An efficient approach to implement dynamic batch means estimators in simulation output analysis

Mingchang Chih\textsuperscript{a} and Wheyming Tina Song\textsuperscript{b*}

\textsuperscript{a}System Development Center, Chung-Shan Institute of Science and Technology, Lungtan Township, Taoyuan, Taiwan 32546, Republic of China; \textsuperscript{b}Department of Industrial Engineering and Engineering Management, National Tsing Hua University, Hsinchu, Taiwan 30013, Republic of China

(Received April 2011; revised October 2011; accepted February 2012)

Estimating the variance of the sample mean is a classical problem of stochastic simulation. Traditional batch means estimators require specification of the simulation run length \textit{a priori}. Dynamic batch means (DBM) is a new approach to implement the traditional batch means in fixed memory by dynamically changing both batch size and number of batches without the knowledge of the simulation run length. This article further improves the DBM by considering small storage requirements and fast computation. The proposed algorithm is useful when the simulation run length is random and extremely long in simulation models.

**Keywords:** variance of the sample mean; batch means estimators; simulation

1. Introduction

Estimating the variance of the sample mean from a stochastic process is essential in assessing the quality of using the sample mean to estimate the population mean. In addition, it is also crucial in calculating the confidence and prediction intervals of the population mean. In addition, it is also crucial in estimating the variance of the sample mean, which is a classical problem of stochastic simulation. Traditional batch means estimators require specification of the simulation run length \textit{a priori}. Dynamic batch means (DBM) is a new approach to implement the traditional batch means in fixed memory by dynamically changing both batch size and number of batches without the knowledge of the simulation run length. This article further improves the DBM by considering small storage requirements and fast computation. The proposed algorithm is useful when the simulation run length is random and extremely long in simulation models.

**Keywords:** variance of the sample mean; batch means estimators; simulation

Batching [1] is a classic methodology for estimating the variance of the sample mean. The idea is to divide the \(n\) observations into \(b\) batches each of which is of size \(m\). Several approaches based on grouping the output data are proposed. For example, non-overlapping batch means (NBM) [8], overlapping batch means (OBM) [7], partial-OBM (PBM, also named as \((100\alpha\% )\text{-OBM}\) for \(0 < \alpha < 1\)) [16], standardized time series [4,8], and Cramér–von Mises [5].

From practitioner’s point of view, a good solution for the estimation of the variance of the sample mean should meet some criteria [6,18], such as good statistical properties, fast computation, small storage requirements, ease of understanding, numerical stability, less user-specified parameters, and amenability for use in algorithms. Therefore, one of the primary tasks for a researcher is to develop procedures or estimators that satisfy the above solution criteria as much as possible.

Traditional batch means estimators (previous mentioned) in simulation output assume that the simulation run length or the sample size \(n\) is known in advance. In other words, the storage requirements for the above estimators are proportional to the sample size \(n\); these estimators require \(O(n)\) space. Fishman and Yarberry [3] proposed the LABATCH NBM method requiring \(O(\log_2 n)\) space. Although \(O(\log_2 n)\) algorithms require much small memory space than \(O(n)\)-memory algorithms, the LABATCH requires knowledge of \(n\) \textit{a priori}. Steiger and Wilson [15] proposed the

---

\*Corresponding author. Email: wheyming@ie.nthu.edu.tw
Automated Simulation Analysis Procedure (ASAP) for building a confidence interval on a steady-state expected simulation response, while the procedure also requires knowledge of \( n \) \textit{a priori}. For fixed sampling methods, these estimators and procedures work very well. However, they are not superior anymore when the sample size is not known in advance as in sequential sampling methods.

This article focuses on the situations where the simulation run length or the sample size \( n \) is not known in advance and therefore traditional batch means estimators cannot be applied directly. Consider two scenarios [12]: (1) The sample size is unknown \textit{a priori}. Consider creating a simulation model of pandemic influenza to evaluate the effectiveness of different decision policies on disease spread and other performance measures [7]. The simulation run length (in terms of the number of patients) is random, so \( n \) is not known in advance. (2) The sample size is extremely large. One example is that of using simulation to test data stream algorithms to estimate entropy (a measure of the rate of transfer of information in a network) from input data that come at a very high rate – a rate so high that it places stress on a limited computing infrastructure. Recent work on data stream algorithms can be found in Zhao et al. [19]. Throughout this article, we use the phrase “finite-memory algorithm” to denote algorithms that require \( O(1) \) memory storage and do not require algorithms that knowledge of \( n \) \textit{a priori}.

