PRÁCTICA 2:
Programación con Maxima

2.1 Resumen de Teoría

2.1.1 Introducción

En la Práctica 1 vimos una introducción global al programa de cálculo simbólico Maxima, donde revisamos su funcionamiento general y presentamos diversos comandos que resultan válidos para asignaturas como Cálculo, Álgebra, etc.

En esta Práctica 2 vamos a seguir profundizando en Maxima, pero ahora ya desde una perspectiva más orientada a los Métodos Numéricos. Para ello veremos distintos comandos destinados a la programación de algoritmos, si bien todavía no profundizaremos en ninguno de los temas clásicos del Análisis Numérico, lo cual ya haremos a partir de la siguiente Práctica.

Comenzaremos repasando algunos conceptos teóricos que, por su carácter general, nos serán de utilidad en todos los temas que trataremos este curso.

2.1.2 Definiciones de Error en los Métodos Numéricos

Los errores numéricos surgen del uso de aproximaciones para representar operaciones y cantidades matemáticas exactas. Estos errores incluyen:

- Los *errores de truncamiento* que resultan del empleo de aproximaciones como un procedimiento matemático exacto, y
- Los *errores de redondeo* que se producen cuando se usan números que tienen un límite de cifras significativas para representar números exactos.

Para ambos tipos de errores, el *Error numérico verdadero*, E_t, es igual a la diferencia entre el valor verdadero y el valor aproximado, es decir:

$$E_t = \text{Valor verdadero} - \text{Valor aproximado}$$

En la expresión anterior se puede tomar el valor absoluto y se habla entonces de *Error absoluto*.

Una desventaja en esta definición es que no toma en consideración el orden de la magnitud del valor que se estima. Por ello se suele normalizar el error, considerando el *Error relativo verdadero*:

$$\varepsilon_t = \frac{\text{Error verdadero}}{\text{Valor verdadero}} = \frac{E_t}{\text{Valor verdadero}}$$

que también se puede multiplicar por 100% para dar el *Error relativo porcentual verdadero*.

En las ecuaciones anteriores *E* y *e* tienen un subíndice *t* que significa que el error ha sido calculado usando el valor verdadero. Sin embargo, en muchas aplicaciones reales, no se conoce *a priori* la respuesta verdadera.

Uno de los retos a los que se enfrentan los métodos numéricos es el de determinar estimaciones del error sin conocer los valores verdaderos. Por ejemplo, ciertos métodos numéricos usan un *método iterativo* para calcular los resultados. En tales métodos se calcula una aproximación basándose en la aproximación anterior. En tales casos, el error se calcula, a menudo, como la diferencia entre la aproximación previa y la actual. Por lo tanto, el *Error relativo porcentual*, ε_a, está dado por:

$$\varepsilon_a = \frac{\text{Aprox actual} - \text{Aprox anterior}}{\text{Aprox actual}} 100\%$$

También es cierto que cuando se realizan cálculos, no importa mucho el signo del error, sino más bien que su valor absoluto porcentual sea menor que una tolerancia porcentual prefijada ε_s. Por lo tanto, en tales casos, los cálculos se repiten hasta que:

$$|\varepsilon_a| < \varepsilon_s$$

Si se cumple la relación anterior, entonces se considera que el resultado obtenido está dentro del nivel aceptable fijado previamente ε_s.

También podemos relacionar estos errores con el número de cifras significativas de la aproximación. Es posible demostrar que si el siguiente criterio se cumple, se tendrá la seguridad que el resultado es correcto en *al menos n* cifras significativas:

$$\varepsilon_s = (0.5 \times 10^{2-n})\%$$

2.1.3 Errores de Redondeo

Como se mencionó antes, los errores de redondeo se originan debido a que la computadora emplea un número determinado de cifras significativas durante un cálculo. Números como π, e ó $\sqrt{2}$, no pueden ser representados exactamente por el ordenador.

Además, debido a que los ordenadores usan una representación en base 2, no pueden representar exactamente algunos números en base 10. Esta discrepancia por la omisión de cifras significativas se llama *error de redondeo*.

Debido a que tenemos 10 dedos en las manos, el sistema de numeración que nos es muy familiar es el *decimal* o de *base* 10. Por ejemplo, el número 86 409 sería:

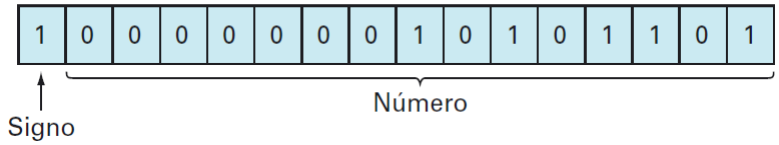
$$(8 \times 10^4) + (6 \times 10^3) + (4 \times 10^2) + (0 \times 10^1) + (9 \times 10^0) = 86\,409$$

En este sentido es como si el ordenador tuviera sólo dos dedos: las unidades lógicas de sus componentes electrónicos de apagado/encendido. Por lo tanto, los números en el ordenador se representan en sistema *binario* o de *base* 2. Por ejemplo, el número binario 11 es equivalente a:

$$(1 \times 2^1) + (1 \times 2^0) = 2 + 1 = 3$$

en el sistema decimal.

Basándose en esto, veamos cómo se representan los enteros en un ordenador. El método más sencillo se denomina *aritmética de punto fijo* y emplea el primer bit de una palabra para indicar el signo: con un 0 para positivo y un 1 para el negativo. Los bits sobrantes se usan para guardar el número.



