Solutions Sheet 02

On Google Colab, you can install the required libraries with the following commands:

```
!apt install libgraphviz-dev
!pip install pygraphviz
!pip install liesel
!pip install plotnine
```

Exercise 1: Statistical models as directed acyclic graphs

- To get an overview of a statistical model, it can be helpful to represent them as directed acyclic graphs. In our usage, they consist of two main building blocks:
  - Nodes: The variables that come up in the model
  - Edges: The connections between the variables. These edges are directed, that means they represent the flow of information from one node to another.
  - The graph is **acyclic**, which means no node can become it's own input (or the input to its inputs).

We differentiate nodes based on two concepts:

- **Strong** and **weak** nodes: Strong nodes introduce new information into the graph, while weak nodes are deterministically calculated from other nodes in the graph.
- **Random** and **nonrandom** nodes. Nodes with an associated probability distribution are random. Other nodes are nonrandom.

We also differentiate between two types of edges:

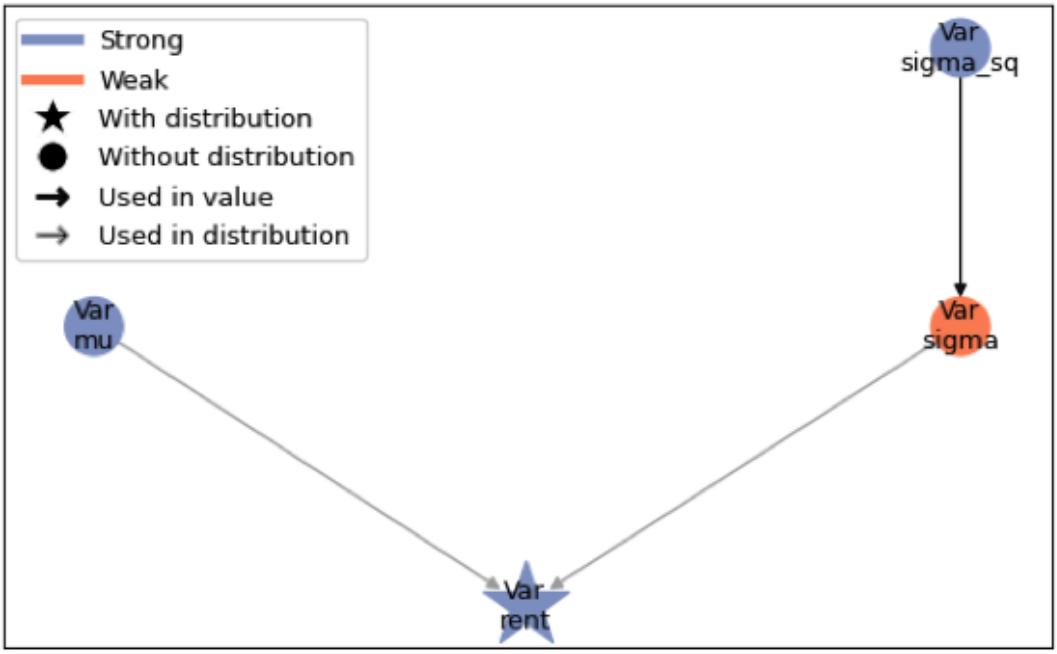
- Used in **value**: For edges that represent the flow of information as inputs for deterministic computations.
- Used in **distribution**: For edges that represent the flow of information as inputs to probability distributions.

Subtask a): The first graph

In this model, we have the following nodes:

- rent: The observed values and observation model for our response variable.
  - This node is *strong*, because the observed values enter the graph with this node - they are not calculated.
  - This node is *random*, because it has an associated probability distribution.
- μ: The mean of the response distribution.
  - This node is *strong*, because the value of μ enters the graph with this node.
  - This node is *nonrandom*, because it does not have a probability distribution.
- σ^2: The variance of the response distribution.
  - This node is *strong*, because the value of σ^2 enters the graph with this node.
  - This node is *nonrandom*, because it does not have a probability distribution.
- σ: The scale of the response distribution. We include it here mainly for illustration.

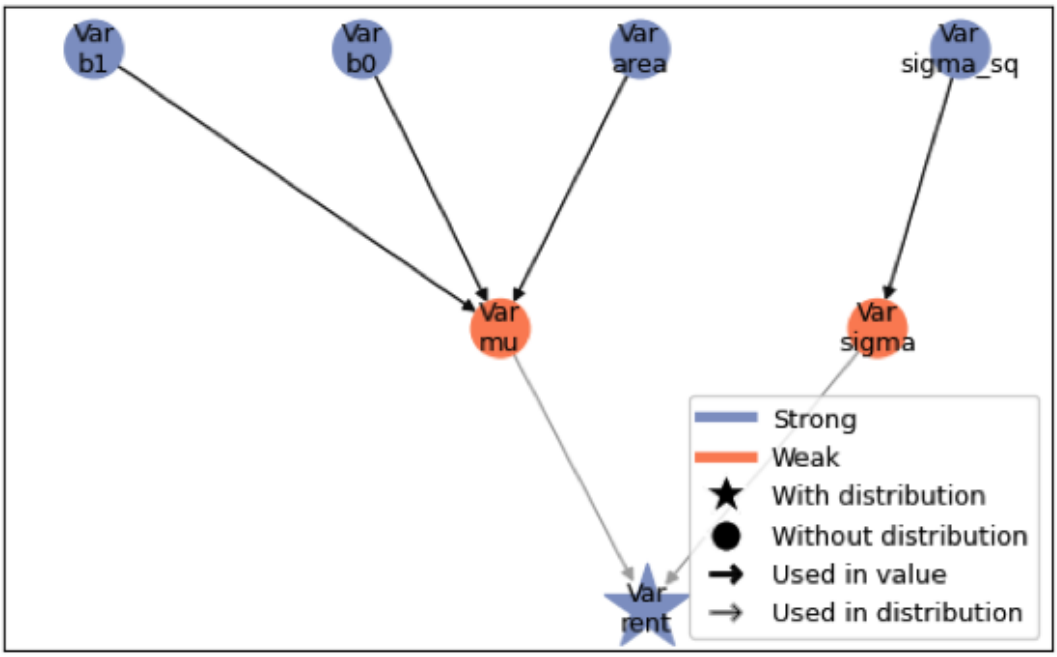
- This node is *weak*, because the value of σ is calculated as the square root of σ^2.
- This node is *nonrandom*, because it does not have a probability distribution.



Subtask b): Adding a regression model

In this model, we observe the following changes:

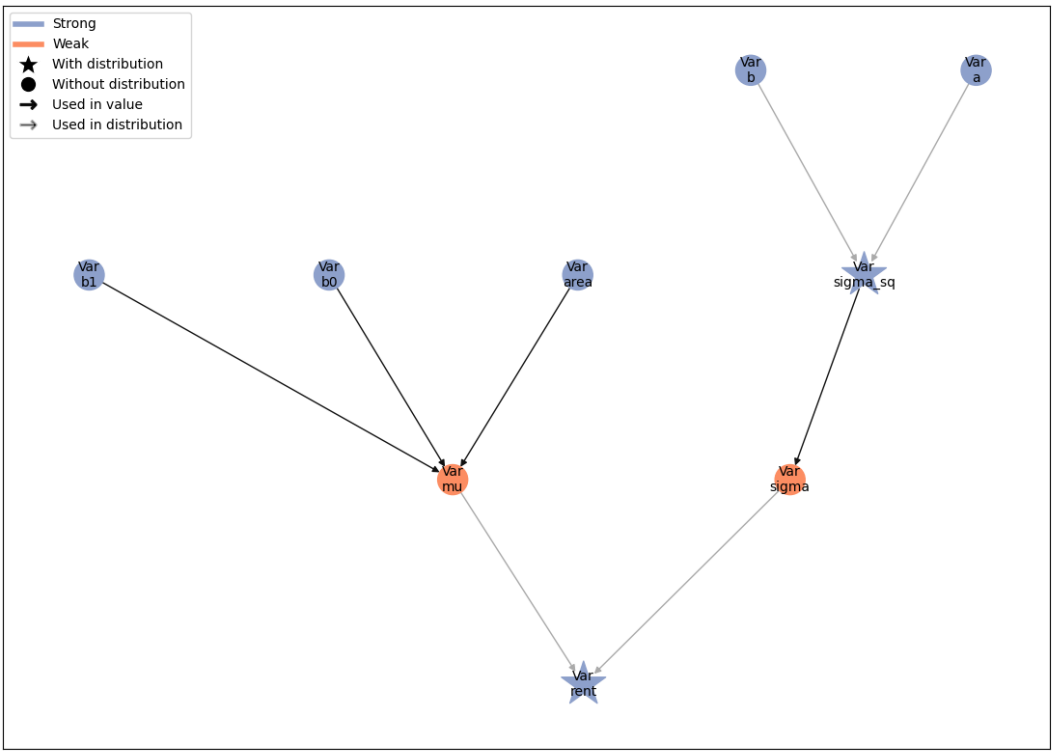
- β_0: The model intercept is added as a new node.
  - This node is *strong*, because the value of β_0 enters the graph with this node.
  - This node is *nonrandom*, because it does not have a probability distribution.
- β_1: The regression coefficient for the covariate area is added as a new node.
  - This node is *strong*, because the value of β_0 enters the graph with this node.
  - This node is *nonrandom*, because it does not have a probability distribution.
- area: The observed values of the covariate area
  - This node is *strong*, because the values of area enter the graph with this node.
  - This node is *nonrandom*, because it does not have a probability distribution.
- μ: The mean of the response distribution is now a deterministic function of other nodes in the graph through the model $\mu_i = \beta_0 + \beta_1 \mathrm{area}_i$. Thus, it changes from a strong node into a *weak* node. It remains a nonrandom node.



