IMPLEMENTATION OPTIONS FOR BLOCK FLOATING POINT DIGITAL FILTERS

Kamen Ralev[1] *Peter Bauer*[2]

Laboratory of Image and Signal Analysis, Department of Electrical Engineering
University of Notre Dame, Notre Dame, IN 46556, USA, fax:+1 (219) 631 4393
[1] phone:+1 (219) 631 6916, email: ralev.1@nd.edu
[2] phone:+1 (219) 631 8015, email: pbauer@mars.ee.nd.edu

ABSTRACT

Different options for block floating point filter implementation are introduced and their efficiency determined. The efficiency is quantified by the additional number of operations over those required for fixed point operation. Some of the implementations are new. It is shown that they are more efficient than the existing ones. Examples are given in which the processing time per recursion of a block floating point implementation on a fixed point processor is approximately the same as the recursion time of the corresponding fixed point implementation. Application of block floating point arithmetic to block implementations is also considered.

1. INTRODUCTION

Currently, digital filters are mostly implemented either in fixed or floating point arithmetic. Floating point is characterized by high dynamic range, while fixed point offers lower complexity and therefore lower cost and power consumption. In block floating point arithmetic a common exponent is assigned to a group of variables. Thus the computations can be carried out in fixed point arithmetic while the exponent can provide the required dynamic range. The application of block floating point arithmetic has been previously suggested for a wide variety of signal processing tasks such as fast Fourier/Hartley transform, direct form filters [1, 2], state space filters [3], and other filter structures [4]. Also in [5] the application of block floating point arithmetic was considered for some general computational problems such as vector orthogonalization, solution of linear systems and others. Most of these references concentrate on roundoff properties of the block floating point systems either disregarding the actual implementation or providing it for a particular structure only. In this paper different implementations of block floating point arithmetic are investigated in terms of their complexity i.e. number of operations. Since they can be efficiently implemented using fixed point processors the comparison is based upon fixed point systems implementation. Usually fixed point implementation requires some sort of scaling which slows the system down. Unless otherwise noted the comparison will be based on unscaled fixed point implementation.

2. STATE SPACE IMPLEMENTATION

Throughout this paper $\mathbf{R}$ denotes the set of real numbers. All vectors $\mathbf{x} \in \mathbf{R}^n$ are boldfaced, p_{ij} denotes i,j element of some matrix P and P^T denotes the transpose of P.

In this section we propose a procedure for implementing the state space discrete system

$$\mathbf{x}(k+1) = A\mathbf{x}(k) + B\mathbf{u}(k) \quad (1)$$

$$\mathbf{y}(k) = C\mathbf{x}(k) + D\mathbf{u}(k) \quad (2)$$

in block floating point arithmetic, which significantly simplifies the realization suggested by Sridharan in [3], and improves roundoff properties. Here $\mathbf{x}(k) \in \mathbf{R}^{n\times 1}$ is the state vector, $\mathbf{u}(k) \in \mathbf{R}^{m\times 1}$ is the input, $\mathbf{y}(k) \in \mathbf{R}^{p\times 1}$ is the output, and A, B, C, D are matrices of appropriate size. To select a block of variables with a common exponent we write the state space system as $\mathbf{x}(k+1) = F\mathbf{z}(k)$ and $\mathbf{y}(k) = G\mathbf{z}(k)$, where $F := [A\ B]$, $G := [C\ D]$ and $\mathbf{z}(k) := [\mathbf{x}^T(k)\ \mathbf{u}^T(k)]^T$. From here we see that we can introduce a common exponent on $\mathbf{z}(k)$, i.e. on $\mathbf{x}(k)$ and $\mathbf{u}(k)$.

Usually the coefficients in block floating point filters are properly scaled fixed point numbers. However a more natural choice is to consider F, G as block floating point matrices. P is a block floating point matrix if it is of the form $P = \hat{P}2^{\mathrm{E}_P}$, where E_P is an integer called the exponent of P and $\hat{P}$ is a fixed point matrix which is block normalized meaning that $0.5 \le \max_{ij} |p_{ij}| < 1$. From the definition of P it follows that

$$\mathrm{E}_P = \left\lfloor \log_2 \left(\max_{i,j} \{|p_{ij}|\} \right) \right\rfloor + 1 \quad (3)$$

where $\lfloor\ \rfloor$ denotes the *floor* function i.e. $\lfloor a \rfloor$ is the integer closest to but not exceeding a. The process of obtaining a block floating point matrix from some arbitrary matrix through a proper quantization will be called block formatting.

