Tutorial 1: Finding Coefficients for a Polynomial Function

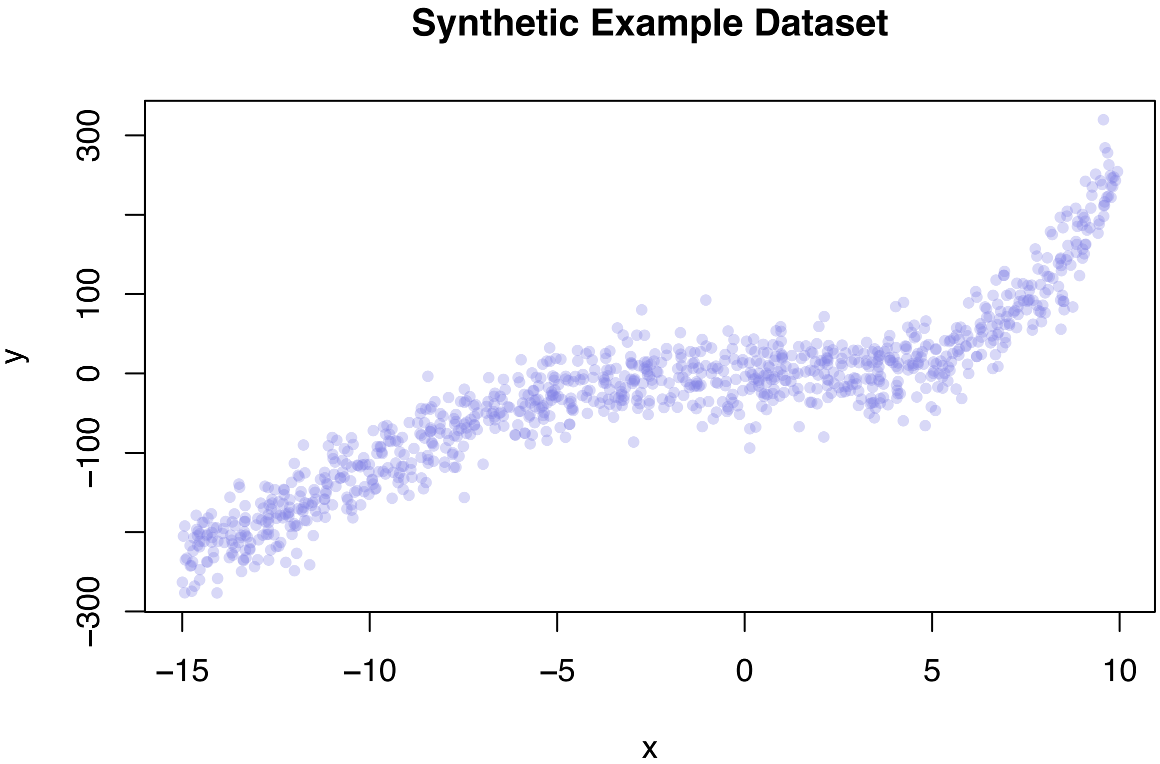
Problem Statement

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 a simple task that can be addressed more robustly by least-squares linear model fitting. However, by starting with this example, we shall focus 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 <- -1.0*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 the original function that describes 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 looks like this:

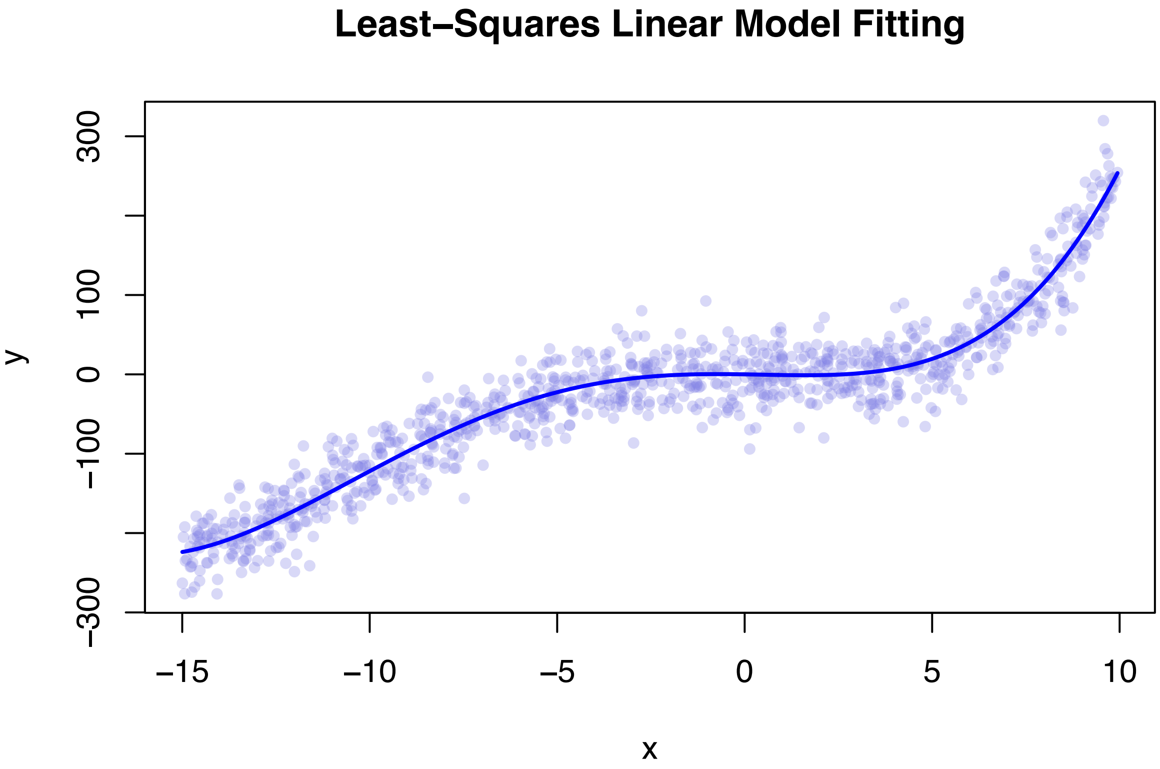


Before we turn to Optimus, let us see how the proper linear model fitting will perform using this data.

```
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, not surprisingly, rather close to the original equation $y = -0.741x - 0.307x^2 + 0.198x^3 + 0.010x^4$:



The root mean squared deviation (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 slightly better in describing the noisy data, as compared to the maximum possible RMSD based on the de-noised data:

```
y.realdep <- -1.0*x - 0.3*x^2 + 0.2*x^3 + 0.01*x^4
sqrt(mean((y-y.realdep)^2))
```

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

Defining Optimus Inputs

Now we can set up the inputs 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 given as inputs in the Optimus function call.

First, we need to create the essential object `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 molecule to be optimised. The only Optimus requirement from `K` is that it should be something alterable (*via* a rule function `r()`, see below) and should influence 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() in one of the tutorials.

```
K <- c(k1=1.0, k2=1.0, k3=1.0, k4=1.0) # 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 O. Please note that the size and shape of K and O are not necessarily to match, depending on the nature of the model used. Operating on K is **one of the hard conditions** on m(), which can optionally operate on other data 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 O (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 provided to the function in an input variable DATA, a list holding the additional data that must be accessed by m() and u() (see below). The variable DATA **must be provided** to Optimus in the function call, and m() **must take it** as an input. In the case that neither m() nor u() require additional data, the two functions should still be created such that they take a variable DATA as an input, and the variable DATA passed to Optimus will be set to NULL).

