Contents

1 Introduction

Applications in scientific computing often operate with large volumes of input and output data. Despite the huge capacity of the modern disk devices, the space required for data storage is often larger than the hardware allows. Data transfer over communication networks is another bottleneck in scientific computing. Text compression tools cannot compress binary numeric data. Image compression algorithms are not intended for such data. Signal compression algorithms do not work directly with numerical data of time series, when time steps are varying. Also, they are typically designed for measured values, not for simulation results and therefore not intended for lossless compression. Therefore new data compression algorithms must be designed.

We assume that application data is stored as one or multiple arrays, i.e. time[1] series. The elements of the arrays are certain quantities that change *smoothly* (see Section 1.1). Informally, a smooth function is a function that is close to some polynomial. Smooth arrays contain a sequence of values of this function. Smooth arrays are typical for numerical dynamic simulations of physical phenomena where scalar values change in time. An investigation of these values reveals several things: properties of the solver, properties of mathematical model, and of course, the physical phenomena

[1] We use *time* as the domain for the steps; however in PDE-based simulations it can be a space axis.

themselves. The values are consequently computed after each other. Often these quantities change so slowly that nearby elements differ in the few last digits only. Sometimes the elements of an array are computed from respective elements of another array so that the correlation between the values can be found. These numerical properties might result in a very high compression ratio. The problem is how to discover all the features of the data and how to use them for data compression.

Algorithm		Good for time steps:	Ratio
Delta-compression	Differences	fixed	good
	Extrapolation with fixed step	fixed (same as above)	good
	Extrapolation with varying step	varying	best
Wavelet algorithms		fixed	poor or N/A

Table 1: Comparison of algorithms described in the paper.

In section 1.2 we discuss why general-purpose compression algorithms do not work with numerical data from simulations. In section 2 we introduce the simplest delta-compression algorithm (see Table 1) based on fixed time steps. Properties of memory representation for real numbers are discussed. In section 3 we reformulate the algorithm by using extrapolation formulas. In section 4 we extend the algorithm for varying steps, because such data arrays are more typical for simulations.

Experiments with artificial data sequences as well as data samples from a realistic application have been carried out and compression ratios are reported in Section 5 .

1.1 Smoothness of the data

The compression algorithm input is a sequence a consisting of values a_i, $i = 1, ..., n$. It works with arbitrary sequences of numerical values. However, it can deliver some considerable compression ratio for smooth data sequences only, where substantial correlation between adjacent data values can be found.

Assume that a function $f : [1, n] \to R$ is evaluated on time interval $[1, n]$. The values a_i are stored in the sequence, a so that $a_i = f(i)$.

An array a is called *smooth of order* m if every a_j $(j > m)$ can be well approximated by the extrapolating polynomial based on previous m values[2]. In the simplest case, if a function that has very small and slow changes (is close to a polynomial of order 0, i.e. a constant) then the corresponding array a can be compressed very well.

In practice smooth functions represent solutions of ordinary differential equations and various continuous quantities that are computed in simulations.

[2]Formally, each polynomial φ_j is created from $a_{j-m}, ..., a_{j-1}$ and $\varphi_j(j) \approx a_j$

1.2 Compression and data representation

The traditional text compression algorithms cannot compress numerical data because these do not utilize correlation between adjacent floating point values. Text compression algorithms represent the data as a string in an small alphabet (0..255) and attempt to find equal substrings, producing lossless compression.

Image compression algorithms represent the data as small rectangular blocks of pixels (usually three values in the 0..255 alphabet) and if the pixels in the block have similar color then the algorithm replaces color information in all the pixels by a single color, giving lossy compression. Such algorithms utilize correlation between adjacent data values, but do not use similarity of the gradient (speed of the changes) of the data.

Another approach to the problem consists of signal compression algorithms. These use data smoothness but do not consider the fact that time steps vary. Very few of them can provide lossless compression, and compression ratio is insufficient in this case.

