In [82]:
```python
# Initialize task.
import jax.numpy as jnp
import numpy as np
import matplotlib.pyplot as plt
import seaborn as snb
import matplotlib.colors as colors
from scipy.stats import norm
from scipy.optimize import minimize
from jax import hessian, random

snb.set_theme(font_scale=1.25)

# Load Data

data = jnp.load('./data_exercise5b.npz')
X = data['day']
y = np.log(data['bike_count'])

#Standardize data
ym, ys = jnp.mean(y), jnp.std(y)
y = (y - ym) / ys
```

Part 1: Fully Bayesian inference for Gaussian process regression

Task 1.1 Choose a value for v such that the prior probability of observing a lengthscale larger than 100 is approximately 1%

First, we identify that our lengthscale parameter follows af half normal distribution

$$\ell \sim \mathcal{N}_+(0, v)$$

We know the condition we need to fulfill is

$$P(\ell > 100) \approx 0.01$$

We can express our distribution $\ell = |Z\sqrt{v}|$ with Z being a standard normal variable, since ℓ is a half normal. Since the standard normal distribution is symmetric we view it as a two-tailed probability.

$$\begin{aligned}
P(\ell > 100) = P(|Z\sqrt{v}| > 100) &\approx 0.01 \\
P(|Z| > \frac{100}{\sqrt{v}}) &\approx 0.01 \qquad \text{(Two-tailed)} \\
2 \cdot P(\frac{Z > 100}{\sqrt{v}}) &\approx 0.01 \\
P(\frac{Z > 100}{\sqrt{v}}) &\approx 0.005 \\
\Rightarrow v = \left(\frac{100}{z_{0.005}}\right)^2 &= 1507.18 \approx 1507
\end{aligned}$$

So when $v \approx 1507$ the probability of the lengthscale parameter being above 100 will be roughly 1%

Task 1.2 Determine the marginalized distribution $p(y, \sigma, k, \ell)$

The joint distribution is as follows

$$\begin{aligned}
p(\boldsymbol{y}, f, \sigma, \kappa, \ell) &= p\left(y \mid \boldsymbol{f}, \sigma^2\right) p(f \mid \kappa, \ell) p(\kappa) p(\ell) p(\sigma) \\
&= \mathcal{N}\left(\boldsymbol{y} \mid \boldsymbol{f}, \sigma^2 \boldsymbol{I}\right) \mathcal{N}(\boldsymbol{f} \mid \boldsymbol{0}, \boldsymbol{K}) \mathcal{N}_+(\kappa \mid 0,1) \mathcal{N}_+(\ell \mid 0, v) \mathcal{N}_+(\sigma \mid 0,1).
\end{aligned}$$

Denoting $\mathcal{N}_+(\kappa \mid 0,1)\mathcal{N}_+(\ell \mid 0, v)\mathcal{N}_+(\sigma \mid 0,1) = p(\kappa, \ell, \sigma)$ we get the following for marginalising out f from the joint distribution:

$$\begin{aligned}
p(\boldsymbol{y}, \sigma, \kappa, \ell) &= \int p(\boldsymbol{y}, f, \sigma, \kappa, \ell) \, \mathrm{d}f \\
&= p(\kappa, \ell, \sigma) \underbrace{\int \mathcal{N}\left(\boldsymbol{y} \mid \boldsymbol{f}, \sigma^2 \boldsymbol{I}\right) \mathcal{N}(\boldsymbol{f} \mid \boldsymbol{0}, \boldsymbol{K}) \, \mathrm{d}f}_{\text{Marginal likelihood} = \text{linear Gaussian system}} \\
&= p(\kappa, \ell, \sigma) \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{0}, \boldsymbol{K} + \sigma^2 \boldsymbol{I})
\end{aligned}$$

Task 1.3: Implement a Metropolis sampler using the proposal distribution

In [83]:
```python
def metropolis(log_target, num_params, tau, num_iter, theta_init=None, seed=0):
    """
    Runs a Metropolis-Hastings sampler.

    Arguments:
        log_target: function evaluating the log target, expecting a vector (size=num_params)
        num_params: number of parameters
        tau:        vector of proposal standard deviations (one per parameter)
        num_iter:   number of iterations
        theta_init: initial parameter vector (or None)
        seed:       random seed

    Returns:
        thetas:   jnp.array of shape (num_iter+1, num_params)
        accepts:  list of acceptance flags
    """
    key = random.PRNGKey(seed)
    if theta_init is None:
        theta_init = jnp.zeros((num_params))
    thetas = [theta_init]
    accepts = []
    log_p_theta = log_target(theta_init)

    for k in range(num_iter):
        key, key_proposal, key_accept = random.split(key, num=3)
        theta_cur = thetas[-1]
        # Elementwise proposal using tau for each parameter.
        theta_star = theta_cur + tau * random.normal(key_proposal, shape=(num_params,))
        log_p_theta_star = log_target(theta_star)
        log_r = log_p_theta_star - log_p_theta
        A = min(1, jnp.exp(log_r))
        if random.uniform(key_accept) < A:
            theta_next = theta_star
            log_p_theta = log_p_theta_star
            accepts.append(1)
        else:
            theta_next = theta_cur
            accepts.append(0)
        thetas.append(theta_next)

    print("Acceptance ratio: %3.2f" % jnp.mean(jnp.array(accepts)))
    thetas = jnp.stack(thetas)
    assert thetas.shape == (
        num_iter + 1,
        num_params,
    ), f"Expected shape {(num_iter+1, num_params)}, got {thetas.shape}."
    return thetas, accepts
```

In [ ]:
```python
def metropolis_multiple_chains(
    log_target, num_params, num_chains, tau, num_iter, theta_init, seeds, warm_up=0
):
    """Runs multiple Metropolis-Hastings chains. The i'th chain should be initialized using the i'th vector in theta_init

    Arguments:
        log_target:         function for evaluating the log joint distribution
        num_params:         number of parameters of the joint distribution (integer)
        num_chains:         number of MCMC chains
        tau:                proposal standard deviation (jnp.array with shape (num_params,))
        num_iter:           number of iterations for each chain (integer)
        theta_init:         array of initial values (jnp.array with shape (num_chains, num_params))
        seeds:              seed for each chain (jnp.array with shape (num_chains))
        warm_up:            number of warm up samples to be discarded

    returns:
        thetas              jnp.array of samples from each chain after warmup (shape: num_chains x (num_iter + 1 - warm_u
        accept_rates        jnp.array of acceptances rate for each chain (shapes: num_chains)

    """

    # verify dimension of initial parameters
    assert theta_init.shape == (
        num_chains,
        num_params,
    ), "theta_init seems to have the wrong dimensions. Plaese check your code."

    # prepare arrays for storing samples
    thetas = []
    accept_rates = []
    # run sampler for each chain
    for idx_chain in range(num_chains):
        print(f"Running chain {idx_chain}. ", end="")
        thetas_temp, accepts_temp = metropolis(
            log_target,
            num_params,
            tau,
            num_iter,
            theta_init=theta_init[idx_chain],
            seed=seeds[idx_chain],
        )
        thetas.append(thetas_temp)
        accept_rates.append(jnp.array(accepts_temp))

    thetas = jnp.stack(thetas, axis=0)
    accept_rates = jnp.stack(accept_rates, axis=0)

    # discard warm-up samples
    thetas = thetas[:, warm_up:, :]

    # verify dimensions and return
    assert thetas.shape == (
        num_chains,
        num_iter + 1 - warm_up,
        num_params,
    ), f"The expected shape of chains is ({num_chains}, {num_iter+1-warm_up}, {num_params}) corresponding to (num_chains,
    assert len(accept_rates) == num_chains
    return thetas, accept_rates
```