Subtask c): Adding a prior

In this model, we observe the following changes:

- σ^2: The variance of the response distribution now has its own probability distribution, the inverse gamma prior, so that it changes from a nonrandom to a *random* node. It remains a strong node.
- a: The prior shape (=concentration) of the inverse gamma prior for σ^2.
  - This node is *strong*, because the value of a enters the graph with this node.
  - This node is *nonrandom*, because it does not have a probability distribution.
- b: The prior scale of the inverse gamma prior for σ^2.
  - This node is *strong*, because the value of b enters the graph with this node.
  - This node is *nonrandom*, because it does not have a probability distribution.



Exercise 2: Your first Liesel model

Liesel is built with the graph representation in mind, providing you the basic building blocks. The four fundamental blocks are:

- `lsl.Var`: A statistical variable, always shows up in the model graph.
  - A `lsl.Var` object is *strong* and *nonrandom* by default.
  - It can be associated with a probability distribution via `lsl.Dist`, making it *random*.
  - It can wrap a function via `lsl.Calc`, making it *weak*.
- `lsl.Dist`: Wraps a probability distribution.
- `lsl.Calc`: Wraps a function.
  - Shows up in the model graph only if wrapped by a `lsl.Var`.
- `lsl.Data`: Holds constant auxiliary data, for example for storing values that are precomputed for convenience or efficiency, but not of major modeling interest.

Liesel also provides two convenience functions:

- `lsl.obs`: Initializes a `lsl.Var` and sets the `lsl.Var.observed` flag to `True`. This makes sure that, if the variable is random, its log probability/density is added to the model's log likelihood.
- `lsl.param`: Initializes a `lsl.Var` and sets the `lsl.Var.parameter` flag to `True`. This makes sure that, if the variable is random, its log probability/density is added to the model's log prior.

```{python}
import liesel.goose as gs
```

Exercise 2a): A minimal model

First, we import the data. Here, we transform the values directly to the data type `float32`, since JAX works with 32-bit floats. Liesel would also convert these values automatically, but we like to be explicit here.

```{python}
import pandas as pd

rent99 = pd.read_csv("https://s.gwdg.de/mzAkHV")
area = rent99.area.to_numpy("float32")
rent = rent99.rent.to_numpy("float32")
```

Next, we set up the leaf nodes. Even though they are not random, we initialize the mean and variance as parameters. Both receive a value and a name.

```{python}
import liesel.model as lsl

mu = lsl.param(0.0, name="mu")
sigma_sq = lsl.param(10.0, name="sigma_sq")
```

Next, we use our first calculator. The `lsl.Calc` class receives the function to execute as its first argument, followed by the inputs. We use pure, jittable functions in `lsl.Calc` objects, and can very often simply resort to using functions available from `jax.numpy`.

We wrap the calculator in a `lsl.Var`, because we want the `"sigma"` node to show up in the model graph.

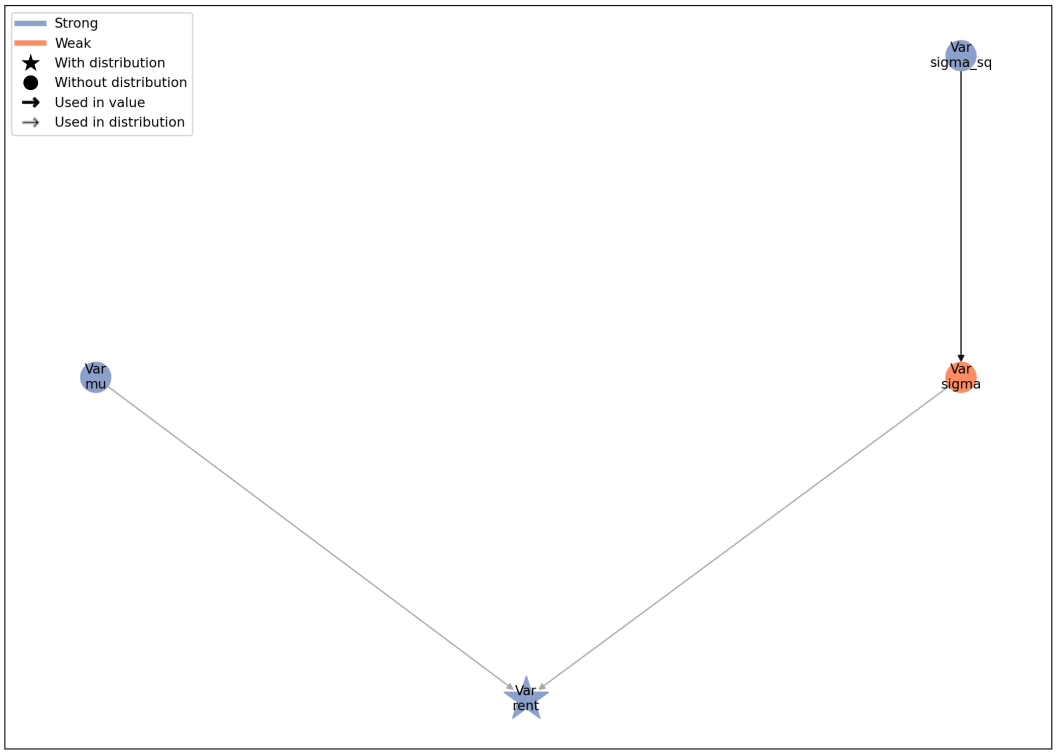
```{python}
import jax.numpy as jnp

sigma = lsl.Var(
  lsl.Calc(jnp.sqrt, sigma_sq), name="sigma"
)
```

Next, we set up the response node with its probability distribution. We actually start with the distribution. The `lsl.Dist` class wraps probability distributions that follow the `tensorflow_probability` interface. The `lsl.Dist` instance is then fed as the second argument to `lsl.obs`.

```{python}
import tensorflow_probability.substrates.jax.distributions as tfd

y_dist = lsl.Dist(tfd.Normal, loc=mu, scale=sigma)
y = lsl.obs(rent, y_dist, name="rent")
```

Our first model is almost complete. We bring everything together by initializing a `lsl.GraphBuilder` and adding our response node. Since all other nodes in the graph can be found as inputs to this response node, we only need the GraphBuilder to know about this one. All other nodes will be found automatically. We finish this task by building our model and plotting our graph.

```{python}
gb = lsl.GraphBuilder().add(y)

model = gb.build_model()
lsl.plot_vars(model)
```



We can inspect the log probability, log likelihood and log prior of the fully built model:

```{python}
print(model.log_prob)
print(model.log_lik)
print(model.log_prior)
```

```
-6534.502
-6534.502
0
```

We can inspect the log probability/density of all of our variables:

```{python}
print(mu.log_prob)
print(sigma_sq.log_prob)
print(sigma.log_prob)
print(y.log_prob) #
print(y.log_prob.sum())
```

```
0.0
0.0
0.0
[-2.2297626 -2.1312566 -2.1213305 ... -2.1781018 -2.093863  -2.850203 ]
-6534.502
```

Exercise 2b): Adding a regression model

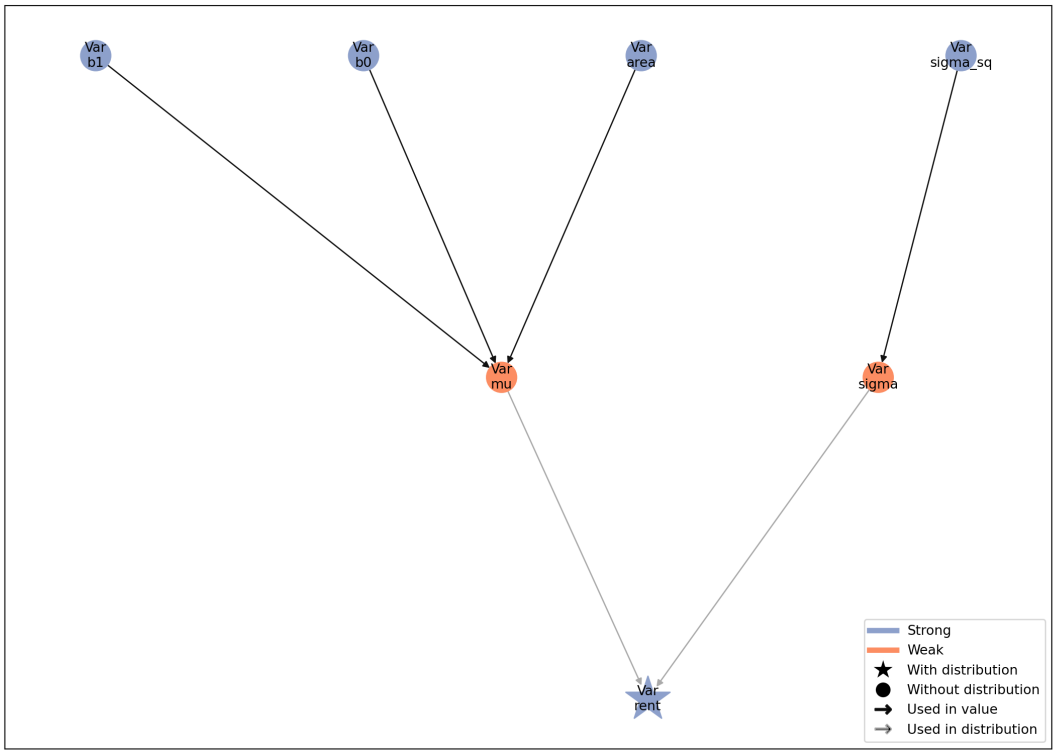
The main difference here lies in the way we define the response's mean. We have three new nodes: the observed values of area, the intercept β_0 and the coefficient β_1.

