STA 141C - Big Data & High Performance Statistical Computing **Spring 2022**

Lecture 2: Introduction to statistical computing

Lecturer: Bo Y.-C. Ning *January 11, 2022*

Announcement

- Sign up Piazza; remote teaching continue to Jan 28, 2022
- Homework 1 posted, due January 26, 11:59pm
- Discussion sessions this week: A01 Numpy and the Jupyter Notebook and A02 RMarkdown
- Discussion sessions Zoom link: `https://ucdavis.zoom.us/j/92671467118`

Today

- Review of linear algebra (self-study)
- Introduction to scientific computing
- BLAS and LAPACK

1 Review of linear algebra (continue, self-study)

Useful book: *The Matrix Cookbook*: `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf`

1.1 Trace

A is a square matrix, $A \in \mathbb{R}^{n \times n}$

- $\text{tr}(A) = \sum_{i=1}^{n} a_{ii}$
- $\text{tr}(A + B) = \text{tr}(A) + \text{tr}(B)$
- $\text{tr}(\lambda A) = \lambda \text{tr}(A)$
- $\text{tr}(A') = \text{tr}(A)$
- In general, $\text{tr}(A_1 A_2 \dots A_k) = \text{tr}(A_k A_1 \dots A_{k-1}) = \text{tr}(A_{j+1} \dots A_k A_1 \dots A_j)$

1.2 Determinant

- If $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$, $\det(A) = a_{11}a_{22} - a_{12}a_{21}$.
- $\det(AB) = \det(A)\det(B)$ for $A, B \in \mathbb{R}^{n\times n}$
- $\det A^{-1} = 1/\det(A)$
- $\det(A) = \det(A')$ and, for a scalar c and $A \in \mathbb{R}^{n\times n}$, $\det(cA) = c^n\det(A)$
- The determinant of an upper or lower triangular matrix is the product of its diagonal elements.
- A is nonsingular (or positive definite) if and only if $\det(A) \neq 0$
- A is singular if and only if if $\det(A) = 0$
- If $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ is a block matrix, $A \in \mathbb{R}^{m\times m}$, $B \in \mathbb{R}^{m\times n}$, $C \in \mathbb{R}^{n\times m}$, and $D \in \mathbb{R}^{n\times n}$, then

$$\det(M) = \det(A - BD^{-1}C)\det(D) = \det(A)\det(D - CA^{-1}B)$$

1.3 Singular value decomposition (SVD)

For $A \in \mathbb{R}^{m\times n}$ and $p = \min\{m, n\}$

- Singular value decomposition: $A = U\Sigma V'$, where
  - $U = (u_1, \ldots, u_m) \in \mathbb{R}^{m\times m}$ is orthogonal
  - $V = (v_1, \ldots, v_p) \in \mathbb{R}^{n\times n}$ is orthogonal
  - $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m\times n}$, $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$
  - σ_i are called the singular values, u_i are the left singular vectors, and v_i are the right singular vectors.
- Thin (compact) SVD: assume $m \geq n$, $A = U\Sigma V'$, where
  - $U \in \mathbb{R}^{m\times n}$, $U'U = I_n$
  - $V \in \mathbb{R}^{n\times n}$, $V'V = I_n$
  - $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n) \in \mathbb{R}^{m\times n}$, $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$
- Relation to eigen-decomposition. Using thin SVD:

$$A'A = V\Sigma UU'\Sigma V' = V\Sigma^2 V'$$

$$AA' = U\Sigma VV'\Sigma U' = U\Sigma^2 U'$$

- Application: Principal component analysis (dimension deduction)
  - Principal components are eigenvectors of the covariance matrix (example: `https://arxiv.org/pdf/1708.00491.pdf`)

2 Introduction to statistical computing

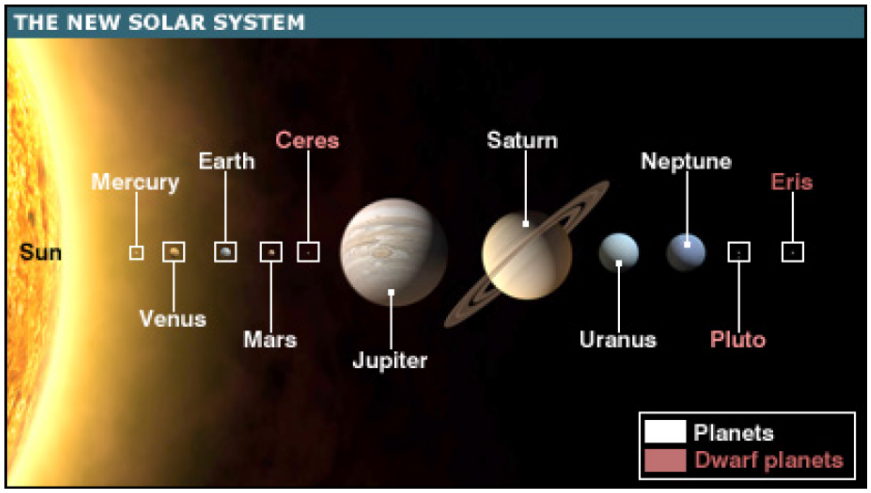
2.1 How Gauss became famous?

