[CSE4140] 수치 컴퓨팅 및 응용

강의 자료 1

(2012년도 2학기)

담당교수: 서강대학교 공과대학 컴퓨터공학과 임 인 성

부동 소수점 숫자를 사용하는 수치 컴퓨팅 예

(Some Examples of Numerical Computing Using Floating-Point Numbers)

Numerical Computing: Example 1

- 컴퓨터를 통한 미분
  - ‘수학의 정석’에 의하면,

$$\boxed{f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}}$$

```
#include <stdio.h>
#define SQ(a) ((a)*(a))
main()
{
  float h = 1.0, numerator, denominator, approx;
  int i;
  for (i = 0; i < 150; i++) {
    h /= 2.0;
    numerator = SQ(1.2 + h) - SQ(1.2);
    denominator = h;
    approx = numerator/denominator;
    printf("*** i = %d: h = %12e, approx = %12e₩n", i, h, approx);
  }
}
```

- Using single precision – 32bits (`float`)

```
*** i = 0: h = 5.000000e-01, approx = 2.900000e+00
*** i = 1: h = 2.500000e-01, approx = 2.650000e+00
*** i = 2: h = 1.250000e-01, approx = 2.525000e+00
*** i = 3: h = 6.250000e-02, approx = 2.462500e+00
*** i = 4: h = 3.125000e-02, approx = 2.431250e+00
*** i = 5: h = 1.562500e-02, approx = 2.415625e+00
*** i = 6: h = 7.812500e-03, approx = 2.407813e+00
*** i = 7: h = 3.906250e-03, approx = 2.403906e+00
*** i = 8: h = 1.953125e-03, approx = 2.401953e+00
*** i = 9: h = 9.765625e-04, approx = 2.400977e+00
*** i = 10: h = 4.882812e-04, approx = 2.400488e+00
*** i = 11: h = 2.441406e-04, approx = 2.400244e+00
*** i = 12: h = 1.220703e-04, approx = 2.400122e+00
*** i = 13: h = 6.103516e-05, approx = 2.400061e+00
*** i = 14: h = 3.051758e-05, approx = 2.400031e+00
*** i = 15: h = 1.525879e-05, approx = 2.400015e+00
*** i = 16: h = 7.629395e-06, approx = 2.400008e+00
*** i = 17: h = 3.814697e-06, approx = 2.400004e+00
*** i = 18: h = 1.907349e-06, approx = 2.400002e+00
*** i = 19: h = 9.536743e-07, approx = 2.400001e+00
*** i = 20: h = 4.768372e-07, approx = 2.400001e+00
*** i = 21: h = 2.384186e-07, approx = 2.400000e+00
    ***
*** i = 32: h = 1.164153e-10, approx = 2.400000e+00
*** i = 33: h = 5.820766e-11, approx = 2.399998e+00
*** i = 34: h = 2.910383e-11, approx = 2.400002e+00
*** i = 35: h = 1.455192e-11, approx = 2.399994e+00
*** i = 36: h = 7.275958e-12, approx = 2.399994e+00
*** i = 37: h = 3.637979e-12, approx = 2.399963e+00
*** i = 38: h = 1.818989e-12, approx = 2.400024e+00
*** i = 39: h = 9.094947e-13, approx = 2.399902e+00
```

```
*** i = 40: h = 4.547474e-13, approx = 2.399902e+00
*** i = 41: h = 2.273737e-13, approx = 2.399414e+00
*** i = 42: h = 1.136868e-13, approx = 2.400391e+00
*** i = 43: h = 5.684342e-14, approx = 2.398438e+00
*** i = 44: h = 2.842171e-14, approx = 2.398438e+00
*** i = 45: h = 1.421085e-14, approx = 2.390625e+00
*** i = 46: h = 7.105427e-15, approx = 2.406250e+00
*** i = 47: h = 3.552714e-15, approx = 2.375000e+00
*** i = 48: h = 1.776357e-15, approx = 2.375000e+00
*** i = 49: h = 8.881784e-16, approx = 2.250000e+00
*** i = 50: h = 4.440892e-16, approx = 2.500000e+00
*** i = 51: h = 2.220446e-16, approx = 2.000000e+00
*** i = 52: h = 1.110223e-16, approx = 4.000000e+00
*** i = 53: h = 5.551115e-17, approx = 0.000000e+00
*** i = 54: h = 2.775558e-17, approx = 0.000000e+00
    ***
*** i = 124: h = 2.350989e-38, approx = 0.000000e+00
*** i = 125: h = 1.175494e-38, approx = 0.000000e+00
*** i = 126: h = 0.000000e+00, approx =          nan
*** i = 127: h = 0.000000e+00, approx =          nan
```

- Using double precision – 64bits (double)

```
*** i = 0: h = 5.000000000000000e-01, approx = 2.899999999999999e+00
*** i = 1: h = 2.500000000000000e-01, approx = 2.650000000000000e+00
*** i = 2: h = 1.250000000000000e-01, approx = 2.525000000000000e+00
*** i = 3: h = 6.250000000000000e-02, approx = 2.462499999999999e+00
*** i = 4: h = 3.125000000000000e-02, approx = 2.431249999999999e+00
*** i = 5: h = 1.562500000000000e-02, approx = 2.415624999999991e+00
*** i = 6: h = 7.812500000000000e-03, approx = 2.407812500000006e+00
*** i = 7: h = 3.906250000000000e-03, approx = 2.403906249999977e+00
*** i = 8: h = 1.953125000000000e-03, approx = 2.401953124999977e+00
*** i = 9: h = 9.765625000000000e-04, approx = 2.400976562499864e+00
*** i = 10: h = 4.882812500000000e-04, approx = 2.400488281250091e+00
*** i = 11: h = 2.441406250000000e-04, approx = 2.400244140624636e+00
*** i = 12: h = 1.220703125000000e-04, approx = 2.400122070312136e+00
*** i = 13: h = 6.103515625000000e-05, approx = 2.400061035154067e+00
*** i = 14: h = 3.051757812500000e-05, approx = 2.400030517579580e+00
*** i = 15: h = 1.525878906250000e-05, approx = 2.400015258783242e+00
*** i = 16: h = 7.629394531250000e-06, approx = 2.400007629388710e+00
*** i = 17: h = 3.814697265625000e-06, approx = 2.400003814662341e+00
*** i = 18: h = 1.907348632812500e-06, approx = 2.400001907371916e+00
*** i = 19: h = 9.536743164062500e-07, approx = 2.400000953581184e+00
*** i = 20: h = 4.768371582031250e-07, approx = 2.400000476744026e+00
*** i = 21: h = 2.384185791015625e-07, approx = 2.400000237859786e+00
*** i = 22: h = 1.192092895507812e-07, approx = 2.400000119581819e+00
*** i = 23: h = 5.960464477539062e-08, approx = 2.400000058114529e+00
*** i = 24: h = 2.980232238769531e-08, approx = 2.400000028312206e+00
*** i = 25: h = 1.490116119384766e-08, approx = 2.400000005960464e+00
*** i = 26: h = 7.450580596923828e-09, approx = 2.400000005960464e+00
*** i = 27: h = 3.725290298461914e-09, approx = 2.399999976158142e+00
*** i = 28: h = 1.862645149230957e-09, approx = 2.399999976158142e+00
*** i = 29: h = 9.313225746154785e-10, approx = 2.399999856948853e+00
*** i = 30: h = 4.656612873077393e-10, approx = 2.400000095367432e+00
*** i = 31: h = 2.328306436538696e-10, approx = 2.399999618530273e+00
*** i = 32: h = 1.164153218269348e-10, approx = 2.399999618530273e+00
*** i = 33: h = 5.820766091346741e-11, approx = 2.399997711181641e+00
*** i = 34: h = 2.910383045673370e-11, approx = 2.400001525878906e+00
*** i = 35: h = 1.455191522836685e-11, approx = 2.399993896484375e+00
*** i = 36: h = 7.275957614183426e-12, approx = 2.399993896484375e+00
```

```
*** i = 37: h = 3.637978807091713e-12, approx = 2.399963378906250e+00
*** i = 38: h = 1.818989403545856e-12, approx = 2.400024414062500e+00
*** i = 39: h = 9.094947017729282e-13, approx = 2.399902343750000e+00
*** i = 40: h = 4.547473508864641e-13, approx = 2.399902343750000e+00
*** i = 41: h = 2.273736754432321e-13, approx = 2.399414062500000e+00
*** i = 42: h = 1.136868377216160e-13, approx = 2.400390625000000e+00
*** i = 43: h = 5.684341886080801e-14, approx = 2.398437500000000e+00
*** i = 44: h = 2.842170943040401e-14, approx = 2.398437500000000e+00
*** i = 45: h = 1.421085471520200e-14, approx = 2.390625000000000e+00
*** i = 46: h = 7.105427357601002e-15, approx = 2.406250000000000e+00
*** i = 47: h = 3.552713678800501e-15, approx = 2.375000000000000e+00
*** i = 48: h = 1.776356839400250e-15, approx = 2.375000000000000e+00
*** i = 49: h = 8.881784197001252e-16, approx = 2.250000000000000e+00
*** i = 50: h = 4.440892098500626e-16, approx = 2.500000000000000e+00
*** i = 51: h = 2.220446049250313e-16, approx = 2.000000000000000e+00
*** i = 52: h = 1.110223024625157e-16, approx = 4.000000000000000e+00
*** i = 53: h = 5.551115123125783e-17, approx = 0.000000000000000e+00
...
```

- 왜 h가 0에 수렴할 경우 컴퓨터가 구한 값은 $f'(1.2) = 2.4$에 수렴하지 않을까?
- 단지 32 비트 대신에 64 비트를 사용하여 계산한다고 문제가 해결되는가?

Numerical Computing: Example 2

- 다항식의 전개
  - 중학교 수학 시간에 배운 바에 의하면,

$$\boxed{(1+\epsilon)^3 - 1 = 3\epsilon + 3\epsilon^2 + \epsilon^3 \text{ for any } \epsilon}$$

- 컴퓨터를 사용하여 각각 양변의 값을 계산할 경우 왜 ε이 작아지면 그 값들이 서로 달라질까?

```c
/*
        CS-140 Numerical Analysis
        Insung Ihm at Sogang University

        This example shows that
          1. the mathematically same expressions could result in different values
             on computers, and
          2. the same programs could produce different results on different CPUs.
        The floating-point operations are VERY dangerous when used carelessly.
*/

#include <stdio.h>
#include <math.h>

#define EPS 1.0e-5

void single_prec(void) {
        float x, y, z;
        float epsilon = (float) EPS;
        double exact, relerror;

        exact = 3*EPS + 3*EPS*EPS + EPS*EPS*EPS;

        printf("*************************\n");
        printf("   Single Precision: EPS = %10.5e\n", EPS);
        printf("*************************\n");

        printf("^^^ Exact value = %20.15e\n", exact);
        // (1 + epsilon)^3 - 1
        x = 1.0;
        x += epsilon;
        y = x*x;
        y *= x;
        y -= 1.0;
        printf("*** (1 + epsilon)^3 - 1 = %15.6e\n", y);

        relerror = fabs((y - exact)/exact);
        printf("    Relative error = about %11.6f\n\n", relerror);

        // 3*epsilon + 3*epsilon^2 + epsilon^3
        x = 3.0;
        x *=  epsilon;

        y = 3.0;
        y *= epsilon;
        y *= epsilon;

        z = epsilon;
        z *= epsilon;
        z *= epsilon;

        x += y;
        x += z;
        printf("*** 3*epsilon + 3*epsilon^2 + epsilon^3 = %15.6e\n", x);

        relerror = fabs((x - exact)/exact);
        printf("    Relative error = about %11.6f\n\n", relerror);
}

void double_prec(void) {
        double x, y, z;
        double epsilon = EPS;
        double exact, relerror;

        exact = 3*EPS + 3*EPS*EPS + EPS*EPS*EPS;

        printf("*************************\n");
        printf("   Double Precision: EPS = %10.5e\n", EPS);
        printf("*************************\n");

        printf("^^^ Exact value = %20.15e\n", exact);
        // (1 + epsilon)^3 - 1
        x = 1.0;
        x += epsilon;
        y = x*x;
        y *= x;
        y -= 1.0;
        printf("*** (1 + epsilon)^3 - 1 = %20.15e\n", y);

        relerror = fabs((y - exact)/exact);
        printf("    Relative error = about %20.15f\n\n", relerror);

        // 3*epsilon + 3*epsilon^2 + epsilon^3
        x = 3.0;
        x *=  epsilon;

        y = 3.0;
        y *= epsilon;
        y *= epsilon;

        z = epsilon;
        z *= epsilon;
        z *= epsilon;

        x += y;
        x += z;
        printf("*** 3*epsilon + 3*epsilon^2 + epsilon^3 = %20.15e\n", x);

        relerror = fabs((x - exact)/exact);
        printf("    Relative error = about %20.15f\n\n", relerror);
}

void main(void) {
        single_prec();
        double_prec();
}

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^ MIPS R12000 CPU^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
*************************
   Single Precision: EPS = 1.00000e-05
*************************
^^^ Exact value = 3.000030000100001e-05
*** (1 + epsilon)^3 - 1 =     3.004074e-05
    Relative error = about     0.001348

*** 3*epsilon + 3*epsilon^2 + epsilon^3 =     3.000030e-05
    Relative error = about     0.000000

*************************
   Double Precision: EPS = 1.00000e-05
*************************
^^^ Exact value = 3.000030000100001e-05
*** (1 + epsilon)^3 - 1 = 3.000030000110954e-05
    Relative error = about     0.000000000003651
```

```

*** 3*epsilon + 3*epsilon^2 + epsilon^3 = 3.000030000100001e-05
    Relative error = about     0.000000000000000

====================================================

*************************
   Single Precision: EPS = 1.00000e-06
*************************
^^^ Exact value = 3.000003000001000e-06
*** (1 + epsilon)^3 - 1 =    2.861023e-06
    Relative error = about     0.046327

*** 3*epsilon + 3*epsilon^2 + epsilon^3 =    3.000003e-06
    Relative error = about     0.000000

*************************
   Double Precision: EPS = 1.00000e-06
*************************
^^^ Exact value = 3.000003000001000e-06
*** (1 + epsilon)^3 - 1 = 3.000002999797857e-06
    Relative error = about     0.000000000067714

*** 3*epsilon + 3*epsilon^2 + epsilon^3 = 3.000003000001000e-06
    Relative error = about     0.000000000000000

====================================================

*************************
   Single Precision: EPS = 1.00000e-07
*************************
^^^ Exact value = 3.000000300000010e-07
*** (1 + epsilon)^3 - 1 =    3.576279e-07
    Relative error = about     0.192093

*** 3*epsilon + 3*epsilon^2 + epsilon^3 =    3.000000e-07
    Relative error = about     0.000000

*************************
   Double Precision: EPS = 1.00000e-07
*************************
^^^ Exact value = 3.000000300000010e-07
*** (1 + epsilon)^3 - 1 = 3.000000301511818e-07
    Relative error = about     0.000000000503936

*** 3*epsilon + 3*epsilon^2 + epsilon^3 = 3.000000300000010e-07
    Relative error = about     0.000000000000000

====================================================

*************************
   Single Precision: EPS = 1.00000e-08
*************************
^^^ Exact value = 3.000000030000000e-08
*** (1 + epsilon)^3 - 1 =    0.000000e+00
    Relative error = about     1.000000

*** 3*epsilon + 3*epsilon^2 + epsilon^3 =    3.000000e-08
    Relative error = about     0.000000

*************************
   Double Precision: EPS = 1.00000e-08
*************************
```

```
^^^ Exact value = 3.000000030000000e-08
*** (1 + epsilon)^3 - 1 = 3.000000003972048e-08
    Relative error = about     0.000000008675984

*** 3*epsilon + 3*epsilon^2 + epsilon^3 = 3.000000030000000e-08
    Relative error = about     0.000000000000000



^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^ Intel PIII CPU^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
*************************
   Single Precision: EPS = 1.00000e-007
*************************
^^^ Exact value = 3.000000300000010e-007
*** (1 + epsilon)^3 - 1 =    3.384186e-007
    Relative error = about     0.128062

*** 3*epsilon + 3*epsilon^2 + epsilon^3 =    3.000000e-007
    Relative error = about     0.000000

*************************
   Double Precision: EPS = 1.00000e-007
*************************
^^^ Exact value = 3.000000300000010e-007
*** (1 + epsilon)^3 - 1 = 3.000000301511818e-007
    Relative error = about     0.000000000503936

*** 3*epsilon + 3*epsilon^2 + epsilon^3 = 3.000000300000010e-007
    Relative error = about     0.000000000000000

==================================================

*************************
   Single Precision: EPS = 1.00000e-008
*************************
^^^ Exact value = 3.000000030000000e-008
*** (1 + epsilon)^3 - 1 =    1.000000e-008
    Relative error = about     0.666667

*** 3*epsilon + 3*epsilon^2 + epsilon^3 =    3.000000e-008
    Relative error = about     0.000000

*************************
   Double Precision: EPS = 1.00000e-008
*************************
^^^ Exact value = 3.000000030000000e-008
*** (1 + epsilon)^3 - 1 = 3.000000003972048e-008
    Relative error = about     0.000000008675984

*** 3*epsilon + 3*epsilon^2 + epsilon^3 = 3.000000030000000e-008
    Relative error = about     0.000000000000000
```

