Short tutorials from old www-page

Haavard Rue (hrue@r-inla.org)

KAUST, Aug 2020

Contents

Introduction

This vignette is a copy and slightly edited FAQ-entries and other short tutorials from the old `www.r-inla.org` page. The content comes is slightly random order, sorry about that.

User-defined priors for the hyperparameters

If you want to use a prior for the hyperparameter that is not yet implemented there are two choices. If you think that your prior should be on the list and that other might use it to, please let us know. Alternatively, you can define your own prior using \verb|prior = "expression: ...."|, or by specifiying a table of x and y values which define the prior distribution.

There are three ways to specify prior distributions for hyperparameters in INLA:

- Use an available prior distribution
- Define your own prior distribution function using `R`-like (not equal) syntax as expression.
- Create a table of (x, y) values which represent your prior distribution.

In the following we will provide more details regarding the two last options. Finally, we will present an example illustrating (and comparing) the three different possibilities by means of the log-gamma distribution for the precision parameter.

A user can specify any (univariate) prior distribution for the hyperparameter θ by defining an expression for the log-density $\log \pi(\theta)$, as a function of the corresponding θ. It is important to be aware that θ is on the internal scale.

The format is

```
    expression: statement; statement; ...; return(value)
```

where "statement" is any regular statement (more below) and "value" is the value for the log-density of the prior, evaluated at the current value for θ.

Here, is an example defining the log-gamma distribution:

```
    prior.expression = "expression:
        a = 1;
        b = 0.1;
        precision = exp(log_precision);
        logdens = log(b^a) - lgamma(a)
           + (a-1)*log_precision - b*precision;
        log_jacobian = log_precision;
        return(logdens + log_jacobian);"
```

Some syntax specific notes: * No white-space before "(.)" in the return statement. * A ";" is needed to terminate each expression. * A "_" is allowed in variable names.

Known functions that can be used within the expression statement are

- common math functions, such as exp, sin, ...
- "gamma" denotes the gamma-function and "lgamma" is its log
- `x^y` is expressed as either `x^y` or `pow(x,y)`

Instead of defining a prior distribution function, it is possible to provide a table of suitable values x (internal scale) and the corresponding log-density values y. INLA fits a spline through the provided points and continues with this in the succeeding computations. Note, there is no transformation into a functional form performed or required. The input-format for the table is a string, which starts with `table:` and is then followed by a block of x-values and a block of the corresponding y-values, which represent the values of the log-density evaluated on x. Thus

```
    table: x_1 ... x_n y_1 ... y_n
```

We illustrate all three different ways of defining a prior distribution for the precision of a normal likelihood. To show that the three definitions lead to the same result we inspect the logmarginal likelihood.

```
## the loggamma-prior
prior.function = function(log_precision) {
    a = 1;
    b = 0.1;
    precision = exp(log_precision);
    logdens = log(b^a) - lgamma(a) + (a-1)*log_precision - b*precision;
    log_jacobian = log_precision;
    return(logdens + log_jacobian)
}

## implementing the loggamma-prior using "expression:"
prior.expression = "expression:
a = 1;
b = 0.1;
precision = exp(log_precision);
logdens = log(b^a) - lgamma(a)
```

```
    + (a-1)*log_precision - b*precision;
log_jacobian = log_precision;
return(logdens + log_jacobian);"

## use suitable support points x
lprec = seq(-10, 10, len=100)
## link the x and corresponding y values into a
## string which begins with "table:""
prior.table = paste(c("table:", cbind(lprec,
                    prior.function(lprec))), collapse=" ", sep="")

# simulate some data
n = 50
y = rnorm(n)

## use the built-in loggamma prior
r1 = inla(y~1,data = data.frame(y),
control.family = list(hyper = list(prec = list(
                      prior = "loggamma", param = c(1, 0.1)))))

## use the definition using expression
r2 = inla(y~1, data = data.frame(y),
          control.family = list(hyper = list(
                                prec = list(prior = prior.expression))))

## use a table of x and y values representing the loggamma prior
r3 = inla(y~1, data = data.frame(y),
          control.family = list(hyper = list(
                                prec = list(prior = prior.table))))

print(round(c(r1$mlik[1], r2$mlik[1], r3$mlik[1]), dig=3))
```

```
## [1] -70.657 -70.657 -70.657
```

Does INLA support the use of different link-functions?

Yes, the type of link function is given in the `control.family` statement using `control.link=...`, and the type of link-functions implemented are listed on the documentation for each likelihood. The default link is `default` which corresponds to the second link function in the list. Here is an example

```
n = 100
z = rnorm(n)
eta = 1 + 0.1*z
N = 2

p = inla.link.invlogit(eta)
y = rbinom(n,  size = N, prob = p)
r = inla(y ~ 1 + z,  data = data.frame(y, z), family = "binomial", Ntrials = rep(N, n),
        control.family = list(control.link = list(model="logit")),
        control.predictor = list(compute=TRUE))

p = inla.link.invprobit(eta)
y = rbinom(n,  size = N, prob = p)
rr = inla(y ~ 1 + z,  data = data.frame(y, z), family = "binomial", Ntrials = rep(N, n),
```

```
        control.family = list(control.link = list(model="probit")),
        control.predictor = list(compute=TRUE))

p = inla.link.invcloglog(eta)
y = rbinom(n,  size = N, prob = p)
rrr = inla(y ~ 1 + z,  data = data.frame(y, z), family = "binomial", Ntrials = rep(N, n),
        control.family = list(control.link = list(model="cloglog")),
        control.predictor = list(compute=TRUE))
```

Other linkfunctions/models are also avilable from within R, see `?inla.link`

How can I do predictions using INLA?

In INLA there is no function `predict` as for `glm/lm` in R. Predictions must to done as a part of the model fitting itself. As prediction is the same as fitting a model with some missing data, we can simply set `y[i] = NA` for those "locations" we want to predict. Here is a simple example

```
n = 100
n.pred = 10
y = arima.sim(n=n, model=list(ar=0.9))
N = n + n.pred
yy = c(y, rep(NA, n.pred))
i = 1:N
formula = yy ~ f(i, model="ar1")
r = inla(formula, data = data.frame(i,yy),
        control.family = list(initial = 10, fixed=TRUE)) ## no observational noise
```

which gives predictions

```
r$summary.random$i[(n+1):N, c("mean", "sd") ]
```

```
##         mean       sd
## 101 2.820339 1.659864
## 102 2.552132 1.862689
## 103 2.314460 2.016377
## 104 2.103390 2.136153
## 105 1.915548 2.231183
## 106 1.748029 2.307544
## 107 1.598327 2.369503
## 108 1.464276 2.420175
## 109 1.344002 2.461896
## 110 1.235879 2.496449
```

Quantiles such like `r$summary.fitted.values` and `r$marginals.fitted.values`, if computed, use the identity link if `y[i] = NA` by default. If you want the `fitted.values` computed with a different link function, then there are two ways to doit.

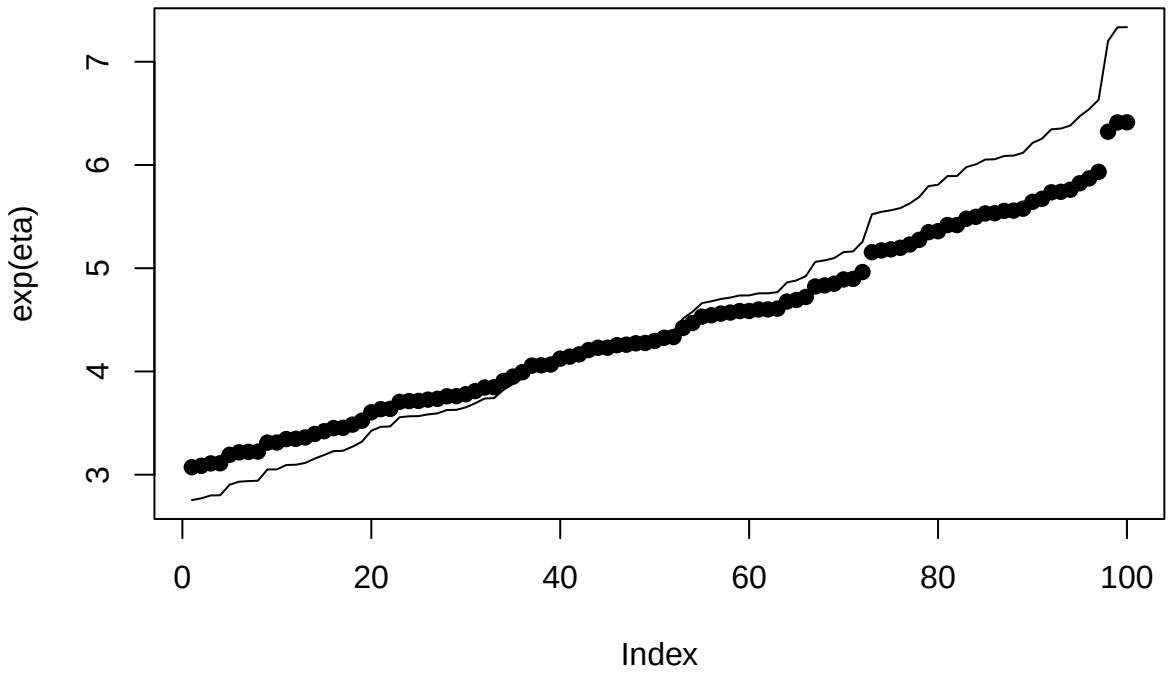
In the case you want to use the link-function from the likelihood already used (most often the case), there is the argument `link` in `control.predictor`. If the response `y[idx] = NA`, then set `link[idx] = 1`, to indicate that you want to compute that fitted value using the link function from `family[1]`. With several likelihoods, set `link[idx]` to family-index which is correct, ie the column number in the response. The following example shows the usage:

```
## simple poisson regression
n = 100
x = sort(runif(n))
eta = 1 + x
```

```
lambda = exp(eta)
y = rpois(n, lambda = lambda)

## missing values:
y[1:3] = NA
y[(n-2):n] = NA

## link = 1 is a shortcut for rep(1, n) where n is the appropriate
## length. here '1' is a reference to the first 'family', ie
## 'family[1]'
r = inla(y ~ 1 + x,  family = "poisson",
        data = data.frame(y, x),
        control.predictor = list(link = 1))
plot(exp(eta),type ="l")
points(r$summary.fitted.values$mean, pch=19)
```