1801 Dr Carl Friedrich Gauss at the year 24 proved Fundamental Theorem of Algebra. He wrote the book *Disquisitiones Arithmetic*, which is still being studied today

1801 Jan 1 - Feb 11 (41 days), astronomer Piazzi observed Ceres (a dwarf planet), which was then lost behind sun

Carl Friedrich Gauss

Portrait of Gauss by Christian Albrecht Jensen (1840)



1801 From August to September, futile search by top astronomers; Laplace claimed it unsolvable

1801 From October to November, Gauss did calculations by method of least squares

1801 On December 31, astronomer von Zach relocated Ceres according to Gauss' calculation

1802 *Summarische Übersicht der Bestimmung der Bahnen der beiden neuen Hauptplaneten angewandten Methoden*, considered the origin of linear algebra

1807 Professor of Astronomy and Director of Göttingen Observatory in remainder of his life

1809 *Theoria motus corporum coelestium in sectionibus conicis solem ambientum* (Theory of motion of the celestial bodies moving in conic sections around the Sun); birth of the Gaussian (normal) distribution, as an attempt to rationalize the method of least squares

1810 Laplace consolidated importance of Gaussian distribution by proving the central limit theorem

1820 Gauss-Markov Theorem. Under Gaussian error assumption (actually only uncorrelated and homoscedastic needed), least square solution is the best linear unbiased estimate (BLUE), i.e., it has the smallest variance and thus MSE among all linear unbiased estimators.

So, how Gauss became famous?

- Motivated by a real problem.
- Heuristic solution: method of least squares.
- Solution readily verifiable: Ceres was re-discovered!
- Algorithmic development: linear algebra, Gaussian elimination, FFT (fast Fourier transform).
- Theoretical justification: Gaussian distribution, Gauss-Markov theorem.

References: Read Teets and Whitehead (1999)

2.2 Computer storage

Memory unit is the amount of data that can be stored in the storage unit. This storage capacity is expressed in terms of Bytes.

A binary digit (bit) is the minimum unit of binary (0 or 1) information stored in a computer system.

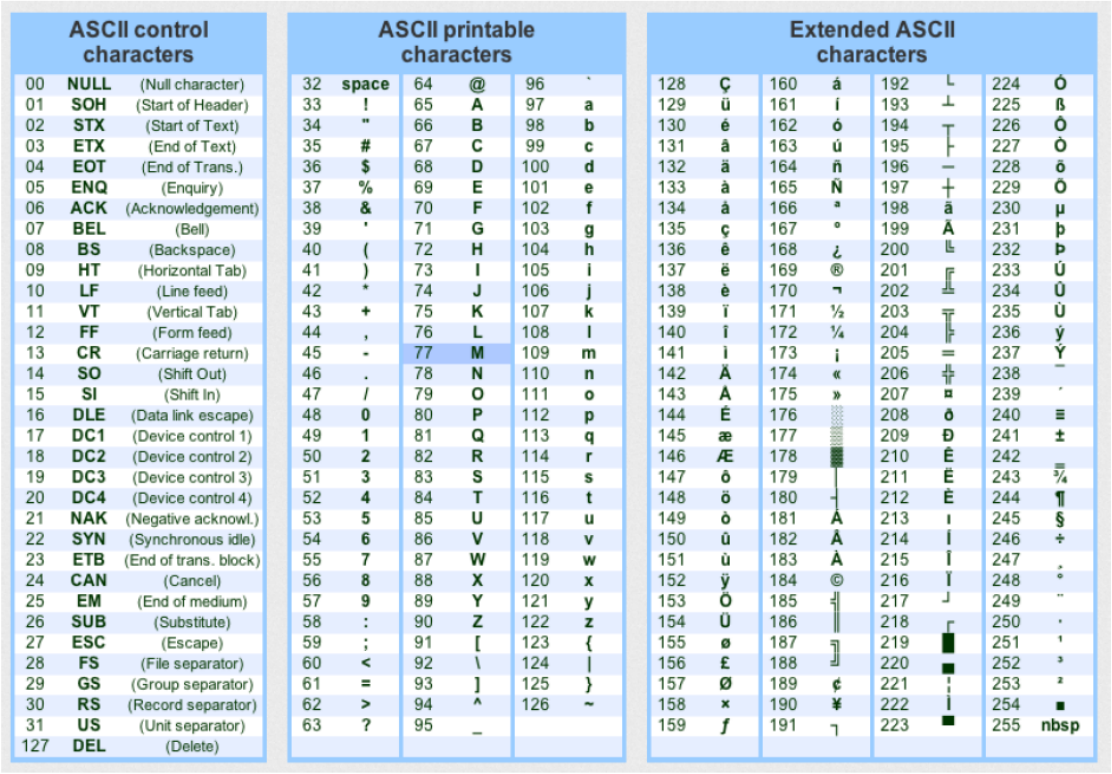
A group of 4 bits is called *nibble*.

