Facultad de Ciencias
Astronómicas
y Geofísicas
UNIVERSIDAD NACIONAL DE LA PLATA

Apunte de Cátedra

Análisis Numérico y Programación

Fabio I. Zyserman y Felipe C. Wachlin

2025

Índice general

Capítulo 1

Programación: elementos de Python

1.1. Presentación

Python es un lenguaje de programación multipropósito. Como tal, ha sido adoptado por gran número de miembros de las comunidades universitarias de todo el mundo; en particular esto es así en geociencias, astronomía, física y análisis numérico, áreas en donde han sido desarrolladas gran cantidad de aplicaciones. Podemos mencionar algunas de las características del lenguaje que nos parecen relevantes:

- Los códigos escritos en Python suelen ser fácilmente entendibles, aún cuando no fueron escritos por uno mismo. Esto es así porque este lenguaje ha sido diseñado para ser legible, reusable y fácilmente mantenible.
- Permite escribir códigos más compactos que otros lenguajes (C, C++, por ejemplo). Dado que se escribe menos, hay menos para controlar, y se ahorra tiempo y esfuerzo. Por otra parte, los códigos Python pueden ser ejecutados inmediatamente, sin necesidad de compilación y linkeo, procesos necesarios en otros lenguajes.
- Python es portable. Es decir, código python puede ser ejecutado en Linux o Windows, y en distintas computadoras sin inconvenientes.[1]
- Existen una gran variedad de recursos de libre acceso para leer sobre Python, escribir y entender códigos con diferentes niveles de dificultad. En la página de la Cátedra pueden encontrar algunos.
- Programadores avezados pueden explotar su facilidad para comunicarse con códigos escritos en otros lenguajes.

Por otro lado, es generalmente aseverado que una desventaja de Python frente a lenguajes compilados utilizados en nuestro ámbito (C, C++, Fortran) es que su velocidad de ejecución es menor. Aunque esto sea así en ciertas situaciones, no son relevantes a lo largo de esta materia, de hecho, tampoco lo van a ser a lo largo de la carrera. De modo que podemos poner manos a la obra, y comenzar con los primeros conceptos del lenguaje.

1.2. Cómo Python ejecuta programas

En su forma más sencilla, un programa Phyton es un archivo de texto que contiene instrucciones Python. Por ejemplo, el archivo -al que llamaremos *prog01.py*- conteniendo las siguientes dos

[1] Debe notarse aquí que Python evoluciona en el tiempo y que las versiones más nuevas no son necesariamente compatibles con las anteriores. En particular, nosotros trabajaremos con la versión 3.9.x

líneas es un programa Python completo:

```
print('Bienvenidos a ANyP!')
print(2**100)
```

Este archivo, también llamado *código fuente*, contiene dos instrucciones `print`, que simplemente imprimen en pantalla una cadena de caracteres (el texto encerrado entre las comillas simples) por un lado y un cómputo numérico por otro. Puede ser generado en cualquier editor de textos; por convención un programa Python siempre tiene un nombre que termina con el sufijo *.py*. Una vez que el archivo está disponible debemos decirle a Python que lo "ejecute", lo que simplemente significa llevar a cabo todas las instrucciones que se encuentran en el mismo de arriba hacia abajo, una después de otra. Existen distintas posibilidades para ejecutar este programa (invocando python desde una terminal, o utilizando un entorno de desarrollo integrado instalado localmente o en línea, o utilizando un compilador en línea en un navegador); por ahora nos interesa saber que si todo anda bien, en cualquiera de los entornos disponibles, debiera aparecer en pantalla lo siguiente:

```
Bienvenidos a ANyP!
1267650600228229401496703205376
```

Ahora, ¿qué es lo que sucede cuando le pedimos a la computadora que corra nuestro programa? Para enterarnos, debemos saber cómo Python procesa nuestras instrucciones. Cuando ordenamos la ejecución nuestro programa, hay algunos pasos que Python sigue antes de que el código realmente comience a procesarse. Específicamente, primero se compila (traduce) a algo llamado "código de bytes" y luego es enviado a algo llamado "máquina virtual".

1.2.1. Código de bytes

Internamente, y en forma semioculta, para ejecutar un programa Python primero compila el código fuente, transformándolo al formato código de bytes. La compilación es simplemente un paso de traducción y el código de bytes es una reepresentación de bajo nivel, independiente de la plataforma (es decir de la computadora y sistema operativo donde se produce), del código fuente. Este proceso se realiza para acelerar la ejecución: el código de bytes se puede ejecutar mucho más rápido que el código fuente original. Dijimos que la compilación se realiza en forma semioculta puesto que si el proceso Python puede escribir en el disco rígido de la computadora, almacenará el código de bytes en archivos que terminan con una extensión *.pyc*, en las misma carpeta o directorio del código fuente.

Python guarda un código de bytes para acelerar la ejecución de un programa. En sucesivas corridas, Python carga los archivos .pyc y omite el paso de compilación, siempre que el código fuente no haya cambiado desde la última vez que se guardó el código de bytes. En caso contrario, la se recompila el código fuente en forma automática. Si los archivos de código de bytes no pueden ser guardados, el código de bytes se genera en la memoria y simplemente se descarta al salir del programa. Podemos comentar finalmente que un archivo de código de bytes permite la ejecución de un programa Python aún en ausencia del correspondiente código fuente.

1.2.2. Máquina Virtual de Python

Una vez que el código de bytes está disponible, es enviado a la llamada Máquina Virtual de Python (MVP) para ser ejecutado. Esta MVP tiene un nombre ampuloso, pero en realidad no es un programa separado ni necesita ser instalado en forma separada. Es simplemente el dispositivo de ejecución de código de Python, el que realmente corre nuestros programas, y

como tal, estará instalado en la computadora en la que compilamos nuestro código fuente.

1.3. Elementos básicos del lenguaje Python

En Python, los datos toman forma de *objetos*, que pueden ser internos (o intrínsecos), o provistos por herramientas externas. Todo programa crea y procesa objetos. En la siguiente tabla 1.1 podemos ver un listado parcial de los objetos provistos por Python, y ejemplos de los mismos.

Objeto	Obj. python	Ejemplo
Números	`int` (Enteros)	`123,1_123`
	`float` (Reales)	`123.,1.23e2,1_345_456.`
	`complex` (Complejos)	`3+4j`
Lógicos	`bool`	`True, False, 4 < 5`
Cadenas de caracteres	`str`	`ÁNyP', "Geofísica"`
Listas	`list`	`[1,[2,tres], 4]`
Diccionarios	`dict`	`{'tipo':'Marga','den':2480,'vp':2100}`
Tuplas	`tuple`	`(1,[2,'tres'], Ónda',1+6j)`
Archivos	`_io.TextIOWrapper`	`open('vel.dat','r')`

Tabla 1.1: Objetos intrínsecos en Python

Si escribimos `1+1` en Python, obtendremos `2`. Es cierto que tendremos el resultado, pero no podemos utilizarlo en cómputos posteriores. De hecho, la computadora lo olvida inmediatamente después de calcularlo. Si queremos mantener un número en la memoria de la computadora para referencia futura, debemos pedirle que le asigne un lugar de almacenamiento en la memoria. Estos lugares de almacenamiento se denominan *variables*. Guardamos números (y otros objetos) en variables. Podemos darles cualquier nombre, con la restricción de utilizar el guión bajo (_), las letras del abecedario, los números del 0 al 9; debemos tener cuidado en que el primer lugar esté ocupado por uno de los dos primeros símbolos mencionados. Así, `_densidad, nombre_de_roca, xy23` son nombres permitidos, pero `1_a, 2pi` no los son. Es importante tener en cuenta que **Python diferencia mayúsculas de minúsculas**; la variable `x` es considerada distinta de la variable `X`.
Existen en Python un listado de palabras que no pueden ser utilizadas como nombres de variables, pues están reservadas para uso específico del lenguaje; su uso genera errores de sintaxis. Las enumeramos en el Apéndice A.
Para que una variable tome un valor determinado, debemos utilizar una instrucción de asignación, que mostramos a continuación.

1.4. Instrucciones de asignación

Su forma general es:

```
nombre_de_variable = expresión
```

Mediante este formato, que no es una igualdad en el sentido usual, el valor calculado en `expresión` es **asignado** a la variable `nombre_de_variable`. Estas instrucciones, como todas las demás en Python, deben ser escritas al comienzo de cada línea sin dejar espacios en blanco, de lo contrario recibiremos un mensaje de error.

```
...
i = 1
```

```
i = i + 1
...
```

Como resultado de esta asignación, la variable `i` toma el valor 2 luego de ejecutadas ambas instrucciones. Debe notarse que estas asignaciones funcionan perfectamente, aún cuando en ninguna instancia previa le hayamos dicho Python que use `i` como una variable, o que ella debiera representar un valor entero. De hecho, en este lenguaje, no debemos preocuparnos por ello; los tipos de objetos son determinados automáticamente al momento de correr el código. Esto diferencia a Python de otros lenguajes, en los que cada variable debe ser declarada de antemano, explicitando su tipo. El esquema de funcionamiento es entonces

- Una variable es creada cuando el código le asigna un valor. Cambios posteriores modifican el valor de la variable ya creada.
- Cuando una variable aparece en una expresión, es inmediatamente reemplazada con el objeto al que en ese momento refiere, cualesquiera sea. Todas las variables deben ser asignadas explícitamente antes que puedan ser usadas; utilizar una variable no asignada produce errores en la corrida.
- Profundizando un poco en la estructura del lenguaje, podemos mencionar que una variable no tiene ningún tipo de información o restricción asociada a ella. La noción de tipo de objeto vive con el objeto mismo. Las variables son siempre genéricas, refieren siempre simplemente a un objeto particular en un momento particular.

Como comentario final de esta sección, mencionamos que Python permite escribir más de una instrucción de asignación por línea de código, separándolas con un punto y coma `;`. Sin embargo, esto no es recomendable, puesto que atenta contra la legibilidad del código.

1.5. Operando con objetos

Daremos ahora precisiones sobre cómo es posible operar con los objetos listados en la tabla 1.1.

1.5.1. Expresiones aritméticas

Como estamos interesados en particular en trabajar con números, sean estos enteros, reales o complejos, analizaremos en primer lugar a las expresiones aritméticas, que son aquellas que combinan operadores aritméticos (ver tabla 1.2) con operandos numéricos enteros, reales o complejos, por ejemplo:

```
a = c*b + d**2 + e*f
```

Acá el resultado de la expresión aritmética es asignado a la variable `a`. En principio, la expresión anterior no tiene nada de extraordinario, salvo que...

...en el campo numérico, el orden en el que se realizan las operaciones, en general, afecta al resultado.

Por eso, necesitamos aprender cómo evalúa Python las expresiones aritméticas (qué hace primero y qué después). Existen en Python los llamdos operadores de asignación, que listamos en la tabla 1.3.

+	Suma	x+y
-	Resta	x-y
*	Multiplicación	x*y
/	División	x/y
//	División entera	x//y
%	Resto de la división	x%y
**	Potencia	x**y

Tabla 1.2: Operadores aritméticos en Python

Operador	Uso	Equivalente a
=	x=y	x=y
+=	x+=y	x=x+y
-=	x-=y	x=x-y
*=	x*=y	x=x*y
/=	x/=y	x=x/y
//=	x//=y	x=x//y
%=	x%=y	x=x%y
**=	x**=y	x=x**y

Tabla 1.3: Operadores de asignación

1.5.2. Orden de precedencia para expresiones aritméticas

Los operadores aritméticos se ordenan de la siguiente manera, según su precedencia:

1. ** (de derecha a izquierda)
2. * / //% (de izquierda a derecha)
3. + - (de izquierda a derecha)

la aclaración entre paréntesis corresponde a "reglas de asociatividad"que se siguen para evaluar una expresión en caso de tener operadores de igual precedencia.

NOTA 1: la importancia de estas reglas se puede ver con el siguiente ejemplo. Consideremos la expresión `2**2**3`. Notar que no es lo mismo hacer

```
(2**2)**3 = 4**3 = 64
```

que

```
2**(2**3) = 2**8 = 256.
```

Para Python, por las reglas de asociatividad, el resultado de la expresión sin paréntesis será 256.

NOTA 2: el paréntesis tiene la máxima precedencia así que, ante la duda, usarlo (sin abusar).

Orden que sigue Python:

1. las expresiones se evalúan de IZQUIERDA a DERECHA,
2. si se encuentra un OPERADOR se lo compara con el siguiente
   a) si el siguiente tiene MENOR precedencia → evalúa el operador previo (`3*5-4`),

b) si tiene IGUAL precedencia → aplica reglas de asociatividad (`3*8*6`),

c) si tiene MAYOR precedencia → busca el siguiente operador (`4+5*7**3*4`).

Ejemplo: identificar el orden de evaluación de la expresión `Z-(A+B/2.)+W*Y`

$$\underbrace{\texttt{z} - \underbrace{(\texttt{a} + \underbrace{\texttt{b/2.}}_{1})}_{2} + \underbrace{\texttt{w} * \texttt{y}}_{4}}_{3}\Bigg)_{5} \tag{1.1}$$

En expresiones numéricas de tipo mixto, es decir, que contengan constantes y/o variables de distinto tipo, Python primero convierte operandos hasta el tipo del operando más complicado y luego realiza los cálculos en operandos del mismo tipo. Python clasifica la complejidad de los tipos numéricos de la siguiente manera: los números enteros son más simples que los números reales, que son más simples que los números complejos. Entonces, cuando un número entero se mezcla con un real, el número entero se convierte primero a un valor real, la o las operaciones se realizan sobre reales, y el resultado es real. De manera similar, cualquier expresión de tipo mixto donde un operando es un número complejo da como resultado que el otro operando se convierta en un número complejo, y la expresión produce un resultado complejo.

1.5.3. Operadores relacionales y lógicos

Podemos ver estos operadores listados en la tabla 1.4. Los operadores relacionales, que testean una condición, actúan sobre números reales y enteros, formando expresiones lógicas. Los operadores lógicos actúan sobre estas últimas y sobre variables booleanas. El resultado evalúa a `True` ó `False`; si es asignado a una variable, ésta es booleana.

Nombre	Op. relacional	Uso	Nombre	Op. lógico	Uso
Igual	`==`	`x==y`	Conjunción	`and`	`x==y and y==z`
Distinto	`!=`	`x!=y`	Disyunción	`or`	`x<y or y==z`
Mayor, Mayor ó igual	`>,>=`	`x>=y`	Negación	`not`	`not (v>w)`
Menor, Menor ó igual	`<,<=`	`x<=y`			

Tabla 1.4: Operadores relacionales y lógicos

Debe notarse que es posible hacer la comparación entre un entero y un real. En ese caso, Python usa la misma convención que vimos, promueve el tipo más simple al más complejo. También es posible concatenar cualquier número de comparaciones; por ejemplo `A<B<C` es equivalente a `A<B and B<C`. Ciertamente, al utilizar esta propiedad del lenguaje debe tenerse cuidado en respetar la lógica de la expresión que desea evaluarse.

1.5.4. Constantes y funciones matemáticas intrínsecas

Ciertamente, para realizar cálculos científicos solemos utilizar distintas funciones y constantes matemáticas. Afortunadamente, las constantes y funciones matemáticas más utilizadas están implementadas en Python. Forman parte de bibliotecas especializadas (archivos que contienen código python) llamadas *módulos*. Hay muchos módulos de Python diferentes. Las funciones matemáticas se pueden encontrar en el módulo `math` así como también en los módulos `numpy` y `scipy`, mientras que para graficar podemos utilizar el módulo `matplotlib`. Por el momento nos

centraremos en el módulo `math`, de los otros nos ocuparemos más adelante.
Como con cualquier otro módulo, para poder hacer uso de los elementos del módulo `math` en nuestro programa, debemos *importarlo*. Esto se hace incluyendo **una** de las siguientes instrucciones en nuestro código:

```
#[2] Se utiliza sólo una de las siguientes opciones

import math              # Opción (a)
import math as m         # Opción (b)
from math import *       # Opción (c)
from math import cos,pi  # Opción (d)
```

Supongamos que necesitamos calcular el valor de $\cos(\pi/5)$. Si utilizamos la opción (a), debemos escribir en nuestro código `math.cos(math.pi/5)` para realizar el cómputo deseado. Podemos evitar usar el nombre completo del módulo si usamos la opción (b). Esto nos permite usar el alias `m` en vez de `math`, de modo que la instrucción para realizar el cómputo es entonces `m.cos(m.pi/5)`. La opción (c) importa todas las funciones y constantes definidas en el módulo. Esto nos permite invocar cualquiera de ellas sin hacer uso del nombre del módulo, de modo que en este caso, la instrucción a escribir es simplemente `cos(pi/5)`. Finalmente, la opción (d) es la que menos memoria utiliza, ya que sólo importa la función coseno y la constante π; con ellas podemos ejecutar la instrucción `cos(pi/5)`.
Aunque la tercera de las opciones es la más cómoda, su uso no es recomendado. Si importamos distintos módulos de esta forma, como todas las funciones y constantes de los mismos son cargados en memoria, si una misma constante tiene distintos valores asignados, no vamos a saber cuál de ellos estamos utilizando.
Debe notarse que un módulo puede ser importado en cualquier instancia de un programa python; obviamente su contenido va a estar disponible sólo luego de ser importado. Discutiremos este punto más en profundidad cuando veamos funciones.
Si trabajamos con Python en forma interactiva, podemos obtener información sobre el contenido de un módulo usando la instrucción `dir()`. En nuestro caso tendremos

```
dir(math)
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', ácos', ácosh',
ásin', ásinh', átan', átan2', átanh', 'ceil', 'comb', 'copysign', 'cos',
'cosh', 'degrees', 'dist', é', érf', érfc', éxp', éxpm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', ínf', ísclose',
ísfinite', ísinf', ísnan', ísqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', ñan', ñextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', úlp']
```

Los nombres con guiones bajos no contienen información de interés en un curso introductorio, sí en cambio los otros, que son las funciones que solemos utilizar. Para acceder al contenido de `sqrt`, por ejemplo, haremos

```
help(math.sqrt)
```

para obtener

```
Help on built-in function sqrt in module math:
```

[2] Todo texto escrito a derecha del símbolo # es interpretado por Python como un comentario. Es conveniente usarlo para detallar qué acciones se siguen en el desarrollo de un programa.

```
sqrt(x, /)
Return the square root of x.
```
[3]

1.5.5. Cadenas de caracteres

Tal como mostramos en la tabla 1.1, una cadena de caracteres (CDC) es una secuencia de letras, números y símbolos delimitados por comillas simples o dobles. Las CDCs son el primer ejemplo de lo que en Python llamamos *secuencias*, esto es, una colección ordenada posicionalmente de otros objetos. Las secuencias mantienen un orden de derecha a izquierda entre los elementos que contienen; los mismos pueden ser individualizados por su posición relativa. Hablando estrictamente, las CDCs son secuencias de CDCs de un único carácter; otro tipo de secuencias incluyen listas y tuplas, de las que nos ocuparemos en próximas secciones.
Vimos que podemos utilizar CDCs cuando deseamos imprimir mensajes utilizando la instrucción `print()`; también podemos asignar CDCs a variables y operar con ellas. Por ejemplo

```
nombre = Álfred'
apellido = "Wegener"
n_y_a = nombre + '' + apellido
print(n_y_a)
```

imprime como resultado `Alfred Wegener`. Operando con cadenas de caracteres, el símbolo `+` las concatena. Si deseamos averiguar la longitud (número de caracteres) de la CDC resultante, podemos utilizar la función intrínseca `len()`. Así, `len(n_y_a)` devuelve el valor 14.
De acuerdo a lo expresado, la cadena de caracteres guardada en la variable `nombre` consiste en seis caracteres a los que puede accederse en forma individual. Por ejemplo `nombre[3]` nos permite acceder al cuarto carácter, es decir, a la letra `r`.[4]
Por otra parte, Python nos permite acceder desde el último carácter hacia la izquierda. Así, `nombre[-1]` devuelve el carácter `d`, `nombre[-2]` devuelve el carácter `e`, y así sucesivamente.

El *operador de corte* `:` nos permite extraer porciones arbitrarias de una CDC; dejando a la original sin cambio alguno y generando una nueva. Su formato más general es el siguiente:

```
CDC[inicio:final:paso]
```

que resulta en una CDC que comienza con el carácter situado en el índice `inicio` y termina con el situado en índice `final-1`, saltando de a `paso`. Cualquiera de los índices pueden faltar; si `paso` no está (en cuyo caso también puede obviarse el segundo `:`) se asume que el paso es 1. Los dos primeros índices pueden tomar valores negativos; si ambos lo hacen, debe suceder que `inicio` $<$ `final`. Veamos algunos casos, tomando el ejemplo anterior

Corte	Resultado	
`n_y_a[0:3]`	`Alf`	
`n_y_a[:]`	`Alfred Wegener`	# La CDC completa
`n_y_a[4:]`	`ed Wegener`	# Desde el 4to carácter hasta el final.
`n_y_a[:8]`	`Alfred W`	# Desde el 1er carácter hasta el séptimo inclusive.
`n_y_a[-4:-1]`	`ene`	# Último carácter no incluido.
`n_y_a[::2]`	`Afe eee`	# Del 1er al último carácter, con paso dos.

Las CDC son objetos *inmutables* en Python. Esto significa que no está permitido cambiar elementos de las mismas una vez definidas. Por ejemplo

[3] El idioma de la respuesta depende de la configuración del sistema operativo.
[4] Python comienza contando desde el 0.

```
n_y_a[0]=a
```

produce un error al correr el código. Ciertamente, podemos obtener el resultado deseado generando una nueva cadena -o sobreescribiendo la original- concatenando el carácter `á'` con `n_y_a[1:]`.
Para finalizar, mencionamos que existen disponibles en Python una variedad de funciones intrínsecas definidas para operar sobre los distintos objetos (CDCs, tuplas, diccionarios) en forma específica. Para diferenciarlas de las que pueden tomar como argumento distintos objetos, se las denomina *métodos*, y son invocadas de una forma particular. Damos algunos ejemplos de métodos para CDCs, utilizando como argumento `n_y_a`, recordando que el resultado siempre es una nueva CDC u objeto, la aplicación del método no altera la original.

Método	Uso	Resultado
`upper()`	`n_y_a.upper()`	`ALFRED WEGENER`
`split()`	`n_y_a.split('f')`	`[Ál', 'red Wegener']`
`replace()`	`n_y_a.replace(Ál','Man')`	`'Manfred Wegener'`

El listado completo de métodos disponibles para CDCs puede encontrarse en la bibliografía de la Cátedra.

1.5.6. Listas

El objeto `lista` de Phyton es la secuencia más general provista por el lenguaje. Las listas son colecciones posicionalmente ordenadas de objetos, y no tienen un tamaño fijo. A diferencia de las CDCs -que como dijimos son inmutables- pueden ser modificadas, tanto por asignación a elementos individuales, como a través de una variedad de métodos intrínsecos disponibles para ellas. Por ejemplo, dada la lista

```
L1=['Geoide',125.4,42],
```

`L1[0]=[3,5,'Pangea']` reemplaza el primer elemento de `L1` por el dado, que en este caso es también una lista.
El operador de corte puede utilizarse para obtener una nueva lista a partir de otra. Así, si a posteriori del reemplazo anterior tomamos `L2=L1[0:2]`, la lista obtenida es `[[3,5,'Pangea'],125.4]]`.
Una característica muy útil de Python es que reporta como error intentar acceder a un elemento, o intentar asignar uno, más allá de los límites de una lista. Así, tanto `print(L2[5])` como `L2[4]=44.6` imprimirán mensajes de error, cortando la ejecución del código, puesto que los índices `5` y `4` están más allá de `1`, que es el índice superior de la lista `L2`.
Como se ve en la lista `L2`, estos objetos pueden también anidarse, es decir, pueden construirse listas de listas, o más generalmente, listas de cualquier objeto, que a su vez pueden contener otros.
Como son secuencias, las listas admiten todas las operaciones que aplicaron a CDCs por su calidad de tales, la diferencia es que los resultados son en general listas, y no CDCs. Por ejemplo, dada la lista `L1`, `len(L1)` devuelve `3`, `L1[1]` devuelve `125.4`, `L1[:-1]` devuelve la lista `['Geoide',125.4]`.
Algunos de los métodos intrínsecos para listas son los siguientes:

Método	Uso	Descripción
`append()`	`L1.append(4.2)`	Agrega un elemento al final de la lista
`copy()`	`L1.copy()`	Devuelve una copia de la lista
`count()`	`L1.count(ÑaCl')`	Número de elementos con un valor específico.
`index()`	`L1.index(ÑaCl')`	Índice del primer elemento con el valor especificado
`insert()`	`L1.insert(3,ÑaCl')`	Agrega un elemento en la posición especificada
`pop()`	`L1.pop(5)`	Quita un elemento en la posición especificada
`remove()`	`L1.remove(ÑaCl')`	Quita el primer elemento con el valor especificado
`reverse()`	`L1.reverse()`	Invierte el orden de la lista
`sort()`	`L1.sort()`	Ordena la lista

Los operadores aritméticos `+` y `*` aplicados a listas tienen el mismo comportamiento que aplicados a CDCs: el primero forma una nueva lista concatenando los elementos de las listas involucradas, ubicando los correspondientes al sumando derecho al final de los del primero. El símbolo `*` por su parte, sólo puede ser aplicado "multiplicando" un entero positivo por una lista; el resultado es una nueva lista, conteniendo los elementos de la lista original, repetidos en forma secuencial tantas veces como el factor utilizado. Así, dadas `L3=[1,2,3]` y `L4=[4,5,6]`, `L3+L4` resulta en `[1,2,3,4,5,6]` y `3*L3` resulta en `[1,2,3,1,2,3,1,2,3]`.

Listas de números -enteros, reales o complejos- ciertamente podrían ser útiles para representar vectores y matrices. Sin embargo, los dos últimos ejemplos nos muestran que las listas de números no siguen en Python las reglas del álgebra de vectores. Para poder trabajar adecuadamente con vectores y matrices, debemos hacer uso del módulo `numpy`, como veremos más adelante.

Para finalizar esta sección, comentamos sobre un comportamiento característico de las listas en Python, que debe ser tenido en cuenta para evitar resultados inesperados. Si `L1` es una lista y hacemos la asignación `L2=L1`, Python no crea una nueva lista, sino sólo una nueva referencia a la misma lista; dicho de otra forma, no se duplican en memoria los datos contenidos en la lista, sino que se crea una nueva referencia que apunta a los datos originales. En algunos textos, a esto se lo denomina crear una nueva *vista* de la lista. Esto implica que cualquier cambio que hagamos en `L1`, igualmente afecta a `L2`.
Si deseamos crear una lista a partir de una dada, pero que no tenga vinculación con ella, debemos usar alguna de las siguientes dos opciones: (a) `L2=L1[:]`, (b) `L2=L1.copy()`. La lista `L2` tiene así existencia independiente de `L1`, y cambios en esta última no la afectan.

1.5.7. Tuplas

Una tupla es una secuencia de objetos, no necesariamente del mismo tipo, separados por comas y encerrados entre paréntesis. Una posible, a la que asignamos a la variable `tupla_1` es

```
tupla_1=(2,'frecuencia',3.5,[1,2],'período',1+3j)
```

Este tipo de estructuras puede ser útil, en nuestras aplicaciones, cuando deseamos que una función retorne más de un objeto. Las tuplas no pueden ser modificadas una vez creadas -son, como las CDCs, inmutables-, pero sus elementos son accesibles según la posición que ocupan en ella. Debe recordarse que el primer elemento tiene asociado el índice 0. Así, `tupla_1[2]` devuelve `3.5`. Más aún, en forma análoga a lo hecho con CDCs y listas, el operador de corte `:` puede ser utilizado para extraer tuplas a partir de otras. Así, `tupla_1[::3]` devuelve la tupla `(2, [1, 2])`. Ciertamente, si deseamos asignar este resultado a una nueva variable, haremos

```
tupla_2=tupla_1[::3]
```

1.5.8. Diccionarios

Un diccionario es una estructura de datos que crea referencias a valores determinados; se lo define encerrando sus elementos entre llaves ({}). En cada entrada la referencia se separa del valor indicado por el símbolo `:`, y las distintas entradas se separan por comas (`,`). Pueden utilizarse si deseamos asociar un conjunto de valores con entradas, por ejemplo, para describir las propiedades de una cosa. Por ejemplo

```
M1={'Roca':Árenisca','Cantidad':4,'Peso':210.5}
```

asigna a la variable `M1` un diccionario de tres elementos, accesibles vía las entradas `Roca`, `Cantidad` y `Peso`. Se accede al valor de los elementos a través de las respectivas entradas. Así, `M1['Roca']` devuelve `Árenisca'`. Debe notarse que esta forma de acceso a los distintos componentes es diferente de lo visto hasta ahora; esto es así porque los elementos de los diccionarios no siguen un orden relativo, como sí lo hacen los elementos de CDCs, listas y tuplas.
A diferencia de estas últimas, los diccionarios pueden ser modificados, cambiando el número de sus elementos y los valores de sus entradas. Si bien brindan no pocas posibilidades, no profundizaremos en ellas en este curso.

1.6. Estructuras de decisión

1.6.1. La condición `if`

En términos simples, la declaración `if` de Python selecciona acciones a realizar. Su formato general es el siguiente:

```
if condición lógica 1:
           instrucción 1
           instrucción 2
           ⋮
           instrucción n
elif condición lógica 2:   # Opcional. Puede existir más de uno dentro del bloque
           instrucción 1
           instrucción 2
           ⋮
           instrucción m
else:                          # Opcional. Puede existir sólo uno dentro del bloque
           instrucción 1
           instrucción 2
           ⋮
           instrucción t
```

Si la `condición lógica 1` es verdadera, las instrucciones 1 a n se cumplen secuencialmente; con esto finaliza el bloque `if`. Si es falsa, se testea la `condición lógica 2`; si es verdadera, las instrucciones 1 a m se cumplen secuencialmente, y se sale del bloque `if`. Si es falsa, se ejecutan las instrucciones 1 a t englobadas dentro del bloque `else`.
Hemos dicho con anterioridad que las instrucciones en un programa Python deben ser escritas sin dejar espacio en blanco alguno antes de su comienzo. Esto cambia parcialmente en los blo-

ques `if`: todas las instrucciones a ejecutar -si alguna de las condiciones es verdadera- **deben** estar indentadas. De hecho, como no hay ningún símbolo explícito que denote el inicio y fin del bloque de instrucciones que deben ejecutarse para cada opción, es precisamente la indentación lo que indica la pertenencia de esa instrucción al bloque.
Para indentar, es posible utilizar tanto espacios en blanco como el tabulador. Es recomendable usar uno u otro, pero no mezclar ambos, puesto que esto puede dar lugar a errores de sintaxis, señalados por el compilador.
Python asume que un bloque `if` finaliza cuando encuentra la primera instrucción sin indentación; ésta no es evaluada como parte del bloque.
Veamos algunos ejemplos:

```
v=1.
w=3.
if v != w: v*=3
print(v,w)
```

En este caso, se muestra como resultado `3 3`. Cuando las acciones a realizar son sencillas, es posible utilizar la *versión de una línea* del bloque `if`, tal como se ve en el caso analizado. No es recomendable hacerlo en casos más complicados (cuando se incluye más de una instrucción en el bloque), puesto que la legibilidad del código se ve perjudicada. Otro caso:

```
v=1.
w=3.
z=5.
if v>w or 3.*v<z:
     v*=3
     print('v= ',v,'w= ',w,'z= ',z)
elif 5.*v > w+z:
     print('v= ',v,'w= ',w,'z= ',z)
else:
     z=5*v-w
```

En este ejemplo, se muestra como resultado `v= 3.0 w= 3.0 z= 5.0`.
Una característica de Python es que toda constante o variable no nula, u objeto no vacío tiene asociado el valor de verdad `True`. Por lo tanto, puede ser utilizado como condición lógica, o parte de una, en un bloque `if`. Así

```
v=1.
if v:
     v+=5
print('v= ',v)
```

devuelve el resultado `v= 6` ya que al ser no nula la variable `v` la condición es evaluada como verdadera, y por lo tanto la instrucción dentro del bloque es ejecutada. El entero `0`, el real `0.`, la CDC vacía `''`, la lista vacía `[]` y el diccionario vacío {} tienen valor de verdad `False`.
Mencionamos finalmente que los bloques `if` pueden anidarse, esto es, un bloque `if` puede ser una de las instrucciones a ejecutar dentro de un bloque `if`. En este caso, debe tenerse cuidado en preservar las indentaciones dentro de los respectivos bloques, para que las cosas funciones como deseamos. Veamos un ejemplo

```
x=1
```

```
if x:
     y=2
     if y:
          print(Éstoy en bloque 2')
     print(Éstoy en bloque 1')
print('Salí del if')
```

1.6.2. La expresión ternaria `if/else`

En ocasiones, se desea asignar a una variable distintas expresiones según una expresión lógica sea verdadera o falsa. Por ejemplo,

```
v=-5.
if v>2:
     a=v**2+44.
else:
     a=-22.
```

Python permite escribir esta condición en una sola línea, haciendo

```
a=(v**2+44.) if v>2. else -22.
```

Esta expresión genera exactamente el mismo resultado que el bloque `if` anterior; esto es, para el valor de la variable `v` elegido, se le asigna a `a` el valor `-22.` En caso que las expresiones dentro del bloque `if` sean sencillas, utilizar este formato para escribir el código puede resultar más simple. Su forma general es

```
variable = expresión_1 if condición lógica else expresión_2
```

1.7. Estructuras de repetición

1.7.1. Bucles `while`

El bucle `while` de Python es la construcción de iteración más general del lenguaje. En términos simples, ejecuta *repetidamente* un bloque de instrucciones siempre que una dada condición lógica siga evaluando un valor verdadero. Se llama "bucle" porque el control continúa volviendo a su inicio hasta que la condición se vuelve falsa. Cuando esto sucede, el control pasa a la instrucción que sigue al bloque `while`. El efecto neto es que el cuerpo del bucle se ejecuta repetidamente mientras la condición establecida es verdadera; si la misma es falsa al empezar, el bloque de instrucciones es directamente salteado.

Forma general

En su forma más compleja, el bucle `while` consta de una línea de encabezado con una condición lógica, un cuerpo de una o más instrucciones y una parte opcional `else` que se ejecuta si (a) la `<condición lógica>` es falsa, o (b) si el control sale del bucle sin que se encuentre una declaración de interrupción `break`; esta instrucción causa una salida inmediata del bucle. Python continúa evaluando la condición lógica en la parte superior y ejecutando las declaraciones anidadas en el cuerpo del bucle hasta que la primera devuelve un valor falso:

```
while condición lógica:
      instrucción 1
      ⋮
```

```
      instrucción n
else:                          # Opcional. Si está, las instrucciones 1 a m corren si el
      instrucción 1            # bucle no terminó con break
      ⋮
      instrucción m
```

Mencionamos finalmente que `continue` es otra instrucción que puede formar parte de un bucle `while`. En caso de estar presente, y ser ejecutada, vuelve inmediatamente al principio del bucle, para que siga corriendo normalmente.
Veamos algunos ejemplos, comenzando con un caso sencillo, en el cual no incluimos la instrucción `else`:

```
x=1
while x <= 100:
     print(x)
     x+=1
```

En este caso se imprimen los primeros cien naturales, uno por línea. Consideremos ahora el cómputo de la posición de un cuerpo en caída libre mediante

$$y = y_0 - \frac{1}{2}gt^2,$$

desde una altura $y_0 = 10$ m, hasta el momento de impacto en el suelo, asumiendo intervalos de medida de 0.1 s.

```
t=0.
dt=0.1
y=y0=10.          # Python permite asignaciones múltiples.
while y >= 0.:
     print('t= ',t,'y= ',y)
     t+=dt
     y = y0 - 0,5 * g * t **2
...
```

En este ejemplo se imprime en pantalla el tiempo transcurrido y la altura correspondiente, se incrementa el tiempo y se computa la nueva altura. Si ésta es mayor o igual a cero, el bucle vuelve a ejecutarse. Cuando la desigualdad deja de cumplirse, el programa sigue con la instrucción debajo del bucle. Una pregunta que es válido plantearse es la siguiente: ¿Podríamos haber escrito la condición del bucle como

```
while y != 0. ?
```

La respuesta es que sintácticamente, esta opción es válida. Sin embargo, dado que incrementamos el tiempo en forma discreta, es muy probable que nunca la altura sea exactamente cero, con lo que el bucle se ejecutaría infinitamente. Entonces, cuando se utilizan números reales, es recomendable usar como condiciones en los bucles `while` expresiones lógicas que contengan desigualdades.
Veamos ahora un ejemplo de descomposición de una cierta sustancia por decaimiento radiactivo, dada por la ley

$$N(t) = N_0\, e^{-\lambda t},$$

utilizando la estructura completa del bucle `while`. Aquí, $N(t)$ es la masa de sustancia al tiempo t, N_0 la masa inicial, y λ la constante de decaimiento radiactivo, que mide la probabilidad de

decaimiento por unidad de tiempo.

```
import math

masa_inicial = 100.0          # Masa inicial (en grs.)
lambda_decaimiento = 0.1      #Constante de decaimiento
masa_umbral = 1.0
tiempo = 0.                   # Tiempo (unidades arbitrarias)
masa = masa_inicial
dt = 0.01                     # Paso de tiempo (unidades arbitrarias)

while masa > masa_umbral:
      masa = masa_inicial * math.exp(-lambda_decaimiento * tiempo)
      tiempo += dt
      print(f"Tiempo: {tiempo}, Masa: {masa:.4f} gramos")    # Ver Sección 1.10
      if masa < masa_umbral:
           print("La masa ha caído por debajo del umbral.")
           break

print("Proceso de decaimiento completo.")
```

Vemos que la instrucción `break` está incluida, como es frecuente, dentro de una condición `if`. En este caso, cuando la masa remanente de sustancia sea menor que la masa umbral establecida, se imprime el mensaje a pantalla, y se ejecuta la instrucción `break`, dando por terminado el bucle. Veamos finalmente un ejemplo de uso de la instrucción `continue`,

```
x = 11
while x:
     x -= -1
     if x% 2 != 0:
          continue
     print(x)
```

donde simplemente se imprimen en orden decreciente los números pares, de `10` a `0`.

1.7.2. Bucles `for`

El bucle `for` es un iterador genérico en Python: puede recorrer de a uno los elementos en cualquier secuencia ordenada u objeto iterable. Es aplicable sobre CDCs, listas y tuplas (y otros iterables intrínsecos del lenguaje, además de iterables definidos por el usuario, de los que no nos ocuparemos en este apunte). Su formato general está dado por

```
for variable_de_control_del_bucle in objeto
      instrucción 1
      ⋮
      if condición: break          # Opcional.
      if condición: continue       # Opcional.
      ⋮
      instrucción n
else:                              # Opcional. Si está, las instrucciones 1 a m corren
      instrucción 1                # si el bucle no terminó con break
```

```
    ⋮
    instrucción m
```

Cuando este bucle es ejecutado, a la `variable_de_control_del_bucle` (VCL) se le asigna uno a uno los elementos de `objeto` -repetimos, que es una secuencia ordenada u objeto iterable- y ejecuta todas las instrucciones de `1` a `n` para cada uno de ellos.
La VCL toma cualquier nombre, siguiendo la misma sintaxis que las variables. Su valor puede ser modificado dentro del bucle, pero el mismo será automáticamente asignado al siguiente elemento de `objeto` cuando el control retorne al inicio del bucle. Al finalizar el bucle, la VCL retiene como valor el último asignado.
Las instrucciones `break` y `continue` funcionan de igual forma que en el bucle `while`; en el primer caso, si la `condición` es evaluada como verdadera, la ejecución del bucle es interrumpida inmediatamente, siguiéndose el flujo del código posterior al bucle. En el segundo caso, las instrucciones dentro del bucle debajo de esta línea no son evaluadas para el valor corriente de la VCL y se vuelve al inicio del bucle, tomando la VCL el siguiente valor. El bloque `else` funciona en forma idéntica que lo hace en el bucle `while`, es decir, las instrucciones asociadas sólo serán ejecutadas si el bucle `for` finalizó sin llevar a cabo un `break`.
Veamos algunos ejemplos:

```
sum=0
for x in [1,2,3,4,5]
    sum +=x
print(sum)
```

Ejecutar este código imprime el valor `15`; la variable `x` retiene el valor `5`.
En este ejemplo, el objeto es una lista, y por lo tanto, es iterable. Como sus elementos son números enteros, podemos sumarlos, el bucle for recorre toda la lista y va sumando de a uno sus elementos.

```
A=ÁNyP'
for letra in A:
    print(letra, end='_')
```

Ejecutar este código imprime `A_N_y_P_`. En este caso, el bucle recorre la cadena de caracteres, imprimiendo de a uno sus elementos; notar que el `print` está dentro del bucle. La opción `end='_'` del `print` establece que a continuación de su ejecución se adose un guión bajo.

