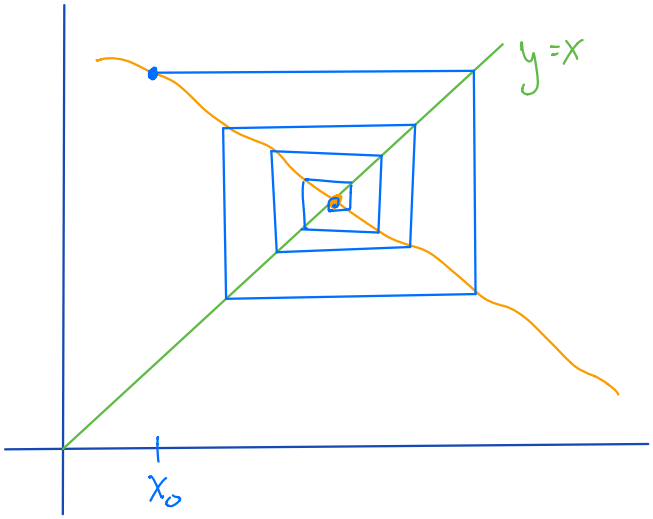
Another Example:

The above examples all had $g' > 0$.

What happens when $g' < 0$, but $|g'| < 1$?

We get oscillatory behavior:



We've seen the relationship between $g'(s)$ and the convergence/stability of $x_{k+1} = g(x_k)$. This can be made into the following theorem.

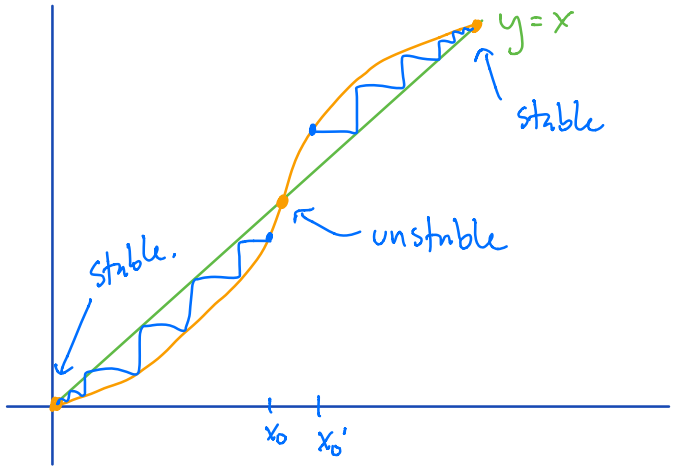
Theorem Suppose g is continuously differentiable in a neighborhood of its fixed point ξ, and let $|g'(\xi)| > 1$. Then, $x_{k+1} = g(x_k)$ does not converge to ξ for any $x_0 \neq \xi$.

Sketch of Proof: Let $I_\delta = [\xi - \delta, \xi + \delta]$ be the δ-neighborhood of ξ. Then by the MVT,

$$\begin{aligned} |x_{k+1} - \xi| &= |g(x_k) - g(\xi)| \\ &= |g'(\eta_k)(x_k - \xi)| \quad \text{for some } \eta_k \\ &\geq L |x_k - \xi| \quad \text{with } L > 1. \end{aligned}$$

So the sequence is diverging from ξ, and eventually will exit the interval I_δ.

Example: Some functions have both stable and unstable fixed points:



More technical matters: Floating-pt. arithmetic

What guarantees the "correct" solution of any numerical algorithm?

1. Bug-free code (obviously)
2. Convergent numerical algorithm (in infinite-precision)
3. "Stable" computations (recall $\cos(2\pi 10^{20})$?)

① & ② are "human" problems

③ is a "computer" problem

[Intel Flaw incorrectly computing quotients
=> $420 million dollars]

[Ariane 5 rocket - programmers didn't allocate enough memory
=> all arrays at least 10000 long!]

The question you should be asking yourself is:

How do computers store numbers?

