Integer vs. floating point arithmetic

fix point number representation:

unsigned integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
2^31	2^30	2^29	2^28	2^27	2^26	2^25	2^24	2^23	2^22	2^21	2^20	2^19	2^18	2^17	2^16

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
2^15	2^14	2^13	2^12	2^11	2^10	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

signed integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-(2^31)	2^30	2^29	2^28	2^27	2^26	2^25	2^24	2^23	2^22	2^21	2^20	2^19	2^18	2^17	2^16

sign bit

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
2^15	2^14	2^13	2^12	2^11	2^10	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

unsigned fractional format (29.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
2^28	2^27	2^26	2^25	2^24	2^23	2^22	2^21	2^20	2^19	2^18	2^17	2^16	2^15	2^14	2^13

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
2^12	2^11	2^10	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^-1	2^-2	2^-3

radix point

signed fractional format (29.3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-(2^28)	2^27	2^26	2^25	2^24	2^23	2^22	2^21	2^20	2^19	2^18	2^17	2^16	2^15	2^14	2^13

sign bit

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
2^12	2^11	2^10	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^-1	2^-2	2^-3

radix point

floating point number representation (IEEE):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SGN	E7	E6	E5	E4	E3	E2	E1	E0	M22	M21	M20	M19	M18	M17	M16

sign bit — exponent — mantissa

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
M15	M14	M13	M12	M11	M10	M9	M8	M7	M6	M5	M4	M3	M2	M1	M0

mantissa

$$\texttt{number} = (-1)^{SGN} * 1.mantissa * 2^{(\text{exponent} - 127)}$$

floating point numbers are very **convenient**, they automatically keep track of the order of magnitude of the value

- number overflows are very unlikely
- many mathematical functions can have outputs spanning several orders of magnitude
- functions and algorithms are easily written

disadvantage:

- floating point numbers are subject to ***rounding errors***.
- ***number underflow***: value maybe set to zero

if you use floating point numbers, you **must** perform a *rounding error analysis,* otherwise **unpredictable behavior may result !**

numerically stable algorithms feature a certain minimum accuracy for all possible input values and they are stable against small variations of the input values (measurement errors, noise).

source of rounding errors:

`x, y ∈ M` (`M` = set of all possible floating point numbers)

`(x + y), (x - y), (x * y), (x / y)`

are not necessarily $\in$ ***M***

rounding: find the value `v ∈ M` closest to `(x + y), (x - y),` etc.

`eps = 2^(-t)` machine accuracy (t = number of mantissa bits = 24) $\approx 10^{-7}$

`|(rd(z)-z)/z| <= 2^(-(t+1))/mantissa(z) <= 2^(-t)`

since `mantissa(z) >= 1/2`

`eps` is the ***relative rounding error***

floating point numbers do not obey the usual rules of arithmetic

example:

`rd(x+y) = x` if `|y|<eps/2 * |x|` no change in result if y is too small

if you want to add a large number of floating point numbers of different magnitude:

1. sort numbers
2. add small numbers first

warning: sorting of numbers is a time-consuming task

subtraction: cancellation of bits:

```
      0.315875
-     0.315872
--------------
      0.000003
```

if the numbers being subtracted are $\in M$ the result will be $\in M$
but: bit cancellation amplifies previous errors

with 2 digits more accuracy the same calculation could have been like this:

```
      0.315874 51
-     0.315872 43
-----------------
      0.000002 08
```

thus, the result will be wrong by about 50% !

error amplification (differential error analysis):

