COMP 350 MIDTERM 1 - Margaret Gu

1.A IEEE OVERVIEW NOTES

2 - REAL NUMBERS

Real Numbers = rational + irrational

Rational Numbers = integers + non-integral fractions

- **Integers**: infinite & countable
- **Non-integral fractions**: ratio of 2 integers

Irrational Numbers = sqrt(2), pi, e >>> Defined as a limit of sequences of rational numbers; uncountable

Ways to Represent Real Numbers

- Decimal and binary representation [*Know how to convert to either and decimal*]
- **Non-integral fractions representation**
  - Some have finite binary representation
  - Some have repeating infinite binary representation
    - Ex. $1/10 = (0.0001100110011...)_2$
- **Irrational numbers representation**
  - infinite & non-repeating for both binary and decimal representations

3 - COMPUTER REPRESENTATION OF NUMBERS

Easiest Way: in binary in 32 bits for only non-negative integers

- 2^{32} is too big to represent in 32 bits: 1 & 32 zeros = 33 bits
- **Biggest 32 bit #** = 2^{32} -1 (32 ones)

Negative Integers Computer Representation

- **Sign-and-modulus**: an extra bit to represent the sign
  - **Biggest #**: 2^{32} -1 >>> since one bit is used not to represent the sign
- **2's complement**
  - **Non-negative integers:** $[0, 2^{31} - 1]$
  - **Negative integers**: $[2^{31}, 2^{32} - 1]$
    - e.g. - x , where $x = [0, 2^{31} - 1]$
      - So (- x) is stored as a positive integer: $2^{32} - x$
    - Basically to verify, bitstrings of x + (-x) = 2^{32}
      - Since $2^{32} - x = (2^{32} - 1 - x) + 1$, and
      - $2^{32} - 1$ is all 1s
      - So change all 1's to 0's and vice versa, and add 1
  - Pros: no special hardware needed for integer subtraction
- **1's complement** = - x is stored as $2^{32} - x - 1$

Non-integral Computer Representation (using binary)

- **Fixed Point Representation**
  - Divided into 3 fields: sign(1) + number before binary decimal point (15) + binary decimal or fraction(16)
  - CONS: limits size of numbers being stored >> only $[2^{-16}, 2^{15})$
- **Floating Point Representation/exponential/scientific notation**
  - **To "normalize" = to use floating point representation**
  - **Nonzero Real Numbers (again in binary notation always)**
    - $x = \mp m \times 2^E$, where $1 <= m < 2$
      - m = significand, E = exponent
      - $m = (b_0b_1b_2...)_2$ with $b_0 = 1$
    - Sign (1) + exponent (8) + significand (23)
    - We can represent exponents from [-128, 127]
      - Because 8 bits in binary converts to 128 in decimal & 2's complement
      - E.g. we can store big numbers like 2^{71}
        - $(1.00)_2 \times 2^{71}$, where we just store 71 in exponent

precision = hidden bits + E bits.

- **Infinite binary representation**: e.g. 1/10 = $(1.100110011...)_2 \times 2^{-4}$
- **Number Zero** - cant be normalized; everything is just zero
- **The Gap** = number 1 and the next largest floating point number (**precision**/machine epsilon
    - $B_0b_1b_2...b_{22}$ = 1.000….0001, $b_{22} = 1$
    - Precision $\epsilon = 2^{-22}$
        - I.e. number of significand bits

$\varepsilon = 2^{-(m \text{ bits})}$

    - *Gap gets bigger as numbers get bigger*
    - **<u>To find gap</u>**: find largest number bigger than 1 in binary and convert into decimal. The gap is the difference
    - **The next bigger floating point number is:** $\epsilon \times 2^E$
    - **Gap between 0 & smallest positive number**: MUCH BIGGER than the other gaps
        - [LATER ON] this gap is filled by **<u>subnormal numbers</u>**

4 - IEEE FLOATING POINT REPRESENTATION

Hidden Bit Normalization: floating point representation but b_0 bit is 1 and is not stored

