Métodos Numéricos

Trabajo práctico N°1

Punto Flotante

Grupo 11: CHEN, Carlos Angel, Legajo
MOLDOVAN LOAYZA, Alexander Stephan, Legajo
MOLINA, Facundo Nicolás, Legajo 60526

Profesor: Pablo Ignacio Fierens

Fecha de entrega: 24 de marzo de 2022

Buenos Aires, Argentina

Consigna

Grupo impar

1. Escribir una clase que implemente punto flotante IEEE 754 de 16 bits. El nombre de la clase debe ser binary16 y debe tener dos miembros accesibles:

- **bits**: una lista de 1s y 0s correspondiente a los bits del número;
- **d**: un número en punto flotante de doble precisi[on (64 bits) equivalente al número representado. En caso de que el número representado sea ±inf o nan, *d* deberá ser ±inf o nan.

La función de inicialización de la clase debe recibir un número en punto flotante de doble precisión que deberá ser convertido a IEEE 754 de 16 bits. La clase también deberá implementar las siguientes operaciones numéricas:

- **producto**: a*b, a*=b;
- **cociente**: a/b, a/=b;
- **comparación por igualdad**: a == b;
- **comparación por desigualdad**: a != b.

2. Escribir en el mismo archivo una función sin argumentos llamada *test* que pruebe el correcto funcionamiento de la clase implementada.

Secciones

Índice de Códigos

1. Ejercicio 1

1.1. Class binary16

La clase cumple con la consigna del ejercicio al implementar punto flotante IEEE 754 de 16 bits. Permite la conversión de un número en punto flotante de doble precisión a sus expresiones binarias y flotantes en IEEE 754 de 16 bits, así como realizar las operaciones requeridas en el mismo formato. Para esto último, la clase sobrecarga las operaciones para realizarlas directamente con los símbolos asociados en python.

Para la implementación de la clase, se emplearon las funciones *copysign* e *isnan* de la biblioteca *math*; $false$, de *sympy*; y la expresión X de la biblioteca re de expresiones regulares en Python.

Para implementar la clase de forma mas orgánica, sin recurrir a bibliotecas adicionales, se definieron constantes y funciones para realizar la conversión de punto flotante de doble precisión a su expresión binaria y flotante equivalente. A saber:

- **custom_log2**:
  - $brief$: realiza la función piso del logaritmo en base 2 del input.
  - *input*: float.
  - *output*: float.
- **float_to_list**:
  - $brief$: realiza la conversión de un número en punto flotante a una lista de bits que representa el número en formato punto flotante IEEE754 de 16bits. Utiliza a la función $custom\_log2$. Contempla casos de NAN, +inf, -inf, +0.0 y -0.0.
  - *input*: float.
  - *output*: arreglo de 1s y 0s (True y False).
- **list_to_float**:
  - $brief$: realiza la conversión de un número en binario punto flotante IEEE754 de 16 bits a su equivalente en float nativo. Es la implementación directa de las operaciones de conversión vistas en clase para todos los casos (normal, subnormal, etc). Contempla casos de NAN, +inf, -inf, +0.0 y -0.0.
  - *input*: arreglo de 1s y 0s (True y False).
  - *output*: float.

Al instanciar la clase **binary16**, se pasa como argumento un número float del tamaño del procesador y el init de la clase realiza la conversión mencionada, guardando los resultados en dos datos miembros: **bits** y **d**, en formato binario (representado por un arreglo) y float, respectivamente.

Dado que Python por defecto realiza las operaciones en el tamaño del procesador, los métodos de sobrecarga de operaciones solicitados se implementaron del siguiente modo: se pasa como argumento el resultado de la operación deseada, cuyo resultado esta en formato de procesador, a una instancia de la misma clase, devolviendo esta última para que luego se pueda acceder a los datos miembros con los resultados en IEEE754 de 16bits.

Como nota adicional, no se definieron las sobrecargas de las operaciones *= y /= dado que Python emplea os métodos de * y / para luego realizar la asignación, por lo que la sobrecarga hubiese resultado innecesaria.

Se adjunta el código del ejercicio.