Representation of numerical data in computer memory is important for our compression algorithm. It is assumed that 64-bit real numerical values are used. This data representation corresponds to the type `double` in most C compilers, operating systems and processor architectures used for numeric computations (Intel, Alpha, Sparc families, etc.).

Let us consider how the numerical data are represented in the memory. If p is a floating point 64-bit number, $\texttt{Int}(p)$ is defined as an integer that is represented by the same 64-bit string as p (see Table 2 and more details in Section 2.1). In this paper we use hexadecimal notation for such numbers to emphasize the byte-level representation.

		byte number	1	2	3	4	5	6	7	8
a_1	2.3667**1**76745585676	$\texttt{Int}(a_1)$	40	02	ef	**09**	**ad**	**18**	**c0**	**f6**
a_2	2.3667**2**76745585676	$\texttt{Int}(a_2)$	40	02	ef	**0e**	**eb**	**46**	**23**	**2f**
a_3	2.3667**3**76745585676	$\texttt{Int}(a_3)$	40	02	ef	**14**	**29**	**73**	**85**	**6a**

Table 2: Integer representation of three real numbers

Three real numbers a_1, a_2, a_3 differ in the 5-th digit after the decimal point only. It is a linearly growing sequence. Each number occupies 64 bits. The first three bytes are almost the same, and the general-purpose algorithms for compression of byte sequences can use this fact for compression. However, the problem is that the five last bytes (in **bold**) are perceived as completely random. There is no correlation between these five bytes of one number and five bytes of another number. Traditional compression algorithms cannot compress these bytes.

2 Fixed-Step Delta-Compression

In this section we consider the simplest variant of delta-compression algorithm. Our algorithm computes the first (and higher-order) differences using 64-bit integer arithmetic; subsequently meaningless 0s or 1s are truncated in the result.

2.1 Internal Representation of Double Values.

The function $\mathtt{Int}(p)$ has been defined as the *integer* representation of the 64-bit memory area allocated for a floating point number p. Here p can be any machine number between $mindouble$ and $maxdouble$[3]. This memory contains[4] concatenation of one sign bit, 11 exponent bits and 52 mantissa bits. The function $\mathtt{Int}$: $[mindouble, maxdouble] \to \{0, ..., 2^{64} - 1\}$ is continuously growing everywhere except in 0. This function is very close to linear on every interval such as $[2^k, 2^{k+1}]$, $-1023 \le k \le 1023$. A fragment of the function graph is shown in the Figure 1(a), where $\mathtt{Int}(x)$ is given in hexadecimal notation.

The function $\mathtt{Real}(x)$ is opposite to $\mathtt{Int}(x)$, i.e. the equality $\mathtt{Real}(\mathtt{Int}(x)) = x$ occurs.

For integer p, the $\mathtt{Bin}(p, r)$ is an r-bit sequence of zeros and ones of the binary representation of p ($1 \le r \le 64$, $|p| < 2^r$).

2.2 Definition of differences

Let us assume that an array b containing 64-bit integers b_i, $i = 1, ..., n$ is given. The first difference is defined as $\Delta^1 b_i = b_i - b_{i-1}$. The m-th difference is defined in a similar way as $\Delta^m b_i = \Delta^{m-1} b_i - \Delta^{m-1} b_{i-1}$.

Instead of storing the whole array b we can just store the m-th differences for b. In this case we need storage for n values:

$$(b_1, b_2, ..., b_m, \Delta^m b_{i+1}, \Delta^m b_{i+2}, ..., \Delta^m b_n)$$

The sequence b can be unambiguously restored from the m-th differences. Since b elements are integers, they are restored without loss of precision.

The difference between two 64-bit integers can be stored in 64 bits. In general case 65 bits would be needed, but we utilize the wrap-around feature of computer integer arithmetics. This feature can be illustrated by computation:

If $b_1 = 2^{63} - 1$ and $b_2 = -(2^{63} - 1)$ then $b_2 - b_1$ produces 2. Also, $b_1 + 2 = b_2$.