We then define μ as the output of a short function and wrap it with a `lsl.Calc` and a `lsl.Var`.

```{python}
x = lsl.obs(area, name="area")

b0 = lsl.param(0.0, name="b0")
b1 = lsl.param(0.0, name="b1")

def linear_model(x, b0, b1):
    return b0 + x*b1

mu = lsl.Var(
  lsl.Calc(linear_model, x=x, b0=b0, b1=b1), name="mu"
)
```

The rest of the model stays the same, so we can copy from above:

```{python}
sigma_sq = lsl.param(10.0, name="sigma_sq")
sigma = lsl.Var(lsl.Calc(jnp.sqrt, sigma_sq), name="sigma")

y_dist = lsl.Dist(tfd.Normal, loc=mu, scale=sigma)
y = lsl.obs(rent, y_dist, name="rent")

gb = lsl.GraphBuilder().add(y)
model = gb.build_model()
lsl.plot_vars(model)
```



Exercise 2c): Adding a prior

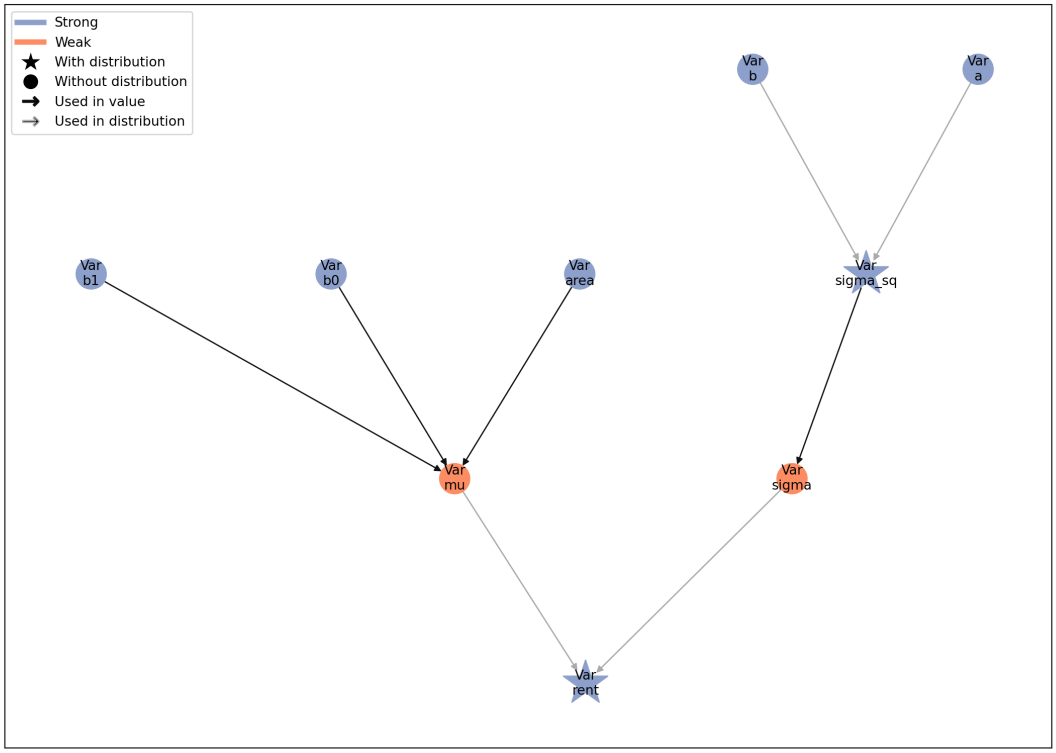
The main difference here lies in the way we define the response's variance. That is, we now set up a prior for the `sigma_sq` node.

```{python}
a = lsl.Var(0.01, name="a")
b = lsl.Var(0.01, name="b")

sigma_sq_dist = lsl.Dist(tfd.InverseGamma, concentration=a, scale=b)
sigma_sq = lsl.param(1.0, sigma_sq_dist, name="sigma_sq")

sigma = lsl.Var(lsl.Calc(jnp.sqrt, sigma_sq), name="sigma")
```

The rest of the model stays the same, so we can copy from above:

```{python}
x = lsl.obs(area, name="area")

b0 = lsl.param(0.0, name="b0")
b1 = lsl.param(0.0, name="b1")

def linear_model(x, b0, b1):
    return b0 + x*b1
```

```
mu = lsl.Var(lsl.Calc(linear_model, x=x, b0=b0, b1=b1), name="mu")

y_dist = lsl.Dist(tfd.Normal, loc=mu, scale=sigma)
y = lsl.obs(rent, y_dist, name="rent")

gb = lsl.GraphBuilder().add(y)
model = gb.build_model()
lsl.plot_vars(model)
```



We can inspect the log probability, log likelihood and log prior of the fully built model:

```{python}
print(model.log_prob)
print(model.log_lik)
print(model.log_prior)
```

```
-4377.3247
-4372.669
-4.65553
```

We can inspect the log probability/density of all of our variables:

```{python}
print(mu.log_prob)
print(sigma_sq.log_prob)
```

```
print(sigma.log_prob)
print(y.log_prob) #
print(y.log_prob.sum())
```

```
0.0
-4.65553
0.0
[-2.5142539 -1.5291944 -1.4299333 ... -1.997647  -1.155258  -8.71866  ]
-4372.669
```

Exercise 3: Manipulate a Graph

Exercise 3a): Update existing prior

We can simply override the `lsl.Var.value` attribute of the relevant nodes.

```{python}
print(model.log_prob)
a.value = 2.0
b.value = 0.5
print(model.log_prob)
```

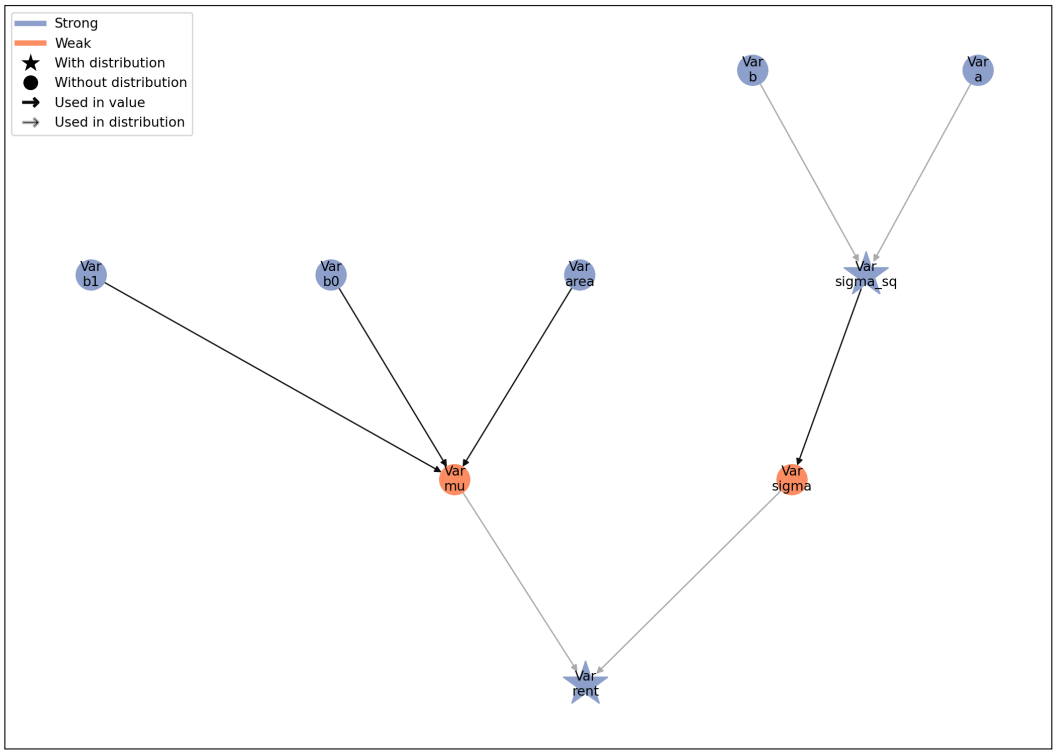
```
-4377.3247
-4374.555
```

Exercise 3b): Replace existing prior

- A `lsl.Model` object assumes a static graph. That means, the values of nodes can change, but the nodes themselves stay fixed.
- To update the graph, we therefore extract the nodes and variables from our model and set up a new graph using our response node.

```{python}
_, vars_ = model.pop_nodes_and_vars()

gb = lsl.GraphBuilder().add(vars_["rent"])
gb.plot_vars()
```

```
GraphBuilder(0 nodes, 1 vars)
```



