# Sensitivity analyses: a brief tutorial with R package pse

Chalom, A. [*†] Mandai, C.Y. [*] Prado, P.I. [*]

Version 0.3.1, November 23, 2013

This document presents a brief practical tutorial about the use of sensitivity analyses tools in the study of ecological models. To read about the underlying theory, please refer to our work in [1].

We presume for this tutorial that you are already familiar with the R programing environment and with your chosen model. We will illustrate the techniques using a simple model of population growth.

You should have installed R [1] along with an interface and text editor of your liking, and the package "pse". This package is based on the "sensitivity" package, and is designed to resemble its uses, so researchers who already use it will be able to write code with the pse package easily. Major differences will be noted on the help pages and in this tutorial.

This tutorial focuses on the parameter space exploration of deterministic models. For a discussion of stochastic models, see the 'multiple' vignette on the same package.

## 1 Input parameters

The first thing that must be done is to determine exactly what are the input parameters to your model. You should list which parameters should be investigated, what are the probability density functions (PDFs) from which the parameter values will be calculated, and what are the arguments to these PDFs.

In the examples, we will use a simple model of logistical population growth, in which a population has an intrinsic rate of growth $r$, a carrying capacity of $K$ and a starting population of $X_0$. In each time step, the population may grow or diminish after the following expression:

$$X_{t+1} = X_t + rX_t\,(1 - X_t/K) \tag{1}$$

[*]Theoretical Ecology Lab, LAGE at Dep. Ecologia, Instituto de Biociências, Universidade de São Paulo, Rua do Matão travessa 14 nº 321, São Paulo, SP, CEP 05508-900, Brazil.

[†]email: andrechalom@gmail.com

[1]This tutorial was written and tested with R version 3.0.1, but it should work with newer versions

We are interested in studying the effects of the parameters $r$, $K$ and $X_0$ on the final population. After researching on our databases, we have decided that, for our species of interest, $r$ and $K$ follow a normal distribution with known parameters. However, we could not reliably determine what the initial population should be, so we have used an uniform distribution covering all the reasonable values. The following table summarizes this:

| Parameter | Distribution | Arguments |
|---|---|---|
| $r$ | normal | $\mu = 1.7$, $\sigma = 0.3$ |
| $K$ | normal | $\mu = 40$, $\sigma = 1$ |
| $X_0$ | uniform | min = 1, max = 50 |

We next translate this table to three R objects that will be used in the sensitivity analyses, containing (1) the names of the parameters, (2) the probability density functions, and (3) *a list containing the lists* with all the parameters to the density functions:

```
> factors <- c("r", "K", "X0")
> q <- c("qnorm", "qnorm", "qunif")
> q.arg <- list( list(mean=1.7, sd=0.3), list(mean=40, sd=1),
+         list(min=1, max=50) )
```

A fundamental question in this stage is to determine whether, inside the ascribed parameter ranges, *every parameter combination* is meaningful. See the next examples on this:

**Example 1:**

We would like to run a model for a species abundance distribution (SAD), and we decided to examine the effect of $N$, the total number of individuals in the community, and $S$, the total number of species. We can run the model with $N = 100$ and $S = 50$ or with $N = 15$ and $S = 3$, so there is nothing wrong with these values. However, the *combination* $N = 15$, $S = 50$ is meaningless, as it would imply that there are more species than individuals. One solution to this problem is to run the models with the parameters modified as following: $N$ is the total number of individuals, and $\hat{s}$ is the average number of individuals for each species. So, $\hat{s} * N = S$, and now every combination of $N$ and $\hat{s}$ is meaningful.

**Example 2:**

In a model of structured population growth, we have estimated independently two parameters for each class: $S$, the probability that a given individual survives and does not move into the next size class, and $G$, the probability that a given individual survives and grows into the next class. We can run the model with $S = 0.2$ and $G = 0.7$, or $S = 0.8$ and $G = 0.1$. However, if we try to run the model with $S = 0.8$ and $G = 0.7$, we arrive at the conclusion that, for every individual in the original size class, in the next time step we will have 0.8 individuals in the same class and more 0.7 in the next, giving a total of 1.5 individuals! The problem is that the sum of $S$ and $G$ must be smaller than 1. One way to solve this is to define new parameters $\hat{s}$ and $\hat{g}$ such that $\hat{s}$ is the survival probability, independently of the individual growing, and $\hat{g}$ is the growth probability for each surviving individual. We can relate these parameters to the former ones, as $G = \hat{s} * \hat{g}$ and $S = \hat{s} * (1 - \hat{g})$.