In [ ]:
```python
# Log half-normal prior.
def log_halfnormal(x, scale=1.0):
    return jnp.log(2 / scale) + norm.logpdf(x, 0, scale)


def squared_exponential(tau, kappa, lengthscale):
    return kappa**2 * jnp.exp(-0.5 * tau**2 / (lengthscale**2))


class StationaryIsotropicKernel(object):

    def __init__(self, kernel_fun, kappa=1.0, lengthscale=1.0):
        """
        the argument kernel_fun must be a function of three arguments kernel_fun(||tau||, kappa, lengthscale), e.g.
        squared_exponential = lambda tau, kappa, lengthscale: kappa**2*np.exp(-0.5*tau**2/lengthscale**2)
        """
        self.kernel_fun = kernel_fun
        self.kappa = kappa
        self.lengthscale = lengthscale

    def contruct_kernel(self, X1, X2, kappa=None, lengthscale=None, jitter=1e-8):
        """compute and returns the NxM kernel matrix between the two sets of input X1 (shape NxD) and X2 (MxD) using the

        arguments:
            X1              -- NxD matrix
            X2              -- MxD matrix
            kappa           -- magnitude (positive scalar)
            lengthscale     -- characteristic lengthscale (positive scalar)
            jitter          -- non-negative scalar

        returns
            K               -- NxM matrix
        """

        # extract dimensions
        N, M = X1.shape[0], X2.shape[0]

        # prep hyperparameters
        kappa = self.kappa if kappa is None else kappa
        lengthscale = self.lengthscale if lengthscale is None else lengthscale

        # compute all the pairwise distances efficiently
        dists = jnp.sqrt(
            jnp.sum((jnp.expand_dims(X1, 1) - jnp.expand_dims(X2, 0)) ** 2, axis=-1)
        )

        # squared exponential covariance function
        K = self.kernel_fun(dists, kappa, lengthscale)
        # add jitter to diagonal for numerical stability
        if len(X1) == len(X2) and jnp.allclose(X1, X2):
            K = K + jitter * jnp.identity(len(X1))


        assert K.shape == (
            N,
            M,
        ), f"The shape of K appears wrong. Expected shape ({N}, {M}), but the actual shape was {K.shape}. Please check yo
        return K


# Log joint function: returns -inf if any parameter ≤ 0.
def log_marginal_likelihood(theta, X, y, v=1507.0):
    kappa, ell, sigma = theta
    n = y.shape[0]
    if (kappa <= 0) or (ell <= 0) or (sigma <= 0):
        return -jnp.inf
    # Priors.
    lp_kappa = log_halfnormal(kappa, scale=1.0)
    lp_ell = log_halfnormal(ell, scale=jnp.sqrt(v))
    lp_sigma = log_halfnormal(sigma, scale=1.0)
    log_prior = lp_kappa + lp_ell + lp_sigma

    # Kernel matrix computation.
    kernel = StationaryIsotropicKernel(squared_exponential, kappa, ell)
    K = kernel.contruct_kernel(X, X)
    C = K + sigma**2 * jnp.eye(n)

    # Compute the Cholesky decomposition.
    L = jnp.linalg.cholesky(C)
    v_vec = jnp.linalg.solve(L, y)

    # Compute log marginal likelihood.
    logdet_term = jnp.sum(jnp.log(jnp.diag(L)))
    quad_term   = 0.5 * jnp.sum(v_vec**2)
    const_term  = -0.5 * n * jnp.log(2 * jnp.pi)
    log_likelihood = const_term - logdet_term - quad_term

    return log_prior + log_likelihood

def log_target(theta):
    """Log target function for the Metropolis-Hastings sampler.

    Arguments:
        theta:             jnp.array of parameters (jnp.array with shape (num_params))

    Returns:
        log_target:        log target distribution (real number)
    """
    # Compute the log target distribution.
    log_mar_likelihood = log_marginal_likelihood(theta, X, y)
    log_target = log_mar_likelihood
    return log_target
```

In [98]:
```
# mcmc settings
num_chains = 4
num_iter = 10000
tau = jnp.array([1.0, 100.0, 0.1])
num_params = 3

warm_up = 0
seeds = jnp.arange(num_chains)

# generate initial values from uniform distribution
key = random.PRNGKey(1)
theta_init = random.uniform(
    key, shape=(num_chains, num_params), minval=0.1, maxval=2.0
)

# sample
chains, accepts = metropolis_multiple_chains(
    log_target,
    num_params,
    num_chains,
    tau,
    num_iter,
    theta_init=theta_init,
    seeds=seeds,
    warm_up=warm_up,
)
# report estimated mean and variance
print(f"\nEstimated mean:\t\t{jnp.mean(chains.ravel()):+3.2f}")
print(f"Estimated variance:\t{jnp.var(chains.ravel()):+3.2f}")
```

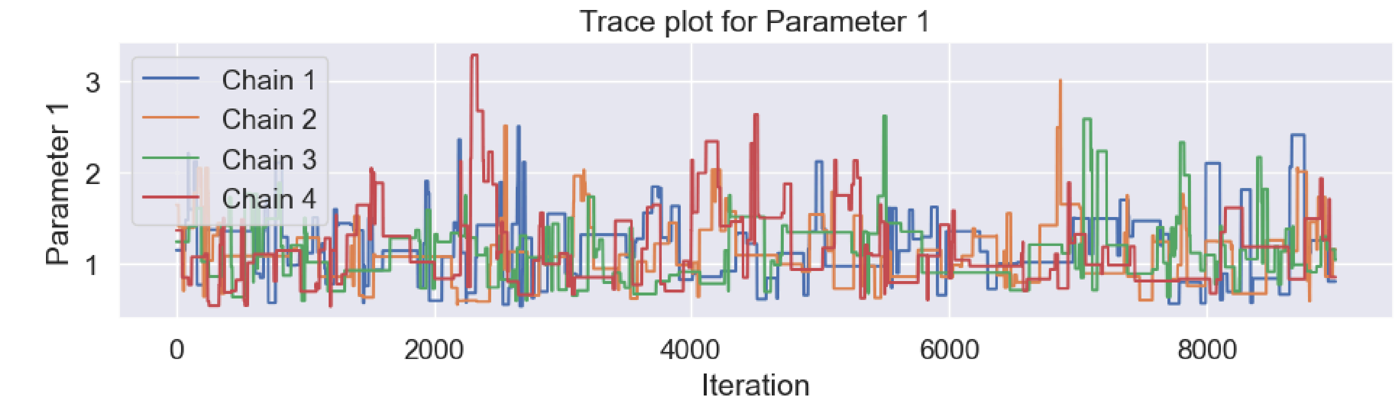
```
Running chain 0. Acceptance ratio: 0.02
Running chain 1. Acceptance ratio: 0.02
Running chain 2. Acceptance ratio: 0.02
Running chain 3. Acceptance ratio: 0.02

Estimated mean:         +18.20
Estimated variance:     +644.37
```

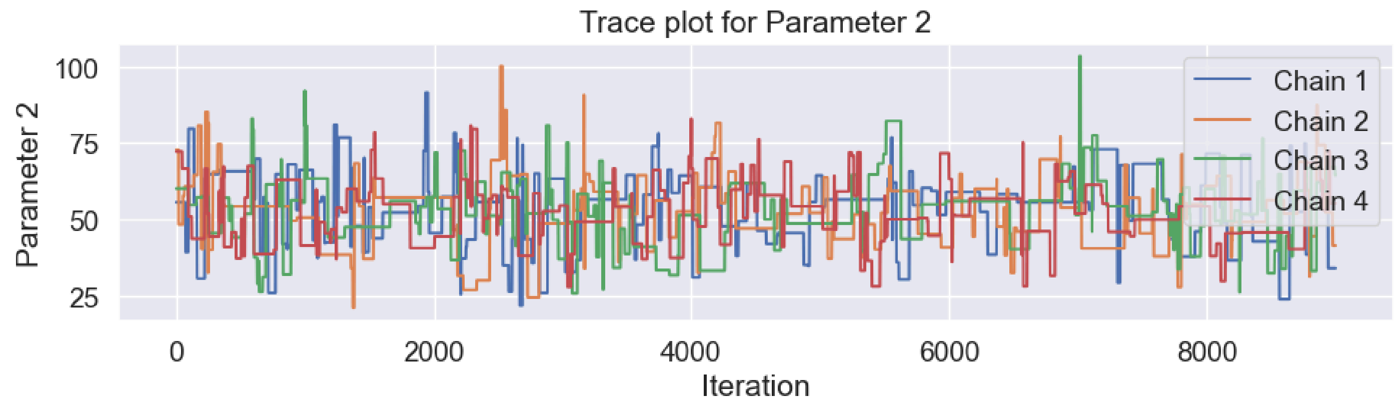
Task 1.4: Plot the trace for each parameter and report the convergence diagnostics