ε: relative error = dx/x

```
1.    x * y          ε(x*y) = ε(x) + ε(y)
2.    x / y          ε(x/y) = ε(x) - ε(y)
3.    x + y          ε(x+y) = (x * ε(x) + y * ε(y)) / (x + y) !!!
4.    x - y          ε(x-y) = (x * ε(x) - y * ε(y)) / (x - y) !!!
5.    sqrt(x)        ε(sqrt(x)) = ε(x)/2
```

summary: ***subtraction is a dangerous operation, if x ≈ y***

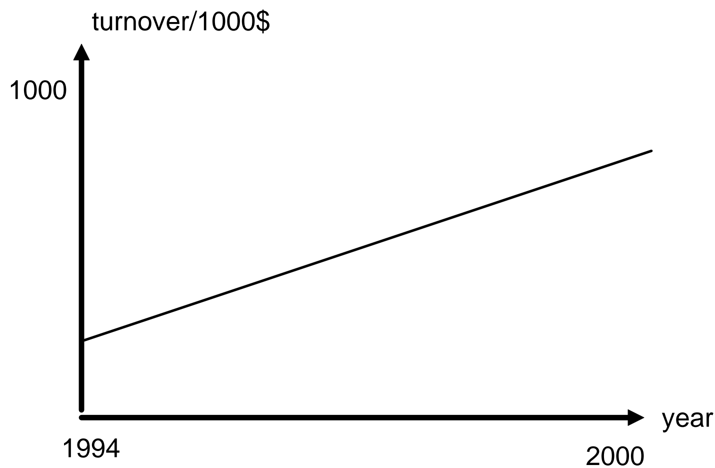
in most other cases the relative rounding error of a calculation will be approximately:

```
err ≈ N * eps
```

where `N` is the number of operations in the calculation

example 1:

linear regression:



$y = ax + b$

$$a = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2} \qquad b = \frac{\sum x_i^2 \sum y_i - \sum x_i \sum x_i y_i}{n\sum x_i^2 - (\sum x_l)^2}$$

calculation of the denominator:

	x	x^2					
	1994	3976036					
	1995	3980025					
	1996	3984016					
	1997	3988009					
			integer		**5 digit float**		**6 digit float**
sum	7982	15928086 * 4 =	63712344	=	6.37123E07	=	6.371234E07
x^2 =			63712324	=	6.37123E07	=	6.371232E07
diff			**00000020**		**0.00000E00 !!**		**2.000000E01**

diff/sum ≈ 3.2E-07

this is in the order of eps (1.0E-07) !

the above formulae are not numerically stable !

DO NOT USE THE ABOVE FORMULAE WITH FLOATING POINT !

possible solutions:

1. numerically stable algorithm (prove with rounding error analysis)

the denominator can be written as

n*var(x) = $n\sum(x_i - \bar{x})^2$

	x	$(x-\bar{x})^2$		
	1994	2.25		
	1995	0.25		
	1996	0.25		
	1997	2.25		
sum	7982	5.0 * 4 =	20.0 =	**2.0E01**
$\bar{x}$	= 7982/4		=	**1995,5**

problem: if the differences from the mean are large, ***values must be sorted before adding***

2. integer arithmetic (with appropriate accuracy)

alternative: integer arithmetic

calculation of integer accuracy:

x < 2048	11 bits		
n < 64K	16 bits	$\sum x_i$	27 bits (I32 = integer)
		x_i^2	22 bits
		$\sum x_i^2$	38 bits (I40 = long)
		$n\sum x_i^2$	54 bits (64 bits: BITN)

VC BITN arithmetic: arbitrary integer precision (e.g. 160 bits)

example 2:

matrix inversion:

Gaussian matrix inversion tends to instability if the determinant is near zero

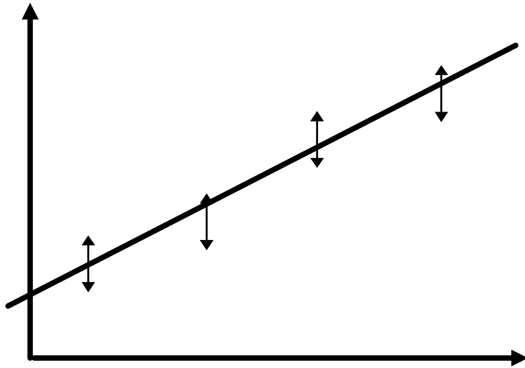
very likely for best-fit algorithms (inversion of the normal equations) although
in most cases problem is *overdetermined*

inconsistency:

overdetermined and linear dependent

better: QR decomposition (Householder)

