Fast implementations of nonparametric curve estimators

J. Fan and J. S. Marron*
Department of Statistics
University of North Carolina
Chapel Hill, N. C. 27599-3260

February 12, 1993

Abstract

Recent proposals for implementation of kernel based nonparametric curve estimators are seen to be faster than naive direct implementations by factors up into the hundreds. The main ideas behind two different approaches of this type are made clear. Careful speed comparisons in a variety of settings, and using a variety of machines and software is done. Various issues on computational accuracy and stability are also discussed. The fast methods are seen to be somewhat better than methods traditionally considered very fast, such as LOWESS and smoothing splines.

1 Introduction

Smoothing techniques can be viewed as a set of useful and powerful methods for finding insights from sets of numbers. There are a wide array of these,

*Research partially supported by NSF Grant DMS-9203135

Keywords: Binning, fast computation, kernel methods, nonparametric curve estimation, smoothing, updating.

AMS 1991 subject classification. Primary 65D10. Secondary 62G07, 65C20.

and they are used and understood in quite different ways. See for example, the books by Eubank [6], Härdle [10] and Silverman [18].

A widely popular choice of smoothers, is kernel - local polynomial methods, which are appealing because of their simplicity and interpretability. These methods have been considered computationally slow in comparison to other methods though. In this paper, we discuss and compare two quite different approaches to this problem. These give implementations of kernel - local polynomial methods which are far faster than naive direct implementations. In particular speed factors well into the $100's$, are available for larger sample sizes. To understand the implications of a factor of 100, note that it is the difference between:

- a picture appearing on the screen in a few seconds, during an interactive data analysis, versus several minutes (we have to meet an analyst who can wait this long, and call it "interactive").
- a "batch type" job (e.g. a bootstrap analysis, or other type of simulation, etc.) which runs over night, versus in several months.

The above comparisons are not academic, but quite realistic. In particular, direct implementation of kernel - local polynomial estimators, with larger data sets, typically requires computation times well into minutes. However both fast methods discussed here are almost always well within our personal "patience zone" of 3 seconds, and frequently need much less than 1 second. These fast methods make kernel - local polynomial estimators at least as good, and occasionally somewhat better than, smoothers that have previously been considered "very fast", such as LOWESS and clever implementations of smoothing splines.

Some have expressed the opinion that because computational speeds are improving so rapidly there is no need to worry about careful programming or fast implementations. We agree that computational speeds are dramatically improving (although it does require a number of generations to give a speed factor of 100!), but disagree with the notion that fast methods are not worthwhile (when the improvement is of this order of magnitude). In particular, we would like to stress that *as computational capability improves, computational appetite grows at least as fast.* Witness the big boom in computationally intensive statistical methods, for example:

- "dimensionality reduction methods", such as Projection Pursuit, Alternating Conditional Expectations, Generalized Additive Models, Sliced Inverse Regression, Multivariate Additive Regression Splines, ...
- dynamic graphics
- bootstrap methods
- Gibbs sampling
- other types of simulation
- smoothing parameter selection
- image analysis

We agree that in the future, we will be able to perform our current tasks much more quickly, but expect that far more sophisticated methods will be attempted. We also feel that smoothers will form essential building blocks for some these, so fast implementations of smoothers is not a topic only of short term interest.

In section 2, we introduce the settings in which we consider smoothers: nonparametric density and regression estimation. A popular use of these kernel - local polynomial smoothers in these settings is graphical analysis of data. For this we recommend evaluation of the estimators at an equally spaced grid of 400 points. We have found fewer than 400 often results in distracting "granularity", while more grid points often gives negligible improvement in the resolution.

In section 3 we describe, and motivate two different fast methods for implementation of kernel - local polynomial estimators. The first is an approximation, based on "binning" the data. The second involves "updating" ideas, based essentially on recursively computing averages. The latter technique involves some deep ideas, that are relatively new, and quite obscure. These are much deeper than the widely known updating ideas, because general polynomial window shapes are allowed.

We have done a very wide array of comparisons of computational speeds of both these implementations are compared with each other, and with direct implementations. A selected representative part of these comparisons are presented in section 4. As noted above it is seen that improvements of

the fast methods over the direct involving speed factors in the 100's are available. Neither of the two methods dominates the other, with one being faster in some situations, but the other faster in different contexts. Models are presented which demonstrate how computational speed for each implementation depends on the sample size and the number of grid points. Comparison is also made across different settings, such as density and regression estimation, and across different kernel shapes.

Section 5 discusses "accuracy issues". For the binned approximation, it is seen that the difference between the binned approximation and the direct versions are surprisingly small. For visual purposes, these completely negligible in almost all cases, when our recommendation of 400 grid points is followed (and in fact for substantially smaller grids as well). The updated methods is seen to have some potential for problems for numerical instability. This is quite often not a problem (in particular for reasonable bandwidths, and not too high a degree polynomial kernel). However, there is potential for complete breakdown of the estimator, so the ideas behind this are worth clarification.

In our comparisons, we also considered quite a wide array of computational environments. In particular, we used different combinations of machines and software languages. The machines included a variety of PC's and workstations. The software languages ranged from the "low level" language C, to the "intermediate level" languages GAUSS and S. See section 6 for representative results of this type.

Section 7 provides comparison of the binned and updated algorithms discussed here, to the popular fast smoother LOWESS, and to fast implementations of the smoothing spline. As noted above, the present methods work impressively well in comparison to the other methods, which have been considered noteworthy because of their computational speed.

Extensions to other estimation settings, including equally spaced designs, derivative estimation, estimation of functionals and estimation in higher dimensions are commented upon in section 8.

While the algorithms suggested in section 3 do provide massive improvements in speed over direct implementations, there is a price to be paid. This comes in terms of extra effort in programming. Hence, we do not expect individuals to use these methods in all situations, although they will be very useful in many cases involving heavy iteration of smoothers. However, in our opinion, for any software package to be called "modern", it should use

algorithms of this type in a central way.

2 Settings

An appealing method for seeing structure in a univariate set of data is the kernel density estimator. Given a set of data $X_1, ..., X_n$ this estimator is defined by

$$\hat{f}_h \triangleq n^{-1} \sum_{i=1}^{n} K_h(x - X_i), \tag{1}$$

where $K_h(\cdot) = \frac{1}{h} K(\frac{\cdot}{h})$, for some "kernel" function, K, using a "bandwidth" h. If the data are thought of as a random sample from a probability density f, then $\hat{f}_h$ may be viewed as an estimate of f. See Silverman [18] for many useful facts about this estimator.

An intuitively attractive scatterplot smoother is a moving local average. Given a set of bivariate data $(X_1, Y_1), ..., (X_n, Y_n)$, the Nadaraya - Watson kernel regression estimator is given by

$$\widehat{m}_h(x) \triangleq \frac{\sum_{i=1}^{n} K_h(x - X_i) Y_i}{\sum_{i=1}^{n} K_h(x - X_i)}, \tag{2}$$

using the above notation. See Härdle [10] and Eubank [6] for discussion of relevant issues for this method.

An improved (significantly so at boundaries and for nonequally spaced data) scatterplot smoother is based on moving local linear regression. Again for bivariate data, this estimator is given by

$$\widehat{\widehat{m}}_h(x) \triangleq \frac{S_2(x)T_0(x) - S_1(x)T_1(x)}{S_2(x)S_0(x) - S_1^2(x)}, \tag{3}$$

where, for $\ell = 0, 1, 2$,

$$\begin{aligned} S_\ell(x) &\triangleq \sum_{i=1}^{n} K_h(x - X_i)(x - X_i)^\ell \\ T_\ell(x) &\triangleq \sum_{i=1}^{n} K_h(x - X_i)(x - X_i)^\ell Y_i \end{aligned} . \tag{4}$$

See Fan [4] and Hastie and Loader [12] for discussion.

Note that the kernel density and regression estimators also have representations in terms of the notation (4):

$$\hat{f}_h(x) = n^{-1} S_0(x), \qquad \widehat{m}_h(x) = \frac{T_0(x)}{S_0(x)}.$$

The most common use of these kernel estimators is construction of a plot for graphical analysis. This requires the estimator be evaluated at a grid of x locations, $x_1, ..., x_g$. The most time consuming part of the simplest direct implementation is usually kernel evaluation. Note that this entails $O(n \cdot g)$ kernel evaluations. In section 3.1 we describe a far faster method, requiring only $O(g)$ kernel evaluations. In section 3.2 we discuss another far faster method which implicitly calculates kernel evaluations, using essentially $O(n)$ operations.

The main family of kernel functions considered here is the "symmetric Beta family"

$$K(x) \triangleq C_\alpha \left(1 - x^2\right)_+^\alpha,$$

where the subscript + denotes "positive part" (which is assumed to be taken *before* the exponentiation, so this function is supported on $[-1, 1]$), and the constant $C_\alpha \triangleq \Gamma(2\alpha + 2)\Gamma(\alpha + 1)^{-2}2^{-2\alpha-1}$ makes K integrate to 1. As noted in Marron and Nolan [16] this family includes most widely used kernel functions, including the Gaussian, $K(x) = \phi(x)$, in the limit as $\alpha \to \infty$. We explicitly treat here the "uniform" with $\alpha = 0$, the "Epanechnikov" with $\alpha = 1$, the "biweight" with $\alpha = 2$, the "triweight" with $\alpha = 3$, and the Gaussian.

3 Fast Algorithms

3.1 Binning Methods

Binned implementations have been suggested by a number of authors, see Härdle and Scott [11] and Silverman [18] for access to earlier work. The key idea of binned implementations is to greatly reduce the number of kernel evaluations, through the fact that many of these evaluations are nearly the same. This requires that the grid $x_1, ..., x_g$ be equally spaced. The data are also approximated by "equally spaced data". A simple way of doing this is to "bin" the data, by replacing each X_i by the nearest grid point $x_{j(i)}$, denoted by $X_i \longmapsto x_{j(i)}$, and conceptually to do the estimation using this modified data set. The modified data set is conveniently handled through the index sets:

$$I_j \triangleq \{i : X_i \longmapsto x_j\}, \quad j = 1, ..., g.$$

Visual insight comes from thinking of the original data $\{(X_i, Y_i) : i = 1, ..., n\}$ (or the modified data) as being summarized by the "binned data"

$$\left\{\left(x_j, \overline{Y}_j, c_j\right) : j = 1, ..., g\right\},$$

using the "bin averages",

$$\overline{Y}_j \triangleq \text{avg}\left\{Y_i : i \in I_j\right\},$$

and the "bin counts"

$$c_j \triangleq \#\left\{X_i : i \in I_j\right\} = \#(I_j).$$

The most direct use of the modified data is to approximate $T_\ell(x_{j'})$, for $j' = 1, ..., g$, by

$$\begin{aligned} \overline{T}_\ell(x_{j'}) &\triangleq \textstyle\sum_{i=1}^{n} K_h\left(x_{j'} - x_{j(i)}\right)\left(x_{j'} - x_{j(i)}\right)^\ell Y_i \\ &= \textstyle\sum_{j=1}^{g} \sum_{i \in I_j} K_h\left(x_{j'} - x_j\right)\left(x_{j'} - x_j\right)^\ell Y_i \\ &= \textstyle\sum_{j=1}^{g} K_h\left(x_{j'} - x_j\right)\left(x_{j'} - x_j\right)^\ell Y_j^\Sigma, \end{aligned} \tag{5}$$

where $Y_j^\Sigma = c_j \overline{Y}_j = \sum_{i \in I_j} Y_i$. By a similar but easier calculation, $S_\ell(x)$ is approximated by

$$\overline{S}_\ell(x_{j'}) \triangleq \sum_{j=1}^{g} K_h\left(x_{j'} - x_j\right)\left(x_{j'} - x_j\right)^\ell c_j. \tag{6}$$

A naive implementation of (5) or (6) (usually) reduces the number of kernel evaluations from $O(n \cdot g)$ to $O(g^2)$.

But much more dramatic savings are available through *careful* calculation of (5) and (6), because many of the kernel evaluations are the same, as illustrated in Figure 1. In particular, letting $\Delta = x_j - x_{j-1}$ denote the "binwidth" or "grid spacing", note that each $x_j - x_{j-k} = k\Delta$, independent of j.

[put Figure 1 about here]

FIGURE 1: *Many pairwise differences are same for equally spaced grid points.*

This structure gives large computational savings when (5) is calculated as

$$\begin{aligned}\overline{T}_\ell(x_{j'}) &= \sum_{j=1}^{g} K\left(\Delta(j'-j)\right)\left(\Delta(j'-j)\right)^\ell Y_j^\Sigma \\ &= \sum_{j=1}^{g} \kappa_{\ell,j'-j} Y_j^\Sigma,\end{aligned} \tag{7}$$

where $\kappa_{\ell,j} = K_h\left(j\Delta\right)\left(j\Delta\right)^\ell$, since now the number of kernel evaluations is $O(g)$ (and similarly for $\overline{S}_\ell$). With so few kernel evaluations, the time to compute the discrete convolution becomes an important factor. We have experimented with both Fast Fourier Transform methods, such as recommended in section 3.5 of Silverman [18], and with direct implementations. For $g \leq 401$, we have not observed the anticipated large gains for the FFT. In fact we found the FFT to be occasionally slower. The main reason for this was the need to pad with zeros (resulting in a vector of length $2g-2$) to eliminate "boundary problems" caused by the circularity of the FFT. Additional padding may be needed to make $2g-2$ a power of 2 (or an integer with few prime factors). Another reason for the slowness of the FFT, is that it appears impossible to exploit zeros among the c_j, which is important for $n << g$. So we believe the FFT is usually not worth the extra trouble. The major benefits from binning come from the reduction in kernel evaluations, not the FFT calculation of the convolution, as suggested in section 3.5 of Silverman [18].

The number of convolutions required is quite different for the different estimators. In particular

$$\hat{f}_h^B(x) \triangleq n^{-1}\overline{S}_0(x)$$

requires one convolution,

$$\widehat{m}_h^B(x) \triangleq \frac{\overline{T}_0(x)}{\overline{S}_0(x)}$$

requires two convolutions, and

$$\widehat{\widehat{m}}_h^B(x) \triangleq \frac{\overline{S}_2(x)\overline{T}_0(x) - \overline{S}_1(x)\overline{T}_1(x)}{\overline{S}_2(x)\overline{S}_0(x) - \overline{S}_1^2(x)}$$

requires five convolutions. This generalizes to higher degree local polynomial regression, where a polynomial of degree p requires $3p+2$ convolutions.

A refinement of the binning procedure, is "linear binning" also discussed (together with early references) in Silverman [18]. The idea behind this is

illustrated in Figure 2. Conceptually think of "splitting" each point (X_i, Y_i) into two "fractional points", which are shared by the 2 nearest bin centers (i.e. "grid points"). The fraction assigned to each side is taken to be proportional to the distance from X_i to the nearest bin center on the *opposite* side. For this define, for $i = 1, ..., n$ and $j = 1, ..., g$

$$w_{i,j} \triangleq \left(1 - \frac{|X_i - x_j|}{\Delta}\right)_+$$

and "put fraction $w_{i,j}$ into the bin centered at x_j, and fraction $w_{i,j+1}$ into the bin centered at x_{j+1}."