Range

Para problemas donde debemos repetir una operación o serie de operaciones un gran número de veces, como por ejemplo sumar los primeros 1000 naturales, es conveniente la utilización de la función intrínseca `range`, que produce una serie de números enteros sucesivos que pueden ser utilizados como índices en un bucle `for`. Su sintaxis es

```
range(comienzo, fin, paso)
```

Aquí comienzo y paso son opcionales. Si no están presentes, se asume que el la secuencia arranca en `0` y el paso ó incremento es `1`. Debe tenerse en cuenta que la expresión dada genera secuencias hasta `fin`, pero sin incluirlo. Damos un par de ejemplos de uso:

```
range(5)
```

crea el iterable con índices de `0` a `4`. Para visualizarlo, podemos incluirlo en una lista haciendo `print(list(range(5)))`. Por otro lado,

```
range(-1,7,2)
```

crea el iterable con índices de `-1`, `1`, `3`, `5`, ya que como dijimos, el límite superior no es incluido en el resultado.
De acuerdo a lo dicho, el código para el ejemplo mencionado es simplemente:

```
sum=0
for x in range(1,1001):
     sum+=x
```

Hacer `print(sum)` nos mostrará el resultado `500500`, como esperamos.
Los bucles `for` pueden anidarse; en ese caso, para cada valor del iterable más externo, el iterable más interno es recorrido completamente.

```
for i in range(10,30,10):
     for j in range(1,4):
        print(í= ',i,'j= ',j,i+j)
```

resulta en la impresión en pantalla de

```
i= 10 j= 1 11
i= 10 j= 2 12
i= 10 j= 3 13
i= 20 j= 1 21
i= 20 j= 2 22
i= 20 j= 3 23
```

En el caso de bucles anidados, la existencia de la instrucción `break` o `continue` afecta al bucle en el que la misma se ejecuta y a los internos al mismo. Por supuesto, si un `break` es ejecutado en el bucle más externo, el programa salta inmediatamente a la instrucción posterior a los bucles anidados.

Comprensión de listas

Uno de los posibles usos del bucle `for`, juntamente con la instrucción `range` es la construcción de listas. Un caso de interés particular por las aplicaciones que tiene en cómputo numérico, es la de una lista conteniendo un conjunto discreto de abscisas equiespaciadas dentro de un intervalo de reales dado. De acuerdo a lo visto hasta ahora, para obtenerla podemos hacer

```
a=0.;b=10.              # Extremos inferior y superior del intervalo
N=5                     # Número de intervalos
dx=(b-a)/N              # Separación entre dos abscisas consecutivas
L=[]                    # Se inicializa la lista.
for i in range(N+1):
     L.append(i*dx)
```

El resultado es la lista `[0.0, 2.0, 4.0, 6.0, 8.0, 10.0]`. Podemos hacer lo mismo, pero en forma más eficiente, empleando una posibilidad que nos brinda Python que es la *Compren-*

sión de listas, que nos permite crear listas aplicando una expresión a cada elemento de una secuencia, opcionalmente filtrando elementos que cumplen cierta condición. Su sintaxis es

```
[nuevo_elemento for elemento in secuencia if condición]
```

Aquí, `nuevo_elemento` es el resultado a incorporar en la lista, `elemento` es el que está siendo iterado en la `secuencia`, y `condición` es opcional: puede ser utilizada para incluir en la lista solo aquellos elementos que cumplan con un criterio específico. Utilizando este nuevo concepto, el ejemplo anterior puede reescribirse como

```
a=0.;b=10.
N=5
dx=(b-a)/N
L=[i*dx for i in range(N+1)]
```

La lista L es obviamente igual a la anterior.
Para finalizar, dos comentarios: (a) el uso de comprensión de listas permite escribir códigos compactos y es más eficiente que el uso del `for` en el formato del ejemplo anterior. (b) Los bucles `for` pueden anidarse también en la comprensión de listas.

1.8. Arreglos

A lo largo de la carrera vamos a necesitar hacer cálculos con vectores y matrices, o utilizando el lenguaje computacional, con arreglos uni- y bidimensionales. De acuerdo a lo que hemos visto, las *listas* de Python son en principio buenas candidatas para ello. Sin embargo[5] varias de las propiedades que poseen no cumplen con las definidas para los objetos matemáticos vectores y matrices. Por ejemplo, recordemos que, dada la lista `L=[1,2,3]`, hacer `3*L` devuelve la lista `[1,2,3,1,2,3,1,2,3]`, y no la lista `[3,6,9]` como necesitamos que suceda en el contexto de la realización de operaciones matemáticas sobre vectores y matrices.

Afortunadamente, el módulo `numpy` define un conjunto de estructuras de datos y funciones útiles para el cómputo científico, entre ellos vectores y matrices y el álgebra adecuada para operar con ellos. Este módulo define también arreglos multidimensionales, aunque no vamos a profundizar en su utilización.
Para trabajar con arreglos, antes de realizar alguna operación que los involucre, debemos invocar el módulo en cuestión. Para ello hacemos, utilizando la convención usual,

```
import numpy as np
```

Asumiremos en todos los ejemplos de esta sección que esta instrucción fue previamente incorporada al código. Para definir la matriz

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

haremos simplemente

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

Cada fila tiene tres elementos, por lo que escribimos la matriz como una lista con tres elementos,

[5] Recomendamos hacer un repaso de la sección 1.5.6 antes de seguir avanzando!

sus tres filas. Aunque el arreglo `A` parece una lista de listas de Python, no es exactamente lo mismo. Una matriz `numpy` es una matriz propiamente dicha con multiplicación -y demás operaciones entre matrices- propiamente dichas, mientras que la lista de listas de Python, como ya lo hemos comentado, no lo es.

Debe tenerse en cuenta que (a) todos los elementos del arreglo deben ser del mismo tipo, (b) una vez definido el tamaño total -el número de elementos- del arreglo no puede ser modificado y (c) la forma es siempre rectangular, es decir todas las filas de un arreglo bidimensional deben tener igual número de columnas.

Es muy importante recordar que Python comienza contando desde `0`, de modo que al elemento a_{11} se accede escribiendo `A[0,0]` (o `A[0][0]`) y al elemento a_{33}, escribiendo `A[2,2]` o (o `A[2][2]`). Tener esto presente evita cometer errores!

Los arreglos definidos mediante `np.array()` cumplen con todas las propiedades del álgebra de matrices, de modo que pueden ser sumados, restados y multiplicados por un escalar y entre sí.
Sin embargo, hay algunas características de `numpy` que debemos tener en cuenta: Para la suma y resta entre arreglos utilizamos los símbolos `+` y `-`. *(i)* Si sumamos(restamos) un escalar a un arreglo, el escalar se suma(resta) con todos los elementos del arreglo. *(ii)* Si sumamos(restamos) un vector fila $1 \times n$ a una matriz $m \times n$, el vector se suma(resta) a las m filas de la matriz. Análogamente sucede con un vector columna. A este comportamiento característico es denominado *Propagación o Difusión*[6]. *(iii)* Los símbolos `*` y `/` multiplican y dividen, respectivamente, los arreglos *elemento a elemento*, y siguen los mismos preceptos que la suma y resta. *(iv)* El producto usual, para matrices compatibles, se realiza utilizando el símbolo `@`. Por ejemplo, dados

```
A=np.array([[1,2]])
B=np.array([[5,6],[7,8]])
print(A+10)
print(A*B)
print(A@B)
```

imprime en pantalla, respectivamente,

```
[[11 12]]

[[5 12]
 [7 16]]

[[19 22]]
```

que está de acuerdo con las definiciones dadas.

Atributos de arreglos

Los atributos son, en general, datos contenidos en un objeto. `numpy` provee a los arreglos con atributos que permiten o facilitan su manipulación en forma eficiente.
Presentamos aquí a `ndim, shape, size` y `dtype`.
El atributo `ndim` es el número de dimensiones de un arreglo. Cada una de ellas toma el nombre de *eje*; en arreglos bidimensionales, el eje `0` es el correspondiente a las filas, el eje `1` a las colum-

[6] Broadcasting en inglés, el idioma utilizado por los desarrolladores de Python.

nas. Veremos que algunas funciones con arreglos como argumento permiten elegir el eje sobre el cual se aplican.
El atributo `shape` de arreglo es una tupla de enteros no negativos que especifica el número de elementos en cada eje.
El atributo `size` es el número total de elementos en un arreglo. Es un valor fijo.
El atributo `dtype` es el tipo de dato contenido en el arreglo.
Para conocer el valor de un atributo de un arreglo, lo invocamos en forma similar a los métodos, pero sin utilizar paréntesis al final. Por ejemplo, si hacemos

```
A=np.array([[1.,2.],[3.,4.],[5.,6.]])
print(A.ndim,A.shape,A.size,A.dtype)
```

la salida en pantalla es

```
2, (3,2), 6, 'float64'
```

Operador de corte

El operador de corte `:` permite obtener arreglos a partir de porciones de otros, en forma similar a la que fue descripta en la Sección 1.5.6. Por ejemplo, si

`A = np.array([[1,2,3],[4,5,6],[7,8,9]])`,

hacer

```
B=A[0:3:2,1:3]
```

define la matriz 2×2

$$\mathbf{B} = \begin{pmatrix} 2 & 3 \\ 8 & 9 \end{pmatrix}.$$

Una característica muy importante del operador de corte al utilizarlo en arreglos, es que los obtenidos no son arreglos independientes de los originales, sino *vistas* parciales de ellos. De esta manera, cualquier cambio en el arreglo derivado se ve reflejado en el arreglo original. Este comportamiento debe ser tenido en cuenta para evitar errores involuntarios en los programas. Podemos ver un caso si en el arreglo `B` del ejemplo anterior hacemos

```
B[0,0]=55
```

No sólo toma este valor el elemento b_{11}, sino el correspondiente en el arreglo `A`, como podemos ver con un `print(A)`:

```
[[1 55 3]
 [4 5 6]
 [7 8 9]]
```

Si deseamos que el arreglo `B` sea totalmente independiente del arreglo `A`, debemos usar el método `copy()`:

```
B=A[0:3:2,1:3].copy()
```

De esta manera, cambios en el arreglo `B` no alteran al arreglo `A`.

Vectores y cambios de forma y de tamaño

Consideremos las siguientes instrucciones y sus resultados

```
B=np.array([[1.,2.]])
print(B.ndim,B.shape,B.size)

2 (1, 2) 2

C=np.array([1.,2.])
print(C.ndim,C.shape,C.size)

1 (2,) 2
```

Es importante notar la diferencia al definir ambos arreglos; en el primer caso usamos doble corchete, mientras que en el segundo sólo uno. El arreglo `B` es bidimensional, de una fila por dos columnas y tiene dos elementos. El arreglo `C` se denomina *vector* en Phython. Es unidimensional, de dos elementos, y no puede discriminarse si es un vector fila o columna. Una característica de los vectores es que se puede acceder a sus elementos usando un único índice. Así, `print(C[0])` devuelve el valor `1.`. Por el contrario, como el arreglo `B` es bidimensional -a pesar que uno de sus ejes vale 1, y por lo tanto desde el punto de vista algebraico es un vector fila- hacer `print(B[0])` devuelve `[1.,2.]`. Para imprimir el primer elemento de la primer fila debemos utilizar dos índices: `print(B[0,0])`, que ciertamente también devuelve el valor `1.`.
Ambos formatos pueden utilizarse para hacer álgebra de vectores. Sin embargo, puede darse la situación que una función utilice como argumento un vector y no un arreglo bidimensional con un eje de valor `1`; como vimos desde el punto de vista de Python, su forma (`shape`) difiere, y por lo tanto son objetos diferentes. En estas circunstancias, y ciertamente en otras que sea útil, es posible cambiar la forma del vector o arreglo -sin modificar el número de elementos, es decir manteniendo su atributo `size`, mediante la función `reshape`:

La función `reshape` de `numpy` es usada para cambiar la forma de un arreglo sin cambiar sus datos, sin modificar el original. Su sintaxis es la siguiente:

```
np.reshape(arreglo, nueva_forma, order=ópción')
```

Aquí `arreglo` es el arreglo original, `nueva_forma` es la forma deseada para el nuevo arreglo. Puede ser indicado como un entero, o una tupla de enteros. Si el valor de los ejes es `-1`, `numpy` infiere su valor verdadero a partir del tamaño total del arreglo original. Finalmente `order`, que es opcional, indica el orden en el que se van a leer y escribir los elementos; `opción` toma cuatro valores posibles: `'C', 'F', Á', 'K'`. En el primer caso, que es la opción por defecto cuando `order` no es explícito, ubica a los elementos en el nuevo arreglo siguiendo las filas del arreglo original. Si la opción es `'F'`, la ubicación se hace siguiendo las columnas del arreglo original. La tercera opción, `Á'` sigue el formato `'F'` si el arreglo original tiene este mismo, caso contrario sigue el formato `'C'`. Finalmente, la opción `'K'` trata que la apariencia del resultado sea lo más parecida posible al del arreglo de entrada. Mostramos dos ejemplos:

```
a=np.array([[1.,2.],[3.,4.],[5.,6.]])
b=np.reshape(a,(6,),order='F')
print(b)

[1. 3. 5. 2. 4. 6.]
```

```
c=np.reshape(b,(3,2),'C')
print(c)

[[1. 3.]
 [5. 2.]
 [4. 6.]]
```

Notar el ordenamiento de los coeficientes en los arreglos resultantes, de acuerdo a la opción elegida en cada caso. Existen otras funciones para modificar las formas de los arreglos, para realizar variadas operaciones de tipo de trasposición, para cambiar sus dimensiones, para unirlos, dividirlos y para agregar y quitar elementos selectivamente, entre otras. En la siguiente tabla damos mencionamos algunas:

Función	Uso	Descripción
`np.squeeze`	`np.squeeze(arr)`	Elimina las dimensiones de tamaño 1 de un arreglo, reduciendo su dimensionalidad sin afectar los datos. Útil para eliminar dimensiones no deseadas.
`np.newaxis`	`arr[:, np.newaxis]`	Añade una nueva dimensión al arreglo en la posición especificada. En este ejemplo, se añade una nueva dimensión al arreglo para convertir un vector fila en un vector columna.
`np.hstack`	`np.hstack((arr1,arr2))`	Apila arreglos horizontalmente (de manera que las columnas se concatenan). Los arreglos deben tener la misma cantidad de filas.
`np.vstack`	`np.vstack((arr1,arr2))`	Apila arreglos verticalmente (de manera que las filas se concatenan). Los arreglos deben tener la misma cantidad de columnas.
`np.concatenate`	`np.concatenate((arr1,arr2),axis=0)`	Une dos o más arreglos a lo largo de un eje especificado. En este ejemplo, concatena a lo largo del eje 0 (filas).
`np.resize`	`np.resize(arr, (2, 3))`	Cambia el tamaño de un arreglo a una forma específica. Si el nuevo tamaño es mayor que el original, los elementos se repiten.
`np.append`	`np.append(arr, values, axis=0)`	Añade valores al final de un arreglo a lo largo de un eje especificado. Si no se especifica un eje, el arreglo se aplana antes de añadir los valores.

Funciones intrínsecas sobre arreglos

El módulo `numpy` tiene definido un gran número de arreglos especiales, métodos y funciones sobre arreglos. En esta sección mostramos algunos de ellos, de uso frecuente. Para profundizar, consultar en la bibliografía de la cátedra.

Todas las funciones usuales utilizadas en matemáticas (trigonométricas, exponencial, logaritmos, trigonométricas hiperbólicas, etc.) que podemos aplicar sobre variables escalares admiten

arreglos como argumento; su acción se propaga sobre cada uno de los elementos del arreglo. Así, si

```
v=np.pi
a=np.array([[v/2.,v],[3.*v/2.,2.*v]])
print(np.sin(a))
```

resulta en

```
[[1.0000000e+00 1.2246468e-16]
 [-1.0000000e+00 -2.4492936e-16]]
```

que es una matriz en la que la función **sen(x)** ha sido aplicada a cada uno de sus elementos. Este resultado es correcto, dada la precisión con que la computadora realiza el cálculo. Veremos más sobre esto más adelante, al finalizar las clases de Python.

Arreglos y matrices predefinidas

A continuación damos ejemplos de funciones definidas en el módulo `numpy` para la creación de arreglos unidimensionales.

Función	Uso	Descripción
`np.linspace`	`np.linspace(0, 1, 5)`	Genera un arreglo de valores equidistantes dentro de un intervalo especificado. El tercer argumento define el número de puntos.
`np.arange`	`np.arange(0, 10, 2)`	Crea un arreglo con valores que van desde un valor inicial hasta un valor final, con un paso específico. Es similar a la función `range` de Python, pero devuelve un arreglo.
`np.logspace`	`np.logspace(0, 3, 4)`	Genera un arreglo de valores equidistantes en una escala logarítmica (base 10). El primer argumento es el exponente inicial, el segundo es el exponente final, y el tercero define el número de puntos.

Otras funciones permiten la creación de matrices particulares:

Función	Uso	Descripción
`np.eye`	`np.eye(3)`	Crea una matriz identidad. Su tamaño se especifica por el valor de entrada.
`np.ones`	`np.ones((2,4))`	Crea un arreglo de unos. La forma del arreglo se especifica mediante una tupla de dimensiones.
`np.random.rand`	`np.random.rand(2,3)`	Genera un arreglo de la forma especificada de valores aleatorios distribuidos uniformemente entre 0 y 1.
`np.random.randint`	`np.random.randint(0,10,(3,3))`	Genera un arreglo de la forma especificada lleno de enteros aleatorios en un rango determinado (en este caso, de 0 a 9).

Las siguientes funciones realizan cómputos sobre todo el arreglo; en el caso que sea multidimensional, puede opcionalmente elegirse un eje particular sobre el cual operar.

Función	Uso	Descripción
`np.max`	`np.max(arr, axis=0)`	Devuelve el valor máximo de un arreglo. En este ejemplo, calcula el máximo a lo largo del eje 0 (filas).
`np.min`	`np.min(arr, axis=1)`	Devuelve el valor mínimo de un arreglo. En este ejemplo, calcula el mínimo a lo largo del eje 1 (columnas).
`np.sum`	`np.sum(arr, axis=0)`	Calcula la suma de los elementos a lo largo del eje 0 (filas). Si no se especifica un eje, suma todos los elementos del arreglo.
`np.mean`	`np.mean(arr, axis=1)`	Calcula el valor promedio (media aritmética) de los elementos a lo largo del eje 1 (columnas). Si no se especifica un eje, calcula la media de todos los elementos del arreglo.
`np.prod`	`np.prod(arr, axis=0)`	Calcula el producto de los elementos a lo largo del eje 0 (filas). Si no se especifica un eje, multiplica todos los elementos del arreglo.
`np.cumsum`	`np.cumsum(arr, axis=1)`	Calcula la suma acumulativa de los elementos a lo largo del eje 1 (columnas). Devuelve un arreglo de la misma forma.
`np.cumprod`	`np.cumprod(arr, axis=0)`	Calcula el producto acumulativo de los elementos a lo largo del eje 0 (filas). Devuelve un arreglo de la misma forma.

Expresiones condicionales y arreglos lógicos

Cuando es necesario hacer comparaciones entre elementos de arreglos, en el contexto de `numpy` podemos hacerlo fácilmente utilizando los operadores relacionales que vimos en la Sección 1.5.3; todos ellos operan elemento a elemento. Además, vale también en estos casos la propagación de la operación realizada. Consideremos los siguientes ejemplos:

```
a=np.array([1,2,3,4])
b=np.array([4,3,2,1])
print(a>=b)
```

que resulta en

```
[False False True True]
```

Vemos que el resultado es un arreglo de igual tamaño que los comparados cuyos elementos son el resultado de la aplicación del operador relacional elemento a elemento entre los arreglos considerados. En este caso no hay propagación, ya que los arreglos comparados son de igual forma y tamaño. Si en cambio hacemos

```
a=np.array([[1,2,3,4],[1,2,3,4]])
b=np.array([[1],[1]])
print(a<=b)
```

el resultado en este caso es

```
[[True False False False]
 [True False False False]]
```

donde se observa que la comparación del vector columna fue realizada contra todas las columnas del arreglo de mayor dimensión.

Una característica muy útil de los arreglos lógicos es que nos permiten obtener arreglos a partir de otros, seleccionando los elementos que cumplen con una condición dada. Esta propiedad es fácilmente entendible a partir de ejemplos; sea

```
a=np.array([[-1,2,3,-4],[5,2,-3,9],[7,0,8,-6],[-5,-2,3,8]])
b=a[a>0]
print(a>0)
print(b)
```

devuelve el arreglo lógico 4×4

```
[[False True True False]
 [True True False True]
 [True False True False]
 [False False True True]]
```

y el vector

```
[2 3 5 2 9 7 8 3 8]
```

que no es otra cosa que un arreglo formado con los elementos del arreglo `a` que cumplen con la condición de ser positivos. Nótese que si bien el arreglo lógico `a>0` tiene la misma forma que el arreglo `a`, no tiene sentido en principio que el arreglo `b` sea de 4×4, ya que el número de coeficientes positivos no se conoce a priori. Si se desea que éstos preserven sus lugares en un arreglo de igual forma que el original, una posibilidad es hacer

```
c=a*(a>0)
print(c)
```

que resulta en

```
[[0 2 3 0]
 [5 2 0 9]
 [7 0 8 0]
 [0 0 3 8]]
```

¿Por qué sucede esto? En la primer línea de nuestro ejemplo estamos asignando a la variable `c` el resultado del producto elemento a elemento entre un arreglo de números y uno lógico. Esto es lícito en Python; en toda operación que involucre a constantes numéricas y las constantes lógicas `True` o `False`, automáticamente se asigna a la primera el valor uno, y a la segunda el valor 0.
Finalizamos esta sección dando algunas de las funciones lógicas que provee el módulo `numpy`.

Función	Uso	Descripción
`np.all`	`np.all(arr>0)`	Devuelve `True` si todos los elementos de un arreglo cumplen con una condición. Se puede aplicar a lo largo de un eje específico.
`np.any`	`np.any(arr<0)`	Devuelve `True` si al menos un elemento del arreglo cumple con una condición. También se puede aplicar a lo largo de un eje específico.
`np.where`	`np.where(arr>0,1,-1)`	Devuelve un arreglo basado en una condición: si la condición es `True`, devuelve un valor; si es `False`, devuelve otro valor.
`np.choose`	`np.choose([0,1],[a,b])`	Selecciona elementos de varios arreglos basándose en un arreglo de índices. En este ejemplo, elige entre los arreglos `a` y `b` según los índices en el primer argumento.
`np.select`	`np.select([cond1,cond2],[x,y],default=0)`	Selecciona elementos de varios arreglos basándose en múltiples condiciones. Devuelve un arreglo donde los elementos se eligen de acuerdo con la primera condición `True` en el orden especificado.

1.9. Funciones

A lo largo del apunte hemos utilizado, o simplemente listado dando ejemplos de su uso, distintas funciones intrínsecas provistas por Python. En esta sección daremos un marco formal a estas estructuras del lenguaje, y aprenderemos a generar nuevas.
Comenzamos describiendo una función como un dispositivo que agrupa un conjunto de instrucciones para que puedan ejecutarse más de una vez en un programa. Las funciones también pueden calcular un valor de resultado y permitirnos especificar parámetros que sirven como entradas de la función, que pueden diferir cada vez que se ejecuta el código. Codificar una operación como una función la convierte en una herramienta generalmente útil, que podemos usar en una variedad de contextos.
Más fundamentalmente, las funciones son la alternativa a la programación mediante cortar y pegar: en lugar de tener múltiples copias redundantes del código de una operación, podemos reducirlo a una sola función. Al hacerlo, minimizamos radicalmente nuestro trabajo futuro: si la operación debe cambiarse más tarde, solo tenemos una copia para actualizar, no muchas.
Las funciones son la estructura de programa más básica que Python proporciona para maximizar la reutilización del código y minimizar su redundancia, permitiéndonos dividir programas complejos en partes manejables. Su utilización debe ser priorizada cuando los programas crecen en complejidad, dado que facilita su lectura y la detección y consiguiente corrección de errores.

Definiendo funciones

En Python, se comienza definiendo una función usando la instrucción `def`. Luego especificamos su nombre de la función y la/s variable/s de entrada; si son más de una, las separamos con comas. Cerramos la línea de la definición con el operador de corte `:`. A continuación sigue el

bloque de código con el proceso que deseamos realice la función. El resultado generalmente se indica al final de la función con la palabra clave `return`; aunque esto no es obligatorio. De hecho, pueden definirse funciones que tengan más de una instrucción `return`, en cuyo caso el control al código que invoca a la función retorna inmediatamente después de la ejecución de la primera alcanzada, o que no tengan ninguna. En este caso, el control es retornado al código invocante al finalizar la ejecución de todas las instrucciones del bloque. En caso de estar presente, a través del `return` podemos devolver múltiples objetos.
La estructura general de una función es la siguiente:

```
def nombre_de_función(variable 1,variable 2,...):
  instrucción 1
  instrucción 2
  ⋮
  instrucción m
  return resultados
```

Como dijimos, `resultados` puede ser más de una variable; en ese caso, las variables a devolver simplemente se separan con ","de modo tal que forman una tupla; en el módulo que invoca la función podemos acceder a los distintos elementos de esta tupla. Definamos ahora nuestra primer función:

```
def pordos(x):
  return 2*x
```

Debemos notar que *(i)* De la misma manera que en todo programa python, no es necesario declarar el tipo de variable/s que es/son utilizada/s en la función. *(ii)* Es posible realizar operaciones en la línea del `return`. Esto no es recomendable en el caso que dificulte la lectura del código. Hacemos ahora uso de la función recién definida:

```
z=4.
y=pordos(z)
print('y= ',y)
t=ÁNyP'
y=pordos(t)
print('y= ',y)
```

que resulta en

```
y= 8.0
y= ANyPANyP
```

A partir de estos ejemplos, podemos además mencionar que *(i)* La ejecución de la función se realiza simplemente invocando su nombre, habiendo asignado un valor a la variable de entrada. No es necesario que la misma tenga el mismo nombre que la que fue utilizada al definir la función; al invocarla, el programa realiza una copia del valor de la variable que deseamos ingresar y la pasa a la variable de entrada de la función. *(ii)* Es posible invocar a la función pasando directamente una constante; por ejemplo, usar `pordos(5.)` es totalmente válido. *(iii)* Podemos utilizar el resultado de la función como parte de una expresión, sin necesariamente asignar este valor a una variable; si `3.-pordos(1.)+7.` es parte de un cómputo, la función es evaluada y las restantes operaciones son realizadas como de costumbre. *(iv)* Dado que Python no requiere declarar las variables a utilizar en una función, es posible utilizar la misma función para cualquier

tipo de objeto permitido, siempre que la operación realizada entre ellos tenga significado. En el ejemplo mostrado invocamos dos veces a la misma función, en un caso utilizando un valor real, en otro una cadena de caracteres. Python automáticamente detecta el tipo de dato involucrado y otorga a la operación el significado adecuado para cada caso: para constantes numéricas `*` significa multiplicación, para cadenas de caracteres, concatenación.
Veamos ahora otro ejemplo, donde computamos la velocidad de propagación de ondas en un medio elástico, a partir de los parámetros físicos que lo definen. Es sabido que conocidos el módulo de volumen (o incompresibilidad) K, el módulo de corte G y la densidad de masa ρ de un medio elástico, la velocidad de propagación de ondas sísmicas compresional v_P y de corte v_S se computan según

$$v_P = \sqrt{\frac{K + \frac{4}{3}G}{\rho}}, \quad v_S = \sqrt{\frac{G}{\rho}}.$$

Una porción de código incluyendo una función que realiza este cómputo podría ser

```
k=37e9      # Módulo de incompresibilidad del cuarzo (en N/m^2)
g=44e9      # Módulo de corte del cuarzo (en N/m^2)
r=2.65e3    # Densidad del cuarzo (en kg/m^3)

def velocs(mod_b,mod_g,rho):
      from math import sqrt
      v_p=sqrt((mod_b+4*mod_g/3)/rho)
      v_s=sqrt(mod_g/rho)
      return v_p,v_s

vp,vs=velocs(k, g, r)
print(f'La velocidad de onda compresional v_p es {vp:4.2f} m/s')
print(f'La velocidad de onda de corte v_s es {vs:4.2f} m/s')
```

La ejecución de este código produce como salida

```
La velocidad de onda compresional v_p es 6008.38 m/s
La velocidad de onda de corte v_s es 4074.77 m/s
```

Vemos que la función `velocs` devuelve simultáneamente las dos velocidades. Lo hace, como hemos dicho, en una tupla. De la misma manera, al invocar la función, recuperamos ambos resultados a través de otra tupla. Debe prestarse atención en el orden de sus elementos, para no cometer errores mezclando variables. Otro elemento a notar es la forma en que las variables `vp` y `vs` son impresas en la pantalla. Nos referiremos a ello con más detalle en la Sección 1.10; ahora mencionamos que estamos dando formato a sus valores, indicando que tienen que tomar 4 lugares enteros y 2 lugares después del punto decimal.

Accediendo a funciones

Hemos dicho que una buena práctica de programación es la modularización; el empleo de funciones aporta en esa dirección.
Ahora, ¿dónde debemos definir las funciones y cómo accedemos a ellas? La respuesta es que puede hacerse de diferentes maneras. Una opción es definir la función dentro del código que estamos desarrollando; en ese caso pasa a estar disponible para ser invocada a partir de allí. Otra opción es escribir la función archivo separado, e importarlo como módulo en el código que requiera su uso. Esta forma es la recomendada, puesto que nos permite invocar a la función desde distintos programas en forma directa. Siguiendo con el ejemplo de la sección anterior, suponemos que

hemos guardado la función `pordos()` en un archivo, de nombre `misfunciones.py`. Importamos el módulo, previamente indicando a Python la ubicación del mismo:

```
import sys
sys.path.append('/camino/a/carpeta/conteniendo/el/módulo')
import misfunciones as mf
```

Estamos entonces en condiciones de utilizar la función, por ejemplo imprimiendo su evaluación, `print(mf.pordos(4.))`. Nótese la diferencia en el uso de la función respecto del ejemplo anterior; al importarla desde un módulo debemos anteponer a su nombre el alias que elegimos para el mismo.
Podemos guardar las funciones que creemos de a una por archivo, o agruparlas según nuestra conveniencia, todas en un mismo archivo, o en distintos grupos en distintos archivos. En estos dos casos, si no vamos a utilizar en el código todas las funciones que el/los módulos contienen, es conveniente utilizar la última de las opciones de importación que vimos en la Sección 1.5.4, para no reservar espacio en la memoria de la computadora que no va a ser utilizado.

1.9.1. Alcance de las variables

En Python, entendemos por alcance de una variable a la región de un código desde donde la misma es accesible. Valen las siguientes reglas:

(a) Variables definidas dentro de una función sólo son visibles al código dentro de ella. Se dice que son *variables locales.* No es posible utilizarlas afuera de la función; invocarlas genera un error.

```
def ej_alcance_local_1():
      x = 10 # Variable local
      print(f"Dentro de la función, x = {x}")

ej_alcance_local_1()     ⟵ Devuelve el valor 10.
print(x)                 ⟵ Devuelve error.
```

(b) Variables definidas dentro de una función no colisionan con variables definidas fuera de ella, aún cuando sean utilizados los mismos nombres. El nombre `x` asignado fuera de una dada función es una variable completamente diferente de la variable `x` asignada dentro de ella.

```
x = 20

def ej_alcance_local_2():
      x = 10 # Variable local
      print(f"Dentro de la función, x = {x}")

ej_alcance_local_2()                        ⟵ Devuelve el valor 10.
print(f"Fuera de la función, x = {x}")      ⟵ Devuelve el valor 20.
```

(c) Si una variable es asignada fuera de una dada función, es una *variable global* para ella.

```
x = 20

def ej_alcance_global():
      print(f"Dentro de la función, x = {x}")
```

```
ej_alcance_global()          ⟵ Devuelve el valor 20.
```

Nótese que a diferencia del caso anterior, la variable `x` no es asignada dentro de la función, por lo que es "heredada" desde el módulo que invoca a la función.

(d) Generalizando el item (c), si una variable es asignada fuera de toda función, es *global* a todo el módulo, y sólo a ese módulo. Por ejemplo, importando el módulo `numpy`, hacemos accesible el valor de π a todas las funciones que definamos en el mismo archivo. Pero si importamos también otro módulo, donde queremos utilizar a π sin haber importado en él a `numpy`, el programa devuelve error.

(e) Sea `x` una variable que está declarada en una función y simultáneamente en una función anidada dentro de ella. Es posible que cambios realizados a `x` desde la función más interna también afecten a variable en la función exterior, si utilizamos el atributo `nonlocal` al declarar a la variable en la primera. Por ejemplo, la porción de código

```
def func_ext():
      x = 20
      def func_int():
           nonlocal x
           x=30
           print(f"Dentro de la función interna, x = {x}")
      func_int()
      print(f"Dentro de la función externa, x = {x}")
func_ext()
```

produce la salida

```
Dentro de la función interna, x = 30.
Dentro de la función externa, x = 30.
```

(f) Presentamos finalmente el atributo `global`. Si `x` es una variable que está simultánemente definida en un módulo y en una función definida dentro de él, utilizar este atributo al declarar a la variable dentro de la función hace que cambios a `x` dentro de la misma, afecten a la variable externa. Veamos:

```
x = 50

def cambio_variable_global():
      global x
      x = 100
      print(f"Dentro de la función, x = {x}")

cambio_variable_global()
print(f"Fuera de la función, x = {x}")
```

La ejecución de este código produce la salida

```
Dentro de la función, x = 100
Fuera de la función, x = 100
```

1.9.2. Atención!: Arreglos

Hemos visto que las listas y los arreglos de `numpy` son objetos que comparten la propiedad de ser mutables. Esto hace que tengan un comportamiento distinto al descripto para objetos inmutables (que, como hemos dicho, son los números enteros, reales y complejos, CDCs y las tuplas) cuando son utilizados como argumentos de entrada y salida en funciones, o son modificadas dentro de ellas.
En Python todo cambio realizado dentro de una función a un arreglo que (a) es pasado por un argumento, o (b) es heredado desde el módulo que contiene a la función en cuestión, modifica al arreglo exterior. Veamos un ejemplo para estos casos:

```
import numpy as np
a=np.array([1.,2.,3.])
b=np.array([4.,5.,6.])

def reemplazo(x):
      x[0]=100.
      b[1]=200.
      return x

print(reemplazo(a))       ⟵ Devuelve el arreglo [100.,2.,3.]
print(b)                  ⟵ Devuelve el arreglo [4.,200.,6.]
```

Vemos entonces que, a diferencia de lo que vimos más arriba para variables inmutables, modificar a la variable de entrada dentro de la función, o a una variable "heredada", modifica también a la variable externa.

Si no se desea este comportamiento, deberá hacerse una copia independiente del arreglo mediante el método `copy()` antes de hacer modificación alguna, y realizarlas sobre la copia. Veamos un ejemplo:

```
import numpy as np
a=np.array([1.,2.,3.])

def reemplazo(x):
      y=x.copy()
      y[0]=100.
      return y

print(reemplazo(a))     ⟵ Devuelve el arreglo [100.,2.,3.]
print(a)                ⟵ Devuelve el arreglo [1.,2.,3.]
```

Vemos que el arreglo `a`, pasado como argumento de ingreso a la función, no es modificado en el módulo que la invoca.

1.9.3. Funciones lambda o anónimas

Este tipo de funciones es una alternativa a la instrucción `def` para definir una función en Python. Se definen de la siguiente manera:

```
nombre_opcional = lambda variable_1, ..., variable_n: expresión utilizando las variables
```

Las funciones creadas de esta manera son utilizadas de igual forma que las creadas con `defs`, aunque hay algunas diferencias que debemos conocer: (a) Tal como vemos en su definición, las funciones `lambda` pueden opcionalmente no tener nombre. (b) el cuerpo de las funciones lambda es siempre una única expresión; no puede ser un bloque de instrucciones. Veamos un ejemplo:

```
h = lambda x,y,z: x+y+z
print(h(2,3,4))  ⟵ Devuelve el valor 9
```

Un uso de la versión anónima se muestra en el siguiente ejemplo:

```
L=[lambda x: x**2, lambda x: x**3, lambda x: x**4]

for y in L:
     print(y(2))   ⟵Devuelve los valores 4,8,16

print(L[2](4))  ⟵Devuelve el valor 256.
```

1.9.4. Funciones como argumento de funciones

Cuando estudiemos diversos métodos del análisis numérico, veremos que para programarlos resulta útil poder pasar funciones como argumento de otras funciones. En Python es posible hacerlo sin complicación alguna; consideremos el siguiente código:

```
def suma(x,y):
      return x+y

def resta(x,y):
      return x-y

def mifuncion(func,x,y):
      return func(x,y)

a=10.
b=5.

print(févaluo con suma, resultado es {mifuncion(suma,a,b)}')
print(févaluo con resta, resultado es {mifuncion(resta,a,b)}')
print(févaluo con producto, resultado es {mifuncion(lambda x,y: x*y,a,b)}')
```

En este ejemplo, definimos dos funciones, `suma` y `resta`. La tercera, `mifuncion`, es la que acepta a una función como argumento de entrada, a través de la variable `func`. Debe notarse que en el listado de argumentos de entrada de `mifuncion` no existe diferencia alguna respecto del tipo de objeto que será asignado a cada uno al momento de invocar a la función. Que `func` es una función, se expresa en el bloque de `mifuncion`, puesto que el valor que retorna es precisamente la evaluación `func(x,y)`.
A continuación se imprime en forma sucesiva el llamado a `mifuncion` con las funciones que definimos previamente pasadas como argumento.
En el tercer `print` presentamos una forma alternativa de pasar una función como argumento, puesto que invocamos a `mifuncion` definiéndola en el momento del llamado, a través de una función anónima. Esta opción es útil cuando la función argumento es adaptable a este formato. Por supuesto, la implementación de este código imprime como resultado `15.0, 5.0, 50.0` respectivamente dentro de las expresiones dadas.

