千葉商大紀要　第 59 巻　第 1 号（2021 年 7 月）pp. 53-65

〔論　説〕

DiffMA: A Lossy Encoding of Motion Capture Data for JSON

Take-Yuki NAGAO

1. Introduction

This paper aims to provide a mathematical method for eliminating redundancy from the recorded motion capture data and making it possible to store various high-frequency floating point data efficiently in JSON (JavaScript Object Notation) format. In general, motion capture data consist of skeletal information describing the structure of human bones and time series information representing the transformation matrices for skeletal motion. The time series data are lists or arrays of floating point numbers. When transmitting motion data over the network, these floating point numbers need to be encoded efficiently.

To this end, a lossy compression algorithm named DiffMA is proposed below to encode and decode an arbitrary list of floating point numbers into a list of opcodes suitable for ASCII representation. The proposed algorithm is intended for the real number field, which is essential for mathematical theories, and for floating point numbers, which are more complicated but required for implementing applications.

2. DiffMA Algorithm

2.1. **Redundancy of a list.** Given a list $x = [x_j]$ indexed by a set of non-negative integers, a typical redundancy occurs when the element x_j is constant for all j. If we know beforehand that x_j is constant, we can encode x as the single value x_0. Similarly, if x_j is linear in j, viz. $x_j = \alpha j + \gamma$ for some α and γ, then we can encode x as the list $[x_0, x_1]$. This paper aims to automate this kind of encoding by an algorithm that is implementable by computers.

When we use natural languages, we may encode a constant list $[x_0, x_0, \ldots, x_0]$ of length N as $([x_0], \text{knowledge}_1)$, where

$$\text{knowledge}_1 = \text{“the original list is constant and has length } N\text{”},$$

and a linear list $x = [x_j]$ as $([x_0, x_1], \text{knowledge}_2)$, where

$$\text{knowledge}_2 = \text{“the original list is linear and has length } N\text{”}.$$

It is hence reasonable to find a method to encode a given list x as a tuple (t, b) of sublist t of x and some data b representing additional knowledge about the original list x. For the constant case, $t = [x_0]$ and $b = \text{knowledge}_1$. For the linear case, $t = [x_0, x_1]$ and $b = \text{knowledge}_2$. A non-trivial problem is to find a representation of the knowledge b as a mathematical object that is implementable by computers. The main idea of this paper is to design a simple set of opcodes to represent the knowledge b as a list of opcodes. We refer to the sublist t as a look-up table, which contains some of the exact values of the original list. Our goal is to minimize the size of the look-up table for a space-efficient representation of the original data.

Date: 20 May, 2021.

2.2. **Designing Opcodes.** This paper focuses on creating a lossy compression algorithm that takes an arbitrary list $[x_j]$ of real numbers as input and outputs a list of opcodes listed in Table 1 along with a look-up table.

Table 1. List of Opcodes

Opcode	Meaning
`EQL`	Repeat the last exact value.
`NEG`	Flip the sign of the difference of the list.
`APX`(q)	Approximate the value using the quantized value q.
`RAW`	Load the raw value from the look-up table.

We now discuss how to encode a given list efficiently by using the above opcodes. For inferring knowledge, such as constancy and linearity, from a given list $x = [x_j]$ of real numbers, it is natural to look at its difference $x_{j+1} - x_j$ (for $j = 0, 1, 2, \ldots$) and check if it is constant for all j. However, this approach has a disadvantage because it is hard to reconstruct the original list $[x_j]$ from its difference $[x_{j+1} - x_j]$ due to rounding errors when we use floating point numbers. Errors in the difference for multiple j's may accumulate to make a significant difference to the reconstructed values obtained by summation — the reconstructed value in this paper means the value obtained by decoding the encoded value.

To mitigate the above type of errors, we will slightly modify the definition of the difference. When we have encoded the list $[x_j]$ up to the element x_j, we decode the last encoded value. We refer to the reconstructed value as x'_j, which might be different from x_j due to rounding errors. When encoding the next value x_{j+1}, we encode the difference $x_{j+1} - x'_j$ instead of encoding $x_{j+1} - x_j$ directly. The point is that the exact value of x'_j is available at the decoding side and the encoding side, while the precise value of x_j may not be available when decoding.

