Defining a latent model in R: The rgeneric model

Haavard Rue (hrue@r-inla.org)

Nov 28th 2020

Introduction

This is a class of generic models allows the user to define latent model component in R, for cases where the requested model is not yet implemented in INLA, and do the Bayesian inference using INLA. Since the model component has to be interpreted in R, it will run slower compared to a similar model implemented in INLA.

Defining a latent model in R

A rgeneric model is defined in a function rmodel (to be defined later), and the usage is quite simple. First we need to define a inla-rgeneric object

```
model = inla.rgeneric.define(rmodel, ...)
```

with additional variables/functions/etc in ... that we might use in rmodel. This can be the size, prior parameters, covariates, external functions and so on. This object can then be used to define a normal model component in INLA using f(),

```
y ~ ... + f(idx, model=model, ...)
```

where idx can take values $1, 2, \ldots, n$ where n is the size of model. All additional features for f() will still be valid.

Example: The AR1 model

The function rmodel needs to follow some rules to provide the required features. We explain this while demonstrating how to implement the AR1-model. This model already exists, see inla.doc("ar1"). With the parmeterisation we use, the AR1-model is defined as

$$x_1 \sim \mathcal{N}(0, \tau) \qquad \text{and} \qquad x_t \mid x_1, \ldots, x_{t-1} \sim \mathcal{N}(\rho x_{t-1}, \tau_I), \qquad t = 2, \ldots, n.$$

where $\tau_I = \tau/(1-\rho^2)$. The scale-parameter is the *marginal precision* τ, **not** the commonly used innovation precision τ_I. The joint density of x is Gaussian

$$\pi(x|\rho, \tau) = \left(\frac{1}{\sqrt{2\pi}}\right)^n \tau_I^{n/2} (1-\rho^2)^{1/2} \exp\left(-\frac{\tau_I}{2} x^T R x\right)$$

where the precision-matrix is

$$Q = \tau_I R = \tau_I \begin{bmatrix} 1 & -\rho & & & & \\ -\rho & 1+\rho^2 & -\rho & & & \\ & -\rho & 1+\rho^2 & -\rho & & \\ & & \ddots & \ddots & \ddots & \\ & & & -\rho & 1+\rho^2 & -\rho \\ & & & & -\rho & 1 \end{bmatrix}$$

There are two (hyper-)parameters for this model: the marginal precision τ and the lag-one correlation ρ. We will reparameterise these as

$$\tau = \exp(\theta_1), \qquad \text{and} \qquad \rho = 2\frac{\exp(\theta_2)}{1+\exp(\theta_2)} - 1.$$

It is required that the parameters $\theta = (\theta_1, \theta_2)$ have support on $\Re^2$ and the priors for τ and ρ are given as the corresponding priors for θ_1 and θ_2.

A good re-parametersation is required for `INLA` to work well. A good parmeterisation makes, ideally, the *Fisher information matrix* of θ constant with respect to to θ. It is sufficient to check this in a frequentistic setting with data directly from the AR(1) model, in this case. `INLA` will provide the posterior marginals for θ, but `inla.tmarginal()` can be used to convert it to the appropriate marginals for ρ and τ.

We assign Gamma prior $\Gamma(.; a, b)$ (with mean a/b and variance a/b^2) for τ and a Gaussian prior $\mathcal{N}(\mu, \kappa)$ for θ_2, so the joint prior for θ becomes

$$\pi(\theta) = \Gamma(\exp(\theta_1); a, b)\exp(\theta_1) \ \times \ \mathcal{N}(\theta_2; \mu, \kappa).$$

The extra term, $\exp(\theta_1)$ is the Jacobian for the change of variable from τ to θ_1. We will in this example use $a = b = 1$, $\mu = 0$ and $\kappa = 1$.

In order to define the AR1-model, we need to make `R`-functions that returns

- the graph,
- the precision matrix $Q(\theta)$,
- the zero mean,
- the initial values of θ,
- the log-normalising constant, and
- the log-prior

We need to incorporate these functions into `rmodel`, in the following way

```
inla.rgeneric.ar1.model = function(
        cmd = c("graph", "Q", "mu", "initial", "log.norm.const",
                "log.prior", "quit"),
        theta = NULL)
{
    # for reference and potential storage for objects to
    # cache, this is the environment of this function
    # which holds arguments passed as `...` in
    # `inla.rgeneric.define()`.
    envir = parent.env(environment())

    graph = function(){ <to be completed> }
    Q = function() { <to be completed> }
    mu = function() { <to be completed> }
    log.norm.const = function() { <to be completed> }
    log.prior = function() { <to be completed> }
    initial = function() { <to be completed> }
    quit = function() { <to be completed> }

    # sometimes this is useful, as argument 'graph' and 'quit'
    # will pass theta=numeric(0) (or NULL in R-3.6...) as
    # the values of theta are NOT
    # required for defining the graph. however, this statement
    # will ensure that theta is always defined.
    if (!length(theta)) theta = initial()
```

```
    val = do.call(match.arg(cmd), args = list())
    return (val)
}
```

The input parameters are

- **cmd** What to return
- **theta** The values of the θ-parameters

Other parameters in the model definition, like n and possibly the parameters of the prior, goes into the `...` part of `inla.rgeneric.define()`, like

```
model = inla.rgeneric.define(inla.rgeneric.ar1.model, n = 100)
```

and is assigned in the environment of `inla.rgeneric.ar1.model`. Using variable `n` inside this function will then return `100`. This environment can also be accessed as `envir` as defined in the function skeleton. Sometimes this is useful, to hold static varibles or to cache intermediate calculations.

