Highlights of Computer Arithmetic

Nelson H. F. Beebe
University of Utah
Department of Mathematics, 110 LCB
155 S 1400 E RM 233
Salt Lake City, UT 84112-0090
USA
Email: `beebe@math.utah.edu`, `beebe@acm.org`, `beebe@computer.org`
WWW URL: `http://www.math.utah.edu/~beebe`
Telephone: +1 801 581 5254
FAX: +1 801 581 4148

04 November 2019
Version 1.00

Contents

List of Tables

Caution

A complete treatment of computer arithmetic requires a large textbook, and the subject is *much more difficult* than most courses and books on computer programming suggest to the student.

1 Introduction

This document is a supplement to a separate book chapter by this author, *Computer Arithmetic: Perils, Pitfalls, and Practices*, that expands upon many of the points mentioned here. However, we go beyond that chapter's material by treating several applications of floating-point arithmetic.

2 Exact mathematics versus computer arithmetic

Computer arithmetic is only an approximation to the mathematical arithmetic that pupils learn in school, and it is important to recognize the major differences:

- Mathematical numbers have infinite precision and range, but computer numbers have *limited precision* and *limited range*. For example, the C type `double` that is common on modern computers can represent numbers with about 15 decimal digits of precision, and a nonzero range from about 10^{-308} to 10^{+308}.
- For real and complex *numbers*, the *commutative law* holds in both mathematics and computer arithmetic (except for multiplication on some historical Cray supercomputer models):

  $$\begin{aligned} a + b &\equiv b + a, \\ a - b &\equiv -(b - a), \\ a \times b &\equiv b \times a, \end{aligned}$$

  as long as intermediate and final results are representable.
- While computer arithmetic is *exact* when all computed results are representable, the *associative law of addition* does *not* hold if there

is intermediate overflow, underflow, or significance loss:

$$\begin{aligned} a + (b + c) &\neq (a + b) + c \\ &\neq (a + c) + b. \end{aligned}$$

Here is a simple counterexample: compute `R = A + B + C` in IEEE 754 64-bit binary arithmetic when `A` is `1e300`, `B` is `-A` exactly, and `C` is `1e280`. The result is either `0`, or `1e280`: they are not even remotely close!

If we cannot prove correctness of a simple three-term sum, then proving an entire numerical program correct may be a hopeless task.

- Floating-point and integer negation has the expected behavior:

$$a \equiv -(-a).$$

However, negation does *not* hold in two's complement integer arithmetic for the most negative integer: $-(\texttt{INT\_MIN}) \to \texttt{INT\_MIN}$. Its true value is $\texttt{INT\_MAX} + 1$, but that is not representable.

That means that the common idiom in C for an absolute value function

```
#define ABS(x) (((x) < 0) ? -(x) : (x))
```

fails to return the absolute value when `x == INT_MIN`.

- Scaling floating-point numbers by a power of the base is always an *exact operation*, as long as the result does not underflow or overflow. However, if that power cannot be represented exactly as a constant in source code, then it must be generated by a library function. In C, library functions with these prototypes provide such scalings:

```
#include <math.h>

float ldexpf        (float, int);
float scalbf        (float, float);
float scalbnf       (float, int);

double ldexp        (double, int);
double scalb        (double, double);
```

```
    double scalbn       (double, int);

    long double ldexpl  (long double, int);
    long double scalbl  (long double, long double);
    long double scalbnl (long double, int);
```

It is *not advisable* to use the general power function for such scalings, because the uneven implementation quality of the C function `pow(x,y)` and the Fortran operator `x**y` might result in unwanted changes in significand digits.

- Poor implementations of input conversion in many programming languages make it impossible to supply the most negative integers as input, even though that is trivially handled by minor changes in the software. Thus, to represent the most negative 32-bit integer in source code, you might have to replace

```
    const int32_t INT32_MIN = -2147483648;
```

by

```
    const int32_t INT32_MIN = (-2147483647 - 1);
```

or by a type-coerced hexadecimal equivalent using one of these:

```
    const int32_t INT32_MIN = (int32_t)( 0x80000000U);
    const int32_t INT32_MIN = (int32_t)(~0x7fffffffU);
```

- In mathematics, nonzero / zero is generally viewed as ∞ (*Infinity*). For integer arithmetic, there is no representation of Infinity, and CPU designs may raise an interrupt, or just return the numerator, or an unpredictable value. For floating-point arithmetic, some historical designs, and IEEE 754, have a representation of Infinity; others may just abort the job.

- In mathematics, zero / zero is generally viewed as *undefined*, and disallowed. For integer arithmetic, computer behavior is likely to be the same as for a nonzero numerator. For floating-point arithmetic, some historical designs, and IEEE 754, have a representation of *Indefinite* (CDC and Cray) or *NaN* (IEEE 754 Not-a-Number).

- Definitions of the behavior of integer modulo and remainder operations are language dependent, and for some languages, platform dependent, when any operand is negative.

- Base conversion causes interminable difficulties. A human inputs 0.1 to computer program, then has the program print the result, gets 0.0999998, and wants to know: why?

- Numerical programs often process large amounts of data, and do enormous numbers of arithmetic operations. Failure to understand the *computer memory hierarchy* can have huge performance effects if the data are not handled properly.

  Modern computer systems have multiple levels of memory, from registers in the CPU, to one or more levels of cache memory, to local DRAM (dynamic random access memory) on the current CPU, to global DRAM on other CPUs, to memory on other computers accessed over the network, or to local or remote storage devices.

  Typical CPU clock speeds are 100 MHz (one cycle = 10 nsec) to 5000 MHz (one cycle = 0.2 nsec).

  From fastest to slowest, the computer memory hierarchy looks like this:

  Registers : 8 to 128 registers, of sizes 4 to 16 bytes each, often with different register sets for integer and floating-point arithmetic.
  Access time: 1 clock cycle.

  L1-cache : Typical size: 32KB, maybe separate instruction and data caches, one per CPU, shared by all its cores.
  Access time: 3 to 5 clock cycles.

  L2-cache : Typical size: 256KB, one per CPU, shared by all its cores.
  Access time: 5 to 20 clock cycles.

  L3-cache : Typical size: 20480KB, shared by all CPUs and cores.
  Access time: 20 to 50 clock cycles.

  DRAM : Typical sizes: 0.25MB to 32TB, shared by all CPUs and cores, but perhaps partitioned by CPU, with different access times for local and global parts.
  Access time: 500 clock cycles.

  Filesystem : Typical sizes: 1GB to 1PB.
  Access time: 10,000 to 100,000 clock cycles for one *block* (512B to 1MB).

Network : Access time from 5,000 cycles to minutes, depending on network load, latency, and hop count, for one block (1B to 1500B).

- Array storage order is language dependent, and may even be compiler dependent in some languages. Consider a two-dimensional matrix, `M`, of `R` rows by `C` columns:

  Ada : Storage order unspecified.

  awk : Indexing by integers or character strings, with no declared bounds. Elements are referred to as `M[i,j]` or as `M[i SUBSEP j]`, but are allocated only when assigned to. Each element is found by hash search (cost: `Order(1)`), and memory locations of logically sequential elements are unpredictable, and uncontrollable. An unassigned element silently evaluates to an empty string, which is treated as zero in a numeric context. For example, set `Phone["Mary"] = "+1 800 555-6789"` to allow later retrieval of her telephone number.

  Fortran : Column order, indexed from 1 by default. Contiguous allocation, with `M(i,j)` followed by `M(i + 1,j)`, and `M(R,j)` followed by `M(1,j + 1)`.

  C, C++, C#, Go, Rust : Row order, always indexed from 0. Contiguous allocation with `M[i][j]` followed by `M[i][j + 1]`, and `M[i][C - 1]` followed by `M[i + 1, 0]`.

  Java : Always indexed from 0. Row allocation: vector of `R` pointers to contiguous row vector blocks, each of length `C`. `M[i][j]` is followed by `M[i][j + 1]`, but the successor of `M[i][C - 1]`, while logically `M[i + 1][0]`, may be far away in memory, perhaps even on a nonresident memory page that requires loading from external storage.

  Julia : Column order, indexed from 1. Contiguous allocation, with `M[i,j]` followed by `M[i + 1,j]`, and `M[R,j]` followed by `M[1,j + 1]`.

The existence of a huge body of free numerical software written in Fortran, the modern preference of many programmers for C and C++ over Fortran, the availability of Fortran-to-C translators, and language-independent argument passing conventions on most modern system, mean that it is now possible to translate Fortran code to

C, and to call Fortran functions and subroutines from C. However, that Fortran code was likely written by an expert concerned with performance who carefully chose loop order to minimize cache collisions. If Fortran `M(i,j)` is treated as `M[i][j]` in C, then the storage order in C conflicts with the Fortran ordering. Fortran-to-C translator software does nothing to fix such problems.

I once made a Fortran program run *three times faster* by changing a *single digit* in an array dimension: `M(256,C)` to `M(257,C)`. That change radically reduced the number of cache reloads as the matrix was accessed by columns.

For matrix multiplication, it may be much faster to transpose one of the matrices in place, compute their product in optimal storage order, and then transpose the chosen one back again. Transposition has cost `Order(`n^2`)`, while matrix multiplication, and several other important matrix operations, has cost `Order(`n^3`)`. We investigate that in Section 26 on page 52.

Conclusion: *Reasoning about the correctness, and efficiency, of numerical software can be extremely difficult, especially when compilers in some languages may reorder numerical expressions as if they were mathematical ones, when in fact they are not!*

3 Integer arithmetic

- Choice of base (now ‘always’ 2) in compiled languages (Maple is an exception: it uses base 10).
- base digit representation:

binary : `[01];`

octal : `[01234567];`

decimal : `[0123456789];`

hexadecimal : `[0123456789abcdef]` (lettercase is ignored);

base 36 : `[0123456789abcdefghijklmnopqrstuvwxyz]` (lettercase is ignored);

base 64 :
`[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`
`0123456789+/]` (lettercase is *significant*).

- Signed and unsigned integer arithmetic support is language dependent:

  Ada : signed in 8-, 16-, 32-, and 64-bit sizes;

  awk : stored as signed 64-bit floating-point values;

  C/C++ : signed and unsigned in 8-, 16-, 32-, (with C99) 64-, and rarely, 128-bit sizes;

  C#/Go : signed and unsigned in 8-, 16-, 32-, and 64-bit sizes;

  Fortran : signed in 8-, 16-, 32-, and 64-bit sizes;

  Java : signed in 8-, 16-, 32-, and 64-bit sizes;

  Julia : signed and unsigned in 8-, 16-, 32-, 64-, and 128-bit sizes, plus arbitrary-precision signed integers;

  Rust : signed and unsigned in 8-, 16-, 32-, 64-, and 128-bit sizes.

- Programming support for nondecimal integers is language dependent, and rules may differ for source code and input/output. For integer constants in source code, we find:

  Ada : any base from 2 to 16 (e.g., `2#101#`, `8#377#`, `10#255#`, `16#cafe_feed#`);

  awk : base 10;

  C/C++/Go/Java : bases 8, 10, and 16;

  C# : bases 10 and 16;

  Fortran/Julia/Rust : bases 2, 8, 10, and 16.

- Choice of representation: sign-magnitude (with +/-0), one's complement (with +/-0), two's complement (with only one zero, but one more negative, so `abs(x)` is negative if `x` is `INT_MIN`).

  To negate in sign-magnitude: flip sign bit.

  To negate in one's complement: flip all bits.

  To negate in two's complement: flip all bits, then add one.

  All modern CPUs use *two's complement* arithmetic for hardware integers.

- Implications for fast even/odd test:

```
if (x & 1)
    print "x is odd";
```

That is correct *only* for sign-magnitude and two's complement arithmetic. It holds in one's complement arithmetic only for positive nonzero values.

- Integer arithmetic is the building block for multiple precision arithmetic. The latter is common in Lisp, and computer algebra systems, such as Axiom, Maple, Mathematica, and Maxima.

```
% maple
    |\^/|     Maple 2019 (X86 64 LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, ...
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(Pi, 5);
                         3.1416

> evalf(Pi, 10);
                       3.141592654

> evalf(Pi, 25);
               3.141592653589793238462643

> evalf(Pi, 50);
    3.1415926535897932384626433832795028841971693993751

> Digits := 75:

> evalf(sin(22));
-0.008851309290403875921690256815772332463289203951332\
     5664423308352980895201463

> evalf(Pi,500);
3.1415926535897932384626433832795028841971693993751058\
    20974944592307816406286208998628034825342117067982\
    14808651328230664709384460955058223172535940812848\
    11174502841027019385211055596446229489549303819644\
    28810975665933446128475648233786783165271201909145\
    64856692346034861045432664821339360726024914127372\
    45870066063155881748815209209628292540917153643678\
    92590360011330530548820466521384146951941511609433\
    05727036575959195309218611738193261179310511854807\
    44623799627495673518857527248912279381830119491
```

- Integer arithmetic was implemented in floating-point arithmetic on some historical systems (CDC, Cray), with concomitant irregularities (e.g., 48-bit integer hidden in 60-bit floating-point values).

- Behavior on zero divide, overflow, left/right and signed/unsigned shifts by more than word-size bits, are generally left up to the implementation, or defined to have undefined, or implementation-dependent, behavior (e.g., C, C++, Fortran, Go, ...)

