Parallel Algorithms That Locate Local Extrema of a Function of One Variable From Interval Measurement Results

Karen Villaverde, Vladik Kreinovich

***Abstract*— The problem of locating local maxima and minima of a function from approximate measurement results is vital for many physical applications: in spectral analysis, chemical species are identified by locating local maxima of the spectra; in radioastronomy, sources of celestial radio emission and and their subcomponents are identified by locating local maxima of the measured brightness of the radio sky; elementary particles are identified by locating local maxima of the experimental curves. In [17], a sequential algorithm was proposed that solves this problem in linear time (i.e., in time $\leq Cn$, where n is the number of measurements and C is a constant that is independent on n). In this paper, we propose *parallel* algorithms that find local maxima and minima in polylog time ($\leq C\log^2(n)$ on n processors, and $\leq C\log(n)$ on n^2 processors).**

I. Introduction

The problem of locating local maxima and minima of a function $f(x)$ from approximate measurement results is vital for many physical applications (for details, see [17]):

- in *spectral analysis*, chemical species are identified by locating local maxima of the spectra;
- in *radioastronomy*, sources of celestial radio emission and and their subcomponents are identified by locating local maxima of the measured brightness of the radio sky;
- *elementary particles* are identified by locating local maxima of the experimental curves.

In all these cases, we know the results $y_1, ..., y_n$ of measuring the (unknown) function $\bar{f}(x)$ for some values $x_1, ..., x_n$, and we know the accuracy ε of the measuring instrument (i.e., we know that $|\bar{f}(x_i) - y_i| \leq \varepsilon$ for all i).

In some real-life cases, we may have additional information about $\bar{f}$: We may know a *formula* that describes the unknown function $\bar{f}(x)$; e.g., in optical spectroscopy, we often know that the spectrum is a linear combination of several Gaussian functions: $\bar{f}(x) = \sum c_i \exp((x-a_i)^2/\sigma_i)$ (where the parameters c_i, a_i, and σ_i are a priori unknown). In this case, we can find the parameters of this function $\bar{f}$ from the measurement results, and then use the explicit formula for $\bar{f}$ to describe the local extrema. In addition of the accuracy, we may also know *statistical characteristics* of the measurement errors $y_i - \bar{f}(x_i)$. In this case, we can use statistical methods to describe the possible locations of local extrema. In many important cases, however, we do not have any additional information (see, e.g., [1], [15]): we do not know the shape of $\bar{f}(x)$, and we do not know the probabilities of different errors. In these cases, the only information that we have is the values x_i in which $\bar{f}(x)$ was measured, the measured values y_i, and the accuracy ε. The only information we thus have about the value $\bar{f}(x_i)$ of the (unknown) function $\bar{f}(x)$ is that it belongs to an interval $[y_i - \varepsilon, y_i + \varepsilon]$. We can express this fact by saying that we have *interval measurement results.*

For interval measurement results, there are methods that locate local extrema, (see, e.g., [14], [3], [13], [6]), but these methods do not give a guaranteed location of the extrema. In [17], we proposed a linear-time sequential algorithm for locating local extrema, i.e., an algorithm whose running time is bounded by Cn, where n is the number of measurement results. For sequential algorithms, this is as fast as we can get, because we have proved in [17] that every sequential algorithm that finds local extrema must take at least linear time. However, for large n (in practical problems, n can be in thousands), this linear time may still be too long. Since we cannot make it faster on a sequential computer, the natural solution is to make several computers work in parallel. In this paper, we propose parallel algorithms that find local extrema in polylog time ($\leq C\log^2(n)$ on n processors, and $\leq C\log(n)$ on n^2

The authors are with Systems Engineering, Bell Northern Research, P.O. Box 833871 M\S D0-207, Richardson, TX 75083-3871, email villa@bnr.ca (K. Villaverde) and with Computer Science Department, The University of Texas at El Paso, El Paso, TX 79968, email vladik@cs.utep.edu (V. Kreinovich). This work was partially supported by NSF grant No. CDA-9015006 and NASA Research Grant No. 9-757.

processors).

A similar algorithm was first announced in [16].

II. Description of the Problem in Mathematical Terms

Before we describe the algorithms, let us describe the problem in mathematical terms (in these definitions, we are basically following [17]).

Definition 1. Suppose that for some integer n, we are given n real numbers $x_1 < ... < x_n$, n real numbers $y_1, ..., y_n$, and $\varepsilon > 0$. By a *function interval* we mean the set $\mathcal{F}$ of all continuous functions $f(x)$ such that for all $i = 1, ..., n$, $f(x_i) \in I_i = [y_i^-, y_i^+]$, where $y_i^- = y_i - \varepsilon$ and $y_i^+ = y_i + \varepsilon$.

Remark. This definition is a slight modification of the one originally proposed by R. E. Moore (see, e.g., [7], Section 5.1; [8], Section 2.5).

