

Floating-point arithmetic

As a data analyst, you will be concerned primarily with *numerical programs* in which the bulk of the computational work involves floating-point computation. This notebook guides you through some of the most fundamental concepts in how computers store real numbers, so you can be smarter about your number crunching.

WYSInnWYG, or "what you see is not necessarily what you get."

One important consequence of a binary format is that when you print values in base ten, what you see may not be what you get! For instance, try running the code below.

This code invokes Python's [decimal](#) package, which implements base-10 floating-point arithmetic in software.

```
In [2]: from decimal import Decimal
```

```
In [3]: x = 1.0 + 2.0**(-52)

print(x)
print(Decimal(x)) # What does this do?
```

```
1.00000000000000002
1.0000000000000002220446049250313080847263336181640625
5.551115123125782702118158340E-18
```

Aside: If you ever need true decimal storage with no loss of precision (e.g., an accounting application), turn to the `decimal` package. Just be warned it might be slower. See the following experiment for a practical demonstration.

```
In [4]: from random import random

NUM_TRIALS = 250000

print("Native arithmetic:")
A_native = [random() for _ in range(NUM_TRIALS)]
B_native = [random() for _ in range(NUM_TRIALS)]
%timeit [a+b for a, b in zip(A_native, B_native)]

print("\nDecimal package:")
A_decimal = [Decimal(a) for a in A_native]
B_decimal = [Decimal(b) for b in B_native]
```

```
Native arithmetic:
202 ms ± 10.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
Decimal package:
637 ms ± 13.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The same and not the same. Consider the following two program fragments:

Program 1:

```
s = a - b
t = s + b
```

Program 2:

```
s = a + b
t = s - b
```

Let $a = 1.0$ and $b = \epsilon_d/2 \approx 1.11 \times 10^{-16}$. $b = \epsilon_d/2 \approx 1.11 \times 10^{-16}$, i.e., machine epsilon for IEEE-754 double-precision. Recall that we do not expect these programs to return the same value; let's run some Python code to see.

Note: The IEEE standard guarantees that given two finite-precision floating-point values, the result of applying any binary operator to them is the same as if the operator were applied in infinite-precision and then rounded back to finite-precision. The precise nature of rounding can be controlled by so-called *rounding modes*; the default rounding mode is "[round-half-to-even](#)."

```
In [5]: a = 1.0
b = 2.*(-53) # == $epsilon_d$ / 2.0

s1 = a - b
t1 = s1 + b

s2 = a + b
t2 = s2 - b

print("s1:", s1.hex())
print("t1:", t1.hex())
print("\n")
```

```
print("s2:", s2.hex())
print("t2:", t2.hex())

print("")
print("s1: 0x1.fffffffffffffp-1")
print("t1: 0x1.000000000000p+0")

s2: 0x1.000000000000p+0
t2: 0x1.fffffffffffffp-1

1.0 vs. 0.9999999999999999
(t1 == t2) == False
```

By the way, the NumPy/SciPy package, which we will cover later in the semester, allows you to determine machine epsilon in a portable way. Just note this fact for now.

Here is an example of printing machine epsilon for both single-precision and double-precision values.

