Instituto Tecnológico Autónomo de México

Notas de Matemática Computacional y Cálculo Numérico Æ (Actuaría y Economía)

Primavera 2022

Pablo Castañeda

Vesrsión 1.0

16 de junio de 2022

Prólogo

Las presentes notas de estudio tienen como motivación ser un eje conductor para las clases tanto de Matemática Computacional, impartida en la licenciatura de Matemáticas Aplicadas, como de Cálculo Numérico Æ para las carreras de Actuaría y Economía, todas ellas parte de los planes de estudio actuales de Instituto Tecnológico Autónomo de México, ITAM.

Los temarios de ambas materias tienen una gran intersección. Su punto de vista en clase puede ser un poco diferente al enfocarse en demostraciones o aspectos poco conocidos del Cómputo Científico, en el caso de la carrera de Matemáticas Aplicadas, y ser más objetivo y extenso en el caso de la materia para Actuaría y Economía. Sin embargo, los temas comunes se pueden enriquecer con esta visión dupla.

El orden de los capítulos se encuentra en aquel que he usado durante varios semestres para Matemática Computacional; los temas que son exclusivos de esta materia se encuentran en azul medianoche. En turquesa oscuro se encuentran las áreas de estudio exclusivas del curso de Cálculo Numérico Æ. Como ejemplo de la extensión de este temario, basta ver que los capítulos sobre Números aleatorios (Capítulo 6) y sobre Aproximación de ecuaciones diferenciales (Capítulo 9) están redactadas para Actuaría y Economía. De hecho, este último capítulo, como área de estudio se revisa de manera breve en su amplia extensión. En el caso de Matemáticas Aplicadas, estos temas abarcan dos terceras partes del temario de Cálculo Numérico I; curso que es la continuación de Matemática Computacional.

Estos cursos al ser introductorios en el Cómputo Científico tienen la ventaja de ser compuestos por temas varios que puden ser estudiados de manera independiente y en el orden que se desee. El orden actual presupone que se podrán llamar conocimientos de capítulos previos. Como *vox populi* se dice que al resolver un problema numérico, necesariamente se pasa por un problema lineal, estudiados solo hasta en el Capítulo 7. Esto es de propósito para entender varias maneras de enfrentar un problema a través de la máquina. Tal vez sea cierto que el Álgebra Lineal y el Análisis Numérico desde el inicio del Cómputo Científico han sido grandes camaradas, empero, esto se remonta desde tiempos anteriores a las máquinas que conocemos actualmente.

En el caso del temario de Cálculo Numérico Æ partir rápidamente a estos problemas lineales es relevante, por ello, éste aparece un poco antes. Se recomienda estudiar los temas en el siguiente

orden:

1. Modelación, programación y la representación de números, Capítulos 1 y 2.
2. Diferenciación e integración numéricas, Capítulos 4 y 5.
3. Números aleatorios, Capítulo 6.
4. Soluciones de sistemas lineales, Capítulo 7.
5. Ajuste de funciones, Capítulo 8.
6. Localización de raíces y extremos locales, Capítulo 3.
7. Ecuaciones Diferenciales, Capítulo 9.

Los temas de interés en los Capítulos 3, 5 y 8 se ven mermados en Cálculo Numérico Æ. Parte de su estudio tiene que ver con casos específicos que surgen en problemas particulares y, que por tanto, pueden ser dejados de lado. Se recomienda a quienes son estudiantes de Actuaría y Economía que revisen estos temas por su cuenta, del mismo modo que a quienes estudien Matemáticas Aplicadas se les recomienda la lectura del Capítulo 6; el Capítulo 9 será estudiando de manera más amplia en el siguiente curso. La Sección 8.4 siempre se intenta introducir en el curso de Matemática Computacional.

Agradeceré cualquier comentario que tengas al respecto de estas notas, incrementar su calidad siempre será importante. Desde luego, los errores que en ellas persistan serán mi responsabilidad. Espero, sinceramente, que te sean muy útiles.

Pablo Castañeda
Ciudad de México,
5 de Septiembre de 2021.

Índice general

Capítulo 1

Modelaje y programación

En la Matemática Computacional así como en el análisis de problemas a través de una computadora, es importante decidir un lenguaje o programa que nos ayude con la metodología en cuestión. Hay muchos lenguajes como `Fortran`, `C/C++`, `Java` entre otros y algunos intérpretes interesantes como lo son R y MATLAB.

Una de las formas naturales de abordar los problemas es a través del álgebra lineal y por eso intérpretes como MATLAB (del inglés *MATrix LABoratory*) son muy utilizados, su contra parte en *software* libre OCTAVE también ha ganado terreno en los últimos años. Actualmente la variedad de aplicaciones y plataformas es muy grande, para mencionar algunas están `Python` y JUPYTER o `Julia` y MATHEMATICA. En el álgebra lineal, los vectores se escriben naturalmente como vectores columna y a veces nos conviene pensar en puntos como un arreglo horizontal. Esta dicotomía también existe en el mundo numérico, algunos como `Fortran` prefieren los vectores horizontales y `C/C++` los arreglos verticales en columnas. Ahora parece que esto no tiene consecuencias, pero podríamos incluso medir las dimensiones físicas del disco duro a través de métodos numéricos.

Hay que dar una pequeña introducción a `Python` y en especial al uso de SPYDER que podremos utilizar a lo largo de este curso; PYTHON de ahora en adelante. Algunas de sus partes son: *File explorer, IPython console, History log, Editor, ...* y la importancia del comando `help(·)`. En resumen y palabras vagas, podemos decir que `Python`, a través de SPYDER, es una gran calculadora con operaciones básicas (el producto `'*'`, la suma `'+'`, la substracción `'-'`, la división `'/'`, la potencia `'**'`, la división entera `'//'` y el residuo `'%'`, por ejemplo) y claro, tratando de problemas de álgebra lineal tiene las mismas operaciones entre matrices y vectores. Sin embargo, PYTHON incorpora, en el caso de arreglos, todas sus operaciones entrada a entrada; en el caso del producto o la división este es un invento útil y se conocen como las *operaciones de Hadamard*. Más adelante veremos cómo hacer el producto usual entre matrices, pues `A * B` nos retorna una matriz del mismo tamaño tanto de `A` como de `B`, sin importar si son rectangulares, como el resultado del producto entrada a entrada; no será el producto usual que conocemos `AB`.

Si quieres conocer los básicos de MATLAB, te recomiendo la "Breve introducción a Matlab" escrita en conjunto a Rodrigo Zepeda y Jimena Rodríguez Lebrija, en la siguiente liga:

http://departamentodematematicas.itam.mx/sites/default/files/breve-matlab.pdf

dentro de los documentos del Departamento Académico de Matemáticas del ITAM. Precisamente en el material de cursos.

En todo lenguaje, lo importante es conocer la sintaxis. Es bueno comenzar con el conocimiento de algunos elementos importantes de programación básica como lo son los ciclos, bucles o *loops* dados por comandos como `while` o `for` así como los condicionales `if`, `else` y `elif`, así como los operadores lógicos `and`, `or`, `not`, ... son muy útiles y hay que conocerlos. PYTHON tiene la ventaja de usarlos con palabras, con lo cual la lectura de un código es más sencilla. Para escribir $a \neq b$, será necesario `a != b`, donde la admiración funciona como la negación. Estos y otros ejemplos pueden ser encontrados en los códigos introductorios en el Apéndice B.

Ejercicio 1: Los siguientes dos ejercicios tienen como motivación aprender a crear ya sea un guión (*script* en inglés) o funciones

```
def NombreFuncion( in1, in2, ... ):
    return ( out1, out2, ... )
```

las cuales también son llamadas rutinas. El nombre del archivo puede ser el que quieras y puedes introducir varias funciones en el mismo documento. Recuerda estar en el directorio correspondiente para poder usarla.

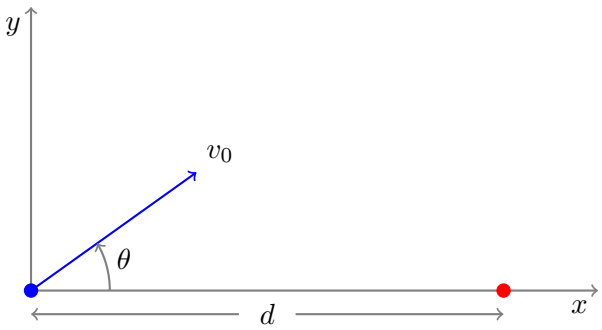
1. Construye una función en PYTHON que escriba la *sucesión de Fibonacci* hasta un n dado.
2. Escribir una rutina que coloque en una matriz los elementos del *triángulo de Pascal*.

Otros comandos importantes en la computación científica tienen que ver con la manipulación visual de los resultados, para ello, usaremos librerías como `matplotlib.pyplot`, por ejemplo.

1.1. Un sencillo problema de aplicación

Nos interesa el uso de la computadora para resolver problemas pero también es importante en este curso aprender a modelar las situaciones para poder entender qué queremos resolver.

Un ejemplo es el del *tiro de cañón*. Tenemos un cañón que dispara una bala a una velocidad v_0 constante, además se puede variar la inclinación de θ grados del disparo; sólo tenemos control sobre este ángulo. Queremos determinar cómo colocar el cañón para que la bala impacte un objetivo que se encuentra una cierta distancia d de la base del arma.



Las *ecuaciones diferenciales ordinarias* (EDO) nos ayudan a entender y modelar el "tiempo de retorno" de la bala, es decir, el tiempo que demora ésta en llegar nuevamente al nivel del suelo; estamos suponiendo que sale desde esta altura. Sabemos que la velocidad vertical es $v_0 \operatorname{sen}\theta$ y como el objeto siente la fuerza de la gravedad, tenemos el siguiente problema de valor inicial (PVI):

$$y''(t) \;=\; -g, \qquad y(0) \;=\; 0, \qquad y'(0) \;=\; v_0 \operatorname{sen}\theta,$$

donde g es la aceleración dada por la gravedad y las primas son derivadas respecto del tiempo. La solución se encuentra simplemente como

$$y(t) \;=\; v_0\, t \operatorname{sen}\theta \;-\; \frac{1}{2} g t^2.$$

(Muestra que satisfacen las tres ecuaciones arriba.)

Tenemos que $y(t) = 0$ tanto para el tiempo inicial t_0, como también para un tiempo positivo T, es decir,

$$t_0 \;=\; 0 \qquad \text{y} \qquad T \;=\; \frac{2v_0}{g} \operatorname{sen}\theta.$$

Ahora tenemos que la distancia recorrida en la horizontal es simplemente $x(t) = v_0\, t \cos\theta$ para cada tiempo. Lo que buscamos resolver es que al tiempo T se haya llegado a la distancia d, es decir, queremos resolver encontrar el ángulo θ tal que

$$x(T) \;=\; \frac{2v_0^2 \operatorname{sen}\theta \cos\theta}{g} \;=\; d$$

sea satisfecho. Sin embargo, nuestra variable es θ y está oculta en nuestra formulación anterior; esto es parte del modelaje también. Además, es más fácil conceptualmente tratar de llegar al cero que a un parámetro d que puede variar, es por ello que buscamos el valor que anula la siguiente función

$$f(\theta) \;=\; \frac{2v_0^2 \operatorname{sen}\theta \cos\theta}{g} \;-\; d \;=\; 0. \tag{1.1}$$

Este es un buen ejemplo, pues la ecuación $f(\theta) = 0$ tiene muchas cualidades dadas por la ecuación nolineal (1.1).

Discusión: Es claro que el modelo es una idealización. ¿cómo saber si el modelo es fiel a la realidad? Podemos tener alturas variables, obstáculos, contar la resistencia al viento, ... pero sin ir tan lejos, hay dos preguntas importantes:

1. ¿Podemos garantizar su solución?
2. Cuando existe, ¿es única esta solución?

Vemos que no es sencillo intentar encontrar un θ que resuelva el problema (1.1). Podría probarse a prueba y error. De algún modo, vemos intuitivamente que si tenemos dos valores θ_1 y θ_2 tales que $f(\theta_1) < 0$ y $f(\theta_2) > 0$, entonces debe de haber un θ^* entre ellos que satisfaga $f(\theta^*) = 0$; este es el principio del *método de la bisección* que veremos más adelante. Es notable que si θ^* es solución, entonces $\theta^* + 2\pi$ también lo es. Hay infinitas soluciones en este caso.

Esto se resuelve al restringir θ al intervalo $[0,\, 2\pi)$, ¿cuántas soluciones hay en este caso? Típicamente hay dos, pues si θ^* es solución, entonces $\pi/2 - \theta^*$ también lo es. ¿Cuál es la mejor solución? Esto dependerá de condiciones que por ahora no han sido introducidas, puede ser el tiempo de vuelo de la bala de cañón, puede depender de la altura por la que se pase para esquivar un obstáculo. Sin embargo, desde el punto de vista matemático y con la información dada hasta ahora por el problema, no podemos garantizar una solución mejor que otra.

Aquí es justo cuando podemos tener la primera moraleja del curso, que será recordada constantemente. Aunque podemos usar *métodos numéricos* para resolver (1.1), en general es mejor tratar analíticamente el problema que tenemos entre manos y, sólo después, atacar numéricamente una versión más sencilla. Por ejemplo, notando que $2\cos\theta \operatorname{sen}\theta = \operatorname{sen} 2\theta$, tenemos que (1.1) se transforma en

$$f(\theta) = \frac{v_0^2 \operatorname{sen} 2\theta}{g} - d = 0 \qquad \Longrightarrow \qquad \theta^* = \frac{1}{2} \operatorname{arc\,sen}\left(\frac{gd}{v_0^2}\right).$$

Vemos directamente que cuando $gd < v_0^2$ se satisface, entonces el problema tiene dos soluciones para el intervalo deseado.

Desde luego, al modelar el sistema o el problema con más complicaciones como la resistencia al viento o alturas variables, la ecuación no puede ser simplificada y es aquí donde haremos uso de la matemática computacional y de los métodos numéricos. No hay que perder de vista que las máquinas deben de idealizar a las matemáticas también, esto ocasiona la introducción de pequeños errores; es por ello que en este curso nos empeñaremos en entender estas discrepancias paras poder controlarlas de la mejor manera posible. Como veremos, las máquinas pueden darnos sorpresas y sólo tendremos que ir un poco más allá para entender cómo sobreponerlas.

Capítulo 2

Representación de números

Muchas veces hemos visto expresiones como `1.2345 -06` en una calculadora, tal vez `1.2345E-06` o versiones similares; lo importante es el significado que tiene esto. Simplemente es una manera compacta de escribir

$$1.2345 \times 10^{-6} \;=\; 0.0000012345.$$

Lo cual parece completamente inofensivo y ese es un error, no lo es.

Grosso modo una computadora o una calculadora (una *máquina* para resumir a ambas) solamente puede guardar un número finito de decimales en su memoria, por lo tanto su "recta numérica" es pequeña. Digamos que tenemos una computadora que expresa números con 4 cifras y hasta 2 cifras para el exponente (positivo o negativo). Así, el mayor número posible es $M = 9.999 \times 10^{99}$ y el menor -9.999×10^{99}, sin embargo, es fácil ver que no tenemos "todos" los reales del intervalo $[-M,\, M]$. Observemos también que el primer número a la derecha del cero es $m = 0.001 \times 10^{-99}$; sí, es pequeño, pero en realidad hemos perdido una infinidad de números entre el creo y m.

Esto desprende desde ya varias preguntas:

- ¿Cuál es una buena manera de expresar los números?
- ¿Qué números estamos realmente definiendo?
- ¿Qué debemos hacer con los números que no están dentro del conjunto elegido?
- ¿Qué errores esto provoca?
- $\cdots$

Arriba, en los ejemplos, y en nuestra vida cotidiana hemos usado comúnmente la base decimal. Así con la base $\beta = 10$ no suena mala idea usar el formato compacto de las calculadoras; un número puede escribirse como

$$a \;=\; f \cdot \beta^e,$$

donde f se conoce como la *fracción*, β es como hemos dicho la *base* y e es el *exponente*. Sin embargo, dado que $12.3 \times 10^2 = 1.23 \times 10^3 = 0.123 \times 10^4$, es importante decidir algo más. Hay dos opciones clásicas $f \in [\beta^{-1}, 1)$ o $f \in [1, \beta)$, el argot define el primer caso como "normalizado", usaremos este término sin importar cuál de los dos casos hemos tomado. Es decir, podemos decir que a está normalizado si $f = x_0.x_1 \cdots$ con $x_0 \neq 0$ y $x_k \in \{0, 1, \ldots, \beta - 1\}$ para $k = 0, 1, \ldots$.

2.1. El sistema binario

Dado que las máquinas suelen trabajar con `0` y `1`, es normal usar la base binaria con $\beta = 2$. Se necesita decidir el número de *bits* (o *bytes*) que se emplearán para guardar cada número. Recordando que un byte está conformado por 8 bits, cada uno de los cuales representan un `0` o un `1`.

Paréntesis: Antes de entrar en los detalles de los números que las computadoras usan, es bueno hacer una pausa para discutir un método que usaban los etíopes en el pasado y con el cual se pueden multiplicar cantidades grandes sabiendo únicamente duplicar un número, encontrar la mitad entera de otro y sumar. Digamos que queremos multiplicar 13 por 25, colocamos los números lado a lado, con el menor de ellos a la izquierda; observa el siguiente desarrollo e intenta entender qué es lo que sucede.

13	25
~~6~~	~~50~~
3	100
1	200

$$13 \times 25 \quad = \quad 25 + 100 + 200 \quad = \quad 325.$$

En la primera columna se van sacando las *mitades enteras* del número anterior, hasta llegar al número uno. En la segunda columna se van duplicando los números anteriores el mismo número de veces que se dividió en la primera columna. Se tachan todos los números pares de la primera columna y sus correspondientes parejas de la segunda columna. Finalmente se suman todos los números que sobraron en la segunda columna.

¿Qué es lo que sucede? ¿Por qué es que este método es tan efectivo? Simplemente porque es un acomodo especial del los números a multiplicar, por ejemplo,

$$13 \quad = \quad 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \quad = \quad 8 + 4 + 0 + 1 \quad = \quad (1101)_2.$$

Luego, $13 \times 25 = (8 + 4 + 1) \times 25 = 200 + 100 + 25$. Intenta otros productos.

2.2. Números enteros

El ejemplo de la multiplicación de los etíopes funciona como una buena manera de recordar la forma de escribir números en base binaria, pues hemos dejado implícito que

$$(a_n a_{n-1} \cdots a_1 a_0)_2 \;=\; a_n \cdot 2^n \;+\; a_{n-1} \cdot 2^{n-1} \;+\; \cdots \;+\; a_1 \cdot 2^1 \;+\; a_0 \cdot 2^0,$$

donde $a_k \in \{0, 1\}$ y normalmente se toma $a_n \neq 0$. En nuestro caso, usaremos n como un múltiplo de 8, el número de bits en un byte.

Los enteros en base binaria son indispensables para entender los números en la *aritmética de punto flotante*; de hecho, las fracciones en binario pueden ser vistas como uno de esos enteros divido por 2^k para algún k natural.

Dos conjuntos importantes en matemáticas y en el día a día son los números naturales y los enteros. En la máquina no los tendremos todos, solamente un conjunto muy grande de ellos. Usando únicamente un byte tenemos los siguientes números.

- "`unsigned integer`"

$$\begin{aligned} \texttt{source}(n) \;&:\; \boxed{b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0} \\ \texttt{value}(n) \;&=\; b_7(2^7) \;+\; b_6(2^6) \;+\; b_5(2^5) \;+\; b_4(2^4) \;+\; b_3(2^3) \;+\; b_2(2^2) \;+\; b_1(2^1) \;+\; b_0(2^0), \end{aligned}$$

que guarda valores entre 0 y 255.

- "`signed integer`"

$$\begin{aligned} \texttt{source}(n) \;&:\; \boxed{b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0} \\ \texttt{value}(n) \;&=\; b_7(-2^7) \;+\; b_6(2^6) \;+\; b_5(2^5) \;+\; b_4(2^4) \;+\; b_3(2^3) \;+\; b_2(2^2) \;+\; b_1(2^1) \;+\; b_0(2^0), \end{aligned}$$

que guarda valores entre -128 y 127.

Ejemplo: Escriba los números 13 y 19 en ambos formatos. Escriba -13 y -19 solamente para el segundo.

El paréntesis sobre los etíopes tiene la guía de un modo rápido de encontrar estos valores, allá se ha hecho ya con el número $13 = (1101)_2$, aquí lo haremos con el número 19.

$$\begin{array}{rrrrr} 0 & 1 & 2 & 4 & 9 \\ 2\,\overline{)\,1} & 2\,\overline{)\,2} & 2\,\overline{)\,4} & 2\,\overline{)\,9} & 2\,\overline{)\,19} \\ 1 & 0 & 0 & 1 & 1 \end{array}$$

Con lo cual vemos que $19 = (10011)_2$. Para colocar los valores guardados, simplemente agrega-

mos los ceros o unos indispensables para llenar el byte. Así, tenemos $\texttt{stored}(13) = 00001101$ y $\texttt{stored}(19) = 00010011$. Notamos que son indiferentes en cual de las dos notaciones estamos, el primer elemento es 0 y es este el que determina el signo o si el valor es superior a 127.

¿Qué sucede con los negativos? Lo más sencillo, dado que $b_7 = 1$, es sumar algo al número que nos garantice ser positivo. Siempre será este valor 2^7, pues $-13 + 2^7 = -13 + 128 = 115$. Ahora podemos proceder como antes:

$$\begin{array}{r|l} 0 & \\ \hline 2 & 1 \\ & 1 \end{array} \qquad \begin{array}{r|l} 1 & \\ \hline 2 & 3 \\ & 1 \end{array} \qquad \begin{array}{r|l} 3 & \\ \hline 2 & 7 \\ & 1 \end{array} \qquad \begin{array}{r|l} 7 & \\ \hline 2 & 14 \\ & 0 \end{array} \qquad \begin{array}{r|l} 14 & \\ \hline 2 & 28 \\ & 0 \end{array} \qquad \begin{array}{r|l} 28 & \\ \hline 2 & 57 \\ & 1 \end{array} \qquad \begin{array}{r|l} 57 & \\ \hline 2 & 115 \\ & 1 \end{array}$$

Hemos obtenido los valores de b_6 a b_0. Vemos entonces que

$$\begin{aligned} \texttt{source}(-13) \;:\; & \boxed{1}\boxed{1\ 1\ 1\ 0\ 0\ 1\ 1} && \text{(Se destaca el 1 del signo)} \\ \texttt{value}(-13) \;=\; & b_1(-2^7) + 1(2^6) + 1(2^5) + 1(2^4) + 0(2^3) + 0(2^2) + 1(2^1) + 1(2^0) \\ =\; & -128 + 64 + 32 + 16 + 2 + 1 \;=\; -128 + 115 \;=\; -13. \end{aligned}$$

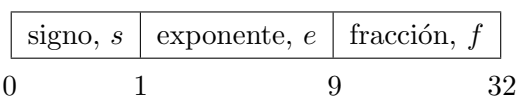
El ejemplo con -19 se deja como ejercicio.

2.3. Ampliando la recta numérica

Veamos ahora cómo aproximar los números reales en la máquina. El sistema de representación más usado en la actualidad es el formato de 64 bits de ANSI/IEEE 754-2008 Standard, de la Asociación Nacional Estadounidense de Estándares así como del Instituto de Ingenieros Eléctricos y Electrónicos. Este formato agregó algunas modificaciones pequeñas al establecido en 1985, el IEEE 754-1985. Actualmente se discute este formato con la implementación del IEEE 754-2019. Éste incorpora la "*half precision*" (con almacenamiento de 16 bits) y la "*quad precision*" (con almacenamiento de 128 bits). Como veremos con la *precisión sencilla* (SP) y la *precisión doble* (DP), estas modificaciones no alteran nuestra teoría sino solamente la cantidad y, por tanto, la precisión con la cual podemos representar los números dentro de una computadora.

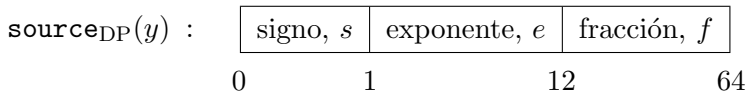
La *single precision standard* o *32-bit IEEE Standard*, que llamaremos de **precisión sencilla**, toma 4 bytes de la siguiente manera:

$\texttt{source}_{\text{SP}}(y)$:



donde se utiliza 1 bit para el signo $s = s(y)$, 1 byte para el exponente $e = e(y)$ y 23 bits para la fracción $f = f(y)$. En el caso de la **precisión doble** (*double precision standard* o *64-bit IEEE*

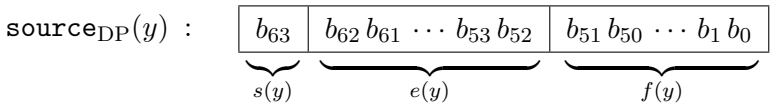
Standard) se toman 8 bytes de la siguiente manera:



donde ahora el signo continúa con un 1 bit, pero se tienen 11 bits para el exponente y 52 bits para la fracción.

Discusión: Observemos la cantidad de números y los números que cada una de las partes (signo, exponente y fracción) pueden representar en cada formato.

Veamos en detalle los números con precisión doble. Colocamos un subíndice para cada bit, el orden por conveniencia lo colocamos de izquierda a derecha y comenzando con el índice 0. La fuente la colocamos entonces como



donde $s(y) \in \{0, 1\}$ ayudará a definir el signo del número, $e(y) \in \{0, 1, \ldots, 2047\}$ es del **exponente parcial** (en inglés se conoce como *biased exponent*) y $f(y) \in \{0, 1, \ldots, 2^{52} - 1\}$ es la fracción que ayudará a construir la *mantisa*.[1]

Las combinaciones de estos números nos darán los números que la máquina usa en la *aritmética de punto flotante*. Tomando $E(y) = e(y) - 1023$ como el **exponente** de y, se define:

$$\mathtt{value}_{\mathrm{DP}}(y) \;=\; (-1)^{s(y)} \left\{ 1 + \frac{f(y)}{2^{52}} \right\} 2^{E(y)}, \tag{2.1}$$

donde la expresión entre las llaves se conoce como la **mantisa** de y. Usaremos indistintamente tanto $\mathtt{value}_{\mathrm{DP}}$ como $\mathtt{fl}_{\mathrm{DP}}$; en general es mejor $\mathtt{fl}_{\mathrm{SP}}$ y $\mathtt{fl}_{\mathrm{DP}}$ o `fl` si no hay confusión.

Nos centraremos en algunas "configuraciones" de estos bits, aquellos que se utilizan para los números en la recta numérica. No hablaremos sobre los símbolos y *números especiales*, nos basta saber que cuando $e(y) = 2047$ se está hablando de alguno de estos números como pueden ser $+\infty$ (`Inf`), $-\infty$ (`-Inf`) y `NaN` entre otros. La mayoría de los números en precisión doble utilizan la fórmula (2.1) tomando en cuenta lo siguiente.

1. Si $e(y) = 2047$, entonces y es un número especial.

2. Para $0 < e(y) < 2047$, y es un **número de punto flotante normalizado** y su valor está determinado por (2.1).

[1] Vemos que $2^{52} - 1 = 4,503''599,627'370,495$ es aproximadamente 4.5×10^{15}, ¡un número realmente grande!

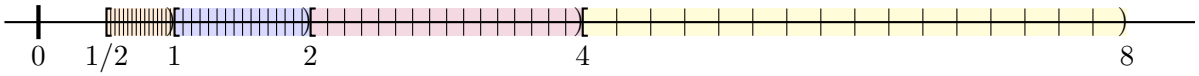
3. Si $e(y) = 0$ y $f(y) \neq 0$, entonces y es un **número de punto flotante desnormalizado** y tiene el valor

$$\texttt{value}_{\text{DP}}(y) \;=\; (-1)^{s(y)} \left\{ 0 + \frac{f(y)}{2^{52}} \right\} 2^{-1022} \;=\; (-1)^{s(y)} \frac{f(y)}{2^{1074}},$$

se pretende llamar ahora a los "desnormalizados" de "subnormalizados".

4. Si $e(y) = f(y) = 0$, entonces $\texttt{value}_{\text{DP}}(y) = \pm 0$.

Hay que observar que para un número normalizado con $E(y) = k$ tenemos que se satisface $2^k \leq |\texttt{value}_{\text{DP}}(y)| < 2^{k+1}$ y que para cada intervalo $[2^k, 2^{k+1})$ tenemos una cantidad fija de números; ésta es de 2^{52} números. Además, estos intervalos no son del mismo tamaño en la recta numérica, expandiéndose o contrayéndose en distintas regiones; son más densos cerca del cero y más esparsos para valores grandes.



Con ayuda de esta imagen es que queda claro el concepto del **épsilon de máquina** ε_{M} como la distancia desde el 1 hasta el siguiente número en la recta numérica. Así, después del 2 tenemos el $2 + 2\varepsilon_{\text{M}}$, o después del 4 el $4 + 4\varepsilon_{\text{M}}$. Piensa cuál es el siguiente número después del 3.

Ejemplo: Vamos a PYTHON para encontrar el valor de su épsilon de la máquina.

Si colocamos el valor de un número en notación científica haciendo uso de la base binaria, tenemos que

$$\texttt{value}(y) \;=\; S(y)\, 2^{E(y)}, \qquad \text{con} \qquad 1 \;\leq\; |S(y)| \;<\; 2,$$

donde $S(y)$ es la mantisa con signo del número en cuestión. Esto también nos da un indicio de que para escribir un número real en su forma de precisión doble, es importante escalarlo con un exponente para encontrar el valor de su mantisa.

Ejemplo: Considera el número $y = -1/10$ y escríbelo en su notación de precisión doble.

Primero notemos que

$$-\frac{1}{10} \;=\; (-1)^{1} \left(\frac{16}{10} \right) 2^{-4} \qquad \Longrightarrow \qquad s(y) \;=\; 1, \quad E(y) \;=\; -4 \;\; (\, e(y) \;=\; 1019\,),$$

pues $16/10 = 8/5 \in [1,\, 2)$. Con lo cual queremos encontrar un $f(y)$ entero tal que

$$\frac{8}{5} \;\approx\; 1 + \frac{f(y)}{2^{52}} \qquad \Longrightarrow \qquad f(y) \;\approx\; \left(\frac{8}{5} - 1 \right) 2^{52} \;=\; \frac{3 \cdot 2^{52}}{5} \;\notin\; \mathbb{N},$$

es decir, como $3 \cdot 2^{52}/5 = 2,702,159,776,422,297.51$ no es natural, tenemos que tomar $f(y)$ como el número entero más próximo, en este caso, $f(-1/10) = 2,702,159,776,422,298$.

Podemos escribir la fuente del número en binario al notar que

$$\begin{aligned} 1 &= (1)_2, \\ 1019 &= (1111111011)_2, \\ f(-1/10) &= (1001100110011001100110011001100110011001100110011010)_2, \end{aligned}$$

que aunque nos muestra exactamente cómo es guardado el número en la máquina es poco útil. Dejaremos estos números en su forma decimal y escribimos

$$\texttt{source}_{\mathrm{DP}}(-1/10) \ : \quad \begin{array}{|c|c|c|} \hline 1 & 1{,}019 & 2{,}702{,}159{,}776{,}422{,}298 \\ \hline \end{array}$$

Sin embargo, lo importante es ver si $\texttt{value}_{\mathrm{DP}}(-1/10)$ es realmente $-1/10$ o no. Tenemos

$$\begin{aligned} \texttt{value}_{\mathrm{DP}}\left(-\frac{1}{10}\right) &= (-1)^1 \left\{1 + \frac{2,702,159,776,422,298}{2^{52}}\right\} 2^{-4} \\ &= -0.1000000000000000055511151231257\cdots \ \neq \ -\frac{1}{10}, \end{aligned}$$

o mejor aún

$$\begin{aligned} \frac{1}{10} + \texttt{value}_{\mathrm{DP}}\left(-\frac{1}{10}\right) &= \left(\left\{1 + \frac{3}{5}\right\} - \left\{1 + \frac{2,702,159,776,422,298}{2^{52}}\right\}\right) 2^{-4} \\ &= -\frac{1}{180143985094819840} \ \approx \ -5.551115 \times 10^{-18}. \end{aligned}$$

Es claro que no son el mismo número, sin embargo, el error absoluto es menor que 6×10^{-18}, ¡tiene 17 cifras correctas!

El ejemplo anterior, muestra que es interesante saber qué números realmente están representados correctamente en la máquina. Por ejemplo, vemos que $\texttt{fl}_{\mathrm{DP}}(-1/10) \neq -1/10$, pero ¿qué números respetan la igualdad? ¿El signo es relevante?

Ejercicio 2: Descubre si los siguientes números están en DP: $-7/4$, $-1/6$, 0.002, $2/5$, 1.7, 3.125. Observa que no es necesario escribir la fuente, sólo saber si la fracción es un natural.

Ejercicio 3: Con la fuente dada para la aritmética de punto flotante en precisión simple, encuentra los análogos a las reglas 1 a 4 de la precisión doble. Repite el Ejercicio 2 para la precisión sencilla. Encuentra un número que se encuentre en DP pero no en SP.

2.4. La frontera interna y externa de la recta numérica

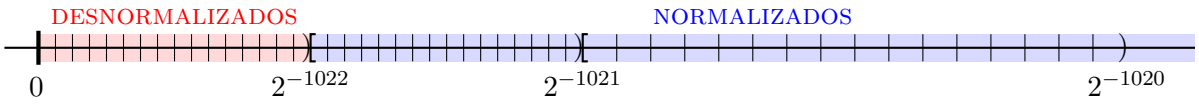
Para entender parte de la aritmética de punto flotante, es importante entender las limitaciones de la recta numérica, por ejemplo en precisión doble. Por un lado, vemos que el mayor exponente posible se da cuando $e = 2046$, con lo cual $E = 2046 - 1023 = 1023$, de manera semejante, la mayor fracción es $f = 2^{52} - 1$ y con el signo $s = 0$, tenemos el mayor valor dado por

$$M_{\text{DP}} \;=\; (-1)^0 \left\{1 + \frac{2^{52}-1}{2^{52}}\right\} 2^{1023} \;\approx\; 2^{1024} \;\approx\; 1.7977 \times 10^{308},$$

al usar la fórmula dada por (2.1). Usando esta misma fórmula, vemos que podemos construir un número pequeñísimo dado por

$$(-1)^0 \left\{1 + \frac{0}{2^{52}}\right\} 2^{-1022} \;=\; 2^{-1022} \;\approx\; 2.2250 \times 10^{-308},$$

sin embargo, es un número normalizado.



En el dibujo arriba, se ve que de este modo aún siendo un número pequeño, la distancia hacia el cero se ve grande. Esta es la razón de la aparición de los números desnormalizados, así, tenemos

$$m_{\text{DP}} \;=\; (-1)^0 \left\{0 + \frac{1}{2^{52}}\right\} 2^{-1022} \;=\; 2^{-1074} \;\approx\; 4.9407 \times 10^{-324},$$

como el menor número positivo, generando una sensación gráfica de una buena distribución.

Cuando tenemos un número fuera de estos límites, se debe decidir qué hacer con ellos. Por ejemplo si $0 < y < m_{\text{DP}}$, cuando es próximo del menor positivo podemos "redondearlo" hacia él, pero si es muy pequeño y más próximo de cero, habrá que tomarlo como éste. Dado que dividir por cero puede provocar errores o multiplicar un número por cero siempre da un resultado nulo, anular un número $y > 0$ puede ser problemático y por tanto denotamos estas acción como hacer un ***underflow***. Cuando tenemos $y \gg M_{\text{DP}}$, se acostumbra tomar este número como `Inf`, el realizar esta acción lleva el nombre de hacer un ***overflow***. Ambos ejemplos pueden llevar a problemas difíciles de contornar.

Ejemplo: Para ilustrar el *sobreflujo/subflujo*, imaginemos que nuestros números tienen solamente exponentes $e(y) \in \{-100, \ldots, 0, \ldots, 100\}$. Digamos además que queremos utilizar el renombrado Teorema de Pitágoras para evaluar la hipotenusa de distintos triángulos rectángu-

los. Simplemente, dados los catetos a y b, se tiene que la hipotenusa es

$$c \;=\; \sqrt{a^2 \;+\; b^2}\,,$$

fácil. Vemos que tanto $a = 10^{60}$ como $b = 1$ son números en nuestra aritmética.

Dado que $a^2 = (10^{60})^2 = 10^{120}$ y el exponente es mayor a 100, entonces lo tomamos como `Inf`; aquí comienzan los problemas. Es claro que $b^2 = 1$, pero $a^2 + b^2 = \texttt{Inf} + 1 = \texttt{Inf}$, finalmente como $\sqrt{\texttt{Inf}} = \texttt{Inf}$, tenemos que $c = \texttt{Inf}$. ¡Nada más alejado de la realidad! Hemos producido un *overflow* que nos da todos los problemas. Es claro que c es un número grande, pero aún así está lejos de ser próximo del mayor número de nuestra aritmética, este es un ejemplo donde debemos de pensar cómo solucionar las limitaciones que tiene la máquina.

La manera de resolver este problema radica en no dejar que los números al cuadrado se salgan de nuestra recta numérica, veamos que la fórmula de Pitágoras es equivalente a lo siguiente

$$c \;=\; s\sqrt{\left(\frac{a}{s}\right)^2 \;+\; \left(\frac{b}{s}\right)^2}, \qquad \text{con} \qquad s \;=\; \text{máx}\{|a|,\, |b|\}.$$

En este caso tenemos

$$\begin{aligned} c \;&=\; 10^{60}\sqrt{\left(\frac{10^{60}}{10^{60}}\right)^2 \;+\; \left(\frac{1}{10^{60}}\right)^2} \;=\; 10^{60}\sqrt{1^2 \;+\; (10^{-60})^2} \\ &=\; 10^{60}\sqrt{1 \;+\; 0} \;=\; 10^{60}, \end{aligned}$$

donde ahora se ha cometido un *underflow* al tomar $(10^{-60})^2 = 10^{-120} \mapsto 0$, que tiene consecuencias menos catastróficas. De hecho, si lo pensamos, tenemos un triángulo larguísimo, para el cual una hipotenusa de la misma dimensión que el cateto mayor es una óptima aproximación dadas las limitaciones de la máquina usada.

Además de los errores que ocurren al salirse de los límites de la recta numérica, se tienen errores clásicos involucrando el ε_{M}. Digamos que como en el ejemplo anterior, nuestros números tienen un exponente de -100 a 100 y además sólo usamos cuatro dígitos para la mantisa. Sin pérdida de generalidad, los números los escribimos como

$$\texttt{fl}(x) \;=\; (-1)^s\, d_1.d_2d_3d_4 \times 10^E,$$

con d_1, d_2, d_3, $d_4 \in \{0,\, 1,\, \ldots,\, 9\}$ pero $d_1 \neq 0$ y con $E \in \{-100,\, \ldots,\, 0,\, \ldots,\, 100\}$.

Ahora realizamos la ingenua operación

$$S \;=\; 10,000 \;+\; \sum_{i=1}^{1000} 1,$$

que es claro tiene como resultado $11,000 = 1.1\times10^4$ que resulta ser un número en nuestra aritmética. Recordemos que la suma se hace de manera binaria, es decir de dos en dos. Así, esta suma resulta ser algo como

$$S \;=\; 10,000 + \underbrace{1 \;+\; 1 \;+\; \cdots \;+\; 1 \;+\; 1}_{1000 \text{ veces}},$$

que nos lleva a un proceso larguísimo, pero con el resultado erróneo de $10,000$. ¿Por qué? Simplemente, vemos que $10,000 + 1 = 10,001 = 1.0001 \times 10^4$, luego en la aritmética escogida, se tomará el valor $10,000$ y esta operación se repite mil veces sin alterar una sola vez la suma. En cambio, si alteramos el orden de la suma de uno a mil con el diez mil, obtendremos el valor correcto.

Ejercicio 4: Encuentra dos números naturales N y M tales que la suma $M + \sum_{i=1}^{N} 1$ sea M o $M + N$ según el orden. Usa la aritmética de DP de PYTHON para este ejercicio.

2.5. Tipos de errores en el análisis

Hasta este momento hemos decidido hacer el redondeo normalmente, es bueno hacer una pausa y pensar las ventajas reales de este procedimiento sobre el *truncamiento*. Consideremos una aritmética con 4 dígitos de precisión, así

$$1.2345678\cdots \longmapsto \begin{cases} 1.234, & \text{cortando} \quad (\textit{truncation}) \\ 1.235, & \text{redondeando } (\textit{rounding}) \end{cases}.$$

Estas son las dos operaciones comunes, ¿cuál es mejor? ¿por qué? Normalmente responderíamos a partir del error producido, esto nos da pie para mencionar los dos errores que usaremos a lo largo de este curso.

1. Error absoluto.

Tomamos $a = \bar{X}.XXXY$ y lo aproximamos por $b = \bar{X}.XXZ$, donde X, $\bar{X}$, Y, $Z \in \{0, 1, \ldots, 9\}$ y $\bar{X} \neq 0$ son números cualesquiera. Para el redondeo el último X en a se transforma en $Z = X + 1$ si $Y \geq 5$, lo "hacemos para arriba". Cuando $Y < 5$ "para abajo", es decir, Z es igual a este último X de a. Así el **error absoluto** es

$$\mathrm{Err}(a,\, b) \;=\; |b \,-\, a| \;\leq\; 5 \cdot 10^{-4} \;=\; \frac{1}{2}10^{-3}.$$

Fácilmente se ve que al redondear con t dígitos lo que se tiene es que

$$\mathrm{Err}(a,\, b) \;=\; |b \,-\, a| \;\leq\; 5 \cdot 10^{-t} \;=\; \frac{1}{2}10^{-t+1}.$$

En el caso de truncar, este error es 10^{-t+1} y se deja como ejercicio; es el doble en magnitud.

Observación: Hay que notar que cuando los números se encuentran en base binaria, el error absoluto es

$$\mathrm{Err}(a,\, b) \;=\; |b \,-\, a| \;\leq\; \frac{1}{2}2^{-t+1} \;=\; 2^{-t}.$$

Además, en el caso del redondeo en binario resulta extremadamente fácil saber cuándo hacerlo o no; no se agrega nada si la cifra en cuestión es 0 y solamente en el caso del dígito ser 1 es que esta se realiza. Por ejemplo, con cinco dígitos

$$\begin{aligned}(\underline{1.010}101)_2 \quad &\longmapsto \quad (1.0101)_2\\ (\underline{1.010}011)_2 \quad &\longmapsto \quad (1.0101)_2\end{aligned}$$

Encuentra ambos errores absolutos al restar los números binarios en la vertical. Recuerda que ahora al "prestar" un 1, este vale $(10)_2 = 2$.

La deficiencia del error absoluto radica en que se registra siempre el error neto o absoluto. En este caso, es lo mismo cometer un error tanto si en lugar de 2 colocamos 3, como si en lugar de tres billones escribimos tres billones y uno; el error es en ambos casos 1. Sin embargo, es claro que las consecuencias no pueden ser las mismas.

2. Error relativo.

Si b es una aproximación de a, para tener una medida de error que lleve en cuenta el tamaño de los objetos comparados con el error en sí, se define (para $a \neq 0$) el **error relativo** de b como aproximación de a por

$$\mathrm{err}(a,\, b) \;=\; \frac{|b \,-\, a|}{|a|} \;\leq\; \frac{\frac{1}{2}10^{-t+1}}{10^{0}} \;=\; \frac{1}{2}10^{-t+1},$$

en el caso del redondeo y 10^{-t+1} en el del truncamiento. Sin embargo, ahora estamos tomando en cuenta el tamaño de a.

Ejemplo: Consideremos una aritmética con 4 dígitos de precisión. Escogemos un número como $7'354,287.173$ que podemos escribir como $a = 0.\underline{7354}287173 \times 10^7$ pero que aproximaremos por $b = 0.7354 \times 10^7$, luego

$$\begin{aligned}\mathrm{err}(a,\, b) \;&=\; \frac{|0.7354 \times 10^7 \;-\; 0.7354287173 \times 10^7|}{|0.7354287173 \times 10^7|} \;=\; \frac{|0.7354 \,-\, 0.7354287173| \times \cancel{10^7}}{|0.7354287173| \times \cancel{10^7}}\\ &=\; \frac{|-\, 0.0000287173|}{|0.7354287173|} \;=\; \frac{0.287173 \times 10^{-4}}{0.7354287173} \;\leq\; \frac{1/2}{10^{-1}}10^{-4} \;=\; \frac{1}{2}10^{-3}.\end{aligned}$$

En lugar de $\mathrm{Err}(a,\, b) = 0.287173 \times 10^3$.

2.6. Operaciones en la aritmética de punto flotante

Al introducir un número en la máquina se produce un error inherente a la precisión con la que estemos trabajando. Esta perturbación del número original se puede propagar a la hora de realizar cálculos dentro de la máquina. Es por ello que necesitamos entender cómo se operan distintas funcionalidades dentro de estos cómputos.

Antes de ir más lejos, sabemos que el número $x \in \mathbb{R}$ introducido en la máquina tendrá un valor $x' = \texttt{fl}(x)$. Dada la formulación (2.1) y la definición del épsilon de máquina, es fácil ver que

$$\text{err}(x,\, x') \;\leq\; \frac{\varepsilon_{\text{M}}}{2},$$

al realizar el redondeo. Este hecho muestra que x' se puede ver como una perturbación de x, en el sentido de que $x' = x(1+\epsilon)$ con $|\epsilon| \leq \varepsilon_{\text{M}}/2$, pues

$$\text{err}(x,\, x') \;=\; \frac{|x - x(1+\epsilon)|}{|x|} \;=\; \frac{|x|\,|\epsilon|}{|x|} \;=\; |\epsilon| \;\leq\; \frac{\varepsilon_{\text{M}}}{2} \;<\; \varepsilon_{\text{M}}.$$

Tomaremos esta última desigualdad absoluta pues simplifica cargar el denominador y además incluye el caso del truncamiento en lugar del redondeo.

Del mismo modo, esperamos que a cada operación en aritmética de punto flotante se produzca un error relativo de a lo sumo ε_{M}. Digamos, al realizar lo siguiente:

$$\texttt{fl}(a \circ b) \;=\; (a \circ b)\,(1+\epsilon), \qquad \text{con} \qquad |\epsilon| \;<\; \varepsilon_{\text{M}} \tag{2.2}$$

y donde $\circ$ es cualquiera de las operaciones "fuertes", división ($\div$) o producto ($\times$).

En la suma y la resta, algo un poco distinto sucede. Veamos que con una aritmética de 4 dígitos de precisión tenemos que

$$1 - 0.9999 \;=\; 0.00001 \;=\; 1 \times 10^{-5},$$

que es un número en esta aritmética donde parece que tiene un registro con 5 dígitos de precisión.

Observemos atentamente que sucede con una suma típica, pues la resta está incluida, tenemos

$$\texttt{fl}(a \pm b) \;=\; a(1 + \epsilon_a) \;\pm\; b(1 + \epsilon_b) \;=\; a \;\pm\; b \;+\; a\epsilon_a \;\pm\; b\epsilon_b,$$

con $|\epsilon_a|$, $|\epsilon_b| < \varepsilon_{\text{M}}$. Ahora, los dos últimos términos pueden dar la casualidad de cancelarse, siendo esto poco probable, vemos que pueden cambiar con el tamaño de a y b y la distancia de éstos a los números en la aritmética de punto flotante. Esto da un primer indicio del *backward error analysis* que veremos más adelante. Por ahora, si $|a| \not\approx |b|$ y tienen el mismo signo, entonces podemos pensar que $\texttt{fl}(a+b) = (a+b)(1+\epsilon)$ como antes en la ecuación (2.2). ¿Qué sucede con la resta?

Paréntesis: En PYTHON se puede retomar esta idea con la funcionalidad de *evocar* o *cast* un tipo específico de número. Existen ya predefinidos `int8`, `int16`, ..., `float16`, `float32`, ... Hagamos el siguiente ejemplo:

```
In [1]: import numpy as np
In [2]: x = np.e; y = np.float32(np.e); x - y
```

¿Qué es lo que ha sucedido? La precisión que observamos es para la SP al hacer el *cast* en `y`. Para algunos ejemplos con precisión fija de dígitos puedes usar otros `float`; investígalos en PYTHON. (Puedes buscar sobre *half-*, *single-*, *double-* o *quadruple-precision.*)

Con la ecuación (2.2) estamos implicando que a cada paso que se realiza en la máquina, un nuevo error puede ser introducido, por ello, es mejor realizar el menor número de cálculos posibles. Por ejemplo, podemos tomar dos versiones del mismo polinomio:

$$p(x) = (1-x)^{10} = x^{10} - 10x^9 + 45x^8 - 120x^7 + 210x^6 - 252x^5 + 210x^4 - 120x^3 + 45x^2 - 10x + 1,$$

con lo cual, podemos evaluar ambas expresiones y ver la sutil diferencia que se produce.

Ejemplo: En este caso usaremos para practicar el `lambda` de PYTHON. Escribimos las dos versiones del polinomio arriba:

```
1 p = lambda x: (1. - x)**10
2 q = lambda x: (x**10 - 10.*x**9 + 45.*x**8 - 120.*x**7 + 210.*x**6
3                - 252.*x**5 + 210.*x**4 - 120.*x**3 + 45.*x**2 - 10.*x + 1.)
```

donde `lambda x` es la "firma" de función y representa que la variable es `x`. El nombre de esta variable puede ser cualquier cosa y puede ser de hecho una colección de variables como `(x1, x2, ..., xn)`. También debemos notar que los números tienen un punto decimal; queremos ser serios al decirle a PYTHON que estos son números en DP.

Acabemos el ejemplo con las siguientes líneas:

```
4 import numpy as np
5 x = np.linspace(0.9, 1.1, 200)
6 import matplotlib.pyplot as plt
7 plt.plot(x, p(x), linewidth = 3)
8 plt.plot(x, q(x), '--r')
9 plt.show()
```

y aproximemos la visión al cero de multiplicidad 10 en $x = 1$, cambiando la línea 5 por

```
5 x = np.linspace(0.99, 1.01, 200)
```

Observa que hemos llamado las librerías de PYTHON para los cálculos numéricos con `numpy` y para los gráficos con `matplotlib.pyplot`.

Es claro que así como nos hemos preocupado con la suma y la resta, el producto y la división tendrán errores asociados con la aritmética de punto flotante. Consideramos $x \odot y$ como el producto $\texttt{fl}(x \cdot y)$. Si tanto x como y son números en la precisión empleada, entonces

$$x \odot y \;=\; (x \cdot y)(1+\epsilon_{\odot}), \qquad \text{con} \qquad |\epsilon_{\odot}| \;\leq\; \frac{\varepsilon_{\mathrm{M}}}{2} \;<\; \varepsilon_{\mathrm{M}}.$$

En el caso que debamos considerar también la incorporación de los números reales a la aritmética de punto flotante, tenemos,

$$\begin{aligned} x' \odot y' &= \texttt{fl}(x' \cdot y') \;=\; (x' \cdot y')(1+\epsilon_{\odot}) \\ &= \big(x(1+\epsilon_x) \cdot y(1+\epsilon_y)\big)(1+\epsilon_{\odot}) \\ &= (x \cdot y)\big((1+\epsilon_x)(1+\epsilon_y)(1+\epsilon_{\odot})\big) \;=:\; (x \cdot y)(1+\epsilon), \end{aligned}$$

donde ahora $|\epsilon| = |(1+\epsilon_x)(1+\epsilon_y)(1+\epsilon_{\odot}) - 1| \leq 3\varepsilon_{\mathrm{M}} + 3\varepsilon_{\mathrm{M}}^2 + \varepsilon_{\mathrm{M}}^3$, lo cual debe de ocurrir más o menos con cualquier operación y dado que $\varepsilon_{\mathrm{M}}^3 \ll \varepsilon_{\mathrm{M}}^2 \ll \varepsilon_{\mathrm{M}}$, podemos ver que es como $3\varepsilon_{\mathrm{M}}$. Tenemos que tener cuidado con la división y se encuentra este ejercicio en la lista de ejercicios.

2.6.1. Cancelamiento catastrófico

A veces aún en las operaciones más sencillas, la aritmética de punto flotante nos puede hacer una mala pasada. Para ilustrarlo veamos lo siguiente.

Ejemplo: Calcula con 5 dígitos de precisión la siguiente suma,

$$37,654 \;+\; 25.874 \;-\; 37,679 \;=\; 0.874.$$

Como vemos, todos los números se encuentran en la aritmética. Realicemos la operación de dos en dos como sabemos hacerlo:

$$\begin{array}{r} 37,654 \\ +\quad 25.874 \\ \hline 37,679.874 \end{array} \quad\Longrightarrow\quad 37,680 \quad\Longrightarrow\quad \begin{array}{r} 37,680 \\ -\;\; 37,679 \\ \hline 1 \end{array}$$

Es decir, en esta aritmética de 5 dígitos hemos producido un error tal que nuestro resultado no tiene ni un dígito correcto con respecto al resultado esperado. En este caso, hemos producido un *cancelamiento catastrófico*.

Si en lugar de seguir ese orden, hacemos $37,654 - 37,679 + 25.874$, el resultado será correcto. Prueba también cambiar el segundo término por 25.074.

La solución se ocurre un poco similar a lo que ocurrió al sumarle 10^n unos al número 10^m o

arreglar la suma sumando primero estos unos. Las oscilaciones en el ejemplo del polinomio, también son causadas por estas ideas. La moraleja es enfrentar los sumandos (positivos y negativos) de la misma magnitud, por pedazos y después agregar al total desde el menor en magnitud hasta el mayor en magnitud.

Para ser más rigurosos en este comentario, vale la pena hacer un análisis del error en una suma grande. Por ejemplo, aproximemos el error al evaluar la suma $x_1 + x_2 + \cdots + x_n$ cuando estos números ya se encuentran en la aritmética de punto flotante empleada.

Recordamos que $\texttt{fl}(x_1 + x_2) = (x_1 + x_2)(1 + \epsilon_1)$, donde en particular $|\epsilon_1| < \varepsilon_{\text{M}}$. Definamos $S_k = \texttt{fl}(x_1 + x_2 + \cdots + x_k)$, luego tenemos

$$
\begin{aligned}
S_2 &= \texttt{fl}(x_1 + x_2) = (x_1 + x_2)(1 + \epsilon_1) = x_1(1 + \epsilon_1) + x_2(1 + \epsilon_1) \\
S_3 &= \texttt{fl}(S_2 + x_3) = (S_2 + x_3)(1 + \epsilon_2) = S_2(1 + \epsilon_2) + x_3(1 + \epsilon_2) \\
&= x_1(1 + \epsilon_1)(1 + \epsilon_2) \\
&+ x_2(1 + \epsilon_1)(1 + \epsilon_2) \\
&+ x_3(1 + \epsilon_2) \\
&\vdots \\
S_n &= \texttt{fl}(S_{n-1} + x_n) = (S_{n-1} + x_n)(1 + \epsilon_{n-1}) = S_{n-1}(1 + \epsilon_{n-1}) + x_n(1 + \epsilon_{n-1}) \\
&= x_1(1 + \epsilon_1)(1 + \epsilon_2) \cdots (1 + \epsilon_{n-1}) \\
&+ x_2(1 + \epsilon_1)(1 + \epsilon_2) \cdots (1 + \epsilon_{n-1}) \\
&+ x_3(1 + \epsilon_2)(1 + \epsilon_3) \cdots (1 + \epsilon_{n-1}) \\
&\quad \vdots \\
&+ x_{n-1}(1 + \epsilon_{n-2})(1 + \epsilon_{n-1}) \\
&+ x_n(1 + \epsilon_{n-1}) =: x_1(1 + \eta_1) + x_2(1 + \eta_2) + \cdots + x_n(1 + \eta_n),
\end{aligned}
$$

donde $\eta_1 = \eta_2 = (1 + \epsilon_1)(1 + \epsilon_2) \cdots (1 + \epsilon_{n-1}) - 1$, $\eta_3 = (1 + \epsilon_2) \cdots (1 + \epsilon_{n-1}) - 1$ y en general $\eta_k = (1 + \epsilon_{k-1}) \cdots (1 + \epsilon_{n-1}) - 1$. Se espera que dado que $1 + \epsilon_k$ es próximo de uno, pues $1 + \eta_k$ también debe serlo.

Para ilustrar esta idea es bueno tomar uno de los últimos casos. Supongamos que $\varepsilon_{\text{M}} = 10^{-10}$, entonces, $|\epsilon_1|, |\epsilon_2|, \ldots, |\epsilon_n| \leq 10^{-10}$, así por ejemplo,

$$
1 + \eta_{n-1} = (1 + \epsilon_{n-2})(1 + \epsilon_{n-1}) = 1 + \epsilon_{n-2} + \epsilon_{n-1} + \epsilon_{n-2}\epsilon_{n-1},
$$

dado que $|\epsilon_{n-2}\epsilon_{n-1}| \leq 10^{-20}$ y $|\epsilon_{n-2} + \epsilon_{n-1}| \leq 2 \cdot 10^{-10}$ vemos que $|\eta_{n-1}| \lesssim 2\varepsilon_{\text{M}}$. (Dado que con la aritmética de punto flotante hemos tomado $|\epsilon_k| < \varepsilon_{\text{M}}$ en lugar de $|\epsilon_k| \leq \varepsilon_{\text{M}}/2$, esta desigualdad puede ser estricta.) Siguiendo estos razonamientos, el número de errores que definen cada η_k nos

dan la cota, es decir,

$$|\eta_1| \lesssim (n-1)\varepsilon_{\mathrm{M}},\ |\eta_2| \lesssim (n-1)\varepsilon_{\mathrm{M}},\ \ldots,\ |\eta_k| \lesssim (n-k+1)\varepsilon_{\mathrm{M}},\ \ldots,\ |\eta_{n-1}| \lesssim 2\varepsilon_{\mathrm{M}},\ |\eta_n| \lesssim \varepsilon_{\mathrm{M}}.$$

Para controlar esta cota de error, tenemos el próximo resultado.

