Supplementary materials for this article are available online.
Please go to www.tandfonline.com/r/JCGS

A Dynamic Programming Algorithm for the Fused Lasso and L_0-Segmentation

Nicholas A. JOHNSON

We propose a dynamic programming algorithm for the one-dimensional Fused Lasso Signal Approximator (FLSA). The proposed algorithm has a linear running time in the worst case. A similar approach is developed for the task of least squares segmentation, and simulations indicate substantial performance improvement over existing algorithms. Examples of R and C implementations are provided in the online Supplementary materials, posted on the journal web site.

Key Words: Optimization; Signal processing; Total variation denoising; Viterbi.

1. INTRODUCTION

The Fused Lasso Signal Approximator (FLSA) by Friedman et al. (2007) smooths noisy, real-valued observations $y_1, \ldots, y_N$ by finding the sequence $\hat{\beta}_1, \ldots, \hat{\beta}_N$ that maximizes the criterion:

$$(-1/2)\sum_{k=1}^{N}(y_k - \beta_k)^2 - \lambda_1 \sum_{k=1}^{N}|\beta_k| - \lambda_2 \sum_{k=2}^{N}|\beta_k - \beta_{k-1}| \quad (1)$$

where λ_1 and λ_2 are tuning parameters.

Friedman et al. (2007) explained that λ_1 encourages "sparsity in the coefficients" and λ_2 encourages "sparsity in their differences." We present an algorithm only for the special case when $\lambda_1 = 0$; however, this is sufficient because lemma 1 by Friedman et al. (2007) showed that the kth component of the solution at (λ_1, λ_2), denoted as $\hat{\beta}_k(\lambda_1, \lambda_2)$, can be found through a soft-thresholding operation: $\hat{\beta}_k(\lambda_1, \lambda_2) = \text{sign}(\hat{\beta}_k(0, \lambda_2))\max\{0, \hat{\beta}_k(0, \lambda_2) - \lambda_1\}$.

When $\lambda_1 = 0$, the FLSA performs a total variation denoising of the signal, and an exact path algorithm for this special case was introduced by Pollak, Willsky, and Huang (2005). The path algorithm was extended to the general Fused Lasso by Hofling (2009b) and implemented in the "flsa" R package. The path algorithm steps through the entire solution path in $O(N \log N)$ operations, and it does so by tracing the trajectory of segments as λ_2 increases from 0 to ∞ with λ_1 fixed at zero; the version by Hofling (2009b) also fixes λ_1 but at an arbitrary value.

Nicholas A. Johnson, Stanford University, Department of Statistics, 390 Sera Mall, Stanford, CA 94305 (E-mail: *nickaj@gmail.com*).

Journal of Computational and Graphical Statistics, Volume 22, Number 2, Pages 246–260
DOI: 10.1080/10618600.2012.681238

Our approach begins with the observation that when $\lambda_1 = 0$, Equation (1) is the complete-data likelihood of a particular Hidden Markov Model (HMM). In this HMM, the β's take the role of the hidden states, and their most probable value may be found via the Viterbi algorithm (a dynamic programming algorithm). The HMM posits an emission probability $\Pr(y_k|\beta_k)$ that is standard normal, and a transition probability $\Pr(\beta_{k+1}|\beta_k)$ that is double exponential with parameter λ_2 (where Pr() denotes probability).

This viewpoint is not necessary to derive our algorithm, but we find it useful since dynamic programming algorithms for HMMs are well known—excellent descriptions of the Viterbi algorithm are available in the literature by Rabiner (1989) and Durbin et al. (1999).

It is helpful to rewrite the objective (1) in a more general form:

$$\sum_{k=1}^{N} e_k(\beta_k) - \lambda_2 \sum_{k=2}^{N} d(\beta_k, \beta_{k-1}), \tag{2}$$

where

$$e_k(b) = \sum_{i=1}^{R} y_{ik} v_i(b). \tag{3}$$

It is assumed that (a) each function $v_i()$ is concave and differentiable, (b) $v_1(b) = b$, and (c) each y_{ik} is nonnegative for $i = 2, \ldots, R$. This positivity constraint ensures concavity of $e_k()$—a necessary assumption in all of the proposed algorithms. The proposed algorithms apply when the $e_k()$ have these properties and the algorithm has linear complexity for any such $v_i()$'s; however, the efficiency and ease of implementation will depend on the $v_i()$'s. The algorithm is fastest and easiest to implement when the $e_k()$ are quadratic.

We call this generalization of the FLSA the EFLSA where the "E" stands for "exponential family." Although our approach cannot be applied to an arbitrary member of the exponential family, the likelihoods of the Bernoulli, binomial, Poisson, and exponential distributions do satisfy the conditions placed on the $v_i()$'s.

A related problem, least squares segmentation, can be solved via a similar algorithm. Least squares segmentation involves optimizing a criterion of the form (2) above with an L_0-norm penalty, $d(b_1, b_2) = 1\{b_1 \neq b_2\}$, and squared error loss $e_k(b) = y_k v_1(b) - (1/2)v_2(b)$ (with $v_1(b) = b$ and $v_2(b) = b^2$). Although this L_0-penalized objective is not a convex function of β, the proposed algorithm still yields an exact solution.

The third and final variation considered is least squares segmentation constrained to a prespecified number of jumps. Timing comparisons with an alternative dynamic programming algorithm for the same task (Bellman 1961) show promising results.

Section 2 introduces the dynamic programming algorithm for the Fused Lasso. Section 3 modifies the algorithm to perform least squares segmentation. Section 4 presents simulations and timing comparisons. Finally, Section 5 concludes and describes some extensions of this work.

2. THE DYNAMIC PROGRAMMING ALGORITHM FOR THE FLSA

This section begins with a derivation of the dynamic programming algorithm and presents one particular implementation. The derivation is straightforward, but it is necessary to introduce notation; the majority of the section is spent showing that the necessary maximizations are easy to perform and that the functions defined have simple representations.