```
In [6]: import numpy as np

EPS_S = np.finfo (np.float32).eps
EPS_D = np.finfo (float).eps

print("Single-precision machine epsilon:", float(EPS_S).hex(), "~ ", EPS_S)
Single-precision machine epsilon: 0x1.000000000000p-23 ~ 1.1920929e-07
Double-precision machine epsilon: 0x1.000000000000p-52 ~ 2.220446049250313e-16
```

Analyzing floating-point programs

Let's say someone devises an algorithm to compute $f(x)$. For a given value x , let's suppose this algorithm produces the value $alg(x)$. One important question might be, is that output "good" or "bad"?

Forward stability. One way to show that the algorithm is good is to show that

$$\frac{|alg(x) - f(x)|}{|alg(x) - f(x)|}$$

is "small" for all x of interest to your application. What is small depends on context. In any case, if you can show it then you can claim that the algorithm is *forward stable*.

Backward stability. Sometimes it is not easy to show forward stability directly. In such cases, you can also try a different technique, which is to show that the algorithm is, instead, *backward stable*.

In particular, $alg(x)$ is a *backward stable algorithm* to compute $f(x)$ if, for all x , there exists a "small" Δx such that

$$alg(x) = f(x + \Delta x).$$

In other words, if you can show that the algorithm produces the exact answer to a slightly different input problem (meaning Δx is small, again in a context-dependent sense), then you can claim that the algorithm is backward stable.

Round-off errors. You already know that numerical values can only be represented finitely, which introduces round-off error. Thus, at the very least we should hope that a scheme to compute $f(x)$ is as insensitive to round-off errors as possible. In other words, given that there will be round-off errors, if you can prove that $alg(x)$ is either forward or backward stable, then that will give you some measure of confidence that your algorithm is good.

Here is the "standard model" of round-off error. Start by assuming that every scalar floating-point operation incurs some bounded error. That is, let $a \odot b$ be the exact mathematical result of some operation on the inputs, a and b , and let $fl(a \odot b)$ be the *computed* value, after rounding in finite-precision. The standard model says that

$$fl(a \odot b) \equiv (a \odot b)(1 + \delta),$$

$$fl(a \odot b) \equiv (a \odot b)(1 + \delta),$$

where $|\delta| \leq \epsilon$, machine epsilon.

Let's apply these concepts on an example.

Example: Computing a sum

Let $x \equiv (x_0, \dots, x_{n-1})$ be a collection of input data values. Suppose we wish to compute their sum.

The exact mathematical result is

$$f(x) \equiv \sum_{i=0}^{n-1} x_i.$$

$$f(x) \equiv \sum_{i=0}^{n-1} x_i.$$

Given x , let's also denote its exact sum by the synonym $s_{n-1} \equiv f(x)$.

Now consider the following Python program to compute its sum:

```
In [7]: def alg_sum(x): # x == x[:n]
        s = 0.
        for x_i in x: # x_0, x_1, \ldots, x_{n-1}
```

In exact arithmetic, meaning without any rounding errors, this program would compute the exact sum. (See also the note below.) However, you know that finite arithmetic means there will be some rounding error after each addition.

Let δ_i denote the (unknown) error at iteration i . Then, assuming the collection x represents the input values exactly, you can show that $\text{alg_sum}(x)$ computes \hat{s}_{n-1} where

$$\hat{s}_{n-1} \approx s_{n-1} + \sum_{i=0}^{n-1} s_i \delta_i,$$

$$s^{n-1} \approx s_{n-1} + \sum_{i=0}^{n-1} s_i \delta_i,$$

that is, the exact sum *plus* a perturbation, which is the second term (the sum). The question, then, is under what conditions will this sum will be small?

Using a *backward error analysis*, you can show that

$$\hat{s}_{n-1} \approx \sum_{i=0}^{n-1} x_i (1 + \Delta_i) = f(x + \Delta),$$

$$s^{n-1} \approx \sum_{i=0}^{n-1} x_i (1 + \Delta_i) = f(x + \Delta),$$

where $\Delta \equiv (\Delta_0, \Delta_1, \dots, \Delta_{n-1}) \Delta \equiv (\Delta_0, \Delta_1, \dots, \Delta_{n-1})$. In other words, the computed sum is the exact solution to a slightly different problem, $x + \Delta$.

To complete the analysis, you can at last show that

$$|\Delta_i| \leq (n-i)\epsilon,$$

$$|\Delta| \leq (n-1)\epsilon,$$

where ϵ is machine precision. Thus, as long as $n\epsilon \ll 1$, then the algorithm is backward stable and you should expect the computed result to be close to the true result. Interpreted differently, as long as you are summing $n \ll \frac{1}{\epsilon}$ values, then you needn't worry about the accuracy of the computed result compared to the true result:

```
In [8]: print("Single-precision: 1/epsilon_s ~= {:.1f} million".format(1e-6 / EPS_S))
```

