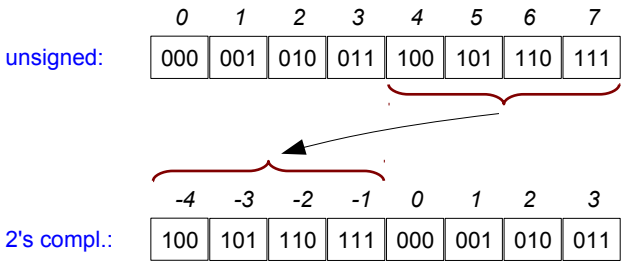
Basic things in numerics

- Presentation of numbers
- Error sources in numerical computing

Presentation of numbers

- **Integer numbers in computers**
  - Two's complement
    - Highest bit: sign (1=negative, 0=positive)
      - $i \rightarrow -i$: complement all bits, add one
      - example for an 8-bit integer
        $12_{10} = 00001100_2$
        $-12_{10} = 11110100_2$
  - Addition and subtraction with same circuitry
  - This is essentially modulo arithmetics: e.g. three-bit integers:



Presentation of numbers

- **Integer types in C**

`char`	8 bit [-128, 127]
`short int`	16 bit [-32768, 32767]
`int`	32 bit [-2147483648, 2147483647]
`long int`	32 bit [-2147483648, 2147483647](*) 64 bit [-9223372036854775808, 9223372036854775807]
`long long int`	64 bit [-9223372036854775808, 9223372036854775807] (**)

(*) On 32 bit machines (tested on 64 bit Ubuntu)

(**) C99 standard.

$9223372036854775807 \approx 9.22 \cdot 10^{18}$

Presentation of numbers

- **Integer types in Fortran90**
  - Intrinsic function **`selected_int_kind(r)`** gives an integer type with range at least $-10^r \ldots 10^r$
  - Example:

```
program integerf90
     integer,parameter :: ik1=selected_int_kind(2)
     integer,parameter :: ik2=selected_int_kind(4)
     integer,parameter :: ik3=selected_int_kind(9)
     integer,parameter :: ik4=selected_int_kind(18)
     integer(kind=ik1) :: i1
     integer(kind=ik2) :: i2
     integer(kind=ik3) :: i3
     integer(kind=ik4) :: i4
     integer :: i
     i1=1; i2=1; i3=1; i4=1
     do i=1,64
          i1=i1*2; i2=i2*2; i3=i3*2; i4=i4*2
     write(6,*) i,i1,i2,i3,i4
     end do
     stop
end program integerf90
```

Presentation of numbers

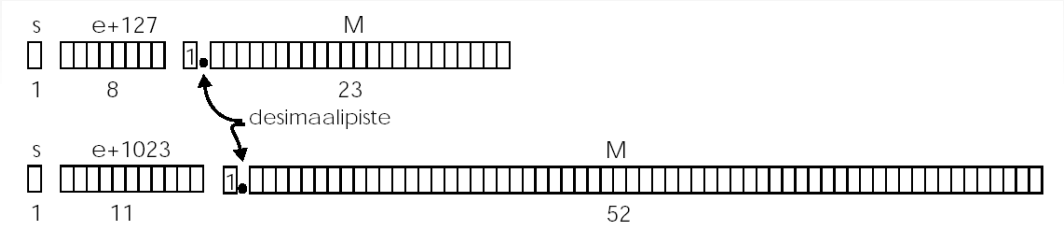
- Output

```
 1    2      2           2                    2
 2    4      4           4                    4
 3    8      8           8                    8
 4   16     16          16                   16
 5   32     32          32                   32
 6   64     64          64                   64
 7 -128    128         128                  128
 8    0    256         256                  256
 9    0    512         512                  512
10    0   1024        1024                 1024
11    0   2048        2048                 2048
12    0   4096        4096                 4096
13    0   8192        8192                 8192
14    0  16384       16384                16384
15    0 -32768       32768                32768
16    0      0       65536                65536
17    0      0      131072               131072
18    0      0      262144               262144
19    0      0      524288               524288
20    0      0     1048576              1048576
        . . .
29    0      0   536870912            536870912
30    0      0  1073741824           1073741824
31    0      0 -2147483648           2147483648
32    0      0           0           4294967296
33    0      0           0           8589934592
        . . .
59    0      0           0   576460752303423488
60    0      0           0  1152921504606846976
61    0      0           0  2305843009213693952
62    0      0           0  4611686018427387904
63    0      0           0 -9223372036854775808
64    0      0           0                    0
```

