**Chapter 5  Cálculo y Análisis.**

Este capítulo trata de todas las funciones dentro de la biblioteca relacionadas con el cálculo infinitesimal y el análisis matemático. Matlab no es un lenguaje de cálculo simbólico, no calcula ni derivadas ni primitivas. Todas las rutinas de este capítulo están enfocadas al cálculo numérico.

**5.1  Funciones elementales.**

Ya hemos comentado que Matlab posee una importante biblioteca de funciones, es el momento de conocerla con un poco más de detalle

**5.2  Polinomios**

En Matlab los polinomios se almacenan como vectores de coeficientes. Cada componente del vector es un coeficiente ordenado del término de mayor a menor orden. La manera más habitual de iniciar un polinomio es utilizar la siguiente función:

**poly**

Inicia un polinomio. Si el argumento es una matriz cuadrada devuelve el polinomio característico, si es un vector devuelve un polinomio cuyas raíces son los elementos del vector.

También nos serán muy útiles las funciónes:

**roots**

Devuelve las raíces del polinomio cuyos coeficientes son los elementos del vector argumento.

**polyval**

Calcula el valor del polinomio en el punto dado

**polyout**

(Octave) Escribe un polinomio en un formato más leíble:

Por ejemplo, si queremos escribir el polinomio *s*5-8*s*4+3*s*2+16:

>> polinomio = [1,-8,0,+3,0,16]

>> polyval(polinomio,44)

ans= 134937280

Los polinomios se pueden manipular analíticamente con las siguientes funciones:

**polyder**

Deriva un polinomio.

**polyint**

Integra analíticamente un polinomio.

**conv**

Multiplica dos polinomios.

**deconv**

Dados dos polinomios *a* e *y* halla *b* y *r* que cumplen:

*y*=(*a*\**b*)+*r*

**residue**

Descompone el cociente de dos polinomios en fracciones con un único polo. Por ejemplo, dados los polinomios *p*=*s*2+*s*+1 y *q*=*s*3-5*s*2+8*s*-4 la descomposición sería:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | *s*2+*s*+1 | | |  | | --- | |  | | | *s*3-5*s*2+8*s*-4 | | = | |  | | --- | | -2 | | |  | | --- | |  | | | *s*-2 | | + | |  | | --- | | 7 | | |  | | --- | |  | | | (*s*-2)2 | | + | |  | | --- | | 3 | | |  | | --- | |  | | | *s*-1 | |

**5.2.1  Evaluación de funciones.**

Los polinomios son una parte importantísima del cálculo numérico porque son su nexo fundamental con el cálculo analítico. Uno de los elementos fundamentales en el cálculo numérico es la aproximación de una función analítica por otra con una expresión numérica lo más equivalente posible. Todas las funciones analíticas no existen realmente dentro de un programa de cálculo numérico por limitaciones del propio ordenador; la aritmética exacta es algo prohibitivo. La evaluación de funciones se realiza por series de potencias almacenadas por sus coeficientes; este es el funcionamiento de la evaluación de cualquier función analítica en un programa de cálculo numérico. Las series de potencias acotan el error con el número de términos y el orden de la serie; a mayor precisión más términos a evaluar. Por ejemplo, si Matlab quiere obtener el valor de la función seno en un punto evaluará la serie:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| sin | *x*= | |  | | --- | | ∞ | | ∑ | | *k*=0 | | |  | | --- | | (-1)*k* | | |  | | --- | |  | | | (2*k*+1)! | | *x*2*k*+1 |

O si quiere evaluar la función de Bessel de primera especie:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Jn*(*x*)= | 漂缂缂輯 | |  | | --- | | *x* | | |  | | --- | |  | | | 2 | |  | |  | | --- | | *n* | |  | |  | | |  | | --- | | ∞ | | ∑ | | *k*=0 | |  | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | (- | |  | | --- | | 1 | | |  | | --- | |  | | | 4 | | *x*2)*k* | | | |  | | --- | |  | | | *k*!(*k*+*n*)! | |

Si este es el funcionamiento interno de Matlab será muy beneficioso que tengamos esto en cuenta cada vez que tengamos que evaluar funciones especialmente pesadas. No necesitamos aritmética exacta, evaluar una precisión mayor a la de la máquina es malgastar recursos.  
  
