Japan J. Indust. Appl. Math., 26 (2009), 145–167 Area ⟨2⟩

Application of the Krawczyk–Moore–Jones Algorithm to Electric Circuit Analysis and Its Further Development

Kohshi Okumura

Faculty of Applied Information Science
Hiroshima Institute of Technology, Hiroshima, Japan
E-mail: o.kohshi@gmail.com

Received March 31, 2008
Revised November 26, 2008

This paper surveys applications of Krawczyk–Moore–Jones's algorithm and presents its further developments. The application is focused on nonlinear electric circuit analysis. The further developments are described on parallel KMJ algorithm, Gray code KMJ algorithm and KMJ processor.

Key words: interval operation, Krawczyk–Moore–Jones's algorithm, parallel algorithm, FPGA implementation, Gray code, all solutions

1. Introduction

The harmonic balance method (abbreviated as HB method) is classical and has been used for the mode analysis of nonlinear oscillations in physical systems [1]. In the HB method the determining equation of amplitudes and phases for each frequency component becomes nonlinear simultaneous equations, in which each equation takes the polynomial form of $f(x) = \sum_k a_k x_1^{\alpha_{1k}} \cdots x_n^{\alpha_{nk}}$. In recent years, the analysts of nonlinear electric and electronic circuits pay a special attention to the HB method and try to study the precise bifurcation analysis [42, 43, 44, 45, 46]. This is most likely because of the development of computer algebra software such as Maple and Mathematica.

Another aspect in electric circuit analysis is on the load-flow problems in a power network. In the power network in order to determine the voltage at each node we need the solutions of specified load-flow equation that is a set of quadratic equations $f(x) = \sum_{i,j} a_{ij} x_i x_j$ with many variables. In particular, finding multiple solutions has become a recent topic because voltage instability occurs when the power network has heavy load, namely, when the load flow equation has possible multiple solutions of the voltages.

Although it is also classical, the last problem to be described here is to determine DC (direct current) operating points of nonlinear resistive circuits. This problem is reduced to solving the specified nonlinear simultaneous equation. In this case the nonlinear part of the equations is mostly given by the specific form $f(x) = \sum_i a_i f_i(x_i)$ because the circuit equation is described by two terminal elements of nonlinear resistors. Usually $f_i(x_i)$ is an exponential or monotonically increasing function.

All these three problems in electric circuits or networks are reduced to finding all real solutions in a given region. For that purposes usually Newton or Newton-type method are being used even now. However, when we use Newton method, we encounter the following common questions:

(a) How can we set the starting point?
(b) Whether or not all solutions in the specified region can be determined?
(c) When no solution exists in the given region, how can its nonexistence be decided?

By contrast to Newton or Newton-type method, there is the powerful method using the interval operation [2, 3, 8]. It is called the interval Newton method [3, 4, 5, 6, 7]. In 1978 there came out the computer oriented strong algorithm developed by R. Moore and S. Jones [11]. This algorithm was originally proposed by R. Krawczyk [9]. We call this algorithm the Krawczyk–Moore–Jones's algorithm, abbreviated as the KMJ algorithm. When a nonlinear equation of n variables is to be solved by the KMJ algorithm, the initial region specified as an n-dimensional rectangular region is partitioned successively into smaller subregions, determining whether or not a unique solution exists in the subregion. Thus, it is possible in principle that all solutions in the initial region can be determined. This procedure makes the KMJ algorithm more powerful and useful for the above questions (a) to (c).

With the KMJ algorithm being a turning point, some modified versions and applications have been appeared from the mathematical point of view [13, 20] as well as the engineering viewpoint [18, 21]. The KMJ algorithm takes the Jacobi-type iteration. After this, the Gauss–Seidel version is proposed [15, 16]. In Ref. [19] the Gauss–Seidel version was first applied to find all the DC operating points in nonlinear resistive circuits although the number of variables was two in the example. The comparison of Gauss–Seidel version with the algorithm was made by applying both iteration types to the load-flow equation with 8 variables [23, 26]. By denoting nonlinear resistive characteristic term by $f_i(x_i)$ and the other linear term by $l_i(x_1, \ldots, x_n)$ the circuit equation is modified into the type $f_i(x_i) = l_i(x_1, \ldots, x_n)$. To this type of interval nonlinear equation Ref. [22] proposed iterative procedure to find all the DC operating points in a given region. Further Ref. [24] gave an improved version of interval method for DC operating points due to extended division operation and the method described in Ref. [5], where the interval Jacobian matrix is represented by the sum of diagonal matrix formed by the diagonal elements and the other matrix by remaining elements.

In the time domain-analysis of switched nonlinear circuits one of the main problems consists in the calculation of the zero of strongly nonlinear function $f(t)$ (t: time) within a given time interval. For this purpose Ref. [25] compares interval Newton's algorithm with Krawczyk's one.

In the KMJ algorithm we need several computational tests to find "the safe starting region" [10, 14]. The non-existence tests, the existence tests and the convergence tests are of essential importance. Specifically, the non-existence test, namely whether or not the interval function includes zero, is simple. Being combined the

test with the method of linear programming, the KMJ algorithm has been able to find all DC operation points in higher computational speed [27, 28]. These specific algorithms are powerful if the number of nonlinear terms is less than that of linear terms.

Practically, however, when the KMJ algorithm is applied to the determining equation with many terms each of which is comprised of many variables as stated before, it requires tremendously fast computing as well as vast amounts of memory. To overcome the defects, the KMJ algorithm has been so far modified from the standpoint of sequential computation. In other words the KMJ algorithm has its own limitation as far as we are interested in a sequential algorithm.

In order to improve this difficult situation the parallelisation of the KMJ algorithm is proposed [32]. In this survey first we outline the parallel KMJ algorithm that has not been attained those days in the field of interval analysis. The parallelisation is carried out in MIMD computer. The algorithm is parallelised naturally through the sequential algorithm of the original KMJ algorithm. The effectiveness of the parallel implementation will be shown.

Secondly, the KMJ algorithm takes a lot of computing time for large initial region because of the many number of the iterative bisections of the regions. If we are able to calculate the upper and lower bounds of the interval from the most significant bits, we can make the computational cost decrease and are able to calculate in arbitrary precision. For this purpose, Gray code KMJ algorithm has been proposed [33, 37, 35, 36, 38].

Recent research to achieve high performance with field-programmable gate arrays (FPGA)-based computing are now accelerating [41]. If the KMJ algorithm processor will become possible, the reduction of computational cost will be able to be attained. In this sense, the implementation of the KMJ algorithm by FPGA are surveyed [39, 40, 38].

2. KMJ algorithm

2.1. Interval operation and Krawczyk function

We denote by $I(\mathbf{R}^n)$ the set of n-dimensional rectangles. Let $X \in I(\mathbf{R})$, $Y \in I(\mathbf{R})$ be the real closed intervals $X = [\underline{x}, \overline{x}]$, $Y = [\underline{y}, \overline{y}]$. The interval operation is defined by

$$
\begin{aligned}
&X + Y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}], \quad X - Y = [\underline{x} - \overline{y}, \underline{x} - \underline{y}],\\
&X \cdot Y = [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}],\\
&X/Y = [\min\{\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}\}, \max\{\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}\}], \quad Y \not\ni 0.
\end{aligned}
$$

The mid point $m(X)$, the width $w(X)$ and the absolute value of X are defined by

$$
\begin{aligned}
&m(X) = \frac{1}{2}(\underline{x} + \overline{x}), \quad w(X) = \overline{x} - \underline{x},\\
&|X| = \max\{|\underline{x}|, |\overline{x}|\}.
\end{aligned}
$$

We denote the set of interval matrices by $I(\mathbf{R}^{n\times n})$. The interval vector $\boldsymbol{X} \in I(\mathbf{R}^{n\times 1})$ and its midpoint $m(\boldsymbol{X}) \in \mathbf{R}^{n\times 1}$ are represented by

$$\begin{aligned}&\boldsymbol{X} = (X_1, X_2, \ldots, X_n)^{\mathrm{t}}, \quad X_k \in I(\mathbf{R}),\ k = 1, \ldots, n,\\ &m(\boldsymbol{X}) = (m(X_1), m(X_2), \ldots, m(X_n))^{\mathrm{t}},\end{aligned}$$

where "t" denotes the transposition. An interval matrix $\boldsymbol{A} \in I(\mathbf{R}^{n\times n})$ and its norm are defined by

$$\begin{aligned}&\boldsymbol{A} = (A_{ij}), \quad A_{ij} \in I(\mathbf{R}),\ i, j = 1, 2, \ldots, n,\\ &\|\boldsymbol{A}\| = \max_i \sum_{j=1}^{m} |A_{ij}|.\end{aligned}$$

