Maximum Likelihood Estimation for Generalized Linear Models

A Complete Guide to Fitting Neural Encoding Models

Table of Contents

---

1. Mathematical Framework

1.1 Generalized Linear Model (GLM)

For a single neuron, the **conditional intensity function** (instantaneous firing rate) is:

$$\lambda(t|H_t) = \exp\left(k^\top x_t + h^\top y_t + \sum_{j \neq i} l_j^\top y_t^{(j)} + \mu\right)$$

λ(t|Ht) = exp (k⊤xt + h⊤yt + j□=i∑ lj⊤yt(j) + μ)

Parameters:

- $x_t \in R^{d_s}$xt ∈ Rds: Stimulus history (past $d_s$ds time bins)
- $y_t \in \{0,1\}^{d_h}$yt ∈ {0, 1}dh: Own spike history (past $d_h$dh time bins)
- $y_t^{(j)} \in \{0,1\}^{d_c}$yt(j) ∈ {0, 1}dc: Spike history of neuron jj
- $k \in R^{d_s}$k ∈ Rds: Stimulus filter (receptive field)
- $h \in R^{d_h}$h ∈ Rdh: Post-spike filter (refractoriness, adaptation)
- $l_j \in R^{d_c}$lj ∈ Rdc: Coupling filter from neuron jj
- $\mu \in R$μ ∈ R: Baseline log-firing rate
- H_tHt: Complete history up to time tt

1.2 Compact Notation

Define the linear predictor:

$$f(t;\theta) = k^\top x_t + h^\top y_t + \sum_{j \neq i} l_j^\top y_t^{(j)} + \mu$$

f(t; θ) = k⊤xt + h⊤yt + j□=i∑ lj⊤yt(j) + μ

Then:

$$\lambda(t;\theta) = \exp(f(t;\theta))$$

λ(t; θ) = exp(f(t; θ))

where $\theta = \{k, h, \{l_j\}_{j \neq i}, \mu\}$θ = {k, h, {lj}j□=i, μ} denotes all model parameters.

1.3 Why This Model?

Advantages:

- **Exponential link**: Ensures $\lambda(t) > 0$λ(t) > 0 (firing rates are positive)
- **Log-concavity**: Optimization is convex (unique global maximum)
- **Multiplicative interactions**: Filters act as gain modulation
- **Biological interpretability**: Components map to neural mechanisms

---

2. Log-Likelihood Derivation

2.1 Point Process Likelihood

For observed spike times $\{t_1, t_2, \ldots, t_n\}$ {t1, t2, … , tn} in interval $[0, T]$[0, T], the likelihood is:

$$p(\{t_1, \dots, t_n\}|\theta) = \left[\prod_{k=1}^{n} \lambda(t_k;\theta)\right] \exp\left(-\int_0^T \lambda(t;\theta)\,dt\right)$$

p({t1 , … , tn}|θ) = [k=1∏n λ(tk ; θ)] exp (− ∫0T λ(t; θ) dt)

Intuition:

- **Product term**: Probability of spikes occurring at observed times
- **Exponential term**: Probability of no spikes occurring elsewhere

2.2 Log-Likelihood Function

Taking the logarithm:

$$L(\theta) = \log p(\{t_1, \dots, t_n\}|\theta)$$

L(θ) = log p({t1 , … , tn}|θ)

$$L(\theta) = \underbrace{\sum_{k=1}^{n} \log\lambda(t_k;\theta)}_{\text{Reward: spikes at } t_k} - \underbrace{\int_0^T \lambda(t;\theta)\,dt}_{\text{Penalty: high rates everywhere}}$$

L(θ) = Reward: spikes at tk

k=1∑n log λ(tk ; θ) − Penalty: high rates everywhere

∫0T λ(t; θ) dt

2.3 Substituting the GLM

Since $\lambda(t;\theta) = \exp(f(t;\theta))$λ(t; θ) = exp(f(t; θ)):

$$L(\theta) = \sum_{k=1}^{n} f(t_k;\theta) - \int_0^T \exp(f(t;\theta))\,dt$$

L(θ) = k=1∑n f(tk ; θ) − ∫0T exp(f(t; θ)) dt

2.4 Discrete-Time Approximation

Discretize time into bins of width ΔtΔt:

- $t_i = i \cdot \Delta t$ti = i · Δt for $i = 1, 2, \dots, N$i = 1, 2, … , N where $N = T/\Delta t$N = T/Δt
- $r_i \in \{0, 1\}$ri ∈ {0, 1} indicates spike in bin ii

The integral becomes:

$$\int_0^T \exp(f(t;\theta))\,dt \approx \sum_{i=1}^{N} \exp(f_i)\Delta t$$

∫0T exp(f(t; θ)) dt ≈ i=1∑N exp(fi)Δt

where $f_i = f(t_i;\theta)$fi = f(ti; θ).

