OBLIGATORIO 1

MÉTODOS NUMÉRICOS

Curso 2017.

DMEL, CenUR LN

Universidad de la Republica

UdelaR.

Profesor: José Vieitez

Sebastián Castro

Grupo:

- Juan Ferrand
- Luciano Acosta

Contenido

Objetivo:

El objetivo del presente trabajo es implementar algoritmos que permitan calcular y^{α} donde y > 0 y $\alpha \epsilon R$ comparándolos con los que brinda Octave.

Ejercicio N° 1

Demostrar que podemos escribir y^{α} como $y^{n} * y^{\beta}$ con $n \epsilon \mathbb{Z}$ y $0 \leq \beta < 1$.

Dem: Para la demostración se consideran 2 casos:

1- $\alpha \epsilon \mathbb{Z}$
2- $\alpha \epsilon \mathbb{R}^{+}$

1- Si $\alpha \epsilon \mathbb{Z}$: tenemos que β está forzada a ser 0 porque $y^{\alpha} = y^{n} * y^{\beta}$, obteniendo que $\alpha = n + 0$ y que $y^{\alpha} = y^{n+0}$

$\rightarrow (utilisando\ propiedades\ de\ potencia) \rightarrow y^{n+0} = y^{n} * y^{0} \rightarrow y^{\alpha} = y^{n} * 1 = y^{n}$

2- Si $\alpha \epsilon \mathbb{R}^{+}$: sabemos que α se puede escribir como la suma de un entero y un decimal;

$$\alpha = n + \beta \rightarrow y^{\alpha} = y^{n+\beta}$$

$$\rightarrow (utilisando\ propiedades\ de\ potencia) \rightarrow y^{n+\beta} = y^{n} * y^{\beta} = y^{\alpha}$$

a) Para $n \geq 0$. Implementar un Módulo que escriba *n* en binario:

El programa que implementamos se llama "**int_a_binario**", recibe de entrada un natural *"a"* y devuelve un vector "*bin"* que es la representación de dicho natural en base

Al comienzo del programa compara si el número recibido es 0, en caso afirmativo devuelve como resultado "0". Si la entrada es distinta de 0, primero se calcula la cantidad de dígitos que serán necesarios para representar este número en base 2 mediante la siguiente ecuación:

$$size = fix(log2(a)) + 1$$

Luego se genera un vector de tamaño "size", este vector se completa mediante un for que en cada iteración divide el número entre dos y guarda el resto desde la última posición hasta la primera. *(ver "**int_a_binario**" en Anexos)*

b) Implementar un módulo que calcule y^n usando el método de multiplicación por duplicación o método egipcio:

El módulo utilizado tiene el nombre de "**exp_egipcia**", este programa tiene como datos de entrada la base "*a*" el exponente *"b*" y da como resultado a^b.

Primero contemplamos los casos triviales:

- 1- Cuando la base es "0" y el exponente cualquier número real retorna "0".
- 2- Cuando el exponente es "0" y la base es cualquier número distinto de "0" retorna "1".

Luego, lo primero que hace la función es pedir la representación binaria del exponente mediante nuestro primer Módulo "**int_a_binario**" guardándola en un vector de nombre "*bin".*Después guardamos el tamaño del vector "*bin*" en la variable *"size"* e inicializamos una nueva variable *"res" = 1.*

Mediante un for loop que varía desde 1 hasta "*size",* se compara en cada entrada si el valor guardado es 1, en caso que esta condición se cumpla, multiplicamos la variable "*res*" por la variable "*a*". antes de finalizar el loop actualizamos el valor de "*a"* multiplicándola por sí misma, de esta forma mantenemos siempre la base elevada a una potencia de dos. *(ver "**exp_egipcia**" en Anexos).*

c) Compare el número de operaciones usado con respecto al método usual:

$$y^n = y * y * y * \ldots * y$$

n veces

- En el método tradicional para el cálculo de y^n el número de operaciones empleadas es *n-1* multiplicaciones.