From above it can be concluded that a sequence b of length n can be stored as m-th differences for b , and it will not occupy more than the same memory i.e. $64n$ bits.

[3]Normally $mindouble \approx -10^{309}$ and $maxdouble \approx 10^{309}$ are predefined compiler constants.

[4]This description can be processor-dependent; it holds for Intel, Alpha and Sparc families. The numbers can be investigated by a trivial C program. Order of bytes is, however, different: Intel and Alpha are “little-endian”, whereas Sparc is “big-endian”. This is taken into account by compression/decompression routines.

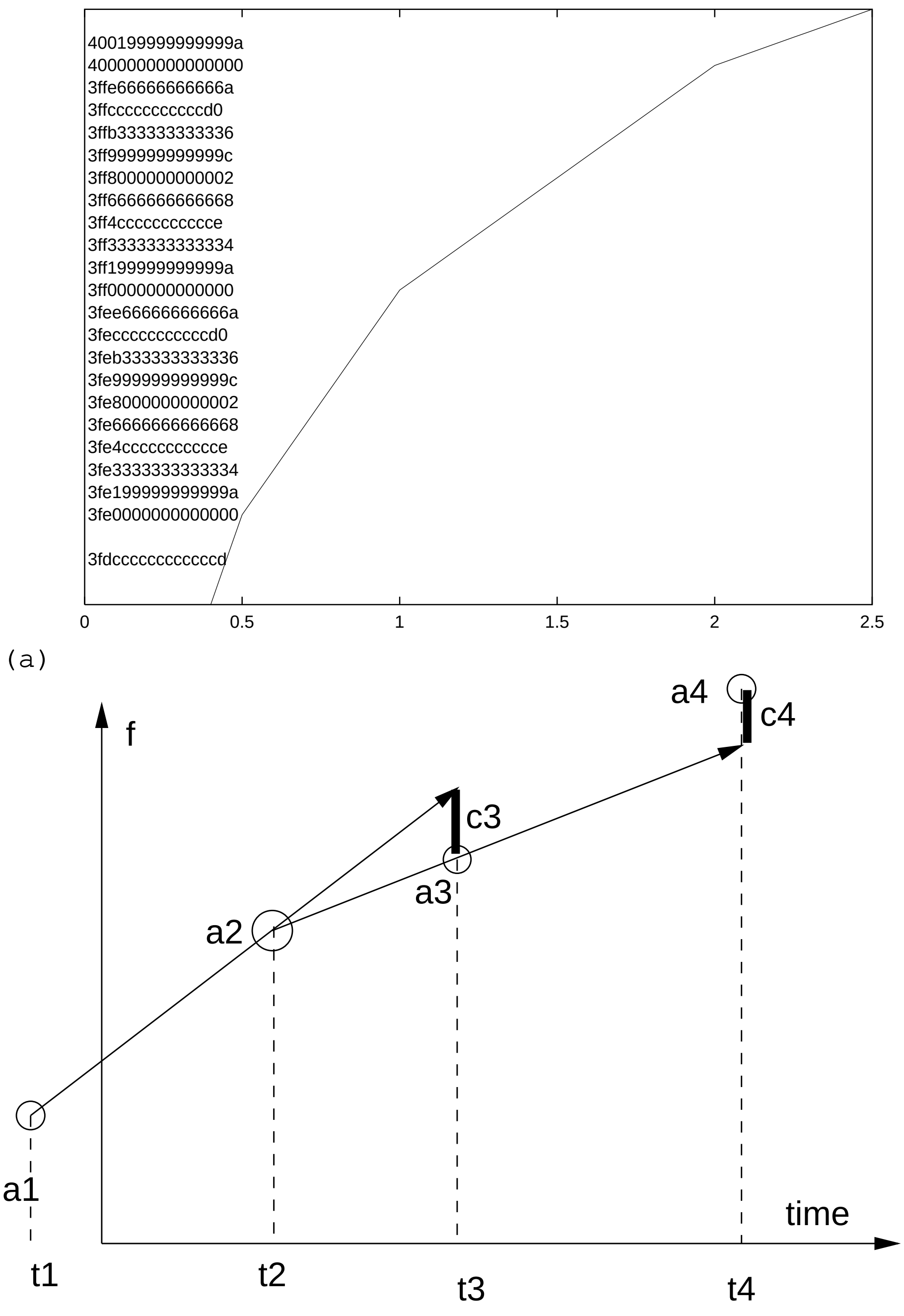

Figure 1: (a) The graph of function $\texttt{Int}(x)$. (b) Finding first order differences c_3 and c_4.

