Introduction to the Liesel Probabilistic Programming Framework

Johannes Brachem[1]

[1]University of Göttingen, Germany

Options for the practical exercises

{bamlss}

- **R package** for Baysian Additive Models for Location, Scale, and Shape
- Familiar modeling syntax for R users
- bamlss.org

liesel

- **Python library** for general graph-based Bayesian model building and MCMC inference
- Provides a toolbox for highly customized MCMC algorithms
- liesel-project.org
- Relatively recent development.

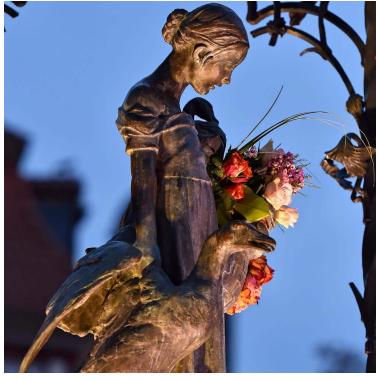
What's the difference?

`{bamlss}`

`liesel`

Images: ChatGPT

`liesel` is named after the "Gänseliesel"



goettingen-tourismus.de

liesel project team

Hannes Riebl
City of Lübeck, Germany

Paul Wiemann
Ohio State University

Johannes Brachem
University of Göttingen, Germany

Gianmarco Callegher
University of Göttingen, Germany

liesel overview

- liesel.model is for model-building
- liesel.goose is for MCMC inference

- Liesel relies heavily on
  - jax (docs) for **automatic differentiation** and **just-in-time compilation**
  - tensorflow_probability for jax-compatible probability **distributions** (docs)

What we pay

- Relatively verbose model setup
- Steep learning curve
- No easy inherent prediction solution, like R's `predict()` method.

Liesel tradeoffs

What we gain

- `liesel.model` **(Model building)**
  - You code a model close to how you might think about it.
  - Existing liesel models can be easily changed.
  - Built to be hacked.
  - Nice debugging through dynamic inspection of objects.
- `liesel.goose` **(MCMC inference)**
  - MCMC algorithm can be flexibly composed, inlcuding blocked sampling.
  - Just-in-time compilation provides good speed.
  - Automatic differentiation saves time.
  - Not locked into Liesel models

You might want to focus on the {bamlss} practicals, if ...

- ... you want to *apply* flexible distributional regression models.
- ... you are familiar with R, but not with Python
- ... you are familiar with R modeling syntax and have limited time or desire to get used to a different syntax.

You might want to focus on the `liesel` practicals, if ...

- ... you want to *develop* out-of-the-box Bayesian models.
- ... you have at least a basic familiarity with Python and `numpy`.
- ... you are curious to invest some time into learning a new approach to modeling.
- ... you are excited about a graph-based modeling approach.
- ... you want to quickly explore different MCMC algorithm configurations.

i We are happy to help

If you think that Liesel is the right tool for your problem, please feel free to approach us for assistance!

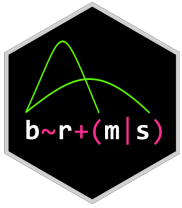
Other options (not covered here) may fit for your usecase

PyMC

- Probabilistic programming framework in Python
- Relatively similar to Liesel in scope
- Different approach to modeling
- PyMC models can be sampled with `liesel.goose`

STAN

- Great probabilistic programming language!
- Strong focus on the NUTS sampler; no Gibbs or blocked sampling
- Models are compiled, cannot be easily examined dynamically



{brms}

- Great R library that builds on Stan
- Focused on multilevel models, but can also do some distributional regression

`liesel.model` is for model-building

`liesel.model` is for model-building


```
import liesel.model as lsl
```

- `lsl.Var` is the central object.
  - `lsl.Var.new_obs()`: Observed data, contributes to likelihood
  - `lsl.Var.new_param()`: Model parameter, contributes to prior
  - `lsl.Var.new_calc()`: Wraps a calculation
  - `lsl.Var.new_value()`: A constant value
  - `lsl.Var.transform()`: Transform a variable with a bijector

- `lsl.Dist` represents distributions.
  - Wraps Tensorflow Probability distributions

- `lsl.Model`
  - Represents the relationships between variables.
  - Computes log-posterior, log-likelihood and log-prior.

