Introduction to numerical computation

Baojian Zhou

DATA830001, Numerical Computation
School of Data Science, Fudan University

Sep. 4th, 2024

Instructor

Baojian Zhou

- **Office:** N106, Zibin Building
- **Web:** `https://baojian.github.io/`
- **Office Hours:** Fri. 9:00am-10:00am
- **Email:** bjzhou@fudan.edu.cn
- **TAs**: Will be updated!

What is numerical computation?

Numerical computation involves studying, developing, and analyzing algorithms to obtain numerical solutions to various mathematical problems.

- Study of algorithms
- Mathematical analysis
- Numerical approximation

What is numerical computation?

Numerical computation involves studying, developing, and analyzing algorithms to obtain numerical solutions to various mathematical problems.

- Study of algorithms
- Mathematical analysis
- Numerical approximation

Why the numerical computation? To "solve" many real-world problems, including root-finding, solving large-scale linear equations, generating real-world images/videos, analyzing deep neural networks, and many others.

Square root calculating

How to calculate $\sqrt{2}$ numerically?

Square root calculating

How to calculate $\sqrt{2}$ numerically?

$$\text{A modern way}: x_{t+1} = \frac{x_t}{2} + \frac{1}{x_t}$$

Babylonian method is about 3600 - 3800 years old (1800-1600 BC)

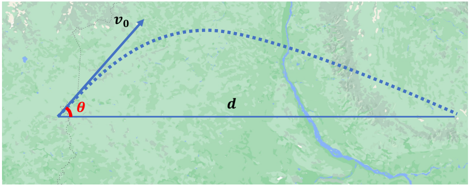
t	x_t with $x_0 = 100$	x_t with $x_0 = 2$	x_t with $x_0 = -100$
0	100.0	2.0	-100.0
1	50.0100000000	1.5000000000	-50.0100000000
2	25.0249960008	1.4166666667	-25.0249960008
3	12.5524580467	1.4142156863	-12.5524580467
4	6.3558946949	1.4142135624	-6.3558946949
5	3.3352816093	1.4142135624	-3.3352816093
6	1.9674655622	1.4142135624	-1.9674655622
7	1.4920008897	1.4142135624	-1.4920008897
8	1.4162413320	1.4142135624	-1.4162413320
9	1.4142150141	1.4142135624	-1.4142150141

Note: $\sqrt{2} \approx 1.4142135623730950488016887$.

- Why does (not) this algorithm work?
- How efficient is this method given fixed precision?

Root finding

An artillery officer wants to shell an enemy camp located d meters away from the position. Given that the shell leaves the cannon at an initial velocity v_0 m/s, disregarding air resistance, what should be the angle θ between the cannon and the horizontal line to hit the target? (Given gravitational acceleration $g = 9.8m/s^2$).

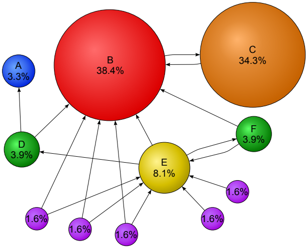


$$f(\theta) := \frac{2v_0^2 \sin\theta \cos\theta}{g} - d = 0.$$

Solving large-scale linear system

PageRank: An algorithm used by Google Search to rank web pages in their search engine results.

How do you rank web pages?



Node: Web page, Edge: Hyperlink

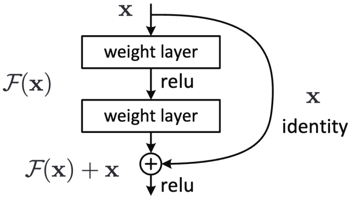
- Foundation of Google's success
- Analyzes web structure
- Determines importance

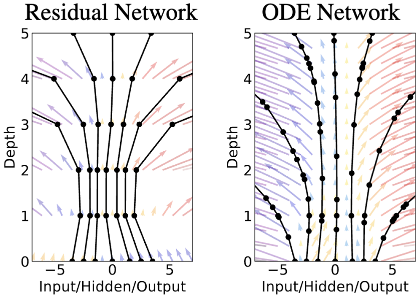
Let $\boldsymbol{\pi}$ be the vector of importance of all web pages, $\boldsymbol{D}$ be the outdegree diagonal matrix, and $\boldsymbol{A}$ be the adjacency matrix of the web graph. To calculate $\boldsymbol{\pi}$, we solve the following

$$\boldsymbol{\pi} = \left(\alpha \boldsymbol{A}^\top \boldsymbol{D}^{-1} + \frac{1-\alpha}{n} \boldsymbol{E} \right) \boldsymbol{\pi},$$

where $\boldsymbol{E}$ is all one matrix and α is the dumping factor.

Solving ordinary differential equation





ResNet:

$$\boldsymbol{x}_{t+1} = \text{ResBlock}\left(\boldsymbol{x}_t, \theta_t\right)$$

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + f\left(\boldsymbol{x}_t, \theta_t\right), t = 0, 1, \ldots, L$$

$$\boldsymbol{y}_{\text{pred}} = \text{ResNet}(\boldsymbol{x}), \quad L\left(\boldsymbol{y}_{\text{pred}}\right) \to \frac{\partial L}{\partial \theta}$$

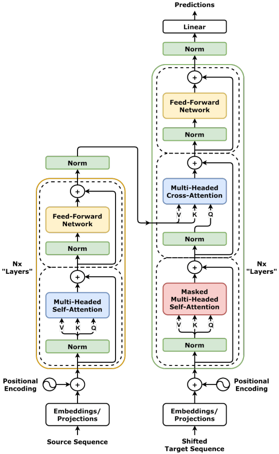
Add more layers and get

$$\frac{d\boldsymbol{x}(t)}{dt} = f(\boldsymbol{x}(t), t, \theta).$$

Euler discretization

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h \cdot f\left(\boldsymbol{x}_n, t_n, \theta\right)$$

Solving ordinary differential equation



A Transformer is a flow map on $(\mathbb{S}^{d-1})^n$: the input sequence $(x_i(0))_{i\in[n]} \in (\mathbb{S}^{d-1})^n$ is an initial condition which is evolved through the dynamics

$$\dot{x}_i(t) = \mathrm{P}^{\perp}_{x_i(t)}\left(\frac{1}{Z_{\beta,i}(t)}\sum_{j=1}^{n} e^{\beta\langle Q(t)x_i(t),K(t)x_j(t)\rangle} V(t)x_j(t)\right)$$

for all $i \in [n]$ and $t \geqslant 0$ where the function

$$\mathrm{P}^{\perp}_{x}(y) = y - \langle x, y\rangle x$$

denotes the projection of $y \in \mathbb{R}^d$ onto $\mathrm{T}_x(\mathbb{S}^{d-1})$. The partition function $Z_{\beta,i}(t) > 0$ reads

$$Z_{\beta,i}(t) = \sum_{k=1}^{n} e^{\beta\langle Q(t)x_i(t),K(t)x_k(t)\rangle}.$$

