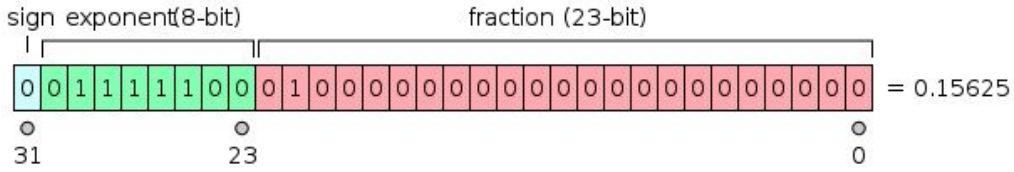
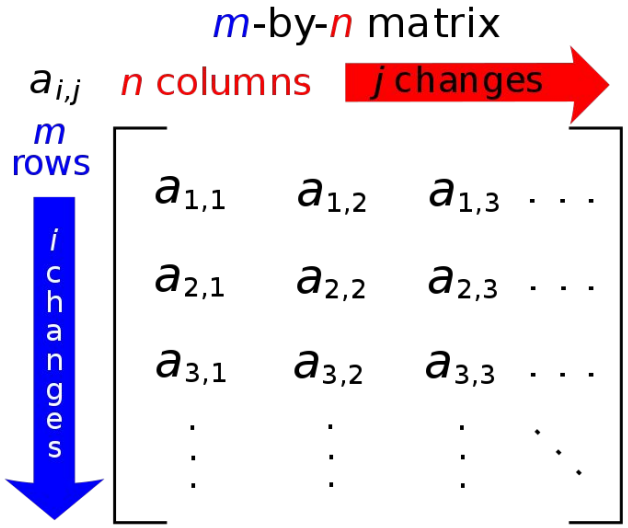
计算物理 第一部分

第2讲　数值计算基础+线性代数回顾





李强　北京大学物理学院西楼227

qliphy0@pku.edu.cn，15210033542

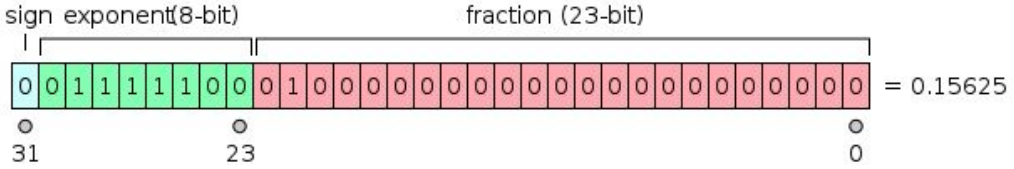
计算机表达的整数

由于计算机都只有有限的内存，因此它只能够表达有限多的数。这不仅仅包括实数，也包括整数。目前经典计算机本质上都是通过记录二进制的数来记录所有的数的。也就是说，最基本的单元是所谓的位(bit)，也就是比特，它是binary digit的缩写，它只有两个可能取值0和1。**每8个位构成一个字节(byte)**。

通常的整数由4个byte构成，也就是说是32位。当然，如果嫌这个不够，也可以用64位(也就是8个byte)。例如**在C语言之中，int型的整数就是4个字节而所谓的long int型的整数则是8个字节**。对任何一个4字节(32bit)的非负整数都可以在二进制中表达为，

$$n = \sum_{i=0}^{31} b_i 2^i$$

正整数的范围因此从0到2^32-1。**如果我们还希望表达负整数，我们只好牺牲一位来标记整数的正负，真正用于记录数字的位数减少到31位**。



An Example: IEEE 754 to Number

In: 110000000 01010000 00000000 0000000

- Step 1: Split into IEEE 754 components

- Step 2 (Sign): 0 is positive, 1 is negative. **Sign is ...?**
- Step 3 (exponent): 128 (**10000000**)-127 (bias)=1
- Step 4 (mantissa): **1.01010000...** = 1.3125
  ($1x2^{-1}$=.5 **$1x2^{-2}$=.25** $1x2^{-3}$=.125 **$1x2^{-4}$=.0625**)
- Step 5 (result): $(-1)^1 \times 1.3125 \times 2^1 = -2.625$

Werner Buchholz (24 October 1922 – 11 July 2019) was a German-American computer scientist. After growing up in Europe, Buchholz moved to Canada and then to the United States. He worked for International Business Machines (IBM) in New York. In June 1956, he coined the term "byte" for a unit of digital information.[1][2][3] In 1990, he was recognized as a computer pioneer by the Institute of Electrical and Electronics Engineers.

Werner Buchholz

German computer scientist

Werner Buchholz was a German-American computer scientist. After growing up in Europe, Buchholz moved to Canada and then to the United States. He worked for International Business Machines in New York. In June 1956, he coined the term "byte" for a unit of digital information. Wikipedia

Born: 24 October 1922 (age 98 years), Detmold, Germany

Awards: IEEE Computer Society Awards - Computer Pioneer Award

浮点数、机器精度、舍入误差

对于实数，一般采用所谓的浮点数来表示。显然，由于实数是无穷多数构成的连续集合，因此在**计算机中能够表达的实数只能是其中的一个有限的部分。这就是浮点数系统。它保证了在任何一个实数的足够接近的邻域内总能找到一个浮点数来近似代表相应的实数。**

计算机中可以表达出来的那些数称为可表达的实数，这个集合的大小取决于我们选择用多少位来表达一个实数。**所有的那些不可表达的实数必须经过舍入的操作**，用它附近的一个可表达的实数来近似代替，相应产生的误差一般被称为舍入误差。

机器上能够表达的**最小**的**正实数**：**机器精度**

⊕ 表示计算机中实现的“加法”

$$\epsilon_M = \min\{g \in A | 1 \oplus g > 1, g > 0\}$$

机器精度的数值依赖于我们在一个实数上愿意投入多少内存。例如，对于通常的单精度的实数(4字节)，~10^{-7}，而对于双精度的实数(8字节)来说，~10^{-16}。

浮点数

选取一个正的自然数β为底(通常β=2)，那么一般实数x在以为β底的系统中可以用下列**整数部分和小数部分分别具有(n+1)位和m位**的小数表达:

$$x_\beta = (-)^s \left[x_n x_{n-1} \cdots x_1 x_0 . x_{-1} x_{-2} \cdots x_{-m}\right], \quad x_n \neq 0$$

$$= (-)^s \left[\sum_{k=-m}^{n} x_k \beta^k\right]$$

其中s=0,1称为符号位，用以标记该数是正还是负。更为经济与方便的计数方法是将实数提出一个公共的因子，实现浮点数表示:

$$x_\beta = (-)^s \cdot (0.a_1 a_2 \cdots a_t) \cdot \beta^e = (-)^s \cdot m \cdot \beta^{e-t}$$

这个表示称为实数的浮点数表示，其中的 s 仍然是符号位，整数 $m = a_1 a_2 \cdots a_t$ 是一个 t 位的整数称为尾数，显然 $0 \leq m \leq \beta^t - 1$；整数 e 则称为指数，它满足 $L \leq e \leq U$，一般选取 $L < 0$ 为一个负整数，$U > 0$ 为一个正整数。于是，对于一个占用 N 位的实数而言，计算机中存储时会让符号位占一位，有效数字 a_i，$i = 1, \cdots, t$ 占 t 位，指数 e 则占据剩下的 $N-t-1$ 位。

浮点数：β,t,L,U

常见的情形是单精度的浮点数(占32位)和双精度的浮点数(占64 位)，它们除了符号位占一位之外，分别具有**t=23和t=52位**来存储有效数字，而存储指数的位数则分别是**8和11位**。一般来说数字 **0会单独有一个特别的表示**。因此，一个一般的浮点数系统F可以用它的基底β、尾数的位数t、指数e的上下限L和U标记为

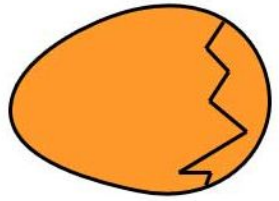
$$F(\beta,t,L,U)=\{0\}\cup\left[x\in R:\ x=(-)^s\beta^e\sum_{k=1}^{t}a_k\beta^{-k}\right]$$

由于我们总是可以移动小数点并相应地改变指数 e 的数值，因此为了保证数字表示的唯一性，**一般假设 a1 != 0**，否则我们可以向右移动小数点一位并减少 e 一个单位即可。经过这样约定的浮点数表示称为**规范化的浮点数表达**。

$F(10,4,-1,4)$ 系统中，实数 1 可以有下列四种表达，

$$0.1000\cdot 10^1,\quad 0.0100\cdot 10^2,\quad 0.0010\cdot 10^3,\quad 0.0001\cdot 10^4$$

