FCEIA - Universidad Nacional de Rosario
Métodos Numéricos - LCC 2020
Profesor: Alejandro G. Marchetti

Práctica 2: Errores Numéricos

Introducción

En sistemas de cómputo simbólico, como por ejemplo Maple [1], Mathematica [4] o Maxima [3], los cálculos se realizan utilizando símbolos matemáticos abstractos. Por lo tanto, no existe pérdida de precisión siempre y cuando no se realize la evaluación numérica. Por otra parte, en sistemas de cálculo numérico, tales como Scilab [6], Matlab [2], u Octave [5], los cálculos se llevan a cabo con números flotantes. Los valores numéricos almacenados en general poseen un error de redondeo.

Mientras que la matemática trata con números reales, los cálculos en computadora se realizan con sus representaciones en punto flotante. Los errores resultantes pueden en algunos casos ser disminuidos utilizando fórmulas adecuadas. En muchas situaciones, Scilab utiliza algoritmos específicamente diseñados para números en punto flotante [7].

Los números reales se almacenan en Scilab como variables de punto flotante de *doble precisión*. En efecto, Scilab utiliza la norma IEEE 754, por lo que los números se almacenan como números de punto flotante de 64 bits, llamados *dobles*. Denotamos $fl(x)$ al número en punto flotante asociado a un número real x.

Mientras que el conjunto de números reales es continuo y no acotado, los números en punto flotante son finitos y acotados. No todos los números reales tienen una representación en punto flotante. En efecto, existen infinitos números reales, pero solo un número finito de números en punto flotante. De hecho, existen 2^{64} números diferentes en punto flotante de 64 bits. Esto conlleva al redondeo, el desbordamiento a cero (underflow) y el desbordamiento (overflow).

Los números flotantes de precisión doble tienen un epsilon de la máquina aproximadamente igual a $2{,}22 \times 10^{-16}$. Este parámetro está almacenado en la variable `%eps` de Scilab:

```
-->%eps
%eps =

    2.220D-16
```

Por lo tanto, en el mejor de los casos podremos esperar tener aproximadamente 16 cifras significativas. El valor de `%eps` es siempre el mismo en Scilab, ya que Scilab usa IEEE de doble precisión, independientemente de la máquina que usemos, o de si se opera en Windows o Linux.

Los números negativos normalizados en punto flotante se encuentran en el rango $[-10^{308}, -10^{-307}]$, y los numeros positivos normalizados se encuentran en el rango $[10^{-307}, 10^{308}]$. Los límites dados en los intervalos anteriores son solo aproximaciones decimales. Cualquier número mayor que 10^{309} o menor que -10^{309} no es representable como un doble y se almacena como *infinito*: en este caso, se dice que ocurre un desbordamiento u overflow. Un número cuya magnitud es menor que 10^{-324} no es representable como un doble y se almacena como *cero*: en este caso, se dice que ocurre un desbordamiento a cero, o underflow.

Visualización de números en formato largo

A partir de Scilab 6.0.0, los dígitos irrelevantes que exceden la precisión relativa dada por `%eps` se visualizan como ceros:

```
-->format(25)
--> %pi
%pi =

    3.141592653589793100000
```

Mientras que en las versiones anteriores de Scilab obtenemos en pantalla:

```
%pi =

    3.141592653589793115998
```

Ejemplo 1: Errores de redondeo y en la evaluación de funciones elementales.

En la siguiente sesión de Scilab 6.0.0 calculamos el número real 0,1 de dos maneras diferentes, pero matemáticamente equivalentes.

```
-->format(25)
-->0.1
 ans =
  0.1
-->1.0 - 0.9
 ans =
  0.099999999999999780000
-->0.1 == 1.0 - 0.9
 ans =
 F
```

Aclaramos que en las versiones de Scilab anteriores a la versión 6.0 se obtenía:

```
-->0.1
    ans =
     0.10000000000000000055511
-->1.0 - 0.9
    ans =
     0.09999999999999999777955
```

A continuación comprobaremos que la función seno es también aproximada ya que el valor de $\sin(\pi)$ no es exactamente igual a cero.

```
-->format(25)
-->sin(0.0)
 ans =
  0.
-->sin(%pi)
 ans =
  0.00000000000000001224647
```

Explicación.

