Approximation of Function Evaluation Over Sequence Arguments via Specialized Data Structures

Tamal T. Biswas*, Kenneth W. Regan*

Department of CSE, University at Buffalo, Amherst, NY 14260 USA

Abstract

This paper proposes strategies for maintaining a database of computational results of functions f on sequence arguments $\vec{x}$, where $\vec{x}$ is sorted in non-decreasing order and $f(\vec{x})$ has greatest dependence on the first few terms of $\vec{x}$. This scenario applies also to symmetric functions f, where the partial derivatives approach zero as the corresponding component value increases. The goal is to pre-compute exact values $f(\vec{u})$ on a tight enough net of sequence arguments, so that given any other sequence $\vec{x}$, a neighboring sequence $\vec{u}$ in the net giving a close approximation can be efficiently found. Our scheme avoids pre-computing the more-numerous partial-derivative values. It employs a new data structure that combines ideas of a trie and an array implementation of a heap, representing grid values compactly in the array, yet still allowing access by a single index lookup rather than pointer jumping. We demonstrate good size/approximation performance in a natural application.

Key words. Data structures, memoization, sequences, metrics, topology, machine learning, cloud computing.

1. Introduction

In many computational tasks, we need to evaluate a function on many different arguments, in applications such as aggregation where we can tolerate approximation. Evaluations $f(x)$ may be expensive enough to demand *memoization* of pre-computed values, creating a fine enough grid of argument-value pairs to enable approximating $f(x)$ via one or more neighboring pairs $(u, f(u))$. In this paper we limit ourselves to external memoization, here meaning building the complete grid in advance. Such applications of grids in high-dimensional real spaces $\mathbb{R}^\ell$ are well known (see history in Campbell-Kelly et al., 2003). What distinguishes this paper is a different kind of space in which the arguments are homogeneous *sequences* not just arbitrary vectors, where f and the space obey certain large-scale structural properties.

To explain our setting and ideas, consider first the natural grid strategy in $\mathbb{R}^\ell$ of employing the Taylor expansion to approximate $f(x)$ via a nearby gridpoint u:

$$f(x) = f(u) + \sum_{i=1}^{\ell}(x_i - u_i)\frac{\partial f}{\partial x_i}(u) + \frac{1}{2}\sum_{i,j}(x_i - u_i)(x_j - u_j)\frac{\partial^2 f}{\partial u_i \partial u_j} + \cdots$$

*Corresponding author. E-mail addresses: {tamaltan,regan}@buffalo.edu

If we make the grid fine enough, and assume that f is reasonably smooth, we can ignore the terms with second and higher partials, since $(x_i - u_i)(x_j - u_j)$ is quadratically small in the grid size. Doing this still requires knowledge of the gradient of f on the grid-points, however. Memoizing—that is, precomputing and storing—all the partials on the gridpoints might be ℓ times as expensive as memoizing the values $f(u)$. Hence, depending on the application, one may employ an approximation to the gradient or a recursive estimation of the partials.

In our setting, we are given a different kind of structure with a little more knowledge. Here we need to compute functions f on sequence arguments $\vec{x} = (x_1, x_2, x_3, \dots)$ under the following circumstances.

(a) The sequence entries and function values belong to $[0, 1]$.

(b) For all $i < j$ and $\vec{x}$, $\partial f/\partial x_i > \partial f/\partial x_j$ at $\vec{x}$.

(c) The sequences are non-decreasing, and for all i, $\partial f/\partial x_i$ becomes small as x_i approaches 1.

(d) While exact computation of $f(\vec{x})$ is expensive, moderate precision suffices, especially when there is no bias in the approximations.

Part (b) says that the initial terms have the highest influence on the result, while (c) together with (b) implies that as the sequence approaches its ceiling, terms lose their influence. Part (c) also allows us to assume all sequences have the same length ℓ, using 1.0 values as trailing padding if needed. Applications obeying (d) include calculation of means and percentiles and other aggregate statistics, as are typical for streaming algorithms (Muthukrishnan, 2005), and various tasks in curve fitting, machine learning (Michie, 1968), complex function evaluation, and Monte Carlo simulations (see Liang, 2009).

For further intuition, note that this setting applies to any function f that is symmetric, that is whose value is independent of the order of the arguments. Such a function really depends on the values of the arguments in a ranking structure. We may without loss of generality restrict the arguments to be sequenced by rank. Then (b) says that the first-ranked arguments matter most, while (c) says that elements with not only poor rank but also poor underlying scores have negligible marginal influence.

The goal is to build a data structure U of arguments and values $f(\vec{u})$ with these properties:

1. Coding: For all argument sequences $\vec{x}$ there are $\vec{u} \in U$ such that $|f(\vec{u}) - f(\vec{x})| < \epsilon$.