Teorema 2.1. *Suponga que* $n\varepsilon_M \leq 0.1$ *y que* $|\epsilon_k| \leq \varepsilon_M$ *para* $k = 1, 2, \ldots, n$. *Entonces*

$$(1+\epsilon_1)(1+\epsilon_2)\cdots(1+\epsilon_n) \;=\; 1+\eta$$

satisface $|\eta| \leq 1.06n \cdot \varepsilon_M$.

Por un lado, digamos que $\varepsilon'_{\mathrm{M}} := 1.06\varepsilon_{\mathrm{M}}$ se conoce a veces como la **unidad de redondeo ajustada**, pues con ella podemos hacer las cuentas arriba sin el menor aproximado. Por el otro lado, vemos que el teorema nos dice que estas cuentas son bastante robustas, si por ejemplo $\varepsilon_{\mathrm{M}} = 10^{-16}$, necesitamos $n \geq 10^{15}$ sumandos para romper la hipótesis. Pensando en que cada operación se realiza en 1ns (*nanosegundo* que equivale a 10^{-9}s), pues entonces tomaremos más de 10^7s o más de 115 días para realizar la suma con un posible error del orden de los sumandos.

Lo que hemos realizado en esta última sección es un análisis clásico de *backward error*. Fue popularizado y desarrollado por James H. Wilkinson y se refiere al análisis que se hace en general de un aproximación al estudiar cotas para los parámetros que no alteren el resultado o que lo dejen dentro de un margen específico de error.

En su libro "Afternotes on Numerical Analysis", Ian Stewart, [14], hace un ejemplo sobre estas ideas con el siguiente diálogo ficticio:

> 8. To emphasize this point, suppose you are a numerical analyst and are approached by a certain Dr. Xyz who has been adding up some numbers.
>
> Xyz: I've been trying to compute the sum of ten numbers, and the answers I get are nonsense, at least from a scientific viewpoint. I wonder if the computer is fouling me up.
>
> You: Well it certainly has happened before. What precision were you using?
>
> Xyz: Double. I understand that it is about fifteen decimal digits.
>
> You: Quite right. Tell me, how accurately do you know the numbers you were summing?
>
> Xyz: Pretty well, considering that they are experimental data. About four digits.
>
> You: Then it's not the computer that is causing your poor results.
>
> Xyz: How can you say that without even looking at the numbers? Some sort of magic?

YOU: Not at all. But first let me ask another question.

XYZ: Shoot.

YOU: Suppose I took your numbers and twiddled them in the sixth place. Could you tell the difference?

XYZ: Of course not. I already told you that we only know them to four places.

YOU: Then what would you say if I told you that the errors made by the computer could be accounted for by twiddling your data in the fourteenth place and the performing the computations exactly?

XYZ: Well, I find it hard to believe. But supposing it's true, you're right. It's my data that's the problem, not the computer.

Lo cual nos deja un vago entendimiento de si en realidad los datos son los responsables de este problema. Hasta ahora nos hemos dedicado a estudiar este tipo de problemas, piensa qué es lo que debe estar sucediendo antes de continuar.

Finalmente Stewart termina apuntando que el diálogo es artificial en tres puntos:

1. Es un problema muy sencillo para caracterizar uno de la vida real.

2. Es tan simple, que es difícil creer que exista tal DR. XYZ así de ingenuo.

3. Finalmente, de existir este DR. XYZ, no dejaría la conversación tan fácilmente sin un argumento que le ayudara mejor.

9. At this point you might be tempted to bow out. Don't. Dr. XYZ wants to know more.

XYZ: But what went wrong? Why are my results meaningless?

YOU: Tell me, how big are your numbers?

XYZ: Oh, about a million.

YOU: And what is the size of your answer?

XYZ: About one.

YOU: And the answers you compute are at least an order of magnitud too large.

XYZ: How did you know that. Are you a mind reader?

YOU: Common sense, really. You have to cancel five digits to get your answer. Now if you knew your numbers to six or more places, you would get one or more accurate digits in your answer. Since you know only four digits, the lower two digits are garbage and won't cancel. You'll get a number in the tens greater instead of a number near one.

> XYZ: What you say makes sense. But does that mean I have to remeasure my numbers to six or more figures to get what I want?
> YOU: That's about it.
> XYZ: Well I suppose I should thank you. But under the circumstances, it's not easy.
> YOU: That's OK. It comes with the territory.

Sin embargo, los puntos a destacar son primero el *análisis de error en retroceso* como una herramienta útil para descartar a la máquina como sospechosa de que algo va mal. En segundo lugar, el análisis del *backward error* es suficiente *per se*. Queremos saber qué va mal, queremos saber quién es el responsable de las dificultades que estamos teniendo, en fin, necesitaremos saber si el problema es *mal condicionado* y en cuyo caso, qué hacer al respecto.

Hemos terminado con esta sección y la lista de ejercicios pueden ser una buena guía para entender detalles que se han quedado entre líneas. De ahora en adelante, iremos a diversas aplicaciones, pero no debemos perder de vista lo estudiado en este capítulo pues es un diccionario entre el lenguaje de la máquina y nosotros así como una serie de ejemplos de donde puede "confundirse" un cálculo.

Capítulo 3

Localización de raíces y extremos locales

En la primera clase construimos la función (1.1) que aquí repetimos por comodidad:

$$f(\theta) \;=\; \frac{2v_0^2 \operatorname{sen}\theta\cos\theta}{g} \;-\; d. \tag{3.1}$$

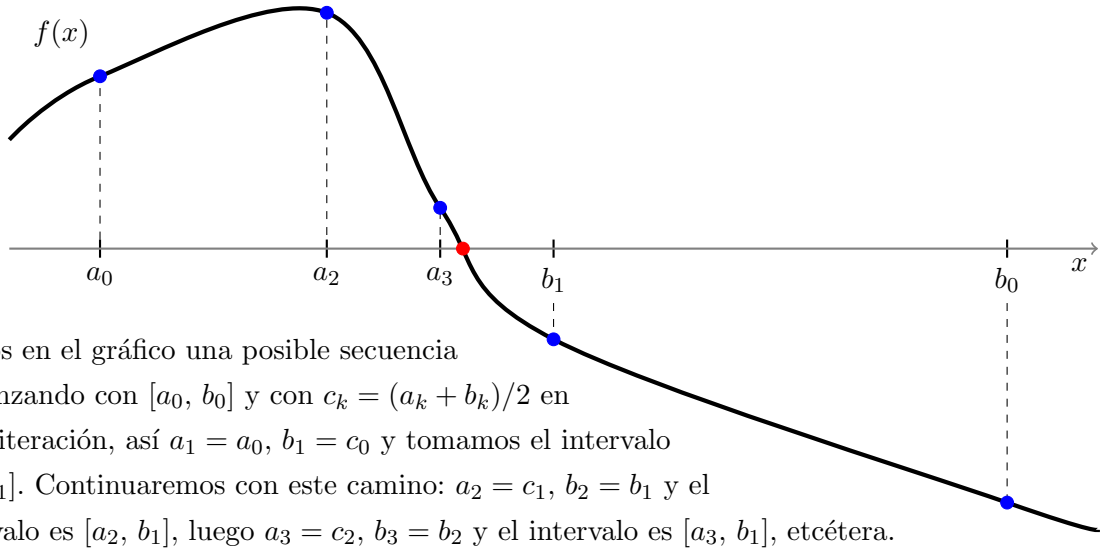
Buscábamos $\theta^* \in [0,\, \pi/2]$ tal que fuera una raíz nula, es decir, tal que $f(\theta^*) = 0$ se satisficiera.

Una de las primeras formas que se nos podría ocurrir y que de hecho hemos comentado en clase, es justo como probablemente alguien podría hacer con un cañón real sin tener a la mano la fórmula o si quiera los parámetros dados de forma explícita. Esta forma se basa en ser *a prueba y error*. Para usar este método, deben de ser consideradas algunas suposiciones, como por ejemplo poder encontrar dos ángulos distintos θ_1 y θ_2 tales que uno se sobre pase en el tiro y que el otro se quede corto. De este modo, podemos encontrar un θ_3 en el medio que nos ayude a dar una mejor aproximación y de este modo tener dos nuevos elementos. Sin embargo, esto implica que existe algún tipo de continuidad, todos los tiros con valores θ_3 entre los dos anteriores deben de reflejarse en un continuo de balas de cañón cayendo al menos entre los dos tiros originales.

En términos más formales, cuando $f(\theta_1) > 0$ y $f(\theta_2) < 0$ para una función continua, entonces podemos encontrar cualquier valor en el medio, este famoso resultado se resume del modo siguiente.

Teorema 3.1 (Valor intermedio)**.** *Si f es continua para $x \in [a,\, b]$, y g es un valor entre $f(a)$ y $f(b)$, entonces existe un valor x en el intervalo tal que $g = f(x)$.*

Es decir, nosotros podemos usar el TVI (Teorema 3.1) para buscar los ceros o las raíces nulas de $f(x) = 0$, claro, desde que tengamos dos valores a y b que satisfagan que $\operatorname{sgn}(f(a)) \cdot \operatorname{sgn}(f(b)) \leq 0$ y que f sea continua entre a y b. Si buscamos un cero, entonces basta que se satisfagan las desigualdades. Luego, si tomamos $c = (a+b)/2$ como el punto medio, podemos colocar un nuevo intervalo menor y comenzar nuevamente; cuando $f(c) \neq 0$, claro.



Vemos en el gráfico una posible secuencia comenzando con $[a_0, b_0]$ y con $c_k = (a_k + b_k)/2$ en cada iteración, así $a_1 = a_0$, $b_1 = c_0$ y tomamos el intervalo $[a_0, b_1]$. Continuaremos con este camino: $a_2 = c_1$, $b_2 = b_1$ y el intervalo es $[a_2, b_1]$, luego $a_3 = c_2$, $b_3 = b_2$ y el intervalo es $[a_3, b_1]$, etcétera.

Discusión: En el ejemplo anterior, los extremos del intervalo siempre serán de la forma $[a_i, b_j]$ y el intervalo a cada paso menor. ¿Cuándo debemos parar? En el límite, cuando el número de pasos tiende a infinito sí habremos llegado a la raíz c que satisface $f(c) = 0$.

En la aritmética de punto flotante, como hemos visto, las cosas son distintas, tal vez el valor $c \neq \texttt{fl}(c)$ y por lo tanto hay que tener cuidado cuándo parar. La solución $c' = c(1 + \epsilon)$ sabemos que es próxima y que este valor depende del ε_{M}. En resumen, buscamos una solución próxima al dictaminar una *tolerancia*, `tol`. Mira el siguiente *pseudocódigo*.

Código: Algoritmo de la bisección (PYTHON)

```
Entradas: f, a, b, tol  -  funcion, intervalo y toleracia
Salidas:            c   -  raiz con f(c) proximo de cero
---------- ---------- ---------- -//- ---------- ---------- ----------
while ( abs(b - a) > tol ):
    c  = (a + b)/2.0
    fc = f(c)
    # Aqui puede ir una PROTECCION
    if ( fc == 0.0 ):
        a = c; b = c; fa = fc; fb = fc
    elif ( fc*fb < 0.0 ):
        a = c; fa = fc
    else:
        b = c; fb = fc
return c, fc
```

El pseudocódigo en general tiene tres partes importantes: el título del algoritmo que resuelve, las entradas y salidas del código y la sintaxis del código. Esta última tiene una notación universal que no usamos sino aparentamos un poco la notación con la que se escriben los códigos en PYTHON.

Hay dos pequeños comentarios en verde con comandos faltantes, los cuales pueden estar dentro o fuera del código principal. El segundo de estos se puede completar con algunas pocas líneas:

```
if ( c == a or c == b ):
    a = c; b = c; fa = fc; fb = fc
    return c, fc
```

que bien puede meterse dentro de la protección del ciclo `while`.

Ejercicio 5: Explica cuál es la importancia de esta última protección y complementa el código de la `INICIALIZACION`. Escribe el algoritmo de la bisección en PYTHON.

Ejercicio 6: Encuentra una de las raíces nulas de $f(c) = 0$ en (3.1) con una $\texttt{tol} = 10^{-6}$ si $v_0 = 50$, $g = 9.807$ y $d = 120$.

Como uno puede notar en este último ejercicio, la parte más complicada del algoritmo es encontrar un intervalo inicial con dos valores que den positivo y negativo. De hecho, a veces podemos necesitar un criterio extra para saber si podemos inicializar el código, por ejemplo, tomando $v_0 = 30$ y $d = 150$, tenemos un caso sin solución, pero no tenemos cómo saber de ello hasta que no hagamos un gráfico de $f(\theta)$ y veamos que toda la función satisface $f(\theta) < 0$ para todo $\theta \in \mathbb{R}$.

Por otro lado, siempre que el algoritmo comienza con valores válidos podemos garantizar que éste converge a una solución de nuestra función. De hecho, notemos que si $L_0 = |b_0 - a_0|$ y en general $L_k = |b_k - a_k|$ para a_k, b_k la k-ésima actualización de los extremos del intervalo, entonces

$$L_1 \;=\; |b_1 - a_1| \;=\; \begin{cases} |c - a_0| = |b_1 - a_0|, & \text{si } \; f(c) * f(a) \;<\; 0 \\ |b_0 - c| = |b_0 - a_1|, & \text{si } \; f(c) * f(b) \;<\; 0 \end{cases} \;=\; \frac{|b_0 - a_0|}{2} \;=\; \frac{L_0}{2}.$$

Con esto vemos que en general, $L_k = |b_k - a_k| = L_0\, 2^{-k}$, para los extremos de la k-ésima iteración.

En nuestro código paramos cuando $L_k \leq \texttt{tol}$, por lo tanto tenemos

$$\frac{L_0}{2^k} \;\leq\; \texttt{tol} \qquad \Longleftrightarrow \qquad \frac{L_0}{\texttt{tol}} \;\leq\; 2^k \qquad \Longleftrightarrow \qquad \log_2 \frac{L_0}{\texttt{tol}} \;\leq\; k.$$

Dado que k debe ser un entero, tomamos K como el primero que satisface la desigualdad, es decir,

$$K \;=\; \left\lceil \log_2 \frac{L_0}{\texttt{tol}} \right\rceil,$$

que en el caso de tener $L_0 = 1$ y $\texttt{tol} = 10^{-6}$, nos dice que son necesarios $K = 20$ pasos para garantizar esta tolerancia. Tenemos $K = 52$ con el épsilon de máquina de DP.

La última curiosidad que nos queda por ahora en este ejemplo con el método de la bisección es que en la condición principal estamos tomando un error absoluto, hemos dicho también que esto no es lo más correcto. No es realmente fácil cambiar esta condición, pues no estamos tan al tanto de quién es el mejor candidato, si a_k o b_k. Para garantizar que ambos elementos entren en la tolerancia podría pensarse en una condición como

```
while ( abs(b - a)/min(abs(a), abs(b)) > tol ):
```

Tampoco es correcta, se deja como incentivo para una mejor condición que escape del ciclo.

3.1. El método de Newton

Aparentemente el método de la bisección es rápido, 20 o 52 pasos en una computadora moderna parecen ser nada. Sin embargo, dado que para una tolerancia igual a 10^{-m} tenemos que el número de pasos es $K = \lceil m \log_2 10 \rceil = \lceil 3.32 \cdot m \rceil$, observamos que cada vez que duplicamos el número de cifras de precisión, duplicaremos el número de pasos necesarios. Esto está bien, pero significa que el método tiene una *convergencia lineal*.

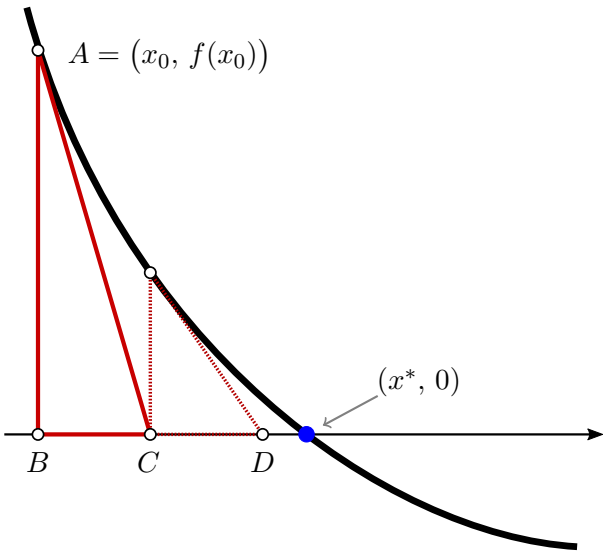
Un método más eficiente en este sentido es el denominado Newton-Raphson y que por cuestiones de comodidad lo llamaremos solamente por el primero de estos autores. Con este método se llega mucho más rápido a la tolerancia escogida a la hora de llegar a $f(x) \approx 0$ y, aunque tendrá algunas desventajas, uno de los beneficios es que basta un punto x_0 de entrada que suponga cierta cercanía con la raíz procurada.

Existen al menos dos maneras distintas de derivar este método. Mostraremos dos que tienen cada una una belleza intrínseca en lo que además pueden decirnos sobre el método.

1. Construcción Geométrica.

Digamos que la raíz que buscamos es x^* y que tenemos una estimativa $x_0 \approx x^*$.

Siguiendo el dibujo al lado tomamos $B = (x_0, 0)$ y $A = (x_0, f(x_0))$ como su primer levantamiento. A partir del punto A trazamos la tangente a la curva y buscamos su intersección en el eje x en el punto $C = (x_1, 0)$. Desde C podemos repetir el procedimiento para encontrar D y continuar sucesivamente hasta tener una buena aproximación de $(x^*, 0)$.



Notamos que de las identidades trigonométricas tenemos que la tangente es:

$$\tan \sphericalangle ACB \;=\; \frac{\overline{AB}}{\overline{BC}} \qquad \text{o también} \qquad f'(x_0) \;=\; \frac{0 - f(x_0)}{x_1 - x_0},$$

al tomar la secante de A a C. Así despejando nos lleva a

$$x_1 \;=\; x_0 \;-\; \frac{f(x_0)}{f'(x_0)}.$$

Puesto que el método es recursivo, tenemos que para todo x_0 dado, la sucesión

$$x_{k+1} \;=\; x_k \;-\; \frac{f(x_k)}{f'(x_k)}, \qquad \text{para} \qquad k \in \mathbb{N}, \tag{3.2}$$

debe de aproximarse cada vez más de la solución x^*.

2. Construcción Analítica (vía Serie de Taylor).
Sin perder de vista la idea geométrica del gráfico, vemos que para x_0 dado, tenemos $f(x_0)$, $f'(x_0)$, ... Así, para $x \approx x_0$ cualquiera, la serie de Taylor es

$$f(x) \;=\; f(x_0) \;+\; (x - x_0)\, f'(x_0) \;+\; \frac{1}{2}\,(x - x_0)^2\, f''(\xi_0), \qquad \text{con} \qquad \xi_0 \;\in\; [x,\, x_0].$$

Si $x_0 \approx x^*$ y $f''(x_0)$ es pequeño, vemos que una aproximación de $f(x)$ puede ser dada por

$$\hat{f}(x) \;=\; f(x_0) \;+\; (x - x_0)\, f'(x_0),$$

que es de orden cuadrático en $x - x_0$, en notación matemática usamos $\mathcal{O}\big((x - x_0)^2\big)$. Al despejar, vemos que obtenemos nuevamente (3.2) si x_1 es tal que $\hat{f}(x_1) = 0$.

Tenemos entonces dos modos de ver el mismo desarrollo, el primero nos da una idea geométrica y por lo tanto entendemos cómo opera el método, por ejemplo, si $f'(x_0) \approx 0$ y $f(x_0)$ es grande, entonces difícilmente el método convergerá. Hagamos más preciso el concepto de convergencia cuando una secuencia $(x_n)_{n\in A}$ converge a x^*, es decir, $x_n \to x^*$ con los elementos de A.

Definición 3.2. *Si existe una constante $|\varrho| \in (0,\, 1)$ tal que*

$$\lim_{n\to\infty} \frac{x_{n+1} - x^*}{x_n - x^*} \;=\; \varrho,$$

*decimos que la convergencia $x_n \to x^*$ es una **convergencia lineal**. Si existe un valor $p > 1$ y una constante positiva C tales que*

$$\lim_{n\to\infty} \frac{|x_{n+1} - x^*|}{|x_n - x^*|^p} \;=\; C,$$

*entonces decimos que es una **convergencia superlineal**, en particular decimos que es de orden p.*

Típicamente, esperamos que una tabla el error $e_n := |x_n - x^*|$ (ya sea el absoluto o el relativo) tenga un comportamiento que reduzca un orden a cada paso en el caso lineal. En el caso cuadrático, el orden de precisión debe doblar a cada paso y en el caso cúbico triplicar. Tenemos la siguiente tabla como ejemplo.

n	e_n-lineal	e_n-cuadrático	e_n-cúbico
0	1	1	1
1	0.1	0.43	0.701
2	0.015	0.0798	0.6897
3	0.0021	0.00345	0.6601
4	0.000341	0.00000432	0.5702
5	0.0000411	4.021×10^{-12}	0.45332
6	0.00000576	2.337×10^{-24}	0.09931
7	0.000000612	1.139×10^{-48}	$0.00314 = 3.14 \times 10^{-3}$
8	⋮	5.601×10^{-97}	4.01×10^{-9}
9		3.012×10^{-194}	5.62×10^{-25}
10		⋮	7.03×10^{-75}
11			8.11×10^{-223}
12			⋮

En la tabla anterior observamos varios fenómenos. Por ejemplo, en la convergencia lineal para cada n, el error es $\mathcal{O}(10^{-n})$, en el caso cuadrático es $\mathcal{O}(10^{-2(n+1)})$ y solamente a partir de $n = 5$. En el caso cúbico tenemos un comportamiento más extravagante, pues es $\mathcal{O}(10^{-3(n-6)})$ a partir de $n = 7$. De algún modo, el comportamiento "cúbico" sólo se desprende a partir de $n \geq 8$. Este es un fenómeno que puede ocurrir comúnmente cuando comenzamos lejos del valor de convergencia de un método, se necesita sobre pasar lo que se conoce como un *cuello de botella*.

3.1.1. Convergencia del método de Newton

Hemos dicho que la construcción geométrica es interesante por el comportamiento e intuición que nos muestra. Por el otro lado, es de la derivación analítica de donde podremos obtener cotas y estimativas para entender la convergencia del método de Newton.

De la ecuación (3.2) obtenemos la fórmula iterativa del método de Newton y definiendo una función auxiliar como

$$\varphi(x) \;=\; x - \frac{f(x)}{f'(x)},$$

vemos que simplificamos la notación de esta iteración simplemente con $x_{n+1} = \varphi(x_n)$ con $n \in \mathbb{N}$.[1]

[1] Los sistemas dinámicos discretos estudian este tipo de iteraciones, de donde sabemos que las iteraciones convergen

Tomaremos para cualquier k, el error de x_k como aproximación de x^* dado por $\eta_k = x_k - x^*$. Notamos entonces que

$$\begin{aligned}
\eta_{k+1} &= x_{k+1} - x^* = \varphi(x_k) - \varphi(x^*) = \varphi(x^* + \eta_k) - \varphi(x^*) \\
&= \left[\cancel{\varphi(x^*)} + (x_k - x^*)\,\cancel{\varphi'(x^*)}^{\,0} + \frac{1}{2}(x_k - x^*)^2\,\varphi''(x^*) + \mathcal{O}(\eta_k^3)\right] - \cancel{\varphi(x^*)} \\
&= \frac{1}{2}\,\eta_k^2\,\varphi''(x^*) + \mathcal{O}(\eta_k^3),
\end{aligned}$$

donde hemos usado que

$$\varphi'(x) = 1 - \frac{f'(x)f'(x) - f''(x)f(x)}{\big(f'(x)\big)^2} = \frac{f''(x)f(x)}{\big(f'(x)\big)^2}, \qquad \text{luego} \qquad \varphi'(x^*) = 0,$$

puesto que $f(x^*) = 0$. (**Nota**: necesitamos que $f'(x^*) \neq 0$ se satisfaga.)

Es decir, tenemos que $\eta_{k+1} \approx \frac{1}{2}\eta_k^2\varphi''(x^*)$ pues supondremos que comenzamos cuando η_k ya es muy pequeño; además, hemos tomado tácitamente que $\varphi''(x^*) \neq 0$. Esta última estimativa muestra que el método de Newton tiene convergencia cuadrática.

Hay pocos problemas reales con el método de Newton. Este resulta ser una muy buena opción en general. Como todo método, si estamos lejos del radio de convergencia, pues demorará en llegar a la convergencia cuadrática como en los ejemplos que hemos dado anteriormente. La manera de resolver esto es dando una buena condición inicial; en el caso de tener un intervalo como en el método de la bisección, una aproximación lineal puede ser suficiente. Otro motivo de dificultad es el término $f'(x_k)$ que debe calcularse a cada paso de tiempo. Este término puede ser complicado e introduce errores siempre que es empleado, o peor aún, no se tiene una expresión para él. Veremos por ello algunos otros métodos.

3.2. Métodos alternativos de Newton

Hemos dejado entreabierto que el cálculo de $f'(x_k)$ en cada iteración del método de Newton puede llevar a distintas complicaciones. Por un lado, si la fórmula de la derivada es muy complicada, podemos introducir errores no deseados de redondeo o cancelamiento catastrófico o, peor aún, podemos fallar a la hora de escribirla y tener una fórmula falsa. Revisar la introducción del código, aún para probar fórmulas, debe ser una rutina constante en el cómputo científico.

Por otro lado, la derivada puede tener expresiones poco claras, la función puede provenir de una serie de raciocinios y no necesariamente tener una derivada explícita para todo punto. En estos

a un *punto fijo* o *solución de equilibrio* $x^* = \varphi(x^*)$ cuando $|\varphi'(x^*)| < 1$. En análisis matemático, la última condición dice que φ puede ser visto como una contracción y por lo tanto hay un punto fijo, que resulta ser la solución de equilibrio que buscamos.

casos podemos aproximar $f'(x_k)$ de alguna forma. En general escribiríamos la iteración como

$$x_{k+1} \;=\; x_k - \frac{f(x_k)}{g_k}, \qquad \text{para} \qquad k \in \mathbb{N}, \tag{3.3}$$

donde la manera de elegir g_k definirá el *método casi-Newton* que usamos, por ejemplo,

1. **Método de la pendiente constante:** Basta tomar todo g_k como la primera derivada en el método de Newton, es decir, $g_k = f'(x_0)$ para todo $k \in \mathbb{N}$.

2. **Método de la secante:** Podemos aproximar la derivada para x_k a partir del valor anterior y del propio a cada paso, es decir, tomamos

$$g_k \;=\; \frac{f(x_k) \;-\; f(x_{k-1})}{x_k \;-\; x_{k-1}},$$

   con el sutil detalle de que g_0 debe ser dado de algún modo en particular.

Esperamos, es claro, que ambos métodos converjan a la raíz x^* cuando x_0 es un valor próximo. Sin embargo, el análisis de convergencia debe verse modificado.

En general es bueno saber que un análisis de convergencia como este puede ser resumido en tres pasos casi invariantes.

Análisis de Convergencia:

1. Obtener una expresión para el $(k+1)$-ésimo error en términos de los errores anteriores, digamos η_k, η_{k-1}, ...

2. Usar la expresión para mostrar que el error converge a cero; en el límite $\eta_k \to 0$ cuando $k \to \infty$.

3. Mostrar la velocidad de esta convergencia.

El primer paso puede ser un poco más complicado de lo que parece, en el método de la secante veremos que se necesitan dos errores anteriores. A veces, las expresiones no son tan sencillas en una primera instancia, por ejemplo, con la pendiente constante, pero haciendo un dibujo podremos convencernos que el método tiene una convergencia lineal.

Ejercicio 7: Muestra que el método de la pendiente constante es un método a lo más lineal. Para ello, considera una función que sea lineal próxima de x^* pero no linear para un valor distante. Comienza en la región nolineal y observa el comportamiento en la región lineal. Ahora modifica la función para que sea completamente nolineal al rededor de x^*, ¿puedes garantizar la convergencia lineal? Trata de escribir los pasos dados arriba.

3.2.1. Convergencia del método de la secante

Veremos como antes que podemos reescribir la ecuación (3.3) como un método iterativo de la forma $x_{k+1} = \varphi(x_k, x_{k-1})$ donde ahora hay que ser explícitos que dependemos de dos pasos pues la secante g_k se obtiene de estos valores. Esta dependencia es la que lo define como un **método de dos pasos** y la prueba que haremos ahora es válida para cualquier método de dos pasos.

Definamos en este caso

$$\varphi(u, v) = u - \frac{f(u)\,(u - v)}{f(u) - f(v)} = \frac{v\,f(u) - u\,f(v)}{f(u) - f(v)},$$

como la función iterativa.

Primero notemos que para una raíz x^* se tiene que se satisface $\varphi(x^*, v) = x^* = \varphi(u, x^*)$ sin importar los valores que tengan u, $v \in \mathbb{R}$. De hecho, pues

$$\varphi(x^*, v) = \frac{v\,f(x^*) - x^*\,f(v)}{f(x^*) - f(v)} = \frac{-x^*\,f(v)}{-f(v)} = x^*;$$

de modo similar para la otra igualdad; es por ello que tomaremos $\varphi(x^*, x^*) = x^*$ aunque no está definida en ese punto, pero completa la continuidad. Así, además tenemos,

$$\frac{\partial}{\partial v}\varphi(x^*, v) = \frac{\partial^2}{\partial v^2}\varphi(x^*, v) \equiv 0 \qquad \text{y} \qquad \frac{\partial}{\partial u}\varphi(u, x^*) = \frac{\partial^2}{\partial u^2}\varphi(u, x^*) \equiv 0,$$

y de modo semejante para las parciales cruzadas. Todo esto es verdad pues estamos derivando la constante x^*.

Ahora podemos usar la serie de Taylor en dos variables, con lo que tenemos:

$$\begin{aligned}
\varphi(x^* + p, x^* + q) &= \varphi(x^*, x^* + q) + \frac{\partial\varphi}{\partial u}(x^*, x^* + q)\cdot p + \frac{1}{2}\frac{\partial^2\varphi}{\partial u^2}(x^*, x^* + q)\cdot p^2 + \mathcal{O}(p^3) \\
&= \varphi(x^*, x^*) + \frac{\partial\varphi}{\partial u}(x^*, x^*)\cdot p + \frac{1}{2}\frac{\partial^2\varphi}{\partial u^2}(x^*, x^*)\cdot p^2 + \mathcal{O}(p^3) \\
&+ \frac{\partial\varphi}{\partial v}(x^*, x^*)\cdot q + \frac{\partial^2\varphi}{\partial u\partial v}(x^*, x^*)\cdot pq + \mathcal{O}(p^2q) \\
&+ \frac{1}{2}\frac{\partial^2\varphi}{\partial v^2}(x^*, x^*)\cdot q^2 + \mathcal{O}(pq^2) \\
&+ \mathcal{O}(q^3) \\
&= x^* + \frac{1}{2}\left[\frac{\partial^2\varphi}{\partial u^2}(x_p^*, x_q^*)\cdot p^2 + 2\,\frac{\partial^2\varphi}{\partial u\partial v}(x_p^*, x_q^*)\cdot pq + \frac{\partial^2\varphi}{\partial v^2}(x_p^*, x_q^*)\cdot q^2\right],
\end{aligned}$$

donde $x_p^* = x^* + \theta\cdot p$ y $x_q^* = x^* + \theta\cdot q$ para $\theta \in [0, 1]$ controlan el error $\mathcal{O}(p^3, p^2q, pq^2, q^3)$ y nos hemos aprovechado nuevamente que las primeras y segundas derivadas se anulan al evaluarlas en

x^*. Usando nuevamente Taylor, tenemos

$$\begin{aligned}
\frac{\partial^2\varphi}{\partial u^2}(x_p^*,\, x_q^*) &= \cancelto{0}{\frac{\partial^2\varphi}{\partial u^2}(x_p^*,\, x^*)} + \theta\cdot\frac{\partial^3\varphi}{\partial u^2\partial v}(x_p^*,\, x^*+\tau_q\theta q)\cdot q,\\
\frac{\partial^2\varphi}{\partial v^2}(x_p^*,\, x_q^*) &= \cancelto{0}{\frac{\partial^2\varphi}{\partial v^2}(x^*,\, x_q^*)} + \theta\cdot\frac{\partial^3\varphi}{\partial u\partial v^2}(x^*+\tau_p\theta p,\, x_q^*)\cdot p,
\end{aligned}$$

para $\tau_p,\, \tau_q \in [0,\, 1]$. Juntando todo esto, tenemos que

$$\begin{aligned}
\varphi(x^*+p,\, x^*+q) &= x^* + \frac{pq}{2}\Bigg[2\,\frac{\partial^2\varphi}{\partial u\partial v}(x_p^*,\, x_q^*)\\
&\quad + \theta\frac{\partial^3\varphi}{\partial u^2\partial v}(x_p^*,\, x^*+\tau_q\theta q)p + \theta\frac{\partial^3\varphi}{\partial u\partial v^2}(x^*+\tau_p\theta p,\, x_q^*)q\Bigg].
\end{aligned}$$

Considerando p y q como perturbaciones de x^*, vemos que el término entre corchetes es referente a la perturbación de una iteración, es por ello que definimos desde el corchete

$$\begin{aligned}
r(e_1,\, e_0) &= 2\,\frac{\partial^2}{\partial u\partial v}\varphi(x^*+\theta e_1,\, x^*+\theta e_0)\\
&\quad + \theta\,\frac{\partial^3}{\partial u^2\partial v}\varphi(x^*+\theta e_1,\, x^*+\tau_{e_0}\theta e_0)\,e_1 + \theta\,\frac{\partial^3}{\partial u\partial v^2}\varphi(x^*+\tau_{e_1}\theta e_1,\, x^*+\theta e_0)\,e_0,
\end{aligned}$$

donde hemos tomado los errores iniciales como $e_0 = x_0 - x^*$ y $e_1 = x_1 - x^*$ y de este modo tenemos los siguientes pasos.

El primer paso: Para la convergencia notamos que $r(0,\, 0) = 2\frac{\partial^2}{\partial u\partial v}\varphi(x^*,\, x^*)$ y en cada paso tenemos

$$e_{k+1} = x_{k+1} - x^* = \frac{e_k\, e_{k-1}}{2}\, r(e_k,\, e_{k-1}), \tag{3.4}$$

por lo tanto para cada $C \in (0,\, 1)$ existe $\delta > 0$ tal que $|u|,\, |v| < \delta$ implica que

$$|v\cdot r(u,\, v)| \le C < 1.$$

Así, con $|e_0|$ y $|e_1| < \delta$, tenemos

$$|e_2| = \left|\frac{e_1}{2}\right| |e_0\cdot r(e_1,\, e_0)| \le \frac{C}{2}|e_1| < \frac{\delta}{2}, \tag{3.5}$$

que siguiendo iterativamente nos lleva al siguiente paso.

El segundo paso: Donde vemos de modo iterativo

$$|e_{k+1}| \;<\; \frac{C^k}{2^k}\,|e_1| \longrightarrow 0, \qquad \text{pues} \qquad \frac{C}{2} \;<\; 1.$$

La última condición expresada en (3.5) es más que suficiente para saber que los errores convergen a cero. En este caso tenemos una *contracción* pues $|e_0|$, $|e_1| < \delta$ implica que $|e_2| < \delta/2$ y de modo semejante $|e_3| < \delta/2$. Continuando tenemos que $|e_{2k}|$, $|e_{2k+1}| < \delta/2^k$ que también converge a cero.

El radio de convergencia no ha sido establecido. Para ello debemos notar que con un método de dos pasos se tiene en general

$$e_{k+1} \;=\; \frac{e_k\, e_{k-1}}{2}\, r(e_k,\, e_{k-1})$$

y que $r(0,\, 0)$ es una constante como se mencionó en el primer paso. Luego, dado que los errores convergen a cero cuando k tiende a infinito, tenemos que e_{k+1} es casi el cuadrado de e_k, de hecho,

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k\, e_{k-1}} \;=\; \frac{\partial^2}{\partial u \partial v}\varphi(x^*,\, x^*) \tag{3.6}$$

se satisface, mostrando una convergencia casi cuadrática. En resumen, si $r(0,\, 0) \neq 0$ decimos que hay un **convergencia a dos puntos**.

Observación: En el sentido de la convergencia basta ver que el error recursivo es de la forma

$$e_2 \;=\; e_1\, e_0 \left[\frac{\partial^2}{\partial u \partial v}\varphi(x^* + \theta e_1,\, x^* + \theta e_0) \;+\; \mathcal{O}(e_1,\, e_0)\right].$$

El término extra ayudará con la velocidad de esta convergencia.

Para la *razón de convergencia* tomamos la recurrencia en (3.4) y recordamos que $r(0,\, 0)$ es una constante dada. Entonces, tenemos que el límite (3.6) se satisface y por lo tanto mostraremos que esto equivale a la razón

$$S_k \;=\; \frac{|e_{k+1}|}{|e_k|^p} \qquad \text{ser convergente para} \qquad p \;=\; \frac{1+\sqrt{5}}{2},$$

donde $p \approx 1.618\cdots$ ¡es el número áureo![2]

Veamos que la secuencia para las razones S_k tiene realmente un límite distinto de cero con p

[2]El *número de oro* es la mayor raíz del polinomio $p^2 - p = 1$. Además de este número existe una gama muy grande de números metálicos que aparecen de manera espectacular en distintas áreas de las matemáticas.

dado como arriba. Tenemos así,

$$|e_k| \;=\; S_{k-1}|e_{k-1}|^p \qquad \text{y} \qquad |e_{k+1}| \;=\; S_k|e_k|^p \;=\; S_k S_{k-1}^p |e_{k-1}|^{p^2}.$$

Ahora, de (3.4), definimos

$$r_k \;=\; \left|\frac{r(e_k,\, e_{k-1})}{2}\right| \;=\; \left|\frac{e_{k+1}}{e_k\, e_{k-1}}\right| \;=\; \frac{S_k S_{k-1}^p |e_{k-1}|^{p^2}}{S_{k-1}|e_{k-1}|^p\,|e_{k-1}|} \;=\; S_k S_{k-1}^{p-1} |e_{k-1}|^{p^2-p-1},$$

que sabemos que converge por (3.6). Ahora, dado que tomaremos p como raíz de $p^2 - p - 1 = 0$, tenemos que $r_k = S_k S_{k-1}^{p-1}$.

Tomamos los logaritmos $\rho_k = \log r_k$ y $\sigma_k = \log S_k$, de este modo, $\rho_k = \sigma_k + (p-1)\sigma_{k-1}$ y queremos mostrar que $\sigma_k = \rho_k - (p-1)\sigma_{k-1}$ converge a un σ^*. Sabemos que convergencia en ρ es

$$\lim_{k\to\infty} \rho_k \;=\; \lim_{k\to\infty} \log r_k \;=\; \log\left|\frac{\partial^2}{\partial u \partial v}\varphi(x^*,\, x^*)\right| \;=\; \rho^*.$$

Entonces, mostrar que el límite σ^* existe, es equivalente a ver que en el límite se satisface la igualdad $\sigma^* = \rho^* - (p-1)\sigma^*$ o que la secuencia definida por

$$(\sigma_k \;-\; \sigma^*) \;=\; (\rho_k \;-\; \rho^*) \;-\; (p \;-\; 1)(\sigma_{k-1} \;-\; \sigma^*)$$

converge a cero.

Teorema 3.3. *Si las raíces de la ecuación*

$$x^n \;-\; a_1\, x^{n-1} \;-\; a_2\, x^{n-2} \;-\; \cdots \;-\; a_{n-1}\, x \;-\; a_n \;=\; 0$$

caen en el círculo unitario y $\lim_{k\to\infty} \eta_k = 0$*, entonces la secuencia* (ε_k) *generada por recursión como*

$$\varepsilon_k \;=\; \eta_k \;+\; a_1\, \varepsilon_{k-1} \;+\; a_2\, \varepsilon_{k-2} \;+\; \cdots \;+\; a_n\, \varepsilon_{k-n}$$

converge a cero para cualesquier valores $\varepsilon_0,\, \varepsilon_1,\, \ldots,\, \varepsilon_{n-1}$.

En nuestro caso: $n = 1$, $\varepsilon_k = \sigma_k - \sigma^*$ y $\eta_k = \rho_k - \rho^*$, así la ecuación es $x + (p-1) = 0$ con raíces en $[-1,\, 1]$. Esta afirmación se da, pues

$$\varepsilon_k \;=\; \eta_k \;-\; (p-1)\,\varepsilon_{k-1}.$$

En general para *convergencia multipasos* se tiene que estos valores, aún siendo supralineales, no llegan a ser cuadráticos. Se tiene $p(2) = 1.61\cdots$, $p(3) = 1.84\cdots$, $p(4) = 1.93\cdots$, $p(5) = 1.96\cdots$.

3.2.2. Aplicaciones poco usadas del método de Newton

Sabemos que si $a > 0$ entonces la única raíz nula de $f(x) = 1/x - a$ es $x^* = a^{-1}$. En general estos cálculos pueden ser engañosos y no siempre es sencillo el hacerlo de modo numérico.

Veamos qué sucede al aplicar el método de Newton en este caso. Para un x_0 dado y próximo de a^{-1}, tenemos

$$x_{k+1} \;=\; x_k \;-\; \frac{\frac{1}{x_k} - a}{-\frac{1}{x_k^2}} \;=\; x_k + \left[x_k - a x_k^2\right] \;=\; 2x_k - a x_k^2, \qquad \text{para todo} \qquad k \geq 0,$$

una manera que sólo involucra productos y sumas.

Ejercicio 8: Determina con la aritmética de punto flotante qué es más preciso $2x_k - ax_k^2$ o $x_k(2 - ax_k)$.

Un otro ejemplo es tomar una función que nos devuelva como su raíz a la raíz cuadrada de a, es decir, queremos $x^* = \sqrt{a}$. Esto se puede conseguir con la función $f(x) = x^2 - a$ que en el método de Newton nos lleva a

$$x_{k+1} \;=\; x_k \;-\; \frac{x_k^2 - a}{2x_k} \;=\; \frac{1}{2}\left(x_k + \frac{a}{x_k}\right), \qquad \text{para todo} \qquad k \geq 0,$$

que de hecho era uno de los métodos empleados por los babilonios para aproximar las raíces cuadradas de los números.

Las dos funciones anteriores y sus raíces muestran hitos de la historia del cómputo científico. Por un lado, encontrar el recíproco de un número no resulta trivial, pues dividir tampoco lo es. De aquí viene la buena costumbre de definir los recíprocos como una variable cuando va ser usado como divisor múltiples veces. Aproximar raíces antes de los logaritmos es fundamental y su uso fue bastante útil.

Por otro lado, el método de Newton se puede ver como parte de un conjunto mayor de conocimientos. En el análisis de la *dinámica del círculo en el círculo* o de la *dinámica discreta*, el estudio de mapeos del tipo $\varphi : [0, 1] \to [0, 1]$ es común y así se puede observar el desarrollo del proceso iterativo $x_{k+1} = \varphi(x_k)$ para $k \in \mathbb{N}$ y una condición inicial x_0 dada.

En este tipo de dinámica es importante encontrar los puntos fijos y el comportamiento en torno de ellos, es decir, buscamos x^* tal que $\varphi(x^*) = x^*$ se satisfaga. Se puede crear una rutina que para cada condición inicial x_0 dada ejecute los siguientes pasos:

1. Grafique las funciones $\varphi(x)$ y $y = x$ (así, la solución está en la intersección de ambas).

2. Haga iterativamente los segmentos de recta que unen el punto (x_k, x_k) con $(x_k, \varphi(x_k)) = (x_k, x_{k+1})$ y el punto (x_k, x_{k+1}) con (x_{k+1}, x_{k+1}).

3. Tome en consideración que el primer punto es mejor que no sea (x_0, x_0) si no $(x_0, 0)$. (Esto último es simplemente por estética.)

Se observa con la función anterior cómo la aproximación de Newton se da de manera gráfica; con la intención se pueda entender mejor el proceso. Es importante observar lo que nos dice la derivada $\varphi'(x^*)$. Por un lado, de la teoría de Sistemas Dinámicos sabemos que es importante que $|\varphi(x^*)| < 1$ se satisfaga para que exista la convergencia, por el otro, el signo de esta derivada nos dice cómo realmente nos aproximamos del punto fijo x^*.

Ejercicio 9: Muestra que el método de Newton es siempre convergente, es decir, muestra que $|\varphi(x^*)| < 1$ se satisface. ¿Qué sucede si se tiene $f'(x^*) = 0$?

Breviario cultural: El método iterativo que se crea con los tres pasos anteriores permite estudiar la dinámica del círculo en el círculo. Además, es un camino para estudiar el *caos*.

Toma el **mapeo logístico** definido por $\varphi(x) := r\,x(1-x)$ y realiza varias rodadas cuando fijas en el intervalo para $x \in [0, 1]$ mientras varías el parámetro r. Observación, los valores posibles de r deben estar en $[0, 4]$. Comienza con números pequeños e incrementa este parámetro para ver el caos surgir en $r_\infty = 3.569946\cdots$. Mucho antes se aprecian fenómenos de *doblamiento de periodo*, por ejemplo, viendo cambios importantes para

$$r \;\in\; \{3,\, 1+\sqrt{6},\, 3.54409\cdots,\, \ldots,\, 1+\sqrt{8},\, \ldots\}.$$

El valor $1+\sqrt{8}$ lo encontró una estudiante de licenciatura en el doblamiento del periodo tres, [10].

3.3. Dos métodos alternativos

Hemos visto ya grandes triunfos o eficiencia en el método de Newton. Hemos visto también como modificarlo para poder atacar dificultades cuando la derivada de la función en cuestión no es sencilla de implementar. Sin embargo, también se ha mencionado en varias ocasiones que sin importar la fuerza de un método, siempre podrá ser construido un ejemplo donde el método falle.

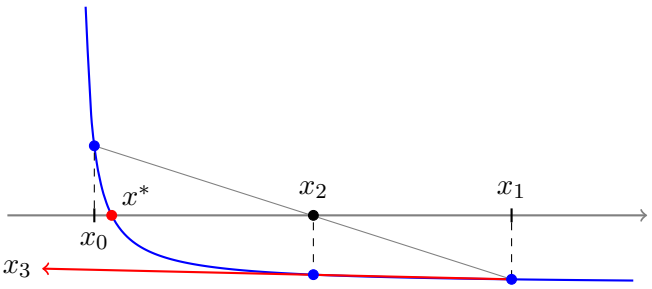
En el caso del método de Newton es complicado encontrar estas funciones, por lo menos, podemos mostrar una donde el método demorará en llegar a la región de convergencia. Supongamos que queremos encontrar la raíz de

$$f(x) \;=\; \frac{1}{x} - a, \qquad \text{cuando} \qquad 0 \;<\; a \;\ll\; 1,$$

digamos a tan pequeño como $\mathcal{O}(10^{-12})$. Si con el método de Newton o el método de la secante

comenzamos con valores de tamaño razonable, estos métodos necesitarán muchas iteraciones para llegar a la región de interés. Tomar aproximaciones lineales o el método de la bisección, no ayudan en gran cosa. Como alternativa, estudiaremos el *método de Muller* capaz de encontrar raíces complejas de una función y el *método fraccional lineal* que no se incomodará con asíntotas.

3.3.1. Método fraccional lineal



Como hemos mencionado el método de la secante puede tener sus inconvenientes, como en el dibujo al lado donde vemos que x_3 puede salir del dominio de definición de la función aún cuando la raíz x^* se encontraba inicialmente en el intervalo formado por los dos puntos iniciales, es decir, $x^* \in [x_0, x_1]$. Se podría pensar en el método de Muller, pero puede ser complicada su implementación con se muestra en el ejercicio al final de la siguiente sección.

La idea es proponer una función que se adecue mucho más a este tipo de situaciones. Digamos que podemos aproximar $f(x)$ por una función

$$\tilde{g}(x) \;=\; \frac{x-a}{bx-c},$$

que teniendo tres parámetros libres satisface algunas propiedades interesantes. Veamos que debe ser un método de tres puntos para llenar los requisitos de a, b y c.

Discusión: Es importante destacar las asíntotas de esta función $g(x)$, así como ver que sabemos de modo directo dónde se encuentra su raíz nula.

Este método no sólo satisface ser una curiosidad de un método alternativo para funciones poco comunes, resulta ser un ejercicio bonito de cómo organizar las ideas a la hora de implementar el algoritmo. Se puede notar que es un paso sencillo en su construcción la localización de la raíz y es de esto que podemos aprovecharnos para localizar el origen en un lugar más conveniente. Tomaremos $y_i = x_i - x_k$ para los índices $i = k,\, k-1,\, k-2$. De este modo,

$$g(y) \;=\; \frac{y-a}{by-c}, \qquad \text{satisface} \qquad f(x_i) \;=\; g(x_i),$$

que es justamente

$$y_i \;-\; a \;=\; f(x_i)\,(by_i - c), \qquad \text{para} \qquad i = k,\, k-1,\, k-2. \tag{3.7}$$

Dado que $y_k = 0$, entonces buscamos $y_{k+1} = a$ o $x_{k+1} = x_k + a$, con lo cual solo nos hace falta

determinar el valor de a.

Comencemos con tres valores `x0`, `x1`, `x2` y sus respectivos valores `f0`, `f1`, `f2` al ser evaluados respectivamente en $f(x)$. Entonces tomamos

$$\texttt{y0} \;=\; \texttt{x0} - \texttt{x2}, \quad \texttt{y1} \;=\; \texttt{x1} - \texttt{x2}, \quad \text{y} \quad \texttt{y2} \;=\; 0.$$

De este modo, tenemos de (3.7):

$$\texttt{y0} - a \;=\; \texttt{f0}\,(b \cdot \texttt{y0} - c) \qquad \text{y} \qquad \texttt{y1} - a \;=\; \texttt{f1}\,(b \cdot \texttt{y1} - c),$$

desde luego, lo más importante es que de esta manera se llega a

$$a \;=\; \texttt{f2} \cdot c;$$

que al sumarse a las dos anteriores nos lleva a

$$\begin{aligned} \texttt{y0} \;&=\; \texttt{f0} \cdot \texttt{y0} \cdot b \,+\, (\texttt{f2} - \texttt{f0}) \cdot c, \\ \texttt{y1} \;&=\; \texttt{f1} \cdot \texttt{y1} \cdot b \,+\, (\texttt{f2} - \texttt{f1}) \cdot c. \end{aligned}$$

Para simplificar la notación se definen los valores `fy0` = `f0 * y0` y `fy1` = `f1 * y1` junto con las diferencias `df0` = `f2` − `f0` y `df1` = `f2` − `f1`, lo cual se traduce en

$$\begin{aligned} \texttt{y0} \;&=\; \texttt{fy0} \cdot b \,+\, \texttt{df0} \cdot c, \\ \texttt{y1} \;&=\; \texttt{fy1} \cdot b \,+\, \texttt{df1} \cdot c, \end{aligned} \qquad \Longrightarrow \qquad c \;=\; \frac{\texttt{fy0} \cdot \texttt{y1} \,-\, \texttt{fy1} \cdot \texttt{y0}}{\texttt{fy0} \cdot \texttt{df1} \,-\, \texttt{fy1} \cdot \texttt{df0}},$$

al usar la *regla de Cramer*.

Ahora lo que necesitamos es el valor de la nueva aproximación, es decir, tenemos

$$\texttt{x3} \;=\; \texttt{x2} + \texttt{f2} \cdot c \;=\; \texttt{x2} + \texttt{f2}\,\frac{\texttt{fy0} \cdot \texttt{y1} \,-\, \texttt{fy1} \cdot \texttt{y0}}{\texttt{fy0} \cdot \texttt{df1} \,-\, \texttt{fy1} \cdot \texttt{df0}}.$$

Con lo cual podemos crear un código limpio con unos pocos pasos de iteración una vez que se tienen los valores `x0`, `x1`, `x2`, para los cuales tenemos que obtener `x3` y su evaluación `f3`.

Código: Núcleo del algoritmo fraccional lineal (PYTHON)

```
y0  = x0 - x2;    y1  = x1 - x2
fy0 = f0 * y0;    fy1 = f1 * y1
df0 = f2 - f0;    df1 = f2 - f1
c   = (fy0*y1 - fy1*y0)/(fy0*df1 - fy1*df0)
x3  = x2 + f2*c
```

Sólo notamos que tiene que haber un reacomodo con los valores de x_k y f_k, aunque también podemos crear un vector con todas las entradas de las coordenadas por las cuales pasa el método como dos vectores o un único arreglo.

3.3.2. Método de Muller

El método de Muller depende de tres condiciones iniciales, es decir, será necesario tener x_k, x_{k-1} y x_{k-2} para obtener la aproximación x_{k+1}. De la Sec. 3.2.1 vemos que este es un método de tres pasos y por lo tanto que su convergencia debe ser de orden $p = 1.84\cdots$.

Vamos un nivel arriba. Hemos tratado de aproximar funciones nolineales de modo lineal con la tangente de la función o con el método de la secante. Podemos hacer interpolaciones superiores al aproximar con curvas de segundo orden, con parábolas, etcétera.

Entonces, el **método de Muller** se define a través de dos sencillos pasos, al tomar tres condiciones iniciales x_2, x_1, x_0 y sus correspondientes alturas $f_i := f(x_i)$ para $i \in \mathbb{N}$, dados por

1. Se encuentra el polinomio de grado dos $g(x)$ que pasa por los puntos $(x_k,\, f_k)$, $(x_{k-1,}\, f_{k-1})$ y $(x_{k-2,}\, f_{k-2})$, es decir, tal que $g(x_i) = f_i$.

2. Tomamos x_{k+1} como la raíz nula de $g(x)$ más próxima de x_k.

Se completa el método al iterar estos dos simples pasos. Sin embargo, hay cuestiones que se deben resolver. Por ejemplo, hacen falta condiciones sobre los nodos x_k, x_{k-1} y x_{k-2} para garantizar la existencia y unicidad de x_{k+1}.

Discusión: ¿Qué nos dicen estas condiciones? ¿Será que alguno de los f_i debe tener un signo distinto a los otros dos? Si tomamos esta condición, entonces también tenemos que tener un criterio para tomar x_{k+1} que conserve la misma situación con los nodos restantes x_k y x_{k-1}.

Además de esto, necesitamos construir la función $g(x)$ en cada paso de la iteración, usando por ejemplo los *multiplicadores de Lagrange* que veremos en la Sec. 8.1.1. Tenemos:

$$\begin{aligned} g(x) &= g(x;\, x_k,\, x_{k-1},\, x_{k-2}) \\ &= f_k \frac{(x-x_{k-1})(x-x_{k-2})}{(x_k-x_{k-1})(x_k-x_{k-2})} + f_{k-1}\frac{(x-x_k)(x-x_{k-2})}{(x_{k-1}-x_k)(x_{k-1}-x_{k-2})} \\ &+ f_{k-2}\frac{(x-x_k)(x-x_{k-1})}{(x_{k-2}-x_k)(x_{k-2}-x_{k-1})}, \end{aligned}$$

donde es fácil ver que se satisface ser un polinomio de grado dos y, que además, $g(x_i) = f_i$ para $i = k,\, k-1,\, k-2$.

Ejercicio 10: Implementa el método de Muller y encuentra las raíces de $f(x) = 1/x - 2$ para $x_k = k$ con $k = 0, 1, 2$. Prueba otras combinaciones en el orden de los nodos. Busca alguna de las raíces de $f(x) = x^2 + 1$, ¿qué observas?

Ejercicio 11: Como con el método fraccional lineal, toma $y_k = x_k - x_2$ y define la parábola $g(y) = ay^2 + by + c$. Muestra que $c = f(x_2)$ y así tienes un sistema de dos ecuaciones con dos incógnitas.

3.4. Comentario final sobre la convergencia

Recordamos que no evaluamos $f(x)$ en la máquina, sino $\tilde{f}(x) = f(x) + e(x)$, con $e(x)$ un error a ser considerado. Así, $\tilde{f}(x)$ es tal que $x \approx x^*$ es una raíz. Sin embargo, esto no es fácil, pues el tamaño del error $e(x)$ es referente al valor de la función $f(x)$. Es por ello que todo este tiempo hemos considerado la raíz con $x \in [a, b]$ tal que $\text{err}(x, x^*) \leq \epsilon$ y no $|f(x)| \leq \epsilon$ que nos daría una mayor incertidumbre.

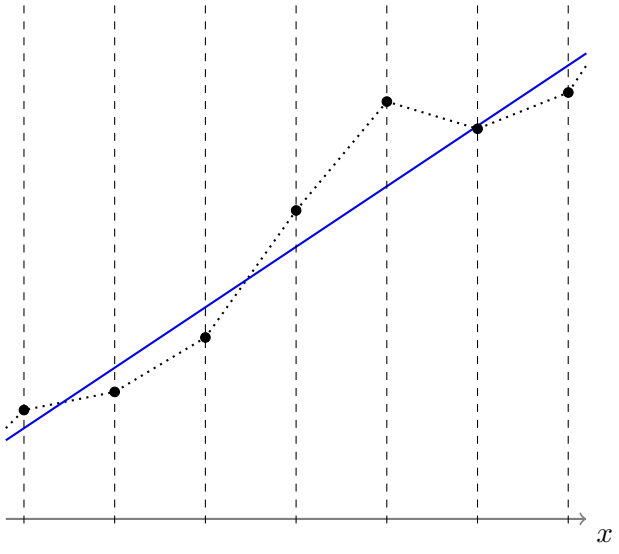
Para entender esta dicotomía cuando se tratan de sistemas lineales con matrices, por favor consulta el Apéndice A. Ahí también se introducen algunos conceptos referentes al Álgebra Lineal como las normas matriciales.