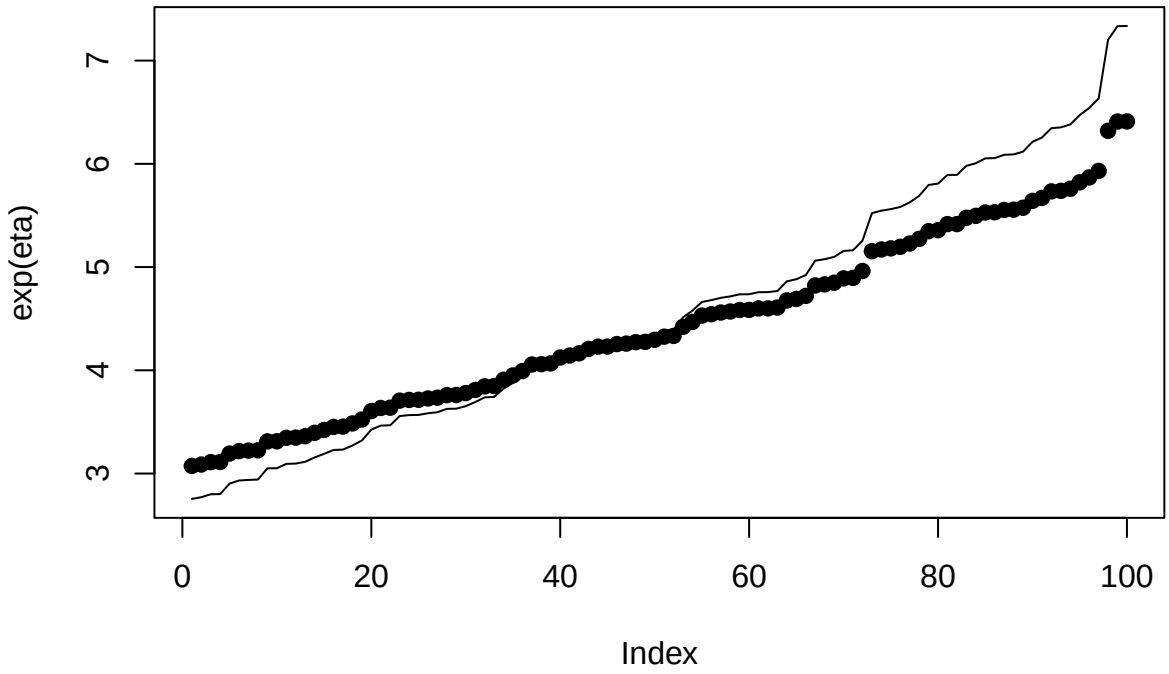
We only need to define `link` where there are missing values. Entries for which the observation is not `NA`, is ignored.

For more than one likelihood, use '2' to refer to the second likelihood. Here is an example where we split the data in two, and assign the second half the `nbinomial` distribution.

```
n2 = n %/% 2L
Y = matrix(NA, n, 2)
Y[1:n2, 1] = y[1:n2]
Y[1:n2 + n2, 2] = y[1:n2 + n2]
link = rep(NA, n)
```

```
link[which(is.na(y[1:n2]))] = 1
link[n2  + which(is.na(y[1:n2 + n2]))] = 2

r = inla(Y ~ 1 + x,  family = c("poisson", "nbinomial"),
        data = list(Y=Y, x=x),
        control.predictor = list(link = link))
plot(exp(eta),type ="l")
points(r$summary.fitted.values$mean, pch=19)
```



We can transform marginals manually using the function `inla.marginal.transform` or compute expectations using `inla.emarginal`, like in this example (taken from `demo(Tokyo)`).

```
## Load the data
data(Tokyo)
summary(Tokyo)
```

```
##        y                n              time
##  Min.   :0.0000   Min.   :1.000   Min.   :  1.00
##  1st Qu.:0.0000   1st Qu.:2.000   1st Qu.: 92.25
##  Median :0.0000   Median :2.000   Median :183.50
##  Mean   :0.5246   Mean   :1.997   Mean   :183.50
##  3rd Qu.:1.0000   3rd Qu.:2.000   3rd Qu.:274.75
##  Max.   :2.0000   Max.   :2.000   Max.   :366.00
```

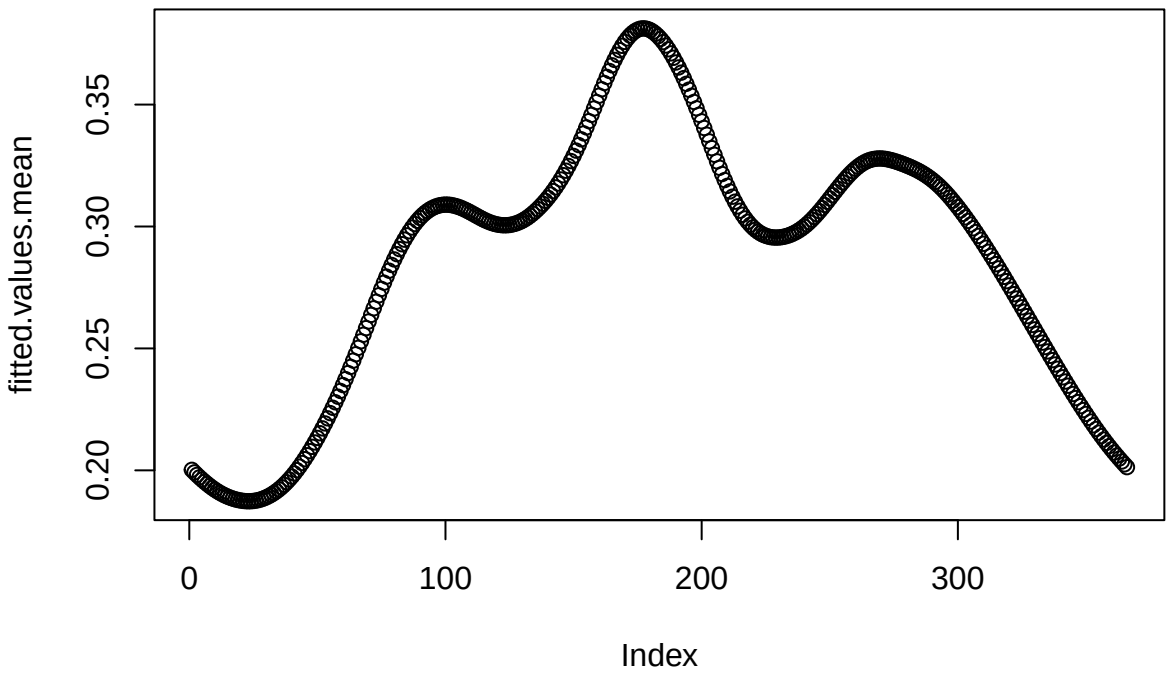
```
Tokyo$y[300:366] <- NA

## Define the model
```

```
formula = y ~ f(time, model="rw2", scale.model=TRUE,
                constr=FALSE, cyclic=TRUE,
                hyper = list(prec=list(prior="pc.prec",
                                       param=c(2,0.01)))) -1

## We'll get a warning since we have not defined the link argument
result = inla(formula, family="binomial", Ntrials=n, data=Tokyo,
    control.compute = list(return.marginals.predictor = TRUE),
    control.predictor=list(compute=T))

## need to recompute the fitted values for those with data[i] = NA,
## as the identity link is used.
n = 366
fitted.values.mean = numeric(n)
for(i in 1:366) {
    if (is.na(Tokyo$y[i])) {
        if (FALSE) {
            ## either like this, which is slower
            marg = inla.marginal.transform(
                           function(x) exp(x)/(1+exp(x)),
                           result$marginals.fitted.values[[i]] )
            fitted.values.mean[i] = inla.emarginal(function(x) x, marg)
        } else {
            ## or like this,  which is faster
            fitted.values.mean[i] = inla.emarginal(
                           function(x) exp(x)/(1 +exp(x)),
                           result$marginals.fitted.values[[i]])
        }
    } else {
        fitted.values.mean[i] = result$summary.fitted.values[i,"mean"]
    }
}
plot(fitted.values.mean)
```



Some of the models needs a graph, how do I specify it?

