How to fit models with species occurrence data using `xsdm`

Daniel C. Reuman[1] and Angel Luís Robles Fernández[1]

[1]Department of Ecology and Evolutionary Biology and Center for Ecological Research, University of Kansas

Abstract

After reading the document entitled "The xsdm model", this document introduces the statistically and computationally sophisticated user to the main functions of the `xsdm` package. The package implements a frequentist analysis of the xsdm model. This document should get the user to the point of maximizing the likelihood of the model and profiling in "easy" cases, with callouts to several troubleshooting documents about what to do when the process does not go smoothly. Alternative fitted models can also be compared by AIC or another criterion. The example worked out here is for a virtual species (i.e., generated data), so we also demonstrate that fitting an xsdm model can recover the true parameters if model mis-specification is not an issue.

1 Introduction to the running example used throughout this document

This manual uses a small built-in example included with the `xsdm` R package to illustrate a baseline workflow. The example contains (i) an occurrence table for a virtual species, and (ii) two environmental `terra` raster time series representing CHELSA-derived bioclimatic variables for a region of interest over 39 years.

Load the data and get oriented to the first bioclimatic variable, which is mean annual temperature in units of 100×°C for 39 years for a particular region of interest. And make it units of °C:

```
library(xsdm)
library(terra)
data("example_1")
bio_1 <- terra::unwrap(example_1$bio01)
class(bio_1)

## [1] "SpatRaster"
## attr(,"package")
## [1] "terra"

dim(bio_1)

## [1] 128 123  39
```

```
mm <- terra::global(bio_1,fun=c("min","max"))
min(mm$min) #global min over all years and all locations

## [1] -154

max(mm$max) #global max

## [1] 2061

bio_1 <- bio_1/100 #Make it degrees C
```

Now load the second environmental variable, which is annual total precipitation, in units of kg/m^2, for the same 39 years and the same region of interest:

```
bio_12 <- terra::unwrap(example_1$bio12)
class(bio_12)

## [1] "SpatRaster"
## attr(,"package")
## [1] "terra"

dim(bio_12)

## [1] 128 123  39

mm <-  terra::global(bio_12,fun=c("min","max"))
min(mm$min) #again the global min, all years and locations

## [1] 45

max(mm$max) #global max

## [1] 1761
```

If the scales of your environmental variables are vastly different, it can cause problems with optimization (see the document "Troubleshooting: Optimization problems related to scaling"). So we change the units for precipitation to make the range of values similar to those for temperature. Now the units are going to be dg/m^2:

```
bio_12 = bio_12/100
mm <-  terra::global(bio_1,fun=c("min","max"))
min(mm$min)

## [1] -1.54

max(mm$max)

## [1] 20.61
```

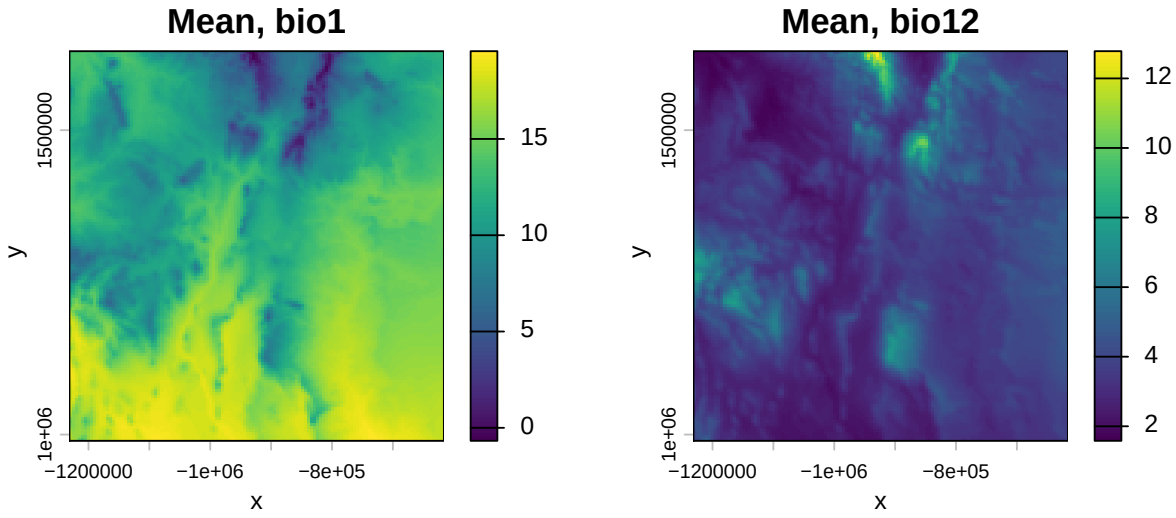
```
mm <- terra::global(bio_12,fun=c("min","max"))
min(mm$min)

## [1] 0.45

max(mm$max)

## [1] 17.61
```

Next load data on detections and non-detections/pseudo-absences for a species:

```
d <- example_1$occ_df
class(d)

## [1] "tbl_df"     "tbl"        "data.frame"

dim(d)

## [1] 2000    4

head(d)

## # A tibble: 6 x 4
##   name               lon      lat presence
##   <chr>            <dbl>    <dbl>    <int>
## 1 Mus virtualis  -918723 1472222.        1
## 2 Mus virtualis  -873723 1517222.        0
## 3 Mus virtualis -1188723 1482222.        1
## 4 Mus virtualis  -928723 1387222.        0
## 5 Mus virtualis -1173723 1532222.        1
## 6 Mus virtualis  -738723 1562222.        1

sum(d$presence == 1)

## [1] 826
```

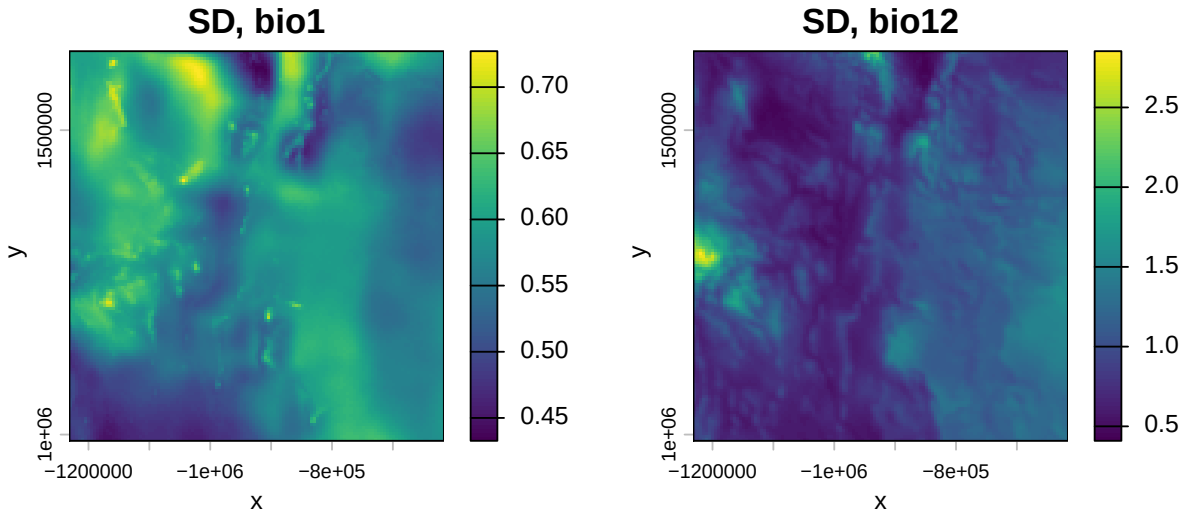
Now take a basic look at the environmental variables and the species data, just to see what we have. Start by getting the means through time of the environmental time series and plotting:

```
m_bio_1 = terra::app(bio_1,mean)
m_bio_12 = terra::app(bio_12,mean)
par(mfrow=c(1,2))
terra::plot(m_bio_1,axes=TRUE,main="Mean, bio1",xlab="x",ylab="y",legend=TRUE)
terra::plot(m_bio_12,axes=TRUE,main="Mean, bio12",xlab="x",ylab="y",legend=TRUE)
```