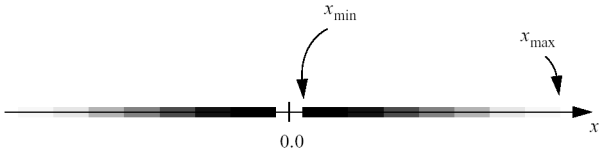
Note: Integer overflow not necessarily detected.

Presentation of numbers

- **Floating point numbers in computers (standard IEEE 745)**
  - Presented in form $a = s \times M \times 2^{e-E}$
    - s sign bit
    - M mantissa (or significand)
    - e exponent
    - E exponent offset
  - Presentation of **`float`** and **`double`**:



  - Normalized floating point numbers: 1st bit of mantissa always 1 → it can be left out (save one bit)
  - Smallest positive normalized number: $x_{min} = 2^{e_{min}}$
  - Largest positive normalized number: $x_{max} = 2^{e_{max}}$



Presentation of numbers

- **Example**
  From: www.math.nyu.edu/faculty/goodman/ teaching/SciComp2003/Book/sourcesOfError.pdf

For example, the number $2.752 \cdot 10^3 = 2572$ can be written

$$\begin{aligned} 2752 &= 2^{11} + 2^9 + 2^7 + 2^6 \\ &= 2^{11} \cdot \left(1 + 2^{-2} + 2^{-4} + 2^{-5}\right) \\ &= 2^{11} \cdot (1 + (.01)_2 + (.0001)_2 + (.00001)_2) \\ &= 2^{11} \cdot (1.01011)_2 \quad . \end{aligned}$$

Altogether, we have, using $11 = (1011)_2$,

$$2752 = +(1.01011)_2^{(1011)_2} \ .$$

Thus, we have sign $s = +$. The exponent is $e - 127 = 11$ so that $e = 138 = (10001010)_2$. The fraction is $f = (01011000000000000000000)_2$. The entire 32 bit string corresponding to $2.752 \cdot 10^3$ then is:

$$\underbrace{1}_{s} \ \underbrace{10001010}_{e} \ \underbrace{01011000000000000000000}_{f} \quad .$$

Check in Fortran90:

```
program floatingbits

  implicit none
  integer :: i,s,iargc
  character (len=32) :: chk
  character (len=80) :: argu
  real(4) :: x

  do i=1,iargc()
     call getarg(i,argu)
     read(argu,*,iostat=s) x
     if (s /= 0) exit
     write(chk,'(b32.32)') x
     write(0,'(g20.12,5x,a,x,a,x,a)') &
       & x,chk(1:1),chk(2:9),chk(10:32)
  enddo

  stop

end program floatingbits
```

```
./floatingbits 2752
   2752.00000000         0 10001010 01011000000000000000000
```


Presentation of numbers

- **Special numbers**
  - **Signed infinity** (exponent: all bits 1; mantissa: all bits 0)
    $\infty \pm x = \infty \qquad \infty \times x = \infty \qquad x/\infty = 0$
  - **NaN** (not a number) (all bits 1)
    - Results in e.g. when calling **`sqrt`** with a negative real argument or **`acos`** with argument >1.0
    - Two NaNs are never equal:

```
#include <stdio.h>
#include <math.h>
int main()
{
  float a,x,y;
  x=acos(2.0);
  a=0.0;
  y=1.0/a;
  printf("%f\n",x);
  printf("%f %f %f\n",y,y+1.0e10,1.0/y);
  if (x != x) {
    printf("Two NaNs are not equal.\n");
  }
  return 0;
}
```

```
progs> icc -o nan_inf nan_inf.c -lm
progs> ./nan_inf
nan
inf inf 0.000000
progs> gcc -o nan_inf nan_inf.c -lm
progs> ./nan_inf
nan
inf inf 0.000000
Two NaNs are not equal.
progs>
```

  - Infs and NaNs might sometimes be useful.
  - However, they indicate that something is wrong and the error should be found out by some other means.

