Optimization for Maximum-Likelihood Estimation in kergp

Yves Deville*

September 23, 2019

This report[1] is devoted to the use of numerical optimization for Maximum Likelihood Estimation in the **kergp** package. It gives some indications on the packages and functions used for optimization. Is also explains how the optimization can be tuned.

1 GP models and "gp" objects

1.1 GP models

The package **kergp** is dedicated to Gaussian Processes (GP) regression models; These relate a continuous response variable Y to a vector $\mathbf{x}$ of d *inputs*, usually assumed to be in a domain $\mathcal{X}$ of the Euclidean d-dimensional space $\mathbb{R}^d$. A model corresponds to partial observations having the form

$$y_i = \underbrace{\mathbf{f}(\mathbf{x}_i)^\top \boldsymbol{\beta}}_{\text{trend}} + \underbrace{\zeta(\mathbf{x}_i)}_{\text{GP}} + \underbrace{\varepsilon_i}_{\text{noise}}, \qquad i = 1, \ldots, n \tag{1}$$

where the three terms at right-hand side are as follows.

- *Trend.* The trend function $\mathbf{f}(\mathbf{x})$ is vector-valued, and $\boldsymbol{\beta}$ is a vector of p trend parameters.
- *GP.* $\zeta(\mathbf{x})$ is an unobserved smooth GP[2] with mean zero and with its covariance kernel $K(\mathbf{x}, \mathbf{x}'; \boldsymbol{\theta})$ depending on a vector $\boldsymbol{\theta}$ of covariance parameters;
- *Noise.* The n r.vs ε_i are assumed to be i.i.d., normal, centered, and to have a common variance σ^2_ε. These noise r.vs are assumed to be independent of the GP $\zeta(\mathbf{x})$.

Matrix notations are convenient for (1). Let $\mathbf{X}$ be the $n \times d$ *input* (or *design*) matrix with the transposed input vectors $\mathbf{x}_i^\top$ as it rows, and let $\mathbf{F}$ be the $n \times p$ matrix with rows $\mathbf{f}(\mathbf{x}_i)^\top$. Then (1) writes as

$$\mathbf{y} = \mathbf{F}\boldsymbol{\beta} + \boldsymbol{\zeta} + \boldsymbol{\varepsilon}. \tag{2}$$

The notation $\mathbf{K}(\mathbf{X}; \boldsymbol{\theta})$ or simply $\mathbf{K}(\mathbf{X})$ will be used for the $n \times n$ covariance matrix $[K(\mathbf{x}_i, \mathbf{x}_j; \boldsymbol{\theta})]_{i,j}$.

1.2 gp objects

Class and creator

A GP model as described above will be represented by an object with S3 class `"gp"`, which will be created by using the `gp` "creator" function. Figure 1 shows a call which is nearly minimal – the argument `noise` still could have been omitted.

*deville.yves@alpestat.com

[1] Compiled with **R** 3.6.1 using **kergp** 0.5.0.

[2] With continuous paths.

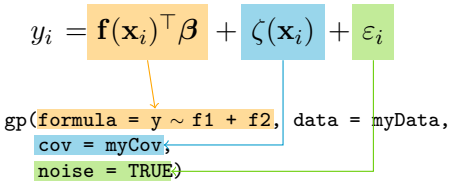

Figure 1: Terms in the model and formal arguments of gp in a (minimal) illustrative call.

The classical situation is when all parameters $\boldsymbol{\beta}$, $\boldsymbol{\theta}$ and σ_ε^2 are unknown and must be estimated from the n observations y_i, which is done in the creation step. For now, **kergp** favours Maximum Likelihood (ML) estimation. Note that the first and the third terms at the right hand side of (1) can be removed from the model by using a suitable choice of the formal arguments of gp, see below. However the GP term must exist.

For illustration we will use a deterministic function to generate the "true" response and add a noise term with rather small variance, see Figure 2.

```
library(kergp)
n <- 100
x <- seq(from = 0.0, to = 1.0, length.out = n)
set.seed(123)
y <- 1 + sin(2.0 * pi * x) + rnorm(n, sd = 0.2)
myData <- data.frame(x = x, y = y)
```

In order to define a gp we first need a parameterized kernel.

The cov formal: a parameterized kernel object

Remind that **kergp** puts much emphasis on *kernel objects* which are used to define centered GPs as $\zeta(\mathbf{x})$ in (1). This argument is used to pass a covariance kernel object inheriting from the `"covAll"` class e.g., an object in one of the classes `"covMan"` and `"covRadial"`.

The covariance kernel object embeds a specific value for the vector $\boldsymbol{\theta}$ of covariance parameters; This value can be extracted by using the `coef` method and it can be changed by using the `coef<-` replacement method. It also contains bounds on the parameters, that can be extracted with `coefLower` and `coefUpper` and can be set by using `coefLower<-` and `coefUpper<-`. The bounds often play a major role in the estimation.

```
myCov <- kMatern(nu = "3/2")
inputNames(myCov) <- "x"
coef(myCov)

## theta_1  sigma2
##       1       1

coef(myCov) <- c(1, 4)
```

The kernel object can be used in gp.

```
myGp <- gp(y ~ 1, data = myData, cov = myCov)
coef(myGp)

## (Intercept)     theta_1      sigma2    varNoise
##      0.9781      0.3576      0.5749      0.0328
```

The `predict` method of the `"gp"` class can then be used to obtain the elements shown on Figure 2 (code not shown here).