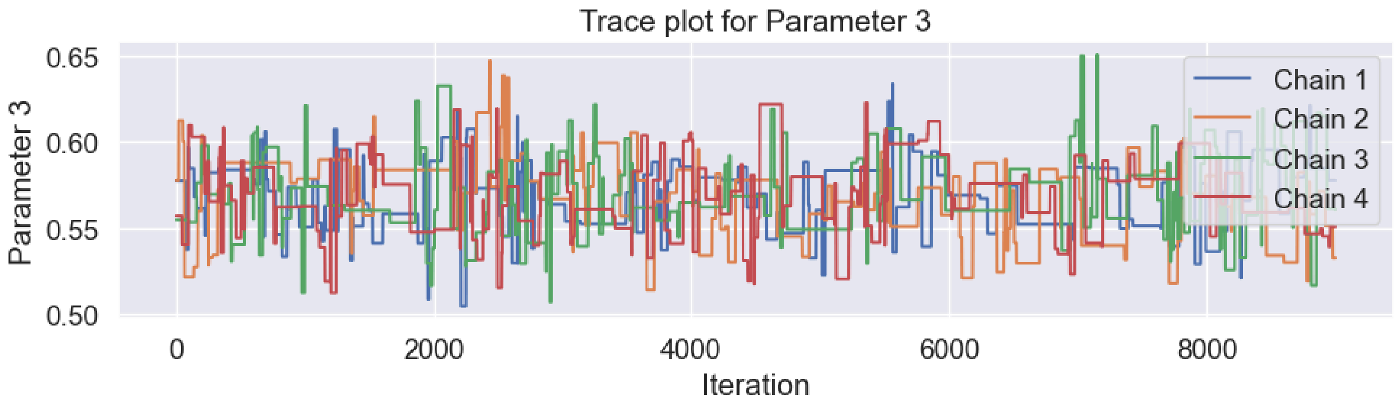
In [108…

```
def compute_Rhat(chains):
    """
    Compute the Gelman-Rubin Rhat diagnostic for each parameter.
    Expects chains to be a jnp.array of shape (num_chains, num_samples, num_params).
    Returns a jnp.array of shape (num_params,).
    """
    num_chains, num_samples, num_params = chains.shape
    chain_means = jnp.mean(chains, axis=1)  # shape: (num_chains, num_params)
    overall_mean = jnp.mean(chains, axis=(0, 1))  # shape: (num_params,)
    B = (
        num_samples
        / (num_chains - 1)
        * jnp.sum((chain_means - overall_mean) ** 2, axis=0)
    )  # between-chain variance.
    # Use ddof=1 for unbiased sample variance
    W = jnp.mean(jnp.var(chains, axis=1, ddof=1), axis=0)  # within-chain variance.
    var_plus = ((num_samples - 1) / num_samples) * W + (1 / num_samples) * B
    Rhat = jnp.sqrt(var_plus / W)
    return Rhat


def compute_effective_sample_size(chains):
    """
    Compute a basic effective sample size (S_eff) approximation for each parameter.
    This implementation uses the lag autocorrelations of the merged chains.
    Returns a jnp.array of shape (num_params,).
    """
    num_chains, num_samples, num_params = chains.shape
    total_samples = int(num_chains * num_samples)
    S_eff = jnp.zeros(num_params)
    chains_np = np.array(chains)  # switch to numpy for autocorrelation computations

    for p in range(num_params):
        # merge chains for parameter p
        x = chains_np[:, :, p].reshape(-1)
        # compute autocorrelation for increasing lag until the correlation becomes negative
        ac_sum = 0.0
        for lag in range(1, num_samples):
            # compute autocorrelation at lag 'lag'
            corr = np.corrcoef(x[:-lag], x[lag:])[0, 1]
            if corr < 0:
                break
            ac_sum += corr
        S_eff = S_eff.at[p].set(total_samples / (1 + 2 * ac_sum))
    return S_eff

warm_up = 1000
# Compute number of warm-up samples discarded (per chain)
total_warm_up = num_chains * warm_up
print(f"Number of warm-up samples discarded: {total_warm_up}")

# Compute estimated overall mean and variance, pooling all samples
all_samples = chains.reshape(-1, chains.shape[-1])  # shape: (total_samples, num_params)

# Compute convergence diagnostics.
Rhat = compute_Rhat(chains)
total_samples = num_chains * (chains.shape[1])  # after warm-up
S_eff = compute_effective_sample_size(chains)

# Print results
for p in range(all_samples.shape[1]):
    print(f"\nParameter {p+1}:")
    print(f"  Effective sample size:  {S_eff[p]:.0f}")
    print(f"  ˆR:                     {Rhat[p]:3.2f}")

# Plot trace for each parameter.
num_params = chains.shape[-1]
fig, axes = plt.subplots(num_params, 1, figsize=(10, 3 * num_params))
if num_params == 1:
    axes = [axes]  # ensure axes is iterable

for p in range(num_params):
    for chain in range(chains.shape[0]):
        axes[p].plot(chains[chain, warm_up:, p], label=f"Chain {chain+1}")
    axes[p].set_xlabel("Iteration")
    axes[p].set_ylabel(f"Parameter {p+1}")
    axes[p].set_title(f"Trace plot for Parameter {p+1}")
    axes[p].legend()

plt.tight_layout()
plt.show()
```

```
Number of warm-up samples discarded: 4000

Parameter 1:
  Effective sample size:  304
  ˆR:                     1.00

Parameter 2:
  Effective sample size:  414
  ˆR:                     1.00

Parameter 3:
  Effective sample size:  440
  ˆR:                     1.00
```







Task 1.5: Estimate and report the posterior mean for each hyperparameter. Report the MCSE for each estimate.

In [109…

```python
# Compute estimated overall mean and variance, pooling all samples
all_samples = chains.reshape(-1, chains.shape[-1])  # shape: (total_samples, num_params)
estimated_mean = jnp.mean(all_samples, axis=0)
estimated_var = jnp.var(all_samples, axis=0)
estimated_std = jnp.sqrt(estimated_var)
MC_error = estimated_std / jnp.sqrt(S_eff)

for p in range(all_samples.shape[1]):
    print(f"\nParameter {p+1}:")
    print(f"  Estimated mean:         {estimated_mean[p]:+3.2f}")
    print(f"  MCSE:                   {MC_error[p]:+3.2f}")
```

```
Parameter 1:
  Estimated mean:         +1.15
  MCSE:                   +0.02

Parameter 2:
  Estimated mean:         +52.89
  MCSE:                   +0.56

Parameter 3:
  Estimated mean:         +0.57
  MCSE:                   +0.00
```

Task 1.6: Estimate a 95% posterior credibility interval for each hyperparameter.

