Defining a latent model in R or C: The rgeneric and cgeneric model

Haavard Rue (hrue@r-inla.org)

Dec 9th 2021

Introduction

This is a class of generic models allows the user to define latent model component in R or C, for cases where the requested model is not yet implemented in INLA. The model component implemented in R will run slower, whereas the model component implemented in C will run just slightly slower, compared to a similar model implemented in INLA,

We will first describe the more accessible R interface, then at the end, the C interface as it is buildt on the same ideas.

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.

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

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))
}
```

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)
```

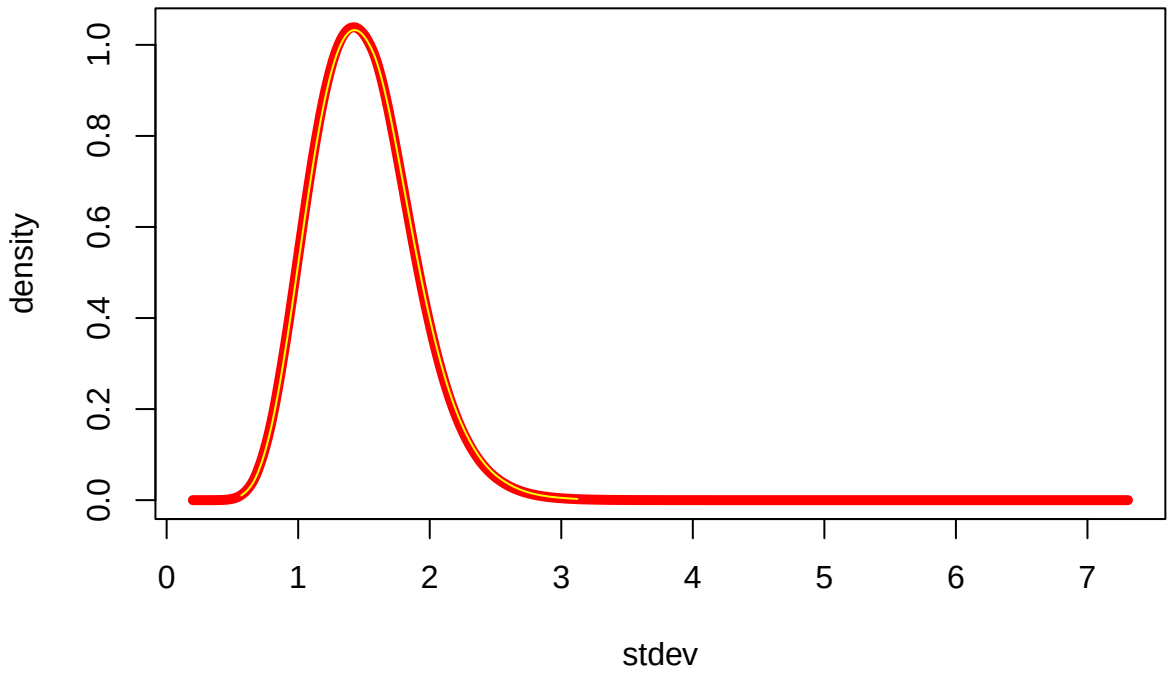
```
y = x + rnorm(n, sd = 0.1)
model = inla.rgeneric.define(inla.rgeneric.ar1.model, n=n)
formula = y ~ -1 + f(idx, model=model)
r = inla(formula, data = data.frame(y, idx = 1:n))
```

