Introducción a Matlab y Octave

Release 0.1

Guillem Borrell i Nogueras

September 11, 2010

Índice general

Prólogo

Esta es una breve introducción al lenguaje de programación Matlab orientada a alumnos que no han asistido nunca a un curso de programación. El único requisito para seguir este manual es una mente abierta. Matlab es útil y sencillo pero está lleno de sutilezas. Cualquiera puede escribir programas sencillos con Matlab después de un par de horas de práctica pero debemos tener siempre en cuenta que programar bien implica ser consciente de los detalles. Se necesita una mente abierta para entender que hacer las cosas bien es siempre importante.

Versión 0.1

Fecha September 11, 2010

Más información en http://iimyo.forja.rediris.es

CAPÍTULO 1

Introducción

Tras encender Matlab la sensación puede ser de saturación. La interfaz gráfica de Matlab no se corresponde a la sencillez de uso real del programa. Al final terminaremos usando un par de cosas e ignorando el resto. Por ahora nos interesan sólo dos herramientas: la consola y el editor.

El editor nos servirá para escribir o modificar los programas y la consola será nuestra vía principal de comunicación con Matlab. Cualquiera de las operaciones de la interfaz gráfica pueden realizarse únicamente escribiendo comandos en la consola. De momento es en lo único que debemos centrarnos, durante esta primera introducción bastará cone escribir comandos detrás del símbolo >>.

1.1 Fundamentos de programación

Nuestro punto de partida es así de simple:

```
>> a = 1;
```

Hay tres elementos en esta línea de código:

- `a` es una variable
- = es el operador asignación
- `1` es el literal que define el número 1.

Una variable es una palabra cualquiera. La única restricción es que no podemos utilizar unos pocos caracteres reservados como +, − o ⋆. Debemos escoger siempre nombres bien descriptivos que permitan descifrar el algoritmo que se está implementando. Las velocidades pueden llamarse `v` y las coordenadas `x` e `y` respectivamente.

El operador asignación almacena en memoria el resutado de la operación de su derecha y la asigna (por eso su nombre) a la variable. En Matlab cada comando sólo puede obtener un operador asignación y a su izquierda sólo puede haber una variable.

Nota: Otros lenguajes de programación como C++ permiten la asignación en cualquier estructura de código. Por ejemplo

```
#include <iostream>

int main(int argc, char *argv[])
{
  int c;
  if(c = 1 > 0){
    std::cout << "Mayor que cero" << std::endl;
  }
  std::cout << c << std::endl;
```

```
  return 0;
}
```

Tiene la siguiente salida por pantalla:

```
Mayor que cero
1
```

Esta estructura es sencillamente imposible en Matlab.

Ahora tenemos un mecanismo para almacenar cualquier resultado independientemente de su naturaleza en la memoria del ordenador. Esta es la esencia de la programación: calcular a partir de datos almacenados en la memoria. Ahora tenemos muchas preguntas por contestar. ¿Qué estamos calculando realmente? ¿Cómo se almacenan los resultados en la memoria? etc. Todas estas preguntas encontrarán respuesta más tarde.

Importante: Matlab distingue entre letras mayúsculas y minúsculas en los nombres de las variables.

Para ayudarnos en las sesiones interactivas Matlab define una variable especial llamada `ans` y que no debemos sobreescribir nunca. Cada vez que ejecutemos un comando y no asignemos su resultado a una variable Matlab hará dicha asignación de manera completamente automática a la variable `ans`.

```
>> 2+2
ans =
      4
```

1.2 Matlab es un lenguaje de programación

Matlab es un lenguaje de programación, un conjunto de reglas para escribir programas de ordenador. Matlab es un lenguaje de programación orientado al Cálculo Numérico (de ahí su nombre Matrix Laboratory) y es difícil encontrarle cualquier otra aplicación. Desde un punto de vista estético y práctico Matlab es un buen lenguaje de programación para realizar programas breves y simples. Matlab no es adecuado para:

- Implementación de algoritmos complejos que requieran de modelos de datos complejos organizados de forma jerárquica. Aunque con Matlab podemos programar utilizando la orientación a objetos no puede considerarse un buen lenguaje para ello.
- Computación de alto rendimiento. El HPC es un caso de uso extremo de los recursos de cálculo. Matlab tiene un rendimiento razonable en la mayoría de los casos pero un buen programador puede multiplicar entre diez y cien veces la velocidad de ejecución de un programa utilizando C o Fortran.
- Grandes proyectos de software. Matlab no es una buena elección para un programa que crece más allá de unos cuantos miles de líneas. No hay una razón única para ello pero se podría decir que la complejidad del código escala mal.

Pero lo realmente valioso de Matlab no son sus capacidades como lenguaje sino las siguientes:

- Existe un uso generalizado de Matlab en Ingeniería, es una herramienta de gran popularidad y es útil para una carrera profesional. Esto lo ha convertido en un estándar de-facto para la escritura de pequeños programas de simulación.
- Matlab cuenta con una extensa biblioteca de funciones que cubren casi todas las disciplinas de la Ciencia y la Ingeniería extensamente documentada y de fácil uso.

1.3 Matlab es un lenguaje interpretado

Los lenguajes de programación, como los lenguajes naturales escritos, no son más que una serie de normas para transmitir conceptos. Mientras el lenguaje escrito sirve para que los seres humanos se comuniquen entre ellos los lenguajes de programación se crearon para comunicarse con los ordenadores mediante una serie finita de claves.

Los lenguajes de programación también tienen gramática y léxico pero son mucho más simples que, por ejemplo, los de la lengua castellana. Los seres humanos estamos educados para convertir palabras y frases en sonidos. Hay que dotar a los ordenadores de un método para convertir el código implementado en un lenguaje de programación en órdenes que sea capaz de cumplir. Hay casi una infinidad de maneras de lograr este objetivo. A diferencia de la mayoría de los cursos sobre lenguajes de programación los describiremos por orden cronológico, aunque no rigurosamente.

Cuando apareció el ordenador programable la única manera de comunicarse con él era describir sin ambigüedad qué sucedía con cada posición de memoria. Este código de bajo nivel, llamado comúnmente ensamblador, es traducido a lenguaje máquina que ya un ordenador es capaz de entender. Aunque hoy este método de programación pueda parecer inverosímil es la mejor manera de mover máquinas lentas y con poca memoria como las de entonces.

El paso siguiente llegó con la aparición de los compiladores. A medida que los ordenadores se hacían más potentes escribir los programas en ensamblador empezó a hacerse una tarea muy laboriosa. El número de direcciones de memoria crecía exponencialmente y las arquitecturas, aunque seguían el modelo de Von Neumann, se hacían más complejas. El siguiente paso fue utilizar el mismo ordenador para traducir desde un lenguaje más humano, de alto nivel, a ensamblador. El ensamblador pasó de ser un lenguaje de uso a un léxico intermedio. El programa que convierte este código de alto nivel se llama compilador.

Este planteamiento tiene una ventaja adicional. El código ensamblador no es el mismo para todas las arquitecturas. Un programa compilado para x86 no puede ejecutarse en SPARC o POWER pero el código es el mismo. El programa de Kernighan y Ritchie [KnR]

```
#include "stdio.h"

int main()
{
  printf("Hello, world!\n");
}
```

Produce exactamente el mismo resultado en cualquier ordenador siempre que disponga de un compilador de lenguaje C. Esto asegura la portabilidad a nivel de código, no a nivel de ejecutable.

El paso siguiente es poder utilizar un ensamblador independiente de cada arquitectura mediante un traductor de código propio a código máquina. Esta aplicación se llama *máquina virtual*. Una máquina virtual es tan lista como se desee (mucho más lista que un procesador) y realizará tareas como la declaración de variables, la liberación de memoria o la gestión del flujo de ejecución. El conjunto compilador y máquina virtual se denomina intérprete y los lenguajes que soportan este funcionamiento se llaman *lenguajes interpretados*. Que el código sea ejecutado por un programa y no por el propio ordenador es mucho más lento, por este motivo las máquinas virtuales no se popularizaron hasta finales de los noventa.

El paso siguiente es hacer desaparecer incluso este ensamblador intermedio y con él el compilador. Ya no existe un compilador y una máquina virtual sino que sólo un programa, el intérprete, realiza todo el trabajo. Este último planteamiento no es necesariamente superior en eficacia o rendimiento a una máquina virtual, simplemente es más fácil de diseñar e implementar. Matlab pertenece a este último grupo.

1.4 Matlab es un lenguaje dinámico

En muchos lenguajes de programación como C o Fortran es imprescindible declarar cada variable. La definición estricta de declaración es la de identificar un determinado espacio en la memoria del ordenador con un nombre. Volviendo otra vez a un C que cualquiera pueda entender la declaración

```
int a;
```

significa que un espacio en la memoria física lo suficientemente grande como para almacenar un entero va a recibir el nombre de `a`. Estos lenguajes, los que asocian variables a memoria, se llaman *estáticos*

La llegada de los lenguajes interpretados permitió manejar la memoria de una manera mucho más versátil. Java, que aunque es interpretado es también estático, incluye un recolector de basura que descarga al programador de la tarea de limpiar la memoria. Pero la mayoría de los lenguajes interpretados modernos como Python o Ruby son

además *dinámicos*. En un lenguaje dinámico no existen declaraciones porque el concepto de variable es distinto, *ya no es el nombre que se asocia a un espacio en la memoria, es el nombre de un valor*. De esta manera la variable tiene un sentido mucho más natural, más matemático. Matlab es un lenguaje dinámico aunque no puede considerarse moderno.

Desde el punto de vista del intérprete cualquier variable o estructuras de variables son mutables en tiempo de ejecución complicando significativamente el manejo de memoria.

Programar con un lenguaje dinámico es completamente distinto hacerlo con uno estático. La mayor versatilidad suele venir acompañada de mayor coste computacional o de nuevos errores de programación. No debemos perder nuca de vista que la programación es la manipulación de datos almacenados en la memoria de un ordenador y con un lenguaje dinámico estamos más lejos de los mismos.

1.5 El intérprete Octave para el lenguaje de programación Matlab

Cuando consideramos Matlab un lenguaje de programación la razón de ser de Octave se hace obvia. Muchos desarrolladores querían utilizar el lenguaje Matlab pero o bien no podían permitirse el coste de una licencia o no estaban dispuestos a utilizar software propietario. Octave no es exactamente un intérprete para el lenguaje Matlab porque es un objetivo móvil, cambia en cada versión y muchas de las funcionalidades deben entenderse por ingeniería inversa. Una diferencia que sí se mantendrá durante mucho tiempo es que, mientras Matlab es un entorno de desarrollo integrado, Octave es sólo un intérprete y necesitaremos otras herramientas para hacerlo verdaderamente funcional.

Octave cuenta con un grupo de desarrolladores entusuasta y una enorme comunidad de usuarios. Si tenéis algún problema utilizando Octave recomiendo encarecidamente darse de alta en la lista de correo. Podéis encontrar más información en http://www.octave.org. Octave funciona en prácticamente cualquier sistema operativo mayoritario como Windows, Linux, MacOS X, Solaris...

Nota: Octave está ganando importancia dentro de entornos grid y en el *cloud computing*. En un entorno grid todos los recursos están abstraídos de manera que el usuario no sabe en realidad dónde está ejecutando cada tarea; es el middleware el que decide cuál es el entorno de ejecución más adecuado. Esto significa que debe haber una licencia de Matlab por cada tarea en grid que lo requiera, algo que puede estar fuera del alcance de la infraestructura por motivos de coste. Octave representa una alternativa a Matlab en estos entornos.

1.6 Lenguajes de programación modernos

Los ordenadores lo han cambiado todo. Fuerno inventados para ayudarnos en tareas repetitivas pero ahora forman parte de cada aspecto de nuestra vida. El primer ordenador que se instaló en España fue un mainframe IBM para calcular declaraciones de hacienda. Ahora hay más teléfonos móviles que habitantes. Pero un ordenador es algo vacío sin software, y cada línea de código ha sido programado en un lenguaje de programación.

Hay cientos de lenguajes de programación pero sólo unos pocos llegan a ser populares. Quizás habéis oído hablar alguna vez de C, C++ o Java. Pero hay muchos más: Python, Ruby, Perl, Erlang, Lua, C#, Fortran, Haskell, Effiel, Smalltalk, Javascript, Ocaml, Ada... Todos ellos tienen miles de usuarios. Hablemos de alguno de ellos.

Google utiliza sólo cuatro lenguajes de programación: C++, Java, Javascript y Python, quizás no conozcáis el último. Python es quizás el lenguaje de programación más consistente y simple. Es directo, fácil de aprender y con todas las posibilidades que se esperan de un lenguaje de programación moderno: orientación a objetos, modularidad, iteradores, una enorme librería estándar... Se dice que Python es tan simple que nunca debería ser el primer lenguaje de programación de nadie: luego el resto parecen demasiado difíciles. Por último y no menos importante: es software libre.

Fortran fue el primer lenguaje de programación y es aún una herramienta común en Ciencia e Ingeniería. Desde su creación a finales de los cincuenta ha visto como una media docena de revisiones, el último estándar es Fortran 2008. Desde el gremio de la informática muchos programadores tildan a Fortran de un lenguaje obsoleto. Quien lo diga probablemente no haya usado Fortran en su vida.

Primer Contacto

2.1 La interfaz gráfica de Matlab

La interfaz gráfica de Matlab es prácticamente idéntica en cualquiera de sus versiones independientemente del sistema operativo.

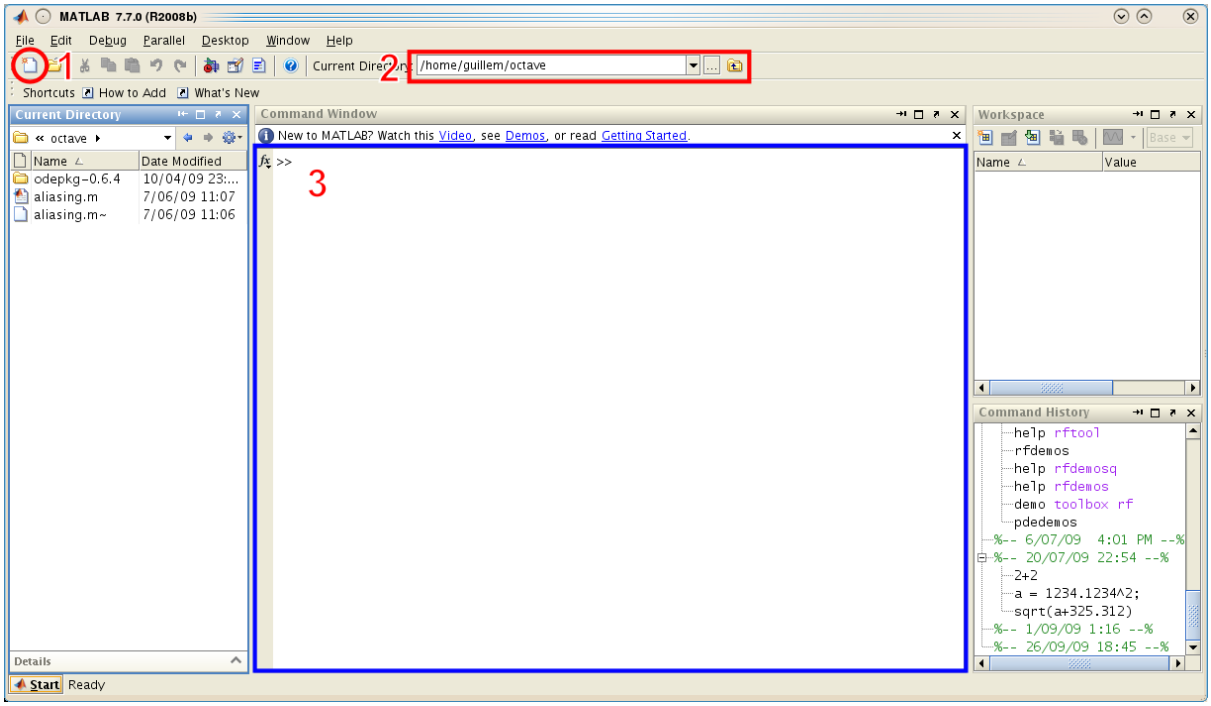

Figura 2.1: Captura de la interfaz gráfica de Matlab R2008b

Vemos que la ventana principal está dividida en apartados con una función específica. Cada uno de estos apartados, a excepción del menú, es una ventana que puede moverse dentro de la propia aplicación. Esto permite que ordenemos Matlab para ajustarlo mejor a nuestras necesidades. Las tres únicas zonas que de momento nos interesan están marcadas con un número en la imagen.