Vemos que el número decimal 0,1 no se representa en forma exacta. El error en los últimos dígitos es consecuencia de la conversión aproximada del número binario interno de doble precisión al número decimal que se muestra en pantalla.
Cualquier número x se puede representar en punto flotante como

$$fl(x) = M \cdot \beta^{E-s} \tag{1}$$

Scilab utiliza números en punto flotante IEEE de doble precisión, los cuales poseen una base $\beta = 2$, un exponente E de 11 dígitos, una mantisa M de 52 dígitos, y un sesgo $s = 1023$. El número 0,1 se almacena en binario con el siguiente exponente y mantisa:

$$\begin{aligned}(E)_2 &= 01111111011\\ (M)_2 &= 1{,}1001100110011001100110011001100110011001100110011010\end{aligned} \tag{2}$$

En la mantisa el uno anterior a la coma no se almacena. En decimal obtenemos $E = 1019$ y $M = 1{,}6$. Luego la representación en punto flotante del número decimal 0,1 es

$$fl(0{,}1) = 1{,}6 \cdot 2^{1019-1023} = 1{,}6 \cdot 2^{-4}$$

Verificando en Scilab, obtenemos:

```
-->1.6*2^(-4)
    ans =
     0.1
```

Vemos que en la mantisa aparece un patrón de pares de 11 y 00. En efecto, el valor verdadero de 0,1 se puede representar mediante la siguiente descomposición binaria infinita:

$$0{,}1 = \left(\frac{1}{2^0} + \frac{1}{2^1} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \ldots\right) \cdot 2^{-4}.$$

Se puede ver que la representación binaria de $x = 0{,}1$ está conformada por una sucesión infinita de dígitos, mientras que en la mantisa de doble precisión sólo se pueden representar 53 bits.
El número en punto flotante anterior a (2) tiene la siguiente mantisa:

$$(M)_2 = 1{,}1001100110011001100110011001100110011001100110011001 \tag{3}$$

donde se reeplazó el 10 en los últimos dos bits por 01.
En decimal obtenemos $M = 1{,}5999999999999999$. En Scilab obtenemos:

```
-->1.5999999999999999*2^(-4)
    ans =
     0.099999999999999920000
```

Vemos que el número exacto 0,1 se encuentra entre dos números de punto flotante consecutivos:

$$1{,}5999999999999999 \cdot 2^{-4} < 0{,}1 < 1{,}6 \cdot 2^{-4} \tag{4}$$

Por defecto, Scilab redondea al número en punto flotante más cercano. En nuestro caso, las distancias de $x = 0{,}1$ a los dos números en punto flotante son:

$$|0{,}1 - 1{,}5999999999999999 \cdot 2^{-4}| = 8{,}33\ldots 10^{-18},$$
$$|0{,}1 - 1{,}6000000000000000 \cdot 2^{-4}| = 5{,}55\ldots 10^{-18}.$$

El cálculo anterior no está realizado con Scilab sino con un sistema de computación simbólica (www.wolframalpha.com).

El número $x = 0{,}9$ tampoco se puede representar en forma exacta como un número binario en punto flotante. El error de representación de 0,9 es distinto al error de representación de 0,1. Cuándo al número 1,0 (que si se puede representar en forma exacta) le restamos $fl(0{,}9)$, obtenemos $fl(1{,}0) - fl(0{,}9) \neq fl(0{,}1)$.

La representación exacta del número π requiere de un número infinito de bits. La variable `%pi` $= fl(\pi)$ corresponde al número binario en punto flotante de doble precisión que mejor aproxima al número π. Además del error de representación de π, en la evaluación de `sin(%pi)` también hay error de aproximación de la función seno. En effecto, para evaluar la función seno la computadora utiliza un algoritmo interno basado en una aproximación polinomial de la función seno.

Ejemplo 2: Ruido en la evaluación de una función.