2.3 Truncating meaningless bits.

We can use the fact that the difference between the array elements is small relatively to the element values.

Small integer numbers have many initial zeroes (in positive numbers) or ones (in negative numbers) in their binary representation. These meaningless bits can be truncated. For instance, zeroes in `0000`**`0101`** can be truncated and just `0101` is stored. Formally, truncating of bit sequences can be described as follows:

Assume, s is a binary digit sequence of k elements $(s_1, ..., s_k)$ where $s_i \in \{0, 1\}$. Then $\texttt{Drop}(s)$ is defined as such substring $(s_l, ..., s_k)$ that all digits at the beginning of the original sequence are equal: $s_1 = s_2 = \ldots = s_l$ after which some other digit follows: $s_l \neq s_{l+1}$.

Two extreme cases are defined: $\texttt{Drop}(00\ldots0) = \texttt{Drop}(0) = 0$ and $\texttt{Drop}(11\ldots1) = \texttt{Drop}(1) = 1$.

For instance, `Drop(0000`**`0101`**`)=0101`, `Drop(1111111`**`101001`**`)=101001`.

2.4 The difference compression algorithm

The algorithm compresses a sequence of real numbers a to bit sequence e using differences of order m. It consists of the following steps:

— The integer values are taken instead of real: $b_i = \texttt{Int}(a_i)$, $i = 1, ..., n$;

— First m values are copied: $c_i = b_i$ for $i = 1, ..., m$;

— The m-th order differences are computed: $c_i = \Delta^m b_i$ for $i = m + 1, ..., n$. See c_3 and c_4 on Figure 1(b). Further the *systematic coding* method is applied to c, i.e.:

— Only necessary bits are selected $d_i = \texttt{Drop}(\texttt{Bin}(c_i, 64))$;

— The bit string length[5] and the bit string itself are stored:

$$e_i = \texttt{Concatenate}(\texttt{Bin}(\texttt{Length}(d_i), 6), d_i)$$

where `Concatenate` is the bit string concatenation operator.

— All e_i are concatenated to the single bit string $e = \texttt{Concatenate}(e_1, ..., e_n)$.

The sequence a can be restored again from e unambiguously by reverse operations[6].

For an example (see b_i in Table 2),
$\Delta^1 b_2 = b_2 - b_1$ = `00 00 00 05 3e 2d 62 39`
$\Delta^1 b_3 = b_3 - b_2$ = `00 00 00 05 3e 2d 62 3b`
$\Delta^2 b_3 = \Delta^1 b_3 - \Delta^1 b_2$ = `00 00 00 00 00 00 00 02`

It can be noted that the first difference requires 5 bytes (more exactly, 36 bits). The second difference requires no more than 3 bits (`010`). Also, 6 bits are used to encode the length. Assuming that the algorithm stores b_1,

[5] Note that there can be various approaches for length storage, for instance $e_i = \texttt{Concatenate}(d_i, \texttt{ENDMARKER})$, but we found that our solution is close to optimal. This length can be coded in 6 bits because $2^6 = 64$.

[6] It can be done since the result of 64-bit integer addition and subtraction is identical on all processors working with 64-bit integers.

b_2 and $\Delta^2 b_3$ and compression ratio $(64*3)/(64*2+6+3) \approx 1.401$ is achieved.

Normally there is no smoothness in the sequence e, therefore it cannot be compressed further[7].

