Errores

Carlos Aguirre Maeso

Escuela Politécnica superior

Representación de números en coma flotante.

- El formato más habitual para la representación de los números en coma flotante es el IEEE 754 normalizado.
- Este formato permite números en precisión simple (32 bits) y en precisión doble (64bits).
- La forma general de un número en formato IEEE 754 es:

$$x = (-1)^s \, 1.f \; * 2^{e-bias}$$

- Los valores que se almacenan son, en este orden, s, e y f:
  - s es el signo del número
  - f es la mantisa sin el 1 inicial (nos ahorramos almacenar un bit)
  - e es el exponente
  - bias es un numero que nos permite representar valores pequeños. (127 en 32 bits y 1023 en 64 bits).

Representación de números en coma flotante.

- En los números en precisión simple (32 bits)
  - s 1 bit
  - e 8 bits
  - f 23 bits
- En los números en precisión doble (64 bits)
  - s 1 bit
  - e 11 bits
  - f 52 bits

Representación de números en coma flotante.

- Ejemplo: representación del número 53'2874 en IEEE 754 para coma flotante de 32 bit.
  - El número es positivo, s=0
  - 53'2874 = 110101'010010, f=00000000000010101010010
  - Hemos movido la coma n=5 posiciones a la izquierda, e=132=10000100

  La representación será (4 bytes)

  01000010 00000000 00000101 01010010

Representación de números en coma flotante.

- IEEE 754 nos permite representar algunos números especiales
- NAN (resultado de una operación imposible, como raíz de número negativo)
  - s=x (vale tanto 0 como 1)
  - e=11111111111
  - f= 1xxxxxxxx (vale cualquier número diferente de 0)
- INF y –INF (infinito y –infinito)
  - s=0 INF, 1 -INF
  - e=11111111111
  - f=0
- 0
  - s=0 +0, 1 -0
  - e=0
  - f=0

Representación de números en coma flotante.

- En la siguiente tabla tenéis todos los posibles casos a representar

Tipo	Signo (s)	Exponente (e)	Mantisa (f)	Valor decimal
Normal	0 ó 1	1<e<1023	Cualquiera	$(-1)^s\ 1.f * 2^{e-1023}$
Normal	0	e=1023	f=0	+∞
Normal	1	e=1023	f=0	-∞
Normal	0 ó 1	e=1023	f≠0	NaN
Normal	0 ó 1	e=0	f=0	0
Subnormal	0 ó 1	e=0	f≠0	$(-1)^s\ 0.f\ * 2^{-1022}$

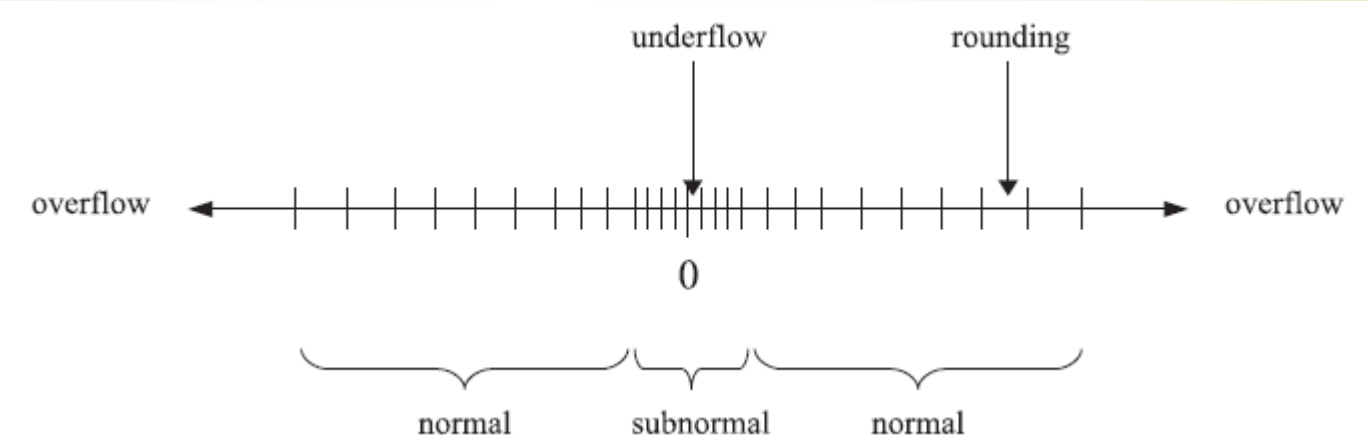
Representación de números en coma flotante.