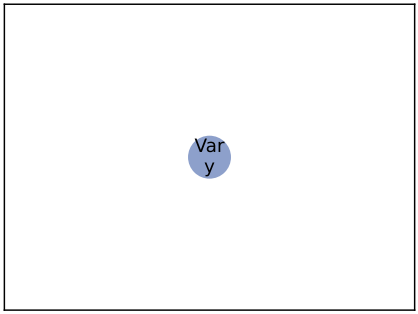
Common imports when working with Liesel


```{python}
import liesel.model as lsl # Liesel model building module
import liesel.goose as gs  # Liesel MCMC module

# Jax: Just-in-time compilation and automatic differentiation
import jax.numpy as jnp    # Behaves like numpy
import jax

# Probability distributions and bijectors
import tensorflow_probability.substrates.jax.distributions as tfd
import tensorflow_probability.substrates.jax.bijectors as tfb
```

lsl.Var.new_obs(value, distribution, name)


```
y = lsl.Var.new_obs(
    value=jnp.array([0.0, -0.5, 1.0]),
    name="y"
)

y.plot_vars()
```

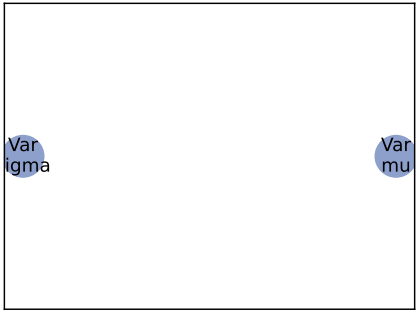
$$y = [0, -0.5, 1]^\mathsf{T}$$



lsl.Var.new_param(value, distribution, name)


```
mu = lsl.Var.new_param(
    value=0.0,
    name="mu"
)

sigma = lsl.Var.new_param(
    value=1.0,
    name="sigma"
)

model = lsl.Model([mu, sigma])
model.plot_vars()
```

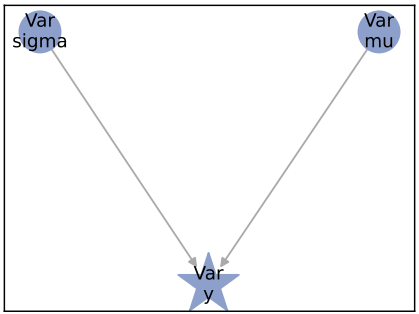
$\mu \sim \text{const}, \qquad \sigma \sim \text{const}$



lsl.Var.new_obs() with distribution


```
dist = lsl.Dist(
    tfd.Normal, loc=mu, scale=sigma
)

y = lsl.Var.new_obs(
    value=jnp.array([0.0, -0.5, 1.0]),
    distribution=dist,
    name="y"
)

y.plot_vars()
```

$$y \sim \mathcal{N}(\mu, \sigma^2), \quad \mu \sim \text{const}, \quad \sigma \sim \text{const}$$



lsl.Dist(distribution, *inputs, **kwinputs)


```
dist = lsl.Dist(
    tfd.Normal, # A tensorflow distribution class
    loc=...,    # Argument names *of the tensorflow class*
    scale=...   # Argument names *of the tensorflow class*
)
```

You can define your own tensorflow distributions!

