Cap 1

Recordemos $Ax = b$
el infame de todo el maldito ramo.

Estándar de Punto Flotante y Números Binarios

En este capitulo, intentemos entender detalladamente el IEEE 754 y los números binarios.
Primero vamos con los números binarios

Números Binarios (basic)

Si no cachas NADA de números binarios, esta sección debería ayudar a tener un entendimiento general. Luego profundizaremos.

Los números binarios son la representación de los números como los conocemos, pero usando solo 2 números que los representan. En este caso es el 0 y el 1.
Un ejemplo sencillo de número binario es el 110, el cual es 6.
Una manera sencilla de entender el cómo armar un numero binario en base a un numero decimal (los que conocemos de toda la vida) es tomar el número dividirlo en 2 e ir anotando el resto que obtengamos de cada división.

Hagamos un ejemplo sencillo, el 6

- 6 / 2 = 3, resto 0
- 3 / 2 = 1, resto 1
- 1 / 2 = 0, resto 1
  Ahora lo que tenemos que hacer, es armar el número binario. Tomamos el valor final, en este caso 0 y vamos anotando hacia arriba, deberíamos quedar con 110.

Hagamos otro número, el 14.

- 14 / 2 = 7, resto 0
- 7 / 2 = 3, resto 1
- 3 / 2 = 1 resto 1
- 1 / 2 = 0, resto 1
  El número es 1110

Traducción a numero decimal

La manera más fácil de devolver un numero binario a su representación decimal es la que podemos aprovechar al conocer la naturaleza de los números binarios.
Lo clave es entender que cada numero de un numero binario es una potencia de dos, de derecha a izquierda vamos a partir de la potencia 2^0 hasta 2^n y luego, esos valores se suman dándonos el numero decimal.
Apliquemos el concepto.
Traduzcamos los números de antes.

- 110:
  - De la derecha hacia la izquierda, tenemos 0, luego 1 y otro 1.
  - Partiendo de la potencia de 2^0 hacia adelante podemos formar la suma.
  - $1 * 2^2 + 1 * 2^1 + 0 * 2^0$
  - $4 + 2 + 0 = 6$
- 1110:
  - Haciendo el mismo proceso de antes:
  - $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$
  - $8+4+2 = 14 $
  Tarea, intenta hacer estos procesos con 5, 61 y 255.

Suma y Resta

Es literal igual que sumar números
2 + 3?
Primero pasamos los números a binario
10 y 11
Luego al sumarlos lo hacemos igual que con números decimales
Si sumo 1 con 0, queda 1.
Si sumo 1 con 1, queda 0 y un 1 se va hacia la izquierda.
Si sumo 1 con 1 y hay excedente de antes, queda 1 y otro 1 se va a la izquierda.
0 con 0 queda 0. Así de simple es.

Decimales?

Si tenemos un numero decimal como el 5.625 tendremos que extender la representación.
La parte entera se representa igual que como se explicó antes.
Para obtener la representación de la parte decimal, tendremos que multiplicar recursivamente la parte decimal y sus resultados hasta que quedemos con un 1.
La parte que usaremos para representar serán todos los enteros que obtengamos de cada multiplicación.

- Con 0.625 hacemos esto:
  - $0.625 * 2 = 1.25$
  - $0.25 * 2 = 0.5$
  - $0.5 * 2 = 1.0$
- Entonces la representación de 0.625 en binario es 0.101
  Hagamos otro número, 4.327
  Esta representación será un poco más grande ya que no es muy "bonito" el numero
- 0.327:
  - $0.327 * 2 = 0.654$
  - $0.654 * 2 = 1.308$
  - $0.308 * 2 = 0.616$
  - $0.616 * 2 = 1.232$
  - $0.232 * 2 = 0.464$
  - $0.464 * 2 = 0.928$
  - $0.928 * 2 = 1.856$
  - $0.856 * 2 = 1.712$
  - $0.712 * 2 = 1.424$
  - $0.424 * 2 = 0.848$

  y sigue xd, me da lata seguir. Pero la representación final es esta: 0.010100111011011001.

La suma y la resta son iguales.