Using block floating point matrices, the system (1,2) can be written as

$$\mathbf{x}(k+1) = \hat{F}\hat{\mathbf{z}}(k)\, 2^{\mathrm{E}_{\mathbf{z}(k)}+\mathrm{E}_F} \quad (4)$$

$$\mathbf{y}(k) = \hat{G}\hat{\mathbf{z}}(k)\, 2^{\mathrm{E}_{\mathbf{z}(k)}+\mathrm{E}_G} \quad (5)$$

The new state and output can be computed by first computing the products $\hat{F}\hat{\mathbf{z}}(k)$ and $\hat{G}\hat{\mathbf{z}}(k)$ in fixed point arithmetic, then determining the new block exponent $\mathrm{E}_{\mathbf{z}(k+1)}$ and finally block formatting $\mathbf{z}(k+1)$. Traditionally, $\mathbf{u}(k)$ and $\mathbf{y}(k)$ were represented in fixed point format [1, 2]. However greater advantages can be obtained if they are in floating

point format [4]. First considered is the case when $\mathbf{u}(k)$ and $\mathbf{y}(k)$ are in floating point format. The fixed point case follows from there.

The computations in (4,5) can be done using the following algorithm which will be called *basic*. Let E_{min} be the minimum representable exponent. Let l_m be the available wordlength and let $\mathrm{m_a}$ denote the mantissa of a i.e. the most significant l_m bits properly quantized. E_a is the exponent of a according to (3). $\mathrm{fxp}(\cdot)$ denotes fixed point operation between the arguments as for example $\mathrm{fxp}(\mathrm{a}2^{\mathrm{E_b}})$ denotes a shift of a by E_b bits with a proper quantization afterwards. The guard bits necessary to accommodate the sum of $n+m$ fixed point numbers in the implementation of (1,2) are assumed to be present. Assume also that $\mathbf{z}(k-1)$ is already block formatted. Now we continue with the computation of $\mathbf{x}(k)$ and the block formatting of $\mathbf{z}(k)$. s_i denotes the state components x_i before the block normalization. E_d is intermediate variable which eventually contains $\mathrm{E}_{\mathbf{z}(k)} - \mathrm{E}_{\mathbf{z}(k-1)}$.

$E_d = E_{min}$
for $i = 1, \cdots, n$
  Compute $s_i = \mathrm{fxp}\left(\sum_{\mathrm{j}=1}^{\mathrm{n}} \hat{\mathrm{f}}_{\mathrm{ij}}\hat{\mathrm{z}}_{\mathrm{j}}(\mathrm{k}-1)\right)$
  Normalize s_i i.e. find m_{s_i}, E_{s_i}
  If $E_d < \mathrm{E}_{s_i}$ then $E_d = \mathrm{E}_{s_i}$
end for
$E = E_d + (\mathrm{E}_{\mathbf{x}(k-1)} + \mathrm{E}_F)$
for $i = 1, \cdots, m$
  If $E < \mathrm{E}_{u_i(k)}$ then $E = \mathrm{E}_{u_i(k)}$
end for
$E_d = E - (\mathrm{E}_{\mathbf{x}(k-1)} + \mathrm{E}_F)$

With this step we have introduced n normalizations, $n+m$ comparisons and 3 additions compared to pure fixed point operation. It is assumed that $(\mathrm{E}_{\mathbf{x}(k-1)} + \mathrm{E}_F)$ is computed only once. So far we have computed $\mathbf{x}(k)$, $\mathrm{E}_{\mathbf{z}(k)} = E$ and $\mathrm{E}_{\mathbf{z}(k)} - \mathrm{E}_{\mathbf{z}(k-1)} = E_d$. In the next step the output $\mathbf{y}(k)$ is computed. First $\mathbf{z}(k)$ is block formatted during the computation of $y_1(k)$. r_i denotes the output components y_i before normalization.

