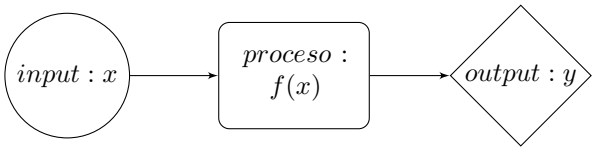
Técnicas de Simulación

04. Sistemas estacionarios

Miguel A. Castellanos

Contents

Si revisamos las tareas que hemos hecho hasta ahora todas siguen la misma lógica y se enmarcan dentro del mismo esquema: partimos de unos datos, realizamos algún análisis con ellos y obtenemos un resultado. Posteriormente analizamos estadísticamente los resultados obtenidos, modificando algunas condiciones de los datos, o de los análisis, para sacar las conclusiones pertinentes. Este esquema sencillo se puede representar en la siguiente figura:



Este es el caso, por ejemplo, que teníamos la semana pasada para estudiar los errores tipo I. Para ello generamos un montón de muestras que pertenecían a una población ($input : x$), con ellas llevábamos a cabo un contraste de *student* para una media ($proceso : f(x)$) y obteníamos el p-valor ($output$). Con esos p-valores estudiábamos si los errores tipo I cambiaban al modificar las condiciones de los supuestos de la prueba.

Este esquema sencillo de **input -> proceso -> salida** suele ser válido para la mayoría de las simulaciones estadísticas, y por tanto no es necesario hacerlo explícito cuando las explicamos; sin embargo, hay situaciones de simulación en las que el esquema es más complejo y se requiere de un modelización más sofisticada, y por tanto, una visualización clara de las relaciones entre los elementos del sistema Es lo que vamos a ver en esta semana, la simulación de sistemas.

1 Tipos de sistemas

La wikipedia nos define de esta manera un sistema: Un sistema (del latín systema, y este del griego sýstēma 'reunión, conjunto, agregado') es *"un objeto complejo cuyas partes o componentes se relacionan con al menos*

alguno de los demás componentes".

Existe toda una rama de las ciencias dedicada al estudio de los sistemas pero nosotros no lo necesitamos en este nivel, lo que necesitamos para nuestras simulaciones es ser capaces de representar y modelizar los sistemas que son más comunes a la psicología.

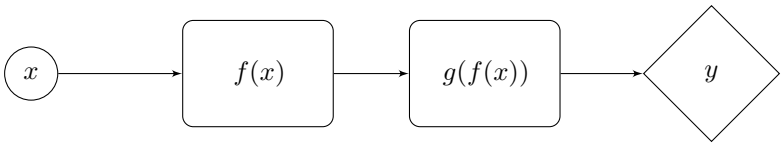
En esta semana vamos a preocuparnos de la modelización de los sistemas y sus procesos, sin insistir mucho en el proceso de crear o conseguir los datos de entrada que se necesitan para la simulación. Hasta ahora estos datos de entrada han sido los generadores de distribuciones aleatorias de R, pero dependiendo de la simulación podrán ser mucho más complejos. Ya lo iremos viendo en semanas sucesivas.

Los símbolos que vamos a utilizar para los nodos son los siguientes:

- círculo: input.
- rectángulo: proceso.
- rombo: output.

1.1 Sistemas con múltiples procesos

Vamos a hacer un poco más complejo el esquema inicial del que hemos partido:



Ahora el proceso en vez de acotarse a un único nodo de procesamiento tenemos varios nodos: el resultado del primero es la entrada del segundo. A veces estos nodos se pueden agregar en un único nodo por ejemplo:

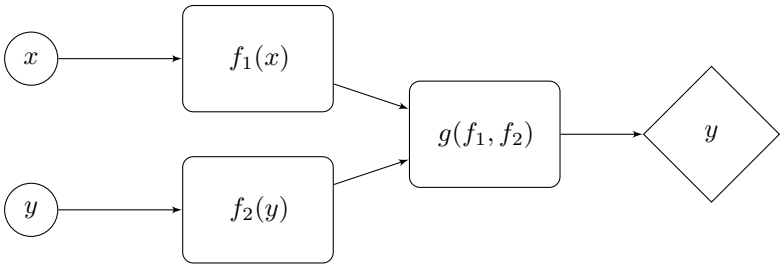
- $f(x) = 2x$
- $g(x) = \sqrt{f(x)}$

Se podrían unir en un único nodo de procesamiento: $\sqrt{2x}$, otras veces no interesa hacerlo, porque pertenecen a procesos sustantivamente diferenciados o porque su integración es más compleja. Cuando los nodos son muchos y con relaciones complejas hay que tener en cuenta el **orden de precedencia de los nodos**, así por ejemplo, para implementar el nodo $g(x)$ debemos estar seguros que se ha calculado previamente el proceso $f(x)$.

Un ejemplo de este tipo de procesos sería el sistema visual temprano humano. Tenemos un conjunto de entradas que son los estímulos visuales (imágenes) que se van procesando secuencialmente en distintas etapas: respuesta de los fotoreceptores, filtrado en frecuencia, detección de bordes. etc.

1.2 Sistemas con múltiples entradas, procesos y/o salidas

El esquema secuencial anterior puede generalizarse a tantos nodos de entrada, proceso y salida como necesitamos, como por ejemplo en la figura siguiente que ejemplificaría un procesamiento en paralelo. No hay mayor complejidad en la implementación que la que se ha explicado en el punto anterior, simplemente comprobar que el modelo es coherente y se puede calcular, y que respetamos la precedencia de los procesos. Cuando existen múltiples entradas. como en la figura, es necesario comprobar si el conjunto de entrada x tiene el mismo número de casos que la entrada y, ya que habrá un nodo ($proceso$: $g(f_1(x), f_2(y))$ que tendrá que integrarlos..

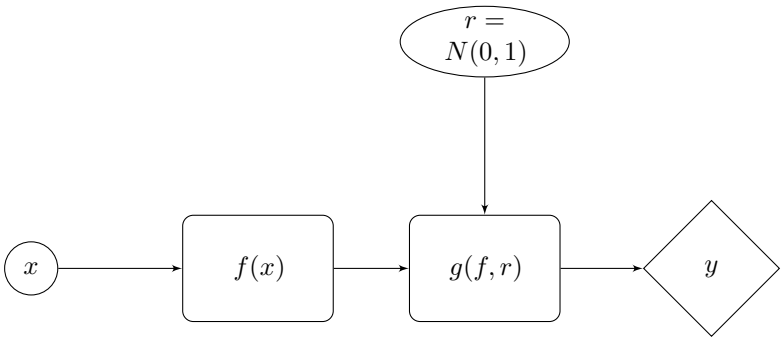


Este tipo de modelos son los habituales en simulación y son equivalentes a los diagramas de flujo utilizados en ecuaciones estructurales (SEM); en realidad son lo mismo, una manera de representar mediante grafos las relaciones existentes entre los componentes de un sistema o modelo. Si en SEM es necesario incluir los pesos de las conexiones de los componentes (porque siempre son un modelo lineal) aquí no es estrictamente necesario, aunque puede hacerse si clarifica. Como ejemplo podemos imaginar cualquier teoría psicológica en la que la autoestima influye en el rendimiento y el aprendizaje, que a su vez influye en la satisfacción, etc.

1.3 Sistemas deterministas vs sistemas estocásticos

Los sistemas anteriores son puramente deterministas, en el sentido de que dada una entrada, o conjunto de entradas, la salida siempre es conocida **e invariante**. Lo único que hace el sistema es calcular un conjunto de ecuaciones y dar el resultado. Es el ejemplo que hemos puesto del sistema visual, dada una entrada la salida del sistema es siempre la misma, no puede ser diferente.