Some of the models in `INLA` needs the user to specify a graph, saying which nodes are neighbours to each other. A 'graph' can be specified in three different ways.

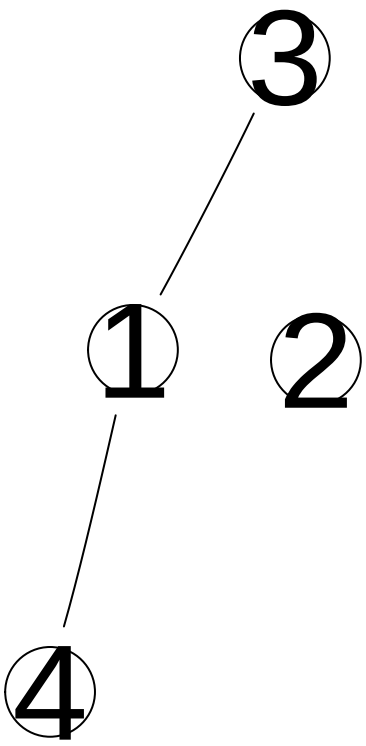
- As an ascii or binary file with a graph specification, or the same contents given as (possible list of) mix of character and numerics arguments.
- As a symmetric (dense or sparse) matrix, where the non-zero pattern of the matrix defines the graph.
- As an inla.graph-object

A graph defined in an ascii-file, must have the following format. The first entry is the number of nodes in the graph, n. The nodes in the graph are labelled $1, 2, \ldots, n$. The next entries, specify the number of neighbours and the neighbours for each node. A simple example is the following

```
4
1 2 3 4
2 0
3 1 1
4 1 1
```

This defines a graph with four nodes, where node 1 has 2 neighbours 3 and 4, node 2 as 0 neighbours, node 3 has 1 neighbour 1, and node 4 has 1 neighbour 1, and the graph looks like this

```
g = inla.read.graph("4 1 2 3 4 2 0 3 1 1 4 1 1")
plot(g)
```



Note that we need to specify the implied symmetry as well. In this example 4 is a neighbour of 1, then we also need to specify that 1 is a neighbour of 4.

Instead of storing the graph specification on a file, it can also be specified as a character string with the same contents as a file, like

```
"4 1 2 3 4  2 0 3 1 1 4 1 1"
```

as used in `inla.read.graph` above.

Due to imitations of the length of a string/line, so in practice, this way specifying the graph, seems more useful for teaching or demonstration purposes than for practical analysis.

Within `INLA`, this would look like

```
formula = y ~ f(idx, model = "besag", graph = "graph.dat")
```

or

```
formula = y ~ f(idx, model = "besag", graph = "4 1 2 3 4  2 0 3 1 1 4 1 1")
```

A graph can also be defined as a symmetric (dense or sparse) matrix, where the non-zero pattern of the matrix defines the graph A neighbour matrix is often used for defining which nodes that are neighbours, with the convention that if `Q[i,j] != 0` then i and j are neighbours if $i \neq j$.

For example, the (dense) matrix C

```
C = matrix(c(1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1),4,4)
C
```

```
##      [,1] [,2] [,3] [,4]
```

```
## [1,]    1    0    1    1
## [2,]    0    1    0    0
## [3,]    1    0    1    0
## [4,]    1    0    0    1
```

defines the same graph show above.

Since graphs tends to be large, we often define them as a sparse matrix

```
C.sparse= inla.as.sparse(C)
C.sparse
```

```
## 4 x 4 sparse Matrix of class "dgTMatrix"
##
## [1,] 1 . 1 1
## [2,] . 1 . .
## [3,] 1 . 1 .
## [4,] 1 . . 1
```

We can then use `graph=C` or `graph=C.sparse` in the formula.

We can also define the graph as an `inla.graph`-object, which is used internally, represent a graph. For example

```
str(g)
```

```
## List of 4
##  $ n   : int 4
##  $ nnbs: num [1:4] 2 0 1 1
##  $ nbs :List of 4
##   ..$ : int [1:2] 3 4
##   ..$ : num(0)
##   ..$ : int 1
##   ..$ : int 1
##  $ cc  :List of 3
##   ..$ id   : int [1:4] 1 2 1 1
##   ..$ n    : int 2
##   ..$ nodes:List of 2
##   .. ..$ : int [1:3] 1 3 4
##   .. ..$ : int 2
##  - attr(*, "class")= chr "inla.graph"
```

and use `graph = g` as the argument.

The internal format, are as follows. `n` is the size of the graph. `nnbs` are the number of neighbours to each node, `nbs` list all the neighbours to each node, and the class is `inla.graph`. The `cc`-list is for internal use only and specify the connected components in the graph.

`INLA` has some functions to work with graphs, and here is a short summary.

- `inla.read.graph()` and `inla.write.graph()`, read and write graphs using any of the graph specifications above.
- You can plot a `inla.graph`-object using `plot()` and get a summary using `summary()`. The plotting requires the `Rgraphviz` package.
- From a graph specification, you can generate the neighbour matrix, using `inla.graph2matrix()`. You can plot a graph specification as a neighbour matrix, using `inla.spy()`
- If you have 'errors' in your graph, you may read it using `inla.debug.graph()`. This is only available for a graph specification in an ascii-file.

How INLA deal with NA

For a formula like

```
formula = y ~ x + f(k, model= <some model>)
```

then NA's in either y, x or k are treated differently.

NA's in the response y. If `y[i] = NA`, this means that `y[i]` is not observed, hence gives no contribution to the likelihood.

NA's in fixed effect x. If `x[i] = NA` this means that `x[i]` is not part of the linear predictor for `y[i]`. For fixed effects, this is equivalent to `x[i]=0`, hence internally we make this change: `x[is.na(x)] = 0`.

NA's in random effect k. If `k[i] = NA`, this means that the random effect does not contribute to the linear predictor for `y[i]`.

NA's in a factor x. NA's in a factor x is not allowed unless NA is a level in itself, or

```
control.fixed = list(expand.factor.strategy = "inla")
```

is set. With this option, then NA is interpreted similarly as a fixed effect, where NA means no contribution from x. The effect of `expand.factor.strategy="inla"`, is best explained with an example.

```
r = inla(y ~ 1 + x, data = data.frame(y=1:3, x=factor(c("a","b","c"))))
as.matrix(r$model.matrix)
```

```
##   (Intercept) xb xc
## 1           1  0  0
## 2           1  1  0
## 3           1  0  1
```

for default value of the argument `contrasts`. The effect of `xa` is removed to make the corresponding matrix non-singular. If we want to expand `x` into each of each three effects, then we can do

```
r = inla(y ~ 1 + x, data = data.frame(y=1:3,x=factor(c("a","b","c"))),
         control.fixed = list(expand.factor.strategy="inla"))
as.matrix(r$model.matrix)
```

```
##   (Intercept) xa xb xc
## 1           1  1  0  0
## 2           1  0  1  0
## 3           1  0  0  1
```

As we see, each level of the factor is now treated symetrically. Although the corresponding frequentist matrix is singular as we have confounding with the intercept, the Bayesian posterior is still proper with proper priors.

With a NA in `x`, we get

```
r = inla(y ~ 1 + x, data = data.frame(y=1:3,x=factor(c("a","b",NA))),
         control.fixed = list(expand.factor.strategy="inla"))
as.matrix(r$model.matrix)
```

```
##   (Intercept) xa xb
## 1           1  1  0
## 2           1  0  1
## 3           1  0  0
```

so that the 3rd element of the linear predictor has no contribution from `x`, as it should.

Can INLA deal with missing covariates?

No, INLA has no generic way to "impute" or integrate-out missing covariates. You have to adjust your model to account for missing covariates, like using one of the measurement error models ("meb", "mec"), or construct a joint model for the data and the covariates, but this is case-specific.

Compute cross-validation or predictive measures of fit

`INLA` provides two types of leave-one-out predictive measures of fit. It is the CPO value, which is

$$\text{Prob}(y_i|y_{-i}),$$

the PIT value

$$\text{Prob}(y_i^{\text{new}} \leq y_i|y_{-i})$$

To enable the computation of these quantities, you will need to add the argument

```
control.compute=list(cpo=TRUE)
```

We can also compute PO values

$$\text{Prob}(y_i|y),$$

when argument `po=TRUE` is added.

If the resulting object is `result`, then you will find the predictive quantities as `result$cpo$cpo` and `result$cpo$pit`.

Implicit assumptions made in for computations, and there are internal checks that these are satisfied. The results of these checks will appear as `result$cpo$failure`. In short, if `result$cpo$failure[i] > 0` then some assumption is violated, the higher the value (maximum 1) the more seriously. If `result$cpo$failure[i] == 0` then the assumptions should be ok.

