Programmable Numerical Function Generators Based on Quadratic Approximation: Architecture and Synthesis Method

Shinobu Nagayama
Dept. of CE
Hiroshima City Univ.
Hiroshima 731-3194, Japan

Tsutomu Sasao
Dept. of CSE
Kyushu Inst. of Tech.
Iizuka 820-8502, Japan

Jon T. Butler
Dept. of ECE
Naval Postgraduate School
CA 93943-5121, USA

Abstract—This paper presents an architecture and a synthesis method for programmable numerical function generators (NFGs) for trigonometric, logarithmic, square root, and reciprocal functions. Our NFG partitions a given domain of the function into non-uniform segments using an LUT cascade, and approximates the given function by a quadratic polynomial for each segment. Thus, we can implement fast and compact NFGs for a wide range of functions. Implementation results on an FPGA show that: 1) our NFGs require only 4% of the memory needed by NFGs based on the linear approximation with non-uniform segmentation; and 2) our NFGs require only 22% of the memory needed by NFGs based on the 5th-order approximation with uniform segmentation. Our automatic synthesis system generates such compact NFGs quickly.

I. INTRODUCTION

Numerical function generators (NFGs) are often used in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics, etc. The functions realized include trigonometric, logarithmic, square root, and reciprocal functions. High-performance CPUs usually have numerical coprocessors. However, embedded CPUs and CPUs on FPGAs do not have such coprocessors. Thus, FPGA implementation of numerical functions \( f(x) \) is needed. Implementation by a single lookup table for \( f(x) \) is simple and fast. For low-precision computations of \( f(x) \) (e.g., \( x \) and \( f(x) \) have 8 bits), this implementation is straightforward. For high-precision computations, however, the single lookup table implementation is impractical due to the huge table size. For such applications, the CORDIC (COordinate Rotation Digital Computer) algorithm [1, 21] has been often used. Although CORDIC is implemented with compact hardware, it is iterative and therefore slow. For numerically intensive applications, faster evaluation of numerical function is required.

For fast evaluation of numerical functions, polynomial approximations have been used [9, 10, 19, 20]. These methods approximate the given numerical functions by piecewise polynomials, and realize the polynomials with hardware. Linear or quadratic approximations offer fast and relatively high-precision evaluation of numerical functions. However, the methods proposed so far are ad-hoc and not systematic. This paper proposes an architecture and a systematic synthesis method for NFGs based on quadratic approximation. By using the LUT cascade [8], many numerical functions are efficiently approximated by piecewise quadratic functions. Our synthesis method can be automated, so that fast and compact NFGs can be produced by non-experts. Fig. 1 shows the synthesis flow for the NFG. It converts the Design Specification described by Scilab [18], a MATLAB-like software, into HDL code. The Design Specification consists of a function \( f(x) \), a domain for \( x \), and an acceptable error. This system first partitions the domain into segments, and then approximates \( f(x) \) by a quadratic function for each segment. Next, it analyzes the errors, and derives the necessary precision for computing units in the NFG. Then, it generates HDL code to be mapped into an FPGA using an FPGA vendor tool. Due to the page limitation, the error analysis for our NFGs is omitted here, but it is available in [14]. This paper extends [17] to quadratic approximations.

II. PRELIMINARIES

**Definition 2.1** The binary fixed-point representation of a value \( r \) has the form

\[
d_{n\text{int}-1} \ldots d_1 d_0 \quad d_{-1} \ldots d_{-n\text{frac}},
\]

where \( d_i \in \{0, 1\} \), \( n\text{int} \) is the number of bits for the integer part, and \( n\text{frac} \) is the number of bits for the fractional part of \( r \). The representation in (1) is two’s complement, and so

\[
r = -2^n\text{int} - 1 d_1 + \sum_{i=-n\text{frac}}^{n\text{int}-2} 2^i d_i.
\]

**Definition 2.2** Error is the absolute difference between the original value and the approximated value. Approximation error is the error caused by a function approximation, and rounding error is the error caused by a binary fixed-point representation. Acceptable error is the maximum error that an NFG may assume. Acceptable approximation error (AAE) is the maximum approximation error that a function approximation may assume.
This paper presents an architecture and a synthesis method for programmable numerical function generators (NFGs) for trigonometric, logarithmic, square root, and reciprocal functions. Our NFG partitions a given domain of the function into non-uniform segments using an LUT cascade, and approximates the given function by a quadratic polynomial for each segment. Thus, we can implement fast and compact NFGs for a wide range of functions. Implementation results on an FPGA show that 1) our NFGs require only 4% of the memory needed by NFGs based on the linear approximation with non-uniform segmentation and 2) our NFGs require only 22% of the memory needed by NFGs based on the 5th-order approximation with uniform segmentation. Our automatic synthesis system generates such compact NFGs quickly.

