Almost Painless Nonstationary Geostatistical Modeling using R package Bidart.

1 Existential questions.

1.1 Who, Where, When ?

This vignette is an illustration of *Nonstationary Spatial Process Models with Spatially Varying Covariance Kernels*, a paper by XXXX, XXXX, and XXXX.

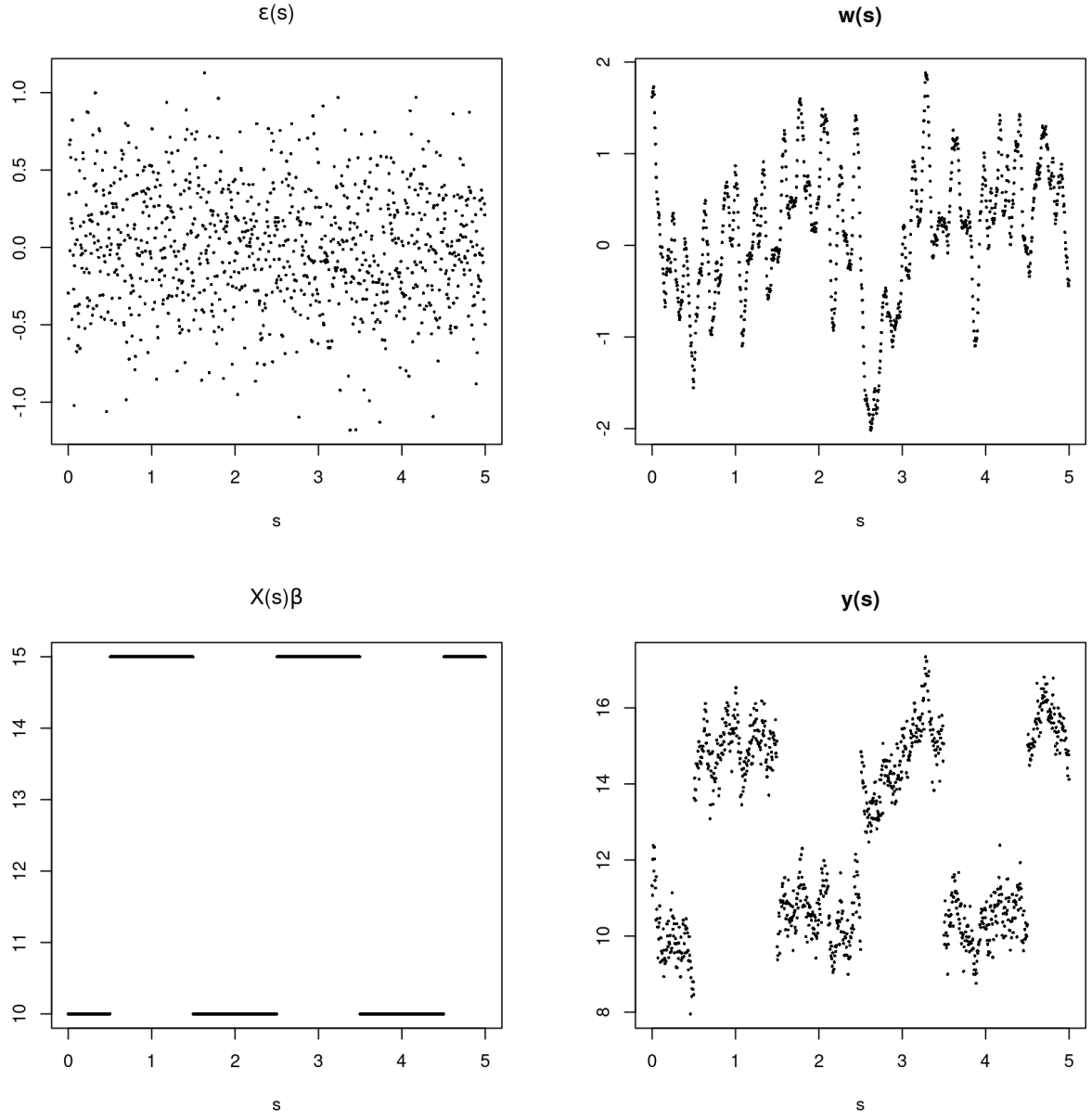
1.2 Whom, Why ?

This vignette **describes, explains, and illustrates** our nonstationary model without gooing too deep into the theoretical nitty-gritty. It gives **hands-on methods to use the model**, as well as **intuitions of how the model works**. It is aimed at **practitioners** brave enough to take a chance with our model.

1.3 What ?

Imagine that you have an interesting **spatially indexed data set**, such as **lead contamination**. You also have some nice **covariates**, such as **the proximity with a road**. You may want to use a **linear model**, but you suspect that there is a **spatial coherence in your errors**. You might want to know about this coherent error to make **better predictions**, but also because not accounting for it might mess the **confidence** / **credibility intervals** of your model.

So, you go for a **geostatistical model**, which pursues a **decomposition of the data** as: \[y(s) = w(s) + x(s)\beta + \epsilon(s)\] \(y(\cdot)\) being the interest variable, \(w(\cdot)\) being a Gaussian Process (GP) with some **spatial coherence**, \(\epsilon(\cdot)\) being a **Gaussian noise**, \(x(\cdot)\) being **independent variables**, \(\beta\) being **regression coefficients**, and eventually \(s\) being a **set of spatial coordinates** where the observations of \(y(\cdot)\) and \(x(\cdot)\) are done. The four panels below illustrate the decomposition of the data.



If you need to read more about what a geostatistical model is, have a look at this (https://becarioprecario.bitbucket.io/inla-gitbook/ch-spatial.html) but thread carefully, you may just want and need to start with a simpler model than those presented in this vignette and come back once you are more experienced.

The **process model** describes the behavior of the latent **Gaussian Process**, who captures the **underlying coherence of the data**. This Gaussian Process is **random**, like the error noise, but it is **spatially coherent**, which is not the case of the noise. In our case, the coherence is parametrized through the **covariance** between two spatial sites. The closer two sites, the higher the covariance. Correlation equal to 1 is reached when the two sites are the same. The nonstationary correlation model is taken from Paciorek's work (https://proceedings.neurips.cc/paper/2003/file/326a8c055c0d04f5b06544665d8bb3ea-Paper.pdf), and allows for **locally varying range and anisotropy**. We use a **Matérn model** for the correlation, with the same parametrization as GpGp (https://cran.r-project.org/web/packages/GpGp/GpGp.pdf). Correlation becomes a covariance when multiplied by standard deviations. We also allow a **locally varying marginal variance**. However, it is not recommended to mix nonstationary range and variance.

For computational reasons, we are using the **Nearest Neighbor Gaussian Process** (NNGP) framework, a member of the family of **Vecchia's approximations** to approximate the Gaussian Process covariance. You can look into the NNGP original paper (https://arxiv.org/pdf/1406.7343.pdf)

or an interesting comparison with other Vecchia's approximations (https://arxiv.org/abs/1708.06302), or just a nice online tutorial (https://andrewcharlesjones.github.io/journal/nngp.html).

Let's go back to the storytelling. You also suspect that the proximity with a road does not only raise the general level of lead in the ground, but also **makes the error term fuzzier, more unpredictable**. You would like your model to account for this. Well, it seems that you need a **nonstationary geostatistical model**, which means that the model will behave differently following the place and/or some variables.

1.4 How ?

You need a machine with a decent amount of RAM (16 Gb is good, 32 Gb is better), and a good CPU. You can also use a SLURM cluster. It is not necessary but very useful to install OpenBLAS (https://www.openblas.net/), which accelerates some matrix operations. Then, go the NonstatNNGP github page (https://github.com/SebastienCoube/Nonstat-NNGP), download the compressed package Bidart_1.0.tar.gz, and install it using *install.packages("Bidart_1.0.tar.gz", dependencies = TRUE)* in the Terminal of Rstudio.

2 Nonstationary Processes

This section gives some **intuitions** about the kind of **nonstationarity** used in the model. It starts with two types of nonstationarity that affect the latent field: the nonstationary range and the nonstationary latent field variance. Right after those two sections comes a warning against hubris in nonstationarity. Later we say a word about smoothness of the latent process, which is critical even though it is not properly nonstationary, and we end with the very important nonstationary noise variance.

2.1 The Russian Doll range model.

This subsection treats how the possibly **nonstationary correlation** is modeled. This is the biggest lump. Correlation is specified through the **range parameter(s)**, even though a nonstarionary smoothness is possible. Our range model is **organized like Russian dolls**. The simplest model is the **stationary** model. This model is encompassed into the **locally isotropic** model, *isotropic* meaning that the correlation is the same in all directions. This model behaves like a stationary isotropic model if you zoom up a small region, but its behavior changes if you take a bigger picture. This model is itself embedded into the **locally anisotropic** model, that is a model where not only the process correlation range is susceptible to change, but also where the **correlation changes with the orientation**.

2.1.1 The logarithmic parametrization of the range.

This subsection tells how the range is parametrized and gives some explanations about matrix logarithm. Intuitions about how those parametrizations impact the model come in the next sections.

In both case, a **logarithmic parametrization** (or matrix-logarithmic parametrization in the case of range ellipses) is used: \[log(\alpha(s)) = x_\alpha(s)\beta_\alpha\] in the scalar case, \(\alpha\) being the range parameter. This is the middle doll. We use \[vech(logM(A(s))) = x_A(s)\beta_A\] in the matrix case, \(A\) being the (matrix) range parameter, \(vech\) being the half-vectorization (https://en.wikipedia.org/wiki/Vectorization_(mathematics)#Half-vectorization) and \(logM\) being the matrix logarithm (https://en.wikipedia.org/wiki/Logarithm_of_a_matrix). Here, \(\beta_A\) is a **matrix with** \(3\) columns. This is the left-hand doll. Note that the range parameters \(\alpha(s)\) or \(A(s)\) are **local**, they depend on the **spatial site** \(**s**\). Therefore, they enforce a **local behavior of the Gaussian process** w(s).

We chose this parametrization because **spatial ranges are essentially sizes**, making the logarithm a relevant tool (https://en.wikipedia.org/wiki/Positive_real_numbers#Ratio_scale). The matrix logarithm, in turn, generalizes the properties of scalar logarithm to the matrix-valued anisotropic range parameters.

Note that \(A(s)\) is parametrized using **matrix logarithm**. What does it imply ? A symmetric matrix \(A\) of size \(2\times 2\) has an **eigendecomposition** (https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix) of the shape \[A = U~diag(\lambda_1, \lambda_2) U^T, \text{ with } U^T U = U U^T = diag(1,1). \] The **matrix exponential** (https://en.wikipedia.org/wiki/Matrix_exponential) of \(A\) is just \[exp(A) = U~diag(exp(\lambda_1), exp(\lambda_2)) U^T. \] Note that \(exp(\lambda_1)\) and \(exp(\lambda_2)\) are greater than \(0\), making \(exp(A)\) **positive-definite** (https://en.wikipedia.org/wiki/Definite_matrix). The matrix exponential **maps the symmetric matrices to the positive-definite matrices**, just like the scalar exponential maps the real numbers to the positive nubers. Positive-definite matrices define **ellipses** (https://en.wikipedia.org/wiki/Ellipse) on the plane using the modified circle equation \((x, y)A^{-1}(x, y)=1\), \(x\) and \(y\) being the two coordinates on the plane (The usual circle equation is \(x^2 + y^2 = 1\), that is \(A = diag(1,1)\)).
Those **ellipses** are later used in the correlation function: the correlation strength will look like an ellipse instead of a circle.

The symmetric matrices are a **vector space** (https://en.wikipedia.org/wiki/Vector_space). If you multiply a symmetric matrix by a number, it is still symmetric. If you add two symmetric matrices, the result is a symmetric matrix. This vector space admits (**orthonormal bases**)(https://en.wikipedia.org/wiki/Orthonormal_basis (https://en.wikipedia.org/wiki/Orthonormal_basis)), which means that you can

express any symmetric matrix as a sum of 3 well-chosen symmetric matrices. Note, however, that the positive-definite matrix are not a vector space but a cone (https://en.wikipedia.org/wiki/Cone). There is no basis allowing to reconstruct a positive-definite matrix with a few elements.

Let's recap. We want ellipses. To get ellipses, we need positive-definite matrices. To get positive-definite matrices, we need symmetric matrices. To get any symmetric matrix, we only need to sum 3 well-chosen symmetric matrices.

The three matrices we chose are:

$$e1 = \left(\begin{array}{cc} 1/\sqrt(2)&0\\ 0&1/\sqrt(2)\\ \end{array}\right), ~ e2 = \left(\begin{array}{cc} 1/\sqrt(2)&0\\ 0&-1/\sqrt(2)\\ \end{array}\right),~ e3 = \left(\begin{array}{cc} 0&1/\sqrt(2)\\ 1/\sqrt(2)&0\\ \end{array}\right).$$

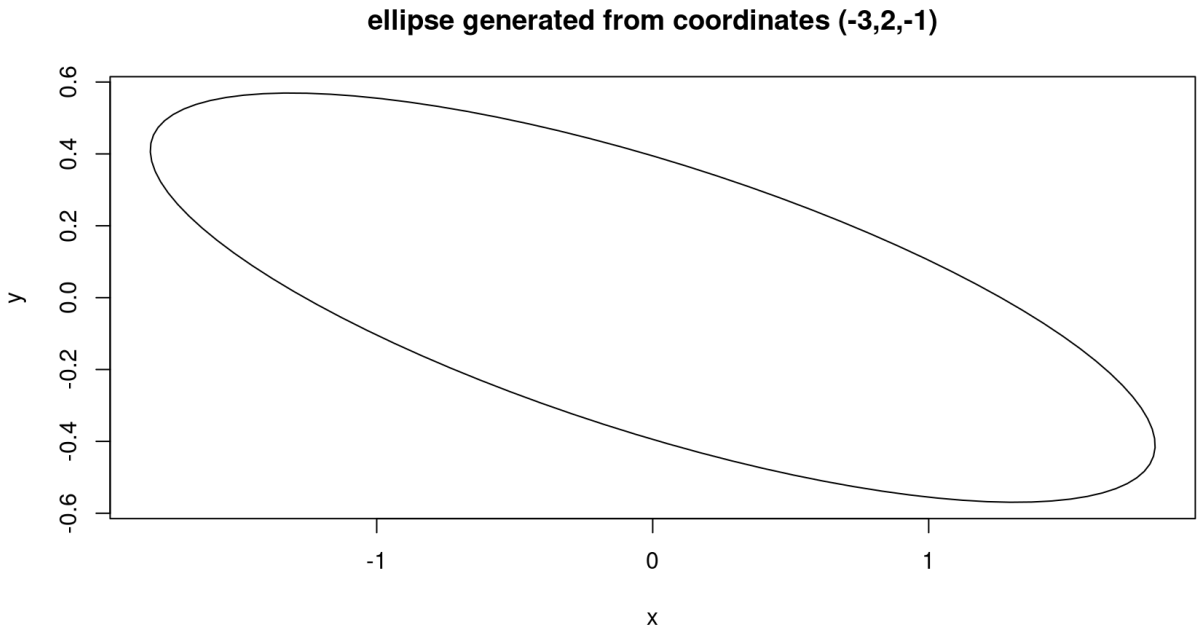
You might ask: "Why this basis and not any other valid orthonormal basis ?". Here is the reason why. Let's take generate an ellipse from our basis. First, we code the basis in R.

```
e1 = diag(c(1/sqrt(2), 1/sqrt(2)))
e2 = diag(c(1/sqrt(2), -1/sqrt(2)))
e3 = matrix(c(0, 1/sqrt(2), 1/sqrt(2), 0), 2)

print(list(e1 = e1, e2 = e2, e3 = e3))
```

```
## $e1
##           [,1]      [,2]
## [1,] 0.7071068 0.0000000
## [2,] 0.0000000 0.7071068
##
## $e2
##           [,1]       [,2]
## [1,] 0.7071068  0.0000000
## [2,] 0.0000000 -0.7071068
##
## $e3
##           [,1]      [,2]
## [1,] 0.0000000 0.7071068
## [2,] 0.7071068 0.0000000
```

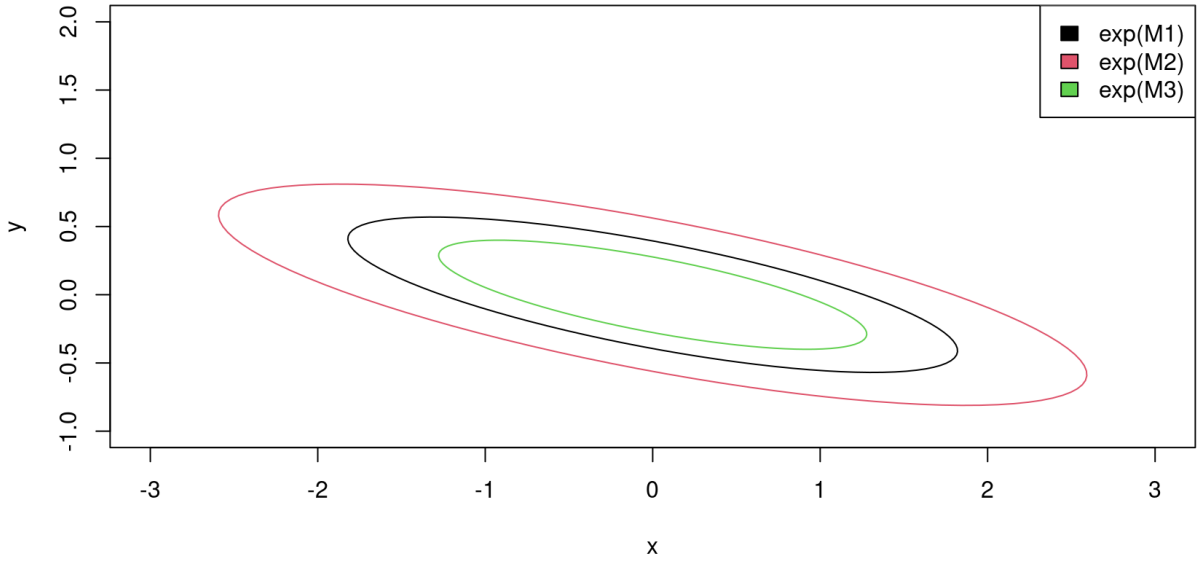
Then, we plot the ellipse associated with a symmetric matrix built using the basis.

```
M1 = -3*e1 + 2*e2 -1*e3 # symmetric matrix
exp_M1 = expm::expm(M1) # exponential
ell = ellipse::ellipse(exp_M1) # ellipse
plot(ell, type = 'l', main = "ellipse generated from coordinates (-3,2,-1)")
```



Good. Now, let's **change only the first coefficient** of the basis, and leave the other alone.

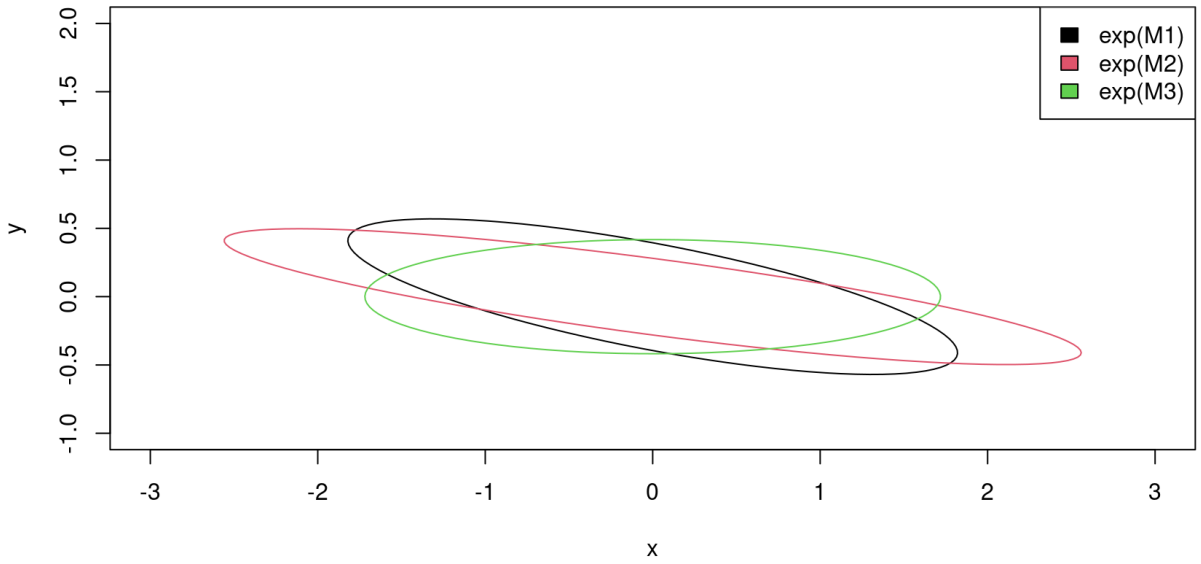
```
M1 = -3*e1 + 2*e2 -1*e3
M2 = M1 +1*e1
M3 = M1 -1*e1
plot(ellipse::ellipse( expm::expm(M1)), type = 'l', xlim = c(-3, 3), ylim = c
(-1, 2))
lines(ellipse::ellipse(expm::expm(M2)), type = 'l', col=2)
lines(ellipse::ellipse(expm::expm(M3)), type = 'l', col=3)
legend("topright" , legend = c("exp(M1)", "exp(M2)", "exp(M3)"), fill = seq
(3))
```



We see that **the ellipse inflates or deflates but does not change in shape, this is an homothety**. If we add more of the first element of the basis to the symmetric matrix, then the range will augment everywhere. If we remove some of it, the range will decrease everywhere. But **the direction and shape of the range will not change** !

Now, we change the **two other coefficients**:

```
M1 = -3*e1 + 2*e2 -1*e3
M2 = M1 +1*e2
M3 = M1 +1*e3
plot(ellipse::ellipse( expm::expm(M1)), type = 'l', xlim = c(-3, 3), ylim = c
(-1, 2))
lines(ellipse::ellipse(expm::expm(M2)), type = 'l', col=2)
lines(ellipse::ellipse(expm::expm(M3)), type = 'l', col=3)
legend("topright" , legend = c("exp(M1)", "exp(M2)", "exp(M3)"), fill = seq
(3))
```



We can see that the **directions and shapes of the ellipses changes**. However, even if it is difficult to see, their **area does not change**. This means that the other two parameters will **not change the overall correlation strength**.

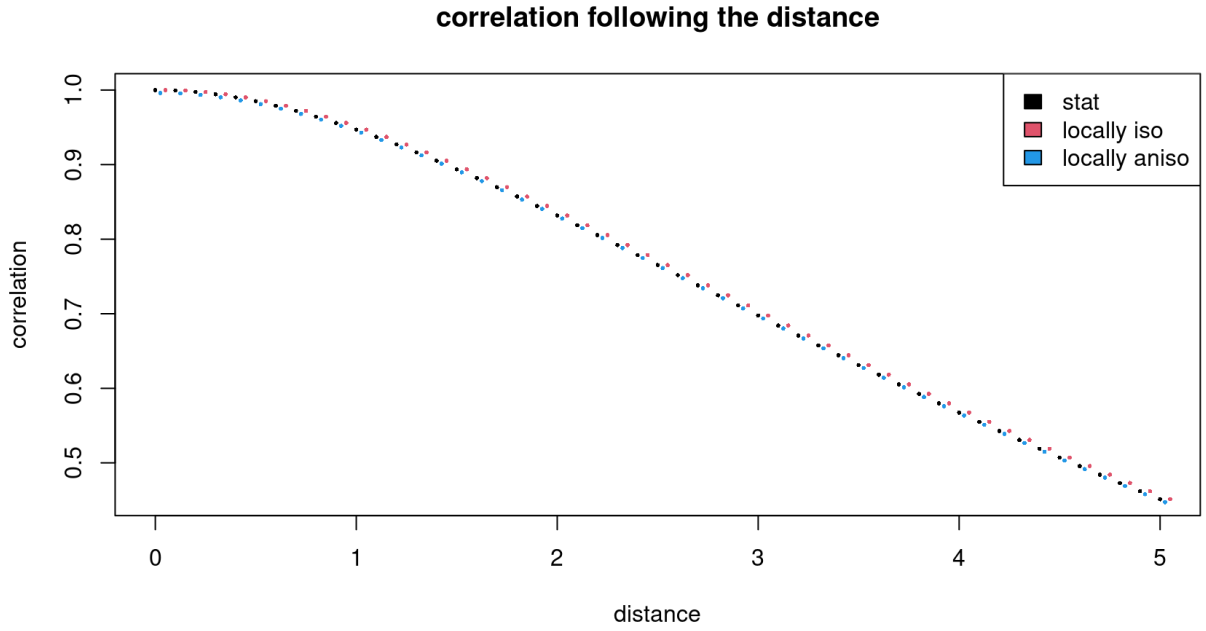
As a conclusion, we need only **three components** to specify **any range ellipse**. The first component controls the correlation **range**, while the other two control the correlation **shape**. If all the components are constant, we get a **stationary** correlation model. If only the first component is allowed to move and the two other are set to \(0\), we get a **nonstationary, locally isotropic** model. If everybody is allowed to move, we have a **nonstationary, locally anisotropic** model.

2.1.2 The stationary model.

To see that, let's represent the correlation of the Gaussian Process on an axis (so much for anisotropy…). We create 3 correlations:

- the **stationary** Matérn correlation given by GpGp, with range parameter equal to the *scalar* \(exp(1)\).
- the **nonstationary, locally isotropic** Matérn correlation where the range parameter is equal to the *matrix* \((1)\) and the range covariates are equal to a one-column matrix filled with ones (just an intercept).
- the **nonstationary, locally anisotropic** Matérn correlation where the range parameter is equal to the *matrix* \((1, 0, 0)\) and the range covariates are also just an intercept.

```
# locations varying between 0 and 5
locs = cbind(seq(0, 5, .1), 0)
# covariate for the range
X_range = matrix(1, nrow(locs))
# regression coefficients for the range
beta_iso = matrix(1)
beta_aniso = matrix(c(1,0,0), 1)
# NNGP setup
NNarray = GpGp::find_ordered_nn(locs, 20)
# locally isotropic covariance
covmat_iso =
  # computing coefficients
  Bidart::compute_sparse_chol(
    range_beta = beta_iso,
    NNarray = NNarray,
    locs = locs,
    range_X = X_range)
  # putting coefficients in precision Cholesly
covmat_iso = Matrix::sparseMatrix(
    x = covmat_iso[[1]][!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)]
    ) %>%
  # converting to covariance
  Matrix::solve() %>%
  Matrix::tcrossprod()
# locally anisotropic covariance
covmat_aniso =
  # computing coefficients
  Bidart::compute_sparse_chol(
    range_beta = beta_aniso,
    NNarray = NNarray,
    locs = locs,
    range_X = X_range,
    anisotropic = T # indicating that the covariance is anisotropic
    )
  # putting coefficients in precision Cholesly
covmat_aniso = Matrix::sparseMatrix(
    x = covmat_aniso[[1]][!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)]
    ) %>%
  # converting to covariance
  Matrix::solve() %>%
  Matrix::tcrossprod()
# stationary covariance
stat_cov = GpGp::matern15_isotropic(c(1,exp(1),0), locs)
# plotting covariance
plot(locs[,1], stat_cov[,1],pch = 16, cex = .4, xlab = "distance", ylab = "co
rrelation", main = "correlation following the distance")
points(locs[,1]+.05, covmat_aniso[,1],pch = 16, cex = .4, col=2)
points(locs[,1]+.025, covmat_aniso[,1] -0.004,pch = 16, cex = .4, col=4)
legend("topright", legend = c("stat", "locally iso", "locally aniso"), fill =
c(1,2,4))
```



We can see that the correlations are the same along the line. That is because we are using very simple nonstationary models every time, where the covariates for the range are only an intercept. The models are in those cases **practically stationary**. It means that **the stationary model is just a state of the nonstationary models, like Russian dolls**.

2.1.3 The locally isotropic model.

Let's introduce some nonstarionarity. We start by making the model **locally isotropic**. To do that, we need more than just an intercept. We will add the first spatial coordinate as a covariate for the range.

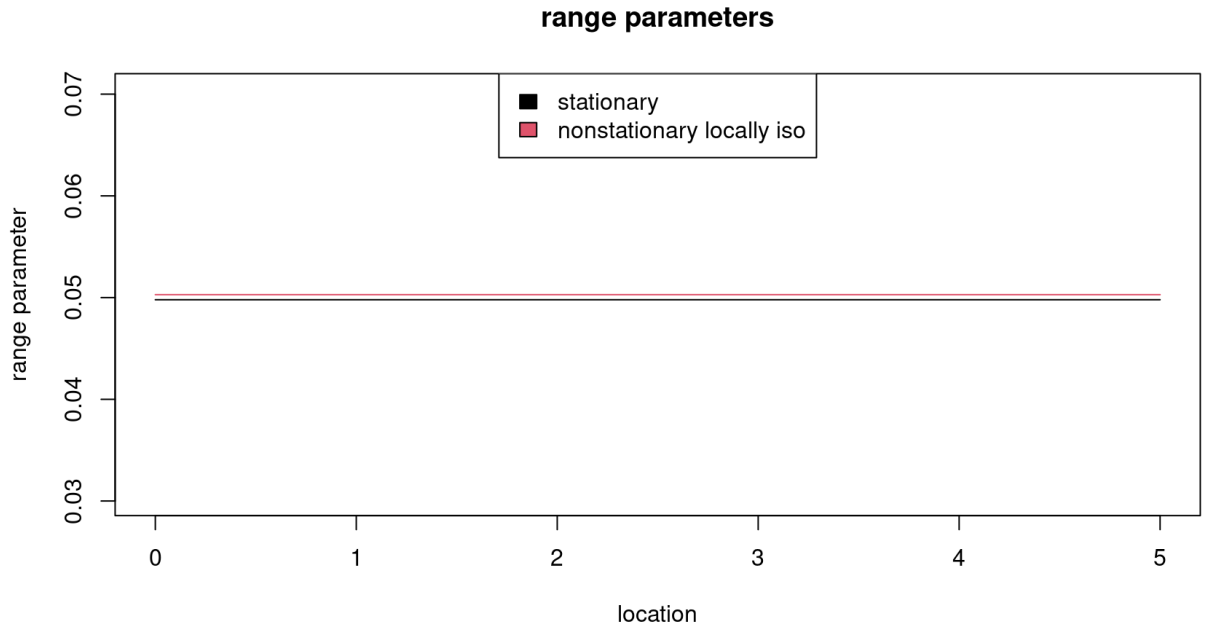
```
# more locations varying between 0 and 5
locs = cbind(seq(0, 5, .01), 0)
# covariate for the range
X_range = cbind(matrix(1, nrow(locs)), locs[,1])
```

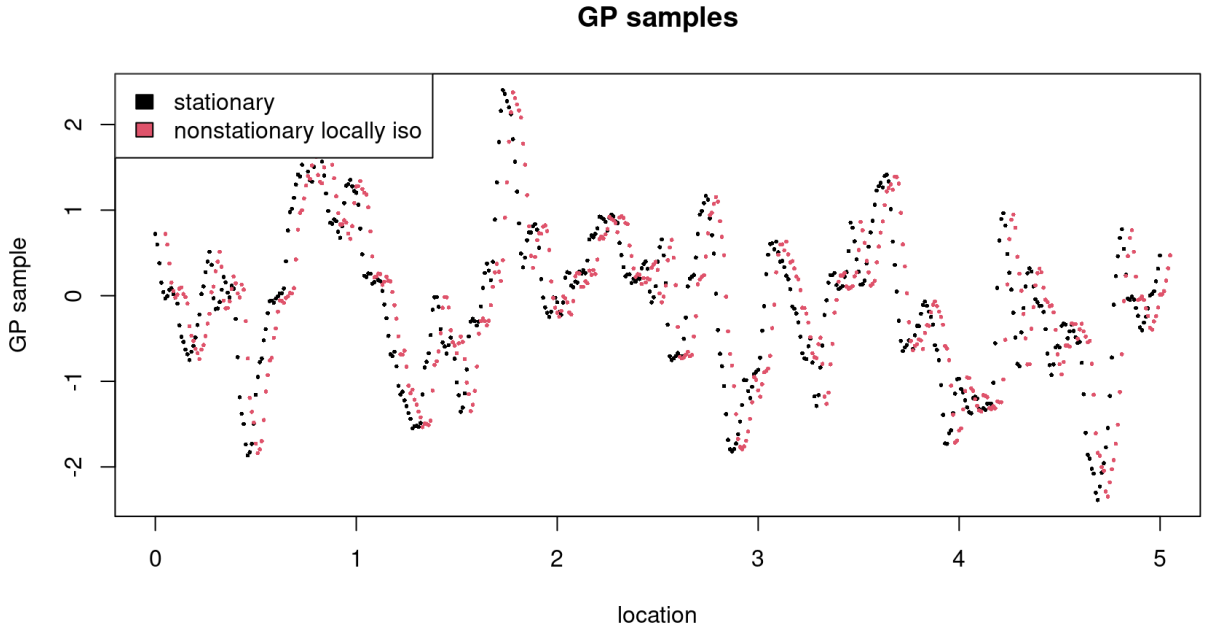
Now, let's **compare samples** of the nonstationary and the stationary model. The following function compares two samples of a stationary model and a locally isotropic model.

```
compare_stat_iso = function(
    range_stat, # the stationary range fed to GpGp's function
    beta_range_iso # the nonstationary range parameter
    ){
  # NNGP setup
  NNarray = GpGp::find_ordered_nn(locs, 20)
  # locally isotropic covariance
  nonstat_vecchia_chol =
    # computing coefficients
    Bidart::compute_sparse_chol(
      range_beta = beta_range_iso,
      NNarray = NNarray,
      locs = locs,
      range_X = X_range)
  # putting coefficients in precision Cholesly
  nonstat_vecchia_chol = Matrix::sparseMatrix(
    x = nonstat_vecchia_chol[[1]][!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)]
  )
  # stationary covariance
  stat_cov_chol = GpGp::matern15_isotropic(c(1,range_stat,0), locs) %>% chol
  # plotting log range
  plot(locs[order(locs[,1]),1], exp(X_range %*% beta_range_iso + .01)[order(l
ocs[,1])], xlab = "location", ylab = "range parameter", type = "l", col=2, ma
in = "range parameters")
  lines(locs[order(locs[,1]),1], rep(range_stat, nrow(locs)), col=1)
  legend("top", legend = c("stationary", "nonstationary locally iso"), fill =
c(1,2))
  # getting Gaussian Process samples
  seed_vector = rnorm(nrow(locs)) # shared white noise
  stat_samples = t(stat_cov_chol) %*% seed_vector# using covariance Cholesky
method
  nonstat_samples = Matrix::solve(nonstat_vecchia_chol, seed_vector)# using p
recision Cholesky method
  plot(locs[,1], stat_samples,pch = 16, cex = .4, xlab = "location", ylab = "
GP sample", main = "GP samples")
  points(locs[,1]+.05, nonstat_samples,pch = 16, cex = .4, col=2)
  legend("topleft", legend = c("stationary", "nonstationary locally iso"), fi
ll = c(1,2))
}
```

Let’s make a first try.

```
compare_stat_iso(range_stat = exp(-3), beta_range_iso = matrix(c(-3,0)))
```

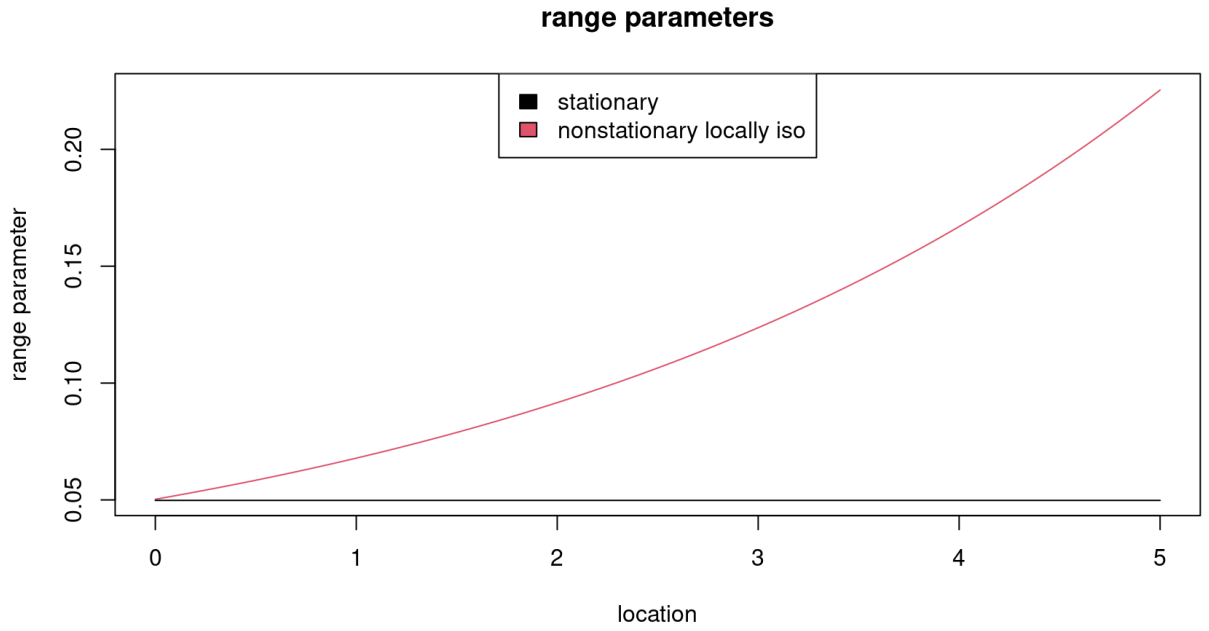


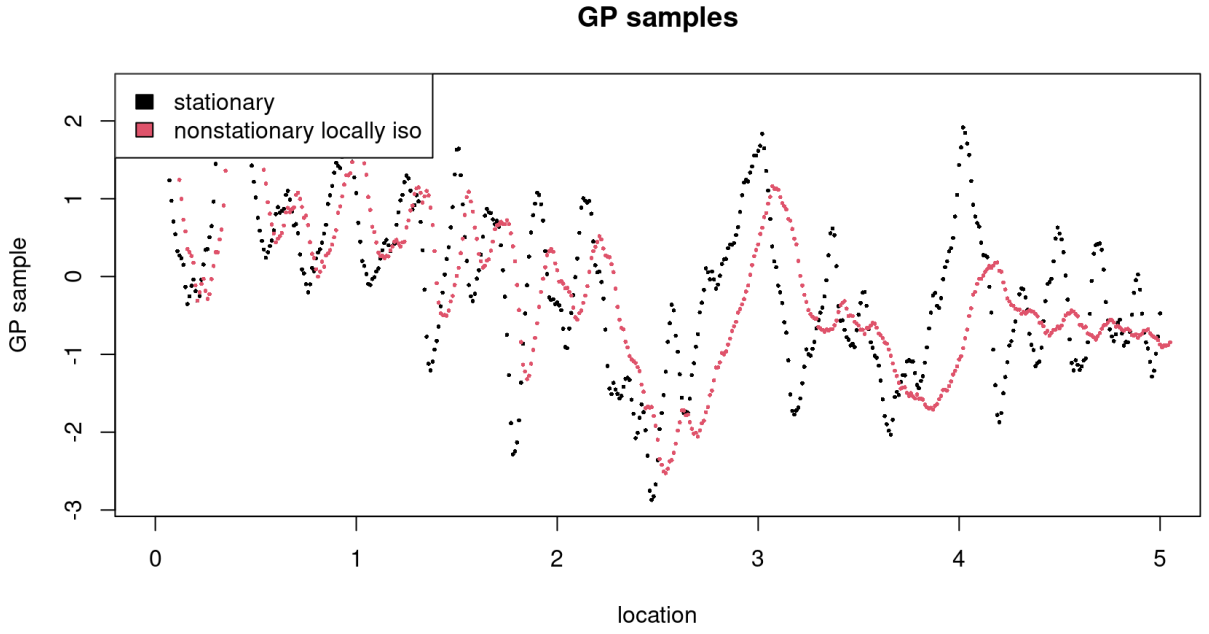


On the first plot, we can see the **field of range parameters**. We can see that the field of nonstationary parameters is (almost) the same as the field of stationary parameters (I added an small term to make them distinguishable). So, there should be little difference between the actual GP samples, if the parameters are the same. The second plot shows the **GP samples**. The samples are exactly the same ! That's because even though we've added a covariate for the range, the associated coefficient is \(0\). As a consequence, the range parameters do not move. Then the field is **practically stationary**, even if the model **formulation is nonstationary**.

