Sensitivity Analysis of Ordinary Differential Equation Models

Methods by Morris and Sobol' and Application in R

Frank Weber, Stefan Theers, Dirk Surmann, Uwe Ligges, and Claus Weihs

May 23, 2018

Contents

1 Introduction **2**

2 Sensitivity Analysis Methods **3**

2.1 Morris Screening – A Derivative-Based Method 3

2.2 Sobol' Sensitivity Analysis – A Variance-Based Method 5

3 The R Package sensitivity **7**

3.1 Morris Screening 9

3.2 Sobol' Sensitivity Analysis 15

4 The R Package ODEsensitivity **17**

4.1 Example Case 1: The Lotka-Volterra Equations 18

4.2 Example Case 2: Low Frequency Oscillations 19

4.3 Morris Screening 22

4.3.1 General ODE Models 22

4.3.2 Low Frequency Oscillations 25

4.4 Sobol' Sensitivity Analysis 27

4.4.1 General ODE Models 27

4.4.2 Low Frequency Oscillations 29

5 Conclusion **31**

5.1 Outlook 31

References **32**

1 Introduction

The goal of sensitivity analysis is to examine how sensitive a mathematical model responds to variations in its input variables (Confalonieri et al., 2010). This incorporates, according to Confalonieri et al. (2010), the identification of relevant model inputs, model balance, model simplification and general model building. Two major approaches to sensitivity analysis can be distinguished: local and global sensitivity analysis. Local sensitivity analysis investigates the model response when only one parameter is varied when holding all other parameters at constant central values. Global sensitivity analysis investigates the model response when every existing parameter in the model is varied (see Saltelli et al., 2010 or Confalonieri et al., 2010). Although local methods are easier to implement, today global sensitivity analysis techniques are more common as their results do not depend on central values. Confalonieri et al. (2010) list three classes of global sensitivity analysis techniques: regression, screening and variance-based methods. The most well-known representatives of the two latter ones will be treated here.

This report will focus on the sensitivity analysis of ordinary differential equation (ODE) models since they can be used to model so-called *Low Frequency Oscillations* (LFOs). LFOs are permanent complex valued voltage oscillations with a frequency of up to 2 Hz occurring in electrical systems like the European electrical transmission system. As an energy network is an "electro-mechanical system" with power produced and consumed by mechanical systems, Surmann et al. (2014) point out that a mechanical system of connected harmonic oscillators is suitable for modeling LFOs. Mathematically, this model is based on a system of ordinary differential equations.

In R (a software environment for statistical computing by R Core Team, 2016) sensitivity analyses (in a general manner) can be performed using the package sensitivity (Pujol et al., 2016). Specifically for the sensitivity analysis of ODE models, the package ODEsensitivity (Theers et al., 2016) has been created. It also provides support for the package ODEnetwork (Surmann, 2015) to simplify the sensitivity analysis of ODE networks that are used to simulate LFOs. ODEsensitivity mainly relies on functions implemented in sensitivity with adaptions to the special needs when analyzing ODE models.

First, two sensitivity analysis methods (one screening and one variance-based method) are described in general in section 2. In section 3, the key functions of the sensitivity package are presented, with a focus on the changes that were necessary. The package ODEsensitivity is described in section 4. Section 5 contains a conclusion and an outlook.

2 Sensitivity Analysis Methods

2.1 Morris Screening – A Derivative-Based Method

Screening methods are probably the easiest way to analyze the sensitivity of a model function $f : \mathbb{R}^k \to \mathbb{R}$ to variations of the k inputs $x_i \in \mathbb{R}$ (with $i = 1, \dots, k$). The following assumptions by Morris (1991) will be made here, too:

1. f is at least once differentiable.
2. For given inputs x_i, the model function f evaluates without any stochastic errors (making f uniquely determined).
3. Each input x_i is a realization of a random variable X_i that is uniformly distributed on $[0, 1]$, in short: $X_i \sim U(0, 1)$. Thus, the vector $x := (x_1, \dots, x_k)^\intercal$ is always a point in the k-dimensional unit hypercube $\Omega := [0, 1]^k$. For violations of this assumption, see the note at the end of this subsection.

Since f is at least once differentiable, partial deviations $\frac{\partial f}{\partial x_i}(x)$ can be analyzed to estimate the sensitivity of f intuitively. See Table 1 for further details concerning the interpretation.

Table 1: Motivation of Morris's elementary effects (cf. Morris, 1991).

$\frac{\partial f}{\partial x_i}(x)$	effect of x_i on f
$= 0$	negligible
$=$ const. $\neq 0$	linear & additive
$= g(x_i)$ nonconst.	nonlinear
$= g(x_{j_1}, x_{j_2}, \dots)$ nonconst. with $j_1, j_2, \dots \neq i$	interactions with other inputs

This makes Morris (1991) formulate his "discretized approach": Consider a regular k-dimensional p-level grid ($p \in \mathbb{N}_{>0}$) with

$$x \in \left\{0, \frac{1}{p-1}, \frac{2}{p-1}, \dots, 1\right\}^k =: \tilde{\Omega} \subseteq \Omega.$$

Then Morris's elementary effects can be defined directly in the form of partial derivatives, see Definition 1:

Definition 1. Morris's Elementary Effects.
The elementary effect of the i-th input variable x_i on f is defined as

$$d_i(x) := \frac{f(x + \Delta\, e_i) - f(x)}{\Delta} \ \sim\ F_i \tag{1}$$

with $\Delta \in \left\{\frac{1}{p-1}, \frac{2}{p-1}, \dots, \frac{(p-1)-1}{p-1}\right\}$, $e_i \in \mathbb{R}^k$ *the* i*-th unit vector and* F_i *the discrete distribution of the* $d_i(x),\ i = 1, \dots, k$.

In order to generate a random sample of those elementary effects, Morris (1991, section 3) proposes an economical design by constructing r "trajectories" in $\tilde{\Omega}$, each one consisting of $k+1$ points (including the random starting point). This design is a so-called *one-factor-at-a-time* (OAT, in the design of experiments also abbreviated as OFAT) design because for every trajectory, each of the k directions is altered once and thus, "two successive points differ by one factor only" (Pujol, 2009, p. 3). This means that every trajectory delivers one elementary effect for each input x_i. Consequently, this approach requires $r \cdot (k+1)$ model evaluations. The typical number of trajectories is around $r \in [10, 50]$, but for ODE models, we recommend a much higher value for r (see subsection 4.3). Campolongo et al. (2007) proposed a *space-filling* optimization of Morris's OAT design: They adduce that the trajectories sampled in Morris's approach might not cover the region of interest Ω optimally. For this reason, they generate a lot more trajectories, for example $r^* \in [500, 1000]$, and finally select those r trajectories with a maximum dispersion. This requires an appropriate measure of distance between two trajectories:

Definition 2. Distance Measure $d(m, l)$.
The distance between two trajectories m *and* l *is defined as*

$$d(m,l) := \begin{cases} 0, & \text{if} \quad m = l \\ \sum_{s=1}^{k+1} \sum_{t=1}^{k+1} \sqrt{\sum_{i=1}^{k} \left(x_i^{(m,s)} - x_i^{(l,t)}\right)^2}, & \phantom{\text{if}} \quad m \neq l \end{cases} \tag{2}$$

with $x_i^{(m,s)}$ *the* i*-th component (i.e. input factor) of the* s*-th point in* Ω *of the* m*-th trajectory.*

Note that the use of the Euclidean metric in Definition 2 is arbitrary. Putting the selection process more precisely, the user sets an $r \in \mathbb{N}$ and looks out for the combination of r trajectories with a maximum sum of squared $d(m, l)$ values (i.e. a brute force search).

Another kind of sampling design was developed by Pujol (2009) and is based on simplexes. However, this design type is out of scope of this report, so please refer to the original article by Pujol (2009) for more details. We just note here that for the simplex-based design, the model is assumed to be "linear without interactions at the scale of a simplex" (Pujol, 2009, p. 8). When using the R package sensitivity, the size of the simplexes can be determined by the user, though. Both sampling design types (Morris's OAT design and Pujol's simplex-based design) are supported by the sensitivity package and can be combined with the space-filling optimization by Campolongo et al. (2007).

The actual sensitivity measures summarize the discrete distribution F_i from equation (1): Let μ_i and σ_i denote estimates for the expected value and the standard deviation of F_i, respectively. Then, μ_i is a measure for the overall influence of input x_i on the output and σ_i reflects the linearity of the influence (Fruth, 2015, p. 20). As can be seen in the third and fourth case in Table 1, "A value [of σ_i] close to 0 suggests linear behavior, a high value nonlinear or interaction behavior" (Fruth, 2015, p. 20). Additionally to μ_i and σ_i, Campolongo et al. (2007) propose another measure of influence for non-monotonic objective functions because a problem called "effects of opposite signs" might occur for such functions. If we consider an input variable for which the objective function is not monotone, the corresponding elementary effects might have opposite signs. This results in μ_i being close to zero even though x_i might have a large impact on f. Campolongo et al. (2007) use the absolute values of the elementary effects to resolve this problem of "effects of opposite signs". Let the distribution of the absolute values of the elementary effects be named G_i with an estimated expected value μ_i^*. Then, using μ_i^* instead of μ_i is a better measure for the "overall influence" of x_i on f, as shown in section 3.1. σ_i is used unrevisedly as an index for the higher order effects of x_i. In the following, μ_i, μ_i^* and σ_i will be called "Morris sensitivity indices".

Usually the domain of the input variables are different intervals than $[0, 1]$. The derived trajectories for the k-dimensional unit hypercube $\Omega = [0, 1]^k$ need to be scaled to the real domains of the input variables to evaluate the model function at points where it is well-defined. Scaling to the unit hypercube affects the calculation of the elementary effects and has to be decided case by case (see subsection 3.1 for a more detailed explanation). If the assumption of a *uniform* distribution on the domain intervals doesn't hold, the Morris screening method cannot be used and the variance-based Sobol' method should be considered instead (see subsection 2.2).

2.2 Sobol' Sensitivity Analysis – A Variance-Based Method

An alternative to the quite intuitive screening methods are variance-based methods with their key concept to partition the model output's variance like in an ANOVA. In particular, the Sobol' approach (Sobol', 1990) enjoys great popularity today and serves as a benchmark for other sensitivity analysis methods. As stated in Fruth (2015), Sobol' indices offer an access to interactions and can be interpreted clearly.

The Sobol' sensitivity measures are based on the functional ANOVA decomposition (see Theorem 1, adapted from Fruth, 2015). Define x_I for an index set $I \subseteq \{1, \ldots, k\}$ as the vector of all x_i with $i \in I$ (and still $x_i \in \mathbb{R}$).

