Implementing positivity constraints with the nucdataBaynet package

Georg Schnabel

2023-01-13

The values of variables or functions may be restricted to non-negative numbers. For instance, excitation functions in nuclear physics, have non-negative values everywhere. In this tutorial, we are going to learn how this constraint can be implemented in a Bayesian network using either a variable transformation or a penalty method.

Loading the required packages and defining the energy range

We will make use of the functionality of the *data.table* package to build up a data table with the information about the nodes present in the network. The *Matrix* package will be used to create a sparse covariance matrix that contains the covariance matrix blocks associated with the nodes. We also need to load the *nucdataBaynet* package to establish the links between the nodes and perform Bayesian inference in the Bayesian network. The *ggplot2* package will be pertinent for plotting experimental data and the estimated function.

```
library(data.table)
library(Matrix)
library(nucdataBaynet)
library(ggplot2)
```

Generating the synthetic data

Let us assume that the true function $f(x)$ is given by a parabola:

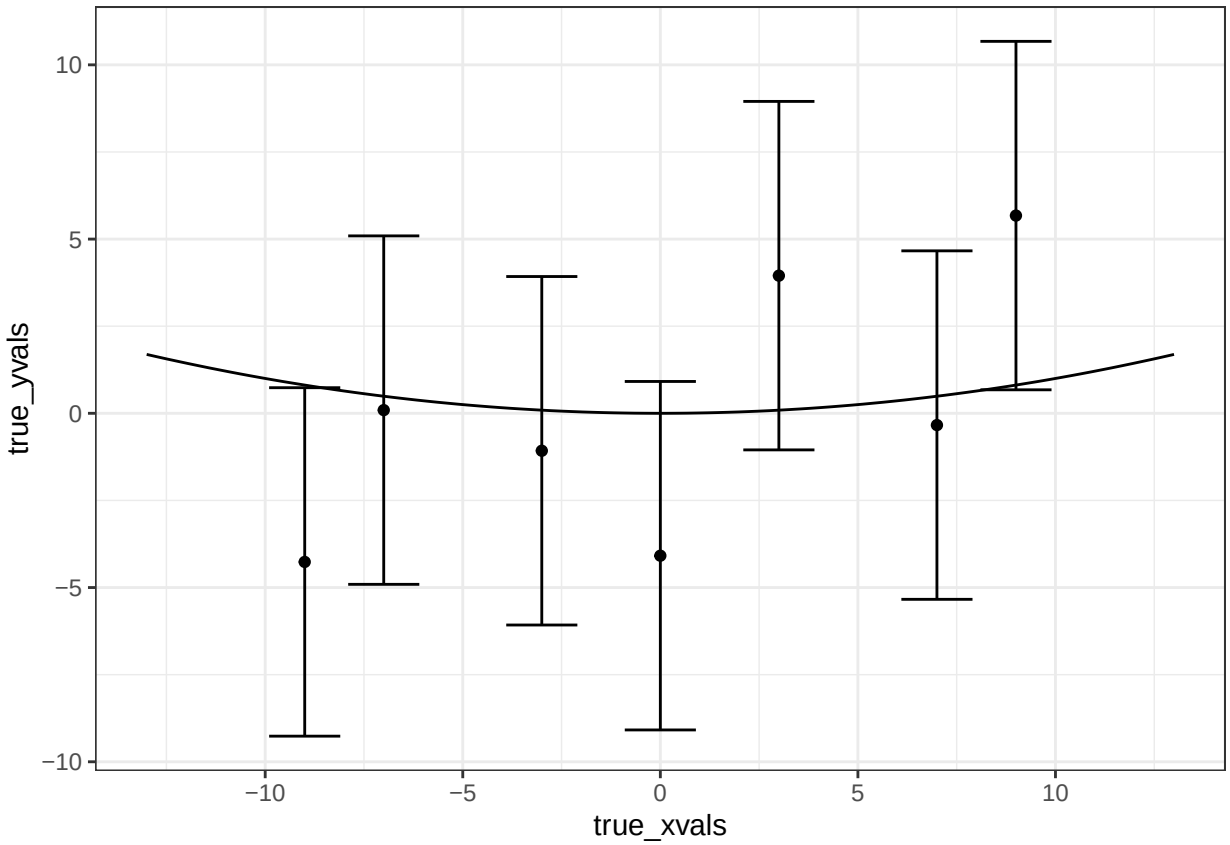
```
truefun <- function(x) { 0.01 * x^2 }
```