To test if the input list $[x_j]$ is constant near j, we have to check if the modified difference $x_{j+1} - x'_j$ is zero or not. If it is zero, the encoder records the opcode `EQL`, which tells the decoder that the next value is identical to the last reconstructed value. In this case, we set $x'_{j+1} = x'_j$.

Observe that the difference $x'_{j+1} - x'_j$ of the reconstructed values is shared among the encoder and the decoder, and also that any quantity derived from $x'_{j+1} - x'_j$ has similar characteristics. Since we intend to encode motion data captured in high-frequency, the absolute difference $|x'_{j+1} - x'_j|$ tends to be small statistically, and its distribution is expected to concentrate around the origin. As we will see below, the moving average of $|x'_{j+1} - x'_j|$ with fixed window size helps to encode the difference efficiently. We write m_j to denote the moving average, viz.

$$m_j = \frac{1}{|K_j|} \sum_{k \in K_j} |x'_k - x'_{k-1}|, \ j = 1, 2, 3, \ldots; \quad m_0 = 0,$$

where K_j is the set of integer k satisfying $\max\{1, j-w+1\} \le k \le j$, and w is a positive constant representing the window size. Since m_j is defined using the reconstructed values x'_j, m_j is a shared quantity among the encoder and the decoder.

Observe now that if the moving average m_j is zero, then x'_k is constant for all $k \in K_j$. The `EQL` instruction can cover this case. So we may assume $m_j \neq 0$. We remark that m_j is constant if x'_j is a linear function of j. In other words, we can check

if the sequence x'_j behaves like a linear function of j by computing the ratio m_j/m_{j-1} and testing it is one or not. If the ratio is one, then x'_j is approximately linear. It is convenient to separate the sign from the difference $x_{j+1} - x'_j$ by writing

$$x_{j+1} - x'_j = s_j|x_{j+1} - x'_j|, \quad s_j = \mathrm{rsgn}(x_{j+1} - x'_j),$$

where $\mathrm{rsgn}(x)$ is the right continuous signum function defined by

$$\mathrm{rsgn}(x) = \begin{cases} 1 & 0 \le x \\ -1 & \text{otherwise.} \end{cases} \tag{2.1}$$

Note that for motion data, it is expected that the sign s_j does not change rapidly. The encoder records the `NEG` opcode only when the sign s_j is different from the previous sign s_{j-1}. The `NEG` opcode tells the decoder to negate the sign s_j for the succeeding opcodes.

To encode the absolute difference $|x_{j+1} - x'_j|$, we normalize it by

$$z_j = \frac{|x_{j+1} - x'_j|}{m_j}$$

and then scale and quantize z_j as

$$q_j = \left\lfloor z_j \frac{2^n}{\beta} + \frac{1}{2} \right\rfloor, \tag{2.2}$$

where n is an integer constant representing the number of bits used for quantization, and β is a positive real number representing the scale factor. The encoder then records the $\texttt{APX}(q_j)$ opcode, which tells the decoder to reconstruct the original value approximately from the quantized value q_j. The actual reconstructed value is

$$x'_{j+1} = x'_j + s_j \frac{q_j \beta m_j}{2^n}$$

in this case.

Observe that we have $z_j = 1$ if x_j is a linear function of j and hence the list $[x_j]$ which is linear in j, is encoded into a list $[\ldots, \texttt{APX}(\kappa), \texttt{APX}(\kappa), \texttt{APX}(\kappa), \ldots, \texttt{APX}(\kappa)]$ with repeated opcodes at the tail, where $\kappa = \lfloor \frac{2^n}{\beta} + \frac{1}{2} \rfloor$ is a constant. Note that if we use the value $\beta = 2^k$ for some positive integer k satisfying $k \le n$, then $z_j = 1$ is mapped to $q_j = 2^{n-k}$ and hence $x'_{j+1} = x'_j + s_j m_j$, meaning that the reconstructed value has no quantization error.

To handle the case where the combination of opcodes $\texttt{EQL}, \texttt{NEG}$, and $\texttt{APX}(q)$ do not work for some reason, we use the `RAW` opcode as a fallback to encode the exact value x_j utilizing a look-up table shared among the encoder and the decoder. In application, the `RAW` opcode is used when q_j given by (2.2) satisfies $q_j \ge 2^n$, which roughly corresponds to the condition $|x_{j+1} - x'_j| > \beta m_j$. The number of patterns for $\texttt{APX}(q)$ is at most 2^n in this case.