Theorem 1. Functional ANOVA Decomposition.
Let $x := (x_1, \ldots, x_k)^\intercal$ be a k-dimensional random vector with independent components and $f : \mathbb{R}^k \to \mathbb{R}$, $f \in L_2$ with L_2 the space of square integrable functions. Then f holds

$$f(x) = f_\emptyset + \sum_{i=1}^{k} f_{\{i\}}(x_i) + \sum_{i<j} f_{\{i,j\}}(x_i, x_j) + \ldots + f_{\{1,\ldots,k\}}(x_1, \ldots, x_k) \tag{3}$$

with the 2^k summands

$$\begin{aligned} f_\emptyset &:= \mathsf{E}\left(f(x)\right), \\ f_{\{i\}}(x_i) &:= \mathsf{E}\left(f(x)|x_i\right) - f_\emptyset, \\ f_{\{i,j\}}(x_i, x_j) &:= \mathsf{E}\left(f(x)|x_i, x_j\right) - \Big[f_{\{i\}}(x_i) + f_{\{j\}}(x_j) + f_\emptyset\Big], \\ &\vdots \end{aligned} .$$

The decomposition in equation (3) is unique if $\mathsf{E}\left(f_I(x_I)\right) = 0$ for every index set $I \subseteq \{1, \ldots, k\}$ $(I \neq \emptyset)$ and if the "non-simplification condition" is fulfilled

$$\mathsf{E}\left(f_I(x_I)|x_J\right) = 0 \quad \textit{for every} \quad J \subset I \subseteq \{1, \ldots, k\}.$$

In this case, the summands in equation (3) are uncorrelated and

$$\begin{aligned} D := \mathsf{Var}\left(f(x)\right) = \mathsf{Var}\left(f_\emptyset\right) &+ \sum_{i=1}^{d} \mathsf{Var}\left(f_{\{i\}}(x_i)\right) + \sum_{i<j} \mathsf{Var}\left(f_{\{i,j\}}(x_i, x_j)\right) \\ &+ \ldots + \mathsf{Var}\left(f_{\{1,\ldots,k\}}(x_1, \ldots, x_k)\right). \end{aligned}$$

The term $f_\emptyset$ is a constant, $f_{\{i\}}(x_i)$ is called the i-th *first order effect* and $f_{\{i,j\}}(x_i, x_j)$ the *second order interaction* between x_i and x_j. Higher order interactions are defined analogously using other index sets $I \subseteq \{1, \ldots, k\}$.

On the basis of Theorem 1, we define four sensitivity indices.

Definition 3. Sobol' Sensitivity Indices.
For a subset $I \subseteq \{1, \ldots, k\}$ the value

i) $D_I := \mathsf{Var}\left(f_I(X_I)\right)$ *is called* unscaled Sobol' sensitivity index.

ii) $S_I := \frac{D_I}{D}$ *is called* scaled Sobol' sensitivity index.

iii) $D_I^T := \sum_{J \cap I \neq \emptyset} D_J$ *is called* unscaled total sensitivity index.

iv) $S_I^T := \frac{D_I^T}{D}$ *is called* scaled total sensitivity index.

$D_{\{i\}}$ reflects the first order effect of the i-th input variable. Hence, $S_{\{i\}} = \frac{D_{\{i\}}}{D}$ is called *first order* Sobol' sensitivity index. We omit the term *scaled* since only scaled indices are usually considered in application. If the index set I in D_I contains more than one element, D_I measures the "pure interaction influence of the variables indexed in I" (Fruth, 2015, p. 10). The total sensitivity index D_I^T measures the "influence of the variables [indexed in I] including all interactions of any order which contain at least one of [those variables]" (Fruth, 2015, p. 11), i.e. it reflects the individual influence of all factors in I plus every interaction effect. Thus, $S_{\{i\}}^T = \frac{D_{\{i\}}^T}{D}$ measures the first order effect of x_i *plus* all interactions with x_i involved. In short, $S_{\{i\}}^T$ is called *total* Sobol' sensitivity index and $S_{\{i\}}$ and $S_{\{i\}}^T$ in combination are called *Sobol' sensitivity indices.*

Fruth (2015) in chapter 3.3.2, Saltelli et al. (2010) in chapter 8.3 or Sobol' (2001) point out that estimations of all these Sobol' sensitivity indices can be obtained by *Monte Carlo simulation.* Jansen (1999) as well as Martinez (2011) both developed methods for estimating the Sobol' sensitivity indices. Both approaches, like several other techniques, are based on Monte Carlo simulation and are implemented in the sensitivity package. This report focuses on the Jansen and the Martinez methods since they are numerically stable and used in the package ODEsensitivity. However, for all methods that are based on Monte Carlo estimation, the underlying distributions of all k input variables need to be known to draw random samples from those distributions.

3 The R Package sensitivity

Both – Morris screening and the variance-based Sobol' method – are implemented in the R package sensitivity. The R version used for this report is 3.2.5. The goal of this section is to exemplarily perform a sensitivity analysis for the so-called *Sobol' g-function.* It is scalar-valued with eight input variables and is one of the most common analytical test functions presented in Saltelli et al. (2010, chap. 2). In addition, it is an example of a non-monotonic function for which the problem of "effects of opposite signs" occurs (see subsection 2.1). The definition of the Sobol' g-function can be found in Saltelli et al. (2010, pp. 39-40):

$$f(x) := \prod_{i=1}^{8} g_i(x_i) \qquad \text{with} \qquad g_i(x_i) := \frac{|4x_i - 2| + a_i}{1 + a_i} \tag{4}$$

and the coefficients $a_1 = 0, a_2 = 1, a_3 = 4.5, a_4 = 9, a_5 = a_6 = a_7 = a_8 = 99$. For the Sobol' g-function, all input variables are uniformly distributed on $[0, 1]$. Thus, both presented methods for sensitivity analysis – Morris screening and the Sobol' variance-based method – can be applied here. The Sobol' g-function is defined in sensitivity as follows:

```
library("sensitivity")
sobol.fun

## function (X)
## {
##     a <- c(0, 1, 4.5, 9, 99, 99, 99, 99)
##     y <- 1
##     for (j in 1:8) {
##         y <- y * (abs(4 * X[, j] - 2) + a[j])/(1 + a[j])
##     }
##     y
## }
## <bytecode: 0x0000000009d93050>
## <environment: namespace:sensitivity>
```

`sobol.fun()` accepts a matrix `X` as input and returns a vector of function values at the eight-dimensional points contained in the n rows of `X`. When running the sensitivity analysis, `X` contains the n sampled combinations of the 8 input variables.

An optimal performance using sensitivity is achieved if the model function $f : \mathbb{R}^k \to \mathbb{R}$ is implemented in R as a function accepting a *matrix* as input and returning a numeric vector.

When analyzing the sensitivity of an m-dimensional ODE model at q distinct timepoints, the runtime can be decreased substantially when returning the function values of all m objective functions at all q timepoints simultaneously. For each *row* of `X`, a vector of length q (for $m = 1$) or a matrix of size $q \times m$ (for $m > 1$) is returned. The model function in R then returns a matrix of size $n \times q$ (for $m = 1$) or a three-dimensional array of size $n \times q \times m$ (for $m > 1$). Prior to version 1.12.1, sensitivity could not deal with model functions returning a matrix or a three-dimensional array. As of version 1.12.1, the functions `morris()`, `soboljansen()` and `sobolmartinez()` support those model functions.

In order to demonstrate this new feature, we extend the result of `sobol.fun()` to a matrix with two columns by appending its original result times 2:

```
sobol.fun_matrix <- function(X){
  res_vector <- sobol.fun(X)
  cbind(res_vector, 2 * res_vector)
}
```

Analogously, a model function returning a three-dimensional array is created by doubling the results from `sobol.fun_matrix()` once again and combining those two matrices to a three-dimensional array of dimension lengths $(n, 2, 2)$:

```
sobol.fun_array <- function(X){
  res_vector <- sobol.fun(X)
  res_matrix <- cbind(res_vector, 2 * res_vector)
  array(data = c(res_matrix, 2 * res_matrix),
        dim = c(length(res_vector), 2, 2))
}
```

For Morris screening, the elementary effects and the sensitivity indices double when analyzing the doubled results. In contrast, the Sobol' sensitivity indices don't change at all since they are scaled.

3.1 Morris Screening

In sensitivity, Morris screening is called by using the `morris()` function. A seed is set in order to get the same pseudo-random numbers when analyzing `sobol.fun_array()` and `sobol.fun_matrix()`:

```
set.seed(2849)
mor_matrix <- morris(model = sobol.fun_matrix, factors = 8, r = 50,
                     design = list(type = "oat", levels = 10, grid.jump = 1),
                     binf = 0, bsup = 1, scale = FALSE)
set.seed(2849)
mor_array <- morris(model = sobol.fun_array, factors = 8, r = 50,
                    design = list(type = "oat", levels = 10, grid.jump = 1),
                    binf = 0, bsup = 1, scale = FALSE)
```

Argument `model` contains the model function to be analyzed (or alternatively a `predict()` method, see the `morris()` help page of sensitivity). Argument `factors` contains the number of input variables k (or a character string with the names of input variables) and `r` sets the number of elementary effects to compute per input variable. The sampling design is specified using argument `design` (which needs to be supplied as a list):

- If `type = "oat"` (as used here), Morris's OAT design is used. In this case, parameter p from subsection 2.1 has to be specified using argument `levels` in the list `design`. Argument `grid.jump` sets the number of levels that are vaulted in each step (in subsection 2.1, `grid.jump` equals 1).
- If `type = "simplex"`, the simplex-based design by Pujol (2009) is used. In this case, the list supplied for `design` contains a numeric argument `scale.factor` which sets an expansion factor for the simplexes (for `scale.factor = 1`, all edges of the simplexes have length one).

For the space-filling optimization by Campolongo et al. (2007), which can be applied to both sampling design types, a vector of length *two* has to be supplied for argument `r`: The first element indicating the number of elementary effects, the second one the number r^* of repetitions from which the trajectories with maximum dispersion are chosen (see subsection 2.1).

Finally, three arguments (`binf`, `bsup` and `scale`) apply to both sampling designs. Arguments `binf` and `bsup` specify the lower and upper boundaries for intervals on which input factors are assumed to be uniformly distributed. The logical argument `scale` rescales the trajectory points to $[0, 1]$ *after* evaluating the model and *before* calculating the elementary effects. The meaning of "impact of an input variable on the output variable" decides if `scale = TRUE` or `scale = FALSE` should be used: Consider two input variables, one varying in a small range, the other one varying in a large range. Let both cause the same *absolute* change of f across their ranges. Then, if `scale = TRUE`, both input variables are considered to have the same impact on f. In contrast, if `scale = FALSE`, the input variable varying in the small range is considered to have a larger impact on f than the one varying in the large range.

