Introducción a la Investigación en Matemática Aplicada
Prácticas de Geometría Computacional

José Luis Bravo Trinidad

Breve introducción a R

R es un software para cálculo estadístico y gráfico. Proporciona, entre otras cosas, un lenguaje de programación, gráficos de alto nivel, *interfaces* para otros lenguajes de programación y depuración de errores. En esta pequeña introducción sólo veremos los conceptos esenciales que necesitamos para el resto del curso.

Ejecuta el programa R. Te aparecerá un mensaje de bienvenida y un símbolo >, donde puedes escribir. Esta es la consola del programa, cualquier orden que introduzcas a la derecha del > se ejecutará al pulsar la tecla de *Enter*.

Vamos a usarlo como calculadora. Escribe las siguientes líneas (pulsa *Enter* detrás de cada línea):

```
34+13
143*(341+123)
100^20
```

Hasta aquí funciona como una calculadora normal[1]. También podemos guardar el resultado de una operación en memoria. Escribe:

```
a<-34+53
```

Ahora a tiene el valor 87. Para ver qué contiene a basta escribir a y pulsar intro. Se dice que a es una variable porque permite guardar valores y cambiarlos. Por ejemplo:

```
a<-3*6
a
```

Ahora a valdría 18. Podemos usar a en cualquier sitio en el que podríamos usar un número, por ejemplo:

```
a+5
```

También podemos modificar su valor introduciendo usando a la vez el valor anterior:

```
a<-a+5
a
```

[1] No hay que olvidarse nunca de poner $*$ para la multiplicación y de los paréntesis.

Aquí lo que hace el programa es calcular el resultado a la derecha de la flecha y guardarlo en a.

Se pueden usar todas las variables que queramos, siempre teniendo en cuenta que el nombre debe comenzar por una letra (y es preferible evitar símbolos que no sean letras o números porque algunos no se pueden utilizar).

```
numero<-34
otronumero<-0.5
resultado<-numero*otronumero
```

Algunos nombres de variable no se pueden utilizar, por ejemplo *pi* es el número π y no se debe usar como nombre de variable.

Funciones

R tiene muchas funciones ya definidas, por ejemplo las trigonométricas (obviamente en radianes), las comparaciones, etc:

```
sin(pi/2)
max(4,39)
```

También podemos definir nosotros nuestras propias funciones. Para ello tendremos que elegir un nombre, poner $< -function$, a continuación los parámetros de entrada (en el ejemplo anterior sin tiene uno, pero max puede tener los que queramos). Escribimos entre llaves las operaciones que necesitemos para calcular el valor de la función. Dentro de esas llaves podemos utilizar los parámetros de entrada y cualquier instrucción que podamos utilizar en R, aunque no variables que hayamos creado antes. Una vez calculemos el valor lo devolveremos con *return* poniendo entre paréntesis el valor que queremos devolver.

```
nombrefuncion<-function(variable1,variable2){
a<-34
b<-3
sol<-variable1*a+variable2*b
return(sol)
}
```

Vectores

Por último, la estructura principal para guardar los datos se denomina vector. Hasta ahora siempre hemos guardado un único dato en una variable, pero en muchas ocasiones lo que tenemos es un montón de datos con los que queremos trabajar. Podemos guardar un montón de datos en una variable tipo vector del siguiente modo:

```
vector<-c(1,3,5,6,7)
vector
```

Ahora vector es una variable tipo vector que contiene los números 1, 3, 5, 6 y 7. Para acceder a cada número tenemos que indicar su posición:

```
vector[1]
vector[2]
vector[5]
```

Para saber cuántos datos guarda un vector se puede usar la orden "length":

```
length(vector)
```

Nos dice que hay cinco datos en el vector, guardados en las posiciones del 1 al 5.

