MANAGEMENT SCIENCE
Vol. 38, No. 2, February 1992
Printed in U.S.A.

VECTOR COMPUTERS, MONTE CARLO SIMULATION AND REGRESSION ANALYSIS: AN INTRODUCTION

JACK P. C. KLEIJNEN AND BEN ANNINK
Department of Information Systems and Auditing, School of Business and Economics, Katholieke Universiteit Brabant (Tilburg University), 5000 LE Tilburg, The Netherlands

Vector computers provide a new tool for management scientists. The application of that tool requires thinking in vector mode. This mode is examined in the context of Monte Carlo experiments with regression models; these regression models may serve as metamodels in simulation experiments. The vector mode needs to exploit a specific dimension of the Monte Carlo experiment, namely the replicates of that experiment. Taking advantage of the machine architecture gives a code that computes Ordinary Least Squares estimates on a Cyber 205 in only 2% of the time needed on a Vax 8700. For Generalized Least Squares estimates, however, the code runs slower on the Cyber 205 than on the VAX, if the regression model is small; for large models the CYBER 205 runs much faster.
(SUPERCOMPUTERS; DISTRIBUTION SAMPLING; MULTIVARIATE DISTRIBUTION; COMMON SEEDS; METAMODEL)

1. Introduction

In this paper we illustrate three important points about the new generation of computers called "supercomputers:"

(i) Efficient supercomputing requires that algorithms be adjusted to take advantage of the specific architecture of the computing hardware.

(ii) Expensive supercomputers are slower than general purpose machines are, for certain types of problems.

(iii) The increased speed of the supercomputing calculation may not outweigh the burden of constructing the specialized code: the researcher's time is valuable too.

We focus on the use of supercomputers in Monte Carlo experiments with regression analysis. So this paper may be of interest to management scientists for several reasons:

(i) Regression analysis is often used by management scientists to analyze simulation data and real-world data. The role of regression analysis in simulation will be explained in §2.

(ii) The study shows how supercomputers can be applied in Monte Carlo experiments. Monte Carlo experiments are related to stochastic discrete event dynamic simulation: both methods use pseudorandom numbers, but Monte Carlo experiments are static whereas simulation models are dynamic (a case in point is a queueing simulation); see Teichroew (1965). So Monte Carlo experiments are simpler. Our study may challenge other researchers to apply supercomputers to Monte Carlo and simulation models.

There are several types of supercomputers: vector computers, traditional scalar computers, and truly parallel computers. Traditional computers, such as the IBM 370 and the VAX series, execute one instruction after the other; so they operate sequentially. Truly parallel computers such as the HYPERCUBE have many Central Processing Units (CPUs) that can operate independently of each other; this is called coarse grain parallelism. Vector computers such as the CRAY 1 and the CYBER 205 have a "vector processing" capability: fine grain parallelism. Consider, for example, the computation of the inner product of two vectors: $v_1'v_2 = \sum_{j=1}^{n} v_{1j}v_{2j}$. This computation requires n identical scalar

* Accepted by James R. Wilson; received September 18, 1989. This paper has been with the authors 6 months for 2 revisions.

0025-1909/92/3802/0170$01.25

operations $v_{1j}v_{2j}$. The vector processor starts computing $v_{1j}v_{2j}$ while the computation of the predecessors $v_{1(j-1)}v_{2(j-1)}, v_{1(j-2)}v_{2(j-2)}, \cdots$ is still in process! So a vector computer works as an assembly line. A technical condition is that the scalar operations do not depend on each other; in the example the computation of the scalar product $v_{1j}v_{2j}$ does not need the other scalar products, especially the predecessors $v_{1(j-1)}v_{2(j-1)}$ through $v_{11}v_{21}$. The architecture of a vector computer is called a pipeline. The pipeline or assembly line requires a fixed set up cost; consequently a vector computer works efficiently only if a "large" number of identical (scalar) operations can be executed independently of each other. In the example, n must be large; a rule of thumb is $n \geq 50$ (we shall return to this issue). Our main issue is how to formulate the Monte Carlo model such that a vector computer can be applied efficiently. We do not discuss the use of truly parallel computers in simulation but refer to Heidelberger (1988), Adams (1990) and Reiher (1990). Technical details on the new generation of "supercomputers" are given by Levine (1982).

Our paper is organized as follows. In §2 we summarize the well-known linear regression model and its application in simulation experiments. We also discuss the role of experimental designs in simulation. The regression model is studied in a Monte Carlo experiment. In §3 we show how the Monte Carlo program can be vectorized: we discover a "third dimension" of Monte Carlo experiments. §4 gives numerical results, and §5 gives conclusions. References and appendices complete the paper.

2. Regression Models and Simulation

2.1. *Regression Models*

Consider the well-known linear regression model

$$E(\mathbf{y}) = \mathbf{X}\beta \tag{2.1}$$

with $\mathbf{y} = (y_1, \ldots, y_i, \ldots, y_n)'$, $\beta = (\beta_1, \ldots, \beta_j, \ldots, \beta_Q)'$ and $\mathbf{X} = (x_{ij})$ where $i = 1, \ldots, n$ and $j = 1, \ldots, Q$. We assume additive errors $\mathbf{e} = (e_1, \ldots, e_i, \ldots, e_n)'$ (the errors are also called disturbance or noise):

$$\mathbf{y} = \mathbf{X}\beta + \mathbf{e}. \tag{2.2}$$

We further assume that $\mathbf{e}$ is n-variate normally (N_n) distributed:

$$\mathbf{e} \sim N_n[\mathbf{0}_n, \mathbf{cov}(\mathbf{e})], \tag{2.3}$$

where $\mathbf{0}_n$ denotes a column of n zeros; $\mathbf{cov}(\mathbf{e})$ denotes the variance-covariance matrix of $\mathbf{e}$; $\mathbf{cov}(\mathbf{e})$ equals $\mathbf{cov}(\mathbf{y})$ because of (2.2); $\mathbf{cov}(\mathbf{y})$ is assumed to be nonsingular.