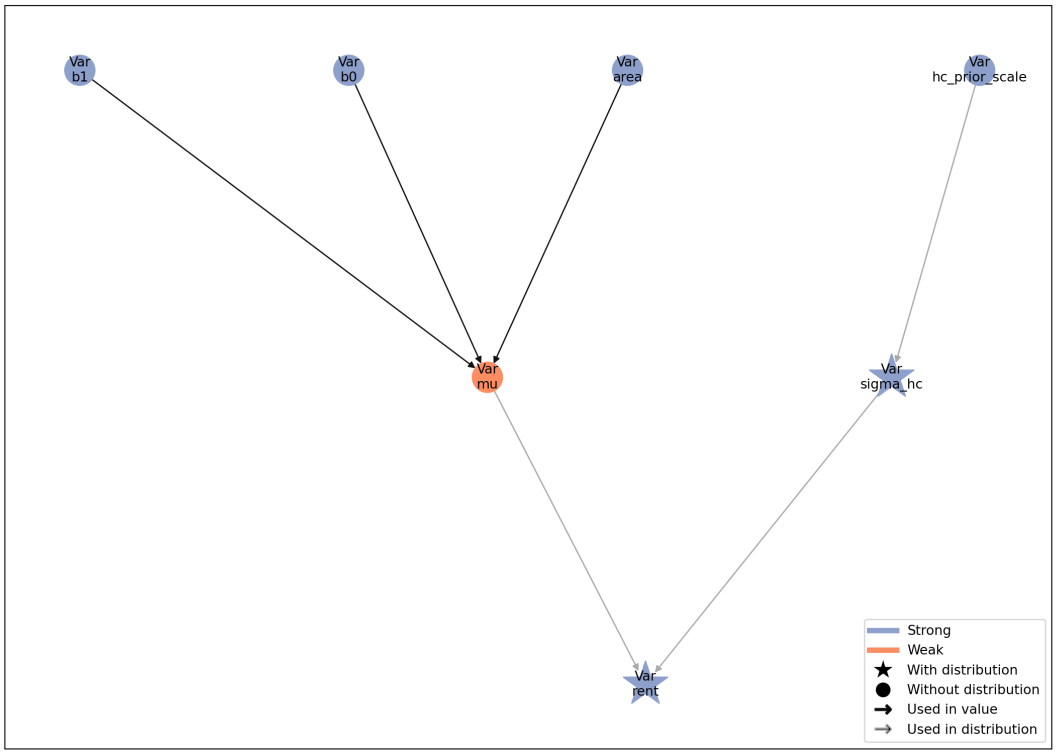
Note that we are replacing a weak, nonrandom `lsl.Var` with a strong, random `lsl.Var`. - the existing σ is weak and nonrandom, because it is the square root of the strong, random node σ^2. - the new σ node will be strong and random, because we include it directly without referring to σ^2, and because we place the prior directly on σ.

```{python}
hc_prior_scale = lsl.Var(25.0, name="hc_prior_scale")

sigma_dist = lsl.Dist(tfd.HalfCauchy, loc=0.0, scale=hc_prior_scale)

sigma_hc = lsl.param(1.0, sigma_dist, name="sigma_hc")
```

The GraphBuilder then allows us to replace the node in question.

```{python}
gb.replace_var(sigma, sigma_hc)
gb.plot_vars()
```

```
GraphBuilder(0 nodes, 1 vars)
GraphBuilder(0 nodes, 1 vars)
```

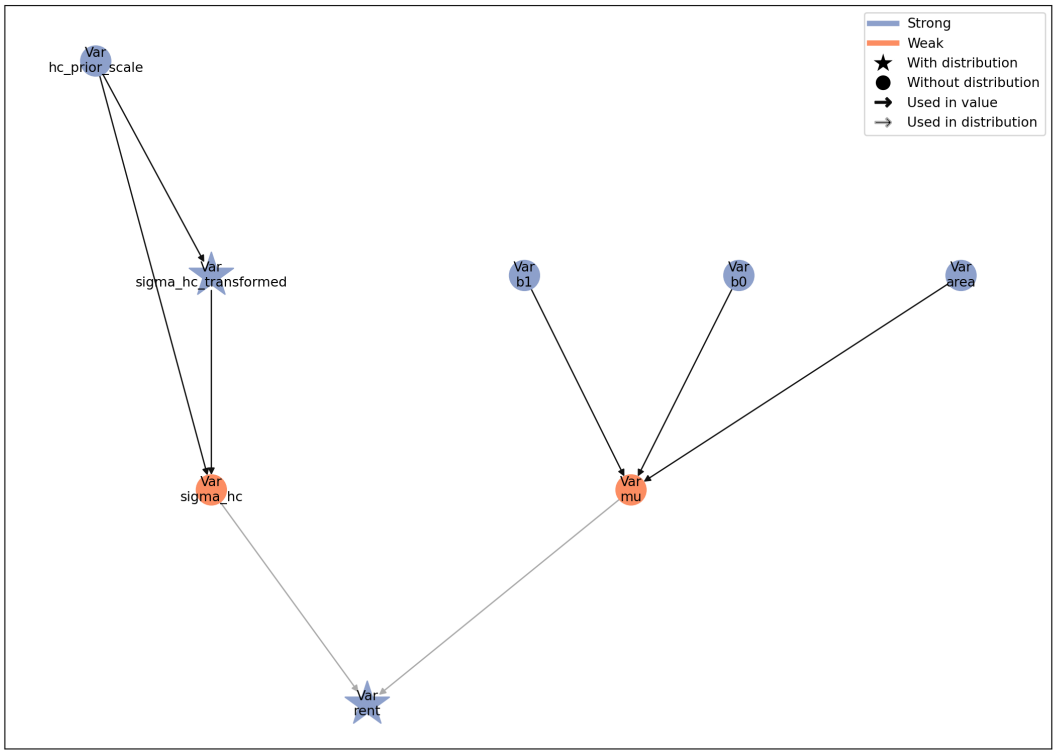


Exercise 3c): Transform a parameter

- To transform a parameter, we again use functionality offered by TensorFlow, namely its bijector interface.
- It offers a wide range of useful bijective functions that we can use a inverse link functions.
- The bijectors are named after their "forward" transformation, i.e. the *inverse* link function.
- In this case, since we want to use a logarithmic transformation of the parameter σ, we need the `Exp` bijector.
- The `lsl.GraphBuilder` offers the function `lsl.GraphBuilder.transform`, which will...
    i. ... create the new transformed node and turns the original node into an appropriate calculator node to hold the output of the calculation.
    ii. ... automatically apply the change-of-variables theorem for us such that the prior will be defined for the transformed variable.

```{python}
import tensorflow_probability.substrates.jax.bijectors as tfb

gb.transform(sigma_hc, tfb.Exp)
gb.plot_vars()
```

```
Var(name="sigma_hc_transformed")
GraphBuilder(0 nodes, 2 vars)
```



Exercise 4: Sample from the posterior using Goose

Here, we apply the knowledge from Exercise Sheet 01.

```{python}
model = gb.build_model()
interface = gs.LieselInterface(model)

eb = gs.EngineBuilder(seed=1, num_chains=4)
eb.add_kernel(gs.NUTSKernel(["b0", "b1"]))
eb.add_kernel(gs.IWLSKernel(["sigma_hc_transformed"]))

eb.set_duration(warmup_duration=1000, posterior_duration=1000)
eb.set_model(interface)
eb.set_initial_values(model.state)
eb.set_engine_seed(seed=2)

engine = eb.build()
```

```
liesel.goose.builder - WARNING - No jitter functions provided. The initial values
won't be jittered
```

```
liesel.goose.engine - INFO - Initializing kernels...
liesel.goose.engine - INFO - Done
```

```{python}
engine.sample_all_epochs()
```