```
# Generating an object that is then to be passed to DATA argument of Optimus().
# No need to call it DATA, as soon as it is passed to the DATA argument of Optimus().
DATA   <- NULL
DATA$x <- x
DATA$y <- y

# Generating the m() function
m <- function(K, DATA){
  x <- DATA$x
  O <- K["k1"]*x + K["k2"]*x^2 + K["k3"]*x^3 + K["k4"]*x^4
  return(O)
}
```

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

In this example, the optimisation goal is for the O 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 needs to be specified and given as an input to the main Optimus() function (as a constituent entry in the DATA argument), just like x was supplied, as 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, this is done by specifying a function u(), which should **necessarily** take as inputs O (the output of m()) and the variable DATA. 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 **must be lower for better performance/version** of K, never 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 O and the complete set of real observables (target) y. Here, we can use RMSD between O and y as a measure of K snapshot quality (Q). Since better 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(O, DATA){
  y <- DATA$y
  Q <- sqrt(mean((O-y)^2))
  E <- Q # For RMSD, <-> negative sign or other mathematical operation
         # is not needed.
  RESULT   <- NULL
  RESULT$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 analogous 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
  move.step <- 0.0005

  # 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:
  K.new[K.ind.toalter] <- K.new[K.ind.toalter] +
                                    sample(size=1, x=c(-move.step, move.step))

  return(K.new)
}
```

All the constructed objects (K) and functions (m, u, r), as well as the data required by m() and u() (stored in the variable to be passed to DATA) should be defined in an R session and given to Optimus as inputs. The users are free to define some dependencies as additional files (for example: initial protein geometry for a Monte-Carlo optimisation), which should be called from within the function definitions.

Acceptance Ratio Simulated Annealing Optimus Run

Having constructed K, dependent data for DATA argument of Optimus(), m(), u() and r(), we are now ready to call Optimus. Let us first investigate the Acceptance Ratio Annealing (SA) version of Optimus on 4 CPUs (the vast majority of personal computers currently have at least 4 CPUs), which can be executed as follows:

```
Optimus(NCPU=4, OPTNAME="poly_4_SA", LONG=FALSE,
        OPT.TYPE="SA",
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=DATA)
```

Note that the field LONG=FALSE is included in the function call so that all data from the optimsation process is saved. Calling Optimus with LONG=TRUE will result in a memory saving optimisation process (more details in the Advanced Usage section in this document). Of the 4 optimisation replicas, the second and fourth CPUs found the best parameter configuration (lowest RMSD) in our trial:

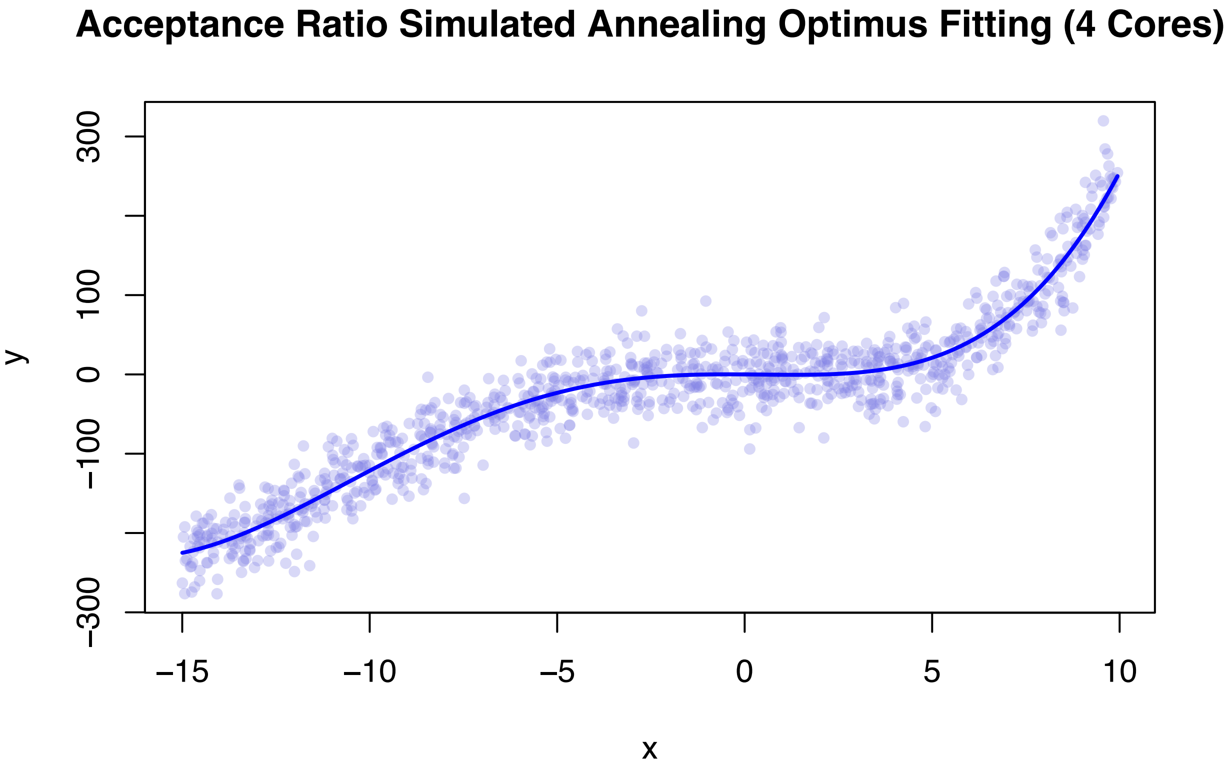


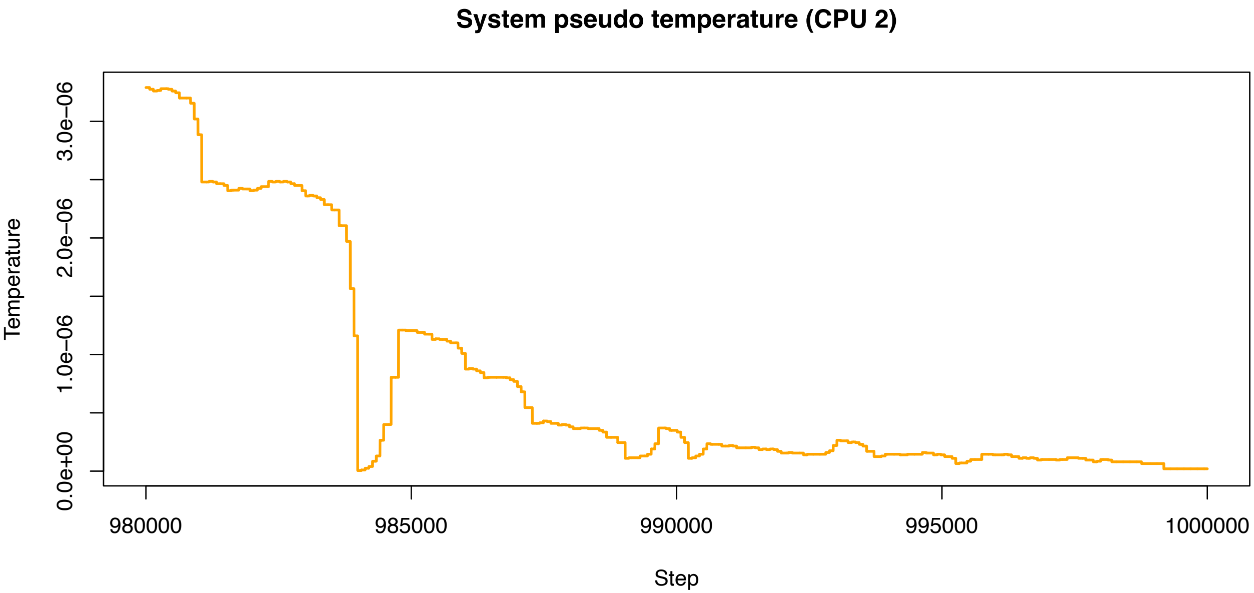
Table 1: 4-core Acceptance Ratio Simulated Annealing run results from Optimus.