Solving stochastic differential equation

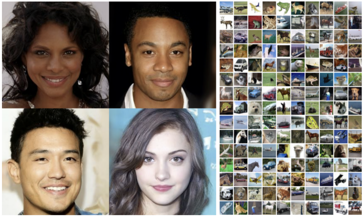


Figure 1: Generated samples on CelebA-HQ 256 × 256 (left) and unconditional CIFAR10 (right)

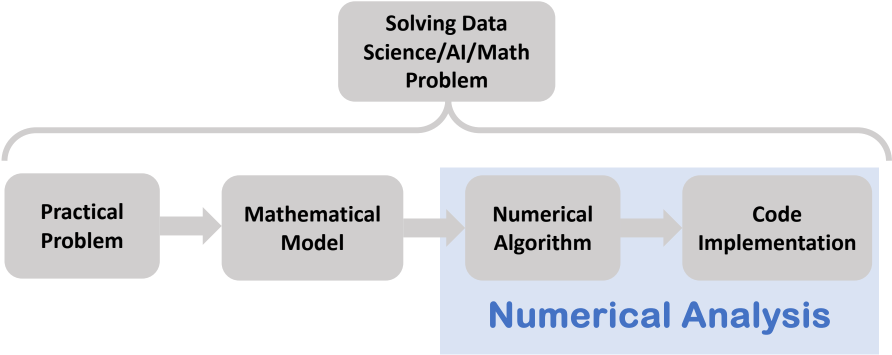
To draw the connection between Denoising Diffusion Probabilistic Models (DDPM) and SDE, we consider the discrete-time DDPM iteration. For $i = 1, 2, \dots, N$:

$$\boldsymbol{x}_i = \sqrt{1-\beta_i}\,\boldsymbol{x}_{i-1} + \sqrt{\beta_i}\,\boldsymbol{z}_{i-1},$$
$$\boldsymbol{z}_{i-1} \sim \mathcal{N}(0, \boldsymbol{I})$$

We can show that this equation can be derived from the forward SDE equation below. The forward sampling equation of DDPM can be written as an SDE via

$$d\boldsymbol{x} = \underbrace{-\frac{\beta(t)}{2}\boldsymbol{x}}_{=f(\boldsymbol{x},t)}\,dt + \underbrace{\sqrt{\beta(t)}}_{=g(t)}\,d\boldsymbol{w}.$$

A general paradigm



Course Topics

1. Fundamentals and computer arithmetic (This lecture)
2. Solving nonlinear equations
3. Solving linear equations ($\boldsymbol{Ax} = \boldsymbol{b}$)
4. Solving large-scale sparse systems
5. (Preconditioning) Conjugate Gradient Method (CGM)
6. Semi-iterative (SI) and Chebyshev method
7. Iterative methods on graphs and localization
8. Eigenvalues and eigenvectors of matrices
9. Interpolation and least squares
10. Numerical differentiation and integration
11. Solving ODE and boundary value problems
12. Randomization and SDE

Course Website and References

Fudan eLearning

- `https://elearning.fudan.edu.cn/`

Recommended books:

- Numerical Analysis (3rd edition), Timothy Sauer.
- Numerical Analysis: Mathematics of Scientific Computing, David Ronald, and Elliott Ward Cheney.
- Matrix Computation (4th), Gene H. Golub and Charles F. Van Loan.

Other references:

- Matrix Analysis, Roger Horn and Charles Johnson
- Numerical Methods, Design, Analysis, and Computer Implementation of Algorithms, Anne Greenbaum and Timothy P. Chartier

Grade & Programming languages

Grading Breakdown

- Homeworks: 45%
- Middle term exam (take home): 5-10%
- Final exam: 40-45%
- Sign-in: 5%

Programming Languages

- Python3+Scipy, Matlab, C/C++ (Recommended)
- R, Octave, Julia, Java, ... (Not Recommended)

For Matlab users

- `http://mvls.fudan.edu.cn/matlab/`

Evaluating a polynomial

What is the best way to evaluate the following polynomial (at $x = 1/2$)

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1.$$

Use as few additions and multiplications as possible.

Evaluating a polynomial

What is the best way to evaluate the following polynomial (at $x = 1/2$)

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1.$$

Use as few additions and multiplications as possible.

Method 1: a straightforward approach

$$P\left(\frac{1}{2}\right) = 2 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} + 3 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} - 3 * \frac{1}{2} * \frac{1}{2} + 5 * \frac{1}{2} - 1$$
$$= \frac{5}{4}.$$

Evaluating a polynomial

What is the best way to evaluate the following polynomial (at $x = 1/2$)

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1.$$

Use as few additions and multiplications as possible.
Method 1: a straightforward approach

$$\begin{aligned} P\left(\frac{1}{2}\right) &= 2 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} + 3 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} - 3 * \frac{1}{2} * \frac{1}{2} + 5 * \frac{1}{2} - 1 \\ &= \frac{5}{4}. \end{aligned}$$

- # of multiplications: 10
- # of additions: 4

Evaluating a polynomial

What is the best way to evaluate the following polynomial (at $x = 1/2$)

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1.$$

Use as few additions and multiplications as possible

Evaluating a polynomial

What is the best way to evaluate the following polynomial (at $x = 1/2$)

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1.$$

Use as few additions and multiplications as possible
Method 2: store some calculated numbers:

$$\frac{1}{2} * \frac{1}{2} = \left(\frac{1}{2}\right)^2, \quad \left(\frac{1}{2}\right)^2 * \frac{1}{2} = \left(\frac{1}{2}\right)^3, \quad \left(\frac{1}{2}\right)^3 * \frac{1}{2} = \left(\frac{1}{2}\right)^4$$

$$P\left(\frac{1}{2}\right) = 2 * \left(\frac{1}{2}\right)^4 + 3 * \left(\frac{1}{2}\right)^3 - 3 * \left(\frac{1}{2}\right)^2 + 5 * \frac{1}{2} - 1 = \frac{5}{4}$$

Evaluating a polynomial

What is the best way to evaluate the following polynomial (at $x = 1/2$)

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1.$$

Use as few additions and multiplications as possible

Method 2: store some calculated numbers:

$$\frac{1}{2} * \frac{1}{2} = \left(\frac{1}{2}\right)^2, \quad \left(\frac{1}{2}\right)^2 * \frac{1}{2} = \left(\frac{1}{2}\right)^3, \quad \left(\frac{1}{2}\right)^3 * \frac{1}{2} = \left(\frac{1}{2}\right)^4$$

$$P\left(\frac{1}{2}\right) = 2 * \left(\frac{1}{2}\right)^4 + 3 * \left(\frac{1}{2}\right)^3 - 3 * \left(\frac{1}{2}\right)^2 + 5 * \frac{1}{2} - 1 = \frac{5}{4}$$