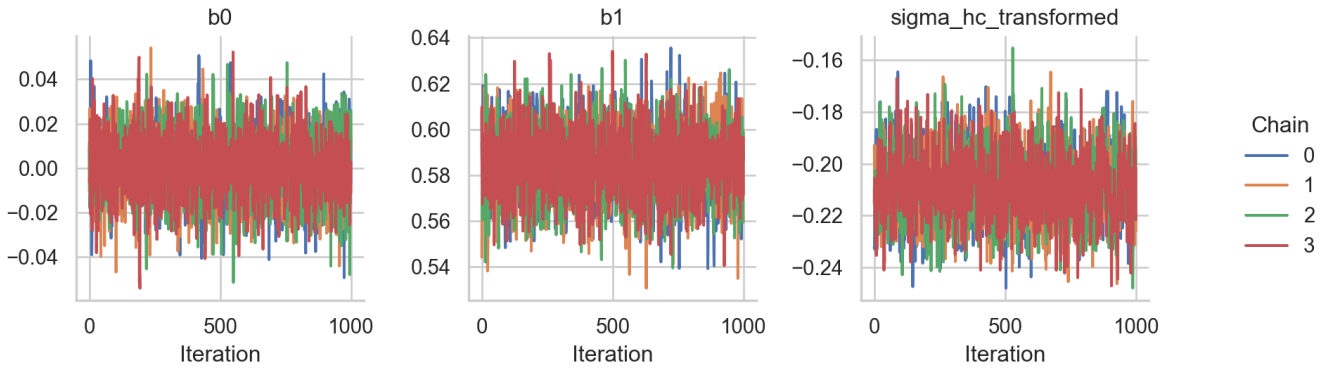
```
liesel.goose.engine - INFO - Starting epoch: FAST_ADAPTATION, 75 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 3, 3, 2, 2 / 75
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 25 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 1, 1, 1, 1 / 25
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 50 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 2, 1, 1, 1 / 50
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 100 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 1, 1, 1, 1 / 100
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 200 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 1, 2, 1, 1 / 200
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 500 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 1, 1, 1, 1 / 500
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: FAST_ADAPTATION, 50 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 1, 2, 1, 1 / 50
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Finished warmup
liesel.goose.engine - INFO - Starting epoch: POSTERIOR, 1000 transitions, 25 jitted
together
liesel.goose.engine - INFO - Finished epoch
```

```{python}
results = engine.get_results()
gs.Summary(results)
```

```
Parameter summary:

                              kernel      mean  ...     ess_tail      rhat
parameter            index                     ...
b0                   ()     kernel_00  0.000352  ...  2379.037561  1.001262
b1                   ()     kernel_00  0.584753  ...  2573.809994  1.000639
sigma_hc_transformed ()     kernel_01 -0.208348  ...  2420.258250  1.001259

[3 rows x 10 columns]

Error summary:

                                                    count relative
kernel    error_code error_msg            phase
kernel_00 1          divergent transition warmup       37  0.00925
                                          posterior     0      0.0

/Users/johannesbrachem/Documents/git/cmstats-tutorial/env/lib/python3.10/site-
packages/liesel/goose/summary_m.py:362: FutureWarning: The behavior of array
concatenation with empty entries is deprecated. In a future version, this will no
longer exclude empty items when determining the result dtype. To retain the old
behavior, exclude the empty entries before the concat operation.
  df = pd.concat({
```

```{python}
gs.plot_trace(results)
```

```
<seaborn.axisgrid.FacetGrid object at 0x2cb539f00>
```



Exercise 5: A semiparametric model in Liesel

Subtask a): Bayesian P-spline

We start by representing $s(\text{area}_i)$ with B-spline bases of order 3 and spline coefficients $\beta_1, \dots, \beta_J$:

$$s(\text{area}_i) = \sum_{j=1}^{J} B_j(\text{area}_i)\beta_j.$$

For the spline coefficients $\boldsymbol{\beta} = [\beta_1, \ldots, \beta_J]^T$, we define a second-order random walk prior, which is a partially improper multivariate normal prior of the form

$$f(\boldsymbol{\beta}|\tau^2) \propto \left(\frac{1}{\tau^2}\right)^{(L-2)/2} \exp\left(-\frac{1}{2\tau^2}\boldsymbol{\beta}^T\mathbf{K}\boldsymbol{\beta}\right).$$

Here, the penalty matrix $\mathbf{K}$ is defined as the crossproduct of a second-differences matrix $\mathbf{D}$ of dimension $(J-2) \times J$, so that $\mathbf{K} = \mathbf{D}^T\mathbf{D}$ with

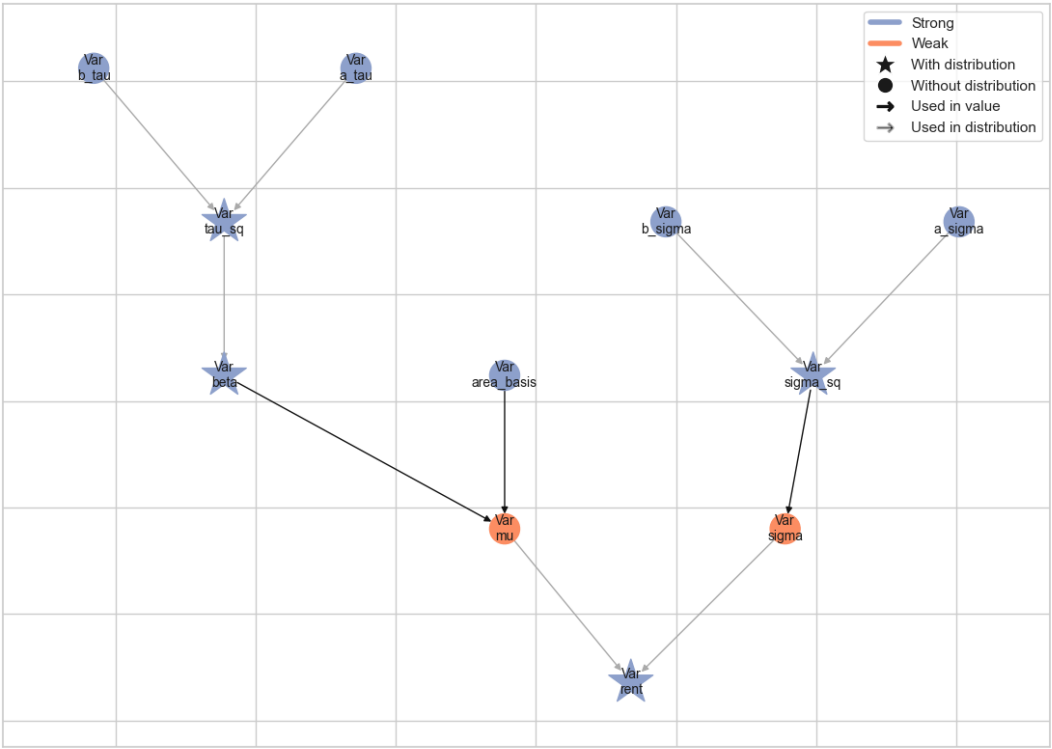
$$\mathbf{D} = \begin{bmatrix} 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & -2 & 1 \end{bmatrix}.$$

The setup is completed by defining a hyperprior for the variance of the random walk, the inverse smoothing parameter τ^2. Here, we choose an inverse gamma prior:

$$\tau^2 \sim \mathcal{IG}(0.01, 0.01).$$

Subtask b): Model graph

This is the model graph:



Subtask c): Implementation in Liesel

Spline coefficient subgraph

Let us start with the subgraph that represents $\boldsymbol{\beta}$. First, we define the hyperprior for τ^2:

```{python}
a = lsl.Var(0.01, name="a")
b = lsl.Var(0.01, name="b")
tau_sq_prior = lsl.Dist(tfd.InverseGamma, concentration=a, scale=b)
tau_sq = lsl.param(10.0, tau_sq_prior, name="tau_sq")
```

Next, we construct the penalty matrix $\mathbf{K}$.

```{python}
nparam = 20
D = jnp.diff(jnp.eye(nparam), n=2, axis=0)
K = D.T @ D
```

Now we can set up the partially improper normal prior:

```{python}
from liesel.distributions import MultivariateNormalDegenerate

beta_prior = lsl.Dist(
  MultivariateNormalDegenerate.from_penalty, loc=0.0, var=tau_sq, pen=K
)
```

Finally, we can set up the $\boldsymbol{\beta}$ node, bringing everything together.

```{python}
beta = lsl.param(jnp.zeros(20), beta_prior, name="beta")
```

Location subgraph

Next, we import the prepared matrix of basis function evaluations using numpy. We also create a `lsl.Var` that holds the basis matrix.

```{python}
import numpy as np
basis_matrix = np.loadtxt("https://s.gwdg.de/LZnQMC")

area_basis = lsl.obs(basis_matrix, name="area_basis")
```

Next, we include the location calculator. This calculator implements

$$\boldsymbol{s}(\mathbf{area}) = \mathbf{B}\boldsymbol{\beta},$$

where $\mathbf{B}$ is the matrix of basis function evaluations.

```{python}
mu = lsl.Var(
  lsl.Calc(jnp.dot, area_basis, beta), name="mu"
)
```

Scale subgraph

This is just an application of what we did earlier.

```{python}
a_sigma = lsl.Var(0.01, name="a_sigma")
b_sigma = lsl.Var(0.01, name="b_sigma")
sigma_sq_dist = lsl.Dist(tfd.InverseGamma, concentration=a_sigma, scale=b_sigma)
sigma_sq = lsl.param(10.0, sigma_sq_dist, name="sigma_sq")
sigma = lsl.Var(lsl.Calc(jnp.sqrt, sigma_sq), name="sigma")
```

Response

Everything now comes together in the response node.