The inclusion monotonic interval extension $\boldsymbol{F}(\boldsymbol{X})$ of $\boldsymbol{f}(\boldsymbol{x}) \in \mathbf{R}^{n\times 1}$ is defined by

$$\boldsymbol{F}(\boldsymbol{X}) = (F_1(\boldsymbol{X}), F_2(\boldsymbol{X}), \ldots, F_n(\boldsymbol{X}))^{\mathrm{t}}.$$

A natural interval extension of the real function is performed by replacing the real variables with the corresponding interval variables. The arithmetic operation is also replaced with corresponding interval operation. The Jacobian matrix of $\boldsymbol{f}(\boldsymbol{x})$ and its interval extension are written by $\boldsymbol{f}'(\boldsymbol{x}) = (f'_{ij}(\boldsymbol{x}))$ and $\boldsymbol{F}'(\boldsymbol{X}) = (F'_{ij}(\boldsymbol{X}))$, $i, j = 1, 2, \ldots, n$, respectively. The Krawczyk operator for the equation $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$ is defined by

$$\begin{aligned}&\boldsymbol{K}(\boldsymbol{X}) = \boldsymbol{y} - \boldsymbol{Y}\boldsymbol{f}(\boldsymbol{y}) + [\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{F}'(\boldsymbol{X})](\boldsymbol{X} - \boldsymbol{y}),\\ &\boldsymbol{y} = m(\boldsymbol{X}),\ \boldsymbol{Y} = [m(\boldsymbol{F}'(\boldsymbol{X}))]^{-1},\end{aligned}$$

where $\boldsymbol{I}$ is the $n \times n$ identity matrix. The conditions to confirm the existence of the solution in $\boldsymbol{X}$ is called Moore-test shown below [12, 17]:

(1) If $\boldsymbol{F}(\boldsymbol{X}) \not\ni \boldsymbol{0}$, then no solution exists in $\boldsymbol{X}$.
(2) If $\boldsymbol{K}(\boldsymbol{X}) \cap \boldsymbol{X} = \emptyset$, then no solution exists in $\boldsymbol{X}$, where $\emptyset$ is empty set.
(3) If $\boldsymbol{K}(\boldsymbol{X}) \subseteq \boldsymbol{X}$ and $\|\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{F}'((X))\| < 1$, then a unique solution exists in $\boldsymbol{X}$.

2.2. KMJ algorithm finding all zeros of $\boldsymbol{f}(\boldsymbol{x})$

Here we show KMJ algorithm in the following:

S1. Set $\boldsymbol{X}$ to $\boldsymbol{X}_{\text{initial}}$. Set list T to be empty.
S2. Test the region $\boldsymbol{X}$ by computing $\boldsymbol{F}(\boldsymbol{X})$ and $\boldsymbol{K}(\boldsymbol{X})$.
  (i) If no solution exist in $\boldsymbol{X}$, i.e., $\boldsymbol{F}(\boldsymbol{X}) \not\ni \boldsymbol{0}$ or $\boldsymbol{K}(\boldsymbol{X}) \cap \boldsymbol{X} = \emptyset$, then go to S4.
  (ii) If a unique solution exists, namely $\boldsymbol{K}(\boldsymbol{X}) \subseteq \boldsymbol{X}$ and $\|\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{F}'((X))\| < 1$ are satisfied, then we apply Newton method to $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$ and find the solution by taking $m(\boldsymbol{x})$ as the starting point. And go to S4.
  (iii) If neither of the above conditions is satisfied, then go to S3.
S3. Bisect the region $\boldsymbol{X}$ according to the rules below, i.e., $\boldsymbol{X} = \hat{\boldsymbol{X}} \cup \check{\boldsymbol{X}}$. Add the remaining region $\check{\boldsymbol{X}}$ to list T and set $\boldsymbol{X}$ to $\hat{\boldsymbol{X}}$; go to S2.

S4. Test list T.
(i) If list T is empty, then terminate.
(ii) If list T is not empty, then set $\boldsymbol{X}$ to region in list T, delete this region from list T and go to S2.

The bisection rule: Bisect $\boldsymbol{X}$ in the coordinate direction which maximizes $w(X_k)$, i.e.,

$$\hat{\boldsymbol{X}} = (X_1, \ldots, \underline{X}_k, \ldots, X_n), \quad \underline{X}_k = [\underline{x}_k, m(X_k)],$$
$$\check{\boldsymbol{X}} = (X_1, \ldots, \overline{X}_k, \ldots, X_n), \quad \overline{X}_k = [m(X_k), \overline{x}_k].$$

3. Parallel KMJ algorithm

3.1. KMJ algorithm parallelised in MIMD computer

Here we describe briefly the parallel KMJ algorithm. In our case parallelisation is carried out in MIMD computer. Namely, we parallelise the procedure of sequential KMJ algorithm on message passing parallel computer systems. The memory is local to a processor and messages must be exchanged between the local memory of the other processors. Efficient communication is very important to this message passing implementation. We show the procedure implemented by a general master-slave algorithm.

Let us consider N processing elements denoted by PE_i $(i = 0, \ldots, N-1)$. These are separated to one master and $N-1$ slave;

$$\begin{aligned} &\text{Master: } \mathrm{PE}_0, \\ &\text{Slave: } \mathrm{PE}_i\,, \quad i \in S,\ S = \{1, 2, \ldots, N-1\}. \end{aligned} \tag{1}$$

The master process is responsible for coordinating the work of the others and the slave processes do not communicate with one another. That is, the communication is limited to

$$C_{0\to i}[\mathit{message}] \quad \text{or} \quad C_{i\to 0}[\mathit{message}], \quad i \in S, \tag{2}$$

where the communication from PE_i to PE_j is denoted by $C_{i\to j}[\mathit{message}]$ and *message* denotes the contents of communication, i.e., a region "$\boldsymbol{X}$," a flag "Terminate" or "Request."

The master PE_0 has the list T and requests the slaves to test the region. When one task by a slave is completed, the master sends other region to the slave for the next task. Once all tasks have been handed out, termination messages are sent instead. The procedure of slaves are shown below.

Algorithm for slaves PE_i, $i \in S$

S_i1. Get the region $\boldsymbol{X}_i$ from master PE_0 $(C_{0\to i}[\boldsymbol{X}])$.
S_i2. Test the region $\boldsymbol{X}_i$ by computing $\boldsymbol{F}(\boldsymbol{X}_i)$ and $\boldsymbol{K}(\boldsymbol{X}_i)$.
(i) If there are no solutions in $\boldsymbol{X}_i$, then go to S_i4.
(ii) If there exists unique solution in $\boldsymbol{X}_i$, then find a solution and go to S_i4.
(iii) If neither of the above two conditions is satisfied, then go to S_i3.

S_i3. Bisect the region $\boldsymbol{X}_i = \hat{\boldsymbol{X}}_i \cup \check{\boldsymbol{X}}_i$. Send the region $\check{\boldsymbol{X}}_i$ back to the master PE_0 ($C_{i\to 0}[\check{\boldsymbol{X}}_i]$) and set $\boldsymbol{X}_i$ to $\hat{\boldsymbol{X}}_i$; go to S_i2.

S_i4. Request the next region from the master ($C_{i\to 0}$[Request]) and go to S_i5.

S_i5. Wait for another task.

(i) If the master sends a new region ($C_{0\to i}[\boldsymbol{X}]$ or $C_{0\to i}[\check{\boldsymbol{X}}_j]$), then set $\boldsymbol{X}_i$ to the new region and go to S_i2.

(ii) If the master sends a termination message ($C_{0\to i}$[Terminate]), then the slave terminates the process.

The procedure of the master PE_0 is described in detail below. List W is the list of waiting slave PE_i. The number of element in the list W is denoted by $|W|$.

Algorithm for master PE_0

S_01. Set the region $\boldsymbol{X}$ to $\boldsymbol{X}_{\text{initial}}$ and send the region $\boldsymbol{X}$ to the slave PE_1 ($C_{0\to 1}[\boldsymbol{X}]$). Set list T to empty and add remaining slaves PE_i ($i = 2, \ldots, N-1$) to list W.

S_02. Wait for the messages from slaves.

(i) If the slave PE_i, $i \in S$ sends the region $\check{\boldsymbol{X}}_i$ ($C_{i\to 0}[\check{\boldsymbol{X}}_i]$), then go to S_03.

(ii) If the slave PE_i, $i \in S$ requests a new region ($C_{i\to 0}$[Request]), then go to S_04.

S_03. Test list W.