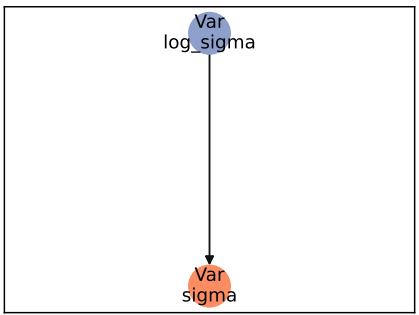
Details on how to do that can be found in the tensorflow probability documentation.

lsl.Var.new_calc(function, *inputs, **kwinputs, name)


```
log_sigma = lsl.Var.new_param(
    value=0.0, name="log_sigma"
)

sigma = lsl.Var.new_calc(
    jnp.exp, log_sigma, name="sigma"
)

sigma.plot_vars()
```

A deterministically computed variable is called a *weak* variable in Liesel models, while variables that introduce new information are called *strong*.

$$\ln(\sigma) \sim \text{const}, \qquad \sigma = \exp(\ln(\sigma))$$



You can use your own functions!

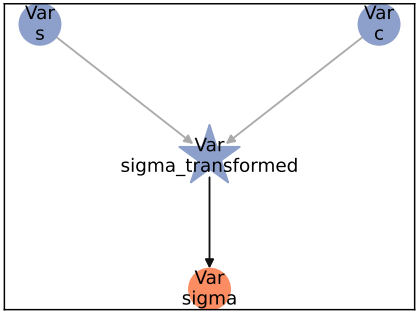
```
def my_function(arg1, arg2):
    return (arg1 + arg2) * jnp.exp(arg1)

lsl.Var.new_calc(
    function=my_function,  # Your function object
    arg1=...,              # Argument names of your function
    arg2=...               # Argument names of your function
)
```

Requirement: The function must not have side-effects. See the section on *pure functions* in the jax documentation.

lsl.Var.transform(bijector)


```
c = lsl.Var.new_param(0.5, name="c")
s = lsl.Var.new_param(1.0, name="s")

prior = lsl.Dist(
    tfd.Weibull, concentration=c, scale=s
)


sigma = lsl.Var.new_param(
    1.0, prior, name="sigma"
)

sigma_transformed = sigma.transform(
    tfb.Exp() # a tensorflow bijector
)

sigma.plot_vars()
```

$$\ln(\sigma) \sim \mathsf{ExpWeibull}(c, s), \qquad \sigma = \exp(\ln(\sigma))$$



lsl.Var.transform()


```
sigma = lsl.Var.new_param(
    1.0, prior, name="sigma"
)

sigma_transformed = sigma.transform(
    tfb.Exp() # a tensorflow bijector
              # the inverse link function
)

sigma.plot_vars()
```

Why transform? Some inference algorithms work better with parameters defined on the whole real line.

Before, the prior is defined on σ directly:

$$\sigma \sim \text{Weibull}(c, s)$$

After, the prior is defined on the logarithm of `scale`:

$$\sigma = \exp(\ln(\sigma))$$
$$\ln(\sigma) \sim \text{ExpWeibull}(c, s)$$

`lsl.Var.transform()` automatically applies the change of variables theorem.

lsl.Model([*variables])


```
prior = lsl.Dist(
    tfd.Normal, loc=0.0, scale=10.0
)
mu = lsl.Var.new_param(
    0.0, prior, name="mu"
)

ydist = lsl.Dist(tfd.Normal, loc=mu, scale=1.0)
y = lsl.Var.new_obs(
    value=jnp.array([0.0, -0.5, 1.0]),
    distribution=ydist,
    name="y"
)

model = lsl.Model([y])
```

```
# unnormalized log posterior
print(model.log_prob)
```

```
-6.603339
```

```
print(model.log_lik)
```

```
-3.3818154
```

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

```
-3.3818154
```

```
print(model.log_prior)
```

```
-3.2215238
```

```
print(mu.log_prob)
```

```
-3.2215238
```

lsl.Model([*variables])

Once initialized, the model updates automatically when values change.

```
mu.value = 1.0
```

```
# unnormalized log posterior
print(model.log_prob)
```

```
-7.60834
```

```
print(model.log_lik)
```

```
-4.381816
```

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

```
-4.381816
```

```
print(model.log_prior)
```

```
-3.2265239
```

```
print(mu.log_prob)
```

```
-3.2265239
```