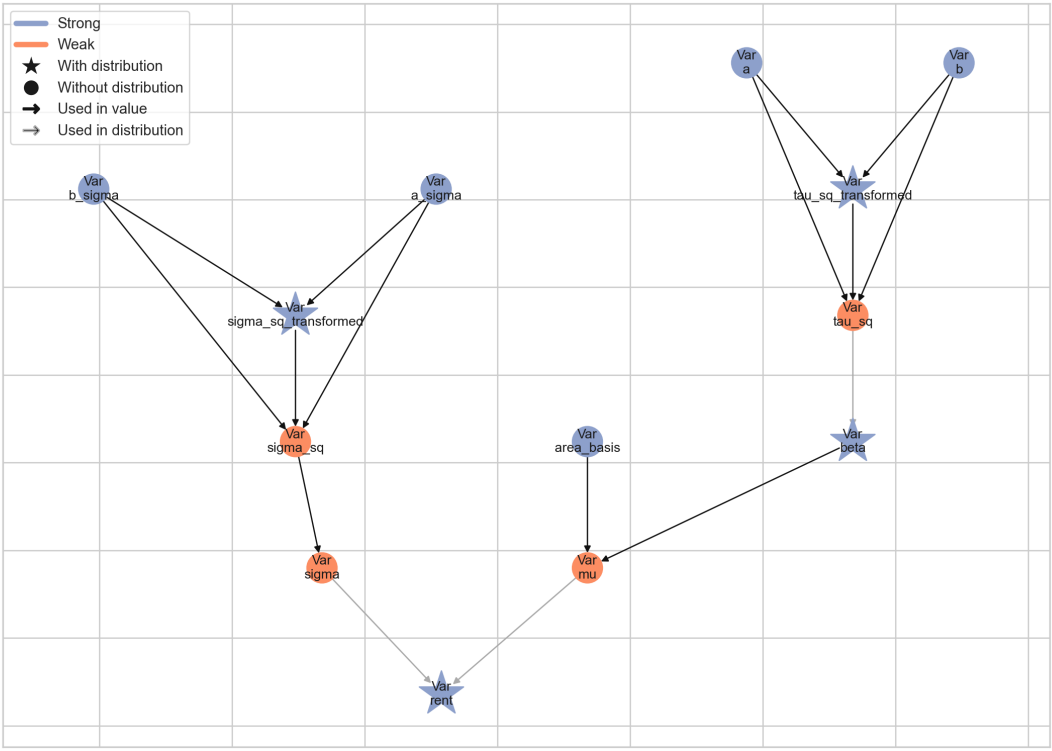
```{python}
y_dist = lsl.Dist(tfd.Normal, loc=mu, scale=sigma)
y = lsl.obs(rent, y_dist, name="rent")
```

Model

We finally build the model.

```{python}
gb = lsl.GraphBuilder().add(y)
gb.transform(tau_sq, tfb.Exp)
gb.transform(sigma_sq, tfb.Exp)
model = gb.build_model()
lsl.plot_vars(model)
```

```
liesel.model.model - INFO - Converted dtype of Data(name="area_basis_value").value
Var(name="tau_sq_transformed")
Var(name="sigma_sq_transformed")
```



Subtask d): Setting up the sampling engine

Setting up the engine works pretty much like in the earlier examples. Just don't forget that τ^2 and σ^2 are sampled after transformation here, so we have to append `"_transformed"` to the respective node names.

```{python}
eb = gs.EngineBuilder(seed=10, num_chains=4)

eb.add_kernel(gs.NUTSKernel(["beta", "tau_sq_transformed"]))
eb.add_kernel(gs.NUTSKernel(["sigma_sq_transformed"]))

eb.set_duration(warmup_duration=1000, posterior_duration=1000)
eb.set_model(gs.LieselInterface(model))
eb.set_initial_values(model.state)

engine = eb.build()
```

```
liesel.goose.builder - WARNING - No jitter functions provided. The initial values
won't be jittered
liesel.goose.engine - INFO - Initializing kernels...
liesel.goose.engine - INFO - Done
```

After the engine has been set up, the sample from all epochs.

```{python}
engine.sample_all_epochs()
```

```
liesel.goose.engine - INFO - Starting epoch: FAST_ADAPTATION, 75 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 4, 7, 4, 3 / 75
transitions
liesel.goose.engine - WARNING - Errors per chain for kernel_01: 3, 2, 5, 2 / 75
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 25 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 1, 1, 1, 2 / 25
transitions
liesel.goose.engine - WARNING - Errors per chain for kernel_01: 1, 1, 1, 1 / 25
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 50 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 2, 2, 4, 1 / 50
transitions
liesel.goose.engine - WARNING - Errors per chain for kernel_01: 1, 2, 1, 1 / 50
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 100 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 5, 2, 1, 3 / 100
transitions
liesel.goose.engine - WARNING - Errors per chain for kernel_01: 3, 1, 1, 2 / 100
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 200 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 6, 3, 4, 2 / 200
transitions
liesel.goose.engine - WARNING - Errors per chain for kernel_01: 2, 1, 1, 1 / 200
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: SLOW_ADAPTATION, 500 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 6, 9, 7, 6 / 500
transitions
liesel.goose.engine - WARNING - Errors per chain for kernel_01: 4, 2, 1, 2 / 500
transitions
liesel.goose.engine - INFO - Finished epoch
liesel.goose.engine - INFO - Starting epoch: FAST_ADAPTATION, 50 transitions, 25
jitted together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 3, 1, 1, 1 / 50
transitions
liesel.goose.engine - WARNING - Errors per chain for kernel_01: 2, 1, 2, 2 / 50
transitions
liesel.goose.engine - INFO - Finished epoch
```

```
liesel.goose.engine - INFO - Finished warmup
liesel.goose.engine - INFO - Starting epoch: POSTERIOR, 1000 transitions, 25 jitted
together
liesel.goose.engine - WARNING - Errors per chain for kernel_00: 1, 3, 0, 0 / 1000
transitions
liesel.goose.engine - INFO - Finished epoch
```

Let's take a first look at the summary.

```{python}
results = engine.get_results()
summary = gs.Summary(results)
summary
```

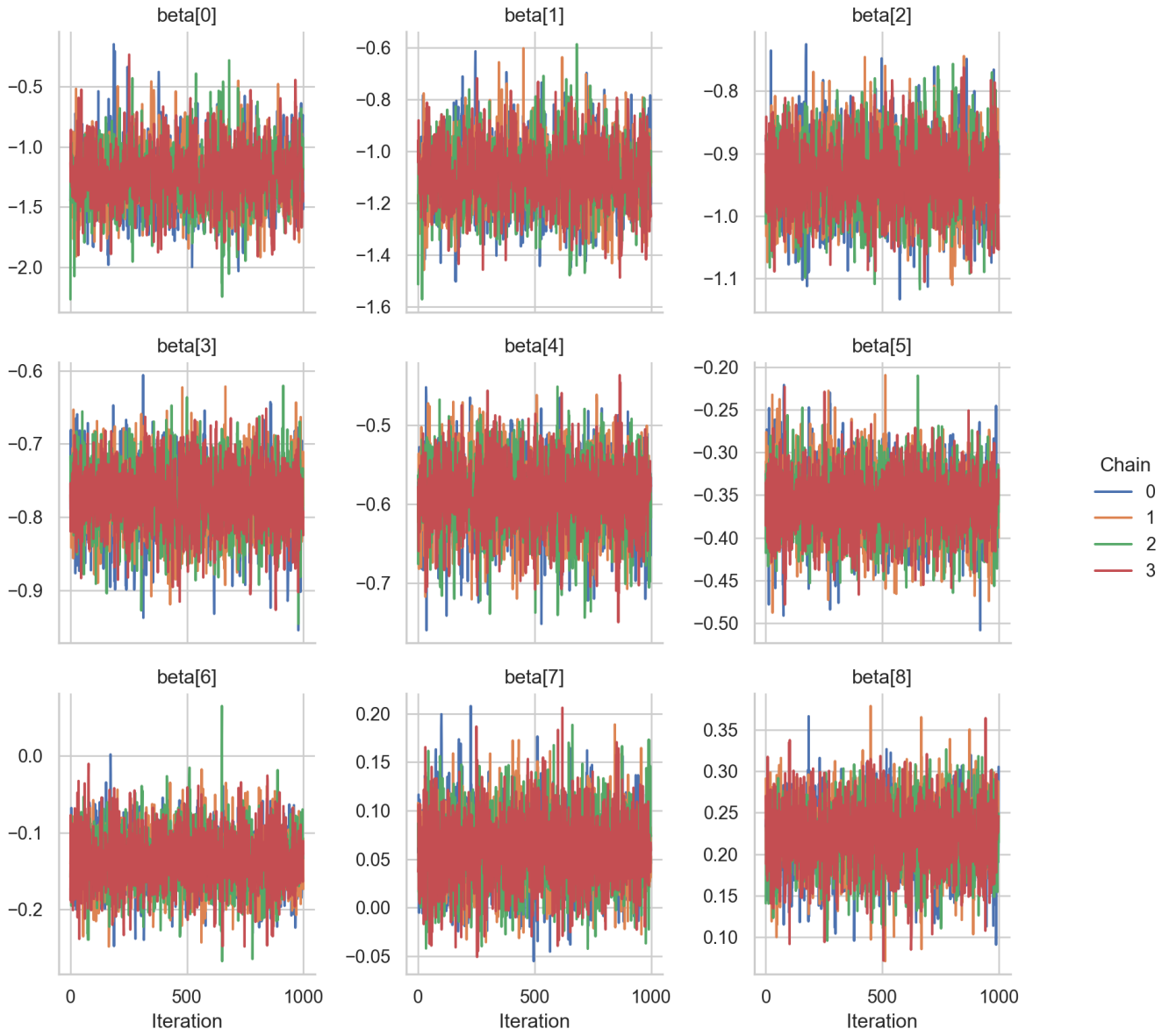
```
Parameter summary:

                              kernel      mean  ...     ess_tail      rhat
parameter            index                      ...
beta                 (0,)   kernel_00 -1.245570  ...  2125.050024  1.002264
                     (1,)   kernel_00 -1.091264  ...  1961.726560  1.002700
                     (2,)   kernel_00 -0.935815  ...  2538.677121  1.000149
                     (3,)   kernel_00 -0.775145  ...  2760.365406  1.000687
                     (4,)   kernel_00 -0.589133  ...  2373.142632  1.001223
                     (5,)   kernel_00 -0.358970  ...  2704.631310  1.000421
                     (6,)   kernel_00 -0.134668  ...  2798.241048  0.999893
                     (7,)   kernel_00  0.060015  ...  2909.504345  1.000759
                     (8,)   kernel_00  0.221116  ...  2877.186410  1.001866
                     (9,)   kernel_00  0.432459  ...  2417.788242  1.000353
                     (10,)  kernel_00  0.690132  ...  2994.071252  1.000312
                     (11,)  kernel_00  0.899978  ...  2167.029598  1.000965
                     (12,)  kernel_00  1.051785  ...  2769.289106  1.000695
                     (13,)  kernel_00  1.209876  ...  2506.205650  1.000923
                     (14,)  kernel_00  1.399641  ...  2367.204132  1.001162
                     (15,)  kernel_00  1.616565  ...  2248.231944  1.000571
                     (16,)  kernel_00  1.883243  ...  2033.742558  1.000361
                     (17,)  kernel_00  2.215253  ...  2164.572753  1.000297
                     (18,)  kernel_00  2.577903  ...  1602.538925  1.002657
                     (19,)  kernel_00  2.941443  ...  2019.561775  1.002814
sigma_sq_transformed ()     kernel_01 -0.419137  ...  1658.046182  1.001765
tau_sq_transformed   ()     kernel_00 -4.656025  ...   855.309815  1.008674

[22 rows x 10 columns]

Error summary:

                                                      count relative
kernel    error_code error_msg            phase
kernel_00 1          divergent transition warmup       92    0.023
                                          posterior     4    0.001
kernel_01 1          divergent transition warmup       49  0.01225
                                          posterior     0      0.0
```