(i) If W is not empty, then send $\check{\boldsymbol{X}}_i$ to a slave PE_j in list W ($C_{0\to j}[\check{\boldsymbol{X}}_i]$) and delete PE_j form list W.

(ii) If list W is empty, add the region $\check{\boldsymbol{X}}_i$ to list T and go to S_02.

S_04. Test list T.

(i) If list T is not empty, then set $\boldsymbol{X}$ to a region in list T, delete this region from list T and send the region to PE_i ($C_{0\to i}[\boldsymbol{X}]$); go to S_02.

(ii) If list T is empty, then go to S_05.

S_05. Test list W for termination.

(i) If $|W| < N-2$, then add PE_i to list W and go to S_02.

(ii) If $|W| = N-2$, then send termination messages to all slaves ($C_{0\to i}$[Terminate], $i \in S$), and terminate.

The flow chart of the parallel KMJ algorithm is shown in Fig. 1. First, the master send the given region to PE_1 and go to the state "Wait." On the other hand the slaves starts from the state "Start." In this case, the message of instruction is $C_{i\to 0}$[Request] or $C_{0\to i}$[Terminate] which are implemented by a communication of an integer flag. The messages of region are $C_{0\to i}[\boldsymbol{X}]$, $C_{i\to 0}[\check{\boldsymbol{X}}_i]$ and $C_{0\to j}[\check{\boldsymbol{X}}_i]$ which consist of $2n$ real numbers.

3.2. Performance results

We apply the parallel KMJ algorithm to the determing equation with eight unknowns given in Ref. [32]. This determining equation has at most 37 nonlinear terms derived from the determining equation derived from the circuit equation (the set of the nonlinear ordinary differential equations) in dealing with the nonlinear oscillation in three-phase circuit. The procedure is implemented on a distributed memory

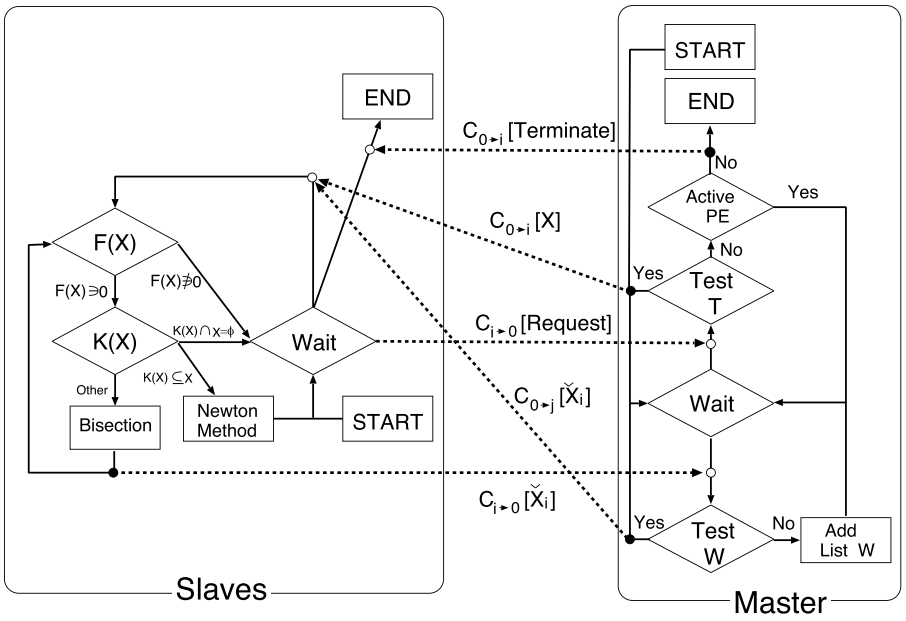

Fig. 1. The flow chart of parallel KMJ algorithm for finding all solutions.

parallel processor system Hitachi SR2201 which has a 3-dimensional crossbar network and whose transmission rate between two processors is 300 megabytes/s. The performance of one processing element is 0.3 GFLOPS. In parallel algorithm it is of utmost importance to be able to assess the speed gain expected from the operation of processors in parallel. In order to test the effect of communication, we compare the computing time of the parallel KMJ algorithm with two processing elements to that of the sequential counterpart. In this parallel implementation, one of the two processing elements is master and the other is slave. The result of performance is shown in Table 1. We can see from this result that the effect of communication defined by

$$1 - \frac{\text{computing time with 1 PE (not parallelised)}}{\text{computing time with 2 PEs (parallelised)}}$$

is about 1.8 %. The computing time by the parallel KMJ algorithm is shown in Fig. 2. From the figure we can see that increasing the number of processors leads to decrease in computing time.

Table 1. Efficiency of communication.

PE	Time [s]
1 PE (not parallelised)	2.13×10^5
2 PEs (parallelised)	2.17×10^5

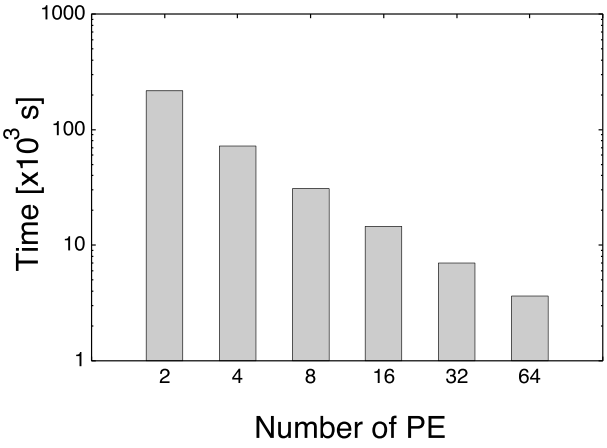

Fig. 2. Computing time and number of processors.

The speed-up ratio of a parallel algorithm [31] defined by

$$\frac{\text{computing time with 2 PEs}}{\text{computing time with } i \text{ PEs}}, \quad i = 4, 8, 16, 32, 64$$

shows the efficiency with respect to the case of two processing elements and is shown in Table 2. In general, because one of the processing elements is used for master, the theoretically optimal speedup in the parallelised algorithm is proportional to the number of slaves

$$(\text{number of processing elements}) - 1. \tag{3}$$

In the case of 4, 8 and 16 PEs, Table 2 shows that the performance achieves the theoretically optimal speed-up. However, the over concentration of communication to master makes the performance worse when the number of PEs is 64.

Table 2. Speedup by parallel computation.

PE	Time [s]	Speedup
2	2.17×10^5	—
4	7.19×10^4	3.0
8	3.10×10^4	7.0
16	1.45×10^4	15.0
32	7.05×10^3	30.5
64	3.62×10^3	59.9

We have briefly presented the parallel KMJ algorithm and have demonstrated its effectiveness, being compared with the sequential one. Furthermore, by saving the communication time between processing elements and idle time, we realize an

efficient implement in message passing parallel computer. Several improvements of the parallel KMJ algorithm can be seen also in Ref. [32]. It suggests a method of optimizing the idle and communication time, realizes the further decrease of communication by distributing master work and confirms the effect.

4. Gray code KMJ algorithm

4.1. Gray code

Gray code [33] is a representation of natural numbers. This code has a property that two successive values differ in only one digit. This property has long been used in electrical engineering to facilitate error correction in digital communications such as cable TV systems.

Recently, Tsuiki [37] has reported another aspect of Gray code. He uses Gray code expansion of real numbers to define computability of real functions. The expansion is an infinite sequence of $\{0, 1\}$ in which at most one undefinedness is allowed. The topological property of the code enables digit sequential arithmetics from the most significant digits (abbreviated as MSD).

Here is an algorithm to covert natural binary code $B(n)$ to Gray code $G(n)$, where $B(n)$ and $G(n)$ are the array of n bits in the usual binary representation. The algorithms of converting natural binary code to Gray code is given by $G(0) := B(0)$, $G(i) := B(i-1) \oplus B(i)$ $(i = 1, \ldots, n-1)$, where $\oplus$ denotes exclusive or. The algorithm to convert Gray code to binary code is given simply by $B(0) := G(0)$, $B(i) := B(i-1) \oplus G(i)$ $(i = 0, 1, \ldots, n-1)$. By introducing undefined bits $\perp$ the real interval can be expressed.

The objectives of KMJ algorithm using Gray code is to demonstrate the reduction of computational cost to obtain all solutions with arbitrary precision. Namely, the seemingly multiple roots in usual method can be discriminated rigorously. In this section the arbitrary precision KMJ algorithm is provided by employing the operation of Gray code interval and some numerical examples are given.

