ROptimus User Manual

08 December 2022

Contents

Introduction

Motivation

For a complex model, where the unknown parameters cannot be determined by conventional linear or non-linear fitting techniques, methods for optimisation based on biased random sampling of the parameter space, are the methods of choice (Cowles and Carlin 1996). In such cases, the quality of the solution found, following some optimisation protocol, depends on that protocol's ability to effectively explore the parameter space (Gilks, Richardson, and Spiegelhalter 1996) and be drawn to more favourable areas therein. Many existing methods perform well for certain modelling scenarios, but fail in others due, in part, to an inefficient exploration of the parameter space and easy trapping into local minima with regard to the metrics for the quality of the found solutions. In this manual, we present ROptimus, a universal Monte Carlo optimisation engine in R with acceptance ratio annealing, replica exchange and adaptive thermoregulation. It can universally interface with any modelling initiative through its interfacing functions, and optimise the model parameters by effectively exploring the parameter space. Controlled by a user, ROptimus can execute either an annealing or a replica exchange procedure, however, both driven by acceptance ratio adjustment, rather than by altering pseudo temperature.

This manual will begin with a basic overview of Monte Carlo optimisation and the common temperature-based simulated annealing framework. It will then proceed with a presentation of the acceptance ratio annealing procedure of ROptimus, its adaptive thermoregulation feature, and its replica exchange procedure. After an explanation of how to download the ROptimus R package from GitHub and install it locally, five stand-alone tutorials will be presented to illustrate how users should employ ROptimus and to demonstrate its flexibility as an optimisation engine across a wide variety of optimisation scenarios. Finally, an "Advanced User Manual" section is included, in which all possible and tested input parameters to ROptimus are outlined and the output formats are detailed.

Briefly on Monte Carlo and Temperature-Driven Simulated Annealing Procedure

Let us assume that our model is a certain function $m()$ that performs operations on the inputted $\mathbf{K}$ coefficient set and returns the observable object $O = m(\mathbf{K})$. Our task is to optimise $\mathbf{K}$, the set of coefficients, so that the error metric $u(O)$ that measures the violations from the target O^{trg} set by the model-generated O is minimal. In a Monte Carlo optimisation procedure, we can define a pseudo-energy $e(u)$ of the system as a function of $u()$, where lower values of the pseudo energy e correspond to better candidate solutions $\mathbf{K}$. In order to find a better set of $\mathbf{K}$, we need to alter it by a certain rule $r()$ that, if repeated many times, enables the sampling of the parameter space for $\mathbf{K}$. One can then evaluate the pseudo energies before and after the alterations:

$$E_1 = e \circ u \circ m(\mathbf{K})$$
$$E_2 = e \circ u \circ m \circ r(\mathbf{K})$$

We then accept or reject the alteration move, meaning we accept the new set of $r(\mathbf{K})$ coefficients as the new $\mathbf{K}$ or revert back to the previous $\mathbf{K}$, guided by the following acceptance probability, as postulated in a Metropolis criterion (Chib and Greenberg 1995), (Chen and Roux 2015):

$$p_{accept} = min(1, e^{-\frac{\Delta E}{T}})$$
$$\Delta E = E_2 - E_1$$

where T is the pseudo temperature, that should be always greater than 0. For a given $\Delta E > 0$ energy difference, one would have a certain stringency for accepting the alteration move, depending on the value of the pseudo temperature T. Therefore, if the Monte Carlo simulation was to modify the $\mathbf{K}$ set to a state where any further move would increase the pseudo energy at a great enough amount for the moves to be almost always rejected, then one could overcome that state and further sample the other values in the parameter space by increasing the value of the pseudo temperature.

To this end, one way to overcome the barriers is by using the technique known as simulated annealing, where we gradually anneal the temperature from a higher value to lower during the course of the simulation (Kirkpatrick 1984). The parts of the simulation where the pseudo temperature is higher, allows relatively unconstrained exploration of the parameter search space, whereas those parts with a lower pseudo temperature limit the search to a more local area of the energy landscape and associated parameter space. Multiple cycles of this annealing procedure can thus be executed to increase the overall sampling.

Acceptance Ratio Simulated Annealing Procedure

A significant limitation of the pseudo-temperature-based simulated annealing, while used only in an optimisation objectives, is that a given scheme of temperature annealing might be efficient for some models or pseudo-energy metrics, but not efficient for others (Ingber 1993). A temperature for a given system at a given point of the annealing cycle that is designated to be quite permissive in terms of accepting the moves, can actually not be permissive for some other models or systems simulated. This depends on the value and scale of the ΔE energy difference, as can be seen in the equation for p_{accept}, and can be affected by any of the $e()$, $u()$, $m()$ components, and the system configuration $\mathbf{K}$. This necessitates a pre-adjustment of the pseudo-temperature values from one modelling objective to another. Furthermore, even within a single model optimisation procedure on a defined system, the pseudo-energy metric can shift into a scale (due to a significant alteration/shift in the system as per the alteration rule $r()$) that does not match with the selected temperature scheme anymore, leading to a poor sampling of the parameter space when optimisation is at stake within restricted time and computational capacity. This can often be the case when the pseudo energy of the system does not exhibit a smooth dependency on $\mathbf{K}$, loosely meaning that close states of $\mathbf{K}$ do not necessarily produce close pseudo-energy values (for instance, due to an alteration rule that randomly expands or trims the system, such as when equation terms are randomly turned on or off).

To this end, in general cases where

A) we do not deal with energies and temperatures that display the smoothness or roughness emulating real physical systems, such as while simulating molecular systems;
B) we are only interested in final optimised configuration of $\mathbf{K}$, and do not need to characterise the statistical populance as an ensemble of reachable states/solutions,

we can anneal a metric for crossing different barriers that is more transferable to different systems and system configurations. As such a metric, ROptimus uses the acceptance ratio, expressed as a fraction of accepted moves within certain number of past steps (`STATWINDOW`).

In a given annealing cycle, ROptimus constructs a linear target acceptance ratio regiment for each step. This is done based on an initial target acceptance ratio, a final acceptance ratio, and the number of iterations in each cycle for a given optimisation run (all of which can be specified as inputs). Once the optimisation process begins, ROptimus calculates an observed acceptance ratio at the end of each `STATWINDOW` (a fixed number of steps which can be specified by the user) by calculating the fraction of the accepted moves from all the past trials in the current `STATWINDOW`. Thereafter, ROptimus compares the observed acceptance ratio with the pre-defined target acceptance ratio based on the annealing schedule and determines whether and how to alter the system pseudo-temperature (adaptive thermoregulation) to align the observed acceptance ratio with the target ratio at the end of the following `STATWINDOW`. Thus, by employing acceptance ratio annealing and adaptive thermoregulation, ROptimus is able to methodically explore the parameter space for $\mathbf{K}$ even when no smooth relationship exists between $\mathbf{K}$ and the system pseudo energy.

Adaptive Thermoregulation

All decisions governing the adaptive pseudo-temperature alterations on the system are made by a Temperature Control Unit (TCU) in ROptimus. Note, that the TCU is completely encapsulated as a backend unit, such that modifications can be easily made by advanced users if needed. This section articulates the exact protocol followed by the default state of the TCU.

The initial system temperature is specified as an input argument. At the end of each `STATWINDOW`, if the observed acceptance ratio is within a fixed range `T.DELTA` (specified as an input argument) of the target acceptance ratio based on the annealing schedule, the TCU will make no change to the current system pseudo temperature. If the observed acceptance ratio is less than the ideal ratio and outside the range of `T.DELTA`, the TCU will increase the system pseudo temperature by a value `T.ADJSTEP` (the initial value of `T.ADJSTEP` is specified as an input argument). Similarly, if the observed acceptance ratio is greater than the ideal ratio and outside the range of `T.DELTA`, the TCU will reduce the temperature by a value `T.ADJSTEP`.

If the observed acceptance ratio keeps being below the ideal acceptance ratio for `TSCLnum` (an integer input argument) number of subsequent `STATWINDOW`s, `T.ADJSTEP` will be increased by a factor `T.SCALING` (an input argument). Similarly, `T.ADJSTEP` will also be decreased by a factor `T.SCALING` if the observed acceptance ratio is greater than the ideal acceptance ratio for `TSCLnum` subsequent `STATWINDOW`s. `T.ADJSTEP` is reset to its original input value whenever a series of subsequent observed acceptance ratios being greater than/less than ideal acceptance ratios is broken. If ever the TCU subtracts `T.ADJSTEP` from the current temperature and the result is a negative value, the system pseudo temperature is set to `T.MIN` (an input argument). The final feature of the TCU is that although the initial system pseudo temperature is specified by the user, if multiple annealing cycles are employed, the initial pseudo temperature for acceptance ratio annealing cycles, after the first cycle, is inferred from the decisions of the TCU on previous cycles.

The above collection of TCU decision rules with their default values result in pseudo-temperature modulations that assure the observed acceptance ratios during ROptimus runs to follow the ideal acceptance ratios remarkably well. Moreover, as will be highlighted in the five tutorials below, large temperature alterations are often required to align the observed acceptance ratios with the ideal ratios, a task which ROptimus excels at, whereas other protocols would have difficulties and result in poor parameter sampling because of fixed or gradually changing temperature regiments.

Acceptance Ratio Replica Exchange Procedure

ROptimus additionally supports acceptance ratio replica exchange as an optimisation mode, which can be selected in place of acceptance ratio annealing, provided that the user has access to multiple processors (ideally at least 4, and preferably 8 or more). The inspiration for this additional mode was taken from the parallel tempering Monte Carlo techniques and Replica Exchange Molecular Dynamics (REMD) simulations (Sugita and Okamoto 1999). In particular, REMD simulation is a technique employed to obtain equilibrium sampling of a molecular system (for instance, a protein) at a certain fixed (usually the lowest in a range) temperature. Let $T = \{T_1, T_2, ..., T_n\}$ be a set of n distinct temperatures for which $T_1 < T_2 < ... < T_n$. In REMD, n replicas of molecular dynamics simulations for a given system are initialised at each $T_i \in T$. Note, that each temperature T_i corresponds to a slightly different potential of the examined system to roam the energy landscape and cross the barriers. States possible to populate at a temperature T_i can become a bit more accessible at a temperature T_{i+1} and so on. If the state configurations in adjacent replicas are allowed to exchange, the simulation will be able to overcome energy barriers at various temperature replicas and thoroughly explore the parameter space. Moreover, for configuration x_n in replica T_i, and configuration x_m in replica T_{i+1}, equilibrium sampling is achieved by selecting appropriate exchange probabilities (Sugita and Okamoto 1999), generally expressed as:

$$p_{re} = min(1, P(T_{i+1} - T_i; E(x_n) - E(x_m))$$

Thus, overall, replica exchange simulations can be executed by simulating n replicas of a simulation at distinct temperatures simultaneously and independently, next randomly selecting two configurations in adjacent replicas and exchanging them with probability p_{re}.

Comparison with Related R Packages

ROptimus strengths are at being a general-purpose optimisation engine that can be applied to a diverse set of problems through the flexibility and modularity of its interfacing functions and the capability to extensively search the parameter space by using a transferable process that uses acceptance ratio-based adaptive thermoregulation, in lieu of the situational tuning of the conventional, more system- and scale-dependent pseudo temperature, and by offering an all-purpose replica exchange option borrowed from parallel tempering often implemented in molecular dynamics simulations and other Monte Carlo methods.

CRAN Task View: Optimization and Mathematical Programming (Version: 2022-04-12) (Schwendinger and Borchers 2022) lists currently available (non-archived), general-purpose and tailored R packages for dealing with optimisation problems. Categories include general-purpose continuous, quadratic programming, least-squares, semidefinite and convex, global and stochastic, mathematical programming, multi-objective and combinatorial solvers, and those for highly specific applications. It also lists optimisation infrastructure packages, interfaces to open source and commercial solvers, and libraries for benchmarking optimisation algorithms. Among the packages indicated, the ones with simulated annealing (SA) capabilities are *NMOF* (Gilli, Maringer, and Schumann 2019), (Schumann 2022), *dclone* (Solymos 2010), *GenSA* (Xiang et al. 2013), *qap* (Hahsler 2017) and *stats* (R Core Team 2021). Not present in the list but also implement SA are *optimization* (Husmann, Lange, and Spiegel 2017), *likelihood* (Murphy 2022), *pomp* (King, Nguyen, and Ionides 2016), (King et al. 2022) and *subselect* (Orestes Cerdeira et al. 2020) packages. Unlike ROptimus, however, the above packages employ the conventional pseudo temperature-based annealing procedure, which subjects them to problems detailed in previous sections e.g. the current temperature scale being suddenly irrelevant due to huge alterations in the system. Since their temperature control (e.g. monotonic, geometric and logarithmic cooling) is not adaptive, users have to pay extra attention to input arguments controlling the thermoregulation still with no assurance of effective parameter space sampling especially for complex problems with many local optimum traps. Finally, the aforementioned packages, except *NMOF*, expect limited or a specific format of a problem (e.g. fitting coefficients, maximum likelihood estimation) and therefore, expect specific data types for inputs (e.g. numeric vectors or lists). Consequently, the alteration rule for generating new candidate solutions, sometimes hard-coded within the function, is constrained making it difficult to repurpose these packages. In contrast, a key strength of ROptimus is its modularity and flexibility, allowing users to design any model, objective/cost and alteration functions that can be made to handle any R objects.

How to Cite

The usage of ROptimus should be accompanied with the following citations:

- Nicholas A.G. Johnson, Liezel Tamon, Xin Liu and Aleksandr B. Sahakyan, *ROptimus: a parallel general-purpose adaptive optimisation engine*, *bioRxiv*, http://doi.org/10.1101/2022.01.18.476810, **2022**.
- Nicholas A.G. Johnson, Liezel Tamon, Xin Liu and Aleksandr B. Sahakyan, http://github.com/SahakyanLab/ROptimus, **2018**.

Installation Instructions

Installing ROptimus locally for immediate use requires only R and a connection to the Internet. It is available both on CRAN (stable release) and on GitHub (latest release). To install the latter version, open an R client and execute the commands below. Note, that the latest version of Rtools is required for this installation to work. If it is not available locally and the installation is attempted from within RStudio, a prompt will appear to download and install Rtools. After following those instructions, restart the RStudio session before reattempting the ROptimus installation.

```r
install.packages("devtools")           # install devtools
library(devtools)                      # load devtools
install_github("SahakyanLab/ROptimus") # install ROptimus
library(ROptimus)                      # load ROptimus
```

To install from CRAN, follow the usual procedure of installing an R package.

Execution Instructions

ROptimus optimisation engine works seamlessly and requires little to no intervention from the user while applied to a wide variety of optimisation tasks. The part where the user must intervene is the specification of $\mathbf{K}$, and the R functions corresponding to $r()$, $m()$, and the combined $e \circ u()$. Those objects serve as the application interface of the ROptimus engine to any optimisation task. The port through which additional user data or objects can be supplied to ROptimus, in case those are needed in the specification of a model, is implemented through the `DATA` argument that is passed to the model function `m()` and `u()`. All the interfacing functions and objects have a minimal and well defined internal compliance rules in ROptimus, and otherwise can be flexible and as simple or complicated as the user requires, to properly plug ROptimus into a desired modelling objective. The whole optimisation procedure with acceptance ratio annealing or replica exchange is then fully taken care of by the ROptimus engine.

The five tutorials below showcase the usage of ROptimus in five, broadly different scenarios, with a particular focus on how to write the interfacing functions to plug the ROptimus engine, and how the acceptance ratio annealing mode performs in comparison to the acceptance ratio replica exchange one. The examples also cover a usage scenario where an external, more complicated, program is linked to the R procedure. Furthermore, we provide a built in function `OptimusExamples()`, which can generate the necessary inputs for any of the mentioned five tutorials, ready for modifications to tailor those to particular needs. The `OptimusExamples()` function requires, at minimum, the tutorial identifier (1 to 5) along with the methodology specification (simulated annealing - SA; replica exchange - RE), in order to generate the interfacing ROptimus input, which by default is saved as an `example.R` file within the current directory (all alterable). The function can also save the vignette corresponding to the requested example, for further information about the example, if needed for further alterations.

The recommended way to use ROptimus would thus be by exploring and understanding the below tutorials, finding the example closest to one's objective, generating the input for that example using `OptimusExamples()`:

```r
OptimusExamples(example=2, method="SA")
```

then modifying the input objects to your specific needs and workflow.

Tutorial 1: Finding Coefficients for a Polynomial Function

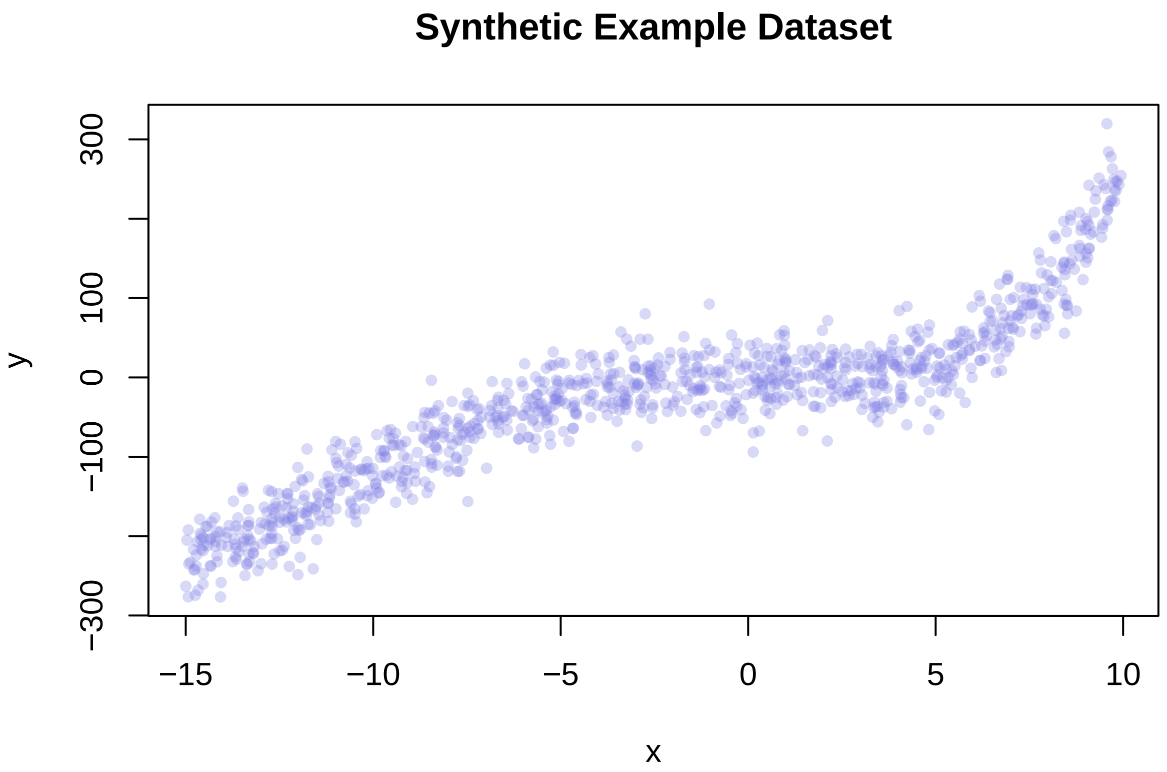
Problem Statement

In this example, we shall use ROptimus 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 ROptimus input, rather than the complexity of the task.

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

```r
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 ROptimus performs at finding the correct coefficients. The synthetic "real world" noisy data that we generated looks like this:

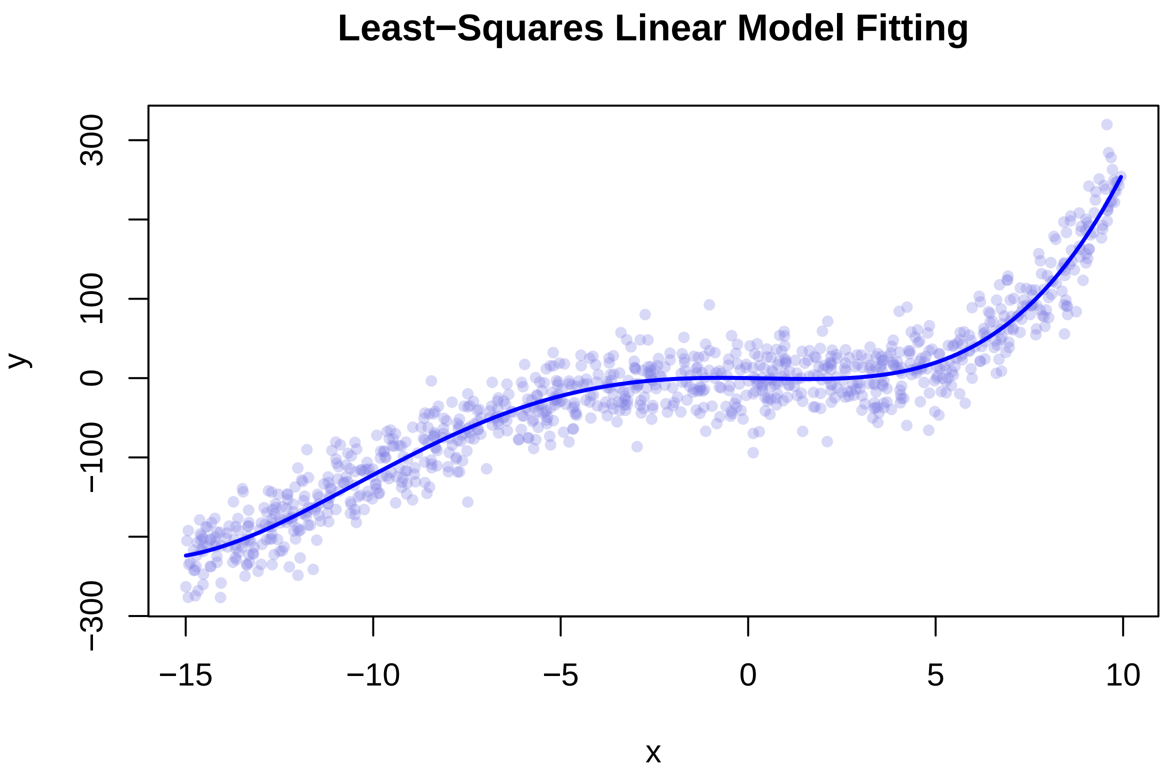


Before we turn to ROptimus, let us see how the proper linear model fitting will perform using this data.