Let's put a **positive coefficient.**

```
compare_stat_iso(range_stat = exp(-3), beta_range_iso = matrix(c(-3,.3)))
```

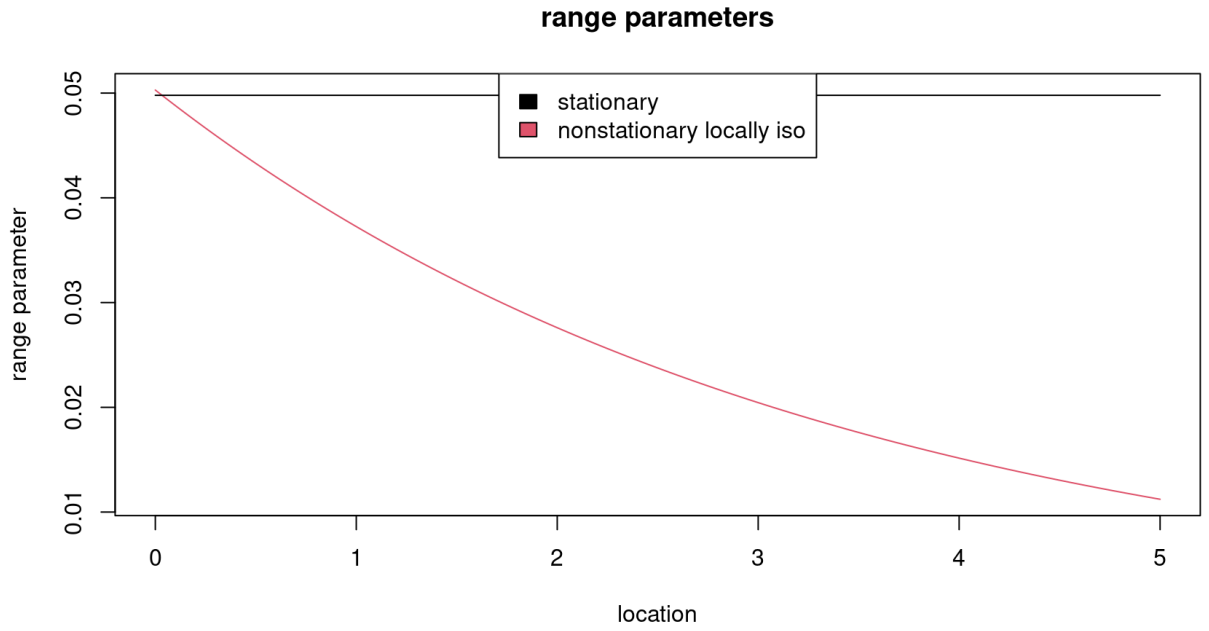


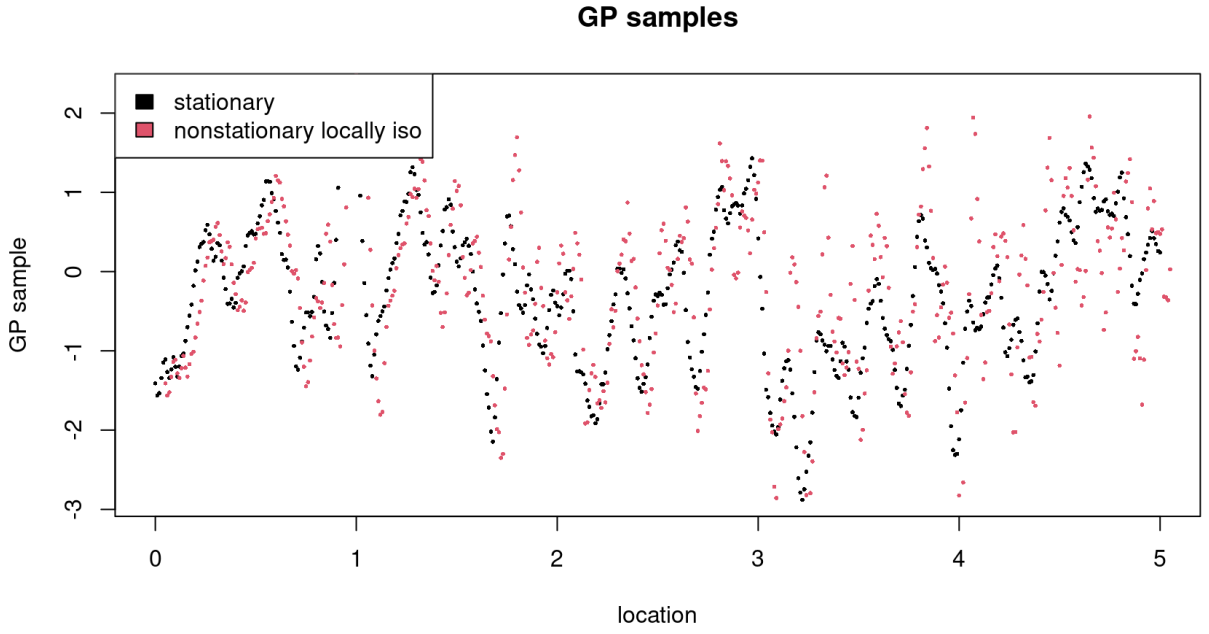


Now the **nonstationary range** becomes (exponentially) **bigger with the location**, and the GP sample becomes **more coherent** when the location is farther from the origin.

Let's put a **negative coefficient**.

```
compare_stat_iso(range_stat = exp(-3), beta_range_iso = matrix(c(-3,-.3)))
```





The nonstationary range becomes (exponentially) slower with the spatial coordinate, and the sample becomes **less coherent** when the location is farther from the origin.

2.1.4 The locally anisotropic model.

Now let's add a second dimension to the spatial locations. The covariates are now the two spatial dimensions. This is the occasion to introduce a **brutish but handy plotting function**: *plot_pointillist_painting*. This function takes spatial coordinates and an interest variable, and gives back a representation as a **pointillist painting** (https://en.wikipedia.org/wiki/Pointillism).

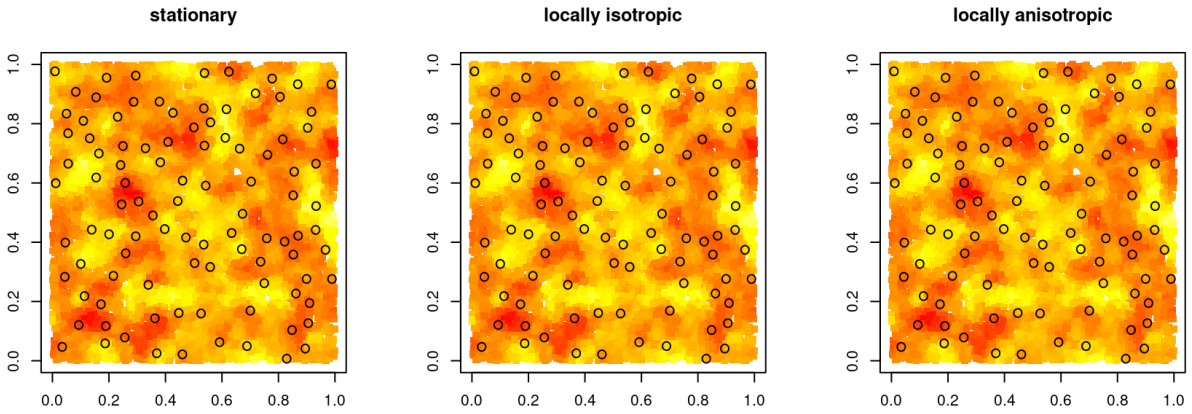
We use another function, *plot_ellipses*, who does **plot ellipses** at specified locations. It can overlay with *plot pointillist painting*.

Let's generate samples from the three Process Models and compare them using *plot_pointillist_painting*. First, let's get the samples:

```
get_samples = function(stat_range, iso_range_beta, aniso_range_beta, locs){
  # covariates for the range
  X_range = cbind(1,locs)
  # getting samples
  NNarray = GpGp::find_ordered_nn(locs, 10) # parent array for NNGP
  precisison_chol_i = row(NNarray)[!is.na(NNarray)] # row indices of sparse p
recision Cholesky
  precisison_chol_j = (NNarray)[!is.na(NNarray)] # col indices of sparse prec
ision Cholesky
  # seed vector
  seed_vec = rnorm(nrow(locs))
  # solving against precision Cholesky
  stat_sample = Matrix::sparseMatrix(
    i = precisison_chol_i, j = precisison_chol_j,
    x = GpGp::vecchia_Linv(
      covparms = c(1, stat_range, 0),
      covfun_name = "matern15_isotropic",
      locs = locs, NNarray = NNarray)[!is.na(NNarray)],
    triangular = T
  ) %>% Matrix::solve(seed_vec) %>% as.vector()
  iso_sample = Matrix::sparseMatrix(
    i = precisison_chol_i, j = precisison_chol_j,
    x = Bidart::compute_sparse_chol(
      range_beta = iso_range_beta,
      locs = locs, range_X = X_range,
      NNarray = NNarray)[[1]][!is.na(NNarray)],
    triangular = T
  ) %>% Matrix::solve(seed_vec) %>% as.vector()
  aniso_sample = Matrix::sparseMatrix(
    i = precisison_chol_i, j = precisison_chol_j,
    x = Bidart::compute_sparse_chol(
      range_beta = aniso_range_beta,
      locs = locs, range_X = X_range, anisotropic = T,
      NNarray = NNarray)[[1]][!is.na(NNarray)],
    triangular = T
  ) %>% Matrix::solve(seed_vec) %>% as.vector()
  list(
    stat_sample = stat_sample,
    iso_sample = iso_sample,
    aniso_sample = aniso_sample)
}
```

Now, let's represent the samples.

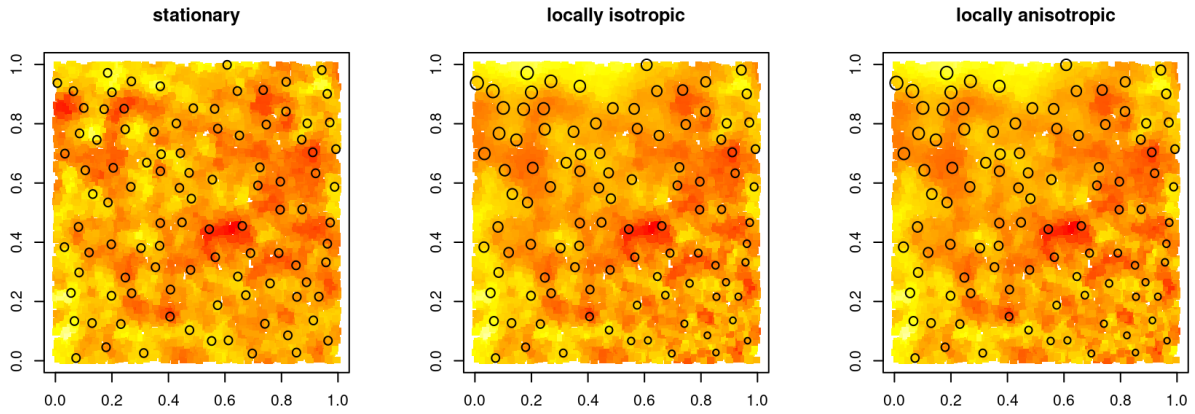
```
locs = cbind(runif(10000), runif(10000))
locs = locs[GpGp::order_maxmin(locs),]
stat_range = exp(-3.5)
iso_range_beta = matrix(c(-3.5,0,0))
aniso_range_beta = matrix(c(-3.5,0,0,0,0,0,0,0,0),3)
samples = get_samples(
  stat_range = stat_range,
  iso_range_beta = iso_range_beta,
  aniso_range_beta = aniso_range_beta,
  locs
  )
par(mfrow = c(1,3))
Bidart::plot_pointillist_painting(locs, samples$stat_sample, main = "stationa
ry")
Bidart::plot_ellipses(locs[seq(100),], log_range = matrix(-3.5, nrow(locs),
1), add = T, shrink = .02)
Bidart::plot_pointillist_painting(locs, samples$iso_sample, main = "locally i
sotropic")
Bidart::plot_ellipses(locs[seq(100),], log_range = cbind(1, locs)%*%iso_range
_beta, add = T, shrink = .02)
Bidart::plot_pointillist_painting(locs, samples$aniso_sample, main = "locally
anisotropic")
Bidart::plot_ellipses(locs[seq(100),], log_range = cbind(1, locs)%*%aniso_ran
ge_beta, add = T, shrink = .02)
```



We can see that **the three samples are the same**. Like before, that's because all coefficients for the range parameters are zero, except for the first coefficient, who parametrizes the intercept of the range. All three models are **practically stationary**, even though two of them are **potentially nonstationary**.

Now, let's add some **locally isotropic nonstationarity**. We can do that by modifying the other coefficients of *iso_range_beta*, and modifying only the **first column of *aniso_range_beta*** .

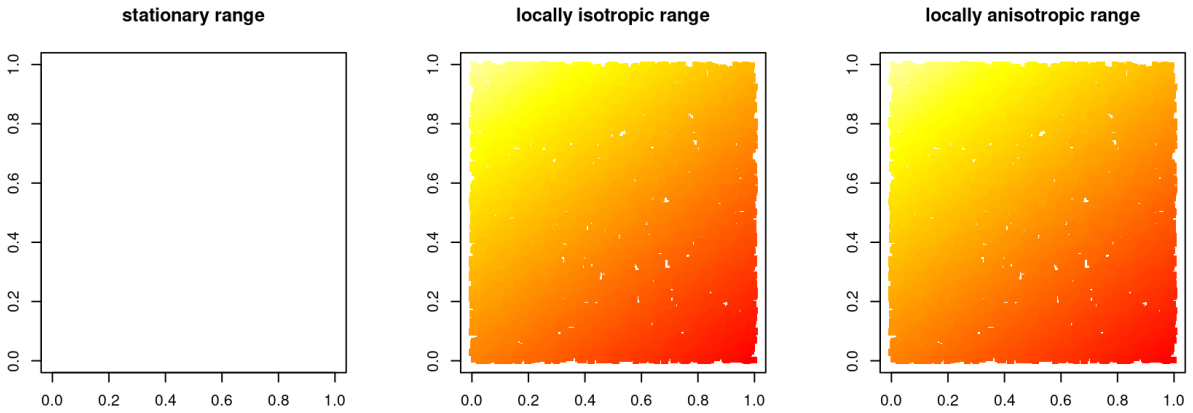
```
locs = cbind(runif(10000), runif(10000))
locs = locs[GpGp::order_maxmin(locs),]
stat_range = exp(-3.5)
iso_range_beta = matrix(c(-3.5,-1,1.4))
aniso_range_beta = matrix(c(-3.5,-1,1.4,0,0,0,0,0,0),3)
samples = get_samples(
  stat_range = stat_range,
  iso_range_beta = iso_range_beta,
  aniso_range_beta = aniso_range_beta,
  locs
  )
par(mfrow = c(1,3))
Bidart::plot_pointillist_painting(locs, samples$stat_sample, main = "stationa
ry")
Bidart::plot_ellipses(locs[seq(100),], log_range = matrix(-3.5, nrow(locs),
1), add = T, shrink = .02)
Bidart::plot_pointillist_painting(locs, samples$iso_sample, main = "locally i
sotropic")
Bidart::plot_ellipses(locs[seq(100),], log_range = cbind(1, locs)%*%iso_range
_beta, add = T, shrink = .02)
Bidart::plot_pointillist_painting(locs, samples$aniso_sample, main = "locally
anisotropic")
Bidart::plot_ellipses(locs[seq(100),], log_range = cbind(1, locs)%*%aniso_ran
ge_beta, add = T, shrink = .02)
```



We can see that the spatial range now depends on the location. **The log-range depends linearly on the coordinates thanks to the range parameters**.

If we want, we can also represent the log-range using *plot_pointillist_painting*:

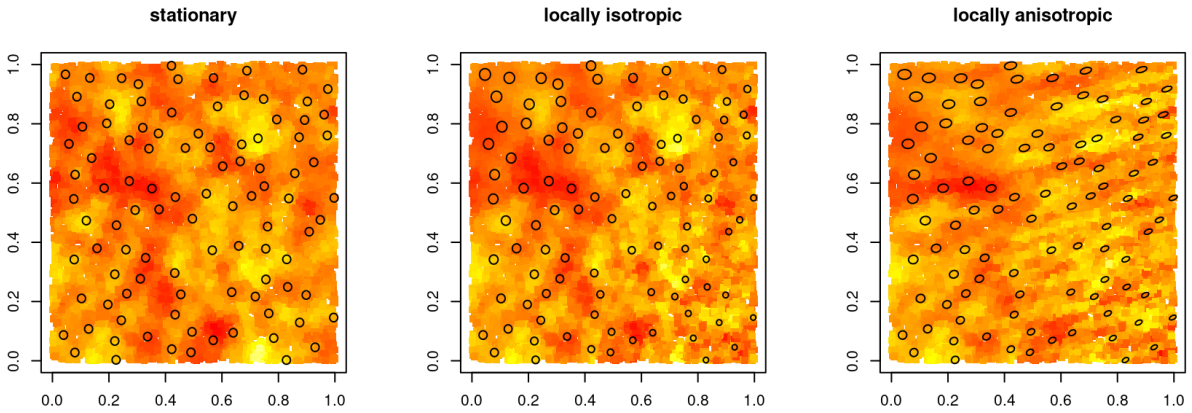
```
par(mfrow = c(1,3))
Bidart::plot_pointillist_painting(locs, rep(log(stat_range), nrow(locs)), mai
n = "stationary range")
Bidart::plot_pointillist_painting(locs, cbind(1, locs)%*%iso_range_beta, main
= "locally isotropic range")
Bidart::plot_pointillist_painting(locs, cbind(1, locs)%*%aniso_range_beta[,
1], main = "locally anisotropic range")
```



The **range field** for the **stationary** model is white because it is **constant**. The other two range fields are **identical**.

Eventually, let's add some **locally anisotropic nonstationarity** by modifying the rest of the coefficients of *aniso_range_beta*.

```
locs = cbind(runif(10000), runif(10000))
locs = locs[GpGp::order_maxmin(locs),]
stat_range = exp(-3.5)
iso_range_beta = matrix(c(-3.5,-1.4,1))
aniso_range_beta = matrix(c(-3.5,-1.4,1,0,.5,.5,0,.5,0),3)
samples = get_samples(
  stat_range = stat_range,
  iso_range_beta = iso_range_beta,
  aniso_range_beta = aniso_range_beta,
  locs
  )
par(mfrow = c(1,3))
Bidart::plot_pointillist_painting(locs, samples$stat_sample, main = "stationa
ry")
Bidart::plot_ellipses(locs[seq(100),], log_range = matrix(-3.5, nrow(locs),
1), add = T, shrink = .02)
Bidart::plot_pointillist_painting(locs, samples$iso_sample, main = "locally i
sotropic")
Bidart::plot_ellipses(locs[seq(100),], log_range = cbind(1, locs)%*%iso_range
_beta, add = T, shrink = .02)
Bidart::plot_pointillist_painting(locs, samples$aniso_sample, main = "locally
anisotropic")
Bidart::plot_ellipses(locs[seq(100),], log_range = cbind(1, locs)%*%aniso_ran
ge_beta, add = T, shrink = .02)
```



We can see that, the **anisotropy** (specifically, the Matrix logarithm of the range ellipse) **depends linearly on the spatial coordinates**.

A last remark. The **range parameters** for the **elliptic model** have **three columns**, while the range parameters for the locally isotropic model have only one:

```
cat("Head of the range parameters for the locally isotropic model: \n")
```

```
## Head of the range parameters for the locally isotropic model:
```

```
head(cbind(1, locs)%*%iso_range_beta)
```

```
##           [,1]
## [1,] -3.687093
## [2,] -3.411689
## [3,] -4.301336
## [4,] -3.459600
## [5,] -4.035119
## [6,] -4.068486
```

```
cat("Head of the range parameters for the locally anisotropic model: \n")
```

```
## Head of the range parameters for the locally anisotropic model:
```

```
head(cbind(1, locs)%*%aniso_range_beta)
```

```
##           [,1]      [,2]      [,3]
## [1,] -3.687093 0.6297968 0.3013931
## [2,] -3.411689 0.4669654 0.1761708
## [3,] -4.301336 0.3672588 0.3199695
## [4,] -3.459600 0.7216767 0.2922819
## [5,] -4.035119 0.5236156 0.3296564
## [6,] -4.068486 0.7455037 0.4290611
```

But we can see that the isotropic parameters and the **first column of the elliptic parameters** are the same. This column determines the **size of the ellipses** but not their shape. We can see that there is

a correspondence between the size of the circles in the locally isotropic model and the size of the ellipses in the locally anisotropic model.

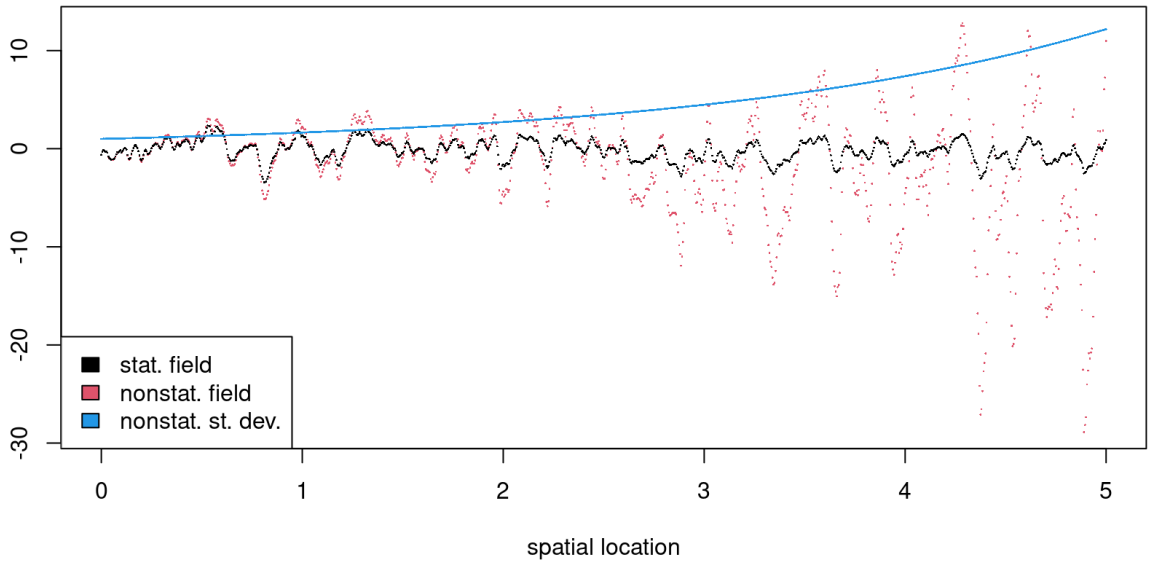
2.2 The marginal variance model.

We can also let vary the **variance of the latent field**. This will **inflate** or **shrink** the Gaussian Process. Like before, the **logarithm parametrization** is used:

\[log(\sigma^2) = X_\sigma\beta_\sigma.\] \(\sigma^2\) is the latent field variance, \(X_\sigma\) is a matrix of covariates, and \(\beta_\sigma\) is the vector of regression coefficients.

In practice, it can look like this:

```
# locations varying between 0 and 5
locs = cbind(seq(0, 5, .005), 0)

# variance model
X_scale = cbind(matrix(1, nrow(locs)), locs[,1])
beta_scale = c(0, 1)
log_scale = X_scale %*% beta_scale

# parents array for NNGP
NNarray = GpGp::find_ordered_nn(locs, 10)
# stationary correlation
chol_precision_stat =
  # computing coefficients
  GpGp::vecchia_Linv(covparms = c(1,.02, 0), covfun_name = "matern15_isotropi
c",
                     locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
chol_precision_stat = Matrix::sparseMatrix(
    x = chol_precision_stat[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
    )

# sampling a stationary latent field
stat_latent_field = as.vector(Matrix::solve(chol_precision_stat, rnorm(nrow(l
ocs))))
# scaling the latent field to make it nonstationary
nonstat_latent_field = exp(.5*c(log_scale)) * stat_latent_field
plot(locs[,1], nonstat_latent_field, pch = 16, cex = .2, col = 2, ylab = "",
xlab = "spatial location")
points(locs[,1],stat_latent_field, pch = 16, cex = .2, col = 1)
points(locs[,1],exp(.5*(X_scale %*% beta_scale)), col=4, cex = .2, pch= 16)
legend("bottomleft", legend = c("stat. field", "nonstat. field", "nonstat. st
. dev."), fill = c(1,2,4))
```

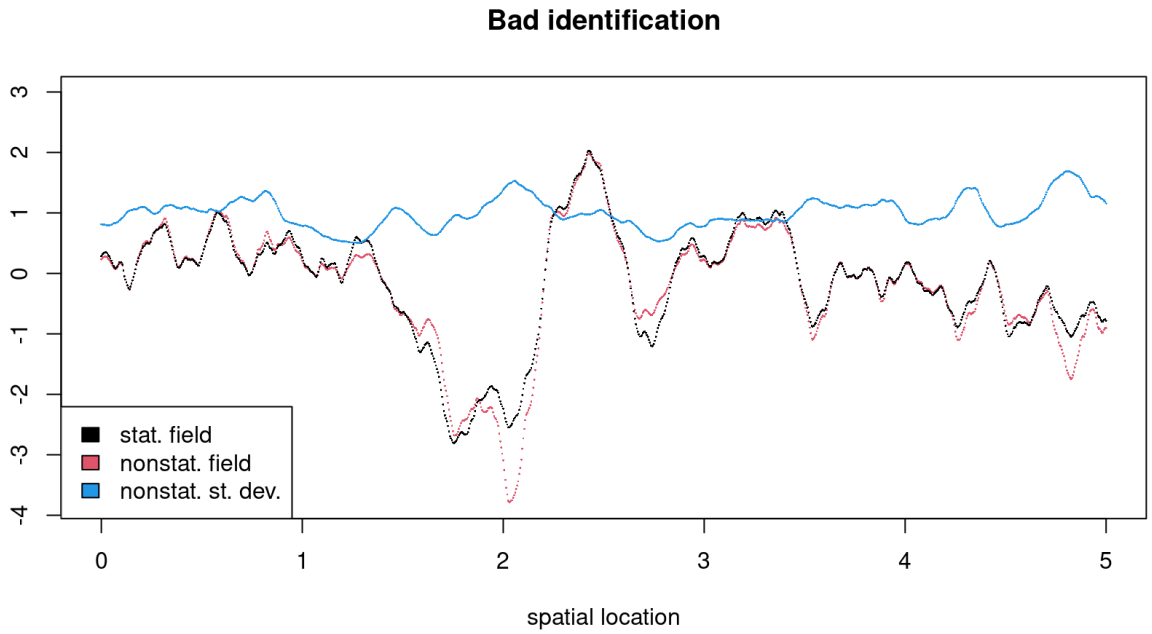


The distance to the origin is used as a covariate. The farther from the origin, the higher the variance.

2.3 Don’t bite more than you can chew! Identificability, interpretability issues.

Always be sure that the regressors you use to model the range and the variance have a strong spatial coherence. More precisely, **the regressors must be much more coherent than the Gaussian latent field**. If that is not the case, the model would be impossible to interpret: for example, in the figure below, one asks “did the nonstationary latent field become bigger because its marginal variance is bigger / its range is smaller, or just because of its intrinsic spatial variation ?”. This will also cause identification issues and computational breakdowns in the model.

```
# locations varying between 0 and 5
locs = cbind(seq(0, 5, .005), 0)

# variance model
beta_scale = c(0, .5)

# parents array for NNGP
NNarray = GpGp::find_ordered_nn(locs, 10)
# stationary correlation
chol_precision_stat =
  # computing coefficients
  GpGp::vecchia_Linv(covparms = c(1,.1, 0), covfun_name = "matern15_isotropi
c",
                     locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
chol_precision_stat = Matrix::sparseMatrix(
    x = chol_precision_stat[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
    )

# sampling a stationary latent field
stat_latent_field = as.vector(Matrix::solve(chol_precision_stat, rnorm(nrow(l
ocs))))
# sampling another stationary latent field used as a regressor for the scale
X_scale = cbind(1, as.vector(Matrix::solve(chol_precision_stat, rnorm(nrow(lo
cs)))))
# obtaining log-scale
log_scale = X_scale %*% beta_scale
# scaling the latent field to make it nonstationary
nonstat_latent_field = exp(.5*c(log_scale)) * stat_latent_field
plot(locs[,1], nonstat_latent_field, pch = 16, cex = .2, col = 2, ylab = "",
xlab = "spatial location", ylim = c(min(nonstat_latent_field), max(nonstat_la
tent_field)+1), main = "Bad identification")
points(locs[,1],stat_latent_field, pch = 16, cex = .2, col = 1)
points(locs[,1],exp(.5*(log_scale)), col=4, cex = .2, pch= 16)
legend("bottomleft", legend = c("stat. field", "nonstat. field", "nonstat. st
. dev."), fill = c(1,2,4))
```



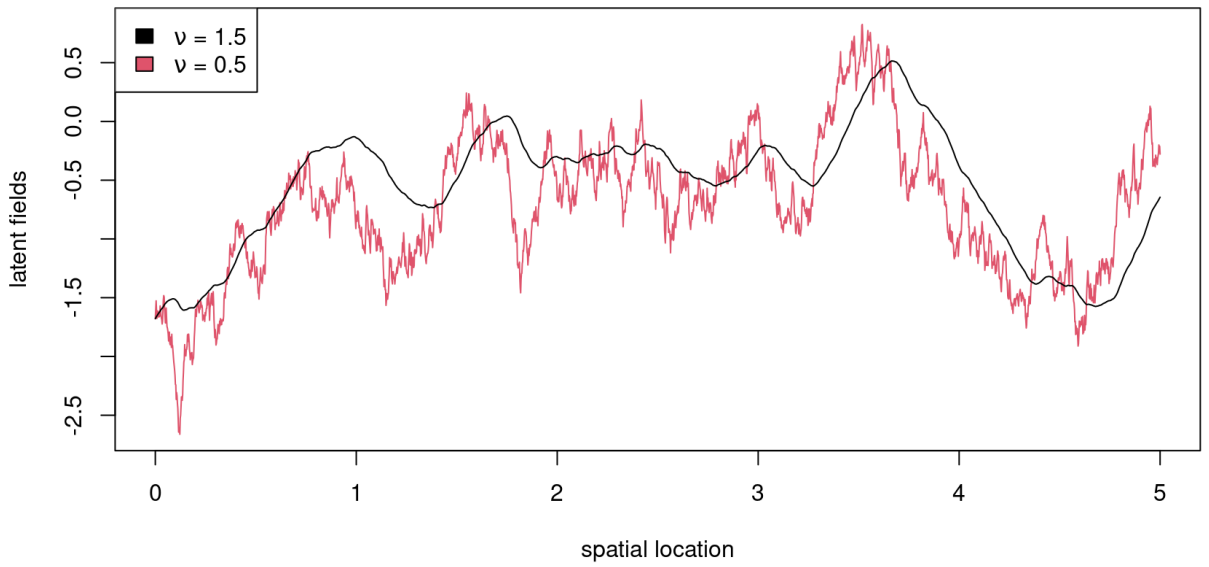
There also is a well-known **lack of identification between the Gaussian Process spatial range and its spatial variance**, even in stationary models (https://www.tandfonline.com/doi/abs/10.1198/016214504000000241). Even though we obtained encouraging results on synthetic data sets, on real data sets it is a risky choice to have both nonstationarities.

2.4 Don’t forget the (stationary) smoothness.

The smoothness \(\nu\) of the Gaussian Process is **fixed by the user**. This parameter may affect **prediction** performance.

```
set.seed(1)
# locations varying between 0 and 5
locs = cbind(seq(0, 5, .002), 0)

# parents array for NNGP
NNarray = GpGp::find_ordered_nn(locs, 10)

# correlation with nu  = 1.5
chol_precision_15 =
  # computing coefficients
  GpGp::vecchia_Linv(covparms = c(1,.4, 0), covfun_name = "matern15_isotropi
c",
                     locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
chol_precision_15 = Matrix::sparseMatrix(
    x = chol_precision_15[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
    )

# correlation with nu  = 0.5
chol_precision_5 =
  # computing coefficients
  GpGp::vecchia_Linv(covparms = c(1,1, 0), covfun_name = "exponential_isotrop
ic",
                     locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
chol_precision_5 = Matrix::sparseMatrix(
    x = chol_precision_5[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
    )

# sampling stationary latent fields
seed_vector = rnorm(nrow(locs))
latent_field_15 = as.vector(Matrix::solve(chol_precision_15, seed_vector))
latent_field_5 = as.vector(Matrix::solve(chol_precision_5, seed_vector))

ordr = order(locs[,1])

plot(
  locs[ordr,1],
  latent_field_5[ordr],
  pch = 16, cex = .2, col = 2,
  ylab = "latent fields", xlab = "spatial location",
  ylim = c(
    min(latent_field_15,latent_field_5),
    max(latent_field_15, latent_field_5)),
  type = "l")
lines(
  locs[ordr,1],
  latent_field_15[ordr],
```

```
  pch = 16,
  cex = .2,
  col = 1)
legend("topleft", legend = c(expression(paste(nu, " = 1.5")), expression(past
e(nu, " = 0.5"))), fill = c(1,2))
```



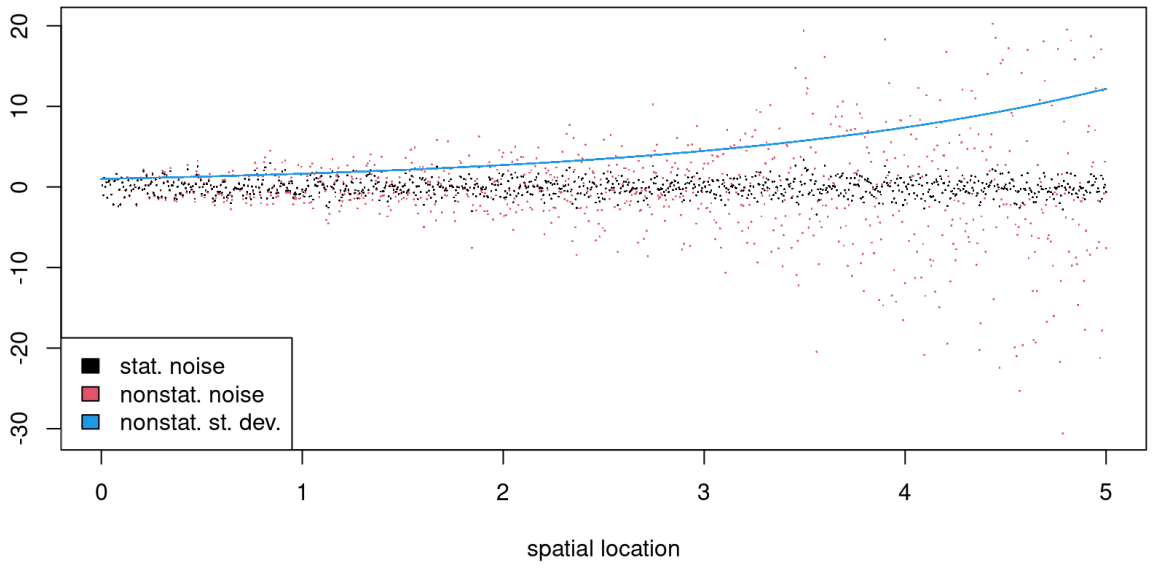
2.5 Nonstationary Gaussian data model.

The data model makes the link between the covariates and the latent field on one hand, and the **observations** on the other hand. It is analogous to the choice of the **link function** in a generalized linear model. Here, we assume a **Gaussian link**, who can nonetheless accommodate **heteroskedasticity**. Like in the process model, we have a **logarithmic parametrization**:

\[\tau^2 = X_\tau \beta_\tau,\] \(\tau^2\) being the noise variance, \(X_\tau\) being covariates, and \(\beta_\tau\) being the regression coefficients. Unlike the range and variance, \(X_\tau\) needs not to be spatially smooth.

