Representing Numeric Data in 32 Bits While Preserving 64-Bit Precision

Radford M. Neal
Dept. of Statistical Sciences and Dept. of Computer Science
University of Toronto
http://www.cs.utoronto.ca/∼radford/
radford@stat.utoronto.ca

8 April 2015

Data files often consist of numbers having only a few significant decimal digits, whose information content would allow storage in only 32 bits. However, we may require that arithmetic operations involving these numbers be done with 64-bit floating-point precision, which precludes simply representing the data as 32-bit floating-point values. Decimal floating point gives a compact and exact representation, but requires conversion with a slow division operation before the data can be used in an arithmetic operation. Here, I show that interesting subsets of 64-bit floating-point values can be compactly and exactly represented by the 32 bits consisting of the sign, exponent, and high-order part of the mantissa, with the lower-order 32 bits of the mantissa filled in by a table lookup indexed by bits from the part of the mantissa that is retained, and possibly some bits from the exponent. For example, decimal data with four or fewer digits to the left of the decimal point and two or fewer digits to the right of the decimal point can be represented in this way, using a decoding table with 32 entries, indexed by the lower-order 5 bits of the retained part of the mantissa. Data consisting of six decimal digits with the decimal point in any of the seven positions before or after one of the digits can also be represented this way, and decoded using a table indexed by 19 bits from the mantissa and exponent. Encoding with such a scheme is a simple copy of half the 64-bit value, followed if necessary by verification that the value can be represented, by checking that it decodes correctly. Decoding requires only extraction of index bits and a table lookup. Lookup in a small table will usually reference fast cache memory, and even with larger tables, decoding is still faster than conversion from decimal floating point with a division operation. I present several variations on these schemes, show how they perform on various recent computer systems, and discuss how such schemes might be used to automatically compress large arrays in interpretive languages such as R.

Introduction

Numbers that originate from reading text in a data file (as well as other sources) often have a small number of significant decimal digits, scaled by a small power of ten. We may therefore hope to reduce memory usage, and possibly processing time, by representing large arrays of such numbers using a 32-bit representation, rather than the standard 64-bit binary floating point representation (IEEE Computer Society, 2008) that is used for numbers in programming languages such as R, and for "double precision" numbers in languages such as C. However, it is highly desirable that arithmetic involving such compactly-represented numbers produce exactly the same results as if they had a 64-bit floating point representation. This is particularly important if conversion to a compact representation is done automatically, as a programming language implementation decision that is meant to have no visible effect (except, of course, on space and time usage).

Requiring that numerical results be identical to those using a 64-bit floating point representation unfortunately eliminates the most obvious, and very likely fastest, compact representation — the 32-bit "single precision" floating-point representation that is also standard (IEEE Computer Society, 2008). Every number with up to seven significant decimal digits maps to a distinct 32-bit single precision value, with no information loss. However, when these single precision values are converted to 64-bit double precision in the standard (hardware-supported) way and then used in arithmetic operations, the results are in general not the same as if a 64-bit floating-point representation had been used. The problem is that the standard conversion by extending the mantissa of a single precision number with zeros does not produce the correct double precision representation of a number, such as 0.1, whose binary expansion is non-terminating.

One compact representation that can produce the same results as 64-bit binary floating point is decimal floating point. Cowlishaw (2003) has advanced several reasons for using a decimal floating point format, such as its ability to distinguish 1.2 from 1.200, but here I consider it only as a way of exactly representing some 64-bit binary floating point values in 32 bits. For this purpose, we might use a 28-bit signed integer, M, and a 4-bit unsigned exponent, e, to represent $M \times 10^{-e}$. The value $e = 15$ might be used to represent a "NaN" or "NA" value, as needed in R to represent missing values. Conversion to 64-bit binary floating point can be done by converting M to binary floating point (exactly), obtaining by table lookup the (exact) binary floating-point representation of 10^e (or a "NaN" value when $e = 15$), and finally performing a 64-bit binary floating-point division. If the decimal floating point value was read from a data file with eight or fewer significant digits, and its magnitude is representable within the exponent range of this scheme, the result of converting it to 64-bit binary floating point should match the result of directly converting the text in the data file, since the latter conversion should be done with exactly the same operations.

Unfortunately, the floating point division operation required to convert from a decimal floating point representation is quite slow (compared to other operations) on most current processors. Multiplication is typically much faster, but in a binary representation, multiplying by 0.1 is not the same as dividing by 10, because 0.1 does not have an exact binary floating point representation. Hardware support for decimal floating point might speed up this conversion to 64-bit binary floating point somewhat, but the need for division may limit the improvement possible. Such hardware support is in any case not common at present.

In this paper I present a framework for compactly representing subsets of 64-bit binary floating point values in 32 bits, from which conversion is faster than from decimal floating point on most current processors. The compact 32-bit representation is simply half the 64-bit floating point representation, consisting of the sign, exponent, and high-order part of the mantissa. Conversion to the full 64-bit floating-point value is done by filling in the low-order 32 bits of the mantissa from a lookup table, indexed by low-order bits of the retained part of the mantissa and in some cases by certain bits from the exponent. Different versions of this scheme allow for larger or smaller sets of useful values, with one tradeoff being the size of the lookup table needed, which affects conversion speed when the table is too large for the processor's memory cache. I investigate possible sets of represented values and associated performance tradeoffs with several recent processor architectures and compilers, in systems with varying cache memory sizes and varying processor/memory speed ratios. Finally, I discuss how such methods might be used in an implementation of an interpretive language such as R, in order to automatically (and invisibly) compress large numeric arrays.

