JOURNAL OF SPACECRAFT AND ROCKETS
Vol. 34, No. 4, July–August 1997

Efficient Calculation of a Jitter/Stability Metric

Daniel P. Giesy
Lockheed Martin Corporation, Hampton, Virginia 23681

A tool for computing a jitter/stability metric used in NASA requirements statements is developed. An efficient algorithm is given for computing this metric. Two ways of implementing it on a computer are discussed. One is optimized for computational speed; the other sacrifices some speed to conserve memory. Timing studies are given to show that the improvement of computation times using the present algorithm over previously existing techniques can run to several orders of magnitude and that previous techniques were so costly that the present algorithm represents enabling technology. Further comparisons show that the memory-conservative implementation runs at about half the speed of the fast implementation but can cut the major data storage requirement of the fast implementation by 95–99%, making the algorithm implementable on much smaller computers, such as personal computers, than it would be otherwise. Software for both implementations is included in version 2.0 of the NASA time- and frequency-domain analysis program PLATSIM.

Nomenclature

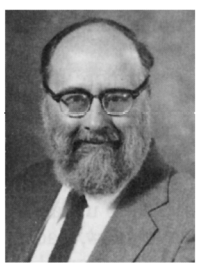
a, b, c, d = constants used in the jitter calculation timing formula, s
$J(y, w)$ = jitter (see Definition 1)
$k, k_1, \ldots, k_m$ = number of data points in a time series sampled at equal time increments that are covered by windows $w, w_1, \ldots, w_m$
l = number of different time signals being simultaneously analyzed by the memory-conservative implementation of the algorithm
$l_1, \ldots, l_m$ = dominant minimum pointers (same units as $t_1, \ldots, t_n$)
m = number of windows for which jitter/stability is to be determined
m_i = number of windows for which jitter/stability is to be determined in timing test case i
N = integer parameter used by the memory-conservative implementation of the algorithm in managing dynamic memory allocation
n, n_i = number of data points in a time series
T = total duration of discrete time signal, $t_n - t_1$ (same units as $t_1, \ldots, t_n$)
$t_1, t_2, \ldots, t_n$ = sample times for a discrete time series (arbitrary time units can be used; for a spacecraft instrument boresight pointing-error application, typical units would be seconds)
$u_1, \ldots, u_m$ = dominant maximum pointers (same units as $t_1, \ldots, t_n$)
$w, w_1, \ldots, w_m$ = lengths of jitter/stability defining windows (same units as $t_1, \ldots, t_n$)
y = discrete time series of n points (units are problem dependent; for a spacecraft instrument boresight pointing-error measurement, typical units would be arcseconds)
$y(i)$ = $y(t_i)$; notation used if time points are equally spaced
$y(t_i)$ = element i of time series y (same units as y)
$y_{J_{(w)}}$ = jitter history (see Definition 1)
$y_{L_{(w)}}$ = lower bound of y (see Definition 1)
y_U = upper bound of y (see Definition 1)
z_i = calculation time per data point of test case i, s
τ_i = execution time of test case i, s

Subscripts

i = time-point number, $1 \leq i \leq n$
i = test-case number, $1 \leq i \leq 3808$
j = window number, $1 \leq j \leq m$

Introduction

THE purpose of this paper is to provide a computational tool whose uses include design and analysis of spacecraft. An efficient method is presented for calculating a metric that has been used by NASA to quantify requirements for pointing jitter and stability in spacecraft instrumentation. This metric has been used to state requirements for the Upper Atmosphere Research Satellite (UARS), the Earth Observing Satellite (EOS), and individual instruments on these satellites. This jitter/stability metric has the advantage of having a high intuitive content; it is easy for an engineer to picture the connection between the jitter/stability value this metric assigns to any given time signal and the level of noise or drift in the signal.



Daniel Giesy received his B.A. and M.A. in 1960 from Ohio State University and his Ph.D. in mathematics in 1964 from the University of Wisconsin–Madison. He then worked in the mathematics faculties at the University of Southern California, Western Michigan University, and Norfolk State University. Since 1977 he has provided technical support services under contract to NASA Langley Research Center. He has designed algorithms and written software for control system design and analysis, specializing in optimization and numerical linear algebra. He is currently working on the Next Generation Revolutionary Analysis Design Environment and the High Speed Research Program. Dr. Giesy has recently taken a position as a mathematician with the Guidance and Control Branch (Mail Stop 161), Flight Dynamics and Control Division, NASA Langley Research Center, Hampton, VA 23681.

Received June 27, 1996; revision received March 22, 1997; accepted for publication March 27, 1997.

Because NASA states requirements in terms of this metric, it is important to be able to calculate it. However, anecdotal information indicated that when attempts were made to calculate the jitter or stability of long time signals, the computational burden was unacceptable. At a minimum, overnight or weekend computer runs were required to complete a single analysis, and some problems, such as the estimated 127-day example presented later, were too computationally intensive to be feasible.

Another feature of this jitter metric is that its very definition immediately leads one to an easy algorithm for its computation. Anecdotal information indicates that this was the algorithm in use when unacceptable computational burdens were encountered. The problem is that this algorithm is not the most efficient way to calculate this jitter metric. The exact degree of inefficiency depends on the signal length, the sampling frequency, and another problem parameter (the window length), but problems of engineering interest have been found for which the easy algorithm is three or four orders of magnitude slower than the algorithm to be presented in this paper. The penalty paid for the additional computational speed of the new algorithm is a vast increase in algorithmic complexity.

Examples where this metric has been used can be found in industry working papers[1,2] and in conference proceedings[3,4]; an occasional mention can even be found in the journal literature.[5] Reference 3 says, "Jitter . . . requirements [are] expressed as a worst-case change in pointing . . . across each time interval of interest . . .." In Ref. 2 "pointing jitter" is defined to be "the peak-to-peak variation of the actual pointing direction over relatively short time intervals," and "pointing stability" has the same definition except that the time intervals are "relatively long."

In this paper, the time intervals used as a parameter in the definition of the jitter/stability metric are called windows. Then the jitter or stability value of a given time signal with respect to one of these windows is found by placing the window over the time signal in such a position as to maximize the peak-to-peak variation of the portion of the signal that is under the window. It is this maximum peak-to-peak variation that is taken as the value of the metric. Because the same algorithm is used to calculate either the jitter metric or the stability metric, the word jitter will be used for the remainder of this paper to refer to either jitter or stability as defined earlier.

The technique presented in this paper can be applied to an arbitrary discrete function of finitely many time points. The units will be determined by the specific use that is being made of it. For quantifying the jitter in the boresight pointing error of an observational instrument on a spacecraft, typical units for time might be seconds and units for the boresight error might be arcseconds. On the other hand, if one is looking at irregularities in tectonic plate drift, the units might well be centuries and centimeters. So, for the most part in this paper, the units will be left unstated.

Figure 1 shows one such window in four of its possible positions on a discretely sampled function of time. The length (duration) of each time window, represented by the width of the rectangle, is the same for all four window positions. The top and bottom of each rectangle are positioned to show the peak-to-peak variations of the function within the window. The tallest rectangle represents one window position where the peak-to-peak variation reaches its worst case value, so its height represents the jitter in this time function for this window size. Note that the leftmost window extends into the negative part of the time axis where there are no data points from the discrete time signal. Including such windows does not change the value calculated for the jitter metric and will be useful in implementing the algorithm.

