

Model “rgeneric”

This is a class of generic models allows the user to define latent (sub-)models in R, for cases where the requested model is not yet implemented in INLA, and do the Bayesian inference using INLA. The actual implementation of this feature is somewhat technical as all the computations are done in a standalone C-program and not within R, so therefore this option is *only* available for Linux and MacOSX, and *not* available for Windows. Also you need to run it on a dual-core (or more cores) computer and have installed the `multicore`-library. It will run more slowly for two reasons:

1. The C-program doing the calculations has to communicate with R to get model-properties such as the precision matrix, the graph, etc, and this communication is not for free.
2. Due the same reason, the C-program doing the calculations has to run in serial mode and cannot make use of multi-cores.

Anyway, we still hope it would be useful.

Defining a latent model in R

The use of this feature, is in short the following. First we pass our definition of the model `model.def`, to define a `inla-rgeneric` object,

```
my.rgeneric.model = inla.rgeneric.define(my.model.def, <args>)
```

Here `args` could be parameters to `my.model.def` such as dimension, prior-settings, etc, so `my.model.def` can be written more generically. Then we can use this object as a latent model

```
y ~ ... + f(i, model="rgeneric", rgeneric = my.rgeneric.model)
```

Example: the AR1 model

The `my.model.def` needs to follow some rules and provide the required features. It is easier with an example, so let us implement the AR1-model (see also `inla.doc("ar1")`), which is defined as¹

$$x_1 \sim \mathcal{N}(0, \tau)$$

and

$$x_t \mid x_1, \dots, x_{t-1} \sim \mathcal{N}(\rho x_{t-1}, \tau_I), \quad t = 2, \dots, n.$$

Although our scale-parameter is the marginal precision τ (and there are good good reasons for this!), the joint is more naturally expressed using the innovation precision $\tau_I = \tau/(1 - \rho^2)$. The joint density of x is Gaussian

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

where the precision matrix is

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

¹The second argument in $\mathcal{N}(,)$ is the precision matrix not the covariance matrix.

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

$$\tau = \exp(\theta_1)$$

and

$$\rho = 2 \frac{\exp(\theta_2)}{1 + \exp(\theta_2)} - 1$$

It is required that the parameters $\theta = (\theta_1, \theta_2)$ have support on \Re and the priors for τ and ρ are given as the corresponding priors for θ_1 and θ_2 . Note that INLA only provide the marginal posteriors for θ , but you can use `inla.tmarginal` to convert it to the appropriate marginals for ρ and τ .

We assign a (Gamma) $\Gamma(\cdot; a, b)$ prior (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

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

We will use $a = b = 1$, $\mu = 0$ and $\kappa = 1$.

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

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

which except for the graph, depends on the current value of θ . We need to wrap this into a common function, which process the request from the C-program. Note the list of commands and its names

```
cmd = c("Q", "graph", "initial", "log.norm.const", "log.prior", "quit"),
```

are fixed and predefined (in the file `inla.h`). Our skeleton for defining a model is as follows.

```
'inla.rgeneric.ar1.model' = function(
  cmd = c("graph", "Q", "initial", "log.norm.const", "log.prior", "quit"),
  theta = NULL,
  args = NULL)
{
  graph = function(n, ntheta, theta){ <to be completed> }
  Q = function(n, ntheta, theta) { <to be completed> }
  log.norm.const = function(n, ntheta, theta) { <to be completed> }
  log.prior = function(n, ntheta, theta) { <to be completed> }
  initial = function(n, ntheta, theta) { <to be completed> }
  quit = function(n, ntheta, theta) { <to be completed> }

  cmd = match.arg(cmd)
  val = do.call(cmd, args = list(
    n = as.integer(args$n),
    ntheta = as.integer(args$ntheta),
    theta = theta))

  return (val)
}
```

The input parameters are

cmd What to return or process

theta The values of the θ -parameters to evaluate at

args User-defined argument(s) from the `inla.rgeneric.define()`-call. In this example we use `n` and `ntheta` as the user-defined arguments and pass them to every function, so we need to pass `n` and `ntheta` when defining the model

```
model = inla.rgeneric.define(inla.rgeneric.ar1.model,  
                             n = 100, ntheta = 2)
```

for example, but you may chose to do this differently.

Our next task, is the “fill in the blanks”, ie to define the functions `graph`, `Q`, `initial`, `log.norm.const`, `log.prior` and `quit` for our AR1-model.