Discrete log-likelihood:

$$L(\theta) = \sum_{i=1}^{N} r_i f_i - \Delta t \sum_{i=1}^{N} \exp(f_i)$$

L(θ) = i=1∑N rifi − Δt i=1∑N exp(fi)

---

3. Gradient and Hessian Computation

3.1 Gradient (First Derivative)

The gradient with respect to parameter θ_jθj is:

$$\frac{\partial L}{\partial \theta_j} = \sum_{i=1}^{N} r_i \frac{\partial f_i}{\partial \theta_j} - \Delta t \sum_{i=1}^{N} \exp(f_i) \frac{\partial f_i}{\partial \theta_j}$$

∂θj ∂L = i=1∑N ri∂θj ∂fi − Δt i=1∑N exp(fi)∂θj ∂fi

$$\frac{\partial L}{\partial \theta_j} = \sum_{i=1}^{N} \left(r_i - \lambda_i \Delta t\right) \frac{\partial f_i}{\partial \theta_j}$$

∂θj∂L = i=1∑N (ri − λiΔt) ∂θj∂fi

Interpretation:

- Difference between **observed spikes** ($r_i$ri) and **predicted spikes** ($\lambda_i \Delta t$λiΔt)
- Weighted by feature $\partial f_i / \partial \theta_j$∂fi/∂θj

3.2 Example: Gradient for Stimulus Filter

For stimulus filter coefficient $k_j$kj:

$$\frac{\partial f_i}{\partial k_j} = x_{i,j}$$

∂kj∂fi = xi,j

Therefore:

$$\frac{\partial L}{\partial k_j} = \sum_{i=1}^{N} r_i x_{i,j} - \Delta t \sum_{i=1}^{N} \lambda_i x_{i,j}$$

∂kj∂L = i=1∑N rixi,j − Δt i=1∑N λixi,j

$$= \underbrace{\sum_{i:\, r_i = 1} x_{i,j}}_{\text{Spike-triggered average}} - \underbrace{\Delta t \sum_{i=1}^{N} \lambda_i x_{i,j}}_{\text{Rate-weighted average}}$$

= Spike-triggered average

i:ri=1∑ xi,j − Rate-weighted average

Δt i=1∑N λixi,j

3.3 Hessian (Second Derivative)

The Hessian matrix element (i, j)(i,j) is:

$$H_{ij} = \frac{\partial^2 L}{\partial \theta_i \partial \theta_j}$$

Hij = ∂θi∂θj∂2L

$$H_{ij} = -\Delta t \sum_{t=1}^{N} \lambda_t \frac{\partial f_t}{\partial \theta_i} \frac{\partial f_t}{\partial \theta_j}$$

Hij = −Δt t=1∑N λt∂θi∂ft∂θj∂ft

Key Properties:

- HH is **negative semi-definite**: LL is concave
- HH is **sparse**: Only non-zero for connected neurons
- $-H$−H is the **Fisher Information Matrix**

3.4 Fisher Information Matrix

The expected Hessian:

$$I_{ij} = -E[H_{ij}] = \Delta t \sum_{t=1}^{N} \lambda_t \frac{\partial f_t}{\partial \theta_i} \frac{\partial f_t}{\partial \theta_j}$$

Iij = −E[Hij] = Δt t=1∑N λt∂θi∂ft∂θj∂ft

Provides:

- **Parameter uncertainty**: Cov($\hat{\theta}$) ≈ I^{-1}Cov(θ^) ≈ I−1
- **Cramér-Rao bound**: Minimum variance bound for unbiased estimators
- **Asymptotic distribution**: $\hat{\theta} \sim N(\theta^*, I^{-1})$θ^ ∼ N (θ∗, I−1)

4. Optimization Algorithm

4.1 Newton-Raphson Method

The **Newton-Raphson algorithm** iteratively updates parameters:

$$\theta^{(t+1)} = \theta^{(t)} - \eta H^{-1}(\theta^{(t)})\nabla L(\theta^{(t)})$$

θ(t+1) = θ(t) – ηH–1(θ(t))∇L(θ(t))

where:

- $\eta \in (0, 1]$η ∈ (0, 1]: Learning rate (often $\eta = 1$η = 1)
- HH: Hessian matrix
- ∇L∇L: Gradient vector

Why Newton-Raphson?