3 Using Fixed Step Extrapolation

The algorithm of differences of order m can be formulated differently in terms of extrapolation of order $(m-1)$. When this reformulation is performed, just slightly different[8] computations take place, and these can be seen from different point of view. We introduce the extrapolation technique here (instead of differences) in order to proceed later to varying step extrapolation algorithm in Section 4.

The fixed step difference algorithm works well if the sequence $\mathtt{Int}(a)$ can be approximated by polynomials. In real applications, however, it would be better to assume that a can be approximated by polynomials[9].

To explore this approach the traditional Lagrange extrapolation formula [5] should be used. The Lagrange rule for extrapolation of order $m-1$ states that for function $f(x)$ and extrapolation points $x_1, ..., x_m$ there exists a polynomial $\varphi_m(x)$ such that $\varphi_m(x_i) = f(x_i)$ for $i = 1, ..., m$. The polynomial[10] can be found as $\varphi_m(x) = L_1(x)f_1 + \ldots + L_m(x)f_m$, where $f_i = f(x_i)$ and

$$L_i(x) = \frac{(x-x_1)\ldots(x-x_{i-1})(x-x_{i+1})\ldots(x-x_m)}{(x_i-x_1)\ldots(x_i-x_{i-1})(x_i-x_{i+1})\ldots(x_i-x_m)}$$

The compression algorithm using fixed step extrapolation of order $m-1$ first takes a sequence of real numbers a and produces a sequence of integers c.

First, m first values are copied: $c_j = \mathtt{Int}(a_j)$ for $j = 1, ..., m$.

Subsequently, every c_j where $j = m+1, ..., n$ is sequentially computed as follows:

1. The m extrapolation points to the left of j are chosen: $x_1 = j - m, ..., x_m = j-1$.

2. Correspondingly, function values are set as $f_1 = a_{j-m}, ..., f_m = a_{j-1}$.

3. The predicted value $\varphi_m(x)$ for $x = j$ is computed using the Lagrange formula.

4. The extrapolation residual (difference between actual and predicted value) is stored (therefore our method is a variant of *predictive coding*).

---

[7] Relatively small additional smoothness can be found in the sequence of $\mathtt{Length}(d_i)$, but we ignore this for brevity.

[8] Discussed in more detail in [2] .

[9] The correlations between two and more arrays $(a^{[1]}, a^{[2]}, a^{[3]}, ..., a^{[r]})$ of the same length can be taken into account. It might produce a high compression ratio, specially if appears that $a^{[k]} \approx p(t, a^{[1]}, a^{[2]}, a^{[3]}, ...)$ and p is a polynomial. These correlations can be discovered automatically, however this is rather time consuming.

[10] Note that the term $x_i - x_i$ is always skipped in the divider. If $x_1 = j-3$, $x_2 = j-2$, $x_3 = j-1$ then $\varphi_3(j) = f(j-3) - 3f(j-2) + 3f(j-1)$.

Since we expect that this difference is very small, the values are first converted to the integer representation and then subtracted: $c_j = \mathtt{Int}(a_j) - \mathtt{Int}(\varphi_m(j))$

Subsequently, the necessary operations with c are performed just like in the Section 2.4.

3.1 Decompressing

The original sequence a can be unambiguously restored from the compressed sequence:

First, sequence of integers c is restored from the bit string e.

Then first m values are copied: $a_j = \mathtt{Real}(c_j)$ for $j = 1, ..., m$.

After that from every c_j where $j = m+1, ..., n$ the value a_j is sequentially computed as follows:

1. The m extrapolation points to the left of j are chosen: $x_1 = j - m, ..., x_m = j - 1$.
2. Correspondingly, function values are set as $f_1 = a_{j-m}, ..., f_m = a_{j-1}$.
3. The predicted value $\varphi_m(x)$ for $x = j$ is computed using the Lagrange formula.
4. The actual value is computed as the sum of predicted value and the residual:

$$a_j = \mathtt{Real}(\mathtt{Int}(\varphi_m(j)) + c_j)$$

Evaluation of $\varphi_m(j)$ includes double precision arithmetic that can potentially can produce different results on different processors, since they use different techniques to round up multiplication or division results to fit it into 64-bit space. To guarantee lossless decompression, it should be performed on the same processor family as compression. Otherwise an error in the last bit might appear, accumulate and lead to losing numerical accuracy[11].