2.2. *Regression Metamodels in Simulation*

The regression model can be applied not only to analyze real-world data but also to analyze the results of a simulation experiment. We first consider a simplistic simulation experiment, namely a $G/G/1$ queue with a fixed arrival rate of (say) one and n different service rates μ_i ($i = 1, \ldots, n$). The n simulation responses are the average sojourn times (waiting plus service times) $\bar{w}_i$ ($= \sum_{t=1}^{T} w_t(\mu_i)/T$ assuming T customers are simulated for each μ_i). In the specification of a metamodel we are guided by the exact mathematical analysis of the steady-state mean sojourn time for a simpler $M/M/1$ queue; see Gross and Harris (1985). So we may model the response curve $E(\bar{w}) = f(\mu)$ by the second-order approximation in which the independent variables are powers of μ^{-1}, the reciprocal of the service rate: $E(\bar{w}) = \alpha_1(\mu^{-1}) + \alpha_2(\mu^{-1})^2$ with $\min_i \mu_i \leq \mu \leq \max_i \mu_i$, the experimental area. In the notation of (2.1) we then have $y = \bar{w}$, $\beta_1 = \alpha_1$, $x_1 = \mu^{-1}$, $\beta_2 = \alpha_2$, $x_2 = (\mu^{-1})^2$. Such an approximation is called a regression metamodel, since it is a model

of the input/output behavior of the underlying simulation model. Kleijnen (1987) gives details including realistic examples.

2.3. *Experimental Design Theory*

To illustrate the role of experimental design theory in regression modeling, we extend the $G/G/1$ example a bit. Suppose we study not only the effect of the service rate μ, but also the effect of the priority rule. Until now that rule was implicitly first-in-first-out (FIFO), but now we also examine an alternative rule, say short-jobs-first (SJF). Suppose further that we extend the queueing model such that S servers are simulated with $S = 1, 2, 3, 4$. Then there are three "factors" in the simulation experiment: service rate, priority rule, and number of servers. The statistical theory of *experimental design* helps to decide which combinations of factor "levels" or "values" to simulate. That theory assumes a regression metamodel! Suppose (just for illustration purposes) we assume that the response surface can be modeled by a first-order approximation: $y = \beta_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4$ with $y = \bar{w}$, $x_1 = 1$; $x_2 = -1$ if $\mu = \min_i \mu_i$ and $x_2 = +1$ if $\mu = \max_i \mu_i$; $x_3 = -1$ if $S = 1$ and $x_3 = +1$ if $S = 4$; $x_4 = -1$ if FIFO applies and $x_4 = +1$ if SJF holds. We can estimate these four regression parameters (β_1 through β_4) if we simulate only $4 = 2^{3-1}$ combinations of these three factors; see Table 1. We point out that the four column vectors $\mathbf{x}_j$ ($j = 1, 2, 3, 4$) are mutually orthogonal; $\mathbf{x}_1$ is a constant, not a factor, and is not shown in Table 1. For a second-order approximation, design theory gives analogous tables, albeit that more than two levels per factor must be included and that some columns are not orthogonal. There is a vast literature on experimental design; Kleijnen (1987) gives designs that are particularly useful in simulation experiments.

In the social sciences, the analysts cannot fix the independent variables $\mathbf{X}$ in (2.1); they can only observe those variables. (This lack of experimental control implies that it is virtually impossible to replicate specific situations; we shall briefly return to replication.) In simulation experiments the analysts can perfectly control all factors or simulation inputs; also see Kleijnen (1987, pp. 158–160). Then $\mathbf{X}$ follows from the experimental design matrix $\mathbf{D} = (d_{ih})$ with $h = 1, \ldots, k$ (and $i = 1, \ldots, n$); Table 1 gives an example of $\mathbf{D}$ with $k = 3$ and $n = 4$. Indeed $\mathbf{X}$ *follows* from $\mathbf{D}$; for example, a first-order approximation implies $X = (\mathbf{1}_n, \mathbf{D})$ where $\mathbf{1}_n$ denotes a column of n ones, and a second-order approximation implies $X = (\mathbf{1}_n, \mathbf{D}, \mathbf{D}_2)$ where $\mathbf{D}_2 = (d_{ih} d_{ig})$ with $g = h, \ldots, k$. Note that in some applications we force the regression equation through the origin, so the dummy column $\mathbf{1}_n$ vanishes (see the $G/G/1$ queueing example in §2.2). The number of regression parameters is denoted by Q; the example of Table 1 implies $Q = 4$. In a second-order approximation $Q = 1 + k + k(k-1)/2 + k$. In a well-designed simulation experiment it is easy to *replicate* each factor combination; that is, row i of $\mathbf{X}$ or $\mathbf{D}$ is observed $m_i \geq 2$ times. So a terminating simulation is repeated with m_i independent pseudorandom number streams, whereas in nonterminating or steady-state simulations m_i subruns may be obtained; see Kleijnen (1987, pp. 8–10, 63–83).