2.3. **DiffMA encoding.** The proposed encoding algorithm is given below in Algorithm 1, which is implementable by various programming languages. The algorithm uses the opcode `APX`(q) with $q = 0, 1, \ldots, 2^n - 1$ and thus $n = 6$ is a reasonable choice — any single opcode in Table 1 can be mapped to a single ASCII character in this case.

Algorithm 1: DiffMA Encode

Input: A list x of real numbers to encode, positive real numbers β, τ, and positive integers n, w.
Output: A list b of opcodes and a sublist t of x.

Initialize b, t, ℓ to be empty lists;
$s' := 1$;
if *x is not empty* **then**
    Remove the head element v from x;
    Append v to t;
    $v' := v$;
end
while *x is not empty* **do**
    Remove the head element v from x;
    $d := v - v'$;
    Let s be the sign of d, where $s = 1$ if $d \geq 0$ and $s = -1$ otherwise;
    Append opcode `NEG` to b if $s \neq s'$;
    Let c be undefined;
    Compute the mean value m of ℓ (set $m = 0$ if ℓ is empty);
    if $d = 0$ **then**
        $c :=$ `EQL`;
        $\hat{v} := v'$;
    else if $m > 0$ **then**
        $q := \left\lfloor \frac{2^n |d|}{\beta m} + \frac{1}{2} \right\rfloor$;
        $\hat{v} := v' + s \frac{q \beta m}{2^n}$;
        if $q < 2^n$ *and* $|v - \hat{v}| < \tau$ **then**
            $c :=$ `APX`(q);
        end
    end
    if *c is undefined* **then**
        $c :=$ `RAW`;
        Append v to t;
        $\hat{v} := v$;
    end
    Append opcode c to b;
    Append $|\hat{v} - v'|$ to ℓ;
    Drop the head element of ℓ if the length of ℓ is larger than w;
    Set $v' := \hat{v}$ and $s' := s$;
end

The following theorem shows that if we use real numbers instead of floating point numbers, then there is a formula to describe the encoding result and the output list of opcodes contains a repeated list at the tail, provided that the input list is linear. Therefore we can reduce the redundancy of linear data by applying Algorithm 1 in combination with existing run-length encoding and entropy coding algorithms.

Theorem 2.1. *Suppose $x = [x_j]$ is a non-empty list of length N that satisfies $x_j = \alpha j + \gamma$ for some real numbers α, γ and $j = 0, 1, \ldots, N-1$. Suppose further that Algorithm 1 is applied to the list x with $\beta = 2^k, \tau = \infty$ and $k \in \{0, 1, 2, \ldots, n\}$. The following statements hold:*

(1) If $\alpha = 0$, then we have

$$t = [x_0], \quad b = [\texttt{EQL}, \texttt{EQL}, \ldots, \texttt{EQL}]. \tag{2.3}$$

The list b has length $N - 1$.

(2) If $\alpha \neq 0$, then we have

$$t = [x_0, x_1], \quad b = \begin{cases} [\texttt{RAW}, \texttt{APX}(2^{n-k}), \ldots, \texttt{APX}(2^{n-k})] & \text{if } \alpha > 0, \\ [\texttt{NEG}, \texttt{RAW}, \texttt{APX}(2^{n-k}), \ldots, \texttt{APX}(2^{n-k})] & \text{if } \alpha < 0, \end{cases} \tag{2.4}$$

where the opcode $\texttt{APX}(2^{n-k})$ is repeated $(N-2)$-times in the formula for b.

Proof. (1) Right before the while loop at line 8 of Algorithm 1, we have

$$t = [x_0], \quad b = [], \quad \ell = [], \quad x = [x_1, x_2, \ldots], \quad v' = x_0, \quad s' = 1.$$

In the first iteration of the while loop, we have $v = x_1, d = v - v' = x_1 - x_0 = 0$ and `EQL` opcode is generated at line 16. After the first iteration, we have

$$t = [x_0], \quad b = [\texttt{EQL}], \quad \ell = [0], \quad \hat{v} = x_1, \quad x = [x_2, x_3, \ldots], \quad v' = x_1, \quad s' = 1.$$

Similarly, after the j-th iteration, we have

$$\begin{gathered} t = [x_0], \quad b = [\texttt{EQL}, \texttt{EQL}, \ldots, \texttt{EQL}], \\ \ell = [0, 0, \ldots, 0] \text{ (with length } \min\{j, w\}), \\ \hat{v} = x_j, \quad x = [x_{j+1}, x_{j+2}, \ldots], \quad v' = x_j, \quad s' = 1 \end{gathered}$$

and b has length j. This proves the statement.