Por ejemplo, en un ordenador de 16 bits, se tiene el primer bit para el signo. Los 15 bits restantes pueden contener los números binarios de 0 a 111111111111111. El límite superior se convierte en un entero decimal:

$$(1 \times 2^{14}) + (1 \times 2^{13}) + \ldots + (1 \times 2^{1}) + (1 \times 2^{0}) = 32\,767$$

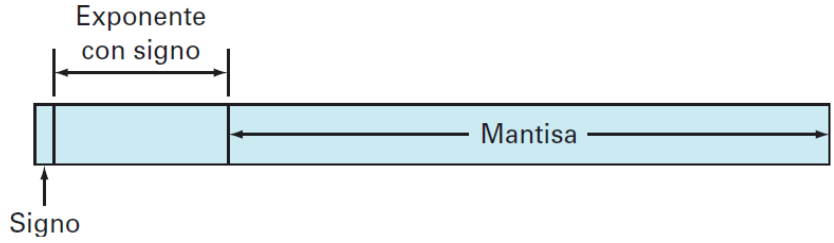
Así, un ordenador de 16 bits puede guardar en memoria un entero decimal en el rango de –32767 a 32767. Estos números se representan exactamente pero los números por encima o por debajo de este rango no pueden representarse y se genera un error de desbordamiento o "overflow".

Una limitación más importante se encuentra en el almacenaje y la manipulación de cantidades fraccionarias. Las cantidades fraccionarias generalmente se representan en un ordenador usando la forma de *aritmética de punto flotante*. Con este método, el número se expresa como una parte fraccionaria, llamada *mantisa*, y una parte entera, denominada *exponente*, esto es:

$$m \cdot b^{e}$$

donde m es la mantisa, b es la base del sistema numérico que se va a utilizar y e es el exponente.

Por ejemplo, el número 156.78 se representa como 0.15678×10^{3} en un sistema de base 10 de punto flotante. En 1a figura se muestra una forma en que el número de punto flotante se guarda en un ordenador.



El primer bit se reserva para el signo; la siguiente serie de bits, para el exponente con signo; y los últimos bits, para la mantisa.

La representación de punto flotante permite que tanto fracciones como números muy grandes se almacenen en un ordenador. Sin embargo, tiene algunas desventajas. Para empezar, los números de punto flotante requieren más espacio y más tiempo de computación que los números enteros. Pero más importante aun es que su uso introduce una fuente de error debido a que la mantisa conserva sólo un número finito de cifras significativas. Por lo tanto, se introduce un error de redondeo.

A modo de resumen presentamos varios aspectos en los que influyen la representación de punto flotante, y los errores de redondeo.

1. *El rango de cantidades que pueden representarse es limitado.* Como en el caso de los enteros, hay números grandes positivos y negativos que no pueden representarse. Al intentar emplear números fuera del rango aceptable dará como resultado el llamado *error de desbordamiento* (*overflow).*

2. *El grado de precisión es limitado.* Es evidente que, por ejemplo, los números irracionales no pueden representarse de manera exacta. Consideremos, por ejemplo, que vamos a guardar:

$$\pi = 3.14159265358\ldots$$

en un sistema de numeración de base 10 con 7 cifras significativas. Un método de aproximación podría ser simplemente omitir, o "cortar":

$$\pi = 3.141592$$

Esta técnica de mantener sólo términos significativos fue originalmente conocida como "truncamiento" en la jerga computacional. Preferimos llamarla *corte* para distinguirla de los errores de truncamiento. Otra posibilidad es el conocido redondeo. Así, en el ejemplo de π, el primer dígito descartado es 6. El último dígito retenido debería redondearse a:

$$\pi = 3.141593$$

El redondeo produce un error absoluto menor que el de corte y por ello algunos ordenadores emplean redondeo. Sin embargo, esto aumenta el trabajo computacional y, en consecuencia, muchas máquinas simplemente usan el corte.

3. *Épsilon de Máquina.* En una aritmética de punto flotante, se llama épsilon de la máquina (ε-mach) al número decimal más pequeño que, sumado a 1, el ordenador nos devuelve un valor diferente de 1, es decir, que no es redondeado.

Representa la exactitud relativa de la aritmética del computador. La existencia del épsilon de la máquina es una consecuencia de la precisión finita de la aritmética en punto flotante.

4. *Precisión extendida*. Los ordenadores que usan el formato IEEE permiten 24 bits para ser usados por la mantisa, lo cual se traduce en cerca de siete cifras significativas de precisión en dígitos de base 10 con un rango aproximado de 10^{-38} a 10^{39}.

Parece suficiente pero aún así hay casos donde el error de redondeo resulta crítico. Por tal razón muchos ordenadores permiten la especificación de precisión extendida. La más común de estas especificaciones es la doble precisión, en la cual se duplica el número de palabras utilizado para guardar números de punto flotante. Esto proporciona de 15 a 16 dígitos decimales de precisión y un rango aproximado de 10^{-308} a 10^{308}.

En muchos casos el uso de cantidades de doble precisión llega a reducir, en gran medida, el efecto del error de redondeo. Sin embargo, el precio que se paga consiste en mayores requerimientos de memoria y de tiempo de ejecución.

2.2 Precisión en Maxima

En esta práctica vamos a seguir analizando aspectos generales de wxMaxima, pero en esta ocasión ya enfocados a la asignatura que nos ocupa: Métodos Numéricos.