Capítulo 4

Diferenciación numérica

A veces es común pensar que si en el cálculo obtener la derivada de una función resulta en un proceso sencillo y que es al integrar donde nos encontramos un proceso complicado, pues nos llevaremos una sorpresa en esta ocasión. En el mundo numérico resulta que debemos tener más cuidado cuando buscamos la velocidad de una trayectoria discreta que cuando calculamos el área bajo una curva que está dada por partes.

Imaginemos que la función que intentamos aproximar es casi lineal, a modo de la gráfica al lado. Sin embargo, se han introducido los datos de modo descuidado generando pequeños redondeos hacia arriba o truncamientos bobos hacia abajo; de lejos, no tenemos problemas, la función es casi la misma. Juntando los puntos, vemos que el área bajo ambas "curvas" es semejante, pero si nos guiamos por la pendiente a cada dos puntos consecutivos, vemos que la derivada puede variar abruptamente devolviendo inclusive valores negativos.



Aún así, es a partir de la información que tenemos a la mano que podremos aproximar de mejor forma una derivada, teniendo al menos una estimativa del error que se produce en nuestro procedimiento. Para ello usamos normalmente las series de Taylor y con las cotas de error estimar qué es lo que estamos haciendo.

Por ejemplo, para $0 < h \ll 1$ tenemos de la serie de Taylor con residuo que

$$f(x+h) = f(x) + h\,f'(x) + \frac{h^2}{2}f''(\xi), \qquad \text{con} \qquad \xi \in [x, x+h]. \tag{4.1}$$

Como es usual, tenemos que $(h^2/2)f''(\xi) \in \mathcal{O}(h^2)$, así, despejando la derivada tenemos que una

buena aproximación a la derivada puede ser dada por

$$f'(x) \;=\; \frac{f(x\,+\,h)\;-\;f(x)}{h} \;+\; \mathcal{O}(h), \tag{4.2}$$

que llamaremos la **diferencia hacia adelante** con paso h, la denotaremos por $\delta_h^+[f](x)$ o simplemente por δ_h^+ y vemos que tiene precisión de orden lineal, es decir, es $\mathcal{O}(h)$.

En general las series de Taylor serán un gran camino para la construcción de aproximaciones numéricas de derivadas. Es común *discretizar* un espacio haciendo una *malla* o tomando intervalos equiespaciados. En muchas aplicaciones, también obtenemos solamente los valores de las funciones en punto discretos.

Escribamos por extenso la expansión en series de Taylor de $f(x+h)$ y de $f(x-h)$:

$$\begin{aligned} f(x+h) &= f(x) \,+\, h\,f'(x) \,+\, \frac{h^2}{2}\,f''(x) \,+\, \frac{h^3}{3!}\,f'''(x) \,+\, \cdots \\ f(x-h) &= f(x) \,-\, h\,f'(x) \,+\, \frac{h^2}{2}\,f''(x) \,-\, \frac{h^3}{3!}\,f'''(x) \,+\, \cdots \end{aligned}$$

Ahora podemos hacer combinaciones lineales de éstas a modo que $f'(x)$ no desaparezca, de hecho, podemos sumar múltiplos de $f(x)$ para hacer cancelaciones importantes. Por ejemplo

$$\begin{aligned} 1\cdot f(x+h) \;+\; 0\cdot f(x) \;+\; (-1)\cdot f(x-h) & \\ = \;& \left[f(x) \,+\, h\,f'(x) \,+\, \frac{h^2}{2}\,f''(x) \,+\, \cdots\right] \,-\, \left[f(x) \,-\, h\,f'(x) \,+\, \frac{h^2}{2}\,f''(x) \,+\, \cdots\right] \\ = \;& 2h\,f'(x) \,+\, 2\frac{h^3}{6}f'''(x) \,+\, \cdots \end{aligned}$$

con lo cual, al despejar $f'(x)$ se obtiene

$$f'(x) \;=\; \frac{f(x+h)\;-\;f(x-h)}{2h} \;+\; \left(-\frac{f'''(x)}{6}\right) h^2 \;+\; \cdots \tag{4.3}$$

o un método de segundo orden para la derivada. Este se conoce como la **diferencia centrada**

$$\delta_h^0[f](x) \;=\; \frac{f(x+h)\;-\;f(x-h)}{2h} \;=\; f'(x) + \mathcal{O}(h^2).$$

Discusión: En los procedimientos anteriores hemos reducido la cuenta con los puntos suspensivos por un orden fijo, a saber, $\mathcal{O}(h^2)$. ¿Qué sucede con el resto de los términos? ¿Por qué no necesitamos expresar el término $-f'''(x)/6$ de modo explícito?

Ejercicio 12: Calcula la *diferencia hacia atrás* $\delta_h^-[f](x)$ al tomar la serie de Taylor para $f(x-h)$. ¿De qué orden es la diferencia encontrada?

Ejercicio 13: Muestra que se satisface la fórmula

$$\frac{1}{2}\left[\delta_h^+ + \delta_h^-\right][f](x) \;=\; \delta_h^0[f](x).$$

Observa que las diferencias a la izquierda tienen un cierto orden de precisión, ¿esto te dice algo de la precisión de la diferencia centrada?

A veces, sin embargo, es importante tener una cota del error que estamos produciendo al aproximar la derivada. Por ejemplo de la ecuación (4.1) a la ecuación (4.2) vemos que la precisión $\mathcal{O}(h)$ proviene en realidad de una segunda derivada, de hecho,

$$\delta_h^+[f](x) \;=\; \frac{f(x+h) \,-\, f(x)}{h} \;=\; f'(x) \,+\, \frac{h}{2}f''(\xi)$$

con $\xi \in [x,\, x+h]$. Es decir que el error está explícitamente dado por una segunda derivada. Dado que no conocemos la localización exacta de este ξ, estimamos con una cota:

$$\left|\delta_h^+[f](x) \,-\, f'(x)\right| \;=\; \left|\frac{h}{2}f''(\xi)\right| \;\leq\; \frac{h}{2}\max_{\xi\in[x,\,x+h]}|f''(\xi)|. \tag{4.4}$$

Con lo cual, si tenemos una estimativa de la segunda derivada de $f(x)$, entonces podríamos limitar el error a nuestro gusto.

Trabajando de modo semejante, vemos desde (4.3) que

$$\delta_h^0[f](x) \;=\; f'(x) \,-\, \frac{h^2}{6}f'''(\xi), \qquad \text{con} \qquad \xi \;\in\; [x-h,\, x+h],$$

donde notamos que ahora el intervalo es mayor, pero nos conduce a una aproximación de un orden superior.

Ejemplo: El tipo de desarrollos realizados arriba nos permite construir aproximaciones a nuestro antojo, por ejemplo:

$$\begin{array}{rrrcl}
4 & : & f(x+h) & = & f(x) \,+\, h\,f'(x) \,+\, \frac{h^2}{2}\,f''(x) \,+\, \frac{h^3}{3!}\,f'''(x) \,+\, \cdots \\
-3 & : & f(x) & = & f(x) \\
-1 & : & f(x-2h) & = & f(x) \,-\, 2h\,f'(x) \,+\, \frac{(2h)^2}{2}\,f''(x) \,-\, \frac{(2h)^3}{3!}\,f'''(x) \,+\, \cdots \\
\hline
 & & 4f(x+h) \,-\, 3f(x) \,-\, f(x-2h) & = & 6hf'(x) \,+\, \frac{4}{6}h^3f'''(x) \,+\, \frac{8}{6}h^3f'''(x) \,+\, \cdots
\end{array}$$

Entonces

$$f'(x) \;=\; \frac{4f(x+h) \;-\; 3f(x) \;-\; f(x-2h)}{6h} \;-\; \frac{2h^2}{6} f'''(\xi),$$

con $\xi \in [x-2h,\, x+h]$. Es notable que, como en la diferencia centrada, se obtiene una aproximación de orden 2. Sin embargo, el coeficiente del error es el doble. Este tipo de aproximaciones por lo tanto sólo son útiles en sistemas específicos y es por ello que no nos preocuparemos con un nombre formal; las tres diferencias que hemos visto δ_h^+, δ_h^0, δ_h^-, son las más usadas con precisiones de primer y segundo orden.

4.1. Derivadas de orden mayor

Como se ha mencionado las series de Taylor son de gran ayuda no solamente para construir diferencias que aproximen derivadas, como también nos dan una manera de estimar el error producido. Así, de manera análoga podemos calcular aproximaciones para derivadas de orden mayor, por ejemplo,

$$f(x+h) \;-\; 2f(x) \;+\; f(x-h) \;=\; h^2\, f''(x) \;+\; \frac{h^4}{12} f^{(iv)}(x) \;+\; \cdots,$$

con lo cual obtenemos la **diferencia centrada de segundo orden** dada por

$$\delta_h^2[f](x) \;=\; \frac{f(x+h) \;-\; 2f(x) \;+\; f(x-h)}{h^2} \;=\; f''(x) \;+\; \frac{h^2}{12} f^{(iv)}(\xi),$$

con $\xi \in [x-h,\, x+h]$.

Ejercicio 14: Encuentra qué derivada puede aproximarse con la combinación lineal siguiente

$$\frac{-3f(x) \;+\; 4f(x+h) \;-\; f(x+2h)}{2h}.$$

Encuentra también la expresión para el error correspondiente.

Ejercicio 15: Construye una fórmula para aproximar la tercera derivada con los puntos de malla que quieras.

4.2. Limitaciones en el proceso dentro de la máquina

El problema de aproximar una derivada vía una diferencia finita se ve agravado no solamente por los errores entre el método y la derivada real, si no también por factores asociados a cómo se introduce un número con errores por causa de la aritmética de punto flotante. Si denotamos $D[f](x) = f'(x)$ al operador de la derivada y nos olvidamos de denotar explícitamente el lugar x donde se está

evaluando, tenemos de (4.2), por ejemplo,

$$\delta_h^+[f] \; - \; D[f] \; = \; -\frac{h}{2} f''(\xi),$$

con lo cual si $|f''(\xi)| \leq M_2$ para todo $t \in \mathbb{R}$, o en el intervalo de interés, entonces

$$\left|\delta_h^+[f] \; - \; D[f]\right| \; \leq \; \frac{M_2 h}{2}. \tag{4.5}$$

Claramente cuando $h \to 0$, esta fórmula nos dice que el error debe ir a cero también; este es el corazón de las derivadas en el límite vistas en Cálculo Diferencial.

Sin embargo, recordamos que en la máquina nosotros no evaluamos realmente un valor $f(t)$ sino

$$\tilde{f}(t) \; = \; f(t) \; + \; e(t),$$

donde $e(t)$ representa el error al computar $f(t)$, que sabemos que es limitado por $|f(t)|\varepsilon_{\text{M}}$.

Ejercicio 16: Muestra que las diferencias finitas son todas operadores lineales como las derivadas tradicionales, es decir, que satisfacen

1. Para dos funciones f, g cualesquiera, $\delta_h[f+g](x) \; = \; \delta_h[f](x) \; + \; \delta_h[g](x)$.
2. Para una constante α cualquiera, $\delta_h[\alpha f](x) = \alpha \delta_h[f](x)$.

Aquí hemos tomado x como cualquier valor en el dominio de definición de la función tal que sus nodos adyacentes utilizados también lo estén y donde δ_h es una diferencia finita genérica.

Dado que δ_h^+ es un operador lineal, se sigue que $\delta_h^+[\tilde{f}] = \delta_h^+[f] + \delta_h^+[e]$, con lo que tenemos $\delta_h^+[\tilde{f}] - \delta_h^+[f] = \delta_h^+[e]$. En particular si el error estimado es $|e| < \epsilon$ para todo t de interés, se obtiene

$$\left|\delta_h^+[\tilde{f}] \; - \; \delta_h^+[f]\right| \; = \; \left|\frac{e(x+h) \; - \; e(x)}{h}\right| \; \leq \; \frac{2\epsilon}{h}.$$

Combinando con la igualdad (4.4) tenemos que

$$\left|\delta_h^+[\tilde{f}] \; - \; D[f]\right| \; \leq \; \left|\delta_h^+[\tilde{f}] \; - \; \delta_h^+[f]\right| \; + \; \left|\delta_h^+[f] \; - \; D[f]\right| \; \leq \; \frac{2\epsilon}{h} \; + \; \frac{M_2 h}{2}. \tag{4.6}$$

Observa el comportamiento de ambos términos para $h > 0$.

Esta última desigualdad entre la derivada $D[f]$ que queremos evaluar y lo que realmente podemos calcular, $\delta_h^+[\tilde{f}]$, nos muestra que en una máquina no siempre es verdad que al disminuir el espacio entre los puntos, la aproximación es más acertada. De hecho, cuando $h \to 0$ vemos que la cota es inútil pues $2\epsilon/h \to \infty$ quitando cualquier control sobre el error. Una pregunta natural sería

¿podemos encontrar un paso óptimo h^* tal que el lado derecho de (4.6) sea mínimo?

Encontrar el mínimo es sencillo, por lado tenemos una razón monótona que crece cuando $h \to 0$, por el otro, una ecuación lineal que es siempre creciente. Ambas partes son positivas y su suma por lo tanto tiene un único mínimo para $h \in \mathbb{R}$. Llamando el lado derecho de (4.6) de $g(h)$ vemos que

$$g'(h) \;=\; \frac{M_2}{2} \;-\; \frac{2\epsilon}{h^2} \;=\; 0 \qquad \Longleftrightarrow \qquad h^* \;=\; 2\sqrt{\frac{\epsilon}{M_2}}\,,$$

con lo cual en h^* tenemos $|\delta_h^+[\tilde{f}] - D[f]| \leq g(h^*) = 2\sqrt{M_2\epsilon}$. Por ejemplo, si $\epsilon = \varepsilon_{\mathrm{M}}$ y $M_2 = 1$, el menor error que podemos acotar es $2\sqrt{\varepsilon_{\mathrm{M}}} \approx 2.98 \times 10^{-8}$. Recordemos que en general M_2 es libre y $\epsilon = \mathcal{O}(f(x)\varepsilon_{\mathrm{M}})$, lo cual puede mover un poco la situación dependiendo de la función en cuestión.

Ejercicio 17: Los ejercicios que se desprenden de estos análisis son directos:

1. Encuentra el orden de precisión y el paso óptimo para la diferencia centrada de primer orden que aproxima $D[f]$.

2. Aplica la concatenación de diferencias hacia adelante y hacia atrás para mostrar que $\delta_h^2[f] = \delta_h^+\left[\delta_h^-[f]\right] = \delta_h^-\left[\delta_h^+[f]\right]$. ¿Por qué no es buena idea hacer una aproximación $\delta_h^+\left[\delta_h^+[f]\right]$ o $\delta_h^-\left[\delta_h^-[f]\right]$? (Sugerencia, observa el valor x donde evalúas.)

3. Puedes entonces aproximar la tercera derivada con combinaciones de las diferencias hacia atrás y hacia adelante. Por ejemplo, $\delta_h^+\left[\delta_h^-\left[\delta_h^+[f]\right]\right]$ o $\delta_h^-\left[\delta_h^+\left[\delta_h^-[f]\right]\right]$; compárala con la construiste en un ejercicio anterior.
   ¿Será que $\delta_h^+\left[\delta_h^-\left[\delta_h^+[f]\right]\right] = \delta_h^+\left[\delta_h^+\left[\delta_h^-[f]\right]\right]$? ¿Por qué?

4. Encuentra el paso óptimo para evaluar la diferencia centrada de segundo orden.

5. En ocasiones no tenemos todos los nodos que queremos, utilizando la expansión en series de Taylor de $f(x+h)$ y de $f(x+2h)$ construye una aproximación de la primera derivada $f'(x)$ que tenga precisión $\mathcal{O}(h^2)$ y utilice únicamente los elementos $f(x)$, $f(x+h)$ y $f(x+2h)$. Encuentra su paso óptimo.

Capítulo 5

Integración numérica

Hemos dicho que a través del uso de la máquina es más sencillo integrar que derivar; el opuesto de lo que ocurre en el Cálculo Diferencial e Integral. Sin embargo, tenemos que darnos cuenta que ahora nos interesa tener una forma estrictamente cuantitativa de las integrales, pues por ejemplo podemos tener integrales indefinidas como

$$\int e^x \, \mathrm{d}x \; = \; e^x \; + \; C$$

(con C una constante) y no tenerlas para $\int \exp(-x^2) \, \mathrm{d}x$.

En la máquina la cuestión no es tener una fórmula para una integral, sino simplemente un método que nos ayude a evaluar la integral en un dominio definido como un intervalo. De todos modos hay que mencionar que algunas cuantificaciones son las que han dado lugar a funciones definidas a través de sus integrales y no por funciones elementales.

Un caso muy importante en las ramas numéricas, del análisis de perturbaciones y del cálculo numérico es la **función de error** (de Gauss) que se define como

$$\operatorname{erf}(x) \; = \; \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, \mathrm{d}t.$$

Parece que su primer estudio es de 1871 en un artículo en la *Philosophical Magazine (Series 4)* y se debe a J.W.L. Glaisher, quien usó una normalización un poco diferente.

En este capítulo aprenderemos cómo construir fórmulas para la *cuadratura* (o integración) de funciones y veremos que éstas resultan ser mucho más sencillas que las formas de derivación. Además, en general resulta más fácil entender cómo construir una de estas reglas pues son más directas que las deducciones que hacen falta en el caso de las derivadas.

5.1. Cambio de dominio

Es normal dar las reglas de cuadratura para un único intervalo con $x \in [0, 1]$ por ejemplo. Es claro que normalmente este no será el dominio de integración, por ello es importante estudiar cómo cambiar el intervalo $[0, 1]$ por un intervalo genérico $[a, b]$.

Digamos que consideramos la *regla de Simpson*:

$$\int_0^1 f(x)\,\mathrm{d}x \;\approx\; \frac{1}{6}f(0) + \frac{2}{3}f\left(\frac{1}{2}\right) + \frac{1}{6}f(1) \;=\; \frac{1}{6}\left\{f(0) + 4f\left(\frac{1}{2}\right) + f(1)\right\}. \tag{5.1}$$

Como deseamos aproximar la integral en el intervalo $[a, b]$, entonces buscamos un cambio de variables tal que $0 \mapsto a$, $1 \mapsto b$, que al ser de modo lineal, será suficiente. Digamos entonces que tomamos $x = a + (b - a)y$, así $\mathrm{d}x = (b - a)\mathrm{d}y$ y

$$\int_a^b f(x)\,\mathrm{d}x \;=\; (b-a)\int_0^1 f\big(a + (b-a)y\big)\,\mathrm{d}y \;=\; (b-a)\int_0^1 g(y)\,\mathrm{d}y,$$

donde hemos definido $g(y) = f\big(a + (b - a)y\big)$.

Ahora con la regla de Simpson tenemos $g(0) = f(a)$, $g(1/2) = f\big((a+b)/2\big)$ y $g(1) = f(b)$, luego de (5.1) para g tenemos

$$\int_a^b f(x)\,\mathrm{d}x \;\approx\; \frac{b-a}{6}\left\{f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right\}$$

que es la regla de Simpson en el intervalo $[a, b]$.

Sin embargo, comenzaremos con reglas de cuadratura más sencillas que provienen de conceptos del Cálculo Diferencial e Integral y son conocidas como las *sumas de Riemann*.

5.2. Cuadraturas de las sumas de Riemann

En los cursos de Cálculo Diferencial e Integral se define la integral como el área debajo de una curva. Para ello, tomamos pequeños intervalos del dominio de integración y aproximamos el área como la suma ya sea de los paralelepípedos o los trapecios que se generan con los puntos extremos de los intervalos y de sus evaluaciones sobre la función en cuestión.

De estas ideas, veíamos que la **regla del paralelogramo** tenía al menos dos maneras de ser definida, con las evaluaciones a la izquierda del intervalo, $\mathcal{P}^-[f;\, a,\, b]$, o con las evaluaciones a la derecha, $\mathcal{P}^+[f;\, a,\, b]$:

$$\mathcal{P}^-[f;\, a,\, b] \;=\; (b - a)\, f(a) \;\approx\; \int_a^b f(x)\,\mathrm{d}x,$$

$$\mathcal{P}^+[f;\,a,\,b] \;=\; (b\,-\,a)\,f(b) \;\approx\; \int_a^b f(x)\,\mathrm{d}x$$

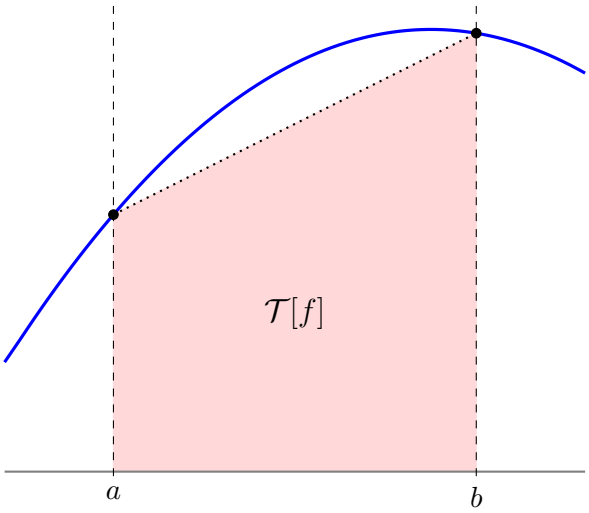
y la **regla del trapecio** como la media de estos dos métodos:

$$\mathcal{T}[f;\,a,\,b] \;=\; \frac{b\,-\,a}{2}\Big(f(a)+f(b)\Big) \;\approx\; \int_a^b f(x)\,\mathrm{d}x.$$

Observa que $\mathcal{T}[f] = (\mathcal{P}^-[f] + \mathcal{P}^+[f])/2$, donde ya no haremos mención del intervalo si no es necesario. Para ver esta igualdad, hay dos modos de verlo, una es geométrica y es suficiente para ver que

$$\mathcal{T}[f] \;\neq\; (b-a)\,f\left(\frac{a+b}{2}\right),$$

que define la **cuadratura del punto medio**; que es la regla del paralelogramo escogiendo el punto medio de cada intervalo.



5.2.1. El error en la regla del trapecio

Otra forma de construir la regla del trapecio es a través de una forma analítica que nos dará suficientes herramientas para poder estimar el error producido en esta aproximación.

De ahora en adelante tomemos h como el tamaño del intervalo a ser usado, sin pérdida de generalidad sea este $I = [0,\,h]$. Así aproximaremos $f(x)$ en I por una línea recta

$$\ell(x) = f(0) + \frac{f(h)-f(0)}{h}x,$$

con lo cual,

$$\begin{aligned}\mathcal{T}_h[f] \;&=\; \int_0^h \ell(x)\,\mathrm{d}x \;=\; \left[f(0)\,x \,+\, \frac{f(h)-f(0)}{h}\,\frac{x^2}{2}\right]_0^h \\ &=\; f(0)\,h \,+\, \frac{f(h)-f(0)}{h}\,\frac{h^2}{2} \;=\; \frac{h}{2}\,[f(0)\,+\,f(h)]\end{aligned}$$

es la regla del trapecio.

Para estimar el error que se comete con la regla del trapecio tenemos que mostrar que para todo $x \in [0,\,h]$ existe $\xi_x \in [0,\,h]$ tal que

$$f(x)\;-\;\ell(x) \;=\; \frac{f''(\xi_x)}{2}\,x(x\,-\,h). \tag{5.2}$$

Para probar esta afirmación usaremos una función auxiliar que ahora parece sacada de la manga, pero que en el Capítulo 7 quedará claro su origen.

Para un valor $x \in [0,\, h]$ dado, definamos la función $\varphi(t) = \varphi(t;\, x)$ como

$$\varphi(t) \;=\; f(t) \;-\; \ell(t) \;+\; c(x)\big[t(h-t)\big],$$

con la constante $c(x)$ a ser definida de modo que $\varphi(x) = 0$ sea satisfecho.

Dado que $\varphi(0) = \varphi(x) = \varphi(h) = 0$, la función $\varphi(t)$ tiene al menos tres ceros en I. Así, del Teorema de Rolle sabemos que $\varphi'(t)$ tiene al menos dos ceros y por lo tanto $\varphi''(t)$ tiene al menos un cero en $[0,\, h]$, digamos en ξ_x. Notemos que

$$\varphi''(t) \;=\; f''(t) \;-\; \cancelto{0}{\ell(t)} \;+\; c(x)[-2].$$

Luego, dado que $\varphi''(\xi_x) = 0$, entonces la constante $c(x)$ es $f''(\xi_x)/2$. Es decir, como tenemos además que $\varphi(x) = 0$, pues al despejar obtenemos (5.2).

Ahora para calcular el error del trapecio, tenemos

$$\begin{aligned}\int_0^h f(x)\,\mathrm{d}x \;-\; \mathcal{T}_h[f] \;&=\; \int_0^h f(x) \;-\; \ell(x)\,\mathrm{d}x \;=\; \int_0^h \frac{f''(\xi_x)}{2}\,x(x-h)\,\mathrm{d}x \\ &=\; \frac{f''(\eta)}{2}\int_0^h x^2 - hx\,\mathrm{d}x \;=\; -\frac{f''(\eta)}{12}\,h^3,\end{aligned}$$

para algún $\eta \in [0,\, h]$, pues el Teorema del Valor Medio se aplica al notar que $x(x-h)$ tiene un único signo. Es decir, que el error en el método del trapecio está relacionado con la segunda deriva de f, tenemos en particular,

$$\left|\int_0^h f(x)\,\mathrm{d}x \;-\; \mathcal{T}_h[f]\right| \;\leq\; \frac{h^3}{12}\,\max_{t\in I}|f''(t)|.$$

Por ejemplo, para $|f''(t)| \leq 1$ para todo $t \in I$ y con $h = 10^{-2}$, el error es menor que 10^{-7}.

Es claro que normalmente no esperamos tener una precisión tan fina si tenemos un intervalo arbitrario $[a,\, b]$. Es por ello que se acostumbra subdividir el intervalo original en subintervalos donde realizaremos la aproximación en cada uno de ellos. Así, si esperamos o queremos n subintervalos, definimos el *mallado* con un espaciamiento $h = (b-a)/n$ y tomamos los puntos $x_k = a + kh$ para $k = 0,\, 1,\, \ldots,\, n$, es decir, tenemos los nodos

$$a \;=\; x_0 \;<\; x_1 \;<\; x_2 \;<\; \cdots \;<\; x_{n-1} \;<\; x_n \;=\; b,$$

donde $x_{k+1} - x_k = h$ sin importar el k en específico que se tome.

Notamos entonces, que cada subintervalo sigue la regla del trapecio con

$$\mathcal{T}_h[f;\, x_{k-1},\, x_k] \;=\; \frac{h}{2}\big[f(x_k) \,+\, f(x_{k-1})\big] \;\approx\; \int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x,$$

con lo cual al juntar todos los subintervalos, se tiene que

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x \;&=\; \sum_{k=1}^{n}\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x \;\approx\; \sum_{k=1}^{n}\mathcal{T}_h[f;\, x_{k-1},\, x_k] \;=\; \sum_{k=1}^{n}\frac{h}{2}\big[f(x_k) \,+\, f(x_{k-1})\big] \\ &=\; h\bigg[\frac{f(x_0)}{2} \,+\, f(x_1) \,+\, \cdots \,+\, f(x_{n-1}) \,+\, \frac{f(x_n)}{2}\bigg],\end{aligned}$$

que se conoce como la **regla del trapecio compuesta** y la denotaremos por $\mathcal{CT}_h[f]$.

Ahora podemos derivar la estimativa del tamaño del error para esta regla compuesta. Juntando las piezas tenemos,

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x \,-\, \mathcal{CT}_h[f] \;&=\; \sum_{k=1}^{n}\left[\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x \,-\, \mathcal{T}_h[f;\, x_{k-1},\, x_k]\right] \\ &=\; -\frac{h^3}{12}\sum_{k=1}^{n} f''(\eta_k) \;=\; -\frac{h^2(b-a)}{12}f''(\eta),\end{aligned}$$

pues $hn = b - a$ y hemos tomado $f''(\eta)$ como el valor que media la suma $\sum_{k=1}^{n} f''(\eta_k)$ por una variante del Teorema del Valor Medio. Es decir, el error puede ser limitado a través de máximo de estas segundas derivadas:

$$\left|\int_a^b f(x)\,\mathrm{d}x \,-\, \mathcal{CT}_h[f]\right| \;\leq\; \frac{h^2(b-a)}{12}M_2 \;=\; \mathcal{O}(h^2),$$

donde, como antes, $M_2 = \max_{t\in[a,\,b]} |f''(t)|$.

5.3. Fórmulas de Newton-Cotes y coeficientes indeterminados

En la teoría, una manera muy elegante de deducir fórmulas para reglas de cuadratura son las llamadas *fórmulas de Newton-Cotes*. En este caso suponemos tener una serie de puntos $a = x_0 < x_1 < \cdots < x_n = b$, que aunados a sus respectivos $f_k = f(x_k)$ para $k = 0,\, 1,\, \ldots,\, n$, pueden generar una aproximación de $f(x)$ vía un polinomio de grado n. Luego, nos gustaría determinar constantes $A_0,\, A_1,\, \ldots,\, A_n$ tales que

$$\deg(f) \;\leq\; n \qquad \Longrightarrow \qquad \int_a^b f(x)\,\mathrm{d}x \;=\; A_0 f_0 \,+\, A_1 f_1 \,+\, \cdots \,+\, A_n f_n,$$

que puede ser exacto dados los grados de libertada del polinomio $f(x)$.

Esta regla tiene una fórmula sencilla de ser calculada, pero costosa y en la práctica no resulta ser tan útil:

Ejemplo: Si $\ell_i(x)$ es el polinomio de grado n tal que $\ell_i(x_j) = 0$ si $i \neq j$ y $\ell_i(x_i) = 1$, entonces los coeficientes de la fórmula de Newton-Cotes satisfacen

$$A_i \;=\; \int_a^b \ell_i(x)\,\mathrm{d}x,$$

pues en particular si tomamos todos los coeficientes así y escribimos

$$f(x) \;=\; f_0 \cdot \ell_0(x) \;+\; f_1 \cdot \ell_1(x) \;+\; \cdots \;+\; f_n \cdot \ell_n(x),$$

entonces notamos que

$$\int_a^b f(x)\,\mathrm{d}x \;=\; \sum_{k=0}^{n} \int_a^b f_k \cdot \ell_k(x)\,\mathrm{d}x \;=\; \sum_{k=0}^{n} f_k \int_a^b \ell_k(x)\,\mathrm{d}x \;=\; \sum_{k=0}^{n} A_k f_k.$$

Un otro modo de ver la validez de estos razonamientos es la siguiente, tenemos

$$\int_a^b \ell_i(x)\,\mathrm{d}x \;=\; \sum_{k=0}^{n} A_k\,\ell_i(x_k) \;=\; A_i \underbrace{\ell_i(x_i)}_{1},$$

pues las otras evaluaciones de ℓ_i son nulas.

Una manera de contornar este problema es colocando funciones que sean fáciles de ser integradas, pero satisfaciendo llegar hasta el grado deseado. De este modo, determinar los coeficientes que satisfacen la fórmula de Newton-Cotes resulta más sencillo. Dado que todos los polinomios de grado n se escriben de modo único, podemos descomponer su escrita en formas más sencillas. Sin ir más lejos, veamos que podemos construir in sistema de ecuaciones para el conjunto de funciones $\{f_0, f_1, \ldots, f_n\}$ y los puntos $\{x_0, x_1, \ldots, x_n\}$ donde cada una de ellas debe satisfacer

$$\int_a^b f_k(x)\,\mathrm{d}x \;=\; A_0 \cdot f_k(x_0) \;+\; A_1 \cdot f_k(x_1) \;+\; \cdots \;+\; A_n \cdot f_k(x_n), \tag{5.3}$$

para $k = 0, 1, \ldots, n$ y los coeficientes $A_0, A_1, \ldots, A_n$ como incógnitas. Es decir, conociendo los valores de las $n+1$ integrales a la izquierda, tenemos que encontrar los $n+1$ **coeficientes indeterminados** a la derecha.

Este método tiene la ventaja de ser más amplio, pero hay que tener cuidado con el conjunto de funciones, estas deben ser un *conjunto linealmente independiente*, es decir, se debe satisfacer que

$$c_0 f_0(x) \;+\; c_1 f_1(x) \;+\; \cdots \;+\; c_n f_n(x) \;\equiv\; 0 \quad\Longleftrightarrow\quad c_0 \;=\; c_1 \;=\; \cdots \;=\; c_n \;=\; 0.$$

Una de las reglas más conocidas, que puede obtenerse con este método, surge al tomar $x_0 = 0$, $x_1 = 1/2$ y $x_2 = 1$, para el intervalo $[0, 1]$. Dado que $n = 2$, necesitamos tres funciones que sean base de los polinomios de segundo grado, por ejemplo, $f_0(x) = 1$, $f_1(x) = x$ y $f_2(x) = x^2$, que ayuda a descomponer un polinomio de grado dos en sus partes más sencillas. Buscamos entonces A_0, A_1, A_2 satisfaciendo (5.3), es decir, tales que

$$\begin{aligned} 1 \cdot A_0 \;+\; 1 \cdot A_1 \;+\; 1 \cdot A_2 \;&=\; \int_0^1 1 \,\mathrm{d}x \;=\; 1\,, \\ 0 \cdot A_0 \;+\; \frac{1}{2} \cdot A_1 \;+\; 1 \cdot A_2 \;&=\; \int_0^1 x \,\mathrm{d}x \;=\; \frac{1}{2}\,, \\ 0 \cdot A_0 \;+\; \frac{1}{4} \cdot A_1 \;+\; 1 \cdot A_2 \;&=\; \int_0^1 x^2 \,\mathrm{d}x \;=\; \frac{1}{3}\,, \end{aligned}$$

que tiene por solución $A_0 = A_2 = 1/6$ y $A_1 = 2/3$. Construimos de este modo la siguiente regla de cuadratura:

$$\int_0^1 f(x) \,\mathrm{d}x \;=\; \frac{1}{6} f(0) \;+\; \frac{2}{3} f\left(\frac{1}{2}\right) \;+\; \frac{1}{6} f(1), \tag{5.4}$$

conocida como la **regla de Simpson**.

Algunos otros usos prácticos de los coeficientes indeterminados serán vistos más adelante, como en el uso de funciones periódicas con bases de funciones dadas, por ejemplo, por $\{\cos x,\, \sin x,\, \cos 2x,\, \sin 2x,\, \dots\}$ cuando las funciones a integrar tienen propiedades específicas. También podemos aprovechar el uso de los nodos $x_0 = 1/4$ y $x_1 = 3/4$ cuando, por ejemplo, la función a integrar tiende a infinito en alguno de los extremos del intervalo; en este caso, podemos tomar $f_0(x) = 1$ y $f_1(x) = x$.

5.4. La regla de Simpson compuesta

Para llegar a la ecuación (5.4), vimos un modo de deducir la regla de Simpson para el intervalo $[0, 1]$. Al igual que lo hemos hecho con anterioridad, podemos construir una regla compuesta. Si dividimos un intervalo genérico $[a, b]$ y tomamos $a = x_0 < x_1 < \cdots < x_n = b$, donde $x_{k+1} = x_k + h$ para $h = (b - a)/n$, entonces

$$\mathcal{CS}_h[f] \;=\; \frac{h}{6} \sum_{k=1}^{n} \left[f(x_{k-1}) \;+\; 4f\left(\frac{x_{k-1} + x_k}{2}\right) \;+\; f(x_k) \right],$$

donde vemos que además de los puntos de malla usuales x_k, necesitamos sus puntos medios

$$x_{k+\frac{1}{2}} \;=\; \frac{x_k + x_{k+1}}{2} \;=\; x_k \;+\; \frac{h}{2}.$$

Comentario: Estos *nodos auxiliares* a la mitad de la malla generan confusión de varias maneras. Por un lado, tenemos que para la regla de Simpson compuesta expresada tenemos n intervalos, éstos tienen a los nodos tradicionales en sus extremos, es decir, $[x_{k-1}, x_k]$ son los subintervalos en los cuales dividimos $[a, b]$; no tenemos intervalos con un nodo auxiliar como extremo. Tal vez valdría la pena incluir a los nodos auxiliares en una notación más sencilla, lo cual utilizan algunos autores. Particularmente, no me gusta esta manera de trabajar, pues en primer lugar, el tamaño del intervalo efectivo es el doble que la separación de los nodos. En segundo lugar, el número de nodos n es solamente válido cuando n es par. Así, por el otro lado, la segunda confusión se da cuando se estudia este método en referencias donde se han incluido los nodos auxiliares como aquellos de la malla, es decir, para $a = x_0 < x_1 < \cdots < x_{2n} = b$ los nodos con índice par generan a los intervalos y los nodos con índice impar son los nodos auxiliares.

¿Cómo saber entonces cuál es la regla que está usando cada autor? Esto es sencillo, pues el valor de h cambia de una representación para la otra. En (5.4) aparece un 6 dividiendo, en la otra convención aparece un 3. Para hacer notoria esta diferencia, tomaré los elementos g_k como los que están definidos a mediados de los nodos.

Retomemos la notación $f_k = f(x_k)$ para los nodos originales de la malla $a = x_0 < x_1 < \cdots < x_n = b$ y escribamos $g_k = f((x_{k-1} + x_k)/2)$, de este modo tenemos

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x &\approx \frac{h}{6}\sum_{k=1}^{n}\left[f_{k-1} + 4g_k + f_k\right] \\ &= \frac{h}{6}\left[f_0 + 4g_1 + f_1 + f_1 + 4g_2 + f_2 + \cdots + f_{n-1} + 4g_n + f_n\right] \\ &= \frac{h}{6}\left[f_0 + 4\sum_{k=1}^{n} g_k + 2\sum_{k=1}^{n-1} f_k + f_n\right],\end{aligned}$$

que es la aproximación de la regla compuesta de Simpson. Las dos sumas sugieren una manera inteligente de evaluar en único ciclo las sumas dentro del código. Si hemos construido los x_k y a partir de ellos hemos evaluado los valores de $f(x)$, tenemos que el corazón de código puede ser dado por el siguiente extracto.

```
simp = f[0] + f[n] + 4*g[n];
for k = 1..(n - 1) do {
    simp = simp + 2*f[k] + 4*g[k];
};
simp = h*simp/6;
```

El extracto arriba está en una notación distinta a Python, por ejemplo. En Matlab no existe

el índice cero y las llamadas de entradas nos son con corchetes sino con paréntesis, vemos que este código no es para ser implementado así, solamente es una guía de su desarrollo, una idea más de lo que podría ser un pseudocódigo. En PYTHON podemos tener algo así:

Código: Regla de Simpson (PYTHON)

```
def simpson(f, a, b, n = 3):
    h = (b - a)/n
    x = numpy.linspace(a, b, n + 1)
    y = x - h/2.0
    simp = f(a) + f(b) + 4.0*f(y[-1])
    for k in range(1, n):
        simp += 2.0*f(x[k]) + 4.0*f(y[k])
    return simp*h/6.0
```

Prueba dejar el ciclo de fuera y calcular la diferencia del tiempo empleado. Para ver esto, puedes usar `time.time()` de la librería `time`.

Ejercicio 18: Implementa el `Código: Regla de Simpson ( PYTHON )` y pruébalo. Coloca los comentarios que explican las partes que no son evidentes. Además, puedes agregarle una documentación en línea colocando comentarios desde la primera línea como

```
" Algoritmo para evaluar la regla de Simpson "
```

Después puedes continuar con los comentarios (`"""`) y así al escribir en la línea de comando

```
In [1]: help(simpson)
```

aparecerá este texto donde puedes dejar un ejemplo para ser ejecutado.

5.4.1. El error en la regla de Simpson

Para calcular el error producido por esta cuadratura, necesitamos la cota de error para la regla de Simpson sencilla. Analizar esta cota no es sencillo. En la regla del trapecio se aprovechó que $\omega(x) = x(x-h)$ siempre tiene el mismo signo para todo el intervalo, en el caso de Simpson necesitamos $\omega(x) = x(x-h/2)(x-h)$ que cambia de signo en el intervalo y complica las cosas.

Tenemos que la regla de Simpson satisface

$$\mathcal{S}_h[f] = \int_0^h q(x)\,\mathrm{d}x = \frac{h}{6}\left[f(0) + 4f\left(\frac{h}{2}\right) + f(h)\right],$$

para $q : [0, h] \to \mathbb{R}$ una función cuadrática tal que $q(0) = f(0)$, $q(h/2) = f(h/2)$ y $q(h) = f(h)$.

Se puede observar que $f(x) - q(x)$ es dependiente de la cuarta derivada de $f(x)$ y de un polinomio

de grado 3, siguiendo en el camino de la prueba de la regla del trapecio, tenemos que

$$f(x) - q(x) \;=\; C\, f^{(4)}(\xi_x)\, \omega(x), \qquad \text{con} \qquad \xi_x \;\in\; [0,\, h]$$

y $\omega(x) = x(x - h/2)(x - h)$, que se anula justamente para $x = 0$, $h/2$ y h.

Como en el caso de la regla de trapecio definimos, para un $x \in [0,\, h]$ fijo, la función auxiliar

$$\varphi(t) \;=\; f(t) \;-\; q(t) \;-\; c(x)[t(t - h/2)(t - h)],$$

con $c(x)$ la constante a ser definida que satisfaga $\varphi(x) = 0$. Dado que $\varphi(0) = \varphi(h/2) = \varphi(h) = \varphi(x) = 0$, entonces tiene al menos cuatro raíces y por lo tanto φ' tiene tres, φ'' dos y finalmente φ''' tiene al menos una raíz en digamos ξ_x.

Notamos que

$$\varphi'''(t) \;=\; f'''(t) \;-\; \cancelto{0}{q(t)} \;-\; c(x)[6],$$

luego

$$\varphi'''(\xi_x) \;=\; 0 \qquad \Longrightarrow \qquad c(x) \;=\; \frac{1}{6}\, f'''(\xi_x).$$

Por lo tanto,

$$\int_0^h f(x)\,\mathrm{d}x \;-\; \mathcal{S}_h[f] \;=\; \frac{1}{6}\int_0^h f'''(\xi_x)\, x \left(x - \frac{h}{2}\right)(x - h)\,\mathrm{d}x,$$

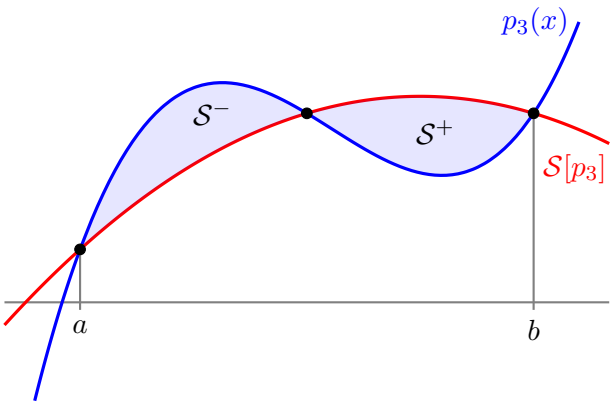
pero ahora $\omega(x)$ cambia de signo en $h/2$.

A modo de tener simplemente una motivación, imaginemos que $f'''(\xi_x)$ es constante, es fácil ver que

$$\int_0^h x \left(x - \frac{h}{2}\right)(x - h)\,\mathrm{d}x \;=\; 0,$$

lo cual demostrará que $\varphi(t)$ tiene un cero adicional. Lo cual indica una casualidad fantástica en la regla de Simpson: fue creada para calcular de modo exacto polinomios de grado dos, pero esta regla aproxima de modo exacto a los polinomios de grado tres.

Para mostrar esta última afirmación, tomemos $f(x) = ax^3 + bx^2 + cx + d = ax^3 + p_2(x)$, con $p_2(x)$ claramente el polinomio de grado dos que deja solamente a la parte cúbica de $f(x)$. Es decir, dado que $\int_0^h p_2(x)\,\mathrm{d}x = \mathcal{S}_h[p_2]$, sólo hace falta ver el error que se produce en la diferencia entre integrar ax^3 o aproximarla con la regla de Simpson. ($\mathcal{S}^- = \int_a^{(a+b)/2} f(x) - p_2(x)\,\mathrm{d}x = \int_{(a+b)/2}^b f(x) - p_2(x)\,\mathrm{d}x = \mathcal{S}^+$.)



Por un lado tenemos que

$$\int_0^h ax^3 \,\mathrm{d}x \;=\; \left[a\frac{x^4}{4}\right]_0^h \;=\; \frac{ah^4}{4}.$$

Por el otro

$$\mathcal{S}_h[ax^3] \;=\; \frac{h}{6}\left[a0^3 \;+\; 4a\left(\frac{h}{2}\right)^3 \;+\; ah^3\right] \;=\; \frac{h}{6}\left[a\frac{3h^3}{2}\right] \;=\; \frac{ah^4}{4}.$$

Con esta simple cuenta se muestra que la regla de Simpson calcula de manera exacta los polinomios de grado tres; lo cual es realmente notable dado que se construyó para polinomios de grado dos. Además, esto resulta muy oportuno, pues con tres puntos sólo usa uno más que en la regla del trapecio, pero con una aproximación dos grados arriba.

Es el cambio de signo de $\omega(x)$ que cancela los términos de manera maravillosa. Por lo tanto la cota de error depende de una derivada arriba y esta es

$$\int_a^b f(x)\,\mathrm{d}x \;-\; \mathcal{S}[f;\,a,\,b] \;=\; -\frac{(b-a)^5}{2880}\,f^{(4)}(\xi),$$

para un valor $\xi \in [a,\,b]$. Utilizando cuatro puntos en el intervalo, también conseguiríamos un método $\mathcal{O}((b-a)^5)$, es por ello, que la regla de Simpson resulta ser una buena opción para aproximar integrales; va un nivel arriba de lo que debería ir.

Para la *regla compuesta de Simpson* también podemos encontrar la cota de error, ésta está dada por

$$\left|\int_a^b f(x)\,\mathrm{d}x - \mathcal{CS}_h[f]\right| \;\leq\; \frac{(b-a)\,f^{(4)}(\eta)}{2880}h^4,$$

para $\eta \in [a,\,b]$.

Comentario: En la literatura el divisor en la cota de error para la regla simple es 90 y en la regla compuesta es 180. La diferencia es gigante en ambos casos contra 2880. Recordemos que cuando toman $h = (b-a)/2$, la división es distinta a lo que nosotros presentamos aquí.

5.5. Cuadraturas de Gauss

Del mismo modo que hemos creado las reglas de cuadratura para el trapecio y Simpson, podemos crear muchas otras. Hay un camino directo que supone tomar los nodos como fijos y equiespaciados; este camino genera las reglas de Newton-Cotes. Sin embargo, el uso de distancias más libres nos permite tener una mejor relación con las curvas que tenemos y la aproximación de éstas para la integral de una función. Como ejemplo, el paralelepípedo con el nodo en el punto medio.

Digamos que usaremos un equivalente a la regla del trapecio para aproximar con una recta la función $f(x)$ en el intervalo $x \in [-1,\,1]$ de modo que el error sea minimizado al suponer cierto

grado en la función como un polinomio dado. La simetría del intervalo se toma por convención. Recordando que la regla de Simpson evalúa de modo correcto polinomios de grado tres, ¿será que podemos hacer lo mismo con sólo dos nodos acomodados de modo especial?

Antes de continuar, deberíamos notar que la simetría del intervalo sugiere que los nodos también tienen que respetar una simetría. Así, un número impar de nodos implicaría en que el nodo $x_k = 0$ debe estar presente.

Ejemplo: Buscamos los valores de los *pesos* o coeficientes indeterminados ω_0 y ω_1 que en conjunto con los nodos x_0, $x_1 \in [-1, 1]$ satisfaga que la regla

$$\int_{-1}^{1} f(x)\mathrm{d}x \;\approx\; \omega_0 f(x_0) \,+\, \omega_1 f(x_1)$$

sea exacta para polinomios de grado tres.

Es decir, como en coeficientes indeterminados, tomamos la base $\{1,\, x,\, x^2,\, x^3\}$ y queremos que se satisfagan las igualdades siguientes

$$\begin{aligned}
\omega_0 \cdot 1 \;+\, \omega_1 \cdot 1 \;&=\; \textstyle\int_{-1}^{1} 1\,\mathrm{d}x \;&=\; 2,\\
\omega_0 \cdot x_0 \,+\, \omega_1 \cdot x_1 \;&=\; \textstyle\int_{-1}^{1} x\,\mathrm{d}x \;&=\; 0,\\
\omega_0 \cdot x_0^2 \,+\, \omega_1 \cdot x_1^2 \;&=\; \textstyle\int_{-1}^{1} x^2\,\mathrm{d}x \;&=\; \dfrac{2}{3},\\
\omega_0 \cdot x_0^3 \,+\, \omega_1 \cdot x_1^3 \;&=\; \textstyle\int_{-1}^{1} x^3\,\mathrm{d}x \;&=\; 0.
\end{aligned}$$

Como vemos tenemos cuatro ecuaciones y cuatro incógnitas, lo cual nos dice que el sistema es completo. Falta ver si tiene solución. La segunda y cuarta igualdad nos remiten que $\omega_0 = \omega_1$ y $x_0 = -x_1$ es un buen camino. Con esto en mente, se puede mostrar que estas igualdades se satisfacen al tomar $\omega_0 = \omega_1 = 1$, $x_0 = -1/\sqrt{3}$ y $x_1 = 1/\sqrt{3}$.

De este modo, podemos definir la **regla de cuadratura gaussiana con dos nodos** por

$$\mathcal{G}^2[f] \;=\; f\left(-\frac{1}{\sqrt{3}}\right) \,+\, f\left(\frac{1}{\sqrt{3}}\right),$$

tal que, $\int_{-1}^{1} f(x)\,\mathrm{d}x - \mathcal{G}^2[f] = \mathcal{O}(M_4)$, con $M_4 = \max_{x\in[-1,\,1]} f^{(4)}(x)$.

Algunas otras de estas cuadraturas están dadas en la siguiente tabla. Los nodos se conocen como los *ceros de Legendre*, al ser las raíces de su n-ésimo polinomio. Una manera de definirlos es con la *fórmula de Rodríguez*:

$$L_n(x) \;=\; \frac{1}{2^n n!}\frac{\mathrm{d}^n}{\mathrm{d}x^n}\left[(x^2-1)^n\right], \qquad \text{para} \qquad n \,\in\, \mathbb{N}.$$

Otra es a través de la regla de recurrencia $(n+1)L_{n+1}(x) = (2n+1)xL_n(x) - nL_{n-1}(x)$, al saber que $L_0(x) = 1$ y $L_1(x) = x$, de la formula de Rodríguez, por ejemplo.

n nodos	Nodos en x_k	Pesos ω_k	Exactas con polinomios
1	0	2	Lineales
2	$\pm\sqrt{1/3}$	1	Cúbicos
3	0	$8/9$	
	$\pm\sqrt{3/5}$	$5/9$	
4	$\pm\sqrt{(3-2\sqrt{6/5})/7}$	$(18+\sqrt{30})/36$	
	$\pm\sqrt{(3+2\sqrt{6/5})/7}$	$(18-\sqrt{30})/36$	
5	0	$128/225$	
	$\pm\frac{1}{3}\sqrt{5-2\sqrt{10/7}}$	$(322+13\sqrt{70})/900$	
	$\pm\frac{1}{3}\sqrt{5+2\sqrt{10/7}}$	$(322-13\sqrt{70})/900$	

Ejercicio 19: Completa la tabla anterior colocando para qué grado de polinomio las reglas de cuadratura gaussiana son exactas.

Veamos con la regla del ejemplo, el cambio de variables para el intervalo $I = [a, b]$. Así, para el intervalo estándar $[0, h]$ tenemos la cuadratura gaussiana $\mathcal{G}_h^2[f]$ dada por

$$\int_0^h f(x)\,\mathrm{d}x \;\approx\; \frac{h}{2}\left(f\left(h\frac{1-1/\sqrt{3}}{2}\right) + f\left(h\frac{1+1/\sqrt{3}}{2}\right)\right).$$

En este caso el error es $\mathcal{O}(h^4)$. Sin embargo, vemos la ventaja de pedir el intervalo simétrico y es por ello que normalmente las cuadraturas gaussianas se realizan en éste.

Ejercicio 20: Escribe, con ayuda de la última aproximación, la regla compuesta para la cuadratura gaussiana con dos nodos, es decir, $\mathcal{CG}_h^2[f]$.

5.6. Tratamiento de singularidades

En la Matemática Computacional y en el Análisis Numérico debemos de preocuparnos siempre con casos especiales. Aunque la teoría sea simple, su implementación a veces no lo es. Si tenemos una función $f(x) \approx c/\sqrt{x}$, entonces su integral en el intervalo $[0, 1]$ existe, pero tanto la regla del trapecio como la de Simpson nos darán errores garrafales, ¿qué es lo que uno debe hacer?

Es claro que no podemos olvidar que el primer intervalo (en caso de usar una regla compuesta) es donde se producirá un error desmesurado si utilizamos una de las reglas de cuadratura vistas hasta ahora. Es por ello que podemos pensar en implementar una regla con Newton-Cotes para $x_0 = 1/4$ y $x_1 = 3/4$, aunque un gráfico muestre que esta regla deja mucho que desear.

Ejercicio 21: Encuentra con coeficientes indeterminados esta regla para $x_0 = 1/4$ y $x_1 = 3/4$ y con las funciones $f_0(x) = 1$ y $f_1(x) = x$.

La regla construida en el ejercicio anterior convergerá lentamente y tendremos el problema que h no puede ser muy pequeño, pues en ese caso, $f(h/4)$ puede producir un *overflow* y con ello la catástrofe de nuestra implementación. Lo mejor que podemos hacer, es incorporar la singularidad en un método que aproxime esta integral.

Digamos que podemos definir una función que satisfaga $g(x) = \sqrt{x}\, f(x)$, con lo cual $g(x) \approx c$ es una constante o tiene poca variación. Entonces, podemos construir una regla que aproxime la integral

$$\int_0^1 g(x)\,\frac{\mathrm{d}x}{\sqrt{x}} \;=\; \int_0^1 \frac{g(x)}{\sqrt{x}}\,\mathrm{d}x \;=\; \int_0^1 f(x)\,\mathrm{d}x,$$

pero con el lado izquierdo con una función $g(x)$ bien comportada al rededor del cero.

Usemos el método de coeficientes indeterminados con $x_0 = 1/4$ y $x_1 = 3/4$ y tomando $g_0(x) = 1$ y $g_1(x) = x$. Utilizando los pasos anteriores, tenemos el sistema

$$\begin{aligned} 2 &= \int_0^1 \frac{1}{\sqrt{x}}\,\mathrm{d}x = A_0\cdot 1 + A_1\cdot 1 \\ \frac{2}{3} &= \int_0^1 \frac{x}{\sqrt{x}}\,\mathrm{d}x = A_0\cdot\frac{1}{4} + A_1\cdot\frac{3}{4} \end{aligned} \qquad\Longrightarrow\qquad A_0 = \frac{5}{3},\; A_1 = \frac{1}{3}.$$

Es decir, hemos construido una regla de cuadratura específica que considera una función $f(x) \approx c/\sqrt{x}$ para x próximo de cero. Esta regla es

$$\begin{aligned} \int_0^1 f(x)\,\mathrm{d}x &= \int_0^1 \frac{g(x)}{\sqrt{x}}\,\mathrm{d}x \;\approx\; \frac{5}{3}g(1/4) + \frac{1}{3}g(3/4) \\ &= \frac{5}{3}\sqrt{\frac{1}{4}}\,f(1/4) + \frac{1}{3}\sqrt{\frac{3}{4}}\,f(3/4) \;=\; \frac{1}{6}\big[5f(1/4) + \sqrt{3}\,f(3/4)\big]. \end{aligned}$$

Esta fórmula cambia muy poco para el intervalo $[0,\, h]$ y debe ser fácil darse cuenta que es

$$\int_0^h f(x)\,\mathrm{d}x \;=\; \frac{h}{6}\big[5f(h/4) + \sqrt{3}\,f(3h/4)\big], \tag{5.5}$$

que debería ser probada contra la fórmula sin peso.

Ejercicio 22: Considera la función $f(x) = (1+x^{10})\,x^{-1/2}$ y aproxímala por la implementación de la regla compuesta que se extrae del Ejercicio 21 y por la regla compuesta para (5.5). Toma el intervalo $[0,\, 1]$ y los valores de $h = 2^{-k}$ para $k = 0,\, 1,\, \ldots,\, 8$. Repite la operación para el intervalo $[0,\, 0.1]$. ¿Qué observas de la convergencia? Explica tus resultados.

5.7. Integración en dos dimensiones

Los métodos vistos en las secciones anteriores pueden ser empleados para calcular integrales múltiples al tratarlas por cada una de sus partes. Por ejemplo, miremos la igualdad en la siguiente integral

$$\int_a^b \int_c^d f(x,\, y)\,\mathrm{d}y\,\mathrm{d}x \;=\; \int_a^b \left(\int_c^d f(x,\, y)\,\mathrm{d}y \right) \mathrm{d}x.$$

Es claro que podemos hacer la aproximación de la integral entre paréntesis resultando en funciones que dependen de x. Luego, cada una de estas puede ser tratada del mismo modo. A saber, podemos crear reglas de cuadratura para integrales de funciones de $\mathbb{R}^2$ en $\mathbb{R}$ usando la regla de Simpson o del trapecio, entre otras.