(2) In the first iteration of the while loop, we have $v = x_1, d = \alpha$. Since we have $d \neq 0$ and $m = 0$, the if statement at line 15 and the else if statement at line 18 are skipped. `RAW` opcode is then generated at line 26. Hence, after the first iteration, we have

$$t = [x_0, x_1], \quad \ell = [|\alpha|], \quad \hat{v} = x_1, \quad x = [x_2, x_3, \ldots], \quad v' = x_1, \quad s' = \operatorname{rsgn} \alpha,$$

where rsgn is defined by (2.1). The list b depends on the sign of α and we have

$$b = [\texttt{RAW}] \text{ if } \alpha > 0, \quad b = [\texttt{NEG}, \texttt{RAW}] \text{ if } \alpha < 0.$$

Similarly, in the second iteration, we have

$$v = x_2, \quad d = x_2 - x_1 = \alpha, \quad m = |\alpha|, \quad q = \left\lfloor \frac{2^n}{\beta} + \frac{1}{2} \right\rfloor = 2^{n-k}.$$

Hence the algorithm generates the opcode $\texttt{APX}(2^{n-k})$. Moreover,

$$\hat{v} = v' + s\frac{q\beta m}{2^n} = x_1 + \alpha \left\lfloor \frac{2^n}{\beta} + \frac{1}{2} \right\rfloor \frac{\beta}{2^n} = x_1 + \alpha = x_2.$$

It follows that we have

$$t = [x_0, x_1], \quad \ell = [|\alpha|, |\alpha|], \quad \hat{v} = x_2, \quad x = [x_3, x_4, \ldots], \quad v' = x_2, \quad s' = \operatorname{rsgn} \alpha,$$

$$b = [\texttt{RAW}, \texttt{APX}(2^{n-k})] \text{ if } \alpha > 0, \quad b = [\texttt{NEG}, \texttt{RAW}, \texttt{APX}(2^{n-k})] \text{ if } \alpha < 0$$

after the second iteration.

Similar arguments show that, after j-th iteration, we have (2.4) and

$$\ell = [|\alpha|, \ldots, |\alpha|] \text{ (with length } \min\{j, w\}),$$
$$\hat{v} = x_j, \quad x = [x_{j+1}, x_{j+2}, \ldots], \quad v' = x_j, \quad s' = \operatorname{rsgn} \alpha.$$

This completes the proof. □

2.4. **DiffMA decoding.** The decoding algorithm is shown in Algorithm 2.

Algorithm 2: DiffMA Decode

Input: Lists b and t obtained by Algorithm 1 with parameters β, τ, n, w.
Output: A list $\tilde{x}$ of real numbers.

Initialize $\tilde{x}, l$ to be empty lists;
$s := 1$;
if *t is not empty* **then**
    Remove the head element v from t;
    Append v to $\tilde{x}$;
    $v' := v$;
end
while *b is not empty* **do**
    Remove the head element c from b;
    if $c = \texttt{NEG}$ **then**
        $s := -s$;
        continue;
    else if $c = \texttt{EQL}$ **then**
        $\hat{v} := v'$;
    else if $c = \texttt{RAW}$ **then**
        Remove the head element a from t;
        $\hat{v} := a$;
    else
        Find an integer q such that $c = \texttt{APX}(q)$;
        Compute the mean value m of ℓ;
        $\hat{v} := v' + s\frac{q\beta m}{2^n}$;
    end
    Append $\hat{v}$ to $\tilde{x}$;
    Append $|\hat{v} - v'|$ to ℓ;
    Drop the head element of ℓ if the length of ℓ is larger than w;
    $v' := \hat{v}$;
end

The following theorem shows that if we use the real number field instead of floating point numbers, the input list can be reconstructed precisely by the decoding algorithm,

provided that the input is linear. Therefore the proposed DiffMA algorithm can encode and decode any linear list without a loss of information.