In practice, it can look like this:

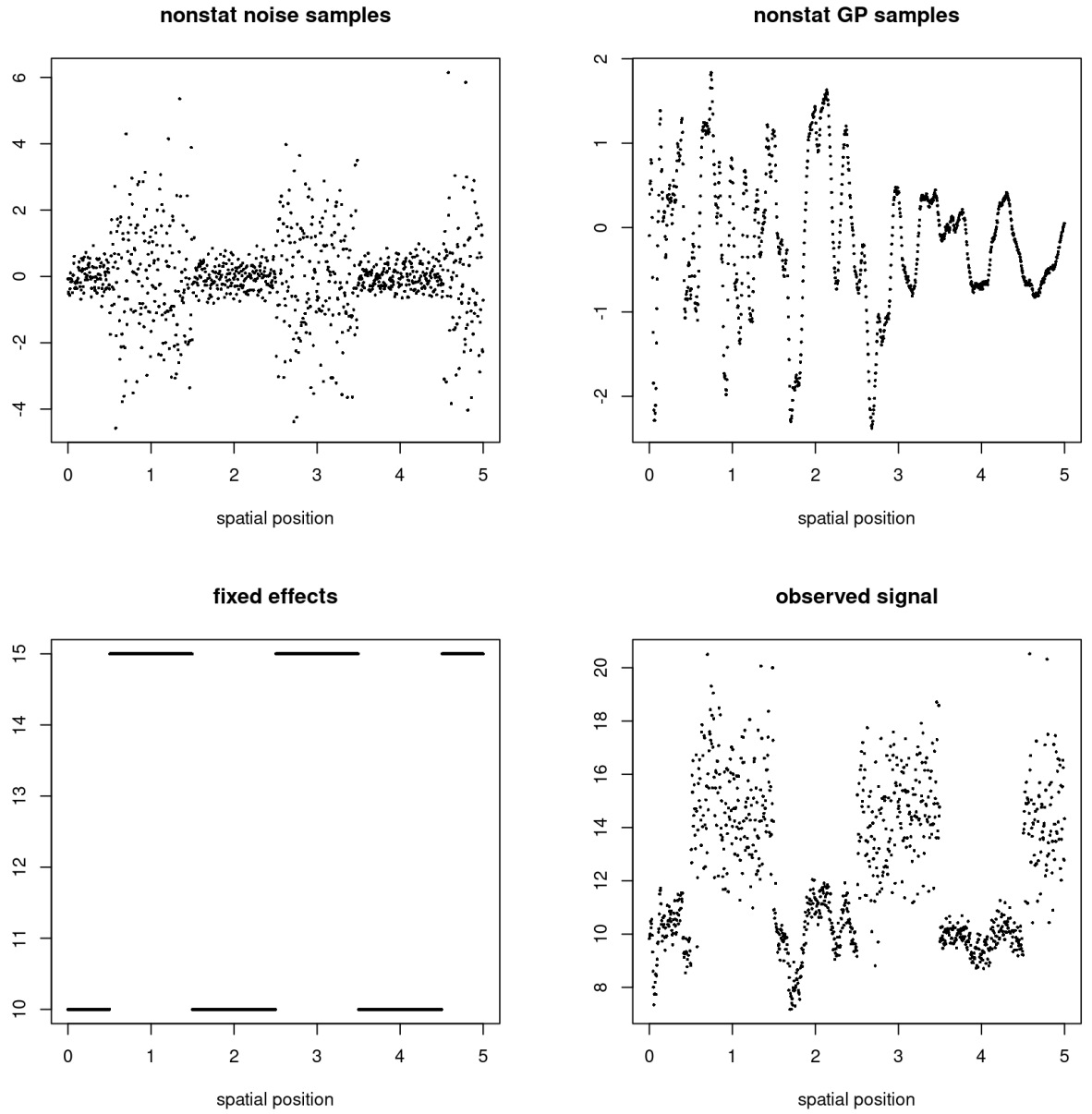
```
# locations varying between 0 and 5
locs = cbind(seq(0, 5, .005), 0)

# noise variance model
X_noise = cbind(matrix(1, nrow(locs)), locs[,1])
beta_noise = c(0, 1)
log_noise_var = X_noise %*% beta_noise

# sampling a white noise
white_noise = rnorm(nrow(locs))
# scaling the noise to make it nonstationary
nonstat_noise = exp(.5*c(log_noise_var)) * white_noise
plot(locs[,1], nonstat_noise, pch = 16, cex = .2, col = 2, ylab = "", xlab =
"spatial location")
points(locs[,1],white_noise, pch = 16, cex = .2, col = 1)
points(locs[,1],exp(.5*(X_noise %*% beta_noise)), col=4, cex = .2, pch= 16)
legend("bottomleft", legend = c("stat. noise", "nonstat. noise", "nonstat. st
. dev."), fill = c(1,2,4))
```



2.6 Conclusion for *Nonstationary Processes*.

We have a **Matérn Gaussian Process model**, with **user-chosen smoothness** (\(\nu = 0.5\) or \(\nu = 1.5\) are the two options). We have **3 range models of increasing complexity**, with the **simpler models included in the complex models**. Spatially-driven **marginal variance** is available as well, but should not be mixed with spatially-driven range in real data sets even though we observed some good behavior on the simulations. We have a **heteroskedastic Gaussian data model**. Like in most geostatistical models, we also have **fixed effects who link the response variable to some exogenous variables**. We are therefore able to carry out a data decomposition as the one that follows:

```
# locations varying between 0 and 5
locs = cbind(seq(0, 5, .005), 0)

# noise variance model
X_noise = cbind(matrix(1, nrow(locs)), round(locs[,1])%%2)
beta_noise = c(-2, 3)
log_noise_var = X_noise %*% beta_noise
# range model
X_range = cbind(matrix(1, nrow(locs)), locs[,1])
beta_range = matrix(c(-4,.5))
# fixed effects
X = cbind(matrix(1, nrow(locs)), round(locs[,1])%%2)
beta = c(10, 5)

# NNGP setup
NNarray = GpGp::find_ordered_nn(locs, 20)
# locally isotropic covariance
nonstat_vecchia_chol =
  # computing coefficients
  Bidart::compute_sparse_chol(
    range_beta = beta_range,
    NNarray = NNarray,
    locs = locs,
    range_X = X_range)
# putting coefficients in precision Cholesly
nonstat_vecchia_chol = Matrix::sparseMatrix(
  x = nonstat_vecchia_chol[[1]][!is.na(NNarray)],
  i = row(NNarray)[!is.na(NNarray)],
  j = (NNarray)[!is.na(NNarray)]
)

# getting nonstat Gaussian Process samples using precision Cholesky method
nonstat_GP_samples = Matrix::solve(nonstat_vecchia_chol, rnorm(nrow(locs)))
# getting nonstat noise
nonstat_noise_samples = exp(.5*log_noise_var) * rnorm(nrow(locs))
# getting fixed effects
fixed_effects = X %*% beta

par(mfrow = c(2,2))
plot(locs[,1], nonstat_noise_samples, ylab = "", main = "nonstat noise sample
s", pch = 16, cex = .4, xlab = "spatial position")
plot(locs[,1], nonstat_GP_samples, ylab = "", main = "nonstat GP samples", pc
h = 16, cex = .4, xlab = "spatial position")
plot(locs[,1], fixed_effects, ylab = "", main = "fixed effects", pch = 16, ce
x = .4, xlab = "spatial position")
plot(locs[,1], nonstat_noise_samples + nonstat_GP_samples + fixed_effects, yl
ab = "", main = "observed signal", pch = 16, cex = .4, xlab = "spatial positi
on")
```



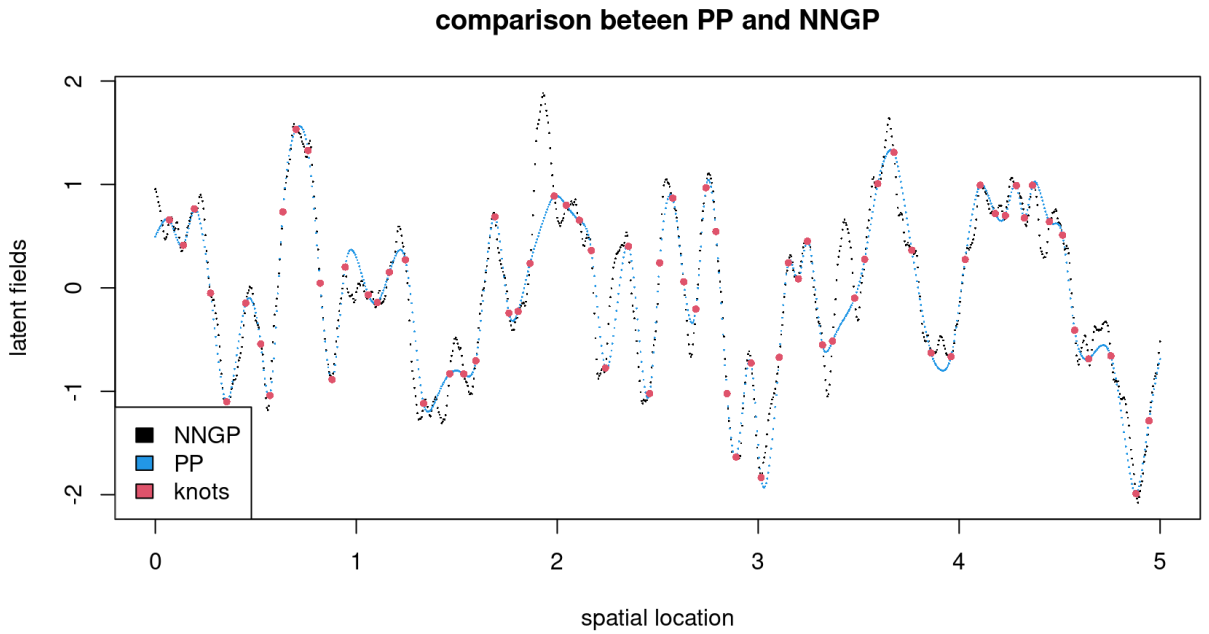
3 The Predictive Process (PP) prior for spatially-indexed parameters.

In the past walkthrough, we have only used the two spatial coordinates as **covariates for the parameter fields**. We can obviously put more interesting covariates than this. For example, it is often a good idea to include **elevation**. However, what if we don't (only) want to pass some informative variables to the model, but also to let it **find spatial patterns in the nonstationarity** on its own? That's what the **Predictive Processes** are for.

3.1 What are Predictive Processes ?

The Predictive Processes (https://rss.onlinelibrary.wiley.com/doi/10.1111/j.1467-9868.2008.00663.x) (PPs) are a **Gaussian Process approximation** that replaces the value of the target Gaussian Process by the **conditional expectation knowing a restricted set of locations, the knots**. As a side note, in the case of NNGPs and Vecchia's approximation, we obtain a PP by keeping only the first columns of the NNGP-induced precision Cholesky factor. This method is related to **P-splines**. The shortcoming of this method is that it causes **over-smoothing** (see graph below).

```
# locations varying between 0 and 5
locs = cbind(seq(0, 5, .005), 0)
# reordering the locations to have a well-spreat set of knots
locs = locs[GpGp::order_maxmin(locs),]

# parents array for NNGP
NNarray = GpGp::find_ordered_nn(locs, 10)

# NNGP
chol_precision =
  # computing coefficients
  GpGp::vecchia_Linv(covparms = c(1,.06, 0), covfun_name = "matern15_isotropi
c",
                     locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
chol_precision = Matrix::sparseMatrix(
    x = chol_precision[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
    )

# sampling the processes
seed_vector = rnorm(nrow(locs))
latent_field = as.vector(Matrix::solve(chol_precision, seed_vector))
# keeping only first columns of the NNGP factor
PP_dropout = rep(0, nrow(locs)); PP_dropout[seq(70)] = 1
PP = as.vector(Matrix::solve(chol_precision, seed_vector * PP_dropout))

plot(locs[,1],latent_field, pch = 16, cex = .2, col = 1, xlab = "spatial loca
tion", ylab = "latent fields", main = "comparison beteen PP and NNGP")
points(locs[,1],PP, pch = 16, cex = .2, col = 4)
points(locs[seq(70),1],PP[seq(70)], pch = 16, cex = .7, col = 2)
legend("bottomleft", legend = c("NNGP", "PP", "knots"), fill = c(1,4, 2))
```



We can see that even though the PP gives a good general idea of the NNGP, some details are missed.

3.2 The covariance parameters of the Predictive Process.

The PPs have themselves covariance parameters. In the original paper (https://rss.onlinelibrary.wiley.com/doi/10.1111/j.1467-9868.2008.00663.x), the range, variance, and smoothness of the PPs are estimated. In our case, because of the identification problems and need for simplicity, we **estimate only the marginal variance parameter**. Of course, we do not any nonstationarity in the PP.

Therefore, the covariance of the PP is parametrized as: \[\sigma_{PP}\times C_{PP},\] \(\sigma_{PP}\) being the PP variance parameter, and \(C_{PP}\) being the user-specified **degenerate correlation matrix** of the PP.

One point you may want to overlook for now, but which is quite important, is that in our model the PP is parametrized in terms of **regression coefficients**, just like the covariates who explain the range, the process variance, and the noise variance. This may seem incoherent with the above formulation with a **covariance matrix**, but it is not. In fact, \(C_{PP}\) is **degenerate** and can be written as \(C_{PP} = B~B^T\), \(B\) being a matrix of PP **basis functions** with as many columns as there are knots, that is in practice much less that the number of spatial locations. To sample (https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=759e42e9e29595f9060f842365dee34ad086a33e) from the PP, you need to multiply \(B\) by a vector of random coefficients, and it is those coefficients who are estimated by our model for the sake of **parsimony**.

3.3 Don’t bite bigger than you can chew! (again)

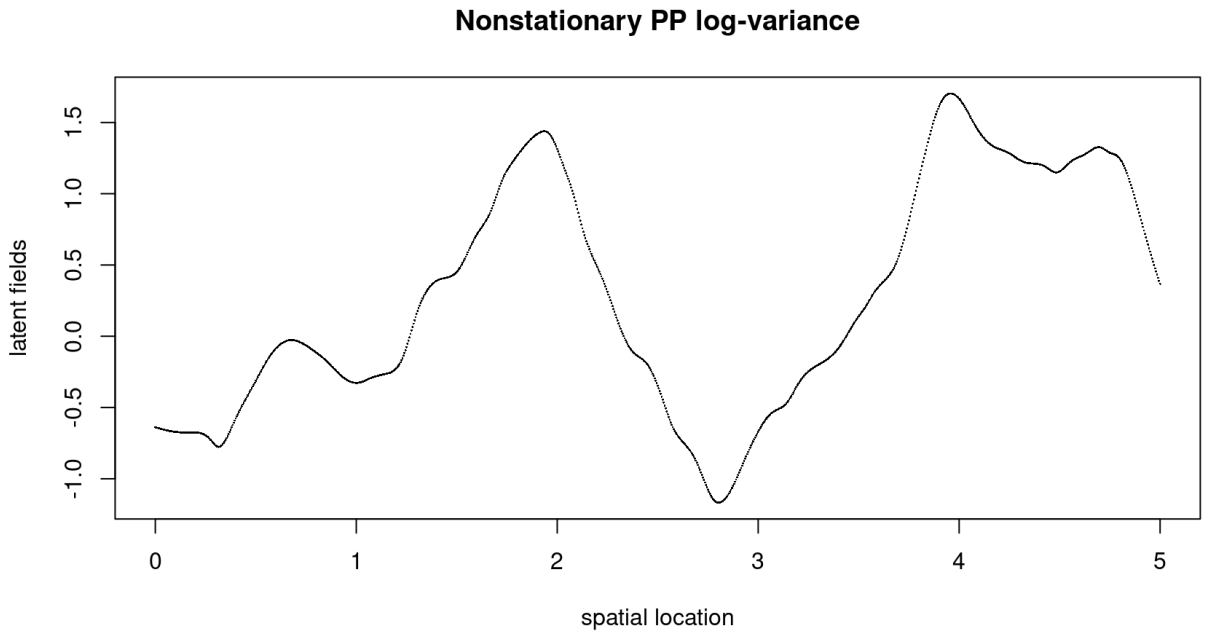
The model we propose looks for large variations of the covariance parameters, due to the aforementioned identification problems. Over-smoothing should therefore nor be too crippling. Actually, we remarked that putting many knots caused problems in the MCMC behavior, so **going for an over-smoothed prior is the safer option**. We did no sensitivity analysis with respect to the parameters of the PP, but processes with high range cannot identify the variance (https://www.tandfonline.com/doi/abs/10.1198/016214504000000241), so if the PP range is lowered, the variance should adjust and give an equivalent process. The author of the vignette actually experienced this: while doing a run on a synthetic data set, he saw, with displeasure, that the PP variance estimation was completely off. Then, he remarked that the PP range, which is a non-estimated user-specified hyperparameter, was off as well. The PP variance did compensate the PP range.

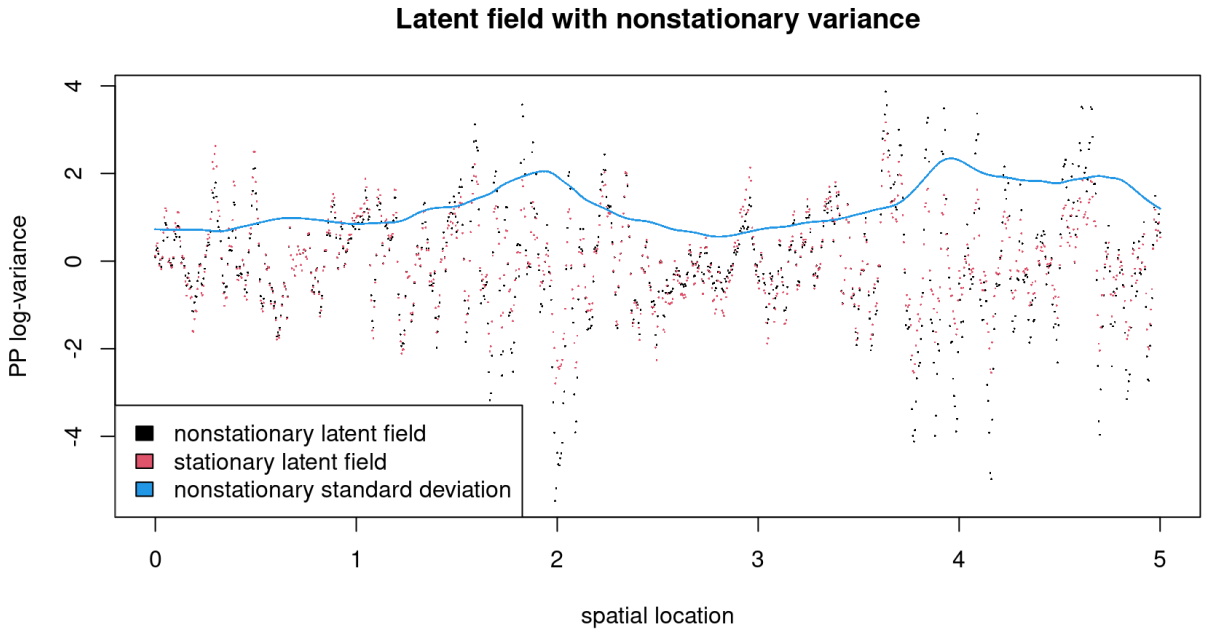
3.4 An example of Predictive Process prior for Covariance parameters.

In the following, a **PP is sampled** and then used to generate a latent field with **nonstationary marginal variance**.

```
# locations varying between 0 and 5
locs = cbind(seq(0, 5, .005), 0)
# reordering the locations to have a well-spread set of knots
locs = locs[GpGp::order_maxmin(locs),]

# parents array for NNGP
NNarray = GpGp::find_ordered_nn(locs, 10)

# Making the PP
# First make a NNGP
chol_precision =
  # computing coefficients
  GpGp::vecchia_Linv(covparms = c(1,.4, 0), covfun_name = "matern15_isotropi
c",
                     locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
chol_precision = Matrix::sparseMatrix(
    x = chol_precision[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
    )

# sampling the processes
seed_vector = rnorm(nrow(locs))
# keeping only first columns of the NNGP factor
PP_dropout = rep(0, nrow(locs)); PP_dropout[seq(70)] = 1
PP = as.vector(Matrix::solve(chol_precision, seed_vector * PP_dropout))

# sampling a stationary latent field
# (it will later be multiplied by the nonstationary standard deviation)

chol_precision =
  # computing coefficients
  GpGp::vecchia_Linv(covparms = c(1,.01, 0), covfun_name = "matern15_isotropi
c",
                     locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
chol_precision = Matrix::sparseMatrix(
    x = chol_precision[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
    )
# sampling the stationary process
seed_vector = rnorm(nrow(locs))
stat_latent_field = as.vector(Matrix::solve(chol_precision, seed_vector))

# Multiplying the stat latent field to get a nonstat latent field
nonstat_latent_field = exp(.5*PP) * stat_latent_field

# Plotting
par(mfrow = c(2, 1))
plot(locs[,1],PP, pch = 16, cex = .2, xlab = "spatial location", ylab = "late
```

```
nt fields", main = "Nonstationary PP log-variance")
plot(locs[,1],nonstat_latent_field, pch = 16, cex = .2, xlab = "spatial locat
ion", ylab = "PP log-variance", main = "Latent field with nonstationary varia
nce")
points(locs[,1],stat_latent_field, pch = 16, cex = .2, col = 2)
points(locs[,1],exp(.5*PP), pch = 16, cex = .2, col = 4)
legend("bottomleft", legend = c("nonstationary latent field", "stationary lat
ent field", "nonstationary standard deviation"), fill =c(1,2,4))
```





3.5 Predictive process as a prior for anisotropic range parameters.

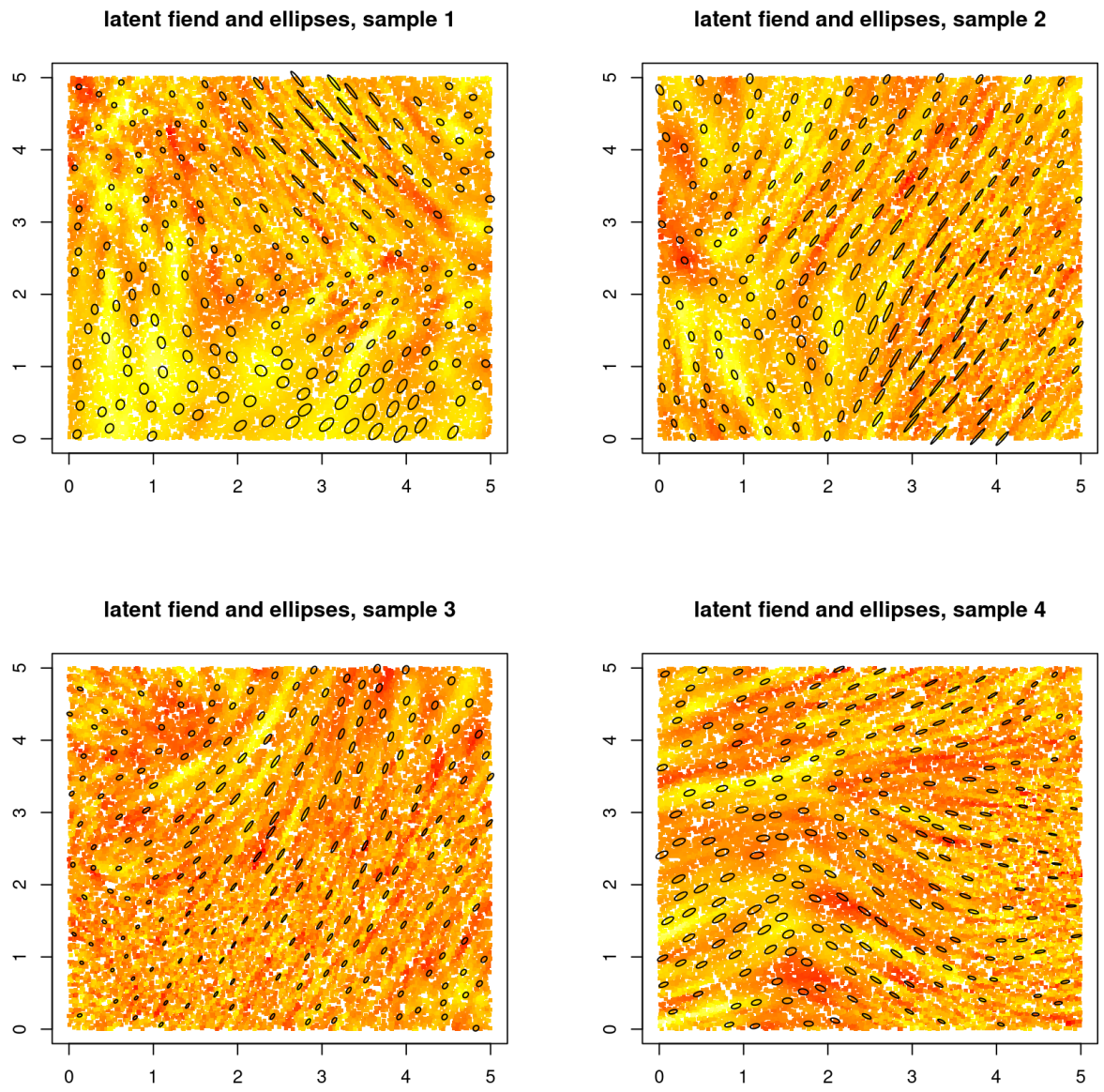
In the case of **anisotropic range** (log_range), the PP must have **three variables** in order to model the **half-vectorization of the matrix logarithm**. The variance of those three parameters is parametrized as a \(3\times3\) matrix. The PP covariance now becomes

\[\Sigma_{PP}\otimes C_{PP}, \] \(\otimes\) being Kronecker's product (https://en.wikipedia.org/wiki/Kronecker_product), \(\Sigma_{PP}\) now being a \(3\times 3\) covariance matrix, and \(C_{PP}\) being a correlation matrix like before. Since the first component of the matrix logarithm controls the spatial range, the **upper-left coefficient** of \(\Sigma_{PP}\) allows the **range to vary in space**. Since the other two components of the matrix logarithm control the anisotropy, the **lower-right** \(2\times 2\) sub-matrix of \(\Sigma_{PP}\)allows the **anisotropy** to vary in space. Let's make a function that gives a **sample of the PP** and of **the nonstationary Gaussian Process**, given a PP variance parameter.

```
Sigma_PP_demo = function(Sigma_PP, sample_num){
  locs = 5*cbind(runif(20000), runif(20000))
  locs = locs[GpGp::order_maxmin(locs),]
 # getting samples
  NNarray = GpGp::find_ordered_nn(locs, 10) # parent array for NNGP
  precisison_chol_i = row(NNarray)[!is.na(NNarray)] # row indices of sparse p
recision Cholesky
  precisison_chol_j = (NNarray)[!is.na(NNarray)] # col indices of sparse prec
ision Cholesky

  # Making the PP
  # First make a NNGP
  chol_precision =
    # computing coefficients
    GpGp::vecchia_Linv(covparms = c(1,1, 0), covfun_name = "matern15_isotropi
c",
                       locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
  chol_precision = Matrix::sparseMatrix(
    x = chol_precision[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
  )

  # getting PP basis functions
  PP = as.matrix(Matrix::solve(chol_precision, diag(1, nrow(locs), 20)))

  # getting PP multiplicator
  PP_beta = matrix(rnorm(60), 20) %*% chol(Sigma_PP)

  seed_vec = rnorm(nrow(locs))
  aniso_sample = Matrix::sparseMatrix(
    i = precisison_chol_i, j = precisison_chol_j,
    x = Bidart::compute_sparse_chol(
      range_beta = rbind(c(-2,0,0), PP_beta),
      locs = locs, range_X = cbind(1, PP), anisotropic = T,
      NNarray = NNarray)[[1]][!is.na(NNarray)],
    triangular = T
  ) %>% Matrix::solve(seed_vec) %>% as.vector()
  Bidart::plot_pointillist_painting(locs, aniso_sample, cex = .5, main = past
e("latent fiend and ellipses, sample", sample_num))
   Bidart::plot_ellipses(locs[seq(200),], cbind(1, PP[seq(200),]) %*% rbind(c
(-3,0,0), PP_beta), add = T, shrink = .05)
}
```

Let's sample four examples of latent fields obtained with an identity matrix.

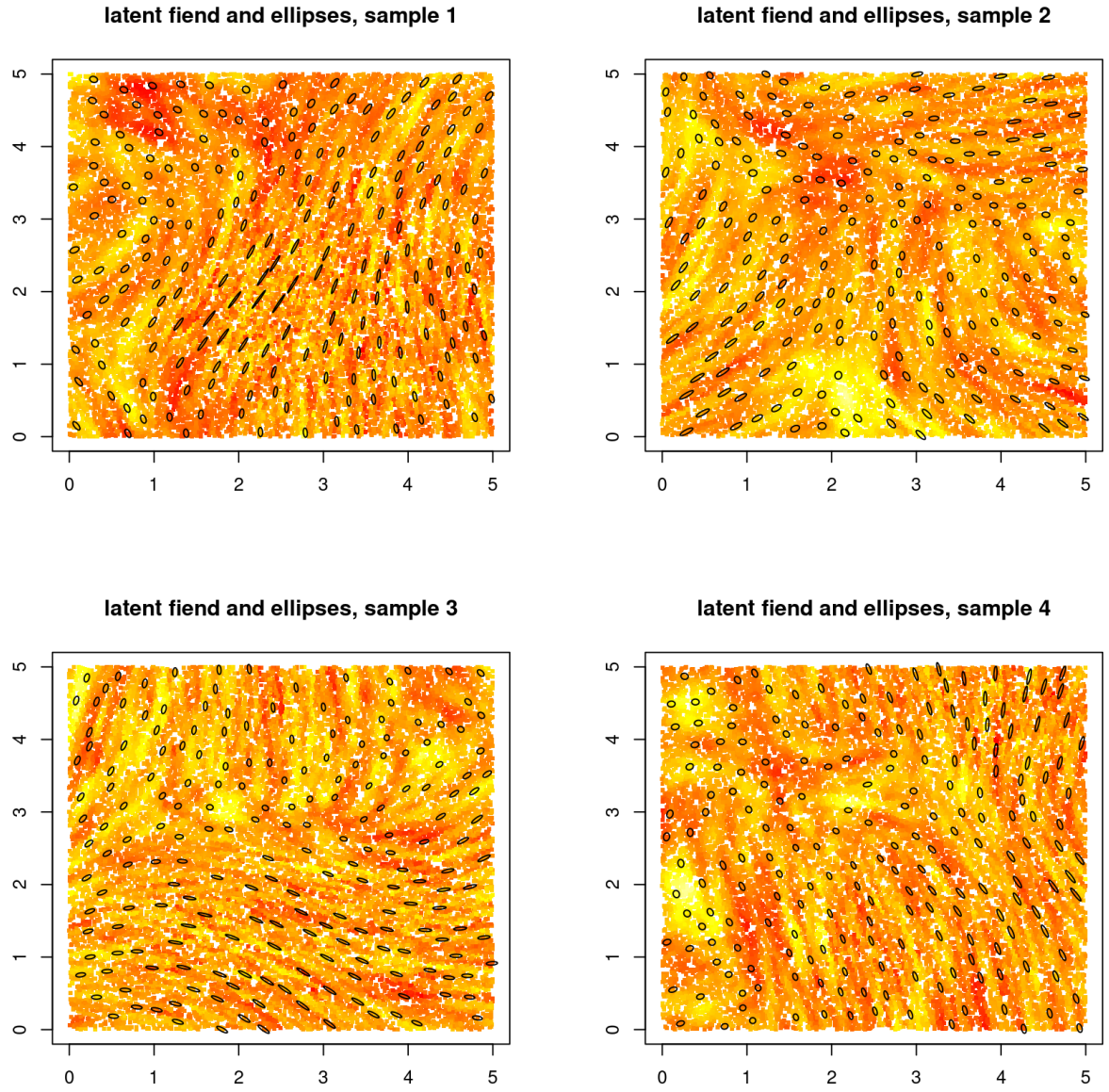
```
par(mfrow = c(2,2))
Sigma_PP = diag(3)
Sigma_PP_demo(Sigma_PP, 1)
Sigma_PP_demo(Sigma_PP, 2)
Sigma_PP_demo(Sigma_PP, 3)
Sigma_PP_demo(Sigma_PP, 4)
```



We can see that the ellipses **change in direction and in size** because all the components of the variance matrix are high.

Now, let's sample latent fields obtained with a diagonal matrix where the **first term is very small**.

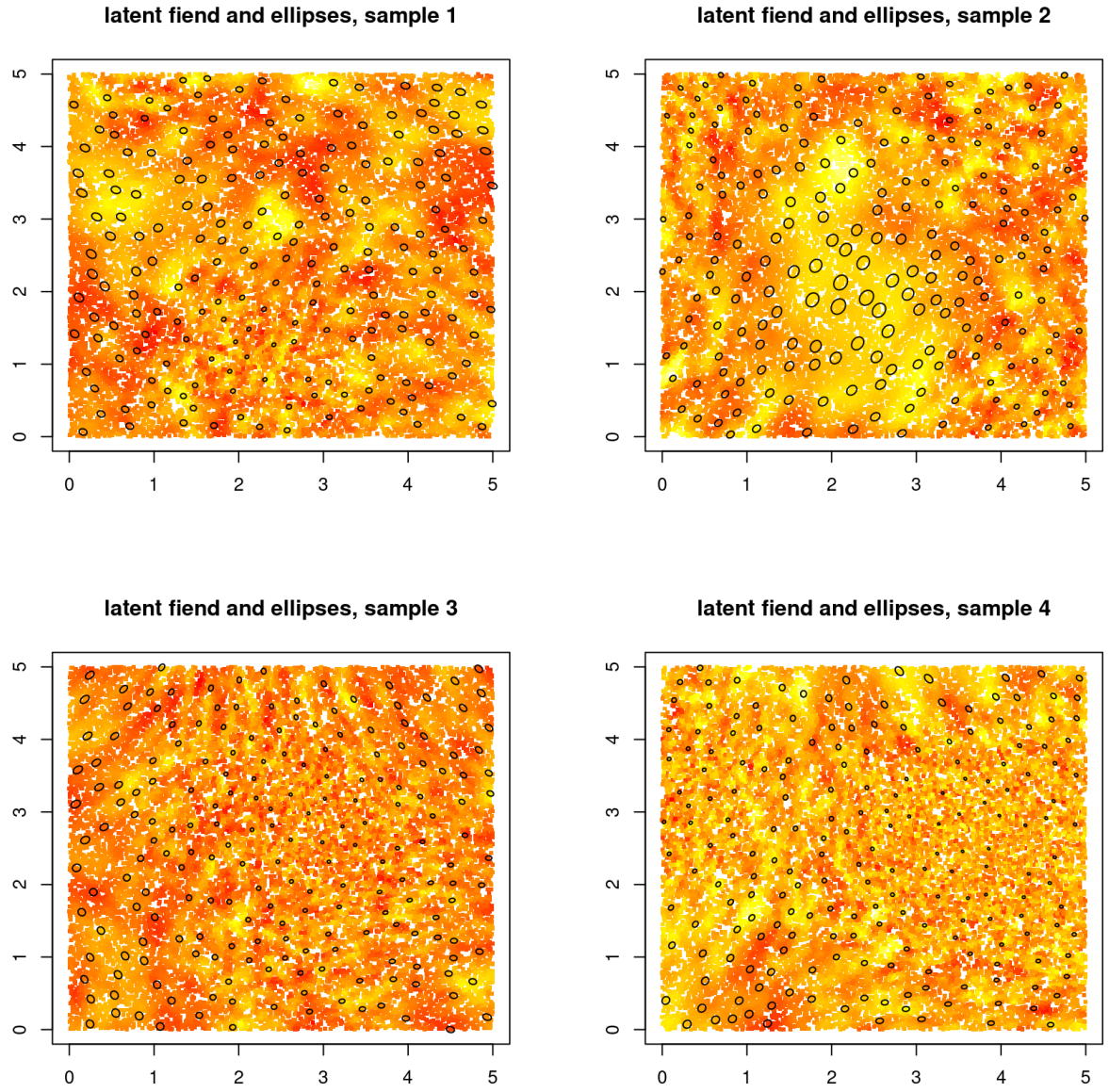
```
par(mfrow = c(2,2))
Sigma_PP = diag(c(.05, 1,1))
Sigma_PP_demo(Sigma_PP, 1)
Sigma_PP_demo(Sigma_PP, 2)
Sigma_PP_demo(Sigma_PP, 3)
Sigma_PP_demo(Sigma_PP, 4)
```



In this example, we can see that the **orientation of the correlation varies**, but the **range does not vary (much)**.

Now, let's sample latent fields obtained with a diagonal matrix where the **two last terms are very small**

```
par(mfrow = c(2,2))
Sigma_PP = diag(c(1, .05, .05))
Sigma_PP_demo(Sigma_PP, 1)
Sigma_PP_demo(Sigma_PP, 2)
Sigma_PP_demo(Sigma_PP, 3)
Sigma_PP_demo(Sigma_PP, 4)
```



Here, on the other hand, the **orientation** of the correlation **does not vary much**, but the range does. We could go on with lots of other configurations, as long as *Sigma_PP* is a valid \(3\times3\) variance matrix.

3.6 Conclusion for *The Predictive Process (PP) prior for spatially-indexed parameters*.

PPs allow to model covariance parameters with some **spatial coherence**. They combine the **simplicity** of splines and the **regulation** of Gaussian Processes. However, they must be **kept simple** in order to have a good model behavior.

4 Running the model

Now that we had a soft tour of the model's framework, let's get into the nitty-gritty of how to run it.

4.1 Initialization

First, we need to set up the model.

4.1.1 The data

You need:

- a **matrix** of **spatial coordinates** with size \(n\times 2\) .
- a **vector** of **response variable** with length \(n\) . You can add:
- optional **data.frames** of **explanatory variables** with size \(n\) rows for the fixed effects of the interest variable, the range parameters, the field variance parameters, the noise variance parameters.

The important point is that you need **one spatial site, for one observation of the interest variable, for one observation of the explanatory variables**. It is always a good idea to **have a look at your data using *plot pointillist painting*** .

We are going to **create a synthetic data set** to illustrate the following section. It has:

- 12000 spatial locations in 2 dimensions.
- stationary process spatial correlation.
- nonstationary process marginal variance, specified through a PP.
- nonstationary noise marginal variance, specified through a PP.

In the following, we **simulate a toy example** with \(12000\) observations. The first \(10000\) will be used for training, while the rest will be used for prediction and comparison.

4.1.1.1 Simulating spatial locations

First, let's get some **spatial locations**.

```
# locations varying between 0 and 5
locs = cbind(5*runif(12000, 0, 5), 5*runif(12000, 0, 5))
# reordering the locations to have a well-spreat set of knots
locs = locs[GpGp::order_maxmin(locs),]
```

4.1.1.2 Simulating a Predictive Process for the covariance parameters

Let's move on with getting a **Predictive Process** to simulate **nonstationary covariance parameters**. Note that we have two parameters, *true_scale_log_scale* and *true_noise_log_scale*, who control the respective **predictive process variance** of the nonstationary marginal variance and noise variance.

```
# parents array for NNGP
NNarray = GpGp::find_ordered_nn(locs, 10)

# First make a NNGP
chol_precision =
  # computing coefficients
   GpGp::vecchia_Linv(covparms = c(1,2, 0), covfun_name = "matern15_isotropi
c",
                      locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
chol_precision = Matrix::sparseMatrix(
    x = chol_precision[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
    )
# keeping only first columns of the NNGP factor
PP_dropout = rep(0, nrow(locs)); PP_dropout[seq(100)] = 1
PP_scale = as.vector(Matrix::solve(chol_precision,  rnorm(nrow(locs)) * PP_dr
opout))
PP_noise = as.vector(Matrix::solve(chol_precision,  rnorm(nrow(locs)) * PP_dr
opout))
```

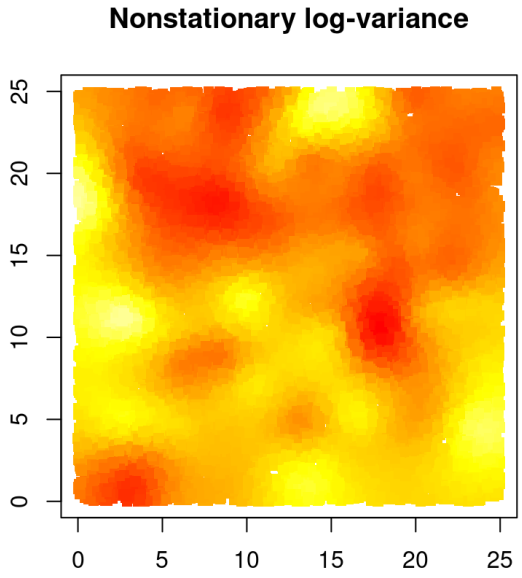
4.1.1.3 Simulating a nonstationary latent field

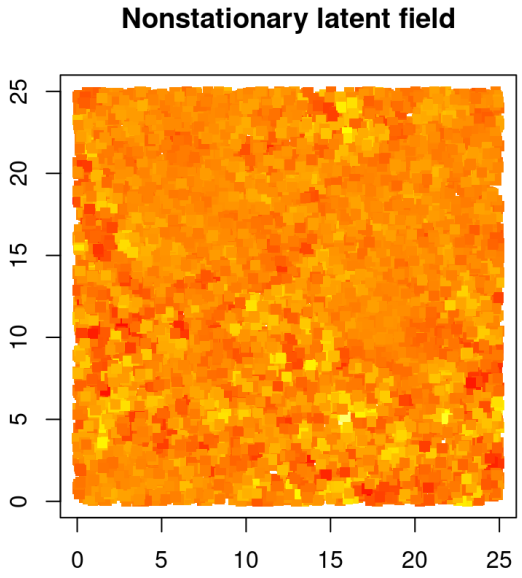
Let's now get a **nonstarionary latent field**. Note that here, we have a parameter called *true_scale_beta*, who is the **intercept for the latent process variance**, equivalent to the logarithm of the stationary latent process variance parameter. We have another parameter, *true_scale_log_scale*, who controls the **PP variance** of the variance.

```
true_scale_beta = 1
true_scale_log_scale = 0

# Sampling the nonstationary latent field
# First, sample a stationary latent field
chol_precision =
  # computing coefficients
   GpGp::vecchia_Linv(covparms = c(1,.2, 0), covfun_name = "matern15_isotropi
c",
                      locs = locs, NNarray = NNarray)
  # putting coefficients in precision Cholesly
chol_precision = Matrix::sparseMatrix(
    x = chol_precision[!is.na(NNarray)],
    i = row(NNarray)[!is.na(NNarray)],
    j = (NNarray)[!is.na(NNarray)],
    triangular = T
    )
# sampling the stationary process
seed_vector = rnorm(nrow(locs))
stat_latent_field = as.vector(Matrix::solve(chol_precision, seed_vector))
# nonstationary scale
true_log_scale = (PP_scale * exp(true_scale_log_scale * .5) + true_scale_bet
a)

# Multiplying the stat latent field to get a nonstat latent field
nonstat_latent_field = exp(.5*true_log_scale) * stat_latent_field
```

Let's have a look at the latent log-variance.

```
par(mfrow=  c(1,2))
Bidart::plot_pointillist_painting(locs,PP_scale, main = "Nonstationary log-va
riance")
Bidart::plot_pointillist_painting(locs,nonstat_latent_field, main = "Nonstati
onary latent field")
```





```
par(mfrow=  c(1,1))
```

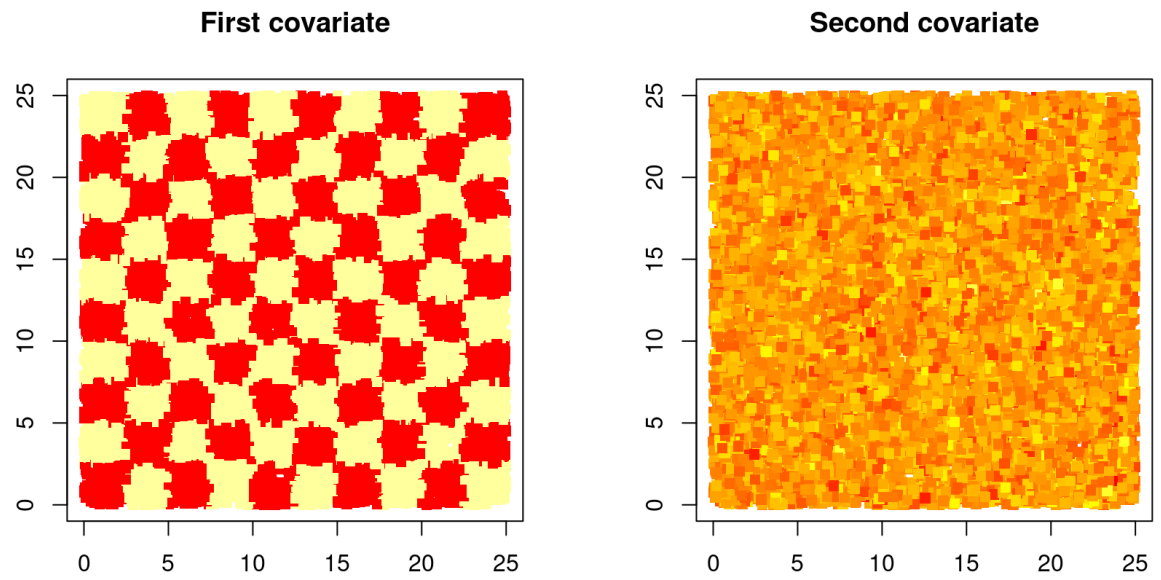
We can see that the field has more yellow (high) and red (low) values in the places where the nonstationary variance is high.