Definition 2. We say that a function $f(x)$ attains a *local maximum* at a point x_0 if there exists an interval $[x^-, x^+]$ such that $x^- < x_0 < x^+$, and $f(x_0) \ge f(x)$ for all $x \in [x^-, x^+]$. Likewise, we say that a function $f(x)$ attains a *local minimum* at a point x_0 if there exists an interval $[x^-, x^+]$ such $x^- < x_0 < x^+$, and $f(x_0) \le f(x)$ for all $x \in [x^-, x^+]$.

Definition 3. Suppose that a function interval $\mathcal{F}$ is given. We say that an interval I *contains a local maximum* of $\mathcal{F}$ if all functions $f \in \mathcal{F}$ attain a local maximum at some point from I. We say that an interval I *contains a local minimum* of $\mathcal{F}$ if all functions $f \in \mathcal{F}$ attain a local minimum at some point from I.

Main Problem: *to generate intervals $I_1, ..., I_k$ that contain local maxima and local minima.*

Remarks.

- There exist various algorithms that locate the *global maxima* of an intervally defined function (see, e.g., [8], [2], [12], [9]). However, the *input* for these methods *is* very *different*: namely, an expression for the function. Besides, for these algorithms, *local maxima are the* main *obstacle* that has to be overcome (and *not the* desired *result*). For these two reasons, we cannot apply these algorithms to locate all *local* maxima.
- Evidently, if I contains a local maximum, then any bigger interval $J \supset I$ also contains it. We want to find the *smallest* possible locations I. In other words, we want an *optimal* interval estimate in the sense of [10] and [11] (see also [5]). Let's describe this demand in mathematical terms.

Definition 4. Suppose that a function interval $\mathcal{F}$ is given. We say that intervals I and J *contain the same local maximum* if there exists an interval K that contains a local maximum and such that $K \subseteq I$ and $K \subseteq J$. We say that a list $I_1, ..., I_k$ *locates all local maxima* if any other interval I that contains a local maximum, contains the same interval as one of these I_i.

Definition 5. Suppose that a function interval $\mathcal{F}$ is given. We say that an interval I *locates the local maximum* (precisely), if I contains a local maximum, and no proper subinterval $I' \subset I$ contain it.

Similar definitions can be repeated for *local minima.* Taking this into consideration, we can reformulate the main problem as follows:

Main Problem: *to locate all local maxima and local minima precisely.*

III. Auxiliary Result: Testing Whether an Approximately Given Function Can Be Monotonic

Before locating local extrema, it is necessary to know whether the unknown function has any local extrema at all or it is monotonic. In other words, it is necessary to solve the following problem: *given a function interval $\mathcal{F}$, is it possible that a function $f \in \mathcal{F}$ is monotonic?* This problem is sometimes of a separate interest for physical applications. Its solution is also used as a method to accelerate the global optimization algorithms (see, e.g., [12], Section 3.11). A linear-time algorithm for solving this problem was given in [17].

THEOREM 1. *Suppose that we have p processors working in parallel. Then, there exists an algorithm that given a function interval $\mathcal{F}$ of size n, returns "yes" if and only if this interval contains a monotone non-decreasing function. This algorithm requires $C(n/p) + C\log(p)$ computation time.*

Comments

1) For readers' convenience, all the proofs are moved to the last section.
2) In particular, if we have n processors, we get the following result:

COROLLARY 1. *There exists an algorithm that given a function interval $\mathcal{F}$ of size n, returns "yes" if and only if this interval contains a monotone non-decreasing function. This algorithm uses n processors and requires $C\log(n)$ computation time.*

For *non-increasing* functions, the results are similar:

THEOREM 2. *Suppose that we have p processors working in parallel. Then, there exists an algorithm that given a function interval $\mathcal{F}$ of size n, returns "yes" if and only if this interval contains a* *monotone non-increasing function. This algorithm requires $C(n/p) + C\log(p)$ computation time.*

COROLLARY 2. *There exists an algorithm that given a function interval $\mathcal{F}$ of size n, returns "yes" if and only if this interval contains a monotone non-increasing function. This algorithm uses n processors and requires $C\log(n)$ computation time.*

IV. Main Results

THEOREM 3. *There exists an algorithm that uses n^2 processors and locates all local maxima and all local minima precisely in time $\le C\log(n)$ (for some constant C).*

Comment. We can prove that for a certain class of algorithms, $\approx n^2$ processors is a lower bound. Indeed, according to [17], an interval (x_k, x_l) is a possible location of the local maximum iff there exists an i such that $k < i < l$ such that $y_k < y_i - 2\varepsilon$, $y_l < y_i - 2\varepsilon$, and for all j from $k+1$ to $l-1$, $y_j \ge y_i - 2\varepsilon$ and $y_i \ge y_j - 2\varepsilon$. These inequalities make perfect sense if we take into consideration the fact that the actual value $\bar{f}(x_i)$ of the function $\bar{f}(x)$ for $x = x_i$ can be ε−different from y_i, so:

- the inequality $y_k < y_i - 2\varepsilon$ is the one that guarantees that $\bar{f}(x_k) < \bar{f}(x_i)$, and
- the inequality $y_k \ge y_i - 2\varepsilon$ is equivalent to the possibility that $\bar{f}(x_k) \ge \bar{f}(x_i)$.