Our next task, is to *fill in the blanks* and define the functions required. To help us, we will add a function that return a list of the **real** parameters in the model from θ,

```
interpret.theta = function() {
    return(list(prec = exp(theta[1L]),
                rho = 2 * exp(theta[2L])/(1 + exp(theta[2L])) - 1))
}
```

Since `theta` exist already within `inla.rgeneric.ar1.model` we do not need to pass it as an argument.

We also assume that variable `n` is defined as an argument in `inla.rgeneric.define()`.

Function `graph()`

This is normally an easy function to add, as it is essentially the matrix Q. One can construct cases where this is not so[1], and for this reason it exists as a separate function. The only thing that matter is if the elements are zero or non-zero. Also, it should return a `sparse matrix` as we do not want to pass n^2 elements when $\mathcal{O}(n)$ are sufficient. Also, only the upper triangular matrix (diagonal included) are actually used, since the graph must be symmetric.

```
graph = function() {
    return (Q())
}
```

function `Q()`

This is normally the most tricky function, as we need to return the precision matrix (as a sparse matrix) for the given values of θ. Only the upper triangular matrix (diagonal included) are read.

A *dense matrix* version is as follows, and is easier to read

```
Q = function() {
    p = interpret.theta()
    Q = p$prec/(1 - p$rho^2) *
        toeplitz(c(1 + p$rho^2, -p$rho, rep(0, n - 2L)))
    Q[1, 1] = Q[n, n] = p$prec/(1 - p$rho^2)
    return (inla.as.sparse(Q))
}
```

[1] Depending on θ an element Q_{ij} might be exactly zero

The function `inla.as.sparse()` convert a matrix or sparse matrix, into the appropriate sparse matrix format used internally in `INLA`. This version of `Q()` creates a dense matrix and then make it sparse, and is not the way to do it. The better way, is to define the (upper triangular) sparse matrix directly using `sparseMatrix`.

```
Q = function() {
    p = interpret.theta()
    i = c(1L, n, 2L:(n - 1L), 1L:(n - 1L))
    j = c(1L, n, 2L:(n - 1L), 2L:n)
    x = p$prec/(1 - p$rho^2) *
        c(1L, 1L, rep(1 + p$rho^2, n - 2L),
          rep(-p$rho, n - 1L))
     return (sparseMatrix(i = i, j = j, x = x, giveCsparse = FALSE))
}
```

This is both faster and requires less memory, but it gets somewhat unreadable and hard to debug. The dense matrix version above, is at least easier to debug against for reasonable values of n.

Function `mu()`

This function must return the mean which might depend on θ. The convention, is that if `numeric(0)` is returned, then the mean is identical to zero (and then there is no need to check for this later)

```
mu = function() {
    return(numeric(0))
}
```

Function `log.norm.const()`

This function must return the log of the normalising constant. For the AR1-model the normalising constant is

$$\left(\frac{1}{\sqrt{2\pi}}\right)^n \tau_I^{n/2}(1-\rho^2)^{1/2}$$

where

$$\tau_I = \tau/(1-\rho^2).$$

The function can then be implemented as

```
log.norm.const = function() {
   p = interpret.theta()
   prec.i  = p$prec / (1.0 - p$rho^2)
   val = n * (- 0.5 * log(2*pi) + 0.5 * log(prec.i)) +
         0.5 * log(1.0 - p$rho^2)
   return (val)
}
```

Since the normalizing constant is known, we can ask `INLA` to evaluate

$$-\frac{n}{2}\log(2\pi) + \frac{1}{2}\log(|Q(\theta)|)$$

and $\log|Q(\theta)|$ can be computed from the sparse Cholesky factorisation of $Q(\theta)$. In this case we can return `numeric(0)` (which is a code for "`compute it yourself, please!`")

```
log.norm.const = function() {
   return (numeric(0))
}
```

Unless the log-normalizing constant is known analytically (and the precision matrix depends on θ) it is both better, and easier, just to return `numeric(0)`.

Function `log.prior()`

This function must return the (log-)prior of the prior density for θ. For the AR1-model, we have for simplicity chosen this prior

$$\pi(\theta) = \Gamma(\exp(\theta_1); a, b) \exp(\theta_1) \times \mathcal{N}(\theta_2; \mu, \kappa)$$

so we can implement this as with our choices $a = b = 1$, $\mu = 0$ and $\kappa = 1$ as

```
log.prior = function() {
    p = interpret.theta()
    val = dgamma(p$prec, shape = 1, rate = 1, log=TRUE) + theta[1L] +
          dnorm(theta[2L], mean = 0, sd = 1, log=TRUE)
    return (val)
}
```

The parameters in the joint prior can also be defined in the `inla.rgeneric.define()` call, by adding arguments `a=1`, `b=1` and so on.

Note that `log.prior()` must return the log prior for θ, and not the prior for the more natural parameters defined in `interpret.theta()`.