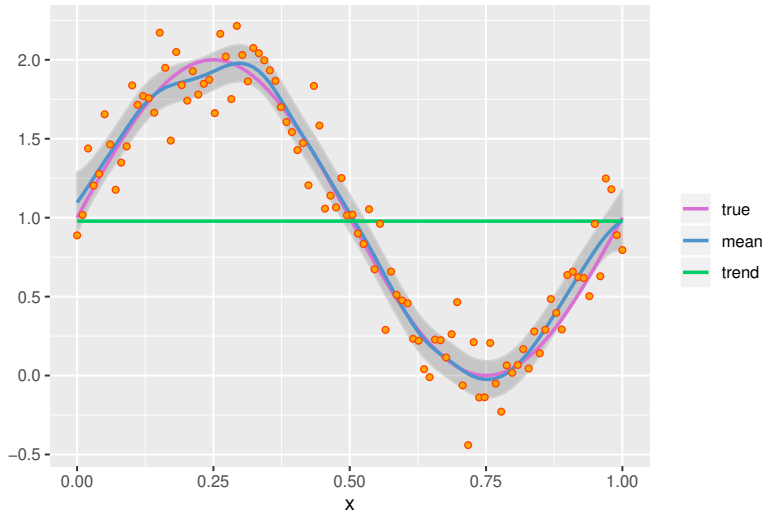

Figure 2: Simple example data and prediction from the `gp` object. The ribbon in gray shows the 95% confidence interval for the regression mean.

Trend and noise

The trend is coped with by using the classical formula-data interface, as in `lm` and in many other statistical modelling functions in R, see Figure 1. A model with no trend can be obtained by using a formula with right-hand side `~ 1` and by setting then the value of the trend parameter $\boldsymbol{\beta}$ to the scalar value zero by using the optional argument `beta` according to `beta = 0`.

Remark 1. The trend variables do not really need to be functions of the input vector $\mathbf{x}$ and covariates can be used classically.

The logical argument `noise` can be used to fit a model with no noise, in which case no duplicated value should exist among the input vectors $\mathbf{x}_i$.

What is a gp?

Since `"gp"` is a S3 class, it does not have a formal representation. A `gp` object is a list with elements containing the results of the estimation as well as some intermediate results useful for prediction. This choice was made to allow changes in the structure of the class, which still may evolve in the future. So you can extract some elements of the list for your needs, but we do not warrant that the list will remain exactly the same in future versions.

```
names(myGp)

##  [1] "call"       "terms"      "inputNames" "dim"        "y"
##  [6] "X"          "F"          "L"          "betaHat"    "eStar"
## [11] "FStar"      "sseStar"    "RStar"      "covariance" "noise"
## [16] "varNoise"   "trendKnown" "optValue"   "MLE"        "logLik"

names(myGp$MLE)

##  [1] "logLik"     "parIni"     "opt"        "cov"        "noise"
##  [6] "varNoise"   "trendKnown" "trendRes"   "parTracked" "logLikFun"
## [11] "report"

names(myGp$MLE$opt)
```

```
##  [1] "x0"                    "eval_f"
##  [3] "lower_bounds"          "upper_bounds"
##  [5] "num_constraints_ineq"  "eval_g_ineq"
##  [7] "num_constraints_eq"    "eval_g_eq"
##  [9] "options"               "local_options"
## [11] "nloptr_environment"    "call"
## [13] "termination_conditions" "status"
## [15] "message"               "iterations"
## [17] "objective"             "solution"
## [19] "version"               "num.evals"
## [21] "par"                   "value"
## [23] "convergence"
```

An element of the `gp` list can be a list and even be a list of lists. Here `myGp$MLE$opt` is essentially a copy of the result returned by the optimization function. Remind that the content of the list depends on the call. For instance with `estim = FALSE` the element `MLE` of the list would be `NULL`. By changing the optimization function, the content of `myGp$MLE$opt` would be changed because the two allowed optimization functions return quite different list objects.

Remark 2. The names of the variables in the code of the package: `X`, `F`, `beta`, ... are closely related to the matrices $\mathbf{X}$, $\mathbf{F}$, $\boldsymbol{\beta}$, ... used in the matrix notation (2).

1.3 Optimize, or not

The creator function `gp` relates $\boldsymbol{\beta}$ to `beta`, and the noise variance σ_ε^2 relates to `varNoise`. Depending on whether the parameters are known or not, `gp` will be used differently and will invoke a different *method*[3], see Figure 3.

```
myGpNoEst <- gp(y ~ 1, data = myData, cov = myCov,  estim = FALSE,
                varNoise = TRUE)
coef(myGpNoEst)

## (Intercept)     theta_1      sigma2    varNoise
##       1.079       1.000       4.000       1.000

myGpNoEst$MLE

## NULL
```

We see that the trend parameter $\boldsymbol{\beta}$ was estimated, but that neither the covariance parameter $\boldsymbol{\theta}$ nor the noise variance σ_ε^2 were estimated. When a noise term exists, it will be convenient to consider $\boldsymbol{\theta}$ and σ_ε^2 as a single vector of non-trend parameters $\boldsymbol{\psi} := [\boldsymbol{\theta}, \sigma_\varepsilon^2]$. When no noise term is found in the model, we consistently set $\boldsymbol{\psi} := \boldsymbol{\theta}$. The `gp` creator will work differently depending on what has to be estimated.

- *Known trend $\boldsymbol{\beta}$ and known non-trend $\boldsymbol{\psi}$.* Use `estim = FALSE`, and give a suitable vector of trend coefficients in the formal argument `beta`. In this case, the creator function `gp` will call the `gls` method, which will simply compute auxiliary matrices and vectors useful for prediction.