So, in principle, in order to find the local maxima, we do not need to know the actual values of y_i: it is sufficient to know whether $y_i \ge y_j - 2\varepsilon$ or not for different i and j. It turns out that if we allow only this checking, then we do need at least $\approx Cn^2$ processors:

PROPOSITION 1. *There exists an algorithm that locates all local maxima and all local minima of a function, and whose only access to the data y_i is a possibility to check, for any pair (i,j), whether $y_i < y_j - 2\varepsilon$.*

PROPOSITION 2. *Every algorithm that locates all local maxima and all local minima of a function, and whose only access to the data y_i is a possibility to check, for any pair (i,j), whether $y_i < y_j - 2\varepsilon$, requires at least $n^2/4$ total computation steps.*

COROLLARY 3. *Every algorithm that locates all local maxima and all local minima of a function in time $\le O(\log^p(n))$, and whose only access to the data y_i is a possibility to check, for any pair (i,j), whether $y_i < y_j - 2\varepsilon$, requires at least $C \cdot n^2/\log^p(n)$ processors.*

Comment. If we do not impose this restriction on the algorithm, then we can locate the local extrema by using n processors:

THEOREM 4. *There exists an algorithm that uses n processors and locates all local maxima and all local minima precisely in time $\le C\log^2(n)$ (for some constant C).*

V. Proofs

The proofs will be given in the following order: Theorems 1, 2, 4, 3, and then, Propositions 1 and 2.

A. Proof of Theorem 1

According to [17], a function interval contains a monotone non-decreasing function iff for every $i < j$, we have $y_j \ge y_i - 2\varepsilon$. Since we have p processors, we can divide the values $y_1, \dots, y_n$ into p groups: from 1 to $r = \lceil n/p \rceil$, from $r+1$ to $2r$, ..., from $(p-1)r+1$ to n. To each group, we can apply a linear-time algorithm described in [17], and check (in time $\le C(n/p)$) whether $y_j \ge y_i - 2\varepsilon$ for all $i < j$ that belong to one of the intervals $(1,r)$, $(r+1, 2r)$, ..., $((p-1)r+1, n)$. If one of these checks reveals that there is a non-monotonicity, then the entire function is not monotonic. If, however, all the checks are positive, then in order to check whether the given function interval contains a non-decreasing function, we must check whether the inequality $y_j \ge y_i - 2\varepsilon$ is true for all $i < j$ that belong to different intervals $(1,r)$, $(r+1,2r)$, ..., $((p-1)r+1, n)$.

The inequality $y_j \ge y_i - 2\varepsilon$ is true for all $i \in A$ and $j \in B$ iff it is true for the smallest of y_j and for the largest of y_i, i.e., if the inequality

$$\min_{j \in B} y_j \ge \max_{i \in A} y_i - 2\varepsilon$$

holds. So, to check this inequality, we must compute for each interval $(1,r)$, $(r+1,2r)$, ..., $((p-1)r+1, n)$, the maximum and the minimum over this interval. Computing the maximum and minimum requires linear time $Cr = C(n/p)$. Therefore, computing these maxima and minima in parallel will take $\le C(n/p)$ time.

As a result of these computations, we have p minima $m_1, \dots, m_p$ and p maxima $M_1, \dots, M_p$. To verify that a given function interval contains a non-decreasing function, we must check that whenever $i < j$, we have $m_j \ge M_i - 2\varepsilon$. We can perform this checking as follows:

- First, we divide p values into $p/2$ pairs: 1 and 2, 3 and 4, ... For each pair $(i, i+1)$, we check whether $m_{i+1} \ge M_i - 2\varepsilon$. If all these checks return "yes", then we have thus verified the inequality $y_j \ge y_i - 2\varepsilon$ for all $i < j$ from the *double* intervals $(1, 2r)$, $(2r+1, 4r)$, ... As a result, instead of p intervals, we now have $p/2$ double intervals, on each of which monotonicity condition is checked.
- To continue checking for monotonicity, we must now check that the $y_j \ge y_i - 2\varepsilon$ is true for i

and j from different double intervals. This is equivalent to checking that

$$\min_{j \in B} y_j \ge \max_{i \in A} y_i - 2\varepsilon$$

is true for double intervals A and B. To verify that inequality, we must compute the maximum and minimum over each double interval. This is easy to do: the maximum over a double interval is simply the maximum of the two values: the maximum over the first subinterval, and the maximum over the second subinterval. Similarly, it is easy to compute the minimum over the double subintervals. The time for computing min and max for each double interval is equal to the time necessary for two elementary operations (computing one max and one min).