In the equations below, sequences of variables $(x_1, x_2, \ldots, x_k)$ are denoted by the shorthand $x_{1:k}$.

The first step is to rewrite the maximization of the criterion in Equation (2) in the following way:

$$\max_{\beta_{1:N}} \left[\sum_{k=1}^{N} e_k(\beta_k) - \lambda_2 \sum_{k=2}^{N} d(\beta_k, \beta_{k-1}) \right] \tag{4}$$

$$= \max_{\beta_N} \left[e_N(\beta_N) + \max_{\beta_{1:(N-1)}} \left[\sum_{k=1}^{N-1} e_k(\beta_k) - \lambda_2 \sum_{k=2}^{N} d(\beta_k, \beta_{k-1}) \right] \right] \tag{5}$$

$$f_N(\beta_N) := \max_{\beta_{1:(N-1)}} \left[\sum_{k=1}^{N-1} e_k(\beta_k) - \lambda_2 \sum_{k=2}^{N} d(\beta_k, \beta_{k-1}) \right] \tag{6}$$

$$= \max_{\beta_{N-1}} \Bigg[e_{N-1}(\beta_{N-1}) + \lambda d(\beta_N, \beta_{N-1}) + \max_{\beta_{1:(N-2)}} \left[\sum_{k=1}^{N-2} e_k(\beta_k) - \lambda_2 \sum_{k=2}^{N-1} d(\beta_k, \beta_{k-1}) \right] \Bigg] \tag{7}$$

The maximization is further iterated and functions $f_{N-1}(\beta_{N-1})$, $f_{N-2}(\beta_{N-2})$, ..., $f_2(\beta_2)$ are defined similarly. Intermediate functions are introduced to summarize the algorithm (with k ranging from 2 to N):

$$\delta_1(b) := e_1(b) \tag{8}$$

$$\psi_k(b) := \text{argmax}_{\tilde{b}} \left[\delta_{k-1}(\tilde{b}) - \lambda_2 |b - \tilde{b}| \right] \tag{9}$$

$$f_k(b) := \delta_{k-1}(\psi_k(b)) - \lambda_2 |b - \psi_k(b)| \tag{10}$$

$$\delta_k(b) := e_k(b) + f_k(b) \tag{11}$$

The functions $\psi_k()$ take part in the backward pass of the algorithm, and they are referred to as "back-pointers." This backward pass computes $\hat{\beta}_1, \ldots, \hat{\beta}_N$ through a recursion identical to that of the Viterbi algorithm for HMMs:

$$\hat{\beta}_N = \text{argmax}_b \{\delta_N(b)\} \tag{12}$$

$$\hat{\beta}_k = \psi_{k+1}(\hat{\beta}_{k+1}) \quad \text{for k} = N-1, N-2, \ldots, 1 \tag{13}$$

The key to the computational feasibility of the algorithm is the following result that is stated for the FLSA; however, it applies equally well to the EFLSA after some modifications (discussed in the Appendix available online with the Supplementary Materials).

Theorem 1. If each $e_k(b)$ is a quadratic function $y_{0,k} + y_{1,k}b + y_{2,k}b^2$ with $y_{2,k} < 0$, then each function $\delta_k(b)$ is concave, differentiable, and piecewise quadratic (PWQ). The back-pointer $\psi_k(b)$ is of the form $(b \wedge b_k^+) \vee b_k^-$ and b_k^- and b_k^+ are such that $\delta'_{k-1}(b_k^-) = \lambda_2$ and $\delta'_{k-1}(b_k^+) = -\lambda_2$ (where $\delta'_{k-1}()$ denotes the derivative).

Algorithm 1 is a simple iteration of two steps followed by a final "backtrace." It is simplified by Theorem 1 that results in the elimination of $\psi_k()$ since it is completely described by the two points b_k^- and b_k^+. The result is that Equations (9) and (10) are combined into a single, concise "clamping" operation on the derivative $\delta'_k()$ (Line 3 of Algorithm 1).

In Algorithm 1, the function Clamp($f(), r$) is defined as follows. The input function $f()$ is assumed to be continuous and decreasing (the functions $\delta'_k()$ are both), and Clamp() determines values x and y such that the function $g(b) = u \wedge (f(b) \vee -u)$ is also equal to $f(y \wedge (b \vee x))$. If one or both solutions to $f(b) = \pm u$ do not exist, they are defined to be the appropriate boundary of the domain of $f()$ (this only occurs for some EFLSA algorithms and never for the FLSA). The return value of Clamp($f(), r$) is the triple $(g(), x, y)$. Details of the clamp operation are given in Algorithm 2.

Algorithm 1 The dynamic programming algorithm for the FLSA and EFLSA.

1: Initialize $\delta'_1(b) = e'_1(b)$ ▷ ($h'()$ denotes the derivative of a function $h()$)
2: **for** $k = 1, \ldots, N$ **do**
3: $(f'_{k+1}(), b_{k+1}^-, b_{k+1}^+) = \text{Clamp}(\delta'_k(), \lambda_2)$ ▷ see text for details
4: Accumulate: $\delta'_{k+1}(b) = f'_{k+1}(b) + e'_{k+1}(b)$
5: **end for**
6: Solve: $\hat{\beta}_N$ such that $0 = \delta'_N(\hat{\beta}_N)$
7: **for** $k = N-1, N-2, \ldots, 1$ **do** ▷ the backtrace
8: $\hat{\beta}_k = b_{k+1}^+ \wedge (\hat{\beta}_{k+1} \vee b_{k+1}^-)$
9: **end for**

The "Clamp" and "Accumulate" steps (Lines 3 and 4) of Algorithm 1 will be presented in further detail, but this high-level view of the algorithm would be lost if each step were expanded out. Readers may see more efficient choices or simpler approaches than those presented below in Algorithm 2.

