Lecture 4
Representing Data on the Computer

Ramani Duraiswami

AMSC/CMSC 662

Fall 2009

- $x = \pm(1+f)\times 2^{e}$
- $0 \cdot f < 1$
- $f = (\text{integer} < 2^{52}) / 2^{52}$

- $-1022 \le e \le 1023$
- $e = \text{integer}$

Effects of floating point

Finite f implies finite *precision*.

Finite e implies finite *range*

Floating point numbers have discrete spacing, a maximum and a minimum.

Effects of floating point

- eps *is the distance from* 1 *to the next larger floating-point number.*
- eps = 2^{-52}
- In Matlab

	Binary	Decimal
eps	2^(-52)	2.2204e-16
realmin	2^(-1022)	2.2251e-308
realmax	(2-eps)*2^1023	1.7977e+308

Rounding vs. Chopping

- **Chopping**: Store x as c, where $|c| < |x|$ and no machine number lies between c and x.
- **Rounding**: Store x as r, where r is the machine number closest to x.
- **IEEE standard arithmetic uses rounding**.

Machine Epsilon

- **Machine epsilon** is defined to be the smallest positive number which, when added to 1, gives a number different from 1.
  - Alternate definition (1/2 this number)
- **Note:** Machine epsilon depends on d and on whether rounding or chopping is done, but does not depend on m or M!

Some numbers cannot be exactly

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \frac{1}{2^{12}} + ...$$

$$t = (1 + \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^3} + ... + \frac{9}{16^{12}} + \frac{10}{16^{13}}) \cdot 2^{-4}$$

```
x = 1; while 1+x > 1, x = x/2, pause(.02), end

x = 1; while x+x > x, x = 2*x, pause(.02), end

x = 1; while x+x > x, x = x/2, pause(.02), end
```

Floating point operations

- Basic arithmetic operations
  - addition, subtraction, multiplication, and division (and sometimes the square root).
- The IEEE standard specifies that these operations must return the correctly rounded result (provided they are within the range of normalized oating-point numbers).
- (a o b) = (a o b)(1 + ε);

- Order of computations becomes important
- sum1 = a + (b + c)   sum2 = (a + b) + c
  - sum1 and sum2 may not have the same value.
  - 472635.0000+ 27.5013- 472630.0000= 32.5013
  - If we compute this sum in the order given in six digit arithmetic, we get
  - 472635.0000+ 27.5013 =472663.0000
  - 472663-472630=33
  - which is accurate to only two digits. On the other hand
  - 472635 - 472630= 5:   5.0000 + 27.5013 =32.5013

Errors can be magnified

- Errors can be magnified during computation.
- Let us assume numbers are known to 0.05% accuracy
- Example: 2.003×10^0 and 2.000×10^0
  - both known to within $\pm .001$
- Perform a subtraction. Result of subtraction:

$$0.003 \times 10^0$$

- but true answer could be as small as 2.002 - 2.001 = 0.001,
- or as large as 2.004 - 1.999 = 0.005!
- Absolute error of 0.002
- Relative error of 200% !
- **Adding or subtracting causes the bounds on absolute errors to be added**

Error effect on multiplication/division

- Let x and y be true values
- Let $X=x(1+r)$ and $Y=y(1+s)$ be the known approximations
- Relative errors are r and s
- What is the errors in multiplying the numbers?
- XY=xy(1+r)(1+s)
- Absolute error $=|xy(1-rs-r-s-1)|= (rs+r+s)xy$
- Relative error $= |(xy-XY)/xy|$

  $= |rs+r+s| <= |r|+|s|+|rs|$
- If r and s are small we can ignore $|rs|$
- Multiplying/dividing causes relative error bounds to add