Least squares fit of lines

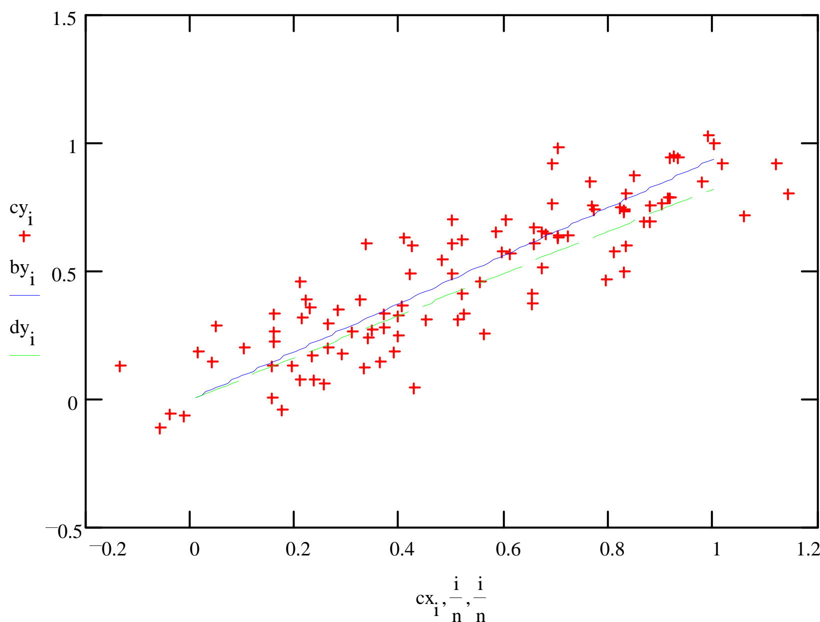


standard method: the sum of squares of the **vertical** distances line is minimized
problems, if |slope| >> 0

our method: distances **perpendicular** to line - best fit independent of slope

Test of different best-fit methods for lines

```
m          :=   0.95
noise (x) :=   0.4
noise (y) :=   0.4
```



results:

standard method:	m(std) = 0.823	Δm =	0.13
our method:	m(our) = 0.937	Δm =	0.01

example:

file: tan.c fast and precise calculation of tan and atan

```
/*******************************************************************/
/*                                                                 */
/*     atn()                                                       */
/*     calculation of atan in the range of (-1 und +1)             */
/*                                                                 */
/*******************************************************************/

float atn(float x)
{
static float a[6] =  {
                 2.0835100E-02,
                -8.5133000E-02,
                 1.8014100E-01,
                -3.3029950E-01,
                 9.9986600E-01,
               };

return(x * horner(5,x*x,a));
}

float horner(int n, float x, float *coeff)
{
int i;
float y=0.0;

for(i=0;i<n;i++)
  {
  y = y * x + *coeff++;   coeff[i]
  }
return(y);
}

void printf(float f)
{
int a,b;

a = (int) f; b = (int) ((f - (float) a) * 10000.0 );
print("%d.%04d\n",a,b);
}
```

Important things to remember:

1.) floating point calculations are subject to rounding errors
2.) for floating point calculations, a ***rounding error analysis*** is mandatory !
3.) otherwise ***unpredictable results*** are likely
4.) the most serious error is ***bit cancellation*** i.e. the subtraction of 2 numbers of almost equal magnitude
5.) double precision does ***not solve this problem***
6.) integer arithmetic is free of rounding errors

Bibliography:

Josef Stoer: Einführung in die numerische Mathematik I, Springer Verlag 1976

William H. Press et al.: Numerical Recipes in C §1.3, §5.6, §20.1

Milton Abramowitz, Irene A. Stegun: Handbook of Mathematical Functions, Dover Publications, 9th printing 1972, ISBN 0-486-61272-4