Although most implementations of these steps will yield excellent performance for random sequences, they will not have $O(N)$ worst-case performance unless some care is taken. For example, the "Clamp" step implicitly includes the search for the points b_k^- and b_k^+, and this must not be performed in a naive way. The remainder of this section outlines an implementation with $O(N)$ worst-case performance.

We claim that the function $\delta'_k()$ can be written in the following two ways:

$$\delta'_k(b) = \sum_{t=1}^{m(k)} s_t 1\{b \geq x_t\} \sum_{i=1}^{R} v'_i(b) a_{ti} \tag{14}$$

$$= \sum_{t=1}^{m(k)} -s_t 1\{b \leq x_t\} \sum_{i=1}^{R} v'_i(b) a_{ti}. \tag{15}$$

This representation is in terms of sorted knots $x_1 < x_2 < \cdots < x_{m(k)}$ with "signs," $s_1, \ldots, s_{m(k)} \in \{-1, 1\}$, and coefficients, $((a_{ti})_{i=1}^{R})_{t=1}^{m(k)}$. The knots, signs, and coefficients are the same in both equations. The number of knots, $m(k)$, may vary with the index k, but the concave, differentiable functions $v_1(), \ldots, v_R()$ do not. Finally, the end-point knots will always be at the boundaries of the parameter space and have signs $+1$ and -1; for example, $x_1 = -\infty$, $s_1 = +1$, $x_{m(k)} = \infty$, and $s_{m(k)} = -1$.

The claimed representation can be shown by induction. We make the argument informally, but the "proof" is in the pseudocode given later. When two functions $u'(b)$ and $v'(b)$ have the form (14) and (15), then so too does the clamped function $(u'(b) \vee -\lambda_2) \wedge \lambda_2)$ and the sum $u'(b) + v'(b)$. The base case is that $\delta'_1(b) = e'_1(b)$ has the form (14) and (15).

The existence of such a representation implies that b_k^- may be found by stepping from left to right through the knots. Likewise, the point b_k^+ is located by stepping from right to left. In the process of locating these derivatives, the algorithm need not visit every knot. Furthermore, every knot that is accessed is deleted and never used again. Pseudocode is given below in Algorithm 2.

Lines 27 through 31 of Algorithm 2 perform the "accumulate" step, $e'_{k+1}() + f'_{k+1}()$, by adding the coefficients $(y_{i,(k+1)})_i$ of the error function $e_{k+1}()$ at the end-point knots.

This particular representation and "in-place" transformation from $\delta'_k()$ to $\delta'_{k+1}()$ is necessary to ensure that the number of operations grows linearly in the sequence length. We explain now why the algorithm is $O(N)$ even in the worst case.

Because Algorithm 2 marks a knot as "to delete" in each iteration of the loops at Lines 5 and 16, it suffices to count the number of times the algorithm enters, say, either Line 8 or 19. A knot could be marked "to delete" at most twice, and the representation of $\delta'_{k+1}()$ requires $m(k) - m_d + 4$ knots where m_d is the number of knots deleted from $\delta_k()$. It follows that Lines 8 and 19 can only be entered $8N$ times. The final function $\delta'_N()$ is represented by at most $4N$ knots and therefore the final maximization $\text{argmax}_b \delta_N(b)$ can be performed in $O(N)$ time. Finally, the backtrace involves $O(N)$ operations, and so each of the forward and backward passes of the algorithm have $O(N)$ worst-case complexity. The EFLSA generalizations described below have the same worst-case complexity.

Algorithms for the EFLSA are most efficient when the equations in Lines 8 and 19 of Algorithm 2 can be solved in closed form. This is possible for the following Poisson,

Algorithm 2 Inner loop: "Clamp" and "Accumulate" steps for the FLSA.

function CLAMPACCUMULATE($\delta_k()$, $e_{k+1}()$)
  Represent $\delta'_k()$ as in Equation (14)
  Represent $e'_{k+1}(b)$ as $\sum_{i=1}^{R} y_{i,(k+1)} v'_i(b)$

  Initialize $a_{0i} \leftarrow 0$ for $i = 1, \ldots, R$
  for $t = 1, 2, \ldots, m(k)$ **do** ▷ for each knot-index from left to right
    $a_{0i} \leftarrow a_{0i} + s_t a_{ti}$ for $i = 1, \ldots, R$ ▷ accumulate coefficients
    Mark tth knot as "to delete"
    Find $\tilde{b}$ such that $\lambda_2 = \sum_{i=1}^{R} v'_i(\tilde{b}) a_{0i}$
    if $\tilde{b} < x_{t+1}$ **then** ▷ the derivative λ_2 lies in the current interval
      $b^-_{k+1} \leftarrow \tilde{b}$ ▷ store new knot
      $A^-_i \leftarrow a_{0i}$ for $i = 1, \ldots, R$ ▷ store coefficients
      Exit the loop
    end if
  end for

  Initialize $a_{0i} \leftarrow 0$ for $i = 1, \ldots, R$
  for $t = m(k), m(k) - 1, \ldots, 1$ **do** ▷ for each knot index from right to left
    $a_{0i} \leftarrow a_{0i} - s_t a_{ti}$ for $i = 1, \ldots, R$ ▷ accumulate coefficients
    Mark tth knot as "to delete"
    Find $\tilde{b}$ such that $-\lambda_2 = \sum_{i=1}^{R} v'_i(\tilde{b}) a_{0i}$
    if $\tilde{b} > x_{t-1}$ **then** ▷ the derivative $-\lambda_2$ lies in the current interval
      $b^+_{k+1} \leftarrow \tilde{b}$ ▷ store new knot
      $A^+_i \leftarrow a_{0i}$ for $i = 1, \ldots, R$ ▷ store coefficients
      Exit the loop
    end if
  end for