Numerical Computing: Example 3

- 이차 방정식에 대한 근의 공식의 계산
  - 중학교 수학 시간에 배운 바에 의하면,

$$\boxed{0.5x^2 + bx + c = 0 \longrightarrow x = -b \pm \sqrt{b^2 - 2c}}$$

    - 이 중 값이 큰 근에 대하여,

$$\boxed{-b + \sqrt{b^2 - 2c} = \frac{-2c}{b + \sqrt{b^2 - 2c}}}$$

- 컴퓨터를 사용하여 위 식의 각 변의 값을 계산할 경우, 어떤 *b*, *c* 값에 대해서는 양변의 값이 서로 달라지는데 그 이유는 무엇일까?
- 이 두 식 중 어떤 식을 사용하는 것이 수치적으로 더 안전할까?
- 왜 왼쪽의 수식을 어떻게 프로그래밍하는가에 따라 그 결과가 다르게 나오곤 할까?

```c
/*
        CS-140 Numerical Analysis
        Insung Ihm at Sogang University

        This example shows that
          1. the mathematically same expressions could result in different values
             on computers, and
          2. the same programs could produce different results on different CPUs.
        The floating-point operations are VERY dangerous when used carelessly.
*/

#include <stdio.h>
#include <math.h>

void example(float b, float c) {
        float w, x, y, z;
        double better;

        printf("****************************************\n");
        printf("  b = %15.6e,  c = %15.6e\n", b, c);
        printf("****************************************\n");

        // Double Prec.
        better = -2.0*c/(b + sqrt(b*b - 2.0*c));
        printf("*** 1. Double better: -2c/(b+sqrt(b^2 - 2c)) = %20.15e\n\n", better);

        // Single Prec: Left side 1
        x = -b;
        y = b*b;
        y -= 2.0*c;
        z = sqrt(y);
        x += z;
        printf("*** 2. Single left 1: -b+sqrt(b^2 - 2c) = %15.6e\n", x);

        // Single Prec: Right side 1
        x = b;
        y = b*b;
        y -= 2.0*c;
        z = sqrt(y);
        x += z;
        w = -2*c;
        w /= x;
        printf("*** 3. Single right 1: -2c/(b+sqrt(b^2 - 2c)) = %15.6e\n\n", w);

        // Single Prec.: Left side 2
        x = -b + sqrt(b*b - 2.0*c);
        printf("*** 4. Single left 2: -b+sqrt(b^2 - 2c) = %15.6e\n", x);

        // Single Prec.: Right side 2
        w = -2.0*c/(b + sqrt(b*b - 2.0*c));
        printf("*** 5. Single right 2: -2c/(b+sqrt(b^2 - 2c)) = %15.6e\n", w);

        printf("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n\n");
}


void main(void) {
        example((float) 10.0, (float) 1.03);
        example((float) 100.0, (float) 1.03);
        example((float) 500.0, (float) 1.03);
        example((float) 1000.0, (float) 1.03);
        example((float) 10000.0, (float) 1.03);
}
```

```
****************************************
  b =    1.000000e+01,  c =    1.030000e+00
****************************************
*** 1. Double better: -2c/(b+sqrt(b^2 - 2c)) = -1.035359821186406e-01

*** 2. Single left 1: -b+sqrt(b^2 - 2c) =    -1.035357e-01
*** 3. Single right 1: -2c/(b+sqrt(b^2 - 2c)) =    -1.035360e-01

*** 4. Single left 2: -b+sqrt(b^2 - 2c) =    -1.035360e-01
*** 5. Single right 2: -2c/(b+sqrt(b^2 - 2c)) =    -1.035360e-01
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

****************************************
  b =    1.000000e+02,  c =    1.030000e+00
****************************************
*** 1. Double better: -2c/(b+sqrt(b^2 - 2c)) = -1.030053021851162e-02

*** 2. Single left 1: -b+sqrt(b^2 - 2c) =    -1.029968e-02
*** 3. Single right 1: -2c/(b+sqrt(b^2 - 2c)) =    -1.030053e-02

*** 4. Single left 2: -b+sqrt(b^2 - 2c) =    -1.030053e-02
*** 5. Single right 2: -2c/(b+sqrt(b^2 - 2c)) =    -1.030053e-02
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

****************************************
  b =    5.000000e+02,  c =    1.030000e+00
****************************************
*** 1. Double better: -2c/(b+sqrt(b^2 - 2c)) = -2.060004186396789e-03

*** 2. Single left 1: -b+sqrt(b^2 - 2c) =    -2.075195e-03
*** 3. Single right 1: -2c/(b+sqrt(b^2 - 2c)) =    -2.060004e-03

*** 4. Single left 2: -b+sqrt(b^2 - 2c) =    -2.060004e-03
*** 5. Single right 2: -2c/(b+sqrt(b^2 - 2c)) =    -2.060004e-03
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

****************************************
  b =    1.000000e+03,  c =    1.030000e+00
****************************************
*** 1. Double better: -2c/(b+sqrt(b^2 - 2c)) = -1.030000501840288e-03

*** 2. Single left 1: -b+sqrt(b^2 - 2c) =    -1.037598e-03
*** 3. Single right 1: -2c/(b+sqrt(b^2 - 2c)) =    -1.030001e-03

*** 4. Single left 2: -b+sqrt(b^2 - 2c) =    -1.030001e-03
*** 5. Single right 2: -2c/(b+sqrt(b^2 - 2c)) =    -1.030001e-03
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

****************************************
  b =    1.000000e+04,  c =    1.030000e+00
****************************************
*** 1. Double better: -2c/(b+sqrt(b^2 - 2c)) = -1.029999976694270e-04

*** 2. Single left 1: -b+sqrt(b^2 - 2c) =     0.000000e+00
*** 3. Single right 1: -2c/(b+sqrt(b^2 - 2c)) =    -1.030000e-04

*** 4. Single left 2: -b+sqrt(b^2 - 2c) =    -1.030000e-04
*** 5. Single right 2: -2c/(b+sqrt(b^2 - 2c)) =    -1.030000e-04
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

```
        TITLE   C:\home\ihm\Work\Teaching\Cs140\02\loimU[@K~\rootformula\m.cpp
        .386P
include listing.inc
if @Version gt 510
.model FLAT
else
_TEXT   SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT DWORD USE32 PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT DWORD USE32 PUBLIC 'BSS'
_BSS    ENDS
$$SYMBOLS       SEGMENT BYTE USE32 'DEBSYM'
$$SYMBOLS       ENDS
$$TYPES SEGMENT BYTE USE32 'DEBTYP'
$$TYPES ENDS
_TLS    SEGMENT DWORD USE32 PUBLIC 'TLS'
_TLS    ENDS
;       COMDAT ??_C@_0CP@FJNA@?$CK?$CK?$CK?5Single?5left?51?3?5?9b?$CLsqrt?$CIb?$F02?5?9@
CONST   SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST   ENDS
;       COMDAT ??_C@_0CP@FGMI@?$CK?$CK?$CK?5Single?5left?52?3?5?9b?$CLsqrt?$CIb?$F02?5?9@
CONST   SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST   ENDS
;       COMDAT _main
_TEXT   SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT   ENDS
FLAT    GROUP _DATA, CONST, _BSS
        ASSUME  CS: FLAT, DS: FLAT, SS: FLAT
endif
PUBLIC  _main
PUBLIC  ??_C@_0CP@FJNA@?$CK?$CK?$CK?5Single?5left?51?3?5?9b?$CLsqrt?$CIb?$F02?5?9@ ; 'string'
PUBLIC  ??_C@_0CP@FGMI@?$CK?$CK?$CK?5Single?5left?52?3?5?9b?$CLsqrt?$CIb?$F02?5?9@ ; 'string'
EXTRN   _printf:NEAR
EXTRN   _sqrt:NEAR
EXTRN   __chkesp:NEAR
EXTRN   __fltused:NEAR
;       COMDAT ??_C@_0CP@FJNA@?$CK?$CK?$CK?5Single?5left?51?3?5?9b?$CLsqrt?$CIb?$F02?5?9@
; File C:\home\ihm\Work\Teaching\Cs140\02\loimU[@K~\rootformula\m.cpp
CONST   SEGMENT
??_C@_0CP@FJNA@?$CK?$CK?$CK?5Single?5left?51?3?5?9b?$CLsqrt?$CIb?$F02?5?9@ DB '*'
        DB      '** Single left 1: -b+sqrt(b^2 - 2c) = %15.6e', 0aH, 00H ; 'string'
CONST   ENDS
;       COMDAT ??_C@_0CP@FGMI@?$CK?$CK?$CK?5Single?5left?52?3?5?9b?$CLsqrt?$CIb?$F02?5?9@
CONST   SEGMENT
??_C@_0CP@FGMI@?$CK?$CK?$CK?5Single?5left?52?3?5?9b?$CLsqrt?$CIb?$F02?5?9@ DB '*'
        DB      '** Single left 2: -b+sqrt(b^2 - 2c) = %15.6e', 0aH, 00H ; 'string'
CONST   ENDS
;       COMDAT _main
_TEXT   SEGMENT
_b$ = -4
_c$ = -8
_x$ = -16
_y$ = -20
_z$ = -24
_main   PROC NEAR                                       ; COMDAT

; 27    : void main(void) {

        push    ebp
        mov     ebp, esp
```

```
        sub     esp, 104                                ; 00000068H
        push    ebx
        push    esi
        push    edi
        lea     edi, DWORD PTR [ebp-104]
        mov     ecx, 26                                 ; 0000001aH
        mov     eax, -858993460                         ; ccccccccH
        rep stosd

; 28    :       float b, c;
; 29    :       float w, x, y, z;
; 30    :       double better;
; 31    :
; 32    :       b = 1000.0; c = 1.03;

        mov     DWORD PTR _b$[ebp], 1148846080          ; 447a0000H
        mov     DWORD PTR _c$[ebp], 1065604874          ; 3f83d70aH

; 33    :       // Single Prec: Left side 1
; 34    :       x = -b;

        fld     DWORD PTR _b$[ebp]
        fchs
        fstp    DWORD PTR _x$[ebp]

; 35    :       y = b*b;

        fld     DWORD PTR _b$[ebp]
        fmul    DWORD PTR _b$[ebp]
        fst     DWORD PTR _y$[ebp]

; 36    :       y -= 2.0*c;

        fld     DWORD PTR _c$[ebp]
        fadd    ST(0), ST(0)
        fsubp   ST(1), ST(0)
        fst     DWORD PTR _y$[ebp]

; 37    :       z = sqrt(y);

        sub     esp, 8
        fstp    QWORD PTR [esp]
        call    _sqrt
        add     esp, 8
        fstp    DWORD PTR _z$[ebp]

; 38    :       x += z;

        fld     DWORD PTR _x$[ebp]
        fadd    DWORD PTR _z$[ebp]
        fst     DWORD PTR _x$[ebp]

; 39    :       printf("*** Single left 1: -b+sqrt(b^2 - 2c) = %15.6e\n", x);

        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET FLAT:??_C@_0CP@FJNA@?$CK?$CK?$CK?5Single?5left?51?3?5?9b?$CLsqrt?$CIb?$F
5?9@ ; 'string'
        call    _printf
        add     esp, 12                                 ; 0000000cH

; 40    :
; 41    :       // Single Prec.: Left side 2
```

```
; 42    :        x = -b + sqrt(b*b - 2.0*c);

        fld     DWORD PTR _b$[ebp]
        fchs
        fstp    QWORD PTR -40+[ebp]
        fld     DWORD PTR _b$[ebp]
        fmul    DWORD PTR _b$[ebp]
        fld     DWORD PTR _c$[ebp]
        fadd    ST(0), ST(0)
        fsubp   ST(1), ST(0)
        sub     esp, 8
        fstp    QWORD PTR [esp]
        call    _sqrt
        add     esp, 8
        fadd    QWORD PTR -40+[ebp]
        fst     DWORD PTR _x$[ebp]

; 43    :        printf("*** Single left 2: -b+sqrt(b^2 - 2c) = %15.6e\n", x);

        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET FLAT:??_C@_0CP@FGMI@?$CK?$CK?$CK?5Single?5left?52?3?5?9b?$CLsqrt?$CIb?$FO2?
5?9@ ; 'string'
        call    _printf
        add     esp, 12                                 ; 0000000cH

; 44    : }

        pop     edi
        pop     esi
        pop     ebx
        add     esp, 104                                ; 00000068H
        cmp     ebp, esp
        call    __chkesp
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
END
```

Numerical Computing: Example 4

- 부동 소수점 숫자의 저장
  - Little-endian vs Big-endian 문제

```
/*
	CS-140 Numerical Analysis
	Insung Ihm at Sogang University

	IEEE Standard 754-1985 for Binary Floating Point Arithmetic Example

	Hex 41 60 00 00 = 1.11(binary)*2^3 = 14(decimal)
*/

#include <stdio.h>

void main(void) {
	float *f;
	unsigned int i;
	unsigned char c[4], *d;

	c[0] = 0x41; /* 65 */
	c[1] = 0x60; /* 96 */
	c[2] = 0x00;
	c[3] = 0x00;
	f = (float *) c;
	printf("Print c: %e \n", *f);

	i = 0x41600000;
	f = (float *) &i;
	printf("Print i: %e\n", *f);

	d = (unsigned char *) &i;
	printf("Stored i: %u %u %u %u\n", d[0], d[1], d[2], d[3]);
}

== MIPS R12000 CPU on UNIX ==
Print c: 1.400000e+01
Print i: 1.400000e+01
Stored i: 65 96 0 0

== INTEL PIII CPU on Windows ==
Print c: 3.452940e-041
Print i: 1.400000e+001
Stored i: 0 0 96 65
```

Numerical Computing: Example 5

- CPU에 따른 실수에 대한 저장 값 차이

> - 왜 컴퓨터는 1.1과 같은 단순한 숫자조차 정확하게 표현을 하지 못할까?
> - 왜 사용하는 CPU에 따라 동일한 실수에 대하여 실제로 저장한 내용이 다를까?

- 컴파일러 옵션에 따른 부동 소수점 연산 결과의 불확실성
  - Disable(Debug) 옵션 vs Maximize Speed 옵션

> - 소프트웨어 개발 시 디버그 모드에서 정확한 수치 계산 결과를 확인한 후, 최종적으로 컴파일러 옵션을 사용하여 코드를 최적화하여 수행하면 수치적으로 다른 결과 값이 나오는 경험을 해본 적이 있는가?