4.1.1.4 Simulating covariates

We can now create some **covariates**:

- one who is spatially shaped like **a chess board**.
- one who is just a **white noise**.

```
# chess board
X_1 = ((locs %/% 2.5)%*%c(1,1))%%2 ==1
# white noise
X_2 = rnorm(nrow(locs))
# putting them in data.frame
X= as.data.frame(cbind(X_1, X_2))
colnames(X) = c("chess", "white_noise")

par(mfrow=  c(1,2))
Bidart::plot_pointillist_painting(locs,X[,1], main = "First covariate")
Bidart::plot_pointillist_painting(locs,X[,2], main = "Second covariate")
```

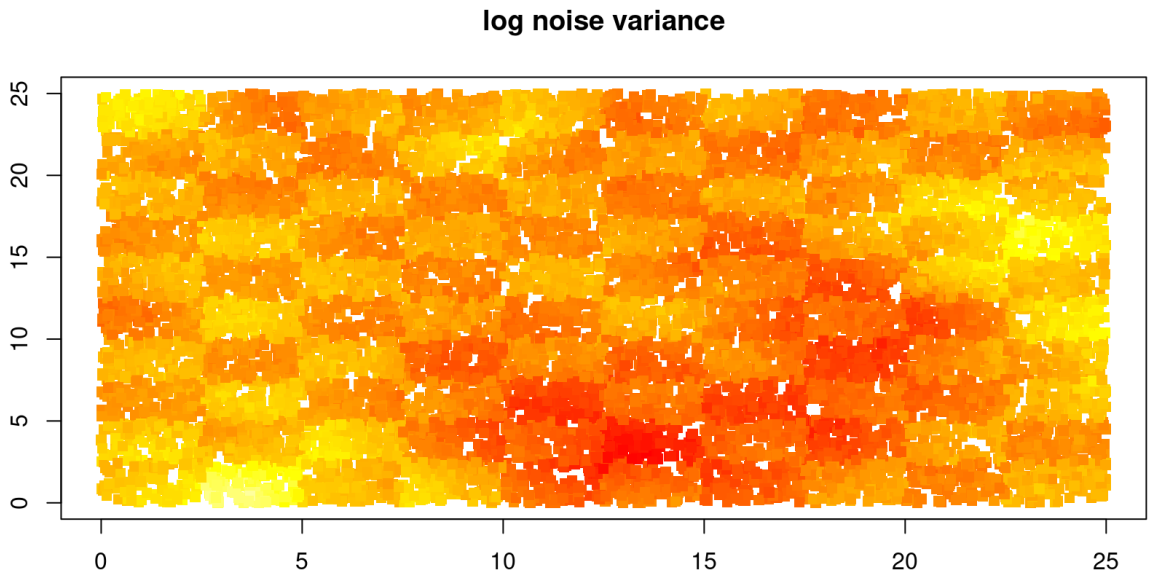



```
par(mfrow=  c(1,1))
```

4.1.1.5 Simulating a nonstationary noise

We now create a **nonstationary noise**, depending on an **PP** and **covariates as well**, including the **intercept**. Note that here, **the covariates do not need to be spatially smooth**.

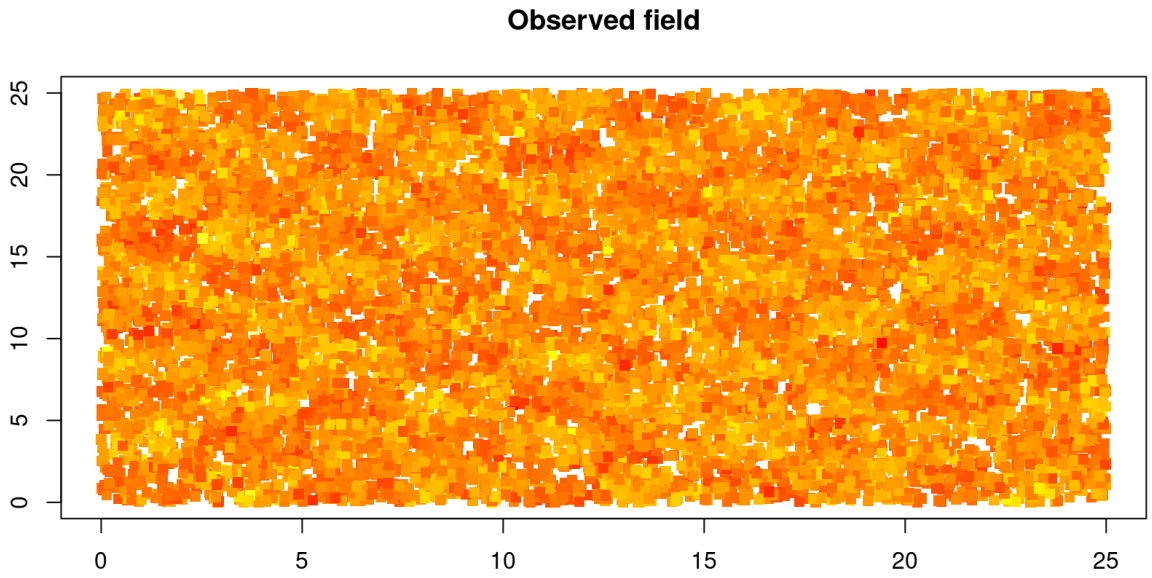
```
true_noise_beta = c(1,1,.2)
noise_var =
  PP_noise + cbind(1, as.matrix(X)) %*% true_noise_beta

Bidart::plot_pointillist_painting(locs, noise_var, main = "log noise varianc
e")
```



4.1.1.6 Putting things together

Eventually, we can **combine** all those components to get an **observed signal**. Note that we have regression coefficients for the **fixed effects** as well.

```
# regression coefficients for the fixed effects
fixed_beta = c(1,5,3)

observed_field =
  nonstat_latent_field + # latent field
  as.matrix(cbind(1, X)) %*% fixed_beta + # fixed effects
  noise_var * rnorm(nrow((locs))) # noise

Bidart::plot_pointillist_painting(locs, observed_field, main = "Observed fiel
d")
```



It's difficult to tell the ingredients with naked eye !

4.1.2 The Predictive Processes

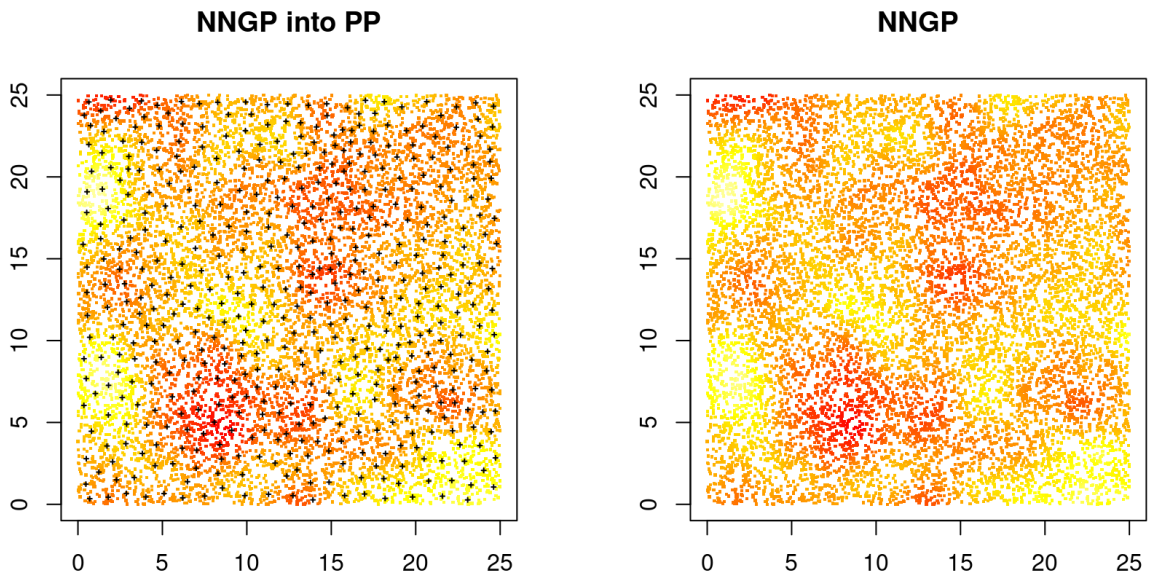
Let's start with the **Predictive Processes** described previously. If you want to use PPs, this is the first step of the initialization. If you do not, skip this paragraph. Those PPs are obtained with the fuction *get_PP*. Its arguments are:

- *observed_locs*, the sites where the observations are done.
- *matern_range*, the Matern range of the PP covariance (the Matern smoothness is always \(1.5\)).
- *n_PP*, the number of knots. The knots are automatically placed with a k-means algorithm.
- *m*, the number of parents in the NNGP used to construct the PP. It is automatically set, so don't worry about that.

Remember that **the PP spatial range must be much larger than the latent process range** (except, arguably, if you are only modelling the noise variance). At those ranges, it is likely that the **PP range does not matter much**.

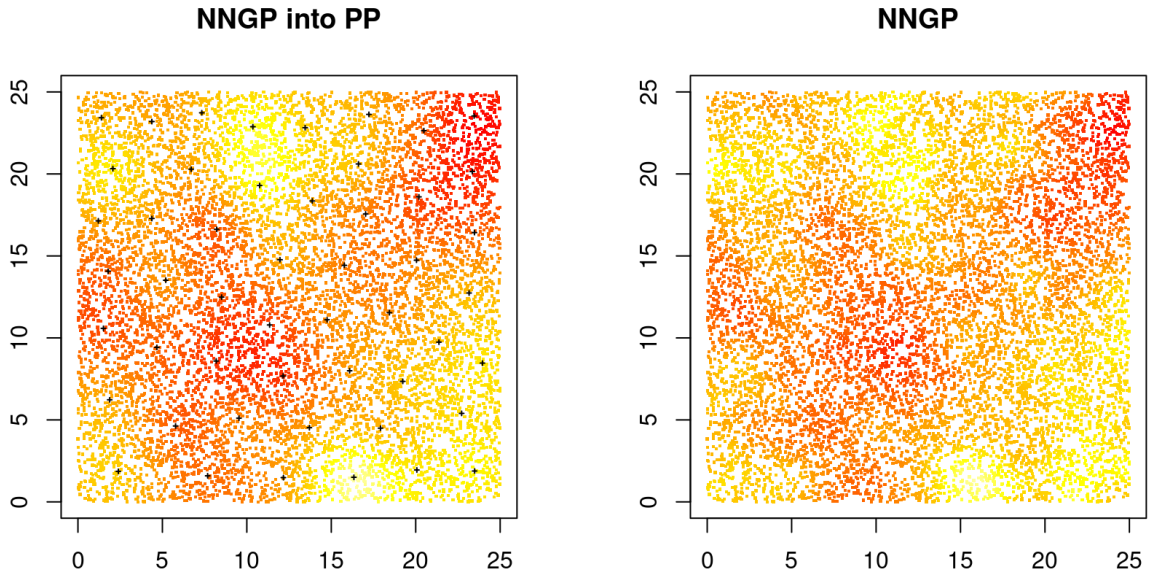
Don't put too many knots. All you need is to have a PP **close enough to the NNGP**. The function *compare_PP_NNGP* comes in handy to assess the number of knots. For example, in the following, we can see that **the number of knots is overkill** and may cause trouble.

```
PP = Bidart::get_PP(
  observed_locs = locs[seq(10000),], # spatial sites
  matern_range = 2,
  n_PP = 500, # number of knots
  m = 15 # number of NNGP parents
)
# comparison between PP and NNGP
Bidart::compare_PP_NNGP(PP)
```



For example, in the following, we can see that **the number of knots is reasonable**.

```
PP = Bidart::get_PP(
  observed_locs = locs[seq(10000),], # spatial sites
  matern_range = 2,
  n_PP = 50, # number of knots
  m = 15 # number of NNGP parents
)
# comparison between PP and NNGP
Bidart::compare_PP_NNGP(PP)
```



4.1.3 The mcmc_nngp_list

Now that you have a suitable Predictive Process, you need to **initialize the mcmc chains**. This is done using the *mcmc_nngp_initialize_nonstationary* function.

Its **essential arguments** are:

- *observed_locs*, a **matrix** of **spatial coordinates**.
- *observed_field*, a **vector** for the **interest variable**.

Rememember the Russian Dolls : if you provide only those arguments, you will get a **stationary geostatistical model**. Each row of *observed_locs* must correspond to an observation of *observed_field*.

A **matching of the coordinates with the unique coordinates** is done internally, so a little **rounding of the spatial locations** will help the model if the data is very **dense in the space**. If you feel that this is cheating, remember that **INLA also reduces the spatial dimension** (https://becarioprecario.bitbucket.io/spde-gitbook/ch-intro.html#sec:mesh) of the data set through its **tent function triangulation**. The author of this vignette had a conversation with a great statistician, who told him : “You don’t need to do statistics like in the 1990’s and model every observation. The way to do statistics today, is to make a structure, and the observations just land on the top of the structure”.

An important family of arguments are the **covariates** explaining the **range**, the process and noise **variance**, and the fixed effects of thee **response variable**. We have:

- *scale_X*, the **data.frame** of covariates used to model the **process marginal variance**.
- *noise_X*, the **data.frame** of covariates used to model the **noise variance**.
- *range_X*, the **data.frame** of covariates used to model the **process range**.
- *X*, the **data.frame** of covariates used to model the **response variable**.

There is **no need to put an intercept**, which is added automatically. An important point is that **scale_X and range_X must not vary within the same spatial location**. Those arguments are obviously important for the modeler, but can be left empty, in which case there will only be an intercept, and the model will be stationary.

Another important family of arguments are the **Predictive Processes**:

- *PP* is the Predictive Process obtained through the process described previously.
- *noise_PP* is a Boolean indicating if the PPs should be used to model the variance of the Gaussian noise.
- *scale_PP* is a Boolean indicating if the PPs should be used to model the marginal variance of the latent process.
- *range_PP* is a Boolean indicating if the PPs should be used to model the range of the latent process.

Here comes **a portmanteau of important parameters**.

- *nu* is the Matérn smoothness, only 0.5 and 1.5 are accepted. It is equal to 1.5 by default. This parameter often impacts **prediction**, larger smoothness usually avoiding over-fitting.
- *anisotropic* is a **Boolean** indicating if the covariance is anisotropic. It is false by default.
- *sphere* is a **Boolean** indicating if the locations are longitude-latitude coordinates. It is false by default. You can model on the **pole-less sphere**.
- *m* is the **integer** number of parents in the NNGP, 10 or 12 should be well. It is 10 by default.

Finally, we have **a portmanteau of un-important parameters**.

- *n_chains* is the number of MCMC chains. It is by default equal to 2. Due to the burn-in and the memory overhead, it is not advised to have many chains.

- *seed*, the random seed. Automatically set to \(1\). This parameter may be relevant when some starting points cause crashes early in the run (this is rare).
- *noise_beta_mean* is the **matrix** mean of the Normal prior for the regression coefficients associated with *noise_X*. It is automatically set to \(0\).
- *scale_beta_mean* is the **matrix** mean of the Normal prior for the regression coefficients associated with *scale_X*. It is automatically set to \(0\).
- *range_beta_mean* is the **matrix** mean of the Normal prior for the regression coefficients associated with *range_X*. It is automatically set to \(0\). When *anisotropic* is *T*, a *matrix* with \(3\) columns must be provided.
- *noise_beta_precision* is a *vector* indicating the precision diagonal of the Normal prior for the regression coefficients associated with *noise_X*. It is automatically set to a **very vague precision**.
- *scale_beta_precision* is a *vector* indicating the precision diagonal of the Normal prior for the regression coefficients associated with *scale_X*. It is automatically set to **very vague precision**.
- *range_beta_precision* is the precision diagonal of the Normal prior for the regression coefficients associated with *range_X*. It is automatically set to **very vague precision**. When *isotropic* is *F*, it is a a *vector* like for the two other parameters. When *anisotropic* is *T*, a *matrix* with \(3\) columns must be provided.

Let's play with our data set and try several model settings.

We start with **the simplest model**.

```
# initialization with a complicated range model
mcmc_nngp_list = Bidart::mcmc_nngp_initialize_nonstationary(
  observed_locs = locs[seq(10000),],
  observed_field = c(observed_field)[seq(10000)]
)
```

```
## Setup done, 1.36931276321411 s elapsed
```

Lo and behold, a message indicates that the setup is done !

Now, let's move to the "true" model with **covariates** for the **fixed effects and the noise**, and **PP** for **the noise and the process variances**.

```
mcmc_nngp_list = Bidart::mcmc_nngp_initialize_nonstationary(
  observed_locs = locs[seq(10000),],
  observed_field = c(observed_field)[seq(10000)],
  PP = PP, # Predictive Process
  noise_PP = T, # Noise variance PP activated
  scale_PP = T, # Latent proccess variance PP activated
  X = X[seq(10000),], # covariates for the fixed effects
  noise_X = X[seq(10000),] # covariates for the noise variance
)
```

```
## noise_log_scale_prior was automatically set to an uniform on (-6, 2)
```

```
## scale_log_scale_prior was automatically set to an uniform on (-6, 2)
```

```
## Setup done, 1.54299378395081 s elapsed
```

The **priors** for the **PP log-variance** are set automatically, resulting in a message. The priors of range_beta, noise_beta, and scale_beta are also set automatically.

If we want, we can **explicitly** indicate the prior mean and precision of those parameters:

```
mcmc_nngp_list = Bidart::mcmc_nngp_initialize_nonstationary(
  observed_locs = locs[seq(10000),],
  observed_field = c(observed_field)[seq(10000)],
  PP = PP,
  noise_PP = T,
  scale_PP = T,
  X = X[seq(10000),],
  noise_X = X[seq(10000),],
  noise_log_scale_prior = matrix(c(-6, 2)),
  noise_beta_precision = matrix(rep(.001, 3)),
  noise_beta_mean = matrix(rep(0, 3))
)
```

```
## scale_log_scale_prior was automatically set to an uniform on (-6, 2)
```

```
## Setup done, 1.49275922775269 s elapsed
```

Now, you see that only the message for *scale_log_scale_prior* appears.

4.1.4 Model specification recap.

Go for **noise heteroskedasticity**, it is computationally **cheap**, and almost always worth it in terms of **model improvement**. Don't go for nonstationary marginal variance and range at the same time because of identification problems, except if you have a really good reason to. Nonstationary marginal variance is computationally cheaper than range and seems to bring similar improvements, even though sometimes one of the two is slightly better. Keep in mind that anisotropy costs much more, but may however provide useful insight and some prediction improvement. Keep the PP range much higher what you expect the process range to be in order to avoid identification problems. Like we said before, **don't worry too much about PP spatial range as long as it is high** since there is a mis-identification between range and variance for high spatial ranges.

4.2 Launching the run

Once we have a *mcmc_nngp_list*, we want to **produce MCMC samples**. To do that, there are **two options**: the *mcmc_nngp_run_nonstationary_socket* or the *mcmc_nngp_run_nonstationary_nonpar* functions are used. The former uses **socket parallelization**. The later is **not parallel**, which is usually slower, but reduces the overhead.

The important point is that this function **takes a *mcmc_nngp_list*** generated previously and **gives back a *mcmc_nngp_list* with more MCMC samples**, which **replaces the previous *mcmc_nngp_list*.** Note that this gives a lot of **safety and flexibility**: you can start your run, save it, and finish it later. If you have to shut down for one reason or another, your run is not lost as long as you took care of saving it at the checkpoints.

The most important argument is obviously *mcmc_nngp_list*.

Then, come arguments related with the **MCMC**. They will mostly **not affect the behavior of the MCMC chains**. Examples are given in the next section.

- *seed* is the simulation seed. It is set to 1 by default.
- *burn_in*, a **number** between \(0\) and \(1\), is the proportion of discarded iterations when computing the MCMC diagnostics.
- *starting_proportion*, a **number** between \(0\) and \(1\), is the starting proportion to compute the Gelman-Rubin-Brooks curves.
- *thinning*, a **number** between \(0\) and \(1\), is the proportion of discarded iterations because of thinning. This will not make the MCMC algorithm any better, and is just useful to cut corners with the RAM (https://mc-stan.org/docs/2_18/reference-manual/effective-sample-size-section.html#thinning-samples).
- *plot_diags* is a **Boolean** indicating whether the diagnostic plots should be plotted.
- *plot_PSRF_fields* is a **Boolean** indicating whether the Gelman-Rubin-Brooks should be plotted for the latent field. It takes ages, so it is not a good idea to tick this option.

Eventually, come arguments related to **computation**.

- *n_cores* is an **integer** indicating the number of sockets for *mcmc_nngp_run_nonstationary_socket*.
- *num_threads_per_chain* is an **integer** indicating the number of Open Multi-Process used within each process.
- *lib.loc* is a **string** indicating the address of the libraries if they are not installed in the standard address. This is useful for clusters where you don't necessarily have the right to install shared packages.
- *debug_outfile* is a **string** creating a outfile for debugging. That's for developing.

Let's **run the MCMC chain** for a short while.

```
for(i in seq(12))mcmc_nngp_list =
  Bidart::mcmc_nngp_run_nonstationary_socket(
    mcmc_nngp_list = mcmc_nngp_list, # the list of states, data, Vecchia appr
oximation, etc
    burn_in = .5, # MCMC burn-in
    thinning = .1, # MCMC thinning
    n_cores = 2, # Parallelization over the chains
    num_threads_per_chain = parallel::detectCores()/2, # Parallellization wit
hin each chain
    seed = 1, # MC seed
    plot_diags = F, # MCMC diagnostics in the plot window
    plot_PSRF_fields = F, # PSRF of the latent fields - very costly
    lib.loc = NULL # not needed on my laptop, useful on SLURM cluster
    )
```

There is noting here because I don't want to make the vignette unreadable, but **Gelman-Rubin-Brools and Effective Sample Size diagnostics plot automatically**.

4.3 MCMC cuisine.

Our method is fit using Markov Chain Monte-Carlo (https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo) (MCMC).

4.3.1 The problem with MCMC.

Remember that in MCMC, the parameter space is explored by a Markov chain, who **jumps** from state to state like a **flea**. After some time, the chain leaves its starting point and starts wallowing in the zone where the model parameters are likely to be. This is what is called **mixing**. The Ergodic Theorems of MCMC tell us that averaging the states of the chain will, after many iterations, give us good estimates of our parameters. But what does *many* mean ? In particular:

- how do we know that the chain has reached the **interest region** ?
- how do we know that we have **enough samples**, given the fact that there is **auto-correlation** ?

4.3.2 Trace plots.

The first option is to **plot the chain of parameters** and interpret it. If the chains **wallow** around in a region, it is a **good** sign. However, trace plots are **deceptive** and **not synthetic** (if you have 200 parameters, too bad, you have to assess 200 chains visually), even though it's always a good idea to have a look.

4.3.3 Gelman-Rubin-Brooks diagnostics.

A more rigorous approach is the famous **Gelman-Rubin-Brooks Potential Scale Reduction Factor** (https://mc-stan.org/docs/2_18/reference-manual/notation-for-samples-chains-and-draws.html#potential-scale-reduction). This method detects that the chains have **left their initial regions**. It is based on two or more **parallel Markov chains**. It performs an **analysis of variance** in order to determine if the chains have the same behavior. The idea is that if the chains start from over-dispersed initial states, they will have a similar behavior only when they all reach the interest region and start mixing. This statistic indicates a good mixing when it is **close to** 1.

4.3.4 Effective Sample Size.

Another way to tackle the problem is the **Effective Sample Size** (https://mc-stan.org/docs/2_18/reference-manual/effective-sample-size-section.html) accounts for this correlation. We sum Coda's ESS (https://rdrr.io/cran/coda/man/effectiveSize.html) over the chains. The **MCMC error** is proportional to the inverse of the **square root** of the ESS. Therefore, it is easy to get a rough estimate, but hard to get a tight estimate. In practice, it is difficult to get ESS in the order of thousands for the high-level parameters. As a consequence, complex integrals of the high-level parameters and histograms should be treated carefully. However, a rough integration over the high-level parameters is enough to provide a performance improvement. Also, some basic statistics such as credibility intervals were satisfying in simulated experiments.

4.3.5 Burn-in and Thinning.

The **initial states** of the chains are heavily influenced by the choice of the starting point. Those states introduce a bias in the estimates and worsen the convergence diagnostics. Therefore, they should be **discarded**, that's what is called a **burn-in**. A conservative choice is to throw away the **first half** of the iterations. However, the **trace plots** can help finding a smaller value that will allow to **keep more iterations**.

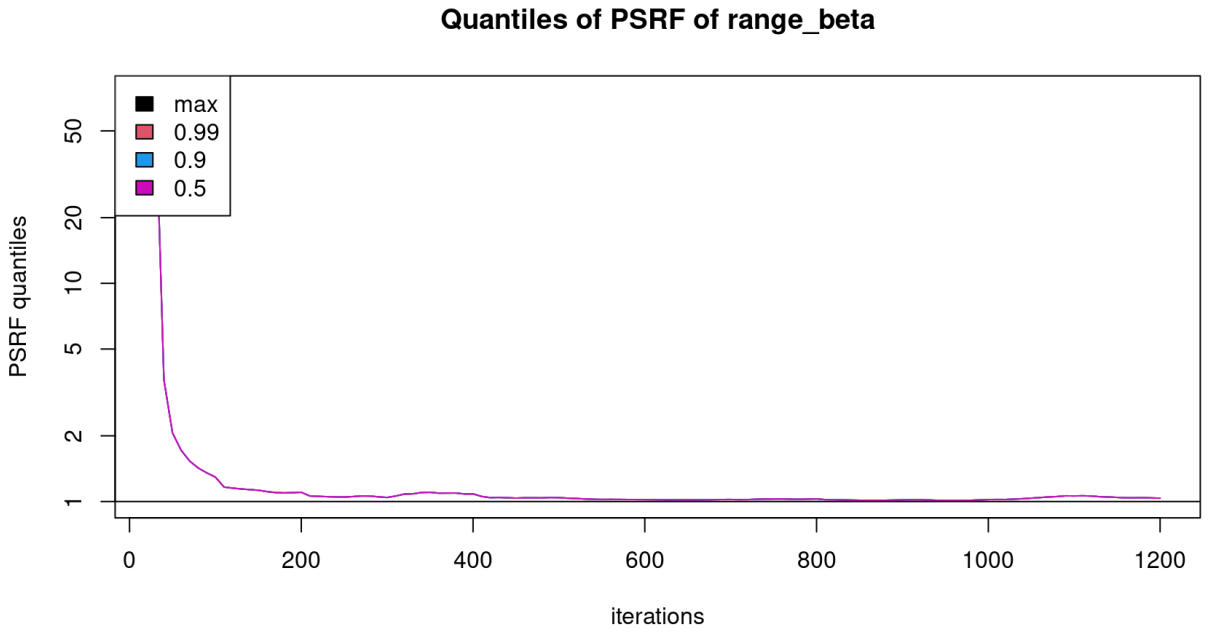
There also is **auto-correlation** in the samples, which makes **many iterations** necessary. In order to **save RAM**, it can be useful to keep only a fraction of the iterations, this is **thinning**. Again, this method will not improve the statistical effciency (https://mc-stan.org/docs/2_18/reference-manual/

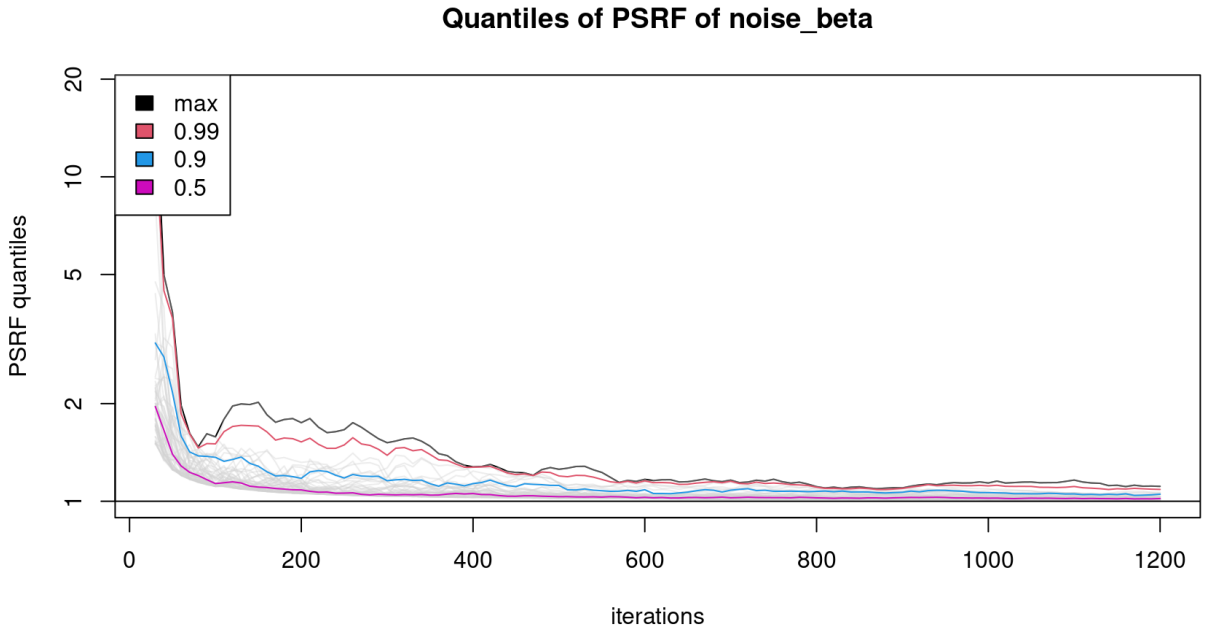
effective-sample-size-section.html#thinning-samples).

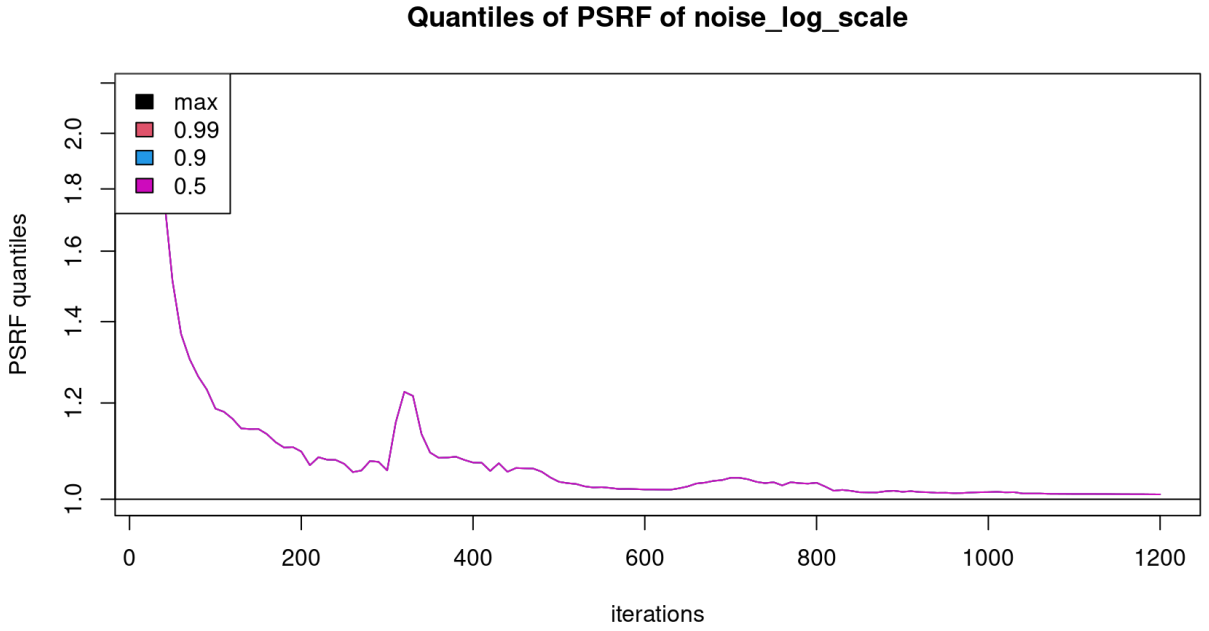
4.3.6 Diagnostics in practice.

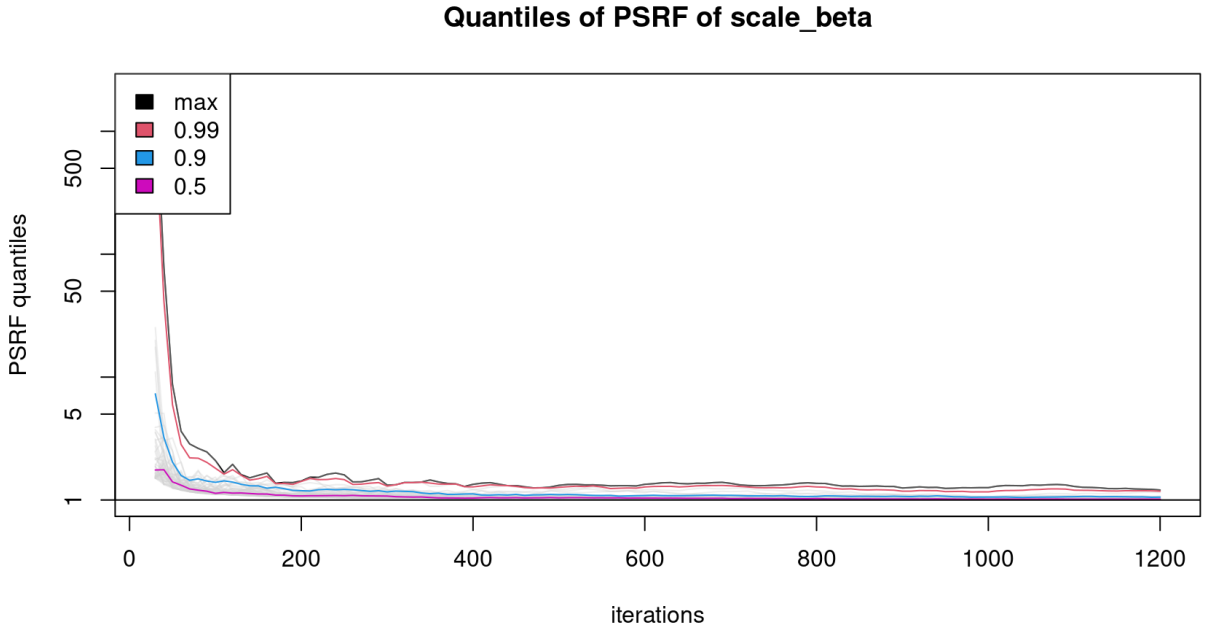
The diagnostics **pop up automatically** in the **Console** and the **Plots** windows of Rstudio (I have not tested any other UI). They can also be called on a *mcmc_nngp_list* after you have saved it on your laptop. The **trace plots** and **Gelman-Rubin-Brooks dianostics** can be called using *diagnostic_plots*. The arguments are the same who are used in the mcmc run function.