15. SUBJECT TERMS

Programmable Numerical Function Generators Based on Quadratic Approximation: Architecture and Synthesis Method

16. SECURITY CLASSIFICATION OF:

<table>
<thead>
<tr>
<th>a. REPORT</th>
<th>b. ABSTRACT</th>
<th>c. THIS PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>unclassified</td>
<td>unclassified</td>
<td>unclassified</td>
</tr>
</tbody>
</table>
**Definition 2.3** Precision is the total number of bits for a binary fixed-point representation. Specially, \( n \)-bit precision specifies that \( n \) bits are used to represent the number; that is, \( n = n_{jnt} + n_{\text{frac}} \). An \( n \)-bit precision NFG has an \( n \)-bit input.

**Definition 2.4** Accuracy is the number of bits in the fractional part of a binary fixed-point representation. Specially, \( m \)-bit accuracy specifies that \( m \) bits are used to represent the fractional part of the number; that is, \( m = n_{\text{frac}} \). An \( m \)-bit accuracy NFG is an NFG with \( m \)-bit fractional part of the input, \( m \)-bit fractional part of the output, and a \( 2^{-m} \) acceptable error.

### III. Quadratic Approximation Algorithm

To approximate the numerical function \( f(x) \) using quadratic functions, first, we partition the domain for \( x \) into segments. For each segment, we approximate \( f(x) \) using a quadratic function \( g(x) = c_2 x^2 + c_1 x + c_0 \). In this case, the approximation error depends on the segmentation method and the values of coefficients \( c_2, c_1, \) and \( c_0 \) in the approximation polynomial.

For piecewise polynomial approximations, in many cases, the domain is partitioned into uniform segments [2, 6, 19]. Such methods are simple and fast, but for some kinds of numerical functions, too many segments are required, resulting in large memory.

For a given error, non-uniform segmentation of the domain uses fewer segments than the uniform segmentation [9, 17]. However, a non-uniform segmentation often requires a complicated segment index encoder (see Section IV), and results in larger and slower NFGs. To overcome this problem, a special non-uniform segmentation has been proposed [9]. This method produces a simple segment index encoder by restricting the segmentation points, and results in fewer segments as well as faster and more compact NFGs than produced by uniform segmentation. However, it is ad-hoc and non-optimum for the given function. Our NFG can implement any non-uniform segmentation with a fast and compact segment index encoder by using an LUT cascade [17] with a synthesis method that can be automated.

Selection of the approximation polynomial influences the number of non-uniform segments as well as the approximation error. In this paper, we use the 2nd-order Chebyshev approximation to approximate \( f(x) \) with fewer non-uniform segments, and compute the approximated value. Since coefficients of the Chebyshev approximation polynomial are easily computed, it is suitable for automatic synthesis.

#### A. Segmentation Algorithm

For a segment \([s, e]\) of \( f(x) \), the maximum approximation error \( \varepsilon_2(s, e) \) of the 2nd-order Chebyshev approximation [11] is given by

\[
\varepsilon_2(s, e) = \left( \frac{e - s}{192} \right)^3 \max_{s \leq x \leq e} |f^{(3)}(x)|,
\]