Si escribimos, por ejemplo:

```
(%i1)    4^200;
(%o1)    258224987808690858965591917200[61 dígitos]280137831435903171972747493376
```

El resultado no pone todos los dígitos, de hecho nos informa que ha omitido 61 dígitos, y muestra los 30 primeros y los 30 últimos. Para saber los dígitos omitidos vamos al menú **Maxima-Cambiar pantalla 2D** y escogemos ascii, si ahora repetimos la operación obtendremos:

```
(%i2)    set_display('ascii)$

(%i3)    4**200;
(%o3) 258224987808690858965591917200301187432970579282922351283065935654064762\
 201684119462964535328013783143590317197274749337 6
(%i4)    2^500;
(%o4) 327339060789614187001318969682759915221664204604306478948329136809613379\
 640467455488327009232590415715088668412756007100921725654588539305332852758937
```

Ahora sí los muestra todos. La barra que aparece al final de la línea no es la de cociente, sólo nos informa de que el número sigue. Para volver al modo por defecto de pantalla, vamos al menú **Maxima-Cambiar pantalla 2D** y escogemos xml, como figura a continuación:

```
(%i5)    set_display('xml)$
```

Maxima es un programa de cálculo simbólico y hace las operaciones encomendadas de forma exacta, por ejemplo la suma de fracciones devuelve otra fracción y lo mismo la raíz cuadrada, a no ser que se le pida usando la sentencia: **float**(número) que da la expresión decimal de número con 16 dígitos. La instrucción número**,numer** realiza el mismo efecto y también da la expresión decimal de número con 16 dígitos.

Por otro lado **bfloat**(número) da la expresión decimal larga de número acabada con *b* seguido de un número *n*, que significa multiplicar por 10^n.

La precisión que nos brinda el programa (por defecto 16 dígitos) se puede modificar con la instrucción **fpprec:** que indica el número de dígitos a utilizar. Veamos algunos ejemplos:

```
(%i6)    sqrt(2);
(%o6)    √2
(%i7)    sqrt(2),numer;
(%o7)    1.414213562373095
(%i8)    float(sqrt(2));
(%o8)    1.414213562373095
```

```
(%i9)    bfloat(sqrt(2));
(%o9)    1.4142135623730950b0

(%i10)   fpprec: 100;
(fpprec) 100

(%i11)  bfloat(sqrt(2));
(%o11)  1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753432764157 3b0

(%i12)   set_display('ascii)$

(%i13)  bfloat(sqrt(2));
(%o13) 1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753432764157 3b0

(%i14)    fpprec:1000;
(%o14)                                  1000

(%i15)  bfloat(%pi);
(%o15) 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526193118817101000313783875288658753320838142061717766914730359825349042875546873115956286388235378759375195778185778053217122680661300192787661119590921642019 9b0
```

Volvamos al estado original:

```
(%i16)   fpprec : 16;
(%o16)                                   16
(%i17)   set_display('xml)$
```

2.3 Operadores lógicos

Los operadores lógicos son:

and = y (conjunción)
not = no (negación)
or = o (disyunción)

Maxima puede averiguar la certeza o falsedad de una expresión mediante el comando:

is(expresión) decide si la expresión es cierta o falsa

```
(%i1)    kill(all);
(%o0)    done
(%i1)    is (7<8 and 9>10);
(%o1)    false
(%i2)    is (8>9 or 5<6);
(%o2)    true
(%i3)    is (x^2-1>0);
(%o3)    unknown
```

También podemos hacer hipótesis y olvidarnos de ellas por medio de los comandos:

assume(expresión) = suponer que la expresión es cierta
forget(expresión) = olvidar la expresión o hipótesis hecha

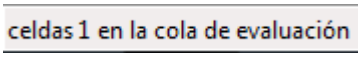
```
(%i5)    assume (x>1)$
         is(x^2-1>0);
(%o5)    true

(%i7)    forget(x>1)$
         is(x^2-1>0);
(%o7)    unknown
```

Como se observa en la última salida al olvidarnos de la hipótesis de ser $x > 1$, ya no sabe si es verdadera o falsa la expresión.

Cuando Maxima tiene dudas sobre algún dato que puede influir en cuál sea su respuesta, antes de darla hace preguntas. Por ejemplo:

```
(%i8)    integrate(exp(a*x),x,0,inf);
Is a positive, negative or zero?
```

Para calcular el valor de esta integral impropia, Maxima necesita saber el signo del parámetro *a*. La celda como vemos está resaltada y además en la esquina inferior izquierda de la pantalla aparece este mensaje:



que nos indica que hasta que no contestemos a la pregunta no podremos seguir trabajando. Para contestar basta escribir la inicial de la respuesta. En este caso para indicar que es negativo pondremos:

```
(%i8)    integrate(exp(a*x),x,0,inf);
Is a positive, negative or zero?n;
```

$$(\%o8) \quad -\frac{1}{a}$$

El comando **assume** permite evitar las preguntas dando de antemano las respuestas.

