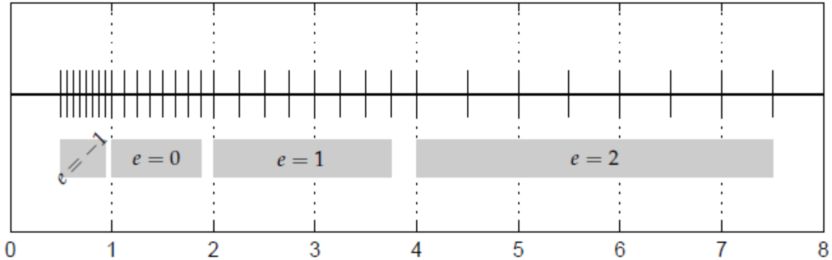
Chapter 1

Numbers, Problems and Algorithms



Objectives

- Learn how numbers are represented in the computer
- Examine consequences of floating point arithmetic
- Begin to study numerical algorithms
- Learn to identify when problems can cause numerical problems:
  - From subtraction of closely spaced numbers
  - From the problem itself: conditioning
  - From the numerical algorithm: stability

1.1

Floating point numbers

$f \in [0, 1)$

- The set ***F*** of floating point numbers is of the form $\pm(1+f)2^{-e}$
- e is the exponent, and is an integer.
- f is the mantissa, with $f = \sum_{i=1}^{d} b_i 2^{-i}$, with d binary digits
- b_i is a binary digit (0 or 1), i is the binary place
- Factoring out 2^{-d} we can rewrite f this way:

$$f = 2^{-d} \sum_{k=0}^{d-1} b_{d-k} 2^k = 2^{-d} z,$$

- In this form, z is an integer and $z \in \{0, 1, \ldots, 2^d - 1\}$
- Because of this, there are 2^d evenly-spaced numbers between 2^e and 2^{e+1}

$f = [0.b_1 b_2 \cdots b_d]_2$ eg $f = [0.101]_2 = \frac{1}{2} + \frac{0}{4} + \frac{1}{8} = \frac{5}{8}$

Properties of $\boldsymbol{F}$

- Keep in mind that $$f = 2^{-d} \sum_{k=0}^{d-1} b_{d-k} 2^k = 2^{-d} z,$$
- If $z = 1$, we are at the smallest number in the interval, so the first number bigger than unity is $1 + 2^{-d}$

$1 + \frac{1}{4} 2^{-d} = 1$

- That number 2^{-d} is special and it is denoted ϵ_M and called machine epsilon
- Define rounding $\mathrm{fl}(x)$ as converting real number x into the nearest member of $\boldsymbol{F}$
- Then one finds $|\mathrm{fl}(x) - x| \leq \frac{1}{2}(2^{e-d}) = 2^{e-d-1}$

$x \in [2^e, 2^{e+1})$

- Rearranging indicates small relative error:

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq \frac{2^{e-d-1}}{2^e} \leq \frac{1}{2}\epsilon_M. = \eta$$

$\eta = \frac{1}{2}\epsilon_M$ is the <u>unit roundoff</u>

Scientific notation, significant digits

- Consider Planck's constant given by 6.626068×10^{-34} m² kg/s. If we change the last digit by 1, then the relative change is

6.626069×10^{-34}

$$\frac{0.000001 \times 10^{-34}}{6.626068 \times 10^{-34}} \approx 1.51 \times 10^{-7}.$$

- The relative error is about 10^{-7}, so we can say that the original number had 7 significant digits.
- More generally,

$$\text{digits} = -\log_{10}\left|\frac{\tilde{x} - x}{x}\right| \qquad (1.1.5)$$

- This is different than decimal places.