  Delete all knots marked as "to delete" from $\delta_k()$
  Denote the vector of coefficients $(\lambda_2 1\{i = 1\})_{i=1}^{R}$ by d
  Prepend (knot,coefficients,sign) $(b^-_{k+1}, -d + (A^-_i)_{i=1}^{R}, +1)$ to $\delta_k()$
  Prepend $(-\infty, d + (y_{i,(k+1)})_{i=1}^{R}, +1)$
  Append $(b^+_{k+1}, d + (A^+_i)_{i=1}^{R}, -1)$
  Append $(+\infty, -d + (y_{i,(k+1)})_{i=1}^{R}, -1)$
  return(modified-$\delta_k()$, b^-_{k+1}, b^+_{k+1}) ▷ modified $\delta_k()$ is $\delta_{k+1}()$
end function

binomial, and exponential log-likelihoods:

$$
\begin{array}{lll}
e_k(b) = -b + y_k \log(b) & b \in [0, \infty) & (y_k \sim \text{Pois}(b)) \\
e_k(b) = -e^b + y_k b & b \in \mathbb{R} & (y_k \sim \text{Pois}(e^b)) \\
e_k(b) = y_k^1 \log(b) + y_k^2 \log(1-b) & b \in [0, 1] & \left(y_k^1 \sim \text{Binom}\left(b; y_k^2\right)\right) \\
e_k(b) = y_k^1 b + y_k^2 \log(1 + \exp(b)) & b \in \mathbb{R} & \left(y_k^1 \sim \text{Binom}\left((1+e^{-b})^{-1}; y_k^2\right)\right) \\
e_k(b) = -b y_k - \log b & b \in [0, \infty) & (y_k \sim \text{Exp}(b)) \\
e_k(b) = -e^b y_k - b & b \in \mathbb{R} & (y_k \sim \text{Exp}(e^b))
\end{array}
\tag{16}
\tag{17}
$$

For example, when the algorithm is applied to the exponential model (17), the functions $e_k()$ and $\delta_k()$ are represented using the two functions $v_1(b) = b$ and $v_2(b) = -\exp(b)$, and the solution to $x v'_1(b) + y v'_2(b) = z$ is available in closed form: $b = \log((x - z)/y)$ (when

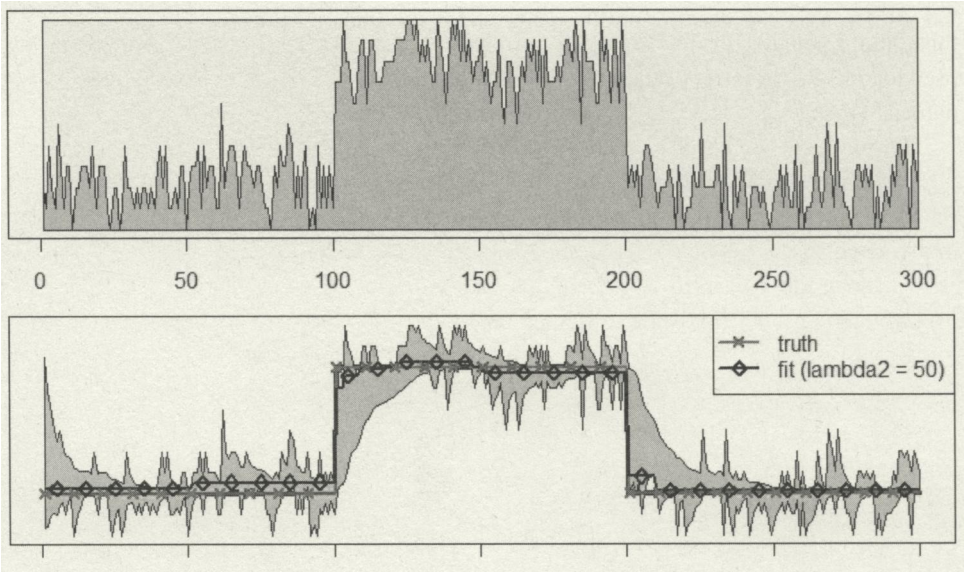

Figure 1. A simulated binomial sequence shown as the height of the polygon (top panel). The true probabilities that generated the data are shown as the red line in the bottom panel and the EFLSA fit ($\lambda_2 = 50$) is the black line. The intervals, $[b_k^-, b_k^+]$, are also plotted as the gray region.

this is defined). The Appendix (available online with the Supplementary Materials) gives some discussion on how to handle nonexistence of the solution.

The EFLSA is not meant to smooth the observed sequence itself—rather it produces an estimate of the mean at each index of the observation sequence. This generalization, specifically the Poisson EFLSA, may be a useful extension of the work by Tibshirani and Wang (2008). Copy number variations (CNVs) can now be detected from the output of next generation sequencing platforms. These platforms produce data in the form of sequences of nonnegative integers, and the Poisson model is an obvious candidate.

Figure 1 depicts a simulated binomial sequence, the intervals $[b_k^-, b_k^+]$, and the EFLSA fit, $\hat{\beta}$. The fit, $\hat{\beta}$, (black curve) exhibits "rounded edges" near the jumps in underlying signal, and this behavior is shown to be expected. Theorem 2 in the Appendix (available online with the Supplementary Materials) considers the FLSA fit in the presence of jumps in the underlying signal. It shows that the FLSA will not yield clean segmentations unless either (a) the jump is large compared to the noise level, or (b) the FLSA fit is highly biased.