where \( f^{(3)} \) is the 3rd-order derivative of \( f \). From (2), \( \varepsilon_2(s, e) \) is a monotone increasing function of segment width \( e - s \). Using this property, we partition a domain into as wide segments as possible such that the approximation error is less than the specified error. Fig. 2 shows the non-uniform segmentation algorithm. The inputs for this algorithm are a numerical function \( f(x) \), a domain \([a, b]\) for \( x \), and an acceptable approximation error \( \epsilon \). Then, this algorithm approximates \( f(x) \) with the acceptable approximation error \( \epsilon \), and produces \( t \) segments \([s_0, e_0], [s_1, e_1], \ldots, [s_{t-1}, e_{t-1}]\). For step 2 in Fig. 2, the accurate computation of the value \( p \) where \( \varepsilon_2(s, p) = \epsilon \) is difficult. Thus, we obtain the maximum value \( p' \) satisfying \( \varepsilon_2(s, p') \leq \epsilon \). Such \( p' \) can be found by scanning values of \( n \)-bit input \( x \). However, it requires \( O(2^n) \) search, and is time-consuming. Therefore, we compute the maximum value \( p' \) by setting 0 or 1 from MSB to LSB of \( x \) such that \( \varepsilon_2(s, p') \leq \epsilon \). This requires \( O(n) \) search. In the computation of \( \varepsilon_2(s, p') \), the value of \( \max_{s \leq x \leq p'} |f^{(3)}(x)| \) is computed by the nonlinear programming algorithm, which is one of the most efficient [7].

#### B. Computation of Approximate value

For each \([s_i, e_i] \), \( f(x) \) is approximated by the corresponding quadratic function \( g_i(x) \). That is, the approximated value \( y \) of \( f(x) \) is computed as follows:

\[
y = g_i(x) = c_2 x^2 + c_1 x + c_0.
\]

where the coefficients \( c_2, c_1, \) and \( c_0 \) are derived from the 2nd-order Chebyshev approximation polynomial [11]. Substituting \( x = q - q_i \) for \( x \) in (3) yields the transformation:

\[
g_i(x) = c_2(x - q_i)^2 + (c_1 + 2c_2q_i)(x - q_i) + c_0 + c_1q_i + c_2q_i^2.
\]

In (4), let \( c'_1 = c_1 + 2c_2q_i \) and \( c'_0 = c_0 + c_1q_i + c_2q_i^2 \). Then, we have

\[
g_i(x) = c_2(x - q_i)^2 + c'_1(x - q_i) + c'_0.
\]

This transformation reduces the multiplier size.

### IV. Architecture for NFGs

Fig. 3 shows the architecture that realizes (5). It uses 7 units: the segment index encoder that computes the index \( i \) for segment \([s_i, e_i] \) including the input value \( x \); the coefficients table for \(-q_i, c_2, c'_1, \) and \( c'_0 \); the adder for \( x + (-q_i) \); the squaring unit; two multipliers; and the final adder.
Let segment index functions are realized by compact LUT cascades.

In LUT cascades, the interconnecting lines between adjacent LUTs are called rails. The size of an LUT cascade depends on the number of rails. The next theorem shows that the segment index functions are realized by compact LUT cascades.

**Theorem 4.1** [16] Let $\text{seg}_{\text{func}}(x)$ be a segment index function with $t$ segments. Then, there exists an LUT cascade for $\text{seg}_{\text{func}}(x)$ with at most $\lceil \log_2 t \rceil$ rails.

Our synthesis system uses heterogeneous MDDs (Multi-valued Decision Diagrams) [13] to find compact LUT cascades. Since the LUT cascade is suitable for the pipeline processing, it offers a fast and compact circuit. In Section VI, we will show that our architecture produces fast and compact NFGs for various numerical functions.

### V. IMPLEMENTATION WITH FPGA

Modern FPGAs consist of logic elements (LEs) or configurable logic blocks (CLBs), synchronous memory blocks, multipliers (DSP units), etc. Our synthesis system efficiently generates NFGs using these components. Each unit for the NFG shown in Fig. 3 is implemented by the following components in an FPGA: 1) Segment index encoder (LUT cascade) and coefficients table: by synchronous memory blocks; 2) Squaring unit: by logic elements; 3) Multiplier: by DSP units; and 4) Adder: by logic elements. Our synthesis system derives the appropriate bit-width for each component by automatic error analysis.

#### A. Size Reduction of Multiplier

Although modern FPGAs have dedicated multipliers, large multipliers are slow. In our architecture, the multiplier often has the longest delay time among all the units. Thus, to implement a fast NFG, reducing multiplier size is important. Since the size of multipliers depends on the number of bits for $c_{2i}$, $c_{1i}$, and $x-q_i$, it is important to reduce the number of bits to represent these values.