The algorithm is fast since the coefficients for Lagrange formula are computed efficiently and only once (see [2] for details)

4 A Varying Step Extrapolation Algorithm

The previous algorithms assumed that the sequence a can be approximated by polynomials with fixed steps between extrapolation points. However, in practice simulations use adaptive ODE solvers and produce state variable values for varying, non-equidistant time steps. Therefore we should consider smooth functions with values taken with varying steps, and adapt the compression algorithm for such application data.

Assume that a function $f : [t_{min}, t_{max}] \to R$ is evaluated during the simulation.

[11]Our experiments with Sparc and Alpha processor families show that the difference is never greater than the 2-3 last bits when extrapolation of 3rd order is used and $n = 100$.

A finite number (n) of function values is produced by the solver for time steps $(t_1, ..., t_n)$ (where $t_{min} = t_1$, $t_{max} = t_n$, $t_i < t_{i+1}$) and these are stored in the sequence a so that $a_i = f(t_i)$.

The sequence t is used for compression and decompression of a. The sequence t itself should be compressed by the fixed step difference algorithm.

The compression algorithm using varying step extrapolation of order $m - 1$ first takes the sequence of real numbers a and t and produces a sequence of integers c.

First, m first values are copied: $c_j = \mathtt{Int}(a_j)$ for $j = 1, ..., m$. Subsequently every c_j where $j = m + 1, ..., n$ is sequentially computed as follows:

1. The m extrapolation points to the left of j are chosen: $x_1 = t_{j-m}, ..., x_m = t_{j-1}$.

2. Correspondingly, function values are set as $f_1 = a_{j-m}, ..., f_m = a_{j-1}$.

3. The predicted value $\varphi_m(x)$ for $x = t_j$ is computed using the Lagrange formula.

4. The residual (difference between actual and predicted value) is stored.

Since we expect that this difference is very small, the values are first converted to the integer representation and then subtracted: $c_j = \mathtt{Int}(a_j) - \mathtt{Int}(\varphi_m(t_j))$

Subsequently the necessary operations with c are performed just like in the Section 2.4.

The original sequence a can be unambiguously restored from the compressed sequence under conditions described in the Section 3.1.

5 Experiments

In this section we describe the experimental application of both our algorithms — higher order differences (suitable for fixed steps) (orders 2, 4, 6, 8, 10) and varying step extrapolation (of orders 1, 3, 5, 7, 9). The compression ratios are compared with two wavelet algorithms (Section 5.1). There were two major tests: artificially designed test sequences and output from a high-precision numerical simulation of a mechanical model using the ODE solver.

5.1 Experiments with wavelet-based algorithms

A widely used family of algorithms for numerical data compression are wavelet transforms. Without going into details about wavelet theory and taxonomy of transforms we just describe two transforms we experimented with.

Assuming that a sequence has some correlation between neighboring elements, the wavelet transform computes an "average" value s and "difference value" d. For arbitrary integer $q > 0$ the transform compresses a sequence $a_0, ..., a_Q$, where $Q = 2^{q+1} - 1$, by running through levels

$r = q, q-1, ..., 0$. On the level r the sequence considered is $a_0, ..., a_R$, where $R = 2^{r+1} - 1$. Furthermore there are certain rules defining how to compute the sequence for level $r-1$.

The simplest wavelet transform, **Haar** wavelet [1], computes on level p by the formulae

$$s_i = (a_{2i} + a_{2i+1})/2, \quad d_i = a_{2i+1} - a_{2i} \quad i = 0, ..., 2^r - 1$$

The number of bits needed for d_i is relatively small; lossy compression algorithms using wavelets might ignore it; the lossless algorithm stores them. The elements s_i become a_i on the next level of transform.

