Instituto Tecnológico de Buenos Aires

93.54 - Métodos Numéricos

Trabajo Práctico 1: Punto Flotante

Grupo 8

Crespo, Elisabet del Pilar	59098
Martorell, Ariel	56209
Vekselman, Alan	59378

Profesores

Fierens, Pablo Ignacio
Grisales Campeón, Juan Pablo

Fecha de entrega: 24/03/2021

1. Introducción

Con el propósito de estudiar el estándar de punto flotante IEEE 754, en este trabajo se propuso realizar un conversor de decimal a punto flotante de 16 bits. Éste está compuesto por 1 bit de signo, 5 bits de exponente y 10 bits de mantisa. El resultante número decimal puede luego dividirse en cinco categorías, según su forma de ser representado por el estándar.

- Cero: El número 0 es representado por un exponente igual a cero y una mantisa igual a cero. Para distinguir entre el "0 positivo" y el "0 negativo" se utiliza el bit de signo. Esta diferencia simboliza el hecho de estar aproximando a cero un número que originalmente pudo haber sido positivo o negativo.
- Infinito: Se representa con todos unos en el exponente y todos ceros en la mantisa. Para distinguir entre $+\infty$ y $-\infty$, se utiliza el bit de signo.
- NaN: La representación de "Not a Number" está dada por todos unos en el exponente y al menos algún bit distinto de cero en la mantisa.
- Números normales: Los números normales son aquéllos cuyo valor absoluto está comprendido entre 2^{1-bias} y 2^{1+bias}, donde *bias* equivale a $2^{n_e-1}-1$, y n_e representa el número de bits del exponente.
- Números subnormales: Los números subnormales son aquéllos cuyo valor absoluto está comprendido entre 2^{1-n_m-bias} y 2^{1-bias}, donde n_m representa el número de bits de la mantisa. Su exponente en representación de punto flotante siempre es cero.

En particular, cuando se trabaja con 16 bits, los límites para las últimas dos categorías serán:

- Números normales: Entre 2^{-14} y 2^{16}.
- Números subnormales: Entre 0 y 2^{-24}.

2. Implementación en Python

2.1. Algoritmo utilizado

En esta sección se hará referencia al código adjunto en el Anexo 3.1. La función llamada "dec2binf", que convierte números decimales a punto flotante IEEE 754 de 16 bits, es un caso particular de la función "dec2binf_gen", que funciona para cualquiera de los estándares IEEE 754. A menos que se especifique lo contrario, se hablará sobre esta última.

Yendo paso a paso por la función, se puede ver que la primera línea es programación defensiva, para cubrirse ante casos donde los parámetros ingresados no sean correctos. Inmediatamente después, se define el parámetro "bias", mencionado en la introducción. Luego se chequea si el número ingresado es NaN. Para este trabajo, se tomó como convención representar al NaN como un número positivo con todos unos en la mantisa.

Si la función continuó, se definen los parámetros "sign" y "abs_num", que representan el signo del número ingresado y el valor absoluto del número, respectivamente. Con este último se trabajará durante toda la función. Se procede a chequear si el número es mayor al máximo representable o menor al mínimo, devolviéndose la representación de $\pm\infty$ o ± 0, respectivamente.

Luego se procede a chequear si el número es normal o subnormal. Para esto, se lo compara con el mínimo valor correspondiente a los normales. En caso de ser mayor, el número será normal, y en caso contrario será subnormal.

Si el número es normal, su exponente real será el entero inferior más cercano a $log_2(num)$. Al dividir al número original por 2^{exp}, se obtiene un número con parte entera igual a uno. Para obtener su parte decimal, se le resta uno, y ese valor se guarda en la variable "dec".

En caso contrario, de ser subnormal, su exponente real será $1-bias$. Sin embargo, para ser consistente con el resultado para números normales, en la variable "exp" se guarda sólo -*bias*. Para llevar al número al formato de números subnormales, es decir, con parte entera igual a cero, se lo debe dividir por 2^{exp+1}. Este último valor se guarda en la variable "dec".

Para la mantisa, se utiliza la función "dec2bin", que convierte un número entero a su representación binaria. Su funcionamiento específico se explicará posteriormente. En particular, se utiliza dicha función para convertir a binario la parte entera del número resultante de multiplicar a la parte decimal por 2^{n_m}, rellenando con ceros hasta n_m bits. La explicación detrás de esto es que, al multiplicar al número por 2^{n_m}, en base dos se está desplazando el número hacia la izquierda n_m cantidad de bits. Al tomar la parte entera de este número, efectivamente se está truncando. Finalmente, la mantisa será dicho número en representación binaria.

Por su parte, el exponente del número en punto flotante será igual a la representación en binario del exponente real más el *bias*, con n_e bits. El resultado final será la concatenación de tres vectores. En primer lugar, el bit de signo; en segundo lugar, el exponente conseguido en el punto anterior; por último, la mantisa.