```
Single-precision: 1/epsilon_s ~= 8.4 million
Double-precision: 1/epsilon_d ~= 4.5 quadrillion
```

Based on this result, you can probably surmise why double-precision is usually the default in many languages.

In the case of this summation, we can quantify not just the *backward error* (i.e., $\Delta_i \Delta_i$) but also the *forward error*. In that case, it turns out that

$$|\hat{s}_{n-1} - s_{n-1}| \lesssim n\epsilon \|x\|_1.$$

$$|s^{n-1} - s_{n-1}| \lesssim n\epsilon \|x\|_1.$$

Note: Analysis in exact arithmetic. We claimed above that $\text{alg_sum}()$ is correct *in exact arithmetic*, i.e., in the absence of round-off error. You probably have a good sense of that just reading the code.

However, if you wanted to argue about its correctness more formally, you might do so as follows using the technique of [proof by induction](#). When your loops are more complicated and you want to prove that they are correct, you can often adapt this technique to your problem.

First, assume that the `for` loop enumerates each element $p[i]$ in order from $i=0$ to $n-1$, where $n=\text{len}(p)$. That is, assume p_i is $p[i]$.

Let $p_k \equiv p[k]$ be the k -th element of p . Let $s_i \equiv \sum_{k=0}^i p_k$; in other words, s_i is the *exact* mathematical sum of $p[0:i+1]$. Thus, s_{n-1} is the exact sum of p .

Let \hat{s}_{i-1} denote the initial value of the variable s , which is 0. For any $i \geq 0$, let \hat{s}_i denote the *computed* value of the variable s immediately after the execution of line 4, where $i = i$. When $i = 0$, $\hat{s}_0 = \hat{s}_{-1} + p_0 = p_0$, which is the exact sum of $p[0:1]$. Thus, $\hat{s}_0 = s_0$.

Now suppose that $\hat{s}_{i-1} = s_{i-1}$. When $i = i$, we want to show that $\hat{s}_i = s_i$. After line 4 executes, $\hat{s}_i = \hat{s}_{i-1} + p_i = s_{i-1} + p_i = s_i$. Thus, the computed value \hat{s}_i is the exact sum s_i .

If $i = n$, then, at line 5, the value $S = \hat{s}_{n-1} = s_{n-1}$, and thus the program must in line 5 return the exact sum.

A numerical experiment: Summation

Let's do an experiment to verify that these bounds hold.

Exercise 0 (2 points). In the code cell below, we've defined a list,

```
N = [10, 100, 1000, 10000, 100000, 1000000, 10000000]
```

- Take each entry $N[i]$ to be a problem size.
- Let t be a list, which will hold computed sums.
- For each $N[i]$, run an experiment where you sum a list of values $x[N[i]]$ using $\text{alg_sum}()$. You should initialize x so that all elements have the value 0.1. Store the computed sum in $t[i]$.

```
In [9]: N = [10, 100, 1000, 10000, 100000, 1000000, 10000000]
```

```
# Initialize an array t of size len(N) to all zeroes.
t = [0.0] * len(N)

# Your code should do the experiment described above for
# each problem size N[i], and store the computed sum in t[i].

#
# YOUR CODE HERE
#
```

```
[0.9999999999999999, 9.999999999999998, 99.9999999999986, 1000.0000000001588, 10000.000000018848, 100000.00000133288, 999999.9998389754]
```

```
In [10]: # Test: `experiment_results`
import pandas as pd
from IPython.display import display

import matplotlib.pyplot as plt
%matplotlib inline

s = [1., 10., 100., 1000., 10000., 100000., 1000000.] # exact sums
t_minus_s_rel = [(t_i - s_i) / s_i for s_i, t_i in zip(s, t)]
rel_err_computed = [abs(r) for r in t_minus_s_rel]
rel_err_bound = [n*EPS_D for n in N]