Effects of floating point errors

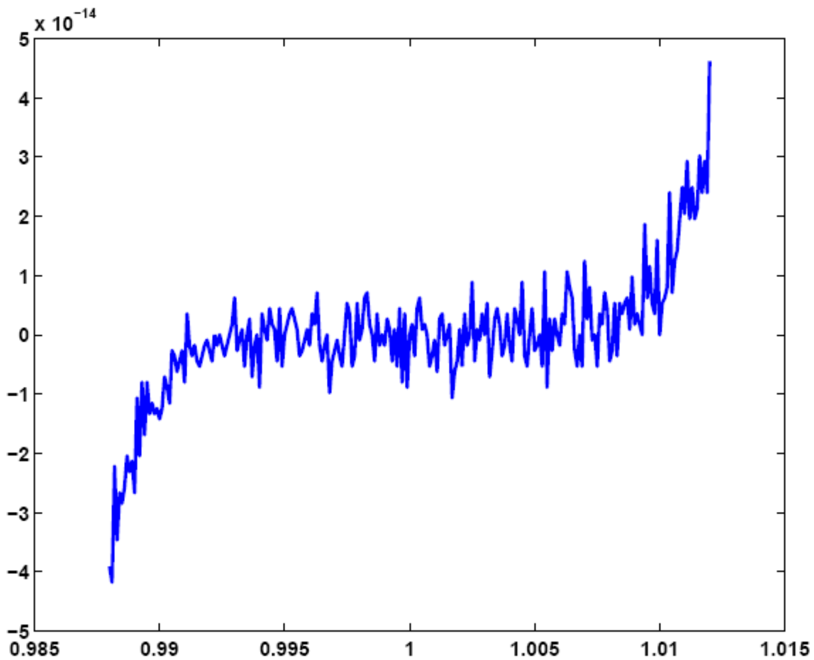
- Singular equations will only be nearly singular
- Severe cancellation errors can occur

$$
\begin{aligned}
17x_1 + 5x_2 &= 22 \\
1.7x_1 + 0.5x_2 &= 2.2
\end{aligned}
$$

```
A = [17 5; 1.7 0.5]
b = [22; 2.2]
x = A\b
```

```
x = 0.988:.0001:1.012;
y = x.^7-7*x.^6+21*x.^5-35*x.^4+35*x.^3-21*x.^2+7*x-1;
plot(x,y)
```

produce




```
x =
  -1.0588
   8.0000
```

Measuring error

- **Absolute error** in c as an approximation to x:

$$|x - c|$$

- **Relative error** in c as an approximation to nonzero x:

$$|(x - c)/x|$$

Floating point exceptions

Exceptions set a flag that can be queried by the programmer

Some are trapped and the system can be set to abort the process

Operations involving Infs and NaNs have a certain logic, and essentially continue to produce them

Overflow leads to Infs

0/0 inf/inf, sqrt of negative number etc. lead to NaN

Underflow is a non fatal exception

IEEE requires gradual underflow

of $\pi \cdot 10^k$ as k approaches the underflow point.

k	representation
-98	3.14159 -98
-99	3.14159 -99
-100	0.31416 -99
-101	0.03142 -99
-102	0.00314 -99
-103	0.00031 -99
-104	0.00003 -99
-105	0.00000 00

Bit operations

- Bitwise logical operations
  - combine corresponding bits of two words to give another word.
  - operations: **and**, **or**, **xor** (exclusive or), complement
- Example
  - 0011 or 0101 = 0111
  - 0011 and 0101 = 0001
  - 0011 xor 0101 = 0110
  - 0011 complement = 1100
- Bitwise shift operations come in two flavors: logical and arithmetic
- Logical shifts simply shift the bits and replace spaces with zeros
  - 10110110 shifted right by three bits is 00010110.
  - Shifted left by three bits it becomes 10110000.
  - Bits that are shifted out are lost.
  - Main use is quick multiplication and division by 2
- The main use for shifts: quickly multiply and divide by powers of 2 of integers
  - multiplying by 00101 by 2 amounts to doing a left shift to 01010
  - multiplying by 4 amounts to doing two left shifts to 10100
- If numbers are too large, multiplication doesn't produce valid results
  - e.g., 10000000 (128d) cannot be left-shifted to obtain 256 using 8-bit values
- Similarly, dividing by powers of two amounts to doing right shifts:
  - right shifting 10010 (18d) leads to 01001 (9d)

