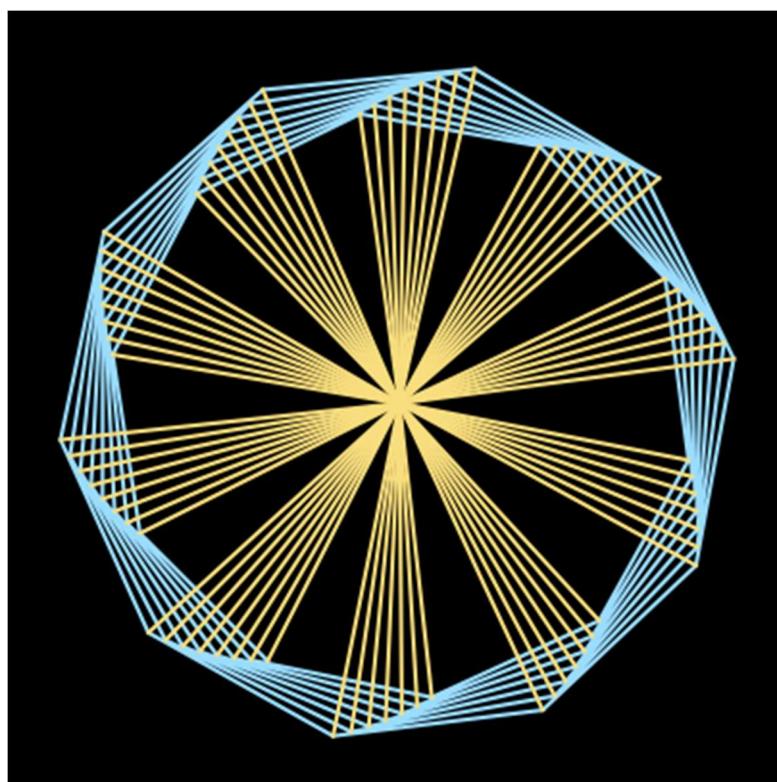


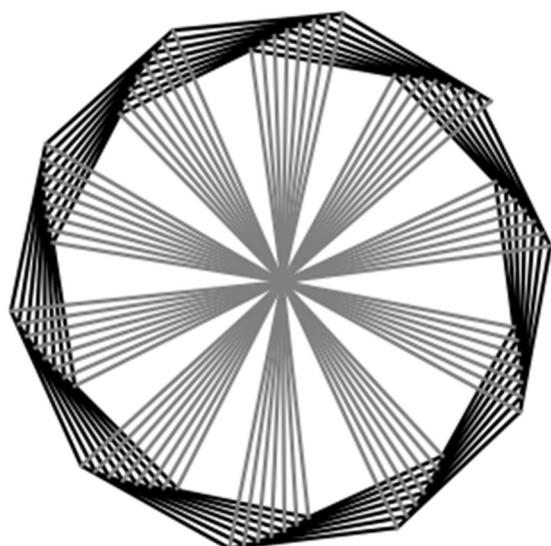
Métodos numéricos básicos para ingeniería con Python y Excel

2^a. Edición

Carlos Armando De Castro Payares



tutor.cadecastro.com



© Carlos Armando De Castro Payares. 2^a. Edición. 2024.

Se permite su reproducción total o parcial con fines educativos y académicos, no lucrativos, dando crédito al autor como "Carlos A. De Castro P." o "Carlos Armando De Castro".

Latinoamérica

<https://tutor.cadecastro.com>

CONTENIDO

Introducción	3
1. Interpolación	5
1.1. Interpolación lineal	5
1.2. Polinomios de Lagrange	7
2. Regresiones.....	11
2.1. Regresión lineal por mínimos cuadrados.....	11
2.2. Regresión con uso de Excel.....	14
2.3. Regresiones polinómicas con Python	16
3. Ecuaciones algebraicas no lineales.....	18
3.1. Método de punto fijo o iteración directa	18
3.2. Método de Newton-Raphson.....	22
3.3. Método de la secante	25
4. Sistemas de ecuaciones lineales.....	28
4.1. Método de Jacobi.....	29
4.2. Método de Gauss-Seidel	29
5. Sistemas de ecuaciones no lineales	33
5.1. Método de punto fijo o iteración directa multivariable	33
5.2. Método de punto fijo o iteración directa actualizada	37
5.3. Método de Newton-Raphson multivariable	39
6. Derivación e integración numéricas	42
6.1. Derivación por diferencias finitas	42
6.2. Integración por método de los trapecios.....	45
7. Ecuaciones diferenciales con valor inicial	48
7.1. Método de Euler	48
7.2. Método de Runge-Kutta de 4º orden.....	50
8. Ecuaciones diferenciales con valores en la frontera.....	54
8.1. Solución por diferencias finitas.....	54
Bibliografía	59
Sobre el autor	59
Asesorías en Matemáticas, Física e Ingeniería	60

Introducción

¿Qué son y para qué sirven los métodos numéricos? Voy a tomar lo que dice Wikipedia¹:

“El análisis numérico o cálculo numérico es la rama de las matemáticas que se encarga de diseñar algoritmos para, a través de números y reglas matemáticas simples, simular procesos matemáticos más complejos”.

Algo que falta en esa definición es decir que son iterativos, prácticamente a prueba y error. Los métodos numéricos nos facilitan la vida ya que simplifican los problemas a cálculos que puede hacer un computador, es más, la mayoría de los problemas de ingeniería terminan siendo resueltos de esta forma y no analíticamente en ecuaciones largas y complejas; sin embargo, se necesita que el ingeniero que los está utilizando entienda lo que hay detrás de ellos y sepa lo que está haciendo, ya que una persona capacitada puede saber si los resultados que está recibiendo son válidos y tienen sentido o son la respuesta dada por un método mal implementado o un algoritmo defectuoso.

En este libro se muestran los métodos numéricos que a mi criterio son los más sencillos y útiles de implementar en algunos problemas comunes de ingeniería. En los temas presentados no se hacen deducciones matemáticas complejas o profundas ni discusiones largas sobre el origen de los métodos (lo cual se puede conseguir en libros más avanzados sobre el tema para quien esté interesado en la formalidad que requiere la matemática) sino que se muestran éstos con alguna sencilla forma de visualizar la base detrás de éste (*por qué funciona*), se presenta un algoritmo en Python o en pseudocódigo cuando no sea posible hacer uno general en el software, y luego se procede a ilustrar con algún ejemplo.

Los métodos numéricos mostrados han sido utilizados por el autor en algún momento cuando estudiaba o en su trabajo como ingeniero. De forma particular me gusta ilustrar el uso de Excel ya que este software o similares como LibreOffice son comunes en las empresas, Python como un lenguaje poderoso y *gratuito* es también extremadamente útil y por tanto esta nueva edición (2024) usa éste y no Matlab como las anteriores.

Para que los códigos de Python acá enseñados funcionen es menester importar las siguientes librerías:

```
import numpy as np , matplotlib.pyplot as plt , pandas as pd
```

¹ http://es.wikipedia.org/wiki/An%C3%A1lisis_num%C3%A9rico . Un profesor no daría como válida una referencia de Wikipedia, pero pues acá no estamos en un ambiente académico formal y esta definición da a entender el punto más allá de discusiones en torres de marfil.

Todos los ejemplos hechos con Excel se anexan en el archivo descargable de la página web <https://tutor.cadecastro.com> (la cual enlaza a mi perfil de [Github](#)).

Ésta versión es la segunda edición con códigos en Python, ya había habido otras versiones anteriores de este libro (que están disponibles en varias páginas web) con los métodos implementados en Matlab, software el cual por ser comercial (y costoso) he decidido como autor deprecarlo y de hecho ya no lo utilizo en mi trabajo.

1. Interpolación

En la práctica de la ingeniería se utilizan mucho las tablas de datos, como en el caso de las tablas de propiedades en la termodinámica, por decir solo una. En la mayoría de los casos el dato que necesitamos no se encuentra explícito en la tabla sino entre dos valores de ésta, para lo cual es necesario estimarlo de entre los valores que presenta la tabla en un proceso conocido como *interpolación*.

La idea básica de la interpolación es hallar un polinomio o función que cumpla con pasar por todos los puntos de un conjunto de datos $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$, y poder estimar los valores entre ellos por medio del polinomio.

1.1. Interpolación lineal

La interpolación lineal es la más utilizada en el manejo de datos de tablas. Consiste en trazar una recta entre cada par de los puntos de datos, razón por la cual también es llamada interpolación por *trazadores lineales* o *splines de primer orden*. Consideremos un conjunto de datos $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$, entre dos puntos consecutivos del conjunto de datos se puede trazar un segmento de recta:

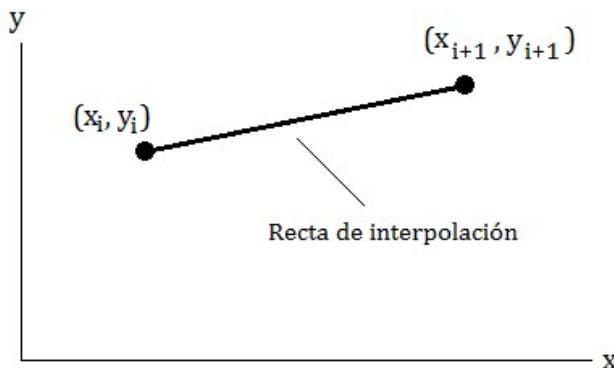


Figura 1.1. Interpolación lineal.

La pendiente de esta recta es $m = (y_{i+1} - y_i)/(x_{i+1} - x_i)$ y como pasa por el punto inicial (x_i, y_i) se tiene entonces la ecuación de la recta que interpola entre ese par de puntos es

$$y = \left(\frac{y_{i+1} - y_i}{x_{i+1} - x_i} \right) (x - x_i) + y_i \quad (1.1)$$

Hay que tener en cuenta que la interpolación lineal se hace por pedazos y no entrega un solo polinomio para todo el conjunto de datos.

La implementación de la interpolación lineal en Python teniendo en cuenta que es a pedazos se muestra en el algoritmo 1.1.

Algoritmo 1.1: Interpolación lineal en Python

```
def interpolacion_lineal( x, X, Y ):
    """
    x: valor buscado para interpolar
    X: datos de entrada
    Y: datos de entrada
    Entrega valor interpolado y
    """

    if x<min(X) or x>max(X):
        raise Exception('Valor de x fuera de los límites.')
    for i in range( len(X)-1 ):
        if x>=X[i] and x<=X[i+1]:
            y=(Y[i+1]-Y[i])/(X[i+1]-X[i])*(x-X[i])+Y[i]
    return y
```

Ejemplo 1.1. Termodinámica (Excel)

Una parte de la tabla de presión y temperatura de saturación del agua es²:

P [kPa]	T [°C]
35	72.7
45	78.7

Necesitamos saber la presión requerida para que el agua se sature a una temperatura de 75°C, en la tabla vemos este valor no aparece explícito por lo que debemos interpolar, entonces aplicando la ecuación (1.1) donde x son las presiones e y las temperaturas:

$$P = \left(\frac{45 - 35}{78.7 - 72.7} \right) (75 - 72.7) + 35$$

$$P = 38.83 \text{ kPa}$$

Utilizando una hoja de Excel programada con la ecuación (1.1) tenemos:

Tabla de datos		Valor deseado x
x	y	
72.7	35	
78.7	45	Interpolado y 38.83

² http://www.engineeringtoolbox.com/saturated-steam-properties-d_101.html

Y la gráfica de esta interpolación se muestra a continuación:

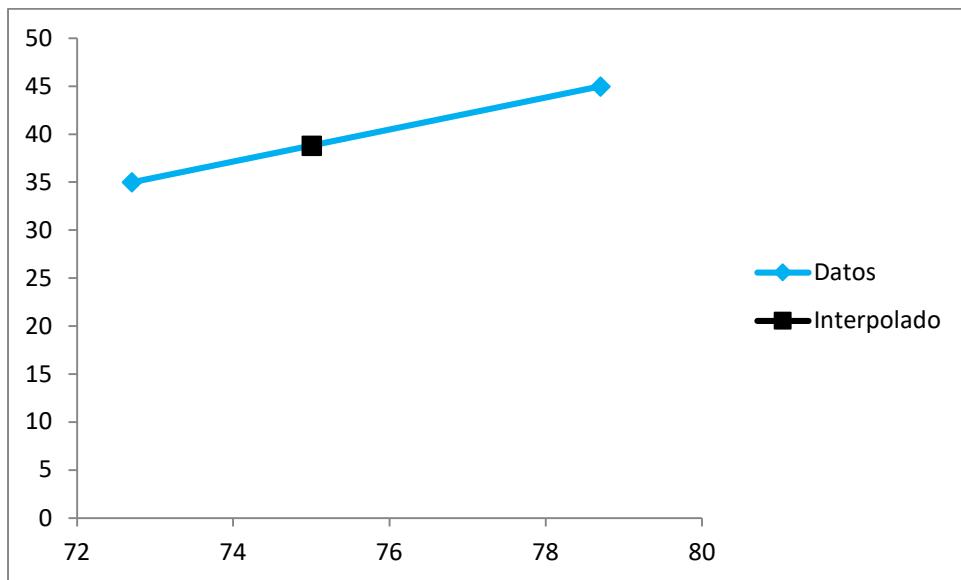


Figura 1.2. Interpolación de la tabla de saturación del agua.

1.2. Polinomios de Lagrange

Este método de interpolación no lo he usado mucho en la práctica y lo pongo en carácter informativo. Para ilustrar cómo funciona la interpolación por polinomios de Lagrange considérese un conjunto de datos de tres puntos $(x_1, y_1), (x_2, y_2), (x_3, y_3)$. El polinomio interpolador en este caso es

$$P(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} y_1 + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} y_2 + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} y_3$$

Obsérvese que en el punto $x = x_1$ sólo queda el primer término con su numerador y denominador cancelándose entre sí, por lo cual $P(x_1) = y_1$. Lo mismo sucede con los demás puntos, por lo que se ve que el polinomio cumple con la condición de pasar por todos los puntos de datos. Generalizando, para n puntos de datos, el polinomio de Lagrange es:

$$P(x) = \sum_{i=1}^n y_i \prod_{\substack{j=1 \\ j \neq i}}^n \frac{(x - x_j)}{(x_i - x_j)} \quad (1.2)$$

Una forma mucho más sencilla de ver la (1.2) es en forma de un algoritmo, el cual se muestra escrito para Python en el algoritmo 1.1.