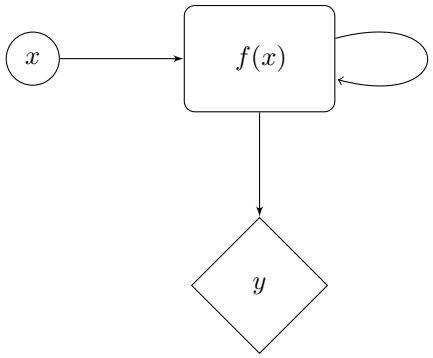
En los sistemas estocásticos se introduce la probabilidad como resultado de un proceso, por ejemplo tirar una moneda o que se produzca un resultado según una determinada distribución de probabilidad, y por tanto el resultado a una entrada x no necesariamente tiene que ser el mismo siempre. Hay diferentes formas de explicitar que se trata de un sistema estocástico, aquí voy a utilizar la siguiente: Cuando hay un proceso estocástico al generador aleatorio lo representamos con una elipse que va conectado con el nodo que utiliza ese componente aleatorio.



Un ejemplo de este tipo puede ser cualquier proceso en el que haya que tomar una decisión, por ejemplo, en detección de señales la respuesta estará mediada por un componente aleatorio. O la respuesta a una pregunta tipo test no es absolutamente determinista.

1.4 Sistemas estacionarios (o estáticos) y sistemas dinámicos

Una última división que es interesante hacer es entre sistemas estacionarios o estáticos y sistemas dinámicos. Los primeros son aquellos en los que el procesamiento se produce sin la variable tiempo, es decir, todos los procesos son cohetaneos (aunque no lo sean realmente sí lo son a efectos de cálculo). Los sistemas dinámicos son aquellos en los que alguna parte del sistema (o toda) cambia al fluir el tiempo.



Un ejemplo de sistema dinámico es la meteorología, en el que el sistema depende del momento anterior del propio sistema. O la epidemiología, en el que los casos infectados van cambiando según avanza el tiempo.

En esta semana trabajaremos con sistemas estáticos y en las siguientes semanas pasaremos a trabajar con sistemas dinámicos que son bastante más complejos.

2 Vectorización

Por vectorizar se entiende la estrategia de convertir los bucles de nuestro código en operaciones con vectores y matrices. En general, en la mayoría de lenguajes de programación modernos, las operaciones de álgebra lineal están optimizadas para que sean más rápidas y eficientes, así que utilizarlas hace que nuestro código sea mucho más rápido y reduce drásticamente los tiempos de ejecución. La idea es huir, todo lo posible, de bucles y loops y sustituirlos por operaciones vectoriales.

R, por su diseño, es bastante bueno y rápido realizando operaciones vectorizadas, y de hecho, mucha de la sintaxis que utilizamos habitualmente están vectorizadas, como la mayoría de las funciones básicas: sum, mean, sd, etc.

Aunque muchas de las veces sí es posible vectorizar las operaciones no todos los cálculos son susceptibles de ser vectorizados. Además, vectorizar puede ser incompatible con otras estrategias de optimización, como la paralelización por lo que habrá que estudiar y decidir qué estrategia, o combinación de estrategias, nos resulta más útil para nuestro propósito. Vectorizar casi siempre requiere de matemáticas complicadas, y resulta difícil sin una formación sólida en matemáticas. Otras veces no compensa por complejidad o para mantener la coherencia y el estilo del código.

En general, como hemos dicho durante el curso, las demandas de recursos suelen estar bastante por debajo de nuestros sistemas físicos (*hardware*) así que no necesitamos obsesionarnos con un código especialmente optimizado. O dicho de otra manera, los recursos computacionales de los que disponemos hoy en día suelen ser tales que no es imprescindible optimizar al máximo nivel el código. Con un código subóptimo vamos a poder llevar a buen puerto nuestras tareas de simulación, por eso a partir de ahora, la norma en este curso será:

> *Si puedes vectorizar el código, hazlo, y si no puedes, no te agobies y sigue trabajando con bucles.*

Por supuesto hay áreas de la computación moderna como el *deep learning, machine learning, IA,* etc. en los que sería inviable realizar los proyectos sin código adecuadamente vectorizado.

