Chapter 9. Smoothing Filters

W. Gander and U. von Matt

9.1 Introduction

In many applications one is measuring a variable that is both slowly varying and also corrupted by random noise. Then it is often desirable to apply a smoothing filter to the measured data in order to reconstruct the underlying smooth function. We may assume that the noise is independent of the observed variable. Furthermore we assume that the noise obeys a normal distribution with mean zero and standard deviation δ.

In this chapter we will discuss two different approaches to this smoothing problem, the Savitzky-Golay filter and the least squares filter. We will analyze the properties of these two filters with the help of the test function

$$F(x) := e^{-100(x-1/5)^2} + e^{-500(x-2/5)^2} + e^{-2500(x-3/5)^2} + e^{-12500(x-4/5)^2}. \tag{9.1}$$

This function has four bumps of varying widths (cf. Figure 9.1).

In MATLAB we can generate a vector $\mathbf{f}$ of length $n = 1000$ consisting of measured data corrupted by random noise of standard deviation $\delta = 0.1$ by the statements of Algorithm 9.1. We get the sample data shown as Figure 9.2.

In the following sections we will denote the measured data by f_i, $i = 1, \ldots, n$, and the smoothed data by g_i, $i = 1, \ldots, n$.

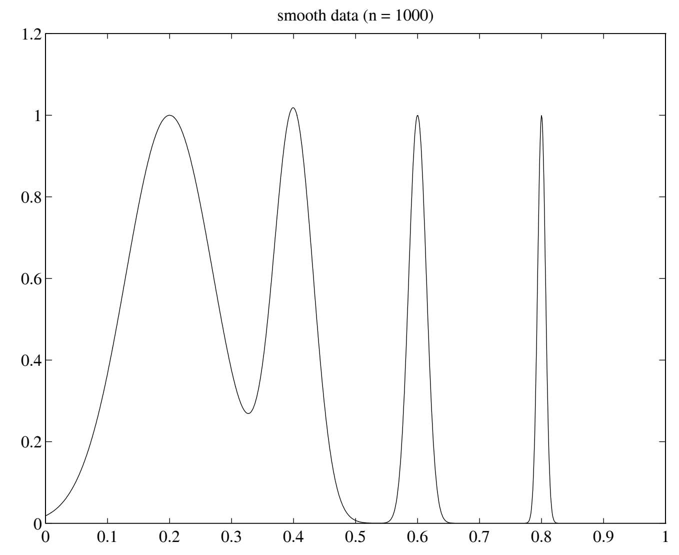
9.2 Savitzky-Golay Filter

This approach to smoothing has been introduced by A. Savitzky and M.J.E. Golay in 1964 [10]. The original paper contains some errors that are corrected in [11]. The reader can also find another introduction to this subject in [8].

ALGORITHM 9.1. *Generation of Noisy Data.*

```
n = 1000;
delta = 0.1;
x = [0:n-1]'/(n-1);
F = exp (- 100*(x - 1/5).^2) + exp (-  500*(x - 2/5).^2) + ...
    exp (-2500*(x - 3/5).^2) + exp (-12500*(x - 4/5).^2);
randn ('seed', 0);
f = F + delta * randn (size (x));
```

FIGURE 9.1. *Smooth Function $F(x)$.*



The key idea is that the smoothed value g_i in the point x_i is obtained by taking an average of the neighboring data. The simplest method consists in computing a moving average of a fixed number of f_i's.

More generally we can also fit a polynomial through a fixed number of points. Then the value of the polynomial at x_i gives the smoothed value g_i. This idea is also shown as Figure 9.3, where n_L denotes the number of points to the left of x_i, and n_R denotes the number of points to the right of x_i. By $p_i(x)$ we denote a polynomial of degree M which is fit in the least squares sense through the $n_L + n_R + 1$ points. Then, we have $g_i = p_i(x_i)$.

9.2.1 Filter Coefficients

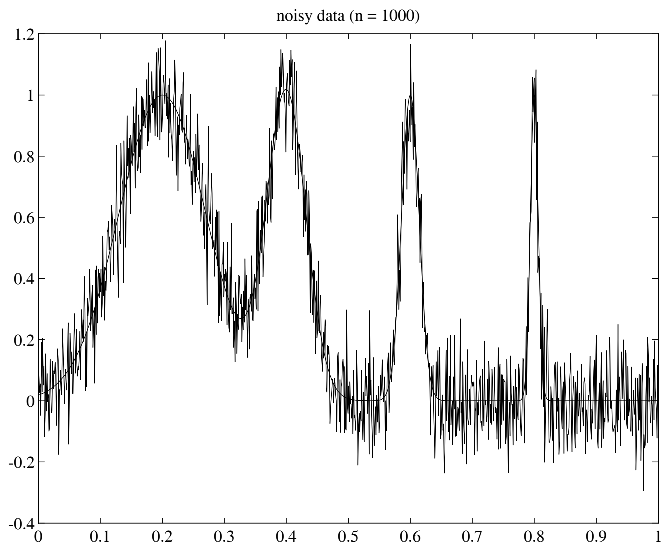
The polynomial $p_i(x)$ of degree M, which is to be fitted through the data f_i, can be written as

$$p_i(x) := \sum_{k=0}^{M} b_k \Big(\frac{x - x_i}{\Delta x}\Big)^k. \tag{9.2}$$

We assume that the abscissas x_i are uniformly spaced with $x_{i+1} - x_i \equiv \Delta x$. In order to fit $p_i(x)$ in the least squares sense through the measured data we have to determine the coefficients b_k such that