[put figure 2 about here]

FIGURE 2: *Linear binning "splits" data points between two nearest bin centers.*

Evaluation of (7) and its $\overline{S}_l$ analog requires storage, for $j = 1, ..., g$, of

$$\begin{array}{l} c_j = \sum_{i=1}^n w_{i,j} \\ Y_j^{\Sigma} = \sum_{i=1}^n w_{i,j} Y_i \end{array},$$

which are used as above to calculate the $\overline{S}_\ell$ and $\overline{T}_\ell$. At first glance, both linear and simple binning appear to require $O(ng)$, operations (essentially a double loop through both i and j). But a much faster $O(n)$ algorithm is available through recognition of the fact that simple binning is essentially the quotient of an integer division. Linear binning can also be done in $O(n)$ operations, since $w_{i,j}$ is the remainder of that division. Because linear binning is also more precise, we use it in all examples in this paper. The $O(n)$ linear binning operation is based on the linear transformation $L(\cdot) = \frac{1}{\Delta}(\cdot - x_1) + 1$, which maps $\{x_1, ..., x_g\}$ onto $\{1, ..., g\}$. Then for $i = 1, ..., n$, the integer part of $L(X_i)$, denoted here by $j'(i) = \lfloor L(X_i) \rfloor$, indicates the two nearest bin centers to X_i, and the fractional part, $L(X_i) - j'(i)$, gives the "weights" assigned to the two nearest bin centers, as

$$w_{i,j'(i)} = 1 - (L(X_i) - j'(i)), \quad w_{i,j'(i)+1} = L(X_i) - j'(i).$$

Note that only a loop through i is required (not the double loop it is natural to first consider).

For handling endpoints, we have found that in some situations it is desirable to delete observations "outside the grid" (i.e. outside the interval $[x_1 - \Delta/2, x_g + \Delta/2]$), and in others it is preferable to move such observations to the nearest bin centers. Hence we prefer programs which allow a choice between these.

Careful comparison of the binned and direct implementations is done in section 5.1.

3.2 Updating Methods

A time honored approach to fast computation of smoothers is based on an "updating" idea. Simple understanding of this is provided by regression at an equally spaced grid of design points. In particular, consider a uniform kernel Nadaraya-Watson smooth of data $\{(1, Y_1), ..., (n, Y_n)\}$, to be evaluated at the same design points $\{1, ..., n\}$. For $j = 1, ..., n$, each $\widehat{m}_h(j)$ is essentially an ordinary average of $\{Y_{j-i_h}, ..., Y_{j+i_h}\}$, for some i_h determined by the bandwidth. After $\widehat{m}_h(j)$ has been computed, calculation of $\widehat{m}_h(j+1)$ does not require recalculation of the entire average, but instead only deletion of Y_{j-i_h}, and insertion of Y_{j+i_h+1}. For evaluation at the entire grid, this requires only $O(n)$ operations, compared to $O(n^2 h)$ for a direct implementation.

For nonequally spaced data, there are two different directions in which this idea may be extended. One is the "nearest neighbor" approach which keeps the idea of updating the moving average by one observation at each end of the moving window, and thus necessitates different bandwidths at different locations. We do not recommend this, as the result is inefficient (in the statistical sense) and hard to interpret, as noted in sections 2.5 and 5.2.1 of Silverman [18] and in Fan and Marron [5]. Another direction is more in the spirit of conventional kernel estimation, keeping the bandwidth the same, but updating by an appropriate (and differing) number of observations each time.

There are some clever further generalizations of updating. The LOWESS idea of Cleveland [3] is a further extension of the nearest neighbor approach, to robust local linear estimation. The "supersmoother" of Friedman [7] uses updating ideas for local bandwidth adjustment.

A serious drawback to all of the above ideas, is that the uniform kernel is at their heart, which entails rough looking plots. Extension to smoother nonuniform kernels is not obvious, because they involve *weighted* local aver-

ages, and all of the weights change with location. However, updating ideas have been adapted for polynomial kernels (e.g the beta family above, for integer α) by the Heidelberg school, see Gasser and Kneip [8] (there is also some interesting unpublished work on this by Brockmann, Engel, Gasser and Herrman). The key is to expand the polynomial into terms, in such a way that each can be calculated rapidly by "updating".

For a given point, x, when the kernel function is supported on $[-1,1]$, the data points which have a contribution to the $S_\ell(x)$ and $T_\ell(x)$ are indexed by

$$I_x \triangleq \left\{ i : \left| \frac{x - X_i}{h} \right| < 1 \right\}.$$

For a kernel of the form $K(x) = C_\alpha (1-x^2)_+^\alpha = C_\alpha \sum_{\nu=0}^{\alpha} \binom{\alpha}{\nu} (-x^2)^\nu$, note that

$$\begin{aligned} T_\ell(x) &= \textstyle\sum_{i=1}^n \frac{1}{h} K\left(\frac{x-X_i}{h}\right) (x - X_i)^\ell Y_i \\ &= C_\alpha \textstyle\sum_{i \in I_x} \sum_{\nu=0}^{\alpha} \binom{\alpha}{\nu} h^{-2\nu-1} (-1)^\nu (x - X_i)^{2\nu+\ell} Y_i \\ &= C_\alpha \textstyle\sum_{i \in I_x} \sum_{\nu=0}^{\alpha} \binom{\alpha}{\nu} h^{-2\nu-1} (-1)^\nu \sum_{k=0}^{2\nu+\ell} \binom{2\nu+\ell}{k} x^k (-X_i)^{2\nu+\ell-k} Y_i \\ &= C_\alpha \textstyle\sum_{\nu=0}^{\alpha} \binom{\alpha}{\nu} (-1)^\nu \sum_{k=0}^{2\nu+\ell} \binom{2\nu+\ell}{k} x^k \sum_{i \in I_x} h^{-2\nu-1} (-X_i)^{2\nu+\ell-k} Y_i. \end{aligned} \tag{8}$$

For a grid of x_j values, "updating" ideas can be used to rapidly compute summations of the form $\sum_{i \in I_x} h^{-2\nu-1} (-X_i)^{2\nu+\ell-k} Y_i$. The same idea is the key to LOWESS [3] and "supersmoother" [7], in the case of the uniform kernel ($\alpha = 0$). This is accomplished by first sorting the data, so that we may assume that $X_1 \le \cdots \le X_n$ (this is assumed to be already done through the rest of this section). Next note that differencing the cumulative sums $\tilde{U}_m \triangleq \sum_{i=1}^m h^{-2\nu-1} (-X_i)^{2\nu+\ell-k} Y_i$ yields

$$\sum_{i \in I_{x_j}} h^{-2\nu-1} (-X_i)^{2\nu+\ell-k} Y_i = \tilde{U}_{\overline{I}(x_j)} - \tilde{U}_{\underline{I}(x_j)},$$

for appropriately chosen indices $\underline{I}(x_j)$ and $\overline{I}(x_j)$. Exactly the same formulas, except with Y_i replaced by 1 are used to calculate the S_ℓ.

Note that the calculation of the cumulative sums requires only $O(n)$ operations, although the presorting (which needs to be done only once) requires at worst $O(n \log n)$ operations (of a type which is generally much faster). From a practical point of view (i.e. ignoring sorting time), this results in an

algorithm with $O(n)$ behavior. Note however that the constant coefficient in these speed calculations grows rapidly with α.

A major strength of this algorithm, over the binned methods described in section 3.1, is that they allow for bandwidth variation. There are two major types of bandwidth variation, where one replaces $K_h(x - X_i)$ by either $K_{h(x)}(x - X_i)$ or $K_{h_i}(x - X_i)$. To use Gasser's fast method with the first type of bandwidth variation, calculate

$$T_\ell(x) = C_\alpha \sum_{\nu=0}^{\alpha} \binom{\alpha}{\nu} (-1)^\nu \sum_{k=0}^{2\nu+\ell} \binom{2\nu+\ell}{k} x^k h(x)^{-2\nu-1} \sum_{i \in I_x} (-X_i)^{2\nu+\ell-k} Y_i.$$

based on the cumulative sums $\tilde{U}_m = \sum_{i=1}^{m} (-X_i)^{2\nu+\ell-k} Y_i$. For the second type, calculate

$$T_\ell(x) = C_\alpha \sum_{\nu=0}^{\alpha} \binom{\alpha}{\nu} (-1)^\nu \sum_{k=0}^{2\nu+\ell} \binom{2\nu+\ell}{k} x^k \sum_{i \in I_x} h_i^{-2\nu-1} (-X_i)^{2\nu+\ell-k} Y_i.$$

based on the cumulative sums $\tilde{U}_m = \sum_{i=1}^{m} h_i^{-2\nu-1} (-X_i)^{2\nu+\ell-k} Y_i$.

Explicit representations for these formulas in the cases $\alpha = 0, 1, 2, 3$ and $\ell = 0, 1, 2$ are given in the appendix.

4 Speed Comparisons

For simulated density estimation data, we drew 10,000 pseudo random realizations from the $N\left(0, \frac{1}{2\pi}\right)$ distribution (this scale makes the maximum height = 1, which assists in comparing accuracies). For the simulated regression data, we drew 10,000 pseudo random pairs (X, Y), where X was distributed uniformly on $[0, 1]$, and $Y \mid X = m(X) + \epsilon$, for $m(x) = 64x^3(1-x)^3$ (again chosen to have maximum height = 1) and ϵ distributed $N(0, 1/4)$. Visual impression for this regression setting is given in Figure 7.

For each of the sample sizes n = 25, 100, 250, 1000, 2500, and 10,000, we split the data into $10,000/n$ samples of size n, and calculated the average time required to compute each of the estimators discussed above at an equally spaced grid of $g = 401$ points, using each implementation, for the each of the kernels given at the end of section 2. Estimates were constructed for grids over the intervals $[-1.2, 1.2]$ for density estimation and $[0, 1]$ for

regression. Computation speed for most methods depends on the choice of bandwidth. As a reasonably representative choice we started with the Asymptotic Mean Integrated Squared Error optimal bandwidth h_{AMISE} (see for example, (2.2) of Härdle, Hall and Marron [9]). The binned and updated methods involve some "preprocessing" that needs only be done once for a given data set, e.g. binning and sorting. This need not be repeated for the calculation of additional estimates, e.g. with a different bandwidth, so we also timed each method for a total of three estimates, using the bandwidths $\frac{h_{AMISE}}{3}$, h_{AMISE}, $3h_{AMISE}$.

We tried these using a variety of hardware and software, discussed in detail in section 6 below. All calculations were done in double precision. In this section, we only present results using the programming language GAUSS on a 486 PC, and the programming language C on a SUN SPARC 2.

4.1 Comparison of Implementations

Figure 3 allows graphical comparison of the computation times for the various implementations.

[put Figure 3 about here]

FIGURE 3: *Comparison of average speeds for Direct, Binned and Updated Implementations, for the biweight kernel, for one and three estimates. Vertical bars show factors of 100 and 10. Figure 3a is for density estimation, running GAUSS on a 486 PC. Figure 3b is for local linear regression, using C on a SPARC 2.*

The horizontal line indicates a time of 3 seconds. We felt this was a reasonable "comfort threshold", meaning when doing interactive data analysis, most people have no problem waiting this long for a picture to appear, but become impatient with longer times. Note that in both cases the Direct Implementation leaves this range for n between 100 and 1000. However both of the fast implementations are essentially within this range for all sample sizes considered.

A rough rule of thumb is that for $n = 25$, there are not important differences between any of the implementations. However, important differences rapidly appear. In Figure 3a, the Direct Implementation is already slower

by a factor of nearly 10 for $n = 100$. For large samples this factor can be as large as 100.

	1 Estimate	3 Estimates
Direct:	$C_D ng$	$3C_D ng$
Binned (bin + convolve):	$C_{B1} n + C_{B2}\gamma(g)$	$C_{B1} n + 3C_{B2}\gamma(g)$
Updated (sort + update):	$C_{U1} n \log n + C_{U2}(n+g)$	$C_{U1} n \log n + 3C_{U2}(n+g)$

TABLE 1: *Approximate dependence of computation times on sample and grid sizes. The convolution time $\gamma(g)$ depends on the convolution algorithm used.*

The approximate shapes of the curves in Figure 3 are explained by the entries in Table 1. The curves for the Direct Implementation are nearly linear (with slope 1 on the log-log scale), because these speeds grow linearly in n (recall g is fixed at 401). They are separated by different intercepts, which essentially reflects the fact that it takes three times as long to calculate three estimates. The curves for the Binned Implementation are not linear, because they are essentially the sums of 2 components, the binning time ($C_{B1} n$, which is thus linear with slope 1, on this log-log scale), and the time for evaluation of the rest of the estimate ($C_{B2}\gamma(g)$, roughly constant with respect to the sample size, with precise form depending on computational method used). Hence they are essentially constant for n small, and then become linear (although in Figure 3b, this "asymptotic effect has not yet set in" for $n = 10,000$). The curves for the Updated Implementation are also not linear, because there are again essentially two components, the time for sorting ($C_{U1} n \log n$), plus the time for the updating part of the calculation (roughly of the form $C_{U2}(n+g)$). Thus these are also roughly constant for small n and become nearly linear for larger n. Figure 4 shows how these binning and sorting times (the left hand term in each entry of the table) determine the behavior seen in Figure 3.

[put figure 4 here]

FIGURE 4: *Display of effects of binning and sorting times on speeds shown in Figure 3. Upper curve of each type is the time for one estimate shown in Figure 3, and the lower curve is the preprocessing time.*

The curve for binning times is essentially (except for problems at the low end, caused by difficulties in the timing process with measuring times so small) linear (with slope 1), because as indicated in Table 1, binning time grows linearly with n. However the curves for the total time converge to the binning curves, because the time for the rest of the calculation is independent of n (again for fixed g). Also for this reason, the curves for three estimates in Figure 3a also converge to the binning time curve. This will also happen in Figure 3b, had we considered larger n, but the asymptotic effects have not yet "kicked in" for the sample sizes considered. Clearly these trends continue for larger n. A reasonable rule of thumb is that for large enough n, the Binned Implementation is faster than the Direct Implementation by roughly a factor of 100 (i.e. $C_{Dg} \approx C_{B1}/100$). The difference is even more dramatic when doing several smooths for the same data set. However there is very little significant difference between the binned and updated methods. For smaller sample sizes, sometimes one is better, sometimes the other. Note that $C_{Dg} << 10(C_{U1} + C_{U2})$, so for reasonable sample sizes updating is significantly better, although the improvements are not quite so dramatic as for binning over the direct method.

The curves for the Binned Implementations in Figures 3b and 4b are concave (at first glance surprising, since all other curves are convex), because for smaller n, a convolution algorithm was used which exploited the sparsity of the data ($n < g$). For the larger n in these figures, the time is roughly constant, because the binning time has not yet become an important component. Note that some of the binned implementation times decrease in n, because for larger samples, the bandwidths (recall these are all multiples of the asymptotically optimal bandwidth) decrease, which means fewer terms in the convolution are needed.

The curves for sorting times are not quite linear, because this requires order $n \log n$ operations. The total time for the Updated Implementation does not get very close to the sorting time in these pictures, because the remainder of the time grows linearly in n. Asymptotically, the time for the Updated Implementation will be the same as the sorting time. This will eventually be slower than even the Direct Implementation, but because of the relative magnitudes of the constant coefficients, this will require astronomically large samples (e.g. roughly speaking in this picture, the "trade-off point" will be of the order of magnitude of $n = e^g = e^{401} \approx 10^{174}$). However, for reasonable sample sizes, the Updated Implementation is quite competitive with the

Binned Implementation.

4.2 Comparison of Settings

This section compares how computation times vary across the settings of density estimation, Nadaraya-Watson regression and local linear regression. Figure 5 shows computation times, organized in a convenient way for this purpose.

