SAVE: an R package for the Statistical Analysis of Computer Models

Jesus Palomo
URJC

Rui Paulo
ISEG and CEMAPRE

Gonzalo García-Donato
UCLM

Abstract

This paper introduces the R package **SAVE** which implements statistical methodology for the analysis of computer models. Namely, the package includes routines that perform emulation, calibration and validation of this type of models. The methodology is Bayesian and is essentially that of Bayarri, Berger, Paulo, Sacks, Cafeo, Cavendish, Lin, and Tu (2007). The package is available through the Comprehensive R Archive Network, CRAN. We illustrate its use with a real data example.

Keywords: R package, computer models, calibration, emulation, Bayesian analysis, Gaussian processes.

1. The analysis of complex computer models

Complex computer models are implementations of sophisticated mathematical models that aim at reproducing a particular real process. The R package **SAVE** (**S**tatistical **A**nalysis and **V**alidation **E**ngine) implements statistical methodology developed for the analysis of this type of models, which is based on Craig, Goldstein, Seheult, and Smith (1996), Kennedy and O'Hagan (2001), Kennedy, O'Hagan, and Higgins (2002), Higdon, Kennedy, Cavendish, Cafeo, and Ryne (2004), and most directly on Bayarri *et al.* (2007). The package is available through the Comprehensive R Archive Network, CRAN.

The following aspects of the statistical analysis of a computer model are addressed in **SAVE**:

- *Emulation.* A crucial characteristic of these models is that they are often computationally very demanding and a single run may take several minutes to complete. It is then important to produce fast approximations to the output of these models, and these approximations are referred to as emulators.

- *Calibration.* Computer models usually depend on a vector of unknown inputs that needs to be specified before the model can be run. Calibration refers to the process of determining estimates of these calibration parameters based on field observations of the real process.

- *Validation.* Ultimately, we want to assess the degree to which the computer model is an effective surrogate for the real process. We do so by producing predictions of reality and associated tolerance bounds, which measure the degree of the accuracy of said predictions.

Related R packages include **BACCO**, Hankin (2005), and the suite of **Dice** packages: **DiceKrigging** and **DiceOptim** (Roustant, Ginsbourger, and Deville 2012), **DiceEval** (Dupuy and Helbert 2013) and **DiceDesign** (Franco, Dupuy, Roustant, Damblin, and Iooss 2013). **DiceKrigging** computes estimates of Gaussian process parameters, and our package takes advantage of its functionalities, while **DiceOptim** is dedicated to optimimization of complex computer models based on krigging models. **DiceDesign** facilitates the construction of space-filling designs for computer experiments. **DiceEval** tackles the problem of validation but follows an approach that is quite different from ours; in particular, it is not Bayesian.

BACCO is a package that is similar to ours in its goals. However, it implements the methodology in Kennedy and O'Hagan (2001) which, although Bayesian, is distinct from that of Bayarri *et al.* (2007), the one we implement [**BACCO** also implements the methods in Kennedy and O'Hagan (2000), which is a topic we do not cover]. The most important distinction between **BACCO** and **SAVE** is that we explore the posterior distribution of the parameters of the statistical model using simulation-based techniques, namely Markov chain Monte Carlo, whereas **BACCO** relies either on analytical or on numerical integration. In that sense, with **SAVE** one can for instance explore the posterior distribution of calibration parameters, which often have a physical meaning, and take advantage of all the benefits that come with simulation-based inference. This will be illustrated in Section 4.

We should note that the **SAVE** package relies on C code to perform computer intensive calculations. Additionally, in order to maintain numerical stability as much as possible, we make from our own C code extensive calls to numerical routines written in Fortran, notably those available from BLAS and LAPACK.

The rest of this paper is organized as follows. In the next section we describe the statistical methodology establishing links with the package. In Section 3 we describe the structure of the package, and in Section 4 we illustrate its use in the context of a real example. Technical details are, whenever possible, relegated to the appendices.

2. Introducing the statistical framework

Denote the output of the computer model by $y^M(\boldsymbol{x}, \boldsymbol{u})$, where $\boldsymbol{x}$ is a vector of controllable inputs and $\boldsymbol{u}$ is a vector of unknown calibration and/or tuning parameters in the model. We have access to the data obtained by evaluating the computer model at a design set consisting of N points $D^M = \{(\boldsymbol{x}_1, \boldsymbol{u}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{u}_N)\}$. We denote by $\boldsymbol{y}^M$ the vector of model evaluations.

A preliminary but central question in the analysis is the construction of an emulator of the computer model, that is, a method to produce estimates of the output at untested configurations along with an associated measure of uncertainty. This is stage I of the analysis of a computer model. For this task, we follow the popular strategy (cf. Sacks, Welch, Mitchell, and Wynn (1989), Kennedy and O'Hagan (2000) and Bayarri *et al.* (2007)) of using a Gaussian process-based response-surface approximation to the model output. This approach results in that, conditional on $\boldsymbol{y}^M$ and on a set of parameters specifying the Gaussian process, $y^M(\cdot)$ follows a Gaussian process with mean and covariance functions which are available in closed form. The approach that **SAVE** currently implements for emulating $y^M(\cdot)$ estimates the unknown parameters, denoted by $\boldsymbol{\theta}^M$ and $\boldsymbol{\theta}^L$, by maximum likelihood, using the R package **DiceKrigging** (Roustant *et al.* 2012).

Analytic expressions for the mean and covariance functions can be found in Appendix A, along with a description of the parameters $\boldsymbol{\theta}^M$ and $\boldsymbol{\theta}^L$. In practical terms, the output of the computer model at a set of untested configurations (given $\boldsymbol{y}^M$ and the parameter estimates) follows a multivariate normal distribution with known mean vector and covariance matrix. The function `predictcode` returns iid draws from this multivariate distribution, along with its mean vector and covariance matrix.

Field data consists of noisy observations of the real process, possibly with replicates. To be more precise, we have a design set $D^F = \{\boldsymbol{x}_1^\star, \ldots, \boldsymbol{x}_n^\star\}$ and we observe

$$y^F(\boldsymbol{x}_j^*) = y^R(\boldsymbol{x}_j^*) + \varepsilon_j, \quad j = 1, \ldots, n$$

where $y^R(\cdot)$ represents the real process and ε_j are independent and identically distributed $N(0, 1/\lambda^F)$ random variables. Notice that the field data may contain replicates, that is, independent measurements of the experiment using the same configuration of the controllable inputs. We denote the set of field observations by $\boldsymbol{y}^F$.