In [ ]:

```python
# Compute 95% credibility intervals for each hyperparameter.
cred_intervals = np.percentile(np.array(all_samples), [2.5, 97.5], axis=0)

for p in range(cred_intervals.shape[1]):
    print(
        f"Parameter {p+1} 95% Credibility Interval: [{cred_intervals[0, p]:.2f}, {cred_intervals[1, p]:.2f}]"
    )
```

```
Parameter 1 95% Credibility Interval: [0.60, 2.11]
Parameter 2 95% Credibility Interval: [30.05, 74.45]
Parameter 3 95% Credibility Interval: [0.53, 0.61]
```

In [ ]:

```python
data = jnp.load('Gustav/data_assignment3.npz')
x_data, y_data = data['x'], data['t']
```

In [ ]:

```python
def design_matrix(x):
    return np.column_stack((x,np.ones(len(x))))
```

In [ ]:

```python
# Normal distribution
def log_npdf(x, m, v):
    return -0.5 * (x - m) ** 2 / (v) - 0.5 * jnp.log(2 * jnp.pi * v)


def npdf(x, m, v):
    return jnp.exp(log_npdf(x, m, v))


# Half-normal distribution
def log_half_npdf(x, m, v):
    return jnp.log(2) - 0.5 * (x - m) ** 2 / (v) - 0.5 * jnp.log(2 * jnp.pi * v)


def half_npdf(x, m, v):
    return jnp.exp(log_half_npdf(x, m, v))

# Logistic function
def sigmoid(z):
    return 1 / (1 + jnp.exp(-z))
```

In [ ]:

```python
def plot_summary(
    ax,
    x,
    s,
    interval=95,
    num_samples=0,
    sample_color="k",
    sample_alpha=0.4,
    interval_alpha=0.25,
    color="r",
    legend=True,
    title="",
    plot_mean=True,
    plot_median=False,
    label="",
    seed=0,
):
    b = 0.5 * (100 - interval)
    lower = jnp.percentile(s, b, axis=0).T
    upper = jnp.percentile(s, 100 - b, axis=0).T

    if plot_median:
        median = jnp.percentile(s, 50, axis=0).T
        lab = "Median"
        if label:
            lab += " " + label
        ax.plot(x.ravel(), median, label=lab, color=color, linewidth=4)

    if plot_mean:
        mean = jnp.mean(s, axis=0).T
        lab = "Mean"
        if label:
            lab += " " + label
        ax.plot(x.ravel(), mean, "--", label=lab, color=color, linewidth=4)

    ax.fill_between(
        x.ravel(),
        lower.ravel(),
        upper.ravel(),
        color=color,
        alpha=interval_alpha,
        label=f"{interval}% Interval",
    )

    if num_samples > 0:
        np.random.seed(seed)
        idx_samples = np.random.choice(s.shape[0], size=num_samples, replace=False)
        ax.plot(x, s[idx_samples, :].T, color=sample_color, alpha=sample_alpha)

    if legend:
        ax.legend(loc="best")

    if title:
        ax.set_title(title, fontweight="bold")
```

Part 2: Regression modelling using mixture of experts

Mixture of experts (MoE) model for regression

We consider two different linear models $\boldsymbol{y}_n = \boldsymbol{w}_0^T \boldsymbol{x}_n + e_n$ and $\boldsymbol{y}_n = \boldsymbol{w}_1^T \boldsymbol{x}_n + e_n$, with weights $\boldsymbol{w}_0$ and $\boldsymbol{w}_1$ in respect to two different regions of $\boldsymbol{x}_n$, controlled by the latent variable $z_n \in \{0,1\}$ which we let be Bernoulli distributed given the data $\boldsymbol{x}_n$

$$z_n \mid \boldsymbol{v} \sim \mathrm{Ber}\big(\sigma\left(\boldsymbol{v}^T \boldsymbol{x}_n\right)\big),$$

where (v) are parameters and $\boldsymbol{x}_n = [x_n, 1]$.
We then construct the conditional likelyhood as Gaussians for both data regions determined by the latent variable such that

$$p\left(y_n \mid x_n, z_n, \boldsymbol{w}_0, \boldsymbol{w}_1, \sigma_0^2, \sigma_1^2\right) = \begin{cases} \mathcal{N}\left(y_n \mid \boldsymbol{w}_1^T \boldsymbol{x}_n, \sigma_1^2\right) & \text{if} \quad z_n = 1, \\ \mathcal{N}\left(y_n \mid \boldsymbol{w}_0^T \boldsymbol{x}_n, \sigma_0^2\right) & \text{if} \quad z_n = 0, \end{cases} = \mathcal{N}\left(y_n \mid \boldsymbol{w}_{z_n}^T \boldsymbol{x}_n, \sigma_{z_n}^2\right),$$

with noise variance $\sigma_{z_n}^2$. Adding the following generic priors, we write up the joint distribution

$$\begin{aligned} \tau, \sigma_0, \sigma_1 &\sim \mathcal{N}_+(0,1), \\ w_0, w_1, \boldsymbol{v} &\sim \mathcal{N}\left(\boldsymbol{0}, \tau^2 \boldsymbol{I}\right), \\ y_n \mid z_n &\sim \mathcal{N}\left(\boldsymbol{w}_{z_n}^T \boldsymbol{x}_n, \sigma_{z_n}^2\right), \end{aligned}$$

joint distribution:

$$\begin{aligned} p\left(\boldsymbol{y}, \boldsymbol{w}_1, \boldsymbol{w}_0, \boldsymbol{v}, \boldsymbol{z}, \tau, \sigma_0, \sigma_1 \mid \boldsymbol{x}\right) = &\left[\prod_{n=1}^{N} \mathcal{N}\left(y_n \mid \boldsymbol{w}_{z_n}^T \boldsymbol{x}_n, \sigma_{z_n}^2\right) \mathrm{Ber}\big(z_n \mid \sigma\left(\boldsymbol{v}^T \boldsymbol{x}_n\right)\big)\right] \mathcal{N}\left(\boldsymbol{w}_0 \mid \boldsymbol{0}, \tau^2 \boldsymbol{I}\right) \\ &\mathcal{N}\left(\boldsymbol{w}_1 \mid \boldsymbol{0}, \tau^2 \boldsymbol{I}\right) \mathcal{N}\left(\boldsymbol{v} \mid \boldsymbol{0}, \tau^2 \boldsymbol{I}\right) \mathcal{N}_+\left(\tau \mid 0,1\right) \mathcal{N}_+\left(\sigma_0 \mid 0,1\right) \mathcal{N}_+\left(\sigma_1 \mid 0,1\right). \end{aligned}$$

Task 2.1: Marginalize out each z_n from to joint model.

Using the sum rule we marginalize out z_n from the joint distribution. Remembering that z_n is discrete, we have to sum and not integrate over z_n.

To simplify the sum, we will denote all Gaussians that are not dependent on z_n as $p(\neg z)$ and sum over all 2^N combinations of z_n, $\boldsymbol{z} = \{z_n\}_{n=1}^N$. Since each observation is independent of each other, we can move the sum inside the product $\left(\sum_{\boldsymbol{z}} = \sum_{z_1}\sum_{z_2}\cdots\sum_{z_N} = \prod_{n=1}^N \sum_{z_n}\right)$.

$$\begin{aligned}\sum_{\boldsymbol{z}} p(\boldsymbol{y}, \boldsymbol{w}_1, \boldsymbol{w}_0, \boldsymbol{v}, \boldsymbol{z}, \tau, \sigma_0, \sigma_1 \mid \boldsymbol{x}) &= p(\neg \boldsymbol{z}) \sum_{\boldsymbol{z}} \left[\prod_{n=1}^N \mathcal{N}\left(y_n \mid \boldsymbol{w}_{z_n}^T \boldsymbol{x}_n, \sigma_{z_n}^2\right) \mathrm{Ber}\left(z_n \mid \sigma\left(\boldsymbol{v}^T \boldsymbol{x}_n\right)\right)\right] \\ &= p(\neg \boldsymbol{z}) \left[\prod_{n=1}^N \sum_{z_n} \mathcal{N}\left(y_n \mid \boldsymbol{w}_{z_n}^T \boldsymbol{x}_n, \sigma_{z_n}^2\right) \sigma\left(\boldsymbol{v}^T \boldsymbol{x}_n\right)^{z_n} \left(1 - \sigma\left(\boldsymbol{v}^T \boldsymbol{x}_n\right)\right)^{1-z_n}\right] \\ &= p(\neg \boldsymbol{z}) \prod_{n=1}^N \underbrace{\left[\mathcal{N}\left(y_n \mid \boldsymbol{w}_1^T \boldsymbol{x}_n, \sigma_1^2\right) \sigma\left(\boldsymbol{v}^T \boldsymbol{x}_n\right) + \mathcal{N}\left(y_n \mid \boldsymbol{w}_0^T \boldsymbol{x}_n, \sigma_0^2\right) \sigma\left(-\boldsymbol{v}^T \boldsymbol{x}_n\right)\right]}_{\text{Likelihood for each observation}}\end{aligned}$$

We see that the likelihood for each observation has become a mixture of gaussians after the marginalization of z_n.

