1.6 Rounding

floating point numbers

- recall that we are computing in the number system $\mathbb{F}_B(t)$ with $B = 2$ or $B = 10$
- a typical $x \in \mathbb{F}_B(t) \setminus \{0\}$ is of the form

$$x = \pm \left(\frac{x_1}{B} + \frac{x_2}{B^2} + \cdots + \frac{x_t}{B^t} \right) B^e$$

  with exponents $e \in \mathbb{Z}$ and digits $x_k \in \{0, \ldots, B-1\}$ and $x_1 \neq 0$
- in short we write this as

$$x = 0.x_1 x_2 \ldots x_t \, B^e$$

  - for example one has $x = -0.4521 \cdot 10^5$

Rounding function $\phi(x)$

- mapping

$$\phi : \mathbb{R} \to \mathbb{F}_B(t)$$

  - there are many different rounding functions including truncation
    - example: $\phi(0.346) = 0.34$
  - set of floating point numbers discrete, thus rounding function piecewise constant
- the rounding error for some $x \in \mathbb{R}$ is then

$$\epsilon(x) = \phi(x) - x$$

  - error piecewise linear, not continuous
  - rounding errors of complicated expressions can be hard to predict
  - $\phi(x)$ monotone: $\phi(x) \leq \phi(y)$ if $x \leq y$

Questions

- When do you use rounding?
- In projects, when estimating costs and time?
- How many digits do you usually require?
- What happens if we do not round?

Optimal Rounding

- A rounding function ϕ is called *optimal* if

$$|\phi(x) - x| \leq |y - x|, \quad y \in \mathbb{F}_B(t)$$

- Truncation is not optimal: e.g., for $\mathbb{F}_{10}(2)$ one has for $\phi =$ truncation

$$\phi(0.4563) = 0.45$$

but

$$|0.45 - 0.4563| = 0.0063 > |0.46 - 0.4563| = 0.0037$$

Proposition *An optimal rounding satisfies*

$$\phi(x) - x = \delta x$$

where $|\delta| \leq 0.5B^{-t+1}$

Proof.

- if $x \in \mathbb{F}_B(t)$ then $\phi(x) = x$ by optimality
- if $x \notin \mathbb{F}_B(t)$ then $\phi(x) \in \{x_1, x_2\}$ where $x_i \in \mathbb{F}_B(t)$ are the two closest numbers to x
- by optimality $|\phi(x) - x| \leq |x_2 - x_1|/2$
- If $x > 0$ one has $x = \sum_{j=1}^{\infty} c_j B^{-j+e}$ and

$$x_1 = \sum_{j=1}^{t} c_j B^{-j+e} < x_2 = x_1 + B^{-t+e}$$

  and so $(x_2 - x_1)/2 = 0.5B^{-t+e}$ and similar for $x < 0$
- Normalisation: $B^{-1+e} \leq c_1 B^{-1+e} \leq x$ and thus

$$\frac{x_2 - x_1}{2x} \leq \frac{0.5B^{-t+e}}{B^{-1+e}} = 0.5B^{-t+1}$$

- Optimal rounding is not unique, for example, if $x = 0.745$ both both 0.74 and 0.75 are optimal (in $\mathbb{F}_{10}(2)$)
    - a common choice is in this case to select the number with even least significant digit, i.e., 0.74

examples

1. floating point decimal numbers in $\mathbb{F}_{10}(2)$:
   - $\phi(3.452) = 3.5$
   - $\phi(0.675) = 0.68$
   - $\phi(1/9) = 0.11$
2. floating point binary number in $\mathbb{F}_2(3)$:
   - $\phi(3.1875) = 3$ as $3.1875 = 0.110011_2 \cdot 2^2$ and $\phi(0.110011_2 \cdot 2^2) = 0.11_2 \cdot 2^2 = 3$

