Sheet 00 - Solutions

First, we import the python libraries which we need to solve the exercise.

```
import jax
import jax.numpy as jnp
from jax.flatten_util import ravel_pytree
import tensorflow_probability.substrates.jax as tfp

import matplotlib.pyplot as plt
from tqdm.notebook import tqdm

from typing import NamedTuple, TypeAlias

Array: TypeAlias = jax.Array
```

Then we generate some data. More precisely, we generate 10 observations from a Gaussian distribution with expectation 1.4 and variance 0.8^2.

```
mu = -1.4
sigma = 0.8
n = 10
obs = tfp.distributions.Normal(mu, 0.8**2).sample(n, seed=jax.random.PRNGKey(0))
```

Exercise 01

Define functions for the log prior, log likelihood and log unnormalized posterior assuming known variance. Since $\mu \in R$, we do not need to perform a parameter transformation.

```
def log_prior(params) -> Array:
    return tfp.distributions.Normal(0, 10).log_prob(params)

def log_likelihood(params, obs: Array) -> Array:
    dist = tfp.distributions.Normal(params, sigma**2)
    lps = dist.log_prob(obs)
    return jnp.sum(lps)

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

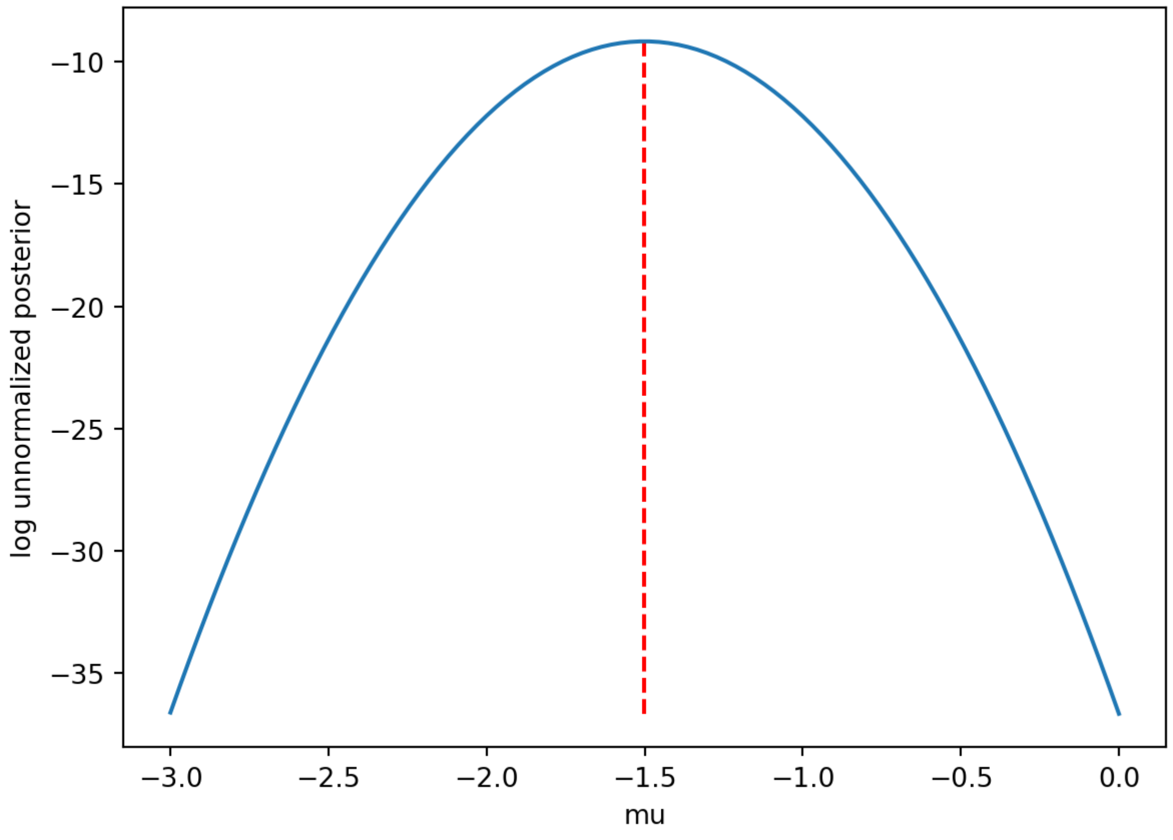
Next we use Newton’s method to find the posterior mode.