[Example 1.1.2]

Double precision numbers

- IEEE standard 754 specifies how to store so-called double precision numbers
- 64 bits per number, d=52 digit mantissas, 11 digits for exponent e, and a sign bit.
- In this case, $\epsilon_M = 2^{-52} \approx 2.2 \times 10^{-16}$; this is about 16 digits
- Biggest number: $2^{1024} \approx 2 \times 10^{308}$
- If bigger, "overflow"
- Smallest number: $2^{-1022} \approx \underline{2 \times 10^{-307}}$
- If smaller, "underflow"
- How can we have any problem with arithmetic or algorithms with so many digits and such range?

$1+f$ normalized

$0+f$ denormalized

Floating point arithmetic

- Consider multiplication
- For two exact numbers x and y
- Exact product xy, floating product $\mathrm{fl}(xy)$
- One finds that

$$\frac{|\,\mathrm{fl}(xy) - (xy)|}{|xy|} \leq \epsilon_M$$

- This is a potential error in the 16th digit
- If we have very many operations, e.g. 10^{20} then it's possible that this could add up.
- Other operations are not as forgiving.

[Example 1.1.3]

1.2 Problems and condition numbers

- Putting the number x in the computer is $\mathrm{fl}(x) = x(1+\epsilon)$
- We can write that the computer implementation of $y = x + 1$ as $y = x(1+\epsilon) + 1$
- Then, the relative error becomes

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

- For x near -1, the relative error can become very large
- Say we have 5 digits and add -1.0012 to 1; then we get -1.2×10^{-3}
- Only two digits now are correct: subtractive cancellation!
- Important source of error!

$x = -1.0012435$

$\mathrm{fl}(x) = -1.0012$

Condition numbers

- We can measure how bad an operation or problem is with the *condition number*
- Let the exact number x become $\tilde{x} = \mathrm{fl}(x) = x(1+\epsilon)$
- Then considering only changes due to x, one gets

$|\varepsilon| \le \eta = \frac{1}{2}\varepsilon_M$

$$\frac{|f(x) - f(x(1+\epsilon))|}{|\epsilon f(x)|}$$

- In the limit of small error (ideal computer)

$$\kappa(x) = \lim_{\epsilon \to 0} \left| \frac{f(x+\epsilon x) - f(x)}{\epsilon f(x)} \right| = \left| \lim_{\epsilon \to 0} \frac{f(x+\epsilon x) - f(x)}{\epsilon x} \cdot \frac{x}{f(x)} \right| = \left| \frac{x f'(x)}{f(x)} \right|$$

- The condition number indicates the magnification of errors in computation $f(x)$: compares size of output to size of input

Compare $\alpha = \dfrac{|x-\tilde{x}|}{|x|}$ vs $\beta = \dfrac{|f(x)-f(\tilde{x})|}{|f(x)|}$.

$\dfrac{|x - x(1+\varepsilon)|}{|x|} = \varepsilon$

input error

output error

We know α will be small: $\alpha \le \frac{1}{2}\varepsilon_M$.

We would like β to be not much larger than α. ie $\beta = K\cdot\alpha$, K small

$$K = \frac{\beta}{\alpha} = \frac{|f(x)-f(\tilde{x})|}{|\varepsilon f(x)|}$$

$$\lim_{\varepsilon\to 0} \frac{|f(x)-f(x(1+\varepsilon))|}{|\varepsilon f(x)|} = \left|\frac{x f'(x)}{f(x)}\right| =: K_f(x)$$

$\therefore$ for $\varepsilon > 0$ small, $K \approx K_f(x)$.

$$\therefore \quad \frac{|f(x)-f(\tilde{x})|}{|f(x)|} \approx K_f(x)\,|\varepsilon|$$

error magnification

output error

input error

$K_f(x) \approx 10^d \implies$ "loss" of d digits of accuracy in computing $f(x)$

Condition number examples

- *Example*: Return to addition, and consider $f(x) = x - c$
- (Before, we had $c = -1$)
- Use

$$\kappa(x) = \left|\frac{xf'(x)}{f(x)}\right|$$

eg $\left|\frac{-1.0012435}{0.0012435}\right|$

$= 805.18\ldots$

- Applying the formula,

$$\kappa(x) = \left|\frac{(x)(1)}{x-c}\right| = \left|\frac{x}{x-c}\right|$$

$-\log_{10}(805) \approx -2.9$

digits lost

- The condition number is large when $x \approx c$; conditioning is poor there

Condition number examples

- *Example*: Multiplication by constant c, $f(x) = cx$.

