Problema 1.

Escriba un programa que determine los límites del *underflow* y *overflow* para Python (dentro de un factor 2) en su computadora.

Para determinar el límite del bajo flujo (*underflow*) se debe encontrar el número positivo de punto flotante más pequeño que Python no considere cero. Como se puede dar bajo flujo en flotantes negativos, también debe encontrarse el número negativo más grande que no es considerado cero.

La solución consiste en dividir sucesivamente un número inicial (digamos, `1`) entre `2`. Con esto, su tamaño disminuirá a la mitad en cada iteración, hasta volverse tan pequeño que sea prácticamente 0, alcanzando el límite del *underflow*.

En el caso del bajo flujo positivo, se emplea el siguiente algoritmo

1. Sea `aux_under = 1`
2. Mientras `aux_under` sea diferente de `0`:
    a. Guarda el valor de `aux_under` en `under`
    b. Divide `aux_under` entre `2`
3. Imprimir `under`

Al imprimir `under` tras finalizar el ciclo, se tiene el último valor que no es considerado cero. Como consecuencia de que se divide sucesivamente entre `2`, este valor está dentro de un factor de `2` del límite del underflow.

En el caso del límite del underflow negativo, el procedimiento es análogo excepto que se debe empezar con un valor negativo para `aux_under` (digamos, `-1`).

Para determinar el límite del sobre flujo (*overflow*), se debe encontrar el número positivo (negativo) más grande (más pequeño) que no sea considerado por Python como *infinito* (*infinito negativo*).

Para esto, la representación flotante del infinito en Python puede escribirse como `float(inf)`. La solución consiste en multiplicar sucesivamente un número inicial (digamos, `1`) por `2`. Con esto, su tamaño aumentará al doble por cada iteración, hasta volverse tan grande como `float(inf)`, alcanzando el límite del overflow.

En el caso del sobre flujo positivo se emplea el siguiente algoritmo

1. Sea `aux_over=1`
2. Mientras `aux_over` no sea `float(inf)`:
    a. Guarda el valor de `aux_over` en `over`
    b. Multiplica `aux_over` por `2`
3. Imprimir `over`

Al imprimir `over` tras terminar el ciclo, se tiene el último valor que no se considera infinito, dentro de un factor de `2`. En el caso del límite negativo del overflow, simplemente se debe comenzar con un valor negativo para `aux_over` (digamos, `-1`) y continuar mientras su valor sea distinto de `-float(inf)`.

Los resultados obtenidos para los límites de *overflow* y *underflow* se muestran en la Figura 1.

```
In [1]: runfile('/home/diego/Desktop/Fisica_Numerica/Tarea_1/
t1p1_overflow_underflow.py', wdir='/home/diego/Desktop/Fisica_Numerica/Tarea_1')
Límite de underflow positivo:
4.940656458412465e-324
Límite de underflow negativo:
-4.940656458412465e-324


Límite de overflow positivo:
8.98846567431158e+307
Límite de overflow negativo:
-8.98846567431158e+307
```

Figura 1. Resultados del programa `overflow_underflow.py`.

Las soluciones dadas para los límites de sobre y bajo flujo tienen la característica de estar dentro de un factor de 2 del que podría llamarse el límite real como consecuencia del uso de 2 como factor/divisor. Esta elección resulta adecuada pues, al elegir un factor más grande, se obtendrán resultados dentro de un factor igual de grande, resultando en una mayor incertidumbre de los límites reales (aunque, la convergencia a tal resultado podría ser más rápida).

Problema 2.

Escriba un programa y determine la *precisión de máquina* ϵ_m (dentro de un factor de 2) de su computadora.