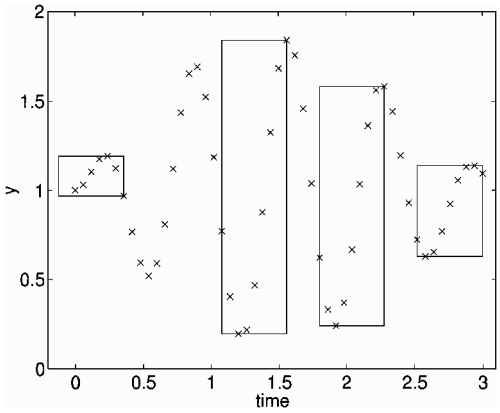


Fig. 1 Jitter window in four positions.

To calculate this jitter metric with respect to a given window, a position must be found for this window that maximizes the peak-to-peak variation of the time signal under the window. The easy algorithm to accomplish this calculation is to simply try all window positions and pick the worst case from among them. If the signal is discrete and of finite length (discrete signals of finite length can arise if the data come from sampling the output of some sensor or if the data are generated by a numerical simulation), then there are only a finite number of window positions with the property that any two of them cover different sets of points; so this is an algorithm that can be implemented on a computer. This search can be performed in a systematic fashion by first placing the window at the beginning of the time signal and calculating the peak-to-peak variation under it, then moving the window forward in time until it covers one additional point of the time signal and repeating the peak-to-peak variation determination, and continuing to advance the window one point at a time and determine the new peak-to-peak variation until the end of the signal is reached. The measure of jitter is then the worst of these peak-to-peak variations. This is easily programmed in Fortran using a nested pair of DO loops or in MATLAB® (MATLAB is a registered trademark of The MathWorks, Inc.) using a vectorized determination of the peak-to-peak variation under the window in a given position inside a single FOR loop.

If the data points in a discrete signal are equally spaced, the computational complexity of the jitter calculation for each window is determined by two parameters: n, the number of points in the data sequence, and k, the number of points that lie under the chosen window. These numbers can be quite large. For example, the data may come from a simulation driven by disturbances with high-frequency components, so that accuracy requires an integration interval that is small compared to the window size, resulting in a large value for k. A small integration interval will also drive up the value of n, as will a need to consider long-term effects.

If the calculation of jitter is programmed on a computer using the easy algorithm, the time to calculate the peak-to-peak variation of the signal under the window in one position is $O(k)$, and $n-k+1$ window positions must be considered, so the total calculation time is $O[k(n-k+1)]$. If n is large and $1 \ll k \ll n$ (in one jitter analysis of the EOS AM-1 spacecraft,[4] values of n and k on the order of 10^5 and 10^4, respectively, were common), this takes a substantial amount of time. In an analytic jitter study, the computational expense can be driven up by such factors as these.

1) Each output signal may need to be analyzed over several different windows (four to seven in the EOS AM-1 study).

2) For each disturbance scenario, many output signals need to undergo the analysis in step 1 (20–30 in the EOS AM-1 study; representative times for a jitter calculation at this level using both the easy algorithm and the one given in this paper are given in Table 1).

3) Step 2 must be repeated for many disturbance scenarios (over 20 in the EOS AM-1 study).

4) Step 3 must be repeated for each design iteration.

At this point, the need for a more efficient jitter calculation should be apparent. At the same time, approximation techniques should be avoided if possible.

The purpose of this paper is to present a much more efficient algorithm that calculates the NASA jitter metric exactly. This algorithm can be used to simultaneously calculate jitter in a time series for each of several windows. A list is maintained of potentially significant maxima and another of minima as the time series is scanned. At each time step, information from these lists is used to update variables that accumulate the metric of jitter for each window, leading, at the end of the scan, to a determination of jitter values for all windows. If there are m windows, the time to do this is $(am+b)n$, where the constants a and b are independent of the actual window sizes. The calculation time for the first window is $(a+b)n$. Comparing this

with the timing estimate $O[k(n-k+1)]$ for the easy algorithm, it is seen that the removal of the dependence on the window length k has replaced a timing formula that is jointly quadratic in n and k with one that is linear in n and independent of k. The incremental calculation time for each additional window is an. This incremental time for each additional window has been empirically observed to be about 30% of the time for the first window, so the computational time to calculate jitter for several windows simultaneously is substantially better than to calculate it for each window individually.

This algorithm has been implemented in software and included in version 2.0 of the PLATSIM software package.[6] PLATSIM is a MATLAB-based software package, developed at NASA Langley Research Center, which performs time- and frequency-domain analysis of the response of a controlled or uncontrolled flexible structure to disturbances. Version 1.0 of PLATSIM[7,8] contains an earlier version of this algorithm (without the memory-conservative option).

First, the algorithm is given. Then, notes on two implementations of the algorithm are presented. The first implementation emphasizes computational speed; the second implementation sacrifices some computational speed in order to conserve computer memory. Results of timing studies to quantify the improvements in computational time and to show the dependence of timing on problem parameters are shown. A few remarks on the amount of memory saved using the memory-conservative implementation of the algorithm are made. The paper ends with conclusions.

Jitter Calculation Algorithm

In this section the problem of calculating the jitter in a time series over one or more jitter windows is dealt with. The windows under which the maximum peak-to-peak excursions of the time series are being observed are thought of as sliding continuously down the time axis, but for purposes of determining jitter, a window need only be considered when it is positioned so that its right end is coincident with one of the points of the time series. (A window in this position is referred to as being in standard position; the windows shown in Fig. 1 are placed in standard position.) This can be seen by observing that, if a window in standard position starts sliding to the right, the maximum peak-to-peak excursion under that window cannot increase until a new point of the time series comes into the window. This happens exactly when the window reaches the next standard position. So no jitter information is lost by ignoring the window in any of the intermediate positions.

Notation. Let n be the number of points in the time series, and denote the time series by $\{y(t_i) \mid 1 \le i \le n\}$. Assume that $t_1 < t_2 < \cdots < t_n$. Suppose that there are m windows and they have lengths $w_1 < \cdots < w_m$. If the time points are equally spaced, the time series may be written as $\{y(1), y(2), \ldots, y(n)\}$, and k_j is used to denote the number of points covered by a window of length w_j in standard position.

In Fig. 1, a time series is shown with $n = 51$, $t_1 = 0$, and $t_n = 3$. One window is shown (in four different positions), so $m = 1$. For this window, $w_1 = 0.48$, and so $k_1 = 9$.