Supongamos que tenemos una función con una gran cantidad de funciones trigonométricas. Es probable que alguna vez en nuestra vida tengamos funciones de dos y tres líneas de código. Evaluarla entera es una pérdida de recursos. Imaginemos además que estemos integrando una ecuación diferencial cuyo término es esta misma función. Evaluando todos los términos no estamos ganando precisión; estamos sumando y restando polinomios de un orden mayor al que queremos con valores que se cancelan continuamente. Utilizando la interpolación polinómica podemos acelerar cualquier código hasta en un órden de magnitud.

**5.3  Derivadas**

Todas las funciones de este apartado se refieren a la derivada numérica. No son derivadas de una función analítica sino el resultado de aplicar las ecuaciones de diferencias de orden enésimo a una serie discreta de puntos.

**diff**

Aplica la fórmula de diferencias enésimas a una serie de puntos

**gradient**

Calcula el gradiente numérico de una matriz. Esta función es muy interesante por su relación íntima con la función quiver. Si queremos expresar un determinado campo vectorial a partir de su función potencial, siendo el potencial la función φ(*x*,*y*) entonces el campo es en cada punto (∂*x*φ,∂*y*φ), es decir, ∇φ. Para pintar este campo vectorial sólo tenemos que hacer quiver(gradient(phi)) .

**5.4  Integrales**

La integración numérica es un vasto campo del cálculo numérico y así se refleja en Matlab. Podemos hacer tanto integrales en una dimensión como en dos. Las rutinas más relevantes son:

**quad**

Integración numérica de una función analítica introducida a través de un archivo de función o por una sentencia inline, es en este tipo de funciones donde el inlining tiene más utilidad. Por ejemplo, para integrar ∫010exp(*x*4) *dx* haremos quad(inline('exp(-x4)'),0,10). La función nos permite introducir la tolerancia requerida para el resultado, cuando pasemos este argumento es importante que no sea mayor que la precisión numérica con la que estemos trabajando o haremos que el programa produzca un error. También podemos pasarle otros argumentos para aumentar la precisión como marcar ciertos puntos donde la función puede presentar singularidades.

**quadl**

Su uso es idéntico que la función anterior pero contiene un algoritmo de cuadratura de mayor precisión. Obviamente a mayor precisión mayor tiempo de cálculo.

**dblquad**

quad2dc y quad2dg en Octave. Calcula la integral doble de una función de dos variables. Obviamente la frontera de integración serán cuatro constantes.

**triplequad**

No existe aún en Octave. Calcula la integral de volúmen de una función de tres variables.

**trapz**

Aplica la regla del trapecio para calcular la integral numérica de una serie discreta de puntos.

**5.5  Ecuaciones diferenciales** **ordinarias.**

Se podría decir que hay un método de integración para cada problema. Existen multitud de esquemas de integración y escoger uno de ellos no es fácil. Uno de los objetivos de las funciones de Matlab es intentar ser lo más universales posible. La función perfecta sería la que sin configuración alguna integrara cualquier EDO con una precisión aceptable; desgraciadamente no existe y probablemente no exista nunca.  
  
Una consideración esencial antes intentar resolver una EDO es saber si el paso temporal viene condicionado por la precisión numérica o por criterios de estabilidad. El segundo caso es bastante más preocupante que el primero. Cuando la función introduce gradientes elevados que acortan significativamente el paso de la iteración se dice que el problema es *stiff*. En estos casos será apropiado utilizar un esquema de integración implícito que a costa de mayor coste computacional por paso permite llegar al resultado con muchas menos iteraciones. Saber si un problema es stiff o no es tan laborioso o más que resolverlo. Para saber si utilizar un esquema explícito o implícito la estrategia será la siguiente:

1. Represenataremos la función a integrar para comprobar si tiene discontinuidades o gradientes muy elevados
2. Si no tiene ninguna singularidad utilizaremos un esquema explícito (no stiff) calculando el tiempo de proceso.
3. Si el tiempo nos parece demasiado elevado cambiaremos a un esquema implícito
4. Si el tiempo sigue siendo demasiado elevado y la programación es correcta puede ser que la función sea demasiado costosa de evaluar. Para solucionarlo escribiremos la función en otro lenguaje de programación como C, C++ o Fortran y la convertiremos en un archivo que Matlab entienda.

La precisión numérica es también muy importante pero el mismo esquema de integración es capaz de controlarla. Daremos como válido que a mayor precisión numérica mayor dificultad para obtener una solución estable.  
  