We further assume that we don't know this function but have made some measurements at locations x_1, x_2, etc. Due to imprecision in the measurement process, we did not observe $f(x_1)$, $f(x_2)$, ... but values $y_i = f(x_i) + \varepsilon_i$ with an unknown statistical error. In this notebook, we assume that we know that each error contribution ε_i was generated from a normal distribution with zero mean and standard deviation `5`. Following we create the synthetic data according to these assumptions:

```
unc_obs <- 5
x_obs <- c(-9, -7, -3, 0, 3, 7, 9)
y_obs <- truefun(x_obs) + rnorm(length(x_obs), 0, unc_obs)
```

Before we will be moving to the construction of the Bayesian network, let's plot the data in comparison to the true function.

```
true_xvals <- seq(-13, 13, length=100)
true_yvals <- truefun(true_xvals)
ggp <- ggplot() + theme_bw()
ggp <- ggp + geom_line(aes(x=true_xvals, y=true_yvals))
ggp <- ggp + geom_errorbar(aes(x=x_obs, ymin=y_obs-unc_obs, ymax=y_obs+unc_obs))
ggp <- ggp + geom_point(aes(x=x_obs, y=y_obs))
ggp
```



Constructing the basic Bayesian network skeleton

First, we will be constructing a Bayesian network without the non-negativity constraint. To this end, we define a node data table and then create the mappings to link the nodes. Afterwards, we are going to implement the constraint that solution curves must be non-negative in two different ways.

Defining the nodes

To define the node data table, we first define each node as its own data table and afterwards glue them together to a single data table.

Nodes associated with the true function We define a node that is associated with the true function to be estimated. We discretize the function on a computational mesh of x-values and assume that values in-between mesh points can be obtained by linear interpolation. This is a simple and flexible construction, which can be made arbitrarily precise by increasing the number of mesh points. The definition of the data table linked to the true function looks as follows:

```
truefun_dt <- data.table(
  NODE = "truefun",
  PRIOR = 0,
  UNC = 1e30,
  OBS = NA,
  X = seq(-13, +13, length=100)
)
```

The `NODE` field specifies the label for the node. With `PRIOR=0` we specify that the most likely value of the function without considering the data is zero. However, the huge prior uncertainty of `1e30` makes the prior

uninformative.

To regularize the solution, we define a data table for the node corresponding to the second derivative:

```
truefun_2nd_deriv_dt <- data.table(
  NODE = "truefun_2nd_deriv",
  PRIOR = 0,
  UNC = 1,
  OBS = 0,
  X = truefun_dt[NODE=='truefun', head(X[-1], n=-1)]
)
```

The second derivative at a mesh point x_i is computed by an expression involving finite differences to x_{i-1} and x_{i+1} so the first and last x-value of `truefun` are not included in the field `X` here. Doing so would raise an error later on in the setup of the mapping.

Node associated with the synthetic datapoints Next, we make use of the variables associated with the synthetic data to create a data table for the observation node:

```
obs_dt <- data.table(
  NODE = "obs",
  PRIOR = rep(0, length(x_obs)),
  UNC = unc_obs,
  OBS = y_obs ,
  X = x_obs
)
```

Merging the individual data tables Finally, we merge the three data tables defined above to a single one and augment it with an index column:

```
node_dt <- rbindlist(list(truefun_dt, truefun_2nd_deriv_dt, obs_dt))
node_dt[, IDX:=seq_len(.N)]
```

Creating the basic mapping

To obtain estimates of the function associated with the node `truefun`, we need to define a mapping between `truefun` and the observation node `obs`. We will be using an interpolation mapping to propagate the values from the `truefun` node to values that can be compared with the observed values of the `obs` node.

The following list contains all required information for the setup of the linear interpolation mapping. The precise meaning of the field names can be looked up by executing `?create_linearinterpol_map` in the R console to open the help page.

```
truefun_to_obs_mapdef <- list(
  mapname = "truefun_to_obs",
  maptype = "linearinterpol_map",
  src_idx = node_dt[NODE=="truefun", IDX],
  tar_idx = node_dt[NODE=="obs", IDX],
  src_x = node_dt[NODE=="truefun", X],
  tar_x = node_dt[NODE=="obs", X]
)
```

We also define the mapping from `truefun` to the node `truefun_2nd_deriv` to enforce the regularization (see `?create_derivative2nd_map` for an explanation of the parameters):

```
truefun_to_2nd_deriv_mapdef <- list(
  maptype = "derivative2nd_map",
  mapname = "truefun_to_2nd_deriv",
```

```
  src_idx = node_dt[NODE=="truefun", IDX],
  tar_idx = node_dt[NODE=="truefun_2nd_deriv", IDX],
  src_x = node_dt[NODE=="truefun", X],
  tar_x = node_dt[NODE=="truefun_2nd_deriv", X]
)
```