Calibrating a model stands for finding estimates of the vector of calibration parameters based on field observations of the real phenomenon. This is achieved by postulating a statistical model relating the output of the model and the real phenomenon which introduces the notion of model discrepancy (cf. Craig, Goldstein, Seheult, and Smith (1997), Kennedy and O'Hagan (2001) and Goldstein (2010)), namely,

$$y^R(\boldsymbol{x}) = y^M(\boldsymbol{x}, \boldsymbol{u}^\star) + b(\boldsymbol{x}) \ , \tag{1}$$

where $b(\boldsymbol{x})$ stands for the bias or discrepancy function, and $\boldsymbol{u}^\star$ is the unknown value of the calibration vector which we are ultimately interested in estimating. For ease of notation we refer to $\boldsymbol{u}^\star$ simply as $\boldsymbol{u}$.

The approach that this package implements is (partially) Bayesian and therefore requires the specification of a prior for all the unknown quantities that appear in the statistical model, namely, $b(\cdot)$, $\boldsymbol{u}$, and λ^F. In line with Bayarri *et al.* (2007), we specify these priors in a fashion that requires very little input from the user. An exception is the prior on $\boldsymbol{u}$ which should reflect expert opinion about the calibration parameter. Details of the prior specification are available in Appendix B. Let $(\boldsymbol{u}, \lambda^b, \boldsymbol{\beta}^b, \lambda^F) \equiv (\boldsymbol{u}, \boldsymbol{\theta}^F)$ denote the vector of unknown parameters at this stage of the analysis, which we refer to as stage II. (The bias function $b(\cdot)$ gets a Gaussian process prior; λ^b denotes the precision and the vector $\boldsymbol{\beta}^b$ controls the correlation structure of this process.)

The posterior distribution is obtained using Markov chain Monte Carlo (MCMC) methods and is ultimately represented by a sample of correlated draws, which we denote by $\{\boldsymbol{u}_i, \boldsymbol{\theta}_i^F,\ i = 1, \ldots, M\}$. Details on the sampling method used are described in Appendix C. The **SAVE** function that implements this task is called `bayesfit`.

Once the posterior distribution of all the unknowns in the model is obtained, we can proceed to validate the computer model at D^V, a set of configurations for the controllable inputs. To do so, we must obtain draws from the distribution

$$\int f(y^M(D^V_{\boldsymbol{u}}), b(D^V) \mid \boldsymbol{y}^M, \boldsymbol{y}^F, \boldsymbol{u}, \boldsymbol{\theta}^F)\ \pi(\boldsymbol{u}, \boldsymbol{\theta}^F \mid \boldsymbol{y}^M, \boldsymbol{y}^F)\ d\boldsymbol{u}\ d\boldsymbol{\theta}^F \tag{2}$$

which are obtained by drawing the vectors $\boldsymbol{y}_i^M, \boldsymbol{b}_i$ from $f(y^M(D^V_{\boldsymbol{u}_i}), b(D^V) \mid \boldsymbol{y}^M, \boldsymbol{y}^F, \boldsymbol{u}_i, \boldsymbol{\theta}_i^F)$ for every $(\boldsymbol{u}_i, \boldsymbol{\theta}_i^F)$ in the previously constructed MCMC sample drawn from the posterior distribution. The **SAVE** function that performs this task is called `predictreality`. Above, the set $D^V_{\boldsymbol{u}}$ corresponds to the design that results from augmenting each of the configurations in D^V with the vector $\boldsymbol{u}$ for the calibration parameter. In general, we denote by $h(D)$ the vector that results from evaluating the function h at the elements of D.

Having obtained the sample $\{\boldsymbol{y}_i^M, \boldsymbol{b}_i,\ i = 1, \ldots, M\}$, we can compute several quantities which will aid us in the validation task (Bayarri *et al.* 2007):

- The bias-corrected prediction of the real process (i.e., reality) at D^V
$$\hat{\boldsymbol{y}}^R = \frac{1}{M}\sum_{i=1}^{M}(\boldsymbol{y}_i^M + \boldsymbol{b}_i)$$
- The tolerance bars measuring the accuracy of $\hat{\boldsymbol{y}}^R$ as a predictor of $y^R(D^V)$ are computed as follows: pick $\gamma \in (0,1)$; then, compute $\boldsymbol{\tau} = (\tau(\boldsymbol{x}) : \boldsymbol{x} \in D^V)$ such that $(1-\gamma)\times 100\%$ of the samples satisfy
$$|\hat{\boldsymbol{y}}^R - (\boldsymbol{y}_i^M + \boldsymbol{b}_i)| \leq \boldsymbol{\tau}\ ,$$
with the inequality interpreted in a component-wise fashion. We can then state that, for each $\boldsymbol{x} \in D_V$, $\Pr(|y^R(\boldsymbol{x}) - \hat{y}^R(\boldsymbol{x})| < \tau(\boldsymbol{x}) \mid \boldsymbol{y}^F, \boldsymbol{y}^M) = \gamma$.
- The pure-model prediction of reality at D^V is obtained by selecting an estimate of $\boldsymbol{u}$, $\hat{\boldsymbol{u}}$, say, which can be, for instance, its posterior mean or median. Then, output of the model at $D^V_{\hat{\boldsymbol{u}}}$ is computed by either actually running the model or by exercising the emulator. Function `predictcode` obtains draws from the emulator, but also returns the mean vector of the emulator, which can be used as an estimate of the output of the model. Denote this estimate by $\hat{\boldsymbol{y}}^M$.
- The tolerance bars measuring the accuracy of $\hat{\boldsymbol{y}}^M$ as a predictor of $y^R(D^V)$ are computed in a similar fashion: pick $\gamma \in (0,1)$; then, compute $\boldsymbol{\tau} = (\tau(\boldsymbol{x}) : \boldsymbol{x} \in D^V)$ such that $(1-\gamma)\times 100\%$ of the samples satisfy
$$|\hat{\boldsymbol{y}}^M - (\boldsymbol{y}_i^M + \boldsymbol{b}_i)| \leq \boldsymbol{\tau}$$
with the inequality interpreted in a component-wise fashion. We can then state that, for each $\boldsymbol{x} \in D_V$, $\Pr(|y^R(\boldsymbol{x}) - \hat{y}^M(\boldsymbol{x})| < \tau(\boldsymbol{x}) \mid \boldsymbol{y}^F, \boldsymbol{y}^M) = \gamma$.
- It is also possible to estimate the bias associated with the pure-model prediction, $\boldsymbol{b}_{\hat{\boldsymbol{u}}} = y^R(D^V) - \hat{\boldsymbol{y}}^M$: samples from its posterior predictive distribution can be obtained by computing $\{\boldsymbol{y}_i^M + \boldsymbol{b}_i - \hat{\boldsymbol{y}}^M\}$ so that a point estimate is $\hat{\boldsymbol{b}}_{\hat{\boldsymbol{u}}} = \hat{\boldsymbol{y}}^R - \hat{\boldsymbol{y}}^M$ and γ pointwise credible intervals can be determined by computing the associated $\gamma/2 \times 100\%$ and $(1-\gamma/2)\times 100\%$ sample quantiles.