- **Quadratic convergence** near optimum
- **Uses curvature information** (second-order method)
- **Optimal for convex problems**

4.2 Algorithm Pseudocode

```
Algorithm: Newton-Raphson MLE
──────────────────────────────────────────

Input: Spike data {r_i}, stimulus {x_i}, spike histories {y_i}
Output: Optimal parameters θ*

1. Initialize θ^(0) randomly or from spike-triggered average
2. Set convergence tolerance ε = 10^(-6)
3. Set max_iterations = 100

4. For t = 1 to max_iterations:

   a. Compute linear predictor:
      f_i = k·x_i + h·y_i + Σ_j l_j·y_i^(j) + μ  ∀i

   b. Compute firing rates:
      λ_i = exp(f_i)  ∀i

   c. Compute log-likelihood:
      L = Σ_i r_i·f_i - Δt·Σ_i λ_i

   d. Compute gradient:
      ∇L[j] = Σ_i (r_i - λ_i·Δt)·∂f_i/∂θ_j  ∀j

   e. Compute Hessian:
      H[j,k] = -Δt·Σ_i λ_i·(∂f_i/∂θ_j)·(∂f_i/∂θ_k)  ∀j,k

   f. Newton step:
      Δθ = -H^(-1)·∇L

   g. Update parameters:
      θ^(t+1) = θ^(t) + η·Δθ

   h. Check convergence:
      If ||Δθ|| < ε:
         Return θ^(t+1)

5. Return θ^(max_iterations) (convergence not reached warning)
```

4.3 Computational Complexity

For NN neurons with MM connections each and TT time bins:

Operation	Complexity	Notes
Compute $f_i$fi	$O(NMT)$O(NMT)	Filter convolutions
Compute λ_iλi	$O(NT)$O(NT)	Exponentials
Gradient	$O(NMT)$O(NMT)	Sum over time
Hessian	$O(N^2M^2T)$O(N2M2T)	Can be sparse
Invert Hessian	$O(N^3M^3)$O(N3M3)	Most expensive

Sparsity helps: With regularization, Hessian is sparse, reducing inversion cost.

---

5. Regularization for Sparsity

5.1 Why Regularize?

Problems without regularization:

- **Overfitting**: Too many coupling filters (702 potential connections for 27 neurons)
- **Poor generalization**: Model fits noise in training data
- **Computational cost**: Dense Hessian matrix

5.2 L^(1/2) Penalty

The paper uses a **sublinear penalty** on coupling filters:

$$L_{\text{pen}}(\theta) = L(\theta) - \alpha \int_0^\infty \left(\sum_j |l_j(\tau)| \right)^{1/2} d\tau$$

Lpen(θ) = L(θ) – α ∫0∞ (j∑ |lj(τ)|)1/2 dτ

Discrete form:

$$L_{\text{pen}}(\theta) = L(\theta) - \alpha \sum_t \left(\sum_j |l_{j,t}| \right)^{1/2}$$

Lpen(θ) = L(θ) – α t∑ (j∑ |lj,t|)1/2

where $\alpha > 0$α > 0 is the regularization strength.

5.3 Why L^(1/2) Instead of L^1?

Penalty	Form	Effect	Sparsity
L^2	$\sum l_j^2$∑ lj2	Smooth shrinkage	None
L^1	$\sum	l_j	$
L^(1/2)	$(\sum	l_j	)^{1/2}$
L^0	$\sum 1(l_j \neq 0)$∑ 1(lj ≠ 0)	Perfect sparsity	Combinatorial

The L^(1/2) penalty:

- **Stronger than L^1**: More aggressively eliminates weak connections
- **Convex surrogate for L^0**: Still allows efficient optimization
- **Group sparsity**: Encourages entire filters to be zero

5.4 Optimization with Penalty

Modified Newton-Raphson:

$$\theta^{(t+1)} = \theta^{(t)} - \eta H_{\text{pen}}^{-1} \nabla L_{\text{pen}}(\theta^{(t)})$$

θ(t+1) = θ(t) – ηHpen−1∇Lpen(θ(t))

where:

$$\nabla L_{\text{pen}} = \nabla L - \alpha \nabla \text{Penalty}$$

∇Lpen = ∇L – α∇Penalty

$$H_{\text{pen}} = H - \alpha H_{\text{Penalty}}$$

Hpen = H – αHPenalty

Implementation note: The penalty gradient and Hessian can be computed analytically.