- Ejercicio (5 mins)
  - Ejecutad Python, iPython o Jupyter Notebook.
  - Teclead los siguientes comandos
    - from sys import *
    - print(float_info.max)
    - print(float_info.min)
    - print(float_info)
  - Intendad averiguar que significa cada uno de los campos de la clase float_info.
  - Ejecutad ahora el siguiente código Python. ¿ que tipo de error se produce ?

```
large = 2.**1021
for i in range(3):
        large *= 2
        print(i, large)
```

Errores.

- A la vista de la representación anterior, observamos que:
  - Hay números que no se pueden representar.
    - **Overflow:** Por ser demasiado grandes en valor absoluto.
    - **Underflow:** Por ser demasiado pequeños en valor absoluto
    - **Redondeo:** Por no existir un representación binaria que tenga exactamente el mismo valor que el número.
  - El conjunto **A** (números de máquina) de números representables en un ordenador es finito.



Errores.

- Por tanto a la hora de representar un número es coma flotante es posible que introduzcamos un error (error de redondeo) al representarlo mediante un número que sí está en **A**.
- También es posible introducir errores a la hora de realizar un cálculo o una serie consecutiva de cálculos (error de aproximación), por ejemplo, la división de dos números que están en **A**, puede ocurrir que no esté en **A.**
- Finalmente es posible que introduzcamos errores también en la obtención de los datos (errores de medida, falta de precisión de dispositivos), estos errores se tratan de forma diferente, mediante algoritmos de filtrado.

Errores.

- El error que cometemos al representar un número $\boldsymbol{x}$ en coma flotante o el valor aproximado que obtenemos de $\boldsymbol{x}$ tras una serie de operaciones aritméticas $\tilde{\boldsymbol{x}}$ se mide de dos maneras
  - Error absoluto $e_{abs}(\boldsymbol{x}) = \tilde{\boldsymbol{x}} - \boldsymbol{x}$
  - Error relativo $e_{rel}(\boldsymbol{x}) = \frac{\tilde{\boldsymbol{x}} - \boldsymbol{x}}{\boldsymbol{x}}$
- En ocasiones, se toma como error absoluto y relativo el valor absoluto $|\ e_{abs}\ |$ y $|\ e_{rel}|$ de las cantidades anteriores.
- En ocasiones, el error relativo se da como porcentaje (es decir $e_{rel}\ *100\ \%$.)
- Ejercicio (5 mins): Calcular el error absoluto y el error relativo cuando aproximamos los valores. ¿ Cual de las dos es mejor aproximación ? ¿ Por qué ?

$\boldsymbol{x}$=0,999 mediante el número $\tilde{\boldsymbol{x}}$=1.0

$\boldsymbol{x}$=999 999 999 mediante el número $\tilde{\boldsymbol{x}}$=1000000000.0

Errores.

- En general nos gustaría que el error absoluto al aproximar un número estuviese acotado, es decir,
$$|e_{abs}(x)| = |\tilde{x} - x| \leq \varepsilon$$
- Desafortunadamente, no existe tal cota al representar los números reales mediante el conjunto **A** de números representables de forma exacta. No confundir ε con el valor epsilon de Python, que es solo una cota inferior de ese error.
- Lo que si es posible obtener en ocasiones es la cota $\boldsymbol{\varepsilon}$ que se tiene al calcular el valor aproximado $\tilde{x}$ de un cierto valor x que no conocemos.
- Si obtenemos la cota ε para un $\tilde{x}$ y x particulares entonces $-\varepsilon \leq x - \tilde{x} \leq \varepsilon$ y por tanto $\tilde{x} - \varepsilon \leq x \leq \tilde{x} + \varepsilon$

Errores.