TABLE 1

2^{3-1} *Experimental Design for Three Factors*

Combination i	Factor Levels		
	x_2	x_3	x_4
1	−1	−1	+1
2	+1	−1	−1
3	−1	+1	−1
4	+1	+1	+1

We may simulate the n queueing systems with the same pseudorandom number sequence. This means that the n combinations of factor levels use the same seed. We repeat each combination a number of times, namely m times (so m_i of the preceding paragraph reduces to a constant m). Obviously $\mathbf{cov}(\mathbf{e})$ in (2.3) is nondiagonal if common seeds are used.

2.4. *Focus of Paper*

In the remainder of this paper we focus on the regression model specified by (2.1) through (2.3). Originally we wished to examine different estimators of the regression parameters β and different tests for validating the fit of the resulting regression model. For that study we use Monte Carlo simulation: we select $\mathbf{X}$, β, $\mathbf{cov}(\mathbf{e})$ and m; next we use those selected data to generate responses $\mathbf{Y} = (y_{ir})$; that data $\mathbf{X}$ and $\mathbf{Y}$ yield β estimators; these estimators are compared with the true parameter vector β (which is known in the Monte Carlo experiment); this comparison is repeated (say) $L = 100$ times to obtain reliable Monte Carlo results. That whole experiment is reported in Kleijnen (1991). Originally we planned to use a vector computer for that experiment, but it turned out that a vector computer may be inefficient in this case. In the present paper we explain why this is so. So we concentrate on those aspects of the original experiment that we need to explain the use of vector computers in Monte Carlo experiments with regression models applied to simulation data that are obtained by a sound experimental design.

Table 2 summarizes the data that are available to estimate the regression parameters β. The responses y_{ir} yield the following unbiased estimators of $\sigma_{if} = \text{cov}(y_i, y_f) = \text{cov}(y_{ir}, y_{fr})$ where y_{ir} is the rth replication of the ith factor combination:

$$\hat{\sigma}_{if} = \frac{\sum_{r=1}^{m}(y_{ir} - \bar{y}_i)(y_{fr} - \bar{y}_f)}{m-1} = \left(\sum_{r=1}^{m} y_{ir}y_{fr} - \bar{y}_i\bar{y}_f m\right)/(m-1) \qquad (2.4)$$

$$(i, f = 1, \ldots, n)\ (m \geq 2),$$

with the averages $\bar{y}_i = \sum_{r=1}^{m} y_{ir}/m$; by definition $\sigma_{ii} = \sigma_i^2$. Neely (1966) discusses the different numerical accuracies of the two expressions in (2.4). In matrix notation the last expression in (2.4) becomes

$$\widehat{\mathbf{cov}}(\mathbf{y}) = (\mathbf{Y}\mathbf{Y}' - \bar{\mathbf{y}}\bar{\mathbf{y}}'m)/(m-1), \qquad (2.5)$$

with $\widehat{\mathbf{cov}}(\mathbf{y}) = (\hat{\sigma}_{if})$ and $\bar{\mathbf{y}} = (\bar{y}_i)$. It is simple to prove that $\widehat{\mathbf{cov}}(\mathbf{y})$ is singular for $m \leq n$.

We consider two different point estimators for the regression parameters β. The simple classic estimator uses *Ordinary Least Squares* or OLS:

$$\hat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\bar{\mathbf{y}}, \qquad (2.6)$$

which assumes $n \geq Q$ and $\text{rank}(\mathbf{X}) = Q$. However, if $\mathbf{cov}(\mathbf{y})$ were known, then a better estimator would use *Generalized Least Squares* (GLS). Since $\mathbf{cov}(\mathbf{y})$ is unknown in

TABLE 2

Regression Data

Combination i (effects: $\beta_1 \cdots \beta_j \cdots \beta_Q$)	Responses y_{ir} (seed 1)$\cdots$(seed r)$\cdots$(seed m)	Average Response $\bar{y}_i$	Estimated (co)variances $\hat{\sigma}_{ih}$
$x_{11} \cdots x_{1j} \cdots x_{1Q}$	$y_{11} \cdots y_{1r} \cdots y_{1m}$	$\bar{y}_1$	$\hat{\sigma}_1^2 \hat{\sigma}_{12} \cdots \hat{\sigma}_{1n}$
$x_{i1} \cdots x_{ij} \cdots x_{iQ}$	$y_{i1} \cdots y_{ir} \cdots y_{im}$	$\bar{y}_i$	$\hat{\sigma}_i^2 \cdots \hat{\sigma}_{in}$
$x_{n1} \cdots x_{nj} \cdots x_{nQ}$	$y_{n1} \cdots y_{nr} \cdots y_{nm}$	$\bar{y}_n$	$\hat{\sigma}_n^2$

practice, we may replace it by the estimator $\widehat{\mathbf{cov}}(\mathbf{y})$ of (2.5), and use *Estimated Generalized Least Squares* or EGLS:

$$\tilde{\beta} = (\mathbf{X}'[\widehat{\mathbf{cov}}(\mathbf{y})]^{-1}\mathbf{X})^{-1}\mathbf{X}'[\widehat{\mathbf{cov}}(\mathbf{y})]^{-1}\bar{\mathbf{y}}, \tag{2.7}$$

which assumes that $\widehat{\mathbf{cov}}(\mathbf{y})$ is nonsingular; also see (2.3) and Dijkstra (1970).