- So need a special technique to represent ZERO, since its leading bit is not 0
- So in the significand, if you see all zeros...does not mean zero (0.00) but 1.0000…
- Significand field is called **<u>fraction field</u>** now bc we ignore the leading bit b_0 (which is just 1)
    - ...what's being displayed is everything after the binary decimal point

SUBNORMAL/Special Numbers

- 0 & - 0 are 2 **different representations** for the **<u>same value</u>** 0
- INF & - INF are 2 **different numbers**
    - INF allows dividing by 0, and return INF instead of DNE
- NaN = Not a Number: error pattern
- **<u>Subnormal number</u>** are represented with special bit pattern in **exponent** field

3 IEEE Floating Point Arithmetic

know this IEEE table below, decimal & binary

1. Single precision
2. Double precision
3. Extended precision

IEEE Single	$\epsilon = 2^{-23} \approx 1.2 \times 10^{-7}$
IEEE Double	$\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}$
IEEE Extended	$\epsilon = 2^{-63} \approx 1.1 \times 10^{-19}$

Single Precision

Table 1: IEEE Single Precision

exp field fraction field

0 = positive, 1 = negative

$\pm$	$a_1a_2a_3 \ldots a_8$	$b_1b_2b_3 \ldots b_{23}$

If exponent bitstring $a_1 \ldots a_8$ is	Then numerical value represented is
$(00000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3 \ldots b_{23})_2 \times 2^{-126}$
$(00000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3 \ldots b_{23})_2 \times 2^{-126}$
$(00000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3 \ldots b_{23})_2 \times 2^{-125}$
$(00000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3 \ldots b_{23})_2 \times 2^{-124}$
$\downarrow$	$\downarrow$
$(01111111)_2 = (127)_{10}$	$\pm(1.b_1b_2b_3 \ldots b_{23})_2 \times 2^{0}$
$(10000000)_2 = (128)_{10}$	$\pm(1.b_1b_2b_3 \ldots b_{23})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111100)_2 = (252)_{10}$	$\pm(1.b_1b_2b_3 \ldots b_{23})_2 \times 2^{125}$
$(11111101)_2 = (253)_{10}$	$\pm(1.b_1b_2b_3 \ldots b_{23})_2 \times 2^{126}$
$(11111110)_2 = (254)_{10}$	$\pm(1.b_1b_2b_3 \ldots b_{23})_2 \times 2^{127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1 = \ldots = b_{23} = 0$, NaN otherwise

these 2 are special NOT normalized

- **Zero (1st Line) >>** need a special zero bit for both exponent and fraction field
    - Initial unstored bit: 0
    - Why 2^{-126}???
        - $(1.0...) \times 2^{-126}$ is the ***<u>smallest normalized</u>*** # we can represent
        - We can actually represent numbers SMALLER than that...**subnormal**
- **Normalized Numbers:** everything not 1st & last line
    - Initial unstored bit: 1
- **Subnormal Numbers [MORE LATER]**
    - **Definition**: (1) Exponent field has zero bitstring (2) Fraction field has nonzero bitstring
    - **<u>Why are subnormal numbers less accurate</u>**?
        - Less room for nonzero bits in fraction field
    - E.g. $2^{-127} = (0.1)_2 \times 2^{-126}$ >>>>>>>>>>>>>>>>>>>>

0	00000000	10000000000000000000000

    - The fraction field is *100…00* bc the leading bit b_0 is 0 and hidden
    - **Smallest Subnormal Number**: $2^{-149} = (0.00...01)_2 \times 2^{-126}$ ...where the 1 is at the end of the fraction field
- **Exponent Field Representation**
    - Does not use 2's complement etc.
    - Uses **<u>biased representation</u>**: bitstring stored is simply the binary representation of **<u>E + 127</u>** >> **<u>Exponent bias</u>**: 127
    - E.g. $1 = (1.000...0)_2 \times 2^0$
        - Stored as >>>>>>>