浮点数：特殊字符及存储

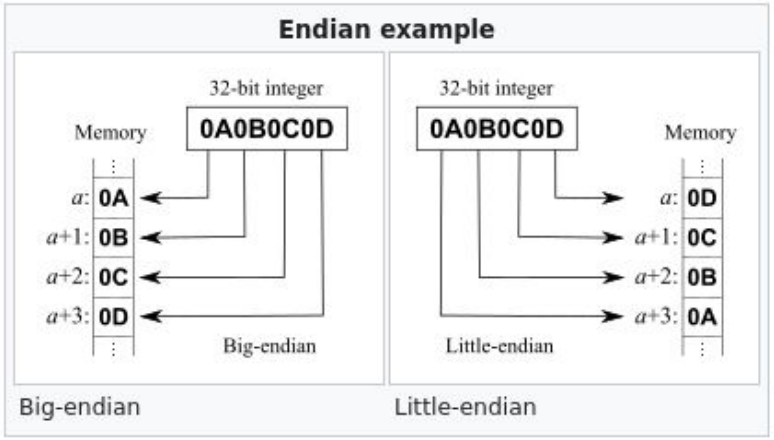


BIG ENDIAN - The way people always broke their eggs in the Lilliput land

除了 0 这个特殊字符之外，还有另外两个特殊字符

- Inf: 表示无穷，由有限数字除以0 产生，比如1/0，
- NaN: 意思是 Not a number，由没有定义或者不确定的操作产生，比方说 0/0，0*Inf 或者 Inf/Inf。

数据存放在内存或数据文件里，是顺序存放数据信息还是倒序存放，被分为**big endian**(低地址存放最高有效字节)和**little endian**(低地址存放最低有效字节)。字节序的问题，牵涉到CPU的两大派系。那就是PowerPC系列的CPU采用big endian方式存储数据，Intel的x86系列CPU则采用little endian方式存储数据。




```
#include <bits/stdc++.h>
using namespace std;
int main()
{   unsigned int i = 1;
    char *c = (char*)&i;
    if (*c)
        cout<<"Little endian";
    else
        cout<<"Big endian";
    return 0;
}
```

In the program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is dereferenced it will contain only first byte of integer. If machine is little endian then *c will be 1 (because last byte is stored first) and if machine is big endian then *c will be 0.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
    unsigned int i = 1;
//     unsigned int i = pow(2,9);
//  unsigned int i = 1;
   char *c = (char*) &i;
   if(*c)
      cout<<"Little endian "<<int(*c)<<endl;
   else
      cout<<"Big endian "<<int(*c)<<endl;
   return 0;
}
```

