Tutorial for the Different Terrains model in R-INLA

Haakon Bakka[1]

[1]Department of Mathematical Sciences, Norwegian University of Science and Technology,
<bakka@r-inla.org>

October 10, 2016

1 Description

This document is a knitr-based tutorial for the Different Terrains model Bakka et al. (2016). The code herein builds on the latest R-INLA version. You can download INLA in this way.

```
install.packages("INLA", repos="http://www.math.ntnu.no/inla/R/testing")
```

An up-to-date version of this tutorial can be found at `www.r-inla.org/barrier-models`, together with additional material. If there are any errors in the up-to-date version, when using the latest version of R-INLA, please do not hesitate to send me an email!

From here on out, the lines of code that are written in the tutorial are exactly those that are being run. To avoid a messy tutorial, a set of functions have been separated out. These functions are either 1) available in the R-file "TutorialDT_functions_S.R", or 2) added in an appendix of this document. For information about these functions, please see the last section of this document. Running all the code in this entire tutorial takes less than 5 minutes on my laptop.

Before we can start, we need

```
library(INLA)
source('TutorialDT_functions_S.R')
```

We need two additional functions to make the `knitr` output look good.

```
summary2 = function(x) {
# - This is a knitr specific tool
# - You may write summary(x) instead
  writeLines(strwrap(capture.output(summary(x)), width = 80))
}
print2 = function(x) {
# - This is a knitr specific tool
# - You may write print(x) instead
  writeLines(strwrap(capture.output(print(x)), width = 80))
}
```

2 Simulation and inference with the Barrier model

In this section we show an example of the Barrier model, with simulation, inference and plotting. The example terrain/map/study-area has a barrier going from west to east, and a gap in the middle of the barrier.

This section is a self-contained example of a good solution. However, while creating this solution I did several errors, that I suggest you avoid. In addition, I want to give some extra information on

how to do things that are slightly different. All of this you will find in the next section "Comments to simulation and inference example".

2.1 Input variables, mesh and barrier polygon

In this subsection we set up the input variables, create the finite element mesh, and the representation of the barrier.

First we set the input variables. Feel free to experiment with different values. Changing the `smalldist` can result in markedly different behaviour.

```
max.edge.length = 0.4
# - The coarseness of the finite element approximation
# - Corresponds to grid-square width in discretisations
# - - Except that finite element approximations are much better
# - Should be compared to size of study area
# - Should be less than a fourth of the estimated (posterior) spatial range
# - Computational time up to *8 when you halve this value
smalldist = 0.2
# - the width of the opening in the barrier
set.seed(2016)
set.inla.seed = 2016
```

We set up a square polygon `p` to define the barrier. In this example, I know I want to have a $[0, 10] \times [0, 10]$ study area, and I construct the barrier to fit inside this. In general, to use theese barrier models, you need the barrier region (e.g. land) to be represented by a polygon. In most cases, you will use shapefiles or maps or other sources for determining your barrier. In the next section we will present an example using `maps`. Talking about polygons in general, however, is out of the scope of this tutorial. More information about polygons can be found in the R-packages `sp` and `maps`.

```
width = .5
# - The width/thickness of the barrier
p2 = bakka.square.polygon(xlim=c(-1, 5-smalldist/2), ylim=5+width*c(-.5, .5),
                          ret.SP = T)
p3 = bakka.square.polygon(xlim=c(5+smalldist/2, 11), ylim=5+width*c(-.5, .5),
                          ret.SP = T)
p = SpatialPolygons(c(p2@polygons, p3@polygons))
# - The total barrier area polygon
# - Use plot(p) to investigate
```

We set up the `mesh` which will be the numerical representation of the spatial study region. We will not describe the process of mesh constrution in this section. Good mesh construction is a somewhat involved process that we will dedicate the next section of this tutorial to. Some of you may know how to construct good meshes already, e.g. from the SPDE-tutorial `www.r-inla.org/examples/tutorials/spde-tutorial`. The `interior` argument of `inla.mesh.2d` makes sure thet the triangles in the mesh respects the boundary of our barrier.

```
loc1 = matrix(c(0,0, 10,0, 0,10, 10,10), 4, 2, byrow = T)
# - This defines the extent of the mesh
# - In an application, if you want the mesh to depend on your data locations,
#   you may use those locations here
seg = inla.sp2segment(p)
# - Transforms a SpatialPolygon to an ''inla polygon''
mesh = inla.mesh.2d(loc=loc1, interior = seg, max.e = max.edge.length,
                    offset=1)
```

```
# - The INLA mesh constructor, used for any INLA-SPDE model
mesh = DT.mesh.addon.posTri(mesh)
# - A special tool needed for these models
```

Next, we will pick out which triangles in the mesh belongs to the barrier area. Then we define the `Omega`) object which is the object that the code for the new Barrier model needs. We also define the `Omega.SP` which is the spatial polygon needed for plotting the boundaries of the barrier area.

```
barrier = over(p, SpatialPoints(mesh$posTri), returnList=T)
# - checking which mesh triangles are inside the barrier area
barrier = unlist(barrier)
Omega = DT.Omega(list(barrier, 1:mesh$t), mesh)
Omega.SP = DT.polygon.omega(mesh, Omega)
# - creates polygons for the different areas: Barrier area and Normal area
```

At this point, it is crucial to check that our mesh, and the two areas (normal area and barrier area) are correct. We make sure to always use `Omega.SP` to plot the barrier from now on, since this is the same as what is used by the algorithm (which may not be true for our original polygon `p`). See Figure 1.

```
oldpar <- par(mar = c(0, 0, 0, 0))
plot(mesh, main="")
plot(Omega.SP[[1]], add=T, col='grey')
plot(Omega.SP[[2]], add=T, col='lightblue')
plot(mesh, add=T)
points(loc1)
```