Ejemplo: Para construir la regla compuesta de Simpson para la integral doble, supongamos que existen h y k tales que $N = (b-a)/h$ y $M = (d-c)/k$ son números naturales. De este modo, para cada $x \in [a,\, b]$ tenemos

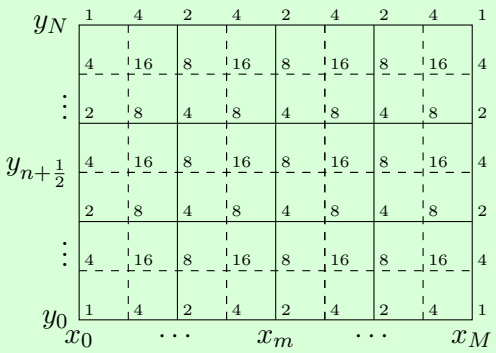
$$\int_c^d f(x,\, y)\,\mathrm{d}y \;\approx\; \frac{k}{6} \sum_{m=1}^{M} \left[f(x,\, y_{m-1}) + 4f\left(x,\, y_{m-\frac{1}{2}}\right) + f(x,\, y_m) \right],$$

donde, como usualmente tenemos $y_0 = c$, $y_m = y_0 + mk$ y $y_{m+\frac{1}{2}} = (y_m + y_{m+1})/2$.

Ahora cada una de las funciones $f(x,\, y_m)$ puede ser aproximada por la misma regla con $x_0 = a$, $x_n = x_0 + nh$ y $x_{n+\frac{1}{2}} = (x_n + x_{n+1})/2$. De este modo,

$$\begin{aligned}
\int_a^b \int_c^d f(x,\, y)\,\mathrm{d}y\,\mathrm{d}x \;&\approx\; \int_a^b \frac{k}{6} \sum_{m=1}^{M} \left[f(x,\, y_{m-1}) + 4f\left(x,\, y_{m-\frac{1}{2}}\right) + f(x,\, y_m) \right] \mathrm{d}x \\
&=\; \frac{k}{6} \sum_{m=1}^{M} \left[\int_a^b f(x,\, y_{m-1})\,\mathrm{d}x + 4\int_a^b f\left(x,\, y_{m-\frac{1}{2}}\right)\,\mathrm{d}x + \int_a^b f(x,\, y_m)\,\mathrm{d}x \right] \\
&\approx\; \frac{hk}{36} \sum_{m=1}^{M} \Bigg[\sum_{n=1}^{N} \left(f(x_{n-1},\, y_{m-1}) + 4f\left(x_{n-\frac{1}{2}},\, y_{m-1}\right) + f(x_n,\, y_{m-1}) \right) \\
&\quad + 4 \sum_{n=1}^{N} \left(f\left(x_{n-1},\, y_{m-\frac{1}{2}}\right) + 4f\left(x_{n-\frac{1}{2}},\, y_{m-\frac{1}{2}}\right) + f\left(x_n,\, y_{m-\frac{1}{2}}\right) \right) \\
&\quad + \sum_{n=1}^{N} \left(f(x_{n-1},\, y_m) + 4f\left(x_{n-\frac{1}{2}},\, y_m\right) + f(x_n,\, y_m) \right) \Bigg].
\end{aligned}$$

Observa que en la última igualdad podemos sacar las sumas y dejar la suma doble antes de los corchetes. Otra manera interesante es ver los nodos en $\mathbb{R}^2$, como el gráfico siguiente.



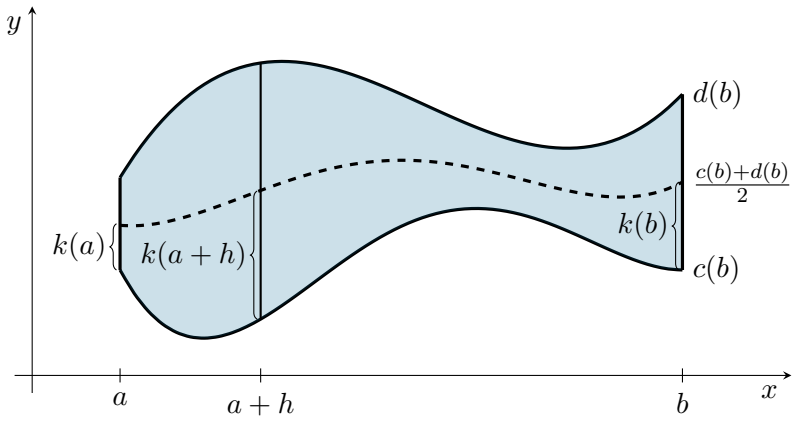
Los nodos están en las coordenadas x_m, y_n, los auxiliares en sus puntos medios; mira las líneas continuas contra las discontinuas. Observa que, bajo productos, los valores de los nodos en las fronteras definen los valores en el interior.

Falta encontrar el error. Para nuestros fines, basta saber que es $\mathcal{O}(h^4, k^4)$. Puedes intentar seguir los pasos arriba para encontrar los valores correctos, solo ten en mente que tendrás que usar el Teorema del Valor Intermedio.

Ejercicio 23: Construye la regla de cuadratura para funciones $f : \mathbb{R}^2 \to \mathbb{R}$ con ayuda de la regla del trapecio. Haz el diagrama sobre los pesos que tiene cada uno de los nodos. Piensa un poco sobre ello, esta idea gráfica es un camino para construir otras reglas de tu interés.

Es importante tener en cuenta que a veces el dominio de integración no es un cuadrado, una de las fronteras puede depender de sus variables, por ejemplo, al integrar

$$\int_a^b \int_{c(x)}^{d(x)} f(x, y)\, \mathrm{d}y\, \mathrm{d}x.$$



La idea es semejante, solo que en lugar de trabajar con una malla de tamaño fijo, es mejor pensar que el espaciamiento depende de la elección de los nodos. Por ejemplo, si tomamos M intervalos en el eje x y N en el eje y. Así, como antes $x_0 = a$ y $x_m = x_0 + mh$ para $h = (b - a)/M$, pero $y_{0;0} = y(x_0)_0 = c(x_0)$ y $y_{n;m} = y(x_m)_n = c(x_m) + nk(x_m)$, donde $k(x_m) = (d(x_m) - c(x_m))/N$ varía según el valor de x. En el esquema arriba, $k(a) < k(b) < k(a+h)$, pero esto puede tener otro orden.

Ejercicio 24: Implementa un código que pida como entradas las funciones $f(x, y)$, $c(x)$ y $d(x)$, los extremos a y b, así como las "resoluciones" N y M. Aproxima la integral

$$\int_{0.1}^{0.5} \int_{x^3}^{x^2} \exp(y/x)\, \mathrm{d}y\, \mathrm{d}x \;\approx\; 0.0333054,$$

con N y M duplicando a cada nuevo intento. Descubre el orden de convergencia de tu método.

Capítulo 6

Números aleatorios

Hemos mostrado ya algunas de las limitaciones de la máquina al tener una recta numérica llena de agujeros. De modo análogo, no podemos esperar que exista un algoritmo determinista que realice la elección de números aleatorios; sin contar que no poseemos todos los números entre cero y uno.

Para ello, se han creado diversos algoritmos que recrean la forma en que podemos recibir un número de modo casi aleatorio. Estos números son llamados de *pseudoaleatorios*, pues siempre podremos recrear la misma secuencia, pero la distribución en la que aparecen recuerda el modo en el que una secuencia aleatoria es formada.

Hay controversia de si este modo de generar los números es suficiente para representar el mundo aleatorio. Sin embargo, a la hora de perder la casualidad, podemos ganar algunos puntos a favor. Los métodos tienen siempre una *semilla* que dice en qué punto de partida comenzará la secuencia deseada. La ventaja de esto es que podemos probar varios algoritmos con la misma secuencia sin necesidad de tenerla almacenada en la memoria, es más, sería complicado tener una gran lista de números si éstos no se crearan bajo la reglas de una función.

La mayoría de los lenguajes e intérpretes de computación tienen rutinas o métodos para generar estos números, Python no es la excepción. Por ejemplo, la siguiente rutina

```
import random
for k in range(5):
    print(k, ": ", random.random())
```

puede tener como resultado

```
0:  0.30904264063598075
1:  0.3275643265455962
2:  0.8514676412321055
3:  0.5518377807642897
4:  0.46126107681705864
```

pero si volvemos a correr la rutina, veremos que los números han cambiado.

```
0:  0.2922810087669848
1:  0.026915223048049053
2:  0.9042458091286675
3:  0.2756673620290111
4:  0.7096623854561892
```

En estos casos estamos usando la librería `random` y parece ridículo tener que escribir todas la veces `random.random()` para generar un número entre cero y uno de modo aleatorio. La sintaxis de PYTHON nos permite llamar inicialmente un método de una librería por su nombre, por ejemplo, en el extracto del código siguiente se usa el comando `from` para simplificar la llada enfrente. En esta rutina siempre tendremos el mismo resultado, haz la prueba.

```
from random import seed
from random import random as rnd
seed(1)
for k in range(5):
    print(k, ": ", rnd())
```

En todas las rodadas obtendrás el mismo resultado sin importar la máquina particular que estés utilizando. Esto sucede porque todas las veces hemos colocado la semilla (`seed`) con el mismo valor, se puede alterar colocando otro número entre paréntesis. Así, otros resultados aparecerán, pero una vez definida la semilla, la secuencia está predeterminada y siempre será ésa.

Una manera tal vez imprecisa de volver al modo aleatorio, es aprovechando los tiempos de la máquina. Los procesos no siempre tardarán lo mismo, pues además de nuestra implementación, hay procesos de fondo en la máquina que llevan tiempo y bajo los cuales no tenemos control. Se me ocurre que el siguiente extracto puede funcionar como una secuencia aleatoria "real".

```
from random import seed
from random import random
from time import time
for k in range(20):
    seed( time() )
    print("{:3}: ".format(k), random())
```

La semilla típicamente necesita números naturales, sin embargo, puede ser usado un número cualquiera.

6.1. Pseudoaleatorios y método de transformación

Tanto en la computación científica teórica como en la criptografía, un *generador pseudoaleatorio* (PGR, por sus siglas en inglés) es un procedimiento determinístico que mapea una semilla en una cadena de números pseudoaleatorios tales que no existe una prueba estadística que pueda distinguirla de una distribución uniforme.

De alguna manera, para dada una cadena de números pseudoaleatorios que pretende ser una distribución uniforme, se deben satisfacer algunos presupuestos que sabemos sobre estas distribucione. Por ejemplo, supongamos que `x` es un vector con estos números y tiene una longitud n muy grande. Entonces, teniendo en cuenta que una variable aleatoria X debe satisfacer

$$\lim_{n\to\infty} \frac{1}{n} \sum_{k=0}^{n} X_k \;=\; \frac{1}{2},$$

es deseable que nuestro vector satisfaga:

1. Para cualesquier $\texttt{i} < \texttt{j} < n$,

$$\frac{1}{\texttt{j} - \texttt{i} + 1} \sum_{k=\texttt{i}}^{\texttt{j}} \texttt{x[}k\texttt{]} \;\approx\; \frac{1}{2},$$

   donde esta aproximación srá más próxima mientras la distancia entre `i` y `j` sea mayor.

2. Además, para cualquier subintervalo $A = [a,\, b] \subset [0,\, 1]$, nos gustaría que

$$\sum_{k=\texttt{i}}^{\texttt{j}} \mathbb{1}_A(\texttt{x[}k\texttt{]}) \;\approx\; b - a,$$

   donde $\mathbb{1}_A(x)$ vale uno si $x \in [a,\, b]$ y nulo en caso contrario; nuevamente, más preciso mientras sea mayor la muestra.

Podemos colocar muchas otras pruebas que se nos ocurran. Esto no es sencillo, pues las muestras tienen que ser grandes lo suficiente para que la *Ley de los grandes números* tenga validez. Es decir, hay dos formas en que una secuencia de desviaciones normales pseudoaleatorias puede considerarse insatisfactoria. La primera es que su distribución puede no ser una representación adecuada de la distribución deseada y la segunda es que la secuencia puede no constituir una muestra suficientemente aleatoria a partir de esa distribución.

Se han considerado muchas clases diferentes de pruebas estadísticas en la literatura, entre ellas la clase de todos los circuitos booleanos de un tamaño determinado. No se sabe si existen buenos generadores pseudoaleatorios para esta clase, pero se sabe que su existencia es en cierto sentido equivalente a los límites inferiores de circuitos en la teoría de la complejidad computacional. Sin embargo, la base de estas pruebas no están completamente demostradas.

Además, estas pruebas son complicadas en su notación y no son relevantes para el estudio de este curso. Por ahora será suficiente saber que estas secuencias de números pseudoaleatorios se generan a través de complicados mecanismos que emulan la aleatoriedad con la distribución deseada.

En PYTHON hay varios métodos para generar números de manera pseudoaleatoria, algunas de estas son

- `random.random()`, para generar un número entre cero y uno con distribución uniforme.
- `random.randint(1, 6)`, para generar un tiro de dado; dará valores enteros entre 1 y 6.
- `random.gauss()`, para que generar un número con distribución gaussiana.

Otros métodos importantes de la librería `random` son `choice(lista)`, `sample(lista, n)` y también `shuffle(lista)` que elegirán un elemento, una cantidad `n` de elementos o que revolverán los elementos de la `lista`, respectivamente.

Suponiendo que `random.random()` nos retorne un número aleatorio con distribución normal, digamos x, es fácil pensar que el número que podemos obtener con el tiro de un dado esté dado por la sencilla fórmula

$$d \;=\; \lceil 6\,x \rceil.$$

Fórmulas semejantes podemos crear para `choice`, `sample`, `shuffle`, pero ¿cómo construir la distribución gaussiana a partir de la normal?

Ejercicio 25: Construye a partir de `random.random()` una distribución gaussiana. Toma en cuenta que el máximo número en DP (y por lo tanto el mínimo) es aproximadamente $\pm 1.7977 \times 10^{308}$.

Es de uso común, dada su sencillez, la transformación de Box-Muller. Esta, a grandes rasgos es dada por un método iterativo para generar números pseudoaleatorios desde una distribución normal.

Sin adelantarnos tanto, veamos un modo de constituir una secuencia de números revuelta. Tomando el multiplicador b, el módulo m y la semilla y_0 como números naturales, se puede construir la secuencia de números naturales $\{y_k\}_{k\in\mathbb{N}}$ por

$$y_{k+1} \;\equiv\; b\,y_k \qquad (\mathrm{mod}\ m).$$

A partir de ésta, tenemos que $\{x_k\}_{k\in\mathbb{N}}$ definida por $x_k = y_k/m$ está distribuida de manera uniforme en $[0,\,1)$. Es claro que mientras mayor sea el número m nuestra resolución en el intervalo unitario será mejor. Además se sabe que mientras más próximo sea b de $m/2$ los números tendrán patrones menos claros, piensa por ejemplo en $b = 2$ o $b = 1$. Aún así, no parece ser una secuencia aleatoria. Es deseable también que $\mathrm{mcd}(b,\,m) = 1$ se satisfaga.

Discusión: Pensar qué valores tomar para b y m es importante, éstos deben de estar determinados por la precisión con la cual la máquina está trabajando. Ya la semilla es una elección propia que puede ser utilizada según cada momento, permitiendo de este modo, tener una condición inicial y dar siempre la misma secuencia.

Entonces, si tenemos una secuencia que se encuentra distribuida como deseamos, lo único que hace falta es que sea dada de manera aleatoria. El método Box-Muller, se aplica correctamente a dos desviaciones uniformes independientes a_1 y a_2 en $(0, 1)$ dando como resultado dos nuevos valores. Consiste en usar las transformaciones

$$z_1 \;=\; (-2\ln a_1)^{1/2}\operatorname{sen} 2\pi a_2 \qquad \text{y} \qquad z_2 \;=\; (-2\ln a_1)^{1/2}\cos 2\pi a_2$$

para dar dos desviaciones normales estándares independientes z_1 y z_2. Ahora, nuestra secuencia $\{x_k\}_{k\in\mathbb{N}}$ puede ser transformada bajo esta transformación al tomar de modo iterativo $a_1 = x_{2k}$ y $a_2 = x_{2k+1}$ y los equivalente z_1 y z_2 generarán nuestra nueva secuencia.

Ejercicio 26: Encuentra el rango de las transformaciones de Box-Muller para valores a_1 y a_2 en $(0, 1)$ cualesquiera. Observa qué sucede si intercambias de lugar tus condiciones iniciales.

Existen otros métodos alternativos para transformar los números del intervalo unitario. Sin embargo, esta transformación es respetada por la sencillez de sus operaciones desde el punto de vista numérico; las librerías para logaritmos y funciones trigonométricas están prácticamente en las bases de cualquier lenguaje.

Una otra versión es la *forma polar* al tomar valores u, v en el intervalo $[-1, 1]$ bajo una distribución normal. Las transformaciones son dadas por

$$z_1 \;=\; u\sqrt{\frac{-2\ln s}{s}} \qquad \text{y} \qquad z_2 \;=\; v\sqrt{\frac{-2\ln s}{s}},$$

para $s = u^2 + v^2$; s se toma como la unidad cuando u y v son nulas. Esta versión tiene la ventaja de no usar funciones trigonométricas.

6.2. Integración de Montecarlo

Los métodos Montecarlo tal vez tuvieron su primera aparición en el *problema de Buffon* o *aguja de Buffon* del s. XVIII formulado por Georges-Louis Leclerc y Comte de Buffon. Sin embargo, el primer problema donde Montecarlo aparece es sólo en 1930 en una solución dada por Enrico Fermi al estudiar la difusión del neutrón. La versión moderna se la debemos a Stanislaw Ulam y data de 1940.

Ulam trabajaba en armas nucleares y John von Neumann se interesó de inmediato en él. Esto le dio un impulso dentro del programa ENIAC (Computador e Integrador Numérico Electrónico, por su siglas en inglés) que fue una de las primeras computadoras de propósito general. Posteriormente, esto lo llevó a estudios más rigurosos en el Laboratorio Nacional de Los Álamos.

El gran uso de los métodos Montecarlo es aproximar soluciones deterministas pero complejas, por números pseudoaleatorios. Dado el *Teorema de los grandes números* sabemos de su convergencia cuando el número de datos N tiende al infinito. Además, el *Teorema del límite central* garantiza que el error decae como $\mathcal{O}(1/\sqrt{N})$.

Para ejemplificar estos métodos, podemos tomar la aguja caída en el s. XVIII.

Problema de Buffon: Se trata de lanzar una aguja sobre un papel en el que se han trazado rectas paralelas distanciadas entre sí de manera uniforme. Se puede demostrar que si la distancia entre las rectas es igual a la longitud de la aguja, la probabilidad de que la aguja cruce alguna de las líneas es $2/\pi$.

De esa manera,

$$\pi \approx \frac{2N}{A},$$

donde N es el número total de intentos y A el número de veces que la aguja ha cruzado alguna de las líneas.

Por otro lado, podemos también crear métodos numéricos para aproximar integrales, uno sencillo sería emulando la regla del trapecio

$$MC[f] = (b-a)\frac{1}{N}\sum_{k=1}^{N} f(X_k) \approx \int_a^b f(x)\,\mathrm{d}x,$$

donde X_k es una variable aleatoria uniformemente distribuida en el intervalo $[a, b]$. Un código sencillo es así el siguiente.

Código: Integración Montecarlo (PYTHON)

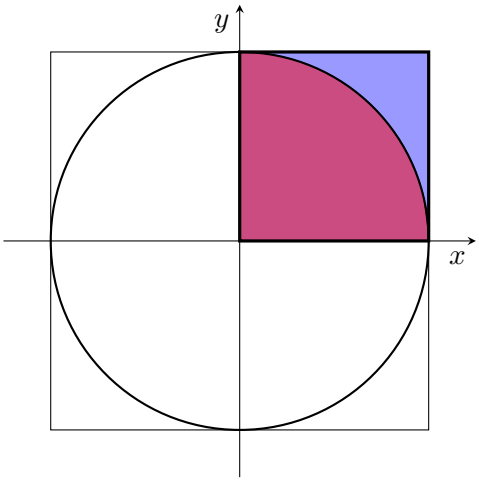
```
def montecarlo(f, a, b, N):
    " Este es un pequeño código para aproximar integrales.
      Utilizamos el método Montecarlo con 'N' desviaciones aleatorias.
      'a' y 'b' representan el intevalo y 'f' la función. "
    suma = 0.0
    I = b - a
    for k in range(N):
        suma += f( a + I*random.random() )
    return I*suma/N
```

Para ver el comportamiento asintótico del error, tenemos que entender el número de intentos N y relacionarlo con el número de intervalos o considerarlo como el espaciamiento de la malla h de las reglas de cuadratura vistas en el capítulo anterior. Por ejemplo, recordando la regla del paralelogramo, tenemos la relación $hn = b - a$, consideremos entonces $h = (b - a)/N$.

Discusión: Piensa cómo podríamos usar un método de Simpson modificado y obtener así una convergencia más rápida.

Es claro que estos métodos no parecen ser tan eficaces como las reglas del Capítulo 5. El uso de la integración Montecarlo es más efectiva cuando el objeto que queremos integrar es más complicado, por ejemplo, el área de una estrella u otro dominio limitado. Como un ejemplo sencillo para cambiar este paradigma, tomemos el círculo de radio unitario que tiene área π.

Podemos tomar una cuarta parte del dominio al usar el cuadrado $[0,\, 1] \times [0,\, 1]$ en el primer cuadrante. Ahora al tomar puntos normalmente distribuidos, aportarán a la cuenta cuando individualmente cada punto esté en el interior del círculo y no lo harán cuando esté fuera, es decir,



$$\lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \mathbb{1}_B(\mathbf{X}_k) \;=\; \frac{\pi}{4} \;=\; \int_0^1 \sqrt{1 - x^2}\, \mathrm{d}x$$

debe ser satisfecho al tomar $B = B(0;\, 1)$ como el disco unitario y $\mathbf{X}_k = (X_k,\, Y_k)$ una variable aleatoria distribuida normalmente en el cuadrado.

Ejercicio 27: Implementa el método Montecarlo para vectores en $\mathbb{R}^2$ y así aproximar el valor de π. Considera dominios más extravagantes como una estrella de cinco picos.

Breviario cultural: El nombre de los métodos Montecarlo proviene de un barrio del principado de Mónaco, célebre por el Casino de Montecarlo y sus juegos de azar. Es por ello que en inglés se escribe Monte Carlo y en otras lenguas Monte-Carlo.

Capítulo 7

Ecuaciones lineales

Hemos llegado a un punto muy importante de la matemática computacional. Es común en matemáticas, y más en el análisis y el cálculo numérico, tratar de resolver problemas nolineales por aproximaciones lineales. Por un lado, los primeros son bastante comunes en nuestro entorno de la naturaleza. Por el otro, hacer buenas predicciones nolineales es complicado y sí entendemos de manera bastante acertada la linealidad. Esta es una fuerte razón para hacer este tipo de aproximaciones en matemáticas. Esto es tan común que se dice coloquialmente que en todo problema numérico siempre se pasará por un problema lineal.

Es claro que la última afirmación es discutible, pues a lo largo de este curso no siempre hemos pasado por un problema lineal desde el punto de vista del álgebra lineal, al menos. Tal vez siendo francos veremos que en ocasiones hemos hecho algunos rodeos para no atacar el problema de esta manera, o su abordaje lineal está disfrazado en la derivada en el método de Newton o en en la cuadratura del paralelogramo o del trapecio, solo por mencionar algunos ejemplos.

Sin más demoras, recordemos los conceptos básicos del álgebra lineal, pues es verdad que todo problema lineal se lleva a esta forma. Hay que recordar a las matrices $A \in \mathbb{R}^{n\times m}$ escritas como

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix},$$

donde los coeficientes a_{ij}, con $i, j \in \mathbb{N}$, pertenecen a los reales. Los vectores los consideramos naturalmente como columna, por ejemplo $\mathbf{x} \in \mathbb{R}^{n\times 1}$, pero los colocamos en el espacio $\mathbb{R}^n \approx \mathbb{R}^{n\times 1}$.

Utilizamos letras en negritas $\mathbf{x}$ para vectores cuyas entradas son los coeficientes x_i en los reales; utilizamos la misma letra. En el caso de las matrices, se usa normalmente una mayúscula para el nombre que está asociada a sus entradas en minúsculas. Podemos recordar algunas operaciones clásicas:

- Suma de A y B cuando ambas pertenecen al mismo $\mathbb{R}^{n\times m}$. Aquí $C = A+B$ tiene $c_{ij} = a_{ij}+b_{ij}$.
- Producto de A con B cuando $A \in \mathbb{R}^{n\times \ell}$ y $B \in \mathbb{R}^{\ell\times m}$. Su resultado es una matriz en $\mathbb{R}^{n\times m}$, a ℓ se le llama la *dimensión interior* y a n, m las *exteriores.* En este caso, para $C = AB$ tenemos $c_{ij} = \sum_{k=1}^{\ell} a_{ik}b_{kj}$.
- El producto por un escalar $\alpha \in \mathbb{R}$ es $\alpha A = (\alpha \cdot a_{ij})$, donde el coeficiente entre paréntesis dice la forma de cada entrada.
- La matriz transpuesta de $A \in \mathbb{R}^{n\times m}$ se encuentra en $\mathbb{R}^{m\times n}$ y $A^{\mathrm{T}} = (a_{ji})$.

También hemos hablado de una operación poco conocida, que puede ser extendida, a saber,

- El útil *producto de Hadamard*, $A \odot_H B = (a_{ij} \cdot b_{ij})$. Esto se puede extender a cualquier operación como lo realiza Python, pues todas las operaciones matriciales se hacen entrada a entrada, es decir, `*` es este producto $\odot_H$ o `**` elevará cada elemento a una cierta potencia, de hecho, se tiene
$$A \texttt{**} p \;=\; (a_{ij}^{p}) \qquad \text{y} \qquad A \texttt{**} B \;=\; (a_{ij}^{b_{ij}}),$$
para un número real p o dos matrices A y B del mismo tamaño. Además, para funciones como el seno, podemos tomar $\operatorname{sen}_H(A) = (\operatorname{sen} a_{ij})$. Notemos por ejemplo que $A + B = A \oplus_H B$.
- En Python hay varios modos de hacer el producto entre matrices. Nosotros optaremos por la arroba, `A@B`, por ejemplo.

Así como algunas operaciones pueden ser utilizadas fuera del contexto clásico del álgebra lineal, algunas características de las matrices son importantes en el uso de computadoras.

Matrices de rango uno

Un resultado clásico del álgebra lineal y que es sencillo de probar es que si $\mathbf{x} \in \mathbb{R}^n$ y $\mathbf{y} \in \mathbb{R}^m$, entonces la matriz
$$W \;=\; \mathbf{x}\,\mathbf{y}^{\mathrm{T}} \;=\; \begin{pmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_m \\ x_2y_1 & x_2y_2 & \cdots & x_2y_m \\ \vdots & \vdots & \ddots & \vdots \\ x_ny_1 & x_ny_2 & \cdots & x_ny_m \end{pmatrix}$$
es de rango uno, *i.e.*, en notación matemática $\operatorname{rango}(W) = 1$. Es fácil mostrar que el converso es también válido.

Ejercicio 28: Prueba las dos direcciones de esta afirmación.

Si tenemos entonces una matriz de rango uno, podemos escribirla como W. En caso de querer el producto $\mathbf{c} = W\mathbf{b}$, podemos construir la matriz W y multiplicar por $\mathbf{b}$ como en el caso tradicional,

obteniendo así que cada entrada de $\mathbf{c} = (c_i) = (\sum_{j=1}^{m} w_{ij} b_j)$. Veamos que para realizar esto hemos hecho un total de m productos y $m-1$ sumas, al tener n entradas usamos $n(2m-1)$ operaciones. Si además debemos construir W, cada entrada de la matriz es un producto y esto nos lleva tener un total de $3nm - n$ operaciones.

En cambio, podemos hacer uso de la forma de W y escribir el producto como

$$\mathbf{c} \;=\; W\mathbf{b} \;=\; (\mathbf{x}\,\mathbf{y}^{\mathrm{T}})\,\mathbf{b} \;=\; \mathbf{x}\,(\mathbf{y}^{\mathrm{T}}\,\mathbf{b}) \;=\; (\mathbf{y}^{\mathrm{T}}\,\mathbf{b})\,\mathbf{x},$$

donde la última igualdad refleja el hecho de que $\mathbf{y}^{\mathrm{T}}\,\mathbf{b} = \sum_{j=1}^{m} y_j b_j$ es un escalar.

Ejercicio 29: Calcula el número de operaciones que se realizan de este modo. Observa el comportamiento cuando n crece. Toma $W \in \mathbb{R}^{n\times n}$ para $n = 10$ y supongamos que el producto lleva 1ns (nanosegundo) en la máquina, ¿cuánto tiempo es necesario para $n = 100, 1000, \ldots, 10^k$. Este tipo de cálculos son mejorías en los *multiplicadores de Householder* o parte del raciocinio en la *descomposición en valores singulares*, por ejemplo.

7.1. Teoría de sistemas lineales

Esta área de estudio es parte de lo que probablemente sea una de las teorías matemáticas más desarrolladas, el álgebra lineal. Además, el cómputo científico y el álgebra lineal están bastante relacionadas desde los inicios del uso de los "ordenadores". De ahora en adelante, nos preocuparemos bastante por resolver el problema clásico del álgebra lineal, es decir, encontrar un vector $\mathbf{x} \in \mathbb{R}^m$ tal que para dadas una matriz $A \in \mathbb{R}^{n\times m}$ y un vector $\mathbf{b} \in \mathbb{R}^n$ se satisfaga la ecuación lineal

$$A\mathbf{x} \;=\; \mathbf{b}. \tag{7.1}$$

Es entonces de suma importancia tener una idea de cómo resolver esta ecuación.

Discusión: Si podemos garantizar que (7.1) tiene solución, ¿cuándo ésta es única? ¿qué más podemos preguntarnos de este sistema desde el punto de vista de la matemática computacional? Es más, cuando tiene una única solución y $n = m$, ¿cómo se resuelve naturalmente este problema?

7.1.1. Inestabilidad de la inversa de una matriz

Sabemos que si A es cuadrada y no singular, entonces la solución de (7.1) es única y ésta es justamente $\mathbf{x} = A^{-1}\mathbf{b}$, donde $A^{-1} = \mathrm{inv}(A)$ se conoce como la *matriz inversa* de A y satisface $A^{-1}A = AA^{-1} = I$, para I la *matriz identidad* con los elementos en la diagonal principal idénticos

a uno y cero en caso contrario, es decir, para $I = (i_{ij})$ tenemos que $i_{ii} = 1$ y $i_{ij} = 0$ si $i \neq j$.

Sin embargo, en la máquina, en general no resulta ser buena idea el concepto entero de buscar la inversa de una matriz. Consideremos por ejemplo $A \in \mathbb{R}^{50\times 50}$ tal que $|A| = \det(A) = 1$. De este modo, A tiene rango total y la inversa de ella nos dará la solución de un sistema (7.1) sin importar el vector $\mathbf{b}$. ¿Qué tiene de malo el raciocinio anterior? No tiene ningún problema aparente, sin embargo, se puede tomar el rescalamiento $B = A/10$, de modo que $|B| = (1/10)^{50}|A| = 10^{-50}$ es realmente un número pequeño. No hay una herramienta clara para saber si es un redondeo grande o no, y por lo tanto, la matriz B puede ser considerada como una matriz casi singular, que seguramente dará problemas en su implementación.

Lo más importante es ver que en general la substitución con la inversa de A no se aconseja. Por ejemplo, el algoritmo:

1. Calcule $C = A^{-1}$,
2. Resuelva con $\mathbf{x} = C\mathbf{b}$,

no es (salvo casos específicos) un buen camino. Por un lado, el número de operaciones es muy elevado; sólo para calcular $C\mathbf{b}$, se requieren $2n^2 - n$ operaciones; crece cuadráticamente con el tamaño de la matriz. Por el otro lado, la inestabilidad del primer paso puede hasta ser intuida en el caso escalar.

Ejemplo: Resolvamos $10x = 5$ con el algoritmo:

1. Tenemos $C = 1/10$, que no es un valor en la aritmética de punto flotante (introduciendo errores de redondeo),
2. luego con $x_N = \text{fl}\left(\frac{1}{10}\right)5 \approx \frac{1}{2}$

vemos que la solución numérica x_N no es la exacta, pues ésta es $x = \text{fl}(0.5) = 0.5$. Claro que el error está acotado por un múltiplo del épsilon de la máquina, sin embargo, hay error. Nuevamente hay un camino certero, pues la substitución directa nos hubiera llevado a $x = 5/10 = 1/2 = 0.5$ de manera exacta.

7.1.2. Factorización LU

La alternativa es considerar una factorización de A que ayude en su resolución. Tomando ésta de la forma $A = LU$, vemos que resolver (7.1) se transforma en $LU\mathbf{x} = \mathbf{b}$. Si llamamos de $\mathbf{y}$ a $U\mathbf{x}$, resolvemos primero $L\mathbf{y} = \mathbf{b}$ y luego con el valor de $\mathbf{y}$, encontramos $\mathbf{x}$ al resolver $U\mathbf{x} = \mathbf{y}$.

Discusión: ¿Qué ventaja tiene esto? Hemos cambiado un simple problema $A\mathbf{x} = \mathbf{b}$ por dos de ellos, el primero $L\mathbf{y} = \mathbf{b}$ y el segundo $U\mathbf{x} = \mathbf{y}$.

Sí tiene ventaja, pues ésta proviene de resolver el problema de manera recursiva en los sistemas triangulares, los cuales son sencillos de solucionar y con un número bajo de operaciones.

El sistema triangular inferior

Tenemos que L es una matriz triangular inferior, es decir, $l_{ij} = 0$ si $i < j$, luego

$$L = \begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{pmatrix}.$$

Queremos resolver $L\mathbf{y} = \mathbf{b}$ y esto se puede hacer con substitución "hacia adelante". Es decir, vemos que esta ecuación se escribe como el sistema

$$\begin{aligned} b_1 &= l_{11}y_1 \\ b_2 &= l_{21}y_1 + l_{22}y_2 \\ b_3 &= l_{31}y_1 + l_{32}y_2 + l_{33}y_3 \\ &\vdots \\ b_n &= l_{n1}y_1 + l_{n2}y_2 + \cdots + l_{nn}y_n \end{aligned}$$

que podemos resolver secuencialmente, pues la primera ecuación es

$$y_1 = \frac{b_1}{l_{11}},$$

así encontramos el segundo valor

$$y_2 = \frac{b_2 - l_{21}y_1}{l_{22}},$$

pues conocemos el valor y_1. Siguiendo de esta manera, el tercer valor es

$$y_3 = \frac{b_3 - l_{31}y_1 - l_{32}y_2}{l_{33}},$$

desde y_1 y y_2. En general llegamos a los términos con

$$y_k = \frac{b_k - \sum_{j=1}^{k-1} l_{kj}y_j}{l_{kk}},$$

donde los valores y_1, y_2, …, y_{k-1} ya son conocidos. Observamos también que los valores b_1, b_2, …, b_{k-1} no son más utilizados, así, pueden ser aprovechados en un algoritmo computacional. Este tipo de prácticas ya no se utilizan tanto, pues ahora los problemas de memoria `RAM` y las capacidades de las computadoras son mucho mayores.

El *pseudocódigo* escrito en la notación de Python puede ser resumido en unas pocas líneas, mostrando que la práctica del reuso de elementos numéricos, también puede ser eficiente en la manera de escribir un algoritmo:

Código: Substitución para adelante por líneas (PYTHON)

```
for i in range(n):
    for j in range(i):
        b[i] = b[i] - L[i, j]*b[j]
    b[i] = b[i]/L[i, i]
```

Advertencia: En PYTHON tenemos que tener cuidado de cómo pasamos los argumentos de los arreglos (matrices y vectores) en las funciones. PYTHON utiliza referencias para ésto, así que dentro de una función podemos modificar un elemento exterior como si fuera una variable global. Hay que tener esto en mente o pensar en hacer copias cuando parezca pertinente.

Como **ejercicio moral**, es bueno ver cómo son recorridos los elementos de L a lo largo de la implementación. Otro ejercicio importante es hacer la substitución para atrás necesaria para resolver el problema $U\mathbf{x} = \mathbf{y}$.

Se puede escribir el *pseudocódigo* para la misma substitución sólo que utilizando el acomodo por columnas en lugar de líneas. Ese primero resulta más eficiente en MATLAB, por ejemplo, que es orientado por columnas. ¿Cómo será el caso de PYTHON?

7.2. Matrices positivas definidas y descomposición de Cholesky

En el álgebra lineal es común aprovechar las estructuras de matrices específicas. Esta es todavía una práctica más común en el análisis numérico, mira por ejemplo el libro renombrado libro de Golub y van Loan, [13]. Un buen ejemplo son las matrices simétricas, es decir, aquellas que satisfacen $A = A^{\mathrm{T}}$, que podemos decir que son...

Discusión: ¿Qué características son evidentes en este caso? ¿Qué sabemos de A?

En este caso, podemos guardar la mitad de la información y si contamos con experiencia, las cuentas realizadas también deben de ser menores.

Hemos mencionado la factorización LU, aunque no la hayamos realizado aún, podemos notar algunas particularidades si A es simétrica. Por un lado, la factorización debe de poder ser del tipo $A = R^{\mathrm{T}}R$ para alguna matriz R triangular superior. Además, si A es regular y $\mathbf{x} \neq 0$, el vector $\mathbf{y} = R\mathbf{x} \neq 0$ y entonces

$$\mathbf{x}^{\mathrm{T}}A\mathbf{x} \;=\; \mathbf{x}^{\mathrm{T}}R^{\mathrm{T}}R\mathbf{x} \;=\; (R\mathbf{x})^{\mathrm{T}}R\mathbf{x} \;=\; \mathbf{y}^{\mathrm{T}}\mathbf{y} \;=\; \sum_{i=1}^{n} y_i^2 \;>\; 0.$$

Esta simple cuenta revela dos hechos importantes. Para una matriz regular A que puede ser fracturada por la matriz R debe satisfacer:

1.- Simétrica,

2.- $\mathbf{x} \neq 0 \implies \mathbf{x}^{\mathrm{T}} A\mathbf{x} > 0$,

que es la definición de una **matriz definida positiva**.

Un hecho sabido del álgebra lineal para construir matrices definidas positivas es utilizando una matriz rectangular $X \in \mathbb{R}^{n\times m}$ con columnas linealmente independientes y premultiplicarla por su transpuesta, es decir, $X^{\mathrm{T}}X$. (Este tipo de matrices son comunes y aparecen constantemente en la discretización de ecuaciones diferenciales parciales del tipo elípticas, es decir, aquellas que estudian los equilibrios de reacciones o temperaturas, entre otras.)

Algunas notas importantes sobre matrices positivas definidas es que son matrices no singulares, *i.e.*, satisfacen $A\mathbf{x} = 0$ si y sólo si $\mathbf{x} = 0$. Además de ello, son matrices definidas positivas por partes, es decir, si

$$A = \begin{pmatrix} \alpha & \mathbf{a}^{\mathrm{T}} \\ \mathbf{a} & A_* \end{pmatrix} \tag{7.2}$$

es una partición de A con $\alpha \in \mathbb{R}$, $\mathbf{a} \in \mathbb{R}^{n-1}$ y A_* de orden $n-1$, entonces necesariamente $\alpha > 0$ y A_* es definida positiva. A saber, para $\tilde{\mathbf{x}} = (1,\, 0,\, \cdots,\, 0)^{\mathrm{T}}$ y $\hat{\mathbf{x}}^{\mathrm{T}} = (0,\, \mathbf{y}^{\mathrm{T}})$ para $\mathbf{y} \in \mathbb{R}^{n-1}$ no nulo, tenemos

$$\begin{aligned} 0 < \tilde{\mathbf{x}}^{\mathrm{T}} A\tilde{\mathbf{x}} &= (1,\, \mathbf{0}^{\mathrm{T}}) \begin{pmatrix} \alpha & \mathbf{a}^{\mathrm{T}} \\ \mathbf{a} & A_* \end{pmatrix} \begin{pmatrix} 1 \\ \mathbf{0} \end{pmatrix} = \alpha, \\ 0 < \hat{\mathbf{x}}^{\mathrm{T}} A\hat{\mathbf{x}} &= (0,\, \mathbf{y}^{\mathrm{T}}) \begin{pmatrix} \alpha & \mathbf{a}^{\mathrm{T}} \\ \mathbf{a} & A_* \end{pmatrix} \begin{pmatrix} 0 \\ \mathbf{y} \end{pmatrix} = \mathbf{y}^{\mathrm{T}} A_* \mathbf{y}, \end{aligned}$$

donde el vector $\mathbf{0} = (0,\, 0,\, \cdots,\, 0) \in \mathbb{R}^{n-1}$.

Observación: Nota que la partición podría haber sido realizada de muchas maneras, siempre y cuando se conserve la diagonal principal en las submatrices. Esto conlleva a que una matriz definida positiva necesariamente satisface que $a_{ii} > 0$ para todos sus índices. Además puede probarse que si los determinantes de todas sus submatrices que compartan la diagonal principal son positivos, entonces la matriz es definida positiva. ¿Quieres intentar esta demostración?

Utilizaremos estas ideas para realizar la descomposición de Cholesky que podría resumirse a una descomposición LU especial para matrices simétricas y definidas positivas. Dado que esta matriz satisface $A = R^{\mathrm{T}}R$ y por lo tanto puede escribirse como (7.2), así tomamos la partición como:

$$A = \begin{pmatrix} \alpha & \mathbf{a}^{\mathrm{T}} \\ \mathbf{a} & A_* \end{pmatrix} = \begin{pmatrix} \rho & \mathbf{0}^{\mathrm{T}} \\ \mathbf{r} & R_*^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} \rho & \mathbf{r}^{\mathrm{T}} \\ \mathbf{0} & R_* \end{pmatrix},$$

que debe satisfacer por bloques:

$$\begin{aligned} \mathbf{1}.- \quad \alpha &= \rho^2, \\ \mathbf{2}.- \quad \mathbf{a}^{\mathrm{T}} &= \rho \mathbf{r}^{\mathrm{T}}, \\ \mathbf{3}.- \quad A_* &= R_*{}^{\mathrm{T}} R_* + \mathbf{r}\mathbf{r}^{\mathrm{T}}. \end{aligned}$$

Estas igualdades se despejan simplemente para encontrar

$$\begin{aligned} \mathbf{1}.- \quad \rho &= \sqrt{\alpha}, \\ \mathbf{2}.- \quad \mathbf{r}^{\mathrm{T}} &= \rho^{-1} \mathbf{a}^{\mathrm{T}}, \\ \mathbf{3}.- \quad R_*{}^{\mathrm{T}} R_* &= A_* - \mathbf{r}\mathbf{r}^{\mathrm{T}} =: \hat{A}_*. \end{aligned}$$

Notamos que las dos primeras definen, a partir de A_*, los valores de la primera línea de R. Además, α es mayor que cero y por lo tanto podemos tomar $\rho > 0$ que está bien definido. La tercera ecuación dice que R_* es la fractura de la matriz menor $\hat{A}_* := A_* - \mathbf{r}\mathbf{r}^{\mathrm{T}}$ si podemos mostrar que es simétrica y definida positiva. De la segunda ecuación, tomamos $\hat{A}_* := A_* - \alpha^{-1}\mathbf{a}\mathbf{a}^{\mathrm{T}}$ que es de un orden menor que A pero provienen todos sus elementos de ella. Veamos que satisface la misma estructura para seguir como paso iterativo.

Simetría.

Directamente tenemos:

$$\hat{A}_*{}^{\mathrm{T}} \;=\; (A_* - \mathbf{r}\mathbf{r}^{\mathrm{T}})^{\mathrm{T}} \;=\; A_*{}^{\mathrm{T}} - (\mathbf{r}\mathbf{r}^{\mathrm{T}})^{\mathrm{T}} \;=\; A_* - \mathbf{r}\mathbf{r}^{\mathrm{T}} \;=\; \hat{A}_*,$$

pues sabemos que A_* es simétrica.

Positividad.

Sabemos que A es positiva definida, entonces tomemos el vector $\mathbf{x}^{\mathrm{T}} = (\eta,\ \mathbf{y}^{\mathrm{T}})$ para cualquier escalar $\eta \in \mathbb{R}$ y $\mathbf{y} \in \mathbb{R}^{n-1}$, con lo cual tenemos:

$$0 \;<\; (\eta,\ \mathbf{y}^{\mathrm{T}}) \begin{pmatrix} \alpha & \mathbf{a}^{\mathrm{T}} \\ \mathbf{a} & A_* \end{pmatrix} \begin{pmatrix} \eta \\ \mathbf{y} \end{pmatrix} \;=\; \alpha\eta^2 + 2\mathbf{a}^{\mathrm{T}}\mathbf{y}\eta + \mathbf{y}^{\mathrm{T}} A_* \mathbf{y}.$$

Ahora consideramos $\eta = -\alpha^{-1}\mathbf{a}^{\mathrm{T}}\mathbf{y}$, y así

$$0 \;<\; \alpha^{-1}(\mathbf{a}^{\mathrm{T}}\mathbf{y})^2 - 2\mathbf{a}^{\mathrm{T}}\mathbf{y}(\alpha^{-1}\mathbf{a}^{\mathrm{T}}\mathbf{y}) + \mathbf{y}^{\mathrm{T}} A_* \mathbf{y} \;=\; \mathbf{y}^{\mathrm{T}} A_* \mathbf{y} - \mathbf{y}^{\mathrm{T}}(\alpha^{-1}\mathbf{a}\mathbf{a}^{\mathrm{T}})\mathbf{y} \;=\; \mathbf{y}^{\mathrm{T}} \hat{A}_* \mathbf{y}.$$

Mostrando lo que queríamos.

Ahora podemos calcular las entradas de $\hat{A}_*$:

$$\hat{\alpha}_{ij} \;=\; \alpha_{ij} - \alpha_{11}^{-1}\alpha_{1i}\alpha_{1j} \;=\; \alpha_{ij} - \alpha_{11}^{-1}\alpha_{i1}\alpha_{1j},$$

donde $A = (\alpha_{ij})$ que además de ser simétrica define los vectores $\mathbf{a} = (\alpha_{12}, \alpha_{13}, \ldots, \alpha_{1n})^{\mathrm{T}}$.

Así, por ejemplo, para un sistema con $n = 4$ tenemos:

$$\begin{aligned}
\alpha_{11}x_1 + \alpha_{12}x_2 + \alpha_{13}x_3 + \alpha_{14}x_4 &= b_1,\\
\alpha_{21}x_1 + \alpha_{22}x_2 + \alpha_{23}x_3 + \alpha_{24}x_4 &= b_2,\\
\alpha_{31}x_1 + \alpha_{32}x_2 + \alpha_{33}x_3 + \alpha_{34}x_4 &= b_3,\\
\alpha_{41}x_1 + \alpha_{42}x_2 + \alpha_{43}x_3 + \alpha_{44}x_4 &= b_4,
\end{aligned}$$

que nos lleva de la primera ecuación a $x_1 = \alpha_{11}^{-1}(b_1 - \alpha_{12}x_2 - \alpha_{13}x_3 - \alpha_{14}x_4)$, que al substituir y simplificar en las otras tres, nos lleva a:

$$\begin{aligned}
(\alpha_{22} - \alpha_{11}^{-1}\alpha_{21}\alpha_{12})x_2 + (\alpha_{23} - \alpha_{11}^{-1}\alpha_{21}\alpha_{13})x_3 + (\alpha_{24} - \alpha_{11}^{-1}\alpha_{21}\alpha_{14})x_4 &= b_2 - \alpha_{11}^{-1}\alpha_{21}b_1,\\
(\alpha_{32} - \alpha_{11}^{-1}\alpha_{31}\alpha_{12})x_2 + (\alpha_{33} - \alpha_{11}^{-1}\alpha_{31}\alpha_{13})x_3 + (\alpha_{34} - \alpha_{11}^{-1}\alpha_{31}\alpha_{14})x_4 &= b_3 - \alpha_{11}^{-1}\alpha_{31}b_1,\\
(\alpha_{42} - \alpha_{11}^{-1}\alpha_{41}\alpha_{12})x_2 + (\alpha_{43} - \alpha_{11}^{-1}\alpha_{41}\alpha_{13})x_3 + (\alpha_{44} - \alpha_{11}^{-1}\alpha_{41}\alpha_{14})x_4 &= b_4 - \alpha_{11}^{-1}\alpha_{41}b_1,
\end{aligned}$$

que no es otra cosa sino la *eliminación gaussiana* de una matriz de orden 4 a una de orden 3. Al notar los coeficientes entre paréntesis, los reconocemos del algoritmo anterior de Cholesky y, por tanto, vemos que la eliminación gaussiana nos da la misma submatriz.

Antes de escribir un código para la descomposición de Cholesky, remarcamos dos consideraciones: (1) las matrices A y $\hat{A}_*$ son simétricas y no necesitamos guardar todos sus elementos pues se repiten y (2) como de α y $\mathbf{a}$ computamos ρ y $\mathbf{r}$, aquellos no son más necesarios y por lo tanto podemos reescribir los nuevos elementos de R en la antigua A. (Aunque hay que recordar que es una práctica antigua y no es indispensable.)

La idea es que en cada paso del algoritmo la matriz sea una matriz triangular superior, inclusive aunque parte de ella provenga de la matriz simétrica A, es decir,

$$\begin{pmatrix}
\rho_{11} & \rho_{12} & \rho_{13} & \rho_{14} & \rho_{15}\\
0 & \rho_{22} & \rho_{23} & \rho_{24} & \rho_{25}\\
0 & 0 & \alpha_{33} & \alpha_{34} & \alpha_{35}\\
0 & 0 & 0 & \alpha_{44} & \alpha_{45}\\
0 & 0 & 0 & 0 & \alpha_{55}
\end{pmatrix}
\quad\mapsto\quad
\begin{pmatrix}
\rho_{11} & \rho_{12} & \rho_{13} & \rho_{14} & \rho_{15}\\
0 & \rho_{22} & \rho_{23} & \rho_{24} & \rho_{25}\\
0 & 0 & \rho_{33} & \rho_{34} & \rho_{35}\\
0 & 0 & 0 & \hat{\alpha}_{44} & \hat{\alpha}_{45}\\
0 & 0 & 0 & 0 & \hat{\alpha}_{55}
\end{pmatrix}$$

y en el k-ésimo paso el renglón k es modificado desde los valores de $\hat{A}_*$ para colocarlos en R. Observamos que también se cambian las líneas para $i > k$.

Código: Algoritmo de Cholesky (PYTHON)

```
for k in range(n):
    A[k, k] = sqrt( A[k, k] )
    for j in range(k + 1, n):
        A[k, j] = A[k, j]/A[k, k]                    # Linea 10
    for j in range(k + 1, n):
        for i in range(k + 1, n):
            A[i, j] = A[i, j] - A[k, i]*A[k, j]      # Linea 20
```

Discusión: Observa que hay dos ciclos con `j in range(k + 1, n)`, piensa el porqué.

Observa que el código está organizado a modo que la orientación de la matrices sea dado por columnas, ¿justo como le gusta a PYTHON? Sin embargo, "hay una ofensa" en la `Linea 20` al pedir el elemento `A[k, i]`; una solución es introducir un nuevo arreglo que lleve los valores de **r**. Por ejemplo, después de la `Linea 10` colocamos

```
r[j] = A[k, j]
```

y en lugar de la `Linea 20` escribimos:

```
A[i, j] = A[i, j] - r[i]*r[j]
```

Tal vez quieras escribir los dos códigos y descubrir cuál es más rápido. Otra opción es tratar de construir el algoritmo por renglones o tomar $L = R^{\mathrm{T}}$ y aprovechar la estructura arriba.

Ejemplo: Ahora vamos a calcular el número de operaciones del algoritmo de Cholesky, bueno, solamente su orden. Observamos que hay tres ciclos "anidados", estos son en las variables i, j, k. Según hemos visto:

$$\begin{aligned}
\text{Ops(Cholesky)} &= \sum_{k=1}^{n}\left[1+\sum_{j=k+1}^{n}1+\sum_{j=k+1}^{n}\sum_{i=k+1}^{n}2\right]\\
&= \sum_{k=1}^{n}\left[1+\sum_{j=k+1}^{n}\left(1+\sum_{i=k+1}^{n}2\right)\right]\\
&\approx \int_{1}^{n}1+\int_{k+1}^{n}1+\int_{k+1}^{n}2\,\mathrm{d}i\,\mathrm{d}j\,\mathrm{d}k\\
&\approx \int_{0}^{n}0+\int_{k+0}^{n}0+\int_{k+0}^{n}1\,\mathrm{d}i\,\mathrm{d}j\,\mathrm{d}k\\
&= \int_{0}^{n}\int_{k}^{n}\int_{k}^{n}1\,\mathrm{d}i\,\mathrm{d}j\,\mathrm{d}k \;=\; \frac{1}{3}n^{3},
\end{aligned}$$

es decir, el algoritmo tiene una *complejidad* $\mathcal{O}(n^3)$.

Se deja nuevamente como **tarea moral** revisar que el número de operaciones exactas del algoritmo de Cholesky es exactamente un polinomio cúbico en n.

Por otro lado, hemos visto que resolver un sistema $R^{\mathrm{T}}\mathbf{y} = \mathbf{b}$ y $R\mathbf{x} = \mathbf{y}$ tiene en cada caso una complejidad $\mathcal{O}(n^2)$. Así, resolver un sistema $A\mathbf{x} = \mathbf{b}$ para una matriz simétrica positiva definida A nos llevará un orden de n^3 operaciones.

Discusión: Consideremos que queremos encontrar la inversa de una matriz A simétrica y definida positiva. Usando Cholesky podemos resolver n sistemas de la forma $A\mathbf{x}_i = \hat{\mathbf{e}}_i$, donde $\hat{\mathbf{e}}_i$ es el i-ésimo vector canónico en $\mathbb{R}^n$.

1. ¿Cuántas operaciones debemos realizar?

2. ¿En general, parece una buena idea encontrar la inversa de una matriz?

Este es un buen ejemplo de los pocos casos en dónde tener el número exacto de operaciones y no solamente el orden de ellas puede ser importante. Por un lado, las operaciones a realizar son $\mathcal{O}(n^3)$ para encontrar desde A la matriz R, después utilizamos n veces dos métodos de complejidad $\mathcal{O}(n^2)$ para resolver $R^{\mathrm{T}}\mathbf{y} = \hat{\mathbf{e}}_i$ y $R\mathbf{x}_i = \mathbf{y}$, así $A^{-1} = [\mathbf{x}_1 \ \mathbf{x}_2 \ \cdots \ \mathbf{x}_n]$ pues $AA^{-1} = I$, para ello, hemos utilizado $\mathcal{O}(n^3) + 2\mathcal{O}(n^2) = \mathcal{O}(n^3)$ operaciones.

Es decir, podemos encontrar la inversa en este caso. Resolver $A\mathbf{x} = \mathbf{b}$ podría hacerse con la inversa usando $\mathcal{O}(n^3)$ que es lo mismo que encontrar R y resolver $R^{\mathrm{T}}\mathbf{y} = \mathbf{b}$ y $R\mathbf{x} = \mathbf{y}$. Sin embargo, esto depende realmente de los coeficientes que aparecen en los órdenes cúbico y cuadrático arriba. Si los consideramos y $\mathcal{O}(n^3) \approx an^3$ y $\mathcal{O}(n^2) \approx bn^2$, tenemos:

i) por Cholesky, $\mathcal{O}(n^3) + 2\mathcal{O}(n^2) \approx an^3 + bn^2$,

ii) con la inversa, $\mathcal{O}(n^3) + 2n\mathcal{O}(n^2) \approx (a + 2b)n^3$,

que puede cambiar radicalmente dada la comparación entre a y b.

Finalmente, como observación, la factorización que hemos mencionado aquí es el algoritmo de Cholesky basado en el "producto exterior" $\mathbf{a}\mathbf{a}^{\mathrm{T}}$. Existe una otra forma de hacerlo con base en el "producto interior" $\mathbf{a}^{\mathrm{T}}\mathbf{a}$. Alguien interesado, puede buscar en el libro de Stewart, [14].

Ejercicio 30: Calcular el número de operaciones exactas o el orden de las operaciones realizadas por un algoritmo es muy importante. Calcula de manera exacta y de modo aproximado:

1. El número exacto y el orden de las operaciones de la substitución hacia adelante.
2. Considera una matriz simétrica y definida positiva tal que solo la diagonal principal y las dos adyacentes son no nulas. Calcula el número exacto y el orden de las operaciones.

7.3. Eliminación gaussiana y la factorización LU

El algoritmo de Cholesky además de eficiente, muestra varias ideas que pueden ser aprovechadas en un algoritmo más general. Además, nos ha mostrado que ni siquiera en casos tan específicos, la

búsqueda de la inversa de una matriz parece ser un camino adecuado.