A group of 8 bits is called *byte*. A byte is the smallest unit, which can represent a data item or a character.

- kB = kilobyte = 10^3 bytes
- MB = megabytes = 10^6 bytes
- GB = gigabytes = 10^9 bytes
- TB = terabytes = 10^{12} bytes (1TB, my computer)
- PB = petabytes = 10^{15} bytes

A computer system normally stores *characters* using the ASCII code. ASCII (American Code for Information Interchange) uses 7 bits, which only can store $2^7 = 128$ characters. Extended ASCII uses 8 bits with $2^8 = 256$ characters

For example: "STA" corresponds to 83 84 65 (Decimal number) = 01010011 01010100 01000001 (Binary number)



ASCII control characters

00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

2.3 Storing real numbers

There are two major approaches to store real numbers (i.e., numbers with fractional component) in modern computing: fixed-point number system and floating-point number system.

2.3.1 Fixed-point number system

In a *fixed-point number system*, there is a fixed number of digits after the decimal point. A fixed-point representation of a fractional number is essentially an *integer* that is to be implicitly multiplied by a fixed scaling factor. For example, the value 1.23 can be stored in a variable as the integer value 1230 with implicit scaling factor of 1/1000 (meaning that the last 3 decimal digits are implicitly assumed to be a decimal fraction).

For negative values, the number of bits and method of representing negative numbers vary from system to system. The integer type in both R and numpy has 32 bits. The first bit indicates sign: 0 for nonnegative numbers, 1 for negative numbers.

The range of representable integers by M-bit storage unit is $[-2^{M-1}, 2^{M-1}-1]$. Note that we don't need to represent 0 anymore so could have capacity for 2^{M-1} negative numbers). For example, for $M = 8$, the range is $[-128, 127]$; for $M = 16$, the range is $[-65536, 65535]$; and for $M = 32$, the range is $[-2147483648, 2147483647]$. Hence, the smallest integer in R is $-2^{31}+1 = -2147483647$. How to check this?

```
# Machine variable
> .Machine$integer.max
[1] 2147483647
> M <- 32
```

```
> big <- 2^{M-1}-1
> small <- - 2^{M-1}
> as.integer(big)
[1] 2147483647
> as.integer(big+1)
[1] NA
Warning message:
NAs introduced by coercion to integer range
> as.integer(small)
[1] NA
Warning message:
NAs introduced by coercion to integer range
> as.integer(small+1)
[1] -2147483647
```

What about in numpy? Try it by yourself.

Two's complement is the most common method of representing signed integers on computers, and more generally, fixed point binary values. To get the two's complement negative notation of an integer, you write out the number in binary. You then invert the digits, and add one to the result.

For example, suppose we're working with 8-bit quantities and we want to find how -28 would be expressed in two's complement notation. First we write out 28 in binary form as 00011100. Then we invert the digits to make 0 become 1 and 1 become 0. We obtain 11100011. Last, we add 1 to get 11100100 That is how one would write -28 in 8 bit binary. Generally speaking, an integer $-i$ in the interval $[-2^{M-1}, -1]$ would be represented by the same bit pattern by which the nonnegative integer $2^M - i$ is represented.

2.3.2 Floating-point number system

Floating-point number system is a computer model for real numbers. The IEEE Standard 754 floating point number is composed of three parts: the sign, exponent, and mantissa.

