Q1 (A) Full Conditional Distributions

First, we write out the joint distribution $f(X, P, n) \propto f(X, P|n) \cdot \pi(n)$.

Given $n \sim \text{Poisson}(\lambda)$, the prior is $\pi(n) = \frac{e^{-\lambda}\lambda^n}{n!}$.

The conditional is

$f(X, P|n) \propto \binom{n}{X} P^{X+\alpha-1}(1-P)^{n-X+\beta-1} = \frac{n!}{X!(n-X)!} P^{X+\alpha-1}(1-P)^{n-X+\beta-1}$.

Multiplying these, the $n!$ cancels out, leaving the joint distribution:

$$f(X, P, n) \propto \frac{1}{X!(n-X)!} P^{X+\alpha-1}(1-P)^{n-X+\beta-1}\lambda^n$$

To find the full conditionals, we isolate the terms for X, P, n.

Full Conditional for P:

Isolating P:

$$f(P|X, n) \propto P^{(X+\alpha)-1}(1-P)^{(n-X+\beta)-1}$$

This is the Beta distribution:

$$P|X, n \sim \text{Beta}(X+\alpha, n-X+\beta)$$

Full Conditional for X:

Isolating X:

$$f(X|P, n) \propto \frac{1}{X!(n-X)!} P^X (1-P)^{-X}$$

Because we are conditioning on P and n, any terms that do not contain X are treated as constants. We can multiply the right side by $n!$ and $(1-P)^n$ without changing the proportionality:

$$f(X|P, n) \propto \left[\frac{n!}{X!(n-X)!}\right] P^X \left[(1-P)^{-X}(1-P)^n\right]$$

$$f(X|P, n) \propto \binom{n}{X} P^X (1-P)^{n-X}$$

This is the Binomial distribution:

$$X|P, n \sim \text{Binomial}(n, P)$$

3. Full Conditional for n:

Isolating terms involving n:

$$f(n|X, P) \propto \frac{1}{(n-X)!}(1-P)^{n-X}\lambda^n$$

Now we rewrite λ^n as: $\lambda^n = \lambda^{n-X} \cdot \lambda^X$.

The term λ^X is a constant with respect to n and gets absorbed into the proportionality constant:

$$f(n|X, P) \propto \frac{1}{(n-X)!}(1-P)^{n-X}(\lambda^{n-X} \cdot \lambda^X) \propto \frac{1}{(n-X)!}(1-P)^{n-X}\lambda^{n-X}$$

Combining the terms with the same exponent:

$$f(n|X, P) \propto \frac{(\lambda(1-P))^{n-X}}{(n-X)!}$$

Let $m = n - X$. The kernel becomes $\frac{(\lambda(1-P))^m}{m!}$, which is the exact shape of a Poisson distribution for m.

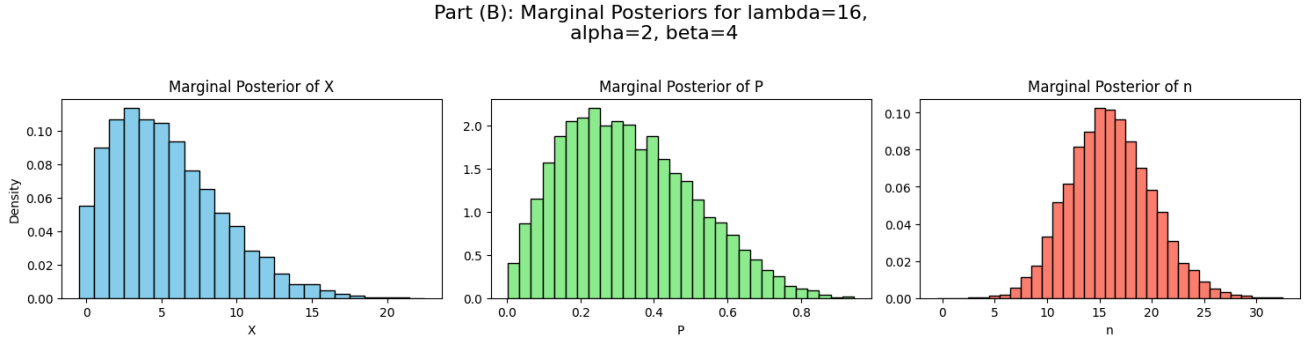
$$n = X + m \quad \text{where} \quad m \sim \text{Poisson}(\lambda(1-P))$$