5.5 Results from Paper

- **Before regularization**: 702 potential coupling filters (27 choose 2 × 2 directions)
- **After regularization**: 243 coupling filters retained (~34%)
- **Structure recovered**: Nearest-neighbor connectivity emerges naturally

---

6. Cross-Validation Strategy

6.1 Data Splitting

The paper uses a three-way split:

```
Total data: 12 minutes of recording at 120 Hz
├── Training set:    7 minutes  (58%)  → Fit model parameters θ
├── Validation set:  5 minutes  (42%)  → Select regularization α
└── Test set:        Separate   (0%)   → Evaluate final model
```

Note: The paper mentions 7 min training + 5 min validation + separate test data for decoding analysis.

6.2 Cross-Validation Procedure

```
Algorithm: Cross-Validation for Regularization
──────────────────────────────────────────────────────────────
Input: Training data D_train, Validation data D_val
Output: Optimal regularization α*

1. Define candidate regularization values:
   α_candidates = [0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1.0, 3.0]

2. For each α in α_candidates:

   a. Fit model on D_train using regularization α:
      θ_α = argmax_θ [L(θ; D_train) - α·Penalty(θ)]

   b. Evaluate log-likelihood on D_val:
      L_val(α) = L(θ_α; D_val)
      (Note: Do NOT include penalty term)

   c. Store validation performance:
      scores[α] = L_val(α)

3. Select best regularization:
   α* = argmax_α scores[α]

4. Refit model on D_train ∪ D_val using α*:
   θ* = argmax_θ [L(θ; D_train ∪ D_val) - α*·Penalty(θ)]

5. Return α*, θ*
```

6.3 K-Fold Cross-Validation (Alternative)

For smaller datasets, use K-fold CV:

```
Algorithm: K-Fold Cross-Validation
─────────────────────────────────────────
Input: Full data D, number of folds K
Output: Optimal α*

1. Partition D into K equal folds: D = D_1 ∪ D_2 ∪ ... ∪ D_K

2. For each α in α_candidates:

  a. Initialize score_sum = 0

  b. For k = 1 to K:
    - Train on D \ D_k (all folds except k-th)
    - Test on D_k
    - score_sum += L(θ_α; D_k)

  c. scores[α] = score_sum / K

3. α* = argmax_α scores[α]

4. Refit on full data D using α*

5. Return α*, θ*
```

6.4 Performance Metrics for Validation

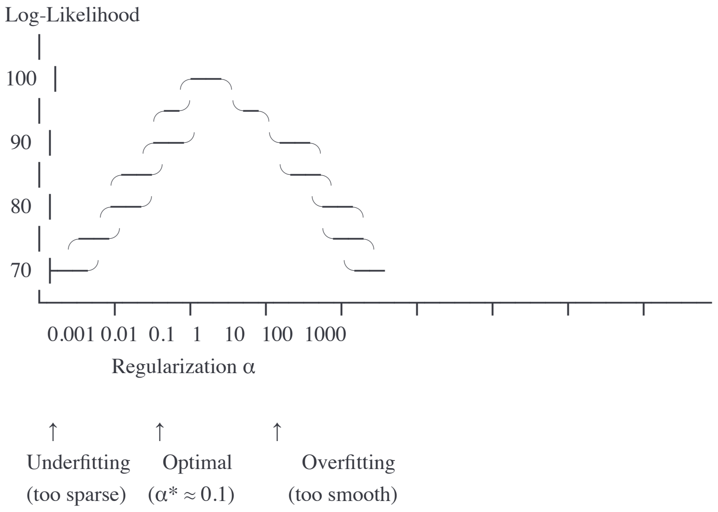
Multiple metrics can guide model selection:

Metric	Formula	Interpretation
Log-likelihood	$L(\theta; D_{\text{val}})$L(θ; Dval)	Direct measure of fit quality
Bits per spike	$\frac{1}{n}\left[\sum \log\lambda_i - \int \lambda dt\right]$n1 [∑ log λi − ∫ λdt]	Information per spike
AIC	$-2L + 2p$−2L + 2p	Penalizes model complexity
BIC	$-2L + p \log n$−2L + p log n	Stronger penalty than AIC

where pp = number of parameters, nn = number of spikes.