3. Vectorizing the Monte Carlo Program

To generate the data $\mathbf{X}$ and $\mathbf{Y}$, to which the regression model is applied, we could have run the queueing simulation of the preceding section (§2). However, for the purpose of this paper such an approach is inferior, since we wish to compare the efficiency of vector computers relative to traditional computers *in the domain of regression analysis*, as we stated in §1 (second paragraph). Regression models can be used to analyze simulation data; and simulation models can be run on vector computers, but these issues are not the focus of this paper. Therefore we generate the data $\mathbf{X}$ and $\mathbf{Y}$ by executing a Monte Carlo experiment with the regression model (2.2) itself.

We might use the vector mode to compute an individual element y_{ir} of $\mathbf{Y}$ through (2.2). The matrix X in (2.2) is $n \times Q$; typically n and Q range between $n = 4$ and $Q = 4$ (see Table 1) and $n = 32$ and $Q = 22$ (see Table 3 later on). But vector computers are inefficient if the number of parallel operations is smaller than 50, as Levine (1982) and SARA (1984) state. So it is inefficient to vectorize the computation of an individual y_{ir}.

Next we consider the vector computation of either the rows or the columns of Table 2. Since there are only n rows (factor combinations), vectorization is again inefficient. The columns of Table 2 are statistically independent (see §2), so vectorization is possible. But since simulation replication is expensive, m will be small in practice (the minimum is $m = n + 1$; otherwise $\widehat{\mathbf{cov}}(\mathbf{y})$ is singular). So vectorizing the columns of Table 2 is also inefficient.

3.1. *The Third Dimension*

The Monte Carlo experiment is replicated $L = 100$ times (to obtain reliable Monte Carlo results; see §2.4). We speak of Monte Carlo replicates l with $l = 1, \ldots, L$, which must be distinguished from the simulation replicates $r = 1, \ldots, m$. The Monte Carlo replicates are statistically independent; they can be vectorized as we shall see. The more of these replicates we obtain, the more efficient the vector computer becomes. We may visualize our problem as filling a three-dimensional box in parallel with errors e_{irl} with $i = 1, \ldots, n$; $r = 1, \ldots, m$; $l = 1, \ldots, L$; this is explained in detail in Steps 1 through 3 below. In Step 4 statistics such as $\widehat{\mathbf{cov}}(\mathbf{y})$ are computed.

Step 1. *Sample pseudorandom numbers*. Kleijnen (1989) evaluates several procedures for the parallel generation of pseudorandom numbers $u \sim U(0, 1)$. Kleijnen and Annink (1990) recommend the following generator. Take a *scalar* multiplicative congruential generator with a multiplier that gives acceptable statistical behavior; such generators are discussed by Park and Miller (1988). To initialize the *vector* version of this generator, first generate—in scalar mode—a vector of J successive pseudorandom integers $\mathbf{K} = (K_0, K_1, K_2, \ldots, K_{J-2}, K_{J-1})'$ with seed K_0 and $K_j = (aK_{j-1}) \bmod m$ for $j = 1, 2, \ldots, J - 1$. To obtain numbers between zero and one, divide by m. Once and for all compute a scalar multiplier $(a^J) \bmod m$. Vector multiplication of the vector $\mathbf{K}$ with this scalar multiplier gives a new vector: $(K_J, K_{J+1}, \ldots, K_{2J-2}, K_{2J-1})'$. In this way the pseudorandom numbers are generated in parallel and yet in exactly the same order as they would have been produced in scalar mode. At the end of the Monte Carlo experiment the vector of the last J numbers should be stored, so that the experiment may be continued later on.

We mentioned that vector computers become more efficient as the number of parallel operations increases. For the CYBER 205, however, there is a technical upper limit: J

$= 2^{16} - 1 = 65{,}535$ (since this computer uses 16 bits for addressing; see SARA 1984, p. 26).

There is a computational problem: overflow occurs when computing $(a^J) \bmod m$. We solve this problem through the computer science techniques of controlled integer overflow and the CYBER 205's two's complement representation of negative integers. The computer program in Appendix 1 gives technical details; Park and Miller (1988) also discuss overflow.

So we generate a vector of J pseudorandom numbers. We store that vector, which is then available to fill the three-dimensional box.

Step 2. *Sample independent standard normal variates*. There are several techniques for generating normal variates; see, for example, Devroye (1986). We take a procedure that fits a vector computer:

$$z_1 = (-2 \ln u_1)^{1/2} \cos 2\pi u_2, \tag{3.1.a}$$

$$z_2 = (-2 \ln u_1)^{1/2} \sin 2\pi u_2, \tag{3.1.b}$$

where the mutually independent pair u_1 and u_2 with $u \sim U(0, 1)$ yields the mutually independent pair z_1 and z_2 with $z \sim N(0, 1)$. To compute the functions ln, cos and sin for a *vector* of numbers, we use the FORTRAN 200 vector functions VLN, VCOS, and VSIN. Given a vector of L independent pseudorandom numbers u, we use the first half to compute $L/2$ independent parallel realizations of $\ln u_1$, and the second half to compute $\cos(2\pi u_2)$ and $\sin(2\pi u_2)$. Figure 1 gives a pseudo-FORTRAN program where π is computed through the arccosine function, as SARA (1984, p. 13) suggests. To convert this pseudo-FORTRAN into a FORTRAN 200 program, we can replace the DO loops by the special syntax of FORTRAN 200. The supercomputer, however, can also automatically translate the FORTRAN program of Figure 1, provided we add CONTINUE statements; see CDC (1986), SARA (1984, p. 17).