- En realidad el error absoluto no nos da una buena idea de cuanto nos estamos equivocando realmente (no es lo mismo un error absoluto de 1 al aproximar 0.004, que al aproximar 434434543345).
- Operando en la fórmula del error relativo se tiene $\tilde{x} = x(1 + xe_{rel}(x))$
- Al igual que hicimos para el error absoluto se puede introducir una cota para el error relativo
- $|e_{rel}(x)| = |\tilde{x} - x|/|x| \leq \varepsilon$
- Donde ahora la cota si es más representativa de la cantidad de error cometido ya que no depende de lo grande que es el valor de x.

Errores en operaciones aritméticas, suma y resta.

- Consideremos la suma/resta de dos números $\mathbf{x} = \mathbf{a} \mp \mathbf{b}$
- Ambos números solo los podemos representar por sus aproximaciones en **A,** $\tilde{\boldsymbol{a}}$, $\tilde{\boldsymbol{b}}$, es decir, podemos obtener $\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{a}} - \tilde{\boldsymbol{b}}$ y como

$$\tilde{\boldsymbol{a}} = \boldsymbol{a} + e_{abs}(\boldsymbol{a}) \text{ y } \tilde{\boldsymbol{b}} = \mathbf{b} + e_{abs}(\mathbf{b})$$

$e_{abs}(\mathbf{x}) = \tilde{\boldsymbol{x}} - \boldsymbol{x} = (\tilde{\boldsymbol{a}} \mp \tilde{\boldsymbol{b}})$ - $\mathbf{x} = (\boldsymbol{a} + e_{abs}(\boldsymbol{a})) \mp (\tilde{\boldsymbol{b}} + e_{abs}(\mathbf{b})) - (\boldsymbol{a} \mp \boldsymbol{b})$

Con lo que operando, se tiene

$e_{abs}(\mathbf{x}) = e_{abs}(\boldsymbol{a}) \mp e_{abs}(\mathbf{b})$ y tomado valor absoluto

$$|e_{abs}(\boldsymbol{x})| \leq |e_{abs}(\boldsymbol{a})| + |e_{abs}(\boldsymbol{b})|$$

- La ecuación anterior dice que los errores absolutos se suman, lo cual, salvo casos excepcionales, son buenas noticias.

Errores en operaciones aritméticas, suma y resta.

- Para el error relativo e tiene

$$|e_{rel}(\boldsymbol{x})| = |\tilde{\boldsymbol{x}} - \boldsymbol{x}|/|\boldsymbol{x}| = |e_{abs}(\boldsymbol{x})|/|\boldsymbol{x}| \leq \frac{|e_{abs}(\boldsymbol{a})| + |e_{abs}(\boldsymbol{b})|}{|\boldsymbol{x}|}$$

Como $|e_{abs}(\boldsymbol{a})| = |e_{rel}(\boldsymbol{a})| * \boldsymbol{a}$ y $|e_{abs}(\boldsymbol{b})| = |e_{rel}(\boldsymbol{b})| * \boldsymbol{b}$ se tiene

$$|e_{rel}(\boldsymbol{x})| \leq \frac{|a|}{|\boldsymbol{a} \mp \boldsymbol{b}|} |e_{rel}(\boldsymbol{a})| + \frac{|a|}{|\boldsymbol{a} \mp \boldsymbol{b}|} |e_{rel}(\boldsymbol{b})|$$

Si además $a \approx b$

$$|e_{rel}(\boldsymbol{x})| \leq (|e_{rel}(\boldsymbol{a})| + |e_{rel}(\boldsymbol{b})|) \frac{|a|}{|\boldsymbol{a} \mp \boldsymbol{b}|}$$

- La situación anterior es especialmente mala en el caso de la resta cuando los números a y b están cerca. Esta situación se llama cancelación catastrófica.

Ejercicio (10 mins.) Calcular una cota para el error relativo al sumar los números a=1.253 y b=1.23 mediante las aproximaciones $\tilde{\boldsymbol{a}}$=1.25 y $\tilde{\boldsymbol{b}}$ = 1.2

Errores en operaciones aritméticas, producto.