```r
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:

```r
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:

```r
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 ROptimus Inputs

Now we can set up the inputs for the ROptimus 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 ROptimus run, are given as inputs in the `Optimus()` 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 ROptimus 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()`, 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 ROptimus, 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 ROptimus 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 ROptimus 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:

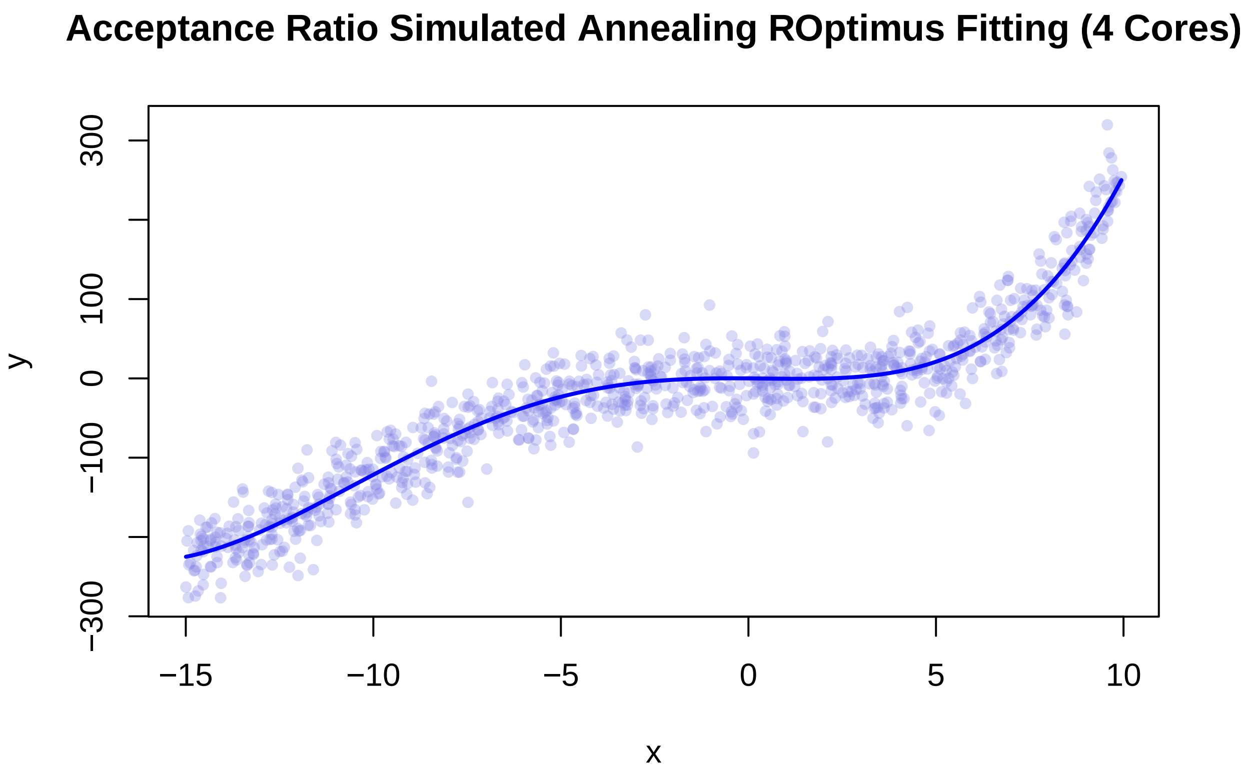


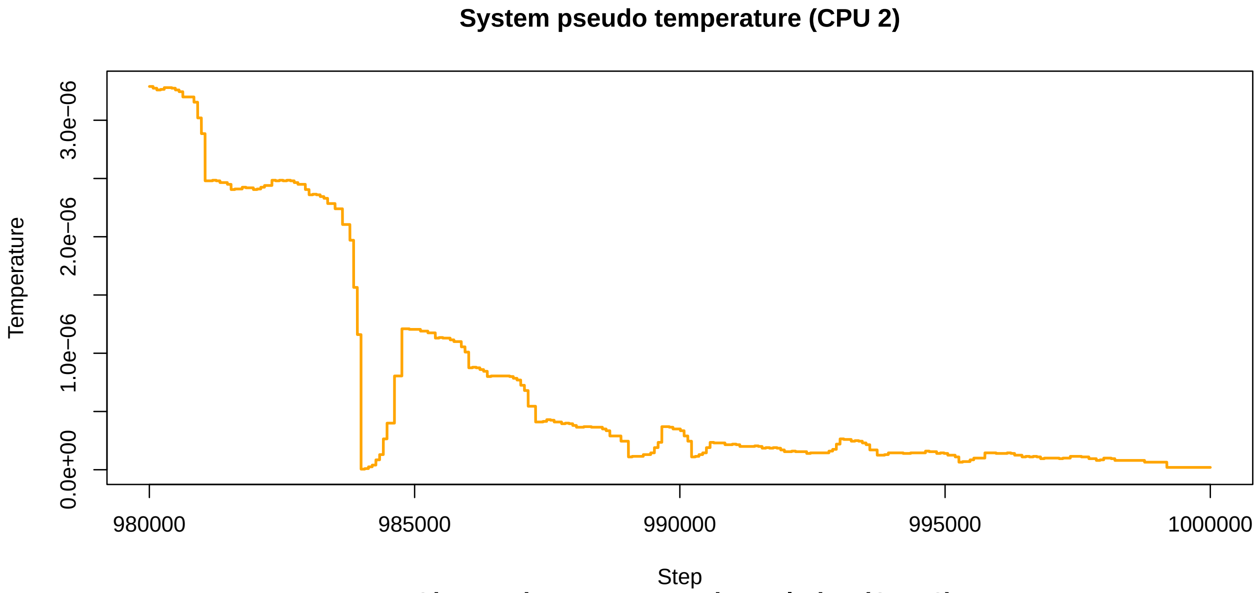
Table 1: 4-core Acceptance Ratio Simulated Annealing run results from ROptimus.

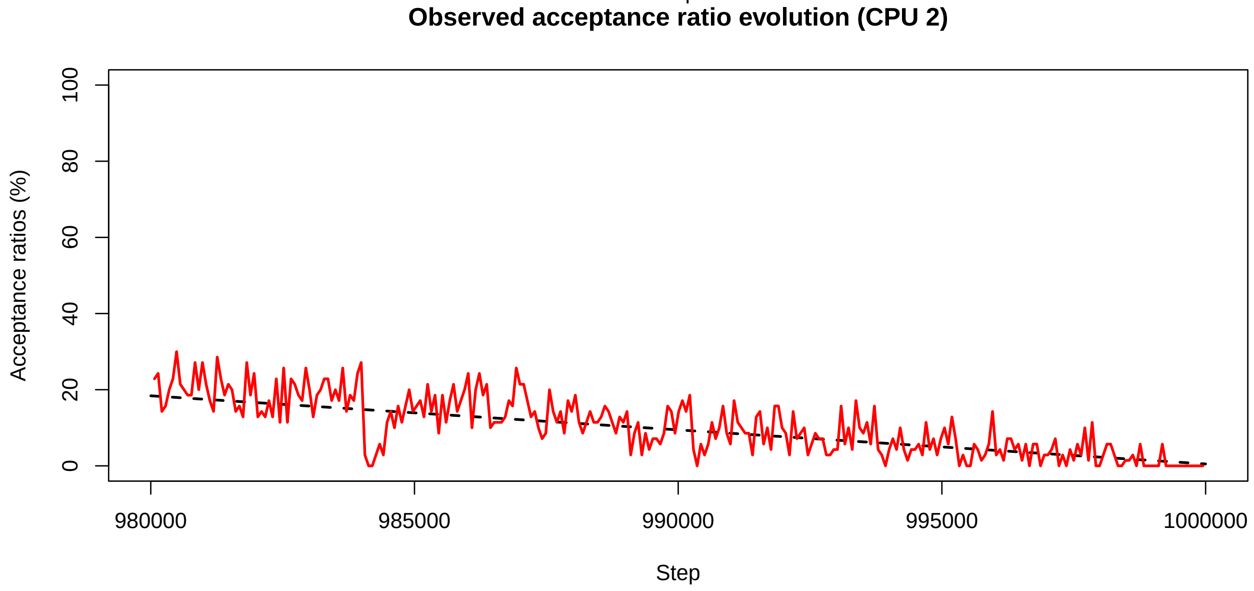
	E (RMSD)	K1	K2	K3	K4
CPU 1	28.857	-0.1560	-0.2850	0.1905	0.0095
CPU 2	28.841	-0.3760	-0.2825	0.1920	0.0095
CPU 3	28.864	-0.1045	-0.2820	0.1905	0.0095
CPU 4	28.841	-0.3760	-0.2825	0.1920	0.0095

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 ROptimus 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 ROptimus to effectively anneal the system acceptance ratio.

Acceptance Ratio Replica Exchange ROptimus Run

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

In addition to the arguments specified above, the Replica Exchange version of ROptimus 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:

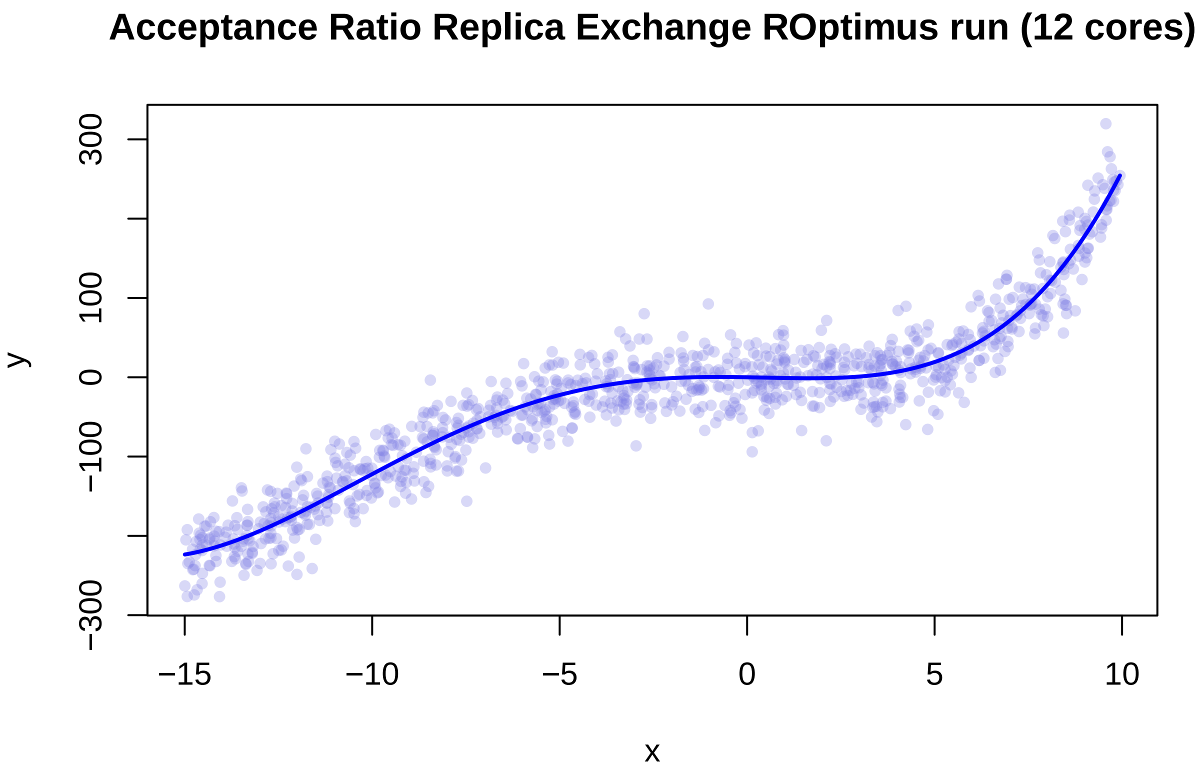


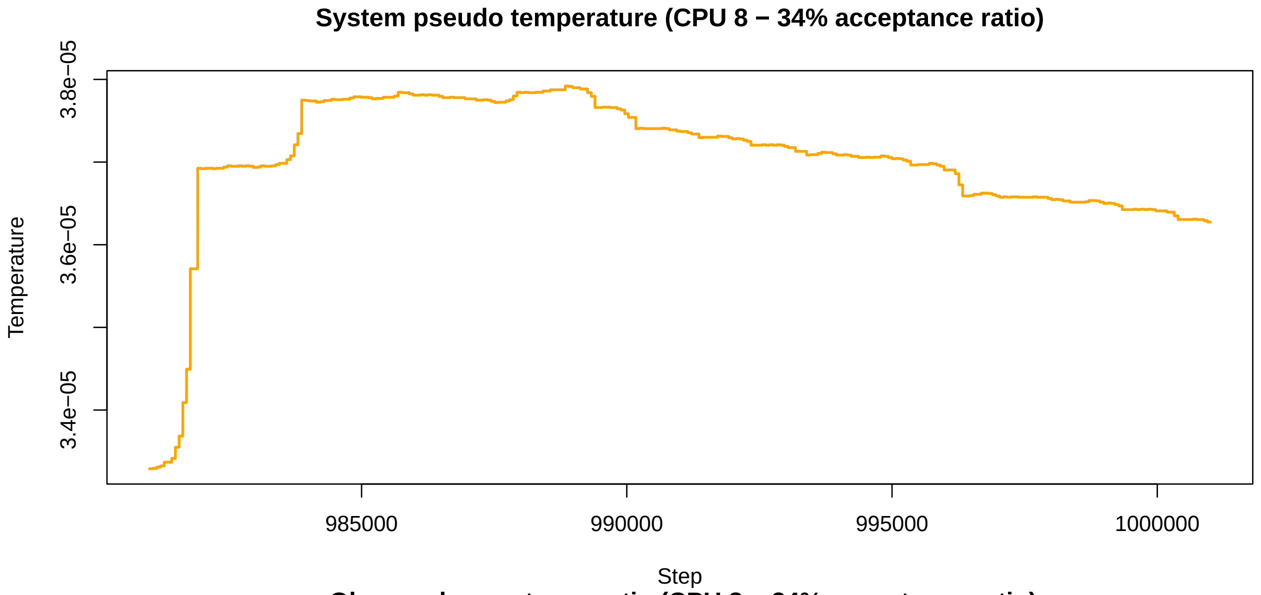
Table 2: 12-core Replica Exchange run results from ROptimus.

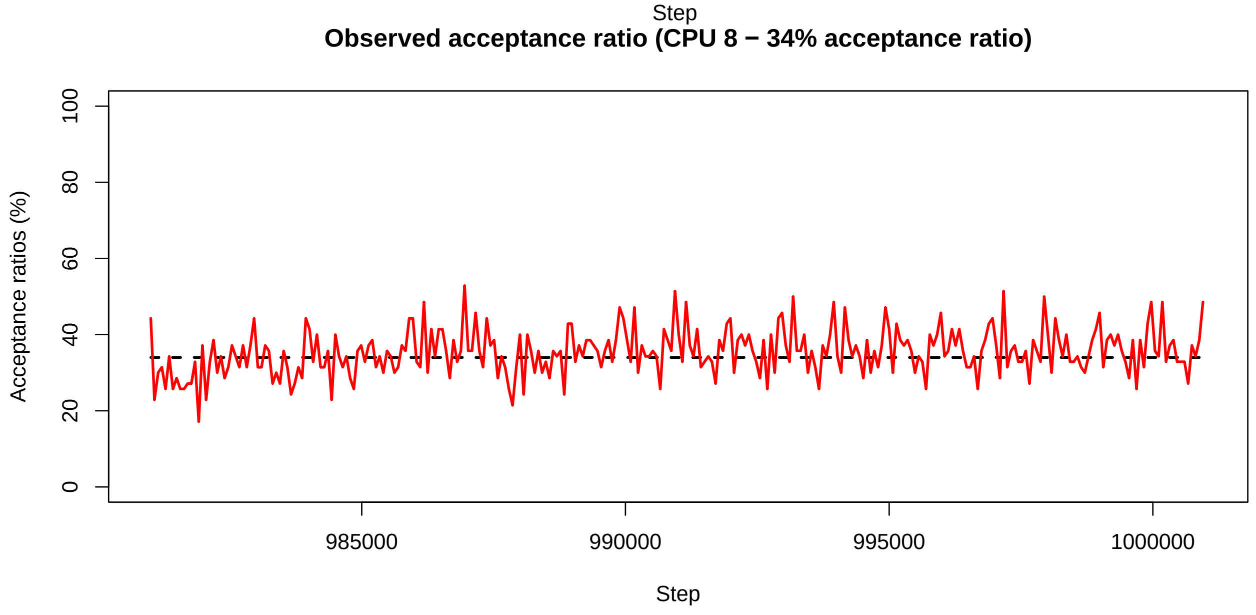
	Replica Acceptance Ratio	E (RMSD)	K1	K2	K3	K4
CPU 1	90	29.71934	-3.2760	-0.5760	0.2395	0.0135
CPU 2	82	29.51819	-3.4445	-0.3755	0.2300	0.0115
CPU 3	74	28.88707	-0.0340	-0.2375	0.1870	0.0090
CPU 4	66	30.10499	-3.8095	-0.6630	0.2435	0.0140
CPU 5	58	29.06293	-2.3575	-0.4180	0.2200	0.0115
CPU 6	50	29.70906	-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 ROptimus. 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
ROptimus (AR Simulated Annealing)	28.84100	-0.3760	-0.2825	0.1920	0.0095
ROptimus (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

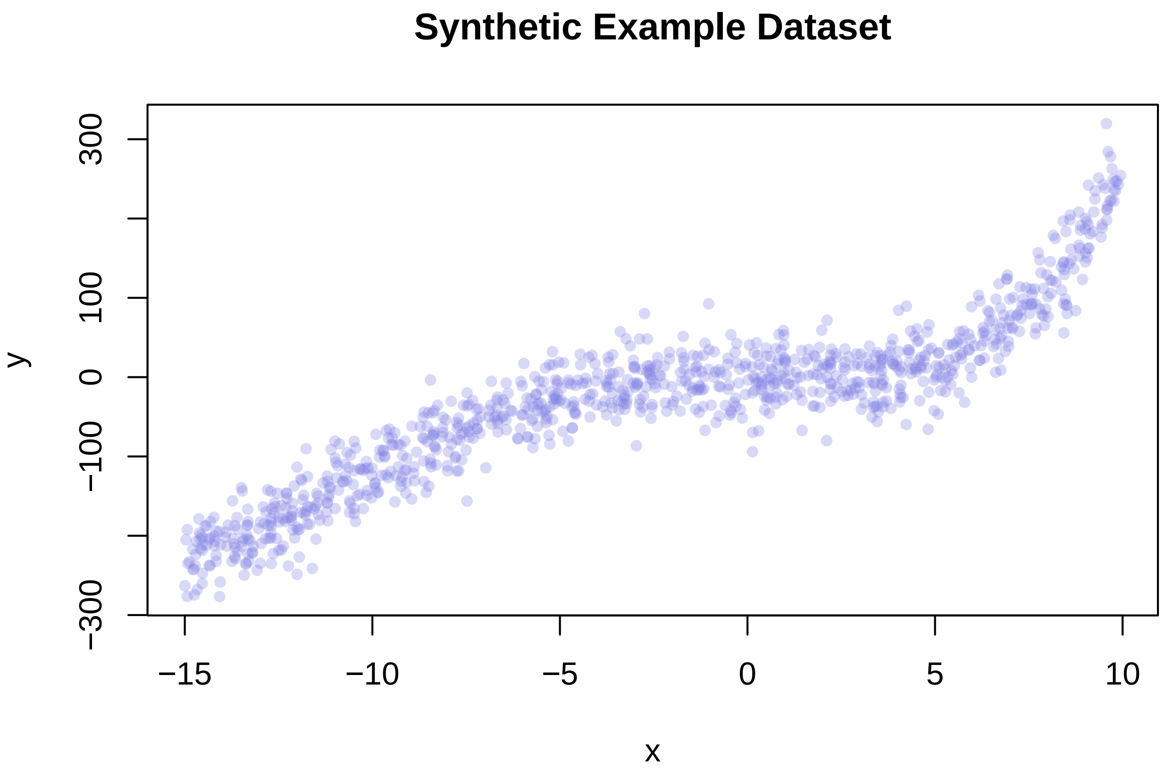
Tutorial 2: Finding an Optimal Equation by Navigating Through a Term Space

Problem Statement

Consider again the synthetic data that was created in **Tutorial 1**. Suppose that we were only provided with the data and, unlike in **Tutorial 1**, had no knowledge of the best terms to be included in a functional representation of said data. In this example, we shall use ROptimus to determine which terms should be used in a least squares fitting of the data to achieve a representation with low RMSD while avoiding overfitting the data with too complicated equation.

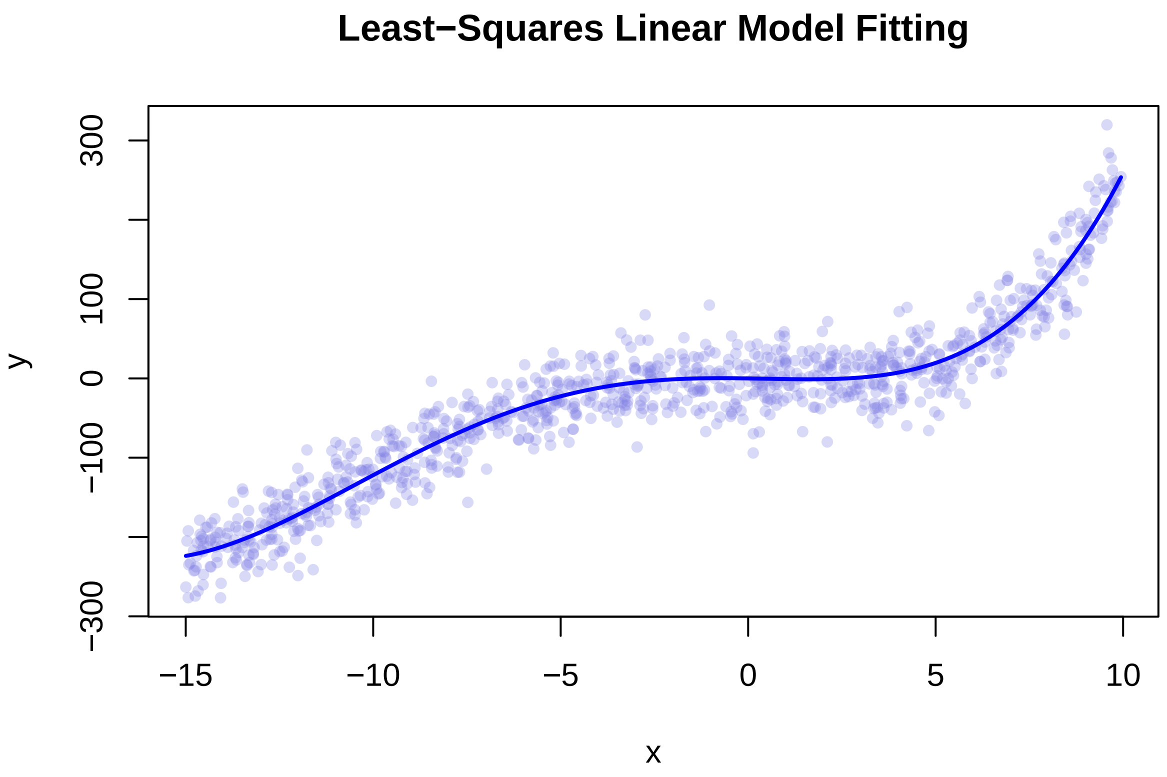
Let us start by generating the same data which was used in **Tutorial 1**:

```
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)
```



From **Tutorial 1**, we know that if presented with this data and under the assumption that the most appropriate model to describe the data is $k_1x + k_2x^2 + k_3x^3 + k_4x^4$, the Least Squares model fitting is $y = -0.741x - 0.307x^2 + 0.198x^3 + 0.010x^4$. We also know that the RMSD between the observed data y and the linear model fitting outcome is:

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



Defining ROptimus Inputs

Let us first define an ordered set *terms* that is a collection of candidate terms to include in the representation of the data:

$terms = \{x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}, e^x, |x|, sin(x), cos(x), tan(x), sin(x)cos(x), sin^2(x), cos^2(x), sin(x^2), sin(x^3), cos(x^2), cos(x^3), sin(x^3)cos(-x), cos(x^3)sin(-x), sin(x^5)cos(-x), cos(x^5)sin(-x), e^x sin(x), e^x cos(x), |x|sin(x), |x|cos(x)\}$

Let $terms_i$ denote the i^{th} term in the set *terms* (for example, $terms_{14} = cos(x)$). We shall use the model:

$$y = b + \sum_{i=1}^{\textbf{card}(terms)} k_i c_i terms_i$$

where each k_i is a binary variable (meaning a variable taking a value of either 0 or 1) indicating whether the i^{th} term is included in the representation, each c_i is a non-zero coefficient for the i^{th} term and b is a real number (the intercept). In our case, $\textbf{card}(terms) = 30$ so explicitly, our model is:

$y = b + k_1c_1x + k_2c_2x^2 + k_3c_3x^3 + k_4c_4x^4 + k_5c_5x^5 + k_6c_6x^6 + k_7c_7x^7 + k_8c_8x^8 + k_9c_9x^9 + k_{10}c_{10}x^{10} + k_{11}c_{11}e^x + k_{12}c_{12}|x| + k_{13}c_{13}six(x) + k_{14}c_{14}cos(x) + k_{15}c_{15}tan(x) + k_{16}c_{16}sin(x)cos(x) + k_{17}c_{17}sin^2(x) + k_{18}c_{18}cos^2(x) + k_{19}c_{19}sin(x^2) + k_{20}c_{20}sin(x^3) + k_{21}c_{21}cos(x^2) + k_{22}c_{22}cos(x^3) + k_{23}c_{23}sin(x^3)cos(-x) + k_{24}c_{24}cos(x^3)sin(-x) + k_{25}c_{25}sin(x^5)cos(-x) + k_{26}c_{26}cos(x^5)sin(-x) + k_{27}c_{27}e^x sin(x) + k_{28}c_{28}e^x cos(x) + k_{29}c_{29}|x|sin(x) + k_{30}c_{30}|x|cos(x)$

Formally, `K` will be a numeric vector of length $\textbf{card}(terms)$ whose i^{th} entry is k_i. `K` uniquely specifies a set $activeTerms = \{terms_i \; \forall \; i \; | k_i = 1\}$. Note that $activeTerms \subseteq terms$. Each binary variable k_i should be initialized randomly as below:

```
K <- c(term1=rbinom(n=1, size=1, prob=0.5),
       term2=rbinom(n=1, size=1, prob=0.5),
       term3=rbinom(n=1, size=1, prob=0.5),
       term4=rbinom(n=1, size=1, prob=0.5),
       term5=rbinom(n=1, size=1, prob=0.5),
       term6=rbinom(n=1, size=1, prob=0.5),
       term7=rbinom(n=1, size=1, prob=0.5),
       term8=rbinom(n=1, size=1, prob=0.5),
       term9=rbinom(n=1, size=1, prob=0.5),
       term10=rbinom(n=1, size=1, prob=0.5),
       term11=rbinom(n=1, size=1, prob=0.5),
       term12=rbinom(n=1, size=1, prob=0.5),
       term13=rbinom(n=1, size=1, prob=0.5),
       term14=rbinom(n=1, size=1, prob=0.5),
       term15=rbinom(n=1, size=1, prob=0.5),
       term16=rbinom(n=1, size=1, prob=0.5),
       term17=rbinom(n=1, size=1, prob=0.5),
       term18=rbinom(n=1, size=1, prob=0.5),
       term19=rbinom(n=1, size=1, prob=0.5),
       term20=rbinom(n=1, size=1, prob=0.5),
       term21=rbinom(n=1, size=1, prob=0.5),
       term22=rbinom(n=1, size=1, prob=0.5),
       term23=rbinom(n=1, size=1, prob=0.5),
       term24=rbinom(n=1, size=1, prob=0.5),
       term25=rbinom(n=1, size=1, prob=0.5),
       term26=rbinom(n=1, size=1, prob=0.5),
       term27=rbinom(n=1, size=1, prob=0.5),
       term28=rbinom(n=1, size=1, prob=0.5),
       term29=rbinom(n=1, size=1, prob=0.5),
       term30=rbinom(n=1, size=1, prob=0.5))
```

Next, we must define the model function `m()` that will operate on the parameter snapshot `K` and return an observable object `O`. For a given set *activeTerms* specified by `K`, `m()` will fit a linear model to the data using the entries in *activeTerms* and using the built in generalised linear model (`glm()`) function in R, thereby determining values for the variables c_i and b. Accordingly, `m()` will require access to the variables x and y, which will be provided as entries in `DATA`, a variable of type list, as in **Tutorial 1**. The object `O` will be the corresponding output of the function `glm()`. In the case that the set *activeTerms* is the empty set (meaning that all entries in `K` are 0), `m()` will fit a model using the relationship $y{\sim}x$.

```
DATA   <- NULL
DATA$x <- x
DATA$y <- y

m <- function(K, DATA){
  y <- DATA$y
  x <- DATA$x

  terms <- c("+x",
             "+I(x^2)",
             "+I(x^3)",
             "+I(x^4)",
             "+I(x^5)",
             "+I(x^6)",
             "+I(x^7)",
             "+I(x^8)",
             "+I(x^9)",
             "+I(x^10)",
             "+I(exp(x))",
             "+I(abs(x))",
             "+I(sin(x))",
             "+I(cos(x))",
             "+I(tan(x))",
             "+I(sin(x)*cos(x))",
             "+I((sin(x))^2)",
             "+I((cos(x))^2)",
             "+I(sin(x^2))",
             "+I(sin(x^3))",
             "+I(cos(x^2))",
             "+I(cos(x^3))",
             "+I(sin(x^3)*cos(-x))",
             "+I(cos(x^3)*sin(-x))",
             "+I(sin(x^5)*cos(-x))",
             "+I(cos(x^5)*sin(-x))",
             "+I(exp(x)*sin(x))",
             "+I(exp(x)*cos(x))",
             "+I(abs(x)*sin(x))",
             "+I(abs(x)*cos(x))")

  ind.terms <- which(K == 1)
  if(length(ind.terms)!=0){
    equation <- paste(c("y~",terms[ind.terms]), collapse="")
  } else {
    equation <-"y~x" # In case there are no active terms, use a simple linear model.
  }

  O <- glm(equation, data=environment())

  return(O)
}
```

Having defined `m`, we can now proceed to define the function `u`, which will determine how well a given configuration of parameters `K` is performing by operating on the observable object `O` outputted by `m()` and on the variable `DATA`. Here, to quantify (and thus be able to compare) the desirability of a given model for the data, we will employ the Akaike Information Criterion (AIC) from information theory, defined as follows:

$$AIC(M) = 2p - 2ln(L)$$

where p is the number of parameters in the fitted model, L is the maximum likelihood of the model M given the data.

The target representation will be the fitted model M (whose terms are elements of $terms$) that minimises the AIC. It is important to note that the $2p$ term in the AIC penalises overfitting by increasing AIC as function of the number of parameters, while the $-2ln(L)$ term rewards models that better represent the data by decreasing AIC as a function of the likelihood of the model.

As articulated in **Tutorial 1**, the output of `u()` should have a component `E` holding a pseudo energy for the parameter snapshot `K`, and a component `Q` that can be used for plotting the optimisation process. In this case, `E` will be equal to the value of AIC (implemented using the built in `AIC()` function in R) and `Q` will be equal to the RMSD between the predicted values of y from the fitted model and the actual y values, used for plotting purpose. Consequently, `u()` will need access to the variable y. The definition of `u()` is below:

```
u <- function(O, DATA){
  y <- DATA$y

  Q <- sqrt(mean((O$fitted.values-y)^2))
  E <- AIC(O)/1000 # Akaike's information criterion.

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

Finally, we need to define the rule function `r()`. We will adopt the following simple procedure: randomly select an equation entry from `K` and switch its value to the other binary value (on to off, or off to on).

```
r <- function(K){
  K.new <- K
  # Randomly selecting a term:
  K.ind.toalter <- sample(size=1, x=1:length(K.new))
  # If the term is on (1), switching it off (0) or vice versa:
  if(K.new[K.ind.toalter]==1){
    K.new[K.ind.toalter] <- 0
  } else {
    K.new[K.ind.toalter] <- 1
  }
  return(K.new)
}
```

Having defined all the necessary inputs, we are now ready to call `Optimus()`.

An important remark is that modelling this problem in this manner results in an objective function (AIC) that is not smooth because small changes in the parameter set `K` (as defined by `r()`) can produce significantly large changes in the objective value. The equation, i.e. the system itself, changes from one step to another. Therefore, an entirely different model is being used to fit the data at each step. Optimisation procedures in such cases are in danger to be trapped in certain minima due to a major change in pseudo temperatures necessary to overcome barriers while the system abruptly evolves. Despite this, we will see that ROptimus will get the job done by arriving at good solutions largely as a consequence of its adaptive thermoregulation and acceptance-ratio-guided optimisation procedure.

Acceptance Ratio Simulated Annealing ROptimus Run

In addition to the inputs defined above, `Optimus()` can optionally take other inputs to dictate the optimisation process (see the Advanced User Manual), all of which have built in default values and some of which will be altered in this example due to the increased computational complexity of the model defined in this tutorial compared to that of **Tutorial 1**. The variable `NUMITER` represents the number of iterations of the optimisation process (per core) and has a default value of 1 000 000. For this example, 200 000 iterations will be used to reduce the running time of ROptimus given that each iteration is more computationally demanding than in **Tutorial 1**. The variable `CYCLES` (unique to Acceptance Ratio Annealing ROptimus runs) denotes the number of acceptance ratio annealing cycles. Its default value is 10, however it will be set to 2 in this example so that each annealing cycle has 100 000 iterations just as in **Tutorial 1** (the number of steps per cycle is calculated as `NUMITER/CYCLES`). Lastly, the variable `DUMP.FREQ`, the frequency (in steps) with which the best found model is assessed and outputted by the function, will be set to 100 000 (its default value is 10 000).

Let us again investigate the Simulated Annealing (SA) version of ROptimus on 4 processors, which can be executed as follows:

```
Optimus(NCPU=4,
        OPT.TYPE="SA", OPTNAME="term_4_SA",
        NUMITER=200000, CYCLES=2, DUMP.FREQ=100000, LONG=FALSE,
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=DATA)
```

Interestingly, each of the 4 computing cores arrive at the same solution in this instance.

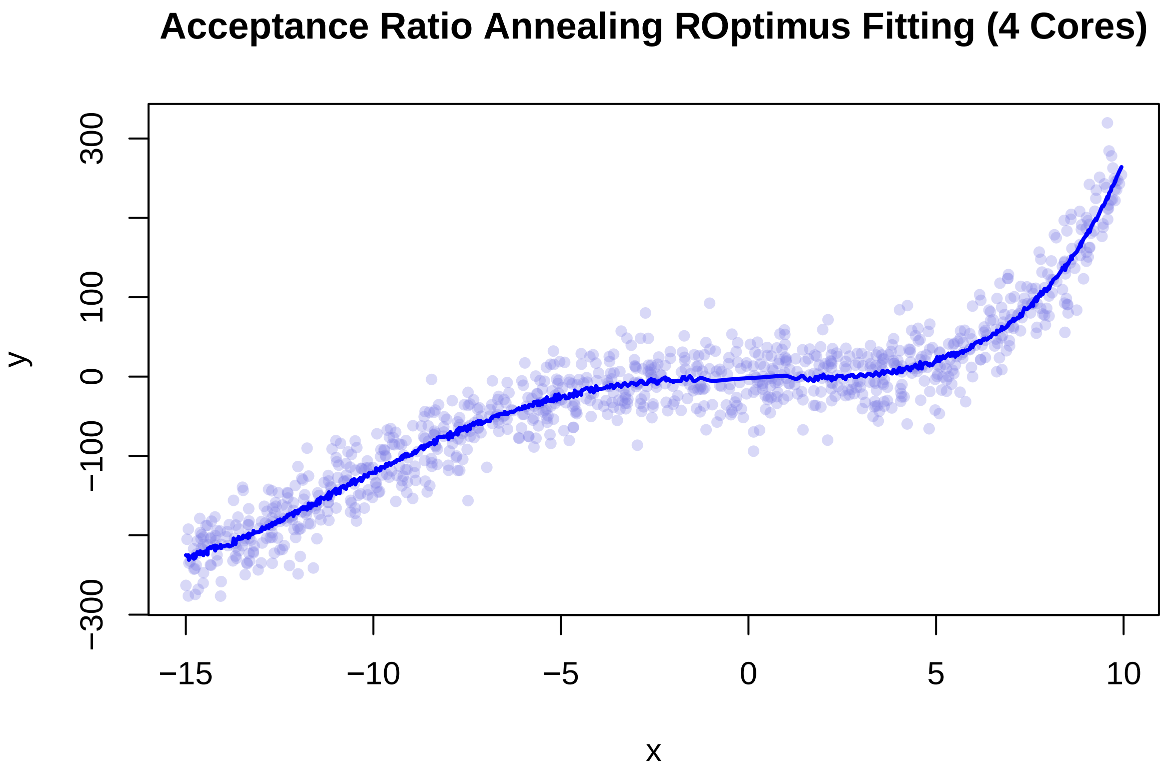


Table 4: 4-core Acceptance Ratio Simulated Annealing results from ROptimus.

	CPU 1	CPU 2	CPU 3	CPU 4
E (AIC)	9.567	9.567	9.567	9.567
Q (RMSD)	28.693	28.693	28.693	28.693
Term 1	0.000	0.000	0.000	0.000
Term 2	1.000	1.000	1.000	1.000
Term 3	1.000	1.000	1.000	1.000
Term 4	1.000	1.000	1.000	1.000
Term 5	0.000	0.000	0.000	0.000
Term 6	0.000	0.000	0.000	0.000
Term 7	0.000	0.000	0.000	0.000
Term 8	0.000	0.000	0.000	0.000
Term 9	0.000	0.000	0.000	0.000
Term 10	0.000	0.000	0.000	0.000
Term 11	1.000	1.000	1.000	1.000
Term 12	0.000	0.000	0.000	0.000
Term 13	0.000	0.000	0.000	0.000
Term 14	0.000	0.000	0.000	0.000
Term 15	0.000	0.000	0.000	0.000
Term 16	0.000	0.000	0.000	0.000

	CPU 1	CPU 2	CPU 3	CPU 4
Term 17	0.000	0.000	0.000	0.000
Term 18	0.000	0.000	0.000	0.000
Term 19	0.000	0.000	0.000	0.000
Term 20	1.000	1.000	1.000	1.000
Term 21	0.000	0.000	0.000	0.000
Term 22	0.000	0.000	0.000	0.000
Term 23	0.000	0.000	0.000	0.000
Term 24	0.000	0.000	0.000	0.000
Term 25	0.000	0.000	0.000	0.000
Term 26	1.000	1.000	1.000	1.000
Term 27	0.000	0.000	0.000	0.000
Term 28	0.000	0.000	0.000	0.000
Term 29	0.000	0.000	0.000	0.000
Term 30	0.000	0.000	0.000	0.000

Thus, the optimal functional representation found by ROptimus has the following form:

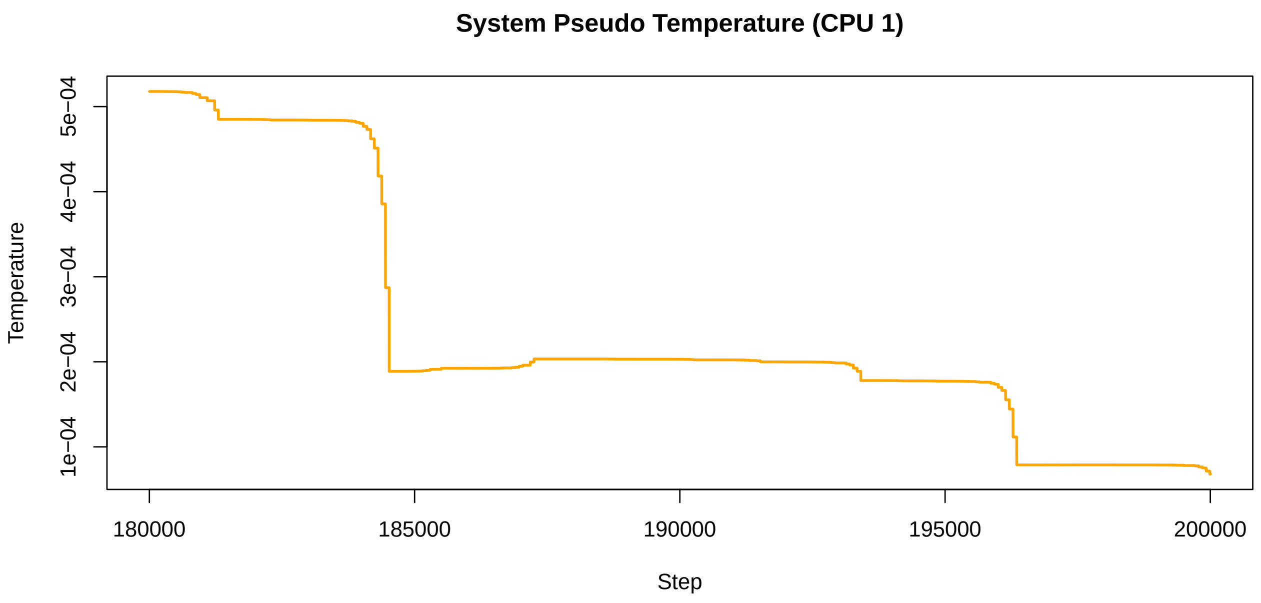
$$y = b + c_2x^2 + c_3x^3 + c_4x^4 + c_{11}e^x + c_{20}sin(x^3) + c_{26}cos(x^5)sin(-x)$$

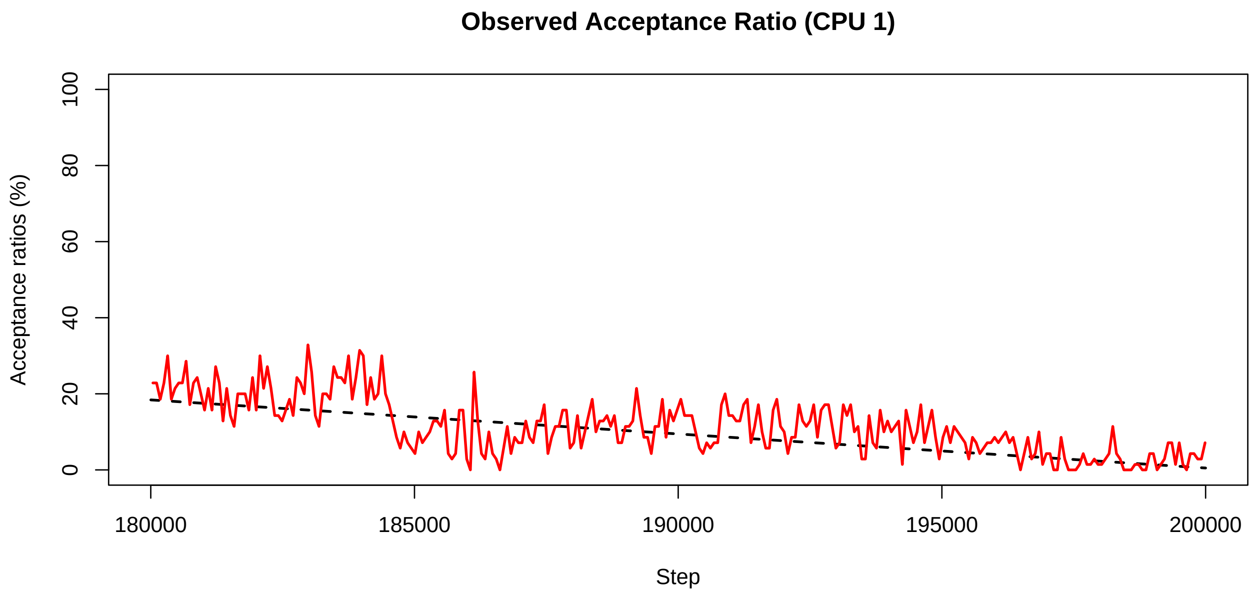
Below is the explicit representation after determining the coefficients c_i and b:

```
##
## Call:  glm(formula = equation, data = environment())
##
## Coefficients:
##          (Intercept)                I(x^2)                I(x^3)
##            -1.988699             -0.253437              0.178807
##               I(x^4)             I(exp(x))          I(sin(x^3))
##             0.008567              0.001569              1.878796
## I(cos(x^5) * sin(-x))
##            -2.770457
##
## Degrees of Freedom: 999 Total (i.e. Null);  993 Residual
## Null Deviance:       10850000
## Residual Deviance: 823300    AIC: 9567
```

Notice that the solution selected by ROptimus results in an RMSD of 28.693 which is lower than the RMSD of the Least Squares Solution (28.827) that assumes the appropriate model is $k_1x + k_2x^2 + k_3x^3 + k_4x^4$. ROptimus selected a model which does not include all terms from the form used to generate the data. If a user were concerned by the fact that the model ROptimus selected contains more terms (6) than are used in the representation of the de-noised data (4), the user could either increase the multiplicative factor associated with the term p in the AIC to more strongly penalise representations involving a greater number of parameters. Alternatively, the user could also modify the function `r()` to ensure that only a fixed number of terms are ever active.

Let us now take a look at how the adaptive thermoregulation performed given this highly non-smooth objective. The graphs below should now feel very familiar, they represent data taken from the last 20 000 iterations of the optimisation protocol executed by CPU 1.





Despite optimising a completely different model with a non-smooth objective function, the TCU succeeds in dynamically adjusting the system pseudo-temperature such that the observed acceptance ratio follows the annealing schedule rather well.

Acceptance Ratio Replica Exchange ROptimus Run

Let us now consider the Acceptance Ratio Replica Exchange version of ROptimus on 12 CPUs with the variable `ACCRATIO` defined as in **Tutorial 1**.

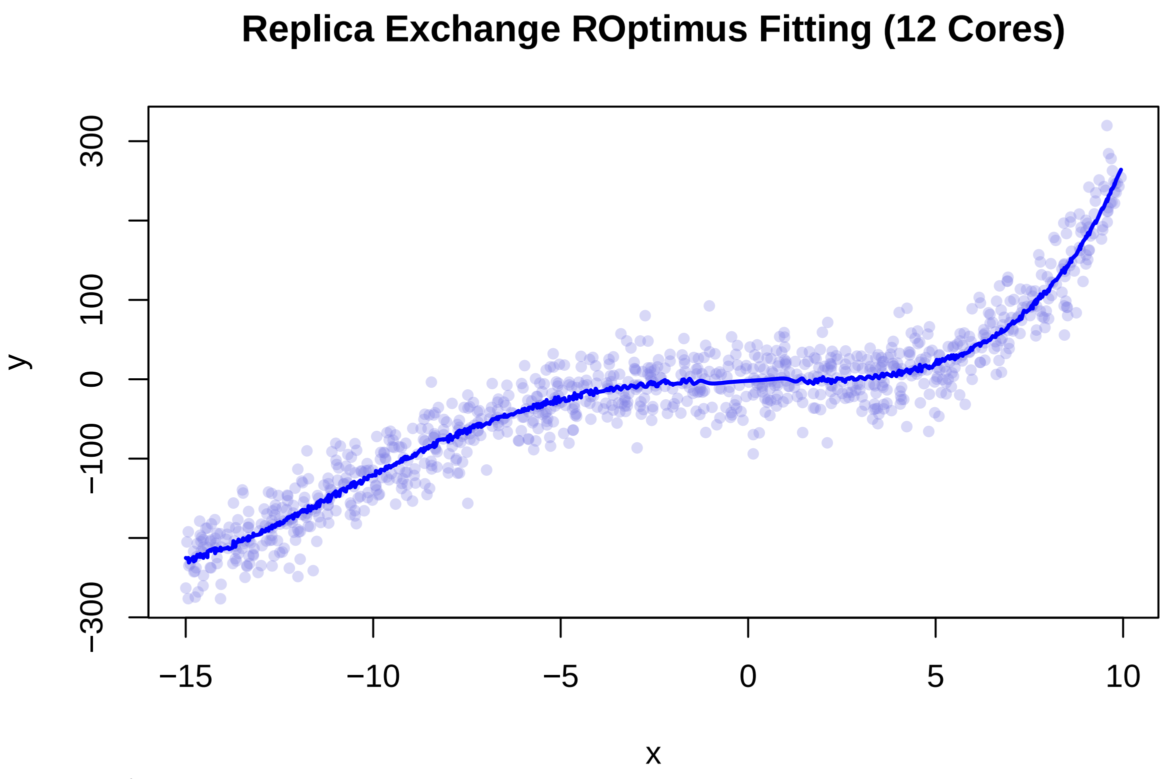
```
ACCRATIO <- c(90, 82, 74, 66, 58, 50, 42, 34, 26, 18, 10, 2)
```

As in the Acceptance Ratio Simulated Annealing run above, we will again execute the optimisation procedure for 200 000 iterations. The Replica Exchange version of ROptimus takes an input argument `EXCHANGE.FREQ` (default value 1000) which specifies the total number of exchanges that will occur during the optimisation process. Consequently, the number of optimisation iterations that occur between subsequent exchanges between replicas can be calculated as `NUMITER/EXCHANGE.FREQ`, which is 200 iterations in this case.

Here we will set the input parameter STATWINDOW to have value 50 (its default value is 70). This signifies that the TCU will update the system pseudo temperature once every 50 iterations on each optimisation replica. This guarantees that 4 temperature adjustments will be made if a given replica is involved in two subsequent exchanges (because the number of iterations between exchanges is 200, as explained in the preceding paragraph) as opposed to merely 2 adjustments which would be the case if STATWINDOW were left to take its default value and could result in poor agreement between the observed acceptance ratio of the replica in question and the target acceptance ratio. The following line executes ROptimus with the above specified inputs:

```
Optimus(NCPU=12, OPTNAME="term_12_RE",
        NUMITER=200000, STATWINDOW=50, DUMP.FREQ=100000, LONG=FALSE,
        OPT.TYPE="RE", ACCRATIO=ACCRATIO,
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=DATA)
```

Nine of the optimisation replicas (CPUs 1, 2, 3, 5, 6, 8, 9, 10 and 12) recovered the same solution that was found by the Acceptance Ratio Simulated Annealing ROptimus run. Moreover, this solution is better (lower *AIC*) than those recovered by CPUs 4, 7 and 11. Thus, in this example, the Acceptance Ratio Simulated Annealing and Replica Exchange versions produce the same solution.



Please note that for convenience, only those replicas that produced a unique solution are listed in the table below (replicas 1, 2, 3, 6, 8, 9, 10 and 12 produced the same solution as replica 5; replica 11 produced the same solution as replica 7).

Table 5: 12-core Acceptance Ratio Replica Exchange results from ROptimus run.

	CPU 4	CPU 5	CPU 7
Replica Acceptance Ratio	66.0000	58.0000	42.0000
E (AIC)	9.5673	9.5672	9.5673
Q (RMSD)	28.6664	28.6932	28.6951
Term 1	0.0000	0.0000	0.0000

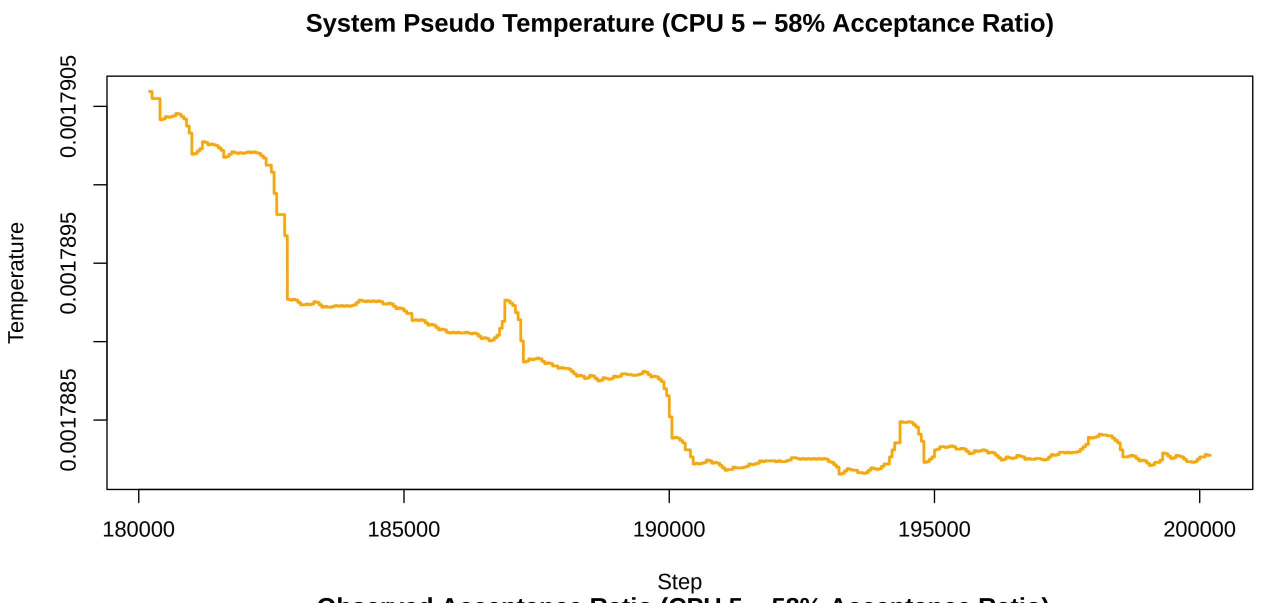
	CPU 4	CPU 5	CPU 7
Term 2	1.0000	1.0000	1.0000
Term 3	1.0000	1.0000	1.0000
Term 4	1.0000	1.0000	1.0000
Term 5	0.0000	0.0000	0.0000
Term 6	0.0000	0.0000	0.0000
Term 7	0.0000	0.0000	0.0000
Term 8	0.0000	0.0000	0.0000
Term 9	0.0000	0.0000	0.0000
Term 10	0.0000	0.0000	0.0000
Term 11	1.0000	1.0000	0.0000
Term 12	0.0000	0.0000	0.0000
Term 13	1.0000	0.0000	1.0000
Term 14	0.0000	0.0000	0.0000
Term 15	0.0000	0.0000	0.0000
Term 16	0.0000	0.0000	0.0000
Term 17	0.0000	0.0000	0.0000
Term 18	0.0000	0.0000	0.0000
Term 19	0.0000	0.0000	0.0000
Term 20	1.0000	1.0000	0.0000
Term 21	0.0000	0.0000	0.0000
Term 22	0.0000	0.0000	0.0000
Term 23	0.0000	0.0000	0.0000
Term 24	0.0000	0.0000	0.0000
Term 25	0.0000	0.0000	0.0000
Term 26	1.0000	1.0000	1.0000
Term 27	0.0000	0.0000	0.0000
Term 28	0.0000	0.0000	1.0000
Term 29	0.0000	0.0000	0.0000
Term 30	0.0000	0.0000	0.0000

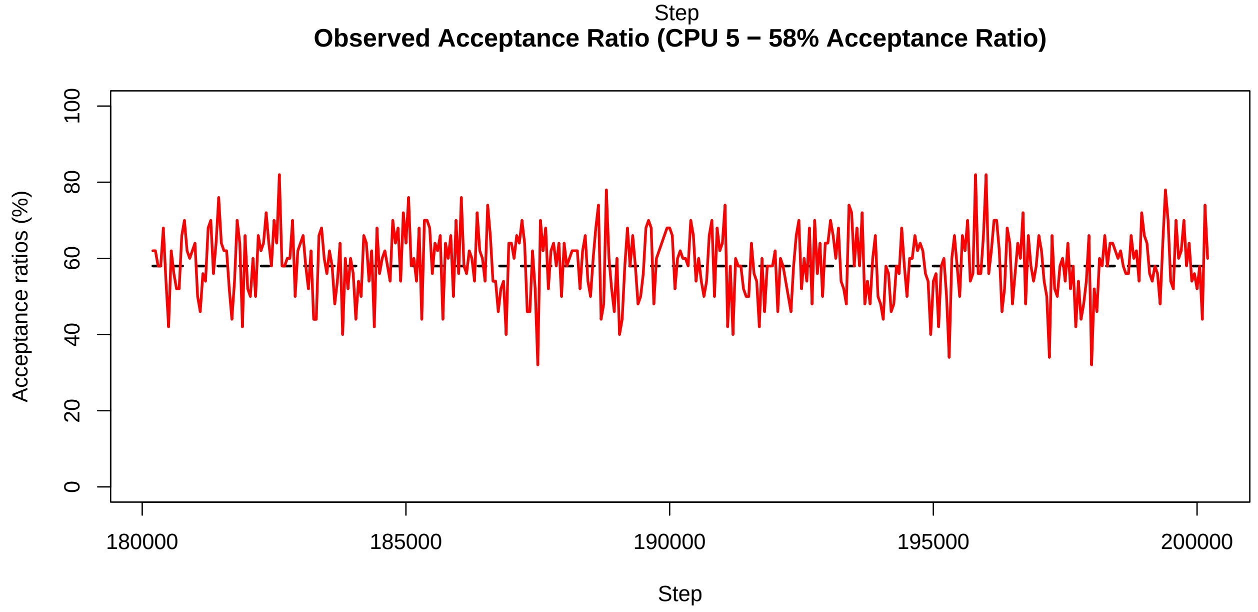
Note that the various replica outcomes illustrate the penalising effects of the *AIC* on models using a greater number of parameters. Consider the solutions found by the 66% acceptance ratio replica and the 58% acceptance ratio replica (CPUs 4 and 5 respectively). Let y_i denote the solution found by CPU i. Then, we have:

$$y_4 = b + c_2x^2 + c_3x^3 + c_4x^4 + c_{11}e^x + k_{13}c_{13}six(x) + c_{20}sin(x^3) + c_{26}cos(x^5)sin(-x)$$

$$y_5 = b + c_2x^2 + c_3x^3 + c_4x^4 + c_{11}e^x + c_{20}sin(x^3) + c_{26}cos(x^5)sin(-x)$$

Although the RMSD of y_4, 28.6664, is lower than the RMSD of y_5, 28.6932, y_5 has a lower value for *AIC* because it contains one less term than y_4. Since *AIC* was specified as the objective metric, ROptimus (perhaps counterintuitively) selected y_5 as the more optimal solution to reduce overfitting.





The above graphs are produced using data from the last 20 000 iterations of the 58% acceptance ratio replica (CPU 5). It is clear that the observed acceptance ratio more strongly oscillated around the target acceptance ratio than was the case in the Acceptance Ratio Simulated Annealing run from the previous part of this tutorial. More generally, it should be expected that the observed acceptance ratio fluctuates more significantly around the target acceptance ratio in Replica Exchange than in Simulated Annealing, especially when the objective function is non-smooth as is the case in this example. This is because an exchange between two replicas has the same effect as restarting a Monte Carlo optimisation from a random initial configuration with a temperature that very likely is not conducive to the target acceptance ratio for the given configuration. As such, each time an exchange occurs, significant deviations from the target acceptance ratio may occur and may require several `STATWINDOW`s for the TCU to correct. Despite this challenge, the TCU performs satisfactorily.

Summary

We now understand how to employ ROptimus to solve a more general problem than was addressed in **Tutorial 1** and one with a non-smooth objective function. Additionally, we have a better understanding of the adaptive thermoregulation. Using the Akaike Information Criterion (AIC) as a metric with which to evaluate the performance of a candidate model, taking into account the desire to represent the data while avoiding to overfit the data, both the Acceptance Ratio Simulated Annealing and Replica Exchange versions of ROptimus recovered a better functional form to describe the data than the form which was assumed in **Tutorial 1** (based on how the data had been generated), obviously with some overfitting to adapt to the noise while with the stringency used in this example.

Table 6: Summary of solutions.

	E (AIC)	Q (RMSD)
Least Squares (Tutorial 1)	9.5705	28.82655
ROptimus (AR Simulated Annealing)	9.5672	28.69324
ROptimus (AR Replica Exchange)	9.5672	28.69324

Least Squares (**Tutorial 1**):

$$y = c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4$$

ROptimus (Acceptance Ratio Simulated Annealing):

$$y = b + c_2 x^2 + c_3 x^3 + c_4 x^4 + c_{11} e^x + c_{20} sin(x^3) + c_{26} cos(x^5) sin(-x)$$

ROptimus (Acceptance Ratio Replica Exchange):

$$y = b + c_2 x^2 + c_3 x^3 + c_4 x^4 + c_{11} e^x + c_{20} sin(x^3) + c_{26} cos(x^5) sin(-x)$$

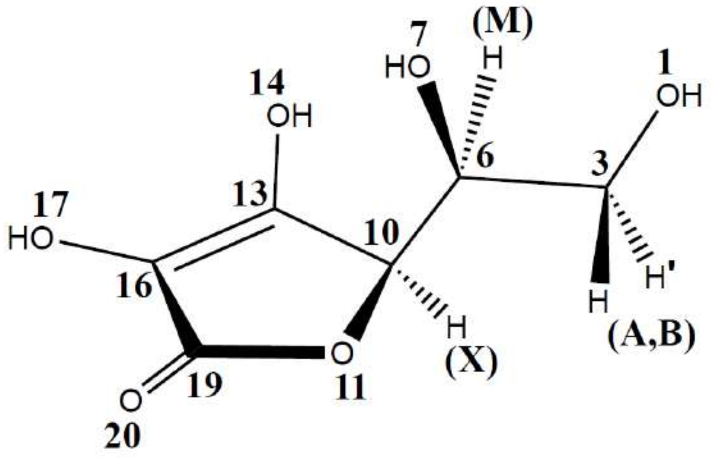
Tutorial 3: Geometry Optimisation of Vitamin C Molecule

Problem Statement

The focus of this tutorial is to depart from problem classes involving the search for functions to represent data, and demonstrate how ROptimus can be flexibly applied to arbitrary problem classes provided that they are formulated in accordance with ROptimus specifications. Additionally, this tutorial will illustrate that ROptimus can act as an optimisation kernel while calling external programs to execute a significant amount of the necessary computation for the optimisation process.

In this tutorial, ROptimus will be used, as an illustrative example, to 3D geometry optimise a molecular structure. Specifically, ROptimus will be used to determine the optimal values of two dihedral angles in the L-ascorbic acid (Vitamin C) molecule such that the molecule is in its ground state energy conformation. Vitamin C was selected to be the studied molecule because it has more than one freely rotating carbon-carbon bond and the potential for intramolecular hydrogen bonding due to the presence of multiple hydroxyl groups. Moreover, Vitamin C is not a particularly large molecule. Due to these circumstances, Vitamin C can serve as a non-trivial case (as opposed to simpler molecules like ethane for instance), but one that does not require several days or weeks of calculations to arrive to optimal solutions (the optimisation procedures below took roughly 14-18 hours to terminate).

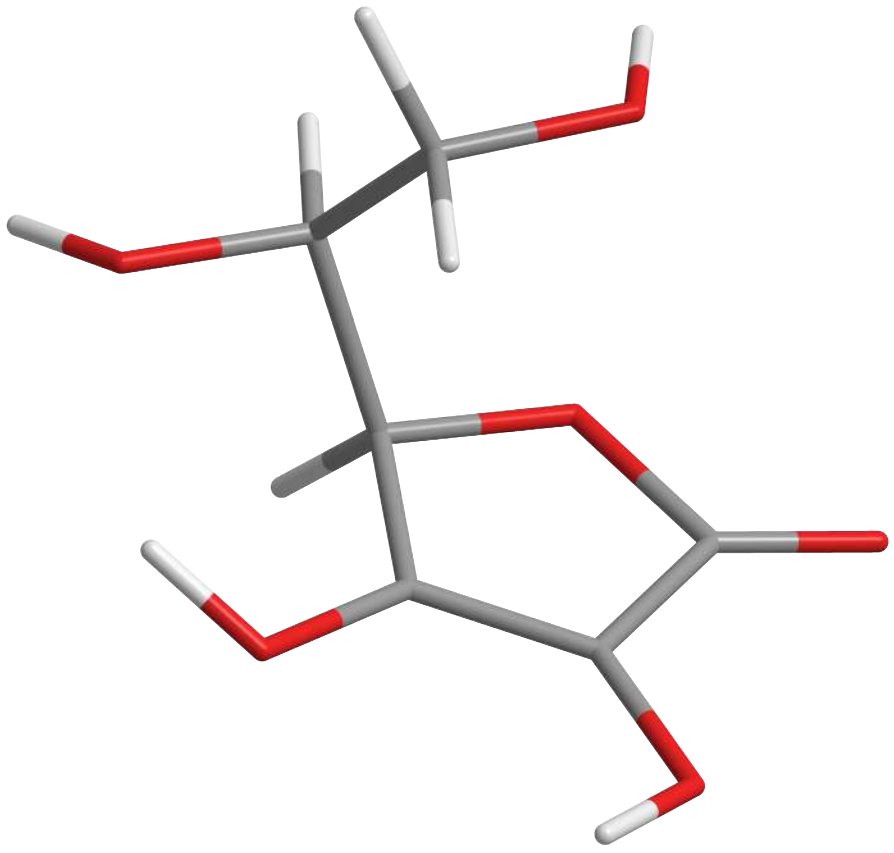
This is the molecular structure of Vitamin C, with the numbering of non-hydrogen atoms provided from the scheme used in geometry specification:

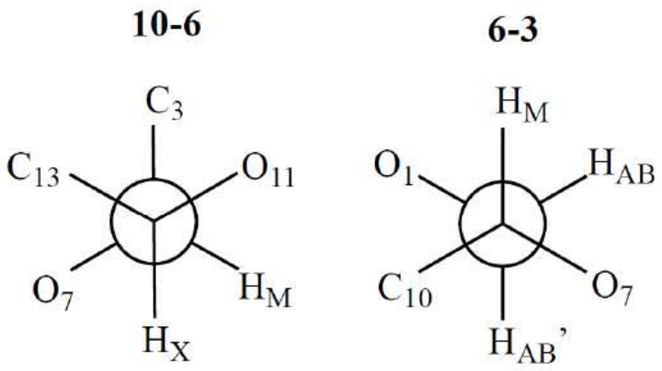


The major geometric features that drive the overall state of the molecule are the two C-C bonds in this structure: the bond joining carbon 3 and 6, and the bond joining carbon 6 and 10. The ground state conformation of Vitamin C will likely be a conformation such that steric clashes are minimised while also allowing for close proximity and right orientation between hydrogen bond donor and acceptor atoms. In the following sections, we formalise this optimisation problem and use ROptimus to arrive at the solution.

Defining ROptimus Inputs

As in the previous tutorials, we must first rigorously define the parameters that we are optimising. Let us begin by defining a dihedral angle as it will be used to our molecular geometry: a dihedral angle is the angle between two intersecting planes, where each plane is specified by 3 atoms of which 2 are common between both planes. Thus, a total of 4 atoms are needed to specify a dihedral angle. The conformation of Vitamin C with respect to its two freely rotating C-C bonds can be specified *via* two dihedral angles. Let ψ be the dihedral angle defined by the atoms numbered 1, 3, 6 and 7 and let ϕ be the dihedral angle defined by the atoms numbered 7, 6, 10 and 11. Having defined these two angles, we can now define the parameter set `K` as a numeric vector of length 2 whose entries are ψ and ϕ. We will arbitrarily initialise ψ and ϕ to have value 180. The corresponding Vitamin C conformation is illustrated below using a 3D structure and Newman projections along the two rotatable carbon-carbon bonds:





In the 3D structure, grey denotes carbon, red denotes oxygen and white denotes hydrogen atoms.

```
K <- c(PHI=180, PSI=180)
```

Now we will specify a model function `m()` that will operate on `K`. Starting from an arbitrary molecular conformation, altering the value of `K` will likely cause certain clashes or non-optimal interactions between atoms in the molecule that are not used in the definition of the angles ψ and ϕ. As such, after receiving an input set of parameters `K`, `m()` will have to alter the 3D location of constituents atoms while holding `K` fixed to arrive at the most stable geometry for the input `K`. Here, unlike in previous tutorials, to accomplish this task `m()` will call an external program MOPAC. MOPAC is a program for semiempirical quantum mechanics (QM) calculations, and can perform constrained and unconstrained geometry optimisations to arrive at a stationary state (note that calling MOPAC for a single initial geometry instance does not guarantee a global minimum will be found). MOPAC takes as input the specification of an initial molecular geometry in addition to an indication of which molecules the program is able to displace (or angles it can alter) and outputs a nearby local minimum molecular conformation with its corresponding energy in kcal/mol. For this optimisation problem, the input to MOPAC will be structured as a Z matrix, a common form for describing a molecular conformation which consists of using lengths, angles and dihedral angles with respect to previously defined atoms to define new atoms in the conformation.

The function `m()` will construct a Z matrix for Vitamin C using the input dihedral angles `K` and default values for the remaining geometric relationships needed to define the molecule. `m()` will then call MOPAC with the newly constructed Z matrix, specifying that all relationships may be altered by QM optimisation, except the input dihedral angles `K`. Finally, `m()` will return the energy calculated by MOPAC *via* PM6 Hamiltonian.

Note that to avoid non-convergence issues when calling MOPAC, `m()` returns a default energy value of -100 kcal/mol if a call to MOPAC does not terminate within 10 seconds (over-simplifications just for the sake of this illustrative example). Also, note that although `m()` requires no additional data on top of `K` to operate, `m()` must still be defined to take an input `DATA` in accordance with ROptimus specifications. Lastly, note that a local installation of MOPAC (2016) is required to execute this optimisation procedure. Below is the definition of `m()`:

```
m <- function(K, DATA){

  notconvergedE = -100.00
  # this should be your local path to MOPAC
  mopac.cmd = "/home/group/prog/mopac2016/MOPAC2016.exe"
  clean = TRUE

  # MOPAC semiempirical QM input file preparation, with given PHI and PSI
  # dihedral angles.

  geo <- c(
    "RHF PM6 EF GEO-OK MMOK T=10 THREADS=1",
    "Vitamin C with two controllable dihedral angles psi(7,6,3,1) and phi(11,10,6,7)",
    "  ",
    "O     0.00000000  0    0.0000000  0    0.0000000  0     0     0     0",
    "H     0.98468620  1    0.0000000  0    0.0000000  0     1     0     0",
    "C     1.43651250  1  110.7230618  1    0.0000000  0     1     2     0",
    "H     1.10751723  1  103.6603154  1 -167.5282722  1     3     1     2",
    "H     1.10658657  1  110.2236860  1  -51.3620456  1     3     1     2",
    "C     1.53950336  1  112.8074046  1 -123.2791585  1     3     4     5",
    paste0("O     1.42824262  1  103.4315186  1 ",K["PSI"]," 0     6     3     1"),
    "H     0.99584949  1  109.9022382  1 -165.7055126  1     7     6     3",
    "H     1.11472171  1  108.4417082  1   75.1535637  1     6     7     8",
    "C     1.54244170  1  109.4042184  1 -120.8240216  1     6     7     9",
    paste0("O     1.46313669  1  105.7792445  1 ",K["PHI"]," 0    10     6     7"),
    "H     1.11252563  1  112.8336666  1 -114.5813834  1    10     6    11",
    "C     1.51686608  1  113.4849244  1 -112.8332453  1    10    12    11",
    "O     1.34410484  1  125.3617342  1  179.6090511  1    13    10    11",
    "H     1.03381724  1  110.9736522  1  -13.3419919  1    14    13    10",
    "C     1.36084908  1  124.8906459  1  167.6242325  1    13    14    15",
    "O     1.35614887  1  131.9374989  1   -0.0333000  1    16    13    14",
    "H     1.00338885  1  109.4220239  1    0.3798200  1    17    16    13",
    "C     1.49109250  1  118.0837177  1 -179.7749947  1    16    17    18",
    "O     1.18961787  1  136.9144035  1   -0.6060924  1    19    16    17",
    "  "
  )

  # Submitting the MOPAC optimisation job, where all the spatial parameters
  # are relaxed except the pre-set PHI and PSI angles. The job is run requesting
  # maximum 10 seconds of time limitation. Most (if not all) complete within
  # half a second. Cases with unrealistic clashes will likely take much longer,
  # hence the job will be interrupted and notconvergedE value will be returned
  # for the energy evaluation.
  random.id <- as.character(sample(size=1, x=1:10000000))
  write(geo, file=paste0(random.id,".mop"))
  system(paste0(mopac.cmd," ",random.id,".mop"))

  if( file.exists(paste0(random.id,".arc")) ){
    e.line <- grep("HEAT OF FORMATION",
                   readLines(paste0(random.id,".arc")),
                   value=TRUE)
    e.line <-  strsplit(e.line," ")[[1]]
    O <- as.numeric(e.line[e.line!=""][5])
  } else {
    O <- notconvergedE
  }

  if(clean){
    file.remove(grep(random.id, dir(), value=TRUE))
  }

  return(O) # heat of formation in kcal/mol
}
```

Next, we define the function `u()` which returns an energy `E` and a quality `Q` of the candidate solution. Since the `m()` will already output a value for the physical energy of the candidate Vitamin C conformation, `u()` can simply set `E` to be the same return value of `m()`. We will make `u()` set `Q` to be the negative of the return value of `m()` such that candidate conformations with lower energies produce higher values of quality `Q`. Again, although `u()` does not require any additional data to accomplish this functionality, it must nevertheless be written to optionally accept an input parameter `DATA`.