# Plot of the relative error bound
plt.loglog(N, rel_err_computed, 'b*', N, rel_err_bound, 'r--')

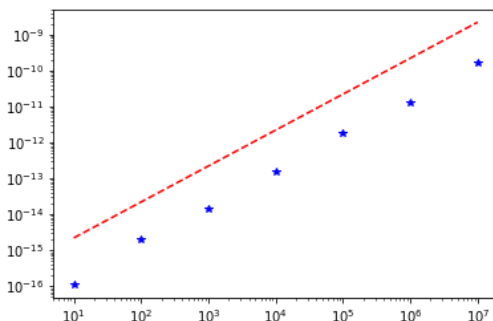
print("Relative errors in the computed result:")
display(pd.DataFrame({'n': N, 'rel_err': rel_err_computed, 'rel_err_bound': [n*EPS_D for n in N]}))

assert all([abs(r) <= n*EPS_D for r, n in zip(t_minus_s_rel, N)])
```

Relative errors in the computed result:

	n	rel_err	rel_err_bound
0	10	1.110223e-16	2.220446e-15
1	100	1.953993e-15	2.220446e-14
2	1000	1.406875e-14	2.220446e-13
3	10000	1.588205e-13	2.220446e-12
4	100000	1.884837e-12	2.220446e-11
5	1000000	1.332883e-11	2.220446e-10
6	10000000	1.610246e-10	2.220446e-09

(Passed!)



Computing dot products

Let x and y be two vectors of length n , and denote their dot product by $f(x, y) \equiv x^T y$.

Now suppose we store the values of x and y *exactly* in two Python arrays, $x[0:n]$ and $y[0:n]$. Further suppose we compute their dot product by the program, `alg_dot()`.

```
In [11]: def alg_dot(x, y):
p = [xi*yi for (xi, yi) in zip(x, y)]
s = alg_sum(p)
```

Exercise 1 (OPTIONAL -- 0 points, not graded or collected). Show under what conditions `alg_dot()` is backward stable.

Hint. Let (x_k, y_k) denote the exact values of the corresponding inputs, $(x[k], y[k])$. Then the true dot product, $x^T y = \sum_{k=0}^{n-1} x_k y_k$. Next, let \hat{p}_k denote the k -th computed product, i.e., $\hat{p}_k \equiv x_k y_k (1 + \gamma_k)$, where γ_k is

the k -th round-off error and $|\gamma_k| \leq \epsilon |y_k| \leq \epsilon$. Then apply the results for `alg_sum()` to analyze `alg_dot()`.

Answer. Following the hint, `alg_sum` will compute $\hat{s}_{n-1} s^{n-1}$ on the *computed* inputs, $\{\hat{p}_k\} \{p^k\}$. Thus,

$$\begin{aligned}\hat{s}_{n-1} &\approx \sum_{i=0}^{n-1} \hat{p}_i (1 + \Delta_i) \\ &= \sum_{i=0}^{n-1} x_i y_i (1 + \gamma_i) (1 + \Delta_i) \\ &= \sum_{i=0}^{n-1} x_i y_i (1 + \gamma_i + \Delta_i + \gamma_i \Delta_i) \\ s^{n-1} &\approx \sum_{i=0}^{n-1} p_i (1 + \Delta_i) = \sum_{i=0}^{n-1} x_i y_i (1 + \gamma_i) (1 + \Delta_i) = \sum_{i=0}^{n-1} x_i y_i (1 + \gamma_i + \Delta_i + \gamma_i \Delta_i).\end{aligned}$$

Mathematically, this appears to be the exact dot product to an input in which xx is exact and yy is perturbed (or vice-versa). To argue that `alg_dot` is backward stable, we need to establish under what conditions the perturbation, $|\gamma_i + \Delta_i + \gamma_i \Delta_i| |y_i + \Delta_i + \gamma_i \Delta_i|$, is "small." Since $|\gamma_i| \leq \epsilon |y_i| \leq \epsilon$ and $|\Delta_i| \leq n\epsilon |\Delta_i| \leq n\epsilon$,

$$\begin{aligned}|\gamma_i + \Delta_i + \gamma_i \Delta_i| &\leq |\gamma_i| + |\Delta_i| + |\gamma_i| \cdot |\Delta_i| \leq (n+1)\epsilon + \epsilon(n\epsilon^2) \approx (n+1)\epsilon. \\ |y_i + \Delta_i + \gamma_i \Delta_i| &\leq |y_i| + |\Delta_i| + |\gamma_i| \cdot |\Delta_i| \leq (n+1)\epsilon + O(n\epsilon^2) \approx (n+1)\epsilon.\end{aligned}$$

More accurate summation

Suppose you wish to compute the sum, $s = x_0 + x_1 + x_2 + x_3$ $s = x_0 + x_1 + x_2 + x_3$. Let's say you use the "standard algorithm," which accumulates the terms one-by-one from left-to-right, as done by `alg_sum()` above.

For the standard algorithm, let the i -th addition incur a roundoff error, δ_i . Then our usual error analysis would reveal that the absolute error in the computed sum, \hat{s} , is approximately:

$$\begin{aligned}\hat{s} - s &\approx x_0(\delta_0 + \delta_1 + \delta_2 + \delta_3) + x_1(\delta_1 + \delta_2 + \delta_3) + x_2(\delta_2 + \delta_3) + x_3\delta_3. \\ s^{\wedge} - s &\approx x_0(\delta_0 + \delta_1 + \delta_2 + \delta_3) + x_1(\delta_1 + \delta_2 + \delta_3) + x_2(\delta_2 + \delta_3) + x_3\delta_3.\end{aligned}$$

And since $|\delta_i| \leq \epsilon |\delta_i| \leq \epsilon$, you would bound the absolute value of the error by,

$$\begin{aligned}|\hat{s} - s| &\lesssim (4|x_0| + 3|x_1| + 2|x_2| + 1|x_3|)\epsilon. \\ |s^{\wedge} - s| &\lesssim (4|x_0| + 3|x_1| + 2|x_2| + 1|x_3|)\epsilon.\end{aligned}$$

Notice that $|x_0| |x_0|$ is multiplied by 4, $|x_1| |x_1|$ by 3, and so on.

In general, if there are n values to sum, the $|x_i| |x_i|$ term will be multiplied by $n - i + 1$.

Exercise 2 (3 points). Based on the preceding observation, implement a new summation function, `alg_sum_accurate(x)` that computes a more accurate sum than `alg_sum()`.

Hint 1. You do **not** need `Decimal()` in this problem. Some of you will try to use it, but it's not necessary.

Hint 2. Some of you will try to "implement" the error formula to somehow compensate for the round-off error. But that shouldn't make sense to do. (Why not? Because the formula above is a *bound*, not an exact formula.) Instead, the intent of this problem is to see if you can look at the formula and understand how to interpret it. That is, what does the formula tell you?

