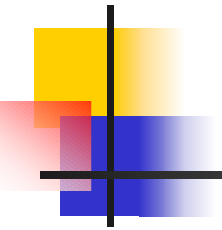
數值方法中的主要誤差來源

國立交通大學 運輸科技與管理學系
王晉元

誤差來源– run off errors

- 由於在電腦上進行數值計算時，誤差主要由實數的運算而來，以下的討論就放在實數上
  - 但是實數的表示方式會因為不同的系統而有所差別，因此以下只能介紹觀念
- 主要是由於在電腦中數字（浮點數）的表達是由有限的位元（bit）數來表示的
  - 在電腦系統中的實數，其實是不連續的
  - 不可能用有限的bit來完整表示無限（連續）的數字
  - 因此免不了就有些進位或是去尾的誤差



- 在某些情況下，導致計算結果沒有意義
- 許多此類問題可以透過適當的程式寫作技巧來避免

不同進位的數字表示方式

- r為進位的基礎數字表示的符號

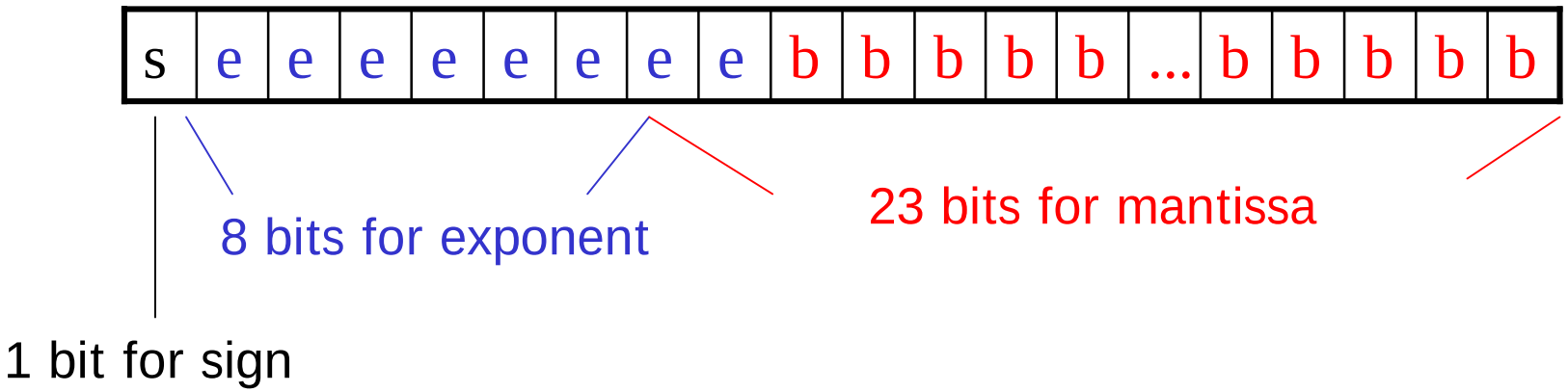
$(abcdef.ghi)_r$

$= ar^5+br^4+cr^3+dr^2+er^1+fr^0+gr^{-1}+hr^{-2}+ir^{-3}$

$(101.01)_2 = 1*2^2+0*2^1+1*2^0+0*2^{-1}+1*2^{-2}$

$= 5.25$

- 通常而言，一個單精確度的浮點數是用32個bits來表示



所代表的數字為（實數的表現方式之一）

if g (exponent) > 0

$x = (sign)(0.1bbb\ bbbb\ bbbb\ bbbb\ bbbb\ bbbb)_2\ 2^{(g-p)}$

if g (exponent) = 0

$x = 0$

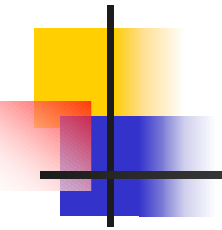
其中p通常使用的數值為128，g為指數，b為23bits的mantissa，因為第一個bit永遠是1，所以有效位數為24

$(0.1bbb\ bbbb\ bbbb\ bbbb\ bbbb\ bbbb)_2$ | $2^{(g-128)}$

- 因為g用8個bits來表示，所以0<=g<=255，但是0與255被保留作為特殊的意義
- 因此 $2^{-127} <= 2^{(g-p)} <= 2^{126}$
- 最小以及最大的mantissa分別為$(0.1)_2$=0.5與$(0.1111\ldots.)_2 = 1-2^{-24}$
- 可以表示的最小實數為

  $0.5 \times 2^{-127} = 2.938735\ldots \times 10^{-39}$
- 可以表示的與最大實數為

  $(1-2^{-24}) \times 2^{126} = 1.70141\ldots \times 10^{38}$



- 既然是用有限的bit來表示實數，因此可以表示的時數也有限
  - 也就是在兩個連續能夠表示的實數間一定有個間隔
- 一個名詞：machine epsilon (e)
  - 數字1與電腦可表示最接近數字1（大於1，但不是1）間的差值
  - 通常大約是$1.19x10^{-7}$
  - 對一個實數R而言，到下一個實數的間隔大約是(epsilon * R)

1.7×10^{38} (largest positive floating number)

(second largest floating number)

gap

1.000000119 (1 plus epsilon)

1.0

$2.9 \times 10^{-39} + 3.45 \times 10^{-46}$ (second smallest floating number)