- # of multiplications: 7
- # of additions: 4

Evaluating a polynomial

What is the best way to evaluate the following polynomial (at $x = 1/2$)

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1.$$

Use as few additions and multiplications as possible

Evaluating a polynomial

What is the best way to evaluate the following polynomial (at $x = 1/2$)

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1.$$

Use as few additions and multiplications as possible
Method 3: Nested multiplication

$$\begin{aligned} P(x) &= -1 + x\left(5 - 3x + 3x^2 + 2x^3\right) \\ &= -1 + x\left(5 + x\left(-3 + 3x + 2x^2\right)\right) \\ &= -1 + x * (5 + x * (-3 + x * (3 + 2 * x))). \end{aligned}$$

- # of multiplications: 4
- # of additions: 4

**Further explore the problem structure;
a better method may be possible.**

Evaluating a polynomial

Horner's method: For $P(x) = \sum_{i=0}^{k} c_i x^i$, rewrite this polynomial

- Rewrite $P(x)$ as: $P(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + \cdots + x(c_k))))$
- # Multiplications: k
- # Additions: k
- Or Rewrite $P(x)$ as:
  $P(x) = c_0 + (x - r_1)(c_1 + (x - r_2)(c_2 + (x - r_3)(c_3 + \cdots + (x - r_k)(c_k))))$
  with $r_1 = r_2 = \cdots = 0$.

Evaluating a polynomial

Horner's method: For $P(x) = \sum_{i=0}^{k} c_i x^i$, rewrite this polynomial

- Rewrite $P(x)$ as: $P(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + \cdots + x(c_k))))$
- # Multiplications: k
- # Additions: k
- Or Rewrite $P(x)$ as:
  $P(x) = c_0 + (x - r_1)(c_1 + (x - r_2)(c_2 + (x - r_3)(c_3 + \cdots + (x - r_k)(c_k))))$
  with $r_1 = r_2 = \cdots = 0$.

Example: Evaluating the polynomial $P(x) = 4x^5 + 7x^8 - 3x^{11} + 2x^{14}$.

Solution:

$$\begin{aligned} P(x) &= x^5(4 + 7x^3 - 3x^6 + 2x^9) \\ &= x^5 * (4 + x^3 * (7 + x^3 * (-3 + x^3 * (2)))). \end{aligned}$$

(7 *, 3 +)

Binary numbers

Binary numbers are expressed as

$$\dots b_2 b_1 b_0 . b_{-1} b_{-2} \dots,$$

where each $b_i \in \{0, 1\}$. To the base 10 equivalent number, we have

$$\dots b_2 2^2 + b_1 2^1 + b_0 2^0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} \dots$$

Representing numbers

- Binaries: $(100.0)_2$, $(1111.0)_2$, $(0.0)_2$
- Decimals: $(4.0)_{10}$, $(15.0)_{10}$, $(0.0)_{10}$

We have

$$(100.0)_2 = (4.0)_{10}$$
$$(1111.0)_2 = (15.0)_{10}, \dots$$

Binary numbers

Decimal to Binary: Given any decimal number $(x)_{10} = (y)_{10} + (z)_{10}$, where $(y)_{10}$ is the integer part and $(z)_{10}$ is the fractional part. For the integer part $(y)_{10}$, we have

$$(y)_{10} = \left\lfloor \frac{(y)_{10}}{2} \right\rfloor \cdot 2 + (y)_{10}\%2$$

Key idea: Start recording the calculated remainders from the decimal point and move sequentially from right to left. Example: $(53)_{10}$

$$\begin{aligned} 53/2 &= 26\ R\ 1 \\ 26/2 &= 13\ R\ 0 \\ 13/2 &= 6\ R\ 1 \\ 6/2 &= 3\ R\ 0 \\ 3/2 &= 1\ R\ 1 \\ 1/2 &= 0\ R\ 1. \end{aligned}$$

$$(53)_2 = 110101$$

Binary numbers

Decimal to Binary: Given any decimal number $(x)_{10} = (y)_{10} + (z)_{10}$, where $(y)_{10}$ is the integer part and $(z)_{10}$ is the fractional part. For fractional part $(z)_{10}$, we have

$$(z)_{10} \cdot 2 = \text{Integer part of } (z)_{10} \cdot 2 + \text{ fractional part of } (z)_{10} \cdot 2$$

Key idea: Start recording the calculated integers from the decimal point and move sequentially from left to right. Example: $(0.7)_{10}$

$$\begin{aligned} .7 \times 2 &= 1 + .4 \\ .4 \times 2 &= 0 + .8 \\ .8 \times 2 &= 1 + .6 \\ .6 \times 2 &= 1 + .2 \\ .2 \times 2 &= 0 + .4 \\ .4 \times 2 &= 0 + .8. \end{aligned}$$

$$(0.7)_2 = 0.1\overline{0110}$$

Binary numbers

Binary to decimal: For the integer part, simply add up powers of 2 as we did before. The binary number $(10101)_2$ is simply $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (21)_{10}$. Fractional part, if the fractional part is finite (a terminating base 2 expansion), proceed the same way. For example,

$$(.1011)_2 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = \left(\frac{11}{16}\right)_{10}.$$

What about $x = (0.\overline{1011})_2$?

Binary numbers

Binary to decimal: For the integer part, simply add up powers of 2 as we did before. The binary number $(10101)_2$ is simply $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (21)_{10}$. Fractional part, if the fractional part is finite (a terminating base 2 expansion), proceed the same way. For example,

$$(.1011)_2 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = \left(\frac{11}{16}\right)_{10}.$$

What about $x = (0.\overline{1011})_2$? Try to use the following trick

$$\begin{aligned} 2^4 x &= 1011.\overline{1011} \\ x &= 0000.\overline{1011} \end{aligned}$$

Subtracting yields $(15)_{10} x = (1011)_2 = (11)_{10}$ and $x = 11/15$.