This new mapping is combined with the basic mapping to a compound mapping:

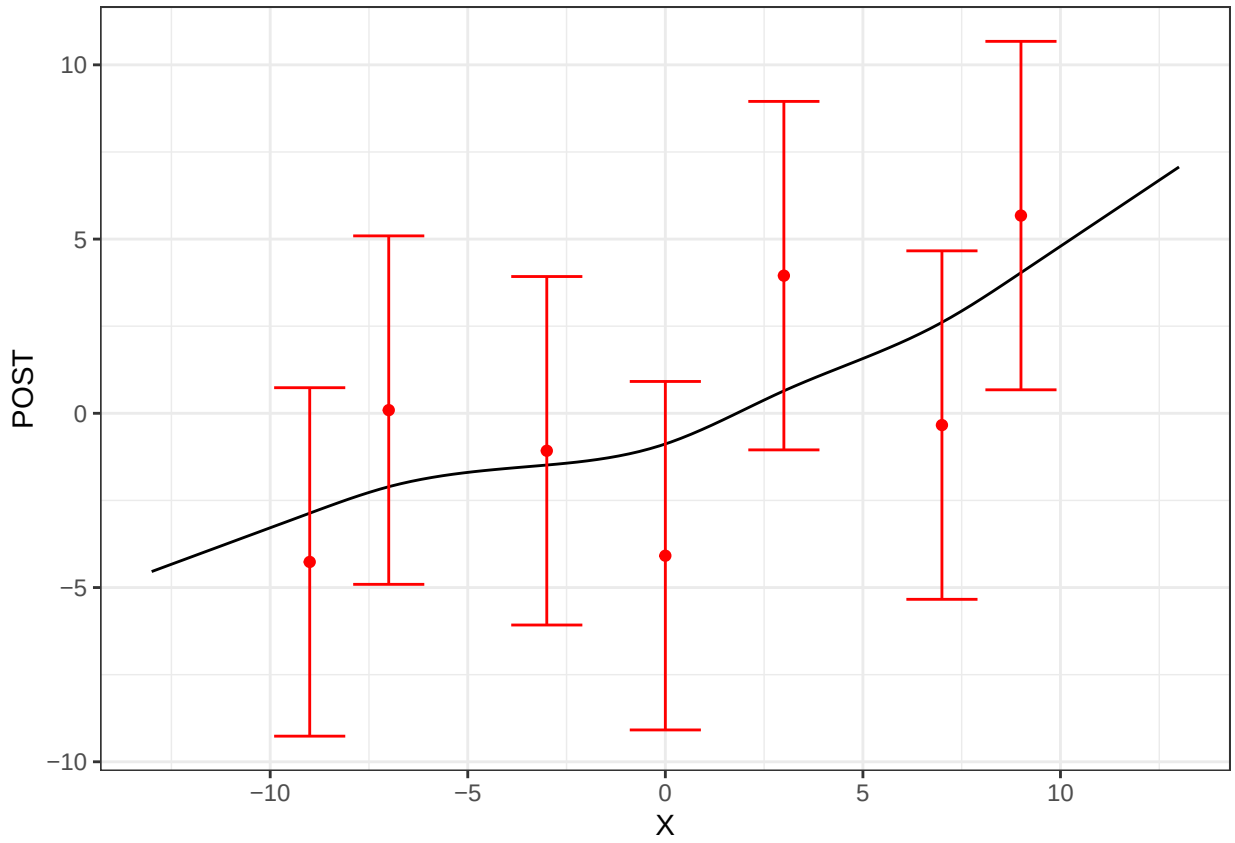
```
compmap_def <- list(
  maptype = "compound_map",
  mapname = "notimportant",
  maps = list(truefun_to_obs_mapdef, truefun_to_2nd_deriv_mapdef)
)
compmap <- create_map(compmap_def)
```

Now we can do the inference in the Bayesian network:

```
U_prior <- Diagonal(x=node_dt$UNC^2)
optres <- LMalgo(compmap, zprior=node_dt$PRIOR, U=U_prior, obs=node_dt$OBS)
node_dt[, POST:=compmap$propagate(optres$zpost)]
```

The following plot shows the comparison between the most likely posterior function (MAP estimate) and the data:

```
ggp <- ggplot() + theme_bw()
ggp <- ggp + geom_line(aes(x=X, y=POST), data=node_dt[NODE=='truefun',])
ggp <- ggp + geom_errorbar(aes(x=X, ymin=OBS-UNC, ymax=OBS+UNC), data=node_dt[NODE=='obs'], col='red')
ggp <- ggp + geom_point(aes(x=X, y=OBS), data=node_dt[NODE=='obs'], col='red')
ggp
```



Even though the data points were generated from a parabola with function values $y \geq 0$ everywhere, some of them lie below the x-axis due to the contribution of the statistical error.