Function `initial()`

This function returns the initial values for θ, like

```
initial = function() {
    return (rep(1, 2))
}
```

or `numeric(0)` is there are no θ's. For a precision parameters it is generally advisable to chose a high precision as the initial value, as this helps the optimizer. `INLA` generally use initial value 4 for log precisions.

Function `quit()`

This function is called when all the computations are done and before exit-ing the `C`-program. If there is some cleanup to do, you can do this here. In our example, there is nothing do to.

```
quit = function() {
    return (invisible())
}
```

Example of usage

Here is an example of use. The function `inla.rgeneric.ar1.model()` contains the functions given above, and can be used directly like this.

```
n = 100
rho=0.9
x = arima.sim(n, model = list(ar = rho)) * sqrt(1-rho^2)
y = x + rnorm(n, sd = 0.1)
model = inla.rgeneric.define(inla.rgeneric.ar1.model, n=n)
formula = y ~ -1 + f(idx, model=model)
r = inla(formula, data = data.frame(y, idx = 1:n))
```

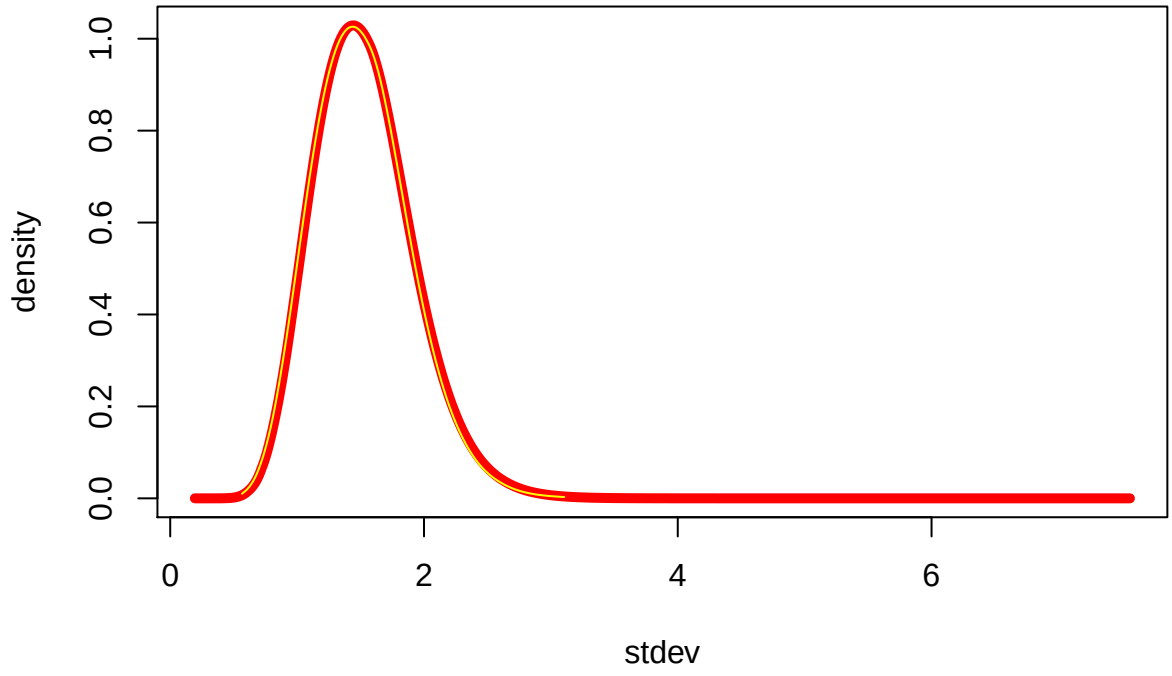
We can also compare with the buildt-in version, if we make sure to use the same priors

```
fformula = y ~ -1 +
    f(idx, model = "ar1",
      hyper = list(prec = list(prior = "loggamma", param = c(1,1)),
```