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

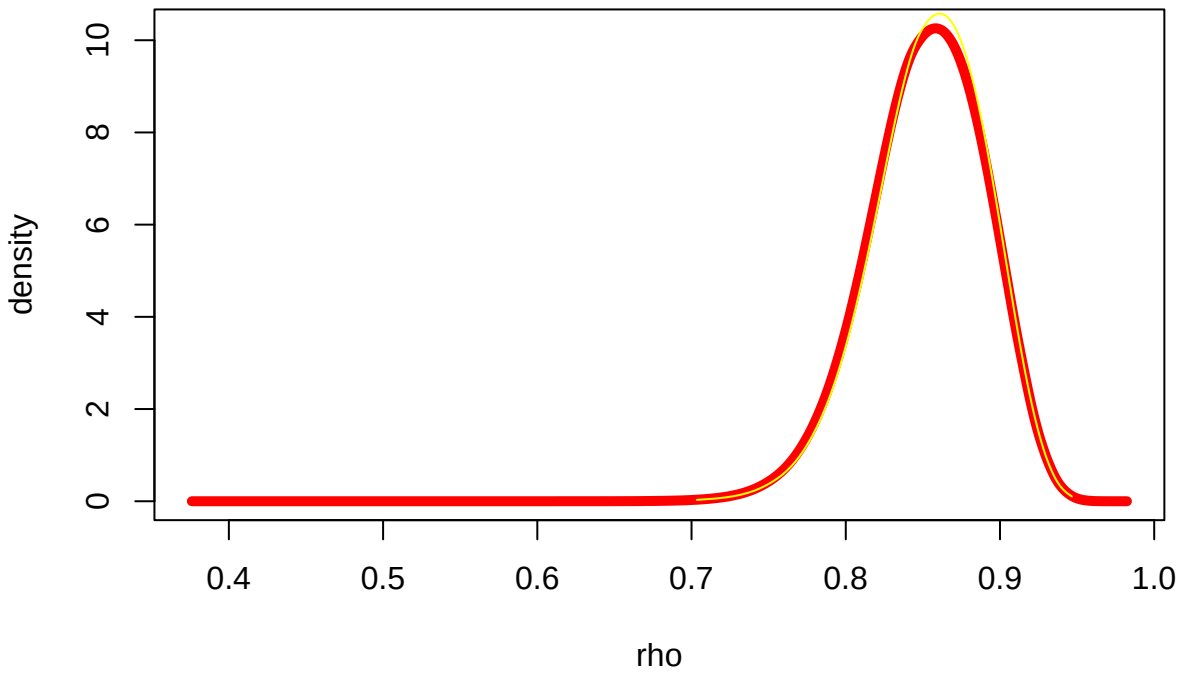
```
fformula = y ~ -1 +
    f(idx, model = "ar1",
      hyper = list(prec = list(prior = "loggamma", param = c(1,1)),
      rho = list(prior = "normal", param = c(0,1))))
rr = inla(fformula, data = data.frame(y, idx = 1:n))
```

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   0.632   0.380 0.018 1.030
## rgeneric 0.883   2.686 0.077 3.646
```

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)
## }
## <bytecode: 0x55fd5ffb9660>
## <environment: namespace:INLA>
```

