Optimus Tutorial

Aleksandr B. Sahakyan

19 December 2015

Example 1: Finding coefficients for a polynomial function that best describes tha data.

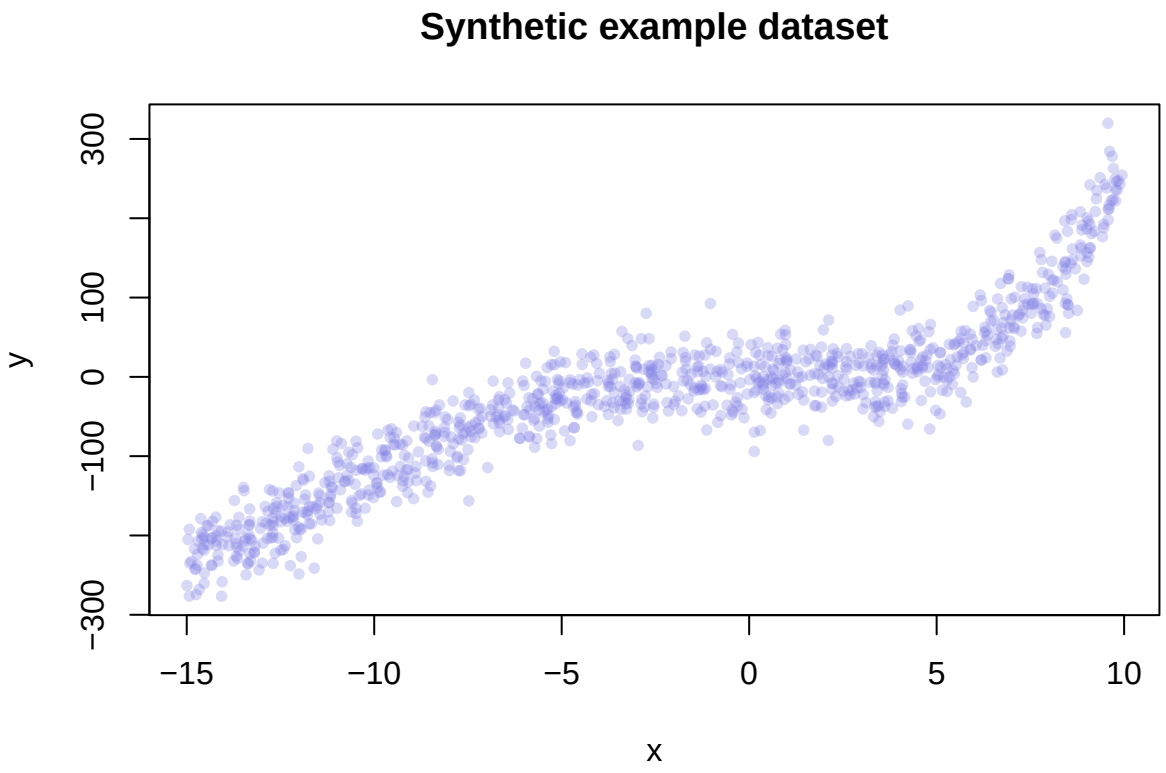
In this example, we shall use Optimus to find the coefficients of the polynomial function that is known to represent the observations y the best. This, of course, is an extremely simple task that can be addressed more robustly by least-squares linear model fitting, however, starting our way with this will focuse our attention on the organisation of the Optimus input, rather than the complexity of the task.

First of all, let us create some data for the example.

```
set.seed(845)
x <- runif(1000, min = -15, max = 10)
y <- -1 * x - 0.3 * x^2 + 0.2 * x^3 + 0.01 * x^4 + rnorm(length(x), mean = 0,
    sd = 30)
```

The good side of this noisy data generation is that we know what the original function is to describe it - $y = -1.0x - 0.3x^2 + 0.2x^3 + 0.01x^4$. Hence, we can check how well Optimus performs at finding the correct coefficients. The synthetic "real world" noisy data that we generated look like:

```
plot(main = "Synthetic example dataset", x = x, y = y, col = rgb(0.5, 0.5, 0.9,
    0.3), pch = 16, cex = 0.8)
```