If we know that the function values should be greater or equal zero everywhere, we can integrate this knowledge into our prior specification to obtain more reliable estimates complying with the constraint.

Enforcing a positivity constraint

We will explore two ways to integrate the constraint that functions values should be greater than or equal to zero in the Bayesian network inference.

First approach using the penalty method

The idea of the first approach consists of linking the `truefun` node associated with the true function to a new node we name `trunc_truefun`. The link to this new node will be given by a non-linear mapping that propagates any value in `truefun` as it is if it is negative and substitutes it by zero otherwise, $y_i = min(x_i, 0)$. We can then assume that the values in `trunc_truefun` have been observed to be zero with small uncertainty. The introduction of these pseudo-observations penalizes solutions with negative values in the Bayesian inference, thereby driving the negative values towards zero. This approach implements what is referred to as penalty method in the domain of constrained optimization.

So let us define the new node `trunc_truefun`:

```
trunc_truefun_dt <- data.table(
  NODE = "trunc_truefun",
  PRIOR = 0,
  UNC = 1e-1,
  OBS = 0,
```

```
  X = truefun_dt[, X]
)
```

For a fast convergence in optimization procedure to find the MAP estimate, the value of `UNC` should not be too small as it would make it harder for the optimization to converge but also not be too large, which could potentially lead to unacceptable violations of the positivity constraint. The value above was found after some optimization trial attempts. A good choice will also depend in general on the specifics of the mesh spacing (here about `0.3` units) and the uncertainty of the pseudo-observations of the second derivative to enforce a certain degree of smoothness. To understand this better, take a look at the formula for the second derivative mapping (`?create_derivative2nd_map`). As a next step, we are augmenting the node data table with the extra node `trunc_truefun`:

```
ext_node_dt1 <- copy(node_dt)
ext_node_dt1[, IDX := NULL]
ext_node_dt1[, POST := NULL]
ext_node_dt1 <- rbindlist(list(ext_node_dt1, trunc_truefun_dt))
ext_node_dt1[, IDX := seq_len(.N)]
```

The node `truefun` needs to be linked to `trunc_truefun`, which is achieved by the following mapping definition of a non-linear mapping (consult `?create_nonlinear_map` for the meaning of the parameters):

```
truefun_to_trunc_truefun_mapdef <- list(
  mapname = "truefun_to_trunc_truefun_map",
  maptype = "nonlinear_map",
  src_idx = ext_node_dt1[NODE=="truefun", IDX],
  tar_idx = ext_node_dt1[NODE=="trunc_truefun", IDX],
  funname = "limiter",
  minvalue = -Inf,
  maxvalue = 0
)
```

The `limiter` mapping, propagates the values as they are if they lie between `minvalue` and `maxvalue`, otherwise they are mapped to zero. As we want to penalize negative values, we map positive values to zero and propagate negative values as they are.

Now we can combine all individual mappings to a compound mapping:

```
ext_compmap1_mapdef <- list(
  mapname = "compmap",
  maptype = "compound_map",
  maps = list(truefun_to_obs_mapdef, truefun_to_2nd_deriv_mapdef, truefun_to_trunc_truefun_mapdef)
)
ext_compmap1 <- create_map(ext_compmap1_mapdef)
```

And do the Bayesian inference:

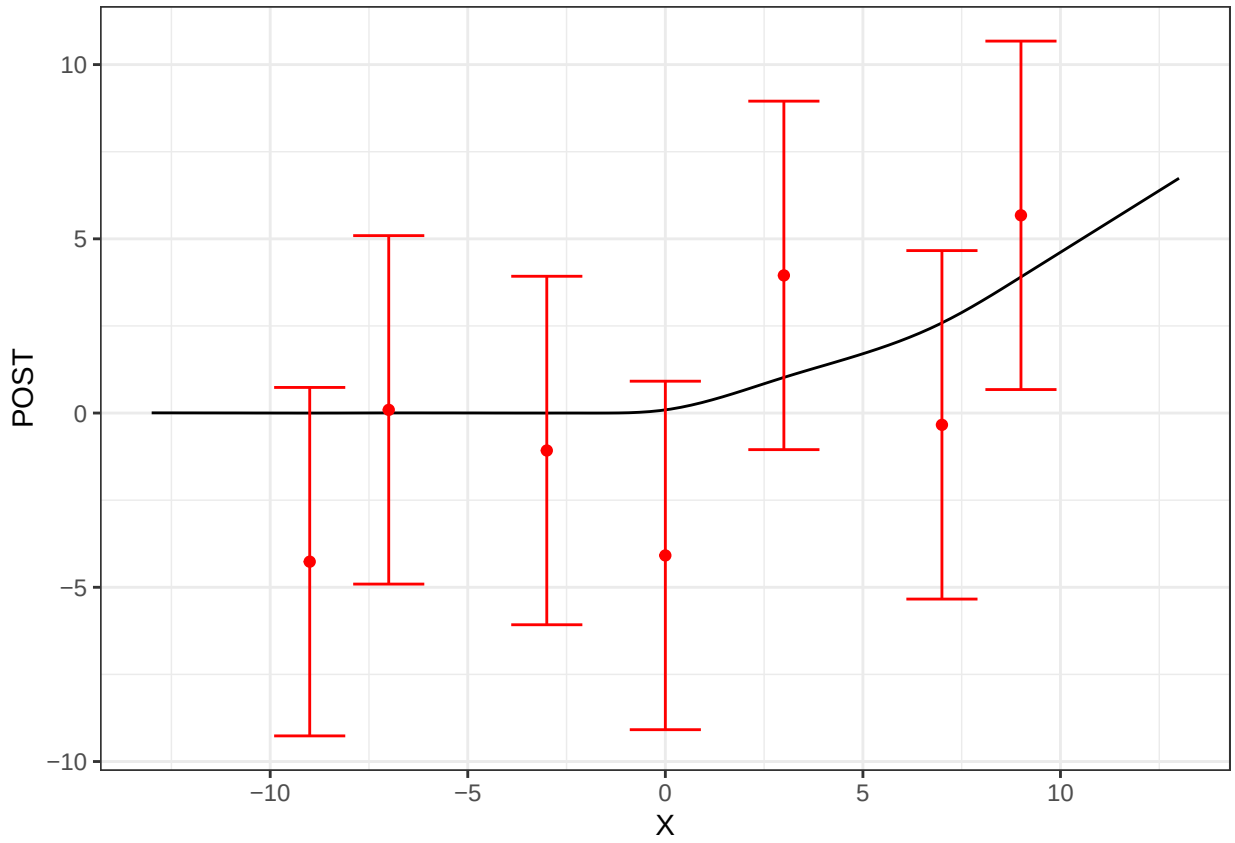
```
U_prior <- Diagonal(x=ext_node_dt1$UNC^2)
optres <- LMalgo(ext_compmap1, zprior=ext_node_dt1$PRIOR, U=U_prior, obs=ext_node_dt1$OBS, control=list
ext_node_dt1[, POST:=ext_compmap1$propagate(optres$zpost)]
```

The optimization becomes slightly more delicate with a penalization node and we need to be a bit more careful with the optimization parameters. There is a control parameter called `tau` that specifies how much the gradient should be trusted initially to find the next proposal candidate in the iterative optimization scheme. The default of `tau=1e-10` leads to a too large step size and many proposals need to be discarded in a row because they don't lead to an improvement. Therefore we specified `tau=1` to start out with smaller steps at the beginning. Alternatively, we could have increased `maxreject` to tolerate more subsequent proposal rejections and give the algorithm more time to find a suitable value for `tau`. See `?LMalgo` for all control

parameters of the optimization algorithm.

Finally, we want to plot the result (the MAP estimate) in comparison with the synthetic data:

```
ggp <- ggplot() + theme_bw()
ggp <- ggp + geom_line(aes(x=X, y=POST), data=ext_node_dt1[NODE=='truefun',])
ggp <- ggp + geom_errorbar(aes(x=X, ymin=OBS-UNC, ymax=OBS+UNC), data=ext_node_dt1[NODE=='obs'], col='r
ggp <- ggp + geom_point(aes(x=X, y=OBS), data=ext_node_dt1[NODE=='obs'], col='red')
ggp
```