0	01111111	00000000000000000000000

        - **Exponent Bitstring**: binary representation for 0 + 127
        - **Fraction Bistring:** binary representation for 0 (the fractional part of 1.0)

$= 2^{k-1} - 1$

$= 2^1 - 1$

$= 1$

        - **Practices:** 11/2 and 1/10
- **Exponent Field for <u>Normalized Numbers</u>**
    - Range is $[1, 254]_{10}$ or $[00000001, 11111110]_2$ or [-126, 127]
        - Representing actual exponents values of $E_{min} = -126$ & $E_{max} = 127$
        - **Smallest Normalized Number**: 1.000×2^{-126}
        - **Largest Normalized Number:** $1.1111 \times 2^{127} = (2 - 2^{-23}) \times 2^{127}$
- **Last Line:** when everything is just 1 >>>> Represent ∓INF & NaN...**<u>depends on fraction field</u>**

Double Precision: More accurate bc can store more bits (total **<u>64 bits</u>**)

Table 2: IEEE Double Precision

$\pm$	$a_1a_2a_3\ldots a_{11}$	$b_1b_2b_3\ldots b_{52}$

If exponent bitstring is $a_1\ldots a_{11}$	Then numerical value represented is
$(00000000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1022}$
$(00000000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1022}$
$(00000000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1021}$
$(00000000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{-1020}$
$\downarrow$	$\downarrow$
$(01111111111)_2 = (1023)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{0}$
$(10000000000)_2 = (1024)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11111111100)_2 = (2044)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1021}$
$(11111111101)_2 = (2045)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1022}$
$(11111111110)_2 = (2046)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \times 2^{1023}$
$(11111111111)_2 = (2047)_{10}$	$\pm\infty$ if $b_1 = \ldots = b_{52} = 0$, NaN otherwise

Extended Precision: 80 bit word, sign(1) + exponent (15) + fraction (64)

- Leading bit is NOT hidden! **<u>No hidden bit</u>**
- **First number larger than 1 >>>** $1 + 2^{-63}$

5 - ROUNDING AND CORRECTLY ROUNDED ARITHMETIC

<u>Floating Point Numbers</u>

- Floating point numbers is a small subset of real numbers
- Subnormal
- Normal
- ∓ 0
- ∓ INF
- ** DOES NOT INCLUDE NaN!!!

When X is Not a Floating Number, you either:

1. x_ : closest floating point number LESS than x
2. x_+**:** closest floating point number GREATER than x

Rounding with Single Precision Example

- $x = (b_0b_1b_2\ldots b_{23}b_{24}b_{25}\ldots)_2 \times 2^E$
- When x is positive
    - x_ is between 0 & $x = (b_0b_1b_2\ldots b_{23}b)_2 \times 2^E$ >>> Truncating *m* at the 23rd bit
- When x is negative
    - x_+ is between 0 & $x = (b_0b_1b_2\ldots b_{23}b)_2 \times 2^E$ >>> Truncating *m* at the 23rd bit

Rounding Modes

- <u>Round down</u>: Round(x) = x_
- <u>Round up:</u> Round(x) = x_+
- <u>Round towards zero:</u> either x_ or x_+...**whichever is between 0 & x**
    - **Truncating extra bits**
- <u>Round to nearest</u>: either one, whichever is closest to x. [**most used**]
    - If tied, chose one with its *least significant bit equal to zero*

** always better to round rather than truncate

- Ex. 3.14159… 3.142 is more accurate than 3.141

Absolute Rounding Error (ARE): difference between round(x) and x, but depends on rounding mode

- ARE is less than gap between [x_, x_+]
- ARE for <u>round to nearest</u> is no more than ½ of that gap^
- **For round to nearest:** $|\ round(x) - x\ | < \frac{1}{2}\epsilon \times 2^E$
- **For other rounding modes:** $|\ round(x) - x\ | < \epsilon \times 2^E$