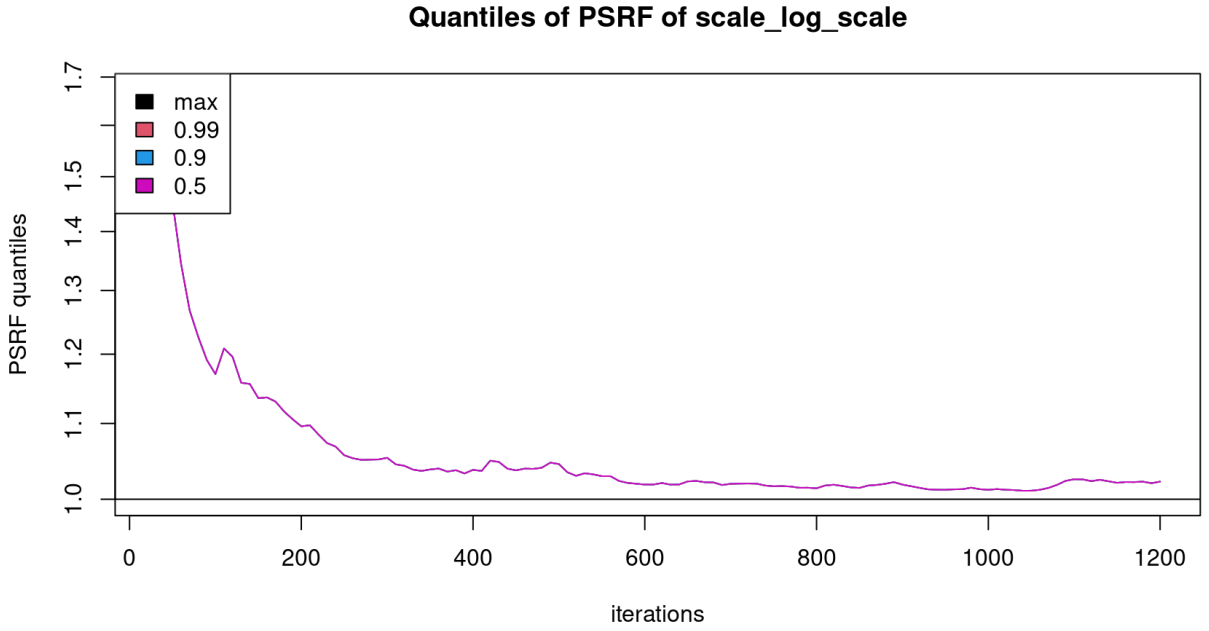
```
Bidart::diagnostic_plots(
  mcmc_nngp_list,
  burn_in = .1, # burn-in = 10%
  starting_proportion = .02 # plot chains from 1% of the iterations to 100%
  )
```

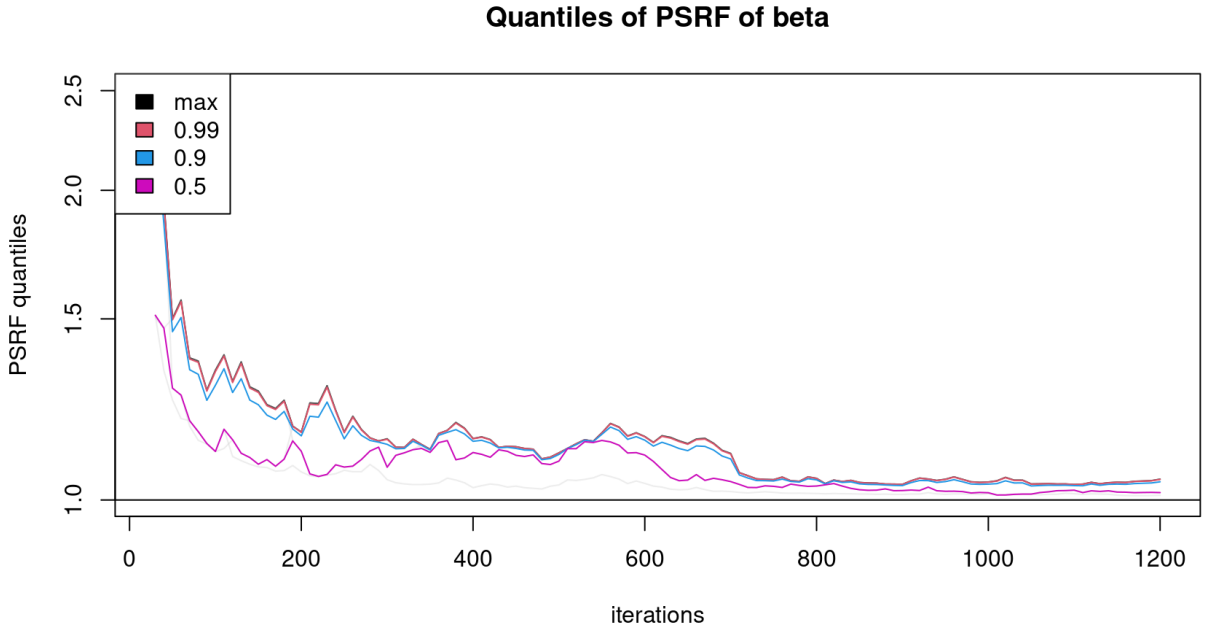


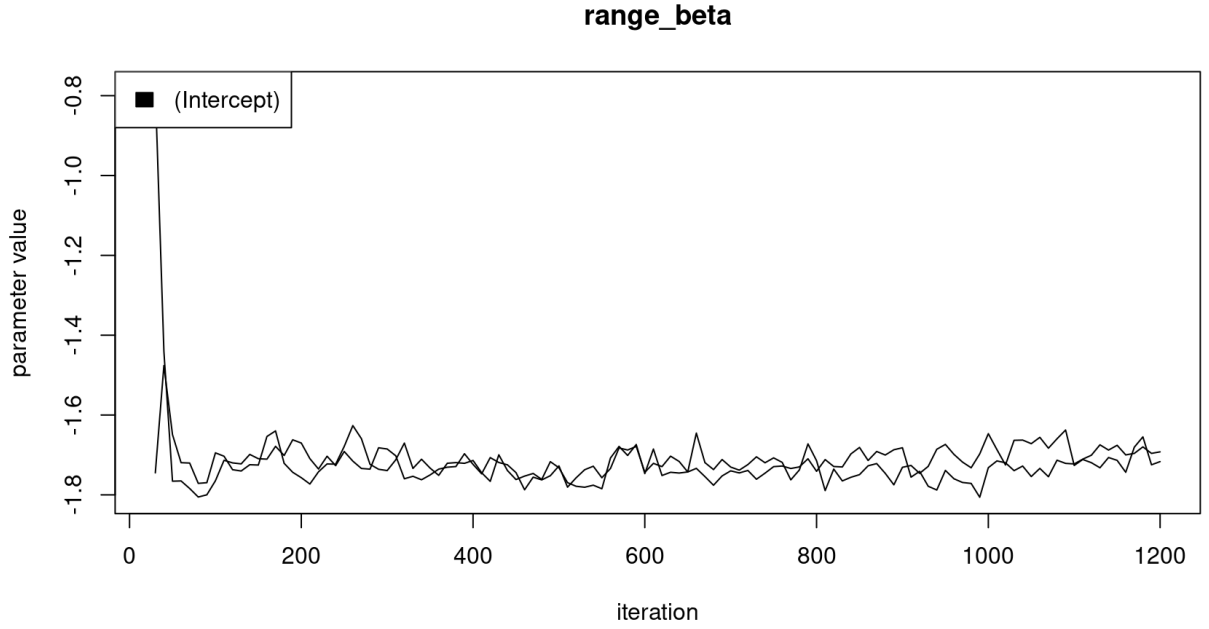


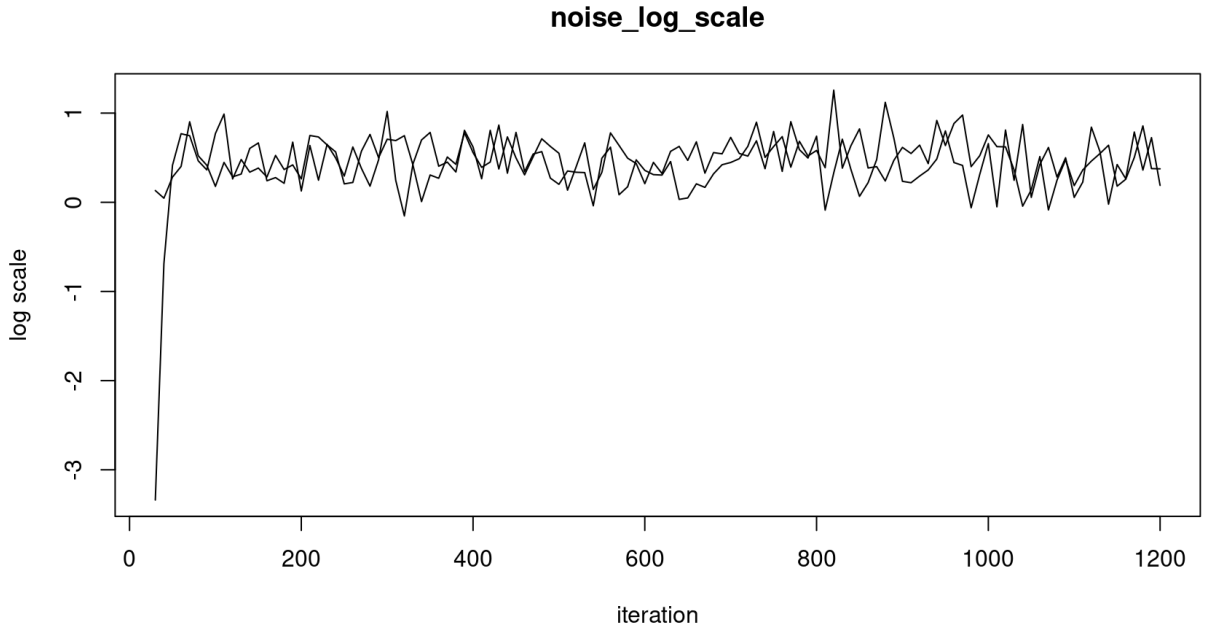

Quantiles of PSRF of noise_log_scale
max
0.99
0.9
0.5
PSRF quantiles
1.0
1.2
1.4
1.6
1.8
2.0
0
200
400
600
800
1000
1200
iterations

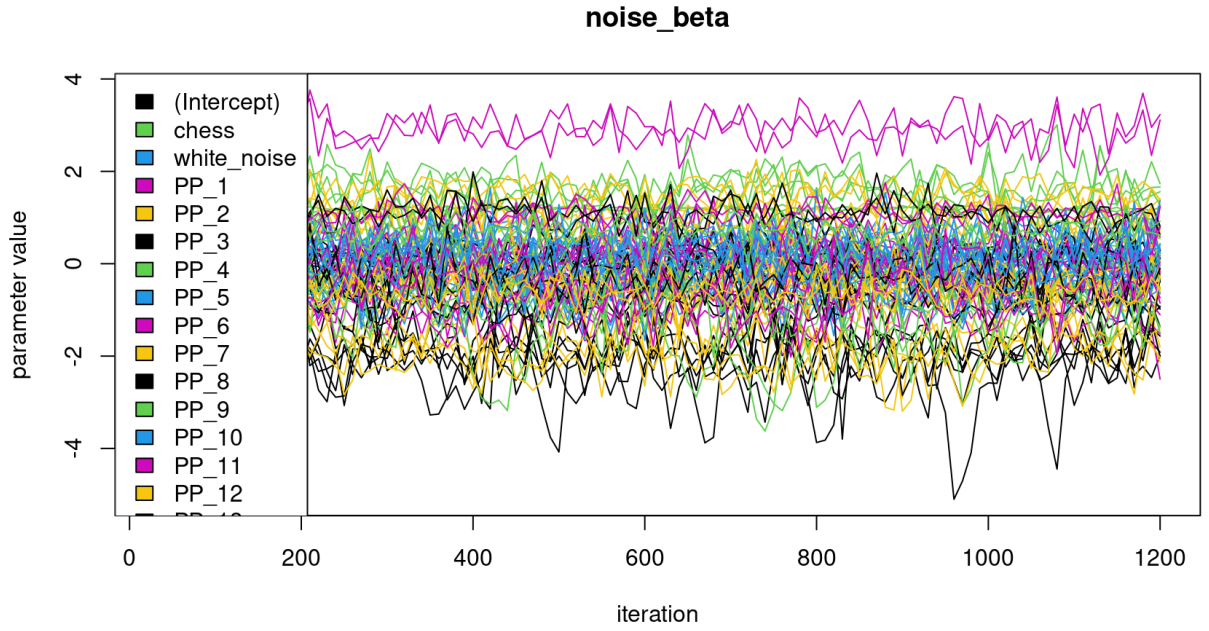

Quantiles of PSRF of scale_beta
max
0.99
0.9
0.5
PSRF quantiles
1
5
50
500
0
200
400
600
800
1000
1200
iterations

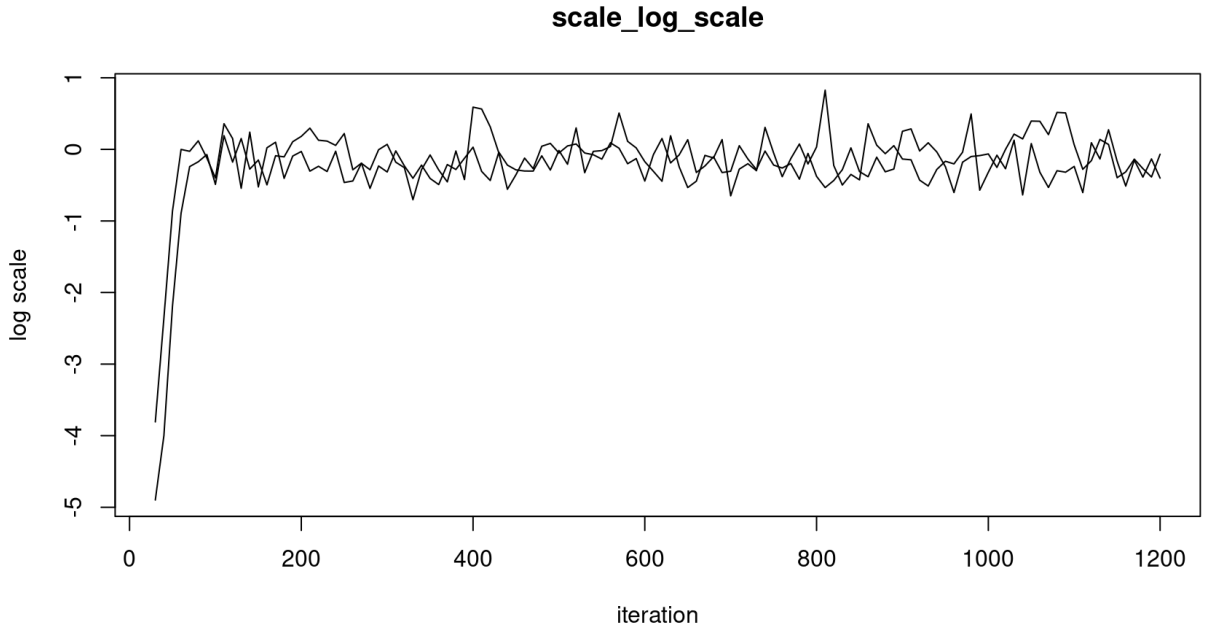

Quantiles of PSRF of scale_log_scale
max
0.99
0.9
0.5
PSRF quantiles
1.0
1.1
1.2
1.3
1.4
1.5
1.7
0
200
400
600
800
1000
1200
iterations

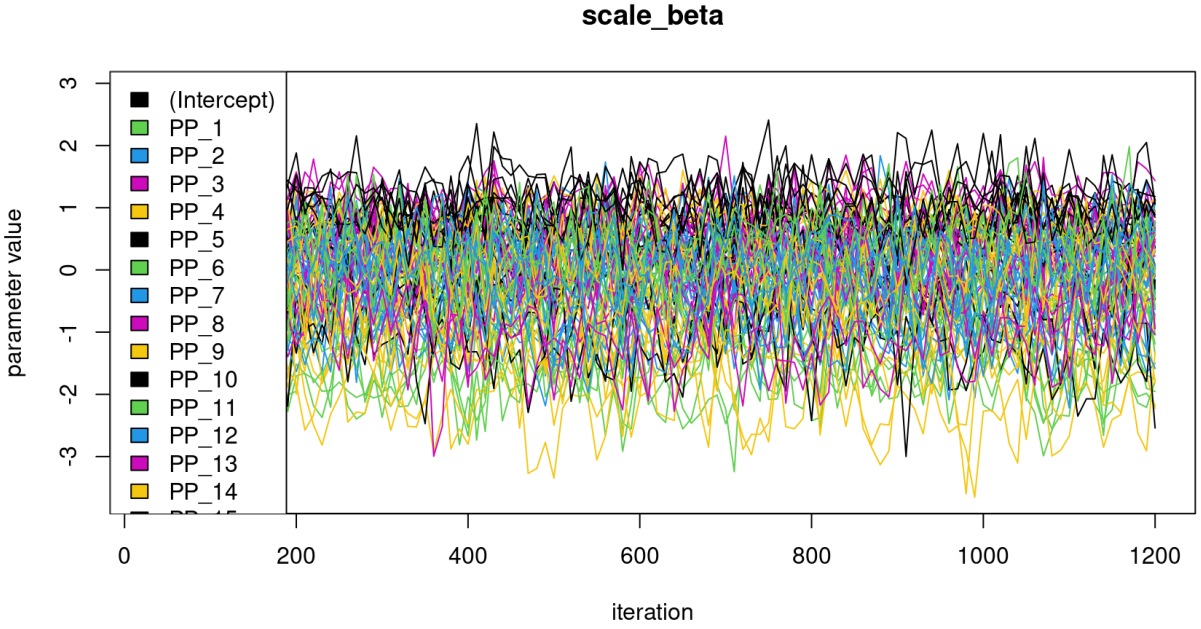

Quantiles of PSRF of beta
max
0.99
0.9
0.5
PSRF quantiles
1.0
1.5
2.0
2.5
0
200
400
600
800
1000
1200
iterations


range_beta
(Intercept)
parameter value
-0.8
-1.0
-1.2
-1.4
-1.6
-1.8
0
200
400
600
800
1000
1200
iteration


noise_log_scale
log scale
1
0
-1
-2
-3
0
200
400
600
800
1000
1200
iteration


noise_beta
parameter value
iteration
(Intercept)
chess
white_noise
PP_1
PP_2
PP_3
PP_4
PP_5
PP_6
PP_7
PP_8
PP_9
PP_10
PP_11
PP_12
4
2
0
-2
-4
0
200
400
600
800
1000
1200


scale_log_scale
log scale
iteration
1
0
-1
-2
-3
-4
-5
0
200
400
600
800
1000
1200



Let's first comment the **trace plots**, those who **come last**. First, remember that we have **two chains**, so the trace plots show a **family of parameters** for both chains at the same time. The meaning of those families of parameters is explained later. In *range_beta*, we have only one parameter since the range model is stationary, so we have two curves (one for each chain). Good news: the chains wallow, in the same place. We can also see that the starting values are off and the chain quickly goes to the zone of high density. Those first 100 or so iterations are those that we want to discard using the **burn-in**. On the other hand, *noise_beta* and *scale_beta* have many parameters since the model is nonstationary, so it is quite a mess. Howevever, the **parameters are separated by color**, and we can see that the curves go by color couple, which means that the parameters of the first chain mix in the same region as the parameters of the second chain.

The first plots are the **Gelman-Rubin-Brooks diagnostics**. Remember: the closer to 1, the better. They are **presented as curves** because they can reach a good value by chance, only to grow again in the next iterations. It is better to compute the diagnostics for several chain lengths. The diagnostics are presented using their **quantiles**, so that we have a **synthetic vision** of what is going on. We can see that all PSRFs go steadily to one, so we are satisfied.

The **Effective Sample Size** can be called using the function *ESS*.

```
Bidart::ESS(
  mcmc_nngp_list,
  burn_in = .1 # burn-in of 10%
)
```

```
## $range_beta
##                  [,1]
## (Intercept) 66.35497
##
## $scale_beta
##                   [,1]
## (Intercept) 269.17423
## PP_1        109.64259
## PP_2        114.76363
## PP_3        283.91279
## PP_4        263.74454
## PP_5        214.00000
## PP_6        235.63688
## PP_7        108.15995
## PP_8        214.00000
## PP_9        504.60049
## PP_10       294.24524
## PP_11       214.00000
## PP_12       312.27563
## PP_13       166.58966
## PP_14       186.11484
## PP_15       145.74852
## PP_16       429.44314
## PP_17       135.48484
## PP_18       187.22598
## PP_19       214.00000
## PP_20       399.47045
## PP_21       264.40791
## PP_22       267.52485
## PP_23       140.31354
## PP_24       152.82703
## PP_25       214.00000
## PP_26       222.88804
## PP_27       253.31849
## PP_28       199.48379
## PP_29        95.55615
## PP_30       416.69635
## PP_31       180.26011
## PP_32       214.00000
## PP_33       214.00000
## PP_34       133.14110
## PP_35        66.43040
## PP_36       208.39507
## PP_37        98.66812
## PP_38       214.00000
## PP_39       108.64647
## PP_40       214.67141
## PP_41       259.99505
## PP_42       274.63684
## PP_43       326.12805
## PP_44       214.00000
## PP_45       218.95799
## PP_46       304.05228
## PP_47        91.01793
```

```
## PP_48       214.00000
## PP_49       214.00000
## PP_50       144.35843
##
## $noise_beta
##                   [,1]
## (Intercept) 466.42328
## chess       191.50574
## white_noise 286.93209
## PP_1        256.06595
## PP_2        408.94634
## PP_3        214.00000
## PP_4        176.81180
## PP_5        214.00000
## PP_6        214.00000
## PP_7        324.97567
## PP_8        214.00000
## PP_9        297.56775
## PP_10       130.55534
## PP_11       256.83652
## PP_12       491.57676
## PP_13       214.00000
## PP_14       214.00000
## PP_15       311.55339
## PP_16       689.54198
## PP_17       214.00000
## PP_18       214.00000
## PP_19       183.29382
## PP_20       214.00000
## PP_21       258.87306
## PP_22       248.24782
## PP_23       111.09074
## PP_24       380.00639
## PP_25       269.69806
## PP_26       568.26393
## PP_27       164.74080
## PP_28       214.00000
## PP_29       319.09132
## PP_30       214.00000
## PP_31       214.00000
## PP_32       214.00000
## PP_33       214.00000
## PP_34        81.53930
## PP_35        34.14707
## PP_36       262.03486
## PP_37       501.32692
## PP_38       149.70740
## PP_39       247.55243
## PP_40       291.29647
## PP_41       127.49333
## PP_42       214.00000
## PP_43       214.00000
## PP_44       276.45746
## PP_45       214.00000
## PP_46       259.89086
```

```
## PP_47       291.78374
## PP_48       260.97474
## PP_49       214.00000
## PP_50       247.99938
##
## $beta
##                 [,1]
## (Intercept) 378.4307
## chess       307.6496
## white_noise 214.0000
##
## $noise_log_scale
##          [,1]
## [1,] 298.9371
##
## $scale_log_scale
##          [,1]
## [1,] 152.2866
```

Going over the ESS parameters, we can see that all values are fairly high.

```
print(paste(
  "The worst ESS is",
  min(unlist(Bidart::ESS(
    mcmc_nngp_list,
    burn_in = .1 # burn-in of 10%
  )))
))
```

```
## [1] "The worst ESS is 34.1470745691746"
```

4.4 Interpretation of the parameters.

What do all those parameters correspond to ?

4.4.1 Regression coefficients.

The regression coefficients - *range_beta*, *noise_beta*, and *scale_beta* - are the parameters who **link the process range and variance, and the noise variance, with covariates, including Predictive Process basis functions**. *range_beta* corresponds to \(\beta_A\) or \(\beta_\alpha\) in the range model, *scale_beta* corresponds to \(\beta_\sigma\) in the latent variance model, and *noise_beta* corresponds to \(\beta_\tau\) in the noise variance model.

All those parameters have **as many components as there are covariates and PPs**. For example, in the previous example, we can see that *range_beta* has only one component since the range is stationary. On the other hand, *noise_beta* has many components because of the PP basis functions.

The fact that PP parameters are treated as regression coefficients may feel unsettling and was discussed previously.

4.4.2 Log-scale parameters.

The log-scale parameters control the **variance of the PP** for the range, scale, and noise. Note that

here, there is no PP for the range, since the range is stationary. We do have PPs for the scale and the noise.

4.4.3 Diagnosing over-modelling.

Those log-range parameters are especially useful to **avoid over-fitting** and diagnose **over-modelling**, that is when the **model is too complex for the data**.

Let's do an example. Here, we keep the same model, but we change the data. There is no more nonstationarity of the noise and latent field variance. The chain is run for a few hundred iterations.

```
observed_field_stat =
  stat_latent_field + # latent field
  as.matrix(cbind(1, X)) %*% fixed_beta + # fixed effects
  rnorm(nrow((locs))) # noise

mcmc_nngp_list_over_modeling = Bidart::mcmc_nngp_initialize_nonstationary(
  observed_locs = locs[seq(10000),],
  observed_field = c(observed_field_stat)[seq(10000)],
  PP = PP, # Predictive Process
  noise_PP = T, # Noise variance PP activated
  scale_PP = T, # Latent proccess variance PP activated
  X = X[seq(10000),], # covariates for the fixed effects
  noise_X = X[seq(10000),] # covariates for the noise variance
)
```

```
## noise_log_scale_prior was automatically set to an uniform on (-6, 2)
```

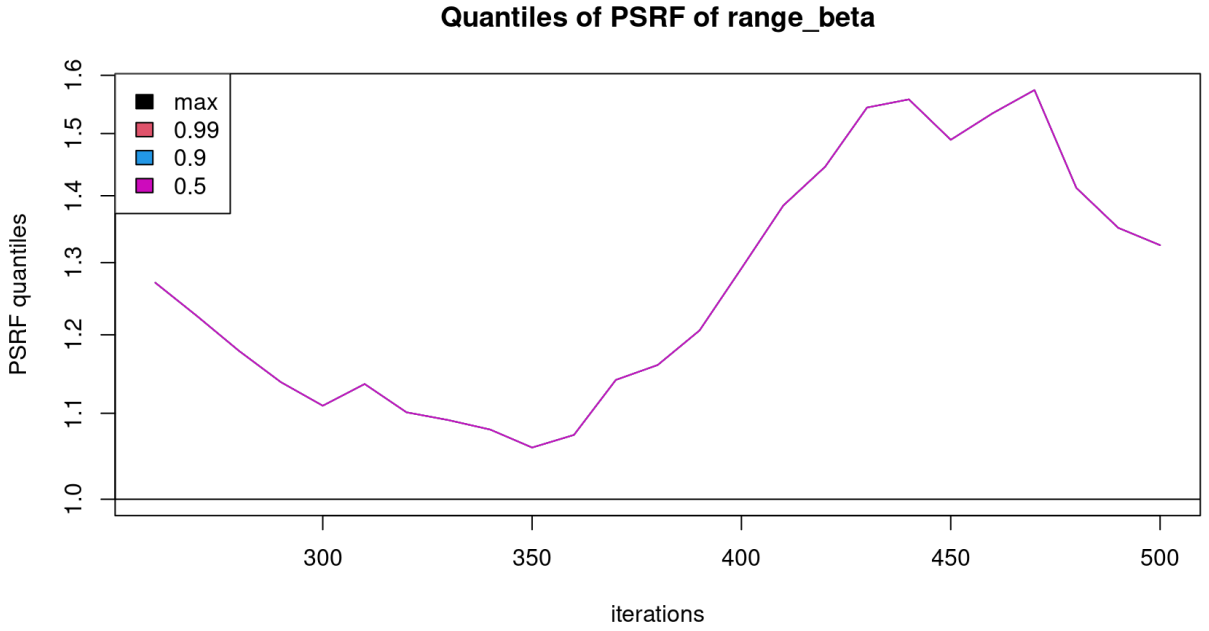
```
## scale_log_scale_prior was automatically set to an uniform on (-6, 2)
```

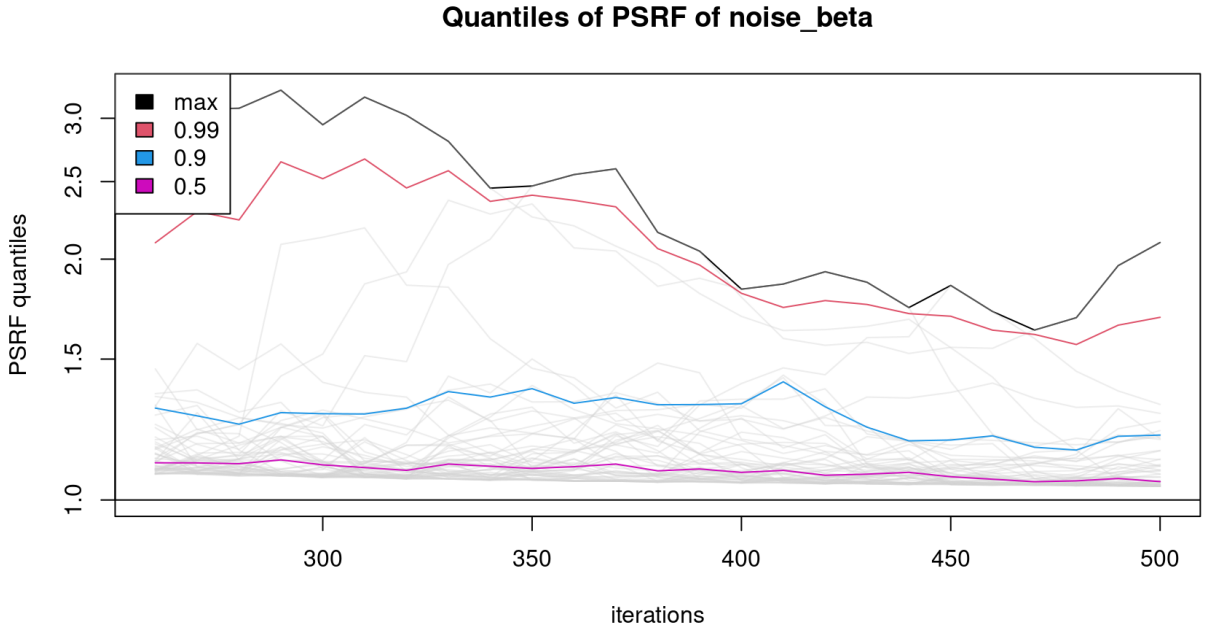
```
## Setup done, 1.2315833568573 s elapsed
```

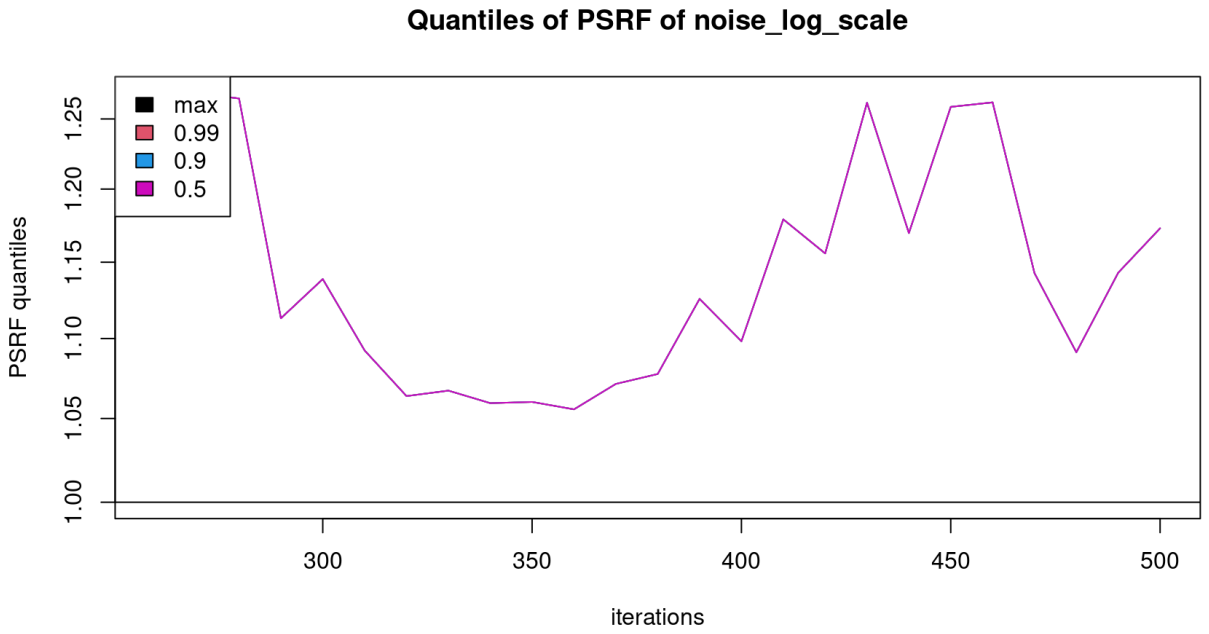
```
for(i in seq(5))mcmc_nngp_list_over_modeling =
  Bidart::mcmc_nngp_run_nonstationary_socket(
    mcmc_nngp_list = mcmc_nngp_list_over_modeling, # the list of states, data
, Vecchia approximation, etc
    burn_in = .5, # MCMC burn-in
    thinning = .1, # MCMC thinning
    n_cores = 2, # Parallelization over the chains
    num_threads_per_chain = parallel::detectCores()/2, # Parallellization wit
hin each chain
    seed = 1, # MC seed
    plot_diags = F, # MCMC diagnostics in the plot window
    plot_PSRF_fields = F, # PSRF of the latent fields - very costly
    lib.loc = NULL # not needed on my laptop, useful on SLURM cluster
    )
```

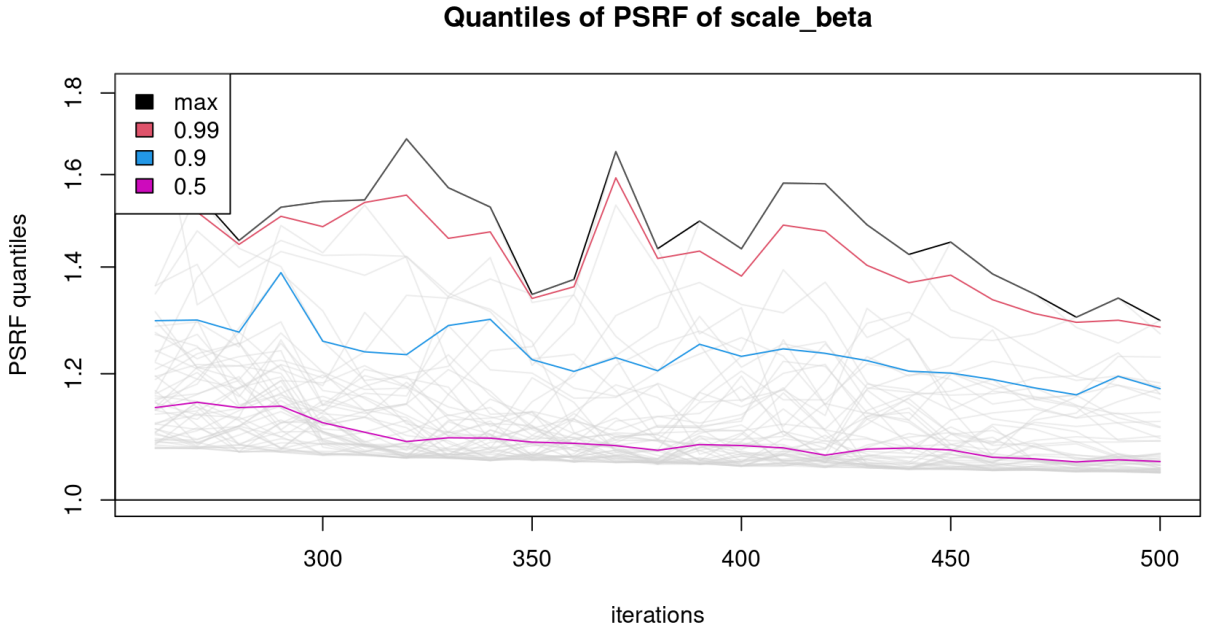
Now, let's plot the chain diagnostics:

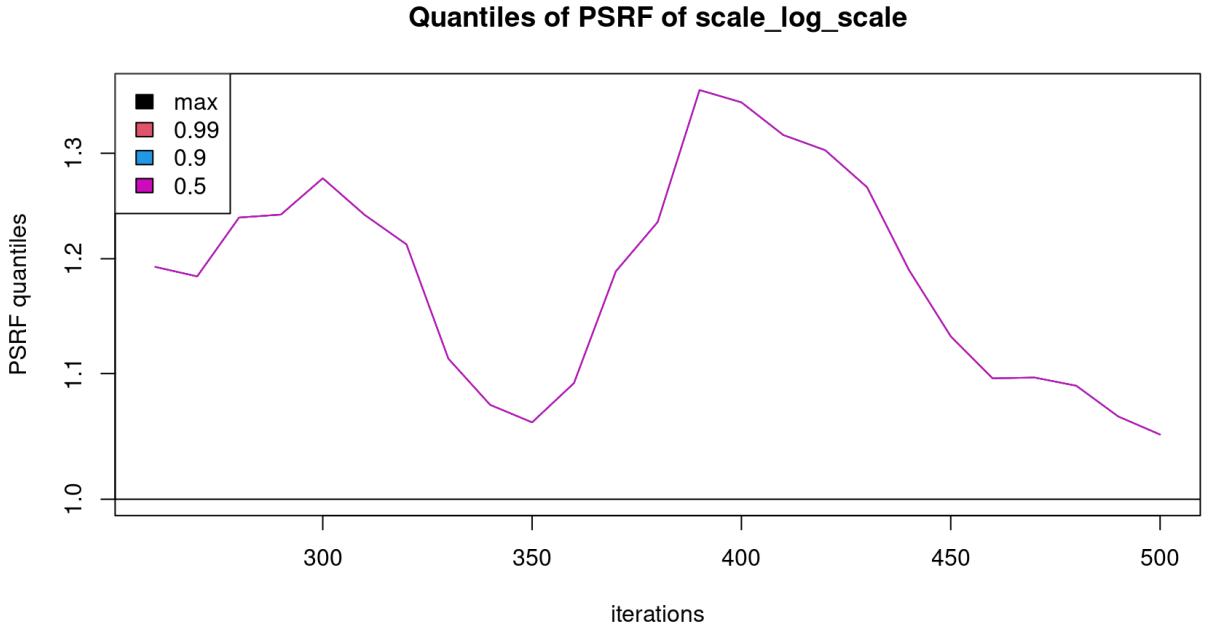
```
Bidart::diagnostic_plots(mcmc_nngp_list_over_modeling)
```

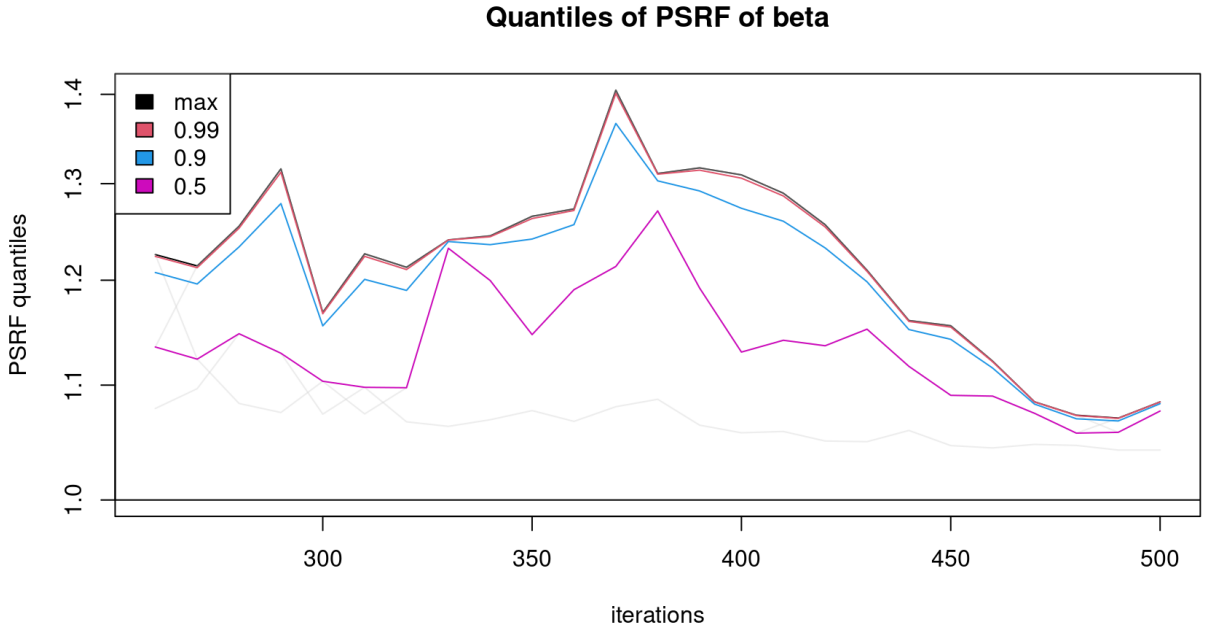

Quantiles of PSRF of range_beta
max
0.99
0.9
0.5
PSRF quantiles
1.0
1.1
1.2
1.3
1.4
1.5
1.6
300
350
400
450
500
iterations