$$\sum_{j=i-n_L}^{i+n_R} (p_i(x_j) - f_j)^2 = \min. \tag{9.3}$$

FIGURE 9.2. *Noisy Function $f(x)$.*



Let us define the matrix

$$A := \begin{bmatrix} (-n_L)^M & \dots & -n_L & 1 \\ \vdots & & \vdots & \vdots \\ 0 & \dots & 0 & 1 \\ \vdots & & \vdots & \vdots \\ n_R^M & \dots & n_R & 1 \end{bmatrix} \in \mathbf{R}^{(n_L+n_R+1)\times(M+1)} \tag{9.4}$$

and the two vectors

$$\mathbf{b} := \begin{bmatrix} b_M \\ \vdots \\ b_1 \\ b_0 \end{bmatrix} \in \mathbf{R}^{M+1} \tag{9.5}$$

and

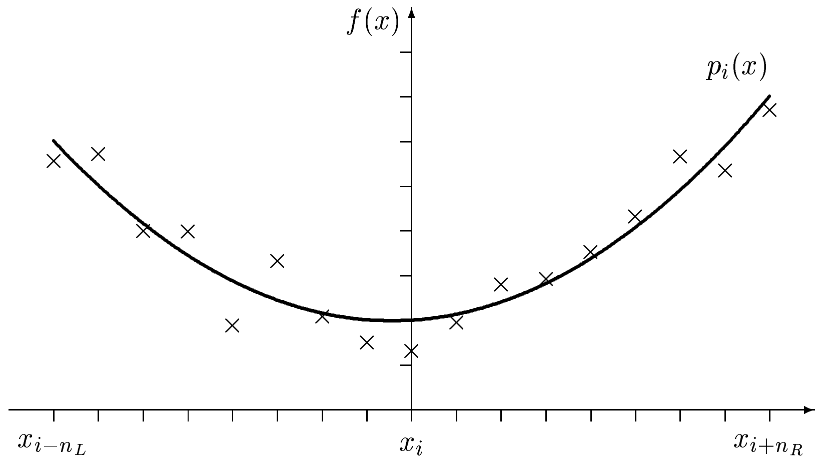
$$\mathbf{f} := \begin{bmatrix} f_{i-n_L} \\ \vdots \\ f_i \\ \vdots \\ f_{i+n_R} \end{bmatrix} \in \mathbf{R}^{n_L+n_R+1}. \tag{9.6}$$

It should be noted that the matrix A neither depends on the abscissa x_i nor on the spacing Δx.

Using these definitions we can restate the least squares problem (9.3) in matrix terms as

$$\|A\mathbf{b} - \mathbf{f}\|_2 = \min\,. \tag{9.7}$$

FIGURE 9.3. *Least Squares Polynomial* $p_i(x)$.



It would now be possible to solve (9.7) for $\mathbf{b}$ by means of the QR-decomposition of A (cf. [5, Chapter 5]). For our filtering purpose, however, we only need to know $g_i = p_i(x_i) = b_0$. The solution $\mathbf{b}$ of (9.7) can also be expressed as the solution of the normal equations

$$A^{\mathrm{T}} A\mathbf{b} = A^{\mathrm{T}}\mathbf{f}. \tag{9.8}$$

Thus, we get

$$g_i = \mathbf{e}_{M+1}^{\mathrm{T}}(A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}\mathbf{f}, \tag{9.9}$$

where $\mathbf{e}_{M+1}$ denotes the $(M+1)$st unit vector.

Obviously, we can represent g_i as a linear combination of the f_i's. We define the vector

$$\mathbf{c} := A(A^{\mathrm{T}}A)^{-1}\mathbf{e}_{M+1} \tag{9.10}$$

containing the filter coefficients $c_{-n_L}, \ldots, c_{n_R}$. Since $\mathbf{c}$ does not depend on x_i and Δx, it only needs to be evaluated once. Then, all the smoothed values g_i can be computed by the simple scalar product

$$g_i = \mathbf{c}^{\mathrm{T}}\mathbf{f} = \sum_{j=i-n_L}^{i+n_R} c_{j-i} f_j. \tag{9.11}$$

The calculation of the vector $\mathbf{c}$ according to Equation (9.10) may be inaccurate for large values of M. This loss of accuracy can be attributed to the fact that the condition number of A is squared in forming $A^{\mathrm{T}}A$. On the other hand it is well known that the least squares system (9.7) can be stably solved by means of the QR-decomposition

$$A = QR. \tag{9.12}$$

By Q we denote an orthogonal $(n_L + n_R + 1)$-by-$(M+1)$ matrix, and by R we denote an upper triangular $(M+1)$-by-$(M+1)$ matrix. If we substitute

ALGORITHM 9.2. *Savitzky-Golay Smoothing Filter.*

```
function g = SavGol (f, nl, nr, M)

A = ones (nl+nr+1, M+1);
for j = M:-1:1,
  A (:, j) = [-nl:nr]' .* A (:, j+1);
end
[Q, R] = qr (A);
c = Q (:, M+1) / R (M+1, M+1);

n = length (f);
g = filter (c (nl+nr+1:-1:1), 1, f);
g (1:nl) = f (1:nl);
g (nl+1:n-nr) = g (nl+nr+1:n);
g (n-nr+1:n) = f (n-nr+1:n);
```

decomposition (9.12) into (9.10) we get for the vector **c** the expression

$$\mathbf{c} = \frac{1}{r_{M+1,M+1}} Q\mathbf{e}_{M+1}. \tag{9.13}$$

This is the numerically preferred way of computing **c**. Amazingly enough, this has been pointed out neither in [8] nor in [10].