6.5 Visualization of Cross-Validation

Typical CV curve:



7. Preventing Overfitting

7.1 Sources of Overfitting in GLMs

1. **Too many parameters**:
   - Stimulus filters: $5 \times 5 \times 30 = 7505 \times 5 \times 30 = 750$ parameters per cell
   - Coupling filters: Up to $26 \times 4 = 10426 \times 4 = 104$ parameters per cell
   - Total: ~900 parameters per neuron
2. **Limited data**:
   - 7 minutes at 1kHz = 420,000 time bins
   - But only ~100-500 spikes per neuron
   - Parameter-to-spike ratio can be high
3. **Correlated features**:
   - Nearby pixels in stimulus are correlated
   - Spike histories of coupled neurons overlap

7.2 Overfitting Prevention Strategies

Strategy 1: Regularization (Primary Method)

L^(1/2) penalty on coupling filters:

- Forces weak connections to exactly zero
- Reduces effective parameter count
- Encourages biological connectivity patterns

Implementation:

python

```python
def penalized_likelihood(theta, data, alpha):
    """Compute penalized log-likelihood"""
    # Standard log-likelihood
    L = compute_log_likelihood(theta, data)

    # L^(1/2) penalty on coupling filters
    coupling_filters = theta['coupling']  # Shape: (n_neurons, n_lags)
    penalty = 0
    for lag in range(n_lags):
        penalty += np.sqrt(np.sum(np.abs(coupling_filters[:, lag])))

    return L - alpha * penalty
```

Strategy 2: Dimensionality Reduction

Rank-2 stimulus filter parametrization:

Instead of estimating all 750 parameters of kk, decompose:

$$k(x, y, t) = k_s^{(1)}(x, y) \cdot k_t^{(1)}(t) - k_s^{(2)}(x, y) \cdot k_t^{(2)}(t)$$

k(x, y, t) = ks(1)(x, y) · kt(1)(t) − ks(2)(x, y) · kt(2)(t)

This reduces parameters from 750 to $2 \times (25 + 10) = 702 \times (25 + 10) = 70$.

Implementation:

python

```python
def compute_stimulus_filter(ks1, kt1, ks2, kt2):
    """
    Rank-2 decomposition of stimulus filter
    ks1, ks2: spatial components (5x5)
    kt1, kt2: temporal components (30,)
    """
    filter_1 = np.outer(ks1.ravel(), kt1)
    filter_2 = np.outer(ks2.ravel(), kt2)
    return filter_1 - filter_2  # Shape: (25, 30)
```

Strategy 3: Temporal Basis Functions

Raised cosine basis for temporal filters:

Instead of $d_h = 50$dh = 50 free parameters for post-spike filter, use 10 basis functions:

$$h(t) = \sum_{j=1}^{10} \beta_j \cdot b_j(t)$$

h(t) = j=1∑10 βj · bj(t)

where $b_j(t) = \frac{1}{2}\left[1 + \cos\left(a\log(t+c) - \phi_j\right)\right]$bj(t) = 21 [1 + cos (a log(t + c) − ϕj)]

Reduces parameters from 50 to 10.

Strategy 4: Early Stopping

Monitor validation likelihood during optimization:

python

```python
def fit_with_early_stopping(data_train, data_val, patience=5):
    """Fit model with early stopping"""
    best_val_score = -np.inf
    patience_counter = 0
    best_theta = None

    for iteration in range(max_iterations):
        # Update parameters
        theta = newton_step(theta, data_train)

        # Evaluate on validation set
        val_score = log_likelihood(theta, data_val)

        if val_score > best_val_score:
            best_val_score = val_score
            best_theta = theta.copy()
            patience_counter = 0
        else:
            patience_counter += 1

        if patience_counter >= patience:
            print(f"Early stopping at iteration {iteration}")
            return best_theta

    return best_theta
```

Strategy 5: Ensemble Methods

Fit multiple models and average predictions:

python

```python
def fit_ensemble(data, n_models=5):
    """Fit ensemble of models with different initializations"""
    models = []

    for i in range(n_models):
        # Different random initialization
        theta_init = initialize_random(seed=i)
        theta_fit = fit_model(data, theta_init)
        models.append(theta_fit)

    return models

def predict_ensemble(models, stimulus):
    """Average predictions across ensemble"""
    predictions = [predict(m, stimulus) for m in models]
    return np.mean(predictions, axis=0)
```

7.3 Diagnosing Overfitting

Checklist:

✓ Training likelihood >> Validation likelihood

- Solution: Increase regularization

✓ Filters are noisy/high-frequency

- Solution: Smooth basis functions or L2 penalty

✓ Many weak coupling connections

- Solution: Stronger sparsity penalty

✓ Poor generalization to new stimuli

- Solution: More training data or simpler model

✓ Unstable parameter estimates

- Solution: Regularization or better initialization

7.4 Validation Curves

Plot performance vs. model complexity:

```
Performance
    │  Validation
    │     ╱╲
    │    ╱  ╲
    │   ╱    ╲_____________ Training
    │  ╱
    │ ╱
    │╱_____________________________
        Model Complexity

      ↑
    Sweet spot        Overfitting
    (good generalization)  region
```

---

8. Implementation Details

8.1 Numerical Stability

Problem: Computing $\exp(f_i)$exp(fi) for large $|f_i|$|fi| causes overflow/underflow.