Relative Rounding Error

$\delta = \frac{round(x)}{x} - 1 = \frac{round(x) - x}{x}$ >>> $round(x) = x\,(1 + \delta)$

Since for normalized numbers >>> $x = m \times 2^E,\ where\ m \geq 1$

- **For round to nearest:** $|\delta| \le \frac{\frac{1}{2}\varepsilon\times 2^E}{2^E} = \frac{1}{2}\varepsilon$
- **For other rounding modes:** $|\delta| < \frac{\varepsilon\times 2^E}{2^E} = \varepsilon$

Floating Point Operations

$x \oplus y = (x+y)(1+\delta)$ … δ depends on rounding mode ^^ refer above

Rounding Rules Do Not apply for SEQUENCE OF OPERATIONS

E.g. a + b - c, where a = 1, b = 2^{-25}, c = 1 & using **single precision** and **rounding to nearest**

- First, turn values into binary
  - a = c = 1.0×2^0 and b = 1.0×2^{-25}
- First operation: a + b = 1.0000000000000...001
  - Not single precision point so **<u>rounded to just 1</u>**
- Last operation: 1 - c = 1 - 1 = 0
- **NOT CORRECT:** should be 2^{-25}

<u>How do we correctly round arithmetic operations? Complicated</u>

E.g. x + y, single precision & $x = m \times 2^E$ and $y = p \times 2^F$

Case 1: E = F

- Just add significands *m & p* >>> $(m+p) \times 2^E$
- **normalization >> if m+p > 2 or m+p < 1**
  - Ex. 3 = $(1.100)_2 \times 2^1$ 2 = $(1.000)_2 \times 2^1$

$$\begin{array}{rll} & (\ 1.10000000000000000000000 &)_2 \times 2^1 \\ + & (\ 1.00000000000000000000000 &)_2 \times 2^1 \\ = & (10.10000000000000000000000 &)_2 \times 2^1 \\ \text{Normalize:} & (\ 1.01000000000000000000000 &)_2 \times 2^2. \end{array}$$

Case 2: E > F

- Align the significands; i.e. changing the exponents of one number so E = F
  - So one number will NOT be normalized
- Ex. 3 = $(1.100)_2 \times 2^1$ 3/2 = $(1.100)_2 \times 2^{-1}$
- Normalize if necessary

$$\begin{array}{rll} & (\ 1.10000000000000000000000 &)_2 \times 2^1 \\ + & (\ 0.01100000000000000000000 &)_2 \times 2^1 \\ = & (\ 1.11100000000000000000000 &)_2 \times 2^1. \end{array}$$

Case 3: when one exponent is wayyy to big

Now consider adding 3 to 3×2^{-23}. We get

$$\begin{array}{rll} & (\ 1.10000000000000000000000 &)_2 \times 2^1 \\ + & (\ 0.00000000000000000000001|1 &)_2 \times 2^1 \\ = & (\ 1.10000000000000000000001|1 &)_2 \times 2^1 \\ \text{Round Down:} & (\ 1.10000000000000000000001 &)_2 \times 2^1 \\ \text{or Round Up:} & (\ 1.10000000000000000000010 &)_2 \times 2^1. \end{array}$$

Result is not in single precision floating point syntax

- Has more than 24 bits, shown by "|"
- Normalize first then round
- **Must round; there is a tie so use 24th bit**
  - **Round up**

Cancellation: **all bits in 2 numbers cancel each other**

E.g. x - y; x = $(1.0)_2 \times 2^0$; y = $(1.11....1) \times 2^{-1}$

- y has 23 bits after the binary point; y is the next floating number smaller than x)

$$\begin{array}{rll} & (\ 1.00000000000000000000000| &)_2 \times 2^0 \\ - & (\ 0.11111111111111111111111|1 &)_2 \times 2^0 \\ = & (\ 0.00000000000000000000000|1 &)_2 \times 2^0 \\ \text{Normalize:} & (\ 1.00000000000000000000000|0 &)_2 \times 2^{-24}. \end{array}$$