9.2.2 Results

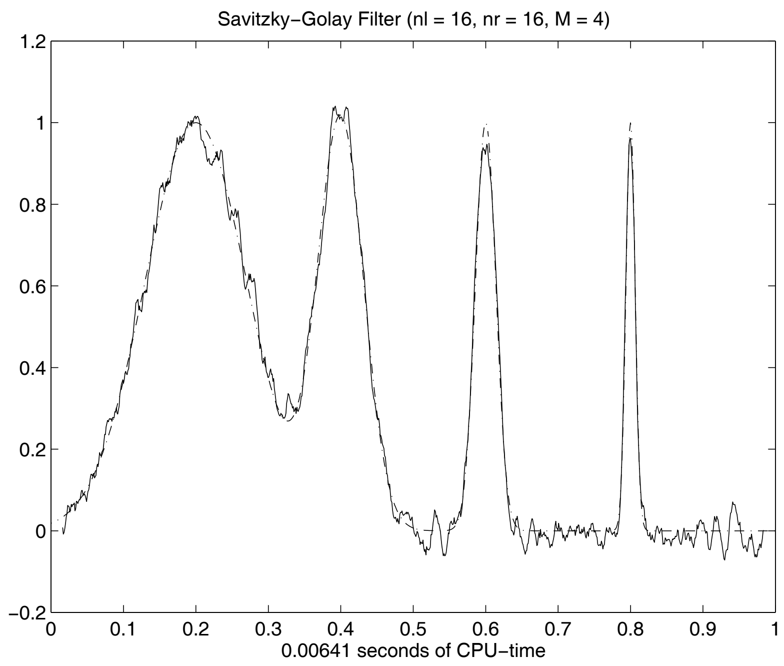
We present the Savitzky-Golay smoothing filter as Algorithm 9.2. If we apply this algorithm to our initial smoothing problem from Figure 9.2, we get the smoothed curve in Figure 9.4. For easy reference we have also superimposed the graph of the function F from Equation (9.1). We have chosen the parameters $n_L = n_R = 16$ and $M = 4$ which seem to be optimal for this test case. The execution of Algorithm 9.2 needs about 6.41 milliseconds of CPU-time on a PC with an Intel Pentium Pro running at 200 MHz.

The main advantage of a Savitzky-Golay filter is its speed. For given values of n_L, n_R, and M, the filter parameters **c** need to be evaluated only once. Then each filtered value g_i can be computed by the simple scalar product (9.11) of length $n_L + n_R + 1$. It is conceivable that this operation could even be implemented in hardware for special purpose applications.

It is a disadvantage that it is not obvious how to choose the filter parameters n_L, n_R, and M. In [1, 8, 13] some practical hints are given. But in many cases some visual optimization is needed in order to obtain the best results.

Finally, the boundary points represent yet another problem. They cannot be smoothed by a filter with $n_L > 0$ or $n_R > 0$. Either these boundary values are just dropped, as it is done in the case of Figure 9.4, or one constructs a special Savitzky-Golay filter with $n_L = 0$ or $n_R = 0$ for these special cases.

FIGURE 9.4.
Data of Figure 9.2 Smoothed by a Savitzky-Golay Filter.



9.3 Least Squares Filter

Another approach to our filtering problem consists in requiring that the filtered curve $g(x)$ be as smooth as possible. In the continuous case we would require that

$$\int_{x_{\min}}^{x_{\max}} g''(x)^2 \, dx = \min . \tag{9.14}$$

Since the function $f(x)$ is only available as measured data f_i at discrete points x_i we will only compute the smoothed values g_i at the same points. Therefore, we must express the second derivative of $g(x)$ by a finite difference scheme. A popular choice is

$$g''(x_i) \approx \frac{g_{i+1} - 2g_i + g_{i-1}}{\Delta x^2}. \tag{9.15}$$

We could give similar stencils for non-equally spaced abscissas x_i. Consequently, condition (9.14) is replaced by the condition

$$\sum_{i=2}^{n-1} (g_{i+1} - 2g_i + g_{i-1})^2 = \min . \tag{9.16}$$

Besides this smoothness condition we also require that the function $g(x)$ approximates $f(x)$ within the limits of the superimposed noise. If we assume the f_i's to be corrupted by random noise of mean zero and standard deviation δ

we require that

$$|g_i - f_i| \leq \delta \tag{9.17}$$

on the average. For n samples this condition can also be written as

$$\sum_{i=1}^{n} (g_i - f_i)^2 \leq n\delta^2. \tag{9.18}$$

Let us now define the matrix

$$A := \begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \end{bmatrix} \in \mathbf{R}^{(n-2)\times n}. \tag{9.19}$$

Then, we can restate the optimization problem (9.16,9.18) in matrix notation as the minimization of

$$\|A\mathbf{g}\|_2 = \min \tag{9.20}$$

subject to the quadratic inequality constraint

$$\|\mathbf{g} - \mathbf{f}\|_2^2 \leq \alpha^2 := n\delta^2. \tag{9.21}$$

9.3.1 Lagrange Equations

Let us now study the solution of the optimization problem (9.20,9.21). First we assume that there are elements from the null space $\mathcal{N}(A)$ of the matrix A satisfying the constraint (9.21). In this case we want to determine the vector $\mathbf{g}$ which satisfies the constraint (9.21) best, i.e. we consider the minimization problem

$$\|\mathbf{g} - \mathbf{f}\|_2 = \min \tag{9.22}$$

subject to the linear equality constraint

$$A\mathbf{g} = \mathbf{0}. \tag{9.23}$$

It is not hard to show that the unique optimal solution in this case is given by

$$\mathbf{g} = \mathbf{f} - A^{\mathrm{T}}\mathbf{z}, \tag{9.24}$$

where $\mathbf{z}$ can be computed as the unique solution of the linear least squares problem

$$\|A^{\mathrm{T}}\mathbf{z} - \mathbf{f}\|_2 = \min. \tag{9.25}$$

The vector $\mathbf{z}$ can also be characterized as the solution of the normal equations

$$AA^{\mathrm{T}}\mathbf{z} = A\mathbf{f}, \tag{9.26}$$

which correspond to (9.25).