Solution: Use log-sum-exp trick:

python

```python
def stable_log_likelihood(f, r, dt):
    """Numerically stable log-likelihood computation"""
    # Clip extreme values
    f_clipped = np.clip(f, -20, 20)

    # Log-likelihood
    spike_term = np.sum(r * f_clipped)
    integral_term = dt * np.sum(np.exp(f_clipped))

    return spike_term - integral_term
```

8.2 Initialization Strategy

Good initialization accelerates convergence:

python

```python
def initialize_parameters(stimulus, spikes):
    """Initialize from spike-triggered average"""
    # Stimulus filter: spike-triggered average
    k_init = spike_triggered_average(stimulus, spikes)

    # Post-spike filter: auto-correlation
    h_init = compute_autocorrelation(spikes)

    # Coupling filters: cross-correlation
    l_init = compute_cross_correlation(spikes, other_spikes)

    # Baseline rate
    mu_init = np.log(np.mean(spikes) / dt)

    return {'k': k_init, 'h': h_init, 'l': l_init, 'mu': mu_init}
```

8.3 Convergence Criteria

Multiple criteria for stopping:

python

```python
def check_convergence(theta_old, theta_new, L_old, L_new):
    """Check if optimization has converged"""
    # Parameter change
    param_change = np.linalg.norm(theta_new - theta_old)
    param_converged = param_change < 1e-6

    # Likelihood change
    likelihood_change = np.abs(L_new - L_old)
    likelihood_converged = likelihood_change < 1e-6

    # Gradient norm
    gradient = compute_gradient(theta_new)
    gradient_converged = np.linalg.norm(gradient) < 1e-5

    return param_converged and likelihood_converged and gradient_converged
```

8.4 Sparse Hessian Computation

Exploit sparsity for efficiency:

python

```python
from scipy.sparse import lil_matrix

def compute_sparse_hessian(theta, lambda_rates, features):
    """Compute sparse Hessian matrix"""
    n_params = len(theta)
    H = lil_matrix((n_params, n_params))

    # Only compute for non-zero connections
    for i, j in connected_pairs:
        H[i, j] = -dt * np.sum(lambda_rates * features[i] * features[j])
        H[j, i] = H[i, j]  # Symmetric

    return H.tocsr()  # Convert to efficient format
```