[put Figure 5 about here]

FIGURE 5: *Comparison of average speeds across estimation settings, for one (top curves of each type) and three (bottom curves of each type) estimates, all using the biweight kernel. Vertical bars show factors of 100 and 10. Figure 5a is for the Binned Implementation running in GAUSS on a 486 PC, and Figure 5b is for the Updated Implementation*

Figure 5a is quite representative of a number of similar pictures we have made for the Binned Implementation. Note that the computation times for the Nadaraya-Watson estimator are roughly twice those for the density estimator, for all n. This is because both the binning time ($C_{B1}n$), and the convolution time ($c_{B2}\gamma(g)$) are essentially doubled. The local linear estimator is slower than the Nadaraya-Watson estimator by a factor of slightly more than two for small samples, decreasing to one for large samples. The reason for this is that local linear estimation requires 5 convolutions, versus two for the Nadaraya-Watson, which is important for small samples (where the convolution time $C_{B2}\gamma(g)$ is dominant). However for large samples, the binning time $C_{B1}n$ becomes dominant, and this is the same for both.

Figure 5b is also quite representative, for the Updated Implementation. The Nadaraya-Watson estimator is slower than density estimation, by a fairly constant factor of about 1.3. This is much less than 2, because the similar forms of S_0 and T_0 (needed by the Nadaraya-Watson) allow simultaneous calculation with much less than twice the effort for S_0 alone (as needed for density estimation). The local linear estimator is far slower (roughly a factor of 2 for smaller samples), because S_1, S_2, and T_1 require a different structure, and calculation of more terms. For larger samples, there is less difference between the settings, because the sorting time ($C_{U1}n \log n$) becomes an important component of the total time.

4.3 Comparison of Kernels

This section studies the relative computation times for the different kernels.

[put Figure 6 about here]

FIGURE 6: *Comparison of average speeds across kernels, for one Nadaraya Watson estimate. Vertical bars show factors of 100 and 10. Figure 6a is for the Binned Implementation running in GAUSS on a 486 PC, and Figure 6b is for the Updated Implementation running in C on a SPARC 2.*

These pictures are very representative of the corresponding ones for other settings. In all cases, the symmetric beta family is faster for smaller α, by a factor which is linear in α.

For the Binned Implementation, shown in Figure 6a, the triweight kernel is about 1 to 1.4 times as slow as the Uniform. The Gaussian is somewhat slower than all members of the beta family (by a factor of 1.2 - 3). One reason is that the Gaussian is infinitely supported (thus requiring more terms in the convolution). This can be improved somewhat, by truncating the Gaussian kernel, e.g. beyond 4 standard deviations (this value was suggested by M. Wand). From Figure 6a it is apparent that the gains from this can be noticeable but limited. Another reason that the Gaussian is slower is that it requires exponentiation in the kernel evaluation. For larger sample sizes, the times converge, because the constant binning time ($C_{B1}n$) becomes dominant.

For the Updated Implementation, shown in Figure 6b, there is no Gaussian kernel, because that idea only applies to polynomial kernels. There is now a more important difference between each of the symmetric beta kernels, because higher degree polynomials take more time to compute using this algorithm. The triweight is slower than the Uniform by factors of 2 - 3, depending on n. For larger sample sizes, the times converge somewhat, because the sorting time ($C_{U1}n \log n$) plays an increasingly significant role.

5 Accuracy Issues

There are two important points to consider about accuracy. One is how close the Binned Implementation is to direct versions. The other is assessing round

off error in the Updated Implementation.

5.1 Binned vs. Direct Implementations

An interesting issue in comparing the direct and binned estimators is: which should be regarded as "truth"? Certainly the direct estimator has historical precedence, but each is after all only an estimate. Since the binned estimator is so much more useful (because it can be computed much more rapidly), a case can be made for considering *this* to be "the" estimator, and the direct method an approximant which is perhaps easier to analyze, and intuitively understand. Fortunately, we will see in this section that the point is essentially moot, because there is rarely an important practical difference.

Figure 7 demonstrates the visual difference between the Binned and Direct Implementations, for some different choices of g and n.

[put Figure 7 about here]

FIGURE 7: *Simulated Data, with Binned and Direct Nadaraya Watson estimates. Figures 7a and c each show three estimates using the Gaussian kernel, at* $h_{AMISE}/3$, h_{AMISE}, and $3h_{AMISE}$. *Figures 7b and d each show one estimate using the Uniform kernel, at* h_{AMISE}.

The Gaussian kernel gives both implementations remarkably close to each other, even for the quite sparse grid with $g = 51$. Although there are 6 separate curves, 3 dotted and 3 dashed, they look much like 3 dot and dashed curves. Careful inspection shows a few separations, which are easiest to see for the smaller bandwidth, and $g = 51$. But for the goal of visual data analysis, the differences are clearly negligible in all cases.

There are much larger differences for the Uniform kernel. The moving of observations done in the binning operation has a much larger effect for this kernel, because slight movements in the x direction can determine whether or not an observation is in or out of a given window. This effect is worse for smaller bandwidths, because averages with fewer observations change more when points are added or deleted, although to save space we decided not to include a figure demonstrating this. This effect is also worse for $g = 51$, because observations are moved much further. For $g = 401$, the Binned and

Direct Implementations look visually fairly similar, although their sup norm difference is actually fairly large, because a slight "horizontal shift" in the step function, can mean a "big vertical jump". For example, in Figure 7d, near $x = .09$, note that the Direct Implementation appears to "jump up" one bin center before the Binned Implementation.

Figure 8 contains a summary of part of a more extensive comparison of the Binned and Direct Implementations. For kernel density estimation, this shows the "Relative Accuracy Measure" given by

$$\frac{\left\| \hat{f}_h^B - \hat{f}_h \right\|_\infty}{\max\left(\hat{f}_h\right) - \min\left(\hat{f}_h\right)}, \tag{9}$$

for different choices of h and g, calculated for just one set of the data used in the speed comparisons.

[put Figure 8 about here]

FIGURE 8: *Relative Accuracy Measures for kernel density estimation, sample size* $n = 1000$. *Bandwidths are:* $h_{AMISE}/3$ *in Figure 8a,* h_{AMISE} *in Figure 8b.*

The horizontal lines show error thresholds of 5% (considered a lower bound for "visually distracting") and 1% (considered a lower bound for "visually noticeable"). Except for the uniform kernel, for $g \geq 100$ all kernels are always below the "noticeable threshold". The uniform kernel usually gets better for increasing g, with exceptions caused by random variation (most noticeable at the smallest bandwidth, for the reason explained above). We do not advocate use of the uniform kernel, but for those who prefer it, the picture is not so gloomy as presented here, for the reasons given above about the difference between "visual impression" and the sup norm.

We also looked at the same picture for $3h_{AMISE}$, but do not include it, because (except for the uniform kernel) the relative accuracy is essentially indistinguishable from 0. In addition, we made all 3 pictures for $n = 100$, but the ideas are essentially the same, so these are not shown either.

This is only one example, and the specific numbers may vary in other examples (particularly for other density shapes), but we believe the main lessons are generally applicable. Although $g = 401$ is more than enough for

acceptable accuracy, we prefer this value for resolution reasons. In particular, smaller values often result in a picture which is "too granular", with the grid discreteness being visually less pleasant especially for curves with sharper features.

See Jones [14] for asymptotic results, and earlier references, showing how well these binned estimators approximate the unbinned versions, in the case of density estimation.

5.2 Instability in the Updated Implementation

To see how updating methods can have trouble with numerical instability, consider the Epanechnikov kernel for example. Note that $S_0(x_j)$ is calculated essentially as

$$\sum_{i\in I_{x_j}}\left[1-\left(\frac{x_j-X_i}{h}\right)^2\right]=\left[1-\left(\frac{x_j}{h}\right)^2\right]\sum_{i\in I_{x_j}}1+\left(\frac{2x_j}{h}\right)\sum_{i\in I_{x_j}}\left(\frac{X_i}{h}\right)-\sum_{i\in I_{x_j}}\left(\frac{X_i}{h}\right)^2.$$

For small bandwidths, each of the three summations will involve large values, which upon subtraction, results in a loss of precision. This effect is illustrated in Figure 9.

[put Figure 9 here]

FIGURE 9: *Relative accuracy (9) for the updated method, compared to the direct. For Nadaraya Watson regression, with one data set of size* $n = 1000$, *using* $g = 401$. *Vertical bars show the asymptotically optimal bandwidth for each kernel.*

As indicated by the above heuristics, the accuracy is worse for smaller bandwidths. The horizontal lines show the same error thresholds discussed in the previous section. Note that in most cases, there is no visual difference between estimators. The exception is the triweight kernel, which for small enough bandwidths gives an estimator which is completely different from the direct one (relative accuracy > 1, i.e. more than the vertical range of $\widehat{m}_h$!). However, note that this only occurs for bandwidths $\leq h_{AMISE}/10$, which in most situations is a very small amount of smoothing. Hence we feel that in most situations, this problem is negligible (but double precision calculations

are strongly recommended). However, this is only one example, and the breakdown is alarmingly complete when it does occur, so the potential for this problem should be kept in mind.

Note also that errors are worse for larger α. The log-log scale in Figure 9 makes it clear that relative accuracy $\sim h^{-2\alpha}$. This fits with the above heuristics, because for large α the terms in the resulting summations depend on h through $h^{-2\alpha}$. For example the uniform kernel has relative accuracy which is essentially double precision roundoff error, because no subtraction of large terms is done, regardless of how small the bandwidth is. Also note that the relative accuracies converge at $h = 1$.

6 Machines and Software

In this section we compare the machine and software language combinations C on a SUN SPARC 2, GAUSS on a 33 mHz 486 PC, GAUSS on a SUN SPARC 2, and S on a SUN SPARC IPX. Figure 10 shows curves that are of the same type as in Figures 3-6, but organized to highlight this comparison. The main ideas seen here again apply quite generally to the other settings we have considered.

[put Figure 10 about here]

FIGURE 10: *Comparison of average speeds across computation environments, for one Nadaraya-Watson regression estimate. Vertical bars show factors of 100 and 10. Figure 10a is for the binned implementation using the biweight kernel. Figure 10b is for the Updated Implementation using the biweight kernel. Figure 10c is for the direct implementation and the uniform kernel. Figure 10d is for the direct implementation and the Gaussian kernel.*

The software language S gives performance much slower than all the others, by factors of 10-100. GAUSS on the SPARC 2 is only marginally better than GAUSS on the PC. C on the SPARC 2 is better than all others we tried by roughly a factor of 3-5.

The excruciatingly slow performance of S is due to extremely poor memory management. In Figure 10c, the computation times grow faster than any

polynomial (recall polynomials are linear on this log-log scale), for this reason. For this same reason, we were not even able to complete the $n = 10,000$ part of our program, which is why this is missing in Figure 10d. S does not appear in Figure 10b, because we did the computations for the updated method later in the study, and had decided S was just not sufficiently competitive to continue with it.

We made similar pictures (but do not show them to save space) for comparisons of different generations of PC's. Rough rules of thumb are:

- The 33 mHz 486 is faster than a 20 mHz 386 by a factor of 3-5.
- The 20 mHz 386 is faster than a 6 mHz 286 by a factor of 10.

Another issue which we studied, (but again have decided not to add a figure to this paper), is the issue of looped vs. matrix implementations. This is important for "intermediate level" languages like GAUSS and S (vs. "low level" languages such as C or FORTRAN), where one has a choice, and most people's backgrounds leave them naturally thinking in terms of DO loops. We found that in S we were not able to run the looped version of any of our programs, again because of the memory management problems. In GAUSS we found that the difference was very large indeed, with roughly a speed factor of 100, when comparing a looped direct implementation with a matrix based direct implementation. When this is combined with the speed factor of up to 100 that is available for a binned or updated implementation, one has a speed factor of up to 10,000 available between what might be one's first attempt, and what is available using the ideas in this paper!

7 Other Smoothers

In this section we discuss how our fast implementations of kernel estimators compare with other popular smoothers, that are known for being computationally fast. Both are regression estimators, so we consider only that setting in this section.

7.1 LOWESS

The smoother LOWESS, proposed by Cleveland [3], has achieved popularity because it is prominently featured in the software package S. Current imple-

mentations are not exactly comparable with those presented here, because the estimate may only be evaluated at the data points. Although evaluation only at the data points is desirable for certain applications, it is less pleasing for visual data analysis, unless n is close to a reasonable value, and the data are reasonably "uniform". Smaller n (and /or data with "large gaps" in the x direction) will result in excessively granular pictures. For larger n this can be computationally inefficient.

A major weakness of LOWESS is that it only allows the uniform kernel, which entails substantial roughness in the resulting estimate as discussed in section 5. It is possible to overcome this difficulty, using the updating ideas discussed in section 3.2, but this is not currently available. A less important weakness of LOWESS is that it uses "nearest neighbor", versus fixed window methods, which have problems as discussed in section 3.2. An important strength of LOWESS is that it does a type of robust regression, giving an estimate which is less sensitive to "outliers" among the Y_i than any other method considered here.

Figure 11a shows how this method compares in terms of speed with the estimators we have considered, which are closest in spirit. In our computations, we chose the smoothing parameter to approximate those we used for the above methods. We used the default value in S for determining the "amount of robustness".

[put Figure 11 here]

FIGURE 11: *Comparison of average speeds for 1 and 3 estimates, using other popular methods. Vertical bars show factors of 100 and 10. LOWESS vs. using the binned and updated local linear methods, with the uniform kernel in Figure 11a. A smoothing spline vs. the fastest and slowest binned regression estimators in Figure 11b.*

For smaller n all three estimators are roughly comparable, but for larger n LOWESS is substantially slower, by a factor of 10 or more, relative to both the binned and updated methods. The slower speed of LOWESS at the larger sample sizes is not surprising because it is evaluating at more places ($g = n$ vs. $g = 401$). However we expected LOWESS to be substantially faster for smaller n because then it is evaluating at fewer locations ($g = n << 401$),

but note that these anticipated gains are not realized. There are other factors which also make it difficult to compare. For example, both LOWESS and our updating methods require a sort of the data, but it is unclear to us which sorting algorithm is used by LOWESS. Also the robustness part of LOWESS requires additional time.

However, it seems clear that methods described in this paper are very competitive (in terms of computational speed) with LOWESS, which has been considered a useful smoother because it is viewed as being very fast to compute.

7.2 Smoothing Splines

Calculation of smoothing splines is most simply done for evaluation of the estimate at the "design points", $X_1, ..., X_n$. It involves solving a $O(n) \times O(n)$ system of linear equations. However, the coefficient matrix has a banded structure, which allows a solution in order $O(n)$ calculations, so this is usually considered a very fast method for smoothing. As noted above, evaluation at the design points yields poor resolution for small n and is wasteful for large n. To overcome this problem, and to do as fair a comparison as possible, we first binned the data, and applied smoothing spline ideas to the pseudo data $(x_1, c_1, \overline{Y}_1), ..., (x_g, c_g, \overline{Y}_g)$, in particular fitting a weighted smoothing spline, with weights of each observation proportional to c_j. This essentially gave an estimate at the same grid as the other methods discussed in this paper. One exception was bins with $c_j = 0$, which had to be omitted from the system of equations, so at those locations, the spline was evaluated by interpolation.

Figure 11b shows how the spline compares with our fastest and slowest binned implementation, where all programs are done in GAUSS on the 486 PC. Note that performance is roughly comparable, although the spline is slightly slower for all n. It may be surprising that the spline is slowest for $n = 100, 250$. This is because of the interpolation process over the bins with $c_j = 0$. For $n \geq 400$, there was essentially no interpolation to be done (and interpolation was slower than equation solving in our algorithm).