You *may* want to recompute those with non-zero failure. However, this must be done *manually* by removing `y[i]` from the dataset, fit the model and then predict `y[i]`. To provide a more efficient implementation of this, we have provided

```
improved.result = inla.cpo(result)
```

which take an inla-object which is the output from `inla()`, and recompute (in an efficient way) the cpo/pit for which `result$cpo$failure > 0`, and return 'result' with the improved estimates of cpo/pit. See `?inla.cpo` for details.

I have access to a remote Linux/MacOS server, is it possible to run the computations remotely and running R locally?

Yes! This option allow `INLA` to use a remote server to do the computations. In order to use this feature, you need to do some setup which is different from (various) Linux distributions, Mac and Windows. In short:

- install `R` and `R-INLA` a remote server, for example `foo.bar.org`.
- Install your public ssh-key on `foo.bar.org` to setup password free access to the remove server using `ssh`. And please check that this is indeed working before moving forward!
- On your local host, run `inla.remote()` to initialise the init-file `~/.inlarc` and then edit this file to fit your needs.
- You may now have to log out and log in again, to make sure your ssh key is signed out.
- You can now use option `inla.call="remote"` to do the computations on your remote server, or set this globally with `inla.setOption("inla.call", "remote")`

You can also submit a job on the remote server, so you do not need to sit and wait for it to finish, but you can collect the results later. Basically, you do

```
r = inla(..., inla.call = "submit")
```

which will start the job on the server. You can start many jobs, and list them using

```
inla.qstat()
```

and you can fetch the results (for the job above) using

```
r = inla.qget(r)
```

You can also delete jobs and fetch the jobs from another machine; see `?inla.q` for further details.

HOWTO setup ssh-keys: For the unexperienced user, this is somewhat tricky; sorry about that. The easiest is to find a friend that knows this and can help you. Newer system do a lot of these things very nicely these days.

It is also possible to setup this from Windows using CYGWIN, and INLA can work with this interface as well. Please see the old web-page for details, which are long and technical. HOWEVER, I am no longer convinced that this work anymore, as I haven't seen this is use for years. It is much much easier to use a virtual machine with Linux on Windows.

Posteriors for linear combinations

I have some linear combinations of the nodes in the latent field that I want to compute the posterior marginal of, is that possible? Yes! These are called 'linear combinations'. There are handy functions, 'inla.make.lincomb()' and 'inla.make.lincombs()', to define one or many such linear combinations. Single linear combinations made by using 'inla.make.lincomb()' can easily be joined into many. Its use is easiest explained using a rather long example...

Here is the example, that explains these features.

```
## A simple model
n = 100
a = rnorm(n)
b = rnorm(n)
idx = 1:n

y = rnorm(n) + a + b
formula = y ~ 1 + a + b + f(idx, model="iid")

## assume we want to compute the posterior for
##
##  2 * beta_a + 3 * beta_b + idx[1] - idx[2]
##
## which we specify as follows (and giving it a unique name)

lc1 = inla.make.lincomb(a=2, b=3, idx = c(1,-1,rep(NA,n-2)))
names(lc1) = "lc1"

## strictly speaking, it is sufficient to use `idx = c(1,-1)', as the
## remaining idx's are not used in any case.

r = inla(formula, data = data.frame(a,b,y),
        ## add the linear combinations here
        lincomb = lc1,
```

```
        ## force noise variance to be essiatially zero
        control.family = list(initial=10, fixed=TRUE))

## to verify the result, we can compare the mean but the variance and
## marginal cannot be computed from the simpler marginals alone.
lc1.1 = 2 * r$summary.fixed["a", "mean"] + 3 * r$summary.fixed["b",
    "mean"] + r$summary.random$idx$mean[1] -
    r$summary.random$idx$mean[2]
lc1.2= r$summary.lincomb.derived$mean
print(round(c(lc1.1 = lc1.1, lc1.2 = lc1.2), dig=3))
```

```
## lc1.1 lc1.2
## 5.152 5.152
```

The marginals are available as `r$marginals.lincomb$...`

There is an another function which is handy for specifying many linear combinations at once, that is `inla.make.lincombs()` (note the plural s). Here each 'row' define one linear combination

```
## let wa and wb be vectors, and we want to compute the marginals for
## beta_a * wa[i] + beta_b * wb[i], for i=1..m. this is done
## conveniently as follows

m = 10
wa = runif(m)
wb = runif(m)
lc.many = inla.make.lincombs(a = wa, b=wb)

## we can give them names as well, but there are also default names, like
print(names(lc.many))
```

```
##  [1] "lc01" "lc02" "lc03" "lc04" "lc05" "lc06" "lc07" "lc08" "lc09" "lc10"
```

```
r = inla(formula, data = data.frame(a,b,y),
        lincomb = lc.many,
        control.family = list(initial=10, fixed=TRUE))
print(round(r$summary.lincomb.derived, dig=3))
```

```
##      ID  mean    sd 0.025quant 0.5quant 0.975quant  mode kld
## lc01  1 0.402 0.040      0.324    0.402      0.481 0.402   0
## lc02  2 1.096 0.088      0.924    1.096      1.269 1.096   0
## lc03  3 0.945 0.077      0.794    0.945      1.096 0.945   0
## lc04  4 1.158 0.121      0.919    1.158      1.396 1.158   0
## lc05  5 0.389 0.034      0.323    0.389      0.455 0.389   0
## lc06  6 1.570 0.120      1.334    1.570      1.807 1.570   0
## lc07  7 1.453 0.128      1.201    1.453      1.704 1.453   0
## lc08  8 1.074 0.084      0.908    1.074      1.240 1.074   0
## lc09  9 0.974 0.073      0.830    0.974      1.117 0.974   0
## lc10 10 1.495 0.120      1.260    1.495      1.731 1.495   0
```

Terms like 'idx' above, can be added as `idx = IDX` into `inla.make.lincombs()`, where IDX is a matrix. Again, each column of the arguments define one linear combination.

There is a further option available for the derived linear combinations, that is the option to compute also the posterior correlation matrix between all the linear combinations. To activate this option, use

```
control.inla = list(lincomb.derived.correlation.matrix = TRUE)
```

and you will find the resulting posterior correlation matrix as

```
result$misc$lincomb.derived.correlation.matrix
```

Here is a small example where we compute the correlation matrix for the predicted values of a hidden AR(1) model with an intercept.

```
n = 100
nPred = 10
phi = 0.9
x = arima.sim(n,  model = list(ar=phi)) * sqrt(1-phi^2)
y = 1 + x + rnorm(n,  sd=0.1)

time = 1:(n + nPred)
Y = c(y, rep(NA, nPred))
formula = Y ~ 1 + f(time, model="ar1")

## make linear combinations which are the nPred linear predictors
B = matrix(NA, nPred, n+nPred)
for(i in 1:nPred) {
    B[i, n+i] = 1
}
lcs = inla.make.lincombs(Predictor = B)

r = inla(formula,  data = data.frame(Y, time),
        control.predictor = list(compute=TRUE),
        lincomb = lcs,
        control.inla = list(lincomb.derived.correlation.matrix=TRUE))

print(round(r$misc$lincomb.derived.correlation.matrix,dig=3))
```

```
##       lc01  lc02  lc03  lc04  lc05  lc06  lc07  lc08  lc09  lc10
## lc01 1.000 0.608 0.425 0.315 0.241 0.189 0.151 0.123 0.102 0.086
## lc02 0.608 1.000 0.697 0.514 0.391 0.305 0.243 0.196 0.161 0.135
## lc03 0.425 0.697 1.000 0.734 0.556 0.431 0.340 0.273 0.223 0.184
## lc04 0.315 0.514 0.734 1.000 0.753 0.580 0.454 0.362 0.292 0.240
## lc05 0.241 0.391 0.556 0.753 1.000 0.764 0.594 0.469 0.376 0.305
## lc06 0.189 0.305 0.431 0.580 0.764 1.000 0.771 0.604 0.479 0.386
## lc07 0.151 0.243 0.340 0.454 0.594 0.771 1.000 0.776 0.610 0.486
## lc08 0.123 0.196 0.273 0.362 0.469 0.604 0.776 1.000 0.779 0.615
## lc09 0.102 0.161 0.223 0.292 0.376 0.479 0.610 0.779 1.000 0.782
## lc10 0.086 0.135 0.184 0.240 0.305 0.386 0.486 0.615 0.782 1.000
```

INLA seems to work great for near all cases, but are there cases where INLA is known to have problems?