**Note:**

When transforming parameters like done on the above examples, it is important to remember that the new parameters may not have the same probability density functions as the original ones.

## 1.1 Optional: More details about the quantiles

The quantile functions used can be any of the built-in quantile functions as **qnorm** for normal, **qbinom** for binomial, **qpois** for poison, **qunif** for uniform, etc; less common distributions can be found on other packages, like the truncated normal distribution on package “msm”. You can even define other quantile functions, given that their first argument is the probability, and that they are able to work on a vector of probabilities. For example:

The quantiles of an empirical data set can be used by creating a wrapper function for the **quantile** function:

```
> qdata <- function(p, data) quantile(x=data, probs=p)
```

A discrete uniform density function, usefull for parameters that must be integer numbers, can be given by

```
> qdunif<-function(p, min, max) floor(qunif(p, min, max))
```

## 2 Your model

The model that you wish to analyse must be formulated as an R function that receives a *data.frame*, in which every column represent a different parameter, and every line represents a different combination of values for those parameters. The function must return an array with the same number of elements as there were lines in the original data frame, and each entry in the array should correspond to the result of running the model with the corresponding parameter combination. We will cover the case in which a model outputs more than a single number in section 4.

If your model is already written in R, and accepts a single combination of values, it is easy to write a “wrapper” using the function **mapply** to your model. In the example below, the function **oneRun** receives three numbers, corresponding to $r$, $K$ and $X_0$, and returns a single value corresponding to the final population. The function **modelRun** encapsulates this function, in a manner to receive a data.frame containing all parameter combinations and returning the results in one array.

Make **SURE** that the order in which the parameters are defined above is the same in which they are being passed to the function.

```
> oneRun <- function (r, K, Xo) {
+     X <- Xo
+     for (i in 0:20) {
+        X <- X+r*X*(1-X/K)
+     }
+     return (X)
+ }
> modelRun <- function (my.data) {
+         return(mapply(oneRun, my.data[,1], my.data[,2], my.data[,3]))
+ }
```

If your model is written in a different language, as C or Fortran, it is possible to write an interface with R by compiling your model as a shared library, and dynamically loading this library [2]. Also, you should consider uncoupling the simulation and the analyses (see section 5).

## 3 Uncertainty and sensibility analyses

We first use the **LHS** function to generate a hypercube for your model. The mandatory arguments for this function are: *model*, the function that represents your model; *factors*, an array with the parameter names; *N*, the number of parameter combinations to be generated; *q*, the names of the PDF functions to generate the parameter values; and *q.arg*, a list with the arguments of each pdf. We have already constructed suitable objects to pass to this function above, so now we simply call the **LHS** function:

```
> library(pse)
> myLHS <- LHS(modelRun, factors, 200, q, q.arg, nboot=50)
```

The extra parameter *nboot* is used to bootstrap the correlation coefficients (see below).