Recodemos la eliminación gaussiana para una matriz A de orden cuatro y así colocar un poco de la notación en día. Tenemos el sistema:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1,\\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= b_2,\\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= b_3,\\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= b_4, \end{aligned}$$

que con $m_{i1} := a_{i1}/a_{11}$ para $i = 2, 3, 4$ nos ayuda a simplificar el sistema al manipular el renglón r_i y transformarlo en el renglón $r_i - m_{i1}r_1$, es decir, simplificamos en

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1,\\ a'_{22}x_2 + a'_{23}x_3 + a'_{24}x_4 &= b'_2,\\ a'_{32}x_2 + a'_{33}x_3 + a'_{34}x_4 &= b'_3,\\ a'_{42}x_2 + a'_{43}x_3 + a'_{44}x_4 &= b'_4, \end{aligned}$$

donde $a'_{ij} = a_{ij} - m_{i1}a_{1j}$ y $b'_i = b_i - m_{i1}b_1$. Podemos seguir del mismo modo con $a''_{ij} = a'_{ij} - m_{i2}a'_{2j}$ y $b''_i = b'_i - m_{i2}b_2$, donde $m_{i2} := a'_{i2}/a'_{22}$ para $i, j = 3, 4$, simplificando así en

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1,\\ a'_{22}x_2 + a'_{23}x_3 + a'_{24}x_4 &= b'_2,\\ a''_{33}x_3 + a''_{34}x_4 &= b''_3,\\ a''_{43}x_3 + a''_{44}x_4 &= b''_4. \end{aligned} \tag{7.3}$$

Finalmente, para la última línea con $m_{43} = a''_{43}/a''_{33}$ tenemos simplemente $a'''_{44}x_4 = b'''_4$ donde $a'''_{44} = a''_{44} - m_{34}a''_{34}$ y $b'''_4 = b''_4 - m_{34}b''_3$.

Vemos que a cada paso, con la creación y uso de todos los m_{ik} para un k fijo, obtenemos una nueva matriz donde los elementos bajo a_{kk} son nulos. Considereamos $A_1 = A$ y en general A_k como la matriz que tiene la columna k completa y es "triangular superior" para las columnas previas. Así, tomando M_k como aquella matriz cuyas entradas satisfacen

$$\ell_{ij} = \begin{cases} 1, & \text{si } i = j,\\ m_{ik}, & \text{si } j = k \text{ y } i > j,\\ 0, & \text{en otro caso,} \end{cases}$$

tenemos que $M_kA_k = A_{k+1}$. De este modo, es claro que este algoritmo, llamado *eliminación gaus-*

siana, puede ser extendido de modo natural para matrices de orden n, al usar

$$m_{ik} \;=\; a_{ik}^{(k-1)}/a_{kk}^{(k-1)}, \qquad \text{para} \qquad i \;=\; k+1,\, k+2,\, \ldots,\, n,$$

donde (k) significa el número de apóstrofes de un elemento y $a_{ij}^{(0)}$ es el elemento inicial a_{ij}. El algoritmo funciona de modo recursivo al tomar:

$$\begin{array}{rcl} a_{ij}^{(k)} & = & a_{ij}^{(k-1)} - m_{ik}a_{kj}^{(k-1)}, \\ b_{i}^{(k)} & = & b_{i}^{(k-1)} - m_{ik}b_{k}^{(k-1)}. \end{array} \qquad \text{para} \qquad i,\, j \;=\; k+1,\, k+2,\, \ldots,\, n, \tag{7.4}$$

Observamos que los índices ij son los extremos i a la izquierda y j a la derecha, el índice intermedio k sólo sirve como el elemento de la suma.[1]

Nuestro objetivo es factorizar la matriz A como el producto LU, donde la matriz L es triangular inferior (*lower*, en inglés) y U es una matriz triangular superior (*upper*, en inglés). El algoritmo que hemos seguido nos determina de modo directo la matriz U, mira el sistema (7.3). Para la matriz L, basta observar que el procedimiento que nos llevó de A para U.

La secuencia de operaciones

$$A \;=\; A_1 \;=\; M_1^{-1}A_2 \;=\; M_1^{-1}M_2^{-1}A_3 \;=\; \cdots \;=\; M_1^{-1}M_2^{-1}\cdots M_{n-1}^{-1}A_n \;=\; LU,$$

se satisface. Observamos que M_k^{-1}, para $k = 1,\, 2,\, \ldots,\, n-1$, es una matriz triangular inferior cuyas entradas ℓ_{ij} (de la matriz inversa) satisfacen:

$$\ell_{ij} \;=\; \begin{cases} 1, & \text{si } i = j, \\ -m_{ik}, & \text{si } j = k \text{ y } i > j, \\ 0, & \text{en otro caso.} \end{cases}$$

Además, la matriz A_k, con $k = 1,\, 2,\, \ldots,\, n$, satisface que sus entradas a_{ij} son nulas si $j < k$ e $i > j$. En los renglones restantes, sus otras entradas son $a_{ij}^{(\ell)}$ con $\ell = i-1$ para $i < k$ y $\ell = k-1$. Así, $L = M_1^{-1}M_2^{-1}\cdots M_{n-1}^{-1}$ es una triangular inferior y $U = A_n$ una matriz triangular superior.

El algoritmo nos entrega de este modo las matrices de la forma

$$L \;=\; \begin{pmatrix} 1 & 0 & \cdots & 0 \\ m_{21} & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ m_{n1} & m_{n2} & & 1 \end{pmatrix} \qquad \text{y} \qquad U \;=\; \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & & a_{nn}^{(n-1)} \end{pmatrix}.$$

[1] Esto está curiosamente en la misma filosofía que las sumas de Einstein que simplifican la notación en la Teoría de la Relatividad, en especial cuando se opera con tensores.

De este modo, un código que aproveche la estructura de la descomposición puede utilizar la misma matriz de entrada para guardar los resultados de L y U. Parece que tenemos un problema con la duplicación de elementos sobre la diagonal, pero observando que $L-I$ tiene diagonal nula, podemos guardar la salida como $(L-I)+U$.

Ejercicio 31: Muestra que la matriz L descrita arriba realmente proviene del producto $M_1^{-1}M_2^{-1}\cdots M_{n-1}^{-1}$. Muestra también que $A_k = M_k^{-1}A_{k+1}$ para $k = 1, 2, \ldots, n-1$. Concluye entonces que $A = LU$ se satisface y por lo tanto tenemos la factorización LU.

Código: Factorización LU (Python)

```
for k in range(n):
    for i in range(k + 1, n):
        A[i, k] = A[i, k]/A[k, k]
    for j in range(k + 1, n):
        for i in range(k + 1, n):
            A[i, j] = A[i, j] - A[i, k]*A[k, j]
```

7.3.1. Pivoteo parcial

Matemáticamente parece muy sencillo realizar la factorización LU a partir de la eliminación Gaussiana, sin embargo, hay que poner atención en algunas situaciones. Hay matrices regulares que no tienen factorización LU. Un ejemplo muy sencillo es la matriz

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Por otro lado, un caso de matriz singular con descomposición LU es

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix},$$

donde L es la matriz identidad, por ejemplo, y U sigue siendo la misma matriz triangular superior. Esto ya es un indicio de que no es trivial hacer la descomposición LU de modo automático y menos a ciegas.

Ejercicio 32: Considera de modo numérico, la matriz

$$M = \begin{bmatrix} 0.1 & & & & 1 \\ -1 & 0.1 & & & \\ & -1 & 0.1 & & \\ & & -1 & 0.1 & \\ & & & -1 & 1 \end{bmatrix}$$

con cuatro dígitos de precisión. Realiza su factorización LU y observa el resultado del producto $LU = A$ al compararlo con M.

Observación: A lo largo de este texto, las matrices entre paréntesis son matrices con elementos reales, las matrices entre corchetes son matrices en algún tipo de aritmética de punto flotante; a menos que se especifique, se considera siempre la precisión doble (*DP*, de sus siglas en inglés).

Veamos el ejemplo dado por Stewart (véase [14]) para una aritmética de cuatro dígitos. Consideramos la matriz de orden 3 siguiente.

$$\begin{aligned} A = \mathtt{A}_1 &= \begin{bmatrix} 0.001 & 2.000 & 3.000 \\ -1.000 & 3.712 & 4.623 \\ -2.000 & 1.072 & 5.643 \end{bmatrix} \\ &= \mathtt{M}_1^{-1}\mathtt{A}_2 = \begin{bmatrix} 1.000 & 0.000 & 0.000 \\ -1000 & 1.000 & 0.000 \\ -2000 & 0.000 & 1.000 \end{bmatrix} \begin{bmatrix} 0.001 & 2.000 & 3.000 \\ 0.000 & 2004 & 3005 \\ 0.000 & 4001 & 6006 \end{bmatrix} \\ &= \mathtt{M}_1^{-1}\mathtt{M}_2^{-1}\mathtt{A}_3 = \begin{bmatrix} 1.000 & 0.000 & 0.000 \\ -1000 & 1.000 & 0.000 \\ -2000 & 1.997 & 1.000 \end{bmatrix} \begin{bmatrix} 0.001 & 2.000 & 3.000 \\ 0.000 & 2004 & 3005 \\ 0.000 & 0.000 & 5.000 \end{bmatrix}. \end{aligned}$$

Así, para $U = \mathtt{A}_3$, tenemos que el elemento $u_{33} = 5.000$ debería de ser realmente $5.922\cdots$, lo cual como vemos es un error relativo grande:

$$\rho = \frac{5.922 - 5}{5.992} = 0.1655\cdots,$$

es decir, un error casi del 17 %. De hecho, la aproximación final de $L = \mathtt{M}_1^{-1}\mathtt{M}_2^{-1}$ y U es como si hubiéramos realizado la descomposición LU para la matriz

$$\mathtt{A} = \begin{bmatrix} 0.001 & 2.000 & 3.000 \\ -1.000 & 4.000 & 5.000 \\ -2.000 & 1.000 & 6.000 \end{bmatrix}.$$

Esto no es una mera cuestión estética, si queremos resolver el problema $A\mathbf{x} = \mathbf{b}$ con $\mathbf{b} \in \mathbb{R}^3$, no es lo mismo la solución con la A original que con la $\mathtt{A}$ numérica. Para tratar de solucionar estos problemas, vamos a buscar dónde fue que realizamos los pasos con el mayor error.

Esto ocurre en el primer paso cuando calculamos $\mathtt{A_2}$, pues los multiplicadores m_{i1} son "grandes" y perdemos así varias cifras significativas; esto es lo que produce que el vector $\mathbf{x}$ sea distinto del vector numérico $\mathtt{x}$, donde $\mathtt{Ax} = \mathbf{b}$.

La manera de minimizar estos errores es tomando en cada paso el *pivote* de cada columna como el elemento para realizar las cuentas. Para el j-ésimo paso cuando estamos anulando elementos de la columna j, definimos el **pivote** como el elemento activo de mayor magnitud, es decir, a_{kj} es este pivote si

$$|a_{kj}| \;=\; \max_{i=j,\, j+1,\, \cdots,\, n} |a_{ij}|.$$

De este modo, premultiplicamos por una matriz de permutaciones que intercambie los renglones j y k. La matriz $P_j(k)$ tiene todos sus elementos nulos, con excepción de los elementos $p_{ii} = 1$ si $i \neq j$, k y $p_{kj} = p_{jk} = 1$ que fungen como la permutación. Dejaremos implícito el valor k en la matriz P_j. A este intercambio de renglones le llamaremos **pivoteo parcial**.

Si en diferentes pasos realizamos un permutación del tipo del pivoteo parcial, entonces, tendremos que la matriz triangular superior proviene de los pasos concatenados por

$$U \;=\; M_{n-1}P_{n-1}M_{n-2}\cdots M_2P_2M_1P_1A. \tag{7.5}$$

Lo importante de esta forma es que las matrices de permutación y de multiplicadores admiten la conmutabilidad si el índice de la permutación es mayor que la del múltiplo.

Ejercicio 33: Considerando la descomposición en (7.5), muestra lo siguiente.

1. Las matrices satisfacen la conmutabilidad $P_kM_l = M_lP_k$ cuando $k > l$.
   Recuerda que P_k permuta el renglón k con uno que tiene un índice mayor.
2. Muestra entonces que $U = L^{-1}PA$, para P una matriz de permutación,
   escribe su forma.

El ejercicio anterior muestra que podemos factorizar A al premultiplicarla por una permutación como $LU = PA$. Esto ayuda a resolver el problema $A\mathbf{x} = \mathbf{b}$, pues

$$PA\mathbf{x} \;=\; P\mathbf{b} \quad\Longrightarrow\quad LU\mathbf{x} \;=\; \tilde{\mathbf{b}} \quad\Longrightarrow\quad \begin{cases} L\mathbf{y} \;=\; \tilde{\mathbf{b}} \\ U\mathbf{x} \;=\; \mathbf{y} \end{cases}.$$

Observemos que no es necesario guardar el valor de la matriz de permutación entera; es *esparsa* o *rala*. Basta guardar el vector de las permutaciones y aplicarlo directamente a $\mathbf{b}$ para obtener $\tilde{\mathbf{b}}$. Además, ese resultado muestra que si aplicamos la descomposición LU directamente a la matriz

PA, entonces, tanto L como U serán como descritas arriba, es decir,

$$\begin{aligned} P &= P_{n-1}P_{n-2}\cdots P_2P_1, \\ U &= L^{-1}PA, \\ L &= M_1^{-1}M_2^{-1}\cdots M_{n-1}^{-1}, \end{aligned}$$

recordando que P puede ser dado como $\mathbf{p}$.

Código: Factorización LUP (Python)

```
for k in range(n - 1):
    maxa = abs( A[k, k] )
    p[k] = k
    for i in range(k + 1, n):                          # Linea 10
        if ( abs( A[i, k] ) > maxa ):
            maxa = abs( A[i, k] )
            p[k] = i
    if ( p[k] != k ):
        for j in range(n):                             # Linea 20
            temp       = A[k, j]
            A[k,  j]   = A[p[k], j]
            A[p[k], j] = temp
    for i in range(k + 1, n):                          # Linea 30
        A[i, k] = A[i, k]/A[k, k]
    for j in range(k + 1, n):
        for i in range(k + 1, n):
            A[i, j] = A[i, j] - A[i, k]*A[k, j]
```

Ejemplo: Notamos que en cada intercambio de renglones para un vector, se realizan 3 operaciones, pues hay que guardar un valor en algún lugar temporal. Sin embargo, realizar el producto de $P\mathbf{b}$ requiere n^2 productos y $n(n-1)$ sumas, es decir, $2n^2-n$ operaciones en lugar de las $3(n-1)$ por los intercambios directos.

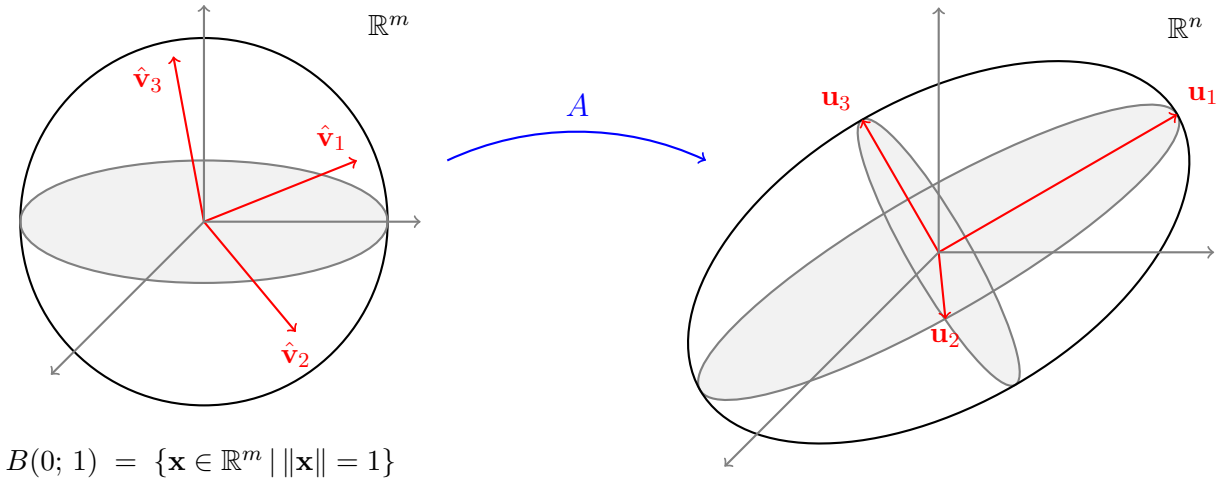
Observando este código contra la Factorización LU sin pivoteo, vemos que solamente el texto introducido antes de la `Linea 30` es nuevo, el resto es igual. Es decir, en este caso reutilizamos A al entregar la salida $(L-I)+U$, además, se entrega el vector $\mathbf{p}$ de permutaciones.

Ejercicio 34: Explica cómo debe ser el algoritmo para encontrar $\tilde{\mathbf{b}}$ a partir de $\mathbf{b}$ y $\mathbf{p}$. Observa y explica qué es lo que sucede en cada uno de los ciclos de las `Linea 10` y `Linea 20`.

7.4. Descomposición en Valores Singulares (SVD)

Hasta ahora nos hemos preocupado con problemas de la forma $A\mathbf{x} = \mathbf{b}$, en los cuales hemos tratado la información de la matriz A como meramente la transformación de un vector $\mathbf{x}$ en el objetivo $\mathbf{b}$. No debemos de perder de vista que las matrices tienen muchas aplicaciones y significados, entre otras, pueden guardar grandes cantidades de información de modo ordenado.

Si consideramos de modo general una matriz $A \in \mathbb{R}^{n\times m}$, podemos entender la información guardada en ella a través de su significado geométrico como una transformación lineal entre espacios vectoriales, digamos $T : \mathbf{x} \in \mathbb{R}^m \to \mathbb{R}^n$ tal que $T(\mathbf{x}) = A\mathbf{x}$. De este modo, la siguiente imagen nos ayuda a entender algunos de estos conceptos.



$$B(0;\,1) \;=\; \{\mathbf{x} \in \mathbb{R}^m \,|\, \|\mathbf{x}\| = 1\}$$

Dado que la transformación es lineal, basta entender qué sucede con la bola de radio uno y centada en el origen, $B(0;\,1)$; ésta es mapeada en un elipsoide en el espacio imagen. ¿Siempre? Aquí entran ideas importantes: (1) si $n = m$ y la matriz es de rango completo, sí, (2) si $m > n$ y el rango de A es n, también, pero (3) si $m < n$, no importa el rango k de la matriz, este satisface $k \leq m$ y es la dimensión del subespacio imagen, en este caso, no tenemos un elipsoide lleno, sólo uno de estos en una dimensión menor a la del espacio. El rango k de la matriz dicta así la dimensión de la imagen de todo el espacio $\mathbb{R}^m$ en $\mathbb{R}^n$.

Sin embargo, todo elipsoide (aún el de dimensión $k < n$ en $\mathbb{R}^n$) puede ser definido a través de vectores ortogonales que forman sus semiejes, digamos los $\mathbf{u}_1$, $\mathbf{u}_2$, $\ldots$, $\mathbf{u}_k$ en la imagen. Recordando el Teorema Espectral para matrices $A \in \mathbb{R}^{n\times n}$, donde existe una matriz Q ortonormal y una matriz D diagonal tales que $A = QDQ^\top$, nos gustaría pensar que aquí sucede algo semejante. Por ejemplo, ¿qué tal si las preimágenes $\hat{\mathbf{v}}_1$, $\hat{\mathbf{v}}_2$, $\ldots$, $\hat{\mathbf{v}}_k \in \mathbb{R}^m$, tales que $A\hat{\mathbf{v}}_i = \mathbf{u}_i$, fueran ortogonales? De hecho, ortonormales dado que están en $B(0;\,1)$ y es por ello que hemos colocado el "sombrero" para recordarlo. Para esto, tenemos el siguiente resultado que no mostraremos, pues mostraremos una versión más analítica.

Teorema 7.1 (SVD geométrica). *Sea $A \in \mathbb{R}^{n\times m}$, entonces siempre existen dos bases ortonormales de $\mathbb{R}^m$ y $\mathbb{R}^n$, a saber,*

$$\mathcal{V} = \{\hat{\mathbf{v}}_1, \hat{\mathbf{v}}_2, \ldots, \hat{\mathbf{v}}_m\} \subset \mathbb{R}^m, \qquad y \qquad \mathcal{U} = \{\hat{\mathbf{u}}_1, \hat{\mathbf{u}}_2, \ldots, \hat{\mathbf{u}}_n\} \subset \mathbb{R}^n,$$

respectivamente, además existen $k = \min\{n, m\}$ números reales ordenados como $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k \geq 0$ y tales que

$$A\hat{\mathbf{v}}_i = \sigma_i \hat{\mathbf{u}}_i, \qquad \text{para todo} \qquad i = 1, 2, \ldots, k. \tag{7.6}$$

De este resultado viene la *descomposición en valores singulares* o SVD de sus siglas en inglés. Los vectores $\hat{\mathbf{u}}_i$ se llaman los **vectores singulares a la izquierda**, los números σ_i son los **valores singulares** y los vectores $\hat{\mathbf{v}}_i$ son los **vectores singulares a la derecha**. De las igualdades en (7.6) y al definir las matrices

$$\bar{V} = [\hat{\mathbf{v}}_1\ \hat{\mathbf{v}}_2\ \cdots\ \hat{\mathbf{v}}_m] \in \mathbb{R}^{m\times m}, \qquad \bar{U} = [\hat{\mathbf{u}}_1\ \hat{\mathbf{u}}_2\ \cdots\ \hat{\mathbf{u}}_n] \in \mathbb{R}^{n\times n},$$
$$\text{y} \qquad \bar{\Sigma} = \operatorname{diag}(\sigma_1, \sigma_2\ \ldots, \sigma_k) \in \mathbb{R}^{n\times m},$$

tenemos que $A\bar{V} = \bar{U}\bar{\Sigma}$ se satisface. Recordamos que la inversa de una matriz ortonormal (típicamente *ortogonal*) es su transpuesta, es decir, $\bar{U}\bar{U}^\top = I_n \in \mathbb{R}^{n\times n}$ y $\bar{V}\bar{V}^\top = I_m \in \mathbb{R}^{m\times m}$, para I_k la identidad de orden k. De este modo se tiene que $A = \bar{U}\bar{\Sigma}\bar{V}^\top$.

Ejemplo: Sin entrar en detalles de cómo obtener las matrices descritas arriba, vemos que

$$\begin{pmatrix} 0 & 0 & 2 \\ -4 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 4 & 0 & 0 \\ 0 & -2 & 0 \end{pmatrix} \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix},$$

donde $A \in \mathbb{R}^{2\times 3}$, $\bar{U} \in \mathbb{R}^{2\times 2}$, $\bar{\Sigma} \in \mathbb{R}^{2\times 3}$ y $\bar{V} \in \mathbb{R}^{3\times 3}$. Esta es la **factorización en valores singulares** y notamos que los términos en rojo no contribuyen a la formación de A, removiendo esta columna y esta línea, tenemos que

$$\begin{pmatrix} 0 & 0 & 2 \\ -4 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 4 & 0 \\ 0 & -2 \end{pmatrix} \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

donde ahora podemos definir la matriz $\Sigma \in \mathbb{R}^{k\times k}$ como la parte cuadrada de $\bar{\Sigma}$ (en este caso la matriz diagonal de orden 2). Esto también redefine los tamaños de las otras matrices, al tomar $U \in \mathbb{R}^{2\times 2}$ igual que la $\bar{U}$ anterior y $V \in \mathbb{R}^{2\times 3}$ como una submatriz de $\bar{V}$. En ambos casos estamos tomando las primeras k columnas de sus versiones con barra. Esta definición más compacta es la **descomposición en valores singulares** con consecuencias importantes.

Recordando al inicio de este capítulo el tratamiento sobre matrices de rango uno, veremos que cuando el rango k de A es inferior a n y m, podemos economizar información en el interior de la máquina, por otro lado, también podemos forzar a tener una compresión de ésta perdiendo poca resolución.

Teorema 7.2 (Descomposición en Valores Singulares)**.** *Sea $A \in \mathbb{R}^{n\times m}$, entonces siempre existen dos bases ortonormales $\mathcal{V} \subset \mathbb{R}^m$ y $\mathcal{U} \subset \mathbb{R}^n$ y números reales $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k \geq 0$ para $k = \min\{n, m\}$ tales que $A\bar{V} = \bar{U}\bar{\Sigma}$ se satisface con $\bar{U}$ formada por la base de $\mathcal{U}$ y $\bar{V}$ formada por la base de $\mathcal{V}$, $\bar{\Sigma}$ es una matriz de la forma de A con los valores singulares en la diagonal, donde*

1) *Los vectores singulares a la izquierda son los eigenvectores de $AA^\top$.*
2) *Los vectores singulares a la derecha son los eigenvectores de $A^\top A$.*
3) *Los cuadrados de los valores singulares son los eigenvalores de $AA^\top$ y de $A^\top A$*

Para demostrar este teorema tendremos que hacer uso del Teorema Espectral para matrices simétricas. Notamos que tanto $A^\top A$ como $AA^\top$ son matrices reales y simétricas. Por tanto, existen los eigenpares $(\lambda_i, \hat{\mathbf{v}}_i)$ para $i \in \{1, 2, \ldots, m\}$ tales que $A^\top A\hat{\mathbf{v}}_i = \lambda_i\hat{\mathbf{v}}_i$, así como los eigenpares $(\mu_i, \hat{\mathbf{u}}_i)$ para $i \in \{1, 2, \ldots, n\}$ tales que $AA^\top\hat{\mathbf{u}}_i = \mu_i\hat{\mathbf{u}}_i$. Por un lado, tenemos las bases ortonormales $\mathcal{V}$ y $\mathcal{U}$, respectivamente, sin embargo, es difícil ver que $\lambda_i = \mu_i = \sigma_i^2$ se satisfacen. En cambio, podemos mostrar que se satisface la igualdad entre matrices que queremos.

Primero notemos que tenemos que descubrir que los eigenvalores son positivos y que por lo tanto los valores singulares pueden ser tomados como las raíces positivas de estos. En segundo lugar, basta mostrar que para estos valores, la igualdad (7.6) se satisface. Consideremos los eigenpares $(\lambda_i, \hat{\mathbf{v}}_i)$, ahora notemos que

$$0 \leq \|A\hat{\mathbf{v}}_i\|^2 = (A\hat{\mathbf{v}}_i)^\top A\hat{\mathbf{v}}_i = \hat{\mathbf{v}}_i^\top A^\top A\hat{\mathbf{v}}_i = \hat{\mathbf{v}}_i^\top\lambda_i\hat{\mathbf{v}}_i = \lambda_i\|\hat{\mathbf{v}}_i^2\|^2 = \lambda_i,$$

y por tanto los eigenvalores son no negativos. Después de ordenarlos, tomamos $\sigma_i = \sqrt{\lambda_i}$. Por conveniencia, si $k = m$, entonces definimos $\lambda_i = 0$ para toda $i > k$, es decir, σ_i en estos casos también es nulo. A partir de $(\lambda_i, \hat{\mathbf{v}}_i)$ construiremos la base $\hat{\mathbf{u}}_i$, de la siguiente forma:

$$\begin{aligned} \text{Si} \quad \lambda_i \neq 0 \quad &\Longrightarrow \quad \hat{\mathbf{u}}_i = A\hat{\mathbf{v}}_i/\sigma_i \\ \lambda_i = 0 \quad &\Longrightarrow \quad \hat{\mathbf{u}}_i \perp \operatorname{span}\{\hat{\mathbf{u}}_1, \hat{\mathbf{u}}_2, \ldots, \hat{\mathbf{u}}_{i-1}\}. \end{aligned}$$

Es claro que la segunda condición garantiza la ortonormalidad, es por ello que sólo debemos de preocuparnos con la primera. Si $\lambda_i, \lambda_j \neq 0$, entonces,

$$\hat{\mathbf{u}}_i \cdot \hat{\mathbf{u}}_j = \left(\frac{A\hat{\mathbf{v}}_i}{\sigma_i}\right)^\top \frac{A\hat{\mathbf{v}}_j}{\sigma_j} = \frac{\hat{\mathbf{v}}_i A^\top A\hat{\mathbf{v}}_j}{\sigma_i\sigma_j} = \frac{\hat{\mathbf{v}}_i\lambda_j\hat{\mathbf{v}}_j}{\sigma_i\sigma_j} = \frac{\lambda_j}{\sigma_i\sigma_j}\hat{\mathbf{v}}_i \cdot \hat{\mathbf{v}}_j,$$

lo cual muestra la ortonormalidad en el resto de los casos, pues la última fracción es uno cuando

$i = j$. Esto le da así sentido a la construcción de las matrices para la factorización en valores singulares.

Discusión: Salvo permutaciones, sabemos que la descomposición generada por el Teorema Espectral es única, ¿qué se puede decir de la unicidad en el caso de la factorización del Teorema 7.2? Piensa en qué sucede si se hace una SVD a una matriz simétrica y definida positiva.

Ejercicio 35: Construye la factorización de la matriz del ejemplo anterior encontrando primero los eigenpares para $A^\top A$ y por separado para $AA^\top$. Observa las ventajas de hacer una u otra. Nota también que en ambos casos consigues básicamente los resultados del mismo ejemplo.

Lema 7.3 (Aproximaciones en SVD)**.** *Considérese $A \in \mathbb{R}^{n\times m}$ y su SVD como en el Teorema 7.2. Se definen las matrices de rango r a partir de*

$$A_r = \sum_{i=1}^{r} \sigma_i \hat{\mathbf{u}}_i \hat{\mathbf{v}}_i^\top.$$

Entonces, $\|A - A_r\|_2 = \sigma_{r+1}$, en particular $A_k = A$. De hecho, si $B \in \mathbb{R}^{n\times m}$ es de rango r, entonces, $\|A - A_r\|_F \leq \|A - B\|_F$; la matriz A_r es la mejor aproximación de A de rango r.

No es difícil mostrar que si σ_r es positivo, entonces A_r tiene rango r, se deja como **ejercicio moral**. Por otro lado, el uso de la norma 2 y de la norma de Frobenius tienen razón geométrica y analítica respectivamente. Para ver cómo se construyen estas normas mira el Apéndice A. De la definición del elipsoide se intuye la primera de las afirmaciones del lema.

Un gráfico de i contra σ_i, muestra que el comportamiento típico de los valores singulares es que su valor decae en magnitud rápidamente, por tanto, unos pocos valores pueden representar medianamente bien la matriz entera. Esto puede generar un ahorro terrible de información.

Ejemplo: Supongamos que $A \in \mathbb{R}^{n\times m}$ con $n = 10^{10}$ y $m = 10^6$, es tal que sus valores sigulares satisfacen $\sigma_i \geq 1$ para $i \leq 100$ y $\sigma_i \ll 1$ para los restantes. De este modo tenemos que A tiene $nm = 10^{16}$ entradas que tenemos que guardar en la memoria, pero A_{100} que satisface $\|A - A_{100}\|_2 \ll 1$ tiene solamente 100 valores singulares, de σ_1 a σ_{100}, además necesitamos 100 vectores $\hat{\mathbf{u}}_i$ y 100 vectores $\hat{\mathbf{v}}_i$, es decir, $100n = 10^{12}$ y $100m = 10^8$ datos respectivamente. Con ello, A_{100} produce efectos semejantes a la matriz A, pero usando solamente $10^{12} + 10^8 + 100$ datos, del orden de 10^{12} que es mucho menos que 10^{16}, ¡diez mil veces menos!

El ejemplo arriba es utilizado de manera exitosa en la compresión de archivos de imagen o audio, por ejemplo. El siguiente extracto de código toma una imagen, la guarda como los valores de una matriz con las gamas de rojo (R), verde (G) y azul (B) en cada una de las entradas, por

ejemplo, teniendo una imágen de 1200 por 800 pixeles, creamos una matriz de 3600×800 que descompodremos en sus valores singulares. Después se rearman varias de las matrices con 1 %, 2 %, 4 %, ..., 64 % del rango orginial para comparar los resultados contra la imagen de entrada.

```
""" EJEMPLO CON SVD:
 La idea de este código es usar la Descomposición en Valores Singulares, o SVD por
 su sigla en inglés, para guardar menos información de una figura, por ejemplo.
     El código tiene todavía detalles a ser mejorados en el manejo de las
 imágenes. Los colores llamativos y disonantes en los resultados no es lo común,
 estos deberían ser matizados con los colores del entorno. """
import matplotlib.pyplot as plt
import numpy as np
import time
from PIL import Image

""" Aquí se llama la imagen que debe estar en el mismo directorio que SVD.py """
img = Image.open("Ejemplo.jpg")
m, n = img.size[0], img.size[1]
""" ***** Atención especial se tiene en cuenta pues las imágenes y los arreglos
 tienen las dimensiones en opuesto. Por eso tomamos n y m al contrario. ***** """
# Queremos que las imágnes entren en la pantalla. (Esto es opcional.)
if n > 1500 or m > 1000:
    r = min(1500/n, 1000/m)
    m, n = int(r*m), int(r*n)
    img = img.resize((n, m))
img.show(title = 'Original')                  # Es bueno tener una imagen original

# Separamos la imagen en sus colores...
fuente = img.split()
# ... y las representamos como arreglos.
imgR, imgG, imgB = np.array(fuente[0]), np.array(fuente[1]), np.array(fuente[2])

# Hay que darle las dimensiones correctas...
img_mat = np.empty([3*n, m])
# ... y concatenarlo en un único arreglo con las tres bandas
img_mat[:n, :], img_mat[n:2*n, :], img_mat[2*n:3*n, :] = imgR, imgG, imgB

# Aquí utilizamos la SVD implementada en la libreria numpy de Python
U, s, V = np.linalg.svd(img_mat, full_matrices = False)
# Con el tamaño completo como falso, estamos calculando la descomposición, no la
# factorización; más económico.

tot = len(s)                          # Es util tener Sigma como la matriz completa
plt.plot(np.log(s))
plt.show()
time.sleep(1)
```

```
# Utilizaremos distintos porcentajes de la información proporcionada con SVD
for k in range(7):
    res = int(tot*(2**k)/100.)                  # A saber 1, 2, 4, 8, 16, 32, 64%
    # En cada una de estas resoluciones, tenemos:
    Ak = np.uint8( (U[:, :res]*s[:res])@V[:res, :] )
    # Debememos juntar cada una de las bandas...
    Rk = Image.fromarray(Ak[   :  n, :], mode = None)
    Gk = Image.fromarray(Ak[  n:2*n, :], mode = None)
    Bk = Image.fromarray(Ak[2*n:3*n, :], mode = None)
    # ... en una única imagen que visualizamos.
    reconK = Image.merge('RGB', (Rk, Gk, Bk))
    reconK.show()
    time.sleep(0.1)
```

7.5. Factorización QR

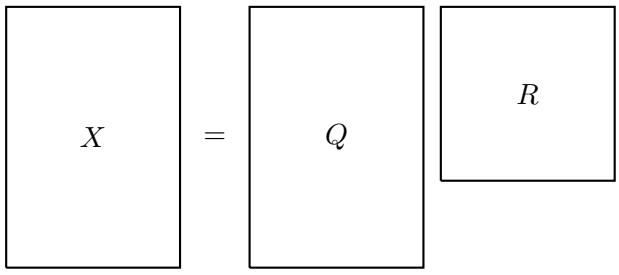
De momento hemos resuelto el principalmente el problema de considerar la información de una matriz cuadrada $A \in \mathbb{R}^{n\times n}$ de diversos modos. Sin embargo, en aplicaciones resulta que podemos tener información precaria o excedente, digamos con una matriz $X \in \mathbb{R}^{n\times k}$. Una matriz así, sin estructura, puede llevarnos a cálculos complejos que podemos simplificar de manera eficiente.

Afirmación 7.4. *Sea $X \in \mathbb{R}^{n\times k}$ con columnas linealmente independientes, así $k \leq n$. Entonces existe $Q \in \mathbb{R}^{n\times k}$ con columnas ortonormales y una matriz triangular superior $R \in \mathbb{R}^{k\times k}$ con elementos positivos en la diagonal, tales que*

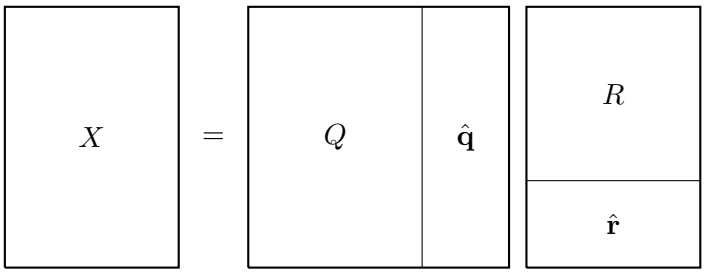
$$X = QR \tag{7.7}$$

se satisface con Q y R definidas de manera única.

Veamos la estructura de la fórmula (7.7). Dado que las columnas son linealmente independientes, tenemos que $k \leq n$ necesariamente, luego del tamaño vemos que la estructura es como sigue



con R la única matriz cuadrada. Esta es la **descomposición QR**. Sin embargo, Q puede ser complementada de forma que sea una matriz cuadrada y ortogonal de orden n. Es decir, podemos escribir a X como



donde $\hat{Q} = [Q\ \hat{\mathbf{q}}] \in \mathbb{R}^{n\times n}$ es una matriz ortonormal y ahora $\hat{R} = [R^\top\ \hat{\mathbf{r}}^\top]^\top \in \mathbb{R}^{n\times k}$ es rectangular con $\hat{\mathbf{r}} \in \mathbb{R}^{(n-k)\times k}$ una matriz nula. Esta es una **factorización QR** que no es única pues cualquier permutación de $\hat{\mathbf{q}}$ continua satisfaciendo $X = \hat{Q}\hat{R}$.

El concepto de ortogonalidad requiere un par de comentarios pues, en $\mathbb{R}^n$, dos vectores son ortogonales cuando su producto interior es nulo, es decir, si $\langle \mathbf{x}, \mathbf{y}\rangle = 0$ se satisface o de modo más algebraico, recordando la *igualdad de Cauchy* en la norma 2 (que usamos en esta sección),

$$\mathbf{x}^\top \mathbf{y} \;=\; \cos\theta\, \|\mathbf{x}\|\, \|\mathbf{y}\|$$

para un único ángulo $\theta \in [0, \pi)$ y $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. Este concepto es útil para pensar en los planos ortogonales o entender que la matriz Q en (7.7) satisface $Q^\top Q = I_k \in \mathbb{R}^{k\times k}$.

Esto nos ayuda para rever la igualdad en (7.7) del modo siguiente. Consideremos $S = R^{-1}$, entonces

$$[\mathbf{q}_1\ \mathbf{q}_2\ \cdots\ \mathbf{q}_k] \;=\; [\mathbf{x}_1\ \mathbf{x}_2\ \cdots\ \mathbf{x}_k] \begin{pmatrix} s_{11} & s_{12} & \cdots & s_{1k} \\ & s_{22} & \cdots & s_{2k} \\ & & \ddots & \vdots \\ & & & s_{kk} \end{pmatrix},$$

pues R es invertible y su inversa debe ser triangular superior. Podemos ver que línea a línea tenemos

$$\begin{aligned} \mathbf{q}_1 &= s_{11}\mathbf{x}_1\,, \\ \mathbf{q}_2 &= s_{12}\mathbf{x}_1 + s_{22}\mathbf{x}_2\,, \\ &\vdots \\ \mathbf{q}_k &= s_{1k}\mathbf{x}_1 + s_{2k}\mathbf{x}_2 + \cdots + s_{kk}\mathbf{x}_k\,, \end{aligned}$$

de donde vemos que $\mathbf{q}_1$ es solamente $\mathbf{x}_1$ normalizado, $\mathbf{q}_2$ es una combinación lineal normalizada de $\mathbf{x}_1$ y $\mathbf{x}_2$ que además es perpendicular a $\mathbf{q}_1$. ¿Por qué $\mathbf{q}_2$ se define entonces de manera única? Vemos que hay dos posibilidades para este vector de norma uno, sin embargo, hemos pedido que s_{22} sea positivo. Continuando con los vectores, vemos que $\mathbf{q}_i$ es una combinación lineal normalizada de los i primeros $\mathbf{x}_j$, pero además es ortogonal al espacio $\text{Gen}\,\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{i-1}\} = \text{Gen}\,\{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{i-1}\}$ y con $s_{ii} > 0$, lo cual garantiza su unicidad.

Aunque la existencia y unicidad de la descomposición QR esté resumida en el párrafo anterior, otra forma conlleva a su construcción. Supongamos que la descomposición y construcción son válidas para la submatriz de tamaño $k-1 \geq 1$, entonces tomamos

$$X \;=\; [X_1\ \mathbf{x}_k] \;=\; [Q_1\ \mathbf{q}_k]\begin{pmatrix} R_{11} & \mathbf{r}_{1k} \\ [0] & \rho_{kk} \end{pmatrix} \;=\; QR$$

en forma particionada, donde $X_1 = Q_1 R_{11} \in \mathbb{R}^{n\times(k-1)}$ es la factorización QR de X_1. Así, $\mathbf{q}_k \in \mathbb{R}^n$ es un vector ortonormal a $Q_1 \in \mathbb{R}^{n\times(k-1)}$, $\mathbf{r}_{1k} \in \mathbb{R}^{k-1}$ y $\rho_{kk} \in \mathbb{R}^+$ son los elementos a buscar.

Las ecuaciones restantes del producto con matrices particionadas nos dicen que queremos

$$\mathbf{x}_k \;=\; Q_1\mathbf{r}_{1k} \;+\; \rho_{kk}\mathbf{q}_k,$$

donde como $Q_1^\top \mathbf{q}_k = \mathbf{0}$ debe ser satisfecho, implica que $Q_1^\top \mathbf{x}_k = Q_1^\top Q_1 \mathbf{r}_{1k} = \mathbf{r}_{1k}$ también debe ser válido. Así, en este caso tenemos

$$\rho_{kk}\mathbf{q}_k \;=\; \mathbf{x}_k \;-\; Q_1\mathbf{r}_{1k} \qquad \Longrightarrow \qquad \rho_{kk} \;=\; \|\mathbf{x}_k \;-\; Q_1 Q_1^\top \mathbf{x}_k\|, \tag{7.8}$$

o mejor aún, dado que $Q_1 = X_1 R_{11}^{-1}$, tenemos $\mathbf{x}_k - Q_1\mathbf{r}_{1k} = \mathbf{x}_k - X_1(R_{11}^{-1}\mathbf{r}_{1k})$.

Veamos que de la subdivisión en las matrices particionadas tenemos que $\mathbf{x}_k = Q_1\mathbf{r}_{1k} + \rho_{kk}\mathbf{q}_k$ y de la expresión anterior es importante la forma del vector $\mathbf{r}_{1k} = Q_1^\top X_k$ que podemos ver como

```
R[:k, k] = numpy.transpose( Q[:, :k] ) @ X[:, k]
```

y de la forma para $\rho_{kk}\mathbf{q}_k$,

```
Q[:, k] = ( X[:, k] - Q[:, :k]@R[:k, k] )/R[k, k]
```

de donde tenemos, para la norma 2,

```
R[k, k] = numpy.linalg.norm( ( X[:, k] - Q[:, :k]@R[:k, k] )/R[k, k] )
```

Con estas ideas, tenemos el siguiente núcleo para el código.

Código: Algoritmo de Gramm-Schmidt (PYTHON)

```
for j in range(k):
    Q[ :, j] = X[:, j]
    R[:j, j] = numpy.transpose(Q[:, :j])@Q[:, j]          # Linea 10
    Q[ :, j] = Q[:, j] - Q[:, :j]@R[:j, j]                # Linea 20
    R[ j, j] = numpy.linalg.norm( Q[:, j] )
    Q[ :, j] = Q[:, j]/R[j, j]
```

Los pasos en `Linea 10` y `Linea 20` pueden ser reescritos como ciclos y después aprovechar esta estructura para escribir el algoritmo de Gramm-Schmidt modificado. Es importante mostrar que el

algoritmo es análogo a cambiar estas líneas por

```
for i in range(j - 1):
    R[i, j] = numpy.transpose(Q[:, i])@Q[:, j]
    Q[:, j] = Q[:, j] - Q[:, i]*R[i, j]
```

Ejercicio 36: Muestra que realmente obtienes la misma descomposición QR y cuenta el número de operaciones de ambas.

Observa que en la ecuación (7.8), la verdad estamos tomando $\rho_{kk}\mathbf{q}_k$ como la componente de X_k perpendicular al espacio $\text{Gen}\{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{k-1}\}$, es decir, $\rho_{kk}\mathbf{q}_k = (I - P)\mathbf{x}_k$ para I la identidad y P la proyección ortogonal de $\mathbf{x}_k$. Dado que podemos ver $X\mathbf{v}$ como una transformación lineal de $\mathbb{R}^k$ en $\mathbb{R}^n$, la imagen de $X\mathbb{R}^k$ es un subespacio $V \subset \mathbb{R}^n$ y $QQ^\top \in \mathbb{R}^{n\times n}$ tiene la virtud de tomar vectores en $\mathbb{R}^n$ y enviarlos a V. Además, definiendo $P_X = QQ^\top$ como la *proyección a este subespacio* y $P_\perp = I - P_X$ como la *proyección al subespacio ortogonal*, se tiene

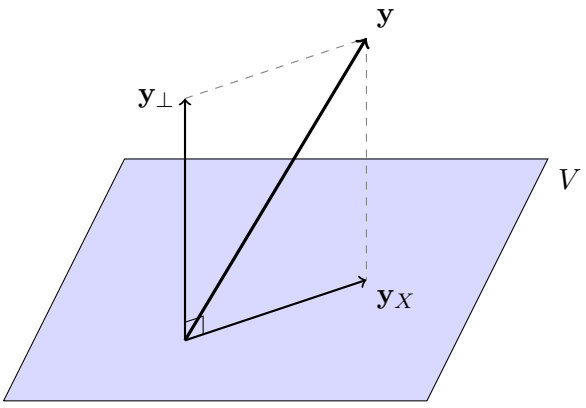
$$\begin{aligned} \forall \mathbf{x} \in V \quad &\Longrightarrow \quad P_X\mathbf{x} = \mathbf{x} \quad \text{y} \quad P_\perp\mathbf{x} = \mathbf{0}, \\ \forall \mathbf{x} \perp V \quad &\Longrightarrow \quad P_X\mathbf{x} = \mathbf{0} \quad \text{y} \quad P_\perp\mathbf{x} = \mathbf{x}, \end{aligned}$$

así, para todo $\mathbf{y} \in \mathbb{R}^n$ tenemos $\mathbf{y} = P_X\mathbf{y} + P_\perp\mathbf{y}$, que podemos ver fácilmente pues $\mathbf{y}_X = Q(Q^\top\mathbf{y})$ está en el generado por $\{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k\}$ y como mostramos $\mathbf{y}_\perp = \mathbf{y} - \mathbf{y}_X$ está en el subespacio ortogonal.

7.5.1. Mejor aproximación en espacios con producto interno

Implícitamente en la proyección hemos usado el concepto del producto interior, pues calculamos $\mathbf{q}_j^\top\mathbf{x}_k$ con $j = 1, 2, \ldots, k-1$ varias veces. Es fácil concebir que si buscamos una solución $\mathbf{y}_X$ en el subespacio $V \leq \mathbb{R}^n$ y tenemos un vector genérico $\mathbf{y} \in \mathbb{R}^n$ podemos, como antes, tomar

$$\mathbf{y} \;=\; \mathbf{y}_X + \mathbf{y}_\perp \;=\; P_X\mathbf{y} + P_\perp\mathbf{y}$$



y considerar $\mathbf{y}_X$ como la mejor aproximación de $\mathbf{y}$ en V, pues $\|\mathbf{y} - \mathbf{y}_X\| \leq \|\mathbf{y} - \mathbf{x}\|$ para todo $\mathbf{x} \in V$.

Otra manera de verlo es buscar coeficientes $\beta_1, \beta_2, \ldots, \beta_k$ tales que

$$\mathbf{y}_X \;=\; \beta_1\mathbf{x}_1 + \beta_2\mathbf{x}_2 + \cdots + \beta_k\mathbf{x}_k \;\approx\; \mathbf{y}$$

se satisface. Es decir, con $X = [\mathbf{x}_1\ \mathbf{x}_2\ \cdots\ \mathbf{x}_k]$, $\mathbf{b} = (\beta_1, \beta_2, \ldots, \beta_k)^\top$ y $\mathbf{y} \in \mathbb{R}^n$, queremos

$$\text{minimizar } \|\mathbf{y} - X\mathbf{b}\|,$$

que al final de cuentas es equivalente a minimizar

$$\begin{aligned} \|\mathbf{y} - X\mathbf{b}\|^2 &= \|P_X(\mathbf{y} - X\mathbf{b}) + P_\perp(\mathbf{y} - X\mathbf{b})\|^2 \\ &= \|P_X(\mathbf{y} - X\mathbf{b})\|^2 + \|P_\perp(\mathbf{y} - X\mathbf{b})\|^2 \qquad \text{(Pitágoras)} \\ &= \|P_X\mathbf{y} - X\mathbf{b}\|^2 + \|P_\perp\mathbf{y}\|^2. \end{aligned}$$

Observamos que no tenemos control sobre $P_\perp\mathbf{y}$; es un valor fijo. Así, nuestro problema de minimización se resuelve al minimizar el primer término de la última igualdad. Además, vemos que para el problema en cuestión existe un único $\mathbf{b} \in \mathbb{R}^k$ tal que $X\mathbf{b} = P_X\mathbf{y}$, con lo cual, este término es nulo y así hemos minimizado $\|\mathbf{y} - X\mathbf{b}\|$. Un problema clásico en esta dirección está dado por los *mínimos cuadrados*.

Ejemplo: Supongamos que tenemos las emisiones radioactivas de una mezcla de isótopos distintos, uno de ellos con un decaimiento $ae^{-\lambda t}$ y el otro $be^{-\mu t}$ con los parámetros a, b desconocidos, pero con las razones de decaimiento λ y μ estimadas. Tenemos diversas mediciones de la radiación y_i a los tiempos $t_1, t_2, \ldots, t_n$ y queremos determinar los parámetros a, b, c tales que

$$\begin{aligned} y_1 &= ae^{-\lambda t_1} + be^{-\mu t_1} + c, \\ y_2 &= ae^{-\lambda t_2} + be^{-\mu t_2} + c, \\ &\vdots \\ y_n &= ae^{-\lambda t_n} + be^{-\mu t_n} + c, \end{aligned}$$

sea lo mejor aproximado posible y donde hemos introducido el parámetro c que incluye cierta radiación de fondo. Buscamos la solución a este problema y vemos que está sobre determinado, lo cual sugiere que puede no tener una solución.

Si consideramos cada error r_i al determinar a, b, c como la diferencia

$$r_i = y_i - (ae^{-\lambda t_i} + be^{-\mu t_i} + c),$$

para $i = 1, 2, \ldots, n$, entonces nos gustaría minimizar todos estos errores. Considerando la

norma euclidiana, tenemos que el error total es

$$\rho(a,\, b,\, c) \;=\; \sqrt{r_1^2 + r_2^2 + \cdots + r_n^2}\,.$$

Podemos diferenciar $\rho^2(a,\, b,\, c)$ con respecto de a, b, c y encontrar el mínimo, lo que nos lleva a las *ecuaciones normales* del problema de mínimos cuadrados.

Reformulando como antes

$$\mathbf{y} \;=\; \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \qquad \mathbf{b} \;=\; \begin{pmatrix} a \\ b \\ c \end{pmatrix} \qquad \text{y} \qquad X \;=\; \begin{pmatrix} e^{-\lambda t_1} & e^{-\mu t_1} & 1 \\ e^{-\lambda t_2} & e^{-\mu t_2} & 1 \\ \vdots & \vdots & \vdots \\ e^{-\lambda t_n} & e^{-\mu t_n} & 1 \end{pmatrix},$$

queremos minimizar $\|\mathbf{y} - X\mathbf{b}\|$, que puede probarse que es resolver el problema lineal

$$(X^\top X)\mathbf{b} \;=\; X^\top \mathbf{y}$$

y es análogo a encontrar las ecuaciones normales. En nuestro abordaje sabemos que $X^\top(\mathbf{y} - X\mathbf{b}) = 0$ se satisface dada la ortogonalidad, pero esto no es aprovechado arriba.

La manera moderna de resolver esta cuestión es con la factorización QR. Existe el camino geométrico, pero notamos directamente que $X = QR$ con R triangular superior e invertible y Q ortogonal, entonces,

$$\begin{aligned} (X^\top X)\mathbf{b} \;=\; X^\top \mathbf{y} \quad &\Longrightarrow \quad (R^\top Q^\top QRX)\mathbf{b} \;=\; R^\top Q^\top \mathbf{y} \\ &\Longrightarrow \quad R^\top(R\mathbf{b}) \;=\; R^\top(Q^\top \mathbf{y}) \qquad \Longrightarrow \qquad R\mathbf{b} \;=\; Q^\top \mathbf{y}, \end{aligned}$$

donde este último sistema de ecuaciones es llamado **ecuación QR**.

Es claro que resolver la ecuación QR tiene sus ventajas dado que tenemos un producto por una matriz ortogonal para después encontrar $\mathbf{b}$ con una substitución hacia adelante. Además, este problema resulta ser muy estable.

Ejercicio 37: Calcula el número de operaciones que se realizan para resolver el problema de mínimos cuadrados vía la ecuación QR si comienzas con $X \in \mathbb{R}^{n\times k}$.

Observamos que aunque parece costoso encontrar Q y R antes de resolver esta ecuación QR, el gasto no es mucho mayor que evaluar el producto $X^\top X$. Además, cuando $n \gg k$, una forma muy práctica de hacer esto es factorar Q con las *transformaciones de Householder* que son matrices de la forma

$$H \;=\; I - \mathbf{u}\mathbf{u}^\top, \qquad \|\mathbf{u}\| \;=\; \sqrt{2}$$

de modo que

$$H^\top H \;=\; (I - \mathbf{u}\mathbf{u}^\top)^2 \;=\; I - 2\mathbf{u}\mathbf{u}^\top + \mathbf{u}(\mathbf{u}^\top\mathbf{u})\mathbf{u}^\top \;=\; I - 2\mathbf{u}\mathbf{u}^\top + 2\mathbf{u}\mathbf{u}^\top \;=\; I$$

se satisface.

Además, las matrices ortogonales satisfacen tener número de condición bajo, de hecho, $\kappa(Q) = 1$ se sastisface en la norma 2. Es por esto que en el análisis numérico se les tiene tanto aprecio. Mira el Apéndice A para ver el concepto del número de condición.

Ejercicio 38: Muestra que en la norma 2, la condición $\kappa(Q)$ de una matriz ortonormal es uno, lo cual dice que numéricamente es una matriz estable. ¿Qué sucede si calculas la condición con otra norma que sea equivalente?

Capítulo 8

Ajuste de funciones

Un problema clásico en matemáticas y en áreas de aplicación es encontrar una función $f : \mathbb{R} \to \mathbb{R}$ que represente de mejor manera posible los nodos o puntos coordenados en un plano, digamos (x_0, y_0), (x_1, y_1), $\ldots$, (x_n, y_n). De hecho, si es posible, tal que se satisfaga $f(x_k) = y_k$ para todo $k = 0, 1, \ldots, n$.

En principio, uno puede tener información extra de la función que estamos procurando y así aprovecharse de esta situación para desarrollar métodos específicos. Algunos de estos métodos nos pueden llevar a buscar polinomios de grado n y que llamaremos *interpolación*, o tal vez, polinomios de grado $k < n$ para ciertas regiones, garantizando la continuidad y algún nivel de diferenciabilidad, conocido dentro de los métodos *splines*. Otro método es, como hemos hecho con *mínimos cuadrados*, al buscar funciones de forma específica que aún sin pasar por los puntos, aproximan éstos de modo que se genera el menor error posible dentro de cierto criterio.

8.1. Interpolación

Veremos que hay que ser cuidadosos aún en el caso más simple que es la interpolación. Por un lado recordamos la aritmética de punto flotante y cómo el calcular magnitudes de distintos órdenes puede introducir redondeos o truncamientos indeseados; también tenemos que tener presente el cancelamiento catastrófico. Sin embargo, por otro lado, la teoría estará de nuestro lado, pues el teorema fundamental del álgebra garantiza que por $n+1$ puntos pasa un único polinomio de grado n. A saber, dados los puntos (x_k, y_k) para $k = 0, 1, \ldots, n$, existen valores a_k determinados de manera única y tales que el polinomio

$$p(x) \;=\; a_0 \,+\, a_1\, x \,+\, a_2\, x^2 \,+\, \cdots \,+\, a_n\, x^n,$$

satisface $p(x_k) = y_k$ para toda $k = 0, 1, \ldots, n$. Esta es una de las representaciones de un polinimio, veremos otras y compararemos cuál puede ser más útil.

Vemos entonces que plausible encontrar nuestro polinomio a partir de un sistema lineal de ecuaciones, pues queremos resolver

$$\begin{aligned} a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_n x_0^n &= y_0, \\ a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_n x_1^n &= y_1, \\ &\vdots \\ a_0 + a_1 x_n + a_2 x_n^2 + \cdots + a_n x_n^n &= y_n. \end{aligned}$$

Sabemos cómo reescribir este problema como un problema de la forma $V\mathbf{a} = \mathbf{y}$, para $V = V(\mathbf{x})$, que ya sabemos cómo trabajar. Sin embargo, la **matriz de Vandermonde** definida por las líneas con las distintas potencias de las abscisas de los puntos dados es bastante ineficiente. Específicamente escribimos esta matriz como

$$V = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}.$$

Ahora podemos notar que por ejemplo, para las abscisas $x_0 = 10,000$, $x_1 = 10,001$, $x_2 = 10,002$, el *mal condicionamiento* de ella.

Discusión: En PYTHON la *condición* de esta matriz se puede calcular al evaluar explícitamente su inversa y sus normas con `numpy.linalg.norm`; el modo directo es con `numpy.linalg.cond`. Notemos que $\|V\|_2 = \mathcal{O}(10^8)$ y entendamos porqué la inversa dará resultados vagos para a_0. Podemos resolver el problema cuando

$$\mathbf{y} = (100035403420111,\, 345,\, 0.000004324270000132064)^{\mathrm{T}}$$

y calcular el error relativo de $p(\mathbf{x})$ evaluado así y los valores reales de $\mathbf{y}$.