  In C#, integer overflow is not caught unless the `checked` qualifier is prefixed to the expression.

  In Java, integer exceptions are never raised, except for zero divide.

- Many CPU designs can set one or more exception flags, or otherwise signal an error, on integer overflow, but some do not (e.g., DEC Alpha). MIPS CPUs set the integer overflow flag on signed add/subtract, but not on multiply; unsigned operations *never* set the flag. That variability means that ISO language standards designers are unlikely to provide standard functions for access to integer exception flags, sigh....

- Ranges for n-bit integers:

  sign-magnitude : $[-(2^{n-1}-1), +2^{n-1}-1]$, with both -0 and $+0$;

  one's complement : same as sign-magnitude;

  two's complement : $[-(2^{n-1}), +2^{n-1}-1]$;

  When both -0 and $+0$ exist, they compare equal, but have different bit patterns.

  Table 1 shows the ranges of common computer integer sizes, and you need to consider those limits when you write software to do integer arithmetic on things like these:

  - time in nanoseconds or seconds,
  - populations (fauna, flora, stars, taxpayers, ...),
  - vehicle license plate numbers,
  - computer memory addresses,
  - Internet IPv4 32-bit and IPv6 128-bit addresses,
  - financial accounts and national debts in various currencies, and so on.

Table 1: Ranges of integer sizes in two's complement arithmetic.

Bits	Signed Range	Unsigned Range
8	$[-128, +127]$	$[0, +255]$
16	$[-32768, +32767]$	$[0, +65535]$
18	$[-131072, +131072]$	$[0, +262143]$
32	$[-2147483648, +2147483647]$	$[0, +4294967295]$
36	$[-34359738368, +34359738368]$	$[0, +68719476735]$
48	$[-140737488355328, +140737488355328]$	$[0, +281474976710655]$
64	$[-9223372036854775808, +9223372036854775808]$	$[0, +18446744073709551615]$
72	$[-2361183241434822606848, +2361183241434822606848]$	$[0, +4722366482869645213695]$

4 Fixed-point arithmetic

- Generally sign-magnitude with fractional point location chosen in a particular language or machine architecture.
- Rare in hardware and modern programming languages (exceptions: Ada, Cobol, PL/1, METAFONT, and TEX).
- TEX dimensions are stored in an n-bit integer that has a 1-bit sign, a $(n-17)$-bit integer, and a 16-bit fraction. The bit adjacent to the sign is used for overflow detection on multiply, but overflow is not caught in addition.
- TEX's maximum dimension is named `\maxdimen`, and with a 32-bit integer, its value is 16383.99998pt, or about 18.89 feet, or 5.75 meters. You cannot typeset an e-book as a single galley without dealing with that limit.
- For TEX dimensions with a 32-bit integer, we find this output, with comments added to indicate unusual behavior:

```
% tex
This is TeX, Version 3.14159265 (TeX Live 2019) \
(preloaded format=tex)
**\relax
```

```
*\dimen0 = \maxdimen
*\showthe \dimen0
> 16383.99998pt.

*\dimen1 = \dimen0
*\dimen1 = 2\dimen0  % DETECTED overflow
! Dimension too large.
*\showthe \dimen1
> 16383.99998pt.     % assignment suppressed on overflow

*\dimen1 = \dimen0
*\advance \dimen1 by \dimen0
*\showthe \dimen1
> 32767.99997pt.     % UNCAUGHT overflow into detection bit

*\advance \dimen1 by \dimen0
*\showthe \dimen1    % UNCAUGHT overflow, wraps to negative
> -16384.00005pt.

*\dimen1 = \dimen0
*\multiply \dimen1 by 2  % DETECTED overflow
! Arithmetic overflow.
*\showthe \dimen1
> 16383.99998pt.     % assignment suppressed on overflow
```

- Manual rescaling, and detection and prevention of overflow, in fixed-point arithmetic are huge problems for programmers, and for that reason, floating-point arithmetic was soon introduced to computers, and is now universal in CPUs used for desktop and server computing.

- Exponent-free fixed-point representation remains common for specifying floating-point numbers in input, output, and software.

5 Floating-point arithmetic

- Common format: sign, exponent, and significand with sizes of each set by hardware design, or by international standard (e.g., IEEE 754-

2019). The relative storage layout of those fields has varied across CPU designs, but the given order is now universal.

- Design choices for exponent:
  - exponent sign + magnitude?
  - one's complement?
  - two's complement?
  - biased magnitude?
- Design choices for significand:
  - integer?
  - fraction in $[0, 1)$?
  - fixed in $[0, B)$ (base-B)?
  - normalized (no leading zero base-B digits), or unnormalized?
  - hidden bit (if always normalized)?
- Reserved operand representation (DEC PDP-11 and VAX: -0.0 causes fatal interrupt on load or arithmetic instructions).
- Reserved exponent fields to represent *Indefinite* (CDC, Cray), *NaN* (IEEE 754), *Infinity* (IEEE 754, CDC, Cray)
- Floating-point arithmetic is *exact*, unless rounding is required; it is *not* fuzzy! Thus, most scripting languages offer only a single numeric type (generally implemented as IEEE 754 64-bit binary, i.e., C's `double`), and use it for both integer and floating-point computations.
- Careless coding can suffer from premature underflow and overflow, and significance loss, that produces completely incorrect results over much of the floating-point range. For example, the Pythagorean theorem taught in elementary school says that in a right triangle with adjacent sides x and y, the square of the opposite side (h, the *hypotenuse*) is equal to the sum of the squares on the adjacent sides. Thus, mathematically, $h = \sqrt{x^2 + y^2}$. A naive computer implementation as `sqrt(x*x + y*y)` is found in countless programs, yet is **wrong**! Correct code in C uses a library function, `hypot(x,y)`, that does the calculation carefully, without intermediate underflow or overflow, and without introducing unnecessary rounding errors.

For the related expression, $d = \sqrt{x^2 - y^2}$, factor the argument and rewrite it as $d = \sqrt{(x-y)(x+y)}$. When x and y are close in value, one of those factors is *exact*, and the other is correct within rounding error, so there are at most three rounding errors in the result. Only when one of the factors overflows is the result completely incorrect.

Similarly, naive coding to compute the roots of $ax^2 + bx + c = 0$ using the schoolbook formula $x = (-b \pm \sqrt{b^2 - 4ac})/(2a)$ suffers from premature overflow/underflow, and also from significance loss; it may then return results with no correct digits whatever.

6 Floating-point design and exception handling

- historical systems:
  - terminate job on overflow;
  - terminate job on zero divide;
  - flush-to-zero on underflow;
  - some allow trap reporting for underflow (e.g., DEC PDP-10 and IBM System/360);
  - rounding is often done by truncation (extra trailing digits are simply discarded), but some systems attempted to round to nearest, but with only limited correctness;
  - in higher-precision formats, there may be unused (i.e., *wasted*) bits in the significand;
  - higher-precision formats may have the same exponent range as the smallest one.
- IEEE 754 floating-point arithmetic:
  - **Design goal**: Provide *uniform* and *predictable* behavior across all implementations, with these features:
    - Computation with any of four rounding modes;
    - Five basic operations (`+`, `-`, `*`, `/`, and `sqrt()`) always produce *correctly rounded* results;
    - Nonstop operation for high-performance computation;

    * Must support 32-bit and 64-bit formats, with optional 80-bit and 128-bit format extensions;
    * Exceptions set sticky flags (e.g., clear all flags, then compute, then test flags; do something different if any flags were set);
    * Implementations may choose to supply additional elementary functions, provided that they have the same rounding behavior as the five basic ones.

- Begun as IEEE working group P754 in 1976–1977.
- First draft in 1980, on which the Intel 8087 coprocessor was based, implemented, and marketed that year.
- First official standard in 1985.
- IEEE-754 revised in 2008 and 2019.
- Default on all new CPU designs since about 1990, but implementation quality varies.
- Flush-to-subnormal on underflow (default, also called gradual underflow).
- Flush-to-zero on underflow (fast, nonstandard, only choice on original Alpha CPU).
- Subnormals (formerly called denormals) may be exact, but otherwise, silently suffer precision loss.
- Nonzero / zero produces *Infinity* of correct sign.
- Zero / zero, Infinity − Infinity, Infinity / Infinity, and $\sqrt{\text{negative nonzero real}}$ produce a quiet NaN (*Not a Number*).
- NaN may be quiet (QNaN) or signaling (SNaN); quiet NaNs propagate, and signaling NaNs raise a trappable exception.
- A NaN always has a sign bit, but no meaning is attached to it, and its value has no effect on any operation that involves a NaN.
- NaNs compare unequal to everything, and their existence means that tests for negative, zero, and positive values do not cover all cases. In particular, the Fortran three-way branch conditional `IF (x) n1,n2,n3` behaves unpredictably if `x` is a NaN, and the C code

```
if (x < 0.0)
    (void)printf("x is negative nonzero\n");
else if (x == 0.0)
```

```
        (void)printf("x is zero\n");
    else
        (void)printf("x is positive nonzero\n");
```

incorrectly reports a NaN as a positive nonzero. It must be rewritten as

```
    if (x < 0.0)
        (void)printf("x is negative nonzero\n");
    else if (x == 0.0)
        (void)printf("x is zero\n");
    else if (x > 0.0)
        (void)printf("x is positive nonzero\n");
    else
        (void)printf("x is a NaN\n");
```

- Alas, Intel x86 and x86-64 CPUs, and the C# and Java programming languages and their virtual machines, have only quiet NaNs, and those two languages omit critical properties of other parts of IEEE 754 arithmetic.
- No numeric operation ever delivers a signaling NaN; any such value must have been intentionally created by the programmer (e.g., for memory initialization).
- In the binary formats, Infinity and NaN have the same exponent (the maximum possible), but Infinity has a *zero significand*, and NaN has a *nonzero significand*. One architecture-defined significand bit distinguishes between quiet and signaling NaN; the remaining bits can usefully carry a payload, such as the variable's memory address, or a distinctive pattern, such as `0xdeadfeed`, to indicate a likely uninitialized value.
- `NaN OP any` produces NaN, but which NaN results is implementation dependent: that matters when NaNs carry a payload.
- Numeric functions that receive NaN arguments should always return a NaN, preferably, the first argument found to be a NaN, so that any payload is preserved. They should also perform a small computation that generates a NaN, so that floating-point exception flags are set correctly.
- Alas, `fmax(x,y)` and `fmin(x,y)` in C99 ignore NaN arguments, unless both are NaNs. C99 `pow(0.0,x)` is required to return 1.0, even when `x` is a NaN.

- `try{} / catch {} / throw ()` blocks available in C++, C#, Java, and Lisp to deal with exceptions are too onerous to use for routine numerical work.
- Subnormals arise on underflow: with the most negative exponent, the normalization requirement is relaxed, and leading zero bits are permitted. That decreases precision, but preserves important mathematical properties, such as these:

$$\begin{aligned} x \neq y &\implies x - y \neq 0 \\ (x - y) + y \approx x & \quad \text{within rounding error of the larger of } x, y \\ (1/x) \neq 0 &\implies (1/(1/x)) \approx x \quad \text{when } x \text{ is normalized.} \end{aligned}$$

- Inexact subnormals can reduce the precision of all subsequent operations that depend on them.
- Some systems supply a library function call to switch underflow behavior at runtime between flush-to-zero and flush-to-subnormal, but that facility has never been standardized,

7 IEEE 754 binary range and precision

The design parameters of the extended IEEE 754 binary formats are presented in Table 2.

All but the 80-bit format have a hidden leading significand bit that is not stored, but is supplied in hardware for arithmetic operations.

The 256-bit format is an extension of the author's MathCW library to encourage future compiler support of that type.

8 IEEE 754 decimal range and precision

The design parameters of the extended IEEE 754 decimal formats are shown in Table 3.

A hidden leading significand digit is only possible when the base is two; it does not exist in any nonbinary system. When the decimal storage format doubles in size, the precision is always increased from n to $2n + 2$ digits, and the exponent size is increased by at least one bit. That is an

Table 2: IEEE 754 *binary* range and precision.

	single	**double**	**extended**	**quadruple**	**octuple**
Format length	32	64	80	128	256
Stored significand bits	23	52	64	112	236
Precision (t)	24	53	64	113	237
Biased-exponent bits	8	11	15	15	19
Minimum exponent	-126	-1022	$-16\,382$	$-16\,382$	$-262\,142$
Maximum exponent	127	1023	16 383	16 383	262 143
Exponent bias	127	1023	16 383	16 383	262 143
Machine epsilon	2^{-23}	2^{-52}	2^{-63}	2^{-112}	2^{-236}
(2^{-t+1})	$\approx$ 1.19e−07	$\approx$ 2.22e−16	$\approx$ 1.08e−19	$\approx$ 1.93e−34	$\approx$ 9.06e−72
Largest normal	$(1-2^{-24})2^{128}$	$(1-2^{-53})2^{1024}$	$(1-2^{-64})2^{16\,384}$	$(1-2^{-113})2^{16\,384}$	$(1-2^{-237})2^{262\,144}$
	$\approx$ 3.40e+38	$\approx$ 1.80e+308	$\approx$ 1.19e+4932	$\approx$ 1.19e+4932	$\approx$ 1.611e+78 913
Smallest normal	2^{-126}	2^{-1022}	$2^{-16\,382}$	$2^{-16\,382}$	$2^{-262\,142}$
	$\approx$ 1.18e−38	$\approx$ 2.23e−308	$\approx$ 3.36e−4932	$\approx$ 3.36e−4932	$\approx$ 2.482e−78913
Smallest subnormal	2^{-149}	2^{-1074}	$2^{-16\,445}$	$2^{-16\,494}$	$2^{-262\,378}$
	$\approx$ 1.40e−45	$\approx$ 4.94e−324	$\approx$ 3.64e−4951	$\approx$ 6.48e−4966	$\approx$ 2.25e−78 984

intentional design choice to guarantee that double-length products can be computed *exactly*, and *without underflow or overflow*, in all but the largest format.