Now, let us assume that there is no element from the null space $\mathcal{N}(A)$ which is compatible with the constraint (9.21). In particular, this means that the vector $\mathbf{g} \in \mathcal{N}(A)$ defined by the Equations (9.24) and (9.25) satisfies

$$\|\mathbf{g} - \mathbf{f}\|_2 > \alpha. \tag{9.27}$$

We can study the least squares problem (9.20,9.21) by introducing the Lagrange principal function

$$\Phi(\mathbf{g}, \lambda, \mu) := \mathbf{g}^{\mathrm{T}} A^{\mathrm{T}} A \mathbf{g} + \lambda(\|\mathbf{g} - \mathbf{f}\|_2^2 + \mu^2 - \alpha^2), \tag{9.28}$$

where μ denotes a so-called slack variable. By differentiating Φ with respect to $\mathbf{g}$, λ, and μ we get the Lagrange equations

$$(A^{\mathrm{T}} A + \lambda I)\mathbf{g} = \lambda \mathbf{f}, \tag{9.29}$$

$$\|\mathbf{g} - \mathbf{f}\|_2^2 + \mu^2 = \alpha^2, \tag{9.30}$$

$$\lambda \mu = 0. \tag{9.31}$$

For $\lambda = 0$, we would have $A^{\mathrm{T}} A \mathbf{g} = \mathbf{0}$ or $A\mathbf{g} = \mathbf{0}$, since the matrix A has full rank. But according to our above assumption, the vector $\mathbf{g}$ cannot be an element of the null space $\mathcal{N}(A)$. Therefore we will assume $\lambda \neq 0$ from now on.

Because of $\lambda \neq 0$, we have $\mu = 0$ from Equation (9.31). Therefore, we can simplify the Lagrange Equations (9.29,9.30,9.31) to

$$(A^{\mathrm{T}} A + \lambda I)\mathbf{g} = \lambda \mathbf{f}, \tag{9.32}$$

$$\|\mathbf{g} - \mathbf{f}\|_2 = \alpha. \tag{9.33}$$

Since $\lambda \neq 0$, we can make use of Equation (9.32) to express $\mathbf{g}$ by

$$\mathbf{g} = \mathbf{f} - A^{\mathrm{T}} \mathbf{z}, \tag{9.34}$$

where

$$\mathbf{z} = \frac{1}{\lambda} A \mathbf{g}. \tag{9.35}$$

By substituting the expression (9.34) for $\mathbf{g}$ into (9.32,9.33), we get the dual Lagrange equations

$$(A A^{\mathrm{T}} + \lambda I)\mathbf{z} = A \mathbf{f}, \tag{9.36}$$

$$\|A^{\mathrm{T}} \mathbf{z}\|_2 = \alpha. \tag{9.37}$$

As soon as λ and $\mathbf{z}$ have been determined, we can compute $\mathbf{g}$ according to Equation (9.34).

In [3, 4] it is shown that there is a unique Lagrange multiplier $\lambda > 0$ and an associated vector $\mathbf{z}$ which solve the dual Lagrange Equations (9.36,9.37). Furthermore, the vector $\mathbf{g}$ from (9.34) will then solve the least squares problem (9.20,9.21).

Algorithm 9.3.
Solution of the Constrained Least Squares Problem (9.20,9.21).

```
Solve the linear least squares problem (9.25) for $\mathbf{z}$.
**if** $\|A^{\mathrm{T}}\mathbf{z}\|_2 > \alpha$ **then**
    Solve the secular equation (9.38) for the unique zero $\lambda > 0$.
    Solve the linear system (9.36) for $\mathbf{z}$.
**end**
$\mathbf{g} := \mathbf{f} - A^{\mathrm{T}}\mathbf{z}$
```

The dual Lagrange Equations (9.36,9.37) have the advantage that the matrix $AA^{\mathrm{T}} + \lambda I$ is nonsingular for $\lambda \geq 0$. The Lagrange multiplier λ can be found by solving the nonlinear secular equation

$$s(\lambda) := \|A^{\mathrm{T}}(AA^{\mathrm{T}} + \lambda I)^{-1}A\mathbf{f}\|_2^2 = \alpha^2. \tag{9.38}$$

This will be discussed in more detail in Section 9.3.2.

As the result of our analysis we can present Algorithm 9.3 for the solution of the constrained least squares problem (9.20,9.21). The next sections will concentrate on the practical issues of the implementation in Matlab.

9.3.2 Zero Finder

The solution of the secular Equation (9.38) for its unique zero $\lambda > 0$ represents the major effort of Algorithm 9.3. We show the graph of the secular function $s(\lambda)$ as Figure 9.5. Since $s(\lambda)$ is a nonlinear function an iterative method is needed to solve the secular Equation (9.38). A good choice would be Newton's method which is defined by the iteration

$$\lambda_{k+1} := \lambda_k - \frac{s(\lambda_k) - \alpha^2}{s'(\lambda_k)}. \tag{9.39}$$