Task 2.2: Python function to evaluate the marginalized log joint distribution

We will separate the terms from above, such that we have:

$$\log p(\boldsymbol{y}, \boldsymbol{w}_0, \boldsymbol{w}_1, \boldsymbol{v}, \tau, \sigma_0, \sigma_1 | \boldsymbol{x}) = \log p(\neg \boldsymbol{z}) + \log p(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{w}_0, \boldsymbol{w}_1, \boldsymbol{v}, \sigma_0, \sigma_1)$$

$$\log p(\neg \boldsymbol{z}) = -3\log(2\pi) - 6\log\tau - \frac{\boldsymbol{w}_0^T \boldsymbol{w}_0 + \boldsymbol{w}_1^T \boldsymbol{w}_0 + \boldsymbol{v}^T \boldsymbol{v}}{2\tau^2} + \frac{3}{2}\log(2) - \frac{3}{2}\log(\pi) - \frac{\tau^2 + \sigma_0^2 + \sigma_1^2}{2}$$

$$\log p(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{w}_0, \boldsymbol{w}_1, \boldsymbol{v}, \sigma_0, \sigma_1) = \sum_{n=1}^N \log\left[\mathcal{N}\left(y_n \mid \boldsymbol{w}_1^T \boldsymbol{x}_n, \sigma_1^2\right) \sigma\left(\boldsymbol{v}^T \boldsymbol{x}_n\right) + \mathcal{N}\left(y_n \mid \boldsymbol{w}_0^T \boldsymbol{x}_n, \sigma_0^2\right) \sigma\left(-\boldsymbol{v}^T \boldsymbol{x}_n\right)\right]$$

We see that we can vectorize the Gaussian mixture terms wrt. the number of data points for the logaritmic transformation which was not possible for the joint distribution.

```
In [ ]: def log_joint(y, x, theta):
            """
            Evaluates the log joint distribution

            Parameters:
              y      : array of N observations (shape: (N,))
              x      : design matrix for N observations (shape: (N,2)); each row is [x_n, 1]
              theta  : vector of parameters of shape (9,), where
                       theta[0:2]  -> w0 (slope and intercept)
                       theta[2:4]  -> w1 (slope and intercept)
                       theta[4:6]  -> v  (slope and intercept)
                       theta[6]    -> tau
                       theta[7]    -> sigma0
                       theta[8]    -> sigma1

            Returns:
              log joint density evaluated at theta.
            """

            # Unpack parameters
            w0 = theta[0:2]
            w1 = theta[2:4]
            v = theta[4:6]
            tau = theta[6]
            sigma0 = theta[7]
            sigma1 = theta[8]


            # ---------- Prior term (log p(~z)) ----------
            # log p(~z) = - 3log(2pi) - 6log(tau) - (w0^T w0 + w1^T w1 + v^T v)/(2*tau^2)
            #             + 1.5 log(2) - 1.5 log(pi) - (tau^2+sigma0^2+sigma1^2)/2
            log_p_neg_z = (
                -3 * jnp.log(2 * jnp.pi)
                - 6 * jnp.log(tau)
                - (jnp.dot(w0, w0) + jnp.dot(w1, w1) + jnp.dot(v, v)) / (2 * tau**2)
                + 1.5 * jnp.log(2)
                - 1.5 * jnp.log(jnp.pi)
                - (tau**2 + sigma0**2 + sigma1**2) / 2
```

```
    )

    # ---------- Likelihood term ----------
    # Compute predictions:
    # For each observation x_n (a row vector of shape (2,)), we compute:
    #   m1 = w1^T x_n, m0 = w0^T x_n, and v_dot = v^T x_n
    m1 = jnp.sum(x * w1, axis=1)
    m0 = jnp.sum(x * w0, axis=1)
    v_dot = jnp.sum(x * v, axis=1)

    # For each observation, form the likelihood contribution (a mixture) as:
    # term1 = N(y_n | m1, sigma1^2) * sigmoid(v_dot)
    # term2 = N(y_n | m0, sigma0^2) * sigmoid(-v_dot)
    term1 = jnp.exp(log_npdf(y, m1, sigma1**2)) * sigmoid(v_dot)
    term2 = jnp.exp(log_npdf(y, m0, sigma0**2)) * sigmoid(-v_dot)

    # Sum of both (for each observation) and then take the log and sum over all data points.
    log_likelihood = jnp.sum(jnp.log(term1 + term2))



    return log_p_neg_z + log_likelihood
```

Task 2.3: Metropolis-Hasting sampler to infer all parameters

Below is the code for the Metropolis-Hasting sampler, 'MetH'. Since the we have a mixed Gaussian as target distribution, we use unimodal Gaussians as proposal distributions for the parameters.

In [ ]:
```
def MetH(log_target, num_params, tau, num_iter, theta_init=None, seed=0):
    """Runs a Metropolis-Hastings sampler

    Arguments:
    log_target:         function for evaluating the log target distribution, i.e. log \tilde{p}(theta). The function expe
    num_params:         number of parameters of the joint distribution (integer)
    tau:                standard deviation of the Gaussian proposal distribution (positive real)
    num_iter:           number of iterations (integer)
    theta_init:         vector of initial parameters (np.array with shape (num_params) or None)
    seed:               seed (integer)

    returns
    thetas              np.array with MCMC samples (np.array with shape (num_iter+1, num_params))
    """

    # set initial key
    key = random.PRNGKey(seed)

    if theta_init is None:
        theta_init = jnp.zeros((num_params))

    # prepare lists
    thetas = [theta_init]
    accepts = []
    log_p_theta = log_target(theta_init)

    for k in range(num_iter):

        # update keys: key_proposal for sampling proposal distribution and key_accept for deciding whether to accept or r
        key, key_proposal, key_accept = random.split(key, num=3)

        # get the last value for theta and generate new proposal candidate
        theta_cur = thetas[-1]
        theta_star = theta_cur + tau * random.normal(key_proposal, shape=(num_params,))

        # evaluate the log density for the candidate sample
        log_p_theta_star = log_target(theta_star)

        # compute acceptance probability
        log_r = log_p_theta_star - log_p_theta
        A = min(1, jnp.exp(log_r))

        # accept new candidate with probability A
        if random.uniform(key_accept) < A:
            theta_next = theta_star
            log_p_theta = log_p_theta_star
            accepts.append(1)
        else:
            theta_next = theta_cur
            accepts.append(0)
        thetas.append(theta_next)

    print("Acceptance ratio: %3.2f" % jnp.mean(jnp.array(accepts)))

    # return as np.array
    thetas = jnp.stack(thetas)

    # check dimensions and return
    assert thetas.shape == (
        num_iter + 1,
        num_params,
    ), f"The shape of thetas was expected to be ({num_iter+1}, {num_params}), but the actual shape was {thetas.shape}. Pl
    return thetas
```

Looking at the data and the derived log joint distribution, we expect $z_n = 1$ to be the first part with a positive slope and $z_n = 0$ to be the second part with a negative slope. Because of this convention, we expect the slope parameter of $\boldsymbol{v}$ to be negative. We also notice the variance for the data points in region $z_n = 0$ is much larger than in the in region $z_n = 1$. With good initial guesses for the parameters, we might have a good chance of being close to convergence (for relatively few iterations), when picking a low proposal variance to get a high accpetance ratio.