En los vectores podemos hacer las operaciones habituales de suma, producto por un escalar, etc. Si aplicamos una función sobre un vector se aplica a cada posición.

```
v<-c(1,2,3,4)
w<-c(4,3,2,1)
v+w
2*v
sin(v*pi)
```

Otra operación habitual es definir el vector vacío, al cual le iremos añadiendo elementos.

```
v<-numeric()
v<-c(v,1)
v<-c(v,2)
v
v<-c(v,v)
v
```

Si lo que queremos es un vector con un patrón concreto (por ejemplo, los números del 1 al 10), se utiliza la orden **seq** (en ocasiones también se puede usar los dos puntos).

```
seq(0, 1, length.out=11)
seq(stats::rnorm(20))
seq(1, 9, by = 2)
seq(1, 9, by = pi) # si se pasa, se queda con el menor número
seq(1, 6, by = 3)
seq(1.575, 5.125, by=0.05)
seq(17) # equivalente a 1:17
seq(4,12) # equivalente a 4:12
```

Podemos quedarnos con los elementos de un vector que cumplan una cierta condición.

```
v<-c(5,2,6,3,1,8)
v[v>3]
```

Con la orden **order**, obtenemos el orden de menor a mayor de los elementos de un vector.

```
order(v)
v[order(v)]
```

La siguiente orden es más complicada. Piensa lo que hace.

```
v[order(v)[length(v)]]
```

Para obtener el máximo de un vector es más sencillo usar la orden **max**.

```
max(v)
```

A partir de un vector podemos generar una matriz.

```
m1<-matrix(c(1,2,3, 11,12,13), nrow = 2, byrow=TRUE)
m2<-matrix(c(1,2,3, 11,12,13), ncol = 2, byrow=FALSE)
```

Las matrices serán el tipo de dato que más utilizaremos. Igual que los vectores, se puede seleccionar un elemento.

```
m1[1,2]
```

Pero también una fila o una columna.

```
m1[1,]
m1[,2]
```

O también una submatriz, eligiendo algunas columas algunas filas.

```
m<-matrix(1:100,ncol=10,byrow=TRUE)
m[seq(1,10,by=2),1:5]
```

Finalmente, podemos añadir filas o columnas a una matriz.

```
m1<-cbind(m1,c(4,2))
m1
m1<-rbind(m1,c(1,2,3,4))
m1
```

En general existen muchas maneras de leer una matriz desde un fichero, según el formato del mismo. R puede leer ficheros de texto, ficheros Excel, etc. Vamos a ver únicamente un ejemplo de lectura de un fichero generado por un GIS, con el que luego trabajaremos.

```
library(foreign)
municipios <- read.dbf("CENTROIDES.DBF")
```

Con los datos obtenidos podemos hacer las operaciones que nos interesen.

```
# Los códigos de los municipios extremeños empiezan
# por 6 y tiene 3 cifras detrás o por 10 y tienen 3 cifras detrás.
ex<-municipios[(municipios[,1]>=6000)&(municipios[,1]<=6999)|(municipios[,1]>=10000)&(munici
# Nos quedamos con los que tengan más de 10000 habitantes y los pintamos
ex10000<-ex[ex[4]>10000,]
```

Estructuras de control

En muchos casos dependiendo de ciertas condiciones una función tiene un valor u otro. Por ejemplo, el valor absoluto de una variable x es x si $x \geq 0$ y es $-x$ si $x < 0$. Para programarlo se utiliza el condicional:

```
absoluto<-function(x){
if(x>=0){
  absx<- x
}
else{
  absx<- -x
}
return(absx)
}
```

Si la condición que hay entre paréntesis detrás del "if" es cierta, se ejecutan las líneas dentro de las llaves a continuación. Si no se cumple la condición, se ejecutan las líneas que hay entre las llaves detrás del "else". La parte del "else" es opcional; si no se pone, no se hace nada cuando la condición no se cumple.