We define how to plot spatial fields for this example. The input variable `field` must have the same length as there are mesh nodes in the mesh. We have also defined the part of space that is to be plotted, in any future plots, namely `xlim=ylim=c(2,8)`. (Since we only care about this region, the mesh extension is large enough. We discuss mesh extensions later.)

```
local.plot.field = function(field, ...){
  xlim = c(2, 8); ylim = xlim;
  proj = inla.mesh.projector(mesh, xlim = xlim, ylim = ylim, dims=c(300, 300))
  # - Can project from the mesh onto a 300 by 300 grid for plotting
  field.proj = inla.mesh.project(proj, field)
  # - Do the projection
  image.plot(list(x = proj$x, y=proj$y, z = field.proj),
             xlim = xlim, ylim = ylim, ...)
  # - Use image.plot to get nice colors and legend
}
print2(mesh$n)
## [1] 2613
# - This is the appropriate length of the field variable
```

Next, we compute the Finite Element matrices needed to solve the SPDE for the non-stationary Barrier model.

$$u(s) - \nabla \cdot \frac{r^2}{8} \nabla u(s) = r\sqrt{\frac{\pi}{2}}\sigma_u \mathcal{W}(s), \text{ for } s \in \Omega_n$$

$$u(s) - \nabla \cdot \frac{r_b^2}{8} \nabla u(s) = r_b\sqrt{\frac{\pi}{2}}\sigma_u \mathcal{W}(s), \text{ for } s \in \Omega_b,$$

To learn more about these equations, see the paper on Arxiv Bakka et al. (2016), and the appendix therein. Essentially, we are doing all the computations we can in advance, so that solving the system

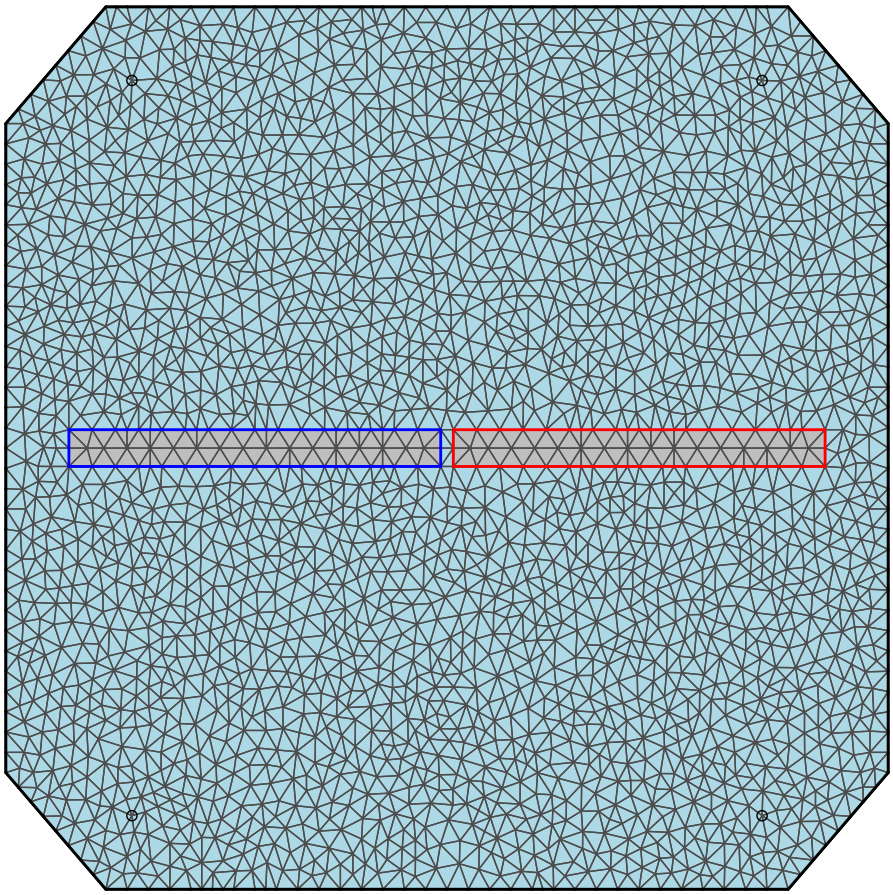

Figure 1: In this figure we see the entire mesh. The grey background is for the barrier area. The light blur background is for the normal area. The blue and red squares are the boundaries of the barrier area. The black circles are our initial locations (here, we use them only to determine the extent of the mesh).

of equations happens quickly at each step of simulation and inference. We will not comment on the internal workings of this function now, but it will be described in detail in a later section of this tutorial.

```
spde.DT = DT.FEMmatrices(mesh, Omega)
# - Set up the matrices for solving the SPDE
# - May be time consuming
```

2.2 Simulate data

In this subsection, we simulate the data that we will do inference on in the next subsection.

First we set the ranges, with a range 0.3 on the first region, i.e. the barrier, and a range of 3 in the second region, i.e. the normal region. The range is the distance at which correlation is essentially zero. Then we compute the precision matrix Q (inverse covariance matrix) for the spatial random field

$$u \sim \mathcal{N}(0, Q^{-1})$$

```
ranges = c(0.3, 3)
# - the first range is for the barrier area
# - - it is not sensitive to the exact value here, just make it ''small''
# - the second range is for the normal area
Q = DT.precision.new(spde.DT, ranges)
# - the precision matrix for fixed ranges
```

Now, we sample a spatial field using the precision matrix Q. This is done through a sparse Cholesky factorisation Rue and Held (2005). Then we plot that field, adding the barrier area in grey on top. The value of the field on the barrier is of no interest, as there can never be observations there.

```
u = inla.qsample(n=1, Q=Q, seed = set.inla.seed)
u = u[ ,1]
local.plot.field(u)
# - The true spatial field
plot(Omega.SP[[1]], add=T, col='grey')
```

```
# - The barrier
```

Now we simulate the spatial locations and the data. We never have observation locations in the barrier region (e.g. no measurements of fish on land), so the easiest way to sample locations is to sample everywhere, and then just delete those samples that end up in the barrier region. The projector matrix is the function taking a field-defined-on-the-mesh into a field-defined-on-the-data-locations.

```
num.try = 500
# - try to sample this number of data locations
# - locations sampled inside the barrier will be removed in a few lines
loc.data.before = matrix(runif(num.try*2, min=2, max=8), num.try, 2)
temp = SpatialPoints(loc.data.before)
ok.locations = !is.na(over(temp, Omega.SP[[2]]))
# - only allow locations that are not inside the Barrier area
loc.data = loc.data.before[ok.locations, ]
A.data = inla.spde.make.A(mesh, loc.data)
# - the projector matrix required for any spatial model
# - this matrix can transform the field-defined-on-the-mesh to the
#   field-defined-on-the-data-locations
c(dim(A.data), mesh$n, nrow(loc.data))
```