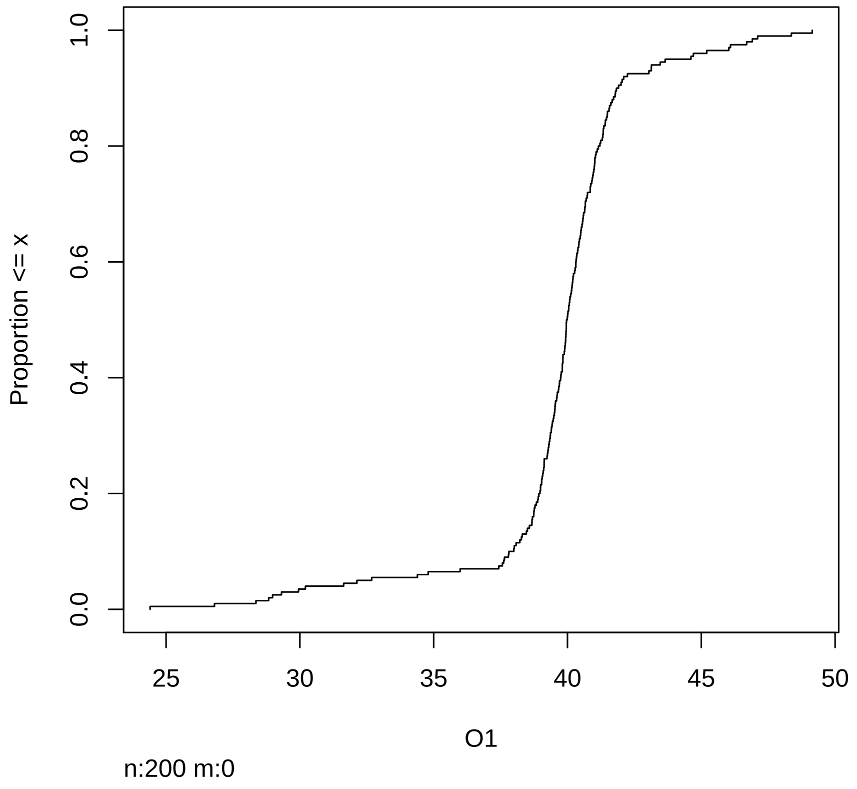
To access the values of the parameters used in the model, use the function **get.data(myLHS)**. To access the results, use **get.results(myLHS)**.

With the object returned by the function **LHS**, we will exemplify in this section four techniques that can be used: the uncertainty analysis using the **ecdf**, scatterplots of the correlation between each parameter and the result using the function **plotscatter**, partial rank correlation using the function **plotprcc** and agreement between different hypercube sizes with the function **sbma**.

## 3.1 ECDF

The ecdf, short for empirical cumulative distribution function, may be used to illustrate the distribution of the model results, in our case the final population. With this graph, we can see that the final population for our species is, with high probability, between 35 and 45.

```
> plotecdf(myLHS)
```

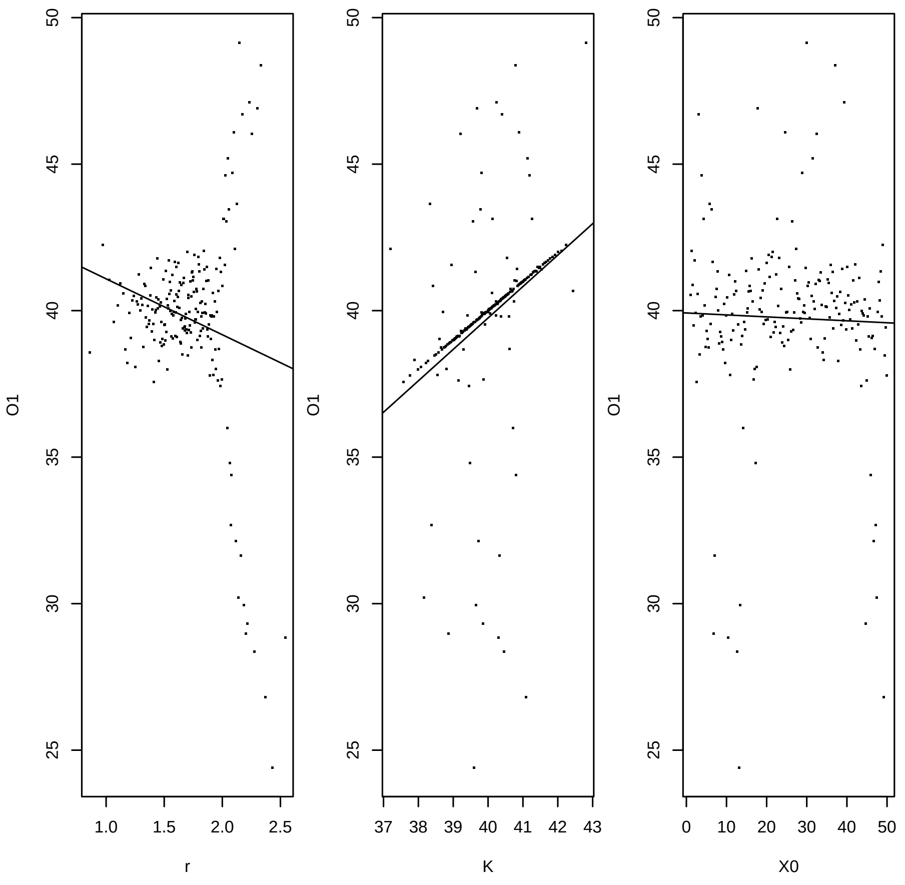


## 3.2 Scatterplots

Here, we can see a scatterplot of the result as a function of each parameter. As all the parameters are being changed for each run of the model, the scatterplots look like they were randomly generated, even if the underlying model is deterministic. Actually, what scatterplots show is the distribution of values returned by the model in the parameter space sampled by the hypercube and how sensible are these model responses to the variation of each parameter.