```c
/*
        CS-140 Numerical Analysis
        Insung Ihm at Sogang University

        IEEE Standard 754-1985 for Binary Floating Point Arithmetic: Examples
*/

#include <stdio.h>

double func(int n) {
        int i;
        double res = 1.0;

        for (i = 0; i < n; i++) res /= 2.0;
        return res;
}

void main(void) {
        float *f, g, x, y, z;
        unsigned int i, j;
        unsigned char c[4], *d;

        printf("*** Size of float = %dbytes\n", sizeof(float));
        printf("*** Size of double = %dbytes\n", sizeof(double));
        printf("*** Size of long double = %dbytes\n\n", sizeof(long double));

        c[0] = 0x41; /* 65 */
        c[1] = 0x60; /* 96 */
        c[2] = 0x00;
        c[3] = 0x00;
        f = (float *) c;
        printf("^^^ [0x41 0x60 0x00 0x00] = %20.15e\n", *f);

        i = 0x41600000;
        f = (float *) &i;
        printf("^^^ 0x41600000 = %20.15e\n", *f);

        d = (unsigned char *) &i;
        printf("^^^ Stored [0x41 0x60 0x00 0x00]: [%2x %2x %2x %2x]\n", d[0], d[1], d[2], d[3]);

        i = 0xc1600000;
        f = (float *) &i;
        printf("^^^ 0xc1600000 = %20.15e\n", *f);

        i = 0xff800000;
        f = (float *) &i;
        printf("^^^ 0xff800000 = %20.15e\n", *f);

        i = 0x7f800100;
        f = (float *) &i;
        printf("^^^ 0x7f800100 = %20.15e\n", *f);

        i = 0x41600001;
        f = (float *) &i;
        printf("^^^ 0x41600001 = %20.15e\n", *f);

        g = 1.0;
        d = (unsigned char *) &g;
        printf("^^^ 1.0 = [%u %u %u %u]\n", d[0], d[1], d[2], d[3]);

        for (j = 20; j < 26; j++) {
            g = 1.0 + func(j);
          printf("^^^ 1.0 + 2^{-%2d} = [%u %u %u %u]: %20.15e\n",
                           j, d[0], d[1], d[2], d[3], g);
        }

        g = 1.0 + func(24); g += func(24);
        printf("^^^ (1.0 + 2^{-24}) + 2^{-24} = [%u %u %u %u]: %20.15e\n",
                d[0], d[1], d[2], d[3], g);

        g = func(24) + func(24); g += 1.0;
        printf("^^^ 1.0 + (2^{-24} + 2^{-24}) = [%u %u %u %u]: %20.15e\n",
                d[0], d[1], d[2], d[3], g);

        g = 1.1;
        d = (unsigned char *) &g;
        printf("^^^ 1.1 = [%2x %2x %2x %2x]: %20.15e\n", d[0], d[1], d[2], d[3], g);

        x = 123456.7890;
        d = (unsigned char *) &x;
        printf("^^^ 123456.7890 = %20.15e: [%2x %2x %2x %2x]\n",
                x, d[0], d[1], d[2], d[3]);

        y = x + g;
        d = (unsigned char *) &y;
        printf("^^^ 123457.8890 = %20.15e: [%2x %2x %2x %2x]\n",
                y, d[0], d[1], d[2], d[3]);

        z = y - x;
        d = (unsigned char *) &z;
        printf("^^^ 1.1 = %20.15e: [%2x %2x %2x %2x]\n",
                z, d[0], d[1], d[2], d[3]);
}

&&& MIPS R12000 Unix &&&

*** Size of float = 4bytes
*** Size of double = 8bytes
*** Size of long double = 16bytes

^^^ [0x41 0x60 0x00 0x00] = 1.400000000000000e+01
^^^ 0x41600000 = 1.400000000000000e+01
^^^ Stored [0x41 0x60 0x00 0x00]: [41 60  0  0]
^^^ 0xc1600000 = -1.400000000000000e+01
^^^ 0xff800000 =                  -inf
^^^ 0x7f800100 =                   nan
^^^ 0x41600001 = 1.400000095367432e+01
^^^ 1.0 = [63 128 0 0]
^^^ 1.0 + 2^{-20} = [63 128 0 8]: 1.000000953674316e+00
^^^ 1.0 + 2^{-21} = [63 128 0 4]: 1.000000476837158e+00
^^^ 1.0 + 2^{-22} = [63 128 0 2]: 1.000000238418579e+00
^^^ 1.0 + 2^{-23} = [63 128 0 1]: 1.000000119209290e+00
^^^ 1.0 + 2^{-24} = [63 128 0 0]: 1.000000000000000e+00
^^^ 1.0 + 2^{-25} = [63 128 0 0]: 1.000000000000000e+00
^^^ (1.0 + 2^{-24}) + 2^{-24} = [63 128 0 0]: 1.000000000000000e+00
^^^ 1.0 + (2^{-24} + 2^{-24}) = [63 128 0 1]: 1.000000119209290e+00
^^^ 1.1 = [3f 8c cc cd]: 1.100000023841858e+00
^^^ 123456.7890 = 1.234567890625000e+05: [47 f1 20 65]
^^^ 123457.8890 = 1.234578906250000e+05: [47 f1 20 f2]
^^^ 1.1 = 1.101562500000000e+00: [3f 8d  0  0]

&&& Intel PIII on Windows with Release C/C++ Optimization = Disable(Debug) &&&

*** Size of float = 4bytes
*** Size of double = 8bytes
*** Size of long double = 8bytes
```

```
^^^ [0x41 0x60 0x00 0x00] = 3.452939545942782e-041
^^^ 0x41600000 = 1.400000000000000e+001
^^^ Stored [0x41 0x60 0x00 0x00]: [ 0  0 60 41]
^^^ 0xc1600000 = -1.400000000000000e+001
^^^ 0xff800000 = -1.#INF00000000000e+000
^^^ 0x7f800100 = 1.#QNAN0000000000e+000
^^^ 0x41600001 = 1.400000095367432e+001
^^^ 1.0 = [0 0 128 63]
^^^ 1.0 + 2^{-20} = [8 0 128 63]: 1.000000953674316e+000
^^^ 1.0 + 2^{-21} = [4 0 128 63]: 1.000000476837158e+000
^^^ 1.0 + 2^{-22} = [2 0 128 63]: 1.000000238418579e+000
^^^ 1.0 + 2^{-23} = [1 0 128 63]: 1.000000119209290e+000
^^^ 1.0 + 2^{-24} = [0 0 128 63]: 1.000000059604645e+000
^^^ 1.0 + 2^{-25} = [0 0 128 63]: 1.000000029802322e+000
^^^ (1.0 + 2^{-24}) + 2^{-24} = [1 0 128 63]: 1.000000119209290e+000
^^^ 1.0 + (2^{-24} + 2^{-24}) = [1 0 128 63]: 1.000000119209290e+000
^^^ 1.1 = [cd cc 8c 3f]: 1.100000023841858e+000
^^^ 123456.7890 = 1.234567890625000e+005: [65 20 f1 47]
^^^ 123457.8890 = 1.234578906250000e+005: [f2 20 f1 47]
^^^ 1.1 = 1.101562500000000e+000: [ 0  0 8d 3f]a

&&& Intel PIII on Windows with Release C/C++ Optimization = Maximize Speed &&&

*** Size of float = 4bytes
*** Size of double = 8bytes
*** Size of long double = 8bytes

^^^ [0x41 0x60 0x00 0x00] = 3.452939545942782e-041
^^^ 0x41600000 = 1.400000000000000e+001
^^^ Stored [0x41 0x60 0x00 0x00]: [ 0  0 60 41]
^^^ 0xc1600000 = -1.400000000000000e+001
^^^ 0xff800000 = -1.#INF00000000000e+000
^^^ 0x7f800100 = 1.#QNAN0000000000e+000
^^^ 0x41600001 = 1.400000095367432e+001
^^^ 1.0 = [0 0 128 63]
^^^ 1.0 + 2^{-20} = [8 0 128 63]: 1.000000953674316e+000
^^^ 1.0 + 2^{-21} = [4 0 128 63]: 1.000000476837158e+000
^^^ 1.0 + 2^{-22} = [2 0 128 63]: 1.000000238418579e+000
^^^ 1.0 + 2^{-23} = [1 0 128 63]: 1.000000119209290e+000
^^^ 1.0 + 2^{-24} = [0 0 128 63]: 1.000000059604645e+000
^^^ 1.0 + 2^{-25} = [0 0 128 63]: 1.000000029802322e+000
^^^ (1.0 + 2^{-24}) + 2^{-24} = [0 0 128 63]: 1.000000059604645e+000
^^^ 1.0 + (2^{-24} + 2^{-24}) = [1 0 128 63]: 1.000000119209290e+000
^^^ 1.1 = [cd cc 8c 3f]: 1.100000023841858e+000
^^^ 123456.7890 = 1.234567890625000e+005: [65 20 f1 47]
^^^ 123457.8890 = 1.234578906250000e+005: [f2 20 f1 47]
^^^ 1.1 = 1.100000023841858e+000: [cd cc 8c 3f]
```

```
=== Optimization Option = Disable(Debug) ===
; 74   :
; 75   :          g = 1.1;

        mov     DWORD PTR _g$[ebp], 1066192077          ; 3f8ccccdH

; 76   :          d = (unsigned char *) &g;

        lea     eax, DWORD PTR _g$[ebp]
        mov     DWORD PTR _d$[ebp], eax

; 77   :          printf("^^^ 1.1 = [%2x %2x %2x %2x]: %20.15e\n", d[0], d[1], d[2], d[3], g);

        fld     DWORD PTR _g$[ebp]
        ...
        call    _printf
        add     esp, 28                                 ; 0000001cH

; 78   :
; 79   :          x = 123456.7890;

        mov     DWORD PTR _x$[ebp], 1206984805          ; 47f12065H

; 80   :          d = (unsigned char *) &x;

        lea     eax, DWORD PTR _x$[ebp]
        mov     DWORD PTR _d$[ebp], eax

; 81   :          printf("^^^ 123456.7890 = %20.15e: [%2x %2x %2x %2x]\n",
; 82   :                  x, d[0], d[1], d[2], d[3]);

        ...

; 83   :
; 84   :
; 85   :          y = x + g;

        fld     DWORD PTR _x$[ebp]
        fadd    DWORD PTR _g$[ebp]
        fstp    DWORD PTR _y$[ebp]

; 86   :          d = (unsigned char *) &y;

        lea     eax, DWORD PTR _y$[ebp]
        mov     DWORD PTR _d$[ebp], eax

; 87   :          printf("^^^ 123457.8890 = %20.15e: [%2x %2x %2x %2x]\n",
; 88   :                  y, d[0], d[1], d[2], d[3]);

        ...

; 89   :
; 90   :          z = y - x;

        fld     DWORD PTR _y$[ebp]
        fsub    DWORD PTR _x$[ebp]
        fstp    DWORD PTR _z$[ebp]

; 91   :          d = (unsigned char *) &z;

        lea     eax, DWORD PTR _z$[ebp]
        mov     DWORD PTR _d$[ebp], eax

; 92   :          printf("^^^ 1.1 = %20.15e: [%2x %2x %2x %2x]\n",
; 93   :                  z, d[0], d[1], d[2], d[3]);

        ...


=== Optimization Option = Maximize Speed ===
; 74   :
; 75   :          g = 1.1;

        mov     DWORD PTR _g$[esp+52], 1066192077       ; 3f8ccccdH

; 76   :          d = (unsigned char *) &g;
; 77   :          printf("^^^ 1.1 = [%2x %2x %2x %2x]: %20.15e\n", d[0], d[1], d[2], d[3], g);

        ...

; 78   :
; 79   :          x = 123456.7890;

        mov     DWORD PTR _x$[esp+80], 1206984805       ; 47f12065H

; 80   :          d = (unsigned char *) &x;
; 81   :          printf("^^^ 123456.7890 = %20.15e: [%2x %2x %2x %2x]\n",
; 82   :                  x, d[0], d[1], d[2], d[3]);

        ...

; 83   :
; 84   :
; 85   :          y = x + g;

        mov     DWORD PTR _y$[esp+108], 1206984946      ; 47f120f2H

; 86   :          d = (unsigned char *) &y;
; 87   :          printf("^^^ 123457.8890 = %20.15e: [%2x %2x %2x %2x]\n",
; 88   :                  y, d[0], d[1], d[2], d[3]);

        ...

; 89   :
; 90   :          z = y - x;

        mov     DWORD PTR _z$[esp+52], 1066192077       ; 3f8ccccdH

; 91   :          d = (unsigned char *) &z;
; 92   :          printf("^^^ 1.1 = %20.15e: [%2x %2x %2x %2x]\n",
; 93   :                  z, d[0], d[1], d[2], d[3]);

        ...
```

Numerical Computing: Example 6

- 왜 이 두 함수의 결과가 다를 수가 있을까?
- 항상 n번이 수행되게 하려면 어떻게 하면 될까?