$r_1 = 0$
$E = \mathrm{E}_{\mathbf{z}(k)} + \mathrm{E}_G$
for $j = 1, \cdots, n$
  $\hat{z}_j(k) = \mathrm{fxp}\left(\mathrm{m}_{\mathrm{s_j}} 2^{\mathrm{E_{s_j}} - \mathrm{E_d}}\right)$
  $r_1 = \mathrm{fxp}\left(\mathrm{r}_1 + \hat{\mathrm{g}}_{\mathrm{ij}}\hat{\mathrm{z}}_{\mathrm{j}}(\mathrm{k})\right)$
end for
for $j = 1, \cdots, m$
  $\hat{z}_j(k) = \mathrm{fxp}\left(\mathrm{m}_{\mathrm{u_j(k)}} 2^{\mathrm{E_{u_j(k)}} - \mathrm{E_{z(k)}}}\right)$
  $r_1 = \mathrm{fxp}\left(\mathrm{r}_1 + \hat{\mathrm{g}}_{\mathrm{ij}}\hat{\mathrm{z}}_{\mathrm{j}}(\mathrm{k})\right)$
end for. Normalize r_1 i.e. find m_{r_1} and E_{r_1}
$y_1(k)$ is given by $\mathrm{m}_{\mathrm{y_1(k)}} = \mathrm{m}_{\mathrm{r_1}}$, $\mathrm{E}_{y_1(k)} = E + \mathrm{E}_{r_1}$.

Next we continue with the computation of the rest of $y_i(k)$.

for $i = 2, \cdots, p$
  $r_i = \mathrm{fxp}\left(\sum_{\mathrm{j}=1}^{\mathrm{n+m}} \hat{\mathrm{g}}_{\mathrm{ij}}\hat{\mathrm{z}}_{\mathrm{j}}(\mathrm{k})\right)$
  Normalize r_i i.e. find $\mathrm{m}_{\mathrm{r_i}}$ and E_{r_i}
  $y_i(k)$ is given by $\mathrm{m}_{\mathrm{y_i(k)}} = \mathrm{m}_{\mathrm{r_i}}$, $\mathrm{E}_{y_i(k)} = E + \mathrm{E}_{r_i}$.
end for.

Comparing with fixed point operation the output computation requires additional p normalizations, $n+m$ shift operations and $n+m+p+1$ summations.

The total number of *extra* operations required by the approach presented here is $4+n+m+p$ summations, $n+m$ shifts, $n+p$ normalizations, $n+m$ comparisons. If the input and the output are fixed point numbers it not necessary to normalize the output variables nor to compare E with all $\mathrm{E}_{u_i(k)}$. In this case the number of *extra* opearations required is $4+n$ summations, $n+m+p$ shifts, n normalizations, $n+1$ comparisons. This is significantly better than Sridaran's approach given in [3] which for a SISO system, i.e. $m = p = 1$, introduces up to n^2+2n shifts, $2n$ comparisons, n normalizations and n^2 additions. An exact estimation of the computation time increase is impossible without considering a specific processor.

A similar analysis can show that a pure floating point realization of this system on a fixed point processor will require up to $(n+m-1)(n+p-1)$ extra additions and the same number of shifts and normalizations.

3. APPLICATION TO OTHER STRUCTURES

The same approach can be applied to any system of the form

$$\mathbf{x}(k+1) = \phi(\hat{\mathbf{x}}(k), \hat{\mathbf{u}}(k))\, 2^{\mathrm{E}_{\mathbf{z}(k)} + \mathrm{E}_F} \quad (6)$$

$$\mathbf{y}(k) = \psi(\hat{\mathbf{x}}(k), \hat{\mathbf{u}}(k))\, 2^{\mathrm{E}_{\mathbf{z}(k)} + \mathrm{E}_G} \quad (7)$$

where ϕ, ψ, are some functions determining how the new state and outputs are calculated from the previous one. For example, for lattice and ladder structures ϕ and ψ are linear and the realization (6,7) requires that all coefficients needed to compute $\mathbf{x}(k+1)$ have a common exponent E_F and those needed to compute $\mathbf{y}(k)$ have a common exponent E_G. In all cases the block floating point overhead will be the same as in the state space structures.

Generally the block floating point arithmetic is not as efficient with direct form structures as with the other structures. The reason for that is the small number of fixed point operations required relatively to the constant amount of block floating point overhead. If block floating point direct form implementation is really needed Sridaran's method [2] can be considered.