2. Size: $|U|$ is not too large, as a function of ϵ.

3. Efficiency: A good neighbor or small set of neighbors $\vec{u}$ can be found in time proportional to the length of the sequence, with only $O(1)$ further computation needed to retrieve the value $f(\vec{u})$.

4. Only a "black box" memo table of values $f(\vec{u})$ is needed, with the remainder of the approximation algorithm staying (essentially) independent of f.

Our main contribution is the construction of a family $\mathcal{G}$ of grid structures U and a simple memoizing algorithm A that is parameterized by a weighting function $wt(\cdots)$ used in lieu of the gradient. Among functions we consider are $wt(\cdots) = 0$, which means ignoring the gradient, and

$$wt(i, \vec{x}, -) = \frac{1}{i}(1 - x_i). \tag{1}$$

The intuition for this is that the first i elements have size no larger than x_i, which is to say equal or higher rank and influence on $f(\vec{x})$ to x_i. If they were all equal, then each would have a share $1/i$ of their total influence. The multiplier $(1 - x_i)$ aims to moderate the influence as the argument component x_i itself increases. It is also the simplest multiplier that goes to zero as x_i goes to 1. Our algorithm A uses weights in a balancing strategy that reflects the other sequence elements x_j for $j < i$.

We report success on a fairly general range of functions f that arise from a natural application in which probabilities are estimated. Some of the f represent salient basic mathematical problems in their own right. Key features of our data structure, algorithm, and overall strategy are:

- The grid is not regular but "warped": it starts fine but becomes coarse as i increases, eventually padding with nonce 1 values.
- The grid has an efficient compact mapping to an array, like a "warped" array implementation of a heap, so that values $f(\vec{u})$ can be looked up by one index rather than pointer jumping.
- The actual gradient of f is replaced by the universal substitute (1) and various other weighting method described later, which reflects properties (b) and (c) above.
- The algorithm to find a good grid neighbor $\vec{u}$ to the given argument $\vec{x}$ uses weights $w_i = wt(i, \cdots)$ to balance rounding.

This seems like a "cookbook" approach, but our point is that we have more structure to work with, before the steps of the recipe that have f as a particular ingredient need to be acted on. In our application there is no connection between the weights w_i and the functions f except for the axiomatic properties (b) and (c) and lack of unusual pathology in f. We demonstrate performance that is almost ten times faster than without memoization, and with four-place accuracy from a grid that starts with only two-place fineness. First we describe the application.

2. Estimating Probabilities and Means

Although our application comes from the chess decision-making model of (Regan and Haworth, 2011), the present computational task is simple and general and can be described without reference to chess. The main model-design parameter is a real function h, and varying h implicitly defines the functions f treated in this paper. The argument is a non-*increasing* sequence of ℓ-many real numbers a_i beginning with $a_1 = 1$. The goal is to fit ℓ-many probability values p_i according to the equations

$$\frac{h(p_i)}{h(p_1)} = a_i; \qquad 0 \le p_i \le 1, \quad \sum_{i=1}^{\ell} p_i = 1.$$

In modeling our grid, we transform $x_i = 1 - a_i$ so that the most influential values x_i as those closest to 0. The function value $f(\vec{x})$ is just the first probability, p_1. This is hence a fairly broad setting of curve-fitting. Our application further involves data with myriad sequences $\vec{x}$, for which we need to compute sums $M = \sum_{\vec{x} \in S} f(\vec{x})$ over sampled subsets S of the data, which when divided by $|S|$ become means. When S is moderately large, it suffices to have good approximation for f provided it is unbiased. Again this is highly typical in computational applications.

When h is the identity function id, we can solve for f in closed form:

$$f_{id}(\vec{a}) = \frac{1}{\sum_i a_i}, \quad \text{so} \quad f_{id}(\vec{x}) = \frac{1}{\ell - \sum_i x_i}.$$

For other functions h, however, we know only iterative techniques to compute $f(\vec{x}) = p_1$, and from p_1 the other probabilities, to desired precision. These iterations are expensive, so we seek to save them. When h is the logarithm function—more precisely the function $h(p) = 1/\log(1/p)$—we obtain the equations:

$$\frac{\log(1/p_1)}{\log(1/p_i)} = a_i, \quad \text{so} \quad p_i = p_1^{1/a_i}.$$

Viewing $b_i = 1/a_i$ as a general non-decreasing sequence, not necessarily beginning at 1, defines the following mathematical problem: Given y and real numbers $b_1, \ldots, b_\ell$, find a real number p such that

$$y = p^{b_1} + p^{b_2} + \cdots + p^{b_\ell}.$$

When $\ell = 1$ so there is just one number b, then $p = y^{1/b}$, so this is just the problem of root-finding. Hence we think of this problem as computing "vectorized roots" $\vec{\sqrt[b]{y}}$. Above we have the case $y = 1$ and also $b_1 = 1$.