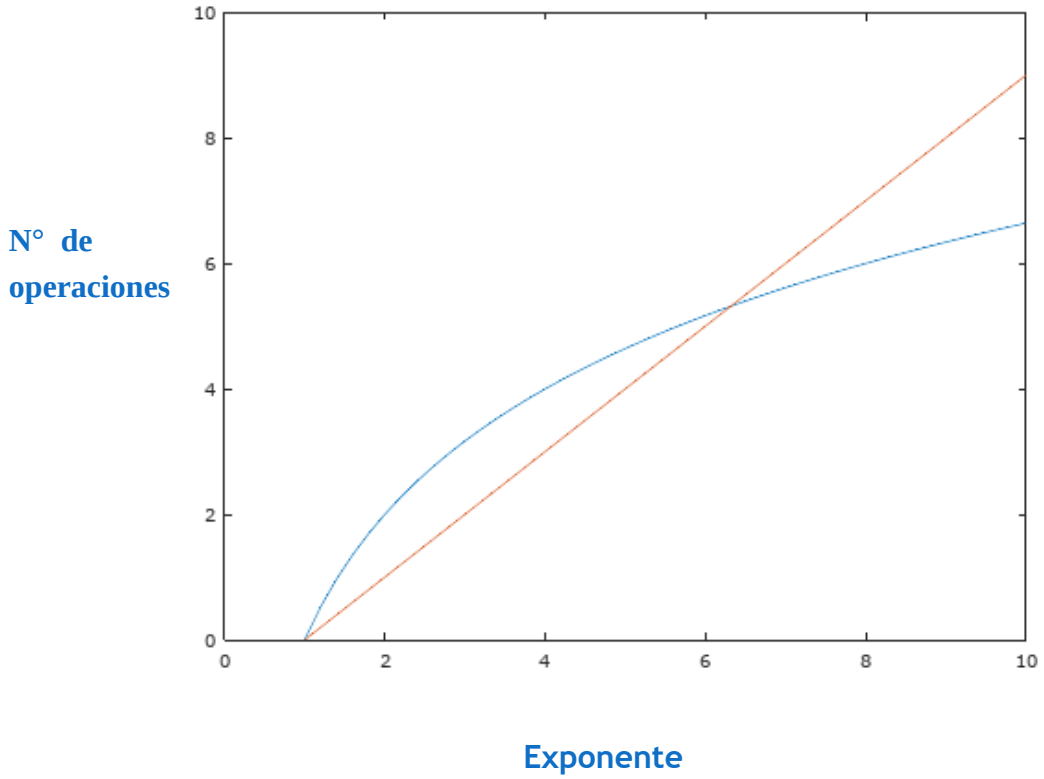
- En el método mediante la implementación de la multiplicación egipcia, el número de operaciones empleadas se puede calcular de la siguiente manera:

$$y^n = y^1 * y^2 * y^4 * \ldots * y^{log2(n)}$$

Debido a que la cantidad de términos de la multiplicación proviene de la descomposición en base 2 del exponente n (descomposición binaria), tenemos *log2(n)+1* términos, lo que se traduce en *log2(n)* multiplicaciones.

También sabemos que los y^i se calcula como $y^i = y^{i-1} * y^{i-1}$ para todo i diferente que 1, dando como resultado *log2(n)* operaciones para calcular todas las potencias de 2.

Podemos concluir que el número de operaciones utilizadas para el cálculo de y^n mediante este Método es ***2*log2(n)***. En la siguiente grafica se puede observar la diferencia mencionada. La línea azul representa $2 * log_2(exponente)$ y la línea roja representa el número de cuentas que utiliza el método usual ($exponente - 1$).



d) Experimente con y = 1.0001 y n = 1000; 10000; 100000; 1000000. A que se parece y^n cuando n = 10000?

n	exp_egipcia(y,n)	y^n	Diferencia
1000	1.105165393	1.10516539	0
10000	2.718145927	2.71814593	0
100000	22015.45605	2.20E+04	0
1000000	2.67E+43	2.67E+43	0
y= 1.0001			

Tabla Nº1: Comparación de Valores calculados con y^n y exp_egipcia

En la tabla se aprecia que los resultados obtenidos con el algoritmo predefinido de octava para calcular y^n y los resultados derivados de nuestro modulo "**exp_egipcia**" son muy similares, tanto así que al calcular la diferencia existente entre ellos obtenemos como resultado *"0"*. Consideramos que este fenómeno se debe a la cancelación catastrófica, es decir, al restar dos números muy próximos (muchas cifras significativas de igual valor), se pierden las cifras que la computadora no haya podido guardar.

Cuando *n = 10000*, se observa que el valor obtenido es muy cercano al número "e", tanto calculando de forma predeterminada con octave, como con nuestro programa.

Ejercicio N° 2

a) Recurriendo, si es necesario, a los apuntes sobre Taylor en la página EVA del curso de Métodos Numéricos escribir el desarrollo de Taylor de $(1+x)^{\alpha}$ hasta el orden n, colocando el resto en la forma de Lagrange y en la forma de Cauchy.