3. SEGMENTATION

Optimal least squares segmentation requires maximization of the objective function

$$-(1/2)\sum_{k=1}^{N}(y_k - \beta_k)^2 - \lambda_2 \sum_{k=2}^{N} 1\{\beta_k \neq \beta_{k-1}\}. \tag{18}$$

The term $\sum_{k=2}^{N} 1\{\beta_k \neq \beta_{k-1}\}$ counts the number of jumps in the solution; if the solution $\hat{\beta}$ has M jumps, the resulting segmentation will minimize the squared error

$\sum_{k=1}^{N}(y_k - \hat{\beta}_k)^2$ among all segmentations with M or fewer jumps. The algorithm for this "jump-penalized" segmentation problem is named "DpSegJump" in tables and pseudocode. A variation of the algorithm that explicitly sets the number of jumps will be named "DpSegN."

The functions $e_k()$ and $v_i()$ retain their definitions from Section 2, and the functions $v_i()$ are still assumed to be differentiable and concave. Although the algorithms are stated for more general $e_k()$, we stress that only squared error loss currently has an efficient implementation.

We derive the algorithm by iterating the maximization of Criterion (18) as in Equations (4) through (7). This yields the following recursion:

$$b_k^* = \text{argmax}_{\tilde{b}}\delta_{k-1}(\tilde{b}) \tag{19}$$

$$J_k := \{b : \delta_{k-1}(b) < \delta_{k-1}(b_k^*) - \lambda_2\} \tag{20}$$

$$f_k(b) := \max_{\tilde{b}}[\delta_{k-1}(\tilde{b}) - \lambda_2 1\{b \neq \tilde{b}\}] \tag{21}$$

$$= \max_{\tilde{b}\in\{b,b_k^*\}}\left[\delta_{k-1}(\tilde{b}) - \lambda_2 1\{b \neq \tilde{b}\}\right] \tag{22}$$

$$= \delta_{k-1}(b) \vee [\delta_{k-1}(b_k^*) - \lambda_2] \tag{23}$$

$$= \delta_{k-1}(b)1\{b \notin J_k\} + (\delta_{k-1}(b_k^*) - \lambda_2)1\{b \in J_k\} \tag{24}$$

$$\delta_k(b) := e_k(b) + f_k(b). \tag{25}$$

The functions $\delta_k()$ (and $f_k()$) have piecewise quadratic representations and are described by a vector of sorted knots, $-\infty = x_1 < x_2 < \cdots < x_{m(k)+1} = \infty$, constants, $(c_t)_{t=1}^{m(k)+1}$, and coefficients, $((a_{ti})_{i=1}^{R})_{t=1}^{m(k)+1}$. This representation is of $\delta_k()$ itself rather than $\delta_k'()$:

$$\delta_k(b) = \sum_{t=1}^{m(k)} 1\{b \in [x_t, x_{t+1})\}\left[c_t + \sum_{i=1}^{R} a_{ti}v_i(b)\right]. \tag{26}$$

To simplify the code and formulas, the representation always includes a final $(m(k) + 1)$ dummy element with zero coefficients and knot $x_{m(k)+1}$ set to $+\infty$.

Pseudocode in Algorithms 3 and 4 shows how to maximize $\delta_k()$, find the intervals J_k, and perform the "flood" operation (Equation (23)). Figure 2 shows a simulated sequence

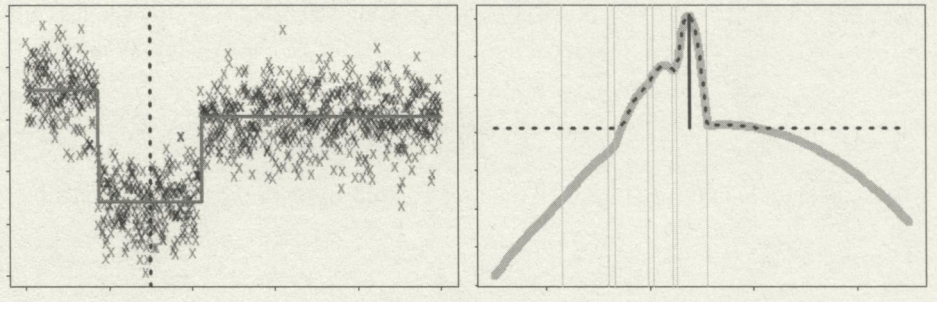

Figure 2. An example sequence and least squares segmentation ($\lambda_2 = 15$ in the left panel). The right panel depicts $f_{k+1}()$ (dotted black) and $\delta_k()$ (solid gray) for $k = 300$. The solid black line indicates the location $b_{k+1}^* = \text{argmax}_b\delta_k(b)$ and the threshold $\delta_k(b_{k+1}^*) - \lambda_2$.

Algorithm 3 The "DpSegPen" algorithm (Part 1)

function MAXIMIZE($\delta_k()$)
  $b_k^* \leftarrow -\infty$
  $m_k^* \leftarrow -\infty$
  for $t = 1, 2, \ldots, m(k)$ **do** ▷ for each segment
    Solve: $\tilde{b}$ s.t. $0 = \sum_{i=1}^{R} a_{ti} v_i'(\tilde{b})$ ▷ maximize a concave function
    if $\tilde{b} > x_{t+1}$ **then**
      $\tilde{b} \leftarrow x_{t+1}$ ▷ max over $[x_t, x_{t+1}]$ is at x_{t+1}
    end if
    if $\tilde{b} < x_t$ **then**
      $\tilde{b} \leftarrow x_t$ ▷ max over $[x_t, x_{t+1}]$ is at x_t
    end if
    if $m_k^* < c_t + \sum_{i=1}^{R} a_{ti} v_i(\tilde{b})$ **then** ▷ Current max is largest so far
      $m_k^* \leftarrow c_t + \sum_{i=1}^{R} a_{ti} v_i(\tilde{b})$
      $b_k^* \leftarrow \tilde{b}$
    end if
  end for
  return (b_k^*, m_k^*)
end function