```
void CASE1(void) {
    srand(11301);
    for (int i = 0; i < 10; i++) {
            float x = 100*rand() + 2;
            int n = 20, k = 0;
            float dy = x/n;

            for (float y = 0; y < x; y += dy) { k++; }
            printf("*** x = %f, dy = %f, n = %d, k = %d\n", x, dy, n, k);
    }
}

void CASE2(void) {
    srand(11301);
    for (int i = 0; i < 10; i++) {
            float x = 100*rand() + 2;
            int n = 20, k = 0;
            float dy = x/n;

            for (float y = 0; y < x; y += dy) { k++; }
            printf("*** n = %d, k = %d\n", n, k);

    }
}
```

Debug mode

```
========= [CASE 1] ==========
*** x = 417402.000000, dy = 20870.099609, n = 20, k = 21
*** x = 448402.000000, dy = 22420.099609, n = 20, k = 21
*** x = 2059202.000000, dy = 102960.101563, n = 20, k = 20
*** x = 31302.000000, dy = 1565.099976, n = 20, k = 21
*** x = 1808402.000000, dy = 90420.101563, n = 20, k = 20
*** x = 1989902.000000, dy = 99495.101563, n = 20, k = 20
*** x = 2211002.000000, dy = 110550.101563, n = 20, k = 20
*** x = 1406402.000000, dy = 70320.101563, n = 20, k = 20
*** x = 2694002.000000, dy = 134700.093750, n = 20, k = 21
*** x = 1945902.000000, dy = 97295.101563, n = 20, k = 20

========= [CASE 2] ==========
*** n = 20, k = 21
*** n = 20, k = 21
*** n = 20, k = 20
*** n = 20, k = 21
*** n = 20, k = 20
*** n = 20, k = 20
*** n = 20, k = 20
*** n = 20, k = 20
*** n = 20, k = 21
*** n = 20, k = 20

Press any key to continue
```

Release mode

```
======== [CASE 1] =========
*** x = 417402.000000, dy = 20870.100311, n = 20, k = 20
*** x = 448402.000000, dy = 22420.100334, n = 20, k = 20
*** x = 2059202.000000, dy = 102960.101534, n = 20, k = 20
*** x = 31302.000000, dy = 1565.100023, n = 20, k = 20
*** x = 1808402.000000, dy = 90420.101347, n = 20, k = 20
*** x = 1989902.000000, dy = 99495.101483, n = 20, k = 20
*** x = 2211002.000000, dy = 110550.101647, n = 20, k = 20
*** x = 1406402.000000, dy = 70320.101048, n = 20, k = 20
*** x = 2694002.000000, dy = 134700.102007, n = 20, k = 20
*** x = 1945902.000000, dy = 97295.101450, n = 20, k = 20

======== [CASE 2] =========
*** n = 20, k = 21
*** n = 20, k = 21
*** n = 20, k = 20
*** n = 20, k = 21
*** n = 20, k = 20
*** n = 20, k = 20
*** n = 20, k = 20
*** n = 20, k = 20
*** n = 20, k = 20
*** n = 20, k = 20

Press any key to continue
```

Debug mode

```
; 11   :              float x = 100*rand() + 2;
    call    _rand
    imul    eax, 100                    ; 00000064H
    add     eax, 2
    mov     DWORD PTR -28+[ebp], eax
    fild    DWORD PTR -28+[ebp]
    fstp    DWORD PTR _x$928[ebp]
; 12   :              int n = 20, k = 0;
    mov     DWORD PTR _n$929[ebp], 20 ; 00000014H
    mov     DWORD PTR _k$930[ebp], 0
; 13   :              float dy = x/n;
    fild    DWORD PTR _n$929[ebp]
    fdivr   DWORD PTR _x$928[ebp]
    fstp    DWORD PTR _dy$931[ebp]
; 14   :
; 15   :    for (float y = 0; y < x; y+=dy) {
    mov     DWORD PTR _y$932[ebp], 0
    jmp     SHORT $L933
$L934:
    fld     DWORD PTR _y$932[ebp]
    fadd    DWORD PTR _dy$931[ebp]
    fstp    DWORD PTR _y$932[ebp]
$L933:
    fld     DWORD PTR _y$932[ebp]
    fcomp   DWORD PTR _x$928[ebp]
    fnstsw  ax
    test    ah, 1
    je      SHORT $L935
```

CASE1

```
; 16   :                      k++;
    mov     ecx, DWORD PTR _k$930[ebp]
    add     ecx, 1
    mov     DWORD PTR _k$930[ebp], ecx
; 17   :              }
    jmp     SHORT $L934
$L935:
```

Release mode

```
; 11   :              float x = 100*rand() + 2;
    call     _rand
    lea      eax, DWORD PTR [eax+eax*4]
    lea      eax, DWORD PTR [eax+eax*4]
    lea      ecx, DWORD PTR [eax*4+2]
    mov      DWORD PTR -4+[esp+8], ecx
; 12   :              int n = 20, k = 0;
    xor      ecx, ecx
    fild     DWORD PTR -4+[esp+8]
; 13   :              float dy = x/n;
    fld      ST(0)
    fmul     DWORD PTR __real@4@3ffaccccccccccccd000
; 14   :
; 15   :     for (float y = 0; y < x; y+=dy) {
    fld      DWORD PTR __real@4@00000000000000000000
    fld      DWORD PTR __real@4@00000000000000000000
    fcomp    ST(3)
    fnstsw   ax
    test     ah, 1
    je       SHORT $L978
$L933:
    fadd     ST(0), ST(1)
; 16   :                        k++;
    inc      ecx
    fcom     ST(2)
    fnstsw   ax
    test     ah, 1
    jne      SHORT $L933
$L978:
```

CASE1

```
; 27   :              float x = 100*rand() + 2;
    call     _rand
    lea      eax, DWORD PTR [eax+eax*4]
    lea      eax, DWORD PTR [eax+eax*4]
    lea      ecx, DWORD PTR [eax*4+2]
    mov      DWORD PTR -8+[esp+12], ecx
; 28   :              int n = 20, k = 0;
    xor      ecx, ecx
    fild     DWORD PTR -8+[esp+12]
    fstp     DWORD PTR _x$944[esp+12]
; 29   :              float dy = x/n;
    fld      DWORD PTR _x$944[esp+12]
    fmul     DWORD PTR __real@4@3ffaccccccccccccd000
    fstp     DWORD PTR _dy$947[esp+12]
; 30   :
; 31   :     for (float y = 0; y < x; y+=dy) {
    fld      DWORD PTR __real@4@00000000000000000000
    fld      DWORD PTR __real@4@00000000000000000000
    fcomp    DWORD PTR _x$944[esp+12]
    fnstsw   ax
    test     ah, 1
    je       SHORT $L986
$L949:
    fadd     DWORD PTR _dy$947[esp+12]
; 32   :                        k++;
    inc      ecx
    fcom     DWORD PTR _x$944[esp+12]
    fnstsw   ax
    test     ah, 1
    jne      SHORT $L949
$L986:
```

CASE2

Numerical Computing: Example 7

$$y_n = \int_0^1 \frac{x^n}{x+5}\,dx,\ \ n = 0, 1, 2, \cdots\ (y_n > y_{n+1} > 0)$$

```c
#include <stdio.h>
#include <math.h>

main () {
    int n;
    double yn, yn_1;

    printf("\n^^^ In ascending order ^^^\n");
    yn_1 = log(1.2);
    printf(" ^^^ y(%d) = %15.9e \n", 0, yn_1);
    for (n = 1; n <= 30; n++) {
        yn = 1.0/n - 5.0*yn_1;
        printf(" ^^^ y(%d) = %15.9e \n", n, yn);
        yn_1 = yn;
    }

    printf("\n^^^ In descending order ^^^\n");
    yn = 0.0;
    printf(" ^^^ y(%d) = %15.9e \n", 20, yn);
    for (n = 20; n > 0; n--) {
        yn_1 = 1.0/(5.0*n) - yn/5.0;
        printf(" ^^^ y(%d) = %15.9e \n", n-1, yn_1);
        yn = yn_1;
    }
}
```

$$\begin{aligned} y_n &= \frac{1}{n} - 5y_{n-1} \\ y_0 &= \int_0^1 \frac{1}{x+5}\,dx = ln(x+5)|_0^1 = \log_e 1.2 \end{aligned}$$

- **[계산 I]** *numerically unstable!*

^^^ In ascending order ^^^

^^^ y(0) = 1.823215568e-001
^^^ y(1) = 8.839221603e-002
^^^ y(2) = 5.803891985e-002
^^^ y(3) = 4.313873409e-002
^^^ y(4) = 3.430632955e-002
^^^ y(5) = 2.846835223e-002
^^^ y(6) = 2.432490554e-002
^^^ y(7) = 2.123261516e-002
^^^ y(8) = 1.883692422e-002
^^^ y(9) = 1.692648999e-002
^^^ y(10) = 1.536755006e-002
^^^ y(11) = 1.407134059e-002
^^^ y(12) = 1.297663038e-002
^^^ y(13) = 1.203992502e-002
^^^ y(14) = 1.122894631e-002
^^^ y(15) = 1.052193510e-002
^^^ y(16) = 9.890324511e-003
^^^ y(17) = 9.371906857e-003
^^^ y(18) = 8.696021271e-003
^^^ y(19) = 9.151472591e-003
^^^ y(20) = 4.242637045e-003
^^^ y(21) = 2.640586239e-002
^^^ y(22) = -8.657476652e-002
^^^ y(23) = 4.763520935e-001
^^^ y(24) = -2.340093801e+000
^^^ y(25) = 1.174046900e+001
^^^ y(26) = -5.866388348e+001
^^^ y(27) = 2.933564544e+002
^^^ y(28) = -1.466746558e+003
^^^ y(29) = 7.333767272e+003
^^^ y(30) = -3.666880303e+004

- **[계산 II]** *numerically stable!*

^^^ In descending order ^^^

^^^ y(50) = 0.000000000e+000
^^^ y(19) = 1.000000000e-002
^^^ y(18) = 8.526315789e-003
^^^ y(17) = 9.405847953e-003
^^^ y(16) = 9.883536292e-003
^^^ y(15) = 1.052329274e-002
^^^ y(14) = 1.122867479e-002
^^^ y(13) = 1.203997933e-002
^^^ y(12) = 1.297661952e-002
^^^ y(11) = 1.407134276e-002
^^^ y(10) = 1.536754963e-002
^^^ y(9) = 1.692649007e-002
^^^ y(8) = 1.883692421e-002
^^^ y(7) = 2.123261516e-002
^^^ y(6) = 2.432490554e-002
^^^ y(5) = 2.846835223e-002
^^^ y(4) = 3.430632955e-002
^^^ y(3) = 4.313873409e-002
^^^ y(2) = 5.803891985e-002
^^^ y(1) = 8.839221603e-002
^^^ y(0) = 1.823215568e-001

- 이 수열 값은 직관적으로 볼 때 항상 0보다 크고 n에 대해 단조 감소를 해야 하는데, 왜 첫 번째 방법에서는 위의 수열 식을 19번 반복하였을 때 이상한 값이 나왔을까?

- 두 번째 방법에서, 대략 $y_{30} = 0$이라고 가정하고 수열 식을 반대로 계산했을 때 왜 y_0 값이 정확하게 계산이 되었을까?

- 과연 y_{16}을 어떻게 하면 쉽고 정확하게 계산할 수 있을까?

Conclusions

- 컴퓨터 상에서의 수치 계산은 우리가 머리 속에서 생각하는 수학적 계산과는 실제로 상당히 다른 결과를 초래할 수 있음.
  - 우리 머리: 연속 공간 (continuous space)에서 계산
  - 컴퓨터: 이산 공간 (discrete space)에서 계산

- 그 원인은 매우 다양하기 때문에 어느 정도 크기의 소프트웨어의 경우 종종 수치 계산 결과의 정확도에 대한 분석이 매우 어려움.

- 따라서 코드 최적화를 통한 속도 향상도 중요하지만, 부동 소수점 숫자를 통한 수치 계산을 코딩할 경우 항상 수치 계산의 정확도를 높일 수 있도록 주의를 해야 함.
  - 이를 위하여 다양한 상황에 대한 경험 및 관련 이론의 습득이 필요함.

By courtesy of D. N. Arnold
(http://www.ima.umn.edu/~arnold/disasters/patriot.html)

Still Can't Believe it? - The Patriot Missile Failure

On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, **failed to track and intercept an incoming Iraqi Scud missile**. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people. … It turns out that **the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors**. Specifically, the time in tenths of second as measured by the system's internal clock **was multiplied by 1/10** to produce the time in seconds. This calculation was performed using **a 24 bit fixed point register**. In particular, **the value 1/10, which has a non-terminating binary expansion**, was chopped at 24 bits after the radix point. **The small chopping error, when multiplied by the large number giving the time in tenths of a second, led to a significant error**. Indeed, the Patriot battery had been up around 100 hours, and an easy calculation shows that **the resulting time error due to the magnified chopping error was about 0.34 seconds**. (The number 1/10 equals $1/2^4+1/2^5+1/2^8+1/2^9+1/2^{12}+1/2^{13}+$.... In other words, the binary expansion of 1/10 is 0.0001100110011001100110011001100.... Now the 24 bit register in the Patriot stored instead 0.00011001100110011001100 introducing an error of 0.0000000000000000000000011001100... binary, or about 0.000000095 decimal. Multiplying by the number of tenths of a second in 100 hours gives 0.000000095×100×60×60×10=0.34.) A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in this time. This was far enough that the incoming Scud was outside the "range gate" that the Patriot tracked. **Ironically, the fact that the bad time calculation had been improved in some parts of the code, but not all, contributed to the problem, since it meant that the inaccuracies did not cancel.**

다음 강의 내용을 이해한 후, 각자 이 문제를 어떻게 해결할 수 있을지 생각해보자!

The following paragraph is excerpted from the GAO report.

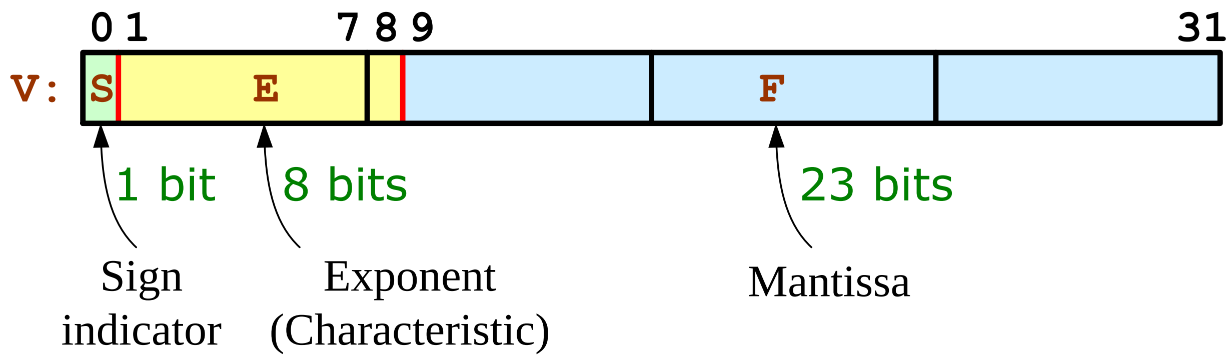
The range gate's prediction of where the Scud will next appear is a function of the Scud's known velocity and the time of the last radar detection. Velocity is a real number that can be expressed as a whole number and a decimal (e.g., 3750.2563...miles per hour). Time is kept continuously by the system's internal clock in tenths of seconds but is expressed as an integer or whole number (e.g., 32, 33, 34...). The longer the system has been running, the larger the number representing time. To predict where the Scud will next appear, both time and velocity must be expressed as real numbers. Because of the way the Patriot computer performs its calculations and the fact that its registers are only 24 bits long, the conversion of time from an integer to a real number cannot be any more precise than 24 bits. This conversion results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate's calculation is directly proportional to the target's velocity and the length of the system has been running. Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the center of the target, making it less likely that the target, in this case a Scud, will be successfully intercepted.

부동 소수점 숫자: 표현 및 연산 (Floating-Point Numbers: Representation and Operations)

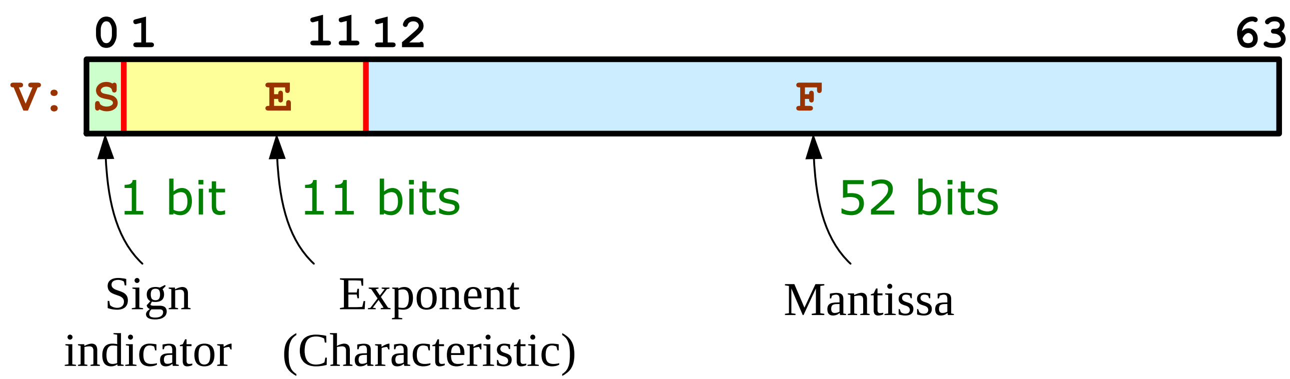
Representation of Floating-Point Numbers

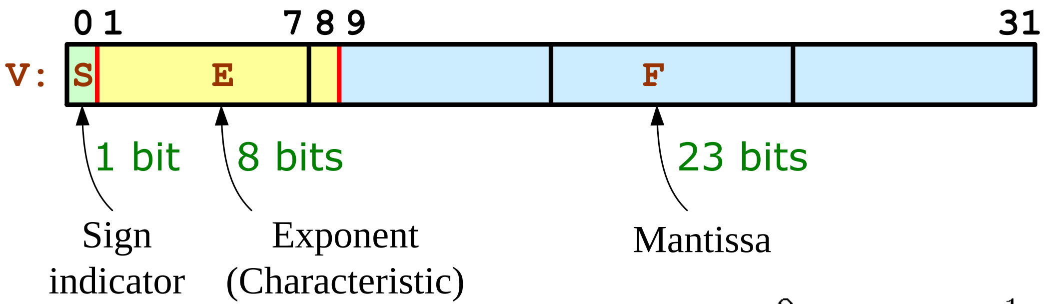
- *IEEE Standard 754 for Binary Floating-Point Arithmetic* (1985)
  - Three formats of floating-point numbers
    - Single precision: C's `float` (4 bytes)
    - Double precision: C's `double` (8 bytes)
    - Double-extended precision: C's `long double` ($\geq$ 10 bytes (?))
  - Four main goals
    - To make floating-point arithmetic as accurate as possible
    - To produce sensible outcomes in exceptional situations
    - To standardize floating-point operations across computers
    - To give the programmer control over exception handling

- Representation
  - **Single precision**의 경우 (4 bytes = 32 bits)



  - **Double precision**의 경우 (8 bytes = 64 bits)





$$F = f_1 f_2 f_3 \cdots f_{23}, \; f_i = 0 \text{ or } 1$$

$$F = 1 \cdot 2^0 + f_1 \cdot 2^{-1} + f_2 \cdot 2^{-2} + f_3 \cdot 2^{-3} + \cdots f_{23} \cdot 2^{-23}$$

Single Precision

- $E = 255$
  - $F \neq 0 \rightarrow V = \text{NaN}$
  - $F = 0$
    - $S = 1 \rightarrow V = -\text{Infinity}$
    - $S = 0 \rightarrow V = \text{Infinity}$
- $0 < E < 255 \rightarrow V = (-1)^S \cdot (1.F)_2 \cdot 2^{\underbrace{E-127}_{-126 \leq \; \leq 127}}$: Normalized Numbers
  - biased by 127
- $E = 0$
  - $F \neq 0 \rightarrow V = (-1)^S \cdot (0.F) \cdot 2^{-126}$: Unnormalized Numbers (Subnormals
  - $F = 0$
    - $S = 1 \rightarrow V = -0$
    - $S = 0 \rightarrow V = 0$

- 특징
  - Can represent only a *finite number* of floating points numbers.
    - At most 2^{32} for single precision
    - Is $(1.1)_{10}$ a machine number when this format is used?
  - Can represent only a *limited range* of floating points numbers.

$$\boxed{\mathrm{MIN}_{single} \le |V| \le \mathrm{MAX}_{single}}$$

$$\mathrm{MIN}_{single} = \begin{cases} (1.00\cdots 0)_2 \cdot 2^{-126} = 2^{-126} \approx 1.8 \cdot 10^{-38}\ (N) \\ (0.00\cdots 1)_2 \cdot 2^{-126} = 2^{-23} \cdot 2^{-126} = 2^{-149} \approx 1.4 \cdot 10^{-45}\ (SUBN) \end{cases}$$

$$\mathrm{MAX}_{single} = (1.11\cdots 1)_2 \cdot 2^{127} = \{(10.00\cdots 0)_2 - 2^{-23}\} \cdot 2^{127} = (1-2^{-24}) \cdot 2^{128} \approx 3.4 \cdot 10^{38}$$

  - The interval between machine numbers increases as the numbers grow in magnitude.

Format	Min Subnormal	Min Normal	Max Finite	Sig. Digits in Dec.
Single	1.4E-45	1.2E-38	3.4E38	6 - 9
Double	4.9E-324	2.2E-308	1.8E308	15 - 17

