comp20005 Engineering Computation

Additional Notes

Numeric Computation, Part A

Lecture slides prepared by Alistair Moffat

Algorithmic objectives

For symbolic processing (for example, sorting strings), desire algorithms that are:

- Above all else, correct
- Straightforward to implement
- Efficient in terms of memory and time
- (For massive data) Scalable and/or parallelizable
- (For simulations) Statistical confidence in answers and in the assumptions made.

Algorithmic objectives

For numeric processing, desire algorithms that are:

- Above all else, correct
- Straightforward to implement
- Effective, in that yield correct answers and have broad applicability and/or limited restrictions on use
- Efficient in terms of memory and time
- (For approximations) Stable and reliable in terms of the underlying arithmetic being performed.

The last one can be critically important.

Example 1

Wish to compute

$$f(x) = x \cdot \left(\sqrt{x+1} - \sqrt{x}\right)$$

and

$$g(x) = \frac{x}{\sqrt{x+1} + \sqrt{x}}\,.$$

So write the obvious functions, and print some values...

- `sqdiff.c`

Hmmmm, why did that happen?

Example 2

Wish to compute

$$h(n) = \sum_{i=1}^{n} \frac{1}{i}$$

So write the obvious function, and print some values...

- `logsum.c`

Hmmmm, why did that happen?

Pitfalls

In all numeric computations need to watch out for:

- subtracting numbers that are (or may be) close together, because absolute errors are additive, and relative errors are magnified
- adding large sets of small numbers to large numbers one by one, because precision is likely to be lost
- comparing values which are the result of floating point arithmetic, zero may not be zero.

And even when these dangers are avoided, numerical analysis may be required to demonstrate the convergence and/or stability of any algorithmic method.

Numbers – bits, bytes, and words

Inside the computer, everything is stored as a sequence of binary digits, or bits.

Each bit can take one of two values – "0", or "1".

A byte is a unit of eight bits, and on most computers a word is a unit of either four or eight bytes.

That is, a word typically stores 32 or 64 bits.

The interpretation of those bits depends on the type of the variable associated with that word.

Variables and sizes

The preprocessor “function” `sizeof()` can be supplied with either a type or a variable, and at compilation time is replaced by the number of bytes occupied by that type:.

- `sizeof.c`

Binary numbers

In `char`, `short`, `int`, and `long` variables, the bits are used to create a binary number.

In decimal, the number 345 describes the calculation $3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$.

Similarly, in binary, the number 1101 describes the computation $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, or thirteen in decimal.

Binary numbers

Binary counting: 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, and so on.

With a little bit of practice, you can count to 1,023 on your fingers; and with a big bit of practice, to 1,048,575 if you use your toes as well.

There are two further issues to be considered:

- negative numbers, and
- the fixed number of bits w in each word.

Integer representations

In an unsigned $w = 4$ bit system, the biggest value than can be stored is 1111, or 15 in decimal.

Adding one then causes an integer overflow, and the result 0000.

Integer values are stored in fixed words, determined by the architecture of the hardware and design decisions embedded in the compiler.

Integer representations

The second column of Table 13.3 (page 232) shows the complete set of values associated with a $w = 4$ bit unsigned binary representation.

When $w = 32$, the largest value is $2^{32} - 1 = 4{,}294{,}967{,}295$.

Integer representations

Bit pattern	Integer representation		
	unsigned	sign-magn.	twos-comp.
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	−0	−8
1001	9	−1	−7
1010	10	−2	−6
1011	11	−3	−5
1100	12	−4	−4
1101	13	−5	−3
1110	14	−6	−2
1111	15	−7	−1

Sign-magnitude representation

To handle negative numbers, one bit could be reserved for a sign, and $w-1$ bits used for the magnitude of the number.

The third column of Table 13.3 shows this sign-magnitude interpretation of the 16 possible $w=4$-bit combinations.

There are two representations of the number zero.

Adding one to `INT_MAX` gives -0.

Twos-complement representation

The final column of Table 13.3 shows twos-complement representation. In it, the leading bit has a weight of $-(2^{w-1})$, rather than 2^{w-1}.

If that bit is on, and $w = 4$, then subtract $2^3 = 8$ from the unsigned value of the final three bits.

So 1101 is expanded as $1 \times -(2^3) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, which is minus three.

Twos-complement representation

The advantages of twos-complement representation are that

- there is only one representation for zero, and
- integer arithmetic is easy to perform.

For example, the difference $4 - 7$, or $4 + (-7)$, is worked out as $0100 + 1001 = 1101$, which is the correct answer of minus three.

Most computers use twos-complement representation for storing integer values.

Twos-complement representation

Revisiting an old program, it should now make more sense.