¿Qué tal trasladar el sistema coordenado y tomar en lugar de las abscisas x_k, a las abscisas $t_k := x_k - x_0$? De este modo, tenemos que $t_0 = 0$, $t_1 = 1$ y $t_2 = 2$, con lo cual su matriz de Vandermonde es más sencilla:

$$\tilde{V} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{pmatrix}, \qquad \text{y así} \qquad \tilde{V}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -3/2 & 2 & -1/2 \\ 1/2 & -1 & 1/2 \end{pmatrix}.$$

En este caso tenemos $\|\tilde{V}\|_1 = 5$ y $\|\tilde{V}^{-1}\|_1 = 3$, con lo que obtenemos una condición (en norma 1) $\kappa_1(\tilde{V}) = 15$ que es increíblemente menor que la condición de la matriz original

$$\kappa_1(V) = \|V\|_1 \, \|V^{-1}\|_1 = 300060005 \times 100040003 = 3.0018 \cdots \times 10^{16}.$$

Observa que la condición de V nos dice que utilizar la matriz original para resolver el sistema no garantiza ni una cifra correcta aún con la precisión doble.

Ejercicio 39: Resuelve el problema anterior con los valores de $\mathbf{y}$ dados pero en el sistema coordenado con abscisa t. ¿Cuál es la diferencia con el error relativo?

En general las matrices de Vandermonde no son un buen camino en la práctica numérica. Por un lado son bastante mal condicionadas y sus procesos pueden llevar a errores relativamente grandes. Por otro lado, aunque resolvamos de modo analítico, la eliminación gaussiana nos tomará tiempo en un proceso $\mathcal{O}(n^3)$; esto lo eliminaremos en breve.

Un otro problema que no es tan grave, es el hecho de que no hemos garantizado que estas matrices sean regulares y siempre tengan solución. No es difícil darse cuenta de que si $x_i \neq x_j$ para toda $i \neq j$, esto debe ser verdad. Un camino es mostrar el siguiente ejercicio.

Ejercicio 40: Una matriz de Vandermonde de orden n se escribe como $V = \left[\left(x_{i-1}^{j-1}\right)_{ij}\right]$; determina los valores posibles de i, j. Muestra que su determinante no es nulo si las abscisas son distintas, en especial, muestra que la fórmula

$$\det(V) \;=\; \prod_{0 \leq i \leq j \leq n} (x_j - x_i).$$

siempre se satisface. Intenta con el uso de los menores, escribir explícitamente la inversa para $n = 2$, 3 y 4.

Piensa que este resultado garantiza la existencia y unicidad del problema $V\mathbf{a} = \mathbf{b}$, con lo cual se garantiza que existe un único $p(x)$ pasando por los puntos (x_k, y_k) para $k = 0, 1, \ldots, n$ si $x_i \neq x_j$ cuando $i \neq j$.

8.1.1. Polinomios de Lagrange

Todavía no hemos garantizado realmente ni la existencia ni la unicidad del problema que queremos resolver solamente ha quedado demostrado entre líneas. La segunda cuestión se resuelve fácilmente con la teoría algebraica mencionada previamente.

Teorema 8.1 (Fundamental del álgebra)**.** *Si un polinomio de grado n se anula en $n+1$ puntos distintos, entonces ese polinomio es nulo.*

Luego como $p(x)$ es de grado menor o igual a n podemos suponer que si $q(x)$ también es un polinomio que pasa por los puntos (x_k, y_k) y además satisface que $\text{grado}(q) \leq n$, entonces $r(x) := p(x) - q(x)$ se anula en x_k para $k = 0, 1, \ldots, n$. Es decir, $r(x)$ es de grado menor o igual a n y se anula en $n+1$ puntos; del Teorema 8.1 vemos que $p(x) \equiv q(x)$.

La idea entonces es generar una serie de polinomios, la suma de los cuales nos dé como resultado el polinomio que buscamos. Lagrange vino con la idea de construir $n+1$ polinomios $\ell_k(x)$ tal que cada uno de ellos cumpliera con un propiedad muy específica, ésta se basa en que se satisfagan ciertas igualdades en las abscisas x_k, a saber,

$$\ell_k(x_i) \;=\; \begin{cases} 0, & \text{si } i \neq k, \\ 1, & \text{si } i = k, \end{cases} \qquad \text{para} \qquad i \;=\; 0,\, 1,\, \ldots,\, n.$$

Es claro por el Teorema 8.1 que estos $n+1$ **polinomios de Lagrange** son únicos. Además, sus cualidades son tales que nos permiten construir el polinomio deseado como

$$p(x) \;=\; y_0\,\ell_0(x) \;+\; y_1\,\ell_1(x) \;+\; \cdots \;+\; y_n\,\ell_n(x),$$

que resulta ser un polinomio de grado n, pues cada uno de sus polinomios ℓ_k formantes lo es.

Una manera directa de mostrar que este polinomio satisface las características que pedimos, viene del hecho que cada componente de Lagrange se anula en todas las abscisas menos una,

$$p(x_i) \;=\; \sum_{k=0}^{n} y_k\,\ell_k(x_i) \;=\; \sum_{k=0}^{n} y_k\,\delta_{k,i} \;=\; y_i, \qquad \text{para todo} \qquad i \;=\; 0,\, 1,\, \ldots,\, n,$$

aquí hemos usado la *delta de Kronecker* que resume las propiedades de Lagrange al ser definida como $\delta_{k,i} = 0$ si $i \neq k$ y $\delta_{k,i} = 1$ solo cuando $i = k$.

Ejemplo: El polinomio que pasa por los puntos $(-1,\,1)$, $(0,\,0)$ y $(1,\,1)$ tiene los polinomios de Lagrange

$$\ell_0(x) \;=\; \frac{(x-0)(x-1)}{(-1-0)(-1-1)} \;=\; \frac{x(x-1)}{2}, \qquad \ell_1(x) \;=\; \frac{(x-(-1))(x-1)}{(0-(-1))(0-1)} \;=\; 1-x^2,$$

$$\ell_2(x) \;=\; \frac{(x-(-1))(x-0)}{(1-(-1))(1-0)} \;=\; \frac{x(x+1)}{2}.$$

Por lo tanto, este polinomio es

$$p(x) \;=\; 1\,\ell_0(x) \;+\; 0\,\ell_1(x) \;+\; 1\,\ell_2(x) \;=\; \frac{x(x-1)+x(x+1)}{2} \;=\; x^2,$$

como podemos verificar fácilmente.

Ejercicio 41: Escribe con los polinomios de Lagrange arriba la recta interpolada en los tres puntos $(-1,\,-1)$, $(0,\,0)$ y $(1,\,1)$.

Este ejemplo, muestra de manera sencilla cómo construir los polinomios de Lagrange. Para obtener las evaluaciones nulas en x_i cuando $k \neq i$, simplemente necesitamos que exista el factor multiplicativo $(x - x_i)$. Para garantizar que el producto de todos estos es uno cuando evaluamos en x_k dividimos por el mismo factor evaluado en x_k, es decir, cada elemento puede verse como el producto de las razones $(x - x_i)/(x_k - x_i)$ cuando $i \neq k$. El producto de estos factores es el k-ésimo polinomio de Lagrange. En términos matemáticos, tenemos para cada $k = 0, 1, \ldots, n$,

$$\begin{aligned} \ell_k(x) &= \frac{x - x_0}{x_k - x_0} \frac{x - x_1}{x_k - x_1} \cdots \frac{x - x_{k-1}}{x_k - x_{k-1}} \frac{x - x_{k+1}}{x_k - x_{k+1}} \cdots \frac{x - x_n}{x_k - x_n} \\ &= \frac{(x - x_0) \cdots (x - x_{k-1})\,(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)} = \prod_{i \neq k} \frac{x - x_i}{x_k - x_i}. \end{aligned}$$

Ahora tenemos un modo directo de escribir un polinomio que satisface nuestras restricciones. Sin embargo, esta manera de construir un polinomio puede ser numéricamente muy inestable. Basta pensar en la cantidad de cálculos que realizaremos para una recta interpolada por $n \gg 1$ puntos.

Discusión: Habría que pensar en la cantidad de cálculos, en los cancelamientos castastróficos y en la suma de valores de distintos órdenes de magnitud. En Python podemos retomar el ejemplo de la pág. 17 para calcular y graficar

$$p(x) = (x - 1)^{10}$$

y su expresión por extenso

$$q(x) = 1 - 10x + 45x^2 - 120x^3 + 210x^4 - 252x^5 + 210x^6 - 120x^7 + 45x^8 - 10x^9 + x^{10},$$

al rededor de $x = 1$. Se puede también pensar en otras formas de ordenar $q(x)$ o factorizar pedazos pequeños de alguna manera. ¿Qué debe suceder en estos casos?

8.2. División sintética

Como hemos visto, el problema de encontrar el polinomio interpolador puede ser complicado, ser mal condicionado o inclusive ser inestable a la hora de evaluar; este es un segundo problema que también atacaremos. En los polinomios de Lagrange la evaluación de

$$(x - x_0)\,(x - x_1) \cdots (x - x_n)$$

puede fácilmente tener un *overflow*, un *underflow* o más comúnmente cancelaciones catastróficas cuando además consideramos el numerador.

Parece que para el segundo problema que atacaremos, hay que poner especial atención simplemente de que la evaluación se realiza para el polinomio en su forma canónica, es decir, multiplicando constantes por distitnas potencias de x. A saber,

$$p(x) \;=\; a_n\, x^n \,+\, \cdots \,+\, a_2\, x^2 \,+\, a_1\, x \,+\, a_0\,, \tag{8.1}$$

lo cual nos puede dar problemas en su evaluación, pues no tenemos control del tamaño de cada sumando para todos los puntos del intervalo de interés. Una manera de minimizar esto, es utilizando una evaluación anidada o una *división sintética*. Si observamos bien, vemos que (8.1) puede ser reescrito, a modo de quitar las potencias, por

$$p(x) \;=\; \Big(\big((\cdots\big((a_n\, x \,+\, a_{n-1})\, x \,+\, a_{n-2}\big)\cdots\big)\, x \,+\, a_2\big)\, x \,+\, a_1\Big)\, x \,+\, a_0.$$

Esto induce una evaluación sucesiva

$$\begin{aligned}
s_0 &= a_n,\\
s_1 &= (a_n)\, x \,+\, a_{n-1} \;=\; (s_0)\, x \,+\, a_{n-1},\\
s_2 &= \big((a_n)\, x \,+\, a_{n-1}\big)\, x \,+\, a_{n-2} \;=\; (s_1)\, x \,+\, a_{n-2},\\
&\;\;\vdots\\
s_{n-1} &= (s_{n-2})\, x \,+\, a_1,\\
s_n &= (s_{n-1})\, x \,+\, a_0,
\end{aligned}$$

que simplemente nos dice que a cada paso debemos de multiplicar por el valor de x y sumar el próximo coeficiente. Esta manera de anidar las potencias de x nos lleva a un algoritmo económico computacionalmente que se programa además en pocas líneas. Tenemos n productos y n sumas en el siguiente código.

Código: Evaluación de un polinomio (PYTHON)

```
p = a[-1]
for k in range(n - 1, 0, -1):
    p = p*x + a[k]
```

Así, en las entradas de esta rutina, sólo es necesario dar el valor de x donde el polinomio será evaluado y los coeficientes de este.

Ejercicio 42: Toma el desarrollo utilizado en la última discusión para ver la mejora en este sentido en PYTHON.

8.2.1. La forma interpolante de Newton

Por un lado vemos que la evaluación anidada es fácil de realizarse, pero es difícil conseguir el polinomio en esta forma. Por el otro lado, con Lagrange encontramos el polinomio de modo sencillo con una evaluación poco satisfactoria. El término medio nos lo dará la forma interpolante de Newton.

En esta sección evaluaremos los polinomios en valores $t \in \mathbb{R}$ en lugar de x, así no habrá confusión con el vector $\mathbf{x} = (x_0, x_1, \ldots, x_n)$ que contiene las abscisas de los nodos. La idea básica es construir de forma anidada el polinomio en sí, esto lo conseguiremos al considerar la base de polinomios

$$\{1, t - x_0, (t - x_0)(t - x_1), \ldots, (t - x_0)(x - x_1) \cdots (t - x_{n-1})\},$$

de modo que podemos buscar el polinomio

$$\begin{aligned} p(t) &= c_n (t - x_{n-1})(t - x_{n-2}) \cdots (t - x_0) + \cdots + c_2 (t - x_1)(t - x_0) + c_1 (t - x_0) + c_0 \\ &= \Big(\big((\cdots (c_n (t - x_{n-1}) + c_{n-1}) (t - x_{n-2}) + c_{n-2}) \cdots \big) (t - x_1) + c_1 \Big) (t - x_0) + c_0. \end{aligned}$$

Nota que podemos crear un algoritmo sencillo para esta evaluación. En lugar del código anterior ahora tenemos que dar el vector $\mathbf{c} = (c_0, c_1, \ldots, c_n)$ con los coeficientes del polinomio, el vector referente a los nodos $\mathbf{x}$ y el valor t donde será evaluado. Llegamos a un código sencillo que realiza $2n$ sumas y n productos.

Código: Evaluación de la división sintética (PYTHON)

```
""" Vemos que es importante dar tanto el arreglo de coeficientes c[·]
    como el arreglo de los nodos del polinomio, x[·]. """
p = c[-1]
for k in range(n - 1, -1, -1):
    p = p*(t - x[k]) + c[k]
```

El costo aun mayor que en la forma original (8.1), continua siendo $\mathcal{O}(n)$. La mejora vendrá a la hora de buscar los coeficientes.

Discusión: Debemos explicar la existencia de los coeficientes $c_0, c_1, \ldots, c_n$. Además de ello, debemos de garantizar que existe una única forma de encontrarlo.

Construyamos un algoritmo para encontrar los valores en $\mathbf{c}$. Para ello, notemos que si el polinomio debe pasar por los nodos $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, entonces se debe satisfacer el sistema de ecuaciones lineales

$$
\begin{aligned}
p(x_0) &= c_0, \\
p(x_1) &= c_0 + c_1\,(x_1 - x_0), \\
p(x_2) &= c_0 + c_1\,(x_2 - x_0) + c_2\,(x_2 - x_0)(x_2 - x_1), \\
&\vdots \\
p(x_n) &= c_0 + c_1\,(x_n - x_0) + c_2\,(x_n - x_0)(x_n - x_1) + \cdots + c_n\,(x_n - x_0)\cdots(x_n - x_{n-1}),
\end{aligned}
$$

que podemos reescribir como una ecuación lineal $A\mathbf{c} = \mathbf{y}$ donde la matriz A es triangular inferior

$$
A = \begin{pmatrix}
1 & 0 & 0 & \cdots & 0 \\
1 & x_1 - x_0 & 0 & \cdots & 0 \\
1 & x_2 - x_0 & (x_2 - x_1)(x_2 - x_0) & & 0 \\
\vdots & \vdots & \vdots & \ddots & \\
1 & x_n - x_0 & (x_n - x_1)(x_n - x_0) & \cdots & (x_n - x_{n-1})\cdots(x_n - x_1)(x_n - x_0)
\end{pmatrix},
$$

que se resuelve de modo sencillo con la substitución hacia adelante vista en la Sec. 7.1.2.

Ejercicio 43: Muestra que la matriz A descrita arriba es la matriz de Vandermonde para la base $\{1,\, t - x_0,\, \ldots,\, (t - x_0)\cdots(t - x_{n-1})\}$ en lugar de la base usual $\{1,\, t,\, \ldots,\, t^n\}$. Comprueba que se cumple un análogo al del Ej. 40.

Ejercicio 44: Definamos $f[x_k] = f(x_k)$ para $k = 0,\, 1,\, \ldots,\, n$ y de modo recursivo

$$
f[x_i,\, x_{i+1},\, \ldots,\, x_{i+j+1}] = \frac{f[x_{i+1},\, x_{i+2},\, \ldots,\, x_{i+j+1}] - f[x_i,\, x_{i+1},\, \ldots,\, x_{i+j}]}{x_{i+j+1} - x_i}
$$

para $0 \leq i,\, j,\, i + j + 1 \leq n$. Muestra que de este modo se tienen los coeficientes de la forma interpolante de Newton pues se satisface

$$
c_k = f[x_0,\, x_1,\, \ldots,\, x_k], \qquad \text{para toda} \qquad k \in \{0,\, 1,\, \ldots,\, n\}.
$$

Sugerencia. Observa la creación de los términos Q_{ij} en el *método de Neville* de la Sec. 8.3.1.

8.3. Error en la interpolación

Deberíamos hacer una pequeña pausa para poder explicar el error que se produce con los valores interpolados y ver porqué no es una buena idea hacer extrapolación a menos que no tengamos mayor información. Veremos con el denominado *fenómeno de Runge* que una mala elección de nodos nos puede llevar a oscilaciones gigantescas.

Teorema 8.2. *Si $x_0, x_1, \ldots, x_n$ son números distintos en el intervalo $I = [a, b]$ y si $f \in \mathcal{C}^{n+1}(I)$, entonces para cada $x \in I$, existe un número $\xi(x) \in (a, b)$ tal que*

$$f(x) \;=\; p(x) \;+\; \frac{f^{n+1}\big(\xi(x)\big)}{(n+1)!}\,(x-x_0)(x-x_1)\cdots(x-x_n), \tag{8.2}$$

para $p(x)$ el polinomio interpolante en $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$.

Prueba. Dado que $f(x_k) = p(x_k)$ y los multiplicandos a la derecha de (8.2) se anulan para $x = x_k$ con $k = 0, 1, \ldots, n$, tenemos que $\xi(x_k)$ puede ser tomado de modo arbitrario en estos nodos. Para cada $x \neq x_k$ fijo, definimos la función auxiliar $\varphi : [a, b] \to \mathbb{R}$ por

$$\begin{aligned}\varphi(t) \;&=\; f(t) \;-\; p(t) \;-\; \big[f(x) \;-\; p(x)\big]\,\frac{(t-x_0)\,(t-x_1)\,\cdots\,(t-x_n)}{(x-x_0)\,(x-x_1)\,\cdots\,(x-x_n)}\\ &=\; f(t) \;-\; p(t) \;-\; \big[f(x) \;-\; p(x)\big]\,\prod_{k=0}^{n}\frac{t-x_k}{x-x_k}.\end{aligned}$$

Notamos que $\varphi \in \mathcal{C}^{n+1}(I)$ y que $\varphi(x_k) = 0$ para $k = 0, 1, \ldots, n$. Además, se tiene que

$$\begin{aligned}\varphi(x) \;&=\; f(x) \;-\; p(x) \;-\; \big[f(x) \;-\; p(x)\big]\,\prod_{k=0}^{n}\frac{x-x_k}{x-x_k}\\ &=\; f(x) \;-\; p(x) \;-\; \big[f(x) \;-\; p(x)\big] \;=\; 0,\end{aligned}$$

es decir, $\varphi(t)$ se anula en $n+2$ puntos y por el teorema de Rolle sabemos que existe al menos un $\xi = \xi(x)$ tal que $\varphi^{(n+1)}(\xi) = 0$, en extenso, tenemos

$$\begin{aligned}0 \;=\; \varphi^{(n+1)}(\xi) \;&=\; f^{(n+1)}(\xi) \;-\; p^{(n+1)}(\xi) \;-\; \big[f(x) \;-\; p(x)\big]\,\frac{\mathrm{d}^{n+1}}{\mathrm{d}t^{n+1}}\left[\prod_{k=0}^{n}\frac{t-x_k}{x-x_k}\right]_{t=\xi}\\ &=\; f^{(n+1)}(\xi) \;-\; 0 \;-\; \big[f(x) \;-\; p(x)\big]\,\frac{(n+1)!}{\prod_{k=0}^{n}(x-x_k)}.\end{aligned}$$

Al despejar esta identidad, se obtiene el resultado deseado. ■

En el fenómeno de Runge, los errores son medidos y hasta cierto modo podríamos pensar como producidos por el término

$$\omega_n(x) \;=\; (x-x_0)\,(x-x_1)\,\cdots\,(x-x_n),$$

cuando el número de puntos n crece. Así, si los nodos tienen el orden $x_0 < x_1 < \cdots < x_n$, al escoger un valor de $x \in (x_{n-1}, x_n)$ tendrá tamaño relativamente grande cuando se tomen los multiplicandos $(x - x_k)$ para los primeros valores de k, con lo cual, si estos son muchos, este valor crece.

Ejemplo: Consideremos la función

$$f(t) \;=\; \frac{1}{1+t^2}, \qquad \text{con} \qquad t \;\in\; [-5,\,5].$$

Para ella, hagamos una rutina que para un n dado, realice las siguientes operaciones:

1. Considere los puntos que dividen $I := [-5,\,5]$ en n subintervalos iguales (por ejemplo, para $n = 2$ toma $x_0 = -5$, $x_1 = 0$, $x_2 = 5$ y los respectivos $f_i = f(x_i)$),
2. Aproxime numéricamente el polinomio que pasa por los $n+1$ puntos $(x_i,\, f_i)$,
3. Grafique en el intervalo el polinomio $p_n(x)$ y la función $f(x)$ en el intervalo I,
4. Calcule en I el máximo error absoluto e_M y el máximo error relativo ρ_M entre el polinomio y la función original. ¿Cuántos puntos deberíamos usar para estas pruebas?

Al incrementar n paulatinamente veremos que para $x = 0$ las aproximaciones son cada vez mejores, sin embargo, comienzan a haber oscilaciones grandes e indeseables en los extremos.

Nos encontramos en una situación donde la cantidad cada vez mayor de elementos de tamaño próximo a 10 en los productos $(x - x_k)$ para k pequeño y x próximo de cinco, produce un producto que va como $\mathcal{O}(10^n)$; un poco menor, la verdad.

Podemos minimizar el error producido en el fenómeno de Runge al buscar los mejores valores para evaluar la función dentro del intervalo. Estos locales son conocidos como los **puntos de Chebyshev** que provienen del siguiente problema:

Dado un número n, se deben encontrar los $n+1$ nodos $-1 \le x_0 < x_1 < \cdots < x_n \le 1$, tales que se minimice el valor

$$\max_{x\in[-1,\,1]} \big|\omega_n(x)\big|.$$

Es decir, es un problema min-max.

Este problema es parte del origen de los *polinomios de Chebyshev* las raíces de los cuales están dados por los valores

$$x_k \;=\; \cos\left(\frac{2(n-k)+1}{2n+2}\,\pi\right), \qquad \text{para} \qquad k \;=\; 0,\,1,\,\ldots,\,n,$$

que son los puntos de Chebyshev para el intervalo $[-1,\,1]$.

Ejercicio 45: Realiza los puntos del ejemplo anterior con los puntos de Chebyshev, utiliza n impar para ver la simetría. ¿Qué ves en la convergencia en los extremos y del centro?

8.3.1. El método de Neville y la inversa de un polinimio

Uno de los grandes problemas dados por el término de error presentado en el teorema 8.2 es que no podemos decidir *a priori* el número de nodos para cubrir una cierta precisión con la cual podemos hacer una interpolación; para ello necesitamos tener toda una serie de cálculos que puede resultar más exhaustiva que la propia aproximación. A veces, para aproximar por ejemplo, $f(0.75)$ para una lista de nodos tales como

x	0.0	0.3	0.6	0.9	1.2
$f(x)$	0.7651977	0.6200860	0.4554022	0.2818186	0.11036623

se realizan todas las interpolaciones posibles que contengan los nodos $x_2 = 0.6$ y $x_3 = 0.9$. Hay seis interpolaciones distintas: una lineal, dos cuadráticas, dos cúbicas y una de cuarto orden.

La cantidad de coeficientes que debemos encontrar para calcular todas estas interpolaciones para un único punto, no valen la pena. Además, se puede seguir del desarrollo para la fórmula interpolante de Newton, que ciertos factores lineales ayudan a crear los cuadráticos y así en adelante. Aquí por ejemplo, la información de la interpolación lineal para x_3, x_4 es útil para ambas de las interpolaciones cuadráticas, a saber aquellas que contienen los nodos x_2, x_3, x_4 y x_3, x_4, x_5, respectivamente.

Para simplificar la notación, llamaremos p_{34} al polinomio (lineal) que satisface pasar por x_3 y x_4, así el polinomio $p_{i_0\cdots i_k}$ es de grado k y tal que $p_{i_0\cdots i_k}(x_{i_j}) = f(x_{i_j})$ para $j = 0, 1, \ldots, k$. La construcción siguiente se basa en los polinomios de Lagrange, pues tenemos el siguiente resultado que será útil en el desarrollo.

Afirmación 8.3. *Para cualesquiera i, j distintos, tenemos que si $p(t)$ es el polinomio interpolante por $x_0, x_1, \ldots, x_n$, entonces*

$$p(t) \;=\; \frac{(t - x_j)\, p_{0\cdots(j-1)(j+1)\cdots n}(t) \;-\; (t - x_i)\, p_{0\cdots(i-1)(i+1)\cdots n}(t)}{x_i \;-\; x_j}$$

es satisfecho.

Utilizando recurrentemente esta fórmula, podemos obtener aproximaciones de $f(0.75)$ con varios polinomios, por ejemplo,

$$\begin{aligned}
p_{23}(0.75) &= \frac{(0.75 - 0.6)p_3(0.75) \;-\; (0.75 - 0.9)p_2(0.75)}{0.9 - 0.6} \\
&= \frac{(0.15)0.2818186 \;-\; (-0.15)0.4554022}{0.3} \;=\; 0.3686104, \\
p_{123}(0.75) &= \frac{(0.75 - 0.3)p_{23}(0.75) - (0.75 - 0.9)p_{12}(0.75)}{0.9 \;-\; 0.3} \;=\; 0.369722875, \\
p_{234}(0.75) &= \frac{(0.75 - 0.6)p_{34}(0.75) - (0.75 - 1.2)p_{23}(0.75)}{1.2 \;-\; 0.6} \;=\; 0.36834399625.
\end{aligned}$$

Podríamos de este modo obtener el valor $p_{1234}(0.75)$ a partir de los dos últimos valores. Sin embargo, esto resulta demasiado artesanal. La manera de mejorarlo requiere también cambiar un poco la notación que tenemos.

Observemos que como en la fórmula interpolante de Newton, necesitamos construir iterativamente diferentes valores, que en esta ocasión, son aproximaciones de la función en el valor dado. Para construir el **método de Neville** tomamos Q_{ij} como el polinomio interpolante de grado j en los $j+1$ números x_{i-j}, x_{i-j+1}, ..., x_i evaluado en t. De este modo,

$$Q_{ij} \;=\; p_{(i-j)(i-j+1)\cdots(i-1)(i)}(t),$$

que en la iteración del método se crea a partir de los elementos anteriores

$$Q_{i,j-1} \;=\; p_{(i-j+1)(i-j+2)\cdots(i-1)(i)}(t) \qquad \text{y} \qquad Q_{i-1,j-1} \;=\; p_{(i-j)(i-j+1)\cdots(i-2)(i-1)}(t),$$

bajo la fórmula iterativa

$$Q_{ij} \;=\; \frac{(t-x_{i-j})Q_{i,j-1} \;-\; (t-x_i)Q_{i-1,j-1}}{x_i \;-\; x_{i-j}},$$

para cada $j = 1, 2, \ldots$ e $i = j, j+1, \ldots$. Para dar inicio al algoritmo, tomamos $Q_{i0} = f(x_i)$.

Código: Iteración, interpolación de Neville (PYTHON)

```
Q = np.zeros[n + 1];      %   El vector X es el valor de los
Q[:, 0] = f(X);           %   nodos, en t está el valor donde
D = t - X;                %   aproximaremos f(t).
for i in range(1, n + 1):
    for j in range(1, i):
        Q[i, j] = ( (D(i + 1 - j)*Q(i, j - 1)
                    - D(i)*Q(i - 1, j - 1))
                    /(x(i) - x(i + 1 - j)) )
```

Este algoritmo podría aprovecharse un valor t libre. En PYTHON esto es posible, pero el cálculo simbólico como el que se realiza en MAPLE tiene mejores características. De este modo obtenemos una fórmula para el polinomio interpolante de los valores dados por `X` o, como dijimos antes, por el vector $\mathbf{x} = (x_0, x_1, \ldots, x_n)$. Sin embargo, esto no es del todo recomendable por el número de operaciones a ser realizadas en caso de querer evaluar el polinimio en una lista grande valores, como para hacer un gráfico de éste. En ese caso, aún con el cálculo simbólico resultará de mayor provecho utilizar la fórmula interpolante de Newton y de manera simbólica el cálculo de la división sintética.

En la forma de crear la división sintética hablamos de la "diagonal" de los elementos $f[x_0]$, $f[x_0, x_1]$ y así hasta $f[x_0, x_1, \ldots, x_n]$ que darán los valores c_0, c_1 hasta c_n, respectivamente. En el método de Neville también podemos ordenar estos valores, por ejemplo para los datos arriba:

x_0 :	Q_{00}				
x_1 :	Q_{10}	Q_{11}			
x_2 :	Q_{20}	Q_{21}	Q_{22}		
x_3 :	Q_{30}	Q_{31}	Q_{32}	Q_{33}	
x_4 :	Q_{40}	Q_{41}	Q_{42}	Q_{43}	Q_{44}

Notamos que los valores en rojo son justamente los seis interpolantes mencionados al inicio de esta sección, por ejemplo, Q_{42} es justamente $p_{234}(t)$. (En azul el intervalo original.)

8.3.2. La raíz polinomial de $f(x)$

El método de Neville tiene una aplicación poco usual en la cual podemos usarlo para encontrar la raíz nula de una función $f(x)$. Este, por su puesto, es un método alternativo a la bisección, Muller, Newton, etcétera.

Tenemos una única restricción, debemos tomar valores para $\mathbf{x}$ tales que el comportamiento para estos valores ordenados al evaluarlos en f nos muestren un comportamiento monótono, ya sea creciente o decreciente.

Supongamos que tenemos una modificación de la *función de Runge*, digamos,

$$f(x) = \frac{1}{1+x^2} - \frac{1}{25},$$

que tiene su raíces en $x_\pm = \pm 2\sqrt{6}$, muy próximas de ± 5. Podemos así, dar los valores

x	0.0	1.0	2.0	3.0	4.0	5.0
$f(x)$	0.96000000	0.46000000	0.16000000	0.06000000	0.018823534	−0.00153846

invertir el orden y escribir $y = f(x)$ y $g(y) = x$, es decir, $g(y)$ la inversa de $f(x)$. Un modo de encontrar la raíz es evaluando $g(0) = x^*$ que será $f(x^*) = 0$. Con las aproximaciones polinomiales y el método de Neville esto es una tarea sencilla.

En general, esta manera de invertir la función y encontrar raíces para ella funciona bien cuando $f(x)$ es monótona en x. El ejercicio planteado implícitamente aquí podría reproducir parte del fenómeno de Runge.

8.4. Curvas suaves con dos derivadas continuas

Como hemos visto con el fenómeno de Runge, las oscilaciones en una interpolación pueden ser bastante indeseables, especialmente si tenemos una función que sabemos, por ejemplo, que tiene poca variación. También podría ser el caso, si exageramos al suponer que la función que aproximamos es siempre monótona.

Si vemos los gráficos de PYTHON descubrimos que aunque solo hemos dado una colección finita de puntos, la "curva" es dada de modo continuo. Esto no solamente es agradable visualmente sino que representa quizá el fenómeno deseado. Sin embargo, no tiene continuidad en sus derivadas y la interpolación nos daba una curva suave. En el medio del camino, podríamos buscar curvas que tengan regularidad por segmentos y en las uniones también. Por ejemplo, que tenga segunda derivada continua. Las curvas que nos darán este comportamiento se llaman *splines cúbicos*, pues cada segmento será un polinomio cúbico y en los nodos garantizaremos la continuidad tanto de la función como de sus dos primeras derivadas.[1] (Los gráficos de PYTHON podrían ser considerados como *splines lineales*.)

Es decir, buscamos una función $S : I \to \mathbb{R}$ para un cierto intervalo I tal que $S \in \mathcal{C}^2(I)$, pero que de hecho es $S \in \mathcal{C}^\infty(I \setminus \{x_0, \ldots, x_n\})$; sólo pierde la diferenciabilidad absoluta en puntos aislados que son denotados por los nodos. Típicamente, el intervalo I estará subdividido en intervalos $I_i = [x_{i-1}, x_i]$, así, definimos al ***spline* cúbico S** restringido al intervalo I_i como el polinimio cúbico

$$S_i(t) \;=\; a_i + b_i(t - x_{i-1}) + c_i(t - x_{i-1})^2 + d_i(t - x_{i-1})^3, \qquad \text{para} \qquad x_{i-1} \le t \le x_i,$$

con $i = 1, \ldots, n$. De este modo tenemos 4 coeficientes por intervalo (a_i, b_i, c_i, d_i) y n intervalos, es decir, tenemos $4n$ incógnitas por determinar. Las condiciones de continuidad nos darán las restricciones necesarias para encontrar estos coeficientes.

8.4.1. ¿Cómo encontrar los coeficientes de un *spline* cúbico?

Tenemos las condiciones y restricciones siguientes:

- Para el intervalo $I = [x_0, x_n]$, tenemos los nodos $x_0 < x_1 < \cdots < x_n$ con distancias $h_i = x_i - x_{i-1}$ para $i = 1, \ldots, n$.
- Supongamos que conocemos $f_i = f(x_i)$ para $i = 0, 1, \ldots, n$.
- Además, el *spline* S y sus derivadas S', S'' deben ser continuas, decimos que $S \in \mathcal{C}^2(I)$.

[1] Lo siguiente también se encuentra en [2, pp. 128–138] y [1, pp. 340–344].

- Para simplificar cálculos definimos el siguiente valor como un coeficiente artificial:

$$2c_{n+1} \;=\; S_n''(x_n) \;=\; 2c_n + 6d_n h_n. \tag{8.3}$$

- Vamos a ver que se requieren dos condiciones adicionales para poder identificar (de manera única) un *spline* cúbico. Digamos que las dos condiciones adicionales son las derivadas $f'(x_0)$ y $f'(x_n)$, denotadas por f'_0 y f'_n, respectivamente. Este es el supuesto que realmente define el tipo de *spline* que tendremos.

La condición de interpolación para $f_{i-1} = f(x_{i-1})$ en el intervalo $I_i = [x_{i-1},\, x_i]$ está a la derecha, es decir, $f_{i-1} = S_i(x_{i-1})$ equivale a

$$a_i \;=\; f_{i-1} \qquad \text{para} \qquad i \;=\; 1,\, \ldots,\, n. \tag{8.4}$$

Esto reduce el número a $3n$ incógnitas.

Al final, usaremos también la condición de continuidad al lado izquierdo de cada intervalo, es decir, $f_i = S_i(x_i)$. Las condiciones que piden la continuidad de S' y S'' permiten eliminar otros coeficientes:

- Pedimos que S'' sea continua y usamos la definición (8.3) para $i = n$, *i.e.*,

$$S_i''(x_i) \;=\; S_{i+1}''(x_i) \quad\Longleftrightarrow\quad 2c_i + 6d_i h_i \;=\; 2c_{i+1} \quad\Longleftrightarrow\quad 3d_i h_i \;=\; c_{i+1} - c_i \tag{8.5}$$

  para $i = 1,\, \ldots,\, n$. Así, cada d_i depende solo de dos valores c_i, de modo que hay que encontrar estos c_i.

- Pedimos que S' sea continua. Utilizando la ecuación (8.5) eliminamos d_i:

$$\begin{aligned} S_i'(x_i) \;=\; S_{i+1}'(x_i) \quad &\Longleftrightarrow\quad b_i + 2c_i h_i + 3d_i h_i h_i \;=\; b_{i+1} \\ &\Longleftrightarrow\quad b_i + (c_{i+1} + c_i) h_i \;=\; b_{i+1}, \end{aligned} \tag{8.6}$$

  para $i = 1,\, \ldots,\, n-1$.

- Finalmente, pedimos la continuidad $f_i = S_i(x_i)$ y usamos ecuación (8.5) para eliminar d_i:

$$\begin{aligned} S_i(x_i) \;=\; f_i \quad &\Longleftrightarrow\quad f_{i-1} + b_i h_i + c_i h_i^2 + d_i h_i^3 \;=\; f_i \\ &\Longleftrightarrow\quad f_{i-1} + b_i h_i + \frac{h_i^2}{3}(c_{i+1} + 2c_i) \;=\; f_i \\ &\Longleftrightarrow\quad b_i \;=\; \frac{1}{h_i}(f_i - f_{i-1}) - \frac{h_i}{3}(c_{i+1} + 2c_i), \end{aligned} \tag{8.7}$$

  para $i = 1,\, \ldots,\, n$, con lo cual cada b_i depende solamente de valores f_i y c_i.

Por un momento, supongamos que conocemos los parámetros $\{c_i\}_{i=0}^n$. Entonces, las relaciones (8.5) y (8.7) permiten calcular los $\{d_i\}_{i=0}^n$ y $\{b_i\}_{i=0}^n$. Concluimos que en este caso conocemos el *spline*, pues conocemos cada uno de sus coeficientes. En este argumento se ve que no usamos las relaciones (8.6). De hecho, estas relaciones se usarán para construir un sistema tridiagonal para los $\{c_i\}_{i=0}^n$.

Primero, usamos la ecuación (8.7) para eliminar b_i, b_{i+1} de la ecuación (8.6). Con eso llegamos a $n-1$ ecuaciones para $i = 1, \ldots, n-1$:

$$\begin{aligned}
\tfrac{1}{h_i}(f_i - f_{i-1}) - \tfrac{h_i}{3}(c_{i+1} + 2c_i) + (c_{i+1} + c_i)h_i &= \tfrac{1}{h_{i+1}}(f_{i+1} - f_i) - \tfrac{h_{i+1}}{3}(c_{i+2} + 2c_{i+1}) \\
&\iff \\
\tfrac{h_{i+1}}{3}(c_{i+2} + 2c_{i+1}) + \tfrac{h_i}{3}(2c_{i+1} + c_i) &= \tfrac{1}{h_{i+1}}(f_{i+1} - f_i) - \tfrac{1}{h_i}(f_i - f_{i-1}) \\
&\iff \\
h_i c_i + 2(h_i + h_{i+1})c_{i+1} + h_{i+1}c_{i+2} &= \tfrac{3}{h_{i+1}}(f_{i+1} - f_i) - \tfrac{3}{h_i}(f_i - f_{i-1}) =: DF_{i+1}.
\end{aligned}$$

Hay que notar que hemos definido el lado derecho como distintos DF_i.

Notamos que esta última ecuación es válida para $i = 1, 2, \ldots, n-1$, es decir, nos faltan 2 ecuaciones (pues $\{c_i\}_{i=0}^n$ tiene $n+1$ elementos). Con dos condiciones adicionales podemos obtener dos ecuaciones más. Por ejemplo, si conocemos $f_0' = f'(x_0)$, entonces pedimos que $f_0' = S_1'(x_0)$. Eso equivale a $b_1 = f_0'$ y con la ecuación (8.7) obtenemos:

$$\frac{h_1}{3}(c_2 + 2c_1) = \frac{1}{h_1}(f_1 - f_0) - f_0' \iff 2h_1c_1 + h_1c_2 = \frac{3}{h_1}(f_1 - f_0) - 3f_0'. \tag{8.8}$$

Por otro lado, si conocemos $f_n' = f'(x_n)$, entonces pedimos que $f_n' = S_n'(x_n)$. Esto permite extender la ecuación (8.6) para $i = n$. Después, eliminando b_n con la ecuación (8.7), llegamos a

$$\begin{aligned}
\frac{1}{h_n}(f_n - f_{n-1}) - \frac{h_n}{3}(c_{n+1} + 2c_n) + (c_{n+1} + c_n)h_n &= f_n' \\
\iff \quad \frac{h_n}{3}(2c_{n+1} + c_n) &= f_n' - \frac{1}{h_n}(f_n - f_{n-1}) \\
\iff \quad h_n c_n + 2h_n c_{n+1} &= 3f_n' - \frac{3}{h_n}(f_n - f_{n-1}).
\end{aligned} \tag{8.9}$$

Por último, juntando los tres tipos de ecuaciones que hemos generado en (8.8), la definición de DF_i y en (8.9), tenemos un sistema lineal con una matriz tridiagonal. Estos sistemas, como ya podemos intuir tienen una resolución sencilla, especialmente cuando son diagonalmente dominantes.

Este sistemas está dado por

$$\begin{pmatrix} 2h_1 & h_1 & & & & \\ h_1 & 2(h_1+h_2) & h_2 & & & \\ & h_2 & 2(h_2+h_3) & h_3 & & \\ & \ddots & \ddots & \ddots & & \\ & & h_{n-1} & 2(h_{n-1}+h_n) & h_n & \\ & & & h_n & 2h_n \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \\ c_{n+1} \end{pmatrix} = \begin{pmatrix} \frac{3}{h_1}(f_1 - f_0) - 3f_0' \\ DF_2 \\ DF_3 \\ \vdots \\ DF_n \\ 3f_n' - \frac{3}{h_n}(f_n - f_{n-1}) \end{pmatrix}.$$

Así, una vez resuelto este sistema, se evalúan directamente los valores b_i, d_i de las identidades (8.5) y (8.7). Recordando que tenemos los valores $a_i = f_{i-1}$, tenemos todos los coeficientes del *spline* $S : I \to \mathbb{R}$.

Ejemplo: Consideremos ahora los puntos dados en la siguiente tabla:

x	-1	0	1
$f(x)$	0	0	4

Encontremos el *spline* cúbico para este caso, completando con $f'(-1) = 0$ y $f'(1) = 8$.

Notamos primero que necesitamos dos funciones cúbicas,

$$\begin{aligned} S_1(t) &= a_1 + b_1(t+1) + c_1(t+1)^2 + d_1(t+1)^3, \quad \text{para} \quad t \in [-1, 0], \\ S_2(t) &= a_2 + b_2 t + c_2 t^2 + d_2 t^3, \quad \text{para} \quad t \in [0, 1], \end{aligned}$$

de donde tenemos $h_1 = h_2 = 1$ y directamente de los valores de la tabla vemos que $a_1 = a_2 = 0$. Para encontrar los otros coeficientes de los polinomios cúbicos recordamos que tenmos que construir los c_i y a partir de ellos los b_i y d_i. Vamos a construir el sistema $A\mathbf{x} = \mathbf{b}$ donde

$$A = \begin{pmatrix} 2 & 1 & \\ 1 & 4 & 1 \\ & 1 & 2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 3\big(f(0) - f(-1)\big) - 3f'(-1) \\ DF_2 = 3\big(f(1) - 2f(0) - f(-1)\big) \\ 3f'(1) - 3\big(f(1) - f(0)\big) \end{pmatrix} = \begin{pmatrix} 0 \\ 12 \\ 12 \end{pmatrix}.$$

Este sistema tiene por solución $\mathbf{x} = (-1, 2, 5)^\top$, que recordamos nos remite a

$$c_1 = -1, \qquad c_2 = 2, \qquad c_3 = 5;$$

de (8.6) y (8.5) obtenemos respectivamente

$$b_1 = 0, \quad b_2 = 1 \quad \text{y} \quad d_1 = 1, \quad d_2 = 1.$$

Así, finalmente tenemos

$$\begin{aligned} S_1(t) &= 0 + 0 - (t+1)^2 + (t+1)^3 = t(t+1)^2, \\ S_2(t) &= 0 + t + 2t^2 + t^3 = t(1+2t+t^2) = t(t+1)^2. \end{aligned}$$

Es decir, los puntos pertenecen a un único polinomio de grado 3. Estos valores son escogidos así para ver la relación entre las partes. Por ejemplo, de los elementos dados, tenemos que $f(-1) = f(0) = 0$ con $f'(-1) = 0$ que se antoja como un polinimio nulo, sin embargo, la conexión en $x = 0$ nos da la oscilación en el intervalo $[-1, 0]$.

Discusión: En el ejemplo anterior vemos que la relación entre los distintos *splines* es importante en el sentido que se comunican entre ellos. Otro punto destacable es la forma de la matriz A que para n grande con nodo equiespaciados es de la forma

$$A = \begin{pmatrix} 2 & 1 & & & & & & \\ 1 & 4 & 1 & & & & & \\ & 1 & 4 & 1 & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & 1 & 4 & 1 & & \\ & & & & 1 & 4 & 1 & \\ & & & & & 1 & 2 & \end{pmatrix}$$

o un múltiplo h de ésta. Notamos la simetría de la matriz y es fácil mostrar que es una matriz definida positiva, por lo que usar la descomposición de Cholesky, Sec. 7.2, es una opción oportuna que encuentra la solución de $A\mathbf{x} = \mathbf{b}$ en tiempo lineal.

Ejercicio 46: Muestra que la matriz A de la discusión anterior es realmente simétrica y definida positiva.

Capítulo 9

Aproximación de ecuaciones diferenciales

Este capítulo de las notas trata sobre las ecucaciones diferenciales y algunas formas para integralas numéricamente. Estos temas pueden ser muy extensos y solo tendremos una visión muy breve de cómo atacarlos. El dominio de las ecuaciones diferenciales tiene muy variadas aplicaciones que radican en la física, la química, la biología y la economía, entre otras. En este caso buscaremos funciones de una o varias variables que satisfagan una ecuación o un sistema de ecuaciones en las cuales la función incógnita aparece junto con sus derivadas. Dado que algunos métodos teóricos se refieren en resolver la ecuación bajo el signo de la integral, en ocaciones a las soluciones se les llama *primeras integrales* o la integración de la ecuación.

En el caso de las ecuaciones diferenciales ordinarias (EDO) solamente veremos problemas de valor inicial (PVI), quedarán de fuera los problemas de contorno. Los problemas de valores finales son análogos a los PVI y sólamente tendremos que cambiar la flecha de la dirección del tiempo. Por otro lado, las EDO que son definidas con derivadas de orden n, pueden ser transformadas rápidamente al modo en el que se atacan en Sistemas Dinámicos; comunmente se cambia el paradigma de una única ecuación a un sistema de n ecuaciones. La ventaja de esto es tener derivadas de a lo más grado uno. Numéricamente implica también en no implementar un sinnúmero de casos particulares sino una forma robusta de atacar un sinnúmero de problemas.

En el caso de las ecuaciones diferenciales parciales (EDP), los temas son todavía más extensos; la idea de transformar una EDP de grado n en un sistema, no es tan sencillo. Además, existen varias metodologías para atacar problemas, para citar algunos se encuentran los *elementos finitos*, los *métodos espectrales*, el *front tracking*, entre muchos otros. Aprovecharemos las diferencias finitas del Capítulo 4 para construir el método de *diferencias finitias para EDP*. El mundo clásico de las EDP se divide en ecuaciones elípticas, hiperbólicas y parabólicas. Sólo veremos un ejemplo de este último tipo con la idea de atacar la afamada ecuación de Black-Scholes.

9.1. Ecuaciones diferenciales ordinarias

Típicamente se define una EDO de grado n como la relación dada por su *forma general*

$$G\left(t,\, x,\, \dot{x},\, \ldots,\, x^{(n)}\right) \;=\; 0, \tag{9.1}$$

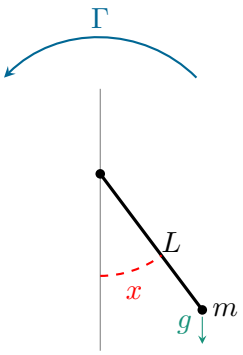
donde para el tiempo t en el intervalo $I \subset \mathbb{R}$, está definida la función $G : \mathbb{R}^{n+2} \to \mathbb{R}$ con coeficientes que pueden depender del tiempo para cada una de las variables t, $x(t)$, $\ldots$, $x^{(n)}(t)$ y este último término no debe ser nulo. Nuestra incógnita es la función $x : I \to \mathbb{R}$ y buscamos que satisfaga la relación planteada en (9.1).

Esta manera resulta confusa, pero práctica para sus definiciones. Por ejemplo, el grado de la EDO es n, es *lineal* si la función es lineal en todos los términos que dependen de x y sus derivadas; pueden estar multiplicadas por funciones no lineales en t. Además, es *autónoma* si G no depende explícitamente de t o *no autónoma* en caso contrario. Es *periódica* y de perido T si $G(0,\, x,\, \dot{x},\, \ldots,\, x^{(n)}) = G(T,\, x,\, \dot{x},\, \ldots,\, x^{(n)})$, pero es distinta para toda $t \in (0,\, T)$.

Una manera más práctica de trabajar con una EDO está dada por la *forma normal* en la que se destaca la derivada de orden mayor, así la relación (9.1) puede reescribirse como

$$x^{(n)} \;=\; g\left(t,\, x,\, \dot{x},\, \ldots,\, x^{(n-1)}\right),$$

donde todas las definiciones anteriores se aplican.



Como ejemplo, podemos comenzar con la ecuación completa del péndulo, es decir, con fricción y forzante. En este caso, la ecuación puede ser escrita como

$$mL^2\ddot{x} \,+\, b\dot{x} \,+\, mgL\,\operatorname{sen} x \,-\, \Gamma \;=\; 0, \tag{9.2}$$

donde el lado izquierdo se puede escribir como la función $G(t,\, x,\, \dot{x},\, \ddot{x})$, m representa la masa del péndulo a distancia L, g es la constante de la gravedad, b es un factor de fricción y Γ una torca o forzante externo. En este caso, $x(t)$ representa el ángulo de la cuerda contra el eje vertical.

Ejercicio 47: Clasifica la ecuación del péndulo en términos de su grado, autonomía, etcétera. Esta no es una ecuación periódica, ¿por qué?

Ahora vamos transformar la ecuación del péndulo (9.2) en un sistema. Tomamos $x_1(t) = x(t)$ y $x_2(t) = \dot{x}(t)$. De este modo, tenemos que $\dot{x}_2(t) = \ddot{x}_1(t) = \ddot{x}(t)$. Despejando (9.2) tenemos el sistema

$$\begin{cases} \dot{x}_1 &=& x_2(t) \\ \dot{x}_2 &=& -\frac{b}{mL^2}x_1(t) \,-\, \frac{g}{L} \,+\, \Gamma \end{cases},$$

que es un sistema autónomo de primer orden. De hecho, cualquier sistema no autónomo puede transformarse en uno autónomo al tomar $x_0(t) = t$ o $\dot{x}_0 = 1$.

Del ejemplo anterior, podemos ver que toda ecuación puede llevarse a un sistema $\dot{\mathbf{x}}(t) = F(\mathbf{x})$ con $\mathbf{x} : I \to \mathbb{R}^n$, al denotar también una condición inicial, digamos $\mathbf{x}(t_0) = \mathbf{x}_0$, obtenemos un PVI

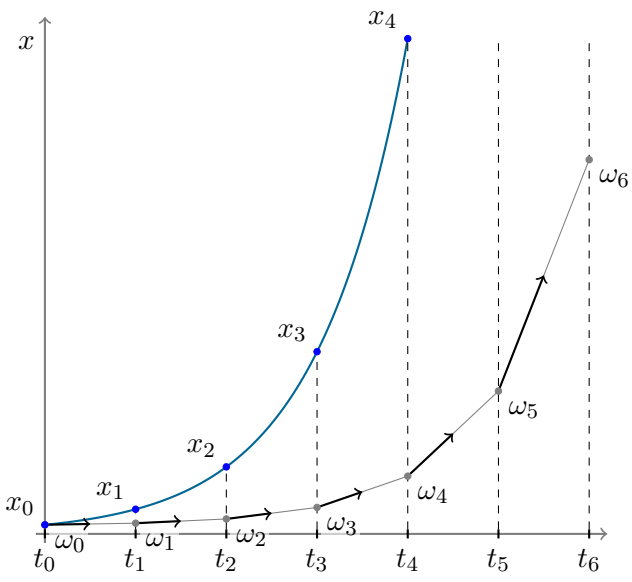
$$\begin{cases} \dot{\mathbf{x}} &= F(\mathbf{x}), \qquad t > t_0 \\ \mathbf{x}(t_0) &= \mathbf{x}_0 \end{cases}. \tag{9.3}$$

Normalmente haremos una traslación en el tiempo y tomaremos $t_0 = 0$. A la función $F : \mathbb{R}^n \to \mathbb{R}^n$ se le llama el *flujo* de la EDO (o del sistema en este caso).

9.1.1. El método de Euler

Es curioso que el nombre de Euler aparezca en un tema moderno como el de las ecuaciones diferenciales numéricas. El desarrollo aquí se basa en sus ideas para entender el comportamiento de una EDO. El sistema (9.3) puede ser visto como una *ecuación escalar* cuando $\mathbf{x}(t) = x(t) \in \mathbb{R}$; usamos también $f(x)$ en lugar de $F(\mathbf{x})$. En este caso es más sencillo ver que si $\phi(t)$ es una función que satisface la condición inicial $\phi(0) = x_0$ y $\dot{\phi}(t) = f\big(\phi(t)\big)$, entonces es solución del PVI. Además, por un lado, notamos que $\dot{\phi}(t)$ es la pendiente de la recta tangente, por el otro, $f(\phi)$ puede ser evaluado en distintos valores de ϕ, así, colocando los vectores $(1,\ f(\phi))$ para cada punto $(t,\ \phi)$ en el plano, obtenemos un *campo vectorial* al cual debe ser tangente la solución $\phi(t)$.

En el gráfico al lado, está la modificación de esta idea llevada al campo de la computación científica. Si nuestra solución $\phi(t)$ es tangente al campo vectorial, entonces, podemos colocarnos en la condición inicial $(t_0,\ x_0)$, buscar el vector tangente $\big(1,\ f(x_0)\big)$, dar un pequeño paso de tamaño h y repetir la operación cuando lleguemos al siguiente punto.



Para no confundir la solución exacta de la *solución discreta*, nuestra aproximación numérica, tomaremos los pasos de tiempo

$$t_0 < t_1 := t_0 + h < \cdots < t_k := t_0 + kh < \cdots$$

o mallado con espaciamiento h. Así, la solución pasa por los puntos $(t_k,\ x_k)$ para $x_k := \phi(t_k)$. En el gráfico esta curva está en azul. Las soluciones numéricas pasarán por los puntos $(t_k,\ \omega_k)$, donde tendremos que explicar la manera de encontrar los valores ω_k; la curva a pedazos en la figura.

Cuando no tenemos los valores exactos x_k, por lo menos podemos comenzar con la condición inicial, es decir, $\omega_0 = x_0$. Ahora, ω_1 se puede tomar como el valor que satisface en el gráfico que $(t_1, \omega_1) = (t_0, \omega_0) + h\big(1, f(\omega_0)\big)$, donde nos interesa solamente la segunda entrada; la primera se satisface por la definición del mallado temporal. Haciendo esto de modo iterativo, nos lleva al **método de Euler (explícito)**, tenemos la regla de recursión

$$\begin{cases} \omega_0 &= x_0 \\ \omega_{k+1} &= \omega_k + h\, f(\omega_k), \qquad \text{para} \qquad k \;=\; 0,\, 1,\, \ldots \end{cases} \tag{9.4}$$

y donde, como antes, $t_k = t_0 + hk$.

De modo semejante, podemos aproximar la tangente no por el punto en que estamos, (t_k, ω_k) y por lo tanto $f(\omega_k)$, sino por el punto al que llegaremos, (t_{k+1}, ω_{k+1}) y por lo tanto $f(\omega_{k+1})$. Usando esta idea, obtenemos el **método de Euler (implícito)** dado por la regla de recursión

$$\begin{cases} \omega_0 &= x_0 \\ \omega_{k+1} &= \omega_k + h\, f(\omega_{k+1}), \qquad \text{para} \qquad k \;=\; 0,\, 1,\, \ldots \end{cases} \tag{9.5}$$

donde ahora tenemos que despejar la útima igualdad para tener una fórmula en la cual a partir del punto al tiempo t_k tengamos la altura en el tiempo t_{k+1}.

Ejemplo: Para ver la diferencias, podemos resolver por ambos métodos de Euler el PVI

$$\begin{cases} \dot{x} &= \alpha\, x, \\ x(0) &= x_0, \end{cases} \tag{9.6}$$

para $t > 0$ y pensando en las constantes α y x_0 como parámetros. La ventaja de esta ecuación es que conocemos su solución explícitamente, $\phi(t) = x_0\, e^{\alpha t}$. Muéstralo.

Tomemos de modo analítico la solución propuesta por el método de Euler explícito (9.4),

$$\omega_0 \;=\; x_0,\; \omega_1 \;=\; (1+h\alpha)\omega_0,\; \omega_2 \;=\; (1+h\alpha)\omega_1 \;=\; (1+h\alpha)^2\omega_0,\; \cdots,\; \omega_k \;=\; (1+h\alpha)^k\omega_0,\; \cdots.$$

Ahora, tomando el límite cuando $h \to 0$ pero manteniendo el valor $t = hn$, es decir, $n = t/h$ para el tiempo fijo, tenemos

$$\lim_{n\to\infty} \omega_n \;=\; \lim_{n\to\infty} (1 + h\alpha)^n \omega_0 \;=\; \lim_{n\to\infty} \left(1 + \frac{\alpha t}{n}\right)^n x_0 \;=\; x_0\, e^{\alpha t}$$

como en la solución analítica.

De modo semejante, tomando ahora la solución propuesta por el método de Euler implícito

(9.5), tenemos que las interaciones nos llevan a

$$\omega_1 \;=\; \omega_0 + h(\alpha\omega_1) \quad\Longrightarrow\quad (1-h\alpha)\omega_1 \;=\; \omega_0 \quad\Longrightarrow\quad \omega_1 \;=\; (1-h\alpha)^{-1}\omega_0,$$

que al llevarlo de modo recursivo, tenemos la igualdad

$$\omega_k \;=\; (1-h\alpha)^{-1}\omega_{k-1} \;=\; \cdots \;=\; (1-h\alpha)^{-k}\omega_0 \;=\; (1-h\alpha)^{-k}x_0.$$

Fijando el valor de t y tomando el límite cuando $h \to 0$, tenemos

$$\omega_0 \;=\; \lim_{n\to\infty}(1-h\alpha)^n\omega_n \;=\; \lim_{n\to\infty}(1-h\alpha)^n\omega_n \;=\; \lim_{n\to\infty}\left(1+\frac{(-\alpha)t}{n}\right)^n\omega_n \;=\; e^{-\alpha t}\lim_{n\to\infty}\omega_n,$$

que despejando, nos lleva nuevamente a la relación deseada.