→ Binary / base-2 arithmetic

Decimal: $10 = 1 \cdot 10 + 0 \cdot 1$

Binary: $1010 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$

Binary addition:

$23 = 16 + 7$

$= 16 + 4 + 2 + 1$

$= 2^4 + 2^2 + 2^1 + 2^0$

$= (10111)_2$

$17 = 16 + 1$

$= 2^4 + 2^0$

$= (10001)_2$

$17 + 23 = 40 = 32 + 8$

$= 2^5 + 2^3$

$= (101000)_2$

$$\begin{array}{r} 10111 \\ +\ 10001 \\ \hline \boxed{101000} \end{array}$$

Binary decimals:

$\frac{7}{2} = 3.5$ in decimal notation

In binary notation: $3 = (11)_2$

$.5 = (0.1)_2$

$\frac{1}{2}$

$.25 = \frac{1}{4} = (0.01)_2$

$\frac{1}{2^2}$

Each 0 or 1 is a "bit".

Fixed representation: XXXX.XXXX 8 bit fixed

Not how computers store numbers

Floating point representation (like scientific notation)

$$\pm m \times 2^E, \quad m \in [1,2)$$

Example: $23 = (10111)_2$

$= (1.0111 \times 2^4)$ Every multiplication by 2 moves the decimal over by one.

$\frac{1}{9} = 0.001 = 1.0 \times 2^{-3}$

How many bits does a computer use?

single precision "word": 32 bits

1 bit for sign

8 bits for exponent

23 bits for mantissa (significand)

Ex: $5.5 = 1.011 \times 2^2$

always 1 (can improve, except for 0) hidden-bit rep.

$= (0 \mid \underbrace{E = 00000010}_{\text{exponent}} \mid \underbrace{1011\ldots0}_{\text{Mantissa}})$

↑ sign

Non-repeating ~~decimals~~ binary are "floating-point numbers"

What is the precision of this?

Machine precision is the distance between 1 and the next fp number. In this case,

$$1 = (0 \mid E=0 \mid 1.0\underbrace{\ldots\ldots}_{23\text{ bits}}0)$$

$$1 + 2^{-23} = (0 \mid E=0 \mid 1.0\ldots01)$$

$$\hookrightarrow \approx 1.2\times10^{-7}$$

Usually calculations are done in double precision:

sign: 1 bit

exponent: 11

mantissa: 52

machine precision: $eps = 2^{-52} \approx 2.2\times10^{-16}$

How to calculate machine precision? Demo

Floating point representations were standardized by the IEEE in 1980s

↳ Inst. of Electrical & Electronics Engin.

Consistent Rules

- Representation (# bits)
- Correct / consistent rounding
- sensible treatment of exceptions: $\frac{1}{0}$

Hidden bit rep.: 0, NaN, $\pm\infty$ (over/underflow)

single precision: 32 bits ($\epsilon \sim 10^{-7}$)

double precision: 64 bits ($\epsilon \sim 10^{16}$)

extended precision: 80 bits ($\epsilon \sim 10^{-20}$) (no hidden bits)

Rounding

Example: Decimal notation: $\frac{1}{10} = 0.1$

Binary notation: $\frac{1}{10} = (1.100\overline{1100})_2 \times 2^{-4}$

How is this rounded?

$\frac{1}{3} - \frac{1}{4} = \frac{1}{12} * \frac{1}{16}$

$\frac{1}{12} - \frac{1}{16} = \frac{4}{48} - \frac{3}{48}$

$= \frac{1}{48} - \frac{1}{64}$

4 options: Up

Down

to 0

nearest.

$1/10 = 1.100\overline{1100} \times 2^{-4}$

$= 1.100\,1100\,1100\,1100 \cdots 1100\,1$ (down, toward 0)

$= 1.100\,1100\,1100\,1100 \cdots 1101\,0$ (up, (nearest) ← default)

Absolute rounding error = $|\text{round}(x) - x|$

Relative rounding error = $\dfrac{|\text{round}(x) - x|}{x}$

Most important rule: IEEE says $\text{Round}(a+b) = a \oplus b$

($\text{Round}(a+b)$: exact addition; $\oplus$: computer addition)

$= (a+b)(1+\delta)$

(δ: $\leq \epsilon$ machine precision)