We feel the important lesson here is that binned and updated methods are very competitive with smoothing splines, which have previously been considered outstanding in terms of speed. In addition, when there is a premium on speed, use of the uniform kernel Nadaraya-Watson estimator can give big savings over the smoothing spline.

We also did a comparison of binned and updated methods with the default smoothing spline in S. No pictures are included here to save space and because the main lessons are the same.

8 Extensions and Related Issues

8.1 Equally Spaced Designs

The regression examples in this paper are all for "random design regression", where the X_i are *not* equally spaced. However, there are many real data situations, e.g. "designed experiments" and "time series data", where a smooth is desired based on data where the X_i are equally spaced and in increasing order, and it is desired to use these same points as the grid for plotting. In this case substantial savings in computational speed are available for both the binned and updated implementations discussed here.

For the binned method, the savings come from the fact that there is no need to bin, but the equal spacing of the data themselves may be exploited as described in section 3.1 to yield a fast algorithm. This requires no new programming, since the same algorithms can be used, with the substitutions $j \leftarrow i$, $Y_j^{\Sigma} \leftarrow Y_i$, $c_j \leftarrow 1$. The effect of this on the computation times is easily understood by noting that the first term of the entries of Table 1 now disappears.

For the updated method, big savings come from the fact that there is no need to sort the data. This also requires no new programming. Here again the first term in Table 1 disappears.

Exact comparison, of the binned and updated methods with the direct, in this case requires additional computation (because g is no longer constant). We do not feel a strong desire to explicitly do this, because we think it is clear that improvements of the suggested methods over the direct will be even more dramatic.

Note also the error analysis, as in section 5.1, is no longer such an interesting issue. In particular the binned method now is an exact version of the original estimator. The numerical instabilities in the updated method will still follow the pattern indicated in section 5.2.

8.2 Derivatives

Proposed estimates of derivatives of densities and regression functions have been based on differentiating kernel estimates, and also on using higher degree coefficients in local polynomial estimates. Binned and updated ideas provide computational savings of similar orders of magnitude, compared to direct implementation, to those observed in this paper.

The error analyses in section 5 will be different here though. The difference between the binned and direct estimators will be some what larger (because the corresponding kernels now have "more detail", e.g oscillations). A more important difference appears for the updated method. In particular, the numerical instabilities indicated in Figure 9 will be a more important problem. In some unpublished work by Brockman, Engel, Gasser and Hermann, this problem is attacked by some clever "restart" ideas. The idea is that after the error accumulates to unacceptable levels, the updating is discontinued, and a fresh start is made.

8.3 Functionals

Kernel - local polynomial methods are also very useful for estimation of functionals. For example the functional $\int (f^{(k)})^2$, for various k is of great interest in such diverse fields as bandwidth selection, robust estimation, and classical rank based nonparametric statistics. A natural estimator involves substituting a density estimate, i.e. $\int (\widehat{f}_h^{(k)})^2$. There are many closely related functionals in the regression case, and estimators based on similar local polynomial ideas.

Binned and updated ideas extend naturally and simply to such functionals (modulo the above considerations about derivative estimation), but we do not give details here. The speed improvements in using fast methods in this context are potentially even more dramatic than shown in the settings of this paper (for example, a binned implementation requires $O(g)$ kernel evaluations, vs. $O(n^2)$ required for calculating this directly, and $O(n)$ operations vs. $O(n^2)$ for the updated version). Precise speed and accuracy comparisons, of the type done in this paper is an interesting open problem. See Aldershof [1] for some work in this direction.

Binned implementations have already been used in bandwidth selection for kernel density estimation, see Jones, Marron and Sheather [15] for discus-

sion and access to earlier uses. Note that in the simulations presented in that paper, sample sizes as large as $n = 100,000$ are considered, while in some contemporary simulation studies, see e.g. Park and Turlach [17], where only far smaller samples of $n = 100$, are considered, because only direct implementations were used there. Cao, Cuevas and Gonzáles-Mantiega [2] do use a binned method, with FFT calculation of the convolution, but also only treat $n = 100$. An interesting updated version of least squares cross-validation, for density estimation, has been proposed by Lee and Kim [13].

8.4 Higher Dimensions

Computational speed is a very important consideration for estimation of functions in higher dimensions, for example, a regression or density in $\mathbb{R}^d$ is typically evaluated on a rectangular grid with g^d points. We do not see how to extend updating ideas, but binned methods do extend naturally, with again dramatic savings. Interesting work is underway by Hall and Wand, who report large savings. An important difference in their conclusions and ours, is that they find FFT based convolutions provide large improvements over the direct ones we have found preferable in $\mathbb{R}^1$. An alternative to FFT's is sophisticated handling of pointers to "zero bins", as discussed in Härdle and Scott [11]. This issue appears to be unsettled as yet. Accuracy questions are also still wide open, but we anticipate the answers will again be different from those here.

9 Appendix

Explicit formulas, for convenience in programming, are given here for the calculation of the polynomials

$$P_{\alpha,\ell}(a,\underline{b}) = C_\alpha \sum_{\nu=0}^{\alpha} \binom{\alpha}{\nu} (-1)^\nu \sum_{k=0}^{2\nu+\ell} \binom{2\nu+\ell}{k} a^k b_{2\nu+\ell-k},$$

where b is a sequence of coefficients, in the cases $\alpha = 0,1,2,3$ and $\ell = 0,1,2$. Note for each α and each ℓ, that $S_\ell(x_j) = h^\ell P_{\alpha,\ell}\left(\frac{x_1}{h}, \underline{b}\right)$ and $T_\ell(x_j) = h^\ell P_{\alpha,\ell}\left(\frac{x_i}{h}, \underline{b}\right)$ for $\underline{b}$ with components $b_{2\nu+\ell-k} = \sum_{i \in I_x} \left(\frac{-X_i}{h}\right)^{2\nu+\ell-k}$ and $b_{2\nu+\ell-k} = \sum_{i \in I_x} \left(\frac{-X_i}{h}\right)^{2\nu+\ell-k} Y_i$ respectively.

Straightforward calculations show that

$$P_{0,0} = \frac{1}{2} b_0$$

$$P_{0,1} = \frac{1}{2} \left[a b_0 + b_1 \right]$$

$$P_{0,2} = \frac{1}{2} \left[a^2 b_0 + 2 a b_1 + b_2 \right]$$

$$P_{1,0} = \frac{3}{4} \left[(1 - a^2) b_0 - 2 b_1 - b_2 \right]$$

$$P_{1,1} = \frac{3}{4} \left[a(1 - a^2) b_0 + (1 - 3a^2) b_1 - 3 a b_2 - b_3 \right]$$

$$P_{1,2} = \frac{3}{4} \left[a^2(1 - a^2) b_0 + a(2 - 4a^2) b_1 + (1 - 6a^2) b_2 - 4 a b_3 - b_4 \right]$$

$$P_{2,0} = \frac{15}{16} \left[(1 - a^2)^2 b_0 - 4a(1 - a^2) b_1 - 2(1 - 3a^2) b_2 + 4 a b_3 + b_4 \right]$$

$$\begin{aligned} P_{2,1} = \tfrac{15}{16} [& a(1 - a^2)^2 b_0 + (1 - a^2)(1 - 5a^2) b_1 - a(6 - 10a^2) b_2 \\ & -2(1 - 5a^2) b_3 + 5 a b_4 + b_5] \end{aligned}$$

$$\begin{aligned} P_{2,2} = \tfrac{15}{16} [& a^2(1 - a^2)^2 b_0 + 2a(1 - a^2)(1 - 3a^2) b_1 \\ & +(1 - 12a^2 + 15a^4) b_2 - 4a(2 - 5a^2) b_3 \\ & -(2 - 15a^2) b_4 + 6 a b_5 + b_6] \end{aligned}$$

$$\begin{aligned} P_{3,0} = \tfrac{35}{32} [& (1 - a^2)^3 b_0 - 6a(1 - a^2)^2 b_1 - 3(1 - a^2)(1 - 5a^2) b_2 \\ & +4a(3 - 5a^2) b_3 + 3(1 - 5a^2) b_4 - 6 a b_5 - b_6] \end{aligned}$$

$$\begin{aligned} P_{3,1} = \tfrac{35}{32} [& a(1 - a^2)^3 b_0 + (1 - a^2)^2(1 - 7a^2) b_1 \\ & -3a(1 - a^2)(3 - 7a^2) b_2 - (3 - 30a^2 + 35a^4) b_3 \\ & +5a(3 - 7a^2) b_4 + 3(1 - 7a^2) b_5 - 7 a b_6 - b_7] \end{aligned}$$

$$\begin{aligned} P_{3,2} = \tfrac{35}{32} [& a^2(1 - a^2)^3 b_0 + 2a(1 - a^2)^2(1 - 4a^2) b_1 \\ & +(1 - a^2)(1 - 17a^2 + 28a^4) b_2 - 4a(3 - 15a^2 + 14a^4) b_3 \\ & -(3 - 45a^2 + 70a^4) b_4 + a(18 - 56a^2) b_5 + (3 - 28a^2) b_6 - 8 a b_7 - b_8] \end{aligned}$$

References

[1] Aldershof, B. K. (1991) *Estimation of Integrated Squared Density Derivatives*, Institute of Statistics, Mimeo Series #2053.

[2] Cao, R., Cuevas, A. and Gonzáles-Mantiega, W. (1993) "A comparative study of several smoothing methods in density estimation", to appear in *Computational Statistics and Data Analysis.*

[3] Cleveland, W. S. (1979) "Robust locally weighted regression and smoothing scatterplots" , *Journal of the American Statistical Association,* 87, 829-836.

[4] Fan, J. (1992) "Design adaptive nonparametric regression", *Journal of the American Statistical Association,* 87, 998-1004.

[5] Fan, J. and Marron, J. S. (1993) Discussion of "Local Regression: Automatic Kernel Carpentry", by Hastie and Loader, *Statistical Science*, to appear.

[6] Eubank, R. L. (1988) *Spline Smoothing and Nonparametric Regression*, Dekker, New York.

[7] Friedman, J. H. (1984) "A variable span smoother", Stanford University LCS Technical Report No.5, SLAC PUB-3477.

[8] Gasser, T. and Kneip, A. (1989) Discussion of "Linear smoothers and additive models", by Hastie and Tibshirani, *Annals of Statistics*, 17, 453-555.

[9] Härdle, W., Hall, P. and Marron, J. S. (1988) "How far are automatically chosen smoothing parameters from their optimum?", *Journal of the American Statistical Association*, 83, 86–101.

[10] Härdle, W. (1990) *Applied Nonparametric Regression*, Cambridge University Press, Boston.

[11] Härdle, W. K. and Scott, D. W. (1992) "Smoothing by weighted averaging of shifted points", *Computational Statistics*, 7, 97-128.

[12] Hastie, T. and Loader, L. (1993) "Local Regression: Automatic Kernel Carpentry", to appear in *Statistical Science*.

[13] Lee, B. G. and Kim, B. C. (1990) "An efficient algorithm for the least squares cross-validation with symmetric and polynomial kernels", *Communications in Statistics - Simulation*, 19, 1513-1522.

[14] Jones, M. C. (1989) "Discretized and interpolated kernel density estimates", *Journal of the American Statistical Association*, 84, 733-741.

[15] Jones, M. C., Marron, J. S. and Sheather, S. J. (1992) "Progress in data based bandwidth selection for kernel density estimation", North Carolina Institute of Statistics Mimeo Series #2088.

[16] Marron, J. S. and Nolan, D. (1988) "Canonical kernels for density estimation", *Statistics and Probability Letters*, 7, 195-199.

[17] Park, B. U. and Turlach, B. A. (1992) "Practical performance of several data driven bandwidth selectors" (with discussion). *Computational Statistics*, 7, 251-270.

[18] Silverman, B. W. (1986) *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, London.