The resulting objects `mor_matrix` and `mor_array` are both lists of class `"morris"` with the following structures:

```r
str(mor_matrix[c("X", "y", "ee")], vec.len = 1, give.attr = FALSE)

## List of 3
##  $ X : num [1:450, 1:8] 0.111 ...
##  $ y : num [1:450, 1:2] 0.961 ...
##  $ ee: num [1:50, 1:8, 1:2] -2.42 ...

str(mor_array[c("X", "y", "ee")], vec.len = 1, give.attr = FALSE)

## List of 3
##  $ X : num [1:450, 1:8] 0.111 ...
##   $ y : num [1:450, 1:2, 1:2] 0.961 ...
##   $ ee: num [1:50, 1:8, 1:2, 1:2] -2.42 ...
```

`X` is the matrix of the $r \cdot (k+1)$ sampled points in the k-dimensional parameter space. In this example, there are $r \cdot (k+1) = 50 \cdot 9 = 450$ sampled points. Since we use the same seed, `mor_matrix` and `mor_array` contain the same matrix `X`. Element `y` contains the corresponding 450 function values and `ee` the $r = 50$ elementary effects for all eight input variables and all dimensions of `y` except the first. If we use `mor_matrix`, `ee` is a three-dimensional array with the third dimension corresponding to the columns of `y`. For `mor_array`, `ee` is a four-dimensional array with the third and fourth dimension corresponding to the second and third dimension of `y`, respectively.

The Morris sensitivity indices μ_i^* and σ_i (here for $i = 1, \ldots, 8$) need to be calculated manually from the elementary effects in element `ee` (the following code can also be found on the `morris()` help page in sensitivity):

```r
# For sobol.fun_matrix():
mu_matrix <- apply(mor_matrix$ee, 3, function(M){
  apply(M, 2, mean)
})
mu.star_matrix <- apply(abs(mor_matrix$ee), 3, function(M){
  apply(M, 2, mean)
})
sigma_matrix <- apply(mor_matrix$ee, 3, function(M){
  apply(M, 2, sd)
})

# For sobol.fun_array():
mu_array <- sapply(1:dim(mor_array$ee)[4], function(i){
  apply(mor_array$ee[, , , i, drop = FALSE], 3, function(M){
    apply(M, 2, mean)
  })
}, simplify = "array")
mu.star_array <- sapply(1:dim(mor_array$ee)[4], function(i){
  apply(abs(mor_array$ee)[, , , i, drop = FALSE], 3, function(M){
    apply(M, 2, mean)
  })
}, simplify = "array")
```

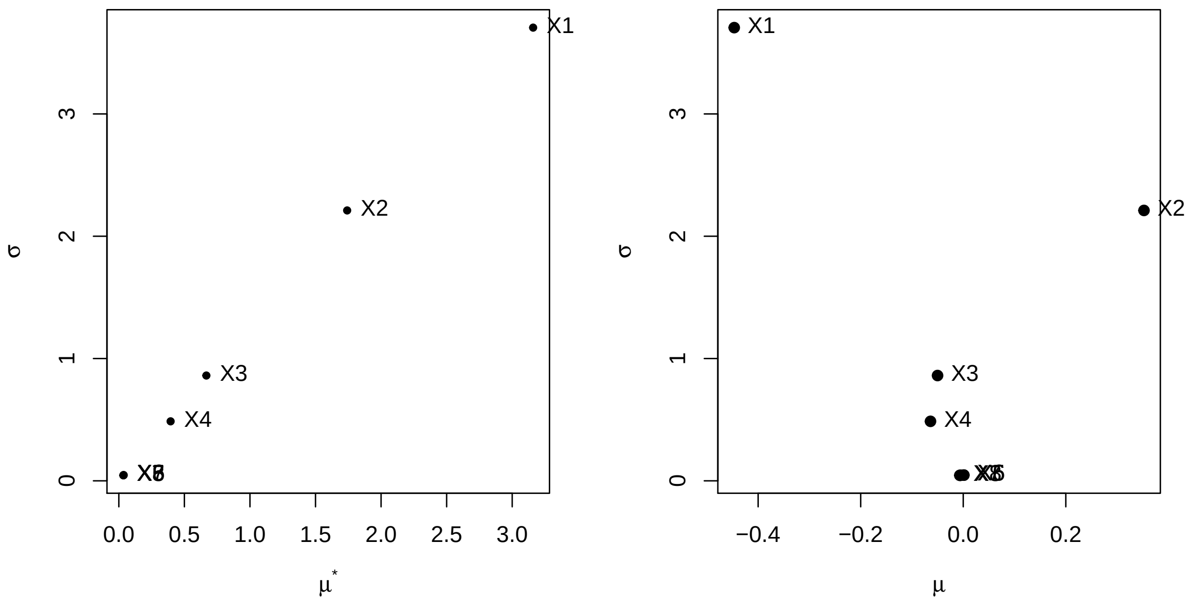
```
sigma_array <- sapply(1:dim(mor_array$ee)[4], function(i){
  apply(mor_array$ee[, , , i, drop = FALSE], 3, function(M){
    apply(M, 2, sd)
  })
}, simplify = "array")
```

If the model output is doubled, the elementary effects are doubled as well and so are the Morris sensitivity indices:

```
# Elementary effects are doubled:
c(all.equal(2 * mor_matrix$ee[, , 1], mor_matrix$ee[, , 2]),
  all.equal(2 * mor_matrix$ee, mor_array$ee[, , , 2]))

## [1] TRUE TRUE

# Morris sensitivity indices are doubled:
c(all.equal(2 * mu_matrix[, 1], mu_matrix[, 2]),
  all.equal(2 * mu.star_matrix[, 1], mu.star_matrix[, 2]),
  all.equal(2 * sigma_array[, , 1], sigma_array[, , 2]))

## [1] TRUE TRUE TRUE
```

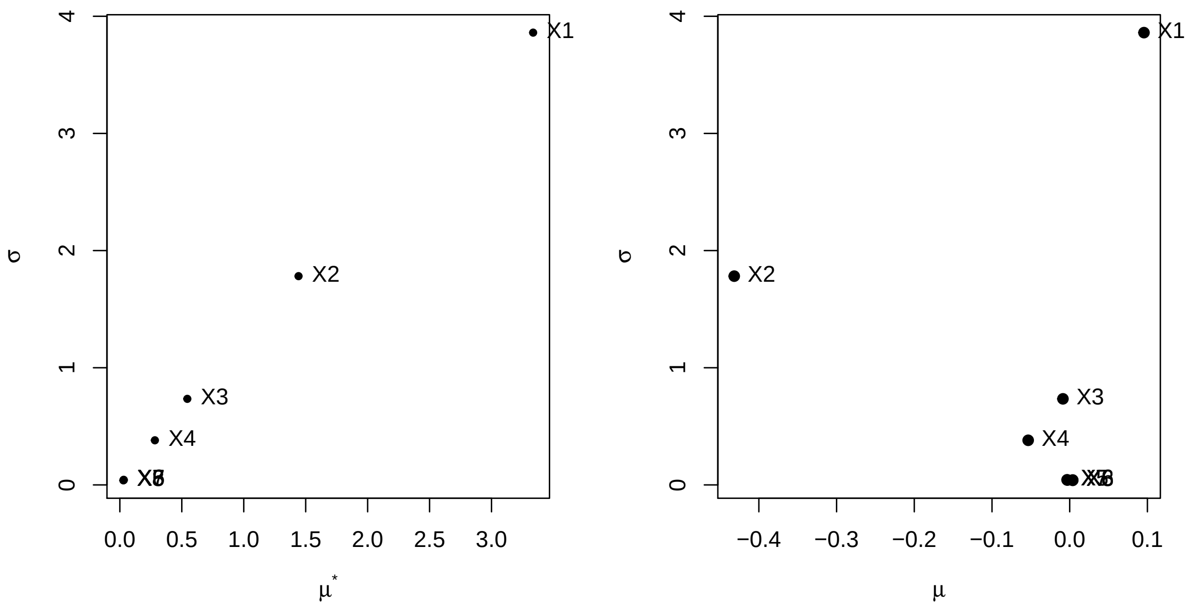
sensitivity provides a `plot()` method for objects of class `"morris"` to visualise the Morris sensitivity indices. At first, consider a plot of the Morris sensitivity indices for the first column of `mor_matrix$y` (the original `sobol.fun()`-results). Differences between μ and μ^* are illustrated in a plot of μ_i against σ_i (for $i = 1, \ldots, 8$) for the same column of `mor_matrix$y`:

```
par(mfrow = c(1, 2), mar = c(5, 4, 2, 2) + 0.1, xpd = TRUE)
plot(mor_matrix)
plot(mu_matrix[, 1], sigma_matrix[, 1], xlab = expression(mu),
     ylab = expression(sigma), pch = 19)
text(mu_matrix[, 1], sigma_matrix[, 1], labels = paste0("X", 1:8), pos = 4)
```



The μ^*-plot on the left shows the large impact of x_1 on f, followed by x_2, x_3 and x_4 (in this order). The impact of all other input variables can't be distinguished from another because they are all nearly zero. Apart from that, the influence of x_1 seems to be nonlinear or dominated by interactions. The same applies lesser for x_2. An interpretation of μ is more difficult than that of μ^* if we study the same situation with a different seed:

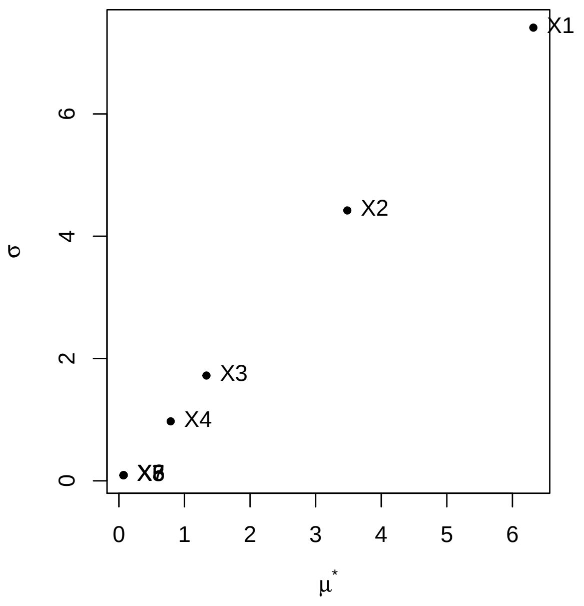
```
set.seed(2015)
mor_matrix2 <- morris(model = sobol.fun_matrix, factors = 8, r = 50,
                      design = list(type = "oat", levels = 10, grid.jump = 1),
                      binf = 0, bsup = 1, scale = FALSE)
mu_matrix2 <- apply(mor_matrix2$ee, 3, function(M){
  apply(M, 2, mean)
})
sigma_matrix2 <- apply(mor_matrix2$ee, 3, function(M){
  apply(M, 2, sd)
})
par(mfrow = c(1, 2), mar = c(5, 4, 2, 2) + 0.1, xpd = TRUE)
plot(mor_matrix2)
plot(mu_matrix2[, 1], sigma_matrix2[, 1], xlab = expression(mu),
     ylab = expression(sigma), pch = 19)
text(mu_matrix2[, 1], sigma_matrix2[, 1], labels = paste0("X", 1:8), pos = 4)
```