Note that Petersen (1988) generates z in parallel, not through (3.1.a) and (3.1.b), but through Teichroew's procedure, which is described in Naylor et al. (1966, p. 94).

To fill the three-dimensional box with e_{irl}, we store the *vectors* $\mathbf{z}$ (with L elements) of Figure 1 into a three-dimensional *array* Z.

Step 3. *Sample n-variate normally distributed variates*. The errors within a column of Table 2 are statistically dependent: they are n-variate normal. We first consider a computer program for $n = 2$. In that case we sample the independent univariate standard normal variates z_1 and z_2, and compute the linear transformations $e_1 = \sigma_1 z_1$ and $e_2 = \sigma_2(\rho z_1 + (1 - \rho^2)^{1/2} z_2)$ where $\rho = \sigma_{12}/(\sigma_1 \sigma_2)$. Next we consider the general case. The sampling subroutine for multivariate normal $\mathbf{e}$ with covariance matrix $\mathbf{cov}\,(\mathbf{e})$ is

$$\mathbf{e} = \mathbf{Cz}, \tag{3.2}$$

with $\mathbf{z} = (z_1, \ldots, z_i, \ldots, z_n)'$ and independent $z_i \sim N(0, 1)$, and $\mathbf{C}$ a lower triangular matrix defined by

$$\mathbf{CC}' = \mathbf{cov}\,(\mathbf{e}). \tag{3.3}$$

```
      L2 = L/2; PI = ACOS(-1.0); C = 2 * PI
      DO 20 LL = 1, L2
20        HELP1(LL) = SQRT(-2 * LOG(U(LL)))
      DO 30 LL = 1, L2
          HELP2(LL) = COS(U(LL + L2) * C)
          HELP3(LL) = SIN(U(LL + L2) * C)
          Z(LL) = HELP1(LL) * HELP2(LL)
30        Z(L2 + LL) = HELP1(LL) * HELP3(LL)
```

FIGURE 1. Parallel Computation of L Variates $z \sim N(0, 1)$.

```
      DO    10    LL = 1,L
        DO    10    R = 1,M
          DO    10    I = 1,N
            DO    10    J = 1,I
10              E(I,R,LL) = E(I,R,LL) + C(I,J) * Z(J,R,LL)
```

FIGURE 2. Naive FORTRAN Program for **e**.

The matrix **C** is computed by Choleski's technique; see Naylor et al. (1966, pp. 97–99) and standard software libraries such as IMSL and NAG. Once **C** is computed, **e** is generated through the linear transformation (3.2) of **z**. We do not vectorize that transformation since n is too small.

To obtain M observations and L Monte Carlo replicates of **e**, we might apply the naive FORTRAN program of Figure 2, where M denotes the maximum value of m in the experiment (Table 3 means $M = 33$) and $E(I, R, LL)$ is zero initially. Note that **C** or $C(I, J)$ does not vary over seeds (R) and Monte Carlo replicates (LL); it does vary with **cov** (**y**).

SARA (1984, pp. 15, 20–21, 33) states that when vectorizing a program, the *inner* DO loop should be long, and the elements of the array should be stored columnwise so that the innermost DO loop controls the first index of the array. Therefore we move the LL loop and replace $E(I, R, LL)$ by $E(LL, R, I)$ to get Figure 3. (The inner loop forms a so-called "linked triad," which can be vectorized; see SARA 1984, pp. 18–19.)

The quantities m and n vary with the different cases investigated in the Monte Carlo experiment, as Table 3 will show. So a case may use only part of the pseudorandom numbers stored in the "box" $E(LL, R, I)$. Figure 3 not only saves computer time, but also implements common seeds since all cases pull pseudorandom numbers from the same box.

Note that we could generate $M*L$ instead of L elements in parallel, if we replaced the loops for R and LL respectively in Figure 3 by the single loop $LR = 1, \ldots, M*L$, which would yield the two-dimensional array $E(LR, I)$ of Figure 4. Then, however, we would have to rearrange this array into the three-dimensional array $E(LL, R, I)$, because the latter array is needed to compute statistics such as $\widehat{\mathbf{cov}}$ (**y**), as we shall see in the next step.

Step 4. *Compute statistics* $\widehat{\mathbf{cov}}$ (**y**), $\hat{\beta}$ and $\tilde{\beta}$. Given the three-dimensional array **e**, we can easily compute estimates such as $\widehat{\mathbf{cov}}$ (**y**). We reformulate (2.5) as

$$\widehat{\mathbf{cov}}\,(\mathbf{y}) = \mathbf{e}\mathbf{e}'/(m-1) - \bar{\mathbf{e}}\bar{\mathbf{e}}'m/(m-1), \tag{3.4}$$

with $\bar{\mathbf{e}} = (\bar{e}_1, \ldots, \bar{e}_i, \ldots, \bar{e}_n)'$ and $\bar{e}_i = \sum_{r=1}^{m} e_{ir}/m$. Figure 5 shows the vectorizable FORTRAN program for the computation of $\bar{\mathbf{e}}$. This program can be compiled and vectorized automatically. Alternatively we can use special FORTRAN 200 instructions such as Q8SSUM, which computes sums like $\sum e_{ir}$. The computation of $\widehat{\mathbf{cov}}$ (**y**) in (3.4) can be programmed analogous to Figure 5. Alternatively we can program inner products ($\mathbf{e}\mathbf{e}'$ and $\bar{\mathbf{e}}\bar{\mathbf{e}}'$) through the special function Q8SDOT, as SARA (1980, pp. 22, 30) mentions.

```
      DO    20    I = 1,N
        DO    20    J = 1,I
          DO    20    R = 1,M
            DO    20    LL = 1,L
              E(LL,R,I) = E(LL,R,I) + C(I,J) * Z(LL,R,J)
```

FIGURE 3. Vectorized FORTRAN Program for **e**.

```
      ML = M*L
      DO    20    I = 1,N
        DO    20    J = 1,I
          DO    20    LR = 1,ML
20            E(LR,I) = E(LR,I) + C(I,J) * Z(LR,J).
```

FIGURE 4. Alternative Vectorized FORTRAN Program for $\mathbf{e}$.