Figure 1

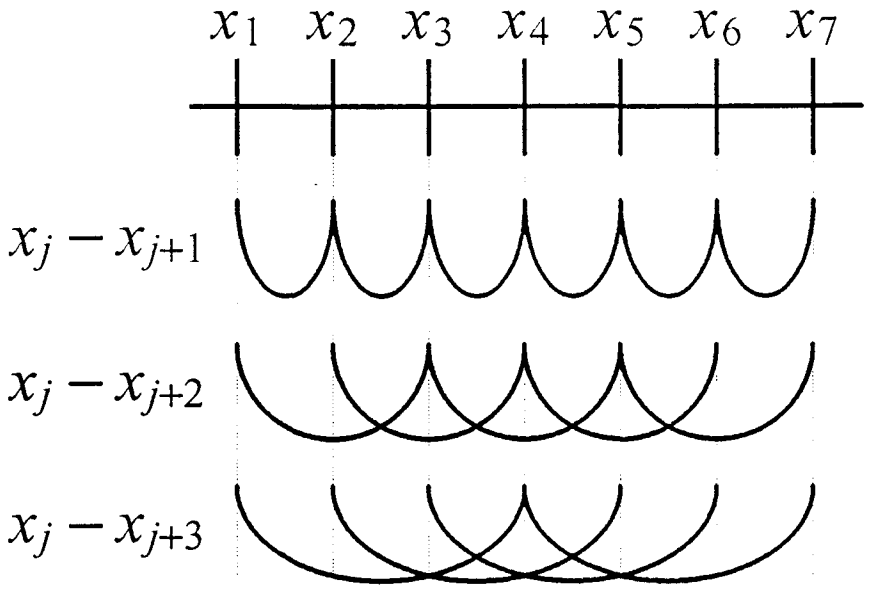

Figure 2

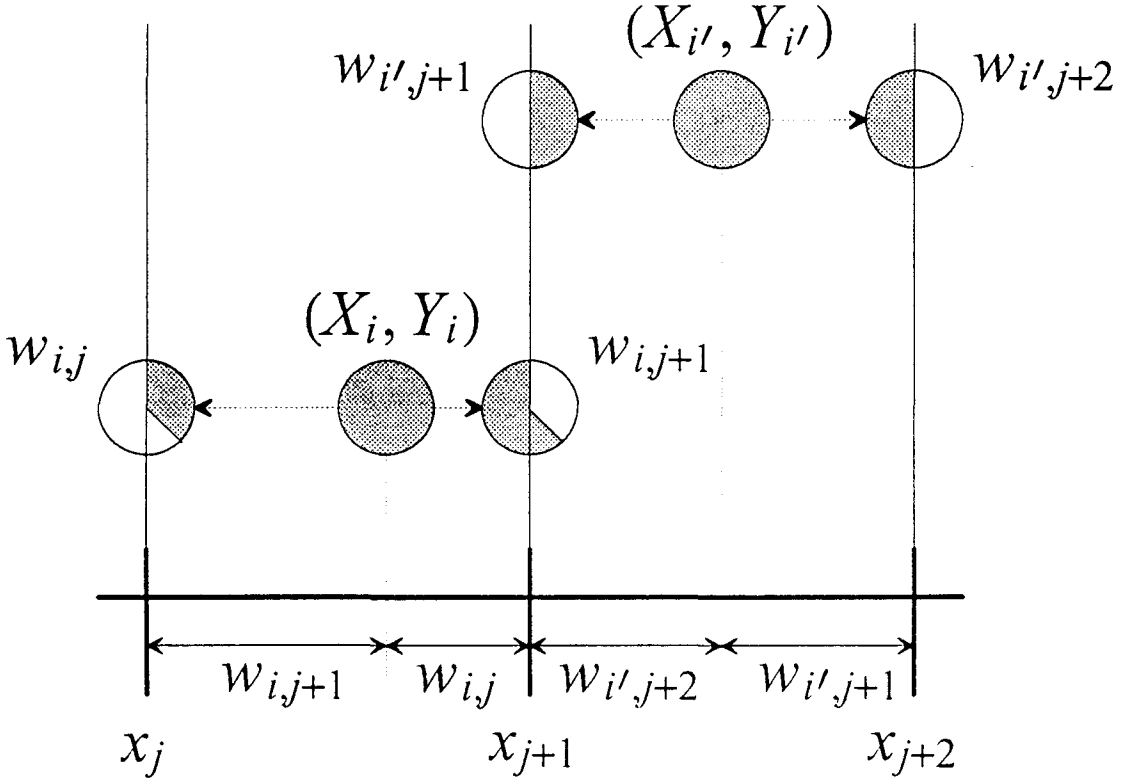

Figure 3a

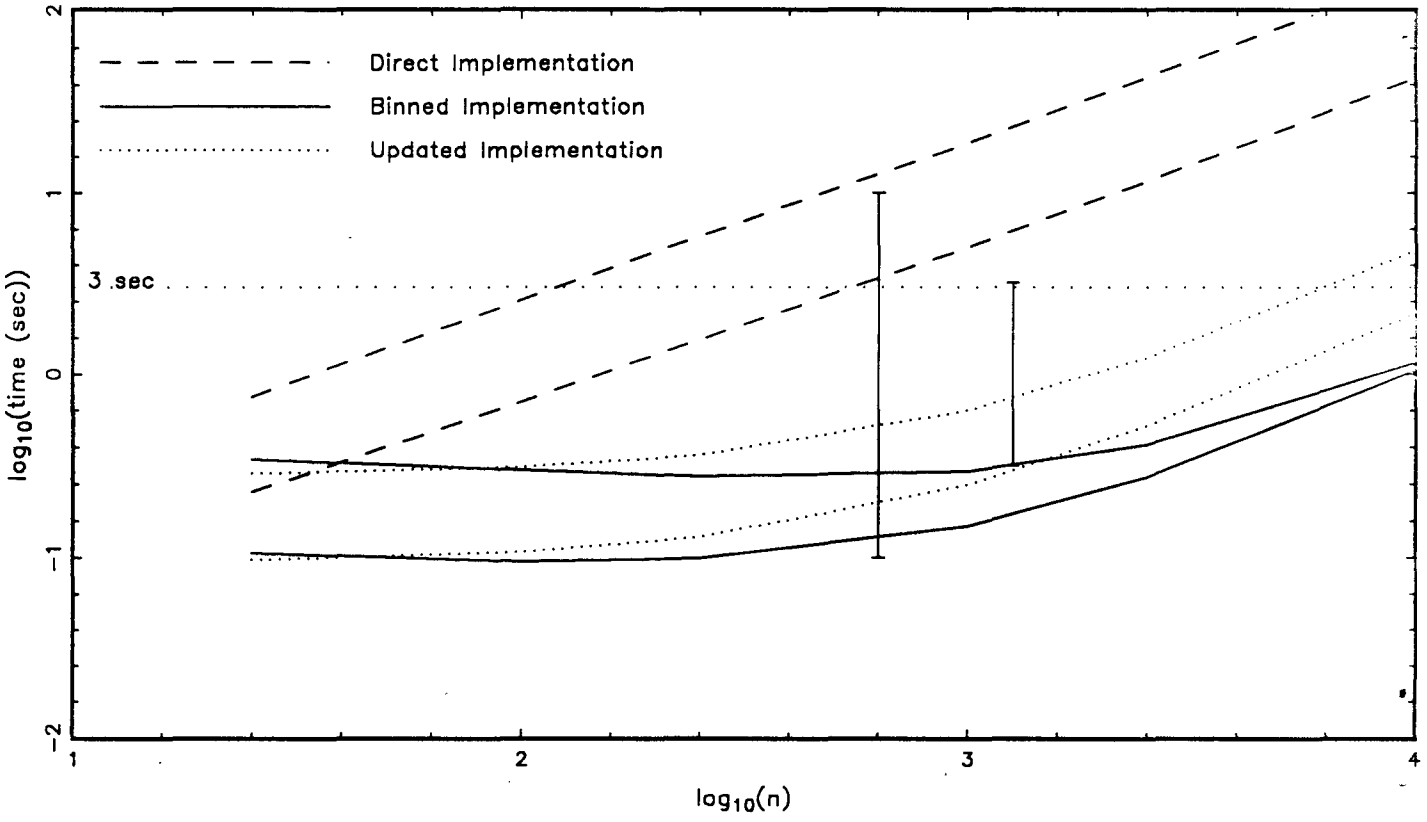

Figure 3b

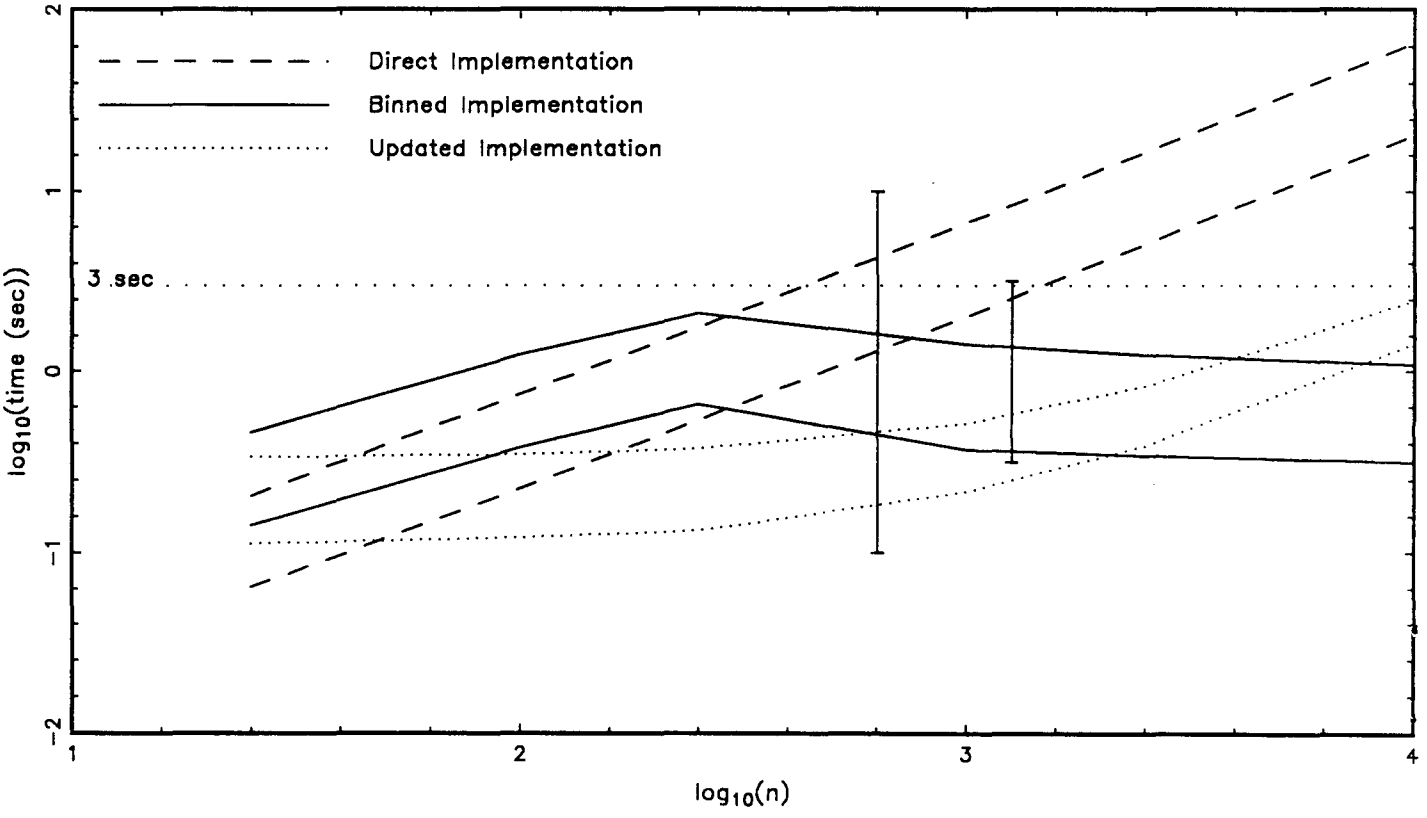

Figure 4a

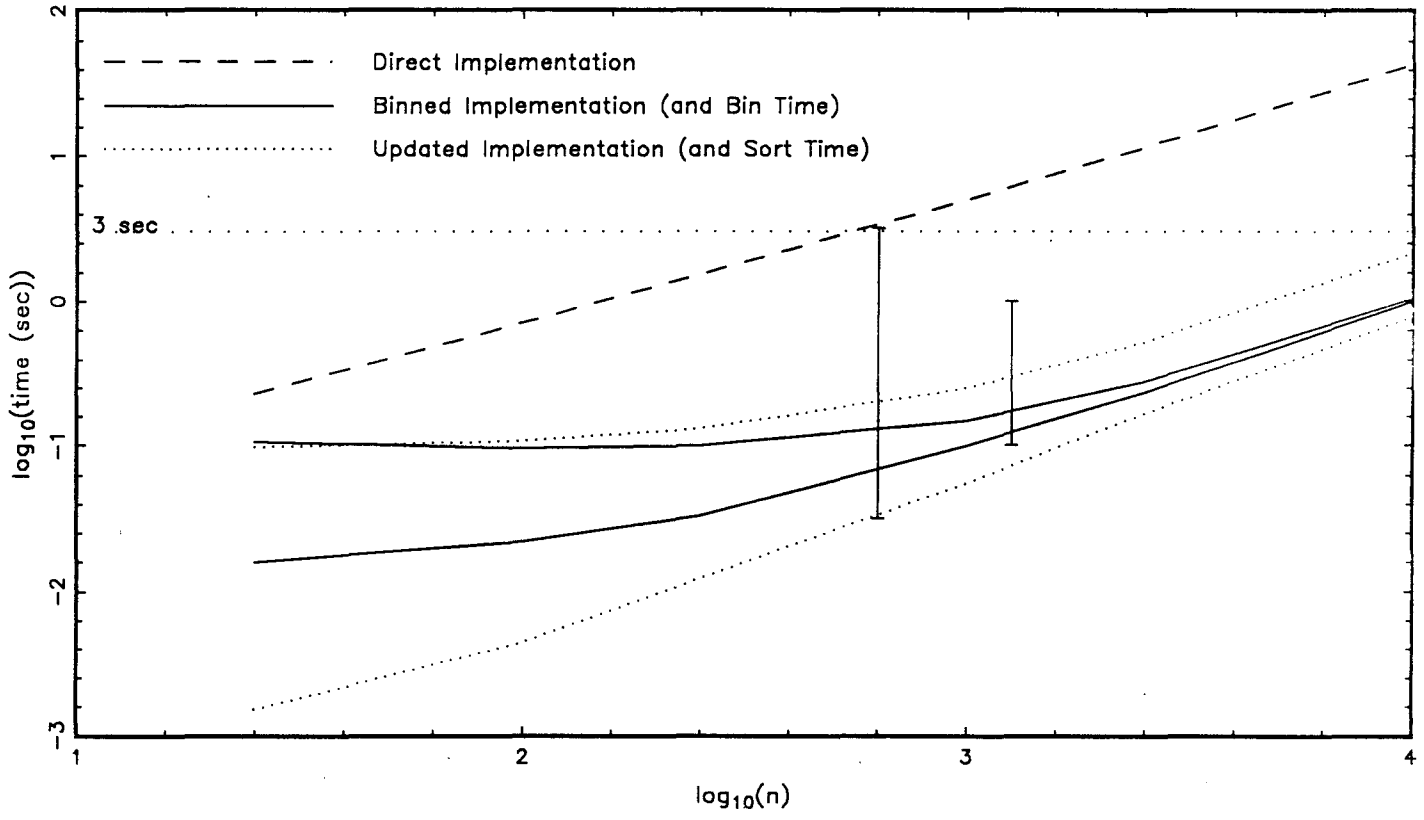

Figure 4b

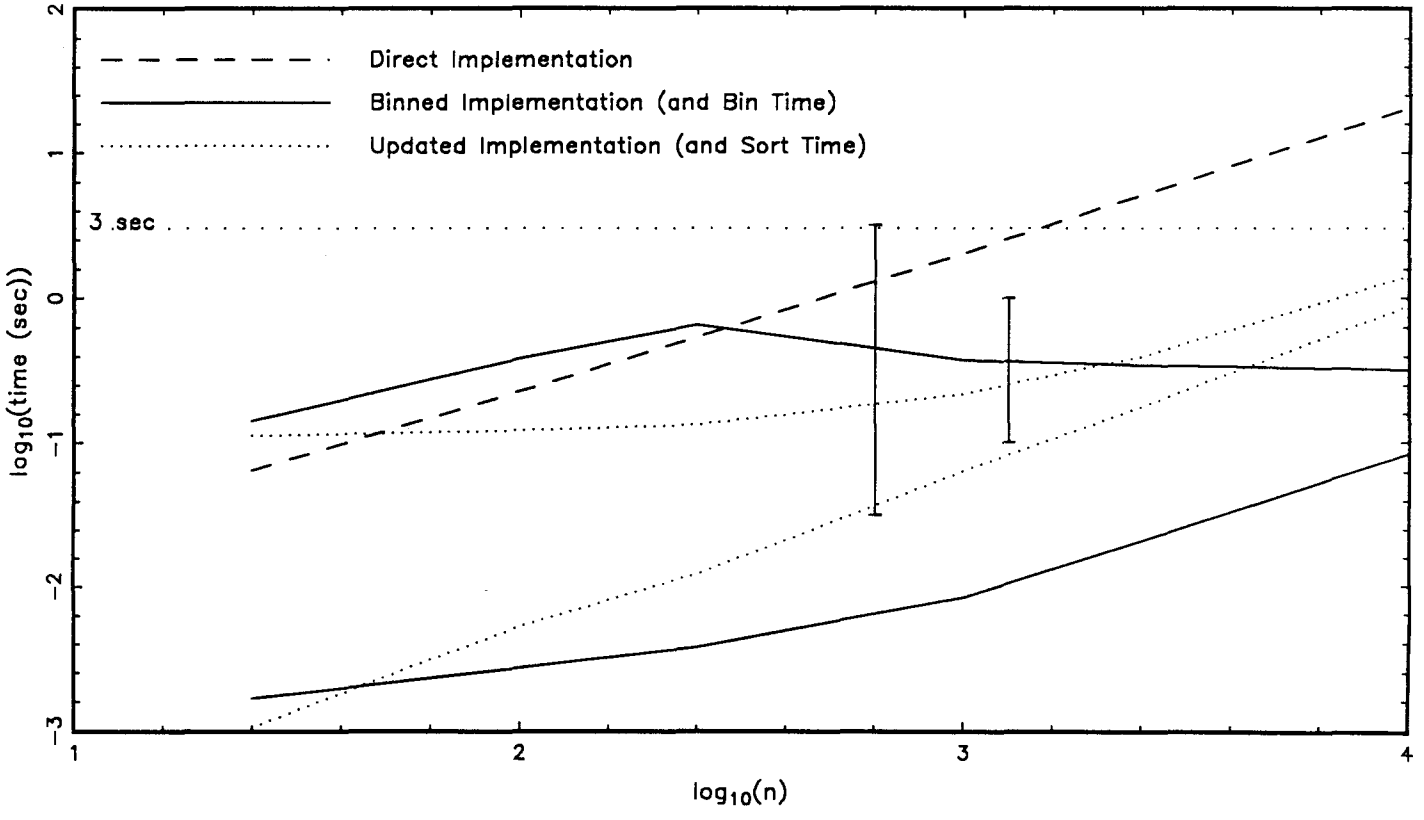

Figure 5a