Definition 1. For a window of length w, denote the running tallies of the upper and lower bounds of y under each window position by $y_U^{(w)}$ and $y_L^{(w)}$, respectively, and define them by

$$y_U^{(w)}(t_i) = \max\{y(t_j) \mid t_i - w \le t_j \le t_i\} \tag{1}$$

$$y_L^{(w)}(t_i) = \min\{y(t_j) \mid t_i - w \le t_j \le t_i\} \tag{2}$$

where

$$1 \le i \le n \tag{3}$$

Denote the jitter time history of y for this window by $y_J^{(w)}$, and define it by

$$y_J^{(w)}(t_i) = y_U^{(w)}(t_i) - y_L^{(w)}(t_i), \qquad 1 \le i \le n \tag{4}$$

Finally, denote the jitter by $J(y, w)$, and define it by

$$J(y, w) = \max\left\{y_J^{(w)}(t_i) \mid 1 \le i \le n\right\} \tag{5}$$

Where the window length is clear from context, the superscript (w) will be omitted.

In Fig. 1, the windows shown correspond to $i = 7$, 27, 39, and 51. The numbers $y_U(t_i)$, $y_L(t_i)$, and $y_J(t_i)$ represent, respectively, the top y coordinate, the bottom y coordinate, and the height of the corresponding rectangle. Note that, even though $k = 9$, the leftmost pictured window covers only seven points; its left side extends into negative time, where the time series is not defined.

One easily sees that, in Eq. (5), the same value will be computed for $J(y, w)$ if the lower limit of 1 is changed to a larger number, so long as t_1 lies under the first window used. If jitter were to be calculated using the easy algorithm, this lower limit would be k, the number of points covered by a window of length w positioned at the beginning of the time series. Doing the calculation by using the easy algorithm refers to actually doing the calculations in Eqs. (1), (2), (4), and (5). Referring to Fig. 1, if one wished to calculate jitter using the easy algorithm, it would be inefficient to use the first window position shown there. It would be more efficient to start with a window positioned with its left edge at time 0. If the time steps are evenly spaced so that there are k points under a window in standard position, then doing the calculations in Eqs. (1) and (2) requires k references into the y vector for each entry in y_U and y_L. Because the first $k-1$ entries of y_J are dominated by $y_J(k)$, these need not be computed, and only the last $n-k+1$ values of y_U and y_L are needed.

The comparative speed of the algorithm to be presented in this paper can now be explained. When jitter is calculated using the easy algorithm, then for typical values of n and k, most entries in y are being referenced k times. The algorithm presented here reduces the number of references per time-series entry from k (for most time-series elements) to a small number that is bounded independently of k. Some overhead is incurred on each pass through a time series. This is amortized over all of the windows being considered in the pass, so the bound on the number of times a given series element is referenced per window improves as more windows are considered simultaneously.

If one only wishes to calculate $J(y, w)$, it is not necessary to spend computer memory to contain the full arrays y_U, y_L, and y_J. For each i, one could calculate the ith element of each of these arrays, accumulate the necessary information from this calculation in the jitter-to-date tally, and discard these ith elements.

The following discussion details the calculation of $y_U(t_i)$. The calculation of y_L is parallel. One need only reverse the direction of inequalities involving elements of y and replace "decreasing" by "increasing" in statements about the monotonicity of potentially significant extrema lists of elements of y.

The outer loop of the easy algorithm can be thought of as making a pass through the time signal. Each step of this pass places the scanning window in a new position. The inner loop then scans every element of the time signal under the window to determine the maximum and minimum so the peak-to-peak variation can be computed. If this calculation needs to be done for more than one window, the calculation for each window is traditionally done independently of that for the others.

The present algorithm also makes a pass through the y vector. However, instead of scanning the length of a window back from the current position to determine the maximum and minimum under that window, the information needed to find that maximum and minimum has already been recorded in some bookkeeping arrays in a manner that allows for efficient retrieval and update. Another advantage to this technique is that the jitter calculation can be done for several windows simultaneously at less computational cost per window than performing the calculation for each window individually. Two lists are being maintained, one a list of potentially significant maxima and the other a list of potentially significant minima. The list of potentially significant maxima (respectively, minima) contains elements of the y vector that have already been scanned and that have the potential of being the maximum (respectively, minimum) element in one of the windows in the present and/or some future position of the pass. Each window has three bookkeeping items associated with it: a jitter-to-date tally, a dominant-maximum pointer, and a dominant-minimum pointer. The jitter-to-date tally

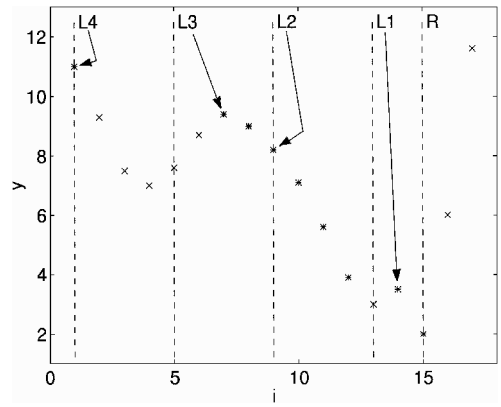
records how much jitter there is in the time series up to the point most recently scanned. The dominant-maximum pointer points at the location in the list of potentially significant maxima that gives the maximum value of the time series under the window when its right end is at the current element of the pass. There is an analogous relationship between the dominant-minimum pointer and the list of potentially significant minima. All of these bookkeeping elements must be updated with each step through the y vector. The efficiency of the algorithm comes in that this update can be accomplished in an average time bounded by an expression of the form $am + b$, where a and b are absolute constants that are, in particular, independent of the lengths of the individual windows.

The process of making one step through the y vector can be thought of as occurring in the following sequence. Initially, all windows have their right endpoints at the last element of the y vector that has been scanned; for each window, the jitter in y up to the last scanned point has been entered into a jitter-to-date tally; all of the information in the lists of potentially significant extrema is relevant to windows in this position and future positions; and the dominant-element pointers are correct for this window position. Then, the next point of the y vector is scanned, and the windows all slide from their old positions to their new positions with their right endpoints all coincident with the time at which the new y point occurs. Then, the new lists of significant local extrema are formed from the old. Then, the dominant-element pointers are updated to be correct for the new window position. Finally, the jitter-to-date tally for each window is updated with new information from this step.

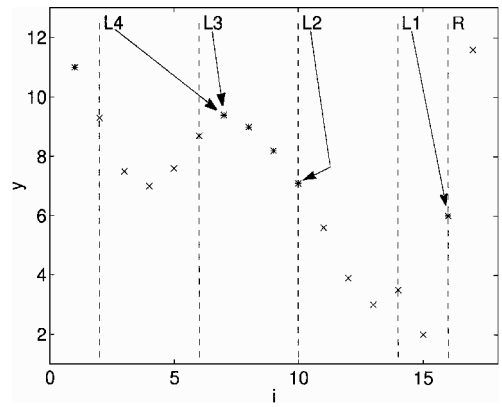
The process of advancing the calculation of y_U by one time step can be observed from Figs. 2a and 2b. In both figures, the same generic time sequence y is shown. It is sampled at 17 equally spaced points, so $n = 17$. There are four windows with respect to which jitter is being calculated ($m = 4$). The window lengths are given by the point counts $k_1 = 3$, $k_2 = 7$, $k_3 = 11$, and $k_4 = 15$. Figure 2a shows the situation in computing y_U after the scan of y has reached $y(15)$, and Fig. 2b shows the situation after reaching $y(16)$. The four windows overlap, having a common right edge at the dashed line labeled R. The left edge of w_1 is at the dashed line labeled L1, the left edge of w_2 is at L2, and so on. Potentially significant maxima are shown as stars, other data points as crosses. Dominant-element pointers are shown for each window. Thus, $y_U^{(w_1)}(15) = y(14)$, $y_U^{(w_2)}(15) = y(9)$, etc.

