Defining a latent model in R or C

Haavard Rue (hrue@r-inla.org)

October 15, 2025

Defining models in R or C

Haavard Rue (hrue@r-inla.org)

August 18, 2025

Introduction

This document describes how to a user can define latent model component in `R` or `C`, for cases where the requested model is not 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.

In Aug-2025, we added the functionality for likelihood models as well for `C` only. (Otherwise it will simply be to slow.) The implementation is similar to a latent model and described at the end of this vignette.

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")
```

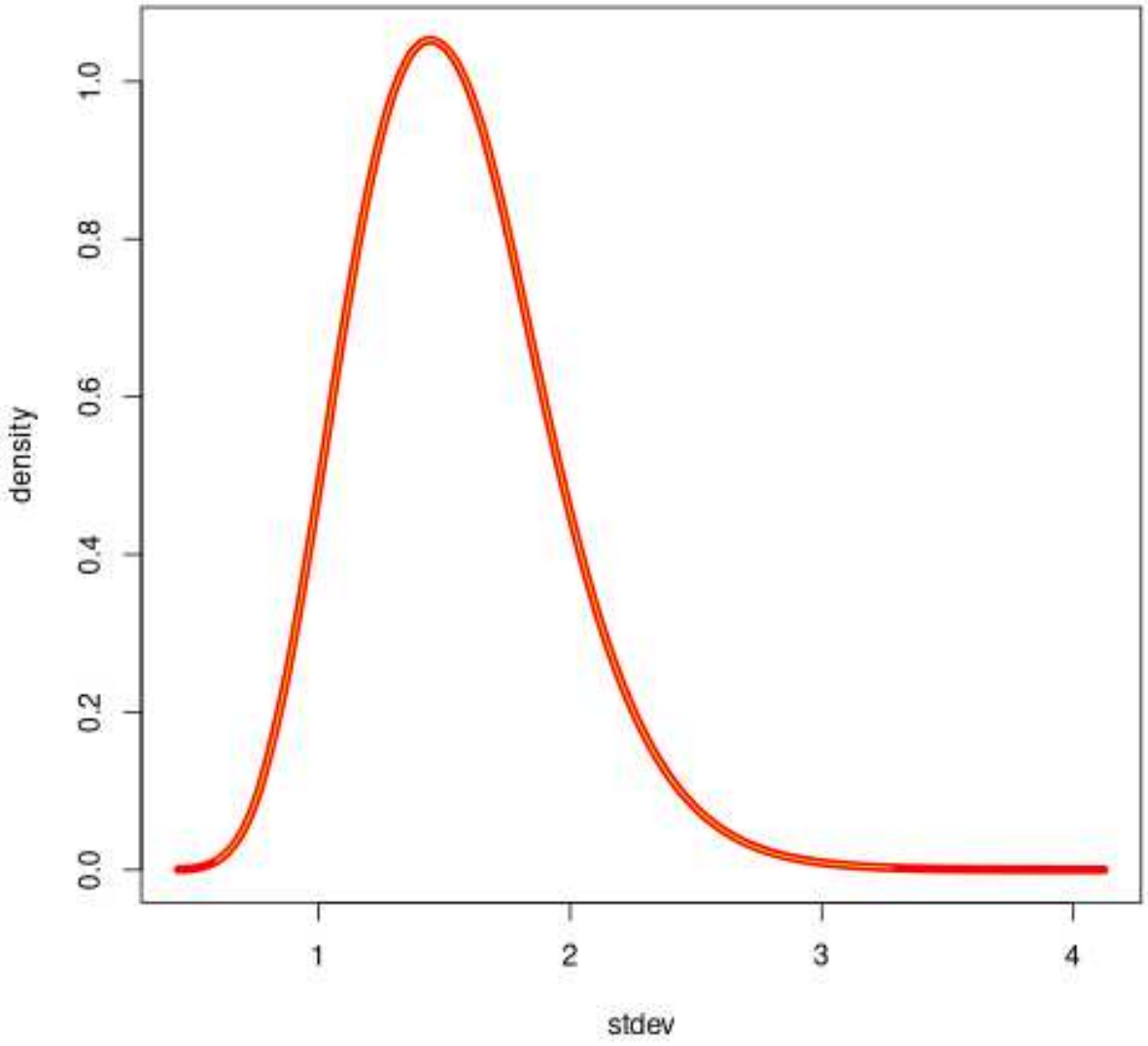
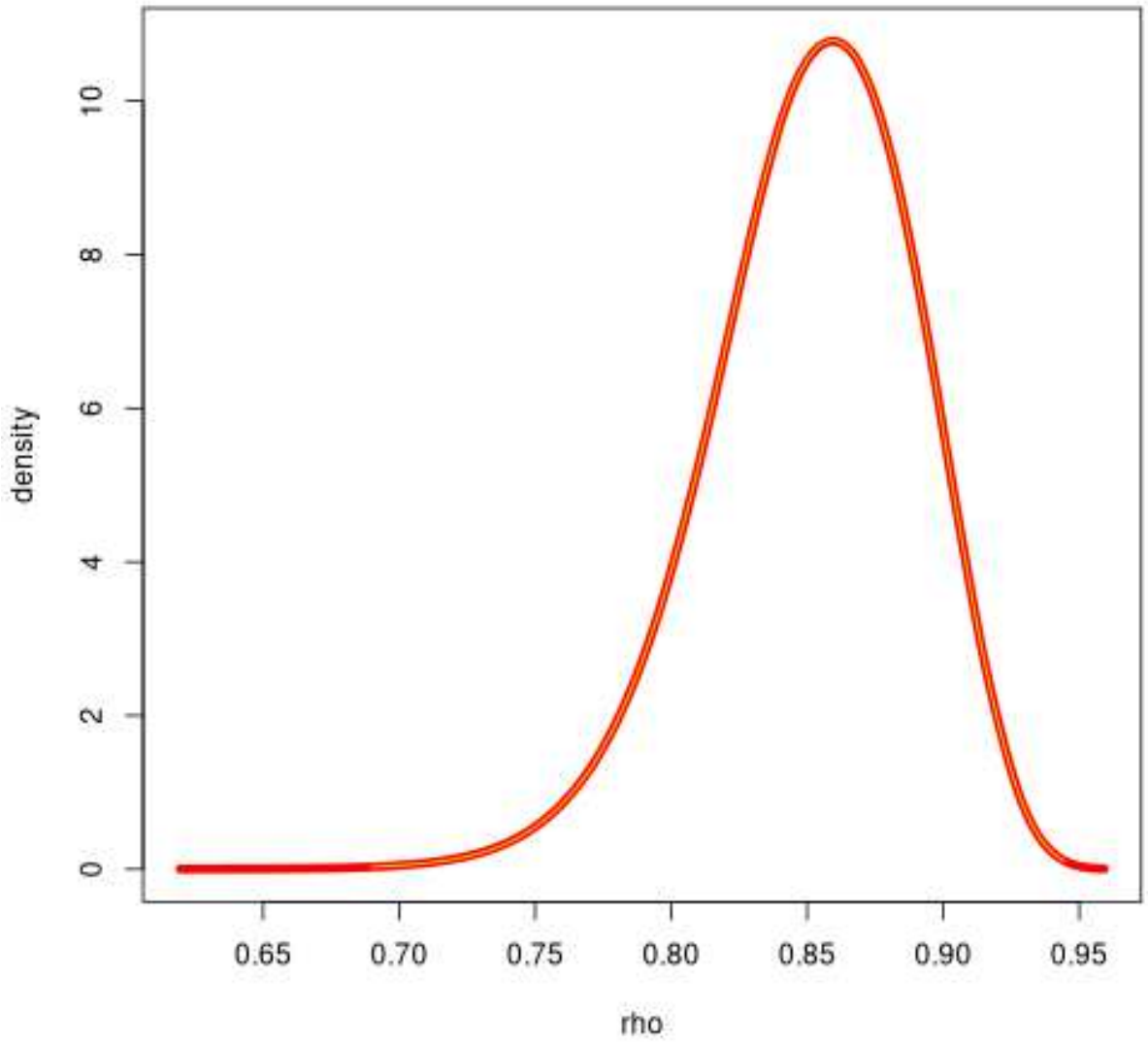


Figure 1: plot of chunk unnamed-chunk-17



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   0.353   0.399 0.051 0.803
## rgeneric 0.693   2.118 0.061 2.872
```