```
float abc;

abc = 1.1e-38;  printf("abc = %20.8e\n", abc);

abc = 1.1e-43;  printf("abc = %20.8e\n", abc);

abc = 1.1e-44;  printf("abc = %20.8e\n", abc);

abc = 1.1e-45;  printf("abc = %20.8e\n", abc);

abc = 1.1e-46;  printf("abc = %20.8e\n", abc);
```

```
abc =       1.09999996e-038
abc =       1.09301280e-043
abc =       1.12103877e-044
abc =       1.40129846e-045
abc =       0.00000000e+000
```

```
float abc, def, ghi;

scanf("%f %f", &abc, &def);
printf("abc = %20.8e\ndef = %20.8e\n", abc, def);

ghi = 1.0 - abc*def;
printf("ghi = %20.8e\n", ghi); // 4e-8

ghi = 1.0 - 1.0002*0.9998;
printf("ghi = %20.8e\n", ghi); // 4e-8
```

Intel Core i7 CPU M620

```
1.0002 0.9998
abc =       1.00020003e+000,
def =       9.99800026e-001
ghi =      -1.96032914e-008
ghi =       3.99999998e-008
Press any key to continue
```

A cheap SHARP calculator

```
0.00000004
```

- **연산의 예**

큰 수와 작은 수의 덧셈

```
                        (e = 0)
g = 1.1 =               0011 1111 1000 1100 1100 1100 1100 1101
                         3    f    8    c    c    c    c    d

                        (e = 16)
x = 123456.7890 =       0100 0111 1111 0001 0010 0000 0110 0101
                         4    7    f    1    2    0    6    5

x + g =                 111 0001 0010 0000 0110 0101
                                      1000 1100 | 1100 1100 1100 1101
                   -----------------------------------------------------
                        111 0001 0010 0000 1111 0001 | 1
```

비슷한 수끼리의 뺄셈

```
y =                     0100 0111 1111 0001 0010 0000 1111 0010
x =                     0100 0111 1111 0001 0010 0000 0110 0101
                   --------------------------------------------
y - x =                 0100 0111 1000 0000 0000 0000 1000 1101

z =                     0011 1111 1000 1101 0000 0000 0000 0000
                         3    f    8    d    0    0    0    0
```

Floating-point Numbers on Mobile Devices

From OpenGL EL Shading Language 1.0

여기서 의미하는 floating-point number는 몇 bit를 사용하는 것들일까?

Precision and Precision Qualifiers [4.5]

Any floating point, integer, or sampler declaration can have the type preceded by one of these precision qualifiers:

highp	Satisfies minimum requirements for the vertex language. Optional in the fragment language.
mediump	Satisfies minimum requirements for the fragment language. Its range and precision is between that provided by **lowp** and **highp**.
lowp	Range and precision can be less than **mediump**, but still represents all color values for any color channel.

For example:

```
lowp float color;
varying mediump vec2 Coord;
lowp ivec2 foo(lowp mat3);
highp mat4 m;
```

Ranges & precisions for precision qualifiers (FP=floating point):

	FP Range	FP Magnitude Range	FP Precision	Integer Range
highp	$(-2^{62}, 2^{62})$	$(2^{-62}, 2^{62})$	Relative 2^{-16}	$(-2^{16}, 2^{16})$
mediump	$(-2^{14}, 2^{14})$	$(2^{-14}, 2^{14})$	Relative 2^{-10}	$(-2^{10}, 2^{10})$
lowp	$(-2, 2)$	$(2^{-8}, 2)$	Absolute 2^{-8}	$(-2^{8}, 2^{8})$

A precision statement establishes a default precision qualifier for subsequent int, float, and sampler declarations, e.g.:

precision **highp** int;

예: NVIDIA GeForce 8800 GPU

From a lecture note by David Kirk and Wen-mei Hwu
(ECE 498AL, Univ. of Illinois, Urbana-Champaign)

Different Floating Point Features of Processors

	G80	SSE	IBM Altivec	Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	Round to nearest and round to zero	All 4 IEEE, round to nearest, zero, inf, -inf	Round to nearest only	Round to zero/truncate only
Denormal handling	Flush to zero	Supported, 1000's of cycles	Supported, 1000's of cycles	Flush to zero
NaN support	Yes	Yes	Yes	No
Overflow and Infinity support	Yes, only clamps to max norm	Yes	Yes	No, infinity
Flags	No	Yes	Yes	Some
Square root	Software only	Hardware	Software only	Software only
Division	Software only	Hardware	Software only	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit	12 bit
log2(x) and 2^x estimates accuracy	23 bit	No	12 bit	No

참고 : 이 테이블의 내용을 정확히 이해할 것.

Rounding Off

- Given a real number *x*, find the nearby machine number *fl*(*x*)!
  - 편의상 'normalized single' 가정하고 sign 무시

$$\boxed{x = (1.f_1 f_2 \cdots f_{23} f_{24} f_{25} \cdots)_2 \cdot 2^m \quad (f_i = 0 \text{ or } 1) \longrightarrow fl(x)}$$

  - Roundoff error 발생!

① Chopping

$$\boxed{x = (1.f_1 f_2 \cdots f_{23})_2 \cdot 2^m}$$

② Rounding up

$$\boxed{x = \{(1.f_1 f_2 \cdots f_{23})_2 + 2^{-23}\} \cdot 2^m}$$

③ Rounding (closest)

$$\boxed{x = \begin{cases} \{(1.f_1 f_2 \cdots f_{23})_2 + 2^{-23}\} \cdot 2^m & \text{if } f_{24} = 1 \\ (1.f_1 f_2 \cdots f_{23})_2 \cdot 2^m & \text{if } f_{24} = 0 \end{cases}}$$

Error

- Assume p^* is an approximation to p.
  - *Error* = $p - p^*$
  - *Absolute error* = $| p - p^* |$
  - *Relative error* = $| (p - p^*)/p |$

- Roundoff error in $fl(x)$
  ① Chopping and rounding up

$$\left|\frac{p - fl(p)}{p}\right| \leq \frac{2^{-23} \times 2^m}{(1.F)_2 \times 2^m} \leq \frac{2^{-23}}{1} = 2^{-23}$$

  ② Rounding (closest)

$$\left|\frac{p - fl(p)}{p}\right| \leq \frac{\frac{1}{2} \cdot 2^{-23} \times 2^m}{(1.F)_2 \times 2^m} \leq 2^{-24}$$

- Machine epsilon ε (unit roundoff error)

$$fl(p) = p(1+\delta),\ |\delta| \le \epsilon \text{ where } |\delta| = \left|\frac{p - fl(p)}{p}\right|$$

  - The machine epsilon ε is the largest floating-point number *x* such that *x*+1 can not be distinguished from 1 on the computer.
  - The machine epsilon ε is the smallest number that your computer recognizes as being bigger than zero.

$$\boxed{\epsilon = \max\{\, x \mid x + 1 = 1 \ \text{ in computer arithmetic. }\}}$$

  - The machine epsilon ε is the smallest positive float that can be added to one and produce a sum that is greater than one.

$$\boxed{\epsilon = \min\{\, x \mid x + 1 > 1 \ \text{ in computer arithmetic. }\}}$$

  - If *x* can be represented exactly, then the next larger float is (1+ ε)*x* and the next smaller float is (1- ε)*x*.

- Machine epsilon ε 계산