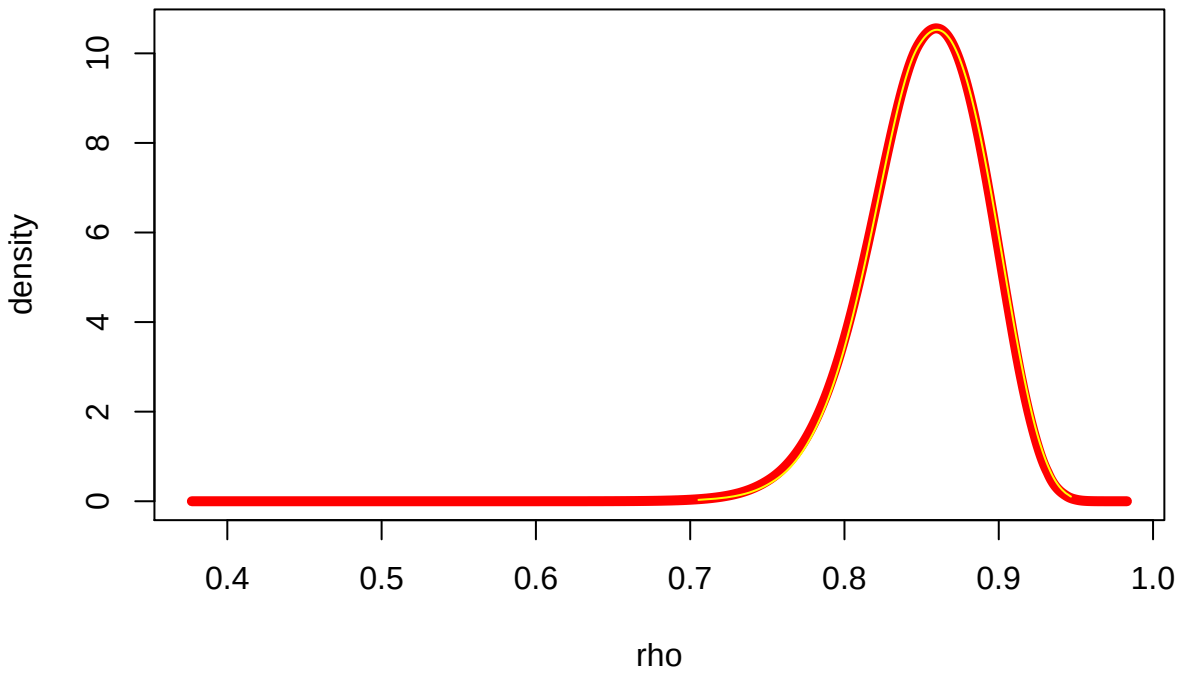
```
      rho = list(prior = "normal", param = c(0,1))))
rr = inla(fformula, data = data.frame(y, idx = 1:n))
```

and plot the hyperparameters in the same scale

```
plot(inla.smarginal(rr$marginals.hyperpar[[2]]),
     type="l", lwd=5, col="red", xlab="stdev", ylab="density")
lines(inla.tmarginal(exp, r$internal.marginals.hyperpar[[2]]),
     col="yellow")
```




```
plot(inla.smarginal(rr$marginals.hyperpar[[3]]),
     type="l", lwd=5, col="red", xlab="rho", ylab="density")
lines(inla.tmarginal(function(x) 2*exp(x)/(1+exp(x))-1,
                     r$internal.marginals.hyperpar[[3]]),
     col="yellow")
```



The running time will of'course be quite different

```
round(rbind(native = rr$cpu.used,
            rgeneric = r$cpu.used), digits = 3)
```

```
##            Pre Running  Post Total
## native   2.376   0.461 1.632 4.469
## rgeneric 1.931   3.287 0.443 5.661
```

Example: The iid-model

The following function defines the `iid`-model, see `inla.doc("iid")`, which we give without further comments. To run this model in `R`, you may run `demo(rgeneric)`.

```
inla.rgeneric.iid.model
```

```
## function (cmd = c("graph", "Q", "mu", "initial", "log.norm.const",
##     "log.prior", "quit"), theta = NULL)
## {
##     envir <- parent.env(environment())
##     interpret.theta <- function() {
##         return(list(prec = exp(theta[1L])))
##     }
##     graph <- function() {
##         G <- Diagonal(n, x = rep(1, n))
##         return(G)
##     }
##     Q <- function() {
```

```
##         prec <- interpret.theta()$prec
##         Q <- Diagonal(n, x = rep(prec, n))
##         return(Q)
##     }
##     mu <- function() {
##         return(numeric(0))
##     }
##     log.norm.const <- function() {
##         prec <- interpret.theta()$prec
##         val <- sum(dnorm(rep(0, n), sd = 1/sqrt(prec), log = TRUE))
##         return(val)
##     }
##     log.prior <- function() {
##         prec <- interpret.theta()$prec
##         val <- dgamma(prec, shape = 1, rate = 1, log = TRUE) +
##             theta[1L]
##         return(val)
##     }
##     initial <- function() {
##         ntheta <- 1
##         return(rep(1, ntheta))
##     }
##     quit <- function() {
##         return(invisible())
##     }
##     if (!length(theta)) {
##         theta <- initial()
##     }
##     val <- do.call(match.arg(cmd), args = list())
##     return(val)
## }
## <environment: 0x55837aeff520>
```

Example: A model for the mean structure

Up to now, we have assumed zero mean. In this example, we will illustrate how to add a non-zero mean model, focusing on the mean model only. We can of'course have a mean model and a non-trivial precision matrix together.

```
## In this example we do linear regression using 'rgeneric'.
## The regression model is y = a + b*x + noise,  and we
## define 'a + b*x + tiny.noise' as a latent model.
## The dimension is length(x) and number of hyperparameters
## is 2 ('a' and 'b').
```

```
rgeneric.linear.regression =
    function(cmd = c("graph", "Q", "mu", "initial", "log.norm.const",
                     "log.prior", "quit"),
             theta = NULL)
{
    envir = parent.env(environment())

    ## artifical high precision to be added to the mean-model
    prec.high = exp(15)
```

```
    interpret.theta = function() {
        return(list(a = theta[1L], b = theta[2L]))
    }

    graph = function() {
        G = Diagonal(n = length(x), x=1)
        return(G)
    }

    Q = function() {
        Q = prec.high * graph()
        return(Q)
    }

    mu = function() {
        par = interpret.theta()
        return(par$a + par$b * x)
    }

    log.norm.const = function() {
        return(numeric(0))
    }

    log.prior = function() {
        par = interpret.theta()
        val = (dnorm(par$a, mean=0, sd=1, log=TRUE) +
               dnorm(par$b, mean=0, sd=1, log=TRUE))
        return(val)
    }

    initial = function() {
        return(rep(0, 2))
    }

    quit = function() {
        return(invisible())
    }

    val = do.call(match.arg(cmd), args = list())
    return(val)
}
```