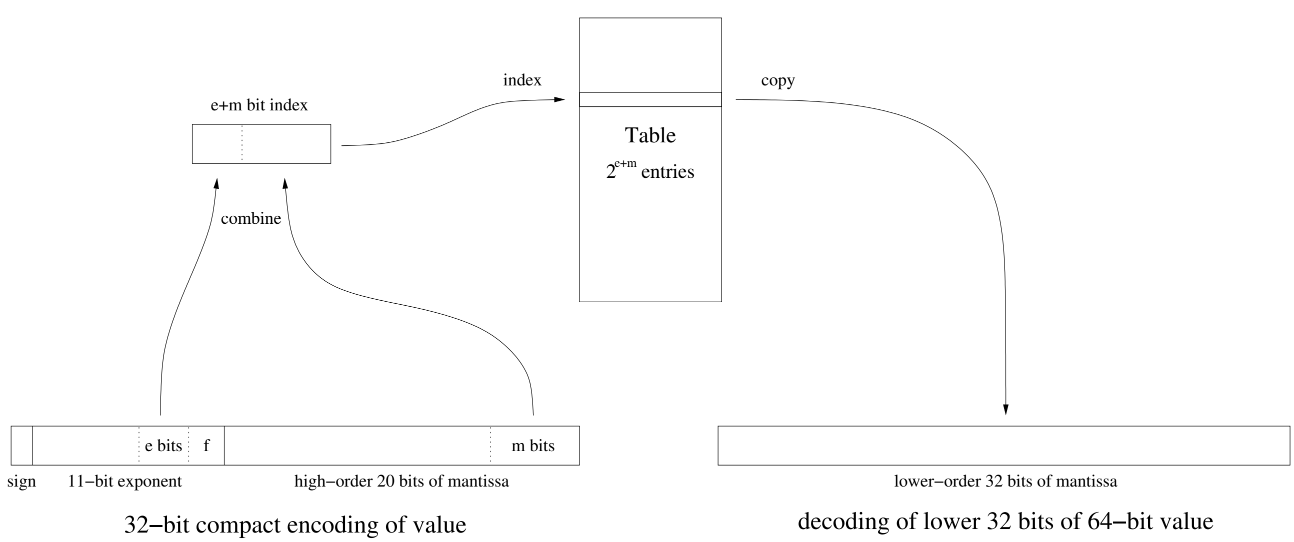
C programs for the methods and performance tests are provided as supplementary information.

Compact representation by half of a 64-bit floating-point value

Although the schemes developed here are motivated by a desire to compactly represent 64-bit floating point values that are derived from decimal representations, these schemes can be seen more generally as compactly representing numbers in a subset, $\mathcal{S}$, of 64-bit floating-point values that includes some desired subset, which I will denote by $\mathcal{D}$.

For all these schemes, the compact 32-bit representation of a 64-bit floating-point value is simply the 32 bits consisting of the sign, exponent, and high-order mantissa bits of the 64-bit floating-point value. I will consider only 64-bit floating-point values in the IEEE standard format (IEEE Computer Society, 2008), which consist of 1 sign bit, 11 exponent bits, and 52 mantissa bits, which give 53 effective bits of precision, since the highest-order bit is alway an implicit 1 (except for denormalized numbers). The 32-bit compact representation therefore has 1 sign bit, 11 exponent bits, and 20 mantissa bits (giving 21 bits of precision).

In all the schemes I consider here, the compact 32-bit value is converted (decoded) to a 64-bit floating-point value using m bits from the mantissa part of the compact value, and e (possibly 0) bits from exponent part, offset from the bottom of the exponent by f. These $m + e$ bits are used to index a table of 2^{m+e} possible values for the lower 32 bits of the mantissa. The decoded 64-bit floating-point value consists of the value looked up in this table together with the 32-bit compact representation. This procedure is illustrated here:



Pseudo-code is shown below for the decoding procedure, which takes as inputs a 32-bit compact value, t, and an array Table with 2^{m+e} elements, indexed starting at 0:

> Set i to the integer whose binary representation consists of bits $0 \ldots (m-1)$ of the mantissa part of t and bits $f \ldots (f+e-1)$ of the exponent part of t.
>
> Fetch Table[i], and concatenate it with t to produce the decoded 64-bit floating-point value.

Here, bits are numbered starting with 0 for the lowest-order bit. The bits from t combined to produce i may be combined in any consistent order, though using the bits from the mantissa as the low-order bits of the index (as in the illustration above) has implementation advantages. Note that in some schemes e is zero, in which case the m bits from the mantissa are used directly as the index.

Converting a 64-bit value to compact form is done by simply copying the 32-bit half that constitutes the compact representation. If the 64-bit value being encoded is not known to be in the set $\mathcal{S}$, the compact representation is then decoded, the decoded value is compared to the original 64-bit value, and failure is reported if these values do not match exactly.

To design a particular scheme in this framework, we need to decide which bits of the 32-bit compact value are used to index the lookup table — that is, choose m, e, and f — and what values this lookup table will contain. We aim to create a scheme that can represent any value in some set $\mathcal{D}$ — for example, $\mathcal{D}$ might consist of the 2000000 values of the form $\pm$`ddd.ddd`, where each `d` is a decimal digit from 0 to 9. The choice of m, e, and f can be done by trial and error, perhaps looking for the smallest value of $m+e$ for which a scheme that can represent all values in $\mathcal{D}$ is possible. For given m, e, and f, the procedure below can be used to fill in the lookup table, or to report failure if representing all values in $\mathcal{D}$ is not possible (with the chosen values of m, e, and f):

> For i from 0 to $2^{m+e}-1$, set Table[i] to 0 (or some other default value).
>
> For i from 0 to $2^{m+e}-1$, set Used[i] to False.
>
> For each v in $\mathcal{D}$ (in any order):
>
> > Set t to the 32 bits of v containing the sign, the exponent, and the high-order 20 bits of the mantissa.
> >
> > Set i to the integer whose binary representation consists of bits $0 \ldots (m-1)$ of the mantissa part of t and bits $f \ldots (f+e-1)$ of the exponent part of t.
> >
> > If Used[i] is True, report failure and stop.
> >
> > Otherwise, set Table[i] to the 32 lowest-order mantissa bits of v, and set Used[i] to True.

The way in which bits from t are combined to give i above must match the way that will later be used in the decoding procedure. The output of this procedure is the array Table, which is then used for decoding. The Used array is discarded once the design procedure has finished.