Algoritmo 1.2: Polinomios de Lagrange en Python

```
def interpolacion_lagrange( x , X , Y ):
    """
        x: valor buscado para interpolar
        X: datos de entrada
        Y: datos de entrada
        Entrega valor interpolado y
    """

    if x<min(X) or x>max(X):
        raise Exception('Valor de x fuera de los límites.')
    y=0
    for i in range( len(X) ):
        L=1
        for j in range( len(X) ):
            if j!=i:
                L=L*(x-X[j])/(X[i]-X[j])
        y=y+L*Y[i]

    return y
```

Ejemplo 1.2. Tablas de saturación de agua (Python)

Tomamos un fragmento de las tablas de temperatura y presión de saturación de agua del libro de Cengel:

```
#Datos de agua saturada del libro de Termodinámica de Yunus Cengel:
T = [ 0.01, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 ]
P = [ 0.6117, 0.8725, 1.2281, 1.7057, 2.3392, 3.1698, 4.2469, 5.6291, 7.3851, 9.5953, 12.352 ]
```

Implementamos las funciones definidas en los algoritmos 1.1 y 1.2 en un ciclo y gráfica como se muestra en el código siguiente:

```
xs = np.linspace( 0.01 , 50 , 100 )
y_lagr = np.empty( shape=100 )
y_lineal = np.empty( shape=100 )

for i in range( len(xs) ):
    y_lagr[i] = interpolacion_lagrange( xs[i], T , P )
    y_lineal[i] = interpolacion_lineal( xs[i], T , P )
```

```

plt.figure( 1, figsize=(10,4) )
plt.scatter( T , P , color='navy', label='Datos' )
plt.plot( xs, y_lagr, color='forestgreen', label='Pol. Lagrange' )
plt.plot( xs, y_lineal, color='purple', label='Interp. Lineal' )
plt.title('Interpolaciones de tablas de agua saturada', color='navy', weight='bold')
plt.ylabel('Presión de saturación (kPa)', rotation=90 ,style='italic')
plt.xlabel('Temperatura (°C)', style='italic')
plt.legend( ncol=3 )
plt.grid()

```

La gráfica de los puntos y los polinomios interpoladores es la siguiente:

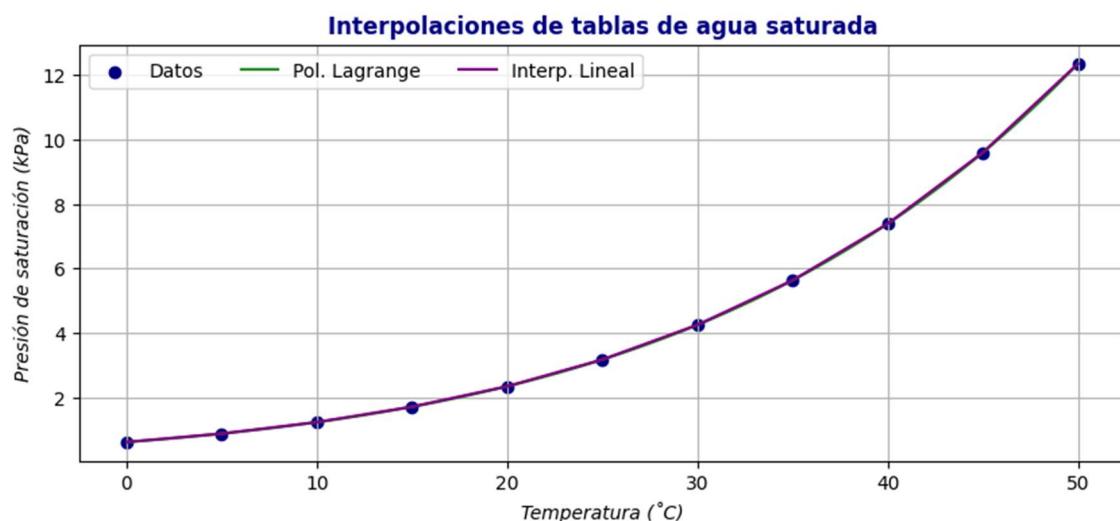


Figura 1.3. Datos de saturación de agua interpolados linealmente y por polinomios de Lagrange.

En este caso ambos métodos dan resultados aceptables, sin embargo, los polinomios de Lagrange en ocasiones presentan problemas como veremos en el siguiente ejemplo.

Ejemplo 1.3. Ensayo de tensión (Python)

Tomamos los datos de un ensayo de tensión del libro *Ciencia e Ingeniería de los materiales* de Donald R. Askeland:

```
#Datos de ensayo de tensión del libro "Ciencia e ingeniería de materiales", 6a. Edición de Donald R. Askeland, Tabla 6-1:
deformacion = [ 0, 0.0005, 0.0015, 0.0025, 0.0035, 0.0150, 0.0400, 0.06, 0.08, 0.1025 ] #in/in
esfuerzo = [ 0, 4993, 14978, 24963, 34948, 37445, 39442, 39941, 39691, 37944 ] #psi
```

Implementamos las funciones definidas en los algoritmos 1.1 y 1.2 en un ciclo como se muestra en el código siguiente:

```

xs = np.linspace( 0 , 0.1025 , 200 )
y_lagr = np.empty( shape=200 )
y_lineal = np.empty( shape=200 )

for i in range( len(xs) ):
    y_lagr[i] = interpolacion_lagrange( xs[i], deformacion , esfuerzo )
    y_lineal[i] = interpolacion_lineal( xs[i], deformacion , esfuerzo )

```

La gráfica de los puntos y los polinomios interpoladores es la siguiente:

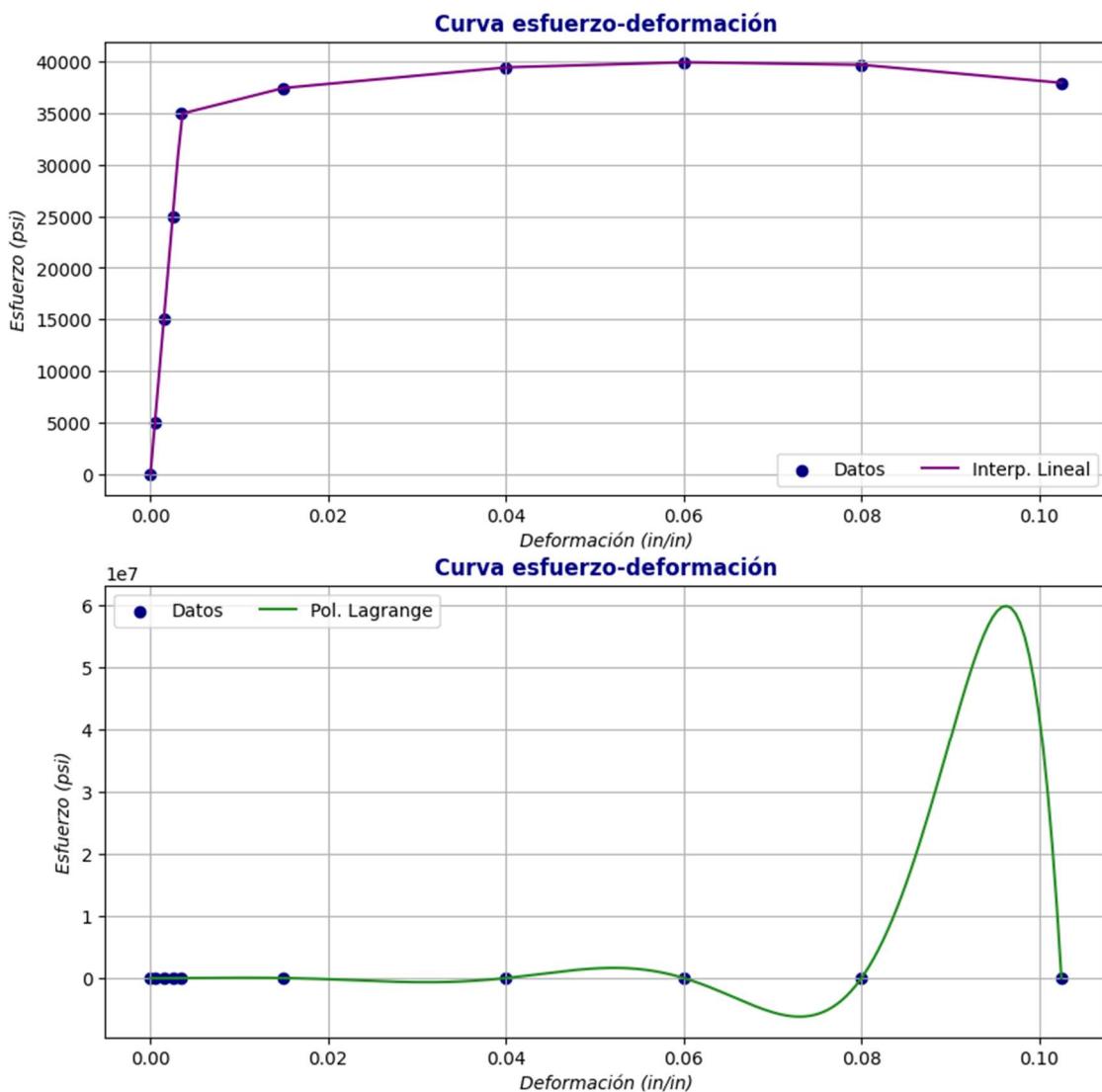


Figura 1.4. Datos de ensayo de tensión interpolados linealmente y por polinomios de Lagrange, se observa que el último método entrega unas oscilaciones *enormes* en la parte plástica (después de la zona lineal inicial) por lo que **no es adecuado** para la interpolación de estos datos.

2. Regresiones

En ocasiones se tiene un conjunto de datos experimentales y se desea hallar una función analítica que los represente de forma satisfactoria. Para ello es necesario hacer una aproximación de la función a los datos con el menor error posible.

2.1. Regresión lineal por mínimos cuadrados

Los mínimos cuadrados es un método basado en minimizar el error entre los datos y la función de aproximación. Para un conjunto de datos $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$, si $y = f(x)$ es la función de aproximación, la suma del cuadrado de los errores es:

$$R = \sum_{i=1}^n (f(x_i) - y_i)^2 \quad (2.1)$$

La idea es que la función f de aproximación minimice el valor de R descrito por (2.1). En el caso de una recta, se tiene

$$R = \sum_{i=1}^n (mx_i + b - y_i)^2 \quad (2.2)$$

Para minimizar el error derivamos (2.2) respecto a los parámetros de la recta, por lo que debe cumplirse $\frac{\partial R}{\partial m} = 0, \frac{\partial R}{\partial b} = 0$, entonces

$$\frac{\partial R}{\partial m} = 2 \sum_{i=1}^n (mx_i + b - y_i)x_i = 0$$

$$\frac{\partial R}{\partial b} = 2 \sum_{i=1}^n (mx_i + b - y_i) = 0$$

De donde surge el sistema de ecuaciones lineal:

$$m \sum_{i=1}^n x_i^2 + b \sum_{i=1}^n x_i = \sum_{i=1}^n x_i y_i$$

$$m \sum_{i=1}^n x_i + bn = \sum_{i=1}^n y_i$$

Los parámetros de la recta de aproximación que minimiza R son entonces:

$$b = \frac{\sum_{i=1}^n x_i^2 \sum_{i=1}^n y_i - \sum_{i=1}^n x_i \sum_{i=1}^n x_i y_i}{n \sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2} \quad (2.3)$$

$$m = \frac{\sum_{i=1}^n x_i y_i}{\sum_{i=1}^n x_i^2} - \frac{\sum_{i=1}^n x_i}{\sum_{i=1}^n x_i^2} \left(\frac{\sum_{i=1}^n x_i^2 \sum_{i=1}^n y_i - \sum_{i=1}^n x_i \sum_{i=1}^n x_i y_i}{n \sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2} \right) \quad (2.4)$$

Algoritmo 2.1: Aproximación lineal por mínimos cuadrados en Python

Entradas: vectores conteniendo los puntos X y Y.

Salidas: pendiente m e intercepto b de la recta de aproximación.

```
def mincuadlin(X, Y):
    n=len(X)
    A = np.empty( shape=(2,2) )
    B = np.empty( shape=2 )
    A[0,0] = np.sum( X*X )
    A[0,1] = np.sum( X )
    A[1,0] = np.sum( X )
    A[1,1] = n
    B[0] = np.sum( X*Y )
    B[1] = np.sum( Y )
    sol = np.linalg.solve( A , B )
    m = sol[0]
    b = sol[1]
    return m, b
```

Hay además funciones propias de Numpy o de SciPy para esto. El análisis hecho para la aproximación por medio de una recta puede hacerse de manera análoga para hallar los coeficientes de cualquier función f que se quiera utilizar para aproximar.

Ejemplo 2.1. Temperatura de una placa al Sol (Python)

Se tienen los siguientes datos de una prueba en la que se midió la diferencia de temperatura de una placa expuesta al Sol y la ambiente, se desea aproximar por medio de una recta para hallar un coeficiente de transferencia de calor equivalente:

Rad. Sol. [W/m ²]	ΔT [°C]
300	5.4
350	6.5
450	7.1
500	8.1
600	9.5
800	12.3
1000	15.8

El coeficiente es el inverso de la pendiente de la recta de aproximación, corriendo el algoritmo 2.1 en Python se tiene:

$$m = 0.0145$$

$$b = 0.9437$$

Por lo que el coeficiente equivalente es $1/m = 68.97 \text{ W/m}^2\text{C}$. La gráfica de la recta de aproximación con los puntos de datos se muestra a continuación:

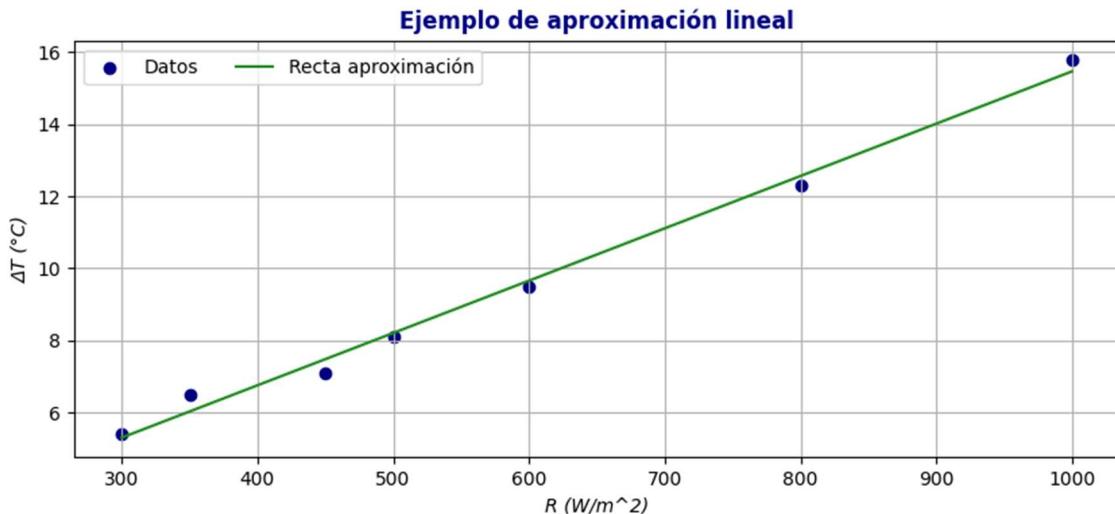


Figura 2.1. Aproximación lineal por mínimos cuadrados en Python.

El código usado para esto fue el siguiente:

```
R = np.array( [ 300, 350, 450, 500, 600, 800, 1000 ] )
dT = np.array( [ 5.4, 6.5, 7.1, 8.1, 9.5, 12.3, 15.8 ] )

m, b = mincuadlin( R, dT )

xs = np.linspace( 300 , 1000 , 50 )
ys = np.empty( shape=50 )
ys = m*xs + b

plt.figure( 1, figsize=(10,4) )
plt.scatter( R , dT , color='navy' , label='Datos' )
plt.plot( xs, ys, color='forestgreen' , label='Recta aproximación' )
plt.title('Ejemplo de aproximación lineal', color='navy',
weight='bold')
plt.ylabel('ΔT (°C)', rotation=90 ,style='italic')
plt.xlabel('R (W/m^2)' , style='italic')
plt.legend( ncol=2 )
plt.grid()
```

2.2. Regresión con uso de Excel

El software Excel tiene una ventaja muy grande y es que hace por sí mismo las aproximaciones a un conjunto de datos, desde lineales, cuadráticas o el grado de polinomio que uno desee hasta aproximaciones por medio de logaritmos.