To restore some of the intuition from chess, the x_i values are the perceived differences of various chess moves m_i in a chess position from the optimal move m_1. These are obtained from the differences $\Delta(m_1, m_i)$ given by analysis from a strong computer chess program, after factoring in model parameters representing the skill of a particular chess player P. The p_i are estimates for the probabilities that P will choose the respective moves, and in particular p_1 is the probability of finding the move the computer thinks is best. The value M becomes the expected number of agreements with the computer on the set S of moves. We imagine that for testing a small set of a few hundred moves one might pay the time for exact computation, but for *training* the model on sets of many tens of thousands, using sequences of values from 10–20 different levels of analysis on each move, explains the need for memoization. The better moves have x_i close to zero. If there are k such moves, then p_1 will be order-of $1/k$, and so a $(k+1)$st good move will have influence only at most about $1/(k+1)$. Moves with x_i close to 1 are "blunders," and the exact value of a blunder matters little in the phenomenon of player choice that we are modeling. Hence the axiomatic properties in the Introduction are fulfilled in an intuitive sense based on chess, but our point here is that they flow originally from a quite general mathematical system of equations.

3. The Tapered Grid Data Structures

When working with sequences, it is natural first to think of a *trie* data structure, that is a tree whose root branches to the possible first sequence elements (often restricted to those actually used), each of those nodes to possible second elements, and so on. When presented with a new sequence

$\vec{x}$, upon replacing entries x_i by ones in the domain if needed, one follows tree links until reaching a node for which $\vec{x}$ is known to be the unique sequence through it, whereupon $f(\vec{x})$ can be stored at that node, or the node is deep enough to store a good approximation. The main advantage of tries is flexibility to add new sequences and values dynamically and compactly, but this comes with high use of main memory and cache misses with pointer jumps.

We sacrifice the dynamism to store pre-computed values. This brings, however, the need to cover all possible next elements, not just those present in the dynamically built data. Since chess positions can have 50 or more legal moves we regard $\ell = 50$, so to store the result for all possible $\vec{x}$ with 2-decimal-place accuracy would involve powers of 101 that bust the information capacity of the universe. Hence our first task is to finalize the ordered vector $\vec{s}$ such that $u_i \in \vec{s}$ for all $\vec{u} \in U$ where i belong to $[1, \ell]$. Keeping in mind the constraint that $u_i \leq u_j \iff i \leq j$, the grid that we construct is one we call a self-similar tapered tree.

Definition 1. A *self-similar tapered tree* is a tree where every node has a branching label b, which is also its number of children. Its children have branches labeled $b, b-1, \ldots, 1$ going from left to right.

One more definition is useful in this context:

Definition 2. The *branching factor* of the grid is the maximum number of children any node can have at any depth. The branching factor of any depth is maximum number of children any node can have at that particular depth.

The branching factor of the root is the same as that of the grid, which is the length $|\vec{s}|$ of $\vec{s}$ in our case. The choice of elements in $\vec{s}$ and its length are implementation dependent, though we usually start with compact branches and later space out the gaps as the argument entries approach the ceiling 1.0.

Because the grid grows exponentially in size with increasing depth, we need to use at least one of the two compromises:

- Truncate: cut off the *depth* of the grid at some depth $d_0 \ll \ell$.
- Warp: reduce the *branching factor* geometrically as the depth of the grid d increases.

To demonstrate the ideas, we develop two schemes. The first idea is utilized by both of our schemes. For best exposition we implement the first before introducing the second. After choosing a fixed-spacing vector $\vec{s}$, we parameterize our first scheme by the maximum depth d and a fixed branching factor B at every depth. Our *reduced-branching* scheme replaces B by a specifier G of a rounded geometric progression to define the family $\mathcal{G}$ of grids $U = U(d, G)$, in which the omission of "ℓ" as a parameter is deliberate.

For our first scheme, once we finalize the branching factor b, the depth d and the spacing vector $\vec{s}$, we can generate all the sequences, where the root always contains the value 0.0. Of all the sequences, $(0.0, 0.0, 0.0, \ldots)$ is the infimum (in which all moves have equal value) and $(0.0, 1.0) \equiv (0.0, 1.0, 1.0, 1.0, \ldots)$ is the supremum. All other monotone sequences are totally ordered between them by prefix lexicographical order. We evaluate $f(\vec{u})$ for each of the sequences in the same order and store the output in a file, array or any sequential data structure that supports random access.

For our second implementation, we add the idea of "warping". The grid is built in such a way that, at every subsequent depth, the branching factor gets reduced by half. If the branching factor for the root is $2^n + 1$, then the maximum depth possible for such a grid is $n + 2$. The leaf nodes will contain values u_0 and $u_{|\vec{s}|-1}$.

It is easy to generate all the sequences in increasing order, evaluate the function and just store the evaluations in file. The real complication is calculating the location/index where the evaluation for the given vector is stored in the file.