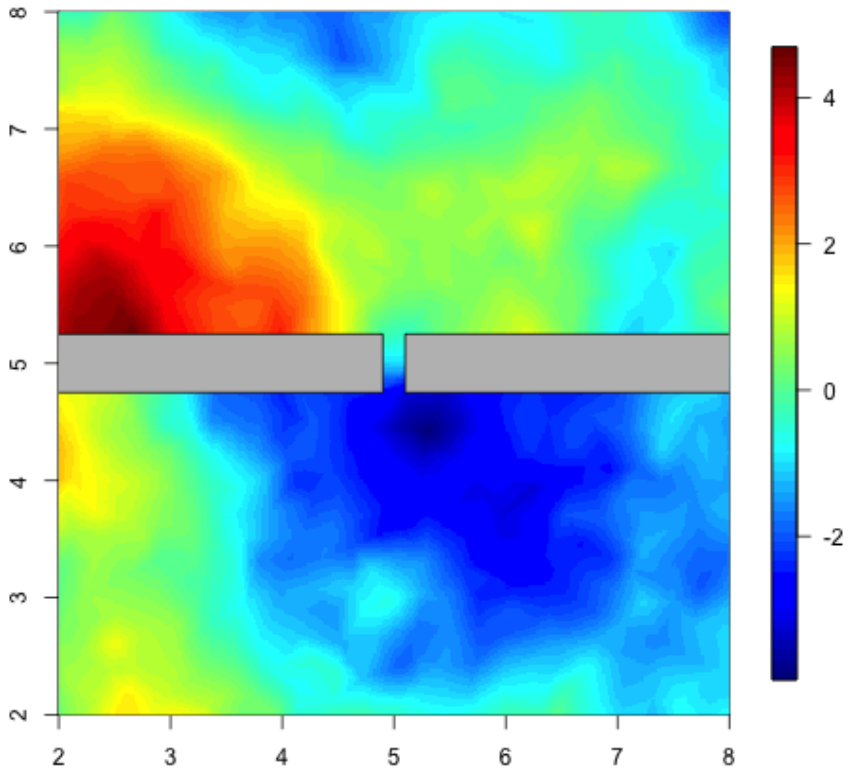

Figure 2: The true (simulated) spatial field

```
## [1]  469 2613 2613  469
# - shows that the dimensions are correct
u.data = A.data %*% u
# - project the field from the finite element representation to the data locations
df = data.frame(loc.data)
# - df is the dataframe used for modeling
names(df) = c('locx', 'locy')
sigma.u = 1
# - size of the random effect
# - feel free to change this value
sigma.epsilon = 0.2
# - size of the iid noise in the Gaussian likelihood
# - feel free to change this value
df$y = drop(sigma.u*u.data + sigma.epsilon*rnorm(nrow(df)))
# - sample observations with gaussian noise
```

We have now completed the simulation of the spatial dataset.

```
summary(df)
##       locx            locy             y
##  Min.   :2.016   Min.   :2.033   Min.   :-3.3661
##  1st Qu.:3.420   1st Qu.:3.663   1st Qu.:-1.3952
##  Median :5.038   Median :5.524   Median :-0.2316
##  Mean   :4.985   Mean   :5.146   Mean   :-0.1700
##  3rd Qu.:6.465   3rd Qu.:6.715   3rd Qu.: 0.6865
##  Max.   :7.993   Max.   :7.991   Max.   : 4.4706
```

2.3 Stationary INLA model

The model in this subsection not the new model, but the common spatial model in INLA. We add this for comparison between stationary and Barrier model. This means that the model in this subsection is not the model used for simulation (that model will come in the next subsection).

The stack will be used both for the stationary model and for the Barrier model. This is just a way of organising the different variables. We could have written this example without using the stack, but if you want to use this model for anything more advanced, you will need this stack functionality.

```
stk <- inla.stack(data=list(y=df$y), A=list(A.data, 1),
                  effects=list(s=1:mesh$n, intercept=rep(1, nrow(df))),
                  remove.unused = FALSE, tag='est')
# - this is the common stack used in INLA SPDE models
# - see the SPDE-tutorial
# - - http://www.r-inla.org/examples/tutorials/spde-tutorial
```

Next, we set up a standard `INLA` call for a stationary spatial effect. We use the PC prior for the gaussian σ_ϵ. We start the numerical optimiser with a good value in `control.mode`, to reduce the computational time (you may remove this line, but it will take longer). We know where to start the optimiser because we have run this model several times. In your case, you may run a subset of your data to find good starting values, especially for space-time models.

```
spde.model = inla.spde2.matern(mesh)
# - Set up the model component for the spatial SPDE model: Stationary Matern model
# - I assume you are somewhat familiar with this model
```

```
formula <- y ~ 0 + intercept + f(s, model=spde.model)
# - Remove the default intercept
# - - Having it in the stack instead improves the numerical accuracy of the
#     INLA algorithm
# - Fixed effects + random effects

res.stationary <- inla(formula, data=inla.stack.data(stk),
            control.predictor=list(A = inla.stack.A(stk)),
            family = 'gaussian',
            control.family = list(hyper = list(prec = list(
              prior = "pc.prec", fixed = FALSE, param = c(0.2,0.5)))),
            control.mode=list(restart=T, theta=c(4,-1.7,0.25)))
```

We look at the standard `INLA` summary.

```
summary2(res.stationary)
```

```
##
## Call:
## c("inla(formula = formula, family = \"gaussian\", data = inla.stack.data(stk),
## ", " control.predictor = list(A = inla.stack.A(stk)), control.family =
## list(hyper = list(prec = list(prior = \"pc.prec\", ", " fixed = FALSE, param =
## c(0.2, 0.5)))), control.mode = list(restart = T, ", " theta = c(4, -1.7,
## 0.25)))")
##
## Time used:
## Pre-processing Running inla Post-processing Total
## 1.2557 9.4774 0.1751 10.9082
##
## Fixed effects:
## mean sd 0.025quant 0.5quant 0.975quant mode kld
## intercept -0.1653 1.4749 -3.3205 -0.1608 2.9527 -0.1572 0
##
## Random effects:
## Name Model
## s SPDE2 model
##
## Model hyperparameters:
## mean sd 0.025quant 0.5quant
## Precision for the Gaussian observations 27.0180 2.7386 22.016 26.8889
## Theta1 for s -1.3634 0.0644 -1.487 -1.3646
## Theta2 for s -0.6589 0.2916 -1.286 -0.6338
## 0.975quant mode
## Precision for the Gaussian observations 32.7727 26.6415
## Theta1 for s -1.2339 -1.3684
## Theta2 for s -0.1518 -0.5554
##
## Expected number of effective parameters(std dev): 206.52(10.24)
## Number of equivalent replicates : 2.271
##
## Marginal log-Likelihood: -174.34
```

We plot the result of the stationary model in Figure 3.