Un segundo tipo de estructura que necesitamos para las funciones es la iteración, es decir, que una variable vaya tomando secuencialmente una serie de valores y para cada valor hagamos una cosa. Por ejemplo, para sumar los números del 1 al 100:

```
suma<-0
for(i in 1:100){
  suma<-suma+i
}
suma
```

La primera instrucción es para que nos ponga a cero la variable suma. Entre paréntesis detrás del for se incluye la variable que queremos que vaya tomando valores secuencialmente (i) y entre qué valores se mueve (en este caso, entre 1 y 100). Las instrucciones que estén en las llaves las irá ejecutando para cada valor de i, es decir, i tomará el valor 1 y hará $suma < -suma + 1$, con lo que suma pasa a valer 1. Después i toma el valor 2 y $suma < -suma + 2$ valdrá tres (uno que valía suma y dos más que hemos sumado) y así sucesivamente hasta llegar al valor 100 de i.

Para trabajar con un vector podemos o bien ir recorriendo sus posiciones:

```
suma<-0
for(i in 1:5){ # La variable i irá tomando el valor de cada posición del vector
  suma<-suma+vector[i] # vamos sumando cada valor para obtener la suma total
}
suma
```

o bien usar funciones que se apliquen sobre vectores:

```
sum(vector)
mean(vector)
```

Naturalmente, también podemos definir nosotros funciones que reciban vectores y devuelvan valores o vectores:

```
maximo<-function(v){
n<-length(v) # el número de datos que guarda el vector
maximoprovisional<-v[1] # tomamos como máximo provisional el primer elemento
for(i in 2:n){ # recorremos cada uno de los restantes elementos
  if(v[i]>maximoprovisional){ # si encontramos un elemento mayor que el mayor que hemos vist
    maximoprovisional<-v[i] # tomamos ese como nuevo máximo
  }
}
return(maximoprovisional) # Cuando hemos recorrido todos los elementos, tenemos el máximo
}
```

1 Envolvente convexa

Abre el archivo "convexhull.R" con un editor de texto. Por otra parte ejecuta el programa R.

Comencemos con la función para calcular el ángulo que forma un vector. Para ello, escribiremos la función "ang" en el R:

```
ang<-function(x,y){
  if(x>0){
    return(atan(y/x)) # Si x>0, usa la determinación del ángulo correcta (entre$
  }
  else{
    return(pi+atan(y/x)) # Si x<0, usa la determinación del ángulo opuesto, lue$
  }
}
```

Hay varias formas para hacerlo. Una, escribe en el R:

```
source("ruta/convexhull.R")
```

Donde *ruta* es la ruta donde está guardada "convexhull.R". Una segunda opción más fácil para principiantes es copiar del archivo "convexhull.R" y pegar diréctamente sobre el "R".

Carga también los datos iniciales y dibújalos escribiendo:

```
x<-c(0.1,1.3,1.2,0.4,4.4,2.3,3.7)
y<-c(2.1,3.3,0.2,4.4,2.4,1.4,1.5)
plot(x,y)
```

Esto crea dos vectores, uno con las coordenadas x de los puntos (que hemos llamado x) y otro con las y. Al escribir $plot(x, y)$ va cogiendo la primera coordenada de x como x y la primera de y como y y pinta ese punto, después la segunda con la segunda, etc. Vamos a ver qué ángulo forma el vector $(0.1, 2.1)$ con la horizontal. para ello sólo tenemos que escribir:

```
ang(0.1,2.1)
```

También podríamos haber escrito

```
ang(x[1],y[1])
```

Es decir, ángulo que forma el vector que comienza en $(0, 0)$ y termina en coordenada x la primera posición de x ($x[1]$) y coordenada y la primera posición de y ($y[1]$). Prueba para otros vectores.

