Finite precision arithmetic in programming : Tips and pitfalls

Goal of this lecture

- See some examples of what can go wrong with floating point arithmetic.
- Learn how to avoid some common mistakes.

Why do we worry about all of this?

① Consequences of round-off error

- $\underbrace{0.1+0.1+0.1 \ldots 0.1}_{10 \text{ times}} \neq 1$ why not?
- Never compare real numbers with equality
- Choosing tolerances for stopping criteria

② Catastrophic cancellation

$$\frac{(1+x)-1}{x} \neq 1$$

not always. - why not?

- Quadratic formula

③ Floating point arithmetic is not associative.

why not?

$$0.2+(0.3+0.1) \neq (0.2+0.3)+0.1$$

① Consequences of Round-off error.

Recall: Round-off error is the error that is made when we represent real numbers using finite precision arithmetic.

Example: $x = 0.1$ cannot be represented exactly on the computer.

$$x = 0.1 = 1.10011001100110011\overline{0011} \times 2^{-4}$$

repeats

How do we see the effects of this?

Python

```
In [2]:
x = 0.1
print("x = {:30.20f}".format(x))
```

```
x =           0.10000000000000000555
```

Round-off error $\approx 10^{-17}$

"Accumulation of round-off error"

```
In [4]:
print("x+x         = {:30.20f}".format(x+x))
print("x+x+x       = {:30.20f}".format(x+x+x))
print("x+x+x+x     = {:30.20f}".format(x+x+x+x))
print("x+x+x+x+x = {:30.20f}".format(x+x+x+x+x))
```

```
x+x         =          0.20000000000000001110
x+x+x       =          0.30000000000000004441
x+x+x+x     =          0.40000000000000002220
x+x+x+x+x   =          0.50000000000000000000
```

round-off error.

(x = .5 can be represented exactly)

Python

```
In [8]:
def roundoff():
    x = 0;
    for k in range(10):
        x += 0.1
    if x == 1:
        print("x == 1")
    else:
        print("x != 1;   x = {:30.20f}".format(x))

roundoff()
```

```
x != 1;   x =          0.99999999999999988898
```

$\neq 1$

Take home message: Avoid equality comparisons between real numbers

Example: Algorithm approximate the square root $\sqrt{a}$.

In [9]:
```
%matplotlib notebook
%pylab
```

```
Using matplotlib backend: nbAgg
Populating the interactive namespace from numpy and matplotlib
```

In [23]:
```
def squareroot(a):
    x = 1
    tol = 1e-8
    for k in range(20):
        xp = (x + a/x)/2
        if (abs(xp-x) < tol):
            x = xp
            return x
        x = xp
    return x
a = 5
x = squareroot(a)
print("Computed : {:.20f}".format(x))
print("True     : {:.20f}".format(sqrt(a)))
```

use a tolerance

```
Computed : 2.23606797749978980505
True     : 2.23606797749978980505
```

Use tolerances rather than exact equality.

Example

Use Taylor Series to approximate $\cos(x)$:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots = \sum_{n=0}^{\infty} (-1)^k \frac{x^{2n}}{(2n)!}$$

```python
def taylor(x):
    T = 0
    for k in range(15):
        T +=  (-1)**(k)*x**(2*k)/math.factorial(2*k)
        err = abs(cos(x)-T)
        print("{:5d} {:30.20f} {:12.4e}".format(k,T,err))
        if cos(x) == T:
            printf('{:d} terms are needed'.format(k))
            return T
    print("No equality!")
    return T

taylor(1.1)
```

```
    0        1.00000000000000000000   5.4640e-01
    1        0.39499999999999990674   5.8596e-02
    2        0.45600416666666659937   2.4080e-03
    3        0.45354366527777772999   5.2456e-05
    4        0.45359682968278763893   7.0826e-07
    5        0.45359611491689805218   6.5087e-09
    6        0.45359612146891870044   4.3341e-11
    7        0.45359612142535854495   2.1877e-13
    8        0.45359612142557814707   8.3267e-16
    9        0.45359612142557725889   5.5511e-17
   10        0.45359612142557725889   5.5511e-17
   11        0.45359612142557725889   5.5511e-17
   12        0.45359612142557725889   5.5511e-17
   13        0.45359612142557725889   5.5511e-17
   14        0.45359612142557725889   5.5511e-17
No equality!
```

```
0.45359612142557726
```

Question: Why does it stop here?

This loop would continue forever.

Taylor Series: Better:

```python
def taylor_better(x):
    tol = 1e-8
    T = 0
    for k in range(20):
        term = (-1)**(k)*x**(2*k)/math.factorial(2*k)
        T += term
        print("{:5d} {:30.20f} {:12.4e}".format(k,T,abs(term)))
        if abs(term) < tol:
            print('{:d} terms are needed'.format(k))
            return T
    print("No equality!")
    return T

taylor_better(1.1)
```

```
    0          1.00000000000000000000    1.0000e+00
    1          0.39499999999999990674    6.0500e-01
    2          0.45600416666666659937    6.1004e-02
    3          0.45354366527777772999    2.4605e-03
    4          0.45359682968278763893    5.3164e-05
    5          0.45359611491689805218    7.1477e-07
    6          0.45359612146891870044    6.5520e-09
6 terms are needed
```

```
0.4535961214689187
```

Stop when a reasonable tolerance has been achieved.