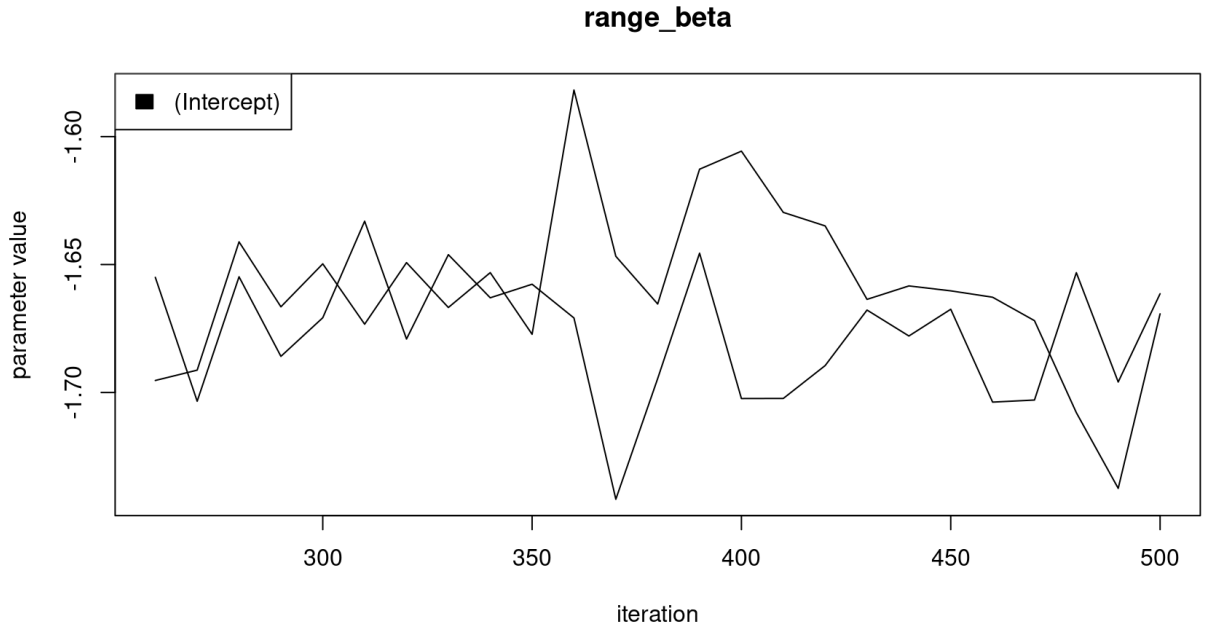

Quantiles of PSRF of noise_beta
max
0.99
0.9
0.5
PSRF quantiles
1.0
1.5
2.0
2.5
3.0
300
350
400
450
500
iterations

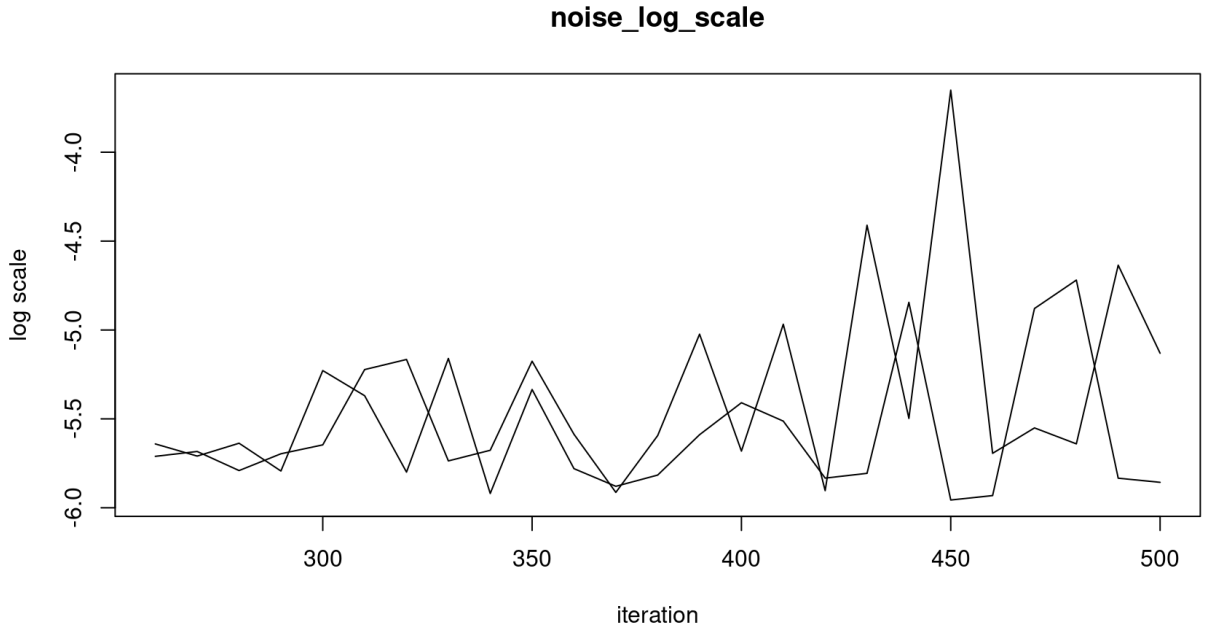

Quantiles of PSRF of noise_log_scale
max
0.99
0.9
0.5
PSRF quantiles
1.00
1.05
1.10
1.15
1.20
1.25
300
350
400
450
500
iterations

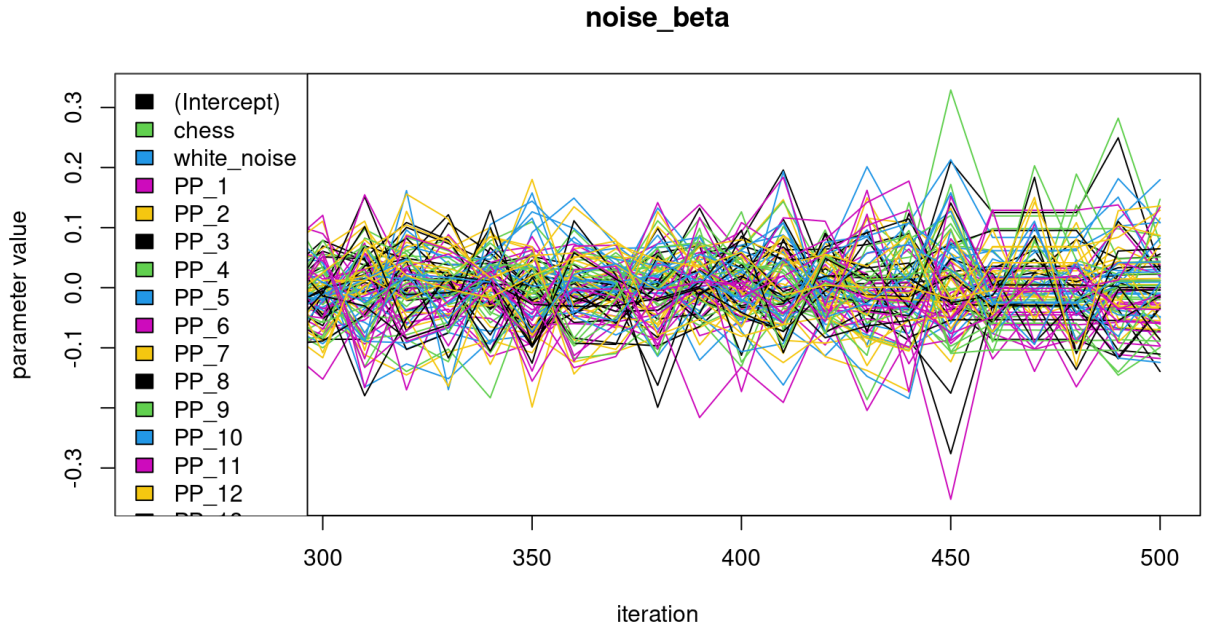

Quantiles of PSRF of scale_beta
max
0.99
0.9
0.5
PSRF quantiles
1.0
1.2
1.4
1.6
1.8
300
350
400
450
500
iterations

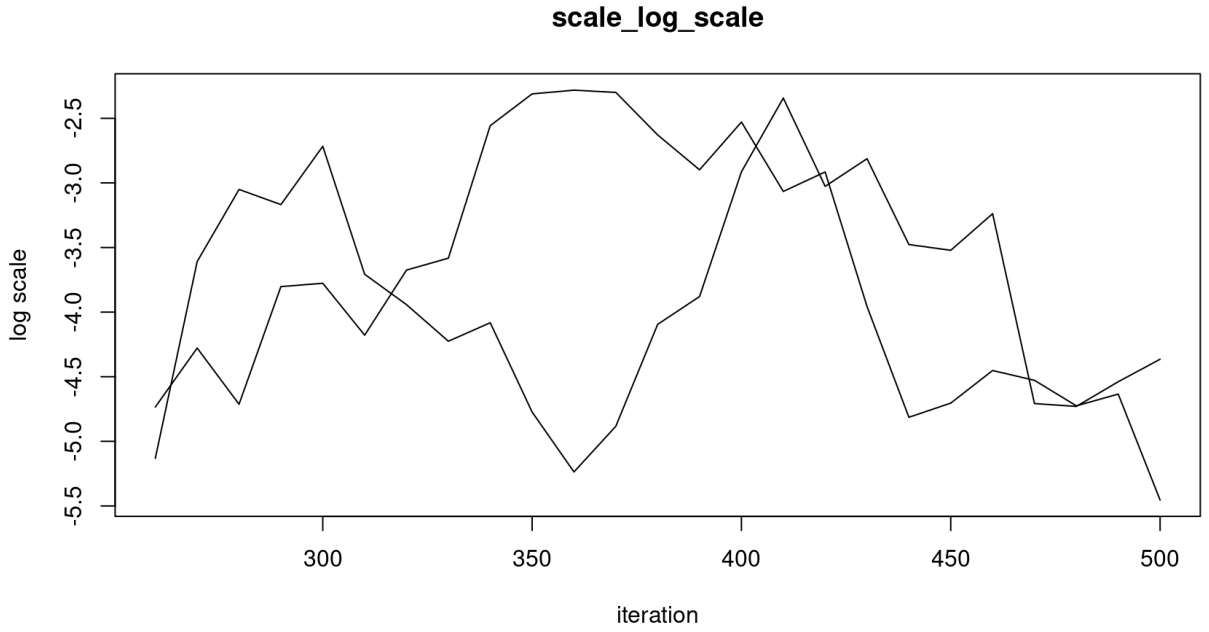

Quantiles of PSRF of scale_log_scale
max
0.99
0.9
0.5
PSRF quantiles
1.0
1.1
1.2
1.3
300
350
400
450
500
iterations

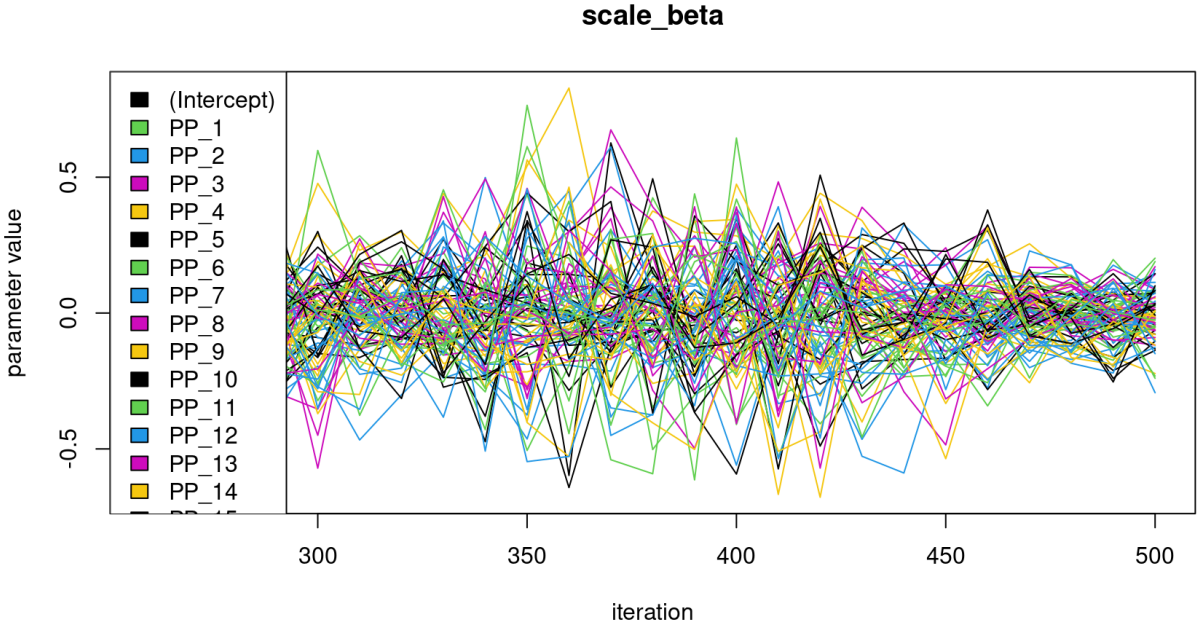

Quantiles of PSRF of beta
max
0.99
0.9
0.5
PSRF quantiles
1.0
1.1
1.2
1.3
1.4
300
350
400
450
500
iterations


range_beta
(Intercept)
parameter value
-1.60
-1.65
-1.70
300
350
400
450
500
iteration


noise_log_scale
log scale
-4.0
-4.5
-5.0
-5.5
-6.0
300
350
400
450
500
iteration


noise_beta
parameter value
0.3
0.2
0.1
0.0
-0.1
-0.3
(Intercept)
chess
white_noise
PP_1
PP_2
PP_3
PP_4
PP_5
PP_6
PP_7
PP_8
PP_9
PP_10
PP_11
PP_12
300
350
400
450
500
iteration


scale_log_scale
log scale
-2.5
-3.0
-3.5
-4.0
-4.5
-5.0
-5.5
300
350
400
450
500
iteration



We can see that **the PP log-variance parameters go to very negative values**, leading the associated **PP coefficients** to **shrink to 0**, inducing a model that is **practically stationary**.

4.4.4 The anisotropic case.

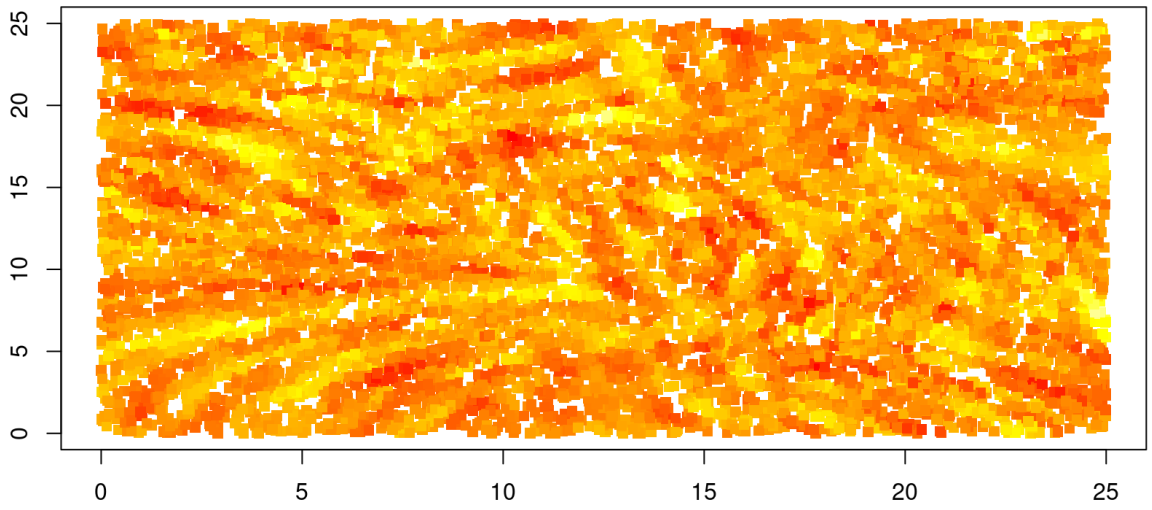
In the anisotropic case, the log-range parameters are symmetric 2×2 matrices. Therefore, they have 3 components. The associated regression coefficients will not be vectors like in the usual case, but matrices with 3 columns. Eventually, **the range PP variance will be a matrix of size** 3×3, as explained before.

Let's **simulate an anisotropic latent field**, using the same spatial locations and the same PP as before. We have 50 PPs, and we also need an intercept. *range_beta* is therefore a 50×3 matrix.

```
range_beta = rbind(
  c(-1, 0, 0), # intercept
  matrix(.5*rnorm(150), 50)
  )
NNarray_aniso = GpGp::find_ordered_nn(locs[seq(10000),], 10)

# getting the coefficients
chol_precision =
  Bidart::compute_sparse_chol(
    range_beta = range_beta, # range parameters
    NNarray = NNarray_aniso, # Vecchia approx DAG
    locs = locs[seq(10000),], # spatial coordinates
     range_X = matrix(1, 10000), # covariates for the range (just an intercep
t)
    PP = PP, use_PP = T, # predictive process
    nu = 1.5, # smoothness
    anisotropic = T # anisotropy
  )[[1]]
  # putting coefficients in precision Cholesly
chol_precision = Matrix::sparseMatrix(
    x = chol_precision[!is.na(NNarray_aniso)],
    i = row(NNarray_aniso)[!is.na(NNarray_aniso)],
    j = (NNarray_aniso)[!is.na(NNarray_aniso)],
    triangular = T
    )
# sampling the anisotropic process
seed_vector = rnorm(10000)
aniso_latent_field = as.vector(Matrix::solve(chol_precision, seed_vector))
aniso_observed_field = aniso_latent_field + .8*rnorm(10000)
```

Let's have a look at this latent field. We can see the anisotropy indeed.

```
Bidart::plot_pointillist_painting(locs[seq(10000),],  aniso_latent_field,  cex
= 1)
```



We move on and initialize a *mcmc_nngp_list* with an **anisotropic model**. The option *anisotropic = TRUE* is given to the function. The model is then run.

```
mcmc_ngp_list_aniso = Bidart::mcmc_nngp_initialize_nonstationary(
  observed_locs = locs[seq(10000),], observed_field = aniso_observed_field,
  nu = 1.5,
  range_PP = T, PP = PP, # use PP for range
  anisotropic = T # Covariance will be anisotropic
)
```

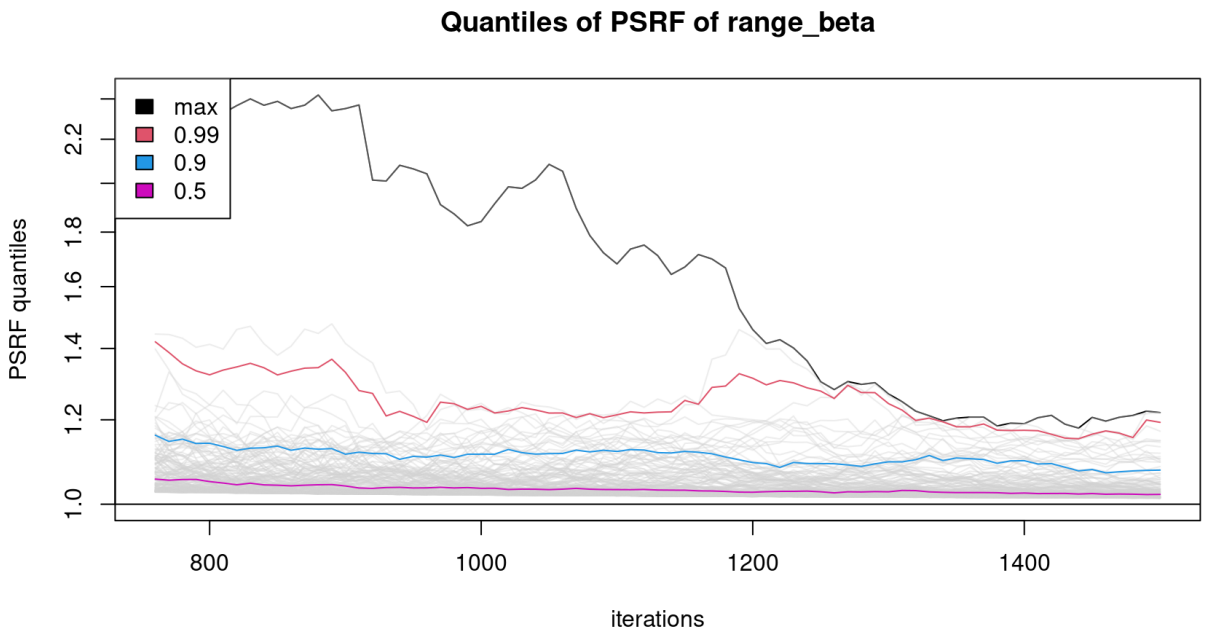
```
## range_log_scale_prior was automatically set to an uniform on (-6, 2)
```

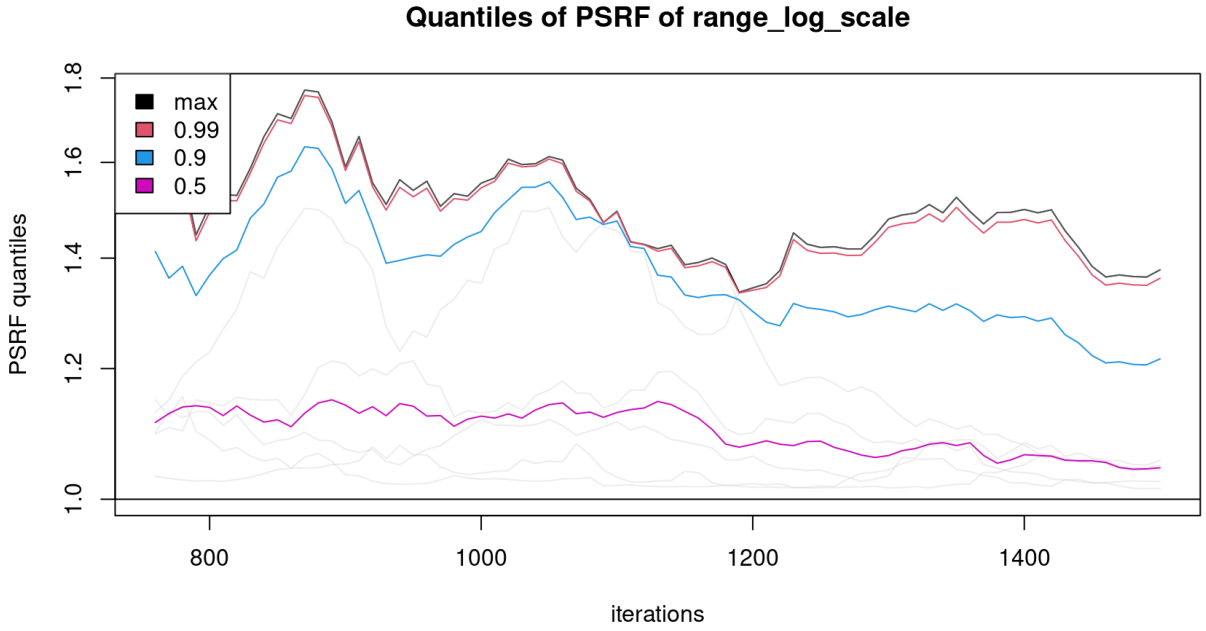
```
## Setup done, 1.77511835098267 s elapsed
```

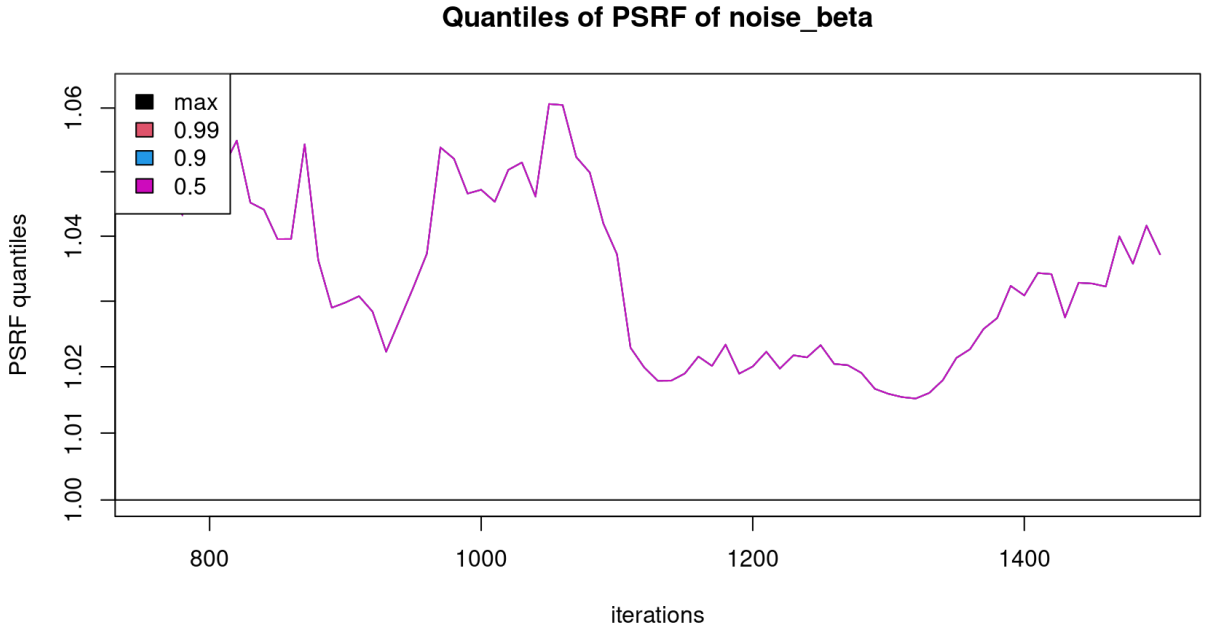
```
for(i in seq(15)) mcmc_ngp_list_aniso = Bidart::mcmc_nngp_run_nonstationary_s
ocket(
    mcmc_nngp_list = mcmc_ngp_list_aniso, # the list of states, data, #Vecchi
a approximation, etc
    n_cores = 2, # Parallelization over the chains
    num_threads_per_chain = parallel::detectCores()/2, # #Parallellization wi
thin each chain
    plot_diags = F, # MCMC diagnostics in the plot window
    )
```

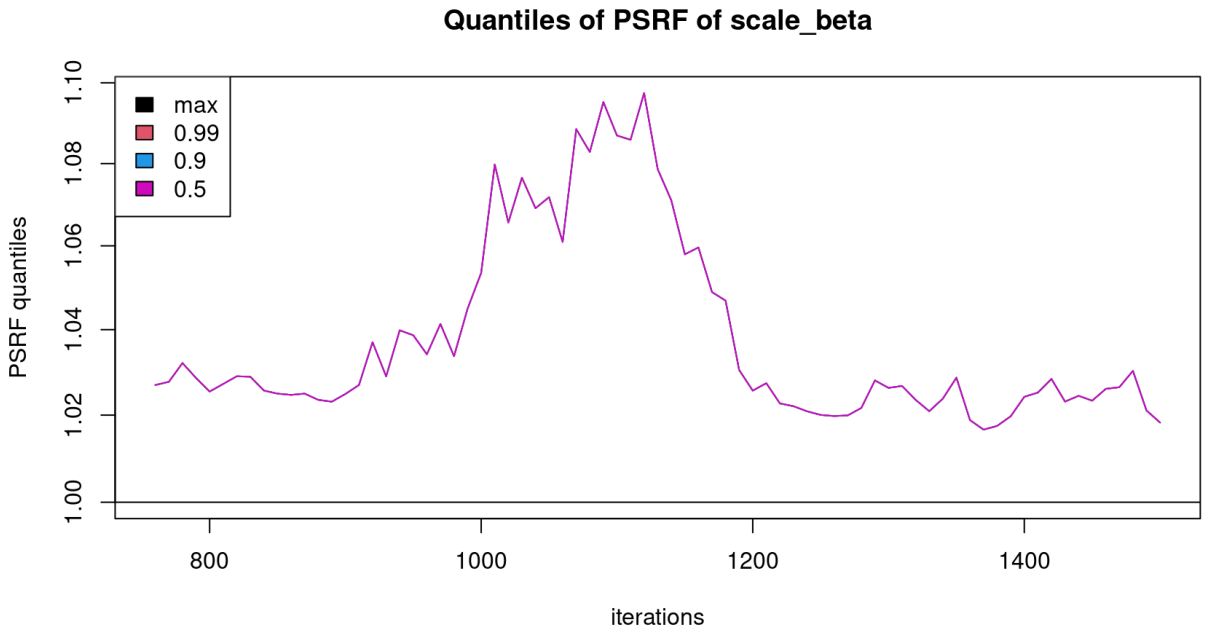
Let's show the diagnostic plots:

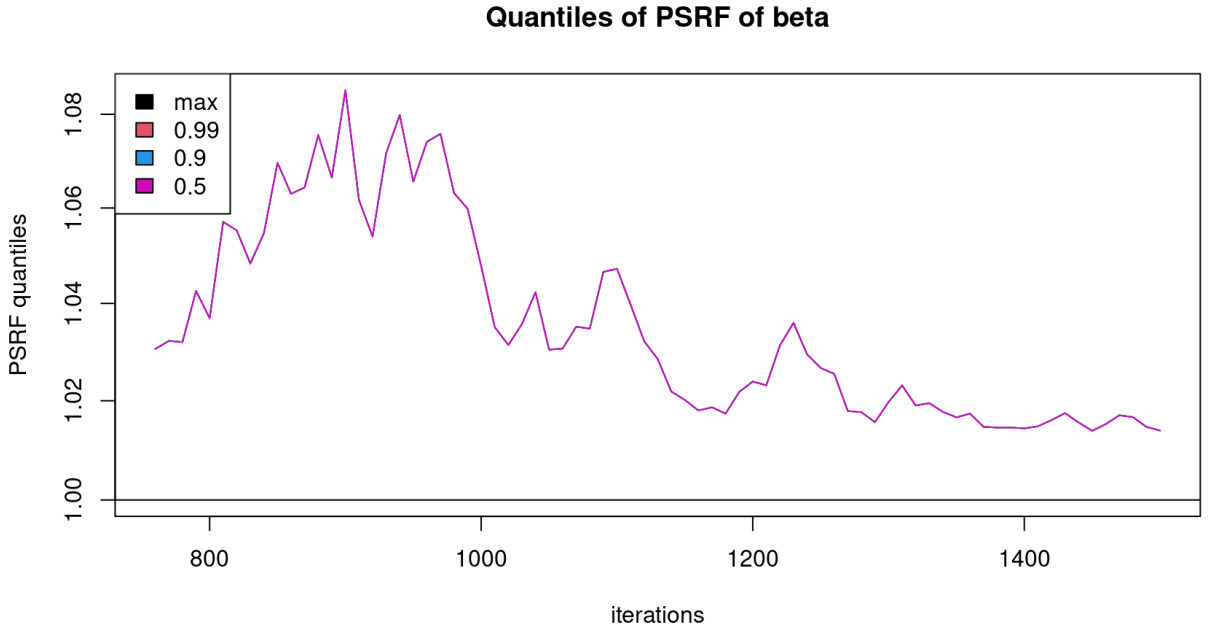
```
Bidart::diagnostic_plots(mcmc_ngp_list_aniso)
```

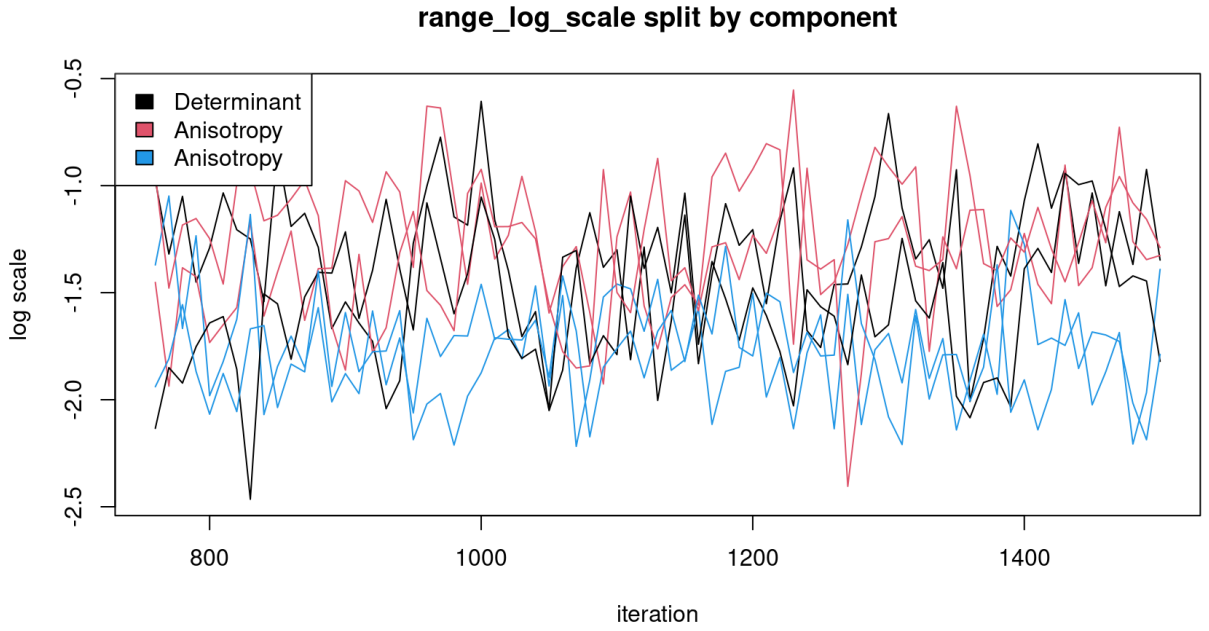


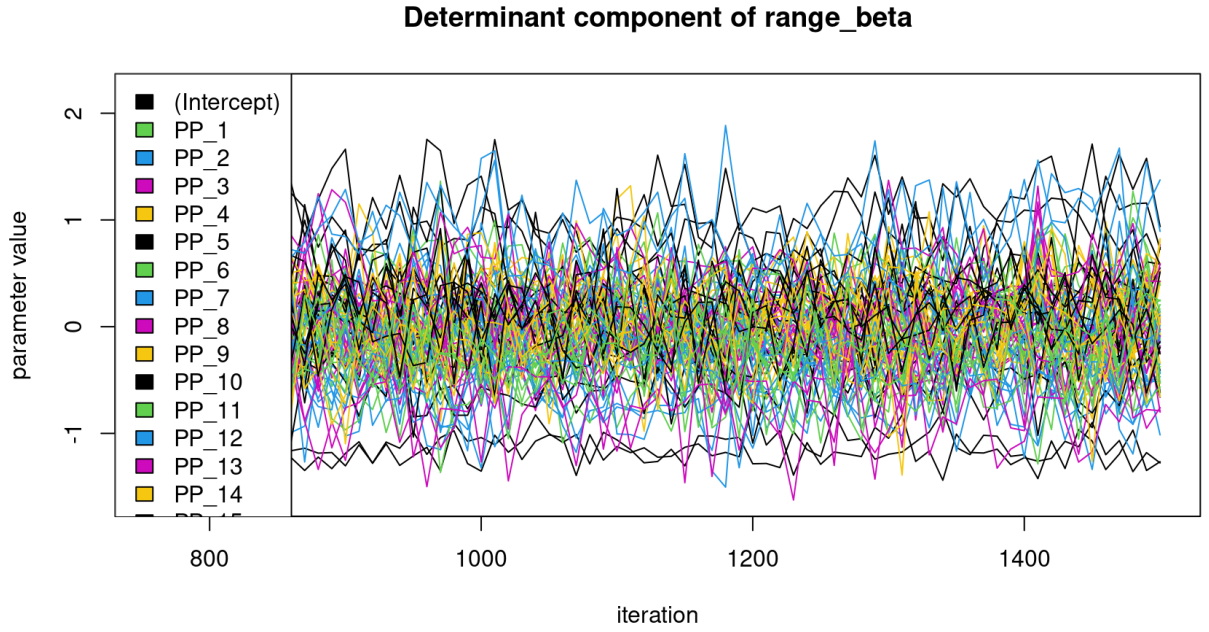

Quantiles of PSRF of range_log_scale
max
0.99
0.9
0.5
PSRF quantiles
1.0
1.2
1.4
1.6
1.8
800
1000
1200
1400
iterations

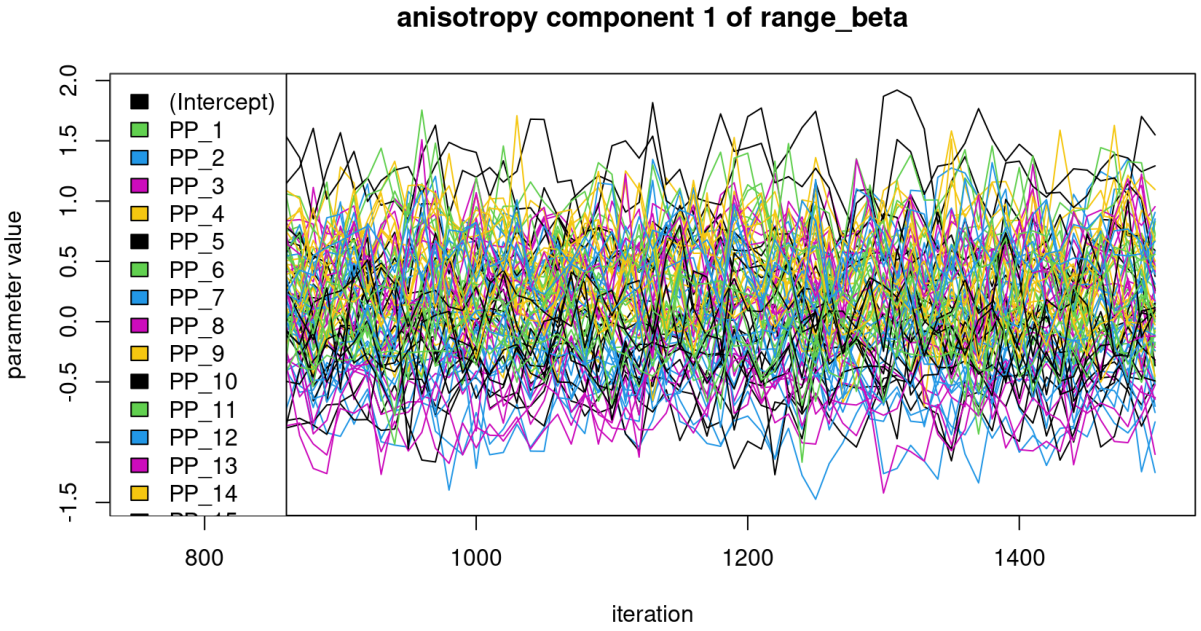

Quantiles of PSRF of noise_beta
max
0.99
0.9
0.5
PSRF quantiles
1.00
1.01
1.02
1.04
1.06
800
1000
1200
1400
iterations