2.4 Bloques y funciones

Es una buena costumbre, al empezar a programar, que antes de definir una función de nombre (el que sea), la borres con **kill**(nombre) (el que sea) o todo, con **kill(all)**.

```
(%i1)    kill(all)$
```

La forma más fácil de escribir programas dentro Maxima es a través de funciones o bloques, por ejemplo:

```
(%i1)    suma(x,y):=x+y;
(%o1)    suma(x,y):=x+y
```

crea una función que devuelve la suma de los argumentos. Para usarla puedes hacer:

```
(%i2)    suma(1,2);
(%o2)    3
```

Ya vimos que para definir funciones se utiliza el comando **:=** o **define**. Si queremos que el cuerpo de la función sea una sucesión de expresiones se hace **f(x):=(expr1,expr2,...,exprn)** y se devuelve sólo la última.

```
(%i3)    f(x):=(x^2,x^3,sqrt(x));
(%o3)    f(x):=(x^2,x^3,sqrt(x))
(%i4)    f(4);
(%o4)    2
```

La forma más común para programar en Maxima es usar el comando **block** para crear subrutinas. El comando **block**, entre otras cosas, permite limitar el campo de acción de una asignación a un bloque de código. Por tanto, con este comando se pueden usar variables locales que no interfieren con las variables que usemos fuera:

f(x):=block([lista de variables locales],cuerpo)
f(x):=block(cuerpo)

O bien:

block([v_1,…, v_n], $expr_1$,…,$expr_m$)
block($expr_1$,…, $expr_m$)

El comando **block** evalúa las expresiones secuencialmente y devuelve el valor de la última expresión evaluada. En caso de usarse, las variables $v_1,\ldots,v_n$ son locales en el bloque y se distinguen de las globales que tengan el mismo nombre. Si no se declaran variables locales entonces se puede omitir la lista.

El valor del bloque es el de la última sentencia o el argumento de la función **return**, que puede utilizarse para salir del bloque. Veamos un ejemplo de **block**.

```
(%i9)    x:4$ y:5$
         x*y;
         block([x,y],x:7, y:8, x*y);
         x*y;
(%o7)    20
 (%o8)    56
 (%o9)    20
```

Como se observa las variables *x* e *y* siguen valiendo lo mismo antes y después del **block** y diferente dentro del mismo por haberlas declarado como variables locales dentro del **block**. Que en un programa **block** se puedan utilizar variables locales que no interfieran con el resto de Maxima, es muy importante para no machacar los valores externos si la variable se llama igual que otra definida fuera.

El siguiente ejemplo pone de manifiesto de nuevo el efecto de las variables locales y globales, que el bloque sólo devuelve la última expresión, y también ilustra el uso de la orden **return** para salirse prematuramente de un **block**.

```
(%i11)   f(x):=block([x],x,x^2,x^3)$
         f(3);
(%o11)   x^3
(%i13)   g(x):=block(x,x^2,x^3)$
         g(3);
(%o13)   27
(%i15)   h(x):=block(x,x^2,return(x),x^3)$
         h(3);
(%o15)   3
```

Los programas **block** de Maxima devuelven la última instrucción realizada, por lo que si alguna vez vamos a usar la salida de un programa como entrada de otro (o del mismo) y se necesitan varios datos de entrada es posible que tengamos que poner la entrada y la salida del programa como una lista de valores.

```
(%i16)   block([x],x:2,a:x^2,b:x^3,[a,b]);
(%o16)   [4,8]
```

2.5 Condicionales y bucles

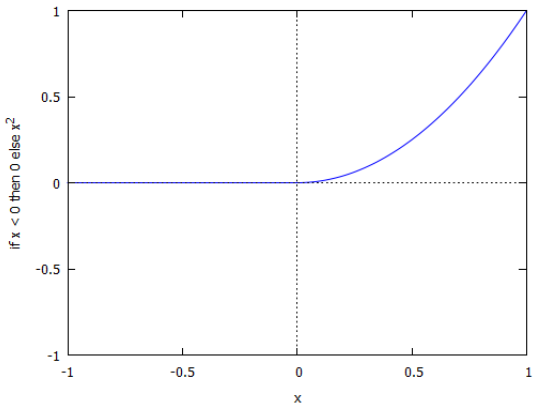
En la programación de algoritmos numéricos es fundamental la utilización de condicionales y bucles, los primeros en Maxima se realizan con la función **if** (si es necesario, acompañada por **else**). La forma básica será:

if condición **then** acción1 **else** acción0

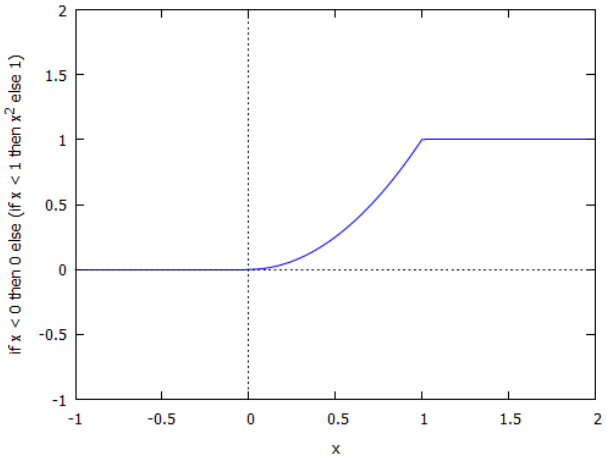
Este comando, si condición es verdadera, ejecuta acción1, en caso contrario, es decir si es falsa, ejecuta la expresión tras el else o sea acción0.

Veamos algunos ejemplos:

```
(%i1)    kill(all)$
(%i2)    F(x):=if x<0 then 0 else x^2$
         plot2d(F(x),[x,-1,1],[y,-1,1])$
```




```
(%i4)    F(x):=if x<0 then 0 else (if x<1 then x^2 else 1)$
         plot2d(F(x),[x,-1,2],[y,-1,2])$
```



Para los bucles disponemos de **for**, que tiene las siguientes variantes:

for < var >:< val1 > **step** < val2 > **thru** < val3 > **do** < expr >
for < var >:< val1 > **step** < val2 > **while** < cond > **do** < expr >
for < var >:< val1 > **step** < val2 > **unless** < cond > **do** < expr >

El primer **for** desde el valor inicial val1 de la variable var la va incrementando con paso val2 y siempre que esta sea menor o igual que val3 calcula expr. En el segundo caso, partiendo del mismo valor inicial y con el mismo incremento, calcula expr mientras se verifique cond. Y en el tercero calcula expr a no ser que se verifique cond.