Lemma 1. *For the grid with fixed branching factor B at every depth, given any vector $\vec{u} = (u_1, u_2, \ldots, u_d)$ where depth $d > 0$, we can find the index of $\vec{u}$ in $O(d)$ time.*

Proof. For performing the indexing, we first need to create a table. The table holds $V_{i,b}$, which is the number of nodes at any depth i for any core branch b. Core branches are branches generated from the root, while i and b index the corresponding row and column of the table respectively. The constructed table has d rows and B columns. The entries for the table for the first implementation can be generated using equation(2).

$$V_{i,b} = \begin{cases} 0 & \text{if } i = 1 \\ 1 & \text{if } i = 2 \\ \sum_{j=b}^{B} V_{i-1,j} & \text{otherwise} \end{cases} \tag{2}$$

or

$$V_{i,b} = \begin{cases} 0 & \text{if } i = 1 \\ 1 & \text{if } i = 2 \\ V_{i-1,b} + V_{i,b+1} & \text{if } (b < B) \wedge (i > 2) \\ V_{i-1,b} & \text{otherwise,} \end{cases}$$

The creation of the table, which is a one-time event, takes $O(Bd)$ time. Once the table is generated, for calculating the index for the vector $\vec{u} = (u_1, \ldots, u_d)$, we first calculate the position of u_i where $i \in \{1, 2, \ldots, d\}$ in the table. We can define a mapping function $h : u_i \to m_i$ where m_i is the corresponding column in the grid for u_i. Then the index would be:

$$\textit{index} = \sum_{i=1}^{d-1} \sum_{j=h(u_i)}^{h(u_{i+1})-1} V_{d+1-i,j}. \tag{3}$$

Calculating the index requires summing $O(Bd)$ terms. By generating a second table that stores the partial sum $\sum_{j=0}^{b} V_{i,j}$ for every b and i, we can perform the inner sum operation of equation (3)in constant time. This makes the whole indexing take $O(d)$ time. □

The fixed branching factor algorithm is well suited for most of the application. But in cases where the precision of evaluating f is most crucial for the initial values of the vector $\vec{u}$, we use the "warping" concept. A warped grid of depth d has initial branching factor $2^d + 1$ and at each subsequent depth, the branching factor gets reduced by half.

Lemma 2. *For a grid with exponentially reducing branching factor B, given any vector $\vec{u} = (u_1, u_2, \ldots, u_d)$ and depth $d > 0$, we can find the index of the sequence in $O(d)$ time.*

Proof. As with the grid for fixed branching, for preforming indexing we first need to generate a table. The constructed table has d rows and B columns. The entries for the table for the second implementation can be generated using equation (4).

$$V_{i,b} = \begin{cases} 0 & \text{if } i = 1 \\ 1 & \text{if } i = 2 \\ V_{i-1,b} + V_{i,b+2^{i-2}} & \text{if } (b + 2^{i-2} \leq B) \wedge (i > 2) \\ V_{i-1,b} & \text{otherwise,} \end{cases} \tag{4}$$

where i ranges from 1 to d, and $V_{i,b}$ is the number of nodes at depth i and for core branch index b. The generation of the table requires $O(Bd)$ time.

The indexing for this scheme is different than that of our first scheme. If the mapping function is $g : (u_i, k) \to m_{i,k}$ we can evaluate $m_{i,k}$ using equation (5).

$$m_{i,k} = \begin{cases} 0 & \text{if } (h(u_i) - 1) \bmod 2^{k-1} \neq 0 \\ B - (B-1)/2^{k-1} + (h(u_i) - 1)/2^{k-1}) & \text{otherwise,} \end{cases} \tag{5}$$

Here the function h is the mapping function used for fixed branched grid and k represents the depth for which the index is sought. while $m_{i,k} = 0$ indicates that the corresponding u_i is not present for the particular depth. This information can be stored for future lookup. The modified indexing function for this reduced branched grid is:

$$\textit{index} = \sum_{i=1}^{d-1} \sum_{j=g(u_i,i)}^{g(u_{i+1},i)-1} V_{d+1-i,j} \tag{6}$$

Again, by the same procedure as described for fixed branching, we can calculate the corresponding index for any vector in $O(d)$ time.

□

We can further specialize the scheme by reducing the branches only at specific depths. This gives us better control on the overall size and precision of the grid.