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)
}
```

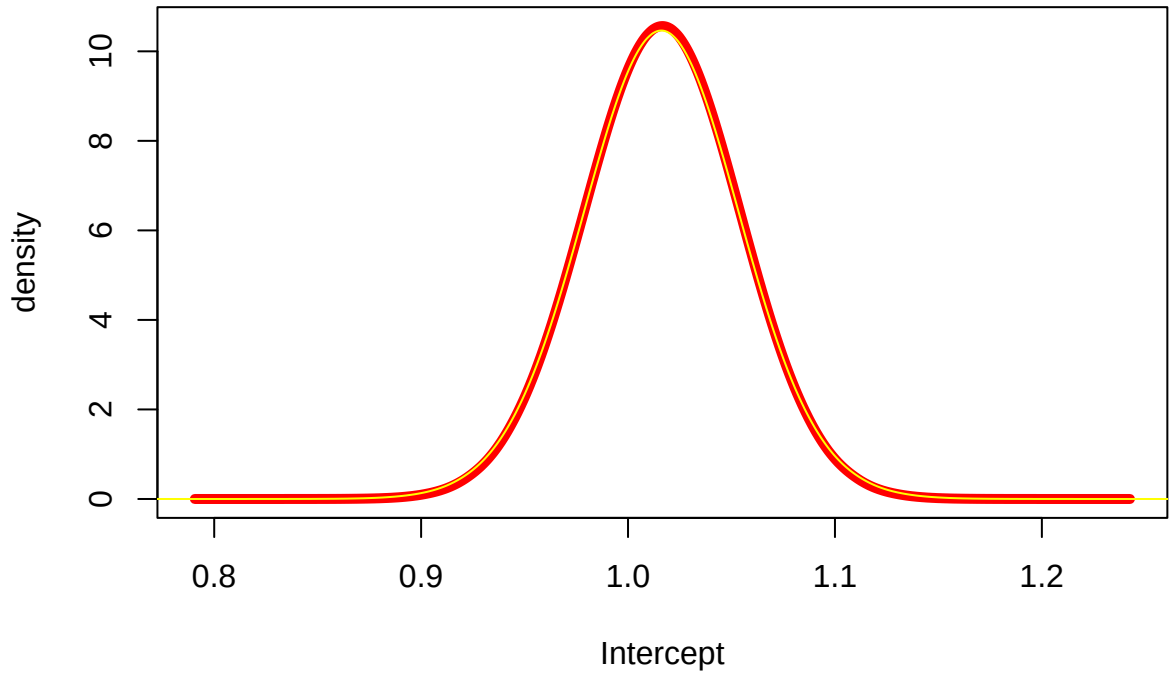
and we can run this as

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

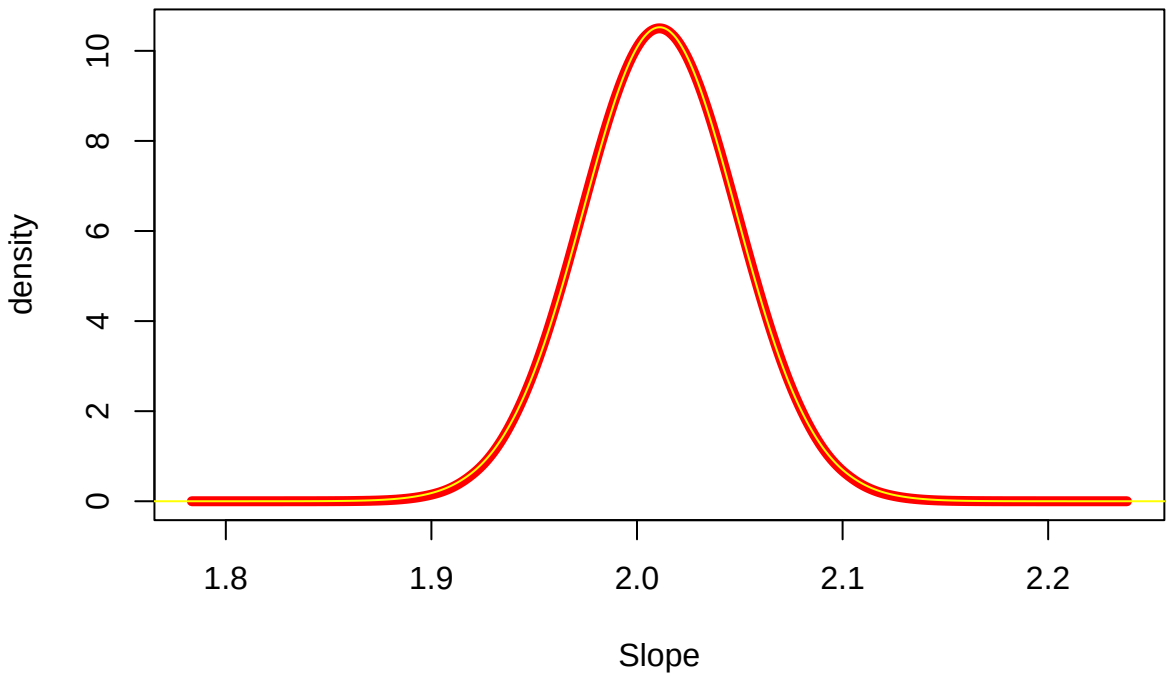
```
rgen = inla.rgeneric.define(model = rgeneric.linear.regression, x=x)
r = inla(y ~ -1 + f(idx, model=rgen),
         data = data.frame(y, idx = 1:n))
rr = inla(y ~ 1 + x,
          data = data.frame(y, x),
          control.fixed = list(prec.intercept = 1, prec = 1))
```

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")
```