Yeh and Schmeiser [18] proposed the first finite-memory algorithm to introduce the method of dynamic batch means (DBM) in simulation output analysis. The method is to implement the batch means in fixed memory by dynamically changing both batch size and number of batches as the simulation runs. DBM require only \( O(1) \) storage space and do not require knowledge of \( n \) \textit{a priori}. The dynamic non-overlapping batch means (DNBM) proposed by Yeh and Schmeiser [18] and the dynamic partial overlapping batch means (DPBM) proposed by Song [10] are two procedures to implement NBM and PBM (75\%OBM) in fixed storage space, respectively. Song [11] generalized the existing procedures to the dynamic 100\% partial overlapping batch means (100\% DOBM), which allows users to implement traditional (100\%)\%OBM for \( \alpha = 0, 1/2, 3/4, 7/8, \ldots \) without knowing the sample size in advance. Our work in this research further improves the 100\% DOBM with the consideration of computational efficiency and memory requirement. We seek approaches that implement dynamic idea with faster computation and smaller storage requirement.

This article is organized as follows. Section 2 reviews traditional batching methods and DBM procedures. Section 3 proposes an efficient approach to implement DBM algorithm. Section 4 is the evaluation via Monte Carlo simulation. Section 5 summarizes the findings.

2. Literature review

2.1 Batch means estimators

Batching is a classic methodology used in estimating the variance of the sample mean from a stationary stochastic process. Conway [1] was the first to introduce the idea of the batching method in digital simulation. The method is based on dividing the observations \( Y_1, Y_2, \ldots, Y_n \) into \( b \) batches, with each batch size being \( m \) (i.e. \( n = bm \)). In other words, the method groups observations into batches and uses these batches as the basic data units for analysis. Song and Schmeiser [13, p.504] summarized and defined the batch means estimators, including NBM [8], OBM [7], and PBM [16], as a function of \( m \) and \( s \) as follows:

\[
\hat{V}(m, s) = \frac{\sum_{i=1}^{b} (\overline{Y}_{s(i-1)+1:m} - \overline{Y}_n)^2}{d},
\]

where \( 1 \leq s \leq n - m \) is the distance between the first observation of any two adjacent batches, \( d = b(nm^{-1} - 1) \) is the denominator value, \( b = \lceil (n - m + s)/s \rceil \) is the number of batches (where \( \lceil x \rceil \) is the greatest integer smaller than or equal to \( x \)),

\[
\overline{Y}_{s(i-1)+1:m} = \sum_{j=1}^{m} Y_{s(i-1)+j}/m
\]

is the \( i \)-th batch mean, and \( Y_{s(i-1)+j} \) is the \( j \)-th observation in the \( i \)-th batch. The meaning of \( m \) and \( s \) are clearly shown in Figure 1.

The NBM estimator with batch size \( m \) is the special case obtained when \( s = m \) (Figure 1a), and is denoted by \( \hat{V}(m) \). The OBM estimator with batch size \( m \) is the special case obtained when \( s = 1 \) (Figure 1b), and is denoted by \( \hat{V}_O(m) \). The PBM estimator with batch size \( m \) is the special case obtained when \( 1 < s < m \) (Figure 1c). The (100\%)\%OBM, \( 0 < \alpha < 1 \), another form of PBM, indicates that there is 100\% overlap among all data between two adjacent batches, where

\[
\alpha = 1 - (s/m)
\]

and \( 1 < s < m \) (i.e. \( 0 \leq \alpha < 1 \)). For example, 50\%OBM and 75\%OBM are \( \hat{V}(m, s) \) estimators for \( s = m/2 \) and \( s = m/4 \), respectively. The spaced batch means estimator (SBM) with batch size \( m \) is the special case obtained when \( s > m \), and is denoted by \( \hat{V}(s) \) (Figure 1d).
The asymptotic properties of batch means estimators are discussed in several studies. The asymptotic relative bias results, discussed in Meketon and Schmeiser [7] and Song and Schmeiser [13], show that all batch means estimators have essentially the same bias, and the asymptotic variance of the (100α)%OBM decreases as α increases. Specifically, the asymptotic variance ratio for NBM, 50%OBM, 75%OBM, and OBM is 1.5: 1.12: 1.003: 1, as shown in Welch [16] and Meketon and Schmeiser [7].