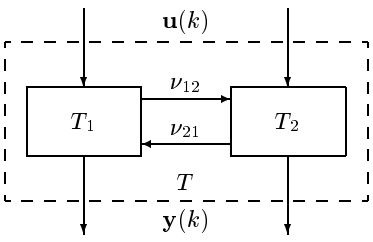

Figure 1. Connecting block floating point subsystems

The block floating point subsystems can be conveniently connected even if they do not share a common exponent. Consider the two subsystems $T_1(n_1, m_1, p_1)$ and $T_2(n_2, m_2, p_2)$ of $T(n, m, p)$ given on Fig. 1, where

$T(n,m,p)$ denotes that the system T has n states, m inputs and p outputs. These are the parameters which block floating point overhead depends on. Assume that there are ν_{12} connections from T_1 to T_2 and ν_{21} from T_2 to T_1. Define $\nu := \nu_{12} + \nu_{21}$. Then $n = n_1 + n_2$, $m + \nu = m_1 + m_2$ and $p + \nu = p_1 + p_2$. Using the analyses in the previous section it can be shown that if these connections transfer floating point numbers they will require $2\nu+4$ summations, ν shifts, ν normalizations, ν comparisons more than a regular single exponent block floating point implementation of T. If fixed-point-number connections are used only 4 summations, $\max\{\nu_{12},\nu_{21}\}$ shifts and 1 comparison will be required but the roundoff noise will be larger and additional care should be taken so that the fixed point numbers are in the proper range. The case of block floating point connections is as complex as the floating point case. However, a substantial simplification is possible, which is considered later.

This technique may be used for reducing the order of the filter components, as in the above example where $n = n_1 + n_2$. It also effectively reduces the number of required guard bits in the summations since they depend on the order of the (sub)system. It is particularly useful in cascade or parallel realizations using a block floating point unit of particular order.

4. ALTERNATIVE REALIZATIONS

A few modifications of the procedure described in Section 2. are possible in order to reduce the number of operations at the expense of roundoff errors or vice versa. For example a common exponent can be introduced for every row of $\hat{F}$ denoted by E_{F_i} and on every row of $\hat{G}$ denoted by E_{G_i}. The overall algorithm is preserved except for substituting E_{s_i} with $\mathrm{E}_{s_i} + \mathrm{E}_{F_i}$ and E_{r_i} with $\mathrm{E}_{r_i} + \mathrm{E}_{G_i}$. This requires $n+p$ summations in addition to those required by the *basic* method.

Another option is to have different exponents for $A, B, C, D, \mathbf{x}(k), \mathbf{u}(k)$. This will introduce 2 additional comparisons and $2n$ shifts in the algorithm. Other divisions of F and G into block floating point matrices can also be considered. Next, presented are two modifications of the algorithm considered of greater importance.

Preshift method In the method described in Section 2. the state is essentially first normalized so that it can be saved and then block normalized. This double "normalization" can be avoided if we prescale the state components on every iteration k, in order to avoid mantissa overflow. This is similar to Oppenheim's approach in [1] for direct form filters. To this end instead of shifting x_i by E_{s_i} during normalization and then by $E_d - \mathrm{E}_{s_i}$ during block normalization we shift them only once by $E_d - S$ during the block normalization. The constant $S := \left\lceil \log_2(\max_{\hat{\mathbf{z}}} \|\phi(\hat{\mathbf{z}})\|_\infty) \right\rceil$ is a scaling factor preventing overflow. $\lceil\ \rceil$ denotes the *ceil* function i.e. $\lceil a \rceil$ is the closest integer greater or equal to a. In this way we save n summations of $E_d - \mathrm{E}_{s_i}$ and replace n normalizations with n exponent determinations which is a faster operation. Another advantage of this method is that the guard bits for the summations are not necessary. The disadvantage is the increase in roundoff noise due to a poor utilization of the block mantissas.

Table 1. Number of additional operations, fixed point input and output

Implement.	sum.	comp.	norm.	exp.	shifts
Basic	$4+n$	$n+1$	n	–	$n+m+p$
Preshift	4	$n+1$	–	n	$n+m+p$
Inc./Pre.	4	3	–	–	$n+m+p$

The same idea can be used in connecting block floating point subsystems. If the output coefficients are pre-scaled so that $y_i(k)$ has no mantissa overflows then $\mathbf{y}(k)$ can be directly inputed to the other subsystem. Note that all $y_i(k)$ have a common exponent $\mathrm{E}_{\mathbf{z}(k)} + \mathrm{E}_G$. Therefore this procedure will require at most 4 summations, $\max\{\nu_{12},\nu_{21}\}$ shifts and 1 comparison, the same as in the fixed-point connection case.