Other shifts

- Arithmetic shift
  – Give the same quick instruction level multiplication ability to signed numbers
  – (note that signed integers are stored in 2’s complement notation, so you will have to understand how they work)
- Rotate shifts
  – Bits that move out from the right (or left) reappear on the left (or right)

Character Representations

- ASCII – PC workstations
- EBCDIC – IBM Mainframes
- Unicode – International Character sets

ASCII

- Original ASCII: American Standard Code for Information Interchange
  7-bit coded character set for information interchange
- Specifies coding of space and a set of 94 characters (letters, digits and punctuation or mathematical symbols) suitable for the interchange of basic English language documents.
- Extended ASCII: 8 bit characters
  - All western European languages
- Groups of characters are called a string
- Several functions in programming languages to manipulate strings

7 – bit ASCII Code Set

Decimal	Hex	Char.	Comment	Decimal	Hex	Char.	Decimal	Hex	Char.	Decimal	Hex	Char.
0	00	NUL	Null	32	20	Space	64	40	@	96	60	`
1	01	SOH	Start of Heading	33	21	!	65	41	A	97	61	a
2	02	STX	Start of Text	34	22	"	66	42	B	98	62	b
3	03	ETX	End of Text	35	23	#	67	43	C	99	663	c
4	04	EOT	End of Transmission	36	24	$	68	44	D	100	64	d
5	05	ENQ	Enquiry	37	25	%	69	45	E	101	65	e
6	06	ACK	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	BEL	Bell (Ding!)	39	27	'	71	47	G	103	67	g
8	08	BS	Backspace	40	28	(	72	48	H	104	68	h
9	09	HT	Horizontal Tab	41	29	)	73	49	I	105	69	i
10	0A	LF	Line Feed	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	Vertical Tab	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	Form Feed (new page)	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	Carraige Return	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	Shift Out	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	Shift In	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	Data Link Escape	48	30	0	80	50	P	112	70	p
17	11	DC1	Device Control 1	49	31	1	81	51	Q	113	71	q
18	12	DC2	Device Control 2	50	32	2	82	52	R	114	72	r
19	13	DC3	Device Control 3	51	33	3	83	53	S	115	73	s
20	14	DC4	Device Control 4	52	34	4	84	54	T	116	74	t
21	15	NAK	Negative Acknowledge	53	35	5	85	55	U	117	75	u
22	16	SYN	Synchronous Idle	54	36	6	86	56	V	118	76	v
23	17	ETB	End of Transmission Block	55	37	7	87	57	W	119	77	w
24	18	CAN	Cancel	56	38	8	88	58	X	120	78	x
25	19	EM	End of Medium	57	39	9	89	59	Y	121	79	y
26	1A	SUB	Substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	File Separator	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	Group Separator	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	Record Separator	62	3E	>	94	5E	^	126	7E	~
31	1F	US	Unit Separator	63	3F	?	95	5F	_	127	7F	DEL (Delete)

Unicode

- Two byte character set to represent all of the world's characters in modern computer use,
- including technical symbols and special characters used in publishing.
- separate values for up to 65,536 characters. Unicode-enabled functions are often referred to as "wide-character" functions.

Performance

- One way to compare algorithms that solve the same problem is to count the number of floating-point operations they perform.
- Although such counts underestimate the execution time, that time is frequently proportional to the count.
- Count depends on the size of the problem (the *order*) and a constant
- In comparing two algorithms of the same order, one must examine the order constant.
- For constants of different order, the one with the higher order will ultimately be faster.
  - But ultimately may never come in practice.