The **SAVE** function that, given D^V and a posterior estimate of $\boldsymbol{u}$, computes the bias-corrected prediction, the pure-model prediction, associated tolerance bounds and estimated bias function is called `validate`.

3. An overview of SAVE

There are three high-level functions in **SAVE** which allow the user to perform all the tasks described in the previous section: `SAVE`, `bayesfit` and `validate`. In general,

- `SAVE` creates an object of the class `SAVE-class` and essentially sets up the problem by filling out a number of slots of this object. The data structures are set up (more on this below). Maximum likelihood calculations are performed using **DiceKriging** (Roustant *et al.* 2012). These calculations serve two purposes: fit the emulator of the computer model, and aid in the specification of the prior of $\boldsymbol{\theta}^F$ (cf. Appendix B).

- `bayesfit` produces a sample from the posterior distribution of the parameters λ^F, λ^b and $\boldsymbol{u}$ ($\boldsymbol{\beta}^b$ is fixed at an estimate throughout the analysis, cf. Appendix B). It takes as an argument an object of the class `SAVE-class` that has been created using `SAVE`, and returns a copy of this object but with several additional slots filled out. These slots pertain to the MCMC sample obtained.

- `validate` ultimately produces the bias-corrected prediction, the pure-model prediction, associated tolerance bounds and estimated bias function for any set of configurations for the controllable inputs and a posterior estimate of the vector of calibration inputs. It performs this task based on the information contained in an object of the class `SAVE-class` which has been produced by a call of the function `bayesfit`.

Two additional functions are available in **SAVE**, but these can be considered low-level routines. The function `predictcode` produces i.i.d. draws from the emulator evaluated at a set of design points, along with its mean vector and covariance matrix. It expects as an argument an object of the class `SAVE-class`. The function `predictreality` expects as an argument an object of the class `SAVE-class` which has been produced by the function `bayesfit`, i.e., containing an MCMC sample. It outputs draws from the distribution in (2) for a design set of configurations for the controllable inputs of the problem. These functions are internally called by `validate`, but can be utilized to further explore the problem.

The output of each of these functions can be appropriately summarized by customized calls to `print`, `summary`, `plot` and `show`.

The data is handled in the following way. **SAVE** assumes that there are two R data frames loaded: one containing all the field data, and another containing all the model data. The response and the input variables in the designs are identified by the `names` associated with each of these data frames, so they must be consistent. For illustrative purposes, consider the synthetic example in Figure 1. The data frame at the top of the figure contains the field data and the one at the bottom contains the model data. If we decide to analyze the problem where the response is the variable is `expand`; controllable inputs are `temp`, `press` and `weight`; and calibration inputs are `delta1` and `shift`, then we must call the function `SAVE` with arguments, `field.data = field`, `model.data = model`, `response.name = "expand"`, `controllable.names = c("temp", "press", "weight")`, `calibration.names = c("delta1", "shift")`. Notice that not all the columns present in the data frames are incorporated in this analysis.

4. An example

In this section we illustrate the use of the package with an analysis of a real example. It is the so-called spotweld example originally analyzed in Bayarri *et al.* (2007). We refer the interested reader to that paper for complete details on the application.

data frame called `field` containing the field data

	temp	press	weight	expand	height
1	35.1	2.65	600	84.1	5.1
2	35.1	2.75	600	90.6	9.2
3	35.1	2.65	600	80.4	6.1
⋮	⋮	⋮	⋮	⋮	⋮
n	30.3	2.65	700	83.4	7.1

data frame called `model` containing the model data

	temp	press	weight	delta1	shift	expand	delta2
1	23.2	2.12	629.1	0.22	-1.1	99.1	0.22
2	43.7	2.11	711.0	0.84	-0.9	70.1	0.83
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
N	34.4	2.12	700.1	0.49	-0.8	67.6	0.33

Figure 1: Synthetic example: the data frame at the top of the figure is called `field` and contains the field data; the data frame at the bottom of the figure is called `model` and contains the model data

In Figure 2 you can find a schematic representation of the spot welding process. Two sheets of metal of a particular thickness (`G`) are compressed by two electrodes under a certain applied load (`L`). Electric current of magnitude `C` is passed through said electrodes and the heat produced by the current flow causes the surfaces under pressure to melt. After cooling, a weld nugget is formed and as a result the two metal sheets are welded together. The scientists are interested in the diameter of this nugget (`N`).

Included in the package are two datasets, `spotweldfield` and `spotweldmodel` that pertain, respectively, to field experiments and computer model experiments associated with this problem. After loading the package (`library(SAVE)`), the dataframes can be loaded using the commands `data(spotweldfield,package="SAVE")` and `data(spotweldmodel,package="SAVE")`. Notice that the columns of the data frames are appropriately named, and that the computer model features an additional input, named `t`, which is a calibration input related to contact resistance.

We start this analysis with setting up the problem by creating `sw`— an object of the class `SAVE-class`— using the function `SAVE`:

```
R> sw <- SAVE(response.name="N", controllable.names=c("C", "L", "G"),
+             calibration.names=c("t"), field.data=spotweldfield,
+             model.data=spotweldmodel, mean.formula=as.formula("~1"),
+             bestguess=list(t=4.0))
```

Here we are specifying which columns correspond to the response and which correspond to the controllable and the calibration inputs. Additionally, we are also

- setting the mean function of the Gaussian process approximation to the output of the computer model as a constant (with the option `mean.formula=as.formula("~1")`) and
- providing an estimate of the vector of calibration inputs as a list (`bestguess`), which will be used in specifying the prior for $\boldsymbol{\theta}^F$ — see Appendix B for further details.

The object `sw` has now been created and several of its slots have been filled. The easiest way of accessing that information is by means of the command `summary(sw)`.