Floating point representation

Many real-world numbers

- $\pi \approx 3.14159265358979323846264 3\ldots$
- $e \approx 2.71828182845904523536028 7\ldots$
- Planck constant: $h = 6.62607015 \times 10^{-34} J \cdot Hz^{-1}$
- Electron mass: $m_{\mathrm{e}} \approx 9.1093837015(28) \times 10^{-31} kg$
- Speed of light: $c = 2.99792458 \times 10^{8} m/s$
- Between 10^{78} to 10^{82} atoms in the observable universe

Floating point representation

Many real-world numbers

- $\pi \approx 3.14159265358979323846264 3\ldots$
- $e \approx 2.71828182845904523536028 7\ldots$
- Planck constant: $h = 6.62607015 \times 10^{-34} J \cdot Hz^{-1}$
- Electron mass: $m_{\mathrm{e}} \approx 9.1093837015(28) \times 10^{-31} kg$
- Speed of light: $c = 2.99792458 \times 10^{8} m/s$
- Between 10^{78} to 10^{82} atoms in the observable universe

Any given real number $(x)_{10}$ can be written in the form

$$\textbf{Scientific notations: } (x)_{10} = \pm m \times 10^{n}, \tag{1}$$

where n is the power and m is the mantissa.

How to save these scientific numbers into a computer?

Floating point representation - IEEE 754

Recognized as an American National Standard (ANSI) IEEE Std 754-1985

An American National Standard

IEEE Standard for Binary Floating-Point Arithmetic

Sponsor
Standards Committee of the IEEE Computer Society

Approved March 21, 1985
Reaffirmed December 6, 1990
IEEE Standards Board

Approved July 26, 1985
Reaffirmed May 21, 1991
American National Standards Institute

David Stevenson, *Chair*

Andrew Allison
William Ames
Mike Arya
Janis Baron
Steve Baumel
Dileep Bhandarkar
Joel Boney
E.H. Bristol
Werner Buchholz
Jim Bunch
Ed Burdick
Gary R. Burke
Paul Clemente
W.J. Cody
Jerome T. Coonen
Jim Crapuchettes
Itzhak Davidesko
Wayne Davison
R.H. Delp
James Demmel
Donn Denman
Alvin Despain
Augustin A. Dubrulle
Tom Eggers
Philip J. Faillace
Richard Fateman
David Feign
Don Feinberg
Smart Feldman
Eugene Fisher
Paul F. Flanagan
Gordon Force
Lloyd Fosdick
Robert Fraley
Howard Fullmer
Daniel D. Gajski
David M. Gay
C.W. Gear
Martin Graham
David Gustavson
Guy K. Haas
Kenton Hanson
Chuck Hastings
David Hough
John Edward Howe
Thomas E. Hull
Suren Irukulla
Richard E. James III
Paul S. Jensen
W. Kahan
Howard Kaikow
Richard Karpinski
Virginia Klema
Les Kohn
Dan Kuyper
M. Dundee Maples
Roy Martin
William H. McAllister
Colin McMaster
Dean Miller
Webb Miller
John C. Nash
Dan O'Dowd
Cash Olsen
A. Padegs
John F. Palmer
Beresford Parlett
Dave Patterson
Mary H. Payne
Tom Pittman
Lew Randall
Robert Reid
Christian Reinsch
Frederic N. Ris
Stan Schmidt
Van Shahan
Robert L. Smith
Roger Stafford
G.W. Stewart
Robert Stewart
Harold S. Stone
W.D. Strecker
Robert Swarz
George Taylor
James W. Thomas
Dar-Sun Tsien
Greg Walker
John Steven Walther
Shlomo Waser
P.C. Waterman
Charles White

IEEE 754-1985,2008,2019

Authors

Real Implementations: C/C++, Matlab, Fortran, Python, Julia, Java, ...

Adopted in almost all programming languages!

Floating point representation - IEEE 754

A floating point number consists of three parts: the sign (+or-), a mantissa, which contains the string of significant bits, and an exponent. The three parts are stored together in a single computer word.

precision	sign	exponent	mantissa	total bits
single	1	8	23	32
double	1	11	52	64
long double	1	15	64	80

IEEE standardized floating-point number

The form of a **normalized** IEEE floating point number is

$$\pm 1.bbb \ldots b \times 2^p,$$

where $b \in \{0, 1\}$, $p \in \mathbb{Z}$.

Floating point representation - Double precision

A binary number is stored as a normalized floating point number: for example, the decimal number 9, which is $(1001)_2$ in binary, would be stored as

$$+1.001 \times 2^3. \tag{2}$$

For double precision, exponent length $M = 11$ and mantissa length $N = 52$. Example: real number 1

$$+1.\boxed{0000000000000000000000000000000000000000000000000000} \times 2^0,$$

where boxed are 52 bits of the mantissa. The next floating point number greater than 1 is

$$+1.\boxed{0000000000000000000000000000000000000000000000000001} \times 2^0,$$

which equals to $1 + 2^{-52}$.

Machine epsilon ϵ_{mach}

Definition 2.1 (Machine epsilon)

The number machine epsilon, denoted ϵ_{mach}, is the distance between 1 and the smallest floating point number greater than 1. For the IEEE double precision floating point standard,

$$\epsilon_{mach} = 2^{-52} \tag{3}$$

The decimal number $9.4 = (1001.\overline{0110})_2$ is left-justified as

$$+1.\boxed{0010110011001100110011001100110011001100110011001100}110\ldots \times 2^3,$$

where we have boxed the first 52 bits of the mantissa.

Question: How do we deal with these remaining infinite binary numbers?

Truncation/Rounding

Chopping

- It throws away the bits that fall off the end.
- It is biased (Why ?)

Rounding (IEEE Rounding to Nearest Rule):

- if bit 53 is 1, then add 1 to bit 52 (round up)
- if bit 53 is 0, then add 0 to bit 52 (round down)
- **Exception**: if the bits following bit 52 are $10000\ldots$ (that is the value 2^{-53}), exactly halfway between up and down, to avoid bias, **round up or round down according to which choice makes the final bit 52 equal to 0.**

There are two equally distant floating point numbers to round to, should be decided in a way that doesn't prefer up or down systematically. This is to try to avoid the possibility of an unwanted slow drift in long calculations due simply to a biased rounding.

IEEE Rounding to Nearest Rule

Given number x, we denote the number of IEEE Rounding to Nearest Rule by $fl(x)$. There are two steps from x to $\color{red}{fl}(x)$: Example, to find $fl(1/6)$, note that $1/6 = 0.0\overline{01} = 0.001010101\ldots$ in binary.

- Justify

$$+1.\boxed{0101010101010101010101010101010101010101010101010101}\color{red}{0}\color{black}{101\ldots \times 2^{-3}}$$

- Round

$$fl(1/6) = +1.\boxed{010101010101010101010101010101010101010101010101010\color{blue}{1}} \times 2^{-3}$$

Example: To find $fl(11.3)$, note that 11.3 is equal to $1011.0\overline{1001}$ in binary.

- Justify

$$+1.\boxed{0110100110011001100110011001100110011001100110011001}\color{red}{1}\color{black}{001\ldots \times 2^{3}}$$

- Round

$$fl(11.3) = +1.\boxed{01101001100110011001100110011001100110011001100110\color{red}{10}} \times 2^{3}$$

Rounding error

Example: $9.4 = (1001.\overline{0110})_2$

$9.4 = +1.\boxed{0010110011001100110011001100110011001100110011001100}\,110\ldots \times 2^3$

where we have boxed the first 52 bits of the mantissa.