El icono señalado con el número 1 siginfica nuevo archivo y sirve para abrir el editor de Matlab. Será nuestra herramienta de trabajo y pronto le dedicaremos una sección.

El recuadro señalado con el número 2 es el diálogo para seleccionar el directorio de trabajo. A medida que vayamos escribiendo código lo guardaremos en algún lugar del ordenador. Para poder utilizarlos en un futuro es importante

que Matlab sepa dónde lo hemos dejado. Matlab ya sabe, porque así viene configurado de fábrica, dónde tiene guardadas las funciones propias de la aplicación y de los distintos toolkits pero no sabe dónde están las que hemos escrito.

> **Advertencia:** Matlab busca funciones y scripts en los directorios especificados por la función `path`. El primero de ellos es siempre el especificado en el diálogo `Current Directory`.

`path` (*path, dir*)
: Sin argumentos imprime en la pantalla los directorios donde Matlab busca los archivos. En el caso de darle dos argumentos, normalmente el primero será simplemente `path` mientras que el segundo será el nombre de un directorio que queramos añadir a la lista.

  Por ejemplo, para añadir un directorio en un sistema operativo UNIX

```
>> path(path,'/home/yo/funciones_matlab')
```

  Para añadir un directorio en un sistema Windows

```
>> path(path,'c:\yo\funciones_matlab')
```

Por último, pero no menos importante, el número 3 es la consola de Matlab. Como quedó claro en la introducción, en realidad Matlab no es más que un intérprete para un lenguaje de programación y nuestra vía directa de comunicación con el mismo es la consola. De hecho, no existe ninguna acción de la interfaz gráfica que no pueda ejecutarse también mediante la consola. De hecho, cuando ejecutamos un programa no es ni siquiera imprescindible que la interfaz gráfica esté abierta.

Truco: Uno de los atajos de teclado más útiles del editor de matlab es utilizar la tecla F5 para guardar y ejecutar el código que estamos escribiendo.

La siguiente pieza es el editor.

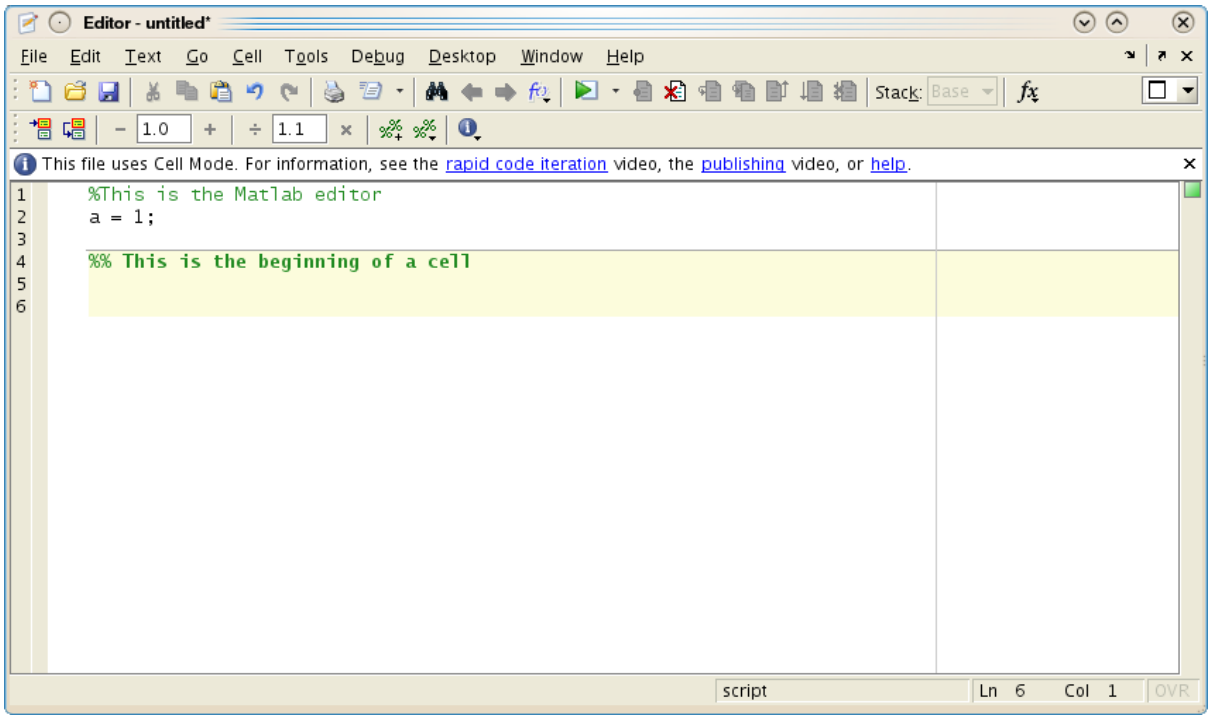

Figura 2.2: Captura del editor de Matlab R2008b

La definición de programar es escribir código y para ser realmente productivos es importante utilizar una buena herramienta y conocerla. No es ni mucho menos necesario utilizar el editor de Matlab para escribir nuestros scripts pero se trata de una buena opción.

El editor cuenta con casi todas las capacidades que se esperan de una herramienta de programación moderna.

- Coloreado de código
- Análisis sintáctico capaz de detectar errores antes de ejecutar el código
- Depurador integrado

Una de las características que ha integrado en las últimas versiones es el modo celda que nos ayudará a dividir grandes archivos en partes ejecutables independientemente sólo comentando el código de una manera particular.

La interfaz gráfica nos sirve también para consultar la documentación del programa. Es completa, extensa y de calidad. Hablaremos con más calma sobre la ayuda de Matlab en el siguiente capítulo.

2.2 La arquitectura de Matlab

Por motivos de licencias, Matlab está dividido en paquetes. Cada uno cumple una función específica y puede ser adquirido a parte. Esto impone una limitación añadida a Matlab porque, aunque una empresa o una universidad se haya gastado grandes cantidades de dinero en licencias de Matlab, es posible que no haya adquirido el toolbox que necesitamos.

2.2.1 Simulink

Simulink es una herramienta de diseño y modelado de sistemas dinámicos. Simulink utiliza Matlab para realizar los cálculos, puede extenderse con Matlab y se distribuye junto con Matlab, pero no es Matlab. Simulink se basa en conectar modelos, expresados por bloques, que se transmiten información.

Simulink tiene sus limitaciones. No siempre un sistema se puede llegar a modelar de manera eficiente sólo con bloques y conexiones debido a que no siempre la información transmitida es equivalente a la información que pasa por un cable. Nunca debe presentarse Simulink como una alternativa a la programación directa de un modelo sino como una plataforma de modelado de sistemas simples o de integración para que varios ingenieros trabajen sin colisionar en el mismo sistema.

2.3 Octave

En su propia documentación se describe Octave como un lenguaje de programación de alto nivel orientado al Cálculo Numérico. Proporciona una consola para resolver problemas lineales y no lineales con el ordenador y para desarrollar experimentos numéricos.

Octave puede ser copiado, modificado y redistribuído libremente bajo los términos de la licencia GNU GPL tal como se publica por la Free Software Foundation.

Octave fue diseñado para ser una herramienta dentro de la línea de comandos del sistema operativo GNU, aunque posteriormente ha sido portado a muchos más sistemas operativos. También en un principio fue un lenguaje de programación independiente pero ha ido convergiendo a Matlab hasta el punto de buscar la compatibilidad con él. Tampoco ha sido nunca un objetivo dotarle de interfaz gráfica pero podemos encontrar ya un par de ellas con calidad suficiente.

Aunque Octave es capaz de ejecutar la mayoría del código escrito en Matlab tanto su historia como su arquitectura interna es completamente distinta. Una de las diferencias más evidentes es que están escritos en lenguajes de programación distintos, Matlab en C y Octave en C++.

Octave es hoy en día una herramienta inferior a Matlab pero para tratarse de algo totalmente gratuito desarrollado por una comunidad de ingenieros, científicos y entusiastas se trata de una herramienta de una calidad altísima. Para pequeños proyectos es una alternativa completamente viable a Matlab además cuenta con la ventaja de utilizar el mismo lenguaje de programación. Otras plataformas de cálculo para Ciencia e Ingeniería como Scilab o IDL cuentan con sus propios lenguajes de programación.

2.3.1 QtOctave

Se trata de la interfaz gráfica más completa disponible para Octave en la actualidad. No es parte de Octave sino que se trata de un proyecto independiente y separado. Al igual que Octave se trata de software que puede copiarse, modificarse y distribuirse siempre que se haga respetando la licencia GNU GPL.

Al igual que Matlab proporciona acceso a la consola y un editor. Aunque aún no dispone de depurador integrado sí proporciona un sistema de control de versiones que puede resultarnos útil cuando el código que escribimos se acerca a los millares de líneas.

2.4 Nuestro primer programa en Matlab

Antes de escribir código o implementar algún algoritmo es necesario que nos familiaricemos con el entorno de desarrollo. Este primer programa constará de una función y de un script que llama a la función. Así construiremos una estructura de programa que se repite en la gran mayoría de casos; nuestros programas serán una colección de funciones que son llamadas por un script que funcionará como un programa principal.

Nota: El lector que haya programado alguna vez en C o cualquier lenguaje basado en C como C++ o Java reconocerá esta manera de trabajar. La diferencia es que en Matlab no hacen falta cabeceras de ningún tipo y el programa principal puede llamarse de cualquier manera. La limitación es que, al no poder crear cabeceras, todas las funciones deberán encontrarse ya en los directorios especificados por `path`.

Abriremos el editor y en él escribiremos lo siguiente

```
aprsin = @(x) x - x.^3/6;
x = linspace(-pi,pi,100);
plot(x,aprsin(x),x,sin(x));
```

Guardaremos el script con el nombre que más nos apetezca, siempre con la extensión `.m`. Luego, en la consola, escribiremos el nombre que hemos utilizado para el archivo sin la extensión. Si todo ha salido bien aparecerá lo siguiente.

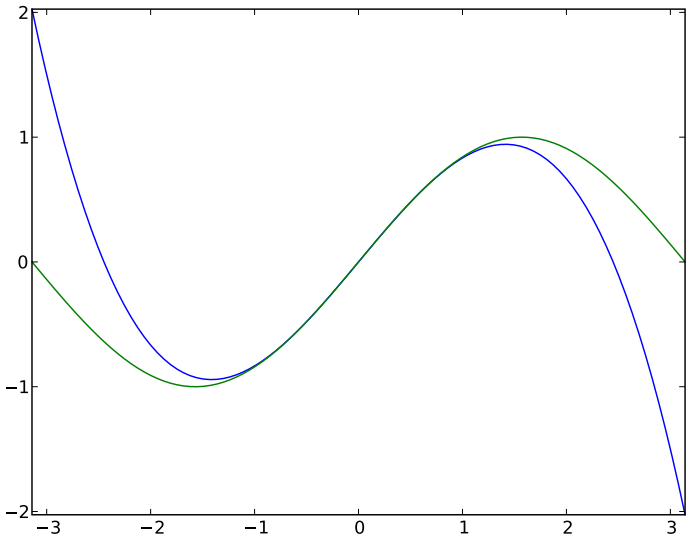

Figura 2.3: Resultado del script

2.5 Nuestro primer programa en Octave

A diferencia de Matlab, Octave es programa diseñado para ser utilizado en la consola del sistema. Dicho de esta manera parece que volvemos a los años 80 antes que se popularizara Windows pero si nos fijamos un poco en la interfaz de Matlab veremos que a medida que nos volvamos más hábiles en el uso del lenguaje de programación usaremos más el intérprete de comando y menos los accesorios que lo rodean.

En Octave uno de los comandos más usados es edit, que también existe en Matlab.

edit()
: Función que controla el editor asociado al intérprete. En el caso de Matlab se trata del intérprete propio mientras que Octave utiliza el intérprete predeterminado del sistema. Por ejemplo, para editar la función nueva `aprsin.m` escribiremos

```
>> edit aprsin.m
```

CAPÍTULO 3

Escalares, vectores y polinomios

El siguiente paso en cualquier curso de programación es dar una visión general de las características del lenguaje. Tratar todas las sentencias, operadores, estructuras... Esto, a parte de ser terriblemente aburrido, no es necesariamente útil en estos casos. Nuestro objetivo es dominar las habilidades básicas para ser capaces de resolver elegantemente problemas simples con Matlab. Será más adecuado ir aprendiendo el lenguaje sobre la marcha

3.1 Scripts y sesiones interactivas

Debemos acostumbrarnos a ir escribiendo nuestro trabajo en el editor, esto es, crear programas (también llamados guiones o scripts) y ejecutarlos a través del intérprete. Una pregunta recurrente de quien empieza con Matlab y lleva un rato utilizando la consola. ¿Cómo puedo guardar todo mi progreso? La respuesta es que nunca deberías haber hecho nada importante con la consola.

La consola, con su línea de comandos, sirve para operaciones simples y para interactuar con nuestros scripts. Cuando escribimos código que de verdad queremos guardar debemos hacerlo en un editor. El primer paso es entender cómo funcona el editor de Matlab, o por lo menos un editor que sea se lleve bien con Matlab.

No necesitaremos el editor en este breve tutorial pero estáis avisados. Aprenderemos más sobre editores, scripts y atajos de teclado en un rato.

3.2 Operaciones aritméticas básicas

Podemos utilizar Matlab como una calculadora asombrosamente potente para realizar operaciones sencillas y para aplicar funciones elementales. Una suma es tan sencilla como podría serlo

```
>> 2 + 2
ans =
      4
```

Recordad que las variables sirven para almacenar resultados en la memoria y volverlos a utilizar

```
>> a = 4;
>> a + 2;
ans =
      6
```

Importante: Por omisión Matlab siempre muestra el resultado del último cálculo. Esto sucede tanto en sesiones interactivas como en los programas que escribamos en el editor. Para prevenir la salida de una ristra interminable de resultados intermedios debemos escribir un punto y coma al final de cada línea.