If this design procedure does not fail, all 64-bit values in $\mathcal{D}$ can be compactly represented in the scheme designed, and exactly decoded. In addition, unless all elements of Used are True, some 64-bit floating-point values that are not in $\mathcal{D}$ will also have compact representations. The full set of such values is the set $\mathcal{S}$, of which $\mathcal{D}$ will be a subset. Since any 32-bit compact value decodes to something, $\mathcal{S}$ will always contain 2^{32} members. The identity of those members of $\mathcal{S}$ not in $\mathcal{D}$ will depend on the choice of default value for table entries used in the first step of the procedure above.

A variation on the decoding procedure that sometimes reduces the space used for the decoding table is possible when the number of distinct entries in Table is no more than $2^{16} = 65536$. We can then use indirect indexing, replacing the 2^{m+e} 32-bit entries in Table with a table of 2^{m+e} 16-bit entries. The entry obtained from this table is then used to index a table containing the distinct 32-bit values in the original table. If the number of distinct entries in Table is much less than 2^{m+e}, this will reduce the memory required by almost a factor of two, and may reduce the time for decoding if table accesses are then more likely to reference cache memory.

Some decodable subsets

The utility of this approach to compact representation of 64-bit floating point values depends crucially on what subsets, $\mathcal{D}$, can be compactly represented, and for those subsets that can be represented, on the size of the decoding table, which will have 2^{m+e} entries, and hence occupy $4 \times 2^{m+e}$ 8-bit bytes (though this may be reduced if indirect indexing is used).

Some sets, $\mathcal{D}$, even if they have no more than 2^{32} members, cannot be compactly represented in a scheme of this type, for any values of m, e, and f. This will be the case if $\mathcal{D}$ contains two distinct 64-bit floating-point values for which the 32-bit portions used as the compact representation are identical. Other subsets, $\mathcal{D}$, may be decodable only with a large value for $m + e$.

The NA value used to represent missing data in R poses a problem in this regard. In current R implementations, it is a NaN ("Not a Number") value with mantissa bits set to the binary representation of the integer 1954, for which the upper 20 bits are all zero. With this choice for the NA value, the 20 bits of the mantissa that are included in its compact representation are identical to those for the number zero, so it would not be possible for both NA and zero to be in the set $\mathcal{D}$ in a scheme with $e = 0$. This issue will also restrict schemes in which $e > 0$. Fortunately, there appears to be no strong reason why NA in R could not be changed to a NaN value in which the upper 20 bits of the mantissa are all ones, and the lower 32 bits are the binary representation of 1954, and this is the choice for NA that I will assume below.

Table 1 shows several subsets, $\mathcal{D}$, that can be represented by a scheme in this framework using only mantissa bits for decoding, along with the required value of m and the resulting table size. For these schemes, indirect tables are pointless, since their total table size is greater than for a direct table.

Scheme A, which can compactly represent any positive or negative number of the form ddddd.d, such as 12345.6 or -888, requires only a a very small table of 8 entries. Schemes B through F represent

	Forms of Numbers	m	Table Entries	Distinct Entries	Bytes Using Direct Table
A	`ddddd.d`	3	8	6	32
B	`dddd.dd`	5	32	26	128
C	`dddd.` `ddd.ddd`	7	128	126	512
D	`ddd.d` `dd.dddd`	10	1024	626	4096
E	`dd.dd` `d.ddddd`	12	4096	3126	16384

F	`dd.`
	`d.ddd`
	`.dddddd`
		14	16384	15626	65536

Table 1: Some subsets of 64-bit floating-point numbers that can be represented compactly with decoding based on mantissa bits only (ie, with $e = 0$). All subsets include the NA value, as well as the numbers matching the decimal representations shown (plus their negations).

	Forms of Numbers	m	e	f	Table Entries	Distinct Entries	Bytes Using Direct Table	Bytes Using Indirect Tables
W	ddddd0.	10	4	1	16384	626	65536	35272
	ddddd.d							
	dddd.dd							
	ddd.ddd							
	dd.dddd							
X	dd0000000.	10	5	1	32768	9435	131072	69172
	dd000000.							
	dddd000.							
	dddddd.							
	ddddd.d							
	dddd.dd							
	ddd.ddd							
	dd.dddd							
	.000dd							
	.0000dd							
	.00000dd							
	.000000dd							
	.0000000dd							
	.00000000dd							
	.000000000dd							
Y	d0000000.	12	5	1	131072	5926	524288	285848
	dddd000.							
	dddddd.							
	ddddd.d							
	dddd.dd							
	ddd.ddd							
	dd.dddd							
	d.ddddd							
	.000ddd							
	.0000ddd							
	.00000ddd							
	.000000ddd							
	.0000000ddd							
	.00000000ddd							
	.000000000ddd							
Z	dd0000000.	14	5	1	524288	15626	2097152	1111080
	ddd00000.							
	dddd000.							
	dddddd.							
	ddddd.d							
	dddd.dd							
	ddd.ddd							
	dd.dddd							
	d.ddddd							
	.dddddd							
	.0000ddd							
	.00000ddd							
	.000000ddd							
	.0000000ddd							
	.00000000ddd							
	.000000000ddd							

Table 2: Some subsets of 64-bit floating-point numbers that can be represented compactly with decoding based on both mantissa bits and exponent bits. All subsets include the NA value, as well as the numbers matching the decimal representations shown (plus their negations).

subsets that allow for increasing numbers of digits to the right of the decimal point, but with the tradeoff of fewer digits to the left of the decimal point, and increased table size.

As shown in Table 2, substantially bigger subsets can be represented in schemes where e is non-zero, though at the cost of additional bit manipulation to combine index bits, and larger tables. For scheme W, the set $\mathcal{D}$ can be described as consisting of any six-digit number with from one to four of the six digits to the right of the decimal place, plus integers with six digits of which the last is zero. Schemes X, Y, and Z expand the set numbers that can be represented, including numbers spread over a considerable range of magnitudes, though with few significant figures, plus numbers having six digits with the decimal place in various positions (for scheme Z, before or after any of these six digits).

Note that these schemes can be extended to include additional useful numbers in their sets $\mathcal{D}$. For example, scheme Y can be extended to also include numbers of the form 1dddddd. and 1ddd.ddd, without any increase in direct or indirect table sizes. Of course, this results in some other numbers being removed from the set $\mathcal{S}$, but those other numbers may not be as useful.