Las diferencias entre Matlab y Octave en este caso son más que significativas. Octave dispone de las funciones de integración de Matlab pero existen sólo por motivos de compatibilidad, son mucho más lentas y no se pueden configurar. Sin embargo Octave integra la suite ODEPACK, uno de los paquetes de integración de EDOs más robusto y efectivo.

**5.5.1  Octave**

Octave cuenta con las funciones de integración de Matlab pero están programadas como archivos .m y no como parte de una biblioteca binaria. Es muy recomendable que utilcemos la función lsode (Livermore Solver for Ordinary Differential Equations), parte de ODEPACK. Utiliza un método multipaso Adams Para problemas no stiff y una BDF (Backward Differentiation Formula) para los problemas stiff.[1](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "note33)

**lsode**

(Octave) Es el driver general de integración de ecuaciones diferenciales ordinarias.

[X, ISTATE, MSG] = lsode (FCN, X\_0, T, T\_CRIT)

Resuelve el problema de Cauchy de la forma:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | *dy* | | |  | | --- | |  | | | *dt* | | =*f*(*x*,*t*) |

dadas las condiciones iniciales.

**FCN**

Matriz de dos elementos tipo cadena de caracteres. El primero es el nombre de la función que queremos integrar y el segundo la función que calcula el jacobiano.

La función a integrar debe definirse necesariamente mediante una función, ya sea dentro del script o en un archivo a parte. La cabecera tendrá sólo dos variables y su forma será

XDOT=FUNC(X,T)

donde tanto X como XDOT pueden ser escalares o vectores. El jacobiano también tendrá la forma

JAC=FJAC(X,T)

donde JAC será la matriz que implemente las funciones derivadas parciales según el orden establecido por la variable X. Se usa para acelerar la integración en problemas stiff.

**X\_0**

Vector de condiciones iniciales.

**T**

Vector que determina los instantes temporales en los que se dará una solución.

**T\_CRIT**

Puntos en los que la rutina no debe integrar por la existencia de singularidades.

**X**

Vector de soluciones en los instantes T

Si la integración no ha sido satisfactoria ISTATE devolverá el valor 2 y MSG nos dará más información.

**lsode\_options**

Función de configuración de lsode

lsode\_options index{lsode options} (OPT, VAL)

Llamada con un único argumento (OPT) nos dirá el valor de la opción de configuración que hayamos solicitado. Pasando dos argumentos podremos cambiarlos. Las opciones disponibles son los de la tabla siguiente:

|  |
| --- |
|  |

Table 5.1: Valores de configuración de lsode\_options

Pronto veremos un ejemplo de cómo usar esta función.

**5.5.2  Matlab**

El planteamiento de las rutinas para ODEs de Matlab es completamente distinto. En vez de tener una función que sirve más o menos para todo nos permite escojer el método de integración. Disponemos de los presentes en la tabla [5.2](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html#cap:Funciones-de-integraci=F3n):

|  |
| --- |
|  |

Table 5.2: Funciones de integración de EDOs de Matlab

La llamada básica es la misma para todas ellas y tiene la forma:

[T,Y] = solver(odefun,tspan,y0)

[T,Y] = solver(odefun,tspan,y0,options)

[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)

odefun es la llamada convencional a una función, preferiblemente un function handle. A diferencia de lsode la función debe tener la forma:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | *dy* | | |  | | --- | |  | | | *dt* | | =*F*(*t*,*x*) |

y su resultado **debe ser un vector columna**. tspan es un vector de dos componentes con el tiempo inicial y final de integración; finalmente y0 es el vector con la condición inicial. Los parámetros de salida son los usuales, T e Y son respectivamente los vectores tiempo y solución del problema. En breve propondremos un ejemplo  
  
También es necesario que conozcamos la función de configuración odeset. La cantidad de opciones es bastante mayor a las disponibles en lsode de modo que lo más sensato será consultar la ayuda.

**5.5.3  Solución de la ecuación de Van der Pol**

La ecuación diferencial de Van der Pol es:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *mx*′′-α *x*′+β | ( | *x*′ | ) | 3+*kx*=0 |

No es más que el movimiento de una masa oscilante con amortiguamiento lineal y no lineal a la vez. Adimensionalizando el problema llegamos a la ecuación.