and we can run this as

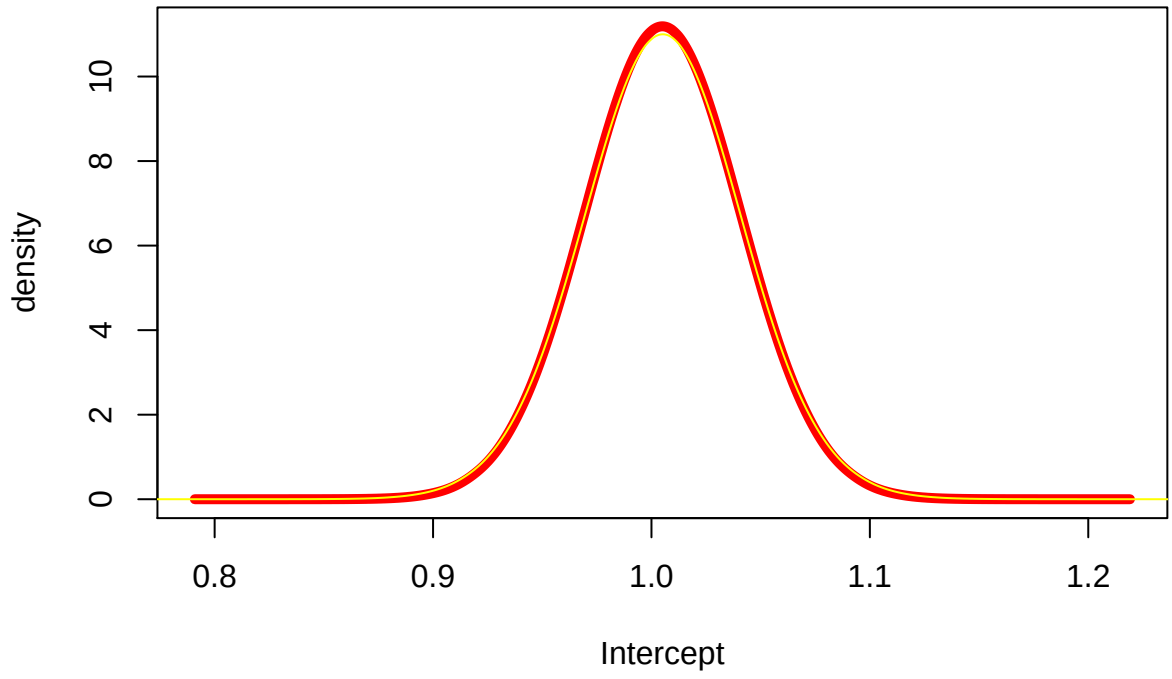
```
a = 1
b = 2
n = 50
x = rnorm(n)
eta = a + b*x
s = 0.25
y = eta + rnorm(n, sd=s)

rgen = inla.rgeneric.define(model = rgeneric.linear.regression, x=x)
r = inla(y ~ -1 + f(idx, model=rgen),
```

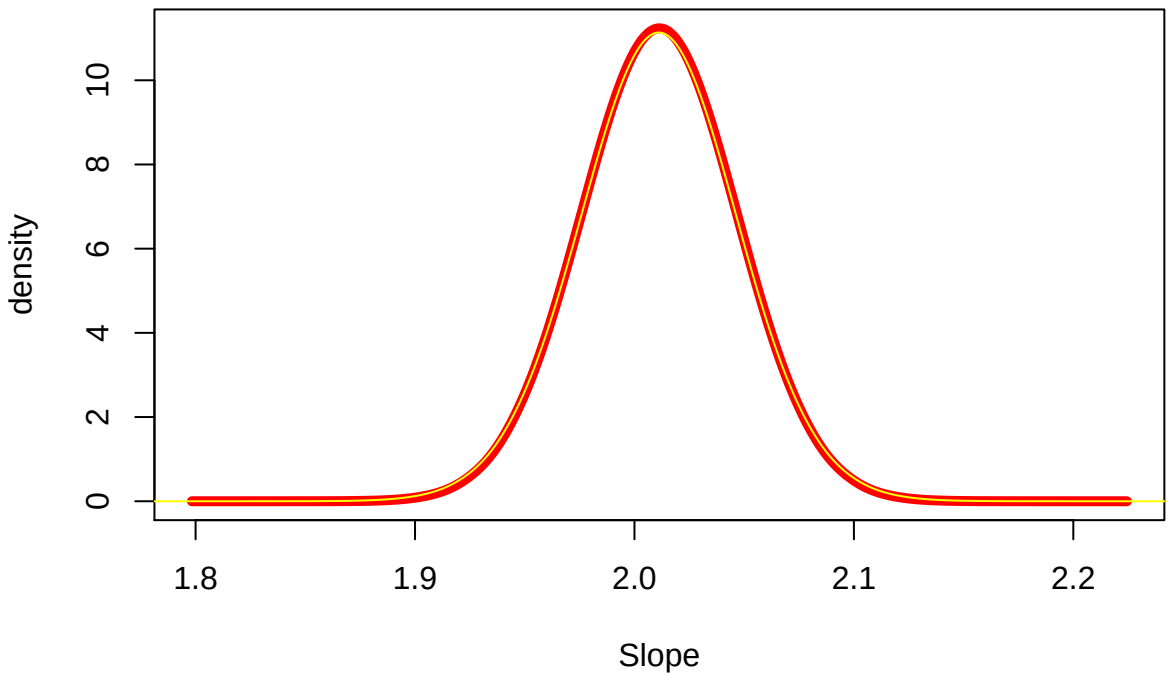
```
          data = data.frame(y, idx = 1:n))
rr = inla(y ~ 1 + x,
          data = data.frame(y, x),
          control.fixed = list(prec.intercept = 1, prec = 1))
```

