iimyo2

Data · April 2014

1 author:

Guillem Borrell
Kernel Analytics

Matemáticas en Ingeniería con Matlab y Octave

Release 0.1

Guillem Borrell i Nogueras et al.

12 de October de 2013

Índice general

Prólogo

Este texto no es un libro, es un conglomerado de material relativo a la programación en Matlab que difícilmente encontraría su sitio. Parte de la idea que *Introducción Informal a Matlab y Octave* es un mal libro con material que puede resultar útil. Cuando tomé conciencia de ello decidí cambiar completamente el planteamiento del texto; algo que obligaba a empezar desde cero.

Si bien el texto anterior nacía de un año sabático y de la necesidad de poner por escrito todo lo que iba aprendiendo de Matlab esta monografía sí tiene una finalidad específica: profundizar en la programación en Matlab a base de ejemplos completos. Decidí definitivamente el rumbo cuando muchos de mis compañeros en la Escuela de Aeronáuticos me pedían ayuda con el mismo problema. En el sentido puramente curricular sabían programar en Matlab, conocían la sintaxis del lenguaje y unas cuantas funciones. Pero cuando se sentaban a programar de verdad para resolver un problema de cierta entidad no sabían por donde empezar. Es más, muchos de ellos tenían experiencia con otros lenguajes de programación y tenían la sensación de no poder aplicar lo ya aprendido a Matlab.

Llevo unos cuantos años peleándome con Matlab. Un poco a mi pesar porque no es ni mucho menos mi primera opción como lenguaje de programación. He movido el suficiente código como para darme cuenta que es un bicho raro, una excepción que te obliga a utilizar una determinada metodología quizás demasiado particular.

Este sigue siendo un libro que llama a la colaboración. Quien crea que tiene una idea que merezca la pena convertirse en un capítulo o un ejemplo puede discutirla libremente conmigo. Estoy también abierto a sugerencias y correcciones y no tengo ningún reparo en ceder la autoría de capítulos enteros. Si mi intención fuera el lucro o la popularidad ni siquiera hubiera empezado.

Espero y deseo que lo que encontréis aquí os sea de utilidad.

Versión 0.1

Fecha 11 de October de 2013

La página web de este proyecto es http://iimyo.forja.rediris.es donde puede encontrarse información detallada sobre cómo colaborar con el proyecto.

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.

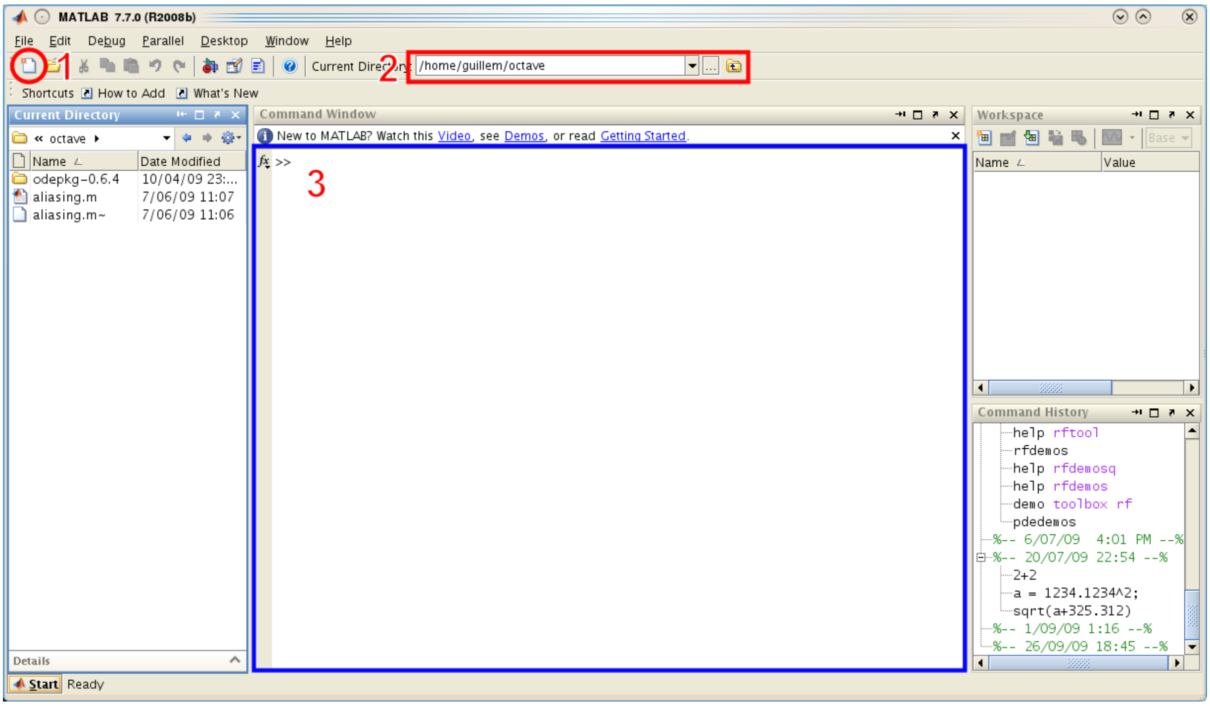

Figura 2.1: Captura de la interfaz gráfica de Matlab R2008b

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.

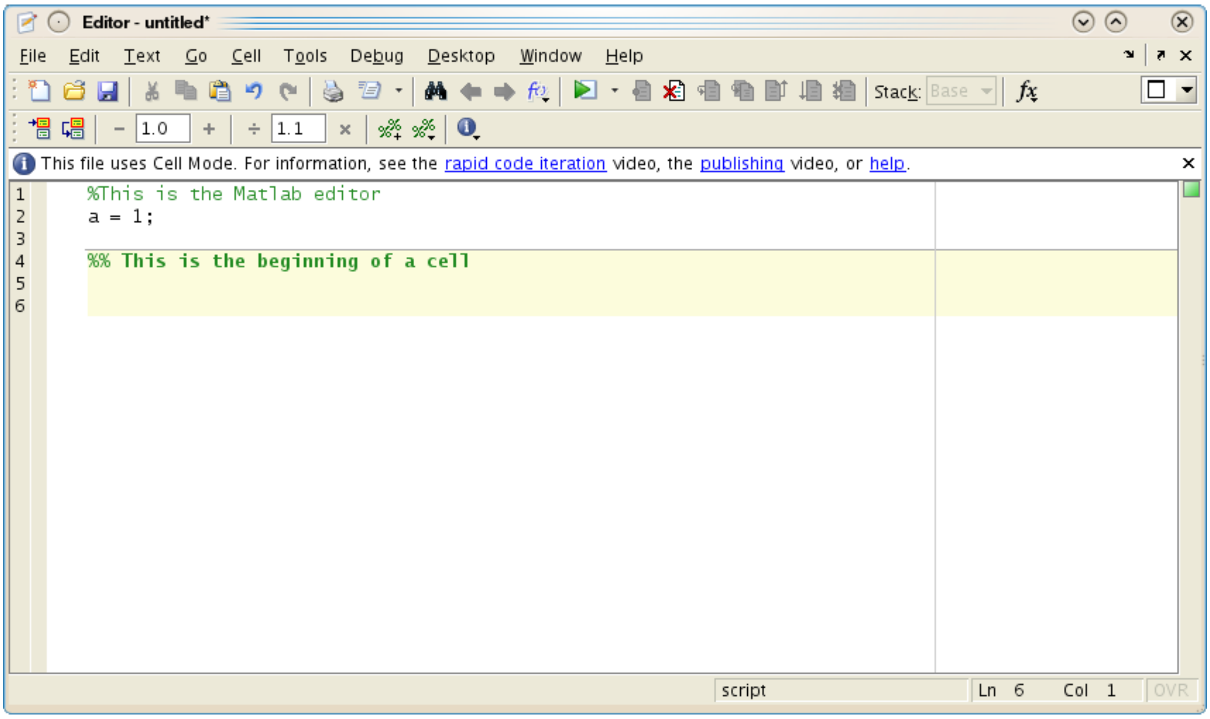

Figura 2.2: Captura del editor de Matlab R2008b

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.

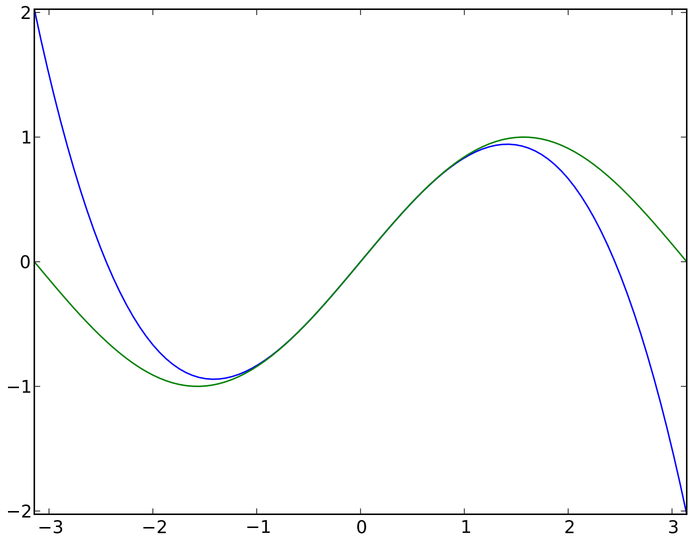

Figura 2.3: Resultado del script

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.

CAPÍTULO 4

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

CAPÍTULO 5

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]);
```

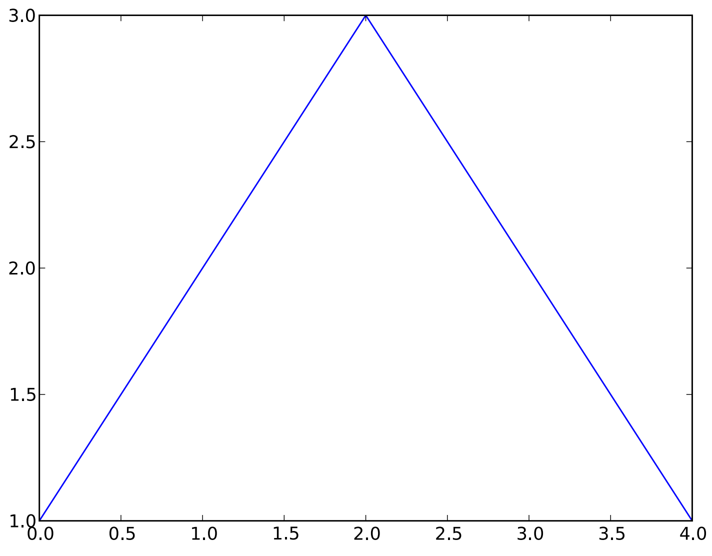

Figura 6.1: Figura generada por el comando anterior

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

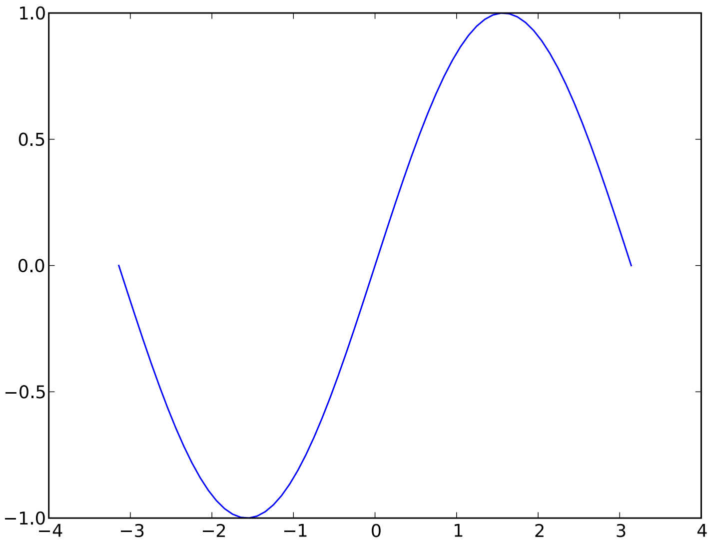

Figura 6.2: Figura generada por el comando anterior

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

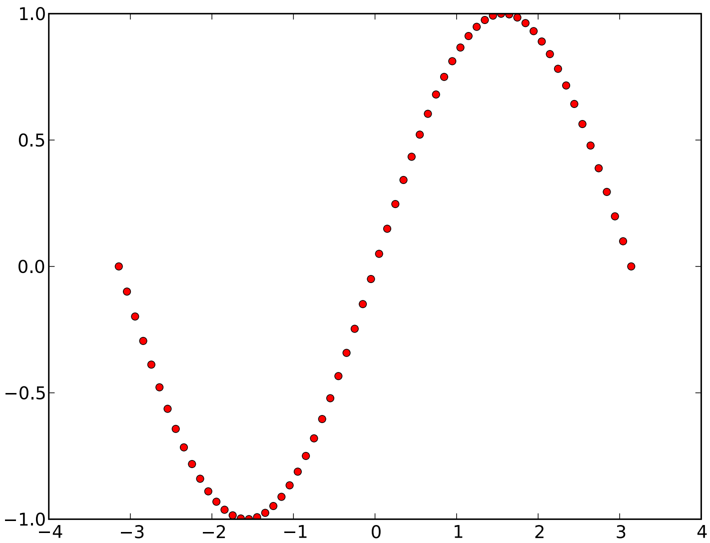

Figura 6.3: Figura generada por el comando anterior

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');
```

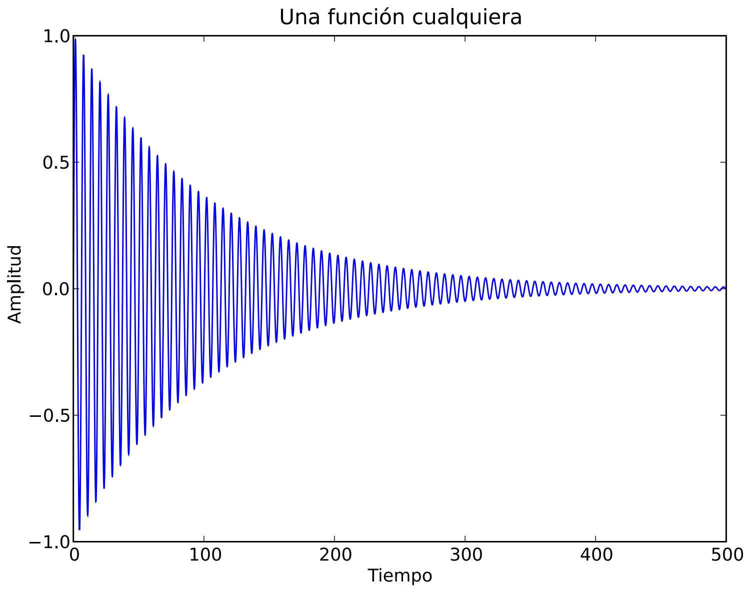

Figura 6.4: Ejemplo de etiquetas

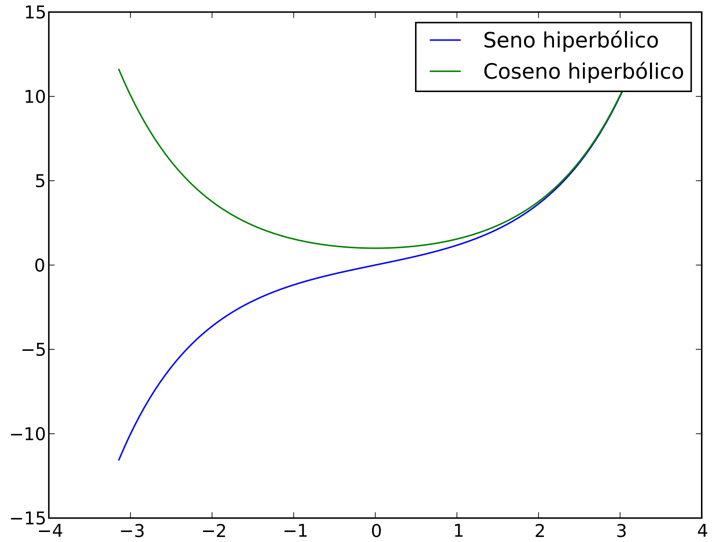

Figura 6.5: Ejemplo de aplicación de la leyenda

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));
```

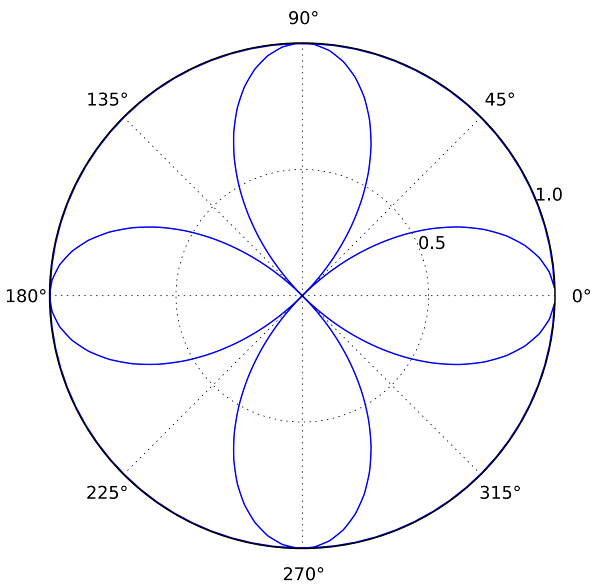

Figura 6.6: Ejemplo de gráfica en coordenadas polares

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');
```