4.2. Gray code interval

We define a Gray code interval $X_{\mathrm{G}n_x}$ using a pair of Gray codes as follows:

$$X = [\underline{X}, \overline{X}] \leftrightarrow X_{\mathrm{G}n_x} \equiv [x_{\mathrm{GL}n_x}, x_{\mathrm{GU}n_x}]. \tag{4}$$

where "$\leftrightarrow$" represents the correspondence between the closed interval X and the Gray code interval $X_{\mathrm{G}n_x}$. Gray code $x_{\mathrm{G}n_x}$ can be expressed by

$$x_{\mathrm{G}n_x} = (x_{\mathrm{GS}} x_{\mathrm{GS}'}) \cdots x_{(-2)} x_{(-1)} . x_0 x_1 \cdots x_{n_x - 1} x_{n_x} \leftrightarrow (\underline{x}, \overline{x}), \tag{5}$$

where the first sign bit x_{GS} is defined by $x_{\mathrm{GS}} = 1$ (if $\underline{x} > 0$), $x_{\mathrm{GS}} = 0$ (if $\overline{x} < 0$), $x_{\mathrm{GS}} = \perp_1$ (if $\underline{x} < 0 < \overline{x}$) and the second sign bit $x_{\mathrm{GS}'}$ is always set to 1.

The $x_{\mathrm{GL}n_x}$ and $x_{\mathrm{GU}n_x}$ are Gray code and satisfy $\underline{x_{\mathrm{GL}n_x}} = \underline{X}$ and $\overline{x_{\mathrm{GU}n_x}} = \overline{X}$. As an example, let us represent the interval $X = [0.75, 0.78125]$ by Gray code. We have

$$\begin{cases} x_{\mathrm{GL}n_x} = (11)0.101000000\bot_1\bot_2 \leftrightarrow (768, 770)2^{-10} = (0.75, 0.75195), \\ x_{\mathrm{GU}n_x} = (11)0.101001000\bot_1\bot_2 \leftrightarrow (798, 800)2^{-10} = (0.77929, 0.78125). \end{cases} \tag{6}$$

The Gray code expression of $x_{\mathrm{GL}n_x}$ and $x_{\mathrm{GU}n_x}$ has a common bits x_i $(-1 \leq i \leq 4)$ and sign bit. Hence we can represent $x_{\mathrm{GL}n_x}$ and $x_{\mathrm{GU}n_x}$ by

$$\begin{cases} x_{\mathrm{GL}n_x} = \underbrace{(11)0.10100}_{i}\underbrace{0000\bot_1\bot_2}_{l}, \\ x_{\mathrm{GU}n_x} = \underbrace{(11)0.10100}_{i}\underbrace{1000\bot_1\bot_2}_{l}. \end{cases} \tag{7}$$

If $x_{\mathrm{GL}n_x}$ and $x_{\mathrm{GU}n_x}$ are close, the upper bits of them are same in Gray code representation. Using this property, we are able to efficiently represent the Gray code interval by common upper bits $i_{x_{\mathrm{G}}}$ and remaining lower bits $\underline{l_{x_{\mathrm{G}}}}$ and $\overline{l_{x_{\mathrm{G}}}}$. Here we define the upper i bits by

$$i_{x_{\mathrm{G}}} = (11)0.10100\bot_1\bot_2 \leftrightarrow (48, 50)2^{-6} = (0.75, 0.78125).$$

From the definition, the Gray code $i_{x_{\mathrm{G}}}$ satisfies $i_{x_{\mathrm{G}}} \subset X$. We define $|i_{x_{\mathrm{G}}}|$ by the number of bits except the sign bits, $\bot_1$ and $\bot_2$. In this example $|i_{x_{\mathrm{G}}}|$ is equal to 6. Next, as for the lower bits of Eq. (7) we represent the part by

$$\begin{cases} \overline{l_{x_{\mathrm{G}}}} = [1]000\bot_1\bot_2, \\ \underline{l_{x_{\mathrm{G}}}} = [0]000\bot_1\bot_2. \end{cases} \tag{8}$$

The first bit of $\underline{l_{x_{\mathrm{G}}}}$ and $\overline{l_{x_{\mathrm{G}}}}$ is the special bit which corresponds to the $\bot_1$ of the upper bits $i_{x_{\mathrm{G}}}$. We define $|l_{x_{\mathrm{G}}}|$ by the number of bits except the $\bot_1$ and $\bot_2$ of $\underline{l_{x_{\mathrm{G}}}}$ or $\overline{l_{x_{\mathrm{G}}}}$. By $i_{x_{\mathrm{G}}}$, $\underline{l_{x_{\mathrm{G}}}}$, $\overline{l_{x_{\mathrm{G}}}}$ the Gray code interval can be represented. This representation leads us to the efficient calculation of interval arithmetics because the common bits $i_{x_{\mathrm{G}}}$ can be reused in the processes of calculation. Fig. 3 shows the effect of $|l_{x_{\mathrm{G}}}|$ using the example given in (7). The two-headed arrow shows the interval X. The dotted lines and solid thin lines are upper bound and lower bound, respectively. The part of $i_{x_{\mathrm{G}}}$ represents the minimal interval which include the interval X. In this case the interval X equals to $i_{x_{\mathrm{G}}}$. According to the increase of $|l_{x_{\mathrm{G}}}|$ the precision of $x_{\mathrm{GL}n_x}$ and $x_{\mathrm{GU}n_x}$ grows.

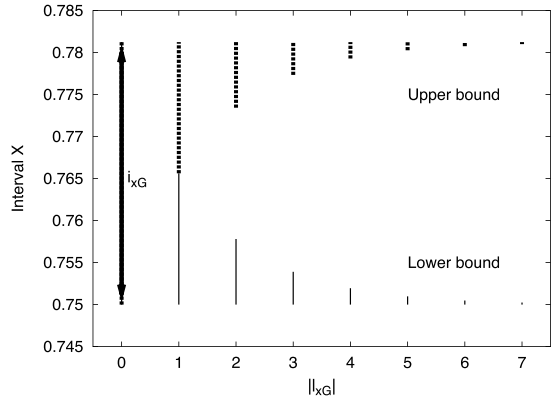

Fig. 3. Representation of an interval by different $|l_{x_{\mathrm{G}}}|$. The two-headed arrow shows the interval X. The dotted lines and solid thin lines are upper bound and lower bound, respectively. The part of $i_{x_{\mathrm{G}}}$ represents the minimal interval which include the interval X. In this case the interval X equals to $i_{x_{\mathrm{G}}}$. According to the increase of $|l_{x_{\mathrm{G}}}|$ the precision of $x_{\mathrm{GL}n_x}$ and $x_{\mathrm{GU}n_x}$ grows.

4.3. Gray code arithmetic

Let the given two input Gray codes be a_{G}, b_{G} and let a two-operand interval arithmetic operation be "$\circ$." Using the Gray code we define arithmetic operations $c_{\mathrm{G}} = a_{\mathrm{G}} \circ b_{\mathrm{G}}$ where c_{G} denotes the output Gray code. Let

$$\begin{aligned} A &= [\underline{A}, \overline{A}] = [a-1, a+1]2^{n_a} \leftrightarrow a_{\mathrm{G}}, \\ B &= [\underline{B}, \overline{B}] = [b-1, b+1]2^{n_b} \leftrightarrow b_{\mathrm{G}}, \\ C &= [\underline{C}, \overline{C}] = [c-1, c+1]2^{n_c} \leftrightarrow c_{\mathrm{G}} \end{aligned}$$

be the interval representations of the Gray codes a_{G}, b_{G}, c_{G} respectively, where n_a, n_b and n_c are called the scaling exponent. We define Gray code c_{G} as the Gray code of maximum scaling exponent n_c (i.e., of minimum interval width of C) which satisfies

$$A \circ B \subset C. \tag{9}$$

The algorithm that performs addition using Gray code expansion is given in Ref. [37] and the algorithms for other arithmetics have been reported in Ref. [38, 35].

4.4. Required accuracy for performing Moore-test

The Gray code interval is serially computed from MSB. Hence we can stop the computation at the moment when we get enough accuracy. In order to estimate the condition $\boldsymbol{F}(\boldsymbol{X}) \not\ni \mathbf{0}$, we have only to estimate the sign bit of the interval bounds of $\boldsymbol{F}(\boldsymbol{X})$. That is, when an interval $F(X)$ which is a coordinate of $\boldsymbol{F}(\boldsymbol{X})$ is represented by interval bounds $[f_{\mathrm{GL}n_f}, f_{\mathrm{GU}n_f}]$, the following propositions are satisfied:

$$\begin{aligned} &f_{\mathrm{GL}n_f} > 0 \text{ or } f_{\mathrm{GU}n_f} < 0 \implies F(X) \not\ni 0, \\ &f_{\mathrm{GL}n_f} < 0 \text{ and } f_{\mathrm{GU}n_f} > 0 \implies F(X) \ni 0. \end{aligned}$$

In consequence, we can stop the calculation of $F(X)$ at the accuracy that the sign bit of the interval bounds $f_{\mathrm{GL}n_f}$ and $f_{\mathrm{GU}n_f}$ are determined. Thus, when all the sign bits of the interval bounds of $\boldsymbol{F}(\boldsymbol{X})$ are determined, we can stop the MSB first calculation of $\boldsymbol{F}(\boldsymbol{X})$. In almost same way, we can attain the required accuracy for $\boldsymbol{K}(\boldsymbol{X})$.