The 256-bit format is an extension of the author's MathCW library to encourage future compiler support of that type.

9 IEEE 754 rounding modes

- In binary arithmetic, there are four rounding modes, accessible with these macro names in C99:

`FE_DOWNWARD`	: toward −Infinity;
`FE_UPWARD`	: toward +Infinity;
`FE_TOWARDZERO`	: magnitude toward zero;
`FE_TONEAREST`	: to nearest representable result, with ties to *even* (i.e., last significand bit is zero)

Use has since been found for another rounding mode that produces

Table 3: IEEE 754 *decimal* range and precision.

	single	**double**	**quadruple**	**octuple**
Format length	32	64	128	256
Stored coefficient digits	7	16	34	70
Precision (p)	7	16	34	70
Biased-exponent bits	8	10	14	22
EC bits	6	8	12	20
Minimum exponent	-95	-383	-6143	$-1\,572\,863$
Maximum exponent	96	384	6144	$1\,572\,864$
Exponent bias	101	398	6176	$1\,572\,932$
Machine epsilon (10^{-p+1})	10^{-6}	10^{-15}	10^{-33}	10^{-69}
Largest normal	$(1-10^{-7})10^{97}$	$(1-10^{-16})10^{385}$	$(1-10^{-34})10^{6145}$	$(1-10^{-70})10^{1\,572\,865}$
Smallest normal	10^{-95}	10^{-383}	10^{-6143}	$10^{-1\,572\,863}$
Smallest subnormal	10^{-101}	10^{-398}	10^{-6176}	$10^{-1\,572\,932}$

the nearest, with *ties to odd*, but no hardware implementations exist, so it must be simulated in software.

- In decimal arithmetic, there can be additional rounding modes:

`FE_HALF_UP`	: round ties upward (magnitude away from zero);
`FE_HALF_DOWN`	: round ties downward (magnitude toward zero);
`FE_HALF_AWAY`	: round ties away from zero;
others	: IBM processors with decimal floating-point hardware also have *round to prepare for shorter precision*, which has specialized application, and is not described further here.

10 Fused multiply add

IEEE 754-2008, and C99, require a fused multiply-add operation: the C function `fma(x,y,z)` computes `x * y` exactly as a double-length value, then adds `z`, producing `x * y + z` with at most *one* rounding error. CPU designs that implement the FMA generally supply variants for `-x * y + z`, `x * y - z`, and `-x * y - z`, to avoid the need for additional argument negations.

Software implementations are complicated, but hardware can do so without performance penalty, as IBM demonstrated with their POWER architecture, introduced in 1990, and shortly thereafter, with their mainframe z-Series CPUs.

The FMA operation has been found to be important for many numerical algorithms, and can be a building block for code that computes a result as the sum of an exact high part, and an accurate, but approximate, low part. Here is a short code block for recovering the rounding error in multiplication.

```
double err, x, y;
volatile double proc;

prod = x * y;
err = fma(x, y, -prod);
```

The *exact result* in binary arithmetic is `prod + err`.

Similar error recovery is also possible for the four other basic operations, but we do not show it here.

IBM POWER and z-Series processors have provided FMA in hardware for about three decades, but the availability of FMA on ARM, Intel, MIPS, and SPARC processors is extremely limited, and correct FMA is absent on DEC Alpha and HP PA-RISC CPUs.

11 Mixed-precision arithmetic

Programming languages differ in their interpretation of numeric expressions involving operands of differing precision or type, and programmers need to exercise particular care, and thoroughly understand the rules of their chosen language.

For example, what is the value of $1/7$? In many languages, its value is 0, because integer division is implied, and such division truncates toward zero. However, in most scripting languages that offer only a single numeric type implemented as a 64-bit floating-point value, its value is approximately $0.142\,857\,142\,857\,142\,857\ldots$.

As another example, how should the mixed-precision assignment `x = 1.0F / 7.0Q` be evaluated? Many languages require promotion of the shorter operands to the longest type in the expression, the arithmetic carried out, and the result then converted to the precision of the target assignment. What should then happen for subexpressions, such as `x = (1.0F / 7.0) + 1.0Q`? Also, can the computation of constant expressions be done at compile time (possibly to much higher precision, as the `gcc` compiler does), or must it be done at run time? And which rounding mode is used? Language practice varies.

12 Base conversion problem

Data conversion between human decimal values, and computer binary values, is a frequent source of confusion for humans.

Because $10 = 5 \times 2$, every finite *binary* floating-point value can be represented exactly as a decimal value. Thus, $2^{-3} = 0.125$, and $2^{-16} = 0.0000152587890625$, both exact.

The reverse is *not* true: the expression $1/10$ is exactly representable as 0.1 in decimal arithmetic, but its value in binary arithmetic has a nonterminating fractional part: `+0x1.999999...p-4`. Its stored value must always be rounded, and that value converted back to decimal differs by a small amount from 0.1.

The question of how many bits, b, does it take to represent a d-digit decimal value, and the reverse, seems simple. The largest unsigned decimal value is then $10^d - 1$, and its logarithm to the base two, rounded up to the nearest integer, should be the value of b. That is, we have

$$\begin{aligned} \log_2(10^d - 1) &< \log_2(10^d) \\ &< d\log_2(10), \\ d\log_2(10) &\le \lceil d\log_2(10)\rceil, \\ b \text{ (bits needed)} &\le \lceil d\log_2(10)\rceil. \end{aligned}$$

In C code, that can be written as a convenient function like this:

```
#include <math.h>

int
needed_dec_to_bin_digits(int d_dec)
{
    return ((int)(ceil((double)d_dec * log2(10.0))));
}
```

The reverse operation seems equally simple:

Table 4: Goldberg and Matula digit counts for base conversion.

Digit counts for base conversion					
storage size	**32**	**64**	**80**	**128**	**256**
binary in	24	53	64	113	237
decimal out	9	17	21	36	73
decimal in	7	16	n/a	34	70
binary out	25	55	n/a	114	234

```
int
needed_bin_to_dec_digits(int b_bits)
{
    return ((int)(ceil((double)b_bits * log10(2.0))));
}
```

Unfortunately, as I. Bennett Goldberg and David Matula showed in 1967–1969, those formulas are sometimes wrong. The correct versions are

```
#include <math.h>

int
correct_dec_to_bin_digits(int d_dec)
{   /* Goldberg formula */
    return ((int)ceil((double)d_dec * log2(10.0) + 1.0));
}

int
correct_bin_to_dec_digits(int b_bits)
{   /* Matula formula */
    return ((int)ceil((double)b_bits * log2(10.0) + 1.0));
}
```

Table 4 shows the results of applying the Goldberg and Matula formulas to IEEE 754 formats.

For example, values in the 32-bit binary format should be output with 9 decimal digits, yet the default in Fortran, C, and many other languages is just 6, which is far from adequate for exact recovery of a value from a prior computation.

In 1977, for his TeX typesetting system, Donald Knuth wrote code to convert fixed-point binary values to as many decimal digits as are needed to recover the binary value exactly in the reverse conversion. He was confident that his short program was correct, and it could checked in a few minutes of computer time by converting all possible $2^{(n-2)}-1$ inputs (for the common case of $n = 32$, that is $1\,073\,741\,823$ values), and then converting the decimal results back to binary for comparison with the original input values. However, he was bothered that he could not prove its correctness at the time, and several years went by before he found the proof. His classic paper on that proof has a wonderful title: *A simple program whose proof isn't*. It was published in a 1990 commemorative book volume that contains an independent proof by David Gries.

The base-conversion problem is one reason why we could reasonably argue that software would be improved for humans if it switched from binary to decimal floating-point arithmetic, and the author's MathCW library makes that possible on many systems.

The problem of producing correctly rounded results for base conversion of floating-point numbers is surprisingly difficult, with solutions found only in the 1990s. In hard cases, the conversion algorithms require computation with thousands of digits, even for the modest precision of the IEEE 754 formats. Consequently, few programming languages today can guarantee always-correct conversions, and that is another reason why a switch to computing in decimal floating-point arithmetic would be beneficial.

13 IEEE 754 exception flags

C99 defines standard names for five exception flags required in IEEE 754 arithmetic:

```
FE_DIVBYZERO   FE_INEXACT   FE_INVALID   FE_OVERFLOW   FE_UNDERFLOW
```

Each must be a distinct integer with only a single nonzero bit.

C99 also defines the symbol `FE_ALL_EXCEPT` as the bitwise-OR of all of the above exception flags, plus any additional ones defined by the implementation.

The precise rules for when those flags are set are complicated, but common cases are obvious.

The FE_INEXACT flag is set whenever a computed result must be rounded to a slightly different representable value, according to the current rounding mode. It is likely to be set in almost any complicated numerical expression, but for code that implements exact integer addition and multiplication via floating-point arithmetic, that flag's value should always be zero.

The FE_INVALID flag is set when the result is not determinable, and would normally then produce a quiet NaN. Examples are operands or operations like these:

- SNaN,
- Infinity − Infinity,
- Infinity / Infinity,
- Zero / Zero,
- Infinity * Zero,
- invalid compare,
- invalid square root,
- invalid integer convert.

Some systems provide additional exception flags, and some designs permit certain exceptions to be *masked*, so their flags cannot be set, but there are no standard interfaces for those features in the C language.

In C99, you can test and set the current flags via functions with these prototypes:

```
#include <fenv.h>

int feclearexcept   (int excepts);
int feraiseexcept   (int excepts);
int fetestexcept    (int excepts);

int fegetexceptflag (fexcept_t *flagp, int excepts);
int fesetexceptflag (const fexcept_t *flagp, int excepts);

int fegetenv        (fenv_t *envp);
int feholdexcept    (fenv_t *envp);
int fesetenv        (const fenv_t *envp);
int feupdateenv     (const fenv_t *envp);
```

Consult their manual pages, or the ISO C Standards, for details.

14 Rounding control

C99 provides these functions and macros for IEEE 754 rounding control:

```
#include <fenv.h>

int fegetround (void);   /* return rounding_direction */
int fesetround (int rounding_direction);

/* rounding_direction is one of these: */
FE_DOWNWARD    round toward -Inf
FE_TONEAREST   round to nearest, ties to even (default)
FE_TOWARDZERO  round toward 0
FE_UPWARD      round toward +Inf
```

Rounding errors are normally of minor importance in most numerical code, because they usually do not accumulate much, especially when the default mode of `FE_TONEAREST` is in effect. It is, however, possible to contrive cases where they *do* grow substantially, as shown in the companion chapter, *Computer Arithmetic: Perils, Pitfalls, and Practices*.

Rounding control is needed for implementing *interval arithmetic*, which represents numbers as pairs [*lower-bound*, *upper-bound*]. Interval arithmetic, when carefully used, can provide rigorous bounds for computed results, provided that the algorithm and code are correct, and the bounds are not so wide as to make the computed value useless. Interval arithmetic is supported in the Sun/Oracle C, C++, and Fortran compilers, which are freely available for Solaris and GNU/Linux operating systems. It is also available in the Matlab toolbox `intlab`, and in the Python modules `pyinterval` and `mpmath` for standard and multiple-precision interval arithmetic.

15 Significance loss

By far the most common cause of numerical difficulties in computations is significance loss from subtraction of two nearly equal numbers of the same sign. Leading digits disappear, and the result has lower precision that contaminates all future computation that depends on that value.

Here is an example from the author's book, with calculations done via upward and downward recurrences for ordinary Bessel functions of the first kind, $J_n(x)$. They look like decaying cosine (n even) and sine (n odd) waves, except that the zeros are not equally spaced.

Starting from correctly rounded seven-digit 32-bit decimal floating-point values of $J_0(1)$ and $J_1(1)$, we get these values for the n-th Bessel function, $J_n(1)$:

```
--------------------------------------------------------
 n    computed Jn(1)
--------------------------------------------------------
 0    7.651977e-01            # expect  0   7.651977e-01
 1    4.400506e-01            # expect  1   4.400506e-01
 2    1.149035e-01            # expect  2   1.149035e-01
 3    1.956340e-02            # expect  3   1.956335e-02
 4    2.476900e-03            # expect  4   2.476639e-03
 5    2.518000e-04            # expect  5   2.497577e-04
 6    4.110000e-05            # expect  6   2.093834e-05
 7    2.414000e-04            # expect  7   1.502326e-06
 8    3.338500e-03            # expect  8   9.422344e-08
 9    5.317460e-02            # expect  9   5.249250e-09
...
15    7.259898e+06            # expect 15   2.297532e-17
16    2.175373e+08            # expect 16   7.186397e-19
17    6.953934e+09            # expect 17   2.115376e-20
18    2.362163e+11            # expect 18   5.880345e-22
19    8.496833e+12            # expect 19   1.548478e-23
20    3.226435e+14            # expect 20   3.873503e-25
--------------------------------------------------------
```

Notice that already at $n = 5$, there is only one correct leading digit, and after that, the numbers rapidly become nonsensical.

