Análisis Numérico para Ingeniería

Michael Karkulik
michael.karkulik@usm.cl

22 de diciembre 2020

Contents

Chapter 1

Introducción al Análisis Numérico

Según Wikipedia, **El análisis numérico o cálculo numérico es la rama de las matemáticas encargada de diseñar algoritmos para, a través de números y reglas matemáticas simples, simular procesos matemáticos más complejos aplicados a procesos del mundo real.**

Es decir, el objetivo de análisis numérico es diseñar métodos que podemos *implementar como algoritmos* en maquinas (computadoras) para *simular* fenomenos reales. Las necesidades de simular fenomenos reales en maquinas son varias:

- Predecir el futuro de sistemas complejos como el tiempo, mercado financieros, o redes digitales.
- Cuando un experimento real es muy caro o no factible. Ejemplos son simulaciónes de choques de autos, simulaciones de aerodinamica de aviones, simulaciónes moleculares para estudiar el efecto de un nuevo medicamento.
- Cuando la cantidad de interes no es accesible (*problemas inversos*). Por ejemplo, en una tomografía se toma varias imagenes con rayos X, y el imagen final se calcula con una reconstrucción numérica.

Para llegar de un fenomeno real a una simulación calculada en una maquina, tenemos que pasar por tres fases:

$$\boxed{\text{Realidad}} \rightarrow \boxed{\text{Modelo matemático}} \rightarrow \boxed{\text{Algoritmo numérico}} \rightarrow \boxed{\text{Simulación}}$$

Vamos a analizar cada fase.

La fase $\boxed{\textbf{Realidad}} \rightarrow \boxed{\textbf{Modelo matemático}}$

Los modelos matemáticos son descripciones matemáticas de fenómenos reales. La siguiente tabla contiene algunos fenomenos del mundo real y los modelos que se usan hoy en día.

Realidad	Modelo matemático
Choque de auto Estabilidad de un puente Terremoto vs. edificio	Ecuaciones de elasticidad
Aerodinámica de un avión	Ecuaciones de Navier-Stokes
Electromagnetismo	Ecuaciones de Maxwell
Matemática financiera Precios de activos	Ecuación de Black-Scholes

Muchas veces, los modelos son ecuaciónes diferenciales ordinarias (EDO) o parciales (EDP). Por ejemplo, la ecuación de Navier-Stokes es una EDP no lineal

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} - \nu \nabla^2 \mathbf{u} = -\nabla w + \mathbf{g}, \tag{1.1}$$

donde $\mathbf{u}$ es la velocidad del fluido. Modelos son simplificaciones de la realidad. Es decir, en esta fase introducimos el **error de modelación**, por ejemplo a taves de

- Cuerdas sin masa.
- Modelos lineales para efectos no lineales.
- ¿El mundo es discreto o continuo?
- Teoría cuántica $\leftrightarrow$ Teoría de la relatividad.

La fase Modelo matemático $\rightarrow$ Algoritmo numérico

En general no es posible determinar una solución analítica (es decir, una fórmula explícita) del modelo, o es muy difícil encontrar una. Por ejemplo, es practicamente imposible obtener una formula para la función $\mathbf{u}$ que resuelve la ecuación de Navier-Stokes (1.1). Hay modelos matemáticos bien simples que ya no permiten soluciónes analíticas. Por ejemplo, se sabe que las raices del polinomio $ax^2 + bx + c = 0$ de grado 2 están dadas por

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Formulas análogas (pero mas complejas) para polinomios de grado 3 y 4 existen. Sin embargo, un resultado importante de la teoría de Galois expresa que no existen formulas basicas (formulas que contienen $+, -, \cdot, \backslash, \sqrt{\cdot},$) para calcular las raices de un polinimio general de grado 5 o mas alto. Las raices existen según el teorema fundamental de Algebra, pero no tenemos formulas para calcularlas. Concluimos que si queremos calcular dichas raizes, entonces nuestra única opción es desarollar métodos para *aproximarlas*. Por ejemplo, un método para aproximar raices de una

función f es el *método de Newton*: Dado un punto inicial x_0, el método de Newton calcula una sucesión $(x_j)_{j=0}^{\infty}$ a traves de la formula recursiva

$$x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)}, \quad j = 0, 1, 2, \ldots.$$

Dadas ciertas condiciones, la sucesión $(x_j)_{j=0}^{\infty}$ converge a una raíz de f, es decir $\lim_{j\to\infty} x_j = x^*$ con $f(x^*) = 0$. Obviamente no podemos calcular toda la sucesión $(x_j)_{j=0}^{\infty}$ por falta de tiempo. En algún momento tenemos que terminar los calculos, y asi calculamos a lo más J terminos, $(x_j)_{j=0}^{J}$. El último elemento x_J será nuestra *aproximación* a la solución excta x^*. En general, $x_J \neq x^*$, pero suficientemente cerca, y eso muchas veces es todo lo que necesitamos.

Por otro lado hay modelos matemáticos que podemos resolver explicitamente *en teoría*, pero el procedimiento de resolución es muy difícil o largo. Por ejemplo, si A es una matriz regular, entonces podriamos resolver el sistema $Ax = b$ explicitamente. Sin embargo, si $A \in \mathbb{R}^{1000\times1000}$ eso será poco practicable, y usaremos una computadora para calcular una aproximación a x con algoritmos numéricos.

La fase **Algorito numérico** $\to$ **Simulación**

Los algoritmos no los vamos a ejecutar a mano, sino en *maquinas*, es decir

- calculadoras,
- computadoras (PC, Notebook),
- servidores con miles de nucleos,
- tarjetas gráficas.

Esta fase tiene que ver entonces con

- entender como funciona la *hardware* de una maquina,
- manejar lenguajes de programación: Matlab, Python, Fortran, C, C++, etc.,
- computación paralela,
- high performance computing.

El objetivo de un curso básico de Análisis Numérico

En este curso estudiaremos principalmente la segunda fase, es decir, vamos a desarollar métodos numéricos para problemas matemáticos. Los problemas matemáticos que vamos a considerar son problemas simples de cálculo, algebra lineal, y ecuaciones diferenciales. Como ya explicamos, la

solución de un método numérico es una *aproximación* a la solución exacta, es decir, lleva un error. En teoria, los métodos que desarollamos deberian ser capaces de llegar a cualquier precisión en caso de tener suficientes recursos computacionales y tiempo. En la practica uno prefiere métodos *eficientes*, es decir, métodos que obtienen la mejor precisión posible con la menor cantidad de recuros posibles. Por lo tanto, al desarollar métodos numéricos nos interesa el **error** del método y también el **costo computacional**. En el capítulo 3 estudiaremos brevemente la primera y la tercera fase. Especificamente, estudiaremos los **errores** que introducen.

Literatura

Hay una multitud de libros de análisis numérico o computación científica, abordando el tema de diferentes angulos. Para la confección del material de este curso recurrimos a los siguientes libros:

- **S. Chapra, R. Canale**, *Métodos numéricos para ingenieros*, Quinta edición, McGrawHill, 2006.
- **W. Dahmen, A. Reusken**, *Numerik fuer Ingenieure und Naturwissenschaftler*, Zweite Auflage, Springer, 2006
- **M. Heath**, *Scientific Computing, A Introductory Survey*, McGrawHill, 1997.
- **M. Holmes**, *Introduction to Scientific Computing and Data Analysis*, Springer, 2016.
- **H. Langtangen**, *A primer on scientific programming with Python*, Fifth Edition, Springer, 2016.
- **A. Quarteroni, F. Salieri**, *Scientific Computing with Matlab and Octave*, Second edition, Springer, 2006.

Programación

La parte computacional y de programación de este curso se llevará a cabo usando `Python`. Para correr los códigos de este curso es suficiente tener

- Python[1] (El paquete básico. La versión 3.7.5 es suficiente, pero seguramente no necesario)
- NumPy[2] (Para cálculos con matrices y vectores)
- Matplotlib[3] (Para graficar datos)

[1] `www.python.org`
[2] `www.numpy.org`
[3] `www.matplotlib.org`

Todos los códigos y ejercicios computacionales del curso son disponibles como `jupyter notebooks`[4]. Se puede instalar todas las herramientas de forma separada, pero la solución mas fácil es instalar `Anaconda`[5], disponible para las plataformas Linux, Mac, Windows.

[4] `www.jupyter.org`
[5] `www.anaconda.com/products/individual`

Chapter 2

Ecuaciones y sistemas no lineales

Ejemplo 1. *Según la ley de la gravitación universal formulada por Isaac Newton, el valor absoluto de la fuerza entre dos masas m_1 y m_2 (en kg), separadas por una distancia r, es*

$$F = G\frac{m_1 m_2}{r^2},$$

donde G es la constante de gravitación universal, $G = 6,67 \cdot 10^{-11} Nm^2/kg$. Consideramos un campo gravitacional en el plano generado por tres masas puntuales m_1, m_2, y m_3 con coordenadas $(x_1, y_1) = (x_1, 0)$, $(x_2, y_2) = (x_2, 0)$, y $(x_3, y_3) = (0, y_3)$. El objetivo es buscar el punto (x, y) tal que para una masa puntual m con coordenada (x, y) las tres fuerzas gravitacionales entre m y m_k, $k = 1, 2, 3$, estén equilibradas.

La fuerza $\mathbf{F}_k$ entre m y m_k está dada por

$$\mathbf{F}_k = G\frac{m_j m}{r_k^2} \cdot \frac{(x_k, y_k) - (x, y)}{r_k}, \qquad \text{donde } r_k = \left((x - x_k)^2 + (y - y_k)^2\right)^{1/2}.$$

y la condición de equilibrio significa nada mas que $\mathbf{F}_1 + \mathbf{F}_2 + \mathbf{F}_3 = \mathbf{0}$. La última expresion contiene dos ecuaciones en dos desconocidas,

$$Gm\sum_{k=1}^{3} \frac{m_k(x_k - x)}{((x - x_k)^2 + (y - y_k)^2)^{3/2}} = 0, \qquad Gm\sum_{k=1}^{3} \frac{m_k(y_k - y)}{((x - x_k)^2 + (y - y_k)^2)^{3/2}} = 0. \tag{2.1}$$

□

En sistemas lineales $Ax = b$ con una matriz $A \in \mathbb{R}^{n\times n}$, el lado izquierdo depende de manera **lineal** de las desconocidas $x = (x_1, \ldots, x_n)$, es decir $A(x+y) = Ax + Ay$. Métodos para resolver sistemas lineales, como la eliminación de Gauss, hacen uso de esta linealidad. Sin embargo, las ecuaciones (2.1) dependen de forma **no lineal** de las desconocidas (x, y), y no tendremos la opción de resolverlas tan facilmente. El objetivo del presente capítulo es presentar métodos para

resolver ecuaciones y sistemas no lineales. Por el resto del presente capítulo, consideramos el problema de resolver una ecuación no lineal o un sistema de ecuaciones no lineales:

Dadas n funciones $f_j : \mathbb{R}^n \to \mathbb{R}$, $j = 1, \ldots, n$, hallar $x^* = (x_1^*, \ldots, x_n^*) \in \mathbb{R}^n$ tal que

$$f(x^*) = f(x_1^*, \ldots, x_n^*) = \begin{bmatrix} f_1(x_1^*, \ldots, x_n^*) \\ f_2(x_1^*, \ldots, x_n^*) \\ \vdots \\ f_n(x_1^*, \ldots, x_n^*) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{2.2}$$

En (2.2) tenemos n ecuaciones en n desconocidas. Si no hay la misma cantidad de ecuaciones y desconocidas, los métodos que presentamos en este capítulo no se aplican. El problema (2.2) es la forma mas general: cualquier ecuación o sistema puede ser reescrito en la forma (2.2), pues, $f(x) = g(x)$ es equivalente a $f(x) - g(x) = 0$.

Ejemplo 2. *(1) El caso $n = 1$: Un ejemplo de una ecuación no lineal en una desconocida es*

$$f(x) = x^3 - 2\cos(x) = 0.$$

(2) El caso $n = 2$: Un ejemplo de un sistema de dos ecuaciones en 2 desconocidas es

$$\begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_1^4 - 2x_2 + 0.5 \\ -x_1 + x_2^2 - \sin(x_2) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

(3) Si $A \in \mathbb{R}^{n \times n}$ es una matriz y $b \in \mathbb{R}^n$, entonces el sistema lineal $Ax = b$ es equivalente a $Ax - b = 0$, es decir, de la forma (2.2) con $f(x) = Ax - b$.

□

Un sistema lineal cuadrado con matriz regular siempre tiene única solución. En el caso de ecuaciones o sistemas no lineales, la existencia y unicidad de soluciones suele ser mas complicado:

(1) $2 + e^x = 0$ no tiene solución en $\mathbb{R}$.

(2) $e^{-x} - x = 0$ tiene una solución en $\mathbb{R}$.

(3) $x^2 - 4\sin(x) = 0$ tiene dos soluciones en $\mathbb{R}$.

(4) $\cos(x) = 0$ tiene una infinita canditad de soluciones en $\mathbb{R}$.

2.1 Ecuaciones no lineales

Vamos a empezar con el caso $n = 1$, es decir, estamos buscando una raíz $x^* \in \mathbb{R}$ de una función real de una variable $f : \mathbb{R} \to \mathbb{R}$,

$$\text{hallar } x^* \text{ tal que } f(x^*) = 0. \tag{2.3}$$

2.1.1 El método de bisección

Un método simple pero muy robusto es el método de bisección. Supongamos que f es continua, y que $a_0 < b_0$ son dos puntos donde

$$f(a_0) \cdot f(b_0) < 0,$$

es decir, $f(a_0)$ y $f(b_0)$ tienen signos opuestos. Por el teorema del valor intermedio existe una raiz x^* de f con $a_0 < x^* < b_0$. Definimos $x_0 := (a_0 + b_0)/2$, entonces hay tres posibilidades:

(1) $f(x_0) = 0$. Es decir, encontramos una raiz de f.

(2) $f(x_0) \cdot f(a_0) < 0$. Es decir, f tiene una raiz en (a_0, x_0). Definimos $a_1 := a_0$ y $b_1 := x_0$.

(3) $f(x_0) \cdot f(b_0) < 0$. Es decir, f tiene una raiz en (x_0, b_0). Definimos $a_1 := x_0$ y $b_1 := b_0$.

En los casos (2) y (3) aplicamos el mismo argumento a los puntos a_1 y b_1. Sea $x_1 := (a_1 + b_1)/2$.

(1) $f(x_1) = 0$. Es decir, encontramos una raiz de f.

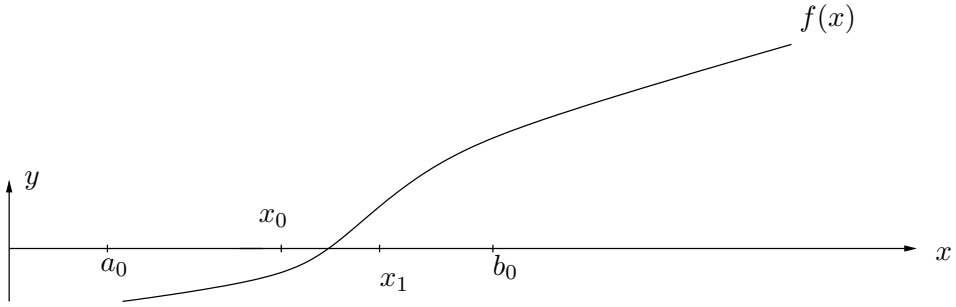
(2) $f(x_1) \cdot f(a_1) < 0$. Es decir, f tiene una raiz en (a_1, x_1). Definimos $a_2 := a_1$ y $b_2 := x_1$.

(3) $f(x_1) \cdot f(b_1) < 0$. Es decir, f tiene una raiz en (x_1, b_1). Definimos $a_2 := x_1$ y $b_2 := b_1$.

Replicando este procedimiento calculamos una sucesión de intervalos

$$[a_0, b_0] \supset [a_1, b_1] \supset [a_2, b_2] \supset \dots.$$

y puntos $x_k \in (a_k, b_k)$. En la Figura 2.1 ilustramos el procedimiento.



El método de bisección.

Teorema 3. *Sea $f : [a_0, b_0] \to \mathbb{R}$ continua y $f(a_0) \cdot f(b_0) < 0$. Entonces el método de bisección genera una sucesión de intervalos $([a_k, b_k])_{k=0}^{\infty}$, tal que con $x_k := (a_k + b_k)/2$ se tiene*

(i) $\lim_{k\to\infty} x_k = x^*$ *con* $f(x^*) = 0$,

(ii) $|x^* - x_k| \leq \frac{b_0 - a_0}{2^{k+1}}$.

Idea de la demostración. El error $e_k := |x_k - x^*|$ es menor que la mitad de la longitud del intervalo $[a_k, b_k]$, es decir $e_k \leq \frac{b_k - a_k}{2}$. Notamos que la longitud de un intervalo $[a_{k+1}, b_{k+1}]$ es la mitad de la longitud del intervalo anterior $[a_k, b_k]$, es decir, despues de k pasos la longitud se redujo por un factor de 2^k, $b_k - a_k \leq \frac{b_0 - a_0}{2^k}$. Concluimos

$$e_k \leq \frac{b_k - a_k}{2} \leq \frac{b_0 - a_0}{2^{k+1}}. \tag{2.4}$$

□

En la practica ejecutamos el método de bisección en una maquina, y claramente es imposible calcular toda la sucesión $(x_k)_{k=0}^{\infty}$, pues, es una infinita sucesión de números. Lo que necesitamos es un *criterio de cancelación*: Consideramos el error $e_k := |x^* - x_k|$ y fijamos cierta tolerancia tol. ¿Cuantos pasos tenemos que hacer para llegar a un error menor que la tolerancia,

$$e_k \leq \text{tol?} \tag{2.5}$$

La cota superior de la desigualdad (2.4) es calculable. Es decir, no depende de la solución x^*, pues, contiene solamente cantidades conocidas. Entonces, para garantizar (2.5) es suficiente tener

$$\frac{b_0 - a_0}{2^{k+1}} \leq \text{tol},$$

es decir

$$\log_2 \left(\frac{b - a}{\text{tol}} \right) - 1 \leq k.$$

La última expresión nos entrega un *criterio de cancelación*: Si queremos obtener un error con cierta tolerancia, tenemos que hacer el número de pasos indicados. Comparado con métodos que vamos a conocer mas adelante, el método de biseccion no es muy rapido, pues, la única información que usa de la función f son sus signos y no sus valores. También por eso es muy robusto, y en la practica se usa entonces para calcular puntos iniciales para métodos que son mas rapidos pero menos robustos.

El método de biseccion lo podemos implementar en Python de la siguiente manera.

```
In [1]: def miBiseccion(f,a,b,tol,N):
          from math import log

          if f(a)*f(b) >= 0:
            print("Error: No se puede garantizar raiz de f")
            return 0

          ktol=log((b-a)/tol-1,2)

          k=0

          while ( (k < N) & (k < ktol) ):
            x = (a+b)/2
            print('k =',k,', x_k =',x)
            if ( f(a)*f(x) < 0 ):
              b = x
            else:
              a = x
            k+=1

          return x
```

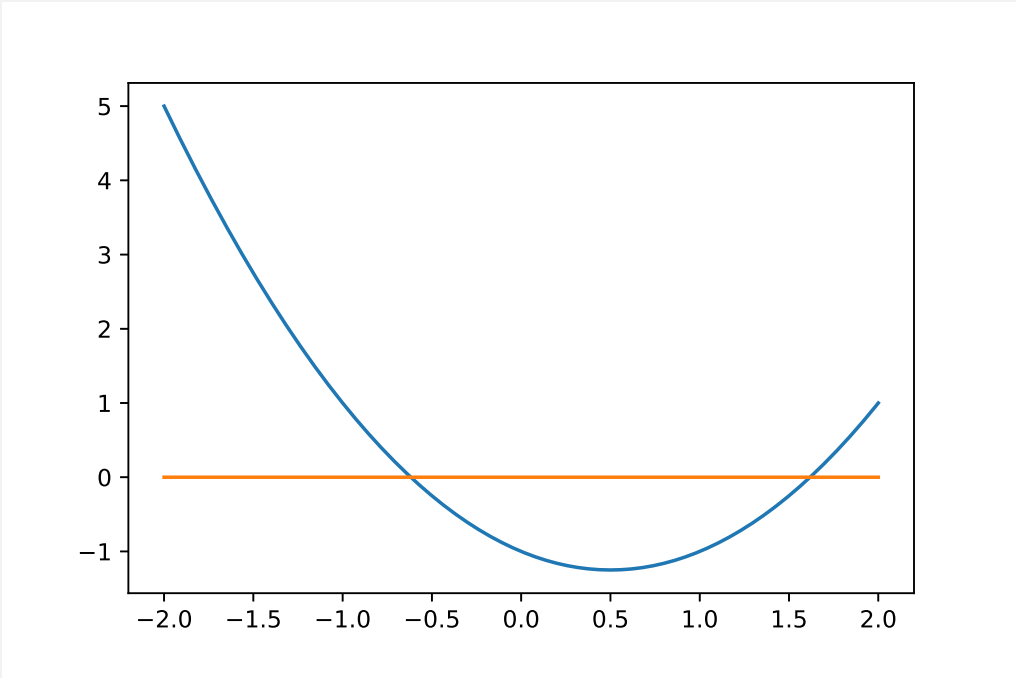
Ejercicio: Aproximamos primero el valor de $\sqrt{2}$ como raíz de $f(x) = x^2 - 2$. Sabemos que $\sqrt{2}$ es la única raíz en el intervalo $(1, 2)$, y $f(1)f(2) < 0$. Es decir, usando $a_0 = 1, b_0 = 2$, el método de bisección convergerá a $\sqrt{2}$. Según lo establecido antes, podemos garantizar un error de 10^{-6} despues de $\log_2(10^6) - 1 \approx 18.9316$ pasos, es decir, 19 pasos.

```
In [2]: f = lambda x: x**2-2
        x=miBiseccion(f,1,2,0.000001,30)
        from math import sqrt
        x-sqrt(2)

k = 0 , x_k = 1.5
k = 1 , x_k = 1.25
k = 2 , x_k = 1.375
k = 3 , x_k = 1.4375
k = 4 , x_k = 1.40625
k = 5 , x_k = 1.421875
k = 6 , x_k = 1.4140625
k = 7 , x_k = 1.41796875
```

```
k = 8 , x_k = 1.416015625
k = 9 , x_k = 1.4150390625
k = 10 , x_k = 1.41455078125
k = 11 , x_k = 1.414306640625
k = 12 , x_k = 1.4141845703125
k = 13 , x_k = 1.41424560546875
k = 14 , x_k = 1.414215087890625
k = 15 , x_k = 1.4141998291015625
k = 16 , x_k = 1.4142074584960938
k = 17 , x_k = 1.4142112731933594
k = 18 , x_k = 1.4142131805419922
k = 19 , x_k = 1.4142141342163086


Out[2]: 5.718432134482754e-07
```

Para obtener una primera idea como elegir el intervalo inicial (a_0, b_0), podemos graficar la función. Por ejemplo, el número áureo se define como raíz del polinomio $p(x) = x^2 - x - 1$. Para grafica p, podemos usar funciones de los modulos `matplotlib` y `numpy`.

```
In [4]: from numpy import linspace, zeros
        from matplotlib.pyplot import plot, show, savefig

        def p(x):
            return(x**2-x-1)

        t = linspace(-2,2,1000) # vector de 1000 puntos equiespaciados en [-10,10]
        y = zeros(len(t)) # declarar la memoria para el vector de las evaluaciones de p
        z = zeros(len(t))

        for i in range(0,1000,1):
            y[i] = p(t[i])

        plot(t,y,t,z)
        savefig("biseccion_plot1.eps")
        show()
```



El número áureo es la raíz positiva, y podemos usar $(1, 2)$ como intervalo inicial.

```
In [5]: miBiseccion(p,1,2,0.000001,30)

k = 0 , x_k = 1.5
k = 1 , x_k = 1.75
k = 2 , x_k = 1.625
k = 3 , x_k = 1.5625
k = 4 , x_k = 1.59375
k = 5 , x_k = 1.609375
k = 6 , x_k = 1.6171875
k = 7 , x_k = 1.62109375
k = 8 , x_k = 1.619140625
k = 9 , x_k = 1.6181640625
k = 10 , x_k = 1.61767578125
k = 11 , x_k = 1.617919921875
k = 12 , x_k = 1.6180419921875
k = 13 , x_k = 1.61798095703125
k = 14 , x_k = 1.618011474609375
k = 15 , x_k = 1.6180267333984375
k = 16 , x_k = 1.6180343627929688
```

```
k = 17 , x_k = 1.6180305480957031
k = 18 , x_k = 1.618032455444336
k = 19 , x_k = 1.6180334091186523


Out[5]: 1.6180334091186523
```

Para aproximar la raíz negativa de p, podemos usar el intervalo inicial $(-1, 0)$:

```
In [6]: miBiseccion(p,-1,0,0.000001,30)

k = 0 , x_k = -0.5
k = 1 , x_k = -0.75
k = 2 , x_k = -0.625
k = 3 , x_k = -0.5625
k = 4 , x_k = -0.59375
k = 5 , x_k = -0.609375
k = 6 , x_k = -0.6171875
k = 7 , x_k = -0.62109375
k = 8 , x_k = -0.619140625
k = 9 , x_k = -0.6181640625
k = 10 , x_k = -0.61767578125
k = 11 , x_k = -0.617919921875
k = 12 , x_k = -0.6180419921875
k = 13 , x_k = -0.61798095703125
k = 14 , x_k = -0.618011474609375
k = 15 , x_k = -0.6180267333984375
k = 16 , x_k = -0.6180343627929688
k = 17 , x_k = -0.6180305480957031
k = 18 , x_k = -0.6180324554443359
k = 19 , x_k = -0.6180334091186523


Out[6]: -0.6180334091186523
```

Ejercicio: Cada año depositamos d unidades de dinero en un fondo con interes anual r. Despues de n años, nuestra inversión se suma a

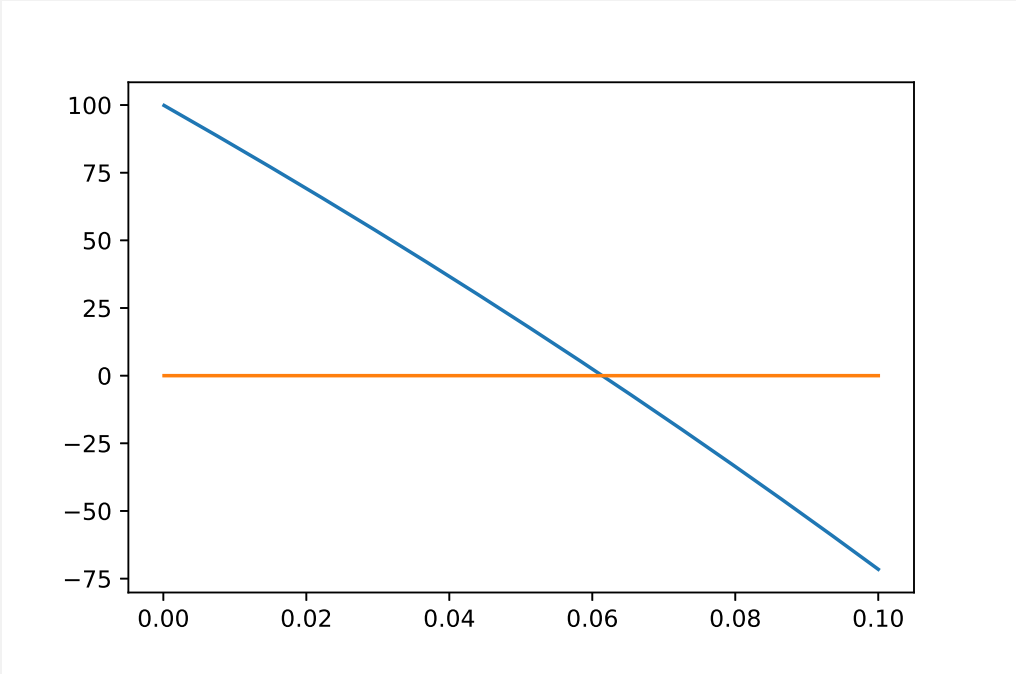
$$D = \sum_{j=1}^{n} d(1+r)^j = d\frac{1+r}{r}((1+r)^n - 1)$$

En el caso de $d = 100$, $n = 5$, $D = 600$, el interes anual es la ra'iz de la funci'on

$$g(x) = 600 - 100 * \frac{1+r}{r}((1+r)^5 - 1)$$

Para determinar el interes anual r, graficamos primero la función g.

```
In [7]: def g(r):
            return(600-100*(1+r)*((1+r)**5-1)/r)

        r = linspace(0.0001, 0.1,1000) # vector de 1000 puntos equiespaciados en [0.0001,0.
        y = zeros(len(r)) # declarar la memoria para el vector de las evaluaciones de g
        z = zeros(len(r))

        for i in range(0,1000,1):
            y[i] = g(r[i])

        plot(r,y,r,z)
        savefig("biseccion_plot2.eps")
        show()
```




```
In [8]: miBiseccion(g,0.05,0.07,0.000001,100)
```

```
k = 0 , x_k = 0.060000000000000005
k = 1 , x_k = 0.065
```

```
k = 2 , x_k = 0.0625
k = 3 , x_k = 0.06125
k = 4 , x_k = 0.061875
k = 5 , x_k = 0.0615625
k = 6 , x_k = 0.061406249999999996
k = 7 , x_k = 0.061328125
k = 8 , x_k = 0.061367187499999996
k = 9 , x_k = 0.06138671875
k = 10 , x_k = 0.061396484375
k = 11 , x_k = 0.0614013671875
k = 12 , x_k = 0.06140380859375
k = 13 , x_k = 0.061402587890625
k = 14 , x_k = 0.0614019775390625


Out[8]: 0.0614019775390625
```

Ejercicio: En un circuito eléctrico, la carga $q(t)$ de un capacitor varia según la formula

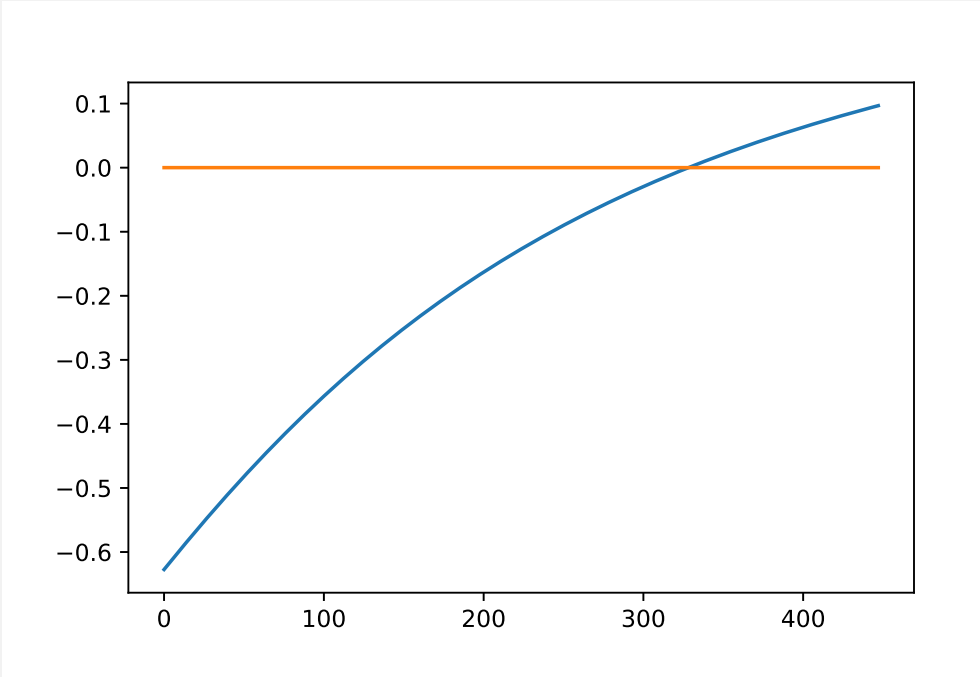
$$q(t) = q_0 e^{-Rt/(2L)} \cos\left(\sqrt{\frac{1}{LC} - (\frac{R}{2L})^2 t}\right),$$

donde q_0 es la carga inicial, L es la inductandica y R la resistancia.
Un objetivo típico en ingeniería eléctrica consiste en determinar R tal que la carga se disipará a un porcentaje fijo en un tiempo dado. Por ejemplo, sea $L = 4$ y $C = 10^{-4}$. Estamos buscando R tal que en $t = 0.05$ se tiene $q = 0.01q_0$. En otras palabras, estamos buscando una raíz R de la función

$$h(R) = e^{-0.005R} \cos\left(\sqrt{2000 - 0.01R^2}0.05\right) - 0.01.$$

```
In [9]: from math import exp, cos, sqrt

        def h(R):
            return( exp(-0.005*R)*cos( sqrt(2000-0.01*R*R)*0.05 ) - 0.01 )

        R = linspace(0, 447,1000) # vector de 1000 puntos equiespaciados en el intervalo de
        y = zeros(len(R)) # declarar la memoria para el vector de las evaluaciones de g
        z = zeros(len(R))

        for i in range(0,1000,1):
            y[i] = h(R[i])
```

```
plot(R,y,R,z)
savefig("biseccion_plot3.eps")
show()
```




```
In [10]: miBiseccion(h,300,400,0.000001,100)

k = 0 , x_k = 350.0
k = 1 , x_k = 325.0
k = 2 , x_k = 337.5
k = 3 , x_k = 331.25
k = 4 , x_k = 328.125
k = 5 , x_k = 329.6875
k = 6 , x_k = 328.90625
k = 7 , x_k = 328.515625
k = 8 , x_k = 328.3203125
k = 9 , x_k = 328.22265625
k = 10 , x_k = 328.173828125
k = 11 , x_k = 328.1494140625
k = 12 , x_k = 328.16162109375
k = 13 , x_k = 328.155517578125
k = 14 , x_k = 328.1524658203125
```

```
k = 15 , x_k = 328.15093994140625
k = 16 , x_k = 328.1517028808594
k = 17 , x_k = 328.1513214111328
k = 18 , x_k = 328.1515121459961
k = 19 , x_k = 328.15141677856445
k = 20 , x_k = 328.1514644622803
k = 21 , x_k = 328.15144062042236
k = 22 , x_k = 328.1514286994934
k = 23 , x_k = 328.1514346599579
k = 24 , x_k = 328.15143167972565
k = 25 , x_k = 328.1514301896095
k = 26 , x_k = 328.15142944455147


Out[10]: 328.15142944455147
```

2.1.2 El método de punto fijo

El problema (2.3) de buscar una **raíz** de una función $f : \mathbb{R} \to \mathbb{R}$

$$\text{hallar } x^* \text{ tal que } f(x^*) = 0.$$

está relacionado con el problema de encontrar un **punto fijo** x^* de una función $g : \mathbb{R} \to \mathbb{R}$,

$$\text{hallar } x^* \text{ tal que } g(x^*) = x^*. \tag{2.6}$$

Hay diversas maneras de transformar el problema (2.3) a un problema de la forma (2.6), o viceversa. Por ejemplo, si $h : \mathbb{R} \to \mathbb{R}$ es una función y $h(x^*) \neq 0$, entonces es fácil verificar que x^* es solución de (2.3) si y solo si es un punto fijo de la función g definida por

$$g(x) = x - h(x)f(x). \tag{2.7}$$

Ejercicio 4. *Sea $f(x) = x^2 + e^x - 2$. El fácil ver que existe una raíz x^* positiva. Muestra que la raíz x^* es un punto fijo de las tres funciones*

$$g_1(x) = x - (x^2 + e^x - 2), \quad g_2(x) = \sqrt{2 - e^x}, \quad g_3(x) = \log(2 - x^2).$$

(Observamos que g_1 es justamente de la forma (2.7) con $h(x) = 1$.) □

Muchas veces conviene reformular el problema de buscar una raíz en un problema de buscar un punto fijo, porque para problemas de punto fijo existe un proceso evidente de aproximación: Elegimos un punto inicial x_0 y calculamos la sucesión $(x_k)_{k=0}^{\infty}$ por la **iteración de punto fijo**

$$x_{k+1} = g(x_k). \tag{2.8}$$

Bajo ciertas condiciones se puede garantizar la existencia de un punto fijo x^* de g y la convergencia de la sucesión $(x_k)_{k=0}^{\infty}$ a el.

Teorema 5 (Teorema punto fijo de Banach en $\mathbb{R}$)**.** *Sea $[a, b] \subset \mathbb{R}$ un intervalo cerrado y $g : [a, b] \to \mathbb{R}$ diferenciable. Si*

(a) $g(x) \in [a, b]$ para todo $x \in [a, b]$,

(b) existe un $L < 1$ tal que $|g'(x)| < L$ para todo $x \in [a, b]$,

entonces:

(i) Existe único punto fijo x^* de g in $[a, b]$.

(ii) Independiente del punto inicial $x_0 \in [a, b]$, la sucesión $(x_k)_{k=0}^{\infty}$ calculada por la iteración (2.8) converge al punto fijo, $\lim_{k\to\infty} x_k = x^*$.

(iii) Se tiene las dos cotas de error

$$|x_k - x^*| \leq \frac{L^k}{1 - L}|x_1 - x_0| \quad y \quad |x_k - x^*| \leq \frac{L}{1 - L}|x_k - x_{k-1}|.$$

Comentario 6. Mencionamos par de comentarios con respecto al último resultado.

- En la condición *(b)* del último teorema **no es suficiente** tener $|g'(x)| < 1$ para todo $x \in [a, b]$.
- La primera cota en *(iii)* se llama cota *a-priori*. Sin conocer x^*, es posible fijar una tolerancia tol y calcular un número de pasos k tal que $|x_k - x^*| \leq \text{tol}$.
- La segunda cota en *(iii)* se llama cota *a-posteriori*. Sin conocer x^*, es posible fijar una tolerancia tol y verificar si $|x_k - x^*| \leq \text{tol}$. En general, cotas a-posteriori son mas exactos que cotas a-priori, pues, usan mas información. Efectivamente, por el teorema del valor medio,

  $$|x_k - x_{k-1}| = |g(x_{k-1}) - g(x_{k-2})| \leq L|x_{k-1} - x_{k-2}| \leq \cdots \leq L^{k-1}|x_1 - x_0|,$$

  es decir,

  $$\frac{L}{1-L}|x_k - x_{k-1}| \leq \frac{L^k}{1-L}|x_1 - x_0|.$$

Ejercicio 7. *Muestre que la funcion* $g(x) = \frac{\sin(x)}{2}$ *cumple con las condiciones del teorema de punto fijo de Banach en el intervalo* $[-\pi, \pi]$. □

Como en el caso de la bisección, necesitamos un *criterio de cancelación*. En este caso, vamos a usar la cota a-posteriori, pues, como mencionamos, es mas exácta. Anotamos el error de la k-ésima iteración $e_k = |x_k - x^*|$, y queremos terminar los calculos si $e_k \leq \text{tol}$. Por la cota a posteriori, eso es cierto si

$$\frac{L}{1-L}|x_k - x_{k-1}| \leq \text{tol},$$

y eso será un primer criterio de cancelación. En la práctica muchas veces no tenemos acceso al número L, y por lo tanto usamos el criterio

$$|x_k - x_{k-1}| \leq \text{tol}.$$

Concluimos que este criterio subestima el error por el factor $L/(1-L)$.

Eso nos lleva al siguiente algoritmo.

```
In [14]: def miIteracion(g,x0,tol,N):
             j=0
             xj = x0
             xjj = g(xj)
             j+=1

             while ( (j < N) & (abs(xj-xjj)>tol) ):
               xj = xjj
               xjj = g(xj)
               j+=1

             return xjj
```

Aplicamos nuestro algoritmo a la función $g(x) = \alpha x$ con diferentes valores de $0 < \alpha < 1$. Notamos que en este caso, g tiene el único punto fijo $x^* = 0$ y cumple con la condición del Teorema de punto fijo de Banach. Como punto inicial usaremos $x_0 = 1$. Dado que $L = \alpha$, veamos que para α cerca a 1 el factor $L/(1 - L)$ será muy grande y nuestro criterio de cancelación sobreestima el error por mucho.

```
In [16]: g = lambda x: 0.5*x
         miIteracion(g,1,0.00000001,1000)

j =  27


Out[16]: 7.450580596923828e-09

In [17]: g = lambda x: 0.9*x
         miIteracion(g,1,0.00000001,1000)

j =  154


Out[17]: 8.98144994103402e-08

In [18]: g = lambda x: 0.999*x
         miIteracion(g,1,0.00000001,1000)

j =  1000


Out[18]: 0.3676954247709635
```

Con respecto al error y al costo del algoritmo podemos decir lo siguiente.