At the point of the calculation shown in Fig. 2a, the algorithm has only seen $y(1), \ldots, y(15)$, and it is computationally efficient for the algorithm to assume that the series goes on indefinitely. One possibility the algorithm must consider is that the series is monotone decreasing for at least another 14 points. If that is the case, every one of the potentially significant maxima either is the maximum element in one of these windows in the present position (these are marked by the dominant-element pointers) or will become the maximum element under one or more windows at some future position (or both). The possibility that it might become a maximum in some future window is the potential significance of each of these elements in computing the maxima of the series y over some jitter windows. Note that the subsequence of y formed by the potentially significant maxima is monotonically decreasing and that each dominant-element pointer points to the leftmost potentially significant element under its window. This always happens and can be established by induction on i.

Now observe what happens when the scan of y advances to $i = 16$ in Fig. 2b. The newly scanned element always becomes potentially significant. In this example, it is bigger than several of the potentially significant maxima at the previous stage (located at $i = 15$, 14, 12, and 11), so they lose their potential significance. Note that the demoted elements were placed consecutively at the right end of the old list of potentially significant maxima. This always happens, and, because of it, the demotion of previously potentially significant extrema is an efficient computation. The dominant-element pointers are updated. For window 1, the previous dominant element lost significance, so the pointer was redirected to the newly scanned element $y(16)$. For windows 2 and 4, the previous dominant element dropped out of the left end of the window as it moved on, so the pointer was moved to the right in the list of potentially significant



a) Before update



b) After update

Fig. 2 Potentially significant maxima and dominant-element pointers.

maxima until an element was found in the window. Because points are equally spaced, the pointer needed only to move to the next element in the list of potentially significant maxima. For unequally spaced points, several steps might have been necessary. The third window could leave its dominant-element pointer where it was in the previous step. Note that, at this point, the same element, $y(7)$, is the maximum under both the third and fourth windows.

Note that, in one more step ($i = 17$), the only potentially significant maximum will be the new point $y(17)$ and all dominant-element pointers will point to it.

The efficiency of the algorithm comes from the way in which the lists of potentially significant extrema and the dominant-element pointers are updated from step to step. As another step is taken, the new element from the y vector is potentially significant as both a maximum and a minimum. Whether either of these potentials is realized will depend on future behavior of the y vector. So the new element will be added to the end of both lists. What is important is that the presence of this new element on a list may remove any potential significance that some of the previous elements had, so they are deleted from the list.

Specifically, when a new element is added to the list of potentially significant maxima, then any element already on the list that is less than or equal to the added element is no longer potentially significant. This loss of potential significance is seen by observing that a window in its new position, or in any future position that covers the earlier, dominated element, also covers the element just added, so the earlier element is not needed to calculate the peak-to-peak excursion in any such window. Thus, the list of potentially significant maxima will be updated by removing all elements less than or equal to the new element and then adding the new element at the end of the resulting list. Consequences of updating the list in this way are that the elements in this list form a monotonically decreasing sequence and that the set of elements deleted from the list as the result of the new addition are located consecutively at the end of the preupdate list. These consequences may be demonstrated

by induction on the position of the scan in the y vector (at the initial step, the list contains a single element and so is vacuously monotonic decreasing). The list of potentially significant minima is updated similarly.

Because the list of potentially significant maxima is a decreasing sequence, the dominant-maximum pointer for each window points as far to the left in the list as it can and still have the element it points to lie in the window in its present position. This characteristic of the positions of the dominant maxima simplifies the process of updating their pointers after each step.

The dominant-element pointers are updated next. For each pointer, there are three possibilities. The first possibility is that the element being pointed to by a given window's pointer may have been removed because it is dominated by the new element. Then the pointer is redirected to this new element. The second possibility is that the element being pointed to by a given window's pointer may no longer be in that window after the window has moved to its new position. This possibility happens when, before the update, the dominant element was at the left end of the window. In this case, the potentially significant element list is scanned starting at the old pointer position and moving to the right until an element is found that is in the window in its new position. If the time points are equally spaced, it is only necessary to move one element down the list. Third, if neither of the two preceding circumstances holds, the pointer is left where it was.

For a given window, the maximum and minimum of y over the window in its new position are determined by the elements in the potentially significant extrema lists pointed to by the window's dominant element pointers. From these values, the peak-to-peak excursion of y over the window in this position can be determined. This is compared with the jitter-to-date tally for this window, and if the new peak-to-peak excursion is greater, the tally is updated with this value.

Once the step involving the last point of y has been completed, the desired jitter values are found in the jitter-to-date tallies. This informal algorithmic description is formalized in algorithmic language as follows.

Algorithm 1

Step 0. Initialization. Initialize the lists of potentially significant maxima and minima to $\{y(t_1)\}$. For $j = 1, 2, \ldots, m$, set $y_U^{(w_j)}(t_1) = y_L^{(w_j)}(t_1) = y(t_1)$ and set $y_J^{(w_j)}(t_1) = J(y, w_j) = 0$. Set the iteration counter i to 1, and set all of the dominant-maximum pointers $u_1, \ldots, u_m$ and all the dominant-minimum pointers $l_1, \ldots, l_m$ to t_1.

Step 1. Increment i by 1. If $i > n$, terminate; the desired jitter values are in $J(y, w_1), \ldots, J(y, w_m)$. Otherwise, continue.

Step 2. Start scanning the list of potentially significant maxima from right to left. Each time a scanned element is less than or equal to $y(t_i)$, remove it. Stop the scan the first time an element is encountered that is greater than $y(t_i)$ or when the list becomes empty. Perform the same operation on the list of potentially significant minima, reversing the inequalities.

Step 3. Add $y(t_i)$ to the right end of both lists of potentially significant extrema.

Step 4. For $j = 1, 2, \ldots, m$.

Step 4.1. Update the pointers to the dominant maxima, and calculate $y_U^{(w_j)}(t_i)$: a) if $y(u_j)$ was removed from the list of potentially significant maxima at step 2, set $u_j = t_i$; otherwise, b) if $u_j < t_i - w_j$, then scan the list of potentially significant maxima, starting at $y(u_j)$ and moving to the right, until the first element $y(t_s)$ is found for which $t_s \geq t_i - w_j$, and set $u_j = t_s$; otherwise, c) u_j remains the same.

Then set $y_U^{(w_j)}(t_i) = y(u_j)$.

Step 4.2. Similarly, update the pointers to the dominant minima and calculate $y_L^{(w_j)}(t_i)$.

Step 5. For $j = 1, 2, \ldots, m$, set $y_J^{(w_j)}(t_i) = y_U^{(w_j)}(t_i) - y_L^{(w_j)}(t_i)$ and if $y_J^{(w_j)}(t_i) > J(y, w_j)$, then set $J(y, w_j) = y_J^{(w_j)}(t_i)$.

