TUTORIAL II

Liesel Devs

GOOSE

WHAT IS GOOSE?

- Goose is the MCMC Library in Liesel
- It provides building blocks to create custom-MCMC algorithms
- Goose works with different model implementations
  - user-defined models (this exercise)
  - Liesel models (later today)
  - PyMC models
  - possibly more
- It provides popular samplers like NUTS
  - that take advantage of auto-differentiation
  - and have auto-tuning implemented
- However, Goose is extensible
  - users can easily add an own sampler implementation

BENEFITS OF USING GOOSE?

- Goose lets users focus on building the MCMC algorithm
  - that is, deciding how a parameter should be sampled

```
builder.add_kernel(gs.NUTS(['mu']))
builder.add_kernel(gs.HMC(['sigma_log']))
```

vs

```
builder.add_kernel(gs.NUTS(['mu', 'sigma_log']))
```

- Goose liberates the user from the accounting:
  - keeping track of the MCMC samples
  - managing the pseudo-random number generator (PRNG) state
  - calling the auto-tuning functions
  - keeping track of error / warnings that happen during sampling
  - providing summaries statisitcs of the samples
- Thus, users can focus on designing the MCMC algorithm

HOW DOES GOOSE WORK

Consider the parameter vector $\boldsymbol{\theta}$ separated into multiple blocks $\boldsymbol{\theta} = (\boldsymbol{\theta}_1^T, \ldots, \boldsymbol{\theta}_K^T)^T$. The MCMC algorithm generates the samples $\boldsymbol{\theta}^{(i)} = (\boldsymbol{\theta}_1^{(i)^T}, \ldots, \boldsymbol{\theta}_K^{(i)^T})^T, i = 1, \ldots, M$.

- When building the MCMC algorithm, the user assigns a kernel $\mathcal{K}_k$ to each subvector
- During sampling, Goose calls the kernels in sequence: $\mathcal{K}_1(\cdot), \mathcal{K}_2(\cdot), \ldots, \mathcal{K}_K(\cdot), \mathcal{K}_1(\cdot), \ldots$

- The kernel $\mathcal{K}_k$ is responsible for the transitions of the k-th subvector of the parameter vector $\boldsymbol{\theta}$

$$\begin{pmatrix} \boldsymbol{\theta}_1^{(i+1)} \\ \vdots \\ \boldsymbol{\theta}_{k-1}^{(i+1)} \\ \boldsymbol{\theta}_k^{(i)} \\ \boldsymbol{\theta}_{k+1}^{(i)} \\ \vdots \\ \boldsymbol{\theta}_K^{(i)} \end{pmatrix} \xrightarrow{\mathcal{K}_k(\cdot)} \begin{pmatrix} \boldsymbol{\theta}_1^{(i+1)} \\ \vdots \\ \boldsymbol{\theta}_{k-1}^{(i+1)} \\ \boldsymbol{\theta}_k^{(i+1)} \\ \boldsymbol{\theta}_{k+1}^{(i)} \\ \vdots \\ \boldsymbol{\theta}_K^{(i)} \end{pmatrix}$$

- After each complete transition, Goose stores the vector in the MCMC chain

HOW DOES GOOSE WORK

- In Goose, the sampling iterations are grouped in epochs
- Each epoch has a defined length and can be part of the warmup sequence or the posterior sampling
- After an warmup epoch ends, Goose can call the kernels' auto-tuning functions

THE ENGINE

- The `Engine` is the core class in Goose
- It handles calling the kernels and the auto tuning functions
- It saves the samples
- Can sample multiple chains in parallel

- The engine has a lot of options that can be configured
- Using the `EngineBuilder` simplifies the process by providing a step-by-step interface

EXTENDED EXAMPLE

One parameter Poisson model

EXTENDED EXAMPLE

We consider the same toy example as eariler.

- 5 iid observations of $Y \sim Poi(\lambda), y_1 = 3, y_2 = 4, y_3 = 4, y_4 = 6, y_5 = 3.$
- We place a prior on λ, namely $\lambda \sim Gamma(1, 1)$.
- The goal is to estimate the posterior distribution of λ using MCMC.

In this example, we want to use the No U-Turn Sampler (NUTS) which requires that the parameter has support defined on the real line. Thus, we use the same log transformation as before. However, this time and similar to the solution on the first exercise sheet, we define a class that represents the model parameters.

IMPORT THE REQUIRED MODULES

```python
from typing import NamedTuple, TypeAlias
from functools import partial

import jax
import jax.numpy as jnp
import tensorflow_probability.substrates.jax as tfp

import liesel.goose as gs

Array: TypeAlias = jax.Array
```

DEFINE THE MODEL

```python
class Params(NamedTuple):
    lambda_log: Array

def log_prior(params: Params) -> Array:
    lambda_ = jnp.exp(params.lambda_log)
    dist = tfp.distributions.Gamma(1.0, 1.0)
    return dist.log_prob(lambda_) + params.lambda_log

def log_likelihood(params: Params, obs: Array) -> Array:
    dist = tfp.distributions.Poisson(jnp.exp(params.lambda_log))
    lps = dist.log_prob(obs)
    return jnp.sum(lps)

def log_uposterior(params: Params, obs: Array) -> Array:
    return log_prior(params) + log_likelihood(params, obs)
```