```
qliphy@qliphy:~/Desktop/Teaching23/2023CP/NEW$ g++ -w -o endian endian.c
qliphy@qliphy:~/Desktop/Teaching23/2023CP/NEW$ ./endian
Little endian 1
```

浮点数范围

规范化表达的浮点数系统F(β;t;L;U)中的所有可表达的实数的**绝对值一定介于**xmin和xmax之间。

$$e = U, \quad 0.a_1a_2\cdots a_t = 0.(\beta-1)\cdots(\beta-1) = 1-\beta^{-t}, \quad \Rightarrow \quad x_{max} = \beta^U(1-\beta^{-t})$$

$$e = L, \quad 0.a_1a_2\cdots a_t = 0.1, \quad \Rightarrow \quad x_{\min} = \beta^{L-1}$$

$$x_{\min} \equiv \beta^{L-1} \le |x| \le \beta^U(1-\beta^{-t}) \equiv x_{max}$$

为了能够表达更多 (特别是绝对值更小) 的数，我们会在 $e = L$ 时的放弃要求 $a_1 \neq 0$(由于 $e = L$，因此这不会破坏数表示的唯一性，只会加入一系列更小的数) 。这可以产生在 $(-\beta^{L-1}, \beta^{L-1})$ 中的实数。这样的操作叫做次规范化，使得最小绝对值下降为 β^{L-t}。如果比这个还要小的数出现，机器会产生所谓的下溢。虽然次规范化增加了所表示数的范围，但新增加的数的精度要低于其他浮点数，因为它们的有效数字较少。

机器精度及浮点数分布

$$\epsilon_M = \min\{g \in A | 1 \oplus g > 1, g > 0\}$$

1 可以写成 `0.1.....0 * β`

$$0.10000 \times \beta^1$$

最小的 g 应该为

$$\beta^{-t} \times \beta^1 = \beta^{1-t}$$

所以我们得到 $\epsilon_M = \beta^{1-t}$。对于二进制来说，单精度数据，如果取 t=23，那么 $\epsilon_M \sim 10^{-7}$；双精度数据，如果 t=52，那么 $\epsilon_M \sim 10^{-16}$。

浮点数分布不是连续的，也不是均匀的。假设x与y是最临近的：

$$x = (-)^s \cdot (0.a_1a_2 \cdots a_t) \cdot \beta^e$$
$$y = (-)^s \cdot (0.a_1a_2 \cdots (a_t + 1)) \cdot \beta^e$$

$$\frac{|y-x|}{|x|} = \frac{\beta^{e-t}}{|x|} = \epsilon_M \frac{\beta^{e-1}}{|x|};$$

因为 $|x|$ 的范围为 $\beta^{e-1} \le |x| < \beta^e$，所以两者最近的距离一定介于 $\beta^{-1}\epsilon_M|x|$ 和 $\epsilon_M|x|$ 之间。

对于浮点数系统F(β;t;L;U)，β代表的是二进制或N进制，t也就是尾数的位数代表数据能达到的精度，L和U代表了这套浮点数系统所表达的数的上限和下限。

offset

"Father" of the IEEE 754 standard

- 1970年代后期, IEEE成立委员会着手制定浮点数标准，1985年完成浮点数标准IEEE754的制定，现在绝大多数计算机都采用IEEE754来表示浮点数。目前使用的标准是：IEEE std 754-2008。 **IEEE 754-2019**
- UC Berkeley 数学教授威廉·卡亨(William Kahan)领衔IEEE754标准制定，获得了1989年图灵奖（A.M. Turing Award，又译"杜林奖"）。

WILLIAM ("VELVEL") MORTON KAHAN

United States – 1989

CITATION

For his fundamental contributions to numerical analysis. One of the foremost experts on floating-point computations. Kahan has dedicated himself to "making the world safe for numerical computations"!

A.M. TURING AWARD

舍入

对于**那些不在系统F中的实数**怎么办呢？计算机必须能够处理它们。最常见的操作称为舍入。这个操作是从一个一般的**实数x∈R到某个浮点数系统F的映射**

$$fl(x) = (-)^s(0.a_1a_2\cdots\tilde{a}_t)\cdot\beta^e, \quad \tilde{a}_t = \begin{cases} a_t, & a_{t+1} < \beta/2 \\ a_t + 1, & a_{t+1} \geq \beta/2. \end{cases}$$

- 采用这样的处理之后，我们能够表达的实数范围大大扩大了。绝对值过大的实数仍然无法在计算机中表达。
- 具体来说，如果$|x|>x_{max}$，那么我们无法将其舍入到任何一个合理的可表达的实数。这时候计算机会产生一个**溢出**，程序的运行一般也会终止。
- 类似的，如果某个实数的绝对值过小，$|x|<x_{min}$，这时候产生的一般称为**下溢**，此时往往系统会将其舍入为 0。
- 当然，除了四舍五入之外，还有一种舍入称为**截断舍入**，就是将从a_{t+1}位开始往后的尾数都截去。

浮点数的误差

对于任意$x \in R$，且满足$x_{min}<|x|<x_{max}$，我们一定有

$$fl(x) = x(1+\delta), \quad |\delta| \leq \frac{1}{2}\epsilon_M = \frac{1}{2}\beta^{1-t}$$

因此我们有时候又称**$\varepsilon_M/2$为舍入误差单位**。这意味着一个实数与它的舍入值之间的误差是完全在控制之中的:

$$E_{\text{rel}}(x) = \frac{|x - fl(x)|}{|x|} \leq \frac{1}{2}\epsilon_M, \quad E(x) = |x - fl(x)| \leq \frac{1}{2}\beta^{e-t}$$

从这个角度来说，**浮点数的表示以及运算都是有误差的**。浮点数比整数更为糟糕的是，即使是在其定义域内，浮点数也仅仅是某个实数的近似表示而不是严格表示，而且两个实数之间的运算的结果，即使它不超出定义域，通常也不是一个可表达的实数，需要将其经过舍入过程转换为一个可表达的实数。

浮点数的误差

一般来说，**由于舍入问题的存在，计算机中的基本运算比如，加减法、乘法等等，并不满足原先数学中的结合律、分配率等基本规律。**

例如，对于三个计算机中可表达的数 a,b,c∈A，在严格的数学中：

(a + b) + c = a + (b + c) = a + b + c,

但是在机器的运算中：

(a ⊕ b) ⊕ c 和 a ⊕ (b ⊕ c)一般是不相等的。

但是，舍入方法的实现应当尽量确保在一次基本运算后造成的舍入误差仅仅在一两个机器精度的范围之内。即使这一点得到满足，由于计算机在复杂的算法中往往需要进行**多次的计算**，因此我们**必须考虑这种误差的传递和累计效应**。这使得我们在数值计算中对算法的稳定性的要求提高了。

误差分析

考虑一个简单的计算问题：函数求值。假设有函数f：R→R，x为函数输入参数的准确值，准确结果为f(x)。
由于数据误差，**实际使用输入值为^x**。同时计算过程也存在近似，使得最后结果为^f(^x)，其中，**^f代表计算过程中的近似**，因此，总误差为

$$\hat{f}(\hat{x}) - f(x) = [\hat{f}(\hat{x}) - f(\hat{x})] + [f(\hat{x}) - f(x)]$$

- 其中第一项为输入同为^x时计算过程的误差，它是一个**单纯的计算误差**。这种误差**包括截断误差和我们之前提到过的舍入误差**。

- 第二项是由于输入数据误差经过精确的函数求值过程产生的误差，即**数据传递误差**。数据传递误差是由问题的**敏感性**决定的。

(1)计算误差

打个比方说，我们要计算的函数是以n阶泰勒级数展开的，我们不可能计算无穷阶级数，如果级数展开收敛性够好，那么在某一处给予截断就好。下面我们来看一个**求一阶导数的差分近似**的例子，来看一下截断误差和舍入误差分别是如何起作用的。

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

$$f(x+h) = f(x) + h \cdot f'(x) + \frac{h^2}{2} f''(\xi), \quad \xi \in [x, x+h]$$

假设M是|f″(x)|的上限，那么这个一阶差分的截断误差为Mh/2。我们假设，计算函数f(x)的过程中，舍入误差为ε，因为在差分中函数被算了两次，因此舍入误差最后进入到f′(x)中的大小为2ε/h。所以总的误差为

$$\varepsilon_{\text{tot}} = \frac{Mh}{2} + \frac{2\varepsilon}{h}$$

所以步长h并不是越大越好，也不是越小越好；大了，截断误差就会比较大，小了，舍入误差会被 1/h放大。

(2)数据传递误差

数据传递误差是由问题的敏感性决定的。我们定义问题的敏感性，是指输入数据的扰动对问题解的影响程度的大小。**敏感性有时候也称问题的病态性。**如果输入数据的相对变化引起解的相对变化不是很大，则称这个问题是不敏感的或者良态(well-conditioned)；反之，如果解的相对变化远远超过输入数据的变化，则称这个问题是敏感，或者病态的(ill-conditioned)。

$$\text{cond} = \frac{||\text{问题的解的相对变化量}||}{||\text{输入数据的相对变化量}||}$$

这里的算符||.||表示范数，它用于度量一个量的大小。对于一个实数或复数来讲，范数就是这个数的模。对于其他类型的量，比如说向量、矩阵和函数，范数的具体定义我们在后面会陆续讲到。**以函数求值问题为例**，其条件数有下面的计算公式

$$\text{cond} = \left| \frac{[f(\hat{x}) - f(x)]/f(x)}{(\hat{x} - x)/x} \right|$$

(2)数据传递误差

输入数据扰动和引起解的相对误差之间满足一个近似的不等式

$$||\text{数据传递的相对误差}|| \leq \text{cond} \times ||\text{输入数据的相对变化}||$$

$$f(\hat{x}) - f(x) \approx f'(x)(\hat{x} - x)$$

$$\text{cond} = \left| \frac{[f(\hat{x}) - f(x)]/f(x)}{(\hat{x} - x)/x} \right| \approx \left| \frac{xf'(x)}{f(x)} \right|$$

例题，用条件数估计计算f(x)=tanx的问题敏感性。因为f′(x)= 1+tan^2(x)，有

$$\text{cond} = \left| \frac{xf'(x)}{f(x)} \right| = \left| \frac{x}{\sin x \cos x} \right|$$

当 x 接近于 $\pi/2$ 的时候，比方说 $\pi/2 \approx 1.570796$，如果我们在 $x = 1.57079$ 附近计算，条件数大概是 2.5×10^5。因此在 $x = \pi/2$ 附近计算 $\tan x$ 是个很敏感的问题。我们可以变换问题形式，以改变敏感性，比方说，计算 $\tan x - (\pi/2 - x)^{-1}$ 的值，条件数就会缩小。

Wilkinson 1960年代给出的例子

$$w(x) = \prod_{i=1}^{20}(x-i) = (x-1)(x-2)\cdots(x-20)$$

$$\begin{aligned} w(x) = x^{20} &- 210x^{19} + 20615x^{18} - 1256850x^{17} + 53327946x^{16} \\ &- 1672280820x^{15} + 40171771630x^{14} - 756111184500x^{13} \\ &+ 11310276995381x^{12} - 135585182899530x^{11} \\ &+ 1307535010540395x^{10} - 10142299865511450x^{9} \\ &+ 63030812099294896x^{8} - 311333643161390640x^{7} \\ &+ 1206647803780373360x^{6} - 3599979517947607200x^{5} \\ &+ 8037811822645051776x^{4} - 12870931245150988800x^{3} \\ &+ 13803759753640704000x^{2} - 8752948036761600000x \\ &+ 2432902008176640000. \end{aligned}$$

If the coefficient of x^{19} is decreased from -210 by 2^{-23} to -210.0000001192,

1.00000	2.00000	3.00000	4.00000	5.00000
6.00001	6.99970	8.00727	8.91725	20.84691
$10.09527 \pm 0.64350i$	$11.79363 \pm 1.65233i$	$13.99236 \pm 2.51883i$	$16.73074 \pm 2.81262i$	$19.50244 \pm 1.94033i$

Wilkinson 1960年代给出的例子

如何理解上述小扰动对方程根的影响？ 我们设扰动量为r，则扰动后的方程为$P(x,r)=P(x)+rx^{19}=0$，$P(x,r)$的零点均为r的函数，记为$x_i(r)$，$i=1,2,\cdots,20$。我们需要研究$x_i(r)$关于r的变化情况。条件数$C_i=\frac{|x_i(r)-x_i(0)|}{|r-0|}\approx|x_i'(0)|$。根据$P(x)+rx^{19}=[x-x_i(r)]\prod_{j=1,j\neq i}^{20}[x-x_j(r)]$，两边对$r$求导，并令$r\to 0$，可得：$x^{19}=-\frac{dx_i(0)}{dr}\prod_{j=1,j\neq i}^{20}(x-j)+(x-i)\left\{\frac{d}{dr}\prod_{j=1,j\neq i}^{20}[x-x_j(r)]\right\}_{r=0}$。最后令$x\to i$，得到：$i^{19}=-\frac{dx_i(0)}{dr}\prod_{j=1,j\neq i}^{20}(i-j)$，即：$\frac{dx_i(0)}{dr}=-\frac{i^{19}}{\prod_{j=1,j\neq i}^{20}(i-j)}$，$i=1,2,\cdots,20$。计算表明，$C_1\approx 8.2\times 10^{-18}$，而$C_7\approx 2.5\times 10^3$，$C_{16}=2.4\times 10^9$达到最大，必定造成病态。根据$|x_i(r)-x_i(0)|\approx|rx_i'(0)|=r|x_i'(0)|$便可估算扰动$r=2^{-23}$对各个根的影响了。

Wilkinson 1960年代给出的例子

JAMES HARDY ("JIM") WILKINSON

United Kingdom – 1970

CITATION

For his research in numerical analysis to facilitiate the use of the high-speed digital computer, having received special recognition for his work in computations in linear algebra and "backward" error analysis.

Jim Wilkinson

Jim Wilkinson with his Turing Award

Born	James Hardy Wilkinson 27 September 1919 Strood, England
Died	5 October 1986 (aged 67) Teddington, England
Nationality	English
Alma mater	Trinity College, Cambridge
Known for	Wilkinson matrix Wilkinson's polynomial
Awards	Chauvenet Prize (1987) ACM Turing Award (1970) FRS (1969)[1]

```
from sympy import *
from scipy import *
from scipy . linalg import eig
x = symbols ('x')
#http://cstl-csm.semo.edu/jwojdylo/MA345/Chapter6/polyroot/polyroot.pdf
def p ( x ) :
  """
  Wilkinson polynomial
  """
  p = 1
  for i in range (1 , 21 ) :
     p *= ( x - i )
  return p
p( 1 ) # a numeric value
p( x ) # a symbolic value
p( 15 )
pp = expand ( p ( x ))
coeff = [ pp . coeff (x , i ) for i in range (0 , 21 ) ]
print ('Coefficients \n', coeff, '\n')
```

