Homework 7 - Part B

Note that there are two different notebooks for HW assignment 7. This is part A. There will be two different assignments in gradescope for each part. The deadlines are the same for both parts.

References

- Lectures 27-28 (inclusive).

Instructions

- Type your name and email in the "Student details" section below.
- Develop the code and generate the figures you need to solve the problems using this notebook.
- For the answers that require a mathematical proof or derivation you should type them using latex. If you have never written latex before and you find it exceedingly difficult, we will likely accept handwritten solutions.
- The total homework points are 100. Please note that the problems are not weighed equally.

Student details

- **First Name:** Hyejun
- **Last Name:** Jeong
- **Email:** jeong139@purdue.edu

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
import seaborn as sns
sns.set_context("paper")
sns.set_style("ticks")

import scipy
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
    """Download a file from a url.

    Arguments
    url            -- The url we want to download.
    local_filename -- The filemame to write on. If not
                      specified
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    if not os.path.exists(local_filename):
        urllib.request.urlretrieve(url, local_filename)
```

In [ ]:
```
# # Run this on Google colab
# !pip install pyro-ppl
```

In [ ]:
```
import pyro
import pyro.distributions as dist
from pyro.infer import MCMC, NUTS
import torch
```

Problem 1 - Bayesian Linear regression on steroids

The purpose of this problem is to demonstrate that we have learned enough to do very complicated things. In the first part, we will do Bayesian linear regression with radial basis functions (RBFs) in which we characterize the posterior of all parameters, including the length-scales of the RBFs. In the second part, we are going to build a model that has an input-varying noise. Such models are called heteroscedastic models.

We need to write some `pytorch` code to compute the design matrix. This is absolutely necessary so that `pyro` can differentiate through all expressions.

In [ ]:
```
class RadialBasisFunctions(torch.nn.Module):
    """Radial basis functions basis.

    Arguments:
    X   -  The centers of the radial basis functions.
    ell -  The assumed length scale.
    """
    def __init__(self, X, ell):
        super().__init__()
        self.X = X
        self.ell = ell
        self.num_basis = X.shape[0]
```

```
    def forward(self, x):
        distances = torch.cdist(x, self.X)
        return torch.exp(-.5 * distances ** 2 / self.ell ** 2)
```

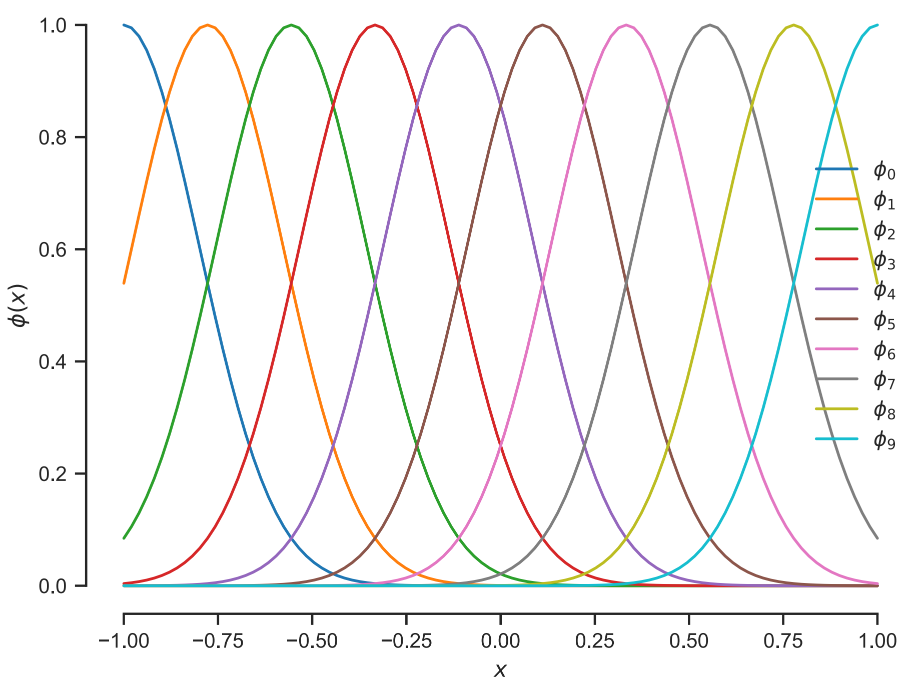
Here is how you can use them:

In [ ]:
```
# Make the basis
x_centers = torch.linspace(-1, 1, 10).unsqueeze(-1)
ell = 0.2
basis = RadialBasisFunctions(x_centers, ell)