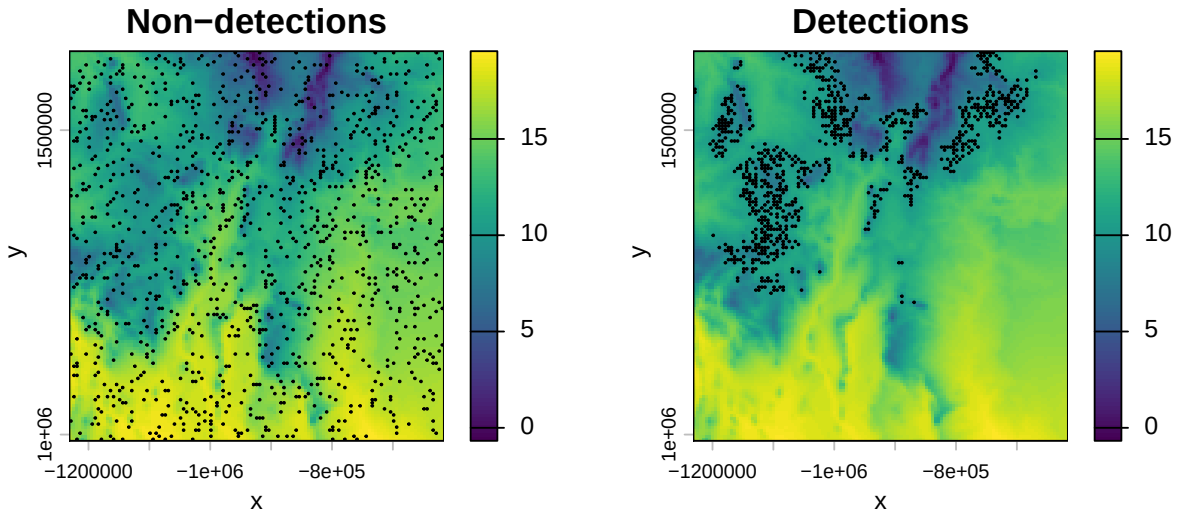
Now do the same for standard deviations:

```
sd_bio_1 = terra::app(bio_1,sd)
sd_bio_12 = terra::app(bio_12,sd)
par(mfrow=c(1,2))
terra::plot(sd_bio_1,axes=TRUE,main="SD, bio1",xlab="x",ylab="y",legend=TRUE)
terra::plot(sd_bio_12,axes=TRUE,main="SD, bio12",xlab="x",ylab="y",legend=TRUE)
```



Now plot the species detections and non-detections/pseudo-absences on a backdrop of the mean of bio 1:

```
d <- as.data.frame(d)
pts_0=terra::vect(d[d$presence==0,],geom=c("lon","lat"),crs=terra::crs(m_bio_1))
pts_1=terra::vect(d[d$presence==1,],geom=c("lon","lat"),crs=terra::crs(m_bio_1))
par(mfrow=c(1,2))
terra::plot(m_bio_1, axes=TRUE,main="Non-detections",xlab="x",ylab="y",legend=TRUE)
terra::plot(pts_0, add=TRUE,col="black",pch=20,cex=.2)
terra::plot(m_bio_1, axes=TRUE,main="Detections",xlab="x",ylab="y",legend=TRUE)
terra::plot(pts_1, add=TRUE,col="black",pch=20,cex=.2)
```



The fitting tools offered by `xsdm` use a more succinct version of the data, since only the environmental time series at the locations for which there is a detection or a non-detection/pseudo-absence matter; though we will return to using rasters after fitting and model selection. So, for now, we cut out only those time series at the species locations and store them in an array using the convenience function `env_data_array`:

```
env_data <-  list(bio_1=bio_1, bio_12=bio_12)  # order defines variable 1 and 2
env_array <- env_data_array(env_data, d)  # (n locations) x (time) x (p vars)
class(env_array)

## [1] "array"

dim(env_array)
```

```
## [1] 2000   39    2
```

And we pull out the presences/pseudo-absences as a binary vector:

```
occ = d$presence
length(occ)

## [1] 2000
```

Now we are ready to think about how the xsdm model applies to our data.

2 Evaluating the likelihood

The xsdm model's likelihood function is described in the document "The xsdm model", where model parameter are also described in detail. Parameters are:

- $\vec{\mu}$, which encodes ideal values for population growth of the two environmental variables (a length-2 vector for the present scenario of two environmental variables, unconstrained values);
- $\tilde{\vec{\sigma}}_L$, which has to do with breadth of the growth-environment function "to the left" with respect to two basis vectors (a length-2 vector for our case, positive entries);
- $\tilde{\vec{\sigma}}_R$, which is similar but "to the right" (a length-2 vector, positive entries);
- $\tilde{c}$ and p_d, which relate to species detection (scalars, $\tilde{c}$ is unconstrained and $0 < p_d \leq 1$); and
- O, which is an orthogonal matrix which describes the basis vectors mentioned above (2×2 for the present example, must be orthogonal, i.e., $OO^\tau = I$ for τ the transpose).

In code, the above parameters are denoted `mu`, `sigltil`, `sigrtil`, `ctil`, `pd`, and `o_mat`.

The likelihood is coded as `loglik_bio`, so as an introduction to the function let's evaluate it at a haphazardly chosen set of parameters:

```
mu <- c(1,1)
sigltil <- c(1,4)
sigrtil <- c(2,3)
ctil <- -10
pd <- 0.8
o_mat <- diag(2)
loglik_bio(env_dat = env_array,
           occ = occ,
           mu = mu,
           sigltil = sigltil,
           sigrtil = sigrtil,
           o_mat = o_mat,
           ctil = ctil,
           pd = pd)

## [1] -1094.157
```

By default, the function gives the log likelihood, but you can also get the linear-scale likelihood:

```
loglik_bio(env_array,occ,mu,sigltil,sigrtil,o_mat,ctil,pd,return_prob=TRUE)

## [1] 0
```

For these haphazardly chosen parameters, the (linear-scale) likelihood is zero to within numeric precision - this is typical. We next optimize.

3 An unconstrained parameter space

Some model parameters are constrained, i.e., only certain values are allowed (specifically, `sigltil`, `sigrtil`, `pd`, and `o_mat`; see previous section). Rather than attempt to perform optimizations subject to these constraints, we here define a transformation from an unconstrained Euclidean space to the space of allowed parameters, and we subsequently optimize on the unconstrained space. Parameters in the unconstrained space are henceforth called "math-scale" parameters, and the constrained parameters described above are called "biological-scale" parameters because they more directly represent biological quantities. When a distinction is needed in mathematical notation, we denote biological-scale parameters with a superscript (b), and math-scale parameters with a superscript (m), i.e., $\vec{\mu}^{(b)}$, $O^{(b)}$, $\tilde{\vec{\sigma}}_L^{(b)}$, $\tilde{\vec{\sigma}}_R^{(b)}$, $\tilde{c}^{(b)}$, and $p_d^{(b)}$ versus $\vec{\mu}^{(m)}$, $O^{(m)}$, $\tilde{\vec{\sigma}}_L^{(m)}$, $\tilde{\vec{\sigma}}_R^{(m)}$, $\tilde{c}^{(m)}$, and $p_d^{(m)}$.

The transformation from math- to biological-scale parameters acts separately on each parameter, as follows for all the parameters except O:

$$\vec{\mu}^{(b)} = \vec{\mu}^{(m)} \tag{1}$$

$$\tilde{\vec{\sigma}}_L^{(b)} = \exp(\tilde{\vec{\sigma}}_L^{(m)}) \tag{2}$$

$$\tilde{\vec{\sigma}}_R^{(b)} = \exp(\tilde{\vec{\sigma}}_R^{(m)}) \tag{3}$$

$$\tilde{c}^{(b)} = \tilde{c}^{(m)} \tag{4}$$

$$p_d^{(b)} = \text{expit}(p_d^{(m)}). \tag{5}$$

Here, exp of a vector is interpreted to be the vector resulting from exp-transforming each component, and expit is the standard logistic sigmoid function $1/(1+\exp(-x))$, i.e., the inverse of the logit function.

The parameter $O^{(m)}$ is interpreted to be an unconstrained Euclidean vector of dimension $\frac{p^2-p}{2}$, where p is the number of environmental variables being considered ($p = 2$ in our running example), and $O^{(b)}$ is obtained from $O^{(m)}$ by using $O^{(m)}$ to form a skew-symmetric matrix of dimensions $p \times p$, and then applying the matrix exponential map to that skew-symmetric matrix. It is known that the matrix exponential of a skew-symmetric matrix is an orthogonal matrix.

The unconstrained space of parameters (the space of math-scale parameters) is thus a Euclidean space of dimensions $3p + \frac{p^2-p}{2} + 2$. The $3p$ term in this expression comes from the parameters $\vec{\mu}^{(m)}$, $\tilde{\vec{\sigma}}_L^{(m)}$, and $\tilde{\vec{\sigma}}_R^{(m)}$; the 2 in this expression comes from the parameters $\tilde{c}^{(m)}$ and $p_d^{(m)}$; and the term $\frac{p^2-p}{2}$ in the expression comes from $O^{(m)}$.