3.2. *A Roadblock to Vectorization*

A problem arises when computing the *inverse* $[\widehat{\mathbf{cov}}(\mathbf{y})]^{-1}$, which is needed to compute the EGLS estimator $\tilde{\beta}$ in (2.7). The trick in the preceding steps was to make the inner loop long; that is, we made the LL loop the inner loop. SARA (1984, p. 23) states that the instruction within the inner loop can be executed in parallel, provided that instruction contains *no function or subroutine references* except for basic functions such as sine: the vector computer can execute in parallel basic operations only. So the computer cannot calculate L inverses in parallel, since inversion requires a subroutine call.

To invert a matrix we call a routine provided by Numerical Algorithms Group or NAG (the routine is F01AAF, which solves linear equations using Crout's method). Obviously a subroutine call can always be avoided, since the subroutine can be replaced by the appropriate lines of code. Moreover, there is often more than one computational technique; for example, matrices can be inverted in several ways, and covariances can be computed in different ways, as (2.4) and (3.4) showed. However, subroutines are there to help the user; so most times the user will call upon a subroutine. This problem illustrates a more general problem: how much effort does the user want to spend on programming in order to fit the problem to a specific computer so that this computer runs faster?

So $[\widehat{\mathbf{cov}}(\mathbf{y})]^{-1}$ must be computed in scalar mode. Once this inverse is available, some matrix multiplications follow; for example, $[\widehat{\mathbf{cov}}(\mathbf{y})]^{-1}\mathbf{X}$. The share of the matrix inversion in the total computation time determines the gain to be obtained through vectorization.

4. Computational Tests

To quantify the ideas formulated in §3, we compute the OLS and the EGLS estimates for a number of cases, comparing a CYBER 205 and a VAX 8700; see Table 3. We select three cases, as follows. We use a regression metamodel for k factors accounting for all $k(k-1)/2$ two-factor interactions besides the overall mean and the k main effects; so $Q = 1 + k + k(k-1)/2$. The experimental design is a 2^{k-p} design with $n = 2^{k-p} \geq Q$. Consequently, if $k = 2$ then $Q = 4$ and $n = 2^2 = 4$. If $k = 4$ then $Q = 11$ and $n = 2^4 = 16$. If $k = 6$ then $Q = 22$ and $n = 2^{6-1} = 32$. We keep the number of simulation replicates at its minimum: $m = n + 1$ (if $m \leq n$ then $\widehat{\mathbf{cov}}(\mathbf{y})$ is singular). To improve

```
      DENOM = 1.0/M
      DO    10    I = 1,N
        DO    10    R = 1,M
          DO    10    LL = 1,L
10            EBAR(LL,I) = EBAR(LL,I) + E(LL,R,I)
      DO    20    I = 1,N
        DO    20    LL = 1,L
20            EBAR(LL,I) = EBAR(LL,I) * DENOM
```

FIGURE 5. Vectorizable FORTRAN Program for $\bar{\mathbf{e}}$.

TABLE 3

Total CPU times (in microseconds) ($m = n + 1$, $L = 100$)

OLS	Case 1 $n = 4$ $Q = 4$	Case 2 $n = 16$ $Q = 11$	Case 3 $n = 32$ $Q = 22$
VAX 8700	710	7,870	29,060
CYBER: scalar mode	544	6,188	24,035
vector mode	11	123	486
EGLS			
VAX 8700 total	29,722	495,450	3,261,370
inversion	22,550	243,150	1,230,120
rest	7,172	252,300	2,031,250
CYBER: scalar total	37,797	361,322	2,058,737
inversion	32,267	168,244	584,393
rest	5,530	193,078	1,474,344
CYBER: vector total	32,437	172,854	625,000
inversion	32,297	168,084	583,639
rest	140	4,770	41,361

the accuracy of our timing data we repeat the computation 100 times. Into the OLS estimator of (2.6) we substitute

$$\mathbf{W} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}' \tag{4.1}$$

and into the EGLS of (2.7) we substitute

$$\mathbf{V} = (\mathbf{X}'[\widehat{\mathbf{cov}}\,(\mathbf{y})]^{-1}\mathbf{X})^{-1}\mathbf{X}'[\widehat{\mathbf{cov}}\,(\mathbf{y})]^{-1}. \tag{4.2}$$

$\mathbf{W}$ needs to be computed only once, but $\mathbf{V}$ is calculated $L = 100$ times since $\widehat{\mathbf{cov}}\,(\mathbf{y})$ changes every time. Appendix 2 gives the main part of the computer program.