Estándar de Punto Flotante

Voy a intentar explicar todo respecto a lo que está en el ppt y los comentarios que puso el Andrés. Más detallado estará en el Apunte oficial del ramo. Muy recomendado leerlo cuidadosamente. Vamos a trabajar con números de precisión doble usando 64 bits.

- 1 bit de signo
- 11 bits de exponente
- 52 bits de mantisa
  La representación es así:
- $$\underbrace{\pm}_{\text{signo}} \underbrace{1}_{\text{Siempre 1}} . \underbrace{bbbbb...}_{\text{mantisa}} \cdot 2^{p \rightarrow \text{exponente}}$$

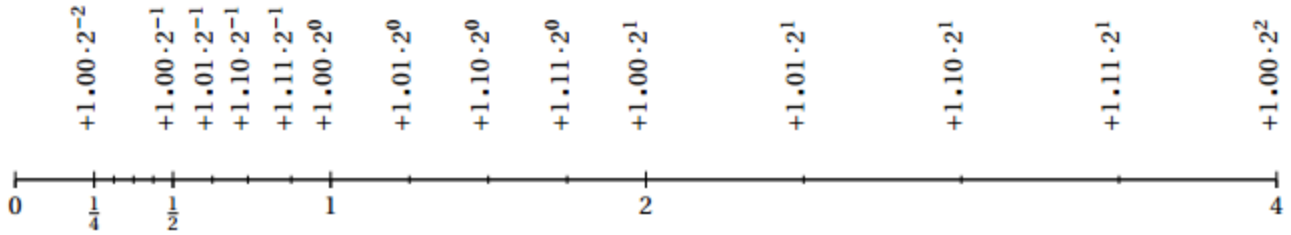
  Estos números se representan de forma **normalizada**. Aquí unos ejemplos
- $9 = 1001$, en punto flotante, necesitamos a un solo numero a la izquierda de la coma. Entonces contamos cuantos son, y guardamos ese número en el exponente.
  $1001 = +1.001 * 2^3$
- $1 = 1 = +1.00 * 2^0$
- $2 = 10 = +1.00 * 2^1$
- $0.5 = 0.1 = +1.00 * 2^{-1}$
- $0.75 = 0.11 = +1.10 * 2^{-1}$

Ahora veamos en más detalle cómo es que se representan estos números de manera completa.
El 1 es un gran ejemplo, simple pero ayuda a establecer una buena base.

- $1 = +1.00000000000000000000000000000000000000000000000000 * 2^0$
  La mantisa está llena de ceros, el exponente es 0 y el signo es positivo.
  Que ocurre con el 2?
- $2 = +1.00000000000000000000000000000000000000000000000000 * 2^1$
  Lo único que cambia es el exponente. Entonces, cuando queramos representar algún numero entre estos dos números, tenemos que usar la mantisa.

Una cosa importante a destacar es que cuando aumentamos el exponente, la precisión bajará y si bajamos el exponente, la precisión aumenta.

Lo que ocurre, es que la cantidad de números que podemos representar es igual siempre. Sin embargo, al cambiar de exponente, la distancia entre la siguiente potencia aumenta y por ende, la precisión disminuye. Piensa en que podemos representar 100 números entre 1 y 2 y luego considera otros 100 números entre 128 y 256. El espacio entre potencias de dos es lo que representa la precisión que podemos tener. Lo mismo pasa con números pequeños, 100 números entre 0.5 y 0.25. En otras palabras, el **espacio** entre números representables aumenta a medida que el exponente se agrande y la precisión disminuye, ya que tengo los mismos recursos para representar. Si exponente disminuye el **espacio** entre números representables será menor.
Esta es una imagen que podría ayudar, si tenemos solo 2 bits de mantisa, podemos representar 4 números:



Machine Epsilon

En corta, es el número más chico que puedo sumar a 1 y no perderlo usando double precision. Si decido sumar un numero más pequeño que este al 1, se pierde y no hay mucho que podamos hacer al respecto.
El número es $2^{-52} \approx 2 \times 10^{-16}$
De este, podemos ir sumando el exponente actual para poder determinar el número más pequeño que podemos sumar a cada potencia de dos. Entonces si el exponente es 1, tenemos al 2 y su ϵ_{mach} es $2^{-52+1} = 2^{-51}$

Algo que podríamos pensar es intentar representar un número entre medio de 1 y ϵ_{mach} lo que ocurrirá será un redondeo.

