

Programming for Data Science

Functions in R language

Marco Beccuti

Università degli Studi di Torino

Dipartimento di Informatica



Function in R

- We have already used several examples of functions:

`mean(x)` `sd(x)` `ggplot(data, ...)` `lm(y ~ x, ...)`

- Functions are typically written if we need to compute the same thing for several data sets;
- Functions have a **name** and a **list of arguments** or **input objects**. For example, the argument to the function `mean()` is the vector `x`;
- Functions can also have a list of **output objects** returned when the function is terminated;
- A function must be written and loaded into R before it can be used.

A simple function in R

- A simple function can be constructed as follows:

```
function_name=function(arg1,arg2,...){  
  command1  
  command2  
  output  
}
```

- You can define a function name;
- The `function` keyword specifies that you are writing a function;
- Inside `()` you can outline the input objects;
- The commands occur inside `{}`;
- The name of whatever output you want goes at the **end of the function**;
- Comments lines are denoted by `#`.

A simple function in R

- An example:

```
mysum=function(x,y){  
  x + y  
}
```

- This function is called `mysum`;
- It has two input arguments, called `x,y`.
- Whatever values are passed for `x` and `y` their sum will be computed and the result visualizes on the screen.
- The function must be loaded into R before being called.

A simple function in R

How to execute a new function:

- Write the function in a text editor;
- Copy the function in the R console.
Type `ls()` into the console: the function now appears;
- Call the function using:

```
> mysum(3, 4)
[1]7
```

```
> mysum(y = 3, x = 4)
[1]7
```

```
> mysum(y = c(3, 6), x = c(4, 4))
[1]7 6
```

- Store the result into a variable `sumXY`:

```
> sumxy = mysum(3, 4)
```

How to load a function from a file

- Command `source()` is used to read the file and execute/load the commands in the same sequence given in the file.

`source(file,echo ...)`

`file` : character string giving the pathname of the file;

`echo` : if TRUE, each expression is printed after parsing, before evaluation.

How to load a function from a file

- Command `source()` is used to read the file and execute/load the commands in the same sequence given in the file.
- Use a text editor to save the following function in the file "myfun1.r":

```
myfun=function(x,y,p){  
  k = (x + y) * p  
  return(k)  
}
```

- Use command `source()` to load the function from the file:

```
> source("myfun1.r")
```

A simple function in R

- An example:

```
myfun=function(x,y,p){  
  k = (x + y) * p  
  return(k)  
}
```

- Function `myfun` has 3 arguments;
- The command `return` specifies what the function returns, here the value of `k`;

```
> myfun(3, 4, 7)
```

```
> res = myfun(3, 4, 7) result is stored in res
```


A more complex function in R

- The following function returns several values in the form of a list:

```
myfun1=function(x){  
  the.mean = mean(x)  
  the.sd = sd(x)  
  the.min = min(x)  
  the.max = max(x)  
  return (list(mean = the.mean, stand.dev = the.sd,  
             minimum = the.min, maximum = the.max))  
}
```

A more complex function in R

- how to call `myfun1`:

```
> x = rnorm(10)
```

```
> res = myfun1(x)
```

```
> res
```

```
res
```

```
$mean
```

```
[1]0.29713
```

```
$stand.dev
```

```
[1]1.019685
```

```
$minimum
```

```
[1] - 1.725289
```

```
$maximum
```

```
[1]2.373015
```

Argument Matching in R

How does R know to match arguments?

Argument matching is done in a few different ways:

- The arguments are matched by their positions. The first supplied argument is matched to the first formal argument and so on.

```
> myfun(3, 4, 7)  x=3, y=4 and p=7
```

- The arguments are matched by name. A named argument is matched to the formal argument with the same name:

```
> myfun(y = 4, x = 3, p = 7)  x=3, y=4 and p=7
```

- Name matching happens first, then positional matching is used for any unmatched arguments.