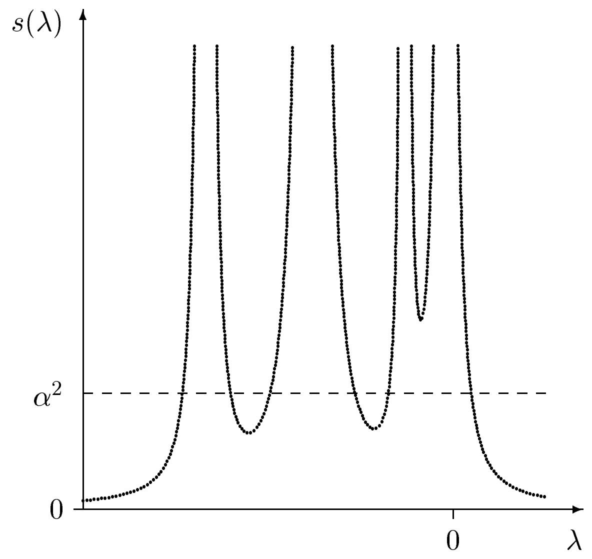
However, for our particular equation, Reinsch [9] has proposed the accelerated Newton iteration

$$\lambda_{k+1} := \lambda_k - 2\frac{s(\lambda_k)}{s'(\lambda_k)}\Big(\frac{\sqrt{s(\lambda_k)}}{\alpha} - 1\Big). \tag{9.40}$$

If this iteration is started with $\lambda_0 = 0$ we get a strictly increasing sequence of λ_k's. Proofs of this key property can be found in [9] and in [12, pp. 65–66]. However, this mathematical property cannot be preserved in floating-point arithmetic. This observation leads to the numerical termination criterion

$$\lambda_{k+1} \leq \lambda_k. \tag{9.41}$$

FIGURE 9.5. *Secular Function $s(\lambda)$.*



9.3.3 Evaluation of the Secular Function

In order to use the iteration (9.40) to solve the secular Equation (9.38), we need a way to evaluate numerically the secular function $s(\lambda)$ and its derivative $s'(\lambda)$. The values of $s(\lambda)$ and $s'(\lambda)$ can be expressed by

$$s(\lambda) = \|A^{\mathrm{T}}\mathbf{z}\|_2^2, \tag{9.42}$$

$$s'(\lambda) = -2\mathbf{z}^{\mathrm{T}}(\mathbf{z} + \lambda\mathbf{z}'), \tag{9.43}$$

where $\mathbf{z}$ and $\mathbf{z}'$ satisfy the equations

$$(AA^{\mathrm{T}} + \lambda I)\mathbf{z} = A\mathbf{f} \tag{9.44}$$

and

$$(AA^{\mathrm{T}} + \lambda I)\mathbf{z}' = -\mathbf{z}. \tag{9.45}$$

The last two Equations (9.44) and (9.45) suggest that we have to solve linear systems involving the matrix $AA^{\mathrm{T}} + \lambda I$. However, we can also read Equation (9.44) as the normal equations corresponding to the linear least squares problem

$$\left\| \begin{bmatrix} A^{\mathrm{T}} \\ \sqrt{\lambda} I \end{bmatrix} \mathbf{z} - \begin{bmatrix} \mathbf{f} \\ \mathbf{0} \end{bmatrix} \right\|_2 = \min. \tag{9.46}$$

Similarly, we can compute $\mathbf{z}'$ as the solution of the least squares problem

$$\left\| \begin{bmatrix} A^{\mathrm{T}} \\ \sqrt{\lambda} I \end{bmatrix} \mathbf{z}' - \begin{bmatrix} \mathbf{0} \\ -\mathbf{z}/\sqrt{\lambda} \end{bmatrix} \right\|_2 = \min, \tag{9.47}$$

provided that $\lambda > 0$. It should be noted that for $\lambda = 0$ the vector $\mathbf{z}'$ is not needed to compute $s'(\lambda)$ in Equation (9.43). Numerically, we prefer computing $\mathbf{z}$ and $\mathbf{z}'$

FIGURE 9.6. *First Stage of the QR-decomposition (9.48).*

$$\begin{bmatrix} \enclose{circle}{1} &&&& \\ -2 & 1 &&& \\ 1 & -2 & 1 && \\ & 1 & -2 & 1 & \\ && \ddots & \ddots & \ddots \\ \enclose{circle}{\sigma} &&&& \\ & \sigma &&& \\ && \sigma && \\ &&& \sigma & \\ &&&& \ddots \end{bmatrix} \longmapsto \begin{bmatrix} \enclose{circle}{r_{11}} &&&& \\ \enclose{circle}{-2} & 1 &&& \\ 1 & -2 & 1 && \\ & 1 & -2 & 1 & \\ && \ddots & \ddots & \ddots \\ 0 &&&& \\ & \sigma &&& \\ && \sigma && \\ &&& \sigma & \\ &&&& \ddots \end{bmatrix} \longmapsto$$

$$\begin{bmatrix} \enclose{circle}{r_{11}} & r_{12} &&& \\ 0 & r_{22} &&& \\ \enclose{circle}{1} & -2 & 1 && \\ & 1 & -2 & 1 & \\ && \ddots & \ddots & \ddots \\ 0 &&&& \\ & \sigma &&& \\ && \sigma && \\ &&& \sigma & \\ &&&& \ddots \end{bmatrix} \longmapsto \begin{bmatrix} r_{11} & r_{12} & r_{13} && \\ 0 & r_{22} &&& \\ 0 & t & r_{33} && \\ & 1 & -2 & 1 & \\ && \ddots & \ddots & \ddots \\ 0 &&&& \\ & \sigma &&& \\ && \sigma && \\ &&& \sigma & \\ &&&& \ddots \end{bmatrix}$$

from the two linear least squares problems (9.46,9.47) over solving the two linear systems (9.44,9.45).