Step 6. Return to step 1.

Note: If time points are equally spaced, then the bookkeeping in this algorithm can be reduced by using the subscript i instead of the time value t_i for the dominant-element pointers and using the number of points k_j under a window instead of its length in time units w_j in the step 4.1b and 4.2b tests.

Starting the algorithm at $i = 1$ would be wasteful if jitter were being calculated using the easy algorithm. However, in this algorithm, it serves the very useful purpose of properly initializing the lists of potentially significant extrema and the dominant-element pointers, and it simplifies the additional bookkeeping necessary for tracking multiple windows simultaneously.

We conclude this section with some remarks on cost. For the most part, these remarks address the case of equal time increments. The time required to calculate jitter using the easy algorithm for a single window covering k points would appear to be proportional to the number of times an element of the y array is referenced. Thus, the time to calculate jitter using the easy algorithm is proportional to $k(n - k + 1)$. If the time series y has duration T, e.g., if $y(1)$ occurs at time 0 and $y(n)$ occurs at time T, and the window has length (duration) w, then k is the smallest integer such that $k \geq (n - 1)w/T$. So, for fixed values of T and w, the time to calculate jitter using the easy algorithm is a quadratic function of the number of points n used in discretizing the time interval. The coefficients of this quadratic function depend on the length w of the window. The total time needed to calculate jitter for several windows is found by adding up the times for each individual window.

On the other hand, an examination of Algorithm 1 shows that the time to calculate jitter using this algorithm may be bounded by an expression of the form $amn + bn + cm + d$, where a, b, c, and d are absolute constants. The $cm + d$ part of this comes from step 0 of the algorithm and, in any realistic example, is an inconsequential part of the total time. The total time for step 1 is proportional to n. The time in step 2 is spent either in removing elements from lists of potentially significant extrema or in examining an element of one of the lists and discovering that it stops further removal. Because a given element of y can be removed from a list only once during the entire course of the jitter calculation, the removal part takes time proportional to n. Because it only takes one element per step to stop a removal back scan, this part of step 2 also takes time proportional to n, as does step 3. Step 4 is executed $n - 1$ times, so step 4.1 is executed $m(n - 1)$ times. Most of the parts of step 4.1 require an execution time that can be bounded by a fixed amount. The only exception is the part of substep b referring to a scan of elements in a list of potentially significant extrema, and that only when the y points are unequally separated in time. But here, considering the entire jitter calculation, a given element of y can only be scanned once per window per list. Thus, the total time spent on step 4.1 is $O(mn)$. Exactly the same considerations apply to step 4.2. Similarly, the time for step 5 is $O(mn)$ and for step 6 is $O(n)$.

Comparing the two times, it is seen that the timing expression for the easy algorithm is a sum of m quadratics in n and so can be thought of as being $O(mn^2)$, whereas the timing expression for the present algorithm is $O(mn)$. For large n, the potential for savings is great. The extent to which that potential is realized will be detailed in a subsequent section.

Implementation Notes

Two implementation scenarios are considered. In the first, it is assumed that the complete time series y is available to the jitter algorithm at each step in the scan. Thus, the jitter analysis code can keep track of any past information it needs from y by an indirect addressing scheme into the y vector itself. This first scenario produces the faster jitter analysis code. In the second, it is assumed that the points of y are fed to the jitter analysis code one at a time and that the code has complete responsibility for any necessary recall. The emphasis in the latter case will be on managing computer memory to keep memory usage within reasonable bounds.

Whereas programming for speed of execution hardly needs justification, in this era of multimegabyte computer memories, the need for care in memory usage may. Consider, then, a time simulation involving a 1-kHz sample rate extending over a 1000-s duration. Suppose that 50 different outputs must be analyzed for jitter. These numbers are within the bounds of parameters that have arisen in actual applications of this technology. Then, each output time history contains 1,000,001 double-precision numbers, which will be assumed to occupy the Institute of Electrical and Electronics Engineers standard of 8 bytes of computer storage. Taking into account

that all 50 output time histories must be generated and stored at one time (running multiple simulations to generate the output time histories in smaller groups has its own penalty in terms of execution time), it is then seen that just storing these 50 y vectors requires over 381 megabytes of computer memory (recall that computer manufacturers use megabyte to refer to 2^{20} bytes, not 10^6). Although it might be possible to equip a high-end workstation to accommodate this in physical memory, a much more common result of trying to run a problem this size would be that a lot of virtual memory paging would be necessary. This would cause a severe increase in wall-clock time of execution.

Fast Implementation

In this scheme, multiple outputs are analyzed one at a time so that the arrays used in potentially significant extrema processing can be reused for each output. It will be seen that this prevents what would otherwise be a doubling of storage needed (assuming that an integer requires half the storage of a double-precision number) to hold just the output time histories.

The lists of potentially significant extrema are realized by arrays of integers that point to the appropriate elements of the y vector, i.e., an indirect addressing scheme. These arrays must have the same length as the y vector. If y is strictly decreasing in time, then in step 2 of Algorithm 1, no elements of y are ever removed from the list of potentially significant maxima, so it must be big enough to hold them all. An increasing y imposes the same restriction on the list of potentially significant minima. For each list, there is an integer variable containing a pointer to the current active right end of the list and an array of m integers that contains pointers to the place in the list pointing to the dominant elements for each window (in effect, a doubly indirect addressing scheme into the y vector). One of the efficiencies of the algorithm comes from the fact that, once an entry is made into one of these lists of pointers to potentially significant extrema, it is never moved (although it may be overwritten). Step-2 removal of dominated elements is accomplished by redefining the right-end pointer to a point farther to the left. Step 3 is accomplished by incrementing the right-end pointer by 1 and storing the current value of i in the new right end of the list. The step-4.1a decision can be made by comparing the dominant-element pointer with the new right-end pointer, and the step-4.1b decision can be made by comparing the value pointed to by the dominant-element pointer, the current value of i, and the number of points in the relevant window. Similar remarks apply to step 4.2. The information needed to complete step 4 and do step 5 comes from following the dominant-element pointers into the lists of pointers to potentially significant extrema and following those pointers into the y vector.

This scheme has been implemented in Fortran 77 code subroutines. Further code provides an interface to MATLAB. All of this code is compiled into a MATLAB MEX-file, which is a file of compiled binaries that can be called directly from MATLAB. In this form, it is part of the PLATSIM (Version 2.0) software package.[6]

Memory-Conservative Implementation

To minimize memory usage, dynamic memory management such as is available in the C programming language is employed. The scheme must be capable of calculating the jitter in several time series, e.g., multiple output channels, simultaneously.