Argument Matching in R

- Default values for some/all arguments can be specified:

```
myfun=function(x,y,p=10){  
  k = (x + y) * p  
  return(k)  
}
```

- If a value for the argument p is not specified in the function call, a value of 10 is used.

```
> l = myfun(3,4)  
> l  
[1]70
```

- If a value for p is specified, that value is used.

```
> l = myfun(3,4,2)  
> l  
[1]14
```

Exercises on functions

- 1 Write a function that when passed a number, returns the number squared, the number cubed, and the square root of the number;
- 2 Write a function that when passed a numeric vector, prints the value of the mean and standard deviation to the screen (Hint: use the `cat()` function in R.) and creates a histogram of the data;

Exercises on function

- Write a function that when passed a number, returns the number squared, the number cubed, and the square root of the number;

```
myfun2=function(x){  
  squared = x * x  
  cubed = x * x * x  
  root = sqrt(x)  
  return (list(squared, cube, root))  
}
```

Exercises on function

- Write a function that when passed a numeric vector, prints the value of the mean and standard deviation to the screen (Hint: use the `cat()` function in R.) and creates a histogram of the data;

```
myfun3=function(x,file="hist.png"){  
  cat(x,": standard deviation is",sd(x),"\n")  
  cat(x,": mean is",mean(x),"\n")  
  library(ggplot2)  
  ggplot(data.frame(x),aes(x)) + geom_histogram()  
}
```

Lexical scoping

- Variables are typically created inside a script or inside the function where they are also used;
- However, **free variables** may also be taken from the environment in which the function was defined;
- Can be useful, but also very confusing.

```
myfun=function(x){  
  res <- -x ^ power  
  return(res)  
}
```

```
> power = 3  
> l = myfun(3)  
> l  
[1]27
```


Lexical scoping

```
# Outside function
power <- 2
# Function definition
powfun <- function(x) {
  res <- x^power
  return(res)
}
# Calling the function
powfun(3)
```

```
# Outside function
power <- 2
# Function definition
powfun <- function(x) {
  res <- x^power
  return(res)
}
# Calling the function
powfun(3)
```

```
# Outside function
power <- 2
# Function definition
powfun <- function(x) {
  res <- x^power
  return(res)
}
# Calling the function
powfun(3)
```

if Statement

- **Conditional execution:** the if statement has the form:

```
if (condition){  
    expr1  
}  
else {  
    expr2  
}
```

Condition is evaluated and returns a logical value (i.e. TRUE or FALSE.)
If the condition is evaluated **TRUE**, *expr₁* is executed , otherwise *expr₂* is executed.

- Logical operators `&&`, `||`, `==`, `!=`, `>`, `<`, `>=`, `<=` are used as the conditions in the if statement.

if Statement: a simple example

- The following function gives a demonstration of the use of `if ... else`:

```
checkMyfunction=function(number){  
  if(number != 1) {  
    cat(number, "is not one \n")  
  }  
  else {  
    cat(number, "is one \n")  
  }  
}
```

```
> checkMyfunction(1)
```

```
1 is one
```

```
> checkMyfunction(2)
```

```
2 is not one
```

if Statement: a second simple example

- The following function gives a demonstration of the use of `&&` :

```
checkBetween=function(number){  
  if((number >= 1)&&(number <= 10)) {  
    cat(number, "is between one and ten \n")  
  }  
  else {  
    cat(number, "isn't between one and ten \n")  
  }  
}
```

```
> checkBetween(2)
```

```
1 is between one and ten
```

```
> checkMyfunction(12)
```

```
12 isn't between one and ten
```

Nested if Statements

- The following function gives a demonstration of the use of `if ... else if ... else`:

```
checkNum=function(number){  
  if(number == 0) {  
    cat(number, "is zero \n")  
  }  
  else if(number < 0) {  
    cat(number, "is negative \n")  
  }  
  else{  
    cat(number, "is positive \n")  
  }  
}
```

