

Gaussian Approximation Credible Interval

Question 3.3

Use the Gaussian approximation to compute a 90% posterior credibility interval for w_0 .

Given a following task, how would I think about approach it? What is it that I am asked to do?

When you're asked to "use the Gaussian approximation to compute a 90% posterior credibility interval for w_0 ," here's how to break down the task:

1. Identify the posterior mode (the MAP estimate)

- You'll have a joint posterior $p(\mathbf{w} \mid \text{data})$. Under the Laplace (Gaussian) approximation, you first find its maximum:

$$\hat{\mathbf{w}} = \arg \max_{\mathbf{w}} p(\mathbf{w} \mid \text{data}) \quad .$$

- In practice this usually means maximizing the log-posterior (often by setting gradient = 0).

2. Compute the curvature at the mode (the Hessian)

- Evaluate the negative Hessian matrix of the log-posterior at $\hat{\mathbf{w}}$:

$$H = -\nabla_{\mathbf{w}}^2 [\log p(\mathbf{w} \mid \text{data})] \Big|_{\mathbf{w}=\hat{\mathbf{w}}} .$$

- Under the Laplace approximation, the posterior is approximated by

$$p(\mathbf{w} \mid \text{data}) \approx \mathcal{N}(\hat{\mathbf{w}}, H^{-1}).$$

3. Extract the marginal for w_0

- From the full covariance H^{-1} , read off the variance element $\sigma_{w_0}^2 = [H^{-1}]_{00}$.
- The approximate marginal is

$$w_0 \mid \text{data} \approx \mathcal{N}(\hat{w}_0, \sigma_{w_0}^2).$$

4. Form the 90% credible interval

- A 90% symmetric Gaussian interval uses the 5th and 95th percentiles of the normal.
- Let $z_{0.95}$ be the 0.95-quantile of a standard Normal (≈ 1.645). Then

$$[\hat{w}_0 - z_{0.95} \sigma_{w_0}, \quad \hat{w}_0 + z_{0.95} \sigma_{w_0}]$$

is your 90% posterior credibility interval.

What you're being asked to do

- **"Gaussian approximation"** \Rightarrow use Laplace's method: approximate the posterior by a Normal at its mode.

- **“Compute a 90% posterior credibility interval”** \Rightarrow find the interval that contains 90% of that Normal’s mass for the parameter w_0 .
- **Key steps:**
 1. Find the MAP estimate \hat{w}_0 .
 2. Compute the (approximate) posterior variance for w_0 .
 3. Use the $\pm 1.645 \sigma$ rule to get the bound.

If you’ve already computed the MAP \hat{w} and the Hessian (or its inverse), you just plug those numbers into the formula above and you’re done.

Can you give me the code for this in jax?

Below is a self-contained JAX snippet that:

1. Defines a generic log-posterior $\log p(w \mid \text{data}) = \log p(\text{data} \mid w) + \log p(w)$
2. Finds the MAP \hat{w} by a few steps of Newton’s method using JAX’s autodiff (you can of course swap in your favorite optimizer)
3. Computes the negative Hessian at the mode, inverts it, and reads off the marginal variance of w_0
4. Forms the symmetric 90% Gaussian interval for w_0

python

```
`import jax
import jax.numpy as jnp

# -----
# 1. Define your model-specific pieces here:
# -----

def log_prior(w, sigma2=1.0):
    # zero-mean Gaussian prior:  $p(w) \propto \exp(-\frac{1}{2} w^T w / \sigma^2)$ 
    return -0.5 * jnp.dot(w, w) / sigma2

def log_likelihood(w, X, y):
    # example: logistic regression likelihood
    #  $p(y|X,w) = \prod \text{sigmoid}(w^T x_i)^{y_i} (1 - \text{sigmoid}(w^T x_i))^{(1-y_i)}$ 
    logits = X @ w
    return jnp.sum(y * logits - jnp.logaddexp(0.0, logits))

def log_posterior(w, X, y, sigma2=1.0):
    return log_likelihood(w, X, y) + log_prior(w, sigma2)

# -----
# 2. Build gradient and Hessian functions via autodiff
# -----

# gradient of the log-posterior
grad_logpost = jax.grad(lambda w, X, y, sigma2: log_posterior(w, X, y, sigma2))

# Hessian of the log-posterior
hess_logpost = jax.hessian(lambda w, X, y, sigma2: log_posterior(w, X, y, sigma2))

# -----
# 3. Find the MAP mode by a few Newton steps
# -----

def find_map_newton(w_init, X, y, sigma2=1.0, n_steps=5):
    w = w_init
```

```

for i in range(n_steps):
    g = grad_logpost(w, X, y, sigma2)          # ∇ log p(w|data)
    H = hess_logpost(w, X, y, sigma2)         # ∇² log p(w|data)
    # Newton update: w ← w - H⁻¹ g
    w = w - jnp.linalg.solve(H, g)
return w

# -----
# 4. Putting it all together
# -----

def credible_interval_w0(X, y, sigma2=1.0, n_newton=5, ci_prob=0.90):
    D = X.shape[1]
    # initialize at zero (or e.g. mle)
    w_init = jnp.zeros(D)

    # 4.1 find MAP
    w_map = find_map_newton(w_init, X, y, sigma2, n_steps=n_newton)

    # 4.2 compute negative Hessian at the mode
    H_map = -hess_logpost(w_map, X, y, sigma2)

    # 4.3 invert to get covariance
    cov_map = jnp.linalg.inv(H_map)

    # 4.4 extract sd of w0
    sigma_w0 = jnp.sqrt(cov_map[0, 0])

    # 4.5 z-value for two-sided ci
    alpha = 1.0 - ci_prob
    z = jax.scipy.stats.norm.ppf(1.0 - alpha/2)

    # 4.6 build interval
    lower = w_map[0] - z * sigma_w0
    upper = w_map[0] + z * sigma_w0

    return float(lower), float(upper), float(w_map[0]), float(sigma_w0)

# -----
# Example usage:
# -----

if __name__ == "__main__":
    # toy data
    import numpy as onp
    onp.random.seed(0)
    N, D = 200, 3
    X = onp.random.randn(N, D)
    true_w = onp.array([1.0, -2.0, 0.5])
    logits = X @ true_w
    probs = 1 / (1 + onp.exp(-logits))
    y = onp.random.binomial(1, probs, size=N)

    # convert to jax arrays
    X_j, y_j = jnp.array(X), jnp.array(y)

    # compute
    lower, upper, w0_map, sd_w0 = credible_interval_w0(X_j, y_j,
                                                       sigma2=10.0,
                                                       n_newton=10,
                                                       ci_prob=0.90)

    print(f"MAP w0      = {w0_map:.3f}")
    print(f"Posterior sd= {sd_w0:.3f}")
    print(f"90% CI       = [{lower:.3f}, {upper:.3f}]")