Q1 (B) Gibbs Sampling Implementation

```
import numpy as np
import matplotlib.pyplot as plt
```

```
def run_gibbs_sampler(lam, alpha, beta, num_iterations=10000, burn_in=1000):
    n = int(lam)
    P = 0.5
    X = np.random.binomial(n, P)
    samples_X = np.zeros(num_iterations)
    samples_P = np.zeros(num_iterations)
    samples_n = np.zeros(num_iterations)

    # Gibbs Sampling Loop
    for i in range(num_iterations):
        # Update P | X, n ~ Beta(X + alpha, n - X + beta)
        P = np.random.beta(X + alpha, n - X + beta)

        # Update X | P, n ~ Binomial(n, P)
        X = np.random.binomial(n, P)

        # Update n | X, P ~ X + Poisson(lam * (1 - P))
        m = np.random.poisson(lam * (1 - P))
        n = X + m

        # Store the samples
        samples_X[i] = X
        samples_P[i] = P
        samples_n[i] = n

    return samples_X[burn_in:], samples_P[burn_in:], samples_n[burn_in:]

# --- Part (B) ---
# Set the given parameters
lam_B = 16
alpha_val = 2
beta_val = 4

# Run the sampler
X_samples, P_samples, n_samples = run_gibbs_sampler(lam_B, alpha_val, beta_val)

# Plotting the marginal histograms
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
figtitle=f"""Part (B): Marginal Posteriors for lambda={lam_B},
alpha={alpha_val}, beta={beta_val}"""

fig.suptitle(figtitle,
             fontsize=16)

# Histogram for X
axes[0].hist(X_samples,
             bins=range(int(max(X_samples))+2),
             density=True,
             color='skyblue',
             edgecolor='black',
             align='left')
axes[0].set_title('Marginal Posterior of X')
axes[0].set_xlabel('X')
axes[0].set_ylabel('Density')

# Histogram for P
axes[1].hist(P_samples, bins=30, density=True,
             color='lightgreen',
             edgecolor='black')
axes[1].set_title('Marginal Posterior of P')
axes[1].set_xlabel('P')

# Histogram for n
axes[2].hist(n_samples,
             bins=range(int(max(n_samples))+2),
             density=True, color='salmon',
             edgecolor='black',
             align='left')
axes[2].set_title('Marginal Posterior of n')
axes[2].set_xlabel('n')

plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```

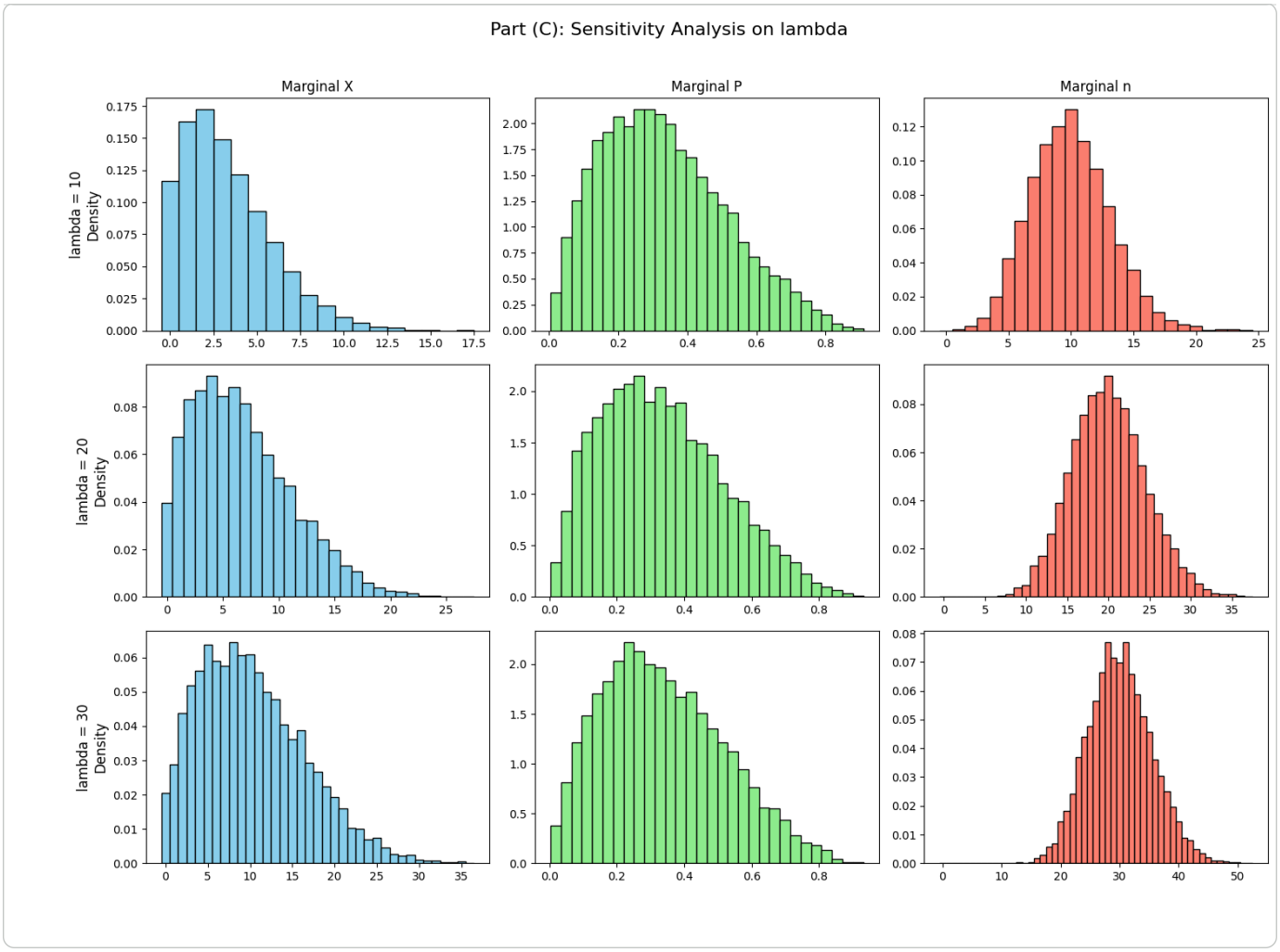
```
# posterior means
print(f"Posterior Mean of X: {np.mean(X_samples):.2f}")
print(f"Posterior Mean of P: {np.mean(P_samples):.2f}")
print(f"Posterior Mean of n: {np.mean(n_samples):.2f}")
```