First, we consider the case where the absolute values of $c_{2i}$ and $c_{1i}$ are large. Our synthesis method uses a scaling method [9]. We represent $c_{2i}$ and $c_{1i}$ as $c_{2i} = c_{2i} \times 2^{-lb_1} \times 2^{lb_2}$ and $c_{1i} = c_{1i} \times 2^{-hb_1} \times 2^{hb_2}$, respectively. That is, instead of the original values of $c_{2i}$ and $c_{1i}$, we store the values of $c_{2i} \times 2^{-lb_2}$, $l_{22}$, $c_{1i} \times 2^{-hb_2}$, and $l_{11}$ in the coefficients table. In this case, the products $c_{2i}(x-q_i)^{2}$ and $c_{1i}(x-q_i)$ are computed using multipliers and shifters. The use of $l_{22}$ and $l_{11}$ reduces the number of bits to represent the values of $c_{2i} \times 2^{-lb_2}$ and $c_{1i} \times 2^{-hb_2}$, but increases the rounding errors. Our synthesis method finds optimum values of $l_{22}$ and $l_{11}$ for each segment such that an acceptable error is achieved. When $l_{22}$ and $l_{11}$ are 0 for all the segments, no shifter is implemented, that is, $c_{2i}(x-q_i)^{2}$ and $c_{1i}(x-q_i)$ are directly implemented with multipliers.

Next, we consider the value of $x-q_i$. The number of bits for $x-q_i$ influences the sizes of the squaring unit and multipliers. Thus, reducing the value of $x-q_i$ reduces the sizes of the squaring unit and multipliers, and also the error. From (5), we can choose any value for $q_i$. To reduce the value of $x-q_i$, for a segment $[s_i, e_i]$, we set $q_i = (s_i + e_i)/2$. Then, we have $|x-q_i| \leq (e_i - s_i)/2$. Thus, reducing the segment width $e_i - s_i$ reduces the value for $x-q_i$. However, this also increases the number of segments, and results in increased memory size. The rest of this section shows a reduction method of segment width without increasing the memory size.

The coefficients table in Fig. 3 has $2^k$ words, where $k = \lceil \log_2 t \rceil$ and $t$ is the number of segments. Therefore, we can increase the number of segments up to $t = 2^k$ without increasing the memory size. From Theorem 4.1, the size of LUT cascade also depends on the value of $k$. However, increasing the number of segments to $t = 2^k$ seldom increases the size of the LUT cascade. We reduce the size of segments by dividing the largest segment into two equal-sized segments up to $t = 2^k$. This method reduces both the number of bits for $x-q_i$ and the error without increasing the memory size.

#### B. Pipeline Processing

To implement a high-throughput NFG in an FPGA, our synthesis system inserts pipeline registers between all units in the architecture. Since all units operate in parallel, and each unit has a short delay time, our NFGs achieve high throughput. Table I shows the units and the number of pipeline stages for them. Our NFGs have $n_{\text{cas}}(5$ or 6) pipeline stages, where $n_{\text{cas}}$ is the number of LUTs for the LUT cascade.

---

**Fig. 3. Architecture for NFGs.**

**Fig. 4. Segment index encoder.**

A **segment index encoder** converts $x$ into a segment index $i$. It realizes the segment index function $\text{seg}_{\text{func}}(x) : B^n \rightarrow \{0, 1, \ldots, t - 1\}$ shown in Fig. 4 (a), where $x$ has $n$ bits, $B = \{0, 1\}$, and $t$ denotes the number of segments. In [9], to simplify the segment index encoder, the values of $s_i$ and $e_i$ are restricted to what can be produced by a simple combinational logic circuit. Such a segmentation method results in many segments since it does not adapt to the given function. Our synthesis system uses the LUT cascade [8, 15, 16] shown in Fig. 4 (b) to realize arbitrary $\text{seg}_{\text{func}}(x)$. It can be designed by functional decomposition using BDDs (Binary Decision Diagrams) representing $\text{seg}_{\text{func}}(x)$. Our synthesis system uses a non-strictive segmentation. It is suitable for automatic synthesis. In LUT cascades, the interconnecting lines between adjacent LUTs are called rails. The size of an LUT cascade depends on the number of rails. The next theorem shows that the segment index functions are realized by compact LUT cascades.