2.1 Funciones vectorizadas en R

En R casi todas las funciones están vectorizadas, ofreciendo unos tiempos de ejecución mucho más eficientes que un código no vectorizado. Ejemplos de funciones vectorizadas son casi todas: sum, colsum, rowsum, mean, var, sd, todas las comparaciones implícitas, operaciones con los vectores, etc. Vamos a ver su eficacia con algunos ejemplos de código:

```r
k <- 1000000

# Creando vectores
system.time({
  x <- 1:k
})
```

```
##    user  system elapsed
##       0       0       0
```

```r
system.time({
  x <- vector(length=k)
  for (i in 1:k) x[i] <- i
})
```

```
##    user  system elapsed
##    0.04    0.00    0.04
```

```r
# Operaciones con vectores

system.time({
  y <- x*2
})
```

```
##    user  system elapsed
##   0.002   0.000   0.003
```

```r
y <- vector(length=k)
system.time({
  for (i in 1:k) y[i] <- x[i]*2
})
```

```
##    user  system elapsed
##   0.043   0.003   0.047
```

```r
# Calcular la media

x <- runif(k, 0, 100)

system.time({
  mean(x)
})
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```

```r
y <- vector(length=k)
system.time({
  sum = 0
  for (i in 1:k) sum <- sum + x[i]
  sum/k
})
```

```
##    user  system elapsed
##   0.024   0.000   0.024
```

```r
# Comparaciones logicas
x <- runif(k, 0, 100)
```

```
system.time({
  x[x<50]
})
```

```
##    user  system elapsed
##   0.006   0.000   0.006
```

```
system.time({
  for (i in 1:k) x[i]<50
})
```

```
##    user  system elapsed
##   0.022   0.000   0.022
```

Como hemos visto en los ejemplos se obtienen tiempos significativamente menores utilizando funciones vectorizadas que utilizando versiones basadas en bucles. Eso es porque el código de R tiene implementadas a muy bajo nivel estrategias de cálculo vectorial que las hace muy rápidas.

2.2 Álgebra lineal en R

En R las operaciones con matrices y vectores en R son las siguientes, para vectores:

```
# Sigo la convencion de usar minusculas para vectores y mayusculas para matrices


# creacion de dos vectores
(a <- 1:5)
```

```
## [1] 1 2 3 4 5
```

```
(b <- 6:10)
```

```
## [1]  6  7  8  9 10
```

```
a + b # suma de vectores
```

```
## [1]  7  9 11 13 15
```

```
a * b # resta de vectores
```

```
## [1]  6 14 24 36 50
```

```
a / b # division de vectores
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444 0.5000000
```

```
t(a)  # traspuesta de un vector
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
```

```
t(t(a)) # traspuesta de la traspuesta
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
## [5,]    5
```

```
a*2 # vector por constante
```

```
## [1]  2  4  6  8 10
```

```
a+2 # vector mas constante
```

```
## [1] 3 4 5 6 7
```

```
# En vectores da igual que esté traspuesto, la operación es siempre by-element
a * t(b)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    6   14   24   36   50
```

```
a * b
```

```
## [1]  6 14 24 36 50
```

```
# producto interno de vectores
sum(a*b)
```

```
## [1] 130
```

Para matrices

```
# creación de dos matrices para los ejemplos
A <- matrix(1:25, ncol=5) # creando una matriz
B <- matrix(26:50, ncol=5) # creando una matriz

A * 2 # matriz por escalar
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2   12   22   32   42
## [2,]    4   14   24   34   44
## [3,]    6   16   26   36   46
## [4,]    8   18   28   38   48
## [5,]   10   20   30   40   50
```

```
t(A) # traspuesta de matriz
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
```

```
A + B # suma de matrices
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   27   37   47   57   67
## [2,]   29   39   49   59   69
## [3,]   31   41   51   61   71
## [4,]   33   43   53   63   73
## [5,]   35   45   55   65   75
```

```
A %*% B # multiplicacion de matrices, ojo: recuerda cuando se podian multiplicar
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] 1590 1865 2140 2415 2690
## [2,] 1730 2030 2330 2630 2930
## [3,] 1870 2195 2520 2845 3170
## [4,] 2010 2360 2710 3060 3410
## [5,] 2150 2525 2900 3275 3650
```

```
# %*% se refiere a operaciones matriciales y * a calculos elemento a elemento (by-element)

# OJO: No es lo mismo A %*% B que A*B

A %*% a  # multiplicacion matricial
```

```
##      [,1]
## [1,]  215
## [2,]  230
## [3,]  245
## [4,]  260
## [5,]  275
```