1. Si queremos un error pequeño, deberiamos hacer una gran cantidad de iteraciones, mas aún si $L \approx 1$.

2. En cada iteración hay que evaluar la función g. Dado que en la practica la evaluación de la función g puede ser muy costosa, deberiamos hacer la menor cantidad de iteraciones posibles.

Obviamente, los dos puntos anteriores son requisitos opuestos. Por eso queremos que la convergencia $\lim_{k\to\infty} x_k = x^*$ sea rapida. Dado que necesitamos x_k para calcular x_{k+1}, vamos a definir la velocidad de convergencia de la siguiente manera.

Definición 8. Sea $(x_k)_{k=0}^{\infty}$ una sucesión convergente a $x^* \in \mathbb{R}$ y $e_k := |x_k - x^*|$, es decir, $\lim_{k\to\infty} e_k = 0$. La sucesión $(x_k)_{k=0}^{\infty}$ se llama

(i) **linealmente convergente** o **convergente de orden** 1, si existe una constante $0 < C < 1$ tal que

$$e_{k+1} \leq Ce_k \quad \text{para todo } k.$$

(ii) **convergente de orden** $p > 1$, si existe una constante $0 < C$ tal que

$$e_{k+1} \leq Ce_k^p \quad \text{para todo } k.$$

□

Observamos que para **convergencia de orden 1** se necesita $C < 1$. En otras palabras, en una sucesión convergente de orden 1, el número de dígitos zeros en el error e_k (o bien el número de dígitos exactos en x_k) se multiplica en cada paso por un factor fijo. Para un método de orden $p = 2$, el número de dígitos zeros en el error se duplica en cada paso, mientras para un método de orden $p = 3$ se triplica en cada paso, etc. Concluimos que un orden alto de convergencia es algo deseable. Para ilustrar la diferencia en orden de convergencia y la importancia de un alto orden el la práctica, mostramos en la Tabla 2.1 el error e_k para tres sucesiónes con orden de convergencia 1, 2, y 3.

El orden de convergencia de una sucesión puede ser aproximado numéricamente. Si suponemos que

$$|x_{k+1} - x_k| \approx C|x_k - x_{k-1}|^q \quad \text{y} \quad |x_k - x_{k-1}| \approx C|x_{k-1} - x_{k-2}|^q,$$

entonces

$$q = \frac{\log \frac{|x_{k+1}-x_k|}{|x_k-x_{k-1}|}}{\log \frac{|x_k-x_{k-1}|}{|x_{k-1}-x_{k-2}|}}.$$

k	$p=1$	$p=2$	$p=3$
1	1	1	1
2	0.5	0.5	0.5
3	0.25	0.01	0.002
4	0.1	$3 \cdot 10^{-4}$	$5 \cdot 10^{-9}$
4	0.04	$2 \cdot 10^{-8}$	$6 \cdot 10^{-27}$
5	0.01	$1 \cdot 10^{-16}$	$5 \cdot 10^{-81}$

El error $|x_k - x^*|$ de tres sucesiónes convergentes $(x_k)_{k=0}^{\infty}$ con diferentes ordenes de convergencia.

```
In [165]: def miConvOrder(x):
              from math import log

              n = x.shape[0]

              for k in range(0,n-3):
                  q = log(abs(x[k+3]-x[k+2])/abs(x[k+2]-x[k+1])) / \
                  log(abs(x[k+2]-x[k+1])/abs(x[k+1]-x[k]))
                  print(q)
```

Definimos tres sucesiones $x_{k+1} = 0.5x_k$, $y_{k+1} = y_k^2$ y $z_{k+1} = z_k^3$, con valores iniciales $x[0] = y[0] = z[0] = 0.5$. Obviamente, las tres sucesiones convergen a 0 con orden 1, 2, y 3.

```
In [104]: import numpy as np

          x1 = np.zeros(10)
          x1[0] = 1/2

          x2 = np.zeros(10)
          x2[0] = 1/2

          x3 = np.zeros(10)
          x3[0] = 1/2

          for j in range(0,9):
              x1[j+1] = 0.5*x1[j] # orden 1
              x2[j+1] = x2[j]**2 # orden 2
              x3[j+1] = x3[j]**3 # orden 3

          #print(x1)
```

```
            #print(x2)
            #print(x3)

In [102]: print('orden de convergencia de x1:')
            miConvOrder(x1)
            print('\norden de convergencia de x2:')
            miConvOrder(x2)
            print('\norden de convergencia de x3:')
            miConvOrder(x3)

orden de convergencia de x1:
1.0
1.0
1.0
1.0
1.0
1.0
1.0

orden de convergencia de x2:
4.043181418614952
2.3315670483661655
2.0432714441901694
2.001404372967655
2.0000027516842116
2.000000000020994
2.0

orden de convergencia de x3:
3.717951237169302
3.011399525171021
3.0000009172428346
3.0
3.0



        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)
```

```
    <ipython-input-102-9e7f1da9484a> in <module>()
      4 miConvOrder(x2)
      5 print('\norden de convergencia de x3:')
----> 6 miConvOrder(x3)


    <ipython-input-92-adc95988b6a7> in miConvOrder(x)
      5
      6     for k in range(0,n-3):
----> 7         q = log(abs(x[k+3]-x[k+2])/abs(x[k+2]-x[k+1])) / log(abs(x[k+2]-x[k+1])
      8         print(q)


    ValueError: math domain error
```

Con respecto al orden de convergencia de la iteración de punto fijo podemos formular el siguiente resultado.

Teorema 9. *Sea x^* punto fijo de $g : \mathbb{R} \to \mathbb{R}$, sea g' continua, y $|g'(x^*)| < 1$. Entonces*

*(i) la sucesión definida por (2.8) es **localmente convergente**, es decir, para x_0 suficientemente cerca de x^* se tiene $\lim_{k\to\infty} e_k = 0$, y la convergencia es **por lo menos lineal**.*

(ii) Particularmente, existe un intervalo cerrado $[a, b]$ con $x^* \in [a, b]$ y un número $L > 1$ tal que g cumple con las condiciones del teorema punto fijo de Banach.

Idea de la demostración. Según el teorema de Taylor, podemos escribir

$$g(x) = g(x^*) + g'(x)(x - x^*) + C(x - x^*)^2 = x^* + g'(x)(x - x^*) + C(x - x^*)^2,$$

donde usamos que x^* es punto fijo de g. Concluimos que

$$|x_{j+1} - x^*| = |g(x_j) - x^*| \leq \left(|g'(x^*)| + C|x_j - x^*|\right) |x_j - x^*|.$$

□

Observamos que convergencia de orden p implica convergencia de orden $p - 1$. El Teorema 9 establece que la convergencia de la iteración de punto fijo es *por lo menos lineal*. Puede ser mas alta, como muestra el siguiente resultado.

Teorema 10. *Sea x^* punto fijo de $g : [a, b] \to \mathbb{R}$, sea g' continua, y $|g'(x^*)| < 1$.*

(i) Si $g'(x^*) = 0$, entonces la convergencia de la iteración de punto fijo es de orden 2.

(ii) Si $g'(x^*) \neq 0$, entonces la convergencia de la iteración de punto fijo es lineal pero ***no es*** *de orden 2.*

Idea de la demostración. Según el teorema de Taylor, podemos escribir

$$g(x) = g(x^*) + g'(x)(x - x^*) + C(x - x^*)^2 = x^* + g'(x)(x - x^*) + C(x - x^*)^2 = x^* + C(x - x^*)^2,$$

donde usamos que x^* es punto fijo de g y adicionalmente la condición $g'(x^*) = 0$. Concluimos que

$$|x_{j+1} - x^*| = |g(x_j) - x^*| \leq C|x_j - x^*|^2.$$

□

Ejercicio 11. *Para calcular $\sqrt{5}$, buscamos la raíz positiva de la función*

$$f(x) = x^2 - 5.$$

Vamos a usar iteraciones de punto fijo, usando las cuatro funciones

$$g_1(x) = 5 + x - x^2, \quad g_2(x) = \frac{5}{x}, \quad g_3(x) = 1 + x - \frac{1}{5}x^2, \quad g_4(x) = \frac{1}{2}\left(x + \frac{5}{x}\right).$$

Verifica que $\sqrt{5}$ es un punto fijo de las cuatro funciones. ¿Que podemos decir sobra la convergencia de las cuatro iteraciones? □

Ejercicio: Verificamos el comportamiento de la iteración de punto fijo del último ejercicio. Primero, modificamos nuestra función miIteracion para que nos devuelve toda la sucesión.

```
In [134]: import numpy as np

          def miIteracion_v1(g,x0,tol,N):
              x = np.zeros(1)
              j=0
              x[j] = x0
              x = np.append(x, g(x[j]) )
              j+=1

              while ( (j < N) & (abs(x[j]-x[j-1])>tol) ):
                  x = np.append(x, g(x[j]) )
                  j+=1

              return x
```

Ahora podemos calcular las iteraciones y llamar nuestra función para determinar el orden numérico de convergencia

```
In [162]: g1 = lambda x: 5+x-x**2
          x1 = miIteracion_v1(g1,2,0.00000001,100)

          g2 = lambda x: 5/x
          x2 = miIteracion_v1(g2,2,0.00000001,100)

          g3 = lambda x: 1+x-x**2/5
          x3 = miIteracion_v1(g3,2,0.00000001,100)

          g4 = lambda x: (x+5/x)/2
          x4 = miIteracion_v1(g4,2,0.00000001,100)

          print(x1)
          print(x2)
          print(x3)
          print(x4)

[ 2.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3.
 -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3.
 -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3.
 -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3.
 -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.  3.
 -1.  3. -1.  3. -1.  3. -1.  3. -1.  3. -1.]
[2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5
 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5
 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5
 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5
 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2.5
 2.  2.5 2.  2.5 2.  2.5 2.  2.5 2.  2. ]
[2.         2.2        2.232      2.2356352  2.23602225 2.23606315
 2.23606747 2.23606792 2.23606797 2.23606798]
[2.         2.25       2.23611111 2.23606798 2.23606798]
```

Notamos que solamente las iteraciones de g_3 y g_4 parecen ser convergentes. Determinamos el orden numérico de convergencia:

```
In [163]: miConvOrder(x3)

1.1868895411865248
1.0297886492308033
```

```
1.0033859763554223
1.0003605481028033
1.000038099139113
1.0000040191892476
1.0000004282570802


In [164]: miConvOrder(x4)

1.997857725311876
1.9999968328569404
```

Es decir, numéricamente observamos lo que dice la teoria.

2.1.3 El método de Newton

El método de Newton es el algoritmo mas importante para resolver problemas no lineales. Nuestro objetivo es (2.3), encontrar una raiz x^* de una función $f : \mathbb{R} \to \mathbb{R}$. Reformulamos el problema como en (2.7) y buscamos el punto fijo x^* de la función g dada por

$$g(x) = x - h(x)f(x),$$

donde h todavia es una función arbitraria. El objetivo ahora es elegir la función h tal que la iteración de punto fijo converge con orden 2. Según el Teorema 10, una condición suficiente para que la iteración de punto fijo converja con orden 2 es

$$g'(x^*) = 0.$$

La ecuación determinará la función h. La regla del producto para calcular derivadas muestra $g'(x) = 1 - h'(x)f(x) - h(x)f'(x)$, y obtenemos

$$g'(x^*) = 0 \Longleftrightarrow h(x^*) = \frac{1}{f'(x^*)}.$$

Es decir, la iteración de punto fijo para la función g dada por

$$g(x) = x - \frac{f(x)}{f'(x)}$$

nos da la posibilidad de obtener convergencia de orden 2, y se llama *el método de Newton*. El método de Newton se lee entonces

$$\begin{gathered} \text{elegir punto inicial } x_0 \in \mathbb{R}, \\ x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \text{ para } k \geq 0. \end{gathered} \tag{2.9}$$

La iteración (2.9) se puede justificar de una manera mas constructiva. Si x_k es la iteración actual, entonces la recta tangente a la función f en x_k es

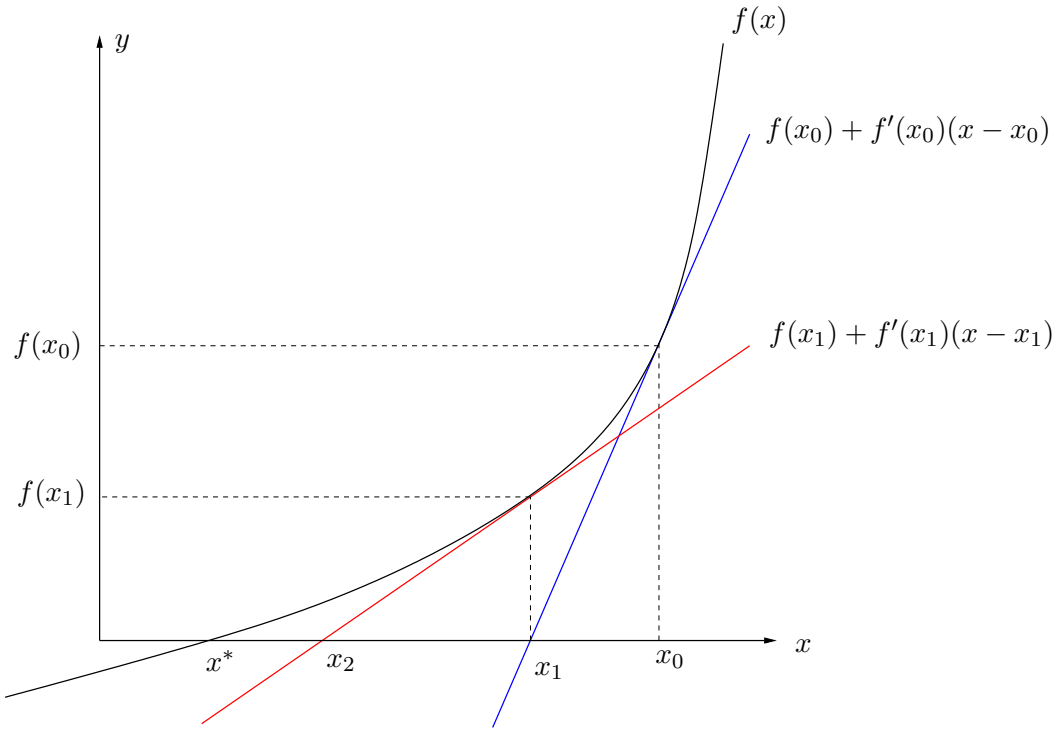
$$y(x) = f(x_k) + f'(x_k)(x - x_k). \tag{2.10}$$

La proxima iteración x_{k+1}, dada por la formula recursiva en (2.9), es nada mas que la raiz de la recta tangente. Eso tiene sentido, dado que la recta tangente es una aproximación a la función f entorno del punto x_k.

Ejercicio 12 (¿Como calcula una calculadora $\sqrt{a}$?)**.** *El objetivo es calcular $\sqrt{a}$ para un número positivo $a > 0$. Obviamente, $\sqrt{a}$ es una raíz de la función*

$$f(x) = x^2 - a.$$

Formule la iteración de Newton. □



El método de Newton en una dimension.

Para analizar el orden de convergencia del método de Newton, usaremos el Teorema 10. Notamos que

$$g'(x) = 1 - \frac{f'(x)^2 - f(x)f''(x)}{f'(x)^2} = \frac{f(x)f''(x)}{f'(x)^2}.$$

Si adicionalmente $f'(x^*) \neq 0$, entonces $g'(x^*) = 0$, y según Teorema 10 la iteración de punto fijo converge cuadraticamente. Si $f'(x^*) = 0$ (es decir, x^* es por lo menos una raíz doble), entonces $g'(s) \neq 0$ por a regla de l'Hopital, y según Teorema 10 la iteración de punto fijo converge linealmente pero no cuadraticamente. En resumen, obtenemos el siguiente resultado.

Teorema 13. *Sea $f : \mathbb{R} \to \mathbb{R}$ una función, x^* una raíz de f, y x_0 suficientemente cerca de x^*.*

(i) Si x^* es raíz simple de f, es decir $f'(x^*) \neq 0$, entonces el método de Newton converge cuadraticamente.

(ii) Si x^* es una raíz multiple de f, es decir $f(x) = (x - x^*)^m r(x)$ con $m \geq 2$ y $r(x^*) \neq 0$, entonces el método de Newton converge linealmente pero no cuadraticamente.

□

Ejercicio 14. *Determine si el método de Newton converge linealmente o cuadraticamente para las siguientes funciones y raizes.*

(a) $f(x) = x^2$, $x^* = 0$,

(b) $f(x) = x^3 - 3x + 2$, $x^* = 1$, $x^* = -2$

□

2.2 Sistemas no lineales, el método de Newton

El método de Newton también se aplica a sistemas no lineales (2.2) donde $f : \mathbb{R}^n \to \mathbb{R}^n$. En vez de una recta tangente (2.10), podemos calcular el plano tangente de f en un punto $x_k \in \mathbb{R}^n$

$$y(x) = f(x_k) + D_f(x_k)(x - x_k),$$

donde D_f es la matriz Jacobiana de f,

$$D_f(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \dots & \frac{\partial f_1}{\partial x_n}(x) \\ \frac{\partial f_2}{\partial x_1}(x) & \frac{\partial f_2}{\partial x_2}(x) & \dots & \frac{\partial f_2}{\partial x_n}(x) \\ & & \vdots & \\ \frac{\partial f_n}{\partial x_1}(x) & \frac{\partial f_n}{\partial x_2}(x) & \dots & \frac{\partial f_n}{\partial x_n}(x) \end{pmatrix}.$$

La iteración de punto fijo (2.9) se transforma en

$$\begin{aligned} &\text{elegir punto inicial } x_0 \in \mathbb{R}^n, \\ &x_{k+1} = x_k - D_f(x_k)^{-1} f(x_k), \end{aligned} \tag{2.11}$$

donde $D_f(x_k)^{-1}$ es la inversa de la matriz Jacobiana evaluada en x_k. Es decir, en cada iteración tenemos que resolver un sistema lineal.

Ejercicio 15. *Considere el sistema no lineal*

$$\begin{aligned} x^2 - y - 1 &= 0 \\ (x-2)^2 + (y - 1/2)^2 - 1 &= 0. \end{aligned}$$

(a) Haga un boceto y concluye que el sistema tiene exactamente dos soluciones.

(b) Usando el punto inicial $(x_0, y_0) = (1.5, 1.5)$*, haga dos pasos del método de Newton.*

□

Procedemos a la implementación del método de Newton en el caso $f : \mathbb{R}^n \to \mathbb{R}^n$ ($n = 1$ para ecuaciones no lineales será un caso especial entonces). Para obtener un criterio de cancelación, notamos lo siguiente. El terminó $\|x^* - x_k\|_2$ no es accesible, pero suponiendo que x_{k+1} es una mejor aproximación a x^* que x_k, el primer criterio de cancelación será terminar el algoritmo despues del paso k si

$$\|x_{k+1} - x_k\|_2 \leq \text{tol}.$$

Una desventaja de este criterio es que tenemos que determinar x_{k+1}, pues, la parte costosa del método de Newton es la evaluación de f y D_f. Sin embargo, notamos que

$$\|x_{k+1} - x_k\|_2 = \|D_f(x_k)^{-1} f(x_k)\|_2 \approx \|D_f(x_{k-1})^{-1} f(x_k)\|_2$$

por la continuidad de D_f. El termino $D_f(x_{k-1})^{-1}$ ya lo tenemos (en una forma que vamos a especificar mas adelante) del paso anterior, asi que solamente hay que determinar $f(x_k)$. Recordamos que en el caso $n = 1$, en vez de $D_f(x_k)^{-1}$ se tiene $f'(x_k)^{-1}$, y en vez de la norma euclidiana $\|\cdot\|_2$ se tiene el valor absoluto $|\cdot|$. En una dimensión podemos implementar el método de Newton de la siguiente manera.

```
In [46]: def miNewton1d(f,df,x0,tol,N):
             import numpy as np

             x = np.zeros(1)
             j=0
             x[j] = x0
             x = np.append( x, x[j] - f(x[j]) / df(x[j]) )
             j+=1

             while ( ( abs( f(x[j]) / df(x[j-1])  ) > tol ) & (j < N) ):
                 x = np.append( x, x[j] - f(x[j]) / df(x[j]) )
                 j+=1

             return x
```

Aproximamos $\sqrt{2}$ como raíz de $f(x) = x^2 - 2$. Notamos que $f'(x) = 2x$, y dado que $f'(\sqrt{2}) \neq 0$, la raíz es simple y Newton convergerá con orden 2:

```
In [47]: f = lambda x: x**2-2
         df = lambda x: 2*x
         x = miNewton1d(f,df,1,1e-8,20)
         print(x)
         miConvOrder(x)

[1.         1.5        1.41666667 1.41421569 1.41421356]
```

```
1.9680992818391112
1.9995089548529266
```

Sin embargo, $g(x) = x^2$ tiene una raíz doble en 0, pues, $g'(x) = 2 * x$. Newton convergerá con orden 1:

```
In [49]: g = lambda x: x**2
         dg = lambda x: 2*x
         x = miNewton1d(g,dg,1,1e-8,10)
         print(x)
         miConvOrder(x)

[1.000000e+00 5.000000e-01 2.500000e-01 1.250000e-01 6.250000e-02
 3.125000e-02 1.562500e-02 7.812500e-03 3.906250e-03 1.953125e-03
 9.765625e-04]
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
```

Para implementar el método de Newton en $\mathbb{R}^n$, tenemos que resolver sistemas lineales y usar normas vectoriales en vez del valor absoluto. Para resolver un sistema lineal en python, por ejemplo

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 17 \\ 39 \end{pmatrix},$$

podemos usar el objeto "array" de numpy:

```
In [208]: import numpy as np

          b = np.array( [17, 39] )
          A = np.array( [ [1,2], [3,4] ] )

          print(b)
          print(A)
```

```
          x = np.linalg.solve(A,b)
          normx = np.linalg.norm(x)
          print(x)
          print(normx)

[17 39]
[[1 2]
 [3 4]]
[5. 6.]
7.810249675906654
```

Entonces, el método de Newton se puede implementar de la siguiente manera:

```
In [200]: def miNewton(f,df,x0,tol,N):
              import numpy as np

              n = x0.shape[0]
              x = np.zeros([1,n])
              j=0
              x[j] = x0

              # calcular próximo paso
              y = x[j] - np.linalg.solve(df(x[j]),f(x[j]))

              # agregarlo al array x
              x = np.append( x, [ y ], axis=0 )
              j+=1

              while ( (j<N) & \
                     ( np.linalg.norm( np.linalg.solve( df(x[j-1]), f(x[j]) ) ) > tol ) ):
                  y = x[j] - np.linalg.solve(df(x[j]),f(x[j]))
                  x = np.append( x, [ y ], axis=0 )
                  j +=1

              return x
```

Vamos a aproximar numéricamente la solución el problema del ejercicio XXX, es decir,

$$f(x,y) = \begin{pmatrix} x^2 - y - 1 \\ (x-2)^2 + (y-2)^2 - 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

En este caso,

$$D_f(x,y) = \begin{pmatrix} 2x & -y \\ 2(x-2) & 2(y-2) \end{pmatrix}.$$

```
In [209]: import numpy as np

          def f(x):
              return np.array([x[0]**2-x[1]-1, (x[0]-2)**2+(x[1]-2)**2-1])

          def df(x):
              return np.array([[2*x[0], -x[1]],[2*(x[0]-2), 2*(x[1]-2)]])

          x=miNewton(f,df,np.array([1,1]),1e-6,10)
          print(x)

[[1.         1.        ]
 [1.5        1.        ]
 [1.46428571 1.14285714]
 [1.46790383 1.15324463]
 [1.46751737 1.15355892]
 [1.46750422 1.15356734]]
```

2.3 Como elegir el punto inicial

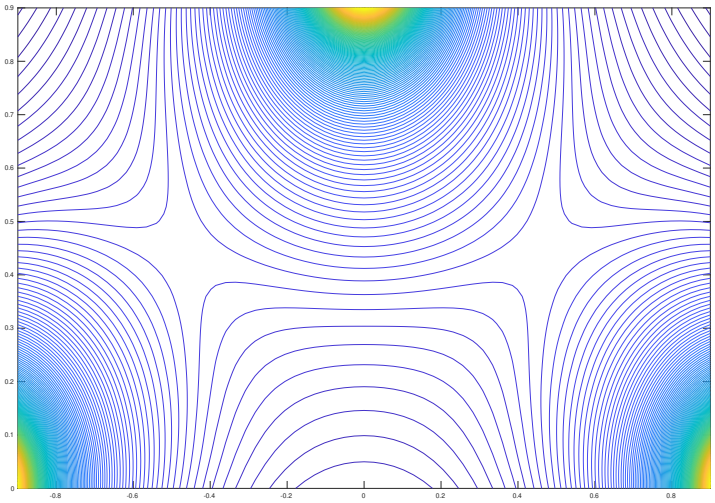
La iteración de punto fijo requiere un punto inicial x_0, y el Teorema 9 garantiza que la iteración converge al punto fijo x^* si x_0 es suficientemente cerca de x^*. La pregunta entonces es, ¿como elegir el punto inicial x_0?

(1) Si no tenemos ninguna idea, entonces la única opción es **adivinar**.

(2) En general resolvemos problemas que tienen algún sentido físico. En el Ejemplo 1 estamos buscando un punto (x,y), tal que las fuerzas gravitacionales inducidas por las tres masa m_1, m_2, m_3 estén equilibradas. Las coordenadas de las tres masas son $(x_1,0),(x_2,0),(0,y_3)$, es decir, constituyen un triangulo. Afuera del triangulo, las fuerzas nunca estarán equilibradas, y por lo tanto vamos a elegir un punto inicial adentro del triangulo. Mas aún, la función $\mathbf{F}_1+\mathbf{F}_2+\mathbf{F}_3$ es el gradiente de un *potencial*

$$U(x,y) = Gm\sum_{k=1}^{3}\frac{m_k}{\sqrt{(x-x_k)^2+(y-y_k)^2}},$$

y eso significa que estamos buscando un *extremo* o un *punto de silla* de U. En la Figura 2.3 mostramos el potencial U. Notamos que tiene dos puntos de silla aproximadamente en $(-0.45, 0.45)$ y $(0.45, 0.45)$.



El potencial U para $m_1 = m_2 = m_3 = 1$ con coordenadas $(-1, 0), (1, 0), (0, 1)$.

(3) A veces se combina dos métodos. Por ejemplo, en el caso de una ecuación escalar $n = 1$ tenemos el método de bisección, que es muy lento pero muy robusto en el sentido de que su convergencia no depende mucho de la elección de los datos a_0, b_0. Por el otro lado tenemos el método de Newton, que converge mas rapido pero solo si el punto inicial x_0 es cerca de la raiz. Podemos combinar ambos métodos y hacer un par de pasos con bisección para generar un punto inicial para el método de Newton.

2.4 Ejercicios

1. Un objeto en caída libre alcanza la velocidad final v, dada por

$$v^2 = \frac{2mg}{\rho A C_d},$$

donde m es la masa del objeto, g es la constante gravitacional, ρ es la densidad del liquido en el cual se está cayendo el objeto, A es el area dado por la "sombra" del objeto, y C_d es el coeficiente de arrastre. El coeficiente de arrastre depende de la velocidad (eso se nota al extender el brazo por la ventana de un auto), es decir, $C_d = C_d(v)$. Formule el problema de determinar la velocidad final en forma de determinar la raíz de una función.

2. Consideramos el polinomio $p(x) = -\frac{x^2}{2} + \frac{5x}{2} + \frac{9}{2}$.

   (a) Determine las raíces reales de p, haciendo un boceto de la gráfica.

   (b) Determine las raíces reales de p, usando la fórmula usual.

   (c) Aproxime la raíz mas grande, usando tres pasos del método de bisección con intervalo inicial $[a_0, b_0] = [5, 10]$.

   (d) Aproxime la raíz mas pequeña, usando tres pasos del método de bisección. Elige el intervalo inicial adecuadamente según sus resultados del punto (a).

3. El objetivo es maximizar la función $f(x) = xe^{-x^2}$ para $x > 0$.

   (a) Determine analíticamente el punto $x^\star > 0$ donde f alcanza su máximo.

   (b) Use bisección para la derivada de f para aproximar $x^\star$. ¿Con intervalo inicial $[a_0, b_0] = [0, 2]$, cuantos pasos tenemos que hacer para garantizar un error menor que 10^{-8}? Aplica el código `miBiseccion` en `Matlab` o `Octave` para verificar su resultado.

4. En la notación del Teorema 3, $e_k := |x^* - x_k|$. ¿Es correcto o falso que

$$e_{k+1} \leq e_k \text{ para todo } k \in \mathbb{N}?$$

5. Muestre que las siguientes funciones cumplen con las condiciones del teorema de punto fijo de Banach en los intervalos respectivos.

   (a) $g(x) = \frac{1}{2}e^{x/2}$ en $[0, 1]$.

   (b) $g(x) = \frac{e^x(x-1)}{4(x+2)}$ en $[-1, 0]$.

6. Verifique que la sucesiones convergen con el orden dado.

   (a) $x_k := 2^{-k}$ con orden 1

   (b) $x_k := 2^{-2^k}$ con orden 2

   (c) $x_k := 3^{-p^k}$ con orden p

7. Considere la sucesión dada por

$$x_0 = \sqrt{2}, \quad x_1 = \sqrt{2+\sqrt{2}}, \quad x_2 = \sqrt{2+\sqrt{2+\sqrt{2}}}, \quad x_3 = \sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2}}}}, \quad \ldots$$

   (a) Determine la función g tal que la sucesión $(x_k)_{k=0}^{\infty}$ corresponde a la iteración de punto fijo de g.

   (b) Determine el punto fijo de g.

   (c) Verifique que la iteración de punto fijo para g es localmente convergente (Teorema 9). ¿Cual es el orden de convergencia?

8. Formule la iteración de Newton para la solución del problema de hallar $(x, y) \in \mathbb{R}^2$ tal que

$$\begin{aligned} x^2 + 2xy + 3y^2 &= 5.9 \\ 2x^2 - y^2 &= 1.1. \end{aligned}$$

   Dado el punto inicial $(x_0, y_0) = (1, 1)$, calcule un paso del método de Newton.

9. El objetivo es resolver el sistema no lineal

$$\begin{aligned} x + 2y - 2 &= 0 \\ x^2 + 4y^2 - 4 &= 0 \end{aligned}$$

   (a) Haga un boceto y concluye que el sistema tiene exactamente dos soluciones.

   (b) Usando su boceto, elige un punto inicial y haga dos pasos del método de Newton. ¿Que puede observar?

Chapter 3

Análisis de errores

Para ilustrar lo que viene, consideramos el siguiente objetivo.

Objetivo: Dado tres masas m_1, m_2, m_3 y sus respectivas coordenadas, calcular un punto $x \in \mathbb{R}^2$ donde la fuerza gravitacional esté equilibrada.

Procedimiento: (1) Como modelo, usaremos la ley de la gravitación universal de Newton como en Ejemplo 1. Eso nos lleva al sistema no lineal (2.1). (2) Determinaremos las tres masas y sus posiciones con instrumentos de medición. (3) Elegimos un punto inicial $x_0 \in \mathbb{R}^2$ y formulamos la iteración de Newton (2.11). Calculamos N pasos de la iteración y aceptamos x_N como resultado numérico final.

En general, el resultado numérico $x_N \in \mathbb{R}^2$ lleva un error. Si $x \in \mathbb{R}^2$ es la solución exacta que queremos aproximar, podemos considerar el **error absoluto**

$$\|x - x_N\|_{\mathbb{R}^2}$$

o el **error relativo**

$$\frac{\|x - x_N\|_{\mathbb{R}^2}}{\|x\|_{\mathbb{R}^2}}.$$

Los errores se componen de varias partes.

(i) **Errores inherentes.**

(a) **Errores de modelamiento.** Los modelos matemáticos, aunque bien establecidos, son abstracciones de la realidad, y muchas veces son simplificaciones de ella.
Con respecto a lo de arriba, la ley de Newton es un modelo mátematico de la realidad que queremos calcular.

(b) **Errores de medición.** En el momento de medir un dato (con una regla, un reloj, o cualquier instrumento de medición), se produce un error. Aqui también consideramos errores en el dato por ser el resultado de un cálculo anterior.
Con nuestros instrumentos de medición no somos capaces de medir de manera exacta las tres masas y sus posiciones, y tampoco la constante de gravitación universal.

(ii) **Errores computacionales.**

(a) **Errores de discretización.** Muchos objetos no pueden ser calculados de manera exacta numéricamente porque contienen límites, sumas infinitas, o una infinta cantidad de elementos. Un ejemplo es nuestro algoritmo de la iteración de punto fijo, que termina despues de una finita cantidad de iteraciónes. Por lo tanto, el resultado numérico que entrega el algoritmo lleva un error de discretizacion.
Calculamos una finita cantidad de iteraciones del método de Newton.

(b) **Errores de redondeo.** La memoria de una calculadora o computadora es finita, y eso implica que no puede representar todos los números en $\mathbb{R}$. ¿Como podemos tener el número π en la memoria si es irracional?
La iteración de Newton la calculamos en una maquina, y no se puede representar todos los números.

3.1 Errores inherentes - condicionamiento

Aceptamos por ahora el error de modelamiento. Nuestros modelos son modelos matemáticos, y por el resto del capítulo los representamos como

$$f : \mathbb{R}^n \to \mathbb{R}^m, \tag{3.1}$$

donde f es una función entre el conjunto de datos $\mathbb{R}^n$ y el conjunto de resultados $\mathbb{R}^m$. Para el ejemplo del princípio del capítulo, $n = 9$ (tres objetos con masa y coordenadas en el plano, y $m = 2$ (las coordenadas del punto donde la fuerza esté equilibrada). En lo que sigue, vamos a suponer que somos capaces de resolver de manera exacta nuestro problema matemático (una suposición no muy realista en la practica). Nuestro objetivo es conocer el resultado $y \in \mathbb{R}^m$, donde $y = f(x)$ con un *dato exacto* $x \in \mathbb{R}^n$. Por los errores de medición tenemos en la practica solamente un *dato inexacto* $\widetilde{x} = x + \Delta x \in \mathbb{R}^n$ a nuestra disposición, y asi podemos calcular solamente $\widetilde{y} = y + \Delta y = f(\widetilde{x})$. Por las razones que explicamos arriba, tenemos que aceptar el error en el dato

$$\|x - \widetilde{x}\|_{\mathbb{R}^n} = \|\Delta x\|_{\mathbb{R}^n},$$

pero en general tenemos una cota para el error, por ejemplo para errores de medición[1]. Lo que podemos pedir entonces de nuestro modelo es que el error en el resultado

$$\|f(x) - f(\widetilde{x})\|_{\mathbb{R}^m} = \|y - \widetilde{y}\|_{\mathbb{R}^m} = \|\Delta y\|_{\mathbb{R}^m}$$

[1]El área de *metrología* se trata de la teoría de medición y sus errores. Una medición sin información adicional sobre el error es practicamente inútil.

sea de la misma magnitud que el error en el dato. Es decir, requerimos que un pequeño error en el dato produce un pequeño error en el resultado. Esta propiedad se llama **condición** del problema. Para caracterizar la condición de un problema, escribimos la función en (3.1) como

$$f(x_1,\dots,x_n) = \begin{pmatrix} f_1(x_1,\dots,x_n) \\ f_2(x_1,\dots,x_n) \\ \vdots \\ f_m(x_1,\dots,x_n) \end{pmatrix},$$

y supongamos que todos los f_j son diferenciables. Por el teorema de Taylor podemos escribir

$$f_j(x+\Delta x) = f_j(x) + \sum_{k=1}^{n} \frac{\partial f_j}{\partial x_k}(x)\cdot \Delta x_k + \sum_{\ell,k=1}^{n} \frac{\partial^2 f_j}{\partial x_\ell \partial x_k}(\xi_{\ell,k})\Delta x_\ell \Delta x_k$$

Si el error en el dato $\|\Delta x\|_{\mathbb{R}^n}$ es pequeño, entonces la última suma no contribuye mucho, y concluimos

$$\left|\frac{f_j(x+\Delta x)-f_j(x)}{f_j(x)}\right| \le \sum_{k=1}^{n} \underbrace{\left|\frac{x_k}{f_j(x)}\right| \cdot \left|\frac{\partial f_j}{\partial x_k}(x)\right|}_{C_{k,j}:=} \cdot \left|\frac{\Delta x_k}{x_k}\right|$$

El lado izquierdo de arriba representa el error relativo en la componente j del resultado. El termino $\left|\frac{\Delta x_k}{x_k}\right|$ representa el error en la componente k del dato. El número $C_{k,j}$ mide entonces la amplificación del error relativo de la componente k del dato que se produce en la componente j del resultado. Los factores $C_{k,j}$ se llaman **factores de condicionamiento** del problema. Un problema de la forma (3.1) tiene $m \cdot n$ factores de condicionamiento. En general, un factor de condicionamiento grande indica una mala condición, mientras un factor de condicionamiento pequeño indica una buena condición.

1. *Grande* y *pequeño* son terminos relativos. Como regla general, vamos a decir que un factor de condicionamiento es pequeño si $C_{k,j} \approx 1$.

2. Los factores de condicionamiento dependen del dato exacto x. Un problema perfectamente puede tener una buena condición para ciertos datos y mala condición para otros datos.

3. El factor $C_{k,j}$ indica como un error en el dato x_k influye el resultado f_j. Por ejemplo, si f_j no depende de x_k, entonces obviamente $C_{k,j} = 0$.

Definición 16. Un problema se llama **bien condicionado**, si todos los factores de condicionamiento son pequeños. En el caso contrario, el problema se llama **mal condicionado**.

Vamos a analizar el condicionamiento de problemas simples.

Ejemplo 17. *Sea $f : \mathbb{R}^2 \to \mathbb{R}$ dada por $f(x_1, x_2) = x_1 + x_2$. En este caso, $n = 2$, $m = 1$. Calculamos*

$$\frac{\partial f}{\partial x_1}(x_1, x_2) = 1, \quad \frac{\partial f}{\partial x_2}(x_1, x_2) = 1.$$

Por lo tanto,

$$C_{1,1} = \left| \frac{x_1}{x_1 + x_2} \right|, \quad C_{2,1} = \left| \frac{x_2}{x_1 + x_2} \right|.$$

Concluimos que $C_{k,1}$ es grande si $|x_k|$ es grande mientras $|x_1 + x_2|$ es pequeño, o bien $x_1 \approx -x_2$. En otras palabras, restar dos números (grandes) casi iguales es un problema mal condicionado. □

Ejemplo 18. *Sea $f : \mathbb{R} \to \mathbb{R}$ dada por $f(x) = e^{3x^2}$. En este caso, $n = m = 1$. Calculamos $\frac{\partial f}{\partial x}(x) = 6xe^{3x^2}$. Por lo tanto,*

$$C_{1,1}(x) = \left| \frac{x}{e^{3x^2}} \right| |6xe^{3x^2}| = 6x^2.$$

Concluimos que si $|x|$ es pequeño, entonces el problema está bien condicionado.

Por ejemplo, para $x = 0.1$ y $\widetilde{x} = 0.10001$, el error relativo en el dato es

$$e_{\text{rel}}(x) := \frac{|x - \widetilde{x}|}{|x|} = 10^{-4},$$

y el error relativo en el resultado es

$$e_{\text{rel}}(f(x)) := \frac{|f(x) - f(\widetilde{x})|}{|f(x)|} \approx 6.0003 \cdot 10^{-6}.$$

Por otro lado, para $y = 4$ y $\widetilde{y} = 4.0004$, el error en el dato es

$$e_{\text{rel}}(y) := \frac{|y - \widetilde{y}|}{|y|} = 10^{-4},$$

pero el error relativo en el resultado es

$$e_{\text{rel}}(f(y)) := \frac{|f(y) - f(\widetilde{y})|}{|f(y)|} \approx 9.65 \cdot 10^{-3}.$$

En el primer caso tenemos una reducción del error relativo, y en el segundo caso una amplificación. Se puede observar bien que

$$\frac{e_{\text{rel}}(f(x))}{e_{\text{rel}}(x)} \approx 0.06 = C_{1,1}(x),$$
$$\frac{e_{\text{rel}}(f(y))}{e_{\text{rel}}(y)} \approx 96 = C_{1,1}(y).$$

□

3.2 Errores computacionales - Aritmetica flotante y estabilidad

Empezamos con una suma simple en Python:

```
In [1]: 0.1+0.1+0.1

Out[1]: 0.30000000000000004
```

Al parecer, en Python, la suma $0.1 + 0.1 + 0.1$ no es igual a 0.3:

```
In [4]: 0.1+0.1+0.1==0.3

Out[4]: False
```

Intentamos lo mismo con la suma $0.1 + 0.1$:

```
In [7]: print(0.1+0.1)
        0.1+0.1==0.2

0.2


Out[7]: True
```

Lo que acabamos de ver es una consecuencia de aritmetica en el sistema punto flotante, el sistema mas común para representar números en maquinas.