Cuando el incremento de la variable es la unidad, se puede obviar la parte de la sentencia relativa a **step**.

Cuando no sea necesaria la presencia de una variable de recuento de iteraciones, también se podrá prescindir de los **for**, como en:

while < cond > **do** < expr >

```
(%i1)    kill(all)$
(%i3)    suma:0;
         for i:1 thru 10 do suma:suma+i;
         suma;
(suma)   0
 (%o2)    done
 (%o3)    55
(%i7)    suma:0;
         n:1$
         while (n<=10) do (suma:suma+n,n:n+1);
         suma;
(suma)   0
 (%o6)    done
 (%o7)    55
```

Como vemos, al utilizar sólo **while**, debemos incluir un contador que vaya incrementado su valor. Además es importante su colocación en el bucle. Observe:

```
(%i11)   suma:0;
         n:1$
         while (n<=10) do (n:n+1,suma:suma+n);
         suma;
(suma)   0
 (%o10)   done
 (%o11)   65
```

Para salidas por pantalla se utilizan los comandos **disp**, **display** y **print.** Veamos unos ejemplos:

```
(%i13)   n:1$
         while (n<4) do (x:n^2,
             print("el valor de x para", n, "es",x),n:n+1)$
el valor de x para  1  es  1
 el valor de x para  2  es  4
 el valor de x para  3  es  9

(%i15)   n:1$
         while (n<4) do (x:n^2,disp(x),n:n+1)$
1
 4
 9

(%i17)   n:1$
         while (n<4) do (x:n^2,display(x),n:n+1)$
x=1
 x=4
 x=9
```

Para realizar sumatorios, como no podía ser de otra forma, Maxima tiene implementada la función **sum**, que adopta la forma:

sum(expr,i,i0,i1)

que hace la suma de los valores de expr según el índice i varía de i0 a il. Veamos por ejemplo la suma de los cuadrados de los 100 primeros números naturales:

```
(%i18)   sum(i^2,i,1,100);
(%o18)   338350
```

Pero si queremos hacerlo con nuestro propio bucle (por ejemplo con **for**) escribiremos:

```
(%i21)   suma:0$
         for i:1 thru 100 do (suma:suma+i**2)$
         print(suma)$
338350
```

Del mismo modo, para realizar productos, Maxima también tiene implementada la función **product**, que adopta la forma:

product(expr,i,i0,i1)

que representa el producto de los valores de expr según el índice i varía de i0 a il. Ejemplo:

```
(%i22)   product(i,i,1,5);
(%o22)   120
(%i25)   producto:1$
         for i:1 step 1 thru 5 do (producto:producto*i)$
         print(producto)$
120
```

En los ejemplos anteriores la variable suma o producto iba actualizando su valor y no definimos ninguna función. Pero Maxima también permite la definición de funciones por recurrencia. Por ejemplo si queremos definir el factorial de un número natural podríamos hacerlo como sigue:

```
(%i26)   fact(n):= if n=1 then 1 else n*fact(n-1);
(%o26)   fact(n):=if n=1 then 1 else n fact(n-1)
(%i27)   fact(5);
(%o27)   120
```

Recordemos que el valor del **block** es el de la última sentencia o el argumento de la función **return**, que puede utilizarse para salir del bloque. La función return puede usarse dentro del **block** para salir de **do** de forma prematura, retornando un valor determinado. Nótese no obstante que un **return** dentro de un **do** que está dentro de un **block** provocará una salida de **do** pero no de **block**. Véase con atención el siguiente ejemplo:

```
(%i29)   block(a:0,for i:1 thru 15 do
              (a:a+1, if i=5 then (return(i)))
              ,8*6);
         print("a=",a)$
(%o28)   48
 a=  5
```

2.6 Funciones útiles para la programación

Entre las muchas variables opcionales de Maxima, repararemos de momento sólo en las siguientes, las cuales por defecto tienen asignado el valor **false**.

numer
showtime

Por ejemplo si declaramos **numer:true** entonces algunas funciones matemáticas con argumentos numéricos se evalúan como decimales de punto flotante. También hace que las variables de una expresión a las cuales se les ha asignado un número sean sustituidas por sus valores y activa la variable **float**. Para volver a su valor predeterminado hay que reiniciar Maxima o bien declararlas como **false**.

```
(%i1)    kill(all);
(%o0)    done
(%i1)    numer:true;
(numer)  true
(%i2)    %pi^5;
(%o2)    306.0196847852814
(%i3)    numer:false;
(numer)  false
(%i4)    %pi^5;
(%o4)    π^5
```

Si hacemos la declaración de la variable **showtime:true** se muestra el tiempo de cálculo con la salida de cada expresión. Y sigue así hasta que se reinicie el programa o se declare como **false**. También existe la función **time(%o)** que devuelve el tiempo, en segundos, que fue necesario para calcular la salida %o. La función **time** sólo puede utilizarse para variables correspondientes a líneas de salida; para cualquier otro tipo de variables, **time** devuelve **unknown**. En la siguiente sección veremos varios ejemplos.