### 2.2 DBM estimators

The approach to develop DBM is by using Fishman’s idea of doubling batch sizes [2]. DBM estimators use only finite storage space, increase batch size dynamically as simulation run length increases, and compute the batch means estimates according to the value of current batch size. In DBM, all data are stored in a vector $L$ with size $2gw$ (i.e., $L = [A_1, A_2, \ldots, A_w]$). $A_j = [A(1), A(2), \ldots, A(2g)]$, $j = 1, 2, \ldots, w$, where $w$ and $g$ are two pre-specified parameters: $w$ is the number of sub-vectors in $L$; $2g$ is the number of cells (memory size) for each sub-vector $A_j$. The key idea behind the DBM involves collapsing the $w$ vectors. Instead of keeping each individual observation, DBM stores the sum of observations for each batch. DBM adds new observation in the current cell(s) if the number of observations stored in the current cell(s) is less than the present batch size $m = 2^k$ at stage $k$. Whenever the first sub-vector $A_1$ in $L$ is full, DBM starts to collapse data. Initially $k$ is set to 0, the value of $k$ is increased by 1 after a collapsing. The collapsing starts with the sub-vector $A_2$, where $z = \min\{k, \ln_2 w\}$. Following the collapsing in $A_2$, the next collapsing occurs in sub-vector $A_{z-1}$, and so on. The sub-vectors in $L$ are updated in that

$$A_j^{(k)}(i) = \begin{cases} A_{(j/2)}^{(k-1)}(2i) + A_{(j/2)}^{(k-1)}(2i+1), & i = 1, \ldots, g-1; \ j \text{ is even}; \\ A_{(j/2)}^{(k-1)}(2g), & i = g; \ j \text{ is even}; \\ A_{(j/2')}^{(k-1)}(2i-1) + A_{(j/2')}^{(k-1)}(2i), & i = 1, \ldots, g; \ j \text{ is odd}. \end{cases}$$

where $A_j^{(k)}(i), i = 1, 2, \ldots, 2g; j = 1, 2, \ldots, w$ is the $i$-th cell of sub-vector $A_j$ at step $k$ where the numerical values (batch sums) stored. That is, after collapsing, the observations in the certain pairs of adjacent batches are combined into one batch sum, and hence half of the vector becomes available to contain new observations. Figure 2 is an illustration for the specific case of DBM with $w = 8$.

Song [11] proposed a general form for DBM, $(100\alpha)\%$ DOBM, $0 < \alpha < 1$, indicates that there is 100e% overlap among all data between two adjacent batches, where

$$\alpha = 1 - 2^{-z},$$

and $z$ could be treated as the number of sub-vectors used at step $k$. For example, assume that we set the pre-specified parameter $w = 2^4 = 16$ for the DBM. We will have a $50\%$ DOBM, if the algorithm stop at the stage $k = 1$. We will have a $75\%$ DOBM, if the algorithm stops at the stage $k = 2$. We will have a $93.75\%$ DOBM, if the algorithm stops at the stage $k \geq 4$. Song [11] showed that the $(100\alpha)\%$ DOBM is algebraically equivalent to the traditional $(100\alpha)\%$ OBM (defined in Equation (2)) with $s = m/2^2$ and batch size $m = 2^k$. Again, the DNBM [18] and DPBM [10] are special cases of the $(100\alpha)\%$ DOBM for $\alpha = 0$ and $3/4$, respectively.
In summary, the (100\%\text{DOBM}) estimator for estimating the variance of the sample mean at step \(k\) is defined as

\[
\hat{\sigma}^2_{(100\%\text{DOBM})} = \frac{1}{q} \sum_{j=1}^{n} \sum_{i=1}^{b_j} \left( \frac{A_j(i)}{m} - \bar{A}_j \right)^2,
\]

where \(m = 2^k\) is the batch size, \(d = b(n/2^k - 1)\) is the denominator, \(b = [(n - m + s)/s]\) is the total number of batches, \(s = m/w\) is the shift, and \(b_j = r_i - 1 + \lfloor m/2^k \rfloor\), \(i = 1, 2, \ldots, w\), are the numbers of full batches in sub-vector \(A_j\). The complete flowchart of the (100\%\text{DOBM}) is shown as Figure 3.

Figure 2. The idea of collapsing in dynamic BM algorithm with 8 vectors.
Note: The numbers 1, 2, … 7 and 8 listed on the arrows indicate the updating order in each iteration.
Figure 3. Flowchart of 100α% DOBM.
3. An efficient implementation of (100α)%DOBM
To meet the small storage requirements, Yang et al. [17] proposed a computational version of DPBM that requires the same storage space as the DNB estimator. In this section, we propose an algorithm to implement the (100α)%DOBM without storing data in whole w vectors.