The methodology needs the full conditional density for the latent field to be "near" Gaussian. This is usually achived by either replications or smoothing/"borrowing strength". A simple example which do not have this, is the following:

```
n = 100
u = rnorm(n)
eta = 1 + u
p = exp(eta)/(1+exp(eta))
y = rbinom(n, size=1,  prob = p)
```

```
idx = 1:n
result = inla(y ~ 1 + f(idx, model="iid",
                        hyper = list(prec = list(prior="pc.prec",
                                                 prior = c(1,0.01)))),
              data =data.frame(y,idx), family = "binomial",
              Ntrials = 1)
summary(result)
```

```
##
## Call:
##    c("inla(formula = y ~ 1 + f(idx, model = \"iid\", hyper = list(prec =
##    list(prior = \"pc.prec\", ", " prior = c(1, 0.01)))), family =
##    \"binomial\", data = data.frame(y, ", " idx), Ntrials = 1)")
## Time used:
##     Pre = 0.401, Running = 0.144, Post = 0.0102, Total = 0.556
## Fixed effects:
##              mean    sd 0.025quant 0.5quant 0.975quant  mode kld
## (Intercept) 0.407 0.205       0.01    0.406      0.813 0.402   0
##
## Random effects:
##   Name     Model
##     idx IID model
##
## Model hyperparameters:
##                         mean         sd 0.025quant 0.5quant 0.975quant  mode
## Precision for idx 84075.62 1408395.78       9.19   211.28  143336.70 13.13
##
## Marginal log-Likelihood:  -68.05
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

For each binary observation there is an iid "random effect" u, and there is no smoothing/"borrowing strength'' (apart from the weak intercept). If you plot the loglikelihood for eta for $y = 1$, say, then its an increasing function for increasing eta, so the likelihood itself would like $\eta = \infty$. With an unknown precision for u we run into problems; INLA has a tendency to estimate a to high precision for u. However, it must be noted that the model is almost singular and you'll have a strong prior sensitivity in the (exact) results as well. There is a similar discussion in here as well for the Salamander data example.

Can I have the linear predictor from one model as a covariate in a different model?

Yes, this is possible. Essentially, you have to set the linear predictor for the first model equal to 'u', and then you can copy 'u' and use the scaling to get the regression coefficient. A simple example will illustrate the idea:

```
## simple example
n = 100
x1 = rnorm(n)
eta1 = 1 + x1
x2 = rnorm(n)
eta2 = 2 + 2*eta1 + 2*x2
y1 = rnorm(n, mean=eta1, sd = 0.01)
y2 = rnorm(n, mean=eta2, sd = 0.01)

## the trick is to create a vector 'u' (iid) which is
## equal to eta1, and then we can copy 'u' to
```

```
## create beta*u or beta*eta1. we do this by
## using 0 = eta1 -u + tiny.noise

formula = Y ~ -1 + intercept1 + X1 + intercept2 + f(u, w, model="iid",
    hyper = list(prec = list(initial = -6, fixed=TRUE))) + f(b.eta2,
    copy="u", hyper = list(beta = list(fixed = FALSE))) + X2

Y = matrix(NA, 3*n, 3)

## part 1: y1
intercept1 = rep(1, n)
X1 = x1
intercept2 = rep(NA, n)
u = rep(NA, n)
w = rep(NA, n)
b.eta2 = rep(NA, n)
X2 = rep(NA, n)
Y[1:n, 1] = y1

## part 2: 0 = eta1 - u + tiny.noise
intercept1 = c(intercept1,  intercept1)
X1 = c(X1,  x1)
intercept2 = c(intercept2,  rep(NA, n))
u = c(u, 1:n)
w = c(w, rep(-1, n))
b.eta2 = c(b.eta2,  rep(NA, n))
X2 = c(X2,  rep(NA, n))
Y[n + 1:n, 2] = 0

## part 3: y2
intercept1 = c(intercept1,  rep(NA, n))
X1 = c(X1,  rep(NA, n))
intercept2 = c(intercept2, rep(1, n))
u = c(u, rep(NA, n))
w = c(w, rep(NA, n))
b.eta2 = c(b.eta2, 1:n)
X2 = c(X2, x2)
Y[2*n + 1:n, 3] = y2

r = inla(formula,
    data =  list(Y=Y, intercept1=intercept1, X1=X1,
        intercept2=intercept2, u=u, w=w, b.eta2=b.eta2, X2=X2),
    family = rep("gaussian", 3),
    control.inla = list(h = 1e-3),
    control.family = list(
        list(),
        list(hyper = list(prec = list(initial = 10, fixed=TRUE))),
        list()))

summary(r)
```

```
##
## Call:
##    c("inla(formula = formula, family = rep(\"gaussian\", 3), data = list(Y
```

```
##    = Y, ", " intercept1 = intercept1, X1 = X1, intercept2 = intercept2, ",
##    " u = u, w = w, b.eta2 = b.eta2, X2 = X2), control.family =
##    list(list(), ", " list(hyper = list(prec = list(initial = 10, fixed =
##    TRUE))), ", " list()), control.inla = list(h = 0.001))")
## Time used:
##     Pre = 0.483, Running = 0.366, Post = 0.0195, Total = 0.868
## Fixed effects:
##             mean    sd 0.025quant 0.5quant 0.975quant  mode kld
## intercept1 0.999 0.001      0.997    0.999      1.000 0.999   0
## X1         1.002 0.001      1.000    1.002      1.003 1.002   0
## intercept2 2.005 0.003      1.999    2.005      2.011 2.005   0
## X2         1.999 0.002      1.996    1.999      2.001 1.999   0
##
## Random effects:
##   Name     Model
##     u IID model
##    b.eta2 Copy
##
## Model hyperparameters:
##                                           mean      sd 0.025quant 0.5quant
## Precision for the Gaussian observations    15431.23 154.394   14880.44 15458.90
## Precision for the Gaussian observations[3]      Inf     NaN       0.00     0.00
## Beta for b.eta2                               2.00   0.002       1.99     2.00
##                                           0.975quant     mode
## Precision for the Gaussian observations      15941.34 15381.62
## Precision for the Gaussian observations[3]        Inf      NaN
## Beta for b.eta2                                  2.00     2.00
##
## Marginal log-Likelihood:  211.38
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

Latent models, likelihoods and priors.