We can solve the two least squares problems (9.46,9.47) with the help of the QR-decomposition

$$\begin{bmatrix} A^{\mathrm{T}} \\ \sqrt{\lambda} I \end{bmatrix} = Q \begin{bmatrix} R \\ 0 \end{bmatrix}, \tag{9.48}$$

where Q and R denote an orthogonal $(2n-2)$-by-$(2n-2)$ matrix, and an upper triangular $(n-2)$-by-$(n-2)$ matrix, respectively. In MATLAB we could compute this decomposition by the statement

```
>> [Q, R] = qr ([A'; sqrt(lambda)*eye(n-2)]);
```

However, this command does not take advantage of the special structure of the matrix, and its numerical complexity increases cubically with the size of A. Fortunately, we can devise an algorithm to compute the QR-decomposition (9.48) whose numerical complexity increases only linearly with the size of the matrix A. We can achieve this improvement by an appropriate sequence of Givens transformations. In Figure 9.6 we show the application of the first three Givens rotations

ALGORITHM 9.4. *Evaluation of the Product* $\mathbf{y} := Q\mathbf{x}$.

```
$\mathbf{y} := \mathbf{x}$
**for** $k := n - 2$ **to** $1$ **by** $-1$ **do**
    rot $(y_k, y_{k+2}, \bar{Q}_{k5}, -\bar{Q}_{k6})$
    rot $(y_k, y_{k+1}, \bar{Q}_{k3}, -\bar{Q}_{k4})$
    rot $(y_k, y_{n+k}, \bar{Q}_{k1}, -\bar{Q}_{k2})$
**end**
```

ALGORITHM 9.5. *Evaluation of the Product* $\mathbf{y} := Q^{\mathrm{T}}\mathbf{x}$.

```
$\mathbf{y} := \mathbf{x}$
**for** $k := 1$ **to** $n - 2$ **do**
    rot $(y_k, y_{n+k}, \bar{Q}_{k1}, \bar{Q}_{k2})$
    rot $(y_k, y_{k+1}, \bar{Q}_{k3}, \bar{Q}_{k4})$
    rot $(y_k, y_{k+2}, \bar{Q}_{k5}, \bar{Q}_{k6})$
**end**
```

when processing the first column. The quantity σ is an abbreviation for $\sqrt{\lambda}$. If we apply this step $n - 2$ times we get the desired QR-decomposition (9.48). We show the same procedure in pseudo-code as Algorithm 9.6. The construction and application of Givens rotations is described by calls of the BLAS routines `rotg` and `rot`. Their precise definition is given in [2, 6]. The actual implementation in MATLAB or in a conventional programming language like C or Fortran is now straightforward.

We stress that the matrix $\bar{Q}$ computed by Algorithm 9.6 is not identical to the matrix Q from the QR-decomposition (9.48). Rather the matrix $\bar{Q}$ contains the information on the Givens rotations needed to apply the matrix Q to a vector $\mathbf{x}$. We present the evaluation of the products $\mathbf{y} := Q\mathbf{x}$ and $\mathbf{y} := Q^{\mathrm{T}}\mathbf{x}$ as Algorithms 9.4 and 9.5.

9.3.4 MEX-Files

Since MATLAB is an interpretative language it is not well suited to run Algorithms 9.4, 9.5, and 9.6. The overhead introduced by its interpreter would be considerable. Rather our code performs much better when implemented in a conventional programming language like C or Fortran. Fortunately, MATLAB provides a means to execute separately compiled code—the so-called MEX-files.

Let us assume we have implemented Algorithms 9.4, 9.5, and 9.6 in the C programming language. If we want to execute this code from inside MATLAB we need an interface procedure for each of these algorithms. In MATLAB a function call has the general syntax

```
[$l_1$, $l_2$, ..., $l_{\mathrm{nlhs}}$] = fct ($r_1$, $r_2$, ..., $r_{\mathrm{nrhs}}$);
```

The l_i's are called output parameters, and the r_i's are called input parameters. If this function is to be implemented as an external C subroutine we need the

Algorithm 9.6.
Calculation of the QR-decomposition (9.48).

```
Allocate the $(n-2)$-by-6 matrix $\bar{Q}$.
$r_{11} := 1$
$t := -2$
**if** $n > 3$ **then**
    $r_{22} := 1$
**end**
**for** $k := 1$ **to** $n-2$ **do**
    tmp := $\sqrt{\lambda}$
    rotg ($r_{kk}$, tmp, $\bar{Q}_{k1}$, $\bar{Q}_{k2}$)
    rotg ($r_{kk}$, $t$, $\bar{Q}_{k3}$, $\bar{Q}_{k4}$)
    **if** $k < n-2$ **then**
        $r_{k,k+1} := 0$
        rot ($r_{k,k+1}$, $r_{k+1,k+1}$, $\bar{Q}_{k3}$, $\bar{Q}_{k4}$)
    **end**
    tmp := 1
    rotg ($r_{kk}$, tmp, $\bar{Q}_{k5}$, $\bar{Q}_{k6}$)
    **if** $k < n-2$ **then**
        $t := -2$
        rot ($r_{k,k+1}$, $t$, $\bar{Q}_{k5}$, $\bar{Q}_{k6}$)
        **if** $k < n-3$ **then**
            $r_{k,k+2} := 0$
            $r_{k+2,k+2} := 1$
            rot ($r_{k,k+2}$, $r_{k+2,k+2}$, $\bar{Q}_{k5}$, $\bar{Q}_{k6}$)
        **end**
    **end**
**end**
```

following interface procedure:

```
#include "mex.h"

void mexFunction (int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
  ...
}
```

The two parameters `nlhs` and `nrhs` give the number of left-hand side arguments and the number of right-hand side arguments with which `fct` has been called in Matlab. The parameter `plhs` is a pointer to an array of length `nlhs` where we must put pointers for the returned left-hand side matrices. Likewise, the parameter `prhs` is a pointer to an array of length `nrhs`, whose entries point to the right-hand side matrices.