Código 1: class binary16

```
from math import copysign,isnan
from re import X
from sympy import false

TBITS = 16              # Número de bits totales que contiene el número flotante a desarrollar
EBITS = 5               # Número de bits que componen el exponente
MBITS = TBITS - (1+EBITS)  # Número de bits que componen la mantisa

BIAS = 2**(EBITS-1)-1      # Sesgo del exponente

BINARY16_BITS_NAN = [True]*TBITS                    # Representación de "NaN" en punto
    ↪ flotante
BINARY16_BITS_INF = [False]+[True]*EBITS+[False]*MBITS  # Representación de "-inf" en
    ↪ punto flotante
BINARY16_BITS_MINF= [True]*(1+EBITS)+[False]*MBITS      # Representación de "+inf" en
    ↪ punto flotante
BINARY16_BITS_0   = [False]*TBITS                   # Representación de +0.0 en punto flotante
BINARY16_BITS_M0  = [True] + [False]*(EBITS + MBITS)    # Representación de -0.0 en punto
    ↪ flotante

# custom_log2
# Recibe: número en formato punto flotante
# Devuelve: función piso del logaritmo en base 2 de dicho número
def custom_log2(float_num: float):
    if isnan(float_num):          # log2(NaN) = NaN
        log =  float('nan')
    elif float_num < 0:           # log2(x) = NaN si x<0
        log =  float('nan')
    elif float_num == float('inf'): # log2(inf) = inf
        log =  float('inf')
    elif float_num == 0:          # log2(0) = -inf
        log =  float('-inf')
    elif float_num == 1:          # log2(1) = 0
        log =  0
    elif float_num < 1:           # Caso 0 < x < 1
        log = 0
        while float_num < 1:          # El número será multiplicado por 2 tantas veces como
            log -= 1                  # sea necesario hasta que tenga la forma 1.mantisa
            float_num *= 2            # Esa cantidad de veces representa el exponente negativo
    else:                         # Caso 1 < x < inf
        log = 0
        while float_num >= 2:         # El número será dividido por 2 tantas veces como
            log += 1                  # sea necesario hasta que tenga la forma 1.mantisa
            float_num /= 2            # esa cantidad de veces representa el exponente positivo
    return log
```

```
42
43 # float_to_list
44 # Recibe: número en formato punto flotante
45 # Devuelve: lista de bits que representan el número en formato float de 16 bits
46 def float_to_list(float_num: float):
47     if isnan(float_num):            # Se analizan por separado los casos: "NaN"
48         return BINARY16_BITS_NAN
49     elif float_num == float('inf'):   # +inf
50         return BINARY16_BITS_INF
51     elif float_num == float('-inf'):  # -inf
52         return BINARY16_BITS_MINF
53
54     sign = copysign(1,float_num) # Obtenemos el signo del númeor ingresado. El motivo
55                             # por el cual se emplea esta función es porque las
56                             # comparaciones como <, <=, > y >= no pueden distinguir
57                             # entre +0.0 y -0.0
58     if float_num == 0:  # Se analiza por separado los casos:
59         if sign == 1:                # +0.0
60             return BINARY16_BITS_0
61         else:                       # -0.0
62             return BINARY16_BITS_M0
63
64     exp = custom_log2(abs(float_num)) # Obtenemos el log2 del número ingresado
65
66     if exp > 2**EBITS-1 - BIAS:       # Si el exponente es superior al máximo representable
67         if sign == 1:               # el número representado será infinito, con el signo correspondiente
68             return BINARY16_BITS_INF
69         else:
70             return BINARY16_BITS_MINF
71
72     man_list = []
73     float_num = abs(float_num)      # Reemplazamos al número ingresado por su valor absoluto
74
75     if exp < 1 - BIAS:            # Si el exponente es menor al mínimo representable, se trata de un
76                             # número "sub normal"
77         float_num /= 2**(1 - BIAS)  # Preparamos al número para ser representado mediante el
    ↪ menor
78         exp = 0                 # exponente permitido (si bien el exponente indicará "0" en binario)
79
80     else:                       # Si el número es "normal"
81         float_num /= 2**exp         # Lo dividimos para que tenga la forma 1.mantisa
82         float_num -= 1            # Le sustraemos el 1, el cual será implícito
83         exp += BIAS               # Le agregamos el sesgo al exponente para representarlo
84
85     for pos in range(MBITS):        # Por cada uno de los bits que compondrán la mantisa
86         float_num*=2              # En binario, movemos a la izquierda una posición todos los bits
87         if float_num >= 1:        # Si a la izquierda del punto decimal hay un 1
88             float_num-=1              # lo restamos
89             man_list += [True]        # y agregamos un 1 lógico a la mantisa
90         else:                   # Si a la izquierda del punto decimal hay un 0
91             man_list += [False]       # agregamos un 0 lógico a la mantisa
```

```
92
93      exp_list = []
94      for pos in range(EBITS):          # Por cada uno de los bits que compondrán el exponente
95          if exp >= 2**(EBITS-(1+pos)):   # Si el bit analizado es 1 (si el número es mayor o igual a
        ↪ "100...00b", con cantidad de 0 decrecientes)
96              exp-= 2**(EBITS-(1+pos))       # lo restamos
97              exp_list+=[True]              # y agregamos un 1 lógico al exponente
98          else:                          # Si el bit analizado es 0
99              exp_list+=[False]             # agregamos un 0 lógico al exponente
100
101     return [sign == -1] + exp_list + man_list   # Armamos la lista juntando todas sus partes
102
103 # list_to_float
104 # Recibe: una lista de bits que representan un número de punto flotante de 16 bits
105 # Devuelve: el mismo número, en formato float nativo
106 def list_to_float(list_num):
107     if list_num == BINARY16_BITS_INF:    # Analizamos por separado los casos: +inf