Note also that schemes can be designed for sets $\mathcal{D}$ that are not defined in terms of decimal representations. For example, the 64-bit floating-point approximations to all rational numbers of the form n/m where n is an integer from -13332 to $+13332$ and m is an integer from 1 to 100 can be represented in a scheme with $m = 13$ and $e = 0$.

Software implementation

Some issues arise when implementing encoding and decoding operations for these schemes in software, particularly when writing in a high-level language and aiming at portability to different machine architectures. Here, I will consider only implementations written in C, and more specifically, following the revised C99 standard (International Standards Organization, 2007), in conjunction with the IEEE standard for floating point arithmetic (IEEE Computer Society, 2008).

The first issue is how to obtain the two 32-bit halves of a 64-bit floating-point value (a C `double`), which requires bypassing the C type system. The C "union" construct is designed to allow this, but its effect may depend on the particular machine architecture being used.

One option is to use a union of a `double` field with an array of two 32-bit unsigned integers, declared as `uint32_t`. However, which of the two 32-bit integers contains the upper bits of the 64-bit double value will depend on whether the machine architecture is "little-endian" (for example, the x86 architecture) or "big-endian" (for example, the SPARC architecture). An alternative that works on both architectures is to use a union of a `double` field with an 64-bit unsigned integer field declared as `uint64_t`, and to then access the upper and lower 32-bit halves of the `double` field by shift and mask operations on the `uint64_t` field. In my experience with modern processors and compilers, the latter approach is also generally faster, and never consistently slower, so I will adopt it here. Note that neither approach is guaranteed to work by the C99 standard, since the representations of double and unsigned integer values might not be consistently either big-endian or little-endian, but this does not appear to be a problem in practice.

When bits from the exponent are used for table lookup when decoding (that is, e is non-zero), they may be combined with the index bits from the mantissa in any consistent way. However, letting the bits from the mantissa be the low-order bits of the index saves a shift operation, and is adopted here. This order is also likely to produce better cache performance (assuming cache lines are larger than 32 bits), since when a series of values are decoded, their mantissa bits are likely to be more variable than their exponent bits.

Finally, it is crucial for good performance that encoding and decoding operations be performed inline, avoiding function call overhead. Using modern C compilers, with standard optimizations enabled, this can be accomplished by defining the small encoding and decoding functions, declared as `static inline`, within a header file included in modules that use them.

Performance on recent computer systems

The speed with which these compact representations can be decoded and used in arithmetic operations will depend on numerous aspects of the computer system used, such as the relative speeds of the processor and memory, the sizes of memory caches, and the degree to which the processor is capable of overlapping integer and floating-point operations with each other and with memory accesses. Here, I will report results of tests on a variety of fairly recent systems, providing some illustration of how such factors affect performance.

These tests are all of sequential operations on large vectors (that is, one-dimensional arrays). Operations on individual values, accessed non-sequentially, might show different behaviour. However, in the context of an interpretive language implementation, such operations may in any case be dominated by interpretive overheads. Also, all the tests performed here use a single execution thread. It would be interesting to look at performance when several threads are executing in parallel, perhaps with more than one accessing the same compressed data, but a meaningful performance evaluation in such a context is beyond the scope of this paper.

The compact representation schemes tested here are Scheme C from Table 1 and Schemes X and Z from Table 2. For Schemes X and Z, both direct table lookup and lookup with indirect indexing were tested. For comparison, the vector operations were also performed on uncompressed data, and on data represented in decimal floating point (using 4 bits to specify a non-negative power of ten and 28 bits for a signed integer scaled by dividing by that power).

Five operations were tested, each using two data distributions. As inputs, these operations take one, two, or three vectors of data, in one of the compact representations (or represented as uncompressed 64-bit floating point values for comparison). They produce as output a vector of uncompressed values, or a single uncompressed value, as follows:

copy	copy of the input
vector sum	sum of all values in the input vector
scalar $\times$ vector	product of 123.456789 times the input vector
vector + vector	vector sum of two input vectors
linear combination	linear combination of three input vectors with coefficients 1.1, 2.2, and 3.3

These operations were programmed in C, with inlined functions used for decoding the compressed representations. A high optimization level was used when compiling, with options to enable code generation optimized for the particular processor on which the code was run. The code, scripts, and output for the tests are available as supplemental information.

For the tests reported, all input vectors contained 3,000,000 randomly generated values. Two distributions for these values were tried. In the first, each value has the form `ddd.ddd`, with each `d` being a random digit from 0 to 9. In the second, the forms `dd.dddd`, `ddd.ddd`, and `dddd.dd` were cycled through, with each `d` again being random. Only the first distribution was used for Scheme C, since it cannot represent all values generated with the second distribution. Each operation, on data generated from each distribution, was repeated 100 times, and the total time was recorded in seconds (to three decimal places, though the last digit is often not meaningful due to random time variation).

The results for six computer systems are reported in Tables 3 to 8. These tables give the times in seconds. For methods other than use of uncompressed data, the times have above them the ratio (preceded by "×") of the time below to the corresponding time using uncompressed data. A ratio less than one indicates that for this combination of operation, data distribution, and compression scheme, operating on compressed data is faster than using uncompressed data. A ratio greater than one indicates the reverse, that operating on the compressed data is slower.

Tables 3 and 4 show performance on a high-end workstation and a low-end desktop computer, both of fairly recent vintage, both using x86 architecture processors (in 64-bit mode). Although the low-end system is roughly three times slower, the relative performances of the various representations are broadly similar for these two systems. This is perhaps surprising, given that the high-end system has 12 MBytes of cache, versus 512 KBytes for the low-end system, which one might think would make a substantial difference at least for the schemes with large decoding tables. However, the processor for the low-end system is substantially slower, while the memory is of the same type, so the low-end system has faster memory in relation to processor speed, which perhaps compensates for its smaller cache.