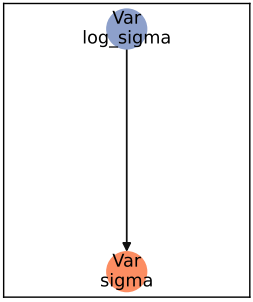
Square-bracket indexing: Positional value inputs


```
log_sigma = lsl.Var.new_param(0.0, name="log_sigma")
sigma = lsl.Var.new_calc(jnp.exp, log_sigma, name="sigma")
```

Access positional arguments by their indices:

```
sigma.value_node[0]
```

```
Var(name="log_sigma")
```



Square-bracket indexing: Positional value inputs


```
log_sigma = lsl.Var.new_param(0.0, name="log_sigma")
sigma = lsl.Var.new_calc(jnp.exp, log_sigma, name="sigma")
```

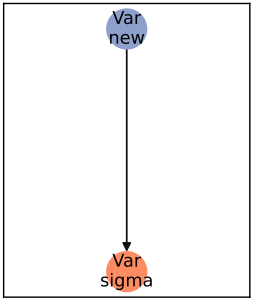
Access positional arguments by their indices:

```
sigma.value_node[0]
```

```
Var(name="log_sigma")
```

Swap out positional arguments by their indices:

```
sigma.value_node[0] = lsl.Var.new_param(2.0, name="new")
```



Square-bracket indexing: Keyword value inputs

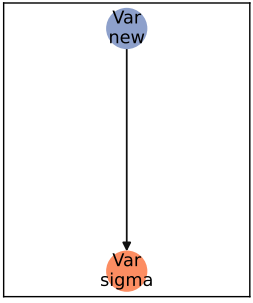
```
def fn(x):
    return x + 3.0

lsigma = lsl.Var.new_param(0.0, name="latent_sigma")
sigma = lsl.Var.new_calc(fn, x=lsigma, name="sigma")
```

Access keyword arguments by their names. Attention: *Argument name*, not *variable name*.

```
sigma.value_node["x"]
```

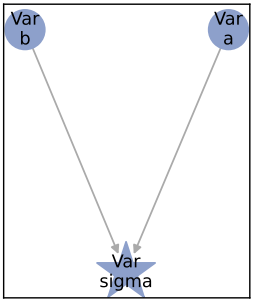
```
Var(name="latent_sigma")
```

Swap out keyword arguments by their names:



Square-bracket indexing: Keyword dist inputs


```
a = lsl.Var.new_param(0.01, name="a")
b = lsl.Var.new_param(0.01, name="b")
prior = lsl.Dist(
    tfd.InverseGamma, concentration=a, scale=b
)

sigma = lsl.Var.new_param(1.0, prior, name="sigma")
```



Access keyword arguments by their names. Attention: *Argument name*, not *variable name*.

```
sigma.dist_node["concentration"]
```

```
Var(name="a")
```

`liesel.goose` is for MCMC sampling

liesel.goose is for MCMC sampling


```
import liesel.goose as gs
```

- `gs.optim_flat`: Find posterior modes as good starting values.
- `gs.EngineBuilder`: Construct an MCMC algorithm
- `gs.Engine`: Represents a completed MCMC algorithm

- Goose comes with some MCMC Kernels
  - `gs.MHKernel`: Basic Metropolis-Hastings kernel.
  - `gs.RWKernel`: Metropolis-Hastings with random walk proposals.
  - `gs.HMCKernel`: Hamiltonian Monte Carlo
  - `gs.NUTSKernel`: No U-Turn Sampler (based on HMC)
  - `gs.IWLSKernel`: Iteratively Re-Weighted Least Squares
  - `gs.GibbsKernel`: Gibbs sampling
- `gs.Summary`: Inspect results