Presentation of numbers

- Floating point number models in Kumpula Linux system

```
rk                4              8              16
             -------------  -------------  --------------
bits                32             64             128
radix                2              2               2
precision            6             15              33
digits              24             53             113
range               37            307            4931
maxexponent        128           1024           16384
minexponent       -125          -1021          -16381
epsilon     0.11920929E-06 0.22204460E-15 0.192592990E-33
tiny        0.11754944E-37 0.22250739-307 0.33621031-4931
huge        0.34028235E+39 0.17976931+309 0.11897315+4933
```

ϵ Machine epsilon: smallest positive number for which $1+\epsilon \neq 1$

When length of the mantissa is m bits $\varepsilon = 2^{-m}$.

- Code snippet to print these out

```
integer,parameter :: p=20,r=100
integer,parameter :: rk=selected_real_kind(p,r)
...
write(0,'(a,i2,a,i3,a,i2)') 'selected_real_kind(',p,',',r,')=',rk
write(0,'(a,i6)')     'radix       = ',radix(x)
write(0,'(a,i6)')     'precision   = ',precision(x)
write(0,'(a,i6)')     'digits      = ',digits(x)
write(0,'(a,i6)')     'range       = ',range(x)
write(0,'(a,i6)')     'maxexponent = ',maxexponent(x)
write(0,'(a,i6)')     'minexponent = ',minexponent(x)
write(0,'(a,g20.10)') 'epsilon     = ',epsilon(x)
write(0,'(a,g20.10)') 'tiny        = ',tiny(x)
write(0,'(a,g20.10)') 'huge        = ',huge(x)
...
```

Fortran intrinsic function `selected_real_kind(p,r)`
`p` = precision in decimal digits
`r` = range of decimal exponent

If the requested number model is not availabe -1 is returned.

Presentation of numbers

- In C you may use `long double` for 128 bit floating point numbers.
  - Need to have a C99 complying compiler
- Which number model to use?
  - As a default, use `double` (or `selected_real_kind(10,40)`, or `double precision` in Fortran).
    - There may be speed differences between `float` and `double`.
    - `double` consumes more memory (also: cache effects)
    - However, the 128-bit numbers may be *really slow* (Intel Core2 Quad CPU Q9650 @3.00GHz):

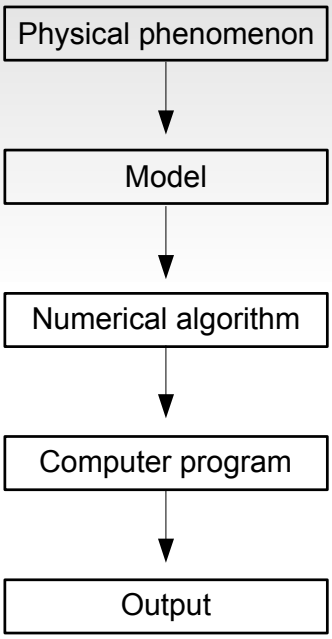
```
program realdoubletest

  implicit none
  ! integer,parameter :: RK=selected_real_kind(5,10)  ! single precision
  integer,parameter :: RK=selected_real_kind(10,40)   ! double precision
  ! integer,parameter :: RK=selected_real_kind(30,200)! quadruple precision
  integer, parameter :: N=10000000
  real(kind=RK),parameter:: dx=0.0001
  real(kind=RK) :: x,y,z
  integer :: i
  x=dx; y=0.0; z=0.0
  do i=1,N
     x=x+dx
     y=y+sin(x)
     z=z+log(x)*cos(x)
  enddo
  write(0,*) x,y,z
  stop
end program realdoubletest
```

	single	double	quadruple	
RK(=bytes)	4	8	16	
digits	24	53	113	
bits	32	64	128	
time(s)	**0.07**	**0.25**	**6.57**	**Intel**
time(s)	***3.46***	***0.91***	**--**	***GNU***

Error sources in numerical computing

- **From a physical phenomenon to a computer program**



Errors and inaccuracies possible in all stages.