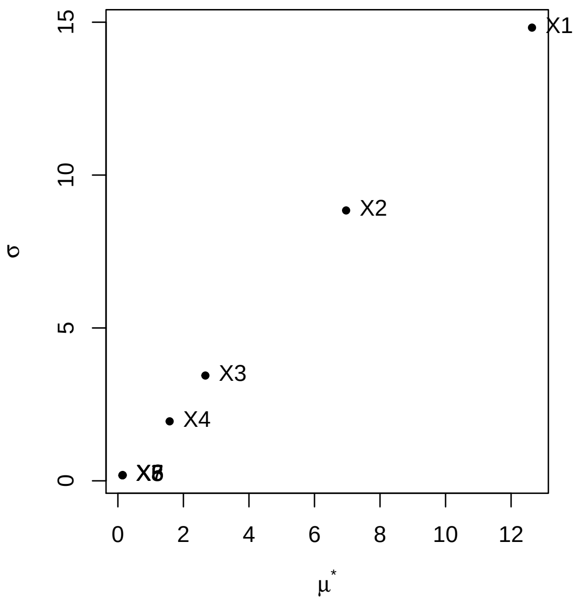


Now, μ_1 and μ_2 have opposite signs and different absolute values whereas all μ_i^* nearly remain unchanged.

Two arguments were added to the `plot.morris()` method in order to support matrices and three-dimensional arrays as model outputs: The first one, `y_col`, sets the column index of `y` for which the Morris sensitivity indices are plotted. The second one, `y_dim3`, does the same for the third dimension of `y` (if there is any). The default for both arguments is 1, so the first column and the first element of the third dimension is used (if existing). To demonstrate the use of `y_col` and `y_dim3`, the sensitivity indices for the second column of the first and second element in the third dimension of `mor_array$y` are plotted. The plots show the expected doubled Morris sensitivity indices if the model output is doubled:

```r
par(mfrow = c(1, 2), mar = c(5, 4, 2, 2) + 0.1, xpd = TRUE)
plot(mor_array, y_col = 2, y_dim3 = 1)
plot(mor_array, y_col = 2, y_dim3 = 2)
```





3.2 Sobol' Sensitivity Analysis

There are various implementations of the Sobol' variance-based sensitivity analysis in the sensitivity package (e.g. `sobol()`, `sobol2002()`, `sobol2007()`, `soboljansen()` and `sobolmartinez()`). The syntax of all these implementations is very similar and only differs in details. However, only `soboljansen()` and `sobolmartinez()` have been adapted for the use with model functions returning a matrix or a three-dimensional array since they are used in the ODEsensitivity package for the sensitivity analysis of ODE models. The use of `soboljansen()` will be exemplarily presented here.

At first, n random samples have to be drawn *twice* from the distribution of $x = (x_1, \dots, x_k)^\top$ for the Monte Carlo estimation of the Sobol' sensitivity indices. For the Sobol' g-function, we can assume a uniform distribution on $[0, 1]$ for all x_i's ($i = 1, \dots, 8$). Thus, the random samples can be generated as follows:

```r
set.seed(9204)
n <- 1000
X1 <- data.frame(matrix(runif(8 * n), nrow = n))
X2 <- data.frame(matrix(runif(8 * n), nrow = n))
```

Next, using these random samples `X1` and `X2`, the sensitivity analysis for `sobol.fun_matrix()` and `sobol.fun_array()` is performed by a call to `soboljansen()`:

```
sob_matrix <- soboljansen(model = sobol.fun_matrix, X1, X2, nboot = 0)
sob_array <- soboljansen(model = sobol.fun_array, X1, X2, nboot = 0)
```

Note that bootstrapping is not recommended for ODE models because of high computational costs, so argument `nboot` is set to 0.

The resulting objects (of class `"soboljansen"`) include both the first order Sobol' sensitivity indices ($S_{\{i\}}$ from subsection 2.2):

```
str(sob_matrix$S, give.attr = FALSE); str(sob_array$S, give.attr = FALSE)


##  num [1:8, 1:2] 0.71 0.1955 0.0616 0.0289 0.0193 ...
##  num [1:8, 1:2, 1:2] 0.71 0.1955 0.0616 0.0289 0.0193 ...
```

and the total Sobol' sensitivity indices ($S^T_{\{i\}}$):

```
str(sob_matrix$T, give.attr = FALSE); str(sob_array$T, give.attr = FALSE)


##  num [1:8, 1:2] 0.755609 0.233964 0.035171 0.010676 0.000106 ...
##  num [1:8, 1:2, 1:2] 0.755609 0.233964 0.035171 0.010676 0.000106 ...
```

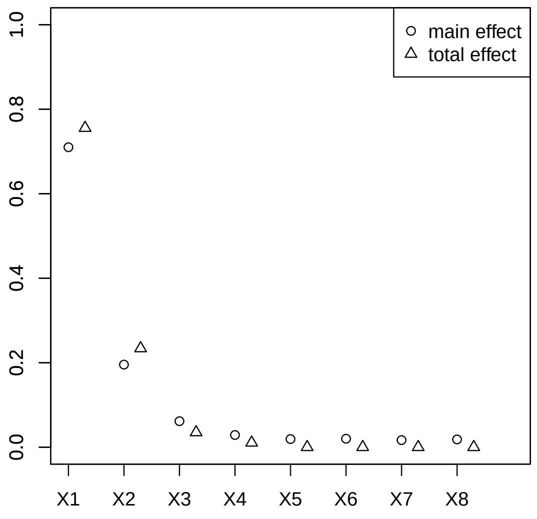
The dimensions of these objects agree with the dimensions of the corresponding object `y`: `sob_matrix$S` and `sob_matrix$T` are matrices with two columns containing the Sobol' sensitivity indices for the two columns of `sob_matrix$y`. `sob_array$S` and `sob_array$T` are three-dimensional arrays containing the Sobol' sensitivity indices for the second and third dimension of `sob_array$y`.

As expected, the (scaled) Sobol' sensitivity indices (both total and first order) don't change at all when analyzing the doubled results from `sobol.fun()`:

```
c(all.equal(sob_matrix$S[, 1], sob_matrix$S[, 2]),
  all.equal(sob_matrix$T[, 1], sob_matrix$T[, 2]),
  all.equal(sob_array$S[, , 1], sob_array$S[, , 2]),
  all.equal(sob_array$T[, , 1], sob_array$T[, , 2]))


## [1] TRUE TRUE TRUE TRUE
```

The Sobol' sensitivity indices can be visualized again by a simple call to `plot()`. The `plot()` methods `plot.soboljansen()` and `plot.sobolmartinez()` also support the two new arguments `y_col` and `y_dim3`. The use of `plot.soboljansen()` is demonstrated below:

```r
par(mar = c(3, 2, 2, 2) + 0.1, xpd = TRUE)
plot(sob_array, y_col = 2, y_dim3 = 2)
```



The results from Sobol' sensitivity analysis correspond to those from Morris screening: With respect to both the first order and the total Sobol' indices, x_1 has the largest impact on f, followed by x_2, x_3 and x_4. The influence of all other input variables can't be distinguished one from another, they are all nearly zero. Note that it's rather the *total* Sobol' sensitivity index which can be compared to μ^* since they both measure the influence of an input variable *and* of all the interactions this variable is involved in.

4 The R Package ODEsensitivity

The sensitivity analysis of ODE models is computationally difficult for two main reasons: Firstly, a separate sensitivity analysis has to be performed for every timepoint of interest. Secondly, for a multi-dimensional ODE model, different output variables (*state variables* in the terminology of ODE models) have to be analyzed simultaneously. To ease the sensitivity analysis of ODE models, the R package ODEsensitivity has been created.

Two techniques of sensitivity analysis are implemented in ODEsensitivity: Morris screening and the Sobol' sensitivity analysis. `ODEmorris()` and `ODEsobol()`, the two main functions of ODEsensitivity, correspond to those two techniques. They rely on the functions `morris()`, `soboljansen()` and `sobolmartinez()` from the sensitivity package. `ODEmorris()` and `ODEsobol()` are generic functions and they both have two methods: one default method for analyzing general ODE models and one method for analyzing ODE networks that are used for simulating low frequency oscillations (LFOs). This is why two example cases will be presented in this section: one for general ODE models (the *Lotka-Volterra equations*) and one for simulating LFOs.

4.1 Example Case 1: The Lotka-Volterra Equations

The *Lotka-Volterra equations* describe a predator and its prey's population development and go back to Lotka (1925) and Volterra (1926). The prey's population at time t (in days) will be denoted with $P(t)$ and the predator's (or rather consumer's) population with $C(t)$. $P(t)$ and $C(t)$ are called *state* variables. This ODE model is two-dimensional, but it should be noted that ODE models of arbitrary dimensions (including one-dimensional ODE models) can be analyzed with ODEsensitivity. The Lotka-Volterra equations as presented in Soetaert et al. (2010) are:

$$\begin{aligned} \dot{P}(t) &:= \frac{\partial}{\partial t}P(t) = r_{\mathrm{G}}\,P(t)\left(1-\frac{P(t)}{K}\right) - r_{\mathrm{I}}\,P(t)\,C(t) \\ \dot{C}(t) &:= \frac{\partial}{\partial t}C(t) = k_{\mathrm{AE}}\,r_{\mathrm{I}}\,P(t)\,C(t) - r_{\mathrm{M}}\,C(t). \end{aligned} \tag{5}$$

In fact, this is a combination of the *normal* and the *competitive* Lotka-Volterra model since it assumes an exponential reduction of the predator's population in the absence of the prey, but a logistic growth of the prey's population in the absence of the predator. The $k := 5$ parameters are the prey's growth rate r_{G}, the carrying capacity K, the consumer's ingestion rate r_{I}, the consumer's assimilation efficiency k_{AE} and the consumer's mortality rate r_{M} (Soetaert et al., 2010). r_{G}, r_{I}, r_{M} and k_{AE} are dimensionless and take on values in $[0, 1]$. The carrying capacity K is of the same dimension as $P(t)$ (and $C(t)$) and is assumed to take on values in $[1, 20]$.