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())
##     for (pkg in c("Matrix")) {
##         if (!(pkg %in% (.packages()))) {
##             library(pkg, quietly = TRUE, character.only = TRUE)
##         }
##     }
##     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: 0x55fe4bc52168>
## <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())

    for (pkg in c("Matrix")) {
        if (!(pkg %in% (.packages()))) {
            library(pkg, quietly = TRUE, character.only = TRUE)
        }
    }

    ## 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")
```

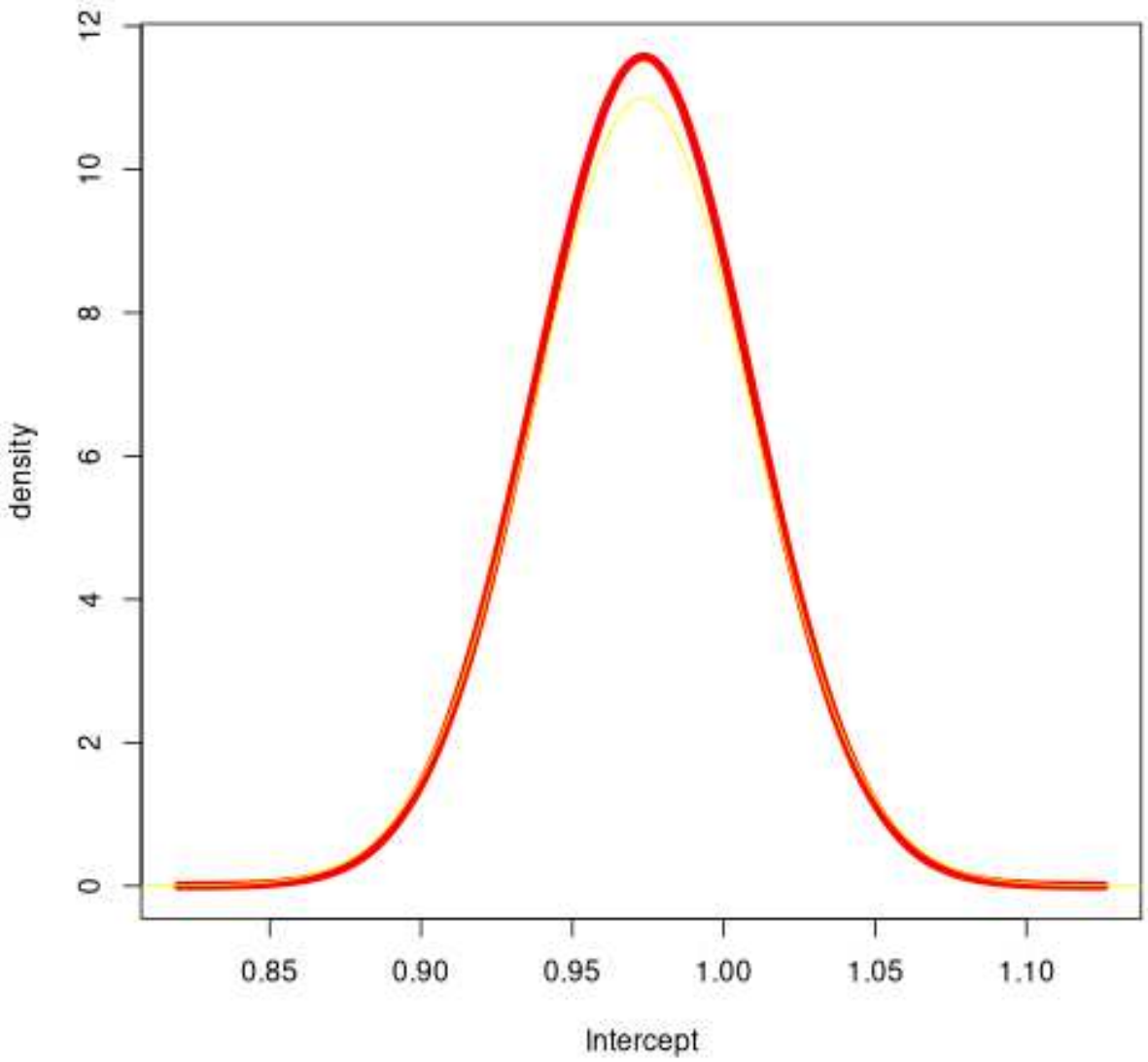
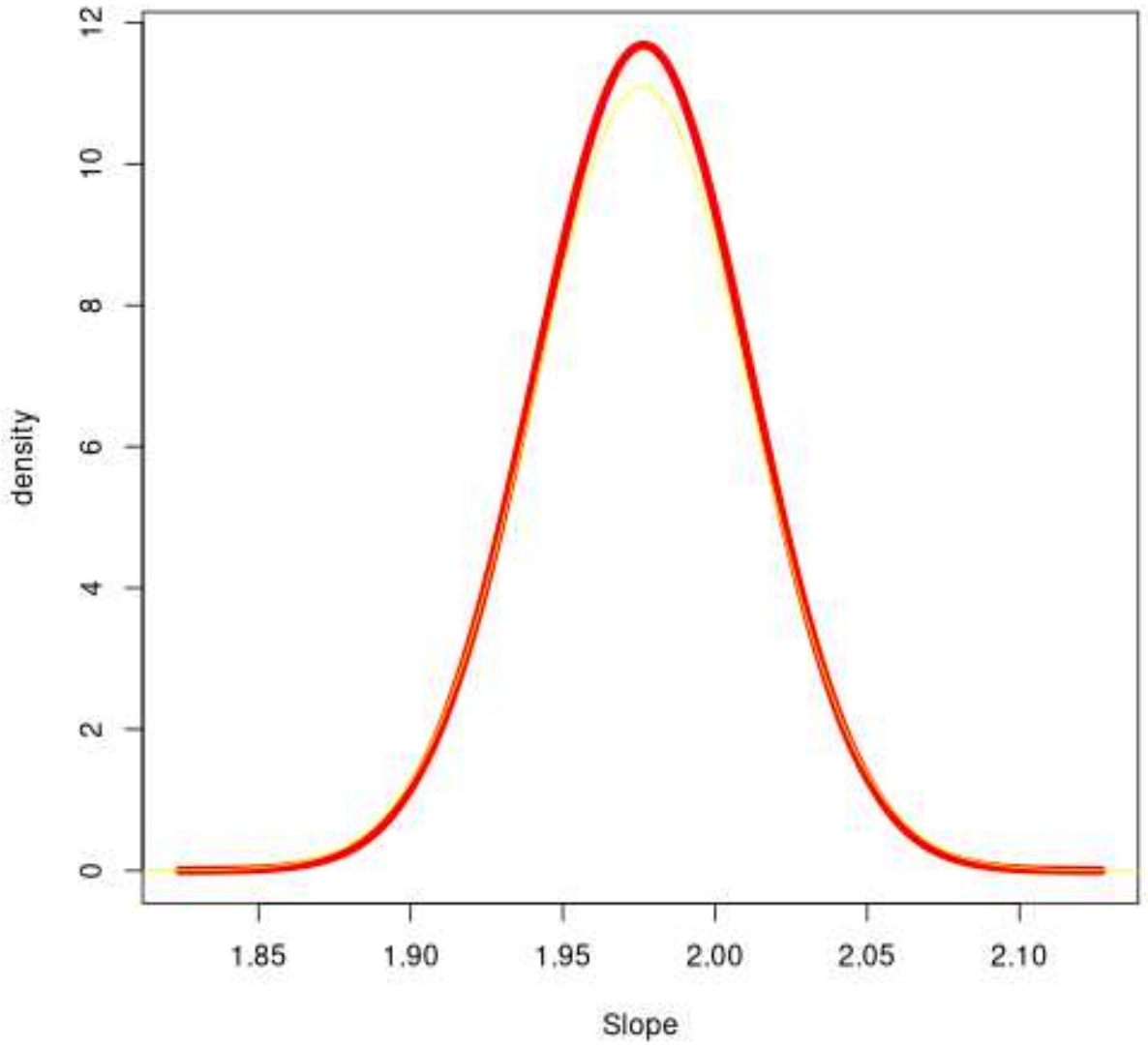


Figure 2: plot of chunk unnamed-chunk-24



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} ###r = inla.hyperpar(r) ### ###Re-plotting the results shows improvement: ###{r} ###plot(inla.smarginal(r$marginals.hyperpar[['Theta1 for idx']]), ### type="l", lwd=5, col="red", xlab="Intercept", ylab="density") ###lines(inla.smargi col="yellow") ### ###{r} ###plot(inla.smarginal(r$marginals.hyperpar[['Theta2 for idx']]), ### type="l", lwd=5, col="red", xlab="Slope", ylab="density") ###lines(inla.smarginal(rr$m 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 specific 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)  2.039116e-06
## log marginal-likelihood (Gaussian)    -9.749331e-07
```