The CYBER 205 can run in vector mode and in scalar mode respectively. The results in Table 3 show that for OLS the scalar mode of this expensive computer runs only slightly faster than the VAX does. In vector mode, however, the CYBER takes less than 2% of the VAX time. In our particular EGLS code, matrix inversion cannot be vectorized. Therefore we measure how much time inversion takes. Obviously scalar mode and vector mode of the CYBER yield the same CPU times for inversion, apart from measurement errors. The "rest" in Table 3 refers to the whole computer code excluding matrix inversion. In "vector" mode we vectorized all instructions that can be vectorized over the L dimension (see again Appendix 2). In the small problem ($n = 4$, $Q = 4$) nonvectorizable inversion takes 85% of total time, so vectorizing the rest can never save more than 15%; actually it saves 14%. In the large problem ($n = 32$, $Q = 22$) inversion takes only 28% of total time; vectorizing the rest saves 70%. We point out that in the small problem, EGLS runs faster on the VAX than on the CYBER, even in vector mode. In Appendix 3 we give some more programming tricks for improving the efficiency of vector computers.

5. Conclusions

Vector computers provide a new challenge for management scientists, since their application requires a new way of thinking, namely "thinking in vector mode." We examined vector computing in Monte Carlo experiments with regression models, which are also used as metamodels in simulation. If the matrix of independent variables $\mathbf{X}$ is relatively small, then vector computers are inefficient if applied straightforwardly. Monte Carlo

experiments, however, are replicated many times, say 100 times. Exploiting this dimension of the problem makes vector computers efficient in applications such as Ordinary Least Squares. Other applications require subroutine calls; for example, Estimated Generalized Least Squares requires matrix inversion. In small problems, vector computers such as the CYBER 205 are then slower than are scalar computers such as the VAX 8700. So the researcher must estimate which fraction of the total computer time can be saved by vectorization. Moreover, exploiting vector computers requires researcher's time to figure out efficient implementations. The potential CPU time savings should be weighted against the coding effort required to assess the net benefit of this vectorization strategy.[1]

[1] The first author was sponsored by the Supercomputer Visiting Scientist Program at Rutgers University, The State University of New Jersey, during July 1988. In 1989/1990, computer time on the CYBER 205 in Amsterdam was made available by SURF/NFS. The editor (Jim Wilson) and three anonymous referees gave many comments that lead to an expanded and better organized paper, and to the elimination of some errors; any remaining errors are the authors' responsibility.

Appendix 1. FORTRAN 200 Program for the Pseudorandom Number Generator

```
      PROGRAM PSEUDORANDOM
      IMPLICIT REAL (U-Z), INTEGER (A-T)
C     N is length of vector of pseudorandom numbers
C     K precedes seed K(0) of initial vector
      PARAMETER (N=65535,K=1)
C     A1 is multiplier
      PARAMETER (A1=37772072706109)
C     MVAST, BVAST and MINT are used in controlled integer overflow
      INTEGER MVAST
      BIT BVAST
      DESCRIPTOR MVAST,BVAST
      DIMENSION T(N) , S1(N)
      DIMENSION X1(N)
      DATA MINT / X'0000800000000000' /
      CALL RANSET(K)
      DO 5 I=1,N
        U=RANF( )
        CALL RANGET(T(I) )
    5 CONTINUE
      ASSIGN MVAST, .DYN.N
      ASSIGN BVAST, .DYN.N
      S1(1;N)=T(1;N)
      S1(1;N)=A1*S1(1;N)
C     controlled integer overflow
      BVAST=S1(1;N) .LT.0
      MVAST=S1(1;N)-MINT
      S1(1;N)=Q8VCTRL(MVAST,BVAST;S1(1;N) )
      X1(1;N)=S1(1;N)/MINT
      FREE
      END
```

Appendix 2. FORTRAN 200 Program for the OLS and EGLS Estimators

```
OLS ESTIMATOR FOR BETA

C     MXM is a user defined routine that multiplies 2 matrices
      CALL MXM(XGT,X,XTX)
      CALL INVERSE(XTX,XTXI)
      CAL MXM(XTXI,XT,W)
C     N denotes # of rows of X; M denotes # of replicates
      DO  5  I=1,N
        DO  5  J=1,M
          YGEM(1,I;LL)=YGEM(1,I;LL)+Y(1,J,I;LL)
```

```
5     CONTINUE
C     Q denotes # of beta's
      DO  10  I=1,Q
        DO  10  J=1,N
          BETA(1,I;LL)=BETA(1,I;LL)+W(I,J) *YGEM(1,J;LL)
10    CONTINUE


EGLS ESTIMATOR FOR BETA

      DO  5  I=1,N
        DO  5  J=1,M
          YGEM(1,I;LL)=YGEM(1,I;LL)+Y(1,J,I;LL)
5     CONTINUE
C     S is estimated covariance matrix
      DO  10  I=1,N
        DO  10  J=I,N
          DO  10  K=1,M
            S(1,I,J;LL)=S(1,I,J;LL)+( (Y(1,K,I;LL) -
     $          YGEM(1,I;LL) )*(Y(1,K,J;LL)-YGEM(1,J;LL) ) )
10    CONTINUE
C     S is symmetric
      DO  15  I=1,N
        DO  15  J=1,N
          S(1,J,I;LL)=S(1,I,J;LL)
15    CONTINUE
C     Invert 2-dim. matrix DU4, which is part of 3-dim. matrix S
      DO  18  K=1,LL
        DO  20  I=1,N
          DO  20  J=1,N
            DU4(J,I)=S(K,J,I)
20    CONTINUE
      CALL INVERSE(DU4,MY4,N)
      DO  25  I=1,N
        DO  25  J=1,N
          SI(K,J,I)=MY4(J,I)
25      CONTINUE
18    CONTINUE
      DO  30  I=1,R
        DO  30  J=1,N
          DO  30  K=1,N
            XTSI(1,I,J;LL)=XTSI(1,I,J;LL)+XT(I,K) *SI(1,K,J;LL)
30    CONTINUE
      DO  35  I=1,R
        DO  35  J=1,R
          DO  35  K=1,N
        XTSIX(1,I,J;LL)=XTSIX(1,I,J;LL)+XTSI(1,I,K;LL) *X(K,J)
35    CONTINUE
      DO  40  K=1,LL
        DO  45  I=1,R
          DO  45  J=1,R
            DU3(J,I)=XTSIX(K,J,I)
45    CONTINUE
      CALL INVERSE(DU3,MY3,R)
      DO  50  I=1,R
        DO  50  J=1,R
          XTSIXI(K,J,I)=MY3(J,I)
50      CONTINUE
40    CONTINUE
      DO  55  I=1,R
        DO  55  J=1,N
          DO  55  K=1,R
            V(1,I,J;LL)=V(1,I,J;LL)+XTSIXI(1,I,K;LL) *XTSI(1,K,J;LL)
```

```
55    CONTINUE
      DO  60  I=1,R
        DO  60  K=1,N
          BETA(1,I;LL)=BETA(1,I;LL)+V(1,I,K;LL) *YGEM(1,K;LL)
60    CONTINUE
```