In R, the function `ode()` from the package deSolve (Soetaert et al., 2016) is able to calculate the values of the state variables at different timepoints for given initial values and given parameter settings. This is what is meant by "solving" an ODE system. For `ode()`, the model function has to be supplied in a specific manner:

```r
LVmod <- function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    Ingestion    <- rIng  * Prey * Predator
    GrowthPrey   <- rGrow * Prey * (1 - Prey/K)
    MortPredator <- rMort * Predator
    dPrey         <- GrowthPrey - Ingestion
    dPredator     <- Ingestion * assEff - MortPredator

    return(list(c(dPrey, dPredator)))
  })
}
```

Each of the state variables $P(t)$ and $C(t)$ corresponds to the model function f from section 2. The $k = 5$ parameters r_G, r_I, r_M, k_AE and K are considered as input variables for the sensitivity analysis. Hence, we will analyze the sensitivity of the prey's and the predator's population with regard to changes in these 5 parameters.

The parameters names, their lower and upper boundaries, the initial values for the state variables and the timepoints of interest are saved in separate vectors. Here, we will use initial values of 1 for the prey's population and 2 for the predator's population. The time interval of interest is $(0, 50]$ (days) which is covered in discrete steps of one day.

```
LVpars  <- c("rIng", "rGrow", "rMort", "assEff", "K")
LVbinf <- c(0.05, 0.05, 0.05, 0.05, 1)
LVbsup <- c(1.00, 3.00, 0.95, 0.95, 20)
LVinit  <- c(Prey = 1, Predator = 2)
LVtimes <- c(0.01, seq(1, 50, by = 1))
```

Note that the vector of initial values (here `LVinit`) has to be a named vector and that the smallest timepoint has to be strictly positive.

4.2 Example Case 2: Low Frequency Oscillations

For modelling low frequency oscillations in an energy network of n nodes, a system of n connected harmonic oscillators can be used (Surmann et al., 2014). Each node of the energy network is then represented by a mass m_i ($i \in \{1, \ldots, n\}$) connected to the ground by a damper d_i and a spring of length r_i. The spring constant of spring i will be denoted by k_i. Masses i and j ($i, j \in \{1, \ldots, n\}$ with $i \neq j$) are interconnected by springs of lengths $r_{i,j}$ and spring constants $k_{i,j}$. The damper between mass i and mass j has a damping constant denoted by $d_{i,j}$. The deflection $x_i(t)$ of mass i at time t corresponds to the deflection of the voltage magnitude or angle (by choice) at node i in the energy network at time t. Figure 1 illustrates the setting and the notation for a network of $n = 4$ oscillators.

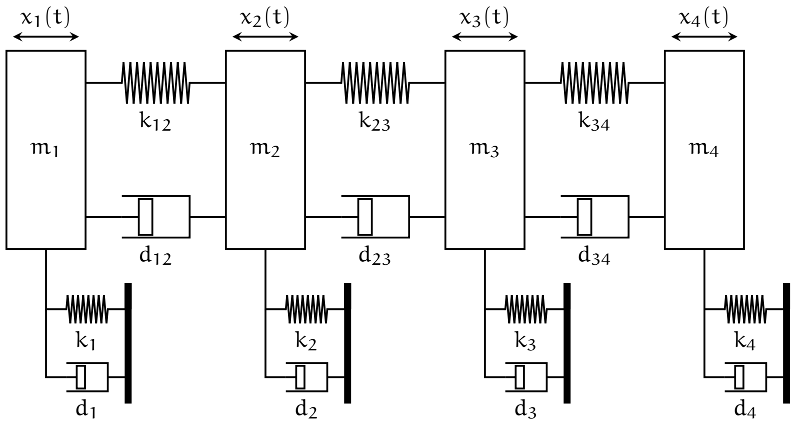


Figure 1: Example of $n = 4$ interconnected mechanical oscillators, adapted from Surmann et al. (2014). Note that the spring and the damper connecting mass 1 and mass 4 (with spring constant $k_{1,4}$ and damping constant $d_{1,4}$, respectively) are not plotted due to clarity.

The oscillation of the masses $m_1, \dots, m_n$ is then modelled by the following n-dimensional system of ordinary differential equations:

$$0 = M\ddot{x}(t) + D\dot{x}(t) + Kx(t) + b \tag{6}$$

$$\Leftrightarrow \quad 0 = \ddot{x}(t) + \underbrace{M^{-1}D}_{=:A}\dot{x}(t) + \underbrace{M^{-1}K}_{=:B}x(t) + \underbrace{M^{-1}b}_{=:g}. \tag{7}$$

with

- $x(t) := (x_1(t), \dots, x_n(t))^\intercal \in \mathbb{R}^n$,
- $M := \text{diag}\{m_1, \dots, m_n\} \in \mathbb{R}^{n\times n}$,
- $D := (d^*_{i,j}) \in \mathbb{R}^{n\times n} \quad \text{with} \quad d^*_{i,j} := \begin{cases} -d_{i,j}, & i \neq j \\ d_i + \sum_{j\neq i} d_{i,j}, & i = j \end{cases} \quad \text{for } i,j \in \{1,\dots,n\},$
- $K := (k^*_{i,j}) \in \mathbb{R}^{n\times n} \quad \text{with} \quad k^*_{i,j} := \begin{cases} -k_{i,j}, & i \neq j \\ k_i + \sum_{j\neq i} k_{i,j}, & i = j \end{cases} \quad \text{for } i,j \in \{1,\dots,n\},$
- $b \in \mathbb{R}^n \quad \text{with} \quad b_i := -k_i\, r_i + \sum_{j\neq i} k_{i,j}\, r^*_{i,j} \quad \text{and} \quad r^*_{i,j} := \begin{cases} r_{i,j}, & i < j \\ r_{j,i} = -r_{i,j}, & i > j. \end{cases}$

The first and second derivatives $\dot{x}(t) \in \mathbb{R}^n$ and $\ddot{x}(t) \in \mathbb{R}^n$ of $x(t)$ with respect to time describe the velocity and acceleration of all n masses, respectively. Equation (6) is transformed to equation (7) since equation (6) is over-parametrized by the masses M (Surmann et al., 2014). Equation (7) (a system of n ODEs of second order) is then transformed into a system of $2n$ ODEs of first order using the substitution $f(t) := (x^\intercal(t), \dot{x}^\intercal(t))^\intercal \in \mathbb{R}^{2n}$:

$$\dot{f}(t) = \underbrace{\begin{bmatrix} 0_n & I_n \\ -B & -A \end{bmatrix}}_{=:C} f(t) + \underbrace{\begin{bmatrix} 0 \\ -g \end{bmatrix}}_{=:h}. \tag{8}$$

As Surmann et al. (2014) describe in more detail, this ODE system can be solved analytically. However, when simulating oscillations that are disturbed by external forces, the solution has to be determined numerically (Surmann et al., 2014).

Here, we will exemplarily perform a sensitivity analysis for an $n := 4$-dimensional ODE network that can be solved analytically. In this example network, the masses will be connected in a "circle", which means that mass i is connected to mass $i + 1$ (for $i = 1, 2, 3$) and mass 4 is connected to mass 1. According to Surmann et al. (2014), we choose $m_i := 2 \ \forall\, i \in \{1, \dots, 4\}$ and $k_i := m_i(2\pi \cdot 0.17)^2 \approx 2.28 \ \forall\, i \in \{1, \dots, 4\}$ to obtain an oscillation frequency of 0.17 Hz for each mass. The spring constants for the interconnecting springs are set to $k_{1,2} := k_{2,3} := k_{3,4} := k_{1,4} := \frac{k_1}{10} \approx 0.228$ and to zero in every other case (if $k_{i,j} = 0$, the masses i and j are not connected). The damping constants are set to $d_i := 0.05 \ \forall\, i \in \{1, \dots, 4\}$ and to $d_{i,j} := 0 \ \forall\, i, j \in \{1, \dots, 4\}$ with $i \neq j$. All lengths of the springs are set to zero ($r_i := 0, r_{i,j} := 0 \ \forall\, i, j \in \{1, \dots, n\}$) to avoid any ground potential for the springs. Next, an object of class `"ODEnetwork"` containing all the information is created in R:

```
M_mat <- rep(2, 4)
K_mat <- diag(rep(2 * (2*pi*0.17)^2, 4))
K_mat[1, 2] <- K_mat[2, 3] <-
   K_mat[3, 4] <- K_mat[1, 4] <- 2 * (2*pi*0.17)^2 / 10
D_mat <- diag(rep(0.05, 4))
library("ODEnetwork")
lfonet <- ODEnetwork(masses = M_mat, dampers = D_mat, springs = K_mat)
```

The state variables in this ODE model are the deflections $x_1(t), \dots, x_4(t)$ and the velocities $v_1(t) := \dot{x}_1(t), \dots, v_4(t) := \dot{x}_4(t)$. The starting velocities are all set to zero: $v_i(0) := \dot{x}_i(0) := 0 \ \forall\, i \in \{1, \dots, 4\}$. The oscillation is only initiated by the starting positions $x_i(0) := 2 \ \forall\, i \in \{1, \dots, 4\}$ differing by two units from $r_i = 0 \,\forall\, i \in \{1, \dots, 4\}$. We create a vector of the parameter names that should be involved in the sensitivity analysis. Those parameters are $k_1, \dots, k_4$ and $d_1, \dots, d_4$. The lower and upper boundaries for $k_1, \dots, k_4$ are set to 0.2 and 20, respectively (together with $m_i = 2$, this gives oscillation frequencies in the (approximate) interval $(0.05, 0.5)$).

The boundaries for $d_1, \dots, d_4$ are set to 0.01 and 0.1, respectively. The time interval of interest is $[25, 150]$ (seconds) and it is covered in steps of 2.5 seconds:

```
lfonet <- setState(lfonet, state1 = rep(2, 4), state2 = rep(0, 4))
LFOpars <- c("k.1", "k.2", "k.3", "k.4",
             "d.1", "d.2", "d.3", "d.4")
LFOtimes <- seq(25, 150, by = 2.5)
LFObinf <- c(rep(0.2, 4), rep(0.01, 4))
LFObsup <- c(rep(20, 4), rep(0.1, 4))
```

4.3 Morris Screening

4.3.1 General ODE Models

The sensitivity analysis of a general ODE model (here the Lotka-Volterra example introduced in subsection 4.1) can be performed by using the generic function `ODEmorris()`. The first six arguments (`mod` to `bsup`) of the default method `ODEmorris.default()` accept the corresponding objects created in subsection 4.1. Note that argument `mod` is specified in the way `ode()` from the package deSolve expects its model functions. Arguments `r`, `design` and `scale` are exactly the same as for `morris()` from the sensitivity package. See subsection 3.1 for more information on these arguments. Here, we choose Morris's OAT design with `design$levels = 10` and `design$grid.jump = 1`. For ODE models, we recommend a much higher number r of replications than the typically used $r \in [10, 50]$ for non-ODE models. Here, we use `r = 500` replications. Argument `ode_method` sets the name of the integrator to use for `ode()` from the package deSolve. In this example, we will use the default integrator `"lsoda"`. The last two arguments indicate if the evaluation of the model should be parallelized and if yes, how many processor cores shall be used. We decide to parallelize using two processor cores.