Then

$$\kappa(x) = \left|\frac{xf'(x)}{f(x)}\right| = \left|\frac{(x)(c)}{cx}\right| = 1.$$

No magnification of error!

- *Example*: $f(x) = \cos(x)$:

$$\kappa(x) = \left|\frac{(x)(-\sin x)}{\cos x}\right| = |x \tan x|.$$

The condition number is large when $x \overset{\approx}{\cancel{=}} a\pi/2$, where a is an odd integer

or when $|x|$ is large and $\tan x \neq 0$.

Condition number examples

- *Example*: Effect of a on roots of quadratic equation $f(x) = ax^2 + bx + c = 0$.

Use implicit differentiation

$f(a) = r$

$$r^2 + 2ar\left(\frac{dr}{da}\right) + b\frac{dr}{da} = 0.$$

roots r_1, r_2

Solve for derivative,

$$\frac{dr}{da} = \frac{-r^2}{2ar + b} = \frac{-r^2}{\pm\sqrt{b^2 - 4ac}},$$

then solve use in condition number definition to get

$ar^2 + br + c = 0$

$$\kappa_{a \mapsto r} = \left|\frac{ar}{\sqrt{b^2 - 4ac}}\right|. = \left|\frac{r}{r_1 - r_2}\right|$$

Conditioning is poor for small discriminant, i.e., near double roots

1.3

Algorithms

efficiency
stability

- Consider evaluating polynomials.
- Evaluate polynomials by converting higher degrees to distributed products.
- *Example:* Consider $p(x) = ax^2 + bx + c.$

We can write $p(x) = (ax + b)x + c$ and evaluate the parens first.

- More generally,

$$\begin{aligned} p(x) &= c_1x^n + c_2x^{n-1} + \cdots + c_nx + c_{n+1} \\ &= \Big(\cdots((c_1x + c_2)x + c_3)x + \cdots + c_n\Big)x + c_{n+1}. \end{aligned}$$

- The second line suggests an algorithm

Horner's method

How efficient is Horner's algorithm?

$$p(x) = c_1 x^n + c_2 x^{n-1} + \cdots + c_n x + c_{n+1}$$

mults: $n + n-1 + \cdots + 1 + 0 = \frac{n(n+1)}{2}$

adds: n

Total ops: $\boxed{\frac{n(n+1)}{2} + n}$

```
p = c[1]
for k = 2:n+1
    p = p*x + c[k]
end
```

← 1 mult., 1 add. (2 ops)

For loop has n iterations.

Total ops: $\boxed{2n}$

Note: First method could be made more efficient by computing $x, x^2, \ldots, x^n$ each once.

```
xk = 1
p = c[n+1]
```

```
for k = n:-1:1
    xk = xk*x
    p = p + c[k]*xk
end
```

3 ops

Horner’s algorithm

```
function p = horner(c,x)
% HORNER  Evaluate polynomial using Horner's rule.
% Input:
%    c      Coefficients of polynomial, in descending order (vector)
%    x      Evaluation point (scalar)
% Output:
%    p      Value of the polynomial at x (scalar)

n = length(c);
p = c(1);
for k = 2:n
  p = x*p + c(k);
end
```

Horner’s algorithm

- *Example:* Consider $p(x) = (x-1)^3$. We can also write in expanded form

The coefficient matrix for matlab in expanded form is c=[1 -3 3 -1].