```
def comp_matrix ( coeff ) :
  n = len ( array ( coeff )) - 1
  c = zeros (( n , n ))
  c [ : ,- 1 ] = - array ( coeff [ : - 1 ] )
  c = c + diag ( ones (( n -1 ,)) , - 1 )
  return c

c = comp_matrix ( coeff )
print ('The roots of the polynomial \n', sort ( eig(c)[0] ), '\n')

c15 = coeff [ 15 ]
coeff [ 15 ] += 0.001 # perturbation
rel_in_error = abs (0.001/c15 )
c = comp_matrix ( coeff )
print ('The new roots of the polynomial \n', sort ( eig(c)[0] ), '\n')
newroot15 = sort ( eig(c)[0] ) [ 14 ]
rel_out_error = abs ( newroot15 - 15 )
condition_15 = rel_out_error / rel_in_error
print(newroot15, rel_out_error, rel_in_error, condition_15)
~
```

```
Coefficients
 [2432902008176640000, -8752948036761600000, 13803759753640704000, -12870931245150988800, 8037811822645
051776, -3599979517947607200, 1206647803780373360, -311333643161390640, 63030812099294896, -10142299865
511450, 1307535010540395, -135585182899530, 11310276995381, -756111184500, 40171771630, -1672280820, 53
327946, -1256850, 20615, -210, 1]

The roots of the polynomial
 [ 1.        +0.j          2.        +0.j          2.99999999+0.j
  4.0000002 +0.j          4.9999963 +0.j          6.00004844+0.j
  6.99955763+0.j          8.00289107+0.j          8.98669304+0.j
 10.04997404+0.j         10.88601694+0.j         12.35865752+0.j
 12.56119339+0.j         14.51895931-0.21330456j 14.51895931+0.21330456j
 16.20679459+0.j         16.88571669+0.j         18.03009727+0.j
 18.99390218+0.j         20.00054209+0.j        ]

The new roots of the polynomial
 [ 1.        +0.j          2.        +0.j          2.99999999+0.j
  4.00000011+0.j          5.00000014+0.j          5.99995304+0.j
  7.00113615+0.j          7.98540872+0.j          9.20693234+0.j
  9.48112626+0.j         11.09221408-1.04340996j 11.09221408+1.04340996j
 13.17470847-1.73164384j 13.17470847+1.73164384j 15.58763052-1.95799431j
 15.58763052+1.95799431j 17.9918351 -1.46534085j 17.9918351 +1.46534085j
 19.81633345-0.36062685j 19.81633345+0.36062685j]