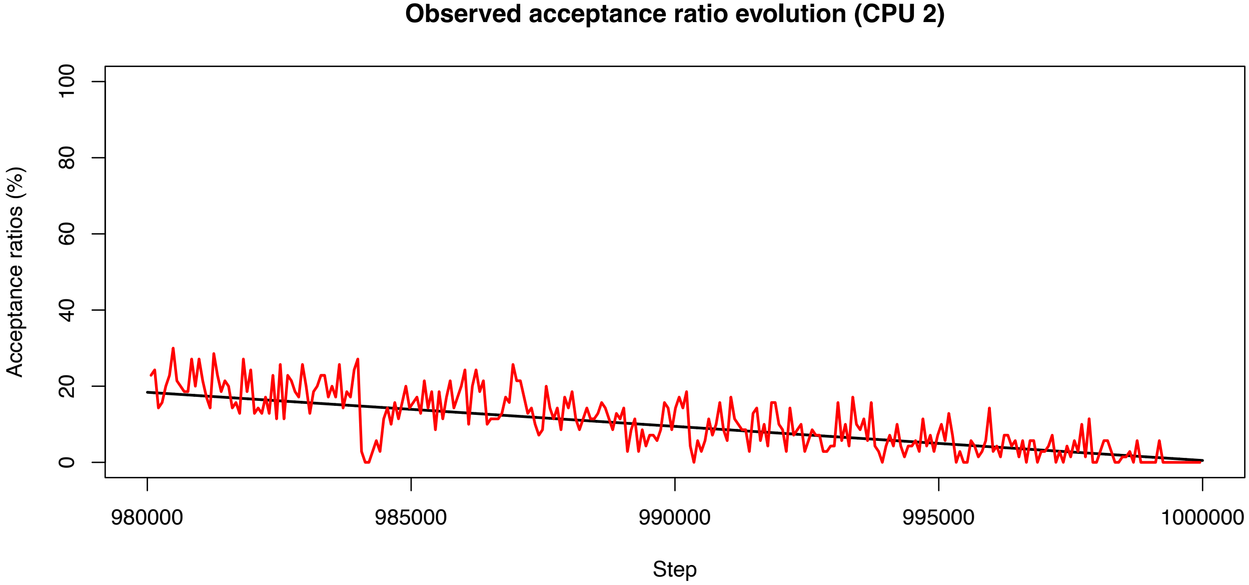
	E (RMSD)	K1	K2	K3	K4
CPU 1	28.857	-0.1560	-0.2850	0.1905	0.0095
CPU 2	28.841	-0.3760	-0.2825	0.1920	0.0095
CPU 3	28.864	-0.1045	-0.2820	0.1905	0.0095
CPU 4	28.841	-0.3760	-0.2825	0.1920	0.0095

The equation recovered by CPU 2 (and 4) is $y = -0.3760x - 0.2825x^2 + 0.1920x^3 + 0.0095x^4$.

Notice that although the RMSD of this solution, 28.841, is greater than the RMSD of the least squares solution, 28.827, it is less than the RMSD of the de-noised data found above, 28.859.

The two graphs below illustrate i) the evolution of the system pseudo temperature, in response to alterations made by the Temperature Control Unit (TCU), as a function of the optimsation step; and ii) the observed acceptance ratio as a function of the optimisation step, respectively. The graphs show data from the last 20 000 steps of the optimisation executed by CPU 2.





In the first plot, the solid red line tracks the observed acceptance ratios calculated by Optimus at the end of each STATWINDOW and the dashed black line tracks the target acceptance ratio based on the annealing schedule. From the above two graphs, notice that while the observed acceptance ratio tracks the target acceptance ratio closely, the system pseudo temperature changes significantly and non-monotonically. This illustrates that the adaptive thermoregulation allows Optimus to effectively anneal the system acceptance ratio.