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::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::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
        ## specific 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)  2.039116e-06
## log marginal-likelihood (Gaussian)    -9.749331e-07
```

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

```
##                                                [,1]
## log marginal-likelihood (integration)  1.097879e-06
## log marginal-likelihood (Gaussian)    -1.183264e-06
```

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

```
##                                                [,1]
## log marginal-likelihood (integration)  1.097879e-06
## log marginal-likelihood (Gaussian)    -1.183264e-06
```

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

```
## NULL
```

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.

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]))
```

The behaviour of the `cgeneric` model can be verified using `inla.cgeneric.q()`, like

```
gcc -Wall -fpic -g3 -O   -c -o cgeneric-demo.o cgeneric-demo.c
gcc -shared -o cgeneric-demo.so cgeneric-demo.o

> n <- 5
> cmodel <- inla.cgeneric.define(model = "inla_cgeneric_ar1_model",
+                                shlib = "cgeneric-demo.so", n = n)
> inla.cgeneric.q(cmodel)
$theta
[1] 1 1

$graph
5 x 5 sparse Matrix of class "dgCMatrix"

[1,] 1 1 . . .
[2,] 1 1 1 . .
[3,] . 1 1 1 .
[4,] . . 1 1 1
[5,] . . . 1 1

$Q
5 x 5 sparse Matrix of class "dgCMatrix"

[1,]  3.45640494 -1.59726402  .           .           .
[2,] -1.59726402  4.19452805 -1.59726402  .           .
[3,]  .          -1.59726402  4.19452805 -1.59726402  .
[4,]  .           .          -1.59726402  4.19452805 -1.59726402
[5,]  .           .           .          -1.59726402  3.45640494

$mu
numeric(0)
```

```
$log.prior
[1] -3.13722036

$log.norm.const
[1] -1.61423464
```

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 <assert.h>
#include <math.h>
#include <stdlib.h>
#include <strings.h>

#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_GRAPH:
    {
        // return a vector of indices with format
```

```
        // c(N, M, ii, jj)
        // where ii<=jj, ii is non-decreasing and jj is non-decreasing for the same ii
        // so like the loop
        // for i=0, ...
        // for j=i, ...
        // G_ij =
        // and M is the total length while N is the dimension

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

    case INLA_CGENERIC_Q:
    {
        // return c(-1, M, Qij) in the same order as defined in INLA_CGENERIC_GRAPH
        int M = N;
        ret = Calloc(2 + N, double);
        assert(ret);
        ret[0] = -1;                        /* REQUIRED! */
        ret[1] = M;                     /* number of (i <= j) */
        for (int i = 0; i < M; i++) {
            ret[2 + i] = prec;
        }
    }
        break;

    case INLA_CGENERIC_MU:
    {
        // return (N, mu)
        // if N==0 then mu is not needed as its taken to be mu[]==0
        ret = Calloc(1, double);
        assert(ret);
        ret[0] = 0;
    }
        break;

    case INLA_CGENERIC_INITIAL:
    {
        // return c(M, initials)
        // where M is the number of hyperparameters
        ret = Calloc(2, double);
        assert(ret);
        ret[0] = 1;
        ret[1] = 4.0;
    }
```

```
        break;

    case INLA_CGENERIC_LOG_NORM_CONST:
    {
        // return c(NORM_CONST) or a NULL-pointer if INLA should compute it by itself
        ret = Calloc(1, double);
        assert(ret);
        ret[0] = N * (-0.9189385332 + 0.5 * lprec);
    }
        break;

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

    case INLA_CGENERIC_VOID:
    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 <assert.h>
#include <math.h>
#include <stdlib.h>
#include <strings.h>

#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_GRAPH:
{
    // return a vector of indices with format
    // c(N, M, ii, jj)
    // where ii<=jj, ii is non-decreasing and jj is non-decreasing for the same ii
    // so like the loop
    // for i=0, ...
    // for j=i, ...
    // G_ij =
    // and M is the length of ii

    int M = N + N - 1, offset, i, k;
    ret = Calloc(2 + 2 * M, double);
    assert(ret);
    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:
{
    // optimized format
    // return c(-1, M, Qij) in the same order as defined in INLA_CGENERIC_GRAPH
    // where M is the length of Qij

    double param = prec / (1.0 - SQR(rho));
    int M = N + N - 1;
    int offset, i, k;
    ret = Calloc(2 + M, double);
    assert(ret);
    offset = 2;
    ret[0] = -1;                        /* REQUIRED */
    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:
    {
        // return (N, mu)
        // if N==0 then mu is not needed as its taken to be mu[]==0

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

    case INLA_CGENERIC_INITIAL:
    {
        // return c(M, initials)
        // where M is the number of hyperparameters

        ret = Calloc(3, double);
        assert(ret);
        ret[0] = 2;
        ret[1] = 1.0;
        ret[2] = 1.0;
    }
        break;

    case INLA_CGENERIC_LOG_NORM_CONST:
    {
        // return c(NORM_CONST) or a NULL-pointer if INLA should compute it by itself

        double prec_innovation = prec / (1.0 - SQR(rho));
        ret = Calloc(1, double);
        assert(ret);
        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:
    {
        // return c(LOG_PRIOR)