```
Posterior Mean of X: 5.31
Posterior Mean of P: 0.33
Posterior Mean of n: 16.06
```

Q1 (C) Sensitivity of Lambda

```
# --- Part (C) ---
lambdas_to_test = [10, 20, 30]

fig, axes = plt.subplots(len(lambdas_to_test), 3,
                         figsize=(15, 3.5 * len(lambdas_to_test)))
fig.suptitle('Part (C): Sensitivity Analysis on lambda',
             fontsize=16, y=1.02)

for i, lam_test in enumerate(lambdas_to_test):
    # Run the sampler for the current lambda
    X_test, P_test, n_test = run_gibbs_sampler(lam_test, alpha_val, beta_val)

    # Plot X
    axes[i, 0].hist(X_test, bins=range(int(max(X_test))+2),
                    density=True, color='skyblue',
                    edgecolor='black', align='left')
    axes[i, 0].set_ylabel(f'lambda = {lam_test}\nDensity', fontsize=12)
    if i == 0: axes[i, 0].set_title('Marginal X')

    # Plot P
    axes[i, 1].hist(P_test, bins=30, density=True,
                    color='lightgreen', edgecolor='black')
    if i == 0: axes[i, 1].set_title('Marginal P')

    # Plot n
    axes[i, 2].hist(n_test, bins=range(int(max(n_test))+2),
                    density=True, color='salmon',
                    edgecolor='black', align='left')
    if i == 0: axes[i, 2].set_title('Marginal n')

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



Conclusion: Sensitivity Analysis and Prior Influence

The marginal histograms generated in Part (C) demonstrate that this Bayesian model is highly sensitive to the choice of the prior parameter λ. As λ increases from 10 to 30, the entire joint distribution shifts dramatically to the right.

