# NUMERIC COMPUTATION

Yujing M. Jiang
The University of Melbourne, 2018

# Things you need to go over before final exam

- numericA.pdf

# 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.

# Algorithmic objectives: Example 1

To calculate: $f(x) = x \cdot (\sqrt{x+1} - \sqrt{x})$

Equvalient to: $\mathrm{g}(x) = \frac{x}{\sqrt{x+1}+\sqrt{x}}$

# Algorithmic objectives: Example 1 Result

```
x = 1.000e+00, f(x) = 4.1421356797e-01, g(x) = 4.1421356797e-01
x = 1.000e+01, f(x) = 1.5434713364e+00, g(x) = 1.5434713364e+00
x = 1.000e+02, f(x) = 4.9875621796e+00, g(x) = 4.9875621796e+00
x = 1.000e+03, f(x) = 1.5807437897e+01, g(x) = 1.5807437897e+01
x = 1.000e+04, f(x) = 4.9998748779e+01, g(x) = 4.9998748779e+01
x = 1.000e+05, f(x) = 1.5811349487e+02, g(x) = 1.5811349487e+02
x = 1.000e+06, f(x) = 4.9999987793e+02, g(x) = 4.9999987793e+02
x = 1.000e+07, f(x) = 1.5811387939e+03, g(x) = 1.5811387939e+03
x = 1.000e+08, f(x) = 0.0000000000e+00, g(x) = 5.0000000000e+03
x = 1.000e+09, f(x) = 0.0000000000e+00, g(x) = 1.5811388672e+04
x = 1.000e+10, f(x) = 0.0000000000e+00, g(x) = 5.0000000000e+04
x = 1.000e+11, f(x) = 0.0000000000e+00, g(x) = 1.5811387500e+05
x = 1.000e+12, f(x) = 0.0000000000e+00, g(x) = 5.0000000000e+05
```

# Algorithmic objectives: Mathematical proof

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

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

When $x$ gets larger $\lim_{x \to \infty} \sqrt{x+1} - \sqrt{x} = 0$

Therefore $f(x)|_{x \to \infty} \to 0$ in C programming

# Algorithmic objectives: Example 2

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

$$h(n) = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

```
int i, sum = 0;
for (i = 1; i < n; i++)
    sum += 1/i;
```

$$h(n) = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}$$

```
int i, sum = 0;
for (i = n; i >= 1; i--)
    sum += 1/i;
```

# Algorithmic objectives: Example 2 Result

```
n=        1, f(x) =  1.00000000000000, g(x) =  1.00000000000000
n=        2, f(x) =  1.50000000000000, g(x) =  1.50000000000000
n=        4, f(x) =  2.08333492279053, g(x) =  2.08333253860474
n=        8, f(x) =  2.71785736083984, g(x) =  2.71785712242126
................................................................
n=    32768, f(x) = 10.97440910339355, g(x) = 10.97444343566895
n=    65536, f(x) = 11.66742801666260, g(x) = 11.66758823394775
n=   131072, f(x) = 12.36008548736572, g(x) = 12.36073207855225
n=   262144, f(x) = 13.05130386352539, g(x) = 13.05388069152832
n=   524288, f(x) = 13.73701763153076, g(x) = 13.74705600738525
n=  1048576, f(x) = 14.40368366241455, g(x) = 14.44023132324219
n=  2097152, f(x) = 15.40368270874023, g(x) = 15.13289928436279
n=  4194304, f(x) = 15.40368270874023, g(x) = 15.82960700988770
n=  8388608, f(x) = 15.40368270874023, g(x) = 16.51415252685547
n= 16777216, f(x) = 15.40368270874023, g(x) = 17.23270797729492
```

# Algorithmic objectives: Mathematical proof

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

The harmonic series diverges.

However, $\lim_{i \to \infty} a_i = 0$

# Number Representation: Abstract

- Similar to ***<u>Scientific Notation</u>***: $+1.234567 \times 10^{12}$
- Stored as 3 parts:
- 1. Sign (+ or −)
- 2. Fraction (aka. Mantissa, $1.234567$)
- 3. Exponential offset (<u>12</u> in $\times\ 10^{\mathbf{12}}$)
- ***<u>However</u>***, The precision of fraction is limited. Normally 6 decimal points (7 digits in total) for `float`.

# Adding a (relatively) small number to a large number

- 1. Precision of fraction is limited.
- 2. Exponential alignment (to the larger number)
- E.g. $1.234567 \times 10^9 + 1.0$ turns to:
- $\quad\ \ 1.234567000 \times 10^9$
- $+\ 0.000000001 \times 10^9$
- $\quad\ \ 1.234567001 \times 10^9$
- However, only 6 decimal points are reserved.
- The output is: $1.234567 \times 10^9$. The same number?

