Punto_Flotante

April 7, 2022

0.0.1 Repaso de representación de número flotante

[1]:
```
using Plots
```

Recordemos que los números flotantes tienen la forma:

$$x = (-1)^s * 0.a_1a_2a_3 \ldots a_t * \beta^e = (-1)^s * m * \beta^e, \qquad \frac{1}{\beta} < m < 1$$

Donde el primer elemento, a_1 es no nulo (distinto de cero). En el caso de la representación binaria, $\beta = 2$, tendremos que necesariamente $a_1 = 1$ y por lo tanto no lo incluiremos.

0.1 Ejemplo de representación de número en punto flotante simple (Float32)

Consideremos el número: x = -118.625

Su representación binaria es: x_bin = -1110110.101

Lo chequeamos:

[2]:
```
-(2^6+2^5+2^4+2^2+2^1+2^(-1)+2^(-3))
```

[2]:
```
-118.625
```

Para pasarlo a la representación de punto flotante 32, primero corremos la coma hasta el primer lugar no nulo de la izquierda (7 lugares):

$x_{bin} = (-1)^1 * (0.1110110101) * 2^7$

Como 7 en binario es: `111` Pero debemos usar la *representación sesgada* y por lo tanto ir a $E = 126 + 7 = 133$ que se representa como: `10000101`

[3]:
```
2^2+2^1+2^0
```

[3]:
```
7
```

[4]:
```
2^7+2^2+2^0
```

[4]:
```
133
```

Como el primer elemento de la mantiza debe ser necesariamente no nulo debe ser un 1, que se omite. La representación será:

```
x_float32 = 1_10000101_110110101000
```

[5]:
```
fl_x = Float32(-118.625)
```

[5]:
```
-118.625f0
```

[6]:
```
bitstring(fl_x)
```

[6]:
```
"11000010111011010100000000000000"
```

0.1.1 Ejemplo con F(2,3,-1,2)

$\beta = 2$, $t = 3$, $L = -1$ y $U = 2$ (sesgado por dos unidades, $E = e + L + 1$)

Listemos primero los **normalizados** (los positivos solamente):

$$
\begin{array}{llll}
(0.100)_2 \times 2^{-1} = \frac{1}{4}, & (0.101)_2 \times 2^{-1} = \frac{5}{16}, & (0.110)_2 \times 2^{-1} = \frac{3}{8}, & (0.111)_2 \times 2^{-1} = \frac{7}{16}, \\
(0.100)_2 \times 2^{0} = \frac{1}{2}, & (0.101)_2 \times 2^{0} = \frac{5}{8}, & (0.110)_2 \times 2^{0} = \frac{3}{4}, & (0.111)_2 \times 2^{0} = \frac{7}{8}, \\
(0.100)_2 \times 2^{1} = 1 & (0.101)_2 \times 2^{1} = \frac{5}{4}, & (0.110)_2 \times 2^{1} = \frac{3}{2}, & (0.111)_2 \times 2^{1} = \frac{7}{4}, \\
(0.100)_2 \times 2^{2} = 2, & (0.101)_2 \times 2^{2} = \frac{5}{2}, & (0.110)_2 \times 2^{2} = 3, & (0.111)_2 \times 2^{2} = \frac{7}{2}.
\end{array}
$$

[7]:
```
L = -1
U = 2
  = 2.

for j in 0:( -1)
    for k in 0:( -1)
        for e in L:U
            println(( ^(-1) + j* ^(-2) + k* ^(-3))* ^e)
        end
    end
end
```

```
0.25
0.5
1.0
2.0
0.3125
0.625
1.25
2.5
0.375
0.75
```

```
1.5
3.0
0.4375
0.875
1.75
3.5
```

Ahora los ordenamos para graficarlos:

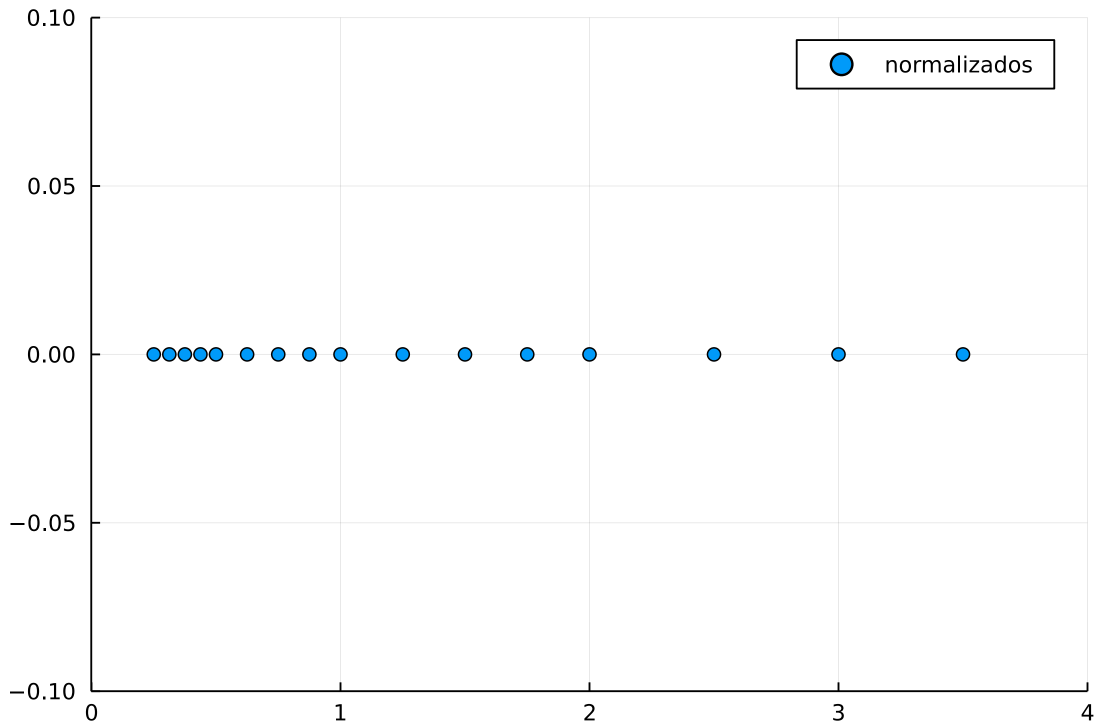
[8]:
```
x_fln = zeros(2*2*4) # 16 lugares

for j in 0:( -1)
    for k in 0:( -1)
        for e in L:U
            x_fln[Int(1+ 2*j+ k +4*(e-L))] = ( ^(-1) + j* ^(-2) + k* ^(-3))* ^e
        end
    end
end
```

[9]:
```
x_fln
```

[9]:
```
16-element Vector{Float64}:
 0.25
 0.3125
 0.375
 0.4375
 0.5
 0.625
 0.75
 0.875
 1.0
 1.25
 1.5
 1.75
 2.0
 2.5
 3.0
 3.5
```

[10]:
```
cero(x) = 0.0
scatter(x_fln,cero.(x_fln), xlim=(-0.001,4.0), ylim=(-0.1,0.1),␣
 ↪label="normalizados")
```

[10]:



0.2 Números especiales

Dada una representación $\mathcal{F}(\beta, t, L, U)$

Además de los números ya incluidos las representaciones modernas (IEEE754) agregan varios más:

1. El cero: mantiza 0 y e = (L-1)
2. Números denormalizados (a_1 puede ser cero) y e = (L-1). Cubren la región cercana a zero.
3. Inf y -Inf resultados de dividir por cero o de overflow (positivo y negativo) mantiza nula y e = U+1
4. NaN que son resultados ilegales, como por ejemplo `0/0`

$$(\pm\texttt{Infinity}\ ) + (+1) = \pm\texttt{Infinity}$$
$$(\pm\texttt{Infinity}\ ) \cdot (-1) = \mp\texttt{Infinity}$$
$$(\pm\texttt{Infinity}\ ) + (\pm\texttt{Infinity}\ ) = \pm\texttt{Infinity}$$
$$(\pm\texttt{Infinity}\ ) + (\mp\texttt{Infinity}\ ) = \texttt{NaN}$$
$$1/(\pm 0) = \pm\texttt{Infinity} \qquad 1/(\pm\texttt{Infinity}\ ) = \pm 0$$
$$0/0 = \texttt{NaN} \qquad (\pm\texttt{Infinity}\ )/(\pm\texttt{Infinity}\ ) = \texttt{NaN}$$
$$0 \cdot (\pm\texttt{Infinity}\ ) = \texttt{NaN}$$

Valor	Exponente	Mantisa
normalizados	$L \leq e \leq U$	$\neq 0$
denormalizados	$L-1$	$\neq 0$
± 0	$L-1$	0
$\pm$`Infinity`	$U+1$	0
`NaN`	$U+1$	$\neq 0$