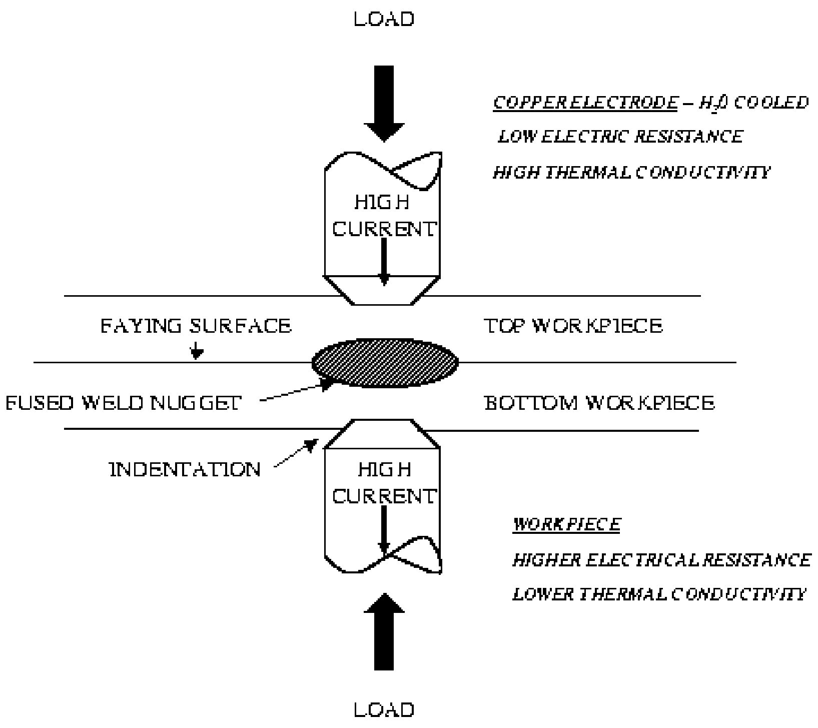

Figure 2: Schematic representation of the spotwelding process

Since at this point the emulator has been fitted, we could now use the function `predictcode` to predict the output of the computer model at a set of input configurations — more on this later. Instead, we will proceed to fit the Bayesian model (1) relating reality to computer model. To do so, we use the function `bayesfit`:

```
R> swbayes <- bayesfit(object=sw, prior=c(uniform("t", upper=8,
+                      lower=0.8)), n.iter=20000, n.burnin=100,
+                      n.thin=2)
```

We have created a new object — `swbayes` — but instead we could have updated the object we have just created, `sw`. Notice that we need to specify a prior for the calibration parameter `t`, and also options pertaining to the MCMC algorithm. We have set a uniform prior for the calibration parameter, 20000 iterations for the MCMC with a burn-in period of 100 iterations, and a thinning of 2. Other options were left at the corresponding defaults — cf Appendix C for additional details.

The object `swbayes` now contains not only the estimates previously computed using `SAVE` but also a sample from the posterior distribution of the calibration parameter `t` and of the field and bias precisions, λ^F and λ^b, respectively. To display this information we can again resort to the command `summary(swbayes)`. We can however also plot the samples obtained:

- `plot(swbayes, option="trace")` will give us the traceplots;

- `plot(swbayes, option="calibration")` will produce an histogram of the posterior samples of the calibration parameters and corresponding priors: see Figure 3; and,
- `plot(swbayes, option="precision")` will plot histograms of the posterior samples of the field and bias precisions. These histograms also include a plot of the prior and of the estimates that are used in constructing the prior, see Figure 4.

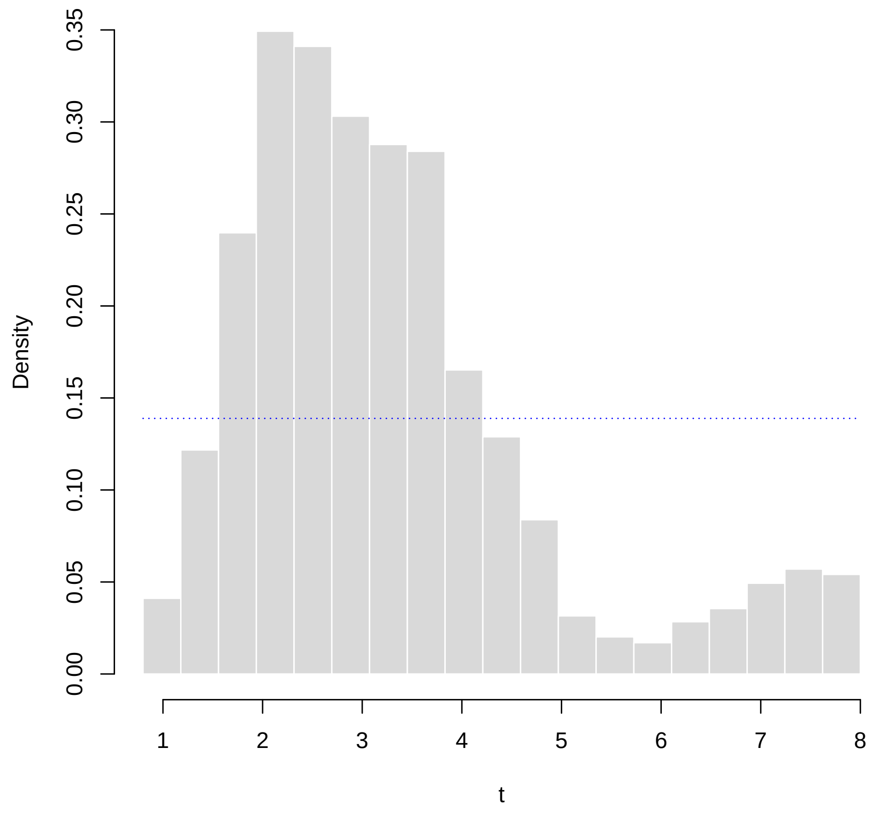

Figure 3: Posterior distribution of the calibration parameter `t`. The dotted line corresponds to the prior used.

Additionally, we can of course access the raw data. By running the command `slotNames(swbayes)` we get a description of the names of all the slots of the object `swbayes`, and it's then clear how to obtain the MCMC samples: `swbayes@mcmcsamples`.