**Theorem 4.1** [16] Let $\text{seg}_{\text{func}}(x)$ be a segment index function with $t$ segments. Then, there exists an LUT cascade for $\text{seg}_{\text{func}}(x)$ with at most $\lceil \log_2 t \rceil$ rails.

Our synthesis system uses heterogeneous MDDs (Multi-valued Decision Diagrams) [13] to find compact LUT cascades. Since the LUT cascade is suitable for the pipeline processing, it offers a fast and compact circuit. In Section VI, we will show that our architecture produces fast and compact NFGs for various numerical functions.
VI. EXPERIMENTAL RESULTS

A. Number of Segments and Computation Time of Algorithm

Table II compares the number of segments for various approximation methods for the functions in [16]. In this table, Entropy, Sigmoid, and Gaussian are

\[
\text{Entropy} = -x \log_2 x - (1-x) \log_2 (1-x),
\]
\[
\text{Sigmoid} = \frac{1}{1 + e^{-4x}}, \quad \text{and} \quad \text{Gaussian} = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.
\]

In Table II, the columns “Linear Non” show the number of non-uniform segments for linear approximation in [17], and the columns “2nd-Chebyshev Uniform” and “2nd-Chebyshev Non” show the number of uniform segments and non-uniform segments for 2nd-order Chebyshev approximation, respectively. The columns “Time” show the CPU time for our non-uniform segmentation algorithm applied to functions, in milliseconds.

Table II shows that, for many functions, the 2nd-order Chebyshev approximations require many fewer segments than the linear approximation. However, for some functions, such as $\sqrt{-\ln(x)}$, the 2nd-order Chebyshev approximation based on uniform segmentation requires many more segments than the linear and 2nd-order Chebyshev approximations based on non-uniform segmentation. Many existing polynomial approximation methods are based on uniform segmentation. For trigonometric and exponential functions, approximation methods based on uniform segmentation require relatively few segments. However, for some kinds of functions such as $\sqrt{-\ln(x)}$, the uniform 2nd-order approximation method requires excessively many segments. On the other hand, our quadratic approximation based on non-uniform segmentation requires fewer segments for a wide range of functions. Also, Table II shows that the CPU time is strongly correlated to the number of segments. Smaller acceptable approximation error (AAE) requires more segments and longer computation time. However, Table II shows that, for all functions in the table, the CPU times are shorter than 1 second when the acceptable approximation error is $2^{-25}$.

These results show that, for various functions, our segmentation algorithm partitions a domain into fewer non-uniform segments quickly, and it is useful for automatic synthesis.

B. Memory Sizes of Various NFGs

This section compares the memory sizes of our NFGs with three existing NFGs [17, 3, 4]. Table III compares NFGs using linear approximation shown in [17]. This linear approximation is based on non-uniform segmentation. In Table III, the columns “R” show the following values:

\[
R = \frac{\text{memory size of quadratic approximation}}{\text{memory size of linear approximation}} \times 100.
\]

Table III shows that NFGs using quadratic approximation require much smaller memory than ones using linear approximation. Especially, 24-bit precision NFGs using quadratic approximation can be implemented with only 4% of the memory size needed for a linear approximation. From the relation between precision and memory size shown in Table III, we can see that increasing the precision decreases the ratio of memory sizes in NFGs.
TABLE III
Comparison with linear approximation based on non-uniform segmentation.

<table>
<thead>
<tr>
<th>Function</th>
<th>16-bit precision</th>
<th>24-bit precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linear</td>
<td>Quad.</td>
</tr>
<tr>
<td>$2^x$</td>
<td>20992</td>
<td>1172</td>
</tr>
<tr>
<td>$1/2^x$</td>
<td>21248</td>
<td>2432</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
<td>43776</td>
<td>5536</td>
</tr>
<tr>
<td>$\log_2(x)$</td>
<td>10316</td>
<td>1104</td>
</tr>
<tr>
<td>$\ln(x)$</td>
<td>20864</td>
<td>2464</td>
</tr>
<tr>
<td>$\sin(x)$</td>
<td>20096</td>
<td>2448</td>
</tr>
<tr>
<td>$\cos(x)$</td>
<td>19584</td>
<td>2336</td>
</tr>
<tr>
<td>$\tan(x)$</td>
<td>19712</td>
<td>2304</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
<td>74240</td>
<td>11264</td>
</tr>
<tr>
<td>$\tan(x) + 1$</td>
<td>37632</td>
<td>4960</td>
</tr>
<tr>
<td>Entropy</td>
<td>106496</td>
<td>10688</td>
</tr>
<tr>
<td>Gaussian</td>
<td>21120</td>
<td>2432</td>
</tr>
<tr>
<td>Gaussian</td>
<td>4416</td>
<td>444</td>
</tr>
</tbody>
</table>