The transformation from math-scale to bio-scale parameters is implemented in `xsdm` using the function `math_to_bio`, which takes an unconstrained numeric vector as its argument and returns a named list of of biological-scale parameters. But it is not so common for the end-user to call

`math_to_bio` directly because we have written the likelihood directly in terms of the math-scale parameters in a function `loglik_math`. We now demonstrate `math_to_bio` and `loglik_math`:

```
#loglik_bio and math_to_bio require a named argument to help prevent errors -
#see below - make_mask_names helps set it up
param_vector = make_mask_names(p = 2) #p = 2 environmental variables in our case
param_vector

##      mu1      mu2 sigltil1 sigltil2 sigrtil1 sigrtil2     ctil       pd   o_par1
##       NA       NA       NA       NA       NA       NA       NA       NA       NA

set.seed(101)
param_vector[1:9] <-  rnorm(9) #fill with random values just to try it
math_to_bio(param_vector)

## $mu
## [1] -0.3260365  0.5524619
##
## $sigltil
## [1] 0.509185 1.239068
##
## $sigrtil
## [1] 1.364474 3.234797
##
## $ctil
## [1] 0.6187899
##
## $pd
## [1] 0.4718462
##
## $o_mat
##           [,1]       [,2]
## [1,] 0.6081818 -0.7937978
## [2,] 0.7937978  0.6081818

loglik_math(param_vector, env_dat = env_array, occ = occ, negative = FALSE)

## [1] -26974.87
```

The function `loglik_math` first transforms parameters to the biological scale using `math_to_bio` and then evaluates `loglik_bio`; so one optimizes it directly (see below). If your favorite optimizer minimizes by default, you can use:

```
loglik_math(param_vector, env_dat = env_array, occ = occ, negative = TRUE)

## [1] 26974.87
```

Before moving on to optimizing, we note that `loglik_math` requires a named vector for its input `param_vector`. This is to reduce the possibility of errors stemming form parameter ordering. There

is a naming convention for arguments which must be followed exactly, and which is described in the documentation for `loglik_math`. The function `make_mask_names` helps the user by giving the parameter names which are required for a model making use of a given number of environmental variables.

4 Optimizing the likelihood

Maximizing the likelihood from one initial parameter guess is straightforward using the built-in R function `optim` (and a variety of other optimizers are also available):

```
optim(par = param_vector,
      fn = loglik_math,
      method = "BFGS",
      env_dat = env_array,
      occ = occ,
      negative = TRUE,
      control = list(trace=100))

## initial  value 26974.871317
## iter  10 value 654.468181
## iter  20 value 593.148121
## iter  30 value 592.493219
## final  value 591.792577
## converged
## $par
##        mu1        mu2   sigltil1   sigltil2   sigrtil1   sigrtil2       ctil         pd
##  9.8333383  3.6633597 -0.3416438 30.4804068 52.6023010 -1.1416818 -4.7067112  1.7022645 -7.6
##
## $value
## [1] 591.7926
##
## $counts
## function gradient
##      108       39
##
## $convergence
## [1] 0
##
## $message
## NULL
```

But multiple optimizations should typically be done starting from different initial conditions to improve chances of finding the global maximum to the likelihood function.

The function `start_parms` can be used to find plausible initial conditions for optimization:

```
num_starts <- 10
starts <- start_parms(env_array[occ == 1, , ], num_starts = num_starts)
starts
```

```
## # A tibble: 10 x 9
##      mu1   mu2 sigltil1 sigltil2 sigrtil1  sigrtil2   ctil     pd o_par1
##    <dbl> <dbl>    <dbl>    <dbl>    <dbl>     <dbl>  <dbl>  <dbl>  <dbl>
##  1  8.67  3.39 -0.256     0.139   -0.0325 -0.217    -1.46   1.92   -5.89
##  2  9.93  4.69  0.438    -0.554   -0.726   0.476    -0.992 -0.275   3.53
##  3 10.6   2.73  0.0910    0.485   -0.379   0.129    -0.759  0.824  -1.18
##  4  9.30  4.04 -0.602    -0.208    0.314   0.823    -1.23  -1.37    8.25
##  5  8.99  3.06  0.611    -0.0346  -0.206   0.649    -1.34  -0.824  -8.25
##  6 10.2   4.36 -0.0823   -0.728    0.487  -0.0439   -0.875  1.37    1.18
##  7  9.62  3.71 -0.429     0.312    0.141   0.303    -0.642 -1.92   -3.53
##  8  8.36  5.02  0.264    -0.381   -0.552  -0.390    -1.11   0.275   5.89
##  9  8.44  3.79  0.481    -0.251   -0.422  -0.000587 -1.02   1.51    5.30
## 10  9.54  3.69  0.00438  -0.121   -0.119   0.216    -0.929  0       0
```

We recommend, for real data, at least 50 initial conditions for optimization, or more if using more than two environmental variables or if warranted after running 50 (see below). But for this exercise we use only 10 to keep run times low. We now optimize from all the start parameters:

```
all_optim_results <- list()
for (counter in 1:(dim(starts)[1]))
{
  all_optim_results [[counter]] = optim(par = starts[counter,],
                                        fn = loglik_math,
                                        method = "BFGS",
                                        env_dat = env_array,
                                        occ = occ,
                                        negative = TRUE,
                                        control=list(trace=0))
}
```

We now want to judge whether it is sufficiently likely that we succeeded in finding the global maximum. One can never be certain, here, but there are various checks that one can use to help assess this. We consider it more likely that the global maximum was located if multiple initial conditions spread widely across parameter space all converged to the same maximized likelihood and the same parameters. So we next assess this based on the optimization results we achieved.

First, we look at the maximized likelihood values:

```
bestlogliks <- sapply(X=all_optim_results,FUN=function(x){x$value})
convergence <- sapply(X=all_optim_results,FUN=function(x){x$convergence})
table(convergence)
```

```
## convergence
##  0
## 10
```

```
inds <- order(bestlogliks)
bestlogliks <- bestlogliks[inds]
bestlogliks
```

```
##  [1] 503.5279 503.5279 503.5279 503.5279 503.5279 503.5279 503.5279 546.4126 714.6485 714.8
```

These results indicate that: 1) most of the 10 optimizations appear to have converged according to the diagnostics of `optim` (0 means convergence for `optim`); and 2) five of the 10 optimizations returned the same, highest log-likelihood to within 3 digits. This, already, is pretty good evidence that multiple optimizations arrived at the same place in parameter space, i.e., the same maximum-likelihood parameters - it would be unusual for two distinct local maxima of the log-likelihood function to have the same height. But we can also check this directly, which is what we do next.

The parameters resulting from our optimizations are:

```
all_optim_results <- all_optim_results[inds]
allpars = sapply(X = all_optim_results, FUN = function(x){x$par})
allpars
```

```
##                [,1]       [,2]       [,3]       [,4]       [,5]       [,6]       [,7]
## mu1       8.9357485  8.9356162  8.9357468  8.9361168  8.9361986  8.9354299  8.9369092 10.654
## mu2       2.5582086  2.5585475  2.5581811  2.5584391  2.5582726  2.5590545  2.5583232  1.923
## sigltil1 -1.4990590 -0.6448095 -1.3575361 -1.4988606 -1.3571566 -0.3092466 -0.3083641 -2.493
## sigltil2 -0.6447884 -0.3091843 -1.4990567 -0.6449561 -1.4988933 -1.3573197 -1.3567490 -0.810
## sigrtil1 -0.3090746 -1.3574434 -0.6447486 -0.3091322 -0.6447920 -1.4983423 -1.4983468  7.038
## sigrtil2 -1.3575403 -1.4987893 -0.3090375 -1.3573319 -0.3088888 -0.6447939 -0.6446483 -0.825
## ctil     -9.9378167 -9.9376666 -9.9373281 -9.9379997 -9.9346303 -9.9360207 -9.9267505 -4.399
## pd        1.8853819  1.8854017  1.8854125  1.8853318  1.8855176  1.8854368  1.8857163  2.059
## o_par1    1.8734093 -2.8390725 -5.9805653 -4.4096644  0.3027231 -1.2684431  5.0153973  2.203
##                [,10]
## mu1       10.4865033
## mu2      -28.7069173
## sigltil1  -1.2842199
## sigltil2  14.9389851
## sigrtil1  67.8206490
## sigrtil2   2.9164143
## ctil      -7.8856021
## pd         0.9305728
## o_par1     3.1635152
```

These are in the same order as the maximized likelihood values above. Note that the first five sets of optimized parameters appear the same, to within a few digits, for the `mu1`, `mu2`, `ctil`, and `pd` parameters, but that they appear to differ from each other with respect to the `o_par1` parameter. And the `sigltil1`, `sigltil2`, `sigrtil1`, `sigrtil2` parameters for one optimization result appear to be the same, up to a few digits, as for another optimization results *if permuted*. These complexities reflect the fact that the `math_to_bio` mapping is many-to-one, and that there are also multiple ways to parameterize the identical xsdm model with distinct biological-scale parameters. These redundancies affect the `sigltil1`, `sigltil2`, `sigrtil1`, `sigrtil2`, and `o_par1` parameters. For models making use of more than two environmental variables, all the `o_par` parameters are affected. In essence, parameters can be the same, in the sense of giving the same xsdm model, even if they appear different; so we need to take this redundancy into account when judging whether different optimizations resulted in the same parameters.