Vamos a cargar ahora la función "aladerecha" (siguiendo los pasos de arriba).

```
aladerecha<-function(x1,y1,x2,y2,x,y){
ux<-x2-x1 # coordenada x del vector entre (x1,y1) y (x2,y2)
uy<-y2-y1
contador<-0
for(i in 1:length(x)){ # Para cada punto (x[i],y[i])
  vx<-x[i]-x1  # Vector entre (x1,y1) y (x[i],y[i])
```

```
  vy<-y[i]-y1
  if(ux*vy-vx*uy<0){ # Podemos ver la posición con el producto vectorial
    contador<-contador+1
  }
}
 return(contador)
}
```

La función anterior cuenta los puntos "a la derecha" de la recta que pasa por $(x1, y1)$ y $(x2, y2)$ como si nos situásemos en $(x1, y1)$ y mirásemos a $(x2, y2)$.

Vamos a probarla escribiendo:

$$aladerecha(0, 0, 5, 5)$$

Ahora cargamos las dos funciones "todosizquierda" y "convexhull".

```
todosizquierda<-function(x,y){
li<-numeric()
lj<-numeric()
angulo<-numeric()
for(i in 1:length(x)){
  for(j in 1:length(x)){
    if(i!=j){
      if(aladerecha(x[i],y[i],x[j],y[j],x,y)==0){
        li<-c(li,i)
        lj<-c(lj,j)
        angulo<-c(angulo,ang(x[j]-x[i],y[j]-y[i]))
      }
    }
  }
}
listaparejas <- matrix(c(li,lj,angulo),ncol=3,byrow=FALSE)
return(listaparejas)
}
convexhull<-function(x,y){
lista<-todosizquierda(x,y)
lista<-lista[order(lista[ ,3]), ]
puntos <- matrix(c(c(x[lista[,1]],x[lista[1,1]]),
                   c(y[lista[,1]],y[lista[1,1]])),ncol=2,byrow=FALSE)
return(puntos)
}
```

Ahora vamos a comparar con el segundo método.

Carga las funciones:

```
encontrarposicion<-function(vpos,x,y,k){
  for(i in 2:length(vpos)){
```

```
    pendiente<-ang(x[vpos[i]]-x[vpos[i-1]],y[vpos[i]]-y[vpos[i-1]])
    if(ang(x[k]-x[vpos[i]],y[k]-y[vpos[i]])>pendiente){
      return(i)
    }
  }
  return(length(vpos)+1)
}
convexhull2<-function(x,y){
puntos<-matrix(c(x,y),ncol=2,byrow=FALSE) # guardamos todos los puntos en una matriz para or
puntos<-puntos[order(x),] # los ordenamos según la coordenada x
x<-puntos[,1]
y<-puntos[,2]
envsup<-c(1,2) # El primer segmento será el formado por los dos primeros puntos
for(i in 3:length(x)){
  pos<-encontrarposicion(envsup,x,y,i) # Nos devuelve la posición de la envolvente convexa c
  envsup<-c(envsup[1:pos-1],i)
}
puntos<-matrix(c(x[envsup],y[envsup]),ncol=2,byrow=FALSE)
return(puntos)
}
```

Ejecutamos "convexhull2" para el ejemplo x, y haciendo:

```
plot(x,y)
envolvente<-convexhull2(x,y)
lines(envolvente[,1],envolvente[,2])
```

Para ver qué hace exactamente "convexhull2" vamos a ir viéndola paso a paso. En primer lugar, volvemos a tomar el ejemplo

```
x<-c(0.1,1.3,1.2,0.4,4.4,2.3,3.7)
y<-c(2.1,3.3,0.2,4.4,2.4,1.4,1.5)
plot(x,y)
```

Ahora, ejecutamos las cinco primeras líneas de "convexhull2", es decir, escribimos

```
puntos<-matrix(c(x,y),ncol=2,byrow=FALSE)
puntos<-puntos[order(x),]
x<-puntos[,1]
y<-puntos[,2]
envsup<-c(1,2)
x
y
envsup
```

Lo siguiente que haría la función es entrar en el bucle, e i tomar el valor 3. Así que ejecutamos

```
pos<-encontrarposicion(envsup,x,y,3)
pos
envsup[1:pos-1]
envsup<-c(envsup[1:pos-1],3)
envsup
```

Como se ve, el punto 3 se añade en la posición correspondiente que nos ha indicado *pos*.