4.5. Bisection procedure

The bisection of the region can be done with the least computational cost by using the property of the Gray code interval. After the region $\boldsymbol{X}$ is updated, $\boldsymbol{F}(\boldsymbol{X})$ and $\boldsymbol{K}(\boldsymbol{X})$ are efficiently computed by using the MSB first computation. Using the shrinking property, we can update efficiently Gray code interval operation. The method of bisection of interval is given in Ref. [36].

4.6. Examples

4.6.1. Multiple operating points of nonlinear circuit

We apply the proposed method to a circuit equation. Fig. 4 shows the well-known nonlinear circuit with two Esaki diodes. The circuit equation is

$$f_1(V_1, V_2) = E - RI_1(V_1) - (V_1 + V_2) = 0, \tag{10}$$

$$f_2(V_1, V_2) = I_1(V_1) - I_2(V_2) = 0, \tag{11}$$

where the characteristics of Esaki diodes are assumed to be represented by

$$I_1(V_1) = 2.5V_1^3 - 10.5V_1^2 + 11.8V_1, \tag{12}$$

$$I_2(V_2) = 0.43V_2^3 - 2.69V_2^2 + 4.56V_2. \tag{13}$$

We fix the circuit parameter $E = 30.0$ and $R = 13.3$ and set the initial region $\boldsymbol{X} = ([-8, 8], [-8, 8])$, i.e.,

$$i_{x_{1\mathrm{G}}} = i_{x_{2\mathrm{G}}} = (\perp_1 1)\perp_2\perp\perp, \quad \begin{cases} \overline{l_{x_{1\mathrm{G}}}} = \overline{l_{x_{2\mathrm{G}}}} = [1]1\perp_1\perp_2, \\ \underline{l_{x_{1\mathrm{G}}}} = \underline{l_{x_{2\mathrm{G}}}} = [0]1\perp_1\perp_2. \end{cases}$$

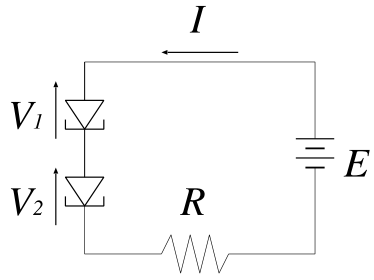

Fig. 4. Circuit with Esaki diode.

Table 3. Comparison of computational cost with interval computation with fixed accuracy bit $|l_x|$ and no reuse.

$\lvert l_x\rvert$ (bit)	4	10	20
Computational cost ($\times 10^8$ bit addition)	13.74	2.804	3.834

All 9 solutions can be found by the Gray code KMJ algorithm. The computational cost which is converted to the Gray code addition cost is 1.556×10^8 bit addition [35]. In order to clarify the efficiency of the Gray code KMJ algorithm, we apply interval computation with fixed accuracy bit $|l_x|$ without reuse of computation (see Table 3). We can confirm that the computational cost is considerably reduced by using the proposed method.

4.6.2. Finding all solutions with high accuracy

By the Gray code KMJ algorithm, we compute with very high accuracy all the solutions of a simultaneous nonlinear equation.

$$\begin{cases} f_1 = (x_1 - 1)^2 + \varepsilon - x_2 = 0, \\ f_2 = x_1^2 + x_2 - 1 = 0 \\ (\varepsilon = 0.6403143403040989). \end{cases} \tag{14}$$

The solutions can not be separately obtained by the ordinary double floating point operation. However, the Gray code KMJ algorithm gives us the separate solutions as follows:

$$\begin{aligned} \text{solution 1} \quad & i_{x_1} = (11)0.1101001 \ldots 010011 \perp_1 \perp_2 \\ & \leftrightarrow (0.6125360615 \ldots, 0.6125360652 \ldots), \\ & i_{x_2} = (11)0.1010111 \ldots 111000 \perp_1 \perp_2 \\ & \leftrightarrow (0.7904426418 \ldots, 0.7904426455 \ldots), \\ \text{solution 2} \quad & i_{x_1} = (11)0.1101001 \ldots 01000 \perp_1 \perp_2 \\ & \leftrightarrow (0.6125360652 \ldots, 0.6125360727 \ldots), \\ & i_{x_2} = (11)0.1010111 \ldots 11100 \perp_1 \perp_2 \\ & \leftrightarrow (0.7904426380 \ldots, 0.7904426455 \ldots). \end{aligned}$$

The solution i_{x_1} has the accuracy of 30 bit and i_{x_2} that of 29 bit. The computational cost is about 20 % when reusing the common bits, compared with the case not reused. It is shown by this example that the reduction in the computational cost in the search process of the solution is possible by reusing of the common bits of interval representation by Gray code.

5. Hardware approach to KMJ algorithm

Although the KMJ algorithm is a powerful software, the main disadvantage is its speed. This algorithm requires tremendously large amount of interval operation. To overcome the speed limitation of the software, specific hardware support is required. There are several works in which processors for interval computation have been designed [29, 30]. Because these processors are specified to interval computation, they can execute the arithmetic operations very fast. Nevertheless, the problem of designing specific hardware to perform the KMJ algorithm has not been studied as yet by means of logic circuits.

In recent years field-programmable gate arrays (FPGA) has gained attention and has demonstrated its potential power to various fields of computation [41]. The FPGA has possibility to realize the hardware implementation of the KMJ algorithm.

This section specialises briefly in an approach to the hardware implementation of the KMJ algorithm presented in Ref. [39]. We have designed logic circuits of Moore-test, have implemented it to the FPGA and have realized the KMJ algorithm.

5.1. Outline of the System

The configuration of the system is shown in Fig. 5. The system consists of CPU in AT-compatible PC and FPGA board on PCI bus. The Moore-test processor executes the part which contains interval computations, that is, the processor gets a region from CPU and tests the existence and nonexistence of solutions in the region. On the other hand, the CPU executes the part of real number operations, that is, the bisection of region and Newton method.

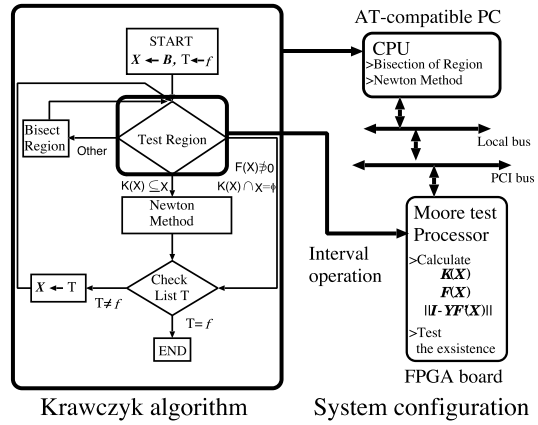

Fig. 5. The outline of Krawczyk algorithm and configuration of the system. The existence of solutions are tested by Moore-test processor. The functions $\boldsymbol{F}(\boldsymbol{X})$ and $\boldsymbol{K}(\boldsymbol{X})$ is the interval extension of equations and Krawczyk function, respectively. The region $\boldsymbol{B}$ is the initial region for Moore-test.

The Moore-test processor consists of five units illustrated in Fig. 6. The functions $\boldsymbol{F}(\boldsymbol{X})$ and $\boldsymbol{K}(\boldsymbol{X})$ are directly implemented in the processor as logic circuits.

Arithmetic Unit This unit executes operations of both floating-point number and interval number. Both operations use the same unit in common in order to reduce the number of Logic Cells (abbreviated as LCs) in FPGA. The output values are stored in registers.

Testing Unit This unit executes the Moore-test. This unit has also some inner modules to compare the floating-point numbers.

Controller This unit controls Arithmetic Unit and Testing Unit in order to calculate the values of $\boldsymbol{F}(\boldsymbol{X})$, $\boldsymbol{K}(\boldsymbol{X})$ and $\|\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{F}'(\boldsymbol{X})\|$.