These issues are described further in the document "Troubleshooting: Dealing with parameter redundancy," but the function `distance_between_params_hungarian` provides an easy way around these complexities. The function directly calculates for the user the distance between

two sets of xsdm model parameters while taking into account the redundancy described; i.e., if `distance_between_params_hungarian` indicates a very small difference between two sets of parameters, they give essentially the same xsdm model and can be considered to be close to each other in parameter space even if they appear different. So use the function as the test of parameter similarity, as follows:

```
dists_to_first <-  NA*numeric(num_starts)
for (counter in 1:num_starts)
{
  dists_to_first[counter] =
    dist_between_params(
      all_optim_results[[1]]$par,
      all_optim_results[[counter]]$par
    )
}
dists_to_first

##  [1]  0.000000000  0.001339857  0.000498297  0.001341510  0.003636068  0.003906200  0.01203
##  [9]  7.499664690 31.748437980
```

Note that the first five parameter optimization results are all reported to be close in parameter space to the first parameter optimization result.

If, as in this case, sufficiently many of the initial conditions optimized to give the same, highest maximized likelihood, and these also gave parameter results which are reported using `distance_between_params_hungarian` to be close to each other in parameter space, we can be sufficiently confident that we have successfully maximized the likelihood. Profiling, which is covered below, provides additional checks.

5 Comparing alternative models

It is straightforward to compare multiple models making use of different combinations of environmental variables, using AIC (or another criterion). We demonstrate how to use xsdm to do that by comparing the previous, two-environmental-variable model with each of the two simpler models that make use of just one of the environmental variables. First, fit the two simpler models:

```
#Fit the two simpler models
starts_1 <- start_parms(env_array[occ==1,,1,drop=FALSE],num_starts=num_starts)
starts_2 <- start_parms(env_array[occ==1, , 2,drop=FALSE],num_starts=num_starts)
all_optim_results_1 = list()
all_optim_results_2 = list()
for (counter in 1:num_starts)
{
  all_optim_results_1[[counter]]=optim(par=starts_1[counter,],
                                       fn=loglik_math,
                                       method="BFGS",
                                       env_dat=env_array[,,1,drop=FALSE],
                                       occ=occ,
                                       negative=TRUE,
```

```
                                        control=list(trace=0))
  all_optim_results_2[[counter]]=optim(par=starts_2[counter,],
                                        fn=loglik_math,
                                        method="BFGS",
                                        env_dat=env_array[,,2,drop=FALSE],
                                        occ=occ,
                                        negative=TRUE,
                                        control=list(trace=0))
}
```

Now examine results of the first of the two simpler models:

```
#Examine results from the first simpler model
convergence_1 = sapply(X=all_optim_results_1,FUN=function(x){x$convergence})
table(convergence_1)

## convergence_1
##  0
## 10

bestlogliks_1 = sapply(X=all_optim_results_1,FUN=function(x){x$value})
inds = order(bestlogliks_1)
bestlogliks_1 = bestlogliks_1[inds]
bestlogliks_1

##  [1] 567.2883 573.7785 573.7785 573.7785 573.7785 573.7785 573.7785 573.7785 573.7785 573.7

#Look at distance in parameter space
all_optim_results_1 = all_optim_results_1[inds]
dists_to_first_1 = NA*numeric(num_starts)
for (counter in 1:num_starts)
{
  dists_to_first_1[counter] =
    dist_between_params(all_optim_results_1[[1]]$par,
                                all_optim_results_1[[counter]]$par)
}
dists_to_first_1

##  [1]   0.0000 166.2141 166.2139 166.2139 166.2139 166.2139 166.2126 166.2132 166.2132 166.2
```

The results suggest that the likelihood function of this model has been adequately maximized.

Now examine results of the second of the two simpler models:

```
#Examine results from the second simpler model
convergence_2 = sapply(X=all_optim_results_2,FUN=function(x){x$convergence})
table(convergence_2)

## convergence_2
```

```
##  0
## 10

bestlogliks_2 = sapply(X=all_optim_results_2,FUN=function(x){x$value})
inds = order(bestlogliks_2)
bestlogliks_2 = bestlogliks_2[inds]
bestlogliks_2

##  [1] 1147.048 1147.049 1147.049 1147.049 1147.050 1147.050 1147.051 1147.051 1147.051 1147.0

#Look at distances in parameter space
all_optim_results_2 = all_optim_results_2[inds]
dists_to_first_2 = NA*numeric(num_starts)
for (counter in 1:num_starts)
{
  dists_to_first_2[counter] =
    dist_between_params(all_optim_results_2[[1]]$par,
                                all_optim_results_2[[counter]]$par)
}
dists_to_first_2

##  [1] 0.000000e+00 4.858290e-04 2.111693e-04 7.880041e-04 1.550861e-04 8.569218e-05 3.677727e
##  [9] 1.323945e-03 2.916366e-04
```

The results suggest that the likelihood function of this model has been adequately maximized.

Now compute the AIC of each model, bearing in mind that optimization results are already negative log-likelihoods:

```
#AIC of the 2-environmental variable model, AIC = 2*(number of parameters)-
#2*(maximized log likelihood)
AIC <- 2*length(make_mask_names(2))+2*bestlogliks[1]
AIC

## [1] 1025.056

#AICs of the two simpler models
AIC_1 <- 2*length(make_mask_names(1))+2*bestlogliks_1[1]
AIC_1

## [1] 1144.577

AIC_2 <- 2*length(make_mask_names(1))+2*bestlogliks_2[1]
AIC_2

## [1] 2304.096
```

The best model (lowest AIC) is the two-environmental-variable model, by a considerable margin. This confirms expectation, since the data come from a virtual species, and we know they were generated using both of the two environmental variables (see "Virtual species and habitat suitability maps" for how the data were generated).

6 Profiling

Profiles are the means by which confidence intervals are constructed, as well as providing other information about the likelihood surface and hence about the nature of the correspondence between the model and the data. Having obtained maximum-likelihood parameters $\hat{\theta}$ for a log-likelihood function $\mathcal{L}(\theta)$, a profile on the ith parameter is obtained by maximizing $\mathcal{L}(\theta)$, subject to the constraint $\theta_i = x$, for each value of x spanning a range surrounding $\hat{\theta}_i$, and then plotting the resulting maxima against x. Standard use of the likelihood function to formulate confidence intervals and compare models requires that the likelihood function is "well-behaved" near its maximum, which can typically be judged by whether the profiles are "dome shaped", i.e., they look approximately like a downward-opening parabola. In that case, confidence intervals can be constructed by use of the likelihood ratio test - see basic texts on maximum likelihood methods for additional details.