For the systems in Tables 3 and 4, Scheme C performs about the same as using uncompressed data — slightly faster for some operations, somewhat slower for others. This scheme therefore seems quite attractive when its set of representable values is sufficient. Scheme X provides a much larger set of useful values, but (using a direct table) it is somewhat slower than Scheme C, though it is still mostly less than a factor of two slower than using uncompressed data. Using indirect tables with Schemes X is slightly slower than using a direct table, though it is possible that the smaller amount of memory required for indirect tables will provide a compensating advantage in the context of an actual application, in which other operations are also done. Scheme Z provides a still larger set of useful values, but at the cost of a further moderate slowdown.

Schemes X and Z are slightly slower when the data is drawn from the second distribution rather than from the first distribution, presumably because the greater variety of data values causes accesses to the decoding tables to be more spread out, making memory caching less effective.

Since, on both of these systems, Schemes X and Z are slower than using uncompressed data, a decision to use one of them must be based on the advantage of decreased memory usage, the significance of which will depend on the wider context of the application and the system on which it runs. However, if we suppose that reducing memory usage is necessary, we can see that all of schemes C, X, and Z give faster performance than using decimal floating point, by a factor of more than three for Scheme X. One of these schemes should therefore be preferred when the set of values that they can represent is sufficient.

Tables 5 and 6 show results for two systems using variations of the Intel Core 2 Duo x86 architecture processor (in 64-bit mode), which are of somewhat earlier vintage than the processors used in the systems of Tables 3 and 4. The MacBook Pro system has a 2.4 GHz processor and 667 MHz memory. The Mac mini system has a 2.0 GHz processor (of slightly more recent design) and 1066 MHz memory. The results for this pair of systems therefore illustrate how the ratio of processor speed to memory speed affects performance.

On both these systems, Scheme C is faster than using uncompressed data for all operations except the vector sum. Scheme X using a direct table is also often faster than using uncompressed data, and Scheme Z using a direct table is mostly less than a factor of two slower than using uncompressed data. The performance (relative to using uncompressed data) of these compression schemes is more favourable for both these systems than for the somewhat newer systems of Tables 3 and 4, though it is unclear

	Data: ddd.ddd					Data: dd.dddd, ddd.ddd, dddd.dd				
	copy	vector sum	scalar × vector	vector + vector	linear comb.	copy	vector sum	scalar × vector	vector + vector	linear comb.
Uncompressed	0.490	0.310	0.470	0.650	0.860	0.500	0.300	0.470	0.660	0.870
Scheme C:	×0.86	×1.26	×1.02	×1.11	×1.42					
direct table	0.420	0.390	0.480	0.720	1.220	–	–	–	–	–
Scheme X:	×1.12	×1.81	×1.40	×1.60	×1.92	×1.12	×1.87	×1.45	×1.62	×1.94
direct table	0.550	0.560	0.660	1.040	1.650	0.560	0.560	0.680	1.070	1.690
	×1.16	×1.94	×1.51	×1.77	×2.13	×1.16	×2.00	×1.51	×1.76	×2.10
indirect table	0.570	0.600	0.710	1.150	1.830	0.580	0.600	0.710	1.160	1.830
Scheme Z:	×1.63	×2.42	×2.02	×2.26	×2.69	×1.90	×2.93	×2.32	×2.58	×3.03
direct table	0.800	0.750	0.950	1.470	2.310	0.950	0.880	1.090	1.700	2.640
	×1.55	×2.29	×1.91	×2.20	×2.57	×2.02	×3.07	×2.45	×2.82	×3.29
indirect table	0.760	0.710	0.900	1.430	2.210	1.010	0.920	1.150	1.860	2.860
	×4.61	×7.55	×5.77	×6.12	×7.98	×4.50	×7.80	×5.77	×6.03	×7.89
Decimal float	2.260	2.340	2.710	3.980	6.860	2.250	2.340	2.710	3.980	6.860

Processor:	Intel X5680, 6 cores, 3.33 GHz, launched 2010
RAM:	24 GBytes, DDR3 1333 MHz
Caches:	32 KByte I & D L1, 256 KByte L2, 12 MByte L3 (shared)
Software:	Ubuntu 12.04.5, gcc 4.8.1

Table 3: Performance on a Dell Precision T7500 high-end workstation.

	Data: ddd.ddd					Data: dd.dddd, ddd.ddd, dddd.dd				
	copy	vector sum	scalar × vector	vector + vector	linear comb.	copy	vector sum	scalar × vector	vector + vector	linear comb.
Uncompressed	1.879	0.710	1.718	2.302	2.639	1.859	0.713	1.697	2.282	2.617
Scheme C:	×0.63	×2.14	×0.87	×1.10	×1.66					
direct table	1.180	1.518	1.490	2.541	4.359	–	–	–	–	–
Scheme X:	×0.91	×2.80	×1.19	×1.72	×2.23	×1.03	×2.99	×1.32	×1.92	×2.47
direct table	1.705	1.988	2.040	3.949	5.858	1.923	2.130	2.243	4.374	6.467
	×0.96	×3.19	×1.27	×1.82	×2.42	×1.01	×3.21	×1.32	×1.89	×2.50
indirect table	1.812	2.267	2.179	4.196	6.372	1.870	2.289	2.239	4.321	6.545
Scheme Z:	×1.36	×4.18	×1.62	×2.49	×3.16	×1.75	×4.56	×2.07	×2.96	×3.66
direct table	2.557	2.971	2.787	5.740	8.299	3.262	3.251	3.515	6.756	9.588
	×1.61	×4.78	×2.02	×2.96	×3.77	×2.12	×5.83	×2.56	×3.74	×4.70
indirect table	3.028	3.392	3.468	6.814	9.904	3.938	4.160	4.345	8.538	12.309
	×3.86	×12.75	×6.43	×6.27	×10.11	×3.89	×12.70	×6.50	×6.34	×10.10
Decimal float	7.255	9.055	11.046	14.438	26.579	7.237	9.058	11.037	14.476	26.427

Processor:	AMD E1-2500, 2 cores, 1.4 GHz, launched 2013
RAM:	4 GBytes, DDR3L 1333 MHz
Caches:	32 KByte I & D L1, 512 KByte L2
Software:	Ubuntu 14.04.1, gcc 4.8.2

Table 4: Performance on a Gateway SX2185 low-end desktop computer.