For each time series, the lists of potentially significant maxima and of potentially significant minima will be maintained using a doubly linked linear list of storage structures. By a storage structure is meant a data storage construct (such as the C programming language data type struct) that can hold several items or fields of data, possibly of dissimilar types. What makes a collection of these structures a doubly linked linear list is its logical organization into a linear ordering, with one being thought of as the first (pictured as the leftmost), possibly followed by a second, a third, and so forth, until a last (the rightmost) is reached. The programming mechanism for this logical ordering is the inclusion of two fields in each structure, one of which contains a pointer to the structure immediately to its left (or a special marker if the structure is at the left end of the list) and the other of which contains a pointer to the structure immediately to its right (or a special marker if the structure is at the right end of the list). Each storage structure must be able to store the dependent value $y(t_i)$ of an element of the time series and also the time t_i or, in the case of equally spaced points, the time index i at which the time series takes on this value.

A first-in, first-out list of these structures will be maintained as a reserve stack to provide new structures as needed and to hold unneeded structures when the information in them becomes obsolete.

For each list of potentially significant extrema, one pointer will keep track of its right end, and for each window an additional pointer will keep track of the leftmost structure in each list that contains information from under this window (these correspond, respectively, to the right-end pointer and the dominant-element pointers of the fast implementation). As each new step in the jitter calculation is taken, each list will be updated using a new data point from one of the time series. Any structures containing information obsoleted by the new value will be moved to the input end of the reserve stack. Structures will be moved from the output end of the reserve stack to the right end of each list of potentially significant extrema to hold the new value. The right-end pointers will be updated to point to the structures just added, and the dominant-element pointers will be updated as necessary.

It is important to realize that, when a structure is spoken of here as having been moved from one list to another, this does not mean that there has actually been a movement of data from one part of physical computer memory to another. What is altered is the values of various pointers that define the position of the structure in one or another of the logical lists.

If there are l time signals being jitter-analyzed, then $2l$ structures must be moved from the reserve stack at each jitter calculation step. Before that step is taken, a check will be made of the number of structures in the reserve stack. If it is fewer than $2l + 1$, an attempt will be made to salvage structures containing unneeded information from the lists of potentially significant extrema. If this does not bring the reserve stack up to at least $2l + 1$ structures, an additional N structures are added to the reserve stack using dynamic memory allocation. The parameter N is $2l$ or larger, so that the stack will contain at least $2l + 1$ structures. The memory-optimal practice would seem to be to add just enough structures to bring the total to $2l + 1$. It is hoped that using the proposed scheme with a larger value of N will reduce overhead by reducing the number of dynamic storage allocation calls necessary. It is only suboptimal from memory usage by at most $N - 1$ structures over the entire run of the jitter calculation. Reducing the number of dynamic storage allocation calls probably has its own storage usage reward as well as reducing execution time. The system must use additional memory to keep track of memory allocated at each call. This additional information will be needed when the system frees up the allocated memory.

This stack management scheme has two important effects.

1) The reserve stack can never become empty. This simplifies the stack maintenance functions.

2) Structures removed from the lists and put back on the reserve stack as a result of Algorithm 1, step 2, are not reused during the current step update process. This means that structures pointed to by dominant-element pointers retain their y information and can be queried even if they have been removed from their list and placed on the stack. The program uses this y information to determine whether such a structure has, in fact, been removed from the list so that the dominant-element pointer can be properly updated.

The scheme to salvage unneeded structures from the lists of potentially significant extrema is based on the following ideas. If a structure on a list of potentially significant extrema contains information that comes from a point in its time series that has been bypassed by the dominant-element pointer of the longest window, then it can be returned to the reserve stack. Such a point can be seen at $i = 1$ in Fig. 2b. As the time series approaches its end, a structure that would need to be kept on a list of potentially significant extrema if the time series were to continue indefinitely need not be kept if the information in it is dominated by that in an earlier structure that will never fall off the left end of its window before the end of the time series is reached. Again, in Fig. 2b, the potentially significant maxima at $i = 7$ and 8 fall into this category. Whenever additional

structures are needed on the reserve stack, any of these structures that can be salvaged will be, and the need for allocating additional storage will be reassessed.

This scheme has been realized in a C code package that integrates jitter analysis with the simulation code that generates the time series and some plot data compression code. Further code provides an interface to MATLAB. All of this code is compiled into a MATLAB MEX-file. In this form, it is part of the PLATSIM (Version 2.0) software package.[6]

Timing Results

Two timing studies are presented. In the first, the times to calculate jitter using the two implementations of Algorithm 1 are compared with each other and with the time to calculate jitter using the easy algorithm. These comparisons are made by calculating jitter in time signals of different lengths. In the second study, experimental verification is sought for the timing formula for Algorithm 1 given in the remarks on cost following the statement of the algorithm.

Comparison of Algorithms and Implementations

The timing studies of the two implementations of the jitter calculation algorithm used data from the EOS AM-1 spacecraft[8] as packaged with the PLATSIM software.[7] In this study, there are seven jitter windows ($m = 7$) of lengths 1, 1.8, 9, 55, 420, 480, and 1000 s.

Two disturbances were used. The disturbance producing the smaller jitter analysis problem is called the MODIS scan mirror. The time history of this disturbance extends over 1000 s at a sample rate of 200 Hz, so $n = 200{,}001$. The same sample rate is used to produce the 27 outputs to be jitter analyzed. This means that when the seven jitter windows are in standard position, they cover 201, 361, 1801, 11,001, 84,001, 96,001, and 200,001 points, respectively.

The disturbance producing the larger jitter analysis problem is called the MOPITT mirror scan. The time history of this disturbance extends over 1200 s at a sample rate of 1000 Hz, so $n = 1{,}200{,}001$. The same sample rate is used to produce the 27 outputs to be jitter analyzed. This means that, when the seven jitter windows are in standard position, they cover 1001, 1801, 9001, 55,001, 420,001, 480,001, and 1,000,001 points, respectively.

To compare execution times of alternative jitter calculations, Algorithm 1 was exercised in both the fast implementation and the memory-conservative implementation, and jitter was also calculated using the easy algorithm. These calculations were made in the context of a PLATSIM time-domain analysis. Timing values were obtained using two timing tools. In the case of the fast implementation and the easy algorithm, the jitter calculation could be isolated from the rest of the code at the MATLAB level, and MATLAB function cputime was used. For all cases, the duration of the total PLATSIM analysis was timed using the UNIX® timing function located, in standard implementations of UNIX, in the file /usr/bin/time. These total times were compared with times found by running the same data through PLATSIM with the jitter calculation option turned off.

The timing studies were run on a Sun SPARC 10/30 workstation. In an attempt to see how precise the timing numbers are, four of the cases involving the smaller jitter analysis problem were rerun five times each on a Sun SPARC 10/512 workstation. Note that, because exactly the same problem was run with exactly the same software five times, exactly the same amount of calculation was done each time; so the CPU time for the calculation should be the same for each run. However, these were both multitasking networked computers. Whereas cputime and /usr/bin/time both purport to measure CPU time for the specific job, they do seem to be influenced by the total load on the computer. Timing runs were made over nights and weekends to try to minimize the effect of other computer workload on the timing measurements, but some multitasking noise, probably from automatically operating system procedures, seems to have affected the timing results.