Desarrollo de Taylor de $y = (1+x)^{\alpha}$ Con $x, x_0 \epsilon [-1, 1]$

$$T_{x_0,k}(x) = (1+x_0)^{\alpha} + \alpha(1+x_0)^{\alpha-1}(x-x_0) + \frac{\alpha(\alpha-1)(1+x_0)^{\alpha-2}(x-x_0)^2}{2!} + \frac{\alpha(\alpha-1)(\alpha-2)(1+x_0)^{\alpha-3}(x-x_0)^3}{3!} + \cdots + \frac{\alpha!\,(\alpha-1)\ldots(\alpha-k+1)(1+x_0)^{\alpha-k}(x-x_0)^k}{k!} + R(x)$$

$$\text{Con } R(x) = f(x) - T_{x_0,n}(x)$$

El resto de LaGrange:

$$R(x) = \frac{\alpha(\alpha-1)\ldots(\alpha-k)(1+x_0+\theta h)^{\alpha-k-1}(h)^{k+1}}{(k+1)!} \; con\; \theta \epsilon (0,1);\; h = x - x_0\,;$$

El resto de Cauchy:

$$R_k(x) = \frac{\alpha(\alpha-1)\ldots(\alpha-k)\,(1+x_0+\theta h)^{\alpha-k-1}\,(h)^{k+1}}{(k)!}(1-\theta)^k \; con\; \theta \epsilon (0,1)\,;\; h = x - x_0\,;$$

b) Muestre que al usar el desarrollo de orden *k* de Taylor, el resto es siempre menor que $\frac{1}{2^n}$:

Para probar que $|R(x)| < \frac{1}{2^n}$ utilizaremos el desarrollo de Mac Laurin y luego consideraremos los casos diferente de θ; el primer caso es cuando $\theta \epsilon \left(0, \frac{1}{2}\right]$ y el segundo caso es cuando $\theta \epsilon [\frac{1}{2},\ 1)$

El desarrollo de Mac Laurin de $y = (1+x)^{\beta}$

$$T_{0,k}(x) = 1 + \beta x + \frac{\beta(\beta-1)x^2}{2} + \cdots + \frac{\beta(\beta-1)(\beta-2)\ldots(\beta-k+1)x^k}{k!} + R(x)$$

con $x \epsilon [-1, 1]$ y $x_0 = 0$; $\beta \epsilon (0, 1)$

Para demostrar el primer caso utilizaremos el Resto de LaGrange

con $x \epsilon [-1, 1]$; $\theta \epsilon (0\ ,\ \frac{1}{2}]$

$$R(x) = \frac{\beta(\beta-1)\ldots(\beta-k)(\theta x)^{k+1}}{(k+1)!}$$

$$|R(x)| = \left|\frac{\beta(\beta-1)\ldots(\beta-k)(\theta x)^{k+1}}{(k+1)!}\right| = \frac{|\beta(\beta-1)\ldots(\beta-k)|\ |(\theta x)^{k+1}|}{|(k+1)!|} \rightarrow$$

Sabemos que:

$(k+1)! > 0 \rightarrow |(k+1)!| = (k+1)!$

$|\beta(\beta-1)(\beta-2)\ldots(\beta-k)| < \beta k!$ (*Por sugerencia de los apuntes*)

$\rightarrow \frac{|\beta(\beta-1)\ldots(\beta-k)|\ |(\theta x)^{k+1}|}{|(k+1)!|} < \frac{\beta k!\ |(\theta x)^{k+1}|}{(k+1)!} \rightarrow$ Utilizando que $\frac{\beta k!}{(k+1)!} < 1$

$$\rightarrow \frac{\beta k!\ \ |(\theta x)^{k+1}|}{(k+1)!} < |(\theta x)^{k+1}|$$

Como $x \epsilon [-1\ , 1]$ y $\theta \epsilon (0, 1/2]$ tenemos que $\theta x \ \leq 1/2$

$|(\theta x)^{k+1}| < 1/2^k$

Llegando a la conclusión que $|R(x)| < 1/2^k$ para $\theta \epsilon (0, 1/2]$

Para el segundo caso utilizaremos el resto de Cauchy con:

$\theta \epsilon \left[\frac{1}{2}, 1\right)$; $x \epsilon [-1\ , 1]$

$R_k(x) = \frac{\beta(\beta-1)(\beta-2)\dots(\beta-k)x^{k+1}}{(k+1)!}(1-\theta)^k \rightarrow |R(x)| =$

$|\frac{\beta(\beta-1)(\beta-2)\dots(\beta-k)x^{k+1}}{(k+1)!}(1-\theta)^k| = \frac{|\beta(\beta-1)(\beta-2)\dots(\beta-k)||x^{k+1}|}{|(k+1)!|}|(1-\theta)^k| \rightarrow$