The list of latent models, likelihood and priors implemented, can be found by doing (or give a spesific section, see `?inla.list.models`)

```
inla.list.models()
```

```
## Section [group]
##      ar                          AR(p) correlations
##      ar1                         AR(1) correlations
##      besag                       Besag model
##      exchangeable                Exchangeable correlations
##      exchangeablepos             Exchangeable positive correlations
##      iid                         Independent model
##      rw1                         Random walk of order 1
##      rw2                         Random walk of order 2
##  Section [hazard]
##      iid                         An iid model for the log-hazard
##      rw1                         A random walk of order 1 for the log-hazard
##      rw2                         A random walk of order 2 for the log-hazard
##  Section [latent]
##      2diid                       (This model is obsolute)
##      ar                          Auto-regressive model of order p (AR(p))
```

```
##      ar1                          Auto-regressive model of order 1 (AR(1))
##      ar1c                         Auto-regressive model of order 1 w/covariates
##      besag                        The Besag area model (CAR-model)
##      besag2                       The shared Besag model
##      besagproper                  A proper version of the Besag model
##      besagproper2                 An alternative proper version of the Besag model
##      bym                          The BYM-model (Besag-York-Mollier model)
##      bym2                         The BYM-model with the PC priors
##      clinear                      Constrained linear effect
##      copy                         Create a copy of a model component
##      crw2                         Exact solution to the random walk of order 2
##      dmatern                      Dense Matern field
##      fgn                          Fractional Gaussian noise model
##      fgn2                         Fractional Gaussian noise model (alt 2)
##      generic                      A generic model
##      generic0                     A generic model (type 0)
##      generic1                     A generic model (type 1)
##      generic2                     A generic model (type 2)
##      generic3                     A generic model (type 3)
##      iid                          Gaussian random effects in dim=1
##      iid1d                        Gaussian random effect in dim=1 with Wishart prior
##      iid2d                        Gaussian random effect in dim=2 with Wishart prior
##      iid3d                        Gaussian random effect in dim=3 with Wishart prior
##      iid4d                        Gaussian random effect in dim=4 with Wishart prior
##      iid5d                        Gaussian random effect in dim=5 with Wishart prior
##      iidkd                        Gaussian random effect in dim=k with Wishart prior
##      intslope                     Intecept-slope model with Wishart-prior
##      linear                       Alternative interface to an fixed effect
##      log1exp                      A nonlinear model of a covariate
##      logdist                      A nonlinear model of a covariate
##      matern2d                     Matern covariance function on a regular grid
##      meb                          Berkson measurement error model
##      mec                          Classical measurement error model
##      ou                           The Ornstein-Uhlenbeck process
##      revsigm                      Reverse sigmoidal effect of a covariate
##      rgeneric                     Generic latent model spesified using R
##      rw1                          Random walk of order 1
##      rw2                          Random walk of order 2
##      rw2d                         Thin-plate spline model
##      rw2diid                      Thin-plate spline with iid noise
##      seasonal                     Seasonal model for time series
##      sigm                         Sigmoidal effect of a covariate
##      slm                          Spatial lag model
##      spde                         A SPDE model
##      spde2                        A SPDE2 model
##      spde3                        A SPDE3 model
##      z                            The z-model in a classical mixed model formulation
##  Section [likelihood]
##      agaussian                    The aggregated Gaussian likelihoood
##      beta                         The Beta likelihood
##      betabinomial                 The Beta-Binomial likelihood
##      betabinomialna               The Beta-Binomial Normal approximation likelihood
##      bgev                         The blended Generalized Extreme Value likelihood
##      binomial                     The Binomial likelihood
```

```
##      cbinomial                The clustered Binomial likelihood
##      cenpoisson               Then censored Poisson likelihood
##      cenpoisson2              Then censored Poisson likelihood (version 2)
##      circularnormal           The circular Gaussian likelihoood
##      coxph                    Cox-proportional hazard likelihood
##      dgp                      Discrete generalized Pareto likelihood
##      exponential              The Exponential likelihood
##      exponentialsurv          The Exponential likelihood (survival)
##      fmri                     fmri distribution (special nc-chi)
##      fmrisurv                 fmri distribution (special nc-chi)
##      gamma                    The Gamma likelihood
##      gammacount               A Gamma generalisation of the Poisson likelihood
##      gammajw                  A special case of the Gamma likelihood
##      gammajwsurv              A special case of the Gamma likelihood (survival)
##      gammasurv                The Gamma likelihood (survival)
##      gaussian                 The Gaussian likelihoood
##      gev                      The Generalized Extreme Value likelihood
##      gompertz                 gompertz distribution
##      gompertzsurv             gompertz distribution
##      gp                       Generalized Pareto likelihood
##      gpoisson                 The generalized Poisson likelihood
##      iidgamma                 (experimental)
##      iidlogitbeta             (experimental)
##      loggammafrailty          (experimental)
##      logistic                 The Logistic likelihoood
##      loglogistic              The loglogistic likelihood
##      loglogisticsurv          The loglogistic likelihood (survival)
##      lognormal                The log-Normal likelihood
##      lognormalsurv            The log-Normal likelihood (survival)
##      logperiodogram           Likelihood for the log-periodogram
##      nbinomial                The negBinomial likelihood
##      nbinomial2               The negBinomial2 likelihood
##      nmix                     Binomial-Poisson mixture
##      nmixnb                   NegBinomial-Poisson mixture
##      poisson                  The Poisson likelihood
##      poisson.special1         The Poisson.special1 likelihood
##      pom                      Likelihood for the proportional odds model
##      qkumar                   A quantile version of the Kumar likelihood
##      qloglogistic             A quantile loglogistic likelihood
##      qloglogisticsurv         A quantile loglogistic likelihood (survival)
##      simplex                  The simplex likelihood
##      sn                       The Skew-Normal likelihoood
##      stochvol                 The Gaussian stochvol likelihood
##      stochvolnig              The Normal inverse Gaussian stochvol likelihood
##      stochvolsn               The SkewNormal stochvol likelihood
##      stochvolt                The Student-t stochvol likelihood
##      t                        Student-t likelihood
##      tstrata                  A stratified version of the Student-t likelihood
##      tweedie                  Tweedie distribution
##      weibull                  The Weibull likelihood
##      weibullcure              The Weibull-cure likelihood (survival)
##      weibullsurv              The Weibull likelihood (survival)
##      wrappedcauchy            The wrapped Cauchy likelihoood
##      xbinomial                The Binomial likelihood (expert version)
```

```
##      xpoisson                       The Poisson likelihood (expert version)
##      zeroinflatedbetabinomial0      Zero-inflated Beta-Binomial, type 0
##      zeroinflatedbetabinomial1      Zero-inflated Beta-Binomial, type 1
##      zeroinflatedbetabinomial2      Zero inflated Beta-Binomial, type 2
##      zeroinflatedbinomial0          Zero-inflated Binomial, type 0
##      zeroinflatedbinomial1          Zero-inflated Binomial, type 1
##      zeroinflatedbinomial2          Zero-inflated Binomial, type 2
##      zeroinflatedcenpoisson0        Zero-inflated censored Poisson, type 0
##      zeroinflatedcenpoisson1        Zero-inflated censored Poisson, type 1
##      zeroinflatednbinomial0         Zero inflated negBinomial, type 0
##      zeroinflatednbinomial1         Zero inflated negBinomial, type 1
##      zeroinflatednbinomial1strata2 Zero inflated negBinomial, type 1, strata 2
##      zeroinflatednbinomial1strata3 Zero inflated negBinomial, type 1, strata 3
##      zeroinflatednbinomial2         Zero inflated negBinomial, type 2
##      zeroinflatedpoisson0           Zero-inflated Poisson, type 0
##      zeroinflatedpoisson1           Zero-inflated Poisson, type 1
##      zeroinflatedpoisson2           Zero-inflated Poisson, type 2
##      zeroninflatedbinomial2         Zero and N inflated binomial, type 2
##      zeroninflatedbinomial3         Zero and N inflated binomial, type 3
##  Section [link]
##      cauchit                        The cauchit-link
##      cloglog                        The complementary log-log link
##      default                        The default link
##      identity                       The identity link
##      inverse                        The inverse link
##      log                            The log-link
##      loga                           The loga-link
##      logit                          The logit-link
##      logitoffset                    Logit-link with an offset
##      loglog                         The log-log link
##      logoffset                      Log-link with an offset
##      neglog                         The negative log-link
##      pquantile                      The population quantile-link
##      probit                         The probit-link
##      quantile                       The quantile-link
##      robit                          Robit link
##      sn                             Skew-normal link
##      special1                       A special1-link function (experimental)
##      special2                       A special2-link function (experimental)
##      sslogit                        Logit link with sensitivity and specificity
##      tan                            The tan-link
##      test1                          A test1-link function (experimental)
##  Section [lp.scale]
##  NA Section [mix]
##      gaussian                       Gaussian mixture
##      loggamma                       LogGamma mixture
##      mloggamma                      Minus-LogGamma mixture
##  Section [predictor]
##      predictor                      (do not use)
##  Section [prior]
##      betacorrelation                Beta prior for the correlation
##      dirichlet                      Dirichlet prior
##      expression:                    A generic prior defined using expressions
##      flat                           A constant prior
```

```
##      gamma                          Gamma prior
##      gaussian                       Gaussian prior
##      invalid                        Void prior
##      jeffreystdf                    Jeffreys prior for the doc
##      linksnintercept                Skew-normal-link intercept-prior
##      logflat                        A constant prior for log(theta)
##      loggamma                       Log-Gamma prior
##      logiflat                       A constant prior for log(1/theta)
##      logitbeta                      Logit prior for a probability
##      logtgaussian                   Truncated Gaussian prior
##      logtnormal                     Truncated Normal prior
##      minuslogsqrtruncnormal         (obsolete)
##      mvnorm                         A multivariate Normal prior
##      none                           No prior
##      normal                         Normal prior
##      pc                             Generic PC prior
##      pc.alphaw                      PC prior for alpha in Weibull
##      pc.ar                          PC prior for the AR(p) model
##      pc.cor0                        PC prior correlation, basemodel cor=0
##      pc.cor1                        PC prior correlation, basemodel cor=1
##      pc.dof                         PC prior for log(dof-2)
##      pc.fgnh                        PC prior for the Hurst parameter in FGN
##      pc.gamma                       PC prior for a Gamma parameter
##      pc.gammacount                  PC prior for the GammaCount likelihood
##      pc.gevtail                     PC prior for the tail in the GEV likelihood
##      pc.matern                      PC prior for the Matern SPDE
##      pc.mgamma                      PC prior for a Gamma parameter
##      pc.prec                        PC prior for log(precision)
##      pc.range                       PC prior for the range in the Matern SPDE
##      pc.sn                          PC prior for the skew-normal
##      pc.spde.GA                     (experimental)
##      pom                            #classes-dependent prior for the POM model
##      ref.ar                         Reference prior for the AR(p) model, p<=3
##      table:                         A generic tabulated prior
##      wishart1d                      Wishart prior dim=1
##      wishart2d                      Wishart prior dim=2
##      wishart3d                      Wishart prior dim=3
##      wishart4d                      Wishart prior dim=4
##      wishart5d                      Wishart prior dim=5
##      wishartkd                      Wishart prior
##  Section [wrapper]
##      joint                          (experimental)
```

Copying a model

We often encounter the situation where an element of a model is needed more than once for each observation. One example is where

```
y = a + b*w + ...
```

for fixed weights w and where (a_i, b_i) is bivariate Normal and all 2-vectors are independent.

Using the model

```
f(idx, model="iid2d", n=2*m, ...)
```

provide a random vector v, say, with length $2m$ stored as

$$v = (a_1, a_2, ..., a_m, b_1, b_2, ..., b_m).$$

To implement this, we simply create an indentical copy of v, v^*, where $v == v^*$ (nearly). Using the copy-feature, we can do

```
idx = 1:m
idx.copy = m + 1:m
formula = y ~  f(idx, model="iid2d", n=2*m) + f(idx.copy, w, copy="idx") + ....
```

recalling that the first m elements is a and the last m elements are b, and where w are the weights.

The second `f()` terms define itself as a copy of `f(idx, ...)`, and it inherit some of its features, like `n` and `values`.