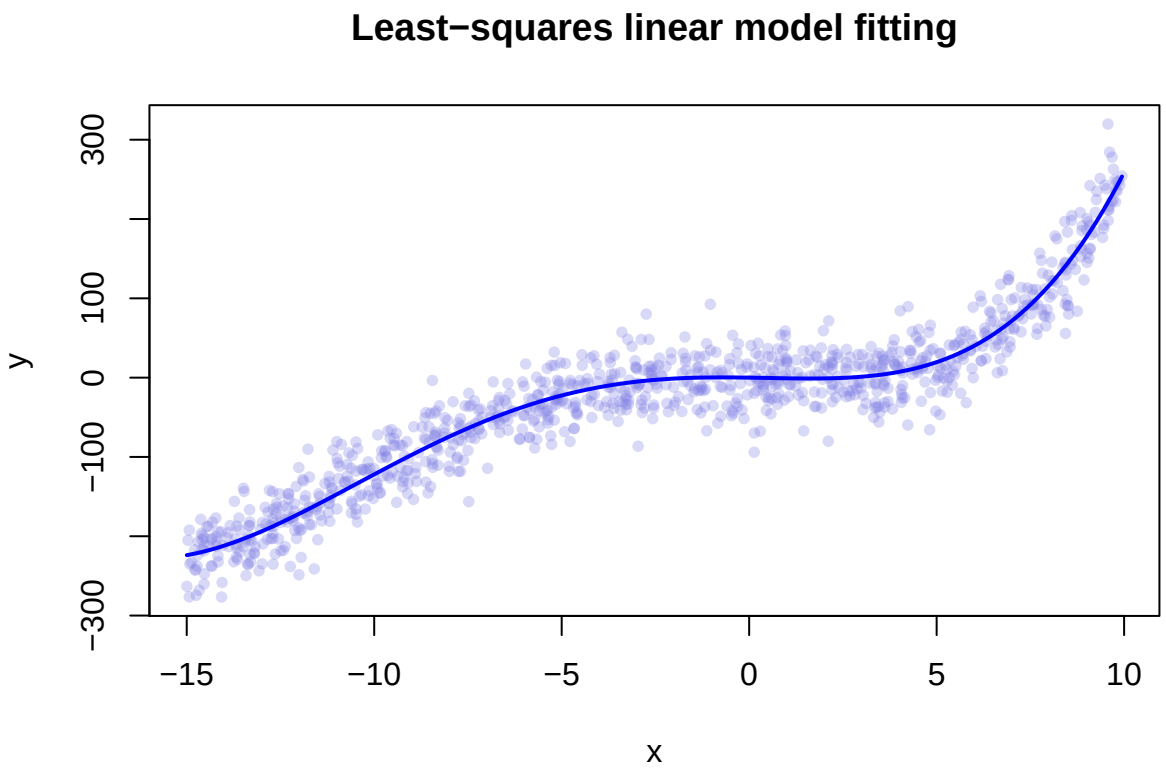
Before we turn to Optimus, let us see how the proper linear model fitting will do on the data.

```
lm.model <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + 0)
lm.model
```

```
##
## Call:
## lm(formula = y ~ x + I(x^2) + I(x^3) + I(x^4) + 0)
##
## Coefficients:
##        x    I(x^2)    I(x^3)    I(x^4)
## -0.74056  -0.30735   0.19777   0.00991
```

The least-squares linear model fitting for the coefficients to the known functional form is quite close to the original equation - $y = -0.741x - 0.307x^2 + 0.198x^3 + 0.010x^4$.

```
plot(main = "Least-squares linear model fitting", x = x, y = y, col = rgb(0.5,
    0.5, 0.9, 0.3), pch = 16, cex = 0.8)
lines(x = sort(x), y = predict(newdata = data.frame(x = sort(x)), object = lm.model),
    col = "blue", lwd = 2)
```



The RMSD between the observed data y and the linear model fitting outcome is:

```
y.pred <- predict(newdata = data.frame(x = x), object = lm.model)
sqrt(mean((y - y.pred)^2))
```

```
## [1] 28.82655
```

which is even a little bit better in describing the noisy data, as compared to the maximum possible RMSD based on the de-noised data:

```
y.realdep <- -1 * x - 0.3 * x^2 + 0.2 * x^3 + 0.01 * x^4
sqrt(mean((y - y.realdep)^2))
```

```
## [1] 28.85858
```

Now we can set up the files for the Optimus run. We shall use the model $k_1x + k_2x^2 + k_3x^3 + k_4x^4$ to fit the y observables based on the values for x. The dependent functions that are needed for setting up an Optimus run, are called by their mathematical notations in the method description in the Optimus manuscript.

First, we need to create an object with a name K, which stores the initial values for the parameter(s) to be optimised. K can be an object of any type. From a single numeric or character value to a vector of values or a data frame holding, say, Cartesian coordinates of a molecules to be optimised. The only requirement from K is that it should be something alterable (via a rule function r(), see below) and something that influences the outcome of another required model function - m() (see below). In this example, we have 4 coefficients to optimise from some random initial state. We can thus make K be a numeric vector of size 4. Let us start from all the components being 1.0, which, as entries in K, can be both named and unnamed. Though not the case here, the entry-named data for K can be essential for some models that specifically use coefficient names, for instance when a system of ODEs is used in the model function m().

```
K <- c(k1 = 1, k2 = 1, k3 = 1, k4 = 1)  # entries are named as k1, k2, k3 and k4
```