Acceptance Ratio Replica Exchange Optimus Run

Let us now consider the Replica Exchange version of Optimus on 12 CPUs. The purpose here is to illustrate how to run an optimisation using the Replica Exchange version of Optimus; this method is of course an overkill for solving this simple task.

In addition to the arguments specified above, the Replica Exchange version of Optimus also requires an input variable ACCRATIO, which is a vector that defines the acceptance ratios to be used for each of the replicas initiated, 12 in this case. Note that the length of ACCRATIO must always be equal to the argument NCPU.

```
ACCRATIO <- c(90, 82, 74, 66, 58, 50, 42, 34, 26, 18, 10, 2)
```

Having defined the acceptance ratios for each level, the optimisation can be executed as follows:

```
Optimus(NCPU=12, OPTNAME="poly_12_RE", LONG=FALSE,
        OPT.TYPE="RE", ACCRATIO=ACCRATIO,
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=DATA)
```

Of the 12 optimisation replicas, replica 8 finds the best parameter configuration (lowest RMSD) in this trial:

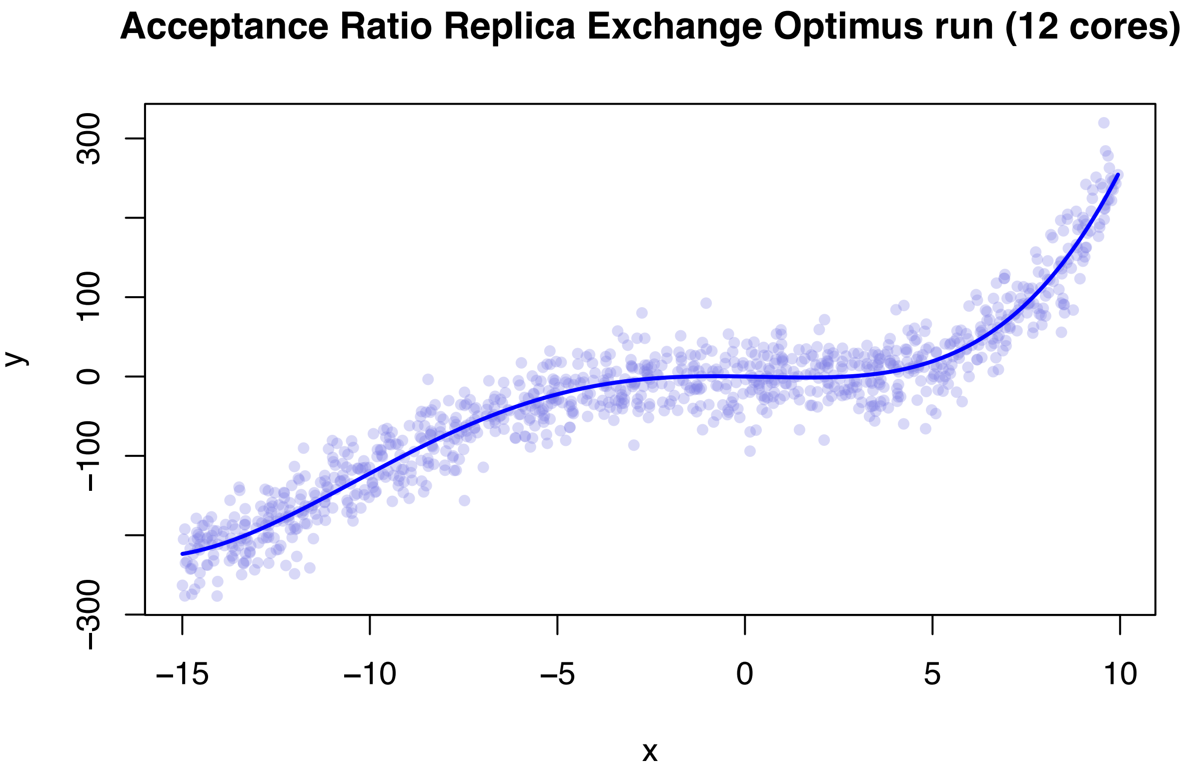


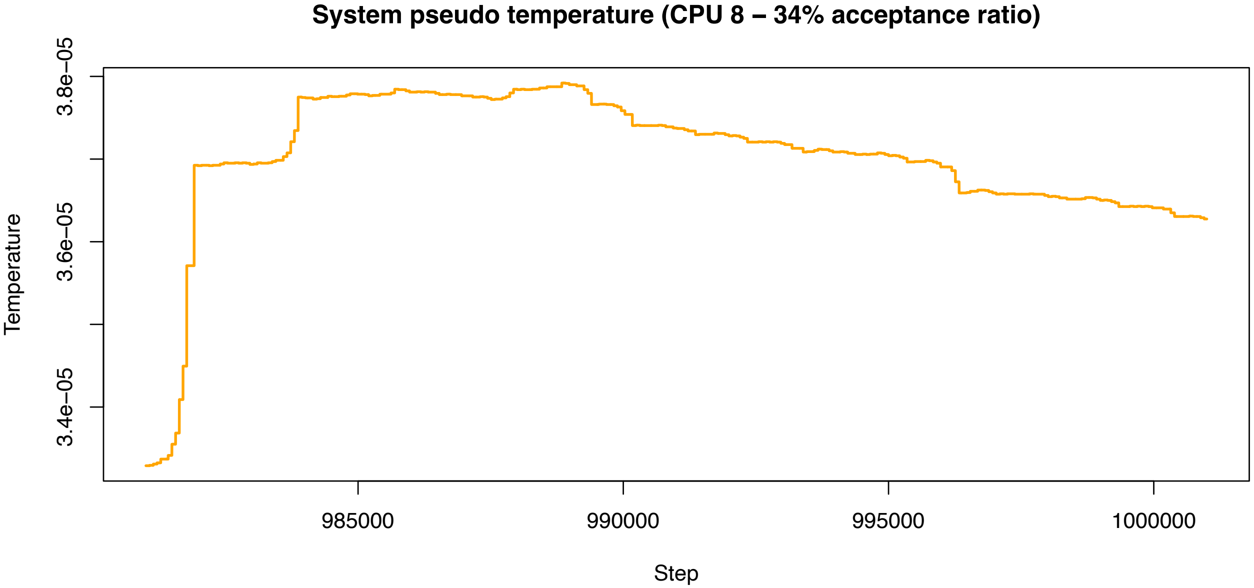
Table 2: 12-core Replica Exchange run results from Optimus.

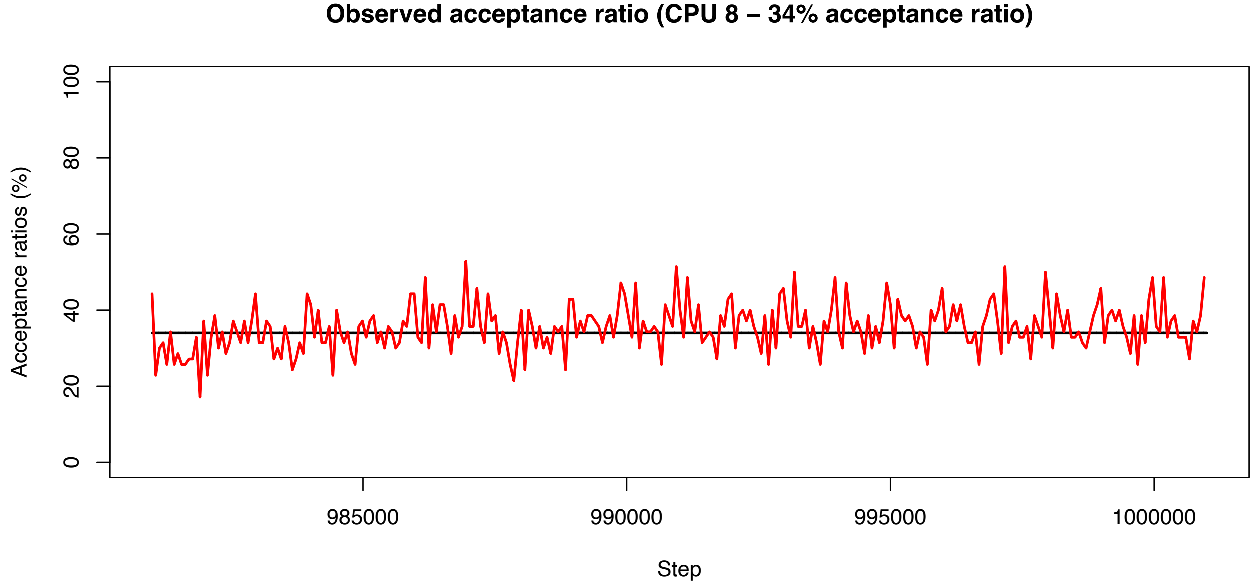