	Data: ddd.ddd					Data: dd.dddd, ddd.ddd, dddd.dd				
	copy	vector sum	scalar × vector	vector + vector	linear comb.	copy	vector sum	scalar × vector	vector + vector	linear comb.
Uncompressed	1.617	0.479	1.611	2.030	2.643	1.618	0.503	1.608	2.034	2.651
Scheme C:	×0.88	×1.26	×0.87	×0.80	×0.79					
direct table	1.419	0.604	1.403	1.633	2.080	–	–	–	–	–
Scheme X:	×0.87	×1.76	×0.89	×0.88	×1.02	×0.87	×1.71	×0.88	×0.89	×1.03
direct table	1.412	0.845	1.429	1.794	2.701	1.409	0.860	1.418	1.802	2.731
	×0.88	×1.86	×0.89	×0.93	×1.11	×0.87	×1.79	×0.90	×0.93	×1.11
indirect table	1.415	0.893	1.437	1.884	2.943	1.408	0.901	1.443	1.889	2.939
Scheme Z:	×0.94	×2.06	×0.95	×1.11	×1.31	×1.01	×2.04	×1.02	×1.24	×1.36
direct table	1.520	0.988	1.528	2.262	3.462	1.639	1.027	1.647	2.514	3.602
	×0.93	×2.32	×0.97	×1.19	×1.52	×1.05	×2.48	×1.10	×1.42	×1.71
indirect table	1.503	1.095	1.555	2.421	4.030	1.703	1.246	1.767	2.883	4.532
	×2.45	×8.44	×2.45	×3.87	×4.53	×2.43	×8.03	×2.47	×3.88	×4.52
Decimal float	3.960	4.042	3.947	7.863	11.980	3.931	4.041	3.979	7.901	11.970

Processor:	Intel Core 2 Duo (T7700), 2 cores, 2.4 GHz, launched 2007
RAM:	4 GBytes, DDR2 667 MHz
Caches:	32 KByte I & D L1, 4 MBytes L2 (shared?)
Software:	Mac OS X 10.10.2 (Yosemite), clang 600.0.56

Table 5: Performance on an Apple MacBook Pro 3,1 laptop computer.

	Data: ddd.ddd					Data: dd.dddd, ddd.ddd, dddd.dd				
	copy	vector sum	scalar × vector	vector + vector	linear comb.	copy	vector sum	scalar × vector	vector + vector	linear comb.
Uncompressed	1.580	0.543	1.568	2.041	2.636	1.609	0.546	1.623	2.068	2.718
Scheme C:	×0.87	×1.29	×0.88	×0.80	×0.89					
direct table	1.370	0.702	1.381	1.639	2.333	–	–	–	–	–
Scheme X:	×0.88	×1.87	×0.91	×0.97	×1.21	×0.87	×1.89	×0.89	×0.98	×1.20
direct table	1.391	1.015	1.429	1.976	3.183	1.396	1.030	1.442	2.018	3.270
	×0.89	×1.98	×0.93	×1.04	×1.32	×0.88	×1.98	×0.90	×1.03	×1.29
indirect table	1.402	1.077	1.456	2.120	3.492	1.409	1.080	1.463	2.138	3.512
Scheme Z:	×0.98	×2.19	×1.03	×1.25	×1.57	×1.04	×2.26	×1.07	×1.28	×1.54
direct table	1.548	1.187	1.610	2.550	4.141	1.675	1.236	1.742	2.647	4.195
	×0.99	×2.45	×1.08	×1.36	×1.77	×1.17	×2.85	×1.23	×1.51	×1.86
indirect table	1.557	1.331	1.699	2.779	4.666	1.876	1.556	1.996	3.123	5.054
	×1.95	×5.64	×1.97	×3.00	×3.58	×1.91	×5.60	×1.90	×2.96	×3.47
Decimal float	3.084	3.060	3.084	6.125	9.428	3.081	3.060	3.087	6.126	9.434

Processor:	Intel Core 2 Due (P7350), 2 GHz, launched 2008
RAM:	2 GBytes, DDR3 1066 MHz
Caches:	32 KByte I & D L1, 3 MBytes L2 (shared?)
Software:	Mac OS X 10.10.2 (Yosemite), clang 600.0.56

Table 6: Performance on an Apple Mac mini 3,1 mid-range desktop computer.

	Data: ddd.ddd					Data: dd.dddd, ddd.ddd, dddd.dd				
	copy	vector sum	scalar × vector	vector + vector	linear comb.	copy	vector sum	scalar × vector	vector + vector	linear comb.
Uncompressed	4.190	4.700	4.960	8.950	13.650	4.180	4.700	4.960	8.950	13.660
Scheme C:	×1.38	×1.25	×2.52	×1.75	×1.76					
direct table	5.790	5.860	12.490	15.660	24.080	–	–	–	–	–
Scheme X:	×2.19	×1.89	×2.62	×2.52	×2.49	×2.56	×2.15	×2.94	×2.88	×2.11
direct table	9.170	8.880	13.020	22.560	34.010	10.710	10.090	14.570	25.780	38.840
	×5.06	×4.41	×2.74	×2.55	×2.50	×5.66	×4.92	×3.27	×3.14	×3.08
indirect table	21.210	20.710	13.570	22.830	34.170	23.670	23.130	16.210	28.060	42.110
Scheme Z:	×2.88	×2.37	×3.22	×3.20	×3.16	×2.94	×2.40	×3.26	×3.24	×3.20
direct table	12.080	11.150	15.950	28.680	43.160	12.280	11.260	16.150	29.000	43.650
	×6.01	×5.25	×3.61	×3.50	×3.46	×6.38	×5.56	×3.93	×3.86	×3.80
indirect table	25.190	24.670	17.910	31.350	47.170	26.680	26.110	19.470	34.560	51.950
	×4.18	×3.92	×3.78	×3.83	×3.90	×4.19	×3.91	×3.78	×3.83	×3.90
Decimal float	17.500	18.420	18.750	34.250	53.240	17.510	18.400	18.750	34.240	53.240

Processors:	Two UltraSPARC T2 Plus processors, 8 cores each, 8 threads per core, 1.2 GHz, launched 2008
RAM:	16 GBytes, DDR2 667 MHz
Caches:	16 KBytes I L1, 8 KBytes D L1, 4 MBytes L2 (shared)
Software:	Solaris 11.2, Oracle Solaris Studio 12.4

Table 7: Performance on a Sun T5140 high-end multithread-optimized server.