Values that used to be negative without incorporating the constraint in the Bayesian inference are now close to zero. Let's check how close to zero by inspecting the `POST` column populated with the result of the Bayesian inference:

```
ext_node_dt1[NODE=="truefun", list(NODE, X, POST)]
```

```
##         NODE           X          POST
##   1: truefun -13.0000000  5.523576e-03
##   2: truefun -12.7373737  5.104644e-03
##   3: truefun -12.4747475  4.685712e-03
##   4: truefun -12.2121212  4.266780e-03
##   5: truefun -11.9494949  3.847849e-03
##   6: truefun -11.6868687  3.428917e-03
##   7: truefun -11.4242424  3.009985e-03
##   8: truefun -11.1616162  2.591053e-03
##   9: truefun -10.8989899  2.172121e-03
##  10: truefun -10.6363636  1.753190e-03
##  11: truefun -10.3737374  1.334258e-03
```

```
##  12: truefun -10.1111111  9.153261e-04
##  13: truefun  -9.8484848  4.963943e-04
##  14: truefun  -9.5858586  7.746249e-05
##  15: truefun  -9.3232323 -3.414693e-04
##  16: truefun  -9.0606061 -7.604011e-04
##  17: truefun  -8.7979798 -5.295560e-04
##  18: truefun  -8.5353535 -4.898352e-05
##  19: truefun  -8.2727273  5.399163e-04
##  20: truefun  -8.0101010  1.188953e-03
##  21: truefun  -7.7474747  1.849938e-03
##  22: truefun  -7.4848485  2.474680e-03
##  23: truefun  -7.2222222  3.014990e-03
##  24: truefun  -6.9595960  3.422676e-03
##  25: truefun  -6.6969697  3.659909e-03
##  26: truefun  -6.4343434  3.745827e-03
##  27: truefun  -6.1717172  3.699568e-03
##  28: truefun  -5.9090909  3.540271e-03
##  29: truefun  -5.6464646  3.287077e-03
##  30: truefun  -5.3838384  2.959122e-03
##  31: truefun  -5.1212121  2.575547e-03
##  32: truefun  -4.8585859  2.155491e-03
##  33: truefun  -4.5959596  1.718092e-03
##  34: truefun  -4.3333333  1.282489e-03
##  35: truefun  -4.0707071  8.678215e-04
##  36: truefun  -3.8080808  4.932280e-04
##  37: truefun  -3.5454545  1.778477e-04
##  38: truefun  -3.2828283 -5.918065e-05
##  39: truefun  -3.0202020 -1.987180e-04
##  40: truefun  -2.7575758 -1.090114e-04
##  41: truefun  -2.4949495 -3.528919e-05
##  42: truefun  -2.2323232 -7.826999e-05
##  43: truefun  -1.9696970 -2.715212e-04
##  44: truefun  -1.7070707 -4.996713e-04
##  45: truefun  -1.4444444 -1.306753e-04
##  46: truefun  -1.1818182  2.418329e-03
##  47: truefun  -0.9191919  8.978864e-03
##  48: truefun  -0.6565657  2.138245e-02
##  49: truefun  -0.3939394  4.146061e-02
##  50: truefun  -0.1313131  7.104488e-02
##  51: truefun   0.1313131  1.119668e-01
##  52: truefun   0.3939394  1.644678e-01
##  53: truefun   0.6565657  2.271995e-01
##  54: truefun   0.9191919  2.988134e-01
##  55: truefun   1.1818182  3.779612e-01
##  56: truefun   1.4444444  4.632942e-01
##  57: truefun   1.7070707  5.534640e-01
##  58: truefun   1.9696970  6.471223e-01
##  59: truefun   2.2323232  7.429204e-01
##  60: truefun   2.4949495  8.395100e-01
##  61: truefun   2.7575758  9.355426e-01
##  62: truefun   3.0202020  1.029670e+00
##  63: truefun   3.2828283  1.120714e+00
##  64: truefun   3.5454545  1.209557e+00
##  65: truefun   3.8080808  1.297077e+00
```

```
##  66: truefun   4.0707071  1.384155e+00
##  67: truefun   4.3333333  1.471672e+00
##  68: truefun   4.5959596  1.560508e+00
##  69: truefun   4.8585859  1.651542e+00
##  70: truefun   5.1212121  1.745655e+00
##  71: truefun   5.3838384  1.843728e+00
##  72: truefun   5.6464646  1.946639e+00
##  73: truefun   5.9090909  2.055271e+00
##  74: truefun   6.1717172  2.170503e+00
##  75: truefun   6.4343434  2.293214e+00
##  76: truefun   6.6969697  2.424286e+00
##  77: truefun   6.9595960  2.564598e+00
##  78: truefun   7.2222222  2.715032e+00
##  79: truefun   7.4848485  2.874581e+00
##  80: truefun   7.7474747  3.041901e+00
##  81: truefun   8.0101010  3.215644e+00
##  82: truefun   8.2727273  3.394462e+00
##  83: truefun   8.5353535  3.577011e+00
##  84: truefun   8.7979798  3.761942e+00
##  85: truefun   9.0606061  3.947909e+00
##  86: truefun   9.3232323  4.133876e+00
##  87: truefun   9.5858586  4.319844e+00
##  88: truefun   9.8484848  4.505811e+00
##  89: truefun  10.1111111  4.691778e+00
##  90: truefun  10.3737374  4.877745e+00
##  91: truefun  10.6363636  5.063712e+00
##  92: truefun  10.8989899  5.249679e+00
##  93: truefun  11.1616162  5.435647e+00
##  94: truefun  11.4242424  5.621614e+00
##  95: truefun  11.6868687  5.807581e+00
##  96: truefun  11.9494949  5.993548e+00
##  97: truefun  12.2121212  6.179515e+00
##  98: truefun  12.4747475  6.365483e+00
##  99: truefun  12.7373737  6.551450e+00
## 100: truefun  13.0000000  6.737417e+00
##         NODE           X          POST
```