and we can compare the results with the native model

```
plot(inla.smarginal(r$marginals.hyperpar[['Theta1 for idx']]),
     type="l", lwd=5, col="red", xlab="Intercept", ylab="density")
lines(inla.smarginal(rr$marginals.fixed$'(Intercept)'), col="yellow")
```




```
plot(inla.smarginal(r$marginals.hyperpar[['Theta2 for idx']]),
     type="l", lwd=5, col="red", xlab="Slope", ylab="density")
lines(inla.smarginal(rr$marginals.fixed$x), col="yellow")
```

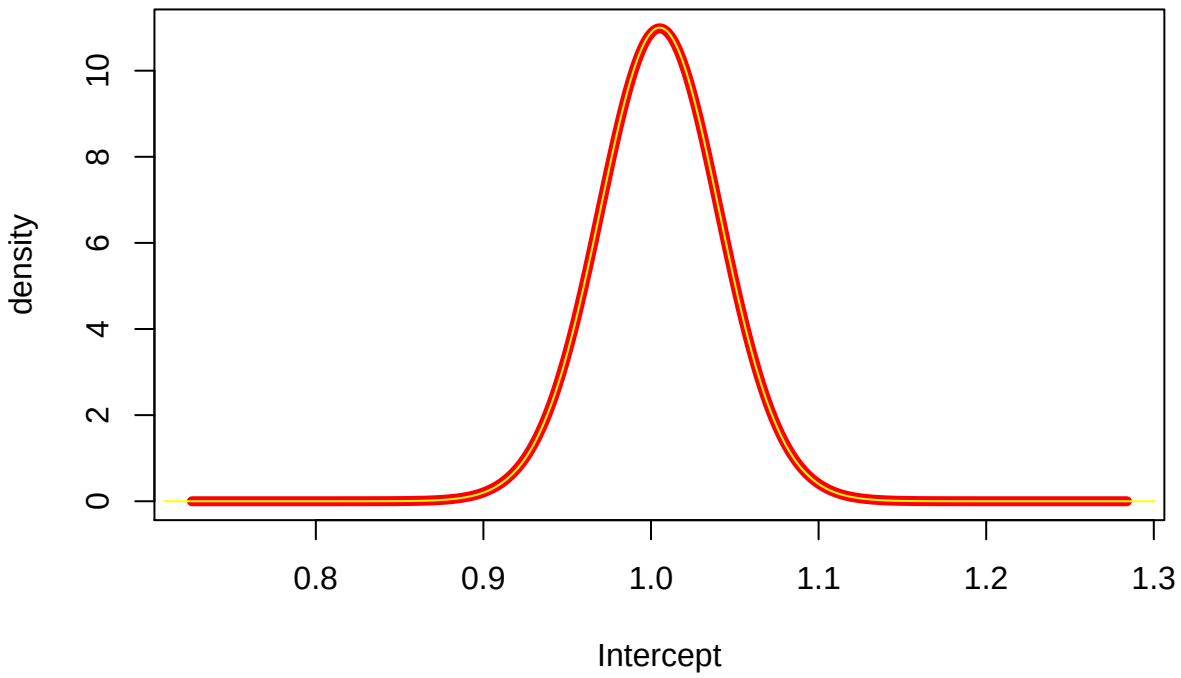


The tiny in-accuacy is due to treatment of `a` and `b` as hyperparameters in the `rgeneric` model. We can improve the estimates using `inla.hyperpar()` as usual,