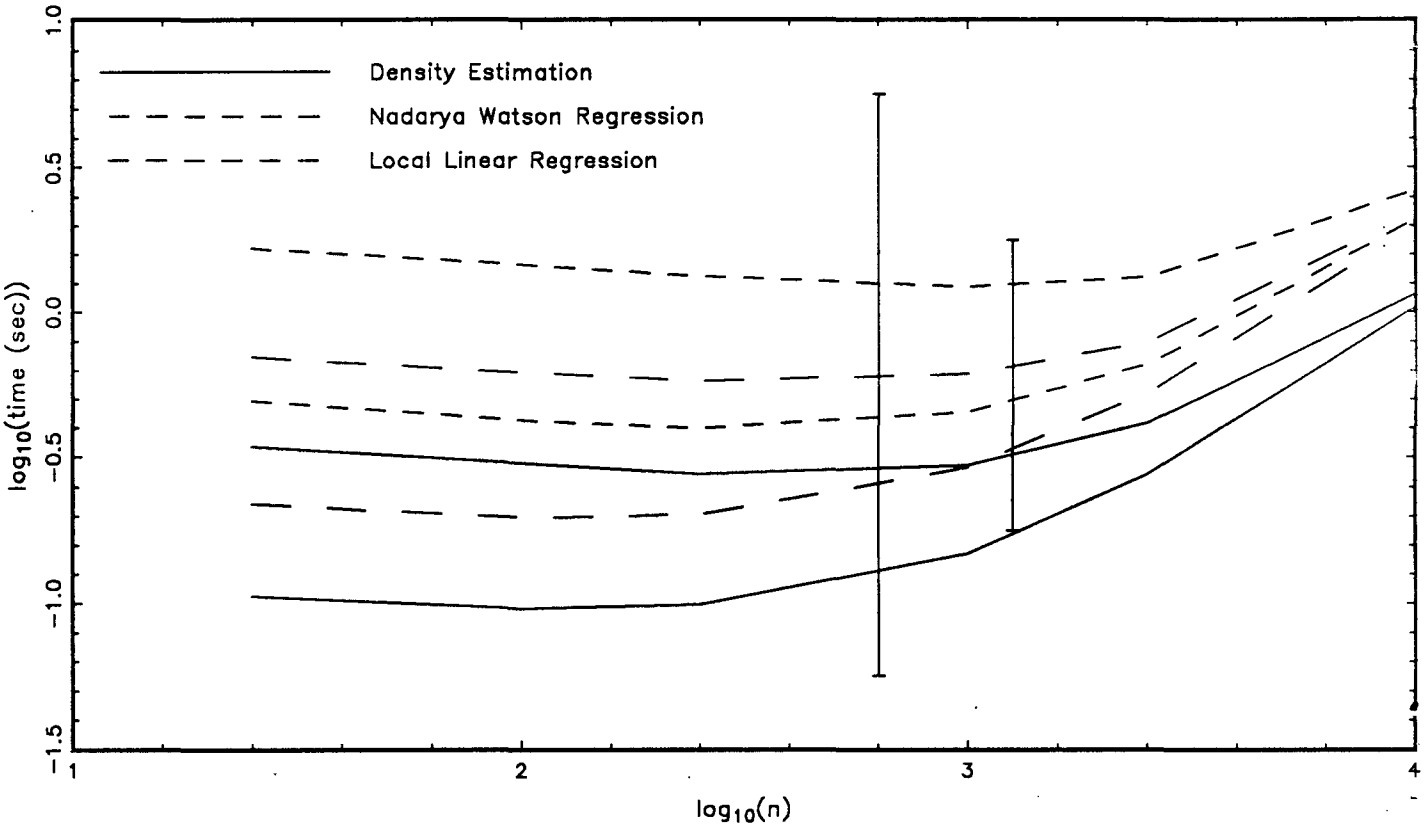

Figure 5b

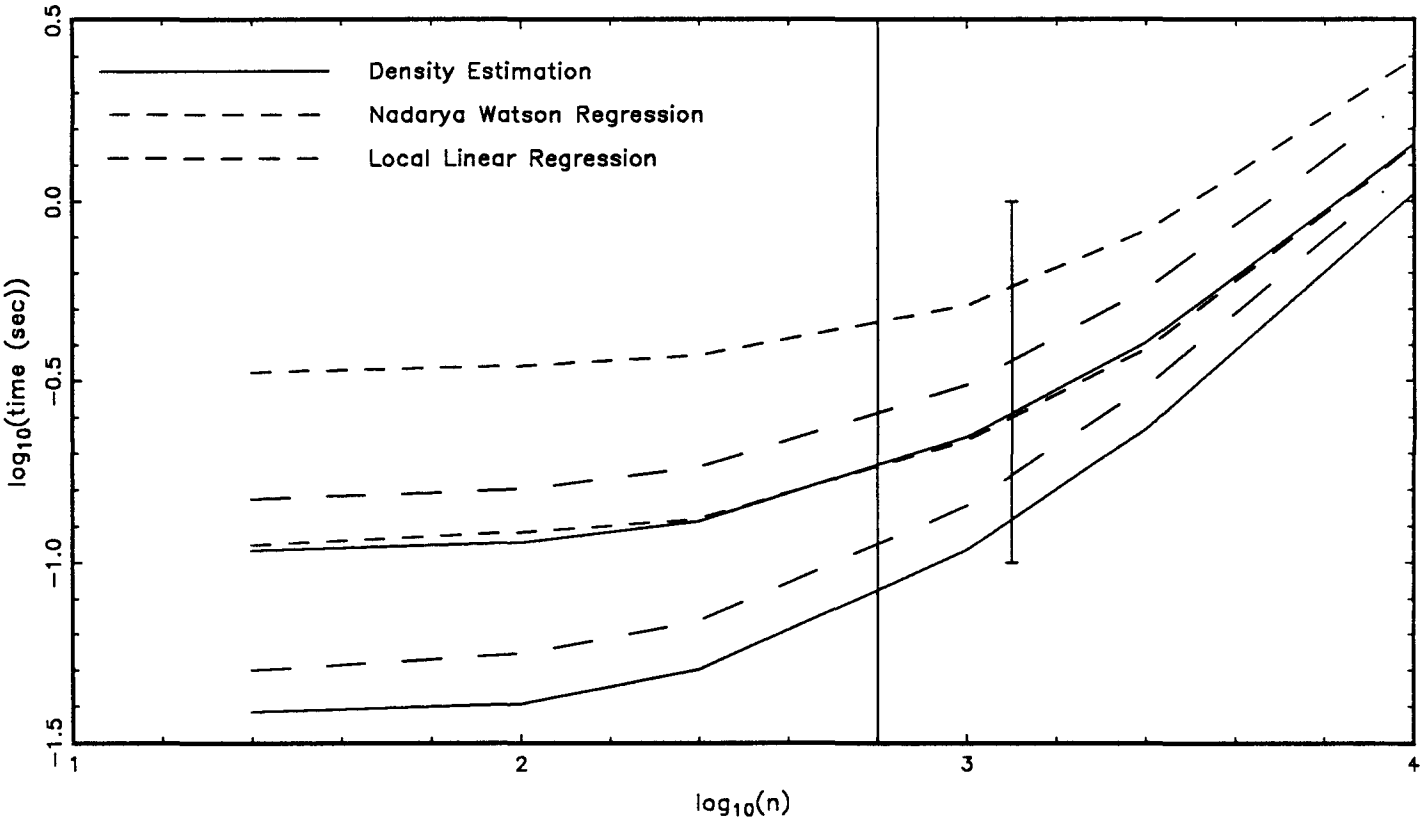

Figure 6a

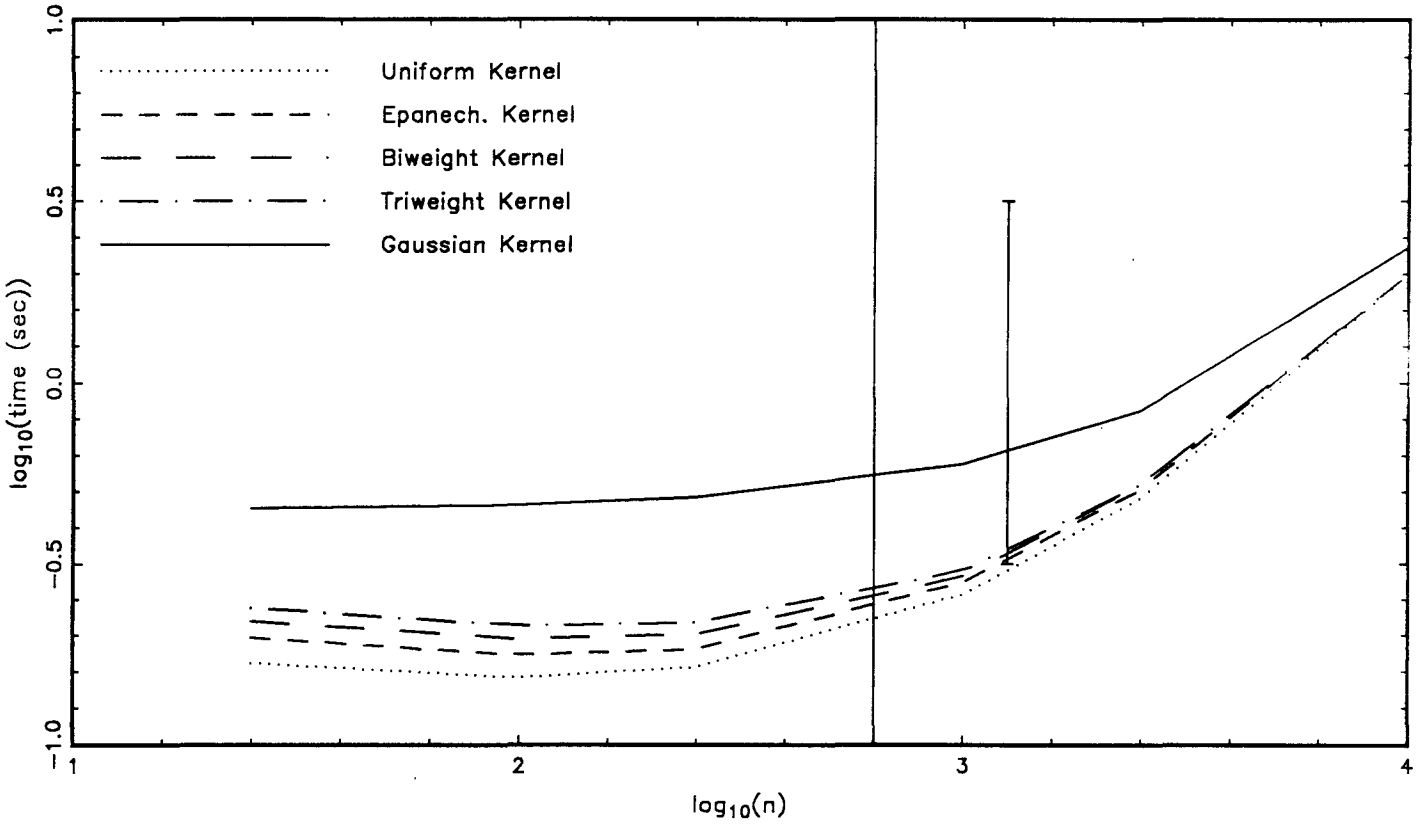

Figure 6b

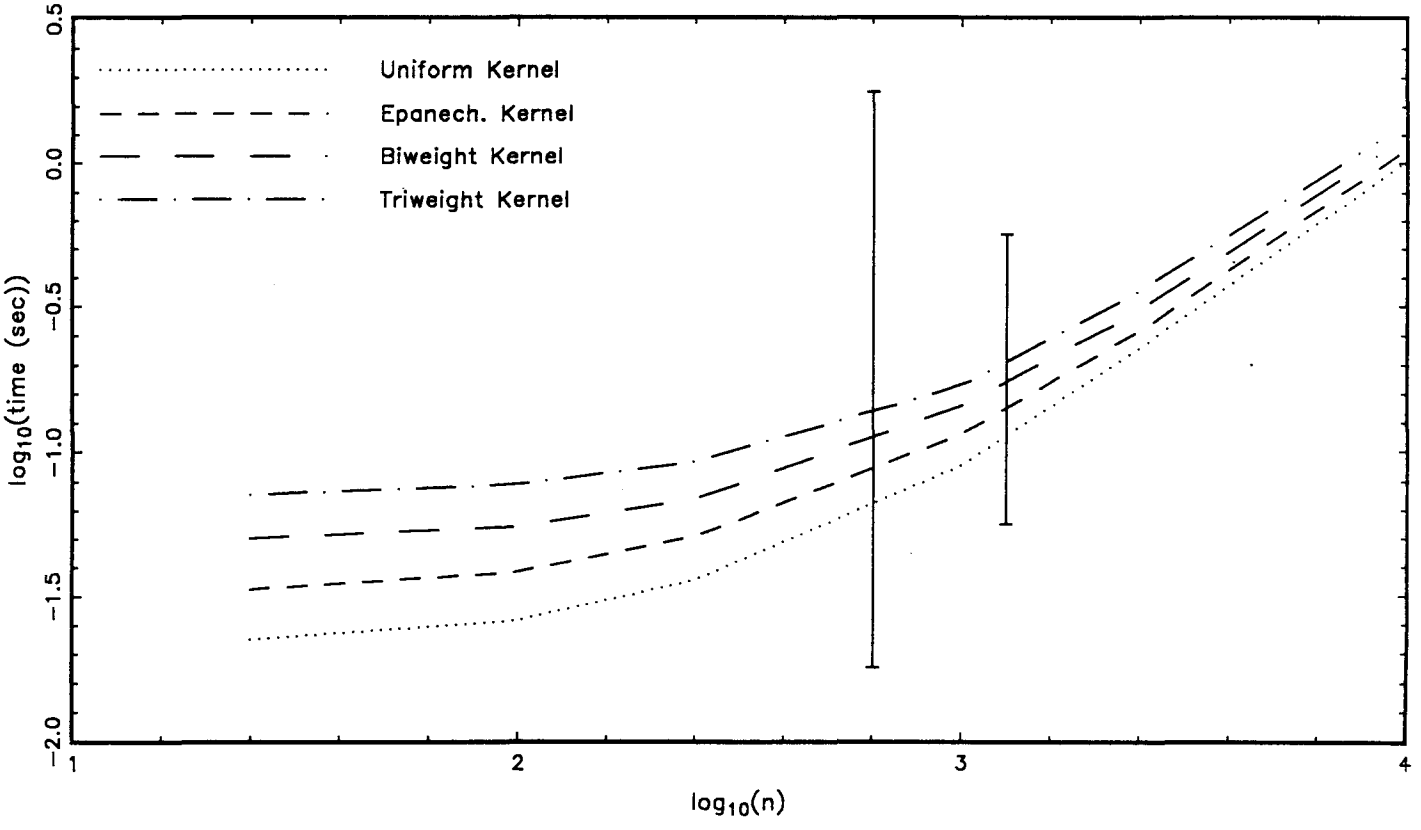

Figure 7a

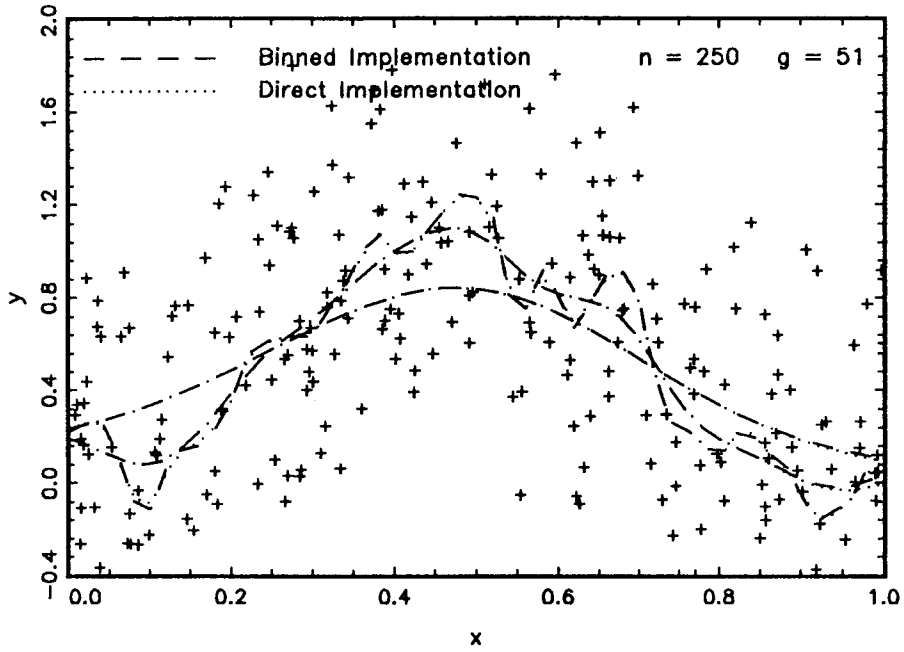

Figure 7b

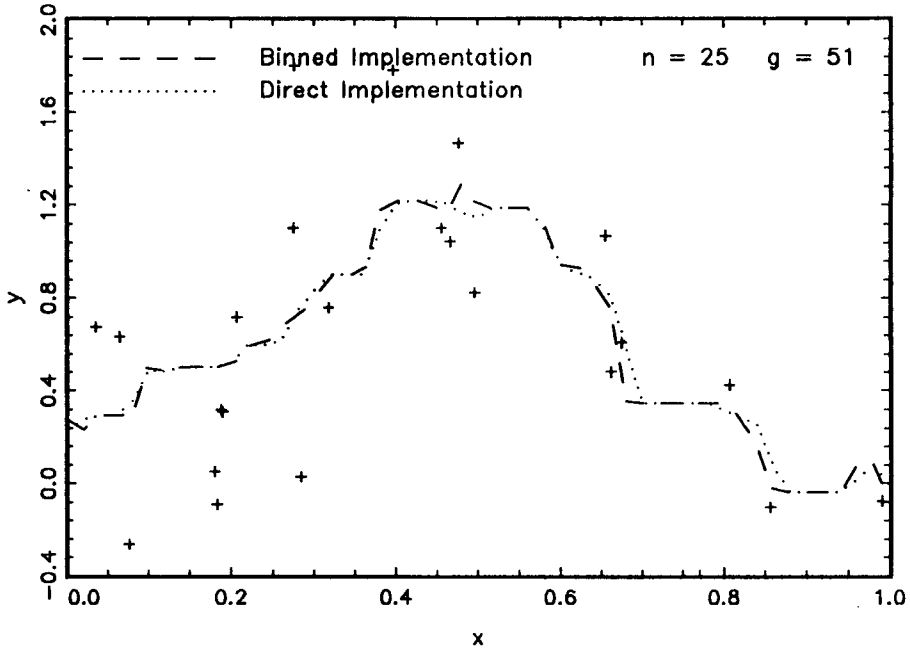

Figure 7c

