Enhancing Target Functions

Gunther Krauss

23. May 2023

Why and How: Enhancing a Target Function

There are many methods for sensitivity analysis and optimisation (calibration) that take a target function, generate parameters and run the function with these parameters.

When performing sensitivity and optimisation tasks,

- we may impose some constraints on the parameters
- we might be interested not only in the final result, but also on all generated parameters to track how the method came to the result

Depending on the implementation of the sensitivity or optimisation method, this options might or might not be available. If the optimisation method does not offer a possibility to limit parameters or to track intermediate runs, then we have to add this functionality to our target function:

- checking parameters and return a high penalty value instead of real function value for parameters outside the boundaries
- recording parameters and results of each function run

Instead of doing this manually by writing the corresponding code inside the target function, we can use R to enhance our target function automatically.

Respecting boundaries

The function `enhanceWithBoundaries` takes as arguments

- our target function
- lower and upper bounds as vectors which have as many elements as our parameterset has

and returns a new function which checks first, if the parameters are within the boundaries. In this case, it calls our original target function. If the parameters are outside the boundaries, it returns Infitiy, i.e. the maximum possible error.

There are some additional optional arguments

- `penalty_value` we may change the penalty value to another value (e.g. -Inf)
- `boundary_fun` maybe we want to supply a more sophisticated function for checking if parameters are within boundaries
- `param_pos` can be used, if our target function takes the parameter set not as the first argument, but as the second or third etc.

```
enhanceWithBoundaries <- function(fun, l_bound, u_bound,
                                  penalty_value=Inf,
                                  boundary_fun=NULL,
                                  param_pos=1,
                                  ...) {
```

```
  function(...) {
    invalid <- FALSE
    if(!is.null(boundary_fun)) {
      invalid <- boundary_fun(...elt(param_pos))
    }
    else {
      invalid <- !all(l_bound <= ...elt(param_pos) & ...elt(param_pos) <= u_bound)
    }
    if(invalid) {
      penalty_value
    }
    else {
      fun(...)
    }
  }
}
```

Example

The function $f(x,y) = (x^2 + y^2 - 1)^2 + (x + y - 1)^2$ has two minima, at (0,1) and at (1,0). Let's assume that for some reasons a value for x smaller than 0.6 would not be acceptable.

```
target_fun <- function(p) (p[1]^2 + p[2]^2 -1)^2 + (p[1] + p[2] -1)^2
o1 <- optim(c(.7,.7), target_fun, control=list(reltol=1e-20))

round(o1$par,10)
```

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

```
round(o1$value,10)
```

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

When running an optimisation of our function, we get the minimum at (0,1), which we could not accept.

```
target_fun_bd <- enhanceWithBoundaries(target_fun,c(.6,-100),c(100,100))

o2 <- optim(c(.7,.7), target_fun_bd,  control=list(reltol=1e-20))
round(o2$par,10)
```

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

```
round(o2$value,10)
```

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

By imposing a boundary on our function, the optimisation process would give us only values with x bigger than 0.6

Recording all runs

When we want to record all parameters and function calls used by the sensitivity or optimisation method, we need some object to store this data.

For this we create a so called *Reference Class* in R. We name this class "FunctionRecorder". The class has a slot `runInfo` that holds the information for each run and some methods:

- `add` to add the information of a run to the `runInfo` list

- `get` and `get_as_df` to get the `runInfo` either as list or dataframe
- `reset` to clear the `runInfo`

And there is the method `enhanceWithRecorder` that takes our target function and returns a new function that takes the parameter set, runs our target function and stores parameters as well as the result in the function recorder.

