Implementing monotonicity and convexity constraints with the nucdataBaynet package

Georg Schnabel

2023-01-18

Some functions are known to be monotonous or convex. In this tutorial, we are going to learn how these constraints can be implemented in a Bayesian network. The underlying mathematical method to take into account these constraints is known by the name penalty method in the field of constrained optimization.

Loading the required packages

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. Finally, we are going to use the *igraph* package to visualize Bayesian networks.

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

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 }
```

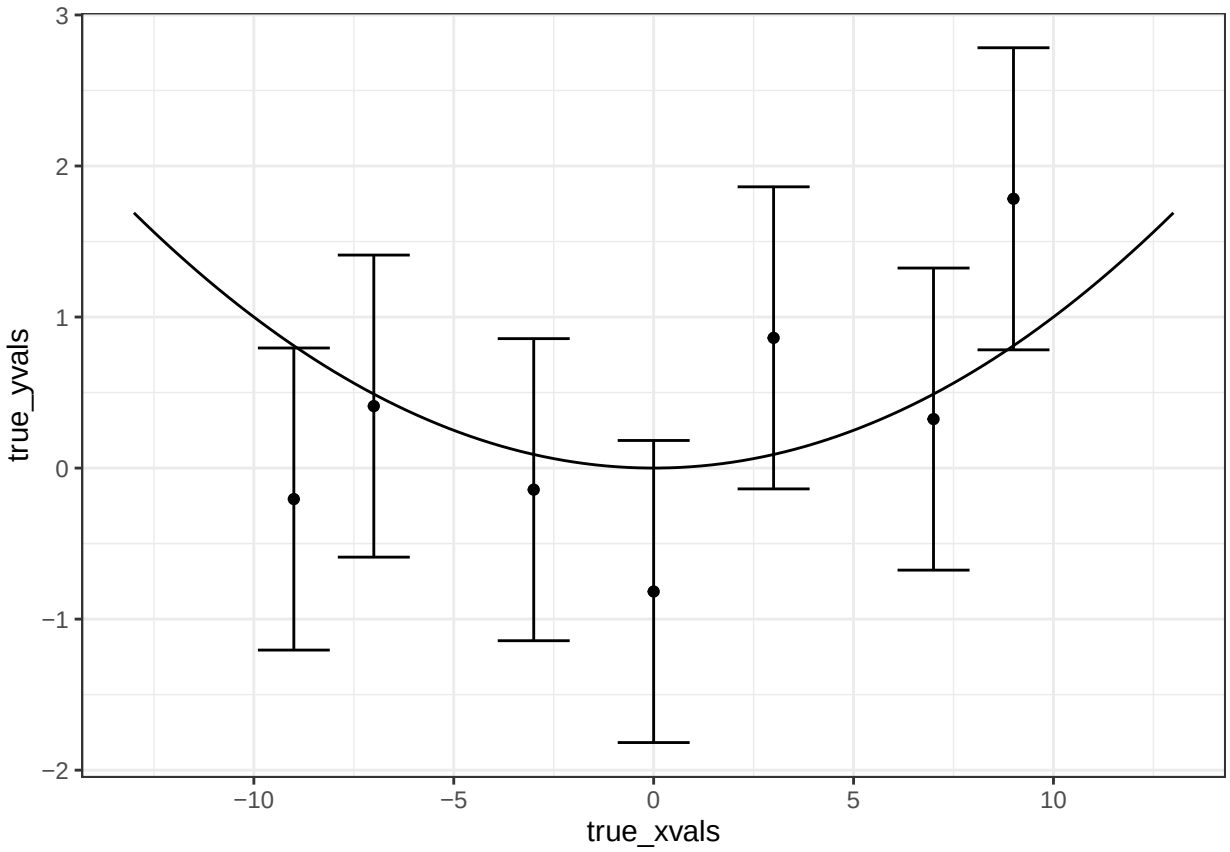
We further assume that we don't know this function but have made some measurements at locations x_1, x_2, etc. Due to imprecision in the measurement process, we did not observe $f(x_1)$, $f(x_2)$, ... but values $y_i = f(x_i) + \varepsilon_i$ with an unknown statistical error. In this notebook, we assume that we know that each error contribution ε_i was generated from a normal distribution with zero mean and standard deviation `1`. The synthetic data according to these assumptions can be created like this:

```
unc_obs <- 1
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
```



It can be seen that the observations are very noisy and consequently it will be very difficult to reconstruct something resembling a parabola without already knowing its shape. However, imposing additional constraints, such as the function being a convex function, we can improve estimates obtained by Bayesian inference, as we are going to explore in this tutorial.

Constructing the basic Bayesian network skeleton

First, we will be constructing a Bayesian network suitable for making inference about the true function underlying the data. We will only introduce the assumption that the function should be sufficiently smooth by imposing pseudo-observations on the second derivative.

To set up the Bayesian network, we define a data table with the information about the nodes and then create the mappings to link the nodes.

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 = Inf,
  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. We associate a prior uncertainty of `Inf` (for infinity) to the function values on the x-mesh to implement an uninformative prior.

For the purpose of comparison, we are also going to create a data table with the true function:

```
truth_dt <- data.table(
  X = truefun_dt[, X],
  Y = truefun_dt[, truefun(X)]
)
```

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. Execute `?create_derivate2nd_map` in the R console to get details on how the second derivative is computed.

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:

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

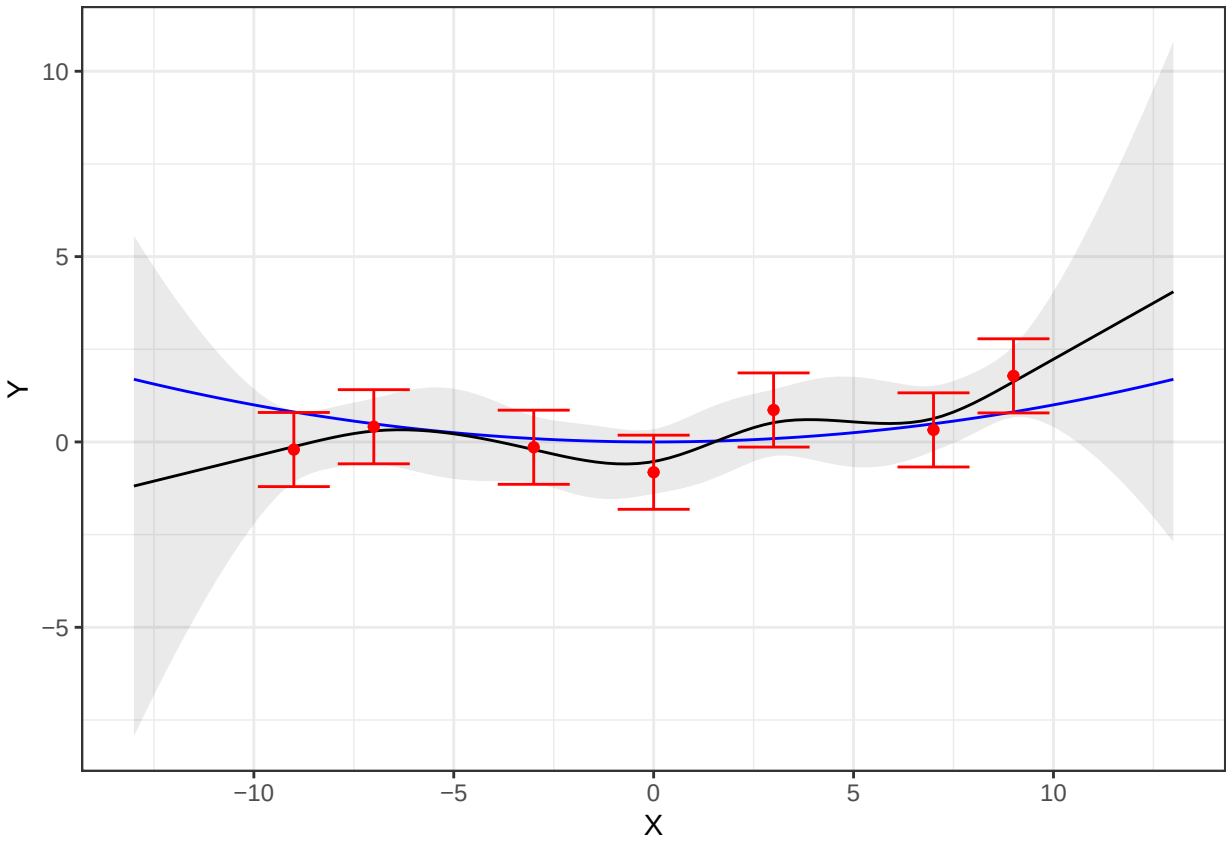
Let us also calculate the posterior uncertainties to be able to plot them:

```
sel_idcs <- node_dt[NODE=="truefun", IDX]
U_post <- get_posterior_cov(compmap, optres$zpost, U_prior, node_dt$OBS,
                            sel_idcs, sel_idcs)
post_unc <- sqrt(diag(U_post))
plot_node_dt[sel_idcs, UNC_POST := post_unc]
```

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

```
ggp <- ggplot() + theme_bw()
ggp <- ggp + geom_ribbon(aes(x=X, ymin=POST-UNC_POST, ymax=POST+UNC_POST),
                         alpha=0.1, data=plot_node_dt[NODE=='truefun'])
ggp <- ggp + geom_line(aes(x=X, y=Y), col='blue', data=truth_dt)
ggp <- ggp + geom_line(aes(x=X, y=POST), data=plot_node_dt[NODE=='truefun',])
ggp <- ggp + geom_errorbar(aes(x=X, ymin=OBS-UNC, ymax=OBS+UNC),
```

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



Even though the data points were generated from a parabola, the most likely function according to the posterior looks very different from the true parabola indicated by the blue line. The 1-sigma posterior uncertainty band around the estimated curve is nevertheless very consistent with the truth and opens up significantly in the extrapolation area left and right of the range covered by the data points. The width of the uncertainty band is a result of the smoothness constraint, i.e., the pseudo-observations of the second derivative. A curious reader may go up to the definition of `truefun_2nd_deriv_dt` and increase or decrease the value of `UNC` to see the impact on the estimated curve and the associated uncertainties.

Integrating additional constraints into the Bayesian network

With the basic Bayesian network established we are going to study two possible extensions to incorporate constraints on the shape of the function.

First, we will be introducing the constraint that the estimated function is monotonically increasing. Of course, this constraint is inconsistent with the underlying true parabola and Bayesian inference is expected to produce a worse solution.

Afterwards, we will instead integrate the constraint in the Bayesian network that the function to be inferred is convex, which is the case for a parabola, so we anticipate to obtain a better solution compared to the basic Bayesian network.

Case 1: Enforcing monotonicity

For a monotonically increasing function, it holds that $f(x_i) \leq f(x_j)$ if $x_i < x_j$. To integrate the constraint that the function is monotonically increasing, we are going to construct a penalty term associated with the negative values of the first derivative.

To be more precise, we will be adding a deterministic node reflecting the first derivative of `truefun`. Then we will be constructing another node that contains the first derivative if it is negative and otherwise zero. By imposing pseudo-observations with low uncertainties on this node associated with the truncated first derivative, we are implementing the penalty method for constrained optimization.

First, we set up the node associated with the first derivative:

```
truefun_deriv_dt <- data.table(
  NODE = "truefun_deriv",
  PRIOR = 0,
  UNC = 0,
  OBS = NA,
  X = tail(head(truefun_dt[, X], -1), -1)
)
```

By assigning an uncertainty `UNC` of zero, we specify this node to be a deterministic function of the nodes that feed into this one, which is here only the node `truefun`.

Next, we define the node `trunc_truefun_deriv`, which is going to contain the truncated first derivatives, i.e., $y = \min(f'(x), 0)$.

```
trunc_truefun_deriv_dt <- data.table(
  NODE = "trunc_truefun_deriv",
  PRIOR = 0,
  UNC = 1e-1,
  OBS = 0,
  X = truefun_deriv_dt[, X]
)
```

By specifying `OBS` equal zero in combination with a small value of `UNC`, we strongly penalize solution curves with a negative first derivative. For a fast convergence of the optimization procedure to obtain a MAP estimate, the value of `UNC` should not be too small as it would make it more difficult to achieve convergence. However, it should also not be too large, which could lead to unacceptable violations of the monotonicity 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 to enforce monotonicity. To get a feeling why the employed value for `UNC` may be a reasonable choice, it may help to take a look at the formula for the first derivative mapping (`?create_derivative_map`).

Now we can add the two defined nodes `truefun_deriv` and `trunc_truefun_deriv` to the Bayesian network:

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

With all nodes being added to the network, we need to link the new nodes appropriately. To link the node `truefun_deriv` to the node `truefun` associated with the true function, we employ a derivative mapping (execute `?create_derivative_map` for details on parameters):

```
truefun_to_deriv_mapdef <- list(
  mapname = "truefun_to_deriv_map",
  maptype = "derivative_map",
  src_idx = ext_node_dt1[NODE=="truefun", IDX],
```