```
A * a    # multiplicacion by-elements
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    4   14   24   34   44
## [3,]    9   24   39   54   69
## [4,]   16   36   56   76   96
## [5,]   25   50   75  100  125
```

```
diag(A) # diagonal de una matriz
```

```
## [1]  1  7 13 19 25
```

```
A <- matrix(runif(9, -1, 1), ncol=3) # crea una matriz aleatoria
B <- solve(A) # inversa de una matriz (solo posible si: cuadradas y no singulares)
A %*% B # matriz identidad
```

```
##               [,1]          [,2]         [,3]
## [1,]  1.000000e+00  0.000000e+00 2.775558e-17
## [2,] -6.938894e-17  1.000000e+00 2.775558e-17
## [3,]  1.110223e-16 -1.110223e-16 1.000000e+00
```

```
diag(1, 3) # crea una matriz diagonal de unos de 3 x 3
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

2.3 Sistemas de ecuaciones con álgebra

En el fondo los modelos estacionarios no son más que un sistema de ecuaciones que pueden ser resultas en forma matricial. Por ejemplo, cuando tenemos un sistema como el del principio en el que hay una entrada, un proceso y una salida; y este proceso es del tipo lineal $y = \beta_0 + \beta_1 x$ puede ser reescrito matricialmente como

$$y = x\beta_1 + \beta_0$$

donde x es un vector fila y β_1 y β_0 son dos escalares, y puede ser fácilmente implementado en R en un única sentencia:

```
x <- 1:5
b0 <- 2
b1 <- 3
```

```
x*b1+b0
```

```
## [1]  5  8 11 14 17
```

Cuando tenemos un proceso que es una combinación lineal de las variables de entrada $x_1, x_2, ..., x_k$, que a su vez están ponderadas por $\beta_0, \beta_1, ..., \beta_k$, como en cualquier modelo lineal $y = \beta_0 + \beta_1 x_1 + ... + \beta_k x_k$ puede ser reescrito matricialmente:

$$y = XB + \beta_0$$

donde X es la matriz de entrada y B es la matriz de coeficientes.

```
X <- matrix(1:15, ncol=3)
B <- c(1, 2, 3)
b0 <- 100

X %*% B + b0
```

```
##      [,1]
## [1,]  146
## [2,]  152
## [3,]  158
## [4,]  164
## [5,]  170
```

O incluso se puede utilizar una manera más condensada en la que se incluye β_0 dentro de la matriz B incluyendo una columna de unos al principio de X:

$$y = XB$$

```
X <- cbind(rep(1,5), matrix(1:15, ncol=3))
B <- c(100, 1, 2, 3)

X %*% B
```

```
##      [,1]
## [1,]  146
## [2,]  152
## [3,]  158
## [4,]  164
## [5,]  170
```

Como se ha visto se pueden ir traduciendo las relaciones entre los elementos del modelo en operaciones vectoriales. Incluso en aquellos procesos que introducen un componente aleatorio se pueden generar inicialmente los valores según la distribución aleatoria e introducirlos en el cálculo de las matrices. Por ejemplo, si tenemos un modelo de detección de señales muy simple en el que la respuesta de un sujeto (V/F) depende del valor de entrada más un componente aleatorio, podemos generar a priori todos los valores aleatorios y luego utilizarlos:

```
k=10000
x <- sample(1:10, k, replace=T) # generamos la entrada

system.time({
  # vectorizado
  r <- rnorm(k,0,2) # generamos el componente aleatorio
  y <- x+r > 5 # modelo
})
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```

```
system.time({
  y <- vector(length=k)
  # con bucles
  for (i in 1:k) y[i] <- x[i]+rnorm(1,0,2) > 5
})
```