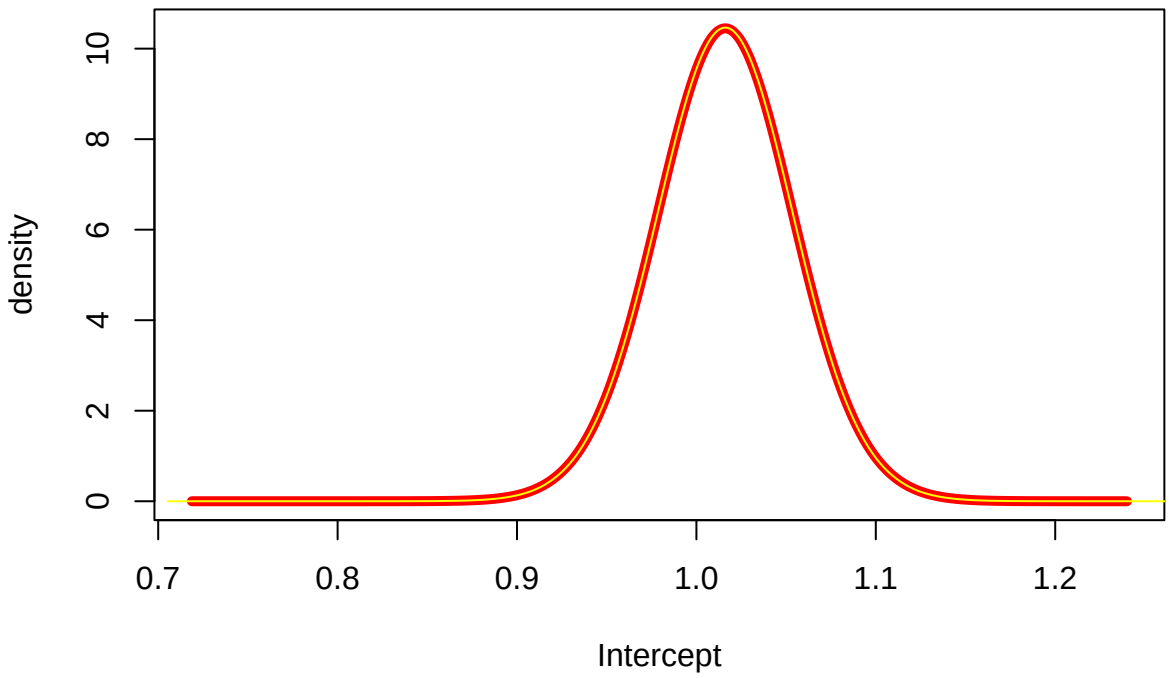


The tiny in-accuacy is due to treatment of `a` and `b` as hyperparameters in the `rgeneric` model. We can improve the estimates using `inla.hyperpar()` as usual,