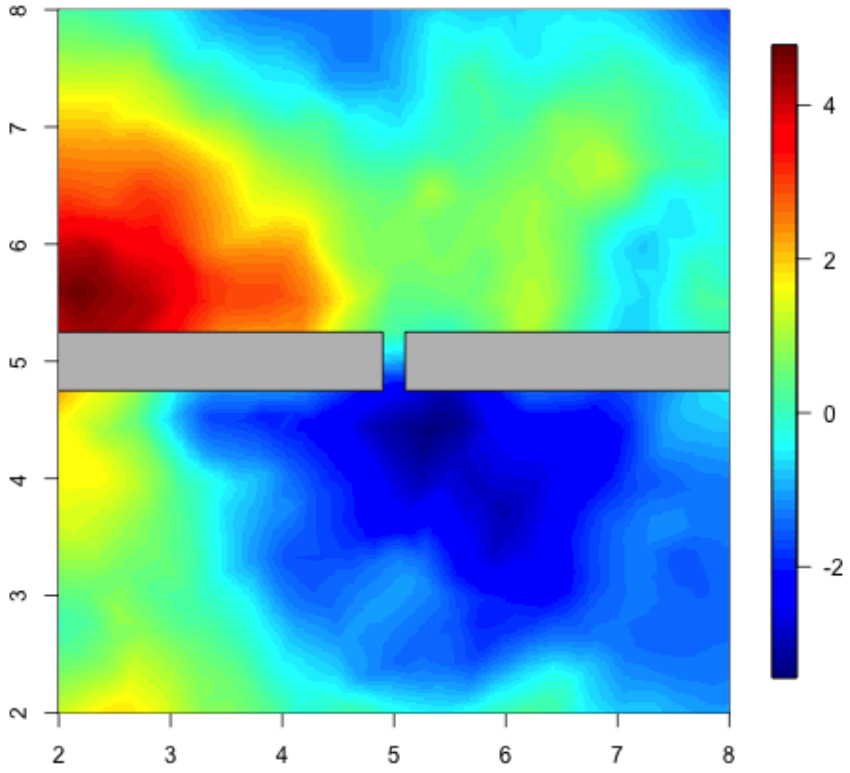

Figure 3: Posterior spatial estimate using the stationary model

```
local.plot.field(res.stationary$summary.random$s$mean)
# - plot the posterior spatial marginal means
# - we call this the spatial estimate, or the smoothed data
plot(Omega.SP[[1]], add=T, col='grey')
```

```
# - Posterior spatial estimate using the stationary model
```

2.4 Barrier model

This is the same model as we used for simulating the data. Except that we have now forgotten the range in the barrier area. This range is never known, so it is unreasonable to use it for inference. However, using any small value for the range in the barrier region is ok, it does not need to be the "true value". For more information about this, see appendix "Choosing barrier range" in Bakka et al. (2016). The Barrier model is a special case of the Different Terrains model, therefore the naming is "DT" in many variables and functions.

First we set up extra variables needed for my implementation of the new model. My implementation is based on `rgeneric` in `R-INLA`, see `inla.doc("rgeneric")`. This part of the code defines what range parameters belong to each area. Here we fix the barrier range to 0.05, and let the normal area range vary (by using `NA`). You can use other priors if you want, through the `prior.function` argument.

```
DT = DT.rgeneric.environment(spde.DT, prior.parameters=c(1, 1),
                             fixed.ranges = c(0.05, NA))
# - This function creates the environment/variables needed to run the internal
#   rgeneric engine inside the C-code in R-INLA
# - We fix the barrier range to a different value than we used for simulations
# - - Why? It does not matter, as long as it is 'small' the models are very
#     similar
# - - This shows that you do not need to know the true 'barrier range'!
# - The prior parameters are the lambdas in the exponential priors for standard
#   deviation and inverse-range

filename.rgeneric.environment = 'temp.rgeneric.init.RData'
save('DT', file = filename.rgeneric.environment)
# - save the new environment file
# - this is not an actual R environment, just a list-of-lists that is loaded
#   at rgeneric startup
```

```
barrier.model = inla.rgeneric.define(DT.rgeneric.dt.model,
                       n=dim(spde.DT$I)[1], ntheta = DT$ntheta,
                       R.init=filename.rgeneric.environment, debug=F)
# - This contains all the needed functions for the spatial model component
# - - Most important is the function computing the precision matrix Q from the
#     hyperparameters theta
# - These functions are called by the C-code in R-INLA when R-INLA is
#   performing inference
# - - The inla(...) call spawns a C algorithm running inference, which again
#     spawns an R algorithm running rgeneric
```

```
formula2 <- y ~ 0 + intercept + f(s, model=barrier.model)
# - The spatial model component is different from before
# - The rest of the model setup is the same! (as in the stationary case)
# - - e.g. the inla(...) call below is the same, only this formula is different
```