Exercises on function

- Write a function that when passed a numeric vector, prints the value of the mean and standard deviation to the screen (Hint: use the `cat()` function in R.) and creates a histogram of the data;

```
myfun3=function(x,file="hist.png"){  
  cat(x,": standard deviation is",sd(x),"\n")  
  cat(x,": mean is",mean(x),"\n")  
  if(!("ggplot2" %in% (.packages())))  
    library(ggplot2)  
  ggplot(data.frame(x),aes(x)) + geom_histogram()  
}
```

Exercises on function

- Write a function that when passed a numeric vector, prints the value of the mean and standard deviation to the screen (Hint: use the `cat()` function in R.) and creates a histogram of the data;

```
myfun3=function(x,file="hist.png"){  
  cat(x,": standard deviation is",sd(x),"\n")  
  cat(x,": mean is",mean(x),"\n")  
  if(!require(ggplot2)) install.packages('ggplot2')  
  library(ggplot2)  
  ggplot(data.frame(x),aes(x)) + geom_histogram()  
}
```

For loop

- To loop/iterate through a certain number of repetitions a **for** loop is used.
Its syntax is:

```
for (condition){  
  command_1  
  command_2  
  .....  
}
```

A simple example of a **for** loop:

```
MyLoop=function(x){  
  cumsum = rep(0, length(x))  
  if(!(is.numeric(x))) {  
    cat(x, "must be numeric \n")  
    return(cumsum)  
  }  
  cumsum[1] = x[1]  
  for(i in 2 : length(x))  
    cumsum[i] = cumsum[i-1] + x[i]  
  return(cumsum)  
}
```


For loop

- You can nest loops. In this cases indenting the code can be useful.

```
for (condition_1){  
  command_1  
  command_2  
  for(condition_2){  
    command_1  
    command_2  
  }  
}
```

- multiply nested `for` loops are generally avoided when possible in R because they can be quite slow.

For loop

- Compare using function `system.time()` the function `MyLoop()`

```
MyLoop=function(x){  
  cumsum = rep(0, length(x))  
  if(!is.numeric(x)) {  
    cat(x, "must be numeric \n")  
    return(cumsum)  
  }  
  cumsum[1] = x[1]  
  for(i in 2 : length(x))  
    cumsum[i] = cumsum[i-1] + x[i]  
  return(cumsum)  
}
```

and `cumsum()`. They have a different execution time.

```
> x = rnorm(1000000)  
> system.time(cumsum(x))  
> system.time(MyLoop(x))
```

For loop

- Compare using function `benchmark` the function `MyLoop2()`

```
MyLoop2=function(x){  
  if(!is.numeric(x)) {  
    cat(x,"must be numeric \n")  
    return(cumsum)  
  }  
  cumsum[1] = x[1]  
  for(i in 2 : length(x))  
    cumsum[i] = cumsum[i-1] + x[i]  
  return(cumsum)  
}
```

and `MyLoop()`. They have a different execution time.

```
> x = rnorm(1000000)
```

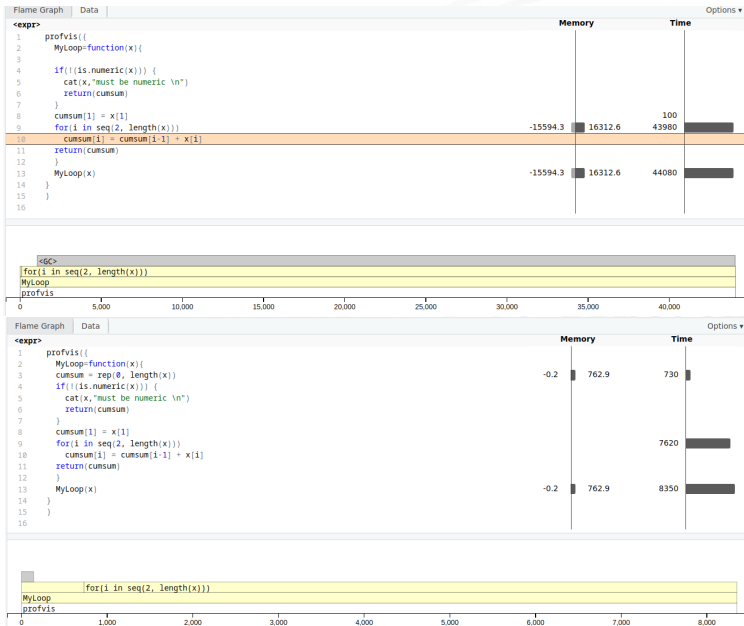
```
> benchmark(MyLoop(x), MyLoop2(x), replications = 5)
```