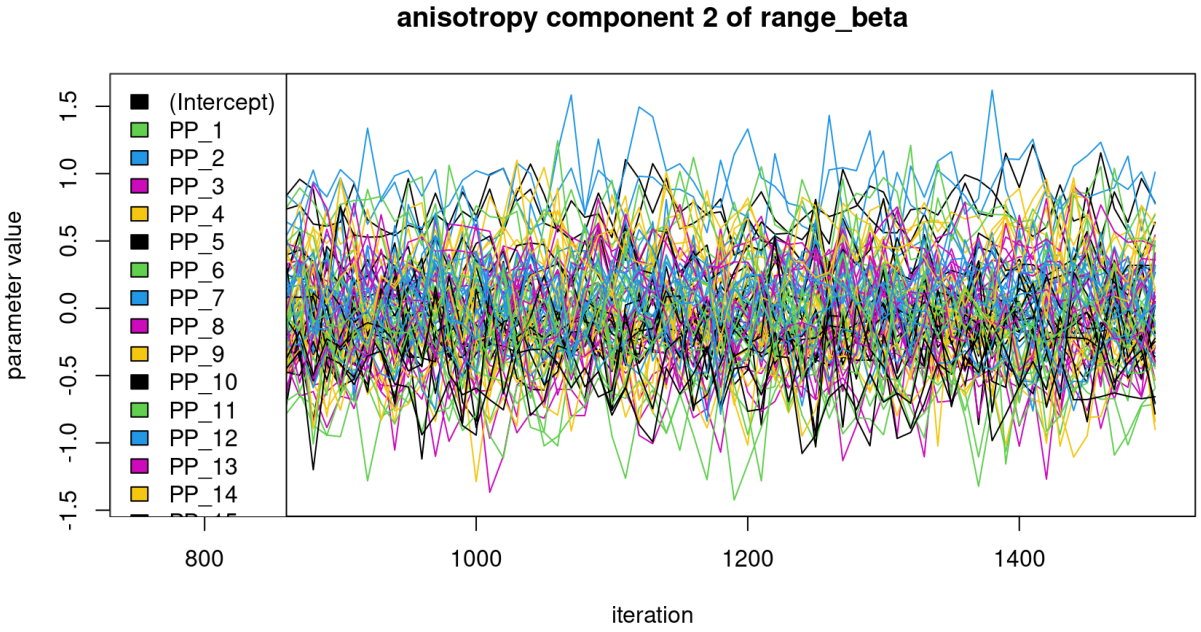

Quantiles of PSRF of scale_beta
max
0.99
0.9
0.5
PSRF quantiles
1.00
1.02
1.04
1.06
1.08
1.10
800
1000
1200
1400
iterations

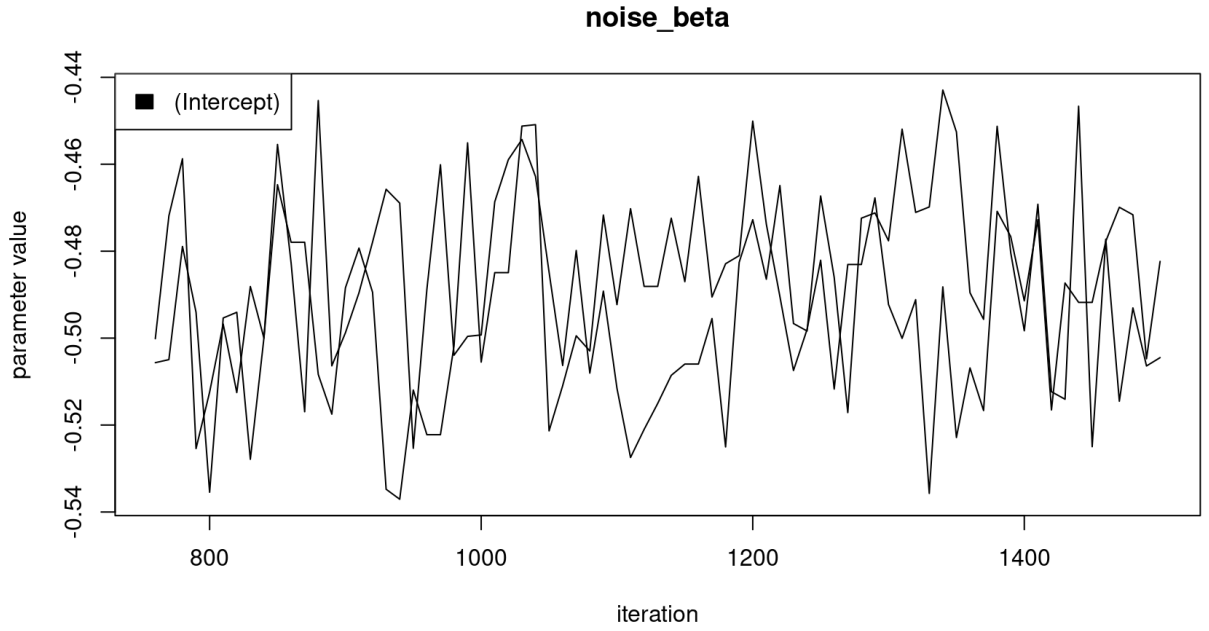

Quantiles of PSRF of beta
max
0.99
0.9
0.5
PSRF quantiles
1.00
1.02
1.04
1.06
1.08
800
1000
1200
1400
iterations

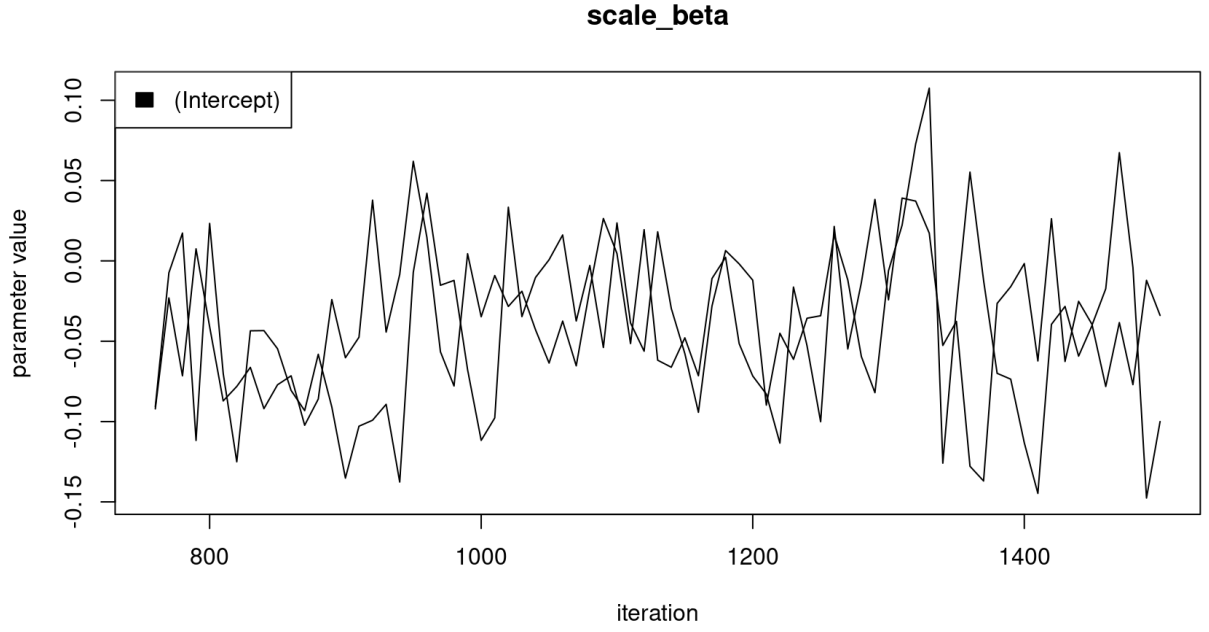

range_log_scale split by component
Determinant
Anisotropy
Anisotropy
log scale
-0.5
-1.0
-1.5
-2.0
-2.5
800
1000
1200
1400
iteration


Determinant component of range_beta
(Intercept)
PP_1
PP_2
PP_3
PP_4
PP_5
PP_6
PP_7
PP_8
PP_9
PP_10
PP_11
PP_12
PP_13
PP_14
parameter value
2
1
0
-1
800
1000
1200
1400
iteration


anisotropy component 1 of range_beta
parameter value
iteration
(Intercept)
PP_1
PP_2
PP_3
PP_4
PP_5
PP_6
PP_7
PP_8
PP_9
PP_10
PP_11
PP_12
PP_13
PP_14
2.0
1.5
1.0
0.5
0.0
-0.5
-1.5
800
1000
1200
1400


anisotropy component 2 of range_beta
parameter value
iteration
(Intercept)
PP_1
PP_2
PP_3
PP_4
PP_5
PP_6
PP_7
PP_8
PP_9
PP_10
PP_11
PP_12
PP_13
PP_14
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
800
1000
1200
1400





We can see that **range_log_scale has now more parameters**, we talked about this before (here,here, and there). Those parameters can be summarized in 3 components. **Two components regulate the anisotropy, and one component regulates the range**. We can see that the trace plots of all those components are quite high: the model is effectively non-stationary. In case of **over-modeling**, if the model identifies a nonstationary range but no anisotropy, the two components corresponding to the anisotropy (red and blue) will drop to very negative values, inducing a simpler model. If the model identifies no nonstationarity, all components will become very negative.

4.5 Conclusion for *Running the model*.

This was a long section. First, we initialize the model. Then, we show how to run it and diagnose MCMC convergence. Eventually, we show how to understand what is going on. Hopefully the previous section will allow a practitioner to be in control of the model and the algorithm, and not getting lost or carried over by the complexity of the available model formulations.

5 After the model has run.

You have a data set. You have set up a model. Your chains have run properly. Now, what ?

5.1 Estimation.

We are doing statistics, so we want to **estimate parameters**. The function *estimate_parameters* allows to retrieve estimates of the **high-level parameters** and the **latent field at the observed locations**. Its arguments are:

- *mcmc_nngp_list*, with samples.
- *burn_in*, a number between \(0\) and \(1\) indicating the proportion of discarded observations.
- *get_samples*, a Boolean indicating if samples should be returned.
- *lib.loc*, a path to the location of the library (useless on laptop).

Here is an example:

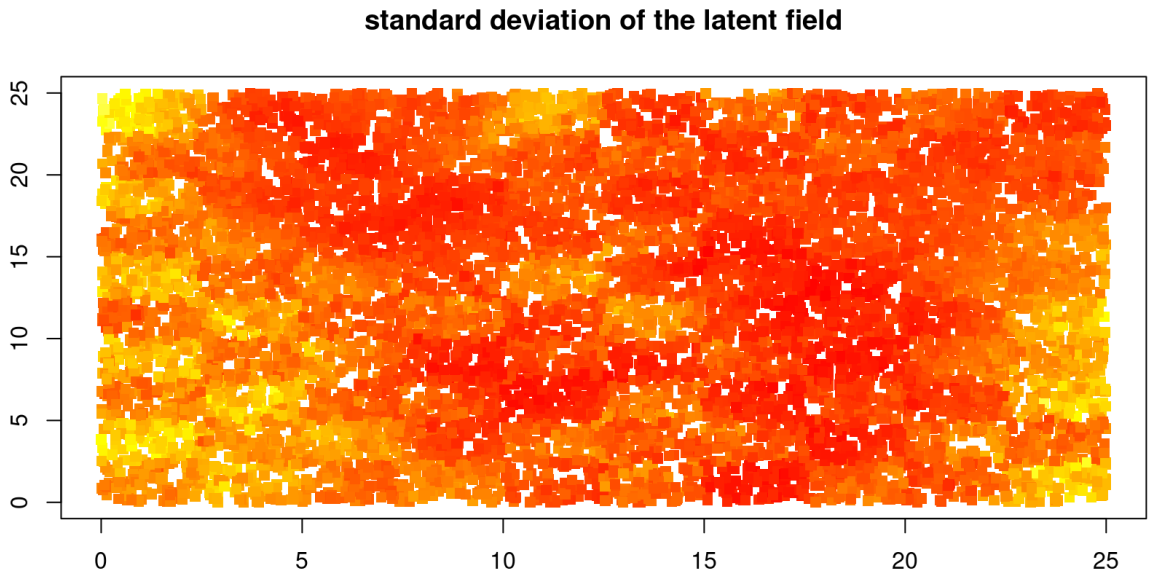
```
estimation = Bidart::estimate_parameters(
  mcmc_nngp_list = mcmc_nngp_list,
  burn_in = .5,
  get_samples = T,
  lib.loc = NULL)
```

The function returns a *list* of two components, *summaries* and *samples*.

As indicated by its name, *summaries* is a list of **summmaries of all the parameters**. Each summary contains an *array*, whose rows correspond to the **mean, median, 2.5% and 97.5% quantiles, and standard deviation**, and whose columns correspond to the components of the parameters. The depth is usually one, except in the case of **anisotropic range**, where the depth is 3.

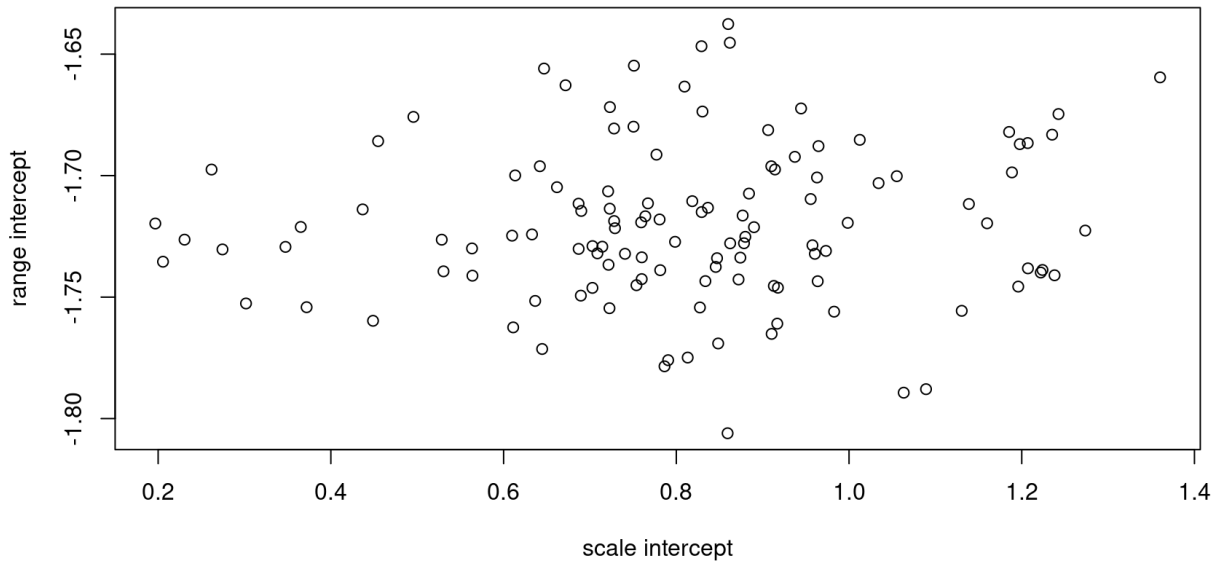
Here is a little example. Let's visualize the standard deviation of the latent field at the observed locations, for the heteroskedastic model we set up and ran in the previous section.

```
Bidart::plot_pointillist_painting(
  mcmc_nngp_list$data$locs,
  estimation$summaries$field["sd",,],
  main = "standard deviation of the latent field"
)
```



The other component contains **samples** and comes in handy for more advanced treatments Each sample is an *array*. Its two first dimensions are the dimensions of the parameter, and the last dimension is the dimension of the sample. For example: I have 150 MCMC samples of the anisotropic range parameter with one intercept, 10 PPs. The sample will have dimension $(11\times 3 \times 150)$. I have 200 MCMC samples of the noise variance parameter with one intercept, 10 covariates, 20 PPs. The sample will have dimension $(31\times 1 \times 200)$. Let's, for example, plot samples of the latent field variance intercept against those of the range intercept.

```
plot(
  estimation$samples$scale_beta[1,1,],
  estimation$samples$range_beta[1,1,],
  xlab = "scale intercept",
  ylab = "range intercept",
  )
```



5.2 Prediction.

Prediction allows to **compare models** and **forecast the interest variable at unobserved locations**. Several functions are used following what is to be predicted.

5.2.1 Latent field.

First the **latent field** is predicted. This allows for **interpolation**. Aside of *mcmc_nngp_list*, its arguments are:

- *predicted_locs*, a *matrix* of **spatial coordinates** where the prediction is done. They must be **distinct from the locations used in the training** (use *estimate parameters instead*). However, there can be duplicates.
- *X_range_pred* and *X_scale_pred* are *data.frame*s of **covariates** for the range and the latent field variance respectively, observed at *predicted_locs*. They must not change within one spatial location !
- *burn_in* is the MCMC burn-in.
- *num_threads_per_chain* is the number of OMP threads.
- *lib.loc* is the location of the libraries.

Let's predict the latent field on the test locations.

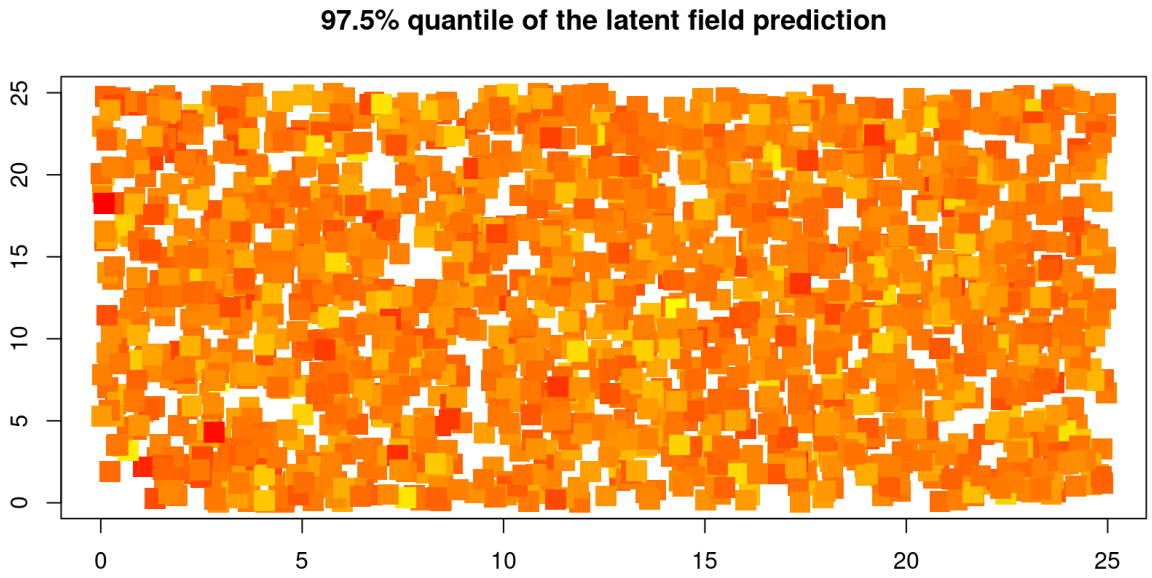
```
predicted_locs = locs[-seq(10000),]
prediction_field = Bidart::predict_latent_field(
  mcmc_nngp_list = mcmc_nngp_list,
  predicted_locs = predicted_locs,
  X_range_pred = NULL, X_scale_pred = NULL,
  burn_in = .5,
  num_threads_per_chain = parallel::detectCores()/2,
  lib.loc = NULL
  )
```

The outputs are:

- *predicted_locs_unique* is a *matrix* of non-redundant, re-ordered prediction spatial coordinates.
- *summaries*, a *list* of *array*s of summaries in the same format as the estimation summaries. It contains summaries of the latent field, the log range, and the log latent variance.
- *summaries*, a *list* of *array*s of MCMC samples in the same format as the estimation samples. It contains samples of the latent field, the log range, and the log latent variance.

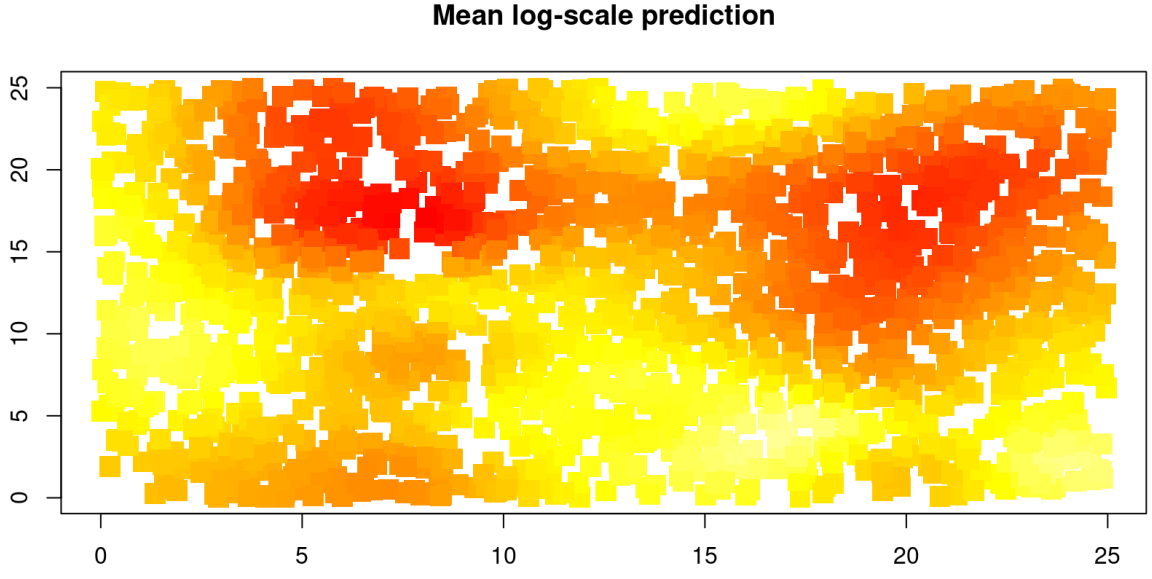
Let's plot a prediction of the latent field 97.5% quantile.

```
Bidart::plot_pointillist_painting(
  locs = prediction_field$predicted_locs_unique,
  field = prediction_field$summaries$field["q0.975",,1],
  cex = 2,
  main = "97.5% quantile of the latent field prediction"
)
```



Let's now represent the **mean predicted log variance of the latent field** (a little subtelty here is that the summaries are indexed with respect to the input locations for the range and scale, but with respect to the *predicted_locs_unique* output for the latent field).

```
Bidart::plot_pointillist_painting(
  locs = locs[-seq(10000),],
  field = prediction_field$summaries$log_scale["mean",,1],
  cex = 2,
  main = "Mean log-scale prediction"
)
```



5.2.2 Noise.

The **noise variance** is predicted using a **similar syntax**, *mutatis mutandis*. Obviously, the only *data.frame* of covariates is now *X_noise_pred*, since we are predicting the noise. There are however

additional subtle yet important **differences** with respect to latent field prediction.

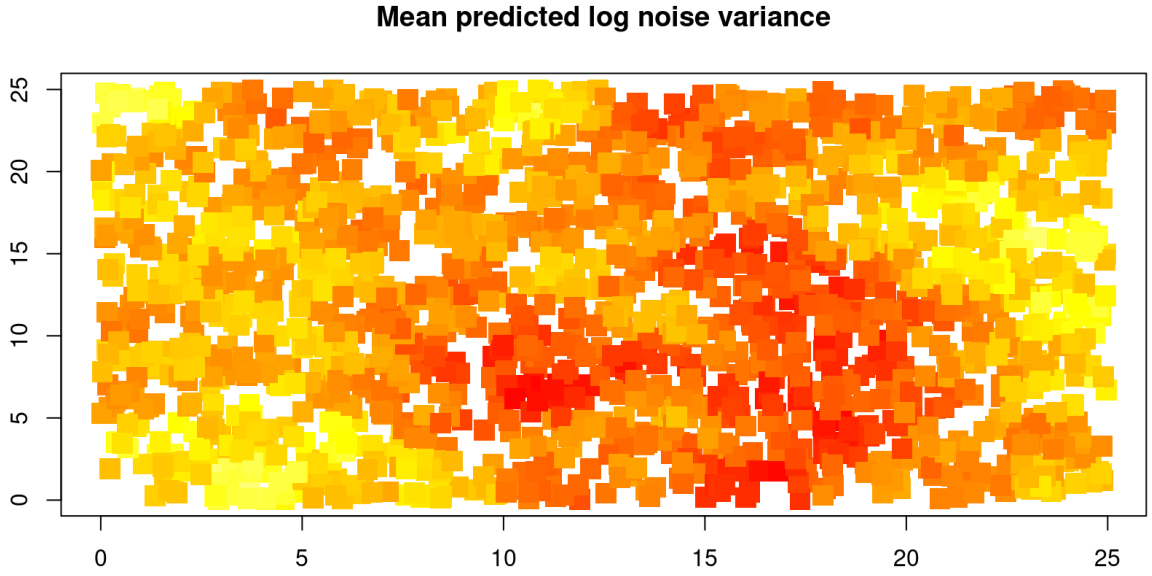
- *X_noise_pred* may change within one spatial location, because the noise does not suffer from the same identification problems as the other parameters.
- *predicted_locs* can also contain **locations from the training set**.

Let's predict the noise on **all locations**.

```
prediction_noise = Bidart::predict_noise(
  mcmc_nngp_list = mcmc_nngp_list,
  predicted_locs = locs, # NB !
  X_noise_pred = X, # NB !
  burn_in = .5
  )
```

We obtain a *list* containig two *arrays*, one of *summaries*, another of *samples*, and a *matrix* of predicted spatial coordinates. Let's visualize the **mean log-noise variance at the test locations**.

```
Bidart::plot_pointillist_painting(
  locs = prediction_noise$predicted_locs[-seq(10000),], # removing the train
locations
  field = prediction_noise$summaries["mean",-seq(10000),1],
  cex = 2,
  main = "Mean predicted log noise variance"
)
```



5.2.3 Fixed effects.

The prediction of the fixed effects simply consists in **multiplying the covariates** taken at the predicted locations by the **samples of the regression coefficients** estimated by the model. Contrary to the previous functions, there is no need to specify *predicted_locs*. However, it is of course necessary to give *X_pred*, the covariates at the predicted locations. Like in the noise prediction, of course, it is possible to predict the fixed effects at the observed locations as well.

Let's do that: predicting the fixed effects at **both the train and test locations**.

```
prediction_fixed = Bidart::predict_fixed_effects(
  mcmc_nngp_list = mcmc_nngp_list,
  X_pred = X,
  burn_in = .5
  )
```

5.3 Model comparison.

Nonstationary spatial modelling, like ordering pizza, can be very frustrating because many seemingly delicious options are available, yet you have to pick only one. However we can have a taste of several models and pick the better one using model comparison. Now that I have started culinary analogies, let me put another layer in a field related to model selection: it is not because you have a lot of ingredients in the fridge that they must all go on the pizza at the same time. Very useful material about model comparison criteria can be found here (https://inlabru-org.github.io/inlabru/articles/prediction_scores.html#poisson-model-example).

5.3.1 Deviance Information Criterion.

The Deviance Information Criterion (https://en.wikipedia.org/wiki/Deviance_information_criterion) (DIC) is a classical tool of comparison for Bayesian models. **The smaller, the better**. It is, however, known to be prone to selecting **overfit models**.

It is computed as follows:

```
Bidart::DIC(mcmc_nngp_list)
```

```
## [1] 34503.63
```

5.3.2 Log-score at observed locations.

5.3.2.1 A glorified Mean Squared Error.

The log-score is the **log-density of the observations knowing the model**, and is written as

\[\int log(p(y_{obs}|w, \beta, \tau)) p(w, \beta, \tau|y_{obs})d(w, \beta, \tau),\] \(y_{obs}\) being the observations, \(w\) being the latent field, \(\beta\) the regression coefficients, and \(\tau\) the noise standard deviation. \(log(p(y_{obs}|w, \beta, \tau))\) tells how well a combination of parameters \((w, \beta, \tau)\) explains the observations \(y_{obs}\). \(p(w, \beta, \tau|y_{obs})\) is the *a posteriori* distribution of the parameters. The above integral means “in average, does the *a posteriori* distributions of the parameters explain well the observed data ?”. Since we are measuring how well the model explains the data, **the bigger the log-density, the better**.

In the **Gaussian noise** model, \(p(y_{obs}|w, \beta, \tau)\) can be written as \[\Pi_{i\in obs}exp((y_i-w_i-X_i\beta)^2/\tau_i^2)/(\sqrt{2\pi}\tau_i)\] After passing to the log and dividing by the number of observations \(n\), we obtain
\[\frac{1}{n}\Sigma_{i\in obs}(y_i-w_i-X_i\beta)^2/\tau_i^2 - log(\sqrt{2\pi}) -log(\tau_i)\]

this criterion can be connected with the so-called **[Mean Squared Error](https://en.wikipedia.org/wiki/Mean_squared_error) (https://en.wikipedia.org/wiki/Mean_squared_error)**, who writes \(\frac{1}{n}\Sigma_{i\in obs}(y_i-w_i-X_i\beta)^2\), with the addition of some heteroskedasticity and spatial effects. The advantage with respect to the MSE is that this method takes into account the **incertitude of the distribution**. The MSE be happy when square be small, and be angry when

square be big; on the other hand, the log-score and its noise penalty **penalize both wrong and confident, and correct but hesitant predictions**, while it is softer on wrong and hesitant predictions and, of course, happiest with correct and confident predictions. Like criteria based on observations, it is prone to **over-fitting**.

This over-fitting may of course be due to the addition of **too many covariates**. However, with nonstationary spatial modeling, there are other traps. The first is **too complex model specification of the spatial effects**. The second is that a **low smoothness** may cause the latent field to adjust better the observed locations, but deteriorate the prediction of locations that are not observed.

5.3.2.2 Implementation

In the above formula, we have an integral, so we will need **MCMC samples**. One may ask: “Cannot we just plug summaries in ?”, but the answer is “No”. The parameters of the model are correlated, and summaries eliminate the correlation. For this reason, the *burn_in* parameters of the predictions must be the same in order to guarantee that the samples are well-aligned. Those samples are obtained through three functions:

- *predict_fixed_effects* gives samples of the **fixed effects**, called *fixed_effects_samples*.
- *estimate_parameters* gives samples of the **latent field**, called *latent_field_samples*.
- *predict noise* gives samples of the **log noise variance**, called *log_noise_samples*.

In addition to that, we need of course **observations**, called *observed_field*.

Let’s give it a go, remembering that we predicted the noise and the fixed effects at both the observed and predicted locations. We need therefore to do some subsetting of our predicted samples.

```
train_log_score = Bidart::log_score_Gaussian(
  observed_field = c(observed_field)[seq(10000)],
  latent_field_samples = estimation$samples$field_at_observed_locs,
   log_noise_samples  =  prediction_noise$predicted_samples[seq(10000),,,drop=
F],
  fixed_effects_samples = prediction_fixed$predicted_samples[seq(10000),,,dro
p=F]
)
```

The result is a *list* with three components.

- The *samples matrix* is a collection of log-densities, at all locations, at all MCMC iterations.
- The *per_obs vector* is a collection of log-densities, at all locations, but averaged over MCMC iterations.
- The *total* is the sum of *per_obs*. It can be divided by the number of observations to give a mean log-density (see below).

```
print(paste("The mean log density at observed locations is", round(train_log_
score$total / 10000, 2)))
```

```
## [1] "The mean log density at observed locations is -1.56"
```

5.3.3 Log-density at unobserved locations.

The problem of over-fitting is addressed by **holding observations out for testing**, and using the same log-density formula.

5.3.3.1 Train-test partition.

The models take quite some time to run. It is therefore not realistic to have a naive “leave one out” approach. Because of **spatial correlation**, drawing a fraction of the observations at random and using them as a test sample must be done carefully. One approach that seems sensible is to use a **max-min** heuristic to select the test locations, in order to have an **even distribution** of the test locations over the spatial domain, and to **avoid clumps** of un-observed locations.

Another approach (https://arxiv.org/abs/1710.05013) is to shroud regions of the spatial domains, **mimicking a cloud cover** for satellite measurements.

Take into the problem you want to address and adjust your selection method accordingly. If your objective is to estimate the regression coefficients for the fixed effects, you may prefer to carve big regions out of the training data and test the center of the regions, so that you get rid of the spatial effects. If you want to interpolate, peppering the spatial domain with test locations may be a better idea.

5.3.3.2 Code sample.

The same function *log_score_Gaussian* as before is used. The only differences are that now, the latent field samples come from *predict_latent_field*, and that subsetting changed for the other samples.

```
test_log_score = Bidart::log_score_Gaussian(
  observed_field = c(observed_field)[-seq(10000)],
  latent_field_samples = prediction_field$predicted_samples$field,
   log_noise_samples  =  prediction_noise$predicted_samples[-seq(10000),,,drop=
F],
  fixed_effects_samples = prediction_fixed$predicted_samples[-seq(10000),,,dr
op=F]
)
print(paste("The mean log density at predicted locations is", round(test_log_
score$total / 10000, 2)))
```

```
## [1] "The mean log density at predicted locations is -0.57"
```

5.4 Conclusion for *After the model has run*

This section would be better named “after the model**s** have run”. Indeed, when many formulations are available, **model comparison using several criteria** is of the essence. Note that it is not always possible to satisfy all the criteria, in which case it it arguably a better idea to select the best predictor rather than the best smoother in order to avoid **overfitting**. Once model selection is done, **estimation** of the parameters of interest and **prediction** of the response at un-observed locations can be carried out.

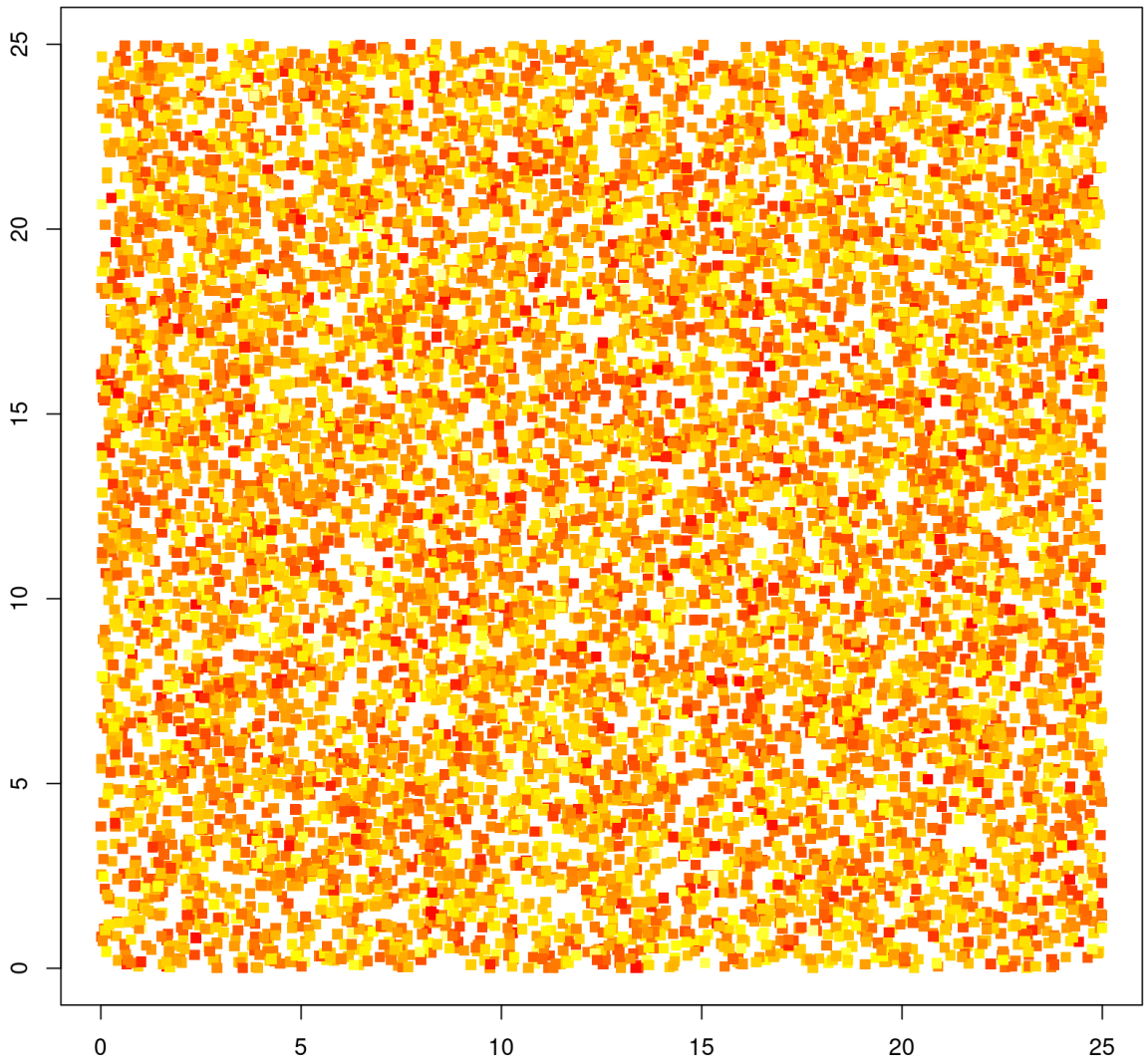
6 Graphic functions

6.1 Pointillist plotting

The *plot_pointillist_pointing* takes as arguments a *matrix* of spatial coordinates called *locs* and a *vector* of values observed at those coordinates called *latent_field*. Optionally, it admits a *numeric* called *cex* and a *character* called *main* working like in *plot*. It returns an image similar to a Pointillist

painting (https://en.wikipedia.org/wiki/Pointillism).