Function `graph`

This function must return a matrix, (must) preferably a `sparseMatrix` (for efficiency reasons), with non-zero entries defining the graph. Only the lower-triangular part of the matrix is used.

```
graph = function(n, ntheta, theta)  
{  
  ## return the graph of the model. the values of Q is only  
  ## interpreted as zero or non-zero.  
  if (FALSE) {  
    ## slow and easy: dense-matrices  
    G = toeplitz(c(1, 1, rep(0, n-2L)))  
    ## you *can* do this to avoid transferring zeros, but then  
    ## its better to use the sparseMatrix()-approach below.  
    G = inla.as.dgTMatrix(G)  
  } else {  
    ## fast and little more complicated: sparse matrices. we  
    ## only need to define the lower-triangular of G!  
    i = c(  
      ## diagonal  
      1L, n, 2L:(n-1L),  
      ## off-diagonal  
      1L:(n-1L))  
    j = c(  
      ## diagonal  
      1L, n, 2L:(n-1L),  
      ## off-diagonal  
      2L:n)  
    x = 1 ## meaning that all are 1  
    G = sparseMatrix(i=i, j=j, x=x, giveCsparse = FALSE)  
  }  
  return (G)  
}
```

An alternative to this `graph`-function, is to replace it with a call to the `Q`-function (which follows next), using parameters `theta` so that no element in the precision that can be non-zero actually *is* non-zero. For example

```
graph = function(n, ntheta, theta)
{
  return (Q(n, ntheta, rep(1, ntheta)))
}
```

whereas choosing `theta = c(1,0)` would give $\rho = 0$ and this function would return a (wrong) diagonal matrix instead of the correct tri-diagonal matrix.

Function Q

This function must return the precision matrix $Q(\theta)$, (must) preferably a `sparseMatrix` (for efficiency reasons). Only the lower-triangular part of the matrix is used. We will make use of the helper function

```
interpret.theta = function(n, ntheta, theta)
{
  ## internal helper-function to map the parameters from the
  ## internal-scale to the user-scale
  return (list(prec = exp(theta[1L]),
               rho = 2*exp(theta[2L])/(1+exp(theta[2L])) - 1.0))
}
```

to convert from θ_1 to τ , and from θ_2 to ρ . The Q-function can then be implemented as follows.

```
Q = function(n, ntheta, theta)
{
  ## returns the precision matrix for given parameters
  param = interpret.theta(n, ntheta, theta)
  if (FALSE) {
    ## slow and easy: dense-matrices
    Q = param$prec/(1-param$rho^2) *
      toeplitz(c(1+param$rho^2, -param$rho, rep(0, n-2L)))
    ## first and last diagonal term is 1*marg.prec
    Q[1, 1] = Q[n, n] = param$prec/(1-param$rho^2)
    ## you *can* do this to avoid transferring zeros, but then
    ## its better to use the sparseMatrix()-approach below.
    Q = inla.as.dgTMatrix(Q)
  } else {
    ## fast and a little more complicated: sparse matrices. we
    ## only need to define the lower-triangular G!
    i = c(
      ## diagonal
      1L, n, 2L:(n-1L),
      ## off-diagonal
      1L:(n-1L))
    j = c(
      ## diagonal
      1L, n, 2L:(n-1L),
      ## off-diagonal
      2L:n)
    x = param$prec/(1-param$rho^2) * c(
      ## diagonal
      1L, 1L, rep(1+param$rho^2, n-2L),
```

```

        ## off-diagonal
        rep(-param$rho, n-1L))
    Q = sparseMatrix(i=i, j=j, x=x, giveCsparse=FALSE)
  }
  return (Q)
}

```

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(n, ntheta, theta)
{
  ## return the log(normalising constant) for the model.
  param = interpret.theta(n, ntheta, theta)
  prec.innovation = param$prec / (1.0 - param$rho^2)
  val = n * (- 0.5 * log(2*pi) + 0.5 * log(prec.innovation)) +
        0.5 * log(1.0 - param$rho^2)
  return (val)
}

```

Function log.prior

This function must return the (log-)prior of the prior density for θ . For the AR1-model, we have 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 $\kappa = 1$:

```

log.prior = function(n, ntheta, theta)
{
  ## return the log-prior for the hyperparameters. the
  ## '+theta[1L]' is the log(Jacobian) for having a gamma prior
  ## on the precision and convert it into the prior for the
  ## log(precision).
  param = interpret.theta(n, ntheta, theta)
  val = (dgamma(param$prec, shape = 1, rate = 1, log=TRUE) + theta[1L] +
        dnorm(theta[2L], mean = 0, sd = 1, log=TRUE))
  return (val)
}

```

Function initial

This function returns the initial values for θ .

```

initial = function(n, ntheta, theta)
{
  ## return the initial values
  return (rep(1, ntheta))
}

```

Function quit

This function is called when all the computations are done and before exit-ing the C-program. Usually, there is nothing in particular to do, but if there is something that should be done, you can do this here.

```

quit = function(n, ntheta, theta)
{
  return ()
}

```

The complete definition of the AR1-model

For completeness, we include here the complete code for the AR1-model, collecting all the functions already defined. The function is predefined in the INLA-library.