The cases that were run five times each involved both implementations of Algorithm 1, run both with jitter calculation enabled and with it disabled. Means and standard deviations of total run times for each case were calculated, as were the mean and standard deviation of the isolated jitter calculation time in the case of the fast implementation with jitter enabled. From these calculations, the mean and standard deviation of the time to do jitter-related calculations with the two implementations were calculated. The results are given by stating the standard deviation as a percentage of the jitter calculation time. There was an 11% standard deviation in the time estimate for jitter analysis by the fast implementation, found by taking the difference of the average time for a complete run including jitter calculation and the average time for a complete run excluding jitter calculation. For the memory-conservative implementation, the standard deviation was 41%. Most of this large standard deviation was due to a single data point, which was well separated from the cluster formed by the other four points. By removing this obvious outlier from the data set, the standard deviation was reduced to 14%. The standard deviation in the directly measured jitter analysis time for the fast implementation was 7%.

With such a small sample size, these numbers must be considered to be rough approximations. However, it is probably safe to conclude from them that timing numbers presented subsequently are approximately correct in their most significant digit.

Further softening the sharpness of timing data is their dependence on which compiler is used in compiling the executable modules and what optimization levels are called for. The results given in this paper are based on Fortran code that was compiled using the Sun f77 compiler, version V1.4, with optimization flag-O and on C code that was compiled using the Sun cc compiler, version SC1.0, with optimization flag-O. However, when some of the memory-conservative implementation tests were rerun using binaries compiled using the GNU gcc compiler with optimization flag-O3, execution times generally improved by 15–20%.

Table 1 gives the times to calculate jitter using Algorithm 1 in both of its implementations and also doing the calculation using the easy algorithm. The easy-algorithm test was made using Fortran 77 subroutines that were compiled using the same optimization flags and linked to MATLAB in the same way as in the fast-implementation test.

The estimate used for the time to calculate jitter in the big problem using the easy algorithm was arrived at by running this case with only 1 output signal instead of the 27 used in the rest of the cases and multiplying the resulting jitter calculation time by 27. This estimation was justified experimentally by running the small problem with one, two, and three outputs. The average time per output varied by about 1 part in 2700 over these 3 tests. Theoretical considerations support the hypothesis that time to calculate jitter using the easy algorithm is linear in the number of outputs, because the work to calculate jitter using the easy algorithm depends only on the signal length and the window sizes and is independent of the signal shape. Note that the estimated time to do the full problem exceeds 127 days. On the basis of this estimate, Algorithm 1 is claimed to be enabling technology for large jitter problems.

Even considering the imprecision of timing numbers noted earlier, some conclusions can be drawn. Note that n for the large case is almost exactly six times as large as for the small case. Times for calculating jitter for the large problem using Algorithm 1 are all roughly six times the comparable times for the small problem, supporting the claim that, for a given number of windows, the time to calculate jitter using Algorithm 1 is proportional to the number of points in

Table 1 Jitter calculation execution times

Case	Time, s: MODIS scan mirror ($n = 200{,}001$)	Time, s: MOPITT mirror scan ($n = 1{,}200{,}001$)
Fast implementation, direct measurement	4.8×10^1	3.0×10^2
Fast implementation, difference of full runs	1.2×10^2	7.5×10^2
Memory-conservative implementation, difference of full runs	2.5×10^2	1.9×10^3
Easy algorithm, direct measurement	2.5×10^5	1.1×10^7 [a]
Easy algorithm, difference of full runs	2.5×10^5	1.1×10^7 [a]

[a] Estimated.

the time series and independent of the actual window lengths. The memory-conservative implementation seems to take about twice as long as the fast implementation. The differences between the fast-implementation times measured directly and as a difference of full runs is at least partly explained by overhead in the PLATSIM program related to outputting the jitter results, which was not included in the direct time measurement; but for these two cases, this overhead should have been about the same. Perhaps the discrepancy is due to the imprecision and/or variability of timing measurements.

The improvement over prior technology is clear. For the smaller problem, the speedup using Algorithm 1 exceeds three orders of magnitude, whereas for the larger problem it is more like four orders of magnitude.

The implications of this on turnaround time for a time-domain analysis of the EOS AM-1 spacecraft using PLATSIM are profound. The time-domain analysis consisted of simulating the response of the 27 output signals to the disturbance time history, calculating the jitter in those 27 signals, and performing a data compression on them so that the PostScript files containing the plots of the output time histories could be made reasonably small. This data compression is done to realize a saving of disk space (from many megabytes per file to a few tens of kilobytes) and time to print the hard copies (from about 30 min per file to more like 2 min). The small problem, which took almost three days to analyze using the old technology, was done in less than an hour using the new technology. The fraction of computation time devoted to jitter analysis dropped from almost 99% to about 4% for the fast method and about 7% for the memory-conservative method (note that the easy algorithm for computing jitter does not have a memory-conservative counterpart). For the large problem, the run time decreased from an estimated time of more than 127 days to about 5 h, with jitter analysis time decreasing from over 99.8% of the total to about 4% for the fast method and about 9.5% for the memory-conservative method

Parameter Dependence

Theoretical analysis suggests that the time to compute jitter using Algorithm 1 should be of the form $(am + b)n$, where n is the number of points in the time series, m is the number of windows, and a and b are constants independent of all the problem parameters, specifically including the actual window lengths. (The $cm + d$ part of the timing formula given in the earlier remarks on cost is in the noise level of the computer execution timing instrument and will be ignored in this study.)

A timing study was run to measure how the jitter calculation time using Algorithm 1 depends on n and m and to see whether it is independent of $w_1, \ldots, w_m$ (the individual window lengths). The study involved 3808 individual cases. The time-series length n ranged from 100,001 to 1,200,001 in increments of 100,000. The time history that provided the input data was the MODIS roll output from the EOS AM-1 simulation using the same MOPITT mirror scan disturbance used for the larger jitter analysis problem in the preceding subsection. For $n < 1{,}200{,}001$, the first n points were used. The basic window set was scaled by each of the factors 0.2, 0.6, 1.0, and 1.3; any windows longer than the truncated time series were discarded; and the jitter calculation was done with each subset of the resulting window set. This gave values of m ranging from 1 to 7 and 26 usable window lengths distributed in a fairly uniform exponential manner from 0.2 to 1000 s.

These parameter-dependence timing tests were made using the fast implementation of Algorithm 1. The test data had been previously calculated and saved for use in these tests. Timing studies were run directly from MATLAB and not in the context of a PLATSIM time-domain analysis. Times were measured using MATLAB's cputime function. Three timing runs were made on each of the 3808 test cases.

The relevant data from the ith case are n_i, the number of points in the time series; m_i, the number of windows; and τ_i, the time to calculate jitter. The dependent variable was taken to be z_i, defined to be τ_i / n_i, and the dependence of z_i on m_i was investigated. A scatter plot of the points (m_i, z_i) showed that the data tended to concentrate along a straight line, but a few outliers were observed. Further analysis showed that no case that gave an outlier in one of the three timing runs gave an outlier in either of the other two runs. It was concluded that the outliers resulted from external influences on the timing instrument. This was taken as justification for eliminating the outliers.