A stable computation is done by downward recurrence, where we assume that $J_n(1) = 0$ for all n beyond some reasonable value:

```
--------------------------------------------------------
 n    computed Jn(1)
--------------------------------------------------------
20    3.873503e-25            # expect 20   3.873503e-25
```

```
19    1.548478e-23           # expect 19   1.548478e-23
18    5.880342e-22           # expect 18   5.880345e-22
17    2.115375e-20           # expect 17   2.115376e-20
16    7.186395e-19           # expect 16   7.186397e-19
15    2.297531e-17           # expect 15   2.297532e-17
...
 9    5.249246e-09           # expect  9   5.249250e-09
 8    9.422337e-08           # expect  8   9.422344e-08
 7    1.502325e-06           # expect  7   1.502326e-06
 6    2.093833e-05           # expect  6   2.093834e-05
 5    2.497577e-04           # expect  5   2.497577e-04
 4    2.476639e-03           # expect  4   2.476639e-03
 3    1.956335e-02           # expect  3   1.956335e-02
 2    1.149035e-01           # expect  2   1.149035e-01
 1    4.400506e-01           # expect  1   4.400506e-01
 0    7.651977e-01           # expect  0   7.651977e-01
--------------------------------------------------------
```

All values are now correct to at most 4 units in the last place, and the last six values are exactly correct.

Because series expansions are commonly used for function computation, one must be careful to avoid those in which terms of both signs occur. For example, consider computation of the tangent and cotangent for small arguments. We can find their power series (with complicated formulas for the general coefficients) like this:

```
% maple
    |\^/|     Maple 2019 (X86 64 LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, ...
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.

> convert(series(cot(x), x = 0, 10), polynom);
                                  3      5     7       9
                                 x    2 x     x     2 x
                   1/x - x/3 - ---- - ---- - ---- - -----
                                 45    945   4725   93555

> convert(series(tan(x), x = 0, 10), polynom);
                             3         5   17   7    62   9
                  x + 1/3 x  + 2/15 x  + --- x  + ---- x
                                         315      2835
```

Notice that terms of the series for `tan(x)` always have the same sign, so there is never significance loss in their summation, which would normally be done by summing upward from the *second* term, then adding that sum to the *exact first term* to minimize accumulation of rounding errors.

However, the leading two terms in the series for `cot(x)` always alternate in sign, so there is massive loss of leading digits when $x \approx \sqrt{3} \approx 1.732$, as can be seen from these Maple computations done in its default precision of 10 decimal digits:

```
> evalf(sqrt(3));
                                  1.732050808

> x := 1.732:

> evalf(1/x);
                                  0.5773672055

> evalf(x/3);
                                  0.5773333333
> evalf(1/x - x/3);
                                  0.0000338722

> x := 1.73205:

> evalf(1/x);
                                  0.5773505384

> evalf(x/3);
                                  0.5773500000

> evalf(1/x - x/3);
                                           -6
                                  0.5384 10
```

The five basic operations (`+`, `-`, `*`, `/`, and `sqrt()`) in complex arithmetic are *all* subject to premature underflow and overflow, and to significance loss (the latter even more so if the products of two reals are not computed to twice working precision).

The elementary functions in complex arithmetic can be computed from exponential, log, hyperbolic, and trigonometric functions of real arguments. If the trigonometric ones lack *exact* argument reduction to a multiple of π or $\pi/2$, plus a small remainder, then all of the complex functions that are computed from trigonometric functions are *catastrophically wrong* for arguments of rather modest size. The complex exponential and log functions are in that class, and many other functions in turn depend on those two.

Complex arithmetic is used much less often than real arithmetic, and implementations of the former are often seriously deficient. The best advice seems to be to either reformulate complex-number expressions into exact mathematical equivalents in real arithmetic, or to use the highest available complex floating-point precision.

16 Precision control

The Intel 8087 floating-point coprocessor, and subsequent implementations of the x86 architecture and its descendants by Intel, AMD, Cyrix, and others, allows control of the precision used in numeric operations.

On those systems, values are normally stored in programs as 32-bit `float`, 64-bit `double`, and 80-bit `long double`. However, when such values are loaded into the floating-point registers, they are all expanded exactly to the 80-bit format, and arithmetic is carried out in registers in that format, until finally a value is stored to memory, at which point, it is converted according to the current rounding mode to the type of the target location.

The 11 extra significand bits, and 4 extra exponent bits, of the 80-bit format, compared to the 64-bit format, are often beneficial in reducing rounding errors and significance loss, and avoiding premature underflow and overflow.

However, the higher intermediate precision increases the number of roundings, and that is sometimes harmful.

These functions may be available in some implementations:

```
#include <fenv.h>

int fegetprec (void);
int fesetprec (int);
```

The arguments and return values are one of these:

```
FE_DBLPREC   FE_FLTPREC   FE_LDBLPREC
```

A negative function return means failure.

For example, setting the precision to FE_FLTPREC forces register operations to produce only 32-bit format results.

17 Writing precision-independent code

In C, and many other languages, numeric types, library function names, input and output format descriptors, and suffixes on constants all refer to a particular floating-point format.

However, in numeric software, it is often a good idea to design algorithms to be precision independent, so how do we achieve that if our code is riddled with markup indicating the precision?

The C preprocessor provides a way to hide precision details, by wrapping all floating-point constants in macros, hiding library function names behind generic names, and selecting different definitions by conditional tests. Here is a short example that makes it easy to select between any of three precisions, with one of them a default unless overridden at compile time. Define a header file, fp.h, with these contents:

```
#if !defined(FP_H)
#define FP_H

#if !defined(FP_T_FLOAT) && !defined(FP_T_DOUBLE) \
 && !defined(FP_T_LONG_DOUBLE)
#define FP_T_DOUBLE
#endif

#if defined(FP_T_FLOAT)

typedef float           fp_t;
#define FMT_G           "%.9g"
#define FP(x)           x ## F
#define COS(x)          cosf(x)
#define SQRT(x)         sqrtf(x)
#define TAN(x)          tanf(x)

#elif defined(FP_T_DOUBLE)
```

```
typedef double          fp_t;
#define FMT_G           "%.17g"
#define FP(x)           x
#define COS(x)          cos(x)
#define SQRT(x)         sqrt(x)
#define TAN(x)          tan(x)

#elif defined(FP_T_LONG_DOUBLE)

typedef long double     fp_t;
#define FMT_G           "%.36Lg" /* %.21Lg if 80-bit */
#define FP(x)           x ## L
#define COS(x)          cosl(x)
#define SQRT(x)         sqrtl(x)
#define TAN(x)          tanl(x)

#else

#error "No FP_T_xxx macro defined"

#endif

static const fp_t PI      =
       FP(3.141592653589793238462643383279502884);
static const fp_t PI_HALF =
       FP(1.570796326794896619231321691639751442);

#endif /* !defined(FP_H) */
```

The outer conditional makes the file safe against multiple inclusions: it would otherwise be an error to repeat the `typedef` and symbolic constant definitions.

Now we can write a short precision-independent program, saved in a file `fptest.c`:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "fp.h"
```

```
int
main(void)
{
    fp_t x, y;

    x = FP(0.75) * PI;
    y = COS(x);
    (void)printf("cos((3/4)pi) = " FMT_G "\n", y);

    return (EXIT_SUCCESS);
}
```

Here are four test runs to show how it works:

```
% cc fptest.c -lm && ./a.out
cos((3/4)pi) = -0.70710678118654746

% cc -DFP_T_FLOAT fptest.c -lm && ./a.out
cos((3/4)pi) = -0.707106769

% cc -DFP_T_DOUBLE fptest.c -lm && ./a.out
cos((3/4)pi) = -0.70710678118654746

% cc -DFP_T_LONG_DOUBLE fptest.c -lm && ./a.out
cos((3/4)pi) = -0.707106781186547524400844362104848992
```

The author's MathCW library uses techniques like that to support ten different floating-point formats for both binary and decimal arithmetic, with common code for all of them, and only rare excursions into base-specific, or precision-specific, code.

Care must, of course, be taken to ensure that any constants wrapped by `FP()` are either exactly representable in all bases and precisions, as our sample of `0.75` is, or else specified with sufficient digits to be correct for all of them.

18 Important algorithms in arithmetic

In this section, we treat some important general algorithms that find wide application to floating-point computation.

18.1 Newton–Raphson iteration

One of the most important procedures for root finding is the *Newton–Raphson* iteration, first published in 1690. To find the root x of the equation $f(x) = 0$, start with a good initial guess, $x = x_0$, and then compute successive improvements via

$$x_{n+1} = x_n - f(x_n)/f'(x_n), \qquad n = 0, 1, 2, 3, \ldots.$$

where $f'(x)$ is the first derivative of $f(x)$ with respect to x.

With sufficiently close initial values, convergence is *quadratic*, doubling the number of correct digits in each iteration.

An important feature is that the entire computational error arises in the *single expression* of the right-hand side: each new result is an improvement on the previous one, so errors from earlier iterations do not propagate to the final result.

Some CPU floating-point designs lack divide and square root instructions. Instead, they provide instructions to compute an approximation to the result, usually by short polynomial fits that can be computed quickly in a single hardware instruction. Compilers are then expected to interleave code for successive *Newton–Raphson* iterations with following code. Because the precision of the initial approximations is known, the quadratic convergence means that the required number of iterations is a small constant, typically two to five.

18.2 Arithmetic–geometric mean (AGM)

AGM iteration is treated in the author's *The Mathematical-Function Computation Handbook*. With care in its implementation, the convergence is also quadratic, but unlike Newton–Raphson iterations, errors accumulate over all of the AGM iterations.

The AGM iteration is not widely known, nor is it treated in most textbooks, yet it can be used in the computation of several important functions, and some mathematical constants.

In some cases, variants of the AGM are known that have higher-order convergence. There is a nonic-order formula for π that increases the number of correct digits by a factor of nine in each iteration! With 2 correct initial digits, we get results to 18, 162, 1 458, 13 122, 118 098 ... digits. Such formulas have been used to compute over a trillion (10^{12}) digits of π!

18.3 Continued fractions

Continued fractions take the form

$$f(x) = \cfrac{a}{b + \cfrac{c}{d + \cfrac{e}{f + \cfrac{g}{h + \cdots}}}}.$$

Here, a is a constant, and one of the two terms in each subsequent denominator depends on x.

They are known for many standard elementary and special functions, and some have the important property that their radius of convergence (the range of usable x values) is often much larger than for series expansions.

The repeated divisions, and the computational observation that the evaluation is often subject to premature underflow or overflow, have long discouraged their use by programmers, and few textbooks on numerical computation even mention them.

However, two good algorithms for their evaluation have been recently found that are stable, easy to program, and require only *one* division. They are discussed further in *The Mathematical-Function Computation Handbook*.

Even if continued fractions are not the final choice in the implementation of important elementary and special functions, they provide *completely independent* routes to function computation that are valuable for assessing accuracy and code correctness.

18.4 High-precision constants

For accurate computation, it is sometimes desirable to represent constants as sums of an exact high part, and an accurate, but approximate, low part. Computer algebra systems can help to generate such values.

Here is an example suitable for the IEEE 754 64-bit binary and decimal formats:

```
#define FP(x) x
typedef double fp_t;
static const fp_t PI_HI = FP(7074237752028440.0) /
                          FP(2251799813685248.0);
static const fp_t PI_LO = FP(1.2246467991473532e-16);

#define FP(x) x ## DD
typedef decimal_double fp_t;
static const fp_t PI_HI = FP(3141592653589793.0) /
                          FP(1.e+15);
static const fp_t PI_LO = FP(2.384626433832795e-16);
```

The `FP()` wrappers are convenient for writing code that is generalized to multiple precisions and bases. The high part is defined as a rational number with an exact integer numerator, and a denominator that is a power of the base. Modern compilers evaluate that value at compile time, producing an exact result.

To compute πx to an almost certainly correctly rounded value, we can then write code like this:

```
#include <math.h>

double pi_x, x;

pi_x = fma(x, PI_HI, x * PI_LO);
```

18.5 Accurate series summations

Computer programs often implement series summations, and they commonly do so by accumulating a sum starting from the first term. Thus, to compute $y = \sum_{k=0}^{n} c_k x^k$, a beginning programmer might code it naively like this:

```
double
brute(int n, double c[n + 1], double x)
{   /* return sum_{k = 0}^n c[k] * x-to-the-k */
    double sum;
    int k;

    sum = 0.0;
```

```
        for (k = 0; k <= n; ++k)
            sum += c[k] * pow(x, (double)k);

        return (sum);
    }
```

Such code has two serious problems: it suffers from excessive rounding error, and it has an expensive power operation in every iteration.

The second problem is easily removed by computing powers of `x` in the summation loop, and avoiding the first of them:

```
    double c[N + 1], x, x_to_k, y;

    sum = c[0];
    x_to_k = x;

    for (k = 1; k <= N; ++k)
    {
        sum += c[k] * x_to_k;
        x_to_k *= x;
    }
```

Reducing rounding errors requires a bit of thought. The trick is to observe that such sums generally involve terms of decreasing magnitude. Thus, most of the rounding error happens in the adjustment to the first term, so it is best to leave that term to last:

```
    double c[N + 1], x, x_to_k, y;

    sum = 0.0;
    x_to_k = x;

    for (k = 1; k <= N; ++k)
    {
        sum += c[k] * x_to_k;
        x_to_k *= x;
    }

    sum += c[0];
```

If the leading coefficient is not exactly representable, rewrite it as a two-part sum as in the preceding section, and replace the last assignment by

```
sum += c0_lo;
sum += c0_hi;
```

In the case $n = \infty$, the sum can be terminated as soon as the new term is sufficiently small, with loop code like this:

```
term = c[k] * x_to_k;
old_sum = sum;
sum += term;

if (old_sum == sum)
    break;  /* done: sum is numerically converged */
```

Notice that there is no magic machine-dependent constant embedded in the code against which we judge *smallness*. Notice also that in a sum that could in principle terminate at the first term, we intentionally add the second term. That is important, because it allows the sum to depend on the current rounding mode.

An alternative approach is to evaluate a sum via the *Horner representation* of a polynomial, $p_n(x)$, of order n, which has these leading cases:

$$\begin{aligned}
p_0(x) &= c_0, \\
p_1(x) &= c_0 + c_1 x, \\
p_2(x) &= c_0 + c_1 x + c_2 x^2, \\
&= c_0 + (c_1 + c_2 x)x, \\
p_3(x) &= c_0 + c_1 x + c_2 x^2 + c_3 x^3, \\
&= c_0 + (c_1 + (c_2 + c_3 x)x)x, \\
p_4(x) &= c_0 + (c_1 + (c_2 + (c_3 + c_4 x)x)x)x, \\
p_5(x) &= c_0 + (c_1 + (c_2 + (c_3 + (c_4 + c_5 x)x)x)x)x, \\
p_6(x) &= c_0 + (c_1 + (c_2 + (c_3 + (c_4 + (c_5 + c_6 x)x)x)x)x)x, \\
p_7(x) &= \ldots.
\end{aligned}$$

The general pattern is clear: we start at the deepest nesting level with the highest-order coefficient, multiply by x, add to the next lower coefficient, and repeat:

```
double
horner(int n, double c[n + 1], double x)
{
    double sum;

    if (n < 1)  /* sanity check */
        n = 0;

    sum = c[n];

    for (k = n; k > 0; --k)
        sum = sum * x + c[k - 1];

    return (sum);
}
```

Notice that the loop operation could be an FMA, so we can reduce rounding error by replacing the loop body by

```
        sum = fma(sum, x, c[k - 1]);
```

In the common case where the polynomial order is known in advance, the entire Horner evaluation can be expanded inline to n consecutive FMA operations, and if the compiler recognizes that `sum` and `x` should be held in registers, then we have perfect code, with one FMA and one memory access per iteration.

The Horner method sums terms in reverse order, so the rounding errors accumulate starting from the highest power. As long as the series terms decrease in magnitude with increasing order, the major rounding error is likely in the final sum, where we add `c[0]` to the previous sum. That means that the polynomial is effectively evaluated with only one rounding error, and the FMA should then produce an almost always correctly rounded result.

If the leading coefficient is not exactly representable, we can instead store `c0_lo` in `c[0]`, then compute the polynomial as `c0_hi + horner(x, c, n)`.

Further improvements are possible by computing an estimate of the rounding error in each iteration, keeping a separate running sum of those errors, and adding that second sum to the final one. We leave the details for textbook treatments; search online sources for the keywords *accurate summation*, *compensated summation*, and *twosum*.

One further improvement that we might make in our code is to store the coefficients in reverse order, because that optimizes cache access. We would just have to replace references to `c[j]` by `c[n - j]` in our Horner function, and in the initialization of `c[]`.

We could then borrow an idea from the Sun Solaris `math.h` header file, and provide these definitions that allow efficient inline expansion when the order is a compile-time constant, and are trivially generalized to higher orders:

```
/* POLYn(x,c) = sum_{k = 0}^n c[n - k] * x-to-the-k */
#define POLY1(x, c)    (fma((c)[1], (x), (c)[0])
#define POLY2(x, c)    (POLY1((x), (c)) * (x) + (c)[2])
#define POLY3(x, c)    (POLY2((x), (c)) * (x) + (c)[3])
#define POLY4(x, c)    (POLY3((x), (c)) * (x) + (c)[4])
#define POLY5(x, c)    (POLY4((x), (c)) * (x) + (c)[5])
#define POLY6(x, c)    (POLY5((x), (c)) * (x) + (c)[6])
#define POLY7(x, c)    (POLY6((x), (c)) * (x) + (c)[7])
#define POLY8(x, c)    (POLY7((x), (c)) * (x) + (c)[8])
#define POLY9(x, c)    (POLY8((x), (c)) * (x) + (c)[9])
```

Alternatively, in our original version, we could prime the cache by referencing the first element of the coefficient array with two statements at the beginning of the main code:

```
volatile double cache_c0;

cache_c0 = c[0];
```

18.6 Polynomial fits

One of the important tools for implementing complicated functions is to replace them with polynomial approximations, possibly with different fits in each of several argument ranges.

In general, computing such approximations requires higher working precision, and code for making the fits is available in some computer algebra systems, where there may be a choice of ordinary polynomials, rational polynomials, Chebyshev expansions, and so on. Of course, it must also be possible to compute the original function to arbitrary precision.

Many of the elementary and special functions are handled that way in practice, often using the Horner representation. Suitable encapsulation of the polynomial evaluation in macros can hide their degree, allowing the same code to be used for multiple precisions and bases.

Consult the textbooks listed in the companion chapter for details.

18.7 Detecting special values

Because NaNs compare unequal to everything, *including themselves*, code similar to this

```
if (x != x)
    (void)printf("x is a NaN\n");
```

should work in *every* programming language, as the designers of IEEE 754 arithmetic intended. Sadly, some compilers contain incorrect optimizations that remove that statement entirely. The recommended safe procedure is to use a standard library function:

```
if (isnan(x))
    (void)printf("x is a NaN\n");
```

You cannot use floating-point operations to determine whether two NaNs have the same sign, type, and payload: instead, use a memory byte comparison function, like this:

```
#include <string.h>

double x, y;

if ( isnan(x) && isnan(y) &&
     (memcmp(&x, &y, sizeof(x)) == 0) )
    (void)printf("x, y are bit-for-bit identical NaNs\n");
```

Testing for Infinity should always be possible with code like this:

```
if (abs(x) >= 1.0 / 0.0)
    (void)printf("x is Infinity\n");
```

but once again, there are misbehaving compilers that reject our sample. The safe way looks like this:

```
if (isinf(x))
    (void)printf("x is Infinity\n");
```

Signed zeros are another area of mishandling by some compilers, and because computer arithmetic requires that $+0$ and -0 compare equal, you cannot compare a variable with `-0.0` to check for a negative zero. Instead, use the sign-transfer function

```
if ( (x == 0.0) && (copysign(1.0, x) == -1.0) )
    (void)printf("x is -0.0\n");
```

or the sign-test function

```
if ( (x == 0.0) && signbit(x) )
    (void)printf("x is -0.0\n");
```

Assignments of negative zero constants are botched on some systems, so to get one, you can use code like this:

```
double neg_zero;

neg_zero = copysign(0.0, -1.0);
```

To test for a subnormal, compare against a standard system constant for the smallest *normalized* number:

```
#include <float.h>
#include <math.h>
#include <stdio.h>

double x;

if ( (x != 0.0) && (fabs(x) < DBL_MIN) )
    (void)printf("x is subnormal\n");
```

A standard function allows a direct test for normal number:

```
#include <float.h>
#include <math.h>
#include <stdio.h>

double x;

if (isnormal(x))
    (void)printf("x is normal "
                 "(not zero, subnormal, Inf, or NaN\n");
```

To extract the exponent, use code like this:

```
#include <math.h>

double frac, sig, x, y;
int nf, ns;

frac = frexp(x, &nf);  /* x = frac * 2-to-the-nf */
                       /* frac = 0 or in +/-[1/2, 1) */
y = ldexp(frac, nf);   /* now y == x */

ns = (int)logb(x);     /* x = significand * 2-to-the-ns */
sig = scalbn(x, -ns);  /* sig = 0 or in +/-[1, 2) */
frac = scalbn(x, -ns - 1); /* frac = 0 or in +/-[1/2, 1)*/
```

To reconstruct the original values, use either of these:

```
x = ldexp(frac, nf);   /* nf = ns - 1; frac = sig/2 */

x = scalbn(sig, ns);   /* ns = nf - 1; sig = 2*frac */
```

To get the sign bit, use

```
int s;

s = signbit(x); /* s = 1 (negative) or 0 (nonnegative) */
```

19 Applications of signed zero

The sign of an IEEE 754 zero value can record its provenance as an underflowed computation from the left, or the right, of zero on the real line, and that may occasionally matter.

In complex arithmetic, many functions have the property of having *branch cuts*, where the surface corresponding to $z = f(x + yi)$ has tears. Correct handling of computations near branch cuts may depend critically on the availability of signed zeros. See W. Kahan's famous 1986 document *Branch Cuts for Complex Elementary Functions or Much Ado About Nothing's Sign Bit*.

20 Floating-point loop increments

One of the common loop constructs in C looks like this:

`for (`*`init`* `;` *`test`* `;` *`step`* `) {` *`statements`* `}`

Most commonly, the parenthesized three clauses contain integer expressions, like this:

```
for (k = 0 ; k <= n ; ++k)
```

In such a case, the iteration count is predictable: here, it is `n + 1`, because `k` takes the values `0, 1, 2, ..., n`.

Novice programmers might write a `for` statement like this:

```
float x;
int n;

for (n = 0, x = 0.0F ; x <= 1.0F ; x += 0.1F)
{
    n++;
    ...
}
```

What then is the iteration count, `n`?

In decimal floating-point arithmetic, the answer is clear: it is **11**, because `x` takes the exactly representable values `0.0, 0.1, 0.2, ..., 1.0`.

However, in binary floating-point arithmetic, `0.1` is not exactly representable, and depending on the precision, its value may be slightly less than, or slight greater than, a tenth.

Test programs with that loop ran 10 iterations in 32-bit arithmetic, but 11 in 64-, 80-, and 128-bit arithmetic.

Had the loop increment been `0.0625` or `0.125`, both of which are exactly representable in all sizes of IEEE 754 binary and decimal arithmetic, the iteration counts would have been independent of base, precision, and machine, and thus, the code would be portable.

If you really need to ensure that a loop with floating-point increments gets within rounding error of its upper limit, or exactly there, switch to integer loop control, and compute the former loop variable in the statement body, with code like this:

```
float dx, x;
int n;
static const int N = 11;
static const float x_beg = 0.0F;
static const float x_end = 1.0F;

dx = (x_end - x_beg) / (float)(N - 1);

for (n = 0; n < N; ++n)
{
    x = x_beg + (float)n * dx;

    /* optional, to force exact end point */
    if (n == (N - 1))
        x = x_end;

    ...
}
```

21 Half- and quarter-precision floating-point

In some applications, notably in signal processing, input data may have precisions of only a few bits, and one can then ask whether there might be use for shorter floating-point formats of sizes 8 and 16 bits, instead of the IEEE 754 32-bit format. The smaller sizes could get two to four times as much data into cache memory, reduce filesystem and network traffic by similar factors, and computations might be able to run somewhat faster.

For about three decades after the original design of IEEE 754 arithmetic, no major vendor implemented those small formats. However, that has recently changed, and some graphics processing units (GPUs), now provide them, and because the GPUs provide hundreds to thousands of specialized arithmetic cores capable of parallel computation, there is an incentive to use them.

Such enthusiasm may, however, be temporary, because the history of floating-point arithmetic has repeatedly shown that many problems can be solved to machine precision if several additional digits are available for intermediate computation, and that situations inevitably turn up where the default working precision is insufficient for the task.

22 Controlling evaluation order

Modern optimizing compilers may do substantial rearrangements of instructions to improve performance, such as moving a load of a value far before its first use, to try to overlap memory and cache delay with execution of other independent instructions. Such optimizations are mostly beneficial, but sometimes you need to ensure that steps are done in a particular order, or that a value used is that from memory, rather than from a register of higher precision and range.

Two tools are available in C for that purpose: the comma operator, and the `volatile` type qualifier.

The comma operator is used in a parenthesized list of assignments, or expressions, to be processed from left to right, and each list member must be evaluated completely before the next one is started. Thus, `z = (w = x, x = y, y = 3)`, finally assigns to `z` the value 3, and ensures that `w` and `x` get the original values of `x` and `y`, respectively.

The `volatile` modifier in type declaration tells the compiler that each use of the value of such a variable must get it from memory, not from a register, and thus, any previous value held in a register must already have been stored. This is commonly useful for controlling the precision of numeric variables, and avoiding the effects of higher intermediate precision.

Here is a test program, `ufl.c`, that combines both techniques, and exhibits surprises when run:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int
main(void)
{
    int n;
    double w, x;
    volatile double y, z;

    n = 0;
    x = 1.0;
```

```
        while ((w = x / 2.0, w) > 0.0)
        {
            x = w;
            ++n;
        }

        (void)printf("nonvolatile smallest value = %a"
                     " in %d steps\n", x, n);

        n = 0;
        y = 1.0;

        while ((z = y / 2.0, z) > 0.0)
        {
            y = z;
            ++n;
        }
        (void)printf("   volatile smallest value = %a"
                     " in %d steps\n", y, n);

        return (EXIT_SUCCESS);
    }
```

The code is expected to compute, and report, the smallest positive nonzero value, and the loop count, the negative of the power of two.