$fl(9.4) = +1.\boxed{0010110011001100110011001100110011001100110011001101} \times 2^3,$

To measure the rounding error,

$$\text{the discarded} : .\overline{1100} \times 2^{-52} \times 2^3 = .\overline{0110} \times 2^{-51} \times 2^3 = .4 \times 2^{-48}$$

$$\text{rounded into} : 2^{-52} \times 2^3 = 2^{-49}.$$

We have

$$\begin{aligned} fl(9.4) &= 9.4 + 2^{-49} - 0.4 \times 2^{-48} \\ &= 9.4 + (1 - 0.8)2^{-49} \\ &= 9.4 + 0.2 \times 2^{-49}, \end{aligned}$$

where we call 0.2×2^{-49} the **rounding error**.

How to measure the error?

- x the quantity we want to store/compute
- x_c the quantity we stored and computed

To measure the error, we can check

- **absolute error** $|x_c - x|$
- **relative error** $\frac{|x_c - x|}{|x|}$ when $x \neq 0$

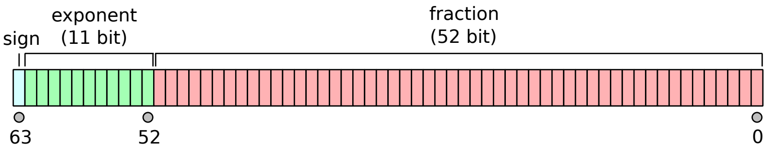
Theorem 2.2 (Relative error)

In the IEEE machine arithmetic model, the relative rounding error of $fl(x)$ is no more than one-half machine epsilon

$$\frac{|fl(x) - x|}{|x|} \leq \frac{1}{2}\epsilon_{mach}.$$

Machine representation

How to represent a double precision floating point number (x)?



Each word has the form

$$s e_1 e_2 e_3 e_4 \ldots e_{11} b_1 b_2 b_3 b_4 \ldots b_{52} \tag{4}$$

- $s = 0$ for positive number, $s = 1$ for negative number.
- exponent $e_1 e_2 e_3 e_4 \ldots e_{11}$
  - 00000000000: 0
  - 00000000001 — 11111111110: 1 — 2046. For each m, we add $2^{10} - 1 = 1023$. So, exponents will be in range $[-1022, 1023]$.
  - 11111111111: 2047

Machine representation

For example, the number 1, or

$$1 = +1.\boxed{0000000000000000000000000000000000000000000000000000} \times 2^0,$$

has double precision machine number form

$$\boxed{0}\boxed{01111111111}\boxed{0000000000000000000000000000000000000000000000000000}.$$

The special exponent value 2047:

- 2047: used to represent ∞ if the mantissa bit string is all zeros and NaN (not a number), otherwise. So, first 12 bits of Inf is $\boxed{0111}\boxed{1111}\boxed{1111}$ and -Inf is $\boxed{1111}\boxed{1111}\boxed{1111}$, the rest 52 bits are all zero.
- The machine number NaN also begins $\boxed{1111}\boxed{1111}\boxed{1111}$ but has a nonzero mantissa.

Machine representation

machine number	Example	Hex format
+Inf	1/0	7FF0000000000000
-Inf	-1/0	FFF0000000000000
NaN	0/0	FFFxxxxxxxxxxxxx

The special exponent value 0: $e_1 e_2 \ldots e_{11} = (00000000000)_2$, to present non-normalized floating point number.

$$\pm 0.\boxed{b_1 b_2 \ldots b_{52}} \times 2^{-1022} \tag{5}$$

We can these as subnormal floating point numbers
Question: **smallest representable positive number**

Machine representation

machine number	Example	Hex format
+Inf	1/0	7FF0000000000000
-Inf	-1/0	FFF0000000000000
NaN	0/0	FFFxxxxxxxxxxxxx

The special exponent value 0: $e_1 e_2 \dots e_{11} = (00000000000)_2$, to present non-normalized floating point number.

$$\pm 0.\boxed{b_1 b_2 \dots b_{52}} \times 2^{-1022} \tag{5}$$

We can these as subnormal floating point numbers

Question: **smallest representable positive number**

$$2^{-52} \times 2^{-1022} = 2^{-1074}.$$

0	00000000000	0000000000000000000000000000000000000000000000000001

.

Machine representation

Subnormal numbers include the most important number 0. Two 0s (They are treated as the same number):

- +0: $(0000000000000000)_{16}$
- -0: $(8000000000000000)_{16}$

What about numbers beyond?

- **overflow**: too large to be stored as a regular floating point number. For double-precision floating point numbers, this means the exponent is greater than 1023. Most computer languages will convert an overflow condition to machine number +Inf, -Inf, or NaN.
- **underflow**: double precision, this occurs for numbers less than 2^{-1074}.

Loss of significant digits

Suppose we have two seven-significant digits; we need to subtract them:

$$123.4567 - 123.4566 = 000.0001.$$

The result has only one-digit accuracy.
Example: $\sqrt{9.01} - 3 \approx 3.0016662 - 3 = 0.0016662$, if we save the result on a 3-decimal-digit computer, then the result will be 0. Can we fix it?

Loss of significant digits

Suppose we have two seven-significant digits; we need to subtract them:

$$123.4567 - 123.4566 = 000.0001.$$

The result has only one-digit accuracy.
Example: $\sqrt{9.01} - 3 \approx 3.0016662 - 3 = 0.0016662$, if we save the result on a 3-decimal-digit computer, then the result will be 0. Can we fix it? Avoid this issue by rewriting the expression:

$$\sqrt{9.01} - 3 = \frac{(\sqrt{9.01} - 3)(\sqrt{9.01} + 3)}{\sqrt{9.01} + 3}$$

Loss of significant digits

Example : $E_1 = \sqrt{x^2+1} - 1, \quad E_2 = \dfrac{x^2}{\sqrt{x^2+1}+1}$

Loss of significant digits

Example : $E_1 = \sqrt{x^2+1} - 1, \quad E_2 = \dfrac{x^2}{\sqrt{x^2+1}+1}$

x	E_1	E_2
1.000000000000000e+00	4.142135623730951e-01	4.142135623730951e-01
1.000000000000000e-01	4.987562112088950e-03	4.987562112089027e-03
1.000000000000000e-02	4.999875006239662e-05	4.999875006249610e-05
1.000000000000000e-03	4.999998750587764e-07	4.999998750000624e-07
1.000000000000000e-04	4.999999969612645e-09	4.999999987500000e-09
1.000000000000000e-05	5.000000413701855e-11	4.999999999875001e-11
1.000000000000000e-06	5.000444502911705e-13	4.999999999998750e-13
1.000000000000000e-07	4.884981308350689e-15	4.999999999999987e-15
1.000000000000000e-08	0.000000000000000e+00	5.000000000000001e-17
1.000000000000000e-09	0.000000000000000e+00	5.000000000000000e-19
1.000000000000000e-10	0.000000000000000e+00	5.000000000000000e-21
9.999999999999999e-12	0.000000000000000e+00	5.000000000000000e-23
1.000000000000000e-12	0.000000000000000e+00	5.000000000000000e-25
1.000000000000000e-13	0.000000000000000e+00	5.000000000000000e-27
1.000000000000000e-14	0.000000000000000e+00	5.000000000000000e-29