Hago énfasis en lo siguiente: *las aproximaciones que hace Excel siguen la misma filosofía presentada en la minimización del error de la sección 2.1.*

Ejemplo 2.2. Temperatura de una placa al Sol (Excel)

Con los mismos datos del Ejemplo 2.1, se grafican utilizando Insertar Gráfico en Excel y tipo de gráfico “Dispersión”:

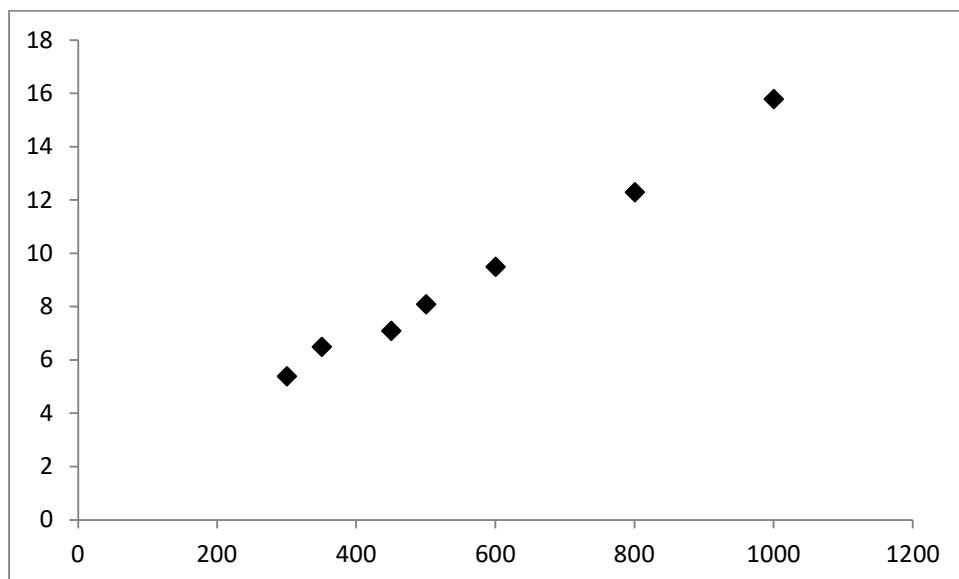
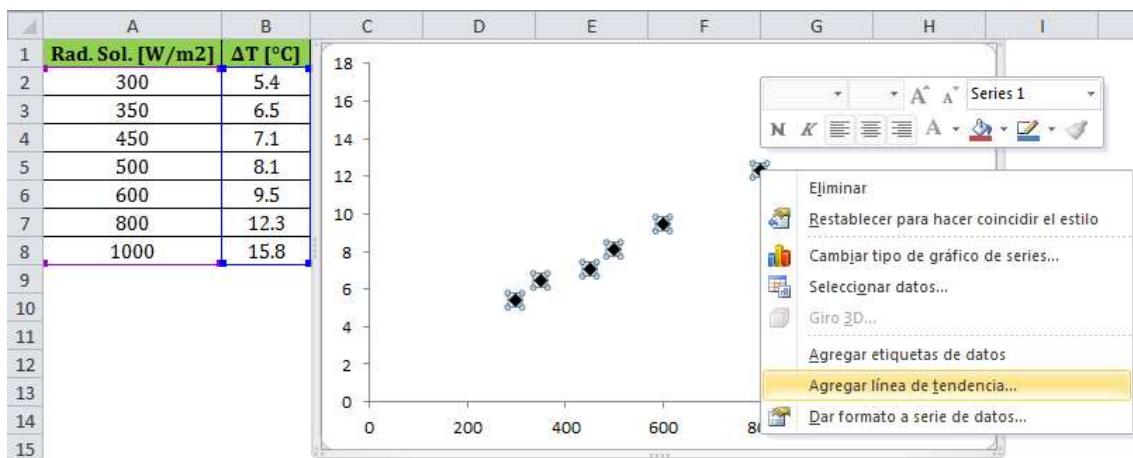
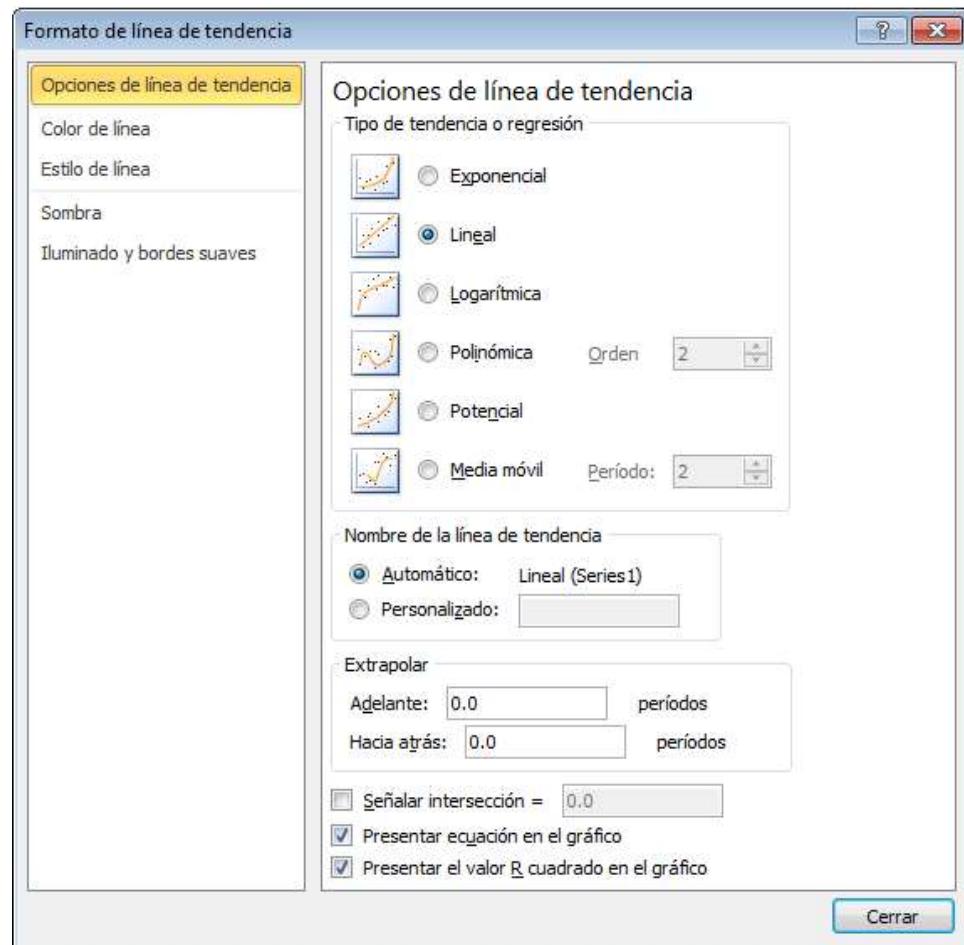


Figura 2.2. Gráfica de los datos del ejemplo.

Se hace click derecho sobre alguno de los puntos y se selecciona “Aregar línea de tendencia”:



Una vez se hace esto aparece el cuadro que permite escoger el tipo de aproximación y da la opción de mostrar la ecuación de la función de aproximación, en este caso escogemos aproximación lineal:



El resultado se muestra en la Figura 2.3, donde podemos observar la pendiente y el intercepto de la recta de aproximación siendo iguales al ejemplo 2.1:

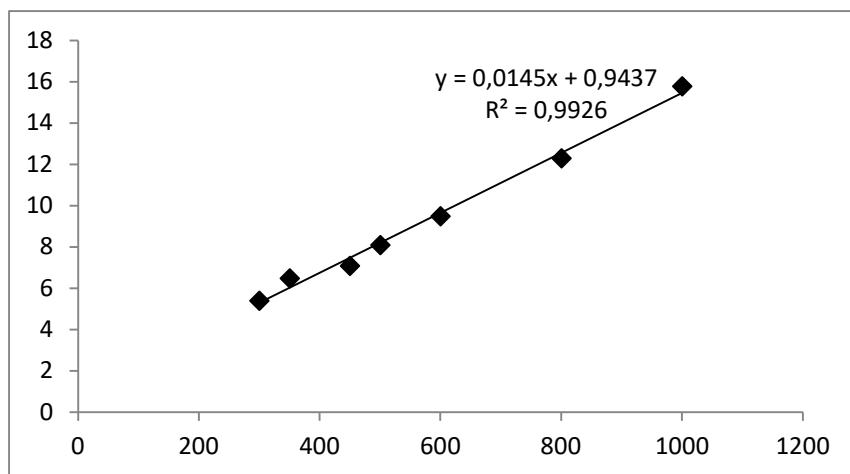


Figura 2.3. Aproximación lineal en Excel.

2.3. Regresiones polinómicas con Python

La librería Numpy de Python tiene una función propia para con un conjunto de datos hacer regresión con polinomios de grado deseado, en la práctica un polinomio de grado 3 es suficiente para la mayoría de los casos.

Algoritmo 2.2: Regresión polinómica en Python

```
def regresion_polinomica( X , Y , n ):
    """
    X: datos de entrada
    Y: datos de entrada
    n = grado del polinomio de regresión, 1 es lineal
    Entrega coeficientes del polinomio p, valores predichos y_pred, valor de R2
    """

    p = np.polyfit( X , Y , n ) #Coeficientes del polinomio de regresión
    y_pred = np.polyval( p , X ) #Valores predichos con los datos originales
    y_m = np.mean(Y) #Promedio de los datos originales
    #Cálculo del valor de R²:
    SSR = np.sum( ( Y-y_pred )**2 )
    SST = np.sum( ( Y - y_m )**2 )
    R2=1-SSR/SST

    return p , y_pred , R2
```

El valor R^2 calculado y entregado por el algoritmo 2.2 es un indicativo de qué tan cercano a los datos originales quedó el polinomio entregado por la regresión, entre más cercano a 1 es mejor la regresión, uno cercano a 0 indica mala aproximación a los datos originales.

Ejemplo 2.3. Regresión tablas de agua saturada

Con los valores de temperatura y presión usados en el ejemplo 1.2 aplicamos el algoritmo 2.2 con $n=1$ para una regresión lineal:

```
#Regresión lineal tablas termodinámica:
p1, y1 , R2_1 = regresion_polinomica( T, P, 1 )
```

Haciendo otros códigos adicionales que no mostraré aquí tenemos la gráfica y el polinomio explícito de esta regresión:

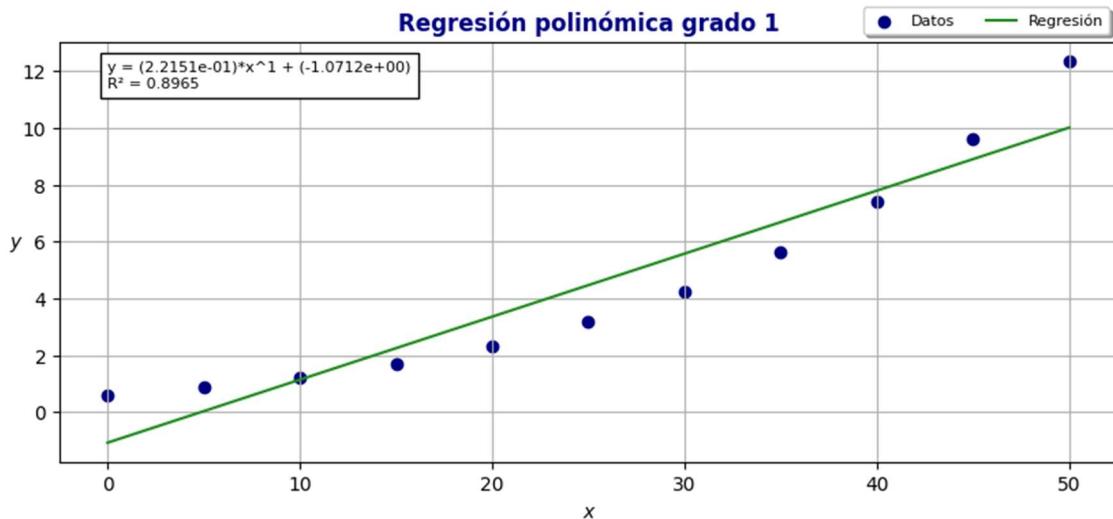


Figura 2.4. Regresión lineal a los datos de agua saturada.

Vemos que el valor de R^2 es cercano a 1 pero visualmente la regresión no es tan cercana a los datos originales, miremos que pasa al aplicar un polinomio de grado 3 a estos datos:

```
#Regresión cúbica tablas termodinámica:  
p2, yp2 , R2_2 = regresion_polinomica( T, P, 3 )
```

Y para esta regresión:

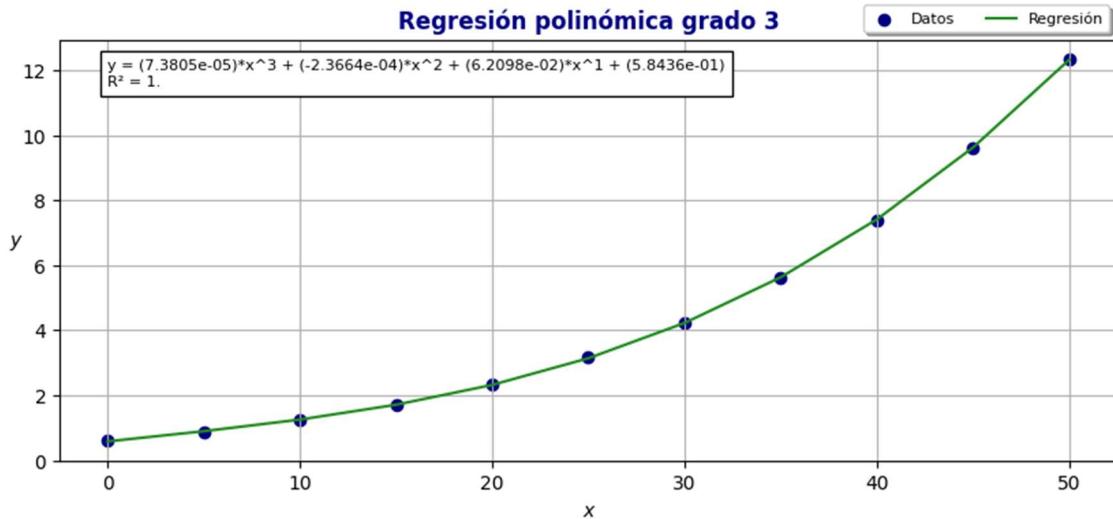


Figura 2.5. Regresión cúbica a los datos de agua saturada.

¡Vemos que la regresión cúbica es prácticamente perfecta para este conjunto de datos!

3. Ecuaciones algebraicas no lineales

En ocasiones en el ámbito de la ingeniería es necesario resolver ecuaciones no lineales que no tienen solución analítica o que es muy engorroso hallarlas. Para estos casos, deben utilizarse métodos de solución numérica de ecuaciones como los presentados en esta sección.

3.1. Método de punto fijo o iteración directa

El método de punto fijo consiste en una forma iterativa de resolver una ecuación de la forma $f(x) = x$. El método consiste en elegir una aproximación inicial x_0 y realizar la iteración

$$x_{k+1} = f(x_k) \quad (3.1)$$

Hasta que la diferencia $|x_{k+1} - x_k|$ sea muy cercana a cero, para lo cual se establece una tolerancia a criterio del usuario. En el algoritmo 3.1 se muestra el algoritmo de punto fijo en Python.

Algoritmo 3.1: Método de punto fijo o iteración directa

```
def metodo_punto_fijo( f:'función' , x0:float , it_max:int , error_max:float , print_results:bool ):
    """
    f: función de iteración previamente definida
    x0: aproximación inicial
    it_max: máximo número de iteraciones a realizar
    error_max: error máximo (tolerancia) admisible para la solución
    print_results: booleano para indicar si imprimir resultados
    """
    sw = 1
    it = 1
    while sw == 1:
        x = f(x0)
        it += 1
        residual = abs( x - x0 )
        if residual<=error_max or it >= it_max:
            sw = 0
        else:
            x0 = x
    if print_results:
        print('*'*30)
        print('')
        print('    Raíz =', np.format_float_scientific( x , 4 ) )
        print('')
        print('    Residual =', np.format_float_scientific( residual , 4 ) )
        print('Iteraciones =', it )
        print('*'*30)
        print('')

    return x, residual, it
```

Ejemplo 3.1. Mecánica de la fractura (Excel)

La ecuación de factor de intensidad de esfuerzos para una placa de ancho w y espesor t con una grieta en el borde de largo a es

$$K = \sigma Y \sqrt{a} \quad (E3.1)$$

Donde Y es un factor geométrico que depende del ancho de la placa y el tamaño de grieta, siendo:

$$Y = \left[1.99 - 0.41 \left(\frac{a}{w} \right) + 18.70 \left(\frac{a}{w} \right)^2 - 38.48 \left(\frac{a}{w} \right)^3 + 53.85 \left(\frac{a}{w} \right)^4 \right] \quad (E3.2)$$

La falla catastrófica de la placa se produce cuando el factor de intensidad de esfuerzos K iguala o supera a la tenacidad a la fractura K_{IC} , entonces el tamaño de grieta crítica es

$$a_f = (K_{IC}/\sigma Y)^2 \quad (E3.3)$$

Como el factor geométrico Y depende de a_f , la ecuación E3.3 debe resolverse por el método de punto fijo. La iteración es entonces:

$$a_{k+1} = \left(\frac{K_{IC}}{\sigma \left[1.99 - 0.41 \left(\frac{a_k}{w} \right) + 18.70 \left(\frac{a_k}{w} \right)^2 - 38.48 \left(\frac{a_k}{w} \right)^3 + 53.85 \left(\frac{a_k}{w} \right)^4 \right]} \right)^2$$

Se tiene un caso de una placa sujeta a tensión donde $w = 2.5\text{in}$, $\sigma = 24.89\text{ksi}$, $K_{IC} = 52\text{ksi}\sqrt{\text{in}}$. Se elige como aproximación inicial $a_0 = 0.250\text{in}$. Una hoja de Excel programada para este caso particular con el método de punto fijo entrega la solución con tres cifras decimales:

$a(i)$	Y	$a(i+1)$	$a(i+1) - a(i)$	Iteración
0.250	2.103	0.987	0.737	1
0.987	3.684	0.322	-0.665	2
0.322	2.180	0.919	0.597	3
↓	↓	↓	↓	↓
0.620	2.655	0.619	-0.001	118
0.619	2.654	0.620	0.001	119
0.620	2.655	0.620	0.000	120

La tabla muestra que la iteración converge en 120 pasos al valor $a_f = 0.620\text{in}$.