Lemma 3. *For a grid with initial branching factor B with a binary vector $\vec{r}$ of size d indicating reduction in branches at any depth, given any vector $\vec{u} = (u_1, u_2, \ldots, u_d)$ and depth $d > 0$, we can find the index of the sequence in $O(d)$ time.*

Proof. Like the earlier two approaches, we first need to construct a table for lookup. The table needs to be constructed recursively from depth $j = d$ to 1 due to the variable nature of $\vec{r}$. The intermediate rows in the table will be rewritten for each j. The constructed table has d rows and B columns. The first two values in $\vec{r}$ are 0 which indicates no reduction in branches while the rest of the values can be either 0 or 1 indicating no reduction and reduction in branches respectively. The entries for the table for this implementation can be generated using equation (7) for any intermediate depth i and final depth j where $R(i,j) = \sum_{k=d-j+1}^{d-j+i} r_k$.

$$V_{i,b,j} = \begin{cases} 0 & \text{if } i = 1 \\ 1 & \text{if } i = 2 \\ V_{i-1,b,j} + V_{i,b+2R(i,j)} & \text{if } (b + 2R(i,j) \leq B) \wedge (i > 2) \\ V_{i-1,b,j} & \text{otherwise,} \end{cases} \tag{7}$$

where i ranges from 1 to j, and $V_{i,b,j}$ is the number of nodes at intermediate depth i and for branch index b for the purpose of generating row j of the table . Once the table is created we can ignore the j parameter for indexing purpose. Due to the recursive nature of creation, the generation of the table requires $O(Bd^2)$ time in comparison to $O(Bd)$ time required for earlier two implementation. After the construction of the table the mapping function $g : (u_i, k) \rightarrow m_{i,k}$ can be evaluated using equation (8) where $R(k) = \sum_{j=1}^{k+1} r_j$.

$$m_{i,k} = \begin{cases} 0 & \text{if } (h(u_i) - 1) \bmod 2^{R(k)} \neq 0 \\ B - B/2^{R(k)} + (h(u_i) - 1)/2^{R(k)}) & \text{otherwise,} \end{cases} \tag{8}$$

The indexing scheme is the same as *reduced-branching* implementation.

□

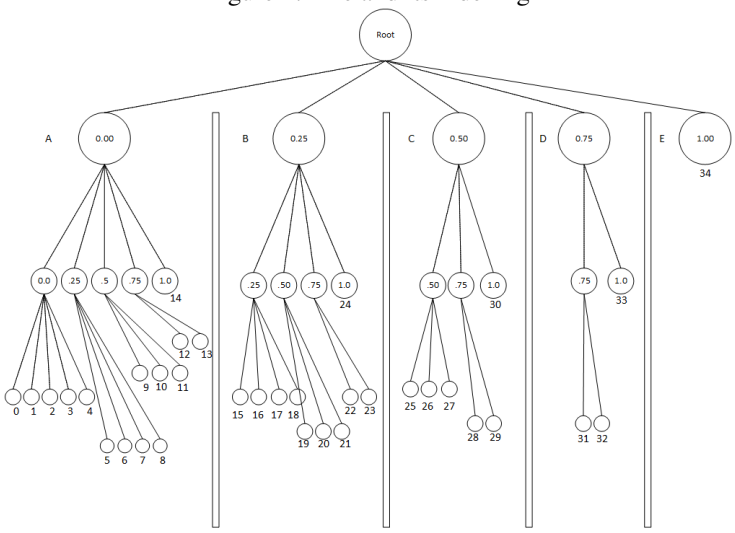
Example of Trie and its Indexing

Figure 1 shows a pictorial representation of a trie and its indexing scheme. The figure represents an example of fixed branching grid with branching factor 5. The branches are shown in table 1. The trie contains 35 entries indexed from 0 to 34 where vector (0.00, 0.00, 0.00, 0.00) and (0.00,1.00) represent index 0 and 34 respectively.

Table 1: Core Branches

1	2	3	4	5
0.00	0.25	0.50	0.75	1.00

Figure 1: Trie and its indexing



Example of Fixed Branching Scheme

Example 1. The example we illustrate uses a grid with branching factor 17. The selection of branches is shown in table 2. Table 3 was generated using equation (2).

Table 2: Core Branches

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90	1.00

Table 3: Fixed Branching Factor

Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Depth 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Depth 2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Depth 3	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Depth 4	153	136	120	105	91	78	66	55	45	36	28	21	15	10	6	3	1
Depth 5	969	816	680	560	455	364	286	220	165	120	84	56	35	20	10	4	1

For our example, we suppose that the initial vector, after interpolation onto the grid, gives the vector $\vec{u} = (0.00, 0.00, 0.02.0.04, 0.80)$. Then the mapping function h maps $\vec{u}$ to the index vector $\vec{m}$ which has the corresponding column for u_i in table 2. The generated mapping is shown in table 4.

Using table 3 for reference which stores all $V_{i,b}$ entries , we can calculate the index using equation (3). Table 5 represents the break-up of the equation (3) which evaluates the index as 328 (zero-based indexing). Seeking the 328^{th} element from the file gives the evaluation for the vector.

Example of Selective Branching Scheme

Example 2. The example we illustrate uses a grid with branching factor 17 and depth 5. We assume branches are equally spaced between 0 and 1 and reduction of branches only happen at depth 4.