# Some points (need to be N x 1)
x = torch.linspace(-1, 1, 100).unsqueeze(-1)

# Evaluate the basis
Phi = basis(x)

# Here is the shape of Phi
print(Phi.shape)
```

```
torch.Size([100, 10])
```

Here is how they look like:

In [ ]:
```
fig, ax = plt.subplots()
for i in range(Phi.shape[1]):
    ax.plot(x, Phi[:, i], label=f"$\phi_{i}$")
ax.set(xlabel="$x$", ylabel="$\phi(x)$")
ax.legend(loc="best", frameon=False)
sns.despine(trim=True);
```



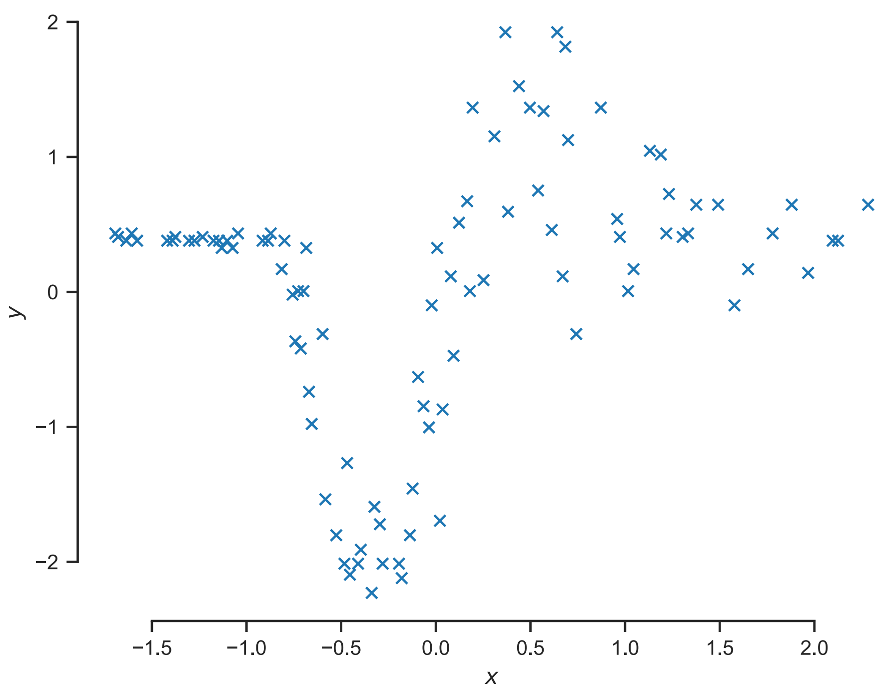
Part A - Hierarchical Bayesian linear regression with input-independent noise

We will analyze the motorcycle dataset. The data is loaded below.

In [ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/lecturebook/data/motor
download(url)
```

We will work with the scaled data:

```
In [ ]: from sklearn.preprocessing import StandardScaler

        data = np.loadtxt('motor.dat')
        scaler = StandardScaler()
        data = scaler.fit_transform(data)
        X = torch.tensor(data[:, 0], dtype=torch.float32).unsqueeze(-1)
        Y = torch.tensor(data[:, 1], dtype=torch.float32)

        fig, ax = plt.subplots()
        ax.plot(X, Y, 'x')
        ax.set(xlabel="$x$", ylabel="$y$")
        sns.despine(trim=True);
```



Part A.I

Your goal is to implement the model described below. We use the radial basis functions (`RadialBasisFunction`) with centers, x_i at $m = 50$ equidistant points between the minimum and maximum of the observed inputs:

$$\phi_i(x;\ell) = \exp\left(-\frac{(x-x_i)^2}{2\ell^2}\right),$$

for $i = 1, \dots, m$. We denote the vector of RBFs evaluated at x as $\boldsymbol{\phi}(x;\ell)$.

We are not going to pick the length-scales ℓ by hand. Instead, we will put a prior on it:

$$\ell \sim \text{Exponential}(1).$$

The corresponding weights have priors:

$$w_j|\alpha_i \sim N(0, \alpha_j^2),$$

and its α_j has a prior:

$$\alpha_j \sim \text{Exponential}(1),$$

for $j = 1, \dots, m$.

Denote our data as:

$$x_{1:n} = (x_1, \dots, x_n)^T, \text{ (inputs)},$$

and

$$y_{1:n} = (y_1, \dots, y_n)^T, \text{ (outputs)}.$$

The likelihood of the data is:

$$y_i | \mathbf{w}, \sigma \sim N(\mathbf{w}^T \boldsymbol{\phi}(x_i; \ell), \sigma^2),$$

for $i = 1, \dots, n$.

$$y_n | \ell, \mathbf{w}, \sigma \sim N(\mathbf{w}^T \boldsymbol{\phi}(x_n; \ell), \sigma^2).$$

Complete the `pyro` implementation of that model:

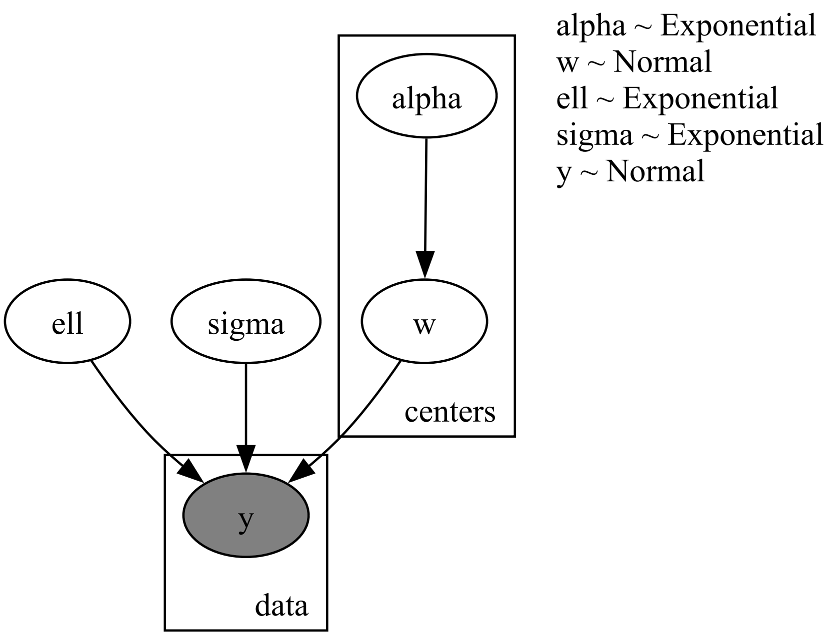
Answer:

In [ ]:
```
def model(X, y, num_centers=50):
    with pyro.plate("centers", num_centers):
        alpha = pyro.sample("alpha", dist.Exponential(1.0))
        # Notice below that dist.Normal needs the standard deviation - not the variance
        # We follow a different convention in the lecture notes
        w = pyro.sample("w", dist.Normal(0.0, alpha))
    ell = pyro.sample("ell", dist.Exponential(1.0)) # Complete the code assign to ell the correct 
    # Hint: Look at alpha.
    sigma = pyro.sample("sigma", dist.Exponential(1.0)) # Complete the code assign to sigma the co
    x_centers = torch.linspace(X.min(), X.max(), num_centers).unsqueeze(-1)
    Phi = RadialBasisFunctions(x_centers, ell)(X)
    with pyro.plate("data", X.shape[0]):
        pyro.sample("y", dist.Normal(Phi @ w, sigma), obs=y)
    # Notice that I'm making the model return all the variables that I have made.
    # This is not essential for characterizing the posterior, but it does reduce redundant code
    # when we are trying to get the posterior predictive.
    return locals()
```

The graph will help to understand the model:

In [ ]:
```
pyro.render_model(model, (X, Y), render_distributions=True)
```

Out[ ]:



Use `pyro.infer.autoguide.AutoDiagonalNormal` to make the guide:

In [ ]:
```
guide = pyro.infer.autoguide.AutoDiagonalNormal(model)
```

We will use variational inference. Here is the training code from the hands-on activity:

In [ ]:
```
def train(model, guide, data, num_iter=5_000):
    """Train a model with a guide.

    Arguments
    ---------
    model    -- The model to train.
    guide    -- The guide to train.
    data     -- The data to train the model with.
    num_iter -- The number of iterations to train.

    Returns
    -------
    elbos -- The ELBOs for each iteration.
    param_store -- The parameters of the model.
    """

    pyro.clear_param_store()

    optimizer = pyro.optim.Adam({"lr": 0.001})

    svi = pyro.infer.SVI(
        model,
        guide,
        optimizer,
        loss=pyro.infer.JitTrace_ELBO()
    )

    elbos = []
    for i in range(num_iter):
        loss = svi.step(*data)
        elbos.append(-loss)
        if i % 1_000 == 0:
            print(f"Iteration: {i} Loss: {loss}")
```

```
    return elbos, pyro.get_param_store()
```

Part A.II

Train the model for 20,000 iterations. Call the `train()` function we defined above to do it. Make sure you store the returned elbo values because you will need them later.

Answer:

In [ ]:
```
elbos, params = train(model, guide, (X, Y), num_iter=20000)
```

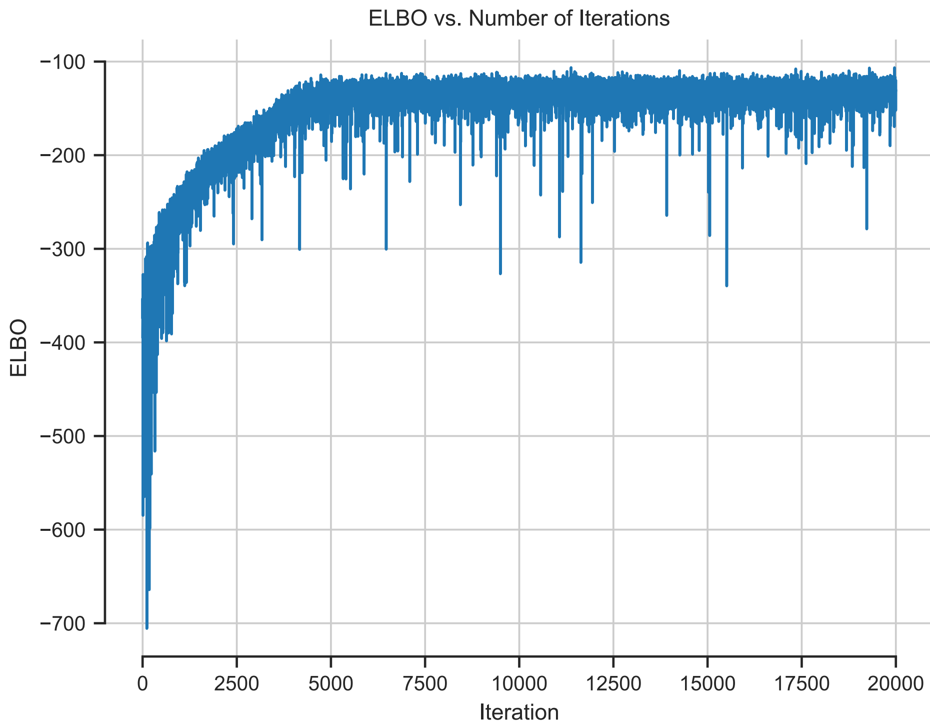
```
Iteration: 0 Loss: 374.2281494140625
Iteration: 1000 Loss: 259.06463623046875
Iteration: 2000 Loss: 203.27891540527344
Iteration: 3000 Loss: 163.80564880371094
Iteration: 4000 Loss: 146.6731414794922
Iteration: 5000 Loss: 133.25064086914062
Iteration: 6000 Loss: 134.96878051757812
Iteration: 7000 Loss: 135.12750244140625
Iteration: 8000 Loss: 133.59410095214844
Iteration: 9000 Loss: 124.08784484863281
Iteration: 10000 Loss: 131.39547729492188
Iteration: 11000 Loss: 134.51174926757812
Iteration: 12000 Loss: 126.70675659179688
Iteration: 13000 Loss: 119.51866912841797
Iteration: 14000 Loss: 134.053466796875
Iteration: 15000 Loss: 131.94847106933594
Iteration: 16000 Loss: 133.11114501953125
Iteration: 17000 Loss: 133.78762817382812
Iteration: 18000 Loss: 131.7386932373047
Iteration: 19000 Loss: 121.98905944824219
```

Part A.III

Plot the evolution of the ELBO.