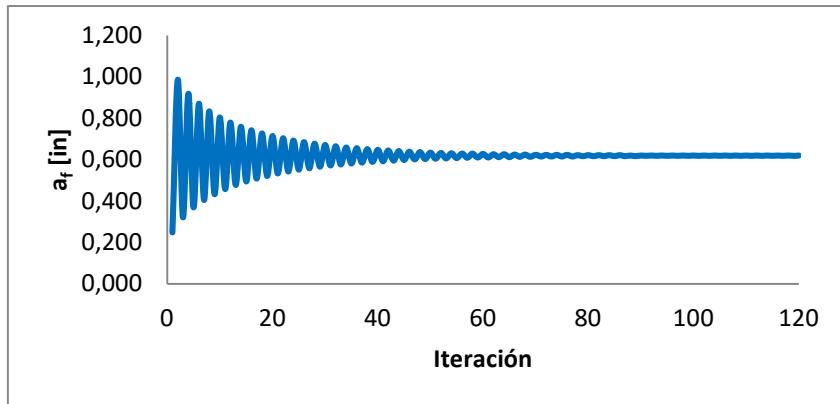


Figura 3.1. Valor calculado de a_f en cada iteración.

Ejemplo 3.2. Factor de fricción de un flujo turbulento (Python)

Para una tubería real con un flujo con número de Reynolds Re y una rugosidad relativa al diámetro k_s/d se tiene por la ecuación de Colebrook-White el factor de fricción que es necesario para calcular las pérdidas de energía:

$$\frac{1}{f^{1/2}} = -2 \log_{10} \left(\frac{k_s/d}{3.7} + \frac{2.51}{Re \cdot f^{1/2}} \right) \quad (E3.4)$$

Nótese que el factor de fricción f depende de él mismo dentro del logaritmo por lo que es el ejemplo perfecto para el método de punto fijo. Implementando el algoritmo 3.1 en un ciclo podemos calcular el factor de fricción para un par de rugosidades relativas y para un rango amplio de números de Reynolds:

```
ks_d1 = 1e-3
ks_d2 = 1e-6
Re_array = np.linspace( 10e3, 1e6 , 500 )
f_array1 = np.empty( shape=len( Re_array ) )
f_array2 = np.empty( shape=len( Re_array ) )

for j in range( len( Re_array ) ):
    Re = Re_array[j]
    def colebrook_white1( x ):
        res = ( -2*np.log10( ks_d1/3.7 + 2.51/( Re*x**0.5 ) ) )**(-2)
        return res
    def colebrook_white2( x ):
        res = ( -2*np.log10( ks_d2/3.7 + 2.51/( Re*x**0.5 ) ) )**(-2)
        return res

    f_array1[j] = metodo_punto_fijo( colebrook_white1 , x0=0.001 , it_max=100 , error_max=1e-6 , print_results=False )[0]
    f_array2[j] = metodo_punto_fijo( colebrook_white2 , x0=0.001 , it_max=100 , error_max=1e-6 , print_results=False )[0]

plt.figure( 1, figsize=(10,4) )
plt.plot( Re_array , f_array1 , color='purple' , label=str( ks_d1 ) )
plt.plot( Re_array , f_array2 , color='forestgreen' , label=str( ks_d2 ) )
plt.title('Factor de fricción según Ec. de Colebrook-White', color='navy', weight='bold')
plt.ylabel('f', rotation=0 ,style='italic')
plt.xlabel('Re', style='italic')
plt.legend( title='ks/d =' , framealpha=1, fancybox=True, shadow=True, ncol=1 )
plt.xscale('log')
plt.grid()
```

El resultado se muestra en la siguiente figura:

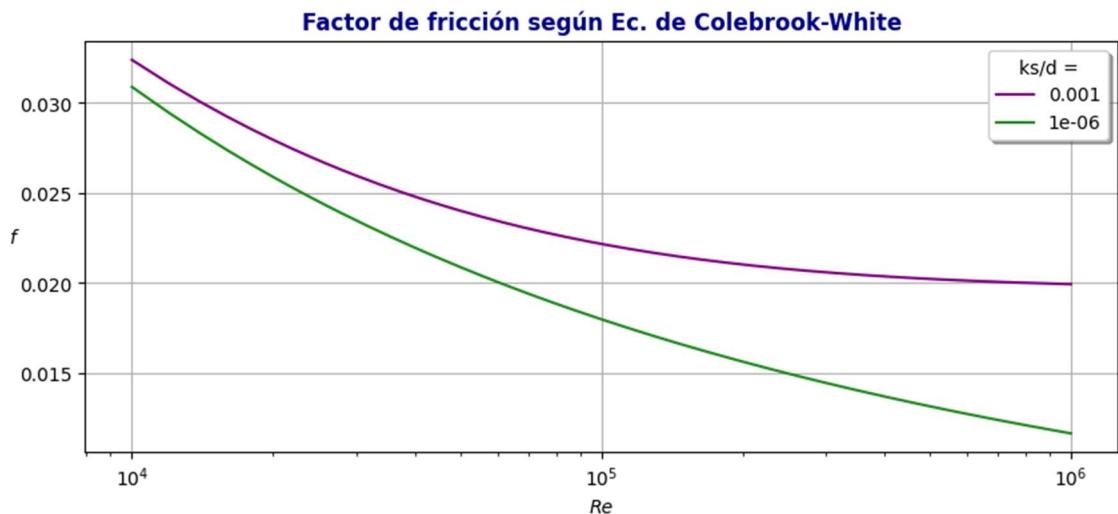


Figura 3.2. Factor de fricción de flujo turbulento para dos rugosidades relativas separadas por tres órdenes de magnitud (flujo rugoso y flujo liso) y varios números de Reynolds (esto es una parte del *Diagrama de Moody*).

(Como sobró espacio, una foto de Rambito)



3.2. Método de Newton-Raphson

Se utiliza para resolver ecuaciones de la forma $f(x) = 0$. El método de Newton se obtiene de la serie de Taylor truncada en el primer término, recordando la serie de Taylor para una función f alrededor de un punto x_0 tenemos:

$$f(x) = f(x_0) + f'(x_0)(x - x_0)$$

En el punto deseado para la solución $f(x) = 0$ por lo que despejando x :

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

El método consiste en elegir una aproximación inicial x_0 y realizar la iteración

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

(3.2)

Hasta que la diferencia $|x_{k+1} - x_k|$ sea muy cercana a cero, para lo cual se establece una tolerancia a criterio del usuario. En el algoritmo 3.2 se muestra el algoritmo de Newton-Raphson en Python.

Algoritmo 3.2: Método de Newton-Raphson

```
def metodo_newton_raphson( f:'función' , df:'función' , x0:float , it_max:int , error_max:float , print_results:bool ):
    """
    f: función previamente definida
    df: derivada de la función previamente definida
    x0: aproximación inicial
    it_max: máximo número de iteraciones a realizar
    error_max: error máximo (tolerancia) admisible para la solución
    print_results: booleano para indicar si imprimir resultados
    """
    sw = 1
    it = 1
    while sw == 1:
        x = x0 - f(x0)/df(x0)
        it += 1
        residual = abs( x - x0 )
        if residual<error_max or it >= it_max:
            sw = 0
        else:
            x0 = x
        if print_results:
            print('*'*30)
            print('')
            print('    Raíz =', round( x , 4 ) )
            print('');
            print('    Residual =', np.format_float_scientific( residual , 4 ) )
            print('Iteraciones =', it )
            print('*'*30)
            print('')
    return x , residual, it
```

Ejemplo 3.3. Raíz de polinomio cúbico (Excel)

Se desea resolver la ecuación $3x^3 - 2x + 8 = 0$ con tres cifras significativas, entonces se tiene $f(x) = 3x^3 - 2x + 8$ y $f'(x) = 9x^2 - 2$. La iteración es entonces

$$x_{k+1} = x_k - \frac{3x_k^3 - 2x_k + 8}{9x_k^2 - 2} \quad (E3.5)$$

Se escoge una aproximación inicial $x_0 = -10$, entonces, una sencilla tabulación en Excel da la raíz $x = -1.546$

x(i)	x(i+1)	x(i+1)-x(i)
-10.000	-6.690	3.310
-6.690	-4.502	2.188
-4.502	-3.079	1.423
-3.079	-2.198	0.881
-2.198	-1.729	0.469
-1.729	-1.566	0.162
-1.566	-1.546	0.020
-1.546	-1.546	0.000

Ejemplo 3.4. Raíz de polinomio cúbico (Python).

Para el mismo polinomio del ejemplo 3.3 tenemos:

```
def g(x):
    res = 3*x**3 - 2*x + 8
    return res

def dg(x):
    res = 9*x**2 - 2
    return res

metodo_newton_raphson( g , dg , x0=-10 , it_max=100 , error_max=1e-4 , print_results=True )
```

Los resultados que entrega de una manera mucho más rápida que el Excel son los siguientes:

```
Raíz = -1.5463
Residual = 5.673e-08
Iteraciones = 10
(-1.546340339564043, 5.672979286330815e-08, 10)
```

¿Cómo sabemos qué suposición inicial utilizar? Con las herramientas computacionales la mejor manera es graficar la función y observar cerca a qué valores tiene la raíz al ojo para así poner como suposición inicial un valor que sea cercano y con el cual el método numérico pueda converger, o si hay varias raíces se pueden tomar varios puntos iniciales de iteraciones para hallar cada una de ellas individualmente.

Gráfica del polinomio cúbico de los ejemplos 3.3 y 3.4

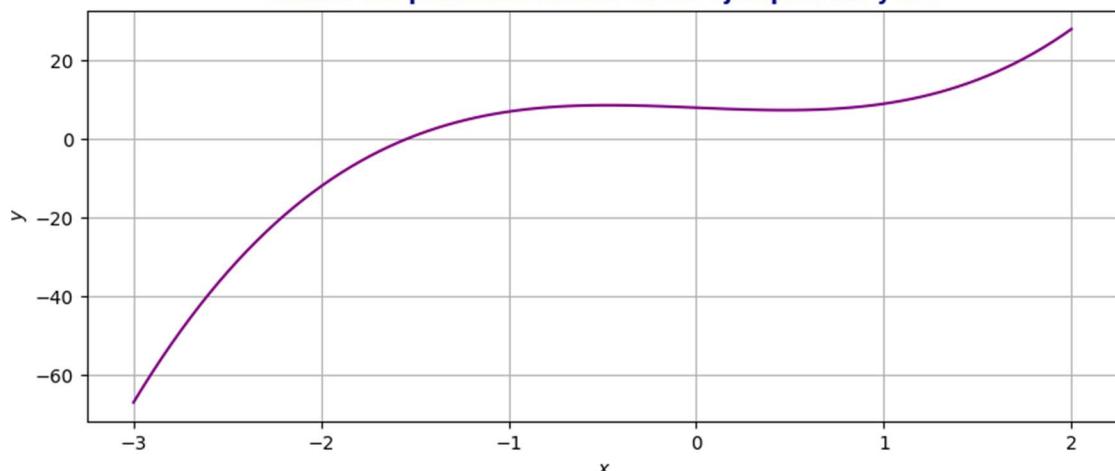


Figura 3.3. Gráfica del polinomio cúbico de los ejemplos 3.3 y 3.4 donde se observa que la gráfica pasa por el cero (la raíz) para un x ubicado entre -2 y -1, como efectivamente las soluciones que hemos hallado comprobaron.



3.3. Método de la secante

Se utiliza para resolver ecuaciones de la forma $f(x) = 0$. El método consiste en elegir dos aproximaciones iniciales x_0, x_1 y realizar la iteración

$$x_{k+1} = x_k - \frac{(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} \frac{f(x_k)}{f(x_k) - f(x_{k-1})} \quad (3.3)$$

Hasta que la diferencia $|x_{k+1} - x_k|$ sea muy cercana a cero, para lo cual se establece una tolerancia a criterio del usuario. El método de la secante tiene la ventaja de que no se debe derivar la ecuación, es una forma discretizada del método de Newton. En el algoritmo 3.3 se muestra el algoritmo del método de la secante en Python.

Algoritmo 3.3: Método de la secante

```
def metodo_secante( f:'función' , x0:float , x1:float , it_max:int , error_max:float , print_results:bool ):
    """
    f: función previamente definida
    x0, x1: aproximaciones iniciales
    it_max: máximo número de iteraciones a realizar
    error_max: error máximo (tolerancia) admisible para la solución
    print_results: booleano para indicar si imprimir resultados
    """
    sw = 1
    it = 1
    while sw == 1:
        x = x1 - (x1-x0)*f(x1)/( f(x1) - f(x0) )
        it += 1
        residual = abs( x - x0 )
        if residual<=error_max or it >= it_max:
            sw = 0
        else:
            x0 = x1
            x1 = x
        if print_results:
            print('*'*30)
            print('')
            print('    Raíz =', round( x , 4 ) )
            print('');
            print('    Residual =', np.format_float_scientific( residual , 4 ) )
            print('Iteraciones = ', it )
            print('*'*30)
            print('')
    return x , residual, it
```

Ejemplo 3.5. Raíz de función trigonométrica (Excel)

Se necesita resolver la ecuación trigonométrica $\sin(x) + 3 \cos(x) = 0$ entre $0 \leq x \leq 10$ con tres cifras significativas. La iteración en este caso es entonces

$$x_{k+1} = x_k - \frac{\sin(x_k) + 3 \cos(x_k)}{\sin(x_k) + 3 \cos(x_k) - [\sin(x_{k-1}) + 3 \cos(x_{k-1})]} \frac{\sin(x_k) + 3 \cos(x_k)}{\sin(x_k) + 3 \cos(x_k) - [\sin(x_{k-1}) + 3 \cos(x_{k-1})]}$$

Con una tabla de Excel programada con el método de la secante (ver archivo adjunto de Excel) se hallan las tres soluciones $x = 1.893, x = 5.034, x = 8.176$.

Ejemplo 3.6. Tasa interna de retorno de un proyecto de inversión (Python)

Suponga que la proyección del flujo de caja de un proyecto de inversión propuesto es la siguiente (con los flujos en miles de dólares):

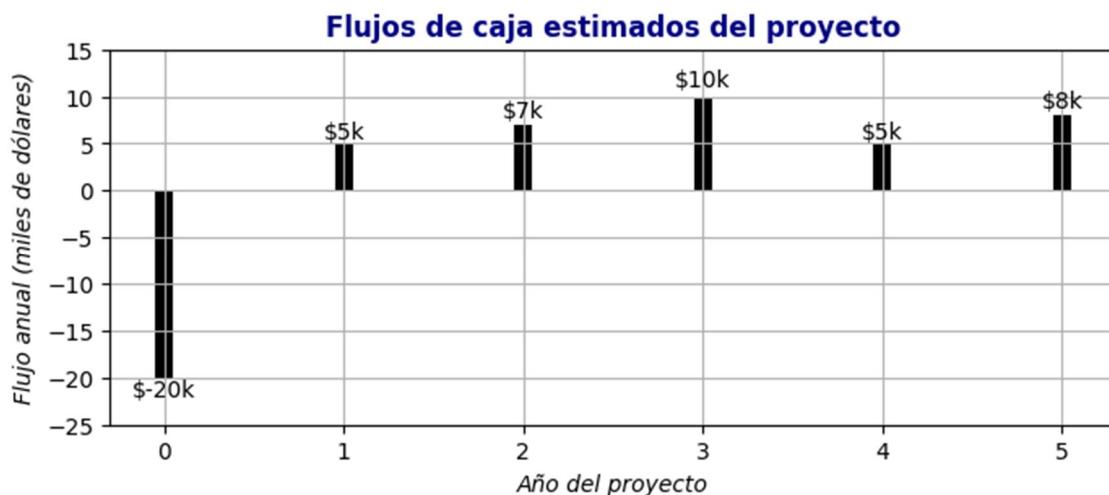


Figura 3.4. Flujos de caja del proyecto analizado.

