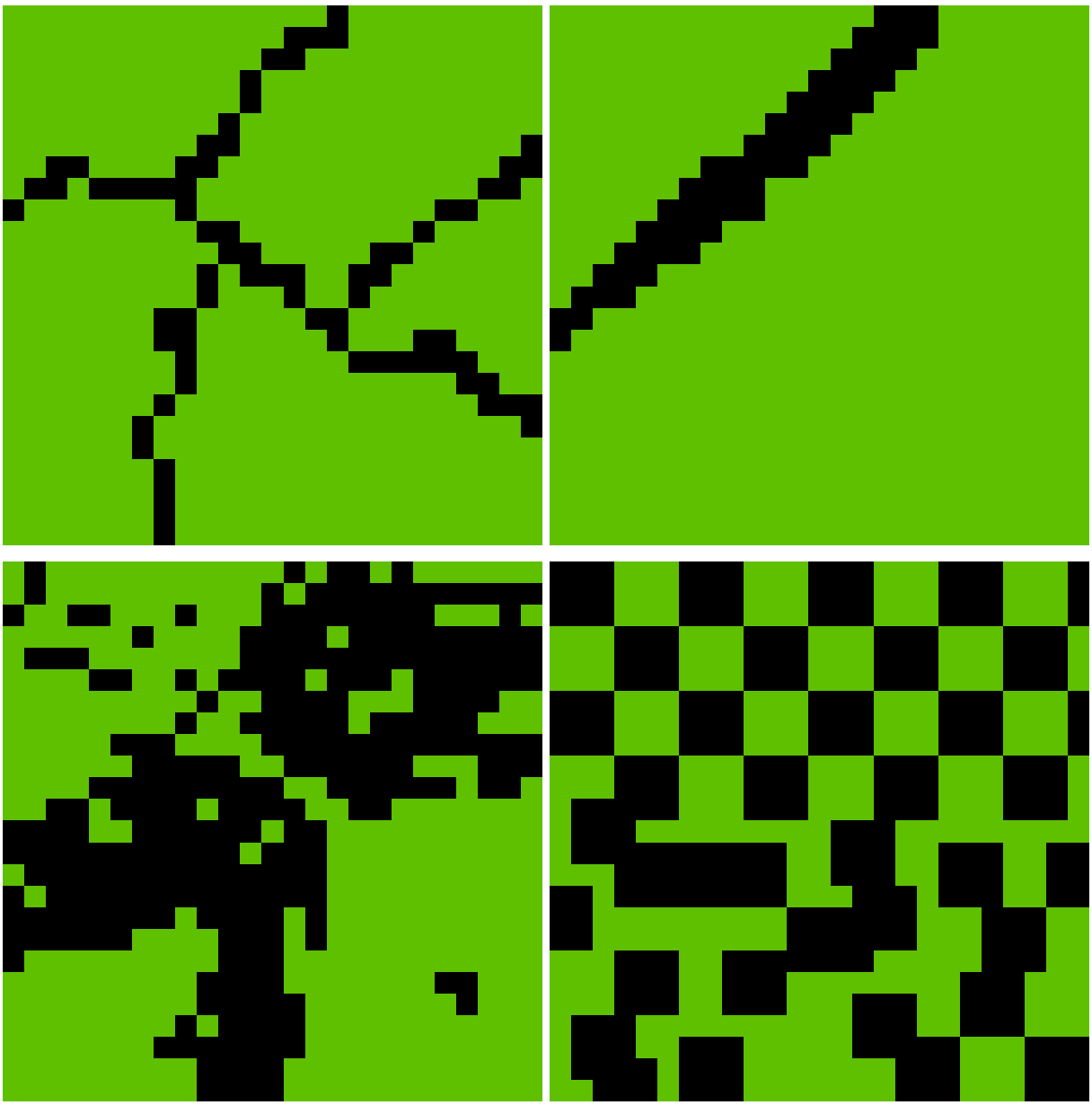
5.3 Tarea 8

Estudia cómo afecta la existencia de múltiples focos a la expansión del fuego. Para ello introduce más filas en la matriz *ini* con las posiciones de los árboles que quieres que empiecen a arder.

ini <- matrix(c(2, 2, 45, 45), ncol=2, byrow=T)

5.4 Tarea 9

Los generadores aleatorios van a producir siempre inicializaciones homogéneas del bosque. Estudia como afectan distintas organizaciones de los árboles, por ejemplo, fíjate en los siguientes dibujos:



La primera simularía caminos en un bosque, la segunda la existencia de un cortafuegos o un río, la tercera una distribución de los árboles por zonas (colinas) y la cuarta una distribución geométrica de los árboles (como en un jardín).

Los ejercicios de esta semana debe ser entregado en un zip que contenga: - El documento pdf final - El código en R - Todos los ficheros .gif que sean relevantes para mostrar el trabajo

6 El juego de la vida

El modelo de incendio es exactamente idéntico a un viejo juego llamado *el juego de la vida*. Este es uno de los juegos matemáticos más conocidos e implementados, porque al intentar simular la vida de célula artificiales y con unas reglas extremadamente sencillas, se producen patrones complejos que a día de hoy siguen sin entenderse. Te propongo que modifiques la función incendio para crear tu propio juego de la vida (en el fondo solo necesitas la función sumVecinos) y pruebes a modificar las reglas de vida y muerte para encontrar los patrones clásicos. No es una tarea obligatoria, así que hazlo solo si quieres. Para aprender más sobre el juego de la vida puedes utilizar:

- wikipedia
- rbloggers