- After that, we have $p/2$ double intervals. For each of these double intervals, we have already computed the maximum $\tilde{M}_a$ and the minimum $\tilde{m}_a$. Now, we must check that if $a < b$, then $\tilde{m}_b \ge \tilde{M}_a - 2\varepsilon$. This problem is similar to the one that we started with, with the only difference that we now have $p/2$ pairs of values $(\tilde{m}_a, \tilde{M}_a)$ instead of p pairs. So, we can apply a similar approach: grouping the double-intervals into $p/4$ double-double intervals. This doubling will result in $p/4$ values, to which we will again apply doubling, etc. After each iteration, we will have checked the desired inequality $y_j \ge y_i - 2\varepsilon$ for all $i < j$ that belong to one merged interval. After $\approx \log(p)$ iterations, we will combine all the intervals into one. Therefore, after the last iteration, we will have checked the desired inequality for all $i < j$.

This procedure requires $\approx \log(p)$ iterations, and constant time on each iteration. Therefore, the time required for checking is $\le C\log(p)$. So, the total computation time of the algorithm is $\le C(n/p) + C\log(p)$. Q.E.D.

B. Proof of Theorem 2

The algorithm is similar, with the only difference that for $i < j$, instead of $y_j \ge y_i - 2\varepsilon$, we must check $y_i \ge y_j - 2\varepsilon$.

C. Proof of Theorem 4

We will describe an algorithm that locates local maxima. Local minima can be located similarly. According to [17], to locate local maxima, it is sufficient to find the largest values y_i that are located inside the corresponding local maxima intervals. As we have proved in [17], these values can be characterized by the following condition: for some $k < i < l$, $y_k < y_i - 2\varepsilon$, $y_l < y_i - 2\varepsilon$, and for all intermediate j (i.e., $k < j < l$), $y_i - 2\varepsilon \le y_j \le y_i$. The resulting interval (x_k, x_l) locates the local maximum precisely. In our parallel algorithm, we will follow the similar idea: find i's and the corresponding k's and l's.

In our algorithm, we will use n processors to handle n values $y_1, ..., y_n$ (i−th processor will handle i−th value). Before we start data processing, the only values x_i that we can immediately rule out as potential locations of the local maxima are the endpoints (x_1 and x_n). So, we can say that all other values x_i are potential locations of the largest values inside the local maximum intervals. We will apply the iterative "sieve" to delete those values x_i that are not, and eventually, we will only have the desired locations. On each iteration, we will have some locations marked as *desired* ones, and some other locations marked as *candidates* for the desired locations. Initially, none are marked as desired, and all n are marked as candidates. On each iteration, the number of candidates will decrease at least in half. Therefore, the required number of iterations will be $\le \log(n)$.

Let us describe each iteration. In the beginning of each iteration, we have some values y_i marked as candidates, and some values marked as desired (with the corresponding k and l in place). Let us denote the locations of these candidates, of the desired values, and of the values 1 and n (that will be used for comparison), by $i_1 < i_2 < ... < i_c$ (here, c denotes the total number of such locations; so, $i_1 = 1$, and $i_c = n$). On each iteration, we will only compare each candidate y_{i_p} with its immediate neighbors in this list (i.e., with $y_{i_{p-1}}$ and $y_{i_{p+1}}$), and with the values that are in between this candidate and its immediate neighbors in this list. Depending on the results of these comparisons, we have seven possibilities:

- If there exist a k such that $i_{p-1} \le k < i_p$ and $y_k < y_{i_p} - 2\varepsilon$, and an l such that $i_p < l \le i_{p+1}$ and $y_l < y_{i_p} - 2\varepsilon$, then the largest of such k and the smallest of such l form the desired location of one of the local maxima. This i_p will then be marked as desired, and the corresponding interval (x_k, x_l) outputted.
- If for all k from i_{p-1} to i_p, we have $y_k \ge y_{i_p} - 2\varepsilon$, i_{p-1} was a candidate, and $y_{i_p} > y_{i_{p-1}}$, then i_{p-1} is no longer a candidate (because it must belong to the same interval of local maximum as i_p, and the value y_{i_p} is larger than the value $y_{i_{p-1}}$ for the former candidate i_{p-1}).
- If for all k from i_{p-1} to i_p, we have $y_k \ge y_{i_p} - 2\varepsilon$, and $y_{i_p} \le y_{i_{p-1}}$, then i_p is no longer a candidate

Indeed, a candidate must be $\ge$ than all the values in its vicinity, until we arrive at really smaller values (i.e., smaller than the value of the candidate -2ε). Here, for i_p, none of the points from i_p to i_{p-1} are "really smaller" in this sense, so i_{p-1} should also belong to what we called the "vicinity". If $y_{i_{p-1}} > y_{i_p}$, then we have a violation of the condition that y_{i_p} should be the maximum in its vicinity. If $y_{i_{p-1}} = y_{i_p}$, then the indices i_{p-1} and i_p describe the same local maximum (if they describe the maximum at all), so, we will only keep the smaller one i_{p-1} as a candidate.