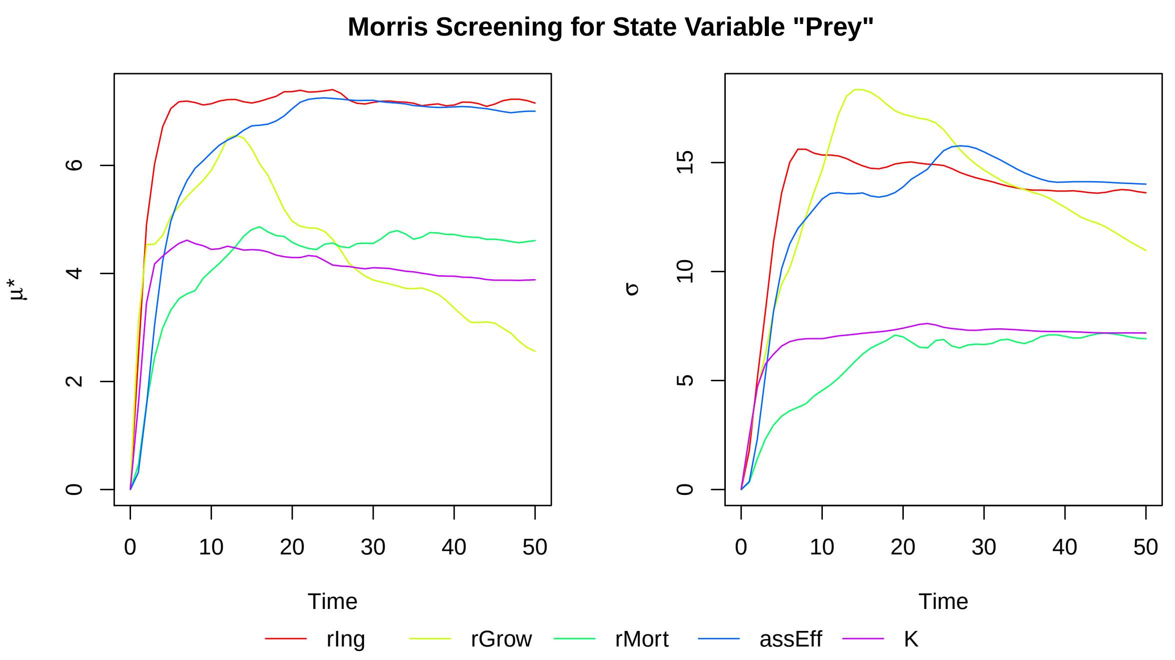
```
library("ODEsensitivity")
set.seed(7292)
LVres_morris <- ODEmorris(mod = LVmod,
                          pars = LVpars,
                          state_init = LVinit,
                          times = LVtimes,
                          binf = LVbinf,
                          bsup = LVbsup,
                          r = 500,
                          design = list(type = "oat",
                                        levels = 10, grid.jump = 1),
                          scale = TRUE,
                          ode_method = "lsoda",
                          parallel_eval = TRUE,
                          parallel_eval_ncores = parallel::detectCores())
```

We will now examine the output of `ODEmorris.default()`:

```
str(LVres_morris, vec.len = 3, give.attr = FALSE)


## List of 2
##  $ Prey    : num [1:16, 1:51] 0.01 -0.019018 0.024223 0.000046 ...
##  $ Predator: num [1:16, 1:51] 1.00e-02 9.31e-03 6.22e-05 -1.80e-02 ...
```

`LVres_morris` is a list of class `"ODEmorris"` with one element for each state variable (here, `Prey` and `Predator`). Those elements are matrices of $3 \cdot$ `length(LVpars)` $+ 1 = 16$ rows and `length(LVtimes)` $= 51$ columns. The first row contains a copy of all timepoints. The other rows contain the Morris sensitivity indices μ, μ^* and σ for all 5 parameters and all 51 timepoints. ODEsensitivity provides a `plot()` method for objects of class `"ODEmorris"`:

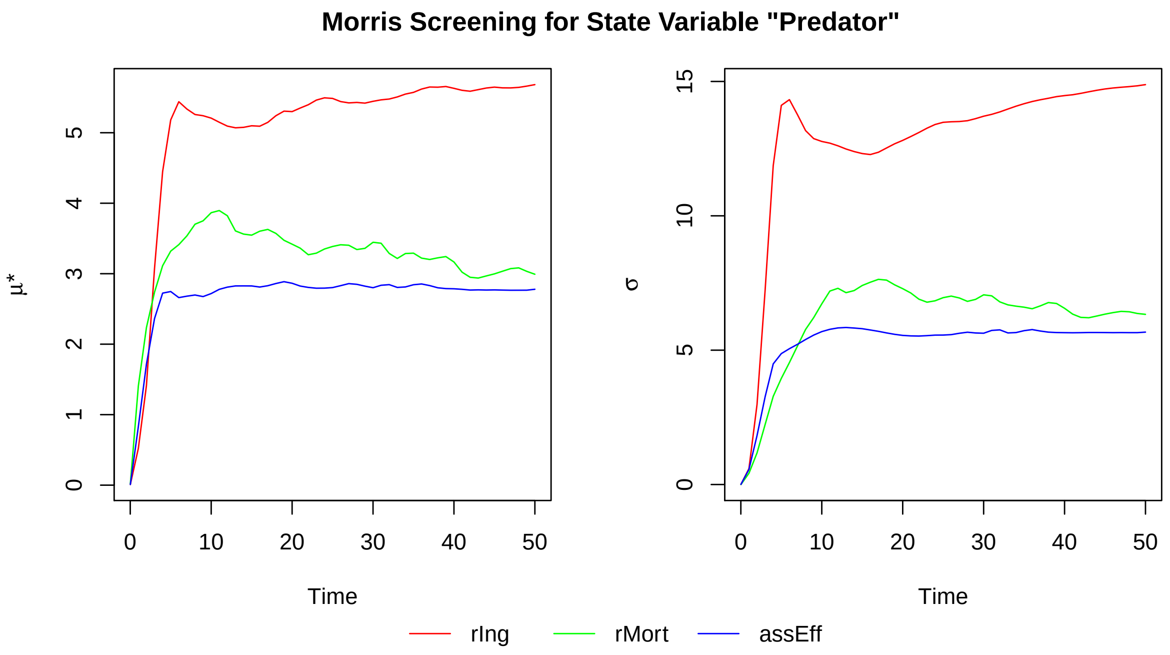
```
plot(LVres_morris)
```



As can be seen from the plot for μ^*, the predator's ingestion rate r_I has the largest overall influence on $P(t)$ for nearly the whole time from day 0 to day 50. r_I is closely followed by the predator's assimilation efficiency k_{AE}. At the beginning, the prey's growth rate r_G also has a large overall influence on $P(t)$, but this influence decreases with time and is exceeded by the overall influences of the predator's mortality rate r_M and the carrying capacity K (at about $t = 25$ and $t = 27$, respectively). Interpretations concerning the nonlinearity or interaction level of the influence of all parameters can be carried out analogously using the plot for σ.

`plot.ODEmorris()` has two important arguments: `pars_plot` and `state_plot`. Using `pars_plot`, a subset of the parameters included in the sensitivity analysis can be selected for plotting (the default is to use all parameters). `state_plot` gives the name of the state variable for which the sensitivity indices shall be plotted (the default being the first state variable):

```r
plot(LVres_morris, pars_plot = c("rIng", "rMort", "assEff"),
     state_plot = "Predator")
```



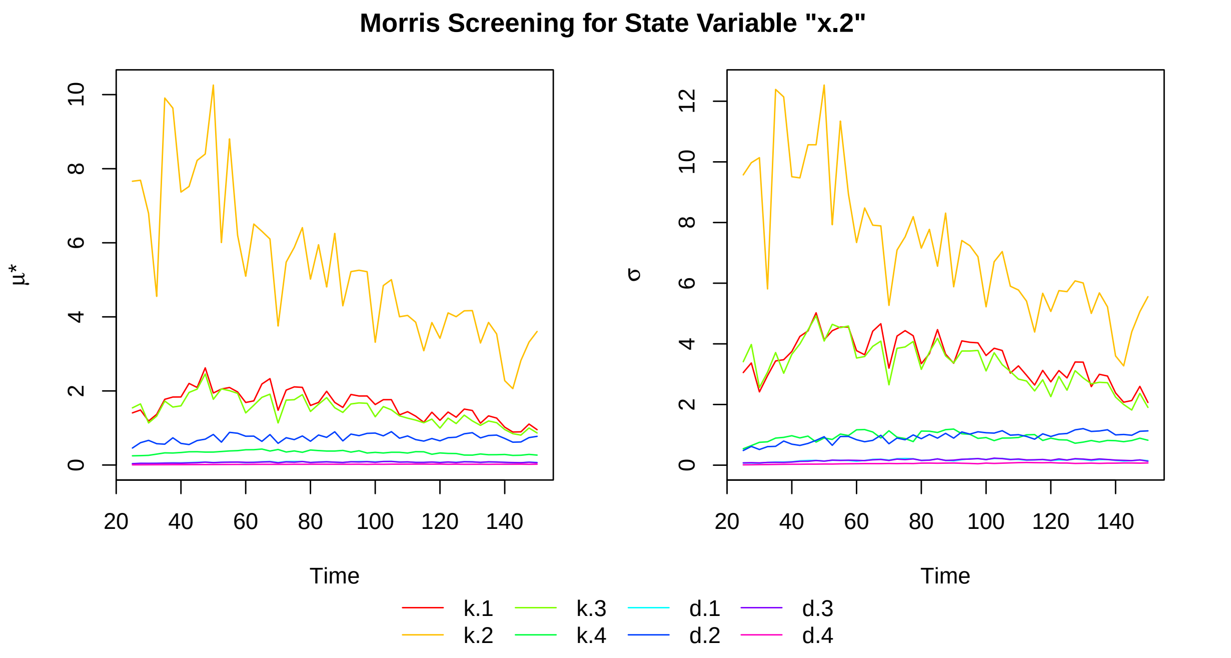
4.3.2 Low Frequency Oscillations

For the LFO example introduced in subsection 4.2, the generic `ODEmorris()` function can be used as well. If an object of class `"ODEnetwork"` is supplied for argument `mod`, `ODEmorris()` automatically passes on to the `ODEmorris.ODEnetwork()` method. In this case, argument `state_init` is not needed since the initial values of the eight state variables `x.1`, . . . , `x.4` and `v.1`, . . . , `v.4` are contained in `lfonet$state`. All other arguments correspond to those of `ODEmorris.default()`. In this example, `ode_method` is not needed because the ODE system is solved analytically. We again point out that for Morris screening, all parameters have to be assumed to be uniformly distributed.

```
set.seed(4628)
LFOres_morris <- ODEmorris(mod = lfonet,
                           pars = LFOpars,
                           times = LFOtimes,
                           binf = LFObinf,
                           bsup = LFObsup,
                           r = 500,
                           design = list(type = "oat",
                                         levels = 10, grid.jump = 1),
                           scale = TRUE,
                           parallel_eval = TRUE,
                           parallel_eval_ncores = parallel::detectCores())
```