Para una base $b \in \mathbb{N}$, $b > 1$, podemos representar cada número real $x \neq 0$ como

$$x = \sigma \left(\sum_{k=1}^{\infty} a_k b^{-k} \right) b^e, \tag{3.2}$$

con los **dígitos** $a_j \in \{0, \dots, b-1\}$, $a_1 \neq 0$, el **signo** $\sigma \in \{\pm 1\}$, y el **exponente** $e \in \mathbb{Z}$. Estamos acostumbrados al sistema decimal $b = 10$, es decir

$$x = \pm(0.a_1a_2a_3\dots) \cdot 10^b = \pm\left(a_1 10^{-1} + a_2 10^{-2} + a_3 10^{-3} \dots\right) 10^b.$$

Por ejemplo, en base $b = 10$,

$$\begin{aligned} -10.37 &= -0.1037 \cdot 10^2, \\ 0.00931 &= 0.931 \cdot 10^{-2} \\ \pi &= 0.314159\dots \cdot 10^1. \end{aligned}$$

Primero notamos que las maquinas que usamos hoy en día son digitales, es decir, usan el sistema binario $b = 2$. Los dígitos en este sistema son $0, 1$ y se llaman **bits**. Por ejemplo,

$$1011 \text{ en base } 2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \text{ en base } 10 = 11.$$

Físicamente, un bit se representa mediante de los dos estados de conducción de un transistor. Segundo, cada maquina real como un ábaco, una calculadora, o una computadora, tiene una cantidad finita de memoria y en general no puede guardar todos los dígitos a_k de la representación (3.2). Por lo tanto, la cantidad de números reales diferentes que se puede representar en una maquina es limitada, y es necesario representar estos números en memoria en un formato bien definido y suficientemente flexible. En la practica, se usa el sistema de **punto flotante normalizado**. En la representación (3.2), se permite solo una cantidad finita $t \in \mathbb{N}$ de dígitos y un exponente en un intervalo fijo $e \in [e_{\min}, e_{\max}]$. El conjunto de números que se pueden representar con estas restricciones lo anotamos como

$$\mathbb{F}(b, t, e_{\min}, e_{\max}) = \{0\} \cup \left\{ x \in \mathbb{R} \mid x = \sigma \left(\sum_{k=1}^{t} a_k b^{-k} \right) b^e, e_{\min} \leq e \leq e_{\max} \right\}.$$

Por ejemplo, $12345.678 = 0.12345678 \cdot 10^5$, y por lo tanto

$$\begin{aligned} &12345.678 \in \mathbb{F}(10, 9, -7, 7) \\ &12345.678 \notin \mathbb{F}(10, 7, -7, 7) \text{ (por la cantidad de dígitos)} \\ &12345.678 \notin \mathbb{F}(10, 9, -7, 2) \text{ (por el exponente).} \end{aligned}$$

El número mas grande y mas pequeño en valor absoluto en $\mathbb{F}(b, t, e_{\min}, e_{\max})$ son

$$\begin{aligned} x_{\max} &= b^{e_{\max}} \sum_{k=1}^{t} (b-1) b^{-k} = b^{e_{\max}} (1 - b^{-t}), \\ x_{\min} &= 1 \cdot b^{-1} b^{e_{\min}} = b^{e_{\min}-1}. \end{aligned} \tag{3.3}$$

No se puede representar números mas grandes en valor absoluto que $x_{\max}$ - este caso se llama **overflow** - o menores que $x_{\min}$ - este caso se llama **underflow**.

Los lenguajes de programmación como C, C++, y Python, y la mayoría de los lenguajes matemáticos, como Matlab, Maple, y también Mathematica, utilizan el estándar IEEE-754 que define diferentes formatos de números en punto flotante. Por ejemplo, el formato de precisión simple (`float` en C y C++), corresponde practicamente a $\mathbb{F}(2, 24, -125, 128)$. El formato de precisión doble (`double` en C y C++, y todos los variables en Matlab y Python por defecto) corresponde practicamente a $\mathbb{F}(2, 53, -1021, 1024)$. De (3.3) conluimos que para precisión doble,

$$\begin{aligned} x_{\max} &= (1 - 2^{-53}) 2^{1024} = 1.797693134862316 \cdot 10^{308} \\ x_{\min} &= 2^{-1022} = 2.225073858507201 \cdot 10^{-308} \end{aligned}$$

Verificamos estos números en el caso del tipo de dato estandar para números en Python:

```
In [11]: import sys
         sys.float_info

Out[11]: sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.22
```

Vemos los números $e_{\max}$ y $e_{\min}$,

```
In [15]: print(sys.float_info.max_exp)
         print(sys.float_info.min_exp)

1024
-1021
```

y los números mas grandes y pequeños en valor absoluto $x_{\max}$ y $x_{\min}$,

```
In [27]: xmax = sys.float_info.max
         xmin = sys.float_info.min
         print(xmax)
         print(xmin)

1.7976931348623157e+308
2.2250738585072014e-308
```

Un resultado mayor que $x_{\max}$, es decir un overflow, se representa en Python con Inf, y un resultado indefinido se representa con nan, Not-A-Number.

```
In [44]: x1 = xmax+xmax
         x2 = x1-x1

         print(x1)
         print(x2)

inf
nan
```

Una división por zero implica un error en Python:

```
In [46]: x3=1/0
```

```
---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-46-ff77b4aa583b> in <module>()
----> 1 x3=1/0


ZeroDivisionError: division by zero
```

Una variable de precisión simple necesita 32 bits de memoria, es decir 4 bytes[2]: un bit para el signo, 23 bits para los dígitos (son 24 dígitos, pero $a_1 = 1$ siempre), y 8 bits para el exponente. Una variable de precisión doble necesita 64 bits, es decir 8 bytes. Mencionamos que también existen Toolboxes en Matlab o bibliotecas para C o Python para representaciónes mas dinámicas que utilizan una cantidad de memoria variable para representar números de precisión cualquiera. Sin embargo, el sistema de punto flotante y el estándar IEEE-754 es lo mas común y útil para computación cientifica y simulaciones numéricas.

Una consecuencia de la representación punto flotante es que las operaciones aritméticas básicas no son cerradas. Por ejemplo,

$$\begin{aligned} 100 = 0.1 \cdot 10^3 &\in \mathbb{F}(10, 2, -5, 5), \\ 4 = 0.4 \cdot 10^1 &\in \mathbb{F}(10, 2, -5, 5), \\ 100 + 4 = 0.104 \cdot 10^3 &\notin \mathbb{F}(10, 2, -5, 5). \end{aligned}$$

Concluimos que para no salir de nuestro sistema de punto flotante al realizar operaciones aritmeticas, el resultado *exacto* de la operación tiene que transformarse a un número del sistema de punto flotante. Por lo tanto, necesitaremos un mecanismo para transformar un número en su número punto flotante mas cerca.

Definición 19. La operación $\mathrm{rd} : \mathbb{R} \to \mathbb{F}(b, t, e_{\min}, e_{\max})$ dada por

$$x = \sigma \left(\sum_{k=1}^{\infty} a_k b^{-k} \right) b^e \to \begin{cases} 0 & \text{si } |x| < x_{\min} \\ \texttt{overflow} & \text{si } |x| > x_{\max} \\ \pm(\sum_{k=1}^{t} a_k b^{-k}) b^e & \text{si } a_{t+1} < \frac{b}{2} \\ \pm(b^{-t} + \sum_{k=1}^{t} a_k b^{-k}) b^e & \text{si } a_{t+1} \geq \frac{b}{2} \end{cases}$$

se llama **redondeo**. El error relativo del redondeo está acotado por

$$\frac{|x - \mathrm{rd}(x)|}{|x|} \leq \frac{b^{1-t}}{2} =: \mathrm{eps} \quad \text{para todo } 0 < |x| \leq x_{\max}.$$

[2] 1 byte = 8 bits

□

El número eps se llama **precisión de la maquina** y representa la resolución del sistema punto flotante usado. La precisión de la maquina es el número x mas pequeño en valor absoluto tal que

$$\mathrm{rd}(1+x) \neq 1.$$

Para precisión doble, $\mathrm{eps} = 2^{-53} \approx 1.11 \cdot 10^{-16}$.
La precisión de la maquina la podemos verificar también en Python:

```
In [50]: eps = sys.float_info.epsilon
         print(eps)

2.220446049250313e-16


In [55]: print(1+eps)
         print(1+eps-1)
         print(1+eps/2)
         print(1+eps/2-1)

1.0000000000000002
2.220446049250313e-16
1.0
0.0
```

Para la aritmetica en un sistema de punto flotanto vamos a suponer que *los resultados de todas las operaciones aritmeticas básicas y todas las funciones básicas como trigonometricas, exponenciales, raices, en un sistema de punto flotante, son los resultados exactos redondeados.* Por ejemplo, si $\oplus$ es la suma en un sistema de punto flotante, eso significa

$$x \oplus y = \mathrm{rd}(x+y).$$

Hay varias consecuencias de la aritmetica en punto flotante.

- La aritmetica en una maquina pierde las propiedades conocidas: por ejemplo, en $\mathbb{F}(10, 2, -5, 5)$,

  $$100 \oplus 4 = \mathrm{rd}(100+4) = \mathrm{rd}(104) = \mathrm{rd}(0.104 \cdot 10^3) = 0.1 \cdot 10^3 = 100,$$
  $$(100 \oplus 4) \oplus 4 = 100 \oplus 4 = 100,$$

  pero

  $$4 \oplus 4 = \mathrm{rd}(8) = 8,$$
  $$100 \oplus (4 \oplus 4) = \mathrm{rd}(108) = \mathrm{rd}(0.108 \cdot 10^3) = 0.11 \cdot 10^3 = 110.$$

  Es decir, $(100 \oplus 4) \oplus 4 \neq 100 \oplus (4 \oplus 4)$, la suma no es asociativa.

- Convergencia numérica donde no hay convergencia analítica: La suma ármonico $\sum_{k=1}^{\infty} \frac{1}{k}$ es divergente. Sin embargo, si verificamos el comportamiento de la suma con aritmenta en punto flotante, entonces en algún momento $\frac{1}{j}$ es menor que la precisión de la maquina, y por lo tanto

$$\frac{1}{j} \oplus \sum_{k=1}^{j-1} \frac{1}{k} = \sum_{k=1}^{j-1} \frac{1}{k},$$

  es decir, numéricamente observamos converencia.

- Cancelación por resta: Ya sabemos que restar dos números casi iguales es un problema mal condicionado. En aritmetica flotante, este efecto se llama *cancelación*: Si queremos restar dos números casi iguales que *no son elementos del sistema de punto flotante usado*, entonces el error relativo de redondear estos números es menor que la precisión de la maquina. Sin embargo, al restar, este error se amplifica mucho por la mala condición de la resta.

Además, cada operación aritmetica introduce un error por redondeo. Un algoritmo es una lista de operaciones aritmeticas y reglas definidas, y concluimos que cada operacion introduce un error de redondeo que se propagará en las operaciones subsiguientes. Si una de las operaciones es mal condicionada (por ejemplo, la resta de dos números casi iguales), los errores de redonde que se producían en las operaciones anteriores se amplificarán tanto que el resultado del algoritmo llevará mucho error. En este caso, el algoritmo se llama **inestable**. En el caso contrario, se llama **estable**.

Un método numérico para un modelo matemático se puede implementar de diferentes formas (algoritmos), y no todas las formas (algoritmos) son estables.

Problema	Algoritmo	resultado número	co
bien condicionado	estable	confiable	
bien condicionado	inestable	no confiable	
mal condicionado		no confiable	

Como ejemplo extremo, consideramos el problema de evaluar la función $f : \mathbb{R} \to \mathbb{R}$ dada por $f(x) = 2$. Notamos que la condición del problema es $C_{1,1} = 0$. Esto tiene sentido, pues, la función es constante.

```
In [63]: def buenaImplementacion(x):
             y = 2
             return y

         def malaImplementacion(x):
             a = x+1
             b = x-1
```

```
            y = a-b
            return y

In [72]: buenaImplementacion(1e16)

Out[72]: 2

In [71]: malaImplementacion(1e16)

Out[71]: 0.0
```

3.3 Normas vectoriales y matriciales

Para medir errores en vectores y matrices es mas conveniente definir normas vectoriales y matriciales. Para matrices $A \in \mathbb{R}^{n\times n}$ y vectores $x \in \mathbb{R}^n$ usaremos la notación

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \quad \text{y} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

Ya sabemos que si X es un espacio vectorial sobre el campo de escalares $\mathbb{R}$, entonces una función $\|\cdot\| : X \to \mathbb{R}$ se llama *norma* y $(X, \|\cdot\|)$ *espacio normado*, si

(i) $\|x\| \geq 0$ para todo $x \in X$,

(ii) $\|x\| = 0$ si y solo si $x = 0$,

(iii) $\|\lambda x\| = |\lambda| \cdot \|x\|$ para todo $x \in X$ y $\lambda \in \mathbb{R}$,

(iv) $\|x + y\| \leq \|x\| + \|y\|$ para todo $x, y \in X$.

Las normas mas importantes sobre $X = \mathbb{R}^n$ son

- la *norma euclidiana* inducida por el producto interno $\|x\|_2 := \left(x^\top x\right)^{1/2} = \left(\sum_{j=1}^n |x_j|^2\right)^{1/2}$,
- las *normas p* $\|x\|_p := \left(\sum_{j=1}^n |x_j|^p\right)^{1/p}$ para $1 \leq p < \infty$,
- la *norma maxima* $\|x\|_\infty := \max_{1\leq j\leq n} |x_j|$.

Las normas sobre $\mathbb{R}^n$ inducen normas sobre el espacio de las matrices.

Lema 20. *Sean $\|\cdot\|_{\mathbb{R}^n}$ y $\|\cdot\|_{\mathbb{R}^m}$ normas vectoriales sobre $\mathbb{R}^n$ y $\mathbb{R}^m$ y define para $A \in \mathbb{R}^{m\times n}$*

$$\|A\| := \sup_{x\in\mathbb{R}^n\setminus\{0\}} \frac{\|Ax\|_{\mathbb{R}^m}}{\|x\|_{\mathbb{R}^n}}.$$

Entonces,

(i) $\|\cdot\| : \mathbb{R}^{m\times n} \to \mathbb{R}$ es una norma, y la llamamos norma matricial inducida.

(ii) Todos los supremos son maximos.

(iii) Para matrices $A \in \mathbb{R}^{m\times p}$ y $B \in \mathbb{R}^{p\times n}$ tenemos $\|A \cdot B\| \leq \|A\| \cdot \|B\|$.

(iv) Si $I_n \in \mathbb{R}^{n\times n}$ es la identidad, entonces $\|I_n\| = 1$.

□

La definición de la norma matricial implica

$$\|Ax\|_{\mathbb{R}^m} \leq \|A\|\|x\|_{\mathbb{R}^n} \quad \text{para todo } x \in \mathbb{R}^n. \tag{3.4}$$

En general es dificil calcular explicitamente una norma matricial, pero en algunos casos especiales se puede obtener otra representación mas útil para calculos numéricos. Recordamos que el *radio espectral* de una matriz A se define como

$$\rho(A) := \max\{|\lambda| \mid \lambda \in \mathbb{C} \text{ es un autovalor de } A\}.$$

Lema 21. *Si usamos sobre $\mathbb{R}^n$ y $\mathbb{R}^m$ las mismas normas vectoriales y anotamos las normas matriciales inducidas con la misma notación, entonces*

$$\|A\|_1 = \max_{k=1,\ldots,n} \sum_{j=1}^{m} |a_{j,k}|, \quad y \quad \|A\|_\infty = \max_{j=1,\ldots,m} \sum_{k=1}^{n} |a_{j,k}|, \quad y \quad \|A\|_2 = \rho(A^\top A)^{1/2}.$$

□

Ejemplo 22. *Calcule las normas $\|A\|_1$, $\|A\|_\infty$, y $\|A\|_2$ para*

$$A = \begin{pmatrix} 1 & 1 \\ 2 & 1 \end{pmatrix}.$$

□

Lema 23. *Sea $A \in \mathbb{R}^{n\times n}$. Entonces*

(i) $\rho(A) \leq \|A\|$ para cada norma matricial inducida,

(ii) para cada $\varepsilon > 0$ existe una norma matrical inducida $\|\cdot\|_\varepsilon$ tal que

$$\rho(A) \leq \|A\|_\varepsilon \leq \rho(A) + \varepsilon.$$

□

En Python existe el tipo *lista*, que puede ser usado para representar vectores y matrices. Sin embargo, el tipo *lista* es muy general y flexible. Para representar y calcular con vectores y matrices de números reales de una forma mas eficiente, usaremos el tipo *array*. Este no es un tipo estandar, es parte del paquete adicional *NumPy*. Destacamos que este tipo permite la *vectorización* de operaciones, es decir, podemos hacer operaciones matemáticas directamente con los arrays en vez de usar bucles. Es importante notar que los índices de vectores y matrices en Python empiezan con 0.

```
In [60]: import numpy as np
         from math import sin

         # vector con 3 elementos como array de numpy
         x = np.zeros(3)

         # dimension de x
         dimx = x.shape[0]
         print('la dimension de x es',dimx)

         # acceso a elementos
         # cuidado: arrays en numpy empiezan con índice 0!
         x[0]=17
         print('x =',x)

         y = np.zeros(3)
         y[1]=-4.5

         w = np.zeros(3)
         z = np.zeros(3)

         # operaciones con bucle
         for i in range(3):
             w[i]=x[i]+y[i]

         # operación vectorizada
         z = x+y

         print('w =',w)
         print('z =',z)

         np.sin(x)

         # operacion con bucle
         for i in range(3):
             w[i]=sin(x[i])

         # operación vectorizada
         z = np.sin(x)
```

```
          print('w =',w)
          print('z =',z)

          # producto interno
          a = np.dot(x,w)
          print(a)

la dimension de x es 3
x = [17.  0.  0.]
w = [17.  -4.5  0. ]
z = [17.  -4.5  0. ]
w = [-0.96139749  0.          0.        ]
z = [-0.96139749  0.          0.        ]
-16.343757361952466
```

```
In [2]: # matriz 2x4 como array de numpy
        A = np.zeros([2,4])

        # dimensiones de A
        dim = A.shape
        print(dim)
        n = dim[0]
        m = dim[1]
        print('A es de tamaño ',n,' por ',m)

        # acceso a elementos
        A[0,1] = 5
        A[1,3] = 17
        print(A)

(2, 4)
A es de tamaño  2  por  4
[[ 0.  5.  0.  0.]
 [ 0.  0.  0. 17.]]
```

Para calcular normas de vectores y matrices, podemos usar el comando *linalg.norm* de *NumPy*:

```
In [59]: from math import sqrt

         x = [1, 2, -4]

         print('norma euclidiana de x =',np.linalg.norm(x))
         print('norma euclidiana de x =',sqrt(np.dot(x,x)))
         print('norma infinita de x =',np.linalg.norm(x,np.inf))
         print('norma 1 de x =',np.linalg.norm(x,1))

         A = [[1,2,3],[4,5,6]]
         print('norma 2 de A =',np.linalg.norm(A))
         print('norma infinita de A =',np.linalg.norm(A,np.inf))
         print('norma 1 de A =',np.linalg.norm(A,1))

norma euclidiana de x = 4.58257569495584
norma euclidiana de x = 4.58257569495584
norma infinita de x = 4.0
norma 1 de x = 7.0
norma 2 de A = 9.539392014169456
norma infinita de A = 15.0
norma 1 de A = 9.0
```

3.4 Ejercicios

1. Sea $\phi(x) = \frac{1}{x} - \frac{1}{x+1}$.
   (a) Establezca el factor de condicionamiento $C_{1,1}$ de ϕ con respecto a x.
   (b) Verifique que el $C_{1,1} \leq 2$ para $x > 0$. ¿Si el error relativo en x está acotada por 10^{-5}, cual error relativo podemos esperar en ϕ?

2. Considere el problema de buscar las dos raízes reales de la ecuación $x^2 - 2px + 1 = 0$ para $p > 1$.
   (a) Verifique que la ecuación tiene dos raízes simples $x_1 \neq x_2$.
   (b) Define la función $f : \mathbb{R} \to \mathbb{R}^2$ que resuelve la ecuación, es decir, $f(p) = (x_1, x_2)$ y calcule los factores de condicionamiento de f con respecto a p.
   (c) Interprete sus resultados. ¿Como se comporta la condición del problema si p se acerca a 1, y que significa eso para las raízes?

3. Ya hemos visto en clases que restar dos números casi iguales es un problema mal condicionado. Verifique que los otros tres operaciones aritméticas básicas, es decir, suma, producto, y división, son problemas bien condicionados.

4. Encuentre números no zeros $x, y \in \mathbb{R}$ con $x \neq y$ tal que en `Matlab` (o cualquier sistema que usa el estándar IEEE-754) la operación

   ```
   >> x/y
   ```

   se evalua a
   (a) 0
   (b) 1
   (c) `Inf`
   (d) `NaN`

5. Determine si los siguientes números x son elementos del sistema de punto flotante indicado. Si no, determine $\mathrm{rd}(x)$.
   (a) $x = 7743.11265$ y $\mathbb{F}(10, 8, -5, 5))$
   (b) $x = 82.42242$ y $\mathbb{F}(10, 8, -5, 5)$
   (c) $x = 12.3312761$ y $\mathbb{F}(10, 8, -5, 5)$
   (d) $x = 12.3312769$ y $\mathbb{F}(10, 8, -5, 5)$
   (e) $x = 12.3312765$ y $\mathbb{F}(10, 8, -5, 5)$

6. Encuentre dos diferentes sistemas de punto flotanto normalizado en base 10, es decir $\mathbb{F}(10, t, e_{\min}, e_{\max})$, que tienen una precisión de la maquina igual a $5 \cdot 10^{-17}$.

7. Verifique que para todo vector $x \in \mathbb{R}^n$ se tiene

$$\|x\|_2 \leq \|x\|_1 \leq \sqrt{n}\|x\|_2,$$

y también

$$\|x\|_\infty \leq \|x\|_2 \leq \sqrt{n}\|x\|_\infty.$$

8. Sea $A \in \mathbb{R}^{3\times 3}$ dada por

$$A = \begin{pmatrix} -1 & 3 & 0 \\ 4 & 2 & -2 \\ 3 & 1 & 9 \end{pmatrix}.$$

Calcule $\|A\|_1$ y $\|A\|_\infty$.

9. Sea $A \in \mathbb{R}^{3\times 3}$ dada por

$$A = \begin{pmatrix} -1 & 3 & -1 \\ 0 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Calcule $\|A\|_2$.

Chapter 4

Sistemas lineales

El objetivo del presente capítulo es lo siguiente:

Dada una matriz $A \in \mathbb{R}^{n\times n}$ y un vector $b \in \mathbb{R}^n$, calcular $x \in \mathbb{R}^n$ tal que

$$Ax = b. \tag{4.1}$$

Notamos que (4.1) son n ecuaciones en n desconocidas,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{aligned}$$

Se llama *sistema lineal*, pues la matriz A introduce una aplicación lineal sobre el espacio $\mathbb{R}^n$. Para el resto del presente capítulo, vamos a considerar matrices invertibles. Si conocemos la matriz inversa A^{-1}, entonces es fácil resolver (4.1), pues, $x = A^{-1}b$.

```
In [1]: import numpy as np

In [2]: A = np.random.rand(10,10)
        b = np.random.rand(10)

        Ainv = np.linalg.inv(A)
        x = Ainv.dot(b)
```

Sin embargo, una regla general de Análisis Numérico es que **no se calcula la inversa de una matriz**, si no es absolutamente necesario. El sistema (4.1) lo podemos resolver sin calcular A^{-1},

usando el método de Gauss-Jordan[1]: mediante operaciones elementales de filas transformamos A a la matriz de identidad, y asi

$$(A \mid b) \to \cdots \to (I_n \mid x).$$

```
In [3]: A = np.random.rand(10,10)
        b = np.random.rand(10)

        x = np.linalg.solve(A,b)
```

En problemas reales, el tamaño n del sistema suele ser muy grande, es decir $n \approx 10^9$. Si queremos implementar algoritmos para resolver sistemas lineales, tenemos que tener en cuenta lo siguiente.

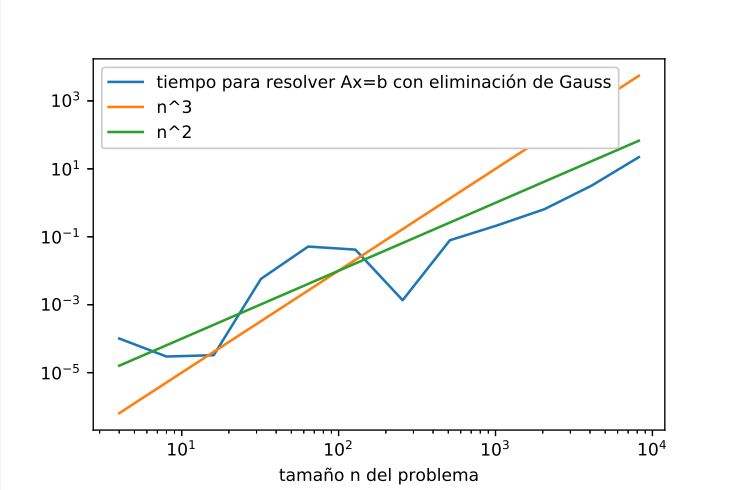
(1) **Costo de almacenamiento:** Una matriz $A \in \mathbb{R}^{n\times n}$ en doble precisión necesita $8 \cdot n^2$ bytes de memoria. Por ejemplo, para $n = 10^6$ necesitamos aproximadamente 8000 Gigabyte de memoria. Destacamos que todos los datos que se quieren usar en una calculación deberian estar en la *memoria de acceso aleatorio* (RAM) de la maquina.

(2) **Costo operacional:** El tiempo de cálculo necesario debe ser lo menor posible. Una medida standard del costo operacional es la cantidad de operaciones aritméticas $(+, -, \cdot, /)$ que requiere un algoritmo. La unidad de una operacion aritmética se llama **flop** (floating point operation). Usualmente, la cantidad de flops es un polinomio en n. Por ejemplo, para calcular el producto Ax con $A \in \mathbb{R}^{n\times n}$ se necesita $n(2n-1) = 2n^2 - n$ flops. Para n grande, el termino n es despreciable comparado con el termino $2n^2$, y se dice que el costo operacional es **asintóticamente** n^2. Visualizaremos la importancia del costo computacional con un ejemplo. La regla de Cramer dice que la solución x de (4.1) es dada por $x_j = \det(A_j)/\det(A)$, donde A_j es la matriz que se obtiene reemplazando en A su columna j-ésima por b. Si los determinantes se calculan mediante la fórmula recursiva usual, el costo operacional de calcular x es asintóticamente $(n+1)!$. La eliminación de Gauss, por el otro lado, tiene un costo asintótico de n^3. En un computador moderno con 1 Gflop por segundo, se obtiene para $n = 20$ un tiempo computacional de 10^{-5} segundos para la eliminación de Gauss, pero 1500 años para la regla de Cramer.

```
In [16]: from timeit import default_timer as timer
         from matplotlib.pyplot import *

         N=14

         problemsize = np.zeros(N-1)
         time = np.zeros(N-1)
```

[1] como en MAT-022

```
for k in range(2,N+1):
    n = 2**k
    A = np.random.rand(n,n)
    b = np.random.rand(n)
    start = timer()
    x = np.linalg.solve(A,b)
    end = timer()
    problemsize[k-2] = n
    time[k-2] = end-start

loglog(problemsize,time,problemsize,1e-8*problemsize**3,problemsize,1e-6*prob
xlabel('tamaño n del problema')
legend(['tiempo para resolver Ax=b con eliminación de Gauss','n^3','n^2'])
savefig('time_Gauss.eps')
show()
```



4.1 La condición de una matriz

Queremos estudiar el condicionamiento del problema (4.1), es decir, como se amplifica un error en b en la solucion x. Podriamos definir una función $f : \mathbb{R}^n \to \mathbb{R}^n$ por $f(b) = x$ y estudiar sus n^2 factores de condicionamiento. En el contexto del problema (4.1) existe un concepto mas simple de condicionamiento.

Teorema 24. *Sea $A \in \mathbb{R}^{n\times n}$ una matriz invertible y $x, \Delta x, b, \Delta b \in \mathbb{R}^n$ vectores con*

$$Ax = b \quad y \quad A(x + \Delta x) = b + \Delta b.$$

Entonces,

$$\frac{\|\Delta x\|_2}{\|x\|_2} \leq \|A\|_2 \cdot \|A^{-1}\|_2 \frac{\|\Delta b\|_2}{\|b\|_2}$$

Proof. Por linealidad de A tenemos $\Delta x = A^{-1}\Delta b$, y por (3.4),

$$\|\Delta x\|_2 = \|A^{-1}\Delta b\|_2 \leq \|A^{-1}\|_2 \|\Delta b\|_2.$$

También por (3.4),

$$\|b\|_2 \leq \|A\|_2 \|x\|_2.$$

Multiplicando las dos desigualdades muestra el resultado. □

Por el último resultado, el número

$$\text{cond}(A) := \|A\|_2 \cdot \|A^{-1}\|_2$$

se llama *condición* de A.

4.2 Matrices triangulares

Primero consideraremos un caso especial.

Definición 25. Una matriz $L \in \mathbb{R}^{n\times n}$ se llama **triangular inferior**, si $\ell_{jk} = 0$ para $k > j$. Es decir, todos los elementos arriba de la diagonal son zero. Una matriz $U \in \mathbb{R}^{n\times n}$ se llama **triangular superior**, si $u_{jk} = 0$ para $j > k$. Es decir, todos los elementos abajo de la diagonal son zero[2]. Las matrices tienen entonces la forma

$$L = \begin{pmatrix} \ell_{11} & 0 & \dots & \dots & 0 \\ \ell_{21} & \ell_{22} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & & \ddots & 0 \\ \ell_{n1} & \ell_{n2} & \dots & \dots & \ell_{nn} \end{pmatrix}, \quad y \quad U = \begin{pmatrix} u_{11} & u_{12} & \dots & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & u_{nn} \end{pmatrix}.$$

□

[2] L por *lower* y U por *upper* en ingles.

Para resolver un sistema $Ux = b$ con una matriz triangular superior, notamos que en terminos de ecuaciones se lee

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-1}x_{n-1} + u_{1n}x_n &= b_1 \\ u_{22}x_2 + \cdots + u_{2,n-1}x_{n-1} + u_{2n}x_n &= b_2 \\ &\vdots \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= b_{n-1} \\ u_{nn}x_n &= b_n. \end{aligned}$$

Empezando con la última ecuacion calculamos primero

$$x_n = \frac{b_n}{u_{nn}}.$$

Ahora conocemos x_n, y usando la penúltima ecuación podemos calcular

$$x_{n-1} = \frac{b_{n-1} - u_{n-1,n}x_n}{u_{n-1,n-1}}.$$

Ahora conocemos x_n y x_{n-1}, y podemos calcular

$$x_{n-2} = \frac{b_{n-2} - u_{n-2,n}x_n - u_{n-2,n-1}x_{n-1}}{u_{n-2,n-2}}.$$

Una vez que conocemos $x_n, x_{n-1}, \ldots, x_{j+1}$, podemos caluclar x_j usando la j-ésima ecuación

$$x_j = \frac{b_j - \sum_{k=j+1}^{n} u_{jk}x_k}{u_{jj}}.$$

Este procedimiento se llama *sustitución ascendente.*

Ejemplo 26. *Resolvamos el sistema*

$$\begin{aligned} 4x_1 - x_2 + 2x_3 + 3x_4 &= 20 \\ -2x_2 + 7x_3 - 4x_4 &= -7 \\ 6x_3 + 5x_4 &= 4 \\ 3x_4 &= 6, \end{aligned}$$

usando sustitución ascendente.

La última ecuación implica

$$x_4 = \frac{6}{3} = 2.$$

Entoces, usando el resultado anterior, la penúltima ecuación implica

$$x_3 = \frac{4 - 5 * 2}{6} = -1.$$

Usando $x_4 = 2$ y $x_3 = -1$, la segunda ecuación implica

$$x_2 = \frac{-7 + 4 * 2 - 7 * (-1)}{-2} = -4.$$

Finalmente, la primera ecuación implica

$$x_1 = \frac{20 - 3 * 2 - 2 * (-1) + 1 * (-4)}{4} = 3.$$

□

```
In [51]: def sustitucionAscendente(U,b):
             # resolver Ux=b, donde
             # U es triangular superior

             n = U.shape[0]
             x = np.zeros(n)
             aux = 0

             for j in reversed(range(0,n)):
                 aux = b[j]
                 for k in range(j,n):
                     aux = aux - U[j,k]*x[k]
                 x[j]=aux/U[j,j]

             return x
```

Corolario 27. *La sustitucion ascendente calcula la solución x del sistema $Ux = b$ con matriz triangular superior en asintóticamente n^2 flops.*

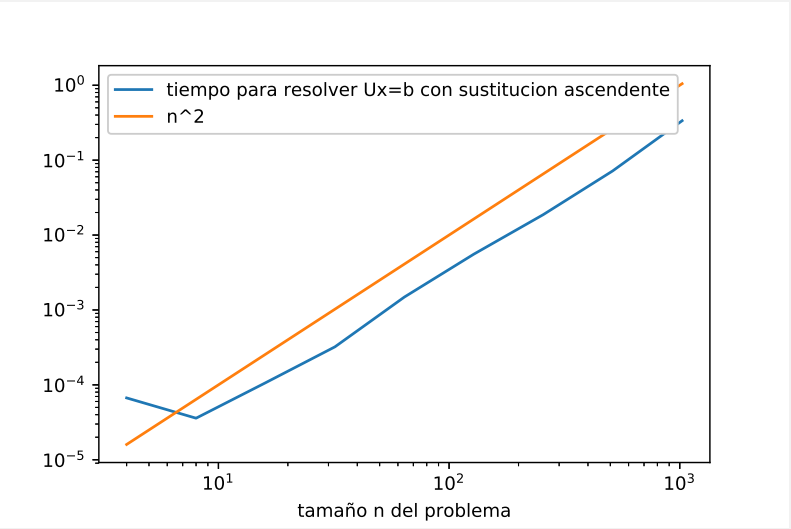
Proof. Ya sabemos que si U es invertible, entonces los elementos de su diagonal no son zeros. Por lo tanto, no se produce una division por zero y el algoritmo está bien definido. En el paso j se calcula $n - j$ productos y restas y 1 division. El número total de operaciones es

$$\sum_{j=1}^{n}(1 + 2(n - j)) = n + 2\sum_{k=1}^{n-1} k = n + \frac{(n-1)n}{2} = n^2.$$

□

In [59]:
```
from timeit import default_timer as timer
from matplotlib.pyplot import *

N=10

problemsize = np.zeros(N-1)
time = np.zeros(N-1)

for k in range(2,N+1):
    n = 2**k
    # generar matriz triangular superior al azar
    A = np.transpose(np.tril(np.random.rand(n,n)))
    b = np.random.rand(n)
    start = timer()
    x = sustitucionAscendente(A,b)
    end = timer()
    problemsize[k-2] = n
    time[k-2] = end-start

loglog(problemsize,time,problemsize,1e-6*problemsize**2)
xlabel('tamaño n del problema')
legend(['tiempo para resolver Ux=b con sustitucion ascendente','n^2'])
```

Out[59]:
```
<matplotlib.legend.Legend at 0x7f4091b1f3d0>
```



La misma idea se aplica al sistema $Lx = b$ con matriz trianglar inferior. En este caso, el algoritmo se llama *sustitución descendente*. Notamos un par de propiedades de matrices triangulares.

Teorema 28. *(1) El producto de dos matrices triangulares inferiores (superiores) es triangular inferior (superior).*

(2) La inversa de una matriz triangular inferior (superior) es inferior (superior).

(3) El determinante de una matriz triangular es el producto de los elementos en la diagonal, $\det(L) = \prod_{j=1}^{n} \ell_{jj}$*, respectivamente* $\det(U) = \prod_{j=1}^{n} u_{jj}$*. Por lo tanto una matriz triangular es invertible si y solo si todos los elementos en la diagonal no son zeros.*

Ya sabemos que un sistema tiene única solución si y solo si el determinante de la matriz no es zero. El determinante de una matriz triangular es el producto de sus elementos diagonales según el último Teorema, y este determinante no es zero si y solo si ningun elemento en la diagonal es zero. Los elementos de la diagonal son justamente los que aparecen en el denominador durante el proceso de sustitución ascentente.

4.3 La factorización LU

4.3.1 Operaciones y matrices elementales

Para resolver el sistema $Ax = b$, el método mas comun es la **eliminación de Gauss**. Usando operaciones elementales fila, se transforma el sistema $Ax = b$ a un sistema equivalente (es decir, un sistema que tiene exactamente la misma solución) con una matriz triangular superior $Ux = \widetilde{b}$, lo cuál se puede resolver usando sustitución ascendente. Recordamos que hay tres operaciones elementales fila.

Definición 29. Las tres operaciones elementales fila son

(i) Multiplicar fila k con el número α, $E_k(\alpha)$,

(ii) Multiplicar fila k con el número α y sumar a la fila j, $E_{jk}(\alpha)$.

(iii) Intercambiar filas j y k, E_{jk},

□

En este contexto es comodo usar la notación de la *matriz aumentada* $(A|b)$, a la cual se aplican las operaciones elementales. Por ejemplo, para resolver el sistema $Ax = b$ con

$$A = \begin{pmatrix} 1 & -3 & 2 \\ -2 & 8 & -1 \\ 4 & -6 & 5 \end{pmatrix}, \qquad b = \begin{pmatrix} 1 \\ 4 \\ 8 \end{pmatrix},$$

generamos la matriz aumentada y aplicamos operaciones elementales fila para generar zeros abajo de la diagonal,

$$\begin{aligned} (A|b) = \left(\begin{array}{ccc|c} 1 & -3 & 2 & 1 \\ -2 & 8 & -1 & 4 \\ 4 & -6 & 5 & 8 \end{array}\right) &\xrightarrow[E_{3,1}(-4)]{E_{2,1}(2)} \left(\begin{array}{ccc|c} 1 & -3 & 2 & 1 \\ 0 & 2 & 3 & 6 \\ 0 & 6 & -3 & 4 \end{array}\right) \\ &\xrightarrow{E_{3,2}(-3)} \left(\begin{array}{ccc|c} 1 & -3 & 2 & 1 \\ 0 & 2 & 3 & 6 \\ 0 & 0 & -12 & -14 \end{array}\right) = (U|\widetilde{b}). \end{aligned} \tag{4.2}$$

El sistema final $Ux = \widetilde{b}$ lo podemos resolver con sustitución ascendente.

En lo que sigue, vamos a representar este proceso en forma matricial. Cada operacion elemental fila de la definición 29 puede ser representada por multiplicación por la izquierda con su *matriz elemental* asociada.