Registers This unit feeds the values and intervals into Arithmetic Unit and stores the calculated data from ALU. This unit also outputs the values and region data directly to Arithmetic Unit and Testing Unit, and receives data directly from the above two unit.

PCI Bus Interface This unit controls signals between PCI bus and the main part of this processor.

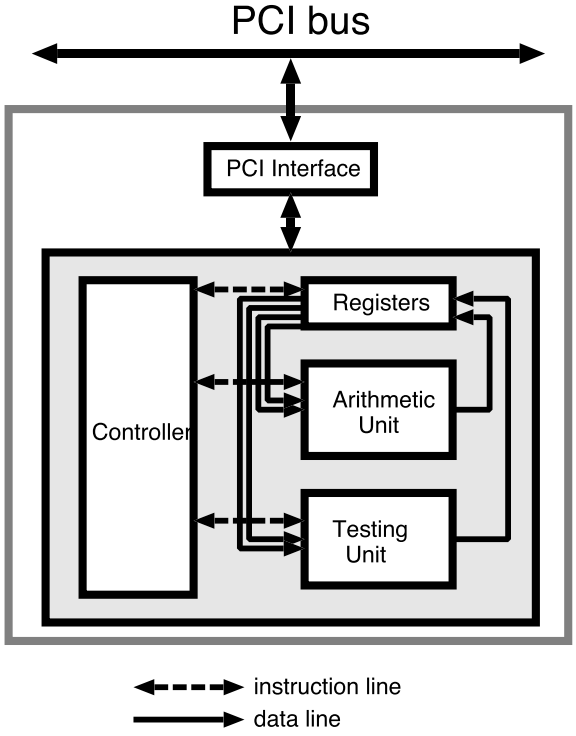

Fig. 6. The outline of configuration of Moore-test processor. The functions $\boldsymbol{F}(\boldsymbol{X})$ and $\boldsymbol{K}(\boldsymbol{X})$ are directly implemented in the processor as logic circuits.

5.2. Arithmetic unit

We use the floating-point number with 1 sign bit, n bits for exponent in excess 2^{n-1} notation and m bits for normalised mantissa. The number m, n can be decided according to the actual demand of the desired equations to be solved and the limitation of the FPGA to be used. As a basic operation, this unit has two sets of floating-point number adder, two set of floating-point number multipliers and one set of floating-point number divider. They can execute operations in parallel. This unit also has some additional unit to execute interval calculation using the same floating-point number unit as the one floating-point number.

5.2.1. Interval operation unit

In order that interval operations are executed in the same (or similar) number of clocks as floating-point number operations, we need the additional modules, namely some pre-processing and post-processing modules to be inserted before and after the basic operation. The followings are a brief description of the configuration of the Arithmetic Unit when it works as interval operation:

Addition Additions can be performed as simple two sets of floating-point number operations. Hence there is no need of pre-processing and post-processing. The action of Arithmetic Unit is shown in Fig. 7. Resister stores the upper values $\overline{a}$ and $\overline{b}$ and the lower values $\underline{a}$ and $\underline{b}$. Adder1 and Adder2 calculate $\overline{a+b}$ and $\underline{a+b}$, and store them in Resister as shown in Fig. 7.

Subtraction The pre-processor for interval subtraction is composed of two inverters and one multiplexer. Interval subtraction needs the inversion of the

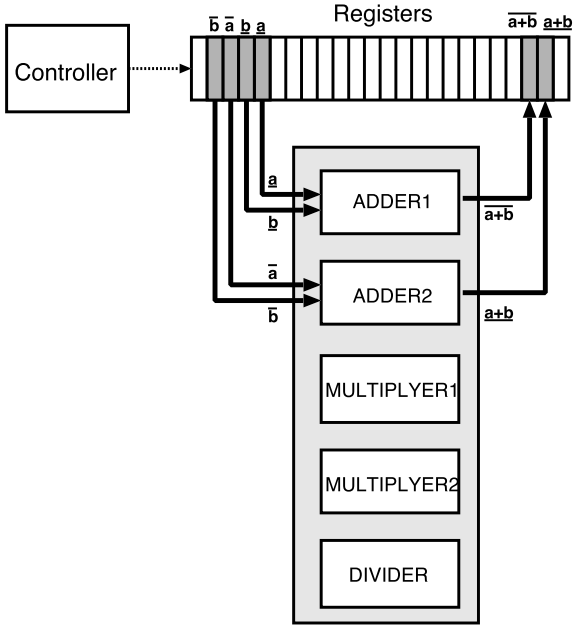

Fig. 7. Configuration of interval addition.

sign of the subtracting value and the exchange of the lower bound and the upper bound.

Multiplication It is possible to let in advance the four possible endpoints $\underline{x}\underline{y}$, $\underline{x}\overline{y}$, $\overline{x}\underline{y}$, $\overline{x}\overline{y}$ define the lower and upper bounds of multiplication XY. The result depends on the signs of X and Y and has two cases. For each case the circuit is designed. These multiplications are realized with two floating-point number multipliers by changing the inputting operands according to the signs of the bounds [39].

Division Interval division generally needs four sets of floating-point number divider. However, no general interval division is executed when we deal with polynomial equations.

5.3. Testing unit

5.3.1. Checking of relations in Moore-test

After the calculations of $\boldsymbol{F}(\boldsymbol{X})$, $\boldsymbol{K}(\boldsymbol{X})$ and $\|\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{F}'(\boldsymbol{X})\|$, the existence and nonexistence of a solution are tested by the four relations in Moore-test. The followings (a) to (d) are the relations to be tested with respective logics. The checking is done component-wise by using comparator. (a) $\boldsymbol{F}(\boldsymbol{X}) \not\ni \boldsymbol{0}$, (b) $\boldsymbol{K}(\boldsymbol{X}) \cap \boldsymbol{X} = \emptyset$, (c) $\boldsymbol{K}(\boldsymbol{X}) \subseteq \boldsymbol{X}$, (d) $\|\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{F}'(\boldsymbol{X})\| < 1$. Since the norm $\|\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{F}'(\boldsymbol{X})\|$ is calculated, this relation can be examined by only being compared with the number 1.

5.3.2. Comparator

There are also Comparators in this system in order to examine the above relations. This comparator includes the inner circuit that outputs 0 or 1 according to the result (output 0: if $a > b$, 1: if $a < b$). Using the result, the whole circuit outputs less or greater values of the two input operands.

5.4. Controller

This unit executes the whole computation by controlling Arithmetic Unit, Testing Unit and sets of registers. By means of this controller, the data fetched from the registers are transfered to Arithmetic Unit, and the data which Arithmetic Unit outputs are stored directly into the registers.

5.4.1. Control over interval computation

The Moore-test processor must execute both float-pont number and interval computation. Here explanation is limited to the control over the interval computation, which is rather complicated than the control over floating-point number computation because Done signals that each Adder transfers to Controller may be generated in the different clock. Fig. 8 displays the addition of two intervals A and B as an example. The figure between state1 and state2 illustrates the process of control over interval computation as follows:

1. When state1 becomes active, Controller generates the instructions and transmits them to the registers which holds the data needed to calculate, and the registers gives the needed values to ADDER2 as well as ADDER1, receiving the signal. Controller also turns on the switch of ADDER1 and ADDER2,

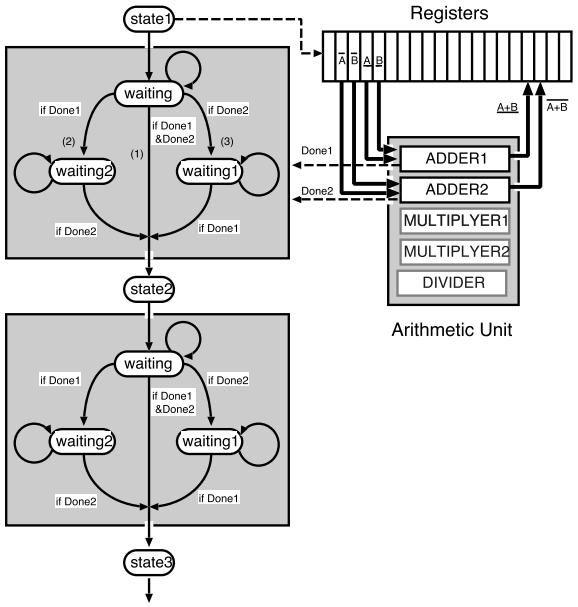

Fig. 8. Control over interval operation.

and both of them prepares for the calculation which starts in the next clock, receiving instructions and data. At the same time Controller goes to waiting state.