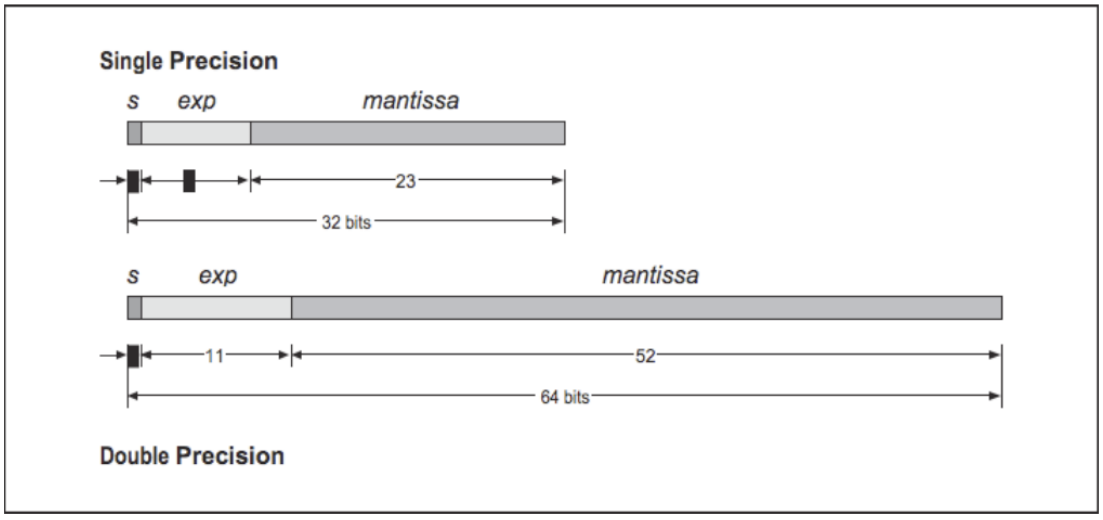
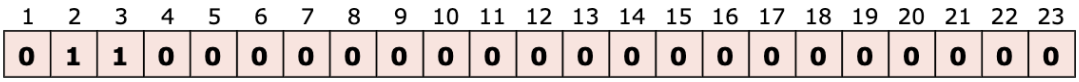


Figure 1.1: Single and double precisions of floating-point number system

The mantissa part is first convert into a decimal by summing all 2^{-n}s, where n is the position of a bit in the mantissa part that has a 1). For example, if the single precision,



The decimal part is $2^{-2} + 2^{-3} = 1/4 + 1/8 = 0.375$.

The formula for converting a single precision number into a decimal is given as

$$(-1)^s + (1 + \text{mantissa}) \times 2^{\exp -127}.$$

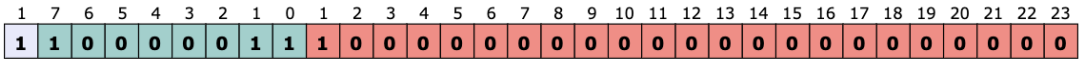


Figure 1.2: Single precision number of -24.0

For example, in Figure 1.2, we have $s = 1$, mantissa $= 2^{-1} = 0.5$, and $\exp = 2^7 + 2^1 + 2^0 = 128 + 2 + 1 = 131$, thus the real number is $(-1)^1 + (1 + 0.5) \times 2^{131-127} = -1.5 \times 2^4 = -24.0$.

To summarize:

- For the sign bit, 0 denotes a positive number, and 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.
- The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, to express an exponent of zero, 127 is stored in the exponent field. A stored value of 200 indicates an exponent of $200 - 127 = 73$. For reasons discussed later, exponents of -127 (all 0s) and $+128$ (all 1s) are reserved for special numbers. Double precision has an 11-bit exponent field, with a bias of 1023.
- The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit (left of the radix point) and the fraction bits (to the right of the radix point).

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number 50 can be represented as any of these:

$$0.050 \times 10^3$$
$$.5000 \times 10^2$$
$$5.000 \times 10^1$$
$$50.00 \times 100$$
$$5000. \times 10^{-2}$$

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, 50 is represented as 5.000×10^1.