The `enhanceWithRecorder` takes an optional argument `info_fun`. We can create a function that takes a parameter set and a result value and restructure them before being stored in the recorder. By default parameters and result are added as a vector. With an own function, we can put them e.g. in a named vector so that the recorded dataframes has our prefered column names.

```
FunctionRecorder <- setRefClass(
  "FunctionRecorder",

  fields=list(runInfo="list",temporaryInfo="list", count="numeric",temporarycount="numeric"),

  methods = list(
    add = function(x) {
      count <<- count + 1
      temporarycount <<- temporarycount + 1
      if(length(temporaryInfo)<temporarycount) {
        ls <- list()
        v <- rep(0.0,length(x))
        for(i in 1:1000) {
          ls[[length(ls)+1]]<-v
        }
        runInfo <<-c(runInfo,temporaryInfo)
        temporarycount <<- 1
        temporaryInfo <<- ls
      }
      temporaryInfo[[temporarycount]]<<-x
    },
    get = function() {
      c(runInfo, temporaryInfo)[1:count]
    } ,
    get_as_df = function() tryCatch({
      df <- data.frame(do.call(rbind, .self$get()),row.names = NULL)
      if(length(.self$get())>0 && is.null(names(.self$get()[[1]]))){
        names(df)<-c(paste0("P",1:(ncol(df)-1)),"Value")
      }
      df
    }),
    reset = function() {
      runInfo <<-list()
      temporaryInfo <<- list()
      temporarycount <<- 0
      count <<- 0
    },
    enhanceWithRecorder = function(fun,info_fun=\(p,r)c(p,r), ...) {
      .self$reset()
      function(...) {
        r <- fun(...)
        .self$add(info_fun(...,r))
        r
```

```
    }
  }
))
```

Notice: As the time to add elements to a list seems to increase with the lenght of the list, we add the information to a shorter temporary list, which we then add to the full list. This will speed up the adding process, if we have a large amount of runs recorded (>100000).

Example

We create a function recorder and enhance our initial target function before supplying it to the optimisation function.

```
FR <- FunctionRecorder$new()

target_fun_rec <- FR$enhanceWithRecorder(target_fun)

o3 <- optim(c(.7,.7), target_fun_rec,  control=list(reltol=1e-20))
round(o3$par,10)
```

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

```
round(o3$value,10)
```

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

After the optimisation, we can get the dataframe of all runs

```
runs3 <- FR$get_as_df()
head(runs3)
```

```
##     P1    P2     Value
## 1 0.70 0.700 0.1604000
## 2 0.77 0.700 0.2277724
## 3 0.70 0.770 0.2277724
## 4 0.63 0.770 0.1601040
## 5 0.56 0.805 0.1346976
## 6 0.56 0.735 0.1083921
```

Same procedure with the function that respects boundaries

```
target_fun_bd_rec <- FR$enhanceWithRecorder(target_fun_bd)

o4 <- optim(c(.7,.7), target_fun_bd_rec,  control=list(reltol=1e-20))
round(o2$par,10)
```

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

```
round(o2$value,10)
```

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

```
runs4 <- FR$get_as_df()

head(runs4)
```

```
##     P1    P2     Value
## 1 0.70 0.700 0.1604000
## 2 0.77 0.700 0.2277724
## 3 0.70 0.770 0.2277724
```

