2016-03-15

Numerically Sane Construction of Polynomial Least Squares Filters via QR Factorization

Borges, Carlos F.
Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/48144
Numerically Sane Construction of Polynomial Least Squares Filters via QR Factorization

Carlos F. Borges

March 15, 2016

Abstract

We carefully develop a numerically sane algorithm for constructing polynomial least squares filters using the QR factorization. Filters of this type appear in a range of practical techniques and are extremely important. Many published methods for constructing these filters rely on algorithms and derivations that are numerically unwise, that are unnecessarily complex, or that are simply incorrect. Moreover there is a lack of proper historical context. We address both issues here.

1 Introduction

In 1815 Gergonne [2] first proposed the following problem:

Given an ordered sequence of \( m + 1 \) distinct points \( x_0 < x_1 < \ldots < x_m \) (called the abcissas or nodes) and a corresponding sequence of values \( y_0, y_1, \ldots, y_m \) (called the ordinates or observations), find a polynomial \( p(x) \) of specified degree \( n < m \) that minimizes the sum of squares

\[
\sum_{i=0}^{m} (y_i - p(x_i))^2
\]

This problem goes by various names (polynomial regression, discrete least squares polynomial approximation, etc.) but I shall refer to it as Gergonne Approximation (GA) in honor of the man who first proposed it. The polynomial \( p(x) \) is known as the least-squares polynomial and Gergonne proposed to use it as a way to smooth noisy data and as a method to estimate derivatives from data by evaluating \( p^{(k)}(x) \).

*Department of Applied Mathematics, Naval Postgraduate School, Monterey CA 93943
Gergonne [2] proceeds in a very recognizable manner. He selects the power basis so that the polynomial takes the form \( p(x) = a_0 + a_1x + \ldots + a_nx^n \) with unknown coefficients \( a_i \) and proceeds to minimize

\[
\sum_{i=0}^{m} (y_i - \sum_{j=0}^{n} a_jx_j^i)^2
\]

using the normal equations approach developed by Legendre and Gauss (see [5] for a beautifully written and detailed history of least-squares). We can summarize this using standard notation from linear algebra by letting

\[
V = \begin{bmatrix}
1 & x_0 & \ldots & x_0^n \\
1 & x_1 & \ldots & x_1^n \\
\vdots & \vdots & \ddots & \vdots \\
1 & x_m & \ldots & x_m^n
\end{bmatrix}, \quad a = \begin{bmatrix}
a_0 \\
a_1 \\
\vdots \\
a_n
\end{bmatrix}, \quad \text{and} \quad y = \begin{bmatrix}
y_0 \\
y_1 \\
\vdots \\
y_m
\end{bmatrix}
\]

and then noting that the normal equations are

\[
V^T V a = V^T y
\]

and hence the *algebraic* solution to the least squares problem is given by

\[
a = (V^T V)^{-1} V^T y
\]

This is precisely how Gergonne proposed to solve the problem more than 150 years ago (although the matrix notation for this problem was not introduced until Aitken [1] in 1934). The least-squares polynomial, \( p(x) \) is then given by

(2)

\[
p(x) = a^T x
\]

where we define \( x = \begin{bmatrix} 1 & x & \ldots & x^n \end{bmatrix}^T \).

It is critical to note that for a fixed value of \( x \) this can be written in the form

\[
p(x) = y^T f_x
\]

where

(3)

\[
f_x = V(V^T V)^{-1} x
\]

This form separates the influence of the location of the nodes and the particular value of \( x \) from that of the ordinates, \( y_i \) and expresses \( p(x) \) in the form of a linear functional or filter. It is a very convenient form for repetitive calculations where the nodes and the value of \( x \) are fixed but the ordinates are changing. And it has the added benefit that \( f_i \), the \( i \)'th element of \( f_x \), gives a direct measure of the sensitivity of the outputs to changes in the ordinates since clearly

\[
\frac{\partial p(x)}{\partial y_i} = f_i
\]
The form in 3 has been used for many years to construct these filters but it is well-known to be susceptible to serious numerical difficulties. Sadly, it still persists in a large number of widely available codes to this day. As an example, consider constructing a Savitzky-Golay \[4\] smoothing filter with 31 points and order 4. In this case the nodes are generally taken to be \(-15, -14, ..., 14, 15\) and the condition number of \(V^T V\) is roughly \(7 \times 10^{60}\) which is a remarkably ill-conditioned \(5 \times 5\) matrix.

A common generalization of this problem is to introduce a set of strictly positive weights, \(w_i > 0\), and to replace 1 with

\[
\sum_{i=0}^{m} w_i^2(y_i - p(x_i))^2
\]

if we let

\[
W = \begin{bmatrix}
w_0 & 0 & 0 & ... & 0 \\
0 & w_1 & 0 & ... & 0 \\
& ... & ... & ... & ...
\end{bmatrix}
\]

then the weighted normal equations are

\[
(WV)^T WV a = (WV)^T Wy
\]

If we now let \(\hat{V} = WV\) then the algebraic solution can be written in the general form

\[
a = (\hat{V}^T \hat{V})^{-1} \hat{V}^T W y
\]

which gives the weighted filter in the form

\[
f_x = W \hat{V} (\hat{V}^T \hat{V})^{-1} x
\]

Note that as \(W\) is diagonal the transposes have been dropped.