Special numbers in floating-point system are stored as

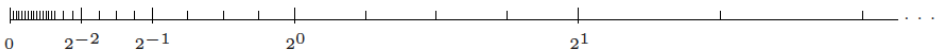


Fig. 2.4. The Floating-Point Number Line, Nonnegative Half

Fig. 2.5. The Floating-Point Number Line, Nonpositive Half

Fig. 2.6. The Floating-Point Number Line, Nonnegative Half; Another View

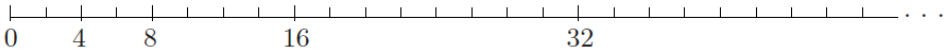


Fig. 2.7. The Fixed-Point Number Line, Nonnegative Half

- $\pm\infty$, sign 0 for positive and 1 for negative, exponents all 1, and mantissa part all 0
- NaN, sign either 0 or 1, exponents all 1, and mantissa anything except all 0

Take home messages:

- R and python uses double (64-bit) and 32-bit integer, they automatically convert between these two classes when needed for mathematical purposes
- Single precision: $\pm 10^{\pm 38}$ with precision up to 7 decimal digits
- Double precision: $\pm 10^{\pm 308}$ with precision up to 16 decimal digits
- All floating point numbers are rational because the mantissa has a fixed length. Irrational numbers stored in floating point are truncated into rational numbers, e.g., π
- The floating-point numbers do not occur uniformly over the real number line. (Further reading: [Here])
- Machine epsilon ϵ_M is defined as the distance (gap) between 1 and the next largest floating point number. In single precision, $\epsilon_M = 2^{-23}$; in double precision, $\epsilon_M = 2^{-52}$.

  One can check machine epsilon in numpy:

```
             1 = 1.00000000000000000000000
1 + epsilon_m = 1.00000011920928955078125
                ^                       ^
              2^0                   2^(-23)
```

Figure 1.3: Machine epsilon for single precision

```python
import numpy as np

# Single Precision
eps_single = np.finfo(np.float32).eps
print("Single precision machine eps = {}".format(eps_single))

# Double Precision
eps_double = np.finfo(np.float64).eps
print("Double precision machine eps = {}".format(eps_double))
```

2.4 Computer storage and catastrophic cancellation

When dealing with big data, one should be aware of the memory storage. For example, human genome has about 3×10^9 bases, each of which belongs to $\{A, C, T, G\}$. How much storage if we store 106 SNPs (single nucleotide polymorphisms) of 1000 individuals (1000 Genome Project) as characters (1GB), single (4GB), double (8GB), int32 (4GB), int16 (2GB), int8 (1GB), PLINK binary format 2bit/SNP (250MB)?

Overflow and *underflow* are both errors resulting from a shortage of space. *Overflow* indicates that we have done a calculation that resulted in a number larger than the largest number we can represent. For example,

```
> .Machine$integer.max + 1L
[1] NA
Warning message:
In .Machine$integer.max + 1L : NAs produced by integer overflow
> class(.Machine$integer.max + 1L)
[1] "integer"
> .Machine$integer.max + 1
[1] 2147483648
> class(.Machine$integer.max + 1)
[1] "numeric"
```

Underflow occurs when the true result of a floating point operation is smaller in magnitude (that is, closer to zero) than the smallest value representable as a normal floating point number. For example:

```
> log(exp(-745))
[1] -744.4401
> log(exp(-746))
[1] -Inf
> log(exp(-70))
[1] -70
```

In most situations, underflow is preferred over overflow. Overflow often causes crashes. Underflow is not a worry if the result is later added to a large number.

```
> 1 + exp(-746)
[1] 1
```

A take away message is "always try to add numbers of similar magnitude". Try to add small numbers together before adding larger ones, and try to add numbers of like magnitude together (paring). When all numbers are of same sign and similar magnitude, add in pairs so each stage the summands are of similar magnitude.

Catastrophic cancellation occurs when an operation on two numbers increases relative error substantially more than it increases absolute error. For example, the true value for $e^{-20} = 2.601\text{e}-09$. But if you evaluate using Taylor's expansion:

$$e^{-x} = 1 - x + \frac{x^2}{2} - \frac{x^3}{3!} + \dots,$$

then you get $6.138\text{e}-09$.

Example 2. $P(A^c) \neq 1 - P(A)$

```
> pnorm(2.5, lower.tail = FALSE)
[1] 0.006209665
> 1 - pnorm(2.5)
[1] 0.006209665
# Same thing, right? But the latter, shorter and simpler though it may seem, suffers from
    catastrophic cancellation.

> x <- 0:20
> data.frame(x, p1 = pnorm(x, lower.tail = FALSE), p2 = 1 - pnorm(x))
    x           p1           p2
1   0 5.000000e-01 5.000000e-01
2   1 1.586553e-01 1.586553e-01
3   2 2.275013e-02 2.275013e-02
4   3 1.349898e-03 1.349898e-03
5   4 3.167124e-05 3.167124e-05
6   5 2.866516e-07 2.866516e-07
7   6 9.865876e-10 9.865877e-10
8   7 1.279813e-12 1.279865e-12
9   8 6.220961e-16 6.661338e-16
10  9 1.128588e-19 0.000000e+00
11 10 7.619853e-24 0.000000e+00
12 11 1.910660e-28 0.000000e+00
13 12 1.776482e-33 0.000000e+00
14 13 6.117164e-39 0.000000e+00
15 14 7.793537e-45 0.000000e+00
16 15 3.670966e-51 0.000000e+00
17 16 6.388754e-58 0.000000e+00
18 17 4.105996e-65 0.000000e+00
19 18 9.740949e-73 0.000000e+00
20 19 8.527224e-81 0.000000e+00
21 20 2.753624e-89 0.000000e+00
```