The sequence a can be recovered by $a_{2i} = s_i - d_i/2, a_{2i+1} = s_i + d_i/2$.

This algorithm is specially successful for sequences which change very slowly and close to a polynomial function. Mainly wavelet algorithms are used for lossy compression.

There is, however, a modification, called **TT-transform** [4], used for lossless compression: $s_i = \lfloor (a_{2i} + a_{2i+1})/2 \rfloor, d_i = a_{2i} - a_{2i+1} + p_i$,

where p_i is defined by $p_i = \lfloor (3s_{i-2} - 22s_{i-1} + 22s_{i+1} - 3s_{i+2} + 32)/64 \rfloor$,

and the sequence can be restored by $a_{2i} = s_i + \lfloor (d_i - p_i + 1)/2 \rfloor, a_{2i+1} = s_i - \lfloor (d_i - p_i)/2 \rfloor$

Both Haar and TT transforms were applied to compression and decompression. Just as in Section 2.4 six bits were always used to encode the length of the bit strings. The experiments show that the compression ratio for lossless compression is insufficient.

5.2 Artificially designed test sequences.

The test sequences for testing the algorithms were designed. These sequences contain 64-bit double precision numbers. We took into consideration that the sequence for the test cases should be rather smooth as a whole, but they should also contain small local non-smooth variations.

The size of the sequence N is chosen as 2^8 or 2^{16} (see Table 3). The longer sequence has smaller differences between adjacent elements and therefore is compressed better.

The sequence a with *fixed* time steps is defined as $a_i = F(i/N)$ where $i = 1 \dots N$ and

$$F(x) = 0.2 + 0.7x - 0.5x^2 + 0.007\cos(100x) + 0.00007\cos(10000x) + 0.1\sin(10x)$$

The time step is constant 1, i.e. $t_i = i$. The table shows that sequences with the fixed time step can be compressed equally well by both our algorithms (see repetitions in columns under "*fixed*")). The best ratio achieved is 3.68.

For testing of a sequence with *varying* time step we use $t_i = t_{i-1} + i\mathtt{mod}4 + 1$ and $a_i = F(2^{-16}t_i/2.5)$ where i and F are as above. Here time steps vary from 1 to 4. Therefore the algorithm using varying step extrapolation provides a better result than the algorithm using differences (3.73 versus 1.3).

		Ratio			
Time step:		fixed		varying	
Number of values:		2^8	2^{16}	2^8	2^{16}
Differences of order m (Section 2.4)	$m = 2$	1.58	1.64	1.45	1.53
	4	1.81	1.9	1.43	1.51
	6	2.12	2.27	1.33	1.42
	8	2.52	2.8	1.27	1.35
	10	3.13	**3.68**	1.23	1.3
Varying step m-th order extrapolation (Section 4)	$m = 1$	1.58	1.64	1.58	1.65
	3	1.8	1.9	1.81	1.91
	5	2.11	2.27	2.12	2.29
	7	2.51	2.8	2.54	2.83
	9	3.11	3.68	3.16	**3.73**
Wavelet	TT	1.7	1.86	1.23	1.67
	Haar	1.39	1.47	1.19	1.4

Table 3: Compression ratios for various data sequences and various algorithms.

5.3 Application to simulation results

The compression algorithms were applied to the output from a numerical solver of ordinary differential equations which serves as a component in our software for dynamic simulation of bearing [3]. The program produces some quantities for every time step and writes them to the output file for analysis and further simulation. Every quantity (position, force etc.) changes very slowly from one step to another. Extreme accuracy and lossless compression is necessary, since a relative error of order 10^{-10} can substantially change the simulation results.

To choose a particular algorithm and its order the compression routines estimate achievable compression ratio by subsampling, trying different algorithms and choosing the best one.

The 2837 arrays from a single simulation were analyzed. The compression ratio varies between 2.5 and 10. The algorithms were automatically chosen as follows:

— difference, first order - 20% of all arrays[12], second order - 3%, 3rd order - less than 1% of all arrays.