Los operadores matemáticos básicos se expresan en Matlab mediante los siguientes símbolos:

- Suma: +.
- Resta: −. El signo menos también sirve como prefijo para expresar que un número es negativo.
- Multiplicación: `.*`.
- División: `./`.
- Potencia: `.^`.

Importante: Probablemente os sorprenda el hecho que los últimos tres operadores utilicen un punto y no sean simplemente el símbolo. La razón la entenderemos en el momento en el que empecemos a operar con matrices.

Hay muchos más operadores aritméticos, de momento nos hemos ceñido a los más básicos.

Matlab es capaz de operar con números complejos gracias a que el número imaginario i es una constante expresada por la variable `i`. Cualquier número multiplicado por `i` será en realidad la componente imaginaria de un número complejo. Por ejemplo

```
>> a = 1;
>> b = 1.*i;
>> a + b
ans = 1 + 1i
>> a .* b
ans = 0 + 1i
```

Advertencia: Matlab no mostrará ningún aviso en el caso que sobreescribamos `i`. Para evitar posibles accidentes podemos utilizar símbolos alternativos para expresar la constante imaginaria: `j`, `I` y `J`.

Lo que convierte a Matlab en una herramienta útil es la enorme biblioteca de funciones que cubre prácticamente cualquier disciplina del cálculo numérico, desde el Álgebra Lineal al análisis de señales pasando por la teoría de juegos o las redes neuronales. Cualquier función tiene argumentos de entrada y argumentos de salida y Matlab los trata de una manera un poco particular. Las funciones más simples tienen sólo un argumento de entrada y otro de salida

```
>> sin(1.4)
ans = 0.98545

>> sqrt(4)
ans = 2
```

Como no hemos asignado el argumento de salida a ninguna variable Matlab ha utilizado la variable especial `ans` de la que hemos hablado en el capítulo anterior. Hay funciones que tienen varios argumentos de entrada y de salida como por ejemplo la función `quad` que calcula la integral numérica de una función en un intervalo dado. `quad` tiene cinco argumentos de entrada y cuatro de salida y es prácticamente imposible que la utilicemos correctamente sin consultar la documentación. Hacerlo es tan sencillo como escribir lo siguiente en el intérprete

```
>> help(quad)
```

Acabamos de aprender el nombre de *la función más importante de Matlab*, `help`. Todas las funciones de la biblioteca de Matlab están perfectamente documentadas y tenemos acceso a esa información a través de `help`.

Siempre que sea matemáticamente consistente cualquier función operará indistintamente con números reales y complejos:

```
>> a = 1.6;
>> b = 3.4.*i;
>> exp(b)
ans = -0.96680 - 0.25554i
```

Ejercicio 1

Define tres variables con los siguientes valores: a = 1.5, b = 3.4 y c = 5.2. Calcula el valor de d para $d = \frac{a}{\frac{b}{c^a} - \frac{c}{b^a}}$

Ejercicio 2

En un Congreso Internacional de Matemáticas se votó como la fórmula más bella $e^{i\pi} = -1$. Comprueba que Matlab piensa que esta fórmula es correcta. Te conviene utilizar la constante `pi`.

Ejercicio 3

Comprueba que el producto de dos números complejos es igual al producto de sus módulos y la suma de sus argumentos. Puede ser que necesites las funciones `angle` y `abs`.

Ejercicio 4

No existe el infinito en Matlab porque sirve para el Cálculo Numérico, no para el Cálculo Simbólico. Pero hay una constante propia llamada `Inf` que es un número lo suficientemente grande como para ser el infinito en la práctica (es un número más grande que el total de átomos de la masa conocida del Universo). La función tangente, `tan` conecta el valor de π con el infinito: $\infty = \tan(\frac{\pi}{2})$. Si utilizamos la expresión anterior para calcular el infinito en Matlab no llegamos a un número tan grande. ¿Puedes dar una explicación?

3.3 Definición de funciones

Ahora ya sabemos operar con escalares y con funciones simples. El siguiente paso es aprender a definir nuestras propias funciones. Hay dos maneras de definir una función en Matlab, de momento nos basta con la más sencilla y a la vez la menos potente: mediante el operador `@()`. La sintaxis queda bien clara mediante el siguiente ejemplo:

```
>> fsin = @(x) x - x.^3/6
fsin =

@(x) x - x .^ 3 / 6
>> fsin(pi)
ans =  8.3093
```

Una función definida por el usuario puede hacer uso tanto de otras funciones independientemente de su origen.

```
>> comp = @(x) fsin(x) - sin(x)
comp =

@(x) fsin (x) - sin (x)
>> comp(0.1)
ans = 3.3325e-004
```

Nota: Técnicamente lo que hemos definido antes no es exactamente una función y de hecho no se llama función sino *función anónima*. Pero de momento no encontraremos ninguna diferencia.

3.4 Vectores

El vector es el tipo derivado más simple de Matlab. Se trata de una concatenación de números ordenados en fila. La característica más importante de los vectores es que son un conjunto ordenado del que podemos tomar uno o varios de sus elementos a partir de su índice.

La manera más sencilla de definir un vector es utilizando un literal:

```
>> v = [11,12,13,14,15,16,117,18,19]
v =

   11   12   13   14   15   16   17 ...
```

Podemos obtener un elemento del vector llamándolo como si fuera una función

```
>> v(2)
ans  =  12
```

Obtener porciones del vector es tan fácil como obtener elementos. Basta con separar el primer ínidice del último con dos puntos

```
>> v(2:4)
ans =

   12   13   14
```

También podemos utilizar otro vector para obtener un vector con elementos individuales

```
>> v([2,4,6,7])
ans =

   12   14   16   17
```

Difíclmente escribiremos nunca un vector largo en Matlab. O lo obtenderemos como dato o utilizaremos una función específicamente diseñada para ello como `linspace` o `logspace`.

`linspace`(*base, limit, N*)
: Devuelve un vector fila con *N* elementos separados linealmente entre *base* y *limit*

  Ejemplo:

  ```
  >> linspace(0,10,11)
  ans =

      0  1  2  3  4  5  6  7  8  9  10
  ```

`logspace`(*base, limit, N*)
: Similar a `linspace` excepto que los valores están espaciados logarítmicamente entre 10^{base} y 10^{limit}.

Cuando un vector se opere con un escalar se operará con cada uno de los elementos del vector.

```
>> v = [1,2,3,4];
>> 3+v
ans =

   4   5   6   7
>> 3.*v
ans =

    3    6    9   12
```

Si los dos operandos son vectores el requisito fundamental es que ambos tengan el mismo tamaño.

```
>> w = [8,7,6,5];
>> v+w
ans =

   9   9   9   9
>> v.*w
ans =

    8   14   18   20
```

Importante: No existe una multiplicación de vectores, la operación anterior es operar los vectores elemento elemento, lo que corresponde más a una tabla que a lo que se esperaría de un vector. De hecho en Cálculo Numérico no hay ninguna diferencia entre un vector y una simple lista de números.

Una operación importante cuando se habla de vectores es el producto escalar, que se define como.

$$u \cdot v = \sum_i u_i v_i \tag{3.1}$$

En Maltab puede calcularse con la función `dot`.

dot (*u, v, dim*)
Calcula el producto escalar de dos vectores. El tercer argumento, *dim* es de utilidad en el caso que alguno de los dos argumentos o ambos sean matrices.

```
>> dot(v,w)
ans =  60
```

Aunque sea mucho menos eficiente también podemos calcular ese producto escalar utilizando la definición de la operación y la función `sum`.

sum (*x, dim*)
Suma los elementos de un vector. *dim* es de utilidad cuando el argumento sea una matriz.

```
>> sum(v.*w)
ans =  60
```

Advertencia: En muchos programas escritos en Matlab encontraremos el producto escalar escrito como

```
>> u'*v
```

Es una operación válida, aunque aún no sepamos qué operaciones son el apóstrofe y el asterisco sin punto respectivamente. El problema de no utilizar la función `dot` es que estamos utilizando una sintaxis ambigua, no sabemos si `u` y `v` son vectores, además de ser una opreación mucho más propensa a fallar sin dar excesiva información del porqué. Recordad que la belleza es importante.

prod (*x, dim*)
Calcula el producto de los elementos de un vector. *dim* es de utilidad cuando el argumento sea una matriz.

Una técnica importante en Matlab es la concatenación de dos vectores que puede hacerse simplemente pegándolos

```
>> a = [1,2,3];
>> b = [4,5,6];
>> [a,b]
ans =

   1   2   3   4   5   6
```

o utilizando la función `cat`.

Ejercicio 5

Cuando Gauss contaba siete años el profesor les puso un ejercicio para tenerlos entretenidos un rato. ¿Cuánto es la suma de todos los números enteros entre 1 y 100? Gauss llegó fácilmente al reultado en sólo unos pocos segundos porque vio que sumando pares de números 1+99, 2+98, 3+97... La operación podía reducirse a $50 \times 99+100$. Con Matlab se puede hacer la operación por fuerza bruta de muchas maneras pero... ¿Eres capaz de hacerlo con sólo una línea de código?

Ejercicio 6

El número pi puede calcularse mediante la siguiente serie:

$$\frac{\pi^2 - 8}{16} = \sum_{n=1}^{\infty} \frac{1}{(2n-1)^2(2n+1)^2}$$

¿Cuántos términos son necesarios para llegar a una precisión de 10^{-12}? ¿Cuánta es la precisión de la suma de 100 términos?

3.5 Polinomios

Se define un polinomio de grado *n* como

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} + a_n x^n \tag{3.2}$$

No es más que una función en la que el valor de la variable se eleva sucesivamente a una potencia hasta *n* y se multiplica por una constante. Utilizando el símbolo del sumatorio la expresión anterior puede compactarse a:

$$p_n(x) = \sum_{i=0}^{n} a_i x^i$$

Si nos fijamos un momento en la expresión (3.2) observaremos que un polinomio puede expresarse fácilmente en forma de vector utilizando sus coeficientes. El orden puede deducirse fácilmente con el número de coeficientes. Matlab utiliza vectores para expresar los polinomios con la única salvedad que los almacena del modo inverso al que hemos escrito (3.2). El polinomio $x^3 - x + 1$ sería en Matlab

```
>> p = [1, 0, -1, 1];
```

La operación más común con un polinomio es evaluarlo en un punto dado, para ello utilizaremos la función `polyval`.

polyval (*p, x*)
Evalúa el polinomio *p* en el punto *x*

Ejemplo

```
>> p = [1, 0, -1, 1];
>> polyval(p,3)
ans =  25
```

La importancia de los polinomios es que, siendo una función, todas las operaciones elementales (suma, resta, multiplicación y división) pueden reducirse sólo a operaciones con sus coeficientes. De esta manera podemos convertir operaciones simbólicas en operaciones puramente numéricas. Tomemos por ejemplo estas dos funciones: $p(x) = 4x^3 - x$ y $q(x) = x^2 + 6$. Sumar y restar estas dos funciones es trivial, pero no multiplicarlas. Como se trata de una operación con coeficientes Matlab la hará sin inmutarse

```
>> p = [4, 0, -1, 0];
>> q = [1, 0, 6];
>> conv(p,q)
ans =
    4    0   23    0   -6    0
```

conv (*u, v*)
Calcula la convolución de dos vectores de coeficientes. En el caso de vectores, la convolución es la misma operación que el producto.

Efectivamente $p(x) * q(x) = 4x^5 + 23x^3 - 6x$.

Dividir dos polinomios nos servirá para aprender cómo tratar las funciones con dos argumentos de salida. De define la división de dos polinomios como

$$p(x) = q(x) * c(x) + r(x)$$

Entonces la división entre $p(x)$ y $q(x)$ tiene como resultado dos polinomios más, el cociente $c(x)$ y el residuo $r(x)$. Si a la salida de `deconv` se le asigna sólo una variable obtendremos el cociente

deconv (*u, v*)
Calcula la deconvolución de dos vectores de coeficientes. En el caso de polinomios esta operación es equivalente al cociente del primero por el segundo.

Devuelve dos argumentos, el cociente y el residuo.

```
>> c = deconv(p,q)
c =
    4   0
```

Si necesitamos también el residuo tendremos que hacer lo siguiente

```
>> [c,r] = deconv(p,q)
c =
    4   0

r =
    0   0  -25   0
```

Hay otras operaciones que son operadores lineales aplicados a los polinomios como por ejemplo la derivada y la integral.

polyderiv(*p*)
: Calcula los coeficientes de la derivada del polinomio *p*. Si le proporcionamos un segundo argumento *q* calculará la derivada del producto de polinomios.

polyinteg(*p*)
: Calcula los coeficientes de la primitiva del polinomio *p*.

Sabemos que los polinomios de orden *n* tienen el mismo número de raíces. Esto nos permite descomponer cualquier polinomio de la siguiente manera:

$$p_n(x) = \sum_{i=0}^{n} a_i x^i = \prod_{i=0}^{n} (x - r_i)$$

roots(*p*)
: Calcula las raíces del polinomio *p*.

Las raíces no son sólo importantes como solución de la ecuación $p_n(x) = 0$ sino que sirven, por ejemplo, para buscar factores comunes entre dos polinomios. Otra función bastante útil para los que utilizan Matlab para el análisis de sistemas dinámicos lineales es la función residue que calcula la descomposición en fracciones parciales del cociente de dos polinomios

residue(*p, q*)
: Calcula la descomposición en fracciones parciales del cociente de dos polinomios *p* y *q* donde el primero es el numerador y el segundo el denominador.

  Por ejemplo

```
>> b = [1, 1, 1];
>> a = [1, -5, 8, -4];
>> help residue
>> [r,p,k,e] = residue(b,a)
r =

  -2.0000
   7.0000
   3.0000

p =

   2.00000
   2.00000
   1.00000

k = [](0x0)
e =
```

```
1
2
1
```

El vector *r* es el numerador de cada uno de los términos, el vector p son los polos del sistema y el vector e es la multiplicidad de cada uno de los polos. Entonces la descomposición en fracciones parciales será:

$$\frac{s^2+s+1}{s^3-5s^2+8s-4} = \frac{-2}{s-2} + \frac{7}{(s-2)^2} + \frac{3}{s-1}$$

3.6 Ejercicio de síntesis

Existe una manera de representar la forma de una función cualesquiera en un punto dado mediante un polinomio. Dicho polinomio converge con mayor orden en los alrededores del punto a medida que se van añadiendo términos. Se trata del desarrollo de Taylor.