Una de las consecuencias inmediatas de los errores de redondeo es que la evaluación de una función $f(x)$ empleando una computadora resultará en una función aproximada $\hat{f}(x)$ que no es continua, si bien esto es aparente solamente si graficamos $\hat{f}(x)$ en una escala suficientemente pequeña. Habrá un error de redondeo en cada operación aritmética que se realiza para evaluar $f(x)$. Como efecto de dichos errores de redondeo, se obtiene una función $\hat{f}(x)$ cuyo error $f(x) - \hat{f}(x)$ se asemeja a un número aleatorio de pequeña magnitud. Este error en $\hat{f}(x)$ se conoce como el "ruido" en la evaluación de $f(x)$. Esto tiene consecuencias para otros programas que utilizan $\hat{f}(x)$. Por ejemplo, el cálculo de una raíz de $f(x)$ empleando $\hat{f}(x)$ resultará en una franja de incertidumbre en la ubicación de la raíz. Esto puede verse en el siguiente ejemplo, en el que compararemos dos polinomios algebraicamente equivalentes:

$$P_1(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$
$$P_2(x) = (x-1)^7$$

Como se puede ver al ejecutar el código, la segunda expresión es numéricamente más estable que la primera.

```
clc // limpia la consola
clear  // borra el contenido de la memoria

// Primera función
function y = P1(x)
 y = x.^7 - 7*x.^6 + 21*x.^5 - 35*x.^4 + 35*x.^3 - 21*x.^2 + 7*x - 1;
endfunction

// Segunda función
function y = P2(x)
 y = (x - 1).^7;
endfunction

// Evaluación de ambas funciones cerca de uno
x = linspace(1-1e-2,1+1e-2,2001);
y1 = P1(x);
y2 = P2(x);

// Gráfica de las funciones
plot(x,y1,'b');
plot(x,y2,'r','thickness',2)
legend(["$P1(x)$";"$P2(x)$"]);
```

Ejemplo 3: Ecuación cuadrática.

Sean $a, b, c \in \mathbb{R}$ coeficientes dados con $a \neq 0$. Consideremos la siguiente ecuación cuadrática:

$$ax^2 + bx + c = 0, \tag{5}$$

donde x representa la variable. El *discriminante* de la ecuación cuadrática se define como $\Delta = b^2 - 4ac$. El signo del discriminante determina la índole y la cantidad de raíces.

- Si $\Delta > 0$ hay dos raíces reales y diferentes:

$$x_- = \frac{-b - \sqrt{\Delta}}{2a}, \tag{6}$$

$$x_+ = \frac{-b + \sqrt{\Delta}}{2a}. \tag{7}$$

- Si $\Delta = 0$ hay una raíz real doble:

$$x_\pm = -\frac{b}{2a}. \tag{8}$$

- Si $\Delta < 0$ hay dos raíces complejas conjugadas:

$$x_\pm = \frac{-b}{2a} \pm i\frac{\sqrt{-\Delta}}{2a}. \tag{9}$$

La siguiente función de Scilab llamada `misraices` es una implementación directa de las fórmulas matemáticas anteriores. Toma como entrada los coeficientes de un polinomio cuadrático, almacenados en la variable vectorial `p`, y regresa las dos raíces en el vector `r`.

```
function r = misraices(p)
 c = coeff(p,0);
 b = coeff(p,1);
 a = coeff(p,2);
 r(1) = (-b + sqrt(b^2-4*a*c))/(2*a);
 r(2) = (-b - sqrt(b^2-4*a*c))/(2*a);
endfunction
```

Supresión de cifras significativas.

A continuación analizaremos los errores de redondeo que aparecen cuando el discriminante de la ecuación cuadrática es tal que $b^2 \gg 4ac$. Para ello consideremos la siguiente ecuación cuadrática:

$$\epsilon x^2 + (1/\epsilon)x - \epsilon = 0 \tag{10}$$

con $\epsilon > 0$. El discriminante de esta ecuación es $\Delta = 1/\epsilon^2 + 4\epsilon^2$. Las dos soluciones reales de la ecuación cuadrática son

$$x_- = \frac{-1/\epsilon - \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon}, \qquad x_+ = \frac{-1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon}. \tag{11}$$