	Data: ddd.ddd					Data: dd.dddd, ddd.ddd, dddd.dd				
	copy	vector sum	scalar × vector	vector + vector	linear comb.	copy	vector sum	scalar × vector	vector + vector	linear comb.
Uncompressed	5.066	1.969	5.069	7.670	10.012	5.057	1.970	5.058	7.671	9.917
Scheme C:	×1.30	×1.21	×1.31	×0.96	×1.12					
direct table	6.578	2.384	6.624	7.396	11.224	–	–	–	–	–
Scheme X:	×1.35	×1.87	×1.46	×1.42	×1.53	×1.40	×2.06	×1.54	×1.54	×1.67
direct table	6.831	3.673	7.405	10.864	15.293	7.104	4.068	7.812	11.836	16.537
	×1.39	×2.22	×1.41	×1.46	×2.11	×1.45	×2.44	×1.49	×1.63	×2.31
indirect table	7.025	4.362	7.162	11.171	21.077	7.333	4.803	7.537	12.528	22.929
Scheme Z:	×1.89	×3.73	×2.19	×2.29	×2.49	×2.41	×4.56	×2.83	×2.89	×3.16
direct table	9.600	7.346	11.089	17.550	24.934	12.164	8.992	14.301	22.159	31.314
	×2.13	×5.50	×2.24	×2.90	×4.03	×2.65	×6.98	×2.77	×3.56	×5.06
indirect table	10.805	10.839	11.371	22.242	40.300	13.422	13.758	13.995	27.315	50.187
	×1.46	×3.43	×1.67	×1.88	×2.15	×1.46	×3.43	×1.67	×1.88	×2.17
Decimal float	7.397	6.758	8.452	14.400	21.537	7.400	6.758	8.456	14.401	21.538

Processor:	ARM v7 Cortex-A9, 4 cores, 1 GHz, launched 2008
RAM:	2 GBytes, DDR3 1066 MHz
Caches:	32 KByte I & D L1, 1 MBytes L2 (shared)
Software:	Debian Linux, gcc 4.8.2

Table 8: Performance on a Cubox i.MX6 miniature computer.

whether the difference is due to any systematic change in more recent processor designs.

The performance of Schemes X and Z is somewhat better on the MacBook Pro than on the Mac mini, in line with expectations of the effect of the higher processor/memory speed ratio for the MacBook Pro. The magnitude of the difference is perhaps less than would be expected from naively assessing this speed ratio using clock rates, however.

For both the MacBook Pro and the Mac mini, Schemes C, X, and Z are all considerably faster than using decimal floating point. Decimal floating point is relatively better on the Mac mini than on the MacBook Pro, presumably due to some improvement in the floating point division method (or in its ability to overlap with other operations), but it remains slower than Scheme X (with a direct table) by factors of between two and three for the various operations

Tables 7 and 8 show results on two systems whose processors do not use the (very common) x86 architecture. The Sun T5140 uses T2 Plus processors that implement the SPARC V9 architecture in a manner designed to support many concurrent threads (128 for the dual-processor T5140), by use of within-core multithreading. It is intended for use as a server that processes a large volume of transactions handled in multiple threads; its single thread performance is slower than earlier SPARC systems that did not have within-core multithreading. The Cortex-A9 implements the ARM v7 architecture. It is used in many tablet computers and smart phones, though the Cubox system tested here is a miniature computer that can be used as a desktop computer or home entertainment system.

On the T2 Plus processor in the T514, the interleaving of many threads within one processor core produces the effect that each thread sees memory as being relatively faster compared to the speed with which the thread executes instructions than would the case in a processor without such interleaving. One would expect that the advantage of using a compressed representation that comes from a reduction in the amount of data transferred would then be less, at least in a test such as this in which only a single thread is executing. And indeed, we see that unlike the x86 systems in Tables 3 to 6, operations using Scheme C are always slower than when using uncompressed data, by at least a factor of 1.25. The relative performances of Schemes X and Z are also worse than for those systems. However, these schemes (with direct tables) are nevertheless faster than decimal floating point (though by fairly small factors).

The Cubox system's Cortex-A9 processor has a low 1 GHz clock rate, but reasonably fast 1066 MHz memory, which may explain why the performance relative to using uncompressed data of Schemes C, X, and Z is generally worse than for the four x86 systems (though better than for the T5140). Puzzlingly, though, the relative performance of decimal floating point is better on this system than on any of the others, though it is still a bit slower than Scheme X.

Automatic use in an interpretive programming language implementation

My principal motivation for investigating these compact representation schemes is their possible use in an interpretive implementation of a language such as R (R Core Team, 2015). The goal is for the implementation to switch automatically to a compact representation for large numeric vectors, matrices, or other objects when such a compact representation is possible, with this switch having no visible effect, apart from its impact on speed and memory usage. In particular, I hope in future to include some version of such a scheme in the pqR implementation of R (Neal, 2013–2015).

Use of a compact representation scheme could instead be implemented as a visible option, which could be explicitly requested by an application program, after which it would be responsible for explicitly handling encoding and decoding operations. One can also imagine using these compact representations for other purposes, such as reducing the space needed to store large databases, either automatically or as an explicit option. In this paper, however, I will consider only use of these schemes for automatic compression of objects stored in main memory.

In this context, the intent is that application programs will not be aware of whether numbers are represented in some compact scheme. The only exceptions to this that I envision are that there might be some way for an application program to request that an object be converted to a compact representation, if this is possible and has not already been done automatically, and there might also be a way of querying whether an object is compactly represented, purely as an aid for understanding performance issues.