Q1 (D) Variational Bayes

To derive a Variational Bayes (VB) algorithm, we use the mean-field approximation:

$$q(X, P, n) = q(X)q(P)q(n)$$

The optimal distribution $q^*(Z_j)$ for any parameter is found by taking the expectation of the log-joint distribution with respect to all other parameters:

$$\ln q^*(Z_j) = E_{i \neq j}[\ln f(X, P, n)] + C$$

From Part (A), the log-joint distribution is:

$$\ln f(X, P, n) = -\ln(X!) - \ln((n - X)!) + (X + \alpha - 1)\ln P + (n - X + \beta - 1)\ln(1 - P) + n \ln \lambda + C$$

Deriving the update for $q(P)$

Isolating the terms containing P and take the expectation with respect to $q(X)$ and $q(n)$:

$$\ln q^*(P) = E_{X,n}[(X + \alpha - 1)\ln P + (n - X + \beta - 1)\ln(1 - P)] + C$$

$$\ln q^*(P) = (E[X] + \alpha - 1)\ln P + (E[n] - E[X] + \beta - 1)\ln(1 - P) + C$$

This matches the log-kernel of a Beta distribution. Therefore:

$$q^*(P) \sim \text{Beta}(\alpha^*, \beta^*)$$

Where $\alpha^* = E[X] + \alpha$ and $\beta^* = E[n] - E[X] + \beta$.

Updates for $q(X)$ and $q(n)$

When we attempt to derive the optimal updates for X and n, we isolate their respective terms and take the expectation under n, P and X, P respectively:

For $q(X)$:

$\ln q^*(X) = -\ln(X!) - E_n[\ln((n-X)!)] + X(E[\ln P] - E[\ln(1-P)]) + C$

For $q(n)$:

$\ln q^*(n) = -E_X[\ln((n-X)!)] + n(E[\ln(1-P)] + \ln\lambda) + C$

The term $E[-\ln((n-X)!)]$ is intractable and is a challenge. Because the expected value operator cannot be pushed inside a non-linear function (a logarithm wrapped around a factorial) where the two random variables are coupled together, calculating this exact mathematical expectation is intractable.

To bypass this intractability, we could apply a **zeroth-order Taylor series expansion** around the expected values.

By substituting the coupled random variables within the factorial with their expected values—such as assuming $(n-X)! \approx (E[n]-X)!$ for the X update, and $(n-X)! \approx (n-E[X])!$ for the n update.

↳ 1 cell hidden

Q2 Metropolis–Hastings algorithm

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
from statsmodels.graphics.tsaplots import plot_acf
from IPython.display import display, Markdown

# Data
w = np.array([1.6907, 1.7242, 1.7552, 1.7842, 1.8113, 1.8369, 1.8610, 1.8839])
y = np.array([6, 13, 18, 28, 52, 53, 61, 60])
n = np.array([59, 60, 62, 56, 63, 59, 62, 60])

# Hyperparameters
a0, b0 = 0.25, 4.0
c0, d0 = 2.0, 10.0
e0, f0 = 2.0, 1000.0

def log_posterior(theta):
    theta1, theta2, theta3 = theta
    mu = theta1
    sigma = np.exp(theta2)
    m1 = np.exp(theta3)

    x = (w - mu) / sigma

    # Prevent overflow and log(0) errors
    x = np.clip(x, -500, 500)
    exp_x = np.exp(x)
    h_w = (exp_x / (1.0 + exp_x))**m1
    eps = 1e-10
    h_w = np.clip(h_w, eps, 1.0 - eps)