1.10. Imprimiendo información

Comenzaremos esta sección dando algunos elementos de la forma de uso de la función `print()`, a la que hemos utilizado en ejemplos a lo largo del apunte.
Si bien esta función imprime por defecto en el *flujo de salida estándar, o stdout*, que usualmente no es más que la pantalla o entorno desde donde estamos ejecutando nuestro código, también puede ser utilizada para escribir en archivos. Es llamada utilizando la siguiente sintaxis:

```
print([objeto, ...][, sep=''][, end='\n'][, file=sys.stdout][, flush=False])
```

En esta expresión, los corchetes indican que el elemento que encierran es de uso optativo; los elementos después de los '='son los dados por defecto. Entonces, en el caso mostrado, o si no están presentes, los objetos a imprimir estarán separados entre sí por espacios en blanco (como consecuencia del uso de `sep`), la impresión finaliza con una línea nueva (por el uso del `end`), y la salida está dirigida, como ya dijimos, a `stdout`. Finalmente, `flush` indica, cuando se está escribiendo a archivos, si el contenido debe volcarse inmediatamente (opción `True`), o si puede hacerse esporádicamente, según conveniencia del sistema operativo (opción `False` por defecto). Ciertamente, es posible modificar los valores preestablecidos según nuestra conveniencia. Por ejemplo,

```
x=44.5
y=4+5j
z=ÁNyP'
print(x,y,z,sep='...',end='$')
```

produce la salida

```
44.5...(4+5j)...ANyP$
```

Debe notarse que un siguiente llamado a `print` imprimirá la correspondiente salida en la misma línea. Esto es consecuencia del cambio de la opción `end`, en donde suprimimos la instrucción de línea nueva, `'\n'`.
Nos referiremos ahora a las cadenas de caracteres con formato, o `f-strings`, que son uno de los objetos más utilizados como argumento de la función `print()`.

f-strings

En algunos de los ejemplos que hemos visto con anterioridad mostramos que las CDCs con formato son una de las posibilidades -probablemente la más práctica- que provee Python de incorporar valores de variables, resultados de expresiones o funciones a cadenas de caracteres, a CDCs, facilitando su posterior impresión. Su forma general es

```
f'CDC {variable o función:formato}'
```

En cuanto a sintaxis, se puede comenzar indistintamente con `f` o `F`, y como siempre, las comillas pueden ser simples o dobles; entre ellas puede haber más de una CDC y más de una expresión evaluada. La f-string es una CDC en sí misma, de modo que además de poder imprimirse puede asignarse a una variable, valiendo para ella todos los métodos y propiedades que ya hemos estudiado para CDCs. En cuanto al formato, se pueden utilizar una variedad de modificadores, algunos de los cuales enumeramos en la siguiente tabla:
Para CDCs pueden usarse solamente los tres primeros modificadores de formato; los demás carecen de sentido. Las dadas son sólo algunas de las posibilidades para manipular las f-strings,

Símbolo	Significado	Ejemplo
<	Alinear a la izquierda	"{ábc':*<10}" ⟶ ábc*******'
>	Alinear a la derecha	"{123:>10}" ⟶ ' 123'
^	Alinear al centro	"{abc:+^10}"⟶ '+++abc++++'
0	Relleno con ceros	"{42:05}" ⟶ '00042'
+	Mostrar signo para números positivos y negativos	"{42:+}" ⟶ '+42'
-	Mostrar signo solo para números negativos	"{42:-}" ⟶ '42'
	Espacio para números positivos	"{42: }" ⟶ '42'
%	Formato de porcentaje	"{0.12377:.2%}" ⟶ '12.38%'
f	Formato de número de punto flotante	"{3.14159:.2f}"⟶'3.14'
e	Notación exponencial (minúscula)	"{12365:.2e}" ⟶ '1.24e+04'
E	Notación exponencial (mayúscula)	"{12345:.2E}" ⟶ '1.23E+04'

más sobre ellas en la bibliografía de la Cátedra.

Escritura/lectura en/desde archivos

Hemos visto que `print()` brinda la posibilidad de escribir salidas a archivos; de hecho, su uso es práctico dado que esta instrucción automáticamente convierte la salida a CDCs y aplica elementos de formato para que el archivo sea fácilmente legible por humanos. Ahora, para poder enviar nuestra salida a un archivo es necesario que previamente "conectemos" el código con el archivo. Esto se realiza utilizando la instrucción `open()`, que tiene la siguiente sintaxis:

```
[variable = ] open(ñombre_archivo'[,'status'])
```

No es mandatorio asignarle nombre al objeto que es devuelto al establecer la conexión con el archivo de nombre `nombre_archivo`, pero sí conveniente para su posterior manipulación. La entrada opcional `status` indica en qué modo estamos abriendo el archivo. Puede ser uno de los siguientes:

Modo	Descripción
'r'	Abre el archivo en modo de solo lectura. Si el archivo no existe, se produce un error. Este es el modo por defecto
'w'	Abre el archivo en modo de escritura. Si el archivo existe, se sobreescribe; si no, se crea uno nuevo.
á'	Abre el archivo en modo de agregar (append). Si el archivo no existe, se crea uno nuevo.
'r+'	Abre el archivo para lectura y escritura. Si el archivo no existe, se produce un error.
'w+'	Abre el archivo para lectura y escritura, pero sobreescribe el contenido si el archivo ya existe, o lo crea si no existe.
á+'	Abre el archivo para lectura y escritura, pero los datos se agregan al final del archivo existente o se crea uno nuevo si no existe.

Junto con estos modos puede usarse `b` (por ejemplo `'rb+'`) para indicar que el archivo debe ser escrito en formato binario, o `t`, para formato texto (este es la opción por defecto).

Existen otras entradas opcionales para la instrucción `open()`; entre ellas mencionamos una que puede resultar útil al escribir/leer en modo texto, y es `encoding`. Si no se da, la codificación del texto es elegida por el sistema operativo. Si se utilizan caracteres especiales (como los acentos en español) se recomienda emplear `encoding=útf-8'`; esto evitará incompatibilidades al procesar el archivo en diferentes computadoras con distintos sistemas operativos.

Veamos un ejemplo, en donde generamos un vector de cien números entre -$\pi/2$ y $\pi/2$ y guardamos en un archivo dos columnas, una con el vector dado y otra con las correspondientes evaluaciones de la función coseno:

```
import numpy as np

abscisas=np.arange(-np.pi/2.,np.pi/2.,np.pi/100.)

ar1=open('datos.res','w',encoding=útf-8')

print(f"{Ábscisas':<10}{Órdenadas':<10}",file=ar1)

for x in abscisas:
      print(f'{x:<10.4f}{np.cos(x):<10.4f}',file=ar1)

ar1.close()
```

Las primeras líneas del archivo datos.res son:

```
Abscisas  Ordenadas
-1.5708   0.0000
-1.5394   0.0314
-1.5080   0.0628
-1.4765   0.0941
```

Debe notarse el uso de `ar1.close()`. Todo archivo que es abierto para lectura o escritura debe ser cerrado antes que finalice el código, para evitar posibles pérdidas de datos. La sintaxis utilizada indica que `close` es un método aplicable al objeto python creado al abrir el archivo.

Existen distintas instrucciones para leer datos de archivos. Una de las posibles es utilizar el módulo `Numpy`, pues brinda una serie de instrucciones para la lectura de archivos, tanto de texto como binarios. Para los primeros, de uso frecuente son `np.loadtxt() y np.genfromtxt()`. Por defecto, Python asigna datos leídos de archivos de texto a variables de tipo CDC. Las dos instrucciones que mencionamos, en forma transparente para el usuario, las convierten a datos numéricos, de allí la conveniencia de su uso. Ambas tienen un gran número de parámetros opcionales, que permiten procesar con facilidad diferentes archivos con distintos formatos, aunque la segunda presenta más flexibilidad para procesar archivos con falta de datos o errores en los mismos. Leamos el archivo escrito en ejemplo anterior con la primera de estas funciones:

```
import numpy as np

datos = np.loadtxt("datos.res",skiprows=1)

abscisas=datos[:,0]
ordenadas=datos[:,1]
```

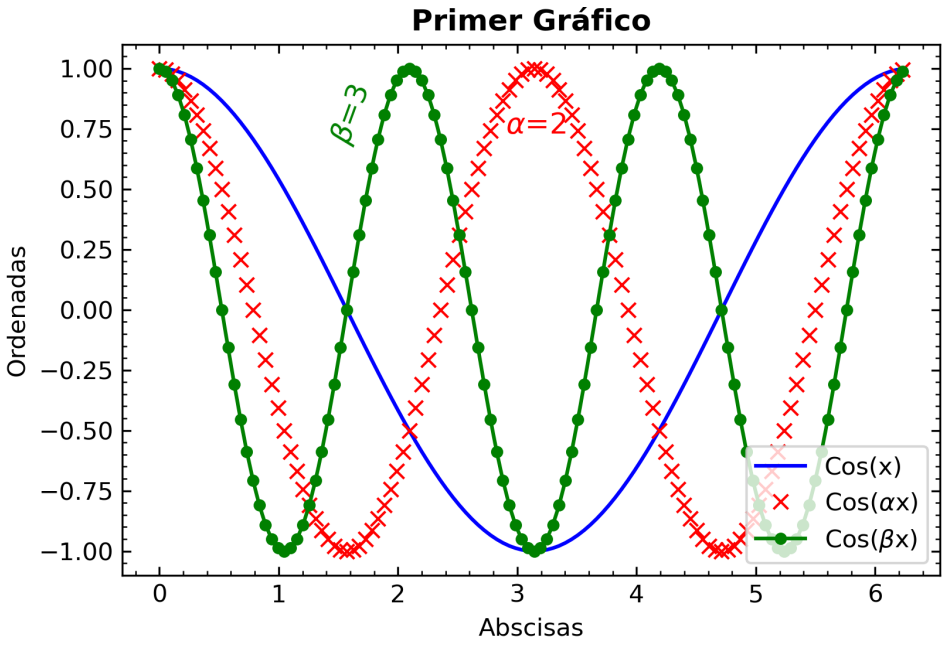
Se importa en primer término, como ya es usual, el módulo `numpy`. La siguiente instrucción, salteando la primer fila del archivo (opción `skiprows=1`), carga toda la información en la variable `datos`. Ésta, dado el archivo leído, es un arreglo bidimensional, que en la porción de código que se muestra es dividida para su posterior uso, en dos unidimensionales.

1.11. Graficando datos

Python nos brinda la posibilidad de generar gráficos con los datos resultantes de nuestros códigos, facilitando así su procesamiento y posterior interpretación. El módulo indicado para estas tareas es `matplotlib`. En particular, se recomienda el uso de un submódulo, llamado `pyplot`, que nos permite manipular en forma sencilla las diversas componentes de un gráfico. Damos un ejemplo que muestra este concepto:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0,2*np.pi,np.pi/60.)
y = np.cos(x)

plt.figure()  #Creamos el objeto que contendrá el gráfico

#Creamos curvas

plt.plot(x, y, '-', label='Cos(x)', color='blue')  #Curva 1

y = np.cos(2 * x)
plt.plot(x, y, 'x', label=r'Cos($\alpha$x)', color='red')  #Curva 2

y = np.cos(3 * x)
plt.plot(x, y, 'o-', label=r'Cos($\beta$x)', markersize=4, color='green')  #Curva 3

#Damos formato a los ejes

plt.minorticks_on()
plt.tick_params(axis='both', which='both', direction='in', top=True, right=True)

#Establecemos el lugar donde aparece la leyenda

plt.legend(loc='lower right')

#Etiquetamos ejes y curvas

plt.xlabel('Abscisas')
plt.ylabel('Ordenadas')
plt.title('Primer Gráfico', fontweight='bold')

plt.text(2.9, 0.73, r'$\alpha$=2', fontsize=12, color='red')
plt.text(1.4, 0.7, r'$\beta$=3', fontsize=12, color='green', rotation=70)

#Guardamos en un archivo

plt.savefig('primergrafico.png', bbox_inches='tight', format='png')

#Mostramos en pantalla

plt.show()
```

La ejecución de este código guarda en la misma carpeta el archivo `primergrafico.png`, que



muestra la figura 1.11. Las distintas instrucciones empleadas presentan múltiples opciones, y además, la apariencia de la figura puede modificarse de diversas formas, por ejemplo, haciendo que las etiquetas de las abscisas sean fracciones y múltiplos de π, o que las ordenadas tengan formato científico. Discutiremos sobre éstas y otras posibilidades en los trabajos prácticos.

1.12. Apéndice A

1.12.1. Palabras reservadas

Listado de palabras reservadas por Python (para la versión 3.2.9, que es la utilizada en este apunte) que no pueden ser utilizadas para nombrar objetos.

```
False   class     finally  is        return
None    continue  for      lambda    try
True    def       from     nonlocal  while
and     del       global   not       with
as      elif      if       or        yield
assert  else      import   pass
break   except    in       raise
```

Capítulo 2

Análisis Numérico

El análisis numérico consiste en desarrollar métodos (algoritmos) para la solución aproximada de problemas matemáticos que no poseen una solución analítica simple.

En este proceso, deben tenerse en cuenta las distintas fuentes de error, a fin de evitar que dichos errores invaliden los resultados. Uno de estos errores son los **errores en los datos de entrada**, cuyo control está por fuera del algoritmo. Otro error con el que hay que lidiar es el **error de redondeo**, que surge cuando los cálculos son realizados con números cuya representación queda restringida a un número finito de dígitos, que es lo que lo que ocurre en una computadora. Veamos un ejemplo.

Supongamos que queremos evaluar la función

$$f(x) = \sqrt{x^2 + 1} - 1 \tag{2.1}$$

en $x = 0{,}25$, utilizando una computadora (ficticia) que hace las cuentas con 3 cifras significativas. Mostramos cómo se halla el resultado, paso a paso:

$$\begin{aligned} f(0{,}25) &= \sqrt{0{,}25^2 + 1} - 1 \\ &= \sqrt{0{,}0625 + 1} - 1 \\ &= \sqrt{1{,}06} - 1 \\ &= 1{,}03 - 1 \\ &= \boxed{0{,}03} \end{aligned} \tag{2.2}$$

Sin embargo, el resultado "correcto" (considerando 3 cifras significativas) debería ser 0,0308, teniendo en cuenta que el resultado exacto es 0,030776.

Si en lugar de trabajar con la expresión anterior la transformamos en su equivalente

$$f(x) = \frac{x^2}{\sqrt{1 + x^2} + 1} \tag{2.3}$$

obtenemos

$$\begin{aligned} f(0{,}25) &= \frac{0{,}25^2}{\sqrt{1 + 0{,}25^2} + 1} \\ &= \frac{0{,}0625}{\sqrt{1 + 0{,}0625} + 1} \\ &= \frac{0{,}0625}{1{,}03 + 1} \end{aligned}$$

$$= \frac{0{,}0625}{2{,}03}$$
$$= \boxed{0{,}0308} \tag{2.4}$$

que está más cerca del resultado exacto.

Conclusión: cuando se utiliza aritmética de precisión finita el resultado dependerá de cómo se realicen las cuentas, aún cuando se traten de expresiones equivalentes.

* * *

Otro tipo de errores son los llamados **errores de aproximación**, que surgen cuando un determinado método produce una solución a un problema que no es la exacta. Uno de esos errores es el llamado **error de truncamiento** que aparece, por ejemplo, cuando se aproxima una serie infinita por una suma finita de términos.

En otras ocasiones, un problema es aproximado por uno similar llevando a cabo una "discretización", por ejemplo

$$\int f(x)dx \quad \to \quad \sum_i f(x_i)\Delta x_i \tag{2.5}$$

$$\frac{df}{dx} \quad \to \quad \frac{f(x_i) - f(x_{i-1})}{\Delta x} \tag{2.6}$$

En estos casos el error de aproximación se conoce como **error de discretización**. De este tipo de errores nos vamos a encargar cuando veamos los distintos métodos para aproximar soluciones.

Comenzamos ahora con el estudio del efecto del redondeo en los cálculos, para lo cual necesitamos introducir algunas ideas.

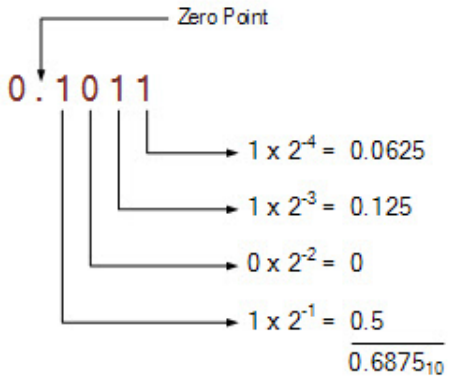
2.1. Representación de números: la notación posicional

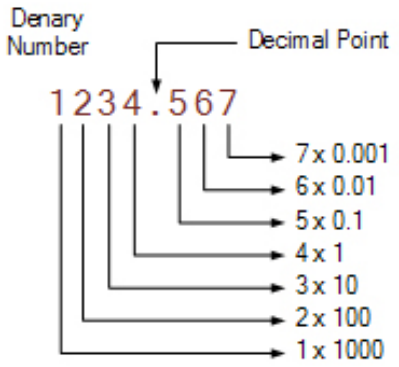
Revisemos primero la *notación posicional estándar*, en la cual el significado de cada dígito depende de su posición relativa con respecto a los demás.

Sea B cualquier entero positivo mayor que la unidad y sea d_i, $i \in \mathbb{Z}$, cualquiera de los números enteros entre 0 y $B-1$. Podemos representar entonces a cualquier real positivo como

$$d_n\, d_{n-1} \ldots d_1\, d_0\, .\, d_{-1}\, d_{-2} \ldots d_{-j} \ldots = \sum_{i=0}^{n} d_i B^i + \sum_{j=1}^{\infty} d_{-j} B^{-j}. \tag{2.7}$$

El número B se llama BASE de la representación. Nosotros estamos acostumbrados a trabajar con base $B = 10$ pero las computadoras lo hacen generalmente con base $B = 2$.





Si consideramos $B = 10$, sabemos que

$$347{,}29 = 3 \times 10^2 + 4 \times 10^1 + 7 \times 10^0 + 2 \times 10^{-1} + 9 \times 10^{-2}. \tag{2.8}$$

En cuanto al número infinito de cifras en la representación, está claro que hay casos, como éste, en donde sólo algunos de los coeficientes son distintos de cero.

Puede ocurrir que un número tenga una cantidad finita de cifras en una representación mientras que en otra requiera de infinitos dígitos. Por ejemplo,

$$B = 10 \qquad \frac{1}{10} = 0{,}1 \tag{2.9}$$

$$B = 2 \qquad \frac{1}{10} = 0{,}0001100110011... \tag{2.10}$$

por lo que 1/10 no tiene representación exacta en una computadora(!).

> Esta es una clara fuente de error al realizar operaciones con las computadoras, ya que uno le suele suministrar los datos en representación decimal y la computadora los debe convertir a representación binaria antes de comenzar a trabajar con ellos.

Por todo esto, conviene revisar cómo se realiza la conversión entre bases antes de seguir avanzando.

2.2. Conversión entre bases

Notación:

Si $x \in \mathbb{R}$, llamaremos $\phi_B(x)$ a su representación en la base B y $\phi_D(x)$ a la correspondiente a la base D. (Nota: usaremos notación decimal para describir a x, B y D.)

Normalmente el procedimiento de conversión se desdobla, tratándose las partes entera y fraccionaria por separado

$$x = N.F \tag{2.11}$$

Consideremos la conversión de la parte entera (N) desde la base D a la base B. Un número entero puede representarse en ambas bases de dos formas equivalentes:

$$N = \sum_{i=0}^{r} a_i D^i$$

$$= \sum_{j=0}^{s} c_j B^j \tag{2.12}$$

Suponemos los a_i conocidos, mientras que los c_j son las incógnitas. Dividimos N por la base a la que deseamos convertir:

$$\frac{N}{B} = \underbrace{\sum_{j=1}^{s} c_j B^{j-1}}_{=N_1} + \frac{c_0}{B} \tag{2.13}$$

En esta expresión, la sumatoria contiene la parte entera del cociente (N_1), mientras que c_0/B constituye la parte fraccionaria, en la cual c_0 es el resto.

Repetimos el proceso, ahora para N_1

$$\frac{N_1}{B} = \underbrace{\sum_{j=2}^{s} c_j B^{j-1}}_{=N_2} + \frac{c_1}{B} \tag{2.14}$$

y así se van obteniendo c_0, c_1,..., c_s.

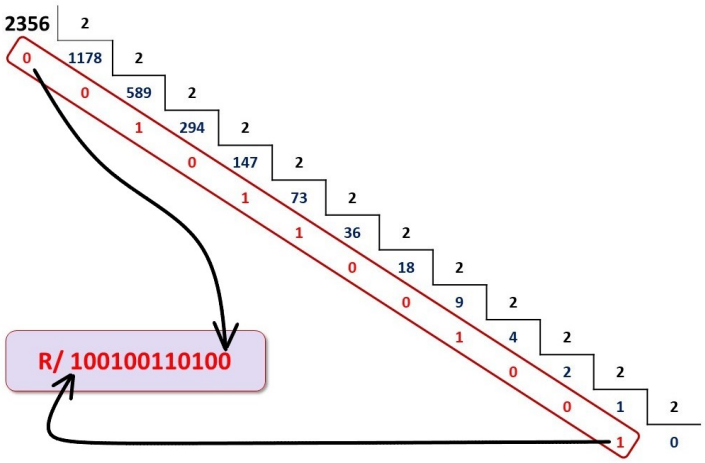


Figura 2.1: Conversión de la parte entera (de decimal a binario)

Para convertir la parte fraccionaria (F) de la base D a la base B escribimos primero su representación en ambas bases

$$F = \sum_{i=1}^{\infty} f_{-i} D^{-i}$$

$$= \sum_{j=1}^{\infty} g_{-j} B^{-j} \tag{2.15}$$

donde consideramos a los f_{-i} como dato y nos interesa hallar los g_{-j}, que son las incógnitas.

Multiplicando F por B

$$BF = g_{-1} + \underbrace{\sum_{j=2}^{\infty} g_{-j} B^{-j+1}}_{=F_1} \tag{2.16}$$

nos queda g_{-1} como la parte entera y el resto la parte fraccionaria (F_1). Podemos repetir el proceso, ahora para F_1

$$BF_1 = g_{-2} + \underbrace{\sum_{j=3}^{\infty} g_{-j} B^{-j+2}}_{=F_2} \tag{2.17}$$

donde g_{-2} es la parte entera y lo demás, la parte fraccionaria. Repitiendo el proceso obtenemos los demás coeficientes.

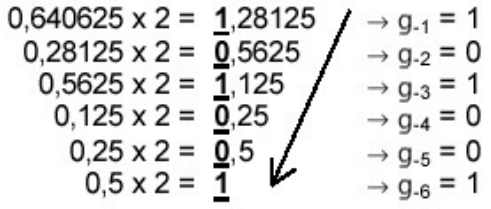


Figura 2.2: Conversión de la parte fraccionaria (de decimal a binario)

(En clase practicaremos el procedimiento convirtiendo dos números, $\phi_{10}(75{,}8)$ y $\phi_{10}(0{,}1)$, a base 2.)

Cuando convertimos desde un sistema con base $D \neq 10$ a la decimal conviene, en vez de utilizar este algoritmo, usar directamente la notación posicional. Por ejemplo, si deseáramos convertir $\phi_2(x) = 1110{,}101$ a decimal $\phi_{10}(x)$, haríamos directamente lo siguiente:

$$\begin{aligned} \phi_{10}(x) &= 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 8 + 4 + 2 + \frac{1}{2} + \frac{1}{8} \\ &= 14{,}625 \end{aligned} \tag{2.18}$$

2.3. Representación de números enteros en una computadora binaria

Aunque en general puede elegirse una base B cualquiera, vamos a trabajara con $B = 2$ que es la que usan las computadoras. Consideremos que disponemos de k dígitos con los que deseamos representar números (enteros) positivos y negativos utilizando las 2^k posibilidades que tenemos. Existen 4 métodos distintos para hacerlo, pero nosotros sólo vamos a ver dos.

2.3.1. Signo y magnitud

En este método se toman los $k-1$ dígitos menos significativos para guardar el valor absoluto del número en cuestión y el bit más significativo se preserva para el signo (0: signo +; 1: signo −).
Ejemplo ($k=3$): en este caso tenemos 2 lugares para el valor absoluto.

000	(0)	100	(−0)
001	(1)	101	(−1)
010	(2)	110	(−2)
011	(3)	111	(−3)

Es decir, podemos representar 7 enteros distintos, desde -3 hasta 3, aunque hay dos formas de representar el cero. Esto puede considerarse una desventaja del método.

2.3.2. Representación en exceso N (o sesgada)

En esta representación se utiliza un número N preestablecido como sesgo. Luego, un valor es representado por el número sin signo que es mayor por N que el valor a obtener.

Por ejemplo, si $N=3$, la representación binaria del cero corresponde a -3, de 1 a -2 y así siguiendo:

0	000	(−3)
1	001	(−2)
2	010	(−1)
3	011	(0)

Esta representación de enteros se utiliza para codificar el exponente de números de punto flotante.

2.4. Representación de punto flotante (números reales)

Las cantidades fraccionarias se representan en la computadora en la forma de punto flotante. Como se dijo anteriormente, las computadoras usan alguna forma de representación binaria para estos números; nosotros vamos a trabajar alternativamente con la base 10 por una cuestión de simplicidad.

Un número N en representación de punto flotante se escribe como

$$N = \pm a \times 10^b \tag{2.19}$$

donde a es el *significando*, b el *exponente* ($b \in \mathbb{Z}$) y 10 la base.

Consideremos el número 156,78. En notación de punto flotante este número se representa como

$$0{,}15678 \times 10^3 \tag{2.20}$$

aunque, evidentemente, podría haberse escrito también como

$$0{,}0015678 \times 10^5. \tag{2.21}$$

Sin embargo, esta última elección queda descartada porque los ceros de más ocupan lugares inútilmente. Así, se conviene utilizar una representación *normalizada*, para la cual se verifica que

$$\frac{1}{10} \leq a < 1. \tag{2.22}$$

La estructura de un número de punto flotante en una computadora es la siguiente:

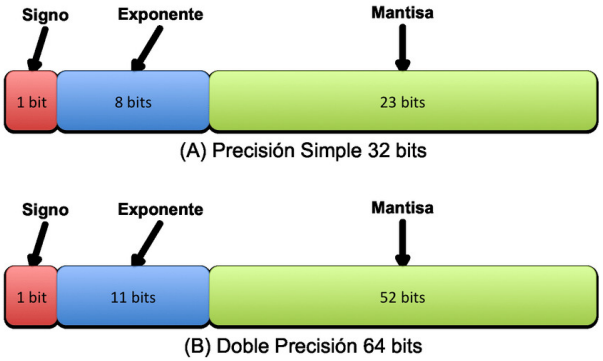


Figura 2.3: Representación de acuerdo con la norma IEEE 754

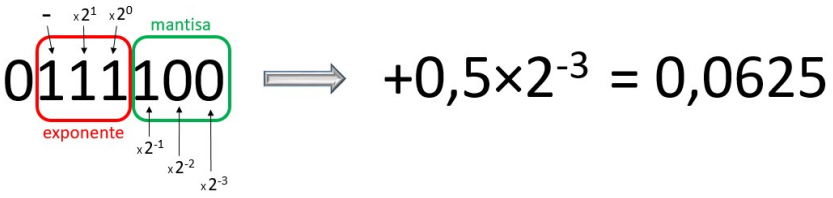
Donde uno establece de antemano el campo total (cantidad de bytes) que va a destinarse para cada número real.

Para fijar ideas, supongamos que se destinan 7 bits por cada número real: 1 bit para el signo, 3 bits para el exponente y 3 bits para el significando.