- If for all k from i_{p-1} to i_p, we have $y_k \geq y_{i_p} - 2\varepsilon$, and $i_{p-1} = 1$ (i.e., $p - 1 = 1$ and $p = 2$), then i_p is no longer a candidate (because for a point i_p to be a location of the local maximum, it must have a preceding point x_k in which $y_k < y_{i_p} - 2\varepsilon$.
- Similarly, if for all l from i_p to i_{p+1}, we have $y_l \geq y_{i_p} - 2\varepsilon$, i_{p+1} was a candidate, and $y_{i_{p+1}} \leq y_{i_p}$, then i_{p+1} is no longer a candidate.
- If for all l from i_p to i_{p+1}, we have $y_l \geq y_{i_p} - 2\varepsilon$, and $y_{i_{p+1}} > y_{i_p}$, then i_p is no longer a candidate.
- If for all l from i_p to i_{p+1}, we have $y_l \geq y_{i_p} - 2\varepsilon$, and $i_{p+1} = n$ (i.e., $p + 1 = c$ and $p = c - 1$), then i_p is no longer a candidate (because for a point i_p to be a location of the local maximum, it must have a following point x_l in which $y_l < y_{i_p} - 2\varepsilon$).

On each iteration, out of each pair of neighboring candidates, either one of them becomes a desired location (and is thus no longer a candidate), or one of them stops being a candidate. In both cases, after each iteration, out of each pair of candidates, no more than one candidate survives.

If a candidate's immediate neighbors in the list $i_1 < \ldots < i_c$ are not candidates, then after the iteration, this candidate either becomes a desired value, or is dropped from this list as a non-candidate. In both cases, it stops being a candidate.

So, the total number of candidates is decreased at least in half.

To describe each iteration, we must explain how this checking of the conditions on k and l is performed. Let us describe checking for k (checking for l is similar).

- For every k from i_{p-1} to i_p, we set a variable v_k to be equal to k if $y_k < y_{i_p} - 2\varepsilon$, and to 0 otherwise. Then, we compute the maximum of all these values v_k. It is well known how to compute maximum of N numbers on N processors in $\approx \log(N)$ steps [4]:
  - on the first step, we divide N numbers into $N/2$ pairs, and compute the maximum of each pair; thus, we get $N/2$ results;
  - on the next step, we divide these $N/2$ results into $N/4$ pairs, and for each pair, find the largest of the corresponding maxima; this will give us $N/4$ results, each of them is the maximum of four consequent values;
  - the same "bisection" is to be repeated again and again, so that we are left with $N/8$, $N/16$, ..., $N/2^s$, ... values. After $s \approx \log(N)$ steps, we get a single value that is equal to the desired maximum.

  To make this algorithm clear, let us give an example. Suppose that we start with 8 values $v_1, \ldots, v_8$. Then, on each step, we will end up with computing the values $M_{i,j} = \max(v_i, v_{i+1}, \ldots, v_j)$ until we compute the desired maximum.

  - First, we divide eight numbers into four pairs, and compute four values $M_{1,2} = \max(v_1, v_2)$, $M_{3,4} = \max(v_3, v_4)$, $M_{5,6} = \max(v_5, v_6)$, $M_{7,8} = \max(v_7, v_8)$ on 4 different processors in parallel.
  - On the second step, we divide the resulting four values into two pairs, and compute $M_{1,4} = \max(M_{1,2}, M_{3,4})$ and $M_{5,8} = \max(M_{5,6}, M_{7,8})$ on two processors in parallel.
  - On a third step, we are left with only one pair, so, we can compute the desired result $M_{1,8} = \max(M_{1,4}, M_{5,8})$.
- If the maximum of v_k is equal to 0, this means that in between i_{p-1} and i_p, there are no k for which $y_k < y_{i_p} - 2\varepsilon$. If this maximum m is not zero, this means that such k exist, and m is the largest of these values.

To check for l, we must compute the *minimum* of *non-zero* values of v_k instead of the *maximum* of *all* v_k.

By applying this method, we can tell which of the seven above-described cases we have. On each iteration, processing one candidate requires $\log(N) \leq \log(n)$ steps. On each iteration of this algorithm, each processor i that lies in between i_{p-1} and i_p is only used twice: for checking k for i_p, and for checking l for i_{p-1}. So, the total running time for each processor on each iteration is $\leq 2\log(n)$.

Totally, there are $\leq \log(n)$ iterations, and each (as we have just shown) requires $\leq 2\log(n)$ steps. Therefore, the total computation time is $\leq 2\log^2(n)$ steps on n processors. Q.E.D.

Example. Let us trace this algorithm on a following "wave-like" example: $n = 11$, $\varepsilon = 1$, $y_1 = -1$,

$y_2 = 0$, $y_3 = 1$, $y_4 = 2$, $y_5 = 1$, $y_6 = 0$, $y_7 = -1$, $y_8 = 0$, $y_9 = 1$, $y_{10} = 2$, $y_{11} = 1$.