La función "dec2binf_gen" fue probada y funciona para los estándares de 16, 32, 64 y 128 bits, con complejidad $\approx$ O(1) en términos de velocidad, es decir, todas las conversiones tardan aproximadamente el mismo tiempo.

En cuanto a la función "dec2bin" anteriormente mencionada, su procedimiento es el siguiente: en caso de recibir un número menor o igual a cero, se guardará en la variable "res" un *NumPy array* con un cero. En caso contrario, primero se deberá encontrar la mínima cantidad de bits necesaria para representar al número. Esto se logra tomando la parte entera de $1 + log_2(num)$. Para obtener el resultado, se hace and bit a bit con máscaras. Para poder convertir a unos y ceros el resultado final, se utiliza *casting* a *bool* y luego se lo devuelve a entero, guardando este último resultado en "res".

En caso de que la cantidad de bits en "res" sea menor a la cantidad de bits pasada como parámetro, se rellena a "res" con ceros a la izquierda. Por último, se devuelve "res".

Cabe destacar que, en caso de recibir un número negativo, se tomó como convención devolver cero, ya que en este trabajo se utilizará "dec2bin" exclusivamente con números positivos.

2.2. Verificación del algoritmo

Para corroborar el correcto funcionamiento del algoritmo implementado, se definió la función "test". En ella se decidió probar algunos casos límites, como números mayores al máximo representable, o menores al mínimo. También se comprobaron los valores especiales, como NaN, y los números tanto normales como subnormales. Los resultados fueron corroborados con el siguiente conversor, teniendo la salvedad de que redondea en lugar de truncar.

Finalmente, la función imprimirá " Ok! " en caso de haber obtenido resultados correctos para todos los casos, o " Failed " en caso contrario.

3. Anexo

3.1. Código

```python
import numpy as np
from math import log2

as_int = np.vectorize(int)

# Decimal (int) to binary.
def dec2bin(num: int, width: int = -1) -> np.array:
    if num <= 0: res = np.zeros(1, dtype = int)
    else:
        length = int(log2(num) + 1)
        expos = np.arange(length - 1, -1, -1)
        res = (num & as_int(2.0**expos)).astype(bool).astype(int)

    if res.size < width:
        res = np.concatenate((np.zeros(width - res.size, dtype = int), res))
    return res

def dec2binf_gen(num: float, ne: int, nm: int) -> np.array:

    if ne < 1 or nm < 1:
        return np.zeros(1 + nm + ne, dtype = int)

    BIAS = 2 ** (ne - 1) - 1

    # Nan
    if np.isnan(num):
        return np.concatenate(([0], np.ones(ne + nm, dtype = int)))

    abs_num, sign_bit = abs(num),  num < 0

    # Infinito o Cero
    if abs_num >= 2 ** (1 + BIAS):
        return np.concatenate(([sign_bit], np.ones(ne), np.zeros(nm))).astype(int)
    elif abs_num < 2 ** (1 - BIAS - nm):
        return np.concatenate(([sign_bit], np.zeros(ne + nm, dtype = int)))

    # Normal o Sub-Normal
    if abs_num >= 2**(1 - BIAS):
        exp = int(np.floor(log2(abs_num)))
        dec = abs_num / 2**exp - 1
    else:
        exp = -BIAS
        dec = abs_num / 2**(exp + 1)

    mantissa = dec2bin(int(2**nm * dec), nm)

    return np.concatenate(([sign_bit], dec2bin(exp + BIAS, ne), mantissa))

dec2binf = lambda num: dec2binf_gen(num, ne = 5, nm = 10)
```

```
def test():

    bin2string = lambda n: ''.join(map(str, n))

    cant_ok, cant_tot = 0, 13

    # Mínimo representable
    cant_ok += bin2string(dec2binf(2**-24)) ==  '0000000000000001'

    # Infinito
    cant_ok += bin2string(dec2binf(np.inf)) == '0111110000000000'

    cant_ok += bin2string(dec2binf(-np.inf)) == '1111110000000000'

    # Mayor al máximo -> infinito
    cant_ok += bin2string(dec2binf(2**16)) == '0111110000000000'

    # Menor al mínimo -> cero
    cant_ok += bin2string(dec2binf(2**-25)) == '0000000000000000'

    cant_ok += bin2string(dec2binf(-2**-25)) == '1000000000000000'

    # NaN
    cant_ok +=  bin2string(dec2binf(np.nan)) == '0111111111111111'

    # Normales
    cant_ok += bin2string(dec2binf(23.18914)) == '0100110111001100'

    cant_ok += bin2string(dec2binf(-56219.098)) == '1111101011011100'

    cant_ok += bin2string(dec2binf(-0.5938)) == '1011100011000000'

    # Subnormales
    cant_ok += bin2string(dec2binf(1.53e-5)) == '0000000100000000'

    cant_ok += bin2string(dec2binf(-4.274e-5)) == '1000001011001101'

    cant_ok += bin2string(dec2binf(-3.26e-5)) == '1000001000100010'

    print('Ok!' if cant_ok == cant_tot else 'Failed')
```

```
if __name__ == '__main__':
    test()
```