Increment method With this method the $n+m$ comparisons in the determination of the new block exponent are reduced to $2+m$, using a generalization of Sridharan's approach. Here the block exponent is increased by one ($E_d = 1$), if a mantissa overflow has occurred in s_i for some i and decreased by one ($E_d = -1$), if all s_i are in mantissa underflow. It is assumed that overflow for some s_i and underflow for all s_i is detected using flags with no additional comparisons. This requires a modification of overflow/underflow flags not present in any off-the-shelf DSP processor. However it should be easier to implement than any hardware implementation of the previous methods. Note that the increase of the block exponent due to input increase in this case is unlimited. The method requires that $\max_{\hat{\mathbf{z}}} \|\phi(\hat{\mathbf{z}})\|_\infty < 2$. To obtain a further decrease of the number of operations the method should be combined with the preshift method, with $S = 1$.

The number of additional operations required by the different methods is summarized in Table 1 for fixed point input and output. The case of floating point input and output is obtained by adding $m+p$ summations, $m-1$ comparison, p normalization, $(-p)$ shifts to every row of Table 1. The block floating point case is given in Table 2.

Table 2. Number of additional operations, assuming block floating point input and output

Implem.	sum.	comp.	norm.	exp.	shifts
Basic	$4+n+p$	$n+1$	$n+p$	–	$n+m+p$
Preshift	$4+p$	$n+1$	p	n	$n+m+p$
Inc./Pre.	$4+p$	$2+1$	p	–	$n+m+p$

5. BLOCK IMPLEMENTATION

There is one case of great practical significance. Suppose we need to implement the SISO system

$$\mathbf{x}(k+1) = A\mathbf{x}(k) + \mathbf{b}u(k) \quad (8)$$

$$y(k) = \mathbf{c}\mathbf{x}(k) + du(k) \quad (9)$$

where $\mathbf{x}(k), \mathbf{b}, \mathbf{c}^T \in \mathbf{R}^{n\times 1}$; $u(k), y(k), d \in \mathbf{R}$ and $A \in \mathbf{R}^{n\times n}$. Let also the input u be in block floating point representation, meaning that $\{u(kL), \cdots, u(kL+L-1)\}$ share a common exponent, where L is the block length. Suppose that this representation should be preserved in the output

Table 3. Computation time for some filters (in processor cycles)

Structure	unscaled fixed point	scaled fixed point	preshift, block floating point
state space	$(n(n+4)+n+13)L+3$	$(n(n+4)+3n+15)L+3$	$(n(n+5)+3n+41)L+3$
lattice, all pole	$(4(n-1)+7)L+3$	$(6(n-1)+8)L+3$	$(5(n-1)+27)L+3$

y i.e. $\{y(kL), \cdots, y(kL+L-1)\}$ should share a common exponent. This situation arises for example in processing of audio signals conforming to NICAM (stereophonic sound system), MUSE (Japanese analog HDTV) or DSR (German Digital Satellite Radio system), which employ block floating point representation [4].

The problem can be solved by using the *basic* algorithm with $m = p = 1$, followed by block formatting of the output. A better approach is based on so called block implementation [6]:

$$\mathbf{x}(kL+L) = A^L\mathbf{x}(kL) + \mathcal{C}\mathbf{u}_L(kL) \quad (10)$$

$$\mathbf{y}_L(kL) = \mathcal{O}\mathbf{x}(kL) + \mathcal{D}\mathbf{u}_L(kL) \quad (11)$$

where $\mathbf{u}_L(kL) = [u(kL), \cdots, u(kL+L-1)]^T$, $\mathbf{y}_L(kL) = [y(kL, \cdots, y(kL+L-1)]^T$, $\mathcal{C} = [A^{L-1}\mathbf{b}, \cdots, \mathbf{b}]$, $\mathcal{O} = [\mathbf{c}^T, \cdots, \left(\mathbf{c}A^{L-1}\right)^T]^T$, and the i, j element of $\mathcal{D}$ is given by

$$\mathcal{D}_{ij} = \begin{cases} 0, & \text{if } i < j \\ d, & \text{if } i = j \\ \mathbf{c}A^{i-j-1}\mathbf{b}, & \text{if } i > j \end{cases}$$