Utilizando las mismas propiedades que en la anterior tenemos:

$|\beta(\beta-1)(\beta-2)\dots(\beta-k)|y^\beta < \beta k!$ (*por sugerencia de los apuntes*)

$(k+1)! > 0 \rightarrow |(k+1)!| = (k+1)!$

$0 < |x^{k+1}| < 1 \ para\ todo\ x\ \epsilon\ [-1,1]$

$\rightarrow \frac{\beta k!|x^{k+1}|}{(k+1)!}|(1-\theta)^k| < |(1-\theta)^k|$ Como $\theta\ \epsilon\ \left[\frac{1}{2},1\right) \rightarrow 0 < (1-\theta) \leq \frac{1}{2} \rightarrow$

$(1-\theta)^k \leq 1/2^k$ Demostrando que $|R(x)| < 1/2^k$ cuando $\theta\ \epsilon\ \left[\frac{1}{2},1\right)$

En conclusión, utilizando el desarrollo de Taylor para $(1+x)^\beta$ podemos asegurar que el resto $|R(x)|$ estara acotado por $1/2^k$ para todo $x\ \epsilon\ [-1,1]$

c) Implementar un módulo que calcule y^β con error de truncamiento menor que tol:

El módulo implementado "**exp_truncada**" es una función que recibe la base *"a"*, el exponente "*b*" y la tolerancia "*tol*" y devuelve el resultado "*res"* = ($y^\beta \pm$ error), con el error menor que la tolerancia.

Inicializamos la variable "*res"* = 0 y *x* = *a* -1, Para construir el polinomio tomamos una variable "*new_b*" que contendrá el producto de los β a medida que se deriva dicho polinomio.

La variable "*k*" es el grado del polinomio que asegura que el error es menor que la tolerancia, se lo calcula como:

$$k = \left[log2\left(\frac{1}{tol}\right)\right] + 1$$

Iterando desde *i*=0 hasta *k*, se construye "*res*" que es continuamente actualizado con la suma de los términos anteriores más el actual; construyendo así el polinomio de grado *k* de Taylor con error menor que la tolerancia. (*ver "**exp_truncada**" en Anexos*).

Ejercicio N° 3

Muestre que sí y >0 entonces existe k ≥ 0 talque $z = \frac{y}{2^k} \epsilon (0, 2)$

$$log_2(z) = log_2\left(\frac{y}{2^k}\right) = \frac{log_2(y)}{log_2(2^k)} = \frac{log_2(y)}{k\ log_2(2)} = \frac{log_2(y)}{k}\ \epsilon\ (-\infty, 1)$$

Si tomamos $k = log_2(y)$ aseguramos que $\frac{y}{2^{log_2(y)}} = \frac{y}{y} = 1$, pero no nos asegura que k ϵ $\mathbb{Z}$, esa condición se cumpliría si y solo si $y = 2^n\ con\ n\ \epsilon\ \mathbb{Z}$ para los otros casos $k\ \epsilon\ \mathbb{R}$

Tomando solo la parte entera, $k = [log_2(y)]$; aseguramos que se cumplen la condición.

$\frac{y}{2^k} = \frac{y}{2^{[log_2(y)]}} \geq \frac{y}{2^{log_2(y)}} \rightarrow$ (Porque $[log_2(y)] \leq log_2(y)) \rightarrow \frac{y}{y} = 1 \rightarrow \frac{y}{2^{[log_2(y)]}} \geq 1$

Se puede asegurar que k es el mínimo porque si tomamos a $k = [log_2(y)]$ -1 tendríamos $\frac{y}{2^{[log_2(y)]-1}} = \frac{2\,y}{2^{[log_2(y)]}}$; sabemos que $\frac{y}{2^{[log_2(y)]}}\ \epsilon\ [1, 2) \rightarrow 2$ por algo que es mayor o igual que 1 nos da como resultado algo que es mayor o igual que 2; demostrando que *k* es el mínimo.

Ejercicio N° 4

Demostración de solución para $\frac{1+x}{1-x} = 2$

$$\frac{1+x}{1-x} = 2 \rightarrow 1 + x = 2 - 2x \rightarrow 1 + 3x = 2 \rightarrow 3x = 1 \rightarrow x = 1/3$$

$$\frac{1+1/3}{1-1/3} = 2 \rightarrow \left(\frac{1+\frac{1}{3}}{1-\frac{1}{3}}\right)^{\beta} = 2^{\beta} \rightarrow \frac{(1+1/3)^{\beta}}{(1-1/3)^{\beta}} = 2^{\beta}$$

Si tomamos

D = $(1 + x_0)^{\beta}$ y N = $(1 - x_0)^{\beta}$ calculamos sus desarollos de Taylor