Theorem 2.2. *Suppose the same assumptions as in Theorem 2.1. Let b and t be the pair of lists output by Algorithm 1 from an input list x. Then, the output $\tilde{x}$ of Algorithm 2 applied to b and t is identical to x.*

Proof. We first consider the case where $\alpha = 0$ and decode b and t given by (2.3) using Algorithm 2. At line 4 we have $v = x_0$ and hence, right before the while loop, we have

$$t = [], \quad b = [\texttt{EQL}, \texttt{EQL}, \ldots, \texttt{EQL}], \quad v' = x_0, \quad \tilde{x} = [x_0], \quad s = 1.$$

It is easy to see that each iteration of the wihle loop appends x_0 to $\tilde{x}$ and the while loop terminates after $N - 1$ iterations. This proves the case $\alpha = 0$.

We next consider the case where $\alpha \neq 0$. Suppose for the moment that $\alpha > 0$. In this case, we apply Algorithm 2 to

$$t = [x_0, x_1], \quad b = [\texttt{RAW}, \texttt{APX}(2^{n-k}), \texttt{APX}(2^{n-k}), \ldots, \texttt{APX}(2^{n-k})].$$

Right before the while loop, we have

$$t = [x_1], \quad v' = x_0, \quad \tilde{x} = [x_0], \quad s = 1.$$

In the first iteration of the while loop, `RAW` at the head of b is extracted and decoded. Hence, after the first iteration, we have

$$\begin{aligned} t = [], \quad \ell = [|\alpha|], \quad b = [\texttt{APX}(2^{n-k}), \texttt{APX}(2^{n-k}), \ldots, \texttt{APX}(2^{n-k})], \\ \hat{v} = v' = x_1, \quad \tilde{x} = [x_0, x_1], \quad s = \operatorname{rsgn} \alpha. \end{aligned} \tag{2.5}$$

Observe that (2.5) holds for the case $\alpha < 0$ as well, since the `NEG` opcode in (2.4) only affects the sign of s.

From now on our arguments are the same for $\alpha > 0$ and $\alpha < 0$. In the second iteration, $\texttt{APX}(q)$ is decoded at the head of b, where $q = 2^{n-k}$. The mean value m of ℓ satisfies $m = |\alpha|$ at line 20 and hence we obtain

$$\hat{v} = v' + s\frac{q\beta m}{2^n} = x_1 + \frac{2^{n-k}2^k\alpha}{2^n} = x_1 + \alpha = x_2. \tag{2.6}$$

Therefore, after the second iteration, we have

$$\begin{aligned} t = [], \quad \ell = [|\alpha|, |\alpha|], \quad b = [\texttt{APX}(2^{n-k}), \texttt{APX}(2^{n-k}), \ldots, \texttt{APX}(2^{n-k})], \\ \hat{v} = v' = x_2, \quad \tilde{x} = [x_0, x_1, x_2], \quad s = \operatorname{rsgn} \alpha. \end{aligned} \tag{2.7}$$

Note that the length of b decreases by one for each iteration. Similar arguments show that, after j-th iteration, we have

$$\begin{aligned} t = [], \quad \ell = [|\alpha|, |\alpha|, \ldots, |\alpha|] \text{ (with length } \min\{j, w\}), \\ b = [\texttt{APX}(2^{n-k}), \texttt{APX}(2^{n-k}), \ldots, \texttt{APX}(2^{n-k})] \text{ (with length } N - j - 1), \\ \hat{v} = v' = x_j, \quad \tilde{x} = [x_0, x_1, x_2, \ldots, x_j], \quad s = \operatorname{rsgn} \alpha. \end{aligned} \tag{2.8}$$

This proves the case $\alpha \neq 0$. □

3. Experiments

3.1. **CMU Motion Capture Database.** As a dataset for evaluation experiments of the proposed algorithm, CMU's motion capture database (CMU Mocap Database) is used in this work. We briefly summarize that database before describing the experiments. The database contains motion capture data recorded by sensors while human actors perform various motions, including acrobatic actions such as cartwheel or somersault. The original database consists of ASF (Acclaim Skeleton File) file and AMC (Acclaim Motion Capture data) file. An ASF file is an ASCII encoded file used to store the structure of a skeletal model, and it is shared among multiple motion capture data that depend on the same skeleton. An AMC file is an ASCII encoded file that stores the actual motion capture data. It contains time series data of all the bones specified in the ASF file [3].