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

```
    case INLA_CGENERIC_VOID:
    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 {
        inla_cgeneric_threads_tp threads;

        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_mats;
        inla_cgeneric_mat_tp **mats;

        int n_smats;
        inla_cgeneric_smat_tp **smats;

        void *cache;
} 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_mats` and `n_smats`. 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
```

Information about threads used, is given in

```
typedef struct {
        // inla .. -t A:B
        // max = maximum number of threads = A*B
        // outer = number of threads in the outer loop (A)
        // inner = number of threads in the inner loop (B)
```

```
        int max;
        int outer;
        int inner;
} inla_cgeneric_threads_tp;
```

where `A` and `B` are what is defined in the `inla`-argument `num.threads`. Note that the `cgeneric` function needs to be thread-safe, and that number of threads used to call this function should be `data->threads.inner`.

You may store cached variables in `data`-struct

```
    void *cache;
```

The macros

```
    CGENERIC_CACHE_LEN(data_)
    CGENERIC_CACHE_ASSIGN_IDX(idx_, data_)
```

can be used to define threads specific cache, where `CGENERIC_CACHE_LEN` gives the require length, and `CGENERIC_CACHE_ASSIGN_IDX` will give the index (or assign an index variable to the index depending on the current and parent thread.

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 row wise (as natural in C), which is NOT how it is stored in `R`.

```
/*
 *      matrix storage is stored row by row.
 *
 *      In R the (default) storage is column by column, like
 *      > matrix(1:6,2,3)
 *      [,1] [,2] [,3]
 *      [1,]    1    3    5
 *      [2,]    2    4    6
 *
 *      hence
 *      > c(matrix(1:6,2,3))
 *      [1] 1 2 3 4 5 6
 *
 *      while in cgeneric, the matrix elements 'x', is stored as
 *      x[] = { 1, 3, 5, 2, 4, 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.
 * If the matrix is not a 'dgTMatrix', it is converted using
 * 'inla.as.sparse()'
 *
 * the matrix is stored in the order it appears, 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
 *
 *      If you want to have this matrix stored this column by column,
 *      then swap @i and @j, and rearrange @x, before passing them into
 *      'inla.cgeneric.define'.
 *
 *      For example, if we want to pass only the upper half of a
 *      symmetric sparse matrix, stored by column, then we can do
 *
 *      A <- inla.as.sparse(A)
 *      ii <- A@i
 *      A@i <- A@j
 *      A@j <- ii
 *      idx <- which(A@i <= A@j)
 *      A@i <- A@i[idx]
 *      A@j <- A@j[idx]
 *      A@x <- A@x[idx]
 *
 *      before passing it on into 'inla.cgeneric.define'
 */
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.

The C-function (defined in `cgeneric.h`)

```
static void inla_cgeneric_data_print(FILE * fp, inla_cgeneric_data_tp * data);
```

can be used to list the contents of `data` if needed.

Reimplementing model `generic0`

Here is an example of reimplementing `generic0` using `cgeneric`. The C-code is in `cgeneric-demo.c`, and the following R-code compare the builtin and cgeneric implementation

```
library(mvtnorm)
n <- 10
A <- matrix(rnorm(n^2), n, n)
Q <- A %*% t(A)
S <- solve(Q)
S <- S / exp(mean(log(diag(S))))
Q <- solve(S)

Cmatrix <- inla.as.sparse(Q)
s <- 0.1
y <- c(rmvnorm(1, sigma = S)) + rnorm(n, sd = s)

r <- inla(
    y ~ -1 + f(idx, model = "generic0", Cmatrix = Cmatrix,
               hyper = list(prec = list(prior = "loggamma",
                                        param = c(1, 1),
                                        initial = 4,
                                        fixed = FALSE))),
    num.threads = "1:1",
    data = data.frame(y, idx = 1:n),
    control.family = list(hyper =
        list(prec = list(initial = log(1/s^2), fixed = TRUE))))

## pass in the Cmatrix so its a good fit for
## the cgeneric-implementation, so we store 'Cmatrix'
## column-wise, and upper half only (diagonal included)
ii <- Cmatrix@i
Cmatrix@i <- Cmatrix@j
Cmatrix@j <- ii
idx <- which(Cmatrix@i <= Cmatrix@j)
Cmatrix@i <- Cmatrix@i[idx]
Cmatrix@j <- Cmatrix@j[idx]
Cmatrix@x <- Cmatrix@x[idx]

cmodel <- inla.cgeneric.define(model = "inla_cgeneric_generic0_model",
                               shlib = "cgeneric-demo.so",
                               n = n, Cmatrix = Cmatrix, debug = FALSE)
rc <- inla(
    y ~ -1 + f(idx, model = cmodel),
    num.threads = "1:1",
    data = data.frame(y, idx = 1:n),
    control.family = list(hyper = list(
```

```
                    prec = list(initial = log(1/s^2), fixed = TRUE))),
    verbose = TRUE)

print(r$mode$theta - rc$mode$theta)
print(r$mode$x - rc$mode$x)
print(r$mlik - rc$mlik)
```

which produces output like

```
> print(r$mode$theta - rc$mode$theta)
Log precision for idx
     -8.710710942e-06

> print(r$mode$x - rc$mode$x)
 [1] -6.988149615e-08  9.218204538e-08  1.151077143e-07 -3.683823807e-08
 [5] -8.793919748e-08 -9.518123112e-08  2.305469839e-08 -2.043514372e-08
 [9] -7.478532010e-08  1.064667019e-07 -6.988848433e-08  9.219126358e-08
[13]  1.151192251e-07 -3.684192190e-08 -8.794799145e-08 -9.519074928e-08
[17]  2.305700385e-08 -2.043718722e-08 -7.479279862e-08  1.064773486e-07

> print(r$mlik - rc$mlik)
                                                    [,1]
log marginal-likelihood (integration) 2.070899825e-07
log marginal-likelihood (Gaussian)    1.150937410e-05
```

Adding a likelihood function in C

As of Aug-2025, we added (experimental) support for user-supplied likelihoods. These acts like a normal `likelihood` with fixed name "cloglike", so we can have several of these and these can be mixed with others likelihood models as well.

The usage follows the use of the `cgeneric`, so to fix ideas we start with a small example.