After fitting the Bayesian model, we can finally produce predictions of reality and also assess the quality of pure-model predictions of reality. The package **SAVE** provides a very convenient function that performs all these calculations. To illustrate its use in the present example, please consider the following R code:

```
R> load <- c(4.0,5.3); curr <- seq(from=20,to=30,length=20); g <- c(1,2)
R> xnew <- as.data.frame(expand.grid(curr,load,g))
```

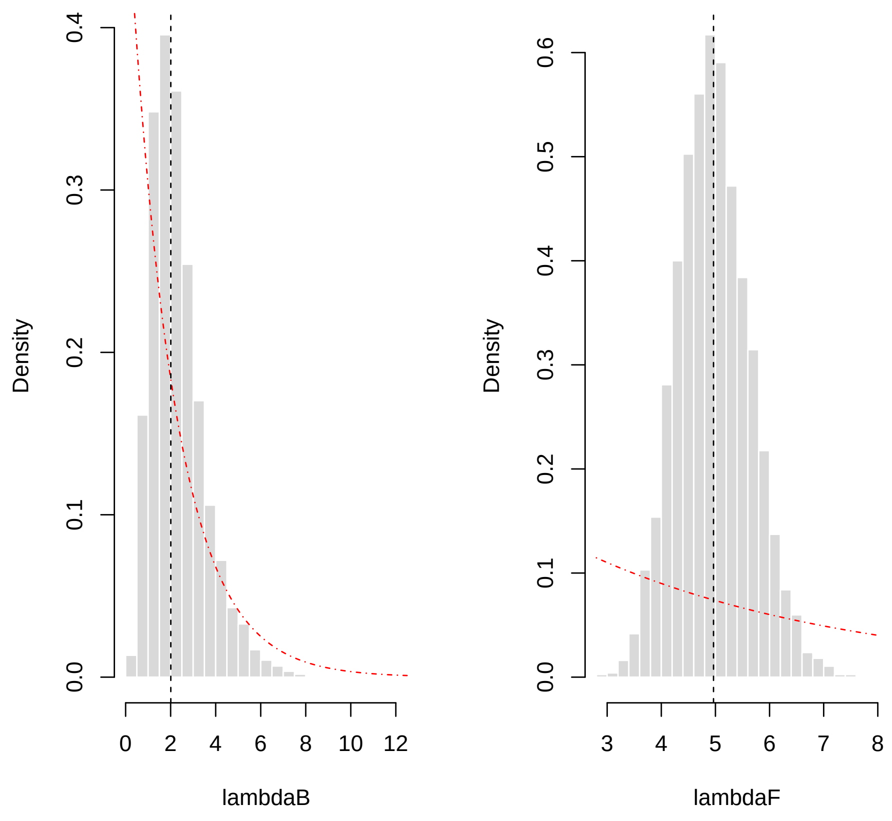

Figure 4: Posterior distribution of the precisions. The dashed vertical lines indicate the estimates of the parameters; the dash-dot lines indicate the priors used.

```
R> names(xnew)<-c("C","L","G")
R>
R> valsw <- validate(object=swbayes,newdesign=xnew,
+                        calibration.value="mean",n.burnin=100)
```

We first construct the design of controllable inputs at which we want to predict reality. For four combinations of load and thickness of the metal plates, we want to predict the weld diameter as a function of current. Regarding the pure-model prediction, we are setting the calibration parameters at the corresponding posterior mean. The resulting object `valsw` contains a slot named `validate` where a matrix is stored. This matrix contains as columns the pure-model prediction of reality (`pure.model`) and associated tolerance bound (`tau.pm`); the estimate of the bias associated with the pure-model prediction and pointwise credible interval for that unknown (`bias.Lower` and `bias.Upper`); the bias-corrected prediction of reality (`bias.corrected`) and associated tolerance bounds (`tau.bc`). This information can be accessed using `summary(valsw)` but can also be plotted (`plot(valsw)`) — you can find the plot in Figure 5. Depending on the problem, this default plot will not always be the most appropriate way of displaying the estimates. Nevertheless, since we have access to the estimates, we can certainly construct customized plots for any problem at hand. As an example, in Figure 6 we can find a plot of the pure-model prediction and associated tolerance bounds as a function of current for the 4 different combinations of load and thickness. The circles correspond to the appropriate field observations.

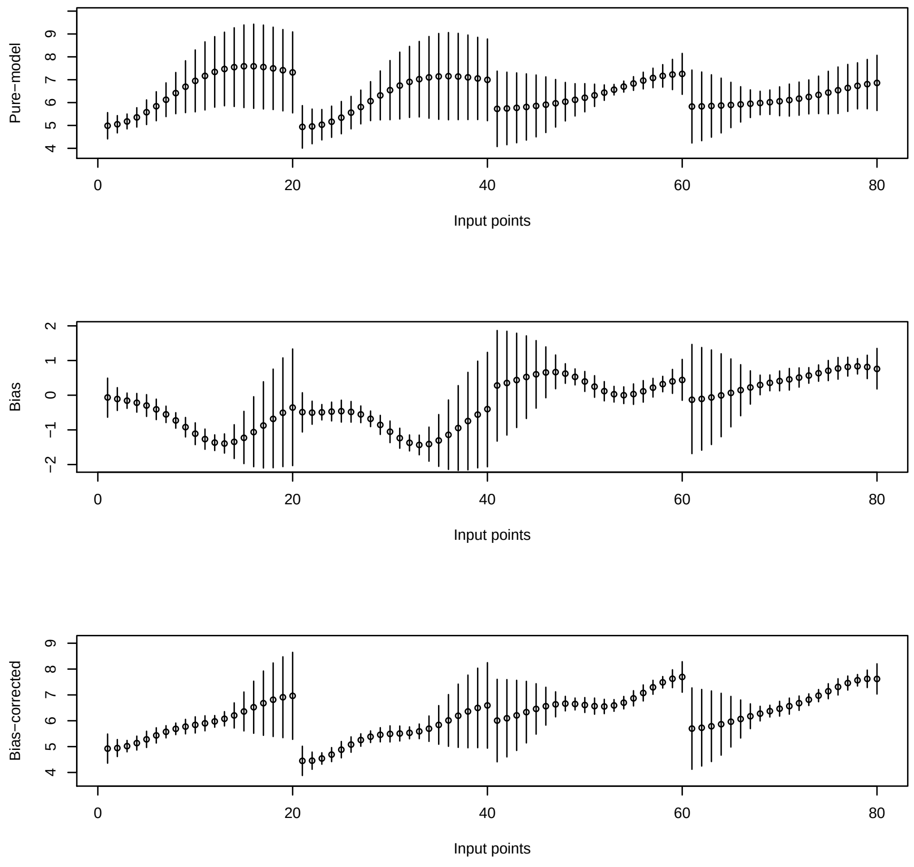

Figure 5: Default validation plot. The plot at the top contains the pure-model predictions (circles) at each of the input configurations in `xnew` and associated 90% tolerance bounds. The middle plot depicts an estimate of the bias of these pure-model estimates. The bottom plot contains the bias-corrected prediction (circles) and associated 90% tolerance bounds.