Answer:

In [ ]:
```
plt.figure()
plt.plot(range(20000), elbos)
plt.title('ELBO vs. Number of Iterations')
plt.xlabel('Iteration')
plt.ylabel('ELBO')
plt.grid()
sns.despine(trim=True)
```



Part A.IV

Take 1,000 posterior samples.

Answer:

I'm giving you this one because it is a bit tricky. You need to use the `pyro.infer.Predictive` class to do it. Here is how you can use it:

In [ ]:
```
post_samples = pyro.infer.Predictive(model, guide=guide, num_samples=1000)(X, Y)
```

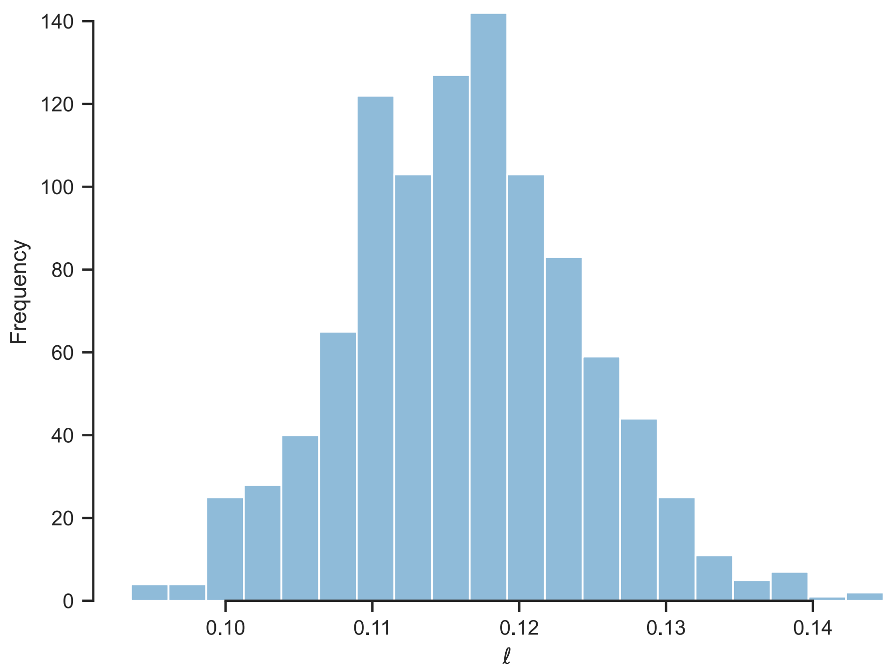
Part A.V

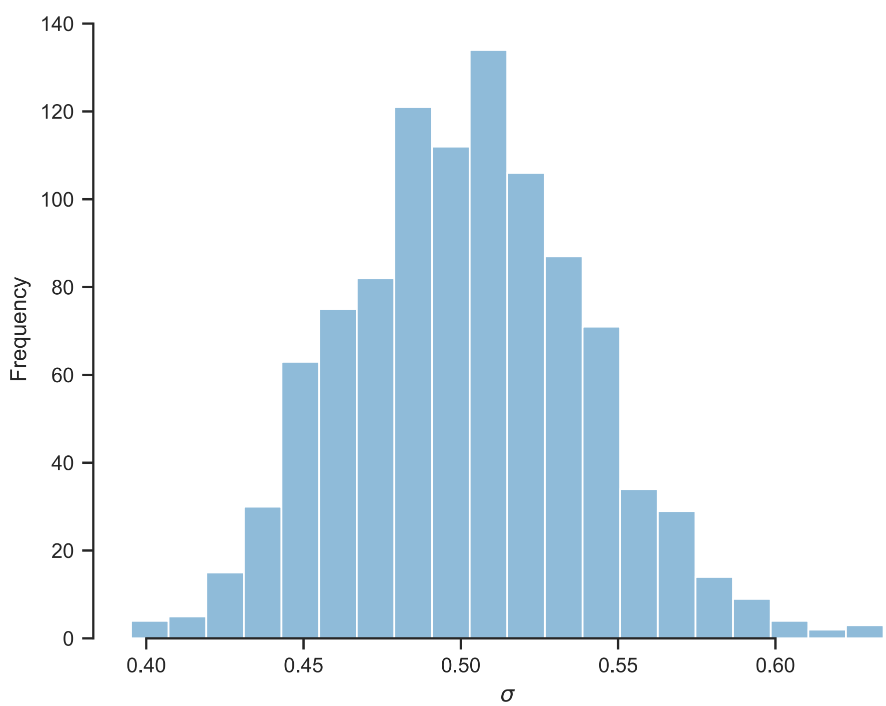
Plot the histograms of the posteriors of ℓ, σ, α_{10} and w_{10}.