Subtask e): Trace plots

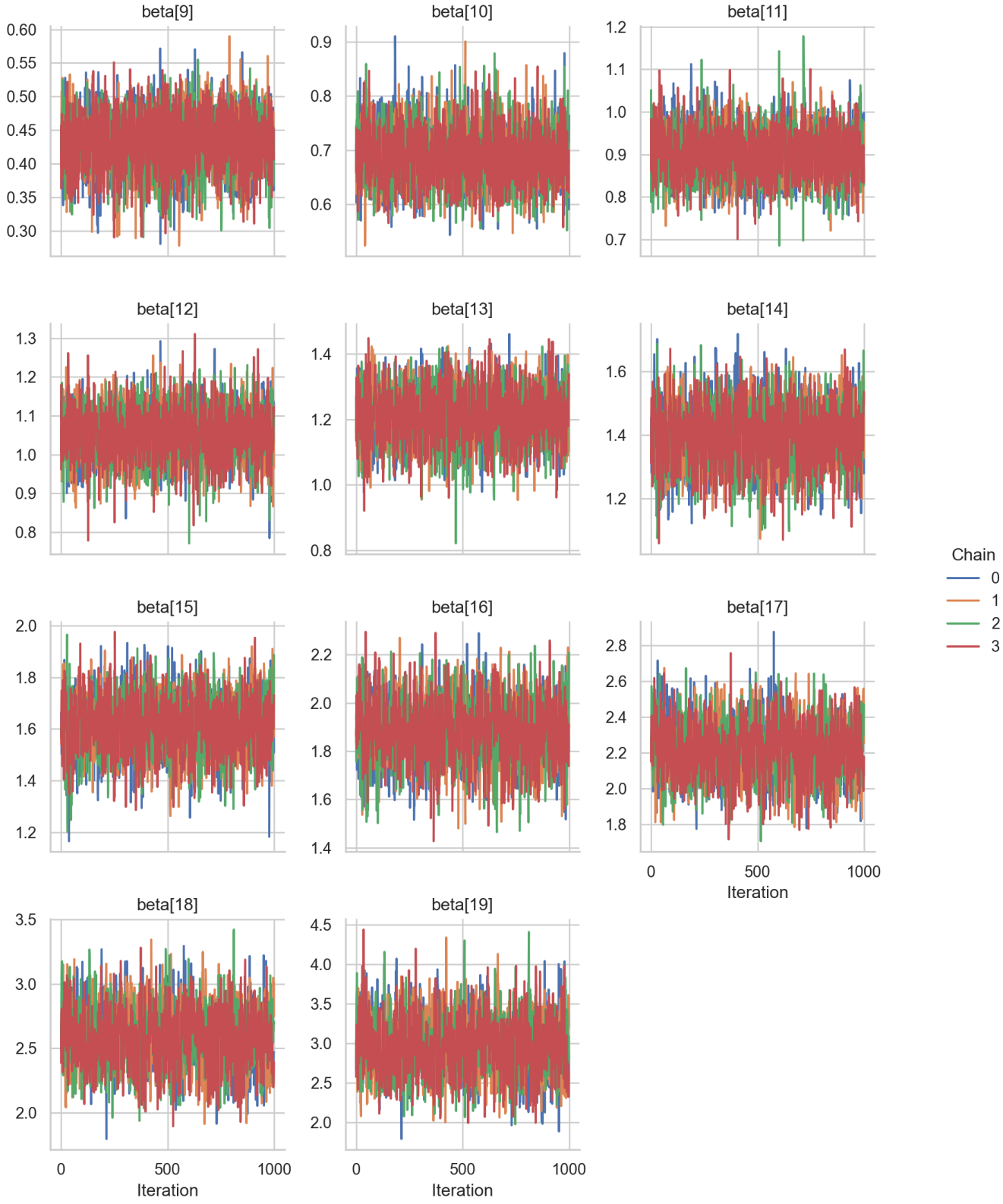
The trace plots look quite encouraging.

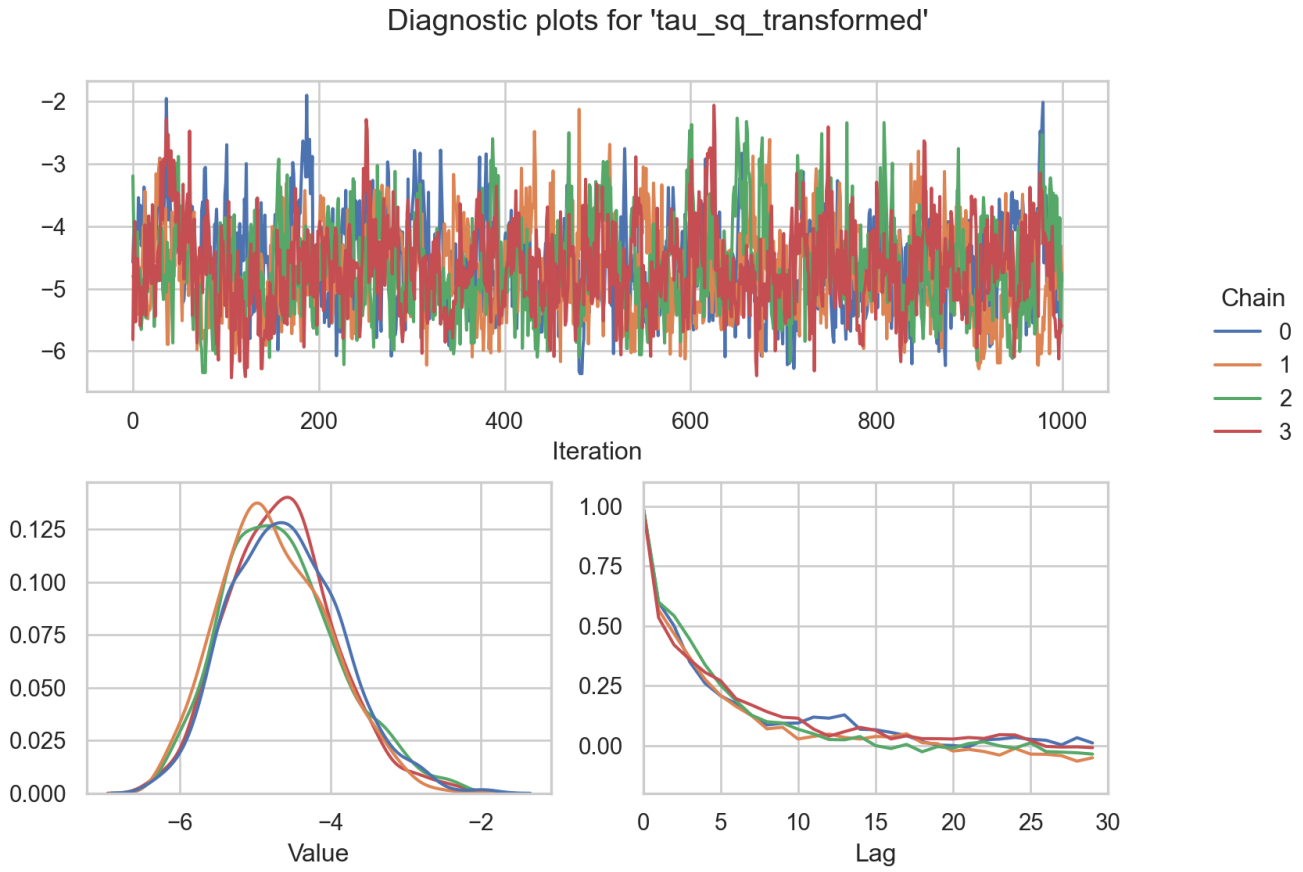
```{python}
gs.plot_trace(results, "beta", range(0, 9))
gs.plot_trace(results, "beta", range(9, 20))
gs.plot_param(results, "tau_sq_transformed")
gs.plot_param(results, "sigma_sq_transformed")
```