Second, we should create the function m for the model. The function m should be designed to operate on the whole set of parameter snapshot K and return the corresponding observable object V. Please note, that the size of K and V are not necessarily to match, depending on the nature of the model used. Operating on K is the only hard condition on m(), which can optionally take other arguments as well. In our situation, the function m() should operate on the provided instance of four coefficients (in the object K), and, additionally on the values x. It should then return a vector of observations V (to be compared with y target observations) of the same size as vector x. Any additional data required by the model, in our case an object with the set of 1000 x values, must be added to the function as an argument in addition to the compulsory K, and should also made available inside the main Optimus() function for the m() function to access it. For now, we shall just add x=x, as an additional argument for m(), turning to how to make x available from within Optimus() later in the tutorial.

```
m <- function(K) {
    V <- K["k1"] * x + K["k2"] * x^2 + K["k3"] * x^3 + K["k4"] * x^4
    return(V)
}
```

At this point, calling m(K=K, x=x) will return the predicted V set from the initial, non-optimal values for K, hence rather far from the target $V^{trg} = y$.

In this example, the optimisation goal is for the V model outcomes to come as close as possible to the target observations y, to be achieved by optimising the coefficients K. The object y holding the target values therefore also need to be specified and made available within the Optimus() function, just like x required, in this example, by the function m().

Now, we need to define how the performance of a given snapshot of coefficients K is to be evaluated. For Optimus run, this is done by specifying a function u(), which should necessarily require V model outcome object and act upon it. The output should have two components, $Q holding a single number of the quality of the K coefficients, and $E holding a (pseudo)energy for the given snapshot K. It is important that the returned (pseudo)energy value is lower for better performance/version of K, nevere vice-versa. The $Q

component of the u() function output is only used for plotting the optimisation process, and, if desired, can just repeat the value of the $E component.

For our example, the u() function will assess the agreement between the snapshot of predictions V and the complete set of real observables (target) y. We thus need a function that, in addition to taking the required V object as an argument, will also take the target y vectors, to evaluate the agreement between V and y and return a pseudo-energy value. Here, we can use RMSD between V and y as a measure of K snapshot quality ($Q). Since bettter agreement means better RMSD, it can be directly used as a pseudo-energy ($E), without putting a negative sign or performing some other mathematical operation on $Q.

```
u <- function(V) {
    Q <- sqrt(mean((V - y)^2))
    E <- Q  # For RMSD, <-> negative sign or other mathematical operation
    # is not needed.

    RESULT <- NULL
    RESULT$Q <- Q
    RESULT$E <- E
    return(RESULT)
}
```

And finally, we need to define the rule, by which the K coefficient vector is to be altered from one step to another. This is done by defining a rule function r() that must take K, and return an object equivalent to K, but with some alteration(s). In this example, for each snapshot of K, we shall randomly select one of its four coefficients, then either increment or decrement (chosen randomly) it by 0.0005, returning the altered set of coefficients.

```
r <- function(K) {
    K.new <- K
    # Randomly selecting a coefficient to alter:
    K.ind.toalter <- sample(size = 1, x = 1:length(K.new))
    # Creating a potentially new set of coefficients where one entry is altered
    # by either +move.step or -move.step, also randomly selected:
    move.step <- 5e-04
    K.new[K.ind.toalter] <- K.new[K.ind.toalter] + sample(size = 1, x = c(-move.step,
        move.step))

    ## Setting the negative coefficients to 0 (not necessary in this example,
    ## useful for optimising rate constants): neg.ind <- which(K.new < 0)
    ## if(length(neg.ind)>0){ K.new[neg.ind] <- 0 }

    return(K.new)
}
```

All the constructed objects (K) and functions (m, u, r), as well as the objects required by the custom version of the functions (x and y) should be defined within a single parameter file (OptMDL.R in our example). The users are free to define some dependencies as additional files (for example initial protein geometry for a Monte-Carlo optimisation), which should however be called from within the parameter file.

We can start creating our parameter file OptMDL.R by first saving there all our synthetic data for x and y, required by the functions m() and u().

```
write(c(paste("x <- c(", paste(x, collapse = ", "), ")", sep = ""), paste("y <- c(",
    paste(y, collapse = ", "), ")", sep = "")), file = "OptMDL.R")
```

Now, you can just copy-paste the above condtructed lines for K, m, u and r into the same file.

poly1 - 30.004

poly2 - 28.862 -0.378391710 -0.329750212 0.193165683 0.009887276

poly3 - 29.311

poly4 - 29.326

y <- -1.0*x* - *0.3*x^2 + 0.2*x^3 + *0.01*x^4

poly1 - 28.842

poly2 - 28.827 -0.737799493 -0.306879925 0.197691668 0.009901995

poly3 - 28.877

poly4 - 28.832

lm

$y = -0.741x - 0.307x^2 + 0.198x^3 + 0.010x^4$