Note that population sizes bifurcate above a given value of parameter $r$. This is a well known behaviour of many population models and will ultimately lead to chaotic solutions [4, 3].

```
> plotscatter(myLHS)
```

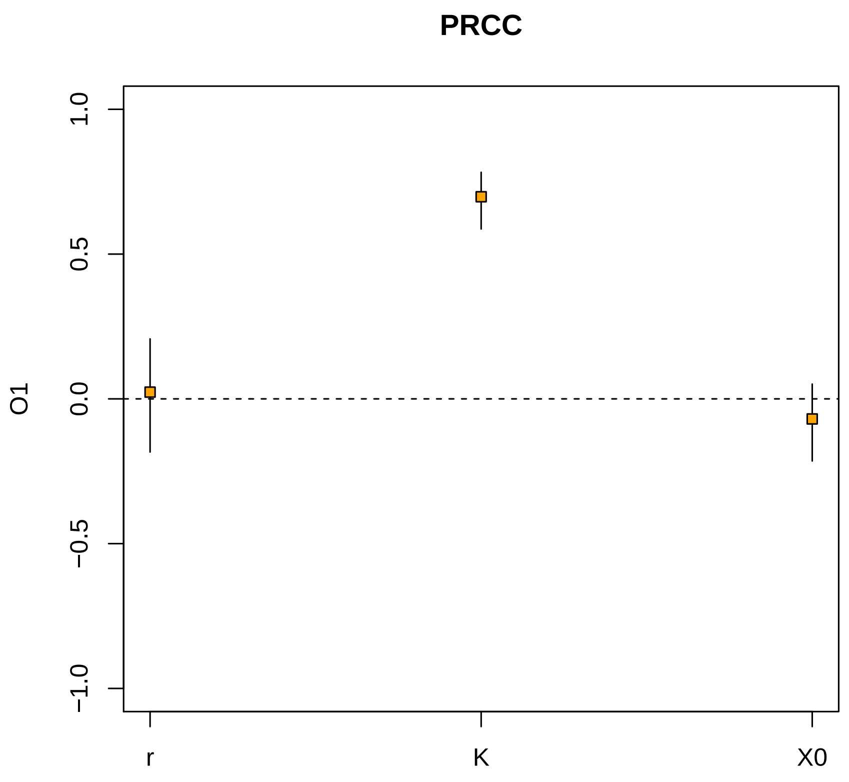


## 3.3 Partial correlation

The partial (rank) correlation coefficient (pcc or prcc) measures how strong are the linear associations between the result and each input parameter, after removing the linear effect of the other parameters.

The confidence intervals shown in this plot are generated by bootstraping.

```
> plotprcc(myLHS)
```



In the ecological literature, it is usual to refer to the partial derivatives of the model response in respect to each parameter as "the sensitivity" of the model response in respect to each parameter. One analog measure in stochastic models is the Partial Inclination Coefficient (pic) of the model response in respect to each parameter.

```
> pic(myLHS, nboot=40)

[[1]]

Call:
pic.default(X = L, y = r, nboot = nboot, conf = conf)

Partial Inclination Coefficients (PIC):
      original        bias std. error  min. c.i. max. c.i.
```

```
r  -1.90799288 0.128319193  1.6951612 -6.3230128 1.5810074
K   1.07256615 0.034158670  0.2028570  0.6415686 1.5568498
X0 -0.00668281 0.003230236  0.0172535 -0.0393534 0.0234695
```

## 3.4 Agreement between runs

In order to decide whether our sample size was adequate or insufficient, we calculate the Symmetric Blest Measure of Agreement (SBMA) between the PRCC coeffients of two runs with different sample sizes.

```
> newLHS <- LHS(modelRun, factors, 250, q, q.arg)
> (mySbma <- sbma(myLHS, newLHS))
```

```
[1] 0.375
```

A value of -1 indicates complete disagreement between the runs, and a value of 1 indicates total agreement. As the SBMA seldom reaches 1 for realistic models, some criterion must be used to indicate when the agreement should be considered good enough. More details about how the SBMA is calculated can be found on [1].