Average: 31415 3704 11 1089095 45906 4

Table IV and Table V compare our NFGs with NFGs using 5th-order Taylor expansion [3] and NFGs using 2nd-order minimax approximation by the Remez algorithm [4], respectively. Both approximations in [3, 4] are based on uniform segmentation. Thus, their NFGs require no segment index encoder. On the other hand, since our approximation is based on non-uniform segmentation, the memory size is obtained by the sum of the coefficients table and the segment index encoder. As shown in [17] and Table II, for trigonometric and exponential functions, the difference of the number of uniform segments and non-uniform segments is not so large under the same approximation polynomial. For such functions, NFGs based on uniform segmentation (needing no segment index encoder) often require smaller memory than non-uniform segmentations. Although our NFGs require the segment index encoder and use approximation polynomials with larger approximation error than approximation polynomials in [3, 4], our NFGs for such functions are implemented with only 22% to 52% of the memory sizes of NFGs in [3], and with memory size comparable to [4]. In [3, 4], memory sizes of NFGs for $\sqrt{x}$ and $\sqrt{-\ln(x)}$ are unavailable. However, from Table II, we can see that the memory size of their NFGs for $\sqrt{x}$ and $\sqrt{-\ln(x)}$ is excessively large. On the other hand, our NFGs can realize a wide range of functions with small memory size.

C. FPGA Implementation Results

Table VI compares the FPGA implementation results of our NFGs with NFGs using linear approximation [17].

Since the architecture of linear NFG is simpler than quadratic NFG, linear NFGs are faster, and require fewer logic elements and DSP units than quadratic NFGs. However, linear approximation requires more segments and larger memory than quadratic approximation, as shown in Table II and Table III. Table VI shows that 24-bit precision linear NFGs cannot realize any function except Gaussian with the FPGA (the smallest device in the Stratix family) due to the excessive memory size although many logic elements and DSP units are unused. The most crucial issue in the FPGA implementation is the constraints on these hardware resources. For 24-bit precision, the linear approximation requires a larger FPGA due to the excessive memory size. However, in the larger FPGA, more logic elements and DSP units are left unused and wasted. On the other hand, the quadratic NFGs can be implemented with a smaller FPGA since they require much less memory size than the linear NFGs and reasonable sizes of logic elements and DSP units. In fact, 24-bit precision quadratic NFGs can be implemented with lower cost and more compact FPGAs (Cyclone II).

VIII. CONCLUSION AND COMMENTS

We have demonstrated an architecture and a synthesis method for programmable NFGs for trigonometric functions, logarithm functions, square root, reciprocal, etc. Our architecture can efficiently realize any non-uniform segmentation using a compact LUT cascade, and approximate many numerical functions by quadratic polynomials. Therefore, our architecture is suitable for automatic synthesis of fast and compact NFGs. Implementation results on an FPGA show that our synthesis method can approximate a wide range of functions with a small number of non-uniform segments, and generate NFGs with small memory size. For 24-bit precision, our NFGs can be implemented with only 4% of the memory size of NFGs based on the linear approximation with non-uniform segmentation, and with only 22% of the memory size of NFGs based on the 5th-order approximation with uniform segmentation. NFGs based on the linear approximation are faster than the quadratic ones, but for high-precision, they require a large FPGA due to the excessive memory size. On the other hand, our quadratic NFGs can be implemented with more compact and low-cost FPGA by using hardware resources on the FPGA efficiently.

ACKNOWLEDGMENTS

This research is partly supported by the Grant in Aid for Scientific Research of the Japan Society for the Promotion of

REFERENCES