For loop

- Compare using function `benchmark` the function `MyLoop2()` with `MyLoop2()`
- Exploit `profvis` to do profiling:

```
profvis({MyLoop2=function(x){  
  if(!is.numeric(x)) {  
    cat(x,"must be numeric \n")  
    return(cumsum)  
  }  
  cumsum[1] = x[1]  
  for(i in 2 : length(x))  
    cumsum[i] = cumsum[i-1] + x[i]  
  return(cumsum)})  
}
```

For loop



While loop

- While loop can be used if the number of iterations required is not known beforehand;
- For example, if loop must continue until a certain condition is met.
- Its syntax is:

```
while (condition){  
  command_1  
  command_2  
  .....  
}
```

The loop continues while `condition == TRUE`.

While loop

- A simple example of a **while** loop:

```
MyLoop1=function(x){  
  cumsum = rep(0, length(x))  
  if(!(is.numeric(x))) {  
    cat(x,"must be numeric \n")  
    return(cumsum)  
  }  
  cumsum[1] = x[1]  
  i = 2  
  while(i <= length(x)){  
    cumsum[i] = cumsum[i-1] + x[i]  
    i = i + 1  
  }  
  return(cumsum)  
}
```

next, break, statements

- The **next** statement can be used to discontinue one particular iteration of any loop. Useful if you want a loop to continue even if an error is found (error checking);
- The **break** statement completely terminates a loop. Useful if you want a loop to end if an error is found.

```
MyLogNext=function(x){  
  for(i in 1 : length(x)){  
    if(x[i] <= 0) {  
      next  
    }  
    x[i] = log(x[i])  
  }  
  return(x)  
}
```

```
MyLogNext1=function(x){  
  for(i in 1 : length(x)){  
    if(x[i] <= 0) {  
      break  
    }  
    x[i] = log(x[i])  
  }  
  return(x)  
}
```


Exercises on loops and functions

- Create a function `find_value()`, which takes as input a number b and a vector m , and returns first occurrence of b in m ;
- Create a function `find_all_value()`, which takes as input a number b and a matrix m , and returns all the occurrences of b in m ;
- Create a function `translate()`, which takes as input a numeric vector c and returns a string vector f such that $f[i] = "P"$ iff $c[i] > 0$ otherwise $f[i] = "N"$.

Exercises on loops and functions

- Create a function `find_value()`, which takes as input a number b and a vector m , and returns first occurrence of b in m ;

```
find_value=function(b,m){  
  if(length(m) < 1) {  
    cat("m size must be greater 0 \n")  
    return(-1)  
  }  
  ind = 1  
  while(ind <= length(m)){  
    if(m[ind] == b)  
      return(ind)  
    ind = ind + 1  
  }  
  return(-1)  
}
```

Exercises on loops and functions

- Create a function `find_all_value()`, which takes as input a number b and a matrix m , and returns all the occurrences of b in m ;

```
find_all_value=function(b,m){  
  f = NULL  
  for(row in 1 : dim(m)[1]){  
    for(col in 1 : dim(m)[2]){  
      if(m[row, col] == b)  
        if(length(f) == 0)  
          f = list(c(row, col))  
        else  
          f = list(f, c(row, col))  
      }  
    }  
  }  
  return(f)  
}
```