La ecuación del valor presente neto de la inversión es entonces para una tasa de interés r la siguiente:

$$VPN = -20 + \frac{5}{1+r} + \frac{7}{(1+r)^2} + \frac{10}{(1+r)^3} + \frac{5}{(1+r)^4} + \frac{8}{(1+r)^5}$$

La *tasa interna de retorno* (TIR) del proyecto es la tasa de interés en la cual el VPN es cero (la mínima tasa de interés que hace que el proyecto sea rentable) por lo que la ecuación que tenemos que resolver es para calcularla en este caso es:

$$0 = -20 + \frac{5}{1+r} + \frac{7}{(1+r)^2} + \frac{10}{(1+r)^3} + \frac{5}{(1+r)^4} + \frac{8}{(1+r)^5}$$

Como es algo aburrido derivarla para usar Newton-Raphson usamos entonces el método de la secante programado usando el algoritmo 3.3 en el siguiente código de implementación:

```
def flujo_caja( r ):
    vpn = -20 + 5/(1+r) + 7/(1+r)**2 + 10/(1+r)**3 + 5/(1+r)**4 + 8/(1+r)**5
    return vpn

TIR = metodo_secante( flujo_caja , x0=0.05 , x1=0.10 , it_max=100 , error_max=1e-4 , print_results=True )[0]
```

El programa nos entrega que la $TIR = 0.2095 = \textcolor{red}{20.95\%}$ después de hacer 7 iteraciones con un error absoluto de $3.2\text{e-}6$.

Miremos gráficamente este resultado:

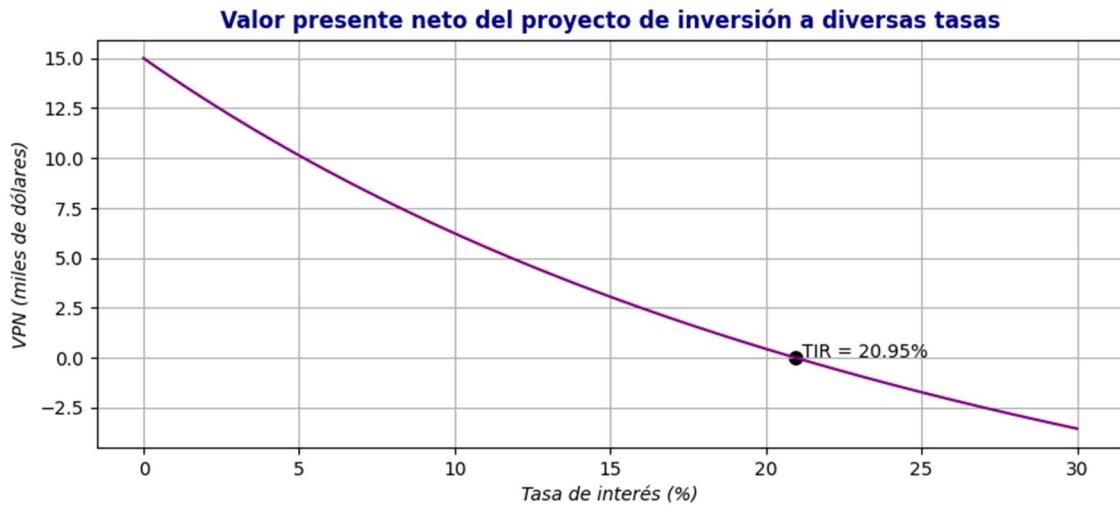


Figura 3.5. VPN para diferentes tasas de interés y la TIR calculada en el ejemplo.

El código de Python para hacer la Figura 3.5 fue el siguiente:

```

tasas = np.linspace( 0 , 30 , 100 )
vpns = np.empty( shape=100 )
for j in range(100):
    vpns[j] = flujo_caja( tasas[j]/100 )

plt.figure( 2, figsize=(10,4) )
plt.plot( tasas , vpns , color='purple' )
plt.scatter( [TIR*100] , [0] , color='black' , s=48 )
plt.text( 1.01*TIR*100 , 0 , 'TIR = '+str( round( TIR*100 , 2 ) )+'%' )
plt.title('Valor presente neto del proyecto de inversión a diversas tasas', color='navy', weight='bold')
plt.ylabel('VPN (miles de dólares)', rotation=90 , style='italic')
plt.xlabel('Tasa de interés (%)', style='italic')
plt.grid()

```

4. Sistemas de ecuaciones lineales

En muchas ocasiones en el ejercicio de la profesión nos encontramos con sistemas de ecuaciones lineales algebraicas, muchos surgen al discretizar ecuaciones diferenciales parciales, en la solución de problemas de optimización y escenarios varios; en la práctica los sistemas pueden ser *muy* grandes y su tratamiento analítico llega a ser engorroso o imposible. En este capítulo presento dos métodos numéricos para solucionar estos problemas, todos los ejemplos son implementados en Excel.

Todo sistema de ecuaciones lineales se puede escribir de forma matricial:

$$[A]\mathbf{x} = \mathbf{b} \quad (4.1)$$

La ecuación (4.1) es para cada fila:

$$\sum_{j=1}^n a_{ij}x_j = b_i \quad (4.2)$$

Despejando la i -ésima incógnita de (4.2) se tiene:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij}x_j \right) \quad (4.3)$$

Los métodos de Jacobi y de Gauss-Seidel presentados en esta sección se basan en la ecuación (4.3), sin embargo, estos métodos sólo convergen y dan resultados cuando la matriz $[A]$ es diagonalmente dominante, es decir, se tiene que cumplir para cada fila de la matriz lo siguiente:

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| \quad (4.4)$$

Al igual que con los métodos de solución numérica de ecuaciones no lineales es necesario hacer una suposición inicial de la solución e iterar; esta iteración se hace hasta que el residual sea menor a una tolerancia establecida a criterio del usuario, el residual que se utiliza generalmente entre iteraciones es:

$$R = \sqrt{\sum_{i=1}^n (x_i^{k+1} - x_i^k)^2} \quad (4.5)$$

4.1. Método de Jacobi

El método de Jacobi consiste en hacer una suposición inicial de la solución e iterar con los valores previos utilizando la ecuación (4.3). La ecuación que describe la iteración k de Jacobi es:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} x_j^k \right) \quad (4.6)$$

4.2. Método de Gauss-Seidel

El método de Gauss-Seidel consiste en hacer una suposición inicial de la solución e iterar utilizando los valores previos de las incógnitas que no se han actualizado y los valores de la iteración actual de las incógnitas que ya se han iterado, todo esto utilizando la ecuación (4.3). La ecuación que describe la iteración k de Gauss-Seidel es:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^n a_{ij} x_j^k \right) \quad (4.7)$$

El método de Gauss-Seidel tiene la ventaja de que converge mucho más rápido que el método de Jacobi, lo cual veremos en el Ejemplo 4.1.

En el algoritmo 4.1 se muestra la solución de un sistema lineal de ecuaciones implementado en Python:

Algoritmo 4.1: Solución sistemas lineales en Python

Entradas: matriz A, vector b

Salidas: valor x tal que Ax=b

```
#Solución directa:
x = np.linalg.solve( A, b )
```

Ejemplo 4.1. Sistema 3x3 (Excel)

Resolver el siguiente sistema de ecuaciones lineales utilizando los métodos de Jacobi y Gauss-Seidel implementados en Excel:

$$\begin{bmatrix} 3 & 1 & 1 \\ 5 & 10 & 4 \\ 5 & -3 & 9 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ 3 \end{bmatrix} \quad (E4.1)$$

La solución en Python se obtiene corriendo el algoritmo 4.1 directamente por la librería de Numpy:

```
#Solución directa:
A = np.array( [ [ 3, 1, 1] , [5, 10, 4], [5, -3, 9] ] )
b = np.array( [ 4, 6, 3 ] )

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

En Excel se obtiene programando una hoja de cálculo (*ver archivo descargable*), los resultados finales se resumen en la siguiente tabla:

EXCEL		
Solución	Jacobi	Gauss
x	1.472	1.472
y	0.051	0.051
z	-0.468	-0.468
Iteraciones	15	15
Residual	9.63E-05	2.58E-08

Se observa que en Python no tenemos que hacer mayor trabajo, lo más interesante es observar los residuales entre el método de Jacobi y el de Gauss-Seidel implementados en Excel, tanto en la tabla como en la Figura 4.1:

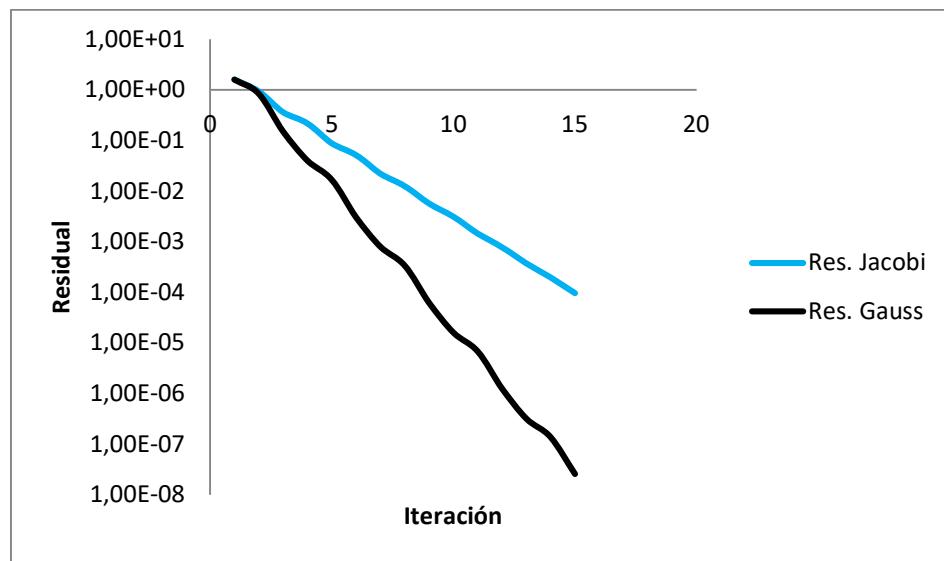


Figura 4.1. Residual en cada iteración.

Es claro que la convergencia y precisión de Gauss-Seidel es mucho mejor que Jacobi y por eso es que recomiendo el primero si es necesario implementar una solución de este tipo en Excel *a falta de acceso a Python*.

Ejemplo 4.2. Análisis de una cercha – Sistema de 8x8 (Python)

Hallaremos las reacciones y las fuerzas internas en los elementos de la siguiente cercha cargada:

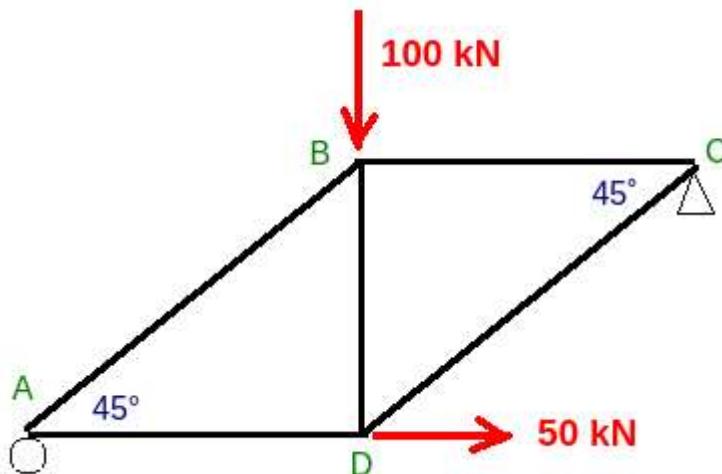


Figura 4.2. Cercha a analizar en el ejemplo.

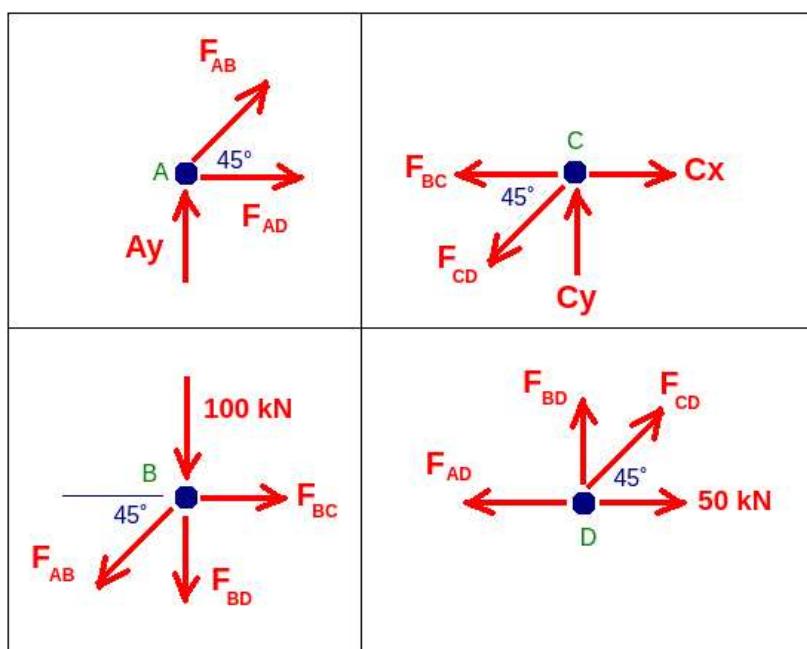


Figura 4.3. Diagrama de cuerpo libre en cada nodo de la cercha.

Haciendo equilibrio analíticamente en x , y para cada nodo se llega a un sistema de ecuaciones lineales de 8x8. La solución implementada en Python usando el algoritmo 4.1 es la siguiente:

```

fuerzas = ['Ay' , 'AB' , 'AD' , 'BC' , 'BD' , 'CD' , 'Cx' , 'Cy' ]

matriz_A = np.array([
    [ 0 , 0.707 , 1 , 0 , 0 , 0 , 0 , 0 ] ,
    [ 1 , 0.707 , 0 , 0 , 0 , 0 , 0 , 0 ] ,
    [ 0 , -0.707 , 0 , 1 , 0 , 0 , 0 , 0 ] ,
    [ 0 , -0.707 , 0 , 0 , -1 , 0 , 0 , 0 ] ,
    [ 0 , 0 , -1 , 0 , 0 , 0.707 , 0 , 0 ] ,
    [ 0 , 0 , 0 , 0 , 1 , 0.707 , 0 , 0 ] ,
    [ 0 , 0 , 0 , -1 , 0 , -0.707 , 1 , 0 ] ,
    [ 0 , 0 , 0 , 0 , 0 , -0.707 , 0 , 1 ] ]
])

vector_b = np.array( [ 0 , 0 , 0 , 100 , -50 , 0 , 0 , 0 ] )

x = np.linalg.solve( matriz_A , vector_b )