On machines with only IEEE 754 binary 32-bit and 64-bit arithmetic, the test run looks like this:

```
% cc ufl.c && ./a.out
nonvolatile smallest value = 0x1.0000000000000p-1074 in \
                             1074 steps
   volatile smallest value = 0x1.0000000000000p-1074 in \
                             1074 steps
```

The results were unchanged in other tests with six different compilers at various optimization levels. The reported values are the smallest *subnormals*.

Next, I ran it on an old machine with a MIPS processor where the default is flush-to-zero underflow, and replaced the unrecognized C99 %a format item with the older %g item. The run looks like this:

```
nonvolatile smallest value = 2.22507e-308 in 1022 steps
```

```
   volatile smallest value = 2.22507e-308 in 1022 steps
```

The reported values are the smallest *normals*.

However, on an Intel x86 machine, where all the floating-point registers have the 80-bit format, the output is:

```
% cc -g ufl.c && ./a.out
nonvolatile smallest value = 0x1p-1074 in 1074 steps
   volatile smallest value = 0x1p-1074 in 1074 steps

% cc -O1 ufl.c && ./a.out
nonvolatile smallest value = 0x0p+0 in 16434 steps
   volatile smallest value = 0x1p-1074 in 1074 steps
```

What happened on the x86 system?

- In the first run, the `-g` option requests debug-level code generation with no optimization, and the results agree with the previous runs.
- In the second run, the level-1 optimization results in `w` being held in a higher-precision register, and the register value, *not* the memory value, is used in the test. At the end of the loop, the register value for `y` is finally stored, and because it is too small to represent, it underflows to zero. The same behavior is observed with higher optimization levels, and with other compilers.
- In the second loop in both runs, `z` is computed, stored, and reloaded, so its value on every iteration is that stored in memory in a 64-bit format. The loop then terminates at the expected iteration with the last value assigned to `y`.

The x86-64 architecture was designed as an upward compatible extension of the x86 architecture. The latter uses the original 1980 Intel 8087 coprocessor design with eight 80-bit floating-point registers that are not directly accessible, but instead are a stack that must be carefully managed by the compiler. Binary operations act on the top two stack elements, pop them off, and push the result back as the new top element. The x86-64 design keeps those, but also adds 32 directly addressable 64-bit floating-point registers. Some CPU models also have additional sets of vector floating-point registers. Compilers on x86-64 systems are free to use all, or subsets, of those register sets. Which of them is used depends on compiler optimization levels, and possibly other options.

Some compilers on x86-64 provide options to force particular behavior of floating-point instructions. Here is the same test run with some of those options:

```
# default optimized compilation
% gcc -O3 uflalt.c && ./a.out
nonvolatile smallest value = 0x0.0000000000001p-1022 in 1074 steps
   volatile smallest value = 0x0.0000000000001p-1022 in 1074 steps

# force use of x86 registers
% gcc -O3 -mfpmath=387 uflalt.c && ./a.out
nonvolatile smallest value = 0x0p+0 in 16445 steps
   volatile smallest value = 0x0.0000000000001p-1022 in 1074 steps

# force use of extended SSE instruction set
% gcc -O3 -mfpmath=sse uflalt.c && ./a.out
nonvolatile smallest value = 0x0.0000000000001p-1022 in 1074 steps
   volatile smallest value = 0x0.0000000000001p-1022 in 1074 steps

# force library calls for floating-point arithmetic
% gcc -O3 -msoft-float uflalt.c && ./a.out
nonvolatile smallest value = 0x0.0000000000001p-1022 in 1074 steps
   volatile smallest value = 0x0.0000000000001p-1022 in 1074 steps

# allow non-IEEE 754 behavior
% gcc -O3 -ffast-math uflalt.c && ./a.out
nonvolatile smallest value = 0x1p-1022 in 1022 steps
   volatile smallest value = 0x1p-1022 in 1022 steps
```

In the last case, underflow behavior was changed to flush-to-zero.

Depending on compiler options to get desired numerical results is *not* a recipe for code portability: it is much better to learn how to program in such a way as to make the code resilient to variations in floating-point behavior.

23 Memory byte order

There are two main addressing conventions for multibyte values, and a given architecture generally supports just one of them, although some can handle both. An n-byte value can be addressed by its high-order byte (*big-endian addressing*), or by its low-order byte (*little-endian addressing*).

As long as software and data do not leave their home architecture, programmers do not need to care about endianness. However, if numeric data are written in binary form to a file, or to a network connection, then it is essential that the writer and subsequent reader agree on the storage order.

DEC PDP-11 and VAX, and Intel IA-64, x86, and x86-64 architectures are little endian, whereas most others, including IBM POWER and z-Series, MIPS, Motorola 68K and 88K, SPARC, and also Internet protocols, are big endian. ARM processors can be either endian, but the choice is generally made by the operating system, and all other code must agree with the O/S.

GNU/Linux systems provide a collection of library functions for endian data conversion; they have prototypes like these:

```
#include <endian.h>

uint16_t htobe16 (uint16_t host_16bits);
uint16_t htole16 (uint16_t host_16bits);
uint16_t be16toh (uint16_t big_endian_16bits);
uint16_t le16toh (uint16_t little_endian_16bits);
```

with companions for 32-bit and 64-bit values.

Many systems supply similar functions for networking software:

```
#include <arpa/inet.h>

uint32_t htonl (uint32_t hostlong);
uint16_t htons (uint16_t hostshort);
uint32_t ntohl (uint32_t netlong);
uint16_t ntohs (uint16_t netshort);
```

There are no language standards for such functions.

Testing for host memory order requires a low-level byte test and type punning through a `union` declaration that overlays two or more objects at the same starting address:

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

union {
    float x;
```

```
        int8_t b[4];
    } u;

    u.x = 1.0F;
    (void)printf("This system is %s-endian\n",
                 (u.b[0] == 0) ? "little" : "big");
```

Here, we exploited the fact that the stored value of 1.0 is `0x3f800000`, so `u.b[0]` is `0x3f` if addressing is from the big end, and `0x00` if from the little end. This kind of code, if needed, should always be hidden in a single place in a public property-test function that hides the messy details.

24 Data alignment

There has been great variation across historical computers about how data are addressed in memory: at least one addressed by bit, many others by word (with word sizes of 4, 8, 12, 16, 18, 24, 32, 36, 48, 60, and 64 bits), and on all modern systems, by byte.

However, the low-level design of memory systems poses restrictions, or performance penalties, on memory addresses. For systems that use byte addressing, some have allowed access to an n-byte value at any address, while others require alignment to an address that is a multiple of n bytes, and still others allow access, but with a time penalty, when the data are not aligned to such a boundary.

Compilers are aware of those considerations, and for normal variable declarations, ensure that data are aligned at a suitable boundary. Thus, for scalars and arrays, data are always optimally aligned for memory access. However, many programming languages have data structures where the programmer chooses the storage order. For example, in C you can create a structure like this:

```
    struct
    {
        char        c[6];
        int8_t        i8;
        int16_t      i16;
        int32_t      i32;
        int64_t      i64;
        float        f32;
        double       f64;
        long double  f80;
        __float128  f128;
    } d;
```

Compilers may insert suitable padding between structure members to ensure suitable alignment of each member, but they are *not required to do so*.

Tests on several systems reported such padding on SPARC and x86-64 CPUs, but the padding was absent on ARM, IA-64, MIPS, PowerPC, and x86 CPUs.

Incorrect alignment might cause much slower execution on systems where an unaligned load or store must trap to the operating system. On other systems, an unaligned access might terminate the process.

The portable solution is therefore to pay attention to alignment issues, order structure elements from largest to smallest if possible, and manually supply intervening dummy padding elements wherever needed.

25 Stack storage

Many languages on modern machines support function recursion, which is most easily done by storing function arguments, and possibly also return values, on a stack. On most Unix-family operating systems, memory above the program code and static data is divided into two blocks: a *heap* that grows upward, and a *stack* that grows downward. Dynamic storage allocation is done from the heap, and stack storage is used for local data, as well as for passing arguments to functions, and sometimes, receiving their returned values.

The default stack size is determined by the operating system, and is typically about 32MB. That is sufficient for most software that contains only scalar data, but it may be inadequate for numerical programs that require large matrices.

Some systems permit the stack size to be adjusted in the command shell with commands like limit and ulimit, but the growth may capped, as shown in this example:

```
% limit
cputime      unlimited
filesize     unlimited
```

```
datasize     2097152 kbytes
stacksize    32768 kbytes
coredumpsize 512 kbytes
memoryuse    247704 kbytes
vmemoryuse   2097152 kbytes

% limit stacksize 64M

% limit stacksize 256M

% limit stacksize 512M

% limit stacksize 1024M
limit: stacksize: Can’t set limit (Operation not permitted)
```

Having to adjust the stack size with a shell command before running a program is not optimal, and the solution is then to switch from stack to static allocation: in a top-level program, change a declaration

```
double M[1000][1000];
```

to

```
static double M[1000][1000];
```

However, if your program requires new instances of large arrays in recursive function calls, the `static` modifier cannot be used in those functions. Instead, you have to resort to dynamic allocation to acquire, use, and release, large storage blocks. Code might then be written like this:

```
#include <stdio.h>
#include <stdlib.h>

#define NEED 1000000

double
frecur(double x)
{
    double *v;

    v = (double *)malloc(NEED * sizeof(double));
```

```
        if (v == (double *)NULL)
        {
            (void)fprintf(stderr,
                  "ERROR: out of memory in frecur()\n");
            abort();
        }

        /* ... work with v[] ... */

        free(v);

        return (/* something */);
    }
```

Dynamic memory management can be expensive, so use it only when there is no alternative.

If you do use it, make sure that your code has no escapes to the parent function between the `malloc()` and `free()` functions: otherwise, you have a *memory leak* that may cause failure later, possibly in a distant part of your program unrelated to the location of the leak, and maybe long after the leak happened. Such bugs are common, and often hard to find. They are especially pernicious in long-running programs, such as text editors, Web browsers, and server processes that might normally run for months.

Diagnostic software aids for plugging leaks include:

- debuggers with leak-checking options;
- replacements for the default memory allocator, such as by the Electric Fence library, `-lefence`;
- static analyzers, such as antic, cppcheck, flawfinder, its4, lint, rats, splint, and uno; and
- run-time analyzers, such as valgrind.

26 Memory access costs

We discussed the computer memory hierarchy at the beginning of this document, and now it is time to investigate the costs of access to that storage.

Computations that deal with large collections of numbers stored consecutively in matrices are good candidates for such experiments, and matrix multiplication is perhaps the easiest to understand. That operation requires that the matrices be *commensurate*: if the first factor has r rows by c columns, then the second must have c rows, and s columns. The product then has r rows and s columns. Square matrices are common, in which case r, c, and s are the same.

Assuming C-style indexing, the product of two matrices, $\boldsymbol{P} = \boldsymbol{A} \times \boldsymbol{B}$, is defined like this:

$$P_{i,j} = \sum_{k=0}^{c-1} A_{i,k} B_{k,j} \qquad \begin{array}{l} \text{for } i = 0, 1, 2, \ldots, r-1, \\ \text{for } j = 0, 1, 2, \ldots, s-1. \end{array}$$

Notice that the i, j element of $\boldsymbol{P}$ is just the *dot product* of the i-th row of $\boldsymbol{A}$ and j-th column of $\boldsymbol{B}$.

Whatever programming language is chosen to implement matrix multiplication, one or the other of the matrix factors is not accessed in storage order. If the language represents matrices via hash tables, then no elements are likely to be adjacent in memory.

In practice, matrix data are often stored in arrays that are bigger than the currently needed size, so a suitable software design must take that into account, with code like this:

```
void
matmul(int nrow_p, int ncol_p, double p[nrow_p][ncol_p],
       int nrow_a, int ncol_a, double a[nrow_a][ncol_a],
       int nrow_b, int ncol_b, double b[nrow_b][ncol_b],
       int r, int c, int s)
{   /* form r-by-s P = A * B for r-by-c A and c-by-s B */
    double sum;
    int i, j, k;

    for (i = 0; i < r; ++i)
    {
        for (j = 0; j < s; ++j)
        {
            sum = 0.0;

            for (k = 0; k < c; ++k)
                sum += a[i][k] * b[k][j];
                p[i][j] = sum;
            }
        }
    }
```

Here, we assume that variable-length arrays introduced with the 1999 ISO C Standard are supported; multidimensional array addressing in older versions of the C language is unpleasant.

A test program uses variants of that code to compute $\boldsymbol{P} = \boldsymbol{AB}$, $\boldsymbol{P} = \boldsymbol{AB^T}$, and $\boldsymbol{P} = \boldsymbol{A^T B}$, where the superscript indicates transposition. The first of those is the normal case, where in C, accesses to $\boldsymbol{B}$ are not consecutive. The second is much better, because both $\boldsymbol{A}$ and $\boldsymbol{B}$ are accessed in storage order, making effective use of cache memory. The third case is the worst of the three, because neither matrix is accessed in storage order.

The test program assumes square matrices, so that transposition is easy, and for the second and third cases, that transposition is done in-place twice, once before the multiplication, and once after, so that the matrix factors are restored to their input form. Of course, that means that they cannot be declared with the `const` attribute.

The tests are run on several vintage and current systems representing a range of CPU architectures that are summarized in Table 5. The matrices can optionally be declared with the `volatile` attribute, with measurable, but small, effects as shown in Table 6.

Figures 1 through 3 display histogram bars of times relative to the worst case, with the shortest bar in each three-case cluster indicating the fastest computation. Transposition to allow accesses to both matrices in storage order can be up to 10 times faster. Yet, even within CPU families, there is noticeable variation across models.

The slowest machine in our benchmarks, the MIPS-R10000, has a 32KB level-1 instruction cache, a 32KB level-1 data cache, and a 1MB level-2 unified cache. That is sufficient to hold three $n \times n$ matrices of type `double` for $n \leq 212$.

The fastest machine benchmarked, the Xeon-E5-1, has four cores, each with two threads. Each core has a 32KB level-1 instruction cache, a 32KB level-1 data cache, and a 256KB level-2 unified cache. In addition, there is a 10MB level-3 cache shared by all cores. Our three matrices for $n \leq 670$ can fit entirely in cache, assuming no other processes are contending for the cache.

The machine with the largest core count and cache size is the Xeon-E5-

Table 5: Benchmark machine descriptions. The operating systems include Apple Mac OS X, GNU/Linux, Sun Solaris, and SGI IRIX.

System ID	**Year**	**Description**
ARMv7l	2014	Wandboard Quad Freescale i.MX6 Cortex-A9 Quad core ARMv7l (1 4-core CPU, 1000 MHz, 2GB RAM)
Core-i7	2014	Apple Mac Mini (1 2-core Intel Core i7 CPU, 3000 MHz, 16GB RAM)
Itanium-2	2004	Dell PowerEdge 3250: Intel Itanium-2 (2 CPUs, 1400 MHz, 4GB RAM)
MIPS-R5000	1997	SGI O2 R5000-SC (1 CPU, 300 MHz)
MIPS-R10000	1997	SGI O2 R10000-SC (1 CPU, 150 MHz)
PowerPC	2002	Apple Power Mac G4 (PowerMac3,6) (2 1420 MHz PowerPC G4 (3.3) CPUs, 2GB RAM)
UltraSPARC-170	2003	Sun Blade 2000 (1 CPU, 1.015 GHz)
UltraSPARC-IIIi-0	2005	Sun Fire V440 (4 CPUs, 1.593 GHz)
UltraSPARC-IIIi-1	2006	Sun Fire V440 (4 CPUs, 1.593 GHz)
UltraSPARC-IIIi-2	2007	Sun Fire V440 (4 CPUs, 1.593 GHz)
UltraSPARC-T2	2009	Sun Enterprise T5240 (2 8-core UltraSPARC T2 Plus CPUs, 1200 MHz, 64GB RAM)
Xeon-E5-0	2008	Apple Mac Pro (1 CPU, 4 core, 2800 MHz, Intel Xeon E5462, 12GB RAM)
Xeon-E5-1	2016	HP Z440 (1 4-core CPU, 3700 MHz Intel Xeon E5-1630v3, 32GB DDR-4 RAM)
Xeon-E5-2	2016	ThinkMate/Supermicro (2 CPUs, 48 cores, 2700 MHz Intel Xeon E5-2697, 256GB RAM)
Xeon-E7	2011	IBM x3850 (8 CPUs, 64 cores, 2000 MHz, Intel Xeon E7-4820, 1024GB RAM)
Xeon-X5	201?	Dell PowerEdge R410 (2 2800 MHz Xeon X5560 CPUs, 12 cores, 24 hyperthreads, 24GB RAM)

Table 6: Relative cost of normal matrix-multiply benchmarks, $P = AB$, with the `volatile` qualifier in the matrix declarations. All floating-point variables have type `double`, corresponding to the IEEE 754 64-bit binary format, and the square matrix dimensions are a power of two.

System ID	**Cost**	**System ID**	**Cost**
ARMv7l	0.995	UltraSPARC-IIIi-1	1.071
Core-i7	1.002	UltraSPARC-IIIi-2	0.980
Itanium-2	1.081	UltraSPARC-T2	1.001
MIPS-R5000	1.010	Xeon-E5-1	1.026
MIPS-R10000	1.000	Xeon-E5-2	0.995
PowerPC	0.991	Xeon-E5-3	0.817
UltraSPARC-170	0.998	Xeon-E7	1.122
UltraSPARC-IIIi-0	1.203	Xeon-X5	0.897

2: it has a 30MB level-3 cache, sufficient for $n \leq 1150$ in our benchmarks.

Our tests all use matrix sizes that require more memory than the caches supply.

I also repeated the tests on the Xeon-E5-2, adding the `-Ofast` compiler option that is equivalent to `-O3`, and allows additional optimizations that may drop standards conformance. The best case showed a *three-fold speedup*.

As a final example of what more can be done to speed up our matrix multiplication benchmark, I added the line