Ahora pasaríamos al valor 4 de i. Ejecutamos

```
pos<-encontrarposicion(envsup,x,y,4)
pos
envsup[1:pos-1]
envsup<-c(envsup[1:pos-1],4)
envsup
```

Es decir, *pos* nos ha devuelto la posición en la que tenemos que colocar el punto 4, lo que hacemos, borrando el punto 3.

Ejercicios envolvente convexa

Ejercicio 1.1. Carga el archivo "CENTROIDES.DBF". Localiza el municipio en el que naciste (o en el que vives o uno que te guste especialmente). Obtén cuál es su identificador y guarda en un vector la línea de datos de ese municipio.

Ejercicio 1.2. Crea un vector con las distancias del municipio seleccionado en el apartado anterior a cada uno de los pueblos de Extremadura (distancia en línea recta).

Ejercicio 1.3. Crea una matriz que en la primera columna tenga el código de cada pueblo y en la segunda la distancia al municipio escogido por tí.

Ejercicio 1.4. Obtén los códigos de los 9 pueblos más cercanos al escogido.

Ejercicio 1.5. Represéntalos en el mapa.

Ejercicio 1.6. Calcula la envolvente convexa de esos puntos y represéntala en el mapa.

Hasta aquí son los ejercicios obligatorios para la sesión. De los siguientes apartados, elige uno y haz los ejercicios que puedas o propón tú un ejercicio que te parezca interesante.

1.1 Opción A (sin programación)

Ejercicio 1.7. Utiliza la segunda función para generar la mitad superior de la envolvente convexa y represéntala.

Ejercicio 1.8. Haz una simetría respecto al eje x de los puntos (cambia y por $-y$).

Ejercicio 1.9. Calcula la envolvente convexa de los puntos después de la simetría con la segunda función.

Ejercicio 1.10. Haz una simetría respecto al eje x de la envolvente convexa del apartado anterior. Represéntala junto a la calculada en el apartado 1.

Ejercicio 1.11. Repite el proceso anterior para todos los pueblos de Extremadura.

1.2 Opción B (con programación)

Ejercicio 1.12. Programa "convexhull3" para que devuelva la mitad inferior de la envolvente convexa.

Ejercicio 1.13. Programa "convexhull" para que devuelva la envolvente convexa entera por el segundo método.

Ejercicios intersección de líneas

Ejercicio 1.14. Cargar en R las funciones de intersección de líneas y los ejemplos que vienen en las mismas. Probar con el ejemplo.

Ejercicio 1.15. Añadir dos segmentos más de modo que el número de intersecciones aumente en al menos cuatro.

Ejercicio 1.16. Generar una nube de 10 puntos entre 0 y 1. Generar su envolvente convexa.

Ejercicio 1.17. Generar una segunda nube de 12 puntos. Generar su envolvente convexa.

Ejercicio 1.18. Calcular la intersección de las dos envolventes convexas.

1.3 Opción A (sin programación)

Ejercicio 1.19. Carga los datos de REGIONES.DBF. En el fichero hay varios campos. El primero es el código de la región, el segundo la coordenada x de cada punto y el tercero la y.

Separa los datos de cada región en una variable diferente.

Ejercicio 1.20. Representa en una gráfica las tres regiones.

Ejercicio 1.21. Calcula los puntos de intersección de las regiones dos a dos y represéntalos en la gráfica.

1.4 Opción B (con programación)

Ejercicio 1.22. Crea una función que reciba dos regiones y devuelva la región intersección de ambas (asumiendo que es única).

1.5 Opción C (con programación)

Ejercicio 1.23. Crea una función que reciba dos regiones y devuelva las regiones definidas por la intersección de ambas.

Ejercicios Diagrama de Voronoi

Carga las funciones para poder calcular el diagrama de Voronoi y los datos del ejemplo de municipios.