*x*′′+*x*+?(*x*′2-1)*x*′=0

Esta ecuación tiene la particularidad de que a medida que el parámetro ? aumenta el problema se vuelve stiff. Con un ? de orden unidad el sistema puede considerarse no stiff mientras que si aumenta hasta ser del orden del millar la ecuación introduce gradientes muy acusados. Este comportamiento tan particular la hace perfecta para ensayar los distintos métodos de integración. Lo primero es descomponer la ecuación en un problema que Matlab pueda entender:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | |  | | --- | | *dy*1 | | |  | | --- | |  | | | *dt* | | =*y*2 | |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | |  | | --- | | *dy*2 | | |  | | --- | |  | | | *dt* | | =? *y*2(1-*y*12)-*y*1 | |

con condiciones de contorno *y*1(0)=2 e *y*:2(0)=0. Resolveremos el problema para ?=1 y ?=1000 porque Matlab ya cuenta con estas funciones (vdp1 y vdp1000). Para agilizar los cálculos escribiremos las funciones para Octave en C++:

#include <octave/oct.h>

DEFUN\_DLD (vdp1,args, ,

"Ecuacion de Van der Pol para mu=1 ")

{

ColumnVector xdot (2);

ColumnVector x (args(0).vector\_value());

float mu=1;

xdot(0) = x(1);

xdot(1) = mu\*x(1)\*(1-x(0)\*x(0))-x(0);

return octave\_value (xdot);

}

Y la función para ?=1000:

#include <octave/oct.h>

DEFUN\_DLD (vdp1000,args, ,

"Ecuacion de Van der Pol para mu=1000 ")

{

ColumnVector xdot (2);

ColumnVector x (args(0).vector\_value());

float mu=1000;

xdot(0) = x(1);

xdot(1) = mu\*x(1)\*(1-x(0)\*x(0))-x(0);

return octave\_value (xdot);

}

**5.5.3.1  Integración del problema no stiff (vdp1****)**

Integrar un problema no stiff es mucho más sencillo que un problema stiff. La inestabilidad numérica tiene efectos mucho más importantes que la pérdida de precisión. Los esquemas de integración que manejaremos son de paso variable. Esto significa que la propia subrutina calcula el paso de integración para, sea cual sea la función y el esquema utilizado, el resultado sea numéricamente correcto.

* Octave

>> lsode\_options('integration method','non-stiff')

>> y=lsode('vdp1',[0 2],linspace(0,20,1000));

* Matlab

>> [tout,xout]=ode45(@vdp1,[0 20],[2 0]);

La diferencia esencial entre Octave y Matlab es que el primero pide por defecto los puntos en los que queremos la solución. Matlab prefiere resolver el problema y dar los puntos en el tiempo en los que se ha pasado durante la integración. Es por ese motivo por el que el argumento ``tiempo'' es en el primer caso un vector de un millar de elementos mientras que en el segundo es tan solo un par de escalares.  
  
La superposición de ambas soluciones es un ejemplo perfecto para ver trabajar los esquemas de paso variable. Apreciamos en la figura [5.1](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html#cap:odenostiff) que el paso de integración disminuye según la curvatura de la solución

Figure 5.1: Solución de la ecuación de Van der Pol con ?=1

**5.5.3.2  Integración del problema stiff (vdp1000****)**

Como los problemas stiff introducen gradientes elevados en la solución el paso de integración de un esquema explícito se va haciendo tan pequeño que su avance es demasiado corto. Es precisamente lo que ensayamos en este caso. Si intentamos resolver la ecuación de Van der Pol con ?=1000 y *t*∈[0,3000] mediante un esquema explícito:

* Octave

>> lsode\_options('integration method','non-stiff')

>> y=lsode('vdp1000',[0 2],linspace(0,3000,100000));

* Matlab

>> [tout,xout]=ode45(@vdp1000,[0 3000],[2 0]);

nos encontramos con la desagradable sorpresa de que el calculo no termina nunca. Esto es porque la solución, aunque es regular, tiene puntos con gradiente casi infinito.  
  