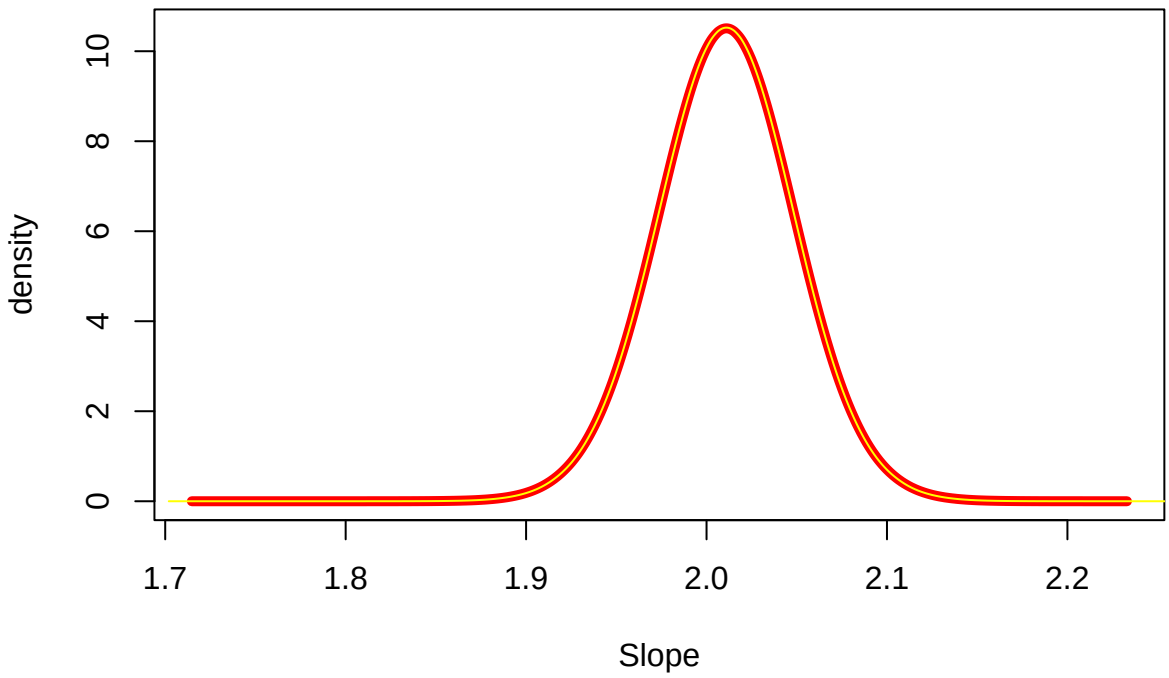
```
r = inla.hyperpar(r)
```

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) 2.972547e-06
```

```
## log marginal-likelihood (Gaussian)    8.241993e-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) 2.972547e-06
## log marginal-likelihood (Gaussian)    8.241993e-05
```

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

```
##                                                [,1]
## log marginal-likelihood (integration) 1.983920e-06
## log marginal-likelihood (Gaussian)    5.932357e-05
```

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

```
##                                                [,1]
## log marginal-likelihood (integration) 1.983920e-06
## log marginal-likelihood (Gaussian)    5.932357e-05
```

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

```
##                  Running
## native           4.512485
## rgeneric.plain   9.718990
## rgeneric.cache   6.092870
```

```
## rgeneric.optimze 5.407536
```

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.

The Cgeneric interface

The new interface in C (as of Dec 2021) allows one to do similar as described above using C code. The advantage is mainly speed but that benefite could be dramatic, as the R interpreter is avoided and the calls to the generic functions are no longer wrapped in an OpenMP critical region which is required due to the serial nature of libR.

This new feature is necessarily much more technical, but is relative straight forward on Linux and Mac. For Windows its currently not tested.

The main idea is to provide a C implementation with similar features as for the R interface, and to pass the name of the function and its binary in a form of a shared/dynamic library. The GNU library `ltdl` is used for cross-platform compatabilty, see `https://www.gnu.org/software/libtool/manual/html_node/Using-libltdl.html`.

Example of usage

In short, the usage is as follows. First compile and build a shared object

```
gcc -Wall -fpic -g -O -c -o cgeneric-demo.o cgeneric-demo.c
gcc -shared -o cgeneric-demo.so cgeneric-demo.o
```

then use this file to define the `cgeneric` model,

```
cmodel <- inla.cgeneric.define(model = "inla_cgeneric_iid_model",
                               shlib = "cgeneric-demo.so", n = n)
```

where `n` is the size of the model. Unfortunately, this needs to be known before the model is loaded.

The usage is similar to `rgeneric`, as

```
rc <- inla(
    y ~ -1 + f(idx, model = cmodel),
    data = data.frame(y, idx = 1:n),
    control.family = list(hyper = list(prec = list(initial = 12, fixed = TRUE))))
```

The following example implements the same model using the builtin model, `rgeneric` and `cgeneric`.