---

9. Python Code Examples

9.1 Complete Implementation

python

```python
import numpy as np
from scipy.optimize import minimize
from scipy.sparse.linalg import spsolve

class GLM:
    """Generalized Linear Model for neural spike trains"""

    def __init__(self, n_stimulus_dims, n_history_lags, n_neurons):
        self.n_stimulus_dims = n_stimulus_dims
        self.n_history_lags = n_history_lags
        self.n_neurons = n_neurons

        # Initialize parameters
        self.k = np.zeros(n_stimulus_dims)  # Stimulus filter
        self.h = np.zeros(n_history_lags)   # Post-spike filter
        self.l = {}  # Coupling filters (sparse dictionary)
        self.mu = 0.0  # Baseline log-rate

    def compute_firing_rate(self, stimulus, spike_history,
                            coupled_histories):
        """Compute instantaneous firing rate"""
        # Linear predictor
        f = np.dot(self.k, stimulus) + \
            np.dot(self.h, spike_history) + \
            self.mu

        # Add coupling contributions
        for neuron_id, coupling_filter in self.l.items():
            f += np.dot(coupling_filter, coupled_histories[neuron_id])

        # Exponential nonlinearity
        return np.exp(f)

    def log_likelihood(self, data, dt):
        """Compute log-likelihood on data"""
        stimulus, spikes, histories = data
        T = len(spikes)

        logL = 0.0
        for t in range(T):
            # Compute firing rate
            lambda_t = self.compute_firing_rate(
                stimulus[t],
                histories['self'][t],
                histories['coupled'][t]
            )

            # Add to log-likelihood
            if spikes[t] > 0:
                logL += np.log(lambda_t)
            logL -= lambda_t * dt

        return logL

    def fit(self, data_train, data_val, alpha=0.1, max_iter=100):
        """Fit model using penalized MLE"""
        dt = 0.001  # 1ms bins
        best_val_logL = -np.inf

        for iteration in range(max_iter):
            # Compute gradient and Hessian
            grad = self._compute_gradient(data_train, dt)
```

```
        hess = self._compute_hessian(data_train, dt)

        # Add regularization
        grad_pen, hess_pen = self._add_regularization(
            grad, hess, alpha
        )

        # Newton step
        delta_theta = spsolve(hess_pen, grad_pen)
        self._update_parameters(delta_theta, step_size=1.0)

        # Evaluate on validation set
        val_logL = self.log_likelihood(data_val, dt)

        if val_logL > best_val_logL:
            best_val_logL = val_logL
            self._save_best_parameters()

        # Check convergence
        if np.linalg.norm(delta_theta) < 1e-6:
            print(f"Converged at iteration {iteration}")
            break

    self._load_best_parameters()
    return self

def _compute_gradient(self, data, dt):
    """Compute gradient of log-likelihood"""
    stimulus, spikes, histories = data
    T = len(spikes)

    grad = np.zeros(self._n_params())

    for t in range(T):
        lambda_t = self.compute_firing_rate(
            stimulus[t],
            histories['self'][t],
            histories['coupled'][t]
        )

        # Residual: observed - predicted
        residual = spikes[t] - lambda_t * dt

        # Gradient contribution
        features = self._extract_features(
            stimulus[t],
            histories['self'][t],
            histories['coupled'][t]
        )
        grad += residual * features

    return grad

def _compute_hessian(self, data, dt):
    """Compute Hessian of log-likelihood"""
    stimulus, spikes, histories = data
    T = len(spikes)
    n_params = self._n_params()

    hess = np.zeros((n_params, n_params))

    for t in range(T):
```

```
            lambda_t = self.compute_firing_rate(
                stimulus[t],
                histories['self'][t],
                histories['coupled'][t]
            )

            features = self._extract_features(
                stimulus[t],
                histories['self'][t],
                histories['coupled'][t]
            )

            # Outer product weighted by firing rate
            hess -= dt * lambda_t * np.outer(features, features)

        return hess

    def _add_regularization(self, grad, hess, alpha):
        """Add L^(1/2) penalty to gradient and Hessian"""
        # Penalty gradient for coupling filters
        penalty_grad = np.zeros_like(grad)
        for neuron_id, l_j in self.l.items():
            for lag in range(len(l_j)):
                denom = 2 * np.sqrt(np.sum(np.abs(l_j)))
                penalty_grad[self._param_index(neuron_id, lag)] = \
                    alpha * np.sign(l_j[lag]) / denom

        grad_pen = grad - penalty_grad

        # Penalty Hessian (diagonal approximation)
        hess_pen = hess.copy()
        # Add small diagonal for numerical stability
        hess_pen -= alpha * 1e-3 * np.eye(len(grad))

        return grad_pen, hess_pen


def cross_validate_regularization(data_train, data_val,
                                  alpha_candidates):
    """Select optimal regularization via cross-validation"""
    results = {}

    for alpha in alpha_candidates:
        print(f"Testing alpha = {alpha}")

        # Fit model
        model = GLM(n_stimulus_dims=25, n_history_lags=10, n_neurons=27)
        model.fit(data_train, data_val, alpha=alpha)

        # Evaluate on validation set
        val_score = model.log_likelihood(data_val, dt=0.001)
        results[alpha] = val_score

        print(f"  Validation log-likelihood: {val_score:.2f}")

    # Select best alpha
    best_alpha = max(results, key=results.get)
    print(f"\nBest alpha: {best_alpha}")

    return best_alpha, results
```

```python
def plot_cross_validation_results(results):
    """Visualize cross-validation results"""
    import matplotlib.pyplot as plt

    alphas = list(results.keys())
    scores = list(results.values())

    plt.figure(figsize=(10, 6))
    plt.semilogx(alphas, scores, 'o-', linewidth=2, markersize=8)
    plt.xlabel('Regularization strength (α)', fontsize=12)
    plt.ylabel('Validation log-likelihood', fontsize=12)
    plt.title('Cross-Validation Curve', fontsize=14, fontweight='bold')
    plt.grid(True, alpha=0.3)

    # Mark best alpha
    best_alpha = max(results, key=results.get)
    best_score = results[best_alpha]
    plt.axvline(best_alpha, color='r', linestyle='--',
                label=f'Best α = {best_alpha:.3f}')
    plt.plot(best_alpha, best_score, 'r*', markersize=15)

    plt.legend()
    plt.tight_layout()
    plt.savefig('cross_validation_curve.pdf')
    plt.show()


# Example usage
if __name__ == "__main__":
    # Generate synthetic data
    from generate_data import load_spike_data

    data_train, data_val, data_test = load_spike_data()

    # Cross-validate regularization
    alpha_candidates = [0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1.0]
    best_alpha, cv_results = cross_validate_regularization(
        data_train, data_val, alpha_candidates
    )

    # Plot results
    plot_cross_validation_results(cv_results)

    # Fit final model with best alpha
    final_model = GLM(n_stimulus_dims=25, n_history_lags=10, n_neurons=27)
    final_model.fit(
        np.concatenate([data_train, data_val]),
        data_test,
        alpha=best_alpha
    )

    print(f"Final model fitted with {len(final_model.l)} coupling filters")
    print(f"Test log-likelihood: {final_model.log_likelihood(data_test, 0.001):.2f}")
```