Exercises on loops and functions

- Create a function `find_all_value()`, which takes as input a number b and a matrix m , and returns all the occurrences of b in m ;

```
find_all_value=function(b,m){  
  f = NULL  
  for(row in 1 : dim(m)[1]){  
    for(col in 1 : dim(m)[2]){  
      if(m[row, col] == b)  
        f = c(f, c(row, col))  
    }  
  }  
  return(f)  
}
```

Exercises on loops and functions

- Create a function `translate()`, which takes as input a numeric vector c and returns a string vector f such that $f[i] = "P"$ iff $c[i] > 0$ otherwise $f[i] = "N"$.

```
translate=function(m){  
  f = NULL  
  if(!is.numeric(x)) {  
    cat(x,"must be numeric \n")  
    return(f)  
  }  
  for(ind in 1 : length(m)){  
    if(m[ind] > 0)  
      f = c(f,"P")  
    else  
      f = c(f,"N")  
  }  
  return(f)  
}
```

How to replace a for loop with apply functions

There are three basic ways to loop over a vector:

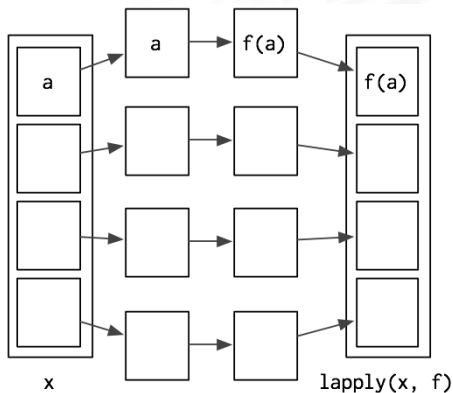
- loop over the elements: `for (x in xs)`
- loop over the numeric indices: `for (i in seq_along(xs))`
- loop over the names: `for (nm in names(xs))`

that can be implemented using **lapply**/**sapply**:

- `lapply(xs, function(x) {})`
- `lapply(seq_along(xs), function(i) {})`
- `lapply(names(xs), function(nm) {})`

How to replace a for loop with apply functions

The `lapply()` function takes a function, applies it to each element of the input, and returns the results in the form of a list.



How to replace a for loop with apply functions

An example showing the conversion:

```
NormalLoop=function(x){  
  for(i in 1 : length(x)){  
    if(x[i] <= 0) {  
      x[i] = abs(x[i])  
    }  
    else  
      x[i] = log(x[i])  
  }  
  return(x)  
}
```



?

How to replace a for loop with apply functions

An example showing the conversion:

```
NormalLoop=function(x){  
  for(i in 1 : length(x)){  
    if(x[i] <= 0) {  
      x[i] = abs(x[i])  
    }  
    else  
      x[i] = log(x[i])  
  }  
  return(x)  
}
```



```
lapply(x, function(x){  
  if(x <= 0) {  
    x = abs(x)  
  }  
  else  
    x = log(x)  
})
```

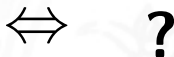
How to replace a for loop with apply functions

An example showing the conversion:

```
> x = c(-1, 20, 4)
```

```
> names(x) = c("o", "p", "o")
```

```
NameLoop=function(x){  
  for(i in 1 : length(x)){  
    if(names(x)[i] == "o") {  
      x[i] = x[i] + 10  
    }  
    else  
      x[i] = x[i] + 100  
  }  
  return(x)  
}
```



How to replace a for loop with apply functions

An example showing the conversion:

```
> x = c(-1, 20, 4)
```

```
> names(x) = c("o", "p", "o")
```

```
NameLoop=function(x){  
  for(i in 1 : length(x)){  
    if(names(x)[i] == "o") {  
      x[i] = x[i] + 10  
    }  
    else  
      x[i] = x[i] + 100  
  }  
  return(x)  
}
```



```
lapply(seq_along(x), function(i){  
  if(name(x)[i] == "o") {  
    x[i] = x[i] + 10  
  }  
  else  
    x[i] = x[i] + 100  
})
```

How to replace a nested loops with apply functions

An example showing the conversion:

```
NestedLoop=function(x,y){  
  el = rep(FALSE, length(x) * length(y))  
  Shared = matrix(el, nrow = length(x))  
  for(i in 1 : length(x)){  
    for(j in 1 : length(y)){  
      if(x[i] == y[j]) {  
        Shared[i,j] = TRUE)  
      }  
    }  
  }  
  return(Shared)  
}
```

⇔ ?