Ejercicio 48: Implementa los métodos de Euler y grafica sus soluciones para el PVI del ejemplo con los valores de $\alpha = 0.5$ y $x_0 = 0.1$. Compara los resulados numéricos hasta el tiempo $t = 2$ contra la solución analítica, usa $h = 1, 1/2, 1/4 \ldots, 1/2^5$, por ejemplo. ¿Qué observas? ¿Podrías arriesgar a decir el tipo de convergencia de cada método? ¿Por dónde se aproximan? Explica este fenómeno.

Ejercicio 49: Toma los promedios de cada paso h en los dos métodos anteriores, ¿qué observas? Implementa el código para el método del trapecio:

$$\begin{cases} \omega_0 &= x_0 \\ \omega_{k+1} &= \omega_k \;+\; \frac{h}{2}\big(f(\omega_k) \;+\; f(\omega_{k+1})\big), \qquad \text{para} \qquad k \;=\; 0, 1, \ldots \end{cases}$$

que resulta de promediar los dos métodos de Euler. ¿Te arriesgarías a decir el orden de convergencia?

Como último comentario, necesitamos saber cuándo un problema de valor inicial como (9.3) tiene solución y cuándo ésta es única. Una versión del resultado más fuerte que tenemos en esta dirección es el siguiente.

Teorema 9.1 (Existencia y unicidad de EDO de primer orden)**.** *Sea $F(\mathbf{x})$ Lipschitz continua para $\|\mathbf{x} - \mathbf{x}_0\| \leq \alpha$. Entonces el PVI*

$$\dot{\mathbf{x}} \;=\; F(\mathbf{x}), \qquad \mathbf{x}(t_0) \;=\; \mathbf{x}_0$$

tiene solución única en un intervalo abierto $I \in (t_0 - h,\, t_0 + h)$. (Además: Si la función es Lipschitz continua en todo $\mathbb{R}^n$, entonces $I = \mathbb{R}$.)

En el enunciado del teorema hablamos de la continuidad tipo Lipschitz. Basta recordar que una función de clase $\mathcal{C}^1(\Omega)$ es Lipschitz en el mismo dominio Ω. Normalmete trabajaremos con flujos cuya primera derivada es continua en todo $\mathbb{R}^n$; lo cual es más fácil de mostrar.

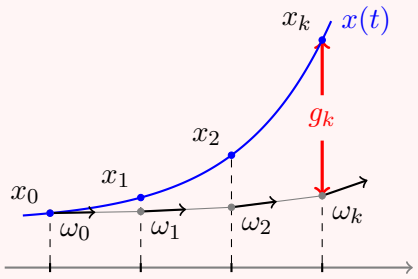
9.1.2. Sobre el error de truncamiento local y global

Es importante notar que a cada paso que damos con un método numérico, estamos produciendo un error al ir por una tangente (por ejemplo) y no por la curva real. A este tipo de error se le llama el *error de truncamiento local*. Sin embargo, lo importante es tener una cota de error cuado hemos realizado varios pasos y llegado al tiempo que queremos, para ello tenemos un error que se va acumulando con es llamado *error de truncamiento global*. Podemos dar definiciones rigurosas al respecto.

Definición IX.1: Definimos el **error de truncamiento global** como el valor

$$g_k \;=\; |x_k \,-\, \omega_k|$$

para $x_k = x(t_k)$ la solución exacta y ω_k su aproximación numérica en t_k.



Definición IX.2: Definimos el **error de truncamiento local** como el valor

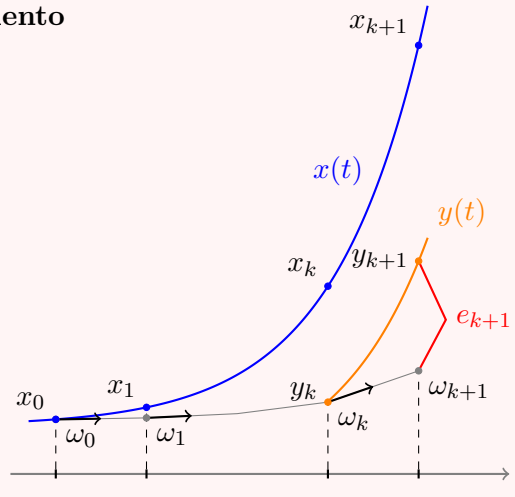
$$e_{k+1} \;=\; |y_{k+1} \,-\, \omega_{k+1}|$$

para $y_{k+1} = y(t_{k+1})$ la solución exacta del PVI

$$\dot{x} \;=\; f(t), \qquad x(t_k) \;=\; \omega_k,$$

y ω_k su aproximación numérica en t_k del PVI

$$\dot{x} \;=\; f(t), \qquad x(t_0) \;=\; x_0.$$



Es importante notar que las definiciones arriba ayudan a entender que el error de truncamiento global no es simplemente la suma de los locales, es decir, en general

$$g_k \;\neq\; \sum_{i=1}^{k} e_i; \quad \text{típicamente} \quad g_2 \;>\; e_1 + e_2.$$

Con el Teorema de estabilidad para soluciones para EDO y el Lema de Gronwall se puede probar que si el error de truncamiento local puede ser acotado por una constante C y una potencia del paso h, digamos como $e_i \leq Ch^{m+1}$, entonces, se satisface que

$$g_k \;\leq\; \frac{Ch^m}{L}\big(e^{Lt_k}-1\big) \;=\; \tilde{C}h^m,$$

donde L es una constante de Lipschitz dependiente de la función de flujo de la EDO y t_k es el tiempo de nuestra iteración. Lo importante es la relación entre los órdenes de los errores de truncamiento.

Afirmación: Considera un método numérico de un paso con un error de truncamiento local de $\mathcal{O}(h^{m+1})$, entonces el error de truncamiento global es $\mathcal{O}(h^m)$.

Ejemplo: Gracias a la afirmación anterior, vemos que basta encontrar el error de truncamiento local de un método de un paso, como es el caso de los métodos de Euler. Supongamos que la solución al paso k es correcta y veamos cómo se aleja la siguiente iteración de la realidad. Es decir, consideramos $\omega_k = x(t_k)$ para $x : \mathbb{R} \to \mathbb{R}$ la solución al PVI.

Suponiendo que la solución es de clase $\mathcal{C}^2(\mathbb{R})$, tenemos por la serie de Taylor,

$$\begin{aligned} x(t_k+h) \;&=\; x(t_k) + h\dot{x}(t_k) + \frac{h^2}{2}\ddot{x}(\tau) \\ \Longrightarrow \quad & x(t_{k+1}) - [x(t_k) + hf(x_k)] \;=\; \frac{h^2}{2}\ddot{x}(\tau) \\ \Longrightarrow \quad & x(t_{k+1}) - \omega_{k+1} \;=\; \frac{h^2}{2}\ddot{x}(\tau), \end{aligned}$$

donde $\tau \in [t_k,\, t_k+h]$. Ahora, tomando M_2 como el máximo de las segundas derivadas en el intervalo, es decir, $|\ddot{x}(t)| \leq M_2$ en $[t_k,\, t_{k+1}]$, tenemos que $e_{k+1} \leq M_2h^2/2$. Haciendo uso de los pasos anteriores, tenemos así que $g_k \leq Ch$ para alguna constante C; ésta depende del tiempo t_k, la constante de Lipschitz L de la función de flujo en cuestión y de la constante M_2 definida arriba, de hecho, $C = M_2(\exp(Lt_k)-1)/(2L)$.

Ejercicio 50: Ahora realiza un análisis semejante al aterior para ver la convergencia del método de Euler implícito.

Ejercicio 51: Revisa la convergencia numérica para el PVI (9.6) al tomar $\alpha = 0.5,\, 1$ y con $x_0 = 0.1,\, 1$ hasta el tiempo $t = 2$.

9.1.3. El método de Runge-Kutta

Una manera de generar métodos de órdenes cada vez más precisos son los llamados métodos de Runge-Kutta. Estos algoritmos tienen precisiones distintas y el más usado de ellos es el `RK4` que veremos más adelante. El método de Euler es un `RK1` y el trapecio aunque es un método de orden 2 no es el `RK2`; puedes ver que el número tiene que ver con la convergencia del error de truncamiento global.

Para tener una noción simple de cómo son generados los métodos Runge-Kutta, veamos que a diferencia del trapecio que es el promedio de `EuE` y `EuI`, podríamos haber buscado que la aproximación lineal se tomara con el valor en el punto medio, es decir, al tiempo $t_{k+1/2} = t_k + h/2$. Este valor es un estado que será utilizado para aproximar la derivada. El **método del punto medio** es:

$$\begin{cases} \omega_0 &= x_0 \\ \omega_{k+1} &= \omega_k + hf\left(t_k + \frac{h}{2}, \omega_{k+\frac{1}{2}}\right), \quad \text{con } k = 1, 2, \ldots \end{cases},$$

donde nos proponemos establecer cómo evaluar $\omega_{k+1/2}$. Se puede hacer de modo explícito e implícito, lo más sencillo es tomar el método de Euler para esto, por ejemplo,

$$\omega_{k+1/2} = \omega_k + \frac{h}{2} f(\omega_k).$$

Típicamente a esta evaluación se le llama un estado s. Así, dados los pesos correctos, la familia de los métodos Runge-Kutta, en particular `RKs` con `s` estados se escribe como

$$\omega_{k+1} = \omega_k + h \sum_{i=1}^{\mathtt{s}} b_i s_i,$$

donde cada estado s_i es una aproximación intermediaria entre t_k y t_{k+1}, los pesos b_i dependen de las localidades de estos estados y h es el tamaño habitual del paso.

Ejercicio 52: Muestra que el método del punto medio tiene un error de truncamiento local $\mathcal{O}(h^3)$, lo que lo transforma en un método de orden cuadrático; `RK2`.

Un ejemplo muy importante y popular es el *método RK4* por su alta precisión de $\mathcal{O}(h^4)$. Esto quiere decir que si para un h fijo el error de truncamiento global en un tiempo T fijo es menor que Ch^4, al refinar la malla sucesivamente por diez esperamos un error menor que $C(h/10)^4 = Ch^4/10^4 = Ch^4/1000$, $Ch^4/10^8$, $Ch^4/10^{12}$, ...; vemos que en pocos refinamientos llegaremos al épsilon de la máquina.

Tal vez conseguir los coeficientes para un `RKs` no sea difícil, pero saber cuál de todas la variantes es la más eficiente es lo que realmente lleva a una demostración muy pesada, es por ello que presentamos el método sin pruebas.

Runge-Kutta 4: Tomamos

$$\omega_{k+1} = \omega_k + \frac{h}{6}(s_1 + 2s_2 + 2s_3 + s_4),$$

donde

$$\begin{aligned} s_1 &= f(t_k, \omega_k) && \text{(Pendiente EuE)} \\ s_2 &= f\left(t_k + \frac{h}{2}, \omega_k + \frac{h}{2}s_1\right) && \text{(Punto medio)} \\ s_3 &= f\left(t_k + \frac{h}{2}, \omega_k + \frac{h}{2}s_2\right) && \text{(Punto medio mejorado)} \\ s_4 &= f(t_k + h, \omega_k + hs_3) && \text{(EuE/EuI mejorado)} \end{aligned}$$

La expresión $\frac{h}{6}(s_1 + 2s_2 + 2s_3 + s_4)$ es una estimación mejorada de la pendiente en el intervalo $[t_k, t_{k+1}]$. (De algún modo el $h/6$ recuerda la *Regla de Simpson* en la integración numérica.)

9.2. Ecucaciones diferenciales parciales

Como hemos anticipado, el estudio de las ecuaciones diferenciales parciales (EDP) es un tema muy extenso. Es por ello que en esta breve introducción a cómo resolver problemas de modo numérico, vía el método de diferencias finitas, miramos solo la ecuación diferencial parcial de segundo orden del tipo parabólica clásica, la *ecuación de difusión* o *ecuación del calor*.

Al resolver un sistema de EDP buscamos encontrar la función $u : \mathbb{R}^n \to \mathbb{R}$ que satisfaga tanto la ecuación diferencial específica como sus condiciones iniciales y de fronterea, en concreto las *condiciones de contorno*. Un sistema para describir el comportamiento del calor en una barra de metal finita puede ser dada por la **ecuación del calor** con condiciones de contorno

$$\begin{cases} u_t = Du_{xx}, & x \in (0, L),\ t > 0, \\ u(x, 0) = u_0(x), & x \in [0, L], \\ u(0, t) = f(t), & t \geq 0, \\ u_x(L, t) = g(t), & t \geq 0, \end{cases} \tag{9.7}$$

donde lo más complicado es explicar la notación. Como hemos dicho $u(x, t)$ es la incógnita que tenemos, en este caso, una función en dos dimensiones cuyos valores en los reales representan la temperatura en la barra de longitud L y por ello para las posiciones $x \in [0, L]$ y al tiempo $t \geq 0$. Los subíndices se utilizan para representar las derivadas parciales, por ejemplo,

$$u_t = \frac{\partial u}{\partial t}, \quad u_x = \frac{\partial u}{\partial x} \quad \text{y} \quad u_{xx} = \frac{\partial^2 u}{\partial x^2}.$$

Además, tenemos que denotar cómo es la temperatura al tiempo inicial descrita en la ecuación (9.7b) con $u_0(x)$ el perfil. Las condiciones (9.7c) y (9.7d) representan cómo se comporta el flujo de calor en la frontera de la barra, ésta está aislada y su temperatura sólo puede ser controlada

en $x = 0$ y $x = L$. Hemos colocado dos tipos de condiciones clásicas, a la izquierda tenemos una *condición de Dirichlet* en la cual $f(t)$ dictamina el valor de la temperatura, digamos porque calentamos este punto o lo enfriamos. A la derecha tenemos una *condición de Neumann* que dice el flujo $g(t)$ del calor en esta frontera, si $g(t) \equiv 0$ es como si la barra estuviera aislada en este punto también y, por tanto, no hay flujo térmico. (Cuando f o g son nulas para todo tiempo, se llaman respectivamente condiciones de Dirichlet y Neumann homogéneas.) Así, la ecuación (9.7a) es la EDP y las ecuaciones (9.7b)-(9.7d) las condiciones de contorno. Una solución de este sistema debe satisfacer tanto la EDP como sus condiciones de contorno.

Ejemplo: Es importante ver cómo se comporta una solución. Hagamos un ejemplo sencillo al considerar en (9.7),

$$u_0(x) \;=\; \operatorname{sen} x, \quad f(t) \;\equiv\; 0 \;\equiv\; g(x),$$

con $L = \pi/2$ y D libre. Vemos que tomando la condición inicial, tenemos

$$u(x = 0,\, 0) \;=\; \operatorname{sen} 0 \;=\; 0 \qquad \text{y} \qquad u_x(x = \pi/2,\, 0) \;=\; u_0'(\pi/2) \;=\; \cos\frac{\pi}{2} \;=\; 0,$$

con lo cual vemos cierta compatibilidad al tomar las condiciones en la frontera (9.7c) y (9.7d) en el tiempo $t = 0$, respectivamente,

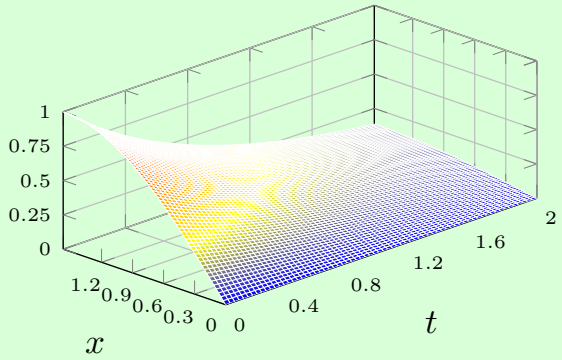
$$u(0,\, t = 0) \;=\; 0 \qquad \text{y} \qquad u_x(\pi/2,\, t = 0) \;=\; 0.$$

Así, decimos que se satisfacen las *condiciones de compatibilidad*.

En este curso, la idea no es resolver de modo analítico las EDP, demos entonces la solución exacta $u(x,\, t) = e^{-Dt} \operatorname{sen} x$ y veamos que es solución. Para ello, debe de satisfacer todas las condiciones del sistema (9.7), a saber,

$$u_t \;=\; -De^{-Dt} \operatorname{sen} x \;\text{ y }\; u_{xx} \;=\; -e^{-Dt} \operatorname{sen} x, \qquad \text{luego} \qquad u_t \;=\; Du_{xx}$$

se satisface para todo $x \in (0,\, \pi/2)$ y $t > 0$. Dado que $e^{-D\cdot 0} = 1$, la codición inicial se satisface, del mismo modo, como $\operatorname{sen} 0 = 0$, la condición de frontera (9.7c) también se satisface. Finalmente, como $u_x(x,\, t) = e^{-Dt} \cos x$ y $\cos \pi/2 = 0$, la condición (9.7d) también se satisface, mostrando así, que nuestra solución es realmente solución del sistema.



Ahora observa que para $t \to \infty$ tenemos $u(x,\, t) \to 0$; la temperatura tiende a uniformizarse y, en general, la solución asintótica es una recta que satisface las condiciones de frontera.

En esta sección, primero desarrollaremos dos esquemas de diferencias finitas para esta ecuación, a saber, el método de Euler implícito en el tiempo y diferencias centradas en el espacio o *backward Euler* y el esquema de seis puntos conocido como *Crank-Nicolson*. Con esto, podremos dar lugar a métodos para resolver una ecuación de la misma familia con aplicaciones en finanzas, la llamada *ecuación de Black-Scholes*. En el caso de esta última ecuación usaremos el *paso de tiempo de Rannacher*. Lo expuesto más adelante se basa fuertemente en las ideas desarrolladas en el artículo de Michael B. Giles y Rebecca Carter (2006), [3].

9.2.1. Diferencias finitas para la ecuación del calor

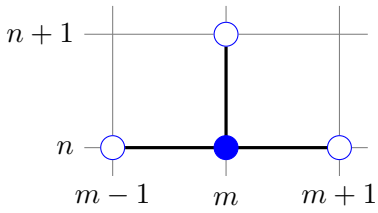
El método de las diferencias finitas es muy natural una vez que hemos abordado las aproximaciones de las derivadas en el Capítulo 4. La idea es cambiar las derivadas de la EDP por sus aproximaciones con ayuda de puntos auxiliares, ya sea a los lados (con h como el espaciamiento en x) o hacia adelante o atrás (con k como el paso del tiempo en t). En general, tomaremos el tiempo inicial como $t_0 = 0$, así con el paso de tiempo, tomaremos valores en los tiempos $t_1 = k$, $t_2 = 2k$, $\ldots$, $t_n = nk$ hasta un tiempo final $t_N = Nk = T$. Por el lado espacial, tomaremos la malla para $x \in [a, b]$ con $x_0 = a$ y $x_M = b$, de modo que $h = (b - a)/M$ nos dice la resolución de nuestra malla para los nodos $x_m = a + mh$.

Es claro, como en la Sección 9.1, que no podemos tener los valores exactos de la solución. Así, para una solución exacta $u(x, t)$ evaluada en la malla, tomaremos una aproximación v_m^n del valor exacto $u(x_m, t_n)$ que también expresaremos por u_m^n. Como antes, esperamos que dados $\xi \in [a, b]$ y $\tau \geq 0$, se tenga $v_m^n \to u(\xi, \tau)$ cuando $h, k \to 0$; hay que tener cuidado, pues debe satisfacerse tanto $m = m(h) = \xi$ como $n = n(k) = \tau$ aún en el límite.

Le llamamos *esténcil* al diagrama de nodos que nos indican cómo se realiza la aproximación de las derivadas. Por ejemplo, para la ecuación del calor (9.7a) podemos aproximar la segunda derivada con la diferencia centrada $\delta_h^2[f](\cdot, t)$ para cada tiempo fijo t_n por

$$u_{xx}(x_m, t_n) \approx \frac{u(x_{m+1}, t_n) - 2u(x_m, t_n) + u(x_{m-1}, t_n)}{h^2} \approx \frac{v_{m+1}^n - 2v_m^n + v_{m-1}^n}{h^2}.$$

El lado izquierdo de la ecuación es más sencillo, pues podemos tomar el método de Euler explícito para el tiempo $t = t_n$, es decir, aproximarnos la EDP por



$$\frac{v_m^{n+1} - v_m^n}{k} = D\,\frac{v_{m+1}^n - 2v_m^n + v_{m-1}^n}{h^2}$$

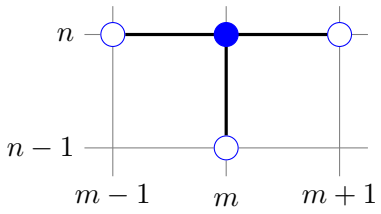
que en una notación más corta es

$$v_m^{n+1} = (1 - 2D\beta)v_m^n + D\beta\big(v_{m+1}^n - v_{m-1}^n\big),$$

al definir $\beta = k/h^2$ que lleva la relación numérica de la malla al ser el recíproco numérico de la

difusión. El esténcil es la "T" invertida. El nodo relleno indica la importancia de expandir todas la derivadas desde un mismo punto, en este caso para (x_m, t_n), los nodos abiertos son aquellos que usaremos en el *esquema*. En este caso, hemos identificado los nodos al tiempo t_{n+1} y colocado del lado izquierdo, al ser solamente uno, podemos resolver de modo explícito una vez que conozcamos los valores v_m^0 podemos aproximar los subsiguiente v_m^n. Este esquema es conocido como *forward Euler* o diferencia centrada en el tiempo y hacia adelante en el tiempo, *FwdT-CS* por su nombre en inglés.

De modo semejante, usando la idea del método de Euler implícito, podemos construir un método al aproximar la derivada temporal de (9.7a) por una derivada hacia atrás, de modo que tenemos el esténcil



.

$$\frac{v_m^n - v_m^{n-1}}{k} = D\,\frac{v_{m+1}^n - 2v_m^n + v_{m-1}^n}{h^2}$$

que en una notación más corta es

$$(1 + 2D\beta)v_m^n - D\beta\big(v_{m+1}^n - v_{m-1}^n\big) = v_m^{n-1}. \tag{9.8}$$

Ahora vemos que los valores v_m^n están dados en una fórmula implícita, es por ello que este esquema se llama *backward Euler* o *BwdT-CS*. Necesitaremos invertir un sistema lineal para encontrar los valores al tiempo t_{n+1} desde el tiempo t_n.

Este último esquema será el que usaremos para hacer el paso de tiempo de Rannacher. Para ello, vale la pena saber cómo resolverlo. Dado el vector de nodos $\mathbf{x} = (x_0, x_1, \ldots, x_M)$, vemos que podemos colocar los valores en los nodos de modo semejante, digamos, con $V^n = (v_0^n, v_1^n, \ldots, v_M^n)^\top$. De este modo, el método de backward Euler puede ser tomado por

$$\begin{aligned} V^0 &= \big(u_0(x_0), u_0(x_1), \ldots, u_0(x_M)\big)^\top, \\ AV^n &= V^{n-1} + \mathbf{c}^n, \qquad n = 1, 2, \ldots, N, \end{aligned}$$

con la matriz A definida por el esquema... pero también por las condiciones de frontera. La fórmula del esquema (9.8) sólo es válida para $m = 1, 2, \ldots, M-1$ y por lo tanto, no podemos colocar la expresión de A de manera directa.

Analicemos qué sucede por ejemplo con el sistema (9.7). Por un lado, $u(0, t) = f(t)$ y directamente, tomamos $v_0^n = f(t_n) = f^n$. Del otro lado, tenemos algo más complicado, $u_x(L, t) = g(t)$. Podemos definir g^n por $g(t_n)$, ¿qué hacer con la derivada? Aproximarla con derivadas finitas, pero para ello, tenemos que recordar que el nodo de referencia (en el esténcil) es (x_m, t_n), hagamos lo mismo con el nodo (x_M, t_n), recordando que éste es el valor más alto en x. Dado que la diferencia centrada que hemos usado es de orden cuadrático, deberíamos tomar una aproximación semejante,

por ejemplo, con

$$u_x(x_M,\, t_n) \;=\; \frac{3u_M^n - 4u_{M-1}^n + u_{M-2}^n}{2h^2} \;+\; \mathcal{O}(h^2).$$

Es decir, podemos pedir que se satisfaga $3v_M^n - 4v_{M-1}^n + v_{M-2}^n = 2h^2 g^n$. Con esto en mente, escribimos la siguiente matriz y vector

$$A \;=\; \begin{pmatrix} 1 & & & & & & \\ -D\beta & 1+2D\beta & -D\beta & & & & \\ & -D\beta & 1+2D\beta & -D\beta & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -D\beta & 1+2D\beta & -D\beta & \\ & & & 1 & -4 & 3 \end{pmatrix}, \qquad \mathbf{c}^n \;=\; \begin{pmatrix} f^n \\ 0 \\ \vdots \\ 0 \\ 2h^2 g^n \end{pmatrix},$$

que complementan lo necesario para correr el algoritmo.

La matriz A tiene la desventaja de ser casi simétrica. Podemos arreglarlo al tomar $\tilde{V}^n = (v_1^n,\, v_2^n,\, \ldots,\, v_{M-1}^n)^\top$, así, con

$$\tilde{A} \;=\; \begin{pmatrix} 1+2D\beta & -D\beta & & & \\ -D\beta & 1+2D\beta & -D\beta & & \\ & \ddots & \ddots & \ddots & \\ & & -D\beta & 1+2D\beta & -D\beta \\ & & & -D\beta & 1+2D\beta \end{pmatrix}, \qquad \tilde{\mathbf{c}}^n \;=\; \begin{pmatrix} D\beta f^n \\ 0 \\ \vdots \\ 0 \\ D\beta \frac{2h^2 g^n + 4v_{M-1}^n - v_{M-2}^n}{3} \end{pmatrix},$$

que en este caso no funciona, pero sí cuando ambas condiciones de frontera son del tipo Dirichlet. Como ventaja, la matriz $\tilde{A}$ es simétrica, definida positiva y tridiagonal, por lo tanto, usando el método de Cholesky, Sec. 7.2, podemos resolver el sistema $\tilde{A}\tilde{V}^n = \tilde{V}^{n-1} + \tilde{\mathbf{c}}^n$ en tiempo lineal haciendo una única factorización de la matriz. Muestra estas últimas afirmaciones.

Entre líneas, tenemos que los métodos de Euler tienen convergencia lineal en el tiempo, pero cuadrática en el espacio. Esto se escribe como métodos $\mathcal{O}(h^2,\, k)$, lo cual no es del todo satisfactorio, pues para ver una convergencia sería deseable escalar $h \to h/10$, por ejemplo, y por lo tanto $k \to k/100$. Para cubrir esta inconformidad, debemos colocar un método que tenga convergencia cuadrática en el tiempo también; este es el método de Crank-Nicolson.

Sabemos que

$$u_t(x,\, t) \;=\; \frac{u(x,\, t+\kappa) - u(x,\, t-\kappa)}{2\kappa} + \mathcal{O}(\kappa^2),$$

sin importar el valor de κ. No podemos saltar un paso de tiempo, por lo que tomamos $\kappa = k/2$ y para que los nodos sean tomados en tiempos kn, la expansión debemos hacerla al tiempo auxiliar

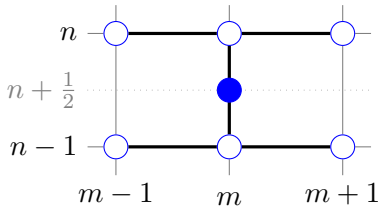
$t_{n+\frac{1}{2}} = t_n + k/2$, de este modo, aproximamos

$$u_t\Big(x_m,\, t_{n+\frac{1}{2}}\Big) \;\approx\; \frac{v_m^{n+1} - v_m^n}{k}.$$

Para el lado derecho de la ecuación del calor, tenemos que recordar que el punto base es en $\big(x_m,\, t_{n+\frac{1}{2}}\big)$, pero dado que

$$u_{xx}\Big(x_m,\, t_{n+\frac{1}{2}}\Big) \;=\; \frac{u_{xx}(x_m,\, t_{n+1}) + u_{xx}(x_m,\, t_n)}{2} + \mathcal{O}(k^2),$$

expandemos las segunda derivadas del promedio.



El esténcil que obtenemos está a la izquierda, el esquema es dado por las partes de las que hemos hablado, a saber,

$$\frac{v_m^{n+1} - v_m^n}{k} \;=\; \frac{D}{2}\left(\frac{v_{m+1}^{n+1} - 2v_m^{n+1} + v_{m-1}^{n+1}}{h^2} + \frac{v_{m+1}^{n} - 2v_m^{n} + v_{m-1}^{n}}{h^2}\right),$$

que nuevamente puede reescribirse separando los pasos de tiempo:

$$(1 + D\beta)v_m^{n+1} \;-\; \frac{D\beta}{2}\big(v_{m+1}^{n+1} + v_{m-1}^{n+1}\big) \;=\; (1 - D\beta)v_m^{n+1} \;+\; \frac{D\beta}{2}\big(v_{m+1}^{n+1} + v_{m-1}^{n+1}\big). \tag{9.9}$$

Este es el **esquema de Crank-Nicolson**, con convergencia $\mathcal{O}(h^2,\, k^2)$.

9.2.2. La ecuación de Black-Scholes

El modelo Black-Scholes es un modelo matemático para la dinámica de un mercado financiero que contiene instrumentos de inversión de derivados. A partir de la ecuación diferencial parcial parabólica en el modelo, conocida como la ecuación de Black-Scholes, se puede deducir una estimación teórica del precio de las opciones de estilo europeo y muestra que la opción tiene un precio único dado el riesgo del valor y su rendimiento esperado (en lugar de reemplazar el rendimiento esperado del valor con la tasa de riesgo neutral). La ecuación y el modelo llevan el nombre de los economistas Fischer Black y Myron Scholes. En ocasiones el nombre de Robert C. Merton aparece también pues fue él quien escribió por primera vez un artículo académico sobre el tema.

La ecuación de Black-Scholes se resume a

$$V_t \;+\; \frac{1}{2}\sigma^2 S^2 V_{SS} \;+\; rSV_S \;-\; rV \;=\; 0, \tag{9.10}$$

donde el retorno de $V(S,\, t)$ (el precio de la opción) proviene de calcular su diferencial vía la *fórmula de Itô*, para el precio de la acción S en el tiempo t. Aquí, σ es el coeficiente de volatibilidad y r es la inversión sin riesgo. El modelo de Black-Scholes de alguna manera presupone un autofinanciamiento

para poder operar y así cancelar la parte estocastica de la versión original de esta ecuación; para los matemáticos este puede ser realmente el paso maestro, para los economistas será un paso un poco más adelante con un cambio de variables.

Antes de continuar, veamos que para resolver un problema en específico, debemos dar las condiciones de contorno. La idea de este modelo es estimar el precio del derivado actual con el pronóstico futuro de su venta ya sea por el *valor de compra* (o *call* en inglés), por lo que llamaremos $C(S,\,t) = V(S,\,t)$ o de por el *valor de venta* (o *put* en inglés) y que llamaremos $P(S,\,t) = V(S,\,t)$. Así, la condición final (no inicial) para un tiempo en el futuro $t = T$ con $t = 0$ el tiempo presente es:

$$\begin{aligned} C(S,\,T) &= (S-E)^+ \qquad \text{(compra)},\\ P(S,\,T) &= (E-S)^+ \qquad \text{(venta)}, \end{aligned}$$

donde la función $(\,\cdot\,)^+ : \mathbb{R} \to \mathbb{R}^+$ está definida como $(x)^+ = \text{máx}\{0,\,x\}$ y E es el denomidado precio base (*strike price*, en inglés), definido como parte del contrato.

Para las condiciones de frontera, vemos que no tiene sentido definir el modelo para precios de acción negativos, por lo que $S = 0$ es una de las fronteras naturales. Sin embargo, el precio puede ser tan alto como querramos y por ello, es normal tomar $S \to \infty$, lo que nos provocará dificultades numéricas. En particular, tenemos

$$\text{Compra:} \qquad \begin{cases} C(0,\,t) = 0, \\ C(S,\,t) - \left[S - Ee^{-r(T-t)}\right] \to 0, \text{ para } S \to \infty. \end{cases}$$

$$\text{Venta:} \qquad \begin{cases} P(0,\,t) = Ee^{-r(T-t)}, \\ P(S,\,t) \to 0, \text{ para } S \to \infty. \end{cases}$$

Lo interesante es que estos problemas pueden ser reducidos a modelos dados por la ecuación del calor con condiciones de contorno tradicionales. Primero tomando el cambio de variables

$$x = \log\frac{S}{E}, \quad \tau = \frac{1}{2}\sigma^2(T-t), \quad w(x,\,\tau) = \frac{1}{E}V\left(Ee^x,\,T - \frac{2\tau}{\sigma^2}\right)$$

y después, reescribiendo

$$w(x,\,\tau) = \exp\left(-\frac{k-1}{2}x - \frac{(k+1)^2}{4}\tau\right)\, v(x,\,\tau),$$

con $k = 2r/\sigma^2$, se obtiene,

$$v_\tau - v_{xx} = 0, \qquad -\infty < x < \infty,\, 0 \le \tau \le T. \tag{9.11}$$

Ejercicio 53: Muestra que con los cambios de variable expuestos arribal, las condiciones finales se transforman en condiciones iniciales, de hecho las condiciones de frontera son introducidas en las manipulaciones. En el caso de la compra:

$$v(x,\,0) \;=\; \begin{cases} e^{\frac{1}{2}(k+1)x} - e^{\frac{1}{2}(k-1)x}, & x \;>\; 0, \\ 0, & x \;\leq\; 0, \end{cases}$$

y en el caso de la venta:

$$v(x,\,0) \;=\; \begin{cases} e^{\frac{1}{2}(k-1)x} - e^{\frac{1}{2}(k+1)x}, & x \;<\; 0, \\ 0, & x \;\geq\; 0. \end{cases}$$

Teniendo en mente $k = 2r/\sigma^2$.

Con el *núcleo del calor* es muy fácil resolver la ecuación (9.11) a partir de la fórmula

$$v(x,\,\tau) \;=\; \frac{1}{\sqrt{4\pi\tau}} \int_{\mathbb{R}} v(y,\,0) \exp\left(-\frac{(x-y)^2}{4\tau}\right) \mathrm{d}y.$$

Que retomando los cambios de variables en la dirección opuesta y con la ayuda de la función auxiliar

$$N(z) \;=\; \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} \exp\left(-\frac{y^2}{2}\right) \mathrm{d}y,$$

dada por la desviación normal estándar, nos lleva a las soluciones

$$C(S,\,t) \;=\; S\,N(d_+) \;-\; Ee^{-r(T-t)}N(d_-), \tag{9.12}$$

$$P(S,\,t) \;=\; Ee^{-r(T-t)}N(-d_-) \;-\; S\,N(-d_+), \tag{9.13}$$

para la compra y la venta respectivamente, donde

$$d_\pm \;=\; \frac{\log(S/E) \;+\; (r \pm \sigma^2/2)(T-t)}{\sigma\sqrt{T-t}}.$$

Tenemos entonces, un caso interesante de ecuaciones diferenciales parciales con solución analítica a la cual le podemos aplicar métodos numéricos y así estudiar su convergencia.

9.2.3. Modelo Black-Scholes desde la visión numérica, un caso inventado

Implementaremos códigos para la ecuación de Black-Scholes (9.10), sólo que de esta vez bajo un cambio de variables poco usual que nos permitirá tener fracciones en su implementación. Tomaremos $x = \log_2 S$ y $\eta = T - t$. Ahora escribimos la solución modificada como $W(x,\,\eta) = V(2^x,\,T-\eta)$ y

obtenemos

$$\frac{\partial W}{\partial \eta} = \frac{\sigma^2/2}{(\log 2)^2}\frac{\partial^2 W}{\partial x^2} + \frac{r-\sigma^2/2}{\log 2}\frac{\partial W}{\partial x} - rW \quad \text{en} \quad I_x \times I_\eta = (x_\ell,\, x_r) \times (0,\, T), \tag{9.14}$$

para el tiempo inicial $t = 0$ y el tiempo terminal $t = T$; así $\eta \in [0,\, T]$ también. Como en el cambio de variables original, esta ecuación tiene la ventaja de que sus coeficientes no dependen de S o t, ahora son constantes, compara con (9.10). Trabajaremos con las condiciones final y de frontera más adelante; tomaremos condiciones de Dirichlet en $x = -2$ (respectivo al precio, $S = 1/4$) y de Neumann en $x = 2$ (resp., $S = 4$).

Consideramos como antes, tomar h como espaciamiento en x y k el paso de tiempo. De este modo, $x_0 = a$ y $x_m = a + mh$, recordamos que $M = (b-a)/h$ y, por tanto, $x_M = b$. Para la variable temoporal tenemos $\eta_0 = 0$ y $\eta_n = nk$ con $\eta_N = T$. Queremos dos métodos de orden 2 en el espacio, por lo tanto, las condiciones de frontera también deben ser así. Sólo necesitamos Neumann del lado derecho.

Discusión: Debemos hacer un paréntesis aquí. Antes de seguir leyendo, reflexiona sobre la afirmación en la última frase.

El modelo original de Black-Scholes considera $S \in [0,\, +\infty)$, al hacer el cambio de variables, el semieje para S determina que x debe estar definido en toda la recta real. Esto produce un problema de implementación, sin embargo, observando las condiciones de frontera a ser utilizadas (*cf.* la ecuación (9.15)), podemos pensar en resolver para el intervalo $S \in [S_\ell,\, S_r]$, para $0 < S_\ell \ll 1 \ll S_r < \infty$, por ejemplo. Si S_ℓ es menor que K en (9.15), podemos pensar que $V(S_\ell,\, t) = 0$ sea una buena aproximación. Del lado derecho, tenemos otros problemas, pero si S_r es distante de K, entonces podemos pensar que $V_S(S_r,\, t)$ puede aproximarse por $V_S(S_r,\, T)$ que en este caso es uno. Así llegamos a la idea de Dirichlet a la izquierda y Neumann a la derecha para $x_\ell = \log_2 S_\ell$ y $x_r = \log_2 S_r$ en (9.14).

Para los métodos que queremos, vamos a consider

$$a = \frac{1}{2}\frac{r-\sigma^2/2}{\log 2} \qquad \text{y} \qquad b = \frac{\sigma^2}{2\,(\log 2)^2},$$

donde hemos colocado un medio en a por conveniencia más adelante. De este modo, el esquema de Euler implícito para (9.14) se resume a

$$\frac{v_m^n - v_m^{n-1}}{k} = r v_m^n - a\frac{v_{m+1}^n - v_{m-1}^n}{h} - b\frac{v_{m+1}^n - 2v_m^n + v_{m-1}^n}{h^2},$$

para $m \in \{1,\, 2,\, \ldots,\, M-1\}$. (Observa que no aparece $2h$ en el denominador de la diferencia centrada

para la derivada de primer orden sino h, este 2 está en a.) Esta ecuación se puede reescribir como

$$(b\beta - a\lambda)v_{m-1}^n \;+\; (1 - rk - 2b\beta)v_m^n \;+\; (b\beta + a\lambda)v_{m+1}^n \;=\; v_m^{n-1},$$

donde $\lambda = k/h$ y $\beta = k/h^2$ se toman como de costumbre. El esquema es $\mathcal{O}(h^2,\, k)$. Simplificaremos la notación más adelante al definir

$$A \;=\; b\beta - a\lambda, \qquad B \;=\; 1 - rk - 2b\beta \quad \text{y} \quad C \;=\; b\beta + a\lambda.$$

La condición de contorno a la izquierda puede ser actualizada desde un inicio o a cada paso con $v_0^n = W(x_0,\, \eta_n)$ que para simplificar llamamos $L(\eta) = W(x_\ell,\, \eta) = V(S_\ell,\, T - \eta)$. La condición de contorno a la derecha, tiene que unirse al sistema matricial, que obtendremos de arriba, con la ecuación

$$v_{M-2}^n - 4v_{M-1}^n + 3v_M^n \;=\; 2h\, V_S(S_r,\, T - \eta_n),$$

que puede incluirse al usar vectores $V^n := (v_1^n,\, v_2^n,\, \ldots,\, v_M^n)^\top$. Nuevamente, simplificamos la notación con $R(\eta) = V_S(2^{x_r},\, T - \eta)$

Podemos entonces resolver el problema lineal $\mathbf{A}V^n = V^{n-1} + F^n$, donde

$$\mathbf{A} \;:=\; \begin{bmatrix} B & C & & & \\ A & B & C & & \\ & \ddots & \ddots & \ddots & \\ & & A & B & C \\ & & 1 & -4 & 3 \end{bmatrix} \in \mathbb{R}^{M\times M}, \qquad F^n = \begin{bmatrix} -A\, L(\eta_n) \\ 0 \\ \vdots \\ 0 \\ 2h\, R(\eta_n) \end{bmatrix} \in \mathbb{R}^M.$$

Observamos que F^n, aunque dependa del tiempo, está definida de entrada, así solamente V^n será incognita.

El caso para el método de Crank-Nicolson es semejante, usando los mismos valores de a, b, λ y β, tenemos el esquema

$$\begin{aligned} v_m^{n+1} - v_m^n \;&=\; rkv_m^{n+1} \;-\; a(v_{m+1}^{n+1} - v_{m-1}^{n+1}) \;-\; b(v_{m-1}^{n+1} - 2v_m^{n+1} + v_{m+1}^{n+1}) \\ &+\; rkv_m^n \;-\; a(v_{m+1}^n - v_{m-1}^n) \;-\; b(v_{m-1}^n - 2v_m^n + v_{m+1}^n), \end{aligned}$$

que tiene precisión $\mathcal{O}(h^2,\, k^2)$. Los elementos entre paréntesis son encontrados en el método implícito también. De este modo, podemos simplifcar las ecuaciones con

$$Av_{m-1}^{n+1} \;+\; (1 + B)v_m^{n+1} \;+\; Cv_{m+1}^{n+1} \;=\; -Av_{m-1}^n \;+\; (3 - B)v_m^n \;-\; Cv_{m+1}^n.$$

(Los términos $1 + B$ y $3 - B$ pueden condensarse en notación.)

Como antes, la condición de contorno a la izquierda es $v_0^n = L(\eta_n)$. La condición de contorno a la derecha satisface

$$v_{M-2}^{n+1} - 4v_{M-1}^{n+1} + 3v_M^{n+1} + v_{M-2}^n - 4v_{M-1}^n + 3v_M^n \;=\; 4h\, R\Big(\Big(n+\frac{1}{2}\Big)k\Big),$$

pues el punto de partida de la EDP para este esquema es al tiempo $\eta_{n+1/2}$.

En este caso, el sistema a resolver es $\mathbf{A}V^{n+1} = \mathbf{B}V^n + F^{n+1/2}$, donde

$$\mathbf{A} \;:=\; \begin{bmatrix} 1+B & C & & & \\ A & 1+B & C & & \\ & \ddots & \ddots & \ddots & \\ & & A & 1+B & C \\ & & 1 & -4 & 3 \end{bmatrix}, \qquad F^{n+1/2} = \begin{bmatrix} A[L(\eta_n) - L(\eta_{n+1})] \\ 0 \\ \vdots \\ 0 \\ 4h\, R\big(\eta_{n+\frac{1}{2}}\big) \end{bmatrix} \in \mathbb{R}^M,$$

$$\mathbf{B} \;:=\; \begin{bmatrix} 3-B & -C & & & \\ -A & 3-B & -C & & \\ & \ddots & \ddots & \ddots & \\ & & -A & 3-B & -C \\ & & -1 & 4 & -3 \end{bmatrix} \in \mathbb{R}^{M\times M},$$

además, escribimos $F^{n+1/2}$ aunque las evaluaciones sean en η_n y η_{n+1} pues provienen del punto medio como promedio de éste.

Ahora, como ejercicio, usaremos estos dos esquemas para hacer un ejercicio de compra (*call*) para una opción europea con *payoff* al tiempo terminal. Es decir, resolveremos el problema de Black-Scholes (9.10) con condición final dada por

$$V(S,\, T) \;=\; \max\{S - K,\, 0\}\,, \tag{9.15}$$

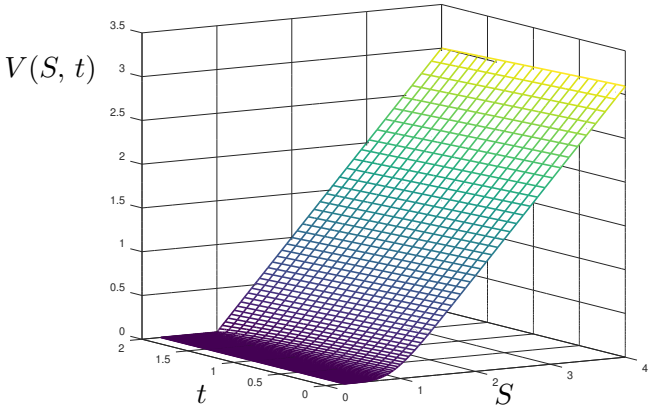
donde S es el precio de la opción, $T = 2$ el tiempo terminal del ejercicio y consideramos el precio base $K = 1$ ya normalizado. Tomamos un dominio numérico para $x \in [-2,\, 2]$ no representa toda la recta positiva, sino solo los valores para la acción con $S \in [1/4,\, 4]$. Las condiciones de contorno serán entonces

$$V(1/4,\, t) \;=\; 0, \quad \text{y} \quad V_S(4,\, t) \;=\; 1,$$

para todo tiempo $t \in [0,\, 2]$. Para colocar algunos valores, consideramos una inversión sin riesgo del cinco por ciento, es decir, $r = 0.05$ y con con una volatibilidad razonable $\sigma = 0.2$.

Primero usemos el método Euler implícito con $h = 1/20$ y $k = 1/10$ y hamos un gráfico 3D con coordenadas $(S,\, t,\, V)$ de la solución obtenida.

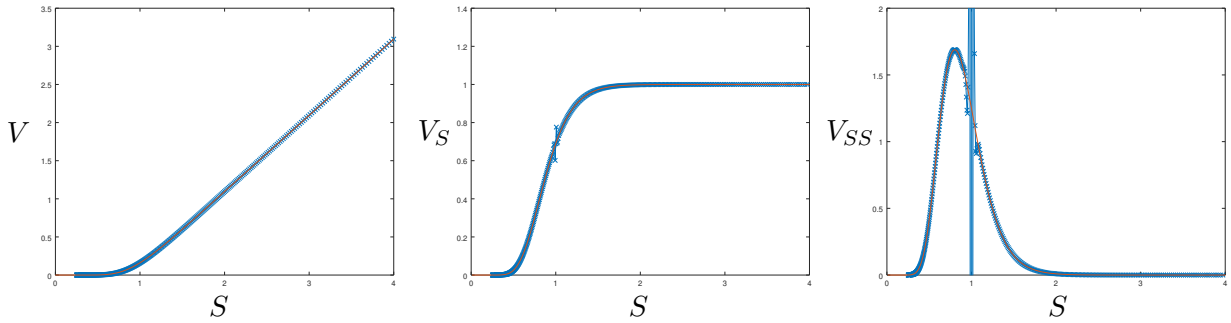
La figura a la derecha nos da una idea del comportamiento de la solución y vemos que se parece a la solución exacta dada en (9.12). De hecho, usando la solución exacta, podemos hacer una tabla conteniendo el paso de tiempo k y espaciamiento h para el máximo de los errores en $t = 0$. Digamos, para los pasos $k = h = \{1/10, 1/20, 1/40\}$ obtenemos



h = k:	1/10	1/20	1/40
Error relativo:	8.132698235	2.419519667	1.000000000
Error absoluto:	1.51778e-03	5.90969e-04	2.53428e-04

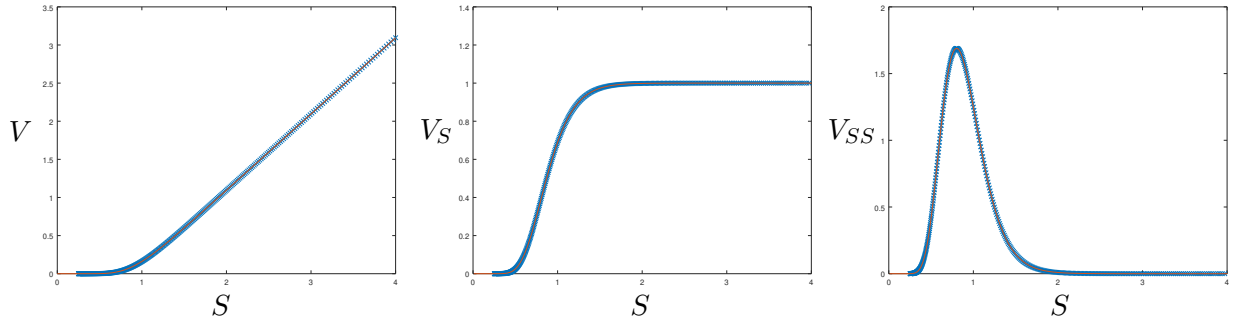
Lo cual es consistente con la teoría, pues el método es $\mathcal{O}(h^2, k)$ y en este caso estamos viendo una mejora poco mayor al orden cuadrático.

La teoría de diferencias finitas para EDP garantiza la convergencia cuadrática en la norma $L^2[x_\ell, x_r]$. Sin embargo, no dice nada sobre la convergencia de las derivadas de la función a la que nos estamos aproximando. En el caso del modelo de Black-Scholes, estas derivadas generan el concepto de las llamadas "Griegas" y, por tanto, es útil su regularidad. Sabemos sobre la estabilidad incondicional de Crank-Nicolson y mostramos su regularidad al comparar los gráficos analíticos y numéricos de V, V_S y V_{SS} al tiempo inicial $t = 0$ con mallado $h = 1/100$, $k = 1/10$.



De la discontinuidad en la primera derivada en $S = K$ para la condición final en (9.15), vemos que se desprenden las oscilaciones en V_S y V_{SS} (*cf.* [3]). Una manera de atenuar esto es considerar cuatro pasos con $\tilde{k} = k/2$ realizados con Euler implícito y ahora la condición $V(S, T - 4\tilde{k})$ como la condición terminal para el esquema de Crank-Nicolson.

La repetición del desarrollo nos lleva a gráficos semejantes para el valor $V(S, 0)$, pero vemos grandes mejoras en las derivadas del activo. En estas figuras se ve claramente que los pequeños picos en la primera derivada han desaparecido. Las grandes oscilaciones de en la segunda derivada desaparecieron completamente. Este es el *método de suavizamiento de Rannacher* que funciona

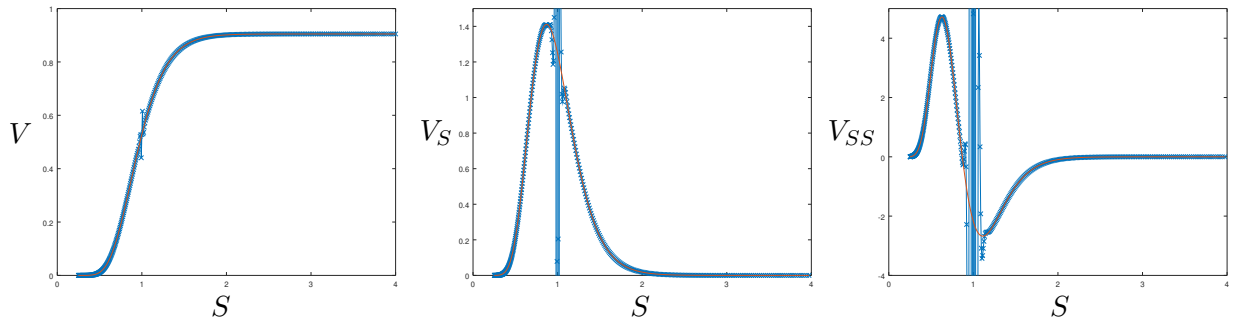


maravillosamente en este caso. Las pequeñas inestabilidades generadas por los 4 medios pasos de Euler implícito suavizan la conidión final aparentando que esta tiene derivadas continuas.

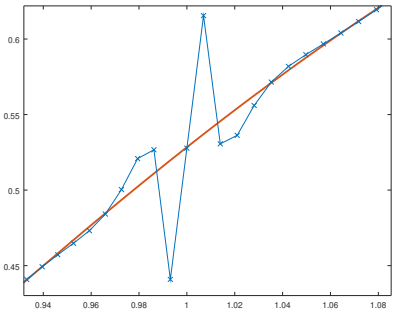
Además, ésto es aun razonable cuando es la condición final que es discontinua. Consideremos, por ejemplo, con una compra para una "opción digital" con condiciones dadas por

$$V(S,T) \;=\; \mathcal{H}(S-K), \quad V(1/4,\,t) \;=\; 0, \quad \text{y} \quad V_S(4,\,t) \;=\; 0,$$

donde $\mathcal{H}(\xi)$ es la función Heaviside. Este ejercicios nos lleva a suavizamientos de Rannacher menores pero aun evidentes, veamos los próximos gráficos.



En las figura arriba las curvas con curces representan la solucion sin suavizamiento. La curva "continua" en rojo tiene el método de Rannacher. Al lado tenemos un detalle de esta solución $V(S,\,0)$ numérica, pues en este caso ya vemos oscilaciones cuando no hay suavizamiento. Es decir, el método de medios pasos implícitos funciona muy bien inclusive en estos casos.



Apéndice A

Ecuaciones lineales, sensibilidad

En esta sección veremos la noción de norma vectorial y matricial. Ambas son la extensión natural del valor absoluto, por ejemplo. Una norma vectorial es una función $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ que satisface las propiedades

1. Si $\mathbf{x} \neq \mathbf{0}$, entonces $\|\mathbf{x}\| > 0$,
2. Para todo $\alpha \in \mathbb{R}$, tenemos que $\|\alpha\mathbf{x}\| = |\alpha| \|\mathbf{x}\|$, si $\mathbf{x} \in \mathbb{R}^n$,
3. Para cualesquier $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$.

Estas propiedades son importantes para el buen desarrollo de una norma, un manera de medir los vectores en el espacio. La tercera de ellas se conoce como la *desigualdad triangular* y una útil variante de ella es

$$\|\mathbf{x} - \mathbf{y}\| \geq \|\mathbf{x}\| - \|\mathbf{y}\|.$$

Entre la infinidad de normas vectoriales, las tres más comunes y que son bastante usadas en la matemática computacional son

1. $\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$,
2. $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$,

∞. $\|\mathbf{x}\|_\infty = \text{máx}_{i \in \{1, \ldots, n\}} |x_i|$,

conocidas como *norma 1*, *norma 2* o *cartesiana* y *norma infinito*, respectivamente. Las tres son parte de las p-normas, las cuales, para $p \in [1, \infty)$ se definen como

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}},$$

y en el caso de la norma infinita como el límite de esta definición.

En el caso de dimensión finita, se sabe que *todas las normas son equivalentes*, es decir, dadas dos normas $\|\cdot\|_a$ y $\|\cdot\|_b$ existen constantes positivas c y C tales que $c\|\mathbf{x}\|_a \leq \|\mathbf{x}\|_b \leq C\|\mathbf{x}\|_a$, para todo vector $\mathbf{x} \in \mathbb{R}^n$. En el caso de las normas dadas arriba, se satisface

$$\|\mathbf{x}\|_1 \geq \|\mathbf{x}\|_2 \geq \|\mathbf{x}\|_\infty.$$

Es un bonito ejercicio hacer estas pruebas de forma gráfica en dimensión 2.

Discusión: Si todas las normas vectoriales nos darán resultados equivalentes, entonces,

1. ¿cuál sería más razonable de usar en la máquina?
2. ¿dónde podemos tener mala interpretación de los datos?

El caso de matrices, tiene normas con propiedades análogas. Para $\|\cdot\| : \mathbb{R}^{n\times m} \to \mathbb{R}$, satisface

1. Si $A \neq 0$, entonces $\|A\| > 0$,
2. Para todo $\alpha \in \mathbb{R}$, tenemos que $\|\alpha A\| = |\alpha|\,\|A\|$, si $A \in \mathbb{R}^{n\times m}$,
3. Para cualesquier A, $B \in \mathbb{R}^{n\times m}$, $\|A + B\| \leq \|A\| + \|B\|$.

En el caso de los vectores, la desigualdad triangular es muy geométrica y se aplica a un concepto directo de la suma de vectores. En las matrices también tenemos el producto entre ellas o por un vector. Diremos que la norma matricial es **consistente** si para cualesquier A, B dadas, se satisface

$$\|AB\| \leq \|A\|\,\|B\|, \tag{A.1}$$

cuando el producto AB está bien definido; hay compatibilidad de dimensiones.

Ejemplo: Para exhibir una norma que no sea consistente, considera $\|A\|_{\text{máx}} := \text{máx}_{i,j}\,|a_{ij}|$. Observa que las propiedades que garanticen que sea norma se satisfacen: como $\|A\|_{\text{máx}} = 0$ si y sólo si todas las entradas de A son nulas, se satisface la primera. La segunda, se muestra al multiplicar por el escalar α cada entrada y se deja de tarea moral mostrar la tercera propiedad.

Bueno, ahora veamos que

$$\left\| \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \right\|_{\text{máx}} = \left\| \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} \right\|_{\text{máx}} = 2,$$

por un lado, pero

$$\left\| \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \right\|_{\text{máx}} \left\| \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \right\|_{\text{máx}} = 1 \cdot 1 = 1,$$

que claramente no es mayor que 2.