Using Matlab, and horner.m, with $x = 1.2$, we get the results at right.

y gives the result from the function, and the last line gives the absolute error, which is about the size of ϵ_M

[Example 1.3.2]

```
>> c = [1 -3 3 -1]
c =
     1    -3     3    -1
>> y = horner(c,1.2)
y =
   0.0080
>> (1.2-1)^3-y
ans =
  2.0990e-16
>> 
```

Stability

- Consider solving the quadratic formula again $ar^2 + br + c = 0$
- It the standard formula is used with $a = c = 1$ and $b = -(10^6 + 10^{-6})$, the exact answer is roots at $r_1 = 10^6$ and $r_2 = 10^{-6}$
- Numerically, the first root is exact in Matlab, but the second root has only 5 correct digits!
- We could do better by using the

following formula for r_1

$$r = \frac{-b - (\operatorname{sign} b)\sqrt{b^2 - 4ac}}{2a}$$

and then $r_2 = (c/a)/r$ will get the answers to many digits

[Example 1.3.3]

```
format long
a = 1;  b = -(1e6+1e-6);  c = 1;
```

The “good” root.

```
x1 = (-b + sqrt(b^2-4*a*c)) / (2*a);
```

The better formula for computing the other root.

```
x2 = c/(a*x1)
```

```
x2 =
     1.000000000000000e-06
```

Stability: quadratic equation

- First computation failed because numerator was difference of closely spaced numbers, which caused loss of significance (from subtractive cancellation).
- The loss of significance caused a much larger relative error than one may expect.
- Avoid the problem by using different formulas to calculate roots.
- Other situations benefit from changing the approach.

[Example 1,3,4]

Stability: approximate exponential integral

- Example from Moler for approximating the exponential integral.
- Use integration by parts to get recursive formula.
- Using the formula one way magnifies error so that approximation becomes negative (it can't) in just a few iterations. That way is *unstable* because it magnifies roundoff error.
- Rewriting the formula and using it differently minimizes error at each step and rapidly approaches the desired results: that approach is *stable*.
- We have to choose or design algorithms that are stable against roundoff error.

Example: (from Ascher/Greif, p. 13)

$$y_n = \int_0^1 \frac{x^n}{x+10}\,dx, \qquad n = 1, 2, \ldots, 30$$

[Note: $0 < y_n < 1,\ \forall n$]

$$y_n + 10\,y_{n-1} = \frac{1}{n}, \qquad y_0 = \ln(11) - \ln(10)$$

Algorithm:

$y[k] \equiv y_{k-1}$

```
y = zeros(31)
y[1] = ln(11) - ln(10)
for n = 2:31
    y[n] = 1/(n-1) - 10 y[n-1]
end
```

error x10 each iteration

This algorithm is <u>not</u> stable.