This interface routine should perform the following tasks:

1. It checks whether the proper number of input and output arguments has

ALGORITHM 9.7. *MEX-File for Algorithm 9.6.*

```
#include "mex.h"

#define max(A, B)  ((A) > (B) ? (A) : (B))
#define min(A, B)  ((A) < (B) ? (A) : (B))

#define n     prhs[0]
#define sigma prhs[1]

#define Qbar plhs[0]
#define R    plhs[1]

void mexFunction (int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{ int size, nnz;

  if (nrhs != 2) {
    mexErrMsgTxt ("spqr requires two input arguments.");
  } else if (nlhs != 2) {
    mexErrMsgTxt ("spqr requires two output arguments.");
  }

  if ((mxGetM (n) != 1) || (mxGetN (n) != 1) ||
      (*mxGetPr (n) < 3.0)) {
    mexErrMsgTxt ("n must be a scalar greater or equal 3.");
  }

  if ((mxGetM (sigma) != 1) || (mxGetN (sigma) != 1)) {
    mexErrMsgTxt ("sigma must be a scalar.");
  }

  size = (int) *mxGetPr (n);
  Qbar = mxCreateDoubleMatrix (size-2, 6, mxREAL);
  if (size == 3) {nnz = 1;} else {nnz = 3*size - 9;}
  R = mxCreateSparse (size-2, size-2, nnz, mxREAL);

  QR (size, *mxGetPr (sigma), mxGetPr (Qbar), R);
}
```

Algorithm 9.8.
Calculation of the QR-decomposition (9.48) in C.

```
void QR (int n, double sigma, double *Qbar, mxArray *R)
{ int    i, j, k, nnz, n2, n3, n4, *ir, *jc;
  double co, *pr, si, t, tmp;

  nnz = mxGetNzmax (R);  n2 = n-2;  n3 = n-3;  n4 = n-4;
  ir = mxGetIr (R);  jc = mxGetJc (R);  pr = mxGetPr (R);
  /* diagonal of R */
  ir [0] = 0;  for (i = 1; i < n2; i++) {ir [3*i - 1] = i;}
  /* first upper off-diagonal of R */
  for (i = 0; i < n3; i++) {ir [3*i + 1] = i;}
  /* second upper off-diagonal of R */
  for (i = 0; i < n4; i++) {ir [3*i + 3] = i;}
  /* columns of R */
  jc [0] = 0;  jc [1] = 1;
  for (j=2; j < n2; j++) {jc [j] = 3*j - 3;}
  jc [n2] = nnz;

#define r(i, j) pr [k = jc [j], k + i - ir [k]]
  r (0, 0) = 1.0;  t = -2.0;  if (n > 3) {r (1, 1) = 1.0;}
  for (j = 0; j < n2; j++) {
    tmp = sigma;
    rotg (&r (j, j), &tmp, &Qbar [j], &Qbar [n2 + j]);
    rotg (&r (j, j), &t, &Qbar [2*n2 + j], &Qbar [3*n2 + j]);
    if (j < n3) {
      r (j, j+1) = 0.0;
      rot (&r (j, j+1), &r (j+1, j+1),
           Qbar [2*n2 + j], Qbar [3*n2 + j]);
    }
    tmp = 1.0;
    rotg (&r (j, j), &tmp, &Qbar [4*n2 + j], &Qbar [5*n2 + j]);
    if (j < n3) {
      t = -2.0;
      rot (&r (j, j+1), &t, Qbar [4*n2 + j], Qbar [5*n2 + j]);
      if (j < n4) {
        r (j, j+2) = 0.0;  r (j+2, j+2) = 1.0;
        rot (&r (j, j+2), &r (j+2, j+2),
             Qbar [4*n2 + j], Qbar [5*n2 + j]);
      }
    }
  }
#undef r
}
```

been supplied.

2. It makes sure that the dimensions of the input matrices meet their specification.
3. It allocates the storage for the output matrices.
4. It calls another subroutine to perform the actual calculation.

The included file `mex.h` contains the MEX-file declarations and prototypes. There are a number of auxiliary subroutines available that can be called by the interface routine. For details the reader is referred to the External Interface Guide of MATLAB.
As an example we present as Algorithm 9.7 the C code which serves as an interface to the QR-decomposition of Algorithm 9.6. A translation of the pseudocode of Algorithm 9.6 into C is shown as Algorithm 9.8. We would also like to remind the reader that all the code is available in machine-readable form (see the preface for more information). In MATLAB we can now execute the statement

```
>> [Qbar, R] = spqr (n, sqrt (lambda));
```

to compute the sparse QR-decomposition (9.48).

In the same way we can implement Algorithms 9.4 and 9.5 by MEX-files. We can call them in MATLAB by the statements `y = Qx (Qbar, x)` and `y = QTx (Qbar, x)`, respectively.

9.3.5 Results

We are now ready to present the implementation of the least squares filter as Algorithm 9.9. The matrix A from equation (9.19) is created as a sparse matrix. In this way only the nonzero entries of A need to be stored. This is accomplished by the function `spdiags` which defines a sparse matrix by its diagonals.

The QR-decomposition (9.48) is computed by a call of `spqr` which implements Algorithm 9.6 as a MEX-file. Similarly, a call of the function `QTx` executes the MEX-file corresponding to Algorithm 9.5.

If we apply Algorithm 9.9 to our test data represented in Figure 9.2 we get the smoothed curve from Figure 9.7. We have set $\delta = 0.1$, which corresponds to the standard deviation of the noise in the function $f(x)$.