```

'inla.rgeneric.ar1.model' = function(
  cmd = c("graph", "Q", "initial", "log.norm.const", "log.prior", "quit"),
  theta = NULL,
  args = NULL)
{
  ## this is an example of the 'rgeneric' model. here we implement
  ## the AR-1 model as described in inla.doc("ar1"), where 'rho' is
  ## the lag-1 correlation and 'prec' is the *marginal* (not
  ## conditional) precision.

  interpret.theta = function(n, ntheta, theta)
  {
    ## internal helper-function to map the parameters from the
    ## internal-scale to the user-scale
    return (list(prec = exp(theta[1L]),
                 rho = 2*exp(theta[2L])/(1+exp(theta[2L])) - 1.0))
  }

  graph = function(n, ntheta, theta)
  {
    ## return the graph of the model. the values of Q is only
    ## interpreted as zero or non-zero.
    if (FALSE) {
      ## slow and easy: dense-matrices
      G = toeplitz(c(1, 1, rep(0, n-2L)))
      ## you *can* do this to avoid transferring zeros, but then
      ## its better to use the sparseMatrix()-approach below.
      G = inla.as.dgTMatrix(G)
    } else {
      ## fast and little more complicated: sparse matrices. we

```

```

    ## only need to define the lower-triangular of G!
    i = c(
        ## diagonal
        1L, n, 2L:(n-1L),
        ## off-diagonal
        1L:(n-1L))
    j = c(
        ## diagonal
        1L, n, 2L:(n-1L),
        ## off-diagonal
        2L:n)
    x = 1 ## meaning that all are 1
    G = sparseMatrix(i=i, j=j, x=x, giveCsparse = FALSE)
  }
  return (G)
}

Q = function(n, ntheta, theta)
{
  ## returns the precision matrix for given parameters
  param = interpret.theta(n, ntheta, theta)
  if (FALSE) {
    ## slow and easy: dense-matrices
    Q = param$prec/(1-param$rho^2) *
      toeplitz(c(1+param$rho^2, -param$rho, rep(0, n-2L)))
    ## first and last diagonal term is 1*marg.prec
    Q[1, 1] = Q[n, n] = param$prec/(1-param$rho^2)
    ## you *can* do this to avoid transferring zeros, but then
    ## its better to use the sparseMatrix()-approach below.
    Q = inla.as.dgTMatrix(Q)
  } else {
    ## fast and a little more complicated: sparse matrices. we
    ## only need to define the lower-triangular G!
    i = c(
        ## diagonal
        1L, n, 2L:(n-1L),
        ## off-diagonal
        1L:(n-1L))
    j = c(
        ## diagonal
        1L, n, 2L:(n-1L),
        ## off-diagonal
        2L:n)
    x = param$prec/(1-param$rho^2) * c(
        ## diagonal
        1L, 1L, rep(1+param$rho^2, n-2L),
        ## off-diagonal
        rep(-param$rho, n-1L))
    Q = sparseMatrix(i=i, j=j, x=x, giveCsparse=FALSE)
  }
  return (Q)
}

```

```

}

log.norm.const = function(n, ntheta, theta)
{
  ## return the log(normalising constant) for the model.
  param = interpret.theta(n, ntheta, theta)
  prec.innovation = param$prec / (1.0 - param$rho^2)
  val = n * (- 0.5 * log(2*pi) + 0.5 * log(prec.innovation)) +
    0.5 * log(1.0 - param$rho^2)
  return (val)
}

log.prior = function(n, ntheta, theta)
{
  ## return the log-prior for the hyperparameters. the
  ## '+theta[1L]' is the log(Jacobian) for having a gamma prior
  ## on the precision and convert it into the prior for the
  ## log(precision).
  param = interpret.theta(n, ntheta, theta)
  val = (dgamma(param$prec, shape = 1, rate = 1, log=TRUE) + theta[1L] +
    dnorm(theta[2L], mean = 0, sd = 1, log=TRUE))
  return (val)
}

initial = function(n, ntheta, theta)
{
  ## return the initial values
  return (rep(1, ntheta))
}

quit = function(n, ntheta, theta)
{
  return (invisible())
}

cmd = match.arg(cmd)
val = do.call(cmd, args = list(
  n = as.integer(args$n),
  ntheta = as.integer(args$ntheta),
  theta = theta))

return (val)
}

```

Example of usage

```

n = 100
rho=0.9
y = arima.sim(n, model = list(ar = rho)) * sqrt(1-rho^2)
idx = 1:n
model = inla.rgeneric.define(inla.rgeneric.ar1.model, n=n, ntheta = 2)

```



```
formula = y ~ -1 + f(idx, model="rgeneric", rgeneric = model)
r = inla(formula, data = data.frame(y, idx), family = "gaussian")
```