La épsilon de la máquina ϵ_m se define como el máximo número positivo que puede sumarse al 1 (almacenado) tal que no se cambia su valor, es decir

`1.0 +` ϵ_m `= 1.0`

Para determinar este valor, se toma un presunto valor inicial de ϵ_m. Mientras el valor de ϵ_m sumado con 1.0 sea diferente a 1.0, se divide entre 2 el valor actual de ϵ_m, generando su nuevo valor. Este procedimiento se representa como

1. Sea `eps_m=1`
2. Mientras `(1.0+eps_m)` sea distinto de `1.0`
   a. Hacer `eps_m=eps_m/2`
3. Imprimir `eps_m`

Al terminar el ciclo, el valor de `eps_m` cumple la definición de ϵ_m dentro de un factor de `2` como consecuencia de dividir sucesivamente entre `2`. Los resultados del programa se muestran en la Figura 2, concluyendo que

$$\epsilon_m \approx 1.1102230246251565 \times 10^{-16}$$

Este método permite aproximar de una buena manera el ϵ_m. La elección de dividir entre 2 es adecuada pues dividir entre un número más grande haría que la aproximación obtenida esté dentro de un factor igual de grande (a pesar de converger más rápidamente a ϵ_m).

```
In [2]: runfile('/home/diego/Desktop/Fisica_Numerica/Tarea_1/t1p2_epsilon_maquina.py',
wdir='/home/diego/Desktop/Fisica_Numerica/Tarea_1')
eps_m                    1.0 + eps_m
------------------------------------------------
0.5                      1.5000000000000000
0.25                     1.2500000000000000
0.125                    1.1250000000000000
0.0625                   1.0625000000000000
0.03125                  1.0312500000000000
0.015625                 1.0156250000000000
0.0078125                1.0078125000000000
0.00390625               1.0039062500000000
0.001953125              1.0019531250000000
0.0009765625             1.0009765625000000
0.00048828125            1.0004882812500000
0.000244140625           1.0002441406250000
0.0001220703125          1.0001220703125000
6.103515625e-05          1.0000610351562500
3.0517578125e-05         1.0000305175781250
1.52587890625e-05        1.0000152587890625
7.62939453125e-06        1.0000076293945312
3.814697265625e-06       1.0000038146972656
1.9073486328125e-06      1.0000019073486328
9.5367431640625e-07      1.0000009536743164
4.76837158203125e-07     1.0000004768371582
2.384185791015625e-07    1.0000002384185791
1.192092895507812e-07    1.0000001192092896
5.960464477539062e-08    1.0000000596046448
2.980232238769531e-08    1.0000000298023224
1.490116119384766e-08    1.0000000149011612
7.450580596923828e-09    1.0000000074505806
3.725290298461914e-09    1.0000000037252903
1.862645149230957e-09    1.0000000018626451
9.313225746154785e-10    1.0000000009313226
4.656612873077393e-10    1.0000000004656613
2.328306436538696e-10    1.0000000002328306
1.164153218269348e-10    1.0000000001164153
5.820766091346741e-11    1.0000000000582077
2.91038304567337e-11     1.0000000000291038
1.455191522836685e-11    1.0000000000145519
7.275957614183426e-12    1.0000000000072760
3.637978807091713e-12    1.0000000000036380
1.818989403545856e-12    1.0000000000018190
9.094947017729282e-13    1.0000000000009095
4.547473508864641e-13    1.0000000000004547
2.273736754432321e-13    1.0000000000002274
1.13686837721616e-13     1.0000000000001137
5.684341886080801e-14    1.0000000000000568
2.842170943040401e-14    1.0000000000000284
1.4210854715202e-14      1.0000000000000142
7.105427357601002e-15    1.0000000000000071
3.552713678800501e-15    1.0000000000000036
1.77635683940025e-15     1.0000000000000018
8.881784197001252e-16    1.0000000000000009
4.440892098500626e-16    1.0000000000000004
2.220446049250313e-16    1.0000000000000002
1.110223024625157e-16    1.0000000000000000

Epsilón de máquina (eps_m):  1.110223024625157e-16
```

Figura 2. Resultados del programa `epsilon_maquina.py`.

Problema 3.