Table 6 shows the core branches for Grid where table 7 was generated using equation (7). For our example, we suppose that the initial vector, after interpolation onto the grid, gives the vector $\vec{u} = (0.00, 0.0625, 0.25.0.50, 0.75)$. We use equation (8) to generate table 8. Finally we calculate the zero based index for selective reducing using equation (6). The breakup of the calculation is shown in table 9.

4. Interpolation Algorithm

Although the members of U are totally ordered lexicographically, the values $f(\vec{u})$ are generally *not* monotone in this ordering, and this makes the neighborhood topology play havoc when given

Table 4: Mapping from Value to Index

Value(u_i)	Index ($h(u_i) = m_i$)
0.00	1
0.00	1
0.02	3
0.04	5
0.80	15

Table 5: Index Calculation

Depth	Table Row	From	To	Sum
1	5	1	1	0
2	4	1	3	153+136
3	3	3	5	15+14
4	2	5	15	1+1+1+1+1+1+1+1+1+1
Sum-total				328 (zero-based index)

Table 6: Core Branches for Selective Reducing

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0.0625	0.125	0.1875	0.25	0.3125	0.375	0.4375	0.5	0.5625	0.625	0.6875	0.75	0.8125	0.875	0.9375	1

a non-grid sequence $\vec{x}$. For instance, if the spacing starts $0.00, 0.02, 0.04, \ldots$ in the first few index places $i \geq 2$, then the sequence

$$\vec{x} = 0.00, 0.01, 0.20, 0.40, 0.60, 0.80, 1.0 \ldots$$

has as its *lower* neighbor the sequence $0.00, 0.00, 1.0, 1.0 \ldots$, which generally gives much *higher* p_1, and its *upper* neighbor the sequence $0.00, 0.02, 0.02, 0.02, 0.02 \ldots$ in which all moves are close to equal and p_1 has nearly its minimum possible value, which is $1/\ell$. Thus attempting to use the neighbors in the *trie* structure of the sequences is bad. Instead, given (any) $\vec{x}$, we define its *component-wise bounds* x^+ and x^-. In this case, assuming the spacing continues $\ldots, 0.20, 0.35, 0.50, 0.75, 1.0$, we get:

$$\begin{aligned} \vec{x}^+ &= 0.00, 0.02, 0.20, 0.50, 0.75, 1.0, 1.0 \ldots \\ \vec{x}^- &= 0.00, 0.00, 0.20, 0.35, 0.50, 0.75, 1.0 \ldots \end{aligned}$$

Since these are members of U, it is plausible to output some weighted average of $f(\vec{x}^+)$ and $f(\vec{x}^-)$. We mix this idea with interpolating to find a better argument $\vec{u}$ between $\vec{x}^+$ and $\vec{x}^-$. For this purpose we use the spacing vector $\vec{s}$, which contains all possible points in ascending order, ending with the maximum, 1.0. We also let $wt(i, \vec{x}, j, \vec{s})$ stand for a weighting function which can be the aforementioned $(1 - x_i)/i$ gradient or might reflect $\vec{s}$ and its index j as well.

Our procedure INTERPOLATE$(\vec{x}, \vec{s}, d; \vec{u})$ creates $\vec{u}$ of depth d by first truncating $\vec{x}$ to length d (or padding $\vec{x}$ with extra 1.0 values if its length is $< d$), initializing j and "credit" c to 0, and then executing the following loop for $i = 1$ to d:

Table 7: Selective Branching Table

Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Depth 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Depth 2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Depth 3	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Depth 4	45	36	36	28	28	21	21	15	15	10	10	6	6	3	3	1	1
Depth 5	285	240	204	168	140	112	91	70	55	40	30	20	14	8	5	2	1

Table 8: Selective Mapping

Depth	From Value(c_i)	From Index ($g(c_i, i)$)	To Value(c_{i+1})	To Index ($g(c_{i+1}, i)$)
1	0.0	1	0.0625	2
2	0.0625	2	0.25	5
3	0.25	11	0.50	13
4	0.50	13	0.75	15

Table 9: Calculation of Index

Depth	Table Row	From	To	Sum
1	5	1	2	285
2	4	2	5	36+36+28
3	3	11	13	7+6
4	2	13	15	1+1
SumTotal				400 (zero based index)

1. while ($\vec{s}[j+1] < \vec{x}[i]$) do $j := j + 1$; //so $\vec{x}[i] \leq \vec{s}[j+1]$
2. let $a = (\vec{s}[j] + \vec{s}[j+1])/2$;
3. if $a > \vec{x}[i] + c$ then do $\vec{u}[i] := \vec{s}[j]$ and $c := c + wt(i, \vec{x}, j, \vec{s}) * (\vec{x}[i] - \vec{s}[j])$;
4. else do $\vec{u}[i] = \vec{s}[j+1]$; $j := j + 1$; and $c := c - wt(i, \vec{x}, j, \vec{s}) * (\vec{s}[j+1] - \vec{x}[i])$;