To calculate profiles using `xsdm`, use the function `profile_likelihood`, here for mu1:

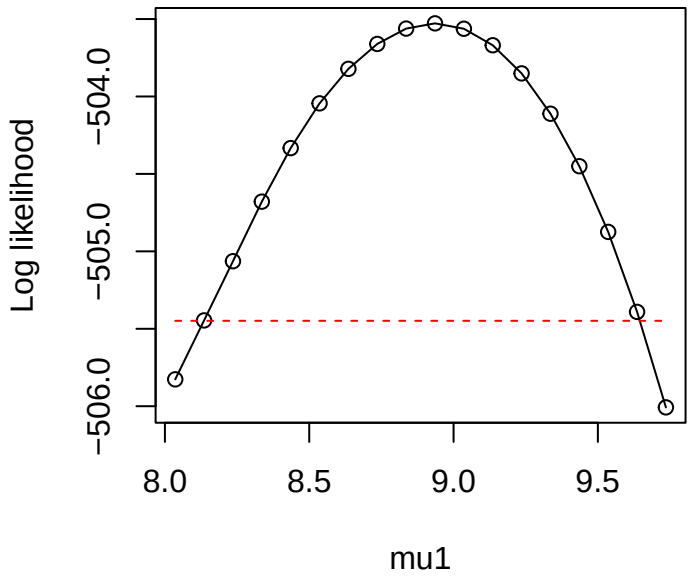
```
prof1 = profile_likelihood(
  profile_parameter = "mu1",
  increment_left = 0.1,
  increment_right = 0.1,
  num_steps_left = 30,
  num_steps_right = 30,
  alpha = 0.95,
  optim_param_vector = all_optim_results[[1]]$par,
  env_dat = env_array,
  occ = occ,
  num_threads=7
)
names(prof1)

## [1] "profile"      "found_better" "threshold"    "parameters"

head(prof1$profile)

##    param value_math    loglik convergence
## 10   mu1   8.035749 -505.8266           1
## 9    mu1   8.135749 -505.4460           1
## 8    mu1   8.235749 -505.0636           1
## 7    mu1   8.335749 -504.6790           1
## 6    mu1   8.435749 -504.3334           1
## 5    mu1   8.535749 -504.0444           1
```

Note that the convergence column is returning the convergence flagged returned by the optimizer `ucminf::ucminf`; the value 1 is among the values which indicate convergence to within the tolerances used.

Two of the outputs of this function contain the profile itself, and the threshold. The region for which the profile is above the threshold is the confidence intervals:

```
plot(prof1$profile$value_math,
     prof1$profile$loglik,
```

```
     type="o",
     xlab = "mu1",
     ylab = "Log likelihood")
lines(range(prof1$profile$value_math),rep(prof1$threshold,2),type="l",
                                          lty="dashed",col="red")
```



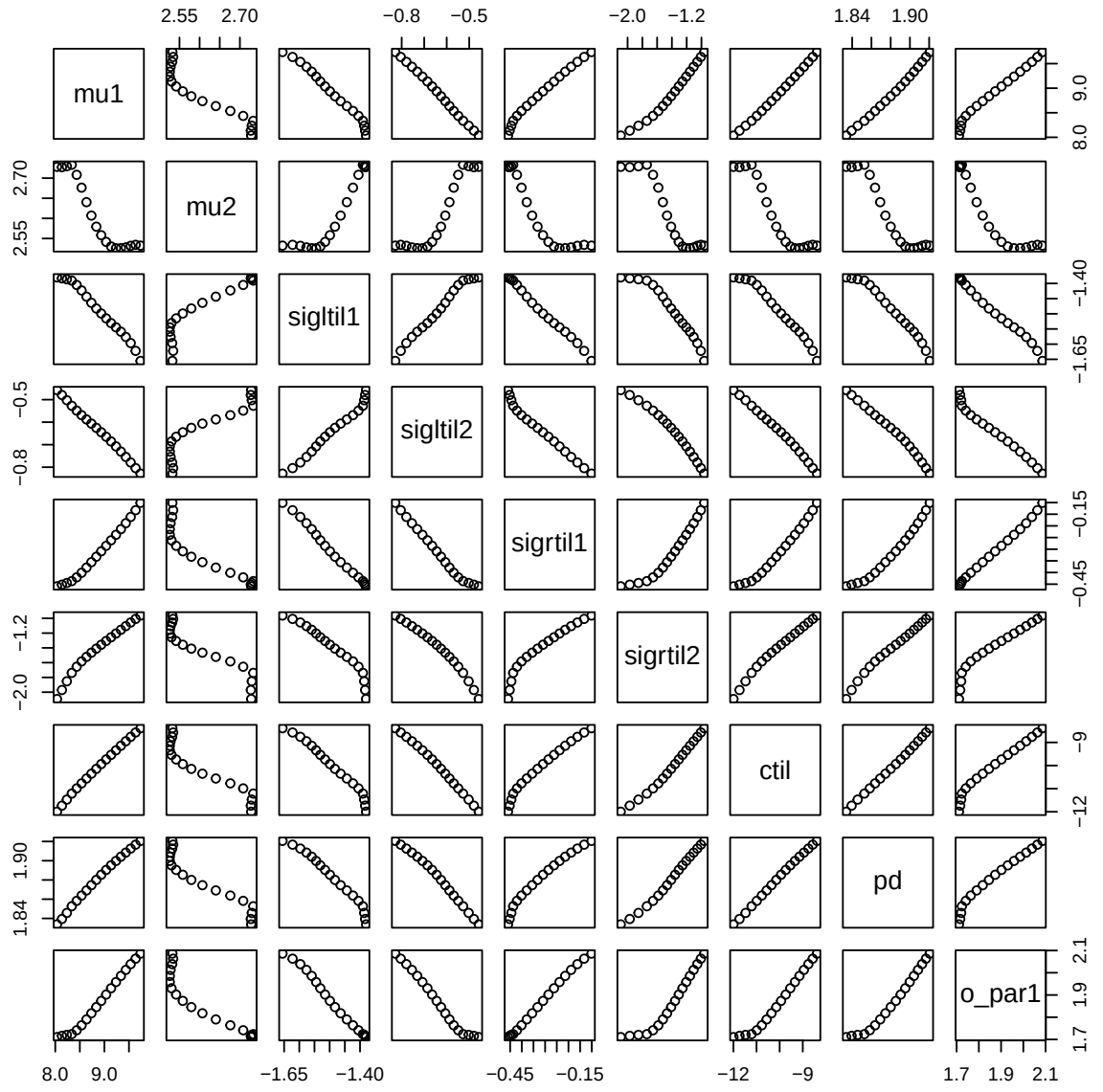
Note that the profiler stops its leftward (respectively, rightward) progress when `num_steps_left` (resp., `num_steps_right`) is exceeded, or when the threshold is crossed, whichever happens first. Profiling is often a trial-and-error process of selecting values for `increment_left`, `increment_right`, `num_steps_left`, and `num_steps_right` to get complete and smooth profiles within the limits of available computational resources. See "Troubleshooting: Profiles" for additional details.

Often one wants to plot the values of the other parameters which optimized the likelihood for each value of the profiled parameter. This can be done using the `parameters` output of `profile_likelihood`:

```
head(prof1$parameters)
```

```
##        mu1      mu2  sigltil1   sigltil2   sigrtil1  sigrtil2      ctil       pd   o_par1
## 1 8.035749 2.728056 -1.380234 -0.4576563 -0.4592180 -2.092792 -11.99685 1.833795 1.710613
## 2 8.135749 2.727185 -1.383251 -0.4793240 -0.4520287 -1.970740 -11.73465 1.839424 1.716190
## 3 8.235749 2.729540 -1.386312 -0.5024554 -0.4448233 -1.852615 -11.47264 1.845705 1.720519
## 4 8.335749 2.733290 -1.390543 -0.5271538 -0.4369405 -1.738965 -11.21248 1.852656 1.723797
## 5 8.435749 2.708252 -1.405023 -0.5486463 -0.4207695 -1.655509 -10.98869 1.858264 1.741356
## 6 8.535749 2.676327 -1.423081 -0.5689028 -0.4011953 -1.583665 -10.77199 1.863767 1.763646
```

```
pairs(prof1$parameters)
```



As a reminder, these are math-scale parameters.

Finally, the `found_better` output of `profile_likelihood` is a flag which tells you whether, in the course of profiling, a higher likelihood was found than what was previously believed to be the maximum likelihood:

```
prof1$found_better

## [1] FALSE
```

In this case, a better value was not found, which provides additional evidence that we had previously already succeeded in adequately maximizing the likelihood. If `found_better` is `TRUE`, it means you have to go back and re-optimize, either with more initial conditions, or with tighter tolerances on the optimizer used, or with a different optimizer. Although sometimes it can be sufficient to simply re-start profiling using the parameters which were found to yield a new highest likelihood.