- Initially, all 11 points are candidates for a maximum so, $c = 11$, $i_1 = 1, ..., i_{11} = 11$. There are no intermediate points between i_{p-1} and i_p, so the only reason for dismissing a candidate i_p would be if $y_{i_p} < y_{i_{p+1}}$ or $y_{i_p} < y_{i_{p-1}}$. These conditions enable us to delete all the points except for $i = 4$ and $i = 10$ from the list of candidates. So, after the first iteration, only two candidates remain: $i = 4$ and $i = 10$.
- On the second iteration, the list of $i_1, ...$ must contain the desired values (none yet), candidates (only two remain), and the end points. So, we have $c = 4$ points: $i_1 = 1$, $i_2 = 4$, $i_3 = 10$, and $i_4 = 11$, out of which two are candidates: 4 and 10. For both candidates, we check for k and for l:
  - For $i_2 = 4$, checking for k leads to $v_1 = 1$ (since $y_1 = -1 < y_4 - 2 = 2 - 2 = 0$), $v_2 = 0$ (because $y_2 = 0 \geq y_4 - 2 = 0$), $v_3 = v_4 = 0$. Therefore, $\max(v_k) = 1 \neq 0$.
  - For $i_2 = 4$, checking for l leads to $v_4 = v_5 = v_6 = 0$, $v_7 = 7$, $v_8 = v_9 = v_{10} = 0$. Here, the only non-zero value is 7, so, the smallest of these non-zero values is also equal to 7. Hence, $l = 7$, and we have got one desired value $i = 4$, and one maximum interval (x_1, x_7).
  - For $i_3 = 10$, checking for k leads to $v_7 = 7$ and $v_i = 0$ for $i \neq 7$ so, $\max v_k = 7$.
  - For $i_3 = 10$, $1 = y_{11} \geq y_{10} - 2\varepsilon = 2 - 2 = 0$, so there are no l with the desired property, and i_4 is the endpoint. Hence, i_3 is no longer a candidate.
- No more candidates are left, so, the process is completed. We have located the only local maximum interval: (x_1, x_7).

Similarly, we can locate the only interval for local minimum: (x_4, x_{10}).

D. *Proof of Theorem 3*

We will describe an algorithm that locates local maxima. Local minima can be located similarly.

In this algorithm, we will use n^2 processors. Each processor will correspond to a pair (i, j), $1 \leq i, j \leq n$. This algorithm consists of the following stages:

- On the first stage, for all $i < j$, we compute $M_{i,j} = \max(y_i, y_{i+1}, ..., y_j)$ and $m_{i,j} = \min(y_i, y_{i+1}, ..., y_j)$. Let us explain how the values $M_{i,j}$ are computed (the values $m_{i,j}$ are computed in a similar manner, with min instead of max). This will be done as follows:
  - On the first sub-stage of this stage, we compute $\max(y_1, ..., y_n)$ using an algorithm described in the proof of Theorem 4. This algorithm requires $\log(n)$ time on n processors. So, we can use processors (i, i), $1 \leq i \leq n$. During these computations, we will compute the values $M_{i,j}$ for all pairs i, j for which $i = p \cdot 2^q + 1$ and $j = (p+1) \cdot 2^q$ for some integers p and q.
  - Then, each processor (i, j) will compute the value $M_{i,j}$ as follows: We represent the interval between i and j as the union of the intervals for which $M_{i,j}$ has already been computed on the first sub-stage, and compute $M_{i,j}$ as the maximum of all these computed values. Suppose that $n = 2^d$ (if $n \neq 2^d$ for any integer d, then we take the smallest d for which $2^d \geq n$). To get the desired representation as a union, we first check whether the interval (i, j) contains the entire interval $1, 2^d = n$. If it does, the problem is solved. If it does not, we check if the desired interval (i, j) contains one of the intervals of size 2^{d-1} ($(1, 2^{d-1})$ and $(2^{d-1} + 1, 2^d)$), etc. If it does not, then we go to the next iteration. If it does, then on the next iteration, we represent the remainder. This remainder consists of at most two intervals; the left one has a right end divisible by 2^{d-1}, and the right one has a left end equal to $1 + p \cdot 2^{d-1}$ for some integer p. Therefore, on the next iteration, these endpoints will coincide with the endpoints of pre-computed intervals, and so, for each of the remaining intervals, only one remainder will be formed. As a result, on each iteration, we have at most 2 remaining intervals. The number of iterations is $d = \log(n)$, therefore, we need $\leq C \log(n)$ steps to compute the values $M_{i,j}$. Similarly, we need $\leq C \log(n)$ steps to compute the values of $m_{i,j}$. Hence, this stage requires time $\leq C \log(n)$.