2. ADDER1 and ADDER2 starts calculating using the operands from the registers, and use several clocks for adding the two operands.
   There are the following three cases to send the signal to the Controller.
3. (1) After several clocks, ADDER1 and ADDER2 finish the calculation at the same time, and store the calculated results into the registers. They also gives Controller signals (Done1 and Done2), which inform Controller that the calculation of the both module are finished simultaneously.
   (2) After a few clocks, ADDER1 finishes the calculation (ADDER2 is still running), and stores the calculated results in the registers. ADDER1 also gives Controller the signal Done1, which get the state forward to waiting2. And after more several clocks, ADDER2 finishes the calculation and gives the Controller the signal Done2.
   (3) After a few clocks, ADDER2 finishes the calculation, (ADDER1 is still running), and stores the calculated results in the registers. ADDER2 also gives Controller the signal Done2, which get the state forward to waiting1. And after more several clocks, ADDER1 finishes the calculation and gives the Controller the signal Done1.
4. Having received both of Done1 and Done2, Controller put its state forward to the next state (state2).
5. Controller continues the same action explained in the above items from 1 to 4.

5.4.2. Control over Testing Unit

Control over the Testing Unit is simpler than that of Arithmetic Unit because this unit outputs the result in the same clock that starts the test.

First, Controller transfers an instruction signal to the registers which hold the necessary regions or values to be tested. Second, Controller transfers the start signal to Testing Unit. Then last, Testing Unit gives back the result of the test to Controller. Controller decides whether to execute the next. This process is shown in Fig. 9.

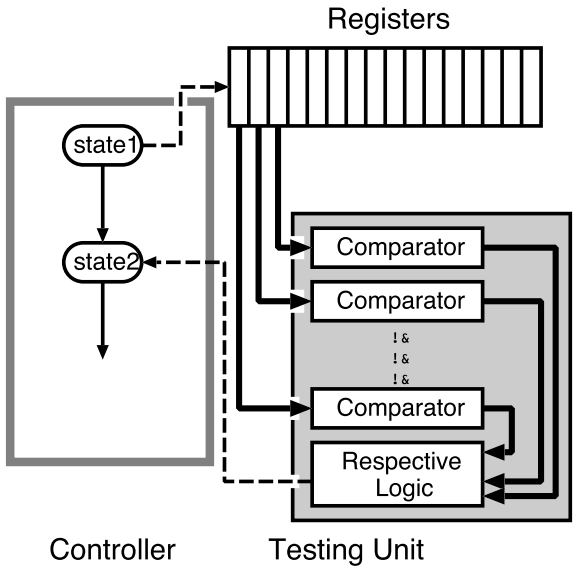

Fig. 9. Control over Testing Unit.

5.4.3. Series of controls

By executing the controls of arithmetic operation and related tests including Moore-test, it becomes possible to calculate $\boldsymbol{F}(\boldsymbol{X})$, $\boldsymbol{K}(\boldsymbol{X})$ and $\|\boldsymbol{I}-\boldsymbol{Y}\boldsymbol{F}'(\boldsymbol{X})\|$, and to examine the specified subregion.

The relations (1) $\boldsymbol{F}(\boldsymbol{X}) \not\ni \boldsymbol{0}$, (2) $\boldsymbol{K}(\boldsymbol{X}) \cap \boldsymbol{X} = \emptyset$, (3) $\boldsymbol{K}(\boldsymbol{X}) \subseteq \boldsymbol{X}$ and (4) $\|\boldsymbol{I} - \boldsymbol{Y}\boldsymbol{F}'(\boldsymbol{X})\| < 1$, are checked in time series, respectively.

As soon as the judgement of the relation (1) to (4) is done in the middle of the process, for example, in the check of relation (1), JudgeDone signal is generated, the result of the information is transmitted to CPU, and the state of Controller goes back to the initial state instead of continuing the rest of the process.

5.5. Example

As an example, we design the Moore-test processor for the following equation:

$$\begin{cases} f_1(x) = x_1^2 + x_2^2 - 1 = 0, \\ f_2(x) = x_1^2 - x_2 = 0. \end{cases} \tag{15}$$

The interval extension $\boldsymbol{F}(\boldsymbol{X})$ and $\boldsymbol{F}'(\boldsymbol{X})$ are as follows:

$$\boldsymbol{F}(\boldsymbol{X}) = \begin{pmatrix} X_1^2 + X_2^2 - 1 \\ X_1^2 - X_2 \end{pmatrix}, \quad \boldsymbol{F}'(\boldsymbol{X}) = \begin{pmatrix} 2X_1 & 2X_2 \\ 2X_1 & -1 \end{pmatrix}. \tag{16}$$

The arithmetic calculations $\boldsymbol{F}(\boldsymbol{X})$, $\boldsymbol{F}'(\boldsymbol{X})$ and $\boldsymbol{K}(\boldsymbol{X})$ are designed in the processor. Particularly the processes of designing $\boldsymbol{K}(\boldsymbol{X})$ is illustrated in Fig. 10. These figures show the order of calculations implemented in the processor. As some of the values are used repeatedly (e.g., $\boldsymbol{Y}$), the values must be computed in order.

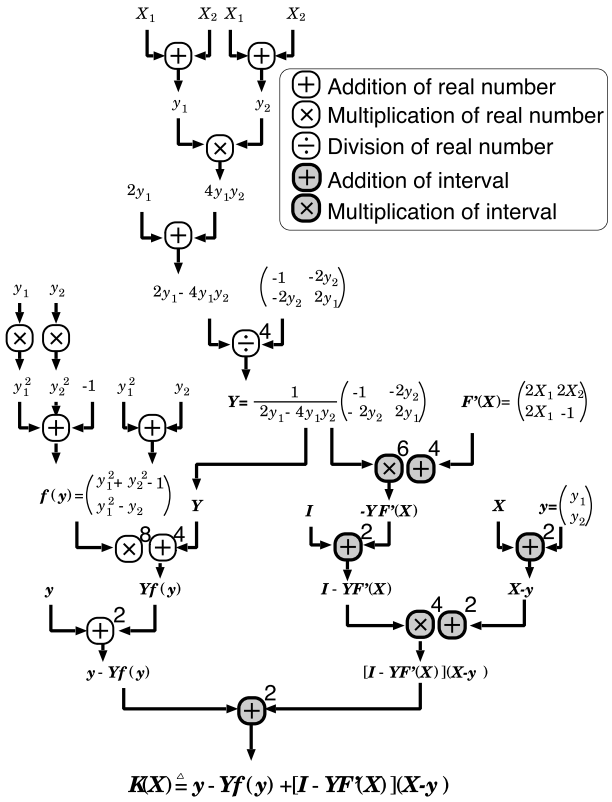

Fig. 10. Calculation of $\boldsymbol{K}(\boldsymbol{X})$. The power shows the number of operations.

Based on the number of LCs in FPGA, we decided to use 8 bits for exponent and 13 bits for the significand ($n = 8$, $m = 13$ shown in Table 4). Because 3 bits are used for sign bit, guard bit and rounding bit, the number of the total bits needed for each arithmetic operation are 16. The number of clock cycles needed to execute floating-point number operations are as follows: [adder] 8 clock cycles, [multiplier] 6 clock cycles, [divider] 6 clock cycles.

Table 4. Expression of floating point number.

exponent (8bits)	sign + significand (14bits)
excess 2^7	signed magnitude

The circuit of Moore-test processor is installed into the FPGA (EPF10K250 AGC599-1) with PCI interface circuit. The percentage of LCs used and the fastest frequency were as follows:

Percentage of LCs used 75 % (which corresponds to 9,134 LCs in EPF10K250 AGC599-1). PCI interface takes around 5 % in the percentage.

Fastest frequency 17.21 MHz.

It is confirmed that this system operated correctly as follows. Given the initial area $(X_1, X_2) = ([-2.00, 2.00], [-2.00, 2.00])$, this system outputs the two following regions as the region with a unique solution.

$$(X_1, X_2) = ([0.75, 1.00], [0.50, 0.75]), ([-1.00, -0.75], [0.50, 0.75]).$$

The process of finding the subregions that contain a unique solution is illustrated in Fig. 11. The number in the figure means the order of the tests of the region.

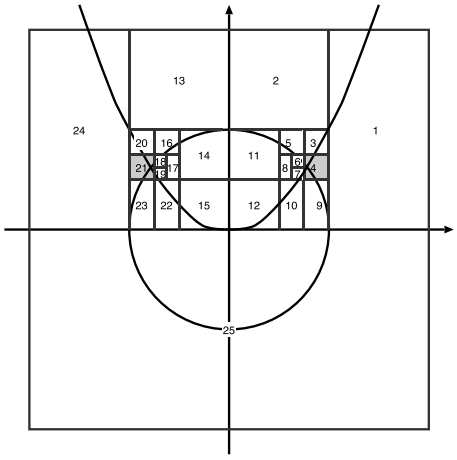

Fig. 11. Process of finding all solutions.