```
n <- 100
y <- rnorm(n)

r <- inla(
    y ~ -1 + f(idx, model = "iid", hyper = list(prec = list(param = c(1, 1)))),
    data = data.frame(y, idx = 1:n),
    control.family = list(hyper = list(prec = list(initial = 12, fixed = TRUE))))

rmodel <- inla.rgeneric.define(inla.rgeneric.iid.model, n = n)
rr <- inla(
    y ~ -1 + f(idx, model = rmodel),
    data = data.frame(y, idx = 1:n),
    control.family = list(hyper = list(prec = list(initial = 12, fixed = TRUE))))

cmodel <- inla.cgeneric.define(model = "inla_cgeneric_iid_model",
                               shlib = "cgeneric-demo.so", n = n)
```

```
rc <- inla(
    y ~ -1 + f(idx, model = cmodel),
    data = data.frame(y, idx = 1:n),
    control.family = list(hyper = list(prec = list(initial = 12, fixed = TRUE))))

print(cbind(r$mlik, rr$mlik-r$mlik, rc$mlik-r$mlik))
print(cbind(r$cpu[2], rr$cpu[2], rc$cpu[2]))
```

Header file and example file

The header-file `cgeneric.h` is included in the package in the `include` directory, and in the `cgeneric` directory, the header-file and the example file given above is included.

The header file `cgeneric.h` needs to be included in any implementation as it defines the data types and definitions needed, like

```
typedef enum {
    INLA_CGENERIC_VOID = 0,
    INLA_CGENERIC_Q,
    INLA_CGENERIC_GRAPH,
    INLA_CGENERIC_MU,
    INLA_CGENERIC_INITIAL,
    INLA_CGENERIC_LOG_NORM_CONST,
    INLA_CGENERIC_LOG_PRIOR,
    INLA_CGENERIC_QUIT
} inla_cgeneric_cmd_tp;
```

to define the various actions.

Example `iid`

It is easier to see how this is one by a simple `iid-model` example

```
#include "cgeneric.h"
#define Calloc(n_, type_)  (type_ *)calloc((n_), sizeof(type_))
#define SQR(x) ((x)*(x))

double *inla_cgeneric_iid_model(inla_cgeneric_cmd_tp cmd,
    double *theta, inla_cgeneric_data_tp * data)
{
    // this reimplement `inla.rgeneric.iid.model` using cgeneric

    double *ret = NULL, prec = (theta ? exp(theta[0]) : NAN),
        lprec = (theta ? theta[0] : NAN);

    assert(!strcasecmp(data->ints[0]->name, "n")); // this will always be the case
    int N = data->ints[0]->ints[0];            // this will always be the case
    assert(N > 0);

    switch (cmd) {
    case INLA_CGENERIC_VOID:
    {
        assert(!(cmd == INLA_CGENERIC_VOID));
        break;
    }
```

```
case INLA_CGENERIC_GRAPH:
{
    ret = Calloc(2 + 2 * N, double);
    ret[0] = N;                     /* dimension */
    ret[1] = N;                     /* number of (i <= j) */
    for (int i = 0; i < N; i++) {
        ret[2 + i] = i;                 /* i */
        ret[2 + N + i] = i;             /* j */
    }
    break;
}

case INLA_CGENERIC_Q:
{
    if (1) {
        // optimized format
        ret = Calloc(2 + N, double);
        ret[0] = -1;                    /* code for optimized output */
        ret[1] = N;                 /* number of (i <= j) */
        for (int i = 0; i < N; i++) {
            ret[2 + i] = prec;
        }
    } else {
        // plain format, but the optimized format above is better to use
        ret = Calloc(2 + 3 * N, double);
        ret[0] = N;
        ret[1] = N;
        for (int i = 0; i < N; i++) {
            ret[2 + i] = i;             /* i */
            ret[2 + N + i] = i;         /* j */
            ret[2 + 2 * N + i] = prec;      /* Q_ij */
        }
    }
    break;
}