  Formal args passed to `gp` with the dots ellipsis ... will be passed to the `gls` method.

- *Know trend $\boldsymbol{\beta}$, unknown non-trend $\boldsymbol{\psi}$.* Give the `beta` formal argument. Then $\boldsymbol{\psi}$ will be estimated by ML and `gp` will call the `mle` method.

  Formal args passed to `gp` with the dots ellipsis ... will be passed to the `mle` method.

[3] In the R terminology meaning.

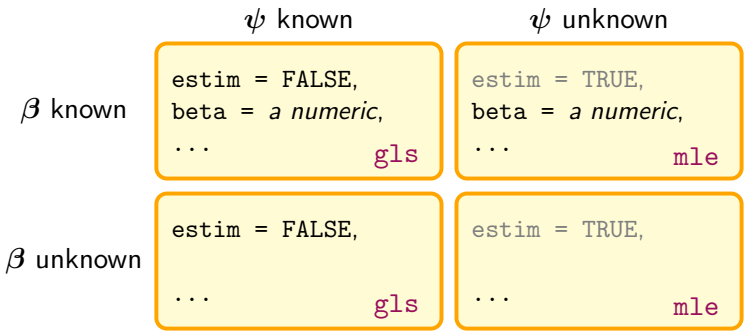

Figure 3: Depending on whether the trend parameter $\boldsymbol{\beta}$ and the non-trend parameter $\boldsymbol{\psi} = [\boldsymbol{\theta}, \sigma_\varepsilon^2]$ are known or not, `gp` will call the `gls` or the `mle` method indicated in red in the right lower corner of the relevant box. The formals in the dots ellipsis are passed to this method.

- *Unknown trend $\boldsymbol{\beta}$ and known non-trend $\boldsymbol{\psi}$.* Use `estim = FALSE`. Then the ML estimation can be achieved by Generalized Least Squares (GLS). In this case, `gp` will call the `gls` method to estimate $\boldsymbol{\beta}$.

  Formal args passed to `gp` with the dots ellipsis `...` will be passed to the `gls` method.

- *Unknown trend $\boldsymbol{\beta}$ and unknown non-trend $\boldsymbol{\psi}$.* The creator function `gp` will call the `mle` method.

  Formal args passed to `gp` with the dots ellipsis `...` will be passed to the `mle` method.

So remind that in the manual of the package, the help about optimization will *not* be found in the page dedicated to `gp`, but rather in that of the `gls` or `mle` methods, which are S4. The first argument of these methods is a (S4) kernel object like that given by the `cov` formal of `gp`. This structure potentially allows the use of an estimation method which takes into account the specific features of the kernel, see section 4.2.

1.4 What to optimize?

The log-likelihood

A numerical optimization is really required only when `gp` is called with `estim = TRUE`. The ML estimate of the vector of parameters is obtained as the argument of a log-likelihood function which is maximized, usually a profile log-likelihood.

In the most general form, the log-likelihood function $\ell := \log L$ of the model (2) has the form

$$\ell(\boldsymbol{\beta}, \boldsymbol{\theta}, \sigma_\varepsilon^2; \mathbf{X}, \mathbf{y}).$$

Its evaluation requires the evaluation of the covariance matrix $\mathbf{K}(\mathbf{X}; \boldsymbol{\theta})$ and its Cholesky decomposition at a cost of $O(n^3)$. We will now on omit the dependence of ℓ on the input matrix $\mathbf{X}$ and the response vector $\mathbf{y}$.

Profile log-likelihood

Given a covariance parameter $\boldsymbol{\theta}$ and a noise variance σ_ε^2, finding the value of $\boldsymbol{\beta}$ which maximizes the log-likelihood ℓ i.e.,

$$\widehat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \sigma_\varepsilon^2) := \arg\max_{\boldsymbol{\beta}} \ell(\boldsymbol{\beta}, \boldsymbol{\theta}, \sigma_\varepsilon^2)$$

is easily achieved by GLS. Indeed, (2) is nothing but a linear regression with correlated errors. Hence the ML estimation boils down to the maximization of so-called *profile* (or *concentrated*) log-likelihood

$$\ell_{\mathrm{pr}}(\boldsymbol{\theta},\, \sigma_\varepsilon^2) := \ell[\widehat{\boldsymbol{\beta}}(\boldsymbol{\theta},\, \sigma_\varepsilon^2),\, \boldsymbol{\theta},\, \sigma_\varepsilon^2] \tag{3}$$

which depends only on the vector $\boldsymbol{\psi} = [\boldsymbol{\theta},\, \sigma_\varepsilon^2]$ of non-trend parameters formed by the covariance parameter $\boldsymbol{\theta}$ and the noise variance σ_ε^2. The profile log-likelihood is easier to maximize than the original log-likelihood. Still, each evaluation of ℓ_{pr} involves computing the matrix $\mathbf{K}(\mathbf{X};\, \boldsymbol{\theta})$ hence is costly.

No noise

When the model has no noise, the profile log-likelihood function that is maximized depends only of the covariance parameter $\boldsymbol{\theta}$, so in this case $\boldsymbol{\psi} = \boldsymbol{\theta}$.

Remark 3. If the chosen kernel is very smooth[4] the optimization can be difficult, because the $\mathbf{K}(\mathbf{X})$ may fail to be numerically definite positive.