function BACKTRACE($b_1^*, \ldots, b_N^*, J_2, \ldots, J_N$)
  Set $\hat{\beta}_N \leftarrow b_N^*$
  for $k = N-1, N-2, \ldots, 1$ **do**
    if $\hat{\beta}_{k+1} \in J_{k+1}$ **then**
      $\hat{\beta}_k \leftarrow b_{k+1}^*$
    else
      $\hat{\beta}_k \leftarrow \hat{\beta}_{k+1}$
    end if
  end for
  return $(\hat{\beta}_1, \ldots, \hat{\beta}_N)$
end function

function SEGMENT($\lambda_2, e_1(), \ldots, e_N()$)
  $\delta_1() \leftarrow e_1()$
  for $k = 1, 2, \ldots, N-1$ **do**
    $m_{k+1}^*, b_{k+1}^* \leftarrow$ Maximize($\delta_k()$)
    $J_{k+1}, f_{k+1} \leftarrow$ Flood($\delta_k(), m_{k+1}^* - \lambda_2$) ▷ See Algorithm 4
    $\delta_{k+1}() \leftarrow f_{k+1}() + e_{k+1}()$
  end for
  $\hat{\beta} \leftarrow$ Backtrace($b_1^*, \ldots, b_N^*, J_2, \ldots, J_N$)
end function

Algorithm 4 "DpSegPen" algorithm (Part 2)

39: **function** FLOOD($\delta_k()$, T)
40: $f_{k+1}() = ()$ ▷ initialize as empty sequence
41: $J_{k+1} = \{\}$ ▷ initialize as empty
42: **if** $c_1 + \sum_{i=1}^{R} a_{1i} v_i(x_1) < T$ **then**
43: $S \leftarrow$ TRUE ▷ set state as below threshold T
44: Append knot to $f_{k+1}()$: $(x_1, T, (0)_{i=1}^{R})$
45: $u \leftarrow -\infty$ ▷ the next interval in J_{k+1} has this left end-point
46: **else**
47: $S \leftarrow$ FALSE ▷ set state as above threshold T
48: Append knot to $f_{k+1}()$: $(x_1, T, (0)_{i=1}^{R})$
49: **end if**
50: **for** $t = 1, \ldots, m(k)$ **do** ▷ loop over segments
51: **if** $S =$ TRUE **then**
52: Solve: $\tilde{b}$ s.t. $c_t + \sum_{i=1}^{R} a_{ti} v_i(\tilde{b}) = T$ and $\sum_{i=1}^{R} a_{ti} v_i'(\tilde{b}) > 0$
53: **if** $\tilde{b}$ exists and $\tilde{b} < x_{t+1}$ **then**
54: Append knot to $f_{k+1}()$: $(\tilde{b}, c_t, (0)_{i=1}^{R})$
55: $S \leftarrow$ FALSE ▷ set state as above threshold T
56: $J_{k+1} \leftarrow J_{k+1} \cup \{[u, \tilde{b}]\}$ ▷ add interval
57: **else if** $t = m(k)$ **then**
58: Append $J_{k+1} \leftarrow J_{k+1} \cup \{[u, +\infty]\}$
59: ▷ We are below the threshold T in the final segment
60: **end if**
61: **else**
62: Append knot to $f_{k+1}()$: $(x_t, c_t, (a_{ti})_{i=1}^{R})$
63: **end if**
64: **if** $S =$ FALSE **then**
65: Solve: $\tilde{b}$ s.t. $c_t + \sum_{i=1}^{R} a_{ti} v_i(\tilde{b}) = T$ and $\sum_{i=1}^{R} a_{ti} v_i'(\tilde{b}) < 0$
66: **if** $\tilde{b}$ exists and $\tilde{b} < x_{t+1}$ **then**
67: Append knot to $f_{k+1}()$: $(\tilde{b}, c_t, (0)_{i=1}^{R})$
68: $S \leftarrow$ TRUE ▷ set state as above below T
69: $u \leftarrow \tilde{b}$ ▷ the next interval in J_k has this left end-point
70: **end if**
71: **end if**
72: **end for** ▷ Add a final knot at the boundary of the parameter space
73: Add knot to $f_{k+1}()$: $(+\infty, -\infty, (0)_{i=1}^{R})$
74: **return** $(J_{k+1}, f_{k+1}())$
75: **end function**

in the left panel and the flood operation in the right panel. The solid gray and dotted black curves plot the functions $\delta_k()$ and $f_{k+1}()$, respectively.

When considering exponential family variations, it is important to note that the steps in Lines 52 and 65 cannot be performed in closed form—an extension to the same exponential families considered for the EFLSA would require efficient and numerically precise code to

perform this inversion. For example, a function of two arguments a and b that returns the value of y such that $y + a\log(y) = b$ would facilitate Poisson segmentation. The reviewer has pointed out that Newton's method may be used, but in our simulations we only consider least squares segmentation in timing comparisons because the implementation is easier to carry out.

An alternative formulation of the segmentation problem minimizes the squared error $\sum_{k=1}^{N}(y_k - \beta_k)^2$ subject to a constraint on the number of jumps: $M = \sum_{k=2}^{N} 1\{\beta_k \neq \beta_{k-1}\}$. The algorithm for this problem is similar, but requires $M+1$ simultaneous recursions (indexed by the subscript $j = 1, \ldots, M+1$). The recursion is similar to jump-penalized version:

$$b^*_{j,k} = \text{argmax}_b \delta_{j-1,k-1}(b) \qquad j = 1, \ldots, M+1 \tag{27}$$

$$J_{j,k} := \{b : \delta_{j,k-1}(b) \leq \delta_{j-1,k-1}(b^*_{j-1,k-1})\} \tag{28}$$

$$f_{j,k}(b) := \delta_{j,k-1}(b) \vee [\delta_{j-1,k-1}(b^*_{j,k})] \tag{29}$$

$$\delta_{j,k}(b) := e_k(b) + f_{j,k}(b). \tag{30}$$

The recursion has the edge cases $f_{k,k}(b) := \delta_{k-1,k-1}(b^*_{k,k})$ and $\delta_{1,k}(b) := \sum_{j=1}^{k} e_j(b)$.