To clarify this algorithm, let us trace it on the following example: assume that $n = 16$, and we want to compute the value $M_{2,9}$. Here, values computed on the first stage are: $M_{i,i} = y_i$ for all i, $M_{1,2}$, $M_{3,4}$, $M_{5,6}$, $M_{7,8}$, $M_{9,10}$, $M_{11,12}$, $M_{13,14}$, $M_{15,16}$, $M_{1,4}$, $M_{5,8}$, $M_{9,12}$, $M_{13,16}$, $M_{1,8}$, $M_{9,16}$, $M_{1,16}$, and $d = 4$. On the first iteration, we check whether the interval $(1, 16)$ is contained in the given interval $(2, 9) = \{2, 3, 4, ..., 8, 9\}$. It is not, so we go to the next iteration, on which we check whether intervals $(1,8)$ or $(9,16)$ are contained in $(2,9)$. Again, the answer is "no". Next, we check intervals of length 4, and we find an interval $(5,8)$ that is contained in $(2,9)$. The difference between $(2,9)$ and $(5,8)$ consists of two disjoint intervals $(2,4)$ and $(9,9)$. On the next step, we check whether any of the remaining intervals contains one of the precomputed intervals of length 2: it does, we have $(3,4)$, and the remainder is $(2,2)$. As a result, we represent $(2,9)$ as the union $(2,9)=(2,2)\cup(3,4)\cup(5,8)\cup(9,9)$. Therefore, $M_{2,9}=\max(M_{2,2},M_{3,4},M_{5,8},M_{9,9})$.

- On the second stage, each processor (i,k) with $i>k$ checks whether the following three conditions are true:

  - $y_k < y_i - 2\varepsilon$,
  - $y_i \ge y_j$ for all j from k to i, and
  - $y_j \ge y_i - 2\varepsilon$ for all $j=k+1,...,i-1$.

  This checking can be simplified if we take into consideration the following two facts:

  - The second condition is equivalent to
  $$y_i \ge \max(y_k, y_{k+1},...,y_i) = M_{k,i}.$$
  - The third condition is equivalent to
  $$y_i - 2\varepsilon \le \min(y_{k+1},...,y_{i-1}) = m_{k+1,i-1}.$$

  Since we have already computed the values $m_{i,j}$ and $M_{i,j}$ on the first stage, this checking is done in two steps:

  - checking whether $y_k < y_i - 2\varepsilon$;
  - checking whether $y_i \ge M_{k,i}$;
  - checking whether $y_i - 2\varepsilon \le m_{k+1,i-1}$.

  If all three conditions are satisfied, then the value k is sent to the processor (i,i). The only case when the processor (i,i) gets a number $k>i$ is when k is the lower bound of the local maximum location interval that contains i. Since there can be only one such interval, each processor (i,i) can receive only one such number.

  Similarly, each processor (i,l) with $i<l$ checks whether the following three conditions are true:

  - $y_i - 2\varepsilon > y_l$;
  - $y_i \ge y_j$ for all j from i to l; and
  - $y_j \ge y_i - 2\varepsilon$ for all $j=i+1,...,l-1$.

  The second condition is equivalent to
  $$y_i \ge \max(y_i, y_{i+1},...,y_l) = M_{i,l}.$$
  The third condition is equivalent to
  $$y_i - 2\varepsilon \le \min(y_{i+1},...,y_{l-1}) = m_{i+1,l-1}.$$
  So, since we have already computed the values $m_{i,j}$ and $M_{i,j}$ on the first stage, this checking is done in three steps:

  - checking whether $y_l < y_i - 2\varepsilon$;
  - checking whether $y_i \ge M_{i,l}$; and
  - checking whether $y_i - 2\varepsilon \le m_{i+1,l-1}$.

  If all three conditions are satisfied, then the value l is sent to the processor (i,i).

- Finally, if a processor (i,i) has received both values k $(<i)$ and l $(>i)$, this means that the interval (x_k,x_l) is the desired (smallest) location of the local maximum. So, the pair (k,l) is sent to the processor (k,l) (the processor (k,l) may receive several identical pairs (k,l) from several processors i). The processors (k,l) that have received such pairs output the intervals (x_k,x_l) to the user.

The last two stages require constant number of steps for each processor; therefore, they require constant time. Hence, the total computation time is $\le C\log(n)$ for some C. Q.E.D.

E. Proof of Proposition 1

This Proposition is an immediate corollary of the above-mentioned result from [17]: we can simply check whether $y_i > y_j - 2\varepsilon$ for all i and j, and then find all the triples (i,k,l) with the desired (above-described) property.

F. Proof of Proposition 2

Let us first show that if for the values $y_1 = ... = y_n = 0$, an algorithm $\mathcal{U}$ misses (i.e., does not check) two pairs (k,i) and (l,i) with $k<i<l$, then this algorithm does not always locate the local maxima correctly. Indeed, the correct answer for this particular set of values is "no local maxima", because all comparisons lead to "no" (i.e., to $y_i \ge y_j - 2\varepsilon$). However, let us consider another set of values $z_1,...,z_n$ that is defined as follows: $z_k = z_l = -(3/2)\varepsilon$, $z_i = (3/2)\varepsilon$, and $z_j = 0$ for all other j ($j \ne i$, $j \ne k$, and $j \ne l$). For any pair (p,q) other than (k,i) or (l,i), the answer to the check is still "no" ($y_p \ge y_q - 2\varepsilon$). Since the algorithm $\mathcal{U}$ only checks these pairs, it will return the same answer as for the sequence $y_1,...,y_n$: "no local maxima". However, one can easily check that for $z_1,...,z_n$, there *is* a

local maximum: its precise location is the interval (x_k, x_l).