Given trend vector $\boldsymbol{\beta}$

When the trend vector $\boldsymbol{\beta}$ is given, the log-likelihood to be optimized no-longer depends on $\boldsymbol{\beta}$: It corresponds to a "slice" of the log-likelihood

$$\ell_{\mathrm{sl}}(\boldsymbol{\theta},\, \sigma_\varepsilon^2) := \ell(\boldsymbol{\beta},\, \boldsymbol{\theta},\, \sigma_\varepsilon^2) \tag{4}$$

for the case where a noise is used. More generally, the slice likelihood function to be maximized depends on the vector $\boldsymbol{\psi}$ of the non-trend parameters.

Gradient

The gradient of the optimized function[5] can be evaluated by chain rule using the gradient of the covariance matrix $\partial\mathbf{K}/\partial\boldsymbol{\theta}$, provided that the later is available. For the sake of efficiency, a single R function should be used to evaluate both the value ℓ_{pr} and that of the gradient $\partial\ell_{\mathrm{pr}}/\partial\boldsymbol{\psi}$. Indeed, a number of intermediate variables such as $\mathbf{K}$ and its Cholesky decomposition are used for both evaluations.

When this is possible, using the gradient makes the optimization much faster: a smaller number of evaluations of ℓ_{pr} is required.

1.5 How to optimize?

Initial values

As is often the case for ML estimation, we use *local* optimization methods as default. For some specific kernels and data, you might have an idea of initial values for $\boldsymbol{\theta}$ and σ_ε^2, and provide these the `parCovIni` and `varNoiseIni`.

[4] Say C^2 or more.

[5] Namely $\partial\ell_{\mathrm{pr}}/\partial\boldsymbol{\psi}$ or $\partial\ell_{\mathrm{sl}}/\partial\boldsymbol{\psi}$ depending on whether $\boldsymbol{\beta}$ is known or not.

Bounds on parameters

The optimization methods should preferably accept bounds on the parameters ψ_k

$$\psi_{\mathrm{L},k} \leqslant \psi_k \leqslant \psi_{\mathrm{U},k} \tag{5}$$

where $\psi_{\mathrm{L},k}$ can be $-\infty$ i.e., `-Inf` and $\psi_{\mathrm{U},k}$ can be ∞ i.e., `Inf`. These bounds can be set by using the formal arguments of the `mle` method for the `"covAll"` class. They can be passed to `mle` from `gp` by the dots ellipsis.

- The bounds on the *covariance* parameters can be given by the `parCovLower` and `parCovUpper` arguments. The default bounds are given by `coefLower(cov)` and `coefUpper(cov)` and they can be set by using the replacement methods `coefLower<-` and `coefUpper<-` before calling `gp` or `mle`.
- The bounds on the *noise variance* σ_ε^2 are set by using `varNoiseLower` and `varNoiseUpper` arguments.

With the default optimization methods, it is possible to give identical lower and upper bounds corresponding to $\psi_{\mathrm{L},k} = \psi_{\mathrm{U},k}$ in (5), thus providing a convenient way to fix the value of ψ_k.

```
myGpConstr <- gp(y ~ 1, data = myData, cov = myCov,
                 compGrad = TRUE,
                 parCovLower = c(1.0, 0.01), parCovUpper = c(1.0, 100.0),
                 trace = 1)

## optimFun :    nloptr::nloptr
## optimMethod : NLOPT_LD_LBFGS

## Warning in nloptr.add.default.options(opts.user = opts, x0 = x0, num_constraints_ineq = num_constraints_ineq,
:  No termination criterium specified, using default (relative x-tolerance = 1e-04)

##
## Optimisation report:
##
##   parIni.theta_1 parIni.sigma2 parIni.varNoise par.theta_1 par.sigma2
## 1              1             4         0.05219           1      6.553
##   par.varNoise logLik nIter convergence
## 1      0.03299  9.208    17        TRUE

## Warning in .local(object, ...):  optimisation did not converge

coef(myGpConstr)

## (Intercept)     theta_1      sigma2    varNoise
##     0.88132     1.00000     6.55303     0.03299
```

So here the range parameter of the kernel has a fixed value 1.0. Similarly, we could have set the value of the noise variance σ_ε^2, still optimizing on the covariance parameter $\boldsymbol{\theta}$.

Multistart

As far as a general covariance kernel is used, no general rule seems to exist for providing good initial values. So trying *several* initial values is an option.

The `multistart` formal argument of the `mle` method can be used to set a number of initial parameter vectors $\boldsymbol{\psi}$ to be tried in optimization. When these values are not found as expected

in `parCovIni`, they are drawn randomly in the hyper-cube defined by the constraints by using the `simulPar` function. Then the vector $\widehat{\psi}$ of estimated parameters is the one for which the optimization resulted in the largest maximized log-likelihood. The optimizations can be run in parallel e.g., by using the **doFuture** package (Bengtsson, H., 2017).

Remark 4. For now, the parallel execution works only when `optimFun = nloptr::nloptr`.

2 Tuning optimization

2.1 Different levels of tuning

Function and method (or algorithm)

In the `mle` method of **kergp**, two standard optimization *functions* can be used: `"nloptr::nloptr"` – which now is the default – and `"stats::optim"`. The **nloptr** package Ypma J. (2014) provides an interface to the famous **NLopt** library (Johnson, S.G., 2017). The choice of the optimization function is done by using the formal argument `optimFun`, for which partial matching is allowed. Both functions allow several choices of the optimization *method* (or algorithm) via its own syntax; The `optimMethod` argument of `mle` can be used for both functions.

For the default optimization function, we have selected *two* optimization algorithms: The choice among these is given by the value of the logical `compGrad` which tells if the gradient is used or not. When `optimFun` is set to `"stats::optim"` the default is `optimMethod = "L-BFGS-B"`, although this implementation does not accept `NA`s.