A copied model may also have an unknown scaling (hyperparameter), which is default fixed to be 1. In the following example, we will use this feature to estimate the unknown scaling (in this case, scaling is 2), assuming we know the precision for z.

```
n=1000
i=1:n
j = i
z = rnorm(n)
w = runif(n)
y = z  + 2*z*w + rnorm(n)
formula = y ~ f(i, model="iid",initial=0, fixed=T) +
              f(j, w, copy="i", fixed=FALSE)
r = inla(formula, data = data.frame(i,j,w,y))
summary(r)
```

```
##
## Call:
##    "inla(formula = formula, data = data.frame(i, j, w, y))"
## Time used:
##     Pre = 0.539, Running = 1.14, Post = 0.0389, Total = 1.72
## Fixed effects:
##               mean    sd 0.025quant 0.5quant 0.975quant   mode kld
## (Intercept) -0.071 0.068     -0.205   -0.071      0.062 -0.071   0
##
## Random effects:
##   Name     Model
##     i IID model
##    j Copy
##
## Model hyperparameters:
##                                          mean    sd 0.025quant 0.5quant
## Precision for the Gaussian observations 0.65 0.115      0.455    0.639
## Beta for j                              1.72 0.144      1.443    1.719
##                                         0.975quant  mode
## Precision for the Gaussian observations      0.908 0.617
## Beta for j                                   2.007 1.713
##
## Marginal log-Likelihood:  -2244.58
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

If the scaling parameter is within given range, then option \verb|range = c(low, high)|, can be given. In this case

```
beta = low + (high-low)*exp(beta.local)/(1+exp(beta.local))
```

and the prior is defined on `beta.local`.

If `low=high` or `range = NULL`, then the identity mapping is used. If `high=Inf` and |verb|low!=Inf|, then the mapping `low + exp(beta.local)` is used. The case low=Inf and high!=Inf is not yet implemented.

A model or a copied model can be copied several times. The degree of closeness of v and v^* is specified by the argument `precision`, as the precision of the noise added to v to get v^*.

Replicate a model

Independent replications of a model with the same hyperparmeters can be defined using the argument `replicate`,

```
f(idx, model = .., replicate = r)
```

Here, `r` is a vector of the same length as `idx`. In this case, we use a two-dimensional index to index this (sub-)model: `(idx, r)`, so `(1,2)` identify the first value of the second replication of this model (component). Number of replications are defined as `max(replicate)`, unless it is defined by the argument `nrep`.

One example is the model 'iid':

```
i = 1:n
formula = y ~  f(i, model = "iid") + ...
```

which has an alternative equivalent formulation as 'n' replications of an iid-model with length 1

```
i = rep(1,n)
r = 1:n
formula = y ~  f(i, model="iid", replicate = r) + ...
```

In the following example, we estimate the parameters in three AR(1) time-series with the same hyperparameters (ie its replicated) but with separate means:

```
n = 100
y1 = arima.sim(n=n, model=list(ar=c(0.9)))+10
y2 = arima.sim(n=n, model=list(ar=c(0.9)))+20
y3 = arima.sim(n=n, model=list(ar=c(0.9)))+30

formula = y ~ mean -1 + f(i, model="ar1", replicate=r)
y = c(y1,y2,y3)
i = rep(1:n, 3)
r = rep(1:3, each=n)
mean = as.factor(r)
result = inla(formula, family = "gaussian",
              data = data.frame(y, i, mean),
              control.family = list(initial = 12, fixed=TRUE))
summary(result)
```

```
##
## Call:
##    c("inla(formula = formula, family = \"gaussian\", data = data.frame(y,
##    ", " i, mean), control.family = list(initial = 12, fixed = TRUE))" )
## Time used:
##     Pre = 0.456, Running = 0.468, Post = 0.0168, Total = 0.941
```

```
## Fixed effects:
##         mean    sd 0.025quant 0.5quant 0.975quant   mode   kld
## mean1 10.088 1.465      7.207   10.071     13.072 10.047 0.001
## mean2 18.118 1.476     15.010   18.168     20.915 18.231 0.001
## mean3 31.066 1.483     27.909   31.128     33.839 31.206 0.001
##
## Random effects:
##   Name     Model
##     i AR1 model
##
## Model hyperparameters:
##                   mean    sd 0.025quant 0.5quant 0.975quant  mode
## Precision for i 0.143 0.044      0.070    0.139      0.241 0.131
## Rho for i       0.936 0.020      0.893    0.938      0.969 0.941
##
## Marginal log-Likelihood:  -431.54
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

All other arguments is interpreted for the basic model and also replicated. Like argument `constr=TRUE`, is interpreted as each replication sums to zero, and additional constraints are also replicated.

Models with more than one type of likelihood

There is no constraint in INLA that the type of likelihood must be the same for all observations. In fact, every observation could have its own likelihood. Extentions include more than one familily, like the Normal, Poisson, etc, but also having in the model groups of observations with separate hyperparameters within each group, where the family, for example, can be the same.

In the formula

```
y ~ a + 1
```

we allow `y` to be a matrix. In this case each column of `y` define one likelihood where the family is the same the hyperparameters are the same. For each row, only one of the columns could (but don't have to) have an observation (non-`NA` value), the other colums must have value `NA`. All other parameters to the likelihood, like `E` `Ntrials`, `offset` and `scale` are used as appropriate. Example: If row i column j is a Poission observation, then `E[i]` is used as the scaling. Similar with the others. This works as only one column for each row is non-`NA`.

The argument `family` is in the case where `y` is a matrix, a list of families. The argument `control.family` is then a list of lists; one for each family.

The first example, is a simple linear regression, where the first half of the data is observed with unknown precision `tau.1` (with a 'default' noninformative prior) and the second half of the data is observed with unknown precision `tau.2`. Otherwise, the two models have the same form for the linear predictor.

```
##  Simple linear regression with observations with two different
##  variances.
n = 100
N = 2*n
y = numeric(N)
x = rnorm(N)

y[1:n] = 1 + x[1:n] + rnorm(n, sd = 1/sqrt(1))
y[1:n + n] = 1 + x[1:n + n] + rnorm(n, sd = 1/sqrt(2))
```

```
Y = matrix(NA, N, 2)
Y[1:n, 1] = y[1:n]
Y[1:n + n, 2] = y[1:n + n]

formula = Y ~ x + 1
result = inla(
        formula,
        data = list(Y=Y, x=x),
        family = c("gaussian", "gaussian"),
        control.family = list(list(prior = "flat", param = numeric()),
                              list()))
summary(result)
```

```
##
## Call:
##    c("inla(formula = formula, family = c(\"gaussian\", \"gaussian\"), data
##    = list(Y = Y, ", " x = x), control.family = list(list(prior = \"flat\",
##    param = numeric()), ", " list()))")
## Time used:
##     Pre = 0.418, Running = 0.195, Post = 0.0181, Total = 0.631
## Fixed effects:
##              mean    sd 0.025quant 0.5quant 0.975quant  mode kld
## (Intercept) 1.029 0.051      0.929    1.029      1.128 1.029   0
## x           1.033 0.048      0.938    1.033      1.128 1.032   0
##
## Model hyperparameters:
##                                           mean    sd 0.025quant 0.5quant
## Precision for the Gaussian observations    0.998 0.142      0.744     0.99
## Precision for the Gaussian observations[2] 3.010 0.425      2.249     2.98
##                                           0.975quant  mode
## Precision for the Gaussian observations          1.30 0.976
## Precision for the Gaussian observations[2]       3.92 2.940
##
## Marginal log-Likelihood:  -248.42
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

The second example shows how to use information from two sources to estimate the effect of the covariate `x`.

```
## Simple example with two types of likelihoods
n = 10
N = 2*n

## common covariates
x = rnorm(n)

## Poisson, depends on x
E1 = runif(n)
y1 = rpois(n, lambda = E1*exp(x))

## Binomial, depends on x
size = sample(1:10, size=n, replace=TRUE)
prob = exp(x)/(1+exp(x))
y2 = rbinom(n, size= size, prob = prob)
```

```
## Join them together
Y = matrix(NA, N, 2)
Y[1:n, 1] = y1
Y[1:n + n, 2] = y2

## The E for the Poisson
E = numeric(N)
E[1:n] = E1
E[1:n + n] = NA

## Ntrials for the Binomial
Ntrials = numeric(N)
Ntrials[1:n] = NA
Ntrials[1:n + n] = size

## Duplicate the covariate which is shared
X = numeric(N)
X[1:n] = x
X[1:n + n] = x

## Formula involving Y as a matrix
formula = Y ~ X - 1

## `family' is now
result = inla(formula,
        family = c("poisson", "binomial"),
        data = list(Y=Y, X=X),
        E = E, Ntrials = Ntrials)
summary(result)
```

```
##
## Call:
##    c("inla(formula = formula, family = c(\"poisson\", \"binomial\"), data
##    = list(Y = Y, ", " X = X), E = E, Ntrials = Ntrials)")
## Time used:
##     Pre = 0.424, Running = 0.116, Post = 0.00552, Total = 0.546
## Fixed effects:
##    mean    sd 0.025quant 0.5quant 0.975quant  mode kld
## X 1.054 0.182      0.674    1.062       1.39 1.078   0
##
## Marginal log-Likelihood:  -28.28
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

If the covariate 'x' is different for the two families, `x` and `xx`, say, then we only need to make the following changes

```
X = numeric(N)
X[1:n] = x
X[1:n + n] = NA

XX = numeric(N)
XX[1:n] = NA
XX[1:n + n] = xx
```

```
formula = Y  ~ X + XX -1
```

and add `XX` into the data.frame. Note how we can express the joint model as a 'union' of models with the use of `NA`'s to remove terms.