Example: Geometric Series

Compute $S_n = \sum_{k=0}^{n} x^k$

exact mathematical expression

$$= 1 + x + x^2 + \cdots + x^n = \frac{1-x^{n+1}}{1-x}$$

```python
def geoseries(x,n):
    strue = (1-x**n)/(1-x)
    S = 0
    for i in range(n):
        S = S*x + 1
    if (S == strue):
        print("Exact sum! S = {:24.20f}".format(S))
    else:
        print("Close : S     = {:24.20f}".format(S))
        print("Close : error = {:24.4e}".format(abs(S-strue)))
```

```python
geoseries(7.3,5)
```

```
Close : S     =  3290.43110000000024228939
Close : error =                 4.5475e-13
```

```python
geoseries(7.1,5)
```

```
Exact sum! S =  2957.58909999999968931661
```

Since we don't know if we can compute a value exactly, we should never use exact equality comparisons with real numbers.

② Catastrophic Cancellation

Try the following:

$$y = \frac{(1+x) - 1}{x} \quad \text{for } x = 10^{-8}$$

We expect to get $y = 1$. But instead, we get:

```
x = 1e-8
y = ((1 + x) - 1)/x
print('y = {:24.20f}'.format(y))
```

```
y =    0.99999999392252902908
```

What goes wrong?

$$y = \frac{(1+x)-1}{x}$$

How do we carry out these operations?

Step 1: Compute $1+x$

16 digits.

$$\begin{array}{rll} & 1.0000000000000000 & x \\ + & 0.0000000100000000 & 10^{-8} \\ \hline & 1.0000000100000000 & \end{array}$$

So far, so good.

Step 2: Subtract 1

??

$$\begin{array}{rl} & 1.0000000100000000 \\ - & 1.0000000000000000 \\ \hline & 0.0000000100000000\,\underline{XXXXXXXX} \end{array}$$

only 8 good digits

$$= 1.00000000\underbrace{\underline{XXXXXXXX}}_{\text{Junk}} \times 10^{-8}$$

Step 3: Divide through by $x = 10^{-8}$

Result: $1.00000000\underbrace{\underline{XXXXXXXX}}_{\text{Junk!}} \times 10^{0}$

Problem occurs because we try to subtract two values that are very close in magnitude

$$1+x \approx 1 \quad , \quad x \ll 1$$

This results in a "catastrophic" loss of accuracy. Matters are only made worse by multiplying by a large number.

Example: Use a "finite difference" formula to compute the derivative of $f(x) = x$:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad , \quad h \ll 1$$

from Calculus!

h is very small.

We know that $f'(x) = 1$.

But due to catastrophic cancellation for small values of h, we get

```python
def f(x):
    return x

x = 1
h = 1e-8
((f(x+h) - f(x))/h)
```

```
0.999999993922529
```

junk!

or an error of about 10^{-9}.

This loss of accuracy limits the usefulness of finite difference formula's for very small values of h.

Example: The quadratic formula:

Solve $0.002x^2 - 47.91x + 6 = 0$

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

If b^2 is much larger than $4ac$,
we have

$$\sqrt{b^2 - 4ac} \approx |b|$$

If $b > 0$, we can expect a loss of accuracy when computing

$$-b + |b|$$

If $b < 0$, we expect to lose accuracy computing

$$-b - |b|$$

Quadratic Formula, continued:

$a = 0.002 \quad b = -47.91 \quad c = 6$

Since $b < 0$, the computation of

$$-b + \sqrt{b^2 - 4ac} \qquad , |b| + |b|$$

does not lose any accuracy. But the computation of

$$-b - \sqrt{b^2 - 4ac} \qquad , |b| - |b|$$

will lose several digits of accuracy.

A better way to compute x:

The roots satisfy $x_1 x_2 = \frac{c}{a}$

Compute x_1 using the quadratic formula; Compute x_2 using

$$x_2 = \frac{c}{a x_1}$$

Quadratic Formula, continued.

Better way to compute roots:

```
def quadratic_formula(a,b,c):
    if b < 0:
        x1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
        x2 = c/(a*x1)
    else:
        x2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
        x1 = c/(a*x2)
    return x1, x2

a = 0.002
b = -47.91
c = 6
x1,x2 = quadratic_formula(a,b,c)
x2_bad = (-b - sqrt(b**2 - 4*a*c))/(2*a)
print('x1            = {:24.16f}'.format(x1))
print('x2            = {:24.16f}'.format(x2))
print('x2 (bad)      = {:24.16f}'.format(x2_bad))
print('Difference    = {:24.4e}'.format((abs(x2-x2_bad))))
```

right way to compute roots

```
x1           =    23954.8747645299954456
x2           =        0.1252354700030452
x2 (bad)     =        0.1252354700032043
Difference   =                1.5918e-13
```

good.

Difference between good and bad.

③ Floating point arithmetic is not associative, and not always commutative!

Associative property:

$$(a+b)+c = a+(b+c)$$

operations carried out in a different order.

Commutative:

$$a+b+c = b+c+a = a+c+b = \ldots$$

(all possible arrangements)

Try this

$$x_1 = .2+(.3+.1)$$

$$x_2 = (.2+.3)+.1$$

are these the same?

Associativity - cont.

```python
x1 = 0.2 + (0.3 + 0.1)
x2 = (0.2 + 0.3) + 0.1
if x1 == x2:
    print("x1 == x2")
else:
    print("What happened?  x1 != x2")
```

```
What happened?  x1 != x2
```

```python
print("{:24.20f}".format(x1))
print("{:24.20f}".format(x2))
```

```
0.60000000000000008882
0.59999999999999997780
```

} only approximately equal.

Order in which numbers are added can matter!

Rule : We can only ever expect two real numbers, when represented on the computer, to be **approximately equal**

the end.