Finally, we are ready to run inference on the simulated dataset. Similarly to the stationary case, you may remove the `control.mode` input. That will only cause the model to run slightly slower. If you have trouble with inference in a different dataset, setting these values (the initial values for the algorithm) often solves the problem. Please note that these are the values for the internal parametrisation, so $\log(precision), \log(sigma), \log(range)$.

```
res.barrier = inla(formula2, data=inla.stack.data(stk),
                   control.predictor=list(A = inla.stack.A(stk)),
                   family = 'gaussian',
                   control.family = list(hyper = list(prec = list(
                         prior = "pc.prec", fixed = FALSE,
                         param = c(0.2,0.5)))),
                   control.mode=list(restart=T, theta=c(3.2, 0.4, 1.6)))
```

```
summary2(res.barrier)
```

```
##
## Call:
## c("inla(formula = formula2, family = \"gaussian\", data = inla.stack.data(stk),
## ", " control.predictor = list(A = inla.stack.A(stk)), control.family =
```

```
## list(hyper = list(prec = list(prior = \"pc.prec\", ", " fixed = FALSE, param =
## c(0.2, 0.5)))), control.mode = list(restart = T, ", " theta = c(3.2, 0.4,
## 1.6)))")
##
## Time used:
## Pre-processing Running inla Post-processing Total
## 0.6244 16.8094 0.1737 17.6075
##
## Fixed effects:
## mean sd 0.025quant 0.5quant 0.975quant mode kld
## intercept -0.2113 1.2287 -2.7749 -0.2109 2.3521 -0.2088 0
##
## Random effects:
## Name Model
## s RGeneric2
##
## Model hyperparameters:
## mean sd 0.025quant 0.5quant
## Precision for the Gaussian observations 24.864 2.3030 20.6507 24.7571
## Theta1 for s 0.535 0.2283 0.1258 0.5197
## Theta2 for s 1.766 0.2529 1.3129 1.7485
## 0.975quant mode
## Precision for the Gaussian observations 29.692 24.545
## Theta1 for s 1.018 0.473
## Theta2 for s 2.301 1.696
##
## Expected number of effective parameters(std dev): 158.00(10.56)
## Number of equivalent replicates : 2.968
##
## Marginal log-Likelihood: -119.61
# - You only need to write summary(res.barrier)
```

We plot the result of the Barrier model in Figure 4. Compare this plot to the simulation in Figure 2 and the stationary solution in Figure 3. Depending on the simulated field, these figures may look very much the same, or be very different. Our experience is that, even when the spatial means look quite different, there may be a big difference in model fit or prediction evaluations. Feel free to also plot the quantiles and sd of the two posterior fields. For a discussion of the results we refer you to the main paper Bakka et al. (2016).

```
local.plot.field(res.barrier$summary.random$s$mean)
# - plot the posterior spatial marginal means
# - we call this the spatial (smoothing) estimate
plot(Omega.SP[[1]], add=T, col='grey')
```

```
# - Posterior spatial estimate using the Barrier model
```

3 Comments to the simulation and inference example

We have additional important comments about the previous example. We chose to add these in a separate section to keep the previous section cleaner.

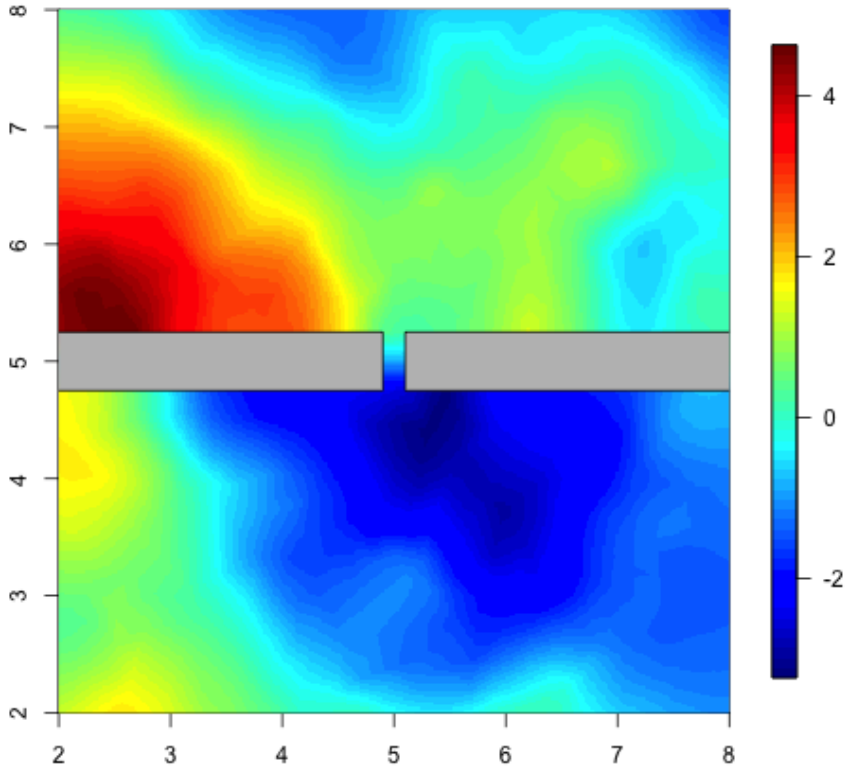

Figure 4: Posterior spatial estimate using the Barrier model

3.1 Mistakes to avoid

I will not discuss potential mistakes from the definition of the mesh or the barrier, as that follows in the next section. Make sure that you do not mix up what area is the barrier, and what is the normal. You do not have to write the barrier area first, but the entry you fix in `fixed.ranges` must be correct with respect to the ordering of the two areas. To be sure, please simulate from the prior model, according to your values in `fixed.ranges` (replace NAs with values) and check that it is sensible.

3.2 How to use this as a model component in other models

Here, I outline how to do things that are slightly different.

To add more covariates in the model, add them in the `inla.stack`. See the SPDE-tutorial for more information on this `www.r-inla.org/examples/tutorials/spde-tutorial`.

You can easily use other likelihoods, by changing the `family` argument of the `INLA`-call. See `r-inla.org` for a description of these. Please make sure you choose goo priors for any hyper-parameters in the likelihood (the choice we made in the code here is a good choice).

You can also add different random effects to the formula (see info about latent models on `r-inla.org`). When you use additional random effects, you must include them into the stack, similarly to how the intercept is here included in the stack. For examples of this, use the SPDE tutorial `www.r-inla.org/examples/tutorials/spde-tutorial`.

4 How to create a good mesh?

In this section we use an example to show how you should go about creating a good mesh. This is not only relevant for the Barrier model, but also for any stationary model, or any other non-stationary model. One reason for writing this is that I have not found any other resource that shows how to create good meshes, with sufficient detail and clarity. In this section we also discuss how to take care of the polygon needed for the barrier area (water in this case).