Error sources in numerical computing

- **Errors may be classified as**
  - Model errors
  - Errors in initial values
  - Truncation errors
  - Roundoff error
  - Bugs
  - Hardware failures

- **Model errors** include making too drastic simplifications in building the model; e.g.
  - Ignoring air resistance in modeling free fall of a body
  - Modeling atoms as mass points obeying classical equations of motion in the quantum mechanical regime
  - Approximation of linear relationships ...
  - All these belong to the application field.

- In a numerical computation task **initial values** can have inaccuracies.
  - One should take care that the algorithm does not amplify these errors during the computation.
  - Sometimes it is not a problem: molecular dynamics simulation of an atomic systems.

Error sources in numerical computing

- **Truncation errors** (going from model to algorithm): ignoring a significant feature of the problem
  - Example: approximating derivative by the difference quotient

$$\Delta_h f(x) = \frac{f(x+h)-f(x)}{h} = \frac{f(x)+h f'(x)+\frac{h^2}{2} f''(\xi)-f(x)}{h} = f'(x)+\frac{h}{2} f''(\xi)$$

$$x < \xi < x+h$$

  - Truncation error $e_t = \frac{h}{2} f''(\xi) = O(h)$

  - Better error behavior: centered difference quotient

$$\delta_h f(x) = \frac{f(x+h)-f(x-h)}{2h}$$

Error sources in numerical computing

- **Roundoff errors** (going from algorithm to program and running the program): finite precision of floating point numbers
  - In converting (and storing) a floating point number the maximum relative error is ε/2
  - *Cancellation* in addition or subtraction ($z = x + y$, $|z| < |x| + |y|$) → error amplification

1) Ill conditioning or sensitivity of the problem
   - Example: roots of a 4th order equation

$$x^4 - 4x^3 + 8x^2 - 16x + 15.99999999 = 0$$

$$x_1 = 2.01, x_2 = 1.99, x_3 = 2 + 0.01i, x_4 = 2 - 0.01i$$

   - Floating point system with $\epsilon > 10^{-10} \Rightarrow 15.99999999 \approx 16.0$ equation reduces to:

$$(x-2)^4 (x^2+4) = 0 \Rightarrow x = 2.0$$

2) Stability of the algorithm
   - Example: Taylor series for

$$e^x = 1 + x + \frac{x^2}{2!} + \ldots$$

   - Problems when x<0: subtracting almost equal numbers (*catastrophic cancellation*)
     - Solution:

$$e^{-x} = 1/e^x$$

Error sources in numerical computing

```
program exp_taylor
  implicit none
  !integer,parameter :: rk=selected_real_kind(20,100)
  integer,parameter :: rk=selected_real_kind(10,40)
  real(kind=rk) :: t,s,x,t1,s1
  integer :: i
  s=0.0
  s1=0.0
  t=1.0
  t1=1.0
  x=-15.0
  do i=1,60
     t=t*x/i
     t1=t1*(-x)/i
     s1=s1+t1
     s=s+t
     write(0,'(i4,4g20.12)') i,t,s,t1,1.0/s1
  end do
  stop
end program exp_taylor
```