As can be seen, the constraint is satisfied with an absolute tolerance of about 10^{-3}. We could do postprocessing now to cut away the slightly negative values. Alternatively, we could decrease the uncertainty in `trunc_truefun` to make the penalty term more dominant and perform the optimization again. Then we should, however, also be careful with control parameters of the `LMalgo` method to ensure that the optimization method manages to converge sufficiently.

Second approach based on auxiliary variables

Another possibility to implement the constraint is to make use of a variable transformation. We can introduce an additional node `aux_truefun` where values are unconstrained and link them via a non-linear mapping to the node `truefun` associated with the true function. The linear mapping will be given by $y_i = max(0, x_i)$ with y_i being the propagated value and x_i being the value in the source node.

The definition of the auxiliary node `aux_truefun` looks as follows:

```
aux_truefun_dt <- data.table(
  NODE = "aux_truefun",
  PRIOR = 1,
  UNC = 1e30,
```

```
  OBS = NA,
  X = truefun_dt[, X]
)
```

The prior is very uninformative due to the large value `UNC=1e30` so the influence of the `PRIOR` values is negligible in the region where data exists. However, due to the non-linear mapping, the gradient of posterior density function with respect to the values in the auxiliary node vanishes for negative values. In order to have a small pull towards positive values, we therefore set `PRIOR=1`. Another possibility would be to provide as starting values in `zref`, which are all positive.

Next, we add the auxiliary node to the node data table:

```
ext_node_dt2 <- copy(node_dt)
ext_node_dt2[, IDX := NULL]
ext_node_dt2[, POST := NULL]
ext_node_dt2 <- rbindlist(list(ext_node_dt2, aux_truefun_dt))
ext_node_dt2[, IDX := seq_len(.N)]
```

Importantly, we want the values in `truefun` to be a deterministic function of the values in `aux_truefun` so we are setting `UNC=0`. The values in the column `PRIOR` of the node `truefun` are already zero, and we keep that as we don't want to add anything to the propagated values from `auxfun`. The column `OBS` is already `NA` and we keep that because we assume `truefun` to be unobserved.

```
ext_node_dt2[NODE=="truefun", UNC := 0]
```

The non-linear mapping is defined as follows:

```
aux_to_truefun_mapdef <- list(
  mapname = "aux_to_truefun_map",
  maptype = "nonlinear_map",
  src_idx = ext_node_dt2[NODE=="aux_truefun", IDX],
  tar_idx = ext_node_dt2[NODE=="truefun", IDX],
  funname = "limiter",
  minvalue = 0,
  maxvalue = Inf
)
```

We go again for the `limiter` option but now with limits from zero to infinity. As the node `truefun` is the target of the mapping, the constraint of values being non-negative is enforced. You may also want to explore the option `funname="exp"`, which would establish $y_i = exp(x_i)$ as mapping.