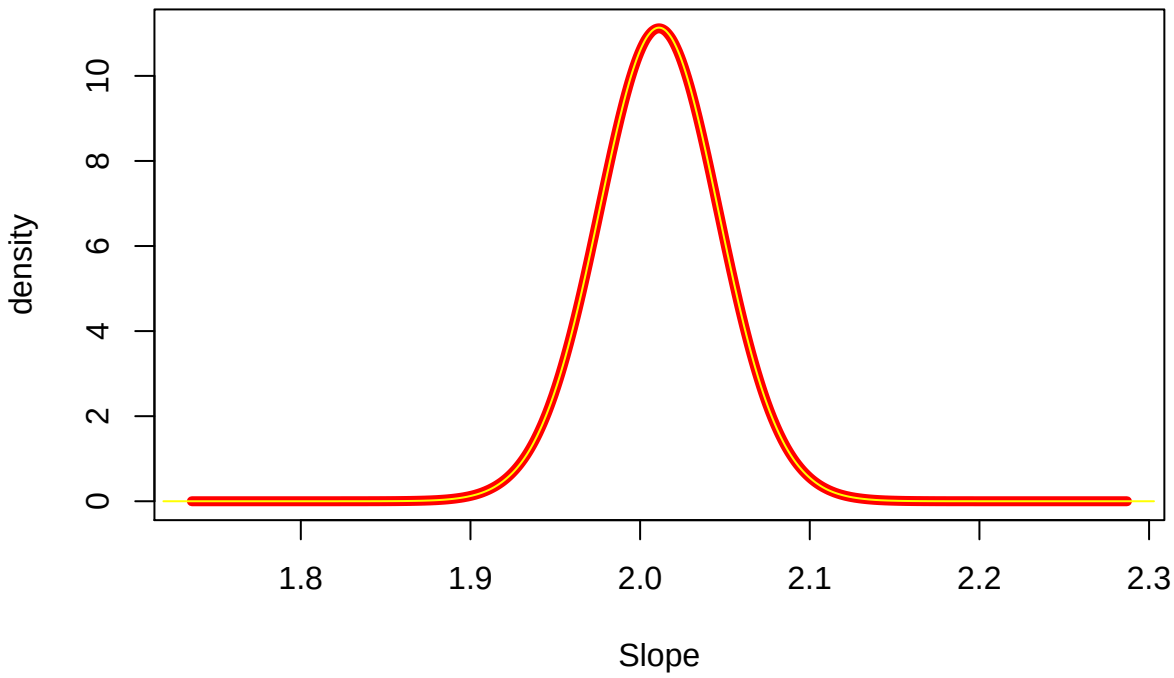
```
r = inla.hyperpar(r)
```

Replotting the results shows improvement:

```
plot(inla.smarginal(r$marginals.hyperpar[['Theta1 for idx']]),
     type="l", lwd=5, col="red", xlab="Intercept", ylab="density")
lines(inla.smarginal(rr$marginals.fixed$'(Intercept)'), col="yellow")
```




```
plot(inla.smarginal(r$marginals.hyperpar[['Theta2 for idx']]),
     type="l", lwd=5, col="red", xlab="Slope", ylab="density")
lines(inla.smarginal(rr$marginals.fixed$x), col="yellow")
```



Some comments on optimization

The rgeneric-interface is not ment to be a replacement for implementing a model component in C, but rather a tool to experiment with new models and adding case spesific model components. Needless to say, it will be a somewhat slower than a model that is implemented in C. For smaller problems the overhead is relative larger than for larger problems, since more less time is used to factorize matrices etc, compared to constructing the matrices.

To discuss some easy steps that can be taken, we can consider this simple Gaussian model with zero mean and precision matrix

$$Q = \tau R.$$

We observe a sample from this matrix with known Gaussian noise.

The rgeneric implementation of this, could be as follows.

```
rgeneric.test = function(
    cmd = c("graph", "Q", "mu", "initial", "log.norm.const", "log.prior", "quit"),
    theta = NULL)
{
    envir = parent.env(environment())

    graph = function() {
        return(matrix(1, n, n))
    }

    Q = function() {
```

```
        R <- matrix(sin(1:n^2), n, n)
        R <- R %*% t(R)
        diag(R) <- diag(R)+1
        Q <- exp(theta[1]) * R
        return(Q)
    }

    mu = function() return (numeric(0))

    log.norm.const = function() {
        return (numeric(0))
    }

    log.prior = function() {
        return (dgamma(exp(theta[1]), shape = 1, rate = 1, log=TRUE) + theta[1])
    }

    initial = function() {
        return(4)
    }

    if (!length(theta)) theta = initial()
    val = do.call(match.arg(cmd), args = list())

    return (val)
}
```

We can now simulate some data and compare the results with the buildt-in implementation

```
n = 200
s = .1
Q <- rgeneric.test("Q", theta = 0)
library(mvtnorm)
S <- solve(as.matrix(Q))
S <- (S + t(S))/2
x <- drop(rmvnorm(1, sigma = S))
y <- x + rnorm(n, sd = s)
cont.family = list(hyper = list(prec = list(initial=log(1/s^2), fixed=TRUE)))

r1 = inla(y ~ -1 + f(idx, model="generic", Cmatrix = Q,
                     hyper = list(prec = list(prior = "loggamma", param = c(1, 1)))),
          data = data.frame(y = y, idx = 1:n), control.family = cont.family)
ld <- 0.5 * log(det(as.matrix(Q)))
r1$mlik <- r1$mlik + ld ## see the documentation for why

model2 = inla.rgeneric.define(rgeneric.test, n=n, optimize = FALSE)
r2 = inla(y ~ -1 + f(idx, model=model2),
          data = data.frame(y = y, idx = 1:n), control.family = cont.family)
```

We can compare the results, with

```
r2$mlik - r1$mlik
```

```
##                                                [,1]
## log marginal-likelihood (integration) -1.820604e-06
```

```
## log marginal-likelihood (Gaussian)     -5.630299e-05
```

there are a couple of things that can be done in order to improve the speed of the rgeneric model. The first is to *cache* intermediate calculations and to make sure the same calculations are not done over and over again.