	Replica Acceptance Ratio	E (RMSD)	K1	K2	K3	K4
CPU 1	90	29.71934	-3.2760	-0.5760	0.2395	0.0135
CPU 2	82	29.51819	-3.4445	-0.3755	0.2300	0.0115
CPU 3	74	28.88707	-0.0340	-0.2375	0.1870	0.0090
CPU 4	66	30.10499	-3.8095	-0.6630	0.2435	0.0140
CPU 5	58	29.06293	-2.3575	-0.4180	0.2200	0.0115
CPU 6	50	29.70906	-3.7360	-0.3785	0.2320	0.0115
CPU 7	42	28.85095	-1.1370	-0.3515	0.2045	0.0105
CPU 8	34	28.82721	-0.8175	-0.3130	0.1990	0.0100
CPU 9	26	28.84057	-0.5760	-0.2740	0.1940	0.0095
CPU 10	18	29.48785	-2.7805	-0.5420	0.2325	0.0130
CPU 11	10	28.85095	-1.1370	-0.3515	0.2045	0.0105
CPU 12	2	29.41377	1.2255	-0.0895	0.1645	0.0070

The equation recovered by CPU 8 is $y = -0.8175x - 0.313x^2 + 0.199x^3 + 0.01x^4$.

Notice that the RMSD of this solution, 28.8272, is less than the RMSD of the Acceptance Ratio Simulated Annealing solution, 28.841, and only slightly greater than the RMSD of the least squares solution, 28.8266.

Let us now briefly examine the evolution of the system pseudo temperature and acceptance ratio compliance in response to the adaptive thermoregulation. The following two graphs represent data from the last 20 000 steps of optimisation replica running on CPU 8 (fixed 34% target acceptance ratio).