We won't examine the structure of the output by `ODEmorris.ODEnetwork()` in detail since it parallels that of `ODEmorris.default()`: For each of the eight state variables, `LFOres_morris` (an object of class `"ODEmorris"`) contains a matrix with the three Morris sensitivity indices μ, μ^* and σ for all of the `length(LFOpars)` $= 8$ parameters and for all 51 timepoints. Thus, the `plot.ODEmorris()` method can be used here as well. We visualize the Morris sensitivity indices μ^* and σ for the deflection $x_2(t)$ of the second mass:

```
plot(LFOres_morris, state_plot = "x.2")
```



μ^* and σ seem to be very unstable when considering their courses over time. Nevertheless, the spring constant k_2 corresponding to the investigated mass $i = 2$ has the largest overall impact on $x_2(t)$ for all $t \in [25, 150]$, even though this impact decreases over time. k_2 is followed by two other spring constants: k_1 and k_3. The overall impact of d_2 and k_4 is very small and the damping constants d_1, d_3 and d_4 have nearly no influence on $x_2(t)$. Almost the same order of the parameters with respect to μ^* applies to σ.

4.4 Sobol' Sensitivity Analysis

4.4.1 General ODE Models

A Sobol' sensitivity analysis of ODE models can be performed using `ODEsobol()` from ODEsensitivity. Just like for `ODEmorris()`, the model function (in the manner as supplied for `ode()`), the names of the parameters, the initial values for the state variables and the timepoints of interest have to be supplied as arguments. However, the arguments following then are specific to `ODEsobol()`.

Firstly, `n` gives the sample size for the Monte Carlo estimation. Here, we use `n = 500` since the default (`n = 1000`) requires a much longer runtime. The precision of the estimators might suffer heavily, though, from reducing `n`.

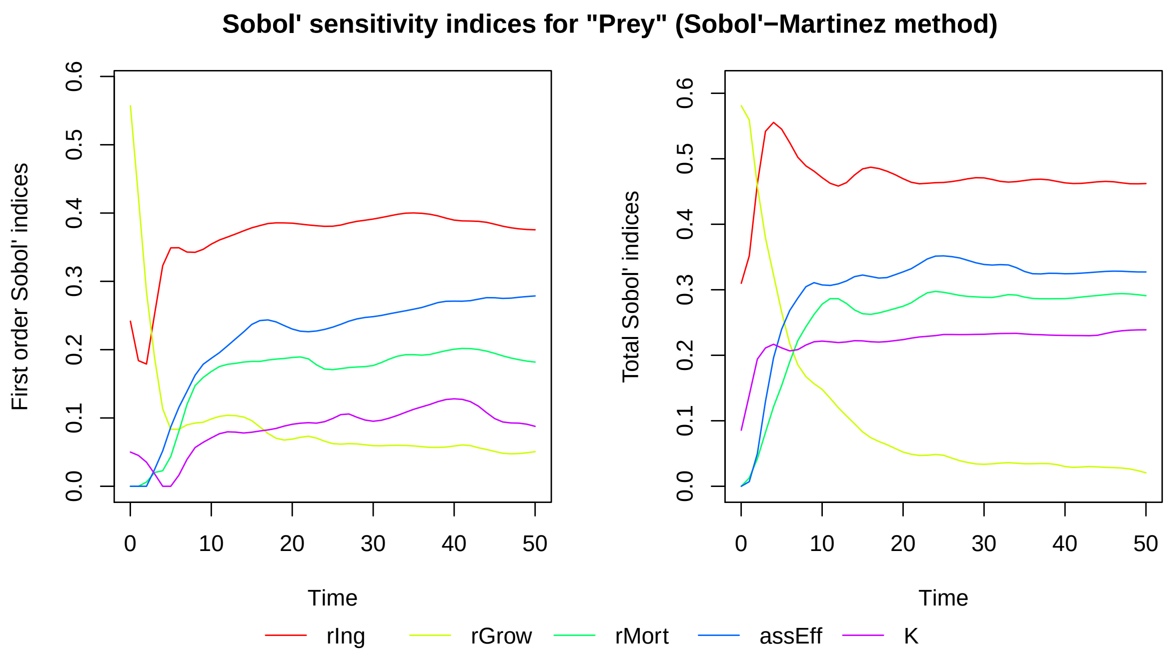
Secondly, via `rfuncs` and `rargs`, the user specifies the distribution of the parameters (by the name of the R function used to generate random numbers from that distribution) and the corresponding distributional parameters (as arguments for the corresponding R function named in `rfuncs`). If different distributions are needed, a vector containing the names of the corresponding R functions has to be supplied for `rfuncs`. The same holds analogously for `rargs` if different distributional parameters are needed. Here, we assume a uniform distribution for all parameters, but with differing lower and upper boundaries. Thus, we set `rfuncs = "runif"` and for `rargs`, we construct a vector of length $k = 5$ from `LVbinf` and `LVbsup`.

Thirdly, argument `sobol_method` indicates if the Sobol'-Jansen or the Sobol'-Martinez method shall be used for estimating the Sobol' sensitivity indices (see subsection 2.2). Results from these two methods shouldn't differ substantially, so it's mainly a question of runtime which method to choose (in tests, the Sobol'-Martinez method performed a little quicker, but not persistently). In this example, we use the Sobol'-Martinez method. The last three arguments `ode_method`, `parallel_eval` and `parallel_eval_ncores` are the same as for `ODEmorris.default()`.

```r
set.seed(59281)
LVres_sobol <- ODEsobol(mod = LVmod,
                        pars = LVpars,
                        state_init = LVinit,
                        times = LVtimes,
                        n = 500,
                        rfuncs = "runif",
                        rargs = paste0("min = ", LVbinf,
                                       ", max = ", LVbsup),
                                  sobol_method = "Martinez",
                                  ode_method = "lsoda",
                                  parallel_eval = TRUE,
                                  parallel_eval_ncores = parallel::detectCores())
str(LVres_sobol, vec.len = 3, give.attr = FALSE)

## List of 2
##  $ Prey    :List of 2
##   ..$ S: num [1:6, 1:51] 0.01 0.242 0.557 0 ...
##   ..$ T: num [1:6, 1:51] 1.00e-02 3.10e-01 5.81e-01 2.30e-06 ...
##  $ Predator:List of 2
##   ..$ S: num [1:6, 1:51] 0.01 0.1852 0.0166 0.636 ...
##   ..$ T: num [1:6, 1:51] 1.00e-02 2.07e-01 1.35e-05 6.31e-01 ...
```

The resulting object `LVres_sobol` (a list of class `"ODEsobol"`) contains one element for each state variable and in each of these elements two matrices: `S` for the first order indices and `T` for the total indices (for all 5 parameters and all 51 timepoints). Just like for Morris screening, ODEsensitivity provides a `plot()` method for objects of class `"ODEsobol"`:

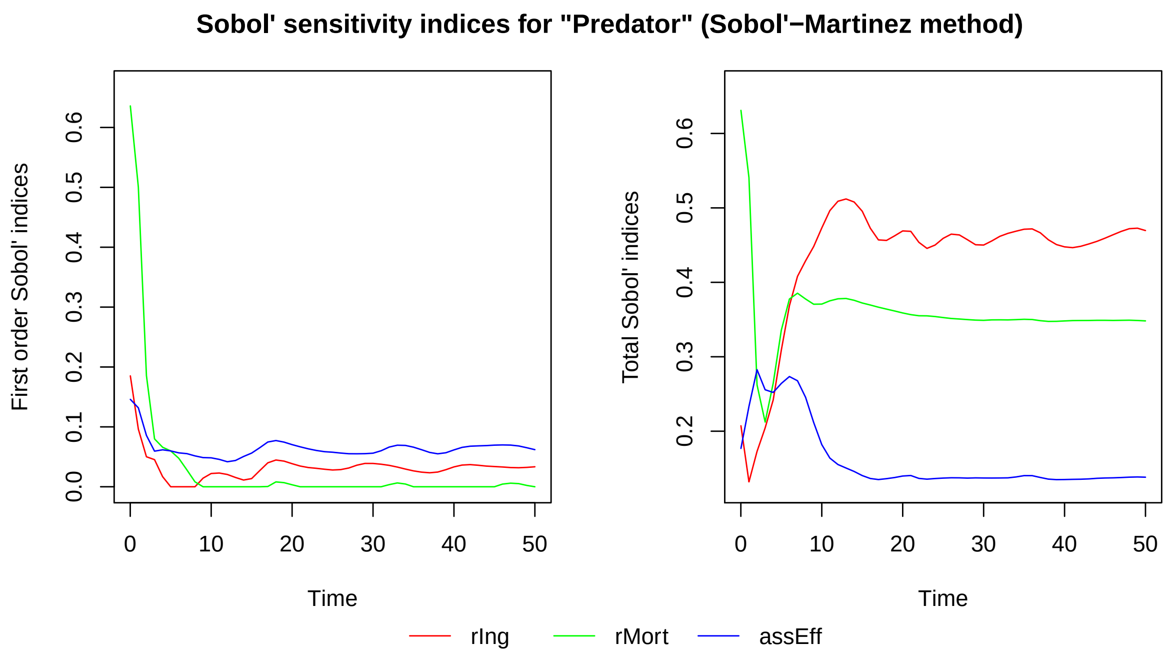
```
plot(LVres_sobol)
```



For the interpretation, we will focus on the *total* Sobol' indices since they are more similar to μ^* from Morris screening. For the prey's population $P(t)$, a clear order of the 5 parameters can be established (at least for $t \geq 10$): r_I has the largest total effect, followed by k_{AE}, r_M, K and r_G (in this order). A similar order had resulted from Morris screening, but we can observe quite evident differences between the plots of μ^* and the total Sobol' indices. However, a comparison of Sobol' and Morris sensitivity indices is rather difficult since they measure different types of "influence" of the input variables on the output variable (see section 5 for a more detailed explanation).