It should be stressed that there is no "magical" number for deciding how close to unity the SBMA should be. It is reasonable to expect agreements around 0.7 to 0.9 in well-behaved models, but two cases require attention. If the total number of factors is very low, the SBMA may converge slowly. Also, if none of the model parameters happen to be monotonically correlated with the output, the agreement between runs may stay as low as 0.2 even for very large hypercubes.

## 4 Multiple response variables

In the previous section, we have examined a model that returned a single number, namely, the final population. However, we might be interested in examining the effects of the parameters in several distinct responses from the model. The responses may be (1) different variables, like "total population" and "species richness", (2) the same variable in different time points, or (3) the same variable calculated by different methods.

In our example, we are interested in determining the effect of the parameters to the population in each of the first 6 time steps. The theory and tools for this analysis remain mostly the same. We will write our model to return an array now, as:

```
> factors <- c("r", "K", "X0")
> q <- c("qnorm", "qnorm", "qunif")
> q.arg <- list( list(mean=1.7, sd=0.3), list(mean=40, sd=1),
+         list(min=1, max=50) )
> Time <- 6
> oneRun <- function (r, K, Xo) {
+         X <- array();
+         X[1] <- Xo; # Caution, X1 gets overwritten
+         for (i in 1:Time) {
+                 X1 <- X[length(X)]
+                 X[i] <- X1 + r*X1*(1-X1/K)
+         }
+         return (X)
+ }
> modelRun <- function (dados) {
+         mapply(oneRun, dados[,1], dados[,2], dados[,3])
+ }
```

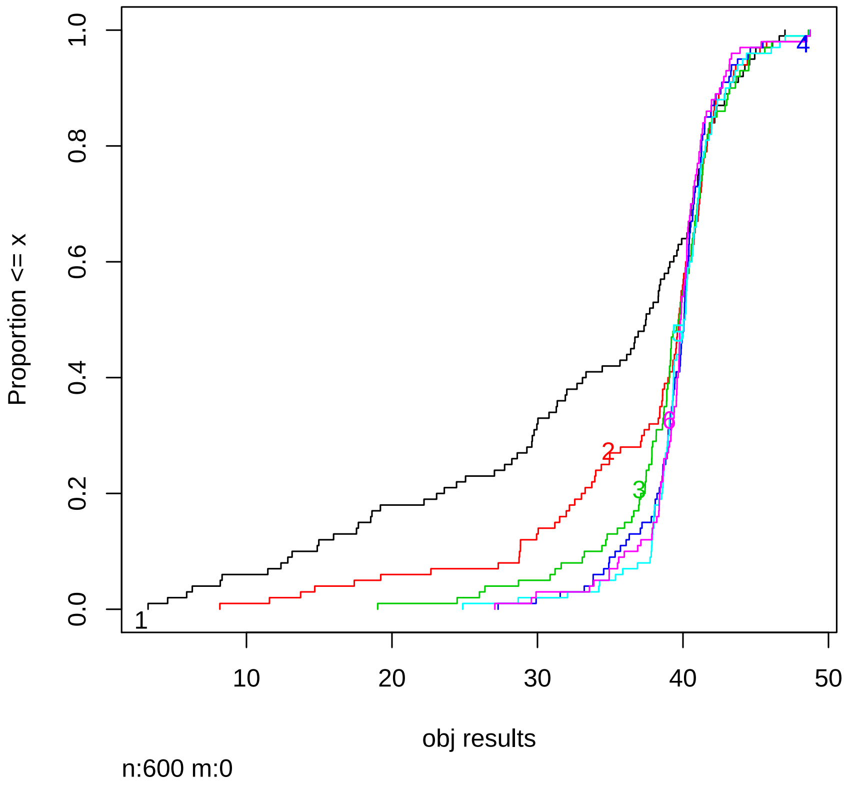
The hypercube is generated exactly in the same way. We also have the option to give names (which will be used in the plots below) to each response variable.

```
> res.names <- paste("Time",1:Time)
> myLHS <- LHS(modelRun, factors, 100, q, q.arg, res.names, nboot=50)
```