Three levels of tuning

The **kergp** package has the following possibilities which will be detailed.

1. Still using one of the optimization functions and one of its default or recommended methods (or algorithms), achieve a *light tuning* e.g., in order to change the maximal number of evaluations of the objective.

   You only need to play with the two formals `optimFun` and `optimMethod` to choose the function and the method, and then pass a list of options via the arguments `opts` or `control`, depending on the value of `optimFun`.

2. Still using one of the two optimization functions, access to another optimization method, and tune its parameters.

   You will have to carefully read the documentation of the `nloptr::nloptr` or of the `stats::optim` function, and possibly some related links.

3. Use another optimization function e.g., coming from an extra package.

   You will have to understand some features of the code of the `mle` method in order to choose the name of the suitable objective function and of the formal arguments of the optimization function that you want to use.

Function: nloptr::nloptr vs stats::optim

The function `nloptr::nloptr` has a very wide choice of optimization methods[6] that can normally be acceded to by using the `opts` formal, a named list. The same formal `opts` contains

[6] $\geqslant$ 20 methods.

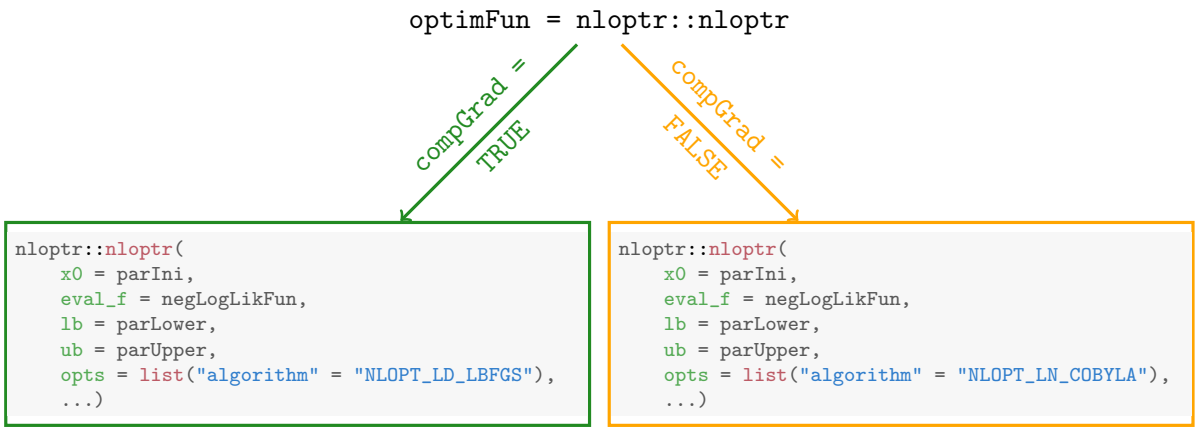

Figure 4: Default settings for the case `optimFun = nloptr::nloptr`. The codes show the call to `nloptr::nloptr` from the `mle` method for the `"covAll"` class, depending on the value of `compgrad`. The objects such as `parLower` have been defined within the method, before the call. Note that `nloptr::nloptr` always *minimizes*, so the negative of the log-likelihood is used.

elements to tune the optimization: for instance, the maximal number of evaluations of the objective is controlled via the `maxeval` element.

The `stats::optim` function has a smaller number of methods[7]. The choice is normally made by using the `method` formal, while the parameters controlling the optimization are given in the `control` formal, a named list. For instance, the maximal number of iterations is controlled via the `maxit` element.

Both for `stats::optim` and `nloptr::nloptr`, there exist pre-defined values corresponding to the cases where `opts` and `control` are not given to the optimization function. These are used by default by the `mle` method. The pre-defined stopping rules of `nloptr::nloptr` differ from that of `stats::optim`.

Most often, box constraints on parameters should be used. For instance, range or variance parameters must be $\geqslant 0$.

2.2 Using nloptr `optimFun = "nloptr::nloptr"`

Default settings

The call to the optimization function from the `mle` method is shown on figure 4. Depending on whether the gradient is available, a different "standard" algorithm is chosen. Inasmuch as you still want to use the standard algorithm which is compliant with the value of `compGrad`, you will change the value of some elements of the list `opts`, but not that of the `algorithm` element, because this will anyway be set to the value of `optimMethod`.

Light tuning with `opts`

To change the default settings for the current algorithm, you can pass a list object in the `opts` formal argument, used via the dots mechanism. Of course, only values differing from the default in `opts` will have an effect. The list of the options that can be tuned is given by the function `nloptr::nl.opts`

[7] 6 methods

```
library(nloptr)
names(nl.opts())

## [1] "stopval"           "xtol_rel"          "maxeval"
## [4] "ftol_rel"          "ftol_abs"          "check_derivatives"
## [7] "algorithm"
```

Remark 5. As far as we are optimizing a *log*-likelihood function which is defined up to an additive constant, it does not really make sense to use `ftol_rel`. Setting `ftol_abs = 1e-3` seems a sensible choice.

Remark 6. No control is done by `nloptr::nloptr` to check the name of the elements of the `opts` list. For instance passing `opts = list("foo" = "bar")` will not warn, but do nothing. This can be misleading when there is a typo in the name of the option, as in `opts = list("maxEval" = 1000)`.