\section{Using QR to find the filter}

Although the formula in 3 is a numerically unwise approach to solving the problem it can be largely rescued by using the QR factorization of \(\hat{V}\). This approach was first proposed by Gram \[3\] in 1883 where he introduced what we now call the modified Gram-Schmidt algorithm. This allows us to orthogonally triangularize \(\hat{V}\) in \(O(mn^2)\) flops. Since \(\hat{V} = QR\) and \(Q\) has orthonormal columns we can rewrite 5 as

\[
f_x = WQR^{-T} \begin{bmatrix} 1 \\
x \\
\vdots \\
x^n \end{bmatrix}
\]

Of course one would never form the inverse and in practice the solution follows a three step process:
1. Construct the skinny QR factorization of $\hat{V}$ using an appropriate algorithm (e.g. modified Gram-Schmidt).

2. Use forward elimination to solve

$$R^T u = \begin{bmatrix} 1 \\ x \\ \vdots \\ x^n \end{bmatrix}$$

3. Set

$$f_x = W(Qu)$$

where the parentheses have been added to emphasize the computational importance of applying the diagonal weighting operator to a vector (which can be done with only $m$ multiplies) rather than applying it to an $m \times n$ matrix (which requires $mn$ multiplies).

We should note at this point that if $x$ is one of the nodal values, that is $x = x_i$ for some $i$, then the process can be further simplified since then $u$ can be recovered directly from $Q$ thus saving the need for the forward elimination in step 2 (and, in fact, eliminating the need to form $R$ in step 1). In particular, since $[w_i \ w_i x_i \ \cdots \ w_i x_i^n]$ is the $i$’th row of $\hat{V}$ it follows that $u^T$ is simply the $i$’th row of $Q$ scaled by $1/w_i$. This is a very common situation and it is worth noting that the entire collection of filters, that is filters for each of the $x_i$ can be constructed simultaneously with the following three-step process:

1. Construct an orthonormal basis $Q$ for the column space of $\hat{V}$ using an appropriate algorithm (e.g. modified Gram-Schmidt where no effort is made to keep $R$).

2. We can then construct a matrix of filters whose $i$’th column is identically $f_{x_i}$ by

$$WQQ^TW^{-1}$$

3 Computing Derivative Filters

If we differentiate 2 with respect to $x$ we find that

$$p'(x) = a^T \begin{bmatrix} 0 \\ 1 \\ 2x \\ \vdots \\ nx^{n-1} \end{bmatrix}$$
and more generally that derivatives of order $k$ with $1 \leq k \leq n$ are given by

\begin{equation}
    p^{(k)}(x) = a^T \begin{bmatrix}
        0 \\
        (1)_k x^{1-k} \\
        \vdots \\
        (n)_k x^{n-k}
    \end{bmatrix}
\end{equation}

where $(x)_k$ is the Pochhammer symbol or falling factorial defined for any positive integer $k$ by

\[ (x)_k = x(x-1)(x-2) \ldots (x-k+1) \]

and we see that the $k$’th derivative filter is given by

\begin{equation}
    f^{(k)}_x = WQR^T \begin{bmatrix}
        0 \\
        (1)_k x^{1-k} \\
        \vdots \\
        (n)_k x^{n-k}
    \end{bmatrix}
\end{equation}

4 Useful Changes of Variable

There are two obvious situations where a change of variables may be useful when constructing filters. The first is to take advantage of the fact that the right hand side of 6 and 7 are much simplified when the particular value of $x$ we are constructing the filter for is $x = 0$. To this end one may simply shift the nodes to $\tilde{x}_i = x_i - x$ which simplifies all of the calculations but causes no change in the values of the filters.

There can also, at times, be problems with scaling in the matrix $V$ depending on the values of the $x_i$ and the degree of the filter. This only appears in fairly extreme cases (e.g. when the magnitudes of the values in $V$ vary by more than $10^{15}$) and is not usually an issue. However, in those cases where it is a problem we can often improve the solution by making a change of variables that maps the $x_i$ linearly into the interval $[-1,1]$. Specifically, let

\[ u = \frac{2}{x_m - x_0} \left( x - \frac{x_m + x_0}{2} \right) \]

We can then construct filters with respect to the new variables and avoid many of the scaling problems. The coefficients for smoothing filters are unaffected by this linear change of variable but derivative filters need to be appropriately rescaled using the chain rule

\[ \frac{d^k}{dx^k} p(u) = \frac{d^k}{du^k} p(u) \left( \frac{2}{x_m - x_0} \right)^k \]
5 MATLAB Code and Testing