```
float (double, long double) eps;
eps = 1;
do {
    eps = eps/2;
    x = 1 + eps;
} while ( x > 1 )
eps = 2* eps;
```

Type	Bytes	Visual C++
float	4	
double	8	
long double	?	

- `DBL_EPSILON` in `float.h` of C/C++
  - `2.220446049250313 1e-16` on a Pentium 4 PC
  - 각자의 PC에서 실험한 내용과 `float.h`에 있는 machine epsilon 값과 비교해볼 것.

#include <float.h>

This file contains a set of various platform-dependent constants related to floating-point #'s. See also `<values.h>` and `<limits.h>`.

- **DBL_DIG :** Number of significant digits in a floating point number.
- **DBL_EPSILON:** The smallest x for which 1.0+x != 1.0.
- **DBL_MANT_BITS :** Number of bits used for the mantissa.
- **DBL_MANT_DIG** : Number of FLT_RADIX digits in the mantissa.
- **DBL_MAX** : The maximal floating point value (see notes about FLT_MAX_EXP).
- **DBL_MAX_10_EXP** : The maximal exponent of a floating point value expressed in base 10 (see notes about FLT_MAX_EXP).
- **DBL_MAX_2_EXP** : The maximal exponent of a floating point value expressed in base 2 (see notes about FLT_MAX_EXP).
- **DBL_MAX_EXP** : The maximal exponent of a floating point value expressed in base FLT_RADIX; greater exponents are principally possible (up to 16383), but not supported in all math functions.
- **DBL_MIN** : The minimal floating point value (see notes about FLT_MIN_EXP).
- **DBL_MIN_10_EXP** : The minimal exponent of a floating point value expressed in base 10 (see notes about FLT_MIN_EXP).
- **DBL_MIN_2_EXP** : The minimal exponent of a floating point value expressed in base 2 (see notes about FLT_MIN_EXP).
- **DBL_MIN_EXP** :The maximal exponent of a floating point value expressed in base FLT_RADIX; smaller exponents are principally possible (up to -16383), but not supported in all math functions.
- **FLT_DIG** : Number of significant digits in a floating point number.

- **FLT_EPSILON** : The smallest x for which 1.0+x != 1.0.
- **FLT_MANT_BITS** : Number of bits used for the mantissa.
- **FLT_MANT_DIG** : Number of FLT_RADIX digits in the mantissa.
- **FLT_MAX** : The maximal floating point value (see notes about FLT_MAX_EXP).
- **FLT_MAX_10_EXP** : The maximal exponent of a floating point value expressed in base 10 (see notes about FLT_MAX_EXP).
- **FLT_MAX_2_EXP** : The maximal exponent of a floating point value expressed in base 2 (see notes about FLT_MAX_EXP).
- **FLT_MAX_EXP** : The maximal exponent of a floating point value expressed in base FLT_RADIX; greater exponents are principally possible (up to 16383), but not supported in all math functions.
- **FLT_MIN** : The minimal floating point value (see notes about FLT_MIN_EXP).
- **FLT_MIN_10_EXP** : The minimal exponent of a floating point value expressed in base 10 (see notes about FLT_MIN_EXP).
- **FLT_MIN_2_EXP** : The minimal exponent of a floating point value expressed in base 2 (see notes about FLT_MIN_EXP).
- **FLT_MIN_EXP** : The minimal exponent of a floating point value expressed in base FLT_RADIX; smaller exponents are principally possible (up to -16383), but not supported in all math functions.
- **FLT_NORMALIZE** : Indicates that floating point numbers should always be normalized.

- **FLT_RADIX** : The base used for representing the exponent.
- **FLT_ROUNDS** : Option for rounding floating point numbers during the addition.
- **LDBL_DIG** : Number of significant digits in a floating point number.
- **LDBL_EPSILON** : The smallest x for which 1.0+x != 1.0.
- **LDBL_MANT_BITS** : Number of bits used for the mantissa.
- **LDBL_MANT_DIG** : Number of FLT_RADIX digits in the mantissa.
- **LDBL_MAX** : The maximal floating point value (see notes about FLT_MAX_EXP).
- **LDBL_MAX_10_EXP** : The maximal exponent of a floating point value expressed in base 10 (see notes about FLT_MAX_EXP).
- **LDBL_MAX_2_EXP** : The maximal exponent of a floating point value expressed in base 2 (see notes about FLT_MAX_EXP).
- **LDBL_MAX_EXP** : The maximal exponent of a floating point value expressed in base FLT_RADIX; greater exponents are principally possible (up to 16383), but not supported in all math functions.
- **LDBL_MIN** : The minimal floating point value (see notes about FLT_MIN_EXP).
- **LDBL_MIN_10_EXP** : The minimal exponent of a floating point value expressed in base 10 (see notes about FLT_MIN_EXP).
- **LDBL_MIN_2_EXP** : The minimal exponent of a floating point value expressed in base 2 (see notes about FLT_MIN_EXP).
- **LDBL_MIN_EXP** : The maximal exponent of a floating point value expressed in base FLT_RADIX; smaller exponents are principally possible (up to -16383), but not supported in all math functions.

Roundoff Errors in the Patriot Missile S/W

- The binary approximation to 0.1
  - `0.1 = 0.00011001100110011001100`$_2$ `= 209715/2097152`
- The roundoff error
  - `1/10 - 209715/2097152 = 1/10485760` (about 0.0001%)
- Integral values of its internal clock were converted to decimal by multiplying the binary approximation to 0.1
- After 100 hours, the error becomes
  - `(1/10 - 209715/2097152)(3600*100*10) = 5625/16384` (about 0.3433second)
- A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in this time.

By courtesy of Wolfram MathWorld
(http://mathworld.wolfram.com/RoundoffError.html)

Additional Examples of Roundoff Errors

- An egregious example of roundoff error is provided by a short-lived index devised at the Vancouver stock exchange (McCullough and Vinod 1999). At its inception in 1982, the index was given a value of 1000.000. After 22 months of recomputing the index and truncating to three decimal places at each change in market value, the index stood at 524.881, despite the fact that its "true" value should have been 1009.811.

- Other sorts of roundoff error can also occur. A notorious example is the fate of the Ariane rocket launched on June 4, 1996 (European Space Agency 1996). In the 37th second of flight, the inertial reference system attempted to convert a 64-bit floating-point number to a 16-bit number, but instead triggered an overflow error which was interpreted by the guidance system as flight data, causing the rocket to veer off course and be destroyed.

Arithmetic underflow

From Wikipedia, the free encyclopedia

Arithmetic underflow (or "floating point underflow", "floating underflow", "underflow") is a condition that can occur when the result of a floating point operation would be smaller in magnitude (closer to zero, either positive or negative) than the smallest quantity representable. Underflow is actually (negative) overflow of the exponent of the floating point quantity. For example, an eight-bit two's complement exponent can represent multipliers of 2^{-128} to 2^{127}. A result less than 2^{-128} would cause underflow.

Depending on the processor, the programming language and the run-time system, underflow may set a status bit, raise an exception, generate a hardware interrupt, or may cause some combination of these effects. Alternatively, the underflow may just be ignored and zero substituted for the unrepresentable value, although this might lead to a later division by zero error which cannot be so easily ignored.

As specified in IEEE 754 the underflow condition is only signaled if there is also a loss of accuracy. Typically this is determined as the final result being inexact. However if the user is trapping on underflow, this may happen regardless of consideration for loss of precision.

Trap (computing)

From Wikipedia, the free encyclopedia

"Kernel trap" redirects here. For the website, see KernelTrap.

In computing and operating systems, a **trap** is a type of synchronous interrupt typically caused by an exceptional condition (e.g. division by zero or invalid memory access) in a user process. A trap usually results in a switch to kernel mode, wherein the operating system performs some action before returning control to the originating process. In some usages, the term *trap* refers specifically to an interrupt intended to initate a context switch to a monitor program or debugger.[1] In SNMP, a trap is a type of PDU used to report an alert or other asynchronous event about a managed subsystem.

Arithmetic overflow

From Wikipedia, the free encyclopedia

The term **arithmetic overflow** or simply **overflow** has the following meanings.

1. In a digital computer, the condition that occurs when a calculation produces a result that is greater in magnitude than what a given register or storage location can store or represent.
2. In a digital computer, the amount by which a calculated value is greater than that which a given register or storage location can store or represent. Note that the overflow may be placed at another location.

Most computers distinguish between two kinds of overflow condition. A carry occurs when the result of an addition or subtraction, considering the operands and result as unsigned numbers, does not fit in the result. Therefore, it is useful to check the carry flag after adding or subtracting numbers that are interpreted as unsigned values. An *overflow* proper occurs when the result does not have the sign that one would predict from the signs of the operands (e.g. a negative result when adding two positive numbers). Therefore, it is useful to check the overflow flag after adding or subtracting numbers that are represented in two's complement form (i.e. they are considered signed numbers).

There are several methods of handling overflow:

1. Design: by selecting correct data types, both length and signed/unsigned.
2. Avoidance: by carefully ordering operations and checking operands in advance, it is possible to ensure that the result will never be larger than can be stored.
3. Handling: If it is anticipated that overflow may occur and when it happens detected and other processing done. Example: it is possible to add two numbers each two bytes wide using just a byte addition in steps: first add the low bytes then add the high bytes, but if it is necessary to carry out of the low bytes this is arithmetic overflow of the byte addition and it necessary to detect and increment the sum of the high bytes. CPUs generally have a way of detecting this to support addition of numbers larger than their register size, typically using a status bit.
4. Propagation: if a value is too large to be stored it can be assigned a special value indicating that overflow has occurred and then have all successive operation return this flag value. This is useful so that the problem can be checked for once at the end of a long calculation rather than after each step. This is often supported in Floating Point Hardware called FPUs.
5. Ignoring: This is the most common approach, but it gives incorrect results and can compromise a program's security.

Division by zero is *not* a form of arithmetic overflow. Mathematically, division by zero within reals is explicitly undefined; it is not that the value is too large but rather that it has *no* value.

An unhandled arithmetic overflow was the primary cause of the crash of Ariane 5 Flight 501, arguably one of the most expensive software bugs in history.

As You Now Understand the IEEE Standard 745 Format, …

[문제] 임의의 양의 부동 소수점 수자 x에 대한 정수부 floor(x) 값은 C/C++ 언어의 (int) 타입 변환 연산자를 사용하여 구할 수 있는데, 문제는 이 연산자가 많은 CPU 사이클을 소비하는 '비싼' 연산 중의 하나라는 사실이다. 컴퓨터그래픽스 연구실의 한 대학원생이 자신의 소프트웨어를 개발하면서 반복적으로 이 연산을 수행해야됨을 발견하고, 아래와 같은 문장을

```
int i;
float f;

i = (int) f;
```

아래와 같은 식으로 대치한 후,

```
int i, *tmp;
float f;

tmp = (int *) &f;
i = ((*tmp & 0x007fffff) | 0x00800000) >>
                    (150 - ((*tmp & 0x7f800000) >> 23));
```

참고 : MS Visual C++ 2005 컴파일러의 경우 "Fast floating-point model"이라는 옵션이 있는데, 실험 결과 이 옵션을 사용할 경우 상황에 따라 아래 코드보다 빠르기도 하고 느리기도 함.

3.4GHz의 Intel Pentium 4 CPU 상에서 실험해본 결과, 3배 이상의 속도 향상을 볼 수 있었다. 아래의 코드가 어떤 조건하에 어떤 방식으로 원하는 값을 올바르게 계산해주는지 생각해보자.

[문제] 일반적으로 성능이 낮은 CPU 상에서 **float** 타입의 수를 **int** 타입으로 반올림 변환해 주는 과정은 비교적 비용이 높은 연산으로 간주된다. 지금 계산 비용을 줄이기 위해 다음과 같은 방식으로 float-to-int 타입의 반올림 변환을 수행하려 한다. 참고로 이러한 방법은 Pentium II CPU 상에서 타입 변환을 하는데 드는 비용을 60 사이클에서 대략 5 사이클 정도로 줄일 수 있음. (이 문제에서는 chopping이 아니라 rounding이 사용이 된다고 가정함)

```
typedef union { int i; float f; } INTORFLOAT;
INTORFLOAT n;
INTOFRLOAT bias;


bias.i = (23 + 127) << 23; // Line (a)
n.f = 8.25f; // Line (b)


n.f += bias.f; // Line (c)
n.i -= bias.i; // Line (d)
```

참고 : 이러한 방식이 성능이 낮은 프로세서 상에서 어떤 효과가 있을까?

4b 00 00 00   41 04 00 00   4b 00 00 08   00 00 00 08

1. Line (a)가 수행된 후의 변수 bias의 내용을 16진수로 표현하라.
2. Line (b)가 수행된 후의 변수 n의 내용을 16진수로 표현하라.
3. Line (c)가 수행된 후의 변수 n의 내용을 16진수로 표현하라.
4. Line (d)가 수행된 후의 변수 n의 내용을 16진수로 표현하라. 이 값은 10진수로 나타내면 어떤 수인가?

[문제] 아래의 C 코드를 VC++의 DEBUG 모드에서 컴파일한 후 수행한 결과를 보면 심각한 문제가 있음을 알 수 있다. 과연 그 이유가 무엇인지 생각해보자 (hint: $33554432 = 2^{25}$).

```
float x; int i;

x = (float) 33554432; i = (int) x;
printf("i = %d\n", i);

x = (float) (33554432 + 1); i = (int) x;
printf("i + 1 = %d\n", i);
x = (float) (33554432 + 4); i = (int) x;
printf("i + 4 = %d\n", i);

=======================
i = 33554432
i + 1 = 33554432
i + 4 = 33554436
Press any key to continue
```

참고 : float와 double 타입의 변수는 자신이 표현할 수 있는 숫자 범위 안의 정수를 모두 표현하지 못함. 따라서 이런 타입의 변수를 정수 타입의 카운터로 사용하지 말 것. 특히 16비트 float를 사용할 경우 이러한 문제는 더 심각해짐.

Floating-Point Number를 사용하여 문제를 풀 때

- 다음과 같은 문제 등을 고려해야 함.
  - 컴퓨터가 실수를 정확하게 저장을 못해서 생기는 문제
  - 컴퓨터가 실수간의 연산을 정확하게 수행을 못해서 생기는 문제
  - 컴퓨터 자체 문제 외에 문제를 푸는 해법, 즉 알고리즘 자체가 본질적으로 unstable (ill-conditioned)해서 생기는 문제
  - 기타

- Floating-Point Arithmetic $x \bullet y \quad (x, y \in \mathrm{R})$
  - On computer,

    ① Store x and y into $fl(x)$ and $fl(y)$, respectively.

    ② Compute $fl(x) \bullet fl(y)$ as correctly as possible.

    ③ Store the result into $fl(fl(x) \bullet fl(y))$.

Normalization and roundoff error

- A bad example (Base = 10, Num. of sig. dig. = 7)

  `a = 0.1234567E0, b = 0.4711325E4, c = -b`

  Compute `a + b + c.`

<u>a + (b + c)</u>

```
ADD r1, b, c
ADD r2, a, r1
```

→ **결과: 0.1234567E0**

<u>(a + b) + c</u>

```
ADD r1, a, b
ADD r2, r1, c
```

→ **결과: 0.123000E0**

- ✓ Computer에서는 결합 법칙조차 성립 안 함!
- ✓ 부동 소수점 연산을 하면 할 수록 정확도가 점점 나빠질 확률이 높음.

- 주의할 상황
  - 비슷한 숫자끼리의 뺄셈 (Loss of significance)
    - $-b+\sqrt{b^2-4ac} \longrightarrow \frac{-4ac}{b+\sqrt{b^2-4ac}}$ when $b>0$ and $b^2 \gg |ac|$
    - $\sqrt{x+\delta}-\sqrt{x} \longrightarrow \frac{\delta}{\sqrt{x+\delta}+\sqrt{x}}$ when $x+\delta, x>0$ and $|\delta| \ll |x|$
    - $\cos(x+\delta)-\cos x \longrightarrow -2\sin\frac{\delta}{2}\sin(x+\frac{\delta}{2})$ when $|\delta| \ll |x|$
    - $x-\sin x \longrightarrow x-(x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots)$ when $|x| \approx 0$
  - 아주 큰 수와 아주 작은 수와의 덧셈/뺄셈
  - 아주 작은 수로의 나눗셈
  - 기타

Taylor Series and Taylor's Theorem

- **Taylor series of *f* at the point *c***

$$f(x) \approx \sum_{k=0}^{\infty} \frac{f^{(k)}(c)}{k!}(x-c)^k$$

- Ex.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

$$\ln(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \frac{(x-1)^5}{5} - \cdots$$

How can you evaluate the series efficiently?
→ Read about Horner's algorithm in textbook p23.

A Taylor series converges rapidly near the point of expansion and slowly (or not at all) at more remote points.

- **Taylor's theorem**

For a function $f \in C^{n+1}[a,b]$, $\quad f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(c)}{k!}(x-c)^k + E_{n+1},$ Error term

where $E_{n+1} = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-c)^{n+1}$ for some $\xi = \xi(c,x) \in (\min(c,x), \max(c,x))$.

For a function $f \in C^{n+1}[a,b]$, $\quad f(x+h) = \sum_{k=0}^{n} \frac{f^{(k)}(x)}{k!}h^k + E_{n+1},$

where $E_{n+1} = \frac{f^{(n+1)}(\xi)}{(n+1)!}h^{n+1}$ for some $\xi = \xi(x,h) \in (x, x+h)$.

- Ex.

$$\sqrt{1+h} = 1 + \frac{1}{2}h - \frac{1}{8}h^2 + \frac{1}{16}h^3\xi^{-\frac{5}{2}},\ \xi \in (1, 1+h), h > 0, \quad \sqrt{1-h} = 1 - \frac{1}{2}h - \frac{1}{8}h^2 - \frac{1}{16}h^3\xi^{-\frac{5}{2}},\ \xi \in (1+h, 1), h < 0$$

$$\sqrt{1.00001} \approx 1 + 0.5\times 10^{-5} - 0.125\times 10^{-10} = 1.00000\ 49999\ 87500$$

$$\frac{1}{16}h^3\xi^{-\frac{5}{2}} < \frac{1}{16}10^{-15} = 0.00000\ 00000\ 00000\ 0625$$

Theorem on Loss of Precision

- Theorem

  $x > y > 0:$ normalized floating-point numbers

  $2^{-p} \leq 1 - (\frac{y}{x}) \leq 2^{-q}$ for some positive integers p, q

  $\rightarrow$ at most p and at least q significant binary bits are lost in $x - y$.

- Proof of the "at least" part

  $x = r \times 2^n, \; y = s \times 2^m, \; \frac{1}{2} \leq r, s < 1$

  $x - y = (r - s2^{m-n}) \times 2^n$

  $r - s2^{m-n} = r\left(1 - \frac{s2^m}{r2^n}\right) = r\left(1 - \frac{y}{x}\right) < 2^{-q}$

  In order to normalize $x - y$, a shift of at least q bits to the left is necessary, causing at least q zeros to be supplied on the right-hand end of mantissa.

37.593621 - 37.584216

$2^{-12} \leq 1 - \frac{y}{x} = 0.0002501754 \leq 2^{-11}$

$\rightarrow$ At least 11 but not more than 12 bits are lost.

Avoiding Loss of Significance in Subtraction

- Problem: $f(x) = x - \sin x,\ x \approx 0$

$$\boxed{f(x) = x - \sin x \quad \text{versus} \quad f(x) = \frac{x^3}{3!} - \frac{x^5}{5!} + \frac{x^7}{7!} - \cdots}$$

x = **1.895494**

- Determine the range in which the series should be used when a loss of significance of at most one bit is allowed.

$$\frac{1}{2} \leq 1 - \frac{\sin x}{x} \rightarrow |x| < 1.9$$

- Determine how many terms in the series should be evaluated when an error of at most 10^{-16} is allowed.

$$E_{n+1} = \frac{f^{(n+1)}(\xi)}{(n+1)!} x^{n+1} \leq \frac{x^{n+1}}{(n+1)!} \leq \frac{1.9^{n+1}}{(n+1)!} < 10^{-16}$$

$$E_{23} \leq \frac{x^{23}}{23!} < 10^{-16} \rightarrow x \leq \sqrt[23]{\frac{23!}{10^{16}}} \approx 1.9022$$

How can you evaluate *f(x)* effectively?
→ Read textbook p66.

→ At least ten terms should be evaluated!

Test Codes