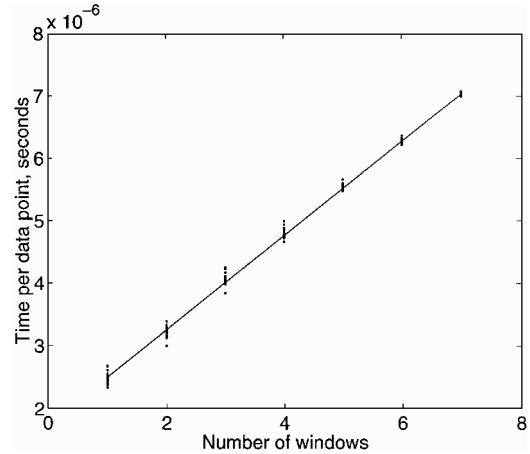


Fig. 3 Linear regression fit to window size timing data.

Outliers were eliminated by taking z_i to be the median value of the ratios τ_i / n_i formed with data from each of the three runs. A scatter plot of (m_i, z_i) now concentrated along a straight line with no wild points.

Using linear least-squares regression on these data, it was found that $z_i \approx am_i + b$, with $a = 7.56 \times 10^{-7}$ s and $b = 1.74 \times 10^{-6}$ s. The root mean square error in this approximation was 2.73×10^{-8} s, which was about 0.6% of the mean z_i value of 4.27×10^{-6} s. This is a tight fit, which experimentally validates the theoretical timing formula. A scatter plot of the median computational time per time-series point as a function of the number of windows is shown in Fig. 3 together with the least-squares linear regression fit line.

Estimating the jitter computation time as $(am + b)n$, it is seen that the time to calculate jitter with respect to one window is about $(a + b)n$, and each additional window adds about an to the time. This means that the time to calculate the jitter with respect to each window after the first is about 30% of the time to calculate it with respect to the first window. The conclusion to be drawn from this is that it is substantially more efficient to calculate jitter over several windows simultaneously than to do the calculations for each window individually.

Memory Conservation Results

In the small problem, $n = 200{,}001$ and $l = 27$. Thus, it takes $200{,}001 \times 27 \times 8$ bytes of memory to store the time histories of the outputs in 8-byte double-precision words. The jitter computation also requires two integer arrays of length n to hold the information for the lists of potentially significant local extrema. Assuming that an integer occupies 4 bytes of storage, and recalling that a megabyte of memory refers to 2^{20} bytes, it is seen that the major data storage part of the memory requirement for the fast implementation of the jitter calculation is almost 43 Mbytes.

The storage structure used in the memory-conservative implementation must hold one double-precision number $y(t_i)$, one integer i, and two pointers. In the implementation of the C programming language used in this study (the standard C compiler cc furnished with the Sun OS 4.x), these individual fields occupy 8, 4, 4, and 4 bytes, respectively. However, the system uses 24 bytes for the complete structure. One can only speculate about those 4 extra bytes. Perhaps it has something to do with aligning structures on word boundaries.

The choice was made to allocate these structures in blocks of 10,000. The small problem required four of these blocks. Thus, the major data storage part of the memory requirement for the memory-conservative implementation of the jitter calculation was $1000 \times 24 \times 4$ bytes, which is less than 1 Mbyte. The memory savings here were almost 97.9%.

The large problem has $n = 1{,}200{,}001$ and $l = 27$. Repeating the preceding calculation with these numbers, it is seen that the major data storage part of the memory requirement for the fast implementation of the jitter calculation is over 256 Mbytes. When it was run with the memory-conservative implementation, nine blocks of structures were allocated, so the major data storage part of the memory requirement for jitter analysis in this run was barely over 2 Mbytes. In this case, the memory savings were over 99.2%.

Note that the memory usage for the fast implementation is completely determined by the length and number of time signals being analyzed but that the memory usage for the memory-conservative implementation also depends on the signal shape and the lengths of the jitter windows. An extremely noisy signal would tend to require less memory, and a signal with long monotone stretches would require more. So, whereas the numbers given here for the memory-conservative implementation are indicative of the savings, results will vary somewhat from case to case.

Conclusions

A fast algorithm for calculating jitter in a time signal has been presented. When applied to large problems typical of those found in spacecraft design analysis, it computes the NASA metric of jitter faster than prior technology by three to four orders of magnitude. In the case of very large problems, this is enabling technology. In any case, it tames the jitter analysis calculation so that, instead of being the overwhelmingly dominant element in the total calculation time of a typical analysis, it is a relatively small part.

Two implementations of the algorithm have been described. The faster one takes advantage of knowledge of the entire time series when this information is available. The memory-conservative version allows the calculation to proceed with much less memory usage, preventing the excessive page swapping and even disk swap-space overflow that can occur for particularly large problems. If jitter must be calculated with respect to several windows, it is much more efficient to calculate the jitter for all of the windows simultaneously than it is to do the calculation for each window independently of the others.

Acknowledgments

This work was supported by NASA Langley Research Center under Contracts NAS1-19000 and NAS1-96014. Thanks are due to P. G. Maghami of NASA Langley Research Center for information on difficulties in calculating the metric in practice and to J. T. Bolek of NASA Goddard Space Flight Center for information on the metric itself.

References

[1] Neste, S. L., "UARS Pointing Error Budget (CDRL 220.04)," GE Space Div., Program Information Release U_1K21_UARS_517, Valley Forge, PA, July 1986.

[2] Ford, T., "EOS Pointing Error Budget, Prediction and Verification Concept," Martin Marietta Astro Space (now Lockheed Martin Missiles and Space), TR EOS-DN-SE&I-043 Rev. A, King of Prussia, PA, Aug. 1993.

[3] Ram, M., and Throckmorton, A., "EOS Instrument Jitter Assessment and Mitigation," AIAA Paper 93-1004, Feb. 1993.

[4] Belvin, W. K., Maghami, P. G., Tanner, C., Kenny, S. P., and Cooley, V., "Evaluation of CSI Enhancements for Jitter Reduction on the EOS AM-1 Observatory," *Dynamics and Control of Large Structures, Proceedings of the Ninth VPI&SU Symposium*, Virginia Polytechnic Inst. and State Univ., Blacksburg, VA, 1993, pp. 255–266.

[5] Ropbeck, L. S., Rathbun, D. B., and Lehman, D. H., "Precision Pointing Control for an Orbital Earth Observing System," *IEEE Control Systems Magazine*, Vol. 11, No. 3, 1991, pp. 46–52.

[6] Maghami, P. G., Kenny, S. P., and Giesy, D. P., "PLATSIM: A Simulation and Analysis Package for Large-Order Flexible Systems," NASA TM-4790 (to appear).

[7] Maghami, P. G., Kenny, S. P., and Giesy, D. P., "PLATSIM: An Efficient Linear Simulation and Analysis Package for Large-Order Flexible Systems," NASA TP-3519, Aug. 1995.

[8] Maghami, P. G., Kenny, S. P., and Giesy, D. P., "The PLATSIM Software Package: A Simulation and Analysis Tool for Large-Order Flexible Systems with Applications to EOS AM-1," *Guidance and Control 1995*, Vol. 88, Advances in the Astronautical Sciences, American Astronautical Society, Univelt, San Diego, CA, 1995, pp. 39–57.

R. B. Malla
Associate Editor