```
  tar_idx = ext_node_dt1[NODE=="truefun_deriv", IDX],
  src_x = ext_node_dt1[NODE=="truefun", X],
  tar_x = ext_node_dt1[NODE=="truefun_deriv", X]
)
```

The node `truefun_deriv` needs to be linked to `trunc_truefun_deriv`, which is achieved by a non-linear mapping that performs the truncation (consult `?create_nonlinear_map` for the meaning of the parameters):

```
truefun_deriv_to_trunc_deriv_mapdef <- list(
  mapname = "truefun_deriv_to_trunc_deriv_map",
  maptype = "nonlinear_map",
  src_idx = ext_node_dt1[NODE=="truefun_deriv", IDX],
  tar_idx = ext_node_dt1[NODE=="trunc_truefun_deriv", 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 of the first derivative, 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_deriv_mapdef, truefun_deriv_to_trunc_deriv_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$OBS, control=list(mincount=10, maxcount=100))
ext_node_dt1[, POST:=ext_compmap1$propagate(optres$zpost)]
```

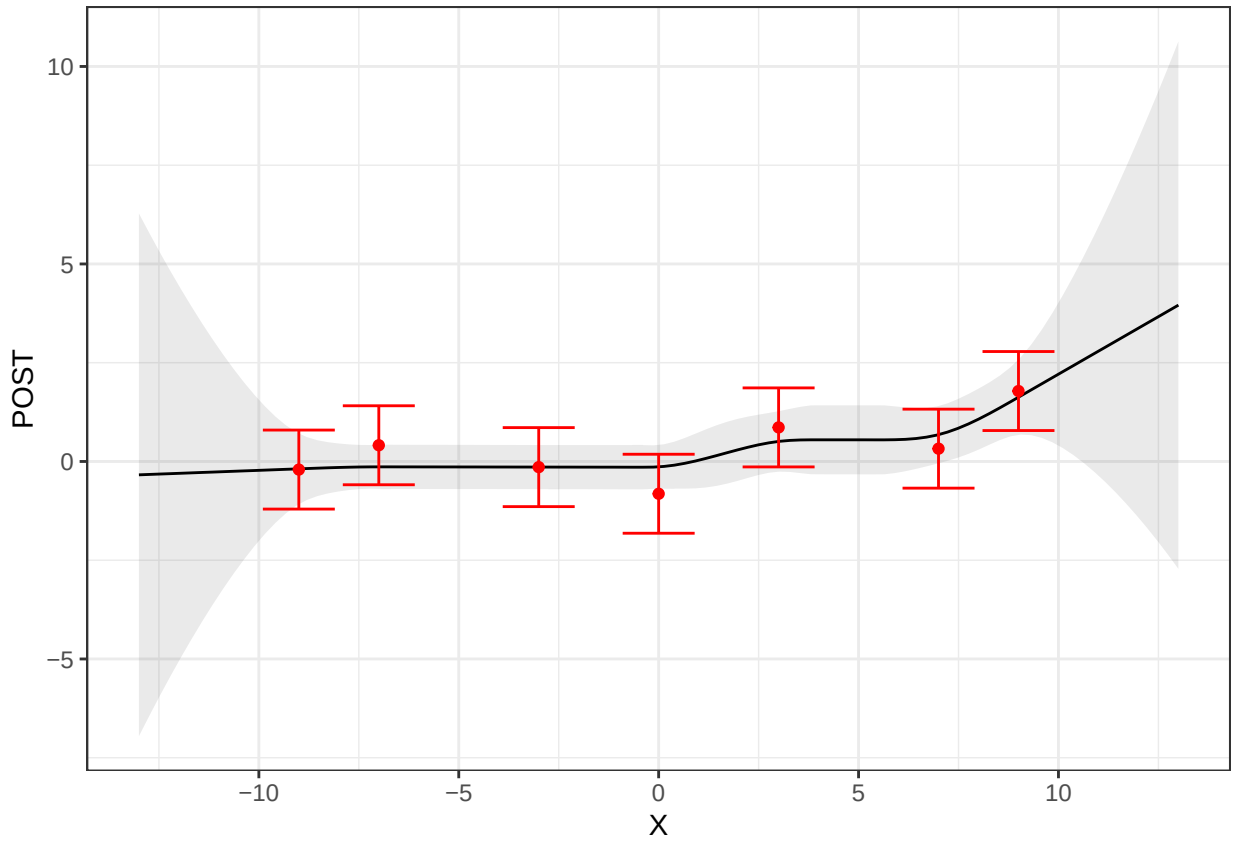
The optimization becomes slightly more delicate with a penalization node and we need to be a bit more careful with the optimization parameters. We can print `optres` to see the number of iterations. If the number of iterations is given by the maximum number of iterations (100 by default), we may want to increase this limit. Also a number of iterations being equal to the minimal number can be suspicious. If the solution looks strange, it may be worthwhile to play with these control parameters. See `?LMalgo` for all control parameters of the optimization algorithm.