Instead of keeping data in vectors \(A_j\), \(i=1, 2, \ldots, w\), we develop a computational version of (100α)%DOBM that requires \(w/4\) vectors for storage requirements. That is, the computational version requires only one quarter of the storage space than that for the theoretical version presented in Figure 2. To form the computational version of (100α)%DOBM, we have to define additional two types of quantities to keep the information of the corresponding batch sums in sub-vectors \(A_j\), \(j=w/4+1, \ldots, w\). The first type of quantities \(A_j\), \(A_j\), and \(A_j\) are used to keep the information of the corresponding batch sums in sub-vectors \(A_j\), \(j=w/4+1, \ldots, w/2\). The second type of quantities are used to keep the information of the corresponding batch sums in sub-vectors \(A_j\), \(j=w/2+1, \ldots, w\). The computational version of (100α)%DOBM can be written as a function of the \(w/4\) actual sub-vectors with these additional quantities since the fact is that each cell in \(A_{2j-1}\) and \(A_{2j}\) is collapsed from \(A_j\), \(j=1, \ldots, w/2\).

3.1 Notation
In this subsection, we first define the notation before we introduce the computational version of (100α)%DOBM.

- \(\alpha\): the total number of observations, which is not known in advance.
- \(k\): the total number of times that collapsing has occurred, \(k=0, 1, 2, \ldots, \lceil \log_2 n/(g-1) \rceil\).
- \(m\): batch size, \(m=2^k\).
- \(A_j(i)\), \(i=1, 2, \ldots, 2g; j=1, 2, \ldots, w\): the numerical values (batch sums) stored in the \(i\)-th cell of vector \(A_j\) at step \(k\).
- \(rA_j, j=1, 2, \ldots, w\): the cells used to store the latest observations in \(A_j\); where \(rA_j=1, \ldots, 2g\).
- \(mA_j, j=1, 2, \ldots, w/4\): the number of observations stored in \(A_j(rA_j)\).
- \(b_j, j=1, 2, \ldots, w\): the “numbers of full batches” (in that each batch contains \(m\) observations) in vectors \(A_j\); \(b_j=rA_j-1+[mA_j/m]\).
- \(A_j\): the sum of all full batches in \(A_j, j=w/4+1, \ldots, w/2\).
- \(A_j\): the sum of square of all full batches in \(A_j, j=w/4+1, \ldots, w/2\).
- \(A_j\): the sum of product of adjacent full batches in \(A_j, j=w/4+1, \ldots, w/2\).
- \(A_{j-1}\): the sum of all full batches in \(A_{j-1}\) and \(A_{j}\), \(j=w/4+1, \ldots, w/2\). \(A_{j-1,j} = \sum_{i=1}^{b_j} A_j(i) + \sum_{i=1}^{b_{j+1}} A_j(i)\).

3.2 Algorithm
The logic used to form the computational version of (100α)%DOBM estimator is shown below. The associated flowchart is illustrated in Figure 4.

Step 0 (Initialization): \(n=1\); \(k=0\); \(m=2^k=1\); \(d=0\); \(A_j(i)=0, i=1, \ldots, 2g\); \(j=1, \ldots, w/4\); \(mA_j=0, rA_j=1, f=1, 2, \ldots, w\).

Step 1: If the \(r_1\)-th cell in \(A_j\) has room \((mA_j < m)\), then set \(mA_j \leftarrow mA_j + 1\) and go to Step 5; else go to Step 2.

Step 2: If \(A_j\) has room (i.e. \(rA_j < 2g\)) then go to (2.1), else go to (2.2).

(2.1) Increment the cell for \(A_j\), i.e., set \(rA_j \leftarrow rA_j + 1\). Go to Step 4.

(2.2) Collapse the vectors to update actual and virtual cells, and set \(d=1\) to show collapsing starts. Let \(J=\min\{2^{k+1}, J, w\}\), \(J=\min\{2^{k+1}, w/4\}\), and \(j=J\). Repeat Steps 2.2.1 to 2.2.3 until \(j=0\).

(2.2.1) If \(j > w/2\), update the quantities \(A_{j-1,j}, A_{j-1,j}\) and \(A_j\) as follows:

\[
\begin{align*}
\hat{A}_{j-1,j} &= 2 \cdot \hat{A}_j - \text{temp} \cdot \hat{A}_j - \text{temp} \cdot 2 \hat{A}_j; \\
\hat{A}_{j-1,j} &= 2 \cdot \hat{A}_j + 2 \cdot \hat{A}_j - (\text{temp} \cdot \hat{A}_j) - (\text{temp} \cdot 2 \hat{A}_j); \\
\hat{A}_{j-1} &= \text{temp} \cdot 2 \hat{A}_j + \hat{A}_j; \\
\hat{A}_j &= \hat{A}_j; \\
mA_{j-1} &= mA_j + 2^k; \\
mA_j &= mA_j; \\
rA_{j-1} &= rA_j; \\
rA_j &= rA_j - 2; \\
&= j - 2.
\end{align*}
\]
Figure 4. Flowchart of computational version of \((100\alpha)\%\) DOBM with memory size = \(w/4\).
(2.2.2) If \( w/4 < j \leq w/2 \), update the quantities
\( temp_1 A_j, temp_2 A_j, A_j, \hat{A}_j \) and \( \check{A}_j \) as follows:

- If \( j \) is even,
  \( temp_1 A_j = A_1^j(2) + A_1^j(3); \)
  \( temp_2 A_j = A_1^j(2g - 2) + A_1^j(2g - 1); \)
  \( \check{A}_j = \sum_{i=1}^{g-1} (A_1^j(2i) + A_1^j(2i + 1)); \)
  \( \hat{A}_j = \sum_{i=1}^{g-2} (A_1^j(2i) + A_1^j(2i + 1))^2; \)
  \( \check{A}_j = \sum_{i=1}^{g-2} [A_1^j(2i + A_1^j(2i + 1)) [A_1^j(2i + 2) + A_1^j(2i + 3)]; \)
  \( \hat{A}_j = A_1^j(2g); \)
  \( m_{A_j} = m_{A_j}; \)
  \( r_{A_j} = g; \)
  \( j = j - 1. \)

- If \( j \) is odd,
  \( temp_1 A_j = A_1^j(1) + A_1^j(2); \)
  \( temp_2 A_j = A_1^j(2g - 3) + A_1^j(2g - 2); \)
  \( \check{A}_j = \sum_{i=1}^{g-1} (A_1^j(2i - 1) + A_1^j(2i)); \)
  \( \hat{A}_j = \sum_{i=1}^{g-2} (A_1^j(2i - 1) + A_1^j(2i))^2; \)
  \( \check{A}_j = \sum_{i=1}^{g-2} [A_1^j(2i + A_1^j(2i + 1))] [A_1^j(2i + 2) + A_1^j(2i + 3)]; \)
  \( \hat{A}_j = A_1^j(2g - 1) + A_1^j(2g); \)
  \( m_{A_j} = m_{A_j} + 2^g; \)
  \( r_{A_j} = g; \)
  \( j = j - 1. \)

(2.2.3) If \( j \leq w/4 \), collapse the vector \( A_j \)

- If \( j \) is even, update \( \check{A}_j \) as follows:
  \( A^{j+1}_j(i) = A_1^j(2i) + A_1^j(2i + 1), \quad i = 1, 2, \ldots, g - 1; \)
  \( A^{j+1}_j(g) = A_1^j(2g); \)
  \( m_{A_j} = m_{A_j} + 2^g; \)
  \( j = j - 1. \)
(7.1.2-2) Update quantities
\[ A_{j+1} = A_j + y_n; \]
\[ m_A = m_A + 1. \]

(7.1.2-3) Set \( j = j - 1 \).

(7.2) If \( w/4 < j \leq w/2 \), add the new observation \( y_n \) in virtual vector by updating the quantities \( temp_2, A_j, \hat{A}_j, \tilde{A}_j \) and \( \tilde{A}_j \) as follows:
\[ \hat{A}_j = A_j + y_n; \]
\[ m_A = m_A + 1. \]

(7.2.1) If the \( r_A \)-th cell in vector \( A_j \) has room \( m_A < m \), then set \( m_A \leftarrow m_A + 1 \) and go to (7.2.3).

(7.2.2) Update quantities
\[ \hat{A}_1 = A_j + \hat{A}_j; \]
\[ \tilde{A}_1 = \hat{A}_j + [\hat{A}_j]^2; \]
\[ \tilde{A}_2 = \tilde{A}_1 + temp_2 A_j \times \hat{A}_j; \]
\[ temp_2 = \hat{A}_j; \]
\[ \tilde{A}_3 = 0; \]
\[ m_A = 0; \]
\[ r_A = r_A + 1. \]

(7.2.3) Set \( j = j - 1 \).

(7.3) If \( j \leq w/4 \), add the new observation \( y_n \) in actual vector \( A_j \).

4. Evaluation for fast computation
This section shows the performance of the proposed computational version of \((100\alpha)\%\text{DOBM}\) algorithm in terms of the CPU running time calculations.

The computational environment is under Intel Core2 Duo CPU 2.40 GHz with 3.50 GB RAM. The run time is collected in millisecond (ms), where 1 ms = 10^-3 s. The estimated CPU time for the theoretical and computational versions of \((100\alpha)\%\text{DOBM}\) algorithm is shown in Table 1. The values listed in the parentheses are the standard errors below the corresponding estimates. We apply leading-digit rules [14] to report all estimates. The standard errors are computed based on 100 replications. We apply leading-digit rules [14] to report all estimates. That is, we report the point estimate through the leading digit of its standard error.