The realization (10,11) using fixed point arithmetic was previously suggested for its improved stability and roundoff properties. For example, the poles of A^L are closer to the origin and if A is normal then A^L is also normal. Other advantages are listed in [6]. However (10,11) is also well suited for block floating point implementation using the preshift method, with scaling factor

$$S = \left\lceil \log_2 \left(\max \left(\|F\|_\infty, \|G\|_\infty \right) \right) \right\rceil, \; F = [A^L \; \mathcal{C}], \; G = [\mathcal{O} \; \mathcal{D}]$$

This implementation requires only 4 summations, $n+L$ shifts, $n+1$ comparisons and n exponent determinations more than a pure fixed point implementation. Although $\mathbf{y}_L(kL)$ may not be block normalized, all components of $\mathbf{y}_L(kL)$ have a common exponent. That is sufficient for many applications.

Many other block implementations exist in the literature and in some of them block floating point arithmetic can be efficiently applied. This will increase the dynamic range and will further improve roundoff noise and the stability properties of the system. The idea of combining "block implementations" with "block floating point implementations" to the authors' knowledge have not been suggested before.

6. EXAMPLES

Next we compare the computation time of some fixed and block floating point filters implemented using the ADSP 2181 processor (Analog Devices) [7]. A nice feature of this processor is the presence of the instruction *expadj* which allows us to determine E_d, basically without extra processor cycles. Due to some restrictions it can be used only with the preshift method.

The comparison is given in Table 3. Considered are SISO systems ($m = p = 1$), with fixed point input and output. All implementations process L samples from the input. The scaled fixed point implementation of the all-pole lattice filter is given in [7, p. 86]. It is clear from the table that in the above cases the computation time of the block floating point implementations is close to the fixed point, especially to the scaled fixed point implementation. In fact the scaled fixed point all pole lattice filter will run slower for $n > 20$ than the corresponding block floating point filter.

In general this will not be true. Block floating point filters are in fact fixed point realization in which the scaling varies and therefore should run slower. In this particular case the fixed point implementation uses reformatting of the coefficients which is not necessary in block floating point format.

The block implementation of Section 5. was also realized. The realization with block length L takes $(n+L)(4+n) + n + L + 46$ processor cycles. This clearly beats all state space fixed point realizations for sufficiently large L.

7. CONCLUSION

This paper shows that the complexity of the block floating point implementations is often very close to the complexity of fixed point implementation. Sometimes, it can be better if the fixed point implementation requires scaling. The superior roundoff properties of block floating point representation were already shown in [4]. Block implementations provide a good way of preserving the block floating point representation of the input, sometimes leading to significant computational advantages.

REFERENCES

[1] A. V. Oppenheim, "Realization of digital filters using block floating point arithmetic," *IEEE Trans. on Audio and Electroacoustics*, vol. 18, pp. 130–136, June 1970.

[2] D. Williamson, S. Sridharan, and P. G. McCrea, "A new approach to block floating point arithmetic in recursive digital filters," *IEEE Trans. Circ. Syst.*, vol. CAS-32, pp. 719–722, July 1985.

[3] S. Sridharan, "Implementation of state-space digital filters using block-floating point arithmetic," in *Proc. of 1987 IEEE International Conference on Acoustics, Speech and Signal Processing*, (Dallas, TX), pp. 908–911, 1987.

[4] K. Kalliojärvi and J. Astola, "Roundoff errors in block-floating point systems," *IEEE Trans. Signal Proc.*, vol. 44, pp. 783–790, Apr. 1996.

[5] J. H. Wilkinson, *Rounding Errors in Algebraic Processes.* Englewood Cliffs, NJ: Prentice Hall, 1963.

[6] C. W. Barnes and S. Shinnaka, "Finite word effects in block-state realizations of fixed-point digital filters," *IEEE Trans. Circ. Syst.*, vol. CAS-27, pp. 345–349, May 1980.

[7] A. Mar, ed., *Digital Signal Processing Applications Using the ADSP-2100 Family.* Englewood Cliffs, NJ: Prentice Hall, 1990.