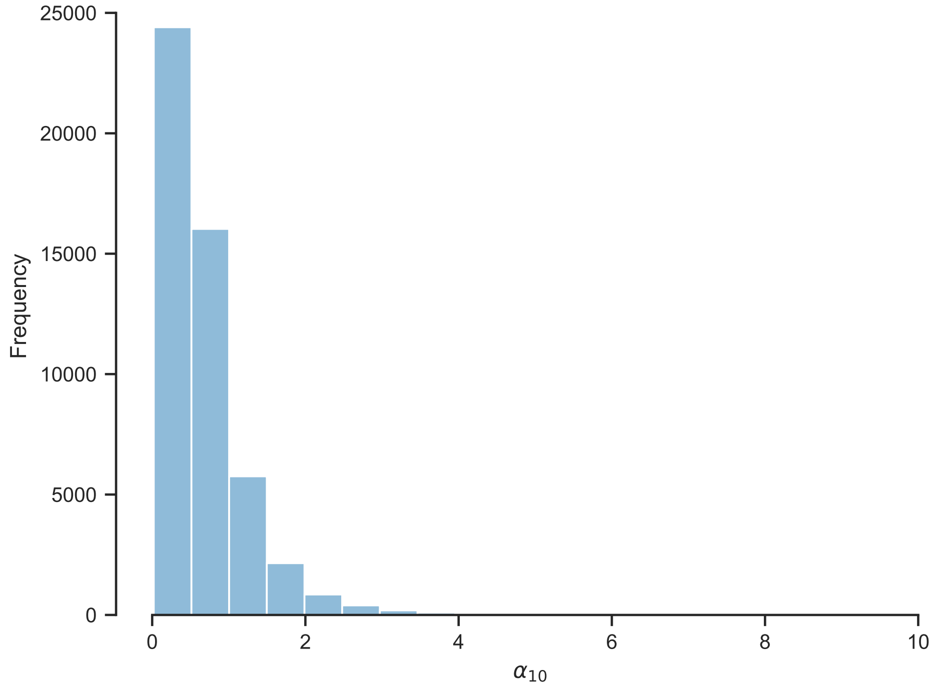
Answer:

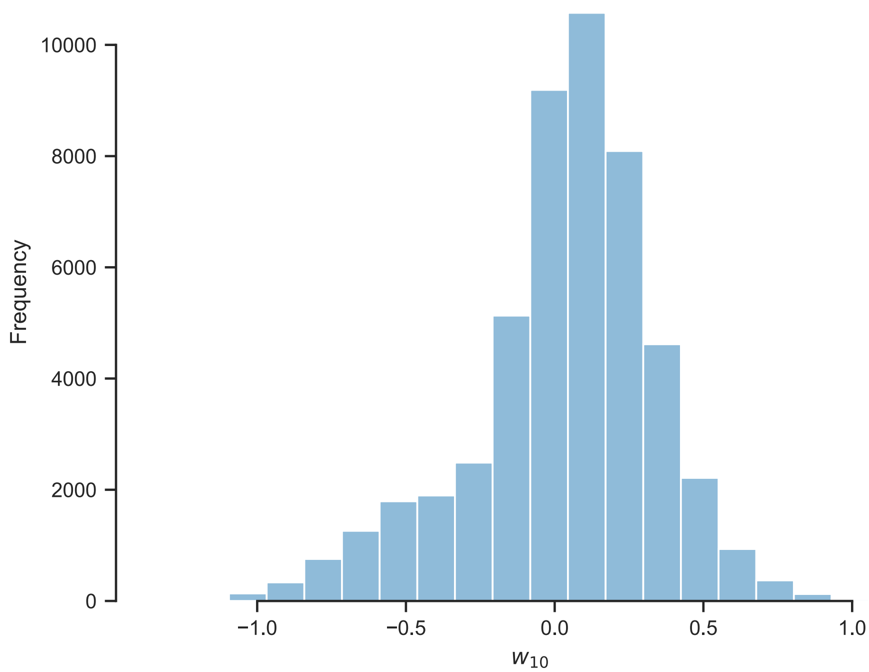
In [ ]:
```
# First, here is how to extract the samples.
ell = post_samples["ell"]
# You can do `post_samples.keys()` to see all the keys.
# But they should correspond to the names of the latent variables in the model.
sigma = post_samples["sigma"]
alphas = post_samples["alpha"]
ws = post_samples["w"]

# Here is the code to make the histogram for the length scale.
fig, ax = plt.subplots()
# **VERY IMPORTANT** - You need to detach the tensor from the computational graph.
# Otherwise, you will get very very strange behavior.
ax.hist(ell.detach().numpy(), bins=20, alpha=.5)
ax.set(xlabel="$\ell$", ylabel="Frequency")
sns.despine(trim=True)

fig, ax = plt.subplots()
ax.hist(sigma.detach().numpy(), bins=20, alpha=.5)
```

```
ax.set(xlabel="$\sigma$", ylabel="Frequency")
sns.despine(trim=True)

fig, ax = plt.subplots()
ax.hist(alphas.detach().numpy().ravel(), bins=20, alpha=.5)
ax.set(xlabel=r"$\alpha_{10}$", ylabel="Frequency")
sns.despine(trim=True)

fig, ax = plt.subplots()
ax.hist(ws.detach().numpy().ravel(), bins=20, alpha=.5)
ax.set(xlabel="$w_{10}$", ylabel="Frequency")
sns.despine(trim=True)
```









Part A.VI

Let's extend them model to make predictions.