Nos interesa principalmente estudiar el caso en que la magnitud de ϵ es muy pequeña. Se puede demostrar que para ϵ cercano a cero, las raíces se pueden aproximar por

$$x_- \approx -1/\epsilon^2, \qquad x_+ \approx \epsilon^2. \tag{12}$$

En el siguiente script de Scilab compararemos los valores de las raíces calculadas por nuestra función `misraices`, con las raíces calculadas por la función `roots` de Scilab. Solo verificamos la raíz positiva $x_+ \approx \epsilon^2$, ya que no hay diferencias apreciables en el cálculo de la segunda raíz por ambas funciones. Consideraremos el caso particular $\epsilon = 0{,}0001 = 10^{-4}$. Por empezar, definimos un polinomio por sus coeficientes mediante la función `poly` de Scilab. Asignamos a la variable `e1` el valor esperado de la raíz positiva $x_+ = \epsilon^2$. Luego calculamos las raíces `r1` y `r2` con las dos funciones `misraices` y `roots`. Finalmente, calculamos los errores relativos `error1` y `error2`.

```
p = poly([-0.0001 10000.0 0.0001],"x","coeff");
e1 = 1e-8;
roots1 = misraices(p);
r1 = roots1(1);
roots2 = roots(p);
r2 = roots2(2);
error1 = abs(r1-e1)/e1;
error2 = abs(r2-e1)/e1;
printf("Esperado :%e\n", e1);
printf("misraices (nuestro) :%e (error=%e)\n", r1, error1);
printf("roots (Scilab) :%e (error=%e)\n", r2, error2);
```

El script anterior produce la siguiente salida.

```
Esperado : 1.000000e-08
misraices (nuestro) : 9.094947e-09 (error = 9.050530e-02)
roots (Scilab) : 1.000000e-08 (error = 0.000000e+00)
```

Como vemos, la aplicación directa de la fórmula clásica para obtener las raíces de la ecuación cuadrática produce una raíz sin ninguna cifra significativa, y con un error relativo por lo menos 14 órdenes de magnitud mayor que el error relativo de la raíz calculada por Scilab (ya que el error relativo de la raíz calculada por Scilab es por lo menos menor a `%eps`).

Este comportamiento se explica al analizar cómo se calcula la raíz positiva x_+ empleando la fórmula de la ecuación (7). Primero consideremos el cálculo del discriminante $\Delta = 1/\epsilon^2 + 4\epsilon^2$. El término $1/\epsilon^2$ es igual a 100000000 y el término $4\epsilon^2$ es igual a 0,00000004. La suma de ambos términos en Scilab es igual a 100000000,000000045. Por lo tanto, la raíz cuadrada del discriminante es

$$\sqrt{1/\epsilon^2 + 4\epsilon^2} = 10000{,}000000000001818989.$$

Vemos que los primeros dígitos son correctos, pero los últimos dígitos están sujetos a errores de redondeo. Luego, al evaluar la expresión $-1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2}$ se realizan los siguientes cálculos:

$$\begin{aligned} -1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2} &= -10000{,}0 + 10000{,}000000000001818989. \\ &= 0{,}0000000000018189894035. \end{aligned}$$

Podemos ver que el resultado está dominado por la supresión de dígitos significativos. Si se disminuye aún más el valor de ϵ, el error en la evaluación de la raíz usando la fórmula clásica empeora.

Método robusto para calcular las raíces.

Es evidente que la función `roots` de Scilab utiliza un método distinto para calcular las raíces de un polinomio cuadrático. Presentaremos a continuación un método robusto para el cálculo de dichas raíces. Éste método es utilizado por Scilab para el caso de polinomios cuadráticos con discriminante positivo.

Supongamos que la ecuación cuadrática (5), con coeficientes reales $a, b, c \in \mathbb{R}$ y $a \neq 0$, tiene un discriminante Δ positivo. Luego, las dos raíces reales están dadas por las ecuaciones (6) y (7). Dividiendo la ecuación cuadrática (5) por $1/x^2$, suponiendo $x \neq 0$, obtenemos

$$c(1/x)^2 + b(1/x) + a = 0 \tag{13}$$

Las dos raíces reales de la ecuación cuadrática (13) son

$$x_- = \frac{2c}{-b + \sqrt{b^2 - 4ac}}, \tag{14}$$

$$x_+ = \frac{2c}{-b - \sqrt{b^2 - 4ac}}. \tag{15}$$

Las expresiones (14) y (15) también se pueden derivar directamente a partir de las ecuaciones (6) y (7) multiplicando sus numeradores y denominadores por $-b + \sqrt{b^2 - 4ac}$.