```
u <- function(O, DATA){
  result   <- NULL
  result$Q <- -O
  result$E <-  O
  return(result)
}
```

Finally, we define the alteration function `r()`. `r()` will randomly select either ψ or ϕ to alter. Thereafter, `r()` randomly increases or decreases the selected angle by 2 degrees. `r()` will also ensure that $\psi, \phi \in [-180.0, 180.0]$ throughout the optimisation process.

```
r <- function(K){
  K.new <- K
  # Setting the alteration angle to 2 degrees:
  alter.by <- 2
  # Randomly selecting a term:
  K.ind.toalter <- sample(size=1, x=1:length(K.new))
  # Altering that term by either +alter.by or -alter.by
  K.new[K.ind.toalter] <-
    K.new[K.ind.toalter] + sample(size=1, x=c(alter.by, -alter.by))

  # Setting the dihedral angles to be always within the -180 to 180 range.
  if( K.new[K.ind.toalter] > 180.0 ){
    K.new[K.ind.toalter] <- K.new[K.ind.toalter] - 360
  }

  if( K.new[K.ind.toalter] < -180.0 ){
    K.new[K.ind.toalter] <- K.new[K.ind.toalter] + 360
  }

  return(K.new)
}
```

The process of determining the energy of a conformation corresponding to a given set of angles ψ, ϕ is the most computationally intensive part of this optimisation formulation. Having defined the necessary inputs for `Optimus()`, it should be apparent that this calculation will entirely be handled by MOPAC. This ability to serve as an optimisation kernel and flexibly be knitted to an external program is one of the many strengths of ROptimus.

Defining a Benchmark Solution

Before calling `Optimus()`, we have established a benchmark solution to be used to independently evaluate the ability of ROptimus to arrive to correct ψ and ϕ combination. In order to explore the energy landscape associated with the parameter space of ψ and ϕ, a PM6 optimisation was performed on 10 conformers picked from the wells pf a more comprehensive potential energy surface scanning through MM2 molecular mechanics force field (the details of PM6 and MM2 are not important for the purposes of this tutorial). This resulted in the identification of 7 energy minima, shown in the table below (listed in increasing order by energy):

Table 7: Seven conformational minima calculated for Vitamin C with PM6.

	E (kcal/mol)	PHI	PSI
Conformation 1	-233.206	92.58	74.19
Conformation 2	-232.877	50.60	-172.79
Conformation 3	-231.800	-169.67	-41.23
Conformation 4	-230.822	47.43	-166.61
Conformation 5	-230.274	-172.20	-54.45
Conformation 6	-225.214	-75.69	-104.44
Conformation 7	-224.875	-73.31	155.02

We will assume that the above listed conformations comprehensively represent most, if not all, of the possible minima in the parameter space of ψ and ϕ. Under this assumption, Conformation 1 should be considered as the ground state conformation of Vitamin C. The accuracy of the results produced by ROptimus can thus be judged by comparing them to the data listed in the above table. It is important to recognize that the "resolution" of ψ and ϕ when being optimised by ROptimus is set to 2 degrees due to the manner in which `r()` was defined. As such, results produced by ROptimus that are within plus or minus 2 degrees of a reference conformation should be tolerated.

Acceptance Ratio Simulated Annealing ROptimus Run

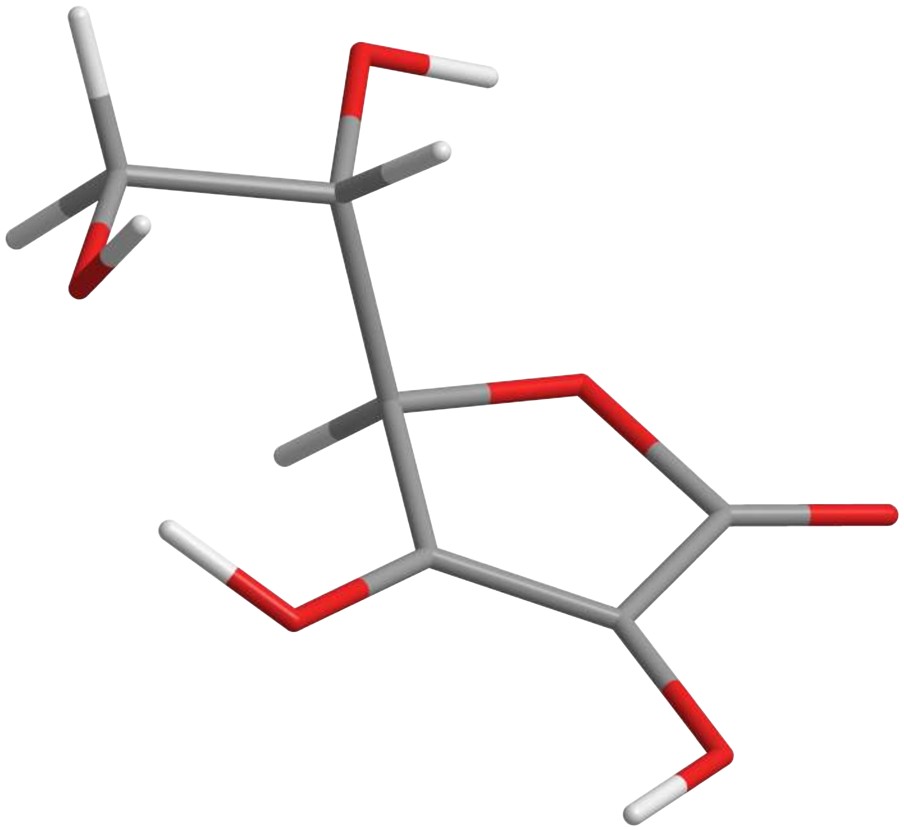
For the Acceptance Ratio Annealing run, we will set `NUMITER` to 100 000 because each optimsation step is more costly due to the relatively computationally expensive calls to MOPAC. Moreover, we will set `CYCLES` to 2. Although this shortens the length of an annealing cycle to 50 000 steps (whereas 100 000 steps per cycle has been kept constant over the previous tutorials), having more than 1 annealing cycle is likely more beneficial than insisting on a cycle lasting 100 000 steps as opposed to 50 000.

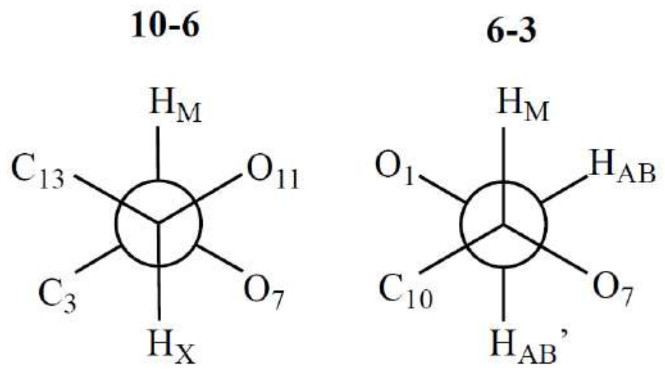
```
Optimus(NCPU=4, OPTNAME="vitamin_4_SA",
        NUMITER=100000, CYCLES=2, DUMP.FREQ=50000, LONG=FALSE,
        OPT.TYPE="SA",
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=NULL)
```

Table 8: 4-core Acceptance Ratio Simulated Annealing results from ROptimus runs.

	E (kcal/mol)	PHI	PSI
CPU 1	-232.874	50	-172
CPU 2	-232.353	-158	30
CPU 3	-232.874	50	-172
CPU 4	-232.874	50	-172

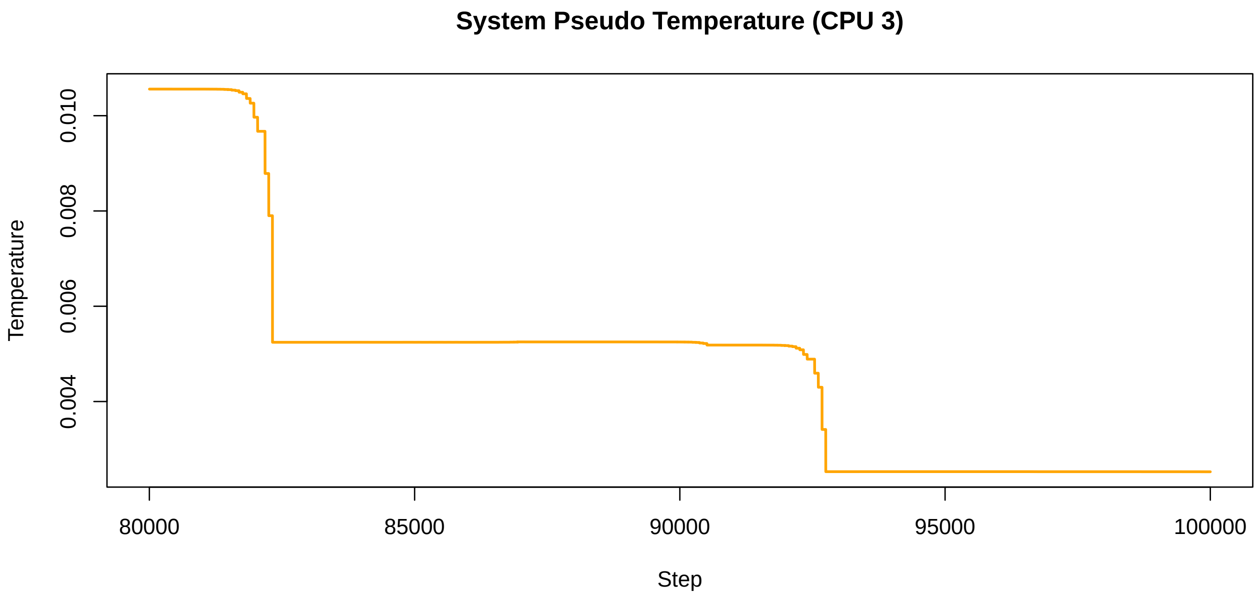
CPUs 1, 3 and 4 all arrived at a conformation defined by $\{\phi = 50, \psi = -172\}$, with an energy of -232.874 kcal/mol. The below 3D structure and Newman projections depict this solution:

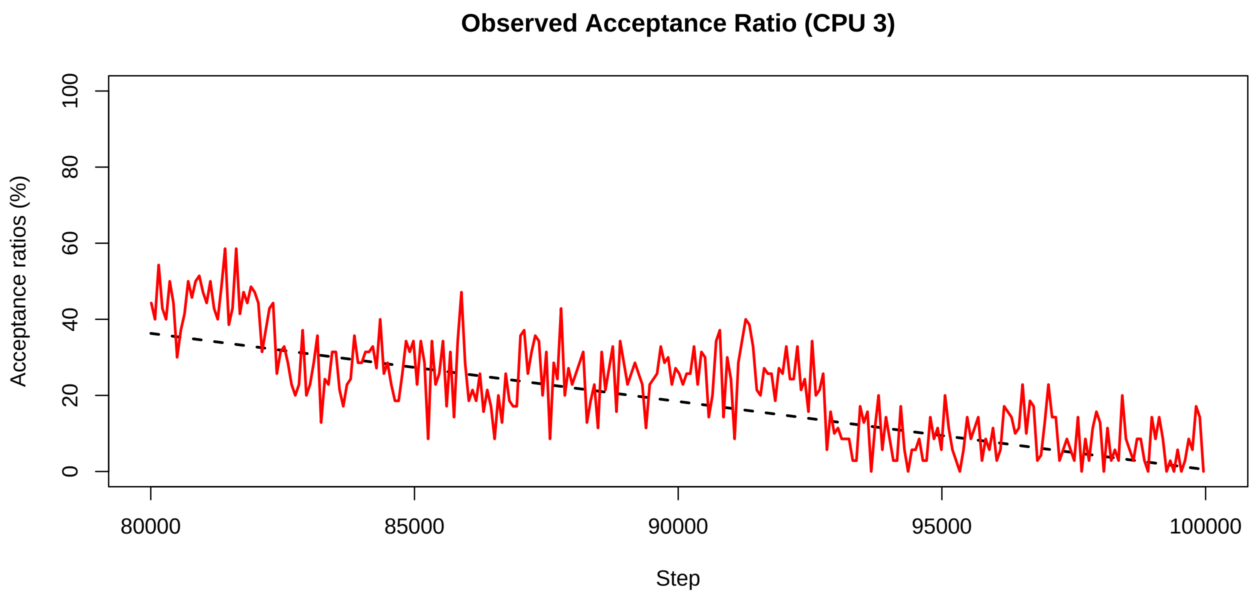




This conformation is equivalent to benchmark Conformation 2. Thus, in this example, Acceptance Ratio Simulated Annealing was able to find the Vitamin C conformation with the second lowest energy in the parameter space. This performance is strong, especially given that the limited steps and cycles executed, and that the energy difference between Conformation 1 and Conformation 2 is only -0.329 kcal/mol.

The graphs below illustrate the system psuedo temperature and observed acceptance ratio for the last 20 000 optimisation iterations executed by CPU 3.





Acceptance Ratio Replica Exchange ROptimus Run

Let us now consider the Replica Exchange version of ROptimus on 12 processors with the variable `ACCRATIO` defined as in the previous tutorials.

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

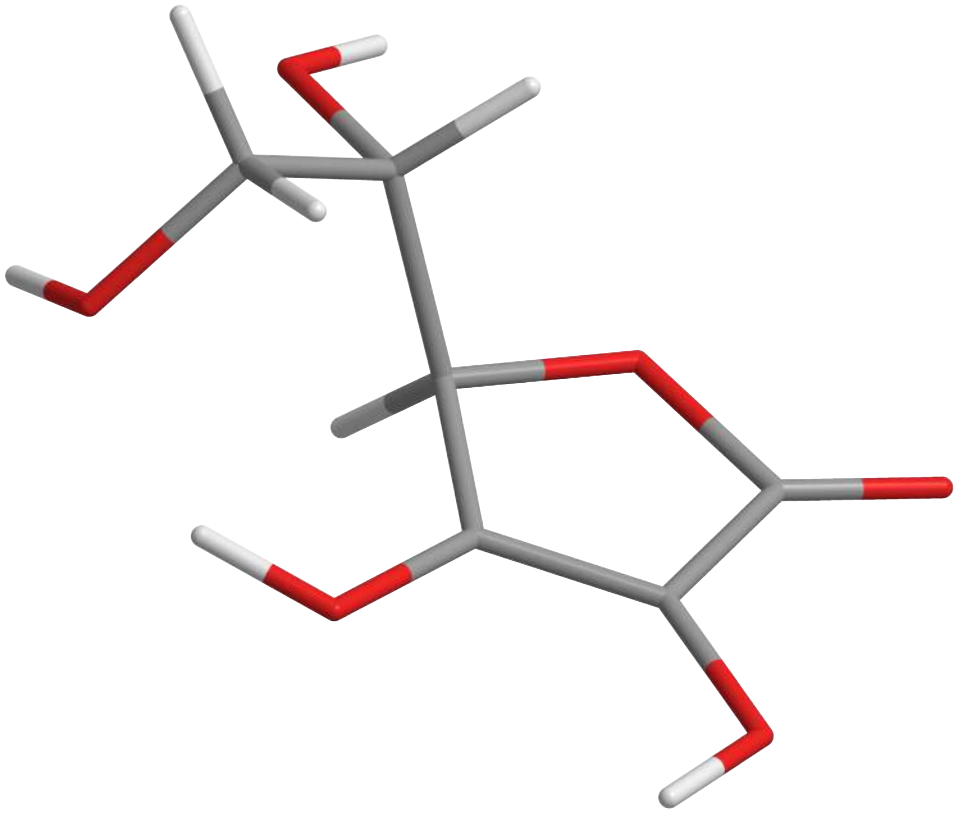
Just as in the Acceptance Ratio Simulated Annealing run, we will set `NUMITER` to 100 000. Furthermore, we will set `EXCHANGE.FREQ` to 500 such that the number of iterations between subsequent exchanges between replicas is 200 as it was in **Tutorial 2**. For the same reasons as in **Tutorial 2**, we will set `STATWINDOW` to 50 for the Replica Exchange run.

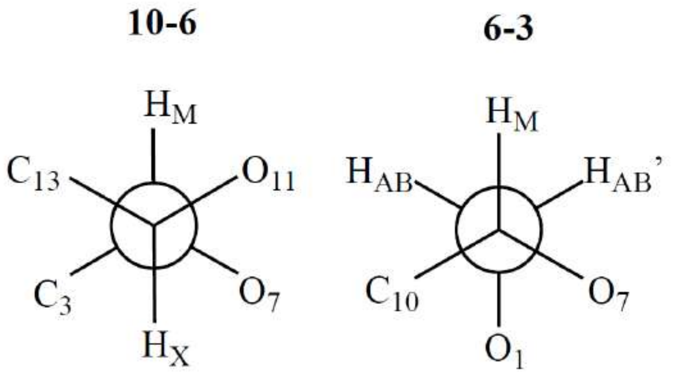
```
Optimus(NCPU=12, OPTNAME="vitamin_12_RE",
        NUMITER=100000, EXCHANGE.FREQ=500, STATWINDOW=50, DUMP.FREQ=50000, LONG=FALSE,
        OPT.TYPE="RE", ACCRATIO=ACCRATIO,
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=NULL)
```

Table 9: 12-core Acceptance Ratio Replica Exchange results from ROptimus run.

	Replica Acceptance Ratio	E (kcal/mol)	PHI	PSI
CPU 1	90	-229.2359	-164	-178
CPU 2	82	-229.2359	-164	-178
CPU 3	74	-233.1453	82	84
CPU 4	66	-233.1979	90	76
CPU 5	58	-229.2359	-164	-178
CPU 6	50	-232.8742	50	-172
CPU 7	42	-233.1947	94	74
CPU 8	34	-229.2359	-164	-178
CPU 9	26	-229.2359	-164	-178
CPU 10	18	-227.6394	180	158
CPU 11	10	-229.2359	-164	-178
CPU 12	2	-229.2359	-164	-178

Of the 12 replicas, CPU 4 recovered the conformation with the lowest energy (-233.1979), defined by $\{\phi = 90, \psi = 76\}$. The below 3D structure and Newman projections depict this solution:

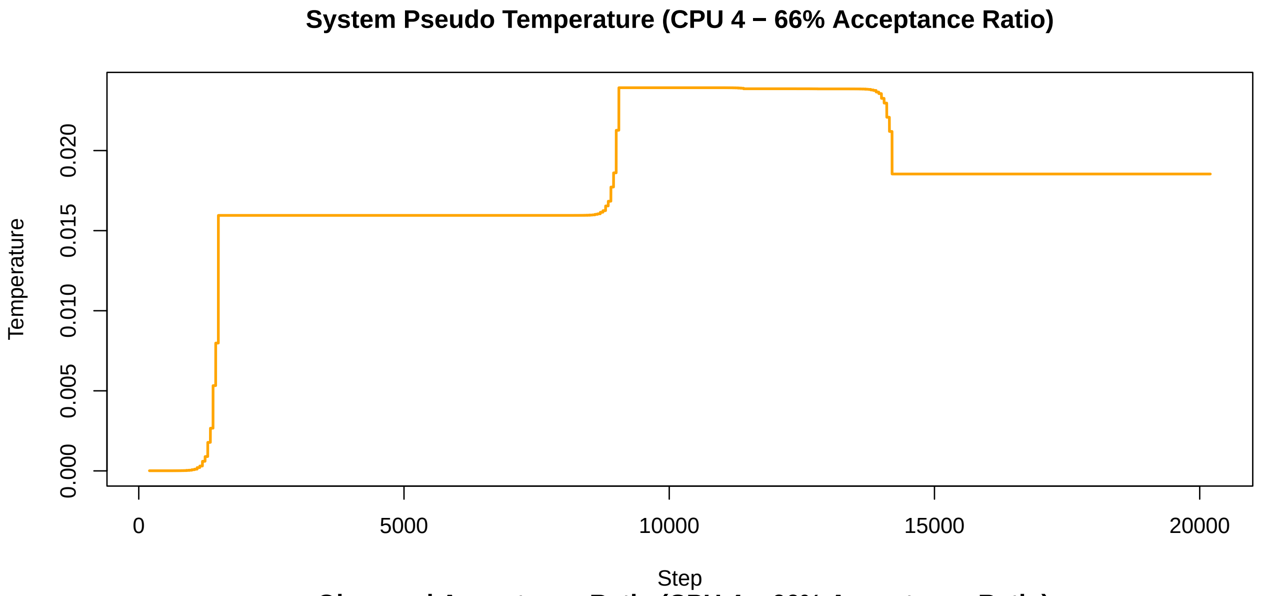


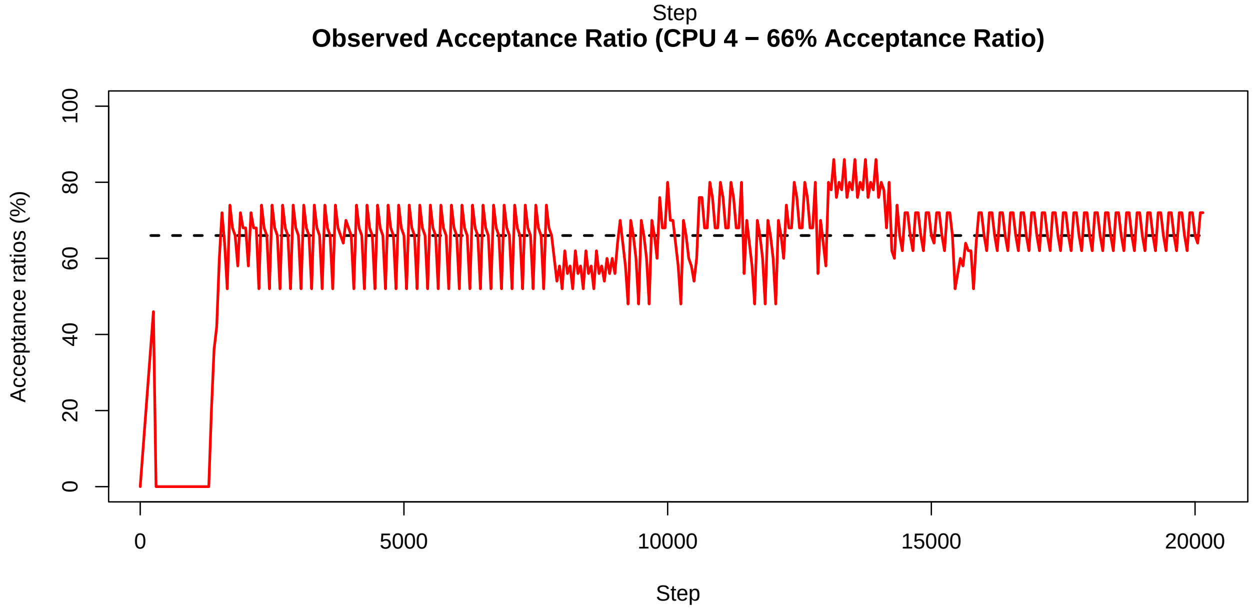


This solution corresponds to reference Conformation 1, the global minimum energy state for Vitamin C. Thus, for this optimisation problem, under the limits of the set number of steps and cycles, the Replica Exchange version of ROptimus outperformed Acceptance Ratio Annealing by succeeding in finding the global minimum of the energy landscape.

If we compare the solution found by CPU 4 to benchmark Conformation 1, it is evident that the value for ϕ found by ROptimus lies slightly outside of the plus or minus 2 degree window that was discussed earlier. Contrarily, CPU 7 finds a solution $\{\phi = 94, \psi = 74\}$ which does lie strictly within the resolution window. Despite this, the solution of CPU 4 has a slightly lower energy (-233.1979) than the solution of CPU 7 (-233.1947) and so represents a better solution. Finally, notice that Replica 6 recovered the same conformation that was identified by Acceptance Ratio Simulated Annealing ROptimus.

The below graphs illustrate the system pseudo temperature and observed acceptance ratio for the first 20 000 optimisation iterations executed by CPU 4 (66% acceptance ratio replica).





When the optimisation process is first initialised, it is very unlikely that the input initial temperature is conducive to the target acceptance ratio. As such, the adaptive thermoregulation alters the system pseudo-temperature considerably and rapidly to align the observed acceptance ratio with the target acceptance ratio, as can be seen in the above two graphs. Moreover, as stated in the previous tutorial, an exchange between two replicas often has a similar effect of introducing a parameter configuration that is not conducive to the current system pseudo temperature, which necessitates significant temperature adjustments. Accordingly, sharp increases or decreases in the system pseudo temperature and significant changes in the value around which the observed acceptance ratio oscillates in the graph above likely indicate steps at which an exchange involving replica 4 occurred.

Summary

We are now familiar with how to structure a more complex optimisation problem, involving an external program, to be solved with ROptimus as a kernel. On the particular example of geometry optimisation here, we saw that the Simulated Annealing mode of ROptimus was able to find the second lowest local minimum (under restricted number of annealing cycles), while the Replica Exchange mode recovered the global energy minimum.

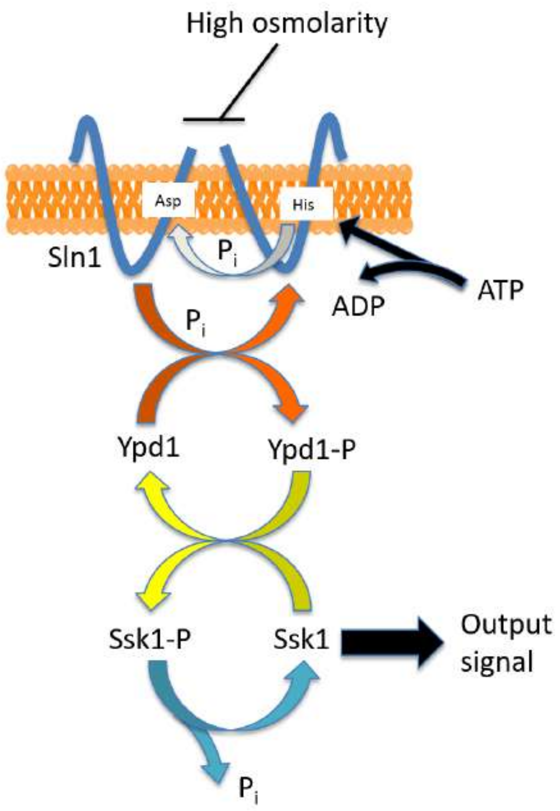
Table 10: Summary of solutions.

	Energy (kcal/mol)	PHI	PSI
Ground State Reference	-233.2060	92.58	74.19
ROptimus (AR Simulated Annealing)	-232.8740	50.00	-172.00
ROptimus (AR Replica Exchange)	-233.1979	90.00	76.00

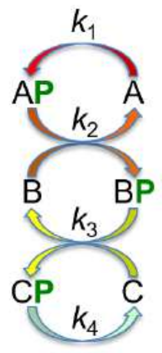
Tutorial 4: Exploring Coupled ODEs Modelling a Biological System

Problem Statement

This Tutorial will demonstrate the use of ROptimus to address a problem from yet another problem class. We will employ ROptimus to recover the rate constants for a system of coupled ordinary differential equations (ODEs) modelling a biological pathway. Specifically, we will study a phosphorelay system from the high osmolarity glycerol (HOG) pathway in yeast. A phosphorelay system is a network involving multiple proteins in which, after an initial phosphorylation event using ATP (or an alternate phosphate donor), the phosphorylation and dephosphorylation events of proteins in the network proceed without further consumption of ATP (Klipp et al. 2009). The below diagram illustrates the phosphorelay system that will be studied in detail (Klipp et al. 2009):



Under normal circumstances, the transmembrane protein Sln1, which is present as a dimer, autophosphorylates at a histidine residue (consuming ATP). The phosphate group is then transferred to an aspartate residue of Sln1. Thereafter, the phosphate is transferred to the protein Ypd1 and finally to the protein Ssk1. Ssk1 is continuously dephosphortylated to give an output signal. The signalling pathway is inhibited by an increase is osmolarity outside of the cell (Klipp et al. 2009). If we let A represent Sln1, B represent Ypd1, C represent Ssk1 and XP represent the phophorylated form of protein X, then the above network can be represented by the below schematic (Klipp et al. 2009):



where each k_i represents the rate constant for the relevant phosphorylation/dephosphorylation reaction.

The above graphic allows us to arrive at the following equations to describe the temporal behavior of the phosphorelay system:

$$\frac{d}{dt}[A] = -k_1[A] + k_2[AP][B]$$
$$\frac{d}{dt}[B] = -k_2[AP][B] + k_3[BP][C]$$
$$\frac{d}{dt}[C] = -k_3[BP][C] + k_4[CP]$$

Moreover, under the generally accepted assumption that the degradation and production of proteins occurs on a time scale that far exceeds that of phosphorylation events, we have the following conservation relationships (Klipp et al. 2009):

$$[A]_{total} = [A] + [AP]$$
$$[B]_{total} = [B] + [BP]$$
$$[C]_{total} = [C] + [CP]$$

where $[A]_{total}$, $[B]_{total}$ and $[C]_{total}$ are constants. Differentiating, we have:

$$\frac{d}{dt}[AP] = -\frac{d}{dt}[A]$$
$$\frac{d}{dt}[BP] = -\frac{d}{dt}[B]$$
$$\frac{d}{dt}[CP] = -\frac{d}{dt}[C]$$

Given this model of the phosphorelay system, the question we desire to answer is as follows: given initial concentrations of the three proteins $\{[A]_i, [B]_i, [C]_i\}$ and target concentrations of the three proteins $\{[A]_t, [B]_t, [C]_t\}$, what are the values $\{k_1, k_2, k_3, k_4\}$ that result in the proteins having the target concentrations at steady state when the system is allowed to equilibrate from the initial concentrations? This formulation assumes that no information is known about the rate constants and that initial and target concentrations can be determined experimentally, which is often the case in practice (Raue et al. 2013). The problem formulation could be altered depending on the information that is known or that can be determined experimentally.

Defining ROptimus Inputs

Having outlined how the behaviour of the phosphorelay system can be modelled using a system of differential equations, we can now proceed with defining input parameters for `Optimus()`. We will create a variable state that will be a numeric vector holding the names and initial concentrations of all species in the network. For this tutorial, we will choose $[A]_i = [B]_i = [C]_i = 100$ and $[AP]_i = [BP]_i = [CP]_i = 0$. Note that the units are arbitrary and that the total sum of units across this vector will remain constant throughout the simulation of the dynamics of the phosphorelay system.