An example of use of `opts` is as follows.

```
myGpTuned1 <- gp(y ~ 1, data = myData, cov = myCov,
                 opts = list("maxeval" = 5, "print_level" = 2))

## iteration: 1
##  f(x) = -4.615403
## iteration: 2
##  f(x) = 255798.106315
## iteration: 3
##  f(x) = 3.022986
## iteration: 4
##  f(x) = -8.605493
## iteration: 5
##  f(x) = -8.606547

coef(myGpTuned1)

## (Intercept)     theta_1      sigma2    varNoise
##     0.95727     0.99899     4.00007     0.03383
```

The effect of `opts` is quite clear: the optimization stops as soon as `maxeval` evaluations of the function have been done, and some information is printed at each evaluation.

Change the algorithm with `optimMethod`

With some care, you can change as well the value of the algorithm used by `nloptr::nloptr`, using the `optimMethod` argument of `gp` and giving one of the following accepted values.

```
om <- optimMethods(optimFun = "nloptr")
head(om)

##         optimFun                  optimMethod
## 1 nloptr::nloptr              NLOPT_GN_DIRECT
## 2 nloptr::nloptr            NLOPT_GN_DIRECT_L
## 3 nloptr::nloptr       NLOPT_GN_DIRECT_L_RAND
## 4 nloptr::nloptr       NLOPT_GN_DIRECT_NOSCAL
## 5 nloptr::nloptr     NLOPT_GN_DIRECT_L_NOSCAL
## 6 nloptr::nloptr NLOPT_GN_DIRECT_L_RAND_NOSCAL

nrow(om)

## [1] 37
```

The list of algorithms could as well have been obtained with

```
nloptr.print.options("algorithm")
```

As suggested by the output message (not shown here), the detailed information on the available algorithms is to be found on the NLopt website, because **NLopt** is not specifically intended to be used with R.

In the names of the algorithms available with `nloptr::nloptr`, the second two-letter word `GN` tells if the algorithm is Global (`G`) of local (`L`) and if it needs a gradient (or derivative `D`) or not (`N`). So the four possible values are `GN`, `LN`, `GD` and `GN`. Remind that when an algorithm requires a gradient, the objective function `negLogLikFun` must return a *list* with the two elements `"objective"` and `"gradient"`, which will be the case only if `compGrad = TRUE` is used. Using a value of `compGrad` which is not compliant with the algorithm may lead to an error message which is difficult to understand.

As an illustration, we can choose a Nelder-Mead "simplex" `LN` method, a Local method `Not` using a gradient.

```
myGpTuned2 <- gp(y ~ 1, data = myData, cov = myCov,
                 optimMethod = "NLOPT_LN_NELDERMEAD", compGrad = FALSE)
coef(myGpTuned2)

## (Intercept)     theta_1      sigma2    varNoise
##     0.97796     0.35054     0.56109     0.03255
```

2.3 Using optim `optimFun = "stats::optim"`

Default settings

In this case, the call to the optimization function from the `mle` method is shown on figure 5. As a difference with `nloptr` before, the same method or algorithm name `"L-BFGS-B"` can used for both values of `compGrad`, because they are now considered as two variants of a same algorithm. The gradient function will not be used nor even be defined when `compGrad` is `FALSE`, and the algorithm will work differently, then requiring more iterations.

Light tuning with `control`

You can change the elements of the `control` list (except of course `fnscale`), or add new elements in this list, see `?stats::optim`. For this aim, use the `control` argument, via the dots mechanism. For instance the number of iterations can be set with the `maxit` element.

```
myGpOptimTuned1 <- gp(y ~ 1, data = myData, cov = myCov,
                      compGrad = TRUE,
                      optimFun = "stats::optim", optimMethod = "L-BFGS-B",
                      control = list(maxit = 10))

## Warning in .local(object, ...):  optimisation did not converge

coef(myGpOptimTuned1)

## (Intercept)     theta_1      sigma2    varNoise
##     0.89588     0.78663     4.12460     0.03299
```

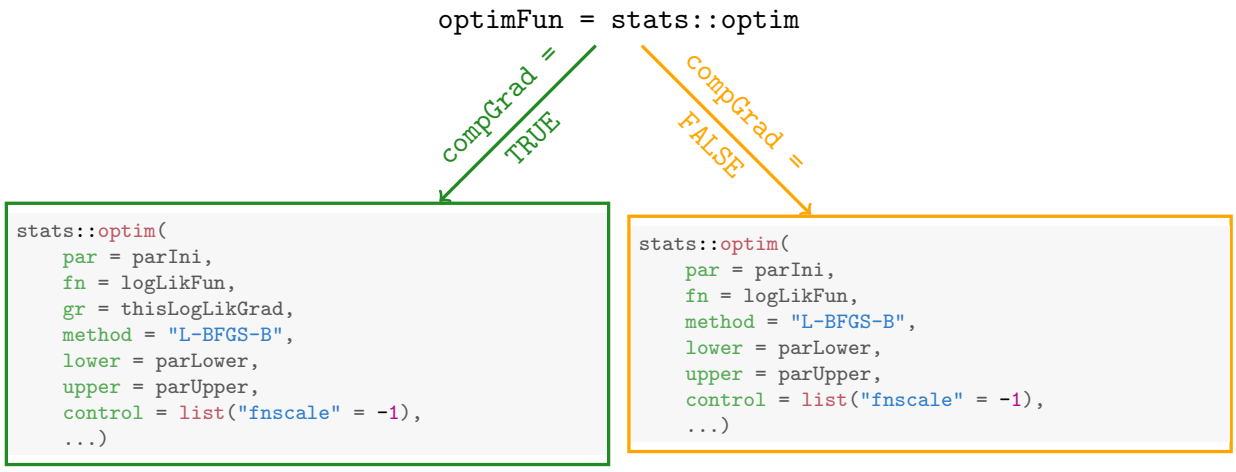

Figure 5: Default settings for the case `optimFun = stats::optim`. Here, for historical reasons, we are maximizing. Although the objective function has the same name in both cases its definition is different: it computes and "caches" the gradient when `compGrad` is `TRUE`. Then, the gradient function passed to `gr` only retrieves the gradient from the cache environment.