108         return float('inf')
109     elif list_num == BINARY16_BITS_MINF: # -inf
110         return float('-inf')
111
112     sign = list_num[0]                    # Separamos las partes del número: signo
113     exp_list = list_num[1:(EBITS+1)]       # exponente
114     man_list = list_num[(EBITS+1):TBITS]    # mantisa
115
116     if exp_list == [True]*EBITS and man_list != [False]*MBITS: # Si el exponente tiene solo 1s, y
        ↪ la mantisa no es sólamente 0s
117         return float('nan')                              # se trata de un "NaN"
118
119     man = 0
120     for position in range(MBITS):                 # Por cada bit de la mantisa (del menos significativo
        ↪  al mas significativo)
121         man += (2**position)*man_list[-(1+position)] # sumamos su aporte al número
122     man /= 2**(MBITS)                             # y corremos todos los bits detrás del punto decimal
        ↪ , obteniendo 0.mantisa
123
124     if exp_list == [False]*EBITS:   # De tratarse de un número "sub normal"
125         exp = 1                       # el exponente considerado será 1
126     else:                         # De tratarse de un número "normal"
127         man += 1                      # Agregamos el 1 implícito de la mantisa
128         exp = 0
129         for position in range(EBITS):   # y armamos el exponente sumando la contribución de cada
        ↪ bit
130             exp += (2**position)*exp_list[-(1+position)]
131     exp -= BIAS                   # Luego, le restamos el sesgo al exponente (obteniendo el valor real
        ↪ del mismo)
132
133     return ((-1)**sign * man * 2**exp)  # Fórmula para calcular el valor de un número de punto
        ↪ flotante "a mano"
134
135
```

```
136
137 class binary16:
138
139     def __init__(self,float_num: float):
140         if type(float_num) is not float:    # Si el número que le pasaron al constructor no es float
141             float_num = float(float_num)    # lo castea a float para asegurarse que el programa
    ↪ funcione
142                                             # de manera uniforme
143         self.bits = float_to_list(float_num)    # Transformamos la variable de entrada en un arreglo
    ↪ de bools,
144                                                 # correspondientes al valor en punto flotante de 16 bits
145         self.d = list_to_float(self.bits)       # Finalmente, transformamos este nuevo valor en punto
    ↪ flotante
146                                                 # nativo de python (64 bits), perdiendose la precisión del valor
    ↪ original
147
148     def __mul__(self,other):
149         return binary16(self.d * other.d)   # No se requiere consideraciones especiales a la hora de
    ↪ realizar el producto
150                                             # de dos números en formato punto flotante nativo (python respeta
    ↪  las reglas
151                                             # pedidas, por ejemplo: inf*0=NaN )
152
153     def __truediv__(self,other):
154         if other.d == 0:                    # Sí se debe tener en cuenta el caso del 0 como divisor (ya que
    ↪ python produce un error)
155
156             if self.d == 0:                 # Si AMBOS operandos son 0:   0/0=NaN
157                 div = float('nan')
158             elif isnan(self.d):             # Si el dividendo es NaN:   NaN/0=NaN
159                 div = float('nan')
160             else:                           # Si el dividendo es distinto de 0 o NaN (esto incluye valores "
    ↪ normales", "sub normales"
161                                             # +inf y -inf):             var/0=inf, con el signo correspondiente
162                 if copysign(1,self.d) == copysign(1,other.d): # operandos con mismo signo
163                     div = float('inf')              # resultado positivo (y viceversa)
164                 else:
165                     div = float('-inf')
166         else:
167             div = self.d / other.d          # Si el divisor NO es 0, entonces python realiza la división de
    ↪ forma acorde a lo espeado
168                                             # (por ejemplo: inf/inf=NaN )
169         return binary16(div)
170
171     # NOTA:
172     # Podrían haberse definido de manera análoga los operadores __imul__ (*=) y __itruediv__
    ↪ (/=)
173     # Sin embargo, en caso de no realizarse una sobrecarga para dichos operadores, por defecto
174     # python emplea los métodos __mul__ y __truediv__ y luego realiza la asignación, por lo cual
175     # dicha sobrecarga de operadores es innecesaria
176
```

```
177     def __eq__(self,other):
178         if isnan(self.d) and isnan(other.d): # Debe analizarse por separado el caso de que ambas
        ↪ variables sean "NaN", debido a que
179             return True                 # python por defecto considera que toda variables es distinta a
        ↪ un "NaN", incluyendo otro "NaN"
180         elif (self.d == 0.0) and (other.d == -0.0):
181             if(self.bits[0] == other.bits[0]):
182                 return True
183             else:
184                 return False
185         return (self.d == other.d)  # Si solo una de las variables es NaN, la respuesta será False, tal
        ↪ como se espera.
186                                 # Si ninguna es NaN, la comparación funciona de la forma esperada
187 
188     def __ne__(self,other):
189         if isnan(self.d) and isnan(other.d): # Análogamente al caso anterior, comparaciones de
        ↪ desigualdad entre un "NaN"
190             return False                # y otra variable (incluyendo "NaN") devolverán siempre "True
        ↪ ". Por ello, se analiza este caso
191         elif (self.d == 0.0) and (other.d == -0.0):
192             if(self.bits[0] == other.bits[0]):
193                 return False
194             else:
195                 return True
196         return (self.d != other.d)
```