Lema 30. *Sea $A \in \mathbb{R}^{n\times n}$ y $I_n \in \mathbb{R}^{n\times n}$ la matriz de identidad.*

*(a) Sea E la **matriz elemental** de una de las operaciones elementales de la definición 29, es decir, la matriz que se obtiene aplicando la operación elemental a la matriz de identidad I_n. Entonces, el resultado de aplicar la operación elemental a una matriz A es igual a $E \cdot A$.*

*(b) Sea P una matriz que se obtiene aplicando una seria de cambios de filas a la matriz de identidad I_n. Entonces P se llama **matriz de permutación**. Se tiene $P^{-1} = P^\top$.*

Para una matriz elemental vamos a usar la misma notación de la operación elemental. Por ejemplo, la operación $E_{2,1}(\alpha)$ en $\mathbb{R}^3$ se representa por la matriz elemental

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{E_{2,1}(\alpha)} \begin{pmatrix} 1 & 0 & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = E_{2,1(\alpha)}.$$

Es decir, la operación elemental

$$\begin{pmatrix} 1 & -3 & 2 \\ -2 & 8 & -1 \\ 4 & -6 & 5 \end{pmatrix} \xrightarrow{E_{2,1}(2)} \begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 4 & -6 & 5 \end{pmatrix} \tag{4.3}$$

la podemos representar en su forma matricial

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -3 & 2 \\ -2 & 8 & -1 \\ 4 & -6 & 5 \end{pmatrix} = \begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 4 & -6 & 5 \end{pmatrix}.$$

Por otro lado, todos las cambios de fila en $\mathbb{R}^3$ son

$$\underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{=E_{12}}, \underbrace{\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}}_{=E_{13}}, \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}}_{=E_{23}}.$$

Es decir, la operación elemental

$$\begin{pmatrix} 1 & -3 & 2 \\ -2 & 8 & -1 \\ 4 & -6 & 5 \end{pmatrix} \xrightarrow{E_{1,3}} \begin{pmatrix} 4 & -6 & 5 \\ -2 & 8 & -1 \\ 1 & -3 & 2 \end{pmatrix} \tag{4.4}$$

la podemos representar en su forma matricial

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & -3 & 2 \\ -2 & 8 & -1 \\ 4 & -6 & 5 \end{pmatrix} = \begin{pmatrix} 4 & -6 & 5 \\ -2 & 8 & -1 \\ 1 & -3 & 2 \end{pmatrix}.$$

En Python podemos aplicar operaciones elementales de forma clásica con loops. Por ejemplo, la operación elemental (ii) se implementa como

```
In [38]: import numpy as np
         A = np.asarray([[1,2,3],[4,5,6],[7,8,9]])
         print(A)

         # aplicar E_{2,1}(-4) a una matriz A
         m = A.shape[1]
         for k in range(m):
             A[1,k] = (-4)*A[0,k] + A[1,k]

         print(A)

[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[ 1  2  3]
 [ 0 -3 -6]
 [ 7  8  9]]
```

Sin embargo en Python existe la noción de *vectorización*, es decir, acceso a y operaciones con vectores completos. Por ejemplo, A[1,3] da acceso al elemento $A_{1,3}$ (es decir, $A_{2,4}$ en notación matemática). Cambiamos uno de los índices por un doble punto :, tenemos acceso a la fila/-columna completa:

```
In [42]: print(A[1,:]) # segunda fila de A
         print(A[:,0]) # primera columna de A

[ 0 -3 -6]
[1 0 7]
```

Mas aún: Si usamos dos listas de índices,, podemos acceder a varios elementos. Eso se llama *broadcasting*. Es imporante que ambas listas tengan la misma dimension. En vez de una de las dos listas podemos usar el doble punto :, que da acceso a todas las filas/columnas:

```
In [62]: j = [0,2] # fila 0 y 2
         k = [1,2] # columna 1 y 2

         print(A[j,k])

         print(A[j,:]) # filas 0 y 2
```

```
        print(A[:,k]) # columnas 1 y 2

        print(A[j,1]) # segundo elemento de filas 0 y 2
```

```
[2 9]
[[1 2 3]
 [7 8 9]]
[[ 2  3]
 [-3 -6]
 [ 8  9]]
[2 8]
```

Con esta notación podemos escribir las operaciones elementales filas de una manera muy corta:

```
In [80]: A[[0,2],:] = A[[2,0],:] # cambiar filas 0 y 2
         A[1,:] = 2*A[1,:] # multiplicar fila 1 con 2
         A[2,:] = A[2,:] + (-3)*A[0,:]
```

Si queremos extraer una fila/columna a partir de un índice j hasta el final, podemos usar la notación doble punto despúes del índice:

```
In [72]: B = np.random.rand(10,10)
         print(B[2,7:]) # fila 2, a partir de columna 7}
         print(B[6:,0]) # columna 0, a partir de fila 6
```

4.3.2 Factorización LU sin pivote

Ahora podemos obtener una representación matricial de la eliminación de Gauss (4.2). Primero notamos que la operación elemental fila $E_{2,1}(2)$ es invertible, y su inversa es obviamente $E_{2,1}(-2)$. Es decir, (4.3) se puede escribir como

$$\begin{pmatrix} 1 & -3 & 2 \\ -2 & 8 & -1 \\ 4 & -6 & 5 \end{pmatrix} \xleftarrow{E_{2,1}(-2)} \begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 4 & -6 & 5 \end{pmatrix},$$

y según Lema 30 eso se lee en forma matricial

$$\begin{pmatrix} 1 & -3 & 2 \\ -2 & 8 & -1 \\ 4 & -6 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 4 & -6 & 5 \end{pmatrix}.$$

La segunda operación elemental $E_{3,1}(-4)$ se explicita entonces como

$$\begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 4 & -6 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 4 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 0 & 6 & -3 \end{pmatrix},$$

y la tercera $E_{3,2}(-3)$ como

$$\begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 0 & 6 & -3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 0 & 0 & -12 \end{pmatrix},$$

Finalmente, podemos representar (4.2) como

$$\begin{aligned} \begin{pmatrix} 1 & -3 & 2 \\ -2 & 8 & -1 \\ 4 & -6 & 5 \end{pmatrix} &= \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 4 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 0 & 0 & -12 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -3 & 2 \\ 0 & 2 & 3 \\ 0 & 0 & -12 \end{pmatrix}, \end{aligned}$$

es decir $A = L \cdot U$ con matriz triangular inferior L y matriz triangular superior U. La matriz U es el resultado de la eliminación de Gauss, y la matriz L contiene todos los factores que usamos en el proceso. Este procedimiento se extiende obviamente a matrices $A \in \mathbb{R}^{n \times n}$. Sea

$$A := A^{(1)} := \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}.$$

(1) **Paso 1:** Multiplicar $m_{j1} := a_{j1}/a_{11}$ con fila 1 y restar de fila j, $j > 1$:

$$A^{(1)} = \underbrace{\begin{pmatrix} 1 & & & & \\ m_{21} & 1 & & & \\ m_{31} & & 1 & & \\ \vdots & & & \ddots & \\ m_{n1} & & & & 1 \end{pmatrix}}_{:=L^{(1)}} \cdot \underbrace{\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ & \vdots & \ddots & \vdots \\ & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix}}_{:=A^{(2)}}$$

(2) **Paso 2:** Multiplicar $m_{j2} := a_{j2}^{(2)}/a_{22}^{(2)}$ con fila 2 y restar de fila j, $j > 2$:

$$A^{(2)} = \underbrace{\begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & m_{32} & 1 & & \\ & \vdots & & \ddots & \\ & m_{n2} & & & 1 \end{pmatrix}}_{:=L^{(2)}} \cdot \underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ & a_{22}^{(2)} & a_{23}^{(2)} & \dots & a_{2n}^{(2)} \\ & & a_{33}^{(3)} & \dots & a_{3n}^{(3)} \\ & & \vdots & \ddots & \vdots \\ & & a_{n3}^{(3)} & \dots & a_{nn}^{(3)} \end{pmatrix}}_{:=A^{(3)}}$$

(3) **Paso k:** Tenemos

$$A^{(k)} = \begin{pmatrix} a_{11} & \dots & \dots & \dots & \dots & a_{nn} \\ & a_{22}^{(2)} & & & & a_{2n}^{(2)} \\ & & \ddots & & & \vdots \\ & & & a_{kk}^{(k)} & \dots & a_{kn}^{(k)} \\ & & & \vdots & & \vdots \\ & & & a_{nk}^{(k)} & \dots & a_{nn}^{(k)} \end{pmatrix}$$

Multiplicamos $m_{jk} := a_{jk}^{(k)}/a_{kk}^{(k)}$ con fila k y restar de fila j, $j > k$:

$$A^{(k)} = \underbrace{\begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & 1 & & \\ & & & m_{(k+1)k} & 1 & \\ & & & \vdots & & \ddots \\ & & & m_{nk} & & 1 \end{pmatrix}}_{:=L^{(k)}} \cdot \underbrace{\begin{pmatrix} a_{11} & \dots & \dots & \dots & \dots & \dots & a_{nn} \\ & a_{22}^{(2)} & & & & & a_{2n}^{(2)} \\ & & \ddots & & & & \vdots \\ & & & a_{kk}^{(k)} & a_{k(k+1)}^{(k)} & \dots & a_{kn}^{(k)} \\ & & & & a_{(k+1)(k+1)}^{(k+1)} & \dots & a_{(k+1)n}^{(k+1)} \\ & & & & \vdots & \ddots & \vdots \\ & & & & a_{n(k+1)}^{(k+1)} & \dots & a_{nn}^{(k)} \end{pmatrix}}_{:=A^{(k+1)}}$$

Despues de $(n-1)$ pasos obtenemos

$$A = L^{(1)} \cdot L^{(2)} \cdot \dots \cdot L^{(n-1)} \cdot A^{(n)},$$

donde $A^{(n)}$ es triangular superior y los $L^{(k)}$ son triangulares inferiores. Por el Teorema 28, la matriz $L := L^{(1)} \cdot L^{(2)} \cdot \cdots \cdot L^{(n-1)}$ es triangular inferior. Mas aún, es fácil calcular

$$L = \begin{pmatrix} 1 & & & & & & \\ m_{21} & 1 & & & & & \\ m_{31} & m_{32} & \ddots & & & & \\ \vdots & \vdots & \vdots & 1 & & & \\ \vdots & \vdots & \vdots & m_{(r+1)r} & 1 & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\ m_{n1} & m_{n2} & \dots & m_{nr} & \dots & \dots & 1 \end{pmatrix}.$$

Es decir, podemos expresar A como un producto de una matriz triangular inferior con una matriz triangular superior.

Teorema 31 (Factorización LU sin pivote)**.** *Sea $A \in \mathbb{R}^{\times n}$ una matriz invertible. Entonces A tiene una factorización LU, $A = L \cdot U$, con L triangular inferior e invertible, U triangular superior e invertible, si y solo si la eliminación de Gauss se puede llevar a cabo sin intercambiar filas. La eliminación de Gauss calcula la descomposicón de LU en asintóticamente n^3 flops.* □

Una primera implementación de la factorización LU sin pivote será lo siguiente:

```
In [26]: import numpy as np

         def myLU(A):
             n = A.shape[0]

             # generar matriz de identidad para L
             L = np.identity(n)

             # loop sobre columnas de A
             for k in range(0,n):

                 # loop sobre filas de A
                 for j in range(k+1,n):

                     # calcular factor m
                     L[j,k] = A[j,k]/A[k,k];

                     # operacion elemental fila
                     for ll in range(k,n):
                         A[j,ll] = A[j,ll] - L[j,k]*A[k,ll]
```

```
            return (L,A)

In [65]: A = np.asarray([[1,2,3],[4,5,6],[7,8,1]])
         (L,U) = myLU(A)
         print(L)
         print(U)

         # verificar resultado,
         # multiplicando L y U
         AA = np.matmul(L,U)
         print(AA)

[[1. 0. 0.]
 [4. 1. 0.]
 [7. 2. 1.]]
[[ 1  2  3]
 [ 0 -3 -6]
 [ 0  0 -8]]
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 1.]]
```

Notamos que esta implementación asigna solamente memoria para el factor L, y sobreescribe A con el factor U.

4.3.3 Pivoteo parcial

El problema obvio con la la descomposición LU sin pivote (es decir, con la eliminación de Gauss sin intercambiar filas) es que no se puede llevar a cabo si nos encontramos con un elemento en la diagonal que es zero. Por ejemplo, como caso extremo consideramos

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

La matriz A es invertible, pero no podemos hacer ni el primer paso en eliminación de Gauss por $a_{11} = 0$. No obstante, observamos que con un cambio de filas

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \Leftrightarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_2 \\ b_1 \end{pmatrix}$$

obtenemos una matriz donde la eliminación de Gauss se puede llevar a cabo. En general, en el paso k de la descomposición LU tenemos

$$A^{(k)} = \begin{pmatrix} a_{11} & \dots & \dots & \dots & \dots & a_{nn} \\ & a_{22}^{(2)} & & & & a_{2n}^{(2)} \\ & & \ddots & & & \vdots \\ & & & a_{kk}^{(k)} & \dots & a_{kn}^{(k)} \\ & & & \vdots & & \vdots \\ & & & a_{nk}^{(k)} & \dots & a_{nn}^{(k)}. \end{pmatrix}$$

Si $a_{kk}^{(k)} = 0$, entonces no podemos seguir con la eliminación de Gauss, pues, los m_{jk} no están bien definidos por una división por zero. En este caso es necesario aplicar un cambio de filas con el fin de obtener un elemento no zero en la diagonal. Aun si $a_{kk}^{(k)} \neq 0$, es aconsejable aplicar un cambio de fila con el fin de obtener un elemento en la diagonal que sea mayor en valor absoluto. Este procedimiento resulta en un algoritmo más estable, que se llama **factorización LU con pivote parcial**. Si la matriz A es invertible, entonces se puede demostrar que existe por lo menos un elemento en la columna restante k de $A^{(k)}$

$$\begin{pmatrix} a_{kk}^{(k)} \\ a_{k+1,k}^{(k)} \\ a_{k+2,k}^{(k)} \\ \vdots \\ a_{n,k}^{(k)} \end{pmatrix}$$

que no es zero. Como ya mencionado, buscamos una fila $\ell \in \{k, \dots, n\}$ con elemento $a_{\ell k}^{(k)}$ mas grande en modulo,

$$|a_{\ell k}^{(k)}| \geq \max_{i=k,\dots,n} |a_{ik}^{(k)}|.$$

Notamos que $a_{\ell k}^{(k)} \neq 0$ si A es invertible, y aplicamos la operacion elemental de cambio de filas $E_{\ell k}$,

$$A^{(k)} = \begin{pmatrix} a_{11} & \dots & \dots & \dots & \dots & a_{nn} \\ & a_{22}^{(2)} & & & & a_{2n}^{(2)} \\ & & \ddots & & & \vdots \\ & & & a_{kk}^{(k)} & \dots & a_{kn}^{(k)} \\ & & & \vdots & & \vdots \\ & & & a_{\ell k}^{(k)} & \dots & a_{\ell n}^{(k)} \\ & & & \vdots & & \vdots \\ & & & a_{nk}^{(k)} & \dots & a_{nn}^{(k)} \end{pmatrix} = E_{\ell k} \cdot \begin{pmatrix} a_{11} & \dots & \dots & \dots & \dots & a_{nn} \\ & a_{22}^{(2)} & & & & a_{2n}^{(2)} \\ & & \ddots & & & \vdots \\ & & & a_{\ell k}^{(k)} & \dots & a_{\ell n}^{(k)} \\ & & & \vdots & & \vdots \\ & & & a_{kk}^{(k)} & \dots & a_{kn}^{(k)} \\ & & & \vdots & & \vdots \\ & & & a_{nk}^{(k)} & \dots & a_{nn}^{(k)} \end{pmatrix} = E_{\ell k} \cdot \widetilde{A}^{(k)}.$$

A la matriz $\widetilde{A}^{(k)}$ podemos aplicar el próximo paso en la eliminación de Gauss, pues $a_{\ell k}^{(k)} \neq 0$. Multiplicamos $m_{jk} := a_{jk}^{(k)} / a_{\ell k}^{(k)}$ con fila k y restar de fila j, $j > k$:

$$A^{(k)} = E_{\ell k} \cdot \underbrace{\begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & 1 & & \\ & & & m_{(k+1)k} & 1 & \\ & & & \vdots & & \ddots \\ & & & m_{nk} & & 1 \end{pmatrix}}_{:=L^{(k)}} \cdot \underbrace{\begin{pmatrix} a_{11} & \dots & \dots & \dots & \dots & \dots & a_{nn} \\ & a_{22}^{(2)} & & & & & a_{2n}^{(2)} \\ & & \ddots & & & & \vdots \\ & & & a_{jk}^{(k)} & a_{j(k+1)}^{(k)} & \dots & a_{jn}^{(k)} \\ & & & & a_{(k+1)(k+1)}^{(k+1)} & \dots & a_{(k+1)n}^{(k+1)} \\ & & & & \vdots & \ddots & \vdots \\ & & & & a_{n(k+1)}^{(k+1)} & \dots & a_{nn}^{(k)} \end{pmatrix}}_{:=A^{(k+1)}}.$$

Al final obtendremos

$$A = P^{(1)} \cdot L^{(1)} \cdot P^{(2)} \cdot L^{(2)} \cdot \dots \cdot P^{(n-1)} \cdot L^{(n-1)} A^{(n)},$$

donde las $L^{(k)}$ representan las eliminaciones de Gauss, y las $P^{(k)}$ son las matrices de permutación de los cambios de filas (si no hay cambio de filas, entonces $P^{(k)} = I_n$ la identidad). Todos los cambios de filas que se producen durante el algoritmo corresponden a una permutación de filas de la matriz original A.

Teorema 32 (Factorización LU con pivote parcial)**.** *Sea $A \in \mathbb{R}^{\times n}$ una matriz invertible. Entonces existe una matriz de permutación P tal que $P \cdot A$ tiene una descomposición LU, es decir $P \cdot A = L \cdot U$, con L triangular inferior e invertible, U triangular superior e invertible. La eliminación de Gauss con pivoteo parcial calcula la descomposicón de LU con pivote parcial en asintóticamente n^3 flops.* □

Tambien se puede hacer un **pivote total**, donde se aplican tambien cambios de columnas. En este caso, la descomposición será $P \cdot A \cdot Q = L \cdot U$, donde P es una matriz de permutación reflejando los cambios de filas, y Q es una matriz de permutación reflejando los cambios de columnas.

Ejemplo 33. *Calcule la descomposición LU con pivote parcial de la matriz*

$$A = \begin{pmatrix} 0.5 & 2 & 8.75 \\ 1 & 2 & 3 \\ 0.5 & 5 & 6.5 \end{pmatrix}.$$

(1) El pivote parcial de la primera columna de A es 1, y

$$A = \begin{pmatrix} 0.5 & 2 & 8.75 \\ 1 & 2 & 3 \\ 0.5 & 5 & 6.5 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 0.5 & 2 & 8.75 \\ 0.5 & 5 & 6.5 \end{pmatrix}.$$

Ahora eliminamos los elementos en la primer columna,

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0 & 1 \end{pmatrix} \cdot \underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 7.25 \\ 0 & 4 & 5 \end{pmatrix}}_{:=A^{(2)}}$$

(2) Consideramos $A^{(2)}$. El pivote parcial de la segunda columna es 4, y por lo tanto

$$A^{(2)} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 7.25 \\ 0 & 4 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 1 & 7.25 \end{pmatrix}.$$

Ahora eliminamos los elementos en la segunda columna,

$$A^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.25 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix}$$

Se tiene entonces:

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.25 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix}$$

Recordamos que permutaciones por la izquierda afectan a filas, y permutaciones por la derecha afectan a columnas. Por lo tanto,

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 0 & 1 \\ 0.5 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0 & 1 \end{pmatrix}$$

Finalmente,

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.25 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix}$$
$$= \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0.25 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix}.$$

Además,

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}^{\top} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix},$$

y asi

$$\underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}}_{P} \cdot A = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0.25 & 1 \end{pmatrix}}_{L} \cdot \underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix}}_{U}.$$

□

Una primera implementación de la factorización LU con pivote parcial será lo siguiente. Para buscar un índice de un elemento mas grande en modulo, usamos los comandos `np.abs`, que calcula los valores absolutos por componentes, y `np.argmax`, que calcula el índice del elemento máximo de un vector. Si hay varios elementos maximales, entonces `np.argmax` devuelve el menor índice:

```
In [18]: import numpy as np

         print(np.abs([1,2,3,0,-3,-2,-1]))
         print(np.argmax([1,2,3,0,-3,-2,-1]))
         print(np.argmax(np.abs([-1,-2,-3])))

[1 2 3 0 3 2 1]
2
2
```

```
In [26]: def myLU(A):
             n = A.shape[0]
```

```
            # generar matrices de identidad para L y P
            L = np.eye(n, dtype=np.double)
            P = np.eye(n, dtype=np.double)

            # loop sobre columnas de A
            for k in range(0,n):

                # buscar elemento mas grande en modulo
                # en columna restante
                # cuidado: hay que sumar k!
                i = k+np.argmax(np.abs(A[k:,k]))

                # aplicar cambio de filas y/o columnas
                L[[k,i],:] = L[[i,k],:]
                L[:,[k,i]] = L[:,[i,k]]
                P[:,[k,i]] = P[:,[i,k]]
                A[[k,i],:] = A[[i,k],:]

                # loop sobre filas de A
                for j in range(k+1,n):

                    # calcular factor m
                    L[j,k] = A[j,k]/A[k,k];

                    # operacion elemental fila
                    A[j,k:] = A[j,k:] - L[j,k]*A[k,k:]

            return (P,L,A)
```

```
In [31]: A = np.asarray([[1,2,3],[4,5,6],[7,8,1.0]])
         (P,L,U)=myLU(A)
         print(P)
         print(L)
         print(U)
         print(P @ L @ U)
```

```
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
[[1.         0.         0.        ]
 [0.14285714 1.         0.        ]
 [0.57142857 0.5        1.        ]]
[[7.00000000e+00 8.00000000e+00 1.00000000e+00]
 [0.00000000e+00 8.57142857e-01 2.85714286e+00]
 [0.00000000e+00 5.55111512e-17 4.00000000e+00]]
[[1. 2. 3.]
 [4. 5. 6.]
```

```
[7. 8. 1.]]
```

Notamos que `myLU` de arriba solicita memoria para las matrices L y P, el factor U se guarda en la memoria de A (es decir, A se sobreescribe por U). Además, se producen cambios de filas y/o columnas *en la memoria*. Implementaciones eficientes de la factorización LU con pivote parcial no intercambian físicamente las filas y columnas, pero trabajan con un *vector de contabilidad* para guardar los cambios que se producen. Este vector también representa la matriz P. Además, se puede ocupar la memoria de A para guardar los factores L y U. (La diagonal de L es contiene solo 1 y no es necesario guardarla). Es decir, una implementación eficiente de LU necesita solamente memoria para *un vector adicional*.

4.3.4 Resolver sistemas lineales con factorización LU

Digamos que nuestro objetivo es

$$\text{hallar } x \in \mathbb{R}^n \text{ tal que } Ax = b. \tag{4.5}$$

Si tenemos una factorización LU como $PA = LU$, entonces podemos proceder de la siguiente manera:

(i) usando sustitución descendente, calcular $y \in \mathbb{R}^n$ tal que

$$Ly = Pb.$$

(ii) usando sustitución ascendente, calcular $x \in \mathbb{R}^n$ tal que

$$Ux = y.$$

Obviamente, si x es la solución calculada en (a)–(b), entonces

$$PAx = LUx = Ly = Pb,$$

lo que implica $Ax = b$ y asi x también es solución del sistema original.

Notamos que la solución de (4.5) con eliminación de Gauss necesita asintóticamente n^3 flops, mientras la solución de los sistemas en (i) y (ii) necesita asintóticamente n^2 flops. Digamos que ahora nuestro objetivo es resolver n sistemas

$$Ax_j = b_j, \quad j = 1, \ldots, n, \tag{4.6}$$

con la misma matriz pero diferentes lados derechos. Si resolvemos cada sistema (4.6) con eliminación de Gauss, necesitamos asintóticamente $n \cdot n^3 = n^4$ flops. Por el otro lado, si calculamos primero una factorización LU como $PA = LU$ y resolvemos despues cada sistema (4.6) con el procedimiento (i)-(ii) de arriba, entonces necesitamos $n^3 + n \cdot n^2 = 2n^3$ flops. Notamos que la condición de L y/o U puede ser *mas grande* que la condición de A. En este caso, resolver un sistema lineal con el procedimiento de arriba resulta en un *algoritmo inestable*. Este efecto es peor para factorización LU sin pivote.

4.4 La factorización de Cholesky

Si la matriz A bajo consideración tiene cierta estructura o propiedad, entonces el costo de almacenamiento y el costo operacional para resolver un sistema pueden optimizarse.

Definición 34. Una matriz $A \in \mathbb{R}^{n\times n}$ se llama

(1) **simétrica**, si $a_{jk} = a_{kj}$ para todo j, k. En otras palabras la matriz y su transpuesta son iguales, $A = A^\top$.

(2) **definida positiva**, si $x^\top \cdot A \cdot x > 0$ para todo $0 \neq x \in \mathbb{R}^n$.

□

Si una matriz A es simétrica, entonces será deseable obtener una descomposición LU con la simetría $U = L^\top$, es decir,

$$A = L \cdot L^\top$$

con L triangular inferior. Para desarollar un algoritmo, simplemente igualamos A y $L \cdot L^\top$ y calculamos los elementos de L en el orden correcto. En el caso de matrices en $\mathbb{R}^{3\times 3}$, obtenemos

$$\begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{pmatrix} \cdot \begin{pmatrix} \ell_{11} & \ell_{21} & \ell_{31} \\ 0 & \ell_{22} & \ell_{32} \\ 0 & 0 & \ell_{33} \end{pmatrix}.$$

Entonces, recorremos la parte triangular inferior de A y obtenemos

1. columna 1
   (a) fila 1: $\ell_{11} = \sqrt{a_{11}}$,
   (b) fila 2: $\ell_{21} = a_{21}/\ell_{11}$,
   (c) fila 3: $\ell_{31} = a_{31}/\ell_{11}$,
2. columna 2
   (a) fila 2: $\ell_{22} = \sqrt{a_{22} - \ell_{21}^2}$,
   (b) fila 3: $\ell_{32} = (a_{32} - \ell_{31}\ell_{21})/\ell_{22}$,
3. columna 3
   (a) fila 3: $\ell_{33} = \sqrt{a_{33} - \ell_{31}^2 - \ell_{32}^2}$.

Las operaciones críticas que se producen durante el algoritmo (raizes, divisiones) son bien definidas si la matriz A, aparte de ser simétrica, es definida positiva.

Teorema 35 (Cholesky). *La matriz $A \in \mathbb{R}^{n\times n}$ es simétrica y definida positiva si y solo si existe $L \in \mathbb{R}^{n\times n}$ triangular inferior e invertible, tal que*

$$A = L \cdot L^\top.$$

Adicionalmente, la matriz L es única bajo la condición $\ell_{jj} > 0$. El algoritmo de Cholesky calcula la matriz L en asintóticamente n^3 flops.

En `NumPy` hay una implementación de la factorización de Cholesky, `np.linalg.cholesky`. Para verificarla, primero fabricamos una matriz simétrica y definida positiva. Si $B \in \mathbb{R}^{n\times n}$ es una matriz, entonces $A = B^\top \cdot B$ es simétrica. Si además B es invertible, entonces A es definida positiva, pues

$$0 = x^\top Bx = x^\top A^\top Ax = (Ax)^\top Ax = \|Ax\|_2$$

implica $Ax = 0$, es decir $x = 0$ por A invertible. Una notación corta para multipliación de matrices en Python es el operador `@`, y `B.T` es la matriz traspuesta.

```
In [43]: B = np.random.rand(5,5)
         A = B.T @ B # una matriz simétrica y definida positiva (si A es invertible)
         L = np.linalg.cholesky(A)
         np.allclose(L @ L.T,A,0,1e-12)

Out[43]: True
```

4.5 Matrices ralas y métodos iterativos

En la siguiente tabla recordamos costo de memoria y computacional para matrices generales:

operación	costo asintótico
almanecenamiento de $A \in \mathbb{R}^{n\times n}$	n^2
$A \cdot x$ para $A \in \mathbb{R}^{n\times n}, x \in \mathbb{R}^n$	n^2
$A + B$ para $A, B \in \mathbb{R}^{n\times n}$	n^2
$A \cdot B$ para $A, B \in \mathbb{R}^{n\times n}$	n^3
resolver $Ax = b$, $A \in \mathbb{R}^{n\times n}$	n^3

En la práctica aparecen matrices que son muy grandes, pero contienen muchos zeros. Estas matrices se llaman *ralas*[3]. Aunque no existe una definición rigurosa, vamos a llamar una matrix $A \in \mathbb{R}^{n\times n}$ *rala* si el número N de elementos no zeros de A es proporcional al tamaño n. El almacenamiento de matrices ralas y operaciones con ellas pueden optimizarse, omitiendo los elementos zeros.

4.5.1 Formato de coordenadas

Para almacenar una matriz rala es suficiente almacenar los elementos $a_{jk} \neq 0$ y las coordenadas j, k. Es decir, necesitamos tres vectores $\mathbf{j}, \mathbf{k}, \mathbf{a} \in \mathbb{R}^N$, donde N es el número de elementos no zeros de A. Para cada elemento $a_{jk} \neq 0$, existe único inidice ℓ tal que $\mathbf{j}_\ell = j$, $\mathbf{k}_\ell = k$, y $\mathbf{a}_\ell = a_{jk}$. Por ejemplo, la matriz

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 4 & -2 & 0 & 0 \\ 0 & -2 & 4 & -3 & 0 \\ 0 & 0 & -3 & 6 & -2 \\ 0 & 0 & 0 & -2 & 4 \end{pmatrix}$$

la podemos representar en este formato como

$$\begin{aligned} \mathbf{j} &= \big(1,1,2,2,2,3,3,3,4,4,4,5,5\big) \\ \mathbf{k} &= \big(1,2,1,2,3,2,3,4,3,4,5,4,5\big) \\ \mathbf{a} &= \big(2,-1,-1,4,-2,-2,4,-3,-3,6,-2,-2,4\big) \end{aligned}$$

Mencionamos que existen otros formatos, el más eficiente para muchas operaciones es el formato CSR (compressed sparse row).

Si queremos implementar el producto de una matriz $A \in \mathbb{R}^{m\times n}$ en formato de coordenadas con un vector x, entonces es suficiente recorrer un loop sobre $\ell = 1, \ldots, N$. Cuidado: en nuestro formato los índices empiezan con 1.

[3] *sparse* en inglés

```
In [ ]: def matVecCOO(m,j,k,a,x):
            # n es el número de filas de la matriz A

            N = j.shape[0]
            y = np.zeros(m)

            for ll in range(0,N):
                y[j[ll]-1] = y[j[ll]-1] + a[ll]*x[k[ll]-1]

            return y
```

Notamos que en esta implementación no será necesario que un par de índices j, k aparezca solo una vez en el formato. Por ejemplo, si hay dos índicies $\ell_1 \neq \ell_2$ con $j_{\ell_1} = j_{\ell_2} = j$ y $k_{\ell_1} = k_{\ell_2} = k$, entonces el código trabaja con la matriz con $a_{j,k} = \mathbf{a}_{\ell_1} + \mathbf{a}_{\ell_2}$.

4.5.2 El problema del *fill in*

En lo siguiente, vamos a mostrar dos matrices y sus factores de Cholesky.

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 & 0 & 0 & 2 \\ 0 & 0 & 1 & 3 & 0 & 0 & 3 \\ 1 & 2 & 3 & 39 & 0 & 0 & 104 \\ 0 & 0 & 0 & 0 & 1 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 1 & 6 \\ 1 & 2 & 3 & 104 & 5 & 6 & 8863 \end{pmatrix}, \quad L_A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 2 & 3 & 18 & 5 & 6 & 92 \end{pmatrix}$$

$$B = \begin{pmatrix} 8863 & 6 & 5 & 104 & 3 & 2 & 1 \\ 6 & 1 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 0 & 0 & 0 \\ 104 & 0 & 0 & 39 & 3 & 2 & 1 \\ 3 & 0 & 0 & 3 & 1 & 0 & 0 \\ 2 & 0 & 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}, \quad L_B = \begin{pmatrix} 94.1435 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0637 & 0.9979 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0531 & -0.0033 & 0.9985 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 1.1047 & -0.0705 & -0.0589 & 6.1458 & 0.0000 & 0.0000 & 0.0000 \\ 0.0318 & -0.0020 & -0.0017 & 0.4823 & 0.8753 & 0.0000 & 0.0000 \\ 0.0212 & -0.0013 & -0.0011 & 0.3215 & -0.1779 & 0.9297 & 0.0000 \\ 0.0106 & -0.0006 & -0.0005 & 0.1607 & -0.0889 & -0.0728 & 0.9802 \end{pmatrix}$$

Observamos que en el caso del la matriz A, el factor de Cholesky L_A reproduce la estructura rala de A. Sin embargo, en el caso de la matriz B ocurre un *fill in*, es decir, aunque B es rala, se llena todo el perfil con elementos no zeros en el factor de Cholesky. En el caso extremo, podemos almacenar una matriz A rala, pero no su factor de Cholesky. El efecto se produce en muchos algoritmos. En el caso de resolver sistemas lineales $Ax = b$, se prefiere entonces *métodos iterativos*, que solo requieren la múltiplicación con la matriz A.

4.5.3 Métodos iterativos

Métodos iterativos calculan una sucesión $(x_k)_{k\in\mathbb{N}}$, $x_k \in \mathbb{R}^n$, que converge a la solución exacta x del sistema $Ax = b$.

Definición 36 (Método iterativo simple)**.** Sea $A \in \mathbb{R}^{n\times n}$ invertible y $b \in \mathbb{R}^n$. Sean $M, N \in \mathbb{R}^{n\times n}$ tal que $A = M - N$. Para un vector inicial $x_0 \in \mathbb{R}^n$ se define la iteración

$$Mx_{k+1} = Nx_k + b, \quad \text{para todo } k \geq 0.$$

□

Para calcular la sucesión $(x_k)_{k\in\mathbb{N}}$, tenemos que resolver entonces en cada paso un sistema de la forma $Mx_{k+1} = \widetilde{b}$. Es decir, la solución de este último sistema tiene que ser muy barato en comparación con el sistema $Ax = b$ para que el método iterativo tenga sentido.

Definición 37. El método iterativo de la Definición 36 se llama convergente si para cada punto inicial $x_0 \in \mathbb{R}^n$ la sucesión $(x_k)_{k\in\mathbb{N}}$ converge. □

En la última definición no pedimos convergencia de $(x_k)_{k\in\mathbb{N}}$ a la solución exacta x. Eso no es necesario, pues, si la sucesión $(x_k)_{k\in\mathbb{N}}$ converge a un vector x, entonces $Mx = Nx + b$, es decir, $Ax = b$. Concluimos que el límite de $(x_k)_{k\in\mathbb{N}}$, si existe, es solución de nuestro problema.

Teorema 38. *El método iterativo de la Definición 36 converge si y solo si*

$$\rho(M^{-1}N) < 1.$$

La matriz $M^{-1}N$ se llama matriz de iteración del método.

Proof. Por el Lema 23 existe una norma matricial $\|\cdot\|$ inducida por una norma vectorial que también vamos a anotar por $\|\cdot\|$, tal que $\|M^{-1}N\| < 1$. Sea x la solución de $Ax = b$, entonces $x = M^{-1}(Nx + b)$. Por lo tanto

$$x_k - x = M^{-1}Nx_{k-1} + M^{-1}b - M^{-1}(Nx + b) = M^{-1}N(x_{k-1} - x).$$

Concluimos que $x_k - x = (M^{-1}N)(x_0 - x)$, y por el Lema 20,

$$\|x_k - x\| \leq \|M^{-1}N\|^k \|x_0 - x\| \tag{4.7}$$

Dado que $\|M^{-1}N\| < 1$, concluimos que $\|x_k - x\| \to 0$. Dado que todas las normas sobre un espacio vectorial de dimension finita son equivalentes, concluimos el resultado. □

En lo que sigue vamos a descomponer la matriz $A \in \mathbb{R}^{n\times n}$ en los tres componentes

$$D = \begin{pmatrix} a_{11} & & \cdots & 0 \\ & \ddots & & \\ & & \ddots & \\ 0 & \cdots & & a_{nn} \end{pmatrix}, L = \begin{pmatrix} 0 & & \cdots & 0 \\ a_{21} & \ddots & & \\ \vdots & \ddots & \ddots & \\ a_{n1} & \cdots & a_{n,n-1} & 0 \end{pmatrix}, U = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ & \ddots & \ddots & \vdots \\ & & \ddots & a_{n-1,n} \\ 0 & \cdots & & 0 \end{pmatrix},$$

es decir $A = D + L + U$. Dos métodos iterativos básicos son

1. **Método de Jacobi:** En este método, $M = D$ y $N = -(L+U)$, es decir

$$Dx_{k+1} = b - (L+U)x_k.$$

La matriz de iteración es $-D^{-1}(L+U)$. El último sistema se resuleve con costo computacional n, pues, D es una matriz diagonal. De hecho,

$$x_{k+1,\ell} = \frac{1}{a_{\ell\ell}}\Big(b_\ell - \sum_{\substack{j=1\\ j\neq\ell}}^{n} a_{\ell j}x_{k,j}\Big).$$

2. **Método de Gauss-Seidel:** En este método, $M = (D+L)$, $N = -U$, es decir

$$(D+L)x_{k+1} = b - Ux_k.$$

La matriz de iteración es $-(D+L)^{-1}U$. En el último sistema se puede resuelve con costo computacional n^2, pues, la matriz $D+L$ es una matriz triangular inferior. De hecho, si reducimos la sustitución descendente, obtenemos

$$x_{k+1,\ell} = \frac{1}{a_{\ell\ell}}\Big(b_\ell - \sum_{j=1}^{\ell-1} a_{\ell j}x_{k+1,j} - \sum_{j=\ell+1}^{n} a_{\ell j}x_{k,j}\Big). \tag{4.8}$$

Ejemplo 39. *Sea*

$$A = \begin{pmatrix} 4 & 1 \\ -1 & 2 \end{pmatrix}.$$

Verifique que el método de Gauss-Seidel para A converge.

Calculamos la matriz de iteración

$$-(D+L)^{-1}U = \begin{pmatrix} 4 & 0 \\ -1 & 2 \end{pmatrix}^{-1}\begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix} = \frac{1}{8}\begin{pmatrix} 2 & 0 \\ 1 & 4 \end{pmatrix}\begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -\frac{1}{4} \\ 0 & -\frac{1}{8} \end{pmatrix}.$$

Es decir, los valores propios de la matriz de iteración son $\lambda_1 = 0$, $\lambda_2 = \frac{1}{8}$, y por lo tanto su radio espectral es menor que uno. Por el teorema 38, el método converge. □

Para matrizes generales podemos implementar el m'etodo de Gauss-Seidel de la siguiente manera: Recordamos que `np.tril(A,r)` calcula la parte triangular de una matriz a partir de la *diagonal* r (es decir, corresponde a elementos $a_{j,j+r}$). Usamos sustitución descendente para resolver el sistema lineal.

```
In [47]: def gs(A,b,x,N):
             # x es el vector inicial
             # N es el número de pasos
             n = A.shape[0]

             DpL = np.tril(A)
             U = np.tril(A.T,-1).T

             for ll in range(0,N):
                 x = sustitucionDescendente(DpL,b-U.dot(x))
                 print(x)

             return x
```

En la implementación arriba usamos un número de pasos fijado por el usuario. En la práctica, se usa un *criterio de cancelación* para determinar el número de pasos a realizar hasta llegar a cierta tolerancia.

```
In [49]: # Ax=b
         A = np.asarray([[4,1],[-1,2]])
         x = np.asarray([2,-3])
         b = A.dot(x)

         # x0 es el vector zero
         x0 = np.zeros(2)

         # Gauss-Seidel con 10 pasos
         z = gs(A,b,x0,10)

[ 1.25  -3.375]
[ 2.09375  -2.953125]
[ 1.98828125 -3.00585938]
[ 2.00146484 -2.99926758]
[ 1.99981689 -3.00009155]
[ 2.00002289 -2.99998856]
[ 1.99999714 -3.00000143]
[ 2.00000036 -2.99999982]
[ 1.99999996 -3.00000002]
[ 2.00000001 -3.        ]
```

Definición 40. Una matriz $A \in \mathbb{R}^{n\times n}$ se llama estrictamente diagonal dominante, si

$$\sum_{\substack{k=1\\k\neq j}}^{n} |a_{jk}| < |a_{jj}|, \quad \text{para todo } j = 1, \ldots, n.$$

□

Lema 41. *Sea $A \in \mathbb{R}^{n\times n}$ estrictamente diagonal dominante. Entonces, A es regular y los métodos de Jacobi y Gauss-Seidel convergen.*

Proof. Si A no es regular, entonces existe un vector $x \neq 0$ tal que $Ax = 0$. Será k tal que $|x_j| = \|x\|_\infty$. Entonces,

$$|a_{jj}||x_j| = |\sum_{\substack{k=1\\k\neq j}}^{n} a_{jk}x_k| \leq |x_j| \sum_{\substack{k=1\\k\neq j}}^{n} |a_{jk}|,$$

es decir, A no es estrictamente diagonal dominante. Vamos a demostrar solamente que el método de Jacobi converge. Sea entonces λ un valor propio de la matriz de iteración $-D^{-1}(L+U)$, es decir, existe un vector $v \neq 0$ con $\|v\|_\infty = 1$ tal que

$$\lambda v = -D^{-1}(L+U)v.$$

Obtenemos

$$|\lambda| = \|\lambda v\|_\infty = \| - D^{-1}(L+U)v\|_\infty \leq \|D^{-1}(L+U)\|_\infty = \max_{j=1,\ldots,n} \frac{1}{|a_{jj}|} \sum_{\substack{k=1\\k\neq j}}^{n} |a_{jk}| < 1.$$

□

A partir de (4.8) podemos escribir

$$x_{k+1,\ell} = \frac{1}{a_{\ell\ell}}\Big(b_\ell - \sum_{j=1}^{\ell-1} a_{\ell j}x_{k+1,j} - \sum_{j=\ell+1}^{n} a_{\ell j}x_{k,j}\Big) = x_{k,\ell} + \frac{1}{a_{\ell\ell}}\Big(b_\ell - \sum_{j=1}^{\ell-1} a_{\ell j}x_{k+1,j} - \sum_{j=\ell}^{n} a_{\ell j}x_{k,j}\Big),$$

y introduciremos un factor de amortiguamiento ω

$$x_{k+1,\ell} = x_{k,\ell} + \frac{\omega}{a_{\ell\ell}}\Big(b_\ell - \sum_{j=1}^{\ell-1} a_{\ell j}x_{k+1,j} - \sum_{j=\ell}^{n} a_{\ell j}x_{k,j}\Big).$$

Eso corresponde a

$$\Big(\frac{1}{\omega}D + L\Big)x_{k+1} = b + \Big(\frac{1}{\omega}D - (D+U)\Big)x_k.$$

Este método se llama SOR (succesive over relaxation), y su matriz de iteración es

$$J_\omega := \Big(D + \omega L\Big)^{-1}\Big((1-\omega)D - \omega U\Big).$$

Para $\omega = 1$, se recupera el método de Gauss-Seidel.