# Algorithmic objectives: Example 2

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

***What happens?***

$$h(n) = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

```
int i, sum = 0;
for (i = 1; i < n; i++)
    sum += 1/i;
```

$$h(n) = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}$$

```
int i, sum = 0;
for (i = n; i >= 1; i--)
    sum += 1/i;
```

# Algorithmic objectives: Logical proof

- Known: Adding a relatively small number to a large number can lose precision.
- $a_i = \frac{1}{i}$, and $\lim_{i \to \infty} a_i = 0$
- For every iteration: Large + Small ➔ Large + ZERO
- However, it ***drops*** all small numbers added.
- Large + many * Small ➔ Large + many * ZERO

$$10{,}000{,}000(1 \times 10^7) + 1 = 10{,}000{,}000$$

$$10{,}000{,}000(1 \times 10^7) + \underbrace{1 + 1 + \ldots\ldots + 1}_{\text{More than 10^7 * 1}} = 10{,}000{,}000\ still!$$

# 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.

# Binary numbers

- 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.

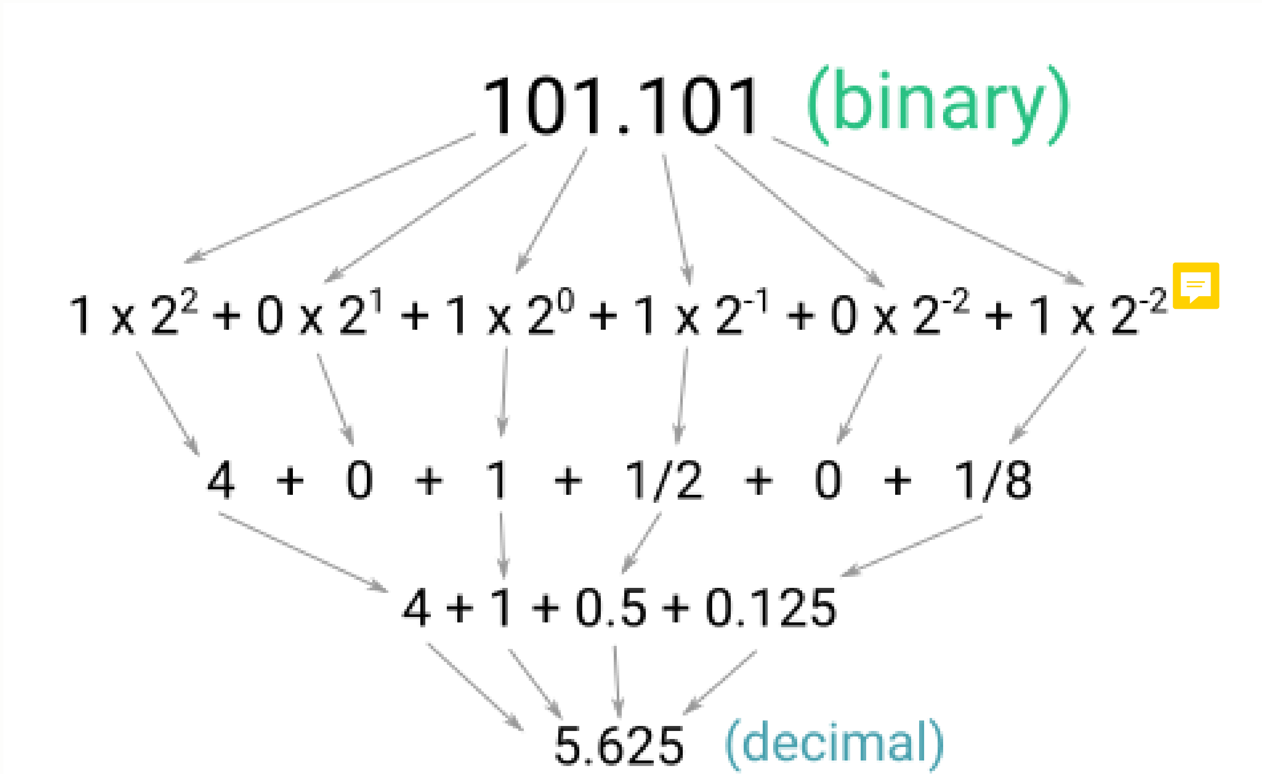
| $10^3$ | $10^2$ | $10^1$ | $10^0$ |
| --- | --- | --- | --- |
| 0 | 3 | 4 | 5 |

Decimal

| $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| --- | --- | --- | --- |
| 1 | 1 | 0 | 1 |