Matrix-vector product

- Matrix-vector multiplication applies a linear transformation to a vector:

$$\boxed{\mathbf{M} \bullet \mathbf{v} = \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \mathbf{M}_{23} \\ \mathbf{M}_{31} & \mathbf{M}_{32} & \mathbf{M}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_z \end{bmatrix}}$$

- How many operations does a matrix vector product take?

matrix vector product

- Access matrix
  - Element-by-element along rows
  - Element-by-element along columns
- In either case there are two loops
- Inner loop has one multiply and add
- Done N times
- Outer loop is done M times

```
[m,n]=size(A);
y = zeros(m,1);
for i=1:m,
   for j=1:n,
        y(i) = y(i) + A(i,j)*x(j);
   end
end
```

```
[m,n]=size(A);
y = zeros(m,1);
for i=1:m,
     y(i) = A(i,:) * x;
end
```

```
[m,n]=size(A);
y = zeros(m,1);
for j=1:n,
    y = y + A(:,j)*x(j);
end
```

Asymptotic Equivalence

- $f(n) \sim g(n)$

$$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = 1$$

Little Oh

- *Asymptotically smaller*:
- $f(n) = o(g(n))$

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$$

Big Oh

- *Asymptotic Order of Growth:*

- $f(n) = O(g(n))$

$$\limsup_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) < \infty$$

The Oh's

$$\text{If } f = \mathrm{o}(g) \;\text{ or }\; f \sim g \;\text{ then }\; f = \mathrm{O}(g)$$

$$\lim = 0 \qquad \lim = 1 \qquad \lim < \infty$$

The Oh's

$$\text{If } f = o(g), \quad \text{then } g \neq O(f)$$

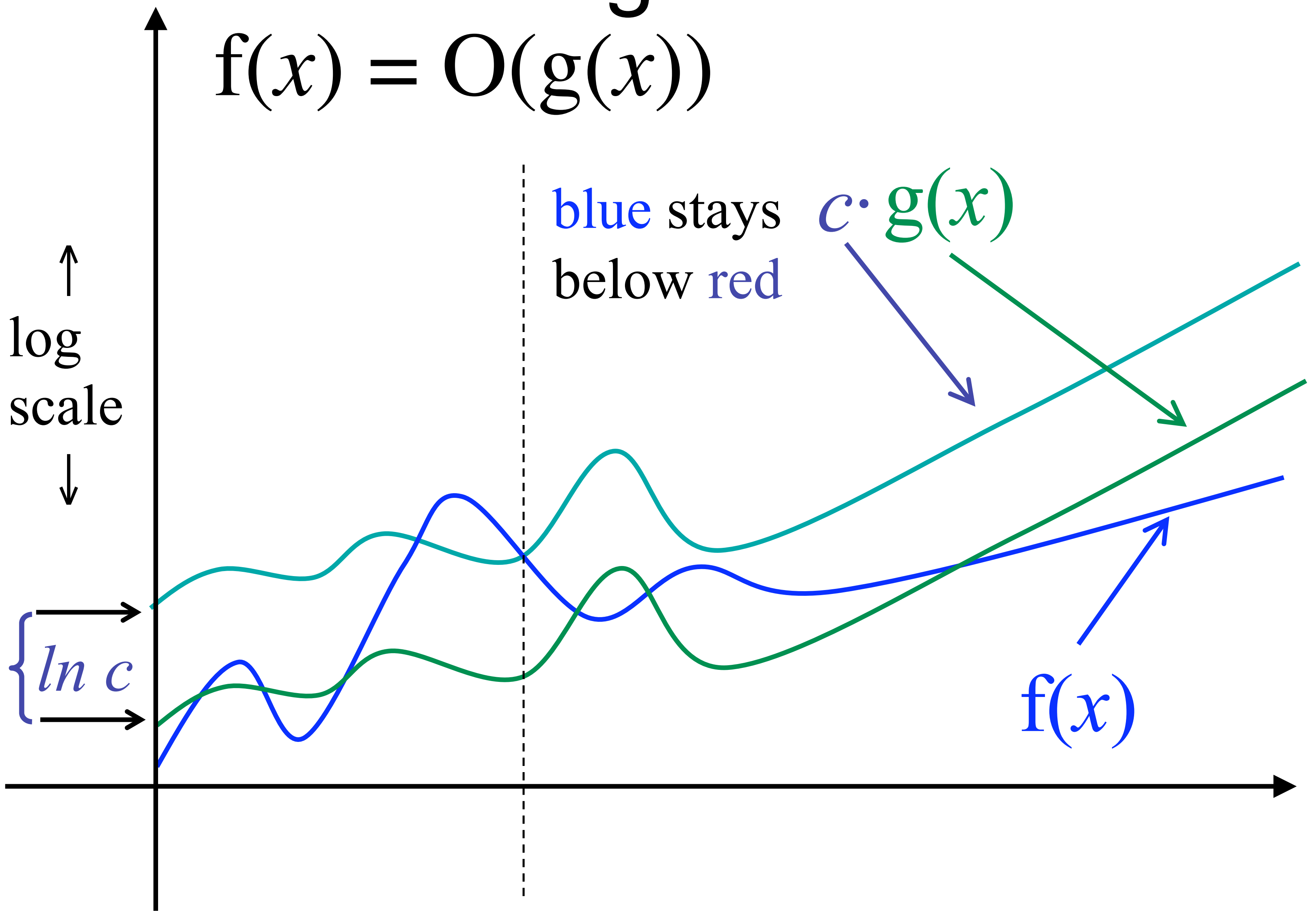
$$\lim \frac{f}{g} = 0 \qquad\qquad \lim \frac{g}{f} = \infty$$