```
float f_x_sinx_naive (float x) {
  float tmp; tmp = x - (float) sin(x); return tmp;
}
double d_x_sinx_naive(double x) {
  double tmp; tmp = x - sin(x); return tmp;
}
float f_x_sinx_robust(float x) {
  float tmp, t, x2; int i;
  if (abs(x) >= 1.9) { tmp = x - (float) sin(x);  }
  else {
    x2 = x*x; t = x*x2/(float) 6.0; tmp = t;
    for (i = 2; i <= 10; i++) { t *= -x2/(float) ((2*i)*(2*i+1)); tmp += t;
  }
  return tmp;
}
double d_x_sinx_robust(double x) {
  double tmp, t, x2; int i;
  if (abs(x) >= 1.9) { tmp = x - sin(x); }
  else {
   x2 = x*x; t = x*x2/6.0; tmp = t;
    for (i = 2; i <= 10; i++) { t *= -x2/((2*i)*(2*i+1)); tmp += t; }
  }
  return tmp;
}
```

Test Results

$x^* = 1.9$가 옳은 선택인가?

x	S_naive	D_naive	S_robust	D_robust
* 3.0000000e+000	2.858880e+000	2.858880e+000	2.858880e+000	2.858880e+000
* 2.9600000e+000	2.779404e+000	2.779404e+000	2.779404e+000	2.779404e+000
* 2.9200001e+000	2.700217e+000	2.700216e+000	2.700217e+000	2.700216e+000
* 2.8800001e+000	2.621381e+000	2.621381e+000	2.621381e+000	2.621381e+000
* 2.8399999e+000	2.542958e+000	2.542959e+000	2.542958e+000	2.542959e+000
* 2.8000000e+000	2.465012e+000	2.465012e+000	2.465012e+000	2.465012e+000
* 2.7600000e+000	2.387601e+000	2.387601e+000	2.387601e+000	2.387601e+000
* 2.7200000e+000	2.310786e+000	2.310786e+000	2.310786e+000	2.310786e+000
* 2.6800001e+000	2.234626e+000	2.234625e+000	2.234626e+000	2.234625e+000
* 2.6400001e+000	2.159178e+000	2.159177e+000	2.159178e+000	2.159177e+000
* 2.5999999e+000	2.084498e+000	2.084499e+000	2.084498e+000	2.084499e+000
* 2.5599999e+000	2.010644e+000	2.010645e+000	2.010644e+000	2.010645e+000
* 2.5200000e+000	1.937669e+000	1.937669e+000	1.937669e+000	1.937669e+000
* 2.4800000e+000	1.865626e+000	1.865626e+000	1.865626e+000	1.865626e+000
* 2.4400001e+000	1.794565e+000	1.794565e+000	1.794565e+000	1.794565e+000
* 2.4000001e+000	1.724537e+000	1.724537e+000	1.724537e+000	1.724537e+000
* 2.3599999e+000	1.655589e+000	1.655589e+000	1.655589e+000	1.655589e+000
* 2.3199999e+000	1.587768e+000	1.587769e+000	1.587768e+000	1.587769e+000
* 2.2800000e+000	1.521119e+000	1.521119e+000	1.521119e+000	1.521119e+000
* 2.2400000e+000	1.455684e+000	1.455684e+000	1.455684e+000	1.455684e+000
* 2.2000000e+000	1.391504e+000	1.391504e+000	1.391504e+000	1.391504e+000
* 2.1600001e+000	1.328617e+000	1.328617e+000	1.328617e+000	1.328617e+000
* 2.1199999e+000	1.267059e+000	1.267060e+000	1.267059e+000	1.267060e+000
* 2.0799999e+000	1.206867e+000	1.206867e+000	1.206867e+000	1.206867e+000
* 2.0400000e+000	1.148071e+000	1.148071e+000	1.148071e+000	1.148071e+000
* 2.0000000e+000	1.090703e+000	1.090703e+000	1.090703e+000	1.090703e+000
* 1.9600000e+000	1.034789e+000	1.034788e+000	1.034789e+000	1.034788e+000
* 1.9200000e+000	9.803545e-001	9.803545e-001	9.803545e-001	9.803545e-001
* 1.8800000e+000	9.274238e-001	9.274238e-001	9.274238e-001	9.274238e-001
* 1.8400000e+000	8.760170e-001	8.760170e-001	8.760170e-001	8.760170e-001
* 1.8000000e+000	8.261523e-001	8.261524e-001	8.261523e-001	8.261524e-001
* 1.7600000e+000	7.778457e-001	7.778457e-001	7.778457e-001	7.778457e-001
* 1.7200000e+000	7.311103e-001	7.311102e-001	7.311103e-001	7.311102e-001
* 1.6799999e+000	6.859567e-001	6.859568e-001	6.859567e-001	6.859568e-001
* 1.6400000e+000	6.423936e-001	6.423936e-001	6.423936e-001	6.423936e-001
* 1.6000000e+000	6.004264e-001	6.004264e-001	6.004264e-001	6.004264e-001
* 1.5599999e+000	5.600582e-001	5.600583e-001	5.600582e-001	5.600583e-001
* 1.5200000e+000	5.212898e-001	5.212899e-001	5.212898e-001	5.212899e-001
* 1.4800000e+000	4.841192e-001	4.841192e-001	4.841192e-001	4.841192e-001
* 1.4400001e+000	4.485417e-001	4.485417e-001	4.485416e-001	4.485417e-001
* 1.4000000e+000	4.145502e-001	4.145503e-001	4.145502e-001	4.145503e-001
* 1.3600000e+000	3.821354e-001	3.821354e-001	3.821354e-001	3.821354e-001
* 1.3200001e+000	3.512849e-001	3.512849e-001	3.512850e-001	3.512849e-001
* 1.2800000e+000	3.219841e-001	3.219841e-001	3.219841e-001	3.219841e-001
* 1.2400000e+000	2.942160e-001	2.942160e-001	2.942160e-001	2.942160e-001
* 1.2000000e+000	2.679610e-001	2.679609e-001	2.679609e-001	2.679609e-001
* 1.1600000e+000	2.431968e-001	2.431969e-001	2.431969e-001	2.431969e-001
* 1.1200000e+000	2.198995e-001	2.198996e-001	2.198996e-001	2.198996e-001
* 1.0800000e+000	1.980422e-001	1.980422e-001	1.980422e-001	1.980422e-001
* 1.0400000e+000	1.775957e-001	1.775958e-001	1.775958e-001	1.775958e-001
* 1.0000000e+000	1.585290e-001	1.585290e-001	1.585290e-001	1.585290e-001

x	RE(S_naive)	RE(D_naive)	RE(S_robust)
* 5.0000000e-001	5.30e-007	5.06e-016	1.33e-008
* 2.5000000e-001	1.98e-006	2.84e-015	9.05e-009
* 1.2500000e-001	1.04e-005	9.00e-015	1.94e-008
* 6.2500000e-002	4.27e-005	5.01e-014	3.01e-008
* 3.1250000e-002	1.71e-004	3.07e-013	4.06e-008
* 1.5625000e-002	5.00e-004	1.99e-013	1.20e-008
* 7.8125000e-003	1.96e-003	5.16e-012	4.17e-008
* 3.9062500e-003	7.81e-003	3.19e-012	1.19e-008
* 1.9531250e-003	3.13e-002	1.17e-011	4.17e-008
* 9.7656250e-004	1.25e-001	4.66e-011	1.19e-008
* 4.8828125e-004	5.00e-001	1.86e-010	4.17e-008
* 2.4414063e-004	1.00e+000	7.45e-010	3.28e-008
* 1.2207031e-004	1.00e+000	1.42e-008	3.05e-008
* 6.1035156e-005	1.00e+000	5.94e-008	3.00e-008
* 3.0517578e-005	1.00e+000	2.38e-007	2.98e-008
* 1.5258789e-005	1.00e+000	9.54e-007	2.98e-008
* 7.6293945e-006	1.00e+000	3.81e-006	2.98e-008
* 3.8146973e-006	1.00e+000	1.53e-005	2.98e-008
* 1.9073486e-006	1.00e+000	6.10e-005	2.98e-008
* 9.5367432e-007	1.00e+000	2.44e-004	2.98e-008
* 4.7683716e-007	1.00e+000	9.77e-004	2.98e-008
* 2.3841858e-007	1.00e+000	3.91e-003	2.98e-008

* 1.1920929e-007	1.00e+000	1.56e-002	2.98e-008
* 5.9604645e-008	1.00e+000	6.25e-002	2.98e-008
* 2.9802322e-008	1.00e+000	2.50e-001	2.98e-008
* 1.4901161e-008	1.00e+000	1.00e+000	2.98e-008
* 7.4505806e-009	1.00e+000	1.00e+000	2.98e-008
* 3.7252903e-009	1.00e+000	1.00e+000	2.98e-008
* 1.8626451e-009	1.00e+000	1.00e+000	2.98e-008
* 9.3132257e-010	1.00e+000	1.00e+000	2.98e-008
* 4.6566129e-010	1.00e+000	1.00e+000	2.98e-008
* 2.3283064e-010	1.00e+000	1.00e+000	2.98e-008
* 1.1641532e-010	1.00e+000	1.00e+000	2.98e-008
* 5.8207661e-011	1.00e+000	1.00e+000	2.98e-008
* 2.9103830e-011	1.00e+000	1.00e+000	2.98e-008
* 1.4551915e-011	1.00e+000	1.00e+000	2.98e-008
* 7.2759576e-012	1.00e+000	1.00e+000	2.98e-008
* 3.6379788e-012	1.00e+000	1.00e+000	2.98e-008
* 1.8189894e-012	1.00e+000	1.00e+000	2.98e-008
* 9.0949470e-013	1.00e+000	1.00e+000	2.98e-008
* 4.5474735e-013	1.00e+000	1.00e+000	2.98e-008
* 2.2737368e-013	1.00e+000	1.00e+000	2.38e-007
* 1.1368684e-013	1.00e+000	1.00e+000	1.91e-006
* 5.6843419e-014	1.00e+000	1.00e+000	1.53e-005
* 2.8421709e-014	1.00e+000	1.00e+000	1.22e-004
* 1.4210855e-014	1.00e+000	1.00e+000	9.77e-004
* 7.1054274e-015	1.00e+000	1.00e+000	7.81e-003
* 3.5527137e-015	1.00e+000	1.00e+000	6.25e-002
* 1.7763568e-015	1.00e+000	1.00e+000	5.00e-001
* 8.8817842e-016	1.00e+000	1.00e+000	1.00e+000

x	S_naive	D_naive	S_robust	D_robust
* 5.0000000e-001	2.057445e-002	2.057446e-002	2.057446e-002	2.057446e-002
* 2.5000000e-001	2.596036e-003	2.596041e-003	2.596041e-003	2.596041e-003
* 1.2500000e-001	3.252700e-004	3.252666e-004	3.252666e-004	3.252666e-004
* 6.2500000e-002	4.068390e-005	4.068216e-005	4.068216e-005	4.068216e-005
* 3.1250000e-002	5.086884e-006	5.086015e-006	5.086015e-006	5.086015e-006
* 1.5625000e-002	6.360933e-007	6.357751e-007	6.357751e-007	6.357751e-007
* 7.8125000e-003	7.962808e-008	7.947262e-008	7.947262e-008	7.947262e-008
* 3.9062500e-003	1.001172e-008	9.934100e-009	9.934100e-009	9.934100e-009
* 1.9531250e-003	1.280569e-009	1.241763e-009	1.241763e-009	1.241763e-009
* 9.7656250e-004	1.746230e-010	1.552204e-010	1.552204e-010	1.552204e-010
* 4.8828125e-004	2.910383e-011	1.940255e-011	1.940255e-011	1.940255e-011
* 2.4414063e-004	0.000000e+000	2.425319e-012	2.425319e-012	2.425319e-012
* 1.2207031e-004	0.000000e+000	3.031649e-013	3.031649e-013	3.031649e-013
* 6.1035156e-005	0.000000e+000	3.789561e-014	3.789561e-014	3.789561e-014
* 3.0517578e-005	0.000000e+000	4.736950e-015	4.736952e-015	4.736952e-015
* 1.5258789e-005	0.000000e+000	5.921184e-016	5.921190e-016	5.921189e-016
* 7.6293945e-006	0.000000e+000	7.401459e-017	7.401487e-017	7.401487e-017
* 3.8146973e-006	0.000000e+000	9.251717e-018	9.251859e-018	9.251859e-018
* 1.9073486e-006	0.000000e+000	1.156412e-018	1.156482e-018	1.156482e-018
* 9.5367432e-007	0.000000e+000	1.445250e-019	1.445603e-019	1.445603e-019
* 4.7683716e-007	0.000000e+000	1.805239e-020	1.807004e-020	1.807004e-020
* 2.3841858e-007	0.000000e+000	2.249931e-021	2.258755e-021	2.258755e-021
* 1.1920929e-007	0.000000e+000	2.779327e-022	2.823443e-022	2.823443e-022
* 5.9604645e-008	0.000000e+000	3.308722e-023	3.529304e-023	3.529304e-023
* 2.9802322e-008	0.000000e+000	3.308722e-024	4.411630e-024	4.411630e-024
* 1.4901161e-008	0.000000e+000	0.000000e+000	5.514538e-025	5.514537e-025
* 7.4505806e-009	0.000000e+000	0.000000e+000	6.893172e-026	6.893172e-026
* 3.7252903e-009	0.000000e+000	0.000000e+000	8.616465e-027	8.616465e-027
* 1.8626451e-009	0.000000e+000	0.000000e+000	1.077058e-027	1.077058e-027
* 9.3132257e-010	0.000000e+000	0.000000e+000	1.346323e-028	1.346323e-028
* 4.6566129e-010	0.000000e+000	0.000000e+000	1.682903e-029	1.682903e-029
* 2.3283064e-010	0.000000e+000	0.000000e+000	2.103629e-030	2.103629e-030
* 1.1641532e-010	0.000000e+000	0.000000e+000	2.629536e-031	2.629536e-031
* 5.8207661e-011	0.000000e+000	0.000000e+000	3.286921e-032	3.286920e-032
* 2.9103830e-011	0.000000e+000	0.000000e+000	4.108651e-033	4.108651e-033
* 1.4551915e-011	0.000000e+000	0.000000e+000	5.135813e-034	5.135813e-034
* 7.2759576e-012	0.000000e+000	0.000000e+000	6.419767e-035	6.419766e-035
* 3.6379788e-012	0.000000e+000	0.000000e+000	8.024708e-036	8.024708e-036
* 1.8189894e-012	0.000000e+000	0.000000e+000	1.003089e-036	1.003089e-036
* 9.0949470e-013	0.000000e+000	0.000000e+000	1.253861e-037	1.253861e-037
* 4.5474735e-013	0.000000e+000	0.000000e+000	1.567326e-038	1.567326e-038
* 2.2737368e-013	0.000000e+000	0.000000e+000	1.959157e-039	1.959157e-039
* 1.1368684e-013	0.000000e+000	0.000000e+000	2.448951e-040	2.448947e-040
* 5.6843419e-014	0.000000e+000	0.000000e+000	3.061136e-041	3.061183e-041
* 2.8421709e-014	0.000000e+000	0.000000e+000	3.826946e-042	3.826479e-042
* 1.4210855e-014	0.000000e+000	0.000000e+000	4.778428e-043	4.783099e-043
* 7.1054274e-015	0.000000e+000	0.000000e+000	6.025583e-044	5.978873e-044
* 3.5527137e-015	0.000000e+000	0.000000e+000	7.006492e-045	7.473592e-045
* 1.7763568e-015	0.000000e+000	0.000000e+000	1.401298e-045	9.341990e-046
* 8.8817842e-016	0.000000e+000	0.000000e+000	0.000000e+000	1.167749e-046

Which One Would be Better Numerically?

- Variance computation

$$\boxed{s_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$s_n^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)$$

- Modulus of a complex number

$$|x| = \sqrt{x_r^2 + x_i^2}$$

$$\boxed{|x| = a\sqrt{1 + \left(\frac{b}{a}\right)^2}} \quad (a = \max(|x_r|,\ |x_i|),\ \ b = \min(|x_r|,\ |x_i|))$$

미분 값의 근사

- 다음과 같은 공식을 사용하여 미분 값을 수치적으로 구하려 할 때 적절한 h 값의 크기는?

$$\boxed{f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}}$$

  - ✓ h가 작아질 수록 이론적인 오차 감소 → truncation error 감소.
  - ✓ h가 작아질 수록 비슷한 숫자끼리의 뺄셈과 아주 작은 수로의 나눗셈이 발생함으로써 오차 증가 → loss of significance 증가.

- 적절한 h값의 유도

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} - \frac{h^2 f^{(3)}(\xi)}{3!}$$

Let ϵ_+ and ϵ_- be the errors introduced when $f(x+h)$ and $f(x-h)$ are computed on a computer, respectively.

$$f'_{comp} = \frac{f(x+h)+\epsilon_+ - f(x-h) - \epsilon_-}{2h} = \frac{f(x+h)-f(x-h)}{2h} + \frac{\epsilon_+ - \epsilon_-}{2h}$$

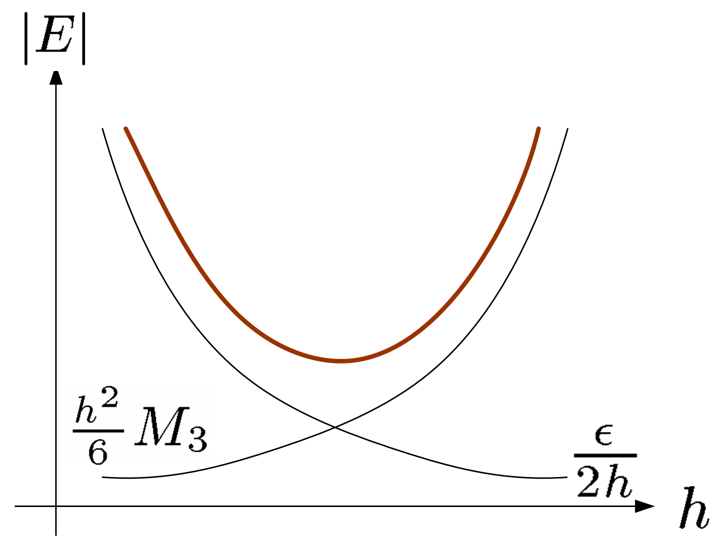
$$f'(x) = f'_{comp} - \frac{\epsilon_+ - \epsilon_-}{2h} - \frac{h^2 f^{(3)}(\xi)}{6} = f'_{comp} - \left(\frac{\epsilon_+ - \epsilon_-}{2h} + \frac{h^2 f^{(3)}(\xi)}{6}\right)$$

$$\begin{aligned} |E| &= \left|\frac{\epsilon_+ - \epsilon_-}{2h} + \frac{h^2 f^{(3)}(\xi)}{6}\right| \leq \left|\frac{\epsilon_+ - \epsilon_-}{2h}\right| + \left|\frac{h^2 f^{(3)}(\xi)}{6}\right| \\ &\leq \frac{\epsilon}{2h} + \frac{h^2}{6}M_3 \quad (|\epsilon_+ - \epsilon_-| \leq \epsilon,\ |f^{(3)}(\xi)| \leq M_3 \text{ for } \xi \in (x-h, x+h)) \end{aligned}$$

roundoff error

truncation error

$$\begin{aligned} g(h) &\equiv \frac{\epsilon}{2h} + \frac{h^2}{6}M_3 \\ g'(h) &= -\frac{\epsilon}{2}\frac{1}{h^2} + \frac{h}{3}M_3 \\ g''(h) &= \epsilon\frac{1}{h^3} + \frac{M_3}{3} > 0 \quad \text{for } h > 0 \end{aligned}$$



So, we compute h^* such that $g'(h^*) = 0$.

$$g'(h^*) = -\frac{\epsilon}{2}\frac{1}{h^{*2}} + \frac{h^*}{3}M_3 = 0 \longrightarrow h^* = (\frac{3\epsilon}{2M_3})^{\frac{1}{3}}$$

If $|\epsilon_+|, |\epsilon_-| \approx 10^{-8}$ and $f(x) = e^x$ at $x = 0$, $\epsilon \approx 2 \cdot 10^{-8}$ and $M_3 \approx 1$.

Hence, $h^* \approx (\frac{3 \cdot 2 \cdot 10^{-8}}{2 \cdot 1})^{\frac{1}{3}} = 10^{-3} \cdot 30^{\frac{1}{3}} \approx 0.003$. $\square$

안정적인 계산
(Stable Computation)

Stable, Unstable, and Conditionally Stable

- 문제
  - 주어진 문제에 대한 input data에 약간의 오차가 개입되었을 때, 이러한 오차가 이 문제를 해결해주는 어떤 알고리즘이 산출하는 output data에 어떤 영향을 미칠 것인가?

  - Stable:
  - Unstable:
  - Conditionally stable:

INPUT → **Algorithm Procedure Solution Program** → **OUTPUT**

Numerical Process

- Condition Number
  - 정의:

$$\boxed{(cond\ f)(x) \equiv \frac{x \cdot f'(x)}{f(x)}}$$

algorithm

$$x \xrightarrow{f} y$$

y = f(x)

- 의미: How sensitive may the solution *f* of a problem be to small relative changes in the input data?

$$
\begin{aligned}
f(x+\Delta x) &= y + \Delta y \\
\Delta y &= f(x+\Delta x) - f(x) = f'(\xi)\cdot \Delta x \ (x \le \xi \le x+\Delta x) \\
&\approx f'(x)\cdot \Delta x \\
\rightarrow \frac{\Delta y}{y} &\approx \frac{f'(x)\cdot \Delta x}{f(x)} = \frac{x\cdot f'(x)}{f(x)} \cdot \frac{\Delta x}{x}
\end{aligned}
$$

relative error in output $\rightarrow$ $\boxed{\frac{\Delta y}{y} \approx (cond\ f)(x) \cdot \frac{\Delta x}{x}}$ $\leftarrow$ relative error in input

✓ If $|(cond\ f)(x)| >> 1$, *f* is called *unstable* or *ill-conditioned*.

✓ If $|(cond\ f)(x)| << 1$, *f* is called *stable* or *well-conditioned*.

Example 1: 연립 방정식의 풀이

- 이원 일차 연립 방정식

$$\begin{aligned} x + \alpha \cdot y &= 1 \\ \alpha \cdot x + y &= 0 \\ (\alpha &\neq 1) \end{aligned} \longrightarrow x = \frac{1}{1-\alpha^2}, y = \frac{-\alpha}{1-\alpha^2}, x + y = \frac{1}{1+\alpha}$$

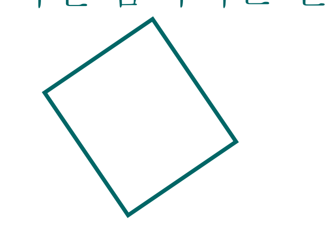
- $x = f_1(\alpha)$

$$\boxed{|(cond\ f_1)(\alpha)| = \frac{2\alpha^2}{|1-\alpha^2|}}$$

$$\alpha \longrightarrow \boxed{f_1} \longrightarrow x$$

직관적 의미

어떤 움직이는 물체가 땅에 닿았는가?



$$\begin{cases} \alpha \approx 1: & f_1 \text{ is ill-conditioned.} \\ \alpha^2 \gg 1: & |(cond\ f_1)(\alpha)| \to 2 \\ \alpha \approx 0: & |(cond\ f_1)(\alpha)| \to 0 \end{cases}$$

✓ y와 $x+y$를 풀어주는 “algorithm”들에 대해서도 분석할 것!

Example 2: 수열의 계산

- 단조 감소 수열

$$y_n = \int_0^1 \frac{x^n}{x+5}\,dx,\ n = 0, 1, 2, \cdots\ (y_n > y_{n+1} > 0)$$

$$\begin{aligned} y_n &= \int_1^0 \frac{x^n}{x+5}\,dx = \int_1^0 \frac{x^n + 5x^{n-1} - 5x^{n-1}}{x+5}\,dx \\ &= \int_1^0 \left(\frac{x^{n-1}(x+5)}{x+5} - \frac{5x^{n-1}}{x+5}\right) dx = \int_1^0 x^{n-1}\,dx - 5\int_1^0 \frac{x^{n-1}}{x+5}\,dx \\ &= \frac{1}{n} - 5y_{n-1} \end{aligned}$$

$$\boxed{\begin{aligned} y_n &= \frac{1}{n} - 5y_{n-1} \\ y_0 &= \int_0^1 \frac{1}{x+5}\,dx = ln(x+5)|_0^1 = \log_e 1.2 \end{aligned}}$$