The experimental results indicate that the performance of the proposed algorithm is better than that of the theoretical algorithm when the sample size \( n \) is large as simulation run length increases. The observations are straightforward and easy to understand. Since the data from simulation are stored in the vector in general-purpose computer languages, e.g. C++, it is a little time consuming in collecting data using actual cells. The proposed algorithm uses less actual cells to store data than that of the theoretical algorithm, some algebra calculations take the place of the cells to keep the information of the batch sums. Consequently, the proposed algorithm performs better than theoretical version in terms of the fast computation.

5. Summary
This article proposes a procedure, called computational version of \((100\alpha)\%\text{DOBM}\), for estimating the variance of the sample mean considering small storage requirements and fast computation, without requiring knowledge of the simulation run length a priori. Theoretical version is conceptually easier to understand while computational version requires one quarter of the storage space than that for the theoretical version. Our simulation results further show that the computational version of algorithm performs better than theoretical version in terms of the CPU running time.
time with large sample size. Both versions are algebraically equivalent.

Acknowledgements
This research is supported by the National Science Council of the Republic of China under Grant No. NSC-93-2213-E-007-060.

Notes on contributors
Mingchang Chih is an Assistant Researcher in the System Development Center at the Chung-Shan Institute of Science and Technology. He received his undergraduate degree in Industrial and Systems Engineering from Chung Yuan Christian University in June 2001 and master’s degree in Industrial Management from National Taiwan University of Science and Technology in June 2003. He received his PhD from the Department of Industrial Engineering and Engineering Management at National Tsing Hua University in July 2009. His research interests are applied operations research, simulation modeling and output analysis, and algorithms.

Wheyming Tina Song is a Professor in the Department of Industrial Engineering and Engineering Management at the National Tsing Hua University in Taiwan. She received her undergraduate degree in Statistics and Master’s Degree in Industrial Engineering at Cheng-Kung University in Taiwan in 1979. She then received her Master’s Degree in Applied Mathematics in 1983 and industrial engineering in 1984, both from the University of Pittsburgh. She received her PhD from the School of Industrial Engineering at Purdue University in 1989. She joined Tsing Hua in 1990 after spending one year as a Visiting Assistant Professor at Purdue IE. Her research interests are applied operations research, probability and statistics, and the statistical aspects of stochastic simulation, including input modeling, output analysis, variance reduction, and ranking and selection.

References
[7] Meketon, M.S. and Schmeiser, B.W., Overlapping batch means: something for nothing? Proceedings of...


Appendix: The C code for implementing the Computational Version of (100α)%OBM

/****************************
** FUNCTION: gdpmc() **
** ** Purpose: implement the Computational version algorithm with only w/4 storage space. **
** ** Input: **
** ** w: total number of vectors, w=1,2,4,8,16,32,... **
** ** g: prespecified memory size. **
** ** tg: memory size of each vector A_i, i=1,...,w, i.e. 2*g. **
** ** ac: storage space, i.e. vector A_i, i=1,...,w/4. **
** ** k: total number of times collapsing has occurred **
** ** m: the batch size at k-th step, m=2^k **
** ** ma_i: numbers of observations stored in A_i, i=1,...,w. **
** ** ra_i: the cell used to stored the latest observations in vector A_i, i=1,...,w. **
** ** data: observation sequence. **
** ** n: number of observations in the sequence. **
** ** dmean: mean of the stochastic process. **
** ** dpbm_full: dpbm estimator considered full batches. **
** ** dpbm_all: dpbm estimator considered all batches. **
** ** Output: **
** ** dpbm_full: dpbm estimator considered full batches. **
****************************/

void gdpmc(long w, long g, long tg, double *ac, long *k, long *m, long *ma, long *ra, double *data, int n,double dmean, double *dpbm_full)
{
    long bb=0;
    double esti=0,db=0;
    int i,1,j=0,jj=0,jjj=0,ss=0,d=0;

double *part;
    long *b; // consider full batch

    *k=0, *m=1, d=0;

temp1A = (double *)calloc( w, sizeof(double));
temp2A = (double *)calloc( w, sizeof(double));
dotA = (double *)calloc( w, sizeof(double));

ddotA = (double *)calloc( w, sizeof(double));

breveA = (double *)calloc( w, sizeof(double));
dotAA = (double *)calloc( w, sizeof(double));

graveA = (double *)calloc( w, sizeof(double));