Taylor de D

$$T_{D,k,x_0}(x) = (1 + x_0)^{\beta} + \beta(1 + x_0)^{\beta-1}(\mathrm{x} - x_0) + \frac{\beta(\beta-1)(1 + x_0)^{\beta-2}(\mathrm{x} - x_0)^2}{2!} + \cdots + \frac{\beta(\beta-1)(\beta-2)\ldots(\beta-k+1)(1 + x_0)^{\beta-k}(\mathrm{x} - x_0)^k}{k!} + R_D(x)$$

Resto de Cauchy

$$R_D(x) = \frac{\beta(\beta-1)(\beta-2)\dots(\beta-k)(1+x_0+\theta h)^{\beta-k-1}(\mathrm{h})^{k+1}}{(k+1)!}(1-\theta)^k$$

Taylor de N

$$T_{N,k,x_0}(x) = (1-x_0)^{\beta} - \beta(1-x_0)^{\beta-1}(\mathrm{x}-x_0) + \frac{\beta(\beta-1)(1+x_0)^{\beta-2}(\mathrm{x}-x_0)^2}{2!} + \cdots + (-1)^k \frac{\beta(\beta-1)(\beta-2)\dots(\beta-k+1)(1+x_0)^{\beta-k}(\mathrm{x}-x_0)^k}{k!} + R_N(x)$$

Resto de Cauchy

$$R_N(x) = (-1)^{k+1}\frac{\beta(\beta-1)(\beta-2)\dots(\beta-k)(1-x_0+\theta h)^{\beta-k-1}(\mathrm{h})^{k+1}}{(k+1)!}(1-\theta)^k$$

Ejercicio N° 5

Escriba un programa que calcule y^{α} en el caso general y > 0

El programa implementado se denomina "**final**", recibe un base "*y*" y un exponente "*a*", donde "*a*" e "*y*" son números reales. Utilizando la propiedad mencionada en el ejercicio 4, sabemos que:

$$y^{\alpha} = y^n * z^{\beta} * (2^{\beta})^k;\ con\ z = \frac{y}{2^k}$$

Nos basaremos en esta propiedad para implementar el módulo de este ejercicio. Las primeras líneas del código definen los valores que vamos a utilizar durante la ejecución:

a_entera = parte entera del exponente

a_decimal = parte decimal del exponente

k = es el valor que cumple $\frac{y}{2^k} \epsilon (0\ ,2)$

tol = valor predefinido para obtener un resultado razonable

z = es el "z" de la descomposición de y^{α}

Con "*exp_egipcia*" calculamos y^n brindándole los siguientes parámetros: *"a_entera*" e "*y*". Luego calculamos 2^{β} con un módulo creado con dicha funcion llamado "**exp_de_dos**" (*ver "**exp_de_dos**" en Anexos*) con los parámetros "*a_decimal*",

"*k*". La variable "z" se lo calcula con el programa "**exp_truncada**" brindándole "*z*", "*a_decimal*" y "*tol*".

Al final simplemente se calcula $total = y^n * z^\beta * (2^\beta)^k$ verificando si $\alpha > 0$ nos devuelve *resultado* = *total*, y si $\alpha < 0$ entonces resultado = $\frac{1}{total}$

Ejercicio N° 6 y N° 7

Corra el programa para y = 1.001, 1.1, 1.96, 2.3, 19.6 y α = 2.33, 3.19, -13.4, -1.94. Comparar los resultados con los obtenidos usando el comando y ^ β de Octave

Para realizar una comparación más sencilla y de forma más visual se realizó una tabla donde se comparan los valores obtenidos con nuestro modulo y los obtenidos con la función predeterminada de Octave, dichos valores se muestran en la Tabla Nº2