case INLA_CGENERIC_MU:
{
    ret = Calloc(1, double);
    ret[0] = 0;
    break;
}

case INLA_CGENERIC_INITIAL:
{
    ret = Calloc(2, double);
    ret[0] = 1;
    ret[1] = 4.0;
    break;
}

case INLA_CGENERIC_LOG_NORM_CONST:
{
```

```
        ret = Calloc(1, double);
        ret[0] = N * (-0.9189385332 + 0.5 * lprec);
        break;
    }

    case INLA_CGENERIC_LOG_PRIOR:
    {
        // prec ~ gamma(1,1)
        ret = Calloc(1, double);
        ret[0] = -prec + lprec;
        break;
    }

    case INLA_CGENERIC_QUIT:
    default:
        break;
    }

    return (ret);
}
```

The return values must be allocated dynamically, and is free'ed in the main program after use.

Example `ar1`

The reimplementation of the AR1 model as in `inla.rgeneric.ar1.model` is as follows.

```
#include "cgeneric.h"
#define Calloc(n_, type_)  (type_ *)calloc((n_), sizeof(type_))
#define SQR(x) ((x)*(x))

double *inla_cgeneric_ar1_model(inla_cgeneric_cmd_tp cmd, double *theta,
    inla_cgeneric_data_tp * data)
{
    // this reimplement `inla.rgeneric.ar1.model` using cgeneric

    double *ret = NULL, prec, lprec, rho, rho_intern;

    if (theta) {
        lprec = theta[0];
        prec = exp(lprec);
        rho_intern = theta[1];
        rho = 2.0 * exp(rho_intern) / (1.0 + exp(rho_intern)) - 1.0;
    } else {
        prec = lprec = rho = rho_intern = NAN;
    }

    assert(!strcasecmp(data->ints[0]->name, "n")); // this will always be the case
    int N = data->ints[0]->ints[0];            // this will always be the case
    assert(N > 0);

    switch (cmd) {
    case INLA_CGENERIC_VOID:
    {
```

```
    assert(!(cmd == INLA_CGENERIC_VOID));
    break;
}

case INLA_CGENERIC_GRAPH:
{
    int m = N + N - 1, offset, i, k;
    ret = Calloc(2 + 2 * m, double);

    offset = 2;
    ret[0] = N;                     /* dimension */
    ret[1] = m;                     /* number of (i <= j) */
    for (k = i = 0; i < N; i++) {
        ret[offset + k] = i;                /* i */
        ret[offset + m + k++] = i;          /* j */
        if (i < N - 1) {
            ret[offset + k] = i;            /* i */
            ret[offset + m + k++] = i + 1; /* j */
        }
    }
    break;
}

case INLA_CGENERIC_Q:
{
    double param = prec / (1.0 - SQR(rho));
    int m = N + N - 1;
    int offset, i, k;
    ret = Calloc(2 + m, double);

    // use optimized format.
    // The order of Q_ij s are then predetermined as the upper triangular of Q:
    //
    // for(i=0; i < n; i++)
    //     for(j=i; j<n; j++)
    //         ...
    //
    // but for only those (i,j)s that is defined in _GRAPH, of course

    offset = 2;
    ret[0] = -1;
    ret[1] = m;
    for (i = k = 0; i < N; i++) {
        ret[offset + k++] = param * (i == 0 || i == N - 1 ? 1.0 : (1.0 + SQR(rho)));
        if (i < N - 1) {
            ret[offset + k++] = -param * rho;
        }
    }
    break;
}

case INLA_CGENERIC_MU:
{
```

```
        ret = Calloc(1, double);
        ret[0] = 0;
        break;
    }

    case INLA_CGENERIC_INITIAL:
    {
        ret = Calloc(3, double);
        ret[0] = 2;
        ret[1] = 1.0;
        ret[2] = 1.0;
        break;
    }