We next produce and plot all the profiles for our AIC-best model, where now the profiles are going to be plotted on the biological scale. Since the data are for a virtual species, they were generated with known true parameters. We also plot the true parameter values together with the profiles, to verify that the fitting process has accurately recovered the true parameters. This exercise is complicated by the redundancy, discussed above, of parameters. See below for details of how to resolve that redundancy. First make the profiles:

```
pnames <- names(make_mask_names(2))
all_profiles <- list()
for (counter in 1:length(pnames))
{
  all_profiles[[counter]] = profile_likelihood(
                              profile_parameter=pnames[counter],
                              increment_left =0.05,
                              increment_right = 0.05,
                              num_steps_left = 50,
                              num_steps_right = 50,
                              alpha = 0.95,
                              optim_param_vector = all_optim_results[[1]]$par,
                              env_dat = env_array,
                              occ = occ,
                              num_threads = 7
                            )
}
names(all_profiles) <- pnames
```

Now, to deal with the parameter redundancy, convert the maximum-likelihood parameters to the biological scale and find the representation of the true parameters which is closest in parameter space to the maximum-likelihood parameters:

```
example_1_true_parameters_bio = list(
  mu=c(9,2.5),
  sigltil=c(0.3,0.2),
  sigrtil=c(0.5,0.8),
  ctil=-9,
  pd=0.8649783,
  o_mat=matrix(c(0.9800666,0.1986693,-0.1986693,0.9800666),2,2)
) #DAN: Note, this needs to be embedded in the package
ML_parameters_bio = math_to_bio(all_optim_results[[1]]$par)
example_1_true_parameters_bio <- dist_between_params(example_1_true_parameters_bio,
                                                     ML_parameters_bio,
                                                     give_closest_rep = TRUE)$representative
example_1_true_parameters_bio

## $mu
## [1] 9.0 2.5
##
## $sigltil
```

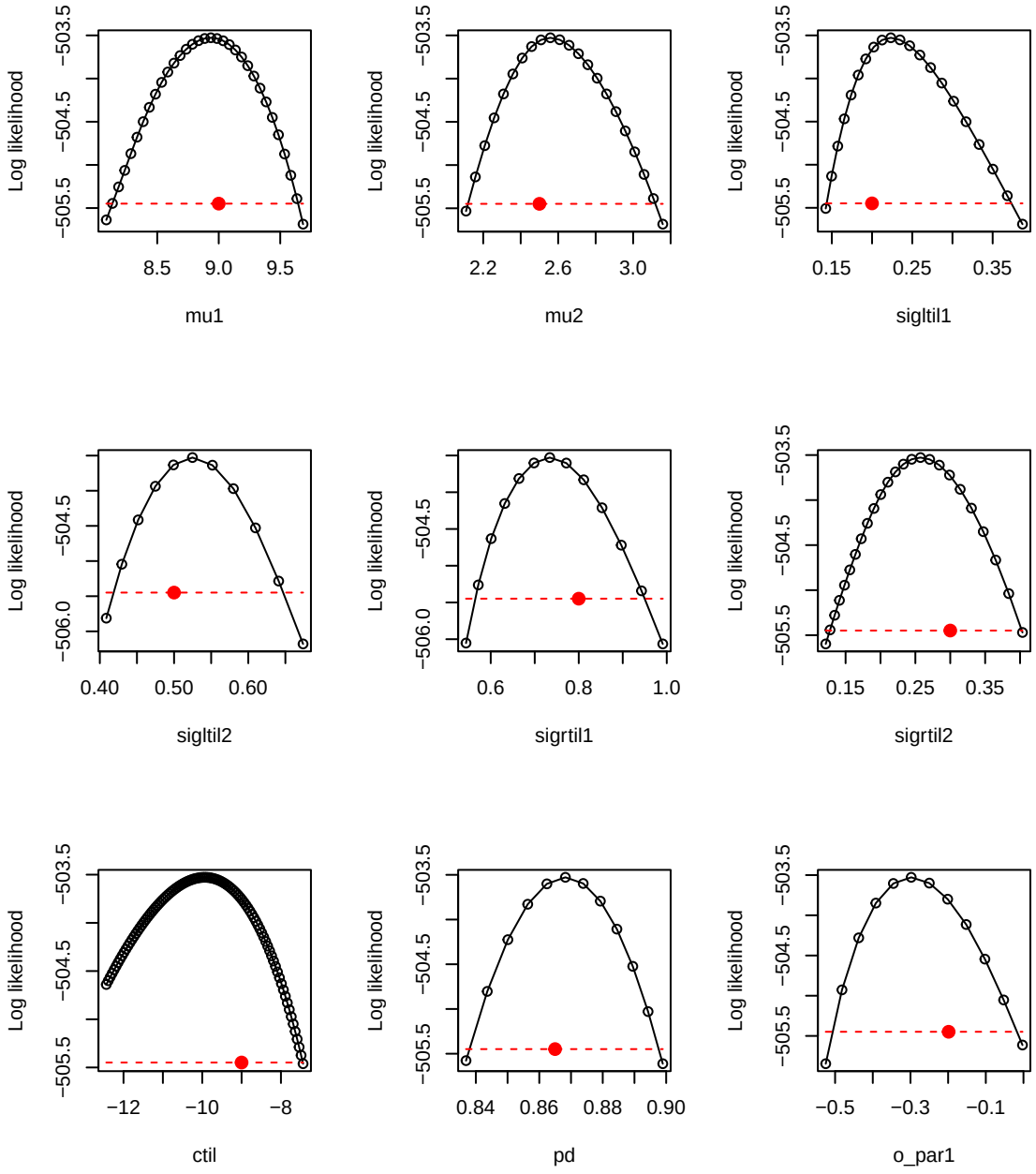
```
## [1] 0.2 0.5
##
## $sigrtil
## [1] 0.8 0.3
##
## $ctil
## [1] -9
##
## $pd
## [1] 0.8649783
##
## $o_mat
##            [,1]       [,2]
## [1,] -0.1986693 -0.9800666
## [2,]  0.9800666 -0.1986693
```

Now plot profiles. Recall we are plotting on the biological scale, so each parameter is transformed before the profile is plotted:

```
par(mfrow=c(3,3))

plot_tool=function(x,y,thresh,xlab,true_param)
{
  plot(x,y,type="o",xlab=xlab,ylab="Log likelihood")
  lines(range(x),rep(thresh,2),type="l",
        lty="dashed",col="red")
  points(true_param,thresh,
        col="red",pch=20,cex=2)
}

#mu1
plot_tool(x = all_profiles$mu1$profile$value_math,
  y = all_profiles$mu1$profile$loglik,
  thresh = all_profiles$mu1$threshold,
  xlab = "mu1",
  true_param = example_1_true_parameters_bio$mu[1])

#mu2
plot_tool(x = all_profiles$mu2$profile$value_math,
  y = all_profiles$mu2$profile$loglik,
  thresh = all_profiles$mu2$threshold,
  xlab = "mu2",
  true_param = example_1_true_parameters_bio$mu[2])

#sigltil1 - need to exp transform to get to the bio scale
plot_tool(x = exp(all_profiles$sigltil1$profile$value_math),
  y = all_profiles$sigltil1$profile$loglik,
```

```
  thresh = all_profiles$sigltil1$threshold,
  xlab = "sigltil1",
  true_param = example_1_true_parameters_bio$sigltil[1])

#sigltil2 - need to exp transform to get to the bio scale
plot_tool(x = exp(all_profiles$sigltil2$profile$value_math),
  y = all_profiles$sigltil2$profile$loglik,
  thresh = all_profiles$sigltil2$threshold,
  xlab = "sigltil2",
  true_param = example_1_true_parameters_bio$sigltil[2])

#sigrtil1 - need to exp transform to get to the bio scale
plot_tool(x = exp(all_profiles$sigrtil1$profile$value_math),
  y = all_profiles$sigrtil1$profile$loglik,
  thresh = all_profiles$sigrtil1$threshold,
  xlab = "sigrtil1",
  true_param = example_1_true_parameters_bio$sigrtil[1])

#sigrtil2 - need to exp transform to get to the bio scale
plot_tool(x = exp(all_profiles$sigrtil2$profile$value_math),
  y = all_profiles$sigrtil2$profile$loglik,
  thresh = all_profiles$sigrtil2$threshold,
  xlab = "sigrtil2",
  true_param = example_1_true_parameters_bio$sigrtil[2])

#ctil
plot_tool(x = all_profiles$ctil$profile$value_math,
  y = all_profiles$ctil$profile$loglik,
  thresh = all_profiles$ctil$threshold,
  xlab = "ctil",
  true_param = example_1_true_parameters_bio$ctil)

#pd - need to expit transform to get to the bio scale
plot_tool(x = expit(all_profiles$pd$profile$value_math),
  y = all_profiles$pd$profile$loglik,
  thresh = all_profiles$pd$threshold,
  xlab = "pd",
  true_param = example_1_true_parameters_bio$pd)

#o_mat - Note that this is a single parameter on the math scale, so we only
#need to profile one of the matrix entries on the biological scale
x = sapply(X=all_profiles$o_par1$profile$value_math,
           FUN=function(x){build_orthogonal_matrix(x)[1,1]})
plot_tool(x = x,
  y = all_profiles$o_par1$profile$loglik,
  thresh = all_profiles$o_par1$threshold,
  xlab = "o_par1",
```

```
true_param = example_1_true_parameters_bio$o_mat[1,1])
```