Observa que las tres apariciones de normas en (A.1) pueden ser distintas. Este es el caso del producto matriz vector, diremos que las normas $\|\cdot\|_M$ para matrices y $\|\cdot\|_{\mathbf{v}}$ para vectores son *consistentes* si

$$\|A\mathbf{x}\|_{\mathbf{v}} \;\leq\; \|A\|_M \, \|\mathbf{x}\|_{\mathbf{v}}$$

se satisface para toda matriz y vector. Nuevamente, si la matriz A es rectangular, la normas vectoriales serán distintas.

Las normas matriciales clásicas son

1. $\|A\|_1 \;=\; \text{máx}_{j\in\{1,\ldots,n\}} \sum_{i=1}^{m} |a_{ij}|$,

F. $\|A\|_F \;=\; \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}$,

∞. $\|A\|_\infty \;=\; \text{máx}_{i\in\{1,\ldots,m\}} \sum_{j=1}^{n} |a_{ij}|$,

llamadas *norma 1*, *norma de Frobenius* y *norma infinito*.

Curiosamente la norma de Frobenius es consistente pero las p-normas definidas así no lo serían. Un método sencillo para definir una norma consistente entre matrices y vectores puede ser dado al definir la norma matricial $\|\cdot\|_p$ a partir de una norma vectorial $\|\cdot\|_p$ como aquella que satisface

$$\|A\|_p \;:=\; \sup_{\|\mathbf{x}\|_p\leq 1} \frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_p} \;=\; \max_{\|\mathbf{x}\|_p=1} \|A\mathbf{x}\|_p.$$

En el caso de la norma 2 para vectores, se encontrará la norma 2 para matrices que no es como la norma de Frobenius. De esta definición arriba puede entenderse la norma 1 y la norma infinito para matrices a partir de sus homónimas para vectores.

De este momento en adelante se consideraran únicamente normas consistentes entre matrices y vectores aunque no se diga de manera explícita.

Ejercicio 54: Busca una matriz $A \in \mathbb{R}^{2\times 2}$ tal que $\|A\|_F \neq \|A\|_2$.

A.1. Error relativo

Con las normas tenemos un símil del valor absoluto y por lo tanto una manera de medir y describir el tipo de error producido en algún desarrollo. El error relativo de $\mathbf{y}$ como aproximación de $\mathbf{x}$ es como antes

$$\varrho \;=\; \frac{\|\mathbf{y}-\mathbf{x}\|}{\|\mathbf{x}\|}.$$

También tenemos el desarrollo hecho en los primeros días de clases en los que mostramos que el error relativo de $\mathbf{y}$ como aproximación de $\mathbf{x}$ es semejante al error relativo de $\mathbf{x}$ como aproximación

de $\mathbf{y}$, es decir, tenemos:

$$\text{Si} \qquad \frac{\|\mathbf{y}-\mathbf{x}\|}{\|\mathbf{x}\|} \;\leq\; \varrho \;<\; 1 \qquad \text{entonces} \qquad \frac{\|\mathbf{x}-\mathbf{y}\|}{\|\mathbf{y}\|} \;\leq\; \frac{\varrho}{1-\varrho}.$$

Así, si $\varrho = 10^{-k}$, entonces $\varrho/(1-\varrho) = 0.\{0\}^{k-1}1\{0\}^{k-1}1\cdots$, para $k-1$ repeticiones de ceros, por ejemplo, con $k=3$, es $0.001001001\cdots$; son del mismo orden.

Ejemplo: Sobre la equivalencia de las normas, a menos que el problema sea creado con una norma en específico como *mínimos cuadrados*, es más sencillo usar la norma infinito. Tomemos

$$\mathbf{x} \;=\; \begin{pmatrix} 1.0000 \\ 0.0100 \\ 0.0001 \end{pmatrix} \qquad \text{y una aproximación dada por} \qquad \mathbf{y} \;=\; \begin{pmatrix} 1.0002 \\ 0.0103 \\ 0.0002 \end{pmatrix}.$$

El error absoluto es

$$\begin{aligned} \|\mathbf{x}-\mathbf{y}\|_2 &= \sqrt{0.0002^2 + 0.0003^2 + 0.0001^2} \;\approx\; 3.74166 \times 10^{-4}, \\ \|\mathbf{x}-\mathbf{y}\|_\infty &= \text{máx}\{0.0002,\, 0.0003,\, 0.0001\} \;=\; 3 \times 10^{-4}. \end{aligned}$$

Son del mismo orden y claramente la primera requiere un esfuerzo mayor; un punto para la norma infinito. Lo mismo sucede con el error relativo, pues

$$\begin{aligned} \frac{\|\mathbf{x}-\mathbf{y}\|_2}{\|\mathbf{x}\|_2} &= \frac{\sqrt{0.0002^2+0.0003^2+0.0001^2}}{\sqrt{1+0.01^2+0.0001^2}} \;\approx\; 3.74147 \times 10^{-4}, \\ \frac{\|\mathbf{x}-\mathbf{y}\|_\infty}{\|\mathbf{x}\|_\infty} &= \frac{\text{máx}\{0.0002, 0.0003, 0.0001\}}{\text{máx}\{1, 0.01, 0.0001\}} \;=\; 3 \times 10^{-4}. \end{aligned}$$

Notemos que en el caso del error absoluto entrada a entrada tenemos que son 2×10^{-4}, 3×10^{-4}, 1×10^{-4}, respectivamente y el error del vector es del mismo orden. Sin embargo, en el caso del error relativo entrada a entrada, tenemos 2×10^{-4}, 3×10^{-2} y 1, lo cual muestra que el error relativo de la tercera entrada es del 100 % y por el otro lado, el error relativo del vector tiene casi cuatro cifras de precisión.

La discusión puede ser manipulada, pues parecería que el error relativo de cada una de las entradas depende de dónde se alteró cada coordenada del vector. Siendo así, ésto causaría problemas si lo que queremos es alterar "relativamente" un vector con un tamaño determinado. Es mejor pensar en el objeto completo, no perder la imagen y ver de este modo al vector representado en el espacio $\mathbb{R}^3$. Notamos que entre $\mathbf{x}$ y $\mathbf{y}$ realmente hay un error pequeño, pues ambos vectores se ven muy semejantes para un observador en el mismo espacio, el cambio en cada coordenada ahora parece irrelevante.

Veamos qué sucede cuando los cambios mínimos se dan en cada entrada de una matriz. Digamos, al resolver

$$A\mathbf{x} \;=\; \mathbf{b},$$

sabemos que la matriz introducida no será A sino una cierta matriz $\tilde{A}$ y, por consiguiente, el resultado no será más $\mathbf{x}$ sino $\tilde{\mathbf{x}}$. Pues la verdad, estaremos resolviendo el problema

$$\tilde{A}\tilde{\mathbf{x}} \;=\; \mathbf{b};$$

donde no tomamos la perturbación de $\mathbf{b}$ por simplicidad en los cálculos a seguir.

Tomemos la diferencia entre las matrices como una matriz que controla el error como $E = \tilde{A} - A$, de modo que $\tilde{A} = A + E$. Si A es una matriz regular, saber si $\tilde{A}$ continúa siendo o no singular es importante.

Afirmación A.1. *Sea A una matriz regular. Si $\|A^{-1}E\| < 1$, entonces $A + E$ es no singular.*

La prueba de este resultado es directo. Queremos mostrar que $(A+E)\mathbf{x} = \mathbf{0}$ solamente si $\mathbf{x} = \mathbf{0}$, y dado que A es regular, tenemos que

$$(A + E)\mathbf{x} \;=\; A(I + A^{-1}E)\mathbf{x} \;\neq\; \mathbf{0} \qquad \text{si y solo si} \qquad (I + A^{-1}E)\mathbf{x} \;\neq\; \mathbf{0}.$$

Además,

$$\|(I + A^{-1}E)\mathbf{x}\| \;=\; \|\mathbf{x} + A^{-1}E\mathbf{x}\| \;\geq\; \|\mathbf{x}\| - |-1|\,\|A^{-1}E\|\,\|\mathbf{x}\| \;=\; (1 - \|A^{-1}E\|)\|\mathbf{x}\| \;>\; \mathbf{0},$$

lo que establece la validez de la afirmación.

Ahora vamos establecer el resultado que nos guiará en saber qué tan bien podemos resolver un sistema lineal. Tomando que resolveremos $\tilde{A}\tilde{\mathbf{x}} = \mathbf{b}$, vamos a multiplicar por la izquierda A^{-1}, luego

$$A^{-1}\tilde{A}\tilde{\mathbf{x}} \;=\; A^{-1}(A + E)\tilde{\mathbf{x}} \;=\; (I + A^{-1}E)\tilde{\mathbf{x}} = A^{-1}\mathbf{b} \;=\; \mathbf{x},$$

de donde sigue que $\mathbf{x} - \tilde{\mathbf{x}} = A^{-1}E\tilde{\mathbf{x}}$, que tomando las normas, nos lleva a

$$\|\mathbf{x} - \tilde{\mathbf{x}}\| \;=\; \|A^{-1}E\tilde{\mathbf{x}}\| \;\leq\; \|A^{-1}E\|\,\|\tilde{\mathbf{x}}\| \qquad \Longrightarrow \qquad \frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\tilde{\mathbf{x}}\|} \;\leq\; \|A^{-1}E\|,$$

donde hemos necesitado la *consistencia* entre la norma vectorial y la norma matricial. Esta última desigualdad nos habla de la proximidad de $\mathbf{x}$ a $\tilde{\mathbf{x}}$. Usando el resultado previo, tenemos que el error relativo de $\tilde{\mathbf{x}}$ como aproximación de $\mathbf{x}$ satisface

$$\frac{\|\tilde{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \;\leq\; \frac{\|A^{-1}E\|}{1 - \|A^{-1}E\|}. \tag{A.2}$$

Esta ecuación no tiene una interpretación directa y es el corazón del análisis de estabilidad.

Primero notemos que para una norma consistente, tenemos que

$$\|A^{-1}E\| \;\leq\; \|A^{-1}\|\,\|E\| \;=\; \kappa(A)\,\frac{\|E\|}{\|A\|},$$

donde hemos tomado $\kappa(A) = \|A\|\,\|A^{-1}\|$ que es definido como el **número de condición de la matriz A**. Así, si $\kappa(A)\|E\|/\|A\| < 1$, entonces de (A.2) tenemos que

$$\frac{\|\tilde{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \;\leq\; \frac{\kappa(A)\frac{\|E\|}{\|A\|}}{1 \;-\; \kappa(A)\frac{\|E\|}{\|A\|}}.$$

Si notamos que $\kappa(A)\|E\|/\|A\|$ es pequeño, entonces, podemos considerar que

$$\varrho \;\leq\; \kappa(A)\frac{\|E\|}{\|A\|}, \qquad \text{donde} \qquad \varrho \;=\; \frac{\|\tilde{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|}$$

es el error relativo de $\tilde{\mathbf{x}}$ como aproximación de $\mathbf{x}$. Dado que $E = \tilde{A} - A$, la fracción a la derecha $\|E\|/\|A\|$ es el error relativo de $\tilde{A}$ como aproximación de A.

Discusión: ¿Qué nos dice este último argumento sobre ϱ?

Sin ir más lejos, veamos una propiedad muy importante del *número de condición de A con respecto a la inversión*, es decir, $\kappa(A)$. Este número es mayor a uno, pues

$$1 \;\leq\; \|I\| \;=\; \|A\,A^{-1}\| \;\leq\; \|A\|\,\|A^{-1}\| \;=\; \kappa(A).$$

Así, si al introducir los números en A obtenemos una variación pequeña por cada entrada, donde $\tilde{a}_{ij} = a_{ij}(1+\epsilon_{ij})$. Hay que notar que éstas perturbaciones satisfacen $|\epsilon_{ij}| \leq \varepsilon_M$ y así para cualquiera de las normas usuales tenemos $\|E\| \leq \varepsilon_M\,\|A\|$.

Finalmente, vemos que al resolver el sistema modificado $\tilde{A}\tilde{\mathbf{x}} = \mathbf{b}$ obtenemos que

$$\frac{\|\tilde{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \;\leq\; \kappa(A)\varepsilon_M.$$

En particular si $\varepsilon_M = 10^{-t}$ y $\kappa(A) = 10^k$, la solución $\tilde{\mathbf{x}}$ puede producir un error de tamaño 10^{k-t}, con lo cual el mayor número (en valor absoluto) puede ser impreciso a partir de la $t-k$ cifra. Lo cual explica el siguiente *dictado*: «Si $\kappa(A) = 10^k$, entonces esperamos perder k cifras al resolver $A\mathbf{x} = \mathbf{b}$.»

Ejemplo: Debemos tener cuidado con el *mal condicionamiento* artificial, por ejemplo, al resolver dos sistemas análogos

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \qquad \text{contra} \qquad \hat{A} = \begin{pmatrix} 10^{-t} & 10^{-t} \\ 1 & 2 \end{pmatrix}$$

al escalar también la primera entrada de $\mathbf{b}$.

Ejercicio 55: considera las matrices del ejemplo anterior y los vectores

$$\mathbf{b} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \qquad \text{contra} \qquad \hat{\mathbf{b}} = \begin{pmatrix} 2 \times 10^{-t} \\ 3 \end{pmatrix}$$

y resuelve en PYTHON con la descomposición LU (sin pivoteo) y LUP para los valores de $t = 5, 10, 15, 20$.

Ejercicio 56: Busca todos los lugares en este texto donde se ha utilizado que las normas son consistetes. Piensa en contraejemplos que muestren que esos pasos no son posibles.

Apéndice B

Códigos en `Python`

Desde Otoño de 2021 se propuso dar los cursos numéricos del ITAM con el uso de *software* libre, en este caso, optamos por `Python`. En este breve apéndice hay tres códigos auto contenidos para dar un golpe inicial al uso de este lenguaje suponiendo el uso de Spyder, por ejemplo.

Ejercicios1: Aquí se mira brevemente el uso de variables, definiciones, ciclos y el uso tanto de la introducción de datos como de la impresión.

```
# -Ejercicios1.py *- coding: utf-8 -*-
"""
Necesidades del estudiante:
    Utilizaremos principalmente los siguientes paneles de ventana: Consola de
    IPython, Editor, Explorador de archivos e Inspector de objetos.
"""
#%% Por ejemplo, este símbolo divide el documento del Editor en celdas.
"""
El triángulo verde de la herramienta de la barra ejecuta todo el archivo (después
de guardarlo), Ctrl-Entrar (Comando-Retorno en un Mac) ejecuta sólo la celda en
la que se encuentra el cursor (pero no guarda).

Instrucciones para cambiar el directorio de trabajo en Spyder: En la parte
superior de la derecha verá una ruta, el directorio de trabajo. A su derecha hay
una carpeta. Haciendo clic en ella se puede cambiar el directorio de trabajo.
Cuando lo haces, puedes hacer clic en el icono situado a la derecha de ese icono
y establecer esa ruta de acceso como el nuevo directorio de trabajo de la consola
IPython.

A continuación, todos los paneles: Editor, Consola de IPython, y el Explorador de
archivos apuntan a este directorio de trabajo actual. En las versiones más
recientes de Spyder, este botón ha sido eliminado y el editor y La consola de
IPython se establece automáticamente en el directorio de trabajo actual.

```

```
Vistazo rápido a las operaciones aritméticas
    +, -, *, **, /, //, %
Estos símbolos suman, restan, múltiplican, elevan a una potencia, dividen, la
divisiónentera (se tira la parte fraccionaria), calcula el resto en la división
para los enteros. Prueba algunos ejemplos de forma interactiva en la consola de
IPython a la derecha.
"""
#%%
def hello():
    """ Imprime el clásico: hello, world! """
    print("Hello, world!")
#%%
def areacirculo(radius):
    """ Calcula el área del círculo del radio dado """
    area = 3.14*radius**2
    print("El área del círculo de radio", radius, " es", area, ".")
#%%
"""
Ejercicio:
    Escribe una función 'def areatriangulo(b,h)' para calcular el área de un
    triángulo, la fórmula es el área = .5*b*h.
La salida debe ser similar a:
    El área de un triángulo de base 3 y altura 5 es 7.5.
Puedes probar la función ejecutando el código siguiente:
"""
#%%
# Los siguientes prueban el areatriangulo()
areatriangulo(3,5)
areatriangulo(2,20)
#%%
""" Solución: """
def areatriangulo(b,h):
    pass  # borra esta línea e introduce tu código
""" Fin de la solución. """
#%%
"""
Para hacer una cadena de caracteres (string) podemos usar [ ' ] o [ " ].
Cualquiera funciona igual de bien. Pero si la cadena contiene alguno de los dos,
necesesitamos usar el otro:
"""
#%%
nombre = "Su nombre es Conan O'Brien"
gato   = 'A mi gato lo llamamos "Manchitas"'
print(nombre)
print(gato)
```

```
#%%
"""
Si necesitas un [ ' ] y un [ " ] en la cadena, puedes utilizar la barra [ | ] que
le dice a Python que se debe tomar el siguiente carácter como el carácter literal
y no es una comilla para delimitar la cadena. Vémoslo en acción escapando de la
[ " ] siguiente:
"""
#%%
ambos = "El nombre de mi gato es \"Manchitas O'Brien\""
print(ambos)
#%%
def fahrenheit_2_celsios(temp):
    """ Convierte tenperatura en grados Fahrenheit a grados Celsios.
        La fórmula es 5/9 de la temperatira menos 32 """
    # Nota: esta linea no es ejecutada
    # end='' mantiene la impresión en la misma línea.
    print("La temperatura en grados Fahrenheit", temp, end='')
    nuevaTemp = 5*(temp - 32)/9
    print(" es equivalente a", nuevaTemp, " grados Celsios")
#%%
"""
Ejercicio:
    Escribe una función 'def celsios_2_fahrenheit(temp)' para convertir grados
    Celsios a la temperatura en Fahrentheit. La fórmula es (9/5) veces temp más
    32.
Imprime el resultado con el formato:
    La temperatura 50.0 grados Celsios es equivalente a 122.0 grados Fahrenheit.
"""
#%%
# Las siguientes líneas harán la prueba de tu solución
celsius_to_fahrenheit(100)
celsius_to_fahrenheit(0)
celsius_to_fahrenheit(50.)
#%%
""" Solución: """
def celsios_2_fahrenheit(temp):
    pass
""" Fin de la solución. """
#%%
def nombre():
    """ Introduce tu nombre y apellido, combinalos en una cadena e imprime """
    nombre   = input("Introduce tu nombre: ")
    apellido = input("Ahora tus apellidos: ")
    nombreCompleto = nombre + " " + apellido
    print("Tu nombre es:", nombreCompleto, ".")
```

```
#%%
"""
Ejercicio:
    Amplía la función nombre escrita en clase para incluir la ciudad y el estado.
    Es decir, haz dos preguntas más para obtener la ciudad y el estado en el que
    vives. Imprime de dónde eres en una nueva línea. Pon la coma habitual entre
    ciudad y estado. Para ahorrar tiempo, aquí está la función de inicio.
La corrida debe ser similar a la siguiente:
    Introduzca tu nombre: Pablo
    Introduce tu apellido: Castañeda
    Entra en la ciudad en la que vives: Coyoacán
    Ingresa el estado en el que vives: CdMx

    Tu nombre es: Pablo Castañeda.
    Vives en: Coyoacán, CdMx.
"""
""" Solución: """
def nombre():
    """ Introduce tu nombre y apellido, combinalos en una cadena e imprime """
    nombre   = input("Introduce tu nombre: ")
    apellido = input("Ahora tus apellidos: ")
    nombreCompleto = nombre + " " + apellido
    print("Tu nombre es:", nombreCompleto, ".")
""" Fin de solución. """
#%%
def condicional_if():
    """ Tres maneras ligeramente distintas del condicional if:
        if, if-else (elif), if-elif-else """
    x = 5
    y = 0
    z = 0
    if x > 0:
        print("x es positivo.")
    if y > 0:
        print("y es positivo.")
    else:
        print("y no es positivo.")
    # elif puede ser usado tantas veces como sea necesario
    if z > 0:
        print("z es positivo.")
    elif z < 0:
        print("z es negativo.")
    else:
        print("z debe ser 0.")
#%%
```

```
"""
Python usa '=' para asignar y '==' para comparar. También '!=' es una prueba de
desigualdad. Intenta estos ejemplos:
"""
#%%
x = 5.
y = 5
z = 6
#%%
"""
Ahora tratamos con lo siguiente:
"""
print("x    es    igual a y:", x == y)
print("x  no  es  igual a y:", x != y)
print("x    es    igual a z:", x == z)
print("x  no  es  igual a z:", x != z)
#%%
"""
La siguiente función usa el condicional 'if'. Nota que la identización marca el
rango de acción de las acciones de 'if', 'elif', 'else'.
"""
def area(tipo_, x):
    """ Calcula el área de un cuadrado o de un círculo.
        tipo_ debe ser la cadena "circulo" (sin acento) o "cuadrado". """
    if tipo_ == "circulo":
        area = 3.14*x**2
        print(area)
    elif tipo_ == "cuadrado":
        area = x**2
        print(area)
    else:
        print("Esa figura no la conozco.")
#%%
"""
Ejercicio:
    Escribe una función valorAbsoluto(num) que calcule el valor absoluto de un
    número. Deberás usar el condicional 'if'. Recuerda que si un número es menor
    que cero, debes multiplicarlo por -1 para hacerlo positivo. La salida debe
    ser de la forma
    El valor absoluto de -5  es  5
"""
#%%   # Corridas de prueba
valorAbsoluto(5)
valorAbsoluto(-5)
valorAbsoluto(4-4)
```

```
#%%
""" Solución: """
def valorAbsoluto(num):
    pass
""" Fin de la solución. """
#%%
"""
Ejemplo:
    Los tres ejemplos siguientes funcionan con la instrucción 'input' y señalan
    algunas de las cosas que es posible que debas tener en cuenta al usar una.
    También muestra cómo utilizar la instrucción 'print' sin que se inicie una
    nueva línea en el final de esta instrucción mediante el uso de un argumento
    'end' en ella.
"""
#%%
def fahrenheit_2_celsios1():
    """ ERROR. No comprueba la entrada antes de usarla.
    La entrada del teclado, que siempre es una cadena, a menudo debe ser
    convertida en int o float.
    Convierte temp en Fahrenheit a Celsios.
    """
    temp_str = input("Introduce una temperatura en Fahrentheit: ")
    temp = int(temp_str)
    print("La temperatura en Fahrenheit", temp, "es equivalente a ", end='')
    nuevaTemp = 5*(temp - 32)/9
    print(nuevaTemp,"en grados Celsios.")
#%%
"""
Prueba el problema anterior con una temperatura como 212. También verifica qué
pasa si simplemente das retorno.
"""
#%%
def fahrenheit_2_celsios2():
    """ MEJORADO. Comprobamos la existencia de la entrada antes de usarla.
    La entrada del teclado, que siempre es una cadena, a menudo debe ser
    convertida en int o float.
    Convierte temp en Fahrenheit a Celsios.
    Usa 'if' para asegurar que la entrada existe.
    """
    temp_str = input("Introduce una temperatura en Fahrentheit: ")
    if temp_str:
        temp = int(temp_str)
        print("La temperatura en Fahrenheit", temp, "es equivalente a ", end='')
        nuevaTemp = 5*(temp - 32)/9
        print(nuevaTemp,"en grados Celsios.")
```

```
#%%
"""
Probemos el programa arriba introduciendo 212 grados o simplemente dando retorno.
Vemos la mejora en el programa. ¿Qué sucede si entramos 'a', por ejemplo?
"""
#%%
def fahrenheit_2_celsios3():
    """ UNA MEJORA SUPERIOR. Comprobamos el tipo de entrada antes de usarla.
    La entrada del teclado, que siempre es una cadena, a menudo debe ser
    convertida en int o float.
    Convierte temp en Fahrenheit a Celsios.
    Usa 'if' para verificar que la entrada es un número la usar el método
    ".isdigit()" de las cadenas (strings) y asegurar que está hecha de dígitos.
    """
    temp_str = input("Introduce una temperatura en Fahrentheit: ")
    if temp_str:
        if temp_str.isdigit():
            temp = int(temp_str)
            print("La temperatura en Fahrenheit", temp, "es equivalente a ",
                  end='')
            nuevaTemp = 5*(temp - 32)/9
            print(nuevaTemp,"en grados Celsios.")
        # Prueba la entrada 'a', después quita la identización del 'else'
        else:
            print("Lo siento, necesito un número. ¡Adios!")
#%%
"""
Probemos el programa arriba introduciendo 212 grados o simplemente dando retorno.
Vemos la mejora en el programa. ¿Qué sucede si entramos 'a', por ejemplo? Vemos
realmente una mejora, se pueden hacer muchas más, pararemos con la que está
comentada en el código, aunque se pueda seguir.
"""
#%%
"""
La siguiente función usa la división entera.
"""
#%%
def pulgadas_2_pies1(pulgadas):
    """ Convierte pulgadas a piés y pulgadas """
    pies = pulgadas//12  # división entera, la fracción se elimina en la cuenta
    pulgadasExtra = pulgadas - 12*pies
    print(pulgadas, "pulgadas son", pies,"piés y", pulgadasExtra, "pulgadas.")
#%%
"""
Ejericio:
```

```
294     Reescribe pulgadas_2_pies1(pulgadas) como pulgadas_2_pies2(pulgadas) usando
295     '%' para calcular las pulgadas extras. Recuerda que 19%5 dará 4 como resultado
296     (el resto). Copia y pega la rutina anterior y modifica la parte necesaria.
297 """
298 """ Solución: """
299 def pulgadas_2_pies2(pulgadas):
300     pass
301 """ Fin de la solución. """
302 #%%
303 """
304 El bucle o ciclo 'while'. Los ciclos son usados para repetir las acciones y el
305 rango de esta repetición está indicada por la identización después del condicional
306 'while'.
307 """
308 #%%
309 def dosMas():
310     """ Imprime 2 4 6 8, ¿Qué nos late?... Nota que todo lo que esta en el ciclo
311     está identizado. La primera línea fuera de la identización es la primera línea
312     fuera del ciclo 'while'. """
313     ct = 2
314     while ct <= 8:
315         print(ct, end=" ") # end = " " previene la creación de una nueva línea
316         ct = ct + 2
317     print()                 # Ahora comezamos una nueva línea
318     print("¿Qué nos late?")
319     print("¡El ITAM!")
320 #%%
321 """
322 Ejercicio:
323     Escribe la función regresiva() que comience una cuenta desde 10 y haga la
324     cuenta regresiva hasta el lanzamiento de un cohete, por ejemplo. La salida
325     debe ser 10 9 8 7 6 5 4 3 2 1 ¡Despeguen!
326     Puedes hacer que los números estén todos en la misma línea o en líneas
327     distintas. Usa el ciclo 'while'.
328 """
329 """ Solución: """
330
331 """ Fin de la solución. """
332 #%%
333 """
334 El ciclo 'for'. Este ciclo se usa como un iterador para determinar cuántas veces
335 se tiene que realizar una acción dentro del ciclo. El iterador que usamos abajo
336 es muy útil, éste es 'range(start, stop, step)'.
337 """
338 #%%
```

```
def dosMas2():
    """ Lo mismo que dosMas(), pero que usa el ciclo 'for' y la función 'range()'.
    El rango debe usar un número de comienzo, uno de parada y otro de paso. """
    for ct in range(2, 9, 2):
        print(ct, end=' ') # end = " " previene la creación de una nueva línea
    print()                # Ahora comezamos una nueva línea
    print("¿Qué nos late?")
    print("¡El ITAM!")

#%%
"""
Ejercicio:
    Escribe una función regresiva1() que comience una cuenta desde 10 y haga la
    cuenta regresiva hasta el lanzamiento de un cohete, por ejemplo. La salida
    debe ser 10 9 8 7 6 5 4 3 2 1 ¡Despeguen!
    Puedes hacer que los números estén todos en la misma línea o en líneas
    distintas. Usa el ciclo 'for' y la función 'range()'. El rango debe usar un
    número de comienzo, uno de parada y otro de paso que PUEDE SER NEGATIVO.
"""
""" Solución: """
def regresiva1():
    pass
""" Fin de la solución. """
#%%
"""
Algunos de nuestros ejercicios implican en la búsqueda y corrección de errores
en el código. Aquí hay un ejemplo. ¿Puedes ver los errores (hay dos)? Ten en
cuenta que el editor está señalando una línea con problemas.
Puedes encontrar el error leyendo el ejemplo cuidadosamente, o tratando de
hacerlo funcionar mediante el uso de 'Shift-Enter' para insertar la función en
IPython y leer qué error da, así puedes también intentarlo al ejecutar la función.
"""
#%%
def favorito():
    mi_juguete = input("¿Cuál es tu juguete favorito? ")
     print("Tu juguete favorito es", mi-juguete)
#%%
""" Solución: """

""" Fin de la solución. """
```

Ejercicios2: Se introducen las listas y métodos para ellas. Esta es una primera guía para el uso de matrices y vectores.

```
# -Ejercicios2.py *- coding: utf-8 -*-
```

```
"""
Python tiene listas, es vacía con []. La siguiente es una lista de un el ele-
mento ["a"] y así es [3]. Aquí hay una lista con 3 elementos ["bola", 3.14, -2].
Vamos a definir una lista, lo llamaré lis y haremos cosas con ella para ilustrar
obtener acceso a los elementos de una lista. Ejecute la celda siguiente con
Ctrl+Entrar; Shift+Entrar, además pasa a la siguiente celda.
"""
#%%
lis = ["a","b","c","d","e","f"]
#%%
"""
Ejercicio:
    Algunas de las cosas que podemos hacer con las listas. Probemos.
    lis[0] es el primer elemento de la lista. (el índice es 0)
    lis[1] es el segundo elemento de la lista y así sucesivamente.
La longitud de la lista es len(lis) y es el número de elementos de la lista.
    lis[-1] es el último elemento de la lista.
    lis[2:4] mostrará los elementos 2 y 3 (pero no 4)
    lis[:4] enumerará los elementos 0, 1, 2, 3 (pero no 4); es decir, todos los
            elementos hasta 4
    lis[3:] mostrará todos los elementos a partir del elemento 3.
    lis.append("g") anexará "g" al final de la lista
    "a" in lis # al ejecutar esta instrucción se devolverá True
    "r" en lis # al ejecutar esta instrucción devolverá False
Todo en Python es un objeto, ya sea una variable como x o una lista como lis. Los
objetos tienen métodos indicados por el punto. Así que .append() es un método del
objeto de lista. Veremos más de esto.
"""
"""
Aquí hay una función ejemplo con uso de lista. Pasamos en una lista de artículos
y comprueba si hay ciertos animales o flores en la lista. Lo probaremos en varias
listas como ['oso'], ['margarita', león'], etc.
"""
#%%
def quien_esta_ahi(lis):
    if "oso" in lis:
        print("Hay un oso.")
    if "leon" in lis or "león" in lis:
        print("Hay un león.")
    if "margarita" in lis or "iris" in lis:
        print("Hay flores.")
    if "margarita" in lis and "iris" in lis:
        print("Hay por lo menos dos flores.")
    if "burro" in lis:
        print("Hay un burro.")
```

```
    if "caballo" not in lis:
        print("No hay caballos en la lista.")
    print("En la lista hay",len(lis), "elementos.")
#%%
"""
Haz algunas listas distintas y pásalas a través de 'quien_esta_ahi' para ver
cómo los argumentos son manejados con varias combinaciones.
"""
#%%
unleon = ['león']
ld = ['león','margarita']
lof = ['leon','oso','iris']
#%%
"""
La siguiente función ilustra el uso de ciclos "for". Nota que el ciclo tiene una
variable 'sea' que lleva los pasos a través de la lista, tomando los valores de
cada uno de los elementos de ésta.
"""
#%%
lis = ["a","b","c","d","e","f"]
lis1 = ["a","b","a","r","c","a","a"]
#%%
def cuenta(lis):
    ct = 0
    for sea in lis:
        if sea == 'a':
            ct = ct + 1
    print('En la lista hay',ct,'letras "a".')
#%%
"""
Tengamos en cuenta que hay un patrón de diseño básico para estas listas. Alguna
variable para la cuenta de los resultados (arriba es ct) se inicia antes de entrar
en el ciclo. Esta variable se actualiza dentro del ciclo. Después esa variable es
utilizada (en este caso ct se imprime).
"""
#%%
"""
Ejercicio:
    Toma la siguiente lista, nlis, y calcula su promedio. Es decir, escribe una
    función 'promedio(numlis)' que utiliza un ciclo 'for' para sumar los números
    en numlis y dividir por la longitud de numlis. Sólo para estar seguro de que
    tenemos todos los números en numlis, imprime cada uno en su ciclo 'for' e
    imprime la longitud de la lista. Cuando se utiliza un ciclo, siempre es
    necesario tener cuidado de que se repite tantas veces como se espera. En este
    caso también imprime el número de elementos de la lista. Precaución: NO
```

```
92      utilices la variable nlis en la función. Esta función debe trabajar en
93      cualquier lista de números. Sólo para asegurarse que la función (sin ningún
94      cambio) funciona tanto en rlis como en nlis.
95  """
96  #%%
97  nlis = [2,4,8,105,210,-3,47,8,33,1]                    # el promedio es 41.5
98  rlis = [3.14, 7.26, -4.76, 0, 8.24, 9.1, -100.7, 4]    # el promedio es -9.215
99  #%%
100 # Algunas pruebas para tu función, asegúrate que funciona con estas
101 average(nlis)
102 average(rlis)
103 #%%
104 """ Solución: """
105 def promedio(numlis):
106     pass
107 """ Fin de la solución. """
108 #%%
109 """
110 Enfaticemos que se puede hacer un ciclo 'for' con solo una lista. Uno puede
111 simplemente pasar el paso a través de una lista para formar el ciclo.
112
113 En este caso el ejemplo es una lista de estados y simplemente estaremos caminando
114 a través del ciclo e imprimiendo los estados.
115 """
116 #%%
117 Mexico = ["Nayarit", "Veracruz", "Morelos", "Guerrero", "Sonora", "Monterrey",
118 "Colima", "Quintana Roo"]
119 def para_estados(elis):
120     for estado in elis:
121         print(estado)
122 #%%
123 """
124 Ten en cuenta que un 'for' puede tener el paso definido por distintos iteradores.
125 """
126 #%%
127 """
128 Ejercicio:
129     Escribe una función 'imprime(lis)' que imprima los elementos de la lista lis.
130     Pruébalo ejecutando las tres pruebas que damos. Esto requiere escribir una
131     función que incluya un ciclo como el anterior, pero utiliza lis para el
132     iterador. Adentro, su función debe usar lis para representar la lista. Si lo
133     hace, tu función debe pasar las tres pruebas a continuación.
134 """
135 #%%
136 lista_letras = ['a', 'b', 'c']
```

```
lista_mayusc = ['A', 'B', 'C', 'D']
lista_varios = ['bola', 3.14, -50, 'universidad', "clase"]
#%%
""" Solución: """

""" Fin de la solución. """
#%%
"""
Hablemos de los tipos de datos (type). Para empezar, Python tiene enteros (como,
40), float o números reales (por ejemplo, 40.0), cadenas ("hola"), listas
( ['a','b','c']), bool (booleano, es decir, Verdadero o Falso). En Python se
llaman int, float, str, list, bool. Puede saberse qué tipo es una variable x
escribiendo type(x). A continuación se muestran varios ejemplos.
"""
#%%
x  = 17                              # integer
y  = 3.14                            # float
z  = "The Walrus and the Carpenter"  # string
z1 = "30"                            # string
z2 = '40'                            # string
vocales = ["a", "e", "i", "o", "u"]  # list of strings
numeros = ['1', '2', '3', '3.14']    # list of strings
frases  = ["Las rosas son rojas",
    "Mariachis de graduación"]       # lista d strings (2 strings, solo)
r = True                             # boolean
s = False                            # boolean
#%%
"""
A menudo se puede convertir un tipo a otro (se llama 'cast'): int(z1) crea y
devuelve un entero (30); float(z2) devuelve un número de punto flotante o real
(40.0); str(y) devuelve la cadena "3.14"; etc. Esto es importante porque z1 + z2
no es 70 (es '3040'); sin embargo int(z1) + int(z2) sí es 70. Aquí hay un
programa simple que muestra cuándo es posible que se utilice esta técnica.
"""
#%%
def multiplica():
    numstr1 = input("Introduce un número: ")
    numstr2 = input("Introduce otro: ")
    num1 = float(numstr1)
    num2 = float(numstr2)
    print("Su producto es ", num1 * num2)
#    print("No sirve: ", numstr1 * numstr2)
#%%
"""
Compara list(range(2,20,3)) y range(2,20,3). La primera es una lista y el segundo
```

```
es lo que Python llama un iterador. El segundo de ellos incluye el siguiente
elemento de la lista cada vez que se llama. (En Python 2 había una lista y una
función xrange() para iterar sin generar la lista; esto no está Python 3.)
"""
#%%
print(list(range(2,20,3)))
print(range(2,20,3))
#%%
"""    *** CUIDADO ***
Las siguientes palabras son importantes para Python (y otros lenguajes). Estas
tienen significados especiales y no deben ser usadas como nombres de variables:
    and         del             from        not         while
    as          elif            global      or          with
    assert      else            if          pass        yield
    break       except          import      print
    class       exec            in          raise
    continue    finally         is          return
    def         for             lambda      try
Obtendremos un error de sintaxis.
"""
#%%
except = 5
#%%
"""
Nota que para hacer legible una rutina puede usar estos nombres con un guión bajo
al finalizar; esta es una costumbre arraigada entre la gente de cómputo. Así, por
ejemplo puedes colocar: and_, class_, etc.
Como Python identifica mayúsculas de minúsculas, también un nombre con un ligero
cambio puede hacer toda la diferencia.
"""
#%%
"""
Imprimamos un pequeño reporte. Digamos que tenemos la población de los alumnos
del Itam. Imprimiremos esto en el sentido de una talba. Esesncialmente esto es
como para_estados(lis) que hicimos arriba, pero colocaremos dos elementos a cada
elemento, lo cual es un poco más sofisticado.
"""
#%%
Itam = [["Actuaría", 2395], ["Economía", 6579], ["Matemáticas", 432],
        ["Relaciones Internacionales", 5331]]
#%%
"""
Ejercicio:
    Antes de escribir la función, entendamos que representa una lista de listas.
    Intentemos resolver las siguientes preguntas.
```

```
    ¿Cuál es el primer elemento de Itam? (i.e., aquel con índice 0)
    ¿Cuál es el segundo elemento?
    ¿Cuál es el nombre de la carrera en el segundo elemento? ¿Cómo lo vemos?
    ¿Cuál es el número de estudiantes del tercer elemento?
"""
#%%
""" Solución: """

""" Fin de la solución. """
#%%
Itam = [["Actuaría", 2395], ["Economía", 6579], ["Matemáticas", 432],
        ["Relaciones Internacionales", 5331]]
def reporte(datos):
    """ imprime el reporte de estudiantes """
    print("Estudiantes \t Carrera")
    for grupo in datos:
        print(grupo[1], "\t\t", grupo[0])
#%%
""" Aquí hay otra manera de hacerlo """
def reporte2(datos):
    """ imprime los datos del reporte """
    print("Estudiantes \t Carrera")
    for i in range(0,len(datos)):
        print("{:11}\t {}".format(datos[i][1], datos[i][0]))
#%%
"""
Encuentra la suma de la población total de estudiantes del Itam. Imprime cuántos
hay. Usa un ciclo básico para diseñar la rutina.
"""
#%%
Itam = [["Actuaría", 2395], ["Economía", 6579], [" Matemáticas", 432],
        ["Relaciones Internacionales", 5331]]
def poblacion(datos):
    """ Suma la población de estudiantes """
    suma = 0
    num_carreras = len(datos)
    for i in range(0, num_carreras):
        una_carrera = datos[i]
        pob = una_carrera[1]
        suma = suma + pob
    print("La población total de estudiantes en esta lista es", suma)
    print("Hay un total de",num_carreras,"carreras.")
#%%
"""
Una versión más sutil usando una sintaxis más clara, en el sentido de los nombres
```

```
de las variables. Esto se puede leer mejor en un programa mayor.
"""
#%%
def poblacion(datos):
    """ Suma la población de estudiantes """
    poblacion = 1
    suma = 0
    num_carreras  = len(datos)
    for carrera in range(0, num_carreras):
        suma = suma + datos[carrera][poblacion]
    print("La población total de estudiantes en esta lista es", suma)
    print("Hay un total de",num_carreras,"carreras.")
#%%
"""
Ejericio:
    Escribe una función 'promedio(nlis)' que utilice un ciclo 'for' y 'range()'
    para resumir los números en nlis y dividir por la longitud de nlis. Sólo para
    estar seguros, que se han utilizado todos los números en nlis, imprime cada
    uno en ciclo 'for' e imprime la longitud de la lista. ¡No utilices la variable
    numlis en la función! Si cambias a una lista diferente funcionará? Para
    numlis, la salida debe ser:
        65 44 3 56 48 74 7 97 95 42
        el promedio es de 53.1
"""
#%%
numlis = [65, 44, 3, 56, 48, 74, 7, 97, 95, 42]  # test on this list
numlis2 = [4,6,8,12,2,7,19]     # test on a second list to be sure
#%%
""" Solución: """
def promedio(nlis):
    pass
""" Fin de la solución. """
#%%
"""     *** Libererías/Bibliotecas. ***
Python es un "pequeño" lenguaje de programación en el sentido de que muchas de las
herramientas que existen no están disponibles de una sola vez, hay que llamarlas.
Muchas de éstas son módulos llamados librerías y que pueden ser cargadas en los
programas solo cuando son necesarias; dejando de este modo las rutinas más ligeras
cuando no lo son. Una manera clásica de llamarlas es:
    import random
que cargará la librería llamada 'random'.
"""
#%%
import random
#%%
```

```
# Cada ejecución del siguiente comando te dará un número "real" entre 0 y 1.
print(random.random())
#%%
# Cada ejecución del siguiente comando te dará un número entero entre 3 y 8.
print(random.randint(3,8))
#%%
"""
En el ejemplo siguiente se crea una oración utilizando varias partes del lenguaje.
Elige aleatoriamente palabras de una lista mediante el uso de random.choice(), una
función o método importado de una biblioteca denominada random. Hemos utilizado un
método de string para poner en mayúscula la primera letra en cada oración.
"""
#%%
import random
verbos      = ["va","cocina","sale","finge","mastica","grita","duerme","vuelve"]
sustantivos = ["oso","león","papá","bebe","hermano","carro","patín del diablo",
               "libro","juguete"]
adverbios   = ["dulcemente","suavemente","amargamente","delicadamente",
               "con fuerza","mansamente","en silencio","con sabrosura"]
articulos   = ["un","el","ese","este","aquel","otro"]
def frase():
    art  = random.choice(articulos)
    sust = random.choice(sustantivos)
    verb = random.choice(verbos)
    advb = random.choice(adverbios)
    la_frase = art + " " + sust + " " + verb + " " + advb + "."
    la_frase = la_frase.capitalize()
    print(la_frase)
#%%
"""
Ejercicio:
    Adapta esta función para escribir un poema de cuatro líneas. Llámalo poema().
    Esencialmente tienes que escribir un ciclo alrededor de frase() para obtener
    4 líneas. Recuerda que el interior o el ámbito del ciclo tiene que haber una
    sangría de 4 espacios.
Nota:
    El menú Edición tiene una forma rápida de aplicar sangría a una serie de
    líneas. La función se repite aquí por conveniencia.
"""
#%%
""" Solución (modifica la copia que está abajo para crea poema()): """
#%%
import random
verbos      = ["va","cocina","sale","finge","mastica","grita","duerme","vuelve"]
sustantivos = ["oso","león","papá","bebe","hermano","carro","patín del diablo",
```

```
                "libro","juguete"]
adverbios   = ["dulcemente","suavemente","amargamente","delicadamente",
                "fuertemente","mansamente","silenciosamente","sabrosamente"]
articulos   = ["un","el","ese","este","aquel","otro"]
def poema():
    art  = random.choice(articulos)
    sust = random.choice(sustantivos)
    verb = random.choice(verbos)
    advb = random.choice(adverbios)
    la_frase = art + " " + sust + " " + verb + " " + advb + "."
    la_frase = la_frase.capitalize()
    print(la_frase)
#%%
""" Fin de la solución. """
#%%
"""
Veamos algunos ciclos y sus formas.
"""
"""
Ejemplo:
    Suma números hasta que lleguemos a un cero. Nota que debemos inicializar la
    variable 'suma'.
"""
#%%
def sumando():
    suma = 0
    while True:             # CUIDADO: el ciclo se recorre para siempre
        num = int(input("Introduce un número, con 0 termina la suma: "))
        if num == 0:
            break           # 'breaks' sirve para escapar del ciclo
        suma = suma + num
    print(suma)
```

Ejercicios3: Colocamos el uso de bibliotecas, funciones matemáticas y gráficos. Hay una muy breve introducción al uso de arreglos. Es importante investigar después sobre `ndarray`.

```
# -Ejercicios3.py *- coding: utf-8 -*-

"""
Como hemos visto, Python tiene listas y a partir de ellas podríamos crear métodos
que trabajaran como matrices y vectores. Sin embargo, dado que el álgebra lineal
es una de las herramientas más utilizadas en el cómputo científico, Python nos
permite llamar una biblioteca especializada para este caso:
"""
#%%
```

```
10 import numpy as np
11 #%%
12 """
13 Hemos llamado la librería como np, de este modo, no necesitaremos escribir todo el
14 texto 'numpy' y sucesivamente el método, simplemente será 'np'. La ventaja de
15 haber trabajado con listas, es que la notación es semejante para los vectores y
16 matrices dados en este formato por arreglos (arrays).
17     *** ADVERTENCIA: ***
18     Sólo en este lugar hemos colocado la llamada a la bilbioteca de 'numpy',
19     lo cual quiere decir que si reinicializamos la consola, debemos llamarla
20     nuevamente. (Obtendremos un mensaje de error en caso contrario.)
21     *** ***
22 """
23 #%%
24 """
25 Otra librería bastante útil es aquella que nos permite graficar funciones y las
26 relaciones entre dos vectores, por ejemplo. Ésta es 'matplotlib', en cuyo caso
27 necesitamos un pedazo de ella, 'pyplot'. Por costumbre la llamaremos como 'plt'.
28     import matplotlib.pyplot as plt
29 El uso de la palabra protegida 'lambda' nos permite definir una función en el
30 sentido matemático. De este modo, separado por dos puntos, nos encontramos con los
31 argumentos y la función en sí.
32 """
33 f = lambda x: np.exp(-x**2)          # La exponencial en 'numpy'
34 g = lambda x: (np.sin(x) + 1)/2.0    # El seno en 'numpy'
35 """
36 Dónde evaluaremos las funciones es con un arreglo en un rango, ahora a diferencia
37 de 'range', el paso puede ser un número "real".
38 """
39 x = np.arange(-3, 3, .05)
40 plt.plot(x, f(x), x, g(x))
41 # La graficación, nos permite colocar títulos, leyendas, etc. con sintaxis de
       LaTeX
42 plt.title('$f(x) = \exp(-x^2),\; g(x) = ($sen$(x) + 1)/2$')
43 plt.show()
44 #%%
45 # Otro ejemplo en el intervalo 0 a 10.
46 x = np.arange(0, 10, 0.5)
47 recta1 = lambda x: 1/2.0*(10 - 1*x)
48 recta2 = lambda x: 1/2.0*(10.4 - 1.1*x)
49 plt.plot(x, recta1(x), 'o-', x, recta2(x), '^-')
50 plt.title('Dos rectas semejantes')
51 plt.legend(('$x + 2\,y = 10$', '$1.1\,x + 2\,y = 10.4$'))
52 plt.grid(True)    # Podemos colocar la malla para ver más detalle, tal vez.
53 plt.show()
```

```
#%%
"""
Para el ejemplo de la primera tarea, también podemos definir las funciones que
utilizaremos con el uso de 'lambda'.
"""
f = lambda x: x/(x*(x - 1.0))
g = lambda x: 1.0/(x - 1.0)
h = lambda x: (x - 1.0)*(x - 2.0)/(x - 1.0)
# Recordamos cómo construir el épsilon de la máquina y el infinito.
eps = 1.0/2**52
inf = 2.0 * 2.0**1023
#%%
"""
Ejercicio:
Construye un arreglo con los números [0.0, eps, 1.0, inf] y evalua las funciones
en estos valores.
"""
"""
Solución:
"""

"""
Fin de la solución.
"""
#%%
"""
La construcción de los 'ndarrays' o arreglos n-dimensionales de 'numpy' son la
guia para trabajar con vectores y matrices. Hagamos algunos ejemplos.
"""
#%%
# Primer ejemplo
arreglo1 = np.array([1, 3, 6, 7])
print(type(arreglo1))
print(arreglo1)
#%%
# Segundo ejemplo
arreglo2 = np.ndarray((3, 3))
print(type(arreglo2))
print(arreglo2)
#%%
"""
Observamos que el tipo de los dos arreglos es de la clase 'numpy.ndarray' aunque
el primero sea un simple arreglo. En el segundo caso, corre varias veces la
rutina, verás que los números del arreglo pueden cambiar. En este caso, sólo
estamos diciendo que el arreglo2 está conformado por 3 renglones y 3 columnas,
```

```
pero no hemos colocado información en su interior.

El siguiente ejemplo utiliza 'numpy.arange(n)' construyendo el arreglo desde el
valor 0 hasta n (sin incluirlo), vimos otra forma de usarlo con los gráficos.
"""
#%%
# Tercer ejemplo
arreglo3 = np.arange(10)
print(arreglo3)
#%%
"""
Como llamado de atención es que un 'ndarray' es como una lista, pudiendo guardar
elementos de distinto tipo, sin embargo, el arreglo entero es de un único tipo,
por lo cual colocará todos los elementos en el tipo más incluyente, mira por
ejemplo los siguientes ejemplos.
"""
#%%
# Cuarto arreglo con varios tipos
arreglo4 = np.array([0, 1, True, False, 2.3])
print(arreglo4)
# Quinto arreglo con varios tipos
arreglo5 = np.array([0, 1, True, False, 2.3, "Hola"])
print(arreglo5)
#%%
"""
Los booleanos y enteros son un subconjunto de los números en aritmética de punto
flotante, pero las cadenas son una clase mayor.
"""
#%%
"""
Ejercicio:
Los índices se colcan como en las litas. Construye un arreglo con 10 elemetos y
destaca el primer elemento, el último, del primero al quinto y del quinto al fin.
"""
"""
Solución:
"""

"""
Fin de la solución.
"""
#%%
"""
Con arreglos en varias dimensiones, los índices tienen dos maneras de ser
utlizados: al extraer un arreglo de dimensión menor o respetando cada dimensión
```

```
"""
matriz = np.array([[1, 2, 3],[4, 5, 6]])
print(matriz)
print(matriz[0])      # Primer elemento, un arreglo
print(matriz[0][0])   # Primer elemento del primer elemento
print(matriz[0, 0])   # Entrada primera de cada elemento
#%%
"""
Ejercicio:
Cada modo tiene sus particularidades y ventajas, intenta las siguientes líneas de
código y después consigue imprimir la última columna o el elemento con el valor 5.
"""
#%%
print(matriz[0][:2])
print(matriz[:][:1])
print(matriz[:][:2])
print("Ahora con índices de matrices:")
print(matriz[0, :2])
print(matriz[:, :1])
print(matriz[:, :2])
#%%%
"""
Solución:
"""

"""
Fin de la solución.
"""
#%%
"""
Con los ejemplos arriba, debería de ser claro que usar el corchete con la coma es
más intuitivo y es en general lo que haremos.
"""
"""
Solo un ejemplo con un arreglo 3 dimensional, podríamos hacer con el número de
dimensiones que querramos. Mira el número de corchetes y la forma en que Python
imprime el arreglo.
"""
#%%
arreglo3d = np.array([[[0, 1, 2], [3, 4, 5], [7, 8, 9]],
                      [[1, 0, 0], [0, 1, 0], [0, 0, 1]]])
print(arreglo3d)
print("\nEl primer elemento...")
print(arreglo3d[0])
print("... de dos formas")
```

```
189 print(arreglo3d[0][:][:])
190 #%%
191 """
192 Algunos métodos interesantes para un 'ndarray' son los siguientes
193 """
194 matriz = np.array([[1, 2, 3],[4, 5, 6]])
195 print(matriz)
196 print("La forma de la matriz es:", matriz.shape)
197 print("   su tamaño es entonces:", matriz.size)
198 print("Su transpuesta:")
199 print(matriz.T)
200 print("La podemos escribir como un vector:")
201 print(matriz.flatten())
202 print("O manualmente darle otro formato:")
203 print(matriz.reshape(6, 1))
204 print("Además podemos evaluar su suma igual a {:5}".format(np.sum(matriz)))
205 print("       el producto de sus elementos es {:5}".format(np.prod(matriz)))
206 print("                o la media de ellos es {:5}".format(np.mean(matriz)))
207 #%%
208 """
209 Las operaciones entre matrices se hacen elemento a elemento, en la suma esto es lo
210 común en el caso de la multiplicación o la división esto se conoce como las
211 operaciones de Hadamard. Observa los siguientes resultados y descubre que es lo
212 que ocurre cuando la operación es con un escalar.
213 """
214 #%%
215 A = np.ones([3, 3])    # Una matriz con todos sus elementos iguales a 1
216 print("A =\n", A)
217 B = np.eye(3)          # La matriz identidad de orden 3
218 print("B =\n", B)
219 A = 2*A
220 B = 3*B
221 print("A =\n", A)
222 print("B =\n", B)
223 print(A + B)
224 print(A - B)
225 print(B == 0)
226 #%%
227 """
228 Observamos el producto de Hadamard y su división.
229 """
230 #%%
231 print("Producto:")
232 print(A*B)
233 print("División:")
```

```
print(B/A)
print("División entera:")
print(B//A)
print("Módulo:")
print(B%A)
#%%
"""
En el caso del producto entre matrices, tenemos dos formas de hacerlo, con el
producto punto (o interior) o con el uso de la aplicación '@'. Ambos formatos
realizan la misma operación.
"""
#%%
print("Producto punto:")
print(np.dot(A, B))
print("Con la aplicación 'at':")
print(A@B)
#%%
"""
Ejercicio:
Considera la matrices A y B arriba, copia la matriz B en una nueva matriz C,
altera su elemento central con
C[1, 1] = -3
y evalua la diferencia entre A@B y A@C. ¿Obtienes lo que esperas? ¿Qué sucede?
"""
"""
Solución:
"""

"""
Fin de la solución.
"""
#%%
"""
En Python algunos elementos son simplemente referenciados a la memoria, es por
ello que cuando alteraste la entrada de C, la entra de B también cambió. Tenemos
que tener extremo cuidado con esto. La solución es hacer una copia de B en C con
el comando 'copy', de este modo, B y C apuntarán a distintas partes de la
memoria y serán independientes.
"""
#%%
A = 2*np.ones([3, 3])
B = 3*np.eye(3)
C = B.copy()
C[1][1] = -3
print("Las matrices\nA =\n", A)
```

```
print("B =\n", B)
print("C =\n", C)
print("Por lo tanto A@B - A@C es:\n",A@B - A@C)
#%%
"""
Algunos otros 'ndarray' importantes son los siguientes, juega con ellos para
entender qué es lo que hace cada uno de ellos.
"""
#%%
x = np.linspace(0, 3, 10)
y = np.logspace(0, 3, 10)
plt.plot(x, y)
plt.show()
cero = np.zeros((2, 3))
unos = np.ones((2, 3))
print("Con ceros y unos:")
print(cero*unos)
print(cero +  unos)
print("Con la diagonal y la identidad:")
diag = np.diag([1, 2], 1)
iden = np.eye(3)
print(diag@iden)
print(diag*iden)
print(diag + iden)
```

Bibliografía

[1] U.M. Ascher, C. Grief, *A First Curse in Numerical Methods*. Computational Science and Engineering Series, SIAM Press.

[2] R.L. Burden & J.D. Faires, *Análisis Numérico*, International Thompson Edition.

[3] M.B. Giles & R. Carter (2006) "Convergence analysis of Crank-Nicolson and Rannacher time-marching", *Journal of Computational Finance* **9**: 89–112.

[4] I. Gladwell, J.G. Nagy & W.E. Ferguson (2008) *Introduction to Scientific Computing using Matlab*. (Disponible en http://www.mathcs.emory.edu/~ale/NAbook_Aug_2008.pdf)

[5] M.T. Heath, *Scientific Computing. An Introductory Survey*. McGraw-Hill.

[6] D.J. Higham & N.J. Higham, *Matlab Guide*, SIAM.

[7] C.E. Mejía Salazar, *Invitación al Análisis Numérico. (Matlab con métodos numéricos)*. (Disponible en https://www.medellin.unal.edu.co/~cemejia/doc/an23.pdf)

[8] D.P. O'Leary, *Scientific Computing with Case Studies*, SIAM.

[9] J. Stoer & R. Bulirsch, *Introduction to Numerical Analysis*. Springer-Verlag.

[10] G. Strang, *Linear Algebra and its Applications*. Cengage Learning.

[11] C.F. van Loan, *Introduction to Scientific Computing*, Prentice-Hall.

[12] C.F. van Loan & K.Y. Daisy Fan, *Insight Through Computing: A Matlab Introduction to Computing Science and Engineering*, SIAM.

[13] G.H. Golub & C.F. van Loan, *Matrix Computations*. The Johns Hopkings University Press.

[14] G.W. Stewart, *Afternotes on Numerical Analysis*. SIAM.

[15] *Numerical Recipes in C: the art of Scientific Computing*, W. H. Press, Cambridge University Press.