Cuando el discriminante Δ es positivo, el problema de supresión de dígitos significativos se puede separar en dos casos:

- Si $b < 0$, luego $-b - \sqrt{b^2 - 4ac}$ puede dar supresión de dígitos ya que $-b$ es positivo y $-\sqrt{b^2 - 4ac}$ es negativo,
- Si $b > 0$, luego $-b + \sqrt{b^2 - 4ac}$ puede dar supresión de dígitos ya que $-b$ es negativo y $\sqrt{b^2 - 4ac}$ es positivo.

Por lo tanto,

- Si $b < 0$ debemos usar la expresión $-b + \sqrt{b^2 - 4ac}$,
- Si $b > 0$ debemos usar la expresión $-b - \sqrt{b^2 - 4ac}$.

La solución consiste en una combinación de las expresiones de las raíces dadas por un lado por las ecuaciones (6) y (7), y por otro lado por las ecuaciones (14) y (15). La siguiente elección permite solucionar el problema de supresión de dígitos significativos:

- Si $b < 0$ calcular x_- a partir de (14) y x_+ a partir de (7),
- Si $b > 0$ calcular x_- a partir de (6) y x_+ a partir de (15).

Ejercicio 1.

Crear una función en Scilab que calcule en forma robusta las raíces de una ecuación cuadrática con discriminante positivo. Usar dicha función para evaluar la raíz positiva de la ecuación cuadrática (10) con $\epsilon = 0{,}0001$ y estimar su error.

Ejemplo 4: Derivada numérica.

En este ejemplo vamos a analizar el cómputo de la derivada numérica de una función. Sea $x \in \mathbb{R}$ un número y $h \in \mathbb{R}$ un paso dado. Suponga que $f : \mathbb{R} \to \mathbb{R}$ es una función dos veces continuamente diferenciable. A partir de una expansión de Taylor de segundo orden, obtenemos

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \mathcal{O}(h^3). \tag{16}$$

Despejando $f'(x)$ obtenemos

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \frac{h}{2}f''(x) + \mathcal{O}(h^2). \tag{17}$$

Para aproximar $f'(x)$ podemos utilizar la expresión en diferencias finitas de primer orden:

$$f'(x) \approxeq \frac{f(x+h) - f(x)}{h}. \tag{18}$$

Por el concepto de límite, sabemos que cuando h tiende a cero, la fórmula nos da el valor exacto de la derivada $f'(x)$. Por lo tanto, intuitivamente podemos esperar que cuanto más pequeño sea h mejor será la aproximación. Sin embargo, debido a los errores de redondeo esto no es así.

En el siguiente script de Scilab consideraremos la función $f(x) = x^2$ cuya derivada es $f'(x) = 2x$. Estimaremos por diferencias finitas de primer orden la derivada en el punto $x = 1$ variando h del siguiente modo: $h = 10^0,\ 10^{-1},\ 10^{-2},\ \ldots,\ 10^{-16}$.

La función `numderivative` de Scilab permite calcular la matriz jacobiana y la matriz hesiana de una función dada. Además, permite usar fórmulas de diferencias finitas de órdenes 1, 2, o 4. La fórmula de orden 1 es la fórmula dada por la ecuación (18). En el scipt analizaremos el error relativo obtenido en función del paso h, y lo compararemos con el error relativo obtenido con la función `numderivative` usando su paso por defecto.

```
clc // limpia la consola
clear  // borra el contenido de la memoria
xdel(winsid())  // cierra ventanas gráficas

// Definición de la función
function y = f(x)
  y = x.*x;
endfunction

// Cálculo de la derivada utilizando diferencias finitas
function y = dfa(f,x,h)
  y = (f(x+h) - f(x))./h;
endfunction

x = 1; // Punto donde vamos a evaluar la derivada
ih = (0:16)';
h = (10.^-ih); // Vector con los valores de h

df_approx = dfa(f,x,h); // Evaluación de la derivada por diferencias finitas
df_scilab = numderivative(f,x,[],order=1); // Derivada obtenida por numderivative
df_true = 2; // Valor verdadero de la derivada en x = 1

// Errores absolutos y relativos
err_abs = abs(df_approx - df_true);
err_rel = err_abs./abs(df_true);
err_abs_sci = abs(df_scilab - df_true);
err_rel_sci = err_abs_sci/abs(df_true);

// Gráfica
plot(ih,log10(err_rel),'b*-'); // Gráfica en escala logarítmica en el eje y
title('Error relativo utilizando diferencias finitas');
xlabel('i');
ylabel('$log_{10} (Err Rel)$');
plot(ih,log10(err_rel_sci*ones(length(ih),1)),'r-');

// Impresión de resultados en pantalla
tablevalue = [ih,h,df_true*ones(length(h),1),df_approx,err_abs,err_rel];
mprintf('%s\n',strcat(repmat('-',1,80)));
mprintf('%4s %8s %12s %18s %14s %14s\n',...
   'i', 'h','Der. exact','Der approx','Abs. error','Rel. error');
mprintf('%s\n',strcat(repmat('-',1,80)));
mprintf('%4d %8.1e %9.6e %18.10e %14.5e %14.5e\n',tablevalue);
mprintf('%s\n',strcat(repmat('-',1,80)));
mprintf('%4.1s %8s %9.6e %18.10e %14.5e %14.5e\n',...
   ' ', 'Scilab',[df_true,df_scilab,err_abs_sci,err_rel_sci]);
mprintf('%s\n',strcat(repmat('-',1,80)));
```

