Model rgeneric

This is a class of generic models allows the user to define latent model-component in `R`, for cases where the requested model is not yet implemented in `INLA`, and do the Bayesian inference using `INLA`. It will run slower as the model properties has to be evaluated in `R` within a `C`-program.

Defining a latent model in `R`

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

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

Here, `rmodel` is model definition encoded as an `R`-function, `debug` is a logical parameter if debug information should be printed, and `...` are further named variables that goes into the environment of `rmodel` (like dimension, prior-settings, etc). Then the model can used as

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

Example: the AR1 model

The `rmodel` needs to follow some rules to provide the required features. As an example, we will show how to implement the AR1-model, see `inla.doc("ar1")`). This model is defined as[1]

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

and

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

The scale-parameter is the marginal precision τ, but the joint density is more naturally expressed using the innovation precision $\tau_I = \tau/(1-\rho^2)$. 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, 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. **IMPORTANT: A good re-parametersation is required**

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

for INLA to work well. A good parmeterisation makes, ideally, the "Fisher information matrix" of θ constant (wrt to θ). It is sufficient to check this in a frequentistic setting with data directly from the AR(1) model, in this case. 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) $\Gamma(.; a, b)$ 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 graph,
- the precision matrix $Q(\theta)$,
- the zero mean,
- 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. The list of commands and its names

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

are fixed. The skeleton-function for defining a model is

```
‘inla.rgeneric.ar1.model‘ = function(
        cmd = c("graph", "Q", "mu", "initial", "log.norm.const",
                "log.prior", "quit"),
        theta = NULL)
{
    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> }

    val = do.call(match.arg(cmd))
    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 to `inla.rgeneric.ar1.model`. This is because several instances of `rgeneric` models will share the same `.GlobalEnv`.

Our next task, is the "fill in the blanks" in this function.

Function graph

This function must return a sparseMatrix, with the non-zero elements of the precision matrix. Only the lower-triangular part of the matrix is used.

```
graph = function()
{
    ## return the graph of the model. the values of Q is only interpreted as zero or
    ## non-zero. return a sparse.matrix
    if (FALSE) {
        ## slow and easy: dense-matrices
        G = toeplitz(c(1, 1, rep(0, n-2L)))
        G = inla.as.sparse(G)
    } else {
        ## faster. 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)
}
```

Function Q

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

```
interpret.theta = function()
{
    ## 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()
{
    ## returns the precision matrix for given parameters
    param = interpret.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)))
        Q[1, 1] = Q[n, n] = param$prec/(1-param$rho^2)
        Q = inla.as.sparse(Q)
    } else {
        ## faster. we only need to define the lower-triangular Q!
        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 `mu`

This function must return the mean of the model. Often, the mean is zero, but sometimes it might depend on the hyperparameters as well. If `numeric(0)` is returned, then this is equivalent that the mean is zero. An alternative in this example, would be to return `rep(0,n)`.

```
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()
{
    ## return the log(normalising constant) for the model
    param = interpret.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)
}
```

NOTE: If the log-normalizing constant in any case need to be computed from scratch as

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

then `INLA` will compute this if `numeric(0)` is returned, like

```
log.norm.const = function()
{
    ## let INLA compute it
    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()
 {
     ## 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()
     val = (dgamma(param$prec, shape = 1, rate = 1, log=TRUE) + theta[1L] +
                dnorm(theta[2L], mean = 0, sd = 1, log=TRUE))
     return (val)
 }
```

An alternative, is to pass the parameters of the these priors as when defining the model using `inla.rgeneric.define` in the `...` argument.

Function `initial`

This function returns the initial values for θ.

```
initial = function()
{
    ## return initial values
    ntheta = 2
    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()
{
    return (invisible())
}
```

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", "mu", "initial",
                                        "log.norm.const", "log.prior", "quit"),
                                  theta = 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()
    {
        ## 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()
{
    ## return the graph of the model. the values of Q is only interpreted as zero or
    ## non-zero. return a sparse.matrix
    if (FALSE) {
        ## slow and easy: dense-matrices
        G = toeplitz(c(1, 1, rep(0, n-2L)))
        G = inla.as.sparse(G)
    } else {
        ## faster. 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()
{
    ## returns the precision matrix for given parameters
    param = interpret.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)))
        Q[1, 1] = Q[n, n] = param$prec/(1-param$rho^2)
        Q = inla.as.sparse(Q)
    } else {
        ## faster. we only need to define the lower-triangular Q!
        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)
    }

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

    log.norm.const = function()
    {
        ## return the log(normalising constant) for the model
        param = interpret.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()
    {
        ## 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()
        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()
    {
        ## return initial values
        ntheta = 2
        return (rep(1, ntheta))
    }

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

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

Example of usage

```
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), family = "gaussian")
```

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 do `demo(rgeneric)`.

```
'inla.rgeneric.iid.model' = function(cmd = c("graph", "Q", "mu", "initial",
                                             "log.norm.const", "log.prior", "quit"),
                                     theta = NULL)
{
    ## this is an example of the 'rgeneric' model. here we implement the iid model as described
    ## in inla.doc("iid"), without the scaling-option

    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())
```

```
    }

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

Example: a model for the mean structure

```
## 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').
## This is a prototype example how similar sitations
## could be approached,  where essentially the latent model is a
## model for the 'mean' only.

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

    ## 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() {
        ## the easiest is to let INLA compute this
        return(numeric(0))
    }

    log.prior = function() {
        par = interpret.theta()
        val = (dnorm(par$a, mean=0, sd = sqrt(1/0.001), log=TRUE) +
               dnorm(par$b, mean = 0, sd = sqrt(1/0.001), log=TRUE))
        return(val)
    }

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

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

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

a = 1
b = 2
n = 100
x = rnorm(n)
eta = a + b*x
y = eta + rnorm(n)

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 = 0.001, prec = 0.001))

## compare the results with the 'truth'
par(mfrow=c(2, 1))
plot(r$marginals.hyperpar[['Theta1 for idx']], type="l",  lwd=2, col="red",
     main = "Posterior for the intercept (red=rgeneric, blue=default)")
lines(rr$marginals.fixed$'(Intercept)', lwd=2, col="blue")

plot(r$marginals.hyperpar[['Theta2 for idx']], type="l",  lwd=2, col="red",
     main = "Posterior for the slope (red=rgeneric,  blue=default)")
lines(rr$marginals.fixed$'x', lwd=2, col="blue")
```