From a visual point of view the least squares filter returns a smoother result than the Savitzky-Golay filter whose output has been presented as Figure 9.4. The results get even better when the number of points n is increased. On the other hand the CPU-times required to execute Algorithm 9.9 are substantially higher than those needed to execute the Savitzky-Golay filter from Algorithm 9.2. We have summarized these CPU-times for a number of different data sizes n as Table 9.1.
We conclude that a least squares filter can produce a smoother curve $g(x)$ than a Savitzky-Golay filter. However, we need to pay for this advantage with a

Algorithm 9.9. *Least Squares Smoothing Filter.*

```
function g = lsq (f, delta)

n = length (f);
alpha = sqrt (n) * delta;
e = ones (n, 1);
A = spdiags ([e -2*e e], 0:2, n-2, n);

lambda = 0;
while 1,
  [Qbar, R] = spqr (n, sqrt (lambda));
  z = QTx (Qbar, [f; zeros(n-2, 1)]);
  z = R \ z (1:n-2);

  F = norm (A' * z)^2;
  if (F <= alpha^2), break; end;

  if (lambda > 0),
    zp = QTx (Qbar, [zeros(n, 1); -z/sqrt(lambda)]);
    zp = R \ zp (1:n-2);
    Fp = -2 * z' * (z + lambda * zp);
  else
    Fp = -2 * z' * z;
  end;

  lambdaold = lambda;
  lambda = lambda - 2 * (F / Fp) * (sqrt (F) / alpha - 1);
  if (lambda <= lambdaold), break; end;
end;
g = f - A' * z;
```

FIGURE 9.7.
Data of Figure 9.2 Smoothed by a Least Squares Filter.

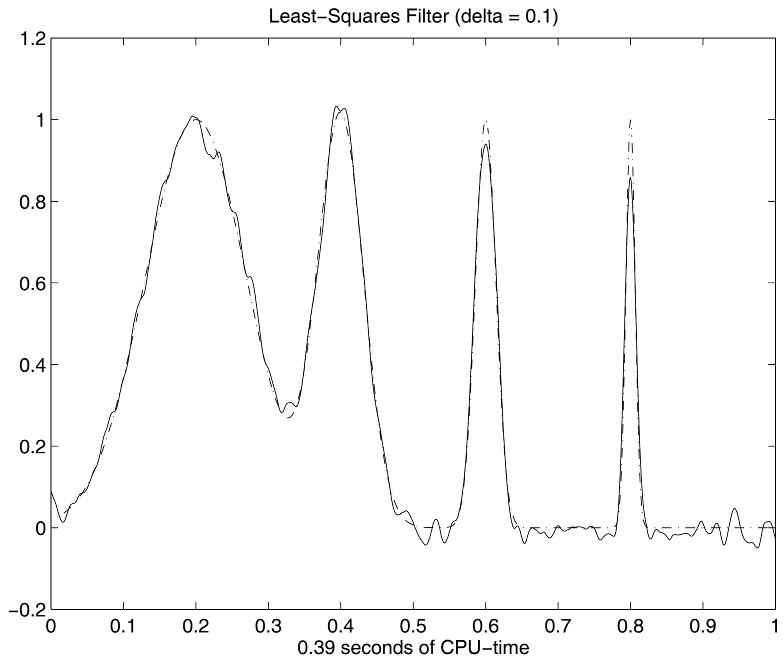


TABLE 9.1.
CPU-Times Corresponding to Different Data Sizes.

n	Savitzky-Golay Filter	Least Squares Filter
10^3	0.006 sec	0.39 sec
10^4	0.036 sec	4.55 sec
10^5	0.344 sec	77.93 sec

more complex algorithm and with an increased CPU-time. A least squares filter has also the advantage that only one parameter, the standard deviation δ of the noise, has to be estimated. This quantity is more directly related to the physical data than the parameters n_L, n_R, and M from the Savitzky-Golay filter.

References

[1] M. U. A. BROMBA AND H. ZIEGLER, *Application Hints for Savitzky-Golay Digital Smoothing Filters*, Analytical Chemistry, 53 (1981), pp. 1583–1586.

[2] J. J. DONGARRA, C. B. MOLER, J. R. BUNCH AND G. W. STEWART, *LINPACK Users' Guide*, SIAM Publications, Philadelphia, 1979.

[3] W. Gander, *On the Linear Least Squares Problem with a Quadratic Constraint*, Habilitationsschrift ETH Zürich, STAN-CS-78-697, Stanford University, 1978.

[4] W. Gander, *Least Squares with a Quadratic Constraint*, Numer. Math., 36 (1981), pp. 291–307.

[5] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Second Edition, The Johns Hopkins University Press, Baltimore, 1989.

[6] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh, *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Trans. Math. Softw., 5 (1979), pp. 308–325.

[7] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, *Numerical Recipes*, Cambridge University Press, Cambridge, 1986.

[8] W. H. Press and S. A. Teukolsky, *Savitzky-Golay Smoothing Filters*, Computers in Physics, 4 (1990), pp. 669–672.

[9] C. H. Reinsch, *Smoothing by Spline Functions. II*, Numer. Math., 16 (1971), pp. 451–454.

[10] A. Savitzky and M. J. E. Golay, *Smoothing and Differentiation of Data by Simplified Least Squares Procedures*, Analytical Chemistry, 36 (1964), pp. 1627–1639.

[11] J. Steinier, Y. Termonia and J. Deltour, *Comments on Smoothing and Differentiation of Data by Simplified Least Square Procedure*, Analytical Chemistry, 44 (1972), pp. 1906–1909.

[12] U. von Matt, *Large Constrained Quadratic Problems*, Verlag der Fachvereine, Zürich, 1993.

[13] H. Ziegler, *Properties of Digital Smoothing Polynomial (DISPO) Filters*, Applied Spectroscopy, 35 (1981), pp. 88–92.