CMU Mocap Database uses a skeleton with 29 bones. Each bone has one to six degrees of freedom, depending on the bone, and the total degrees of freedom is 62 per skeleton. The database contains 2514 AMC files. Each AMC file is a time series of frames containing all the motion data of the bones. The frames are recorded at 120 Hz, and each frame has 62 sets of 32-bit single-precision floating point numbers. The total number of frames in the database is 4,151,474, and the recorded floating point numbers amount to 257,391,388 samples. When no compression is applied, the total size of AMC files is 3,314,159,095 in bytes. Hence, on average, 103 bits are required for storing a floating point number without compression.

In practice, we often apply well-known loss-less compression algorithms to the database for saving the storage. A common choice of such an algorithm is LZMA. When LZMA compression is applied to all AMC files separately, the average data rate, the average length of compressed file per floating point sample, is 24.4±1.9, where the number after ± means the standard deviation.

3.2. **Implementation.** For evaluating the proposed DiffMA algorithm, an implementation was created using Python 3. In the current implementation, opcodes encoded in the list b are mapped to integers by

$$\texttt{NEG} = 0, \quad \texttt{RAW} = 1, \quad \texttt{EQL} = 2, \quad \texttt{APX}(q) = 3 + q,\ q = 0, 1, \ldots, 2^n - 1, \tag{3.1}$$

and transformed further to ASCII characters by using the Base85 encoding [1]. Hence, the list b is represented by a single ASCII string. The numbers in the look-up table t are converted to a JSON array of floating point numbers.

The output of the DiffMA encoder is a JSON object `obj` such that `obj.channels` is an array of channels for the bones (a single channel is assigned to each degree of freedom of the bone), `obj.records` is a JSON object holding lists of opcodes and look-up tables, and `obj.params` stores some encoding parameters. The component `obj.records.b` is an array of Base85 encoded opcodes, and `obj.records.t` is an array of look-up tables. For the j-th channel `obj.channels[j]`, its list of opcodes is stored at `obj.records.b[j]` and its look-up table at `obj.records.t[j]`.

3.3. **An Example of DiffMA Encoded Data.** Listing 1 shows an example of an AMC file containing five frames of motion for a bone named `x`. The bone `x` has three degrees of freedom in this example. The first channel $[1, 1, 1, 1, 1]$ is constant with value 1 for all frames, the second channel $[0, 1, 2, 3, 4]$ is a linear function of the frame number, and the third channel $[0.0, 0.0952, 0.1904, 0.285599, 0.380799]$ is generated by $j\pi/33$ for $j = 0, 1, \ldots, 4$, which fails to be linear due to rounding errors.

```
#!OML:ASF
:FULLY-SPECIFIED
:DEGREES
1
x 1.000000 0.000000 0.000000
2
x 1.000000 1.000000 0.095200
3
x 1.000000 2.000000 0.190400
4
x 1.000000 3.000000 0.285599
5
x 1.000000 4.000000 0.380799
```

Listing 1. A Sample AMC File

```
{
  "channels": [ "x0", "x1", "x2" ],
  "records": {
    "b": [ "2222", "1ZZZ", "1ZYa" ],
    "t": [ [ 1 ], [ 0, 1 ], [ 0, 0.0952 ] ]
  },
  "params": {
    "bits": 6, "window": 64, "bound": 2, "class": "DiffMA",
    "tolerance": 10, "roundOffset": 0, "encoding": "base85"
  }
}
```

Listing 2. A Sample DiffMA Encoded File in JSON Format

Listing 2 shows the result of DiffMA encoding applied to the AMC file of Listing 1. As illustrated by this JSON file, the first channel is encoded as the opcodes represented by the string `2222`, which means $[\texttt{EQL}, \texttt{EQL}, \texttt{EQL}, \texttt{EQL}]$. The look-up table for this channel is an array [1] of a single number. In this case, the channel data are constant, and the opcodes show the pattern (1) of Theorem 2.1.