(15.587630522476498-1.9579943117145864j) 2.044272818303046 5.97985689987164e-13 3418598224895.53
```

舍入误差与算法的稳定性

既然**计算机中的实数都是利用不精确的浮点数来表达的**，我们特别需要讨论算法的稳定性，或者说初始值中可能的误差，随着算法的运行是如何传递到最终的结果中去的。

既然每次计算机的操作往往都伴随着舍入误差，我们就需要分析这个误差是如何随着算法的发展而传递的。以一个简单的例子来说，假定我们需要计算三个实数a, b, c的和，即a+b+c，我们有两个算法来进行这个计算，即 **(a+b)+c** 和 **a+(b+c)**。

$$a \oplus b = (a+b)(1+\varepsilon_1), \quad (a \oplus b) \oplus c = [(a+b)(1+\varepsilon_1)+c](1+\varepsilon_2)$$

$$= (a+b+c)\left[1+\frac{a+b}{a+b+c}\varepsilon_1+\varepsilon_2\right]$$

忽略二阶无穷小量

$$(b \oplus c) \oplus a = (a+b+c)\left[1+\frac{c+b}{a+b+c}\varepsilon_1+\varepsilon_2\right]$$

部分的放大因子分别为(a+b)/(a+b+c)和(b+c)/(a+b+c)。注意，这两个因子可以相差很远: |(a+b)/(b+c)|。

如 a+b~0，那么先进行这两个数的加法的计算是更好的。

病态性、稳定性、准确性

算法稳定性反映了计算过程中扰动对计算结果的影响程度的大小：

- 若计算结果对计算过程中舍入误差不敏感，则相应算法为稳定的，
- 对于包含一系列步骤的计算过程，若计算中小扰动不被放大或者放大不严重，则相应的算法为稳定的。

要保证一个数值计算问题结果的准确性，通常要依次考虑三点

- **病态性**：这是与待求解数学问题的性质有关，**与具体算法无关**，需要最先考虑，
- **稳定性**：这是数值算法的性质，应尽量选择稳定性好的算法，减少计算中误差的扩大，
- **通过定性分析控制舍入误差**，尽量避免下面例子里出现的各种问题，如果内存允许，可以考虑采用位数较多的高精度浮点数。

例1：避免大数相消现象的出现

浮点数系统的一个重要问题是存在抵消现象，也就是两个符号相同、值相近的p位数相减可能使结果的有效数字远小于p位。
比如两个十进制的6位精度数，x=1.92305×10^3 和 y = 1.92137×10^3，则 x-y=1.68，这一步减法计算过程中**未发生舍入，但它的结果却只有3位有效数字**。
舍入是丢弃末尾数位上的数字，而抵消丢失的是高位数字包含的信息，有时候危害更大。

当x<0，且|x|较大时，利用公式 $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$
截断前n项计算，会发生严重的抵消，使误差很大。假设x=-20

$S_{96}(x) = 5.62188 \times 10^{-9}$，此时下一步加的项为 $x^{96}/96! = 7.98930 \times 10^{-26}$，它与 $S_{96}(x)$ 的比值已经小于 $\varepsilon_M/2$，因此后续的计算都不会改变部分和的计算值，而 e^{-20} 的准确值为 2.061×10^{-9}，说明计算结果完全错误。

这里需要说明的是，当x>0时，上述求和式子中每项都大于0，不会有抵消现象，计算是稳定的。**对于x<0的情况，通过公式 1/e^{-x} 来计算e^x是有效、可行的算法。**

例2:避免出现上溢或下溢

$$y = \frac{x_1}{x_2 \cdot x_3 \cdot \cdots \cdot x_n}$$

其中x2比x1小很多，|x1/x2|>3.404×10^38，那么采用单精度浮点数计算x1/x2就会发生上溢。
一般情况下，y的准确值不会超过上溢值。**为了避免上溢，应先计算分母的值z=x2x3...xn，然后再计算y=x1/z。**

例3:避免出现上溢或下溢

$$\sum_{n=1}^{\infty} 1/n$$

我们来看在浮点数系统中计算级数会得到什么样的结果。**首先我们知道这个级数是发散的，但在浮点算术系统中却不是这个样子**。粗略分析，可能有两种情况，一是部分和非常大以至于发生上溢，二是1/n变得越来越小，产生下溢；但事实上，在达到这两种情况之前，计算结果就已经不再变化了。这是因为，一旦增加量1/n 与部分和的值相差悬殊，它们的和就停止变化了

$$\frac{1}{n} \leq \frac{\epsilon_M}{2} \sum_{k=1}^{n-1} \frac{1}{k}$$

由于大数吃小数，后面的级数项都不再起作用。

例4:尽量减少运算次数

比方说我们要进行多项式运算

$$P_n(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$$

如果直接计算每一项再逐项相加，一共需要**n(n+1)/2次乘法和n次加法。一种简单的改进可以是**

$$b := a_0$$

$$\text{For } i = 1, 2, \cdots, n$$

$$\qquad b := xb + a_i$$

$$\text{End}$$

$$P_n(x) := b$$

补偿求和方法

$$S_n = \sum_{i=1}^{n} x_i$$

要控制舍入误差的影响，除了分配更多的内存，提高浮点数的精度，还有一种方法就是所谓的补偿求和方法，也称**Kahan求和算法**。例如我们需要计算上面的求和。其中n≫1，例如n～O(10^7) 或更大。一般来说，每一次加法计算机都会进行舍入。舍入过程一定造成误差ε。对于一个一般的n项求和，如果每次加法造成的舍入误差为ε，那么通常Sn的舍入误差最大可能大约是**O(nε)**。

$s = 0.0$

For $i = 1, \cdots, n$

$s = s + x_i$

End

$s = 0.0, \quad c = 0.0$

For $i = 1, \cdots, n$

$y = x_i - c$ （一开始的时候，这一步什么都没干。）

$t = s + y$ （我们考虑 s 会比 y 大得多，那么实际上加法过程中 y 的低位信息丢失了。）

$c = (t - s) - y$ （我们先计算 $t - s$，这步相减得到实际上是 y 的高位的信息）

（再减 y 得到是 $-y$ 的低位信息。在循环的过程中 $x_i - c$ 会把丢失掉的 y 低位信息补正）

$\text{sum} = t$

End

可证明的是，**这种补偿求和的方法基本可使S_n的误差达到ε的量级**，也就是说，它完全不依赖于求和项数，舍入误差基本上就是每个待加达到数的原始精度。这当然是非常理想的。如还希望进一步高精度，则只能通过提高每个数据x_i的精度来实现了。

计算复杂度

- **计算复杂度是测量一个算法运行所需要的计算资源的测度，体现在时空两个方面**，分别称为**时间复杂度**(计算机时间资源)和**空间复杂度**(储存器的空间资源)。计算复杂度显然取决于被求解问题的规模大小，输入数据以及算法本身。
- **时间复杂度不能简单用绝对时间消耗来测算**。因为不同计算机消耗的时间可能相差很大。**时间的计算复杂度用算法本身的基本运算次数或基本操作量来测量**，这样将算法的效率与具体所使用的计算机隔离开。早期计算机乘除法比加减法耗时长，那时只计入乘除法数量；而20世纪90年代后，计算机的运行速度大幅提高，加减和乘除法的速度相差较小，故现在一般估算运算次数的总和。在实际中，有时甚至要考虑数据传输的时间消耗，因为有的计算机四则运算的速度比数据在计算机与外围设备之间的传输更快。另外，对于大规模并行计算，不同节点间的数据交换时间是需要慎重考虑的重要因素。

计算复杂度

- 另外，**计算复杂度也与数值问题的规模大小n密切相关**。复杂度的阶的概念：如果存在正常数C和n_0，使得$n \geq n_0$时，$T(n) \leq Cf(n)$成立，则称该算法的时间复杂度$T(n)$是$f(n)$阶的，记作$T(n) = O(f(n))$。据此定义，$f(n)$是$T(n)$的上界，例如$\frac{2}{3}n^2 = O(n^2)$，$\frac{n(n-1)}{2} = O(n^2)$.
- 有如下定理：**复杂度阶数的关系：** $O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(n^{100}) < O(2^n) < O(3^n) < O(n!) < n^n$。
  - 用时间复杂度可表示为问题规模n的多项式函数或指数函数来建立区分算法好与坏的分水岭。前者被认为是好的算法，时间复杂度随问题规模的增长可以容忍；但是后者的增长无法容忍。例如，按照定义计算行列式的算法是一个极坏的算法，与数值不稳定的算法一样，不能在实际中使用。
  - 若问题的各种现有算法中最好的是多项式算法，则称该问题是多项式问题。即使是多项式算法，我们也要慎重考虑是否采用或者加以算法改进。例如快速傅里叶变换将传统算法的$O(n^2)$次运算次数降低为$O(n\log_2 n)$。

计算复杂度

- 最后，即使同一规模的数值问题，实例(输入数据)不同，也会使运算量差别巨大。也就是说，**时间复杂度也是输入数据的函数。**
- 输入数据对计算复杂度的影响反映在最好情况、最坏情况和平均情况三种不同情况下的复杂度大小上。但是，计算复杂性的研究是一门专门的学科，非常艰深，与它密切相关的问题是NP理论。

  NP就是Non-deterministic Polynomial的问题，亦即多项式复杂程度的非确定性问题。
- **形象的例子**: 有10把外形完全相同但钥匙不同的锁，有人把锁与钥匙完全搞混乱了，只能通过试开来配对。问此人在最好情况下用几次可以完成配对？最坏的情况要开几次？平均需要开几次？

计算复杂度

- 解: 把锁编上号1,2,...,10。钥匙相对于锁而言是完全随机的，钥匙总的可能排列是10!，每一种出现的几率是对等的，使得试开锁的次数A是随机的。最理想的情况是，钥匙的排列正好与锁的排列对上，只需要试开9次，但是这种最好的几率只有1/10!。最坏的情况是，1号锁的钥匙在第10的位置，试了9次才对上，剩下的9把锁的试开，也按照最坏情况，即2号锁需要8次，3号7次，等等。总的次数加起来是45次。最坏情况出现的几率也只有1/10!。大多数情况下，开锁的复杂度介于最好和最坏之间。那么平均的计算复杂度是多少呢？运用概率论，可以算出随机变量A的数学期望值是29.571，这个期望值被定义为**平均计算复杂度**。它不等于9和45的算术平均值$(9+45)/2$，也不等于它们的几何平均值$\sqrt{9*45}=20.125$。

矩阵

一个数域K上面的 **m行(row) n列(column)**的矩阵A我们一般记为:A∈K^{m×n}。数域K最为常见的情形是复数域C和实数域 R。矩阵可以视为两个矢量空间K^n和K^m之间的一个线性映射。

$A=(a_1,a_2,\ldots a_n)$ m=n时，A为方阵。