for (i=0;i<w;i++)
{
    ma[i]=0;
    ra[i]=1;
    temp1A[i]=0.;
    temp2A[i]=0.;
    dotA[i]=0.;
    ddotA[i]=0.;
    breveA[i]=0.;
    dotAA[i]=0.;
    ddotAA[i]=0.;
    graveA[i]=0.;
}
for (l=0; l<n; l++)
{
    if (ma[0]<=m)
    {
        ma[0] = ma[0] + 1;
    }
    else
    {
        if (ra[0]<g)
        {
            ra[0] = ra[0] + 1;
        }
        else
        {
            d=1;
            if((long)pow(2,(l+k+1))<w)
            {
                j=(int)pow(2,(l+k+1));
                jji=j;
            }
            else
            {
                j=w;
                jji=j;
            }
            if((long)pow(2,(l+k+1))<w/4)
            {
                jj=(int)pow(2,(l+k+1));
            }
            else
            {
                jj=w/4;
            }
            do {
            if(j<w/4)
            {
                if(j%2==0)
                {
                    for(i=1; i<g-1; i++)
                    {
                        ac[(j-1)*tg+i-1] = ac[(j-2-1)*tg+2*i-1] + ac[(j-2-1)*tg+2*i+1-1];
                    }
                    ac[(j-1)*tg+g-1] = ac[(j-2-1)*tg+tg-1];
                    ma[j-1] = ma[j-1];
                }
                else
                {
                    for(i=1; i<g; i++)
                    {
                        ac[(j+1)*tg+i-1] = ac[((j+1)/2-1)*tg+2*i-1-1] + ac[((j+1)/2-1)*tg+2*i+1-1];
                    }
                }
            }
            else
            {
                j = j - 1;
            }
            }
        }
    }
}
else if((j/4&4)&&(j&u/2))
{ if(j%2==0)
{
    temp1A[j-1] = ac[(j/2-1)*tg*2-1] + ac[(j/2-1)*tg*3-1];
    temp2A[j-1] = ac[(j/2-1)*tg*tg-2-1] + ac[(j/2-1)*tg*tg-1-1];
    for(i=1;i<=g-1;i++)
    {
        ddototA[j-1] = ddototA[j-1] + pow(ac[(j/2-1)*tg*2*i-1-1]+ac[(j/2-1)*tg*2*i-1],2);
    }
    for(i=1;i<=g-2;i++)
    {
        breve[j-1] = breve[j-1] + (ac[(j/2-1)*tg*2*i-1-1]+ac[(j/2-1)*tg*2*i-1] + ac[(j/2-1)*tg*2*i+1-1])
        *(ac[(j/2-1)*tg*2*i+2-1]+ac[(j/2-1)*tg*2*i+3-1]);
    }
    graveA[j-1] = ac[(j/2-1)*tg*tg-1-1];
    na[j-1] = na[j-2-1];
    rs[j-1] = g;
  }
  else
  {
    temp1A[j-1] = ac[(j+1)/2-1)*tg*1-1] + ac[(j+1)/2-1)*tg*2-1];
    temp2A[j-1] = ac[(j+1)/2-1)*tg*tg-3-1] + ac[(j+1)/2-1)*tg*tg-2-1];
    for(i=1;i<=g-1;i++)
    {
        dotA[j-1] = dotA[j-1] + ac[(j+1)/2-1)*tg*2*i-1-1] + ac[(j+1)/2-1)*tg*2*i-1];
        ddototA[j-1] = ddototA[j-1] + pow(ac[(j+1)/2-1)*tg*2*i-1-1]+ac[(j+1)/2-1)*tg*2*i-1],2);
    }
    for(i=1;i<=g-2;i++)
    {
        breve[j-1] = breve[j-1] + (ac[(j+1)/2-1)*tg*2*i-1-1]+ac[(j+1)/2-1)*tg*2*i-1] + ac[(j+1)/2-1)*tg*2*i+1-1])
        *(ac[(j+1)/2-1)*tg*2*i+2-1]+ac[(j+1)/2-1)*tg*2*i+3-1]);
    }
    graveA[j-1] = ac[(j+1)/2-1)*tg*tg-1-1];
    na[j-1] = na[(j+1)/2-1] + (long) pow(2,stk);
    rs[j-1] = g;
  }
}
j = j - 1;