[11]:
```
1/0
```

[11]: Inf

[12]:
```
bitstring(Inf)
```

[12]: "0111111111110000000000000000000000000000000000000000000000000000"

Ahora veamos cuales son los **desnormalizados** de la representación $\mathcal{F}(2,3,-1,2)$:

$$0.001 \times 2^{-1} = \frac{1}{16}, \quad 0.010 \times 2^{-1} = \frac{1}{8}, \quad 0.011 \times 2^{-1} = \frac{3}{16}.$$

[13]:
```
x_fld = zeros(4)
for j in 0:1
    for k in 0:1
            x_fld[1+k+2*j] = (0 + j* ^(-2) + k* ^(-3))* ^(L)
```

```julia
    end
end
```

[14]:
```julia
x_fld
```

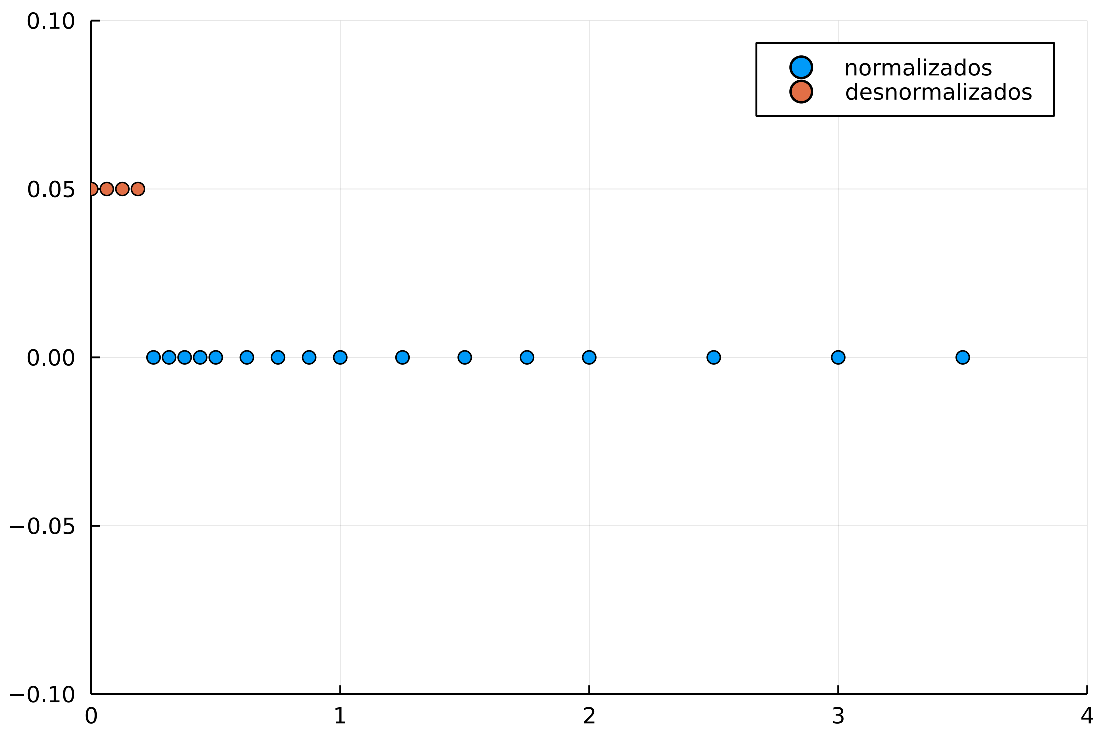
[14]:
```
4-element Vector{Float64}:
 0.0
 0.0625
 0.125
 0.1875
```

[15]:
```julia
3/16
```

[15]:
```
0.1875
```

[16]:
```julia
scatter(x_fln,cero.(x_fln), xlim=(-0.001,4.0), ylim=(-0.1,0.1),␣
 ↪label="normalizados")
scatter!(x_fld,cero.(x_fld).+0.05, label="desnormalizados")
```

[16]:



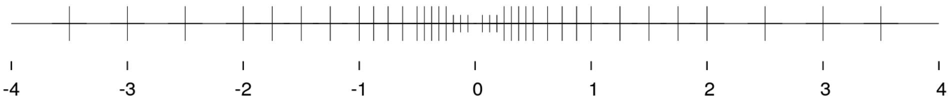
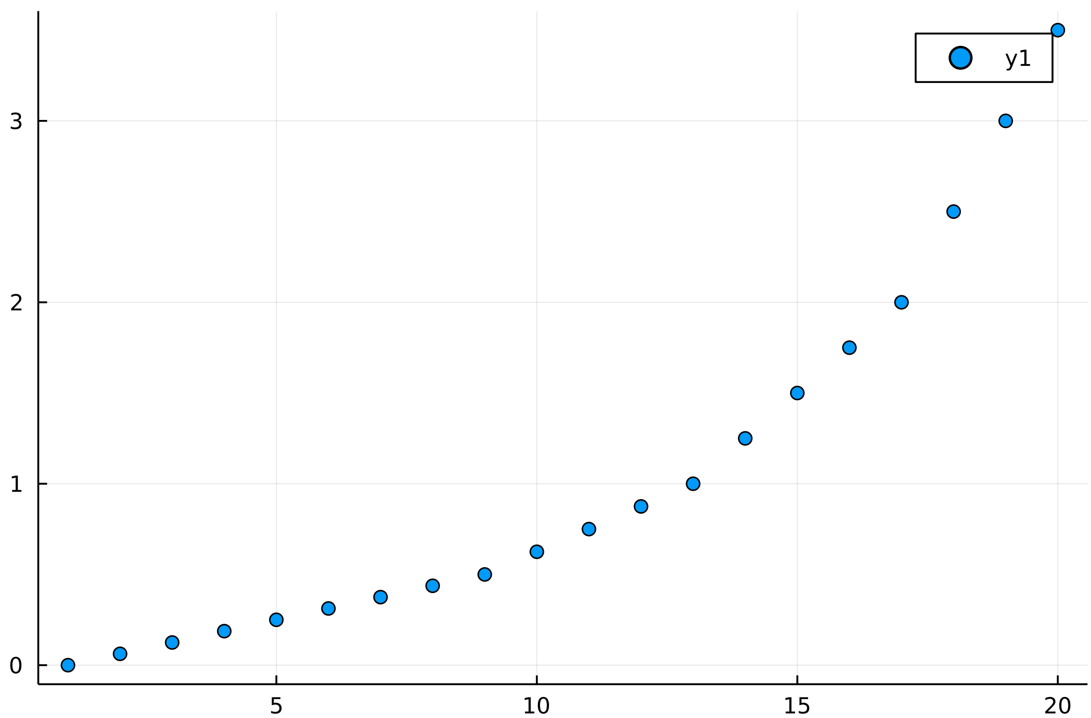


Figura 4. Números de punto flotante, incluyendo los denormalizados, del conjunto $\mathbb{F}(2, 3, -1, 2)$.

[17]:
```
scatter([x_fld;x_fln])
```

[17]:



0.2.1 Los ceros

[18]:
```
bitstring(0.0)
```

[18]: "0000000000000000000000000000000000000000000000000000000000000000"

Hay por lo tanto dos ceros....

[19]:
```
bitstring(-0.0)
```

[19]: "1000000000000000000000000000000000000000000000000000000000000000"

[20]:
```
1/(-0.)
```

[20]:
```
-Inf
```

0.2.2 Distancia entre números consecutivos:

La distancia entre dos números consecutivos en el intervalo $x \in [\beta^e, \beta^{e+1})$:

$$\epsilon(x) = \frac{1}{\beta^t} * \beta^{e+1} = \beta^{e-t+1}$$

La distancia al número más cercano a 1 (POR ARRIBA) es $\epsilon(1) = \beta^{1-t}$ (ya que $1 = 0.1\beta^1$)

La distancia a un número se denomina ϵ y depende de la representación que la máquina y el software haga de los números.

En Julia (*en FORTRAN también*) hay una función que nos devuelve el valor de esta distancia para cada número que le demos:

[21]:
```
?eps()
```

[21]:

```
eps(::Type{T}) where T<:AbstractFloat
eps()
```

Return the *machine epsilon* of the floating point type `T` (`T = Float64` by default). This is defined as the gap between 1 and the next largest value representable by `typeof(one(T))`, and is equivalent to `eps(one(T))`. (Since `eps(T)` is a bound on the *relative error* of `T`, it is a "dimensionless" quantity like `one`.)