Loss of significant digits

$$\textbf{Example}: E_1 = \frac{1-\cos x}{\sin^2 x}, \quad E_2 = \frac{1}{1+\cos x}.$$

Notice that $E_1 = E_2$. But, evaluate them at points near $x = 0$, we have

x	E_1	E_2
1.000000000000000	0.649223205204762	0.649223205204762
0.100000000000000	0.501252086288566	0.501252086288571
0.010000000000000	0.500012500208481	0.500012500208336
0.001000000000000	0.500000124992189	0.500000125000021
0.000100000000000	0.499999998627931	0.500000001250000
0.000010000000000	0.500000041386852	0.500000000012500
0.000001000000000	0.500044450291337	0.500000000000125
0.000000100000000	0.499600361081322	0.500000000000001
0.000000010000000	0.000000000000000	0.500000000000000
0.000000001000000	0.000000000000000	0.500000000000000
0.000000000100000	0.000000000000000	0.500000000000000
0.000000000010000	0.000000000000000	0.500000000000000
0.000000000001000	0.000000000000000	0.500000000000000
0.000000000000100	0.000000000000000	0.500000000000000
0.000000000000010	0.000000000000000	0.500000000000000

Loss of significant digits

Example: Find roots of $x^2 + 9^{12}x = 3$.

Consider two roots

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

It gives

$$x = \frac{-9^{12} \pm \sqrt{9^{24} + 4(3)}}{2}$$

and

$$x_1 = -2.8424 \times 10^{11}, x_2 = \frac{-9^{12} + \sqrt{9^{24} + 4(3)}}{2}.$$

MATLAB calculates $x_2 = 0$.

Exp-normalize trick

Consider the sigmoid function and its derivative

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \sigma'(x) = \sigma(x)(1-\sigma(x))$$

The sigmoid and its derivative are often used in logistic regression for binary classification.

Exp-normalize trick

Consider the sigmoid function and its derivative

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \sigma'(x) = \sigma(x)(1-\sigma(x))$$

The sigmoid and its derivative are often used in logistic regression for binary classification. It is better to consider the following

- if $x > 0$, then calculate $\sigma(x)$ as $\sigma(x) = \frac{1}{1+e^{-x}}$
- if $x \leq 0$, then calculate $\sigma(x)$ as $\sigma(x) = \frac{e^x}{1+e^x}$

Exp-normalize trick

Suppose you want to evaluate a probability distribution $\boldsymbol{\pi}$ parametrized by a vector $\boldsymbol{x} \in \mathbb{R}^n$ as the follows:

$$\pi_i = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}. \tag{6}$$

When $\boldsymbol{x} = [1, -5, 1000]$, it will overflow.

Exp-normalize trick

Suppose you want to evaluate a probability distribution $\boldsymbol{\pi}$ parametrized by a vector $\boldsymbol{x} \in \mathbb{R}^n$ as the follows:

$$\pi_i = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}. \tag{6}$$

When $\boldsymbol{x} = [1, -5, 1000]$, it will overflow. But, we can reformulate it as

$$\pi_i = \frac{exp(x_i - b)exp(b)}{\sum_{j=1}^{n} exp(x_j - b)exp(b)} = \frac{exp(x_i - b)}{\sum_{j=1}^{n} exp(x_j - b)}, \tag{7}$$

where $b = \max\{x_i | i = 1, 2, \ldots, n\}$.

The Log-Sum-Exp Trick

We still assume that $\pi_i = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}$. In many machine learning problems, we want to calculate log-distribution

$$\log \pi_i = \log \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)} \tag{8}$$

How to avoid overflow?

The Log-Sum-Exp Trick

We still assume that $\pi_i = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}$. In many machine learning problems, we want to calculate log-distribution

$$\log \pi_i = \log \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)} \tag{8}$$

How to avoid overflow? Notice that

$$\log \pi_i = \log \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)} = x_i - \text{logsumexp}(\boldsymbol{x}), \tag{9}$$

where $\text{logsumexp}(\boldsymbol{x}) = b + \log \sum_{j=1}^{n} exp(x_j - b)$. Typically, $b = \max\{x_i | i = 1, 2, \ldots, n\}$. Check more in PyTorch:
`https://pytorch.org/docs/stable/generated/torch.logsumexp.html`

Quick Summary

1. Try to avoid the amplification and propagation of rounding errors.
2. Try to avoid subtracting two nearly equal numbers.
3. Try to avoid large numbers "swallowing" small numbers.
4. Try to avoid having a divisor with a very small absolute value.

Solving Equations

Definition 3.1 (Root and problem definition)

Given a function $f : \mathbb{R} \to \mathbb{R}$, we say that $f(x)$ has a **root** at $x = r$ if $f(r) = 0$.

- How do we know a root exists?
- If the root exists, how can we find it?

Solving Equations

Definition 3.1 (Root and problem definition)

Given a function $f : \mathbb{R} \to \mathbb{R}$, we say that $f(x)$ has a **root** at $x = r$ if $f(r) = 0$.

- How do we know a root exists?
- If the root exists, how can we find it?

To check the existence of the root:

Theorem 3.2

Let f be continuous on $[a, b]$, satisfying $f(a)f(b) < 0$. Then f has a root in $[a, b]$, that is, there exists a number $r \in [a, b]$ and $f(r) = 0$.

Let $f(x) := e^x - \sin x - 2$. Then,
$f(0) = -1 < 0, f(\pi) = e^{\pi} - 2 > 0$. It has a root in $[0, \pi]$.

Bisection Method

Assume f has a root $r \in [a, b]$, how to find r?

- **Naive method**: scan all values with d precision from a to b. But the time complexity will be $\mathcal{O}\left((b-a)10^d\right)$.

Bisection Method

Assume f has a root $r \in [a, b]$, how to find r?

- **Naive method**: scan all values with d precision from a to b. But the time complexity will be $\mathcal{O}\left((b-a)10^d\right)$.