- Consideremos el producto de dos números $\mathbf{x} = \mathbf{ab}$
- En este caso podemos obtener $\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{a}}\,\tilde{\boldsymbol{b}}$ y como

$$\tilde{\boldsymbol{a}} = \boldsymbol{a}(\mathbf{1} + e_{rel}(\boldsymbol{a})) \text{ y } \tilde{\boldsymbol{b}} = \mathbf{b}(\mathbf{1} + e_{rel}(\mathbf{b}))$$

- Se tiene

$$e_{rel}(\boldsymbol{x}) = \frac{\tilde{\boldsymbol{x}} - \boldsymbol{x}}{\boldsymbol{x}} = \frac{\tilde{\boldsymbol{a}}\,\tilde{\boldsymbol{b}} - \boldsymbol{ab}}{\boldsymbol{ab}} = \frac{(\boldsymbol{a}(\mathbf{1} + e_{rel}(\boldsymbol{a}))(\boldsymbol{b}(\mathbf{1} + e_{rel}(\boldsymbol{b})) - \boldsymbol{ab}}{\boldsymbol{ab}} =$$

$$\mathbf{1} + e_{rel}(\boldsymbol{a}) + e_{rel}(\boldsymbol{b}) + e_{rel}(\boldsymbol{a})e_{rel}(\boldsymbol{b}) - 1 \cong e_{rel}(\boldsymbol{a}) + e_{rel}(\boldsymbol{b})$$

- Y por tanto

$$|e_{rel}(\boldsymbol{x})| \leq |e_{rel}(\boldsymbol{a})| + |e_{rel}(\boldsymbol{b})|$$

Errores en operaciones aritméticas, funciones.

- Consideremos el cálculo de una función $\mathbf{y} = \mathrm{f}(\mathbf{x})$
- Se tiene $e_{abs}(\boldsymbol{y}) = \tilde{\boldsymbol{y}} - \boldsymbol{y} = \boldsymbol{f}(\tilde{\boldsymbol{x}}) - \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x} + e_{abs}(\boldsymbol{x})) - \boldsymbol{f}(\boldsymbol{x})$
- Por el Teorema de Taylor (orden 1)

  $e_{abs}(\boldsymbol{y}) = \boldsymbol{f}(\boldsymbol{x} + e_{abs}(\boldsymbol{x})) - \boldsymbol{f}(\boldsymbol{x}) \cong \frac{\boldsymbol{df(x)}}{\boldsymbol{dx}} e_{abs}(\boldsymbol{x})$
- El error relativo es fácil de calcular.
- $e_{rel}(\boldsymbol{y}) = \frac{e_{abs}(\boldsymbol{y})}{\boldsymbol{y}} = \frac{\boldsymbol{1}}{\boldsymbol{f(x)}} \frac{\boldsymbol{df(x)}}{\boldsymbol{dx}} e_{abs}(\boldsymbol{x}) = \frac{\boldsymbol{x}}{\boldsymbol{f(x)}} \frac{\boldsymbol{df(x)}}{\boldsymbol{dx}} \frac{e_{abs}(\boldsymbol{x})}{\boldsymbol{x}} = \frac{\boldsymbol{x}}{\boldsymbol{f(x)}} \frac{\boldsymbol{df(x)}}{\boldsymbol{dx}} e_{rel}(\boldsymbol{x})$
- Ejemplo: Sea $f(x) = x^3$

  Se tiene $e_{abs}(\boldsymbol{y}) \cong 3x^2 e_{abs}(\boldsymbol{x})$ y

  $e_{rel}(\boldsymbol{y}) \cong \frac{x}{x^3} 3x^2 e_{abs}(\boldsymbol{x}) = 3\, e_{rel}(\boldsymbol{x})$

Errores en operaciones aritméticas, funciones.

- Consideremos el cálculo de una función de varias variables $(y_1, y_2, \dots, y_k) = f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n), \dots, f_k(x_1, x_2, \dots, x_n)$
- Se tiene por el teorema de Taylor de orden 1 para funciones de varias variables

$$e_{abs}(y_i) \cong \sum_{j=1}^{n} \frac{\partial f_j(x_1, x_2, \dots, x_n)}{\partial x_j} e_{abs}(x_j)$$

- Y el error relativo

$$e_{rel}(y_i) \cong \sum_{j=1}^{n} \frac{x_j}{f_j(x_1, x_2, \dots, x_n)} \frac{\partial f_j(x_1, x_2, \dots, x_n)}{\partial x_j} e_{rel}(x_j)$$

- Observad que cualquier algoritmos puede ser considerado como la aplicación sucesiva de una función sobre un valor inicial.