La única información que necesitamos de la función es su valor y el de sus derivadas en el punto dado x_0. La expresión general es

$$p_n(x-x_0) = f(x_0) + \sum_{i=1}^{n} f^{(i)}(x_0)\frac{(x-x_0)^i}{i!}$$

Para entender mejor cómo este polinomio se ajusta a la función podemos utilizar el desarrollo de la función exponencial en $x = 0$.

$$e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + \frac{1}{120}x^5 + \mathcal{O}\left(x^6\right)$$

Este polinomio puede crearse de muchas maneras pero posiblemente la más sencilla sea utilizar los polinomios en Matlab para tener que generar sólo los coeficientes.

```
>> exp_serie = @(x,n) polyval(1./[factorial(linspace(n,1,n)),1],x)
exp_serie =

@(x, n) polyval (1 ./ [factorial(linspace (n, 1, n)), 1], x)
```

Nota: Esta línea de código sirve para aprender una regla muy importante sobre cómo debemos escribir un programa. Las líneas demasiado largas son difíciles de leer, por lo tanto son un peligro incluso para nosotros mismos. Es recomendable romperlas en algún punto donde romperíamos una operación matemática: después de un operador, justo después de abrir un paréntesis. Para hacerlo debemos escribir tres puntos `...`.

Podemos utilizar esta función para entender de un modo mucho más visual el concepto de convergencia de una serie. Sabemos que a medida que añadamos términos el error que comete el desarrollo de Taylor cerca del punto se reduce. ¿Pero de qué forma? Una confusión habitual es pensar que al aumentar orden del desarrollo aumenta la región donde se acerca a la función pero esto sólo es cierto accidentalmente. Sólo existe una mayor convergencia cerca del punto.

Para verlo mejor calcularemos el error de la aproximación en los puntos 0.2 y 0.1 para distintos órdenes.

```
exp_serie = @(x,n) polyval(1./[factorial(linspace(n,1,n)),1],x)

x_01 = [exp_serie(0.1,1),
        exp_serie(0.1,2),
        exp_serie(0.1,3),
        exp_serie(0.1,4),
        exp_serie(0.1,5),
        exp_serie(0.1,6),
        exp_serie(0.1,7)];
```

```
x_02 = [exp_serie(0.2,1),
        exp_serie(0.2,2),
        exp_serie(0.2,3),
        exp_serie(0.2,4),
        exp_serie(0.2,5),
        exp_serie(0.2,6),
        exp_serie(0.2,7)];


disp('error en 0.1')
err_01 = abs(exp(0.1)-x_01)
disp('error en 0.2')
err_02 = abs(exp(0.2)-x_02)

disp('logaritmo del error en 0.1')
logerr_01 = log(err_01)
disp('logaritmo del error en 0.2')
logerr_02 = log(err_02)
```

El programa anterior tiene la siguiente salida:

```
error en 0.1
err_01 =

   5.1709e-03
   1.7092e-04
   4.2514e-06
   8.4742e-08
   1.4090e-09
   2.0092e-11
   2.5091e-13

error en 0.2
err_02 =

   2.1403e-02
   1.4028e-03
   6.9425e-05
   2.7582e-06
   9.1494e-08
   2.6046e-09
   6.4932e-11

logaritmo del error en 0.1
logerr_01 =

   -5.2647
   -8.6743
  -12.3683
  -16.2837
  -20.3804
  -24.6307
  -29.0137

logaritmo del error en 0.2
logerr_02 =

   -3.8442
   -6.5693
   -9.5753
  -12.8009
  -16.2070
```

```
-19.7660
-23.4577
```

Podemos ver que si tomamos logaritmos la diferencia entre los valores permanece aproximadamente constante.

Matrices y Álgebra Lineal

Para seguir avanzando hacia el Álgebra Lineal es necesario definir el concepto de Matriz en Matlab. Técnicamente no hay ninguna diferencia entre vectores, matrices o tensores. De hecho todos los tipos numéricos en Matlab son arrays sin distinción, cada dimensión que no exista no es más que un índice que se mantiene en 1. Para entenderlo mejor podemos hacer este pequeño experimento

```
>> a = 1;
>> a
a =  1
>> a(1)
ans =  1
>> a(1,1)
ans =  1
>> a(1,1,1)
```

Hemos definido un escalar y lo hemos llamado como un escalar, un vector, una matriz y un array tridimensional. A Matlab le ha dado igual que en su definición pretendiéramos crear un escalar.

Desde este punto de vista, todos los vectores son en realidad matrices que tienen sólo una fila o una columna. La concatenación de vectores fila o vectores columna generará una matriz. El inconveniente es que hasta ahora sólo conocemos vectores fila.

Si en un vector fila los elementos se separan con comas (o con espacios) para generar un vector columna debemos separar los elementos por puntos y comas.

```
>> v = [1;2;3]
v =

   1
   2
   3
```

Como además hemos aprendido que para concatenar vectores sólo tenemos que pegarlos ya podemos generar matrices pegando vectores fila o vectores columna. Por ejemplo:

```
>> [v,v,v]
ans =

   1   1   1
   2   2   2
   3   3   3
```

Acabamos de crear nuestra primera matriz. Matlab dispone de un literal para las matrices que nos ahorra tener que escribir un vector para cada fila o columna. En este literal los elementos de una misma fila se separan con comas y las filas se separan mediante puntos y coma como se ve en el ejemplo siguiente:

```
>> u = [1,2,3;4,5,6;7,8,9]
u =

   1   2   3
   4   5   6
   7   8   9
```

Advertencia: Uno de los grandes defectos de Matlab es la ambiguedad al tratar vectores fila y matrices con una única columna. Esto es debido a que, por omisión, un vector con un único índice es un vector fila mientras que el primer índice de una matriz cuenta los elementos de una misma columna.
Para ejemplificar este problema crearemos el vector *u* y el vector *v* de la siguiente manera:

```
>> u(3) = 1
u =

   0   0   1

>> v(3,1) = 1
v =

   0
   0
   1
```

Si Matlab fuera consistente estas dos instrucciones deberían generar el mismo vector, sin embargo en la primera generamos un vector fila y en la otra un vector columna. Para agravar los efectos de la inconsistencia ambos vectores pueden utilizar la misma notación de índices:

```
>> u(3)
ans =  1
>> v(3)
ans =  1
```

La única manera de no cometer errores graves por culpa del hecho que Matlab está mal pensado es recordar que existe un tipo *vector* y un tipo *matriz* o *array* que no tienen absolutamente nada que ver aunque Matlab sí sea capaz de operar entre ellos porque considera que un *vector* es una *matriz* con una única fila:

```
>> u*v  %Esto es lo mismo que el producto escalar
ans =  1
```

4.1 Rutinas de creación de matrices

Al igual que con los vectores, Matlab dispone de una ingente colección de funciones que, combinadas adecuadamente, nos serviran para generar prácticamente cualquier matriz.

zeros (...)
Crea una matriz con las medidas solicitadas llena de ceros.

La función zeros se puede llamar de muchas maneras. Si sólo se utiliza un índice crea una matriz cuadrada de dimensiones el valor del argumento de entrada. Con dos argumentos creará una matriz del tamaño $n \times m$ siendo *n* el primer argumento y *m* el segundo. Entonces, para crear un vector fila o un vector columna deberemos hacer lo siguiente:

Ejemplo

```
>> zeros(3,1)
ans =

   0
   0
   0

>> zeros(1,3)
ans =

   0   0   0
```

ones (...)
Crea una matriz con las medidas solicitadas llena de unos. Su funcionamiento es análogo al de `zeros`

eye (...)
Crea una matriz con unos en la diagonal principal y ceros en el resto de sus elementos. Su funcionamiento es análogo al de `zeros`.

rand (...)
Crea una matriz cuyos elementos son números aleatorios. Su funcionamiento es análogo al de `zeros`.

Es importante recordar que, al igual que los vectores, cualquier matriz puede juntarse con otra simplemente pegándolas.

```
>> [rand(3),zeros(3)]
ans =

   0.80283   0.71353   0.73322   0.00000   0.00000   0.00000
   0.00527   0.07266   0.73062   0.00000   0.00000   0.00000
   0.73262   0.93908   0.77822   0.00000   0.00000   0.00000
```

Otra función útil para generar matrices es la función `reshape`.

reshape (*A, m, n, ...*)
Devuelve una matriz con dimensiones dadas a partir de los elementos de la matriz A. En el caso que la matriz resultado no tenga el mismo número de elementos que la origen la función dará un error.

Ejemplo

```
>> reshape([1,2,3,4],2,2)
ans =

   1   3
   2   4
```

4.2 Operaciones con matrices

Los operadores de suma, resta, multiplicación, división y potencia también funcionan con matrices siempre que sean del mismo tamaño. También podemos aplicar las funciones elementales a matrices, lo que nos dará el mismo resultado que si hubiéramos aplicado la función a cada uno de los elementos. Por ejemplo

```
>> exp(eye(4))
ans =

   2.7183   1.0000   1.0000   1.0000
   1.0000   2.7183   1.0000   1.0000
   1.0000   1.0000   2.7183   1.0000
   1.0000   1.0000   1.0000   2.7183
```

Pero en el caso de las matrices existen operaciones propias como la multiplicación matricial o la inversa. Estas operaciones también tienen limitaciones: la multiplicación matricial exige que los tamaños de los operandos sean compatibles y sólo las matrices cuadradas no singulares tienen inversa. Caso aparte son las divisiones matriciales puesto que tenemos dos.

Advertencia: La confusión entre operaciones escalares y matriciales es el error más común en Matlab. Es tan recurrente que incluso programadores con varios años de experiencia lo cometen una y otra vez. Para evitarlo en la medida de lo posible es recomendable utilizar, en vez de los operadores que veremos a continuación, las funciones que realizan la misma operación.

4.2.1 Multiplicación matricial

Existen dos maneras de multiplicar matrices en Matlab; la más utilizada es el operador multiplicación matricial `*`, el mismo operador que para la multiplicación escalar pero sin el punto. La otra es la función `mtimes`

mtimes (*x*, *y*)
: Multiplica las matrices *x* e *y* siempre que sus dimensiones sean compatibles, esto es, la traspuesta de *y* debe tener exactamente el mismo número de filas y columnas que *x*. Es equivalente a `x*y`.

En código existente en Matlab veremos pocas veces la función `mtimes`. Históricamente siempre se ha tendido a la brevedad, sin embargo evitar errores transparentes es importante. Un error transparente es un error no evidente viendo los resultados del código paso a paso. Un caso de error transparente es confundir la multiplicación matricial con la escalar con matrices cuadradas. Por ejemplo

```
>> x = rand(3);
>> y = rand(3);
>> x*y
ans =

   0.50380   1.42800   0.79806
   0.92682   1.45590   1.43060
   0.52361   0.90870   0.82197

>> x.*y
ans =

   0.210474   0.435326   0.274738
   0.055776   0.279980   0.101831
   0.457864   0.282493   0.252486
```

La diferencia entre las dos matrices no es evidente. Si nuestro resultado dependiera de este cálculo sería prácticamente imposible descubrir el error a no ser que sospechemos precisamente de esta operación.

4.2.2 División matricial

Existen dos tipos de división matricial aunque las dos operaciones tienen poco que ver. La división de un número puede definirse a partir de la multiplicación invirtiendo uno de los factores. Por ejemplo

$$\frac{x}{y} = xy^{-1}$$

a su vez

$$\frac{y}{x} = x^{-1}y$$

Si nos fijamos en la parte derecha de las dos ecuaciones esto nos podría servir para introducir otro operador de división. En el caso de la primera ecuación, en la que se invierte el segundo operando, estamos delante de la división usual. El número que se invierte es el segundo. Pero también podríamos tratar el segundo caso como una división en la que el operando que se invierte es el primero. Matlab también tiene un operador para eso. En este caso tenemos una división *a derechas* y una división *a izquierdas*.

```
>> mrdivide(2,3)
ans =  0.66667
>> mldivide(2,3)
ans =  1.5000
```

mrdivide (*x, y*)
Calcula la división *a derechas* de dos argumentos

mldivide (*x, y*)
Calcula la división *a izquierdas* de dos argumentos

Estas dos funciones también tienen su equivalente en operador. La división *a derechas* se expresa mediante el símbolo `/`, mientras que la división *a izquerdas* se expresa con el símbolo `\`.

Truco: Hay una regla mnemotécnica sencilla para recordar qué operador corresponde a qué operación. *A derechas* o *a izquierdas* se refiere qué argumento es el que se invierte. En `mrdivide` se invierte el de la derecha, mientras que en `mldivide` se invierte el de la izquierda. Si vemos los dos operadores, distinguiremos el concepto de *derecha* e *izquierda* mirando hacia dónde apunta el operador en dirección ascendente. `/` apunta hacia la derecha, mientras que `\` apunta a la izquierda.

Aunque en escalares estas dos divisiones tienen poco sentido con escalares sí lo tienen si los dos operandos son matrices.

$$\frac{A}{B} = AB^{-1}$$

a su vez

$$\frac{B}{A} = A^{-1}B$$

Pero de todas las operaciones la más importante es la resolución de sistemas de ecuaciones lineales. En cualquier caso estos sistemas de ecuaciones pueden ponerse en forma matricial como

$$y = Ax$$

La solución de este sistema de ecuaciones implica que hay que realizar una división matricial.

$$x = A^{-1}y$$

Llegaremos a la solución utilizando una división *a izquierdas*. Lo más interesante de este operador es que hace bastantes más cosas de las que creemos. $A^{-1}y$ es la inversa de una matriz por un vector, pero no es estrictamente necesario calcular la inversa, se puede resolver directamente el sistema de ecuaciones. Matlab tiene esto en cuenta y aplicará un algoritmo distinto dependiendo de las características de la matriz. Incluso funcionará con un sistema mal condicionado o con una matriz no cuadrada, en tal caso dará una solución minimizando el error cuadrático (pseudoinversa)

Ejercicio 7

Tres planos en el espacio tridimensional tienen las siguientes ecuaciones.

$$\begin{aligned} x - y + z &= \sqrt{2} \\ y + z &= 1 + \sqrt{2} \\ x + y &= 1 + \sqrt{2} \end{aligned}$$

Demostrar que tienen un único punto de intersección y encontrarlo resolviendo el sistema de ecuaciones.

4.2.3 Potencia de matrices

Al igual que con el resto de operaciones aritméticas básicas disponemos de una función y un operador para la potencia de una matriz. Esta operación sólo tiene sentido para matrices cuadradas, para cualquier otra matriz dará un error.

mpow (*X, y*)
Eleva la matriz *X* a la *y* ésima potencia. Es equivalente a utilizar el operador `^`.

4.2.4 Traspuesta y conjugada

Otro de los errores recurrentes si se trabaja con números complejos es confundir el operador traspuesta con el operador traspuesta conjugada.

transpose(*X*)
Calcula la traspuesta de la matriz *X*. Es equivalente a `X.'`.

ctranspose(*X*)
Calcula la traspuesta conjugada (adjunto) de la matriz *X*. Es equivalente a `X'`.

Cuando las matrices sean únicamente de números reales ambas operaciones serán equivalentes pero confundirlos en el caso de números complejos puede ser un error difícil de encontrar.

Truco: Como hemos visto, existe el riesgo real de confundir operaciones escalares y matriciales, lo que puede generar errores catastróficos difíciles de solucionar. Un truco útil para depurar estos errores es sustituir las operaciones matriciales por las funciones equivalentes correspondientes: `mpow`, `transpose`, `mldivide`...