The hatched two subregions (No. 4 and No. 21) are the safe-starting regions from the midpoints of which Newton method starts on the side of computer to obtain the unique solution. This work was published in 2001. In recent research Gray code interval arithmetics are going to be implemented on FPGA [38, 40]. FPGA promises more improved computer technology of the KMJ algorithm.

6. Conclusion

The KMJ algorithm has been surveyed in so far as the applications to the field of electric and electronic circuit analysis. Although the KMJ algorithm is very powerful and attractive method, the researchers in this field who try to use this algorithm are rather few in Japan as well as in the world over. This may be because the existence of the KMJ algorithm is not well known in the society of engineering, although mathematicians who specialise in interval analysis know it. What must be done now is that we inform a lot of engineering researchers of the existence of this strong algorithm.

Acknowledgement. The parallel KMJ algorithm, the Gray code KMJ algorithm and the KMJ algorithm processor have been studied by a lot of members of the author's research group at Kyoto University. In particular, the author expresses his deepest gratitude to Associate Professor T. Hisakado and the doctor course graduate student A. Yonemoto for these researches. Further, he has been collaborated by many master course and undergraduate students at his research group, whom he also thanks so much for the programming and the numerical experiment.

References

[1] C. Hayashi, Nonlinear Oscillations in Physical Systems. McGraw-Hill, New York, 1964.
[2] T. Sunaga, Theory of an interval algebra and its application to numerical analysis. RAAG Memoirs, **2** (1958), 547–564.
[3] R.E. Moore, Interval Analysis. Prentice-Hall, Inc., Englewood Cliffs, 1966.
[4] R.E. Moore, Methods and Applications of Interval Analysis. SIAM, Philadelphia, 1979.
[5] G. Alefeld and J. Herzberger, Introduction to Interval Computations. Academic Press, New York, 1983.
[6] A. Neumaier, Interval Method for Systems of Equations. Cambidge University Press, 1990.
[7] L.V. Kolev, Interval Methods for Circuit Analysis. World Scientific Pub. Co., 1993.
[8] S. Markov and K. Okumura, The contribution of T. Sunaga to interval analysis and reliable computing. Developments in Reliable Computing, T. Cendes (ed.), Kluwer Academic Pub., 1999, 167–188.
[9] R. Krawczyk, Newton-Algorithm zur Bestimmung von Nullstellen mit Fehlershranken. Computing, **4** (1969), 187–201.
[10] R.E. Moore, A test for existence of solutions to nonlinear systems. SIAM J. Numer. Anal., **14** (1977), 611–615.
[11] R.E. Moore and S.T. Jones, Safe starting regions for iterative methods. SIAM J. Numer. Anal., **14** (1977), 1051–1065.
[12] R.E. Moore, A computational test for convergence of iterative methods for nonlinear systems. SIAM J. Numer. Anal., **15** (1978), 1194–1196.
[13] M.A. Wolfe, A modification of Krawczyk's algorithm. SIAM J. Numer. Anal., **17** (1980), 376–379.
[14] R.E. Moore, Interval methods for nonlinear systems. Computing Supple., **2** (1980), 113–120.
[15] E. Hansen and S. Sengupta, Bounding solutions of systems of equations using interval analysis. BIT, **21** (1981), 203–211.
[16] R.E. Moore and L. Qi, A successive interval test for nonlinear systems. SIAM J. Numer. Anal., **19** (1982), 845–850.
[17] L. Qi, A note on the Moore test for nonlinear systems. SIAM J. Numer. Anal., **19** (1982), 851–857.
[18] K. Okumura, S. Kagaya and A. Kishima, An algorithm for searching roots of nonlinear equation by interval analysis. Jour. of IECE, **J65-A** (1982), 1296–1297.

[19] L.V. Kolev, Finding all solutions of non-linear resistive circuit equations via interval analysis. Int. J. Cir. Theor. and Appl., **12** (1984), 175–178.
[20] J.M. Shearer and M.A. Wolfe, An improved form of the Krawczyk–Moore algorithm. Applied Math. Comp., **17** (1985), 229–239.
[21] K. Okumura, S. Saeki and A. Kishima, Improvement of algorithm using interval analysis for solution of nonlinear circuit equations. **69-A** (1986), 489–496.
[22] L.V. Kolev and V.M. Mladenov, An interval method for finding all operating points of nonlinear resistive circuits. Int. J. Cir. Theor. and Appl., **18** (1990), 257–267.
[23] K. Okumura, Several applications of interval mathematics to electrical network analysis. Kokyuroku No. 832, RIMA Kyoto University, 1993, 23–32.
[24] L.V. Kolev and V.M. Mladenov, An interval method for global non-linear DC circuit analysis. Int. J. Cir. Theor. and Appl., **22** (1994), 233–241.
[25] N. Femia, A robust and fast convergent interval analysis method for the calculation of internally controlled switching instants. IEEE Trans. CAS-I, Fundamental Theory and Applications, **43** (1996), 191–199.
[26] K. Okumura, Recent topics of circuit analysis, an application of interval arithmetic. J. of System Control Information Society of Japan, **40** (1996), 393–400.
[27] K. Yamamura, H. Kawata and A. Tokue, Interval solution of nonlinear equations using linear programing. BIT Numerical Mathematics, **38** (1998), 186–199.
[28] K. Yamamura and S. Tanaka, Finding all solutions of systems of nonlinear equations using the dual simplex method. BIT Numerical Mathematics, **42** (2002), 214–230.
[29] M.J. Sculte, Variable-precision, interval arithmetic processors. Application Specific Processing, Kluwer Academic Publishers, Boston, 1997, 1–28.
[30] M.J. Sculte and E.E. Swartzlander Jr., A family of variable-precision, interval arithmetic processors. IEEE Trans. on Computers, **49** (2000).
[31] D. Kuck, The Structure of Computers and Computations, Vol. 1. Wiley, New York, 1978, 33.
[32] T. Hisakado and K. Okumura, An approach to parallelization of Krawczyk's method. Jour. of ISCIE, **15** (2002), 495–501.
[33] F. Gray, Pulse code communications. U.S. Patent 2 632 058, March 17, 1953.
[34] P. Horowitz and W. Hill, The Art of Electronics, 2nd edition. Cambridge University Press, 1989.
[35] T. Hisakado, M. Hamada, A. Yonemoto and K. Okumura, Interval arithmetic using Gray code. Proc. MWSCAS, **3** (2004), 391–394.
[36] T. Hisakado and K. Okumura, Moore test using Gray code. IEEE Proc. ISCAS, 2005, 2803–2806.
[37] H. Tsuiki, Real number computation through Gray code embedding. Theoretical Computer Science, **284** (2002), 467–485.
[38] A. Yonemoto, T. Hisakado, M. Goto and K. Okumura, On-line arithmetic using Gray code and its FPGA implementation. Proc. ECCTD, **2** (2003), 317–320.
[39] T. Hisakado, T. Nishimura and K. Okumura, Hardware implementation of Krawczyk algorithm with FPGA. Tecnical Report of IEICE, NLP2001-43, 2001, 19–26.
[40] H. Nishimura, T. Hisakado and K. Okumura, Design and FPGA implementation of bit serial Gray code adder. IEICE, Annual Convension Report A-3-4, 2005, 69.
[41] D. Buell, T. El-Ghazawi, K. Gaj and V. Kindratenko, High-performance reconfigurable computing. Computer, IEEE Computer Society, **40** (2007).
[42] C. Piccardi, Bifurcations of limit cycles in periodically forced nonlinear systems—The harmonic balance approach. IEEE Trans. Circuits Syst. I, **41** (1994), 315–320.
[43] M. Basso, R. Genesio and A. Tesi, A frequency method for predicting limit cycle bifurcations. Nonlinear Dynamics, **13** (1997), 339–360.
[44] J.L. Moiola and G. Chen, Hopf bifurcation analysis, a frequency domain approach. Series on Nonlinear Science, Series A, Vol. 21, L.O. Chua (ed.), World Scientific, Singapore, 1996.
[45] F. Bonani and M. Gilli, Analysis of stability and bifurcations of limit cycles in Chua's circuit through the harmonic balance approach. IEEE Trans. Circuits Syst. I, **46** (1999), 881–890.
[46] V. Lanza, M. Bonnin and M. Gilli, On the application of the describing function technique to the bifurcation analysis of nonlinear systems. IEEE Trans. Circuits Syst. II, **54** (2007), 343–347.