Binary

# Binary numbers



# 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 |

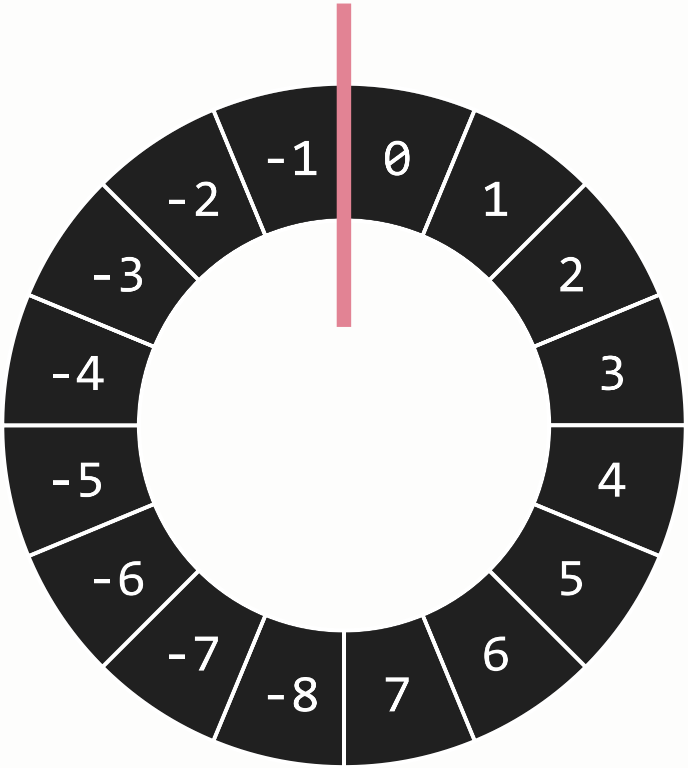
# Binary: Sign-magnitude

| $Sign$ | $2^2$ | $2^1$ | $2^0$ | $Decimal$ |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | + 5 |
| 1 | 1 | 0 | 1 | − 5 |
| 0 | 0 | 0 | 0 | + 0 |
| 1 | 0 | 0 | 0 | − 0 |
| 0 | 0 | 1 | 0 | + 2 |
| 1 | 0 | 1 | 0 | − 2 |

$Sign$: 0 + 1 −

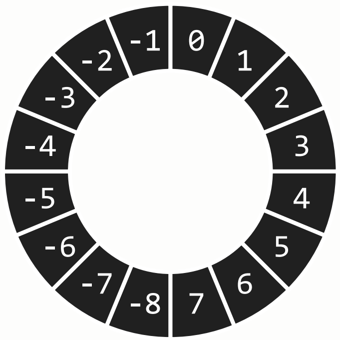
| Bit pattern | representation sign-magn. |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | −0 |
| 1001 | −1 |
| 1010 | −2 |
| 1011 | −3 |
| 1100 | −4 |
| 1101 | −5 |
| 1110 | −6 |
| 1111 | −7 |

# Binary: Two’s complement




| Bit pattern | twos-comp. |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | −8 |
| 1001 | −7 |
| 1010 | −6 |
| 1011 | −5 |
| 1100 | −4 |
| 1101 | −3 |
| 1110 | −2 |
| 1111 | −1 |

# Binary: Two's complement




How to represent a negative number?

1. Flip all bits of the positive binary.
2. Plus 1.

| Bit pattern | twos-comp. |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | −8 |
| 1001 | −7 |
| 1010 | −6 |
| 1011 | −5 |
| 1100 | −4 |
| 1101 | −3 |
| 1110 | −2 |
| 1111 | −1 |

$$
\begin{array}{lccccc}
 & Sign & 2^2 & 2^1 & 2^0 & Decimal \\
 & 0 & 1 & 0 & 1 & +5 \\
Flip: & 1 & 0 & 1 & 0 & \\
+1\ : & 1 & 0 & 1 & 1 & -5 \\
\hline
 & 0 & 0 & 1 & 0 & +2 \\
Flip: & 1 & 1 & 0 & 1 & \\
+1\ : & 1 & 1 & 1 & 0 & -2
\end{array}
$$

# Overflow

In C Programming:

Storage: ***<u>Sign-magn</u>***.

Operation: ***<u>Unsigned</u>***

```
int4 i = 7;
i = i + 1;
printf("%d ", i);
// Output: -0

int4 i = 7;
i = i + 2;
printf("%d ", i);
// Output: -1
```

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

# Number Representation: Decimal

- Similar to *Scientific Notation*: $+1.234567 \times 10^{12}$
- Stored as 3 parts:
- 1. Sign ($+$ or $-$)
- 2. Fraction (aka. Mantissa, $1.234567$)
- 3. Exponential offset (12 in $\times 10^{\mathbf{12}}$)

# Number Representation: Binary

- Similar to ***<u>Scientific Notation</u>***: $+0.101 \times 2^3$ $(\mathbf{101}/\boldsymbol{five})$
- Stored as 3 parts:
- 1. Sign (+ or −)
- 2. Fraction (aka. Mantissa, $0.101$)
- 3. Exponential offset (<u>3</u> in $\times\ 2^3$)
- However, in fraction (mantissa), the standard form is 0.xxxxx

# Number Representation: Binary

- Similar to ***Scientific Notation***: $+0.101 \times 2^3$
- Stored as 3 parts:
- 1. Sign (+ or −)
- 2. Fraction (aka. Mantissa, $1.01$)
- 3. Exponential offset (3 in $\times\, 2^3$)
- However, in fraction (mantissa), the standard form is 0.xxxx.

| 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· · · | 2 | .110010010000 | 0 010 1100 1001 0000 |
| −0.1 | −0.0001100110011 · · · | −3 | .110011001100 | 1 101 1100 1100 1100 |