Answer:

In [ ]:
```
# Again, I'm giving you most of the code here.

def predictive_model(X, y, num_centers=50):
    # First we run the original model get all the variables
    params = model(X, y, num_centers)
    # Here is how you can access the variables
    w = params["w"]
```

```
    ell = params["ell"]
    sigma = params["sigma"]
    x_centers = params["x_centers"]
    xs = torch.linspace(X.min(), X.max(), 100).unsqueeze(-1)
    # Evaluate the basis on the prediction points
    Phi = RadialBasisFunctions(x_centers, ell)(xs)
    # Make the predictions - we use a deterministic node here because we want to
    # save the results of the predictions.
    predictions = pyro.deterministic("predictions", Phi @ w)
    # Finally, we add the measurement noise
    predictions_with_noise = pyro.sample("predictions_with_noise", dist.Normal(predictions, sigma)
    return locals()
```

Part A.VII

Extract the posterior predictive distribution using 10,000 samples. Separate aleatory and epistemic uncertainty.

Answer:

In [ ]:

```
# Here is how to make the predictions. Just change the number of samples to the right number.
post_pred = pyro.infer.Predictive(predictive_model, guide=guide, num_samples=10000)(X, Y)
# We will predict here:
xs = torch.linspace(X.min(), X.max(), 100).unsqueeze(-1)
# You can extract the predictions from post_pred like this:
predictions = post_pred["predictions"]
# Note that we extracted the deterministic node called "predictions" from the model.
# Get the epistemic uncertainty in the usual way:
p_500, p_025, p_975 = np.percentile(predictions, [50, 2.5, 97.5], axis=0)
# Extract predictions with noise
predictions_with_noise = post_pred["predictions_with_noise"]
# Get the aleatory uncertainty
ap_025, ap_975 = np.percentile(predictions_with_noise, [2.5, 97.5], axis=0)
```

Part A.VIII

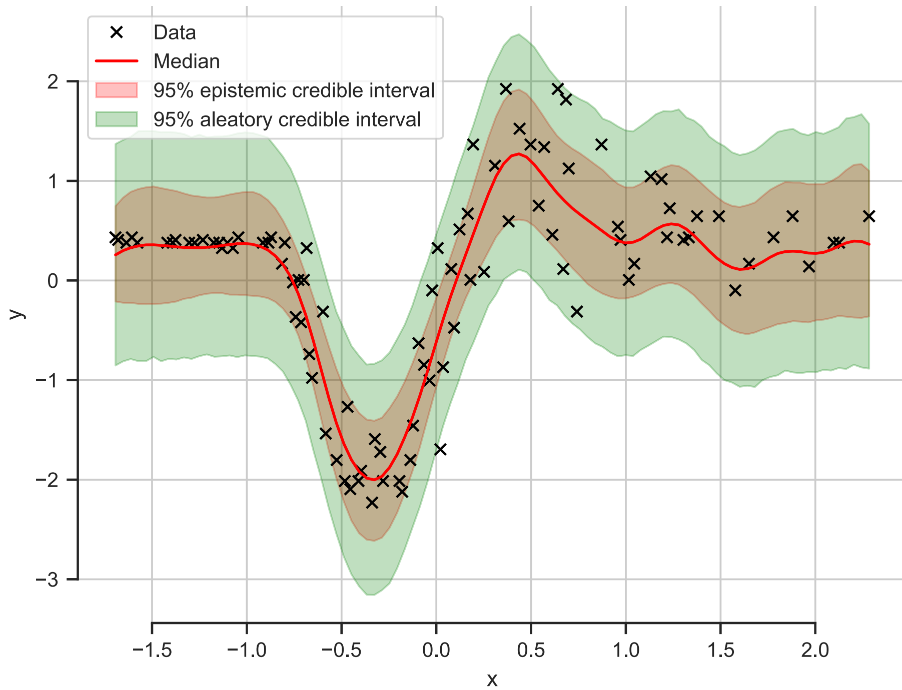
Plot the data, the median, the 95% credible interval of epistemic uncertainty and the 95% credible interval of aleatory uncertainty, along with five samples from the posterior.

Answer:

In [ ]:

```
fig, ax = plt.subplots()
ax.plot(X, Y, 'kx', label='Data')
ax.plot(xs.flatten(), p_500.T, 'r', label='Median')
ax.fill_between(xs.flatten(), p_025[0], p_975[0], color='red', alpha=0.25, label='95% epistemic cr
ax.fill_between(xs.flatten(), ap_025, ap_975, color='green', alpha=0.25, label='95% aleatory credi
# f_post_samples = predictions_with_noise.sample(
#     sample_shape=torch.Size([5])
# )
# ax.plot(
#     xs.numpy(),
#     f_post_samples.T.detach().numpy(),
#     color="red",
#     lw=0.5
# )
# # This is just to add the legend entry
# ax.plot(
#     [],
#     [],
#     color="red",
#     lw=0.5,
#     label="Posterior samples"
```

```
# )
ax.grid()
ax.set(xlabel='x', ylabel='y')
ax.legend()
sns.despine(trim=True)
```



Part B - Heteroscedastic regression

We are going to build a model that has an input-varying noise. Such models are called heteroscedastic models. Here I will let you do more of the work.

Everything is as before for ℓ, the α_j's, and the w_j's. We now introduce a model for the noise that is input dependent. It will use the same RBFs as the mean function. But let's use a different length-scale, ℓ_σ. So, we add:

$$\ell_\sigma \sim \text{Exponential}(1),$$

$$\alpha_{\sigma,j} \sim \text{Exponential}(1),$$

and

$$w_{\sigma,j}|\alpha_{\sigma,j} \sim N(0, \alpha_{\sigma,j}^2),$$

for $j = 1, \dots, m$.