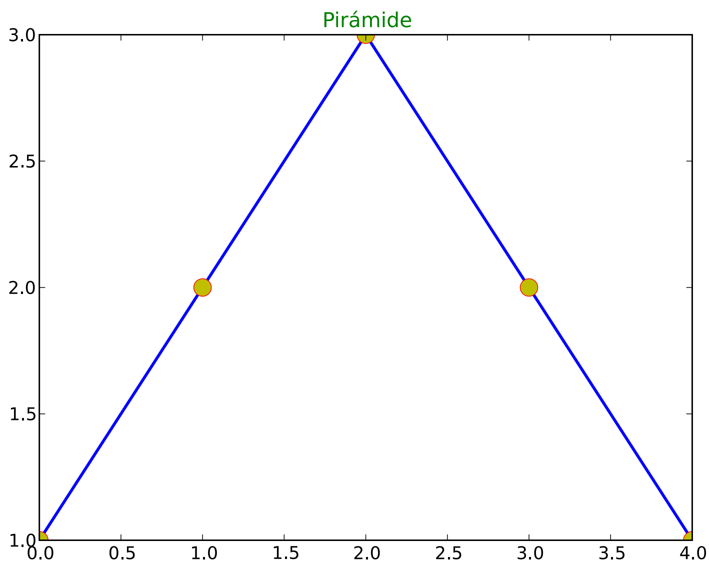

Figura 6.7: Ejemplo de uso de `set`

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')
```

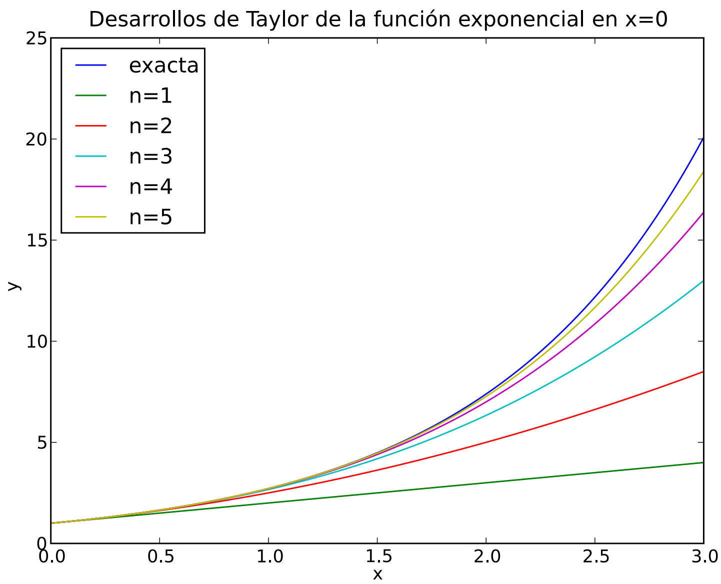

Figura 6.8: La función exponencial y sus desarrollos de Taylor en el origen hasta orden 5.

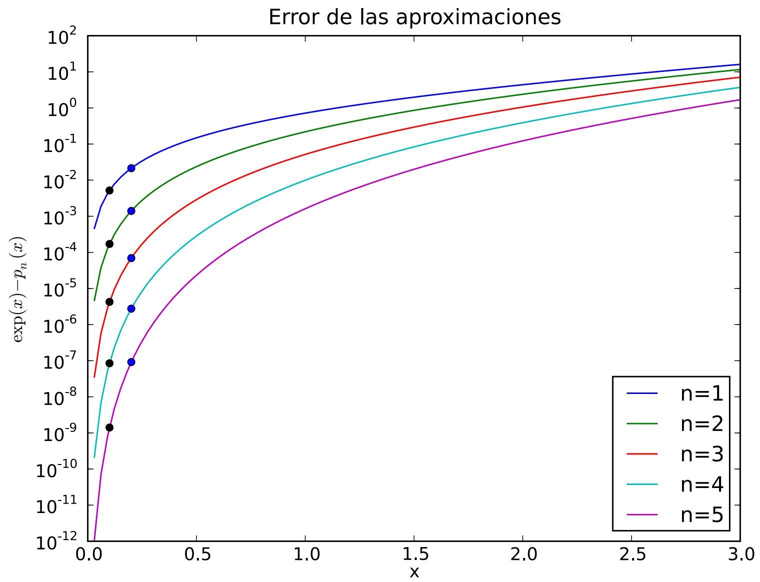

Figura 6.9: Error del desarrollo. Los puntos son los valores en las abcisas x=0.1 y x=0.2 respectivamente

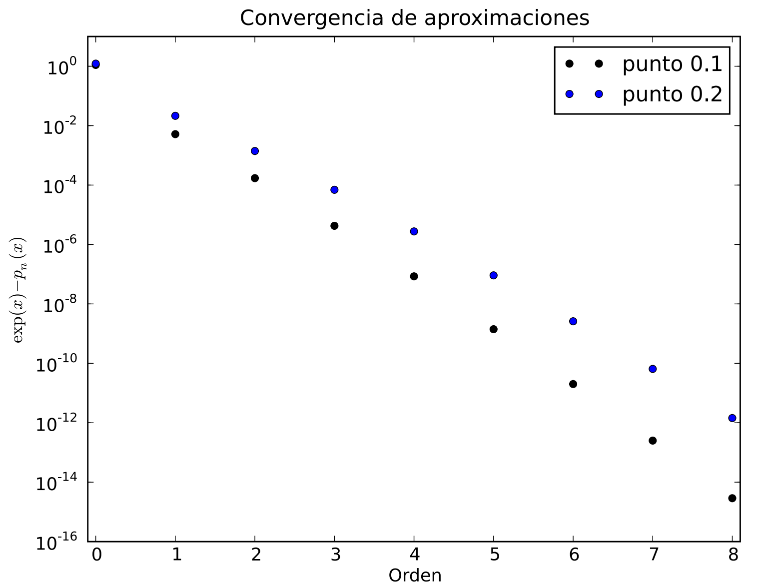

Figura 6.10: Convergencia en función del orden.

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')
```

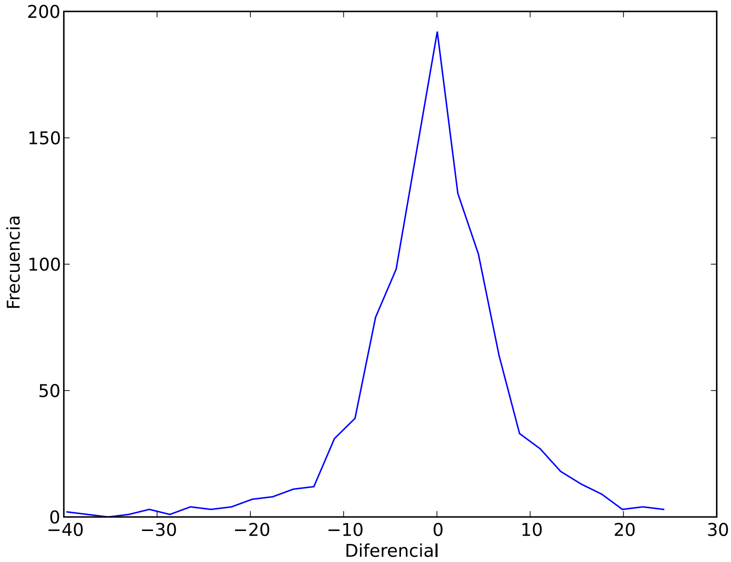

Figura 7.1: Histograma de los diferenciales del stock de Google

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 defineción 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.

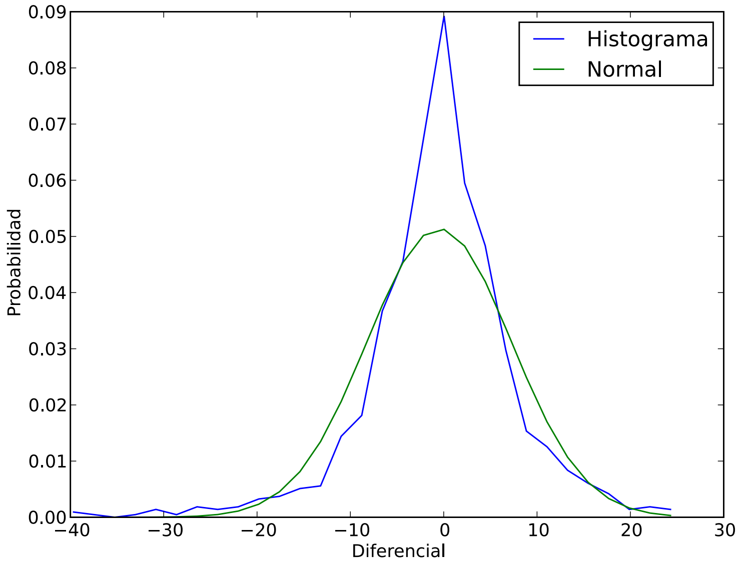

Figura 7.2: Comparación con la FDP Normal

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
```

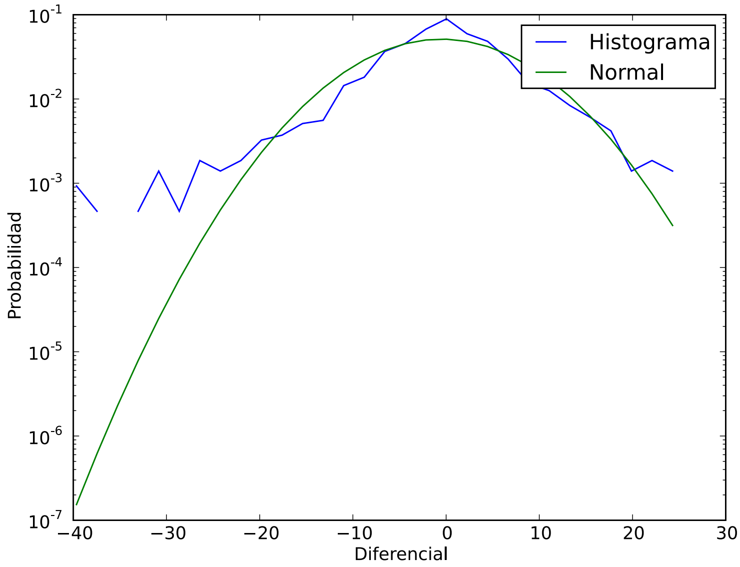

Figura 7.3: Comparación de las colas de la FDP

```
% 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 mediante la fórmula

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

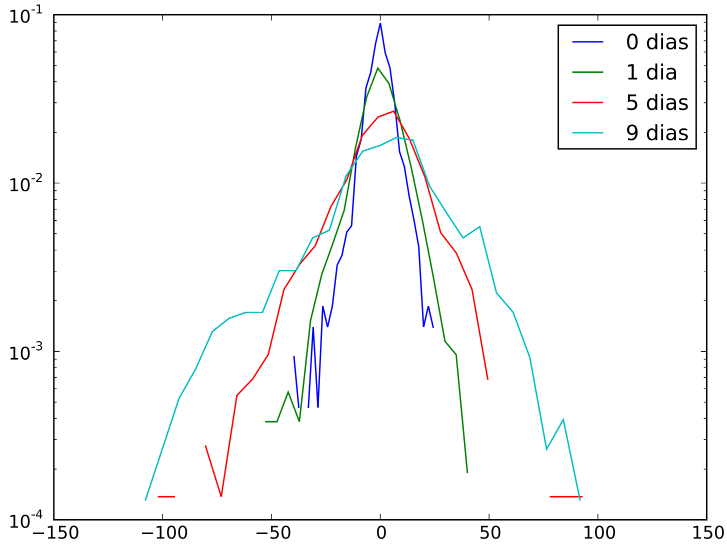

Figura 7.4: Colas anchas debidas a la intermitencia

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

CAPÍTULO 8

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*

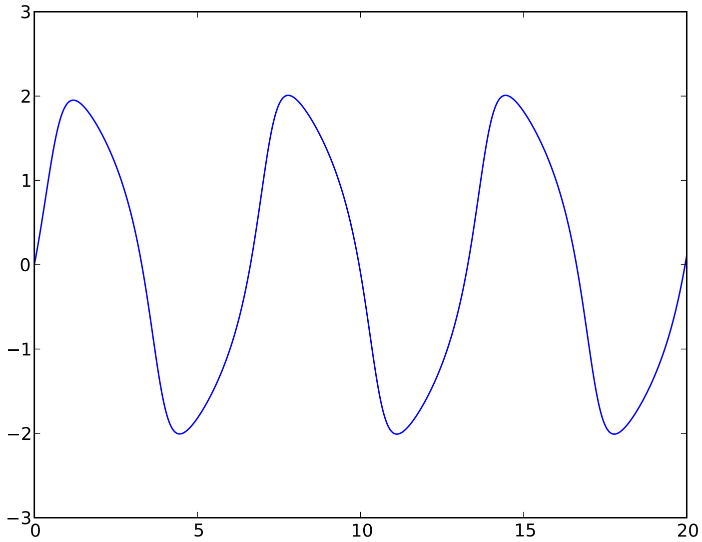

Figura 8.1: Solución del oscilador de Van der Pol para $\mu = 1$

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

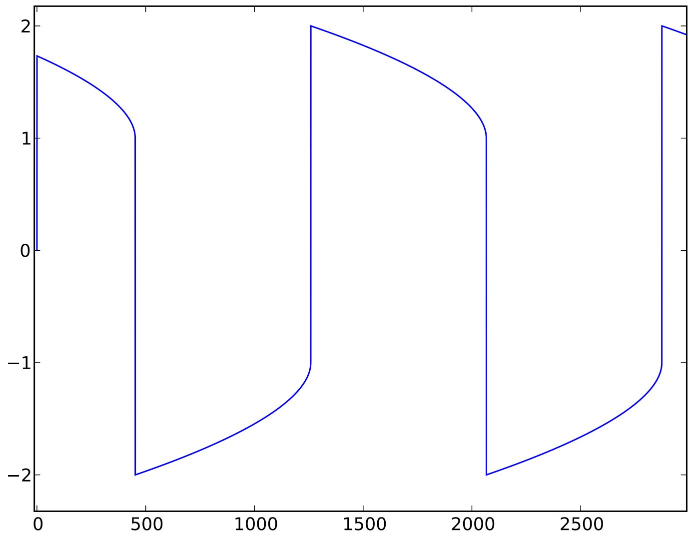

Figura 8.2: Solución del oscilador de Van der Pol para $\mu = 1000$

```
[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.

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

CAPÍTULO 10

Diseño de aplicaciones en Matlab

En ninguno de los muchos libros, cursos, introducciones o tutoriales a Matlab se trata un tema tan esencial como la manera correcta de estructurar un programa en Matlab.

Matlab no es, por desgracia, un lenguaje pensado para grandes aplicaciones sino para pequeños prototipos sin demasiada importancia. En las últimas versiones, gracias a la mejora de su orientación a objetos, ha ganado algo de entidad al respecto pero aún dista de poder ser considerado un lenguaje de propósito general. Matlab se ha acercado a C++ y Java pero va muy a la zaga de los lenguajes dinámicos más modernos como Python o Ruby.

Se pueden diseñar correctamente aplicaciones de cierto tamaño independientemente del paradigma utilizado. El sistema bancario mundial se programó en COBOL antes que los objetos o los módulos hicieran acto de presencia por primera vez. Esta sección propone una manera sistemática de reducir cualquier problema a estructuras propias de Matlab intentando que ninguna parte del resultado parezca ajena al lenguaje.

No es más que la enésima propuesta de cómo se debe programar en Matlab así que nada la mantiene a salvo de posibles detractores. Puede ser mejor o peor que otros métodos pero nunca hay que olvidar que un mal método siempre es mejor que ninguno.

10.1 Los dos axiomas

Los paradigmas de programación son una manera sistemática de reducir la implementación de cualquier algoritmo. Existe la programación procedimental, la modular, la orientación a objetos... Pero un paradigma es lo suficientemente extenso como para permitirnos implementar un algoritmo de muchas maneras distintas. Este capítulo no trata sobre paradigmas, hemos escogido de antemano un acercamiento modular; propone una manera sencilla y sistemática de reducir la mayoría de problemas a un conjunto conectado de bloques.

La programación es un ejercicio creativo y como tal es mucho más fácil valorar si el resultado está bien o mal que proponer de antemano la mejor manera de resolver el problema. Es entonces un problema más fácil de verificar que de resolver.

Es también un proceso de diseño en el que de una serie de piezas, en este caso un lenguaje, creamos la solución a un problema. El resultado del proceso es código.

El diseño axiomático es la disciplina que pretende trasladar lo que se aprende de valorar el resultado para que, antes de empezar a trabajar, nos acerquemos más fácilmente a una solución más óptima de las infinitas posibles. Toda la experiencia en el análisis de los procesos de diseño se ha reducido a dos axiomas:

- Máxima independencia.
- Mínima información.

Entonces la mejor manera de implementar un algoritmo será maximizando la independencia de las unidades de programa y minimizando la información necesaria para que funcionen.

Máxima independencia significa que cada una de sus piezas estará mejor diseñada en la medida que su funcionamiento sea independiente de las demás. En palabras de Albert Einstein "hay que simplificar las cosas pero no ser simplista" .Introducir dependencias entre elementos, aunque pueda reducir su número, suele producir sistemas menos robustos.

En el código de un programa, si declaramos una variable global en el programa principal y la utilizamos en todas las unidades de programa nuestro código será monolítico por mucho que nos parezca estructurado porque esta variable global lo liga absolutamente todo. Por este mismo motivo se demonizó durante los años setenta la cláusula GOTO puesto que ligaba el algoritmo a la manera en la que se había escrito.

La información de un sistema es la cantidad de datos necesarios para construirlo y asegurar su funcionamiento. En un mecanismo la cantidad de información se correspondría a todos los datos mostrados en el plano: medidas, tolerancias, relaciones... La información es lo que valora la simplicidad del producto.

Este axioma es un poco más complejo en nuestro caso porque es difícil definir la cantidad de información que contiene un código. Mi visión personal es que el propio código es información así que en la medida que nuestro código sea más extenso necesitaremos más información para entenderlo y hacerlo funcionar.

Quen tenga un poco de experiencia programando sabrá que estos dos axiomas entran en conflicto. Los códigos puramente monolíticos suelen ser más cortos que los modulares. Romper un algoritmo en rutinas y funciones para maximizar la independencia requiere tiempo y esfuerzo y hace que el programa crezca, aumentando por lo tanto la información que contiene. Una buena metodología de programación encontrará el compromiso entre estos dos axiomas.

Este capítulo propone una metodología para el diseño de aplicaciones con Matlab utilizando la programación modular. Para intentar cumplir los dos axiomas buscará:

- Definir unidades de programa independientes a partir de los datos.
- Sistematizar el desarrollo para que sea aplicable en cualquier caso.

10.2 Parámetros y variables.

La pieza fundamental de cualquier algoritmo en matemáticas son las funciones. Estas funciones dependen de parámetros, que no cambian en el algoritmo, y variables. Por ejemplo, si queremos saber cómo cambia la presión de un gas en función de la temperatura a bajas presiones utilizaremos la ecuación de estado para los gases perfectos

$$p(\rho, T) = R_g \rho T$$

Las variables de esta ecuación son la densidad y la temperatura mientras que la constante de los gases ideales R_g depende del gas para el que calculemos la relación. Será por lo tanto un parámetro.

La diferencia esencial entre variables y parámetros es que nuestra solución depende de los parámetros mientras que las variables son *parte de la solución*. En este caso la ecuación de estado *debe depender de la densidad y la temperatura*. Si nuestro resultado no depende de ninguna variable será una constante pero desgraciadamente no será así.

Esto nos permite crear dos reglas sencillas para descomponer cualquier algoritmo en unidades de programa abstractas llamadas bloques:

- Un bloque depende de parámetros y de funciones.
- Los argumentos de salida de los bloques son funciones.

Para entender el uso de estas dos reglas las aplicaremos a varios casos de dificultad creciente. Para empezar tomaremos la ecuación del oscilador de Van der Pol.

Supongamos que nos piden la solución de esta ecuación diferencial no lineal de segundo orden.

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

Para ello nos sugieren representar gráficamente la solución para un conjunto de diez valores de μ. Se nos pide resolver el problema de la integración de una EDO diez veces con un parámetro, algo que para un principiante puede representar un serio problema.

El primer paso es diferenciar los parámetros de las variables. En este caso μ es el único parámetro mientras que x y el tiempo t son las variables.

El bloque creará una función que dependerá de *x* y *t* a partir del parámetro μ. Para ello utilizaremos el handle `@` de Matlab para hacer que el argumento de salida sea una función.