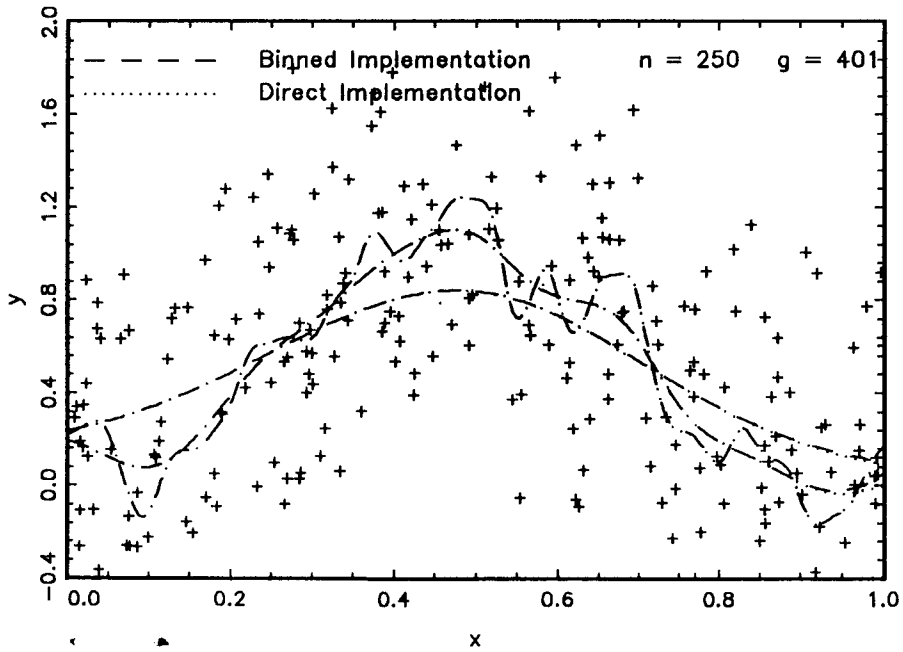

Figure 7d

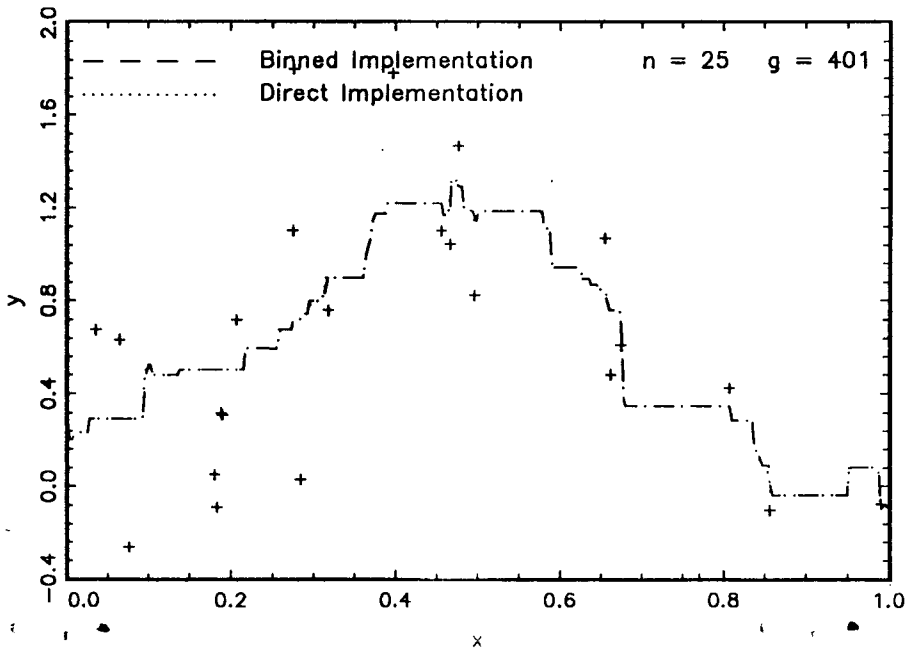

Figure 8a

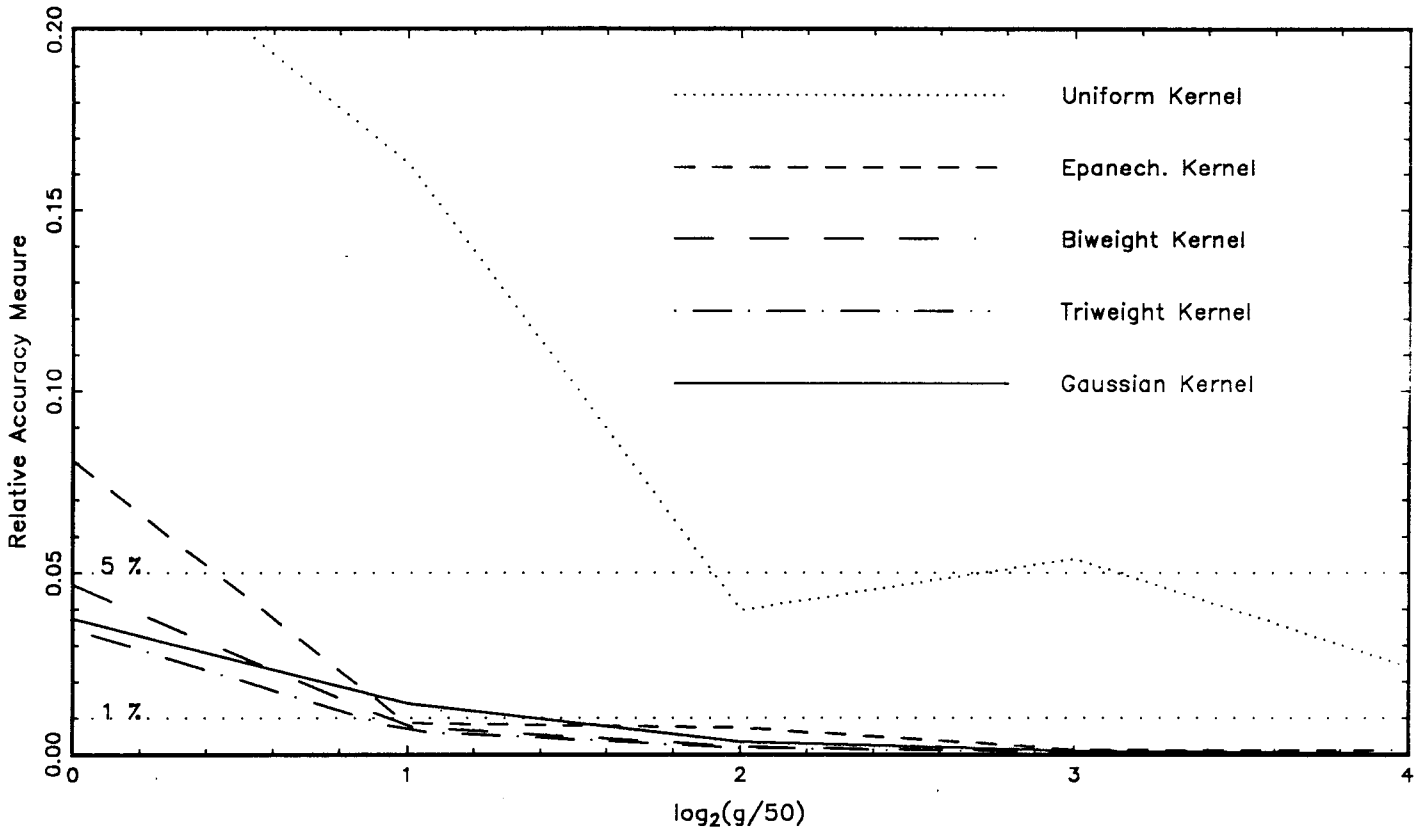

Figure 8b

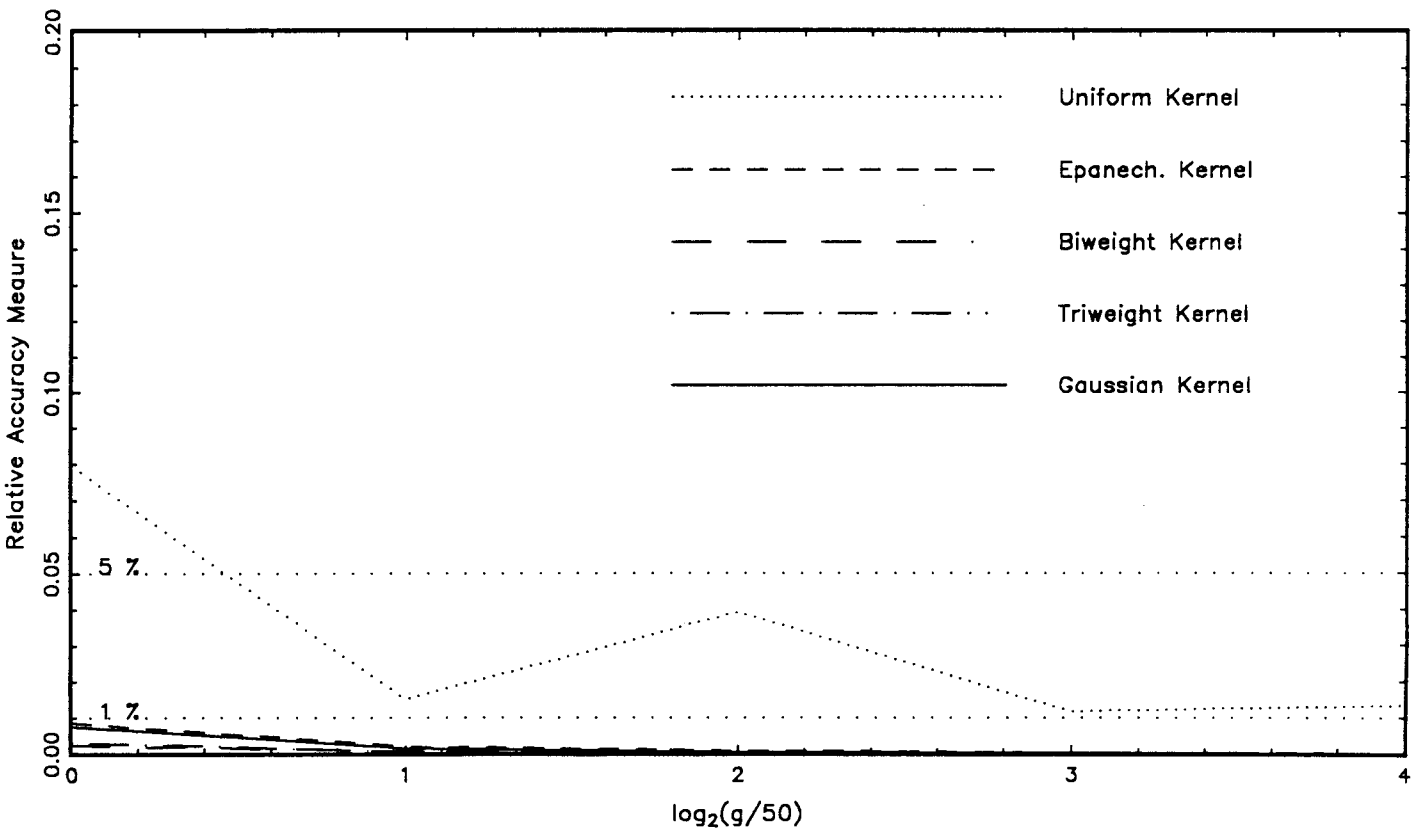

Figure 9

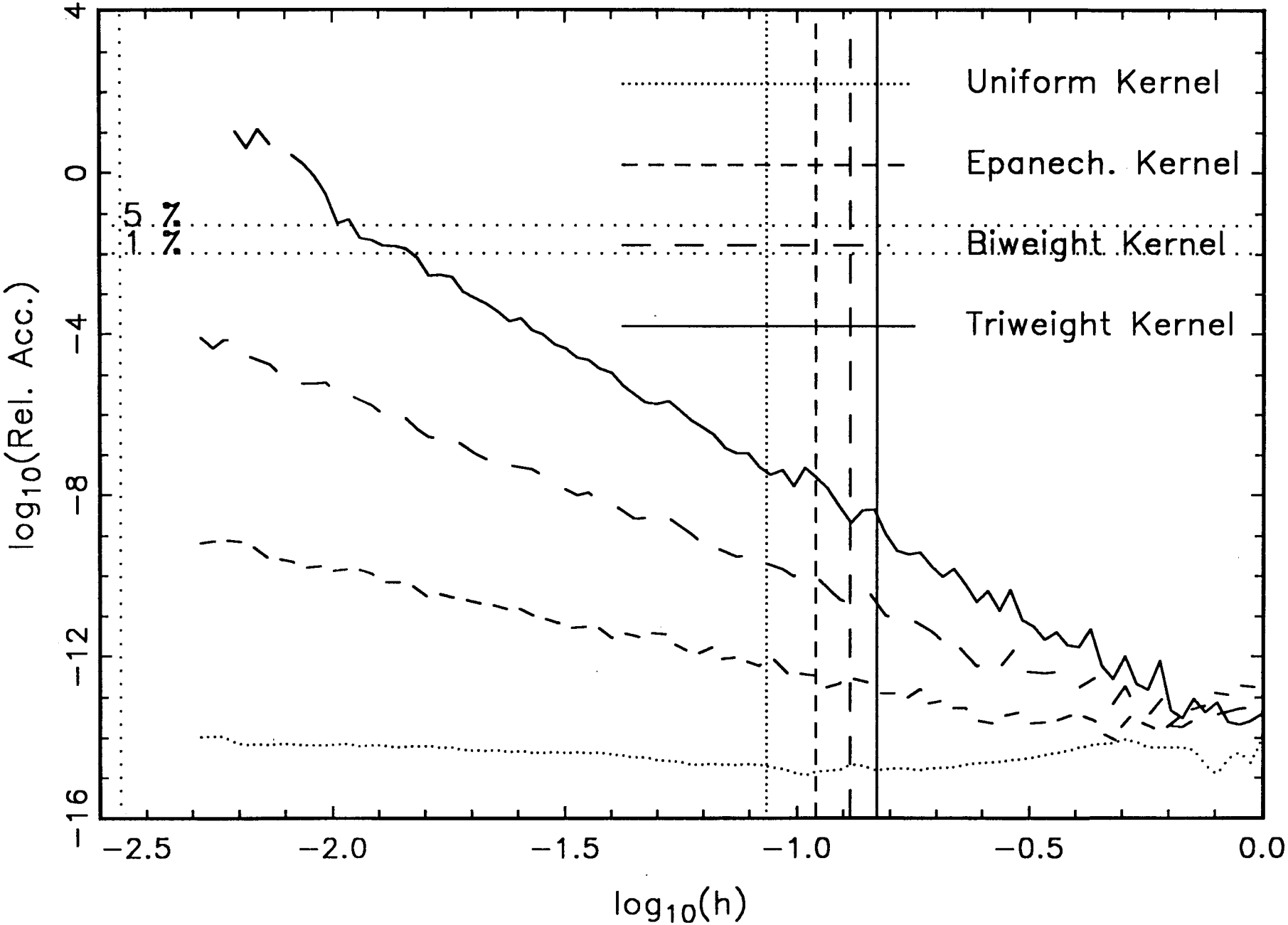
Uniform Kernel
Epanech. Kernel
Biweight Kernel
Triweight Kernel
5 %
1 %
log10(Rel. Acc.)
log10(h)