4.3 Ejercicio de Síntesis

Si volvemos a la definición de polinomio

$$p_n(x) = \sum_{i=0}^{n} a_i x^i$$

Uno de los problemas ante los que podemos toparnos es el de encontrar el polinomio que pasa por una serie de puntos dados. Un polinomio depende de los coeficientes que deciden, de este modo necesitamos tantos puntos como coeficientes tenga el polinomio. También podemos tomar esta conclusión a la inversa. El polinomio que pasa por *n* puntos tendrá como mínimo órden *n-1*.

Podemos enunciar el problema como sigue. Dados *n* puntos $(x, y)_n$ encontrar el polinomio de orden *n-1* que pasa por los puntos dados.

El problema se resuelve planteando una ecuación por cada punto. Si tomamos el polinomio $p_n(x)$ podremos plantear *n* ecuaciones de la forma $p_n(x_i)$. Por ejemplo, para (x_0, y_0)

$$p_n(x_0) = a_0 + a_1 x_0 + a_2 x_0^2 + \ldots + a_{n-1} x_0^{n-1} + a_n x_0^n = y_0$$

Si hacemos lo mismo para todos los puntos llegamos a un sistema de *n* ecuaciones con *n* incógnitas, los coeficientes del polinomio a_i.

El paso siguiente es expresar el sistema de ecuaciones en forma matricial:

$$\begin{bmatrix} x_n^n & \ldots & x_n^2 & x_n & 1 \\ x_{n-1}^n & \ldots & x_{n-1}^2 & x_{n-1} & 1 \\ \vdots & & & \ddots & \vdots \\ x_1^n & \ldots & x_1^2 & x_1 & 1 \\ x_0^n & \ldots & x_0^2 & x_0 & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} y_n \\ y_{n-1} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix}$$

La matriz de este sistema de ecuaciones es la matriz de Vandermonde. Podemos crear esta matriz en Matlab mediante la función `vander`

vander(*c*)
Función que genera la matriz de Vandermonde

$$\begin{bmatrix} c_n^n & \ldots & c_n^2 & c_n & 1 \\ c_{n-1}^n & \ldots & c_{n-1}^2 & c_{n-1} & 1 \\ \vdots & & & \ddots & \vdots \\ c_1^n & \ldots & c_1^2 & c_1 & 1 \\ c_0^n & \ldots & c_0^2 & c_0 & 1 \end{bmatrix}$$

a partir del vector *c*

Ahora supongamos que queremos el polinomio que pasa por los puntos (1,2), (2,1), (3,4), (4,3), (5,0)

```
>> x = linspace(1,5,5)';
>> y = [2,1,4,3,0]';
>> p = vander(x)\y
p =

    0.41667
   -5.50000
   24.58333
  -42.50000
   25.00000

>> polyval(p,1)
ans =  2.0000
>> polyval(p,2)
ans =  1.00000
>> polyval(p,3)
ans =  4.0000
>> polyval(p,4)
ans =  3.0000
>> polyval(p,5)
ans = -7.8160e-14
```

4.4 Ejercicio propuesto

Un proceso para encriptar un mensaje secreto es usar cierta matriz cuadrada con elementos enteros y cuya inversa es también entera. Se asigna un número a cada letra (A=1, B=2... espacio=27; sin eñe y con 0 como relleno al final) y se procede como sigue a continuación. Supongamos que la matriz es:

$$A = \begin{pmatrix} 1 & 2 & -3 & 4 & 5 \\ -2 & -5 & 8 & -8 & 9 \\ 1 & 2 & -2 & 7 & 9 \\ 1 & 1 & 0 & 6 & 12 \\ 2 & 4 & -6 & 8 & 11 \end{pmatrix}$$

y el mensaje que queremos enviar es `ALGEBRA LINEAL`. Para codificar el mensaje romperemos la cadena de números en vectores de cinco elementos y los codificaremos.

- ¿Cuál será la cadena de datos encriptada que enviaremos?
- Descifrar el mensaje: 161, 215, 291, 375, 350, 21, 14, 28, 4, 9, 26, 161, 102, 167, 61, 120, -2, 232, 263, 252, 55, -103, 104, 96, 110

Control de Flujo de Ejecución

En la introducción hemos incidido en el hecho que Matlab es un lenguaje de programación pero aún no hemos visto cómo se implementan algunas de las características que todos los lenguajes tienen en común. Todos los lenguajes permiten controlar el flujo de la ejecución del programa con bucles y condicionales. En esta sección aprenderemos las particularidades de Matlab al respecto

5.1 Iteradores

En los manuales de Matlab escritos con poco cuidado siempre se trata la sentencia `for` como un método para generar bucles en la ejecución del programa. Rigurosamente hablando se trata de un iterador.

La manera más habitual de utilizar la sentencia es como sigue:

```
for i = 1:5
  disp(i)
end
```

```
     1
     2
     3
     4
     5
```

Esto parece un bucle así que aún no entenemos muy bien en qué se diferencia un bucle de un iterador. La parte más importante de un bucle es un índice que se va incrementando a medida que el flujo de ejecución entra en él. Si el ejemplo anterior fuera un bucle cada vez que la ejecución pasa por la sentencia `for` la variable `i` se incrementaría en una unidad seguiría corriendo. Pero no es esto lo que sucede

```
for i = 1:5
  disp(i)
  i=i+5;
end
```

```
     1
     2
     3
     4
     5
```

Hemos incrementado el índice manualmente pero la sentencia `for` ha asignado el nuevo número al índice cada vez que el flujo de ejecución ha pasado por él. *La sentencia `for` itera el índice `i` por la secuencia `1:5` *.

Lo entenderemos aún mejor con el siguiente ejemplo

```
for i = [1,4,3,2,9]
  disp(i)
end
```

```
     1
     4
     3
     2
     9
```

Le hemos pasado a la sentencia `for` un iterable, en este caso un vector, y el índice ha ido avanzando por el mismo hasta el final. En otros lenguajes de programación podemos definir iterables para simplificar el control de flujo pero en Matlab sólo podemos iterar sobre vectores.

También podemos iterar sobre matrices pero sin contradicción con la afirmación anterior. Lo único que hará Matlab será avanzar el índice por cada una de las columnas.

5.2 Condicionales

El otro elemento esencial del control de flujo en cualquier lenguaje de programación es la ejecución condicional de bloques de código. En Matlab, como en cualquier lenguaje de programación creado por una mente mínimamente sana, la palabra clave correspondiente es `if`. A continuación un breve ejemplo

```
a = zeros(4,4);
for i = 1:4
  for j = 1:4
    if i==j
      a(i,j) = 2;
    elseif j == 4
      a(i,j) = -1;
    else
      a(i,j) = 0;
    end
  end
end
```

```
a =
   2     0     0    -1
   0     2     0    -1
   0     0     2    -1
   0     0     0     2
```

Este fragmento de código es bastante autoexplicativo.

Representación Gráfica

La representación de cualquier serie de datos es uno de los puntos fuertes de Matlab. Dispone de funciones para representar series de puntos, superfícies, curvas de nivel... Prácticamente cualquier cosa puede representarse gráficamente en Matlab aunque aquí nos centraremos en los comandos más simples

6.1 Curvas en el plano

La necesidad más simple de la representación gráfica es disponer de dos series de datos x e y y representar en el plano la serie de datos siendo x la coordenada del eje de abcisas e y la coordenada del eje de ordenadas. Esta operación se lleva a cabo mediante la función `plot` independientemente de la complejidad de los datos.

`plot`(...)
: Representa series de datos en el plano. Las posibilidades de uso de esta función son infinitas y la ayuda, que no se reproducirá aquí, es de las más extensas del lenguaje.

Para que nos hagamos una idea aproximada de la potencia del comando plot sirvan estos tres ejemplos.

```
>> plot([1,2,3,2,1]);
```

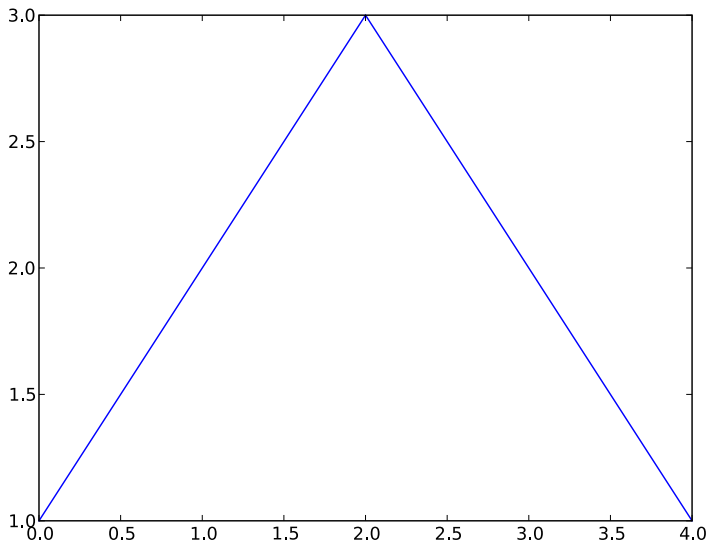

Figura 6.1: Figura generada por el comando anterior

```
>> x = linspace(-pi,pi,64);
>> plot(x,sin(x))
```

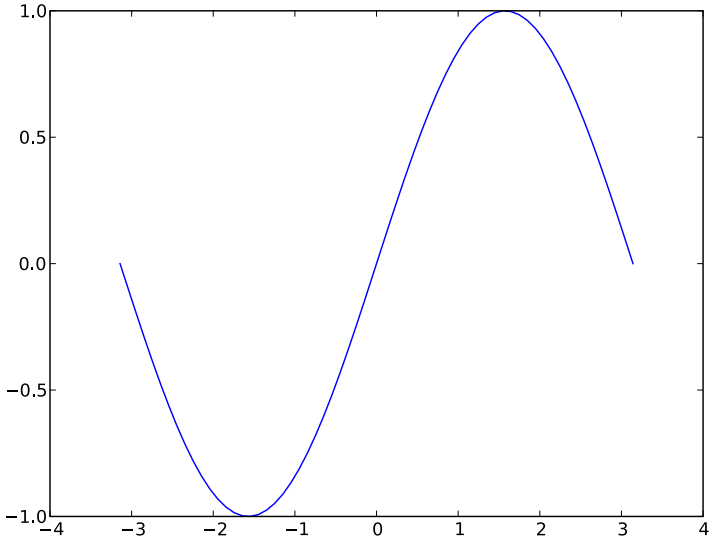

Figura 6.2: Figura generada por el comando anterior

```
>> x = linspace(-pi,pi,64);
>> plot(x,sin(x),'ro','markersize',5)
```

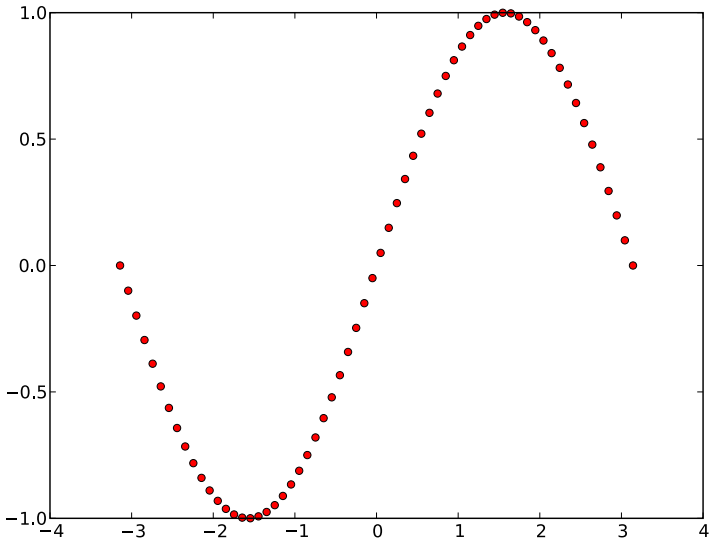

Figura 6.3: Figura generada por el comando anterior

El comando plot puede recibir una cantidad infinita de argumentos pero debemos agruparlos de una manera característica. Los dos primeros argumentos serán siempre datos, ya sean vectores o matrices. El tercer argumento será una cadena de dos caracteres con una letra, designando el color, y un símbolo, designando el tipo de línea o marcador. Seguidamente se pueden modificar los atributos tales como el grosor de la línea, el tamaño del marcador... Una vez terminado un grupo se puede empezar otra vez con dos argumentos con datos y así indefinidamente.

6.2 Figura activa

Lo más importante de representar gráficos en Matlab es el concepto de figura activa. Matlab puede tener abiertas centenares de ventanas al mismo tiempo pero sólo podrá representar datos en una: la figura activa. Podemos controlar dicha figura mediante el comando `figure`

figure(*n*)
: Comando que crea una nueva figura o selecciona como activa la figura dada. Cada figura tiene asignada un número entero, si *n* es una figura no existente creará una nueva y la activará, si *n* existe activará la figura correspondiente.

Otra consideración importante es que cada vez que representemos datos en la figura activa todo lo que esté ya en ella se va a borrar. Este comportamiento no es el deseado en muchos casos y puede modificarse mediante el comando hold

hold
: Cambia el comportamiento de la ventana activa. Funciona como un interruptor: `hold on` hace que cada dato se represente sobre lo anterior y `hold of` borra la ventana antes de pintar en ella. Por omisión se encuentra en `off`.

Un comando bastante útil es `clf`, que borra la figura activa.

6.3 Etiquetas

El paso siguiente es poner etiquetas: un identificador para cada eje y un título si lo creemos necesario. Las etiquetas se aplicarán, tal como se ha justificado en la sección anterior, sólo en la ventana activa.

title(*str*)
: Añade un título a la figura activa

xlabel(*str*)
: Añade una etiqueta al eje *x* de la ventana activa

ylabel(*str*)
: Añade una etiqueta al eje *y* de la ventana activa

Por ejemplo

```
x = linspace(0,500,10000);
plot(x,exp(-x/100).*sin(x));
title('Una funcion cualquiera')
xlabel('Tiempo')
ylabel('Amplitud')
```

Con el código anterior se obtiene la siguiente figura:

El paso siguiente es dotar a los gráficos con más de una curva de una leyenda que las distinga. Esto se consigue mediante la función `legend`.

legend(...)
: Al igual que con `plot` podemos utilizar esta función de múltiples maneras. La más simple es pasarle como argumento tantas cadenas de caracteres como curvas hayamos representado y automáticamente asignará por orden cada curva al identificador.