Reglas de Redondeo

Tenemos 2 posibles rutas

- **Chopping**, quitar los bits que sobran
- **Rounding**, redondear usando un par de reglas
  - Sumar 1 al bit 52 si el bit 53 es 1.
  - Mantener tal cual el bit 52 si el bit 53 es 0.
  - Excepción: Todos los bits después del bit 53 igual a 1 son 0's, se suma solo si el bit 52 es 1.
  Una forma de representar esta regla es usando fl(num) el cual debería darnos un numero redondeado usando estas reglas.
  Ejemplos:
- $fl(+1.001\ldots0001|1001\ldots\cdot 2^8) = +1.0001\ldots0010 \cdot 2^8$
- $fl(+1.001\ldots0001|1000\ldots\cdot 2^5) = +1.0001\ldots0010 \cdot 2^5$
- $fl(+1.001\ldots0100|1000\ldots\cdot 2^8) = +1.0001\ldots0100 \cdot 2^3$

Esta regla de redondeo está explicada para double precision!! Ajustar en caso de usar más o menos bits.
Error relativo de redondeo:

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

Esto es asegurado por las reglas de redondeo.
Un ejemplo:

$$\frac{|fl(9.4) - 9.4|}{|9.4|} = \frac{0.2 \times 2^{-49}}{9.4} = \frac{8}{47} \times 2^{-52} \leq \frac{1}{2}\epsilon_{\text{mach}}$$

Representación de máquina

Recordemos los 3 componentes

- Signo, Exponente y Mantisa.
  En la máquina, se escribe de esta forma.

s	$e_1e_2\ldots e_{10}e_{11}$	$b_1b_2\ldots\ldots\ldots\ldots b_{51}b_{52}$
1 bit	11 bits	52 bits

Ahora, cómo se comporta todo???
Partamos con el **signo**, el más simple de todos.

- s = 1, el número es negativo
- s = 0, el número es positivo

Exponente
Este requiere más explicación

- Con 11 bits, podemos almacenar $2^{11} = 2048$ números.
- Se almacenan como una "lista" $\{0, 1, 2, 3, \ldots, 2046, 2047\}$.
- Debido a que necesitamos exponentes positivo y negativos (como 2^{20} y 2^{-20}) se hace un shift el cual se determina con $2^{11-1} - 1 = 1023$ este se determina *bias*.
    - Algo importante, es que es 11 en este caso porque son 11 bits de exponente. Si tenemos menos bits de exponente, el 11 cambia, por ejemplo a 8 (ejemplo de caso con float) y el bias es $2^{8-1} - 1 = 127$
- El bias traslada los números representables por los 11 bits así:
    - $\{0 - 1023, 1 - 1023, 2 - 1023, 3 - 1023, \ldots, 2047 - 1023\} \to \{-1022, -1021, -1020, \ldots, 1023\}$

Hay dos **casos especiales** con el exponente

- **Exponente 11111111111**
  Tenemos tres situaciones.
- Si el bit del signo es 0 y todos los bits de la mantisa son 0, entonces se representa a $+\infty$. Se obtiene al dividir 1/0
- Si el bit del signo es 1 y todos los bits de la mantisa son 0, entonces se representa a $-\infty$. Es como -1/0
- Si alguno de los bits de la mantisa es distinto de 0, se interpreta como NaN, es decir not-a-number. Se obtiene al dividir 0/0
- **Exponente 00000000000**
  Esta situación corresponde a los numeros no normalizados para representar números subnormales de esta forma:

$$\pm 0.b_1 b_2 b_3 \ldots b_{52} \cdot 2^{-1022}$$

- El exponente es fijo en -1022
    - Este exponente cambia si tenemos menos o más bits de exponente.
    - La fórmula general es: 1 - bias.
        - Si tenemos 8 bits de exponente:
            - $1 - (2^7 - 1) = -126$
- El número junto al signo ya no es 1, sino 0. No se está normalizando el número.
- Solo se pueden modificar los bits de mantisa.
- Esta representación especial permite representar números mucho más chicos que lo que permite la representación normalizada.
  El número más pequeño representable con 64 bits es este:

$$\begin{aligned} \pm 0.0000\ldots01 \cdot 2^{-1022} &= (b_1 \cdot 2^{-1} + b_2 \cdot 2^{-2} + \ldots + b_{52} \cdot 2^{-52} +) \\ &= 1 \cdot 2^{-52} \cdot 2^{-1022} \\ &= 2^{-1074} \approx 4,94 \cdot 10^{-324} \end{aligned}$$

Hay dos formas de representar al 0.