One can use the symmetry of the normal distribution to compute these without catastrophic cancellation and without $\mathsf{lower.tail} = \mathsf{FALSE}$

```
> x <- 7:12
> data.frame(x, p1 = pnorm(x, lower.tail = FALSE), p2 = pnorm(- x))
   x           p1           p2
1  7 1.279813e-12 1.279813e-12
2  8 6.220961e-16 6.220961e-16
3  9 1.128588e-19 1.128588e-19
4 10 7.619853e-24 7.619853e-24
5 11 1.910660e-28 1.910660e-28
6 12 1.776482e-33 1.776482e-33
```

Example 3: Failure of using the short-cut formula for variance

To calculate the variance, some intro stats books call the "short-cut" formula for variance

$$\text{var}(X) = E(X^2) - E(X)^2,$$

but this serves as a perfect example to catastrophic cancellation.

Always use the two-pass algorithm

$$\bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad v_n = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x}_n)^2$$

```
> x <- 1:10
> # short cut
> mean(x^2) - mean(x)^2
[1] 8.25
> moo <- mean(x)
> mean((x - moo)^2)
[1] 8.25
```

So far so good. What if x is large?

```
> x <- x + 1e9
> # short cut
> mean(x^2) - mean(x)^2
[1] 0
> # two pass
> moo <- mean(x)
> mean((x - moo)^2)
[1] 8.25
```

More examples, see `https://www.stat.umn.edu/geyer/3701/notes/arithmetic.html`

2.5 Flops and Big-O notations

Algorithm is loosely defined as a set of instructions for doing something (Input $\rightarrow$ Output). Basic unit for measuring efficiency is flop (floating point operation). A flop consists of a floating point multiply (or divide) and the usually accompanying addition, fetch and store.

How to measure efficiency of an algorithm? We use Big-O notation. If n is the size of a problem, an algorithm has order $O(f(n))$ if, as $n \rightarrow \infty$, the number of computations $\rightarrow cf(n)$, where c is some constant that does not depend on n. More specifically, We say a function $f(n) = O(g(n))$ if $f(n) \leq ag(n)$, a is a constant. This means that $f(n)$ grows at most in the order of $g(n)$. For example,

- $2n^2 - n/2 - 1 = O(n^2)$
- $n = O(n)$, $n = O(n^2)$
- $n \neq O(n \log n)$
- The matrix-vector multiplication for $A \times b$, $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^n$ takes $O(mn)$ flops.
- The matrix-matrix multiplication for $A \times B$, $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$, it takes $O(mnp)$ flops.

Big-O notation is not the only notation being used to measure efficiency of an algorithm, there are other notations, such as Φ and Ω. We will introduce them during the discussion session next week.

Knowing how to count flops is important. Compare flops of the following two expressions:

$$G \times X \times y \qquad \text{and} \qquad G \times (X \times y)$$

for $G \in \mathbb{R}^{p \times p}$, $X \in \mathbb{R}^{p \times n}$, and $y \in \mathbb{R}^n$ We will get familiar with the flop counts for common numerical tasks in statistics.

2.6 Compiled and interpreted languages

Compiled languages are directly compiled to machine code that is executed by CPU. They are fast, take less memory, but also relatively longer development time, hard to debug. Examples include $C/C++$ and Fortran.

Interpreted languages by interpreter. They are fast for prototyping but can be excruciatingly slow for loops (we will talk about this later). Examples include R, Matlab, SAS, ...

Python is a mixed (complied and then interpreted by virtual machine) language. They are extremely convenient for data preprocessing and manipulation and need relatively short development time, but not as fast as compiled language (e.g., dealing with loops)

Here are two suggestions:

- To be versatile in dealing with big data, master at least one language in each category. Take advantage of the resource online
- Don't reinvent wheels. Make good use of libraries BLAS, LAPACK/LINPACK, SciPy, NumPy, ...