2.7 Aritmética exacta versus aritmética de redondeo

Como dijimos al principio Maxima es un programa de cálculo simbólico y hace las operaciones encomendadas de forma exacta, por ejemplo la suma de fracciones devuelve otra fracción y lo mismo la raíz cuadrada u otras funciones cuyo resultado no sea un entero, a no ser que se le pida mediante **float(número)** o **número,numer**, que dan la expresión decimal de número con 16 dígitos o también con **bfloat(numero)** que da la expresión decimal larga de número acabada con *b* seguido de un número *n*, que significa multiplicar por 10^n.

Pero trabajar en aritmética exacta no es conveniente en Cálculo Numérico, pues el tiempo empleado en los cálculos aumenta considerablemente, no siendo proporcional al número de operaciones efectuadas y en muchas ocasiones aunque Maxima calcule las operaciones encomendadas no llega a mostrar los resultados.

Veamos lo que acabamos de afirmar en los siguientes ejemplos, en los que se calcula la suma de los inversos de los cuadrados de los n = 100, 1000, 10000, 100000 primeros números naturales en aritmética exacta y se muestra el tiempo de cálculo empleado en realizar dicha suma:

```
(%i1)    kill(all);
(%o0)    done
(%i1)    showtime:true$
Evaluation took 0.0000 seconds (0.0000 elapsed) using 80 bytes.
(%i2)    sum (1/i^2,i,1,100)$
Evaluation took 0.0156 seconds (0.0156 elapsed) using 69.086 KB.
(%i3)    sum (1/i^2,i,1,1000)$
Evaluation took 0.0468 seconds (0.0468 elapsed) using 1.735 MB.
(%i4)    sum (1/i^2,i,1,10000)$
Evaluation took 1.2636 seconds (1.2636 elapsed) using 124.155 MB.
(%i5)    sum (1/i^2,i,1,100000)$
Evaluation took 102.1495 seconds (102.7256 elapsed) using 11943.080 MB.
```

En general, los ordenadores trabajan en aritmética de redondeo, también llamada de punto flotante, con un número fijo k de dígitos; esta aritmética tiene propiedades algo distintas de las habituales, por ejemplo la suma deja de ser asociativa, hay números no nulos que al ser sumados a otro no alteran la suma, se produce la pérdida de cifras significativas al hacer la diferencia de números próximos, etc.; pero se suelen obtener resultados satisfactorios en un tiempo razonable, en general muchísimo menor que en la aritmética exacta y los tiempos empleados tienden a ser proporcionales al número de operaciones realizadas. Recordemos también que por defecto Maxima trabaja en aritmética exacta, de manera que si queremos que lo haga en aritmética de redondeo podemos declararlo mediante **numer:true.**

Veamos diversos ejemplos con el anterior sumatorio:

```
(%i6)    numer:true$
Evaluation took 0.0000 seconds (0.0000 elapsed) using 100 bytes.
(%i7)    sum (1/i^2,i,1,100);
Evaluation took 0.0000 seconds (0.0020 elapsed) using 40.309 KB.
 (%o7)    1.634983900184892
(%i8)    sum (1/i^2,i,1,1000);
Evaluation took 0.0312 seconds (0.0312 elapsed) using 402.418 KB.
 (%o8)    1.643934566681561
(%i9)    sum (1/i^2,i,1,10000);
Evaluation took 0.2652 seconds (0.2652 elapsed) using 4.036 MB.
 (%o9)    1.644834071848065
(%i10)   sum (1/i^2,i,1,100000);
Evaluation took 2.7768 seconds (2.7924 elapsed) using 41.753 MB.
 (%o10)   1.644924066898242
```

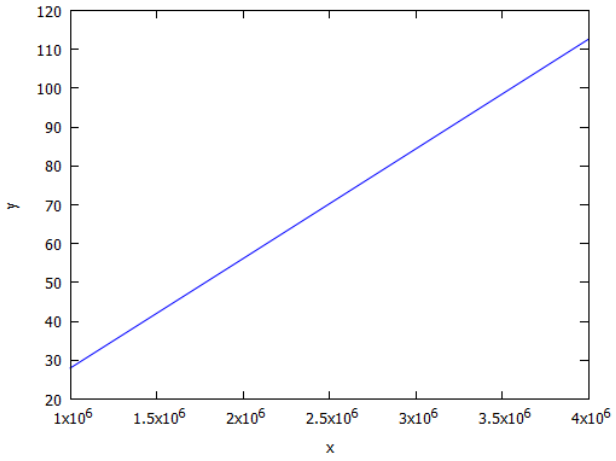
Se observa que la aritmética de redondeo es mucho más rápida que la exacta.

Seguidamente, vamos a aumentar el número de términos en el sumatorio anterior a 1, 2 y 4 millones para ver como el tiempo de cálculo empleado tiende a ser proporcional al número de operaciones realizadas. Recordemos que seguimos en punto flotante de 16 cifras). Mostramos el tiempo de cálculo y realizamos los sumatorios pedidos:

```
(%i11)   sum (1/i^2,i,1,1000000);
Evaluation took 27.9710 seconds (28.1132 elapsed) using 422.765 MB.
  (%o11)   1.64493306684877
(%i12)   sum (1/i^2,i,1,2000000);
Evaluation took 56.1604 seconds (56.3513 elapsed) using 848.165 MB.
  (%o12)   1.644933566848388
(%i13)   sum (1/i^2,i,1,4000000);
Evaluation took 112.6639 seconds (113.0552 elapsed) using 1702.655 MB.
  (%o13)   1.644933816848411
```

Se observa el crecimiento lineal del tiempo de CPU en la aritmética de redondeo (lo que se pone de manifiesto con la gráfica discreta de tiempos empleados frente al número de operaciones), no ocurre lo mismo en la aritmética exacta.

```
(%i1)    kill(all)$
         plot2d([discrete, [[10^6, 27.971],[2*10^6, 56.1604],
         [4*10^6,112.6639]]])$
```



En el comando **plot2d** usamos **[discrete, puntos]** para dibujar los puntos, que se introducen por pares: **[[x0,y0],[x1,y1],…].** Por defecto los une.