```
function y = modvdp(mu)
%%
%% Modulo.  Generador de la ecuacion de Van der Pol
%%
%% Argumentos de entrada:
%%
%%  mu: parametro del amortiguamiento no lineal
%%
%% Argumentos de salida:
%%
%%  y: @(t,x) Ecuacion de Van der Pol

y = @(t,x) [x(2); mu*(1-x(1).^2)*x(2)-x(1)];
```

Truco: Cuando un módulo sólo proporciona una función puede reducirse a una única función anónima. Por ejemplo en el caso del módulo anterior

```
>> modvdp = @(mu) @(t,x) [x(2); mu*(1-x(1).^2)*x(2)-x(1)];
```

Ahora podemos utilizar el argumento de salida del módulo `modvdp` como argumento de entrada de la función de integración ya sea `lsode` en Octave o `ode45` en Matlab.

Si analizamos esta manera de implementar la física del problema con los dos axiomas vemos que el bloque es completamente independiente del resto de funciones y que lo único que necesita para funcionar es el parámetro μ.

A continuación aplicaremos estas dos reglas a ejemplos de dificultad creciente.

10.2.1 Ejemplo. La atmósfera estándar (ISA)

Una de las simplificaciones más sencillas para tratar teóricamente las capas más bajas de la atmósfera es suponer que es un gas perfecto en reposo. Esta simplificación, que no es demasiado adecuada en la zona más cercana al suelo, sí es acertada en alturas de entre doscientos y diez mil metros si se le añade la condición que existe una variación exponencial de la temperatura con la altura. Este modelo estático se conoce como atmósfera estándar o atmósfera ISA. Está gobernada por las siguientes condiciones:

- Gradiente térmico $\lambda = -6{,}5 \cdot 10^{-3} \frac{K}{m}$
- Gravedad a nivel del mar $g = 9{,}81$
- La constante de los gases perfectos para el aire es $R = 287 \frac{J}{kgK}$
- Temperatura a nivel del mar T_0
- Presión a nivel del mar p_0

$$T(h) = T_0 + \lambda h$$

$$p(h) = p_0 \left(\frac{T_0 + \lambda h}{T_0} \right)^{\frac{g}{R\lambda}}$$

$$\rho(h) = \rho_0 \left(\frac{T_0 + \lambda h}{T_0} \right)^{\frac{g}{R\lambda}-1}$$

A estas ecuaciones hay que sumarle el modelo de gas perfecto.

La variable del problema es la altitud h. Siguiendo las dos leyes, la atmósfera estándar podrá convertirse en un bloque que dependerá de los parámetros y que tendrá como argumentos de salida las funciones que dependen de las variables. Una posible implementación sería la siguiente:

```
function [T,p,rho] = ISA(T0,p0)
%
% Modulo que implementa la atmosfera estandar o ISA.
%
% Parametros:
%
%  T0: Temperatura a nivel del mar (altitud cero).
%  p0: Presion a nivel del mar.
%
% Argumentos de salida:
%
%  T: @(h) Temperatura en funcion de la altitud
%  p: @(h) Presion en funcion de la altitud
%  rho: @(h) Densidad a nivel del mar
%

%%% Constantes:

g = 9.81; % Aceleracion de la gravedad a nivel del mar [M s^-2]
Ra = 287; % Constante de gas perfecto para el aire [J kg^-1 K^-1]
lambda = -6.5e-3; % Gradiente termico [K m^-1]

rho0 = p0/Ra/T0; % Densidad a nivel del mar

T =   @(h) T0 + lambda .* h;                           % Temperatura(h)
p =   @(h) p0   .* ( T(h)./T0 ) .^ (-g/Ra/lambda);     % presion(h)
rho = @(h) rho0 .* ( T(h)./T0 ) .^ (-g/Ra/lambda - 1); % densidad(h)
```

El bloque `ISA` crea las funciones correspondientes a la temperatura, la presión y la densidad en función de la altitud. Se usa como se muestra a continuación

```
>> [T,p,rho]=ISA(298,101325);
>> T(1500)

ans =

  288.2500

>> rho(1500)

ans =

  1.0282
```

10.2.2 Ejemplo. El saque de Andy Roddick

La final de la Copa Davis de 2004 se jugó en Sevilla y enfrentó los equipos de España y Estados Unidos. La elección no estuvo exenta de controversia puesto que la candidatura de Sevilla adujo contra la de Madrid que al estar prácticamente a nivel del mar los jugadores tendrían menos dificultades para restar el saque de Andy Roddick. Hacer una simulación realmente precisa del problema para llegar a una conclusión no es necesario puesto que con un poco de análisis dimensional se llega a que el tiempo de vuelo de la pelota es proporcional a la densidad pero para demostrar toda la teoría de diseño de aplicaciones resolveremos el problema completo.

Supongamos que el equipo español de copa Davis quiere saber la diferencia del tiempo de vuelo entre servicio y resto en función de la altura a la que se juegue el partido. Para ello nos pide un estudio pormenorizado en el que se tendrán en cuenta factores como:

- Las características de la atmósfera.

- El desprendimiento de la capa límite alrededor de la bola.
- La velocidad y dirección inicial de la bola sin efectos.

Finalmente nos definen el tiempo de vuelo como el tiempo que transcurre desde el impacto con la raqueta hasta que cruza la vertical del final de la pista en el lado del resto después de un bote completamente elástico en el cuadro de saque. Estiman también que un saque de Andy sale de su raqueta a 200 kilómetros por hora.

Este problema nos servirá no sólo para ejemplificar el significado de las dos leyes, también será un ejemplo sobre cómo debe afrontarse el problema de simular un sistema físico. Esta recomendación es válida en prácticamente cualquier ocasión y puede tomarse como una tercera ley: *siempre hay que adimensionalizar todas las ecuaciones*.

Empezamos planteando la tercera ley de Newton:

$$\sum \vec{F} = m\ddot{\vec{x}}$$

que para cada una de sus componentes será

$$\begin{aligned} m\ddot{x} = & \quad -D\cos\theta \\ m\ddot{y} = & \quad -mg - D\sin\theta \end{aligned}$$

La descomposición anterior es evidente visto el diagrama de la figura.

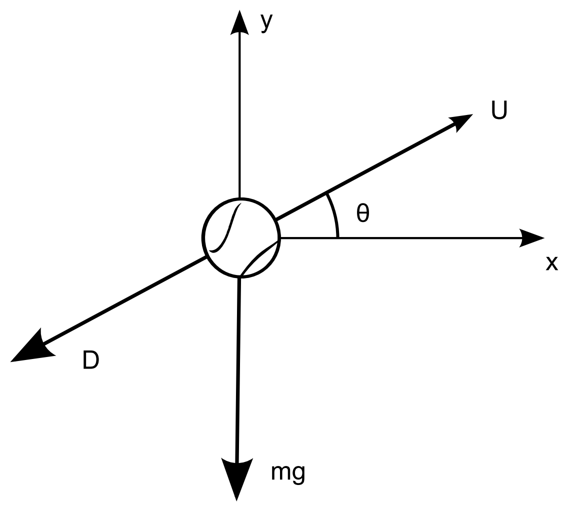

Figura 10.1: Diagrama del movimiento de la pelota

El paso siguiente es utilizar el teorema π, para el que hay que escoger magnitudes características para la masa, la longitud y el tiempo.

$$[M] \propto m, [L] \propto l, [T] \propto \frac{L}{U_0}$$

Llamaremos l a la longitud de la pista de tenis y U_0 a la velocidad inicial del saque. Estas magnitudes sirven para adimensionalizar las incógnitas y legar a las ecuaciones adimensinalizadas

$$\begin{aligned} \ddot{\xi} = & \quad -\frac{Dl}{mU_0^2}\cos\theta \\ \ddot{\eta} = & \quad -\frac{gl}{U_0^2} - \frac{Dl}{mU_0^2}\sin\theta \end{aligned}$$

que pueden reescribirse como

$$\begin{aligned} \ddot{\xi} = & \quad -\delta\cos\theta \\ \ddot{\eta} = & \quad -\gamma - \delta\sin\theta \end{aligned}$$

teniendo en cuenta además que

$$\cos\theta = \frac{\dot{\xi}}{\sqrt{\dot{\xi}^2 + \dot{\eta}^2}}, \quad \sin\theta = \frac{\dot{\eta}}{\sqrt{\dot{\xi}^2 + \dot{\eta}^2}}$$

De este modo toda la física del problema queda reducida a una ecuación, una incógnita $(\xi, \eta, \dot{\xi}, \dot{\eta})$ y los parámetros δ y γ. Resolver el problema pasará por crear los bloques necesarios y utilizar las funciones para realizar los cálculos.

La dificultad adicional de este caso es que estos parámetros pueden depender de otros parámetros con lo que el número de bloques crecerá a medida que nuestro algoritmo se vaya complicando. Es más; podemos encontrarnos que algunos de estos parámetros dependan de la propia variable con lo que en realidad serán funciones.

El primero que abordaremos será $\delta = \frac{Dl}{mU_0^2}$. Es el parámetro dependiente de la resistencia aerodinámica de la pelota. Esta resistencia puede descomponerse en el producto de una fuerza característica con un coeficiente llamado *coeficiente de resistencia parásita* o c_d. La expresión general es

$$D = q_\infty c_d = \frac{1}{2}\rho U_\infty^2 S c_d(Re)$$

donde S es la superficie frontal de la pelota, U_∞ es la velocidad de la pelota y Re es el número de Reynolds del movimiento. El parámetro δ será entonces

$$\delta = \frac{1}{8}\frac{\pi d^2 l}{m}\frac{U^2}{U_0^2}\rho(h)c_d(Re)$$

De este parámetro surgen dos funciones, la primera es la densidad que dependerá de la altitud y el segundo es el coeficiente de resistencia parásita, dependiente del número de Reynolds que a su vez depende de la densidad. Por suerte este coeficiente ya ha sido calculado y tabulado con anterioridad. Debemos tener en cuenta que la superficie de la pelota de tenis es rugosa y el flujo a su alrededor será turbulento o estará cerca de la transición de flujo laminar o turbulento. Puede encontrarse un estudio exhaustivo sobre el tema en [MEH]. De [ME2] puede extraerse que el c_d se mantiene constante y entre 0.6 y 0.7 en el rango de número de Reynolds de interés. De este modo reescribimos el parámetro δ como

$$\delta = \frac{1}{8}\frac{\pi d^2 l}{m}(\dot{\xi}^2 + \dot{\eta}^2)\rho(h)c_d$$

Advertencia: En ningún momento se ha diferenciado parámetros de valor constante de parámetros de valor variable. En el segundo caso el parámetro será una función cuyos argumentos son otros parámetros e incluso las incógnitas. Lo que define un parámetro no es su valor sino el uso que se hace de él. En este caso, δ es un parámetro porque no es una incógnita de la ecuación del sistema aunque dependa tanto de otros parámetros como de incógnitas y constantes.

Todo el problema puede reducirse entonces a bloques que se relacionan por los parámetros como se ve en la figura. Las cajas son funciones o módulos que de alguna manera dependen de las incógnitas. Los nombres rodeados por corchetes son constantes y los rodeados por paréntesis son parámetros.

En el diagrama se puede apreciar que en el fondo un progrma no es más que bloques que procesan parámetros hasta la función que resuelve el problema, en este caso integrar una ecuacion diferencial ordinaria no lineal. Ya conocemos uno de ellos, el bloque *ISA*, definido en el ejemplo anterior.

Ha llegado el momento de empezar a fijar las constantes del problema:

- La pelota de tenis tiene las siguientes características:
  - Debe pesar entre 56 y 59.4 gramos. Tomaremos 57.5
  - Su diámetro debe ser de 65.41 a 68.58 mm. Tomaremos 67
- La aceleración de la gravedad será de 9.81 $\frac{m}{s^2}$
- El coeficiente de resistencia parásita será de 0.65
- Las características de la pista pueden leerse de la figura siguiente

Pero todo lo contado con anterioridad no es más que la formulación para el movimiento de la pelota. El problema planteado no es saber cómo se mueve sino el tiempo de vuelo de un saque perfecto de Andy Roddick en función de la altitud. Conocer la formulación del movimiento es sólo el cálculo básico, tendremos también que descubrir las condiciones iniciales para el saque perfecto y luego volver a resolver el problema con ellas para una cantidad suficiente de altitudes.

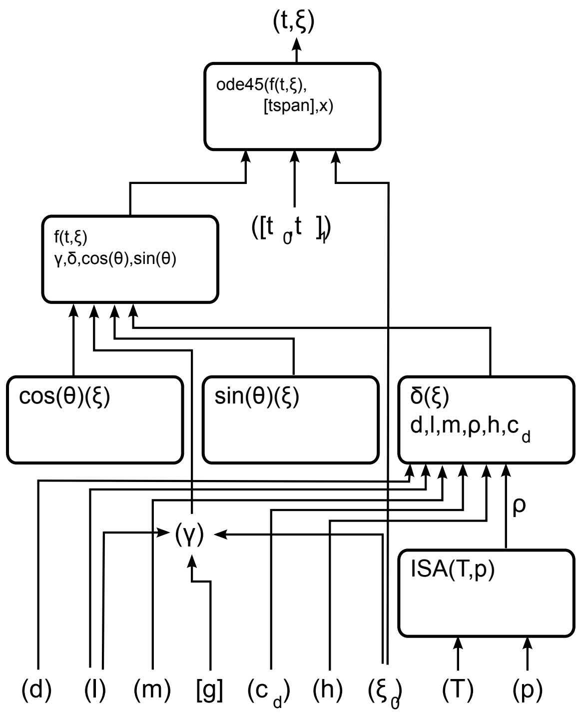

Figura 10.2: Diagrama de bloques del programa

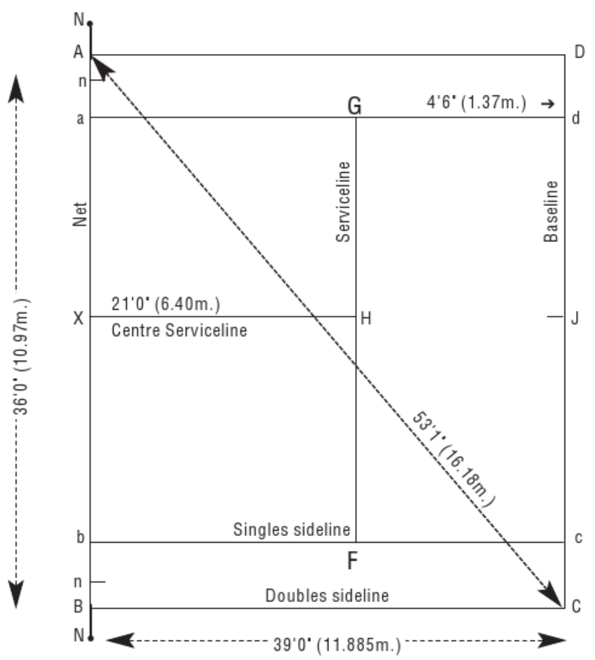

Figura 10.3: Esquema de las medidas de una pista de tenis

Entonces nuestro problema no es un problema de Cauchy sino un problema de contorno. No nos basta con integrar el problema dadas unas condiciones iniciales sino que nuestras condiciones de contorno también fijan la posición de la bola en un instante determinado, en este caso sabemos que debe botar al final del cuadro de saque. Para resolver este problema utilizaremos la técnica del *shooting*. Tenemos que saber para qué ángulo de saque θ_0 se consigue el saque perfecto teniendo en cuenta que se trata de una condición inicial del problema de Cauchy. Hay que conseguir una ecuación de la forma

$$R = F(\theta)$$

donde R es un resíduo que debe ser cero cuando la bola bote exactamente donde debe.

Advertencia: Resolver una ecuación no lineal es una labor de artesanía. Muchas cosas pueden ir mal en el primer intento. Esto es debido a que la función que resuelve el sistema puede probar valores de nuestra variable que no esperábamos. Puede suceder que tome valores de θ_0 para los que la bola no llegue a botar y la iteración no llegue a ningún resultado o lo que es peor, que la función a evaluar nos diga que el valor está fuera de su dominio de interpolación. El hecho de haber adimensionalizado el problema es de gran ayuda por dos razones: sabemos que la bola tiene que botar en tiempos adimensionales τ del orden de la unidad siempre que se saque más o menos plano, entonces nos aseguraremos que la bola bote con seguiridad integrando hasta $\tau = 10$.

Entonces nos topamos con otra dificultad: el resultado de una ecuación diferencial son una serie de puntos en el tiempo y en el espacio, no funciones de las que se pueda buscar una raíz. Será necesario interpolar la solución para encontrar el preciso instante del bote, don de $\eta = 0$

Una vez hallado el tiempo para el bote perfecto todo deberá convertirse en una función que dependa de todos los parámetros del problema, crearemos el interfaz y problema solucionado.