3 BLAS, LINPACK and LAPACK

3.1 BLAS

BLAS stands for basic linear algebra subroutines (see [this] for a complete listing of BLAS functions and [this] explains why it matters), which is the low?level part of your system that is responsible for efficiently performing numerical linear algebra.

```
# find out my BLAS library in numpy
import numpy as np
np.show_config()

blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/Users/yuchien/opt/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/Users/yuchien/opt/anaconda3/include']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/Users/yuchien/opt/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/Users/yuchien/opt/anaconda3/include']
lapack_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/Users/yuchien/opt/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/Users/yuchien/opt/anaconda3/include']
```

```
lapack_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/Users/yuchien/opt/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/Users/yuchien/opt/anaconda3/include']
```

R default use BLAS, see [tutorial] to install optimized BLAS/LAPACK libraries.

A flop serves as a basic unit of computation, which could denote one addition, subtraction, multiplication or division of floating point numbers. Note that, the flop count is just a rough measure of how expensive an algorithm can be.

Table 1.1: Flop counts for BLAS functions

Level	Operation	Name	Dimension	Flops
1	$\alpha \leftarrow x'y$	dot product	$x, y \in \mathbb{R}^n$	n
	$y \leftarrow y + ax$	saxpy	$a \in \mathbb{R}, x, y \in \mathbb{R}^n$	n
2	$y \rightarrow y + Ax$	gaxpy	$A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$	mn
	$A \leftarrow A + yx'$	rank one update	$A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$	mn
3	$C \leftarrow C + AB$	matrix multiplication	$A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{p\times n}$, $C \in \mathbb{R}^{m\times n}$	
	$A \leftarrow AD$	column scaling	$A \in \mathbb{R}^{m\times n}$, $D = diag(d_1, \ldots, d_n)$	mn
	$A \leftarrow DA$	row scaling	$A \in \mathbb{R}^{m\times n}$, $D = diag(d_1, \ldots, d_n)$	mn
	$E \leftarrow E^{-1}$	matrix inversion	$E \in \mathbb{R}^{n\times n}$	$O(n^3)$

Example: Computing $X'W^{-1}X$

There are different ways to compute $X'W^{-1}X$ (such as in the weighted least square), where $X \in \mathbb{R}^{n\times p}$ and $W = diag(w_1, \ldots, w_n) \in \mathbb{R}^{n\times n}$:

- $X' \times$ solve(or np.linalg.solve)$(W) \times X$: takes $O(n^3 + n^2p + np^2)$ flops, matrix inversion is expensive! (transpose requires no flops)
- $X' \times$ diag(or np.diag$(1/w) \times X$: takes $O(n^2p + np^2)$ flops, why?
- Can we improve?
- $X' \times (X/w)$: takes $O(np^2 + np) = O(np^2)$ flops
- In numpy, do X.T×(X/w) and in R, do crossprod(X, X/w): $O(np^2)$ flops

```
# clean up workspace
rm( list=ls() )
# slr design matrix X
X <- cbind( rep(1,6), (1:6) )
X

# weight vector w
w <- sqrt( (1:6) )          # just so they're different
w

# weight matrix
W <- diag(w)
# now (X'inv(W) X) four ways:
#   first          correct, but slow, W takes lots of space
W <- diag(w)
xwx1 <- t(X) %*% solve(W) %*% X
```

```
xwx1

#   second          correct, less slow, takes lots of space
xwx2 <- t(X) %*% diag(1/w) %*% X
xwx2

#   third           wrong, but faster -- recycles w wrong
xwx3 <- ( t(X) / w ) %*% X           # *****wrong*****
xwx3

#   fourth          correct, but looks wrong
xwx4 <- t(X) %*% (X/w)               # uses recycling correctly, fast
xwx4

#                   looks different but same execution
crossprod(X,X/w)                     # correct, fast

# done
rm( list=ls() )
```

Bottom line: Always be aware flop counts when writing code!

3.1.1 Vector computer and memory hierarchy

For high-performance matrix commutations, it is not enough to just minimize flops. Pipelining, effective use of memory hierarchy, data layout in memory,..., all play important roles. Most modern computers are vector machines, which perform vector calculations fast (e.g., saxpy, inner product).