- We must use a **<u>guard bit</u>**: the extra bit after | aka b_{24}...important for **subtraction**
  - Sometimes we use up to 3 guardbits

Multiplication: no need to align significands >>> $x \times y = (m \times p) \times 2^{E+F}$

- Multiply significand
- Add exponents
- Normalize

6 - EXCEPTIONAL SITUATIONS

Division by Zero

Before IEEE, dividing by zero was to: (1) Find largest floating number (2) Generate interrupt error

Programmers need to make sure that dividing by zero never happens...you dont need to for floating point arithmetic

Ex. resistance formula: $T = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0.$

- If one resistor were to have no resistance, then all current will go through that one...so T = 0

- Valid operations & For any **<u>positive value of a</u>**
    - $a / 0 = \infty$
    - $a \times \infty = \infty$
    - $a \pm \infty$
- **<u>Invalid Operations</u>**: shit that makes no sense, **<u>all equal to NaN</u>**
    - $\infty * 0$
    - $0 / 0$
    - $\infty - \infty$
    - Any NaN related operations
- To recover from NaN, we can look at the fraction field
- 0 = - 0 >>> $\infty \neq -\infty$

Arithmetic Comparisons: e.g. comparing a & b

- Normal comparison operations work for**: finite real numbers, floating point numbers &** $\pm\infty$
- **<u>Unordered</u>**: If a or/and b has NaN value: =, < & > dont work
    - $(a \leq b) \neq (not(a > b))$ >> (true), the second one is (false)

Overflow: the result of arithmetic operation is **<u>finite</u>** but **<u>larger</u>** than the largest floating point number that can be stored

Suppose that the overflowed value is **positive**. Then

rounding model	result
round up	∞
round down	N_{max}
round towards zero	N_{max}
round to nearest	∞

- **Underflow** is said to occur when

$$0 < |\text{ true result }| < N_{min},$$

where N_{min} is the **smallest** normal floating point number.

- Historically the response was usually:

replace the result by zero.

- In **IEEE standard**, the result may be a **subnormal** number instead of zero – allowing results **much smaller** than N_{min}.
- But there may still be a significant loss of accuracy, since subnormal numbers have fewer bits of format.

① Decimal to Fraction

$0.27\overline{27}$

Solution :

$$x = 0.27\overline{27}$$
$$100x = 27.\overline{27}$$
$$100x - x = 27$$
$$x = \frac{27}{99}$$

② Binary ⟺ Decimal

* $101.11001_2 = (\ ?\ )_{10}$

$$= 101 + \frac{11001}{2^5}$$
$$= 2^2+2^0 + \frac{(2^0+2^3+2^4)}{2^5}$$
$$= 5 + (24/32)$$

* $0.0101\ldots_2 = (?)_{10}$

$$x = 0.01\overline{01}$$
$$2^2 x = 1.\overline{01}$$
$$3x = 1$$
$$x = 1/3$$

* $(12)_{10} = (?)_2$

2 | 12	0
2 | 6	0
2 | 3	1
2 | 1	1

$(1100)_{10}$

* $(7.35)_{10} = (?)_2 \quad = 111.010\overline{110}$

2 | 7 ÷2	1
2 | 3	1
2 | 1	1

$(111)_2$

$0.35 \times 2 = 0.7$ $\quad (010\overline{110})_2$
$0.7 \times 2 = 1.4$
$0.4 \times 2 = 0.8$
$0.8 \times 2 = 1.6$
$0.6 \times 2 = 1.2$
$0.2 \times 2 = 0.4$
$0.4 \times 2 = 0.8$

③ 2's complement

* normal way: invert & +1
* fast:
  ① from right, copy everything as it is until first 1 (including the first 1)
  ② invert rest before/on left of 1.

11001001000
←
00110111000

CANCELLATION: complete or partial loss of accuracy
Approximating a Derivative by a Difference Quotient: Good Example of Cancellation
Given a continuous differentiable function, where also exists. We have a program that evaluates .
How do we estimate the derivative of at (i.e.)?