9.2 Visualization Code

python

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

def visualize_model_parameters(model):
    """Visualize fitted model parameters"""
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # Stimulus filter
    k_spatial = model.k[:25].reshape(5, 5)
    axes[0, 0].imshow(k_spatial, cmap='RdBu_r')
    axes[0, 0].set_title('Stimulus Filter (Spatial)', fontweight='bold')
    axes[0, 0].axis('off')

    k_temporal = model.k[25:]
    axes[0, 1].plot(k_temporal, linewidth=2)
    axes[0, 1].set_title('Stimulus Filter (Temporal)', fontweight='bold')
    axes[0, 1].set_xlabel('Time lag (ms)')
    axes[0, 1].grid(True, alpha=0.3)

    # Post-spike filter
    time_lags = np.arange(len(model.h))
    axes[0, 2].plot(time_lags, np.exp(model.h), linewidth=2, color='green')
    axes[0, 2].axhline(1.0, color='k', linestyle='--', alpha=0.5)
    axes[0, 2].set_title('Post-Spike Filter (Gain)', fontweight='bold')
    axes[0, 2].set_xlabel('Time after spike (ms)')
    axes[0, 2].set_ylabel('Rate multiplier')
    axes[0, 2].grid(True, alpha=0.3)

    # Coupling filters
    if len(model.l) > 0:
        axes[1, 0].set_title('Coupling Filters', fontweight='bold')
        for neuron_id, coupling in model.l.items():
            axes[1, 0].plot(time_lags[:len(coupling)], coupling,
                            alpha=0.6, linewidth=1.5)
        axes[1, 0].set_xlabel('Time lag (ms)')
        axes[1, 0].set_ylabel('Coupling strength')
        axes[1, 0].grid(True, alpha=0.3)

    # Connectivity matrix
    n_neurons = 27
    connectivity = np.zeros((n_neurons, n_neurons))
    for neuron_id in model.l.keys():
        connectivity[model.neuron_idx, neuron_id] = 1

    axes[1, 1].imshow(connectivity, cmap='binary')
    axes[1, 1].set_title('Connectivity Matrix', fontweight='bold')
    axes[1, 1].set_xlabel('Source neuron')
    axes[1, 1].set_ylabel('Target neuron')

    # Histogram of coupling strengths
    coupling_strengths = [np.max(np.abs(c)) for c in model.l.values()]
    axes[1, 2].hist(coupling_strengths, bins=20, color='steelblue', alpha=0.7)
    axes[1, 2].set_title('Coupling Strength Distribution', fontweight='bold')
    axes[1, 2].set_xlabel('Max |coupling|')
    axes[1, 2].set_ylabel('Count')
    axes[1, 2].grid(True, alpha=0.3, axis='y')

    plt.tight_layout()
    plt.savefig('model_parameters.pdf')
    plt.show()
```

Summary

This document provides a complete mathematical and practical guide to fitting generalized linear models for neural spike trains using **maximum likelihood estimation**. Key takeaways:

1. **Log-likelihood** balances rewarding predicted spikes and penalizing high rates
2. **Newton-Raphson optimization** exploits convexity for fast convergence
3. **L^(1/2) regularization** induces sparsity in coupling connections
4. **Cross-validation** prevents overfitting by selecting optimal regularization
5. **Proper implementation** requires numerical stability and efficient computation

The paper's approach recovered biologically plausible connectivity patterns and demonstrated that **correlations provide 20% more visual information**, highlighting the importance of population-level neural coding.

References

- Pillow et al. (2008). "Spatio-temporal correlations and visual signalling in a complete neuronal population." *Nature* 454, 995-999.
- Paninski (2004). "Maximum likelihood estimation of cascade point-process neural encoding models." *Network* 15, 243-262.
- Truccolo et al. (2005). "A point process framework for relating neural spiking activity to spiking history, neural ensemble, and extrinsic covariate effects." *J Neurophysiol* 93, 1074-1089.