```
# we start with a random guess
params = jnp.array(0.0)

# we only need the gradient and hessian wrt to the parameters,
# the first argument of `log_uposterior_flat`
dlp = jax.grad(log_uposterior, argnums=0)
ddlp= jax.hessian(log_uposterior, argnums=0)
```

```
# if we want, we can jit the functions
jdlp = jax.jit(dlp)
jddlp = jax.jit(ddlp)

# we can now run the optimization
# on the flattend parameter
for i in (pb := tqdm(range(10))):
    params = params - jdlp(params, obs) / jddlp(params, obs)
    pb.set_description(f"mu={params:.2f}, log_upost={log_uposterior(mu, obs):.2f}")
```

```
mu=-1.50, log_upost=-9.29:                                  10/10 [00:00<00:00,

100%                                                        137.90it/s]
```

```
xs = jnp.linspace(-3, 0, 100)
lps = [log_uposterior(x, obs) for x in xs]
plt.vlines(params, min(lps), max(lps), color="red", linestyles="dashed")
plt.plot(xs, lps)
plt.xlabel("mu")
plt.ylabel("log unnormalized posterior")
plt.show()
```



Exercise 02

Here, we repeat the above. This time, the parameter class has the additional field `log_sigma`. We use `log_sigma` instead of `sigma`, since we need for Newton's algorithm that all parameters are defined on ‘R’. Since out mathematical model is defined in terms of sigma, we need to apply the change of variable theorem to sigmas prior.

When solving Exercise 02, we also opt to use a user-defined class to represent the vector. As you can see, it makes referring to the individual parameters less error-prone.

```
class Params2(NamedTuple):
    mu: float
    log_sigma: float

def log_prior2(params: Params2) -> Array:
    sigma = jnp.exp(params.log_sigma)
    lp_mu = tfp.distributions.Normal(0, 10).log_prob(params.mu,)
    lp_sigma =  tfp.distributions.HalfCauchy(0, 1).log_prob(sigma) + params.log_sigma
    return lp_mu + lp_sigma

def log_likelihood2(params: Params2, obs: Array) -> Array:
    sigma = jnp.exp(params.log_sigma)
    dist = tfp.distributions.Normal(params.mu, sigma)
    lps = dist.log_prob(obs)
    return jnp.sum(lps)

def log_uposterior2(params: Params2, obs: Array) -> Array:
    return log_prior2(params) + log_likelihood2(params, obs)
```

It is, however, not clear how a the gradient or Hessian are defined when using a function with a non-vector-valued argument. Luckily, `jax` provides a function that can use any pytree and convert it to a vector and back. This function is called `ravel_tree` and is found in the module `jax.flatten_tree`. We are going to use this functionality to define an additional unnormalized posterior function that works with an input vector and internally converts it back to our parameter type and calls `log_uposterior2`.

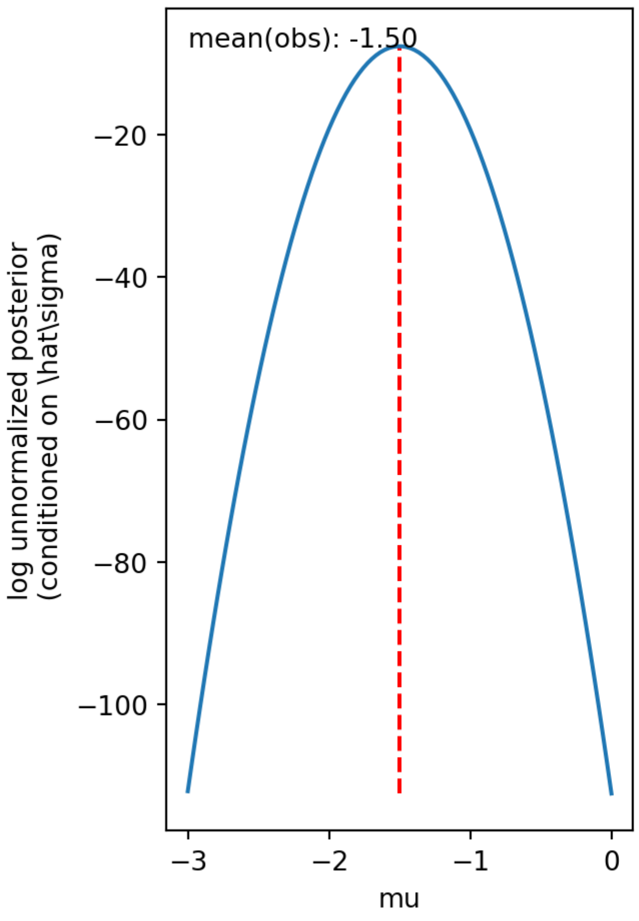
```
# We need an example of our parameter to be able to use ravel_pytree
params2 = Params2(mu=0.0, log_sigma=0.0)

# ravel_pytree flattens the parameters (which could have a complex and
# nested structure) into a single vector.
# The function returns the flattend parameter vector and the inverse function
# that takes a vector and returns an object of the parameter type.
flat_params2, unravel_fn2 = ravel_pytree(params2)