Similarly, the second channel is encoded as the string `1ZZZ`, which means

$$[\texttt{RAW}, \texttt{APX}(\kappa), \texttt{APX}(\kappa), \texttt{APX}(\kappa)]$$

for some constant κ. The look-up table for this channel is $[0, 1]$. In this case, the channel data are linear, and the opcodes follow the pattern (2) of Theorem 2.1.

The third channel is encoded as the string `1ZYa`, which means

$$[\texttt{RAW}, \texttt{APX}(\kappa_1), \texttt{APX}(\kappa_2), \texttt{APX}(\kappa_3)]$$

for some constants $\kappa_j,\ j = 1, 2, 3$. The look-up table is $[0, 0.0952]$ in this case, and the generated opcodes are different from the patterns of Theorem 2.1.

3.4. **Experiments.** Experiments were carried out to examine the effect of the main parameter β (bound) on the quality and the size of DiffMA encoded data. The parameter n was fixed to $n = 6$ in all experiments. In all experiments, all AMC files in CMU's Mocap Database were encoded separately using the same set of parameters β, w, and τ.

For measuring and comparing the quality of the encoded data, PSNR (peak signal-to-noise ratio) was computed for each DiffMA encoded file. PSNR used in this paper is $\mathrm{PSNR} = -10 \log_{10} \mathrm{MSE}$, where MSE is the mean square error. In general, a higher value of PSNR means better quality. Table 2 shows the statistics of PSNR for different values of β obtained by encoding all AMC files separately for each value of β. It is observed that the mean and minimum values of PSNR behave like linear functions of $\log_{10} \beta$. This result shows that we can use the parameter β to control the quality of DiffMA.

Table 2. PSNR in dB across different values of β (with $n = 6, w = 64, \tau = 10$)

bound	mean	std	min	1%	25%	50%	75%	99%	max
1.0	53.50	17.56	36.10	39.96	49.46	52.86	56.31	66.83	644.07
2.0	46.48	17.71	28.72	33.59	42.59	45.85	49.19	59.50	644.07
4.0	40.17	17.85	23.09	27.81	36.23	39.50	42.78	52.92	644.07
8.0	34.09	18.02	16.33	21.20	30.18	33.42	36.72	47.37	644.07
16.0	28.06	18.18	10.82	15.62	24.13	27.43	30.69	41.31	644.07
32.0	22.07	18.34	4.96	9.86	18.13	21.36	24.67	35.54	644.07
64.0	16.14	18.47	-0.89	5.05	12.26	15.37	18.65	29.11	644.07

Table 3. Data rate of DiffMA output in bits per sample across different values of β (with $n = 6, w = 64, \tau = 10$)

bound	mean	std	min	1%	25%	50%	75%	99%	max
1.0	37.82	3.62	24.42	32.91	36.96	37.70	38.47	43.70	146.13
2.0	20.24	3.93	15.42	17.44	19.36	19.91	20.67	24.85	146.13
4.0	12.15	4.06	9.65	9.99	11.32	12.01	12.62	14.81	146.13
8.0	10.68	4.07	8.79	8.93	9.96	10.60	11.09	12.71	146.13
16.0	10.52	4.07	8.77	8.85	9.85	10.42	10.91	12.37	146.19
32.0	10.52	4.06	8.84	8.93	9.86	10.41	10.89	12.35	146.19
64.0	10.59	4.05	9.04	9.12	9.94	10.44	10.92	12.32	146.19

Table 3 summarizes the statistics of the data rate of DiffMA encoded AMC files with JSON output. The data rate here is defined as the number of output bits per floating point sample:

$$\text{Data rate} = \frac{\text{Number of output bits}}{(\text{Number of input channels}) \times (\text{Number of input frames})}.$$

A lower data rate means a better result. The result in Table 3 indicates that the data rate improves as the bound β increases when β is small, but significant improvements can not be expected for $\beta > 4$. Further refinement of data rate was observed when the output JSON file was post-processed by LZMA. The result is shown in Table 4.