Considere la serie infinita para $\operatorname{sen} x$

$$\operatorname{sen} x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{x^{2n-1}}{(2x-1)!}$$

El problema consiste en desarrollar un programa que calcule $\operatorname{sen} x$ para $x < 2\pi$ y $x > 2\pi$, con un error absoluto menor a una parte en 10^8.

a) Escriba un programa que calcule $\operatorname{sen} x$. Presente los resultados en una tabla con títulos N, $suma$ y $|(suma - \sin x)/\sin x|$, donde $\sin x$ es la función correspondiente de Python. Note que la última columna es el error relativo de su cálculo. Realice el cálculo de la suma *inteligentemente* (sin factoriales) e inicie con una tolerancia (error absoluto) de 10^{-8}, compare con el error relativo.

Dada la serie infinita para $\operatorname{sen} x$, se puede escribir de la siguiente manera

$$\operatorname{sen} x = \sum_{n=1}^{\infty} a_n(x)\,, \qquad a_n(x) = (-1)^{n-1} \frac{x^{2n-1}}{(2x-1)!}$$

Para construir una relación de recurrencia entre los términos de la suma veamos que el término anterior inmediato está dado por

$$a_{n-1}(x) = (-1)^{n-2} \frac{x^{2n-3}}{(2n-3)!}$$

su cociente es

$$\frac{a_n(x)}{a_{n-1}(x)} = \frac{(-1)^{n-1} \dfrac{x^{2n-1}}{(2x-1)!}}{(-1)^{n-2} \dfrac{x^{2n-3}}{(2n-3)!}} = -\frac{x^2}{(2n-1)(2n-2)}$$

de donde se obtiene la relación de recurrencia para calcular los términos de la suma en función del término anterior, evitando el cómputo de potencias grandes de x directamente y de factoriales

$$a_n(x) = -\frac{x^2}{(2n-1)(2n-2)} a_{n-1}(x), \qquad a_1(x) = x, \qquad n = 2,3,4,\cdots$$

Entonces, considérese la aproximación de $\operatorname{sen} x$ a través de la suma de N términos, es decir

$$suma(N) \coloneqq \sum_{n=1}^{N} a_n(x) \quad \Longrightarrow \quad \operatorname{sen} x \approx suma(N)$$

El programa desarrollado, denominado `seno.py`, justamente implementa el cálculo de esta aproximación utilizando dos funciones:

- `a_n(n, a_ant)`
  Ésta implementa el cálculo del término a_n de la serie para $n = 2,3,4,\cdots$ dado el término anterior denotado por `a_ant` y dado el `n`.
- `suma(N)`
  Ésta computa la suma de los `N` términos comenzando por el $a_1 = x$ y utilizando sucesivamente a la función `a_n(n, a_ant)`.

Con esto, es posible calcular el error absoluto entre la aproximación (dada por `suma(N)`) y el valor dado por la función `sin(x)` de Python como

```
error_abs = abs( suma(N)-sin(x) )
```

el error relativo está dado por

```
error_rel = abs( error_abs / sin(x) )
```

La condición requisitada es que se realice el cálculo de `suma(N)` mientras `error_abs` sea mayor que un valor de tolerancia inicial de 10^{-8}. A lo largo de la ejecución, el programa imprime una tabla con el formato requisitado para visualizar el progreso de los cálculos, como se ve en la Figura 3.

```
In [5]: runfile('/home/diego/Desktop/Fisica_Numerica/Tarea_1/t1p3_seno.py', wdir='/
home/diego/Desktop/Fisica_Numerica/Tarea_1')
Valor de x:  2.356194490192345

N  Suma(N)             Error relativo
1  2.3561944901923448  2.332162203618775
2  0.1760656611087641  0.7510055541918007
3  0.7812315416655949  0.104828241577126
4  0.7012393831191733  0.008297753922722337
5  0.7074072812445046  0.0004249712574567038
6  0.7070959900908971  1.52609138217989e-05
7  0.7071070681697661  4.058555598081529e-07
8  0.7071067753047948  8.318054511715162e-09
```

Figura 3. Resultados del programa `seno.py`.