Big Oh

- Equivalently,
  - f(n) = O(g(n))

$$\exists c, n_0 \geq 0 \ \forall n \geq n_0 \ |f(n)| \leq c \cdot g(n)$$

Big Oh



Computer Performance

- Complexity
  - Time
  - Memory
  - Communication
  - Notation
- Statistical measurement

Amdahl’s law

- When a task can be divided into independent subtasks, speeding up one of the subtasks will speed up the original task.
- Amdahl's law is a formula that shows the limits of speeding up a subtask and suggests which one to work on. In its form it also embodies the law of diminishing returns.
- E.g.: Two subtasks A and B
- $T=T_A+T_B$
- How much do we gain by altering task B to reduce the time T_B?
- Let T represents the time to execute a program, T_A is the CPU time spent actually spent executing the program, and T_B is the time when the CPU is idle waiting for input.
- To make the notion of `gain' precise, let T_B be reduced to T_B/σ, where $\sigma > 1$. Then the new total time is

$$T_\sigma = T_\mathrm{A} + \sigma^{-1} T_\mathrm{B}.$$

Speedup is defined as

$$S(\sigma) = \frac{T}{T_\sigma} = \frac{T_\mathrm{A} + T_\mathrm{B}}{T_\mathrm{A} + \sigma^{-1} T_\mathrm{B}}.$$

Diminishing returns

$$f_{\mathrm{A}} = \frac{T_{\mathrm{A}}}{T} \quad \text{and} \quad f_{\mathrm{B}} = \frac{T_{\mathrm{B}}}{T}.$$

Thus f_{A} is the fraction of the total time accounted for by task A, a
These quantities are not independent but satisfy the relation

$$f_{\mathrm{A}} + f_{\mathrm{B}} = 1.$$

$$S(\sigma) = \frac{1}{f_{\mathrm{A}} + \sigma^{-1} f_{\mathrm{B}}}.$$

Then as $\sigma \to \infty$, the speedup $S(\sigma)$

$$S(\infty) = \frac{1}{f_{\mathrm{A}}}$$