- is the slope of tangent line at
    - Limit of difference quotient = , as h converges to 0
        - Evaluate how small h can get…
        - For , assume &
        - Error = Iderivative - difference quotientI
        - FINDINGS: GRAPH
            - As h gets smaller, error gets smaller. But when h gets too small, its worse… WHY? Let X = 1
            - When h is slightly bigger than machine epsilon, values(sin(x+h) & sin(x)) partially cancel
            - When h is smaller than ½ of machine epsilon, x+h = 1+h is rounded just one
                - No significant digits
            - For , best to use
            - Discretization Error = using h too big
            - Cancellation Error = using h too small
        - FINDINGS: initial phase where error gets smaller as h gets smaller (when h >)
            - Assume exists , so exits. is between
            -
                - Truncate above to Taylor Series
                -
                    - Difference between difference quotient & exact derivative
                    - Discretization error:
    - Difference quotient = slope of line passing through

SUMMARY: cancellation. Use formula for derivative for function is more accurate than difference quotient
Central Difference Quotient: A more accurate approx

-
- Here we assume exists and is continuous
- When h is small but large enough that cancellation doesn't happen, central difference works better
- (A)
    - between x & x+h
- (B)
    - between x & x-h
- If we subtract (A) - (B), & divide by 2h

$$\frac{f(x+h)-f(x-h)}{2h} - f'(x) = \frac{h^2}{12}(f'''(z_1)+f'''(z_2)).$$
<< discretization error for central difference quotient
Discretization error:

EXAMPLE: numerical cancellation

- See if good approximate of a ratio that has square roots in numerator. Multiple everything by numerator/numerator
- Use Central Difference Quotient

ROOTS
BISECTION METHOD

1. Find a midpoint of interval [a,b], where f(a) & f(b) have different signs… p = midpoint
    a. Why different signs? Then we know one is below & other is above and line goes through x=0 (i.e. root)
2. Find (p) and f(a) and multiply
    a. If negative, we take first half of interval [a, p]
    b. If positive, we take [p, b]
3. Repeat

SECANT

1. Requireds 2 initial approximations x[0] & x[1]
2. To find the next value :

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

Newton's Method

1. Using an initial approximation x[0] calculate x[1],x[2]... Using formula:
2. $g(x) = x - \dfrac{f(x)}{f'(x)}$
3. Keep putting x=x[0] into g(x) till answer converges

LINEAR ALGEBRA

UPPER TRIANGULAR SYSTEM

BACK SUBSTITUTION

Example 1

$2x_1 + 3x_2 - 5x_3 = 0$……………….eqn(1)

$7x_2 + 5x_3 = 3$………………. eqn(2)

$2x_3 = 4$ ………………. eqn(3)

LOWER TRIANGULAR SYSTEM

FORWARD SUBSTITUTION

Example 2

$2x_3 = 4$……………….eqn(1)

$7x_2 + 5x_3 = 3$………………. eqn(2)

$2x_1 + 3x_2 - 5x_3 = 0$……………….eqn(3)

Ex: Find the LU Factorization of $A = \begin{bmatrix} 2 & 2 & 3 \\ 5 & 9 & 10 \\ 4 & 1 & 2 \end{bmatrix}$

Goal $A = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ * & 1 & 0 \\ * & * & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix}}_{U}$ $\quad * =$ any real #

$A \rightsquigarrow U$

$\boxed{\text{Replace } R_i \text{ by } R_i - kR_j}$

$$\begin{bmatrix} 2 & 2 & 3 \\ 5 & 9 & 10 \\ 4 & 1 & 2 \end{bmatrix}$$

$$\begin{array}{l} R_2 - \frac{5}{2}R_1 \\ R_3 - 2R_1 \end{array} \begin{bmatrix} 2 & 2 & 3 \\ 0 & 4 & 5/2 \\ 0 & -3 & -4 \end{bmatrix} \begin{array}{l} k = 5/2 \\ k = 2 \end{array}$$

$$R_3 + \frac{3}{4}R_2 \quad \underbrace{\begin{bmatrix} 2 & 2 & 3 \\ 0 & 4 & 5/2 \\ 0 & 0 & -17/8 \end{bmatrix}}_{= U} \quad k = -3/4$$