```
##    user  system elapsed
##   0.014   0.000   0.014
```

Como se ve en los resultados la versión vectorizada es mucho más rápida que la versión basada en bucles.

2.4 Los *"falsos amigos"* de la vectorización

En R el estilo de código tiende a huir del uso de loops y a escribir la sintaxis de una manera más compacta y funcional. Una familia de funciones que permiten esto son las *apply* por ejemplo: apply, lapplay, sapply, etc. La idea general es aplicar una función, o un conjunto de líneas de código, a elementos determinados de una estructura de datos. La sintaxis para apply es la siguiente:

apply(X, MARGIN, FUN, ...)

donde X es una matriz de f x c, MARGIN indica si la función se debe aplicar a las filas (MARGIN = 1) o a las columnas (MARGIN = 2) y FUN es la función, o código que se quiere aplicar. Con un ejemplo:

```
# una matriz de 5 x 3
m <- matrix(sample(1:10, 15, replace=T), ncol=3)

apply(m, 1, mean) # calcula 5 medias, una por fila
```

```
## [1] 6.666667 4.666667 6.000000 2.333333 2.000000
```

```
apply(m, 2, mean) # calcula 3 medias, una por columna
```

```
## [1] 3.0 4.4 5.6
```

```
# tambien puede aplicarse con tus propias funciones

foo <- function(v){
  return(sum(v)/length(v)) # una media
}

apply(m, 1, foo) # mismo resultado pero con foo
```

```
## [1] 6.666667 4.666667 6.000000 2.333333 2.000000
```

El uso de apply es muy interesante, y puede usarse no solo con matrices sino con otras estructuras de datos. Una lista completa de la familia apply puede ser la siguiente:

- apply()
- eapply()
- lapply()
- mapply()
- rapply()
- sapply()
- tapply()
- vapply()

Cada una de ellas está especializada en un tipo de estructura de datos, por ejemplo apply se utiliza con matrices, lapply con listas, sapplay para listas con una salida simplificada, tapply para data frames, etc. Las funciones tienen sus propias peculiaridades de uso y sus propios argumentos y no nos vamos a detener más en ellas, para aprender más sobre su uso se puede recurrir a los materiales de Jose Carlos Chacón o a los siguientes enlaces: R para principiantes, datacamp

La familia apply es muy utilizada en R y muy querida por la gente que trabaja en este lenguaje, permitiendo de una forma muy condensada escribir loops bastante complejos; sin embargo tienen un pequeño problema para el mundo de las simulaciones, y es que aunque "**parece**" que están vectorizadas en el fondo **la función apply no es más que un *wrap* de un bucle for**, y el resto de funciones de la familia **no son más que un wrap de apply**, ofreciendo todas ellas tiempos de ejecución iguales o peores que programando directamente los bucles. Por ejemplo, fíjate en el siguiente código:

```
# k <- 10000
#
# m <- matrix(rnorm(k^2), ncol=k)
#
# system.time({
#   v1 <- apply(m, 1, sum) # se aplica a las filas
# })
#
#
# v2 <- vector(length = k)
# system.time({
#   for(i in 1:k) v2[i] <- sum(m[i,])
# })
```

Los tiempos obtenidos por la función apply son mucho peores que los obtenidos por el bucle for.

Otra función similar a apply es la función replicate, que permite repetir un proceso o función n veces. Al igual que las anteriores replicate es solo un wrap de un bucle por lo que los tiempos suelen ser similares, y en general peores, a los obtenidos con un simple bucle.