```
state  <- c(cA=100, cB=100, cC=100, cAP=0, cBP=0, cCP=0)
```

Next, we will create a variable target which will be a numeric vector holding the names and target concentration of all species in the network. We will arbitrarily choose target values of $[A]_t = 90$, $[B]_t = 20$, $[C]_t = 70$, $[AP]_t = 10$, $[BP]_t = 80$ and $[CP]_t = 30$. Note that the chosen target values must be consistent with the above defined conservation equations, meaning we must have $[X]_i + [XP]_i = [X]_t + [XP]_t, \forall\, X \in \{A, B, C\}$.

```
target <- c(cA=90, cB=20, cC=70, cAP=10, cBP=80, cCP=30)
```

In order to determine the steady state behavior of the ODE system, we will employ the function `ode()` from the R library `deSolve` (this function interfaces with the Fortran library typically used to solve systems of differential equations). This function requires as input a function that describes the dynamics of the ODE system. We will call this function `model()`. At a high level, `model()` will simply define the equations derived in the previous section that describe the network. It should contain equations that use the objects with the names specified within state above, and should have equations that assign the outcomes to new objects that have the same order and names as specified in state, but with "d" at the beginning (a more detailed description of the requirements of `model()` can be found in the R documentation of `ode()`).

```
model <- function(t, state, K){

  with( as.list(c(state, K)), {
    # rate of change
    dcA   <- -k1*cA+k2*cAP*cB
    dcB   <- -k2*cAP*cB+k3*cBP*cC
    dcC   <- -k3*cBP*cC+k4*cCP
    dcAP  <- -dcA
    dcBP  <- -dcB
    dcCP  <- -dcC
    # return the rate of change
    list(c(dcA, dcB, dcC, dcAP, dcBP, dcCP))
  })
}
```

The variables `state` and `target`, and the function `model()` should be stored as entries in a list `DATA` which will be given to the functions `m()` and `u()` as inputs.

```
DATA <- NULL
DATA$state  <- state
DATA$target <- target
DATA$model  <- model
```

We will make `K` be a numeric vector holding the set of rate constants $\{k_1, k_2, k_3, k_4\}$. We will (arbitrarily) initialize each rate constant to have value 1.0.

```
K <- c(k1=1.0, k2=1.0, k3=1.0, k4=1.0)
```

The function `m()` will take as input the vector `K` of rate constants and the list `DATA`. It will return an object `O` that contains the concentrations of the six species in the network when the system is simulated from the initial state specified in `DATA` using the `K` rate constants for 10 time steps. Note that it is not necessarily guaranteed that the system will reach a steady state after 10 time steps; the number of time steps was chosen such that the optimisation procedure would terminate within 1-2 hours in this example. `m()` will call the function `ode()` from the library `deSolve`, so we must first ensure that `deSolve` is installed.

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

```
library(deSolve)
m <- function(K, DATA){
  state <- DATA$state
  model <- DATA$model

  span = 10.0

  times <- c(0, span)
  O   <- ode(y=state, times=times, func=model, parms=K)[2,2:(length(state)+1)]
  return(O)
}
```

Recall that the function `u()` must return an energy `E` and a quality `Q` of the candidate solution. Here, `u()` will set both `E` and `Q` to be the RMSD between the steady state concentrations of the network corresponding to the current set of rate constants `K`, as determined by `m()`, and the target concentrations.

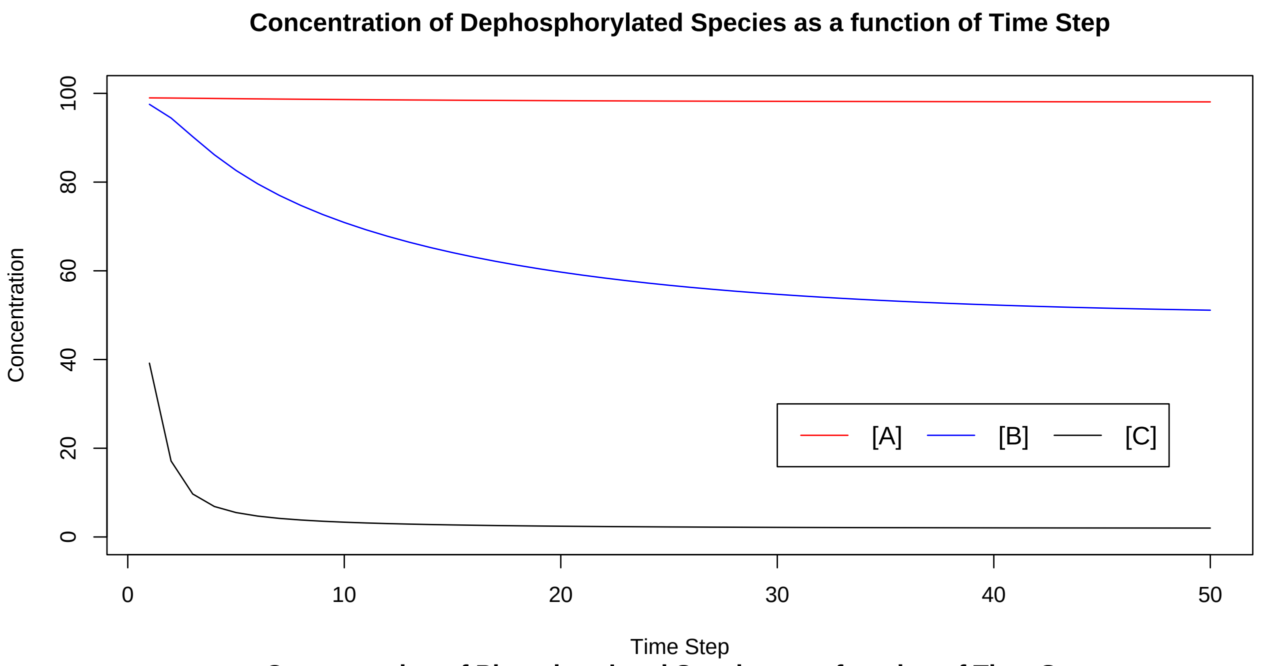
```
u <- function(O, DATA){
  target   <- DATA$target
  RESULT   <- NULL
  RESULT$Q <- sqrt(mean((O-target)^2)) # measure of the fit quality
  RESULT$E <- RESULT$Q # the pseudo energy derived from the above measure

  return(RESULT)
}
```

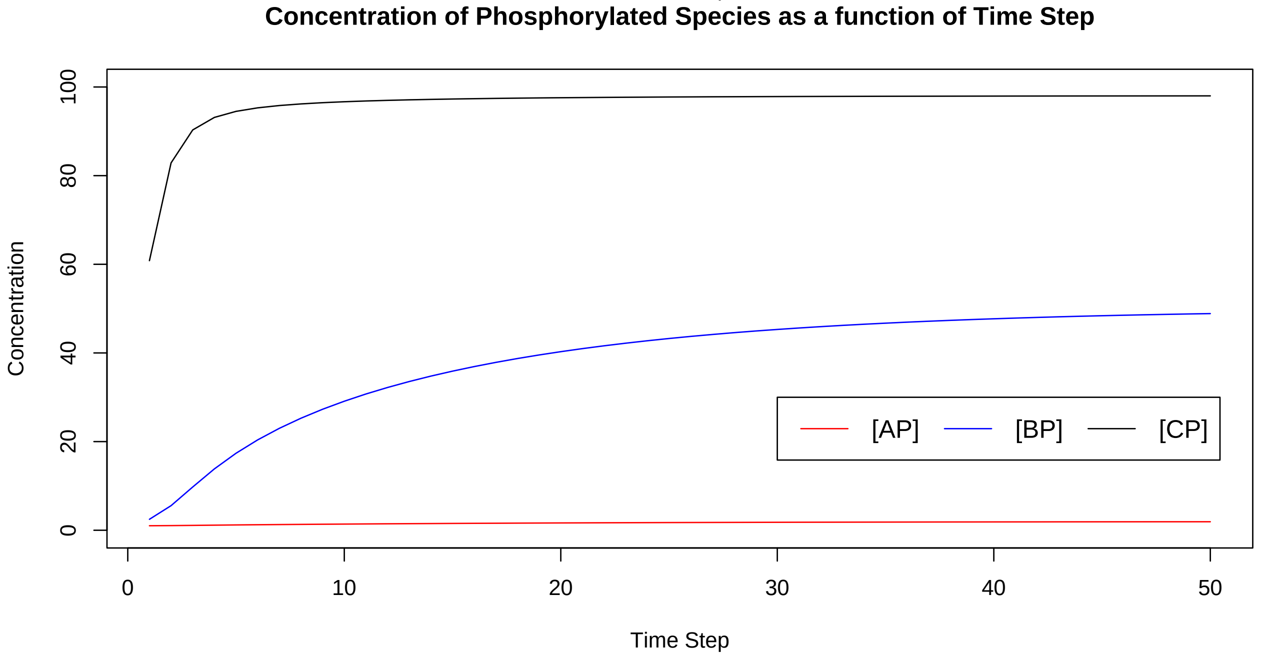
The final mandatory input to `Optimus()` that must be defined is the alteration function `r()`. Just as in **Tutorial 1**, for each snapshot of `K`, we shall randomly select one of its four coefficients, then either increment or decrement (chosen randomly) it by 0.0002, returning the altered set of coefficients. Since we are dealing with rate constants in this case, if ever `r()` were to make an entry in K negative, that entry will automatically be set to 0.

```
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 <- 0.0002
  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
  neg.ind <- which(K.new < 0)
  if(length(neg.ind)>0){ K.new[neg.ind] <- 0 }