Errores en operaciones aritméticas, funciones.

- Los números $e_{rel}(y_i)$ se denominan **números de condición** del algoritmo e indican como se transmite el error relativo al aplicar dicho algoritmo.
- Si estos números tienen un valor pequeño, se dice que el algoritmo está bien condicionado, por el contrario, si son grandes se dice que el problema está
- Ejemplo: Sea f(a,b,c)=a+b+c

$$e_{rel}(y) = \frac{a}{a+b+c} e_{rel}(a) + \frac{b}{a+b+c} e_{rel}(b) + \frac{c}{a+b+c} e_{rel}(c)$$

El problema estará bien condicionado si a,b,c << a+b+b

Errores en operaciones aritméticas, funciones.

- El error absoluto se puede representar de forma matricial

$$\begin{pmatrix} \boldsymbol{e_{abs}(y_1)} \\ \vdots \\ \boldsymbol{e_{abs}(y_k)} \end{pmatrix} \cong \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial x_1} & \cdots & \frac{\partial f_k}{\partial x_n} \end{pmatrix} \begin{pmatrix} e_{abs}(x_1) \\ \vdots \\ e_{abs}(x_n) \end{pmatrix}$$

- Es decir

$$\overrightarrow{e_{abs}}\,(\vec{y}) = D(f)\,\overrightarrow{e_{abs}}\,(\vec{x})$$

Donde D(f) es la matriz jacobiana de la función $= \vec{f} = f_1, f_2, \ldots, f_k$

Ejercicio (10 mins): Expresar el error relativo de forma matricial.

Precisión.

- Se define la precisión de una máquina (precisión, macheps, ε)

como en número mas pequeño que pertenece a **A** y es mayor que 1.

- (Ejercicio 5 mins) Ejecutad el siguiente código Python

```
small = 1/2**50
for i in range(5):
    small /= 2
    print(i, 1 + small, small)
```

- Probad también las siguientes instrucciones

```
1. + 2.3e-16
1. + 1.6e-16
1. + 1.12e-16
1. + 1.1e-16
```

- A la vista de lo anterior ¿Cuándo dirías que vale el épsilon de tu máquina ? ¿Coincide con el valor que daba **float_info** en el parámetro epsilon?

Precisión.

- Se define la precisión de una máquina (precisión, macheps, ε)

como en número mas pequeño que pertenece a **A** y es mayor que 1, es decir, el valor ε tal que (1+ ε) ≠ 1.

- (Ejercicio 5 mins) Vamos a calcular ε de forma experimental, ejecutad el siguiente código Python

```
small = 1/2**50
for i in range(5):
    small /= 2
    print(i, 1 + small, small)
```

- Probad también las siguientes instrucciones

```
1. + 2.3e-16
1. + 1.6e-16
1. + 1.12e-16
1. + 1.1e-16
```

- A la vista de lo anterior ¿Cuándo dirías que vale el épsilon de tu máquina ? ¿Coincide con el valor que daba **float_info** en el parámetro epsilon?

Precisión.

- Vamos a calcular el ε para máquinas de 64 bits.
- Observad que, para representar la suma anterior, necesitamos 1bit para representar el 1, con lo cual, nos quedarían 52 bits para representar la parte decimal.
- El número más pequeño, pero mayor de cero que podemos representar con 52 bits es $a = 1^{-52} = 2{,}2204460492503130\mathit{8e} - 16$
- Por tanto, en máquinas de 64 bits $\varepsilon = 2{,}2204460492503130\mathit{8e} - 16$

Cifras significativas.

- En IEEE 576 de 64 bits, tenemos 52 bits (en la práctica 53) bits para la mantisa.
- Supongamos que tenemos un número $x \in \boldsymbol{A}$. Dicho número lo podemos escribir $x = sign(x)a * 2^b$ con

$$a = 0.\alpha_1\alpha_2\ \alpha_3 \ldots.\ \alpha_{52}\ \alpha_{53}\ con\ \alpha_i = 0\ 1\ \text{y}\ \alpha_1 \neq 0$$

Es decir, tenemos 53 dígitos significativos en base 2.