Our model for the input-dependent noise variance is:

$$\sigma(x; \mathbf{w}_\sigma, \ell) = \exp\left(\mathbf{w}_\sigma^T \boldsymbol{\phi}(x; \ell_\sigma)\right).$$

So, the likelihood of the data is:

$$y_i|\mathbf{w}, \mathbf{w}_\sigma \sim N\left(\mathbf{w}^T \boldsymbol{\phi}(x_i; \ell), \sigma^2(x_i; \mathbf{w}_\sigma, \ell)\right),$$

You will implement this model.

Part B.I

Complete the code below:

```
In [ ]: def model(X, y, num_centers=50):
            with pyro.plate("centers", num_centers):
                alpha = pyro.sample("alpha", dist.Exponential(1.0))
                w = pyro.sample("w", dist.Normal(0.0, alpha))
                # Let's add the generalized linear model for the log noise.
                alpha_noise = pyro.sample("alpha_noise", dist.Exponential(1.0))
                w_noise = pyro.sample("w_noise", dist.Normal(0.0, alpha_noise))
            ell = pyro.sample("ell", dist.Exponential(1.))
            ell_noise = pyro.sample("ell_noise", dist.Exponential(1.0))
            x_centers = torch.linspace(X.min(), X.max(), num_centers).unsqueeze(-1)
            Phi = RadialBasisFunctions(x_centers, ell)(X)
            Phi_noise = RadialBasisFunctions(x_centers, ell_noise)(X)
            # This is the new part 2/2
            model_mean = Phi @ w
            sigma = torch.exp(Phi_noise @ w_noise)
            with pyro.plate("data", X.shape[0]):
                pyro.sample("y", dist.Normal(model_mean, sigma), obs=y)
            return locals()
```

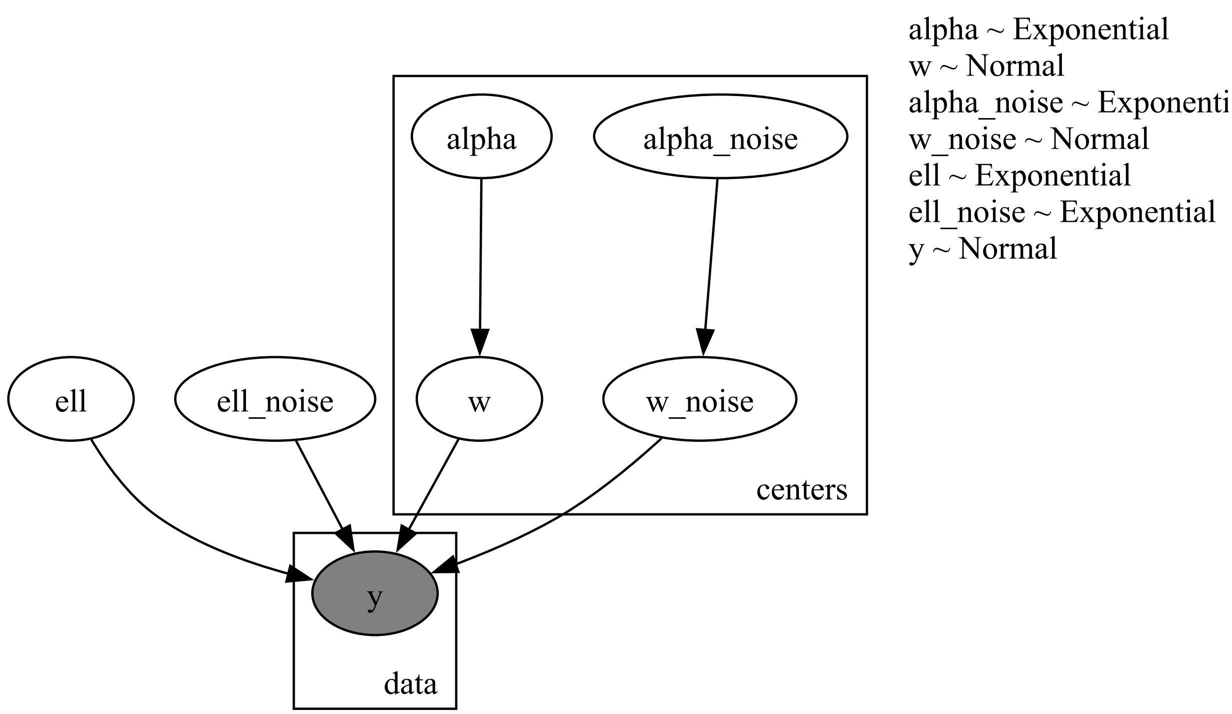
Make a `pyro.infer.autoguide.AutoDiagonalNormal` guide:

```
In [ ]: guide = pyro.infer.autoguide.AutoDiagonalNormal(model)
```

Make the graph of the model using `pyro` functionality:

```
In [ ]: pyro.render_model(model, (X, Y), render_distributions=True)
```

Out[ ]:



Part B.II

Train the model using 20,000 iterations. Then plot the evolution of the ELBO.