2.8 Épsilon de máquina

Cuando trabajamos en aritmética de redondeo, conviene recordar que denominamos épsilon de máquina ε al número positivo más pequeño que al ser sumado a otro número cualquiera a verifica que ε + a > a, de manera que todo número positivo ε´ menor que ε verifica que ε´ + a = a. Por ejemplo, vamos a hallar el primer número positivo ε tal 1 + ε > 1.

En base 2 será de la forma 0.000000001, luego una potencia de exponente negativo de 2. Hemos de hallar pues el primer número *n* tal que $1 + 2^{-n} = 1$, en aritmética de punto flotante, en cuyo caso el cero de máquina será $2^{-(n-1)} = 2^{-n+1}$ y puede obtenerse, por ejemplo, con el programa:

```
(%i1)    kill(all)$
(%i1)    showtime:false$
(%i2)    numer:true$

(%i6)    fpprec:20$
         n:0$
         while (1.0+2^(-n)>1.0) do (n:n+1);
         print("El épsilon de máquina es 2^(",-n+1,") = ",
                     float(2^(-n+1)))$
(%o5)    done
 El épsilon de máquina es 2^( -52 ) =  2.220446049250313 10^-16
```

Notemos que al poner 1.0 en el programa anterior, este fuerza a Maxima a trabajar en aritmética de redondeo. Una forma de obtener el épsilon de máquina la da el programa:

```
(%i9)    epsilon:1.0$
         while ((1+epsilon/2)>1) do(epsilon:epsilon/2)$
         print("El epsilon de máquina de Maxima: ",float(epsilon))$
El epsilon de máquina de Maxima:  2.220446049250313 10^-16
```

Podemos preguntar si el número hallado cumple la definición en la forma:

```
(%i10)   is (1+2.220446049250313*10^-16>1);
(%o10)   true
(%i11)   is (1+(2.220446049250313*10^-16)/2>1);
(%o11)   false
```

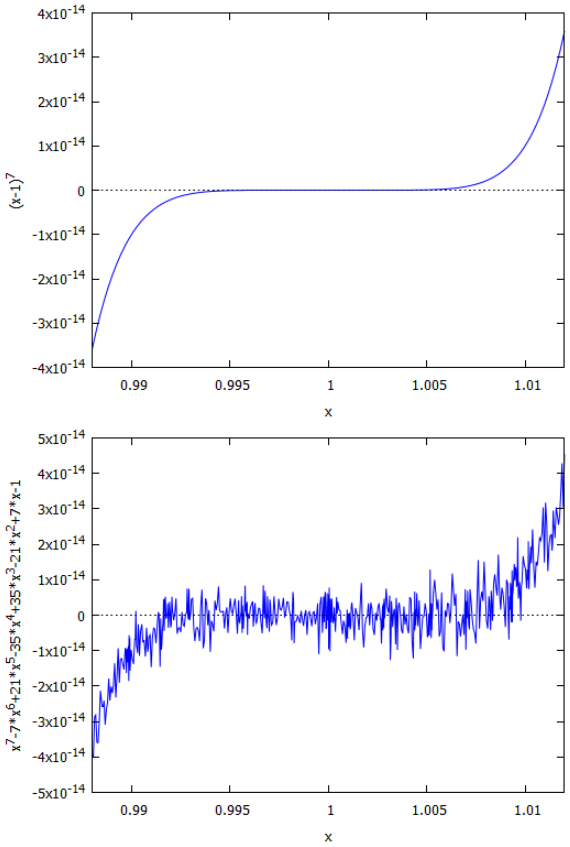
2.9 Fuentes básicas de errores

Los diversos tipos de errores que pueden aparecer al resolver numéricamente un problema son:

- Errores del modelo matemático. Son inevitables si el modelo matemático de la situación real es moderadamente simple.
- Errores producidos por la inexactitud de los datos físicos sobre los que se basa el modelo.
- Errores por equivocación humana al introducir mal los datos o programar incorrectamente
- Errores de truncamiento matemático. Muchos de los modelos matemáticos, incluso los más simples, se corresponden con procesos infinitos: integrar una función, derivar, etc. Como es imposible programarlos con un número infinito de pasos, al implementarlos en el ordenador se cometerá inevitablemente un error al sustituirlos por un número finito de pasos.
- Error de redondeo. Éste se transmite al realizar operaciones y son debidos a que los ordenadores solo guardan un número finito de cifras significativas. Además al realizar operaciones aritméticas el resultado debe ser nuevamente redondeado.