```
%%% Script que modela el servicio de tenis

clear;
clf;

%% Constantes y parametros

m = 57.6E-2; % masa de la pelota [kg]
d = 67E-3; % diametro de la pelota [m]
cd0 = 0.65; % coeficiente de resistencia parasita [adim]
l = 11.885*2; % longitud de la pista [m]
lss = 11.885 + 6.4; % distancia al borde de cuadro de saque [m]

%% Condiciones iniciales tentativas para el saque

U0 = 55.5; % velocidad del saque inicial
y0 = 2.7; % altura del saque inicial

%% Creo el handle que depende solo de la altura y calculo
myservice = @(h) servicefunc(m,d,cd0,l,lss,h,U0,y0);
h = linspace(0,4000,10);
fond = zeros(10,1);
for i = 1:10
    fond(i) = myservice(h(i));
end

plot(h,fond);
xlabel('Altitud [m]');
ylabel('Tiempo de vuelo [s]');

function tfondo = servicefunc(m,d,cd0,l,lss,h,U0,y0)
% Funcion que calcula el tiempo de vuelo del servicio de una pelota
% de tenis seguno los argumentos siguientes
%
% Argumentos de entrada:
%
```

```
%   m: masa de la pelota de tenis
%   d: diametro de la pelota de tenis
%   cd0: coeficiente de resistencia parasita
%   l: longitud de la pista de tenis
%   lss: distancia desde el sacador al bote
%   h: altitud a la que se juega el partido
%   U0: velocidad inicial del saque
%   y0: altura del impacto con la bola desde el suelo
%
% Argumento de salida
%
%   tfondo: tiempo que tarda la pelota en cruzar la pista

%% Constantes

theta0 = -0.2; % Aproximacion inicial para fsolve
g = 9.81; % Aceleracion de la gravedad [m s^-2]

%% Adimensionalizacion
xiss = lss/l;
eta0 = y0/l;
xih = @(theta,eta) [0;eta;cos(theta);sin(theta)]; % handle para
                                 % condiciones iniciales

%% Atmósfera ISA
[T,p,rho] = ISA(289,101325); %Dia estandar

%% parametro gamma
gammah = @(g,l,U0) g.*l/U0.^2;

%% parametro delta
deltah = @(d,l,m,xi,rho,h,cd0) ...
    pi/8*d.^2.*l./m.*(xi(3).^2+xi(4).^2).*rho(h).*cd0;

%% funciones coseno y seno
costheta = @(xi) xi(3)./sqrt(xi(3).^2+xi(4).^2);
sintheta = @(xi) xi(4)./sqrt(xi(3).^2+xi(4).^2);

%% funcion a integrar

fh = @(delta,gamma,costheta,sintheta,t,xi) ...
    [xi(3);xi(4);-delta(xi).*costheta(xi);-gamma-delta(xi).*sintheta(xi)];

%% union de bloques
gamma = gammah(g,l,U0);
delta = @(xi) deltah(d,l,m,xi,rho,h,cd0);
f = @(t,xi) fh(delta,gamma,costheta,sintheta,t,xi);

%% Busco el angulo para el saque perfecto.
errbotef = @(theta) errbote(f,xih,theta,eta0,xiss);
thetaideal = fsolve(errbotef,theta0);

%% Integro hasta el bote
vopt = odeset('RelTol',1e-6,'AbsTol',1e-6);
xi0 = xih(thetaideal,eta0);
[t1,x1] = ode45(f,[0,1],xi0,vopt);
tbote = interp1(x1(:,2),t1,0);

%% Calculo tiempo y velocidad en el bote para rehacer las condicines
% inciales, luego obtengo el tiempo para llegar al fondo de la pista
dxi1 = interp1(t1,x1(:,3),tbote);
deta1 = interp1(t1,x1(:,4),tbote);
xi1 = [xiss;0;dxi1;-deta1];
```

```
[t2,x2] = ode45(f,[tbote,2],xi1,vopt);
tfondo = interp1(x2(:,1),t2,1);
tfondo = tfondo*l/U0; %tiempo con dimensiones

function y = errbote(f,xih,theta,eta0,xiss)
% Funcion que determina el error del bote en el saque
%
% Argumentos de entrada
%
%  f: @(t,xi) Ecuacion de la dinamica de la trayectoria
%  xih: @(theta,y0) Bloque que crea las condiciones iniciales
%  theta: Angulo de saque inicial
%  eta0: altura del impacto de la pelota respecto al suelo [adim]
%  xiss: distancia del punto objetivo al sacador [adim]
%
% Argumentos de salida
%
%  y: distancia entre el punto de bote y el objetivo

%% funcion que obtiene el punto de bote

bote = @(x) interp1(x(:,2),x(:,1),0);
xi0 = xih(theta,eta0);
vopt = odeset();
[t,x] = ode45(f,[0 10],xi0,vopt); %integro hasta tau=1
y = bote(x(:,[1,2]))-xiss;
```

Advertencia: Misteriosamente los resultados para Matlab y Octave del tiempo de vuelo no son exactamente iguales. Sigo buscando el motivo pero bien puede deberse a la configuración de los errores o a algún bug en la rutina que integra la ecuación diferencial.

A continuación se muestra una gráfica del tiempo de vuelo para el saque perfecto a varias alturas.

Gracias al resultado se puede comprobar que la diferencia entre sacar al nivel del mar y a 700 metros de altura es de una milésima de segundo, un tiempo menor al de la latencia del sistema nervioso central. Jugar en Sevilla o en Madrid por la altitud es indiferente.

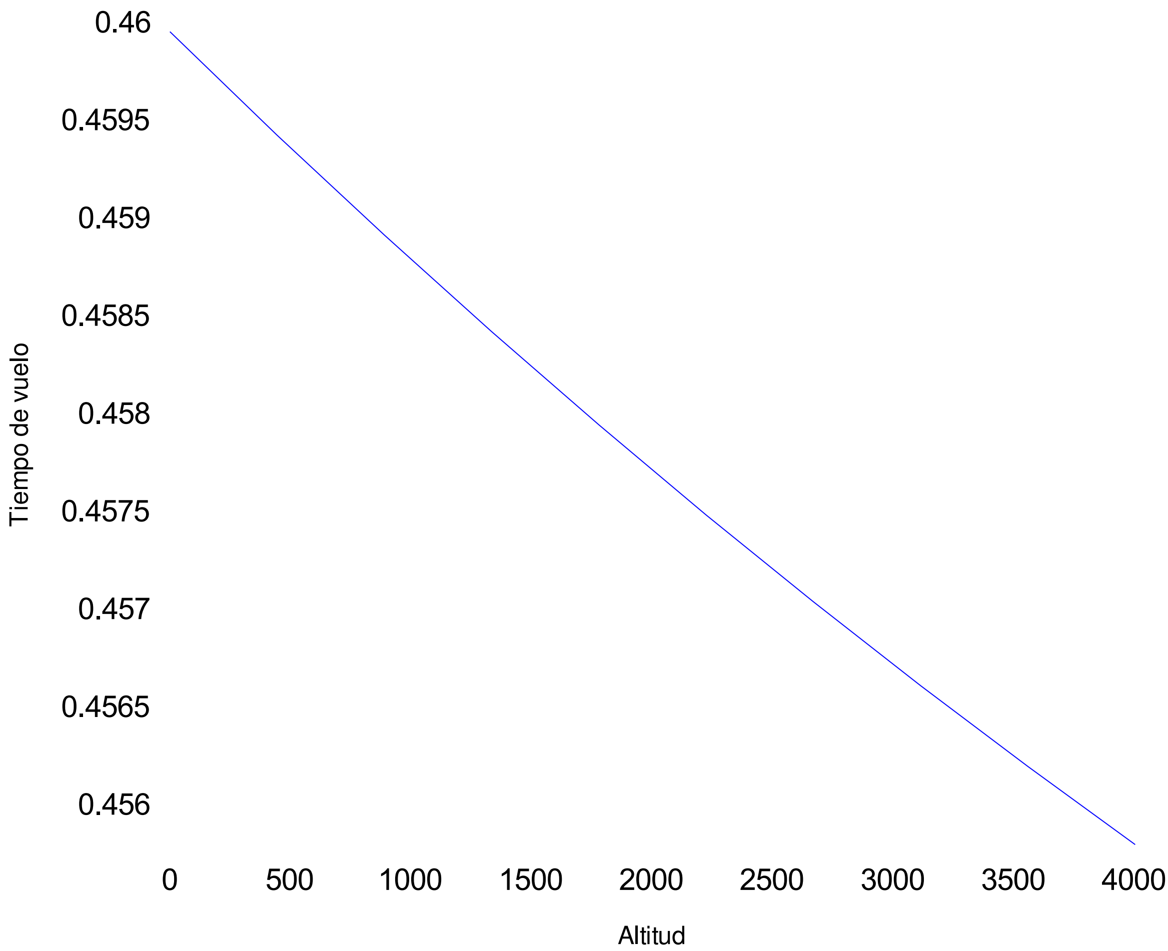

Figura 10.4: Resultado del script

CAPÍTULO 11

Persistencia

En programación, la persistencia se refiere a la capacidad de guardar el contenido de una variable en un archivo.

11.1 Archivos binarios

Podemos guardar los datos de dos maneras: utilizando un formato leíble para los humanos (normalmente ASCII) o utilizando un formato binario comprensible por el ordenador. Es necesario enfatizar que esto se trata únicamente de la manera de *escribir*, un archivo binario y otro en ASCII pueden contener exactamente los mismos datos. La diferencia es que mientras que podemos leer un archivo ASCII sólo viendo cómo está escrito es imposible realizar ingeniería inversa con un archivo binario.

Lo anterior significa que para leer un archivo binario tenemos que saber de antemano cómo leerlo. La solución más habitual a este problema es utilizar un formato conocido y explicitándolo en la extensión. Por ejemplo, si utilizamos el formato propio de Matlab podemos utilizar la extensión `.mat`, mientras que si utilizamos el formato HDF5 la extensión más habitual es la `.h5`.

Importante: Un archivo binario suele ocupar unas ocho veces menos espacio en memoria que uno en ASCII con los mismos datos.

Advertencia: Que el formato de archivo cambie cada versión implica que se rompe la compatibilidad hacia atrás. Si guardamos un archivo en Matlab 7.7 no podrá leerse con un Matlab 7.0 y obviamente tampoco con un Matlab 6.5. Si necesitamos soportar versiones anteriores de Matlab lo tendremos que indicar al utilizar el comando de escritura en disco correspondiente.
Obviamente sí existe compatibilidad hacia adelante, esto es, un archivo escrito por Matlab 6.5 podrá leerse sin problemas en Matlab 7.7.

CAPÍTULO 12

Estudio del péndulo invertido

Un péndulo es uno de los juguetes más básicos para experimentar los conceptos de periodo y gravedad. ¿Qué sucede si la masa se une a una barra rígida y se pone al revés? Entonces se obtiene un péndulo invertido, un sistema aparentemente inestable que es un ejemplo clásico para el control automático.

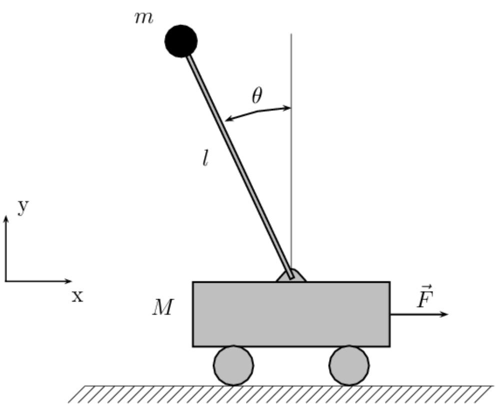

Figura 12.1: Esquema de un péndulo invertido con movimiento horizontal.

Una de las claves del péndulo invertido es intentar controlar el movimiento de la masa moviendo el otro extremo de la barra. En el ejemplo del carrito se demuestra que la barra se puede mantener en posición vertical para una perturbación dada lo suficientemente pequeña [INV].

Existe otra posibilidad, la de mantener la barra en posición vertical moviendo la base del péndulo también con una trayectoria vertical como se muestra en la figura siguiente:

En este caso, la masa tiene la siguiente posición:

$$(l\sin\theta, y + l\cos\theta)$$

y la siguiente velocidad:

$$(l\dot{\theta}\cos\theta, \dot{y} + l\dot{\theta}\sin\theta)$$

La lagrangiana del sistema es entonces:

$$L = \frac{1}{2}m\left(\dot{y}^2 + 2l\dot{\theta}\dot{y}\sin\theta + l^2\dot{\theta}^2\right) - mg(y + l\cos\theta)$$

Y la ecuación del movimiento:

$$l\ddot{\theta} - \ddot{y}\sin\theta = g\sin\theta$$

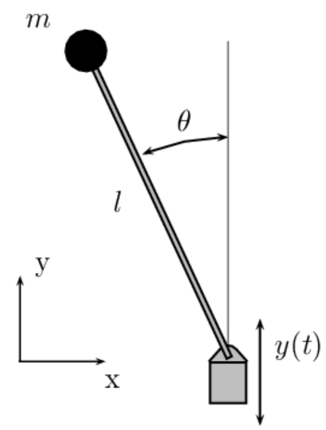

Figura 12.2: Esquema de un péndulo invertido con movimiento vertical.

El paso siguiente es suponer que el ángulo θ se mantiene pequeño en cualquier instante.

Supuesto un movimiento armónico de la base del péndulo $y = a \sin \omega t$, obtener el valor del parámetro $k = \frac{g}{\omega^2 a}$ para el que el péndulo deja de ser estable.

Representar gráficamente el movimiento de la partícula respecto al tiempo adimensional $\omega t = \tau \in [0, 4\pi]$ con *k* = 0.1 y *k* = 10. En todos los casos la longitud del péndulo será de 0.1 metro y la gravedad de 9.8 metros por segundo.

12.1 Solución

El concepto subyacente en este ejercicio es la importancia de adimensionalizar las ecuaciones. No es momento de nombrar las ventajas de utilizar el teorema π pero sí podemos demostrar que tras un poco de manipulación es sencillo llegar a conclusiones simplemente echándole un vistazo a la expresión resultante.

Pero antes hay que adimensionaliar, para ello escogemos una longitud característica a, la amplitud del forzado y un tiempo característico ω^{-1}. La aceleración característica será entonces $a\omega^2$

Con todo lo anterior se define un tiempo adimensional τ y una longitud adimensional λ. Adimensionalizando la ecuación anterior e introduciendo la expresión del forzado se obtienene la siguiente expresión

$$\lambda a\omega^2 \theta'' + a\omega^2 \sin \tau \sin \theta = g \sin \theta$$

Donde el operador $'$ significa la derivada respecto al tiempo adimensional τ. Linealizando la ecuación para ángulos pequeños

$$\lambda a\omega^2 \theta'' + (a\omega^2 \sin \tau)\theta = g\theta$$

Dividiendo ambos lados por $a\omega^2$...

$$\lambda\theta'' + (\sin \tau - k)\,\theta = 0$$

En la que finalmente aparece el parámetro propuesto $k = \frac{g}{\omega^2 a}$ que es el parámetro que relaciona la aceleración debida a la gravedad con las aceleraciones debidas al forzado.

Con el simple hecho de adimensionalizar podemos comparar la importancia del término derivada segunda respecto al término con θ mediante el siguiente número adimensional $\Delta(\tau) = \frac{\sin \tau - k}{\lambda}$

$$\theta'' + \Delta(\tau)\theta = 0$$

Con este primer número adimensional podemos comprobar que la ecuación diferencial se comportará de un modo u otro según su valor. Si $\Delta(\tau)$ es muy pequeño la ecuación resultante será

$$\theta'' = 0$$

Que tiene como solución $\theta(\tau) = A\tau + B$, un crecimiento lineal con el tiempo adimensional. Esto sucederá justamente cuando el numerador de $\Delta(\tau)$ se haga cero, esto es: $\sin\tau - k = 0$. En cualquier otro instante la solución será puramente armónica. Entonces parece razonable pensar que si el numerador nunca se anula entonces el sistema será estable. Para ello bastará que k sea mayor que 1 aunque como se verá a continuación no es un límite estricto.

Para plantear la ecuación se hace el cambio de variable $\dot\theta = u_2$ y $\theta = u_1$ para poder expresar la ecuación de segundo orden en forma de sistema de ecuaciones

---

Nota: Las condiciones iniciales del problema son importantes. Este sistema se encuentra en una posición de equilibrio inestable en $\theta = 0$, esto significa que si el sistema no se perturba en absoluto seguirá indefinidamente en dicha posición. En este caso cualquier perturbación, tanto en velocidad como en desplazamiento, será suficiente para sacar el sistema de su equilibrio pero podría suceder que fuera tan pequeña que no lo notara.

El sistema puede atenuar, mantener o amplificar la perturbación dependiendo de su propia naturaleza pero debemos tener muy en cuenta que la solución final dependerá de las condiciones iniciales y de la perturbación elegida.

$$\begin{aligned}\dot u_1 =& \quad u_2\\ \dot u_2 =& \quad \frac{\sin\tau - k}{\lambda} u_1\end{aligned}$$

Una solución al estudio del problema se presenta a continuación.