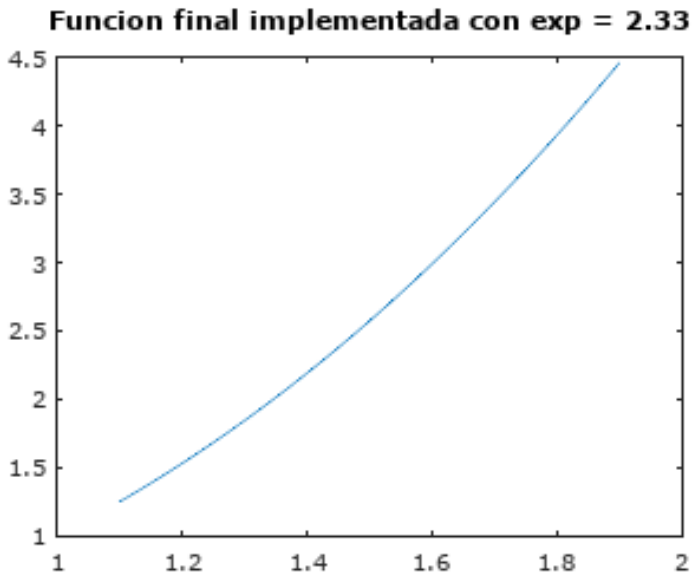
base	exponente	Final (Prog.)	Octave	Error Absoluto	Error Relativo
1.001	2.33	1.00233155	1.00233155	2.22E-16	9.53E-17
	3.19	1.003193494	1.00319349	2.22E-16	6.96E-17
	-13.4	0.986695987	0.98669599	1.11E-16	-8.29E-18
	-1.94	0.998062848	0.99806285	2.22E-16	-1.14E-16
1.1	2.33	1.248662176	1.24866218	4.44E-16	1.91E-16
	3.19	1.355322554	1.35532255	0.00E+00	0.00E+00
	-13.4	0.27882905	0.27882905	5.55E-17	-4.14E-18
	-1.94	0.831185945	0.83118594	1.11E-16	-5.72E-17
1.96	2.33	4.802023267	4.79685561	5.17E-03	2.22E-03
	3.19	8.566632431	8.55651405	1.01E-02	3.17E-03
	-13.4	1.21E-04	1.21E-04	1.12E-07	-8.37E-09
	-1.94	2.71E-01	2.71E-01	8.36E-06	-4.31E-06
2.3	2.33	6.963472389	6.96347239	8.88E-15	3.81E-15
	3.19	1.43E+01	1.43E+01	1.42E-14	4.45E-15
	-13.4	1.42E-05	1.42E-05	0.00E+00	0.00E+00
	-1.94	1.99E-01	1.99E-01	5.55E-17	-2.86E-17
19.6	2.33	4.463219841	1.03E+03	1.57E-03	6.72E-04
	3.19	7.751585736	1.33E+04	2.97E-03	9.32E-04
	-13.4	1.84E-04	4.83E-18	5.56E-08	-4.15E-09
	-1.94	2.88E-01	3.11E-03	2.95E-06	-1.52E-06

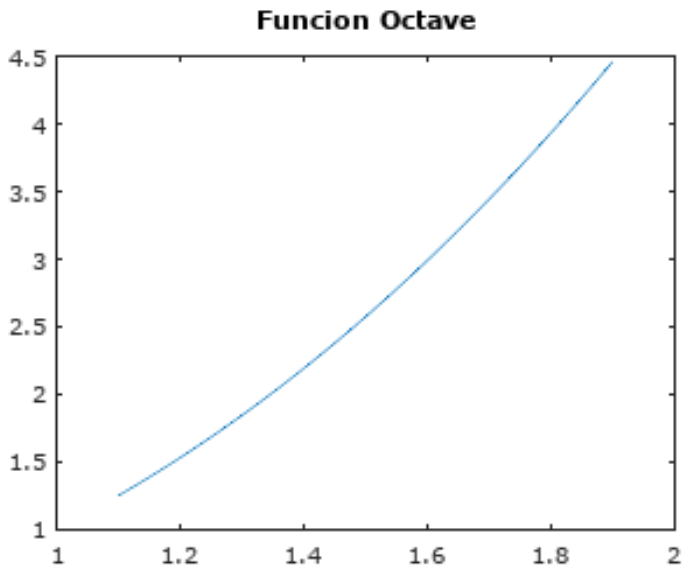
Tabla Nº2: Comparación de los Valores Obtenidos con el módulo "Final" y el predeterminado de Excel.

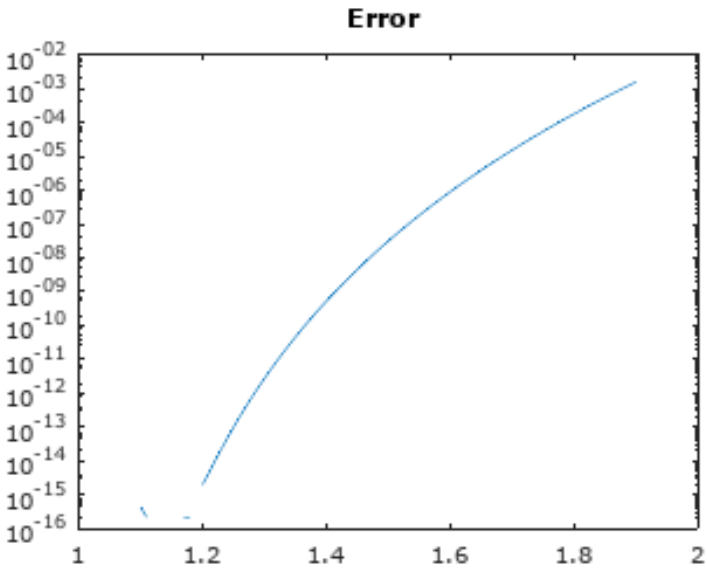
En la Tabla Nº2 se aprecia que los Errores tanto Absolutos como Relativos son significativamente bajos para cualquier valor de la base así también como del exponente. No obstante, los errores más grandes (del orden de 10^-3) son el resultado de implementar el programa con una base muy cercana a 2 (1.96)