for j in range( len( x ) ):
    print( fuerzas[j] + ' = ' , round( x[j] , 3 ) , 'kN' )

```

Que entrega la solución deseada:

Ay	=	75.0 kN
AB	=	-106.082 kN
AD	=	75.0 kN
BC	=	-75.0 kN
BD	=	-25.0 kN
CD	=	35.361 kN
Cx	=	-50.0 kN
Cy	=	25.0 kN

5. Sistemas de ecuaciones no lineales

En ocasiones surgen sistemas de ecuaciones no lineales que deben resolverse, como por ejemplo en intercambio de calor por radiación entre cuerpos. Para solucionar estos problemas existen métodos iterativos, de los cuales presento el método que he utilizado más en mi experiencia, tiene la ventaja de ser muy sencillo de implementar tanto en Python como en Excel, sobre todo en este último.

5.1. Método de punto fijo o iteración directa multivariable

El método de punto fijo multivariable es análogo al caso mostrado en la Sección 3.1, sirve para resolver un sistema de ecuaciones de la forma:

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= x_1 \\ f_2(x_1, x_2, \dots, x_n) &= x_2 \quad (5.1) \\ &\vdots \\ f_n(x_1, x_2, \dots, x_n) &= x_n \end{aligned}$$

El método procede definiendo dos vectores \mathbf{x} y $\mathbf{F}(\mathbf{x})$, tal que

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \quad (5.2)$$

Debe hacerse una aproximación inicial de la solución \mathbf{x}^0 , y luego se hace la iteración

$$\boxed{\mathbf{x}^{k+1} = \mathbf{F}(\mathbf{x}^k)} \quad (5.3)$$

Hasta que el valor residual $R = \|\mathbf{x}^{k+1} - \mathbf{x}^k\|$ sea menor a la tolerancia definida por el usuario. En el algoritmo 5.1 se muestra el algoritmo de punto fijo multivariable en pseudocódigo debido a que no hay forma de escribir un código en Python general para este método.

Ejemplo 5.1. Transferencia de calor (Excel).

Consideremos dos placas paralelas entre sí, la placa superior recibe radiación solar equivalente de 1000 W/m^2 con absorvedad de 0.9 e intercambia calor con el ambiente a 293K con coeficiente de convección de $10 \text{ W/m}^2\text{K}$ y con el cielo a 277K ; la placa inferior intercambia calor por convección a un medio a 300K con coeficiente de convección de $15 \text{ W/m}^2\text{K}$; las emitancias de las placas son de 0.9. Las placas intercambian calor entre ellas por convección a $5 \text{ W/m}^2\text{K}$ y por radiación.

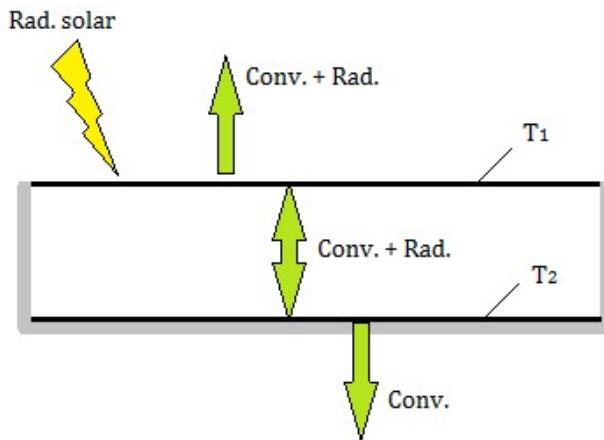


Figura 5.1. Diagrama del sistema.

Las ecuaciones que modelan este sistema son:

$$900 + 10(293 - T_1) + (5.67 \times 10^{-8})(0.9)(277^4 - T_1^4) + (5.67 \times 10^{-8})(0.45)(T_2^4 - T_1^4) + 5(T_2 - T_1) = 0 \quad (E5.1)$$

$$(5.67 \times 10^{-8})(0.45)(T_1^4 - T_2^4) + 5(T_1 - T_2) + 15(300 - T_2) = 0 \quad (E5.2)$$

Calcular las temperaturas de las placas con las ecuaciones E5.1 y E5.2. Despejando de E5.1 y E5.2 tenemos las ecuaciones para iterar:

$$T_1^{k+1} = \frac{1}{15} \left(4130.43 + 5T_2^k + 5.67 \times 10^{-8} \left[0.45(T_2^k)^4 - 1.35(T_1^k)^4 \right] \right)$$

$$T_2^{k+1} = \frac{1}{20} \left(4500 + 5T_1^k + (0.45)(5.67 \times 10^{-8}) \left[(T_1^k)^4 - (T_2^k)^4 \right] \right)$$

Con la suposición inicial:

Suposición inicial	
T1 [K]	T2 [K]
-11	0

Tenemos los resultados de 100 iteraciones:

Resultados			
T1 [K]	T2 [K]	Residual	Iteraciones
332.82	311.76	9.34E-02	100

Nótese que el residual no es un valor aceptable típicamente, dado que se buscan al menos de órdenes de 10^{-3} y entre menores mejor.

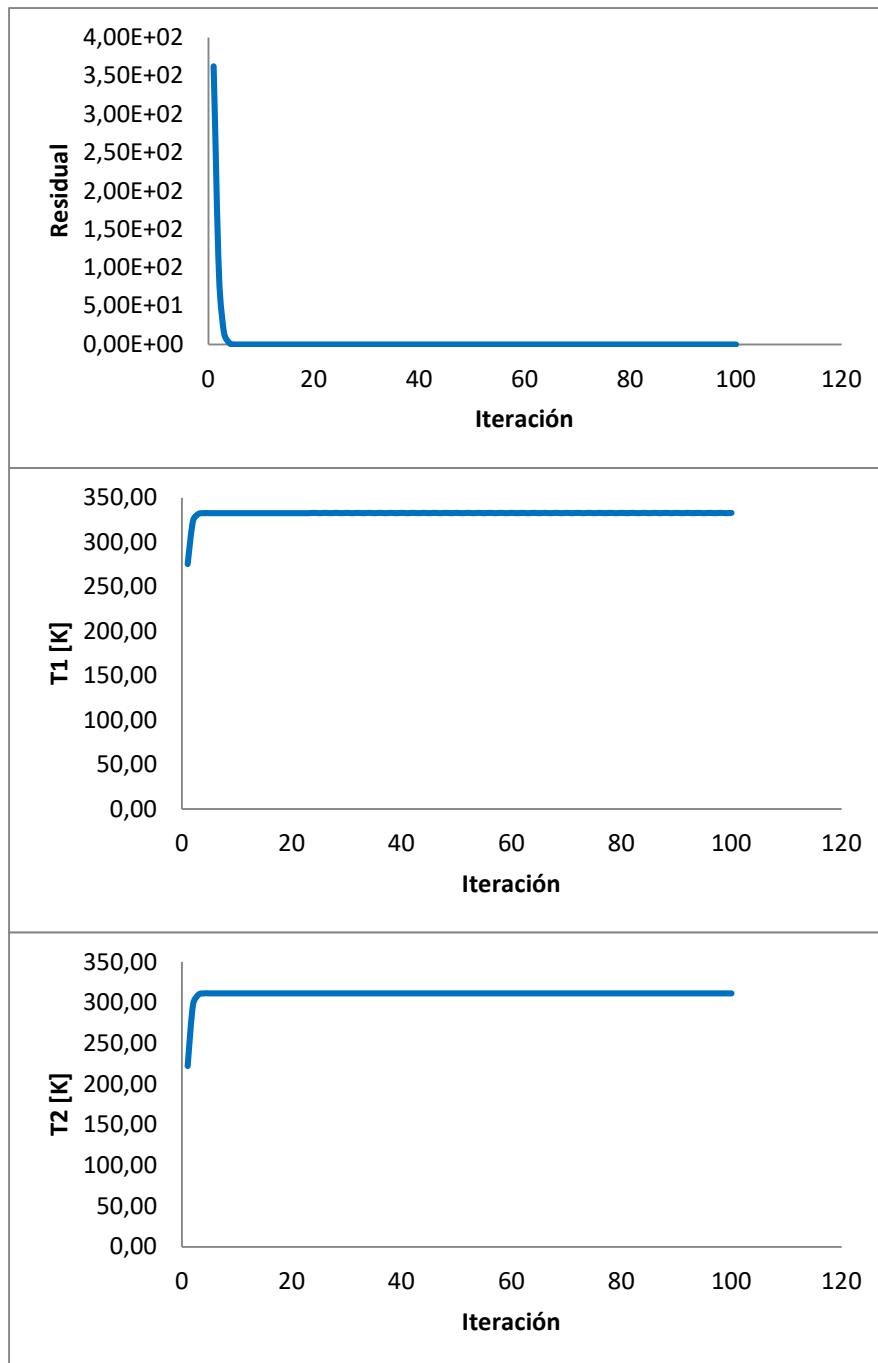


Figura 5.2. Evolución de las iteraciones.

Hay un asunto interesante acá, y es que **la suposición inicial no tiene sentido físico** (no existe en el Universo -11 K de temperatura), de hecho, en este caso la suposición inicial se hizo a ensayo y error para que el sistema no se hiciera inestable. Por ejemplo, con la siguiente suposición inicial más razonable de:

Suposición inicial	
T1 [K]	T2 [K]
300	300

El sistema no converge, como se puede observar en las siguientes gráficas de evolución de las iteraciones:

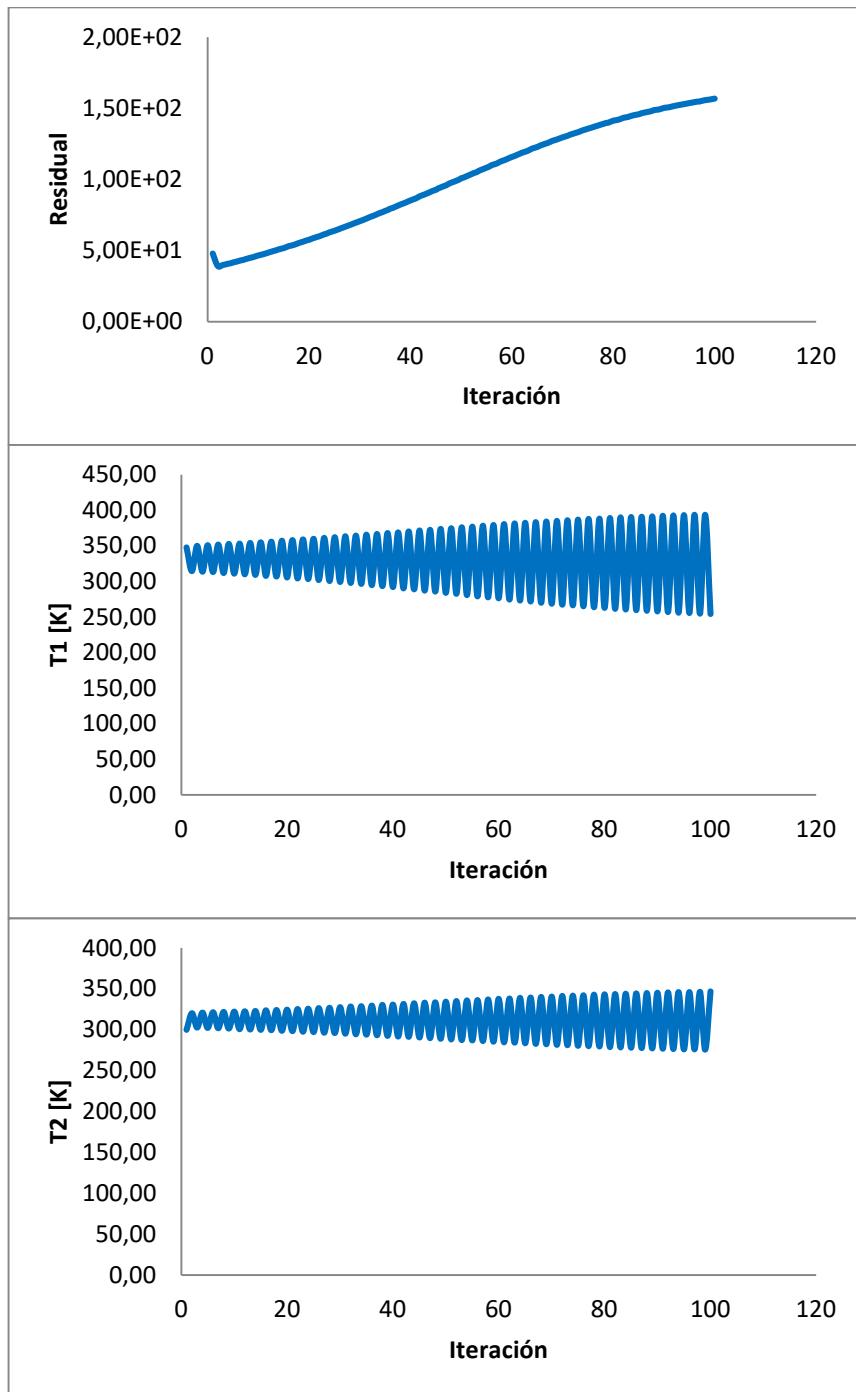


Figura 5.3. Sistema inestable debido a suposición inicial.

¿Hay una forma de hacer más estable el método? La respuesta es SÍ, y es utilizando un método de punto fijo modificado.

5.2. Método de punto fijo o iteración directa actualizada

Es la misma idea del método de Gauss-Seidel (Sección 4.2) de utilizar en la iteración actual las variables ya actualizadas. La iteración se hace de la siguiente forma en este caso:

$$\begin{aligned}x_1^{k+1} &= f_1(x_1^k, x_2^k, \dots, x_n^k) \\x_2^{k+1} &= f_2(x_1^{k+1}, x_2^k, \dots, x_n^k) \quad (5.4) \\&\vdots \\x_n^{k+1} &= f_n(x_1^{k+1}, x_2^{k+1}, \dots, x_n^k)\end{aligned}$$

Hasta que el valor residual $R = \|\mathbf{x}^{k+1} - \mathbf{x}^k\|$ sea menor a la tolerancia definida por el usuario. Este método así presenta convergencia mucho más elevada que el mostrado en la Sección 5.1 y es el que personalmente recomiendo para la solución de sistemas de ecuaciones.

Ejemplo 5.2. El ejemplo 5.1 con el método modificado (Excel).

Volvemos a solucionar el ejemplo 5.1, las iteraciones son:

$$T_1^{k+1} = \frac{1}{15} \left(4130.43 + 5T_2^k + 5.67 \times 10^{-8} [0.45(T_2^k)^4 - 1.35(T_1^k)^4] \right)$$

$$T_2^{k+1} = \frac{1}{20} \left(4500 + 5T_1^{k+1} + (0.45)(5.67 \times 10^{-8}) [(T_1^{k+1})^4 - (T_2^k)^4] \right)$$

Con la suposición inicial:

Suposición inicial	
T1 [K]	T2 [K]
300	300

Los resultados son los siguientes:

Resultados			
T1 [K]	T2 [K]	Residual	Iteraciones
332.78	311.79	1.14E-13	100

Se observa claramente en esta última tabla de resultados y en las gráficas de la Figura 5.4 que el sistema es muy estable y converge de una manera mucho más rápida.

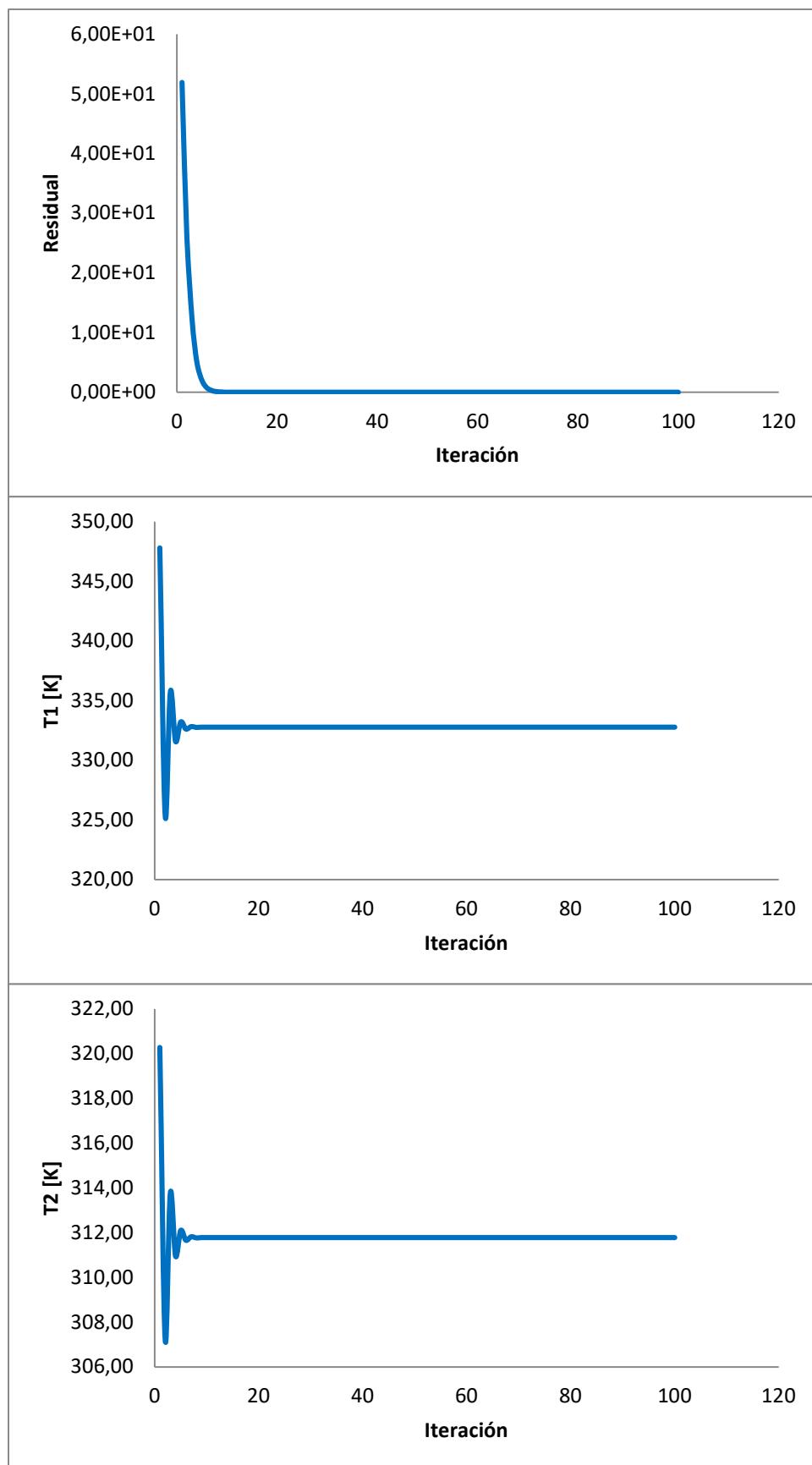


Figura 5.4. Evolución de las iteraciones.

5.3. Método de Newton-Raphson multivariable

Se utiliza cuando tenemos un sistema de n ecuaciones igualadas a cero y se desea resolver:

$$f_1(x_1, x_2, \dots, x_n) = 0$$

$$f_2(x_1, x_2, \dots, x_n) = 0 \quad (5.5)$$

⋮

$$f_n(x_1, x_2, \dots, x_n) = 0$$

Lo que se tiene es entonces dos vectores, el vector \mathbf{x} de las incógnitas y el vector \mathbf{F} de las funciones de estas incógnitas:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

El *jacobiano* de una función vectorial de varias variables es la derivada del vector de funciones respecto al vector de variables, lo cual da una matriz cuadrada para un sistema de $n \times n$:

$$J(\mathbf{x}) = \frac{\partial \mathbf{F}}{\partial \mathbf{x}}$$

O en forma de índices para la fila i y la columna j se tiene:

$$J_{ij} = \frac{\partial f_i}{\partial x_j} \quad (5.6)$$

Usando la derivada en su contexto de cambio tenemos entonces de la definición del jacobiano para una iteración k :

$$\mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k) \approx J(\mathbf{x}_k)(\mathbf{x}_{k+1} - \mathbf{x}_k)$$

Como se busca el cero en $\mathbf{F}(\mathbf{x}_{k+1}) = \mathbf{0}$ y despejando el vector de solución de la ecuación aproximada se tiene entonces la iteración de Newton-Raphson multivariable:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - J(\mathbf{x}_k)^{-1}\mathbf{F}(\mathbf{x}_k)$$

(5.7)

Nótese que en la ecuación 5.7 ¡la matriz jacobiana está invertida!

Ejemplo 5.3. El ejemplo 5.1 con Newton-Raphson (Python).

Volvemos a solucionar el ejemplo 5.1, las ecuaciones a resolver son:

$$900 + 10(293 - T_1) + (5.67 \times 10^{-8})(0.9)(277^4 - T_1^4) + (5.67 \times 10^{-8})(0.45)(T_2^4 - T_1^4) + 5(T_2 - T_1) = 0 \quad (E5.1)$$

$$(5.67 \times 10^{-8})(0.45)(T_1^4 - T_2^4) + 5(T_1 - T_2) + 15(300 - T_2) = 0 \quad (E5.2)$$

En la lógica del método de Newton-Raphson multivariable las funciones vectoriales son entonces:

$$\begin{aligned} f_1(T_1, T_2) &= 900 + 10(293 - T_1) + (5.67 \times 10^{-8})(0.9)(277^4 - T_1^4) \\ &\quad + (5.67 \times 10^{-8})(0.45)(T_2^4 - T_1^4) + 5(T_2 - T_1) \end{aligned}$$

$$f_2(T_1, T_2) = (5.67 \times 10^{-8})(0.45)(T_1^4 - T_2^4) + 5(T_1 - T_2) + 15(300 - T_2) \quad (E5.3)$$

La matriz jacobiana es entonces con las derivadas parciales:

$$J(T_1, T_2) = \begin{bmatrix} -15 - (3.0618 \times 10^{-7})T_1^3 & 5 + (1.0206 \times 10^{-7})T_2^3 \\ 5 + (1.0206 \times 10^{-7})T_1^3 & -20 - (1.0206 \times 10^{-7})T_2^3 \end{bmatrix}$$

Implementamos el método en este caso con el siguiente código de Python:

```
it_max = 100
error_max = 1e-6
error = 1
it = 1
x = np.array( [ 300, 300 ] ) #Inicialización de la solución
J = np.empty( shape=(2,2) )
F = np.empty( shape=2 )
sw = 1
while sw==1:
    #Cálculo de la función:
    F[0] = 900 + 10*(293-x[0]) + 5.67e-8*0.9*(277**4-x[0]**4) + 5.67e-8*0.45*(x[1]**4 - x[0]**4) + 5*( x[1] - x[0] )
    F[1] = 5.67e-8*0.45*(x[0]**4 - x[1]**4) + 5*( x[0] - x[1] ) + 15*( 300 - x[1] )
    #Cálculo del jacobiano:
    J[0,0] = -15-3.0618e-7*x[0]**3
    J[0,1] = 5+1.0206e-7*x[1]**3
    J[1,0] = 5+1.0206e-7*x[0]**3
    J[1,1] = -20-1.0206e-7*x[1]**3
    #Iteración:
    xf = x - np.linalg.solve( J , F ) #Aplicación del método
    error = np.linalg.norm( x - xf ) #Error entre iteraciones
    it += 1 #Actualización de número de iteraciones
    x = xf
    if error<=error_max or it>=it_max:
        sw = 0

print('Solución:')
print( 'T1 = ', round( x[0] , 3 ) , 'K' )
print( 'T2 = ', round( x[1] , 3 ) , 'K' )
print('')
print('      Error = ', np.format_float_scientific( error, 3 ) )
print('Iteraciones = ', it )
```

Los resultados que entrega son los siguientes:

```
Solución:  
T1 = 332.781 K  
T2 = 311.786 K  
  
Error = 1.134e-07  
Iteraciones = 5
```

Nótese que este método converge mucho más rápidamente que el de punto fijo modificado, sin embargo, no siempre es así, la rapidez de convergencia y precisión de cada método depende del sistema de ecuaciones a ser resuelto y debe mirarse en cada caso particular cuál es el mejor (o más fácil/práctico de implementar).

(Volvió a sobrar espacio...)



6. Derivación e integración numéricas

6.1. Derivación por diferencias finitas

La diferenciación numérica es muy útil en casos en los cuales se tiene una función que es muy engorrosa de derivar, o en casos en los cuales no se tiene una función explícita sino una serie de datos experimentales.

Para entender de una manera sencilla la discretización por diferencias finitas de una derivada debe tenerse en cuenta la interpretación geométrica de la derivada en un punto, que es la pendiente de la curva en el punto de interés. Considerense tres puntos intermedios en una curva como se muestra en la figura 6.1:

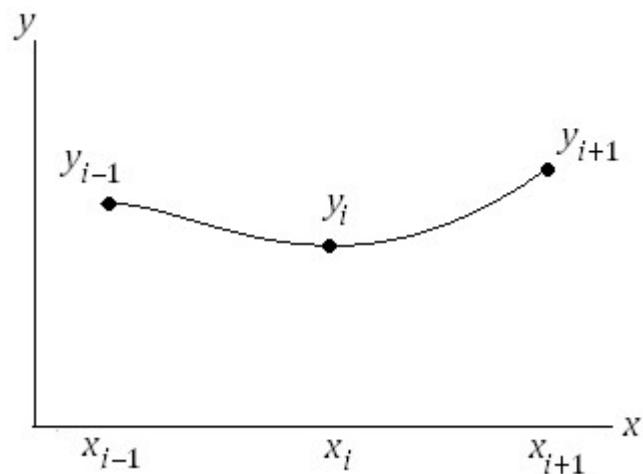


Figura 6.1. Curva discretizada.

Supóngase que interesa la derivada en el punto (x_i, y_i) , tres formas de aproximar la pendiente por recta en ese punto son:

$$\left. \frac{dy}{dx} \right|_{x=x_i} \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \quad (6.1)$$

$$\left. \frac{dy}{dx} \right|_{x=x_i} \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \quad (6.2)$$

$$\left. \frac{dy}{dx} \right|_{x=x_i} \approx \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}} \quad (6.3)$$

Las ecuaciones 6.1 a 6.3 son llamadas *diferencias finitas*. La ecuación 6.1 se recomienda para hallar la derivada del punto inicial de una curva, la ecuación 6.2 se recomienda para hallar la derivada del punto final de una curva, y la ecuación

6.3 es la ecuación de *diferencias finitas centrales*, y se recomienda para hallar la derivada en los puntos intermedios de una curva.

En el caso cuando las diferencias $x_i - x_{i-1} = x_{i+1} - x_i = \Delta x$ son constantes para todo el dominio, las ecuaciones de diferencias finitas quedan

$$\left. \frac{dy}{dx} \right|_{x=x_i} \approx \frac{y_{i+1} - y_i}{\Delta x} \quad (6.4)$$

$$\left. \frac{dy}{dx} \right|_{x=x_i} \approx \frac{y_i - y_{i-1}}{\Delta x} \quad (6.5)$$

$$\left. \frac{dy}{dx} \right|_{x=x_i} \approx \frac{y_{i+1} - y_{i-1}}{2\Delta x} \quad (6.6)$$

La ecuación 6.4 se recomienda para hallar la derivada del punto inicial de una curva, la ecuación 6.5 se recomienda para hallar la derivada del punto final de una curva, y la ecuación 6.6 se recomienda para hallar la derivada en los puntos intermedios de una curva.

El método de derivación por diferencias finitas implementado en Python se muestra en el algoritmo 6.1.

Algoritmo 6.1: Derivación numérica en Python

```
def derivada_numerica(X,Y):
    """
    X, Y: datos de entrada
    Entrega el valor de la derivada para cada punto de X
    """
    N=len(X)
    df = np.empty( shape=N )
    df[0]=( Y[1]-Y[0] )/( X[1]-X[0] )
    df[N-1]=( Y[N-1]-Y[N-2] )/( X[N-1]-X[N-2] )
    for n in range(1, N-1):
        df[n]=( Y[n+1]-Y[n-1] )/( X[n+1]-X[n-1] )

    return df
```

La derivación numérica también puede implementarse de forma muy sencilla en tablas de Excel.

Ejemplo 6.1. Módulo de elasticidad según ensayo de tensión (Python)

El módulo de elasticidad de un material es la pendiente de la zona elástica (la parte lineal) de la curva esfuerzo-deformación hallada mediante ensayo de tensión. Implementamos el algoritmo 6.1 con los datos de ensayo de tensión del ejemplo 1.3 usando el siguiente código:

```
deriv_ED = derivada_numérica(deformacion, esfuerzo)

plt.figure( 1, figsize=(10,4) )
plt.plot( deformacion, deriv_ED, color='purple', label='Derivada numérica' )
plt.title('Derivada numérica curva esfuerzo-deformación', color='navy', weight='bold')
plt.ylabel('dσ/dε (psi)', rotation=90, style='italic')
plt.xlabel('Deformación (in/in)', style='italic')
plt.legend( ncol=2 )
plt.text( 0.005 , 0.8e7 ,
    'Módulo de elasticidad zona elástica:\nE = ' +str( np.format_float_scientific( np.mean( deriv_ED[:4] ) , 3 ) )+' psi' , bbox={'alpha':1, 'facecolor':'white'} )
plt.grid()
```

La gráfica resultante es la siguiente:

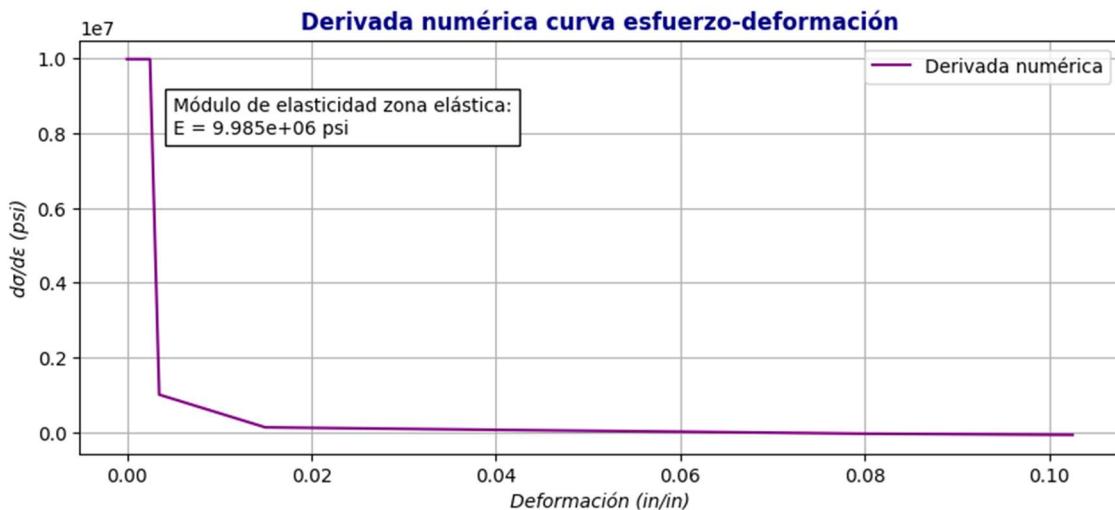


Figura 6.2. Derivada numérica del ensayo de tensión, la zona elástica es donde la derivada es constante (al inicio de la curva) y el módulo de elasticidad es ese valor constante de la derivada en esa región.

6.2. Integración por método de los trapecios

La integración numérica es muy útil en casos en los cuales se tiene una función que es muy engorrosa de integrar o que no posee antiderivada, o en casos en los cuales no se tiene una función explícita sino una serie de datos experimentales. Aunque hay varios métodos de integración numérica, acá solo se mostrará el método de los trapecios, ya que es el más sencillo de implementar y de entender.

Para entender el método de los trapecios debe tomarse en cuenta la interpretación geométrica de una integral como área bajo la curva, siendo así, considérese el área de un trapecio entre dos puntos de una curva como se muestra en la figura 6.6.

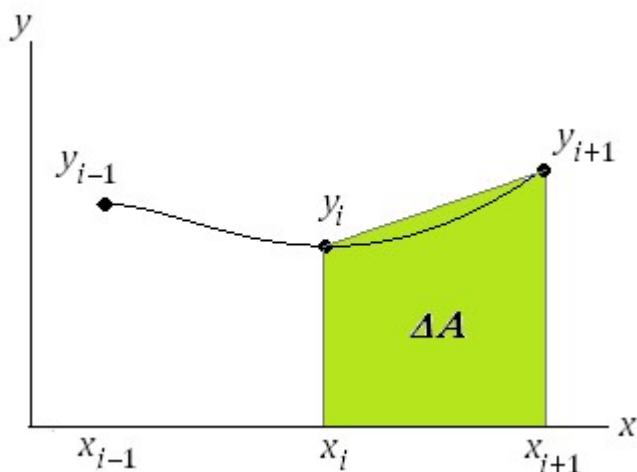


Figura 6.3. Área de un trapecio entre dos puntos de una curva.

El área del trapecio es:

$$\Delta A_{i+1} = \frac{1}{2}(x_{i+1} - x_i)(y_{i+1} + y_i) \quad (6.7)$$

El valor de la integral en un intervalo con $n+1$ puntos x_0 a x_n es entonces la suma de las distintas áreas por sub-intervalo:

$$I = \int_{x_0}^{x_n} y(x) dx \approx \sum_{i=0}^{n-1} \Delta A_i \quad (6.8)$$

En el caso que el tamaño de subintervalo sea un valor Δx constante, la ecuación 6.8 resulta

$$I = \int_{x_0}^{x_n} y(x) dx \approx \frac{1}{2}\Delta x(y_0 + y_n) + \Delta x \sum_{i=1}^{n-1} y_i \quad (6.9)$$

Otra forma de verlo, y más fácil de programar en una hoja de Excel, es la siguiente: el valor acumulado de la integral en el intervalo i (notado como I_i) es

$$I_i = I_{i-1} + \frac{1}{2}(x_i - x_{i-1})(y_i + y_{i-1}) \quad (6.10)$$

Y el valor I de la integral en el dominio de interés es el valor final acumulado de la ecuación 6.10. La ecuación 6.10 puede ponerse fácilmente en términos de fórmula de celdas en una hoja de Excel, descargar el archivo de Excel de la página donde se descarga el libro para ver el ejemplo 6.2 resuelto en Excel.

La implementación en Python del método de los trapecios con la ecuación 6.10 se muestra en el algoritmo 6.2.

Algoritmo 6.2: Método de los trapecios en Python

```
def integral_trapezoidal(I0, X, Y):
    """
    I0: valor inicial de la integral
    X, Y: datos de entrada
    Entrega el valor I de la integral
    """