1 Examples

```
julia> eps()
2.220446049250313e-16

julia> eps(Float32)
1.1920929f-7

julia> 1.0 + eps()
1.0000000000000002

julia> 1.0 + eps()/2
1.0
```

```
eps(x::AbstractFloat)
```

Return the *unit in last place* (ulp) of `x`. This is the distance between consecutive representable floating point values at `x`. In most cases, if the distance on either side of `x` is different, then the larger of the two is taken, that is

```
eps(x) == max(x-prevfloat(x), nextfloat(x)-x)
```

The exceptions to this rule are the smallest and largest finite values (e.g. `nextfloat(-Inf)` and `prevfloat(Inf)` for `Float64`), which round to the smaller of the values.

The rationale for this behavior is that `eps` bounds the floating point rounding error. Under the default `RoundNearest` rounding mode, if y is a real number and x is the nearest floating point number to y, then

$$|y - x| \leq \operatorname{eps}(x)/2.$$

See also: `nextfloat`, `issubnormal`, `floatmax`.

2 Examples

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(prevfloat(2.0))
2.220446049250313e-16

julia> eps(2.0)
4.440892098500626e-16

julia> x = prevfloat(Inf)      # largest finite Float64
1.7976931348623157e308

julia> x + eps(x)/2            # rounds up
Inf

julia> x + prevfloat(eps(x)/2) # rounds down
1.7976931348623157e308
```

```
eps(::Type{DateTime}) -> Millisecond
eps(::Type{Date}) -> Day
eps(::Type{Time}) -> Nanosecond
eps(::TimeType) -> Period
```

Return the smallest unit value supported by the `TimeType`.

3 Examples