As a comparison, a well-known floating point compressor zfp [2] was applied to all AMC files separately, and then PSNR and the data rate were measured. Since zfp requires floating point arrays as input, a multi-dimensional array was constructed from the floating point values in all channels from each file, and the zfp compression

TABLE 4. Data rate of DiffMA output in bits per sample across different values of β when post-processed by LZMA (with $n = 6, w = 64, \tau = 10$)

bound	mean	std	min	1%	25%	50%	75%	99%	max
1.0	13.74	1.76	8.32	11.28	13.17	13.65	14.19	16.57	62.71
2.0	9.16	1.77	5.66	7.09	8.73	9.17	9.56	11.01	62.71
4.0	6.25	1.83	3.47	4.19	5.88	6.34	6.66	7.65	62.71
8.0	4.98	1.85	2.63	3.14	4.61	5.11	5.36	6.14	62.71
16.0	4.14	1.85	2.15	2.55	3.79	4.24	4.51	5.19	62.71
32.0	3.41	1.85	1.79	2.09	3.08	3.44	3.72	4.47	62.97
64.0	2.79	1.83	1.50	1.72	2.48	2.76	3.02	3.80	62.71

in lossy mode with tolerance 0.125 was applied to that multi-dimensional array. The zfp output was post-processed by LZMA. The mean PSNR was 39.719261 ± 0.445092, and the mean data rate was 8.593396 ± 1.103044, where mean values were derived by averaging all AMC files. Let us compare this result with DiffMA. By Table 2, we see that 99% of data has PSNR larger than 39.96 if DiffMA is applied with $\beta = 1$. The corresponding mean data rate is 13.74 ± 1.76 by Table 4. Therefore DiffMA combined with LZMA under $\beta = 1$ requires about 60% larger mean data rate than zfp when PSNR is needed to be above approximately $40dB$.

The data rate of DiffMA with LZMA can be improved by searching for the best value of β for each AMC file independently, rather than using a single value of β for all AMC files. This optimization can be done by minimizing the absolute distance of PSNR to a specified value over different values of β. An experiment showed that if the target PSNR was set to 40dB and the minimization was carried out over $\beta = 2^k$ for $k = 0, 1, \ldots, 6$, then the resulting mean data rate was 6.941915 ± 2.822336, the mean PSNR was 40.462124 ± 17.132779, and the mean value of β was 4.853222 ± 5.009455. This result means that DiffMA shows the data rate mostly comparable to zfp.

4. Conclusion

The DiffMA algorithm proposed in this paper is quite simple and easy to implement by any computer language. Theorems provided by this paper guarantee that DiffMA encodes a constant or linear list of floating point numbers into a simple sequence of opcodes losslessly. When the input list is non-linear, a lossy quantization is applied by DiffMA. Experimental results showed that by optimizing the output of DiffMA combined with LZMA over the main parameter β, we could encode CMU's motion capture data at the data rate of 6.9 ± 2.8 bit per floating point number with PSNR of 40 ± 17 dB. DiffMA can be used in applications for reducing data rate if quantization errors can be tolerated — building level-of-details of animation is a possible application of DiffMA. A further study is needed to develop a method to control the errors of DiffMA and to extend DiffMA to encode quadratic, cubic, and polynomial input losslessly.

5. Acknowledgements

The data used in this project was obtained from mocap.cs.cmu.edu. The database was created with funding from NSF EIA-0196217.

References

[1] Robert Elz. A Compact Representation of IPv6 Addresses. RFC 1924, April 1996.
[2] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20, 08 2014.
[3] research.cs.wisc.edu. Acclaim ASF/AMC. `https://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/ASF-AMC.html`, 1999. [Online; accessed May 16, 2021].

（2021.5.20 受稿，2021.7.1 受理）

— Abstract —

This paper aims to provide a mathematical method for eliminating redundancy from the recorded motion capture data and making it possible to store various high-frequency floating point data efficiently in JSON (JavaScript Object Notation) format. The DiffMA algorithm proposed in this paper is quite simple and easy to implement by any computer language. Theorems provided by this paper guarantee that DiffMA encodes a constant or linear list of floating point numbers into a simple sequence of opcodes losslessly. When the input list is non-linear, a lossy quantization is applied in DiffMA. Experiments are carried out to examine the relationship between the data rate and the parameters of DiffMA. The result shows that we can encode CMU's motion capture data at the data rate of 6.9 ± 2.8 bit per floating point number with PSNR of 40 ± 17 dB by applying DiffMA with LZMA.