Intuition: Find a way to "squash" the interval $[a, b]$, so that location of r can be narrowed down.

Idea of the bisection: Find the middle point $c = (a+b)/2$ if $f(a)f(c) < 0$, then narrow down the interval $[a, b]$ into $[a, c]$ and let $b = c$; if $f(c) = 0$, return $r = c$; if $f(b)f(c) < 0$, then narrow down the interval $[a, b]$ into $[b, c]$ and let $a = c$. Repeat this step until $(b-a)/2$ is small enough.

Bisection Method

Algorithm 1 Bisection(f, a, b, ϵ)

Input: $[a, b]$ and f are such that $f(a)f(b) < 0$, tolerance ϵ
Output: an (approximate) root of f
while $(b - a)/2 > \epsilon$ **do**
    $c = (a + b)/2$
    if $f(c) = 0$ **then**
        return c
    end if
    if $f(a)f(c) < 0$ **then**
        $b = c$
    else
        $a = c$
    end if
end while
Return $(a + b)/2$

Accuracy and time complexity analysis

Accuracy: After n iterations, we have $c_n = (a_n + b_n)/2$. We measure the accuracy of the solution by the solution error, $|r - c_n|$. We have

$$|r - c_n| < \frac{b - a}{2^{n+1}} \tag{10}$$

Proof.

At the beginning ($n = 0$), the distance between c_n and r must be less than $(b - a)/2$. After each iteration, the interval is narrowed down by the half of $(b - a)$. Hence, after n iterations, $|r - c_n|$ must be less than $(b - a)/2^{n+1}$. ∎

Time complexity: The time complexity depends on how many function evaluations needed. The number of function evaluations after n iterations of Bisection is $n + 2$. Hence, $\mathcal{O}(n + 2)$.

Accuracy and iterations

Definition 3.3 (p correct places (p))

A solution is correct within p decimal places if the error is less than 0.5×10^{-p}.

Example 3.4

Use the Bisection method to find a root of $f(x) = \cos x - x$ in $[0, 1]$ to within 6 correct places. How many steps will be needed?

Accuracy and iterations

Definition 3.3 (p correct places (p))

A solution is correct within p decimal places if the error is less than 0.5×10^{-p}.

Example 3.4

Use the Bisection method to find a root of $f(x) = \cos x - x$ in $[0, 1]$ to within 6 correct places. How many steps will be needed?

Solution: $n = 20$.

Fixed-Point Iteration

Using a calculator (in radian mode), if you keep pressing the cos key, you'll find that no matter which number you start with, it will eventually converge to: 0.7390851332. It actually solves $\cos x - x = 0$.

Fixed Point: The number $r \in \mathbb{R}$ is a fixed point of g if $g(r) = r$.

Algorithm 2 FPI(g, x_0)

1: **for** $i = 0, 1, 2, \ldots,$ **do**
2: $\quad x_{i+1} = g(x_i)$
3: **end for**

Theorem 3.5 (The convergences of FPI)

If g is continuous and x_i converges to r, then r is a fixed point.

To prove, note that

$$g(r) = g\left(\lim_{i\to\infty} x_i\right) = \lim_{i\to\infty} g(x_i) = \lim_{i\to\infty} x_{i+1} = r.$$

Fixed-Point Iteration - Example

FPI solves the fixed point problem $g(x) = x$. Can every equation $f(x) = 0$ be turned into a fixed-point problem $g(x) = x$?

Fixed-Point Iteration - Example

FPI solves the fixed point problem $g(x) = x$. Can every equation $f(x) = 0$ be turned into a fixed-point problem $g(x) = x$? Yes, just let $g(x) = f(x) + x$! But, if we know the analytic form of f, we can have different fixed-point reformulations. For example, $f(x) = x^3 + x - 1$, then we have the following possibilities

1. $x = 1 - x^3$, then let $g_1(x) = 1 - x^3$
2. $x = \sqrt[3]{1 - x}$, then let $g_2(x) = \sqrt[3]{1 - x}$
3. add $2x^3$ on both sides, we have $3x^3 + x = 1 + 2x^3$, that is, $x = \frac{1+2x^3}{1+3x^2}$; then let $g_3(x) = \frac{1+2x^3}{1+3x^2}$.

Fixed-Point Iteration (FPI)

Let $f(x) = x^3 + x - 1 = 0$, we can have the following 3 different forms of g

$$g_1(x) := 1 - x^3, \quad g_2(x) := \sqrt[3]{1-x}, \quad g_3(x) = \frac{1+2x^3}{1+3x^2}$$

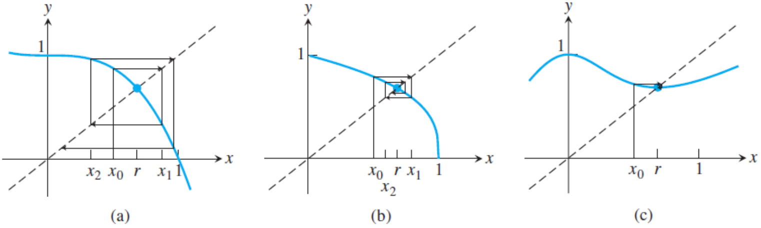
t	$x_t = g_1(x_{t-1})$	$x_t = g_2(x_{t-1})$	$x_t = g_3(x_{t-1})$
0	0.50000000	0.50000000	0.50000000
1	0.87500000	0.79370053	0.71428571
2	0.33007813	0.59088011	0.68317972
3	0.96403747	0.74236393	0.68232842
4	0.10405419	0.63631020	0.68232780
5	0.99887338	0.71380081	0.68232780
6	0.00337606	0.65900615	0.68232780
7	0.99999996	0.69863261	0.68232780
8	0.00000012	0.67044850	-
9	1.00000000	0.69072912	-
10	0.00000000	0.67625892	-
11	1.00000000	0.68664554	-
12	0.00000000	0.67922234	-
13	-	0.68454401	-

All three iteration procedure starts from $x_0 = 0.5$. Some observations:

- $x_{t+1} = g_1(x_t)$ cannot converge properly.
- $x_{t+1} = g_2(x_t)$ converges but relatively slow.
- $x_{t+1} = g_3(x_t)$ converges very fast.

Fixed-Point Iteration - Example

$f(x) = x^3 + x - 1$. Iterations of three different methods.



Convergence

Linear Convergence: Let $e_t = |r - x_t|$ denote the error at step t of an iterative method. If

$$\lim_{t\to\infty} \frac{e_{t+1}}{e_t} = S < 1, \tag{11}$$

the method is said to obey *linear convergence* with rate S.