    case INLA_CGENERIC_LOG_NORM_CONST:
    {
        double prec_innovation = prec / (1.0 - SQR(rho));
        ret = Calloc(1, double);
        ret[0] = N * (-0.5 * log(2.0 * M_PI) +
            0.5 * log(prec_innovation)) + 0.5 * log(1.0 - SQR(rho));
        break;
    }

    case INLA_CGENERIC_LOG_PRIOR:
    {
        ret = Calloc(1, double);
        ret[0] = -prec + lprec - 0.5 * log(2.0 * M_PI) - 0.5 * SQR(rho_intern);
        break;
    }

    case INLA_CGENERIC_QUIT:
    default:
        break;
    }

    return (ret);
}
```

Passing arguments to `inla.cgeneric.define`

The `R` interface is easier when it comes to passing arguments throughout the system, as any non-standard arguments to `inla.rgeneric.define` is stored in environment of the function and is then available when the function is evaluated in `libR`.

For the `C` interface, this has to be done manually, and pointer to a data structure that contains all arguments to `inla.cgeneric.define` is passed to the `C` function. The data structure is as follows

```
typedef struct {
    int n_ints;
    inla_cgeneric_vec_tp **ints;

    int n_doubles;
    inla_cgeneric_vec_tp **doubles;
```

```
    int n_chars;
    inla_cgeneric_vec_tp **chars;

    int n_mat;
    inla_cgeneric_mat_tp **mats;

    int n_smat;
    inla_cgeneric_smat_tp **smats;
} inla_cgeneric_data_tp;
```

where integers, doubles (or numerics), characters (or strings), dense matrices and sparse matrices are stored in named lists. The `n_ints` gives the number of integers or integer vectors, and similar with `n_doubles`, `n_chars`, `n_mat` and `n_smat`. The predefined named arguments in `inla.cmatrix.define` are stored first, then additional named arguments are stored. This implies that the argument `n` is always the first one in the integer list, like used in the examples above

```
    assert(!strcasecmp(data->ints[0]->name, "n")); // this will always be the case
    int N = data->ints[0]->ints[0];            // this will always be the case
```

The integers, doubles and characters, are stored in

```
typedef struct
{
    char *name;
    int len;
    int *ints;
    double *doubles;
    char *chars;
}
    inla_cgeneric_vec_tp;
```

so possible vectors with length `len` or a string with `len` characters.

Dense matrices are stored column wise, like default in `R`

```
/*
 *      matrix storage, stored column by column, like
 *      > matrix(1:6,2,3)
 *      [,1] [,2] [,3]
 *      [1,]    1    3    5
 *      [2,]    2    4    6
 *      > c(matrix(1:6,2,3))
 *      [1] 1 2 3 4 5 6
 */
typedef struct {
    char *name;
    int nrow;
    int ncol;
    double *x;
} inla_cgeneric_mat_tp;
```

and sparse matrices are stored as triplets `(i,j,x)`,

```
/*
 * sparse matrix format, stored used 0-based indices, like
 *
 *      > A <- inla.as.sparse(matrix(c(1,2,3,0,0,6),2,3))
```

```
 *      > A
 *      2 x 3 sparse Matrix of class "dgTMatrix"
 *      [1,] 1 3 .
 *      [2,] 2 . 6
 *      > cbind(i=A@i, j=A@j, x=A@x)
 *           i j x
 *      [1,] 0 0 1
 *      [2,] 1 0 2
 *      [3,] 0 1 3
 *      [4,] 1 2 6
 */
typedef struct {
    char *name;
    int nrow;
    int ncol;
    int n;                              /* number of triplets (i,j,x) */
    int *i;
    int *j;
    double *x;
} inla_cgeneric_smat_tp;
```

R-objects not of the above types cannot be handled, but one can always, for example, store them in a file and pass the filename.

Adding the option `debug=TRUE` to `inla.cgeneric.define` will turn on debug-output and the contents of `inla_cgeneric_data_tp * data` is displayed. Only use this option for small size problems as the output can be excessive.

Thread safe

The implementation of the `cgeneric` model, must be thread safe.