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ \vdots \\ m \end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & 2 & \cdots & n \end{array} \\
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
a_{31} & a_{32} & \cdots & a_{3n} \\
\vdots & \vdots & \vdots & \vdots \\
a_{m1} & a_{m2} & \cdots & a_{mn}
\end{bmatrix}
\end{array}
$$

特殊矩阵

- 正交矩阵：$Q^TQ = QQ^T = I$
- **幺正矩阵**：$U^\dagger U = I$
  - U是mxn的，$U^\dagger U = I$ <=> U的列向量是正交归一的
  - U是方阵，$U^\dagger U = UU^\dagger = I$
- 矩阵A的共轭转置$A^\dagger = A^H = \bar{A}^T$
  - $\left(A^\dagger\right)_{ij} = \overline{A_{ji}}$
  - $(AB)^\dagger = B^\dagger A^\dagger$
- 方阵A的逆矩阵
  - $A^{-1}A = AA^{-1} = I$
- 厄米矩阵$H^\dagger = H$
  - 厄米矩阵是方阵
- **定理**：厄米矩阵$H^\dagger = H$，对于任意的$\boldsymbol{z} \in \mathbb{C}^n$，$\boldsymbol{z}^\dagger H\boldsymbol{z}$是实数
- 证明：$\left(\boldsymbol{z}^\dagger H\boldsymbol{z}\right)^\dagger = \boldsymbol{z}^\dagger H^\dagger \boldsymbol{z} = \boldsymbol{z}^\dagger H\boldsymbol{z}$

方阵的迹和行列式

一个**方阵的迹**就是该矩阵对角元之和：

$$Tr(A) = \sum_{i=1}^{n} a_{ii}$$

矩阵的行列式在求解线性方程中具有重要的意义。它的原初定义为

$$\det(A) = \sum_{\pi \in P} \text{sign}(\pi) a_{1\pi_1} a_{2\pi_2} \cdots a_{n\pi_n}$$

π表示(12 …n)的一个排列而**sign(π)则表示该排列的奇偶性**，即经过奇数/偶数次对换可以恢复到原始排列

对某一固定i

$$\det(A) = \sum_{j=1}^{n} \Delta_{ij} a_{ij}$$

$$A^{-1} = \frac{1}{\det(A)} \Delta^{\mathrm{T}}$$

一个方阵的逆矩阵存在的充分必要条件是它的行列式不为零

其中 $i \in [1, n]$ 是任意一个行指标而 Δ_{ij} 是矩阵元 a_{ij} 的代数余子式。

$\Delta_{ij} = (-)^{i+j} \det(A_{ij})$，$A_{ij}$ 则是将原矩阵的第 i 行第 j 列消去后获得的矩阵。

Cramer's rule

由上可得求解线性方程Ax=b的方法

$$x_j = \Delta_j / \det(A)\ ,\quad j = 1, \cdots, n$$

其中Δ_j是将原矩阵中的第j列换为矢量b所得到矩阵之行列式的值。

但是，上述行列式、逆矩阵以及线性方程求解的公式并不能直接用于数值计算。因为按照这些公式，**行列式的计算涉及到O(n!)的计算量。这对于即使不太大的矩阵来说也过于庞大了。**

例如，即使对于大约100阶的矩阵来说(或者说求解100个联立的线性方程组)，按照这些公式计算的话在我们有生之年都不太可能算出结果。为了获得数值上近似的解，我们需要更聪明的计算方法。

矩阵的秩和核

秩 –记为rank(A)–可以定义为从矩阵A中能够抽取的非奇异的子矩阵(行列式不为零) 的最大的阶数。
当A被视为K^n→K^m 的线性映射时，我们可以定义其值域为:

$$\text{range}(A) = \{y \in K^m : y = A \cdot x, \quad x \in K^n\}$$

而矩阵的秩也可以定义为其值域空间的维数:
rank(A)= dim(range(A))。
另一个重要的概念是矢量的线性相关。**一个矩阵按照行的秩与其按照列的秩定义为线性无关的矢量的数目。**严格来说，我们需要区分矩阵按照行的秩和按照列的秩。但是线性代数的基础知识告诉我们这两个是一致的。

线性映射A的核定义如下，它其实是满足Ax=0的矢量构成的子空间

$$\ker(A) = \{x \in K^n : A \cdot x = 0\}$$

- $\text{rank}(A)=\text{rank}(A^T)$
- $\text{rank}(A)+\dim(\ker(A))=n$

矢量与矩阵的模

$$A\vec{x}=\vec{b} \xrightarrow{\text{数学上}} \vec{x}=A^{-1}\vec{b}$$
$$A\vec{x}=\vec{b} \xrightarrow{\text{计算物理中}} \min||A\vec{x}-\vec{b}||$$

数学中可以对一般的模进行定义。**一个矢量空间V上的模||·||**
一般来说可以定义为满足下列条件的**非负函数**

- 非负性：$||\vec{v}|| \geq 0, \forall \vec{v} \in V$ 且 $||\vec{v}||=0$ 当且仅当 $\vec{v}=0$。
- 均匀性：$||\alpha\vec{v}|| = |\alpha| \cdot ||\vec{v}||; \forall \alpha \in K, \forall \vec{v} \in V$
- 三角不等式：$||\vec{v}+\vec{w}|| \leq ||\vec{v}|| + ||\vec{w}||, \forall \vec{v}, \vec{w} \in V$

Otto Ludwig Hölder

Otto Hölder

Born	22 December 1859 Stuttgart, Germany
Died	29 August 1937 (aged 77) Leipzig, Germany

一个常用的模是所谓的**p-模，又称为Hölder模**，它由下式定义

$$||\vec{x}||_p = \left(\sum_{i=1}^{n}|x_i|^p\right)^{1/p}, \forall \vec{x} \in V, 1 \leq p < \infty$$

取极限p→∞，就得到无穷模，它实际上仅仅挑选出矢量$\vec{x}$的分量中模最大的那个：

$$||\vec{x}||_\infty = \max_{1\leq i\leq n}|x_i|$$

另外一个经常用到的是**p=2**的情形。对实空间和复空间来说，这个模称为相应空间的**欧氏模**

$$||\vec{x}||_2 = (x,x)^{1/2} = (x^\dagger x)^{1/2} = \left(\sum_{i=1}^{n}|x_i|^2\right)^{1/2}$$

模的等价性

矢量空间V上的两个模$||\cdot||_p$和$||\cdot||_q$被称为等价，如果存在两个正的与矢量x无关的常数c>0和C>0使得：

$$c||\vec{x}||_q \leq ||\vec{x}||_p \leq C||\vec{x}||_q, \quad \forall \vec{x} \in V$$

相应的这些常数c和C被称为等价常数。**可以证明三种不同的p-模(p=1,2,∞)都是等价的**

引入一个" 单位圆"，记为 $S = \{\vec{x} | \, ||\vec{x}||_\infty = 1\}$，明显 S 是个有界闭集。由于 $f(x) = ||\vec{x}||_p$ 是 S 上的连续函数，$f(x)$ 在 S 上可以取到最小值和最大值。设最小值为 c，最大值为 C。那么 $c \leq f(x) \leq C$。另一方面，$\forall \vec{x} \in K^n$ 且 $\vec{x} \neq 0$，则 $\vec{x}/||\vec{x}||_\infty \in S$，则有

$$c \leq \left|\left| \frac{\vec{x}}{||\vec{x}||_\infty} \right|\right|_p \leq C \quad \Rightarrow \quad c \leq \frac{||\vec{x}||_p}{||\vec{x}||_\infty} \leq C$$

$$\Rightarrow \quad ||\vec{x}||_p = ||\vec{x}||_\infty$$

事实上，有限维矢量空间中的任何模都是等价的。

矩阵的模

对于K^{m×n}上的矩阵，它的模||·||定义满足：

- 非负性：$||A|| \geq 0, \forall A \in K^{m\times n}$ 且 $||A|| = 0$ 当且仅当 $A = 0$(所有矩阵元为 0)
- 均匀性：$||\alpha A|| = |\alpha| \cdot ||A||; \forall \alpha \in K, \ \forall A \in K^{m\times n}$
- 三角不等式：$||A + B|| \leq ||A|| + ||B||, \forall A, B \in K^{m\times n}$

如果矩阵模和矢量模满足如下关系，我们就称它们兼容。

$$||A\vec{x}|| \leq ||A|| \cdot ||\vec{x}||, \forall \vec{x} \in K^n, \forall A \in K^{m\times n}$$

一个矩阵模||·||被称为**服从乘法模**，如果它满足

$$||AB|| \leq ||A|| \cdot ||B||, \forall A \in K^{n\times m}, \forall B \in K^{m\times q}$$

值得指出的是，**并不是所有的矩阵模都是服从乘法的模**。一个简单的例子是所谓的最大模，其定义为$||A||_\Delta = \max(|a_{ij}|)$。可验证它满足矩阵模的所有条件因而构成一个矩阵模。

$$A = B = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

容易验证 $||AB||_\Delta = 2 > ||A||_\Delta ||B||_\Delta = 1$。

矢量模诱导的矩阵模

$$||A|| = \sup_{\vec{x}\neq 0} \frac{||A\vec{x}||}{||\vec{x}||}$$

这里的sup表示集合的上确界，首先它代表了集合的上界 (upper bound)，而且这个上界确实可以取到的。

- **可证明诱导矩阵模是与诱导它的矢量模兼容的，也是服从乘法的。**
- **存在一个非0矢量y，使得||Ay||=||A||·||y||**

我们可以看到，在诱导矩阵模的定义里，x→αx是不改变模的定义的。于是我们可以把诱导矩阵模写成