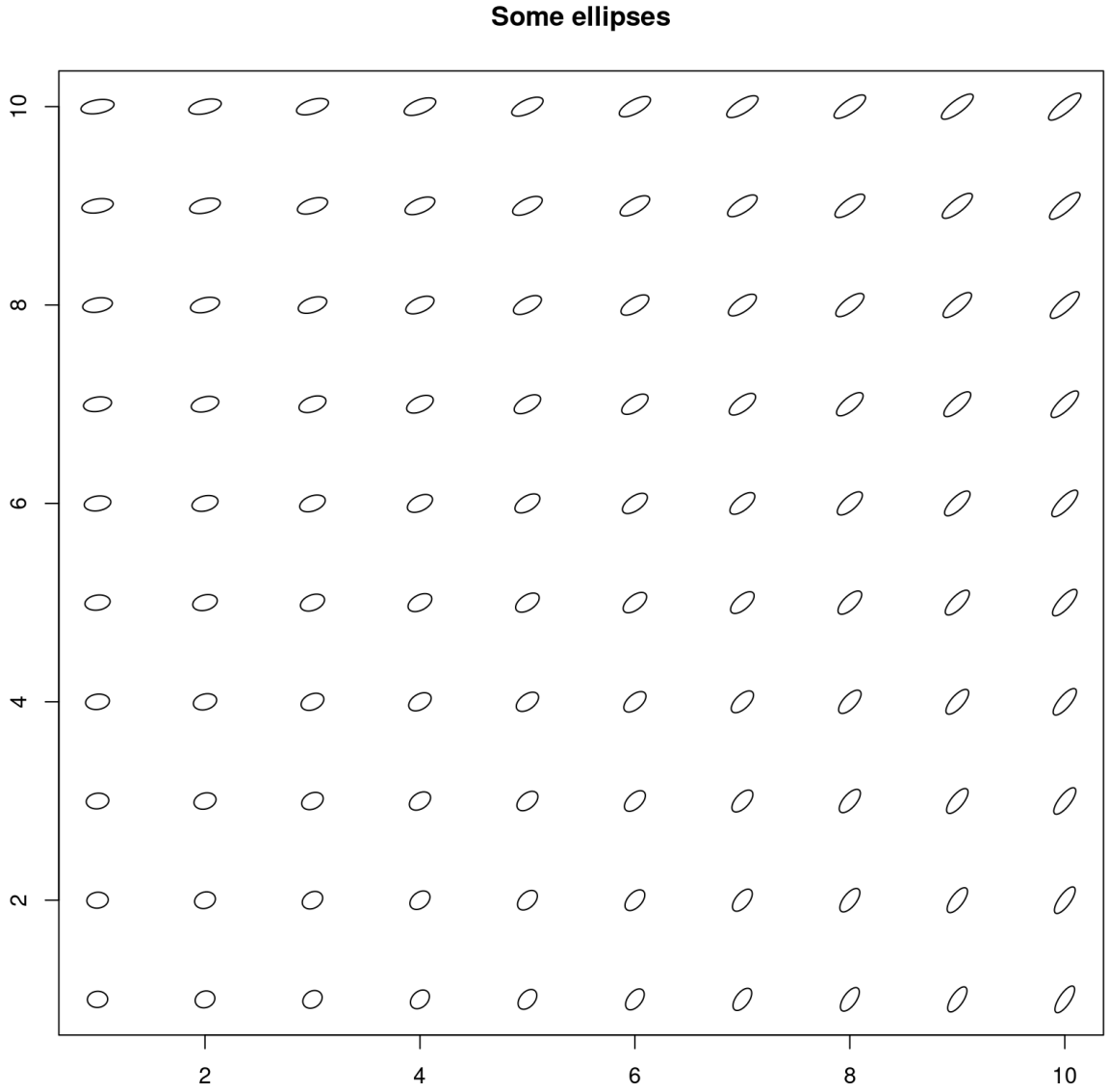
```
Bidart::plot_pointillist_painting(locs[seq(10000),], latent_field, cex = 1)
```



6.2 Ellipse plotting

plot_ellipses takes as arguments a *matrix* of spatial coordinates called *locs* and a *vector* or a *matrix* of log-matrix values observed at those coordinates called *log_range*. The log-matrices are then passed to the matrix exponential to obtain the ellipses. Optionally, it admits a *character* called *main* working like in *plot*. It also has an option *shrink* working like *cex* in *plot*

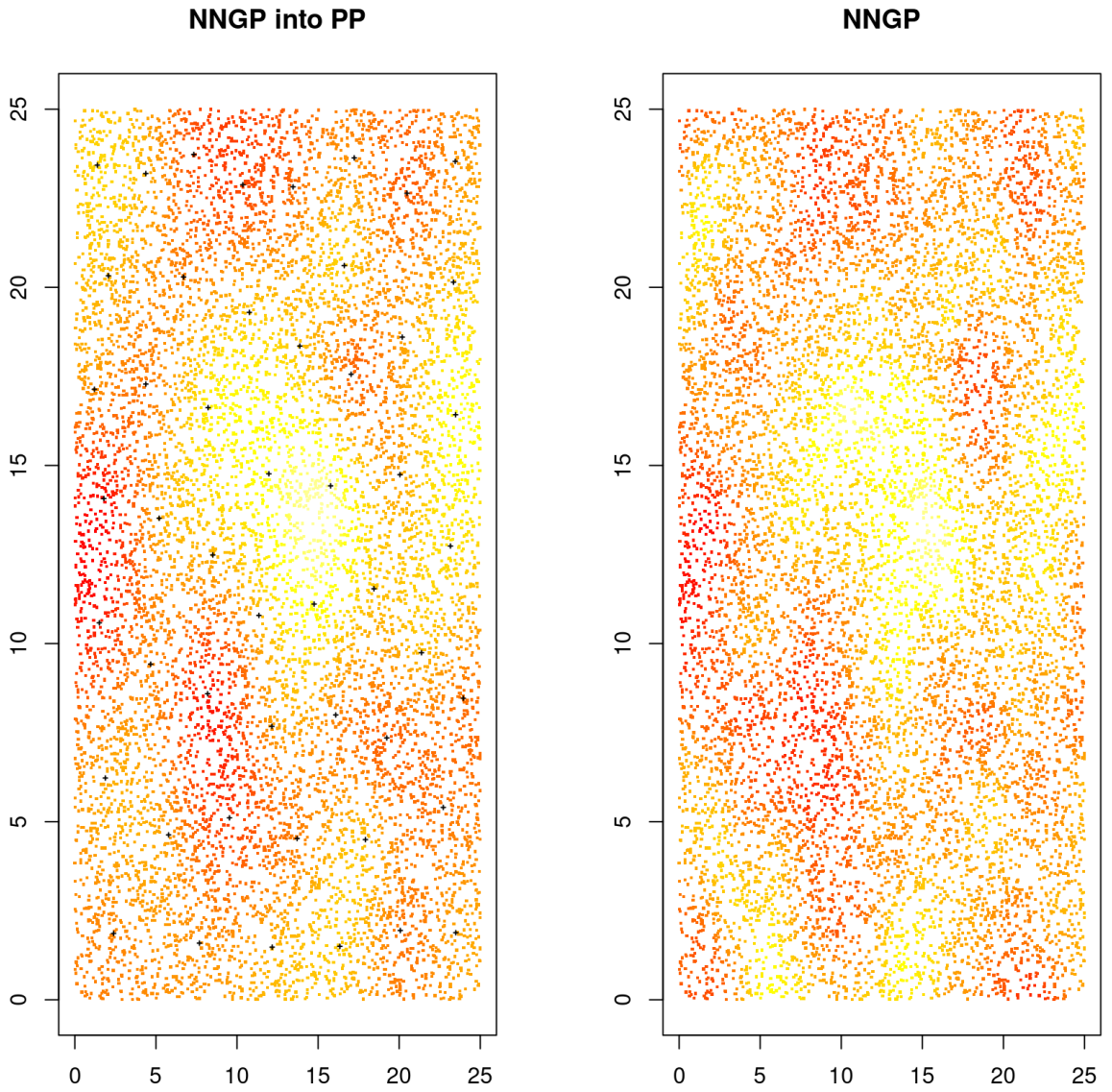
```
Bidart::plot_ellipses(
  locs = as.matrix(expand.grid(seq(10), seq(10))),
  log_range = cbind(1, as.matrix(expand.grid(seq(10), seq(10)))) %*%
    matrix(c(-1, .01,.05, .2, -.1, .1, -.1, .1, .03), 3),
  shrink = .05, main = "Some ellipses"
)
```



6.3 Predictive Processes

compare_PP_NNGP takes a predictive process as an argument and returns two plots. The one on the left is a pointillist painting of a sample of the PP, with the knots coordinates added as black crosses. The one on the right is a full NNGP.

```
Bidart::compare_PP_NNGP(
  PP
)
```



7 Gallery

This section presents real case studies using our model.

7.1 Lead concentration in the United States

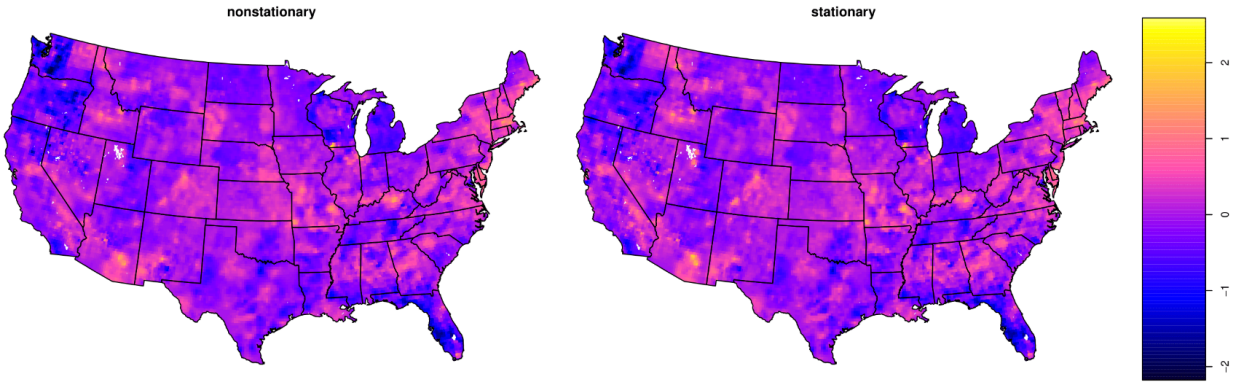
This data set is taken from T. Hengel's A Practical Guide to Geostatistical Mapping (https://edepot.wur.nl/485517). We study **lead contamination** in the ground of the mainland of the United States of America. **Four models** were tested, one with nonstationary **noise variance**, one with nonstationary **latent process variance**, one with **both**, and one with **none**. There are many **explanatory variables**, who were used to explain the **response** and the **noise variance** in models who have a nonstationary noise. The most **spatially coherent** explanatory variables were also used to model the **latent field variance** in relevant models. Model comparison is summarized in the following table. The first column is a **nickname** for each model. The following two columns correspond to the **formulation** of the model. The three next columns correspond to the **model comparison** criteria. The last four columns are some short information about the **MCMC run**.

model's nickname	nonstat noise	nonstat scale	pred	smooth	DIC	time	n.iter	min ESS
The Full Monty	Yes	Yes	-0.71	-0.54	65986	265	2000	110
The Ockham's razor's	Yes	No	-0.72	-0.55	67137	213	2000	103
The Overfitter	No	Yes	-0.96	-0.53	70620	236	2000	66
The Vanilla	No	No	-0.79	-0.64	78135	183	2000	168

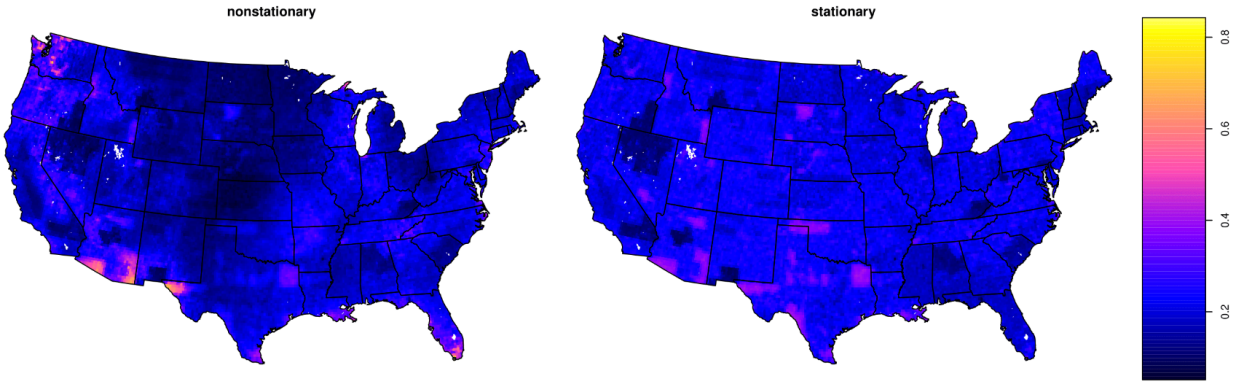
We can see that "The Full Monty", who is the most nonstationary, does much better than "The Vanilla", who is stationary. However, "The Ockham's razor's", does almost as good as the "Full Monty" with a much simpler formulation. Eventually, the "Overfitter" has a good DIC and smoothing density but generalizes poorly to unobserved locations.

Let's **compare "The Full Monty" and "The Vanilla"**.

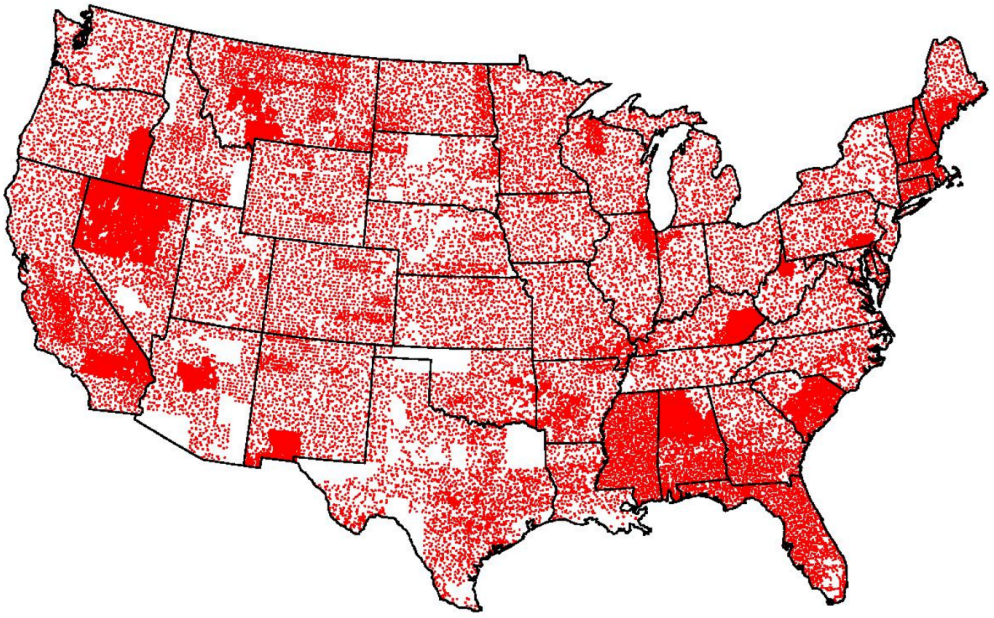
The image below shows the **mean of the latent field**. There is **little difference**.



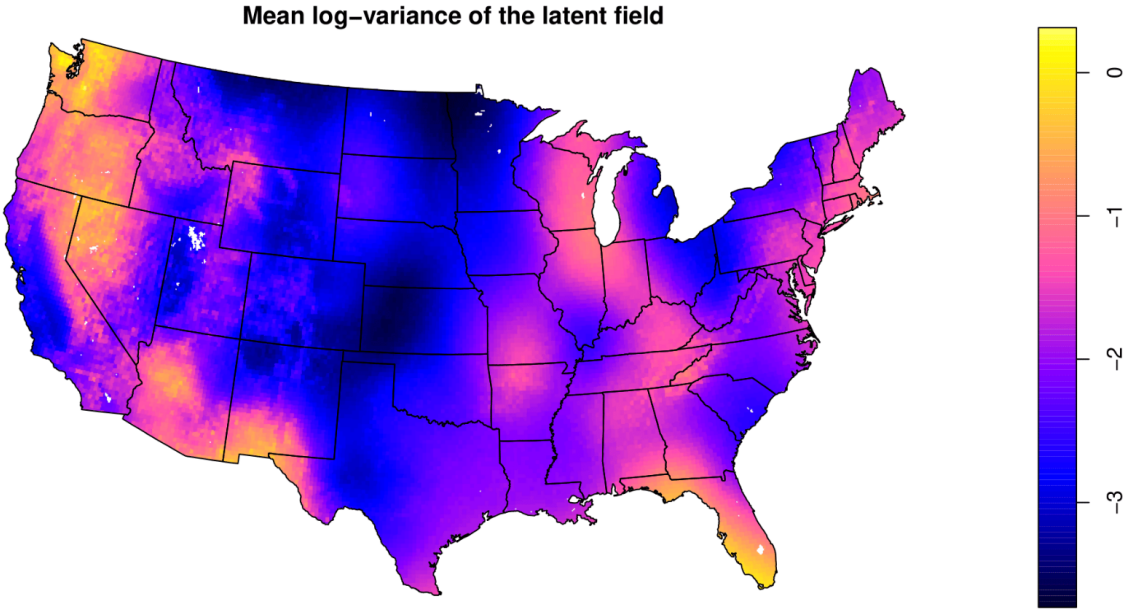
However, when we show the **standard deviation of the latent field**, there is a **huge difference**. Why is that ?



A look at the **sampling sites**, shown below, is useful. We can see that there are places with lots of observations, and gaps. The **stationary** model **predicts confidently** where there are **dense observations**, and less confidently where there are less observations. And that's all.

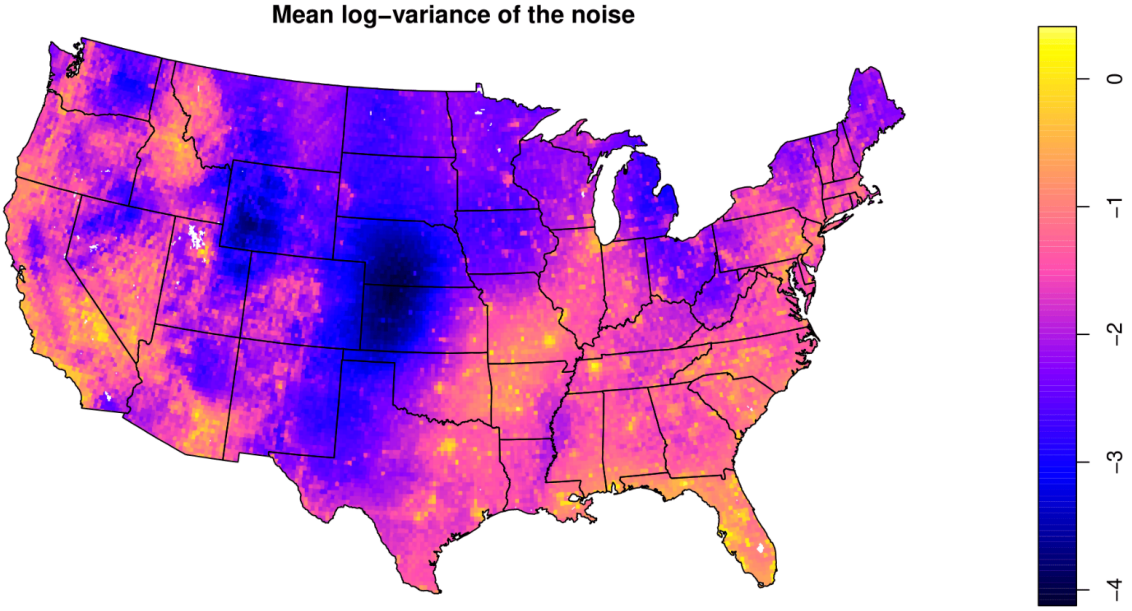


The **nonstationary model** must therefore use its nonstationarity to predict differently. Let's look at the estimate of the **mean log variance of the latent process**. We can a correspondence between the estimated mean log-variance and the standard deviation of the prediction. For example, there are **gaps** both at the **border with Mexico** and the **Midwest**, incurring **higher predicted variance**. However, the gaps at the border have much higher standard deviation, accordingly to the log-variance field.



The last, and **most important** parameter, is the **variance of the noise**. The map of the **mean log variance of the noise** exhibits the same general pattern as the field's variance. However, we can see that there are **hot spots** corresponding to the **effects of covariates**. Those hot spots indicate **cities**

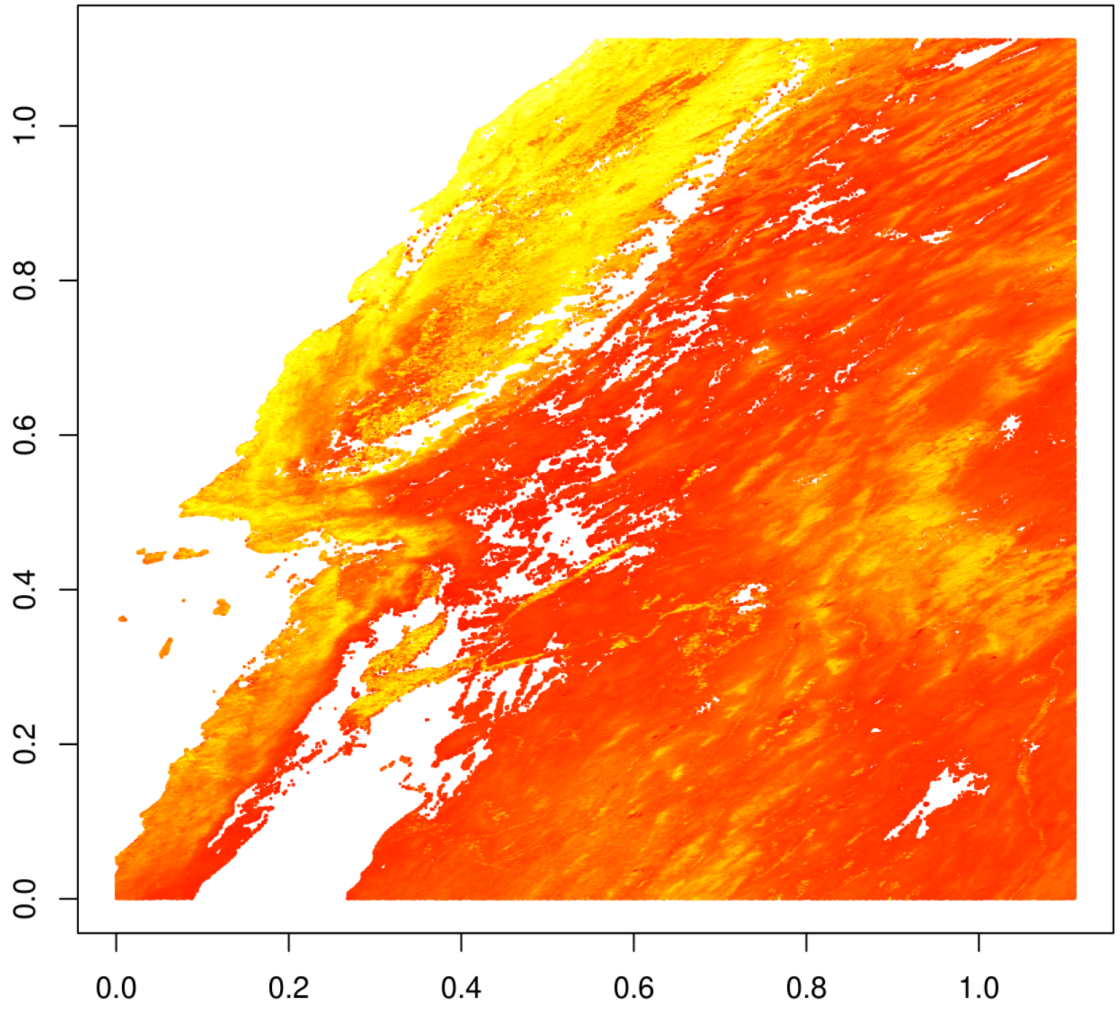
and main roads, where one is probably more likely to have some “bad luck” and dig up a punctually contaminated site, such as an industrial wasteland, who does not represent the overall contamination of the region.



7.2 NDVI with anisotropy and heteroskedastic noise

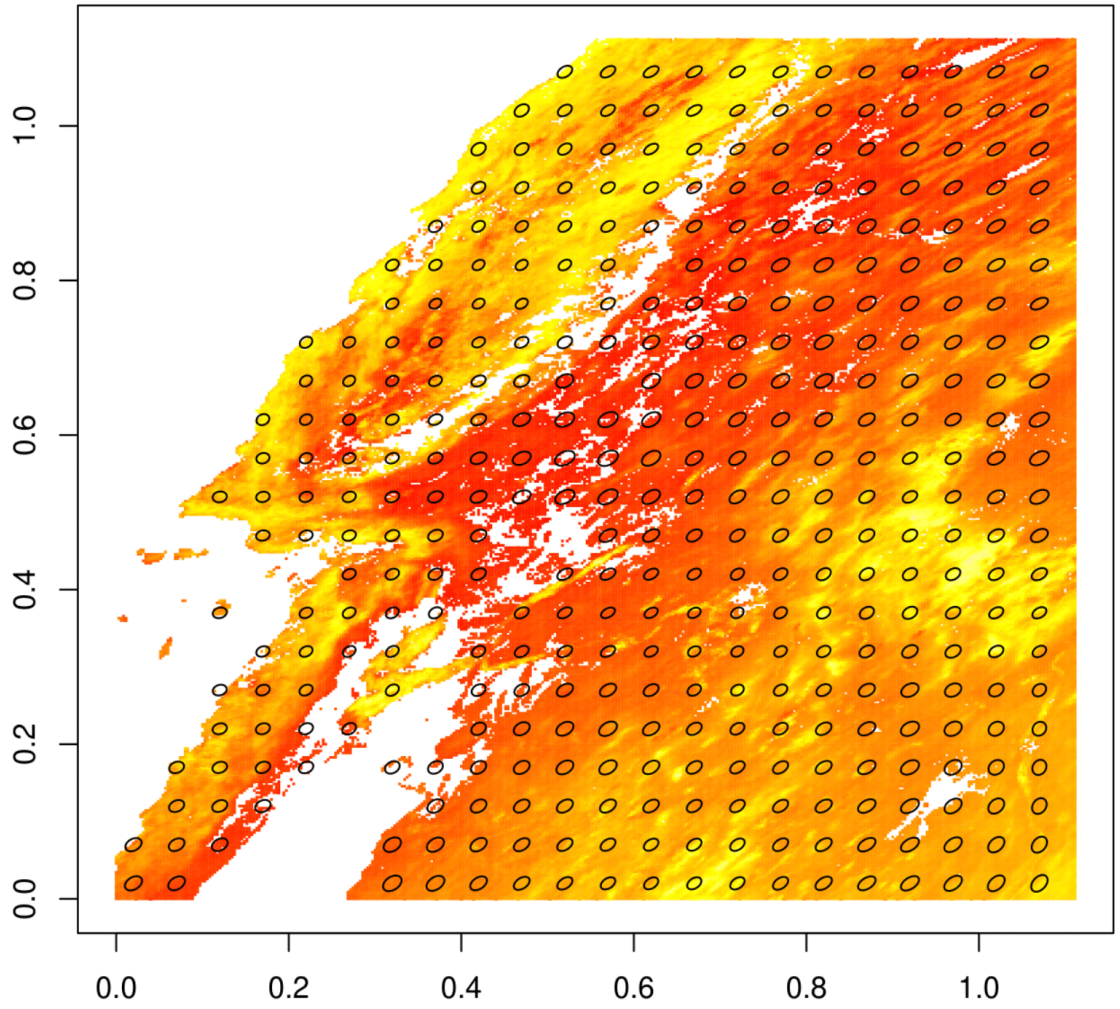
NDVI data from Spatial factor modeling: A Bayesian matrix-normal approach for misaligned data (https://arxiv.org/abs/2006.00595) presents interesting anisotropy and heteroskedasticity patterns. It features over **one million observations**, but a **coordinate rounding** allowed to use all of the observations while reducing the number of distinct spatial locations to one hundred thousand. After a **train-test split** (90% of the observations being used for training, the test observations being selected using a **max-min heuristic**), several models were tested, using several combinations of stationary and nonstationary noise and range. The model with nonstationary anisotropic range and nonstationary noise was selected.

Let’s have a look at the raw data.



We see that some regions are **smooth and noiseless**, while other like the left-hand side are **fuzzy**. We also see that some regions, like the upper-right corner, present clear **anisotropy**, while other do not.

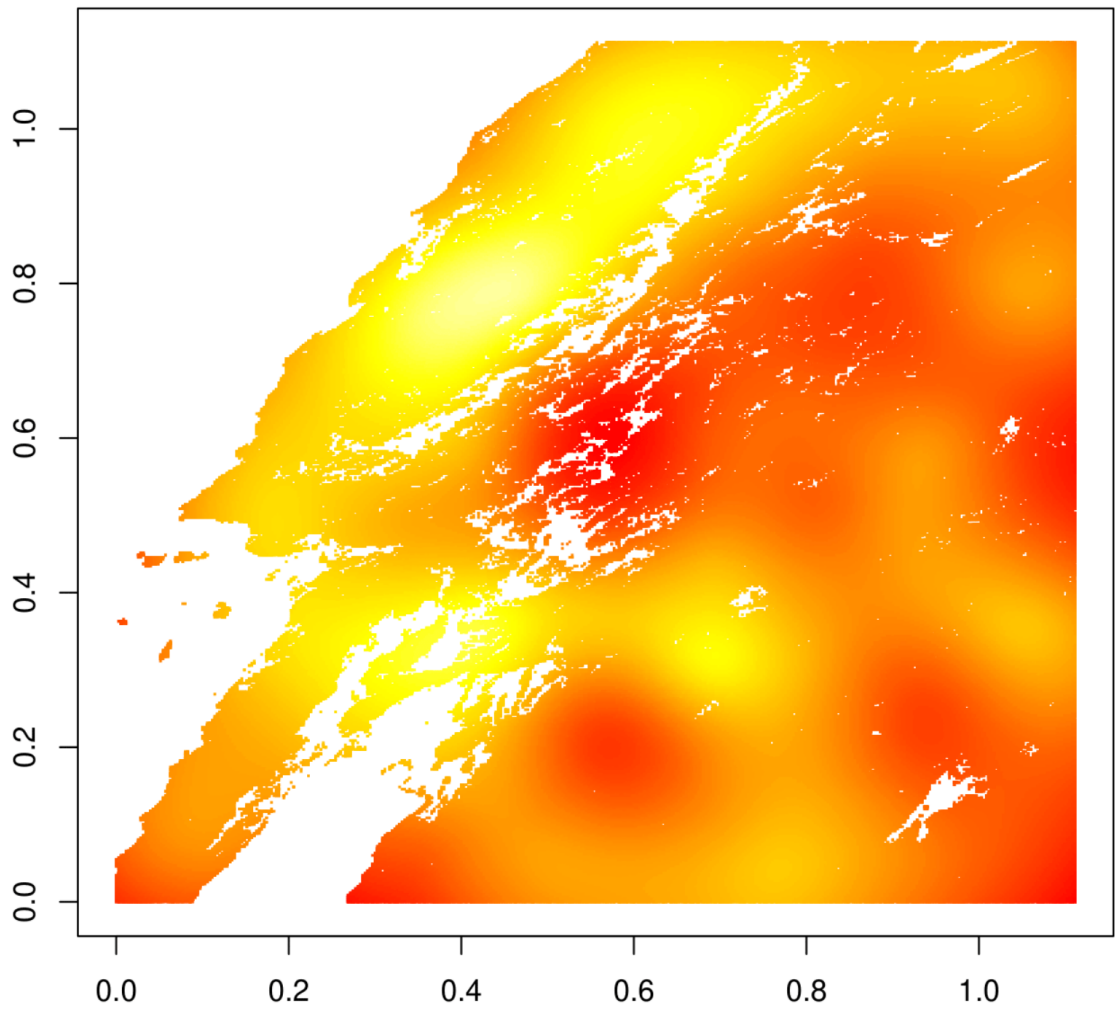
The anisotropic model allows to retrieve the **range ellipses**. We represent them, overlaid on the **smoothed latent field**.



The ellipses do **adapt to the terrain**. In the upper right corner, they are tight and oriented along the field's anisotropy. In the sooth regions, they are big and quite round. In the troubled region of the left, they are much smaller.

The **mean predicted noise variance** also has its interest:

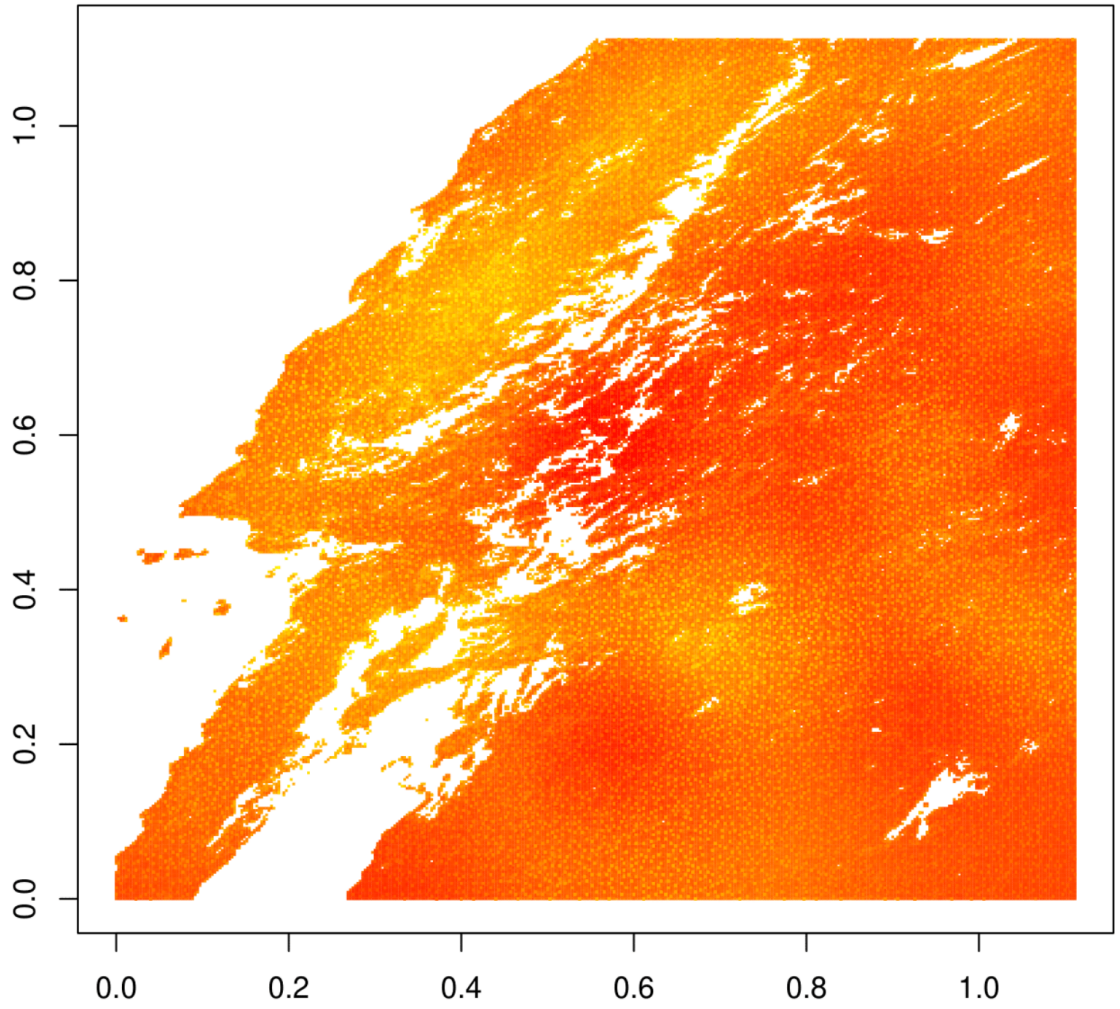
```
img = (png::readPNG("NDVI/noise-1.png"))
plot(1,1)
grid::grid.raster(img)
title(main = "Log noise variance")
```



Clearly, the peaceful regions have low noise, while the fuzzy zones have a high noise.

There is a strong connection of the noise with the **smoothing and prediction standard deviation**.

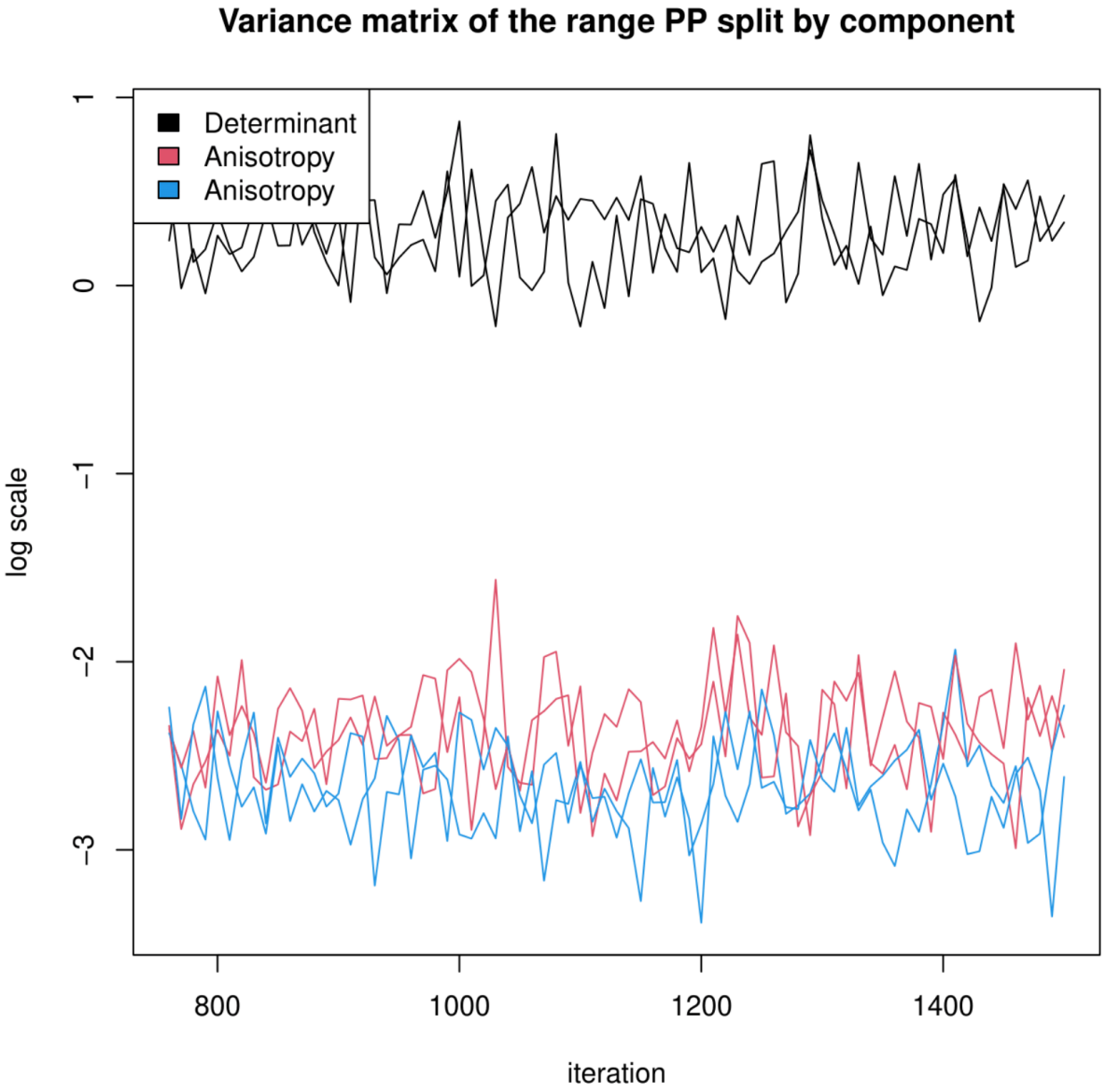
```
img = (png::readPNG("NDVI/log_sd-1.png"))
plot(1,1)
grid::grid.raster(img)
title(main = "Log smoothing and prediction standard deviation")
```



The **bright dots** correspond to the **test locations**, who have **higher variance**. We can see that the standard deviation **follows the noise variance**. The heteroskedastic model allows to **predict confidently where the observations are reliable**, and to **predict cautiously where the observations are fuzzy**.

An interesting point is the interpretation of the **variance of the PP** for the **anisotropic range parameters**. The MCMC samples are presented as curves, as detailed in the run on simulated data. Let's have a look at those samples.

```
img = (png::readPNG("NDVI/range_var-1.png"))
plot(1,1)
grid::grid.raster(img)
```



We can see that the component who regulates the range is much higher than the components who regulate the anisotropy. This can be understood if we look at the maps of the interest variable. The range changes a lot, we do have some regions who are very smooth and other who are really fuzzy. However, the anisotropy changes little, there always is a southwest-northwest orientation of the interest field.

7.3 Conclusion of the Gallery.

Do what you want, but **use the nonstationary noise variance**. The other forms of non-stationarity may prove insightful and may improve the model's predictive performance, but marginally with respect to the noise. They can even be **counter-productive without the noise variance**, as "The Overfitter" shows in the lead case study.