Supongamos que queremos representar el seno hiperbólico y el coseno hiperbólico y para distinguirlos no nos vale acordarnos que Matlab siempre pinta la primera curva en azul y la segunda en verde. Para ello podemos hacer lo siguiente:

```
x = linspace(-pi,pi,100);
plot(x,sinh(x),x,cosh(x));
legend('seno hiperbolico','coseno hiperbolico');
```

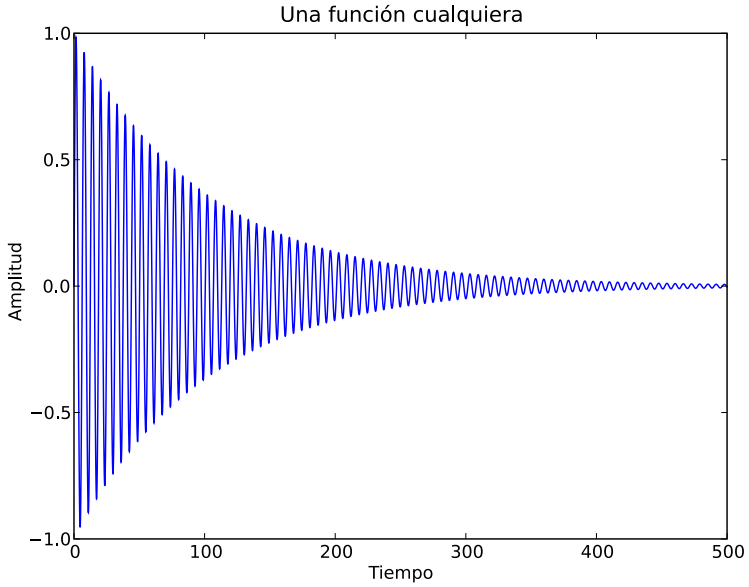

Figura 6.4: Ejemplo de etiquetas

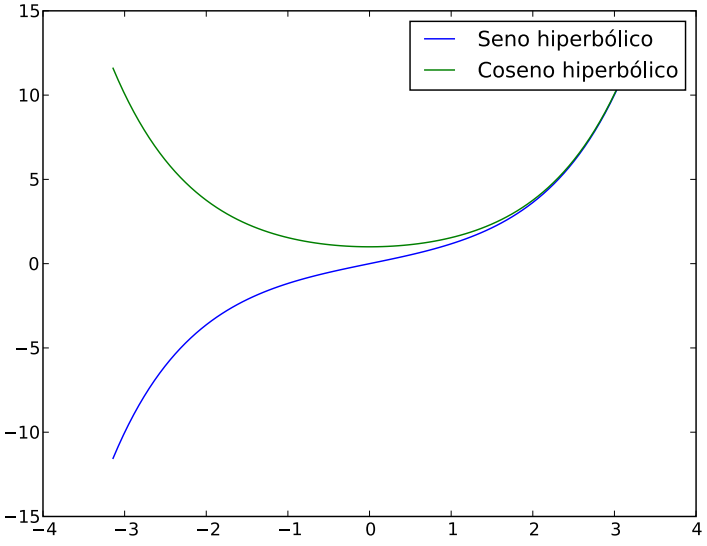

Figura 6.5: Ejemplo de aplicación de la leyenda

6.4 Otros comandos

No todas las curvas en el plano pueden representarse a partir del comando `plot` por los ejes que utiliza. En algunos casos nos vemos obligados a utilizar otros sistemas de coordenadas o a cambiar las referencias de los ejes. Estas funciones nos pueden ser de utilidad.

semilogx(...)
: Su uso y funcionamiento es idéntico al de la función `plot`. Representa gráficamente una serie de puntos tomando logaritmos en el eje x.

semilogy(...)
: Su uso y funcionamiento es idéntico al de la función `plot`. Representa gráficamente una serie de puntos tomando logaritmos en el eje y.

loglog(...)
: Su uso y funcionamiento es idéntico al de la función `plot`. Representa gráficamente una serie de puntos tomando logaritmos en ambos ejes.

polar(...)
: Su uso y funcionamiento es idéntico al de la función `plot`. Representa gráficamente una serie de datos en coordenadas polares. El primer argumento corresponde al ángulo respecto a la dirección principal θ y el segundo a la distancia respecto al centro de referencia ρ.

Un ejemplo de uso de la función `polar` es el siguiente

```
>> x = linspace(-pi,pi,100);
>> polar(x, cos(2.*x));
```

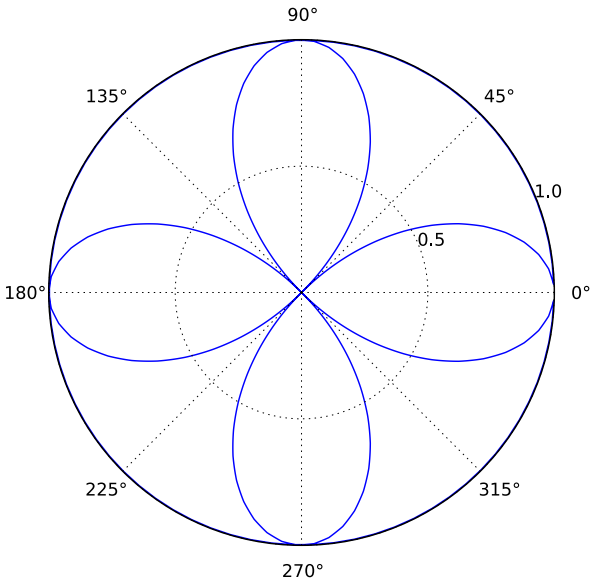

Figura 6.6: Ejemplo de gráfica en coordenadas polares

6.5 Plot handles

Cada comando cuya salida se expresa en una figura puede también devolver un argumento llamado plot handle. Utilicemos como ejemplo la figura anterior.

```
>> x = linspace(-pi,pi,100);
>> h = polar(x, cos(2.*x));
```

Entonces, a parte de representar la curva, *h* es una variable que contiene toda la información correspondiente a la misma y dentro del léxico de Matlab suele recibir el nombre de *handle*. Con la función `get` podemos obtener toda la información del *handle* y mediante la función `set` podemos cambiar sus propiedades según nuestras necesidades. Lo más interesante es que no sólo las curvas devuelven un *handle*; todos los objetos gráficos, incluso los ejes o la propia figura genera un *handle*.

get (*h*)
: Función que obtiene las características de un *handle* gráfico, ya sea una curva, los ejes de la figura o la misma figura

set (*h, attr, val*)
: Funcion que modifica las características de un *handle* gráfico, ya sea una curva, los ejes de la figura o la misma figura. Los argumentos siempre son, por este orden:

  H El *handle*

  Attr Un atributo válido del handle como cadena de caracteres

  Val El nuevo valor del atributo.

En el caso de las curvas o de la propia figura es la propia función (`plot`, `semilogx` o `figure`) la que genera el *handle* pero también podemos utilizar las funciones que devuelven *handles* como argumentos de salida.

gca ()
: No necesita ningún argumento. Devuelve el *handle* de los ejes de la figura activa.

gcf ()
: No necesita ningún argumento. Devuelve el *handle* de la figura activa

Utilizamos algunos de estos comandos en el ejemplo siguiente:

```
p=plot([1,2,3,2,1]);
set(p,'linewidth',2);
set(p,'marker','o');
set(p,'markersize',12);
set(p,'markerfacecolor','y');
set(p,'markeredgecolor','r');
t=title('Pirámide');
set(t,'fontsize',14);
set(t,'color','g');
```

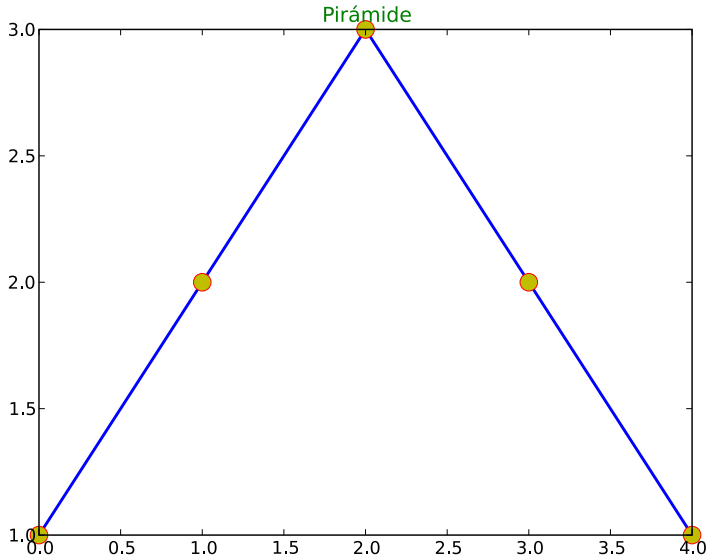

Figura 6.7: Ejemplo de uso de `set`

6.6 Subplots

6.7 Representación de datos en el plano

6.8 Ejercicio de síntesis

El objetivo de este ejercicio es volver al ejercicio de síntesis en el que estudiábamos la convergencia de las series de Taylor. Vimos que, a medida que añadíamos términos a la serie esta se acercaba al valor de la función en las cercanías del punto en el que se calculaba. El resultado al que llegamos era una serie de puntos que lo demostraba para la función exponencial.

Aprovechando que ya disponemos de una función que genera las series vamos a representar gráficamente la función junto con una serie de desarrollos de Taylor con distinto orden. Después representaremos el error de la aproximación restando el desarrollo a la función y finalmente el error en función del orden en dos puntos cercanos al origen.

```
clf
clear all

exp_serie = @(x,n) polyval(1./[factorial(linspace(n,1,n)),1],x);

figure(1)

x = linspace(0,3,100);
plot(x,exp(x),...
     x,exp_serie(x,1),...
     x,exp_serie(x,2),...
     x,exp_serie(x,3),...
     x,exp_serie(x,4),...
     x,exp_serie(x,5));
legend('exacta','n=1','n=2','n=3','n=4','n=5');
title('Desarrollos de Taylor de una funcion exponencial en x=0');
xlabel('x')
ylabel('y')

figure(2)
semilogy(x,exp(x)-exp_serie(x,1),...
         x,exp(x)-exp_serie(x,2),...
         x,exp(x)-exp_serie(x,3),...
         x,exp(x)-exp_serie(x,4),...
         x,exp(x)-exp_serie(x,5));
legend('n=1','n=2','n=3','n=4','n=5');
hold on

semilogy([0.1,0.1,0.1,0.1,0.1],[...
    exp(0.1)-exp_serie(0.1,1),...
    exp(0.1)-exp_serie(0.1,2),...
    exp(0.1)-exp_serie(0.1,3),...
    exp(0.1)-exp_serie(0.1,4),...
    exp(0.1)-exp_serie(0.1,5),...
                    ],'ko')

semilogy([0.2,0.2,0.2,0.2,0.2],[...
    exp(0.2)-exp_serie(0.2,1),...
    exp(0.2)-exp_serie(0.2,2),...
    exp(0.2)-exp_serie(0.2,3),...
    exp(0.2)-exp_serie(0.2,4),...
    exp(0.2)-exp_serie(0.2,5),...
                    ],'ko')
```

```
xlabel('x')
ylabel('exp(x)-p_n(x)')
title('Error de las aproximaciones')

figure(3)
semilogy([0,1,2,3,4,5,6,7,8],[exp(0.1),...
                    exp(0.1)-exp_serie(0.1,1),...
                    exp(0.1)-exp_serie(0.1,2),...
                    exp(0.1)-exp_serie(0.1,3),...
                    exp(0.1)-exp_serie(0.1,4),...
                    exp(0.1)-exp_serie(0.1,5),...
                    exp(0.1)-exp_serie(0.1,6),...
                    exp(0.1)-exp_serie(0.1,7),...
                    exp(0.1)-exp_serie(0.1,8)],...
          'ko')
hold on
semilogy([0,1,2,3,4,5,6,7,8],[exp(0.2),...
                    exp(0.2)-exp_serie(0.2,1),...
                    exp(0.2)-exp_serie(0.2,2),...
                    exp(0.2)-exp_serie(0.2,3),...
                    exp(0.2)-exp_serie(0.2,4),...
                    exp(0.2)-exp_serie(0.2,5),...
                    exp(0.2)-exp_serie(0.2,6),...
                    exp(0.2)-exp_serie(0.2,7),...
                    exp(0.2)-exp_serie(0.2,8)],...
          'bo')

legend('punto 0.1', 'punto 0.2');
axis([-0.1,8.1,1e-16,1e1]);
xlabel('Orden')
ylabel('exp(x)-p_n(x)')
title('Convergencia de aproximaciones')
```

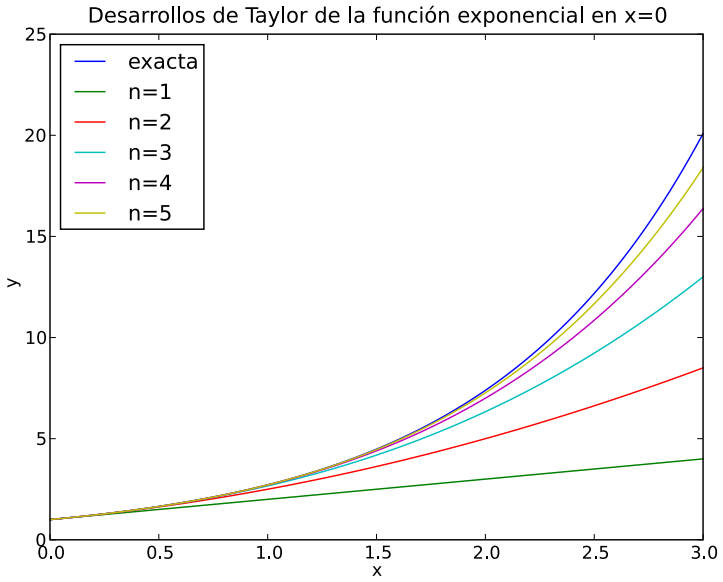

Figura 6.8: La función exponencial y sus desarrollos de Taylor en el origen hasta orden 5.

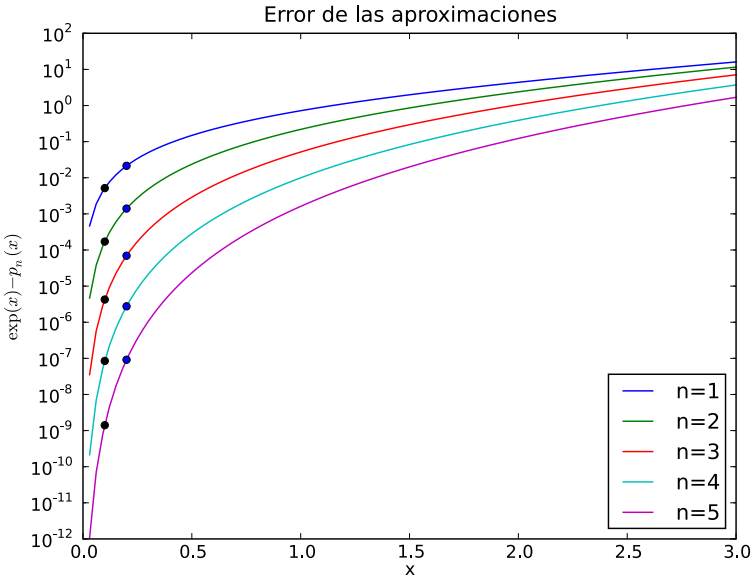

Figura 6.9: Error del desarrollo. Los puntos son los valores en las abcisas x=0.1 y x=0.2 respectivamente

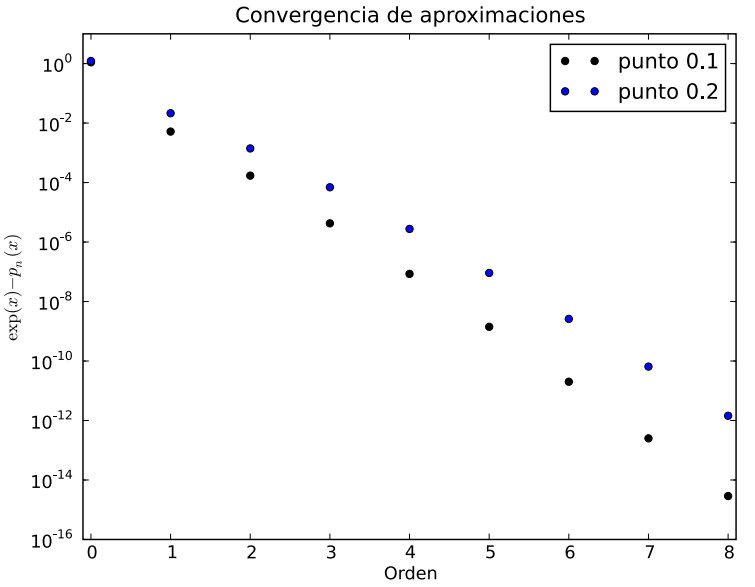

Figura 6.10: Convergencia en función del orden.