```
 sgn    e1    e2    e3    s1    s2    s3
|----|----|----|----|----|----|----|
 bit
```

Para facilitar la descripción supondremos que el exponente se representa mediante el método de signo y magnitud. Por lo tanto, con 3 bits para el exponente podremos representar $2^3 - 1$ $(= 7)$ números enteros (desde -3 hasta 3). Además, sólo trataremos los números positivos (los negativos se obtienen de forma análoga).

Analicemos cuál es el número más pequeño que podemos representar. Dicho número corresponderá al de exponente más pequeño (-3) y menor significando, compatible con la normalización (100). Por lo tanto, en notación binaria, tendremos que el número más pequeño representable (en módulo) será



Los siguientes valores serán

$$0111101 \quad \rightarrow \quad 0{,}078125 \tag{2.23}$$

$$0111110 \quad \rightarrow \quad 0{,}093750 \tag{2.24}$$

$$0111111 \quad \rightarrow \quad 0{,}109375 \tag{2.25}$$

La diferencia entre números consecutivos es de 0,015625, que es lo que aporta un cambio de una unidad en el bit menos significativo, cuando el exponente es el menor posible ($1 \times 2^{-3} \times 2^{-3} = 2^{-6} = 0{,}015625$).

Para seguir generando los demás valores representables, hay que incrementar en uno el exponente. Así, los números que continúan son

$$0110100 \quad \rightarrow \quad 0{,}125000 \tag{2.26}$$

$$0110101 \quad \rightarrow \quad 0{,}156250 \tag{2.27}$$

$$0110110 \quad \rightarrow \quad 0{,}187500 \tag{2.28}$$

$$0110111 \quad \rightarrow \quad 0{,}218750 \tag{2.29}$$

Ahora la diferencia entre números estará dada por el cambio del bit menos significativo cuando el exponente es (-2), es decir, $1 \times 2^{-3} \times 2^{-2} = 2^{-5} = 0{,}03125$. Como se ve, ahora el salto entre números consecutivos aumenta al doble.

Este patrón se repite hasta alcanzar el exponente máximo. Para él, los números consecutivos serán

$$0011100 \quad \rightarrow \quad 4 \tag{2.30}$$

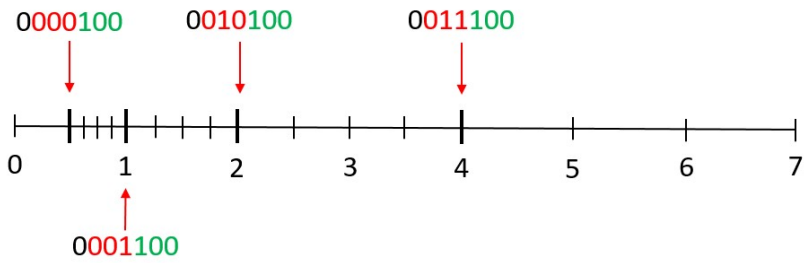
$$0011101 \quad \rightarrow \quad 5 \tag{2.31}$$

$$0011110 \quad \rightarrow \quad 6 \tag{2.32}$$

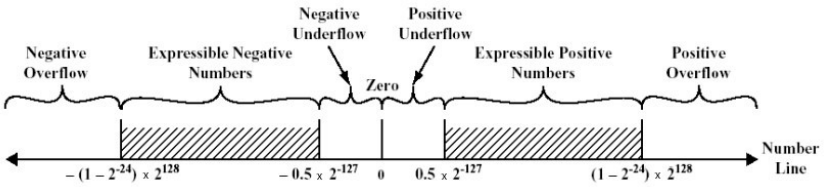
$$0011111 \quad \rightarrow \quad 7 \tag{2.33}$$

2.4.1. Características generales de la representación de punto flotante

- El conjunto de números representable es finito.
- El cero no está incluido, si nos atenemos a la normalización.
- El intervalo del eje real que se representa es acotado.
- La separación entre números aumenta al aumentar su valor absoluto. Esta característica es la que permite que la representación de punto flotante conserve los dígitos significativos. Esquemáticamente, el muestreo de la recta real será:



- Si, como resultado de un cálculo, obtenemos un valor por encima del máximo representable $\rightarrow$ *overflow*. (El máximo valor representable con reales de 4 bytes es de $\approx 3{,}4 \times 10^{38}$; con reales de 8 bytes $\approx 1{,}8 \times 10^{308}$.) Si obtenemos un valor por debajo del mínimo representable (en módulo) $\rightarrow$ *underflow*. Gráficamente, para la representación de 32 bits tenemos:



- En la figura 2.3 se muestra la distribución de bits en reales de 32 y 64 bits (4 y 8 bytes, respectivamente), según la norma IEEE 754. Al duplicar la cantidad de bits quien se lleva la mayor parte es el significando. (¿por qué?)

Como hemos visto, no todos los números reales tienen una representación exacta dentro de este esquema. Esto provoca que, en general, debamos trabajar con valores aproximados cuando realizamos cálculos con una computadora. Para decidir qué valor aproximado elegir, el esquema que se suele utilizar (aunque no siempre) es el de redondeo. A continuación estudiaremos el error de redondeo en la aritmética de punto flotante.

2.4.2. Errores de redondeo y aritmética de punto flotante

Sea $x \in \mathbb{R}$, $x \notin \mathcal{A}$ ($\mathcal{A}$: conjunto de números representables en punto flotante en una computadora). Sea $\mathrm{RD}(x)$ el número al cual se aproxima x y que está en $\mathcal{A}$. Lo que, en general, se le pide a $\mathrm{RD}(x)$ es que verifique

$$|x - \mathrm{RD}(x)| \leq |x - y| \qquad \forall y \in \mathcal{A}, \tag{2.34}$$

es decir, el valor de $\mathrm{RD}(x)$ es, de todos los números de $\mathcal{A}$, el más próximo.

En general, para obtener $\mathrm{RD}(x)$ de un número real cualquiera

$$x = \pm a \times 10^b, \quad \text{con} \quad a = 0.\alpha_1\alpha_2...\alpha_i\alpha_{i+1}... \quad (0 \leq \alpha_i \leq 9, \alpha_1 \neq 0) \tag{2.35}$$

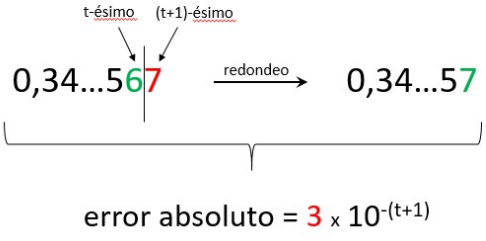
en una computadora (decimal) de t dígitos hacemos

$$\mathrm{RD}(x) = \pm a' \times 10^b \tag{2.36}$$

donde

$$a' = \begin{cases} 0.\alpha_1\alpha_2...\alpha_t & \text{si} \quad 0 \leq \alpha_{t+1} < 5 \\ 0.\alpha_1\alpha_2...\alpha_t + 1 & \text{si} \quad 5 \leq \alpha_{t+1} \leq 9 \end{cases} \tag{2.37}$$

Esquemáticamente,



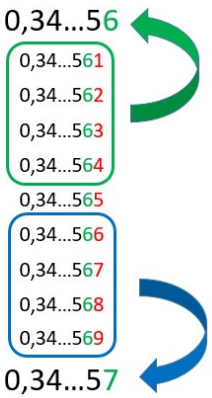


Figura 2.4: Redondeo y error absoluto asociado.

¿Qué número producirá el mayor error de redondeo en el esquema anterior? ¿De cuánto sería?

MALA SUERTE. ¿Podría ocurrir que el número redondeado no perteneciera a $\mathcal{A}$? Técnicamente, sí. Por ejemplo, si tuviéramos que representar el número $x = 0{,}99997 \times 10^{99}$ en una computadora con 2 dígitos para el exponente y el valor redondeado fuera $\mathrm{RD}(x)= 0{,}10 \times 10^{100}$, éste no sería representable y se daría una condición de *overflow*.

Si bien estos casos existen, se presentan esporádicamente, por lo que vamos a suponer que $\mathrm{RD}(x)$ siempre pertenece a $\mathcal{A}$.

Epsilon de la máquina

Resulta importante conocer una cota del error **relativo** que contiene $\mathrm{RD}(x)$. Para ello, si tenemos presente que $|a| \geq 10^{-1}$, podemos hacer la siguiente estimación

$$|\varepsilon| = \frac{|x - \mathrm{RD}(x)|}{|x|} = \frac{|a - a'| \times \cancel{10^{b}}}{|a| \times \cancel{10^{b}}} \leq \frac{5 \times 10^{-(t+\cancel{1})}}{\cancel{10^{-1}}} = 5 \times 10^{-t} = \mathrm{eps} \tag{2.38}$$

Esta cantidad se denomina el *epsilon de la máquina* (eps) y caracteriza el error *relativo* que se comete con el redondeo. Así, se cumple que

$$\mathrm{RD}(x) = x(1 + \varepsilon), \qquad |\varepsilon| \leq \mathrm{eps} \tag{2.39}$$

Una consecuencia de su definición es que el epsilon de la máquina es el número más pequeño que, sumado a uno, da un número mayor que uno.

* * *

Un problema importante que viene de la mano de la representación de punto flotante es que, aún cuando operemos con números x, $y \in \mathcal{A}$, el resultado de las operaciones aritméticas (*,/,+,-), en general, no pertenecerá a $\mathcal{A}$. Para explicitar este hecho, denotaremos las operaciones aritméticas realizadas con error de redondeo con símbolos especiales. Así, tendremos que

$$x \otimes y = \mathrm{RD}(x * y),$$

$$x \oslash y = \mathrm{RD}(x / y),$$

$$x \oplus y = \mathrm{RD}(x + y),$$

$$x \ominus y = \mathrm{RD}(x - y),$$

donde hemos reservado la simbología habitual para referirnos a la operación "exacta", esto es, sin error de redondeo. De acuerdo con lo visto más arriba, podemos escribir estas expresiones como

$$x \otimes y = (x * y)(1 + \varepsilon_1),$$

$$x \oslash y = (x / y)(1 + \varepsilon_2),$$

$$x \oplus y = (x + y)(1 + \varepsilon_3),$$

$$x \ominus y = (x - y)(1 + \varepsilon_4),$$

donde $|\varepsilon_i| \leq \mathrm{eps}$, $i = 1, 2, 3, 4$. Las operaciones de punto flotante no satisfacen propiedades básicas a las que estamos muy acostumbrados. Así, ahora resulta que

$$x \otimes (y \otimes z) \neq (x \otimes y) \otimes z,$$

$$x \oplus (y \oplus z) \neq (x \oplus y) \oplus z,$$

$$x \otimes (y \oplus z) \neq x \otimes y \oplus x \otimes z.$$

2.4.3. Ejemplos de operaciones en aritmética de punto flotante

Consideremos una computadora con un significando de 4 dígitos y un exponente de un dígito en base 10.

Para sumar y restar debemos tener en cuenta que, si los exponentes de los números difieren, el de menor exponente será modificado hasta coincidir con el del mayor (ajustando el significando consecuentemente):

$$\begin{array}{rr} & 0{,}1557 \times 10^{1} \\ + & 0{,}4381 \times 10^{-1} \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{rr} & 0{,}1557 \times 10^{1} \\ + & 0{,}0044 \times 10^{1} \\ \hline & 0{,}1601 \times 10^{1} \end{array} \tag{2.40}$$

El valor exacto es $0{,}160081 \times 10^{1}$. Esta forma de realizar la suma puede provocar lo siguiente:

$$\begin{array}{rr} & 0{,}4000 \times 10^{4} \\ + & 0{,}1000 \times 10^{-3} \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{rr} & 0{,}4000 \times 10^{4} \\ + & 0{,}0000 \times 10^{4} \\ \hline & 0{,}4000 \times 10^{4} \end{array} \tag{2.41}$$

que implica que, si dos números difieren en varios órdenes de magnitud, el número menor no aporte a la suma.

Veamos un ejemplo de sustracción entre números de igual exponente

$$\begin{array}{rr} & 0{,}3641 \times 10^{2} \\ - & 0{,}2686 \times 10^{2} \\ \hline & 0{,}0955 \times 10^{2} \end{array} \quad \rightarrow \quad 0{,}955\boxed{?} \times 10^{1} \tag{2.42}$$

¿qué valor se espera que tenga el dígito recuadrado?

De especial interés es el caso en el que la resta se da entre números muy parecidos

$$\begin{array}{rr} & 0{,}7642 \times 10^{5} \\ - & 0{,}7641 \times 10^{5} \\ \hline & 0{,}0001 \times 10^{5} \end{array} \quad \rightarrow \quad 0{,}1\boxed{?}\boxed{?}\boxed{?} \times 10^{2} \tag{2.43}$$

El problema en este caso es que se pierden los dígitos más significativos, con lo cual pasan a serlo números que (pueden) provenir de cálculos previos, con errores de redondeo. A este fenómenos se denomina "cancelación", y es una de las fuentes más peligrosas de errores numéricos.

En cuanto al producto, como la multiplicación de dos significandos de n dígitos da por resultado uno de $2n$ (dígitos), muchas computadoras ofrecen la posibilidad de guardar resultados intermedios con mayor precisión para luego redondear.

$$\begin{array}{rr} & 0{,}1363 \times 10^{3} \\ \times & 0{,}6423 \times 10^{-1} \\ \hline & 0{,}08754549 \times 10^{2} \end{array} \quad \rightarrow \quad 0{,}8755 \times 10^{1} \tag{2.44}$$

La multiplicación no suele generar problemas, siempre que no nos salgamos del rango representable (casos de *underflow* y *overflow*).

2.5. Breve introducción a la teoría de errores

2.5.1. Definiciones preliminares

Dado un número x, que supondremos exacto (aunque desconocido), y otro número $\widetilde{x}$, que será una aproximación a x, llamaremos *error absoluto* de $\widetilde{x}$ a

$$\boxed{\Delta = \widetilde{x} - x.} \tag{2.45}$$

Por su parte, el *error relativo* de $\widetilde{x}$ estará dado por

$$\boxed{\varepsilon = \frac{\widetilde{x} - x}{x}.} \tag{2.46}$$

Por completitud, definimos también el *error relativo porcentual*:

$$\delta = 100 \cdot \varepsilon. \tag{2.47}$$

Como, en general, no se conoce x, no se pueden calcular Δ y ε en forma exacta y nos tenemos que contentar con establecer cotas para los errores.

Def. Llamamos *cota superior del error absoluto* Δ de $\widetilde{x}$ a todo número real positivo Δ^* que verifique

$$|\Delta| \leq \Delta^*. \tag{2.48}$$

De aquí resulta que $\widetilde{x} - \Delta^* \leq x \leq \widetilde{x} + \Delta^*$, lo que suele denotarse como

$$x = \widetilde{x} \pm \Delta^*. \tag{2.49}$$

Def. Llamamos *cota superior del error relativo* ε de $\widetilde{x}$ a todo número real positivo ε^* tal que

$$|\varepsilon| \leq \varepsilon^*. \tag{2.50}$$

2.5.2. Relación entre el error relativo y el número de cifras significativas de un número aproximado

Decimos que el número $\widetilde{x}$ aproxima a x con k cifras significativas si

$$|\varepsilon| \leq 5 \times 10^{-k}, \tag{2.51}$$

donde k es el **mayor** entero positivo para el que se verifica esta desigualdad.

Ejemplo. Supongamos que hemos obtenido como resultado de una cuenta

$$0{,}127439855 \times 10^5 \tag{2.52}$$

con $|\varepsilon| \leq 10^{-4}$, entonces, el número de cifras confiables, de acuerdo a la expresión anterior, será:

$$|10^{-4}| \leq 5 \times 10^{-(k=4)}. \tag{2.53}$$

Por lo tanto, del número original serán útiles sólo las primeras 4 cifras

$$0{,}1274\,|\,\cancel{39855} \times 10^5 \tag{2.54}$$

2.6. Propagación de errores (sin incluir redondeo)

Consideremos los datos de entrada x que producirán a través de la aplicación de una función φ resultados y, es decir,

$$y = \varphi(x). \tag{2.55}$$

Para darle generalidad al problema, supondremos que

$$\begin{aligned} x &= (x_1, x_2, ..., x_n) \\ y &= (y_1, y_2, ..., y_m) \\ \varphi &: D \in \mathbb{R}^n \to \mathbb{R}^m \end{aligned} \tag{2.56}$$

es decir, que φ es una función vectorial de m componentes, donde

$$y_i = \varphi_i(x_1, x_2, ..., x_n), \qquad i = 1, 2, ..., m. \tag{2.57}$$

La idea es estudiar cómo se propagan los errores presentes en los datos de entrada. Por el momento supondremos que no existen los errores de redondeo, es decir, que nuestra computadora tiene **precisión infinita**. También supondremos que las funciones φ_i tienen derivadas primeras continuas en D y, de acuerdo con las definiciones previas, denotaremos

$$\Delta x_i = \widetilde{x}_i - x_i, \qquad i = 1, 2, ..., n \tag{2.58}$$

a los errores absolutos de las componentes del vector de datos de entrada, que podemos juntar en un vector de errores absolutos

$$\Delta x = \widetilde{x} - x. \tag{2.59}$$

En forma similar, los errores relativos serán

$$\varepsilon_{x_i} = \frac{\widetilde{x}_i - x_i}{x_i}, \qquad \text{si} \quad x_i \neq 0. \tag{2.60}$$

Si los datos de entrada contienen errores, en lugar de suministrarle a φ valores exactos x ingresaremos otros que son aproximados $\widetilde{x}$ y, consecuentemente, la salida reflejará esta diferencia

$$\widetilde{y} = \varphi(\widetilde{x}). \tag{2.61}$$

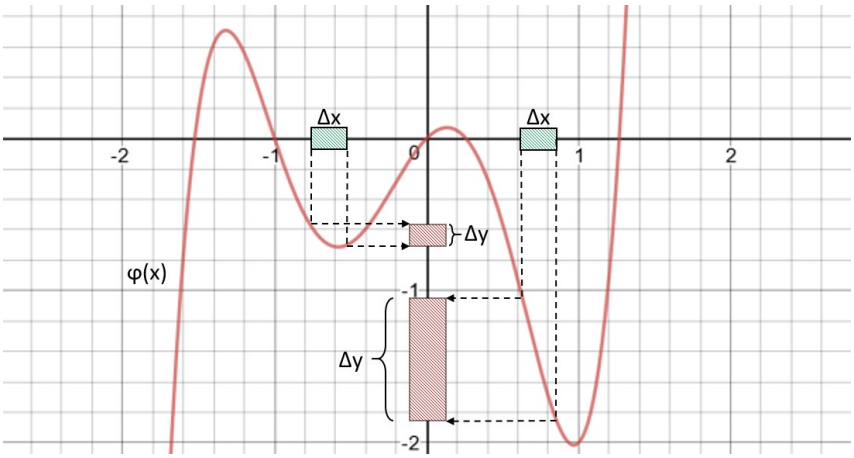


Figura 2.5: Impacto de los errores en x en el cálculo de y.

Notar que la función no cambia, lo que cambia es el resultado por haber modificado los valores de entrada.

Si desarrollamos en serie de Taylor alrededor de los valores verdaderos (desconocidos) y despreciamos términos de orden superior tendremos

$$\Delta y_i = \widetilde{y}_i - y_i = \varphi_i(\widetilde{x}) - \varphi_i(x) \approx \cancel{\varphi_i(x)} + \sum_{j=1}^{n} (\widetilde{x}_j - x_j) \frac{\partial \varphi_i(x)}{\partial x_j} - \cancel{\varphi_i(x)}, \tag{2.62}$$

es decir,

$$\Delta y_i \approx \sum_{j=1}^{n} \frac{\partial \varphi_i(x)}{\partial x_j} \Delta x_j, \qquad i = 1, 2, ..., m. \tag{2.63}$$

Esta expresión puede escribirse de manera más compacta como

$$\Delta y \approx J\varphi(x)\, \Delta x \tag{2.64}$$

donde la matriz

$$J\varphi(x) = \begin{pmatrix} \dfrac{\partial \varphi_1}{\partial x_1} & \dfrac{\partial \varphi_1}{\partial x_2} & \cdots & \dfrac{\partial \varphi_1}{\partial x_n} \\ \vdots & \ddots & & \vdots \\ \dfrac{\partial \varphi_m}{\partial x_1} & \dfrac{\partial \varphi_m}{\partial x_2} & \cdots & \dfrac{\partial \varphi_m}{\partial x_n} \end{pmatrix} \tag{2.65}$$

se denomina la *matriz jacobiana*.

Notar que la cantidad

$$\frac{\partial \varphi_i(x)}{\partial x_j} \tag{2.66}$$

representa la sensibilidad con que el resultado y_i reacciona a la perturbación Δx_j de x_j. Si $y_i \neq 0$, para $i = 1, 2, ..., m$ y $x_j \neq 0$ para $j = 1, 2, ..., n$, puede verse que una fórmula similar vale para los errores relativos

$$\varepsilon_{y_i} \approx \sum_{j=1}^{n} \boxed{\frac{x_j}{\varphi_i(x)} \frac{\partial \varphi_i(x)}{\partial x_j}} \varepsilon_{x_j}. \tag{2.67}$$

donde hemos recuadrado el factor que indica cuán fuertemente afecta el error relativo en x_j al error relativo del resultado y_i.

La diferencia con la fórmula anterior es que estos *factores de amplificación* no dependen de las escalas de x_j e y_i. Los factores de amplificación se llaman **números de condición**. En general, números de condición con valor absoluto grande ($\gg 1$) implican **problemas mal condicionados**[1].

Ejemplo. Vamos a estudiar el condicionamiento de

$$y = \varphi(p, q) = -p + \sqrt{p^2 + q}. \tag{2.68}$$

El error relativo ε_y será

$$\varepsilon_y \approx \frac{p}{\varphi(p,q)} \frac{\partial \varphi(p,q)}{\partial p}\, \varepsilon_p + \frac{q}{\varphi(p,q)} \frac{\partial \varphi(p,q)}{\partial q}\, \varepsilon_q. \tag{2.69}$$

Calculamos las derivadas parciales

$$\frac{\partial \varphi}{\partial p} = -1 + \frac{p}{\sqrt{p^2 + q}} = \frac{(-\varphi)}{\sqrt{p^2 + q}}, \tag{2.70}$$

[1] Hablamos de un problema mal condicionado cuando pequeñas perturbaciones en los datos de entrada generan resultados muy distintos entre sí. El buen o mal condicionamiento de un problema no depende del algoritmo utilizado para resolverlo sino del problema en sí.

$$\frac{\partial \varphi}{\partial q} = \frac{1}{2\sqrt{p^2+q}}. \tag{2.71}$$

Resulta útil reescribir φ de la siguiente manera

$$\varphi(p,q) = (-p+\sqrt{p^2+q})\frac{(p+\sqrt{p^2+q})}{(p+\sqrt{p^2+q})} = \frac{q}{p+\sqrt{p^2+q}} \tag{2.72}$$

lo cual será válido siempre que $p^2 \geq q$.

Reemplazando en la expresión del error

$$\begin{aligned}
\varepsilon_y &\approx \frac{p}{\cancel{\varphi}}\frac{(-\cancel{\varphi})}{\cancel{\varphi}\sqrt{p^2+q}}\,\varepsilon_p + \frac{q}{\varphi}\frac{1}{2\sqrt{p^2+q}}\,\varepsilon_q \\
&= -\frac{p}{\sqrt{p^2+q}}\,\varepsilon_p + \frac{\cancel{q}}{\cancel{q}/(p+\sqrt{p^2+q})}\frac{1}{2\sqrt{p^2+q}}\,\varepsilon_q \\
&= \boxed{-\frac{p}{\sqrt{p^2+q}}\,\varepsilon_p + \frac{p+\sqrt{p^2+q}}{2\sqrt{p^2+q}}\,\varepsilon_q}
\end{aligned} \tag{2.73}$$

Si $q > 0$

$$\left|\frac{p}{\sqrt{p^2+q}}\right| \leq 1, \qquad \text{y} \qquad \left|\frac{p+\sqrt{p^2+q}}{2\sqrt{p^2+q}}\right| \leq 1, \tag{2.74}$$

es decir, en este caso φ está bien condicionada.

Si, en cambio, $q \approx -p^2$, muy probablemente la función estará mal condicionada.

2.6.1. Acotando el error

Si sólo se conocen cotas de los errores absolutos o relativos de las variables x_i, es posible encontrar cotas para los respectivos errores en las variables y_i; por ejemplo, para el caso de los errores relativos tenemos

$$|\varepsilon_{y_i}| \approx \left|\sum_{j=1}^{n}\frac{x_j}{\varphi_i(x)}\frac{\partial \varphi_i(x)}{\partial x_j}\,\varepsilon_{x_j}\right| \leq \sum_{j=1}^{n}\left|\frac{x_j}{\varphi_i(x)}\frac{\partial \varphi_i(x)}{\partial x_j}\right|\,\left|\varepsilon_{x_j}\right|, \qquad (i=1,2,...,m) \tag{2.75}$$

Utilizando la definición de cota superior del error relativo (que vimos antes) podemos escribir

$$\varepsilon^*_{y_i} = \sum_{j=1}^{n}\left|\frac{x_j}{\varphi_i(x)}\frac{\partial \varphi_i(x)}{\partial x_j}\right|\,\varepsilon^*_{x_j}, \qquad (i=1,2,...,m) \tag{2.76}$$

que será una cota superior al error relativo en ε_{y_i}.

Podemos proceder de forma análoga para los errores absolutos, y obtendremos

$$\Delta^* y_i = \sum_{j=1}^{n}\left|\frac{\partial \varphi_i(x)}{\partial x_j}\right|\Delta^* x_j, \qquad (i=1,2,...,m) \tag{2.77}$$

2.6.2. Ejemplo de propagación de errores para casos elementales

Sean $x, y \neq 0$. Calculemos cuál es la propagación del error para cuatro casos elementales.

Producto

$$\varphi(x,y) = x \cdot y \quad \Rightarrow \quad \varepsilon_\varphi \approx \frac{x}{\varphi}\frac{\partial \varphi}{\partial x}\varepsilon_x + \frac{y}{\varphi}\frac{\partial \varphi}{\partial y}\varepsilon_y = \boxed{\varepsilon_x + \varepsilon_y} \tag{2.78}$$

División

$$\varphi(x,y) = x/y \quad \Rightarrow \quad \varepsilon_\varphi \approx \frac{x}{(x/y)}\frac{1}{y}\,\varepsilon_x + \frac{y}{(x/y)}\frac{(-x)}{y^2}\,\varepsilon_y = \boxed{\varepsilon_x - \varepsilon_y} \tag{2.79}$$

Suma y resta

$$\varphi(x,y) = x \pm y \quad \Rightarrow \quad \varepsilon_\varphi \approx \frac{x}{x \pm y}(1)\,\varepsilon_x + \frac{y}{x \pm y}(\pm 1)\,\varepsilon_y = \boxed{\frac{x}{x \pm y}\,\varepsilon_x \pm \frac{y}{x \pm y}\,\varepsilon_y} \tag{2.80}$$

(vale si $x \pm y \neq 0$).

Radicación

$$\varphi(x) = \sqrt{x} \quad \Rightarrow \quad \varepsilon_\varphi \approx \frac{x}{\sqrt{x}}\frac{1}{2\sqrt{x}}\,\varepsilon_x = \boxed{\frac{\varepsilon_x}{2}} \tag{2.81}$$

El análisis de estos resultados, y sus consecuencias, serán hechos en clase.

2.7. Propagación de errores (incluyendo redondeo)

Consideremos ahora el caso más general, en el que incluimos los errores de redondeo que se producen cuando operamos con precisión finita.

Un algoritmo para calcular la función $\varphi : D \to \mathbb{R}^m$, $D \subseteq \mathbb{R}^n$, para algún $x = (x_1, x_2, ..., x_n) \in D$ puede pensarse como una concatenación de funciones o transformaciones en las que φ puede ser descompuesta

$$\varphi = \varphi^{(r)} \circ \varphi^{(r-1)} \circ ... \circ \varphi^{(0)}, \tag{2.82}$$

que nos lleva desde $x^{(0)} \equiv x$, a través de una cadena de resultados intermedios,

$$x = x^{(0)} \to \varphi^{(0)}(x^{(0)}) = x^{(1)} \to ...\varphi^{(r)}(x^{(r)}) = x^{(r+1)} = y \tag{2.83}$$

al resultado y. Las funciones $\varphi^{(i)}$ se llaman transformaciones *elementales*.

Definimos ahora la "transformación restante" como

$$\psi^{(i)} = \varphi^{(r)} \circ \varphi^{(r-1)} \circ ... \circ \varphi^{(i)} : D_i \to \mathbb{R}^m \qquad i = 0, 1, ..., r \tag{2.84}$$

De esta definición, es claro que

$$\psi^{(0)} \equiv \varphi. \tag{2.85}$$

Sean ahora $J\varphi^{(i)}$ y $J\psi^{(i)}$, las matrices jacobianas de las funciones $\varphi^{(i)}$ y $\psi^{(i)}$, respectivamente. Utilizando la siguiente propiedad del jacobiano para la composición de funciones (sin demostración)

$$J(f \circ g)(x) = Jf(g(x)) \cdot Jg(x). \tag{2.86}$$

tenemos entonces que

$$J\varphi(x) = J\varphi^{(r)}(x^{(r)}) \cdot J\varphi^{(r-1)}(x^{(r-1)}) \dots J\varphi^{(0)}(x^{(0)}) \tag{2.87}$$

y

$$J\psi^{(i)}(x^{(i)}) = J\varphi^{(r)}(x^{(r)}) \cdot J\varphi^{(r-1)}(x^{(r-1)}) \dots J\varphi^{(i)}(x^{(i)}). \qquad (i = 0, 1, ..., r) \tag{2.88}$$

Usando aritmética de punto flotante, a los errores de entrada se sumarán ahora los de redondeo, que perturbarán tanto a los valores iniciales como a los resultados intermedios. Así, tendremos que

$$\widetilde{x}^{(i+1)} = \mathrm{FL}\left(\varphi^{(i)}(\widetilde{x}^{(i)})\right) \tag{2.89}$$

donde hemos usado "FL" para simbolizar el efecto de la aproximación de punto flotante. Para analizar el error absoluto de $\widetilde{x}^{(i+1)}$ podemos escribir

$$\begin{aligned}
\Delta x^{(i+1)} &= \widetilde{x}^{(i+1)} - x^{(i+1)} \\
&= \mathrm{FL}\left(\varphi^{(i)}(\widetilde{x}^{(i)})\right) - x^{(i+1)} \\
&= \underbrace{\left[\mathrm{FL}\left(\varphi^{(i)}(\widetilde{x}^{(i)})\right) - \varphi^{(i)}(\widetilde{x}^{(i)})\right]}_{\text{por redondeo}} + \underbrace{\left[\varphi^{(i)}(\widetilde{x}^{(i)}) - \varphi^{(i)}(x^{(i)})\right]}_{\text{por error en } x^{(i)}}
\end{aligned} \tag{2.90}$$

2.7.1. Aporte por el error en $x^{(i)}$

Utilizando la teoría previa (propagación de errores en los datos) podemos estimar el error que aporta la inexactitud de $x^{(i)}$ como

$$\varphi^{(i)}(\widetilde{x}^{(i)}) - \varphi^{(i)}(x^{(i)}) \approx \boxed{J\varphi^{(i)}(x^{(i)})\,\Delta x^{(i)}}. \tag{2.91}$$

2.7.2. Aporte por el error de redondeo

Identificando el error por representación de punto flotante con el redondeo

$$\mathrm{FL}\left(\varphi^{(i)}(u)\right) = \mathrm{RD}\left(\varphi^{(i)}(u)\right), \tag{2.92}$$

y teniendo en cuenta que

$$\varphi^{(i)}(u) = \left(\varphi_1^{(i)}(u),\, \varphi_2^{(i)}(u), \dots,\, \varphi_{n_{i+1}}^{(i)}(u)\right), \tag{2.93}$$

donde $\varphi^{(i)} : D_i \to D_{i+1}$, con $D_{i+1} \subseteq \mathbb{R}^{n_{i+1}}$.

Para cada componente, tenemos entonces que

$$\mathrm{FL}\left(\varphi_j^{(i)}(u)\right) = \mathrm{RD}\left(\varphi_j^{(i)}(u)\right) = (1 + \varepsilon_j)\,\varphi_j^{(i)}(u), \tag{2.94}$$

con $|\varepsilon_j| \leq$eps y $1 \leq j \leq n_{i+1}$. Así, podemos reescribir

$$\mathrm{FL}\left(\varphi^{(i)}(u)\right) = (I + E_{i+1})\varphi^{(i)}(u), \tag{2.95}$$

donde

$$E_{i+1} = \begin{pmatrix} \varepsilon_1 & \dots & 0 \\ \vdots & \ddots & \\ 0 & & \varepsilon_{n_{i+1}} \end{pmatrix}. \tag{2.96}$$

Por lo tanto, los primeros dos términos del error absoluto podrán escribirse como

$$\mathrm{FL}\left(\varphi^{(i)}(\widetilde{x}^{(i)})\right) - \varphi^{(i)}(\widetilde{x}^{(i)}) = (\mathbb{I} + E_{i+1})\varphi^{(i)}(\widetilde{x}^{(i)}) - \cancel{\varphi^{(i)}(\widetilde{x}^{(i)})} = E_{i+1}\varphi^{(i)}(\widetilde{x}^{(i)}), \tag{2.97}$$

que puede ser aproximado por

$$E_{i+1}\,\varphi^{(i)}(\widetilde{x}^{(i)}) \approx E_{i+1}\,\varphi^{(i)}(x^{(i)}), \tag{2.98}$$

ya que los términos de error por los cuales difieren $\varphi^{(i)}(\widetilde{x}^{(i)})$ y $\varphi^{(i)}(x^{(i)})$ son multiplicados por ε en la diagonal de E_{i+1}, dando lugar a términos cuadráticos en el error, que despreciaremos. Así, tendremos que

$$\mathrm{FL}\left(\varphi^{(i)}(\widetilde{x}^{(i)})\right) - \varphi^{(i)}(\widetilde{x}^{(i)}) \approx E_{i+1}\,\varphi^{(i)}(x^{(i)}) = E_{i+1}\,x^{(i+1)} \equiv \boxed{\alpha_{i+1}}. \tag{2.99}$$

Por lo tanto, describiremos con α_{i+1} a la aproximación de los *errores intermedios de redondeo en el paso* $i+1$.

2.7.3. Uniendo todo

Uniendo estos resultados, podremos escribir el error absoluto en el paso $i+1$ como

$$\Delta x^{(i+1)} \approx J\varphi^{(i)}(x^{(i)})\,\Delta x^{(i)} + \alpha_{i+1}, \tag{2.100}$$

para $i = 0, 1, ..., r$ y $\Delta x^{(0)} = \Delta x$. Para hallar el error absoluto total (al finalizar el cálculo) necesitamos ir calculando los errores previos hasta llegar al último paso

$$\Delta x^{(1)} \approx J\varphi^{(0)}(x)\Delta x + \alpha_1 \tag{2.101}$$

$$\begin{aligned} \Delta x^{(2)} &\approx J\varphi^{(1)}(x^{(1)})\Delta x^{(1)} + \alpha_2 \\ &\approx J\varphi^{(1)}(x^{(1)})\left[J\varphi^{(0)}(x)\Delta x + \alpha_1\right] + \alpha_2 \end{aligned} \tag{2.102}$$

$$\begin{aligned} \Delta x^{(3)} &\approx J\varphi^{(2)}(x^{(2)})\Delta x^{(2)} + \alpha_3 \\ &\approx J\varphi^{(2)}(x^{(2)})\left[J\varphi^{(1)}(x^{(1)})\left[J\varphi^{(0)}(x)\Delta x + \alpha_1\right] + \alpha_2\right] + \alpha_3 \\ &\vdots \\ \Delta x^{(r+1)} &\approx J\varphi^{(r)}(x^{(r)})\Delta x^{(r)} + \alpha_{r+1} \\ &\approx J\varphi^{(r)}\,J\varphi^{(r-1)}\dots J\varphi^{(0)}\Delta x + J\varphi^{(r)}\dots J\varphi^{(1)}\alpha_1 + \dots + \alpha_{r+1} \end{aligned} \tag{2.103, 2.104}$$

Recordando que

$$\Delta y \equiv \Delta x^{(r+1)} \tag{2.105}$$

y usando las funciones "transformación restante" (que definimos antes) podemos escribir

$$\Delta y \approx J\varphi(x)\Delta x + J\psi^{(1)}(x^{(1)})\alpha_1 + \dots + J\psi^{(r)}(x^{(r)})\alpha_r + \alpha_{r+1} \tag{2.106}$$

y, finalmente,

$$\boxed{\Delta y \approx J\varphi(x)\,\Delta x + \sum_{k=1}^{r} J\psi^{(k)}(x^{(k)})\,\alpha_k + E_{r+1}\cdot y} \tag{2.107}$$

Observaciones

- Notar que la magnitud de los elementos de las matrices jacobianas $J\psi^{(k)}(x^{(k)})$ es la que determina el efecto de los errores intermedios de redondeo α_i en el resultado final.
- Diferentes algoritmos para calcular el mismo resultado (en otras palabras, diferentes descomposiciones de φ) resultarán en diferentes jacobianos $J\psi^{(k)}(x^{(k)})$ pero no modificarán $J\varphi(x)$. En consecuencia, la elección del algoritmo influirá en el efecto total del redondeo. Siempre deberemos elegir el algoritmo *numéricamente más confiable*, es decir, el de menores errores de redondeo.
- El aporte del error de redondeo introducido en el último paso puede ser acotado

$$|E_{r+1} \cdot y| \leq |y| \,\text{eps} \tag{2.108}$$

  Así, independientemente del algoritmo utilizado para calcular $y = \varphi(x)$, siempre existirá un error de al menos $|y|$ eps en el resultado.
- Por otra parte, el redondeo producido sobre los datos de entrada x causará un error de entrada que podemos acotar

$$|\Delta^{(0)}x| \leq |x| \,\text{eps} \tag{2.109}$$

- De las dos observaciones previas se desprende que TODO algoritmo para computar $y = \varphi(x)$ tendrá este error incorporado, que podemos escribir así

$$\boxed{|\Delta^{(0)}y| = (|J\varphi(x)| \cdot |x| + |y|) \,\text{eps}.} \tag{2.110}$$

  Este error recibe el nombre de *error inherente*. (OJO, en la última expresión las barras de módulo deben interpretarse componente a componente.)
- Llamaremos a los errores intermedios de redondeo α_i como *inofensivos* si su contribución al error total Δy es, como máximo, del orden de magnitud del error inherente, es decir, si se cumple que

$$|J\psi^{(i)}(x^{(i)})\,\alpha_i| \approx |\Delta^{(0)}y|. \tag{2.111}$$

- Si todos los errores de redondeo son inofensivos, el algoritmo es **bien comportado** o **numéricamente estable**.
- De acuerdo con las definiciones previas, un algoritmo puede ser numéricamente más confiable que otro, pero ninguno de los dos ser numéricamente estable. Naturalmente, si ambos son estables, siempre deberemos utilizar aquél que sea numéricamente más confiable.

2.7.4. Ejemplo

Dados $p > 0$, $q > 0$ y $p \gg q$, determínese la raíz

$$y = \varphi(p, q) = -p + \sqrt{p^2 + q} \tag{2.112}$$

con el menor valor absoluto de la ecuación cuadrática

$$y^2 + 2p\,y - q = 0 \tag{2.113}$$

Antes, cuando estudiamos la propagación de errores sin tener en cuenta el redondeo, vimos que el error relativo del resultado quedaba determinado por la siguiente expresión

$$\varepsilon_y \approx \frac{-p}{\sqrt{p^2+q}}\,\varepsilon_p + \frac{p+\sqrt{p^2+q}}{2\sqrt{p^2+q}}\,\varepsilon_q \tag{2.114}$$

Ahora, como $p > 0$ y $q > 0$ tenemos que

$$\left|\frac{-p}{\sqrt{p^2+q}}\right| < 1, \qquad \text{y} \qquad \left|\frac{p+\sqrt{p^2+q}}{2\sqrt{p^2+q}}\right| < 1. \tag{2.115}$$

Por lo tanto, estamos tratando un **problema bien comportado**. El error inherente será

$$\Delta y^{(0)} = \left(\left|\frac{\partial\varphi}{\partial p}\right| p + \left|\frac{\partial\varphi}{\partial q}\right| q + |y|\right) \text{eps} \tag{2.116}$$

que podemos fácilmente transformar en un error inherente relativo

$$\begin{aligned} \varepsilon_{y^{(0)}} &= \left(\left|\frac{\partial\varphi}{\partial p}\right| \frac{p}{|y|} + \left|\frac{\partial\varphi}{\partial q}\right| \frac{q}{|y|} + 1\right) \text{eps} \\ &\leq \boxed{3\,\text{eps}} \end{aligned} \tag{2.117}$$

Ahora que tenemos el error inherente, podremos determinar si los errores de redondeo del algoritmo que utilicemos son inofensivos o no. Vamos a considerar dos algoritmos para el cálculo de $y = -p + \sqrt{p^2+q}$. Con cada algoritmos seguiremos el siguiente orden en las operaciones:

`Algoritmo 1`	`Algoritmo 2`