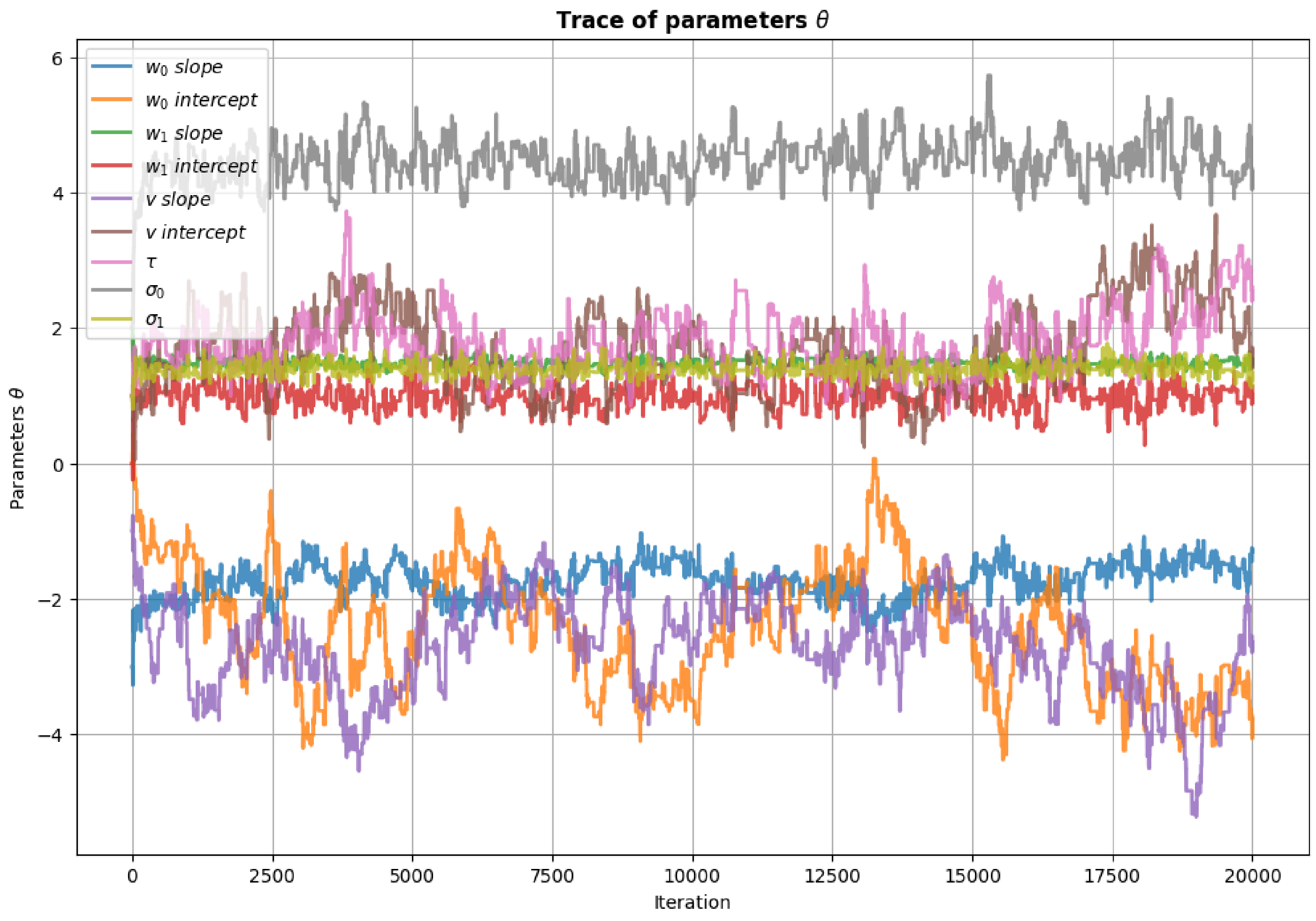
Hence, we guess for the parameters to be near:

$$\theta = \left[\boldsymbol{w}_0 = \begin{pmatrix}-3\\0\end{pmatrix},\ \boldsymbol{w}_1 = \begin{pmatrix}2\\0\end{pmatrix},\ \boldsymbol{v} = \begin{pmatrix}-1\\1\end{pmatrix},\ \tau = 1,\ \sigma_0 = 2,\ \sigma_1 = 1\right]$$

We set the variance of the proposal distribution to 0.2 (called step_size in code) and run the sampler for 20.000 iterations. We expect higher acceptance ratio with the low proposal variance.

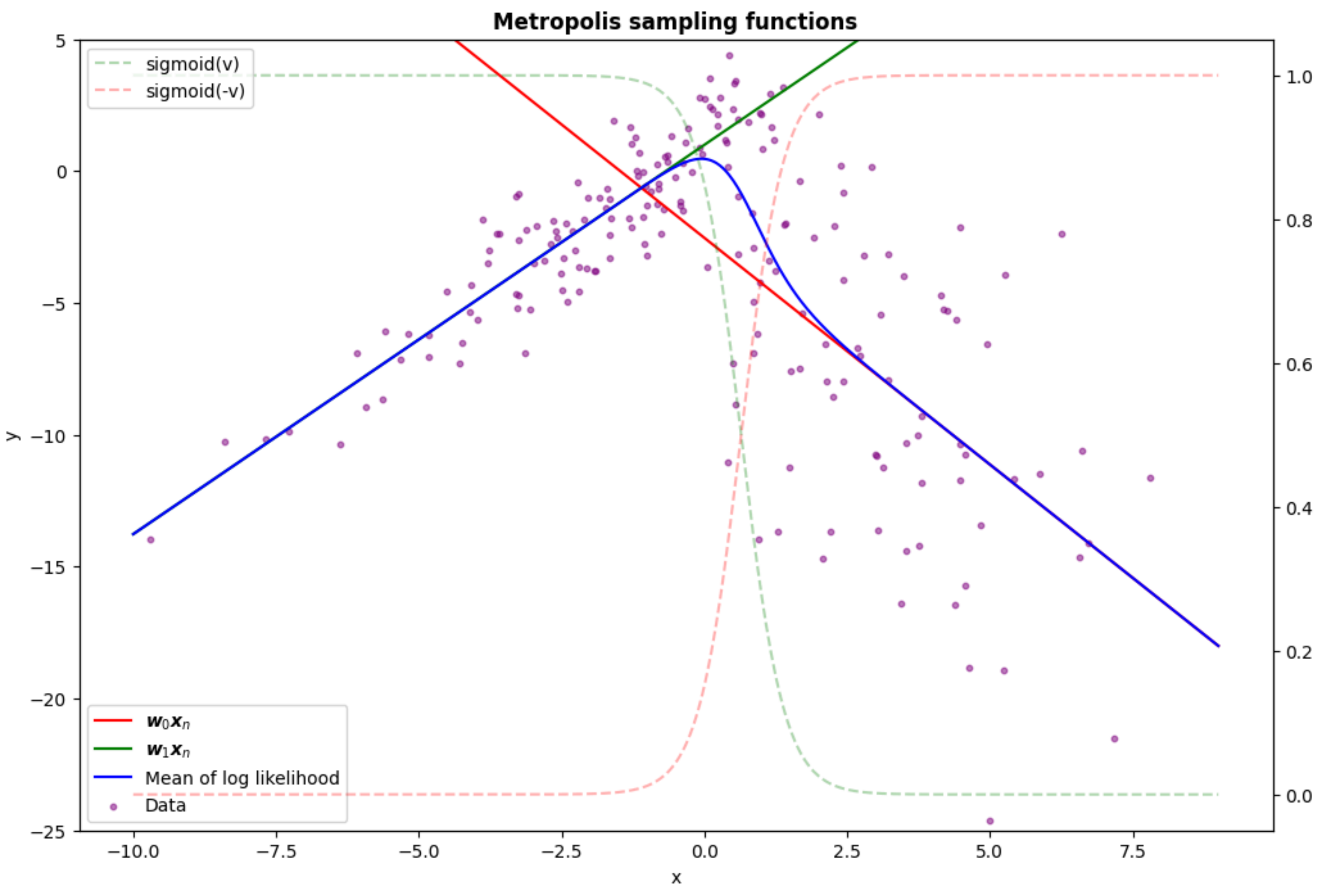
```
In [ ]: # specify number of parameters in the target distribution
num_params = 9

# target distribution = log joint distribution

num_iterations = 20000

# proposal variance for each parameter
step_size = 0.2

# theta = {w0[0], w0[1], w1[0], w1[1], v[0], v[1], tau, sigm0, sigma1}
theta_lab = [r'$w_0\ slope$', r'$w_0\ intercept$', r'$w_1\ slope$', r'$w_1\ intercept$',
             r'$v\ slope$', r'$v\ intercept$', r'$\tau$', r'$\sigma_0$', r'$\sigma_1$']

w0_init = [-3.0, 0.0]
w1_init = [2.0,0.0]
v_init = [-1.0, 1.0]
tau_init = [1.0]
sigma0_init = [2.0]
sigma1_init = [1.0]
theta_init = jnp.array(w0_init + w1_init + v_init + tau_init + sigma0_init + sigma1_init)

p_target = lambda theta: log_joint(y_data, design_matrix(x_data), theta)

# run sampler
thetas = MetH(p_target, num_params, step_size, num_iterations, theta_init=theta_init, seed=0)

# plot resutls
xs = np.linspace(-12, 12, 1000)
fig, axes = plt.subplots(1, 1, figsize=(12, 8))
for i in range(num_params):
    axes.plot(thetas[:,i], alpha=0.8, linewidth=2,label=theta_lab[i])
axes.set_xlabel('Iteration')
axes.set_ylabel('Parameters $\\theta$')
axes.set_title('Trace of parameters $\\theta$', fontweight='bold')
axes.legend()
axes.grid()
plt.show()
```

```
Acceptance ratio: 0.07
```