L: records the k-values

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 5/2 & 1 & 0 \\ 2 & -3/4 & 1 \end{bmatrix}$$

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ 5/2 & 1 & 0 \\ 2 & -3/4 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2 & 3 \\ 0 & 4 & 5/2 \\ & & \end{bmatrix}$$

LU Factorization with no pivoting: https://www.youtube.com/watch?v=a9S6MMcqxr4

GEPP: https://www.youtube.com/watch?v=c0i8hFsOV3A

LU Factorization with partial pivoting: https://www.youtube.com/watch?v=f6RT4BI4S7M&t=59s

Integer Representation
- sign & modulus
- 2's complement $(0, 2^{31}-1)$
  $(2^{31}, 2^{32}-1)$

NonIntegral Representation.
- fixed $(1, 15, 16) = 32$
- floating $(1, 8, 23)$
  - single $(-126, 127) \Rightarrow E + 127$

☆ NaN
- $q \Rightarrow b_1 = 1$
- $s \Rightarrow b_1 = 0$

☆ how do we store subnormal?
- modify fraction field

☆ Find GAP b/w x & next smallest FPN.

$$2^{-23} \times 2^{E}$$

☆ Precision = # bits in significand (+ hidden)

☆ ε = gap b/w 1 & next s FPN

$$\varepsilon = 2^{-23}$$

☆ single: 23 $= (7)_{10}$
double: 52 $(16)_{10}$
quad: 112 $(34)_{10}$
} sigdigs.

☆ Absolute error: $|\text{round}(x) - x| < \varepsilon \times 2^{E}$

☆ relative: $\text{round}(x) = x(1+\delta)$, $|\delta| < \varepsilon$ $= \dfrac{\text{round}(x) - x}{x}$

☆ $x \oplus y = (x \pm y)(1+\delta|$

☆ $a/0 = \infty$, $a/\infty = 0$

☆ Order of Convergence.

$$\lim_{n \to \infty} \frac{|p_{n+1} - p|}{|p_n - p|^{\alpha}} = k$$

$\alpha = 1$ linear
2 quad
3 ....

k = constant of convergence.

P is the convergence.

ex. $p_n = 9^{-2^{n}}$ converges quad to 0

$$\cancel{\lim_{n \to \infty} \frac{|x_{n+1} - x|}{|x_n - x|} = c}$$

$P = 0$

iterations Bisection

$$n \geq \frac{\log\left(\frac{b-a}{\varepsilon}\right)}{\log 2}$$

☆ Bisection Method. (Signs) $\Rightarrow$ guarentee but slow
- If $f(x)$ is continuous on $[a,b]$ & $f(a) \cdot f(b) < 0$, then $\exists r$, where $f(r) = 0$
- linear convergence: $0 < c < 1$

☆ Newton's:

☆ $$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \qquad f'(x_0) \neq 0$$

an appox root.

- quadratic convergence: $c \neq 0$
- error: $e_n = r - x_n$

$$e_{n+1} = C_n (e_n)^2$$

Know when to stop: $\alpha - x_n = x_{n+1} - x_n$

Secant:
- derivative too anoying, so divide diff. to replace f
- superlinearly

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

No pivot!

$A = LU$

$$A = \begin{bmatrix} 1 & 0 & 0 \\ * & 1 & 0 \\ * & * & 1 \end{bmatrix} \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix}$$

L U

① find U first.

- remember k value.

③ L matrix is k-value.

partial pivoting = switching rows.

↳ find entry of largest magnitude

GEPP

① Forward Elimination

$AX = B \Rightarrow UX = C$ (swap if necessary)

② get X using backward sub

$PA = LU$

P = permutation matrix

↳ matrix of every row switching