```
n <- 55
y <- rnorm(n)
Y <- inla.mdata(y)
cloglike <- inla.cloglike.define(model = "inla_cloglike_gaussian",
                                 shlib = "cloglike-demo-gaussian.so",
                                 debug = FALSE)
rr <- inla(y ~ 1,
          data = data.frame(y = Y[, 1]),
          family = "gaussian",
          control.family = list(hyper = list(prec = list(param = c(1, 1)))))
r <- inla(Y ~ 1,
          data = list(Y = Y),
          family = "cloglike",
          control.family = list(cloglike = cloglike))
r$mlik - rr$mlik
r$mode$theta - rr$mode$theta
max(r$mode$x - rr$mode$x)
```

This example reimplements the Gaussian/Normal likelihood, and running the example we can verify the results are the same.

```
                                                    [,1]
log marginal-likelihood (integration) -8.809738432e-07
log marginal-likelihood (Gaussian)     3.487834164e-07
Theta1 for INLA.Data1
     -6.021477074e-07
[1] -3.261696468e-13
```

First we need to define the model, and we need the name of the function in the shared object and the name of the object itself.

```
cloglike <- inla.cloglike.define(model = "inla_cloglike_gaussian",
                                 shlib = "cloglike-demo-gaussian.so",
                                 debug = FALSE)
```

Just like the other `define`-functions, we could here add other `R` variables, like vectors, matrices etc, that the likelihood requires. These will then be available, just as for `cgeneric`, in the `data` argument that is passed through to the likelihood function, using the same data-type definition `inla_cgeneric_data_tp` which is described ealier.

To use the user-defined likelihood, we use the fixed-name "cloglike" and pass the information in the `control.family` using argument `cloglike`.

The information we need from the likelihood function, is the log-likelihood

$$\log \pi(y|x, \theta)$$

and the cummulative distribution function

$$\text{Prob}(\tilde{Y} < y|\theta, x)$$

for a given set of hyperparameters *theta*, linear predictor x, and observation y. We also need to know the number of hyperparameters θ (or `theta`) and their joint prior.

To pass the observations of `cloglike`, we use `inla.mdata()`, which essentially define the observations Y as data.frame, and the i'th row of Y is passed to the user-function to evaluate the log-likelihood from the ith observation(s).

In the example above, we only pass one observation

```
y <- rnorm(n)
Y <- inla.mdata(y)
```

but if we have scaling-parameters in the precision in the Normal, or we need the number trials in the Binomial distribution, we can append another column

```
Y <- inla.mdata(cbind(y, w))
```

Note that we need `cbind` them together into one object. To evaluate the ith log-likelihood, then observations (y_i, w_i) are passed into the user function as the "y". There are no restrictions to how many columns we can use in the `inla.mdata`-call.

Normal example

Here is a simple implementation of the Normal likelihood.

```
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include <strings.h>
```

```c
#include "cgeneric.h"

#define Calloc(n_, type_)  (type_ *)calloc((n_), sizeof(type_))
#define Malloc(n_, type_)  (type_ *)malloc((n_) * sizeof(type_))
#define SQR(x_) ((x_)*(x_))

double *inla_cloglike_gaussian(inla_cloglike_cmd_tp cmd, double *theta,
                  inla_cgeneric_data_tp *data,
                  int ny, double *y, int nx, double *x, double *result)
{
#define LOG_NORMC_GAUSSIAN (-0.91893853320467274178032973640560)    /* -1/2 * log(2*pi) */
#define CDF(x_) (0.5 * (1.0 + erf(M_SQRT1_2 * (x_))))

    double *ret = NULL, prec, lprec;

    if (theta) {
        lprec = theta[0];
        prec = exp(lprec);
    } else {
        prec = lprec = NAN;
    }

    switch (cmd) {
    case INLA_CLOGLIKE_INITIAL:
    {
        // return c(M, initials)
        // where M is the number of hyperparameters

        ret = Malloc(2, double);
        ret[0] = 1;
        ret[1] = 4.0;
    }
        break;

    case INLA_CLOGLIKE_LOG_PRIOR:
    {
        // return c(LOG_PRIOR). with a Gamma(1,1) for precision, this is the log prior for the log(prec
        ret = Malloc(1, double);
        ret[0] = -prec + lprec;
    }
        break;

    case INLA_CLOGLIKE_LOGLIKE:
    {
        // y[0] ~ N(x[i], prec)
        for (int i = 0; i < nx; i++) {
            result[i] = LOG_NORMC_GAUSSIAN + 0.5 * (lprec - (SQR(y[0] - x[i]) * prec));
        }
    }
        break;

    case INLA_CLOGLIKE_CDF:
    {
```

```
        // Prob(y[0] < x[i]) when y[0] ~ N(x[i], prec)
        double sprec = sqrt(prec);
        for (int i = 0; i < nx; i++) {
            double z = (y[0] - x[i]) * sprec;
            result[i] = CDF(z);
        }
    }
        break;

    case INLA_CLOGLIKE_QUIT:
        break;
    }

#undef LOG_NORMC_GAUSSIAN
#undef CDF
    return (ret);
}
```

Let us discuss each part in more detail.

Always include this header-file:

```
#include "cgeneric.h"
```

This is the definition that needs to be used.