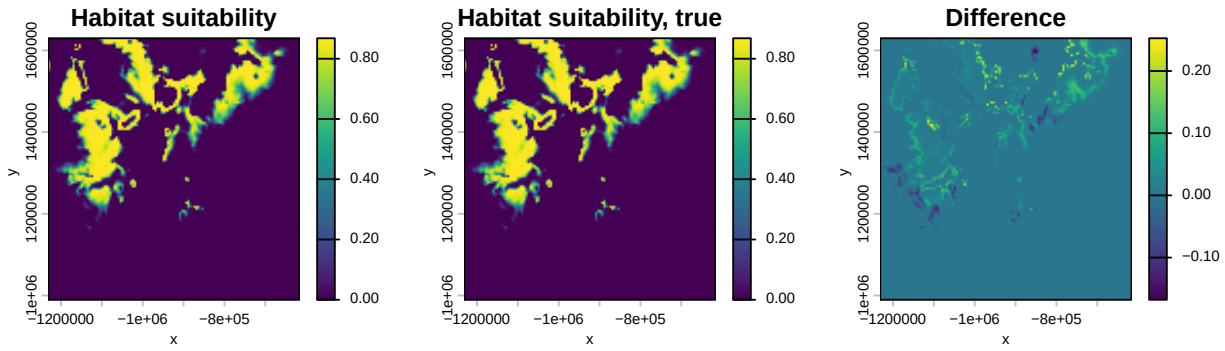


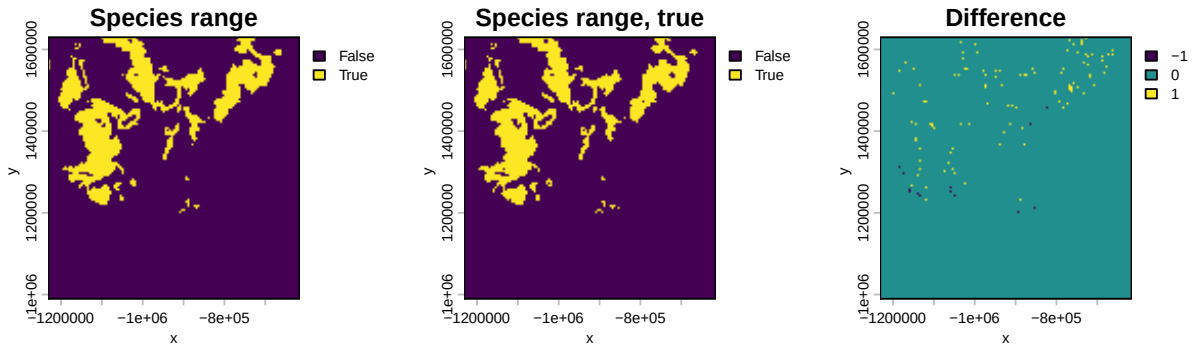
Note that transforming `o_par` profiles to the biological scale will not work the same way for more than two environmental variables because, in that case, there are multiple `o_par` parameters, and they interact with each other in the transformation to the biological scale. In that case, a Bayesian approach may have advantages.

7 Habitat suitability maps

We have now fitted several models, selected a best one, verified that the profiles look good, and shown that fitting captures the true model parameters. Habitat suitability maps are done with the `vsp` function. The `vsp` function has the dual purposes of allowing the user to experiment with creating virtual species, and producing habitat suitability maps. We here use it for the latter.

```
#plot the habitat suitability map
par(mfrow=c(2,3))
hab_suit = habitat_suitability(ML_parameters_bio,env_data)
terra::plot(hab_suit,main="Habitat suitability",xlab="x",ylab="y",legend=TRUE)

#plot the true habitat suitability map
hab_suit_true = habitat_suitability(example_1_true_parameters_bio,env_data)
terra::plot(hab_suit,main="Habitat suitability, true",
            xlab="x",ylab="y",legend=TRUE)

#plot the difference
terra::plot(hab_suit-hab_suit_true,main="Difference",
            xlab="x",ylab="y",legend=TRUE)

#plot the species ranges assuming a threshold of 0.5
terra::plot((hab_suit>.5),main="Species range",
            xlab="x",ylab="y",legend=TRUE)
terra::plot((hab_suit_true>.5),main="Species range, true",
            xlab="x",ylab="y",legend=TRUE)

#plot the difference
terra::plot((hab_suit>.5)-(hab_suit_true>.5),
            main="Difference",
            xlab="x",ylab="y",legend=TRUE)
```






```
#compute percent error in range
area = sum(as.data.frame((hab_suit_true>.5)))
area_setdiff = sum(abs(as.data.frame((hab_suit>.5)-(hab_suit_true>.5))-0)>1e-1)
delta = (area_setdiff)/area
area
```

```
## [1] 2058
```

```
area_setdiff
```

```
## [1] 115
```

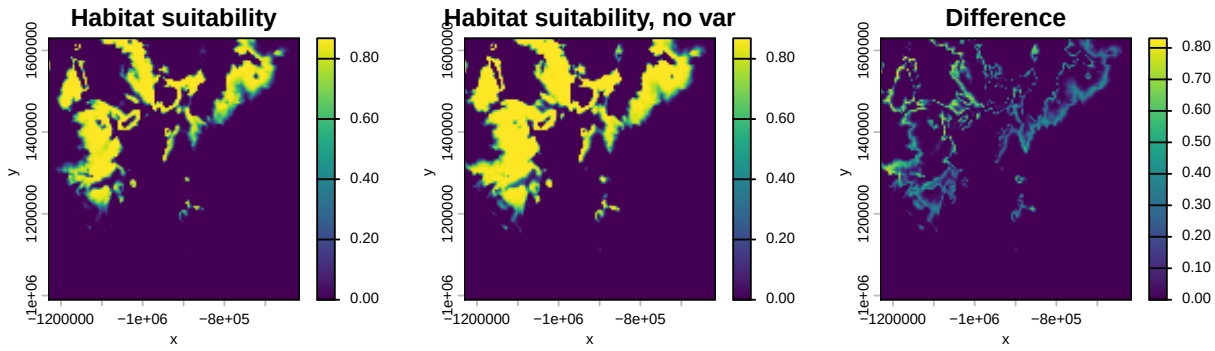
```
delta
```

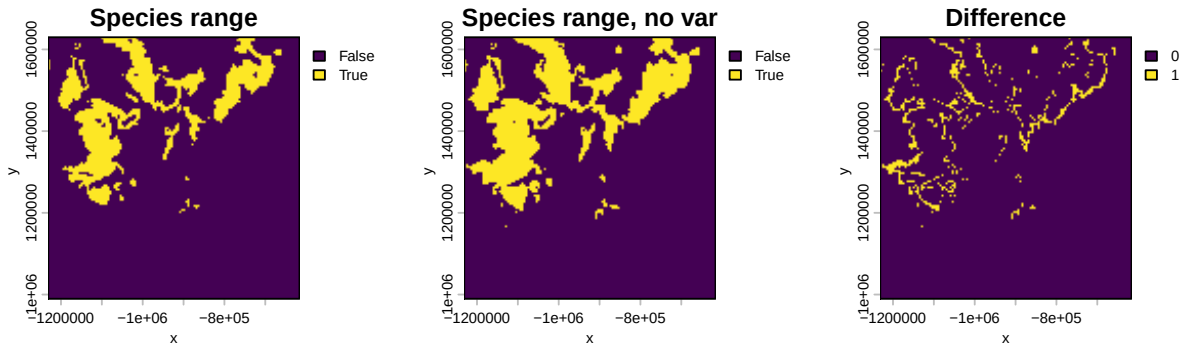
```
## [1] 0.05587949
```

As you can see, both the inferred habitat suitability map and the associated species range are similar to the true versions.