```
sel_idcs <- ext_node_dt1[NODE=="truefun", IDX]
U_post1 <- get_posterior_cov(ext_compmap1, optres$zpost, U_prior,
                             ext_node_dt1$OBS, sel_idcs, sel_idcs)
post_unc <- sqrt(diag(U_post1))
ext_node_dt1[sel_idcs, UNC_POST := post_unc]
```

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

```
ggp <- ggplot() + theme_bw()
ggp <- ggp + geom_ribbon(aes(x=X, ymin=POST-UNC_POST, ymax=POST+UNC_POST),
                         alpha=0.1, data=ext_node_dt1[NODE=='truefun'])
ggp <- ggp + geom_line(aes(x=X, y=POST), data=ext_node_dt1[NODE=='truefun',])
```

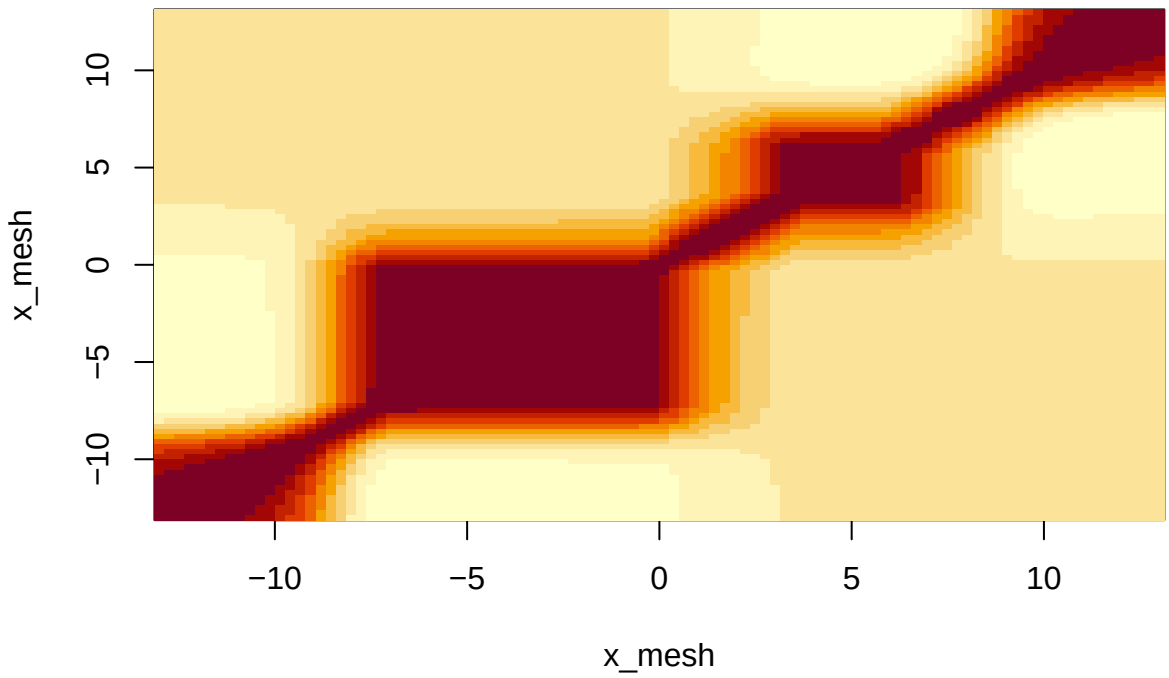
```
ggp <- ggp + geom_errorbar(aes(x=X, ymin=OBS-UNC, ymax=OBS+UNC),
                           data=ext_node_dt1[NODE=='obs'], col='red')
ggp <- ggp + geom_point(aes(x=X, y=OBS),
                        data=ext_node_dt1[NODE=='obs'], col='red')
ggp
```



The monotonicity constraint forces the solution curve to be a horizontal line whenever the data points exhibit locally a negative trend. Compared to the unconstrained solution, the posterior uncertainty band in the range covered by data becomes smaller. More precisely, the uncertainty band becomes more narrow whenever the estimated curve is close to violating the constraint, i.e., when the data points favor a negative trend and the constraint forces the solution being a horizontal line.

To discuss this behavior in more detail, it is insightful to plot the posterior correlation matrix:

```
x_mesh <- ext_node_dt1[NODE=='truefun', X]
image(x=x_mesh, y=x_mesh, cov2cor(as.matrix(U_post1)))
```



In regions where the estimated curve is close to violating the monotonicity constraint, that is where the inequality constraint $f(x_i) < f(x_j)$ for $x_i < x_j$ becomes an equality, strong correlations between function values persist over longer distances between x-values. For instance, all function values between $x = -7.5$ and $x = 0$ are strongly correlated. In contrast, between $x = 0$ and $x = 4$, correlations are more local.

The posterior covariance matrix is calculated only approximately based on a Taylor expansion at the maximum of the posterior density function. At x-values where the estimated curve is not exactly at the boundary defined by the constraint, the corresponding value in the node `trunc_truefun_deriv` is zero. As small local perturbations do not change this fact, the Jacobian matrix (also known as sensitivity matrix) is not influenced by the extra node associated with the pseudo-observations of `trunc_truefun_deriv`. In these regions, the approximate posterior uncertainties of `truefun` are mainly determined by the uncertainties of pseudo-observations of the second derivative node and the uncertainties of the node associated with the synthetic data.

The situation changes, however, if data suggest a negative trend but the monotonicity constraint pushes against that and forces the solution being a horizontal line. Looking at the function values of the posterior, it turns out that it is not really a horizontal line but that the constraint is slightly violated. Consequently, the Jacobian matrix gets influenced by the pseudo-observations associated with `trunc_truefun_deriv` as well, which means that the posterior is strongly biased towards solutions with vanishing first derivative in these regions—horizontal lines. The only degree of freedom that remains in the posterior is shifting all function values up or down in these regions.

In conclusion, if among the solutions that are likely according the posterior distribution, some are affected by the constraint and some are not, an approximate covariance matrix based on a Taylor approximation at the posterior maximum won't necessarily reflect well the true posterior covariance matrix. A mean value and covariance matrix obtained by Monte Carlo sampling from the posterior distribution may summarize the posterior distribution better in those cases.

Case 2: Enforcing convexity

Another possible constraint on the shape of a function is convexity. For a convex function, the slope is expected to increase or stay the same when going from smaller to larger x-values. Mathematically, this could be expressed as $f'(x_i) < f'(x_j)$ if $x_i < x_j$.

We can implement a convexity constraint by introducing a node `truefun_2nd_deriv_det` that is associated with the second derivative of `truefun`. We then link this node to another new node `trunc_truefun_2nd_deriv` by a non-linear mapping. This mapping propagates the second derivative as it is if it is negative and substitutes it by zero otherwise, i.e., $y = \min(f''(x), 0)$. By introducing pseudo-observations of `trunc_truefun_2nd_deriv` we implement the penalty method.

The definition of the `truefun_2nd_deriv_det` node looks as follows:

```
truefun_2nd_deriv_det_dt <- data.table(
  NODE = "truefun_2nd_deriv_det",
  PRIOR = 0,
  UNC = 0,
  OBS = NA,
  X = truefun_dt[NODE=='truefun', head(X[-1], n=-1)]
)
```

The specification `UNC=0` makes this node deterministic, which means that its associated values are fully determined by the propagated values over incoming links from other nodes. We discard the first and last x-value of `truefun` as mesh points in this node, because a second derivative mapping (see `?create_derivative2nd_map`) always needs a neighboring x-value on the left and on the right to determine the second derivative.

Next, we define the node `trunc_truefun_2nd_deriv`, which is associated with the truncated second derivatives:

```
trunc_truefun_2nd_deriv_dt <- data.table(
  NODE = "trunc_truefun_2nd_deriv",
  PRIOR = 0,
  UNC = 1e-1,
  OBS = 0,
  X = truefun_2nd_deriv_det_dt[, X]
)
```

To implement the penalty method, we introduce pseudo-observations by specifying `OBS=0` and a small associated uncertainty in `UNC`.

To add these new nodes to the node data table, we execute:

```
ext_node_dt2 <- copy(node_dt)
ext_node_dt2[, IDX := NULL]
ext_node_dt2 <- rbindlist(list(ext_node_dt2, truefun_2nd_deriv_det_dt,
                               trunc_truefun_2nd_deriv_dt))
ext_node_dt2[, IDX := seq_len(.N)]
```

Now we can create a mapping to link from `truefun` to the node `truefun_2nd_deriv_det` associated with the second derivative:

```
truefun_to_2nd_deriv_det_mapdef <- list(
  maptype = "derivative2nd_map",
  mapname = "truefun_to_2nd_deriv_det",
  src_idx = ext_node_dt2[NODE=="truefun", IDX],
  tar_idx = ext_node_dt2[NODE=="truefun_2nd_deriv_det", IDX],
  src_x = ext_node_dt2[NODE=="truefun", X],
  tar_x = ext_node_dt2[NODE=="truefun_2nd_deriv_det", X]
```

```
)
```

The link of the node `truefun_2nd_deriv_det` to a truncated version of the second derivative is given by a non-linear mapping (see `?create_nonlinear_map`):

```
truefun_2nd_deriv_to_trunc_2nd_deriv_mapdef <- list(
  mapname = "truefun_2nd_deriv_to_trunc_2nd_deriv",
  maptype = "nonlinear_map",
  src_idx = ext_node_dt2[NODE=="truefun_2nd_deriv_det", IDX],
  tar_idx = ext_node_dt2[NODE=="trunc_truefun_2nd_deriv", IDX],
  funname = "limiter",
  minvalue = -Inf,
  maxvalue = 0
)
```

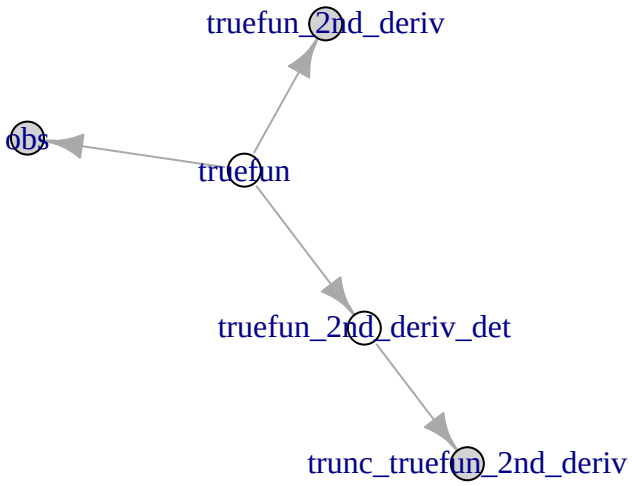
We go again for the `limiter` option with limits from `-Inf` to zero.

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

```
ext_compmap2_mapdef <- list(
  mapname = "compmap",
  maptype = "compound_map",
  maps = list(truefun_to_obs_mapdef,
              truefun_to_2nd_deriv_mapdef, truefun_to_2nd_deriv_det_mapdef,
              truefun_2nd_deriv_to_trunc_2nd_deriv_mapdef)
)
ext_compmap2 <- create_map(ext_compmap2_mapdef)
```

It is insightful to plot the Bayesian network and recall the meanings of the nodes:

```
grph <- get_network_structure(ext_compmap2$getMaps(), ext_node_dt2$NODE, ext_node_dt2$OBS)
plot(grph)
```



Nodes with gray background are observed whereas nodes with white background are not. The node `truefun` is associated with the true function. It links to the measurements in node `obs` and to the node `truefun_2nd_deriv` with the pseudo-observations of the second derivative to enforce smoothness. It also links to `truefun_2nd_deriv_det` with second derivatives. This latter node is only an intermediate step to establish the node `trunc_truefun_2nd_deriv` with truncated second derivatives, which is associated with pseudo-observations to enforce the convexity constraint.

Now we can perform the inference:

```
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, control=list(
                   mincount=10, maxcount=100, maxreject=20))
```

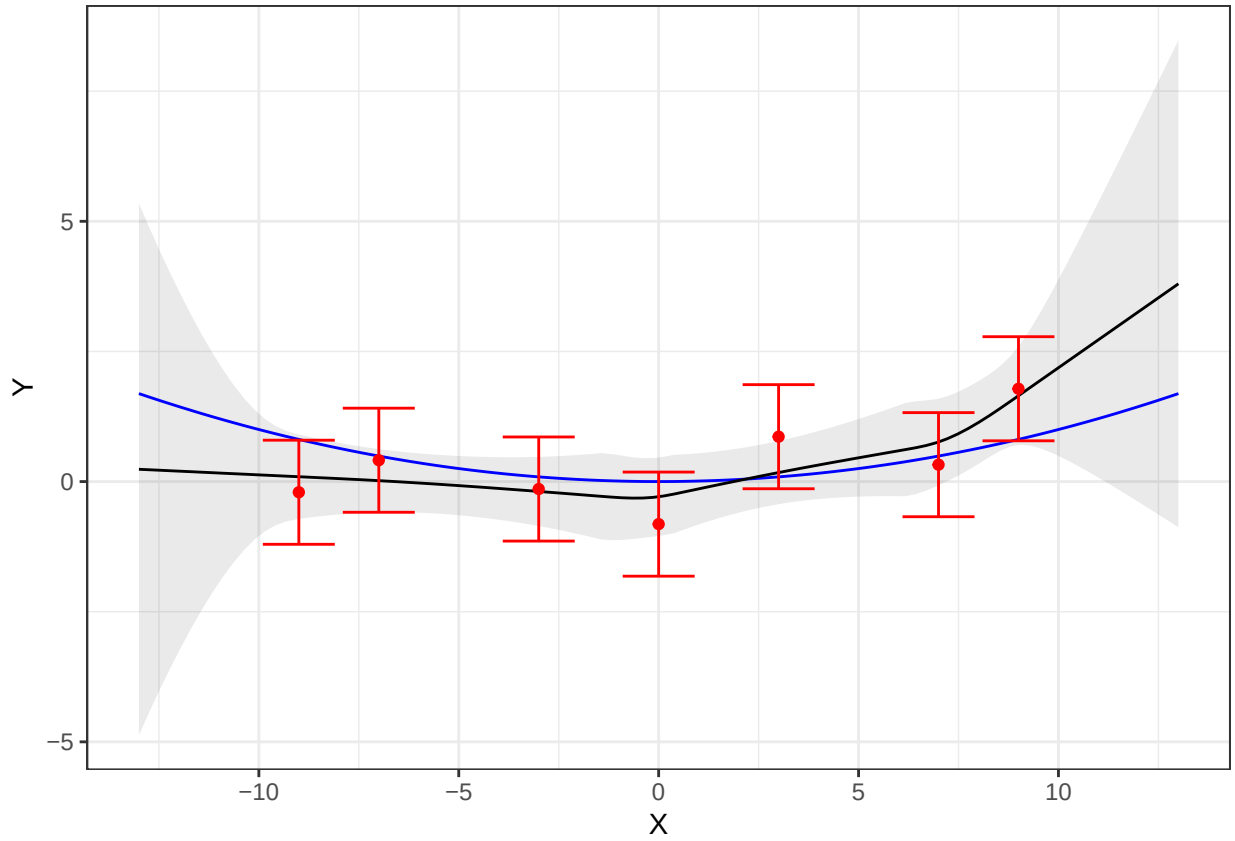
Here is the code to calculate the approximate posterior uncertainties:

```
sel_idcs <- ext_node_dt2[NODE=="truefun", IDX]
U_post <- get_posterior_cov(ext_compmap2, optres$zpost, U_prior,
                            ext_node_dt2$OBS, sel_idcs, sel_idcs)
post_unc <- sqrt(diag(U_post))
ext_node_dt2[sel_idcs, UNC_POST := post_unc]
```

Finally, we can plot the result (the MAP estimate) together with the synthetic data:

```
ext_node_dt2[, POST:=ext_compmap2$propagate(optres$zpost)]
ggp <- ggplot() + theme_bw()
ggp <- ggp + geom_ribbon(aes(x=X, ymin=POST-UNC_POST, ymax=POST+UNC_POST),
                         data=ext_node_dt2[NODE=='truefun'], alpha=0.1)
```

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



The convexity constraint helps that the estimated curve bends more similar to the true parabola. Furthermore, the estimated curve does not exhibit the negative slope in the region left of the data points. Regarding the uncertainty band, the same discussion as for the monotonicity constraint holds: In regions where the solution curve is close to violating the convexity constraint, uncertainties are underestimated. Otherwise, they are potentially overestimated. Monte Carlo sampling can help to get a more reliable picture of allowed solutions according to the posterior.

Summary

In this tutorial, we explored how a monotonicity and convexitiy constraint can be incorporated into a Bayesian network. For both cases, we linked the node associated with the true function to another node associated with the first or second derivative, respectively. This latter node was then linked to another new node by a non-linear mapping. This non-linear mapping propagated values of the derivative as they were if they fell into the range violating the constraint and otherwise replaced them by zero. The specification of pseudo-observations on the node associated with the truncated values of the derivatives then amounted to implementing the penalty method for constrained optimization.