One implication of the pipelining technology is that we need to ship vectors to the pipeline fast enough to keep the arithmetic units (ALU) busy and maintain high throughput. Taking vector addition $z = x + y$ as an example:

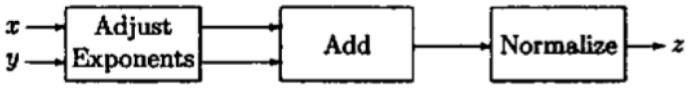


FIG. 1.4.1 *A 3-Cycle Adder*

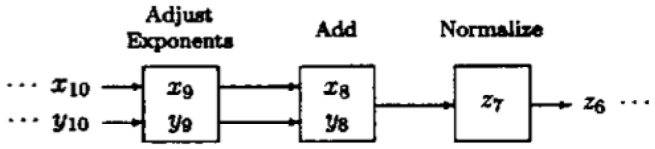


FIG. 1.4.2 *Pipelined Addition*

Knowing memory hierarchy is also important. Upper the hierarchy, faster the memory accessing speed, and more expensive the memory units. Key to high performance is effective use of memory hierarchy. Can we keep the super fast arithmetic units busy with enough deliveries of matrix data and ship the results to memory fast enough to avoid backlog?

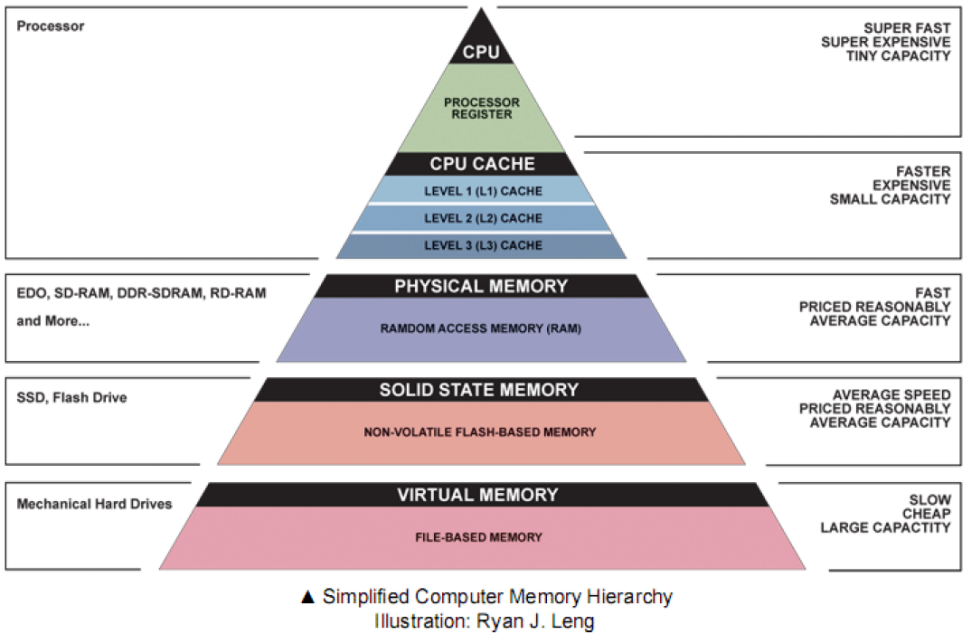


Figure 1.4: Computer memory hierarchy

Table 1.2: Using high-level BLAS

BLAS	Dimension	Mem ReFs	flops	ratio
Lv1: $y \leftarrow y + ax$	$x, y \in \mathbb{R}^n$	$3n$	n	$3:1$
Lv2: $y \leftarrow y + Ax$	$x, y \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$	n^2	n^2	$1:1$
Lv3: $C \leftarrow C + AB$	$A, B, C \in \mathbb{R}^{n \times n}$	$4n^2$	n^3	$4:n$

The answer is to use high-level BLAS as much as possible to keep the super fast arithmetic units busy with enough deliveries of matrix data and ship the results to memory fast enough to avoid backlog.

BLAS 1 tend to be memory bandwidth-limited. Higher level BLAS (3 or 2) make more effective use of ALUs (keep them busy).

In addition to BLAS, LINPACK (**Lin**ear Equations Software **Pack**age) and LAPACK (**L**inear **A**lgebra **Pack**age) are both software packages that provide routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems A distinction between LAPACK and LINPACK is that LAPACK makes use of higher level BLAS as much as possible (usually by smart partitioning) to increase the so-called level-3 fraction. LINPACK makes use of the BLAS (Basic Linear Algebra Subprograms) libraries for performing basic vector and matrix operations. `numpy` and `R` automatically uses LAPACK and BLAS libraries.

References

Teets, D. and K. Whitehead (1999). The discovery of ceres: how gauss became famous. *Math. Mag. 72*(2), 83–93.