  return(K.new)
}
```

Exploring the System Dynamics

Before calling `Optimus()` to solve this problem, let us first simulate the system of ODEs from the chosen initial state using a few sets of arbitrary rate constants to become familiar with how the system evolves. The below graphs illustrate the evolution of the system for 50 time steps for the rate constants $\{k_1 = 1.0, k_2 = 1.0, k_3 = 1.0, k_4 = 1.0\}$:





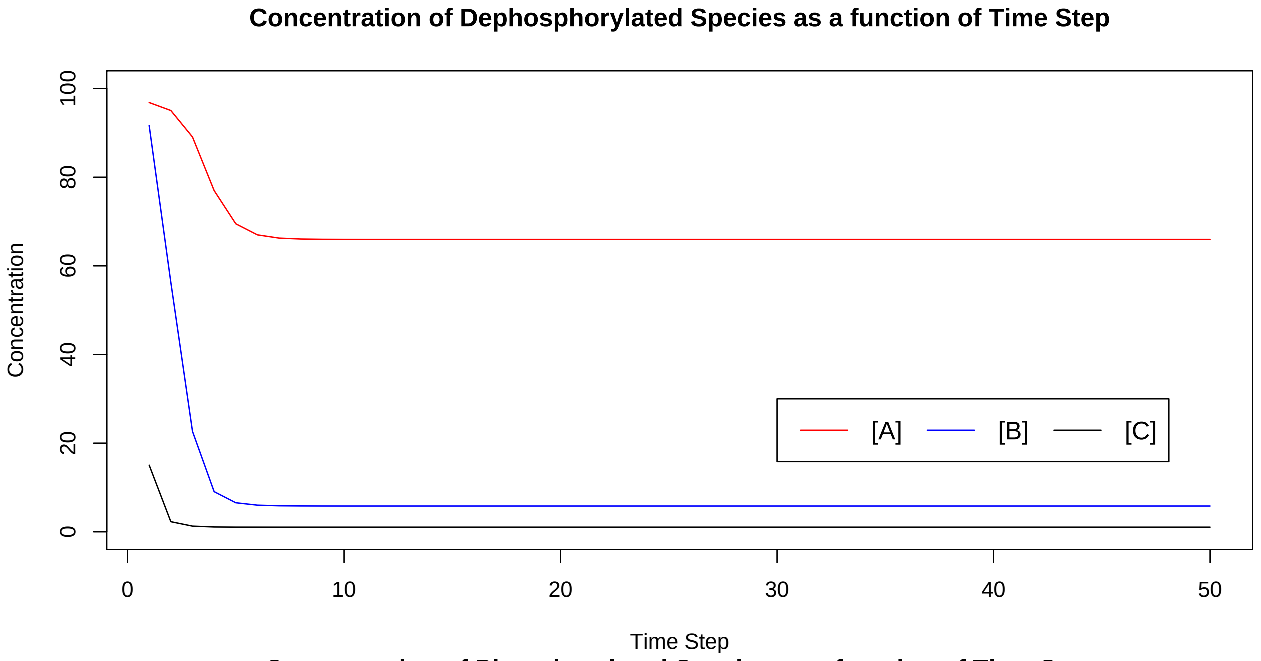
The table below summarises the initial and final concentrations of the various species when the system is simulated for 50 time steps using the rate constants $\{k_1 = 1.0, k_2 = 1.0, k_3 = 1.0, k_4 = 1.0\}$:

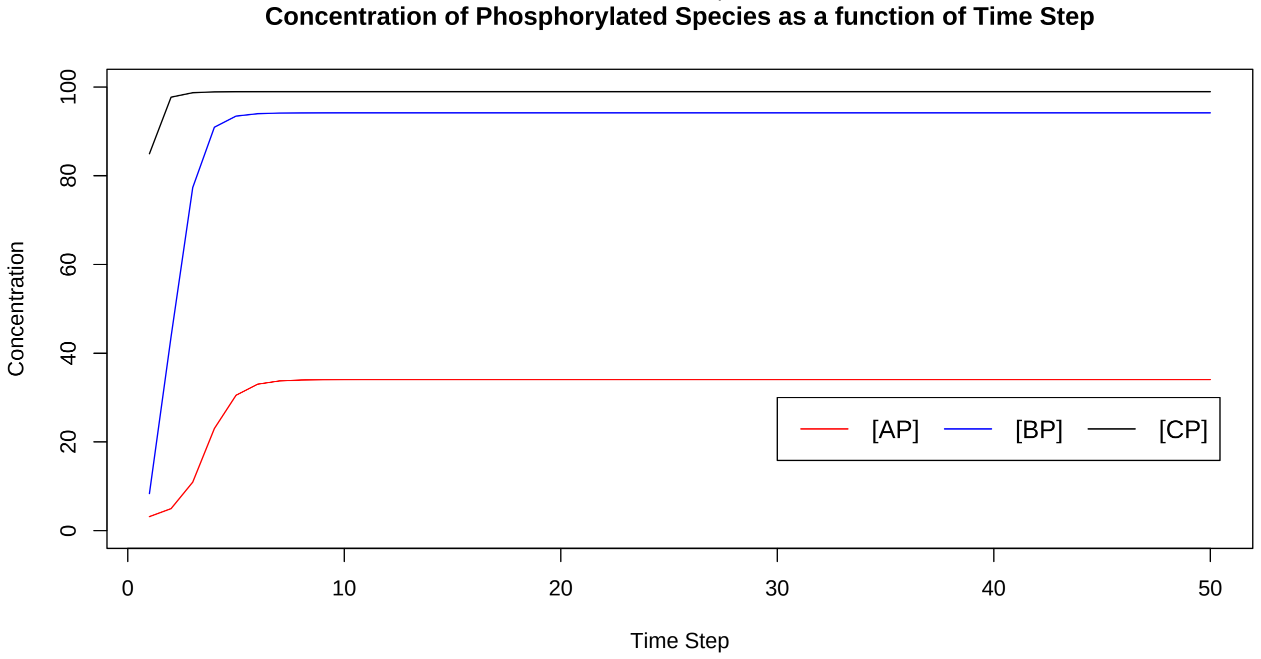
	[A]	[B]	[C]	[AP]	[BP]	[CP]

Table 11: System summary for k1 = k2 = k3 = k4 = 1.0.

	[A]	[B]	[C]	[AP]	[BP]	[CP]
Initial	100.00000	100.00000	100.000000	0.000000	0.00000	0.00000
Final (after 50 time steps)	98.08145	51.12118	2.004924	1.918549	48.87882	97.99508

If instead we use the set of rate constants $\{k_1 = 1.5, k_2 = 0.5, k_3 = 1.0, k_4 = 1.0\}$, the system evolves as follows:





The table below summarizes the initial and final concentrations of the various species when the system is simulated for 50 time steps using the rate constants $\{k_1 = 1.5, k_2 = 0.5, k_3 = 1.0, k_4 = 1.0\}$:

Table 12: System summary for k1 = 1.5, k2 = 0.5, k3 = k4 = 1.0.

	[A]	[B]	[C]	[AP]	[BP]	[CP]
Initial	100.00000	100.000000	100.000000	0.00000	0.00000	0.00000
Final (after 50 time steps)	65.96628	5.814787	1.050583	34.03372	94.18521	98.94942

Acceptance Ratio Simulated Annealing ROptimus Run

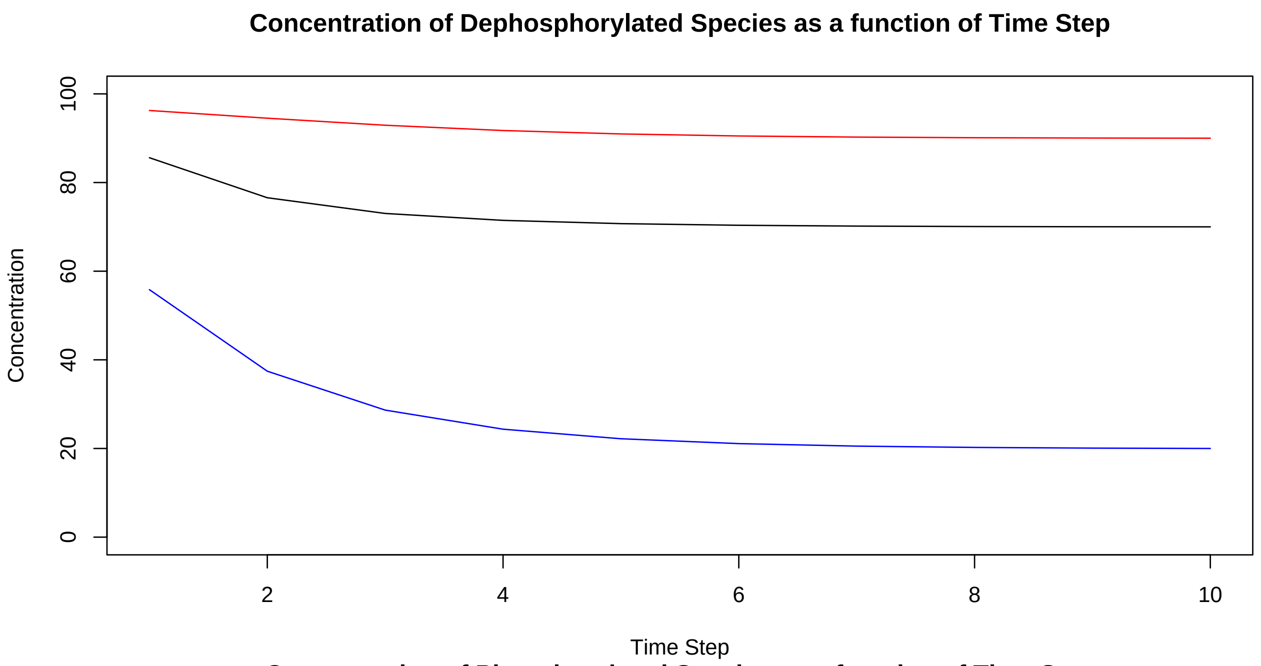
We will now call Acceptance Ratio Simulated Annealing ROptimus to solve our problem. Similarly to **Tutorial 2**, we will execute 200 000 optimisation iterations and perform 2 annealing cycles. We will set `DUMP.FREQ` to have a value of 100 000.

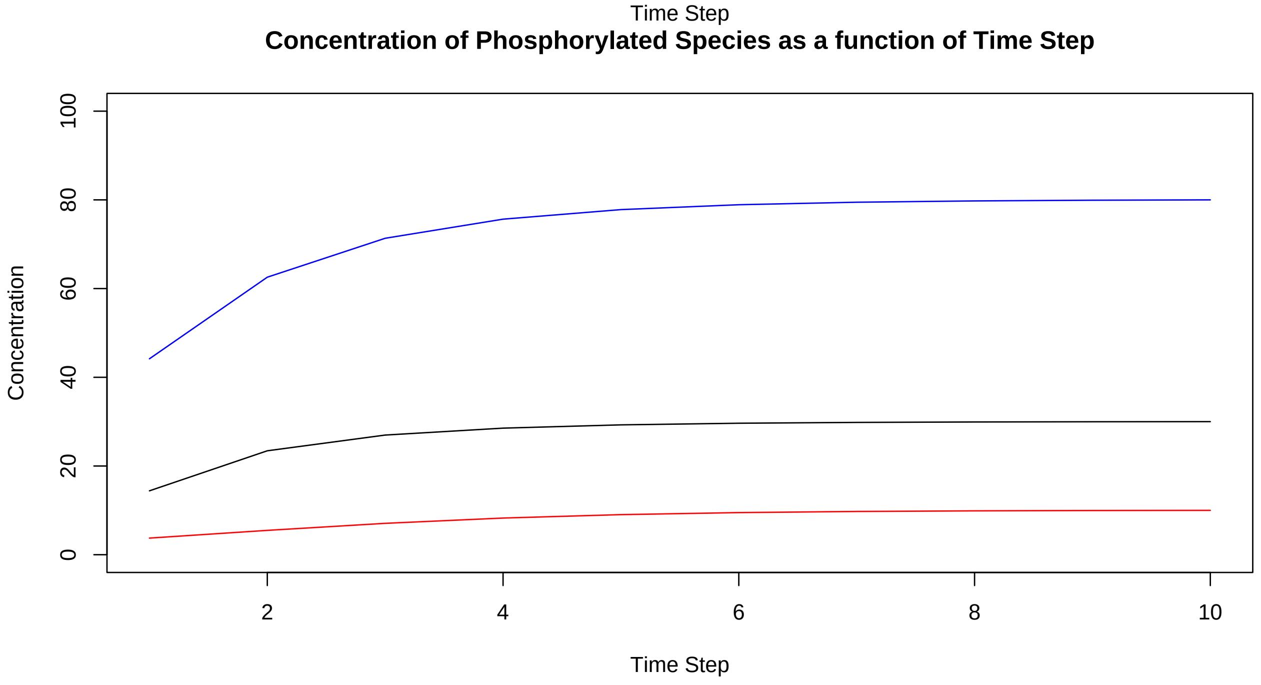
```
Optimus(NCPU=4, OPTNAME="DE_4_SA",
        NUMITER=200000, CYCLES=2, DUMP.FREQ=100000, LONG=FALSE,
        OPT.TYPE="SA",
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=DATA)
```

Table 13: 4-core Acceptance Ratio Simulated Annealing results from ROptimus run.

	E (RMSD)	K1	K2	K3	K4
CPU 1	0.0012516	0.7974	0.3586	0.0128	2.3886
CPU 2	0.0017556	0.7850	0.3532	0.0126	2.3512
CPU 3	0.0013625	0.8098	0.3642	0.0130	2.4262
CPU 4	0.0012516	0.7974	0.3586	0.0128	2.3886

Of the 4 optimisation replicas, CPU 1 and 4 find the best set of rate constants, $\{k_1 = 0.7974, k_2 = 0.3586, k_3 = 0.0128, k_4 = 2.3886\}$. This set of rate constants results in an RMSD (after 10 iterations) of 0.0012516. Let us simulate how the system evolves according to these rate constants for 10 time steps:





The table below summarizes the initial and final concentrations of the various species when the system is simulated for 10 time steps using the rate constants $\{k_1 = 0.7974, k_2 = 0.3586, k_3 = 0.0128, k_4 = 2.3886\}$:

Table 14: System summary for k1 = 0.7974, k2 = 0.3586, k3 = 0.0128, k4 = 2.3886.

	[A]	[B]	[C]	[AP]	[BP]	[CP]
Initial	100.00000	100.00000	100.00000	0.00000	0.00000	0.00000
Final (after 10 time steps)	89.99814	20.00081	69.99924	10.00186	79.99919	30.00076

As can be seen in the above table, the concentration of the species in the system after 10 time steps are very close to the target values $[A]_t = 90$, $[B]_t = 20$, $[C]_t = 70$, $[AP]_t = 10$, $[BP]_t = 80$ and $[CP]_t = 30$. As alluded to earlier, it is possible that the system has not reached a steady state after 10 time steps, however the above graphs suggest that the concentrations after 10 steps are already extremely close to, if not equal to, steady state values. The optimisation process could be re-executed after increasing the value of the parameter `span` in the function `m()` to simulate the system for a larger number of time steps.

Acceptance Ratio Replica Exchange ROptimus Run

Let us now examine how replica exchange ROptimus using 12 cores performs on this task. We will use 200 000 optimisation iterations and set `STATWINDOW` to 50, similarly to **Tutorials 2** and **3**.

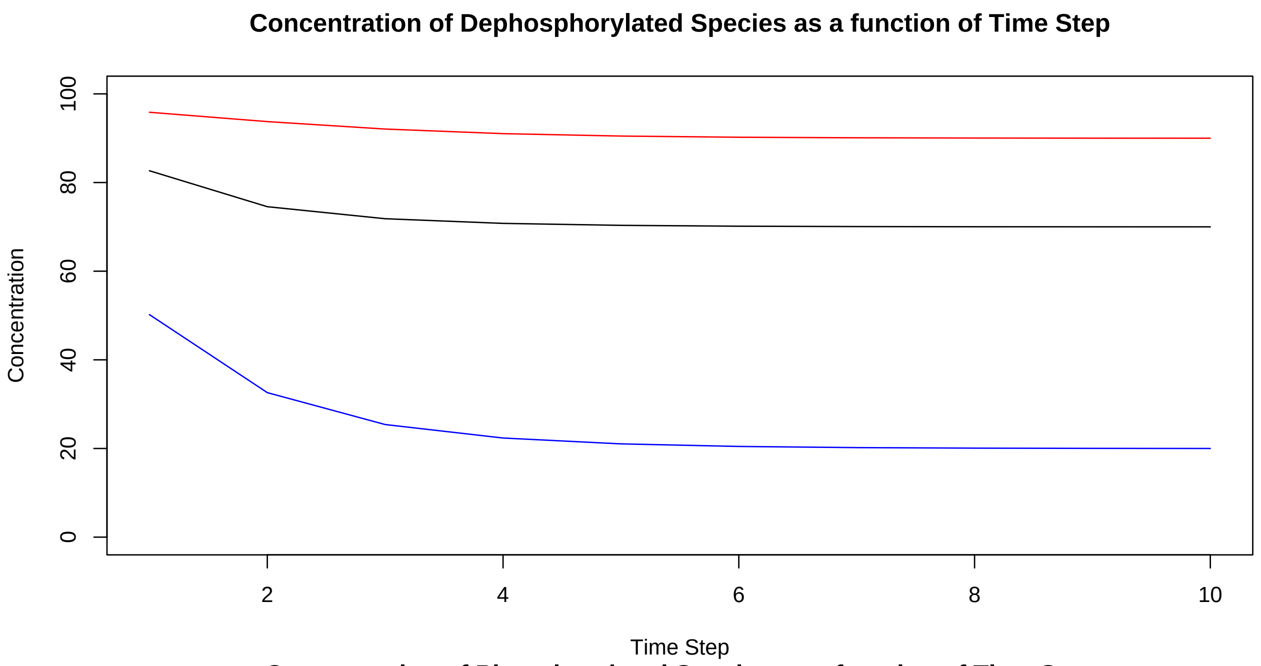
```
ACCRATIO <- c(90, 82, 74, 66, 58, 50, 42, 34, 26, 18, 10, 2)
```

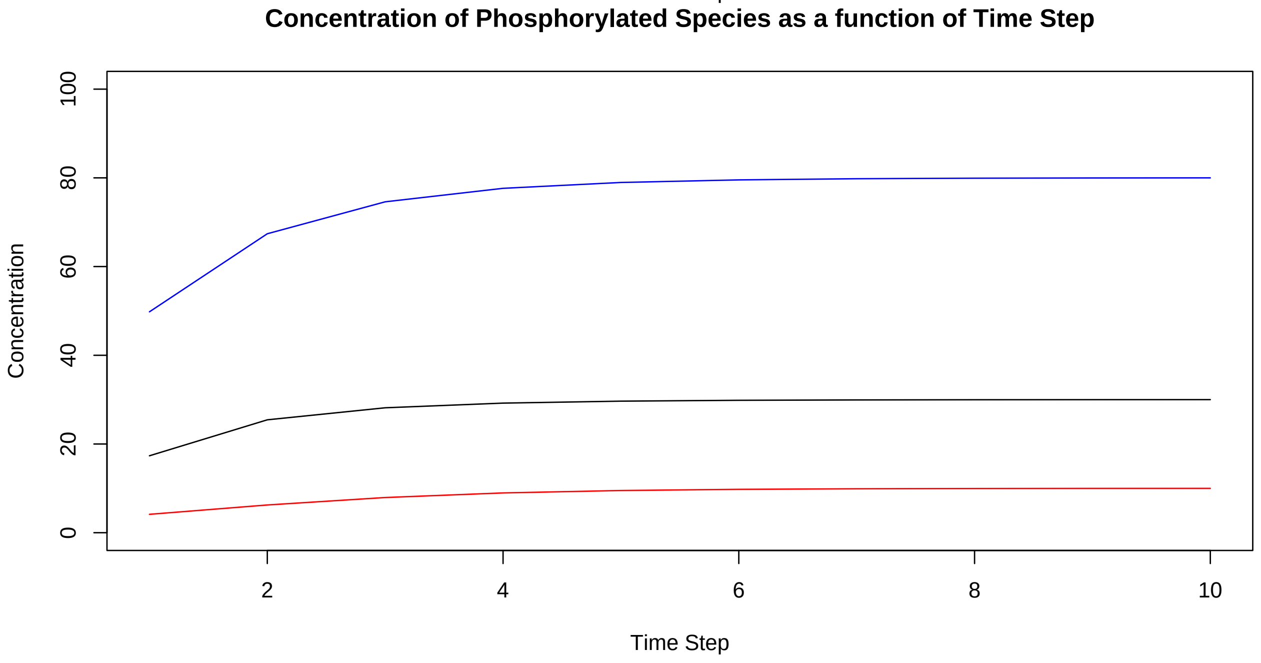
```
Optimus(NCPU=12, OPTNAME="DE_12_RE",
        NUMITER=200000, STATWINDOW=50, DUMP.FREQ=100000, LONG=FALSE,
        OPT.TYPE="RE", ACCRATIO=ACCRATIO,
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=DATA)
```

Table 15: 12-core Acceptance Ratio Replica Exchange results from ROptimus run.

	Replica Acceptance Ratio	E (RMSD)	K1	K2	K3	K4
CPU 1	90	0.0026225	0.9088	0.4090	0.0146	2.7246
CPU 2	82	0.0026343	0.8346	0.3754	0.0134	2.5012
CPU 3	74	0.0019724	0.7234	0.3252	0.0116	2.1644
CPU 4	66	0.0020945	0.9586	0.4312	0.0154	2.8748
CPU 5	58	0.0029243	0.9338	0.4200	0.0150	2.8002
CPU 6	50	0.0025811	0.8964	0.4034	0.0144	2.6872
CPU 7	42	0.0028685	0.8592	0.3866	0.0138	2.5752
CPU 8	34	0.0025121	0.8840	0.3978	0.0142	2.6500
CPU 9	26	0.0011265	0.9834	0.4424	0.0158	2.9492
CPU 10	18	0.0025811	0.8964	0.4034	0.0144	2.6872
CPU 11	10	0.0012516	0.7974	0.3586	0.0128	2.3886
CPU 12	2	0.0017556	0.7850	0.3532	0.0126	2.3512

Of the 12 optimisation replicas, CPU 9 (26% acceptance ratio) finds the best set of rate constants, $\{k_1 = 0.9834, k_2 = 0.4424, k_3 = 0.0158, k_4 = 2.9492\}$. This set of rate constants results in an RMSD (after crude 10 iteration limit) of 0.0011265, which is lower than the RMSD of the solution found by Acceptance Ratio Simulated Annealing ROptimus (0.0012516) done with the use of more modest computational resources. Let us simulate how the system evolves according to these rate constants for 10 time steps:





The table below summarises the initial and final concentrations of the various protein species when the system is simulated for 10 time steps using the rate constants $\{k_1 = 0.9834, k_2 = 0.4424, k_3 = 0.0158, k_4 = 2.9492\}$:

Table 16: System summary for k1 = 0.9834, k2 = 0.4424, k3 = 0.0158, k4 = 2.9492.

	[A]	[B]	[C]	[AP]	[BP]	[CP]
Initial	100.00000	100.00000	100.00000	0.00000	0.00000	0.00000
Final (after 10 time steps)	89.99807	19.99983	70.00022	10.00193	80.00017	29.99978

Here again, we see that the protein concentrations after 10 time steps are remarkably close to the target values $[A]_t = 90$, $[B]_t = 20$, $[C]_t = 70$, $[AP]_t = 10$, $[BP]_t = 80$ and $[CP]_t = 30$ and the graphs suggests that these concentrations have either converged or are very close to converging to the steady state.

Summary

We have seen how ROptimus can be employed to recover rate constants for a system of coupled ODEs that describes a biological pathway. Given an initial state of the system and a target state, both Acceptance Ratio Simulated Annealing (SA) ROptimus and Replica Exchange (RE) ROptimus found a set of rate constants that resulted in the desired system behaviour upon simulation of the system. In the current setup in this tutorial, RE slightly outperformed SA, both however are not directly comparable unless their allocated resources and times are equalised.

Table 17: Summary of protein concentrations after 10 time steps.

	[A]	[B]	[C]	[AP]	[BP]	[CP]
Target	90.00000	20.00000	70.00000	10.00000	80.00000	30.00000
ROptimus (AR Simulated Annealing)	89.99814	20.00081	69.99924	10.00186	79.99919	30.00076
ROptimus (AR Replica Exchange)	89.99807	19.99983	70.00022	10.00193	80.00017	29.99978

Table 18: Summary of recovered rate constants.

	E (RMSD)	K1	K2	K3	K4
ROptimus (AR Simulated Annealing)	0.0012516	0.7974	0.3586	0.0128	2.3886
ROptimus (AR Replica Exchange)	0.0011265	0.9834	0.4424	0.0158	2.9492

Tutorial 5: Constrained Shuffling of Genomic Contacts to Form a Control Set

Problem Statement

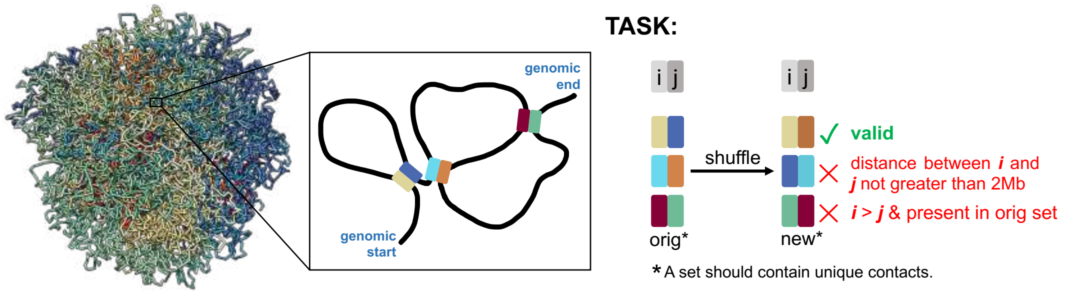
Here, we shall consider a problem from the field of 3D genome organisation, to exemplify how ROptimus can optimise a specific constrained shuffling task given a certain set of conditions.

Mammalian genomes are packaged into a complex three-dimensional (3D) structure inside the nucleus, bringing DNA regions linearly near or far from each other into contacts.

Here, we take a set of 734 pairs of i and j 40-kilobase DNA regions that are known to be in contact inside a cell. Binning the DNA (e.g., a single chromosome) into 40-kb regions, each region is represented as a single integer that is equal to its end position divided by the length of the region, which is 40 kb. For instance, the 1st region, with start and end positions at the 1^{st} and 40000^{th} nucleotides, respectively, is denoted as 1 (40000^{th} base /40000 bases = 1). This simplifies the notation for a contact between two regions to a pair of positive integers as shown below:

```
##     i   j
## 1  96 151
## 2  96 174
## 3 322 374
## 4 309 392
## 5 321 392
## 6 373 460
```

where $i \in Z^+$ and $j \in Z^+$.


genomic end
genomic start
TASK:
i j
i j
shuffle
✓ valid
distance between i and j not greater than 2Mb
i > j & present in orig set
orig*
new*
* A set should contain unique contacts.

Take note that our set of contacts comply with the following criteria.

1. The set only contains **unique, long-range contacts**, with i and j regions always separated by greater than 2 megabases (Mb). In terms of the integer identifiers, the threshold is 50 ($2Mb/40kb = 50$), such that $(j - i) > 50$.
2. i is set to be always less than j. This avoids duplicate contacts in the set definition, such as

```
##        i   j
## [1,] 181 273
## [2,] 273 181
```

3. A region can take part in more than one contact (as in region 96 in rows 1 and 2 in the table above), given that the contacts it forms comply with the two aforementioned criteria.

The task herein is to take the DNA regions from the original set (shown above) and to shuffle them to produce a new set of contacts. This is useful for a research on genomic 3D contacts requiring good quality of negative controls that still have the constraints that the original true data possess. The challenge here is to maximise the number of new control contacts that we can devise and that are valid, meaning that they A) still satisfy the aforementioned 3 criteria, and B) are not present in the original set.

Defining ROptimus Inputs

We can solve this highly constrained shuffling problem using ROptimus by doing the following preparations. Keeping the original order of i's in the loaded real set of contacts (referred hereafter as IJ_ORIG), we can randomly shuffle all j's in the set once, without replacement. The output will be the object K to be supplied to Optimus() for the optimisation.

```r
K <- IJ_ORIG
set.seed(840)
# Shuffle all j's once
K[,"j"] <- sample(x=K[,"j"], size=nrow(K), replace=FALSE)
```

The m() function should then operate on the object K, which, to reiterate, is derived from the original set of contacts with all j's shuffled once (see above). It can also accept a supplementary object DATA that holds data necessary for the model calculations and outcome comparison. Output of m() is the observable object O that is the percentage of contacts not satisfying the aforementioned criteria or are already in the original set. To get O, m() will require a) the original set of contacts (IJ_ORIG), b) number of contacts, and c) gap threshold set between contacting regions in terms of positive integer (50). These are first stored in the DATA object then passed to m().