$$||A|| = \sup_{||\vec{x}||=1} ||A\vec{x}||$$

我们一定可以找到一个矢量y，这里y满足||y||=1，并且

$$\sup_{||\vec{x}||=1} ||A\vec{x}|| = ||A\vec{y}||$$

于是我们就是证明了 $||A|| \cdot ||\vec{y}|| = ||A\vec{y}||$

矢量的 p-模所诱导的矩阵模

$$||A||_p = \sup_{\vec{x}\neq 0} \frac{||A\vec{x}||_p}{||\vec{x}||_p}, \quad \forall \vec{x} \in V, \quad \vec{x} \neq 0$$

$$||A||_1 = \max_{j=1,\cdots,n} \sum_{i=1}^{m} |a_{ij}|, \quad ||A||_\infty = \max_{i=1,\cdots,m} \sum_{j=1}^{n} |a_{ij}|$$

Proof: Let $\bar{\jmath}$ be chosen so that $\max_{0\leq j<n} \|a_j\|_1 = \|a_{\bar{\jmath}}\|_1$. Then

$$\begin{aligned}
\max_{\|x\|_1=1} \|Ax\|_1 &= \max_{\|x\|_1=1} \left\| \left(a_0 \mid a_1 \mid \cdots \mid a_{n-1} \right) \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} \right\|_1 \\
&= \max_{\|x\|_1=1} \|\chi_0 a_0 + \chi_1 a_1 + \cdots + \chi_{n-1} a_{n-1}\|_1 \\
&\leq \max_{\|x\|_1=1} (\|\chi_0 a_0\|_1 + \|\chi_1 a_1\|_1 + \cdots + \|\chi_{n-1} a_{n-1}\|_1) \\
&= \max_{\|x\|_1=1} (|\chi_0|\|a_0\|_1 + |\chi_1|\|a_1\|_1 + \cdots + |\chi_{n-1}|\|a_{n-1}\|_1) \\
&\leq \max_{\|x\|_1=1} (|\chi_0|\|a_{\bar{\jmath}}\|_1 + |\chi_1|\|a_{\bar{\jmath}}\|_1 + \cdots + |\chi_{n-1}|\|a_{\bar{\jmath}}\|_1) \\
&= \max_{\|x\|_1=1} (|\chi_0| + |\chi_1| + \cdots + |\chi_{n-1}|) \|a_{\bar{\jmath}}\|_1 \\
&= \|a_{\bar{\jmath}}\|_1.
\end{aligned}$$

Also,

$$\|a_{\bar{\jmath}}\|_1 = \|Ae_{\bar{\jmath}}\|_1 \leq \max_{\|x\|_1=1} \|Ax\|_1.$$

Hence

$$\|a_{\bar{\jmath}}\|_1 \leq \max_{\|x\|_1=1} \|Ax\|_1 \leq \|a_{\bar{\jmath}}\|_1$$

一个是对列向量的元素进行求和的模，一个是对行向量的元素进行求和的模。所以，我们有 $||A||_1 = ||A^T||_\infty$

https://www.cs.utexas.edu/users/flame/Notes/NotesOnNorms.pdf

本征值、谱半径

$$\rho(A) := \max_{1\le i\le n} |\lambda_i|$$

谱半径(spectral radius)定义为

这里λi是矩阵的本征值。那么一个矩阵的诱导矩阵模一定大于等于该矩阵的谱半径

$$||A|| \geq \rho(A)$$

对于 $\forall\lambda$，$A\vec{x} = \lambda\vec{x}$，我们一定有

$$|\lambda| \cdot ||\vec{x}|| = ||\lambda\vec{x}|| = ||A\vec{x}|| \leq ||A|| \cdot ||\vec{x}||$$

$$\frac{||B\vec{x}||}{||\vec{x}||} = \frac{(\vec{x}^\dagger B^\dagger B\vec{x})^{\frac{1}{2}}}{(\vec{x}^\dagger\vec{x})^{\frac{1}{2}}} = \left(\frac{\sum_{i=1}^n |c_i|^2 v_i}{\sum_{j=1}^n |c_j|^2}\right)^{\frac{1}{2}}$$

我们把矢量 $\vec{x}$ 写成了 $\vec{x} = \sum_{i=1}^n c_i\vec{\alpha}_i$ 的形式。这里 $\vec{\alpha}_i$ 是 $B^\dagger B$ 的正交本征矢。

$$\frac{||B\vec{x}||}{||\vec{x}||} \leq v_n = \left[\rho(B^\dagger B)\right]^{\frac{1}{2}}$$

并且当 $\vec{x} \propto \vec{\alpha}_n$ 时，等号成立。于是有 $||B|| = \left[\rho(B^\dagger B)\right]^{\frac{1}{2}}$。

厄米矩阵的p=2诱导矩阵模就等于该矩阵最大的本征值。但是非厄米矩阵不具备这样性质。

$$||A||^2 = \rho(A^\dagger A) = \rho(A^2) = \rho^2(A)$$

矩阵条件数

给定一个非奇异的方阵A∈C^{n×n}，它的条件数定义为

$$K(A) = ||A|| \cdot ||A^{-1}||$$

$$\|A\| = \max_x \frac{\|Ax\|}{\|x\|}$$

条件数同时描述了矩阵对向量的拉伸能力和压缩能力，换句话说，令向量发生形变的能力。条件数越大，向量在变换后越可能变化得越多。

$$\|A^{-1}\| = \max_y \frac{\|A^{-1}y\|}{\|y\|} = 1/\min_y \frac{\|y\|}{\|A^{-1}y\|} = 1/\min_x \frac{\|Ax\|}{\|x\|}$$

其中的模||·||可以是任意良好定义的模。例如，如果是我们前面定义的p−模，我们会将相应矩阵的模记为**Kp(A)**。**一个矩阵的条件数一般来说依赖于模的选取。不过奇异矩阵的条件数总是趋于无穷大，而一个接近奇异的矩阵则具有非常大的条件数。**虽然任何的矩阵模都可以用于条件数的定义，但是**通常我们总是采取服从乘法的矩阵模**。对于这类矩阵模我们有：

$$1 = ||AA^{-1}|| \leq ||A|| \cdot ||A^{-1}|| = K(A)$$

矩阵条件数

最为常用的是欧氏模定义的K2(A)。我们有，

$$K_2(A) = \frac{\sigma_1(A)}{\sigma_n(A)}, \qquad \text{最大和最小奇异值之比}$$

如果A是个厄米矩阵，而我们用的范数是p=2的欧氏模，那么

$$\text{cond} = \rho(A) \cdot \rho(A^{-1}) = |\lambda|_{\max} / |\lambda|_{\min}$$

所以一个厄米矩阵最大本征值和最小本征值的比值决定了很多问题本身的病态程度。如果问题非常病态，那么，要求得一个精确解是非常困难的。这个时候，**我们就需要采取一些手段，把病态问题变成良态问题。**详见下一章。

特殊形状的矩阵

对角矩阵是指仅仅对角元a_{ii}不为零的矩阵。通常意义下是指方阵，但是此定义也适用于长方阵。

一个矩阵$A\in K^{m\times n}$，如对$i>j$有$a_{ij}=0$，就称矩阵为**上梯形矩阵**。相应的，如对$i<j$有$a_{ij}=0$，就称矩阵A为**下梯形矩阵**。

可以验证，如果$m<n$，上梯形矩阵的非零矩阵元恰好构成一个梯形。

$$\begin{pmatrix} x & x & x & x \\ & x & x & x \\ & & x & x \end{pmatrix}_{3\times 4}$$

对m=n的方阵而言，上/下梯形矩阵分别称为**上/下三角矩阵**。

它的行列式就是对角元的乘积。**而且它的逆矩阵仍然维持原矩阵的上下三角的性质。**如果上/下三角矩阵的对角元都等于1，这样的上/下三角矩阵称为单位上/下三角矩阵。

特殊形状的矩阵

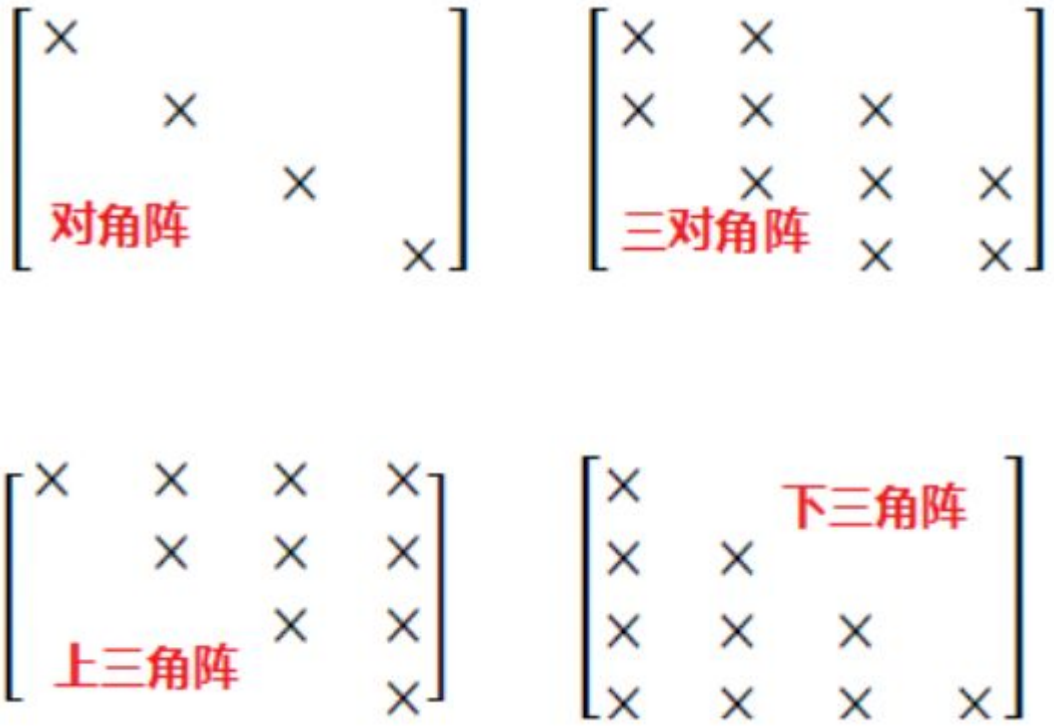
三角矩阵的概念可以稍加推广到所谓的**带型矩阵**。一般来说，对于$A\in K^{m\times n}$，我们称其具有**上带p**，如果对$i>j+p$必定有$a_{ij}=0$；相应的，我们称其具有**下带q**，如果对于$j>i+q$必定有$a_{ij}=0$。

$$\begin{pmatrix} x & x & x & x & x & x \\ x & x & x & x & x & x \\ & x & x & x & x & x \\ & & x & x & x & x \\ & & & x & x & x \end{pmatrix}_{i>j+1}, \quad \begin{pmatrix} x & x & & & & \\ x & x & x & & & \\ & x & x & x & & \\ & & x & x & x & \\ & & & x & x & x \end{pmatrix}_{p,q=1}$$

利用这个概念我们可以统一上面提及的几种矩阵。例如，对角矩阵是p=q=0的带型矩阵；下梯形矩阵是具有p=m-1；q=0的带型矩阵；上梯形矩阵则是具有p=0；q =n-1的带型矩阵。**如果带型矩阵的p=q=1，则该带状矩阵称为三对角矩阵。**另外两种情形是上双对角(p=0,q=1)和下双对角 (p=1,q=0) 矩阵。另外一类我们后面会用到的矩阵是所谓的**上/下Hessenberg矩阵。下Hessenberg 矩阵具有p=m-1；q=1 而上Hessenberg矩阵则具有p=1；q=n-1。**

特殊形状的矩阵

对于稀疏矩阵，常用“×”标记非零元素，这种矩阵图称为威尔金森图(Wilkinson graph)。储存和计算时，尽量只考虑非零元素。如下图：



Karl Hessenberg

German mathematician

Karl Adolf Hessenberg was a German mathematician and engineer. The Hessenberg matrix form is named after him. From 1925 to 1930 he studied electrical engineering at the Technische Hochschule Darmstadt and graduated with a diploma. Wikipedia

Born: 8 September 1904, Frankfurt, Germany