$$e^{-15} \approx 0.305902320502 \times 10^{-6}$$

```
   1  -15.0000000000      -15.0000000000       15.0000000000      0.666666666667E-01
   2   112.500000000       97.5000000000       112.500000000      0.784313725490E-02
   3  -562.500000000      -465.000000000       562.500000000      0.144927536232E-02
     . . .
  13  -312540.319178      -164597.219099       312540.319178      0.842201407875E-06
  14   334864.627691       170267.408591       334864.627691      0.656931353886E-06
  15  -334864.627691      -164597.219099       334864.627691      0.538475793884E-06
  16   313935.588460       149338.369361       313935.588460      0.460611014830E-06
  17  -277001.989818      -127663.620457       277001.989818      0.408491514071E-06
  18   230834.991515       103171.371058       230834.991515      0.373292244687E-06
  19  -182238.151196      -79066.7801380       182238.151196      0.349515381381E-06
  20   136678.613397       57611.8332588       136678.613397      0.333579845930E-06
  21  -97627.5809977      -40015.7477389       97627.5809977      0.323058917510E-06
  22   66564.2597712       26548.5120322       66564.2597712      0.316258048239E-06
  23  -43411.4737638      -16862.9617316       43411.4737638      0.311974874074E-06
     . . .
  41 -0.495807048394E-01  -1.01310725262  0.495807048394E-01  0.305902416607E-06
  42  0.177073945855E-01  -0.995399858030  0.177073945855E-01  0.305902414950E-06
  43 -0.617699811123E-02  -1.00157685614  0.617699811123E-02  0.305902414372E-06
  44  0.210579481065E-02  -0.999471061331  0.210579481065E-02  0.305902414175E-06
  45 -0.701931603548E-03  -1.00017299293  0.701931603548E-03  0.305902414109E-06
  46  0.228890740288E-03  -0.999944102194  0.228890740288E-03  0.305902414088E-06
  47 -0.730502362620E-04  -1.00001715243  0.730502362620E-04  0.305902414081E-06
  48  0.228281988319E-04  -0.999994324231  0.228281988319E-04  0.305902414079E-06
  49 -0.698822413221E-05  -1.00000131246  0.698822413221E-05  0.305902414078E-06
  50  0.209646723966E-05  -0.999999215988  0.209646723966E-05  0.305902414078E-06
  51 -0.616608011665E-06  -0.999999832596  0.616608011665E-06  0.305902414078E-06
  52  0.177867695673E-06  -0.999999654728  0.177867695673E-06  0.305902414078E-06
  53 -0.503399138696E-07  -0.999999705068  0.503399138696E-07  0.305902414078E-06
  54  0.139833094082E-07  -0.999999691085  0.139833094082E-07  0.305902414078E-06
  55 -0.381362983861E-08  -0.999999694899  0.381362983861E-08  0.305902414078E-06
  56  0.102150799248E-08  -0.999999693877  0.102150799248E-08  0.305902414078E-06
  57 -0.268817892759E-09  -0.999999694146  0.268817892759E-09  0.305902414078E-06
  58  0.695218688170E-10  -0.999999694076  0.695218688170E-10  0.305902414078E-06
  59 -0.176750513941E-10  -0.999999694094  0.176750513941E-10  0.305902414078E-06
  60  0.441876284854E-11  -0.999999694090  0.441876284854E-11  0.305902414078E-06
```


Error sources in numerical computing

- Roundoff errors: example
  - Approximating derivative by the difference quotient

$$\Delta_h f(x) = \frac{f(x+h) - f(x)}{h}$$

  - Roundoff error of the expression (assuming error only in storing the function values)

$$e_r = \frac{2|f(x)|\epsilon}{h} = O(1/h)$$

  - Total error

$$e = e_t + e_r = \frac{h}{2}|f''(\xi)| + \frac{2|f(x)|\epsilon}{h}$$

  - Find the optimum step size by minimizing the error:

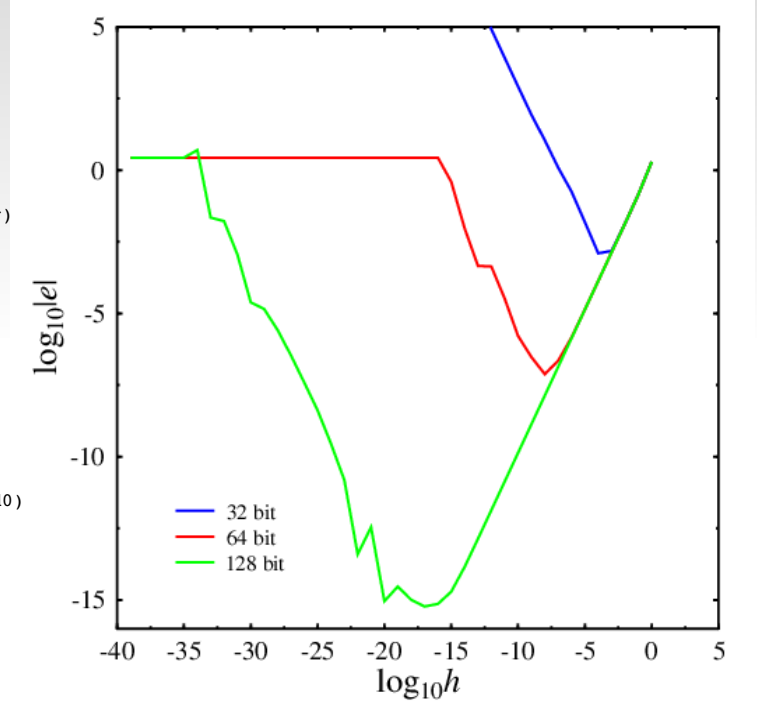
$$\frac{de}{dh} = 0 \rightarrow h = 2\sqrt{\frac{|f(x)|\epsilon}{|f''(\xi)|}} \approx \sqrt{\epsilon}$$