Ejercicio 2

Usando aritmética de cuatro dígitos de precisión (mantiza decimal de 4 dígitos con redondeo), sume la siguiente expresión

$$0,9222 \cdot 10^4 + 0,9123 \cdot 10^3 + 0,3244 \cdot 10^3 + 0{,}2849 \cdot 10^3$$

tanto ordenando los números de mayor a menor (en valor absoluto), como de menor a mayor. Realiza cada operación de forma separada, primero igualando exponentes y luego normalizando el resultado en cada paso.
¿Cuál de las dos posibilidades es más exacta? Justifique los resultados que encuentre.

Ejercicio 3

El algoritmo de Horner se usa para evaluar de forma eficiente funciones polinómicas. Dado un polinomio $p(x) = a_0 + a_1 x + \ldots + a_n x^n$ a coeficientes reales, se genera una secuencia de constantes dadas por:

$$\begin{aligned} b_n &= a_n \\ b_i &= a_i + x_0 b_{i+1}, \quad i = (n-1),\ \ldots,\ 1,\ 0 \end{aligned}$$

donde $b_0 = p(x_0)$.

a) Mostrar que dado un x_0, $b_0 = p(x_0)$.

b) Implementar el algoritmo de Horner para calcular $p(x_0)$ en Scilab.

c) Dado $q(x) = b_1 + b_2 x + \ldots + b_n x^{n-1}$, mostrar que $p'(x_0) = q(x_0)$. Verificar que $p(x) = b_0 + (x - x_0)q(x)$.

d) Implementar una generalización del algoritmo `horner` dado en Scilab, de tal forma que se pueda calcular $p(x_0)$ y $p'(x_0)$ al mismo tiempo.

Ejercicio 4

Implementar en Scilab la función `derivar` que toma una función f, un valor v, un orden n y un paso h y retorna el valor de evaluar la derivada de f de orden n en el punto v. Implementar usando cociente incremental y luego usando el comando `numderivative` implementado en Scilab.

a) ¿Cómo son los errores cometidos en cada caso?

b) ¿Qué hace que el error en la implementación por cociente incremental crezca?

Ejercicio 5

Implementar en Scilab una función `taylor` que calcule el valor de un polinomio de Taylor de grado n de una función f en un punto dado v.

Ejercicio 6

a) Verificar que la aproximación por polinomio de Taylor de grado 10 de la función e^x permite obtener el valor correcto de e^{-2} redondeado a tres dígitos (utilizando la precisión de Scilab).

b) Utilizar el polinomio de Taylor de grado 10 de la función e^x para aproximar

a) e^{-2} b) $1/e^2$

empleando aritmética de punto flotante con redondeo a tres dígitos decimales. El valor correcto en 3 dígitos es 0,135. ¿Cuál es la mejor aproximación y por qué?

Referencias

[1] Maple. Maplesoft. http://www.maplesoft.com.

[2] The MathWorks. Matlab. http://www.mathworks.com.

[3] Maxima. http://maxima.sourceforge.net.

[4] Wolfram Research. Wolfram alpha. http://www.wolframalpha.com.

[5] Octave. http://www.gnu.org/software/octave.

[6] The Scilab Consortium. Scilab. http://www.scilab.org.

[7] Michaël Baudin. Scilab is not naive. The Scilab Consortium - Digiteo, 2010.