```
# proc <- function(k){
#   v <- rnorm(k)
#   return(mean(v))
# }
#
#
# nrep = 10000
# nrand = 10000
#
# system.time({
#   replicate(nrep, proc(nrand))
# })
#
#
# v2 <- vector(length = nrep)
# system.time({
#   for(i in 1:nrep) v2[i] <- proc(nrand)
# })
```

Al igual que antes no hay una ventaja de la función replicate sobre el bucle.

Como se ha visto la utilización de esta funciones no suponen una mejora a la hora de realizar simulaciones y su uso está más relacionado con una cuestión de estilo. En mi modesta opinión las reglas de decisión para optar por una u otra opción serían las siguientes:

1. Si te sientes más cómodo empaquetando todo en funciones para hacer el código más compacto y próximo a una programación funcional entonces utiliza estas funciones.
2. Si tienes que hacer código muy eficiente (millones de repeticiones) no las utilices.

En mi caso yo no las suelo utilizar, porque provengo de C y me siento cómodo con los bucles (en C no hay diferencias de tiempo entre vectorizar y los bucles, ya que son lo mismo) y tiendo más a utilizarlos, pero la mayoría de los usuarios de R las prefieren por coherencia con el estilo de R. Como he explicado, salvo casos extremos de máxima eficiencia, se trata más una decisión personal.

Hay otra función, similar a las apply, que se llama **mclapply** pero que está fuera de la familia apply y que sí es realmente útil. Está función sí es un **"verdadero amigo"**, pero ya la estudiaremos en las siguientes semanas.

En general, y como conclusión, siempre que sea posible se debe optar por versiones vectorizadas de nuestro código, lo que a veces puede ser complejo y exceder nuestros conocimientos. Lo cierto es que en la mayoría de las situaciones podremos obtener resultados igualmente viables con un código sin vectorizar y con tiempos de ejecución razonables

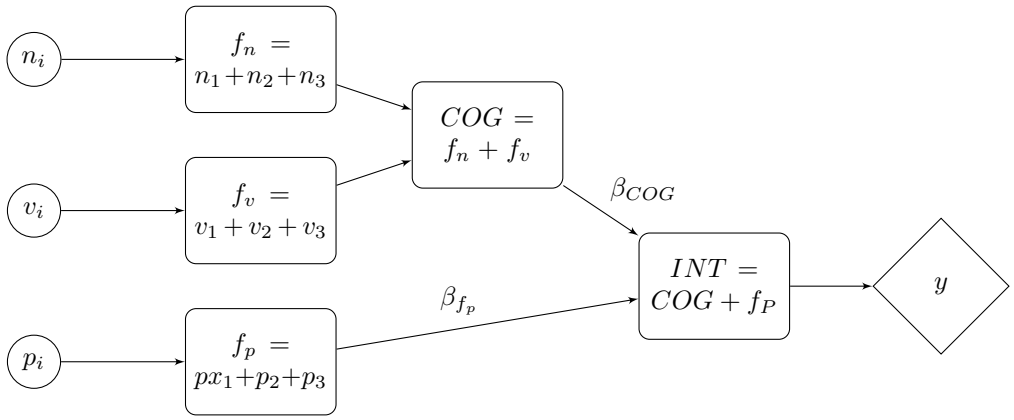
3 Tarea

En el fichero *datos.csv* se incluyen los datos de sujetos que han obtenido valores en un test cognitivo utilizado para estimar la inteligencia general. El modelo de puntuación es el siguiente:

1. Cada sujeto ha realizado 9 tareas, tres tareas numéricas (n1, n2 y n3), tres tareas verbales (v1, v2 y v3) y tres tareas perceptivas (p1, p2, p3). Estas tareas están diseñadas para estimar respectivamente los componentes numérico, verbal y perceptivo de un sujeto. En la estimación de cada componente siempre la primera tarea vale el 50%, la segunda el 30% y la tercera el 20%.
2. Se hipotetiza la existencia de una componente cognitivo (COG), que es la suma ponderada de los componentes verbal y numérico (pesos de 0.5 y 1.2 respectivamente).
3. La inteligencia general de una persona (INT) está determinada por la suma ponderada del componente cognitivo (COG) y el perceptivo. La ponderación está determinada por β_{COG} y β_{f_p} respectivamente, aunque ambos **son desconocidos**.
4. El nivel de actuación de un sujeto en una tarea dada (y) se considera que es una función logística de INT.

$$y = \frac{1}{1 + e^{-INT}}$$

Este modelo puede verse en la siguiente figura:



1. Simule el modelo anterior y obtenga los valores de y' para cada sujeto, donde y' indica que son los valores predichos por su modelo y no la realidad. Cuando un coeficiente del modelo es desconocido se fija a 1.

2. Calcule el ajuste entre sus valores y' y los valores reales de los sujetos y
3. Estime por un procedimiento de **fuerza fruta** los valores de β_{COG} y β_{f_p} que hacen mejor el ajuste entre y' e y.
4. Escriba un informe en .pdf incluyendo toda la información que considere relevante y con las conclusiones oportunas.
5. Reescriba su código lo más vectorizado que sea posible.