```
clear
clc

%% Solucion estable

tau = linspace(0,4*pi,100); % Tiempo adimensional

%%% Generador de la ecuacion:
%%% Parametros: k,l
%%% Variables: x,t

genpend = @(k,l) @(x,t) [x(2),x(1)*(sin(t)-k)/l];

%%%

                                % k = 10
                                % l = 0.1
theta = lsode(genpend(10,0.1),[0.01 0],tau); %Integracion

figure(1);
clf;

subplot(2,1,1);
plot(tau,theta(:,1));
xlabel('\tau')
ylabel('\theta')
title('k=10')

subplot(2,1,2);
plot(tau,sin(tau))
xlabel('\tau')
ylabel('y/a')

print -dpng 'k10.png'
print -deps 'k10.eps'

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Caso inestable
```

```
                                % k = 0.1
                                % l = 0.1
theta = lsode(genpend(0.1,0.1),[0.01 0],tau); %Integracion

figure(2);
clf;

subplot(2,1,1);
plot(tau,theta(:,1));
xlabel('\tau')
ylabel('\theta')
title('k=0.1')

subplot(2,1,2);
plot(tau,sin(tau))
xlabel('\tau')
ylabel('y/a')

print -dpng 'k01.png'
print -deps 'k01.eps'

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Comprobacion del limite de estabilidad de k=1

figure(3);
clf;
tau = linspace(0,10*pi,100); % Mas tiempo adimensional para comprobar la
                                % convergencia de la solucion

idx = 1;
klist = [0.9,1.1,1.2];

for k = klist
  theta = lsode(genpend(k,0.1),[0.01 0],tau); %Integracion
  subplot(3,1,idx);
  plot(theta(:,1),theta(:,2));
  xlabel('\tau');
  ylabel('\theta');
  title(['k=',+num2str(k)]);
  idx++;
end

print -dpng 'convergencia.png'
print -deps 'convergencia.eps'
```

Este script genera las tres figuras siguientes:

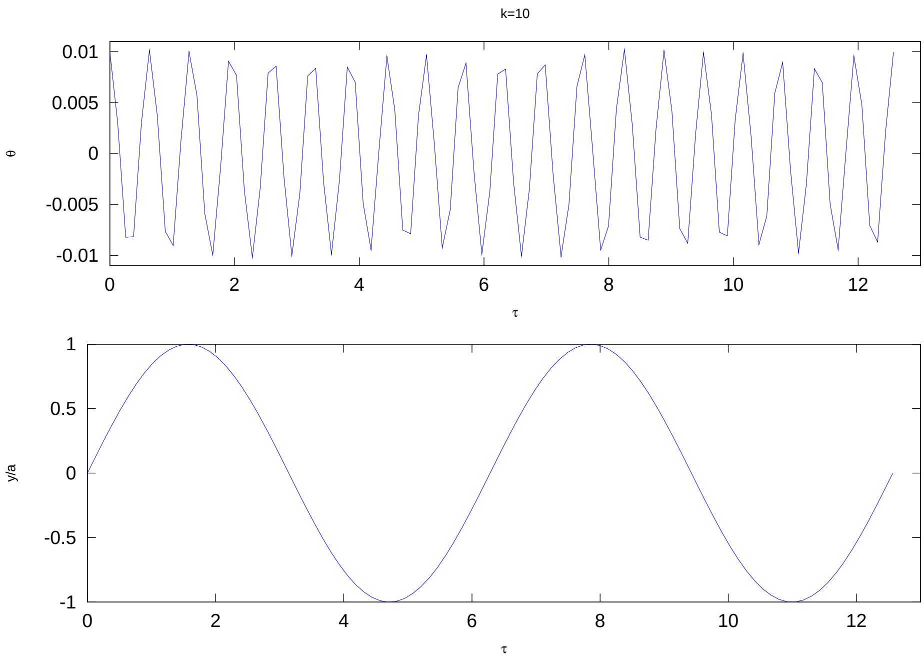

Figura 12.3: Ejemplo de solución en la que la perturbación no es amplificada por el forzado. La oscilación se mantiene prácticamente invariable en el tiempo.

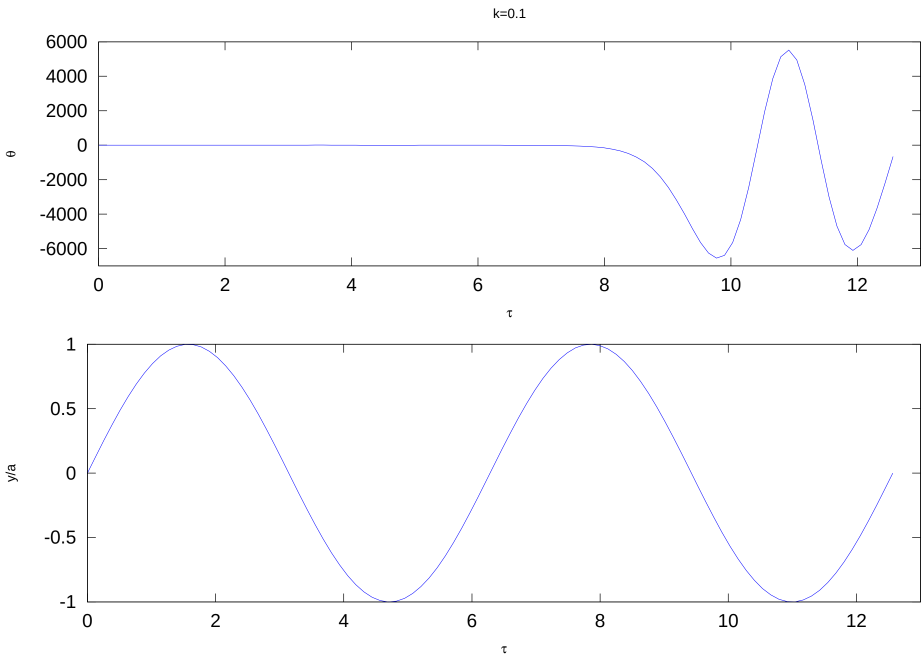

Figura 12.4: Solución con un valor de k que amplifica la perturbación inicial. La amplificación en el último periodo del forzado es tan grande que las primeras oscilaciones quedan fuera de escala.

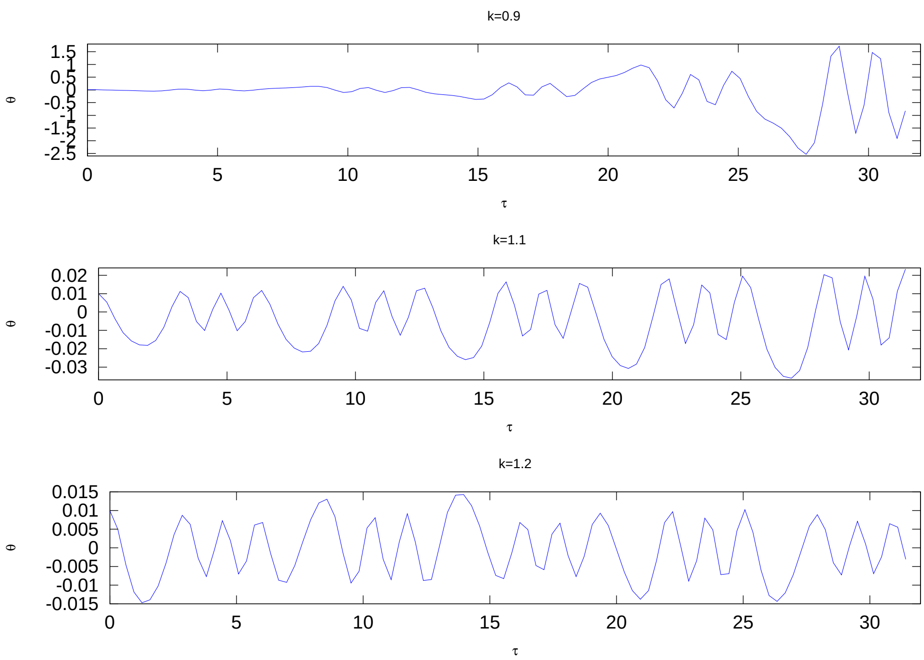

Figura 12.5: Solución de varios problemas con *k* distintas. Como se puede apreciar, con una *k* de 1.1 la perturbación crece haciendo que el sistema sea inestable. Esto significa que el límite estimado de $k = 1$ no es estricto.

CAPÍTULO **13**

Mandar un cohete al espacio

Autor de la sección: David Marchante López

La ecuación fundamental del movimiento vertical de un vehículo con motor cohete es la tercera ley de Newton

$$E - D - mg = \frac{d}{dt}(m\dot{h})$$

Esto es, el empuje menos la fuerza de rozamiento y la gravedad es igual a la variación de la cantidad de movimiento. Reescribiendo ligeramente esta ecuación llegamos a que

$$E - D = \dot{m}\dot{h} + \left(m_0 - \int_0^t \dot{m}\, dt\right)(\ddot{h} + g)$$

Es la ecuación que relaciona las fuerzas externas con la velocidad la aceleración y el gasto másico de combustible que abandona el cohete. Esta fórmula es válida para cualquier motor cohete ha sea de propulsante sólido o líquido.

La resistencia aerodinámica se define mediante el coeficiente de resistencia c_d

$$D = \frac{1}{2}\rho\dot{h}^2 S_f c_d$$

donde se conoce que el coeficiente de resistencia en vuelo puramente vertical es igual a la resistencia parásita que puede ser aproximada por esta función:

$$c_d = 2{,}6M^{1{,}1}\exp(-M) + 0{,}3\sqrt{M}$$

Luego el empuje se define mediante la expresión siguiente

$$E = p_c A_g \Gamma(\gamma)\sqrt{\frac{2\gamma}{\gamma-1}\left[1 - \left(\frac{p_s}{p_c}\right)^{\frac{\gamma-1}{\gamma}}\right]} + \frac{A_s}{A_g}\left(\frac{p_s}{p_c} - \frac{p_a}{p_c}\right)$$

La relación de presiones en la tobera para flujo no desprendido depende únicamente de la relación de áreas entre el área de la garganta y el área de salida

$$\frac{A_s}{A_g} = \frac{\Gamma(\gamma)}{\frac{p_s}{p_c}^{\frac{1}{\gamma}}\sqrt{\frac{2\gamma}{\gamma-1}\left[1 - \left(\frac{p_s}{p_c}\right)^{\frac{\gamma-1}{\gamma}}\right]}}$$

A su vez, el gasto másico del cohete se obtiene de la expresión de la velocidad de recesión de un combustible sólido dada por la siguiente fórmula

$$\dot{r} = kp_c^{0{,}7}$$

Sabiendo además que la velocidad de recesión a una presión de 50 atmósferas es de 0.2 centímetros por segundo. Suponiendo además que el combustible se quema frontalmente y que la superficie quemada es aproximadamente la superficie frontal del cohete se llega a la siguiente expresión para el gasto másico:

$$\dot{m} = \rho_c S_f k p_c^{0,7}$$

Se realizarán las siguientes hipótesis adicionales:

- El cohete asciende de modo completamente vertical por una atmósfera estándar.
- La presión en la cámara de combustión se mantiene constante e igual a 20 MPa.
- El área de la garganta será de 0.01 m^2 y el área de salida será igual al área frontal del cohete.
- En un instante inicial el cohete se encuentra posado en el suelo a nivel del mar y con el motor encendido a régimen estacionario con la presión de cámara de combustión igual a la de diseño.
- La masa inicial del cohete es de 55 kg de los cuales 50 corresponden al combustible.
- La densidad del combustible sólido es de 1800 kg/m^3.

Representar en función del tiempo la altura, la velocidad y la aceleración del cohete en su ascensión hasta que se termina el combustible para áreas frontales de 0.4, 0.6 y 0.8 m^2

13.1 Fórmulas adicionales

Se define la atmósfera ISA con las siguientes fórmulas

- Gradiente térmico $\lambda = -6{,}5 \cdot 10^{-3} \frac{K}{m}$
- Temperatura a nivel del mar $T_0 = 288K$
- Presión a nivel del mar $p_0 = 101325Pa$
- Densidad a nivel del mar $\rho_0 = 1{,}225 \frac{kg}{m^3}$
- La constante de los gases perfectos para el aire es $R = 287 \frac{J}{kgK}$

$$T(h) = T_0 + \lambda h$$

$$p(h) = p_0 \left(\frac{T_0 + \lambda h}{T_0} \right)^{\frac{g}{R\lambda}}$$

$$\rho(h) = \rho_0 \left(\frac{T_0 + \lambda h}{T_0} \right)^{\frac{g}{R\lambda} - 1}$$

La constante necesaria para calcular las relaciones de presión alrededor de toberas bloqueadas en régimen isentrópico es

$$\Gamma(\gamma) = \sqrt{\gamma} \left(\frac{2}{\gamma + 1} \right)^{\frac{\gamma+1}{2(\gamma-1)}}$$

con γ siempre igual a 1.4

13.2 Solución

En esta solución aprenderemos el que es quizás el método sistemático para modelar aplicaciones en Matlab. Se basa en dos normas esenciales

- En las funciones todo son argumentos.

- Los parámetros definien un módulo.

Cuando una variable se pone en la cabecera de la definición de una función pasa a ser un argumento de entrada. En principio la función dependerá sólo de estos argumentos pero esta imposición puede plantear serias dificultades. Supongamos que tenemos que integrar una ecuación diferencial que depende del tiempo, de una variable espacial y de un parámetro. La rutina que integra la ecuación diferencial impone que los únicos argumentos que puede tener la función a integrar son precisamente el tiempo y la variable espacial. Esto obliga a definir de algún modo alternativo el parámetro dentro de la función.

Para evaluar este parámetro dentro de la función existen dos posibilidades:

1. Evaluar una variable del espacio base con la función `evalin`.
2. Definir los parámetros como variables globales.

Ambos planteamientos tienen muchos inconvenientes y prácticamente ninguna ventaja. Si se utiliza la función `evalin` la función depende de cómo se escriba el programa principal y ya no puede ser reutilizada de ninguna manera. Definir variables globales dentro del código es un riesgo innecesario puesto que además del inconveniente anterior puede generar serios conflictos de nombres.

Parece que no hay ninguna manera de conseguir que una función dependa de un parámetro sin tener que adaptar cada una de ellas al problema en particular. Para encontrar la solución hay que utilizar un poco el pensamiento lateral: para cada función se define un interfaz. De este modo, una función que sólo espera argumentos puede adaptarse para que distinga entre argumentos y parámetros.

Este método utiliza una característica muy interesante de los *function handle*, si en ellos se define una variable que no se encuentra en la cabecera automáticamente la busca en el espacio de variables de la base. Sólo lanza un error si no la encuentra. Por ejemplo

```
>> f = @(x) a*x;
>> f(3)
error: 'a' undefined near line 1 column 10
error: called from:
error:    at line -1, column -1
>> a = 3;
>> f = @(x) a*x;
>> f(3)
ans =  9
```

Como se puede comprobar, si la variable `a` no existe aparece un error pero en el momento que se define `a` la función es capaz de encontrar la variable y la utiliza como parámetro. Este planteamiento es útil cuando, para resolver un problema dado, hay que utilizar una rutina que requiere una función con una forma determinada; por ejemplo la integración de una ecuación diferencial.

La ecuación de Van der Pol

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

depende del parámetro μ. Como es una ecuación de uso común para experimentar esquemas de integración temporal Matlab incluye las funciones `vdp1` y `vdp1000` donde μ es 1 y 1000 respectivamente.

Definir dos funciones cuando sólo las diferencia un parámetro es poco estético. Este problema se puede solucionar fácilmente si se define una única función y se cambia el parámetro en el espacio base.