$s = p^2$ $t = s + q$ $u = \sqrt{t}$ $y = -p + u$	$s = p^2$ $t = s + q$ $u = \sqrt{t}$ $v = p + u$ $y = q/v$

Los primeros tres pasos son iguales, por lo que vamos a concentrar nuestro análisis a las operaciones siguientes. En ambos casos, el cálculo de u incluirá error de redondeo que podemos escribir como

$$\text{FL}(u) = \text{FL}(\sqrt{t}) = (1+\varepsilon)\sqrt{t}, \qquad |\varepsilon| \leq \text{eps} \tag{2.118}$$

por lo que

$$\Delta u = \varepsilon\sqrt{t} = \varepsilon\sqrt{p^2+q}. \tag{2.119}$$

Consideremos primero el Algoritmo 1. La función "transformación restante" es, en este caso,

$$\psi(u) = -p + u, \tag{2.120}$$

Ahora podemos calcular la propagación del error de redondeo y su impacto en el resultado final haciendo

$$\frac{\cancel{u}}{\psi}\frac{\partial\psi(u)}{\partial u}\frac{\Delta u}{\cancel{u}} = \frac{\sqrt{p^2+q}}{-p+\sqrt{p^2+q}}\,\varepsilon = \frac{1}{q}\left(p\sqrt{p^2+q} + p^2 + q\right)\varepsilon = k\,\varepsilon. \tag{2.121}$$

Recordando que $p, q > 0$, tenemos que

$$k > \frac{2p^2}{q} > 0 \tag{2.122}$$

y como estamos trabajando en el caso $p \gg q$, este número será grande. En consecuencia, se ve que el algoritmo 1 es numéricamente inestable ya que la propagación de un error de redondeo intermedio resulta muy superior al error inherente.

Consideremos ahora el Algoritmo 2. En este caso, la función transformación restante será

$$\psi(u) = \frac{q}{p+u}, \tag{2.123}$$

por lo cual, el error Δu se propagará ahora de la siguiente manera

$$\varepsilon_y = \frac{\cancel{u}}{\psi}\frac{\partial\psi(u)}{\partial u}\frac{\Delta u}{\cancel{u}} = \frac{\cancel{p+\sqrt{p^2+q}}}{\cancel{q}}\frac{-\cancel{q}}{(p+\sqrt{p^2+q})^{\cancel{2}}}\sqrt{p^2+q}\,\varepsilon = \frac{-\sqrt{p^2+q}}{p+\sqrt{p^2+q}}\,\varepsilon = k\,\varepsilon. \tag{2.124}$$

Evidentemente

$$|k| < 1, \tag{2.125}$$

y, por lo tanto, se ve que el algoritmo 2 es numéricamente estable.

En clase relacionaremos este problema con el de la cancelación numérica.

Capítulo 3

Resolución de ecuaciones no lineales

Vamos a considerar ahora el problema de encontrar las *raíces de una ecuación* o, lo que es lo mismo, el *cero de una función*.

Decimos que α_r es un **cero de la función** $\varphi(x)$ si

$$\varphi(\alpha_r) = 0. \tag{3.1}$$

Por otra parte, decimos que α_r es una **raíz de la ecuación**

$$\varphi(x) = 0, \tag{3.2}$$

si se cumple que

$$\varphi(\alpha_r) = 0, \tag{3.3}$$

es decir, si verifica la ecuación.

(Espacio reservado para discutir el caso $\varphi(x) = e^{-x} - x$.)

Una forma de resolver este tipo de problemas es a través de métodos iterativos que permiten obtener una aproximación a la raíz buscada. En la práctica, se trata de generar una secuencia x_0, x_1, x_2,..., con x_0 propuesto arbitrariamente, de manera tal de que se cumpla que

$$\lim_{n\to\infty} x_n = \alpha_r, \tag{3.4}$$

Es decir, se busca generar una secuencia que converja a la raíz buscada (en teoría, lo hace en un número infinito de pasos). Obviamente, las secuencias que generemos deberán ser finitas, lo que nos obliga a establecer ciertos criterios que determinen si uno está "suficientemente cerca" de la raíz, como para frenar el proceso iterativo. Estos criterios pueden ser:

1. la distancia **absoluta** entre números consecutivos

$$\boxed{|x_n - x_{n-1}| < \Delta} \tag{3.5}$$

2. la distancia **relativa** entre números consecutivos

$$\boxed{\frac{|x_n - x_{n-1}|}{|x_n|} < \Delta} \tag{3.6}$$

3. módulo de la función cuyo cero se busca

$$\boxed{|\varphi(x_n)| < \Delta} \tag{3.7}$$

donde Δ es una tolerancia prefijada.

El primer criterio tiene el inconveniente que existen secuencias divergentes para las cuales la distancia entre elementos tiende a cero. Veremos un ejemplo en clase. El tercer criterio puede fallar también, ya que existen funciones cuyo módulo se hace muy pequeño "lejos" del cero. En general, el segundo criterio es el más seguro (siempre que x_n no se anule), o una combinación del segundo y el tercero.

Si el procedimiento iterativo tiene éxito, iremos obteniendo valores que convergerán a la solución del problema. Para determinar a qué ritmo nos acercamos a la solución es que se define lo siguiente.

Velocidad de convergencia

Definición. Sea $\{a_n\}_{n=1}^{\infty}$ una sucesión que converge a un número α_r y sea $\{b_n\}_{n=1}^{\infty}$ una sucesión que converge a cero, entonces, si existe una constante K tal que

$$|a_n - \alpha_r| \leq K|b_n|, \tag{3.8}$$

a partir de cierto n, decimos que la secuencia $\{a_n\}_{n=1}^{\infty}$ converge a α_r con *velocidad de convergencia* $O(b_n)$, y lo denotamos así

$$a_n = \alpha_r + O(b_n). \tag{3.9}$$

Esta definición nos permite tipificar la velocidad con la que la sucesión $\{a_n\}_{n=1}^{\infty}$ se acerca a α_r utilizando otra sucesión (sencilla) que nos sirve para acotar la diferencia. En general, se utilizan sucesiones de comparación de la forma

$$b_n = \frac{1}{n^p}, \tag{3.10}$$

con $p > 0$.

Ejemplo 1

$$a_n = \frac{n+1}{n^2}, \qquad n \geq 1. \tag{3.11}$$

Buscamos la sucesión $\{b_n\}_{n=1}^{\infty}$ para este caso.

$$\left|\frac{n+1}{n^2}-0\right| \leq \frac{n+n}{n^2} = \frac{2}{n} = 2\cdot\frac{1}{n}, \qquad n \geq 1. \tag{3.12}$$

Por lo tanto, puedo concluir que

$$a_n = 0 + O\left(\frac{1}{n}\right) \tag{3.13}$$

es decir, que $\{a_n\}$ *converge a cero al menos con la misma velocidad con que lo hace* $1/n$*, a partir de* $n = 1$.

Ejemplo 2

$$a_n = \frac{n+3}{n^3}, \qquad n \geq 1. \tag{3.14}$$

Buscamos la sucesión $\{b_n\}_{n=1}^{\infty}$

$$\left|\frac{n+3}{n^3}-0\right| \leq \frac{2n}{n^3} = \frac{2}{n^2} = 2\cdot\frac{1}{n^2}, \qquad n \geq 3. \tag{3.15}$$

Por lo tanto,

$$a_n = 0 + O\left(\frac{1}{n^2}\right). \tag{3.16}$$

* * *

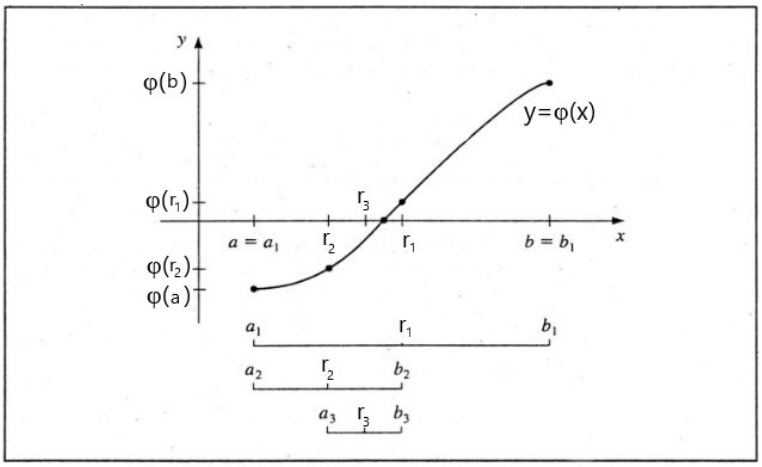
Volviendo al tema de encontrar los ceros de una función, si bien es posible utilizar métodos para funciones con múltiples ceros, vamos a suponer que hemos aislado el cero que nos interesa, es decir, que hemos determinado un intervalo que lo contiene y donde no hay otro. Comenzaremos explicando un método muy sencillo para hallar dicho cero.

3.1. Método de la bisección

Sea $\varphi(x)$ continua en el intervalo $[a, b]$ y sea $\varphi(a)\cdot\varphi(b) < 0$. Con estas hipótesis podemos asegurar que la función $\varphi(x)$ tiene un cero en $[a, b]$ (por el teorema de Bolzano[1]).

El método de la bisección consiste en dividir sucesivamente por la mitad al intervalo dado, tomando como aproximación de la raíz el punto medio del intervalo original.

[1]Teorema de Bolzano. Si una función $f(x)$ está definida y es continua en un intervalo cerrado $[a, b]$, y toma valores de distinto signo en los extremos a y b, entonces existe al menos un punto c del intervalo abierto (a, b) en el que se anula la función.



Sean $a_1 \equiv a$ y $b_1 \equiv b$, definimos la primera aproximación al cero α_r de $\varphi(x)$ como

$$r_1 = \frac{a+b}{2}. \tag{3.17}$$

Evaluamos $\varphi(r_1)$. Si $\varphi(r_1) = 0$, r_1 es el cero buscado. Sin embargo, resulta muy improbable que hallemos así el cero. Lo más normal será que

$$\varphi(r_1) \neq 0, \tag{3.18}$$

en cuyo caso, se dará una de dos situaciones:

$$\begin{cases} a)\ \varphi(a_1)\cdot\varphi(r_1) < 0 \quad \Rightarrow \quad \alpha_r \in [a_1, r_1] \\ \qquad \text{o} \\ b)\ \varphi(b_1)\cdot\varphi(r_1) < 0 \quad \Rightarrow \quad \alpha_r \in [r_1, b_1] \end{cases} \tag{3.19}$$

Si estamos en el caso a), redefinimos el intervalo dentro del cual está el cero haciendo

$$a_2 = a_1,$$
$$b_2 = r_1.$$

En contraposición, si estamos en el caso b) hacemos

$$a_2 = r_1,$$
$$b_2 = b_1.$$

Este proceso se repite en el intervalo $[a_2, b_2]$, determinando r_2, y así sucesivamente.

¿Cómo sabemos si este método converge al cero? Utilizamos el siguiente teorema.

Teorema. Sea $\varphi \in C[a,b]$ y sea $\varphi(a)\cdot\varphi(b) < 0$. El método de la bisección genera una secuencia $\{r_n\}_{n=1}^{\infty}$ que aproxima al cero α_r de $\varphi(x)$ tal que se verifica que

$$|r_n - \alpha_r| < \frac{b-a}{2^n}, \qquad n \geq 1. \tag{3.20}$$

Demostración

Como el método de la bisección finalizaría anticipadamente si se diera la situación de que r_n coincidiera con el cero buscado α_r, vamos a tratar la situación más desfavorable, es decir, cuando $r_n \neq \alpha_r$. Si hacemos, como antes, $a_1 = a$ y $b_1 = b$, es fácil demostrar que en el paso n tendremos que

$$b_n - a_n = \frac{b-a}{2^{n-1}}, \qquad n \geq 1, \tag{3.21}$$

donde se verifica que $\alpha_r \in (a_n, b_n)$. Por otra parte, sabemos que

$$r_n = \frac{a_n + b_n}{2}. \tag{3.22}$$

Por lo tanto,

$$|\alpha_r - r_n| < \frac{1}{2}(b_n - a_n) = \frac{b-a}{2^n}. \tag{3.23}$$

* * *

Observen que como

$$|\alpha_r - r_n| < (b-a)\frac{1}{2^n}, \tag{3.24}$$

podemos decir que

$$r_n = \alpha_r + O\left(\frac{1}{2^n}\right). \tag{3.25}$$

Supongamos ahora que queremos hallar el cero de una función mediante el método de la bisección, y lo queremos con un error absoluto menor a Δ. Dado que conocemos la velocidad de convergencia del método, podemos calcular la cantidad de iteraciones necesarias para obtener la precisión deseada

$$\frac{1}{2^n}(b-a) < \Delta \quad \Rightarrow \quad 2^n > \frac{b-a}{\Delta} \quad \Rightarrow \quad \boxed{n > \frac{\ln\left[(b-a)/\Delta\right]}{\ln 2}.} \tag{3.26}$$

Consideraciones numéricas

- Cuando se implementa este método en la computadora, para calcular la raíz conviene hacer

$$r_n = a_n + \frac{(b_n - a_n)}{2}, \tag{3.27}$$

  en lugar de

$$r_n = \frac{b_n + a_n}{2} \tag{3.28}$$

  ya que introduce menor error de redondeo.

- En vez de evaluar $\varphi(a_n) \cdot \varphi(b_n)$ conviene evaluar

$$\operatorname{sgn}\left(\varphi(a_n)\right) \cdot \operatorname{sgn}\left(\varphi(b_n)\right), \tag{3.29}$$

  para evitar un posible *overflow* o *underflow*.

- El método de la bisección es **lento, pero converge siempre** a la solución. En los paquetes disponibles se lo suele utilizar para generar una mejor aproximación inicial para métodos más veloces.

3.2. Método de punto fijo o de iteración funcional

Para enunciar este método, es necesario primero definir lo que es el punto fijo de una función.

Definición. Un número α_r es un *punto fijo de una función* $\psi(x)$ si

$$\psi(\alpha_r) = \alpha_r. \tag{3.30}$$

Los problemas de encontrar puntos fijos y raíces de ecuaciones son equivalentes.

- Dado un problema de encontrar la raíz α_r de la ecuación

  $$\varphi(x) = 0, \tag{3.31}$$

  podemos definir funciones $\psi(x)$ con un punto fijo en α_r en múltiples formas, por ejemplo, haciendo

  $$\psi_1(x) = x - \varphi(x), \tag{3.32}$$

  o

  $$\psi_2(x) = x + 3\varphi(x). \tag{3.33}$$

- Análogamente, si $\psi(x)$ tiene un punto fijo en α_r, podemos construir una función

  $$\varphi(x) = x - \psi(x), \tag{3.34}$$

  tal que α_r sea la raíz de la ecuación

  $$\varphi(x) = 0. \tag{3.35}$$

Aunque lo que **queremos** es **buscar raíces** de ecuaciones, vamos a transformar el problema en uno de **buscar puntos fijos**, ya que existen métodos muy poderosos para lograr esto último y, como vimos, encontrar el punto fijo de $\psi(x)$ es equivalente a encontrar el cero de $\varphi(x)$.

3.2.1. Teorema de existencia y unicidad de punto fijo

(Existencia) Sea $\psi \in C[a,b]$ y sea $\psi(x) \in [a,b]$, para todo $x \in [a,b]$, entonces $\psi(x)$ tiene uno o más puntos fijos en $[a,b]$.
(Unicidad) Si, además, $\psi'(x)$ existe en (a,b) y existe $0 \leq k < 1$ tal que

$$|\psi'(x)| \leq k, \qquad \forall x \in (a,b) \tag{3.36}$$

entonces el punto fijo de $\psi(x)$ en $[a,b]$ es único.

Demostración

Existencia

Si $\psi(a) = a$ o $\psi(b) = b$, entonces $\psi(x)$ tiene un punto fijo en alguno de los extremos del intervalo y la existencia queda asegurada. Pero este sería un caso fortuito. Consideremos el caso más desfavorable en el que $\psi(a) \neq a$ y $\psi(b) \neq b$. Entonces, como $\psi : [a, b] \to [a, b]$, tendríamos que

$$\psi(a) > a,$$

$$\psi(b) < b.$$

Definiendo una nueva función

$$\eta(x) = \psi(x) - x, \tag{3.37}$$

continua en $[a, b]$, se cumplirá que

$$\eta(a) = \psi(a) - a > 0, \tag{3.38}$$

$$\eta(b) = \psi(b) - b < 0. \tag{3.39}$$

Luego, por el teorema del valor intermedio (teorema de Bolzano), existirá al menos un punto α_r tal que

$$\eta(\alpha_r) = 0. \tag{3.40}$$

En consecuencia, α_r verificará la igualdad

$$\psi(\alpha_r) - \alpha_r = 0 \quad \Rightarrow \quad \boxed{\psi(\alpha_r) = \alpha_r,} \tag{3.41}$$

con lo que queda asegurada la existencia de al menos un punto fijo.

Unicidad

Vamos a demostrar la unicidad proponiendo la existencia de más de un punto fijo y hallando que eso nos conduce a un absurdo.

Sean α_r y α_q puntos fijos de $\psi(x)$ en $[a, b]$, con $\alpha_r < \alpha_q$. Como $\psi(x)$ es derivable en (a, b), por el teorema del valor medio del cálculo diferencial[2], existe un valor $\beta \in (\alpha_r, \alpha_q)$ tal que

$$\frac{\psi(\alpha_r) - \psi(\alpha_q)}{\alpha_r - \alpha_q} = \psi'(\beta). \tag{3.43}$$

Por lo tanto,

$$|\psi(\alpha_r) - \psi(\alpha_q)| = |\psi'(\beta)| \cdot |\alpha_r - \alpha_q| \leq k\,|\alpha_r - \alpha_q| < |\alpha_r - \alpha_q|. \tag{3.44}$$

Utilizando la definición de punto fijo, obtenemos finalmente que

$$|\alpha_r - \alpha_q| < |\alpha_r - \alpha_q|, \tag{3.45}$$

lo cual es un absurdo. Concluimos que el absurdo proviene de haber propuesto la existencia de más de un punto fijo en el intervalo $[a, b]$.

[2]El teorema dice que dada cualquier función f continua en el intervalo $[a, b]$ y derivable en el intervalo abierto (a, b), entonces existe al menos algún punto c en el intervalo (a, b) tal que la tangente a la curva en c es paralela a la recta secante que une los puntos $(b, f(b))$ y $(a, f(a))$. Es decir:

$$\frac{f(b) - f(a)}{b - a} = f'(c). \tag{3.42}$$

Este teorema se conoce también como el *teorema del valor medio de Lagrange*.

Interpretación gráfica

(la haremos en clase)

Ejemplo

Determínese si la función

$$\psi(x) = \frac{x^2 - 1}{3} \tag{3.46}$$

tiene un punto fijo en $[-1, 1]$ y, de existir, si es único.

3.2.2. Iteración de punto fijo

Definición. Llamamos **iteración de punto fijo** al método para aproximar al punto fijo de una función $\psi(x)$ que consiste en elegir un valor inicial x_0 y generar una secuencia $\{x_n\}_{n=0}^{\infty}$ mediante la relación de recurrencia

$$x_n = \psi(x_{n-1}), \qquad n \geq 1. \tag{3.47}$$

Si la secuencia converge, obtendremos una solución del problema $x = \psi(x)$.

3.2.3. Convergencia de la iteración de punto fijo

Teorema. Sea $\psi(x) \in C[a, b]$, $\psi(x) : [a, b] \to [a, b]$. Supongamos, además, que $\psi(x)$ es derivable en (a, b) y que existe $k > 0$ tal que

$$|\psi'(x)| \leq k < 1, \qquad \forall x \in (a, b). \tag{3.48}$$

Entonces, para cualquier $x_0 \in [a,b]$, la secuencia definida por

$$x_n = \psi(x_{n-1}), \qquad n \geq 1 \tag{3.49}$$

converge al único punto fijo de $\psi(x)$ en $[a,b]$.

Demostración

De acuerdo con las hipótesis dadas, sabemos que el punto fijo (α_r) de $\psi(x)$ existe y es único en $[a,b]$. Como $\psi(x):[a,b] \to [a,b]$, la secuencia $\{x_n\}_{n=0}^{\infty}$ está definida para todo $n \geq 0$ y $x_n \in [a,b]$, para todo n. Podemos escribir entonces la distancia entre el punto fijo y x_n como

$$|x_n - \alpha_r| = |\psi(x_{n-1}) - \psi(\alpha_r)| = |\psi'(\xi)(x_{n-1} - \alpha_r)| = |\psi'(\xi)||x_{n-1} - \alpha_r| \leq k|x_{n-1} - \alpha_r|, \tag{3.50}$$

donde hemos hecho uso del teorema del valor medio del cálculo diferencial. En consecuencia,

$$|x_n - \alpha_r| \leq k|x_{n-1} - \alpha_r| \leq k^2|x_{n-2} - \alpha_r| \leq ... \leq k^n|x_0 - \alpha_r|. \tag{3.51}$$

Como $k < 1$, la expresión anterior implica que

$$\lim_{n\to\infty} |x_n - \alpha_r| = 0, \tag{3.52}$$

y queda demostrado que $\{x_n\}_{n=0}^{\infty}$ converge a α_r.

Corolario (sin demostración)

Si $\psi(x)$ satisface las hipótesis del teorema anterior, se puede establecer una cota para el error que se da al aproximar α_r con x_n:

$$|x_n - \alpha_r| \leq \frac{k^n}{1-k}|x_1 - x_0|. \tag{3.53}$$

La velocidad de convergencia depende del valor de la constante k: si $k \to 1^-$, la convergencia se hace lenta, si $k \to 0^+$ es más rápida.

3.2.4. Método de Newton-Raphson

Es uno de los métodos más poderosos y conocidos. Presentaremos primero el método y luego hablaremos de sus ventajas y desventajas.

Deducción

Sea $\varphi(x) \in C^2[a,b]$ y sea x_0 una aproximación al cero α_r de $\varphi(x)$. Desarrollando $\varphi(x)$ alrededor de x_0 y evaluando en α_r tenemos

$$\varphi(\alpha_r) = \varphi(x_0) + (\alpha_r - x_0)\,\varphi'(x_0) + \frac{(\alpha_r - x_0)^2}{2}\varphi''(\xi), \tag{3.54}$$

donde ξ se encuentra entre x_0 y α_r. Si el aporte del término con la derivada segunda es pequeño, cosa que sucederá siempre para $|\alpha_r - x_0|$ suficientemente chico, entonces podemos despreciarlo, y tendremos que

$$\varphi(\alpha_r) \approx \varphi(x_0) + (\alpha_r - x_0)\,\varphi'(x_0). \tag{3.55}$$

Por hipótesis, α_r es un cero de $\varphi(x)$, por lo tanto la expresión anterior se reduce a

$$0 \approx \varphi(x_0) + (\alpha_r - x_0)\,\varphi'(x_0), \tag{3.56}$$

donde α_r es aún desconocido. Despejando α_r obtenemos

$$\alpha_r \approx x_0 - \frac{\varphi(x_0)}{\varphi'(x_0)}, \tag{3.57}$$

válida siempre y cuando $\varphi'(x_0) \neq 0$.

Si la aproximación que hicimos es válida, entonces la evaluación del miembro de la derecha nos permitirá obtener una mejor aproximación a α_r. Este proceso se puede repetir con la idea de seguir mejorando el α_r calculado. Así, obtenemos la expresión de recurrencia del método de Newton-Rahpson

$$\boxed{x_n \approx x_{n-1} - \frac{\varphi(x_{n-1})}{\varphi'(x_{n-1})}.} \qquad n = 1, 2, ... \tag{3.58}$$

Observaciones

- El método de Newton-Raphson construye las sucesivas aproximaciones mediante tangentes a la curva.

- Un problema es que necesita que $\varphi'(x_n) \neq 0$, para todo n; si la derivada es no nula, pero muy pequeña, el método puede no converger.

- Suele considerarse una desventaja tener que calcular analíticamente la derivada $\varphi'(x)$; también el que en cada paso haya que hacer la evaluación de dos funciones (φ y φ'), en lugar de sólo una. Veremos luego métodos que son una variante de éste y que lo evitan.

- Observen que la deducción del método se apoya en que la aproximación inicial esté lo "suficientemente cerca" del cero de la función. Obviamente esto no se sabe de antemano. Lo que se hace es comenzar con alguna estimación que, si es mala, seguramente provocará que el método no converja. También puede comenzarse con algunas iteraciones del método de la bisección para luego continuar con Newton-Raphson.

- Noten que el método de Newton-Raphson es un método de iteración funcional:

$$x_n = \psi(x_{n-1}) = x_{n-1} - \frac{\varphi(x_{n-1})}{\varphi'(x_{n-1})}. \tag{3.59}$$

3.2.5. Convergencia del método de Newton-Raphson

Teorema. Sea $\varphi(x) \in C^2[a, b]$. Si $\alpha_r \in [a, b]$ es tal que $\varphi(\alpha_r) = 0$ y $\varphi'(\alpha_r) \neq 0$, entonces existe $\delta > 0$ tal que la secuencia $\{x_n\}_{n=1}^{\infty}$ generada con el método de Newton-Raphson converge a α_r para cualquier aproximación inicial $x_0 \in [\alpha_r - \delta, \alpha_r + \delta]$.

Demostración

Sea

$$\psi(x) = x - \frac{\varphi(x)}{\varphi'(x)}, \tag{3.60}$$

y sea k una constante tal que $0 \leq k < 1$. Para probar el teorema vamos a demostrar que la función $\psi(x)$ verifica las hipótesis del teorema de convergencia de la iteración de punto fijo en un intervalo como el señalado. Recuerden que dichas hipótesis implican la continuidad de la función $\psi(x)$, que $\psi : [a, b] \to [a, b]$ y que $|\psi'(x)| < 1$, para todo $x \in (a, b)$.

Comencemos por demostrar la continuidad de $\psi(x)$. Por hipótesis sabemos que $\varphi(x)$ y $\varphi'(x)$ son funciones continuas. La continuidad de $\psi(x)$ queda asegurada si $\varphi'(x)$ no se anula en el intervalo que consideremos. Dado que $\varphi'(\alpha_r) \neq 0$ (por hipótesis), podemos asegurar que existe un entorno de α_r en donde $\varphi'(x) \neq 0$. Por lo tanto, existe $\delta_1 > 0$ tal que

$$\varphi'(x) \neq 0, \qquad \forall x \in [\alpha_r - \delta_1, \alpha_r + \delta_1], \tag{3.61}$$

y, en ese entorno, $\psi(x)$ estará bien definida y será continua.

Pasemos ahora a analizar a la derivada de $\psi(x)$:

$$\psi'(x) = 1 - \frac{[\varphi'(x)]^2 - \varphi(x)\,\varphi''(x)}{[\varphi'(x)]^2} = \frac{\varphi(x)\,\varphi''(x)}{[\varphi'(x)]^2} \tag{3.62}$$

Por hipótesis, el numerador es continuo y, por lo visto más arriba, el denominador no se anula si $x \in [\alpha_r - \delta_1, \alpha_r + \delta_1]$. Por lo tanto,

$$\psi(x) \in C^1[\alpha_r - \delta_1, \alpha_r + \delta_1]. \tag{3.63}$$

Evaluando $\psi'(x)$ en α_r y teniendo en cuenta que $\varphi(\alpha_r) = 0$ y que $\varphi'(\alpha_r) \neq 0$, tenemos que

$$\psi'(\alpha_r) = \frac{\varphi(\alpha_r)\,\varphi''(\alpha_r)}{[\varphi'(\alpha_r)]^2} = 0. \tag{3.64}$$

Por ser $\psi'(x)$ una función continua, podemos afirmar que dada $0 \leq k < 1$, existe $\delta > 0$, $\delta < \delta_1$ tal que

$$|\psi'(x)| \leq k, \qquad \forall x \in [\alpha_r - \delta, \alpha_r + \delta]. \tag{3.65}$$

En palabras, estoy diciendo que si una función tiene derivada primera continua y esa derivada se anula en un punto, luego el módulo de la derivada primera está acotado en el entorno de dicho punto.

Sólo nos falta probar que $\psi(x)$ va del intervalo $[\alpha_r - \delta, \alpha_r + \delta]$ en sí mismo. Consideremos $x \in [\alpha_r - \delta, \alpha_r + \delta]$. Usando el teorema del valor medio del cálculo diferencial podemos decir que para η entre x y α_r vale que

$$\psi(x) - \psi(\alpha_r) = \psi'(\eta)(x - \alpha_r). \tag{3.66}$$

Tomando módulo

$$|\psi(x) - \psi(\alpha_r)| = \left|\psi'(\eta)\right| |x - \alpha_r|, \tag{3.67}$$

que podemos acotar

$$|\psi(x) - \psi(\alpha_r)| \leq k\,|x - \alpha_r| < |x - \alpha_r|. \tag{3.68}$$

En consecuencia, como $x \in [\alpha_r - \delta, \alpha_r + \delta]$ tenemos que $|x - \alpha_r| \leq \delta$, lo que implica que

$$\psi(x) \in [\alpha_r - \delta, \alpha_r + \delta], \qquad \forall x \in [\alpha_r - \delta, \alpha_r + \delta]. \tag{3.69}$$

De esta manera, hemos demostrado que todas las hipótesis del teorema de convergencia de la iteración de punto fijo se cumplen. Consecuentemente, podemos afirmar que la secuencia $\{x_n\}_{n=1}^{\infty}$ generada por

$$x_n = \psi(x_{n-1}) = x_{n-1} - \frac{\varphi(x_{n-1})}{\varphi'(x_{n-1})}, \tag{3.70}$$

converge a α_r para todo $x_0 \in [\alpha_r - \delta, \alpha_r + \delta]$.

3.2.6. Método de la secante

Se comentó antes que una eventual desventaja del método de Newton-Raphson era el tener que calcular la derivada de la función $\varphi(x)$. Así, cada paso iterativo implica la evaluación numérica tanto de φ como de φ' en el punto en consideración.

El método de la secante es una variante del método de Newton-Raphson que busca evitar el cálculo de la derivada en forma analítica. Del Análisis Matemático sabemos que

$$f'(x_{n-1}) = \lim_{x \to x_{n-1}} \frac{f(x) - f(x_{n-1})}{x - x_{n-1}}. \tag{3.71}$$

En particular, si x_{n-2} está cerca de x_{n-1}, podemos aproximar

$$f'(x_{n-1}) \approx \frac{f(x_{n-2}) - f(x_{n-1})}{x_{n-2} - x_{n-1}}. \tag{3.72}$$

Reemplazando en el método de Newton-Raphson

$$\boxed{x_n = x_{n-1} - \varphi(x_{n-1}) \frac{x_{n-2} - x_{n-1}}{\varphi(x_{n-2}) - \varphi(x_{n-1})},} \qquad n \geq 2. \tag{3.73}$$

Observen que en este método se necesitan dos valores iniciales, en lugar de uno, y que, además, no se necesita calcular la expresión de la derivada de la función. También es importante señalar que, entre iteraciones, sólo es necesario hacer una única evaluación nueva de la función.

(En clase veremos la interpretación gráfica del método.)

Cuando uno desea comparar los métodos de iteración funcional suele analizar el orden de convergencia, que se define de la siguiente manera.

Definición. Sea $\{x_n\}_{n=0}^{\infty}$ una secuencia que converge a α_r, con $x_n \neq \alpha_r$ para todo n. Si existen constantes positivas γ y λ tales que

$$\lim_{n \to \infty} \frac{|x_{n+1} - \alpha_r|}{|x_n - \alpha_r|^{\gamma}} = \lambda, \tag{3.74}$$

decimos que la secuencia $\{x_n\}_{n=0}^{\infty}$ converge a α_r con orden γ y con constante de error asintótico λ.

Observaciones

- Un método $x_n = \psi(x_{n-1})$ se dice de orden γ si la secuencia $\{x_n\}_{n=0}^{\infty}$ converge a la solución $\alpha_r = \psi(\alpha_r)$ de orden γ.
- En general, una secuencia de orden γ grande converge más rápidamente (es decir, necesita menor número de iteraciones) que una secuencia de menor orden. La constante de error asintótico afecta la velocidad de convergencia, pero no en la misma medida que el orden γ.
- Dos casos que se presentan normalmente son:

$$\begin{cases} \gamma = 1, \ (\lambda < 1) & \text{(convergencia lineal)} \\ \gamma = 2. & \text{(convergencia cuadrática)} \end{cases} \tag{3.75}$$

Ejemplo (convergencias lineal vs. cuadrática)

Sea $\{x_n\}_{n=0}^{\infty}$ una sucesión que converge linealmente a $\alpha_r = 0$ con $\lambda = \frac{1}{2}$

$$\lim_{n\to\infty} \frac{|x_{n+1}|}{|x_n|} = \frac{1}{2}, \tag{3.76}$$

y sea $\{\hat{x}_n\}_{n=0}^{\infty}$ una que lo hace cuadráticamente con la misma constante de error asintótico

$$\lim_{n\to\infty} \frac{|\hat{x}_{n+1}|}{|\hat{x}_n|^2} = \frac{1}{2}. \tag{3.77}$$

Supongamos, para hacer la comparación más simple, que se cumple que

$$\text{(a)} \ \frac{|x_{n+1}|}{|x_n|} \approx \frac{1}{2}, \tag{3.78}$$

$$\text{(b)} \ \frac{|\hat{x}_{n+1}|}{|\hat{x}_n|^2} \approx \frac{1}{2}. \tag{3.79}$$

Entonces, para el caso (a) tendremos

$$\begin{aligned} |x_n - 0| &= |x_n| \\ &\approx 0{,}5\,|x_{n-1}| \\ &\approx 0{,}5^2\,|x_{n-2}| \\ &\approx \dots \\ &\approx \underbrace{0{,}5^n}_{v_1}\,|x_0| \end{aligned}$$

y para el caso (b)

$$\begin{aligned} |\hat{x}_n - 0| &= |\hat{x}_n| \\ &\approx 0{,}5\,|x_{n-1}|^2 \\ &\approx 0{,}5\left(0{,}5\,|x_{n-2}|^2\right)^2 = 0{,}5^3\,|x_{n-2}|^4 \end{aligned}$$

$$\approx 0{,}5^3 \left(0{,}5\,|x_{n-3}|^2\right)^4 = 0{,}5^7\,|x_{n-3}|^8$$

$$\approx ...$$

$$\approx \underbrace{0{,}5^{2^n-1}}_{v_2}\,|x_0|^{2^n}$$

Si $x_0 = \hat{x}_0 = 1$, podemos comparar las velocidades de convergencia v_1 y v_2

n	v_1	v_2
1	0,5	0,5
3	$1{,}25 \times 10^{-1}$	$7{,}8125 \times 10^{-3}$
7	$7{,}8125 \times 10^{-3}$	$5{,}8775 \times 10^{-39}$

Se necesitan 126 iteraciones de la convergencia lineal para llegar al coeficiente v_2 obtenido para $n = 7$.

Claramente, la convergencia cuadrática es mucho más rápida que la convergencia lineal.

* * *

Vamos a ver unos teoremas que prueban que la iteración funcional converge sólo linealmente, mientras que, en general, Newton-Raphson lo hace cuadráticamente.

Teorema. Sea $\psi(x) \in C[a,b]$, $\psi(x) : [a,b] \to [a,b]$. Si además $\psi'(x)$ es continua en (a,b) y existe $0 \le k < 1$ tal que $|\psi'(x)| \le k$, $\forall x \in (a,b)$, entonces, si $\psi' \neq 0$, para cualquier $x_0 \in [a,b]$ la secuencia $\{x_n\}_{n=0}^{\infty}$ obtenida de

$$x_n = \psi(x_{n-1}), \qquad n \ge 1 \tag{3.80}$$

converge sólo linealmente al único punto fijo α_r de $[a,b]$.

Demostración

Las hipótesis aseguran la existencia y unicidad del punto fijo y la convergencia al mismo. Como antes, aplicamos ahora el teorema del valor medio del cálculo diferencial a $\psi(x)$

$$x_{n+1} - \alpha_r = \psi(x_n) - \psi(\alpha_r) = \psi'(\xi_n)(x_n - \alpha_r), \tag{3.81}$$

donde ξ_n está entre x_n y α_r. Ahora, como $\{x_n\}_{n=0}^{\infty}$ converge a α_r, $\{\xi_n\}$ también lo hará. Como pedimos que $\psi'(x)$ sea continua, tendremos que

$$\lim_{n\to\infty} \psi'(\xi_n) = \psi'(\alpha_r), \tag{3.82}$$

luego

$$\lim_{n\to\infty} \frac{|x_{n+1} - \alpha_r|}{|x_n - \alpha_r|} = \lim_{n\to\infty} \left|\psi'(\xi_n)\right| = \left|\psi'(\alpha_r)\right|, \tag{3.83}$$

donde $|\psi'(\alpha_r)| < 1$. Por lo tanto, como se verifica que

$$\boxed{\lim_{n\to\infty} \frac{|x_{n+1} - \alpha_r|}{|x_n - \alpha_r|^1} = \left|\psi'(\alpha_r)\right|,} \tag{3.84}$$

podemos concluir que la iteración de punto fijo converge linealmente con constante de error asintótico $|\psi'(\alpha_r)|$.

* * *

El teorema anterior sugiere que sólo se puede lograr un orden de convergencia mayor en los métodos de iteración funcional si se levanta la condición $\psi'(x) \neq 0$. El siguiente teorema describe las condiciones adicionales que aseguran la convergencia cuadrática.

Teorema (sin demostración). Sea α_r un punto fijo de $\psi(x)$ y sea además $\psi'(\alpha_r) = 0$, $\psi''(x)$ continua tal que $|\psi''(x)| < M$ en un intervalo abierto que contenga a α_r. Entonces, existe $\delta > 0$ tal que para $x_0 \in [\alpha_r - \delta, \alpha_r + \delta]$, la secuencia definida por

$$x_n = \psi(x_{n-1}), \qquad \forall n \geq 1, \tag{3.85}$$

converge **al menos** cuadráticamente a α_r. Más aún, para n grandes

$$|x_{n+1} - \alpha_r| < |x_n - \alpha_r|^2 \frac{M}{2}. \tag{3.86}$$

Observen que este teorema nos dice que para tener métodos de punto fijo con convergencia cuadrática debemos buscar funciones $\psi(x)$ con derivada primera nula en el punto fijo. Veamos cómo tendríamos que construir $\psi(x)$ para lograr esto.

* * *

Supongamos que queremos hallar el cero α_r de la función $\varphi(x)$. Vamos a transformar este problema en el (equivalente) de encontrar el punto fijo de otra función, $\psi(x)$, asociada a $\varphi(x)$ de la siguiente manera

$$\psi(x) = x - \eta(x)\,\varphi(x). \tag{3.87}$$

Si $\eta(x)$ es una función bien comportada en el intervalo de interés, vemos que si α_r es un cero de $\varphi(x)$ también será un punto fijo de $\psi(x)$. Según el teorema que acabamos de ver (sin demostración), para que la convergencia sea al menos cuadrática necesitamos exigir que

$$\psi'(\alpha_r) = 0. \tag{3.88}$$

Calculemos $\psi'(x)$:

$$\psi'(x) = 1 - \eta'(x)\,\varphi(x) - \eta(x)\,\varphi'(x). \tag{3.89}$$

Evaluando en $x = \alpha_r$

$$\psi'(\alpha_r) = 1 - \eta'(\alpha_r)\,\underbrace{\varphi(\alpha_r)}_{=0} - \eta(\alpha_r)\,\varphi'(\alpha_r). \tag{3.90}$$

obtenemos entonces que

$$\psi'(\alpha_r) = 1 - \eta(\alpha_r)\,\varphi'(\alpha_r). \tag{3.91}$$

Por lo tanto,

$$\psi'(\alpha_r) = 0 \quad \Leftrightarrow \quad \eta(\alpha_r) = \frac{1}{\varphi'(\alpha_r)}. \tag{3.92}$$

En consecuencia, si $\varphi'(x) \neq 0$ en el intervalo, podemos hacer

$$\eta(x) = \frac{1}{\varphi'(x)}, \tag{3.93}$$