For this, we use the rgeneric function's environment. We can cache the matrix R and also precompute large parts of the normalizing constant.

```
rgeneric.test.opt.1 = function(
    cmd = c("graph", "Q", "mu", "initial", "log.norm.const", "log.prior", "quit"),
    theta = NULL)
{
    envir = parent.env(environment())

    if (!exists("cache.done", envir = envir)) {
        R <- matrix(sin(1:n^2), n, n)
        R <- R %*% t(R)
        diag(R) <- diag(R)+1
        R.logdet <- log(det(R))
        R <- inla.as.sparse(R)
        idx <- which(R@i <= R@j)
        R@i <- R@i[idx]
        R@j <- R@j[idx]
        R@x <- R@x[idx]
        assign("R", R, envir = envir)
        norm.const <- -n/2 * log(2*pi) + 0.5 * R.logdet
        assign("norm.const", norm.const, envir = envir)
        assign("cache.done", TRUE, envir = envir)
    }

    graph = function() {
        return (R)
    }

    Q = function() {
        return(exp(theta[1]) * R)
    }

    mu = function() return (numeric(0))

    log.norm.const = function() {
        return (norm.const + n/2 * theta[1])
    }

    log.prior = function() {
        return (dgamma(exp(theta[1]), shape = 1, rate = 1, log=TRUE) + theta[1])
    }

    initial = function() {
        return(4)
    }

    if (!length(theta)) theta = initial()
    val = do.call(match.arg(cmd), args = list())
```

```
    return (val)
}
```

We can also go one step further, to add option `optimize=TRUE` when calling `inla.rgeneric.define`, which inform the interpreter that we pass only the matrix values of Q, not the indices! This impose a constraint on the ordering, which must be the that is defined after converting the matrix to `inla.as.sparse` and returning only the upper triangular part. This is a row-based ordering, like

```
A=matrix(1:9,3,3)
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
inla.as.sparse(A)@x
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

The updated model will then be

```
rgeneric.test.opt.2 = function(
    cmd = c("graph", "Q", "mu", "initial", "log.norm.const", "log.prior", "quit"),
    theta = NULL)
{
    envir = parent.env(environment())

    if (!exists("cache.done", envir = envir)) {
        R <- matrix(sin(1:n^2), n, n)
        R <- R %*% t(R)
        diag(R) <- diag(R)+1
        R.logdet <- log(det(R))
        R <- inla.as.sparse(R)
        idx <- which(R@i <= R@j)
        R@i <- R@i[idx]
        R@j <- R@j[idx]
        R@x <- R@x[idx]
        assign("R", R, envir = envir)
        norm.const <- -n/2 * log(2*pi) + 0.5 * R.logdet
        assign("norm.const", norm.const, envir = envir)
        assign("cache.done", TRUE, envir = envir)
    }

    graph = function() {
        return (R)
    }

    Q = function() {
        ## since R was created with 'inla.sparse.matrix' above, the indices are sorted in a
        ## spesific order. This ordering is REQUIRED for R@x to be interpreted correctly.
        return(exp(theta[1]) * R@x)
    }

    mu = function() return (numeric(0))
```

```
    log.norm.const = function() {
        return (norm.const + n/2 * theta[1])
    }

    log.prior = function() {
        return (dgamma(exp(theta[1]), shape = 1, rate = 1, log=TRUE) + theta[1])
    }

    initial = function() {
        return(4)
    }

    if (!length(theta)) theta = initial()
    val = do.call(match.arg(cmd), args = list())

    return (val)
}
```

We can now run the two optimized variants

```
model3 = inla.rgeneric.define(rgeneric.test.opt.1, n=n, optimize = FALSE)
r3 = inla(y ~ -1 + f(idx, model=model3),
          data = data.frame(y = y, idx = 1:n), control.family = cont.family)

model4 = inla.rgeneric.define(rgeneric.test.opt.2, n=n, optimize = TRUE)
r4 = inla(y ~ -1 + f(idx, model=model4),
          data = data.frame(y = y, idx = 1:n), control.family = cont.family)
```

We can now compare the time and results for all models

```
print(r2$mlik - r1$mlik)
```

```
##                                                [,1]
## log marginal-likelihood (integration) -1.820604e-06
## log marginal-likelihood (Gaussian)    -5.630299e-05
```

```
print(r3$mlik - r1$mlik)
```

```
##                                                [,1]
## log marginal-likelihood (integration) -2.030502e-06
## log marginal-likelihood (Gaussian)    -3.215488e-05
```

```
print(r4$mlik - r1$mlik)
```

```
##                                                [,1]
## log marginal-likelihood (integration) -2.030502e-06
## log marginal-likelihood (Gaussian)    -3.215488e-05
```

```
print(rbind(native = r1$cpu[2],
            rgeneric.plain = r2$cpu[2],
            rgeneric.cache = r3$cpu[2],
            rgeneric.optimze = r4$cpu[2]))
```

```
##                   Running
## native            8.50976
## rgeneric.plain   17.18970
## rgeneric.cache   12.12330
```

```
## rgeneric.optimze 11.47230
```

Another general approach to optimizing R-code, is to write critical parts in C or C++. This can also be done here, by calling C-code within the rgeneric model.