La solución es utilizar un esquema implícito o semimplícito. Por ejemplo:

* Octave

>> lsode\_options('integration method','stiff')

>> ystiff=lsode('vdp1000',[2 0],linspace(0,20,1000));

* Matlab

>> [tout,xout]=ode23s(@vdp1000,[0 3000],[2 0]);

Efectivamente, la solución demuestra por qué el problema es stiff (Figura [5.2](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html#cap:odestiff)):

Figure 5.2: Solución de la ecuación de Van der Pol con ?=1000

**5.5.4  Inestabilidades y caos****.**

Uno de los casos en los que el cálculo numérico es una herramienta imprescindible es en la resolución de ecuaciones diferenciales no lineales, tanto ordinarias como en derivadas parciales. La no linealidad de las ecuaciones hace que el carácter de los problemas de evolución sea mucho más complejo que en el caso lineal. Uno de los fenómenos observados es la aparición de caos.[2](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "note34)  
  
Supongamos que queremos resolver la ecuación del dipolo de Rikitake de ecuaciones:

|  |  |  |
| --- | --- | --- |
| *x*1′ | = | -*x*1+*x*3*x*2 |
| *x*2′ | = | -*x*2+(*x*3-3.75)*x*1 |
| *x*3′ | = | 1-*x*1*x*2 |

Despues de introducirlo en la función rikitake.m:

function xdot=rikitake(x,t)

%

% Ecuacion de la dinamo de Rikitake

%

xdot(1,1)=-x(1)+x(3)\*x(2);

xdot(2,1)=-x(2)+x(1)\*(x(3)-3.75);

xdot(3,1)=1-x(1)\*x(2);

Y resolver el problema utilizando la función ode45 Octave[3](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "note35):

>> [t,x]=ode45(@rikitake,[0 300],[1 2 3],pair=0,ode\_fcn\_format=1);

plot(t,x(:,1),';x\_1;',t,x(:,2),';x\_2;',t,x(:,3),';x\_3;')

Nos encontramos con la desagradable sorpresa de que nuestra solución es completamente errática (figura [5.3](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html#cap:rikitake2d)).

Figure 5.3: Solución del dipolo de Rikitake

A primera vista puede parecer que la solución está mal. Si representamos exactamente el mismo resultado pero mediante una curva paramétrica:

>> plot3 (x(:,1),x(:,2),x(:,3))

llegamos a la figura [5.4](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html#cap:rikirake3d).

Figure 5.4: Curva solución del dipolo de Rikitake

De este ejemplo aprendemos que las ecuaciones diferenciales no lineales pueden tener un comportamiento extraño. Un ejemplo claro de ello son las ecuaciones de Navier Stokes. Si simplificamos las ecuaciones para flujo incompresible nos encontraremos un término lineal (el viscoso) y uno no lineal (el convectivo). El estudio analítico de estas ecuaciones aún no ha llegado a la conclusión sobre la existencia y la unicidad de la solución[4](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "note36), sin embargo se resuelven por métodos numéricos.  
  
La turbulencia, uno de los fenómenos físicos más complejos, aparece de forma espontanea cuando intentamos resolver las ecuaciones de N-S. Parte de la disciplina de la CFD (Computational Fluid Dynamics) es estudiar los flujos turbulentos a partir de la resolución numérica de las ecuaciones de N-S.  
  
La conclusión es que la integración numérica de ecuaciones diferenciales es un problema complejo no sólo por la física del problema sino porque además se suman muchos otros factores. En esta pequeña introducción sólo hemos visto los problemas stiff y un caso muy sencillo de caos pero los problemas pueden complicarse muchísimo más.

**5.6  Cálculo simbólico**

La afirmación repetida hasta la saciedad de que Matlab es un programa *estrictamente* de cálculo matricial no es del todo correcta. Se basa en la aplicación del sentido crítico a una herramienta. La versión completa de Matlab incluye el programa de cálculo simbólico más utilizado actualmente, Maple. Los problemas generados por el uso del cálculo simbólico en Matlab son dos:

1. Tenemos acceso al núcleo de cálculo de Maple, pero dicho acceso no es completo. Podemos efectuar las operaciones básicas como derivar, buscar primitivas, crear desarrollos de Taylor... Nunca dispondremos de la potencia de este tipo de programas.
2. La arquitectura de matlab **no está pensada para el cálculo simbólico**. Los programas de cálculo simbólico suelen incluir un interfaz llamado *notebook* que sirve para ver las fórmulas de entrada y salida en una notación mucho más matemática. Las diferencias en el diseño de un programa de cálculo numérico y otro de cálculo simbólico son a mi parecer irreconciliables.

Esto no significa que sea un crimen utilizar las funciones de cálculo simbólico cuando uno disponga de ellas. Somos libres de hacer lo que queramos pero debemos ser conscientes de hasta qué punto no estamos utilizando la herramienta ideal para nuestro fín.  
  