`plot.ODEsobol()` also supports the arguments `pars_plot` and `state_plot`:

```
plot(LVres_sobol, pars_plot = c("rIng", "rMort", "assEff"),
     state_plot = "Predator")
```



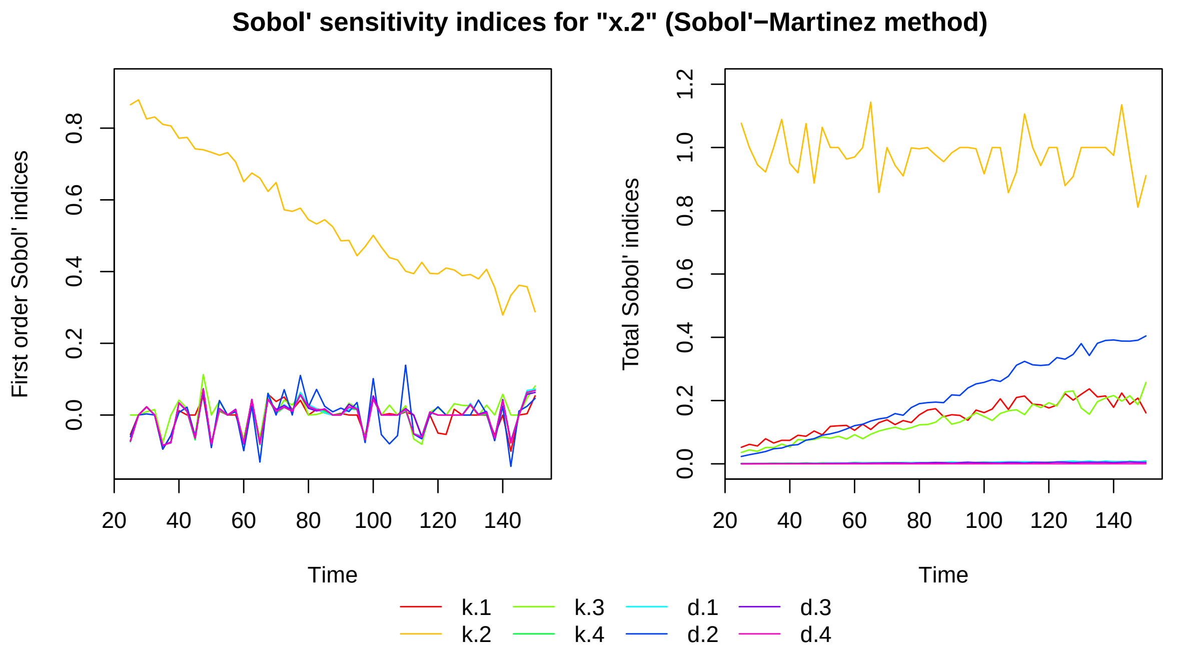
4.4.2 Low Frequency Oscillations

`ODEsobol.ODEnetwork()` supports almost the same arguments as `ODEsobol.default()`. Since we assume a uniform distribution for all parameters, the Sobol' sensitivity analysis of the LFO example from subsection 4.2 can be performed as follows:

```
set.seed(1739)
LFOres_sobol <- ODEsobol(mod = lfonet,
                         pars = LFOpars,
                         times = LFOtimes,
                         n = 500,
                         rfuncs = "runif",
                         rargs = paste0("min = ", LFObinf,
                                        ", max = ", LFObsup),
                         sobol_method = "Martinez",
                         parallel_eval = TRUE,
                         parallel_eval_ncores = parallel::detectCores())
```

The resulting object `LFOres_sobol` has a structure corresponding to that of `LVres_sobol`: It is a list of class `"ODEsobol"` and contains the matrices `S` and `T` for all state variables `x.1`, ..., `x.4`, `v.1`, ..., `v.4`. The warnings that are thrown here refer to a problem that typically occurs for small values of `n`: In this case, the Sobol' sensitivity indices estimated by Monte Carlo simulation might be out of $[0, 1]$. The plots below visualize this issue more clearly.

```
plot(LFOres_sobol, state_plot = "x.2")
```



We will focus again on the interpretation of the total Sobol' indices: k_2 has the biggest overall impact on $x_2(t)$ (for all $t \in [25, 150]$). This could also be observed for Morris screening, but with decreasing values of μ^* over time (the behaviour of μ^* corresponds rather to that of the *first order*

Sobol' index). For d_2, the overall influence seems to increase linearly over time and exceeds the influence of k_1 and k_3 at about $t = 60$. For k_1 and k_3, the overall impacts also seem to increase linearly, but with a smaller slope than for d_2. The parameters k_4, d_1, d_3 and d_4 don't seem to have any influence on $x_2(t)$ at all. Unfortunately, just like for Morris screening, the Sobol' sensitivity indices are very unstable over time, too. This might indicate that this problem is related rather to this ODE system than to the method of sensitivity analysis itself.

5 Conclusion

In this report, two methods – Morris screening and Sobol' sensitivity analysis – have been presented to analyze the sensitivity of a scalar-valued model function with regard to changes in its input variables. When applied to ODE models, the results from those two methods agree only roughly, though. A reason might be that the Morris and the Sobol' sensitivity indices measure different types of "influence" of the input variables on the output: The Morris sensitivity index μ_i measures the "mean slope" of the output function with regard to the i-th input variable, μ_i^* measures the "mean absolute slope" and σ_i the variability of the (non-absolute) slope. In contrast, the first order and total Sobol' sensitivity indices ($S_{\{i\}}$ and $S_{\{i\}}^T$, respectively) measure the *variability* of the output variable with regard to the i-th input variable (plus interactions, for $S_{\{i\}}^T$). Apart from that, it is always important to consider that all methods applied here rely on *randomness* (for generating the parameter combinations where the model function is evaluated). Thus, we can't expect results to coincide exactly anyway. However, in consideration of the fairly distinct deviations between the two methods, this doesn't seem to be the major cause.

We conclude that the sensitivity analysis of ODE models using Morris screening and the Sobol' method is rather explorative and results should be interpreted carefully.

5.1 Outlook

Further investigation might focus on the instability of the Morris and Sobol' sensitivity indices when analyzing LFOs. Especially, results from Morris screening using the simplex-based sampling design could be of interest. With regard to LFOs, the sensitivity of other network structures (not only circles) and networks of different sizes could be analyzed (e.g. $n = 1, \ldots, 5$ and $n = 30$ for the so-called "New England Test System" from Surmann et al., 2014). Additionally, such an analysis could also include other parameters, e.g. $k_{i,j}$ and $d_{i,j}$ for $i, j \in \{1, \ldots, n\}$ with $i \neq j$.

References

Campolongo, F., Cariboni, J., and Saltelli, A. (2007). “An effective screening design for sensitivity analysis of large models”. In: *Environmental modelling & software* 22.10, pages 1509–1518.

Confalonieri, R., Bellocchi, G., Bregaglio, S., Donatelli, M., and Acutis, M. (2010). “Comparison of sensitivity analysis techniques: a case study with the rice model WARM”. In: *Ecological Modelling* 221.16, pages 1897–1906.

Fruth, J. (2015). “New methods for the sensitivity analysis of black-box functions with an application to sheet metal forming”. Dissertation. TU Dortmund University.

Jansen, M. J. W. (1999). “Analysis of variance designs for model output”. In: *Computer Physics Communications* 117.1, pages 35–43.

Lotka, A. J. (1925). *Elements of Physical Biology.* 1st edition. Baltimore: Williams & Wilkins Co.

Martinez, J.-M. (2011). *Analyse de sensibilité globale par décomposition de la variance.* Presentation in the meeting of GdR Ondes and GdR MASCOT-NUM, January, 13th, 2011, Institut Henri Poincare, Paris, France.

Morris, M. D. (1991). “Factorial sampling plans for preliminary computational experiments”. In: *Technometrics* 33.2, pages 161–174.

Pujol, G. (2009). “Simplex-based screening designs for estimating metamodels”. In: *Reliability Engineering and System Safety* 94.7, pages 1156–1160.

Pujol, G., Iooss, B., Janon, A., Boumhaout, K., Da Veiga, S., Fruth, J., Gilquin, L., Guillaume, J., Le Gratiet, L., Lemaitre, P., Ramos, B., Touati, T., and Weber, F. (2016). *sensitivity: Global Sensitivity Analysis of Model Outputs.* Online: `https://CRAN.R-project.org/package=sensitivity`, R package version 1.12.1.

R Core Team (2016). *R: A Language and Environment for Statistical Computing.* Online: `http://www.r-project.org/`. Vienna, Austria.

Saltelli, A., Chan, K., and Scott, E. M. (2010). *Sensitivity Analysis.* 1st edition. New York: Wiley.

Sobol’, I. M. (1990). “On sensitivity estimation for nonlinear mathematical models”. In: *Matematicheskoe Modelirovanie* 2.1, pages 112–118.

Sobol’, I. M. (2001). “Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates”. In: *Mathematics and computers in simulation* 55.1, pages 271–280.

Soetaert, K., Petzoldt, T., and Setzer, R. W. (2010). “Solving Differential Equations in R: Package deSolve”. In: *Journal of Statistical Software* 33.9, pages 1–25.

Soetaert, K., Petzoldt, T., and Setzer, R. W. (2016). *deSolve: Solvers for Initial Value Problems of Differential Equations (ODE, DAE, DDE).* Online: `http://desolve.r-forge.r-project.org/`, R package version 1.13.

Surmann, D. (2015). *ODEnetwork: Network of Differential Equations.* Online: `https://CRAN.R-project.org/package=ODEnetwork`, R package version 1.2.

Surmann, D., Ligges, U., and Weihs, C. (2014). *Modelling Low Frequency Oscillations in an Electrical System.* Energycon 2014 Conference. Dubrovnik: Faculty of Statistics, TU Dortmund University.

Theers, S., Weber, F., and Surmann, D. (2016). *ODEsensitivity: Sensitivity Analysis of Ordinary Differential Equations.* Online: `https://CRAN.R-project.org/package=ODEsensitivity`, R package version 1.0.1.

Volterra, V. (1926). “Fluctuations in the abundance of a species considered mathematically”. In: *Nature* 118, pages 558–560.