One of the main things `xsdm` is good for is assessing the influence of inter-annual climatic variability on species distributions, so we also display what the habitat suitability map would look like if the value of each environmental variable in each location were always equal to the temporal mean for that variable and location. We use inferred parameters.

```
#plot the habitat suitability map again, to ease comparisons
par(mfrow=c(2,3))
hab_suit = habitat_suitability(ML_parameters_bio, env_data)
terra::plot(hab_suit,main="Habitat suitability",xlab="x",ylab="y",legend=TRUE)

#plot the habitat suitability map without variability
m_bio_1_ts = rep(m_bio_1,terra::nlyr(bio_1))
m_bio_12_ts = rep(m_bio_12,terra::nlyr(bio_12))
m_env_data=list(m_bio_1=m_bio_1_ts,m_bio_12=m_bio_12_ts)
m_hab_suit = habitat_suitability(ML_parameters_bio, m_env_data)
terra::plot(m_hab_suit,main="Habitat suitability, no var",
            xlab="x",ylab="y",legend=TRUE)

#plot the difference
terra::plot(m_hab_suit-hab_suit,main="Difference",
            xlab="x",ylab="y",legend=TRUE)

#plot the species ranges assuming a threshold of 0.5
terra::plot((hab_suit>.5),main="Species range",
            xlab="x",ylab="y",legend=TRUE)
terra::plot((m_hab_suit>.5),main="Species range, no var",
            xlab="x",ylab="y",legend=TRUE)

#plot the difference
terra::plot((m_hab_suit>.5)-(hab_suit>.5),
            main="Difference",
            xlab="x",ylab="y",legend=TRUE)
```






```
#compute change in area due to variability
area = sum(as.data.frame((hab_suit>.5)))
area_novar = sum(as.data.frame((m_hab_suit>.5)))
delta_area = (area_novar-area)/area_novar
area
```

```
## [1] 2147
```

```
area_novar
```

```
## [1] 2881
```

```
delta_area
```

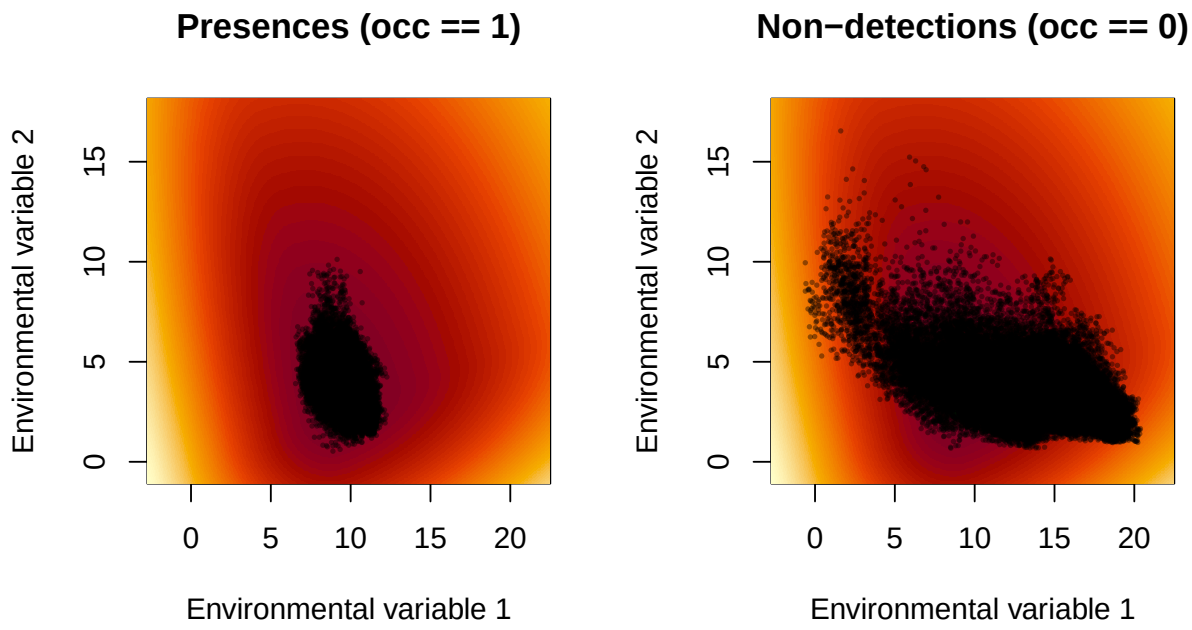
```
## [1] 0.2547726
```

So we see variability reduces area by 25.48% for this virtual species.

8 Interpreting model parameters

We would like to interpret the the parameters of the fitted model to say what we can about how the species responds to the environment. This is rendered a bit more challenging due to the parameter reduction step which was carried out for the model to eliminate structural non-identifiability (see "The xsdm model" for details). The function `interpret_parameters` helps with this, displaying plots which describe the inferred growth-environment function (the relationship between the environment, $\vec{e}_t$, in a given year in a location and the annual net growth rate, λ_t). For instance, for our example, the contours of that function are determined, though their heights are not determined. The contours are enough to tell the user what the optimal environment is for the species, and how sensitive annual net growth is to departures from this optimum for each environmental variable, relative to the other environmental variables:

```
devtools::load_all()

par(mfrow=c(1,2))
interpret_parameters(param_list = ML_parameters_bio,
                     plot_indices=c(1,2),
                     env_dat = env_array,
                     occ = example_1$occ_df$presence
                     )
```

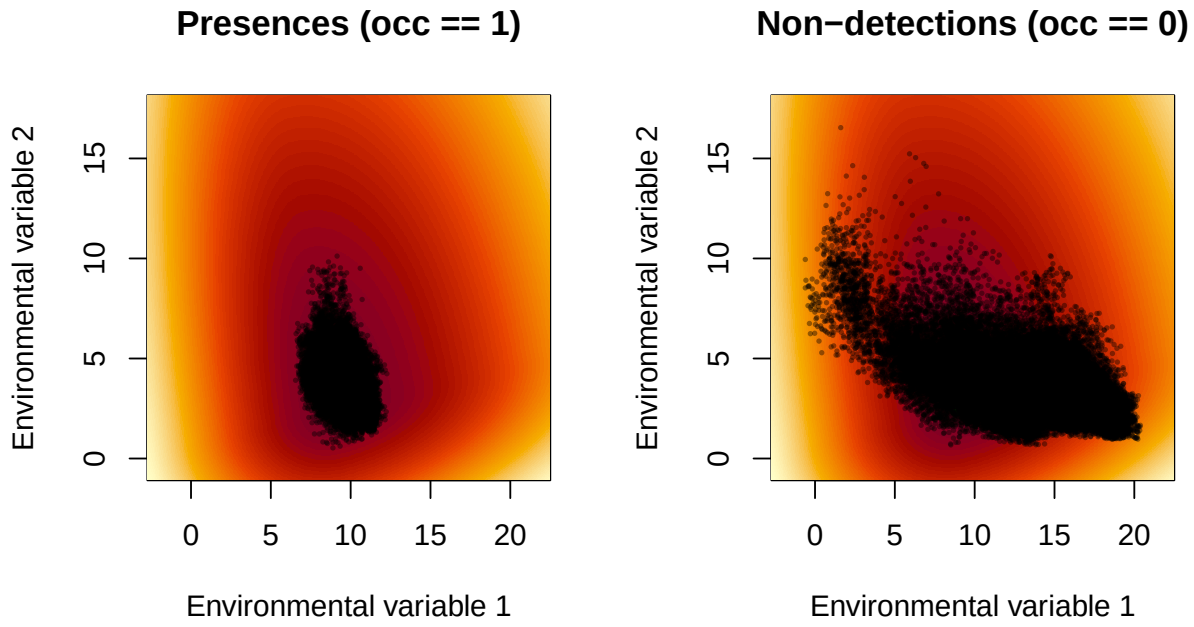


The points displayed here are all values of the environmental variables, in any year, in locations for which the species was detected (left) or assumed absent (right). One can see that growth is inferred to be more sensitive to low temperatures (leftward departures of environmental variable

1 from the optimum) than it is to high temperatures (rightward departures); and growth is more sensitive to drought (downward departures of environmental variable 2 from the optimum) than it is to abundant rainfall (upward departures). One can see that, as expected, the environment is more suitable for population growth, according to the inferred growth-environment function, in locations for where the species was observed. One can see that, even in locations found to be unsuitable for the species (right panel), there were plenty of individual years for which environmental conditions promoted growth in that year, demonstrating the importance of interannual variability.

Now we plot the same thing for the true parameters:

```
par(mfrow=c(1,2))
interpret_parameters(param_list = example_1$par_list,
                     plot_indices=c(1,2),
                     env_dat = env_array,
                     occ =  example_1$occ_df$presence
                     )
```



Results were quite similar. Plots against one environmental variable are also possible, holding the other one at optimal values. See the documentation of `interpret_parameters` for details.