y asegurarnos que cuando $\varphi(\alpha_r) = 0$ se verifique que $\psi'(\alpha_r) = 0$. Así, podemos definir nuestro iterador de punto fijo de la siguiente manera

$$x_n = \psi(x_{n-1}) = x_{n-1} - \frac{\varphi(x_{n-1})}{\varphi'(x_{n-1})} \tag{3.94}$$

que converge cuadráticamente. (Redescubrimos el método de Newton-Raphson.)

* * *

Hay casos en los que el método de Newton-Raphson converge, pero sólo linealmente. Veamos cuáles son y cómo podemos recuperar la velocidad de convergencia cuadrática.

Cero de multiplicidad m

Definición. Decimos que α_r es un cero de multiplicidad m de $\varphi(x)$ si para $x \neq \alpha_r$ podemos escribir

$$\varphi(x) = (x - \alpha_r)^m \, h(x), \tag{3.95}$$

con

$$\lim_{x \to \alpha_r} h(x) \neq 0. \tag{3.96}$$

Teorema (sin demostración). Una función $\varphi(x) \in C^m[a, b]$ tiene un cero de multiplicidad m en (a,b) sí y sólo sí

$$0 = \varphi(\alpha_r) = \varphi'(\alpha_r) = \varphi''(\alpha_r) = ... = \varphi^{m-1}(\alpha_r), \tag{3.97}$$

y

$$\varphi^m(\alpha_r) \neq 0. \tag{3.98}$$

Este teorema provee de un método sencillo para encontrar la multiplicidad de un cero.

¿Cómo afecta este hecho al método de Newton-Raphson? Recordemos que cuando se demostró la convergencia del método de N-R calculamos $\psi'(x)$ y la evaluamos en α_r:

$$\psi'(\alpha_r) = \frac{\varphi(\alpha_r)\varphi''(\alpha_r)}{[\varphi'(\alpha_r)]^2} = 0, \tag{3.99}$$

donde se pedía que $\varphi'(\alpha_r) \neq 0$. De acuerdo con el teorema anterior, esto no se va a dar si la multiplicidad del cero es mayor que uno. ¿Qué pasa con el método en este caso? Si $\varphi'(\alpha_r) = 0$, es decir, si la multiplicidad del cero de $\varphi(x)$ es mayor que uno, el método de Newton-Raphson converge, pero lo hace más lentamente.

Ejemplo

Sea

$$\varphi(x) = e^x - x - 1. \tag{3.100}$$

Encuéntrese el cero de $\varphi(x)$ usando el método de Newton-Raphson, partiendo de $x_0 = 1$.

La función tiene un cero de multiplicidad 2 en $x = 0$, como lo demuestra lo siguiente

$$\varphi(0) = 0, \qquad \varphi'(0) = 0, \qquad \varphi''(0) = 1 \neq 0. \tag{3.101}$$

Aplicando el método de Newton-Raphson obtenemos los siguientes valores

n	x_n
0	1
5	0,0438
10	$1{,}3881 \times 10^{-3}$
15	$4{,}2610 \times 10^{-5}$

que claramente muestra que la convergencia no es cuadrática. Para recuperar la convergencia cuadrática, se aplica el siguiente teorema.

Teorema. Si α_r es un cero de multiplicidad m de $\varphi(x)$, con $\varphi^{(m)}(x)$ continua en un intervalo abierto que incluya a α_r, entonces

$$\psi(x) = x - m\frac{\varphi(x)}{\varphi'(x)}, \tag{3.102}$$

es tal que $\psi'(\alpha_r) = 0$.

Demostración

Consideremos la función propuesta

$$\psi(x) = x - m\frac{\varphi(x)}{\varphi'(x)}. \tag{3.103}$$

Si α_r es un cero de multiplicidad m de $\varphi(x)$, por definición podemos escribir

$$\varphi(x) = (x-\alpha_r)^m h(x), \qquad (h(\alpha_r) \neq 0). \tag{3.104}$$

Reemplazando en $\psi(x)$

$$\begin{aligned}\psi(x) &= x - m\frac{(x-\alpha_r)^m h(x)}{m(x-\alpha_r)^{m-1}h(x) + (x-\alpha_r)^m h'(x)} \\ &= x - m\frac{(x-\alpha_r)h(x)}{mh(x) + (x-\alpha_r)h'(x)}.\end{aligned} \tag{3.105}$$

Derivando esta última expresión y evaluándola en $x = \alpha_r$ obtenemos

$$\psi'(\alpha_r) = 0. \tag{3.106}$$

* * *

Con este resultado, podemos repetir el cálculo del cero de

$$\varphi(x) = e^x - x - 1, \tag{3.107}$$

pero ahora usando como iterador de punto fijo a la función

$$\psi(x) = x - 2\frac{\varphi(x)}{\varphi'(x)}. \tag{3.108}$$

Arrancando de $x_0 = 1$ obtenemos la siguiente secuencia:

n	x_n
0	1
1	$1{,}639 \times 10^{-1}$
3	$3{,}342 \times 10^{-6}$
4	$-2{,}416 \times 10^{-11}$

que converge a la solución mucho más rápidamente que antes.

3.3. Métodos para encontrar ceros de polinomios

Los polinomios son un caso especial de funciones que permiten una implementación muy eficiente del método de Newton-Raphson. Recordemos que este método requiere de la evaluación de la función cuyo cero queremos encontrar, y de su derivada. Como la derivada de un polinomio también es un polinomio, conviene evaluarlos de la manera más eficiente posible, como veremos a continuación. La idea puede visualizarse mejor con un ejemplo. Consideremos el siguiente polinomio

$$P(x) = 3x^3 - 2x^2 + 6x + 5, \tag{3.109}$$

cuya evaluación requiere de 6 multiplicaciones y tres operaciones de adición. El polinomio puede ser evaluado también haciendo

$$((3x - 2)x + 6)x + 5, \tag{3.110}$$

que implica 3 multiplicaciones y 3 adiciones. Por lo tanto, escribiendo el polinomio adecuadamente podemos ahorrarnos operaciones y, con eso, disminuir el error de redondeo. Aumentar el grado del polinomio incrementa esta diferencia sustancialmente.

Basado en esta idea, el método de Horner nos permite calcular $P(x)$ y $P'(x)$ en forma eficiente.

3.3.1. Método de Horner

Teorema. Sea

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0. \tag{3.111}$$

Si definimos $b_n = a_n$ y

$$b_k = a_k + b_{k+1}\, x_0, \qquad k = n-1, n-2, ..., 0, \tag{3.112}$$

entonces $b_0 = P(x_0)$. Además, si

$$Q(x) = b_n x^{n-1} + b_{n-1}x^{n-2} + ... + b_2 x + b_1, \tag{3.113}$$

luego

$$P(x) = (x - x_0)Q(x) + b_0. \tag{3.114}$$

Demostración

Basta con comprobar que

$$P(x) = (x - x_0)Q(x) + b_0, \tag{3.115}$$

para probar que $P(x_0) = b_0$. Partimos de

$$(x - x_0)Q(x) + b_0 = (x - x_0)\left[b_n x^{n-1} + b_{n-1}x^{n-2} + ... + b_2 x + b_1\right] + b_0, \tag{3.116}$$

donde hemos reemplazado la definición de $Q(x)$. Haciendo la distributiva

$$\begin{aligned}(x - x_0)Q(x) + b_0 &= \\ &= (b_n x^n + b_{n-1}x^{n-1} + ... + b_2 x^2 + b_1 x) - (x_0 b_n x^{n-1} + x_0 b_{n-1}x^{n-2} + ... + x_0 b_2 x + x_0 b_1) + b_0 \\ &= \underbrace{b_n}_{a_n} x^n + \underbrace{(b_{n-1} - x_0 b_n)}_{a_{n-1}} x^{n-1} + \underbrace{(b_{n-2} - x_0 b_{n-1})}_{a_{n-2}} x^{n-2} + ... + \underbrace{(b_0 - x_0 b_1)}_{a_0}. \end{aligned} \tag{3.117}$$

Por lo tanto, vemos que

$$P(x) = (x - x_0)Q(x) + b_0, \tag{3.118}$$

y, en consecuencia, que

$$\boxed{P(x_0) = b_0.} \tag{3.119}$$

Observación

Además de ofrecer un mecanismo recursivo muy eficiente para calcular $P(x_0)$, el método de Horner nos permite calcular $P'(x_0)$ con un costo mínimo. Si tenemos en cuenta que

$$P(x) = (x - x_0)Q(x) + b_0, \tag{3.120}$$

entonces

$$P'(x) = Q(x) + (x - x_0)Q'(x). \tag{3.121}$$

Evaluando en $x = x_0$ tenemos que

$$P'(x_0) = Q(x_0). \tag{3.122}$$

Dado que los coeficientes de Q(x) son calculados en el proceso de hallar $P(x_0)$, se puede establecer una doble recursividad para computar simultáneamente $P(x_0)$ y $P'(x_0)$. Lo veremos en clase.

3.3.2. Método de deflación

Hemos visto cómo evaluar polinomios eficientemente a fin de implementar el método de Newton-Raphson en forma eficaz. Supongamos que hemos encontrado una aproximación a un cero x_1 del polinomio $P(x)$ utilizando el método de Newton-Raphson. Entonces, de acuerdo con lo demostrado para el método de Horner, sabemos que podemos escribir $P(x)$ como

$$P(x) = (x - x_1)Q(x) + b_0 = (x - x_1)Q(x) + P(x_1), \tag{3.123}$$

con $Q(x)$ otro polinomio. Si x_1 es una buena aproximación al cero de $P(x)$ tendremos que

$$P(x_1) \approx 0, \tag{3.124}$$

y tenemos entonces que

$$P(x) \approx (x - x_1)Q_1(x), \tag{3.125}$$

donde hemos rebautizado a $Q(x)$ como $Q_1(x)$ para facilitar la notación que sigue. Ahora, para seguir buscando los ceros de $P(x)$ podemos trabajar con el polinomio $Q_1(x)$ y proceder como lo hicimos más arriba. Así, a medida que vayamos hallando ceros podremos escribir

$$P(x) \approx (x - x_1)(x - x_2)...(x - x_k)Q_k(x). \tag{3.126}$$

A este procedimiento se denomina de deflación y se puede utilizar para encontrar los ceros de $P(x)$. Algunas dificultades que puede presentar:

- los ceros de $P(x)$ son aproximados con mayor error a medida que aumenta k; para mejorarlo, puedo usar los x_k como valores iniciales del polinomio $P(x)$ e iterarlos con el método de Newton-Raphson,
- un polinomio de coeficientes reales puede tener raíces complejas, que no vamos a encontrar si la aproximación inicial es un número real; para hallarlas tenemos que comenzar con una aproximación inicial compleja y hacer los cálculos usando aritmética compleja.

3.4. Método de Müller

Este método puede aplicarse a cualquier problema de búsqueda de raíces, pero es muy utilizado para aproximar ceros de polinomios.

Es similar al método de la secante. Recordemos que éste utiliza dos aproximaciones iniciales y construye la siguiente con la intersección de la recta que une esos puntos con el eje x. El método de Müller utiliza 3 puntos iniciales, y construye la siguiente aproximación con la intersección de la parábola determinada por esos 3 puntos con el eje x.

Sea $\varphi(x)$ el polinomio al cual le queremos calcular el cero. Construimos una parábola $P(x)$ que pase por los puntos $(x_0, P(x_0))$, $(x_1, P(x_1))$ y $(x_2, P(x_2))$:

$$P(x) = a(x - x_2)^2 + b(x - x_2) + c, \tag{3.127}$$

donde a, b y c son hallados de las condiciones

$$\begin{aligned} P(x_0) &= \varphi(x_0), \\ P(x_1) &= \varphi(x_1), \\ P(x_2) &= \varphi(x_2). \end{aligned} \tag{3.128}$$

Haciendo las cuentas, obtenemos

$$\begin{aligned} c &= \varphi(x_2), \\ b &= \frac{(x_0 - x_2)^2 \left(\varphi(x_1) - \varphi(x_0)\right) - (x_1 - x_2)^2 \left(\varphi(x_0) - \varphi(x_2)\right)}{(x_0 - x_2)(x_1 - x_2)(x_0 - x_1)}, \\ a &= \frac{(x_1 - x_2) \left(\varphi(x_0) - \varphi(x_2)\right) - (x_0 - x_2) \left(\varphi(x_1) - \varphi(x_2)\right)}{(x_0 - x_2)(x_1 - x_2)(x_0 - x_1)}. \end{aligned} \tag{3.129}$$

Para determinar x_3, un cero de $P(x)$, usamos la fórmula cuadrática en la forma que minimiza el error de redondeo:

$$P(x_3) = 0, \tag{3.130}$$

$$x_3 - x_2 = r_{1,2} \tag{3.131}$$

$$r_1 = \frac{q}{a}, \qquad r_2 = \frac{c}{q}, \tag{3.132}$$

con

$$q = -\frac{1}{2}\left(b + \operatorname{sgn}(b)\sqrt{b^2 - 4ac}\right). \tag{3.133}$$

De las dos raíces posibles, en el método de Müller se elige la de menor valor absoluto, resultando x_3 el cero de $P(x)$ más próximo a x_2. El proceso se repite, usando x_3, x_2 y x_1 para construir x_4, luego x_4, x_3 y x_2 para hallar x_5, y así sucesivamente hasta que se satisfaga un criterio de finalización elegido.

* * *

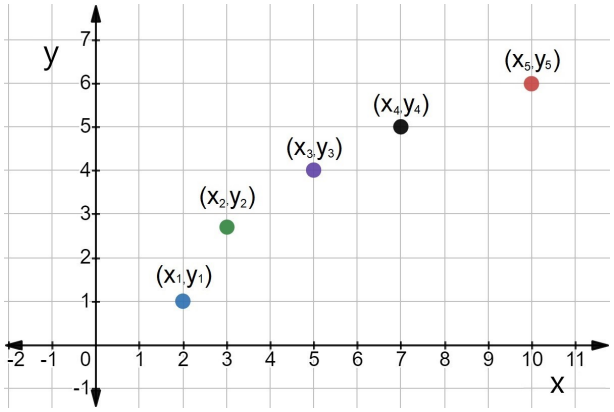
Observaciones

- Como el método involucra el cálculo de $\sqrt{b^2 - 4ac}$ debe considerarse la posibilidad de que esta raíz sea compleja, por lo que usando aritmética compleja, aún para coeficientes y valores iniciales reales, el método converge a la(s) posible(s) raíz(raíces) complejas.
- Es posible probar que el método de Müller es al menos de orden 1.84.
- Las condiciones iniciales deben elegirse de modo tal de garantizar que la parábola que pase por los tres puntos corte al eje x, en caso contrario el método no convergerá.

Capítulo 4

Interpolación

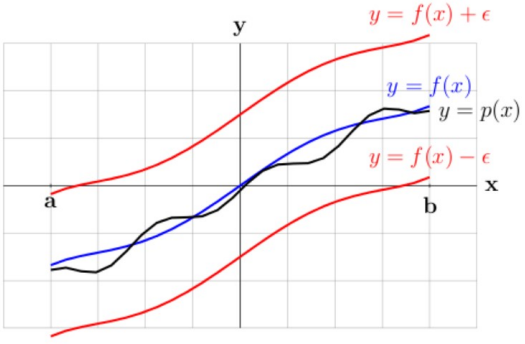
Consideremos la situación en la que nos encontramos ante una secuencia discreta de datos, como la de la siguiente figura



Si no disponemos de la función que dio origen a estos datos o, simplemente, esta función no existe, necesitaremos elaborar alguna estrategia si queremos estimar qué valor tomaría "y" para algún valor intermedio de las abscisas. Una posibilidad (hay otras) sería obtener una función continua que pase por los puntos que ya conocemos. Los polinomios son una buena opción para ello. Dentro de las razones por las cuales los polinomios resultan particularmente útiles tenemos:

1. Teorema de aproximación de Weierstrass: Sea $f : \mathbb{R} \to \mathbb{R}$ continua en $[a, b]$, para cada $\varepsilon > 0$, existe un polinomio $p(x)$ tal que

$$|p(x) - f(x)| < \varepsilon, \quad \forall x \in [a, b]. \tag{4.1}$$



2. Son fáciles de derivar e integrar, y sus derivadas e integrales son a su vez polinomios.

Un candidato de función continua a utilizar podría ser el polinomio de Taylor pero un inconveniente importante que presenta es que concentra toda la información alrededor del punto a partir del cual se realiza el desarrollo (además de requerir el conocimiento de la función y que ésta sea derivable), con lo que puede dar malos resultados para valores de abscisa alejados de ese punto.

A continuación veremos cómo encontrar mejores polinomios que los de Taylor para aproximar a la función subyacente en puntos en los que no disponemos de datos.

4.1. Polinomios de Lagrange

Consideremos la construcción del polinomio $P(x)$ de grado máximo n que pasa por los $n+1$ puntos

$$(x_0, f(x_0)),\ (x_1, f(x_1)), \ldots, (x_n, f(x_n)), \tag{4.2}$$

donde los *nodos* x_i, $i = 0, 1, ..., n$, son todos distintos entre sí. A ese polinomio lo podemos definir como

$$\boxed{P(x) = \sum_{k=0}^{n} f(x_k) L_k(x),} \tag{4.3}$$

con

$$L_k(x) = \frac{(x-x_0)(x-x_1)...(x-x_{k-1})(x-x_{k+1})...(x-x_n)}{(x_k-x_0)(x_k-x_1)...(x_k-x_{k-1})(x_k-x_{k+1})...(x_k-x_n)} = \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{(x-x_i)}{(x_k-x_i)}, \tag{4.4}$$

y recibe el nombre de **n-ésimo polinomio de Lagrange**[1].

Observaciones

• Los $L_k(x)$, $k = 0, 1, ..., n$, son polinomios de grado n, por lo que es evidente que $P(x)$ será un polinomio de grado máximo igual a n.

• Es posible ver que por $n+1$ puntos pasa un único polinomio de grado n o menor. Supongamos que existieran 2 polinomios de grado n que pasaran por $n+1$ puntos. Entonces, la diferencia también sería un polinomio de grado n y ese polinomio tendría $n+1$ ceros, lo cual violaría el Teorema Fundamental del Álgebra.

Ejemplo 1

Construir el polinomio de Lagrange $P(x)$ que pasa por los puntos

$$(x_0, y_0) = (1, 1), \qquad \text{y} \qquad (x_1, y_1) = (4, 7). \tag{4.5}$$

Comenzamos por armar los polinomios $L_k(x)$:

$$L_0 = \frac{x-x_1}{x_0-x_1} = \boxed{\frac{x-4}{1-4}}, \qquad\qquad L_1 = \frac{x-x_0}{x_1-x_0} = \boxed{\frac{x-1}{4-1}}, \tag{4.6}$$

con los cuales construimos el polinomio de Lagrange

$$P(x) = f(x_0)L_0(x) + f(x_1)L_1(x)$$

[1] Analizaremos en clase cómo los polinomios de Lagrange alcanzan el objetivo propuesto.

$$= 1 \cdot \frac{x-4}{1-4} + 7 \cdot \frac{x-1}{4-1}$$

$$= \left(-\frac{1}{3} + \frac{7}{3}\right) x + \frac{4}{3} - \frac{7}{3}$$

$$= \boxed{2x - 1.} \tag{4.7}$$

Ejemplo 2

Supongamos que queremos aproximar a la función $f(x) = 1/x$ a través de un polinomio de Lagrange que intersecte a la misma en $x_0 = 2$, $x_1 = 2{,}5$ y $x_2 = 4$, a fin de estimar su valor en $x = 3$. Para ello, nuevamente debemos calcular primero los $L_k(x)$:

$$L_0(x) = \frac{(x-2{,}5)(x-4)}{(2-2{,}5)(2-4)}, \qquad L_1(x) = \frac{(x-2)(x-4)}{(2{,}5-2)(2{,}5-4)}, \qquad L_2(x) = \frac{(x-2)(x-2{,}5)}{(4-2)(4-2{,}5)}, \tag{4.8}$$

con los que construimos el polinomio de Lagrange

$$P(x) = \sum_{k=0}^{2} f(x_k) L_k(x). \tag{4.9}$$

Haciendo las cuentas encontramos que

$$P(3) = 0{,}325 \tag{4.10}$$

* * *

¿Qué error se comete al aproximar de esta manera?

Teorema (S/D). Sean $\{x_0, x_1, ..., x_n\}$, $n+1$ números distintos en $[a, b]$ y sea $f \in C^{n+1}[a, b]$. Entonces, para cada $x \in [a, b]$ existe un $\alpha(x) \in (a, b)$ (generalmente desconocido) tal que

$$f(x) = P(x) + \frac{f^{(n+1)}(\alpha(x))}{(n+1)!}(x - x_0)(x - x_1)...(x - x_n), \tag{4.11}$$

donde

$$P(x) = \sum_{k=0}^{n} f(x_k) \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{(x - x_i)}{(x_k - x_i)}. \tag{4.12}$$

• Observen que la fórmula del error es similar a la de Taylor, con la diferencia que en el último caso la información está centrada en x_0 (por eso aparece $(x - x_0)^{n+1}$), mientras que en los polinomios de Lagrange utilizan la información en los puntos $x_0, x_1, ..., x_n$.

• Un inconveniente que se tiene para usar esta fórmula es que generalmente NO se conoce la forma de $f^{(n+1)}(x)$, por lo que es difícil acotar el error. La importancia de la fórmula (del error de interpolación) reside en su utilización, por ejemplo, para el cálculo de errores en la integración numérica, lo que veremos más adelante.

• La forma práctica de aproximar un valor es calcular distintas aproximaciones generando polinomios interpoladores que involucren cada vez más datos, hasta lograr una estimación aceptable (que coincida determinado número de decimales en el resultado).

Ejemplo

Veamos cómo se trabaja si no se conoce $f(x)$ y, por ende, tampoco sus derivadas. Supongamos que tenemos una tabla de valores de la función en cinco puntos.

i	x_i	$f(x_i)$
0	1.0	0.7651977
1	1.3	0.6200860
2	1.6	0.4554022
3	1.9	0.2818186
4	2.2	0.1103623

Queremos calcular $f(1{,}5)$. Como no tenemos $f^{n+1}(x)$ es imposible usar la fórmula del error, pero lo que sí se puede hacer es ir construyendo polinomios de Lagrange de grado cada vez mayor e ir comparando el resultados con los de grado menor. Comencemos por construir un polinomio de grado 1, para lo cual siempre conviene elegir los 2 datos más próximos al que nos interesa calcular. En nuestro caso, los datos más próximos a $x = 1{,}5$ serán $x_1 = 1{,}3$ y $x_2 = 1{,}6$, con lo cual armamos el polinomio

$$P_1(x) = \frac{(x-1{,}6)}{(1{,}3-1{,}6)} f(1{,}3) + \frac{(x-1{,}3)}{(1{,}6-1{,}3)} f(1{,}6) \tag{4.13}$$

tal que

$$P_1(1{,}5) = 0{,}5102968. \tag{4.14}$$

Para construir polinomios de grado 2, tenemos dos alternativas. Por un lado, utilizar los puntos $\{x_0, x_1, x_2\}$:

$$P_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2), \tag{4.15}$$

o bien, utilizar $\{x_1, x_2, x_3\}$:

$$P_2'(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} f(x_1) + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} f(x_2) + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} f(x_3). \tag{4.16}$$

Con estos polinomios, obtenemos

$$P_2(1{,}5) = 0{,}5124715, \tag{4.17}$$

$$P_2'(1{,}5) = 0{,}5112857. \tag{4.18}$$

Repitiendo el proceso, pero ahora para 4 puntos:

$$\{x_0, x_1, x_2, x_3\} \quad \longrightarrow \quad P_3(1{,}5) = 0{,}5118127, \tag{4.19}$$

$$\{x_1, x_2, x_3, x_4\} \quad \longrightarrow \quad P_3'(1{,}5) = 0{,}5118302, \tag{4.20}$$

y, finalmente, con todos los puntos

$$\{x_0, x_1, x_2, x_3, x_4\} \quad \longrightarrow \quad P_4(1{,}5) = 0{,}5118200. \tag{4.21}$$

A medida que vamos generando polinomios podemos ir comparando los valores obtenidos para la abscisa que nos interesa. Así, vemos cómo cada vez hay más decimales que no cambian y que, por lo tanto, parecen "correctos". A priori uno tendería a pensar que $P_4(x)$ debería ser el polinomio que nos ofrece el resultado más preciso, sin embargo, esto no necesariamente es cierto. La función tabulada es la función de Bessel de primer tipo y orden cero, cuyo valor para $x = 1{,}5$ es 0,5118277. Se ve que el resultado más próximo al real fue el de P_3'.

* * *

Como vimos en el ejemplo anterior, hasta ahora nunca utilizamos la información del polinomio de grado k para obtener el de grado $k+1$, ni la de éste para obtener el de grado $k+2$. Esto implica que cada vez que uno desea incorporar un nuevo punto para construir un polinomio de Lagrange, tiene que hacer todas las cuentas desde cero. A continuación mostraremos que es posible basarnos en polinomios de Lagrange de grado menor para construir uno de grado mayor. Antes de hacerlo, necesitamos definir una nueva notación.

Definición. Sea $f(x)$ una función definida en los $n+1$ puntos $x_0, x_1, ..., x_n$ y sean m_i, $i = 1, ..., k$, k enteros distintos que verifican que $0 \leq m_i \leq n$, $\forall i$. Denotaremos por

$$P_{m_1,m_2,...,m_k}(x), \tag{4.22}$$

al polinomio de Lagrange que coincide con $f(x)$ en los k puntos $x_{m_1}, x_{m_2}, ..., x_{m_k}$.

Ejemplo

En virtud de la notación introducida, el polinomio de Lagrange $P_{1,2,4}(x)$ que coincide con $f(x)$ en los puntos x_1, x_2 y x_4 será

$$P_{1,2,4}(x) = f(x_1)\frac{(x-x_2)(x-x_4)}{(x_1-x_2)(x_1-x_4)} + f(x_2)\frac{(x-x_1)(x-x_4)}{(x_2-x_1)(x_2-x_4)} + f(x_4)\frac{(x-x_1)(x-x_2)}{(x_4-x_1)(x_4-x_2)}. \tag{4.23}$$

Podemos presentar entonces el siguiente teorema.

Teorema. Sea $f(x)$ una función definida en $\{x_0, x_1, ..., x_n\}$ y sean x_i y x_j, dos números distintos dentro del conjunto. Entonces, el n-ésimo polinomio de Lagrange que interpola a $f(x)$ entre los $n+1$ puntos $\{x_0, x_1, ..., x_n\}$ está dado por

$$P(x) = \frac{(x-x_j)P_{0,1,2,...,j-1,j+1,...,n}(x) - (x-x_i)P_{0,1,2,...,i-1,i+1,...,n}(x)}{x_i - x_j}. \tag{4.24}$$

Demostración

Para simplificar un poco la notación, vamos a hacer

$$Q^i(x) \equiv P_{0,1,2,...,i-1,i+1,...,n}(x), \tag{4.25}$$

$$Q^j(x) \equiv P_{0,1,2,...,j-1,j+1,...,n}(x). \tag{4.26}$$

Como $Q^i(x)$ y $Q^j(x)$ son polinomios de grado $n-1$ (o menor, pero nunca mayor), $P(x)$ tendrá grado máximo n. Consideremos $0 \leq r \leq n$ tal que $r \neq i$ y $r \neq j$. Por construcción, sabemos que

$$Q^i(x_r) = Q^j(x_r) = f(x_r), \tag{4.27}$$

luego

$$P(x_r) = \frac{(x_r-x_j)Q^j(x_r) - (x_r-x_i)Q^i(x_r)}{x_i - x_j} = \frac{x_i - x_j}{x_i - x_j}f(x_r) = f(x_r). \tag{4.28}$$

Además, tenemos que $Q^j(x_i) = f(x_i)$, luego

$$P(x_i) = \frac{(x_i-x_j)Q^j(x_i) - \cancel{(x_i-x_i)Q^i(x_i)}}{x_i - x_j} = f(x_i). \tag{4.29}$$

Análogamente, como $Q^i(x_j) = f(x_j)$, resulta que $P(x_j) = f(x_j)$. Ahora, por definición, $P_{0,1,2,...,n}(x)$ es el único polinomio de grado n que coincide con $f(x)$ en $x_0, x_1, ..., x_n$, por lo tanto

$$\boxed{P(x) = P_{0,1,2,...,n}(x).} \tag{4.30}$$

* * *

Como consecuencia de este teorema, tenemos una forma recursiva (es decir, utilizando información previa) de construir los polinomios de Lagrange. Este mecanismo se llama método de Neville.

Método de Neville

Veamos cómo proceder en la práctica. Supongamos que conocemos 5 puntos de la función: $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$, $(x_3, f(x_3))$ y $(x_4, f(x_4))$. Los sucesivos polinomios de Lagrange pueden ir construyéndose con una tabla como la de abajo, completando líneas de arriba hacia abajo para finalmente obtener el polinomio que comprende a todos los puntos.

x_0	P_0				
x_1	P_1	$P_{0,1}$			
x_2	P_2	$P_{1,2}$	$P_{0,1,2}$		
x_3	P_3	$P_{2,3}$	$P_{1,2,3}$	$P_{0,1,2,3}$	
x_4	P_4	$P_{3,4}$	$P_{2,3,4}$	$P_{1,2,3,4}$	$P_{0,1,2,3,4}$

* * *

Como dijimos antes, a menudo se cree que la utilización de un polinomio de mayor grado asegura una mejor aproximación. Esto en general no es así. Con los polinomiosde Lagrange, a medida que uno aumenta el número de puntos también aumenta el grado y es aquí donde la naturaleza oscilatoria de los polinomios nos puede jugar una mala pasada (*efecto Runge* o *fenómeno Runge*).

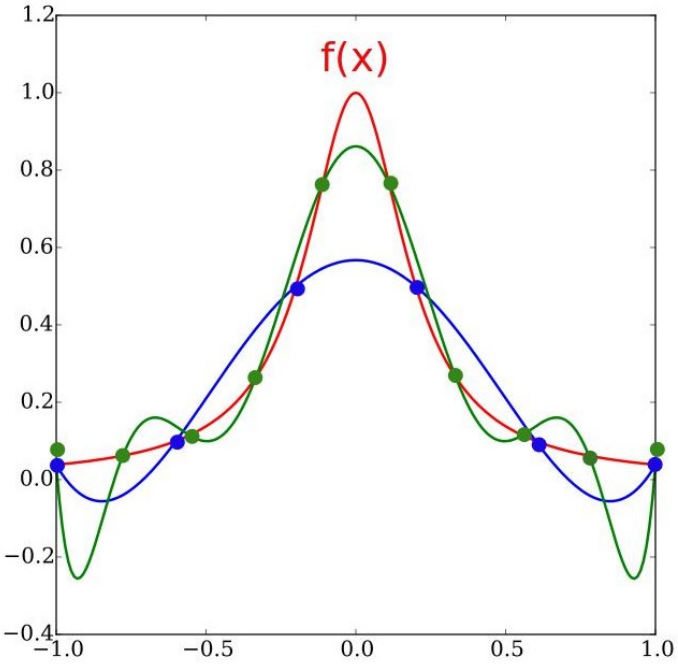


Figura 4.1: Efecto Runge.

Para evitar este problema uno puede recurrir a la estrategia que explicamos a continuación.

4.2. Interpolación cúbica segmentaria (splines)

Introducción (en clase)

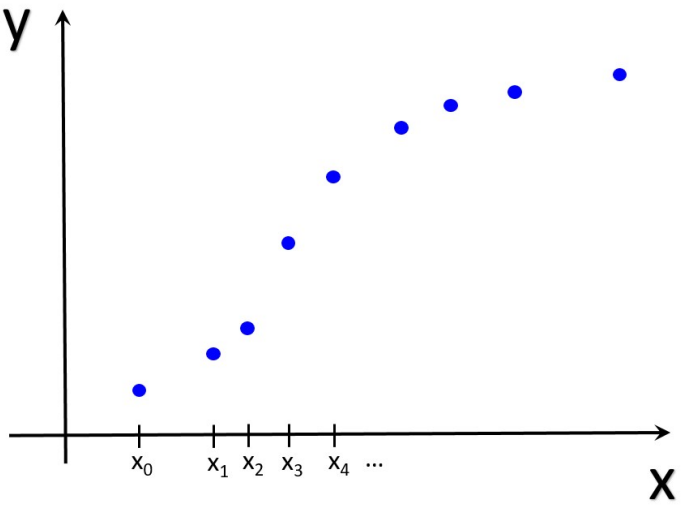


Figura 4.2: Interpolación cúbica segmentaria

Definición. Dada una función $f(x)$ definida en $[a, b]$ y un conjunto de $n + 1$ nodos

$$a = x_0 < x_1 < x_2 < ... < x_n = b, \tag{4.31}$$

un interpolador polinómico cúbico segmentario $S(x)$ para $f(x)$ es una función que satisface las siguientes condiciones:

(a) $S(x)$ es un polinomio cúbico, denotado $S_j(x)$ en el subintervalo $[x_j, x_{j+1}]$, donde $j = 0, 1, 2, ..., n - 1$

(b) $S_j(x_j) = f(x_j)$ y $S_j(x_{j+1}) = f(x_{j+1})$ para $j = 0, 1, 2, ..., n - 1$

(c) $S_j(x_{j+1}) = S_{j+1}(x_{j+1})$ para $j = 0, 1, 2, ..., n - 2$

(d) $S'_j(x_{j+1}) = S'_{j+1}(x_{j+1})$ para $j = 0, 1, 2, ..., n - 2$

(e) $S''_j(x_{j+1}) = S''_{j+1}(x_{j+1})$ para $j = 0, 1, 2, ..., n - 2$

(f) una de las siguientes condiciones de borde es satisfecha

(I) $S''_j(x_0) = S''_j(x_n) = 0$ (condición natural o libre)

(II) $S'_j(x_0) = f'(x_0)$ y $S'_j(x_n) = f'(x_n)$ (condición sujeta)

• Es posible probar que los coeficientes de todos los polinomios interpoladores, que tienen la forma

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3, \tag{4.32}$$

pueden hallarse resolviendo un **sistema lineal tridiagonal**, que siempre es **diagonal dominante**. Consecuentemente, se puede demostrar (usando el teorema de Gerschgorin, que veremos más adelante) que el sistema tiene solución y ésta es única.

4.3. Cuadrados mínimos

Consideremos la situación que se observa en los siguientes gráficos, donde los puntos y_i, $i = 1, ..., m$, pueden contener errores.

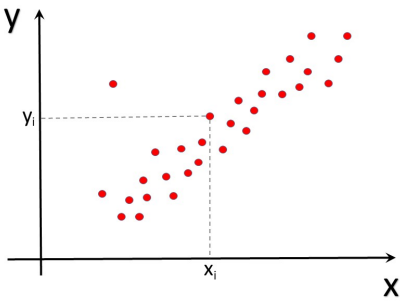


Figura 4.3: Datos A.

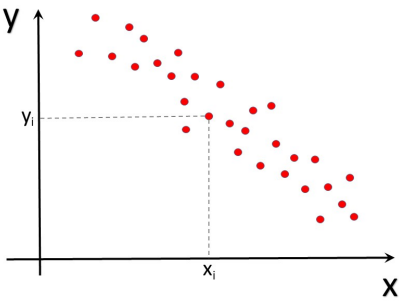


Figura 4.4: Datos B.

En este caso, no tiene sentido interpolar exactamente todos los puntos mediante un polinomio de grado $m-1$, ni intentar realizar un ajuste por *splines*. Noten, sin embargo, que los puntos graficados parecen seguir un comportamiento general de la forma

$$y = a_1 x + a_0, \tag{4.33}$$

lo que de alguna manera refuerza la idea de abandonar la interpolación tal como la vimos. Lo que es evidente es que una recta no va a pasar por todos los puntos, de modo que debemos elegir algún criterio que nos permita determinar con cuál de todas las posibles rectas quedarnos. Existen diferentes criterios que se pueden elegir. A continuación mencionamos algunos.

4.3.1. Criterios para elegir el "mejor" ajuste

1. Podemos intentar hallar el mejor ajuste buscando los valores de a_0 y a_1 que minimicen el estimador de error
$$E_\infty(a_1, a_0) = \max_{\forall i} \{|y_i - (a_1 x_i + a_0)|\}. \tag{4.34}$$
De esta manera, buscamos la recta cuya distancia al punto de peor ajuste sea la mínima. A este criterio se lo denomina **MINIMAX** y se caracteriza por ser difícil de implementar.

2. Otra estrategia se basa en hallar a_0 y a_1 que minimicen el error total
$$E_1(a_1, a_0) = \sum_{i=1}^{n} |y_i - (a_1 x_i + a_0)|. \tag{4.35}$$
$E_1(a, b)$ se llama **desviación absoluta**, y para encontrar el mínimo debemos calcular
$$\frac{\partial E_1}{\partial a_1} = 0, \qquad \text{y} \qquad \frac{\partial E_1}{\partial a_0} = 0, \tag{4.36}$$
lo que implica derivar la función valor absoluto, que en el cero no es derivable, por lo que podemos llegar a tener dificultades para encontrar la solución.

3. Una tercera alternativa es buscar el par de parámetros que minimicen la suma del cuadrado de los errores
$$E_2(a_1, a_0) = \sum_{i=1}^{n} \left[y_i - (a_1 x_i + a_0)\right]^2, \tag{4.37}$$

que es la llamada **aproximación de cuadrados mínimos**. Esta es la opción con la que vamos a trabajar. La recta $y = a_1x + a_0$ con la que nos quedaremos será la que minimice la suma de los cuadrados de las distancias entre el "dato experimental", y_i, y el valor predicho por la recta $a_1x_i + a_0$, para la misma abscisa. Esta opción se elige no sólo porque minimizar $E_2(a_1, a_0)$ es matemáticamente más simple que las opciones anteriores, sino porque estadísticamente es mejor y, además, provee de una distribución más adecuada de los pesos relativos de los errores comparado con los métodos anteriores.

4.3.2. Búsqueda de parámetros en cuadrados mínimos

Para hallar a_0 y a_1 hacemos

$$\frac{\partial E_2}{\partial a_0} = 2\sum_{i=1}^{m}\big[y_i - (a_1x_i + a_0)\big](-1) = 0 \tag{4.38}$$

$$\frac{\partial E_2}{\partial a_1} = 2\sum_{i=1}^{m}\big[y_i - (a_1x_i + a_0)\big](-x_i) = 0 \tag{4.39}$$

y, por lo tanto, habrá que resolver el sistema

$$\begin{cases} a_1\displaystyle\sum_{i=1}^{m} x_i + a_0\sum_{i=1}^{m} 1 = \sum_{i=1}^{m} y_i, \\ a_1\displaystyle\sum_{i=1}^{m} x_i^2 + a_0\sum_{i=1}^{m} x_i = \sum_{i=1}^{m} y_ix_i. \end{cases} \tag{4.40}$$

Este sistema puede reescribirse en forma matricial definiendo

$$\boldsymbol{A} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ & \vdots \\ 1 & x_m \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}, \qquad \boldsymbol{x} = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}, \tag{4.41}$$

de modo que el sistema de dos ecuaciones con dos incógnitas nos queda simplemente como

$$\boxed{\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^T\boldsymbol{b},} \tag{4.42}$$

que es el llamado **sistema de ecuaciones normales**[2] asociado al sistema lineal $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. Como las abscisas x_i se suponen distintas entre sí, resulta que el rango de $\boldsymbol{A}$ es igual a 2, lo que permite asegurar que la solución es única (resultado de la teoría de matrices normales) e igual a

$$\boldsymbol{x} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b} = \frac{1}{m\sum_{i=1}^{m} x_i^2 - \left(\sum_{i=1}^{m} x_i\right)^2}\begin{pmatrix} \sum_{i=1}^{m} x_i^2 & -\sum_{i=1}^{m} x_i \\ \sum_{i=1}^{m} x_i & m \end{pmatrix}\begin{pmatrix} \sum_{i=1}^{m} y_i \\ \sum_{i=1}^{m} y_ix_i \end{pmatrix}. \tag{4.43}$$

[2]Sobre los sistemas de ecuaciones normales:

- Para un sistema de $m \times n$ (m ecuaciones con n incógnitas) $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, el sistema asociado de ecuaciones normales es $\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^T\boldsymbol{b}$. Este nuevo sistema es de $n \times n$.
- $\boldsymbol{A}^T\boldsymbol{A}x = \boldsymbol{A}^T\boldsymbol{b}$ es siempre compatible, aún cuando $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ no lo sea.
- Cuando $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ es compatible, el conjunto de soluciones coincide con el de $\boldsymbol{A}^T\boldsymbol{A}x = \boldsymbol{A}^T\boldsymbol{b}$.
- $\boldsymbol{A}^T\boldsymbol{A}x = \boldsymbol{A}^T\boldsymbol{b}$ tiene solución única sí y sólo sí $r(\boldsymbol{A}) = n$, en cuyo caso $\boldsymbol{x} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b}$.
- Cuando $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ es compatible determinado, lo mismo vale para las ecuaciones normales y la única solución a ambos sistemas es $\boldsymbol{x} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b}$.

Noten que

$$\boxed{E_2 = \sum_{i=1}^{n} \left[y_i - (a_1 x_i + a_0)\right]^2 = (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) = \sum_{i=1}^{n} \varepsilon_i^2 = \boldsymbol{\varepsilon}^T\boldsymbol{\varepsilon},} \tag{4.44}$$

con

$$\varepsilon_i = y_i - (a_1 x_i + a_0). \tag{4.45}$$

Pronto veremos que el funcional de cuadrados mínimos tiene la misma forma para otras funciones que poseen distinto número de parámetros. En ese caso, $\boldsymbol{A}$ y $\boldsymbol{x}$ cambian, pero E_2 puede seguir escribiéndose como en esta fórmula.

Antes de ocuparnos de ese caso, presentamos lo que se denomina el "problema general de cuadrados mínimos".

Problema general de cuadrados mínimos. Sea $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ y $\boldsymbol{b} \in \mathbb{R}^m$; sea además $\boldsymbol{\varepsilon}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}$, el problema general de cuadrados mínimos consiste en encontrar un vector $\boldsymbol{x}$ que minimice