If we round down in step 3, then higher c may influence us to round up in the next iteration. Once we round up, however, the increment in j makes $\vec{s}[j+1]$ the new floor, so we can never round down to a lower value. The $\vec{x}[i]$ values may stay below this floor—then the while-loop in step 1 does nothing, and negative adjustments in c will prevent further rounding up unless $\vec{x}[i']$ itself increases for some $i' > i$. The algorithm runs in $O(d)$ time since j never backtracks. A modified version of this interpolation algorithm can be used for reduced and selective branching implementation where at any depth $\vec{s}$ may have half the elements that its earlier depth in case of reduction of branches at that depth. In that case, we need to replace j by $j/2$. An additional check is required to make sure that for every i, $\vec{u}[i] \geq \vec{u}[i-1]$.

5. Experimental Results

We implemented all of the schemes in C++ with highest optimization on a shared Red Hat Enterprise Linux Server release 5.7 (Tikanga) 32GB 64-bit system configured for non-interactive, CPU-intensive and long-running processes. Our main tests employed the vector-root function from Section 2 as f. The core branches used for the fixed branched grid are shown in table 10. For this scheme, we set the branching factor to 16 and the depth to 15. We generated the evaluation for every possible vector using the core-branch values, and stored the evaluations in a file. The generated file had a total of $77,558,760$ entries, starting from the evaluation for a vector of all 0's to the entry for the length-2 vector $(0, 1)$.

For testing the performance of our scheme, we used 8,000 random vectors $\vec{x}$. The distribution was controlled by a parameter b governing the expected time to hit the 1.0 ceiling—in chess terms

Table 10: Core branches used for fixed-branch grid

Index	1	2	3	4	5	6	7	8	9	10	11	14	13	14	15	16
Value	0.0	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.60	0.70	0.80	0.90	1.0

it expressed how many reasonable (non-blunder) moves to expect in a position. This was done by initializing $p = 0$, making $x_1 = 0$ the first entry, and for each iteration, generating a uniformly distributed random number r between p and 1. Then we select the next element of the vector as $p + \frac{r-p}{b}$ and update p to the same value. We have tested our implementation for various b values ranging from 3 to 6.

For each of those 8,000 vectors, we first calculated the exact vector-root using Newton's method and then calculated the closest neighbor of that vector using various interpolation algorithm (refer Section 4) and found the corresponding index for that vector. Finally we accessed the file to fetch the evaluation at that location. For interpolation, we used various weighting and mapping policies. The details of the various interpolation schemes are as follows:

1. The *simple nearest neighbor* (NN) strategy follows the algorithm INTERPOLATE with $wt(\cdots) = 0$. Each value in the input vector was hence matched to the nearest grid point, subject to the monoticity requirement.

2. The *universal gradient* (UG) strategy sets the weight to $(1 - x_i)/i$, per equation (1).

3. The *gridpoint weight* (GW) strategy sets the weight to $wt(i, \vec{x}, j, \vec{s}) = (1-s_j)(1-x_i)$, where s_j is the nearest gridpoint to which x_i is mapped.

4. The *simple lower bound* (LB) policy maps every element of the vector to the nearest grid value which is smaller or equal to x_i . The generated vector is the pointwise lower bound for $\vec{x}$.

5. The *simple upper bound* (UB) policy maps every element of the vector to the nearest grid value which is equal or greater than x_i. The generated vector is the pointwise upper bound for $\vec{x}$.

The last two furnish comparisons to show that cavalier interpolation produces significantly worse results.

Table 11 presents the average deviation from the exact vector-root computation for the various strategies and values of b. The results show that each of the first three schemes gives good approximation, but there is no clear winner, rather the best performance depends on the distribution of data.

Figure 2 shows a scatter plot of the deviation from the true values of the vector root function where b was set to 5 and the UG strategy was used.

We ran similar tests for reduced-branching implementation. The branching factor of the grid was set to 257, where each branch was equally spaced between 0 and 1, and the depth of the grid was 10. For interpolation we used 'NN' strategy. We tested the implementation with $8,000$

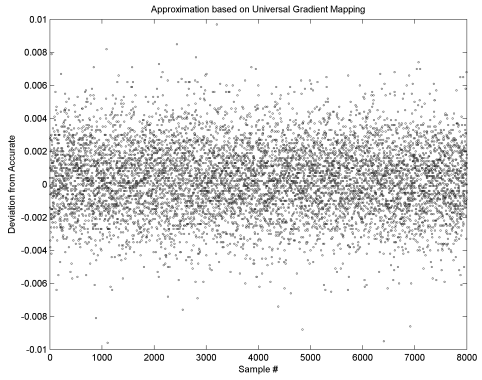
Table 11: Comparison between interpolation algorithms