Change the algorithm with `optimMethod`

The optimization algorithm corresponding to the formal argument `method` of `stats::optim` should be given in the call to `gp` via `optimMethod`, and *not* be passed via the dots mechanism using the `method` formal argument. The possible choices are as follows.

```
om <- optimMethods(optimFun = "stats::optim")
om

##       optimFun optimMethod
## 1 stats::optim Nelder-Mead
## 2 stats::optim        BFGS
## 3 stats::optim          CG
## 4 stats::optim    L-BFGS-B
## 5 stats::optim        SANN
## 6 stats::optim       Brent
```

Note that for all algorithms except `"L-BFGS-B"` and `"Brent"`, the bounds on the parameters can not be used. This can lead to an unfeasible parameter vector containing negative ranges, or a negative noise variance.

```
myGpOTuned2 <- gp(y ~ 1, data = myData, cov = myCov,
                  optimFun = "stats::optim",
                  optimMethod = "Nelder-Mead", compGrad = FALSE)
coef(myGpOTuned2)

## (Intercept)     theta_1      sigma2    varNoise
##      0.9781      0.3576      0.5749      0.0328
```

Now, try a *global* optimization

```
myGpOTuned3 <- gp(y ~ 1, data = myData, cov = myCov,
                  optimFun = "stats::optim",
                  optimMethod = "SANN", compGrad = FALSE)
coef(myGpOTuned3)

## (Intercept)     theta_1      sigma2    varNoise
##     0.94628     0.49691     1.43057     0.03374
```

Strengths and weaknesses of `stats::optim`

The `stats::optim` function is very popular choice among R users who often use `method = "BFGS"` for MLE. Although it works fine out of the box in many contexts, it suffers from limitations making its use tedious for the estimation of `gp` models.

- Among the two (local) BFGS methods `"L-BFGS-B"` and `"BFGS"`, only the later accepts `NA` (or non-finite) values of the objective, see next section. Only the former allows the use of box-constraints on parameters. See Table 1.
- The objective and its gradient are used in separated functions, while important savings can be obtained by using a joint computation. In the `mle` method, the joint computation of the objective and of the gradient is obtained through a hack, using an R environment as a cache.

Remark 7. To a certain extend, box constraints could be coped with by ensuring that the objective function takes a value `NA` when the constraints do not hold. This tip allows dozens of MLE functions in CRAN packages to work correctly although they do not duly use constraints. However this option was not retained in **kergp**.

2.4 Choosing `optimFun` and `optimMethod`

Compared features

Table 1 compares the features of the default and standard algorithms.

Use gradient when possible

Most of the kernel objects shipped with **kergp** provide the gradient so should be used with an optimization algorithm taking advantage of the gradient. If you are using your own `covMan` kernels, consider providing the gradient.

Accept NA

Among the algorithms available in R for numerical optimization, some accept `NA` (or non-finite) values of the objective: If at some point of the algorithm the objective takes a non-finite value, other values of the parameter vector are tested until a valid value of the objective is reached, and the algorithm can then resume its progression toward an optimum.

For the MLE of GP models, using an algorithm which accepts `NA` is a concern, because it is quite common within the optimization process to face a covariance matrix which is not *numerically* positive definite, leading to an undefined value of the objective.

2.5 Using an alternative optimization function ⋆

There are many R packages devoted to optimization, and you might want to use one of these to maximize the likelihood of your `gp` model. As is clear from what precedes, the optimization functions in R do not have the same list of formal arguments (a.k.a. *signature*) and an optimization function can have specific tuning parameters. Thus, no common interface can be proposed for all the usable optimization functions, and it would be tedious to add even a limited number of new usable optimization functions. Yet, it is possible and quite simple to plug any optimization function using *expressions* passed in the `optimCode` formal of `gp`.

optimFun	optimMethod	**Grad.**	**NAs**	**Bounds**
nloptr::nloptr	opts$algorithm = "NLOPT_LD_LBFGS"	Yes	Yes*	Yes
	opts$algorithm = "NLOPT_LN_COBYLA"	**No**	Yes*	Yes
stats::optim	method = "BFGS"	Yes	Yes	**No**
	method = "L-BFGS-B"	Yes	**No**	Yes

Table 1: Standard optimization methods in `kergp::gp`. With the `stats::optim`, a hack is used to evaluate efficiently the gradient. Remind that with `gp`, the method or algorithm is given by `optimMethod`. ($\star$) With `nloptr::nloptr`, `NA` or non-finite values of the objective are transformed into `NaN` to be accepted.