$$E_2(\boldsymbol{x}) = \sum_{i=1}^{m} \varepsilon_i^2 = \boldsymbol{\varepsilon}^T\boldsymbol{\varepsilon} = (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}). \tag{4.46}$$

Cualquier vector $\boldsymbol{x}$ que provea un valor mínimo para esta expresión se llama solución de cuadrados mínimos.

Teorema. Bajo las hipótesis del problema general de cuadrados mínimos, se verifica que:

(a) el conjunto de todas las soluciones de cuadrados mínimos es precisamente el conjunto de soluciones de las ecuaciones normales;

(b) la solución de cuadrados mínimos es única sí y sólo sí $r(\boldsymbol{A}) = n$, en cuyo caso está dada por $(\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b}$;

(c) si el sistema $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ es compatible entonces las soluciones del sistema $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ son las mismas que las de cuadrados mínimos.

Demostración

(a) Vamos a demostrar primero que todo vector que sea solución de cuadrados mínimos verifica las ecuaciones normales.

$$\begin{aligned} E_2(\boldsymbol{x}) = \sum_{i=1}^{m} \varepsilon_i^2 = \boldsymbol{\varepsilon}^T\boldsymbol{\varepsilon} &= (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) \\ &= (\boldsymbol{A}\boldsymbol{x})^T\boldsymbol{A}\boldsymbol{x} - (\boldsymbol{A}\boldsymbol{x})^T\boldsymbol{b} - \boldsymbol{b}^T\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{b} \\ &= \boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{b} - \boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{b} + \boldsymbol{b}^T\boldsymbol{b} \\ &= \boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} - 2\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{b} + \boldsymbol{b}^T\boldsymbol{b}. \end{aligned} \tag{4.47}$$

Para hallar el vector $\boldsymbol{x}$ que minimice la expresión anterior vamos a aplicar las técnicas del análisis matemático para diferenciar a la función matricial

$$f(\boldsymbol{x}) = f(x_1, x_2, ..., x_n) = \boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} - 2\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{b} + \boldsymbol{b}^T\boldsymbol{b}. \tag{4.48}$$

Diferenciar una función matricial es similar a la diferenciación de funciones escalares. Supongamos que tenemos una matriz $\boldsymbol{U} = [u_{ij}]$, entonces

$$\left[\frac{\partial \boldsymbol{U}}{\partial x}\right]_{ij} = \frac{\partial u_{ij}}{\partial x}, \qquad \frac{\partial[\boldsymbol{U}+\boldsymbol{V}]}{\partial x} = \frac{\partial \boldsymbol{U}}{\partial x} + \frac{\partial \boldsymbol{V}}{\partial x}, \qquad \frac{\partial[\boldsymbol{U}\boldsymbol{V}]}{\partial x} = \frac{\partial \boldsymbol{U}}{\partial x}\boldsymbol{V} + \boldsymbol{U}\frac{\partial \boldsymbol{V}}{\partial x}. \tag{4.49}$$

Aplicando estas reglas a la función f

$$\frac{\partial f}{\partial x_i} = \frac{\partial \boldsymbol{x}^T}{\partial x_i}\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} + \boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\frac{\partial \boldsymbol{x}}{\partial x_i} - 2\frac{\partial \boldsymbol{x}^T}{\partial x_i}\boldsymbol{A}^T\boldsymbol{b}, \tag{4.50}$$

y teniendo en cuenta que

$$\frac{\partial \boldsymbol{x}}{\partial x_i} = \boldsymbol{e}_i, \tag{4.51}$$

obtenemos

$$\begin{aligned}\frac{\partial f}{\partial x_i} &= \boldsymbol{e}_i^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} + \boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{e}_i - 2\boldsymbol{e}_i^T\boldsymbol{A}^T\boldsymbol{b} \\ &= 2\boldsymbol{e}_i^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} - 2\boldsymbol{e}_i^T\boldsymbol{A}^T\boldsymbol{b} \\ &= 2(\boldsymbol{A}_{i*}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}_{i*}^T\boldsymbol{b}).\end{aligned} \tag{4.52}$$

Como buscamos el mínimo de f, debemos hallar $\boldsymbol{x}$ tal que

$$\frac{\partial f}{\partial x_i} = 0, \qquad \text{para } i = 1, 2, ..., n. \tag{4.53}$$

Por lo tanto, usando notación vectorial, en el mínimo deberá cumplirse que

$$2(\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^T\boldsymbol{b}) = 0 \qquad \Rightarrow \qquad \boxed{\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^T\boldsymbol{b}.} \tag{4.54}$$

Consecuentemente, queda demostrado que toda solución de cuadrados mínimos verifica las ecuaciones normales.

Ahora deberemos ver que si $\boldsymbol{x}$ es solución de las ecuaciones normales, entonces minimiza a f. Consideremos un vector $\boldsymbol{z}$, que es solución de las ecuaciones normales

$$\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{z} = \boldsymbol{A}^T\boldsymbol{b}. \tag{4.55}$$

Sea otro vector $\boldsymbol{y} \in \mathbb{R}^n$ que NO es solución de las ecuaciones normales. Definimos al vector diferencia

$$\boldsymbol{u} = \boldsymbol{y} - \boldsymbol{z}, \qquad \Rightarrow \qquad \boldsymbol{y} = \boldsymbol{z} + \boldsymbol{u}, \tag{4.56}$$

y evaluamos

$$\begin{aligned}f(\boldsymbol{y}) &= f(\boldsymbol{z}+\boldsymbol{u}) \\ &= (\boldsymbol{z}+\boldsymbol{u})^T\boldsymbol{A}^T\boldsymbol{A}(\boldsymbol{z}+\boldsymbol{u}) - 2(\boldsymbol{z}+\boldsymbol{u})^T\boldsymbol{A}^T\boldsymbol{b} + \boldsymbol{b}^T\boldsymbol{b} \\ &= \boxed{\boldsymbol{z}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{z}} + \boldsymbol{z}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{u} + \boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{z} + \boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{u} - \boxed{2\boldsymbol{z}^T\boldsymbol{A}^T\boldsymbol{b}} - 2\boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{b} + \boxed{\boldsymbol{b}^T\boldsymbol{b}} \\ &= f(\boldsymbol{z}) + \boldsymbol{z}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{u} + \boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{z} + \boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{u} - 2\boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{b} \\ &= f(\boldsymbol{z}) + 2\boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{z} + \boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{u} - 2\boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{b}\end{aligned}$$

$$= f(\boldsymbol{z}) + 2\boldsymbol{u}^T(\cancel{\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{z} - \boldsymbol{A}^T\boldsymbol{b}}) + \boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{u}$$

$$= f(\boldsymbol{z}) + \boldsymbol{u}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{u}$$

$$= f(\boldsymbol{z}) + \boldsymbol{v}^T\boldsymbol{v}, \tag{4.57}$$

donde hemos definido $\boldsymbol{v} = \boldsymbol{A}\boldsymbol{u}$. Como $\boldsymbol{v} \in \mathbb{R}^m$, $\boldsymbol{v}^T\boldsymbol{v} \geq 0$ y por lo tanto concluimos que

$$f(\boldsymbol{y}) \geq f(\boldsymbol{z}), \quad \forall \boldsymbol{y} \in \mathbb{R}^n \tag{4.58}$$

por lo tanto, concluimos que si un vector es solución de las ecuaciones normales entonces la función f tiene un mínimo en ese vector.

Los puntos (b) y (c) no los demostraremos porque son resultados conocidos de la teoría de ecuaciones normales, que ya hemos enunciado.

4.3.3. Nota sobre implementación en computadoras

El conjunto de soluciones del problema de cuadrados mínimos no suele obtenerse de las ecuaciones normales

$$\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^T\boldsymbol{b}, \tag{4.59}$$

ya que generalmente $\boldsymbol{A}^T\boldsymbol{A}$ es muy mal comportada cuando $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ está levemente mal condicionada. Eso se puede ver mejor con un ejemplo:

$$\boldsymbol{A} = \begin{pmatrix} 3 & 6 \\ 1 & 2{,}01 \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} 9 \\ 3{,}01 \end{pmatrix}. \tag{4.60}$$

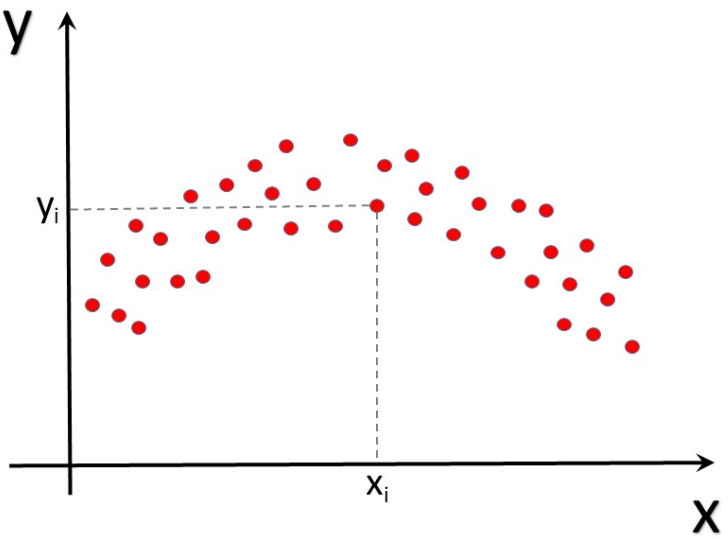
Si se resolviera el sistema con el método de Gauss, con 3 dígitos, obtendríamos $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, que coincide con la solución exacta. Sin embargo, si se usa aritmética de 3 dígitos para armar el sistema de ecuaciones normales asociado obtendríamos

$$\begin{pmatrix} 10 & 20 \\ 20 & 40 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 30 \\ 60{,}1 \end{pmatrix}, \tag{4.61}$$

que resulta ser un sistema incompatible. Por esta razón, las ecuaciones normales son evitadas usualmente en el cálculo numérico, aunque son reconocidas como idea teórica valiosa y dan sustento a ciertos métodos numéricos importantes. Para encontar la solución de cuadrados mínimos suelen usarle métodos alternativos. Uno de ellos es el de la factorización QR, que desarrollaremos oportunamente.

4.3.4. Ajuste polinómico

Consideremos un conjunto de puntos $\{(x_1, y_1), ..., (x_m, y_m)\}$ como el de figura:



Queremos aproximar la "nube de puntos" con el mejor polinomio (en el sentido de cuadrados mínimos) de grado $n-1$, con $n \leq m$. Como en el caso de la recta, supondremos que las abscisas son todas distintas entre sí. Sea

$$p(x) = a_0 + a_1 x + ... + a_{n-1} x^{n-1}, \tag{4.62}$$

de modo que tenemos n parámetos

$$\boldsymbol{\alpha} = \begin{pmatrix} a_0 & a_1 & a_2 & \dots & a_{n-1} \end{pmatrix}^T. \tag{4.63}$$

El error de cuadrados mínimos será

$$E_2(\boldsymbol{\alpha}) = \sum_{i=1}^{m} [p(x_i) - y_i]^2 = (\boldsymbol{A\alpha} - \boldsymbol{y})^T (\boldsymbol{A\alpha} - \boldsymbol{y}), \tag{4.64}$$

donde

$$\boldsymbol{A} = \begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \ddots & & & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{pmatrix}, \qquad \boldsymbol{\alpha} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \qquad \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}. \tag{4.65}$$

De acuerdo a lo que vimos en el teorema general de cuadrados mínimos, el vector solución $\boldsymbol{\alpha}$ de cuadrados mínimos es obtenido de la solución del sistema de ecuaciones normales asociada al sistema $\boldsymbol{A\alpha} = \boldsymbol{y}$. Como $r(\boldsymbol{A}) = n$ (ya que $\boldsymbol{A}$ es una matriz de Vandermonde con $n \leq m$ y, por lo tanto, sus columnas forman un conjunto linealmente independiente), sabemos que la solución de cuadrados mínimos es única y está dada por

$$\boldsymbol{\alpha} = (\boldsymbol{A}^T \boldsymbol{A})^{-1} \boldsymbol{A}^T \boldsymbol{y}. \tag{4.66}$$

* * *

No siempre la función subyacente es lineal con los parámetros. Supongan el caso en el que la aproximación de ajuste requiera ser de la forma

$$y = a_0\, e^{a_1 x}. \tag{4.67}$$

En este caso conviene trabajar con la forma logarítmica

$$\ln y = \ln a_0 + a_1 x. \tag{4.68}$$

Con esto volvemos a tener un problema lineal, aunque la aproximación que obtenga ya no será la de cuadrados mínimos del problema original.

Capítulo 5

Derivación e integración numérica

5.1. Fórmulas para aproximar a la derivada primera

Consideremos $n+1$ puntos distintos $\{x_0, x_1, ..., x_n\}$ pertenecientes a un intervalo I y sea además $f \in C^{n+1}[I]$. De acuerdo a lo que vimos con los polinomios de interpolación de Lagrange, podemos escribir

$$f(x) = \sum_{k=0}^{n} f(x_k)L_k(x) + \frac{f^{(n+1)}[\zeta(x)]}{(n+1)!}(x-x_0)(x-x_1)...(x-x_n), \qquad \zeta \in I. \tag{5.1}$$

Derivando $f(x)$ con respecto a x

$$f'(x) = \sum_{k=0}^{n} f(x_k)L'_k(x) + \frac{d}{dx}\Big[(x-x_0)(x-x_1)...(x-x_n)\Big]\frac{f^{(n+1)}[\zeta(x)]}{(n+1)!} +$$

$$+ \frac{(x-x_0)(x-x_1)...(x-x_n)}{(n+1)!}\frac{d}{dx}f^{(n+1)}[\zeta(x)]. \tag{5.2}$$

Si $x = x_j$, con $j = 0, 1, ..., n$

$$\boxed{f'(x_j) = \sum_{k=0}^{n} f(x_k)L'_k(x_j) + \frac{f^{(n+1)}[\zeta(x_j)]}{(n+1)!} \prod_{\substack{k=0 \\ k\neq j}}^{n} (x_j - x_k).} \tag{5.3}$$

Esta expresión se denomina "fórmula de $n+1$ puntos para aproximar $f'(x_j)$".

* * *

La fórmula más básica que podemos generar resulta de considerar tan sólo dos puntos ($n = 1$):

$$f'(x_0) \approx f(x_0)L'_0(x_0) + f(x_1)L'_1(x_0), \tag{5.4}$$

donde

$$L_0(x) = \frac{x-x_1}{x_0-x_1} \quad \Rightarrow \quad L'_0(x) = \frac{1}{x_0-x_1}, \tag{5.5}$$

$$L_1(x) = \frac{x-x_0}{x_1-x_0} \quad \Rightarrow \quad L'_1(x) = \frac{1}{x_1-x_0}. \tag{5.6}$$

Reemplazando en la expresión de más arriba

$$f'(x_0) \approx \frac{f(x_0)}{x_0-x_1} + \frac{f(x_1)}{x_1-x_0} = \boxed{\frac{f(x_1)-f(x_0)}{x_1-x_0}}. \tag{5.7}$$

El error de truncamiento es

$$E = \frac{f''(\zeta)}{2}(x_0 - x_1). \tag{5.8}$$

A partir de acá vamos a llamar h a la distancia entre puntos contiguos. Así, tenemos que $x_1 = x_0 + h$ y podemos escribir las expresiones para dos puntos como

$$\boxed{f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h},} \qquad \boxed{E = -\frac{f''(\zeta)}{2}h.} \tag{5.9}$$

Si $h > 0$ se llama "diferencia hacia adelante"(forward difference), mientras que si $h < 0$ "diferencia hacia atrás"(backward difference).

Ejemplo

Calcular con la fórmula anterior el valor de la derivada de $f(x) = \ln(x)$ en $x_0 = 1{,}8$. El valor exacto es $f'(x_0) = 0,\widehat{5}$.

$$h = 0{,}1 \quad f'(x_0) \approx 0{,}5407 \tag{5.10}$$

$$h = 0{,}01 \quad f'(x_0) \approx 0{,}5540 \tag{5.11}$$

$$h = 0{,}001 \quad f'(x_0) \approx 0{,}5554 \tag{5.12}$$

La aproximación resulta mejor a medida que achicamos el paso (el error es de $O(h)$).

Observaciones

• Las fórmulas más utilizadas para aproximar la derivada primera de una función son las que realizan evaluaciones de la función $f(x)$ en tres o cinco puntos.

• En condiciones normales la utilización de un polinomio de Lagrange de mayor grado nos permite obtener el resultado con menor error (esto no siempre se cumple). El precio que hay que pagar es, por un lado, un mayor número de evaluaciones de la función y, por el otro, un crecimiento del error de redondeo.

• La experiencia indica que los casos mencionados (cálculo con tres o cinco puntos) dan un buen balance entre el esfuerzo adicional que hay que hacer y la disminución del error que se produce.

* * *

Si en la fórmula general de aproximación de $f'(x_j)$ consideramos los puntos x_0, x_1 y x_2, con $x_1 = x_0 + h$ y $x_2 = x_0 + 2h$ y hacemos las cuentas, resulta:

$$f'(x_0) = \frac{1}{h}\left[-\frac{3}{2}f(x_0) + 2f(x_1) - \frac{1}{2}f(x_2)\right] + \frac{h^2}{3}f^{(3)}(\zeta_0), \tag{a}$$

$$f'(x_1) = \frac{1}{h}\left[-\frac{1}{2}f(x_0) + \frac{1}{2}f(x_2)\right] - \frac{h^2}{6}f^{(3)}(\zeta_1), \tag{b}$$

$$f'(x_2) = \frac{1}{h}\left[\frac{1}{2}f(x_0) - 2f(x_1) + \frac{3}{2}f(x_2)\right] + \frac{h^2}{3}f^{(3)}(\zeta_2). \tag{c}$$

Vamos a reescribir las expresiones (b) y (c). En (c) vamos a rebautizar lo que llamábamos x_2 como x_0. Así, lo que antes era x_1 ahora será $x_0 - h$ y x_0 pasará a ser $x_0 - 2h$. Por su parte, en

(b) rebautizamos a x_1 como x_0 y entonces el x_0 de antes será $x_0 - h$ ahora y x_2 pasará a ser $x_0 + h$. Resumiendo, las 3 expresiones anteriores pasarán a ser

$$f'(x_0) = \frac{1}{h}\left[-\frac{3}{2}f(x_0) + 2f(x_0+h) - \frac{1}{2}f(x_0+2h)\right] + \frac{h^2}{3}f^{(3)}(\zeta_0) \quad \text{(a)}$$

$$f'(x_0) = \frac{1}{h}\left[-\frac{1}{2}f(x_0-h) + \frac{1}{2}f(x_0+h)\right] - \frac{h^2}{6}f^{(3)}(\zeta_1) \quad \text{(e)}$$

$$f'(x_0) = \frac{1}{h}\left[\frac{1}{2}f(x_0-2h) - 2f(x_0-h) + \frac{3}{2}f(x_0)\right] + \frac{h^2}{3}f^{(3)}(\zeta_2) \quad \text{(f)}$$

(5.13)

Noten que las expresiones (a) y (f) son en realidad la misma fórmula si uno cambiara h por $-h$. Ambas nos dan la derivada en un punto extremo de un intervalo, utilizando dos puntos que se encuentran a un mismo lado. Por otro lado, la expresión (e) es distinta, ya que calcula la derivada en un punto utilizando la información en dos puntos vecinos que están uno a cada lado. En resumen, de las tres fórmulas que obtuvimos sólo dos son en realidad distintas, las expresiones (a) y (e). A éstas se les llama **fórmulas de tres puntos para aproximar la derivada primera de una función**.

Es importante reconocer que el error cometido es menor en la expresión (e). Esto proviene del hecho de usar información a ambos lados de x_0, mientras que en las otras fórmulas sólo lo hacemos de un solo lado.
De manera análoga puede derivarse la fórmula de 5 puntos para aproximar también la derivada primera, ahora con un error proporcional a h^4.

* * *

5.2. Aproximación de la derivada segunda

Es posible aproximar la derivada segunda, o de orden superior, usando polinomios de Taylor. Para ello, primero desarrollaremos la función $f(x)$ hasta orden 3 alrededor de x_0 y, luego, la evaluaremos en $x_0 + h$ y en $x_0 - h$. Así, obtenemos

$$f(x_0+h) = f(x_0)+f'(x_0)h+\frac{1}{2}f''(x_0)h^2+\frac{1}{6}f'''(x_0)h^3+\frac{1}{24}f^{(4)}(\xi_1)h^4, \qquad x_0 < \xi_1 < x_0+h \quad (5.14)$$

$$f(x_0-h) = f(x_0)-f'(x_0)h+\frac{1}{2}f''(x_0)h^2-\frac{1}{6}f'''(x_0)h^3+\frac{1}{24}f^{(4)}(\xi_2)h^4. \qquad x_0-h < \xi_2 < x_0 \quad (5.15)$$

Sumando miembro a miembro

$$f(x_0+h) + f(x_0-h) = 2f(x_0) + f''(x_0)h^2 + \frac{h^4}{24}\left[f^{(4)}(\xi_1) + f^{(4)}(\xi_2)\right], \quad (5.16)$$

y despejando $f''(x_0)$

$$f''(x_0) = \frac{1}{h^2}\big[f(x_0+h) - 2f(x_0) + f(x_0-h)\big] - \frac{h^2}{24}\left[f^{(4)}(\xi_1) + f^{(4)}(\xi_2)\right]. \quad (5.17)$$

Si $f^{(4)}(x)$ es continua en $(x_0 - h, x_0 + h)$, existe ξ tal que

$$f^{(4)}(\xi) = \frac{1}{2}\left[f^{(4))}(\xi_1) + f^{(4))}(\xi_2)\right]. \quad (5.18)$$

Por lo tanto, podemos calcular $f''(x_0)$ con la siguiente expresión

$$\boxed{f''(x_0) = \frac{1}{h^2}\big[f(x_0+h) - 2f(x_0) + f(x_0-h)\big] - \frac{h^2}{12}f^{(4)}(\xi).} \quad (5.19)$$

5.3. Errores y derivación numérica

Supongamos que estamos calculando $f'(x_0)$ mediante una de las aproximaciones que usa 3 puntos:

$$f'(x_0) = \frac{1}{2h}\Big[f(x_0+h) - f(x_0-h)\Big] - \frac{h^2}{6}f^{(3)}(\xi_1). \tag{5.20}$$

Como la computadora introduce errores de redondeo, vamos a explicitarlos de la siguiente forma:

$$f(x_0+h) = \widetilde{f}(x_0+h) + e_+, \tag{5.21}$$

$$f(x_0-h) = \widetilde{f}(x_0-h) + e_-, \tag{5.22}$$

donde e_+ y e_- son los errores de redondeo que aparecen al calcular $f(x_0+h)$ y $f(x_0-h)$, respectivamente. Es importante reconocer que en este caso estamos hablando de errores absolutos. Introduciendo estas expresiones en la fórmula para calcular la derivada

$$f'(x_0) - \frac{\widetilde{f}(x_0+h) - \widetilde{f}(x_0-h)}{2h} = \frac{e_+ - e_-}{2h} - \frac{h^2}{6}f^{(3)}(\xi_1). \tag{5.23}$$

Esta expresión nos dice que el error (miembro derecho) tiene una contribución por redondeo y otra por truncamiento. Si suponemos que el error de redondeo está acotado por $\varepsilon > 0$ y que $\left|f^{(3)}(\xi)\right| < M$, tenemos finalmente

$$\boxed{\left|f'(x_0) - \frac{\widetilde{f}(x_0+h) - \widetilde{f}(x_0-h)}{2h}\right| < \frac{\varepsilon}{h} + \frac{Mh^2}{6}.} \tag{5.24}$$

En consecuencia, si reducimos h disminuye el error de truncamiento pero aumenta el de redondeo. A esto se denomina **inestabilidad de la derivación numérica** y es una característica de las técnicas de aproximación de derivadas. Disminuyendo mucho h se puede producir un aumento muy importante del error de redondeo ya que éste va con ε/h.

5.4. Extrapolación de Richardson

La extrapolación de Richardson combina fórmulas de aproximación de bajo orden para obtener aproximaciones de mayor orden. Para poder utilizarla, es necesario que la técnica de aproximación tenga un error de forma predecible y que éste dependa de un parámetro, por ejemplo, el tamaño de la discretización (h).

Supongamos entonces que tenemos una fórmula de aproximación $N(h)$ (por ejemplo, la fórmula para la derivada primera o segunda de una función en un punto) para aproximar un valor desconocido M y que el error de truncamiento es

$$M - N(h) = K_1 h + K_2 h^2 + K_3 h^3 + ... \tag{5.25}$$

donde K_1, K_2,... son constantes desconocidas, pero que no dependen de h. En esta expresión veo que mi aproximación es $O(h)$, por lo tanto, para h pequeño podemos escribir

$$M - N(h) \approx K_1 h. \tag{5.26}$$

Lo que quiero hacer es generar una nueva aproximación $\widetilde{N}(h)$ a partir de $N(h)$, tal que

$$\widetilde{N}(h) \approx O(h^2), \tag{5.27}$$

o, si fuera posible, de mayor orden. Veamos cómo se puede lograr esto. Comencemos por escribir (5.25) como

$$M = N(h) + K_1 h + K_2 h^2 + K_3 h^3 + ... \tag{A}$$

Evaluemos ahora usando $h/2$:

$$M = N(h/2) + \frac{K_1}{2} h + \frac{K_2}{4} h^2 + \frac{K_3}{8} h^3 + ... \tag{B}$$

Si hacemos 2(B)-(A) es posible eliminar el término en h y obtener

$$M = \big[2N(h/2) - N(h)\big] + K_2 \left(\frac{h^2}{2} - h^2\right) + K_3 \left(\frac{h^3}{4} - h^3\right) + ...$$

$$= \big[2N(h/2) - N(h)\big] - \frac{1}{2} K_2 h^2 - \frac{3}{4} K_3 h^3 + ... \tag{5.28}$$

Por lo tanto, si llamamos al algoritmo inicial N como N_1, podemos identificar como algoritmo "mejorado" al siguiente

$$N_2(h) = 2N_1(h/2) - N_1(h) = \boxed{N_1(h/2) + \big[N_1(h/2) - N_1(h)\big]}, \tag{5.29}$$

que nos da una aproximación $O(h^2)$ para M:

$$M = N_2(h) - \frac{1}{2} K_2 h^2 - \frac{3}{4} K_3 h^3 + ... \tag{5.30}$$

* * *

Ejemplo

Sea la función $f(x) = e^x$ cuya derivada en $x = 0{,}5$ es

$$f'(0{,}5) = e^x|_{x=0{,}5} \approx 1{,}64872 \tag{5.31}$$

Para realizar el cálculo numérico vamos a usar el algoritmo

$$N_1(h) = \frac{e^{x+h} - e^x}{h}. \tag{5.32}$$

Implementándolo para $x = 0{,}5$ y aplicando la extrapolación de Richardson se obtiene:

$$N_1(0{,}1) \approx 1{,}73398 \tag{5.33}$$

$$N_2(0{,}1) \approx 1{,}69063 \tag{5.34}$$

* * *

Es posible continuar el proceso descripto más arriba. Para mayor claridad, repetimos acá la expresión (5.30) y la rebautizamos:

$$M = N_2(h) - \frac{1}{2} K_2 h^2 - \frac{3}{4} K_3 h^3 + ... \tag{C}$$

Ahora tomemos $h/2$ en esta expresión para obtener

$$M = N_2(h/2) - \frac{1}{8} K_2 h^2 - \frac{3}{32} K_3 h^3 + ... \tag{D}$$

Finalmente, hagamos 4(D)-(C) con el objetivo de eliminar el término en h^2

$$4M - M = 4\Big[N_2(h/2) - \cancel{\frac{1}{8}K_2h^2} - \frac{3}{32}K_3h^3 + ...\Big] - \Big[N_2(h) - \cancel{\frac{1}{2}K_2h^2} - \frac{3}{4}K_3h^3 + ...\Big] \quad (5.35)$$

con lo cual nos queda

$$3M = 4N_2(h/2) - N_2(h) + \frac{3}{8}K_3h^3 + ... \quad (5.36)$$

que podemos escribir como

$$\boxed{M = N_2(h/2) + \frac{N_2(h/2) - N_2(h)}{3} + \frac{1}{8}K_3h^3 + ...} \quad (5.37)$$

Así, es posible identificar

$$N_3(h) = N_2(h/2) + \frac{N_2(h/2) - N_2(h)}{3}, \quad (5.38)$$

como una fórmula de $O(h^3)$. Noten cómo esta fórmula no requiere del diseño de un nuevo algoritmo, tan sólo se basa en la implementación del algoritmo inicial combinando luego los resultados obtenidos para diferentes pasos h.

El proceso puede continuarse para obtener la siguiente expresión general:

$$\boxed{N_j(h) = N_{j-1}(h/2) + \frac{N_{j-1}(h/2) - N_{j-1}(h)}{2^{j-1} - 1},} \quad (5.39)$$

que nos provee una aproximación de $O(h^j)$. Esta expresión va a ser utilizada cuando integremos numéricamente.

5.5. Integración numérica

La idea básica es aproximar la integral definida

$$\int_a^b f(x)dx, \quad (5.40)$$

mediante una suma

$$\sum_{i=1}^{n} a_i\, f(x_i), \quad (5.41)$$

donde los x_i son puntos dentro del intervalo $[a, b]$ o (a, b), y los pesos a_i se obtienen como veremos a continuación. Esta manera de aproximar una integral se llama "cuadratura" y uno suele decir "integrando por cuadraturas..." cuando utiliza estas técnicas para integrar.

Vamos a utilizar los polinomios de Lagrange que definimos para interpolación:

$$f(x) = P_n(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!}\prod_{k=0}^{n}(x - x_k). \quad (5.42)$$

Integrando $f(x)$ entre a y b obtenemos

$$\int_a^b f(x)dx = \int_a^b P_n(x)dx + \frac{1}{(n+1)!}\int_a^b f^{(n+1)}(\xi(x))\prod_{k=0}^{n}(x - x_k)dx, \qquad a \le \xi(x) \le b. \quad (5.43)$$

Trabajemos con la primera integral:

$$\int_a^b P_n(x)dx = \int_a^b \sum_{k=0}^{n} f(x_k)L_k(x)dx = \sum_{k=0}^{n} f(x_k) \underbrace{\int_a^b L_k(x)dx}_{a_k}. \tag{5.44}$$

De acá puede verse que la idea para aproximar la integral de una función dada consiste en evaluar a la función en un cierto número de puntos y luego sumar los productos de estas evaluaciones por las integrales de los polinomios $L_k(x)$. Estas últimas integrales son las que nos brindan los pesos.

Veamos un caso particular. Sea el intervalo $[a, b]$ y consideremos dos puntos $x_0 = a$ y $x_1 = b$:

$$\int_a^b f(x)dx \approx f(x_0) \int_{x_0}^{x_1} L_0(x)dx + f(x_1) \int_{x_0}^{x_1} L_1(x)dx. \tag{5.45}$$

Haciendo las cuentas podemos verificar que

$$\int_{x_0}^{x_1} L_0(x)dx = \int_{x_0}^{x_1} \frac{x - x_1}{x_0 - x_1}dx = \frac{1}{2}(x_1 - x_0), \tag{5.46}$$

$$\int_{x_0}^{x_1} L_1(x)dx = \int_{x_0}^{x_1} \frac{x - x_0}{x_1 - x_0}dx = \frac{1}{2}(x_1 - x_0). \tag{5.47}$$

En consecuencia

$$\int_a^b f(x)dx \approx \frac{(x_1 - x_0)}{2}[f(x_0) + f(x_1)]. \tag{5.48}$$

¿Qué pasa con el error?

$$E = \int_{x_0}^{x_1} \frac{f''(\xi(x))}{2}(x - x_0)(x - x_1)dx, \qquad x_0 \leq \xi(x) \leq x_1. \tag{5.49}$$

Como $(x - x_0)(x - x_1)$ no cambia de signo en el intervalo $[x_0, x_1]$, podemos escribir la expresión anterior de la siguiente manera

$$E = \frac{f''(\xi_1)}{2} \int_{x_0}^{x_1} (x - x_0)(x - x_1)dx, \qquad x_0 \leq \xi_1 \leq x_1. \tag{5.50}$$

Calculando la integral y haciendo $x_1 - x_0 = h$ obtenemos para el error

$$E = -\frac{f''(\xi_1)}{12}h^3. \tag{5.51}$$

En resumen, cuando usamos dos puntos para aproximar la integral el algoritmo resultante es

$$\boxed{\int_a^b f(x)dx \approx \frac{h}{2}[f(x_0) + f(x_1)],} \tag{5.52}$$

que contiene un error de

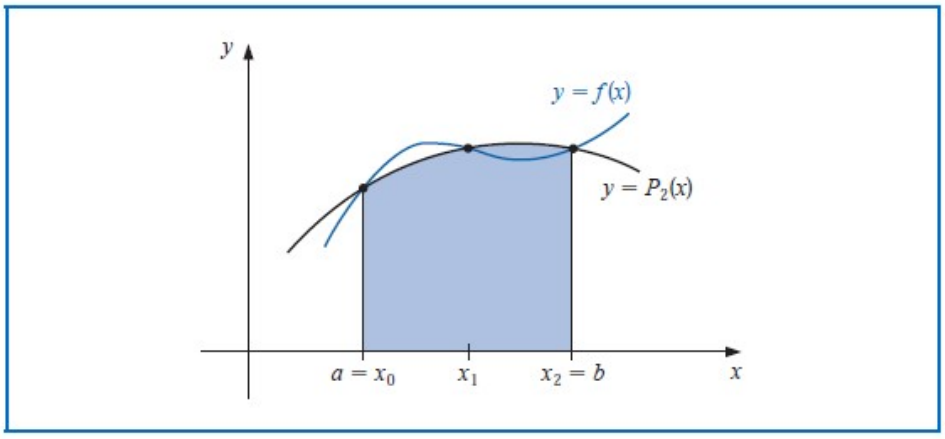
$$\boxed{E = -\frac{f''(\xi_1)}{12}h^3.} \tag{5.53}$$

Esta fórmula de aproximación se denomina **regla del trapecio** ya que si $f(x) > 0$, se aproxima la integral de $f(x)$ por el área del trapecio siguiente (realizaremos la figura en clase):

(*) Observen que esta integral es exacta si $f(x)$ es de la forma $ax + b$, ya que el error va con $f''(x)$.

* * *

Consideremos ahora otro caso, cuando tenemos 3 puntos.



Aproximamos la integral de la siguiente manera

$$\int_{x_0}^{x_2} f(x)dx \approx \sum_{k=0}^{2} f(x_k)a_k, \tag{5.54}$$

donde

$$a_0 = \int_{x_0}^{x_2} \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}dx, \tag{5.55}$$

$$a_1 = \int_{x_0}^{x_2} \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}dx, \tag{5.56}$$

y

$$a_2 = \int_{x_0}^{x_2} \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}dx. \tag{5.57}$$

Resolviendo esta integrales y haciendo nuevamente $x_1 - x_0 = h$ y $x_2 - x_1 = h$ resulta la expresión

$$\boxed{\int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}\left[f(x_0) + 4f(x_1) + f(x_2)\right],} \tag{5.58}$$

que se llama **regla de Simpson**. Puede probarse que

$$\boxed{E = -\frac{h^5}{90}f^{(4)}(\xi),} \tag{5.59}$$

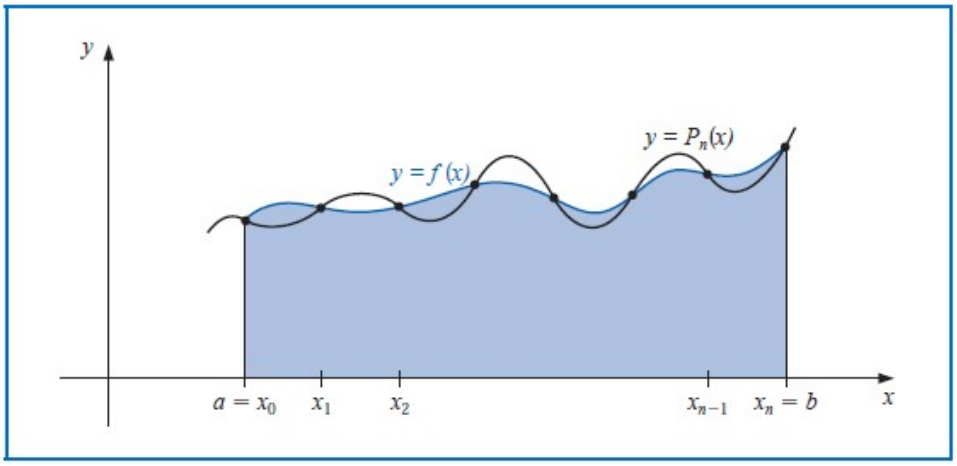
de modo que esta regla de cuadratura es exacta hasta polinomios de grado 3 (aunque estemos considerando 3 puntos, es decir, un polinomio interpolador de grado 2). Veremos luego por qué esto es así.

Tanto la regla del trapecio como la fórmula de Simpson son dos casos particulares de un conjunto de métodos llamados **fórmulas de Newton-Cotes**, que tienen validez cuando las abscisas

están equiespaciadas. Existen dos tipos de estas fórmulas, las ABIERTAS y CERRADAS. Se diferencian en que en las cerradas los extremos del intervalo de integración forman parte del conjunto de puntos a interpolar, mientras que en el caso de las abiertas no lo hacen.

Veamos entonces la forma general de las fórmulas cerradas de Newton-Cotes.

5.5.1. Fórmulas de Newton-Cotes cerradas



De acuerdo con lo que vimos al principio

$$\int_a^b f(x)dx \approx \int_{a=x_0}^{b=x_n} P_n(x)dx = \sum_{k=0}^{n} f(x_k)a_k, \tag{5.60}$$

donde

$$a_k = \int_{x_0}^{x_n} L_k(x)dx. \tag{5.61}$$

Sea $x_k = x_0 + hk$, con $k = 0, 1, ..., n$ y $h = (b-a)/n$. Hagamos el cambio de variable de x a s

$$x = a + hs \quad \Rightarrow \quad dx = h\,ds, \tag{5.62}$$

de manera tal que

$$\begin{aligned} x = x_0 \quad &\Rightarrow \quad s = 0, \\ x = x_n \quad &\Rightarrow \quad s = n. \end{aligned} \tag{5.63}$$

Veamos cómo escribir

$$L_k(x) = \frac{(x-x_0)(x-x_1)...(x-x_{k-1})(x-x_{k+1})...(x-x_n)}{(x_k-x_0)(x_k-x_1)...(x_k-x_{k-1})(x_k-x_{k+1})...(x_k-x_n)} \tag{5.64}$$

en términos de la nueva variable. Los factores del numerador se transforman en

$$\begin{aligned} x - x_0 &= a + hs - a = hs \\ x - x_1 &= a + hs - (a+h) = h(s-1) \\ x - x_2 &= h(s-2) \\ x - x_{k-1} &= h(s-k+1) \\ x - x_{k+1} &= h(s-k-1) \\ x - x_n &= h(s-n), \end{aligned} \tag{5.65}$$

mientras que los del denominador

$$\begin{aligned} x_k - x_0 &= a + hk - a = hk \\ x_k - x_{k-1} &= a + hk - [a + h(k-1)] = h \\ x_k - x_n &= a + hk - (a + hn) = h(k-n). \end{aligned} \tag{5.66}$$

Reemplazando en la expresión de a_k

$$a_k = \int_{x_0}^{x_n} L_k(x)dx = h \int_0^n \frac{s(s-1)(s-2)...(s-(k-1))(s-(k+1))...(s-n)}{k(k-1)(k-2)...(k-(k-1))(k-(k+1))...(k-n)} ds. \tag{5.67}$$

De modo que aquí se ve claramente que los pesos no dependen de la función que estamos integrando sino sólo del número de puntos de los que dispongo y de la abscisa que estoy considerando.

Podemos corroborar estas fórmulas tomando 3 puntos:

$$a_0 = h \int_0^2 \frac{(s-1)(s-2)}{(-1)(-2)} ds = \frac{h}{3}, \tag{5.68}$$

$$a_1 = h \int_0^2 \frac{(s-0)(s-2)}{(1-0)(1-2)} ds = \frac{4}{3}h, \tag{5.69}$$

$$a_2 = h \int_0^2 \frac{(s-0)(s-1)}{(2-0)(2-1)} ds = \frac{h}{3}, \tag{5.70}$$

con lo que obtenemos

$$\boxed{\int_{x_0}^{x_2} f(x)dx = h\left[\frac{1}{3}f(x_0) + \frac{4}{3}f(x_1) + \frac{1}{3}f(x_2)\right].} \quad \text{(regla de Simpson)} \tag{5.71}$$

Error de las fórmulas de Newton-Cotes cerradas

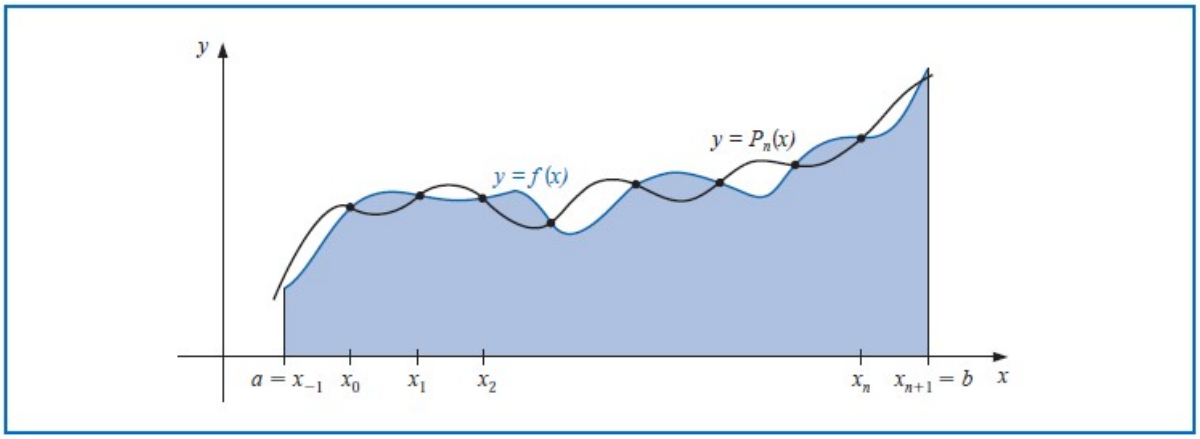
Es posible probar que el error se porta como:

$$\boxed{E = \frac{h^{n+3} f^{(n+2)}(\xi)}{(n+2)!} \int_0^n s^2(s-1)...(s-n)ds,} \quad \text{(para } n \text{ par)} \tag{5.72}$$

$$\boxed{E = \frac{h^{n+2} f^{(n+1)}(\xi)}{(n+1)!} \int_0^n s(s-1)...(s-n)ds.} \quad \text{(para } n \text{ impar)} \tag{5.73}$$

Entonces se ve aquí por qué la fórmula de Simpson tiene error de orden $O(h^5)$: como $n = 2$, debemos usar la fórmula del error para n par, que se porta mejor que la de los impares. En general, conviene usar fórmulas de cuadraturas con un número impar de puntos y, en caso de agregar, hacerlo de a pares y no de a uno.

5.5.2. Fórmulas de Newton-Cotes abiertas



Para el caso de las fórmulas abiertas tenemos $n+1$ puntos dentro del intervalo $[a,b]$, a los que le sumamos los extremos, $a = x_{-1}$ y $b = x_{n+1}$, de modo que

$$h = \frac{b-a}{n+2}, \quad x_0 = a+h, \quad x_1 = a+2h, \ldots \qquad x_n = a+(n+1)h = b-h. \tag{5.74}$$