```
double *inla_cloglike_gaussian(inla_cloglike_cmd_tp cmd, double *theta,
                   inla_cgeneric_data_tp *data,
                   int ny, double *y, int nx, double *x, double *result);
```

The `cmd` is what is being requested. `theta` is a vector of theta-values to be used. The `data` is same as for `cgeneric` and note that the same type is used. `ny` is the number of columns in the `inla.mdata()` call to define the data for this likelihood, hence `ny` is the number of columns. `y` is a vector of the i'th row of Y. `nx` is the number of linear predictors `x` that needs to be evaluated, and the result is written directly into `result`. Note that some requests require information which passed back directly, while other requests using the `result` vector. This is made to optimize its use and to make it similar to the `cgeneric` whenever possible.

The `inla_cloglike_cmd_tp` definition is defined the the header-file and contains the different requests that we need.

```
typedef enum {
        INLA_CLOGLIKE_INITIAL = 1,
        INLA_CLOGLIKE_LOG_PRIOR,
        INLA_CLOGLIKE_LOGLIKE,
        INLA_CLOGLIKE_CDF,
        INLA_CLOGLIKE_QUIT
} inla_cloglike_cmd_tp;
```

Normally, we would map `theta` values to interpretable parameters, which is done here as

```
    if (theta) {
        lprec = theta[0];
        prec = exp(lprec);
    } else {
        prec = lprec = NAN;
    }
```

Note that `theta` might be `NULL` as some requests do not need its values, so we need to check for a `NULL`

theta.

The first action to return the initial values for theta. This includes the number of hyperparameters that is used as well. We return the number of theta's as the first index and then the initial values sequentially. With no hyperparameters, then we just return ret[0]=0. Note that this r is free'd in the main program, hence it must be alloc'ed dynamically.

```
    switch (cmd) {
    case INLA_CLOGLIKE_INITIAL:
    {
        // return c(M, initials)
        // where M is the number of hyperparameters

        ret = Malloc(2, double);
        ret[0] = 1;
        ret[1] = 4.0;
    }
        break;
```

The next request is the log-prior for theta, which is the same as for cgeneric. Be aware that this is the log-prior for theta and NOT the log-prior for the interpretable parameters.

```
    case INLA_CLOGLIKE_LOG_PRIOR:
    {
        // return c(LOG_PRIOR). with a Gamma(1,1) for precision,
        // this is the log prior for the log(precision).
        ret = Malloc(1, double);
        ret[0] = -prec + lprec;
    }
        break;
```

Again, we return it in a new alloc'ed vector.

Next up is the log-likelihood itself.

```
    case INLA_CLOGLIKE_LOGLIKE:
    {
        // y[0] ~ N(x[i], prec)
        for (int i = 0; i < nx; i++) {
            result[i] = LOG_NORMC_GAUSSIAN + 0.5 * (lprec - (SQR(y[0] - x[i]) * prec));
        }
    }
        break;
```

This is just the log of the Normal density. Note that the observation is y[0] and that we need to loop over nx evaluations using the same observation y[0] for each of the nx values of the vector x. The result is written directly to the vector result.

The CDF (cummulative distribution function) is up next and it is similar to the log-likelihood.

```
    case INLA_CLOGLIKE_CDF:
    {
        // Prob(y[0] < x[i]) when y[0] ~ N(x[i], prec)
        double sprec = sqrt(prec);
        for (int i = 0; i < nx; i++) {
            double z = (y[0] - x[i]) * sprec;
            result[i] = CDF(z);
        }
```

```
    }
        break;
```

We hide the `CDF` for the Normal in the macro `CDF`, which is defined earlier as

```
#define CDF(x_) (0.5 * (1.0 + erf(M_SQRT1_2 * (x_))))
```

taking advantage of the C-function `erf`. If the `CDF` is not available, the you can just set `result[i] = NAN` and then some results in the output will not be available (like PIT values).

The last entry, is

```
    case INLA_CLOGLIKE_QUIT:
        break;
```

which is called in the end in case of any cleanup is needed.

Poisson example

Here is another example using the Poisson likelihood.

```
n <- 55
y <- rpois(n, exp(1))
Y <- inla.mdata(cbind(y))

cloglike <- inla.cloglike.define(model = "inla_cloglike_poisson",
    shlib = "cloglike-demo-poisson.so")
rr <- inla(y ~ 1, data = list(y = Y[, 1]), family = "poisson")
r <- inla(Y ~ 1, data = list(Y = Y), family = "cloglike",
    control.family = list(cloglike = cloglike))
## we add it back here
(r$mlik - sum(log(factorial(y)))) - rr$mlik
max(r$mode$x - rr$mode$x)

## section try: precompute the normalizing constant and
## add it as the 2nd column in the data
Y <- inla.mdata(cbind(y, lfactorial(y)))
cloglike <- inla.cloglike.define(model = "inla_cloglike_poisson",
    shlib = "cloglike-demo-poisson.so")
r <- inla(Y ~ 1, data = list(Y = Y), family = "cloglike",
    control.family = list(cloglike = cloglike))
r$mlik - rr$mlik
max(r$mode$x - rr$mode$x)