Este último hecho se puede ver muy bien gráficamente. Consideremos el polinomio $p(x) = (x\text{-}1)^7$. Lo vamos a dibujar escrito factorizado en la variable p1 y expandido en la variable p2 y, vamos a realizar la gráfica en el intervalo [0.988, 1.012]. Observamos que lo que debería de ser una gráfica suave de un polinomio tiene una forma muy angulosa y aparentemente aleatoria, a este fenómeno se le suele llamar ruido al evaluar la función y ocurre cuando el intervalo en que se dibuja la función es demasiado pequeño.

```
(%i1)    kill(all)$
(%i4)    p1:(x-1)^7;
         p2:expand(p1);
         plot2d(p1,[x,0.988,1.012])$
         plot2d(p2,[x,0.988,1.012],[nticks,100])$
(p1)     (x-1)^7
(p2)     x^7-7*x^6+21*x^5-35*x^4+35*x^3-21*x^2+7*x-1
```



Esto ocurre por la cancelación severa de cifras significativas en el cálculo y se debe a que, para x cerca de 1, al calcular p2 se restan muchos números cercanos (como $-35x^4$ y $35x^3$, etc.).

Para finalizar nos parece interesante comentar la relación que existe entre el error de truncamiento matemático y el error de redondeo.

Por ejemplo, si, para minimizar el error de truncamiento al calcular una integral, tomamos una suma con un millón de términos, además de tardar más en efectuarse el cálculo, nos encontraremos que aumenta el número de operaciones y por tanto el error de redondeo acumulado. Recíprocamente, si por disminuir el error de redondeo reducimos mucho el número de términos de la suma, la aproximación a la integral puede ser inaceptable.

Dibujemos el error cometido al aproximar la derivada de *exp*(*x*) en *x* = 1 por la fórmula:

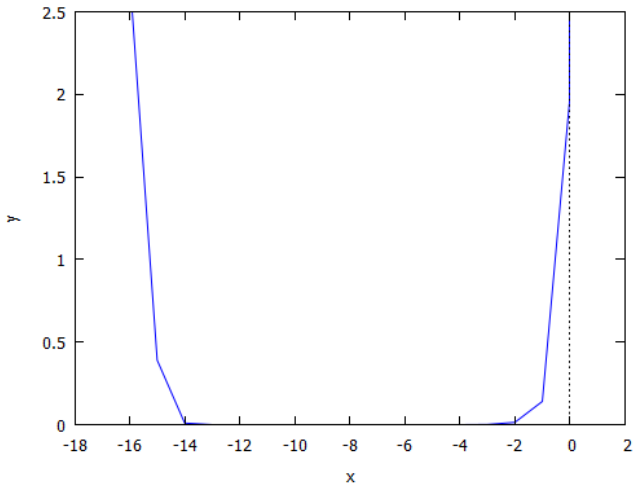
$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

Tomamos el paso *h* como 10^n y dibujamos los valores de *n* y el valor absoluto de:

$$\left| \frac{exp(1+10^n) - exp(1)}{10^n} - exp(1) \right|$$

El comportamiento de la influencia conjunta de los dos errores puede representarse, de modo intuitivo. Si el paso *h* es grande el error de truncamiento se hace grande y por tanto el error total también se hace grande. Si el paso es excesivamente pequeño el error de redondeo se hace grande y por tanto el error total aumenta. Es preciso un compromiso de forma que no se dispare ninguno de los dos errores.

```
(%i8)    kill(errder)$
         errder(n):=ev(abs((exp(1.+10^n)-exp(1))/10^n-exp(1.)),numer)$
         lis:makelist([n,errder(n)],n,-18,1)$
         plot2d([discrete,lis],[y,0,2.5])$
```



2.10 Ejercicios propuestos

Completar con Maxima los ejercicios siguientes:

Ejercicio 1. Considera los siguientes valores de p y p^*:

$$p = 0.12, \qquad p^* = 0.1$$
$$p = 0.1289725\ 10^5, \quad p^* = 0.1289705\ 10^5$$

¿Cuál es el error absoluto y el error relativo al aproximar p^* por p?

Ejercicio 2. Calcular la raíz cuadrada de 5 con 80 dígitos.

Ejercicio 3. Averigua qué número es mayor: 1000^{999} o 999^{1000}. ¿Cómo crees que es su diferencia, grande o pequeña? Calcúlala.

Ejercicio 4. Escribir un programa que dados tres números calcule el mayor. Llama al **block** maxi(*a, b, c*) y compruébalo con algún ejemplo. Mejora el programa anterior y haz que ordene los tres valores de mayor a menor.

Ejercicio 5. Utilizando un **block**, escribe una función cubo(*n*) que nos de la suma de los cubos de los *n* primeros números naturales. Comprueba el resultado con el comando **sum**.

Ejercicio 6. Construye un **block**, que llamarás *f*(*n*), que calcule la suma de los cuadrados de los *n* primeros números naturales. Haz que el mismo bloque devuelva el resultado. Incluye ahora una salida con **return** de forma que si la suma es mayor de 200 se salga de forma prematura. Haz que devuelva la suma y el valor de *n* en el que se salió.

Ejercicio 7. Utilizar un bucle **for** para poner de manifiesto la acumulación de los errores de redondeo, sumando *n* veces la fracción 1/10 pero usando su valor 0.1. Tomando *n* = 1, 10, 100, 1000, 10000, 100000, y 1000000, observar la pérdida de cifras significativas de cada resultado.

Ejercicio 8. Calcula:

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

mediante la orden **sum** y su opción **simpsum**. Busca en la ayuda si es necesario. Ahora haz tu propio **block** para calcular dicha suma, obviamente para un valor finito de *n*. Haz que muestre el tiempo que tarda y compara tu solución con la obtenida con **sum**.

Ejercicio 9. Comprueba con tu propio **block** si vale lo mismo:

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{10000}$$

si lo sumamos empezando por $\frac{1}{1}$ o si empezamos por $\frac{1}{10000}$. Usar aritmética de punto flotante y mostrar el tiempo de ejecución. ¿Cómo lo hace la orden **sum**?