Using a compact representation is worthwhile only when an object is large, and is possible then only when there is some compact coding scheme within which all numerical values in the object are representable. Run-time checks for whether an object is compactly represented will therefore generally be necessary. In a language implemented by compilation to native machine code, such checks would be present in numerous compiled code segments, and the overhead of these checks in both execution time and code size could be significant. Here, I consider only interpretive implementations, in which these checks will be present only in a limited number of places in the interpreter, and the time they take may be small compared to other interpretive overheads.

Nevertheless, modifying every code segment in the interpreter that accesses numerical values in order to add checks for whether these numbers are compactly represented would be arduous. Furthermore, R defines an interface by which an application program can call functions written in C, Fortran, or another language, which may access numeric data that could also be compactly represented. If existing code is to continue to work, it is necessary to provide a default interface in which compactly represented data is never seen. Selected parts of the interpreter could then be modified to be aware of compact representations, and to implement operations using them without conversion. The application program interface might also be extended to provide a way for user-written code to manipulate data that is compactly represented, while still allowing existing user code to be used without change, since objects would automatically be converted to the standard representation if they are passed to such code.

In the pqR implementation of R, a similar issue already arises with objects containing numerical values that may not have been computed yet, either because the computation is being done, or might be done, in some other concurrently-executing thread, or because the computation has been deferred in the hope that it may later be merged with a subsequent computation. This is handled in pqR by making the default versions of functions that return an object — such as by fetching the value of a variable, or by evaluating an expression — wait for any computation on the object to finish before returning it. Code segments that are prepared to handle objects whose computation is pending can, however, call alternative versions of these functions, which will not wait for computations to finish.

A similar approach could be used to handle compactly represented objects. Code that is not aware of compact representations will call the default functions, which will automatically convert any compact object to its standard format. Code that is prepared to operate on compactly represented data can call alternative versions of these functions that do not convert compact data. With this approach, the use of compact representations will be beneficial only if they are mostly manipulated by code that is aware of such representations, so that they can remain in their compact representation, but correct operation does not depend on every code segment in the interpreter and in application-program C code being modified to deal with compact representations.

Conversion from a compact representation to the standard representation will also be necessary if a numeric element in the object is replaced by one that is not in the set, $\mathcal{S}$, of values that can be represented in the scheme being used.

The obvious way to convert from a compact representation to the standard representation is to allocate new space to hold the converted object, decode the compact representation to the standard representation, and then free the space that held the compact representation.

This approach has several problems, however. First, the amount of memory required will temporarily be 50% larger than if the standard representation had been used from the beginning. Second, it is not clear what to do if the object is shared — for example, is the value of several variables. In R, such sharing is done only to save space, and R semantics require that a copy be made when one value is modified, so the other will remain unchanged. It is unclear whether in this context it would be best to replace all references to the compact representation by the converted representation, or only the reference for which the standard representation is now required. If all references are replaced, they would have to be found, which is not easy. But if only one reference is replaced, memory for both representations will remain allocated. Finally, decoding objects to newly-allocated space changes the address of the object, and also could trigger a garbage collection at the point where an operation prompting the conversion is done. Existing code may have been written in a way that assumes such changes in address and initiations of garbage collection do not happen at times when they now could.

A different approach therefore seems to be required, and is possible if the operating system supports allocation of space within the virtual address space of the process without physical memory being allocated until the addresses are actually referenced. Current Linux, Solaris, and Mac OS X operating systems (and likely others as well) have such a facility, and use it in the malloc function in the C library when a large block of memory is requested.

In this approach, when an object is created, space sufficient to hold its standard representation is always reserved for it in the virtual address space. If a compact representation is used initially, only the first half of this space will be referenced, and hence the second half will not occupy any physical memory. If later the object needs to be converted to the standard representation, it can be decoded in place (starting at the end), at which time physical memory will be assigned to the second half. The address of an object stays the same when it is converted, and garbage collection will not be triggered by the conversion operation, since no additional (virtual) memory is allocated. This makes handling automatic conversion to standard format quite analogous to the deferred evaluation mechanism that is already implemented in pqR.

Similarly, when an object that is currently in the standard representation is converted to a compact representation, its virtual address and virtual storage allocation can remain the same, while the physical memory that is no longer required after conversion can be released (if the operating system supports this).

This approach to implementing compact representations will produce the desired benefits as long as physical memory is the limiting resource, with the available virtual address space being larger than physical memory. (A virtual address space a factor of two larger than physical memory should always be sufficient, and in some applications a smaller factor might suffice.) The virtual address space can be expected to be sufficiently large in 64-bit architectures (although considerations other than pointer size might limit it to less than 2^{64} bytes). Systems with a 32-bit address space and no more than 2 GBytes of physical memory should also benefit from using compact representations. However, in a system that combines a large physical memory with a limited 32-bit address space, this approach to implementing compact representations may produce no benefit.

An unusual aspect of the framework for compact representations in this paper is that although there are many different schemes, such as seen in Tables 1 and 2, the compact representation of a number is identical for every such scheme. Because of this, a compactly represented object can be noted as being encoded by not just one such scheme, but by a set of schemes. Decoding according to any of these schemes will produce the same result, so whichever scheme has the fastest decoder can be used. When a numerical value in the object is changed, any schemes that cannot represent this new value will be removed from the set of schemes that can decode the object. If this set becomes empty, the object will have to be converted to the standard, non-compact representation.

Note in particular that when creating an object from data read from a file, compact representations of numbers from the file can be stored as they are read, even though the set of schemes that can be used for decoding these representations will not be known until all data has been read. If a number is read that cannot be represented in any of the schemes that are implemented, the compact representations of previously read numbers can be expanded, and the standard representation used for the remaining data.

Acknowledgements

This research was supported by Natural Sciences and Engineering Research Council of Canada. The author holds a Canada Research Chair in Statistics and Machine Learning.

References

Cowlishaw, M. F. (2003) "Decimal Floating-Point: Algorism for Computers", in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic.*

IEEE Computer Society (2008) *IEEE Standard for Floating-Point Arithmetic.*

International Standards Organization (2007) Combined document with C99 + TC1 + TC2 + TC3, available from `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`

Neal, R. M. (2013–2015) pqR — a pretty quick version of R, `http://pqR-project.org`

R Core Team (2015) *R Language Definition (Draft)*, available at `http://r-project.org`