## 4.1 ECDF

The first plot we will produce will, again, be the ECDF. We may produce several plots using the parameter "stack=FALSE", or stack all the plots in the same graph, using "stack=TRUE":

```
> plotecdf(myLHS, stack=TRUE)
```

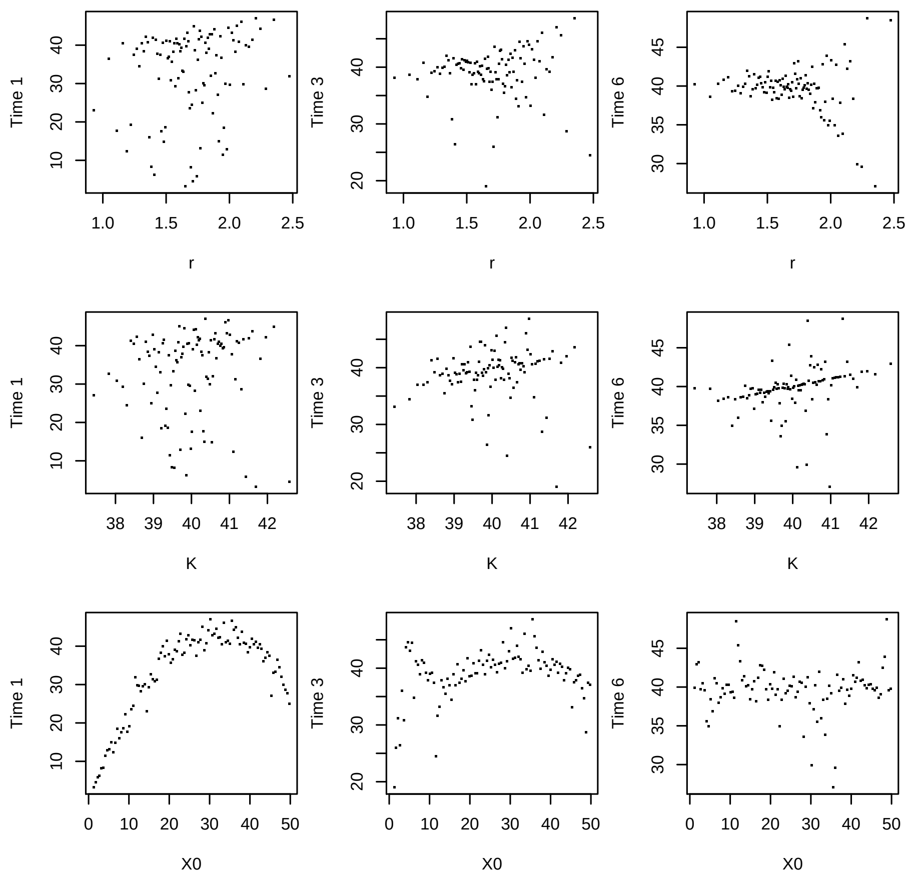


We may notice that the population values are spread over a wider range in the first time steps, but converge to a narrow distribution on time 6.

## 4.2 Scatterplots

Next, we investigate the correlation plots for the variables with each input parameter. To reduce the number of plots, we will present results just for the time steps 1, 3 and 6, using the "index.res" parameter, and supress the linear model from being ploted with the parameter "add.lm":