CAPÍTULO 7

Estadística Descriptiva y análisis de datos

En este capítulo nos centraremos en los cálculos más básicos de la Estadística Descriptiva y de los modelos de datos con Matlab. Esta pequeña introducción no cuenta con una descripción teórica de los cálculos así que se suponen conocimientos básicos de Estadística. En vez de describir de forma sistemática las funciones como en las secciones anteriores, avanzaremos siguiendo un ejemplo.

Advertencia: Algunas de las funcionalidades más específicas para el tratamiento estadístico de datos con Matlab se encuentran en el Statistics Toolbox y deben adquirirse a parte. En el caso de Octave podemos instalar el paquete extra de estadística de Octave-forge en http://octave.sourceforge.net.

7.1 Distribuciones de frecuencias

Cuando analizamos datos la primera pregunta que debemos hacernos es si se ajustan a algún patrón. Los datos pueden ser muy numerosos y estar dispersos así que un ordenador nos puede ser de gran utilidad. Detectar el patrón depende en gran medida de la complejidad de los datos; no es lo mismo analizar cómo funciona una ruleta en un casino que hacer lo mismo con un resultado electoral en Estados Unidos.

Supongamos que estamos ante una serie de datos como, por ejemplo, la cotización de las acciones de Google en la apertura y cierre de NASDAQ durante cuatro años.

Nota: Adjuntos al documento pdf encontraréis los arhivos *googopen.dat* y *googclse.dat* necesarios para repetir los ejemplos.

Nota: Antes de entrar en el tema de la *persistencia* lo único que debemos saber a estas alturas es que para cargar estos dos archivos basta con utilizar el comando `load`.

```
op = load('googopen.dat');
cl = load('googclse.dat');
```

En vez de analizar los datos de apertura y cierre nos centraremos en la diferencia entre ambos valores, la cotización de la sesión, mucho más fáciles de analizar. En breve veremos el porqué.

Lo primero que podemos calcular es el histograma de nuestros datos. Podemos hacerlo de dos maneras: representando gráficamente el histograma con `hist` o calculando las frecuencias con `histc` y utilizando el comando `plot`

```
dif = cl-op
bins = linspace(min(dif),max(dif),30)
```

```
freq = histc(dif,bins);
plot(bins,freq);
xlabel('Diferencial')
ylabel('Frecuencia')
```

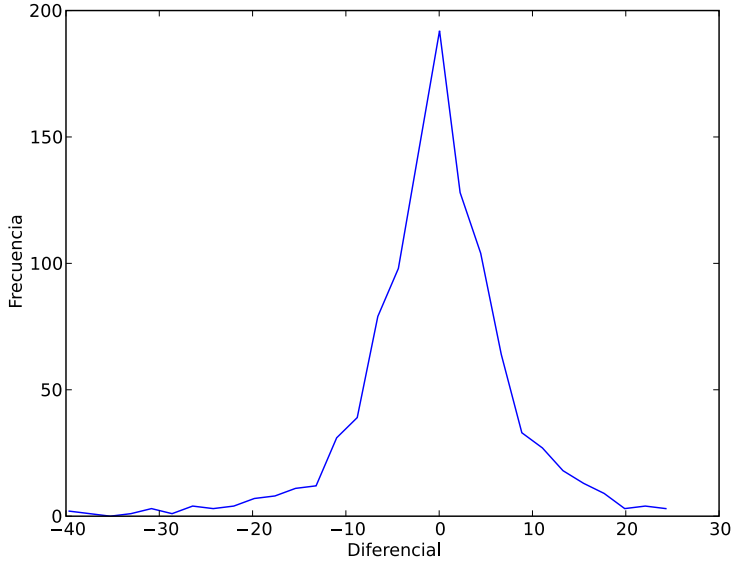

Figura 7.1: Histograma de los diferenciales del stock de Google

El histograma está sólo a un paso de la FDP (Función Densidad de Probabilidad) obtenida a partir de los datos. Para ello la función definida por las frecuencias deberá cumplir la siguiente propiedad:

$$\int_{-\infty}^{\infty} f(x)dx = 1$$

Para normalizar nuestro histograma basta con dividir las frecuencias por el valor de su integral utilizando la función `trapz`

```
pdf = freq/trapz(bins,freq);
```

7.2 Medidas de concentración

Las siguientes funciones sirven para calcular las medidas de tendencia central de una muestra.

mean (*x, dim*)
: Calcula la media aritmética de una muestra. *dim* sirve para seleccionar la dimensión a través de la cual se calcula la media en el caso que los datos tengan forma de matriz.

geomean (*x, dim*)
: Funcionamiento idéntico a `mean`. Calcula la media geométrica de una muestra.

harmmean (*x, dim*)
: Funcionamiento idéntico a `mean`. Calcula la media armónica de una muestra.

median (*x, dim*)
: Funcionamiento idéntico a `mean`. Calcula la mediana de una muestra.

7.3 Medidas de dispersión

Hay dos definiciones para la desviación típica. En algunos libros se llaman respectivamente cuasidesviación típica y desviación típica. En Matlab, por defecto, la desviación típica será calculada con

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2 n_i}$$

std (*x, flag, dim*)
: Calcula la desviación estándar de una muestra. Si el argumento *flag* se omite o *flag* = 0 se utiliza la defincón anterior de la desviación típica. Si se introduce el valor de *flag* = 1 entonces se utiliza la definición anternativa de la desviación típica.

La definición alternativa es

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 n_i}$$

var (*x, flag, dim*)
: Calcula la varianza de una muestra. Es el cuadrado de la desviación típica.

7.4 Funciones de densidad de probabilidad conocidas

Siendo rigurosos el histograma da toda la información que necesitamos sobre nuestros datos pero para tomar hipótesis sobre los mismos el paso siguiente suele ser encontrar alguna función de densidad de probabilidad conocida que se ajuste bien. La más habitual cuando el histograma parece simétrico es la distribución Normal o de Gauss.

normpdf (*x, mu, sigma*)
: Calcula el valor de la función densidad de probabilidad en x dados la media *mu*, μ y la desviación típica *sigma*, σ.

$$p(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

El paso siguiente en nuestro análisis de las acciones de Google puede ser comparar los diferenciales de las sesiones con la distribución normal. Para ello aprovecharemos que ya hemos calculado la FDP de nuestrso datos y la representaremos junto con la normal.

```
plot(bins,pdf,bins,normpdf(bins,mu,sig));
xlabel('Diferenciales')
ylabel('Probabilidad')
legend('Histograma','Normal');
```

Hay dos maneras de acceder a las funciones densisdad de probabilidad, cada una tiene su propia función terminada en pdf, como `betapdf` o `lognpdf` pero podemos utilizarlas todas con la función `pdf`.

pdf (*nombre, x, a, b, c*)
: Calcula el valor de la FDP de nombre *nombre* en el punto x. El número de parámetros necesarios para realizar el cálculo depende de la FDP. Por ejemplo, si *nombre* es `'norm'` tendremos que proporcionar dos parámetros, si es `'t'` para la distribución t de Student bastará con un parámetro.

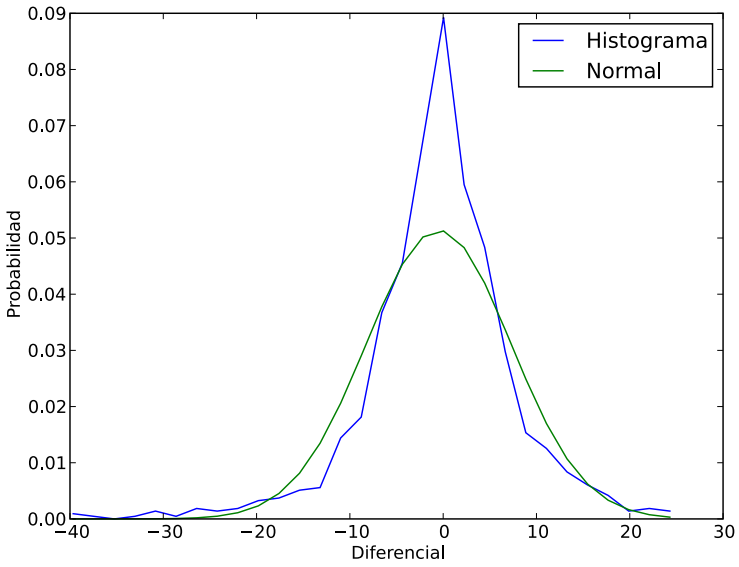

Figura 7.2: Comparación con la FDP Normal

7.5 Ejercicio de Síntesis

Existe un fenómeno físico importante en los sistemas no lineales llamado *intermitencia*. En los fenónemos que muestran intermitencia observamos fluctuaciones mayores cuando separamos nuestros puntos de toma de datos ya sea en el espacio como en el tiempo. Esta propiedad es importante en el estudio de campos como la Turbulencia o en el Análisis Financiero.

Cuanto más intermitente es un sistema más difícil se hace predecir el valor de la variable a largo plazo. Por este motivo se dice que los valores que en un mercado muestran una gran intermitencia entrañan también un gran riesgo.

Este ejercicio pretende también demostrar que predecir el valor de un producto financiero a tiempos mayores a un mes es prácticamente imposible si únicamente se tiene información sobre su valor. Para comprender mejor este ejercicio necesitamos conocer el concepto de "cola ancha" o "fat tail".

Si hacemos el test χ^2 a los diferenciales obtendremos, con un margen de error minúsculo, que los datos se ajustan a una distribución normal. Sin embargo cualquier iniciado en Análisis Financiero sabe perfectamente que asumir que estos datos se ajustan a una distribución normal es algo cercano a un suicidio. La diferencia entre las dos FDP no se encuentra tanto en el los valores centrales sino en las colas. Es algo que se aprecia mucho mejor si, en vez de representar las FDP del modo convencional, utilizamos una gráfica semilogarítmica.

Lo que vemos es que, aunque las dos FDP parezcan parecidas, su comportamiento lejos de los valores centrales es completamente distinto. Mientras la Normal se va rápidamente a valores muy pequeños, nuestra FDP parece no seguir la misma tendencia. *Este comportamiento es muy importante porque implica que la probabilidad de sucesos extremos es relevante*. Este fenómeno, asociado a la intermitencia, se conoce como *cola ancha* o *fat tail* e implica que se corre un gran riesgo asumiendo que el siguiente valor va a ser cercano a la media.

Para comprobar el efecto de la intermitencia representaremos la gráfica logarítmica de distintos histogramas en los que calcularemos el diferencial el mismo día y con 1, 4 y 9 días de diferencia. Para ello nos crearemos la función *tailcheck*

```
function tailcheck(open,clse,dst,n)
% Funcion que representa la cola logaritmica del histograma en funcion
% de la distancia entre las medidas.
% Argumentos de entrada:
%  open: Datos de apertura de sesion
%  clse: Datos de cierre de sesion
```

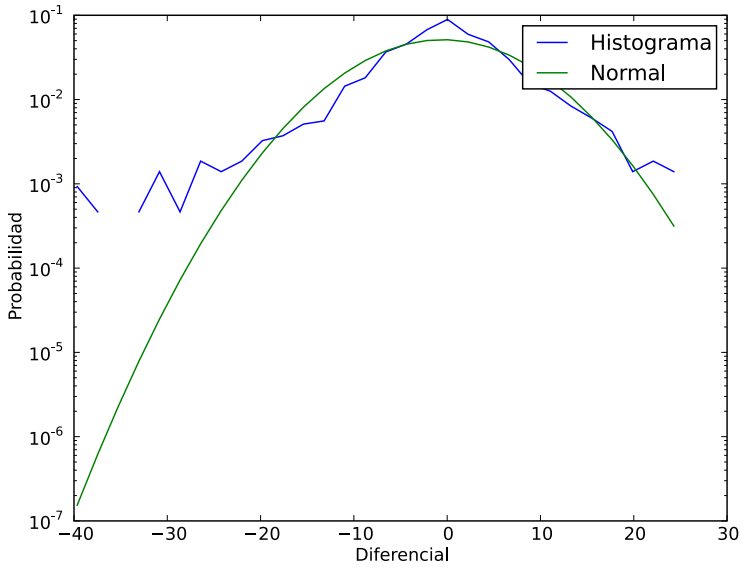

Figura 7.3: Comparación de las colas de la FDP

```
%  dst:  Decalaje entre la apertura y el cierre
%  n:    Numero de puntos del histograma
%  fign: Numero de figura en la que se representara la cola
    dis = clse(1:end-dst)-open(1+dst:end);
    bins = linspace(min(dis),max(dis),n);
    freq = histc(dis,bins);
    pdf = freq/trapz(bins,freq);
    semilogy(bins,pdf)


figure(1)
clf
hold on

open = load('googopen.dat');
clse = load('googclse.dat');

tailcheck(open,clse,0,30);
tailcheck(open,clse,1,30);
tailcheck(open,clse,4,30);
tailcheck(open,clse,9,30);

legend('0 dias','1 dia','5 dias','9 dias')
```

Como se ve claramente, a medida que separamos el tiempo entre los diferenciales la probabilidad de obtener un valor más lejano de la media crece significativamente a la vez que desciende la probabilidad de lo contrario. El fenómeno de *fat tail* crecería indefinidamente acercándose al suceso puramente aleatorio en un caso límite.

7.6 Ejercicio propuesto

Calcular y representar los caminos de un grupo de partículas con movimiento aleatorio confinadas por un par de barreras B^+ y B^- unidades del orígen que es desde donde salen las partículas. Un movimiento aleatorio se calcula

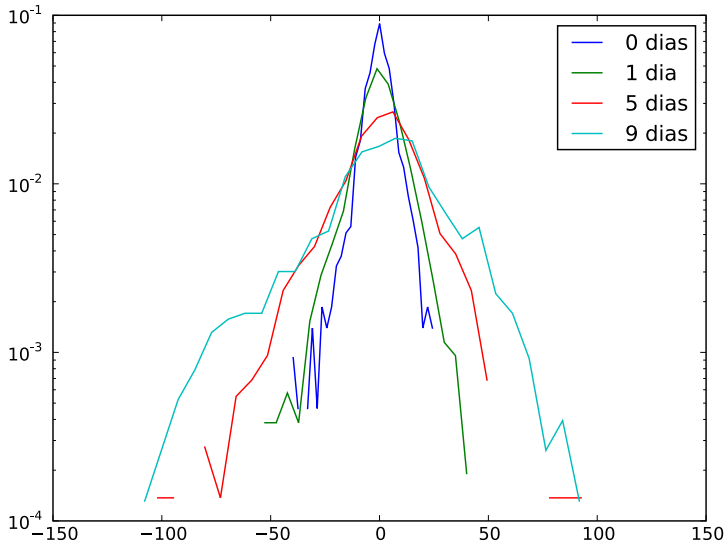

Figura 7.4: Colas anchas debidas a la intermitencia

mediante la fórmula

$$x_{j+1} = x_j + s$$

donde s es el número obtenido de una distribución normal estandarizada de números aleatorios según la función `randn`

Se cambiarán las condiciones de contorno de la siguiente manera:

1. Reflexión. Cuando una partícula se encuentre fuera de la frontera se devolverá al interior del dominio como si hubiera rebotado en una pared
2. Absorción. La partícula muere cuando entra en contacto con la pared.
3. Absorción parcial. Es la combinación de los dos casos previos. La partícula rebota en la pared y la perfección de la colisión depende de una determinada distribución de probabilidad.