Lema 42. *Sea $A \in \mathbb{R}^{n\times n}$ simétrica y definida positiva. Entonces*

$$\rho(J_\omega) < 1 \quad \textit{si y solo si} \quad 0 < \omega < 2.$$

□

Chapter 5

Interpolación

En un **problema de interpolación** tenemos pares de números reales $(x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$ (los datos), y el objetivo es buscar una función g de manera que

$$g(x_j) = y_j, \quad j = 0, \ldots, n. \tag{5.1}$$

Una función que satisface (5.1) se llama *función interpolante*. Obviamente, una condición razonable para garantizar la existencia de una función interpolante es $x_j \neq x_k$ siempre que $j \neq k$. Hay diversos motivos para considerar problemas de interpolación, por ejemplo

- evaluar una función con valores dados por tabla,
- hacer una gráfica de una función suave con valores dados por tabla,
- derivar o integrar una función con valores dados por tabla,
- reemplazar una función *complicada* por una función *facíl*.

Naturalmente hay muchas funciones que satisfacen (5.1). A veces, los datos tienen ciertas propiedades como monotonicidad, periodicidad, o convexidad, y queremos una función interpolante que refleja estas propiedades. A veces estas propiedades se dan si los datos son mediciones te cantidades físicas. También hay que tomar en cuenta el objetivo de la interpolación. Si queremos derivar la función interpolante, tenemos que buscar una función derivable. Las familias de funciones mas usadas para interpolación son

- polinomios,
- polinomios a trozos (*splines* en inglés),
- polinomios trigonometricos,
- funciones exponenciales,

- funciones racionales.

En lo siguiente, vamos a trabajar solamente con funciónes polinomiales, es decir, vamos a considerar *interpolación polinomial*. Es facíl derivar e integrar polinomios, y es facíl caracterizarlos.

5.1 Interpolación polinomial de Lagrange

Recordamos que la formula general de un polinomio de grado menor o igual a $n \in \mathbb{N}$ es

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{k=0}^{n} a_k x^k. \tag{5.2}$$

El espacio vectorial de todos los polinomios de grado menor o igual a $n \in \mathbb{N}$ lo anotamos por $\mathbb{P}_n$,

$$\mathbb{P}_n := \left\{ p : \mathbb{R} \to \mathbb{R} \mid p(x) = \sum_{k=0}^{n} a_k x^k \right\}.$$

Un polinomio $p \in \mathbb{P}_n$ se caracteriza de manera única por sus $n+1$ coeficientes a_k, es decir, la dimension de $\mathbb{P}_n$ es $\dim \mathbb{P}_n = n+1$.

Ejemplo 43. *Sea $p \in \mathbb{P}_3$ dado por $p(x) = 2x^3 - x^2 + 3x - 1$. Calcule $p(1)$, $p(3)$, $p'(2)$, y $\int_0^2 p(x)\,dx$.* □

Nuestro primer objetivo es establecer existencia y unicidad de un polinomio interpolante. Por ejemplo, para un dato (x_0, y_0) existe único polinomio de grado 0 (una constante), que pasa por el punto (x_0, y_0). Para dos datos (x_0, y_0) y (x_1, y_1) existe único polinomio de grado 1 (una recta), que pasa por los dos puntos (x_0, y_0) y (x_1, y_1). Para tres datos existe único polinomio de grado 2 (una parabola) que pasa por los tres puntos, etc.

Teorema 44 (Existencia y unicidad del polinomio interpolante)**.** *Sean $(x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$ $n+1$ pares de números, y sea $x_j \neq x_k$ para $j \neq k$. Entonces existe único polinomio interpolante $p \in \mathbb{P}_n$, es decir,*

$$p(x_j) = y_j, \quad j = 0, \ldots, n. \tag{5.3}$$

□

Aunque el polinomio interpolante en el último Teorema 44 es único, existe una variedad de formas de representarlo. La representación (5.2) es una forma, pero existen otras. Si $\{\ell_0, \ldots, \ell_n\}$ es una base de $\mathbb{P}_n$, entonces un polinomio $p \in \mathbb{P}_n$ se representa como una combinación lineal

$$p(x) = \sum_{k=0}^{n} \lambda_k \ell_k(x), \tag{5.4}$$

con coeficientes $\lambda_0, \lambda_1, \ldots, \lambda_n \in \mathbb{R}$ que dependen de la base. Vamos a conocer varias bases y sus ventajas y desventajas al calcular y manipular polinomios.

Lema 45. *Sea $\mathbb{P}_n$ el espacio vectorial de los polinomios de grado menor o igual a $n \in \mathbb{N}$.*

(i) ***Base monomial:*** *Si definimos los monomios $m_k(x) = x^k$, entonces $\{m_0, \ldots, m_n\}$ es una base de $\mathbb{P}_n$, se llama base monomial.*

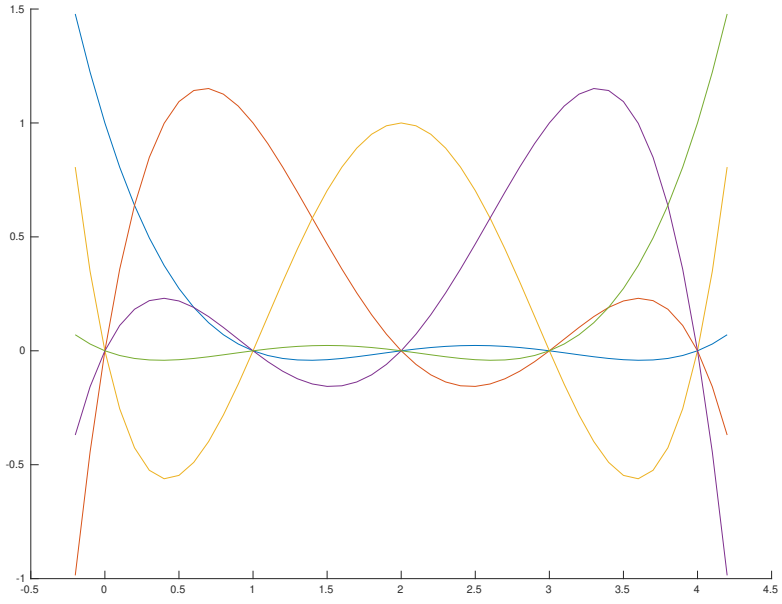
(ii) ***Base de Lagrange:*** *Sean $x_0, \ldots, x_n$ puntos con $x_j \neq x_k$ para $j \neq k$. Se define el polinomio de Lagrange como*

$$L_j(x) = \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}.$$

Entonces, $\{L_0, \ldots, L_n\}$ es una base de $\mathbb{P}_n$, se llama base de Lagrange. Los polinomios de Lagrange tienen la propiedad importante

$$L_j(x_k) = \begin{cases} 1 & j = k \\ 0 & j \neq k \end{cases}. \tag{5.5}$$

□



Los polynomios de Lagrange asociados a los puntos $x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, $x_4 = 4$.

Por ejemplo, un polinomio $p \in \mathbb{P}^n$ representado en la base monomial

$$p(x) = \sum_{k=0}^{n} \lambda_k m_k(x)$$

se puede representar en Python como un array $\mathbf{p} \in \mathbb{R}^{n+1}$. Vamos a ordenar los coeficientes de grado *mayor* a *menor* $\mathbf{p}_k = \lambda_{n-k}$, es decir

$$p(x) = \sum_{k=0}^{n} \mathbf{p}_{n-k} m_k(x)$$

En este formato, es fácil caclular p' y $\int p(x)\,dx$, pues $m_k'(x) = k m_{k-1}(x)$, y

$$\sum_{k=0}^{n-1} \mathbf{p}'_{n-1-k} m_k(x) = p'(x) = \sum_{k=1}^{n} k \mathbf{p}_{n-k} m_{k-1}(x) = \sum_{k=0}^{n-1} (k+1) \mathbf{p}_{n-k-1} m_k(x)$$

Por otro lado, $\int m_k(x)\,dx = \frac{1}{k+1} m_{k+1}(x)$, y

$$\sum_{k=0}^{n+1} P_{n+1-k} m_k(x) = \int p(x)\,dx = \sum_{k=0}^{n} \frac{1}{k+1} \mathbf{p}_{n-k} m_{k+1}(x) = \sum_{k=1}^{n+1} \frac{1}{k} \mathbf{p}_{n-k+1} m_k(x).$$

En Python, eso se lee asi:

```
In [2]: def dpdx(p):
            # derivar polinomio p en base monomial

            n = p.shape[0]-1 # grado de p
            pp = np.zeros(n) # la derivada de p

            for k in range(0,n):
                pp[n-1-k] = (k+1)*p[n-k-1]

            return pp

        def intpdx(p):
            # calcular primitiva de polinomio p en base monomial

            n = p.shape[0]-1 # grado de p
            intp = np.zeros(n+2)

            for k in range(1,n+2):
                intp[n+1-k] = p[n-k+1]/k
```

```
            return intp

In [3]: p = np.asarray([4, -1, 3, 2])
        print(p)
        pp = dpdx(p)
        print(pp)
        intp = intpdx(p)
        print(intp)

[ 4 -1  3  2]
[12. -2.  3.]
[ 1.         -0.33333333  1.5         2.          0.        ]
```

En NumPy hay una clase *poly1d* para operar con polinomios. Esta clase usa la representación de un polinomio de arriba en terminos de los coeficientes en la base monomial de grado mayor a menor.

```
In [14]: p = np.poly1d([4, -1, 3, 2])
         print(p)

         # los coeficientes son un array
         print(p.c)

         # la clase permite evaluar el polinomio
         print(p(1))

         # y operaciones vectoriales
         print(p + 3*p)

         # incluso el producto de polinomios
         print(p*p)

   3     2
4 x - 1 x + 3 x + 2
[ 4 -1  3  2]
8
    3     2
16 x - 4 x + 12 x + 8
    6     5      4      3     2
16 x - 8 x + 25 x + 10 x + 5 x + 12 x + 4
```

5.1.1 Calcular el polinomio interpolante

Supongamos que tenemos $n+1$ pares de números $(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)$ y queremos determinar el polinomio interpolante, es decir, el polinomio $p \in \mathbb{P}_n$ que satisface (5.3). Este objetivo nos lleva a un sistema lineal a resolver.

Ejemplo 46. *Calcule el polinomio interpolante de grado* 2 *que interpola los puntos* $(0,1), (2,-1), (-1,7)$.

Solución: En la notación del Teorema 44 tenemos $n = 2$, es decir, buscamos un polinomio de grado 2. Lo representamos en la base monomial

$$p(x) = a + bx + cx^2.$$

El polinomio p tiene que interpolar los tres puntos dados, es decir $p(0) = 1$, $p(2) = -1$, y $p(-1) = 7$. Eso nos lleva a un sistema lineal de 3 ecuaciones en las 3 desconocidas a, b, c

$$\begin{aligned} a &= 1, \\ a + 2b + 4c &= -1, \\ a - b + c &= 7. \end{aligned}$$

La solución del último sistema es

$$a = 1; \quad b = -\frac{13}{3}, \quad c = \frac{5}{3},$$

es decir, el polinomio que estamos buscando está dado por

$$p(x) = 1 - \frac{13}{3}x + \frac{5}{3}x^2.$$

□

En el último ejemplo usamos la base monomial y calculamos los coeficientes correspondientes. Si usamos la base $\{\ell_0, \dots, \ell_n\}$, entonces vamos a resolver un sistema lineal para calcular los coeficientes $\lambda_0, \dots, \lambda_n$ de la combinación lineal (5.4). La matriz del sistema lineal que hay que resolver (y por lo tanto el costo computacional) depende de la base $\{\ell_0, \dots, \ell_n\}$: Las $n+1$ ecuaciones (5.3) se transforman en un sistema lineal

$$\underbrace{\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}}_{y} = \begin{pmatrix} \sum_{k=0}^n \lambda_k \ell_k(x_0) \\ \sum_{k=0}^n \lambda_k \ell_k(x_1) \\ \vdots \\ \sum_{k=0}^n \lambda_k \ell_k(x_n) \end{pmatrix} = \underbrace{\begin{pmatrix} \ell_0(x_0) & \ell_1(x_0) & \dots & \ell_n(x_0) \\ \ell_0(x_1) & \ell_1(x_1) & \dots & \ell_n(x_1) \\ \vdots & & & \\ \ell_0(x_n) & \ell_1(x_n) & \dots & \ell_n(x_n) \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_n \end{pmatrix}}_{\lambda}$$

que tenemos que resolver. La matriz $A \in \mathbb{R}^{(n+1)\times(n+1)}$ del sistema depende de la base $\{\ell_0, \dots, \ell_n\}$, y la solución λ es el vector que contiene justamente los coeficientes $\lambda_0, \dots, \lambda_n$ de la representación (5.4). Con respecto a las dos bases que ya conocemos podemos observar lo siguiente.

(i) **Base monomial:** Si usamos la base monomial $\{m_0, \ldots, m_n\}$, entonces la representación (5.4) se transforma en la representación conocida

$$p(x) = \lambda_0 + \lambda_1 x + \lambda_2 x^2 + \cdots + \lambda_n x^n.$$

En este caso, la matriz A se llama *matriz de Vandermonde* y tiene la forma

$$A = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix}$$

Observamos que la matriz A es una matriz llena, y por lo tanto el costo computacional para calcular los coeficientes $\lambda_0, \ldots, \lambda_n$ en la base monomial es asintóticamente n^3.

(ii) **Base de Lagrange:** Si usamos la base de Lagrange $\{L_0, \ldots, L_n\}$, observamos primero que la propiedad (5.5) implica

$$A = \begin{pmatrix} L_0(x_0) & L_1(x_0) & \ldots & L_n(x_0) \\ L_0(x_1) & L_1(x_1) & \ldots & L_n(x_1) \\ \vdots & & \ddots & \vdots \\ L_0(x_n) & L_1(x_n) & \ldots & L_n(x_n) \end{pmatrix} = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix},$$

es decir, la matriz A es la identidad. Por lo tanto, el sistema $A\lambda = y$ se reduce a $\lambda = y$, es decir $\lambda_k = y_k$. El costo computacional para calcular los coeficientes $\lambda_0, \ldots, \lambda_n$ en la base de Lagrange es zero, y el polinomio interpolante tiene la representación

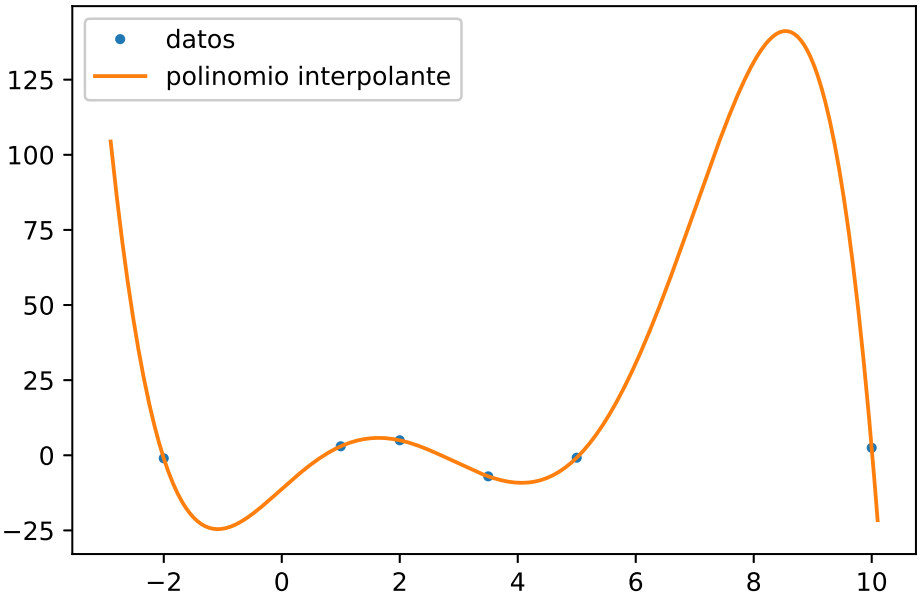
$$p(x) = \sum_{k=0}^{n} y_k L_k(x).$$

En NumPy existe una función *polyfit* para encontra un polinomio de cierto grado con mejor ajuste a los datos. En el caso de que el número m de datos y el grado n del polinomio satisfacen $m = n + 1$, entonces *polyfit* calcula el polinomio interpolante.

```
In [17]: x = np.array([-2.0, 1.0, 2.0, 3.5, 5.0, 10.0, ])
         y = np.array([-1.0, 3.0, 5.0, -7.0, -0.8, 2.5])

         # para 6 datos, polinomio interpolante tiene grado 5
         p = np.polyfit(x, y, 5)

         # p es un array de coeficientes
         print(p)

         # transformarlo en un polinomio
```

```
          p = np.poly1d(p)

          print(p(-2.0))
          print(p(3.5))

          # visualizar los datos y el polinomio
          import matplotlib.pyplot as plt

          xp = np.linspace(-2.9, 10.1, 1000)
          plt.plot(x,y,'.',xp,p(xp))
          plt.legend(["datos","polinomio interpolante"])
          plt.savefig("pinterp.eps")
```

```
[ -0.0568815    0.93590093  -3.78883885  -0.43959164  17.56545936
 -11.2160483 ]
-0.9999999999977032
-7.00000000000092
```



5.1.2 Evaluar el polinomio interpolante

Evaluar el polinomio interpolante en un punto

Supongamos que tenemos $n+1$ pares de números $(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)$, y nuestro objetivo es evaluar el polinomio interpolante p en un solo punto $\overline{x}$, es decir, calcular $p(\overline{x})$. En este caso, no es necesario determinar los coeficientes de p.

Lema 47. *Sean $(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)$ $n+1$ pares de números. Anotamos por $p_{\ell,k} \in \mathbb{P}_{k-\ell}$, $0 \le \ell \le k \le n$ el único polinomio que interpola los puntos (x_j, y_j), $j = \ell, \dots, k$. Entonces*

$$p_{\ell,k}(x) = \frac{x - x_\ell}{x_k - x_\ell} p_{\ell+1,k}(x) + \frac{x_k - x}{x_k - x_\ell} p_{\ell,k-1}(x).$$

□

Si p es el polinomio que interpola los puntos $(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)$, entonces en la notación del último lemma $p = p_{0,n}$, y si si queremos evaluar $p_{0,n}(\overline{x})$, será suficiente evaluar $p_{1,n}(\overline{x})$ y $p_{0,n-1}(\overline{x})$. El mismo lema dice que para evaluar $p_{1,n}(\overline{x})$ será suficiente evaluar $p_{2,n}(\overline{x})$ y $p_{1,n-1}(\overline{x})$, y para evaluar $p_{0,n-1}(\overline{x})$ será suficiente evaluar $p_{1,n-1}(\overline{x})$ y $p_{0,n-2}(\overline{x})$, etc. Con eso podemos armar una tabla que se llama *esquema de Neville*:

$$\begin{array}{lllllllllll}
x_0 & y_0 = p_{0,0}(\overline{x}) & & & & & & & & & \\
 & & \nwarrow & & & & & & & & \\
x_1 & y_1 = p_{1,1}(\overline{x}) & \leftarrow & p_{0,1}(\overline{x}) & & & & & & & \\
 & & \nwarrow & & & & & & & & \\
x_2 & y_2 = p_{2,2}(\overline{x}) & \leftarrow & p_{1,2}(\overline{x}) & & & & & & & \\
 & & \vdots & \vdots & \ddots & & & & & & \\
x_{n-2} & y_{n-2} = p_{n-2,n-2}(\overline{x}) & \leftarrow & p_{n-3,n-2}(\overline{x}) & \dots & p_{0,n-2}(\overline{x}) & & & & & \\
 & & \nwarrow & & \ddots & & \nwarrow & & & & \\
x_{n-1} & y_{n-1} =_{n-1,n-1} (\overline{x}) & \leftarrow & p_{n-2,n-1}(\overline{x}) & \dots & p_{1,n-1}(\overline{x}) & \leftarrow & p_{0,n-1}(\overline{x}) & & & \\
 & & \nwarrow & & \ddots & & \nwarrow & & \nwarrow & & \\
x_n & y_n = p_{n,n}(\overline{x}) & \leftarrow & p_{n-1,n}(\overline{x}) & \dots & p_{2,n}(\overline{x}) & \leftarrow & p_{1,n}(\overline{x}) & \leftarrow & p_{0,n}(\overline{x})
\end{array}$$

Con el esquema de Neville podemos calcular $p_{0,n}(\overline{x})$ en asintóticamente n^2 flops.

Ejemplo 48. *Sea p el polinomio de grado 2 que interpola $(0,1), (1,3), (2,2)$. Evalua p en el punto 0.5 usando el esquema de Neville.*

Armamos el esquema de Neville:

$$\begin{array}{ll} 0 & 1 = p_{0,0}(0.5) \\ 1 & 3 = p_{1,1}(0.5) \\ 2 & 2 = p_{2,2}(0.5) \end{array}$$

Calculamos primero

$$p_{0,1}(0.5) = \frac{0.5 - x_0}{x_1 - x_0} p_{1,1}(0.5) + \frac{x_1 - 0.5}{x_1 - x_0} p_{0,0}(0.5) = \frac{0.5 - 0}{1 - 0} 3 + \frac{1 - 0.5}{1 - 0} 1 = 2$$
$$p_{1,2}(0.5) = \frac{0.5 - x_1}{x_2 - x_1} p_{2,2}(0.5) + \frac{x_2 - 0.5}{x_2 - x_1} p_{1,1}(0.5) = \frac{0.5 - 1}{2 - 1} 2 + \frac{2 - 0.5}{2 - 1} 3 = \frac{7}{2}.$$

Despues, calculamos

$$p_{0,2}(0.5) = \frac{0.5 - x_0}{x_2 - x_0} p_{1,2}(0.5) + \frac{x_2 - 0.5}{x_2 - x_0} p_{0,1}(0.5) = \frac{0.5 - 0}{2 - 0} \frac{7}{2} + \frac{2 - 0.5}{2 - 0} 2 = \frac{19}{8}.$$

□

Vemos entonces que si queremos implementar el esquema de Neville, vamos a necesitar dos loops: Un loop exterior sobre la diferencia entre k y ℓ, y un loop interior sobre k o ℓ.

```
In [103]: def neville(x,y,z):
              # evaluar polinomio interpolante en punto z,
              # usando el esquema de Neville

              n = x.shape[0]-1 # grado del polinomio interpolante

              aux = np.zeros(n+1)

              for k in range(0,n+1):
                  aux[k]=y[k]

              # el loop exterior corre sobre la diferencia de k y l
              for kml in range(1,n+1):
                  # el loop interior corre sobre k
                  for k in range(kml,n+1):
                      l = k-kml
                      aux[l] = (z-x[l])/(x[k] - x[l])*aux[l+1] + (x[k]-z)/(x[k]-x[l])*aux[l]

              return aux[0]

In [104]: x = np.array([-2.0, 1.0, 2.0, 3.5, 5.0, 10.0, ])
          y = np.array([-1.0, 3.0, 5.0, -7.0, -0.8, 2.5])

          print(neville(x,y,6))

30.579291079291078
```

Si ya determinamos el polinomio interpolante, es decir tenemos los coeficientes $\lambda_0, \ldots, \lambda_n$ de la representación (5.4) en la mano, entonces podemos usar esta representación para evaluar el polinomio.

(i) **Base monomial:** Si usamos la base monomial $p(x) = \sum_{k=0}^{n} \lambda_k x^k$, entonces podemos escribir

$$\begin{aligned} p(x) = \lambda_0 + \sum_{k=1}^{n} \lambda_k x^k &= \lambda_0 + x \sum_{k=1}^{n} \lambda_k x^{k-1} \\ &= \lambda_0 + x \left(\lambda_1 + \sum_{k=2}^{n} \lambda_k x^{k-1} \right) \\ &= \lambda_0 + x(\lambda_1 + x(\lambda_2 + x(\dots(\lambda_{n-1} + \lambda_n x)\dots))). \end{aligned}$$

Ahora podemos calcular la última expresión de adentro hacia afuera. Eso se llama *esquema de Horner*. Si usamos la representación de un polinomio p por un vector $\mathbf{p} \in \mathbb{R}^{n+1}$ de antes, es decir

$$p(x) = \sum_{k=0}^{n} \mathbf{p}_{n-k} m_k(x),$$

entonces

$$p(x) = \mathbf{p}_n + x(\mathbf{p}_{n-1} + x(\mathbf{p}_{n-2} + x(\dots(\mathbf{p}_1 + \mathbf{p}_0 x)\dots))).$$

podemos implementar el esquema de Horner de la siguiente manera.

```
In [105]: def horner(p,x):
              # evaluar polinomio en x usando esquema de Horner

              n = p.shape[0]-1

              px = p[0]
              for k in range(1,n+1):
                  px = p[k] + px*x

              return px

In [106]: p = np.asarray([4, -3, 2, -9])
          print(horner(p,2))

15
```

El esquema de Horner evalua un polinomio de grado n en asintóticamente n flops.

(ii) **Base de Lagrange:** Si usamos la base de Lagrange $p(x) = \sum_{k=0}^{n} y_k L_k(x)$, entonces evaluar $L_k(x)$ tiene un costo operacional de asintóticamente n. Dado que tenemos $n+1$ polinomios de Lagrange, el costo operacional de evaluar el polinomio p es asintóticamente n^2.

Evaluar el polinomio interpolante en muchos puntos

Consideramos el siguiente objetivo: Dado $n+1$ pares de números $(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)$, evalua el polinomio interpolante p en n puntos $\overline{x}_j$, $j = 1, \dots, n$. Con las tecnicas que vimos hasta ahora, tenemos tres posibilidades.

1. Aplicar el esquema de Neville para cada punto $\overline{x}_j$. El costo para un punto es asintóticamente n^2, es decir, el costo total será n^3 flops.

2. Determinar el polinomio interpolante en la base monomial (n^3 flops) y aplicar n veces el esquema de Horner ($n \cdot n = n^2$ flops). El costo total será n^3 flops.

3. Determinar el polinomio interpolante en la base de Lagrange (0 flops) y evaluarlo n veces ($n \cdot n^2 = n^3$ flops). El costo total será n^3 flops.

Nos podemos preguntar si existe una base que permite calcular los coeficientes en asintóticamente n^2, y que también permite la evaluación en un punto en asitóticamente n operaciones básicas. Si existiera, podriamos calcular el polinomio y evaluarlo en n puntos con un costo operacional de n^2 - mejor que la base monomial y la base de Lagrange. De hecho, una base con esta propiedad existe, y la vamos a presentar en la proxima sección.

5.1.3 Diferencias divididas de Newton

Volvemos al problema de interpolación: supongamos que tenemos $n+1$ pares de números $(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)$ y queremos determinar el polinomio interpolante. Esta vez definimos los *polinomios de Newton*

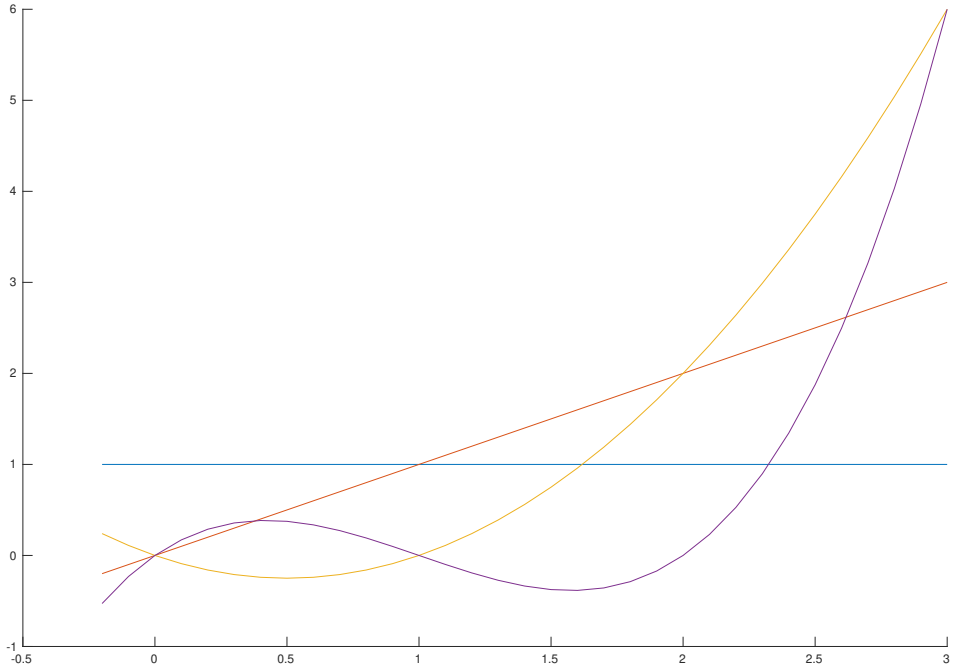
$$\begin{aligned} N_0(x) &= 1, \\ N_j(x) &= \prod_{k=0}^{j-1} (x - x_k), \quad j = 1, \dots, n. \end{aligned}$$

Observamos primero la propiedad importante

$$N_j(x_m) = 0 \quad \text{para } m < j. \tag{5.6}$$

Se puede mostrar que $\{N_0, \dots, N_n\}$ es una base de $\mathbb{P}_n$, y se llama *base de Newton*. Es decir, el polinomio interpolante tiene la representación

$$p(x) = \sum_{k=0}^{n} \lambda_k N_k(x).$$



Los polynomios de Newton asociados a los puntos $x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$.

Para calcular los coeficientes $\lambda_0, \ldots, \lambda_n$, observamos que la propiedad (5.6) implica que la matriz A del sistema $A\lambda = y$ que tenemos que resolver para calcular los coeficientes en la representación con la base de Newton tiene la forma

$$\begin{pmatrix} N_0(x_0) & N_1(x_0) & \ldots & N_n(x_0) \\ N_0(x_1) & N_1(x_1) & \ldots & N_n(x_1) \\ \vdots & & \ddots & \\ N_0(x_n) & N_1(x_n) & \ldots & N_n(x_n) \end{pmatrix} = \begin{pmatrix} N_0(x_0) & 0 & \ldots & 0 \\ N_0(x_1) & N_1(x_1) & \ldots & 0 \\ \vdots & & \ddots & \\ N_0(x_n) & N_1(x_n) & \ldots & N_n(x_n), \end{pmatrix}$$

es decir, es triangular inferior. Por lo tanto, usando sustitución descendente podemos resolver el sistema $A\lambda = y$ en asintóticamente n^2 operaciones. De hecho, la sustitución descendente para resolver este sistema se puede llevar a cabo usando el esquema de *diferencias divididas de Newton*.

Teorema 49 (Diferencias divididas de Newton)**.** *Para $0 \leq j \leq m \leq n$ definimos das diferencias*

divididas

$$y[m] := y_j,$$
$$y[j,\dots,m] := \frac{y[j+1,\dots,m] - y[j,\dots,m-1]}{x_m - x_j}.$$

Entonces, los coeficientes $\lambda_0,\dots,\lambda_n$ del polinomio interpolante en la base de Newton están dados por $\lambda_k = y[0,\dots,k]$. □

El esquema para calcular los diferencias divididas según el ultimo teorema es

$$\begin{array}{ccccccccc}
y_0 = y[0] & \to & y[0,1] & \to & y[0,1,2] & \to & \dots & \to & y[0,\dots,n] \\
 & & & \nearrow & & \nearrow & & \nearrow & \\
y_1 = y[1] & \to & y[1,2] & \to & y[1,2,3] & \to & \dots & & \\
 & & & \nearrow & & \nearrow & & & \\
y_2 = y[2] & \to & y[2,3] & \to & & & & & \\
\vdots & & \vdots & \vdots & & & & & \\
y_{n-1} = y[n-1] & \to & y[n-1,n] & \nearrow & & & & & \\
 & \nearrow & & & & & & & \\
y_n = y[n] & & & & & & & &
\end{array}$$

Ejemplo 50. *Sean $(0,1),(2,-1),(5,3),(-1,2)$ datos. Calcule el polinomio interpolante en la base de Newton usando diferencias divididas.*

Usamos la notación

$$(x_0,y_0) = (0,1),$$
$$(x_1,y_1) = (2,-1),$$
$$(x_2,y_2) = (5,3),$$
$$(x_3,y_3) = (-1,2).$$

Entonces las diferencias divididas de la primera columna son

$$y[0] = 1$$
$$y[1] = -1$$
$$y[2] = 3$$
$$y[3] = 2.$$

Las diferencias divididas de la segunda columna son

$$y[0,1] = \frac{y[1]-y[0]}{x_1-x_0} = \frac{-1-1}{2-0} = -1,$$
$$y[1,2] = \frac{y[2]-y[1]}{x_2-x_1} = \frac{3+1}{5-2} = \frac{4}{3},$$
$$y[2,3] = \frac{y[3]-y[2]}{x_3-x_2} = \frac{2-3}{-1-5} = \frac{1}{6}.$$

Las diferencias divididas de la tercera columna son

$$y[0,1,2] = \frac{y[1,2]-y[0,1]}{x_2-x_0} = \frac{\frac{4}{3}+1}{5} = \frac{7}{15}$$
$$y[1,2,3] = \frac{y[2,3]-y[1,2]}{x_3-x_1} = \frac{\frac{1}{6}-\frac{4}{3}}{-1-2} = \frac{7}{18}.$$

La última columna entonces es

$$y[0,1,2,3] = \frac{y[1,2,3]-y[0,1,2]}{x_3-x_0} = \frac{\frac{7}{18}-\frac{7}{15}}{-1} = \frac{7}{90}.$$

Por lo tanto,

$$\begin{aligned} p(x) &= y[0]+y[0,1](x-x_0)+y[0,1,2](x-x_0)(x-x_1)+y[0,1,2,3](x-x_0)(x-x_1)(x-x_2) \\ &= 1+(-1)x+\frac{7}{15}x(x-2)+\frac{7}{90}x(x-2)(x-5). \end{aligned}$$

□

El esquema para calcular las diferencias divididas de Newton es parecido al esquema de Neville, es decir, un loop exterior sobre la diferencia de m y j, y un loop interior sobre m.

```
In [107]: def divdif(x,y):
              # diferencias divididas de Newton

              n = x.shape[0]-1 # grado del polinomio interpolante

              aux1 = np.zeros(n+1)
              aux2 = np.zeros(n+1)

              for k in range(0,n+1):
                  aux1[k] = y[k]

              aux2[0]=aux1[0]

              for mmj in range(1,n+1):
                  for m in range(mmj,n+1):
                      j = m-mmj
                      aux1[j] = (aux1[j+1]-aux1[j])/(x[m]-x[j])

                  aux2[mmj] = aux1[0]

              return aux2

In [108]: x = np.asarray([0, 2, 5, -1])
          y = np.asarray([1, -1, 3, 2])
```

```
print(divdif(x,y))
```

```
[ 1.         -1.          0.46666667  0.07777778]
```

Ahora que determinamos el polinomio interpolante en la base de Newton, observamos

$$\begin{aligned} p(x) &= \sum_{k=0}^{n} \lambda_k N_k(x) = \lambda_0 + \sum_{k=1}^{n} \lambda_k \prod_{j=0}^{k-1} (x - x_j) \\ &= \lambda_0 + (x - x_0) \sum_{k=1}^{n} \lambda_k \prod_{j=1}^{k-1} (x - x_j) \\ &= \lambda_0 + (x - x_0) \left(\lambda_1 + \sum_{k=2}^{n} \lambda_k \prod_{j=1}^{k-1} (x - x_j) \right) \\ &= \lambda_0 + (x - x_0)(\lambda_1 + (x - x_1)(\lambda_2 + (x - x_2)(\dots(\lambda_{n-1} + \lambda_n(x - x_{n-1}))\dots))), \end{aligned}$$

y la última formula necesita asintóticamente n operaciones básicas para evaluarse. Por lo tanto la podemos evaular en n puntos con un costo de n^2.

5.1.4 Teoria del error de interpolación polynomial

Hasta ahora hemos considerado puntos discretos (x_j, y_j) y el polinomio interpolante $p_n \in \mathbb{P}_n$, es decir, $p_n(x_j) = y_j$. Si los valores y_j están dados por una función $y_j = f(x_j)$, entonces obviamente $f(x_j) - p_n(x_j) = 0$. Sin embargo, nos podemos preguntar como se comporta el error $f(x) - p_n(x)$ en cualquier otro punto x.