```
% Octave
clear all
clc

mu = 1;
vdp = @(x,t) [x(2),mu*(1-x(1).^2)*x(2)-x(1)];
x  = linspace(0,20,1000);
y = lsode(vdp,[0 2],x);
p = plot(x,y(:,1));

mu = 1000;
x = linspace(0,3000,100000);
```

```
vdp = @(x,t) [x(2),mu*(1-x(1).^2)*x(2)-x(1)];
y = lsode(vdp,[0 2],x);
figure(2)
p = plot(x,y(:,1));

%Matlab
clear all
clc

mu=1;
vdp=@(t,x) [x(2); mu*(1-x(1).^2)*x(2)-x(1)];
[t,y]=ode45(vdp,[0 20],[0 2]);
p=plot(t,y(:,1));

mu=1000;
vdp=@(t,x) [x(2); mu*(1-x(1).^2)*x(2)-x(1)];
[t,y]=ode23s(vdp,[0 3000],[0 2]);
figure(2)
p=plot(t,y(:,1));
```

El planteamiento utilizado con los ejemplos anteriores tiene un punto débil. Los interfaces se evalúan en el momento de su definición, no en el momento de su ejecución. Si tras asignar `mu` como 1000 no se hubiera redefinido el interfaz este no hubiera cambiado de valor. Por ejemplo

```
>> a = 1;
>> y = @(x) a*x;
>> y(2)
ans =  2
>> a = 2;
>> y(2)
ans =  2
```

Esto obliga a redefinir cada interfaz después de cambiar cada parámetro, una estrategia que no parece demasiado adecuada porque fuerza a seguir una disciplina. En la programación si el programador está obligado a seguir una disciplina es porque está haciendo el trabajo del ordenador. El paso siguiente es eliminar esa necesidad y utilizar un concepto bastante conocido: los módulos.

Aquí llega la segunda norma de la programación en Matlab, los parámetros definien el módulo. Si se toma un módulo como un generador de funciones, los argumentos de entrada de un módulo serán los parámetros y los de salida las funciones. De este modo hemos unido la definición de los parámetros con la creación de los interfaces. Por ejemplo, este sería un módulo destinado a crear la función de Van der Pol

```
function [vdp] = modvdp(mu)
  vdp=@(t,x) [x(2); mu*(1-x(1).^2)*x(2)-x(1)];
```

El parámetro mu es el argumento de entrada del módulo y la función, que en este caso es la misma que el interfaz, es el argumento de salida. Si un módulo devuelve sólo una función este puede degenerar en una función anónima que devuelve otra función anónima

```
>> modvdp = @(mu) @(t,x) [x(2); mu*(1-x(1).^2)*x(2)-x(1)];
```

Otro ejemplo. La atmósfera ISA o atmósfera estándar se basa en suponer que el gradiente de temperaturas hasta los 11000 metros es lineal.

$$T(h) = T_0 + \lambda h$$

$$p(h) = p_0 \left(\frac{T_0 + \lambda h}{T_0} \right)^{\frac{g}{R\lambda}}$$

$$\rho(h) = \rho_0 \left(\frac{T_0 + \lambda h}{T_0} \right)^{\frac{g}{R\lambda} - 1}$$

Los parámetros asociados a la atmósfera son el gradiente de temperatura, la gravedad, la constante de gas perfecto del aire y la temperatura, la presión y la densidad a nivel del mar. Este caso es especialmente sencillo porque todas las funciones dependen del mismo argumento: la altitud. Una posible manera de implementar este módulo sería

```
function [T,p,rho] = ISA(T0,p0)
%
% Modulo que implementa la atmosfera estandar o ISA.
%
% Parametros:
%
%  T0: Temperatura a nivel del mar (altitud cero).
%  p0: Presion a nivel del mar.
%
% Argumentos de salida:
%
%  T: @(h) Temperatura en funcion de la altitud
%  p: @(h) Presion en funcion de la altitud
%  rho: @(h) Densidad a nivel del mar
%

%%% Constantes:

g = 9.81; % Aceleracion de la gravedad a nivel del mar [M s^-2]
Ra = 287; % Constante de gas perfecto para el aire [J kg^-1 K^-1]
lambda = -6.5e-3; % Gradiente termico [K m^-1]

rho0 = p0/Ra/T0; % Densidad a nivel del mar

T =   @(h) T0 + lambda .* h;                          % Temperatura(h)
p =   @(h) p0   .* ( T(h)./T0 ) .^ (-g/Ra/lambda);     % presion(h)
rho = @(h) rho0 .* ( T(h)./T0 ) .^ (-g/Ra/lambda - 1); % densidad(h)
```

En el script se han considerado que la gravedad, el gradiente térmico y la constante de gas perfecto del aire no son parámetros sino constantes así que no es necesario ni siquiera ponerlos en la cabecera del módulo. Sin embargo las condiciones meteorológicas a altitud cero son variables.

Este método puede utilizarse como punto de apoyo para articular cualquier programa en Matlab por grande que sea. Los parámetros podrán agruparse según el significado físico que tengan. En el caso del motor cohete, algunos parámetros se referirán a la geometría de la tobera, otros a las condiciones ambientales, otros a las características del combustible... Cada familia de parámetros creará un módulo con las funciones o los interfaces necesarios para resolver el problema.

Vamos a aplicar estos conceptos al problema del motor cohete.

```
%%% Programa de simulacion vehiculo con motor cohete a p_c constante
%%% Version para Matlab

clear;

%% ISA
g = 9.81;                  % gravedad terrestre [m/s^2]
gair = 1.4;                % gamma aire
Ra = 287;                  % constante MGP aire [J/kg/K]

[T,p,rho] = ISA(288,101325);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Especificaciones Cohete
p_c = 20e6;                % presion camara de combustion   [Pa]
Sf = 0.4;                  % area frontal                   [m^2]
Ag = 0.01;                 % area garganta                  [m^2]
As = Sf;                   % area salida                    [m^2]
mcohe = 25;                % masa cohete                    [kg]
```

```
mcomb = 30;                 % masa inicial combustible      [kg]
m0    = mcohe + mcomb;      % masa inicial total            [kg]
rho_c = 1800;               % densidad combustible solido   [kg/m^3]

%%% Hallo el consumo
k  = 0.2e-2/(50*101325)^0.7 ;  % constante para hallar consumo
mp = rho_c * Sf * k * p_c^0.7 ;  % consumo    [kg/s]

%%% Hallo la presion de salida
Gam = sqrt(gair) * (2 / (gair+1))^((gair+1) / 2 / (gair-1));
                                % constante Gamma(gamma)

ecuacion = @(p_s) As/Ag - Gam /...
    sqrt( 2*gair * ...
         (1 - (p_s/p_c) ^ ((gair-1)/gair) ) /...
         (gair-1)...
         ) /...
     (p_s/p_c) ^ (1/gair);    % ecuacion que relaciona areas

p_s = fsolve(ecuacion, 5e5);       % presion de salida

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% Resistencia aerodinamica: D(v,h)
Mach = @(v,h) v ./ sqrt( gair .* Ra .* T(h) );        % Mach
Cd =   @(v,h) 2.6 .* Mach(v,h) .^(1.1) .* exp(-Mach(v,h))...
              + 0.6 .* sqrt(Mach(v,h));               % coef. resist.

D = @(v,h) rho(h) .* v.^2 .* Sf .* Cd(v,h) ./ 2;      % resistencia aero.

%% Empuje Motor: E(h)
E = @(h) p_c .* Ag .* Gam .*...
    sqrt( 2*gair * ...
         (1 - (p_s/p_c) ^ ((gair-1)/gair) ) /...
         (gair-1)...
         ) + ...
    As .* ( p_s./p_c - p(h)./p_c ) ./ Ag;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% Datos de entrada a ode45

%%% Sistema de ecuaciones
acel  = @(v,h,t) (E(h) - D(v,h) - mp.*v) ./...
     (m0 - mp.*t) - g;                     % con combustible

acel2 = @(v,h,t) -D(v,h)./mcohe - g;       % sin combustible

%%% NOTA: h = x(1), v = x(2);
%%% Sistema: 1. dh/dt = v
%%%          2. dv/dt = f(h,hp)
F  = @(t,x) [ x(2,1); acel(x(2,1) ,x(1,1),t)];
F2 = @(t,x) [ x(2,1); acel2(x(2,1),x(1,1),t)];

%%% tiempo en el que se termina el combustible
tfin1 = mcomb/mp; %[s]

%%% condiciones iniciales
x0(1,1) = 0;      % A nivel del mar [m]
x0(2,1) = 0;      % En reposo       [m/s]
```

```
%% Integrando...
[tcomb,x] = ode45(F,[0 tfin1],x0);          % con combustible,

x0_2 = [x(length(x),1); x(length(x),2)];    % entrada a 2º sistema
tfin2 = 20;                                 % tiempo fin integracion [s]
[tfin,x2] = ode45(F2,[tfin1 tfin2],x0_2);   % sin combustible,


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% Graficos

% h(m)   frente a t(s)
figure(1);
hold on
plot(tcomb,  x(:,1), '-b',...
     tfin, x2(:,1),'-r', 'LineWidth',1)
plot(tcomb(end),x(end,1),'ko','MarkerSize',10)
hold off
xlabel('t [s]', 'FontSize',11);
ylabel('h [m]', 'FontSize',11);


% v(km/s) frente a t(s)
figure(2);
hold on
plot(tcomb,x(:,2)*1e-3, '-b',...
     tfin,x2(:,2)*1e-3,'-r', 'LineWidth',1)
plot(tcomb(end),x(end,2)*1e-3,'ko','MarkerSize',10)
hold off
xlabel('t [s]', 'FontSize',11);
ylabel('v [ km / s ]', 'FontSize',11);

% Mach frente a t(s)
figure(3);
hold on
plot(tcomb,Mach(x(:,2),x(:,1)), '-b',...
     tfin,Mach(x2(:,2),x2(:,1)),'-r', 'LineWidth',1)
hold off
xlabel('t [s]', 'FontSize',11);
ylabel('M', 'FontSize',11);


% a(km/s^2) frente a t(s)
figure(4);
plot(tcomb, acel(x(:,2),x(:,1),tcomb).*1e-3, '-b',...
     tfin,acel2(x2(:,2),x2(:,1).*1e-3,tfin),'-r', 'LineWidth',1)
xlabel('t [s]', 'FontSize',11);
ylabel('a [ km / s^2 ]', 'FontSize',11);
axis([0,tfin(end),-2*g,2*g]);
```

Una de las diferencias históricas entre Matlab y Octave es la integración de ODEs. Mientras Matlab dispone la colección de funciones ode*, siendo ode45 la más comúnmente utilizada, Octave utiliza la función lsode con una convención mucho más parecida a la rutina en fortran a la que llama.

A parte de la llamada, si ode45 se encuentra un valor complejo, como puede ser la raíz cuadrada de un número negativo, integrará el valor absoluto de la función después de imprimir una advertencia por pantalla. En cambio Octave dará un error y cortará la ejecución.

A continuación de lista la versión del mismo programa compatible con Octave en el que se pueden apreciar las diferencias.

```
%%% Programa de simulacion vehiculo con motor cohete a p_c constante
%%% Version para Octave

clear;

%% ISA
g = 9.81;                 % gravedad terrestre [m/s^2]
gair = 1.4;               % gamma aire
Ra = 287;                 % constante MGP aire [J/kg/K]

[T,p,rho] = ISA(288,101325);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Especificaciones Cohete
p_c = 20e6;               % presion camara de combustion  [Pa]
Sf = 0.4;                 % area frontal                  [m^2]
Ag = 0.01;                % area garganta                 [m^2]
As = Sf;                  % area salida                   [m^2]
mcohe = 25;               % masa cohete                   [kg]
mcomb = 30;               % masa inicial combustible      [kg]
m0    = mcohe + mcomb;    % masa inicial total            [kg]
rho_c = 1800;             % densidad combustible solido   [kg/m^3]

%%% Hallo el consumo
k   = 0.2e-2/(50*101325)^0.7 ;  % constante para hallar consumo
mp = rho_c * Sf * k * p_c^0.7 ;  % consumo   [kg/s]

%%% Hallo la presion de salida
Gam = sqrt(gair) * (2 / (gair+1))^((gair+1) / 2 / (gair-1));
                                % constante Gamma(gamma)

ecuacion = @(p_s) As/Ag - Gam /...
    sqrt( 2*gair * ...
         (1 - (p_s/p_c) ^ ((gair-1)/gair) ) /...
         (gair-1)...
         ) /...
    (p_s/p_c) ^ (1/gair);   % ecuacion que relaciona areas

p_s = fsolve(ecuacion, 5e5);      % presion de salida

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% Resistencia aerodinamica: D(v,h)
Mach = @(v,h) v ./ sqrt( gair .* Ra .* T(h) );        % Mach
Cd =   @(v,h) 2.6 .* Mach(v,h) .^(1.1) .* exp(-Mach(v,h))...
              + 0.6 .* sqrt(Mach(v,h));               % coef. resist.

D = @(v,h) rho(h) .* v.^2 .* Sf .* Cd(v,h) ./ 2;      % resistencia aero.

%% Empuje Motor: E(h)
E = @(h) p_c .* Ag .* Gam .*...
    sqrt( 2*gair * ...
         (1 - (p_s/p_c) ^ ((gair-1)/gair) ) /...
         (gair-1)...
         ) + ...
    As .* ( p_s./p_c - p(h)./p_c ) ./ Ag;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% Datos de entrada a lsode
```

```
%%% Sistema de ecuaciones
acel  = @(v,h,t) (E(h) - D(v,h) - mp.*v) ./...
    (m0 - mp.*t) - g;                      % con combustible

acel2 = @(v,h,t) -D(v,h)./mcohe - g;       % sin combustible

%%% NOTA: h = x(1), v = x(2);
%%% Sistema: 1. dh/dt = v
%%%          2. dv/dt = f(h,hp)
F  = @(x,t) [ x(2);  abs(acel( x(2),x(1),t))];
F2 = @(x,t) [ x(2); -abs(acel2(x(2),x(1),t))];

%%% tiempo en el que se termina el combustible
tcomb = linspace(0,mcomb/mp,1000); %[s]

%%% condiciones iniciales
x0(1,1) = 0;     % A nivel del mar [m]
x0(2,1) = 0;     % En reposo       [m/s]

%% Integrando...
x = lsode(F,x0,tcomb);               % con combustible

x0_2 = [x(length(x),1); x(length(x),2)];     % entrada a 2º sistema
tfin = linspace(tcomb(end),20,1000);         % tiempo fin integracion [s]
x2 = lsode(F2,x0_2,tfin);                    % sin combustible

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% Graficos

% h(m)   frente a t(s)
figure(1);
clf;
hold on
plot(tcomb,x(:,1), '-b','LineWidth',2,...
     tfin, x2(:,1),'-r','LineWidth',2)
plot(tcomb(end),x(end,1),'ko','MarkerSize',10)
hold off
xlabel('t [s]', 'FontSize',11);
ylabel('h [m]', 'FontSize',11);
print -dsvg 'h.svg'

% v(km/s) frente a t(s)
figure(2);
clf;
hold on
plot(tcomb,x(:,2)*1e-3,'-b','LineWidth',2,...
     tfin,x2(:,2)*1e-3,'-r','LineWidth',2)
plot(tcomb(end),x(end,2)*1e-3,'ko','MarkerSize',10)
hold off
xlabel('t [s]', 'FontSize',11);
ylabel('v [ km / s ]', 'FontSize',11);
print -dsvg 'v.svg'

% Mach frente a t(s)
figure(3);
clf;
hold on
plot(tcomb,Mach(x(:,2), x(:,1)), '-b','LineWidth',2,...
     tfin ,Mach(x2(:,2),x2(:,1)),'-r','LineWidth',2)
hold off
```

```
xlabel('t [s]', 'FontSize',11);
ylabel('M', 'FontSize',11);
print -dsvg 'M.svg'