Con esta notación, hacemos

$$\int_a^b f(x)dx = \int_{x_{-1}}^{x_{n+1}} f(x)dx \approx \sum_{k=0}^{n} a_k f(x_k) \tag{5.75}$$

donde, como antes

$$a_k = \int_{x_{-1}}^{x_{n+1}} L_k(x)dx. \tag{5.76}$$

Ojo, notar que los coeficientes no son iguales que los de las fórmulas cerradas.

Es posible probar que las fórmulas de error para este caso son muy parecidas a las cerradas, variando sólo el coeficiente de las mismas.

Hay que tener en cuenta que sobre la base de un número fijo de subintervalos en los que se divide al intervalo $[a,b]$ las fórmulas cerradas interpolan $(n+1)$ puntos y las abiertas $(n-1)$, por lo que la precisión de la cerrada es mayor, al precio de dos evaluaciones más de la función.

Ejemplos de fórmulas abiertas

Caso $n = 0$ (un punto, x_0):

$$\boxed{\int_{x_{-1}}^{x_1} f(x)dx = 2hf(x_0) + \frac{h^3}{3}f''(\xi),} \qquad x_{-1} \le \xi \le x_1, \tag{5.77}$$

que se denomina la **regla del punto medio**.

Caso $n = 1$ (dos puntos, x_0 y x_1):

$$\boxed{\int_{x_{-1}}^{x_2} f(x)dx = \frac{3}{2}h[f_0+f_1] + \frac{3h^3}{4}f''(\xi),} \qquad x_{-1} \le \xi \le x_2 \tag{5.78}$$

aquí se ve claro por qué no conviene agregar de a UN único punto. Notación: $f(x_0) \equiv f_0$, $f(x_1) \equiv f_1$.

Caso $n = 2$ (tres puntos, x_0, x_1 y x_2):

$$\boxed{\int_{x_{-1}}^{x_3} f(x)dx = \frac{4}{3}h[2f_0 - f_1 + 2f_2] + \frac{14h^5}{25}f^{(iv)}(\xi),} \qquad x_{-1} \leq \xi \leq x_3 \tag{5.79}$$

aquí se ve claro por qué no conviene agregar de a UN único punto.

5.6. Integración numérica compuesta

¿Qué sucede si el intervalo de integración es grande[1]? En este caso, si utilizamos las fórmulas de Newton-Cotes debemos usar polinomios interpoladores de grado alto, lo que necesariamente nos lleva a resultados imprecisos dada la naturaleza oscilatoria de los mismos (efecto Runge). Una alternativa es utilizar repetidamente las fórmulas de orden bajo en el intervalo.

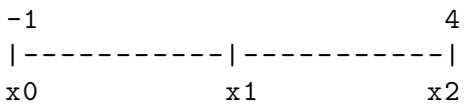
Ejemplo

$$\int_{-1}^{4} x^2 dx = 21.\widehat{6} \tag{5.80}$$

Usando la fórmula del trapecio obtenemos

$$\frac{1}{2}h\left[f(x_0) + f(x_1)\right] = \frac{5}{2}\left[(-1)^2 + 4^2\right] = \frac{5 \times 17}{2} = 42{,}5, \tag{5.81}$$

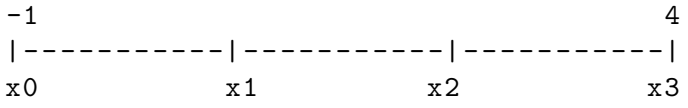
un resultado muy malo. Tomemos ahora la siguiente partición:



donde ahora $h = [4 - (-1)]/2 = 2{,}5$. Apliquemos la fórmula del trapecio entre x_0 y x_1 primero, y luego entre x_1 y x_2. Esto claramente puede hacerse, ya que

$$\begin{aligned}
\int_{x_0}^{x_2} f(x)dx &= \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx \\
&= \frac{h}{2}\left[f(x_0) + f(x_1)\right] + \frac{h}{2}\left[f(x_1) + f(x_2)\right] \\
&= \frac{h}{2}\left[f(x_0) + 2f(x_1) + f(x_2)\right] \\
&= \frac{2{,}5}{2}\left[(-1)^2 + 2 \times 1{,}5^2 + 16)\right] \\
&= 24{,}0625,
\end{aligned} \tag{5.82}$$

un resultado mucho mejor que el anterior. Si en lugar de considerar 3 puntos consideráramos 4:



[1]Hablar de un intervalo "grande" es algo impreciso. A lo que uno se refiere es a un intervalo tal que permita que la función muestre un comportamiento más complejo que el de un polinomio de bajo grado.

ahora $h = [4 - (-1)]/3 = 5/3$ y la solución aproximada que obtendríamos sería

$$\begin{aligned}\int_{x_0}^{x_2} f(x)dx &= \frac{5}{6}\left[f(x_0) + 2f(x_1) + 2f(x_2) + f(x_3)\right] \\ &= \frac{5}{6}\left[(-1)^2 + 16 + \frac{8}{9} + \frac{98}{9}\right] \\ &= 23{,}98 \end{aligned} \tag{5.83}$$

Para la extensión a cualquier número de puntos, damos el siguiente teorema.

Teorema. Sea $f \in C^2[a,b]$, $h = (b-a)/n$ y $x_j = a + jh$, con $j = 0, 1..., n$. Entonces, existe $\mu \in (a,b)$ tal que la regla del trapecio compuesta para n subintervalos es

$$\int_{a=x_0}^{b=x_n} f(x)dx = \frac{h}{2}\left[f(x_0) + 2\sum_{k=1}^{n-1} f(x_k) + f(x_n)\right] - \frac{b-a}{12}h^2 f''(\mu). \tag{5.84}$$

Para el caso de la regla de Simpson compuesta, tenemos este otro teorema.

Teorema. Sea $f \in C^4[a,b]$, n par, $h = (b-a)/n$ y $x_j = a + jh$, con $j = 0, 1..., n$. Entonces existe $\mu \in (a,b)$ tal que la regla de Simpson compuesta para n subintervalos es

$$\int_{a=x_0}^{b=x_n} f(x)dx = \frac{h}{3}\left[f(x_0) + 2\sum_{k=1}^{n/2-1} f(x_{2k}) + 4\sum_{k=1}^{n/2} f(x_{2k-1}) + f(x_n)\right] - \frac{b-a}{180}h^4 f^{(4)}(\mu). \tag{5.85}$$

Observen que en estos dos teoremas no hay mucho secreto, ya que lo único que hay que hacer es sumar los aportes en los subintervalos de las fórmulas de aproximación en ellas; lo mismo para el error, en este último caso hay que encontrar una expresión compacta que sale usando teoremas del Análisis I.

5.7. Estabilidad

Vimos que la derivación numérica es inestable, en el sentido que si $h \to 0$ el error de redondeo crece indefinidamente. ¿Pasa lo mismo con la integración numérica?

Analicemos lo que pasa utilizando la regla de Simpson compuesta aplicada a una función $f(x)$ en subintervalos de [a,b]. Haciendo lo mismo que antes, vamos a escribir

$$f(x_i) = \widetilde{f}(x_i) + e_i, \qquad i = 0, 1, 2, ..., n \tag{5.86}$$

donde $\widetilde{f}(x_i)$ es el resultado impreciso que nos da la computadora y e_i el error de redondeo. Así, el error de redondeo que se acumula por todos los subintervalos será

$$e(h) = \left|\frac{h}{3}\left[e_0 + 2\sum_{j=1}^{n/2-1} e_{2j} + 4\sum_{j=1}^{n/2} e_{2j-1} + e_n\right]\right|$$

$$\leq \frac{h}{3}\left[|e_0| + 2\sum_{j=1}^{n/2-1}|e_{2j}| + 4\sum_{j=1}^{n/2}|e_{2j-1}| + |e_n|\right]. \tag{5.87}$$

Podemos acotar los errores de redondeo, $|e_i| < \varepsilon$, $\forall i = 0, ..., n$ y así encontramos que

$$\begin{aligned} e(h) &\leq \frac{h}{3}\left[\varepsilon + 2\sum_{j=1}^{n/2-1}\varepsilon + 4\sum_{j=1}^{n/2}\varepsilon + \varepsilon\right] \\ &\leq \frac{h}{3}\left[\not{\varepsilon} + 2\left(\frac{n}{2} - \not{1}\right)\varepsilon + 4\frac{n}{2}\varepsilon + \not{\varepsilon}\right] \\ &= nh\varepsilon. \end{aligned} \tag{5.88}$$

Recordando que $h = (b-a)/n$ obtenemos finalmente

$$\boxed{e(h) \leq (b-a)\varepsilon.} \tag{5.89}$$

De modo que la integración numérica (lo probamos para Simpson, pero tiene validez general), es tal que un aumento en el número de puntos no implica un incremento en el error por redondeo.

5.8. Integración por Romberg

En este método, la regla del trapecio compuesta es mejorada con la aplicación de la técnica de extrapolación de Richardson. La idea consiste en calcular una secuencia de aproximaciones con la regla del trapecio compuesto, y luego usarla para generar el resultado mediante extrapolación. Lo que debe ser notado es que el integrando es evaluado solamente durante la utilización de la regla del trapecio, pero no cuando extrapolamos. Recordemos que la fórmula del trapecio compuesta es:

$$\int_{a=x_0}^{b=x_m} f(x)dx = \frac{h}{2}\left[f(a) + f(b) + 2\sum_{k=1}^{m-1} f(x_k)\right] - \frac{b-a}{12}h^2 f''(\mu), \tag{5.90}$$

donde $a < \mu < b$, $h = (b-a)/m$ y $x_j = a + jh$, $j = 0, 1, ..., m$. Sea ahora

$$m_1 = 1, \quad m_2 = 2, \quad m_3 = 4, ... \quad m_n = 2^{n-1}. \tag{5.91}$$

Luego, el paso correspondiente a m_k será

$$h_k = \frac{b-a}{m_k} = \frac{b-a}{2^{k-1}}. \tag{5.92}$$

Con esta notación, reescribimos la fórmula del trapecio compuesta como:

$$\int_a^b f(x)dx = \frac{h_k}{2}\left[f(a) + f(b) + 2\sum_{j=1}^{2^{k-1}-1} f(a + jh_k)\right] - \frac{b-a}{12}h_k^2 f''(\mu_k) \tag{5.93}$$

donde para cada k, corresponde un $a < \mu_k < b$.

Veamos cómo podemos escribir las expresiones para distintos valores de k. Para $k = 1$:

$$R_{1,1} = \frac{h_1}{2}[f(a) + f(b)], \tag{5.94}$$

donde $h_1 = b - a$. Para $k = 2$

$$R_{2,1} = \frac{h_2}{2}[f(a) + f(b) + 2f(a + h_2)], \tag{5.95}$$

con $h_2 = (b - a)/2 = h_1/2$. Haciendo unas cuentas sencillas podemos constatar que

$$\begin{aligned} R_{2,1} &= \frac{1}{2}\frac{h_1}{2}[f(a) + f(b) + 2f(a + h_2)] \\ &= \frac{1}{2}\left[R_{1,1} + h_1 f(a + h_2)\right]. \end{aligned} \tag{5.96}$$

De la fórmula general tenemos para $k = 3$:

$$\begin{aligned} \int_a^b f(x)dx &= \frac{h_3}{2}\left[f(a) + f(b) + 2\sum_{j=1}^{3} f(a + jh_3)\right] \\ &= \frac{1}{2}\left[\frac{h_2}{2}\Big(f(a) + f(b)\Big) + 2\frac{h_2}{2}\sum_{j=1}^{3} f(a + jh_3)\right] \\ &= \frac{1}{2}\left[\frac{h_2}{2}\Big(f(a) + f(b) + 2f(a + h_2)\Big) + h_2 \sum_{\substack{j=1 \\ j\neq 2}}^{3} f(a + jh_3)\right] \\ &= \boxed{\frac{1}{2}\left[R_{2,1} + h_2\Big(f(a + h_3) + f(a + 3h_3)\Big)\right].} \end{aligned} \tag{5.97}$$

En general, para m_k de la forma elegida, vale

$$\boxed{R_{k,1} = \frac{1}{2}\left[R_{k-1,1} + h_{k-1}\sum_{j=1}^{2^{k-2}} f(a + (2j - 1)h_k)\right].} \qquad k = 2, 3, ..., n. \tag{5.98}$$

De esta manera y por la forma elegida para partir el intervalo, podemos calcular una aproximación a la integral en el paso k en base al anterior. Bueno, como ya sabemos, la regla del trapecio converge lento. Ahora introduciremos Richardson. Es posible probar (aunque no es trivial) que, si $f \in C^\infty[a, b]$, vale que

$$\int_a^b f(x)dx - R_{k,1} = \sum_{j=1}^{\infty} K_j h_k^{2j} = K_1 h_k^2 + K_2 h_k^4 + ..., \tag{5.99}$$

donde los K_j son constantes que dependen de los valores de $f^{(2j-1)}(x)$, a y b, pero son independientes de h_k. Esto lo necesito para poder usar la fórmula de extrapolación de Richardson, puesto que la fórmula del error original no tiene la forma adecuada.
Consideremos

$$\begin{aligned} \int_a^b f(x)dx - R_{k+1,1} &= \sum_{j=1}^{\infty} K_j h_{k+1}^{2j} \\ &= \sum_{j=1}^{\infty} K_j \left(\frac{h_k}{2}\right)^{2j} \end{aligned}$$

$$= \frac{K_1 h_k^2}{4} + \sum_{j=2}^{\infty} K_j \left(\frac{h_k}{2}\right)^{2j} \tag{5.100}$$

Haciendo (5.99) $-$ 4×(5.100)

$$\int_a^b f(x)dx - R_{k,1} - 4\left(\int_a^b f(x)dx - R_{k+1,1}\right) = \cancel{K_1 h_k^2} + \sum_{j=2}^{\infty} K_j h_k^{2j} - 4\left(\cancel{\frac{K_1 h_k^2}{4}} + \sum_{j=2}^{\infty} K_j \left(\frac{h_k}{2}\right)^{2j}\right) \tag{5.101}$$

$$-3\int_a^b f(x)dx - R_{k,1} + 4R_{k+1,1} = \sum_{j=2}^{\infty} K_j \left(h_k^{2j} - \frac{h_k^{2j}}{4^{j-1}}\right) \tag{5.102}$$

$$\int_a^b f(x)dx - \left[R_{k+1,1} + \frac{R_{k+1,1} - R_{k,1}}{3}\right] = \sum_{j=2}^{\infty} \frac{K_j}{3}\left(\frac{h_k^{2j}}{4^{j-1}} - h_k^{2j}\right) \tag{5.103}$$

De modo que si tomo como aproximación a la integral la expresión

$$R_{k,2} = R_{k,1} + \frac{R_{k,1} - R_{k-1,1}}{3}, \tag{5.104}$$

tengo un error $O(h_k^4)$, y no de orden h_k^2 como antes de extrapolar. En general, voy a tener entonces

$$\boxed{R_{k,j} = R_{k,j-1} + \frac{R_{k,j-1} - R_{k-1,j-1}}{4^{j-1} - 1},} \tag{5.105}$$

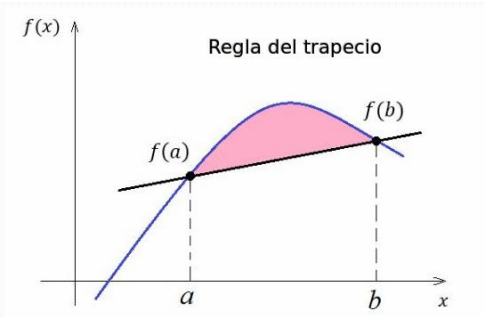
para tener un error de $O(h_k^{2j})$.

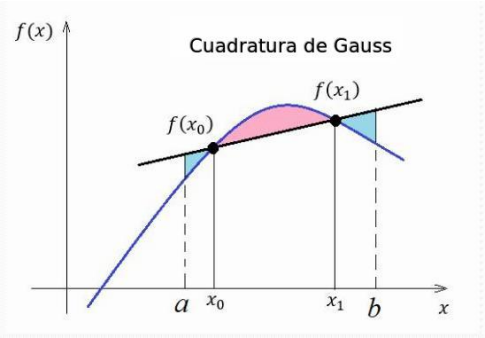
¿Cómo procedemos en la práctica? Si deseo extrapolar debo seguir la siguiente un camino que podemos representar con la siguiente tabla:

(completar con esquema)

Con esto terminamos la parte de integración utilizando las fórmulas de Newton-Cotes. Después retomaremos el tema al tratar las fórmulas adaptativas.

5.9. Cuadratura de Gauss





En la figura de la izquierda usamos la fórmula del trapecio, donde los puntos pasan por los extremos del intervalo. En la de la derecha usamos la misma cantidad de puntos pero no necesariamente tienen que estar equidistantes (válido para más de 2 puntos), sino que los fijamos utilizando algún criterio de conveniencia.

Como para otras cuadraturas, la idea de Gauss fue aproximar la integral definida

$$\int_a^b f(x)dx, \tag{5.106}$$

por medio de la sumatoria

$$\sum_{i=1}^{n} c_i f(x_i), \tag{5.107}$$

de modo tal que esta fórmula sea exacta para polinomios del máximo grado posible.

¿Cómo elegir las abscisas x_i y los pesos c_i para cumplir con este objetivo? En principio no existen restricciones para los pesos c_i, pero sí para los x_i, que deben pertenecer al intervalo de integración. La libertad para elegir los pesos y las abscisas nos da $2n$ grados de libertad (parámetros a fijar). Por otro lado, si consideramos como parámetros a los coeficientes de un polinomio, un polinomio de grado $2n-1$ poseerá $2n$ parámetros. Por lo tanto, resulta "razonable" pensar que éste, y no otro, va a ser el grado máximo posible.

Observen que esta forma de encarar el problema es más ambiciosa que las fórmulas de Newton-Cotes, ya que antes con $n+1$ puntos las fórmulas eran exactas, como máximo, para polinomios de grado $n+1$.

Como ejemplo, tomemos $n = 2$, entonces

$$\int_{a=-1}^{b=1} f(x)dx \approx c_1 f(x_1) + c_2 f(x_2) \tag{5.108}$$

Supongamos que buscamos los pesos y abscisas tales que la cuadratura sea EXACTA para polinomios de grado $2n-1 = 3$, es decir, para

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3. \tag{5.109}$$

Hagamos las cuentas para ver si esto es posible. Reemplazando $f(x)$ en el miembro derecho de la expresión de más arriba tenemos

$$\begin{aligned}\int_{a=-1}^{b=1} f(x)dx &= c_1(a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^3) + c_2(a_0 + a_1 x_2 + a_2 x_2^2 + a_3 x_2^3) \\ &= (c_1 + c_2)a_0 + (c_1 x_1 + c_2 x_2)a_1 + (c_1 x_1^2 + c_2 x_2^2)a_2 + (c_1 x_1^3 + c_2 x_2^3)a_3. \end{aligned} \tag{5.110}$$

El miembro izquierdo es fácil de calcular, de lo que resulta

$$2a_0 + \frac{2}{3}a_2 = (c_1 + c_2)a_0 + (c_1x_1 + c_2x_2)a_1 + (c_1x_1^2 + c_2x_2^2)a_2 + (c_1x_1^3 + c_2x_2^3)a_3. \tag{5.111}$$

Igualando los coeficientes de ambos miembros encontramos cuatro ecuaciones que deben cumplirse simultáneamente:

$$\begin{cases} c_1 + c_2 = 2 \\ c_1x_1 + c_2x_2 = 0 \\ c_1x_1^2 + c_2x_2^2 = \frac{2}{3} \\ c_1x_1^3 + c_2x_2^3 = 0 \end{cases} \tag{5.112}$$

Resolviendo el sistema no lineal encontramos que la solución es

$$c_1 = c_2 = 1, \qquad x_1 = -\frac{\sqrt{3}}{3}, \qquad x_2 = \frac{\sqrt{3}}{3}. \tag{5.113}$$

De modo que si hago

$$\boxed{\int_{-1}^{1} f(x)dx \approx f\left(-\frac{\sqrt{3}}{3}\right) + f\left(\frac{\sqrt{3}}{3}\right),} \tag{5.114}$$

esta integral va a ser exacta para polinomios de hasta grado 3 (aunque esté evaluando en sólo 2 puntos).

De igual manera podría continuar para los demás casos pero existe una forma más elegante de hacer las cosas, a través de los polinomios de Legendre.

5.9.1. Polinomios de Legendre

Estos polinomios pueden obtenerse, para un dado n, a partir de la fórmula de Rodrigues

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}\left(x^2 - 1\right)^n. \tag{5.115}$$

Así, es fácil encontrar que los primeros polinomios de Legendre son:

$$P_0(x) = 1 \tag{5.116}$$

$$P_1(x) = x \tag{5.117}$$

$$P_2(x) = x^2 - \frac{1}{3} \tag{5.118}$$

$$P_3(x) = x^3 - \frac{3}{5}x. \tag{5.119}$$

Como ven, se trata de polinomios "normales" pero que sin embargo, tienen ciertas propiedades que van a ser muy convenientes para la integración.

Propiedades

1. Para cada n, $P_n(x)$ es un polinomio mónico de grado n
2. Satisfacen la siguiente condición de ortogonalidad:

$$\int_{-1}^{1} P_n(x)P_m(x)dx = \begin{cases} 0 & n \neq m \\ \dfrac{2}{2n+1} & n = m \end{cases} \tag{5.120}$$

3. Como consecuencia de la propiedad anterior

$$\int_{-1}^{1} P(x)P_n(x)dx = 0, \tag{5.121}$$

si el grado de $P(x)$ es menor que n.

4. Los ceros de $P_n(x) \in (-1, 1)$ y son simétricos respecto del origen.

* * *

Dadas estas propiedades, podemos enunciar el siguiente teorema.

Teorema. Sean x_1, x_2,..., x_n las raíces del enésimo polinomio de Legendre $P_n(x)$ y sean c_i, con $i = 1, 2, ..., n$ números definidos por

$$c_i = \int_{-1}^{1} \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}\, dx = \int_{-1}^{1} L_i(x)dx. \tag{5.122}$$

Entonces, si $P(x)$ es un polinomio de grado menor que $2n$, se cumple que

$$\int_{-1}^{1} P(x)\, dx = \sum_{i=1}^{n} c_i P(x_i). \tag{5.123}$$

Demostración

Vamos primero a considerar a $P(x)$ un polinomio de grado menor que n. $P(x)$ puede escribirse como un polinomio de Lagrange de grado máximo $n-1$, con nodos en las n raíces del polinomio de Legendre $P_n(x)$. Por lo tanto,

$$P(x) = \sum_{j=1}^{n} P(x_j) \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k} \tag{5.124}$$

e integrando

$$\boxed{\int_{-1}^{1} P(x)dx} = \sum_{j=1}^{n} P(x_j) \int_{-1}^{1} \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}\, dx = \boxed{\sum_{j=1}^{n} P(x_j)c_j.} \tag{5.125}$$

En consecuencia, vale para polinomios de grado menor que n.

Consideremos ahora polinomios $P(x)$ de grado mayor o igual a n pero menor que $2n$. Podemos hacer

$$P(x) = Q(x)P_n(x) + R(x), \tag{5.126}$$

donde $Q(x)$ y $R(x)$ son polinomios de grado menor que n. Si evaluamos en las raíces de los polinomios de Legendre x_i, $i = 1, 2, ..., n$ tenemos

$$P(x_i) = Q(x_i)\cancel{P_n(x_i)} + R(x_i) = R(x_i). \tag{5.127}$$

Por ser el grado de $Q(x)$ menor que n, se cumple que

$$\int_{-1}^{1} Q(x)P_n(x)dx = 0. \tag{5.128}$$

Además, por lo que demostramos para polinomios de grado menor que n, vale que

$$\int_{-1}^{1} R(x)dx = \sum_{i=1}^{n} c_i R(x_i). \tag{5.129}$$

En consecuencia,

$$\boxed{\int_{-1}^{1} P(x)dx} = \cancel{\int_{-1}^{1} Q(x)P_n(x)dx} + \int_{-1}^{1} R(x)dx = \sum_{i=1}^{n} c_i R(x_i) = \boxed{\sum_{i=1}^{n} c_i P(x_i),} \tag{5.130}$$

y vale también para polinomios de grado menor que $2n$, que era lo que queríamos demostrar.

■

Observaciones

• Tanto los ceros de los polinomios de Legendre como los pesos c_i están tabulados, por lo que no es necesario calcularlos.

• Si sólo pudiéramos aplicar la cuadratura de Gauss a integrales definidas en el intervalo $[-1, 1]$ no sería de gran utilidad. Para integrales definidas en otros intervalos podemos hacer lo siguiente. Sea una integral

$$\int_{a}^{b} f(x)dx. \tag{5.131}$$

Proponemos el cambio de variables

$$x = \frac{1}{2}\Big[(b-a)t + a + b\Big] \quad \Rightarrow \quad dx = \frac{b-a}{2}dt, \tag{5.132}$$

de manera tal que para $t = -1$ tendremos que $x = a$ y en $t = 1$, $x = b$. Reemplazando en la integral

$$\int_{a}^{b} f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{1}{2}(b-a)t + \frac{b+a}{2}\right)dt. \tag{5.133}$$

En consecuencia

$$\int_{a}^{b} f(x)dx = \frac{b-a}{2}\sum_{i=1}^{n} c_i f\left(\frac{1}{2}(b-a)x_i + \frac{b+a}{2}\right), \tag{5.134}$$

y podemos usar la cuadratura de Gauss sobre cualquier intervalo $[a, b]$.

• El error de la cuadratura con $P_n(x)$ es

$$\int_{a}^{b} f(x)dx - \sum_{i=1}^{n} c_i f(x_i) = \frac{(b-a)^{2n+1}(n!)^4}{(2n+1)(2n!)^3} f^{(2n)}(\xi). \tag{5.135}$$

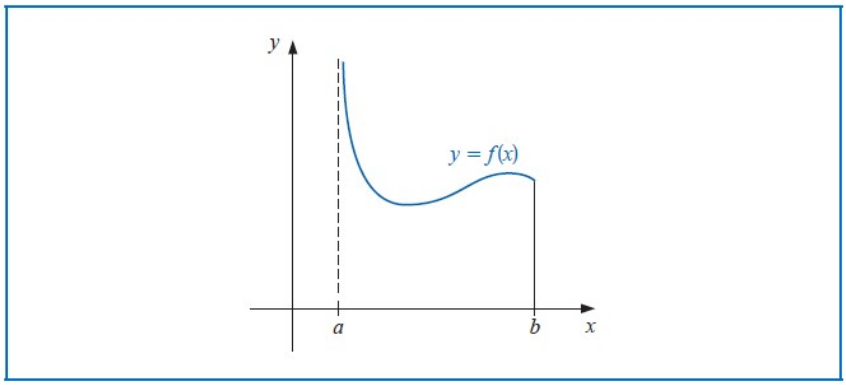
5.10. Cálculo de integrales impropias

Recordemos que hablamos de integrales impropias cuando:

1. dentro del intervalo de integración la función diverge, y/o
2. uno o más límites de integración es infinito.

Consideremos el caso en el que el integrando es singular en el límite inferior de integración

$$\lim_{x\to a^+} f(x) = \pm\infty, \tag{5.136}$$



Del análisis sabemos que

$$\int_a^b \frac{1}{(x-a)^p}dx, \tag{5.137}$$

converge sí y sólo sí $0 < p < 1$, en cuyo caso tenemos que

$$\left.\frac{(x-a)^{1-p}}{1-p}\right|_a^b = \frac{(b-a)^{1-p}}{1-p}. \tag{5.138}$$

Consideremos el caso en que $f(x)$ puede escribirse como

$$f(x) = \frac{g(x)}{(x-a)^p}. \tag{5.139}$$

Si $0 < p < 1$ y $g(x) \in C[a,b]$ entonces $f(x)$ es integrable en $[a,b]$. Para este caso particular es posible usar, por ejemplo, la regla de la Simpson compuesta, exigiendo además que $g(x) \in C^5[a,b]$ (esto es un poco más de lo que se exigía para Simpson). Si esto se cumple, podemos desarrollar a $g(x)$ en polinomio de Taylor hasta orden 4 alrededor de $x = a$

$$P_4(x) = g(a) + g'(a)(x-a) + \frac{g''(a)}{2!}(x-a)^2 + \frac{g'''(a)}{3!}(x-a)^3 + \frac{g^{(4)}(a)}{4!}(x-a)^4. \tag{5.140}$$

Utilizando $P_4(x)$, escribimos la integral original como

$$\int_a^b \frac{g(x)}{(x-a)^p}dx = \int_a^b \frac{g(x)-P_4(x)}{(x-a)^p}dx + \int_a^b \frac{P_4(x)}{(x-a)^p}dx, \tag{5.141}$$

donde segunda integral del miembro derecho puede calcularse analíticamente

$$\int_a^b \frac{P_4(x)}{(x-a)^p}dx = \sum_{k=0}^{4}\int_a^b \frac{g^{(k)}(a)}{k!}(x-a)^{k-p}\,dx = \sum_{k=0}^{4}\frac{g^{(k)}(a)}{k!(k-p+1)}(b-a)^{k-p+1}. \tag{5.142}$$

Este término contiene la principal contribución a la integral, especialmente cuando $P_4(x)$ se mantiene cerca de $g(x)$ en todo el intervalo $[a, b]$.

Calculemos ahora la otra integral

$$\int_a^b \frac{g(x) - P_4(x)}{(x-a)^p} dx. \tag{5.143}$$

Para ello, definimos

$$G(x) = \begin{cases} \dfrac{g(x) - P_4(x)}{(x-a)^p} & \text{si} \quad a < x \leq b \\ 0 & \text{si} \quad x = a. \end{cases} \tag{5.144}$$

Noten que $g^{(k)}(x)\Big|_{x=a} = P_4^{(k)}(x)\Big|_{x=a}$ para $k = 0, 1, 2, 3, 4$ y que, por lo tanto, $G(x) \in C^4[a, b]$. (Lo pueden comprobar haciendo el límite.) En consecuencia, se puede usar la fórmula de Simpson compuesta para aproximar esta integral.

El resultado final, que es la suma de los dos términos que analizamos, tiene el error de la regla de Simpson compuesta (ya que el otro término pudo resolverse en forma analítica).

Ejemplo

Integrar numéricamente

$$\int_0^1 \frac{e^x}{\sqrt{x}} dx, \tag{5.145}$$

utilizando la regla de Simpson compuesta con paso $h = 0{,}25$. Notamos la naturaleza impropia de la integral dada la singularidad en $x = 0$.

Para resolver la integral utilizando el método descripto más arriba primero debemos calcular el polinomio de Taylor de grado 4 del numerador

$$P_4(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}. \tag{5.146}$$

La integral que podemos calcular analíticamente es

$$\begin{aligned} \int_0^1 \frac{P_4(x)}{\sqrt{x}} &= \int_0^1 \left(x^{-1/2} + x^{1/2} + \frac{1}{2}x^{3/2} + \frac{1}{6}x^{5/2} + \frac{1}{24}x^{7/2} \right) dx \\ &= \lim_{a \to 0^+} \left[2x^{1/2} + \frac{2}{3}x^{3/2} + \frac{1}{5}x^{5/2} + \frac{1}{21}x^{7/2} + \frac{1}{108}x^{9/2} \right]_a^1 \\ &= 2 + \frac{2}{3} + \frac{1}{5} + \frac{1}{21} + \frac{1}{108} \approx 2{,}9235450. \end{aligned} \tag{5.147}$$

Ésta será la contribución **principal** a la integral que nos interesa calcular.

Definimos ahora la función $G(x)$

$$G(x) = \begin{cases} \dfrac{e^x - P_4(x)}{\sqrt{x}} & \text{para} \quad 0 < x \leq 1, \\ 0 & \text{para} \quad x = 0. \end{cases} \tag{5.148}$$

Tabulamos los valores aproximados de $G(x)$ para $h = 0{,}25$:

x	$G(x)$
0.00	0
0.25	0.0000170
0.50	0.0004013
0.75	0.0026026
1.00	0.0099485

y aplicamos la regla de Simpson compuesta

$$\int_0^1 G(x)dx \approx \frac{0{,}25}{3}\Big[0+4\times 0{,}0000170+2\times 0{,}0004013+4\times 0{,}0026026+0{,}0099485\Big] = 0{,}0017691. \tag{5.149}$$

Juntando ambos resultados

$$\int_0^1 \frac{e^x}{\sqrt{x}}dx \approx 2{,}9235450 + 0{,}0017691 = 2{,}9253141. \tag{5.150}$$

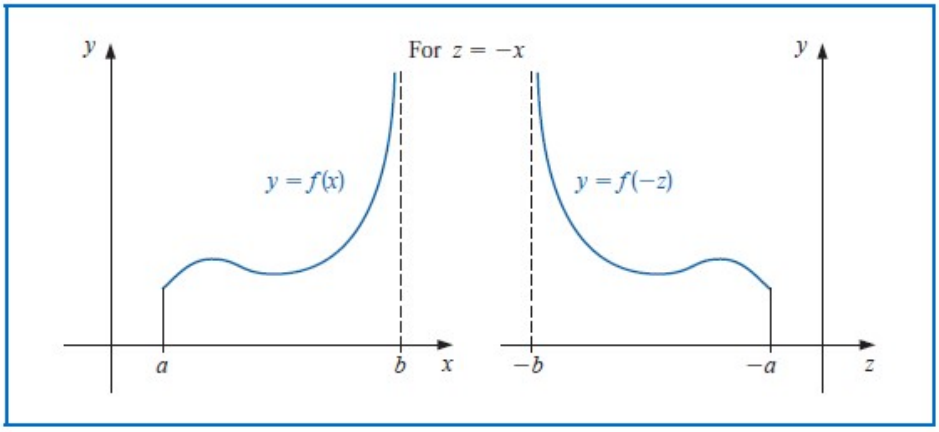
El error está dado por el de la regla de Simpson compuesta que usamos para integrar $G(x)$. Como $|G^4(x)| < 1$ en $[0,1]$ (no lo demostramos), podemos acotar el error por

$$\left|\frac{b-a}{180}h^4G^4(\xi)\right| < \frac{1-0}{180}\times 0{,}25^4 = 0{,}0000217. \tag{5.151}$$

* * *

• Para resolver un problema donde la singularidad esté en el límite superior de integración, se puede:

1. desarrollar por Taylor alrededor de b y repetir el proceso anterior, o
2. hacer
$$\int_a^b f(x)dx = \int_{-b}^{-a} f(-z)dz, \tag{5.152}$$
que tiene la singularidad en $-b$, así usamos el desarrollo previo.



Si la singularidad está en algún punto interior del intervalo, podemos plantear el problema como la suma de dos integrales, una con singularidad en el límite inferior y otra con singularidad en el límite superior.

• Finalmente, consideremos el caso

$$\int_{-\infty}^{\infty} f(x)dx \tag{5.153}$$

y supongamos que $f(x)$ es integrable. Escribimos la integral como

$$\int_{-\infty}^{\infty} f(x)dx = \int_{-\infty}^{c} f(x)dx + \int_{c}^{\infty} f(x)dx. \tag{5.154}$$

La segunda integral puede resolverse haciendo un cambio de variables

$$\int_{c}^{\infty} f(x)dx = \int_{0}^{1/c} t^{-2} f(1/t)dt, \tag{5.155}$$

que tiene una singularidad en $t = 0$. Esta integral puede resolverse como vimos antes.

* * *

Otra posibilidad para resolver este tipo de problemas es hacerlo a través de cuadraturas similares a la gaussiana, aunque ahora usando polinomios de Laguerre (en lugar de los de Legendre), ortogonales sobre el intervalo $[0, \infty)$, o bien polinomios de Hermite, que son ortogonales en el intervalo $(-\infty, \infty)$.

5.10.1. Integración vía polinomios de Laguerre

Los polinomios de Laguerre pueden obtenerse de la siguiente manera:

$$\mathcal{L}_n(x) = e^x \frac{d^n}{dx^n}(e^{-x}x^n), \quad n \geq 0, \tag{5.156}$$

o bien, a través del siguiente método recursivo:

$$\mathcal{L}_{-1}(x) = 0,$$

$$\mathcal{L}_0(x) = 1,$$

$$\mathcal{L}_{n+1}(x) = (2n + 1 - x)\mathcal{L}_n(x) - n^2\mathcal{L}_{n-1}(x), \qquad n \geq 0. \tag{5.157}$$

Para aproximar una integral de la forma

$$\int_{0}^{\infty} f(x)dx, \tag{5.158}$$

lo primero que hacemos es definir una función $\varphi(x) = e^x f(x)$ de manera tal que podemos escribir

$$I = \int_{0}^{\infty} f(x)dx = \int_{0}^{\infty} e^{-x}\varphi(x)dx. \tag{5.159}$$

Luego, basta aplicar la cuadratura de Laguerre que nos da

$$\boxed{I = \sum_{k=1}^{n} \alpha_k \varphi(x_k) + E,} \tag{5.160}$$

donde el error está dado por

$$E = \frac{(n!)^2}{(2n)!}\varphi^{(2n)}(\xi), \qquad 0 < \xi < \infty. \tag{5.161}$$

Los x_k, $k = 1, 2, ..., n$ son los ceros de $\mathcal{L}_n(x)$ y

$$\alpha_k = \frac{(n!)^2 x_k}{(\mathcal{L}_{n+1}(x_k))^2}. \tag{5.162}$$

5.10.2. Integración vía polinomios de Hermite

La cuadratura de Hermite permite aproximar integrales impropias de la forma

$$\int_{-\infty}^{\infty} f(x)dx. \tag{5.163}$$

Para lograr esto, necesitamos de los polinomios de Hermite que pueden hallarse por 2 caminos, haciendo

$$\mathcal{H}_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n}(e^{-x^2}), \quad n \geq 0, \tag{5.164}$$

o a través del siguiente método recursivo:

$$\begin{aligned} &\mathcal{H}_{-1}(x) = 0, \\ &\mathcal{H}_0(x) = 1, \\ &\mathcal{H}_{n+1}(x) = 2x\mathcal{H}_n(x) - 2n\mathcal{H}_{n-1}(x), \qquad n \geq 0. \end{aligned} \tag{5.165}$$

Definiendo una función $\varphi = f(x)e^{x^2}$, podemos escribir

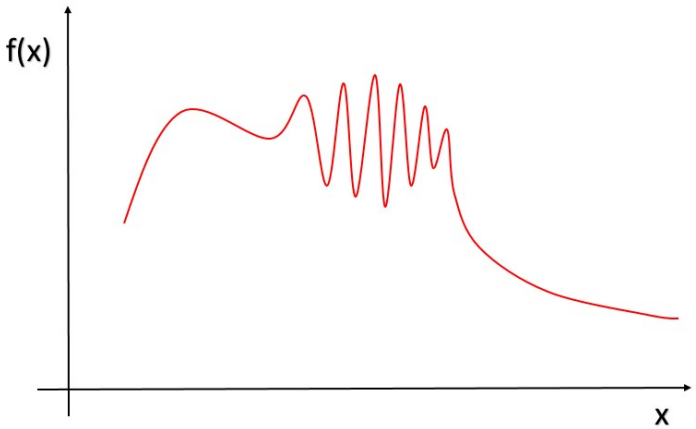
$$\int_{-\infty}^{\infty} f(x)dx = \int_{-\infty}^{\infty} e^{-x^2}\varphi(x)dx = \sum_{k=1}^{n} \alpha_k \varphi(x_k) + E, \tag{5.166}$$

donde x_k, $k = 1, 2, ..., n$ son los ceros de $\mathcal{H}_n(x)$ y

$$\alpha_k = \frac{2^{n+1}n!\sqrt{\pi}}{(\mathcal{H}_{n+1}(x_k))^2}, \qquad E = \frac{n!\sqrt{\pi}}{2^n(2n)!}\varphi^{(2n)}(\xi), \quad \xi \in \mathbb{R} \tag{5.167}$$

5.11. Métodos adaptativos

El objetivo de un algoritmo adaptativo es producir una aproximación a una integral dentro de una dada tolerancia ε utilizando pasos de integración NO UNIFORMES dentro del intervalo $[a, b]$. Un buen algoritmo adaptativo debe ser capaz de establecer automáticamente el tamaño del paso de integración incrementando la densidad de los nodos cuando el integrando presenta fuertes variaciones y espaciándolos más cuando éste varía con suavidad (ver figura). Veremos ahora, usando el caso particular de las fórmulas de Newton-Cotes y, dentro de ellas a Simpson, cómo es posible construir un método adaptativo para no sólo reducir el error de aproximación sino para escribir el mismo en una forma que no dependa de las derivadas de distinto orden del integrando. Naturalmente esta metodología puede extenderse a otras cuadraturas.



Consideremos la integral

$$\int_a^b f(x)dx, \tag{5.168}$$

y sea $[\alpha, \beta] \subseteq [a, b]$ un subintervalo cualquiera de $[a, b]$. Elijo este subintervalo, ya que el método está dirigido a intervalos de cualquier longitud. Sea

$$I_f(\alpha, \beta) = \int_\alpha^\beta f(x)dx, \qquad h = \frac{\beta - \alpha}{2}. \tag{5.169}$$

La aproximación de Simpson para esta integral es

$$S_f(\alpha, \beta) = \frac{h}{3}\Big[f(\alpha) + 4f(\alpha + h) + f(\beta)\Big], \tag{5.170}$$

y su fórmula del error

$$I_f(\alpha, \beta) - S_f(\alpha, \beta) = -\frac{h^5}{90}f^{(4)}(\eta), \qquad \alpha < \eta < \beta. \tag{5.171}$$

Ahora quiero una estimación del error SIN usar la derivada de $f(x)$. Usemos Simpson compuesto, ahora el paso será $h_1 = (\beta - \alpha)/4$. Así, los subintervalos sobre los que aplicaremos Simpson serán:

$$\left[\alpha, \frac{\alpha + \beta}{2}\right], \qquad \text{y} \qquad \left[\frac{\alpha + \beta}{2}, \beta\right], \tag{5.172}$$

y la integral se puede aproximar haciendo

$$S_f^2(\alpha, \beta) = \frac{h_1}{3}\left\{\left[f(\alpha) + 4f(\alpha + h_1) + f\left(\frac{\alpha + \beta}{2}\right)\right] + \left[f\left(\frac{\alpha + \beta}{2}\right) + 4f(\beta - h_1) + f(\beta)\right]\right\}, \tag{5.173}$$

con un error igual a

$$I_f(\alpha, \beta) - S_f^2(\alpha, \beta) = -\frac{h_1^5}{90}\left(f^{(4)}(\eta_1) + f^{(4)}(\eta_2)\right), \tag{5.174}$$

donde $\alpha < \eta_1 < (\alpha + \beta)/2$ y $(\alpha + \beta)/2 < \eta_2 < \beta$. Si suponemos (y esta suposición puede ser fuerte) que

$$f^{(4)}(\eta_1) \approx f^{(4)}(\eta_2) \approx f^{(4)}(\eta), \tag{5.175}$$

entonces podríamos escribir

$$I_f(\alpha, \beta) - S_f^2(\alpha, \beta) \approx -\frac{h^5}{2^5 \cdot 90}2f^{(4)}(\eta), \tag{5.176}$$

lo que reduce el error en un factor 16, comparado con el caso anterior. Comparando esta fórmula del error con la que obtuvimos para Simpson

$$\begin{aligned} I_f(\alpha, \beta) - S_f(\alpha, \beta) - \Big[I_f(\alpha, \beta) - S_f^2(\alpha, \beta)\Big] &\approx -\frac{h^5}{90}f^{(4)}(\eta) + \frac{h^5}{2^5 \cdot 90}2f^{(4)}(\eta) \\ -\Big[S_f(\alpha, \beta) - S_f^2(\alpha, \beta)\Big] &\approx -\frac{15}{16}\frac{h^5}{90}f^{(4)}(\eta). \end{aligned} \tag{5.177}$$

Despejando encontramos

$$\frac{1}{16}\frac{h^5}{90}f^{(4)}(\eta) \approx \frac{1}{15}\Big[\underbrace{S_f(\alpha, \beta) - S_f^2(\alpha, \beta)}_{\varepsilon_f(\alpha, \beta)}\Big]. \tag{5.178}$$

Por lo tanto, podemos escribir finalmente

$$\boxed{\left|I_f(\alpha,\beta) - S_f^2(\alpha,\beta)\right| \approx \frac{|\varepsilon_f(\alpha,\beta)|}{15}.} \tag{5.179}$$

Esta fórmula nos permite estimar el error cometido al aproximar la integral SIN usar las derivadas del integrando; esto la hace útil en algoritmos empaquetados. Usualmente conviene usar, en vez de 1/15, 1/10 para hacer una estimación más conservadora del error (estimamos que el error cometido es mayor).

Así, para asegurar una tolerancia ε para el error total cometido en $[a,b]$ basta con pedir que para cada subintervalo $[\alpha,\beta] \subseteq [a,b]$ se cumpla

$$\boxed{\frac{|\varepsilon_f(\alpha,\beta)|}{10} \leq \varepsilon\frac{\beta-\alpha}{b-a}.} \tag{5.180}$$

Si esta relación no se cumple, hay que achicar el intervalo $[\alpha,\beta]$. Este método usa el error de truncamiento para adaptar los subintervalos de integración.