El hecho de utilizar la relación de recurrencia para el cálculo de los términos de la suma (aproximación a $\operatorname{sen} x$) permite evitar operaciones computacionales complicadas. Para distintos valores de x, la convergencia de la suma al valor dado por $\sin x$ (con la tolerancia 10^{-8}) es relativamente rápida y se completa al cabo de menos de 20 iteraciones en la mayoría de los casos. Solo se presenta el defecto en el cálculo de valores que hacen que $\sin x$ sea cero por la detección de una división entre cero en el cálculo del error relativo.

b) Utilice la identidad $\operatorname{sen}(x + 2n\pi) = \operatorname{sen} x$ para calcular $\operatorname{sen} x$ para valores grandes de x $(x > 2\pi)$.

Para utilizar la identidad de periodicidad, supongamos que se tiene un número x tal que $x > 2\pi$, más aún, tiene la forma $y + 2n\pi$ con $n \in \mathbb{N}$. Para reescalar su valor al intervalo $[0,2\pi)$, es decir, determinar el valor de y, es necesario sustraer de x un múltiplo entero de 2π adecuado; para ello, se determina el valor de n a través de la división entera de $\frac{x}{2\pi}$. Este algoritmo se escribe como

1. Sea `x` el valor a utilizar
2. Si `x>2*pi` entonces
    a. Calcule `v=x/(2*pi)` como división entera
    b. Hacer `x=x-v*(2*pi)`

Con esto, cualquier número mayor queda correctamente reescalado al intervalo $[0,2\pi)$. El desarrollo de esta modificación se realizó en un segundo programa titulado `seno_general.py` y sus resultados se muestran en la Figura 4.

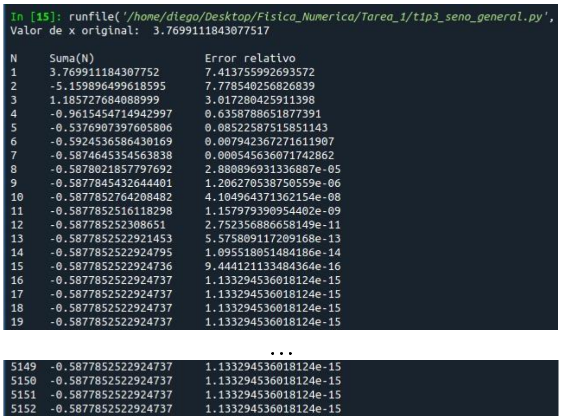
```
In [7]: runfile('/home/diego/Desktop/Fisica_Numerica/Tarea_1/t1p3_seno_general.py',
wdir='/home/diego/Desktop/Fisica_Numerica/Tarea_1')
Valor de x original:  60.73745796940267

Valor de x escalado:  4.188790204786393

N     Suma(N)                 Error relativo
1     4.188790204786393       5.836798304624577
2     -8.06060305162837       8.307582683376705
3     2.685767223410348       4.101256858833273
4     -1.803647522082058      1.082672764794553
5     -0.7096043731826172     0.1806194482497617
6     -0.8841138370591359     0.02088672364072887
7     -0.8644860379399629     0.001777506569379968
8     -0.8661259838664839     0.0001161398748866277
9     -0.8660201955165888     6.01398969144245e-06
10    -0.8660256228806626     2.529905265880198e-07
11    -0.8660253961465405     8.81948632063056e-09
```

Figura 4. Resultados del programa `seno_general.py`.

Como este programa solo implementa el reescalado de valores de x, el comportamiento del cálculo de las aproximaciones es el mismo que el del inciso anterior.

c) Ponga ahora su nivel de tolerancia menor a la precisión de máquina y vea cómo esto afecta su cálculo.