In [ ]:
```python
warm_up = int(0.2*num_iterations)
theta_mean = np.mean(thetas[warm_up:], axis=0)

w0 = jnp.array([theta_mean[0],theta_mean[1]])
w1 = jnp.array([theta_mean[2],theta_mean[3]])
v = jnp.array([theta_mean[4],theta_mean[5]])
tau = theta_mean[6]
sigma0 = theta_mean[7]
sigma1 = theta_mean[8]
x = np.linspace(-10, 9, 500)

# design matrix
X = design_matrix(x)
X1 = X@w1
X0 = X@w0

fig, axes = plt.subplots(1,1, figsize=(12, 8))

#### Metro-Hastings samples mean:

y = sigmoid(X@v)*X1 + sigmoid(-X@v)*X0
ax2 = axes.twinx()

ax2.plot(x,sigmoid(X@v),'--', color='green', alpha=0.3, label='sigmoid(v)')
ax2.plot(x,sigmoid(-X@v),'--', color='red', alpha=0.3, label='sigmoid(-v)')
ax2.legend(loc='upper left')

axes.plot(x, X0, color='red', label=r'$\boldsymbol{w}_0 \boldsymbol{x}_n$')
axes.plot(x, X1, color='green', label=r'$\boldsymbol{w}_1 \boldsymbol{x}_n$')
axes.plot(x, y, color='blue', label='Mean of log likelihood')

# plot contour of log joint distribution and data
axes.set_title('Metropolis sampling functions', fontweight='bold')
axes.set_xlabel('x')
axes.set_ylabel('y')

#log_joint_vectorized = vmap(lambda xi, yi: log_joint(yi, xi, theta_mean))
axes.set_ylim(-25,5)
axes.scatter(x_data, y_data, c='purple', s=10, alpha=0.5, label='Data')
axes.legend(loc='lower left')
```

```
<matplotlib.legend.Legend at 0x1bfc5a74ce0>
```



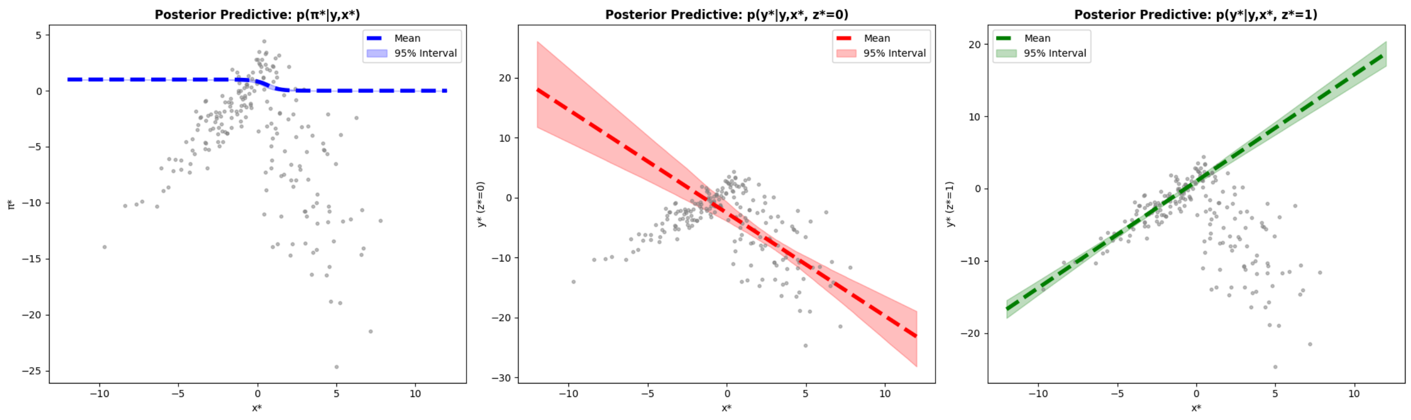
Task 2.4: posterior mean and 95% credibility intervals for all parameters

```
In [ ]: # Assuming you already have these:
        # thetas, warm_up, theta_init

        theta_sample = thetas[warm_up:, :]

        # Round to 3 decimals
        lower = jnp.round(jnp.percentile(theta_sample, 2.5, axis=0), 3)
        upper = jnp.round(jnp.percentile(theta_sample, 97.5, axis=0), 3)
        mean = jnp.round(jnp.mean(theta_sample, axis=0), 3)

        # Labels with HTML
        labels = [
            'w<sub>0</sub>: slope', 'w<sub>0</sub>: intercept',
            'w<sub>1</sub>: slope', 'w<sub>1</sub>: intercept',
            'v slope', 'v intercept',
            '&#x03C4;',                # τ
            '&#x03C3;<sub>0</sub>',  # σ₀
            '&#x03C3;<sub>1</sub>'   # σ₁
        ]

        # Format the intervals as strings: "[lower, upper]"
        low_up = [f"[{l:.3f} ; {u:.3f}]" for l, u in zip(lower, upper)]

        # Create DataFrame
        cred_int = pd.DataFrame({
            'Parameter': labels,
            'Initial': theta_init,
            'Mean': mean,
            '95%-Credibility interval': low_up,
        })

        cred_int = cred_int.set_index('Parameter')

        # Display with HTML rendering
        display(HTML(cred_int.to_html(escape=False)))
```

Parameter	Initial	Mean	95%-Credibility interval
w_0: slope	-3.0	-1.719	[-2.241 ; -1.291]
w_0: intercept	0.0	-2.528	[-3.897 ; -0.686]
w_1: slope	2.0	1.476	[1.356 ; 1.595]
w_1: intercept	0.0	0.997	[0.631 ; 1.363]
v slope	-1.0	-2.682	[-4.247 ; -1.534]
v intercept	1.0	1.727	[0.648 ; 2.991]
τ	1.0	1.821	[1.051 ; 2.970]
σ_0	2.0	4.499	[3.926 ; 5.148]
σ_1	1.0	1.396	[1.208 ; 1.595]