Full pseudocode for this algorithm is omitted because the details are very similar to the jump-penalized algorithm. The piecewise quadratic functions $f_{j,k}()$ and sets $J_{j,k}$ are found through a "flood" operation much like that in Algorithm 4. The backtrace is presented in Algorithm 5.

Algorithm 5 "DpSegN" backtrace

```
Initialize T ← M + 1
β̂_N = argmax_b δ_{M+1,N}(b)
for k = N − 1, N − 2, ..., 1 do
    if T > 1 and β̂_{k+1} ∈ J_{T,k+1} then
        β̂_k ← b*_{T−1,k+1}
        T ← T − 1
    else
        β̂_k ← β̂_{k+1}
    end if
end for
```

The "DpSegN" algorithm and the algorithm described by Bellman (1961) are both examples of dynamic programming, but they take different approaches to this problem. Bellman's algorithm recursively defines following functions on the integers $1, \ldots, N$:

$$L^1(j) := \max_b \sum_{k=1}^{j} e_k(b) \qquad j = 1, \ldots, N-M \tag{31}$$

$$L^t(j) := \max_{i=t-1}^{t} \left(L^{t-1}(i) + \max_b \sum_{k=i+1}^{j} e_k(b) \right) \qquad t = 2, \ldots, M \tag{32}$$

$$L^{M+1}(N) := \max_{i=M}^{t} \left(L^M(i) + \max_b \sum_{k=i+1}^{j} e_k(b) \right). \tag{33}$$

Bellman's algorithm terminates with the calculation of $L^{M+1}(N)$. Through simple backward recursions, one may obtain the optimal K-jump segmentations for K less than $M+1$.

Each of the quantities $L^t(j)$ in Bellman's algorithm answers the question "Given that the tth segment's right end-point is at index j, where is its left end-point?" The analogous intermediate quantities in the DpSegN algorithm answer the question "Given that the tth segment contains index j and has value b, is the previous point in the same segment?" The former approach is flexible and straightforward to implement, but timing comparisons in the next section show that Bellman's algorithm is comparatively inefficient for the special cases DpSegN can handle.

4. TIMING COMPARISONS

This section gives timing comparisons in two sets of simulations. The first set of simulations tests algorithms that optimize the FLSA criterion, and the simulations vary the length of a piecewise-constant sequence with added Gaussian noise. The second set of simulations tests the performance of the proposed least squares segmentation algorithm, and these simulations vary both the sequence length and the number of segments in the noiseless signal.

The first set of simulations tests the performance of the proposed algorithm for the FLSA, which we refer to as "FLDP," and compares with alternative techniques. The algorithms were used to smooth simulated sequences of length N that contained four equal-length segments. Segment means were drawn iid from a mean-zero normal distribution with a variance of 4, and standard-normal iid noise was then added to generate the observed sequence $y_1, \ldots, y_N$.

Table 1 gives timing results for varying sequence lengths N using three algorithms. All algorithms were fit for the single tuning parameter $\lambda_2 = \log N$. These simulations were run on a Linux server with a 3.00 GHz Intel Xeon CPU. All of the tested algorithms are implemented in C or C++ and called through an R interface, and we believe that the majority of computation time is spent within the C code rather than the R interface.

We adapt the algorithm presented by Kim et al. (2009) to solve the FLSA, and the running time is listed under the heading "QP." The algorithm was terminated when it reached a

Table 1. Timing results for fitting sequences of length N fit at $\lambda_2 = \log N$. Our proposed algorithm is listed as "FLDP," a modified version of the interior point method by Kim et al. (2009) is listed under "QP," and the algorithm by Pollak, Willsky, and Huang (2005) and Hofling (2009a) is listed under "flsa." The ratio of timing of "FLDP" and "flsa" is given in the bottom row

N (in 1000s)	1	10	20	50	100	500	1000
FLDP	0.00	0.00	0.00	0.00	0.01	0.03	0.06
flsa	0.00	0.03	0.06	0.20	0.49	3.30	7.30
QP	0.01	0.13	0.23	0.69	1.53		
flsa/FLDP	25.45	47.64	50.83	49.49	68.54	94.86	123.76

Table 2. Timing results for least squares segmentation algorithms in seconds. Timing is in seconds and rounded to two digits. The columns are described in the text. The DpSegPen algorithm was fit with penalty $\lambda_2 = \log(N)$ rather than a fixed number of jumps

No. of jumps	N	DpSegPen	DpSegN	Bellman
2	1000	0	0	0.01
2	10,000	0.02	0.04	1.75
2	100,000	0.14	0.3	179.22
3	1000		0.01	0.03
3	10,000		0.05	3.54
3	100,000		0.57	288.01
5	1000		0	0.06
5	10,000		0.06	5.35
5	100,000		0.9	522.06

duality gap of 10^{-4} (the "duality gap" is an upper bound on $\max_\beta s(\beta) - s(\hat{\beta})$ where $s()$ is the FLSA objective function and $\hat{\beta}$ is the best fit found by the algorithm).

The "flsa" R package (Hofling 2009a) implements algorithms described by Hofling (2009b) and includes the algorithm by Pollak, Willsky, and Huang (2005) for the one-dimensional FLSA as a special case. The algorithm was shown to be moderately faster than coordinate ascent when applied to very long sequence lengths (Hofling 2009b). Timing results for this algorithm are listed under "flsa" in Table 1.