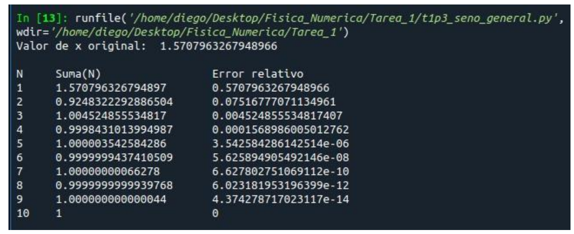
En el Problema 2 se determinó una aproximación de ϵ_m del orden de 1.1×10^{-16}, de modo que un valor de tolerancia menor es 5×10^{-17}, mismo que fue utilizado en el programa `seno_general.py` para este inciso. Los resultados obtenidos se muestran en las Figura 5.



```
In [15]: runfile('/home/diego/Desktop/Fisica_Numerica/Tarea_1/t1p3_seno_general.py',
Valor de x original:  3.7699111843077517

N     Suma(N)                Error relativo
1     3.769911184307752      7.413755992693572
2     -5.159896499618595     7.778540256826839
3     1.185727684088999      3.017280425911398
4     -0.9615454714942997    0.6358788651877391
5     -0.5376907397605806    0.08522587515851143
6     -0.5924536586430169    0.007942367271611907
7     -0.5874645354563838    0.000545636071742862
8     -0.5878021857797692    2.880896931336887e-05
9     -0.5877845432644401    1.206270538750559e-06
10    -0.5877852764208482    4.104964371362154e-08
11    -0.5877852516118298    1.157979390954402e-09
12    -0.587785252308651     2.752356886658149e-11
13    -0.5877852522921453    5.575809117209168e-13
14    -0.5877852522924795    1.095518051484186e-14
15    -0.5877852522924736    9.444121133484364e-16
16    -0.5877852522924737    1.133294536018124e-15
17    -0.5877852522924737    1.133294536018124e-15
18    -0.5877852522924737    1.133294536018124e-15
19    -0.5877852522924737    1.133294536018124e-15
```

...

```
5149  -0.5877852522924737    1.133294536018124e-15
5150  -0.5877852522924737    1.133294536018124e-15
5151  -0.5877852522924737    1.133294536018124e-15
5152  -0.5877852522924737    1.133294536018124e-15
```



```
In [13]: runfile('/home/diego/Desktop/Fisica_Numerica/Tarea_1/t1p3_seno_general.py',
wdir='/home/diego/Desktop/Fisica_Numerica/Tarea_1')
Valor de x original:  1.5707963267948966

N     Suma(N)                Error relativo
1     1.570796326794897      0.5707963267948966
2     0.9248322292886504     0.07516777071134961
3     1.004524855534817      0.004524855534817407
4     0.9998431013994987     0.0001568986005012762
5     1.000003542584286      3.542584286142514e-06
6     0.9999999437410509     5.625894905492146e-08
7     1.00000000066278       6.627802751069112e-10
8     0.9999999999939768     6.023181953196399e-12
9     1.000000000000044      4.374278717023117e-14
10    1                      0
```

Figura 5. Resultados del programa `seno_general.py` utilizando una tolerancia de 5×10^{-17} para distintos valores de x.

Para algunos valores de x, el programa continúa el cálculo de la aproximación por muchas iteraciones ya que no se logra satisfacer un error absoluto con el valor dado por $\sin x$ menor que la precisión de la máquina, es decir, en aproximadamente la cifra decimal 17 después del punto. Sin embargo, para otros valores de x, el comportamiento del programa es normal a excepción de que entrega un resultado más preciso (en términos del valor $\sin x$), alcanzando un error absoluto nulo.

Este comportamiento se puede atribuir a que, dada una tolerancia tan pequeña, a veces resulta imposible satisfacerla dado que la magnitud de términos calculados para la suma prácticamente no contribuye a cambiarlo pues pueden ser menores que ϵ_m y no alteran el valor del resultado almacenado.