```

Notes on adapting to your problem:

- Replace `log_likelihood` with whatever your data model is (e.g. Gaussian regression, Poisson, etc.).

- Adjust `log_prior` to your true prior (and its hyperparameters).
- You can swap Newton's method for any optimizer you prefer (e.g. gradient descent, L-BFGS from [jaxopt](#), etc.).
- The key steps remain the same:
 1. locate the posterior mode
 2. compute the Hessian there
 3. approximate the posterior by $\mathcal{N}(\hat{w}, H^{-1})$
 4. read off σ_{w_0} and use $z_{0.95} \approx 1.645$ for a 90% interval.

How do I compute the hessian in general? Given any likelihood, how would just prepare it for plugging and playing?

At a high level, the Hessian you need is just the matrix of second-derivatives of your log-posterior (or log-likelihood if you're doing a pure MLE approximation) with respect to the parameter vector. In JAX (and most modern autodiff frameworks) you don't have to derive those by hand—you just tell JAX “here's my scalar function of \mathbf{w} , now give me its Hessian.”

Here's a minimal “plug-and-play” recipe:

```
python

`import jax
import jax.numpy as jnp

# -----
# 1) Define your log-likelihood and log-prior as functions of w (and data)
# -----

def log_likelihood(w, *data_args):
    # e.g. Gaussian regression:
    X, y, sigma2 = data_args
    preds = X @ w
    return -0.5 * jnp.sum((y - preds)**2) / sigma2

def log_prior(w, *prior_args):
    # e.g. zero-mean Gaussian prior with variance tau2
    tau2, = prior_args
    return -0.5 * jnp.dot(w, w) / tau2

def log_posterior(w, data_args, prior_args):
    return log_likelihood(w, *data_args) + log_prior(w, *prior_args)

# -----
# 2) Build gradient and Hessian functions once
# -----

# gradient of log-posterior wrt w
grad_logpost = jax.jit( jax.grad(lambda w, d_args, p_args:
                                log_posterior(w, d_args, p_args)) )

# Hessian of log-posterior wrt w
hess_logpost = jax.jit( jax.hessian(lambda w, d_args, p_args:
                                   log_posterior(w, d_args, p_args)) )

# Now grad_logpost(w, data_args, prior_args) returns ∇(w) log p
# hess_logpost(w, data_args, prior_args) returns ∇²(w) log p

# -----
# 3) At the MAP ŵ, compute the (negative) Hessian and invert
# -----
```

```
w_map = ... # assume you've already found your MAP via any optimizer
H = -hess_logpost(w_map, data_args, prior_args) # negative Hessian
cov_approx = jnp.linalg.inv(H) # Gaussian approx covariance

#  $\sigma^2_{\{w_0\}}$  = cov_approx[0,0], etc.
```

What this does for you

- `jax.grad(fn)` gives you a function that computes the gradient vector of `fn(w)`
- `jax.hessian(fn)` gives you a function that computes the full matrix of second-derivatives
- You only need to wrap your *scalar* log-posterior (or log-likelihood) in that API, and JAX handles the rest.

If you ever want to mix in hyperparameters or extra arguments, just bundle them into tuples (`data_args`, `prior_args`) so that your primary function signature is always `fn(w, data_args, prior_args)`.

A few extra tips

- **Efficiency:** For really large models you may prefer a **Gauss-Newton** or **Fisher-information** approximation (i.e. replace the Hessian with an expected curvature) instead of the full Hessian.
- **Diagnostics:** It's often worth checking that your Hessian at the mode is positive-definite (all eigenvalues > 0); negative directions mean your MAP isn't actually a local maximum.
- **Alternatives:** If you're not using JAX, most other autodiff frameworks (PyTorch, TensorFlow, Autograd) have a similar two-step pattern: first get `.grad()`, then take gradients of gradients for the Hessian (or just call their built-in Hessian helpers).

With this in place you can drop in *any* log-likelihood/prior you like and immediately get your curvature matrix for the Laplace (Gaussian) approximation.