```
In [ ]: def alg_sum_accurate(x):
        assert type(x) is list
        #
        # YOUR CODE HERE
        #

In [ ]: # Test: `alg_sum_accurate_test`
from math import exp
from numpy.random import lognormal

print("Generating non-uniform random values...")
N = [10, 10000, 10000000]
x = [lognormal(-10.0, 10.0) for _ in range(max(N))]
print("Range of input values: [{}, {}].format(min(x), max(x))

print("Computing the 'exact' sum. May be slow so please wait...")
x_exact = [Decimal(x_i) for x_i in x]
s_exact = [float(sum(x_exact[:n])) for n in N]
print("==>", s_exact)

print("Running alg_sum()...")
s_alg = [alg_sum(x[:n]) for n in N]
print("==>", s_alg)

print("Running alg_sum_accurate()...")
s_acc = [alg_sum_accurate(x[:n]) for n in N]
print("==>", s_acc)

print("Summary of relative errors:")
ds_alg = [abs(s_a - s_e) / s_e for s_a, s_e in zip(s_alg, s_exact)]
```

```
ds_acc = [abs(s_a - s_e) / s_e for s_a, s_e in zip(s_acc, s_exact)]
display (pd.DataFrame({'n': N,
                      'rel_err(alg_sum)': ds_alg,
                      'rel_err(alg_sum_accurate)': ds_acc}))

assert all([r_acc < r_alg for r_acc, r_alg in zip(ds_acc[1:], ds_alg[1:])], \
           "The 'accurate' algorithm appears to be less accurate than the conventional one!")
```

Generating non-uniform random values...

Range of input values: [3.071408381684977e-28, 1.5742743204836618e+18]

Computing the 'exact' sum. May be slow so please wait...

Done! You have reached the end of Part 1. There are no additional parts, so if you are satisfied, be sure to submit both parts, declare victory, and move on!