# now we define a function that takes a flat parameter vector and returns the
# log unnormalized posterior internally, this function unravels the flat
# parameter vector into the parameter object
def log_uposterior2_flat(flat_params: Array, obs: Array) -> Array:
    params: Params2 = unravel_fn2(flat_params)
    return log_uposterior2(params, obs)
```

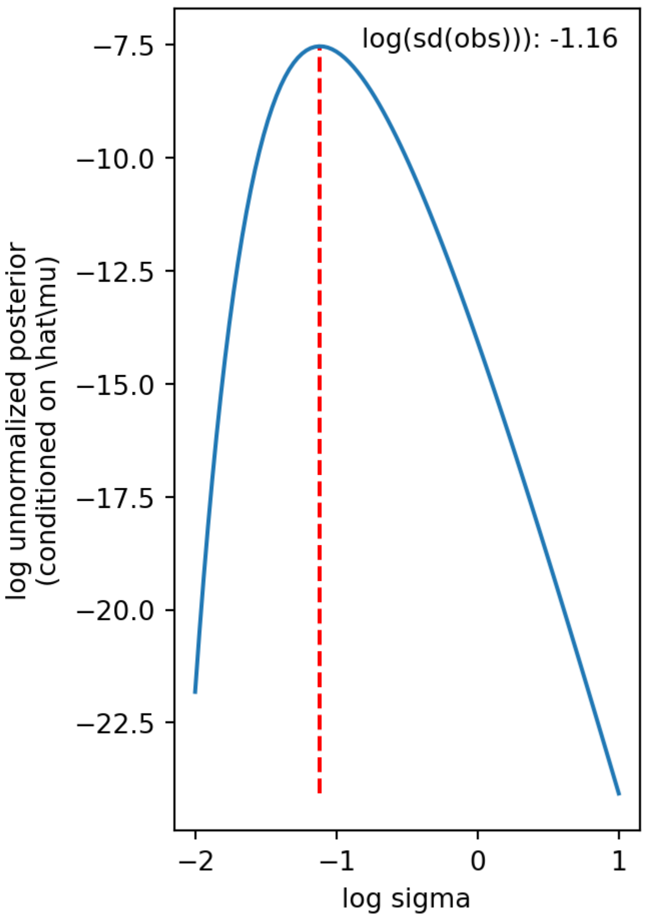
In this optimzation, we use gradient ascent but determine the step-size automatically using the Hessian (see website). Note that the optimization operates on the flattened parameter vector.

```
# we start with a random guess
params2 = Params2(mu=0.0, log_sigma=0.0)
# and flatten it
flat_params2, _ = ravel_pytree(params2)

# we only need the gradient wrt to the parameters, the first argument
dlp2 = jax.grad(log_uposterior2_flat, argnums=0)
ddlp2 = jax.hessian(log_uposterior2_flat, argnums=0)

# if we want, we can jit the functions
jdlp2 = jax.jit(dlp2)
jddlp2 = jax.jit(ddlp2)

for i in (pb := tqdm(range(100))):
    grad = jdlp2(flat_params2, obs)
    hess = jddlp2(flat_params2, obs)
    alpha = (grad ** 2).sum() / (grad.T @ hess @ grad)
    flat_params2 = flat_params2 - alpha * grad

    params2: Params2 = unravel_fn2(flat_params2)
    pb.set_description(
        f"mu={params2.mu:.2f}, "
        f"sigma={jnp.exp(params2.log_sigma):.2f}, "
        f"log_upost={log_uposterior2(params2, obs):.2f}"
    )
```

```
mu=-1.50, sigma=0.33,                                    100/100 [00:00<00:00,
log_upost=-7.53: 100%                                    246.45it/s]
```

```
fig, axs = plt.subplots(1, 2)
xs = jnp.linspace(-3, 0, 100)
lps = [log_uposterior2(Params2(mu=x, log_sigma=params2.log_sigma), obs) for x in xs]
axs[0].vlines(params2.mu, min(lps), max(lps), color="red", linestyles="dashed")
axs[0].plot(xs, lps)
axs[0].set_xlabel("mu")
axs[0].set_ylabel("log unnormalized posterior\n (conditioned on \\hat\\sigma)")
axs[0].text(-3, max(lps), "mean(obs): {:.2f}".format(jnp.mean(obs)), ha="left")


xs = jnp.linspace(-2, 1, 100)
lps = [log_uposterior2(Params2(mu=params2.mu, log_sigma=x), obs) for x in xs]
axs[1].vlines(params2.log_sigma, min(lps), max(lps), color="red", linestyles="dashed")
axs[1].plot(xs, lps)
axs[1].set_xlabel("log sigma")
axs[1].set_ylabel("log unnormalized posterior\n(conditioned on \\hat\\mu)")
axs[1].text(1.0, max(lps), f"log(sd(obs))): {jnp.log(jnp.std(obs)):.2f}", ha="right")
fig.tight_layout()
plt.show()
```


mean(obs): -1.50
log unnormalized posterior (conditioned on \hat\sigma)
mu
−20
−40
−60
−80
−100
−3
−2
−1
0


log(sd(obs))): -1.16
log unnormalized posterior (conditioned on \hat\mu)
log sigma
−7.5
−10.0
−12.5
−15.0
−17.5
−20.0
−22.5
−2
−1
0
1