$b =$	3	4	5	6
NN	0.0010	0.0007	0.0004	-0.0003
UG	0.0009	0.0007	0.0002	-0.0007
GW	0.0008	0.0003	-0.0008	-0.0023
LB	-0.2284	-0.2711	-0.2993	-0.3209
UB	0.1871	0.1487	0.1221	0.1011

Table 12: Comparison between interpolation algorithms for selective branching

$b =$	3	4	5	6
NN	0.0004	0.0007	0.0009	0.0005
UG	0.0004	0.0006	0.0008	0.0004
GW	0.0003	0.0004	0.0005	0.0002
LB	-0.0131	-0.0142	-0.0150	-0.0154
UB	0.0158	0.0172	0.0173	0.0163

Figure 2: Performance analysis for the fixed-branch grid



iterations. Figure 3 shows the closeness of fit. The average deviation for the scheme from the true vector-root value was -0.0008, where the standard deviation was 0.0052. For selective branching implementation, we set the branching factor and depth to 33 and 20 respectively. The branches were equally spaced, and reduction of branches occurred at depth $5, 9, 17$.

From Table 12 we observe the selective reducing scheme can produce output very close to accurate , and among all interpolation algorithm, 'gridpoint weight' strategy works better.

Figure 4 represents the histogram for deviation from actual calculation of vector root. The bias parameter 'b was set to 6 to generate the random vectors used for the histogram.

On an average, the execution time for any scheme was around 10 times faster than the real-time vector-root evaluation.

6. Conclusions and Future Work

In this paper we have developed a special purpose data structure for storing sample points of a function f so that the values of f at other points can be interpolated efficiently. This data structure can be used to store the result of computation intensive function values for faster remote computing.

Figure 3: Approximation performance analysis for the reduced-branched grid

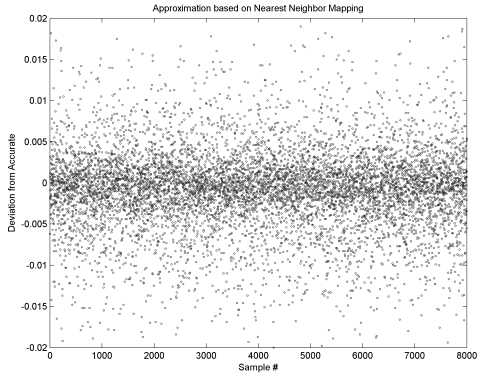
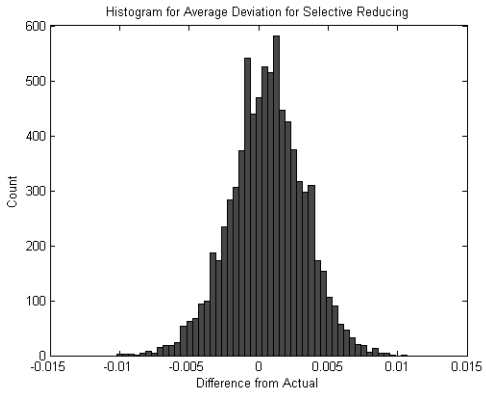


Figure 4: Performance analysis for the selective branched grid



Our main concern is not space efficiency (Khoshnevisan and Afshar, 1996; Khoshnevisan, 1990) rather faster retrieval of function evaluation. As the function is not evaluated in real time, this suits the requirement of embedded system, where both the processing power and conservation of energy is vital (Alidina et al., 1994). Along with these benefits, this data structure is around 10 times faster and provides good approximation in comparison to the real-time evaluation of various functions.

Alidina, M., Monteiro, J. C., Devadas, S., Ghosh, A., Papaefthymiou, M. C., 1994. Precomputation-based sequential logic optimization for low power. IEEE Trans. VLSI Syst. 2 (4), 426–436.

Campbell-Kelly, M., Croarken, M., Flood, R., Robson, E., 2003. The History of Mathematical Tables. Oxford University Press, USA.

Khoshnevisan, H., 1990. Efficient memo-table management strategies. Acta Informatica 28 (1), 43–81.

Khoshnevisan, H., Afshar, M., 1996. Space-efficient memo-functions. Journal of Systems and Software 35 (1).

Liang, F., 2009. Stochastic approximation Monte Carlo for MLP learning. In: Encyclopedia of Artificial Intelligence. Wiley, pp. 1482–1489.

Michie, D., 1968. Memo functions and machine learning. Nature 218 (5136), 19–22.

Muthukrishnan, S., 2005. Data Streams: Algorithms and Applications. Now Publishers.

Regan, K., Haworth, G., 2011. Intrinsic chess ratings. In: Proceedings of AAAI 2011, San Francisco.