    # Log-Likelihood
    log_lik = np.sum(y * np.log(h_w) + (n - y) * np.log(1.0 - h_w))

    # Log-Prior (from the Reparameterization formula)
    log_prior = (a0 * theta3) - (2 * e0 * theta2) - (0.5 * ((theta1 - c0)**2 / d0)) \
                - (np.exp(theta3) / b0) - (np.exp(-2 * theta2) / f0)

    return log_lik + log_prior

# Metropolis-Hastings Sampler
def mh_sampler(n_iter, init_theta, cov_matrix):
    theta_chain = np.zeros((n_iter, 3))
    theta_chain[0] = init_theta
    accepted = 0
```

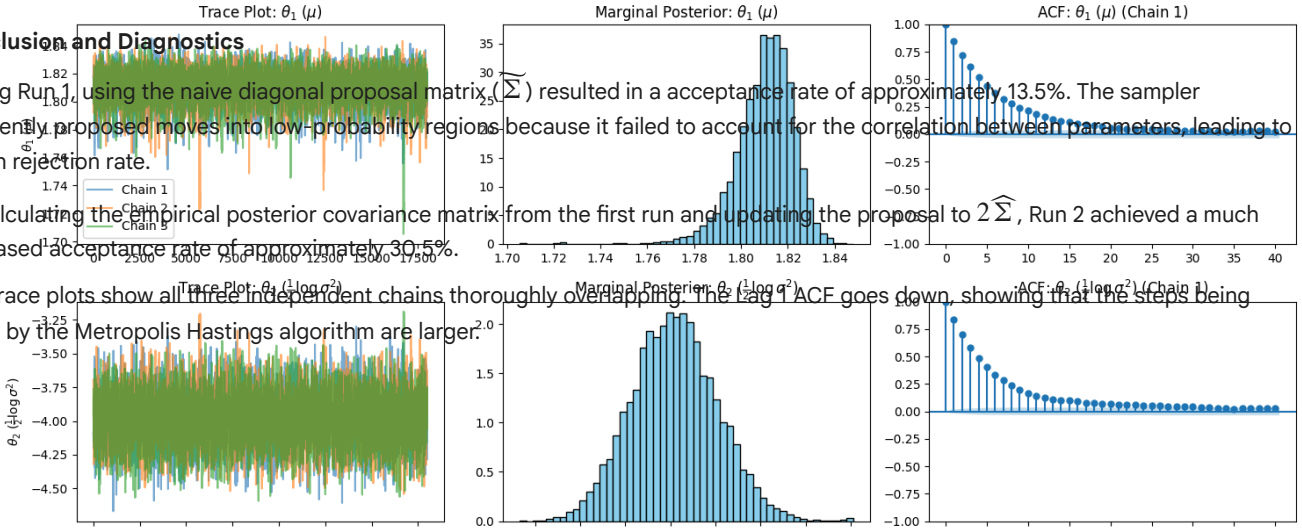
```
    log_p_current = log_posterior(init_theta)

    for t in range(1, n_iter):
        # Propose new step from Multivariate Normal
        theta_prop = np.random.multivariate_normal(theta_chain[t-1], cov_matrix)
        log_p_prop = log_posterior(theta_prop)

        # Acceptance log-odds ratio
        log_r = log_p_prop - log_p_current

        # Accept or reject
        if np.log(np.random.rand()) < log_r:
            theta_chain[t] = theta_prop
            log_p_current = log_p_prop
            accepted += 1
        else:
            theta_chain[t] = theta_chain[t-1]

    acceptance_rate = accepted / (n_iter - 1)
    return theta_chain, acceptance_rate

np.random.seed(42)
n_iter = 20000
burn_in = 2000
n_chains = 3

# Starting covariance matrix
cov_run1 = np.diag([0.00012, 0.033, 0.10])

# Starting dispersed values for the 3 chains
initial_thetas = [
    np.array([1.7, -2.0, 0.0]),
    np.array([2.1, -1.0, 0.5]),
    np.array([1.9, -3.0, -0.5])
]

chains_run1 = []
acc_rates_run1 = []

print("--- RUN 1 ---")
for i in range(n_chains):
    chain, acc = mh_sampler(n_iter, initial_thetas[i], cov_run1)
    chains_run1.append(chain[burn_in:]) # Discard burn-in immediately
    acc_rates_run1.append(acc)
    print(f"Chain {i+1} Acceptance Rate: {acc:.4f}")

pooled_samples_run1 = np.vstack(chains_run1)
hat_Sigma = np.cov(pooled_samples_run1, rowvar=False)