% a(km/s^2) frente a t(s)
figure(4);
clf;
plot(tcomb, acel(x(:,2),x(:,1),tcomb')/g,'-b','LineWidth',2,...
     tfin,acel2(x2(:,2),x2(:,1)/g,tfin'),'-r','LineWidth',2)
xlabel('t [s]', 'FontSize',11);
ylabel('a/g', 'FontSize',11);
axis([0,tfin(end),-20,100]);
print -dsvg 'a.svg'
```

En este ejercicio se aplica directamente la filosofía de las dos normas. Todas las funciones, en este caso function handles, están definidas según su expresión teórica sin evaluar ninguna de sus variables.

Esta aproximación a la resolución de problemas es más clara y más potente, de hecho se ha conseguido resolver el problema entero sin definir ninguna función. Si fuera necesario cambiar algún parámetro, como por ejemplo la sección frontal del cohete, bastaría con cambiar su definición y volver a correr el script.

Estos son los resultados:

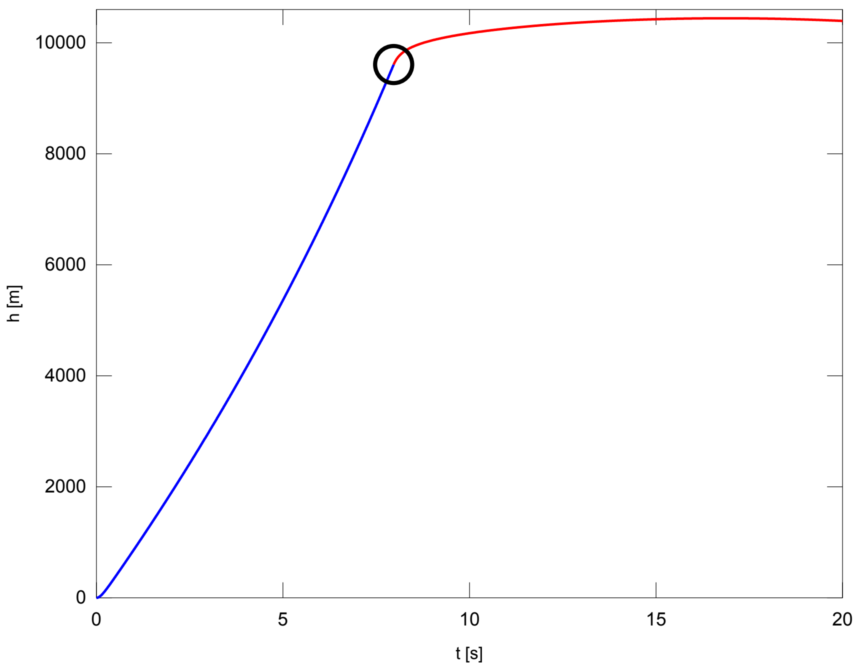

Figura 13.1: Altura en función del tiempo

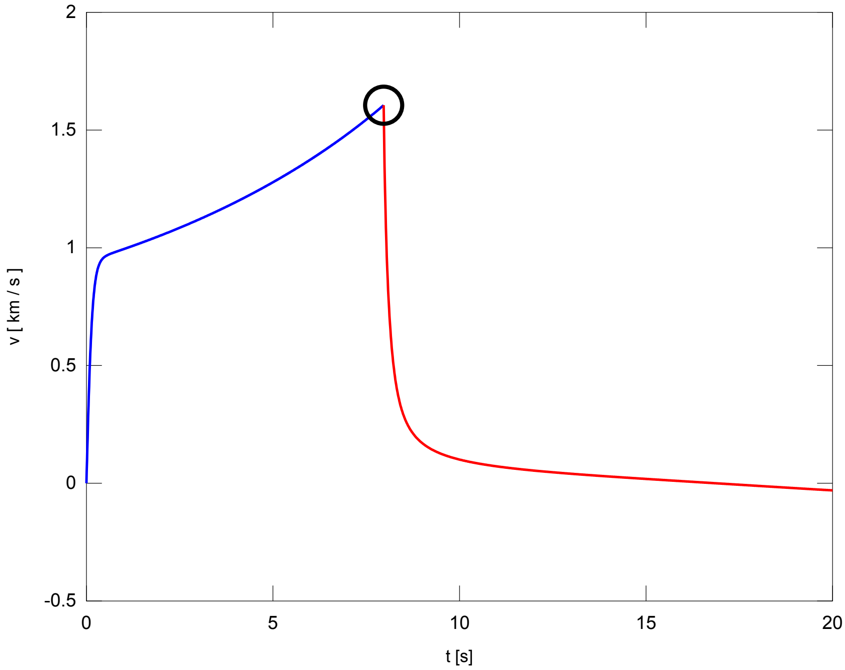

Figura 13.2: Velocidad en función del tiempo

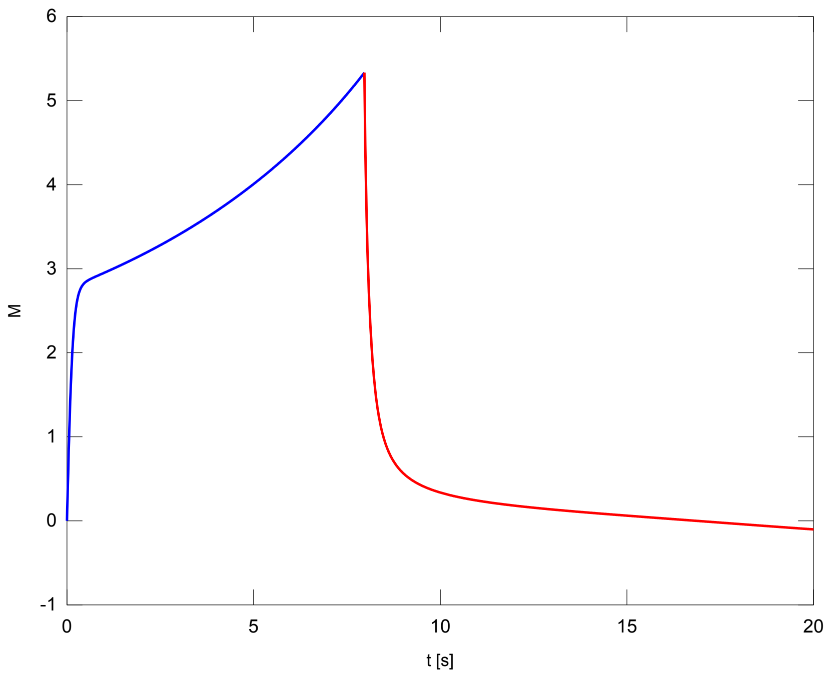

Figura 13.3: Número de Mach en función del tiempo

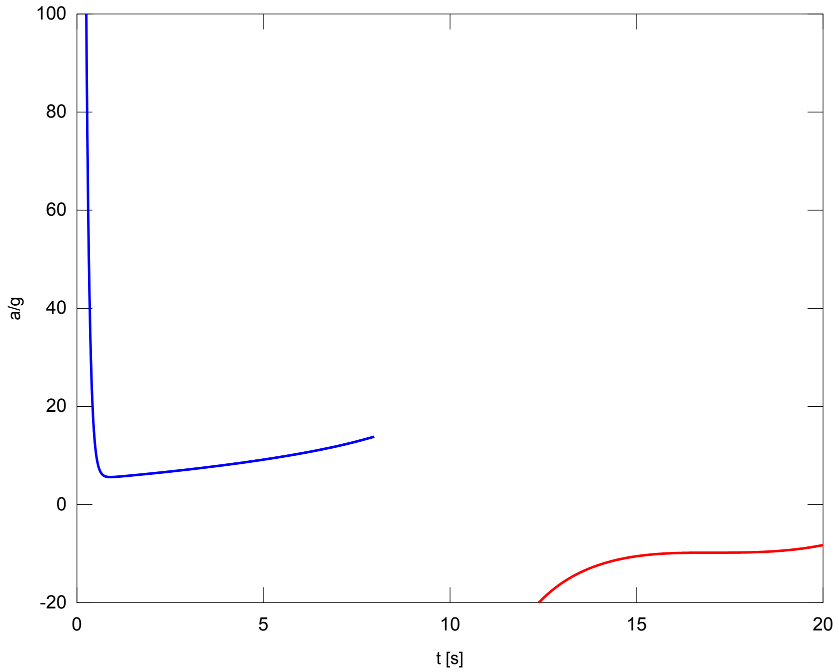

Figura 13.4: Aceleración en función del tiempo

CAPÍTULO 14

La ecuación de Burgers.

Este ejercicio tiene una finalidad puramente didáctica. Veremos algunas de las particularidades de los métodos pseudoespectrales para la resolución de ecuaciones en derivadas parciales en dominios periódicos. El uso de un método espectral, de formulación mucho más complicada, se justifica por ser mucho más rápido. Aunque parezca sorprendente la transformada rápida de Fourier o FFT (por sus siglas en inglés Fast Fourier Transform) es una operación especialmente eficiente en términos computacionales. En este primer ejemplo, en el que se resuelve una ecuación con términos de convección y difusión con un método explícito, no se hacen evidentes las ventajas. Se hacen obvias cuando, resolviendo las ecuaciones de Navier Stokes en dominios bidimensionales o tridimensionales, se usan métodos de integración temporal un poco más sofisticados.

He aquí la ecuación de Burgers

$$\partial_t u + u \cdot \nabla u = \nu \nabla^2 u \tag{14.1}$$

En una dimensión

$$u_t = \nu u_{xx} - u u_x \tag{14.2}$$

La ecuación se esquematiza de la siguiente manera

$$\partial_t u = L(u) - D(u) \tag{14.3}$$

Como el dominio es periódico podemos suponer que la forma de la solución es $u = \hat{u}\exp(ikx)$. que es equivalente a plantear la ecuación para el espectro de la velocidad, su transformada de Fourier.

$$\partial_t \hat{u} = L(\hat{u}) - D(\hat{u}) \tag{14.4}$$

En este caso, la forma del operador lineal es

$$L(\hat{u}) = -k^2 \nu \hat{u} \tag{14.5}$$

Es obvio que sin el término no lineal la resolución de la EDO sería trivial. Bastaría con transformar la condición inicial al espacio de frecuencias e integrar sobre él.

No existe ninguna manera exacta de convertir la operación uu_t en un operador $D(\hat{u})$, este cálculo debe realizarse en el espacio real. Para cada paso temporal será necesario antitransformar la solución, calcular el operador $D(u)$, transformarlo y sumarlo en el espacio de frecuencias.

Nota: Podemos pensar que realizar dos transformadas de Fourier y multiplicar dos vectores es poco rentable. Nada más lejos de la realidad. La FFT necesita del orden de $N \log N$ operaciones. Para que este número sea significativo podemos compararlo con resolver un sistema de ecuaciones lineales que necesita N^2 operaciones o invertir una matriz, que requiere N^3.

La ecuación de Burgers nos reserva un fenómeno no esperado: el aliasing. Cuando un operador diferencial es lineal no se mezcla el espectro de ninguna manera. Por ejemplo, el operador derivada segunda $\partial_{xx}\hat{u} = -k_x^2\hat{u}$ no introduce ninguna operación entre cada uno de las componentes de $\hat{u}$; lo mismo sucede con los filtros proporcionales como el pasa-altos o el pasa-bajos. El operador $D(\hat{u})$ no es lineal e introducirá aliasing en $\hat{u}$.

14.1 Aliasing

Para entender mejor el aliasing como fenómeno podemos tomar una única onda simple para la variable $u = \sin(\phi)$ con, obviamente, $\phi = [0, 2\pi]$. Su derivada en función de la coordenada espacial será $u_\phi = \cos(\phi)$, entonces el término no lineal será $u_\phi u = \cos(\phi)\sin(\phi)$.

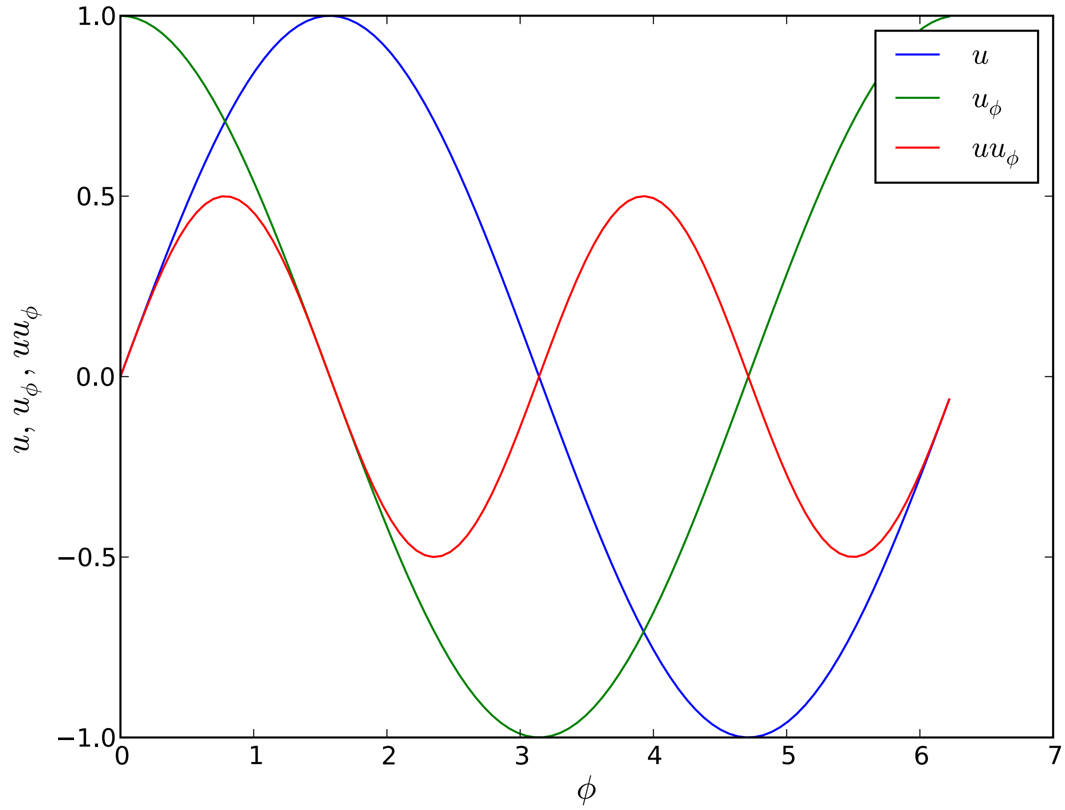

Figura 14.1: Demostración de la creación de armónicos superiores

Como se ve en la figura, el término no lineal genera armónicos superiores hasta el número de onda $2k_N$ que simplemente no se distinguen si no se les da suficiente espacio en el espectro. Que no se distingan no significa que no aparezcan, lo que sucede es que se mezclan con los números de onda superiores, dicha mezcla recibe el nombre en inglés de *aliasing*.

El fenómeno se percibe mucho mejor si representamos el espectro de alguna función un poco más compleja aunque simétrica, para que todos los coeficientes de la transformada sean reales.

CAPÍTULO 15

Algunas inconsistencias

Una vez nombrados todos los elementos propios del lenguaje Matlab, a falta de tratar la práctica de la programación, es el momento de aplicar el sentido crítico al lenguaje y sacarle los defectos.

Desde un punto de vista puramente formal Matlab es un lenguaje que arrastra ciertos malos vicios debido a la compatibilidad hacia atrás. Es curioso comprobar cómo, aunque Mathworks ha mostrado todo el interés posible por añadir al lenguaje las características necesarias para hacerlo más comercial, se han esmerado poco en corregir los errores del pasado.

15.1 Matlab en la enseñanza

Soy un firme detractor del uso de Matlab en cursos de programación en los primeros cursos de una carrera de Ingeniería. Esta es una opinión puramente personal pero está sólidamente fundamentada.

El objetivo primordial de un curso de programación no es ser productivo ni aprender Cálculo Numérico, es aprender a programar. Desgraciadamente existe el empeño generalizado de intentar completar los tres objetivos a la vez. Cuando se cae en ese error se empieza a ver Matlab como una elección adecuada.

El arte o la práctica de programar es el de manipular datos con mediante algoritmos para completar una tarea con un ordenador. Debemos fijarnos que en esta definición no aparecen lenguajes, cálculo numérico, matemáticas... Es algo que uno de los mayores expertos en el tema, Donald Knuth, sabe perfectamente. En su obra enciclopédica *The Art of Computer Programming* utiliza un lenguaje inventado de semántica adecuada para la descripción de algoritmos. Si un lengaje no se le parece es porque el lenguaje está mal.

Pero enseñar a programar sin un lenguaje de programación es terriblemente lento porque obliga a aprender dos veces lo mismo: cómo describir algoritmos y cómo utilizar un lenguaje de programación. Por brevedad la mayoría de cursos de programación se basan en aprender a programar con un lenguaje pero... ¿Qué lenguaje? Si tuviéramos que escoger uno tomaríamos el que tuviera una semántica lo más consistente posible, sin excepciones, para acercarnos a un lenguaje algorítmico. Esta no es, de lejos, la mayor virtud de Matlab. De hecho es su mayor defecto.

Las lenguas más difíciles de aprender son las que, con pequeños cambios en las palabras o la sintaxis, generan resultados completamente distintos. En el chino, por ejemplo, la palabra que para un occidental suena como *ma* puede tener hasta cuatro significados dependiendo de la pronunciación de la vocal. No es muy distinto del hecho de cambiar el comportamiento de un operador como la multiplicación (`*`) con otro (`.*`).

A continuación se listan algunas de las características del lenguaje que deberían ser eliminadas o modificadas para conseguir una mayor consistencia. ¿Es eso posible? Es muy difícil que un lenguaje de programación salga bien a la primera. Fortran ha visto nueve revisiones desde su aparición en los cincuenta. Los desarrolladores de Python se atrevieron a introducir cambios que rompían todo el código existente justo durante la revolución de las aplicaciones web con el único objetivo de llevar su filosofía hasta su última consecuencia. Cambiar los lenguajes es posible y en muchos casos es beneficioso.

Este capítulo no pretende influir en Mathworks sino en el lector. Mitificar cualquier herramienta por fabulosa que pueda parecer es un grave error. Mucho más cuando la popularidad de Matab hace que vea usos totalmente inadecuados como la enseñanza de los fundamentos de la programación.

15.2 El estándar Matlab

Matlab es como Mathworks le conviene que sea. No existe ninguna referencia completa del lenguaje y no hay ningún comité que evalúe las revisiones. Esta característica no es en sí un defecto, la documentación que pudemos encontrar en la ayuda está fabulosamente bien escrita y un manual de referencia no se hace imprescindible.

Pero hay efectos secundarios. Si existiera una referencia, con su correspondiente versión, su historial de cambios y las características obsoletas, el programador de Matlab sabría cómo y cuándo un script quedaría obsoleto y actuaría en consecuencia. Es difícil en la actualidad saber si el código que he escrito en Matlab 2008b ejecutará sin problemas en un Matlab 6.5. Lo único que podemos hacer ahora es probarlo y cruzar los dedos.

Para evitar que una nueva versión rompa todo el código escrito hasta la fecha nunca se declaran características obsoletas mientras siguen añadiéndose nuevas funcionalidades. Matlab está condenado a crecer indefinidamente.

15.3 La verdad sobre la indexación recursiva

Matlab heredó parte de su sintaxis de Fortran, el que fue, es y será el lenguaje del cálculo científico de altas prestaciones por excelencia. Una de las particularidades de Fortran respecto al resto de lenguajes de programación es el tratamiento estricto de las dimensiones de una matriz. Este tratamiento se ha reforzado en las últimas revisiones del lenguaje haciéndolo aún más estricto. Pero hubo una cosa que los creadores de Matlab no entendieron del todo bien: el hecho que en Fortran no existan la indexación múltiple no significa que defina matrices en vez de arrays.

El concepto de matriz es una abstracción matemática mientras que el de array es un concepto computacional. El segundo parte de la base de que en realidad la memoria es plana, esto es, no tiene filas y columnas. De hecho, cuando se habla de las diferencias entre C y Fortran siempre se nombra la manera de ordenar las *matrices*, algo que profundiza en la confusión.

Importante: Como la memoria es plana un array no es más que un vector de vectores de la misma longitud. Un array de *rango* tres es un vector de vectores de vectores. A la vez un vector es un array de escalares.

Si Matlab, como el resto de lenguajes de programación, basara sus cálculos en arrays en vez de en matrices el indexado podría hacer uso de la recursividad.

En C las matrices se declaran precisamente haciendo uso de ese concepto

```
double array[3][4]
```

Esta declaración es totalmente equivalente a

```
double array[12]
```

Python dispone de un módulo extra llamado *numpy* y de manera análoga a las listas también utiliza la recursión para indexar sus elementos

```
>>> from numpy import array
>>> a = array([[1,2,3],[4,5,6],[7,8,9]])
>>> a[0]
array([1, 2, 3])
>>> a[0][0]
1
>>> a[0,0]
1
```

A riesgo de parecer reiterativo, definiendo un array como una recursión de vectores se consigue tanto un sistema para definir arrays independientemente de las dimensiones como dos maneras alternativas para indexarlos.

Aquí llega una diferencia idiomática entre Matlab y Octave, precisamente una de la que Mathworks debería tomar nota. Octave tiene soporte *limitado* para el indexado recursivo. Supongamos que tenemos una función que devuelve una matriz

```
function y = foo(x)

  y = [x,2*x,3*x];
```

Sabiendo lo anterior podemos indexar el resultado junto con la llamada

```
octave>> z = foo(2)(2)
z =  4
```

Esto sucede con cualquier llamada que devuelva una matriz como resultado

```
octave>> y = {[1,2,3],[4,5,6]};
octave>> y{1}(2)
ans =  2
```

Desgraciadamente no se lleva este concepto hasta la última consecuencia para no romper de manera exagerada con la compatibilidad.