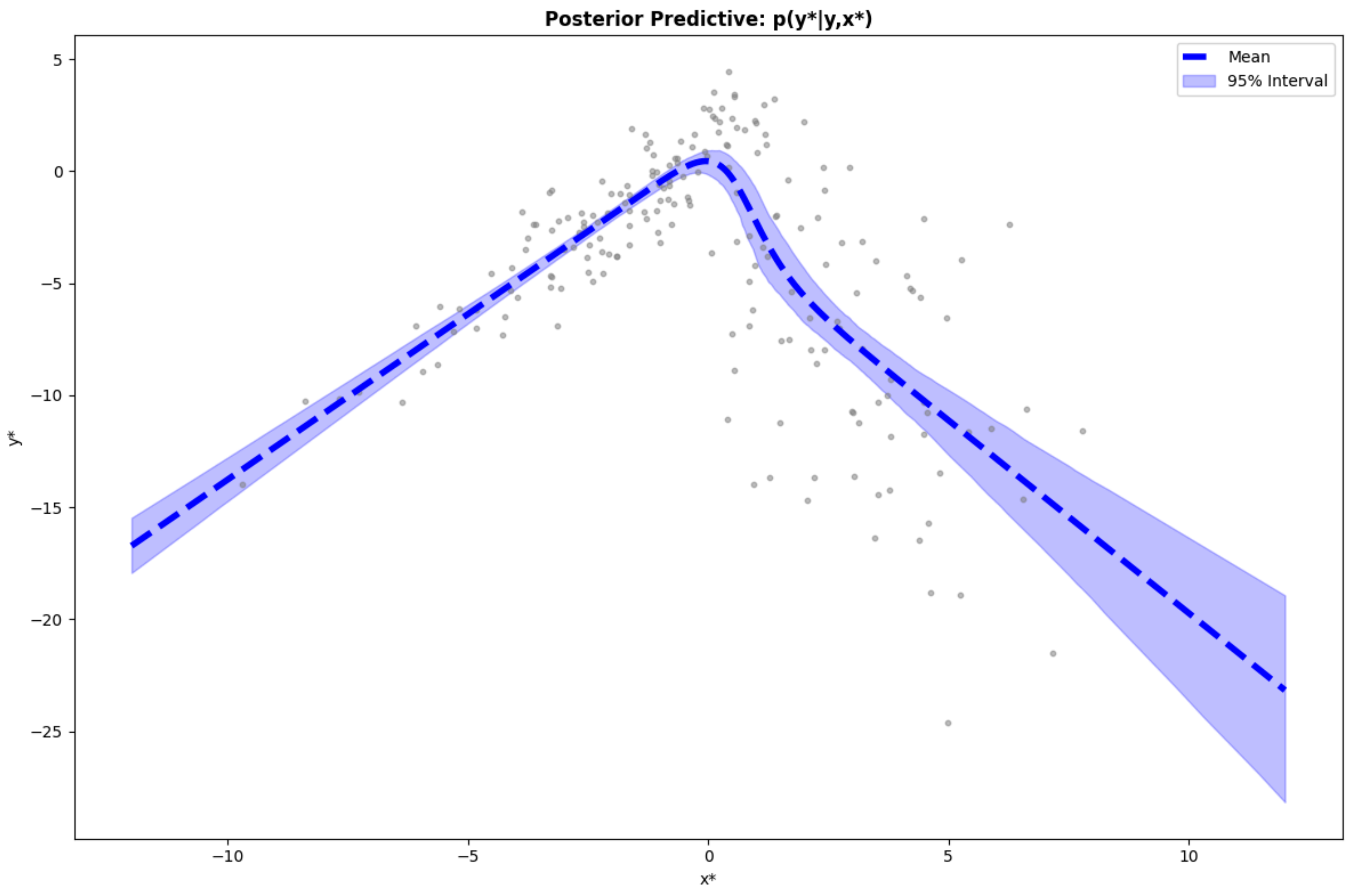
Task 2.5: Plot posterior predictive distribution for $p(\pi^* \mid \boldsymbol{y}, \boldsymbol{x}^*),\ p(y^* \mid \boldsymbol{y}, \boldsymbol{x}^*, z^* = 0)$ and $p(y^* \mid \boldsymbol{y}, \boldsymbol{x}^*, z^* = 1)$

In [ ]:
```python
# --------------------------
# Compute posterior predictive samples
# --------------------------
# Create new x* values
x_star = jnp.linspace(-12, 12, 1000)
X_star = design_matrix(x_star)  # design matrix (1000, 2)

# Assume theta_sample is available from your MCMC sampler (shape: (M, 9))
# Extract the relevant posterior samples:
#   w0_samples: parameters for branch z=0, shape (M, 2)
#   w1_samples: parameters for branch z=1, shape (M, 2)
#   v_samples : parameters for mixing, shape (M, 2)
w0_samples = theta_sample[:, 0:2]
w1_samples = theta_sample[:, 2:4]
v_samples = theta_sample[:, 4:6]

# For each sample we compute the predictive functions:
#   π* = sigmoid(X_star @ v)
#   y*|z*=0 = X_star @ w0
#   y*|z*=1 = X_star @ w1

# Compute π* samples: iterate over M samples to get an array of shape (M, len(x_star))
pi_star_samples = jnp.array(
    [sigmoid(X_star @ v_samples[i]) for i in range(v_samples.shape[0])]
)
# Compute y* samples for branch z*=0
y_star_z0_samples = jnp.array(
    [X_star @ w0_samples[i] for i in range(w0_samples.shape[0])]
)
# Compute y* samples for branch z*=1
y_star_z1_samples = jnp.array(
    [X_star @ w1_samples[i] for i in range(w1_samples.shape[0])]
)

# --------------------------
# Plot predictions on top of the data
# --------------------------
fig, axes = plt.subplots(1, 3, figsize=(20, 6))

# Plot posterior predictive for π*
plot_summary(
    axes[0],
    x_star,
    pi_star_samples,
    title="Posterior Predictive: p(π*|y,x*)",
    color="blue",
)
axes[0].set_xlabel("x*")
axes[0].set_ylabel("π*")
axes[0].scatter(
    np.array(x_data), np.array(y_data), color="grey", s=10, alpha=0.5, label="Data"
)

# Plot posterior predictive for y* with z* = 0
plot_summary(
    axes[1],
    x_star,
    y_star_z0_samples,
    title="Posterior Predictive: p(y*|y,x*, z*=0)",
    color="red",
)
axes[1].set_xlabel("x*")
axes[1].set_ylabel("y* (z*=0)")
axes[1].scatter(
    np.array(x_data), np.array(y_data), color="grey", s=10, alpha=0.5, label="Data"
)

# Plot posterior predictive for y* with z* = 1
plot_summary(
    axes[2],
    x_star,
    y_star_z1_samples,
    title="Posterior Predictive: p(y*|y,x*, z*=1)",
    color="green",
)
axes[2].set_xlabel("x*")
axes[2].set_ylabel("y* (z*=1)")
axes[2].scatter(
    np.array(x_data), np.array(y_data), color="grey", s=10, alpha=0.5, label="Data"
)

plt.tight_layout()
plt.show()
```



Task 2.6: Plot posterior predictive distribution for $p(y^* \mid \boldsymbol{y}, \boldsymbol{x}^*)$

In [ ]:
```python
# Create new x* values
x_star = jnp.linspace(-12, 12, 1000)
X_star = design_matrix(x_star)  # design matrix (1000, 2)

# Extract posterior samples from theta_sample (shape: (M,9))
w0_samples = theta_sample[:, 0:2]
w1_samples = theta_sample[:, 2:4]
v_samples  = theta_sample[:, 4:6]

# Compute predictive y* for each sample:
# y* = sigmoid(X_star @ v) * (X_star @ w1) + (1 - sigmoid(X_star @ v)) * (X_star @ w0)
y_star_pred_samples = jnp.array([
    sigmoid(X_star @ v_samples[i]) * (X_star @ w1_samples[i]) +
    (1 - sigmoid(X_star @ v_samples[i])) * (X_star @ w0_samples[i])
    for i in range(theta_sample.shape[0])
])

# Plot using plot_summary on top of the data
fig, ax = plt.subplots(1, 1, figsize=(12, 8))
plot_summary(ax, x_star, y_star_pred_samples, title="Posterior Predictive: p(y*|y,x*)", color="blue")
ax.set_xlabel("x*")
ax.set_ylabel("y*")
ax.scatter(np.array(x_data), np.array(y_data), color="grey", s=10, alpha=0.5, label="Data")
plt.tight_layout()
plt.show()
```


Posterior Predictive: p(y*|y,x*)
Mean
95% Interval
5
0
−5
−10
−15
−20
−25
y*
−10
−5
0
5
10
x*