How to replace a nested loops with apply functions

An example showing the conversion:

```
NestedLoop=function(x,y){
```

```
  el = rep(FALSE, length(x) * length(y))
```

```
  Shared = matrix(el, nrow = length(x))
```

```
  for(i in 1 : length(x)){
```

```
    for(j in 1 : length(y)){
```

```
      if(x[i] == y[j]) {
```

```
        Shared[i,j] = TRUE
```

```
      }
```

```
    }
```

```
  }
```

```
  return(Shared)
```

```
}
```



```
sapply(y, function(y){
```

```
  sapply(x, function(x){
```

```
    if(x == y) {
```

```
      return(TRUE)
```

```
    }
```

```
  else
```

```
    return(FALSE)
```

```
  })
```

```
})
```

In this case a better solution is to use outer product:

```
>outer(x,y,"==")
```

Exercises on loops and functions

- Use **DeSolve** package to solve the following ODE system between 0 to 10.

$$\frac{dx_1}{dt} = -3x_1 + 4x_2 + 3.5x_3$$

$$\frac{dx_2}{dt} = +3x_1 - 14.5x_2$$

$$\frac{dx_3}{dt} = +10.5x_2 - 3.5x_3$$

$$x_1(0) = 100$$

$$x_2(0) = 10$$

$$x_3(0) = 1.0$$

and plot the evolution of x_1, x_2, x_3 over the time.

Exercises on loops and functions

- Use **deSolve** package to solve the following ODE system between 0 to 10.

$$\frac{dx_1}{dt} = -3x_1 + 4x_2 + 3.5x_3$$

$$\frac{dx_2}{dt} = +3x_1 - 14.5x_2$$

$$\frac{dx_3}{dt} = +10.5x_2 - 3.5x_3$$

$$x_1(0) = 100$$

$$x_2(0) = 10$$

$$x_3(0) = 1.0$$

and plot the evolution of x_1, x_2, x_3 over the time.

```
> install.packages("deSolve")
```

```
> library(deSolve)
```

```
> ?lsode
```

lsode(y, times, func, ...)

- *y* is the initial (state) values for the ODE system;
- *time* is time sequence for which output is wanted;
- *func* is an R-function that computes the values of the derivatives in the ODE system.

Exercises on loops and functions

- Use **deSolve** package to solve the following ODE system between 0 to 10.

$$\frac{dx_1}{dt} = -3x_1 + 4x_2 + 3.5x_3$$

$$\frac{dx_2}{dt} = +3x_1 - 14.5x_2$$

$$\frac{dx_3}{dt} = +10.5x_2 - 3.5x_3$$

$$x_1(0) = 100$$

$$x_2(0) = 10$$

$$x_3(0) = 1.0$$

and plot the evolution of x_1 , x_2 , x_3 over the time.

```
> y = c(100, 10, 1.0)
```

```
> times = seq(0, 10, 0.1)
```

```
> funODE=function(t, x, parms){  
  dx1 = -3*x[1]+4*x[2]+3.5*x[3]  
  dx2 = +3 * x[1] - 14.5 * x[2]  
  dx3 = +10.5 * x[2] - 3.5 * x[3]  
  return(list(c(dx1, dx2, dx3)))  
}
```

```
> res=lsode(y,times,funODE,parms=0)
```

```
> colnames(res)=c("Time", "x1", "x2", "x3")
```


Exercises on loops and functions

- Use **deSolve** package to solve the following ODE system between 0 to 10.