A property of floating point rounding

Lemma

If $\phi_0 : \mathbb{R} \to \mathbb{Z}$ and $\phi : \mathbb{R} \to \mathbb{F}_B(t)$ are rounding functions (with consistent rounding of midpoints) and e is the exponent of $x \in \mathbb{R}$ (normalised) then

$$\phi(x) = \phi_0(B^{t-e}x)/B^{t-e}$$

for proof use: $\{B^{t-1}, \dots, B^t - 1\} = \{B^{t-e}y \mid y \in \mathbb{F}_B(t)\} \in \mathbb{Z}$

```python
# Rounding to Fl_2(t) in Python:

def roundfl2(number, ndigits=1):
    import math
    (xm, xe) = math.frexp(number)
    xr = round(xm*2.0**ndigits)/2.0**ndigits
    return math.ldexp(xr, xe)

roundfl2(3.1875, ndigits=4)
```

```
3.25
```

using Python decimal module for rounding in $\mathbb{F}_{10}(t)$

- Python *decimal* which implements floating point numbers
- we use this module to implement a decimal rounding function
- output in floating point thus additional error

```
# decimal rounding of binary floating point numbers

def roundfl10(x, t=1):
    from decimal import Context
    return float(Context(prec=t).create_decimal(x))

roundfl10(3.1875, t=3)

3.19
```

Applications of Rounding to elementary unary and binary functions

As $\mathbb{F}_B(t)$ is not a ring, we need to approximate all arithmetic operations and an optimal approximation of these operations is

- for example, we replace the sum $x + y$ by $\phi(x + y)$, and the same for multiplications
- any unary function evaluations are also done using rounding, e.g. replace $\sin(x)$ by $\phi(\sin(x))$

In order to assess the error caused through rounding one uses the proposition above to get

- for the binary function evaluations: $(1+\delta_1)(x+y)$
- for unary function evaluations: $(1+\delta_2)\sin(x)$

(of course, the δ_i are not the same but in an ideal case, they are bounded by the same constant)

- the δ_i characterise the relative rounding error which occurs when the functions are done on a computer

arithmetic operations in $\mathbb{F}_B(t)$

- difference in $\mathbb{F}_B(t)$ exact only in exceptional circumstances, a notable case is where x and y are very close as in this case the difference of x and y is also in $\mathbb{F}_B(t)$
- the product $x * y$ of two numbers in $\mathbb{F}_B(t)$ is in $\mathbb{F}_B(2t)$
- the quotient will typically be a floating point number with an infinite number of digits
- IEEE 754 standard suggests that best approximation using rounding should be used to implement arithmetic operations

thus replace any $x \circ y$ by $\phi(x \circ y)$

properties of approximate arithmetic

- commutative law holds for addition and multiplication in $\mathbb{F}_B(t)$

$$\phi(xy) = \phi(yx)$$

- associative law for addition does not hold
  - e.g. $\phi(x + \phi(y + z)) \neq \phi(\phi(x + y) + z)$
  - e.g. in $\mathbb{F}_{10}(3)$

$$\phi(\phi(1.32+0.254)+0.392) = 1.96 \neq 1.97 = \phi(1.32+\phi(0.254+0.392))$$

- neither associative law for multiplication nor the distributive law hold

simple functions

- IEEE 754 also requires that simple functions like exp, sin etc are implemented such that they are the rounded version of the exact function
- For example, we may define $\sin_{\mathbb{F}}(x) := \phi(\sin(x))$ where $\phi : \mathbb{R} \to \mathbb{F}$ is a rounding function

Questions

The next section deals with numeric expressions.

- What is the significance of such expressions, where are they used?
- What is the math behind the expressions?
- Can you think of anything related to your studies, work and life where such expressions play a role?

1.7 error analysis of expressions

- recall: floating point numbers have only a finite fixed number of digits in mantissa
  - consequence: computers need to round almost every arithmetic operation and function evaluation
- most real numbers and even rational numbers (like $1/5$) are not floating point numbers
  - consequence: computers have to round almost all inputs
- the resulting *rounding errors* are unavoidable and occur in every computation

In the following we analyse these errors
Consider, for example the evaluation of

$$f(x) = 2\sin(x_1 x_2) + x_3$$

where

$$x_1 = 3.57, \quad x_2 = 0.0723, \quad \textit{and} \quad x_3 = 1.0.$$

Evaluating this on your computer gives

```
from math import sin
2*sin(3.57*0.0723) + 1.0
```

```
1.5105091672725943
```

Step 1: rewrite expression as sequence of simple assignments

In a first step we rewrite the expression to be evaluated as a sequence of simple expressions of the form

$$
\begin{aligned}
u_0 &= f_0 \\
u_1 &= f_1(u_0) \\
u_2 &= f_2(u_0, u_1) \\
&\dots \\
u_n &= f_n(u_0, \dots, u_{n-1}).
\end{aligned}
$$