As stated above, `validate` is a function which calls two low-level functions, `predictcode` and `predictreality`. There may be situations where one must directly call these functions. We illustrate this in what follows.

Imagine that one is interested in understanding how the nugget diameter `N` varies with the current `C`. One might assess this variation by looking at the derivative of `N` with respect to `C`. Let's do that for `G`=1 and `L`=4. We start by predicting reality at an equally-spaced grid of `C` values between 20 and 30:

```
R> load <- 4; g <- 1; curr <- seq(from=20, to=30, length=80)
R> xnew <- expand.grid(curr,load,g)
```

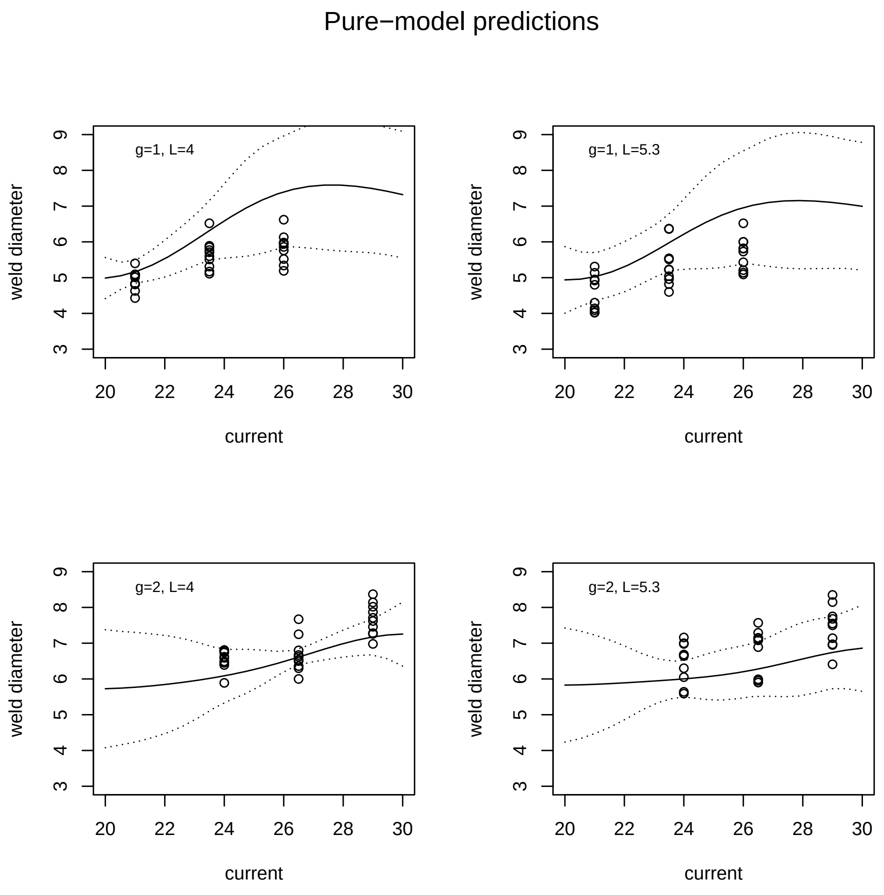

Figure 6: Customized validation plot. Here we can see the pure-model for 4 different combinations of load and thickness as a function of current. The dotted lines indicate the 90% tolerance bounds and the circles represent the observed field data correspoding to that particular combination of controllable inputs.

```
R> names(xnew) <- c("C","L","G")
R>
R> prsw <- predictreality(object=swbayes, newdesign=xnew)
```

The object `prsw` has slots named `modelpred` and `biaspred` where the draws from (2), which we have denoted in Section 2 by $\boldsymbol{y}_i^M, \boldsymbol{b}_i$, are stored. We now obtain the corresponding derivatives with respect to current of each of these draws, to finally obtain draws from the corresponding derivative of reality:

```
R> delta <- diff(curr)[1]
R> model <- prsw@modelpred
R> dmodel <- diff(t(model))/delta
R> bias <- prsw@biaspred
R> dbias <- diff(t(bias))/delta
R>
```

```
R> dreal <- dmodel + dbias
```

These draws can be summarized by computing the corresponding mean and tolerance bounds as explained in Section 2. This is plotted in Figure 7.

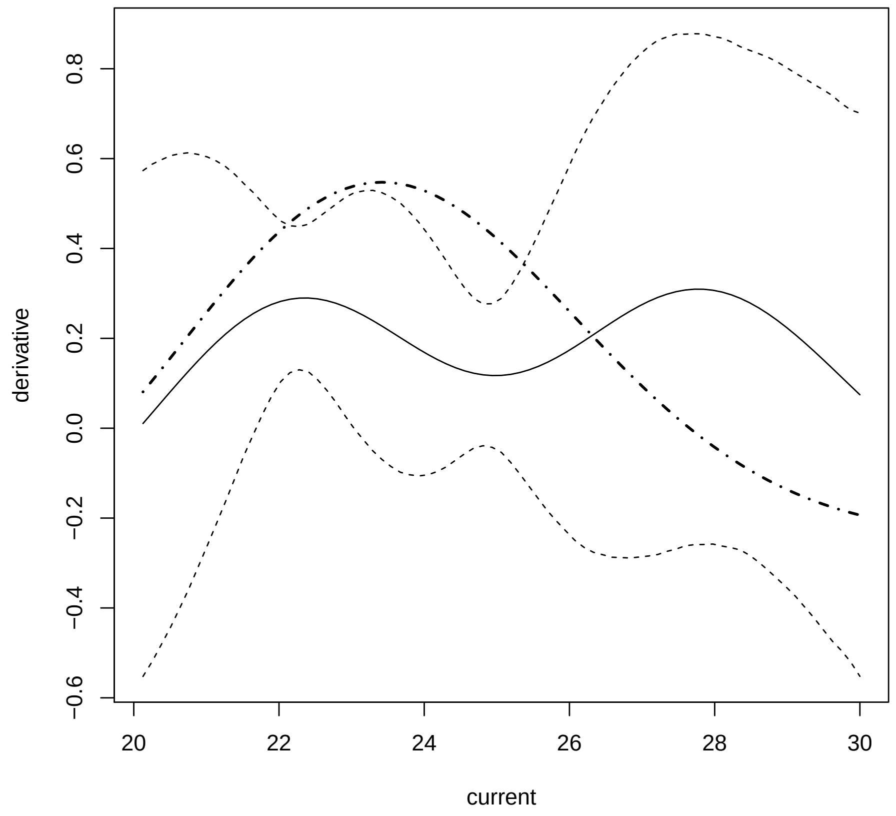

Figure 7: Bias-corrected prediction of the derivative of `N` with respect to `C` — the estimate is the solid line; the dashed lines are the 90% tolerance bounds. The pure-model prediction is the dash-dotted line.