Answer:

In [ ]:
```
elbos, params = train(model, guide, (X, Y), num_iter=20000)
```

```
Iteration: 0 Loss: 493.8812561035156
Iteration: 1000 Loss: 339.9078674316406
Iteration: 2000 Loss: 245.21725463867188
Iteration: 3000 Loss: 199.9209442138672
Iteration: 4000 Loss: 190.2119598388672
Iteration: 5000 Loss: 182.6824951171875
Iteration: 6000 Loss: 180.13934326171875
Iteration: 7000 Loss: 160.67022705078125
Iteration: 8000 Loss: 183.66189575195312
Iteration: 9000 Loss: 173.7488250732422
Iteration: 10000 Loss: 173.80645751953125
Iteration: 11000 Loss: 170.54736328125
Iteration: 12000 Loss: 177.7602996826172
Iteration: 13000 Loss: 165.37124633789062
Iteration: 14000 Loss: 177.99627685546875
Iteration: 15000 Loss: 159.43516540527344
Iteration: 16000 Loss: 172.96884155273438
Iteration: 17000 Loss: 176.72186279296875
Iteration: 18000 Loss: 206.29103088378906
Iteration: 19000 Loss: 184.20729064941406
```

Part B.III

Extend the model to make predictions.

Answer:

In [ ]:
```
def predictive_model(X, y, num_centers=50):
    params = model(X, y, num_centers)
    w = params["w"]
    w_noise = params["w_noise"]
    ell = params["ell"]
    ell_noise = params["ell_noise"]
    sigma = params["sigma"]
    x_centers = params["x_centers"]
    xs = torch.linspace(X.min(), X.max(), 100).unsqueeze(-1)
    Phi = params["Phi"]
    Phi_noise = params["Phi_noise"]
    predictions = pyro.deterministic("predictions", Phi @ w)
    sigma = torch.exp(Phi_noise @ w_noise)
    predictions_with_noise = pyro.sample("predictions_with_noise", dist.Normal(predictions, sigma)
    return locals()
```

Part B.IV

Now, make predictions and calculate the epistemic and aleatory uncertainties as in part A.VII.

Answer:

In [ ]:
```
# Here is how to make the predictions. Just change the number of samples to the right number.
post_pred = pyro.infer.Predictive(predictive_model, guide=guide, num_samples=10000)(X, Y)
# We will predict here:
xs = torch.linspace(X.min(), X.max(), 94).unsqueeze(-1)
# You can extract the predictions from post_pred like this:
predictions = post_pred["predictions"]
# Note that we extracted the deterministic node called "predictions" from the model.
# Get the epistemic uncertainty in the usual way:
p_500, p_025, p_975 = np.percentile(predictions, [50, 2.5, 97.5], axis=0)
# Extract predictions with noise
predictions_with_noise = post_pred["predictions_with_noise"]
```

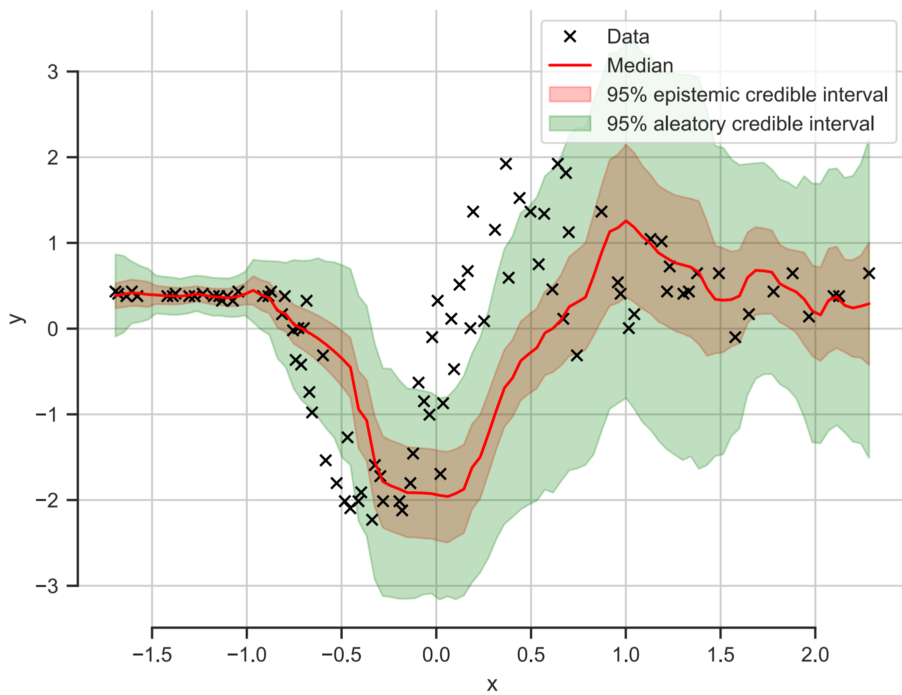
```python
# Get the aleatory uncertainty
ap_025, ap_975 = np.percentile(predictions_with_noise, [2.5, 97.5], axis=0)
```

Part B.V

Make the same plot as in part A.VIII.

Answer:

In [ ]:
```python
fig, ax = plt.subplots()
ax.plot(X, Y, 'kx', label='Data')
ax.plot(xs.flatten(), p_500.T, 'r', label='Median')
ax.fill_between(xs.flatten(), p_025[0], p_975[0], color='red', alpha=0.25, label='95% epistemic cr
ax.fill_between(xs.flatten(), ap_025, ap_975, color='green', alpha=0.25, label='95% aleatory credi
ax.grid()
ax.set(xlabel='x', ylabel='y')
ax.legend()
sns.despine(trim=True)
```



Part B.VI

Plot the estimated noise standard deviation as a function of of the input along with a 95% credible interval.

Answer:

In [ ]:
```python

```

Part B.VII

Which model do you prefer? Why?

Answer: I prefer the first model; it appears more stable overall

Part B.IX

Can you think of any way to improve the model? Go crazy! This is the last homework assignment! There is no right or wrong answer here. But if you have a good idea, we will give you extra credit.

In [ ]:
```
## Your code and answers here
```