The functions f_k are either non-floating point constants, in our example 3.57 and 0.0723 but not 1.0 as this is a floating point number, or simple expressions which are evaluated with a rounding error, for example $2u_3 + 1$ but not $2u_3$ (which is evaluated exactly). In the case of our example we get

$$
\begin{aligned}
u_0 &= 3.57 \\
u_1 &= 0.0723 \\
u_2 &= u_0 u_1 \\
u_3 &= \sin(u_2) \\
u_4 &= 2u_3 + 1
\end{aligned}
$$

Step 2: include errors

We now rewrite the algorithm including rounding errors

- formula for rounding function

$$\phi(x) = (1+\delta)x$$

note: δ depends on x but satisfies

$$|\delta(x)| \leq \epsilon$$

With the substitution one then gets

$$\begin{aligned}
u_0 &= (1+\delta_0)\, f_0 \\
u_1 &= (1+\delta_1)\, f_1(u_0) \\
u_2 &= (1+\delta_2)\, f_2(u_0, u_1) \\
&\dots \\
u_n &= (1+\delta_n)\, f_n(u_0, \dots, u_{n-1}).
\end{aligned}$$

For our **example** we get

$$
\begin{aligned}
u_0 &= (1+\delta_0)\,3.57 \\
u_1 &= (1+\delta_1)\,0.0723 \\
u_2 &= (1+\delta_2)\,u_0 u_1 \\
u_3 &= (1+\delta_3)\,\sin(u_2) \\
u_4 &= (1+\delta_4)\,(2u_3+1)
\end{aligned}
$$

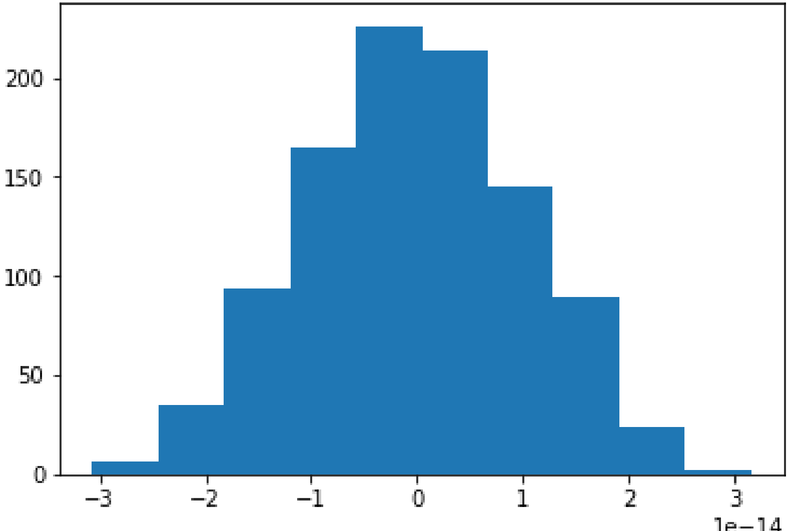
- result u_4 is polynomial in the δ_k
- study using simulation and derive error bounds
- we have avoided dealing with the discontinuous rounding functions!

```python
# study the effect of rounding errors on the result

def g(delta):

    u_0 = (1+delta[0])*3.57
    u_1 = (1+delta[1])*0.0723
    u_2 = (1+delta[2])*u_0*u_1
    u_3 = (1+delta[3])*sin(u_2)
    u_4 = (1+delta[4])*(2*u_3 + 1)

    return u_4
```

```python
# compute "exact" result
import numpy as np
u_4ex = g(np.zeros(5))
print("exact result: ", u_4ex)

# simulate using random rounding errors with uniform
#   distribution for uncertain epsilon[k]
n = 1000
error = np.zeros(n)
epsi = 1e-14
for k in range(n):
    delta = epsi*(np.random.random(5)*2-1)
    error[k] = g(delta) - u_4ex
```

```
exact result:  1.51050916727
```

```
%matplotlib inline
import pylab as plt
plt.hist(error);
```



Revision ideas:

- use above approach to analyse simple expressions like $a * b$ or $a + b + c$
- take a code you might have and include rounding errors to study their effect on the result
- any suggestions on how to automatically include rounding errors?