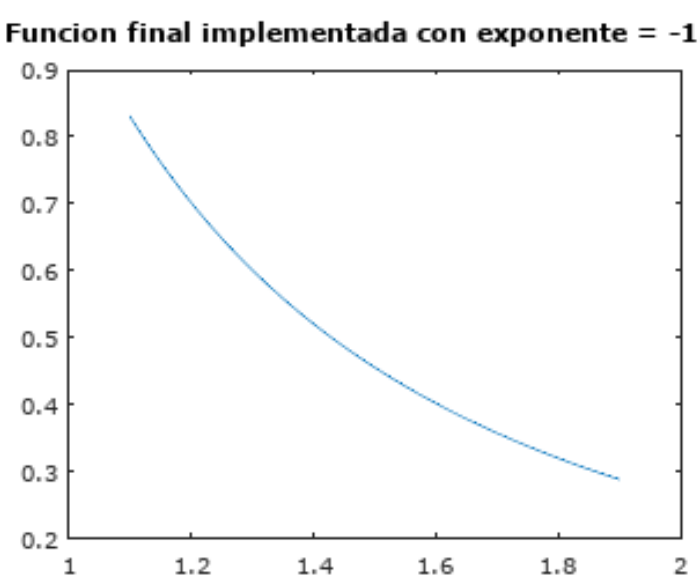
Ejercicio N° 8

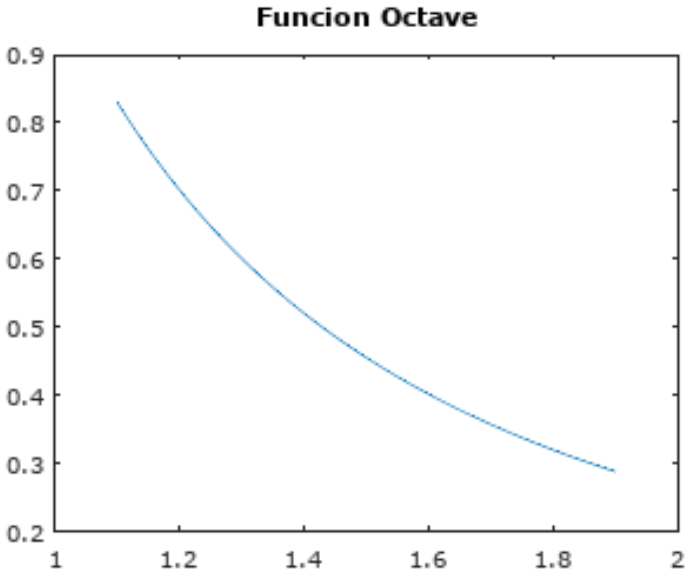
Asumiendo que son ciertos los resultados de Octave, graficar el error en valor Absoluto:

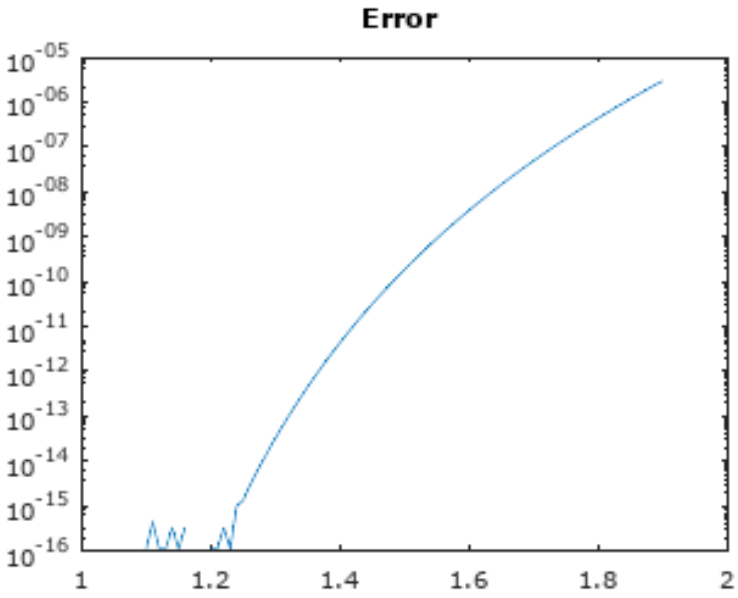







Funcion final implementada con exponente = -1.94
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
1
1.2
1.4
1.6
1.8
2