- `overflow.c`

Adding one the the biggest number in twos-complement gives the smallest number.

Twos-complement representation

On a $w = 32$-bit computer the range is from $-(2^{31}) = -2{,}147{,}483{,}648$ to $2^{31} - 1 = 2{,}147{,}483{,}647$. Beyond these extremes, `int` arithmetic wraps around and gives erroneous results.

If $w = 64$-bit arithmetic is used (type `long long`), the range is $-(2^{63})$ to $2^{63} - 1 = 9{,}223{,}372{,}036{,}854{,}775{,}807$, approximately plus and minus nine billion billion, or 9×10^{18}

The type `char` is also an integer type, and using 8 bits can store values from $-(2^7) = -128$ to $2^7 - 1 = 127$

Floating point representations

The floating point types `float` and `double` are stored as:

- a one bit sign, then
- a w_e-bit integer exponent of 2 or 16, then
- a w_m-bit mantissa, normalized so that the leading binary (or sometimes hexadecimal) digit is non-zero.

Floating point representations

When $w = 32$, a `float` variable has around $w_m = 24$ bits of precision in the mantissa part. This corresponds to about 7 or 8 digits of decimal precision.

In a `double`, around $w_m = 48$ bits of precision are maintained in the mantissa part.

Floating point representations

For example, when $w = 16$, $w_s = 1$, $w_e = 3$, $w_m = 12$, the exponent is a binary numbers stored using w_e-bit twos-complement representation, and the mantissa is a w_m-bit binary fraction:

Number (decimal)	Number (binary)	Exponent (decimal)	Mantissa (binary)	Representation (bits)
0.5	0.1	0	.100000000000	0 000 1000 0000 0000
0.375	0.011	-1	.110000000000	0 111 1100 0000 0000
3.1415	$11.001001000011\cdots$	2	.110010010000	0 010 1100 1001 0000
-0.1	$-0.0001100110011\cdots$	-3	.110011001100	1 101 1100 1100 1100

The exact decimal equivalent of the last value is -0.0999755859375. Not even 0.1 can be represented exactly using fixed-precision binary fractional numbers.

Floating point representations

Floating point representations can also be investigated:

- `floatbits.c`

0.0	is	00000000 00000000 00000000 00000000
1.0	is	00111111 10000000 00000000 00000000
–1.0	is	10111111 10000000 00000000 00000000
2.0	is	01000000 00000000 00000000 00000000
10.5	is	01000001 00101000 00000000 00000000
20.1	is	01000001 10100000 11001100 11001101

Octal and hexadecimal

C supports `int` constants via octal (base 8) and hexadecimal (base 16) values. Beware! Any integer constant that starts with 0 is taken to be octal:

```
int o = 020;
int h = 0x20;
printf("o = %oo, %dd, %xx\n", o, o, o);
printf("h = %oo, %dd, %xx\n", h, h, h);
```

gives

```
o = 20o, 16d, 10x
h = 40o, 32d, 20x
```

Octal and hexadecimal

The standard Unix tool `bc` can be used to do radix conversions:

```
mac: bc
ibase=10
obase=2
25
11001
obase=8
25
31
obase=16
25
19
```

Unsigned types

C offers a set of alternative integer representations, `unsigned char`, `unsigned short`, and `unsigned int` (or just `unsigned`).

Variables that are `unsigned` store positive binary values, and can be used to manipulate raw bit strings. Negative numbers cannot be stored.

But they will get printed out if you use `"%d"` format descriptors. Use `"%u"` instead, or `"%lu"`.

Unsigned types

Modern C implementations also allow the use of the types `int8_t` (normally a `char`), `int16_t` (normally a `short`), and `int32_t` (normally an `int`).

Similarly, `uint8_t`, `uint16_t`, and `uint32_t`.

The types `long long` (usually a 64-bit integer, or `int64_t`) and `long double` (usually a 128-bit floating point value) are also sometimes provided.

Bit-level operators

C provides low-level operations for isolating and setting individual bits in `int` and `unsigned` variables.

These include left-shift (<<), right-shift (>>), bitwise and (&), bitwise or (|), bitwise xor (^), and complement (~).

There are differences between `int` and `unsigned` when bit shifting operations are carried out. If you need to use these operations, make sure you now what they are.

- `intbits.c`

Bit-level operators

Table 13.5 (page 236) gives a final precedence ordering that includes all of these specialized bit operations. If in doubt (always (over (parenthesize))).

More types

And, to end with two programming jokes:

Why do programmers always get Christmas and Halloween confused? Because $31_{Oct} = 25_{Dec}$!.

There are 10 types of people in the world: those who know binary, and those who don't. (Boom boom).