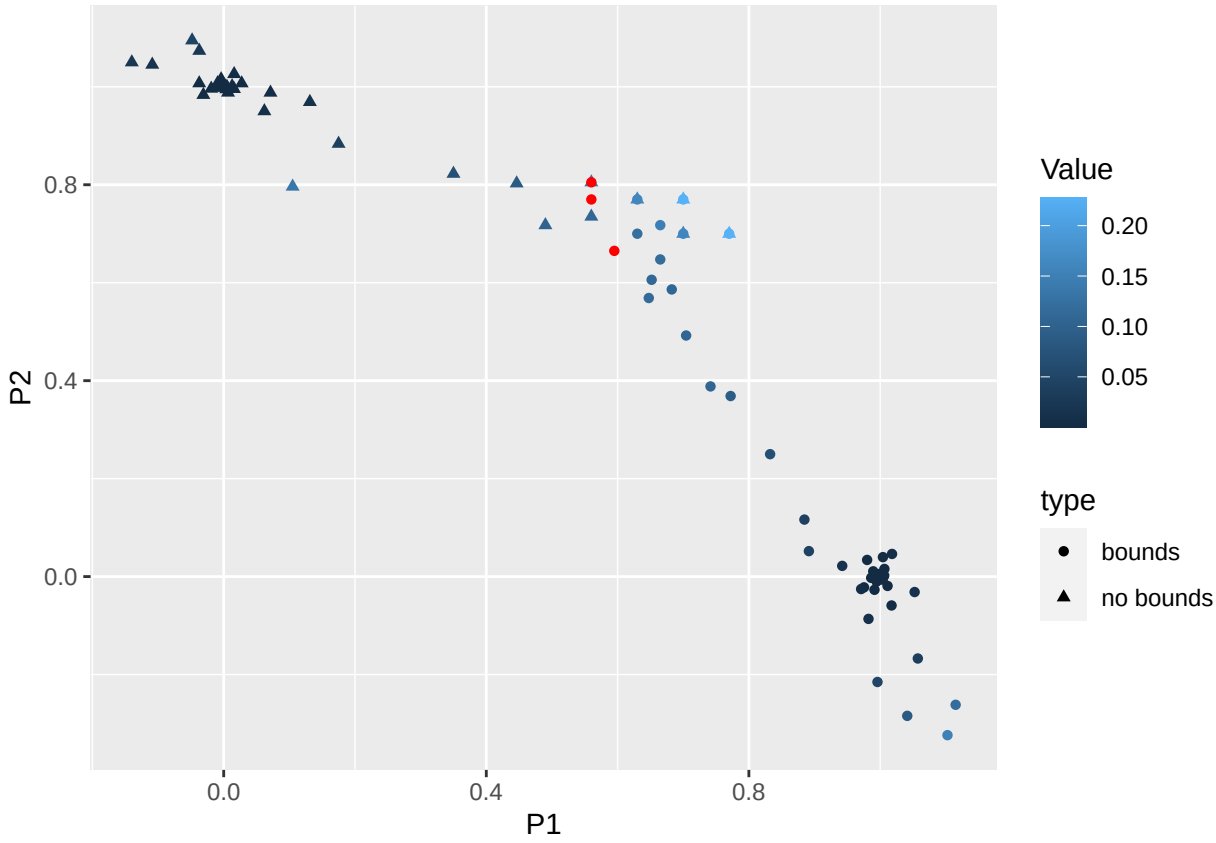
```
## 4 0.63 0.770 0.1601040
## 5 0.56 0.805       Inf
## 6 0.63 0.700 0.1216916
```

Notice the value `Inf` where P1 (i.e x) is smaller than 0.6:

Plotting the runs

```
runs3$type<-"no bounds"
runs4$type<-"bounds"
runs <- rbind(runs3,runs4)

library(ggplot2)
ggplot(runs) + geom_point(aes(x=P1,y=P2,colour=Value,shape=type)) +
  scale_colour_continuous(na.value="red")
```



The parameters for the non-bounded function move during optimisation towards (0,1), whereas those for bounded function are "punished" (red dots) if they move towards 0 on x-Axis. So the optimisation method will move the parameters towards the allowed area and then taking the road towards the other minimum at (1,0)

Performance test

Running pure function 1 million times

```
system.time(
  for(i in 1:1000000)
```

```
    target_fun(c(runif(1,.55,1),runif(1,0,1)))
)
```

```
##        User      System verstrichen
##        5.79        0.83        6.92
```

Running boundary check function 1 million times

```
system.time(
  for(i in 1:1000000)
    target_fun_bd(c(runif(1,.55,1),runif(1,0,1)))
)
```

```
##        User      System verstrichen
##        9.39        0.55       10.18
```

Running recording function 1 million times

```
FR$reset()
tm <- system.time(
  for(i in 1:1000000)
    target_fun_bd_rec(c(runif(1,.55,1),runif(1,0,1)))
)
tm
```

```
##        User      System verstrichen
##      156.58        0.83      160.11
```

Recording the result of 1 million runs adds an overhead of around 2.7 minutes. Considering the fact, that running usual models 1 million times will take rather hours than minutes, then the overhead of the recording will be neglectable for most cases.