2. Ejercicio 2

2.1. test

La función *test*() utiliza un arreglo de valores predeterminados. Estos consisten en los siguientes tipos de números:

- float
- sub-float
- 0.0
- -0.0
- $+\infty$
- $-\infty$
- nan

En base a esta lista de valores predeterminados, se definieron los valores de verificación. Estos están guardados en 4 arreglos diferentes: `mul_verification[]` , `div_verification[]`, `equ_verification[]` y `no_equ_verification[]`.

Es posible observar que el largo de los arreglos `mul_verification[]`, `equ_verification[]` y `no_equ_verification[]` son iguales, debido a que se definió que solamente se analice los 28 casos de cada operando. Esto quiere decir que se consideró que las operaciones `(float * sub-float)` es igual que `(sub-float * float)`. Y así también para las operaciones de == y ! =.

Sin embargo para la operación de división se consideró los 49 casos, teniendo en cuenta que las operaciones `(float / sub-float)` y `(sub-float / float)` son distintas.

Una vez definido los parámetros para el test y los arreglos para la verificación, se definió 4 arreglos vacíos:

- `mul_test[]`
- `div_test[]`
- `equ_test[]`
- `no_equ_test[]`

Cada uno de estos arreglos tendrá el resultado de la operación correspondiente pasando por la clase `binary16()` y guardándose como un objeto.

Finalmente, se calculó la cantidad de casos exitosos totales que tiene que tener (133 casos) y se procedió a iterar con un `for` y comparar los resultados obtenidos con los valores de verificación. Por cada caso exitoso se suma un 1 a la variable acumulador `pass_cases` y una vez que termine de iterar se compara con el valor de casos exitosos ideales para comprar si superó todos los casos de análisis.