```
printf("\n^^^ In ascending order ^^^\n");
yn_1 = log(1.2);
printf(" ^^^ y(%d) = %15.9e \n", 0, yn_1);
for (n = 1; n <= 30; n++) {
   yn = 1.0/n - 5.0*yn_1;
   printf(" ^^^ y(%d) = %15.9e \n", n, yn);
   yn_1 = yn;
}

printf("\n^^^ In descending order ^^^\n");
yn = 0;
printf(" ^^^ y(%d) = %15.9e \n", 20, yn);
for (n = 20; n > 0; n--) {
   yn_1 = 1.0/(5.0*n) - yn/5.0;
   printf(" ^^^ y(%d) = %15.9e \n", n-1, yn_1);
   yn = yn_1;
}
```

- **[계산 I]** *numerically unstable!*

^^^ In ascending order ^^^

^^^ y(0) = 1.823215568e-001
^^^ y(1) = 8.839221603e-002
^^^ y(2) = 5.803891985e-002
^^^ y(3) = 4.313873409e-002
^^^ y(4) = 3.430632955e-002
^^^ y(5) = 2.846835223e-002
^^^ y(6) = 2.432490554e-002
^^^ y(7) = 2.123261516e-002
^^^ y(8) = 1.883692422e-002
^^^ y(9) = 1.692648999e-002
^^^ y(10) = 1.536755006e-002
^^^ y(11) = 1.407134059e-002
^^^ y(12) = 1.297663038e-002
^^^ y(13) = 1.203992502e-002
^^^ y(14) = 1.122894631e-002
^^^ y(15) = 1.052193510e-002
^^^ y(16) = 9.890324511e-003
^^^ y(17) = 9.371906857e-003
^^^ y(18) = 8.696021271e-003
^^^ y(19) = 9.151472591e-003
^^^ y(20) = 4.242637045e-003
^^^ y(21) = 2.640586239e-002
^^^ y(22) = -8.657476652e-002
^^^ y(23) = 4.763520935e-001
^^^ y(24) = -2.340093801e+000
^^^ y(25) = 1.174046900e+001
^^^ y(26) = -5.866388348e+001
^^^ y(27) = 2.933564544e+002
^^^ y(28) = -1.466746558e+003
^^^ y(29) = 7.333767272e+003
^^^ y(30) = -3.666880303e+004

- **[계산 II]** *numerically stable!*

^^^ In descending order ^^^

^^^ y(50) = 0.000000000e+000
^^^ y(19) = 1.000000000e-002
^^^ y(18) = 8.526315789e-003
^^^ y(17) = 9.405847953e-003
^^^ y(16) = 9.883536292e-003
^^^ y(15) = 1.052329274e-002
^^^ y(14) = 1.122867479e-002
^^^ y(13) = 1.203997933e-002
^^^ y(12) = 1.297661952e-002
^^^ y(11) = 1.407134276e-002
^^^ y(10) = 1.536754963e-002
^^^ y(9) = 1.692649007e-002
^^^ y(8) = 1.883692421e-002
^^^ y(7) = 2.123261516e-002
^^^ y(6) = 2.432490554e-002
^^^ y(5) = 2.846835223e-002
^^^ y(4) = 3.430632955e-002
^^^ y(3) = 4.313873409e-002
^^^ y(2) = 5.803891985e-002
^^^ y(1) = 8.839221603e-002
^^^ y(0) = 1.823215568e-001

- Condition number를 통한 분석: **[계산 I]**

$$\boxed{y_n = f_1(y_0)}$$

$$y_0 \longrightarrow \boxed{f_1} \longrightarrow y_n \quad \text{for some } n > 0$$

$$\begin{aligned} y_1 &= 1 - 5y_0 = 1 + (-5)^1 y_0 \\ y_2 &= \frac{1}{2} - 5(1 - 5y_0) = \frac{1}{2} - 5 + (-5)^2 y_0 \\ &\vdots \\ y_n &= c_{n-1} + (-5)^n y_0 \equiv f_1(y_0) \end{aligned}$$

$$c_1^* = |(cond\ f_1)(y_0)| = |\frac{y_0(-5)^n}{y_n}| > |\frac{y_n(-5)^n}{y_n}| = 5^n$$

$$\boxed{\longrightarrow c_1^* > 5^n}$$

- Condition number를 통한 분석: **[계산 II]**

$$y_m \longrightarrow \boxed{f_2} \longrightarrow y_n$$

for some m > n

$$\boxed{y_n = f_2(y_m)}$$

$$\vdots$$

$$y_n = d_n + (-\frac{1}{5})^{m-n} y_m \equiv f_2(y_m)$$

$$c_2^* = |(cond\ f_2)(y_m)| = |\frac{y_m(-\frac{1}{5})^{m-n}}{y_n}| < |\frac{y_n(-\frac{1}{5})^{m-n}}{y_n}| = (-\frac{1}{5})^{m-n}$$

$$\boxed{\longrightarrow c_2^* < (-\frac{1}{5})^{m-n}}$$

- [방법 II]를 사용하여 y_n을 구하려 하는데 상대 오차가 ε보다 작게 하려면 얼마나 큰 m에서 시작을 해야 하는가?

Start with $y_m^* = 0$.

$$|\frac{y_n^* - y_n}{y_n}| \leq (\frac{1}{5})^{m-n} |\frac{y_m^* - y_m}{y_m}| = (\frac{1}{5})^{m-n} \leq \epsilon$$

$$(m-n) \log_e \frac{1}{5} \leq \log_e \epsilon$$

$$\longrightarrow m \geq n - \frac{\log_e \epsilon}{\log_e 5}$$

For example, when $\epsilon = 10^{-15}$, $m \geq n - \frac{\log_e 10^{-15}}{\log_e 5} \approx n + 21.46$

$$\longrightarrow m \geq n + 22. \quad \square$$

$$|y_n^* - y_n| = |\frac{1}{5(n+1)} - \frac{1}{5}y_{n+1}^* - \frac{1}{5(n+1)} + \frac{1}{5}y_{n+1}| = | - \frac{1}{5}(y_{n+1}^* - y_{n+1})|$$

$$\longrightarrow |\frac{y_n^* - y_n}{y_n}| = \frac{1}{5}|\frac{y_{n+1}^* - y_{n+1}}{y_n}| \leq \frac{1}{5}|\frac{y_{n+1}^* - y_{n+1}}{y_{n+1}}|$$

$$\boxed{|\frac{y_n^* - y_n}{y_n}| \leq (\frac{1}{5})^{m-n}|\frac{y_m^* - y_m}{y_m}| \text{ for } n < m.}$$