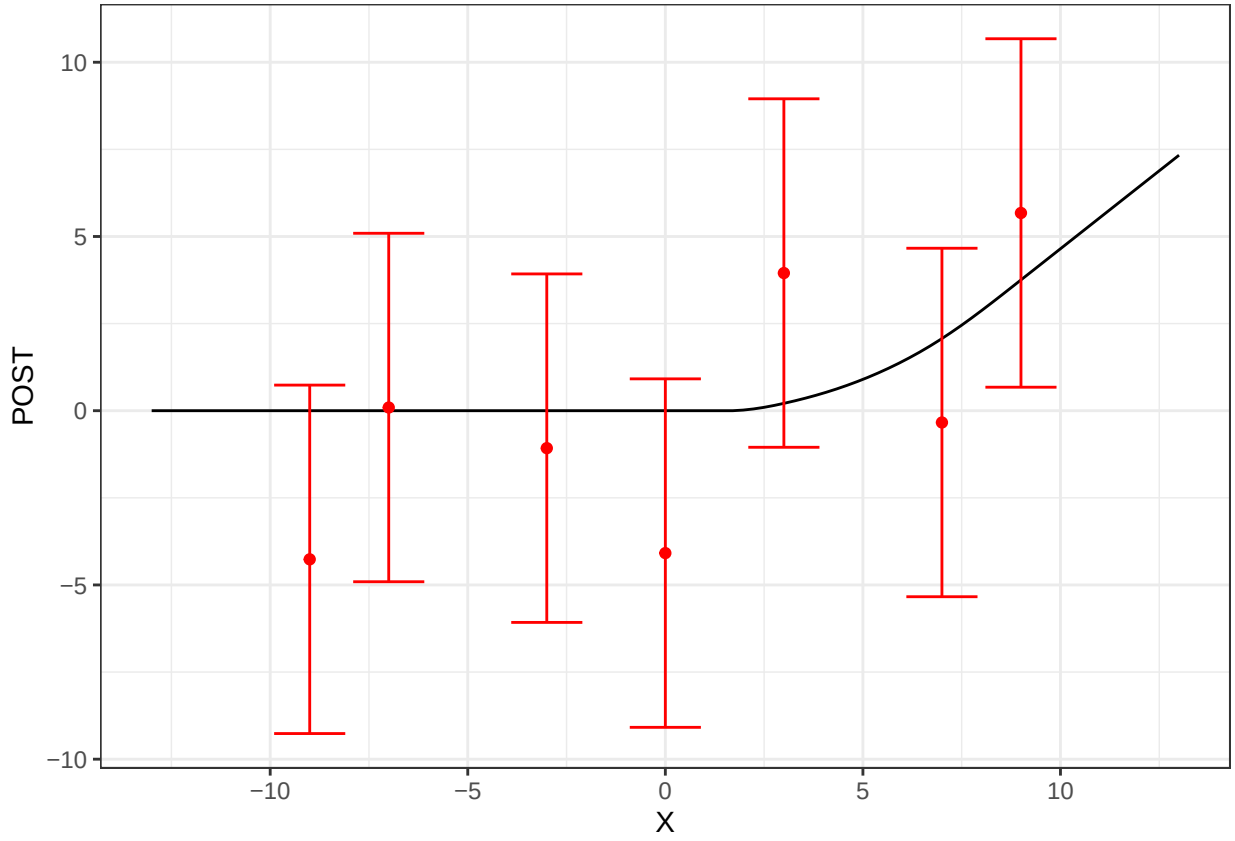
Finally, we bundle all the relevant mappings together to a compound map:

```
ext_compmap2_mapdef <- list(
  mapname = "compmap",
  maptype = "compound_map",
  maps = list(aux_to_truefun_mapdef, truefun_to_obs_mapdef, truefun_to_2nd_deriv_mapdef)
)
ext_compmap2 <- create_map(ext_compmap2_mapdef)
```

Performing the inference and plotting yields:

```
U_prior <- Diagonal(x=ext_node_dt2$UNC^2)
optres <- LMalgo(ext_compmap2, zprior=ext_node_dt2$PRIOR, U=U_prior, obs=ext_node_dt2$OBS)
ext_node_dt2[, POST:=ext_compmap2$propagate(optres$zpost)]
ggp <- ggplot() + theme_bw()
ggp <- ggp + geom_line(aes(x=X, y=POST), data=ext_node_dt2[NODE=='truefun',])
ggp <- ggp + geom_errorbar(aes(x=X, ymin=OBS-UNC, ymax=OBS+UNC), data=ext_node_dt2[NODE=='obs'], col='r
```

```
ggp <- ggp + geom_point(aes(x=X, y=OBS), data=ext_node_dt2[NODE=='obs'], col='red')
ggp
```



Summary

In this tutorial, we explored how the constraint that a functions must have values $f(x) \geq 0$ can be implemented in a Bayesian network. Two approaches were explored, which both relied on introducing an extra node.

In the first approach, the extra node was a mapping from the node associated with the true function to a truncated version with values larger than zero being set to zero. Pseudo-observations with their values being zero and associated with a small uncertainty were then assigned to this extra node, which effectively implemented the penalty method used to solve constrained optimization problems.

In the second approach, an auxiliary node was used with an uninformative prior, whose values were then mapped deterministically to the node representing the true function by a non-linear mapping, which propagates positive values as they are and substitutes negative values by zero.