Código 2: Definición de test

```
def test():
    #Todos los numeros para usar en el testeo
    num_test = [
              ["float", "sub-float", "0", "-0", "inf", "-inf", "nan"],
              [5.984634, 0.345788, 0.0, -0.0, float('inf'), float('-inf'), float('nan')]
              ]

    mul_verification = [35.8125, 2.068359375, 0.0, -0.0, float('inf'), float('-inf'), float('nan'),
                        0.1195068359375, 0.0, -0.0, float('inf'), float('-inf'), float('nan'),
                            0.0, -0.0, float('nan'), float('nan'), float('nan'),
                                0.0, float('nan'), float('nan'), float('nan'),
                                    float('inf'), float('-inf'), float('nan'),
                                        float('inf'), float('nan'),
                                            float('nan')]

    div_verification = [1.0,17.296875,float('inf'),float('-inf'),0.0,-0.0,float('nan'),
                        0.0577392578125,1.0,float('inf'),float('-inf'),0.0,-0.0,float('nan'),
                        0.0,0.0,float('nan'),float('nan'),0.0,-0.0,float('nan'),
                        -0.0,-0.0,float('nan'),float('nan'),-0.0,0.0,float('nan'),
                        float('inf'),float('inf'),float('inf'),float('-inf'),float('nan'),float('nan'),float('nan'),
                        float('-inf'),float('-inf'),float('-inf'),float('inf'),float('nan'),float('nan'),float('nan'),
                        float('nan'),float('nan'),float('nan'),float('nan'),float('nan'),float('nan'),float('nan')]

    equ_verification = [True,False,False,False,False,False,False,
                            True,False,False,False,False,False,
                                True,False,False,False,False,
```

```
                                True,False,False,False,
                                    True,False,False,
                                        True,False,
                                            True]

no_equ_verification = [False,True,True,True,True,True,True,
                            False,True,True,True,True,True,
                                False,True,True,True,True,
                                    False,True,True,True,
                                        False,True,True,
                                            False,True,
                                                False]

#Tomo los numeros de test y lo paso a 16 bits
num_test_bin16 = []
for i in range(len(num_test[1])):
    num_test_bin16.append(binary16(num_test[1][i]))

#Creo arreglo de los resultados multiplicando todos los casos
mul_test = []
for i in range(len(num_test_bin16)):
    for j in range(i, len(num_test_bin16)):
        mul_test.append(num_test_bin16[i]*num_test_bin16[j])

#Creo arreglo de los resultados comparando con == todos los casos
equ_test = []
for i in range(len(num_test_bin16)):
    for j in range(i, len(num_test_bin16)):
        equ_test.append(num_test_bin16[i]==num_test_bin16[j])

#Creo arreglo de los resultados comparando con != todos los casos
no_equ_test = []
for i in range(len(num_test_bin16)):
    for j in range(i, len(num_test_bin16)):
        no_equ_test.append(num_test_bin16[i]!=num_test_bin16[j])

#Creo arreglo de los resultados multiplicando todos los casos
div_test = []
for i in range(len(num_test_bin16)):
    for j in range(len(num_test_bin16)):
        div_test.append(num_test_bin16[i]/num_test_bin16[j])

mul_equ_noequ_cases = 28*3
div_cases = 49
total_PASS = mul_equ_noequ_cases + div_cases

pass_cases = 0
fail_cases = 0
for i in range(len(mul_test)):
    if mul_test[i].d == mul_verification[i]:
        pass_cases+=1
```

```
78      elif isnan(mul_test[i].d) and isnan(mul_verification[i]):
79          pass_cases+=1
80      else:
81          fail_cases+=1
82
83  for i in range(len(equ_test)):
84      if equ_test[i] == equ_verification[i]:
85          pass_cases+=1
86      else:
87          fail_cases+=1
88
89  for i in range(len(no_equ_test)):
90      if no_equ_test[i] == no_equ_verification[i]:
91          pass_cases+=1
92      else:
93          fail_cases+=1
94
95  for i in range(len(div_test)):
96      if div_test[i].d == div_verification[i]:
97          pass_cases+=1
98      elif isnan(div_test[i].d) and isnan(div_verification[i]):
99          pass_cases+=1
100     else:
101         fail_cases+=1
102
103 if pass_cases == total_PASS:
104     print("You pass all cases")
105 else:
106     print("You fail in {fail_cases} cases")
```