This argument shows that if an algorithm returns the precise locations of all local maxima, then it must check at least one pair ((k,i) or (l,i)) of each triple $k < i < l$. Let's fix an i that is different from 1 and n. If we do not check (l,i) for at least one $l > i$, then we have to check (k,i) for all $i-1$ values $k < i$. Therefore, for each i, we have one of the two possibilities:

- We check (l,i) for all $l > i$; this means $n-i$ checks.
- We do not check (l,i) for some $l > i$. Then, we have to run $i-1$ checks for of (k,i) for all $k < i$.

In the first case, we need $\ge n-i$ checks. In the second case, we need $\ge i-1$ checks. In both cases, we need $\ge \min(i-1, n-i)$ checks. Totally, for all i, we thus need at least

$$S = \sum_{i=2}^{n-1} \min(i-1, n-i)$$

checks (and since each check requires at least one computation step, we need at least as many computation steps). Let us estimate this sum. The inequality $i-1 \le n-i$ is equivalent to $i \le (n+1)/2$. Therefore, this sum can be divided into two partial sums:

$$S = \sum_{2 \le i \le (n+1)/2} (i-1) + \sum_{(n+1)/2 < i \le n-1} (n-i).$$

The first sum is $1 + 2 + \ldots + p$, where

$$p = \lfloor (n+1)/2 \rfloor - 1.$$

Therefore, this first sum is equal to $p(p+1)/2$, and since $p \sim n/2$, we have $p(p+1)/2 \sim n^2/8$. Similarly, the second sum is equal to $\sim n^2/8$, so, the original sum $S \sim n^2/4$. Hence, we need at least $\sim n^2/4$ computation steps to locate all local maxima precisely. Q.E.D.

References

[1] P. Bloomfield, **Fourier analysis of time series: an introduction**, Wiley, N.Y., 1976.

[2] J. E. Dennis, R. B. Schnabel, **Numerical methods for unconstrained optimization and nonlinear equations**, Prentice-Hall, Englewood Cliffs, 1983.

[3] I. J. Good and R. A. Gaskins, "Density estimation and bump-hunting by the penalized likelihood method exemplified by scattering and meteorite data", *J. Amer. Stat. Soc.*, 1980, Vol. 75, pp. 42–56.

[4] J. Jájá, **An introduction to parallel algorithms**, Addison-Wesley, Reading, MA, 1992.

[5] H. Kolacz, "On the optimality of inclusion algorithms," In: K. Nickel (editor), **Interval Mathematics 1985**, Lecture Notes in Computer Science, Vol. 212, Springer-Verlag, Berlin, Heidelberg, N.Y., 1986, pp. 67–79.

[6] R. D. Martin and D. J. Thompson, "Robust-resistant spectrum estimation", *Proceedings IEEE*, 1982, Vol. 70, pp. 1097–1115.

[7] R. E. Moore, **Mathematical elements of scientific computing**, Holt, Rinehart and Winston, N.Y., 1975.

[8] R. E. Moore, **Methods and applications of interval analysis**, SIAM, Philadelphia, 1979.

[9] R. E. Moore, "Global optimization to prescribed accuracy", *Computers and Mathematical Applications*, 1991, Vol. 21, No. 6/7, pp. 25–39.

[10] L. B. Rall, "Optimization of interval computations", In: K. Nickel (ed.), **Interval Mathematics 1980,** Academic Press, N.Y., 1980, pp. 489–498.

[11] H. Ratschek, J. Rokne, "Optimality of the centered form", In: K. Nickel (ed.), **Interval Mathematics 1980,** Acad. Press, N.Y., 1980, pp. 499–508.

[12] H. Ratschek, J. Rokne, **New computer methods for global optimization**, Ellis Horwood, Chicester, 1988.

[13] B. W. Silverman, **On a test for multimodality based on kernel density estimates**, Technical Summary Report No. 21–81, U. S. Army Math Research Center, Madison, WI, February 1981.

[14] D. J. Thompson, "Spectrum estimation techniques for characterization and development of WT4 Waveguide", *Bell Syst. Technical Journal*, Part 1, November 1977, Vol. 56, p. 1769; Part 2, December 1977, Vol. 57, p. 1983.

[15] D. J. Thompson, "Spectrum estimation and harmonic analysis", *Proceedings IEEE*, 1982, Vol. 70, pp. 1055–1096.

[16] K. Villaverde, "How to locate maxima and minima of a function in parallel from approximate measurement results", In: V. Kreinovich, B. Traylor, R. Watson (eds.), *Abstracts of the First UTEP Computer Science Department Students Conference*, El Paso, TX, 1991, pp. 43–44.

[17] K. Villaverde and V. Kreinovich, "A linear-time algorithm that locates local extrema of a function of one variable from interval measurement results," *Interval Computations*, 1993, No. 4, pp. 176–194.