The assumption behind this example is that you have some kind of map. The following code takes a map and transforms it into a `SpatialPolygons` object. Instead of this, you may have a shapefile, and a polygon constructed from this shapefile.

```
library('maps'); library('maptools'); library('mapdata')
data("worldHiresMapEnv")
nor_coast_poly <- map("worldHires", "norway", fill=TRUE, col="transparent",
                      plot=FALSE, ylim=c(58,72))
# The following code is taken from example in ?map2SpatialPolygons
nor_coast_poly$names
```

```
##  [1] "Norway"                    "Norway:Tana"
##  [3] "Norway:Stjernoya"          "Norway:Karlsoy"
##  [5] "Norway:Seiland"            "Norway:Soroya"
##  [7] "Norway:Vega"               "Norway:Hareidlandet"
##  [9] "Norway:Froya"              "Norway:Andoya"
## [11] "Norway:Hitra"              "Norway:Gurskoya"
## [13] "Norway:Langoya"            "Norway:Arnoy"
## [15] "Norway:Vannoy"             "Norway:Mageroya"
## [17] "Norway:Hinnoya"            "Norway:Svalbard:Jan Mayen"
## [19] "Norway:Senja"              "Norway:Sunnhordland"
## [21] "Norway:Ringvassoy"         "Norway:Vestvagoy"
## [23] "Norway:Austvagoy"          "Norway:Nordhordland"
## [25] "Norway:Bremangerland"      "Norway:Moskenesoya"
```

```
## [27] "Norway:Kvaloy"              "Norway:Kvaloya"
## [29] "Norway:Smola"               "Norway:Karmoy"
## [31] "Norway:Vikna"
IDs <- sapply(strsplit(nor_coast_poly$names, ":"), function(x) x[1])
nor_coast_poly_sp <- map2SpatialPolygons(nor_coast_poly, IDs=IDs,
                                  proj4string=CRS("+proj=longlat +datum=WGS84"))
```

Now, we have the entire coastline as one big polygon. Assume we have a limited study area, namely $[4, 8] \times [60.6, 61.5]$. This is a small part of the Norwegian coast. We will cut down the polygon to a smaller polygon that is inside our study area. This is to avoid that the mesh is covering all of Norway. A smaller mesh gives faster inference.

```
loc3 = matrix(c(4,60.6, 8,60.6, 4,61.5, 8,61.5), 4, 2, byrow = T)
# - this is set to give the right extent of the mesh
# - you can use data locations instead
# - a bit extra offset will be added by the mesh function (later)
max.edge.length = 0.04
# - this is set to a length that is small compared to the width of the study area
square = bakka.square.polygon(xlim=range(loc3[ ,1]), ylim=range(loc3[ ,2]),
                              ret.SP = T)
# - This is the study area
# - - you are interested in fitting and/or predicting locations inside here
# - You only care about the water area, since land is a barrier
# - If loc3 is your data locations, you may want to make this square slightly bigger
# - - You should never report plots that are bigger than this square
square@proj4string=nor_coast_poly_sp@proj4string
# - This creates the same coordinate system for the square as we have for the map
poly3 = gIntersection(nor_coast_poly_sp, square)
# - The part of Norway inside the study area square
```

In Figure 5 we have plotted `poly3`.

```
plot(poly3, xlim=range(loc3[ ,1]), ylim=range(loc3[ ,2]), axes=T)
```

```
# - We see that poly3 is the land polygon
poly3.water = gDifference(square, poly3)
# - This is the polygon for water
# - This is what we need, since the mesh will be here!
```

We create the mesh (the triangulation of the study area) in three stesps. First we find the mesh of the water area, then we make a convex (not U-shaped) mesh, and then we create an outer mesh extension. The first mesh is a fine mesh for fitting and predictions at fine scale. The convex mesh is there to add small triangles on land near the coastline, and to ensure that there is no corners (corners are difficult for the finite element method). The third mesh is to create a "mesh extension". The mesh extension is needed since the boundary conditions of the mesh are wrong (they are deterministic and not stoachastic), but the solution is correct approximately 1 spatial range away from the boundary. This is a general problem that has little to do with these new models except that it explains why we cannot just stop the mesh at the coastline to deal with land being a barrier to the spatial effect.

The reason we can use the mesh locations and build a mesh iteratively is that the algorithm, known as "constrained refined Delaunay triangulation", is consistent, in that the same locations always give the same mesh.

```
mesh3 = inla.mesh.2d(boundary = poly3.water, max.e = max.edge.length)
# - A mesh over the water area, with the desired fidelity (max.e)
```