We have written and tested a few MATLAB codes implementing the methods described above. The first is a general code that allows the user to submit a set nodes, a point \( x \), and the degree of the polynomial approximation. The code then generates a filter that approximates the derivative of a user specified order. The user can also optionally specify a vector of positive weights to find the weighted filter. The code is as follows:

```matlab
function \[f\] = makefilterqr\(x,\text{nodes},\text{degree},\text{dk},w\)
% A code for constructing polynomial filters using QR. The inputs
% are:
% - \( x \) is the point at which the filter generates estimates.
% - \( \text{nodes} \) is the array of \( N \) nodal points.
% - \( \text{degree} \) is the polynomial degree of the filter, it must be less than \( N \).
% - \( \text{dk} \) is the order of the derivative that the filter estimates. If this
% is missing then \( \text{dk} \) is set to 0 and a smoother is constructed.
% - \( w \) is the optional weight vector. Must be same size as \( \text{nodes} \), all positive.
% The output is \( f \) which contains the desired filter.
% Created on 14 MARCH 2016 by Carlos F. Borges.

% Error checking. WARNING: This is minimal.
N = length\(\text{nodes}\);
if degree >= N
    error('Degree exceeds available data.');
end
if nargin < 4
    dk = 0;
end
if dk > degree
    error('Cannot differentiate to a higher degree than the approximation.');
end

% Construct the Vandermonde matrix of shifted nodes.
V = repmat\(\text{nodes}(:)-x,\text{degree}+1\); V(:,1) = 1;
V = cumprod\(V,2\);

% Construct the appropriate polynomial power basis vector.
px = zeros\(\text{degree}+1,1\);
px\(1+\text{dk}\) = factorial\(\text{dk}\);

% The solution comes in two flavors.
```

% The solution comes in two flavors.
if nargin < 5 % This is the unweighted solution.

    % Construct the skinny QR factorization of the Vandermonde.
    [Q,R] = qr(V,0);

    % Construct the filter.
    f = Q*(R'\px);

else % This is the weighted solution.

    w = w(:); % Force w to be a column.
    % Apply the weighting to the rows of the Vandermonde.
    V = bsxfun(@times,V,w); % Fast equivalent to V = diag(w)*V;

    % Construct the skinny QR factorization of the weighted Vandermonde.
    [Q,R] = qr(V,0);

    % Construct the filter.
    f = w.*(Q*(R'\px));

end

The second code we developed essentially mimics the already existing MATLAB sgolay code. The major difference is that this code asks the user to enter the half-width of the filter rather than the full width. This is done to be more consistent with actual practice and because it eliminates the need to enforce an entered full width that is odd. Furthermore, this code has the user enter the weights as opposed to the squared weights as required by the MATLAB sgolay code. The code has been extensively tested against the MATLAB sgolay code. We note that testing shows that it is not only faster but that it will accurately construct filters of greater width and higher degree than the MATLAB sgolay code. Moreover, testing shows the filters created by this code are several orders of magnitude more accurate. To see this consider that the columns of the smoother matrix $B$ should each sum to 1. When we test this condition we observe that the code presented below is orders of magnitude more accurate.

We note that our testing against the MATLAB sgolay code was limited to unweighted filters. This is because there are errors in the MATLAB sgolay code and it often gives incorrect answers for weighted problems. This can be most easily observed by using uniform weighting that is not identically 1 which should give the same answer as an unweighted filter but in the MATLAB sgolay code it does not.

function [B,G] = sgolayqr(degree,hw,w)
% A code for constructing Savitzky-Golay filters using QR. The inputs
% are not quite the same as the Matlab sgolay code but the outputs are.
%
Sane Construction of Polynomial Least Squares Filters via QR

- degree is the degree of the filter, it must be less than 2*hw.
- hw is the half-width of the filter so that it has 2*hw+1 total points.
- w is the vector of weights. They must be 2*hw+1 of them, all positive.
- We assume that the nodes are the integers -hk to hk and that the midpoint is therefore 0.
- WARNING: There is NO error checking.

% Construct the Vandermonde matrix of nodes.
V = repmat((-hw:hw)',1,degree+1); V(:,1) = 1;
V = cumprod(V,2);

% The solution comes in two flavors.
if nargin == 2 % This is the unweighted solution.
    % Construct the skinny QR factorization of the Vandermonde.
    [Q,R] = qr(V,0);
    % Construct the matrix of smoothers at all of the nodal points.
    B = Q*Q';
    % Construct the smoother and all differentiators at x = 0.
    G = Q/R';
else % This is the weighted solution.
    w = w(:); % Force w to be a column.
    % Apply the weighting to the rows of the Vandermonde.
    V = bsxfun(@times,V,w); % Fast equivalent to V = diag(w)*V;
    % Construct the skinny QR factorization of the weighted Vandermonde.
    [Q,R] = qr(V,0);

    % To construct the matrix of smoothers and differentiators we essentially need the following calculations.
    % B = diag(w)*Q*Q'*diag(1./w); % Construct the matrix of smoothers.
    % G = diag(w)*Q/R'; % Construct the smoother and differentiators at the midpoint.
    % However, this can be done more efficiently as follows:
    G = bsxfun(@times,Q,w); % Faster calculation.
    B = G*bsxfun(@rdivide,Q,w); % Calculate B.
    G = G/R'; % Calculate G.
end

References