Error sources in numerical computing

- ”Experimental”

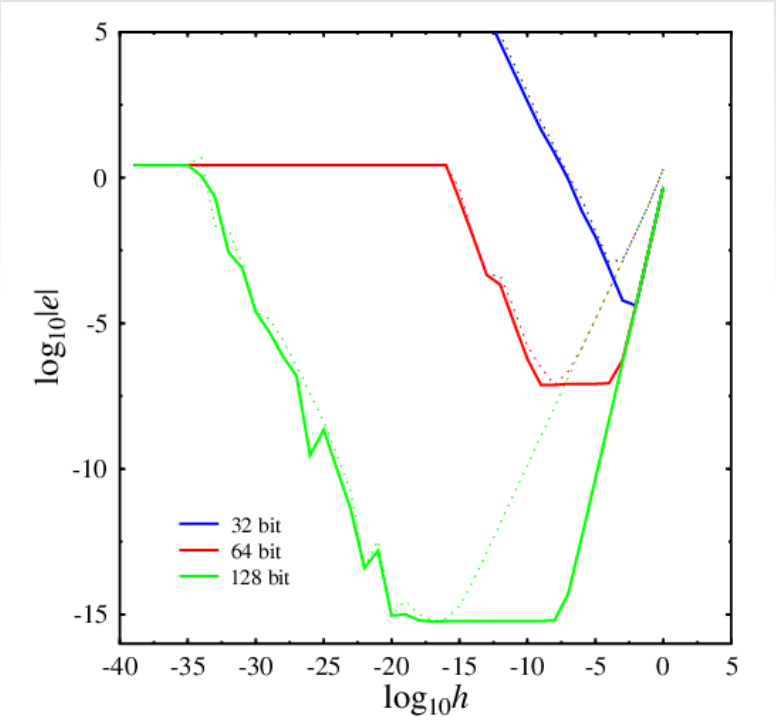
```
program derivative_error
  implicit none
!  integer,parameter :: p=2,r=10
!  integer,parameter :: p=7,r=30
  integer,parameter :: p=20,r=100
  integer,parameter :: rk=selected_real_kind(p,r)
  real(rk) :: fx,fxh,h,d,d0,x
  integer :: iargc,i,e1,e2
  character(len=80) :: argu
  x=1.0
  d0=exp(1.0)
  call getarg(1,argu); read(argu,*) e1
  call getarg(2,argu); read(argu,*) e2
  do i=e1,e2
     h=10.0**real(i,rk)
     fxh=exp(x+h)
     fx=exp(x)
     d=(fxh-fx)/h
     write(6,'(g16.8,3g25.15)') h,d,d-d0,abs(d-d0)
  end do
  stop
end program derivative_error
```

$$\sqrt{\epsilon} = \begin{cases} 3.45\times10^{-4}, 32\,bit \\ 1.49\times10^{-8}, 64\,bit \\ 1.39\times10^{-17}, 128\,bit \end{cases}$$



Error sources in numerical computing

- ”Experimental” using centered difference quotient



Error sources in numerical computing

- **Summary**
  - Ill conditioned problem
    - Answer sensitive to problem parameters and errors in computation
    - No good answers with any algorithm
  - Unstable algorithms
    - Poor results even for well posed problems due to
      - Truncation errors
      - Roundoff errors
      - Error amplification
      - Change the algorithm!
  - Test your algorithm
    - Solve a case for which you know the answer.
    - Test the sensitive to parameters: series of runs with different parameter values.
    - Check also intermediate results.
    - Use visualization!