Notice that in the observed acceptance ratio graph, the dashed line indicating the target acceptance ratio is constant (as opposed to linearly changing as in acceptance ratio annealing). This is because each processor in the replica exchange mode has a single target acceptance ratio, as described above. Here again, adaptive decisions on the pseudo temperature to maintain the desired acceptance ratio result in non-monotonic, and non-uniform pseudo temperature adjustments, while the observed acceptance ratios fluctuate relatively closely around the set target value.

Summary

We now understand the input requirements to interface with the Acceptance Ratio Simulated Annealing and Replica Exchange versions of Optimus. In this example, both versions retrieved solutions having a lower RMSD than the de-noised data, and only a slightly greater RMSD than the optimal least squares solution.

Replica Exchange resulted in a better solution than Simulated Annealing, at the cost of greater computing resources.

Table 3: Summary of solutions.

	E (RMSD)	K1	K2	K3	K4
De-noised Function	28.85858	-1.0000	-0.3000	0.2000	0.0100
Optimus (AR Simulated Annealing)	28.84100	-0.3760	-0.2825	0.1920	0.0095
Optimus (AR Replica Exchange)	28.82721	-0.8175	-0.3130	0.1990	0.0100
Least Squares	28.82655	-0.7406	-0.3074	0.1978	0.0099