Ejercicio 1.24. Elije los 20 municipios de mayor población de Extremadura y obtén su diagrama de Voronoi. Dibuja dicho diagrama.

Ejercicio 1.25. Elije el tercer municipio de mayor población (Mérida) y en el gráfico anterior cambia el color de la región correspondiente a Mérida.

Ejercicio 1.26. Elije uno de los vértices del diagrama de Voronoi de los 20 municipios de mayor población. Añádelo a la lista de 20 municipios (invéntate el código y la población).

Ejercicio 1.27. Calcula el diagrama de Voronoi de los 20 municipio más el punto añadido. Dibujalo sobre el diagrama de los 20 municipios en otro color.

1.6 Opción A (sin programación)

Vamos a dar una estimación de la densidad de población. Para ello tomamos el punto $p = (205000, 4330000)$. Puedes dibujarlo con points(205000,4330000,pch=16, col="red")

Ejercicio 1.28. Estima su "densidad de población" como la media de las poblaciones de los ocho municipios más próximos a p.

Ejercicio 1.29. Para mejorar la estimación haremos lo siguiente: Calcula los ocho municipios más próximos al punto p. Calcula la población de p como la suma de las poblaciones de los ocho municipios, ponderada por el inverso de su distancia a p. Es decir

$$\frac{\frac{1}{d_1}p_1 + \frac{1}{d_2}p_2 + \ldots + \frac{1}{d_8}p_8}{\frac{1}{d_1} + \frac{1}{d_2} + \ldots + \frac{1}{d_8}},$$

donde p_i es la población del municipio i en la lista de los ocho más cercanos y d_i su distancia a p.

Ejercicio 1.30. Calcula el diagrama de Voronoi de todos los pueblos de Extremadura.

Ejercicio 1.31. En el diagrama de Voronoi anterior, calcula la región en la que está p.

Ejercicio 1.32. Calcula las regiones vecinas (vecinos de Thiessen) a la que contiene a p.

Ejercicio 1.33. Calcula la población de p como la media entre las poblaciones de la región del diagrama de Voronoi en la que está y las poblaciones de las regiones vecinas (ponderadas todas ellas por el inverso de la distancia).

1.7 Opción B (con programación)

Ejercicio 1.34. Crea una función que reciba un punto p y devuelva una estimación de su población como la suma de las poblaciones de los ocho municipios más próximos, ponderada por el inverso de su distancia a p. Es decir

$$\frac{\frac{1}{d_1}p_1 + \frac{1}{d_2}p_2 + \ldots + \frac{1}{d_8}p_8}{\frac{1}{d_1} + \frac{1}{d_2} + \ldots + \frac{1}{d_8}},$$

donde p_i es la población del municipio i en la lista de los ocho más cercanos y d_i su distancia a p.

Ejercicio 1.35. Crea una función que reciba un punto p y una lista de municipios y devuelva una estimación de la población de p como la media entre las poblaciones de la región del diagrama de Voronoi en la que está y las poblaciones de las regiones vecinas (ponderadas todas ellas por el inverso de la distancia).

1.8 Opción C (con programación)

Ejercicio 1.36. Crea una función que reciba un punto p y una lista de municipos y devuelva la estimación natural de su población.

Ejercicios Algoritmo del pintor

1.9 Opción A (sin programación)

Ejercicio 1.37. Carga el mapa "decepcion10.txt" y proyéctalo en perspectiva isométrica.

Ejercicio 1.38. Proyecta el mapa anterior en planta, alzado y perfil.

Ejercicio 1.39. Proyecta en perspectiva isométrica sólo la región correspondiente al mar.

Ejercicio 1.40. Proyecta el mapa en perspectiva isométrica usando color verde para la montaña y azul para el mar.

1.10 Opción B (con programación)

Ejercicio 1.41. Crea una función que reciba $xr, yr, mr, pers$, con xr, yr vectores crecientes de n, m elementos, respectivamente, y mr una matriz de $n \times m$ y proyecte la superficie utilizando $pers$ como matriz de transformación.