$$\frac{dx_1}{dt} = -3x_1 + 4x_2 + 3.5x_3$$

$$\frac{dx_2}{dt} = +3x_1 - 14.5x_2$$

$$\frac{dx_3}{dt} = +10.5x_2 - 3.5x_3$$

$$x_1(0) = 100$$

$$x_2(0) = 10$$

$$x_3(0) = 1.0$$

and plot the evolution of x_1 , x_2 , x_3 over the time.

```
>gp=ggplot(data.frame(res),aes(x=Time))
```

```
>gp+geom_line(aes(y=x1),color="red")
```

```
>gp+geom_line(aes(y=x2),color="blue")
```

```
>gp+geom_line(aes(y=x3),color="green")
```

Exercises on loops and functions

- Find α and β which maximize $x_1(t) + x_2(t)$ with $t = 1$.

$$\frac{dx_1}{dt} = \alpha - x_1 + 4x_2 + \beta x_3$$

$$\frac{dx_2}{dt} = \alpha x_1 - 14.5x_2$$

$$\frac{dx_3}{dt} = +10.5x_2 - \beta x_3$$

$$x_1(0) = 100$$

$$x_2(0) = 10$$

$$x_3(0) = 1.0$$

with $10 \leq \alpha, \beta \leq 100$

You can use **GenSA** packages: Generalized Simulated Annealing for Global Optimization. It searches for global minimum of a very complex non-linear objective function with a very large number of optima

Exercises on loops and functions

```
> install.packages("GenSA")
```

```
> library(GenSA)
```

```
> ?GenSA
```

GenSA(*par*, *fn*, *lower*, *upper*, *control* = *list*(), ...)

- *par* initial vector values for the components to be optimized;
- *fn* is the function to be minimized;
- *lower*, *upper* bounds for components;
- *control* is a list that can be used to control the behavior of the algorithm.

Exercises on loops and functions

```
> p0 = c(20, 20)
```

```
> LB = c(10, 10)
```

```
> UB = c(100, 100)
```

```
> x0 = c(100, 10, 1.0)
```

```
> ObjF=function(p){
```

```
  Times = seq(from = 0, to = 1, by = 0.1)
```

```
  res = lode(x0, Times, funODE, parm = p)
```

```
  last = tail(res, 1)
```

```
  fn = -1 * (last[2] + last[3])
```

```
  return(fn)
```

```
}
```

```
> k=GenSA(p0,ObjF,LB,UB,control=list(max.time=5))
```

```
> k.par
```

```
> k.value
```

```
>funODE=function(t, x, parm){
```

```
  dx1 =
```

```
  -parm[1] * x[1] + 4 * x[2] + parm[2] * x[3]
```

```
  dx2 = +parm[1] * x[1] - 14.5 * x[2]
```

```
  dx3 = +10.5 * x[2] - p[2] * x[3]
```

```
  return(list(c(dx1, dx2, dx3)))
```

```
}
```

Exercises on loops and functions

- Find α and β which maximize $x_1(t) + x_2(t)$ with $t = 1$ varying the initial value for the components (i.e. **par** vector).

$$\frac{dx_1}{dt} = \alpha - x_1 + 4x_2 + \beta x_3$$

$$\frac{dx_2}{dt} = \alpha x_1 - 14.5x_2$$

$$\frac{dx_3}{dt} = +10.5x_2 - \beta x_3$$

$$x_1(0) = 100$$

$$x_2(0) = 10$$

$$x_3(0) = 1.0$$

with $10 \leq \alpha, \beta \leq 100$

Exercises on loops and functions

```
> LB = c(10, 10)
> UB = c(100, 100)
> x0 = c(100, 10, 1.0)

> y0 = lapply(seq_along(1 : 10), function(i){runif(2, 10, 100)})

> s = lapply(y0, function(y0){
  GenSA(par = y0, fn = ObjF, upper = UB, lower = LB, control = list(max.time = 5), x0 = x0)
})
```