Although "flsa" is a path algorithm, it was only used to find the solution at a single tuning parameter λ_2; one strength of our approach is that it need not step through the entire solution path. If one requires the entire path, the "flsa" path algorithm has an apparent advantage; however, Table 1 shows that it may be faster to run the FLDP algorithm at many values of λ_2 than to evaluate the "flsa" algorithm even once.

The second set of simulations assesses performance of least squares segmentation algorithms. The simulation setup is the same as before, but now the number of equally spaced jumps is varied. In the tables we refer to the jump-penalized version of our algorithm by "DpSegPen" (with penalty $\lambda_2 = \log(N)$) and the fixed M-jump version by "DpSegN." Bellman's algorithm is listed as "Bellman." The running times of the "DpSegN" and Bellman's algorithm are expected to increase linearly with the number of jumps that each algorithm is set to locate. These simulations were run on a 2.8 GHz Core i2 laptop.

Table 2 suggests that our algorithm scales linearly with the sequence length. We are aware that the worst case is at least $O(N^2)$, but this and other experience suggest that $O(N)$ behavior is more typical in practice.

The two algorithms may also differ in terms of numerical stability. Although Bellman's algorithm and the DpSegN algorithm gave identical output up to machine precision in all simulations, we know that numerical errors may accumulate. It is likely that the accumulation of error is worse for our algorithm—Bellman's algorithm requires only simple summations whereas our algorithm depends on the accuracy of some square-root calculations.

5. CONCLUSIONS

Viewing the FLSA is the most probable path in a particular HMM that leads to a fast algorithm with provably linear, worst-case complexity. Although it lacks provable

computational complexity, our algorithm for least squares segmentation shows promising performance when compared against competing, exact algorithms for this problem.

Our approach to the FLSA can be extended beyond least squares, and it yields a single solution without stepping through the entire path. Furthermore, this solution can be quickly updated if observations arrive sequentially.

We have considered some modest extensions of this work which have been omitted from this article. The first is isotonic regression: the constraint that the solution be monotonically increasing (or decreasing). The proposed algorithm can be modified to take this constraint and still run in $O(N)$ time. A monotonicity constraint may also be imposed in the EFLSA algorithm.

Unimodal regression is another possible extension, but the resulting algorithm may have $O(N^2)$ complexity in the worst case. A referee has informed us that $O(N)$ algorithms for both isotonic and unimodal regression have already been developed by Stout (2008). We do not claim better performance or go into detail here, but we have provided implementations of both isotonic and unimodal regression in the online supplementary materials for the interested reader.

A second extension is the application of the dynamic programming approach to "tree structured" versions of the Fused Lasso. Again, the modifications required may result in $O(N^2)$ complexity in the worst case. We believe the complexity is lower in an average sense, but this is based merely on limited simulations.

A final direction for this work is the development of path algorithms for the EFLSA models under the (unproved) assumption that fused segments remain fused (call this the "fusion property"). One may do so by considering the FLDP algorithm and reducing to the case when each index k belongs to its own segment. One can then determine the path of b_k^- and b_k^+ as a function of λ_2 and locate the value of λ_2 at which the next pair of segments will fuse. We plan to implement these path algorithms and are working to develop a proof of the fusion property for the EFLSA.

SUPPLEMENTARY MATERIALS

Readme: Documentation of attached routines. (readme.txt)

Precompiled libraries: Precompiled C extensions for Linux and Windows platforms. (code/lib/, directory)

C code: C extensions for efficient implementation of FLSA and least squares segmentation. (code/c/, directory)

R code: R wrappers of C extensions, and R implementations of algorithms. (code/r/, directory)

Scripts: R code to generate figures and perform timing comparisons. (code/scripts/, directory)

Appendix: An online Appendix giving the statements and proofs of Theorems 1 and 2.

ACKNOWLEDGMENTS

The author was supported by the Stanford Genome Training Program (T32 HG000044) and a Ric Weiland fellowship. The author would like to thank Holger Hoefling, Nadine Hussaini, Hua Tang, Rob and Ryan Tibshirani,

Wing Wong, and Nancy Zhang for feedback and helpful comments while writing this article. The author also thanks Pei Wang for checking and helping to debug an earlier version of the algorithm while it was incorporated into the cghFLasso R package.

[Received September 2010. Revised February 2012.]

REFERENCES

Bellman, R. (1961), "On the Approximation of Curves by Line Segments Using Dynamic Programming," *Communications of the ACM*, 4, 284. [247,256]

Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. (1999), *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*, Cambridge: Cambridge University Press. [247]

Friedman, J., Hastie, T., Hofling, H., and Tibshirani, R. (2007), "Pathwise Coordinate Optimization," *Annals of Applied Statistics*, 1, 302–332. [246]

Hofling, H. (2009a), *flsa: Path Algorithm for the General Fused Lasso Signal Approximator*, R package version 1.03. Available at *http://CRAN.R-project.org/package=flsa*. [257,258]

Hofling, H. (2009b), "Topics in Machine Learning," Ph.D. dissertation, Stanford University. [246,258]

Kim, S. J., Koh, K., Boyd, S., and Gorinevsky, D. (2009), "L1 Trend Filtering," *SIAM Review*, 51, 339–360. [257]

Pollak, I., Willsky, A. S., and Huang, Y. (2005), "Nonlinear Evolution Equations as Fast and Exact Solvers of Estimation Problems," *IEEE Transactions on Signal Processing*, 53, 484–498. [246,257,258]

Rabiner, L. R. (1989), "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," in *Proceedings of the IEEE*, pp. 257–286. [247]

Stout, Q. F. (2008), "Unimodal Regression via Prefix Isotonic Regression," *Computational Statistics and Data Analysis*, 53, 289–297. [259]

Tibshirani, R., and Wang, P. (2008), "Spatial Smoothing and Hot Spot Detection for CGH Data Using the Fused Lasso," *Biostatistics*, 9, 18–29. [252]