# New proposal covariance
cov_run2 = 2 * hat_Sigma

chains_run2 = []
acc_rates_run2 = []

print("\n--- RUN 2 (Optimized Covariance) ---")
for i in range(n_chains):
    chain, acc = mh_sampler(n_iter, initial_thetas[i], cov_run2)
    chains_run2.append(chain[burn_in:])
    acc_rates_run2.append(acc)
    print(f"Chain {i+1} Acceptance Rate: {acc:.4f}")
```

```
--- RUN 1 ---
Chain 1 Acceptance Rate: 0.1343
Chain 2 Acceptance Rate: 0.1388
Chain 3 Acceptance Rate: 0.1389

--- RUN 2 (Optimized Covariance) ---
Chain 1 Acceptance Rate: 0.3048
Chain 2 Acceptance Rate: 0.2961
Chain 3 Acceptance Rate: 0.2915
```

```
# Diagnostics & Plotting
labels = [r'$\theta_1$ ($\mu$)', r'$\theta_2$ ($\frac{1}{2}\log\sigma^2$)', r'$\theta_3$ ($\log m_1$)']
```

```
fig, axes = plt.subplots(3, 3, figsize=(15, 10))
fig.suptitle("Diagnostics for RUN 2 (Optimized Sampler)", fontsize=16)

for j in range(3):
    # 1. Trace Plots
    for i in range(n_chains):
        axes[j, 0].plot(chains_run2[i][:, j], alpha=0.6, label=f'Chain {i+1}')
    axes[j, 0].set_ylabel(labels[j])
    axes[j, 0].set_title(f"Trace Plot: {labels[j]}")
    if j == 0: axes[j, 0].legend()

    # 2. Marginal Histograms
    pooled_j = np.hstack([chains_run2[i][:, j] for i in range(n_chains)])
    axes[j, 1].hist(pooled_j, bins=50, density=True, color='skyblue', edgecolor='black')
    axes[j, 1].set_title(f"Marginal Posterior: {labels[j]}")

    # 3. Autocorrelation (Using Chain 1 as representative)
    plot_acf(chains_run2[0][:, j], ax=axes[j, 2], lags=40, title=f"ACF: {labels[j]} (Chain 1)")

plt.tight_layout()
plt.subplots_adjust(top=0.92)
plt.show()

# Calculate Lag-1 ACF for comparison
def lag1_acf(x):
    return np.corrcoef(x[:-1], x[1:])[0, 1]

print("\n--- Lag-1 Autocorrelation Comparison (Chain 1) ---")
for j in range(3):
    acf_run1 = lag1_acf(chains_run1[0][:, j])
    acf_run2 = lag1_acf(chains_run2[0][:, j])
    output_string = f"{labels[j]}: Run 1 = **{acf_run1:.4f}** | Run 2 = **{acf_run2:.4f}**"
    display(Markdown(output_string))
```

Diagnostics for RUN 2 (Optimized Sampler)



Conclusion and Diagnostics

During Run 1, using the naive diagonal proposal matrix ($\widetilde{\Sigma}$) resulted in a acceptance rate of approximately 13.5%. The sampler frequently proposed moves into low-probability regions because it failed to account for the correlation between parameters, leading to a high rejection rate.

By calculating the empirical posterior covariance matrix from the first run and updating the proposal to $2\widehat{\Sigma}$, Run 2 achieved a much increased acceptance rate of approximately 30.5%.

The trace plots show all three independent chains thoroughly overlapping. The Lag 1 ACF goes down, showing that the steps being taken by the Metropolis Hastings algorithm are larger.