else
{
    - pow(temp2A[j/2-1],2));
    graveA[j-1] = graveA[j/2-1];
    ma[j-1] = ma[j/2-1];
\[
\text{ma}[j-1-1] = \text{ma}[j/2-1] + \text{(long)} \text{pow}(2, *k);
\text{ra}[j-1] = g;
\text{ra}[j-1-1] = g;
\]
\[
j = j - 2;
\]
\[
} \text{ while}(j>0);
\]
\[
\text{ra}[0] = g + 1;
*\text{k} = *\text{k} + 1;
*\text{m} = \text{(long)}\text{pow}(2, *\text{k});
\]
\[
\text{if}(jj>=2)
\{
\text{for}(i=2; i<=jj; i++)
\{
\text{ra}[i-1] = g;
\}
\}
\]
\[
\text{ac}[\text{ra}[0]-1] = 0,
\text{ma}[0] = 1;
\}
\]
\[
\text{ac}[\text{ra}[0]-1] = \text{ac}[\text{ra}[0]-1] + \text{data}[l];
\]
\[
\text{if}(d==1)
\{
\text{for}(j=2; j<=jj; j++)
\{
\text{if}(j<=\text{u}/4)
\{
\text{if}(\text{ma}[j-1]==*\text{m})
\{
\text{ra}[j-1] = ra[j-1] +1;
\text{ac}[(j-1)*tg+\text{ra}[j-1]-1] = 0;
\text{ma}[j-1] = 0;
\}
\text{ac}[(j-1)*tg+\text{ra}[j-1]-1] = \text{ac}[(j-1)*tg+\text{ra}[j-1]-1] + \text{data}[l];
\text{ma}[j-1] = \text{ma}[j-1] + 1;
\}
\text{else if}((j>\text{u}/4)\&\&(j<=\text{u}/2))
\{
\text{graveA}[j-1] = \text{graveA}[j-1] + \text{data}[l];
\text{ma}[j-1] = \text{ma}[j-1] + 1;
\}
if (ma[j-1]==m)
{

    temp2A[j-1] = graveA[j-1];
    graveA[j-1] = 0.;
    ma[j-1] = 0;
    ra[j-1] = ra[j-1] +1;
}
else
{
    if (j%2==0)
    {
        ma[j-1] = ma[j-1] + 1;

        if (ma[j-1]==m)
        {
            ddotAA[j-1-1] = ddotAA[j-1-1] + pow(graveA[j-1], 2);

            graveA[j-1]=0.;
            ma[j-1] = 0;
            ra[j-1] = ra[j-1] +1;
        }
    }
    else
    {
        ma[j-1] = ma[j-1] + 1;

        if (ma[j-1]==m)
        {
            ddotAA[j-1] = ddotAA[j-1] + pow(graveA[j-1], 2);

            graveA[j-1]=0.;
            ma[j-1] = 0;
            ra[j-1] = ra[j-1] +1;
        }
    }
}
}
if (w > m)
{
    ss = 1;
}
else
{
    ss = (int) ceil((m+ss)/w);
}

bb = (long)floor((n - *m + ss)/ss);
db = (double) bb * n / (*m) - bb;

part = (double *)calloc(w, sizeof(double));
b = (long *)calloc(w, sizeof(long));

for (i=0; i<w; i++)
{
    part[i] = 0.0;
    b[i] = ra[i] - 1 + (long)floor(ra[i] / *m);
}

for (l=0; l<w/4; l++)
{
    for (i=0; i<b[l]; i++)
    {
        part[i] = part[i] + pow((ac[l+tg]+l)/(n-m)+dmean,2.0)/db;
    }
    esti = esti + part[i];
}

for (l=w/4; l<j; l++)
{
    part[i] = ddotA[i]/pow(m,2) - dmean*(dotA[i]/pow(2,(*k-1)) - b[l]*dmean);
    esti = esti + part[i]/db;
}

for (l=j; l<w/2; l++)
{
    part[i] = ddotA[i]/pow(m,2) - dmean*(dotA[i]/pow(2,(*k-1)) - b[l]*dmean);
    esti = esti + part[i]/db;
}

*dpbm_full = esti;

free(part);
free(b);
free(temp1A);
free(temp2A);
free(dotA);
free(ddotA);
free(breveA);
free(dotAA);
free(ddotAA);
free(graveA);

return;
}
於模擬輸出分析中高效率執行動態批量估計式之方法

遲銘璋
中山科學研究院系統發展中心
32546桃園縣龍潭鄉佳安村中正路佳安段481號
桑慧敏
國立清華大學工業工程與工程管理學系
30013新竹市光復路二段101號工程一館

摘要

估計樣本平均數變異數是模擬學領域中一個相當重要的課題。傳統的分批估計式 (batch means estimators) 必須事先預知或給定該次模擬實驗的樣本數目 (simulation run length) 才可使用。動態分批估計式 (dynamic batch means) 是目前最新的演算法, 可於固定的儲存空間之下利用動態改變批量大小 (batch size) 及批次量 (number of batches) 來執行傳統的分批估計式, 完全可由事先預知樣本數目。本論文提出了一個改善後的動態分批估計式, 其所需要的儲存空間更少與其運算速度更有效率, 本研究提出的演算法將更有利於模擬實驗的樣本數目未知與樣本數目極大的情況所使用。

關鍵詞: 樣本平均數變異數、分批估計式、模擬

(*聯絡人:wheyming@ie.nthu.edu.tw)