- Por otro lado, el número entero más grande que podemos representar con 53 bits es $x = 2^{53} = 9.007.199.254.740.992$, que tiene 16 dígitos. Por tanto, cualquier número decimal de 15 dígitos (cuyo primer dígito es distinto de 0) y muchos (pero no todos) de 16 dígitos pueden ser representados de forma exacta.
- El número de dígitos que se pueden representar de forma exacta se denomina el número de cifras significativas **t**, en el caso de coma flotante de doble precisión t=15.

Ejercicio (2 mins): Calcular el número de cifras significativas **t** en los números en coma flotante de precisión simple.

Redondeo, error de redondeo.

- Supongamos que tenemos un número en base 10. Dicho número lo podemos escribir $x = sign(x)a * 10^b$ con

$$a = 0.\alpha_1\alpha_2\ \alpha_3 \ldots.\ \alpha_i\ \alpha_{i+1}\ \ldots. con\ 0 \leq \alpha_i \leq 9\ \text{y}\ \alpha_1 \neq 0$$

- Consideremos ahora que nuestra máquina es capaz de representar de forma exacta **t** dígitos, y construyamos el siguiente número (redondeo)

$$\tilde{a} = \begin{cases} 0.\alpha_1\alpha_2\ \alpha_3 \ldots.\ \alpha_t & si\ \alpha_{t+1} < 5 \\ 0.\alpha_1\alpha_2\ \alpha_3 \ldots.\ \alpha_t + 10^{-t} & si\ 5\ \geq \alpha_{t+1} \end{cases}$$

- Por tanto $\tilde{x} = sign(x)\tilde{a} * 10^b$
- Ahora tenemos

$$|e_{rel}(\boldsymbol{x})| = \frac{|\tilde{\boldsymbol{x}} - \boldsymbol{x}|}{|\boldsymbol{x}|} = \frac{|\tilde{\boldsymbol{a}} - \boldsymbol{a}|}{|\boldsymbol{a}|} \leq \frac{5 * 10^{-(t+1)}}{|\boldsymbol{a}|} \leq 5 * 10^{-t} ya\ que\ |a| \geq 10^{-1}$$

y por tanto

$$\tilde{x} = x(1 + r)\ donde\ |r|\ \leq 5 * 10^{-t}$$

Redondeo, error de redondeo.

- *Una construcción similar en base dos (Ejercicio 14) nos indica que* $|e_{rel}(x)| \leq 2^{-t}$ *y que por tanto r=ε=* 2^{-t}
- ***Es decir, el error relativo en números en forma binaria, es menor o igual que el épsilon de la máquina.***
- *En 64 bits el error relativo por truncamiento es aproximadamente*

$$\varepsilon = 2^{-t} \cong 2.220446049250313e-16$$

- *El valor anterior es una cota superior, por ejemplo, si* $x \in A$ *el error relativo será 0, ya que su representación será exacta.*

Comparaciones.

- *En Python (y en cualquier otro lenguaje), la comparación de dos números en realidad se realiza con una resta.*
- *Hemos visto que las restas son especialmente delicadas cuando ambos números están próximos.*
- *Por tanto, en números en coma flotante (incluso con pocos dígitos significativos) no suele ser buena idea comparar dos números de forma directa, sino que, es conveniente comprobar si la resta es menor que un cierto valor r.*

Comparaciones.

- *Ejecutad el siguiente código en Python*

```
xt = 0.1 + 0.2

yt = 0.3

print (xt == yt)
```

Comparaciones.

- *Bueno, lo que está pasando se puede ver fácilmente, añadid la siguiente línea a vuestro programa*

```
print (yt)
```

- La suma ha introducido un error en el resultado, y por tanto, la comparación es falsa.
- Probad ahora a ejecutar el siguiente código

```
xt = 0.1 + 0.2

yt = 0.3

print (abs(xt - yt)< 1.e-12)
```

Comparaciones.