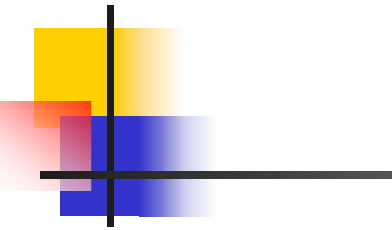
2.9×10^{-39} (smallest positive floating number)

gap

0

在電腦中的Run-Off Errors

- 在電腦中主要的誤差來源就是不得不以有限的bit來表示實數
  - 介於1與(1+epsilon)數字是沒有辦法在電腦中表示的
  - 如果0 < a < (epsilon/2)，則任何(1+a)的數字都會用1來表示
  - 如果a >= (epsilon/2)，則任何(1+a)的數字都會(1+epsilon)來表示



- 也就是對數字1而言，(epsilon/2)可以看成是最大的誤差
- 或者說當在電腦中出現一個數字1時，其原始數字可能介於(1-e/2)與(1+e/2)之間
- 對於任何實數R，round-off error大約是
  - (epsilon * R) /2 ，如果是進位
  - (epsilon * R)，如果去尾

來看一個計算epsilon的程式

```
#include <stdio.h>
void main(void){
  float ex, g = 1, eps;

  do {
     g = g/2;
     ex = g + 1;
     ex = ex - 1;
     printf("g=%16.8e,ex=%16.8e\n", g, ex);
     if (ex > 0)
           eps = ex;
  } while (ex > 0);
  printf("Epsilon = %16.8e\n\n", eps);
}
```

g = 5.00000000e-001, ex = 4.90000000e-001
g = 2.50000000e-001, ex = 2.45000000e-001
g = 1.25000000e-001, ex = 1.22500000e-001
g = 6.25000000e-002, ex = 6.12500000e-002
g = 3.12500000e-002, ex = 3.06250000e-002
…….
g = 8.88178420e-016, ex = 8.88178420e-016
g = 4.44089210e-016, ex = 4.44089210e-016
g = 2.22044605e-016, ex = 2.22044605e-016
g = 1.11022302e-016, ex = 0.00000000e+000
Machine Epsilon = 2.22044605e-016

Run-off errors的影響

- 較大誤差通常發生在
  - 從一個很大的數字加（或減）一個很小的數字的時後
  - 當兩個很接近的數字相減的時後
- 以下的程式將0.00001相加一萬次，看看會發生什麼結果

```
include <stdio.h>
void main(void){
  float sum = 1.0;
  int i, k = 1;

  for(i = 1; i <= 10000; i++){
     sum = sum + 0.00001;
  }
  printf("\nSum = %f\n", sum);
}
```

上述程式的執行結果為**1.10036**，但是正確結果應該為**1.1**

❑來看看原因，以1+0.00001為例

- ➢ $(1)_{10}$ = $(0.1000\ 0000\ 0000\ 0000\ 0000\ 0000)_2 * 2^1$
- ➢ $(0.00001)_{10}$ = $(0.1010\ 0111\ 1100\ 0101\ 1010\ 1100)_2 * 2^{-16}$
- ➢ 將這兩個數字相加

$(1)_{10} + (0.00001)_{10}$ =

$(0.1000\ 0000\ 0000\ 0000\ 0101\ 0011\ 1110\ 0010\ 1101\ 0110\ 0)_2 \times 2^1$ =

$(0.1000\ 0000\ 0000\ 0000\ 0101\ 0011\ 1110\ 0010\ 1101\ 0110\ 0)_2 \times 2^1$ =

$(0.1000\ 0000\ 0000\ 0000\ 0101\ 0011)_2 \times 2^1$ =

$(1.0000\ 1001\ 36)_{10}$

- 相同的困擾也會發生在減法上面
  - 1.00001-1=

    $(0.1000\ 0000\ 0000\ 0000\ 0101\ 0100)_2 * 2^1 - (0.1)_2 * 2^1 =$

    $(0.0000\ 0000\ 0000\ 0000\ 0101\ 0100)_2 * 2^1 =$

    $(0.1010\ 1)_2 \times 2^{-16} = 1.00136 \times 10^{-5}$

處理策略

- 使用雙倍精確度的浮點數
  - 使用64個bits來表示一個實數（增加mantissa的個數）
  - 增加有效位數，減少run-off errors
- 使用群組
  - 減少run-off errors發生的次數
- 當然還有其他方式

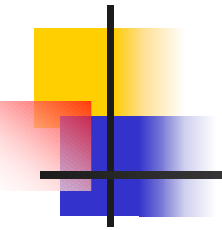
還是使用1加上一萬次0.00001的例子—群組方式

```
void main(void){
  float x, gr_total, sum = 1.0;
  int i, k = 0;

  for(i = 1; i <= 100; i++){
     gr_total = 0.0;
     for(k = 1; k <= 100; k++){
        gr_total = gr_total + 0.00001;
     }
     sum = sum + gr_total;
  }
  printf("Sum = %15.8f\n", sum);
}
```

- 沒有使用群組
  - 1.10036
- 使用群組
  - 1.10000467
- 可以看出有效減少誤差

還是使用1加上一萬次0.00001的例子—雙倍精確浮點數方式

```
include <stdio.h>
void main(void){
  double sum = 1.0;
  int i, k = 1;

  for(i = 1; i <= 10000; i++){
     sum = sum + 0.00001;
  }
  printf("\nSum = %lf\n", sum);
}
```



- 使用單倍精確度浮點數（float）
  - 1.10036
- 使用雙倍精確度浮點數（double）
  - 1.10000
- 可以看出有效減少誤差