Appendix 3: Programming Tricks

There are several "tricks" for improving the efficiency of vector computers like the Cyber 205. These tricks should be applied in any computer program, not only Monte Carlo experiments.

1. Scalar divides take relatively much computer time: 54 cycles versus 5 cycles for multiplication; 1 cycle takes 20 nanoseconds. The computation of denominators like $1/m$ (see Figure 5) and $1/(m-1)$ (see equation (3.4)) should therefore be separated by several lines of code; see SARA (1984, pp. 5, 7).

2. Double precision is slow and excludes vector mode; SARA (1984, p. 6).

3. There are special vectorized instructions, namely V-functions and $Q8$-functions. We presented some examples; also see SARA (1984, pp. 27, 30).

4. The compiler can optimize the standard FORTRAN program; next special programs (such as SPY and CIA) can measure which parts of the program take most time during execution and are candidates for customized optimization.

References

ADAMS, D. A., "Parallel Processing Implications for Management Scientists," *Interfaces*, 20 (1990), 88–98.

CDC, *FORTRAN* 200 *Version* 1 *Reference Manual*, Publication no. 60480200, Control Data Corporation, Sunnyvale, CA 94088-3492, December 1986.

DEVROYE, L., *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.

DIJKSTRA, R. L., "Establishing the Positive Definiteness of the Sample Covariance Matrix," *Ann. Math. Statist.*, 41 (1970), 2153–2154.

GROSS, D. AND C. M. HARRIS, *Fundamentals of Queueing Theory*, (Second Ed.), John Wiley & Sons, New York, 1985.

HEIDELBERGER, P., "Discrete Event Simulations and Parallel Processing: Statistical Properties," *SIAM J. Statist. Comput.*, 39 (1988), 1114–1132.

KLEIJNEN, J. P. C., *Statistical Tools for Simulation Practitioners*, Marcel Dekker, Inc., New York, 1987.

———, "Analyzing Simulation Experiments with Common Random Numbers," *Management Sci.*, 34 (1988), 65–74.

———, "Pseudorandom Number Generation on Supercomputers," *Supercomputer*, 6 (1989), 34–40.

———, *Regression Metamodels for Simulation with Common Random Numbers: Comparison of Validation Tests and Confidence Intervals*, Katholieke Universiteit Brabant (Tilburg University), January 1991. (Accepted by *Management Sci.*)

——— AND B. ANNINK, *Pseudorandom Number Generators for Supercomputers and Classical Computers: a Practical Introduction*. Katholieke Universiteit Brabant (Tilburg University), December 1990. (Accepted by *European J. Oper. Res.*)

LEVINE, R. D., "Supercomputers," *Scientific Amer.*, (1982), 112–125.

NAYLOR, T. H., J. L. BALINTFY, D. S. BURDICK AND K. CHU, *Computer Simulation Techniques*, Wiley, New York, 1966.

NEELY, P. M., "Comparison of Several Algorithms for Computation of Means, Standard Deviations, and Correlation Coefficients," *Comm. ACM*, 9 (1966), 496–499.

PARK, S. K. AND K. W. MILLER, "Random Number Generators: Good Ones Are Hard to Find," *Comm. ACM*, 31 (1988), 1192–1201.

PETERSEN, W. P., "Some Vectorized Random Number Generators for Uniform, Normal, and Poisson Distributions for CRAY X-MP," *J. Supercomputing*, 1 (1988), 327–335.

REIHER, P. L., "Parallel Simulation Using the Time Warp Operating System," *Proceedings of the* 1990 *Winter Simulation Conference*, O. Balci, R. P. Sadowski and R. E. Nance (Eds.), (1990) 38–45.

SARA, *Cyber* 205 *User's Guide*; *Part* 3, *Optimization of FORTRAN Programs*. SARA (Stichting Academisch Rekencentrum Amsterdam/Foundation Academic Computer Centre Amsterdam), Amsterdam, 1984.

TEICHROEW, D., "A History of Distribution Sampling Prior to the Era of the Computer and its Relevance to Simulation," *J. Amer. Statist. Assoc.*, (1965), 27–49.