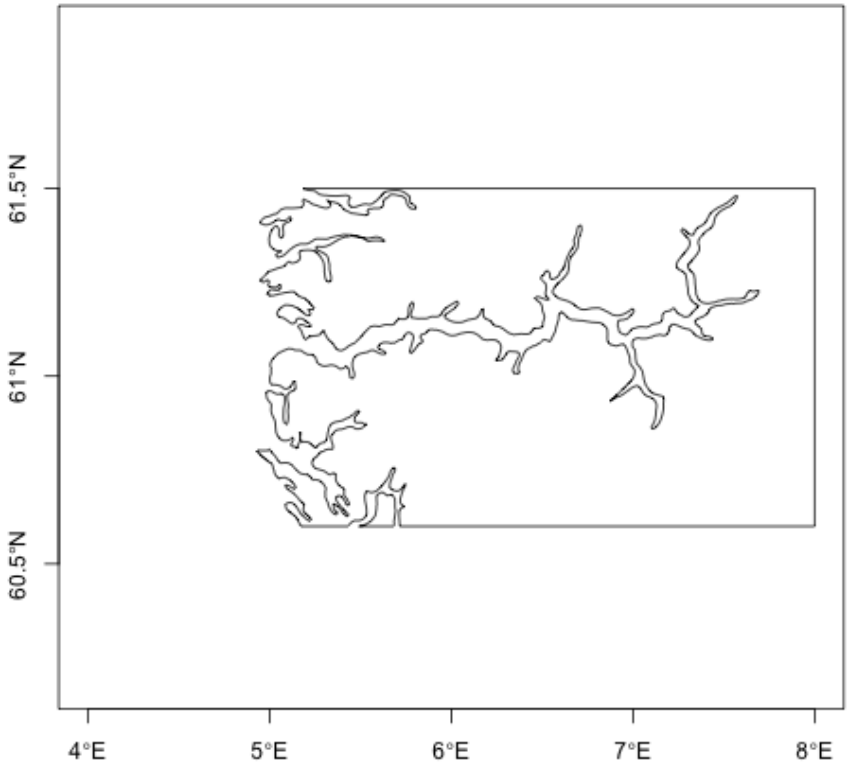

Figure 5: The land polygon for our study area

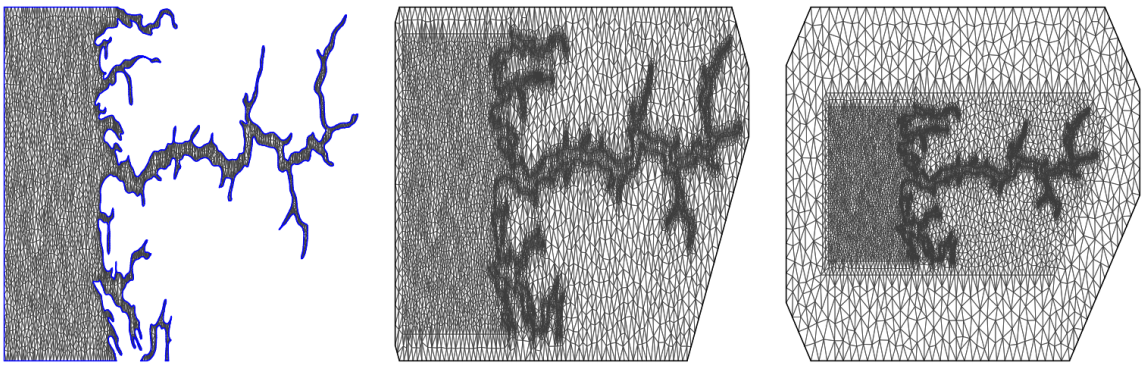

Figure 6: The three meshes

```
# - Many users fit (stationary) models to this mesh, but that is wrong!
# - - This is because you need some 'free space' between the study region and the
#     border of the mesh for the finite element approximation to be correct
oldpar <- par(mar = c(0, 0, 0, 0))
plot(mesh3, main="")
mesh3$n
## [1] 4432
mesh3 = inla.mesh.2d(loc=mesh3$loc, max.edge = max.edge.length*2,
                     offset=max.edge.length*2)
# - adds a convex hull with a small extension
# - here the mesh is coarser
# - this is done to have a good approximation on the border of the barrier
plot(mesh3, main="")
mesh3$n
## [1] 6897
mesh3 = inla.mesh.2d(loc=mesh3$loc, max.edge = max.edge.length*4, offset=0.5)
# - This adds an outer extension
# - This is needed since the spatial field is incorrect at the outer regions of
#   the mesh (inflated range and variance)
# - This outer extension will not be used for fitting or prediction, it is just
#   there to fix SPDE boundary approximation issues
plot(mesh3, main="")
mesh3$n
## [1] 7449
```

Next we have a special line of code that adds the central positions of the mesh triangles to the `mesh` obect.

```
mesh3 = DT.mesh.addon.posTri(mesh3)
```

We go back to the original polygon, and define the barrier as those triangles being on land there. What if you do not have the larger polygon? Then you can just draw some simple polygons to represent what you think the nearby coast is. For example, in this case, just continue the coastline vertically to the north and the south

```
## Define land triangles by using the big polygon
# - Previously we attempted to only use the polygon in our study area
# - But we extended the mesh!
```

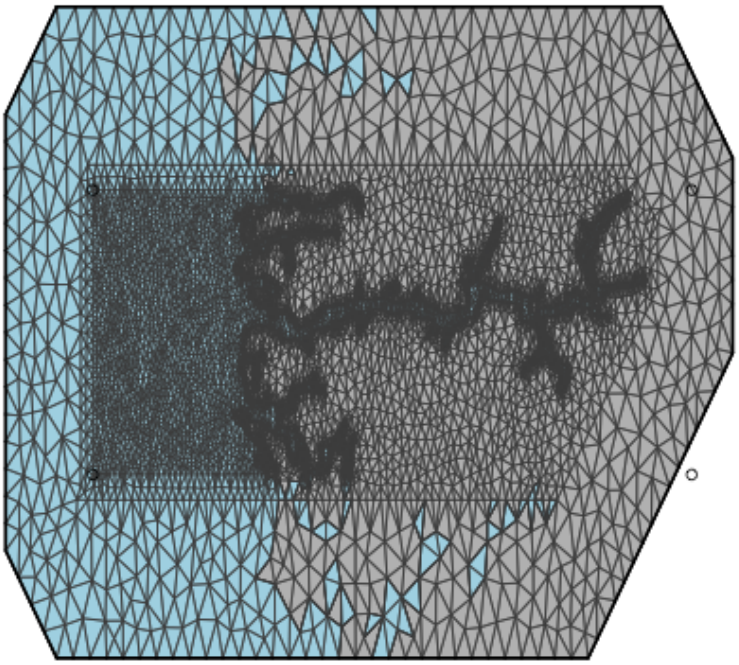

Figure 7: The correct way of defining the land (barrier) triangles in the Norway example.

```
points = SpatialPoints(mesh3$posTri)
points@proj4string = poly3@proj4string
barrier3 = over(nor_coast_poly_sp, points, returnList=T)
barrier3 = unlist(barrier3)
Omega3 = DT.Omega(list(barrier3, 1:mesh3$t), mesh3)
Omega.SP3 = DT.polygon.omega(mesh3, Omega3)
```

In Figure 7 we plot the mesh and barrier, and we see that the barrier looks good. The definition is coarse outside of our study area, but this is the best approach, as it is both computationally efficient and a good approximation. The reason it is good is because, per definition, we will not have data, or do any predictions, outside of the study area (the `square`).

```
## Visually check correctness
plot(mesh3, main="")
plot(Omega.SP3[[1]], add=T, col='grey')
plot(Omega.SP3[[2]], add=T, col='lightblue')
plot(mesh3, add=T)
points(loc3, col='black')
```

```
# - Check that the grey area represents the Barrier in a good way
# - - This is good!
```

5 Comments to the "how to create a good mesh" example

We have additional important comments about the previous example. We chose to add these in a separate section to keep the previous section cleaner. There are mainly two types of comments; "Mistakes to avoid" which is about mistakes I have done when using this or related code, and mistakes that are easy to do, and "How to do more" which is tips on how to use this code in other examples and more complicated models.