gs.optim_flat maximizes the log posterior


```
mu = lsl.Var.new_param(0.0, name="mu")
ydist = lsl.Dist(tfd.Normal, loc=mu, scale=1.0)
y = lsl.Var.new_obs(jnp.array([1.0, 5.0, 3.0]), ydist, name="y")
model = lsl.Model([y])
```

```
result = gs.optim_flat(
    model_train=model, # pass a lsl.Model
    params=[mu.name],  # a list of variable names to optimize

    model_validation=model, # here, I just pass the same model
    stopper=gs.Stopper(max_iter=1000, atol=0.1, patience=10),
)
```

```
result.position
```

```
{'mu': Array(2.2249236, dtype=float32)}
```

gs.EngineBuilder sets up the MCMC algorithm


```
eb = gs.EngineBuilder(num_chains=4, seed=42)

eb.set_model(gs.LieselInterface(model))
eb.set_initial_values(model.state)

eb.add_kernel(gs.NUTSKernel([mu.name]))
eb.set_duration(warmup_duration=200, posterior_duration=100)

engine = eb.build()
```

`gs.Engine.sample_all_epochs` draws posterior samples

```
engine.sample_all_epochs()
```

Phase 1: Warmup

During warmup, each kernel's hyperparameters are adjusted to reach a (customizable) target acceptance probability.

Phase 2: Blocked Posterior Sampling

After warmup, the hyperparameters are kept fixed, and the kernels take turns in sampling from their parameter's full conditional posterior distribution.

Take the following simple model:

$$y \sim \mathcal{N}(\mu, \sigma^2)$$

Goose's blocked sampling will operate like this:

For $t = 1, \ldots, T$:

1. Draw $\mu_{[t]}$ from $p(\mu|y, \sigma^2_{[t-1]})$ using the chosen kernel.
2. Draw $\sigma^2_{[t]}$ from $p(\sigma^2|y, \mu_{[t]})$ using the chosen kernel.

You don't *have* to use blocks!

With blocks:

```
eb.add_kernel(gs.NUTSKernel(["mu"]))
eb.add_kernel(gs.NUTSKernel(["sigma"]))
```

Without blocks:

```
eb.add_kernel(gs.NUTSKernel(["mu", "sigma"]))
```

Structured Additive Terms in Liesel

Liesel does not re-implement `mgcv`!

Option 1: `{rliesel}`

- Companion R package to Liesel
- Formula-style syntax to set up a full model
- Interface R and Python in **RStudio via Quarto / Rmarkdown**
- liesel-devs.github.io/rliesel/

Option 2: `rpy2`

- Set up terms in R with `mgcv` directly
- Pull them to Python with `rpy2`
- Interface R and Python in **Jupyter notebooks**

Using rpy2 in a Jupyter Notebook


```r
%%R
s_age <- mgcv::s(age, bs="ps", m=c(3,2), k=20) |>
    mgcv::smoothCon(data = zambia, absorb.cons = TRUE)

basis_age_r <- s_age[[1]]$X
pen_age_r <- s_age[[1]]$S[[1]]
```

```python
from rpy2 import robjects

basis_age_r = robjects.globalenv['basis_age_r']
pen_age_r = robjects.globalenv['pen_age_r']
```

MultivariateNormalDegenerate distribution in Liesel

- The potentially singular multivariate normal prior for structured additive terms is not implemented in Tensorflow probability directly.

```
from liesel.distributions import MultivariateNormalDegenerate as MVND

prior = lsl.Dist(
    MVND.from_penalty,
    loc=...,        # usually zero

    var=...,        # the variance parameter tau^2
    penalty=...,    # the penalty matrix

    # to avoid re-computations, we can provide:
    rank=...,       # rank of the penalty matrix
    log_pdet=...    # log pseudo-determinant of the penalty matrix
)
```

What's the difference?

`{bamlss}`

`liesel`

Images: ChatGPT

Thanks for your attention!

Check out the preprint on arXiv: **link**

Riebl, H., Wiemann, P. F. V., & Kneib, T. (2023). Liesel: A probabilistic programming framework for developing semi-parametric regression models and custom Bayesian inference algorithms. http://arxiv.org/abs/2209.10975

Check out the Python package on GitHub

github.com/liesel-devs/liesel