Additionally, we can use `predictcode` to obtain draws from the emulator evaluated at a posterior estimate of the calibration input, `t`. This allows us to produce the pure-model estimate of the derivative:

```
R> u <- 3.2; load <- 4; g <- 1;
R> xnewpure <- expand.grid(curr,load,g,u)
R> names(xnewpure) <- c("C","L","G","t")
R>
R> pmsw <- predictcode(object=swbayes,newdesign=xnewpure,n.iter=20000)
R>
R> puremodel <- pmsw@samples
R> dpuremodel <- diff(t(puremodel))/delta
```

The mean of the samples in `dpuremodel` is the pure-model prediction of the derivative. This is plotted in Figure 7. Notice how the two estimates, pure-model and bias-corrected, are in this case even qualitatively different.

Acknowledgements

This research was funded by Portuguese National Funds through FCT - Fundação para a Ciência e a Tecnologia, project PTDC/MAT/105349/2008 and by the Spanish Ministry of Education and Science under grant MTM2010-19528. An initial version of this software was produced while the authors were postdoctoral researchers at the National Institute of Statistical Sciences (NISS), and at the times their research was supported in part by grants from General Motors and the National Science Foundation (Grant DMS-0073952). M.J. Bayarri (Valencia), James O. Berger (Duke), Jerry Sacks (NISS) were actively involved in that research, and so were J.A. Cafeo, J. Cavendish, C.H. Lin and J. Tu from General Motors.

A. Details on the emulator

We assume that *a priori* $y^M(\cdot)$ follows a stationary Gaussian process with mean and covariance functions governed by unknown parameters $\boldsymbol{\theta}^L$ and $\boldsymbol{\theta}^M = (\lambda^M, \boldsymbol{\alpha}^M, \boldsymbol{\beta}^M)$, respectively. The mean function of the Gaussian process is assumed to be of the form $\boldsymbol{\Psi}'(\cdot)\boldsymbol{\theta}^L$ where $\boldsymbol{\Psi}(\boldsymbol{z})$ is a specified $k \times 1$ vector function of the input $\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{u})$ and $\boldsymbol{\theta}^L$ is a $k \times 1$ vector of unknown regression parameters. This mean function is specified through the argument `mean.formula` of the `SAVE` function. Note that one of the restrictions of **SAVE** is that $\boldsymbol{\Psi}$ can only be function of $\boldsymbol{x}$, the vector of controllable inputs.

The parameter λ^M is the precision (the inverse of the variance) of the Gaussian process and the other parameters $(\boldsymbol{\alpha}^M, \boldsymbol{\beta}^M)$ control the correlation function of the Gaussian process, which we assume to be of the form

$$c^M(\boldsymbol{z}, \boldsymbol{z}^\star) = \exp\left(-\sum_{j=1}^{d} \beta_j^M |z_j - z_j{}^\star|^{\alpha_j^M}\right).$$

Here, d is the number of coordinates in $\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{u})$, the α_j^M are numbers between 0 and 2, and the β_j^M are positive parameters.

After observing $\boldsymbol{y}^M$, the conditional posterior distribution of y^M given the hyperparameters, $f(y^M(\cdot) \mid \boldsymbol{y}^M, \boldsymbol{\theta}^L, \boldsymbol{\theta}^M)$, is a Gaussian process with updated mean and covariance functions given respectively

$$\begin{aligned}\mathsf{E}[y^M(\boldsymbol{z}) \mid \boldsymbol{y}^M, \boldsymbol{\theta}^L, \boldsymbol{\theta}^M] &= \boldsymbol{\Psi}'(\boldsymbol{z})\,\boldsymbol{\theta}^L + \boldsymbol{r}_z{}'(\boldsymbol{\Gamma}^M)^{-1}(\boldsymbol{y}^M - \boldsymbol{X}\boldsymbol{\theta}^L)\\ \mathsf{COV}[\,y^M(\boldsymbol{z}), y^M(\boldsymbol{z}^\star) \mid \boldsymbol{y}^M, \boldsymbol{\theta}^L, \boldsymbol{\theta}^M] &= \frac{1}{\lambda^M}\, c^M(\boldsymbol{z}, \boldsymbol{z}^\star) - \boldsymbol{r}_z{}'(\boldsymbol{\Gamma}^M)^{-1}\boldsymbol{r}_{z^\star},\end{aligned}$$

where $\boldsymbol{r}_z{}' = \frac{1}{\lambda^M}\,(c^M(\boldsymbol{z}, \boldsymbol{z}_1), \ldots, c^M(\boldsymbol{z}, \boldsymbol{z}_N))$, $\boldsymbol{\Gamma}^M$ is given above and $\boldsymbol{X}$ is the matrix with rows $\boldsymbol{\Psi}'(\boldsymbol{z}_1), \ldots, \boldsymbol{\Psi}'(\boldsymbol{z}_N)$.

To obtain an emulator for y^M, we replace in the formulae above the unknown parameter values by the corresponding maximum likelihood estimates.

B. Details on the stage II prior

The stage II unknowns are $b(\cdot)$, the bias function, $\boldsymbol{u}$, the vector of calibration parameters, and λ^F, the precision of the field measurement error. The prior for $\boldsymbol{u}$ is specified using expert knowledge. Currently, the distributional choices are limited to uniform and normal, this last one truncated to an interval. This enters the function `bayesfit` through argument `prior`.