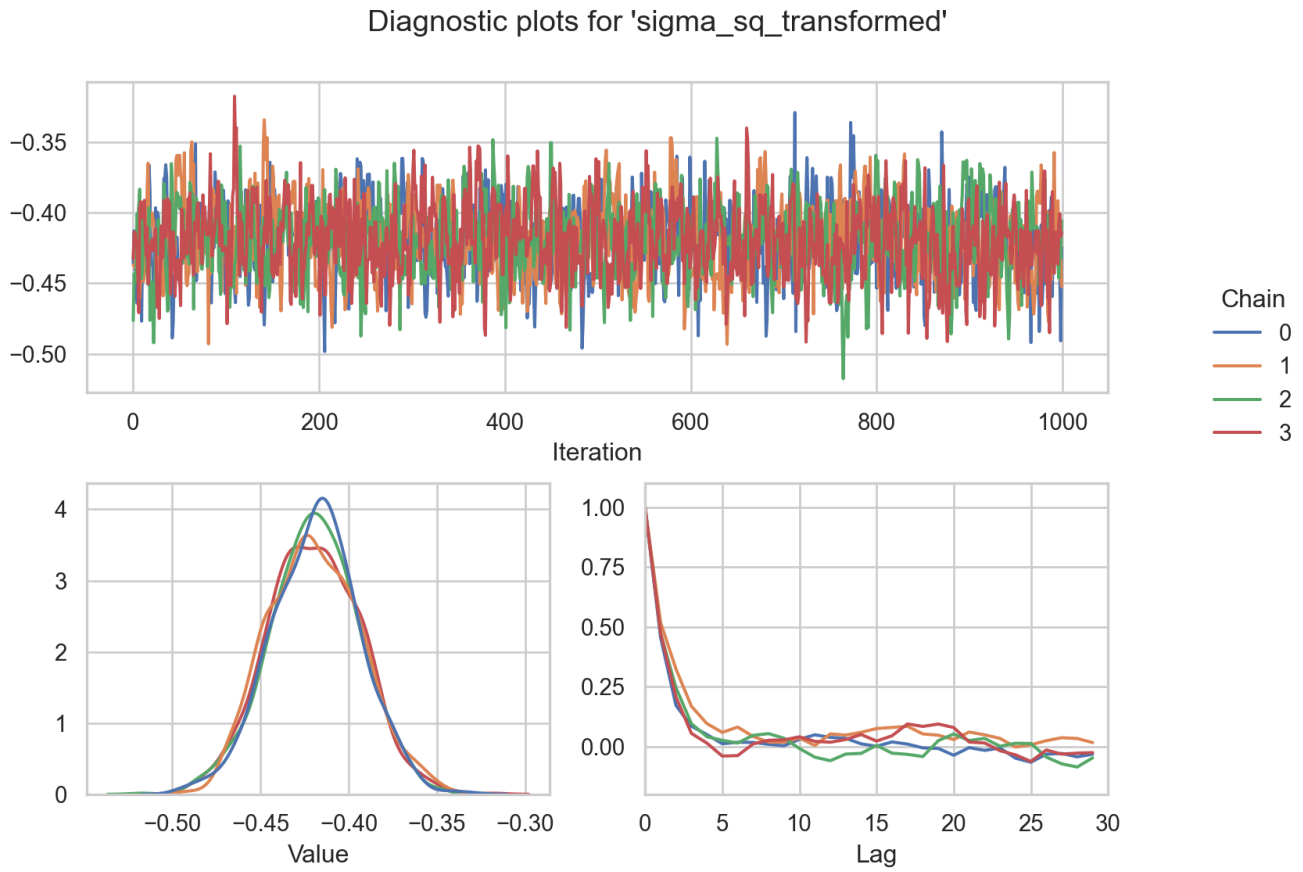
```
<seaborn.axisgrid.FacetGrid object at 0x2cb5a7e80>
```




```
<seaborn.axisgrid.FacetGrid object at 0x2e3dfc340>
```


beta[9]
beta[10]
beta[11]
beta[12]
beta[13]
beta[14]
beta[15]
beta[16]
beta[17]
beta[18]
beta[19]
Iteration
Chain
0
1
2
3





Subtask f): Plotting the posterior mean function

The summary object gives us access to a number of useful summary statistics. We can see the names by printing out the keys of the `gs.Summary.quantities` dictionary.

```{python}
list(summary.quantities)
```

```
['mean', 'var', 'sd', 'quantile', 'hdi', 'rhat', 'ess_bulk', 'ess_tail', 'mcse_mean',
'mcse_sd']
```

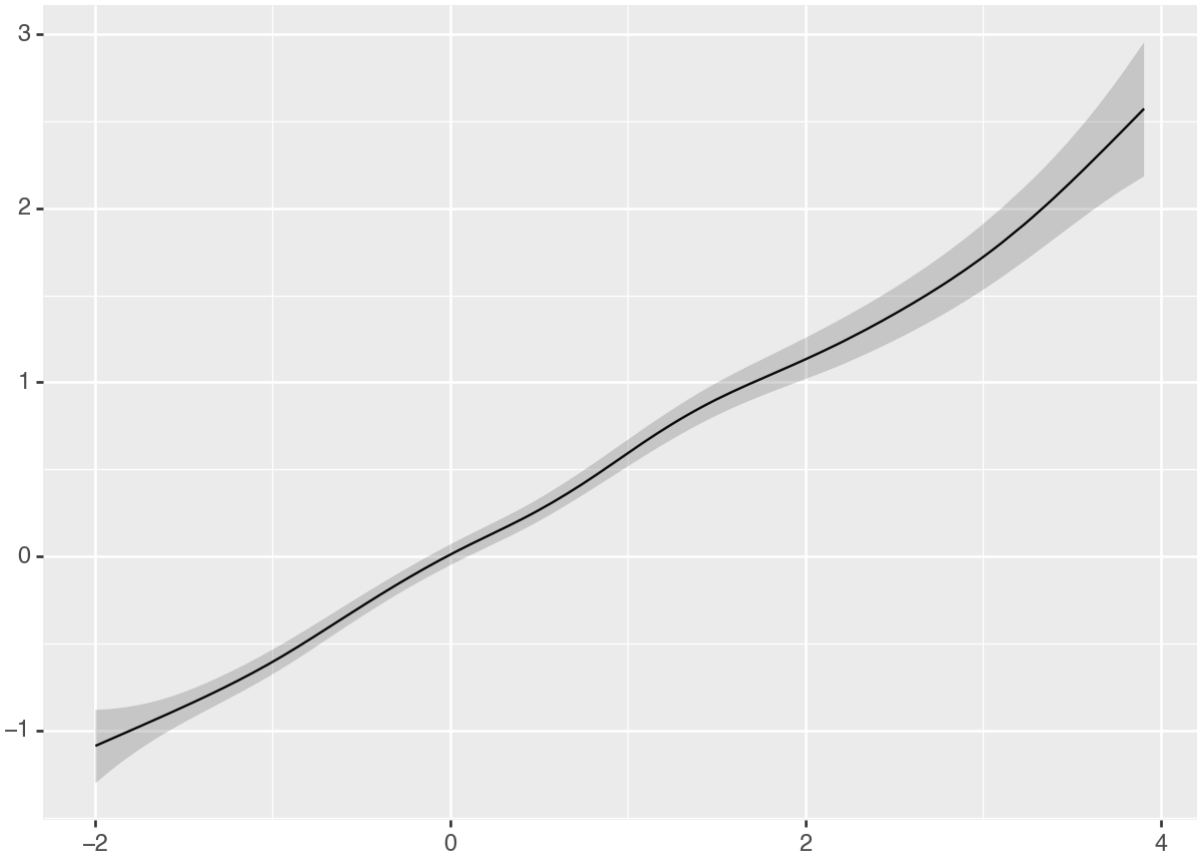
We can use these summary statistics to extract some values for plotting. In addition to the mean, we use the highest posterior density interval here to quantify uncertainty.

```{python}
beta_mean = summary.quantities["mean"]["beta"]
beta_hdi = summary.quantities["hdi"]["beta"]

s_mean = basis_matrix @ beta_mean
s_hdi_lo = basis_matrix @ beta_hdi[0,:]
s_hdi_hi = basis_matrix @ beta_hdi[1,:]
```

For plotting, we can use the Python library `plotnine`, which practically ports `ggplot2` from R to Python:

```{python}
from plotnine import ggplot, aes, geom_line, geom_ribbon

(
  ggplot()
  + aes(area, s_mean)
  + geom_line()
  + geom_ribbon(aes(ymin = s_hdi_lo, ymax = s_hdi_hi), alpha = 0.2)
)
```

```
<Figure Size: (640 x 480)>
```


3
2
1
0
−1
−2
0
2
4