```
#pragma omp parallel for private(sum, i, j, k)
```

immediately before each of the three outer loops in the benchmark program, then set the environment variable `OMP_NUM_THREADS` to 48 on the 48-core test system Xeon-E5-2, and reran the tests with the options `-fopenmp` and `-Ofast`. While the clock tick reports changed only moderately, the benchmark program ran about 16 times faster than without `-fopenmp`, thanks to the OpenMP support in the compiler that split the computation over multiple parallel threads. Further test runs could experiment with the setting of `OMP_NUM_THREADS` to find which values provide the fastest *throughput*, which is the usual goal of parallel computation.

Figure 1: Relative timing of square matrix multiplication, under compilation with `cc` *without* a `-O`*n* optimization-level option.
Each three-bar cluster shows times for matrix multiplication in C for (1) standard code: $P = AB$; (2) cache favorable code: $P = AB^T$; and (3) cache conflicts code: $P = A^T B$. The times include the before- and after-in-place transpositions of one of the matrix factors. Matrix sizes are chosen to make benchmarks run two to fifteen minutes. The sizes are normally a power of two, but clusters marked with **prime** increase the size to the next larger prime. Clusters marked with **volatile** indicate declaration of the three matrices with the `volatile` attribute.

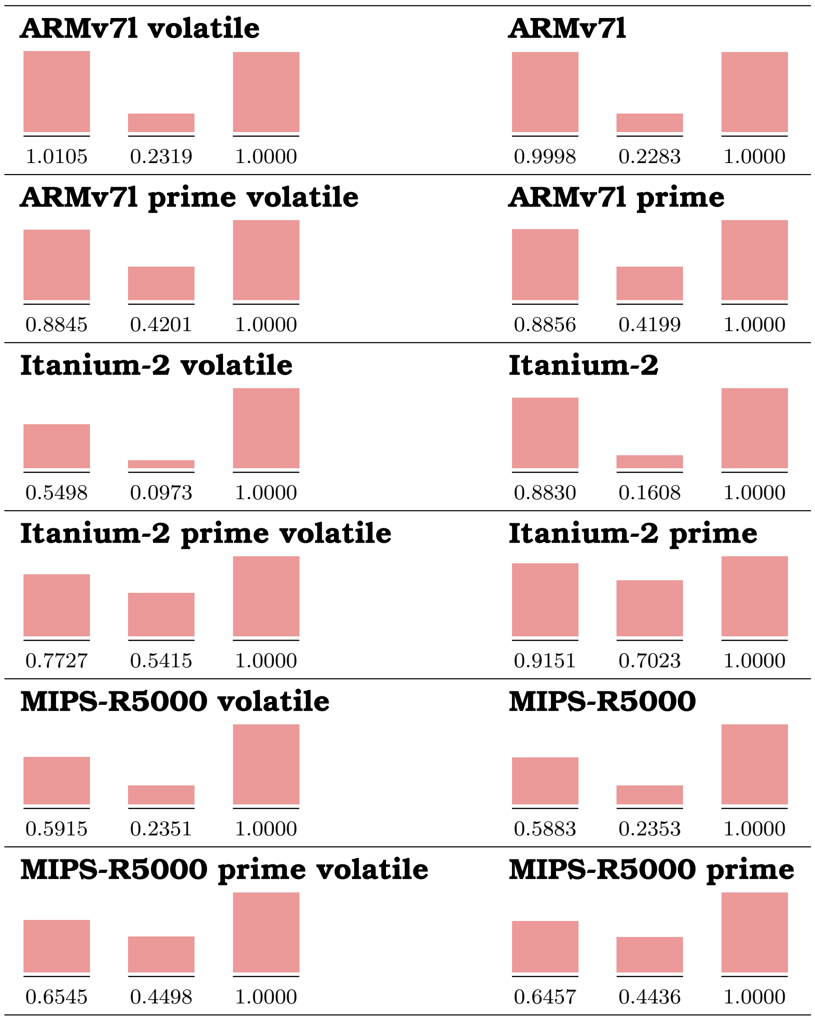


Figure 2: Relative timing of square matrix multiplication. See Figure 1 for a description of the clusters and labeling.

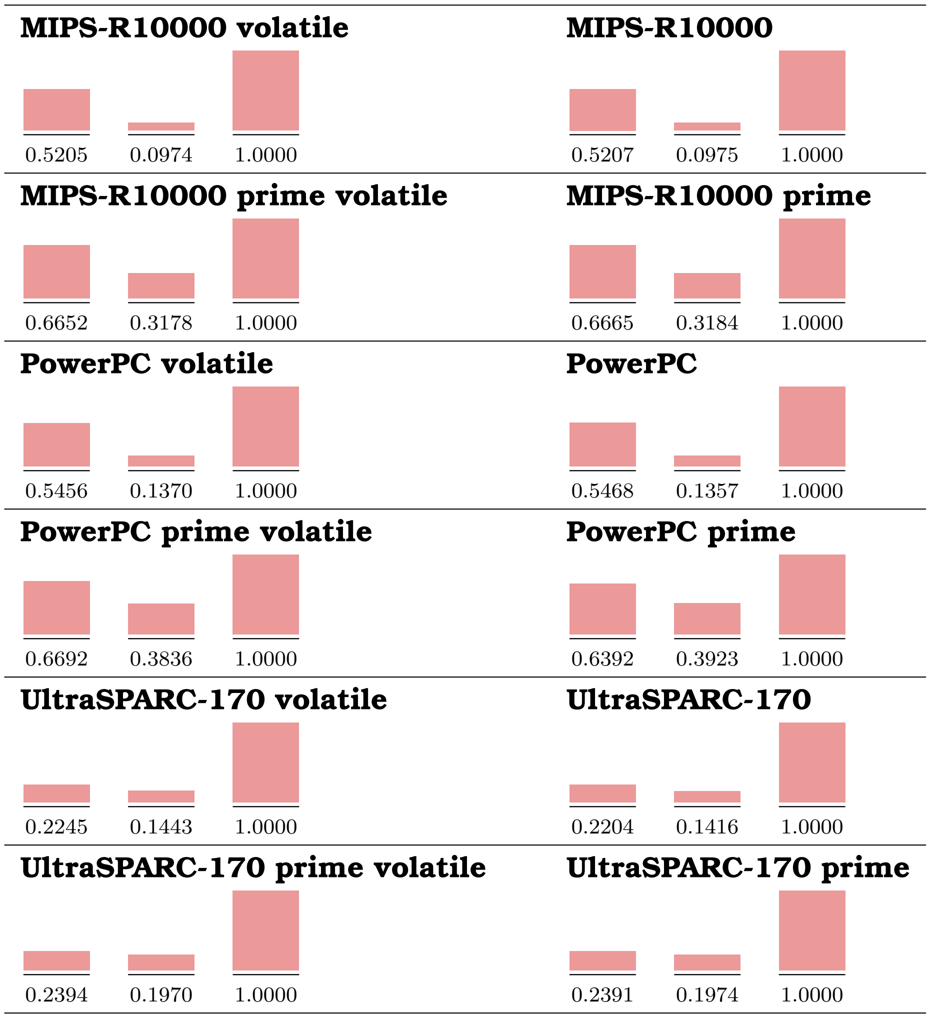


Figure 3: Relative timing of square matrix multiplication. See Figure 1 for a description of the clusters and labeling.

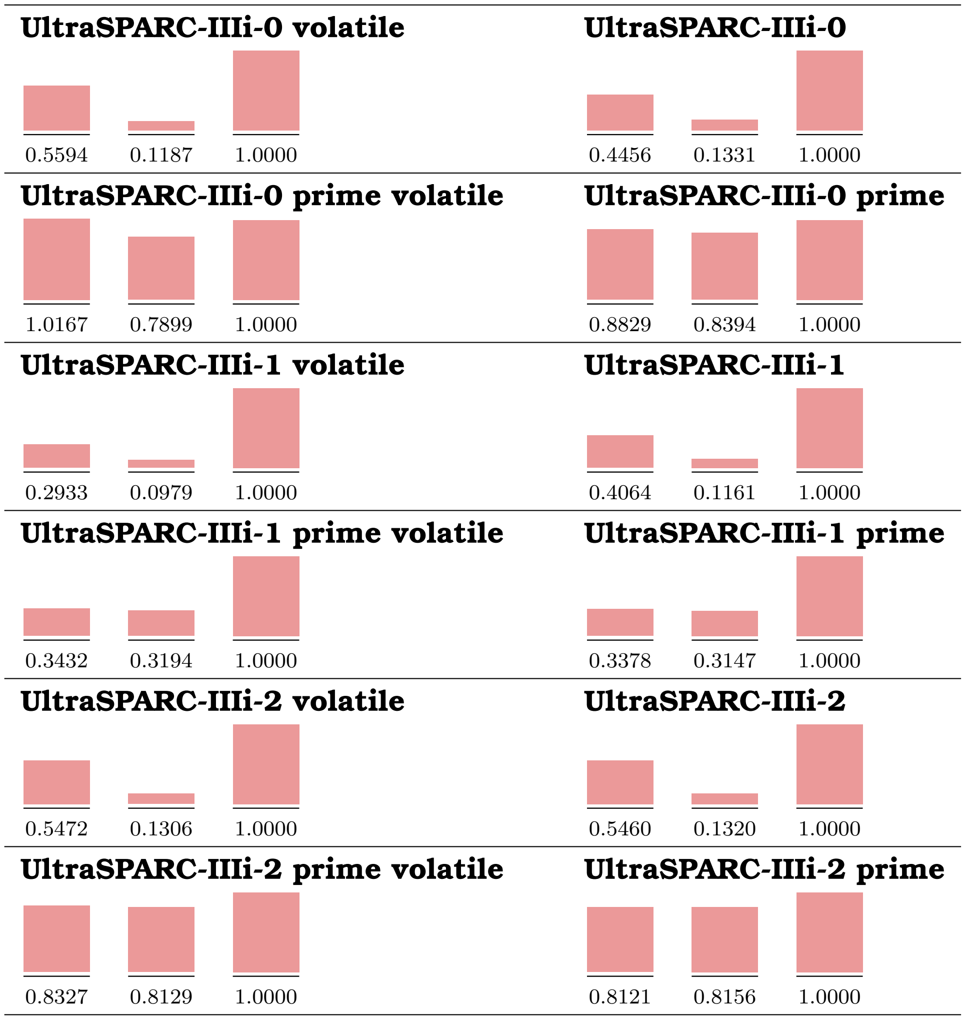


Figure 4: Relative timing of square matrix multiplication. See Figure 1 for a description of the clusters and labeling.

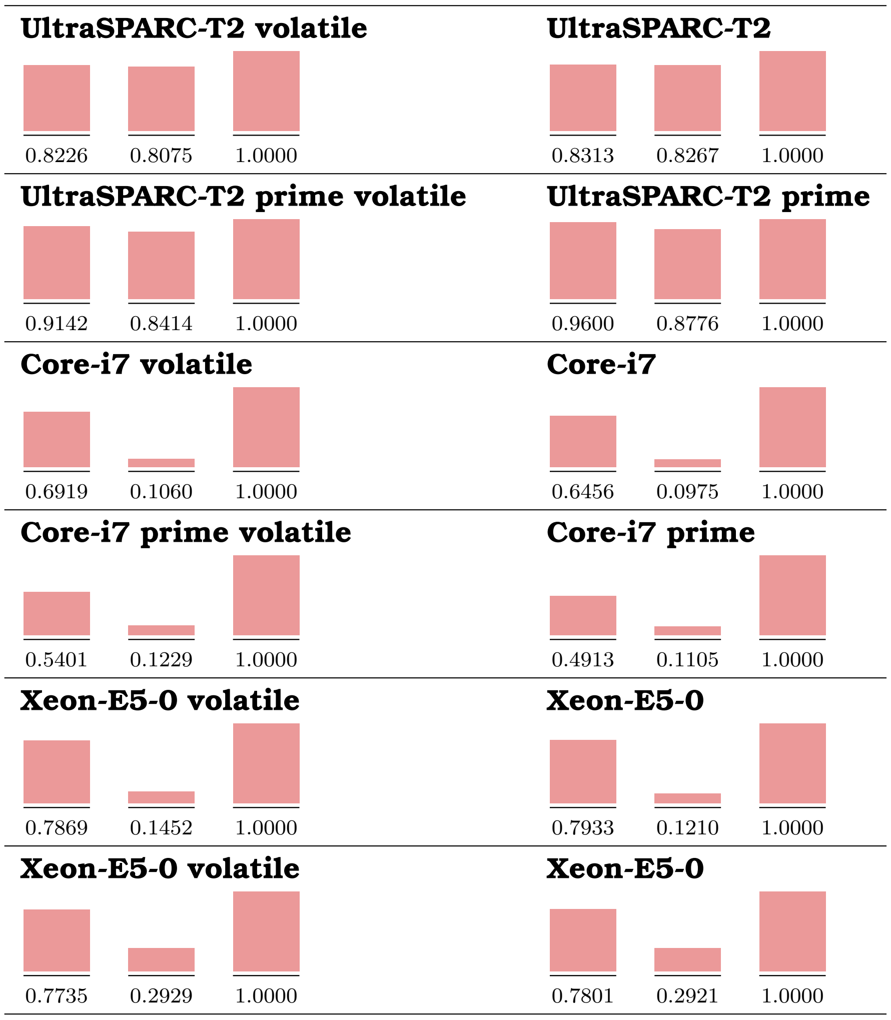
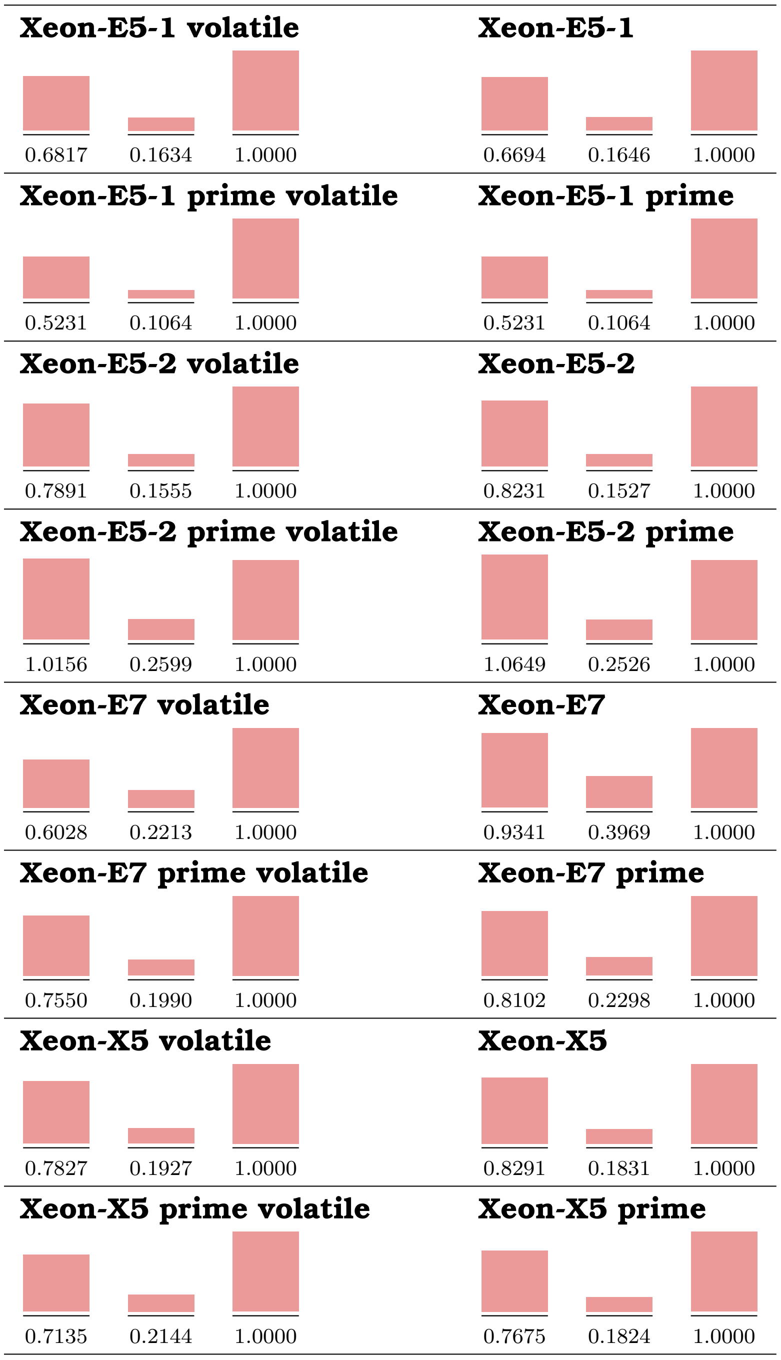


Figure 5: Relative timing of square matrix multiplication. See Figure 1 for a description of the clusters and labeling.



27 Forcing data from registers to memory

For CPU architectures where floating-point registers might have larger exponent range and precision than corresponding variables of certain predefined types, it is sometimes important to ensure that intermediate values are forced to storage format. The C and C++ languages provide the type qualifier `volatile` that can be used like this:

```
volatile double seventh;
double approx_one;

seventh = 1.0 / 7.0;
approx_one = 7.0 * seventh;
```

The compiler must assume that the value of a `volatile` variable might change unpredictably in memory, and thus, must be retrieved from there on each use, rather than being cached in a register for faster access. That language feature was introduced to deal with systems where two or more processors or devices have access to certain memory addresses, and use those locations for communication.

Few other programming languages address that issue, despite the fact that the most common workstation and server architectures today have such floating-point registers.

Here is a first stab at implementing something equivalent in Fortran:

```
real seventh
double approx_one

seventh = 1.0 / 7.0
approx_one = 7.0 * store(seventh)

...

real function store(x)
real x
store = x
end
```

The problem is that the compiler might receive the function argument in an argument register, and then just copy that register to the result register, without any memory references at all.

This workaround should do the job, *provided* that the two functions are stored and compiled separately, so as to foil compilers with interprocedural optimization:

```
real function store(x)
external fetch
real fetch
real y
common /storecb/ y
y = x
store = fetch()
end

real function fetch()
real y
common /storecb/ y
fetch = y
end
```

Because Fortran passes arguments by address, every function or subroutine can potentially modify its arguments, so compilers cannot cache arguments in registers across calls. Thus, one could instead pass the variable to a subroutine:

```
real seventh
double approx_one
seventh = 1.0 / 7.0
call store(seventh)
approx_one = 7.0 * seventh
...
subroutine store(x)
real x
end
```

Once again, separate compilation is strongly recommended.

28 Further fun

We have surveyed the most important features of (primarily) binary arithmetic on computers. To learn it well, you now need to practice it a lot. You can gain experience by doing so in multiple programming languages, If you have suitable access, you should test your code on multiple CPU architectures and operating systems. Even if you currently have only a personal laptop or desktop, you might consider acquiring one of the low cost (under US$100) systems with an ARM processor that can run any of several different operating systems in the BSD and GNU/Linux families. Alternatively, you might be able to buy online, or from government or school surplus property offices, a used workstation with a PowerPC or SPARC processor, for which there are still numerous operating system choices.

Write code for numerical algorithms that you care about, and then try your programs in multiple precisions, and with different rounding modes.

Experiment with exception handling, if your programming language provides it, so that you can write code to handle traps from things like overflow and SNaN operands.

Try to learn the rudiments of at least one computer algebra system, so that you can gain access to a huge function library, built-in graph plotting, and arbitrary-precision arithmetic.

If you program in C, C++, Julia, or Python, you can easily have access to arbitrary-precision arithmetic via their interfaces to the **G**NU **M**ultiple **P**recision arithmetic library (`-lgmp`) and its extension for computation with correct rounding, the **M**ultiple **P**recision **F**loating-Point **R**eliably library (`-lmpfr`).

For the Go language, use `import "math/big"` for access to arbitrary-precision arithmetic for integers, rational numbers, and floating-point values.

You could also experiment with implementing numerical algorithms in interval arithmetic in C, C++, Matlab, and Python.

The deficient noncontiguous array storage order in Java can be mostly remedied with techniques described in a research paper `https://doi.org/10.1002/cpe.793`, or with the packed objects defined in the class hierarchy `com.ibm.jvm.packed.*`. In both approaches, source code changes are required in how matrix elements are accessed.

The SIMH project has portable simulators for scores of historical machines dating back to the 1950s; you can then run small test programs (most likely in Fortran or C) to learn how their arithmetic behaved.

On modern operating systems on x86-64 systems, there are free virtual machine systems, such as bhyve, OVirt, QEMU/KVM, VirtManager, VirtualBox, and VMware Workstation Player. Some of those are also available on the ARM architecture. They let you install other operating systems on top of your base O/S, and there are often pre-installed virtual disk images for them that can be downloaded and started immediately.

Consider downloading and installing the pre-built MathCW library and `gcc` compilers from `http://www.math.utah.edu/pub/mathcw` to give you full access to decimal floating-point arithmetic, and a much expanded mathematical function repertoire, in C.