```r
DATA <- list(IJ_ORIG=IJ_ORIG,
             gaplimit=50,
             numContacts=nrow(IJ_ORIG))

m <- function(K, DATA){
  # Total number of erroneous contacts in the new set
  errCont <- sum(
    # Contacts not satisfying the threshold value of the gap between their two regions
    ( (K[,"j"]-K[,"i"]) <= DATA$gaplimit ) |
    # Contacts also present in the original set
    ( K[,"j"]==DATA$IJ_ORIG[,"j"]        ) |
    # Duplicated contacts in the new set
    ( duplicated(K)                      )
  )
  # Percentage of erroneous contacts
  O <- (errCont/DATA$numContacts)*100
  return(O)
}
```

In the u() function, the error percentage O will be directly used as the pseudo energy E, since we do want to minimise this value. Meanwhile, the quality Q for the given snapshot of K, which we want to increase, can be the negative of O.

```r
u <- function(O, DATA){
  RESULT <- NULL
  RESULT$Q <- -O
  RESULT$E <- O
  return(RESULT)
}
```

Finally, r(), which defines how object K will be altered at every step, takes two j's and then swaps them, without changing the order of their partner i's.

```r
r <- function(K){
  K.new <- K
  # Indices of two randomly chosen j's to swap
  new.ind <- sample(x=1:length(K.new[,"j"]), size=2, replace=FALSE)
  # Swap the j's
  K.new[new.ind,"j"] <- K.new[rev(new.ind),"j"]
  return(K.new)
}
```

Acceptance Ratio Simulated Annealing ROptimus Run

We first use the Acceptance Ratio Simulated Annealing scheme in ROptimus to solve the problem. As in **Tutorials 2** and **4**, `Optimus()` is set to do 200 000 steps, 2 annealing cycles, and is to produce 4 independent replicas of the same simulation.

```r
Optimus(NCPU=4, OPTNAME="IJ.NEW.OPTI.SA",
        NUMITER=200000, CYCLES=2, DUMP.FREQ=100000, LONG=FALSE,
        OPT.TYPE="SA",
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=DATA)
```

Note that if additional data is required by `m()` or `u()`, that object should be assigned to `DATA` argument of `Optimus()`, otherwise `DATA` defaults to `NULL`.

The 4 new sets obtained as optimums from each of the 4 replicates of independent simulated annealing runs have the following error percentages:

Table 19: 4-core Acceptance Ratio Simulated Annealing results from ROptimus run.

Set	Erroneous Contact %
1	11.444
2	10.899
3	12.262
4	11.717

Acceptance Ratio Replica Exchange ROptimus Run

Now, we can use the Acceptance Ratio Replica Exchange mode of ROptimus using 12 cores and the same number of optimisation steps (200 000) as in the Acceptance Ratio Simulated Annealing ROptimus run. Similar to **Tutorials 2** to **4**, `STATWINDOW` is set to 50 and the acceptance ratios are as follows:

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

Optimus(NCPU=12, OPTNAME="IJ.NEW.OPTI.RE",
        NUMITER=200000, STATWINDOW=50, DUMP.FREQ=100000, LONG=FALSE,
        OPT.TYPE="RE", ACCRATIO=ACCRATIO,
        K.INITIAL=K, rDEF=r, mDEF=m, uDEF=u, DATA=DATA)
```

Set	Replica Acceptance Ratio	Erroneous Contact %

Table 20: 12-core Acceptance Ratio Replica Exchange results from ROptimus run.

Set	Replica Acceptance Ratio	Erroneous Contact %
1	90	34.060
2	82	29.155
3	74	24.523
4	66	18.937
5	58	18.937
6	50	17.847
7	42	18.256
8	34	18.392
9	26	18.529
10	18	18.529
11	10	18.529
12	2	18.529

Notably, in this setup, none of the 12 new sets of contacts from the replica exchange run are better than the sets produced by the Acceptance Ratio Simulated Annealing mode of ROptimus.

Summary

We have added yet another one to the diverse applications of ROptimus, this time by performing a constrained shuffling of pairs of positive integers that, in this case, represent contacts between DNA regions, to form a new set of control contacts. Additionally, ROptimus showed to be a platform, where certain restrictions or conditions can easily be incorporated to a given shuffling task, which usually is required for specific research objectives. The table below shows the % of erroneous contacts of the best outputs from the acceptance ratio annealing and replica exchange ROptimus runs, with the former producing the better set.

Table 21: Summary of solutions.

Method	Erroneous Contact %
ROptimus (AR Simulated Annealing)	10.899
ROptimus (AR Replica Exchange)	17.847

Advanced User Manual

This section will document all possible non-experimental input arguments to `Optimus()` and will describe the output format of ROptimus. The code snippet below is the first part of the definition of `Optimus()`, which lists all non-experimental input arguments. We will refer to input arguments without default values as mandatory input arguments and those with default values as optional (with the exception of `K.INITIAL`, which will be considered a mandatory input argument).

```
Optimus <- function(NUMITER       = 1000000,
                    STATWINDOW    = 70,
                    T.INI         = 0.00001,
                    T.ADJSTEP     = 0.000000005,
                    TSCLnum       = 2,
                    T.SCALING     = 3,
                    T.MIN         = 0.000000005,
                    T.DELTA       = 2,
                    DUMP.FREQ     = 10000,
                    LIVEPLOT      = TRUE,
                    LIVEPLOT.FREQ = 100000,
                    PDFheight     = 29,
                    PDFwidth      = 20,
                    NCPU          = 4,
                    LONG          = TRUE,
                    SEED          = 840,
                    OPTNAME       = "",
                    DATA          = NULL,
                    K.INITIAL     = 0,
                    rDEF,
                    mDEF,
                    uDEF,
                    EXCHANGE.FREQ = 1000,
                    ACCRATIO      = c(90, 50, 5, 1),
                    CYCLES        = 10,
                    ACCRATIO.IN   = 90,
                    ACCRATIO.FIN  = 0.5,
                    OPT.TYPE = "SA"
                    ){...}
```

Mandatory Input Arguments

All mandatory input arguments have necessarily already been defined in the tutorials. However, we will reiterate these definitions in this section. `Optimus()` has four mandatory inputs, `mDEF`, `uDEF`, `rDEF` (referred to as `m()`, `u()` and `r()` respectively in the tutorials) and `K.INITIAL`.

mDEF

`mDEF` must be of type closure. `mDEF` should be a function 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. `mDEF` must necessarily take `K` and `DATA` as input variables, and it must necessarily operate on `K` to produce `O` (operating on `DATA` in optional, see **Tutorial 3** for an illustration).

uDEF

`uDEF` must be of type closure. `uDEF` should be a function designed to evaluate the performance of a given snapshot of coefficients `K`. `uDEF` should necessarily take as inputs `O` (the output of `mDEF`) and the variable `DATA`. The output of `uDEF` should have two components, `Q` holding a single number of the quality of the `K` coefficients (also used for plotting), 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`, never vice versa. The `Q` component of the `uDEF` function output is only used for plotting the optimisation process, and, if desired, can just repeat the value of the `E` component.

rDEF

`rDEF` must be of type closure. `rDEF` should be a function that defines a rule by which the `K` coefficient vector is to be altered from one step to another. `rDEF` must accept `K` as an input and return an object equivalent to `K`, but with some alteration(s).

K.INITIAL

`K.INITIAL` is an object of any type, which stores the initial values for the parameter(s) to be optimised. The only requirement for `K` is that it should be something alterable *via* `rDEF` and something that influences the outcome of `mDEF`.

Optional Input Arguments

NUMITER

`NUMITER` is a variable of type double that is the number of steps of the optimisation process per processing core. It has a default value of 1 000 000.

STATWINDOW

`STATWINDOW` is a variable of type double that is the number of steps executed between subsequent temperature adjustments executed by the Temperature Control Unit. `STATWINDOW` is also the number of steps used to calculate the observed acceptance ratio). It has a default value of 70.

T.INI

`T.INI` is a variable of type double that represents the initial system pseudo temperature at the beginning of the optimisation procedure. It has a default value of 0.00001.

T.ADJSTEP

`T.ADJSTEP` is a variable of type double that represents the baseline temperature change step-size for temperature auto-adjustment. It has a default value of 0.000000005.

TSCLnum

`TSCLnum` is a variable of type double that indicates the maximum number of `STATWINDOWS` for which the observed acceptance ratio can sequentially be below or above the ideal acceptance ratio before `T.ADJSTEP` is scaled by `T.SCALING`. It has a default value of 2.

T.SCALING

`T.SCALING` is a variable of type double that represents the value by which `T.ADJSTEP` is scaled in accordance with the condition specified by `TSCLnum`. It has a default value of 3.

T.MIN

`T.MIN` is a variable of type double and is the value that the system pseudo temperature is automatically set to if at any time, the Temperature Control Unit attempts to make the pseudo temperature a negative value. It has a default value of 0.000000005.

T.DELTA

`T.DELTA` is a variable of type double. If after a `STATWINDOW`, the observed acceptance ratio is within `T.DELTA` of the ideal acceptance ratio, the Temperature Control Unit will make no change to the system pseudo temperature. It has a default value of 2.

DUMP.FREQ

`DUMP.FREQ` is a variable of type double. It is the frequency in steps of printing the best found model to the working directory. It has a default value of 10 000.

LIVEPLOT

`LIVEPLOT` is a variable of type logical that indicates whether the optimisation process will be plotted in a PDF file in the working directory. It has a default value of `TRUE`.

LIVEPLOT.FREQ

`LIVEPLOT.FREQ` is a variable of type double that indicates the frequency in steps of printing the optimisation process in a PDF (this variable is only relevant if `LIVEPLOT = TRUE`). It has a default value of 100 000.

PDFheight

`PDFheight` is a variable of type double that indicates the height of the PDF that is produced (if `LIVEPLOT = TRUE`). It has a default value of 29.

PDFwidth

`PDFwidth` is a variable of type double that indicates the width of the PDF that is produced (if `LIVEPLOT = TRUE`). It has a default value of 20.

NCPU

`NCPU` is a variable of type double that indicates the number of optimisation replicas to execute. If calling the Acceptance Ratio Replica Exchange mode of ROptimus, `NCPU` must be greater than 1. `NCPU` has a default value of 4.

LONG

`LONG` is a variable of type logical. If `LONG = TRUE`, a memory-friendly version of ROptimus will be activated (in anticipation of a long simulation), and only data from the optimal explored parameter configuration and the last 10 000 optimisation iterations will be stored. `LONG` has a default value of TRUE.

SEED

`SEED` is a variable of type double which sets the seed for the random number generator. It has a default value of 840.

OPTNAME

`OPTNAME` is a variable of type character that can be thought of as the name of the optimisation process. `OPTNAME` is used when creating the file names of the ROptimus output. `OPTNAME = ""` is the default value.

DATA

`DATA` is a variable of type list holding any additional data that must be accessed by `mDEF` and `uDEF`. The default value for `DATA` is `NULL`.

OPT.TYPE

`OPT.TYPE` is a variable of type character, which specifies the mode of ROptimus to execute and should always be equal to “SA” or “RE.” If equal to “SA” (for Simulated Annealing), the Acceptance Ratio Simulated Annealing version of ROptimus will be executed. If equal to “RE” (for Replica Exchange), the Acceptance Ratio Replica Exchange version of ROptimus will be executed. The default value of `OPT.TYPE` is “SA.”

DIR

`DIR` is a variable of type character, which specifies the directory to save the ROptimus outputs. The default value for `DIR` is “.” that means the current directory while running ROptimus.

starcore

`starcore` is an experimental variable of type list, holding some parameters for in-lab starcore use only. For example, `starcore` can be `c("MAXCOEFORDER"=4)`, which will specify the maximum coefficient order. The default value for `starcore` is `NULL`, i.e. not activated.

Optional Inputs Specific to Acceptance Ratio Simulated Annealing

The below optional input arguments only impact the Acceptance Ratio Simulated Annealing mode of ROptimus.

CYCLES

`CYCLES` is a variable of type double that specifies the number of annealing cycles to execute per core during the optimisation process. It has a default value of 10.

ACCRATIO.IN

`ACCRATIO.IN` is a variable of type double that specifies the initial target acceptance ratio for the annealing schedule in each annealing cycle. It has a default value of 90.

ACCRATIO.FIN

`ACCRATIO.FIN` is a variable of type double that specifies the final target acceptance ratio for the annealing schedule in each annealing cycle. It has a default value of 0.5.

Optional Inputs Specific to Acceptance Ratio Replica Exchange Optimisation

The below optional input arguments only impact the Acceptance Ratio Replica Exchange mode of ROptimus.

EXCHANGE.FREQ

`EXCHANGE.FREQ` is a variable of type double that specifies the total number of exchanges between adjacent replicas that will occur during the optimisation process. It has a default value of 1000.

ACCRATIO

`ACCRATIO` is a vector of doubles that must have length equal to the value of `NCPU`. `ACCRATIO` specifies the target acceptance ratio for each optimisation replica. It has a default value of `c(90, 50, 5, 1)`.

ROptimus Output

ROptimus creates multiple output files in a user's working directory during an optimisation run. Each core used will generate 4 or 5 output files (depending on the value of `LIVEPLOT`):

1) [OPTNAME][Core number]_model_ALL
2) [OPTNAME][Core number]_model_K
3) [OPTNAME][Core number]_model_O
4) [OPTNAME][Core number]_model_QE
5) [OPTNAME][Core number]

Note that the 5th file is only produced if `LIVEPLOT=TRUE`. As an illustration, for the Acceptance Ratio Simulated Annealing run from **Tutorial 1**, ROptimus generates 20 output files with the following names: poly_4_SA1_model_ALL, poly_4_SA1_model_K, poly_4_SA1_model_O, poly_4_SA1_model_QE, poly_4_SA1, poly_4_SA2_model_ALL, poly_4_SA2_model_K, poly_4_SA2_model_O, poly_4_SA2_model_QE, poly_4_SA2, poly_4_SA3_model_ALL, poly_4_SA3_model_K, poly_4_SA3_model_O, poly_4_SA3_model_QE, poly_4_SA3,poly_4_SA4_model_ALL, poly_4_SA4_model_K, poly_4_SA4_model_O, poly_4_SA4_model_QE and poly_4_SA4.

[OPTNAME][Core number]__model__ALL

This is the most important output file in that essentially all contents from the other output files is contained in this file. The *__model__ALL file is an R workspace that contains a variable `OUTPUT` of type list. For Acceptance Ratio Simulated Annealing, `OUTPUT` has 12 fields (numbered 1-12 below), while for Acceptance Ratio Replica Exchange, `OUTPUT` contains an additional 14 fields (numbered 13-26 below). Note that the additional fields of `OUTPUT` in Replica Exchange were included to facilitate the writing of the code; they can largely be ignored by the user.

1) `K.stored`
2) `O.stored`
3) `STEP`
4) `PROB.VEC`
5) `T.DE.FACTO`
6) `IDEAL.ACC.VEC`
7) `ACC.VEC.DE.FACTO`
8) `STEP4ACC.VEC.DE.FACTO`
9) `ENERGY.DE.FACTO`
10) `Q.STRG`
11) `ACCEPTANCE`
12) `STEP.STORED`
13) `E.stored`
14) `E.old`
15) `Q.old`
16) `K`
17) `T`
18) `Step.stored`
19) `ENERGY.TRIAL.VEC`
20) `STEP.add`
21) `NumofAccRatSMIdeal`
22) `NumofAccRatGRIdeal`
23) `t.adjstep`
24) `AccR.category`
25) `new.T.INI`
26) `instanceOFswitch`

`K.stored` holds the optimal parameter configuration found by the given processor. `O.stored` holds the object `O` generated by `mDEF` from the optimal parameter configuration `K.stored`. `STEP` is a double that holds the current optimisation iteration number. `PROB.VEC` is a vector that holds the acceptance probability for each optimisation step. `T.DE.FACTO` is a vector that holds the system pseudo temperature during each optimisation iteration. `IDEAL.ACC.VEC` is a vector that holds the target acceptance ratio for each optimisation iteration in the case of Acceptance Ratio Simulated Annealing. In the case of Acceptance Ratio Replica Exchange, `IDEAL.ACC.VEC` holds the same value as the input variable `ACCRATIO`. `ACC.VEC.DE.FACTO` is a vector that holds the observed acceptance ratio at the end of each `STATWINDOW`. `STEP4ACC.VEC.DE.FACTO` is a vector that holds the optimisation step numbers that correspond to the end of a `STATWINDOW`. `ENERGY.DE.FACTO` is a vector that holds the actual system energy E at each optimisation step. `Q.STRG` is a vector that holds the system quality Q at each optimisation step. `ACCEPTANCE` is a vector of binary variables whose i^{th} entry is 1 if the candidate parameter configuration was accepted on the i^{th} optimisation step and 0 otherwise. `STEP.STORED` is a vector storing the number of each optimisation step.

`E.stored` is a double that stores the energy E associated with the optimal parameter configuration `K.stored`. `E.old` is a double that stores the energy E associated with the most recently considered parameter configuration. `Q.old` is a double that stores the quality Q associated with the most recently considered parameter configuration. `K` holds the most recently considered parameter configuration. `T` is a double that holds the current system temperature. `Step.stored` is a double that holds the optimisation step number on which the optimal parameter configuration `K.stored` was discovered. `ENERGY.TRIAL.VEC` is a vector that holds the energy E of the candidate parameter configuration at every optimisation step. `STEP.add` is a double that facilitates indexing into output vectors. `NumofAccRatSMIdeal` is a double that represents the number of times the observed acceptance ratio has sequentially been smaller than the target acceptance ratio. `NumofAccRatGRIdeal` is a double that represents the number of times the observed acceptance ratio has been sequentially greater than the target acceptance ratio. `t.adjstep` is a double holding the current value by which the system pseudo temperature is increased or decreased each `STATWINDOW` by the Temperature Control Unit. `AccR.category` is a character that indicates whether the observed acceptance ratio was above or below the target acceptance ratio during the previous `STATWINDOW`. `new.T.INI` is a double that stores an estimate for the ideal initial system pseudo temperature deduced by the Temperature Control Unit. Finally, `instanceOFswitch` is a double that tracks the number of times the observed acceptance ratio has transitioned from being less than the target ratio to greater than the target ratio or vice versa.

[OPTNAME][Core number]_model_K

The *_model_K file is an R workspace that contains a variable `K.stored` of the same type as `K.INITIAL`. It holds the optimal parameter configuration found by the given processor.

[OPTNAME][Core number]_model_O

The *_model_O file is an R workspace that contains a variable `O.stored` that holds the object `O` generated by `mDEF` from the optimal parameter configuration `K.stored`.

[OPTNAME][Core number]_model_QE

The *_model_QE file is a text file that stores the values of E and Q that are produced from the optimal parameter configuration `K.stored` and, in the case of the Acceptance Ratio Replica Exchange mode of ROptimus, stores the target acceptance ratio associated with the given replica.

[OPTNAME][Core number]

The * file is a pdf file that includes 5 plots:

1) acceptance probability as a function of optimisation step;
2) system psuedo temperature as a function of optimisation step;
3) observed (red solid line) and target (black dashed line) acceptance ratio as a function of optimisation step;
4) energy E as a function of optimisation step;
5) quality Q as a function of optimisation step.

References

Ballard, Andrew, and Christopher Jarzynski. 2009. "Replica Exchange with Nonequilibrium Switches." *Proceedings of the National Academy of Sciences of the United States of America* 106 (July). https://doi.org/10.1073/pnas.0900406106.

Chen, Yunjie, and Benoit Roux. 2015. "Generalized Metropolis Acceptance Criterion for Hybrid Non-Equilibrium Molecular Dynamics - Monte Carlo Simulations." *The Journal of Chemical Physics* 142 (January). https://doi.org/10.1063/1.4904889.

Chib, Siddhartha, and Edward Greenberg. 1995. "Understanding the Metropolis-Hastings Algorithm." *The American Statistician* 49 (November): 327–35. https://doi.org/10.2307/2684568.

Cowles, Mary, and Bradley Carlin. 1996. "Markov Chain Monte Carlo Convergence Diagnostics: A Comparative Review." *Journal of the American Statistical Association* 91 (June): 883–904. https://doi.org/10.1080/01621459.1996.10476956.

Gilks, W.R., S. Richardson, and D.J. Spiegelhalter. 1996. *Markov Chain Monte Carlo in Practice. Chapman & Hall.*

Gilli, Manfred, Dietmar Maringer, and Enrico Schumann. 2019. "Numerical Methods and Optimization in Finance 2nd Edition." *Elsevier/Academic Press.*

Hahsler, Michael. 2017. "Qap: Heuristics for the Quadratic Assignment Problem (QAP). R Package Version 0.1-1." https://CRAN.R-project.org/package=qap.

Husmann, Kai, Alexander Lange, and Elmar Spiegel. 2017. "The R Package Optimization: Flexible Global Optimization with Simulated-Annealing." https://github.com/kaihusmann/optimization.

Ingber, Lester. 1993. "Simulated Annealing: Practice Versus Theory." *Mathematical and Computer Modelling* 18 (December): 29–57. https://doi.org/10.1016/0895-7177(93)90204-C.

King, Aaron A., Edward L. Ionides, Carles Martinez Breto, Stephen P. Ellner, Matthew J. Ferrari, Sebastian Funk, Steven G. Johnson, et al. 2022. "Pomp: Statistical Inference for Partially Observed Markov Processes. R Package Version 4.1." https://kingaa.github.io/pomp/.

King, Aaron A., Dao Nguyen, and Edward L. Ionides. 2016. "Statistical Inference for Partially Observed Markov Processes via the R Package Pomp." *Journal of Statistical Software.* https://doi.org/10.18637/jss.v069.i12.

Kirkpatrick, Scott. 1984. "Optimization by Simulated Annealing: Quantitative Studies." *Journal of Statistical Physics* 34: 975–86. https://doi.org/10.1007/BF01009452.

Klipp, Edda, Wolfram Liebermeister, Cristoph Wierling, Axel Kowald, Hans Lehrach, and Ralf Herwig. 2009. *Systems Biology. Wiley-VCH.*

Murphy, Lora. 2022. "Likelihood: Methods for Maximum Likelihood Estimation. R Package Version 1.8." https://CRAN.R-project.org/package=likelihood.

Orestes Cerdeira, Jorge, Pedro Duarte Silva, Jorge Cadima, and Manuel Minhoto. 2020. "Subselect: Selecting Variable Subsets. R Package Version 0.15.2." https://CRAN.R-project.org/package=subselect.

R Core Team. 2021. "R: A Language and Environment for Statistical Computing." *R Foundation for Statistical Computing, Vienna, Austria.* https://www.R-project.org/.

Raue, Andreas, Marcel Schilling, Julie Bachmann, Andrew Matteson, Max Schelke, Daniel Kaschek, Sabine Hug, et al. 2013. "Lessons Learned from Quantitative Dynamical Modeling in Systems Biology." *PLOS ONE*, September. https://doi.org/10.1371/journal.pone.0074335.

Schumann, Enrico. 2022. "Numerical Methods and Optimization in Finance (NMOF). R Package Version 2.5-1." https://github.com/enricoschumann/NMOF.

Schwendinger, Florian, and Hans W. Borchers. 2022. "CRAN Task View: Optimization and Mathematical Programming. Version 2022-04-12." https://CRAN.R-project.org/view=Optimization.

Solymos, Peter. 2010. "Dclone: Data Cloning in R." *The R Journal*. https://datacloning.org/.

Sugita, Yuji, and Yuko Okamoto. 1999. "Replica-Exchange Molecular Dynamics Method for Protein Folding." *Chemical Physics Letters* 314 (November): 141–51. https://doi.org/10.1016/S0009-2614(99)01123-9.

Xiang, Yang, Sylvain Gubian, Brian Suomela, and Julia Hoeng. 2013. "Generalized Simulated Annealing for Efficient Global Optimization: The GenSA Package for R." *The R Journal*. https://journal.r-project.org/archive/2013/RJ-2013-002/index.html.