## use the cache-version, see the C-code
Y <- inla.mdata(cbind(y))
cloglike <- inla.cloglike.define(model = "inla_cloglike_poisson_cache",
    shlib = "cloglike-demo-poisson.so")
r <- inla(Y ~ 1, data = list(Y = Y), family = "cloglike",
    control.family = list(cloglike = cloglike))
r$mlik - rr$mlik
max(r$mode$x - rr$mode$x)
```

which gives these results

```
                                                    [,1]
log marginal-likelihood (integration) -3.848811048e-09
```

```
log marginal-likelihood (Gaussian)    -3.848811048e-09
[1] -8.679723607e-13
                                                    [,1]
log marginal-likelihood (integration) -1.184289999e-09
log marginal-likelihood (Gaussian)    -1.184289999e-09
[1] -2.147837463e-12
                                                   [,1]
log marginal-likelihood (integration) 8.881215763e-10
log marginal-likelihood (Gaussian)    8.881215763e-10
[1] -7.382983114e-13
```

This example implement various variations, including how to cache expensive operations so we do not need to compute them again.

Here is the code. The comments in the code also explains what is going on. (Both C-files are available in the repository as `cloglike-demo-gaussian.c` and `cloglike-demo-poisson.c`).

```c
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include <strings.h>

#include "cgeneric.h"

#define Calloc(n_, type_)  (type_ *)calloc((n_), sizeof(type_))
#define Malloc(n_, type_)  (type_ *)malloc((n_) * sizeof(type_))
#define SQR(x_) ((x_)*(x_))

double poisson_cdf(int y, double lambda)
{
    // simple standalone code. of course it is better to compute
    // it using the incomplete Gamma-distribution/external library.
    double res = 0.0;
    if (y >= 0) {
        double p = exp(-lambda);
        res = p;
        for (int yy = 1; yy <= y; yy++) {
            p *= lambda / yy;
            res += p;
        }
    }
    return res;
}

double *inla_cloglike_poisson(inla_cloglike_cmd_tp cmd, double *theta,
                  inla_cgeneric_data_tp *data,
                  int ny, double *y, int nx, double *x, double *result)
{
    double *ret = NULL;

    switch (cmd) {
    case INLA_CLOGLIKE_INITIAL:
    {
        ret = Malloc(1, double);
```

```
        ret[0] = 0;
    }
        break;

    case INLA_CLOGLIKE_LOG_PRIOR:
    {
        // not in use
        assert(0 == 1);
    }
        break;

    case INLA_CLOGLIKE_LOGLIKE:
    {
        // if ny=2, then the 2nd column is log(y[0]!), otherwise, we
        // ignore the constant normalizing constant (which only
        // involves the marginal likelihood calculations)
        if (ny == 1) {
            for (int i = 0; i < nx; i++) {
                result[i] = x[i] * y[0] - exp(x[i]);
            }
        } else if (ny == 2) {
            for (int i = 0; i < nx; i++) {
                result[i] = x[i] * y[0] - exp(x[i]) - y[1];
            }
        } else {
            assert(0 == 1);
        }
    }
        break;

    case INLA_CLOGLIKE_CDF:
    {
        for (int i = 0; i < nx; i++) {
            result[i] = poisson_cdf(y[0], exp(x[i]));
        }
    }
        break;

    case INLA_CLOGLIKE_QUIT:
        break;
    }

    return (ret);
}

double *inla_cloglike_poisson_cache(inla_cloglike_cmd_tp cmd, double *theta,
                    inla_cgeneric_data_tp *data,
                    int ny, double *y, int nx, double *x, double *result)
{
    // this code shows how to 'cache' expensive calculations, in this case log(y[0]!)

    typedef struct {
        int ymax;
```

```
        double *lfactorial;
    } Cache_tp;

    if (!(data->cache)) {
        // use a random name
#pragma omp critical (Name_e2814d0ff0cb393dee01d0eb049e6e976f56cce8)
        if (!(data->cache)) {
            Cache_tp *c = Malloc(1, Cache_tp);
            c->ymax = 1024;                 /* or something */
            c->lfactorial = Malloc(c->ymax + 1, double);
            c->lfactorial[0] = 0.0;
            for (int k = 1; k <= c->ymax; k++) {
                c->lfactorial[k] = c->lfactorial[k - 1] + log(k);
            }
            // IMPORTANT: need to assign data->cache as the last
            // expression within this 'if(!(data->cache))',
            // otherwise, we can run into race-conditions
            *((Cache_tp **) (&data->cache)) = c;
        }
    }
    Cache_tp *cache = *((Cache_tp **) (&data->cache));

    double *ret = NULL;
    switch (cmd) {
    case INLA_CLOGLIKE_INITIAL:
    {
        // no hyperparameters
        ret = Malloc(1, double);
        ret[0] = 0;
    }
        break;

    case INLA_CLOGLIKE_LOG_PRIOR:
    {
        // not in use
        assert(0 == 1);
    }
        break;

    case INLA_CLOGLIKE_LOGLIKE:
    {
        // if y[0] is to large, we have to rebuild the cache...
        int iy = (int) y[0];
        assert(iy <= cache->ymax);
        double lfac = cache->lfactorial[iy];

        if (iy == 0) {
            for (int i = 0; i < nx; i++) {
                result[i] = -exp(x[i]) - lfac;
            }
        } else {
            for (int i = 0; i < nx; i++) {
                result[i] = x[i] * iy - exp(x[i]) - lfac;
```

```
            }
        }
    }
        break;

    case INLA_CLOGLIKE_CDF:
    {
        for (int i = 0; i < nx; i++) {
            result[i] = poisson_cdf(y[0], exp(x[i]));
        }
    }
        break;

    case INLA_CLOGLIKE_QUIT:
        break;
    }

    return (ret);
}
```