Died: 22 February 1959, Frankfurt, Germany

Education: Technische Universität Darmstadt

Siblings: Kurt Hessenberg

二次型和正定矩阵

二次型起源于矢量空间中的**标量积运算**。矢量空间V中的标量积可以视为V×V到K的一个映射(.;.)，它满足：

- 双线性：$(\alpha\vec{x}+\beta\vec{y},\vec{z})=\alpha(\vec{x},\vec{z})+\beta(\vec{y},\vec{z}),\ \forall\vec{x},\vec{y},\vec{z}\in V,\ \forall\alpha,\beta\in K$
- 厄米性：$(\vec{x},\vec{y})=(\vec{y},\vec{x})^*,\ \forall\vec{x},\vec{y}\in V$
- 正定性：$(\vec{x},\vec{x})>0,\ \forall\vec{x}\in V$ 除非 $\vec{x}=0$

对于空间C^n来说，可以定义内积为

$$(\vec{x},\vec{y})=\vec{y}^{\dagger}\cdot\vec{x}=\sum_{i=1}^{n}y_i^*x_i \qquad (A\vec{x},\vec{y})=(\vec{x},A^{\dagger}\vec{y})$$

如果对于C^{n×n}(或R^{n×n})中的矩阵A以及任意的非零矢量 x∈V 都有(Ax;x)是正的实数，我们就称矩阵A是正定的。如果 (Ax;x)≥0且等号有可能成立，我们就称A是半正定的。**对于 C^{n×n}中的复矩阵A，它是正定的条件要求A必定是厄米的(从而其本征值均为实数)并且所有本征值都是正的。**

复厄米正定矩阵

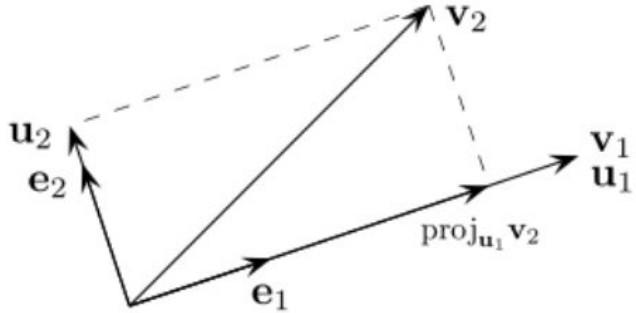
令A∈C^{n×n}为厄米矩阵。那么它是正定矩阵**当且仅当下列等价的条件之一**获得满足：

- $(A\vec{x}, \vec{x}) > 0, \forall \vec{x} \neq 0, \vec{x} \in C^n$
- A 的主子矩阵的本征值都是正的
- A 的主子矩阵的行列式都是正的 (又称 Sylvester 判据)
- 存在一个非奇异矩阵 $H \in R^{n\times n}$ 使得 $A = H^{\dagger}H$

所谓**n阶主子矩阵**，是指任意取n行，再选取相同行号的列，所构成的矩阵。

正是这最后一个条件使得我们对于正定的厄米矩阵可以采用所谓的Cholesky分解。事实上，非奇异的H不仅仅是存在的，我们还可以将其选为上三角矩阵(从而H^diag为下三角矩阵)。

Gram-schmidt正交化



We define the projection operator by

$$\operatorname{proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle} \mathbf{u},$$

where $\langle \mathbf{u}, \mathbf{v} \rangle$ denotes the inner product of the vectors **u** and **v**. This operator projects the vector **v** orthogonally onto the line spanned by vector **u**. If $\mathbf{u} = \mathbf{0}$, we define $\operatorname{proj}_{\mathbf{0}}(\mathbf{v}) := \mathbf{0}$, i.e., the projection map $\operatorname{proj}_{\mathbf{0}}$ is the zero map, sending every vector to the zero vector.

The Gram–Schmidt process then works as follows:

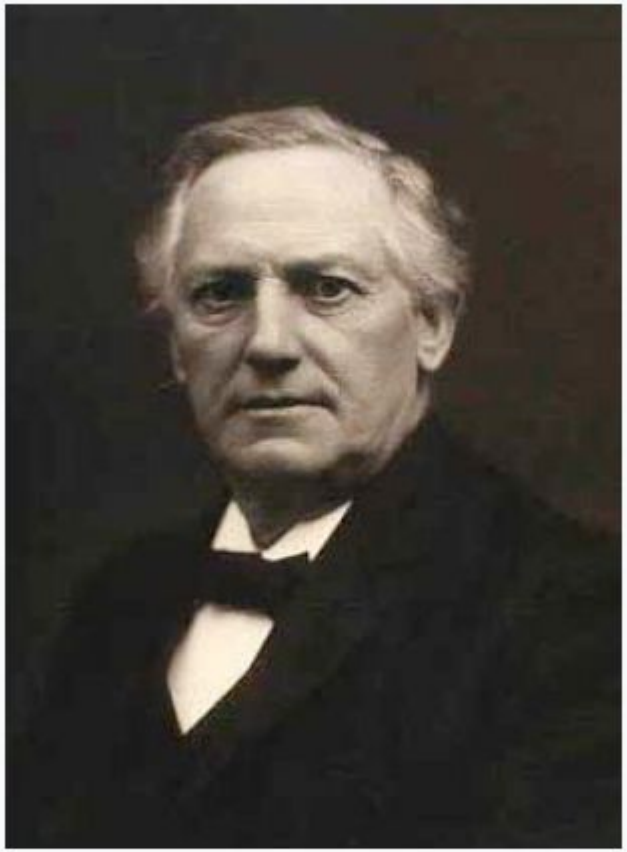
$$\begin{aligned}
\mathbf{u}_1 &= \mathbf{v}_1, & \mathbf{e}_1 &= \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|} \\
\mathbf{u}_2 &= \mathbf{v}_2 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_2), & \mathbf{e}_2 &= \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} \\
\mathbf{u}_3 &= \mathbf{v}_3 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_3), & \mathbf{e}_3 &= \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} \\
\mathbf{u}_4 &= \mathbf{v}_4 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_3}(\mathbf{v}_4), & \mathbf{e}_4 &= \frac{\mathbf{u}_4}{\|\mathbf{u}_4\|} \\
&\vdots & &\vdots \\
\mathbf{u}_k &= \mathbf{v}_k - \sum_{j=1}^{k-1} \operatorname{proj}_{\mathbf{u}_j}(\mathbf{v}_k), & \mathbf{e}_k &= \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}.
\end{aligned}$$

The Gram–Schmidt process takes a finite, linearly independent set of vectors $S = \{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ for $k \le n$ and generates an orthogonal set $S' = \{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$ that spans the same k-dimensional subspace of $\mathbf{R}^n$ as S.

The sequence $u_1, \ldots, u_k$ is the required system of orthogonal vectors, and the normalized vectors $e_1, \ldots, e_k$ form an *orthonormal* set.

矩阵A=($v_1, \ldots, v_n$)

Jørgen Pedersen Gram



Born 27 June 1850
Nustrup, Duchy of Schleswig, Denmark

Died 29 April 1916 (aged 65)
Copenhagen, Denmark

Erhard Schmidt

Erhard Schmidt (courtesy MFO)

Born 13 January 1876
Tartu, Governorate of Livonia (now Estonia)

Died 6 December 1959
(aged 83)
Berlin

Gram矩阵

In linear algebra, the **Gram matrix** (or **Gramian matrix**, **Gramian**) of a set of vectors $v_1, \ldots, v_n$ in an inner product space is the Hermitian matrix of inner products, whose entries are given by $G_{ij} = \langle v_i, v_j \rangle$.[1] If the vectors $v_1, \ldots, v_n$ are real and the columns of matrix X, then the Gram matrix is $X^\top X$.

An important application is to compute linear independence: a set of vectors are linearly independent if and only if the Gram determinant (the determinant of the Gram matrix) is non-zero.

$$G(x_1, \ldots, x_n) = \begin{vmatrix} \langle x_1, x_1 \rangle & \langle x_1, x_2 \rangle & \ldots & \langle x_1, x_n \rangle \\ \langle x_2, x_1 \rangle & \langle x_2, x_2 \rangle & \ldots & \langle x_2, x_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle x_n, x_1 \rangle & \langle x_n, x_2 \rangle & \ldots & \langle x_n, x_n \rangle \end{vmatrix}.$$

$$G = A^T A = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix} \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^T\mathbf{a}_1 & \mathbf{a}_1^T\mathbf{a}_2 & \cdots & \mathbf{a}_1^T\mathbf{a}_n \\ \mathbf{a}_2^T\mathbf{a}_1 & \mathbf{a}_2^T\mathbf{a}_2 & \cdots & \mathbf{a}_2^T\mathbf{a}_n \\ & & & \\ \mathbf{a}_n^T\mathbf{a}_1 & \mathbf{a}_n^T\mathbf{a}_2 & \cdots & \mathbf{a}_n^T\mathbf{a}_n \end{bmatrix},$$

对任一个实矩阵A，G是半正定的

$$\mathbf{x}^T A^T A \mathbf{x} = (A\mathbf{x})^T (A\mathbf{x}) = \|A\mathbf{x}\|^2 \geq 0$$

m×n*n×m -> m×m