5.1 Mistakes to avoid: Forgetting the barrier in the extended mesh

```
## Attempt 1: Define which triangles that belong to land
# - This attempt is going to be incorrect, but it is important that you know
#   why, and how to check
points = SpatialPoints(mesh3$posTri)
points@proj4string = poly3@proj4string
barrier3 = over(poly3, points, returnList=T)
barrier3 = unlist(barrier3)
Omega3 = DT.Omega(list(barrier3, 1:mesh3$t), mesh3)
Omega.SP3 = DT.polygon.omega(mesh3, Omega3)
```

In Figure 8 we see the first attempt at defining which triangles belong to land and water. The problem here is that there are many blue triangles north and south of our study area. This means that the model thinks there is water there, but that is wrong.

```
## Attempt 1: Visually check correctness
plot(mesh3, main="")
plot(Omega.SP3[[1]], add=T, col='grey')
plot(Omega.SP3[[2]], add=T, col='lightblue')
plot(mesh3, add=T)
points(loc3, col='black')
```

```
# - Check that the grey area represents the Barrier in a good way
# - - This is not good enough!
```

Compare Figure 7 to Figure 8.

5.2 How small does `max.edge.length` need to be?

One of the big advantages with the finite element representation, i.e. the mesh, is the the spatial field is approximated not only at the mesh nodes, but at all locations inside the study area. This means that the `max.edge.length` can be much larger than the size of any grid cell in a gridded approach. For a stationary field, the rule is that `max.edge.length` should be at least 1/3 of the spatial range. You do not know the spatial range until after you have completed the inference, however. When you have a coastline (in a stationary, barrier, or other non-stationary model), the size of the triangles must be small enough that there is a good resolution of the coastline. This is taken care of directly, by having a suitable precise polygon of the coast, and you do not need to reduce `max.edge.length` to achieve this.

5.3 Mistakes to avoid: Inappropriate polygons

Make sure the coastline is precise enough, but not "too wiggly". If your coastline is too coarse, or is missing islands, you must use a different source for constructing the polygon. Note that the example code is missing the islands, as that is not part of the chosen map package! This error should be fixed before the method is applied to a real dataset in that study region! I recommend using google maps

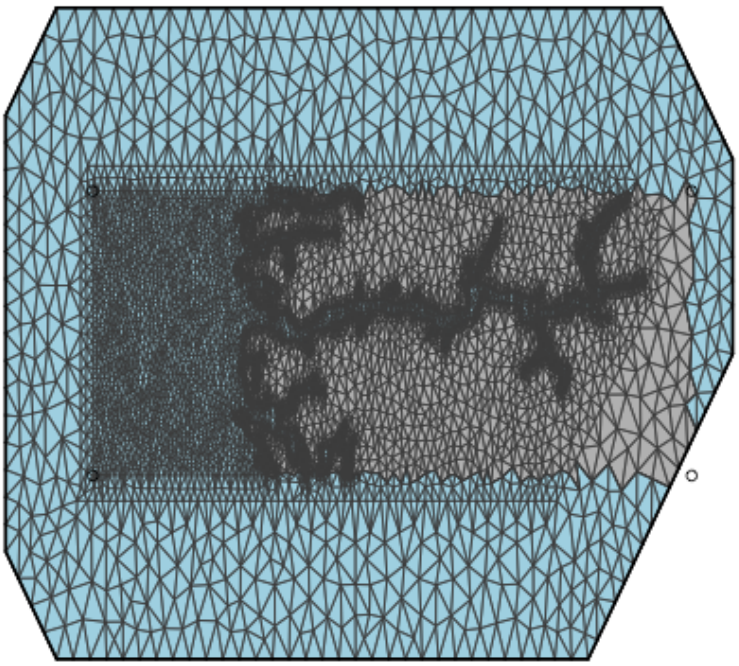

Figure 8: The first attempt at defining the land (barrier) triangles in the Norway example. This attempt is not successful, because many triangles that should have been land has been marked as water.

and comparing the satelite image to your polygon. In our case, look up a map over Sognefjorden (you will see several big islands).

If your polygon is very very fine, you may be creating a much finer mesh near the coast than you actually need, resulting in longer computational times, and, in extreme cases, in numerical errors. If you have a "too wiggly" polygon, you can use `inla.simplify.curve`. Alternatively, or in addition, you can use the `cutoff` option in `inla.mesh.2d`. A generally recommended value of `cutoff` is 1/5 or 1/10 of `max.edge.length`.

5.4 How to do more: Rasters

As long as you can find a suitable polygon you can use the code in the example. One way to create a polygon is from a raster, and this is what we did in the main paper Bakka et al. (2016). Detailed code on how to is out of the scope of this tutorial, but we want to discuss on the general approach to do this.

If you use a raster, you can use `inla.nonconvex.hull`, and a lot of trial and error. This function does not give a `SpatialPolygons` object, but an `INLA`-specific polygons object that you can use in the `INLA`-call.

To check whether your solutions are correct, plot your mesh and barrier as in the code "Visually check correctness" above. The main difficult is making the polygon coarse, so that the number of nodes in the mesh is limited. The raster grid size is typically way too small for the mesh to be defined on this scale.

6 The implementation of the Different Terrains model

In this section we discuss the implementation, as found in the function file, and the model itself.

Coming in 2016...

7 The general Different Terrains model

Coming in 2017...

References

Bakka, H., Vanhatalo, J., Illian, J., Simpson, D., and Rue, H. (2016). Accounting for physical barriers in species distribution modeling with non-stationary spatial random effects. arXiv preprint arXiv:1608.03787, Norwegian University of Science and Technology, Trondheim, Norway.

Rue, H. and Held, L. (2005). *Gaussian Markov Random Fields: Theory and Applications*, volume 104 of *Monographs on Statistics and Applied Probability*. Chapman & Hall, London.