Funcion Octave
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
1
1.2
1.4
1.6
1.8
2


Error
10-05
10-06
10-07
10-08
10-09
10-10
10-11
10-12
10-13
10-14
10-15
10-16
1
1.2
1.4
1.6
1.8
2

Conclusiones

Una de las primeras conclusiones que obtuvimos en la realización de este trabajo, fue que al implementar una adaptación del Método de multiplicación egipcia se logra calcular una potencia con un exponente natural de manera tal, que el número de operaciones utilizadas es significativamente menor que la cantidad de operaciones utilizadas con la forma tradicional. Al disminuir el número de cuentas arrastramos una menor cantidad de errores, acercándose a los valores que devuelve el procedimiento utilizado por Octave.

En el momento de implementar el ultimo modulo “final” observamos que era necesario un polinomio de Taylor de grado mayor a diez para poder obtener valores tales que al compararlos con los resultados de Octave obtenemos diferencias que fluctúan entre 10^{-3} y 10^{-17} y en algunos casos particulares una diferencia tan pequeña que supera al ε_{mach}.

Mediante la implementación de los módulos que exigía el obligatorio y utilizando diferentes propiedades sugeridas, logramos construir una función que utiliza una combinación de todos los módulos para obtener un programa cuyo propósito es calcular y^{α} con $y \geq 0$, $\alpha \in \mathbb{R}$ sin utilizar “^” o “**” de octave, obteniendo resultados que son coherentes y aplicables.

Anexo

Int_a_binario

```
function bin = int_a_binario(a)
 if (a == 0)
   bin = 0;

 elseif (a != 0)
  size = fix(log2(a)) +1;
  bin = zeros(1,size);

  for i = size:-1:1
    bin(i) = mod(a,2);
    a = fix(a/2);

    endfor
    endif
```

exp_egipcia

```
function res = exp_egipcia(a,b)
  ## a es la base
  ## b es el exponente
  if(a == 0)
  res = 0;
  elseif(b == 0)
  res=1;
  else
  bin = int_a_binario(b);
  size = length(bin);
  res=1;
  for i = size:-1:1
    if (bin(i) == 1)
      res = res * a;
      endif
      a = a*a;
   endfor
  endif
```

exp_truncada

```
function res = exp_truncada(a, b, tol)
 res = 0;
 x = a-1;

 new_x = 1;

 new_b = 1;

 k = fix( abs( log2(1/tol) ) ) +1 ;

for i = 0 : k
  res = res + ( ( new_b * new_x)/ factorial(i)) ;
  new_b = new_b * (b - i);
  new_x = new_x*x;

  endfor
```

exp_de_dos

```
function res = exp_de_dos(b)
 N =0;
 D = 0;
 x=1/3;
 k = 27;

 new_b = 1;
 new_x = 1;



 for i = 0: k
  N = N + (new_b*new_x / factorial(i) );
  new_b = new_b * (b-i);
  new_x = new_x * x;
  endfor

  new_b = 1;
  new_x = 1;

 for i = 0 : k
   D = D + ( (-1)^i * new_b*new_x /
factorial(i) );
  new_b = new_b * (b-i);
  new_x = new_x *x;
  endfor

  res = N/D;
```

final

```
function res = final(y, a)
  negativo = true;

  if a < 0;
    negativo = true;
    a = abs(a);
  else negativo = false;
    endif

  a_entera = fix(a);
  a_decimal = a- a_entera;
  k = fix(log2(y));
  tol = (10^-5);

  z =  (1/(2^k) * y);



  Primero calculamos y^n -> y_n
  y_n = exp_egipcia(y, a_entera);
  Segundo calculamos 2^b -> dos_beta
  dos_beta = exp_de_dos(a_decimal);
```

cont…

Cont…

```
## luego calculamos 2^b^k ->
dos_beta_k
  dos_beta_k = exp_egipcia(dos_beta,
k);


Exponenciación truncada se pasa (la
base, exponente, tolerancia)
  z = exp_truncada(z,a_decimal, tol);



  total = z*y_n*dos_beta_k;

  if negativo ==true ;
    res = (1/total);
  else
    res = total;
    endif
```