Teorema 51. *Sean $x_0, \ldots, x_n \in (a, b)$ y f una función $n+1$ veces continuamente diferenciable en (a, b). Sea $p_n \in \mathbb{P}_n$ el polinomio interpolante que satisface $p_n(x_j) = f(x_j)$, $j = 0, \ldots, n$. Entonces, para cada $x \in (a, b)$ existe un punto $\xi \in (a, b)$ tal que*

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{j=0}^{n} (x - x_j).$$

Idea de la demostración: Definimos $w(y) := \prod_{j=0}^{n}(y - x_j)$ y

$$F(y) := [f(x) - p_n(x)]\, w(y) - [f(y) - p_n(y)]\, w(x).$$

Notamos primero que $w(x_j) = 0$ y $f(x_j) - p_n(x_j) = 0$ para $j = 0, \ldots, n$. Eso implica $F(x_j) = 0$ para $j = 0, \ldots, n$. También se tiene $F(x) = 0$. Es decir, F tiene $n + 2$ raizes en (a, b). Por el teorema de Rolle, F' tiene $n + 1$ raizes en (a, b), F'' tiene n raizes en (a, b), etc., $F^{(n+1)}$ tiene una raiz en (a, b), la llamamos ξ. Calculamos

$$F^{(n+1)}(y) = [f(x) - p_n(x)]\, (n+1)! - f^{(n+1)}(y) w(x),$$

y asi

$$0 = [f(x) - p_n(x)]\, (n+1)! - f^{(n+1)}(\xi) w(x).$$

□

En el último teorema tenemos una *identidad* para el error. Sin embargo, este resultado no sirve mucho en la practica, pues, no conocemos ξ. Por eso tomamos el máximo de todos los $x \in (a, b)$ del valor absoluto, y asi llegamos a

$$\max_{x \in (a,b)} |f(x) - p_n(x)| \leq \frac{\max_{x \in (a,b)} |f^{(n+1)}(x)|}{(n+1)!} \max_{x \in (a,b)} |\prod_{j=0}^{n} (x - x_j)|. \tag{5.7}$$

Ejemplo 52. *Sea $f(x) = \log(1+x)$, $x_0 = 0$, y $x_1 = 1$. Sea p_1 el polinomio interpolante de grado 1. Calcule p_1 y cuota para el error, es decir un número $c \in \mathbb{R}$ tal que*

$$\max_{x \in (0,1)} |f(x) - p_1(x)| \leq c.$$

Los dos polinomios de Lagrange son

$$L_0(x) = \frac{x-1}{-1},$$
$$L_1(x) = \frac{x}{1},$$

entonces

$$p_1(x) = -\log(1)(x-1) + \log(2)x = \log(1) + x\left(\log(2) - \log(1)\right).$$

Para estimar el error con (5.7)*, calculamos*

$$f''(x) = -\frac{1}{(1+x)^2},$$

y asi

$$\max_{x\in[0,1]} |f''(x)| = 1.$$

Además,

$$\max_{x\in[0,1]} |x(x-1)| = \frac{1}{4},$$

y por (5.7) *obtenemos*

$$\max_{x\in(0,1)} |f(x) - p_1(x)| \leq \frac{1}{8}.$$

□

Los límites de interpolación polinomial

Vamos a analizar el error polinomial (5.7) usando puntos *equiespaciados*, es decir

$$x_j = a + j\frac{b-a}{n}, \quad j = 0, \ldots, n.$$

Para maximizar la función $\prod_{j=0}^{n}(x - x_j)$ supongamos que $x_j \leq x \leq x_{j+1}$ para $0 \leq j \leq n$. Si anotamos $h = (b-a)/n$, entonces

$$\begin{aligned} |\prod_{j=0}^{n}(x - x_j)| &\leq (j+1)h \cdot jh \cdot \cdots \cdot 2h \cdot (x - x_j) \cdot (x_{j+1} - x) \cdot 2h \cdot \cdots \cdot (n-j)h \\ &= \frac{(x_{j+1} - x_j)^2}{4} h^{n-1}(j+1)!(n-j)! \\ &\leq \frac{h^{n+1}}{4} n! \end{aligned}$$

Concluimos

$$\max_{x\in(a,b)} |f(x) - p_n(x)| \leq \max_{x\in(a,b)} |f^{(n+1)}(x)| \frac{h^{n+1}}{4(n+1)}.$$

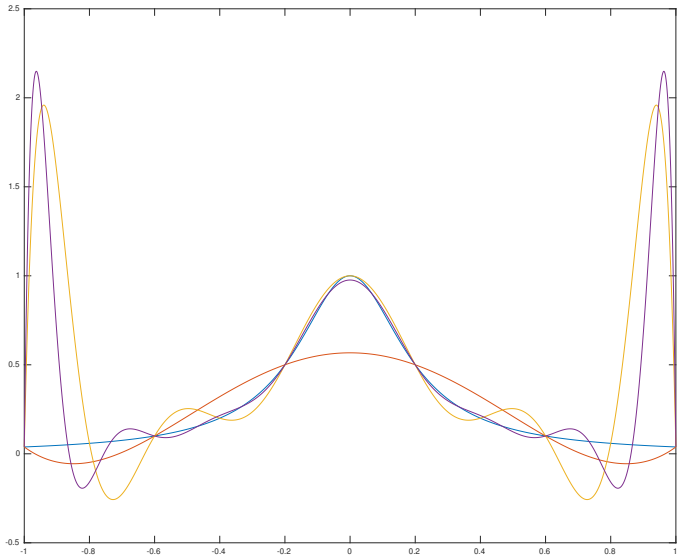
Dado que

$$\frac{h^{n+1}}{4(n+1)} \to 0$$

para $n \to \infty$, será razonable esperar que también

$$\max_{x\in(a,b)} |f(x) - p_n(x)| \to 0,$$

pues, usamos mas y mas puntos para aproximar la función f. Lamentablemente no es asi. Hay funciones donde $\max_{x\in(a,b)} |f^{(n+1)}(x)|$ crece tan rapido que el error de interpolación explota si $n \to \infty$. El ejemplo clásico es el *fenomeno de runge*, donde $a = -1$, $b = 1$, $f(x) = 1/(1 + 25x^2)$. En la figura 5.3 presentamos los polinomios de grado 5, 10, y 15. Se nota que en los extremos del intervalo los polinomios empiezan a oscilar.



Fenomeno de Runge: Los polynomios interpolantes a f de grado $5, 10, 15$ en puntos equiespaciados.

5.2 Interpolación polinomial a trozos - splines

La interpolación polinomial puede presentar oscilaciónes y errores grandes si se usa un grado polinomial muy alto para funciones no suaves, eso es un efecto que la "rigidez" que se produce al exigir que la función interpolante sea globalmente suave. Una alternativa es interpolación a trozos, donde se reduce el grado de diferenciabilida de la función interpolante en los nodos.

Definición 53 (Splines)**.** Sea $[a, b]$ un intervalo, y $\Delta = \{x_0, \ldots, x_n\}$ unos nodos, es decir, $a = x_0 < x_1 < \cdots < x_n = b$. Se define el espacio de los splines de grado $k \in \mathbb{N}$ por

$$\mathcal{S}_k(\Delta) = \left\{ s \in C^{k-1}([a,b]) \mid s \text{ restringida a } [x_{j-1}, x_j] \text{ es elemento de } \mathbb{P}_k, \quad j = 1, \ldots, n \right\}$$

Los casos mas importantes en la práctica son splines lineales $\mathcal{S}_1(\Delta)$ y splines cúbicos $\mathcal{S}_3(\Delta)$. Para el ojo humano, una función $\mathcal{S}_3(\Delta)$ es *suave*, pues, el ojo humano no puede distinguir saltos en la tercera derivada. Es por eso que los splines se usan en la práctica para computación gráfica (específicamente para gráfica vectorial) y diseño asistido por computadora (CAD, *computer-aided design*). La última evolución en este contexto son los B-splines racionales no uniformes (NURBS, *non-uniform rational B-spline*).

Nuestro objetivo es lo siguiente: dada una funcion $f : [a, b] \to \mathbb{R}$, queremos buscar un spline interpolante $s \in \mathcal{S}_k(\Delta)$, es decir

$$s(x_j) = f(x_j), \quad j = 0, \ldots, n.$$

Como en el caso de la interpolación polinomial nos preguntamos

1. por la existencia y unicidad de un spline interpolante,

2. y por el error de la interpolación y su relación con los nodos Δ, es decir, con la resolución global

$$h = \max_{j=1,\ldots,n} |x_j - x_{j-1}|.$$

5.2.1 Splines lineales

Para splines lineales $\mathcal{S}_1(\Delta)$ es facil analizar la existencia y unicidad de un spline interpolante, pues obviamente

$$s(x) = s_{j-1}(x) = f(x_{j-1})\frac{x - x_j}{x_{j-1} - s_j} + f(x_j)\frac{x - x_{j-1}}{x_j - x_{j-1}} \quad \text{para } x \in [x_{j-1}, x_j] \tag{5.8}$$

es el único polinomio de grado 1 que interpola $(x_{j-1}, f(x_{j-1}))$ y $(x_j, f(x_j))$. Obviamente, las *funciones techo*

$$\varphi_0 = \begin{cases} \frac{x-x_0}{x_0-x_1} & x_0 \le x \le x_1 \\ 0 & x_1 < x \end{cases},$$

$$\varphi_j = \begin{cases} \frac{x-x_{j-1}}{x_j-x_{j-1}} & x_{j-1} \le x \le x_j \\ \frac{x-x_{j+1}}{x_j-x_{j+1}} & x_j \le x \le x_{j+1} \\ 0 & x \le x_{j-1} \text{ o } x_{j+1} \le x \end{cases}, j = 1, \ldots, n-1,$$

$$\varphi_n = \begin{cases} \frac{x-x_{n-1}}{x_n-x_{n-1}} & x_{n-1} \le x \le x_n \\ 0 & x \le x_{n-1} \end{cases},$$

son una base de $\mathcal{S}_(\Delta)$, y concluimos que $\dim \mathcal{S}_1(\Delta) = n+1$. Podemos escribir

$$s(x) = \sum_{k=0}^{n} f(x_k)\varphi_k(x).$$

Por la identidad (5.8) también obtenemos el siguiente resultado.

Lema 54. *Sea $[a,b]$ un intervalo y $\mathcal{S}_1(\Delta)$ un espacio de splines lineales. Sea f dos veces diferenciable en $[a,b]$. Entonces existe una constante $C > 0$ que no depende de f o Δ, tal que*

$$\max_{x\in[a,b]} |f(x) - s(x)| \le \frac{h^2}{8} \max_{x\in[a,b]} |f''(x)|.$$

□

Ejemplo 55. *Sea $[a,b] = [0,1]$ y $f(x) = \cos(2\pi x)$. Sean $\Delta = \{a = x_0, \ldots, x_n = b\}$ nodos y s el spline lineal interpolante. Para lograr un error*

$$\max_{x\in[0,1]} |f(x) - s(x)| \le 10^{-6},$$

será suficiente tener

$$\frac{h^2}{8} 4\pi^2 \max_{x\in[0,1]} |\cos(2\pi x)| = \frac{h^2}{2}\pi^2 \le 10^{-6},$$

es decir,

$$h \le \frac{\sqrt{2}\cdot 10^{-3}}{\pi} \approx 0.000450158$$

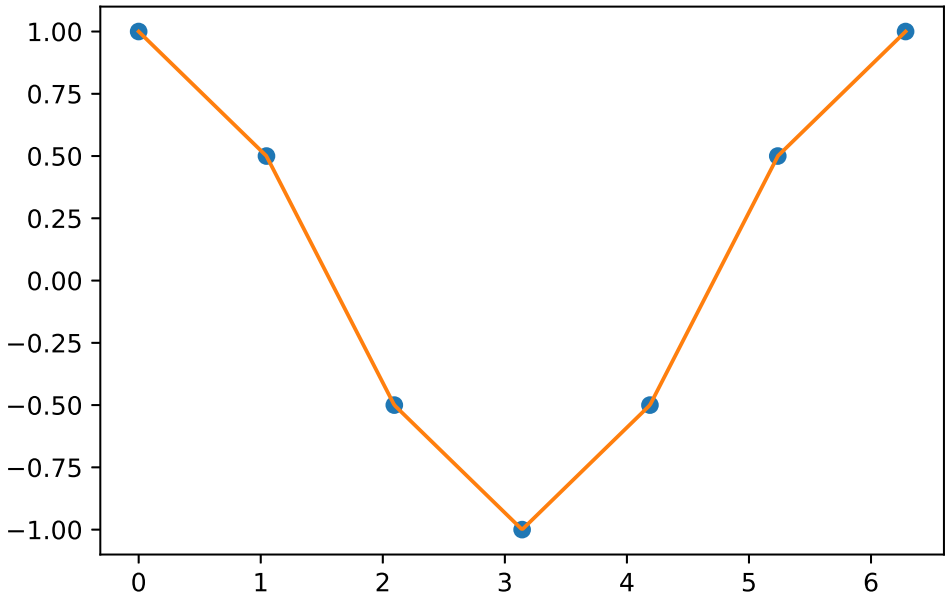
Si usamos nodos equiespaciados, es decir $x_j = jh$, $j = 0, \ldots, n$, $h = 1/n$, entonces

$$n = \frac{1}{h} \ge 2221.44,$$

o bien $n \ge 2222$. □

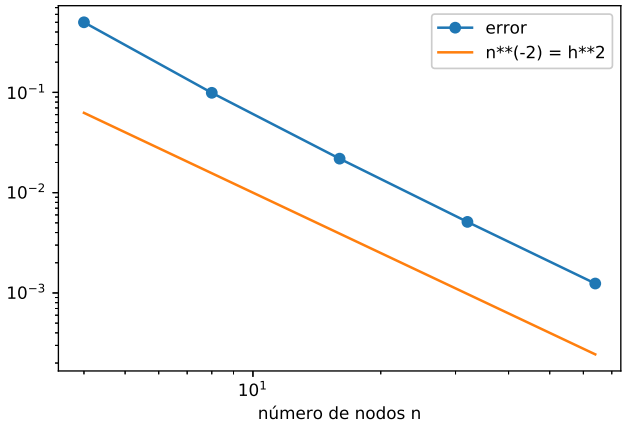
En en modulo scipy de Python existen funciones para determinar splines y operar con ellos.

```
In [69]: import numpy as np
         import matplotlib.pyplot as plt
         from scipy.interpolate import interp1d

         x = np.linspace(0, 2*np.pi, num=7, endpoint=True)
         y = np.cos(x) # interpolamos el coseno
         f = interp1d(x, y) # determinar el spline lineal interpolante

         xnew = np.linspace(0, 2*np.pi, num=500, endpoint=True)
         plt.plot(x,y,'o', xnew, f(xnew), '-')
         plt.savefig("cos_spline_lin.eps")
         plt.show()
```



Verificamos numéricamente la cota del error del último Lema. Si usamos $n+1$ nodos equiespaciados, es decir $x_j = jh$, $j = 0, \ldots, n$ con $h = 1/n$, entonces esperamos que el error se comporta como n^{-2} si la función f es dos veces diferenciable en $[a, b]$. Verificaremos este comportamiento en un plot doble logaritmico.

```
In [68]: N = 5

         ps = np.zeros(N)
         error = np.zeros(N)

         for k in range(0,N):
             n = 2**(k+2)
             ps[k] = n
             x = np.linspace(0, 2*np.pi, num=n, endpoint=True)
             y = np.cos(x) # interpolamos el coseno
             f = interp1d(x, y) # determinar el spline lineal interpolante

             # "calcular" el error
             xnew = np.linspace(0, 2*np.pi, num=500, endpoint=True)
             error[k] = max(np.abs(np.cos(xnew) - f(xnew)))

         plt.loglog(ps,error,'o-',ps,ps**(-2))
         plt.xlabel("número de nodos n")
         plt.legend(["error","n**(-2) = h**2"])


Out[68]: <matplotlib.legend.Legend at 0x7f19fe25a950>
```



Ejemplo: Sea Δ la distribución de nodos

$$x_0 < x_1 < \cdots < x_n.$$

Consideramos los splines cuadraticos $\mathcal{S}_2(\Delta)$.

(i) ¿Cual es la dimensión de $\mathcal{S}_2(\Delta)$?

(ii) Sea $\Delta = \{-2, 0, 2\}$. El objetivo es calcular $s \in \mathcal{S}_2(\Delta)$ tal que $s(-2) = s(0) = s(2) = 0$ y $s(1) = 1$. ¿Según (i), existe una función s con estas propiedades? ¿Es única? Si es asi, determinela.

Solución:

(i) Funciones en $\mathcal{S}_2(\Delta)$ son polinimios de grado 2 en cada subintervalo, continuas, y con primera derivada continua. Hay n subintervalos, y una función que es un polinomio de grado 2 a trozos tiene entonces $3n$ coeficients libres. Hay $n-1$ nodos interiores $x_1, \ldots, x_{n-1}$, y la continuidad de la función y de su primera derivada se traducen a $n-1$ y $n-1$ condiciones. En total, observamos que tenemos $3n - 2(n-1) = n + 2$ coeficientes libres, es decir, la dimension de $\mathcal{S}_2(\Delta)$ es $n + 2$.

(ii) En este caso, $n = 2$, es decir, tenemos 4 coeficientes libres. Dado que tenemos 4 condiciones linealmente independientes, existe única función s como pedido. Anotamos

$$s(x) = \begin{cases} a_0 + b_0 x + c_0 x^2 & x \in [-2, 0], \\ a_1 + b_1 x + c_1 x^2 & x \in [0, 2]. \end{cases}$$

Para que $s \in \mathcal{S}_2$, se necesita $s_0(0) = s_1(0)$, es decir $a_0 = a_1 =: a$, y $s_0'(0) = s_1'(0)$, es decir $b_0 = b_1 =: b$. Dado que $s_0(0) = 0$, concluimos $a = 0$. Nos quedan entonces tres ecuaciones en tres desconocidas,

$$\begin{aligned} 0 &= s(-2) = -2b + 4c_0 \\ 0 &= s(2) = 2b + 4c_1 \\ 1 &= s(1) = b + c_1. \end{aligned}$$

La solución del último sistema es $b = 2$, $c_0 = 1$, $c_1 = -1$.

5.2.2 Splines cúbicos

Consideramos ahora splines cúbicos $\mathcal{S}_3(\Delta)$ sobre un intervalo $[a,b]$. Para que s pertenezca a $\mathcal{S}_3(\Delta)$, tiene que ser un polinomio de grado 3 sobre cada subintervalo, es decir

$$s(x) = s_{j-1}(x) = a_{j-1} + b_{j-1}(x - x_{j-1}) + c_{j-1}(x - x_{j-1})^2 + d_{j-1}(x - x_j)^3 \quad \text{para } x \in [x_{j-1}, x_j].$$

En total tenemos $4n$ coeficientes para determinar s en la última representación. Para que s sea un spline cúbico, las derivadas hasta el orden 2 de s en los nodos tienen que ser continuas,

$$\begin{aligned} s_{j-1}(x_j) &= s_j(x_j), \quad j = 1, \dots, n-1, \\ s'_{j-1}(x_j) &= s'_j(x_j), \quad j = 1, \dots, n-1, \\ s''_{j-1}(x_j) &= s''_j(x_j), \quad j = 1, \dots, n-1, \end{aligned} \tag{5.9}$$

es decir, $3n-3$ ecuaciones. Concluimos que $\dim \mathcal{S}_3(\Delta) = 4n - (3n-3) = n+3$. Para que s sea un spline interpolante para una función f, se pide

$$s(x_j) = f(x_j), \quad j = 0, \dots, n. \tag{5.10}$$

es decir, $n+1$ condiciones. Para fijar las dos condiciones restantes, lo mas común es agregar

(i) **condiciones de Hermite:** $s'(x_0) = y_0$ y $s'(x_n) = y_n$,

(ii) **condiciones naturales:** $s''(x_0) = s''(x_n) = 0$,

(iii) **condiciones periodicas:** $s'(x_0) = s'(x_n)$ y $s''(x_0) = s''(x_n)$.

Antes del uso de computadoras en diseño, se utilizaron vigas de madera delgadas y flexibles para dibujar curvas (particularmente en la construcción naval). Las vigas se fijaron en puntos del plano (las condiciones de interpolación), y entre medio los puntos, las vigas tomaron una forma para minimizar la energía elastica, es decir, la curvatura. Eso se refleja justamente en el siguiente resultado.

Lema 56. *Sea $[a,b]$ un intervalo y $\mathcal{S}_3(\Delta)$ el espacio de splines cúbicos con nodos Δ. Sea f una función continua en $[a,b]$. Entonces, existe único $s \in \mathcal{S}_3(\Delta)$ que satisface (5.10), (5.9), y una de las condiciones (i)-(iii). Además, el spline s minimiza la energía elastica, es decir,*

$$\int_a^b |s''(x)|^2 \, dx \le \int_a^b |y''(x)|^2 \, dx$$

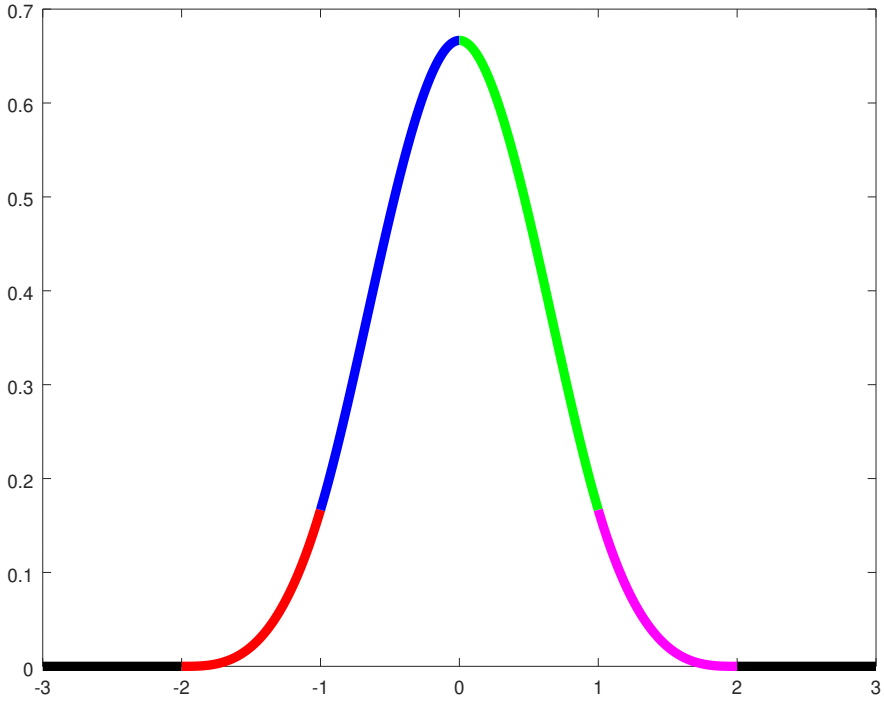
para todas las funciones y que son dos veces continuamente diferenciable en $[a,b]$ que cumplen con $y(x_j) = f(x_j), \quad j = 0, \dots, n$, y la respectiva condicion de (i)-(iii). □

Para determinar entonces un spline cúbico interpolante en la forma indicada al principio, hay que resolver un sistema lineal de $4n \times 4n$ para los coeficientes $a_{j-1}, b_{j-1}, c_{j-1}, d_{j-1}$, $j = 1, \dots n$, donde las ecuaciones se fabrican a traves de (5.9), (5.10), y una de las condiciones (i)–(iii). Sin

embargo, $\dim \mathcal{S}_3(\Delta) = n+3$, lo que refleja el número de datos que nosotros tenemos. Para tener que resolver solamente un sistema $(n+3) \times (n+3)$, necesitamos una base de $\mathcal{S}_3(\Delta)$. Se define la función

$$B(x) = \begin{cases} \frac{2}{3} - x^2(1 - \frac{1}{2}|x|) & |x| \leq 1 \\ \frac{1}{6}(2 - |x|)^3 & 1 \leq |x| \leq 2 \\ 0 & 2 \leq |x|. \end{cases}$$

Se puede usar escalamientos y traslaciones de B para definir una base de $\mathcal{S}_3(\Delta)$ en el caso de



El spline básico B.

nodos equiespaciados, es decir,

$$x_j - x_{j-1} = h \quad \text{para todo } j = 1, \ldots, n.$$

Definomos además los nodos artificiales

$$x_{-k} = a - kh, \quad x_{n+k} = b + kh, \quad k \geq 1,$$

y las funciones

$$B_j(x) := B\left(\frac{x - x_j}{h}\right), \quad \text{para todo } j = -1, \ldots, n+1.$$

Lema 57 (B-Splines)**.** *Sean $\Delta = \{x_0, \ldots, x_n\}$ nodos equiespaciados, es decir, $a = x_0 < x_1 < \cdots < x_n = b$ y $x_j - x_{j-1} = h = 1/n$ para todo $j = 1, \ldots, n$. Sean $x_{-k} = a - kh$, y $x_{n+k} = b + kh$ para $k \geq 1$. Las funciones B_j, $j = -1, 0, \ldots, n, n+1$ se llaman B-splines. Se tiene los siguientes resultados.*

(i) $B_j \in \mathcal{S}_3(\Delta)$, el soporte de B_j es el intervalo $[x_{j-2}, x_{j+2}]$, y para cada $j = 0, \ldots, n$, los únicos B-splines que no desaparecen en x_j son B_{j-1}, B_j, y B_{j+1}.

(ii) $B_j''(x_j) = -2/h^2$, $B_j''(x_{j-1}) = B_j''(x_{j+1}) = 1/h^2$.

(iii) Las funciones $\{B_{-1}, B_0, \ldots, B_n, B_{n+1}\}$ son una base de $\mathcal{S}_3(\Delta)$.

□

Es decir, cada $s \in \mathcal{S}_3(\Delta)$ tiene única representación

$$s(x) = \sum_{j=-1}^{n+1} \lambda_j B_j(x).$$

Para determinar los coeficientes λ_j, notamos que por el Lema 57 (i) tenemos

$$\begin{aligned} f(x_j) = s(x_j) &= \lambda_{j-1} B_{j-1}(x_j) + \lambda_j B_j(x_j) + \lambda_{j+1} B_{j+1}(x_j) \\ &= \frac{1}{6}(\lambda_{j-1} + 4\lambda_j + \lambda_{j+1}), \end{aligned}$$

para $j = 0, \ldots, n$. Si agregamos condiciones naturales $s''(x_0) = s''(x_n) = 0$, obtenemos por (ii)

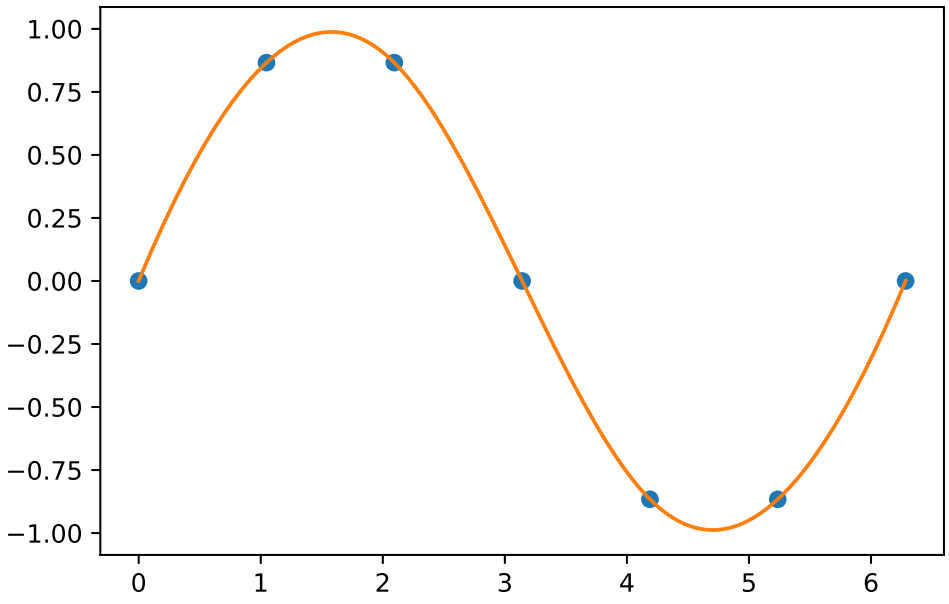
$$\begin{aligned} 0 = s''(x_j) &= \lambda_{j-1} B_{j-1}''(x_j) + \lambda_j B_j''(x_j) + \lambda_{j+1} B_{j+1}''(x_j) \\ &= \frac{1}{h^2}(\lambda_{j-1} - 2\lambda_j + \lambda_{j+1}). \end{aligned}$$

Finalmente, llegamos al sistema lineal de tamaño $(n+3) \times (n+3)$

$$\frac{1}{6}\begin{pmatrix} 1 & -2 & 1 & & & & & \\ 1 & 4 & 1 & & & & & \\ & 1 & 4 & 1 & & & & \\ & & 1 & 4 & 1 & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & 1 & 4 & 1 & \\ & & & & & 1 & 4 & 1 \\ & & & & & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} \lambda_{-1} \\ \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_n \\ \lambda_{n+1} \end{pmatrix} = \begin{pmatrix} 0 \\ f(x_0) \\ \vdots \\ f(x_n) \\ 0 \end{pmatrix}.$$

Con *interp1d* de Scipy se puede determinar también un spline cúbico natural interpolante.

```
In [67]: import numpy as np
         import matplotlib.pyplot as plt
         from scipy.interpolate import interp1d

         x = np.linspace(0, 2*np.pi, num=7, endpoint=True)
         y = np.sin(x) # interpolamos el seno
         f = interp1d(x, y, kind='cubic') # determinar el spline lineal interpolante

         xnew = np.linspace(0, 2*np.pi, num=500, endpoint=True)
         plt.plot(x,y,'o', xnew, f(xnew), '-')
         plt.savefig("sin_spline_cubic.eps")
         plt.show()
```



5.3 La transformación rapida de Fourier

Si queremos interpolar una función 2π-periodica, entonces tendrá sentido usar un interpolante 2π-periodico. Una función compleja 2π-periodica f pueder ser representada por su serie de Fourier,

$$f(x) \approx \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(kx) + b_k \sin(kx).$$

Usamos el símbolo $\approx$, porque la convergencia de la suma y sus valores dependen de las propiedades de la función f. Usando la identidad de Euler $e^{i\theta} = \cos(\theta) + i\sin(\theta)$ podemos escribir

$$f(x) \approx \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(kx) + b_k \sin(kx) = \sum_{k=-\infty}^{\infty} c_k e^{ikx},$$

donde $c_0 = a_0/2$, $c_k = (a_k - ib_k)/2$, $c_{-k} = (a_k + ib_k)/2$. Podemos cortar la suma del lado derecho e interpolar f por una suma de la forma

$$\sum_{k=-N}^{N} c_k e^{ikx} = e^{-iNx} \sum_{k=0}^{2N} c_{k-N} e^{ikx}.$$

5.3.1 La transformación discreta de Fourier (DFT)

Sea $\mathbb{T}_{N-1} := \left\{ p : [0, 2\pi] \to \mathbb{C} \mid p(x) = \sum_{k=0}^{N-1} c_k e^{ikx} \right\}$ el espacio vectorial de los polinomios trigonometricos sobre $\mathbb{C}$.

Lema 58. *Se tiene* $\dim \mathbb{T}_{N-1} = N$, *y para nodos* $x_0, \ldots, x_{N-1} \in [0, 2\pi)$ *distintos entre si y valores* $y_j \in \mathbb{C}$ *existe único polinomio trigonometrico interpolante* $p \in \mathbb{T}_{N-1}$ *con* $p(x_j) = y_j$ *para todo* $j = 0, \ldots, N-1$. □

A partir de ahora consideramos nodos *equiespaciados*, es decir

$$x_j = 2\pi \frac{j}{N}, \quad j = 0, \ldots N-1. \tag{5.11}$$

En este caso podemos encontrar una formula explicita para calcular los coeficientes del polinomio trigonometrico interpolante. Introducimos la N-ésima raiz unitaria

$$\omega_N = e^{2\pi i/N}.$$

Lema 59. *Para* $0 \le k, \ell \le N-1$ *se tiene*

$$\frac{1}{N} \sum_{j=0}^{N-1} \left(\omega_N^{k-\ell} \right)^j = \delta_{k,\ell}.$$

Proof. El caso $k = \ell$ es fácil, dado que $\omega_N^0 = 1$. Para $k \neq \ell$ notamos $\omega_N^{k-\ell} \neq 1$, y la suma es una suma geometrica. Calculamos

$$\sum_{j=0}^{N-1} \left(\omega_N^{k-\ell}\right)^j = \frac{1 - \omega_N^{(k-\ell)N}}{1 - \omega_N^{k-\ell}} = 0,$$

donde usamos $\omega_N^{(k-\ell)N} = e^{2\pi i(k-\ell)} = 1$. □

Lema 60. *Sean los nodos $x_0, \ldots, x_{N-1}$ dados por (5.11), y sean $y_j \in \mathbb{C}$. Sea $p \in \mathbb{T}_{N-1}$ el único polinomio trigonometrico interpolante, es decir $p(x_j) = y_j$ para todo $j = 0, \ldots, N-1$. Entonces*

(1) El polinomio p está dado por

$$p(x) = \sum_{k=0}^{N-1} c_k e^{ikx}, \qquad \text{donde } c_k = \frac{1}{N} \sum_{\ell=0}^{N-1} \omega_N^{-k\ell} y_\ell.$$

(2) Con la matriz $V_N \in \mathbb{C}^{N\times N}$ dada por $(V_N)_{k,j} := \omega_N^{-kj}$, $k, j = 0, \ldots, N-1$ y los vectores $y = (y_0, \ldots, y_{N-1})^\top \in \mathbb{C}^N$, $c = (c_0, \ldots, c_{N-1})^\top \in \mathbb{C}^N$ se tiene

$$c = \frac{1}{N} V_n y.$$

Proof. Para demostrar (1) es suficiente verificar $p(x_j) = y_j$. Notamos $e^{ikx_j} = e^{2\pi ikj/N} = \omega_N^{kj}$, y con Lemma 59 concluimus

$$p(x_j) = \sum_{k=0}^{N-1} \frac{1}{N} \sum_{\ell=0}^{N-1} \omega_N^{-k\ell} y_\ell e^{ikx_j} = \sum_{\ell=0}^{N-1} y_\ell \frac{1}{N} \sum_{k=0}^{N-1} \omega_N^{(j-\ell)k} = \sum_{\ell=0}^{N-1} y_\ell \delta_{j,\ell} = y_j.$$

□

La aplicación lineal

$$\mathcal{F}_N : \begin{cases} \mathbb{C}^N \to \mathbb{C}^N \\ y \mapsto \frac{1}{N} V_N y \end{cases}$$

se llama *transformación discreta de Fourier*[1]. Efectivamente, la formula para calcular los c_k refleja la formula de la transformación de Fourier

$$\mathcal{F}[f](\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x\xi}\, dx.$$

[1] *Discrete Fourier transform, DFT.*

Notamos que V_N contiene solamente los N elementos diferentes $\omega_N^{-\ell}$, $\ell = 0, \ldots, N-1$. Es decir, la computación $\mathcal{F}_N(y)$ a traves de la multiplicación matriz-vector necesita asintóticamente N operaciones para ensamblar la matriz V_N y N^2 operaciones para la multiplicación matriz-vector, en total N^2. Explicitamos por ahora el caso $N = 4$ (y omitimos el factor $1/N$ y usamos $\omega := \omega_4$):

$$\begin{aligned} c_0 &= y_0\omega^0 + y_1\omega^0 + y_2\omega^0 + y_3\omega^0 \\ c_2 &= y_0\omega^0 + y_1\omega^{-1} + y_2\omega^{-2} + y_3\omega^{-3} \\ c_3 &= y_0\omega^0 + y_1\omega^{-2} + y_2\omega^{-4} + y_3\omega^{-6} \\ c_4 &= y_0\omega^0 + y_1\omega^{-3} + y_2\omega^{-6} + y_3\omega^{-9}, \end{aligned}$$

donde observamos 16 productos y 12 sumas. Sin embargo, dado que

$$\begin{aligned} \omega^0 = \omega_{-4} &= 1, \\ \omega^{-2} = \omega^{-6} &- 1, \\ \omega^{-1} = \omega^{-9} &= -i, \\ \omega^{-3} &= i, \end{aligned}$$

podemos escribir

$$\begin{aligned} c_0 &= (y_0 + y_2\omega^0) + \omega^0(y_1 + y_3\omega^0) \\ c_1 &= (y_0 - y_2\omega^0) + \omega^{-1}(y_1 - y_3\omega^0) \\ c_2 &= (y_0 + y_2\omega^0) + \omega^{-2}(y_1 + y_3\omega^0) \\ c_3 &= (y_0 - y_2\omega^0) + \omega^{-3}(y_1 - y_3\omega^0), \end{aligned}$$

donde observamos 8 sumas/restas y 6 productos. De hecho, hemos reducida la DFT de una sucesión de 4 elementos a dos DFTs de subsucesiones.

5.3.2 La transformación rapida de Fourier (FFT)

Una transformación rapida de Fourier es un algoritmo que realiza la operación $V_N y$ con un costo computacional $\mathcal{O}(N \log N)$. Hay varios algoritmos, el mas famoso es el algoritmo de *Cooley-Tukey* (1965). Se basa en la siguiente observación.

Lema 61. *Sea $N = 2M$ y $\omega = e^{\pm 2\pi i/N}$. Entonces, las sumas*

$$c_k = \sum_{j=0}^{N-1} \omega^{kj} y_j, \quad k = 0, \ldots, N-1,$$

pueden ser calculadas como

$$c_{2\ell} = \sum_{j=0}^{M-1} \xi^{\ell j} g_j, \quad c_{2\ell+1} = \sum_{j=0}^{M-1} \xi^{\ell j} h_j, \quad \ell = 0, \ldots, M-1$$

donde $\xi = \omega^2$ y $g_j = y_j + y_{j+M}$, $h_j = (y_j - y_{j+M})\omega^j$. □

El Lema 61 muestra que la transformación discreta de un vector en $\mathbb{C}^N$ puede ser realizada usando dos transformaciones discretas de vectores en $\mathbb{C}^{N/2}$: Para $y \in \mathbb{C}^N$ sea $c = \mathcal{F}_N(y)$. Se define $g_k = y_k + y_{k+N/2}$, $h_k = (y_k - y_{k+N/2})\omega_N^k$, y según Lema 61 tenemos

$$(c_0, c_2, \ldots, c_{2M-2}) = \frac{1}{2}\mathcal{F}_{N/2}(g_0, \ldots, g_{N/2-1})$$
$$(c_1, c_3, \ldots, c_{2M-1}) = \frac{1}{2}\mathcal{F}_{N/2}(h_0, \ldots, g_{N/2-1}).$$

Si aplicamos esta idea de manera recursiva, llegamos al siguiente algoritmo.

Algorithm 1: c = FFT(y) %Cooley-Tukey radix-2 FFT

Input: $N = 2^p$, $p \in \mathbb{N}_0$, $y = (y_0, \ldots, y_{N-1}) \in \mathbb{C}^N$
if $N = 1$ **then**
  $c_0 = y_0$;
else
  $\omega = e^{-2\pi i/N}$;
  $M = N/2$;
  Calcula $g = (g_0, \ldots, g_{M-1})$ con $g_k = y_k + y_{k+M}$;
  Calcula $h = (h_0, \ldots, h_{M-1})$ con $h_k = (y_k - y_{k+M})\omega^k$;
  $(c_0, c_2, \ldots, c_{N-2}) = \frac{1}{2}FFT(g)$;
  $(c_1, c_3, \ldots, c_{N-1}) = \frac{1}{2}FFT(h)$;
end
Output: $c = (c_0, \ldots, c_{N-1}) \in \mathbb{C}^N$

Corolario 62. *Algoritmo 1 calcula la tranformación discreta de Fourier de $y \in \mathbb{C}^N$ con un costo computacional asintótico de $N \log_2 N$.*

Proof. Sea $N = 2^p$ y a_p el número de sumas/restas y m_p el número de productos para calcular $\mathcal{F}_N$. Entonces, aplicamos inducción con respecto a p para demostrar que $a_p = p2^p$ y $m_p = p2^{p-1}$. Para $p = 0$ tenemos $a_0 = 0$ y $m_0 = 0$. Luego,

$$a_{p+1} = 2a_p + 2 \cdot 2^p = 2p2^p + 2 \cdot 2^p = (p+1) \cdot 2 \cdot 2^p = (p+1) \cdot 2^{p+1},$$
$$m_{p+1} = 2m_p + 2^p = p2^p + 2^p = (p+1)2^p.$$

En total,

$$a_p + m_p = p2^p + p2^{p-1} = \frac{3}{2}p2^p = \frac{3}{2}N \log_2(N).$$

□

Chapter 6

Ajuste lineal de curvas

En un problema de ajuste de curvas, el objetivo es *ajustar* una curva con ciertas condiciones adicionales a un conjunto de puntos. En la practica uno se encuentra con el problema de ajustar una cantidad pequeña de parametros de un modelo matemático a una alta cantidad de mediciones. Eso genera un problema *sobredeterminado*, que en general no tiene solución. Por el otro lado ya sabemos que mediciones siempre llevan un error, y por lo tanto no será necesario que el modelo predice de manera exacta todas las mediciones. Por ejemplo, si lanzamos una roca al tiempo $t = 0$ con velocidad v de manera vertical, entonces, según física, al tiempo $t > 0$ la altura será $y(t) = vt - \frac{1}{2}gt^2$, donde g es la constante de gravitación. En teoria, seria suficiente medir la altura en dos instantes t_1 y t_2 para determinar v y g del sistema lineal

$$\begin{pmatrix} t_1 & -\frac{1}{2}t_1^2 \\ t_2 & -\frac{1}{2}t_2^2 \end{pmatrix} \begin{pmatrix} v \\ g \end{pmatrix} = \begin{pmatrix} y(t_1) \\ y(t_2) \end{pmatrix}.$$

Sin embargo, en la practica no somos capaces de medir algo de manera exacta. Entonces tendrá sentido hacer una gran cantidad de mediciones en instantes t_j, $j = 1, \dots, m$ y esperar que el error de medición se compensa en promedio. Eso nos lleva a un sistema lineal que tiene mas filas que columnas

$$\begin{pmatrix} t_1 & -\frac{1}{2}t_1^2 \\ t_2 & -\frac{1}{2}t_2^2 \\ \vdots & \vdots \\ t_m & -\frac{1}{2}t_m^2 \end{pmatrix} \begin{pmatrix} v \\ g \end{pmatrix} = \begin{pmatrix} y(t_1) \\ y(t_2) \\ \vdots \\ y(t_m) \end{pmatrix},$$

y por lo tanto no podemos esperar que tenga una solución.

En terminos mas abstractos, el problema abordado arriba resulta en un sistema lineal

$$Ax = b \quad \text{con } A \in \mathbb{R}^{m \times n}, \text{ donde } m > n. \tag{6.1}$$

El número m respresenta la cantidad de *mediciones* o *condiciones* que tenemos, y el número n representa la cantidad de *parametros* o *grados de libertad* que tiene nuestro modelo. Dado que

en (6.1) la matriz A no es cuadrada, el sistema no tiene solución exacta en general. Una solución exacta existe solamente si el lado derecho b es elemento del imagen $\mathrm{Im}(A)$ de A, el espacio generado por las columnas de A. En el contexto del ejemplo arriba eso significaria que tenemos un modelo exacto del fenómeno y mediciones exactas. En la practica eso no va a pasar, y no podemos determinar una solución exacta de (6.1). Vamos a buscar una **solución de mínimos cuadrados**, es decir, donde el error $\|b - Ax\|_2$ **sea lo más pequeño posible**. Este método se atribuye a Gauss.