4
0
−4
−8
−12
−16
−2.5
−2.0
−1.5
−1.0
−0.5
0.0

Figure 10a

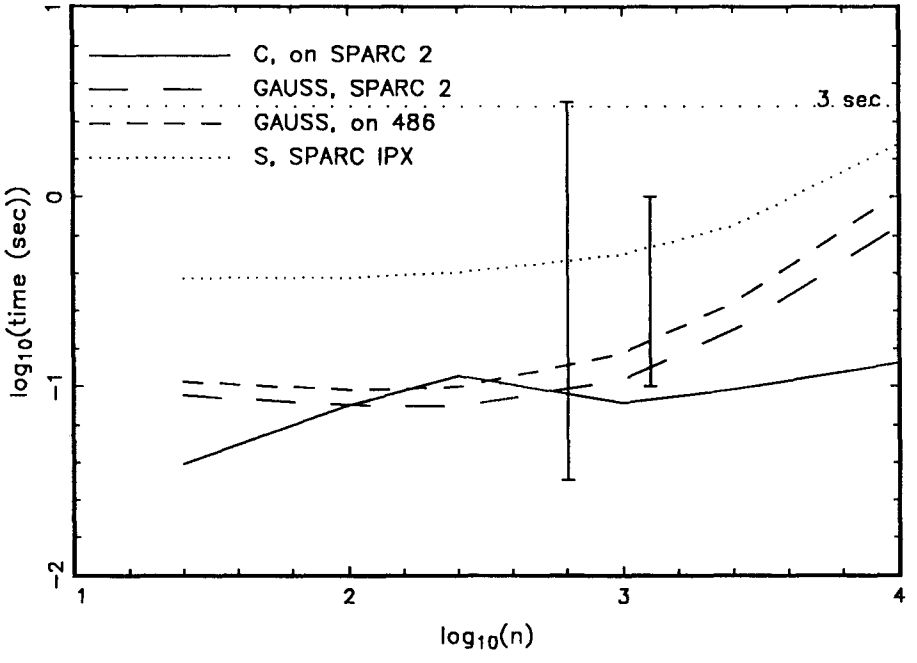

Figure 10b

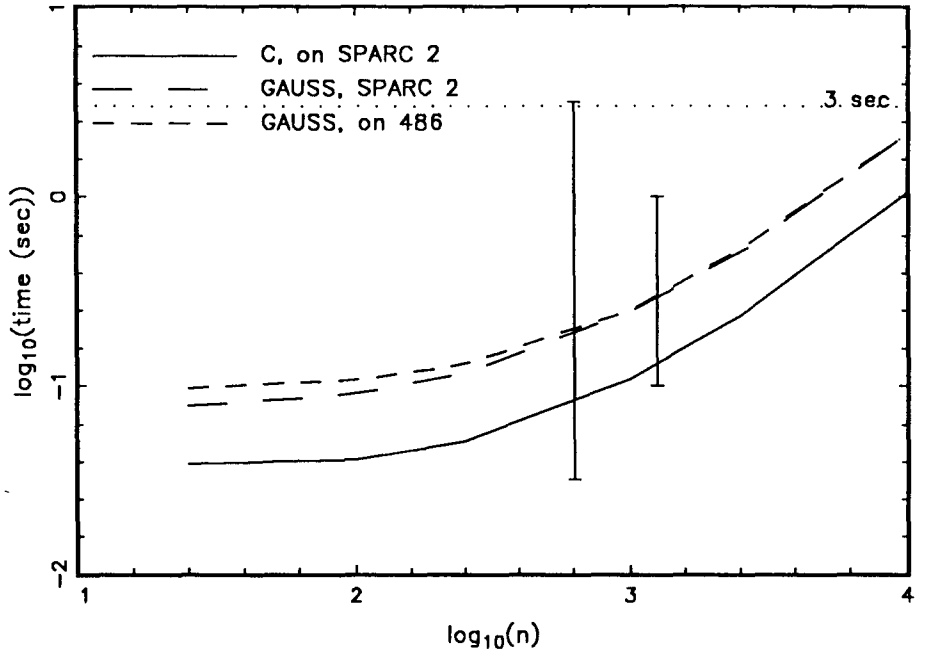

Figure 10c

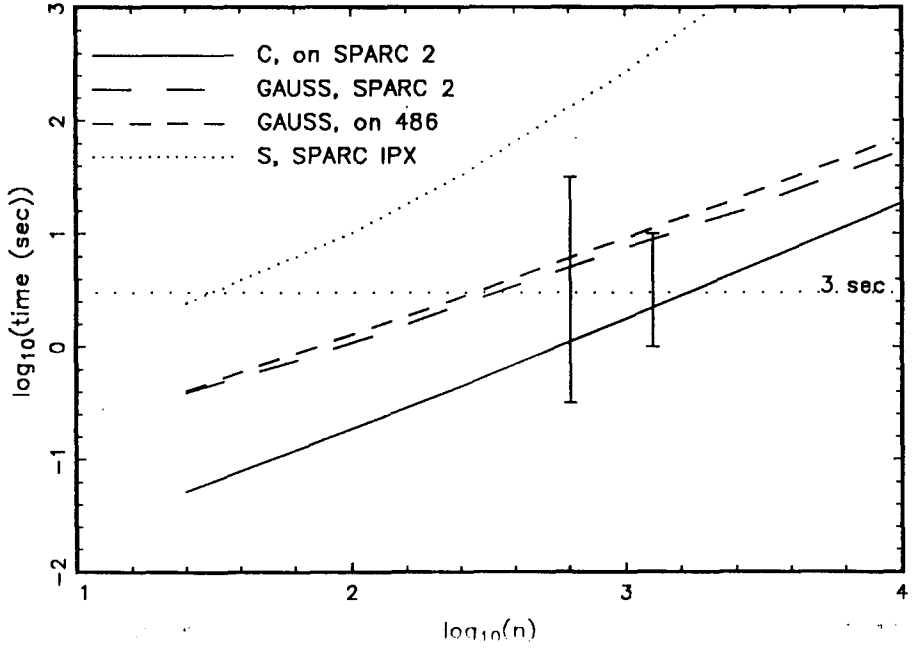

Figure 10d

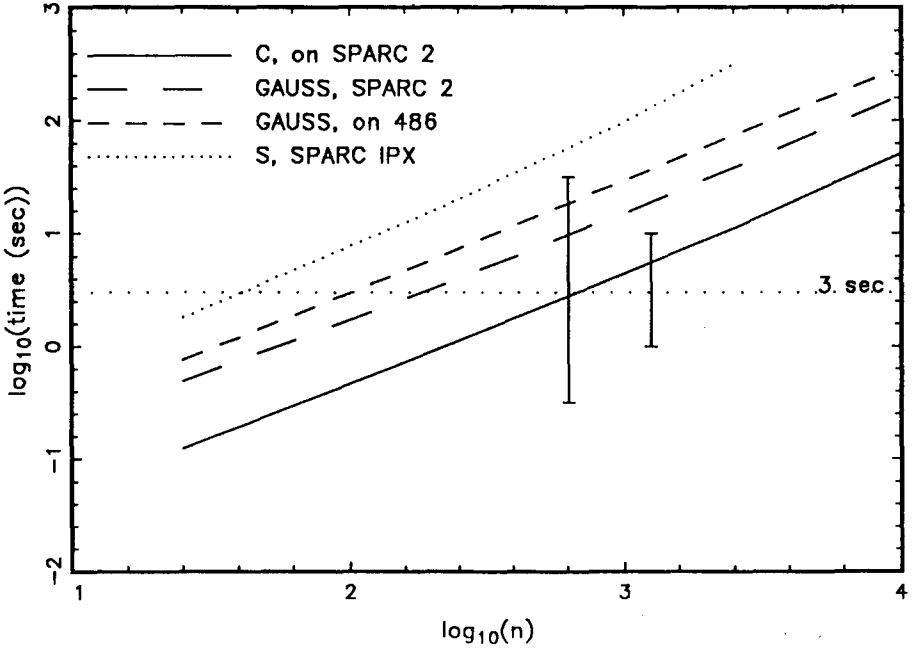

Figure 11a

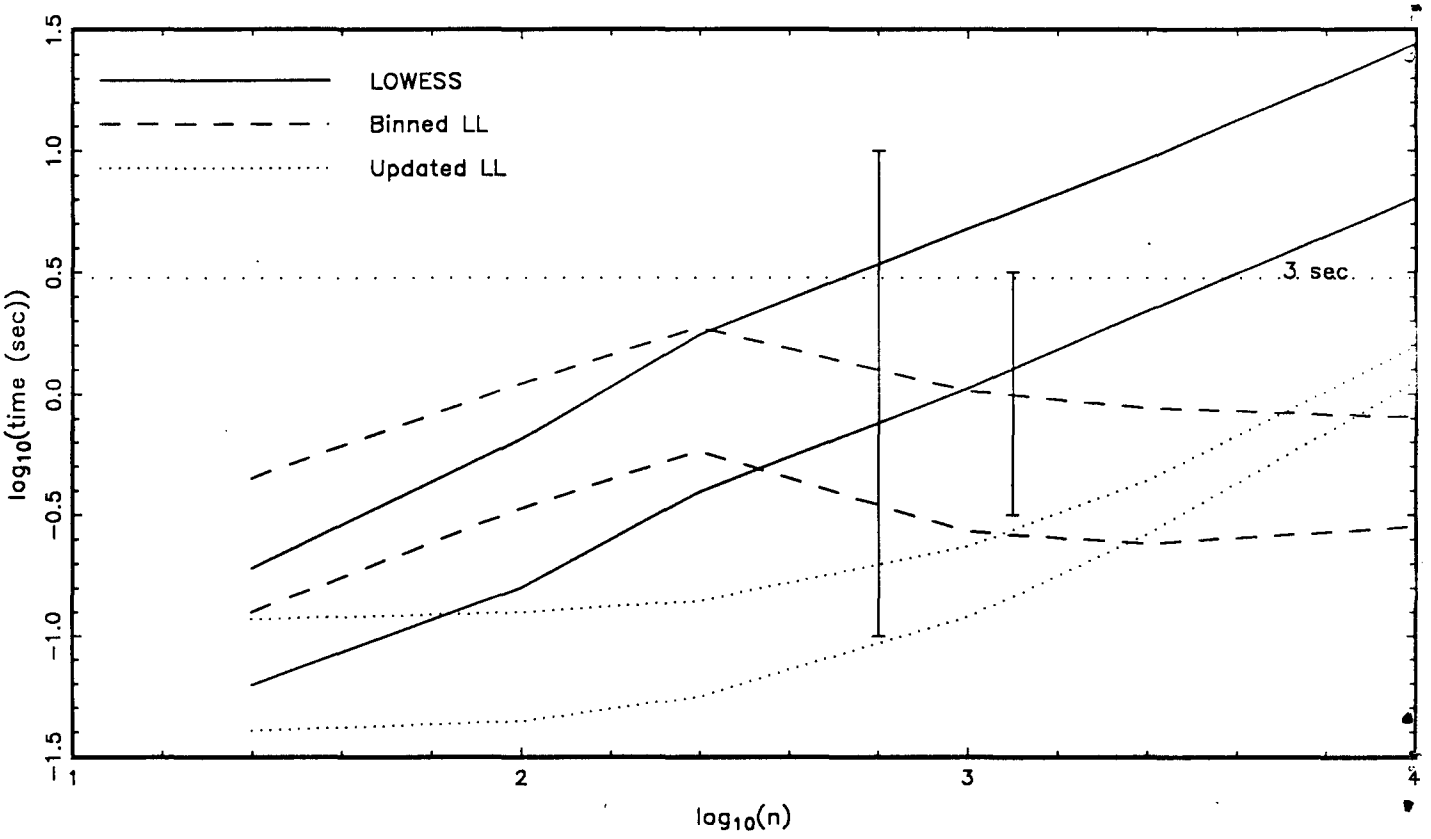

Figure 11b