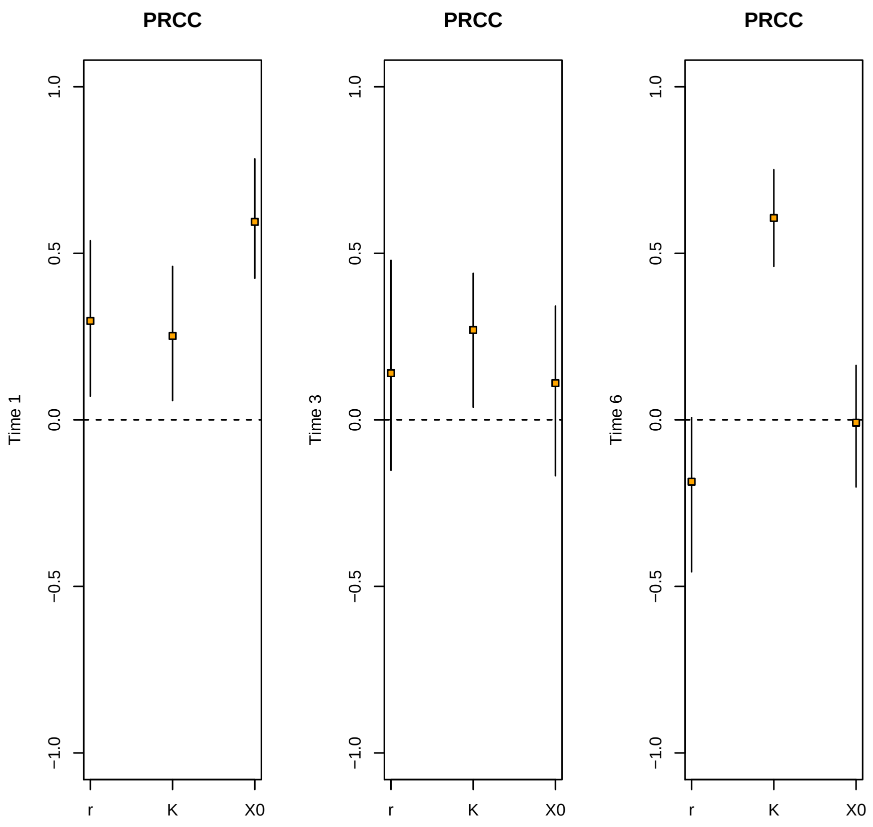
```
> plotscatter(myLHS, index.res=c(1,3,6), add.lm=FALSE)
```



## 4.3 Partial correlation

The partial correlation plots also accept the "index.res" argument:

```
> plotprcc(myLHS, index.res=c(1,3,6))
```



## 4.4 Agreement between runs

We have seen how the function **sbma** measures the agreement between two runs in the previous section. Now, we will use the function **target.sbma** to run several hypercubes until a pair of runs provides us with an agreement equal to or better than a limit specified in its first argument.

```
> targetLHS <- target.sbma (target=0.3, modelRun, factors,
+         q, q.arg, res.names, FUN=min)

[1] "INFO: initial run..."
[1] "INFO: LHS with N = 105"
[1] "sbma of -1 (target 0.3)"
[1] "INFO: LHS with N = 205"
[1] "sbma of 0.375 (target 0.3)"
```

As the SBMA is calculated for each response variable independently, we must decide how to combine these values. The argument "FUN=**min**" is telling the function to consider only the minimum value, and may be ignored for models that return a single response variable.

## 5 Uncoupling simulation and analysis

In many scenarios, it is necessary to run the simulation and the analyses at different times, and even in different computers. It may be the case, for example, that your lab computer is not fast enough to run the simulations, but that the high-performance cluster in which the simulations are to be run does not have R installed. In order to do this, however, you must generate the Latin Hypercube in the lab computer, transfer this information to the cluster, run the simulations there, transfer the results back to the lab computer, and then run the analyses.

In order to generate the samples without running a model, use the function **LHS** with the parameter *model=NULL* and save the samples in the desired format:

```
> uncoupledLHS <- LHS(model=NULL, factors, 50, q, q.arg)
> write.csv(get.data(uncoupledLHS), file="mydata.csv")
```

Then run the model using the data. To incorporate the results into the LHS object, use the function **tell**: [2]

```
> coupledLHS <- tell(uncoupledLHS, myresults)
```

Then you may proceed with the analyses using prcc, ecdf, etc.

[2]Please note that the **tell** method implemented in the sensitivity package alters its argument. This is **not** the case with the LHS **tell** method.

## References

[1] A. Chalom and P.I.K.L. Prado, *Parameter space exploration of ecological models*, arXiv:1210.6278 [q-bio.QM], 2012.

[2] C. Geyer, *Calling C and Fortran from R*, http://users.stat.umn.edu/ geyer/rc/

[3] J.D. Murray, *Mathematical Biology I: An Introduction, vol. 17 of Interdisciplinary Applied Mathematics*, Springer, New York, NY, USA,2002.

[4] R.M. May, *Simple mathematical models with very complicated dynamics*, Nature 261: 459–466, 1976.