6.1 Mínimos cuadrados

Digamos que tenemos n datos (x_j, y_j), $j = 1, \ldots, n$, y nuestro objetivo es determinar una recta $y(x) = p + qx$ que se ajusta a los datos en el sentido de mínimos cuadrados. Es decir, queremos minimizar el error $\|b - Ax\|_2$ donde

$$A = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix}, x = \begin{pmatrix} p \\ q \end{pmatrix}, b = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}. \tag{6.2}$$

Minimizar $\|b - Ax\|_2$ es equivalente a minimizar

$$E(p, q) := \|b - Ax\|_2^2 = \sum_{j=1}^{n} \left(y_j - (p + qx_j)\right)^2,$$

lo que explica el nombre *mínimos cuadrados*. Para minimizar el error tenemos que buscar un punto crítico (p, q), es decir,

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \nabla E(p, q) = \begin{pmatrix} \frac{\partial E}{\partial p}(p, q) \\ \frac{\partial E}{\partial q}(p, q) \end{pmatrix} = \begin{pmatrix} -2\sum_{j=1}^{n} \left(y_j - (p + qx_j)\right) \\ -2\sum_{j=1}^{n} \left(y_j - (p + qx_j)\right) x_j \end{pmatrix}.$$

Esto nos lleva al sistema lineal

$$\begin{pmatrix} n & \sum_{j=1}^{n} x_j \\ \sum_{j=1}^{n} x_j & \sum_{j=1}^{n} x_j^2 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{n} y_j \\ \sum_{j=1}^{n} y_j x_j \end{pmatrix}.$$

Ejemplo 63. *Se tiene los siguientes datos de un proceso físico*

j	*1*	*2*	*3*	*4*	*5*
x_j	*1*	*7*	*8*	*12*	*15*
y_j	*9*	*53*	*63*	*86*	*110*

Se sabe que el proceso físico tiene, en teoria, la forma $y(x) = p + qx$. Determina p y q usando mínimos cuadrados.

Calculamos

$$\sum_{j=1}^{5} x_j = 1 + 7 + 8 + 12 + 15 = 43,$$
$$\sum_{j=1}^{5} x_j^2 = 1^2 + 7^2 + 8^2 + 12^2 + 15^2 = 483,$$
$$\sum_{j=1}^{5} x_j y_j = 1 \cdot 9 + 7 \cdot 53 + 8 \cdot 63 + 12 \cdot 86 + 15 \cdot 110 = 3556,$$
$$\sum_{j=1}^{5} y_j = 9 + 53 + 63 + 86 + 110 = 321.$$

Para calcular p y q tenemos que resolver entonces el sistema

$$\begin{pmatrix} 5 & 43 \\ 43 & 483 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix} = \begin{pmatrix} 321 \\ 3556 \end{pmatrix}.$$

La solución del sistema es

$$p = 3.0124, \quad q = 7.1148.$$

□

El ejemplo de arriba es un caso especial del problema de calcular una solución de mínimos cuadrados $x \in \mathbb{R}^n$ de un sistema lineal $Ax = b$ con $A \in \mathbb{R}^{m \times n}$ y $m > n$. Vamos a aplicar la misma idea y, como ya explicamos al principo del capítulo, pedir que se minimiza el error $b - Ax$ en la norma Euclidiana (su cuadrado, pero es lo mismo)

$$\|b - Ax\|_2^2 = \sum_{j=1}^{m} (b_j - (Ax)_j)^2.$$

Aunque uno puede usar cualquier norma, la norma Euclidiana es muy útil en este contexto dado que esta asociado con un *producto interno*, $\|x\|_2^2 = x^\top x$. Tenemos

$$\|b - Ax\|_2^2 = (b - Ax)^\top \cdot (b - Ax) = b^\top b - 2x^\top A^\top b + x^\top A^\top Ax.$$

Igualando a zero las derivadas parciales con respecto a x_j de la última expresión, obtenemos

$$0 = 2A^\top Ax - 2A^\top b.$$

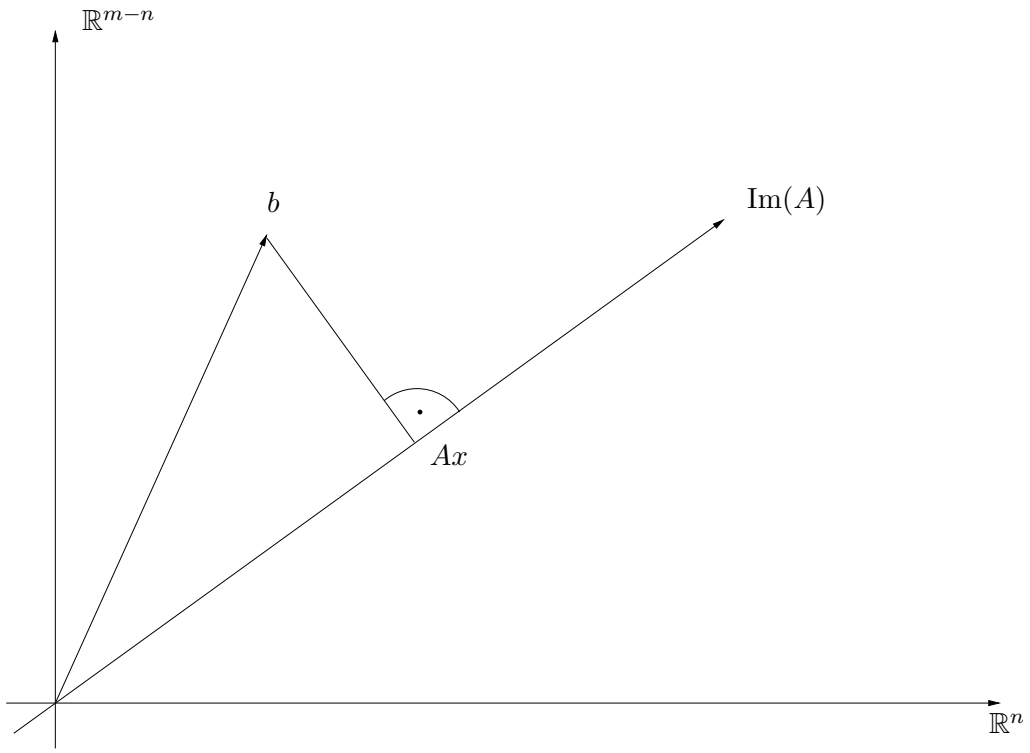
Esto nos lleva al siguiente resultado.

Teorema 64. *Sea $A \in \mathbb{R}^{m\times n}$ con $m > n$ y rango n, y sea $b \in \mathbb{R}^m$. Entonces hay único $x \in \mathbb{R}^n$ que minimiza $\|b - Ax\|_2$, y este x es la única solución del sistema*

$$A^\top Ax = A^\top b.$$

□

Si $A \in \mathbb{R}^{m\times n}$, entonces $A^\top A \in \mathbb{R}^{n\times n}$ es cuadrada. En la practica, para n grande no se resuelve el sistema $A^\top Ax = A^\top b$, dado que en general este problema tiene muy mala condición comparado con el problema original. Existen varios métodos para evitar un aumento en condición, pero son parte de un curso mas avanzado de análisis numérico. Las ecuaciones $A^\top Ax = A^\top b$ se llaman *ecuaciones normales*. En la figura 6.1 se visualiza el caso donde b no es elemento de $\mathrm{Im}(A)$. Obviamtente, para que $\|b - Ax\|_2$ se minimiza, el error $b - Ax$ tiene que ser ortogonal a $\mathrm{Im}(A)$. Dado que $\mathrm{Im}(A)$ es el espacio generado por las columnas de A, la ortogonalidad de $b - Ax$ a $\mathrm{Im}(A)$ se puede escribir como $A^\top(b - Ax) = 0$. Eso explica el nombre *ecuaciones normales*.



Para que $\|b - Ax\|_2$ se minimiza, el error $b - Ax$ tiene que ser ortogonal al $\mathrm{Im}(A)$.

Finalmente, para relacionar el ejemplo del principio de esta sección con la notación más general,

recordamos (6.2) y explicitamos las ecuaciones normales,

$$A^\top A = \begin{pmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_n \end{pmatrix} \cdot \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} = \begin{pmatrix} n & \sum_{j=1}^n x_j \\ \sum_{j=1}^n x_j & \sum_{j=1}^n x_j^2 \end{pmatrix},$$

$$A^\top b = \begin{pmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_n \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^n y_j \\ \sum_{j=1}^n y_j x_j \end{pmatrix},$$

lo mismo que obtuvimos antes.

Ajustar un modelo

$$y_{a_1,\ldots,a_n}(x),$$

que depende de parametros a_j, $j = 1, \ldots, n$ a un conjunto de puntos (x_j, y_j) usando mínimos cuadrados resulta en un sistema lineal (las ecuaciones normales) si y solo si el modelo depende *linealmente* de los parametros, es decir, si

$$y_{a_1,\ldots,a_n}(x) + y_{b_1,\ldots,b_n}(x) = y_{a_1+b_1,\ldots,a_n+b_n}(x),$$
$$c \cdot y_{a_1,\ldots,a_n}(x) = y_{c\cdot a_1,\ldots,c\cdot a_n}(x).$$

Ejemplo 65. *Verifique que el modelo* $y_{p,q}(x) = p + q\frac{1}{x}$ *es lineal en sus argumentos y ajustelo a los datos* $(1, 3.1), (2, 1.9), (4, 1.57), (8, 1.22)$ *usando mínimos cuadrados.*

El modelo es lineal en sus parametros por

$$y_{p_1,q_1}(x) + y_{p_2,q_2}(x) = p_1 + q_1\frac{1}{x} + p_2 + q_2\frac{1}{x} = (p_1 + p_2) + (q_1 + q_2)\frac{1}{x} = y_{p_1+p_2,q_1+q_2}(x),$$

la otra condición sale de la misma manera. La matriz A *y el vector* b *tienen entonces la forma*

$$A = \begin{pmatrix} 1 & \frac{1}{1} \\ 1 & \frac{1}{2} \\ 1 & \frac{1}{4} \\ 1 & \frac{1}{8} \end{pmatrix}, \quad b = \begin{pmatrix} 3.1 \\ 1.9 \\ 1.57 \\ 1.22 \end{pmatrix}$$

Las ecuaciones normales son

$$A^\top A = \begin{pmatrix} 4 & 1.875 \\ 1.875 & 1.3281 \end{pmatrix}, \quad A^\top b = \begin{pmatrix} 7.79 \\ 4.595 \end{pmatrix},$$

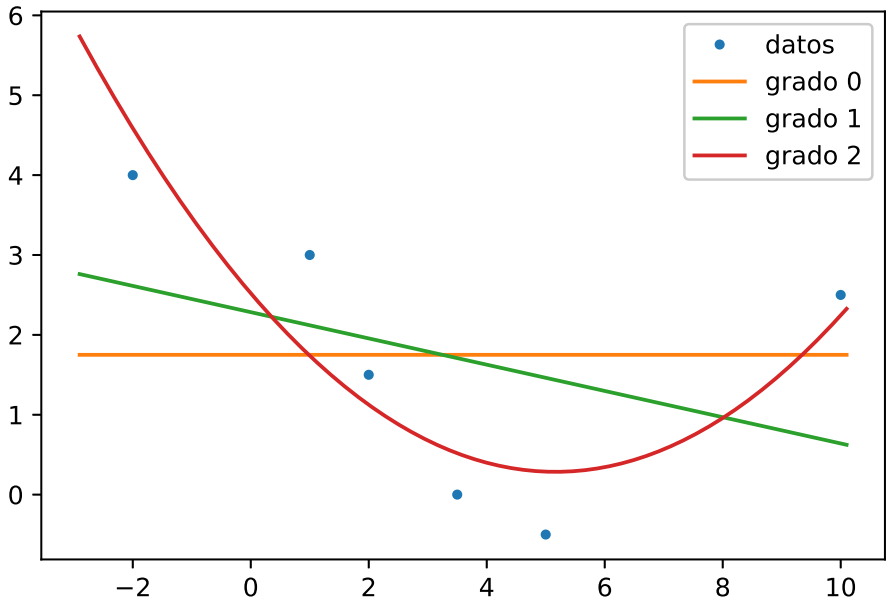
y la solución es

$$x = \begin{pmatrix} p \\ q \end{pmatrix} = \begin{pmatrix} 0.963 \\ 2.1002 \end{pmatrix}.$$

Para calcular un polinomio solución de un pŕoblema de mínimios cuadrados, podemos usar la función *polyfit*. Esta función ya la usamos en el contexto de interpolación polinomial. La función *polyfit* requiere como datos k puntos en el plano y el grado n del polinomio. Si $k = n + 1$, entonces el resultado de *polyfit* será el polinomio interpolante. Si $n < k-1$, entonces el resultado de *polyfit* será el polinomio que se ajusta a los datos en el sentido de mínimos cuadrados.

```
In [83]: # 6 datos
         x = np.array([-2.0, 1.0, 2.0, 3.5, 5.0, 10.0 ])
         y = np.array([4.0, 3.0, 1.5, 0.0, -0.5, 2.5])
```

```
# mínimos cuadrados con polinomio de grados 0,1,2
p0 = np.polyfit(x, y, 0)
p1 = np.polyfit(x, y, 1)
p2 = np.polyfit(x, y, 2)

# transformar a polinomios
p0 = np.poly1d(p0)
p1 = np.poly1d(p1)
p2 = np.poly1d(p2)

# visualizar los datos y el polinomio
import matplotlib.pyplot as plt

xp = np.linspace(-2.9, 10.1, 1000)
plt.plot(x,y,'.',xp,p0(xp),xp,p1(xp),xp,p2(xp))
plt.legend(["datos","grado 0","grado 1","grado 2"])
plt.savefig("pminquad1.eps")
```



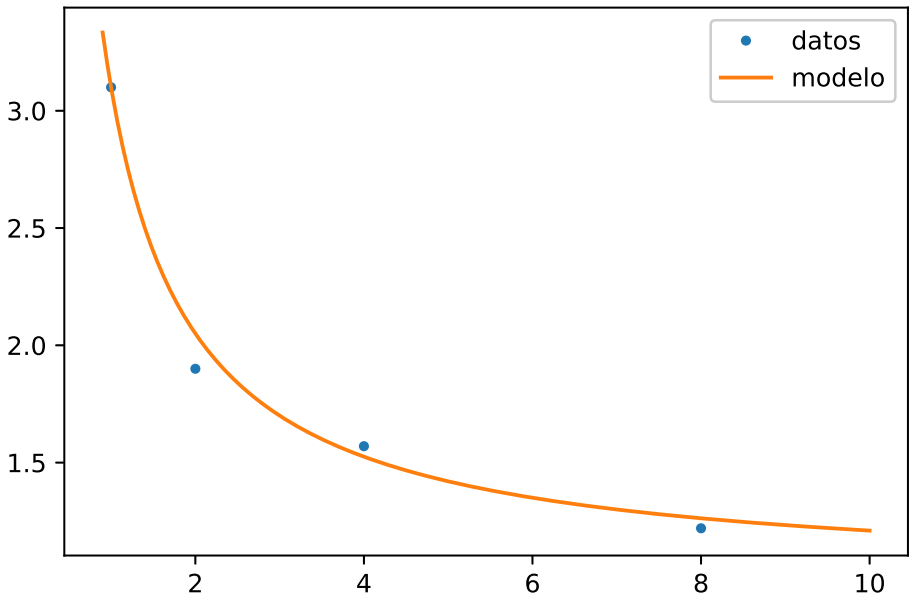
La teoría de este capítulo nos permite también ajustar un modelo general a un conjunto de datos en Python. En ejemplo 65 ajustamos una función de la forma

$$y(x) = p + q\frac{1}{x}$$

a unos datos (x_j, y_j), $j = 0, \ldots, n$. Es decir, estamos buscando parámetros p y q que minimizan la norma euclidana $\|b - Az\|_2$, donde

$$z = \begin{pmatrix} p \\ q \end{pmatrix}, \qquad b = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}, \qquad A = \begin{pmatrix} 1 & \frac{1}{x_0} \\ 1 & \frac{1}{x_1} \\ \vdots & \vdots \\ 1 & \frac{1}{x_n} \end{pmatrix}.$$

```
In [84]: def myLS(x,y):
             n = x.shape[0]
             b = np.zeros(n)
             A = np.zeros([n,2])

             for j in range(0,n):
                 b[j] = y[j]
                 A[j,0] = 1
                 A[j,1] = 1/x[j]

             # resolver ecuaciones normales
             z = np.linalg.solve(A.T@A,A.T@b)

             return z

In [85]: x = np.array([1.0, 2.0, 4.0, 8.0])
         y = np.array([3.1, 1.9, 1.57, 1.22])
         z = myLS(x,y)

         xp = np.linspace(0.9, 10, 1000)
         yp = 1 + z[1]/xp
         plt.plot(x,y,'.',xp,yp)
         plt.legend(["datos","modelo"])
         plt.savefig("pminquad2.eps")
```



6.2 Modelos no lineales

En el momento que un modelo no depende linealmente de sus parametros, tendremos que resolver un sistema no lineal para encontrar una solución de mínimos cuadrados. Sin embargo, a veces podemos transformar el modelo a uno que depende linealmente de sus parametros.

1. Nuestro objetivo es ajusta un modelo de la forma $y(x) = ae^{bx}$ con parametros $a, b \in \mathbb{R}$, a un conjunto de puntos (x_j, y_j), $j = 1, \ldots, m$. Notamos que y no depende linealmente a sus parametros (de hecho, no depende linealmente de b). Aplicamos el logaritmo al modelo y obtenemos

   $$\ln(y(x)) = \ln(ae^{bx}) = \ln(a) + bx.$$

   Introduciremos el cambio de variables

   $$\widetilde{y} = \ln(y), \quad \widetilde{a} = \ln(a), \quad \widetilde{b} = b,$$

   Entonces podemos ajustar la curva $\widetilde{y}(x) = \widetilde{a} + \widetilde{b}x$ a los puntos $(x_j, \widetilde{y}_j) = (x_j, \ln(y_j))$ usando mínimos cuadrados. Despues calculamos

   $$a = e^{\widetilde{a}}, \quad b = \widetilde{b}.$$

2. Nuestro objetivo es ajustar un modelo de la forma

$$y(x) = \frac{p}{q+x} \tag{6.3}$$

con parametros $p, q \in \mathbb{R}$ a un conjunto de puntos (x_j, y_j), $j = 1, \ldots, m$. Notamos que y no depende linealmente a sus parametros (de hecho, no depende linealmente de q). Escribmos (6.3) como

$$y(x) = \frac{p}{q} - y(x)x\frac{1}{q}.$$

Introduciremos el cambio de variables

$$\widetilde{x} = y(x)x, \quad \widetilde{p} = \frac{p}{q}, \quad \widetilde{q} = \frac{1}{q},$$

entonces el modelo

$$y = \widetilde{p} - \widetilde{q}\widetilde{x}$$

es lineal en sus parametros y lo podemos ajustar a los puntos $(\widetilde{x}_j, y_j) = (y_j x_j, y_j)$.

Chapter 7

Integración numérica

El objetivo de integración numérica es calcular la integral definida

$$I(f) = \int_a^b f(x)\,dx.$$

Por el Teorema fundamental de cálculo será suficiente calcular una primitiva de f. En la practica eso no sirve mucho, pues, puede ser difícil calcular la primitiva, o puede ser que no tenemos una formula explicita para f. Aún asi, notamos que estamos buscando solamente el valor de la integral definida, es decir, *un número*. Una integral definida es un límite de sumas, y por lo tanto una primera idea será aproximar la integral por una suma finita de la forma

$$Q_a^b(f) = \sum_{j=0}^{n} w_j f(x_j).$$

El Q_a^b se llama *regla de integración numérica*. Los w_j se llaman *pesos* y los x_j se llaman *nodos* de la regla. En general el error no es zero, $|I(f) - Q_a^b(f)| \neq 0$. La idea es elegir los nodos y los pesos para obtener el mejor error con un mínimo de costo computacional. El costo computacional para integración numérica se mide en números de evaluaciónes de f. Por ejemplo, para la regla de integración Q_a^b, el costo es $n+1$. La integración numérica que presentaremos se basa en interpolación numérica: se calcula el polinomio interpolante p_n de f en los puntos x_j, y se define $Q_a^b(f) = \int_a^b p_n(x)\,dx$. En la práctica, el polinomio interpolante p_n no se calcula explicitamente. Dado que

$$p_n(x) = \sum_{j=0}^{n} f(x_j) L_j(x),$$

donde L_j son los polinomios de Lagrange asociados a los puntos $x_0, \ldots x_n$, obtenemos

$$Q_a^b(f) = \int_a^b p_n(x)\,dx = \sum_{j=0}^{n} f(x_j) \int_a^b L_j(x)\,dx, \tag{7.1}$$

es decir, los pesos son $w_j = \int_a^b L_j(x)\,dx$. Reglas de integración numérica de esta forma, con nodos x_j *equiespaciados* se llaman *reglas de Newton-Cotes*, y los discutiremos con mas detalles en la sección 7.2.

7.1 Conceptos básicos

Grado de exactitud: Consideramos la regla Q_a^b definida en (7.1). Ya sabemos que si $f \in \mathbb{P}_n$, entonces $p_n = f$, y por lo tanto $Q_a^b(f) = \int_a^b f(x)\,dx$. Eso nos lleva a la próxima definición.

Definición 66. Se dice que una regla Q_a^b tiene grado de exactitud m si

$$Q_a^b(p_m) = I(p_m) \quad \text{para todos los polinomios } p_m \in \mathbb{P}_m.$$

□

Es decir, la regla (7.1) tiene grado de exactitud por lo menos n. De hecho, si elegimos bien los nodos x_j, podemos lograr un grado mas alto de exactitud. Eso los discutimos en detalle en la sección 7.4.

Regla de sustitución: Es suficiente definir una regla de integración numérica sobre un intervalo fijo, por ejemplo $[0,1]$ o $[-1,1]$. Sea una aproximación a la integral $\int_0^1 \hat{f}(x)\,dx$ dada por la regla

$$Q_0^1(\hat{f}) = \sum_{j=0}^{n} \hat{w}_j \hat{f}(\hat{x}_j)$$

con nodos $\hat{x}_j$ y pesos $\hat{w}_j$. Usando la sustitución

$$\int_a^b f(x)\,dx = (b-a)\int_0^1 f(a+\hat{x}(b-a))\,d\hat{x}$$

obtenemos una regla de integración numérica para aproximar $\int_a^b f(x)\,dx$,

$$Q_a^b(f) = \sum_{j=0}^{n} (b-a)\hat{w}_j f\left(a+\hat{x}_j(b-a)\right), \tag{7.2}$$

con nodos $a+\hat{x}_j(b-a)$ y pesos $(b-a)\hat{w}_j$.

Reglas compuestas: Podemos descomponer el intervalo $[a,b]$ en subintervalos

$$a = x_0 < x_1 < \cdots < x_n = b$$

y escribir

$$\int_a^b f(x)\,dx = \sum_{j=0}^{n-1} \int_{x_j}^{x_{j+1}} f(x)\,dx.$$

Para cada subintervalo $[x_j, x_{j+1}]$ podemos usar una regla $Q_{x_j}^{x_{j+1}}$ y definir una regla compuesta

$$Q_a^b(f) = \sum_{j=0}^{n-1} Q_{x_j}^{x_{j+1}}(f).$$

Eso lo discutiremos en detalle en la sección 7.3

7.2 Reglas simples de Newton-Cotes

Como ya explicamos al principio del capítulo, la idea para aproximar $\int_a^b f(x)\,dx$ es reemplazar la función a integrar f por su polinomio interpolante $p \in \mathbb{P}_n$ en ciertos nodos $x_0 < x_1 < \cdots < x_n$ e integrar el polinomio interpolante, cf. (7.1). Si los nodos x_j son *equiespaciados*, es decir, la distancia entre dos nodos $x_{j+1} - x_j$ es constante, entonces las reglas se llaman *reglas de Newton-Cotes*. También hemos visto que es suficiente definir primero una regla en un intervalo fijo y despues aplicar la regla de sustitución. Mencionamos un par de reglas de Newton Cotes y identidades de error.

1. **regla del punto medio en [0,1]**, $n = 0$ y $x_0 = \frac{1}{2}$, entonces $L_0(x) = 1$, y asi

   $$w_0 = \int_0^1 L_0(x)\,dx = 1.$$

   Es decir,

   $$Q_0^1(f) = f(\frac{1}{2}).$$

   Aplicando la sustitución (7.2), concluimos que para cualquier intervalo $[a, b]$, la regla es

   $$Q_a^b(f) = (b-a)f(\frac{a+b}{2}).$$

   El error de la regla es

   $$Q_a^b(f) - \int_a^b f(x)\,dx = f^{(2)}(\xi)\frac{1}{3}\left(\frac{b-a}{2}\right)^3,$$

   donde $\xi \in (a, b)$. De la construcción ya sabemos que la regla es exacta para polinomio de grado 0. De hecho, por simetria, la formula es exacta para polinimios de grado 1, lo que se refleja en la formula para el error.

2. **regla trapezoidal en [0,1]**, $n = 1$ y $x_0 = 0$ y $x_1 = 1$, entonces $L_0(x) = 1 - x$ y $L_1(x) = 0$, y asi

   $$w_0 = \int_0^1 L_0(x)\,dx = \frac{1}{2}, \quad w_1 = \int_0^1 L_1(x)\,dx = \frac{1}{2}.$$

Es decir,

$$Q_0^1(f) = \frac{1}{2}f(0) + \frac{1}{2}f(1).$$

Aplicando la sustitución (7.2), concluimos que para cualquier intervalo $[a, b]$, la regla es

$$Q_a^b(f) = \frac{b-a}{2}\left(f(a) + f(b)\right).$$

El error de la regla es

$$Q_a^b(f) - \int_a^b f(x)\,dx = f^{(2)}(\xi)\frac{(b-a)^3}{12},$$

donde $\xi \in (a, b)$. De la construcción ya sabemos que la regla es exacta para polinomio de grado 1, lo que también se refleja en la formula para el error.

3. **regla de Simpson en [0,1]**, $n = 2$ y $x_0 = 0$, $x_1 = \frac{1}{2}$, y $x_2 = 1$, entonces

$$L_0(x) = 2(x - 1/2)(x - 1), \quad L_1(x) = -4x(x - 1), \quad L_2(x) = 2x(x - 1/2).$$

Calculamos

$$w_0 = \int_0^1 L_0(x)\,dx = \frac{1}{6}, \quad w_1 = \int_0^1 L_1(x)\,dx = \frac{2}{3}, \quad w_2 = \int_0^1 L_2(x)\,dx = \frac{1}{6}.$$

Es decir,

$$Q(f) = \frac{1}{6}f(0) + \frac{2}{3}f(\frac{1}{2}) + \frac{1}{6}f(1).$$

Aplicando la sustitución (7.2), concluimos que para cualquier intervalo $[a, b]$, la regla es

$$Q_a^b(f) = \frac{b-a}{6}\left(f(a) + 4f(\frac{a+b}{2}) + f(b)\right).$$

El error de la regla es

$$Q_a^b(f) - \int_a^b f(x)\,dx = f^{(4)}(\xi)\frac{1}{90}\left(\frac{b-a}{2}\right)^5,$$

donde $\xi \in (a, b)$. De la construcción ya sabemos que la regla es exacta para polinomio de grado 2. De hecho, por simetria, la formula es exacta para polinimios de grado 3, lo que se refleja en la formula para el error.

4. **regla de 3/8 en [0,1]**, $n = 3$. En este caso llegaremos a

$$Q(f) = \frac{1}{8}f(0) + \frac{3}{8}f(1/3) + \frac{3/8}{f}(2/3) + \frac{1}{8}f(1).$$

Aplicando la sustitución (7.2), concluimos que para cualquier intervalo $[a, b]$, la regla es

$$Q_a^b(f) = \frac{b-a}{8}\left(f(a) + 3f(\frac{2a+b}{3}) + 3f(\frac{a+2b}{3}) + f(b)\right).$$

El error de la regla es

$$Q_a^b(f) - \int_a^b f(x)\,dx = f^{(4)}(\xi)\frac{3}{80}\left(\frac{b-a}{3}\right)^5,$$

donde $\xi \in (a, b)$. De la construcción ya sabemos que la regla es exacta para polinomio de grado 2, lo que también se refleja en la formula para el error.

Con lo anterior podemos implementar las reglas de punto medio, trapezoidal, y simpson sobre intervalos $[a, b]$.

```
In [2]: import numpy as np
        from matplotlib.pyplot import *

        def puntomedio(f,a,b):
            return (b-a)*f((a+b)/2)

        def trapezoidal(f,a,b):
            return (b-a)*(f(a)+f(b))/2

        def simpson(f,a,b):
            return (b-a)*(f(a)+4*f((a+b)/2)+f(b))/6
```

Vamos a comparar los tres métodos con respecto al error. Vamos a usar las tres reglas para integrar numéricamente

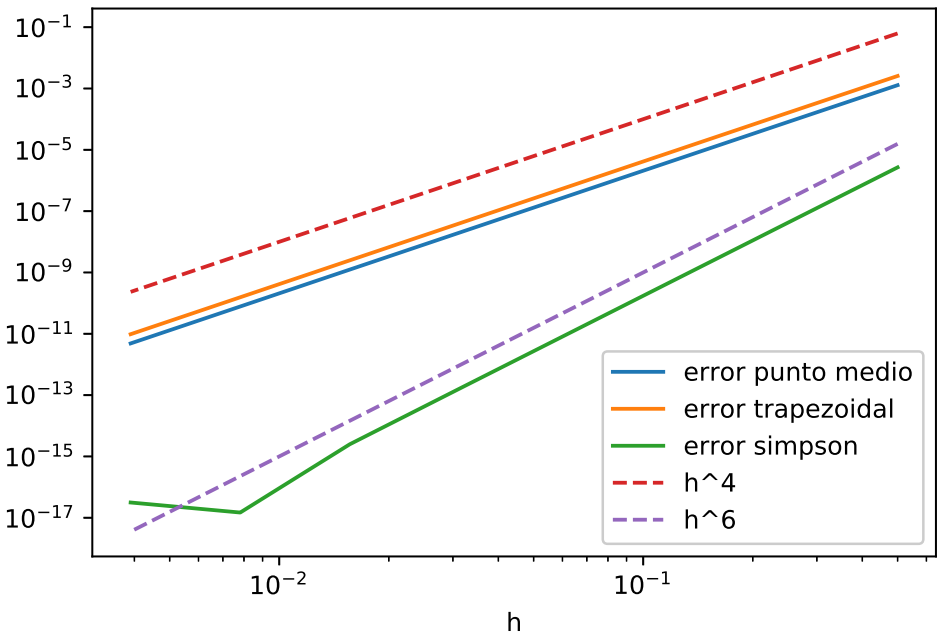
$$\int_0^h sin(x)\,dx$$

y visualizar la convergencia con respecto a h en un plot doble logarítmico. Vamos a detallar lo que esperamos en el caso de la regla de punto medio. Según lo anterior, el error es

$$|Q_0^h(sin) - \int_0^h sin(x)\,dx| = \frac{|sin(\xi)|}{24}h^3$$

para un $\xi \in (0, h)$. Dado que $\sin(\xi) \approx \xi$ para $\xi \to 0$, esperamos entonces un orden de convegencia h^4. Lo mismo es cierto para la regla trapezoidal. Para la regla de simpson esperamos entonces un orden de convergencia de h^6.

```
In [19]: def f(x):
             return np.sin(x)

         N = 8

         h = np.zeros(N)
         error_puntomedio = np.zeros(N)
         error_trapezoidal = np.zeros(N)
         error_simpson = np.zeros(N)

         for k in range(0,N):
             b = 2**(-k-1)
             h[k] = b
             error_puntomedio[k] = np.abs(np.cos(0)-np.cos(b) - puntomedio(f,0,b))
             error_trapezoidal[k] = np.abs(np.cos(0)-np.cos(b) - trapezoidal(f,0,b))
             error_simpson[k] = np.abs(np.cos(0)-np.cos(b) - simpson(f,0,b))

         loglog(h,error_puntomedio,h,error_trapezoidal,h,error_simpson,h,h**4,'--',h,0.001*h**6,'-
         xlabel('h')
         legend(['error punto medio','error trapezoidal','error simpson','h^4','h^6'])
         savefig('int1.eps')
```



7.3 Reglas compuestas

En teoria se puede calcular formulas de Newton-Cotes para cualquier n. Notamos que interpolación polinomial no es un proceso convergente, entones on podemos esperar que integración númerica por formulas de Newton-Cotes lo sea. Aparte de eso, para $n > 6$ se producen pesos w_j negativos, lo que tiene la consecuencia que hay funciones positivas con integral numérica negativa. En la practica se consigue la precisió necesaria usando *reglas compuestas*, es decir, partir el intervalo $[a, b]$ en pequeños subintervalos y en cada una de estos aplicar una regla fija de bajo grado como trapezoidal, Simpson, o punto medio.

Para definir los subintervalos de $[a, b]$, tomamos $m \in \mathbb{N}$, definimos $h = \frac{b-a}{m}$ y $x_j = a + jh$ para $j = 0, \ldots, m$. Fijamos un regla simple $Q^{[0,1]}_{\text{simple}}$ y la transformamos a cada subintervalo $Q^{[x_j,x_{j+1}]}_{\text{simple}}$. Dado que

$$\int_a^b f(x)\,dx = \sum_{j=0}^{m-1} \int_{x_j}^{x_{j+1}} f(x)\,dx,$$

podemos definir una regla compuesta para aproximar $\int_a^b f(x)\,dx$ por

$$Q^{[a,b]}_{\text{comp}}(f) = \sum_{j=0}^{m-1} Q^{[x_j,x_{j+1}]}_{\text{simple}}(f)$$

Si usamos las reglas simples de la última sección obtenemos la siguientes reglas compuestas.

1. **Regla punto medio compuesta:** Dado que $Q^{[x_j,x_{j+1}]}_{\text{simple}} = hf(\frac{x_j+x_{j+1}}{2})$, obtenemos

$$Q^{[a,b]}_{\text{comp}}(f) = h \sum_{j=0}^{m-1} f(\frac{x_j + x_{j+1}}{2}).$$

2. **Regla trapezoidal compuesta:** Dado que $Q^{[x_j,x_{j+1}]}_{\text{simple}} = \frac{h}{2}\left(f(x_j) + f(x_{j+1})\right)$, obtenemos

$$\begin{aligned} Q^{[a,b]}_{\text{comp}}(f) &= \frac{h}{2} \sum_{j=0}^{m-1} \left(f(x_j) + f(x_{j+1})\right) \\ &= \frac{h}{2} \left(f(x_0) + f(x_1) + f(x_1) + f(x_2) + \cdots + f(x_{m-2}) + f(x_{m-1}) + f(x_{m-1}) + f(x_m)\right) \\ &= \frac{h}{2} \left(f(x_0) + 2 \sum_{j=1}^{m-1} f(x_j) + f(x_m) \right). \end{aligned}$$

3. **Regla de Simpson compuesta:** Dado que $Q_{\text{simple}}^{[x_j,x_{j+1}]} = \frac{h}{6}\left(f(x_j) + 4f(\frac{x_j+x_{j+1}}{2}) + f(x_{j+1})\right)$, obtenemos

$$\begin{aligned} Q_{\text{comp}}^{[a,b]}(f) &= \frac{h}{6}\sum_{j=0}^{m-1}\left(f(x_j) + 4f(\frac{x_j+x_{j+1}}{2}) + f(x_{j+1})\right) \\ &= \frac{h}{6}\left(f(a) + 2\sum_{j=1}^{m-1} f(x_j) + 4\sum_{j=1}^{m-1} f(\frac{x_j+x_{j+1}}{2}) + f(b)\right). \end{aligned}$$

Ya sabemos que las reglas simples que hemos presentado (de hecho todas las reglas de Newton Cotes) satisfacen la identidad del error

$$\int_{x_j}^{x_{j+1}} f(x)\,dx - Q_{\text{simple}}^{[x_j,x_{j+1}]} = Ch^k f^{(k-1)}(\xi_j),$$

donde $C > 0$ es un número que no depende de j, y $\xi_j \in [x_j, x_{j+1}]$. Para la regla compuesta basada en Q_{simple} obtenemos entonces

$$\begin{aligned} |\int_a^b f(x)\,dx - Q_{\text{comp}}(f)| = Ch^k|\sum_{j=0}^{m-1} f^{(k-1)}(\xi_j)| &\le Ch^k m \max_{x\in[a,b]}|f^{k-1}(x)| \\ &= Ch^{k-1}(b-a)\max_{x\in[a,b]}|f^{k-1}(x)|, \end{aligned}$$

donde usamos $mh = b-a$ en la última desigualdad. Por lo tanto, obtenemos las siguientes cotas para el error.

1. **Regla punto medio compuesta:**

$$|\int_a^b f(x)\,dx - Q_{\text{comp}}(f)| \le \frac{1}{24}h^2(b-a)\max_{x\in[a,b]}|f^{(2)}(x)|.$$

2. **Regla trapezoidal compuesta:**

$$|\int_a^b f(x)\,dx - Q_{\text{comp}}(f)| \le \frac{1}{12}h^2(b-a)\max_{x\in[a,b]}|f^{(2)}(x)|.$$

3. **Regla de Simpson compuesta:**

$$|\int_a^b f(x)\,dx - Q_{\text{comp}}(f)| \le \frac{1}{2880}h^4(b-a)\max_{x\in[a,b]}|f^{(4)}(x)|.$$

7.4 Reglas simples de Gauss

Al determinar numéricamente una integral, se busco cierto error con la menor cantidad de evaluaciones de la función a integrar. Se espera que reglas con un alto grado de exactitud son útiles en este contexto. Por unicidad del polinomio interpolante, una regla de Newton-Cotes con $n+1$ nodos

$$Q^{[a,b]}(f) := \int_a^b p_n(x)\,dx = \sum_{j=0}^{n} f(x_j)w_j$$

tiene grado de exactitud por lo menos n. Algunas reglas de Newton-Cotes, como la de Simpson o de punto medio, tienen grado de exactitud aún $n+1$. Entones surge la siguiente pregunta: *¿cual es el grado de exactitud mas alto que podemos obtener con una regla de integración numérica con $n+1$ nodos?* El siguiente resultado nos indica una cota superior.

Lema 67. *Sea $Q^{[a,b]}(f) = \sum_{j=0}^{n} f(x_j)w_j$ una regla de integración numérica con $n+1$ puntos para la aproximación de la integral $\int_a^b f(x)\,dx$. Entonces existe un polinomio $p \in \mathbb{P}_{2n+2}$ tal que*

$$Q^{[a,b]}(p) \neq \int_a^b p(x)\,dx.$$

Proof. Sea

$$p(x) := \prod_{j=0}^{n}(x - x_j)^2.$$

Notamos qua las raizes de p son justamente los nodos de la regla $Q^{[a,b]}$. Por lo tanto,

$$\int_a^b p(x)\,dx > 0, \quad \text{pero} \quad Q^{[a,b]}(p) = 0.$$

□

El último resultado dice entonces que una regla de $n+1$ nodos nunca puede tener un grado de exactitud $2n+2$. Sin embargo, el grado de exactitud $2n+1$ es posible. Una primera heurística es la siguiente: En una regla de $n+1$ nodos tenemos $2n+2$ grados de libertad: $n+1$ nodos y $n+1$ pesos. Si queremos integrar de manera exacta polinomios de grado $2n+1$, entonces son $2n+2$ ecuaciones. Es decir, tenemos $2n+2$ ecuaciones en $2n+2$ desconocidas. Aunque el sistema es no lineal en las desconocidas, eso es una primera indicación que si puede funcionar.

Teorema 68. *Sea $[a,b]$ un intervalo y $n \geq 0$. Entonces existen nodos $x_0, \ldots, x_n \in (a,b)$ y pesos dados por $w_j := \int_a^b L_j(x)\,dx$, tal que*

1. *$w_j \geq 0$ para $j = 0, \ldots, n$,*

n	nodos	pesos
0	0	2
1	$-1/\sqrt{3}$, $1/\sqrt{3}$	1, 1
2	$-\sqrt{3/5}$, 0, $\sqrt{3/5}$	5/9, 8/9, 5/9

reglas de Gauss para $n = 0, 1, 2$ in $[-1, 1]$.