```
julia> eps(DateTime)
1 millisecond

julia> eps(Date)
1 day
```

```
julia> eps(Time)
1 nanosecond
```

[22]:
```julia
eps(1.)
```

[22]:
```
2.220446049250313e-16
```

[23]:
```julia
bitstring(eps(0.))
```

[23]:
```
"0000000000000000000000000000000000000000000000000000000000000001"
```

[24]:
```julia
1. + eps(1.)*0.5 == 1.
```

[24]:
```
true
```

[25]:
```julia
1. + eps(1.)*0.6
```

[25]:
```
1.0000000000000002
```

[26]:
```julia
eps(10.)
```

[26]:
```
1.7763568394002505e-15
```

[27]:
```julia
10. + eps(1.) == 10.
```

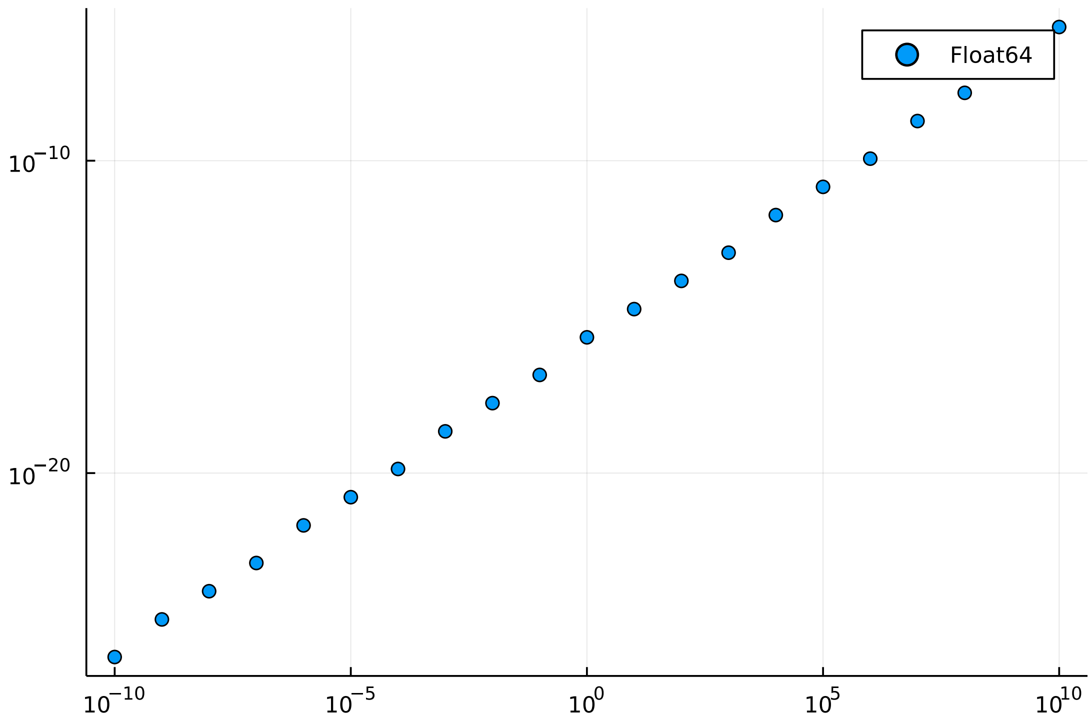
[27]:
```
true
```

Veamos en un gráfico como crece la distancia entre números consecutivos:

[28]:
```julia
x = [10.0^i for i   -10:10];
```

[29]:
```julia
scatter(x,eps.(x),yscale=:log10, xscale=:log10, label="Float64")
```

[29]:



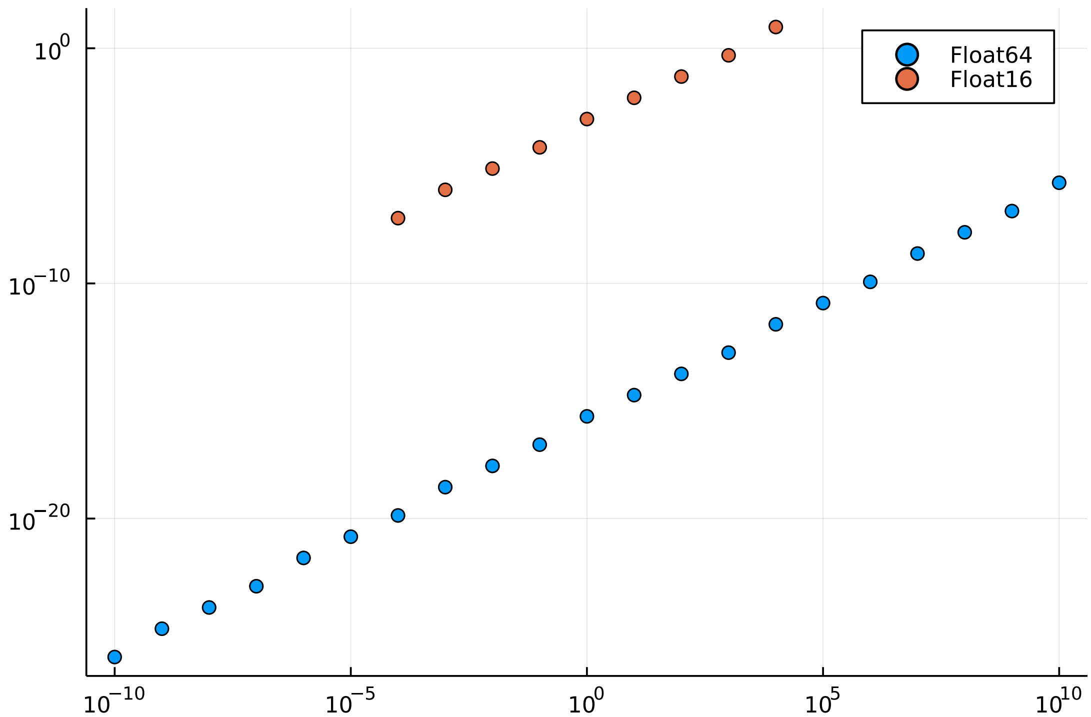
[30]:
```
eps(Float16(1))
```

[30]: Float16(0.000977)

[31]:
```
x16 = [Float16(10.0)^i for i   -4:4];
```

[32]:
```
scatter!(x16,eps.(x16)
    , yscale=:log10
    , xscale=:log10
    , label="Float16"
)
```

[32]:



3.1 Los límites de la representación:

El número finito más grande que podemos representar en Float64 es:

[33]:
```
prevfloat(Inf)
```

[33]:
```
1.7976931348623157e308
```

El más pequeño (en módulo):

[34]:
```
nextfloat(0.)
```

[34]:
```
5.0e-324
```

El primer número cercano a zero es un número denormalizado.

[35]:
```
bitstring(eps(0.0))
```

[35]:
```
"0000000000000000000000000000000000000000000000000000000000000001"
```

[36]:
```
bitstring(eps(Float32(0.0)))
```

[36]:
```
"00000000000000000000000000000001"
```

[37]:
```
1 + eps(1.)*0.6 == 1.
```

[37]: false

[38]:
```
bitstring(NaN)
```

[38]: "0111111111111000000000000000000000000000000000000000000000000000"

[39]:
```
bitstring(Inf)
```

[39]: "0111111111110000000000000000000000000000000000000000000000000000"

[ ]:
```

```