0	0	...	0	0	0	...	0	0
1	0	...	0	0	0	...	0	0
s	e_1	...	e_{11}	b_1	...	...	b_{51}	b_{52}

Sería +0 y -0, pero como es 0, no importa.

Formula IMPORTANTE

Esto lo pusieron en el lab 1 pasado y me lo eché pq no lo ví.
La fórmula importante para convertir un número punto flotante a decimal es la siguiente:

$$(-1)^{signo} * 1 + mantisa * 2^{(exponente - sesgo)}$$

La mantisa se calcula de esta forma:

$$1 + \sum_{i=1}^{52} b_i \cdot 2^{-i}$$

El exponente:

$$p = e_1 \cdot 2^{10} + 3_2 \cdot 2^9 + \ldots + e^{11} \cdot 2^0 - 1023$$

$$p = \underbrace{\left(\sum_{i=1}^{11} e_{12-i} \cdot 2^{i-1} \right)}_{\text{exponente}} - \underbrace{1023}_{\text{sesgo}}$$

Quedamos son:

$$\pm 1.b_1 b_2 \ldots b_{52} \cdot 2^p = (-1)^{signo} * 1 + mantisa * 2^{(exponente - sesgo)}$$

Pérdida de Importancia

Ya que tenemos una cantidad limitada de bits podrían ocurrir cosas con las reglas de redondeo y el almacenamiento de números periódicos que podrían causar problemas.
Un ejemplillo

- $n_1 = +1.0100100000000000000000000000000000000000000000000011 \cdot 2^{-2}$
- $n_2 = +1.1000000000000000000000000000000000000000000000000001 \cdot 2^0$
- Para sumarlos los alineamos con el exponente más significativo.
- $n_1 = +0.0101001000000000000000000000000000000000000000000000|11 \cdot 2^0$
- $n_2 = +1.1000000000000000000000000000000000000000000000000001 \cdot 2^0$
- $n_1 + n_2 = +1.1101001000000000000000000000000000000000000000000001|11 \cdot 2^0$
- Aplicamos redondeo
- $fl(n_1 + n_2) = +1.1101001000000000000000000000000000000000000000000010 \cdot 2^0$
  Al aplicarse este redondeo para poder almacenar el número debido al espacio limitado se puede ver que matemáticamente.

Veamos otro ejemplo

- Primero tengamos dos algoritmos de suma.
  - Algoritmo 1: $(m_1 + m_2) + m_3$.
  - Algoritmo 2: $(m_1 + m_3) + m_2$.
- ahora veamos esos m.
  - $m_1 = +1.00000000000000000000000000000000000000000000000000000 \cdot 2^0$
  - $m_2 = +1.00000000000000000000000000000000000000000000000000000 \cdot 2^{-55}$
  - $m_3 = -1.00000000000000000000000000000000000000000000000000000 \cdot 2^0$
- Está mas detallado en la página 49 del apunte. Pero lo que realmente importa es que en este caso, la suma del algoritmo 1 no permite guardar la suma correctamente, ya que m_2 tiene mucha distancia entre sus bits significativos, recuerda que el ϵ_{mach} lo definimos antes, ve a mirarlo si lo olvidaste.
- En cortas palabras, en el algoritmo 1 ocurre pérdida de importancia debido a que las reglas de redondeo eliminan bits muy a la derecha.
- Algoritmo 1: suelta 0, ya que los bits del m2 se salen de la mantisa de m1.
- Algoritmo 2: suelta m2
  La solución en este caso fue operar números grandes con grandes.
  Idealmente siempre se intenta operar números de tamaños similares para evitar esta pérdida.

Ver notebook: "`Bonus - 02 - Quadratic formula. ipynb`" en el github que está en aula. (y ver otros más, son útiles para saber cómo se enfrentan los problemas en el ramo.)