In the next example, we use also the `replicate` feature to estimate three replicated AR(1) models with the same hyperparamters, each observed differently.

```
## An example with three independent AR(1)'s with separate means, but
## with the same hyperparameters. These are observed with three
## different likelihoods.

n = 100
x1 = arima.sim(n=n, model=list(ar=c(0.9))) + 0
x2 = arima.sim(n=n, model=list(ar=c(0.9))) + 1
x3 = arima.sim(n=n, model=list(ar=c(0.9))) + 2

## Binomial observations
Nt = 10 + rpois(n,lambda=1)
y1 = rbinom(n, size=Nt, prob = exp(x1)/(1+exp(x1)))

## Poisson observations
Ep = runif(n, min=1, max=10)
y2 = rpois(n, lambda = Ep*exp(x2))

## Gaussian observations
y3 = rnorm(n, mean=x3, sd=0.1)

## stack these in a 3-column matrix with NA's where not observed
y = matrix(NA, 3*n, 3)
y[1:n, 1] = y1
y[n + 1:n, 2] = y2
y[2*n + 1:n, 3] = y3

## define the model
r = c(rep(1,n), rep(2,n), rep(3,n))
rf = as.factor(r)
i = rep(1:n, 3)
formula = y ~ f(i, model="ar1", replicate=r, constr=TRUE) + rf -1
data = list(y=y, i=i, r=r, rf=rf)

## parameters for the binomial and the poisson
Ntrial = rep(NA, 3*n)
Ntrial[1:n] = Nt
E = rep(NA, 3*n)
E[1:n + n] = Ep

result = inla(formula, family = c("binomial", "poisson", "normal"),
              data = data, Ntrial = Ntrial, E = E,
              control.family = list(
                      list(),
                      list(),
                      list()))
summary(result)
```

```
##
## Call:
##    c("inla(formula = formula, family = c(\"binomial\", \"poisson\",
##    \"normal\"), ", " data = data, E = E, Ntrials = Ntrial, control.family
##    = list(list(), ", " list(), list()))")
## Time used:
##     Pre = 0.413, Running = 0.532, Post = 0.0173, Total = 0.963
## Fixed effects:
##       mean    sd 0.025quant 0.5quant 0.975quant   mode kld
## rf1 -0.480 0.095     -0.670   -0.478     -0.298 -0.475   0
## rf2  1.490 0.040      1.409    1.491      1.566  1.493   0
## rf3  1.107 0.001      1.105    1.107      1.109  1.107   0
##
## Random effects:
##   Name     Model
##     i AR1 model
##
## Model hyperparameters:
##                                                mean       sd 0.025quant
## Precision for the Gaussian observations[3] 1.74e+04 1.72e+04   1380.251
## Precision for i                            2.29e-01 4.60e-02      0.147
## Rho for i                                  8.61e-01 2.80e-02      0.802
##                                            0.5quant 0.975quant     mode
## Precision for the Gaussian observations[3] 1.24e+04   6.32e+04 3843.856
## Precision for i                            2.27e-01   3.26e-01    0.224
## Rho for i                                  8.62e-01   9.11e-01    0.863
##
## Marginal log-Likelihood:  -847.82
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

Models where the response/data depends on linear combinations of the "linear predictor" (or the sum of "fixed" and "random" effects)

In some cases, the data/response might depend on a linear combination of the "linear predictor" defined in the formula, like

```
y ~ 1 + z
```

then this implies that `y[1]` depends on `intercept + beta*z[1]`. Suppose if `y[1]` depends on `2*intercept + beta*z[1] + beta*z[2]`? Although it is possible to express this, using the tools we already have, it is more convenient to add another layer into the model. Let `A` be a `m x n` matrix, which defines new linear predictors, `eta~` from `eta`, like

```
eta~ = A %*% eta
```

Here, `eta` is the ordinary linear predictor defined using the formula, and the data depends on the linear predictor `eta~`. So we might express this as

```
y ~ 1 + z, with addition matrix A
```

meaning in short, that

```
y    ~ eta~   ## no intercept...
eta~ = A %*% eta
eta  = intercept + beta*z
```

This is specified by adding the A-matrix, using

```
control.predictor=list(A=A)
```

The argument `offset`, which might be defined in the formula as `offset(value)` or as an argument `inla(..., offset = value)`, does have different interpretation in the case where the A-matrix is used. The rule is that `offset` in the formula, goes into `eta`, whereas an argument `offset` goes into `eta~`. If we write out the expressions above adding offsets, `offset.formula` and `offset.arg`, we get

```
eta~ = A %*% eta + offset.arg
eta = intercept + beta*z + offset.formula
```

In the case where there is no A-matrix, then \verb|offset.total = offset.arg + offset.formula|.

The following example should provide more insight. You may change n and m, such that $m < n$, $m = n$ or $m > n$. Note that since the response has dimension m and the covariates dimension n, we need to use `list(y=y, z=z)` and not a `data.frame()`. This example also illustrates the use of offset's.

```
## 'm' is the number of observations of eta*, where eta* = A eta +
## offset.arg, and A is a fixed m x n matrix, and eta has length n. An
## offset in the formula goes into 'eta' whereas an offset in the
## argument of the inla-call, goes into eta*
n = 10
m = 100
offset.formula = 10+ 1:n
offset.arg = 1 + 1:m

## a covariate
z = runif(n)

## the linear predictor eta
eta = 1 + z + offset.formula

## the linear predictor eta* = A eta + offset.arg.
A = matrix(runif(n*m), m, n);
##A = inla.as.sparse(A)  ## sparse is ok
## need 'as.vector', as 'Eta' will be a sparseMatrix if 'A' is sparse
## even if ncol(Eta) = 1
Eta = as.vector(A %*% eta) + offset.arg

s = 1e-6
Y = Eta + rnorm(m, sd=s)

## for a check, we can use several offsets. here, m1=-1 and p1=1, so
## they m1+p1 = 0.
r = inla(Y ~ 1+z + offset(offset.formula) + offset(m1) + offset(p1),
        ## The A-matrix defined here
        control.predictor = list(A = A, compute=TRUE, precision = 1e6),
        ## we need to use a list() as the different lengths of Y
        ## and z
        data = list(Y=Y, z=z,
                m1 = rep(-1, n),
                p1 = rep(1, n),
                offset.formula = offset.formula,
                offset.arg = offset.arg),
        ## this is the offset defined in the argument of inla
```

```
        offset = offset.arg,
        ##
        control.family = list(initial = log(1/s^2), fixed=TRUE))
```

```
## Warning in inla(Y ~ 1 + z + offset(offset.formula) + offset(m1) + offset(p1), : The A-matrix in the p
##   but an intercept is in the formula. This will likely result
##   in the intercept being applied multiple times in the model, and is likely
##   not what you want. See ?inla.stack for more information.
##   You can remove the intercept adding ``-1'' to the formula.
```

```
summary(r)
```

```
##
## Call:
##    c("inla(formula = Y ~ 1 + z + offset(offset.formula) + offset(m1) + ",
##    " offset(p1), data = list(Y = Y, z = z, m1 = rep(-1, n), p1 = rep(1, ",
##    " n), offset.formula = offset.formula, offset.arg = offset.arg), ", "
##    offset = offset.arg, control.predictor = list(A = A, compute = TRUE, ",
##    " precision = 1e+06), control.family = list(initial = log(1/s^2), ", "
##    fixed = TRUE))")
## Time used:
##     Pre = 0.392, Running = 0.116, Post = 0.00636, Total = 0.515
## Fixed effects:
##             mean    sd 0.025quant 0.5quant 0.975quant mode kld
## (Intercept)    1 0.001      0.999        1      1.001    1   0
## z              1 0.001      0.997        1      1.003    1   0
##
## Marginal log-Likelihood:  -47639.10
## Posterior summaries for the linear predictor and the fitted values are computed
## (Posterior marginals needs also 'control.compute=list(return.marginals.predictor=TRUE)')
```

```
## this should be a small number
print(max(abs(r$summary.linear.predictor$mean - c(Eta, eta))))
```

```
## [1] 8.041647e-06
```

Here is a another example where the informal formula is

```
y = intercept + s[j] + 0.5*s[k] + noise
```

Instead of using the copy feature, we can implement this model using the A-matrix feature. What we do, is to first define a linear predictor being the intercept and s, then we use the A-matrix to 'construct the model'.

```
n = 100
s = c(-1, 0, 1)
nS = length(s)
j = sample(1L:nS, n, replace=TRUE)
k = j
k[j == 1L] = 2
k[j == 2L] = 3
k[k == 3L] = 1

noise = rnorm(n, sd=0.0001)
y = 1 + s[j] + 0.5*s[k] + noise

## build the formula such that the linear predictor is the intercept
## (index 1) and the 's' term (index 2:(n+1)). then kind of
```

```
## 'construct' the model using the A-matrix.
formula = y ~ -1 + intercept + f(idx)
A = matrix(0, n, nS+1L)
for(i in 1L:n) {
  A[i, 1L]         = 1
  A[i, 1L + j[i]] = 1
  A[i, 1L + k[i]] = 0.5
}

data = list(intercept = c(1, rep(NA, nS)), idx = c(NA, 1L:nS))
result = inla(formula, data=data, control.predictor=list(A=A))
## should be a straight line
plot(result$summary.random$idx$mean, s, pch=19)
abline(a=0,b=1)
```