The prior for the bias is a stationary zero-mean Gaussian process with covariance λ^b and correlation function given by

$$c^b(\boldsymbol{x}, \boldsymbol{x}^\star) = \exp\left(-\sum_{j=1}^{p} \beta_j^b |x_j - x_j{}^\star|^2\right).$$

Here, p is the number of coordinates in $\boldsymbol{x}$, and the β_j^b are positive parameters. Let $\boldsymbol{\beta}^b = (\beta_1^b, \ldots, \beta_p^b)$. We need to specify a prior for $\boldsymbol{\theta}^F = (\lambda^b,\ \boldsymbol{\beta}^b,\ \lambda^F)$, and we do so in a nearly automatic fashion as follows: we start by selecting a best guess for the vector of calibration parameters, denoted by $\tilde{\boldsymbol{u}}$, which is the argument `bestguess` is function `SAVE`. Then, using the emulator, we predict the output of the computer model at $D^F_{\tilde{\boldsymbol{u}}}$, denoted $y^M(D^F_{\tilde{\boldsymbol{u}}})$. Next, treat $\boldsymbol{y}^F - y^M(D^F_{\tilde{\boldsymbol{u}}})$ as a realization of a Gaussian process with a nugget, namely as a realization of a multivariate normal with mean zero and covariance matrix $c^b(D^F)/\lambda^b + I/\lambda^F$ to get maximum likelihood estimates $\hat{\lambda}^b$, $\hat{\boldsymbol{\beta}}^b$, $\hat{\lambda}^F$. Then,

- $\boldsymbol{\beta}^b$ is fixed throughout the analysis at $\hat{\boldsymbol{\beta}}^b$;
- λ^b and λ^F are independent exponentially distributed quantities centered at a multiple of the corresponding estimates, $\hat{\lambda}^b$ and $\hat{\lambda}^F$. This multiple is set through the parameter `mcmcMultmle` in `bayesfit` and its purpose is to allow the user to specify a prior which is relatively flat in the region where the posterior distribution accumulates.

The function `SAVE` computes these maximum likelihood estimates (with the help of the package **DiceKriging**, Roustant *et al.* (2012)) and stores these in the slot `mle` of the corresponding object of class `SAVE-class`.

C. Details on the MCMC

Full details on the sampling mechanism can be found in Bayarri *et al.* (2007). The algorithm implemented in **SAVE** requires very little input apart from the necessary length of the simulation, burn-in and thinning numbers. This is because all unknowns are sampled directly from their full conditionals with the exception of the vector of calibration parameters. This vector is sampled using a Metropolis-Hastings step, for which the user needs to decide on three aspects:

- The proposal distribution is a mixture between the prior and a local move. The user needs to specify the probability of sampling from the prior, which is argument `prob.prop` of the `bayesfit` function;
- The algorithm performs a fixed number of Metropolis-Hastings steps before deciding on a move; the user must set this number, and this is argument `nMH` of the `bayesfit` function;

- The package implements two alternative methods: `method=2` specifies that computer model and bias are analytically integrated out before sampling $\boldsymbol{u}$; this is the default and preferred method. If `method=1`, these vectors are not integrated out.

References

Bayarri MJ, Berger JO, Paulo R, Sacks J, Cafeo JA, Cavendish J, Lin CH, Tu J (2007). "A Framework for Validation of Computer Models." *Technometrics*, **49**, 138–154.

Craig P, Goldstein M, Seheult A, Smith J (1996). "Bayes linear strategies for history matching of hydrocarbon reservoirs." In JM Bernardo, JO Berger, AP Dawid, D Heckerman, AFM Smith (eds.), *Bayesian Statistics 5*. Oxford University Press: London. (with discussion).

Craig PS, Goldstein M, Seheult AH, Smith JA (1997). "Pressure matching for hydrocarbon reservoirs: a case study in the use of Bayes linear strategies for large computer experiments." In C Gatsonis, JS Hodges, RE Kass, R McCulloch, P Rossi, ND Singpurwalla (eds.), *Case Studies in Bayesian Statistics*, volume III, pp. 36–93.

Dupuy D, Helbert C (2013). ***DiceEval**: Construction and evaluation of metamodels.* R package version 1.2, URL http://CRAN.R-project.org/package=DiceEval.

Franco J, Dupuy D, Roustant O, Damblin G, Iooss B (2013). ***DiceDesign**: Designs of Computer Experiments.* R package version 1.3, URL http://CRAN.R-project.org/package=DiceDesign.

Goldstein M (2010). "External Bayesian analysis of computer simulators." In JM Bernardo, MJ Bayarri, JO Berger, AP Dawid, D Heckerman, AFM Smith, M West (eds.), *Bayesian Statistics 9*. Oxford University Press: London. (with discussion).

Hankin RKS (2005). "Introducing BACCO, an R bundle for Bayesian Analysis of Computer Code Output." *Journal of Statistical Software*, **14**.

Higdon D, Kennedy MC, Cavendish J, Cafeo J, Ryne RD (2004). "Combining field data and computer simulations for calibration and prediction." *SIAM Journal on Scientific Computing*, **26**, 448–466.

Kennedy MC, O'Hagan A (2000). "Predicting the output from a complex computer code when fast approximations are available." *Biometrika*, **87**(1), 1–13.

Kennedy MC, O'Hagan A (2001). "Bayesian calibration of computer models (with discussion)." *Journal of the Royal Statistical Society B*, **63**, 425–464.

Kennedy MC, O'Hagan A, Higgins N (2002). "Bayesian analysis of computer code outputs." In CW Anderson, V Barnett, PC Chatwin, AH El-Shaarawi (eds.), *Quantitative Methods for Current Environmental Issues.*, pp. 227–243. Springer-Verlag: London.

Roustant O, Ginsbourger D, Deville Y (2012). "DiceKriging, DiceOptim: Two R Packages for the Analysis of Computer Experiments by Kriging-Based Metamodeling and Optimization." *Journal of Statistical Software*, **51**(1), 1–55. URL http://www.jstatsoft.org/v51/i01/.

Sacks J, Welch WJ, Mitchell TJ, Wynn HP (1989). "Design and analysis of computer experiments (C/R: p423-435)." *Statistical Science*, **4**, 409–423.

Affiliation:

Jesus Palomo
Department of Business Administration (Finance)
Rey Juan Carlos University
28032 Madrid, Spain
Email: jesus.palomo@urjc.es

Rui Paulo
Department of Mathematics and CEMAPRE
ISEG, Technical University of Lisbon
1200 Lisboa, Portugal
Email: rui@iseg.utl.pt

Gonzalo García-Donato
Department of Economics and Finance
Universidad de Castilla-La Mancha
02071, Albacete, Spain
Email: gonzalo.garciadonato@uclm.es