Create function that binds the data

```python
ys = jnp.array([3, 4, 4, 6, 3])
def log_upost_data(params: Params) -> Array:
    return log_prior(params) + log_likelihood(params, ys)
```

BUILD AND RUN THE SAMPLER

STEPS:

1. Create an EngineBuilder
2. Configure the EngineBuilder
    - Connect the model
    - Provide an initial value for the parameter
    - Configure the sequence of epochs
    - Add the kernels
3. Build the Engine
4. Run the Sampler
5. Inspect the results

CREATE THE BUILDER

- Here, we create the builder, provide a seed and state that we want to sample 4 MCMC chains.

```
builder = gs.EngineBuilder(seed=3, num_chains=4)
```

- Next, we connect the builder to the unnormalized log-posterior using the NamedTupleInterface.

```
builder.set_model(gs.NamedTupleInterface(log_upost_data))
```

- Then, we provide a starting value for the parameter

```
init_params = Params(lambda_log=0.0)
builder.set_initial_values(init_params)
```

ADD A KERNEL

- Next, we configure the kernel that should be used to sample `lambda_log`.

```
builder.add_kernel(gs.NUTSKernel(['lambda_log']))
```

CONFIGURE THE EPOCHS

- We need to tell Goose how many MCMC samples the `Engine` should generatre and how the warmup sequence should look like.
- One can configure this with a lot of detail.
- It is often sufficient to just set the length of the warmup and posterior phases and let Goose use the default warmup scheme.

```
1  builder.set_duration(warmup_duration=1000, posterior_duration=1000)
```

BUILD THE ENGINE AND SAMPLE

- Finally, the engine, which ties everything together, can be built.
- Afterwards, we can let it run the four MCMC chains.
- This executes all the warmup epochs and the 1000 iterations in the posterior epoch.

```
1 engine = builder.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
```

```
1 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: 1, 3, 2, 3 / 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, 2, 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: 1, 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, 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: 1, 1, 0, 1 / 200 transitions
liesel.goose.engine - INFO - Finished epoch
```

INSPECT THE RESULTS

We can inspect the result summary.

```
1 results = engine.get_results()
2 gs.Summary(results)
```

Parameter summary:

		kernel	mean	sd	q_0.05	q_0.5	q_0.95	sample_size	e
parameter	**index**								
lambda_log	()	kernel_00	1.232773	0.219971	0.859837	1.238524	1.578157	4000	1

Error summary:

				count	relative
kernel	**error_code**	**error_msg**	**phase**		
kernel_00	1	divergent transition	warmup	35	0.00875
			posterior	0	0.0

ADDING THE MISSING LAMBDA

For that, we can add provide an additional chain to the Summary class.

```
1 add_chain = {'lambda': jnp.exp(results.get_posterior_samples()['lambda_log'])}
2 gs.Summary(results, additional_chain=add_chain)
```

Parameter summary:

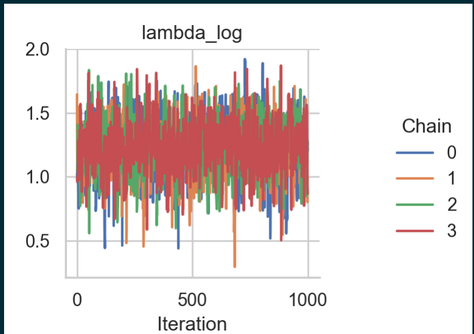
		kernel	mean	sd	q_0.05	q_0.5	q_0.95	sample_size	
parameter	index								
lambda_log	()	kernel_00	1.232773	0.219971	0.859837	1.238524	1.578157	4000	1
lambda	()	-	3.513436	0.764000	2.362774	3.450517	4.846015	4000	1

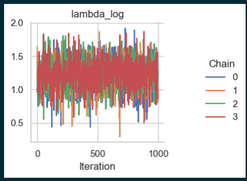
Error summary:

				count	relative
kernel	error_code	error_msg	phase		
kernel_00	1	divergent transition	warmup	35	0.00875
			posterior	0	0.0

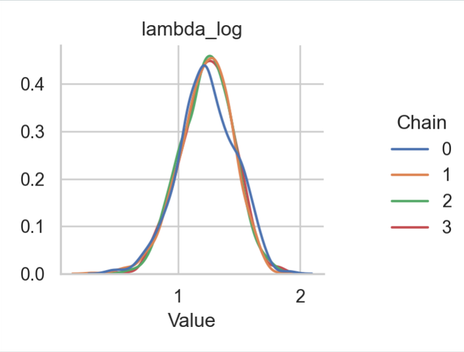
INSPECTING SOME PLOTS

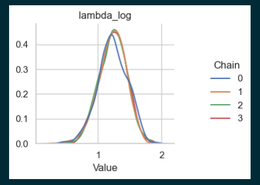
```
1  gs.plot_trace(results)
```






```
1  gs.plot_density(results)
```




lambda_log
0.4
0.3
0.2
0.1
0.0
1
2
Value
Chain
0
1
2
3

EXERCISE SHEET

Please start working on the exercise sheet