Despite of its name, the value passed to `optimCode` will generally not be a character object containing a R code, but rather an *expression*, usually written as a block limited by curly braces `{` and `}`. The next code chunk provides an example; Although it simply changes the settings for `nloptr::nloptr`, it could clearly have used any optimization function.

```
myOptimCode <- expression({

    optOpts <- list("algorithm" = "NLOPT_LD_LBFGS",
                    "xtol_rel" = 1.0e-8, "xtol_abs" = 1.0e-8, "ftol_abs" = 1e-8,
                    "maxeval" = 3000,
                    "check_derivatives" = TRUE)

    opt <- try(nloptr::nloptr(x0 = parIni, eval_f = negLogLikFun,
                              lb = parLower, ub = parUpper,
                              opts = optOpts, ...))

    if (!inherits(opt, "try-error")) {
        opt$par <- opt$solution
        opt$value <- -opt$objective
        opt$convergence <- opt$status > 0
    } else stop("Error in my optim")

    report <- NULL

})
```

The expression `optimCode` will be evaluated at some chosen point in the execution of `mle` method. So it can use objects existing at this point such as `parUpper`. It must define an object `opt`: a list containing the results of the optimization with

- `opt$par` the value of the parameter vector,
- `opt$value` the value of the maximized log-likelihood,
- `opt$convergence` a logical indicating if the optimization is thought of as eventually converged.

The code can also define an object `report`, a list intended to report about optimization. Note that by using `try`, a suitable error control can be obtained.

Now let us try our optimization code defined above. We use `trace = 1` to be sure that our code was really run.

```
myGpOptimCode <- gp(y ~ 1, data = myData, cov = myCov,
                    optimCode = myOptimCode, trace = 1)

## Using the provided value of 'optimCode'
## ========================================

## Checking gradients of objective function.
## Derivative checker results:  0 error(s) detected.
##
##  eval_grad_f[ 1, 1 ] =  8.444e+00 ~  8.444e+00  [3.588e-06]
##  eval_grad_f[ 2, 1 ] = -5.952e-01 ~ -5.952e-01  [1.243e-05]
##  eval_grad_f[ 3, 1 ] =  3.159e+02 ~  3.159e+02  [1.970e-07]
## Warning in .local(object, ...):  optimisation did not converge

coef(myGpOptimCode)

## (Intercept)     theta_1      sigma2    varNoise
##      0.9781      0.3576      0.5749      0.0328
```

3 Getting more information on the optimization

3.1 Printing

By default, the optimization will print nothing and only warnings of errors will be seen. The `trace` argument of the `mle` method can be used to obtain more information on the optimization steps. This argument can be passed from `gp` via the dots ellipsis `...`

3.2 Tracking the parameters

The `mle` method has a `parTrack` formal argument which can be used to track the values of the parameter ψ for which an evaluation of the log-likelihood was made. When `estim` and `parTrack` are both set to `TRUE`, the `MLE` list in the created `gp` object contains a matrix `parTracked` having the iterates as its rows. This matrix can be used e.g, with `pairs` to visualize the path of the iterates in the parameter space.

For now, using `parTrack` in a parallel setting is unwarranted.

4 Further customization

4.1 Use the `logLikFun` closure given by `gp`

The `gp` creator function returns a log-likelihood "function" as the `logLikFun` element of the `MLE` list. More precisely, this is a *closure*, not a plain function: it keeps links with some data according to the *lexical scoping* mechanism. As is typical with log-likelihood functions, the data[8] are not considered as formal arguments.

```
myGp$MLE$logLikFun

## function (par)
## {
##     ll <- .logLikFun0(par, object, y = thisy, X, F = thisF, compGrad = compGrad,
##         noise = noise, gradEnv = gradEnv, trace = trace)
```

[8] Here `y`, `X`, ...

```
## }
## <bytecode: 0xb3dc350>
## <environment: 0x9d42140>
```

The `logLikFun` closure has only one formal argument named `par`. What the `par` vector is assumed to be depends on the values of the values of `noise` when the `mle` method was called. The `par` argument stands for the vector $\boldsymbol{\psi}$ in section 1.4 above.

Remind that `logLikFun` depends much on the value of the optional argument `beta` that was given when the `mle` method was called. If `beta` is `NULL`, the closure represents the profile log-likelihood with $\boldsymbol{\beta}$ concentrated out, while if `beta` is a provided vector, the closure represents a "slice" of the profile likelihood obtained by fixing $\boldsymbol{\beta}$ to the given value.

The `logLikFun` closure can be used in any optimization function in the session, but a parallel optimization will generally not work.

At the best, the `logLikFun` object is to be used in the plain R session in which `mle` was called, usually via `gp`. It is unwise to use the `logLikFun` object in a parallel computing framework or even to store it in view of reusing it. With these limitations, you can use any optimization function available to you in order to get the ML estimates. Remind that `logLikFun` will sometimes return `NA`, so the chosen optimization function should preferably allow a `NA` objective value.

4.2 Writing classes and methods

Although **kergp** comes with several classes of kernel objects[9], the `mle` method is for now written only for the `"covAll"` virtual class. Within the `mle` method, the `covMat` method is invoked to compute the matrix $\mathbf{K}(\mathbf{X})$.

You can write extra covariance classes and use them with `gp`. For some classes of kernels a specific `mle` method can be implemented and then automatically used via the dispatch mechanism. A typical situation where considerable savings can be achieved is for a kernel object representing a Markov GP or a semi-Markov GP. As an example when $d = 1$ and Matérn kernels with shape $\nu = 1/2,\ 3/2,\ \ldots$ are used, the GP is a Continuous Auto-Regressive process; the MLE can be performed in $O(n)$ operations as well as with reduced numerical problems for dense designs.

References

Bengtsson, H. (2017). *doFuture: A Universal Foreach Parallel Adaptor using the Future API of the 'future' Package*. R package version 0.6.0.

Johnson, S.G. (2017). The NLopt Nonlinear-Optimization Package. `http://ab-initio.mit.edu/nlopt`.

Ypma J. (2014). *R interface to NLopt*. R package version 1.0.4.

[9] `covMan`, `covRadial`, ...