Calcular una serie relevante de trayectorias y calcular:

1. La posición media de las partículas en función del tiempo.
2. La desviación típica de las posiciones de las partículas en función del tiempo.
3. ¿Influyen las condiciones de contorno en las distribuciones?
4. Para los casos de absorción y absorción parcial representar gráficamente el número de partículas supervivientes en función del número de saltos temporales.

7.7 Análisis de Datos

Integración y Ecuaciones Diferenciales Ordinarias

Este capítulo trata sobre dos disciplinas bastante extensas dentro del Cálculo Numérico. El motivo es que existen muchos métodos para integrar una función o una ecuación diferencial y algunos funcionarán mejores que otros según el caso. Nos centraremos en las integrales definidas

$$I = \int_{x_i}^{x_f} f(x)dx$$

y en los problemas de Cauchy; ecuaciones diferenciales ordinarias en los que conocemos la ecuación diferencial y las condiciones iniciales del sistema

$$\frac{d\vec{x}}{dt} = f(\vec{x}, t) \qquad \vec{x}(0) = \vec{x}_0$$

El resultado del primer problema es bien un número o una función que depende de los límites de integración. El resultado del segundo problema es una trayectoria en el espacio $\vec{x}(t)$.

8.1 Integración Numérica

La complejidad de la integración depende de las dimensiones de nuestro problema. Si queremos integrar una función con sólo una variable $f(x)$ entonces necesitaremos la función (definida como función anónima) y dos escalares que harán de límites de integración. Si la función depende de dos variables $f(x, y)$ la cosa se complica porque el límite de integración es una curva en el plano. Para esta introducción nos frenaremos en el caso más fácil.

quad (*fun, a, b*)
: Calcula la integral definida de una función entre los intervalos de integración *a* y *b*. *fun* debe ser una función anónima.

Por ejemplo, supongamos que queremos saber la probabilidad que el siguiente suceso de un fenómeno de media 5.4 y desviación típica 0.4 tenga un valor mayor que 7.

```
>> mu = 5.4;
>> sig = 0.4;
>> f = @(t) normpdf(t,mu,sig);
>> prob = quad(f,7,100)

prob =

   3.2601e-05
```

Este sencillo ejemplo nos sirve para entender perfectamente cómo funciona la función `quad`. Hay otras variables de entrada y de salida como la tolerancia en la integración o la estimación del error de la operación. Podemos obtener más información del cálculo anterior con

```
>> [prob,err] = quad(f,7,100,1e-10)

prob =

   3.1671e-05


err =

    65
```

En este caso hemos exigido al algoritmo un error entre la precisión simple y la doble precisión y hemos pedido también que nos muestre el número de veces que ha evaluado la función para llegar al resultado. Nos ha servido para comprobar que el error de la primera integración ronda el 3 % Otra función para realizar exactamente la misma operación es `quadl`

quadl (*fun, a, b*)
: Ver función `quad`. Según la documentación de Matlab `quadl` es más efectivo cuando se pide mayor precisión con una función sin demasiadas oscilaciones.

Para comprobar que muchas veces la documentación no se cumple intentaremos hacer la misma integral con el algoritmo alternativo.

```
>> [prob,err] = quadl(f,7,100,1e-10)

prob =

   3.1671e-05


err =

   138
```

Vemos que necesita más evaluaciones para llegar al mismo resultado con la misma precisión. Si queremos hacer la integral impropia (aunque es convergente), tanto `quad` como `quadl` fallan. Sí podemos utilizar `quadgk` para ello

```
>> [prob,err] = quadgk(f,7,Inf)

prob =

   3.1671e-05


err =

   3.1140e-17
```

Utilizando la FDP normal acumulada podemos obtener el resultado correcto.

```
>> 1-normcdf(7,5.4,0.4)

ans =

   3.1671e-05
```

Advertencia: El ejemplo anterior demuestra que la integración numérica, aunque en la mayoría de los casos no entrañará ninguna dificultad, puede proporcionar resultados imprecisos. En algunos casos puede ser una buena idea comprobar los resultados de forma aproximada.

Lo mismo puede decirse de la integración bidimensional y tridimensional.

Nota: La integración en más de una dimensión tiene fronteras más complejas. En la integración bidimensional es una curva cerrada en el plano y en la tridimensional una superficie también cerrada. Las funciones `dblquad` y `triplequad` sólo permiten integrar con límites constantes.

8.2 Integración de problemas de Cauchy

Los sistemas de ecuaciones diferenciales no lineales suelen no tener solución analítica. Es entonces un coto particular del Cálculo Numérico y Matlab cuenta con una gran artillería de herramientas para resolver estos problemas. Lo aprendido será fácilmente aplicable a los problemas de condiciones de contorno. Por lo que respecta a los problemas lineales, Matlab dispone también de funciones específicas para resolverlos en el espacio de Laplace.

Nos centraremos en los problemas de condiciones iniciales o problemas de Cauchy. Para resolverlos necesitaremos la función del sistema, las condiciones iniciales y el intervalo de tiempos de la solución.

Desde un punto de vista puramente práctico lo único que debemos saber para resolver estos problemas satisfactoriamente es si el problema es stiff o no. Para entender el significado de la *rigidez* de un sistema es muy recomendable seguir un buen curso sobre resolución numérica de ecuaciones en derivadas parciales. Aquí sólo diremos que un sistema es stiff cuando introduce gradientes fuertes y un método de integración explícito es incapaz de resolverlos.

El ejemplo clásico para entender este problema es utilizar la ecuación del oscilador de Van der Pol.

$$x'' + \mu(x^2 - 1)x' + x = 0$$

Esta ecuación de segundo orden es stiff con valores de μ elevados. Para comprobarlo podemos intentar resolver el problema com $\mu = 1$ y la función `ode45`

Nota: Este ejemplo es tan popular que Matlab dispone ya de las funciones `vdp1` y `vdp1000` para la ecuación con $mu = 1$ y $mu = 1000$. Esta primera vez y a modo de ejemplo escribiremos la función

```
vdp1 = @(t,y) [y(1);y(2)*(1-y(1))-y(1)];
[t,y] = ode45(vdp1,[0,20],[0;2]);
plot(t,y(:,1))
```

ode45 (*fun, tspan, y0*)
: Integra la función *fun* que debe ser una función de dos variables de la forma *dy = fun(t,y)* donde **tanto y como dy deben ser vectores columna**.

  tspan es un vector de dos elementos con el intervalo de tiempos e *y0* es el vector de condiciones iniciales.

  Devuelve dos vectores de longitud arbitraria. *t* son los tiempos en los que se ha hallado la solución e *y* es un vector que contiene los vectores solución del problema en cada instante.

Al representar la solución llegamos al siguiente resultado.

Si, por el contrario intentamos resolver el mismo problema con $\mu = 1000$ nos encontraremos con la desgradable sorpresa de que el Matlab no termina nunca de calcular.

El motivo es que en los problemas *stiff* el paso de tiempo necesario para que un esquema explícito como el de `ode45` llegue a una solución tiende a cero. Esto es debido a que, antes de dar una solución errónea, el esquema de paso variable va reduciendo el paso temporal sin ninguna limitación. Obviamente, si el paso temporal tiende a cero, el tiempo necesario para llegar a la solución tiende a infinito.

La solución es utilizar un esquema implícito como el ode23s.

Nota: En Matlab, los esquemas de integración que terminan con una s son implícitos y pueden integrar sistemas *stiff*

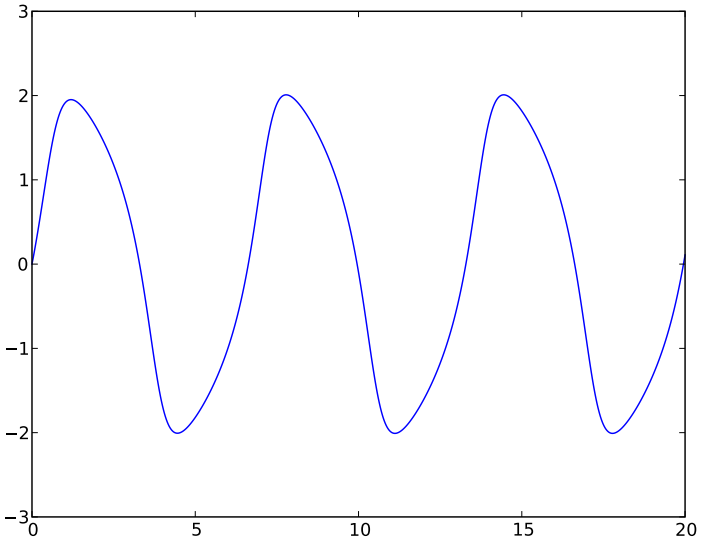

Figura 8.1: Solución del oscilador de Van der Pol para $\mu = 1$

Una vez llegamos a la solución entendemos por qué no eramos capaces de integrarla.

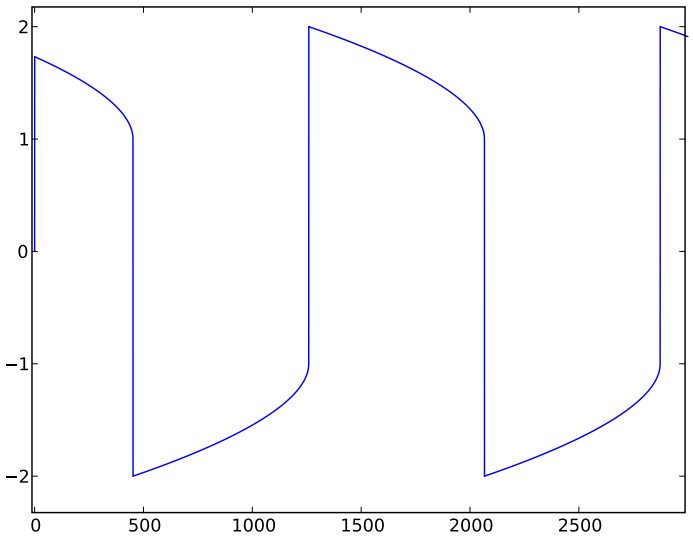

Figura 8.2: Solución del oscilador de Van der Pol para $\mu = 1000$

```
[t,y] = ode23s(@vdp1000,[0,3000],[0;2]);
plot(t,y(:,1))
```

Advertencia: Sobre Octave. Aunque el paquete odeint de Octave Forge proporciona las funciones `ode45` y `ode23s` entre otras, Octave dispone de un driver completamente distinto para resolver problemas de Cauchy llamado `lsode`. Para utilizarlo debemos tener en cuenta que la función a integrar se escribe con los argumentos permutados *dy = fun(y,t)* y que en la llamada, en vez de proporcionar el intervalo de tiempos, debemos proporcionar el vector de tiempos en los que queremos la solución.
Otra diferencia importante entre ambos es que en Matlab las opciones de los esquemas de integración se modifican utilizando las funciones `odeset` y `odeget`, mientras que en Octave debemos utilizar la función `lsode_options`.
Por ejemplo, las llamadas anteriores deberíamos efectuarlas en Octave como:

```
%Por omisión Octave utiliza un método implícito.
lsode_options('integration metod','non-stiff');
t = linspace(0,20,1000);
y = lsode(vdp1,[0;2],t);
```

8.3 Ejercicio propuesto

Representar en el espacio mediante la función `plot3` la trayectoria de la partícula que, saliendo desde el punto (1,1,1) y durante 50 unidades de tiempo, cumple la ecuación siguiente:

$$\begin{aligned} \dot{x} &= a(y-x) \\ \dot{y} &= x(b-z)-y \\ \dot{z} &= xy-cz \end{aligned}$$

con a = 10, b = 28 y c = 8/3

8.4 Ejercicio propuesto

Hallar el área de la región del plano comprendida entre la curva

$$y = \frac{x^2-1}{x^2+1}$$

y su asíntota. La solución es 2π.

CAPÍTULO 9

Programación en Matlab

Antes de entrar en la práctica de la programación es importante dar una pequeña nota sobre paradigmas de programación. Un paradigma es una metodología que viene impuesta por el propio lenguaje de programación. Los hay a decenas: programación imperativa, estructurada, modular, orientada a objetos, diseño por contrato...

Aunque Matlab soporta la programación modular y la orientada a objetos casi toda su librería está formada por funciones, no por módulos ni clases ni cualqueir otra cosa. Esto significa que, mientras el código que escribamos nosotros puede seguir prácticamente cualquier paradigma, si hacemos un uso extensivo de las funciones propias de Matlab nos veremos forzados utilizar la programación estructurada.

9.1 Funciones

Las funciones que hemos visto hasta ahora no eran funciones de verdad sino funciones anónimas. Aunque en muchos casos son más prácticas que las funciones es el momento de aprender a encapsular tareas apropiadamente.

Abriremos el editor y escribiremos en él el siguiente código:

```
function out = aprsin(x)
  out  = x - x.^3/6;
```

y lo guardaremos en el directorio de trabajo con el nombre `aprsin.m`. Esta pequeña función nos sirve para entender su sintaxis.

`function`
: Identificador de función. Las funciones sirven para encapsular tareas que dependen de argumentos de entrada y devuelven argumentos de salida. La sintaxis es la siguiente

  - Función con tres argumentos de entrada y un argumento de salida:

    ```
    function argout = nombre(argin1, argin2, argin3)
      ...
      argout = ...
    ```

  - Función con dos argumentos de entrada y dos de salida:

    ```
    function [argout1, argout2] = nombre(argin1, argin2)
      ...
      argout1 = ...
      argout2 = ...
    ```

  Una función puede depender de cualquier cantidad de argumentos de entrada y de salida y debe cumplir siempre estos dos requisitos:

1. Al final de la función todos los argumentos de salida deben tener un valor asignado
2. El nombre de la función (en los ejemplos `nombre`) es el nombre por el que se podrá llamar la función y el nombre con el que deberá guardarse en el directorio de trabajo.

Cualquier error en uno de los dos puntos anteriores hará que la función o bien de un error o no podamos llamarla correctamente.

9.2 Ejercicio de síntesis

Bibliografía

[KnR] El Lenguaje de Programación C. Brian W. Kernighan, Dennis M. Ritchie. Pearson Educación (2ª Ed. 1991)

Índice

X

Y

Z