```
octave>> a = [1,2,3;4,5,6;7,8,9];
octave>> a(1)
ans =  1
octave>> a(3)
ans =  7
```

Operación que, por cierto, no tiene ninguna lógica y es otra de las raras excepciones del lenguaje.

15.3.1 ¿Qué es una matriz?

Acabamos de ver que la indexación en Matlab es algo confusa y poco versátil. El problema de las inconsistencias es que afloran por doquier en los sitios más insospechados. El último ejemplo del resultado anterior es muy significativo. Como la indexación múltiple no existe en Matlab al indexar una matriz con menos subíndices de lo previsto provoca un resultado imprevisto: no ha aparecido ningún error.

Siguiendo con el último ejemplo, `a` es un array de *rango* dos, al darle sólo un subíndice obtengo un resultado de *rango* cero. Ahora forcemos más la sintaxis

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

¡No existe ninguna relación entre el número de subíndices y el *rango* del resultado! ¡He intentado indexar incorrectamente una matriz y no ha dado ningún error! Si ahora sumamos a que, como lenguaje dinámico, Matlab no comprueba los tipos en cada asiganción el peligro de cometer un error se multiplica.

Otra vez somos víctimas del concepto difuso y poco estricto de matriz en Matlab.

Ahora veamos como un lenguaje dinámico y consistente lidia con el problema del rango y el indexado.

```
>>> from numpy import array
>>> a = array([[1,2,3],[4,5,6],[7,8,9]])
>>> a[0]
array([1, 2, 3])
>>> a[0,0]
1
>>> a[0,0,0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: invalid index
```

15.3.2 La innecesaria distinción entre filas y columnas

A diferencia de los arrays, las matrices tienen filas y columnas, distinción importante para las operaciones matriciales. Por ejemplo, un producto escalar será la multiplicación matricial entre un vector fila y otro columna. Con arrays simplemente es el la suma del producto de cada elemento. No es necesario hacer ninguna distinción.

Siendo estrictos no es necesario distinguir filas de columnas desde un punto de vista algebraico, sólo hay que fijarse si la operación aumenta, mantiene constante o disminuye el rango. Podemos definir un producto escalar o un producto externo sin el concepto de matriz, lo necesitamos si queremos unir ambos en una multiplicación matricial.

Entonces el problema de los vectores fila y columna viene de la propia naturaleza de la multiplicación matricial. El empeño de reducir una colección bastante extensa de operaciones con matrices y vectores a la multiplicación para reducir la cantidad de operadores o funciones termina siendo una complicación añadida e inútil. No es lo mismo una multiplicación matricial que un producto escalar, la multiplicación conserva el rango y el producto escalar lo reduce. El hecho que el algoritmo de la multiplicación sea hacer productos escalares no justifica que ambas operaciones tengan el mismo operador. También la multiplicación es una sucesión de sumas y a nadie se le ocurriría denotarlas con el mismo símbolo.

Como la multiplicación, el producto externo y el producto escalar tienen el mismo operador es necesario distinguir entre filas y columnas obligando a distinguir también entre dos tipos de vectores; algo que va en contra de cualquier sentido estético y genera multitud de errores. En mis clases de Matlab digo a mis alumnos que ignoren los vectores, que no existen, que en realidad sólo hay matrices que tienen una columna o una fila. Entonces las secuencias, que en realidad son vectores fila, rompen toda la dialéctica.

La multiplicación en Matlab es un caso claro de decisión poco meditada en el diseño de un lenguaje de programación. Un intento de reducir la información al unir tres operaciones provoca un aumento de la complejidad al tener que aprender multitud de casos particulares.

Nota: Matlab dispone de las funciones necesarias para no caer en la confusión anterior, `dot` es el producto escalar y `kron` sirve para calcular el producto exterior. Aunque estas funciones pueden ahorrarnos multitud de errores su uso es completamente marginal puesto que en todas las guías de programación en Matlab se hace incidencia sobre la "fabulosa" brevedad del operador `*`.

---

dot (*a*, *b*)
Calcula el producto escalar de los dos vectores *a* y *b*

Parámetros

- **a** – Vector fila o columna
- **b** – Vector fila o columna

Return type Escalar

kron (*a*, *b*)
Producto tensorial de Kronecker de dos tensores. Cuando *a* y *b* son vectores esta operación se llama producto exterior.

Parámetros

- **a** – Escalar, vector o matriz de cualquier dimensión
- **b** – Escalar, vector o matriz de cualquier dimensión

Return type La dimensión del resultado siempre será la suma de las dimensiones de los argumentos

15.4 ¿Qué es una celda?

Fijémonos en la cabecera de la definición de una función

```
function [x,y,z] = foo(p,q,r)
```

Si analizamos sintácticamente la frase tenemos una sentencia como `function` que anuncia la declaración de una función, posteriormente viene una matriz que contiene tres variables, el operador asignación y finalmente el enunciado de cabecera de función.

Fijémonos ahora en el elemento `[x,y,z]`, rigurosamente hablando es una matriz que contiene tres variables pero en realidad es una asignación triple. Esto suele llamarse un triple o un tuple de tres elementos y es un tipo presente en muchos lenguajes de programación dinámicos. ¿Entonces en caso de la asignación múltiple los corchetes designan matrices o tuples? Vamos a comprobarlo

```
>> [x,y,z] = [1,2,3]
??? Too many output arguments.
```

Pues ahora que lo de la izquierda es un tuple y lo de la derecha es una matriz. Parece que llegamos a una conclusión, cuando algo delimitado por corchetes está al lado izquierdo de una asignación es un tuple y si está en el lado derecho es una matriz. Hasta que definimos la función `foo`

```
function [x,y,z] = foo(p,q,r)
  x = p;
  y = q;
  z = r;
```

Y probamos lo siguiente

```
>> x = foo(1,2,3)
x =
     1
```

¿Entonces qué retorna una función? La cabecera establece claramente una asignación triple pero al encontrar sólo un argumento de salida lo convierte en una asignación simple e ignora los otros argumentos. Entonces la cabecera no sirve para nada y establece una jerarquía de argumentos según su orden sin ningún control estricto sobre la cantidad.

El defecto subyacente es que Matlab no tiene un operador asignación completamente consistente que establece la excepción de las llamadas a funciones. Este defecto podría solucionarse si los *cell arrays* se comportaran como tuples y soportaran la asignación múltiple. Uniendo lo anterior a que las funciones tuvieran a celdas como argumentos de salida podríamos arreglar ese defecto en el operador asignación. Este sería un ejemplo del declaración.

```
function {x,y,z} = foo(p,q,r)
  x = p;
  y = q;
  z = r;
```

Y este su funcionamiento

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

>> {x,y,z} = foo(1,2,3)
x =
     1

y =
     2

z =
     3

>> x = foo(1,2,3)
??? Not enough output arguments.

>> x = foo(1,2,3){1}
```

```
x =
     1
```

¿Qué son entonces los cell arrays? Paraecen una manera un poco más sofisticada de ordenar valores pero es difícil encontrar el por qué de su existencia. Permitiendo la asignación múltiple con cell arrays y la indexación múltiple se podría dotar al operador asignación de un significado verdadero.

15.5 Funciones y archivos

Las limitaciones de una única función por archivo y de no poder definir funciones dentro de la sesión del intérprete es sumamente ridícula. No consigo entender cómo ha llegado Matlab a esas cotas de popularidad con semejante inconveniente. Sin su posición dominante en el mercado sería imposible que se introdujera en él puesto que la calidad del intérprete y de el ecosistema de cálculo es infinitamente inferior al de la competencia.

Es, además, una limitación tecnológicamente inaceptable puesto que incluso el Octave lo soporta como extensión al lenguaje. En muchos casos hay que hacer encaje de bolillos para no terminar con el programa partido en decenas de archivos.

15.5.1 ¿Cuál es el paradigma de Matlab?

Los lenguajes de programación soportan uno o varios paradigmas. Lisp sigue la programación funcional, C es un lenguaje procedimental y modular, Java es un lenguaje estático orientado a objetos, Python soporta mejor o peor todos los paradigmas conocidos. El paradigma de Matlab es Matlab.

Es difícil hacer programación procedimental en Matlab porque cada función debe estar en un archivo, esto impide juntar todo lo que es esencial en el script principal si una función tiene que estar en él.

Hacer programación modular en Matlab tiene más que ver con el talento en el uso del lenguaje que con las facilidades de las que disponemos. Uno puede, gracias a una función y unos function handles, acercarse al paradigma modular, pero no será más que un sucedáneo. Los módulos, por definición, son estructuras de funciones y parámetros de las que uno puede tomar lo que le apetezca mediante un mecanismo de *import*. C dispone de las cabeceras que no son más que archivos donde se lista el contenido de una librería; es mas fácil llegar al paradigma modular mediante este planteamiento que con Matlab.

Matlab fuerza a programar de una determinada manera, con un estilo muy concreto, a base del uso indiscriminado de los function handle que es lo único que lo convierte en un lenguaje verdaderamente dinámico. Sin esta estructura Matlab puede compilarse simplemente anotando las cabeceras de las funciones. Si bien esta característica habla bien del compilador de Matlab (bastante caro, por cierto) habla terriblemente mal de sus bondades como lenguaje de programación.

Nota: Existe toda una rama de desarrollo en los lenguajes de programación dinámicos y su compilación a estructuras estáticas para aumentar su rendimiento. Una de ellas es la identificación de estructuras para anotarlas, asignar tipos a todas las variables y pasarlas a ensamblador. Este es el esquema de funcionamiento de un compilador JIT (Just In Time). Otro aspecto es el de descubrir en tiempo de compilación los tipos de cada variable en vez de dejar que el intérprete lo descubra en tiempo de ejecución. A este proceso se le llama dynamic typing y es terriblemente complejo en algunos lenguajes de programación porque es imposible generar estructuras estáticas a partir de cualquier estructura dinámica.

15.5.2 La orientación a objetos

La orientación a objetos en Matlab es una de estas estrategias de marketing sin demasiado sentido. Con la popularidad de Java y de C++ llegó un momento en el que tu lenguaje era orientado a objetos o los programadores lo despreciaban sin complejos. Como Matlab es un producto comercial y su objetivo es vender terminaron añadiendo OO al lenguaje. Pero fue un absoluto desastre. Tanto que terminaron cambiándola completamente porque su primer intento era simplemente imposible de utilizar. De este modo Mathworks introdujo el primer gran cambio que rompía la compatibildad con versiones anteriores.

Pero este no es el único problema. ¿Tiene sentido un lenguaje orientado a objetos con una biblioteca en la que no hay ni una sola clase? Ahí no terminan los inconvenientes. Siempre se criticó a PHP salvajemente por no soportar namespaces, Matlab no tiene y parece que a nadie le molesta.

En los lenguajes modernos *todo* es un objeto. Cualquier tipo tiene métodos asociados y puede derivarse para generar descendencia. No se termina la historia permitiendo definir clases, con la herencia y el polimorfismo. Esto significa que Matlab tampoco es una buena opción para introducir a nadie en la OO.

Por lo menos alguien entró en razón y se esforzaron en una implementación del paradigma razonable, infinitamente mejor que la primera iteración.

15.6 El punto de la muerte

Ya hemos hablado del poco apropiado concepto de la multiplicación matricial. Hemos olvidado de forma premeditada otra posibilidad para la multiplicación, la operación de producto elemento a elemento cuando los dos operandos tienen el mismo tamaño.

Incomprensiblemente y haciendo gala de una tremenda falta de ingenio estos dos operadores críticos se diferencian únicamente por un punto. Lo mismo sucede con la división y la potencia. Esto es la fuente del 90 % de los errores de programación en Matlab y es algo tan obvio que no entiendo cómo no se les pasó por la cabeza. Desgraciadamente es algo tan arraigado a Matlab que dudo que cambie nunca. La alternativa es utilizar otro lenguaje.

15.7 El punto y coma absurdo

Que el comportamiento por omisión de un comando sea mostrar el resultado en la salida estándar es otra de estas convenciones tan arraigadas como inútiles. Es el único lenguaje de programación cuya salida no viene condicionada por una función o un comando, simplemente sucede. Es mucho más común olvidarse de poner el punto y coma que ponerlo sin querer.

Lo peor del asunto es que la solución no requiere demasiados cambios ni demasiadas discusiones. Bastaría con sacar una nota antes de Matlab 8.0 diciendo que a partir de esta versión el comportamiento por omisión es no imprimir el resultado. ¿Por qué existe entonces la función `disp`?

15.8 Funciones y sentencias o cómo ahorrarse paréntesis

Hay una distinción fundamental entre funciones y sentencias. Una sentencia, como lo es `for` o `if` son partes del lenguaje y se encargan del control de flujo de ejecución. Una función encapsula una tarea, no es más que un bloque de ejecución.

En algunos casos, cuando una función es muy habitual y penalizando la consistencia, se convierte en una sentencia pero sigue siendo una función porque se llama con argumentos.

En Matlab existen dos casos paradigmáticos que rompen totalmente con la sintaxis con la única finalidad de ahorrarse un par de paréntesis y un par de comillas como `hold` y `print`.

Fijémonos en el uso de la sentencia `hold`. Si quiere activarse el redibujo se escribe

```
hold on
```

Para alguien que no hubiera visto nunca Matlab esto es una sentencia que recibe una variable como argumento. Ni *hold* es una sentencia ni *on* es una variable.

`print` es de patología más severa. Aunque puede llamarse como una función está diseñado para ser llamado como un comando de consola al estilo UNIX. Las opciones se pasan con el prefijo `-`, algo que parecerá habitual a los usuarios de los sistemas operativos serios. Lo más grave es que, cuando *print* se llama como función los argumentos también deben utilizar el mismo prefijo.

Importante: Uno de los muchos motivos de la transición entre Python 2 y Python 3 fue precisamente la sentencia `print`. Imprimir en la consola es una función tan utilizada que en algunos lenguajes tiene el estatus especial de sentencia, *pero es una excepción*. Si uno de tus objetivos es buscar la máxima consistencia debes cumplirlo eliminando las excepciones de tu lenguaje. En Python 3 `print` es una función y debe llamarse con argumentos. Este cambio aparentemente nimio significa romper prácticamente todo el código escrito hasta la época, aunque en este caso portarlo sea trivial. Aunque sean menos evidentes los otros cambios causaron peores dolores de cabeza.

El motivo de la conversión de funciones en sentencias es obtener mayor brevedad al escribir una frase muy habitual. Una llamada consistente sería

```
hold(true)
```

CAPÍTULO 16

Índices y Tablas

- *genindex*
- *modindex*

Bibliografía

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

[MEH] Mehta RD. Aerodynamics of sports balls. Annual Review of Fluid Mechanics 1985; 17: 151–189.

[ME2] Rabindra Mehta, Firoz Alam and Aleksandar Subic. Review of tennis ball aerodynamics. Sports Technology, Volume 1, Issue 1 (p 7-16)

[INV] Inverted Pendulum. Wikipedia. http://en.wikipedia.org/wiki/Inverted_pendulum .

[KNU] Donald E. Knuth. The Art Of Computer Programming. http://www-cs-faculty.stanford.edu/~knuth/taocp.html

[OLI] Travis Oliphant. Guide to Numpy. http://scipy.org

[SCH] 16. Naughton, H. Scildt. Java, Manual de Referencia. McGraw Hill (2000)

Índice

V

X

Y

Z