— varying step extrapolation, first order -10 %, second order - 17%, third order - 51%. If the 4th order extrapolation is proposed, it takes 30%, but compression ratio is almost the same as in the 3rd order extrapolation.

5.4 Lossy compression

There are four different applications of data saved by the compression algorithm:

— simulation can be restarted from any time step;

[12]These arrays are not smooth. It took 40 bits per 64-bit number to compress them. Compression ratio was 1.6.

— simulation results are used in another computation;

— intermediate simulation results are sent between nodes in parallel simulation;

— simulation results are visualized in form of 2D function graphs and 3D model animations.

Only the last application allows using lossy compression. Other applications require a lossless one.

The lossy compression is an extension of the basic algorithm. It can be parameterized in order to adjust the trade-off between the precision and the compression ratio.

The lossy compression can be achieved by cutting away some c bits at the end of the bit string representation. To compensate for the error, one exact value is followed by some p lossy compressed values.

The user would be interested to choose the pair (c,p) for given sequence a in such a way that during decompression the absolute and relative error do not exceed ε_{abs} and ε_{rel} correspondingly.

There is a straightforward way to do this, but it requires some extra computations during compression.

First we use an interval $[a_i^{min}, a_i^{max}]$, where $a_i^{min} = \min(a_i - \varepsilon_{abs}, a_i - \varepsilon_{rel}|a_i|)$, $a_i^{max} = \max(a_i + \varepsilon_{abs}, a_i + \varepsilon_{rel}|a_i|)$.

Subsequently interval arithmetic is used in order to obtain $[d_i^{min}, d_i^{max}]$. Then d_i is easily chosen from this interval in such way that it occupies the minimal possible number of bits.

6 Conclusion

A lossless algorithm for floating-point data compression has been developed. It has similarities to image compression since it works on bit level. It resembles wavelet compression since it uses floating-point computations for compression and decompression. The algorithm works best if the data are values of a function at some points, and this function is close to a polynomial.

The algorithm uses subtraction of one 64-bit integer representation of a floating-point value from another ($\texttt{Int}(a_j) - \texttt{Int}(\varphi_m(t_j))$). If the difference were computed between *floating-point* representations ($a_j - \varphi_m(t_j)$), there would be no gain in data storage.

The algorithm is implemented as a C++ class and linked to an industrial-level application. The measurements show a high compression ratio (in comparison with traditional tools) as well as high speed[2]. In the future we are going to test and measure the algorithm with data samples from other applications.

Acknowledgments

Professor Robert Forchheimer (ISY, Linköping University) contributed many suggestions regarding the algorithms discussed.

References

[1] A. Certain, J. Popovic, T. DeRose, T. Duchamp, D. H. Salesin, W. Stuetzle. Interactive multiresolution surface viewing. *Proceedings of SIGGRAPH 96*, in *Computer Graphics Proceedings*, Annual Conference Series, 91-98, August 1996, http://www.cs.washington.edu/research/projects/grail2/www/pub/abstracts.html #InterMultSurfView

[2] Vadim Engelson, Dag Fritzson, Peter Fritzson,*On Delta-compression Algorithm for Numerical Data from ODE-based Applications in Scientific Computing*
Technical report, Linköping Electronic Articles in Computer and Information Science, ISSN 1401-9841, Vol. 5 (2000), to appear.

[3] Dag Fritzson, Peter Fritzson, Patrik Nordling, Tommy Persson. Rolling Bearing Simulation on MIMD Computers. *International Journal of Supercomputer Applications and High Performance Computing*, 11(4), 1997.

[4] M. J. Gormish, E. L. Schwartz, A. Keith, M. Boliek, A. Zandi, Lossless and nearly lossless compression for high quality images *Proc. of IS&T/SPIE's 9th Annual Symposium*, Vol. 3025, San Jose, CA, February 1997.

[5] L. Råde, B. Westergren, *Beta - Mathematics Handbook*, Studentlitteratur and Chartwell-Bratt, 1988, p. 336.