Stability and backward error

- Forward Error: algorithm $\tilde{f}(x)$ for problem $f(x)$ has forward error

$$\frac{\left|\tilde{f}(x) - f(x)\right|}{|f(x)|}$$

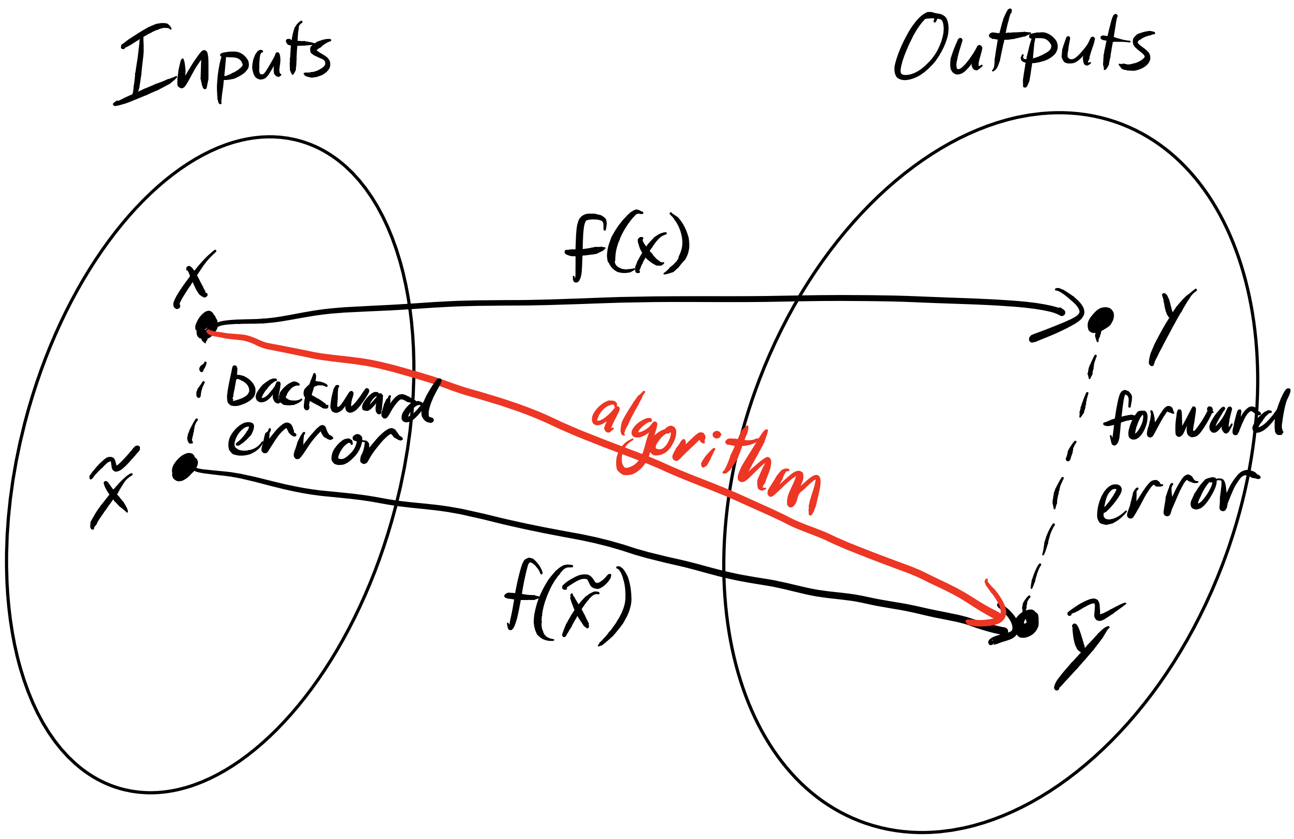
- Backward Error: Say we can find approximate input data such that

$$f(\tilde{x}) = \tilde{f}(x)$$

Then the backward error is

$$\frac{|\tilde{x} - x|}{|x|}$$

- If backward error is small, then the algorithm "gives the correct answer to nearly the right problem" (Trefethen and Bau).
- Polynomial example of text: forward error in roots is poorly conditioned at double root, but those roots satisfy a polynomial very close to original



If an algorithm is guaranteed to give a small <u>backward error</u>, we call it <u>backward stable</u>.

Later we will see that

small backward error	+	well conditioned problem	=	accurate sol'n.
(algorithm)		(problem)		

Stability and backward error

- Compute roots of 6th degree polynomial
- One pair is a double root
- Those roots have large forward error
- Using the roots to go backward and get coefficents gives very close polynomial

```
r = [-2 -1 1 1 3 6]';
p = poly(r)
```

```
p =
     1      -8       6      44     -43     -36      36
```

```
r_computed = sort( roots(p) )
```

```
r_computed =
   -2.0000
   -1.0000
    1.0000
    1.0000
    3.0000
    6.0000
```

```
abs(r - r_computed) ./ r
```

```
ans =
   1.0e-08 *
   -0.0000
   -0.0000
    0.8534
    0.8534
         0
    0.0000
```

```
(p_computed - p) ./ p
```

```
ans =
   1.0e-14 *
         0   -0.0777   -0.1628   -0.1130   -0.1157   -0.3158   -0.3355
```

[Example 1.3.5]

Stability and backward error

- Small backward error is the best we can hope in a finite precision environment.
- Showing small backward error implies stability: the algorithm doesn't magnify error. This is the polynomial example.
- But, stability doesn't imply small backward error: subtraction is an example.