2. *la regla de integración numérica*

$$Q^{[a,b]}(f) := \sum_{j=0}^{n} f(x_j) w_j$$

tiene grado de exactitud $2n + 1$,

3. *la regla* $Q^{[a,b]}$ *definida en el punto anterior es la única regla con* $n + 1$ *nodos y grado de exactitud* $2n + 1$.

Las reglas del último resultado se llaman *reglas de Gauss*. Para calcular los nodos de una regla de Gauss se usa la teoria del polinomios ortogonales. En la tabla 7.1 presentamos solamente las primeras reglas de Gauss en el intevalo $[-1, 1]$.

Ejercicio 69. *Verifique que la regla de Gauss con 2 puntos es exacta para polinomios de grado* 3.

Por linealidad de la integral y de la regla numérica, es suficiente verificar que se integran de manera exacta todos los elementos de una base de $\mathbb{P}_3$. *Usamos la base de monomios* $m^k(x) = x^k$, $k = 0, 3$. *Calculamos*

$$\begin{aligned}
\int_{-1}^{1} m^0(x), dx = 2, \quad Q(m^0) = 2, \\
\int_{-1}^{1} m^1(x), dx = 0, \quad Q(m^1) = 0, \\
\int_{-1}^{1} m^2(x), dx = 2/3, \quad Q(m^2) = 2/3, \\
\int_{-1}^{1} m^3(x), dx = 0, \quad Q(m^0) = 0.
\end{aligned}$$

□

7.5 Integración adaptativa

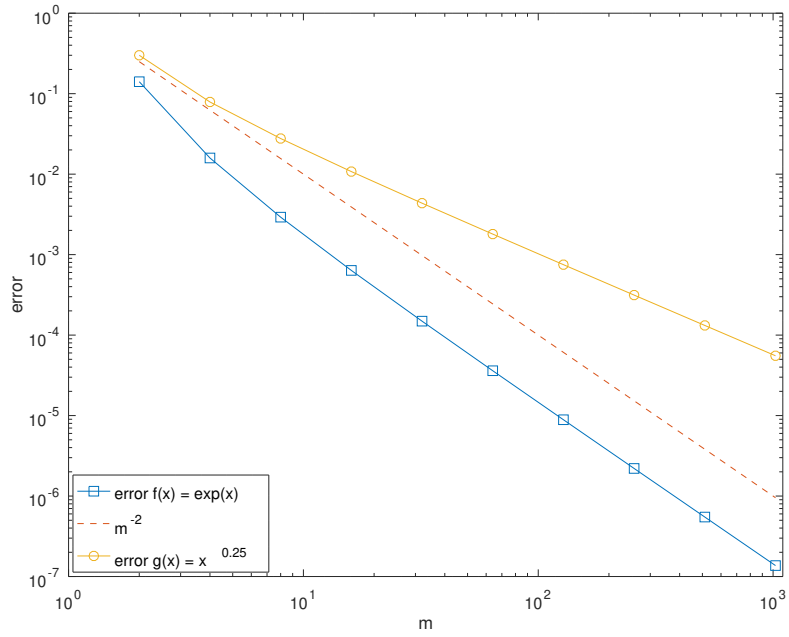
Hemos visto los ordenes de convergencia de varias reglas compuestas. Por ejemplo, para la regla trapezoidal compuesta $Q(\cdot)$ y subintervalos uniformes $x_j = a + jh$, $h = (b-a)/m$,

$$|\int_a^b f(x)\,dx - Q(f)| \leq \frac{1}{12}h^2(b-a)\max_{x\in[a,b]}|f^{(2)}(x)| = \frac{1}{m^2}\frac{(b-a)^3}{12}\max_{x\in[a,b]}|f^{(2)}(x)|,$$

Si usamos el número m de subintervalos como una medida del costo computacional, entonces vemos que el error converge como m^{-2}, bajo la condición que $f \in C^2([a,b])$. Esta condición es esencial: al integrar funciones que no tienen esta regularidad, el método pierde orden de convergencia. Por ejemplo, usaremos la regla trapezoidal compuesta para aproximar las integrales

$$\int_0^1 e^x\,dx \quad \text{y} \quad \int_0^1 x^{0.25}\,dx.$$

En figure 7.1 visualizamos los ordenes de convergencia en un plot doble logarítmico de error sobre m.



Convergencia de la regla trapezoidal compuesta con subintervalos uniformes.

Dada que $f(x) = e^x$ cumple con $f \in C^2([a,b])$, observamos el orden indicado m^{-2}. Sin embargo, para $g(x) \in x^{0.25}$ tenemos $g \notin C^2([a,b])$, y observamos un orden reducido. Sin embargo, la función g tiene cierta estructura: Solo su comportamiento en $x = 0$ determine su falta de regularidad,

afuera del origen, la función g es suave. Para recuperar el orden de convergencia optimal m^{-2} en este caso, tenemos que distribuir de manera adecuada los recursos computacionales. Dado que la función g tiene una singularidad en 0, tendrá sentido condensar los recursos entorno al origen. En otras palabras, vamos tener que usar subintervalos mas pequeños entorno al origen y subintervalos mas grandes en el resto del intervalo. Elegir manualmente donde y como distribuir los recursos computacionales requiere un conocimiento a priori de la función a integrar, y mucha experiencia del usuario del método. Será mejor tener un automatismo, es decir, un algoritmo que decide de manera automatica donde y como distribuir los recursos computacionales. Estos algoritmos se llaman **algoritmos adaptativos**. Para lo que sigue, sea $0 = x_0 < \cdots < x_m = 1$ una subdivisión del intervalo $[0, 1]$, y $\Delta := \{x_0, \ldots, x_m\}$ el conjunto de nodos asociados. Con Q_Δ anotamos la regla trapezoidal compuesta asociada, es decir,

$$Q_\Delta(f) = \sum_{j=0}^{m-1} \frac{x_{j+1} - x_j}{2} \left(f(x_j) + f(x_{j+1})\right)$$

El prototipo de una version adaptativa de la regla trapezoidal compuesta será lo siguiente:

Input: Subdivisión inicial Δ_0, función f a integrar.
Define contador $\ell := 0$

(1) Calcula $Q_{\Delta_\ell}(f)$

(2) "Mide" la contribución de cada subintervalo $[x_j, x_{j+1}]$ al error

$$|\int_0^1 f(x)\,dx - Q_{\Delta_\ell}(f)|.$$

(3) Para todos los subintervalos $[x_j, x_{j+1}]$ donde el "error es grande," introduce el punto medio $(x_j + x_{j+1})/2$. Sea $\Delta_{\ell+1}$ la subdivisión que se produce de esta manera.

(4) Incrementa $\ell = \ell + 1$ y sigue con (1).

Faltar especifica como "medir" las contribuciones de cada subintervalo al error, y lo que queremos entender por un "error grande". Para medir las contribuciones de cada subintervalo al error, observamos que no tenemos acceso a la integral exacta de f. Por lo tanto, vamos a reemplazar la integral exacta por una *mejor aproximación que* Q_{Δ_ℓ}. Si $\Delta := \{x_0, \ldots, x_m\}$, entonces introduciremos en cada subintervalo el punto medio

$$\widetilde{x}_j := \frac{x_j + x_{j+1}}{2},$$

y asi obtenemos la subdivisión

$$\widetilde{\Delta}_\ell := \{x_0, \widetilde{x}_0, x_1, \widetilde{x}_1, x_2, \widetilde{x}_2, \ldots, x_{m-1}, \widetilde{x}_{m-1}, x_m\}.$$

Esperamos que $Q_{\widetilde{\Delta}_\ell}(f)$ es una mejor aproximación a la integral exacta que $Q_{\Delta_\ell}(f)$, y asi

$$|\int_0^1 f(x)\,dx - Q_{\Delta_\ell}(f)| \approx |Q_{\widetilde{\Delta}_\ell} - Q_{\Delta_\ell}(f)|.$$

Además,

$$\begin{aligned}
\int_0^1 f(x)\,dx &= \sum_{j=1}^{m-1} \int_{x_j}^{x_{j+1}} f(x)\,dx, \\
Q_{\Delta_\ell}(f) &= \sum_{j=1}^{m-1} Q^{[x_j,x_{j+1}]}_{\text{trapezoidal}}(f), \\
Q_{\widetilde{\Delta}_\ell}(f) &= \sum_{j=1}^{m-1} \left(Q^{[x_j,\widetilde{x}_j]}_{\text{trapezoidal}}(f) + Q^{[\widetilde{x}_j,x_{j+1}]}_{\text{trapezoidal}}(f) \right).
\end{aligned}$$

Por lo tanto, obtenemos

$$|\int_0^1 f(x)\,dx - Q_{\Delta_\ell}(f)| \approx \sum_{j=0}^{m-1} |Q^{[x_j,\widetilde{x}_j]}_{\text{trapezoidal}}(f) + Q^{[\widetilde{x}_j,x_{j+1}]}_{\text{trapezoidal}}(f) - Q^{[x_j,x_{j+1}]}_{\text{trapezoidal}}(f)|.$$

Es decir, el termino

$$\eta_j := |Q^{[x_j,\widetilde{x}_j]}_{\text{trapezoidal}}(f) + Q^{[\widetilde{x}_j,x_{j+1}]}_{\text{trapezoidal}}(f) - Q^{[x_j,x_{j+1}]}_{\text{trapezoidal}}(f)|$$

es una medida para la contribución del subintervalo $[x_j, x_{j+1}]$ al error. Para decidir lo que significa "error grande", vamos a introducir un parametro $\theta \in (0,1)$, y vamos a buscar un conjunto de subintervalos que contribuyen un porcentaje θ al error global $\sum_{j=0}^{m-1} \eta_j$, y que contenga la menor cantidad de subintervalos posible. En formulas, buscamos un subconjunto $M_\ell \subset \{0,\ldots,m-1\}$ con menor cardinalidad tal que

$$\sum_{j\in M_\ell} \eta_j \geq \theta \sum_{j=0}^{m-1} \eta_j.$$

Finalmente, nuestro algoritmo adaptativo queda asi:

Input: Subdivisión inicial Δ_0, función f a integrar, parametro θ.
Define contador $\ell := 0$

(1) Calcula $Q_{\Delta_\ell}(f)$

(2) Para cada subintervalo $[x_j, x_{j+1}]$ de la subdivisión actual Δ_ℓ calcula

$$\eta_j := |Q^{[x_j,\widetilde{x}_j]}_{\text{trapezoidal}}(f) + Q^{[\widetilde{x}_j,x_{j+1}]}_{\text{trapezoidal}}(f) - Q^{[x_j,x_{j+1}]}_{\text{trapezoidal}}(f)|,$$

donde $\widetilde{x}_j := \frac{x_j+x_{j+1}}{2}$.

(3) Determina un subconjunto $M_\ell \subset \{1, \ldots, m-1\}$ con menor cardinalidad tal que

$$\sum_{j\in M_\ell} \eta_j \geq \theta \sum_{j=0}^{m-1} \eta_j.$$

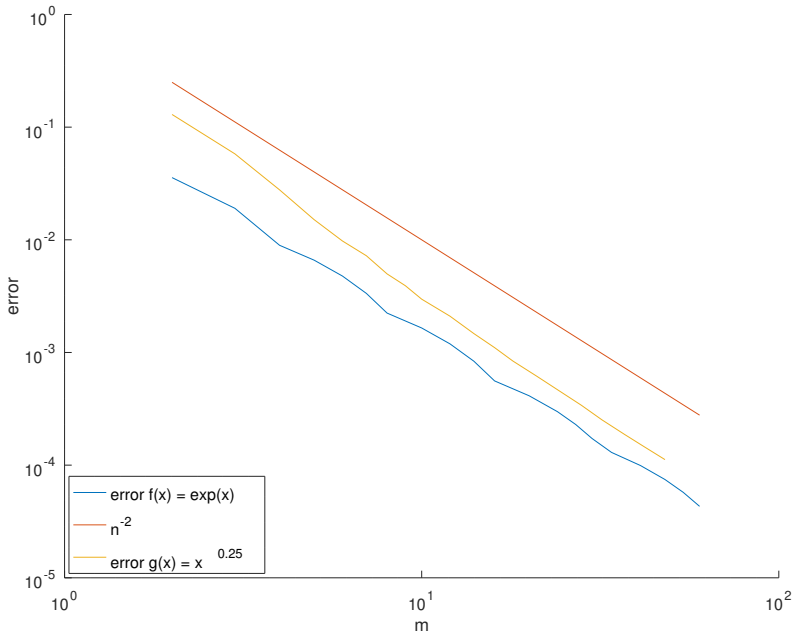
Para cada $j \in M_\ell$ introduce el punto medio $\widetilde{x}_j := \frac{x_j+x_{j+1}}{2}$ como nodo nuevo. Sea $\Delta_{\ell+1}$ la subdivisión que se produce de esta manera.

(4) Incrementa $\ell = \ell + 1$ y sigue con (1).

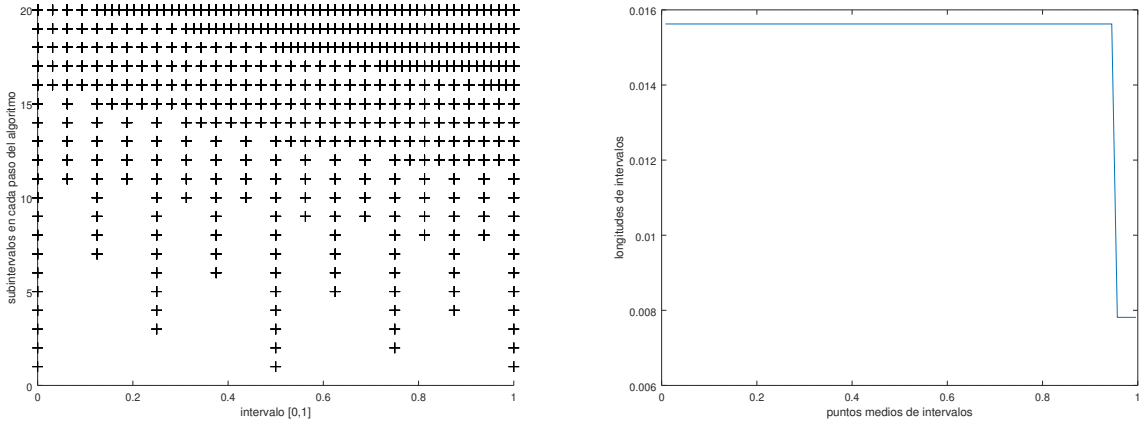
Aplicaremos el algoritmo adaptativo a las integrales de antes,

$$\int_0^1 e^x\,dx \quad \text{y} \quad \int_0^1 x^{0.25}\,dx.$$

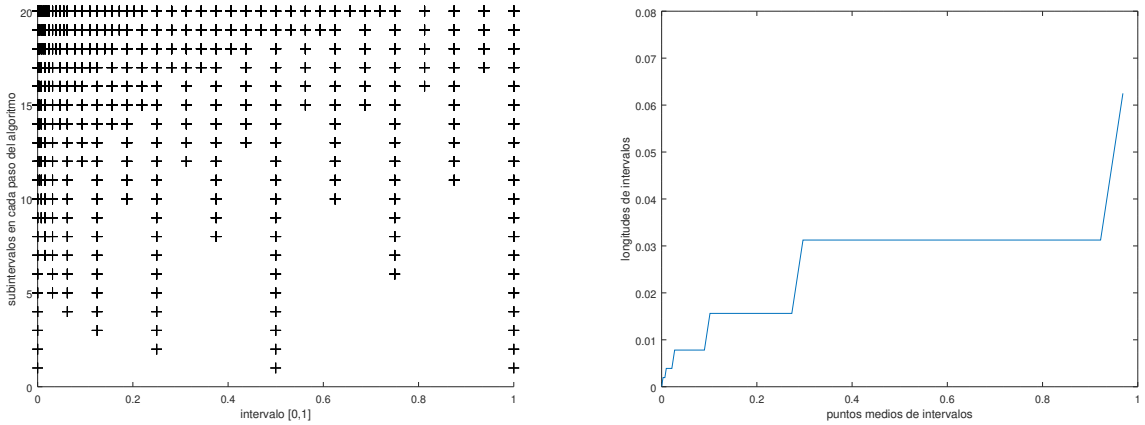
En Figure 7.2 vemos que el algoritmo adaptativo converge como m^{-2} en el caso de la función $f(x) = e^x$. También para el caso de la función $g(x) = x^{0.25}$, el algoritmo recupera la convergencia optimal m^{-2}. En Figure 7.3 visualizamos las mallas en cada paso del algoritmo tal como la longitud de los intervalos sobre sus puntos medios, en el caso de $f(x) = e^x$. Observamos que el algoritmo adaptativo determina practicamente una sucesión uniforme de mallas. Eso tiene sendito, pues, mallas uniformes ya son optimales en este caso. En Figure 7.4 visualizamos los mismos datos en el caso de $g(x) = x^{0.25}$. Observamos que el algoritmo produce malla que se condensan en 0, justamente donde g tiene la singularidad.



Convergencia de la regla trapezoidal compuesta adaptativa.



Izquierda: Mallas en cada paso del algoritmo adaptativo para $f(x) = e^x$. Derecha: longitudes de subintervalos sobre puntos medios en el último paso para $f(x) = e^x$.



Izquierda: Mallas en cada paso del algoritmo adaptativo para $f(x) = x^{0.25}$. Derecha: longitudes de subintervalos sobre puntos medios en el último paso para $f(x) = x^{0.25}$.

Chapter 8

Métodos numéricos para EDO

El objetivo del presente capítulo es presentar métodos numéricos para ecuaciones diferenciales ordinarias (EDO). Estas ecuaciones modelan, en general, procesos y sistemas que dependen del *tiempo*, y es por eso que la variable independiente se llamará t. El problema mas básico en este contexto es lo siguiente: encontrar una función real $y : \mathbb{R} \to \mathbb{R}$, que satisface la ecuacion diferencial

$$y'(t) = f(t, y(t)), \quad \text{para todo } t \in (0, T), \tag{8.1a}$$

y la condición inicial

$$y(0) = y^0. \tag{8.1b}$$

La funcón f y el valor inicial y^0 están dados. El problema de encontrar una función y que satisface (8.1a) y (8.1b) se llama *problema de valor inicial.* En la ecuación (8.1a) aparece solamente la primera derivada de y, y por eso la ecuación se llama *de primer orden.* La teoría de existencia y únicidad de soluciones al problema de valor inicial es materia de cursos basicos de matemática.

También vamos a considerar *sistemas de ecuaciones diferenciales ordinarias*, eso es, encontrar n funciones $y_j = y_j(t)$, $j = 1, \ldots, n$, que satisfacen el sistema de ecuaciones diferenciales

$$\begin{aligned} y_1'(t) &= f_1(t, y_1(t), \ldots, y_n(t)), \\ y_2'(t) &= f_2(t, y_1(t), \ldots, y_n(t)), \\ &\vdots \\ y_n'(t) &= f_n(t, y_1(t), \ldots, y_n(t)), \end{aligned} \tag{8.2a}$$

para todo $t \in (t_0, T)$, y las condiciones iniciales

$$\begin{aligned} y_1(0) &= y_1^0, \\ y_2(0) &= y_2^0, \\ &\vdots \\ y_n(0) &= y_n^0. \end{aligned} \tag{8.2b}$$

Introducimos los objetos vectoriales

$$y(t) := \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{pmatrix}, \quad f(t,y) = \begin{pmatrix} f_1(t, y_1(t), \ldots, y_n(t)), \\ f_2(t, y_1(t), \ldots, y_n(t)), \\ \vdots \\ f_n(t, y_1(t), \ldots, y_n(t)), \end{pmatrix}, \quad y_0 := \begin{pmatrix} y_1^0 \\ y_2^0 \\ \vdots \\ y_n^0 \end{pmatrix},$$

entonces podemos escribir el problema (8.2) en la forma (8.1).

1. La función $y(t) = (1-t^2)^{-1}$ es solución de

$$\begin{aligned} y'(t) &= 2ty^2(t), \quad t \geq 0, \\ y(0) &= 1. \end{aligned}$$

   Notamos que $T = 1$ en este caso.

2. La función

$$y(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = \begin{pmatrix} 2\cos(t) \\ cos(t) + \sin(t) \end{pmatrix}$$

   es solución de

$$\begin{aligned} y'(t) &= \begin{pmatrix} y_1'(t) \\ y_2'(t) \end{pmatrix} = \begin{pmatrix} \frac{1}{2}y_1 - y_2 \\ 2y_1 - 2y_2 + 3\sin(t) \end{pmatrix} \\ y(0) &= \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}. \end{aligned}$$

Notamos que es facil verificar si una función dada es solución de una EDO, pero en general es difícil encontrar una solución. Por lo tanto, es necesario desarollar métodos numéricos. En un problema de valor inicial estamos buscando una función, es decir, un objeto en un espacio de dimension infinita. Vamos a determinar una aproximacion en un espacio de dimension finita a traves de reemplazar la ecuación diferencial por ecuaciones algebraicas. Este proceso se llama *discretización*, e incluye un parametro de resolución $h > 0$.

8.1 Métodos de paso simple

Para obtener un método numérico para la aproximación de (8.1) o (8.2) notamos que para un parametro $h > 0$

$$y(t+h) \approx y(t) + hy'(t) = y(t) + hf(t, y(t)).$$

Es decir,

$$y(h) \approx y^1 := y^0 + hf(0, y^0).$$

Por el mismo argumento,

$$y(2h) \approx y(h) + hy'(h) = y(h) + hf(h, y(h)).$$

No tenemos $y(h)$, pero la aproximación y^1, y por lo tanto definimos

$$y(2h) \approx y^2 := y^1 + hf(h, y^1).$$

Eso nos lleva al **método de Euler explicito:** Sea $t_0 = 0$. Dado un paso $h > 0$, calcule

$$\begin{aligned} t_{j+1} &:= t_j + h, \\ y^{j+1} &:= y^j + hf(t_j, y^j). \end{aligned} \tag{8.3}$$

Otra manera de motivar el método de Euler explicito es por el teorema fundamental de cálculo

$$y(t+h) = y(t) + \int_t^{t+h} y'(s)\, ds = y(t) + \int_t^{t+h} f(s, y(s))\, ds. \tag{8.4}$$

Si aproximamos la integral del lado derecho por

$$\int_t^{t+h} f(s, y(s))\, ds \approx hf(t, y(t)),$$

entonces obtenemos el método de Euler explicito. Si aproximamos la integral del lado derecho por

$$\int_t^{t+h} f(s, y(s))\, ds \approx hf(t+h, y(t+h)),$$

entonces llegamos a

$$y(t+h) \approx y(t) + hf(t+h, y(t+h)).$$

Eso nos lleva al **método de Euler implícito:** Sea $t_0 = 0$. Dado un paso $h > 0$, calcule

$$\begin{aligned} t_{j+1} &:= t_j + h, \\ y^{j+1} &:= y^j + hf(t_{j+1}, y^{j+1}). \end{aligned} \tag{8.5}$$

Notamos que el método (8.3) es realmente explícito en el sentido que para determinar y^{j+1} se usa solamente el valor y^j. Por el otro lado, (8.5) es un método implícito: para determinar y^{j+1} hay que resolver una ecuación o un sistema que en general es no lineal.

Si aproximamos la integral del lado derecho en (8.4) por la regla del punto medio

$$\int_t^{t+h} f(s, y(s))\, ds \approx f(t + \frac{h}{2}, y(t + \frac{h}{2})),$$

entonces $y(t + \frac{h}{2})$ no corresponde ni a y^j ni a y^{j+1}. Vamos aproximarlo entonces por Euler explícito con paso $h/2$,

$$y(t + \frac{h}{2}) \approx y(t) + \frac{h}{2} f(t, y(t)).$$

Asi llegamos al **método de Euler mejorado**

$$\begin{aligned} t_{j+1} &:= t_j + h, \\ y^{j+\frac{1}{2}} &:= y^j + \frac{h}{2} f(t_j, y^j), \\ y^{j+1} &:= y^j + h f(t_j + \frac{h}{2}, y^{j+\frac{1}{2}}). \end{aligned} \tag{8.6}$$

Ejercicio 70. *Consideramos el problema de valor inical*

$$\begin{aligned} y'(t) &= 2y(t) - 2\cos(t) - \sin(t) \\ y(0) &= 2. \end{aligned}$$

La función f en esto caso es $f(t, y) = 2y - 2\cos(t) - \sin(t)$. Vamos a aplicar uno o dos pasos de cada método mencionado.

1. ***Euler explícito:*** *Vamos a hacer dos pasos*

   $$\begin{aligned} &y^0 = 2, \\ &y^1 = y^0 + hf(t_0, y^0) = 2 + hf(0, 2) = 2 + h\,(4 - 2) = 2 + 2h, \\ &y^2 = y^1 + hf(t_1, y^1) = 2 + 2h + hf(h, 2 + 2h) = 2 + 2h + h\,(2(2 + 2h) - 2\cos(h) - \sin(h)) \end{aligned}$$

2. ***Euler implícito:*** *Vamos a hacer un paso*

   $$\begin{aligned} &y^0 = 2, \\ &y^1 = y^0 + hf(t_1, y^1) = 2 + hf(h, y^1) = 2 + h\left(2y^1 - 2\cos(h) - \sin(h)\right), \end{aligned}$$

   por lo tanto

   $$y^1 = \frac{2 - 2h\cos(h) - h\sin(h)}{2 - 2h}.$$

3. ***Euler mejorado:***

Dado que la aplicación de un método implicito requiere la solución de una ecuación, nos podemos preguntar **¿porque usar métodos implicitos?** Consideramos el problema de valor inicial

$$y'(t) = \lambda y(t),$$
$$y(0) = y_0$$

con una $\lambda < 0$. La solución exacta es obviamente $y(t) = y_0 e^{\lambda t}$, y dado que $\lambda < 0$, obtenemos

$$\lim_{t\to\infty} y(t) = 0.$$

El Euler explícito en este caso se lee $y_{j+1} = y_j + h\lambda y_j = y_j(1 + \lambda h)$, es decir, $y_j = y_0(1 + \lambda h)^j$. Por lo tanto, si $|\lambda h| > 2$, entonces $|1 + \lambda h| > 1$, y concluimos

$$\lim_{j\to\infty} |y_j| = \infty.$$

En otras palabras, el método numérico no refleja el comportamiento cualitativo de la solución. Por otro lado, el Euler implícito se lee $y_j = y_0/(1 - \lambda h)^j$ y no sufre del mismo efecto.

8.2 Ecuaciones de orden mayor y reducción a sistemas

Muchas veces un quiere resolver un problema de valor inical con una EDO de orden mayor, es decir

$$y^{(n)}(t) = f(t, y(t), y'(t), y''(t), \dots, y^{(n-1)}(t)), \tag{8.7a}$$

con las $n-1$ condiciones iniciales

$$y^{(k)}(0) = y_k^0, \quad k = 1, \dots, n-1. \tag{8.7b}$$

Para aplicar los métodos de la última sección, vamos a transforma el problema (8.7) en un sistema de n ecuaciones. Para ello, se define las n funciones

$$\begin{aligned} y_1 &= y, \\ y_2 &= y', \\ &\vdots \\ y_n &= y^{(n-1)}. \end{aligned}$$

Entonces, el problema (8.7) se transforma al siguiente sistema: encontrar las n funciones y_j, $j = 1, \dots, n$, que satisfacen el sistema de ecuaciones

$$\begin{aligned} y_1' &= y_2, \\ y_2' &= y_3, \\ &\vdots \\ y_{n-1}' &= y_n, \\ y_n'(t) &= f(t, y_1(t), y_2(t), \dots, y_n(t)), \end{aligned}$$

y las condiciones iniciales

$$y_k(0) = y_k^0, \quad k = 1, \dots, n-1.$$

Ejercicio 71. *Consideramos el problema de valor inicial*

$$\begin{aligned} y''' &= -2y'' + y' + y^3 + \sin(t), \quad t \in (0, T), \\ y(0) &= 1, \quad y'(0) = 0, \quad y''(0) = 2. \end{aligned}$$

La EDO tiene orden 3*, y se transforma en un sistema de primer orden*

$$y'(t) = \begin{pmatrix} y_1'(t) \\ y_2'(t) \\ y_3'(t) \end{pmatrix} = \begin{pmatrix} y_2(t) \\ y_3 \\ -2y_3(t) + y_2(t) + y_1^3(t) + \sin(t) \end{pmatrix} = f(t, y_1(t), y_2(t), y_3(t)) = f(t, y(t))$$

con condiciones iniciales

$$y(0) = \begin{pmatrix} y_1(0) \\ y_2(0) \\ y_3(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$$

Aplicamos un paso con Euler explicito con parametro $h > 0$:

$$y^0 = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix},$$

$$y^1 = y^0 + hf(0, y^0) = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} + h \begin{pmatrix} y_2^0 \\ y_3^0 \\ -2y_3^0 + y_2^0 + \left(y_1^0\right)^3 + \sin(0) \end{pmatrix}$$

$$= \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} + h \begin{pmatrix} 0 \\ 2 \\ -2 \cdot 2 + 0 + 1^3 + 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2h \\ 2 - 3h \end{pmatrix}.$$

8.3 Error de consistencia y convergencia

Métodos de paso simple para problemas (8.1) tienen la forma

$$y^{j+1} = \Psi(t_j, t_{j+1}, y^j),$$

donde Ψ es una función que depende del problema de valor inicial (8.1) bajo consideración. Para el método de Euler explícito, por ejemplo,

$$y^{j+1} = \Psi(t_j, t_{j+1}, y^j) = y^j + hf(t_j, y^j), \quad \text{donde } h = t_{j+1} - t_j. \tag{8.8}$$

Para el método de Euler mejorado, por ejemplo,

$$\Psi(t_j, t_{j+1}, y^j) = y^j + hf(t_j + \frac{h}{2}, y^j + \frac{h}{2} f(t_j, y^j)), \quad \text{donde } h = t_{j+1} - t_j.$$

Para métodos implícitos no es tan fácil explicitar la función Ψ. Para llegar al tiempo final T, vamos a tener que hacer aproximadamente $n \approx \frac{T}{h}$ pasos, y finalmente nos interesa como se comporta el error

$$e_h := \max_{j=0,\ldots,n} |y(t_j) - y^j|$$

con respecto al parametro de discretización h. El procedimiento (8.8) se puede entender como *un paso* de Euler explícito aplicado al problema

$$y'(t) = f(t, y(t)),$$
$$y(t_j) = y^j.$$

Eso nos lleva al concepto de *consistencia*.

Definición 72. (i) Para valores $\widetilde{t}$, $\widetilde{y}$ sea $y_{\widetilde{t},\widetilde{y}}$ la solución de

$$y'_{\widetilde{t},\widetilde{y}}(t) = f(t, y_{\widetilde{t},\widetilde{y}}(t)), \quad t \in (\widetilde{t}, T),$$
$$y_{\widetilde{t},\widetilde{y}}(\widetilde{t}) = \widetilde{y}.$$

Sea $y^1_{\widetilde{t},\widetilde{y}}$ una aproximación a $y_{\widetilde{t},\widetilde{y}}(\widetilde{t}+h)$ obtenido con un método de paso simple. El termino

$$\delta_h(\widetilde{t}, \widetilde{y}) = |y_{\widetilde{t},\widetilde{y}}(\widetilde{t}+h) - y^1_{\widetilde{t},\widetilde{y}}|$$

se llama **error de truncamiento** del método.

(ii) Sea y una solución para el problema

$$y'(t) = f(t, y(t)), \quad t \in (0, T),$$
$$y(0) = y^0.$$

Un método de paso simple se llama **consistente de orden** p para el último problema, si existe una constante $C > 0$ tal que para todo $(\widetilde{t}, \widetilde{y})$ suficientemente cerca de $(t, y(t))$ para un $t \in (0, T)$ se tiene

$$\delta_h(\widetilde{t}, \widetilde{y}) \leq Ch^{p+1}.$$

Se puede mostrar que Euler explícito e implícito son consistentes de orden 1, mientras Euler mejorado es consistente de orden 2. Podemos verificar estos ordenes de consistencia para el problema simple

$$y'(t) = y(t),$$
$$y(0) = 1.$$

Aplicamos los tres métodos de Euler con parametro $h > 0$ al problema y calculamos el error de truncamiento $\delta_h = \delta_h(0, 1) = |y(h) - y^1|$ despues de un paso. Notamos que la solución exacta es

$$y(t) = e^t = \sum_{k=0}^{\infty} \frac{t^k}{k!} = 1 + t + \frac{t^2}{2} + \frac{t^3}{6} + \dots$$

1. **Euler explícito:** Dado que $y^1 = (1 + h)$, obtenemos

$$\delta_h^{\text{expl}} = \frac{h^2}{2} + \frac{h^3}{6} + \cdots \approx h^2.$$

2. **Euler impícito:** Dado que $y^1 = 1 + hy^1$ obtenemos

$$y^1 = \frac{1}{1-h} = 1 + h + h^2 + h^3 + h^4 + \dots,$$

y por lo tanto,

$$\delta_h^{\text{impl}} = \left(h^2 + h^3 + h^4 + \dots\right) - \left(\frac{h^2}{2} + \frac{h^3}{6} + \dots\right) \approx h^2.$$

3. **Euler mejorado:** Calculamos $y^1 = 1 + h + \frac{h^2}{2}$, y por lo tanto

$$\delta_h^{\text{mejor}} = \frac{h^3}{6} + \frac{h^4}{4!} + \cdots \approx h^3.$$

El próximo resultado implica que si un método es consistente de orden p, entonces también es convergente de orden p.

Teorema 73. *Sea y una solución para el problema*

$$\begin{aligned} y'(t) &= f(t, y(t)), \quad t \in (0, T), \\ y(0) &= y^0. \end{aligned}$$

Sea $nh = T$ y

$$y^{j+1} = \Psi(t_j, t_{j+1}, y^j)$$

un método de paso simple. Si f y Ψ son suficientemente suaves y el método tiene orden de consistencia p, entonces el método es convergente de orden p, es decir,

$$e_h = \max_{j=0,\ldots,n} |y(t_j) - y^j| \leq Ch^p.$$

8.4 Métodos de Runge-Kutta

Usando el desarollo de Taylor de y, es posible obtener métodos de paso simple de cualquier orden. Sin embargo, estos métodos contenerán derivadas de la función f y por lo tanto no se aplican en la practica. Los métodos de Runge-Kutta son métodos de paso simple y orden mayor que, en vez de usar derivadas de f, usan varias evaluaciones de f para obtener un orden alto. La idea es aproximar la integral en (8.4)

$$\int_t^{t+h} f(s, y(s))\, ds \approx h \sum_{\ell=1}^{m} \gamma_\ell k_\ell$$

donde γ_ℓ son pesos adecuados y los k_ℓ son evaluaciones adecuadas de f, es decir

$$k_\ell = f(s_\ell, \widetilde{y}_\ell).$$

El número m se llama *número de etapas* del método. Es decir, un método Runge-Kutta de m etapas tiene la forma

$$y^{j+1} = y^j + h \sum_{\ell=1}^{m} \gamma_\ell k_\ell.$$

Por ejemplo, el método de Euler explícito $y^{j+1} = y^j + hf(t_j, y^j)$ es de una etapa $m = 1$, y $k_1 = f(t_j, y^j)$. El método de Euler mejorado

$$\begin{aligned} t_{j+1} &:= t_j + h, \\ y^{j+\frac{1}{2}} &:= y^j + \frac{h}{2} f(t_j, y^j), \\ y^{j+1} &:= y^j + hf(t_j + \frac{h}{2}, y^{j+\frac{1}{2}}). \end{aligned}$$

tiene dos etapas $m = 2$, y

$$\gamma_1 = 0, \quad \gamma_2 = 0, \quad k_2 = f(t_j + \frac{h}{2}, y^j + \frac{h}{2} f(t_j, y^j)).$$

La idea es elegir los pesos y las evaluaciones para obtener un método con orden de consistencia mas álto posible con menor número de etapas posibles.

Un método clásico es el **RK-4** de 4 etapas, dado por

$$\begin{aligned} k_1 &:= f(t_j, y^j), \\ k_2 &:= f(t_j + \frac{h}{2}, y^j + \frac{h}{2} k_1), \\ k_3 &:= f(t_j + \frac{h}{2}, y^j + \frac{h}{2} k_2), \\ k_4 &:= f(t_j + h, y^j + hk_3), \\ y^{j+1} &= y^j + \frac{h}{6} \left(k_1 + 2k_2 + 2k_3 + k_4 \right). \end{aligned}$$

El RK-4 es un método explícito: dado y^j, podemos calcular, en este orden, k_1, k_2, k_3, k_4, y despues y^{j+1}. El orden de consistencia de RK-4 es 4.

Ejercicio 74. *Consideramos el problema simple*

$$\begin{aligned} y'(t) &= y(t), \quad t \geq 0, \\ y(0) &= 1 \end{aligned}$$

con solución exacta $y(t) = e^t$. La función f está dada por $f(t, y) = y$. Tenemos $y^0 = 1$, y un paso con RK-4 será

$$\begin{aligned} k_1 &= f(0, 1) = 1, \\ k_2 &= f(h/2, 1 + h/2) = 1 + h/2, \\ k_3 &= f(h/2, 1 + h/2(1 + h/2)) = 1 + h/2 + h^2/4, \\ k_4 &= f(h, 1 + h(1 + h/2 + h^2/4)) = 1 + h + h^2/2 + h^3/4, \end{aligned}$$

y finalmente

$$\begin{aligned} y^1 &= 1 + \frac{h}{6}\left(1 + 2 + h + 2 + h + h^2/2 + 1 + h + h^2/2 + h^3/4\right) \\ &= 1 + h + \frac{h^2}{2} + \frac{h^3}{6} + \frac{h^4}{24}. \end{aligned}$$

Para el error de truncamiento $\delta_h(0,1)$ *obtenemos entonces*

$$\delta_h(0,1) = |y(h) - y^1| \leq \sum_{i=5}^{\infty} \frac{h^i}{i!} \leq Ch^5,$$

es decir, el método es consistente de orden 4. □

La forma general de un método de Runge-Kutta de m etapas es

$$\begin{aligned} k_i &= f(t_j + \alpha_i h, y^j + h\sum_{\ell=1}^{m} \beta_{i,\ell} k_\ell), \quad i = 1, \ldots, m, \\ y^{j+1} &= y^j + h\sum_{\ell=1}^{m} \gamma_\ell k_\ell. \end{aligned} \tag{8.9}$$

En general, se anota este método en una *tabla de Butcher* de la forma

$$\begin{array}{c|ccc} \alpha_1 & \beta_{1,1} & \cdots & \beta_{1,m} \\ \alpha_2 & \beta_{2,1} & \cdots & \beta_{2,m} \\ \vdots & \vdots & & \vdots \\ \alpha_m & \beta_{m,1} & \cdots & \beta_{m,m} \\ \hline & \gamma_1 & \cdots & \gamma_m \end{array}$$

En (8.9), cada k_i depende de todos los k_ℓ, $\ell = 1, \ldots, m$. Es decir, para obtener los k_i hay que resolver un sistema no lineal, y el método de Runge-Kutta se llama *implícito*. Si $\beta_{i,\ell} = 0$ para $\ell \geq i$, entonces el k_i depende solamente de $k_1, \ldots, k_{i-1}$, y el método es *explícito*. La tabla de Butcher de un método de Runge-Kutta explícito tiene entonces la forma

$$\begin{array}{c|ccccc} \alpha_1 & & & & & \\ \alpha_2 & \beta_{2,1} & & & & \\ \alpha_3 & \beta_{3,1} & \beta_{3,2} & & & \\ \vdots & \vdots & & \ddots & & \\ \alpha_m & \beta_{m,1} & \cdots & \cdots & \beta_{m,m-1} & \\ \hline & \gamma_1 & \gamma_2 & \cdots & \cdots & \gamma_m \end{array}$$

Por ejemplo, el Euler explícito es un método de Runge-Kutta de una etapa, y su tabla de Butcher es

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}$$

El Euler mejorado es un método de Runge-Kutta de dos etapas, y su tabla de Butcher es

$$\begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array}$$

Por otro lado, la tabla de Butcher del Euler implícito es

$$\begin{array}{c|c} 0 & 1 \\ \hline & 1 \end{array}$$

Ejercicio 75. *Determine la tabla de Butcher del RK-4.*

Ejercicio 76. *Considere el método de Heun, dado por la tabla de Butcher*

$$\begin{array}{c|cc} 0 & & \\ \frac{2}{3} & \frac{2}{3} & \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array}$$

¿Es explćito o implícito el método de Heun? Realice un paso del método al problema

$$\begin{aligned} y'(t) &= y(t), \quad t \geq 0, \\ y(0) &= 1. \end{aligned}$$

¿Cual será el orden de consistencia en este caso?