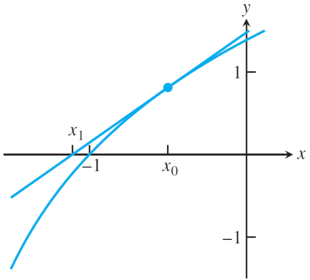
Locally convergent: An iterative method is called locally convergent to r if the method converges to r for initial guesses sufficiently close to r.

Theorem 3.6 (Linear convergence of FPI)

Assume that g is continuously differentiable, that $g(r) = r$, and that $S = |g'(r)| < 1$. Then Fixed-Point Iteration converges linearly with rate S to the fixed point r for initial guesses sufficiently close to r.

Newton's method

Key Idea: If f is differentiable, we draw the tangent line at x_t and use the intersection of this tangent with the x-axis as an approximate.



One point on the tangent line is $(x_0, f(x_0))$. The point-slope formula for the equation of a line is $y - f(x_0) = f'(x_0)(x - x_0)$. The intersection point can be found by letting $y = 0$. That is, $y - f(x_0) = f'(x_0)(x - x_0)$

$$x - x_0 = -\frac{f(x_0)}{f'(x_0)}, \rightarrow x = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Algorithm 3 Newton(f, x_0)

x_0 = initial guesses
for $t = 0, 1, 2, \ldots,$ **do**
    $x_{t+1} = x_t - f(x_t)/f'(x_t)$
end for
Return x_{t+1}

Newton's method

Algorithm 4 Newton(f, x_0)

1: x_0 = initial guesses
2: **for** $t = 0, 1, 2, \ldots,$ **do**
3: $\quad x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$
4: **end for**
5: **Return** x_{t+1}

Consider $f(x) = x^3 + x - 1$ and use the Newton's method to find a root of $f(x) = 0$. The iteration table of the Newton's method shows as the following:

t	x_t	$e_t = \lvert x_t - x^*\rvert$	e_t/e_{t-1}^2
0	-0.69999999999999995559107901499374	1.38232780e+00	-
1	0.12712550607287453896532269936870	5.55202298e-01	0.290556
2	0.95767811917566125767820039982325	2.75350315e-01	0.893271
3	0.73482779499450145976879866793752	5.24999912e-02	0.692449
4	0.68459177068492671480726130539551	2.26396686e-03	0.821394
5	0.68233217420448422085854645047220	4.37037646e-06	0.852666
6	0.68232780384433244780240102045354	1.63131730e-11	0.854084
7	0.68232780382801927476776882031118	0.00000000e+00	-
8	0.68232780382801927476776882031118	-	-

Newton's method - Convergence

Definition 3.7 (Quadratically convergent)

Let $e_t = |x_t - x^*|$ denote the error after step t of an iterative method. The iteration is quadratically convergent if

$$M = \lim_{t\to\infty} \frac{e_{t+1}}{e_t^2} < \infty. \tag{12}$$

Theorem 3.8 (Quadratically convergent of the Newton's)

Let f be twice continuously differentiable and $f(x^*) = 0$. If $f'(x^*) \neq 0$, then Newton's Method is locally and quadratically convergent to x^*. The error e_t at step t satisfies

$$\lim_{t\to\infty} \frac{e_{t+1}}{e_t^2} = M, \text{ where } M = \frac{f''(x^*)}{2f'(x^*)}. \tag{13}$$

Secant method

The Newton's method converges very fast. But, it needs to have derivative information, which may not be available. Can we do any approximation based the Newton's method?

Secant method

The Newton's method converges very fast. But, it needs to have derivative information, which may not be available. Can we do any approximation based the Newton's method?

Key Idea: Approximate the derivative by constructing a secant line!

$$\text{An approximation of } f'(x_t) \text{ at} x_t : f'(x_t) \approx \frac{f(x_t) - f(x_{t-1})}{x_t - x_{t-1}}. \tag{14}$$

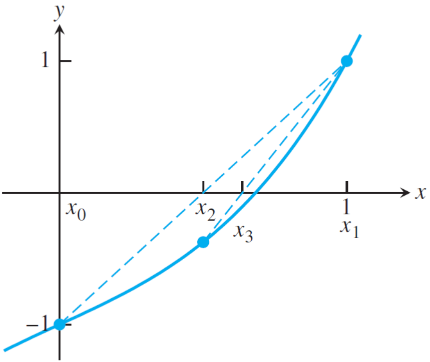
Algorithm 6 Secant(f, x_0, x_1)

1: x_0, x_1 be initial guesses
2: **for** $t = 1, 2, \ldots,$ **do**
3: $\quad x_{t+1} = x_t - \frac{f(x_t)(x_t - x_{t-1})}{f(x_t) - f(x_{t-1})}$
4: **end for**
5: **Return** x_{t+1}

Example

Consider $f(x) = x^3 + x - 1$ and use the Secant method to find a root of $f(x) = 0$. Let $x_0 = 0, x_1 = 1$ and check $f(x_0)f(x_1) = -1 < 0$.
The iteration table of the Secant method shows as the following:

t	x_t	$\lvert x_t - x^*\rvert$
0	0.00000000000000	6.8232e-01
1	1.00000000000000	3.1767e-01
2	0.50000000000000	1.8232e-01
3	0.63636363636364	4.5964e-02
4	0.69005235602094	7.7245e-03
5	0.68202041964819	3.0738e-04
6	0.68232578140989	2.0224e-06
7	0.68232780435903	5.3100e-10
8	0.68232780382802	8.8817e-16
9	0.68232780382802	0.0000e+00



Summary of secant method

Advantages:

1. Under some conditions, it converges faster than a linear rate.
2. It does not require the derivative information.
3. Compared with Newton's method, it requires only one function evaluation per iteration.

Disadvantages:

1. It may not converge.
2. There is no guaranteed error bound for the computed iterates.
3. It is likely to have difficulty if $f'(x^*) = 0$. This means the x-axis is tangent to the graph of $y = f(x)$ at $x = x^*$.

Brent's method

Can we take advantage of the above methods?

Brent's method

Can we take advantage of the above methods?
Richard Brent devised a method combining the advantages of the bisection and secant methods.

- It is guaranteed to converge.
- It has an error bound, which will converge to zero in practice.
- For most problems $f(x) = 0$, with $f(x)$ differentiable about the root x^*, the method behaves like the secant method.
- In the worst case, it is not too much worse in its convergence than the bisection method.

Practical Implementations

Implementations

- Matlab: fzero
  `https://www.mathworks.com/help/matlab/ref/fzero.html`
- Python:
  - scipy.optimize.brenth: Find a root of a function in a bracketing interval using Brent's method with hyperbolic extrapolation.
  - scipy.optimize.bisect: Find root of a function within an interval using bisection.
  - scipy.optimize.ridder: Find a root of a function in an interval using Ridder's method.
  - scipy.optimize.brentq: Find a root of a function in a bracketing interval using Brent's method.