- *Bueno, no del todo resuelto, ejecutad ahora este código*

```
xt = 1234567845439125434453 4
yt = 1234567845439125434453 4/34235437
print (abs(xt - 34235437*yt)< 1.e-12)
```

Comparaciones.

- *Probad a añadir ahora la siguiente línea de código a vuestro programa*

```
print(34235437*yt)
```

- *El problema es que hemos usado un épsilon absoluto, que no tiene en cuenta la magnitud de los números. La solución es usar un épsilon relativo. Probad ahora el siguiente código*

```
xt = 12345678454391254344534

yt = 12345678454391254344534/34235437

print (abs(xt-34235437*yt)/max(abs(xt),abs(34235437*yt)) < 1.e-12)
```

Problemas comunes.

- *Probad a ejecutar ahora el siguiente programa.*

```
print((0.7 + 0.1) + 0.3)

print(0.7 + (0.1 + 0.3))
```

o aun peor, ejecutad este

```
xt = 1.e20; yt = -1.e20; zt = 1.

print((xt + yt) + zt)

print(xt + (yt + zt)
```

- *El problema es que vosotros no sabéis si en vuestro programa os va a suceder alguna de las situaciones anteriores.*
- *Por eso es importante tener cuidado, y no dar por sentado los resultados que salgan de vuestros programas numéricos.*

Problemas comunes.

- *Para el problema de las sumas, hay una mejora, que consiste en ir haciendo una corrección cada vez que se añade un sumando.*
- *El método se llama suma compensada o suma de Kahan.*

```
def kahansum(xs):
    s = 0.; e = 0.
    for x in xs:
        temp = s
        y = x + e # realizo la compensacion
        s = temp + y
        e = (temp - s) + y  #calculo la nueva compensacion
    return s
```

Problemas comunes.

- *Probad el siguiente programa, y cambiad el orden de los números en el array, veréis que la suma funciona bien*

```
def kahansum(xs):
    s = 0.; e = 0.
    for x in xs:
        temp = s
        y = x + e #añado la compensacion
        s = temp + y
        e = (temp - s) + y  #calculo la nueva compensacion
    return s
xs = [0.7, 0.1, 0.3]
print(kahansum(xs))
```

Probad el siguiente programa, y cambiad el orden de los números en el array, veréis que pasa (moraleja: cuidado con las funciones de Python, no siempre son las mejores implementaciones).

```
xs = [0.7, 0.1, 0.3]
print(sum(xs))
```

Problemas comunes

- *Ejecutad ahora el siguiente programa, que calcula algunos valores de la función* $f(x) = \frac{1}{\sqrt{x^2+1}-x}$

```
from math import sqrt
def naiveval(x):
    return 1/(sqrt(x**2 + 1) - x)
xs = [10**i for i in range(4,8)]
ys = [naiveval(x) for x in xs]
for x, y in zip(xs, ys):
    print(x, y)
```

Problemas comunes

- *El problema es que el denominador de la función* $f(x) = \frac{1}{\sqrt{x^2+1}-x}$ *va a ser próximo a 0 (y eso ya sabemos que es malo).*
- *Una solución es darse cuenta que* $f(x) = \frac{1}{\sqrt{x^2+1}-x} = \sqrt{x^2+1}+x$
- *Si ahora ejecutais*

```
from math import sqrt
def naiveval(x):
    return sqrt(x**2 + 1) + x
xs = [10**i for i in range(4,8)]
ys = [naiveval(x) for x in xs]
for x, y in zip(xs, ys):
    print(x, y)
```

Problemas comunes

- *Otro ejemplo interesante es el cálculo del polinomio de McLaurin para la función exponencial.*

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} \dots$$

- *Aquí el problema viene por partida triple (de hecho, cuadruple),el cálculo del término $\frac{x^n}{n!}$, además de muy costoso (problema 1), puede ser conflictivo bien porque x^n puede ser grande muy rápido (problema 2), $\frac{1}{n!}$ puede ser pequeño muy rápido (problema 3), y el producto de un número muy grande por uno muy pequeño, también es problemático (problema 4).*
- *La solución es darse cuenta de lo siguiente*

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

Con lo cual, para cada nuevo sumando solo tengo que multiplicar el valor fijo x/n por el término anterior.