Octave también dispone de un soporte muy limitado para realizar operaciones simbólicas. Se podría decir que lo único que puede hacer es derivar y resolver sistemas de ecuaciones lineales. Tampoco parece que en un futuro cercano estas funcines se amplíen hasta un soporte equivalente al de Matlab. No son compatibles pero las funciones son tremendamente parecidas.  
  
Los siguientes apartados sirven como introducción general al uso de variables simbólicas pero no conseguiremos más que arañar la superficie del total de las posibilidades del toolkit.

**5.6.1  Definición de variables funciones simbólicas**

*Una de las diferencias más esenciales entre Matlab y Octave en lo que respecta a cálculo simbólico es que en Octave es necesario activar el toolkit con el siguiente comando*:

>> symbols

Tanto en Matlab como en Octave el modo más fácil de definir una variable simbólica es crear una variable que contenga un argumento simbólico del mismo nombre:

**sym**

Define una variable que contiene un argumento simbólico. Es muy recomendable que la variable y el argumento tengan el mismo nombre para evitar confusiones

>> x = sym('x')

**poly2sym**

Define un polinomio simbólico a partir de sus coeficientes.

>> poly2sym([1 2 3 -2 10])

ans =

10.0+(-2.0+x\*(3.0+(2.0+x)\*x))\*x

**5.6.2  Funciones simbólicas elementales.**

La aplicación de funciones elementales a variables simbólicas es completamente distinta en Matlab y en Octave. Mientras Matlab no diferencia las funciones simbólicas de las numéricas Octave prefiere llamarlas de un modo distinto. Esta distinción suele ser que la primera letra de la función es una mayúscula[5](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "note37). Entonces, para conseguir la función simbólica sin*x* en Matlab haremos:

>> x=sym('x')

x =

x

>> y=sin(x)

y =

sin(x)

Mientras que en Octave haremos lo siguiente:

>> x=sym('x')

x =

x

>> y=Sin(x)

y =

sin(x)

En la colección podemos encontrar todas las funciones trigonométricas, exponenciales, logaritmos...

**5.6.3  Operaciones simbólicas**

Casi todas las operaciones disponibles en el paquete simbólico de Octave están dedicadas a la manipulación simbólica de polinomios (muy interesante en el cálculo numérico). En cambio Matlab dispone de la mayoría de las operaiciones simbólicas existentes... Derivadas, primitivas, desarrollos en serie... La única función interesante en lo referente a cálculo simbólico que tiene Octave es la siguiente:

**subs**

Substituye una o varias variables de una expresión simbólica.

**collect**

Agrupa los términos polinómicos en una variable dada.

**expand**

Desarrolla una expresión simbólica.

**differentiate**

Sólo en Octave. Calcula la n-ésima derivada de una expresión simbólica.

Función que en Matlab recibe un nombre distinto:

**diff**

Sólo en Matlab. Calcula la n-ésima derivada de una expresión simbólica.

Matlab dispone además de las siguientes operaciones:

**int**

Encuentra la primitiva de una función simbólica

**limit**

Calcula el límite hacia un determinado valor de una función simbólica.

**taylor**

Calcula el desarrollo en serie de Taylor respecto a un punto dado de una función simbólica.

**symsum**

Calcula la suma de dos series.

Una de las pocas operaciones compatibles es la representación de una curva determinada por una función simbólica:

**splot**

Dibuja la curva representación de una función simbólica de una variable. El segundo argumento es un vector de dos elementos con el intervalo de la variable independiente.

[1](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "text33)

Personalmente creo que éste es uno de los pocos casos en el que Octave es superior a Matlab. Ni entiendo ni comparto la organización de las rutinas de integración de Matlab, me parece confusa, desacertada y poco óptima.

[2](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "text34)

Esta sección es sólo una pequeña introducción a los efectos del caos. No soy un absoluto un experto en el tema pero se comenta en esta sección porque puede tener efectos desconcertantes en la solución. Pido disculpas si existe algun fallo garrafal en alguna de las afirmaciones.

[3](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "text35)

¿Por qué hemos tenido que utilizar el argumento adicional ode\_fcn\_format?

[4](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "text36)

El estudio de existencia y unicidad de la solución de las ecuaciones de Navier Stokes para el caso incompresible es uno de los grandes problemas no resueltos de las matemáticas.

[5](http://torroja.dmt.upm.es/%7Eguillem/matlab/cursolatex008.html" \l "text37)

Siempre viene bien recordar que la arquitectura de Matlab distingue entre mayúsculas y minúsculas