    N=len(X)
    I=I0
    for n in range(1,N):
        I += 0.5*(Y[n]+Y[n-1])*(X[n]-X[n-1])

    return I
```

Ejemplo 6.2. Ecuación de Paris en mecánica de la fractura (Excel)

El crecimiento de una grieta en el borde de una placa por ciclo de esfuerzos viene dado por la ecuación de Paris

$$\frac{da}{dN} = A(\Delta\sigma Y \sqrt{a})^m \quad (E6.1)$$

Donde N es el número de ciclos, A y m son constantes del material, $\Delta\sigma$ es la diferencia de esfuerzos a tensión sobre la pieza y Y viene dado por la ecuación E3.2. Cuando se observa una grieta de tamaño a_0 , el número de ciclos restante para fractura catastrófica de la pieza se obtiene separando las variables e integrando la ecuación E6.1:

$$N_f = \int_{a_0}^{a_f} \frac{da}{A(Y\Delta\sigma\sqrt{a})^m} \quad (E6.2)$$

Supóngase que se tiene la placa del ejemplo 3.1 con $\Delta\sigma = 17.78 \text{ ksi}$, $A = 6.6 \times 10^{-9}$, $m = 2.25$, con una grieta inicial de $a_0 = 0.25 \text{ in}$. El número de ciclos restante para falla es

$$N_f = \int_{0.25}^{0.62} \frac{da}{6.6 \times 10^{-9} \left(17.78 \left[1.99 - 0.41 \left(\frac{a}{2.5} \right) + 18.70 \left(\frac{a}{2.5} \right)^2 - 38.48 \left(\frac{a}{2.5} \right)^3 + 53.85 \left(\frac{a}{2.5} \right)^4 \right] \sqrt{a} \right)^{2.25}}$$

Aplicando la ecuación (6.2) en una hoja de Excel tenemos la gráfica de crecimiento de la grieta:

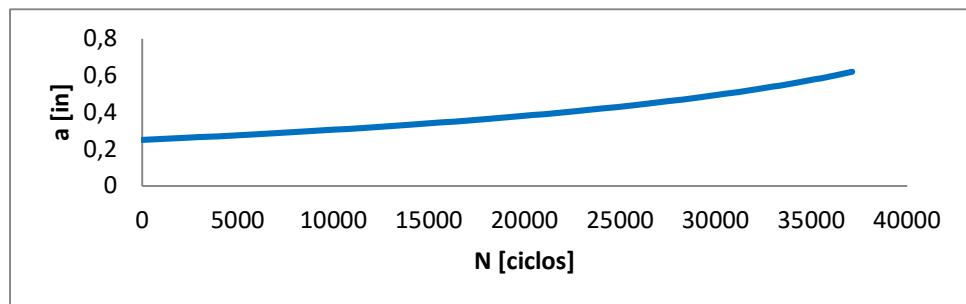


Figura 6.4. Crecimiento de grieta calculado por integración numérica.

Según el valor final de la integral los ciclos de falla son:

$$N_f = 37120$$

Ejemplo 6.3. Resiliencia según ensayo de tensión (Python)

La resiliencia de un material es la cantidad de energía por unidad de volumen que éste es capaz de absorber antes de su ruptura, gráficamente es el área bajo la curva de esfuerzo-deformación, aplicando el algoritmo 6.2 a los datos del ensayo de tensión que hemos estado usando de ejemplo tenemos:

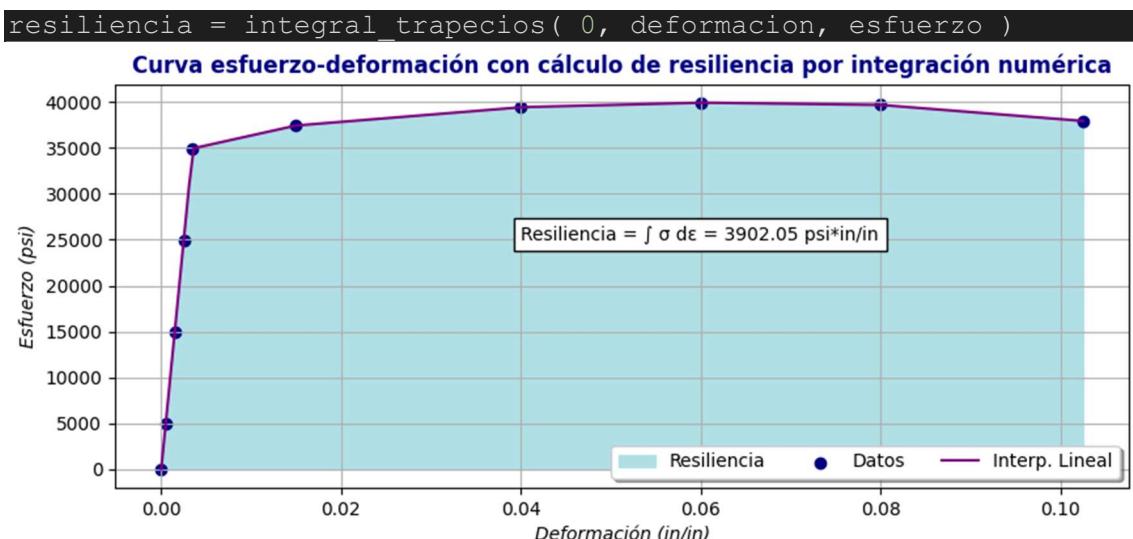


Figura 6.5. Ensayo de tensión con la resiliencia calculada por método de trapecios.

7. Ecuaciones diferenciales con valor inicial

Los métodos numéricos para ecuaciones diferenciales que se presentan aplican para ecuaciones diferenciales ordinarias de primer orden con condiciones iniciales en la siguiente forma:

$$\frac{dy}{dx} = f(x, y) \quad (7.1)$$

$$y(x_0) = y_0$$

Estos métodos son muy útiles cuando se tienen ecuaciones diferenciales que no pueden resolverse por los métodos analíticos o cuya solución analítica es muy engorrosa.

Nota: estos métodos son generalizables a sistemas de ecuaciones diferenciales, aplicando la discretización o método de solución a cada ecuación del sistema por separado y uniendo las soluciones a medida que se hallan.

7.1. Método de Euler

El método de Euler consiste en aproximar la derivada de la ecuación 7.1 por diferencias finitas como en la ecuación 6.4, entonces la ecuación diferencial resulta

$$\frac{y_{n+1} - y_n}{\Delta x} = f(x_n, y_n)$$

Por lo cual el valor de la función en el intervalo de tiempo $n+1$ es

$$y_{n+1} = y_n + \Delta x \cdot f(x_n, y_n) \quad (7.2)$$

El método de Euler tiene la desventaja de que se vuelve inestable y la solución diverge si el tamaño de paso de tiempo Δx es muy grande.

En el algoritmo 7.1 se muestra el algoritmo del método de Euler en Python:

Algoritmo 7.1: Método de Euler

```
#Método de Euler:
def metodo_euler( f:'funcion', x0:float, y0:float, dx:float, N:int ):
    """
    f: función f(x,y) previamente definida
    x0: valor inicial variable independiente
    y0: valor inicial variable dependiente
    dx: tamaño de paso variable independiente
    N: número de puntos para la solución
    """

    x = np.empty( shape=N )
    y = np.empty( shape=N )
    y[0] = y0
    x[0] = x0
    for n in range(1,N):
        y[n] = y[n-1] + dx*f( x[n-1] , y[n-1] )
        x[n] = x[n-1] + dx
    return x, y
```

Ejemplo 7.1. Mecánica de la fractura (Excel)

El crecimiento de una grieta en el borde de una placa por ciclo de esfuerzos viene dado por la ecuación de Paris (E6.1). Una forma de estimar el crecimiento de una grieta con el número de ciclos diferente a la integración numérica del ejemplo 6.2 es resolviendo la ecuación E6.1 por el método de Euler, en este caso se tiene la ecuación diferencial discretizada

$$\frac{a_{n+1} - a_n}{\Delta N} = A(\Delta\sigma)^m a_k^{m/2} \left[1.99 - 0.41 \left(\frac{a_n}{2.5} \right) + 18.7 \left(\frac{a_n}{2.5} \right)^2 - 38.48 \left(\frac{a_n}{2.5} \right)^3 + 53.85 \left(\frac{a_k}{2.5} \right)^4 \right]^m$$

Y despejando a_{n+1} se tiene el valor del tamaño de la grieta al siguiente ciclo de carga

$$a_{n+1} = a_n + \Delta N A(\Delta\sigma)^m a_k^{\frac{m}{2}} \left[1.99 - 0.41 \left(\frac{a_n}{2.5} \right) + 18.7 \left(\frac{a_n}{2.5} \right)^2 - 38.48 \left(\frac{a_n}{2.5} \right)^3 + 53.85 \left(\frac{a_k}{2.5} \right)^4 \right]^m$$

(E7.1)

Implementando en una hoja de Excel la ecuación (E7.1) tenemos la curva de crecimiento de grieta (Figura 7.12) y la solución del número de ciclos para falla:

$$N_f = 37100$$

Compare este resultado y la curva de crecimiento con los calculados por integración numérica en el ejemplo 6.2.

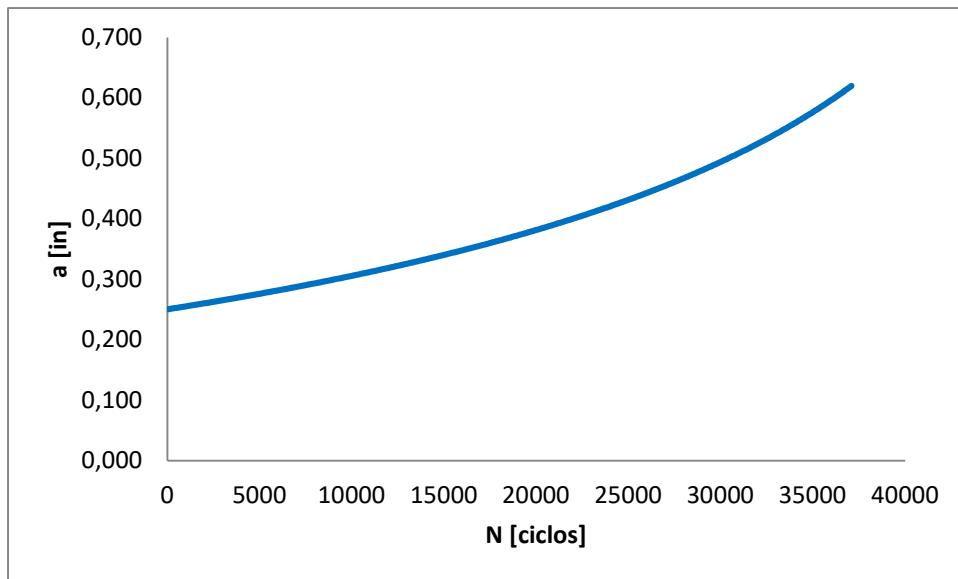


Figura 7.1. Crecimiento de grieta calculado con el método de Euler.

7.2. Método de Runge-Kutta de 4º orden

Uno de los métodos más utilizados para resolver numéricamente problemas de ecuaciones diferenciales ordinarias con condiciones iniciales es el método de Runge-Kutta de cuarto orden, el cual proporciona un pequeño margen de error con respecto a la solución real del problema y es fácilmente programable en un software para realizar las iteraciones necesarias. Hay variaciones en el método de Runge-Kutta de cuarto orden pero el más utilizado es el método en el cual se elige un tamaño de paso Δx y un número máximo de iteraciones N tal que

$$k_1 = \Delta x \cdot f(x_k, y_k)$$

$$k_2 = \Delta x \cdot f\left(x_k + \frac{\Delta x}{2}, y_k + \frac{k_1}{2}\right)$$

$$k_3 = \Delta x \cdot f\left(x_k + \frac{\Delta x}{2}, y_k + \frac{k_2}{2}\right)$$

$$k_4 = \Delta x \cdot f(t_k + \Delta x, y_k + k_3)$$

$$y_{k+1} = y_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Para $k = 0, \dots, N-1$. La solución se da a lo largo del intervalo $(x_0, x_0 + \Delta x \cdot N)$. En el algoritmo 7.2 se muestra el algoritmo del método de Runge-Kutta de orden 4 en Python:

Algoritmo 7.2: Método de Runge-Kutta de orden 4

```
#Método de Runge-Kutta orden 4:
def metodo_rk4( f:'funcion', x0:float, y0:float, dx:float, N:int ):
    """
    f: función f(x,y) previamente definida
    x0: valor inicial variable independiente
    y0: valor inicial variable dependiente
    dx: tamaño de paso variable independiente
    N: número de puntos para la solución
    """

    x = np.empty( shape=N )
    y = np.empty( shape=N )
    y[0] = y0
    x[0] = x0
    for n in range(1,N):
        k1 = dx*f( x[n-1] , y[n-1] )
        k2 = dx*f( x[n-1]+dx/2 , y[n-1]+k1/2 )
        k3 = dx*f( x[n-1]+dx/2 , y[n-1]+k2/2 )
        k4 = dx*f( x[n-1]+dx , y[n-1]+k3 )
        y[n] = y[n-1] + 1/6*( k1 + 2*k2 + 2*k3 + k4 )
        x[n] = x[n-1] + dx
    return x, y
```

Ejemplo 7.2. Velocidad en medios con arrastre (Excel)

La ecuación diferencial que rige la velocidad v de un cuerpo de masa m y área proyectada A que cae en un medio de densidad ρ es

$$\frac{dv}{dt} = g - \frac{\rho A v^2}{2m} \quad (E7.2)$$

El cuerpo adquiere su velocidad terminal de caída cuando no acelera más, es decir, la derivada de la velocidad es cero. De acuerdo a E7.2, la velocidad terminal teórica es

$$v_{f,teórica} = \sqrt{\frac{2mg}{\rho A}} \quad (E7.3)$$

Supóngase una moneda con $m = 0.010\text{kg}$ y $A = 3.1416 \times 10^{-4} \text{ m}^2$, que cae de un edificio, entonces $\rho = 1\text{kg/m}^3$. La velocidad terminal según E7.3 es $v_{f,teórica} = 24.98\text{m/s}$. La iteración de Runge-Kutta se hace como sigue para este caso particular:

$$k_1 = \Delta t [9.8 - 0.0157 v_n^2]$$

$$k_2 = \Delta t \left[9.8 - 0.0157 \left(v_n + \frac{k_1}{2} \right)^2 \right]$$

$$k_3 = \Delta t \left[9.8 - 0.0157 \left(v_n + \frac{k_2}{2} \right)^2 \right]$$

$$k_4 = \Delta t [9.8 - 0.0157 (v_n + k_3)^2]$$

$$v_{n+1} = v_n + \frac{1}{6} (k_1 + 2k_2 + 2k_3 + k_4)$$

Tomando intervalo $\Delta t=1\text{s}$ y velocidad inicial nula, se tiene el método implementado en una hoja de Excel con los valores en la tabla, y se muestra cómo se desarrolla la velocidad en la figura 7.2:

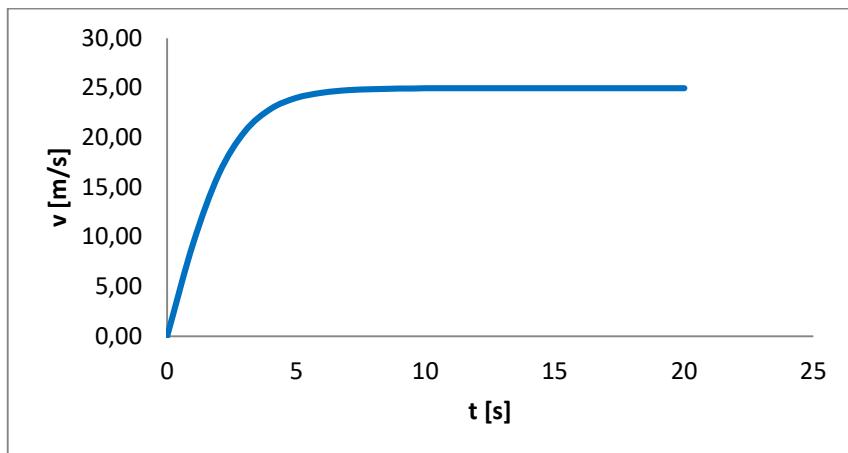


Figura 7.2. Velocidad de la moneda a medida que cae.

Ejemplo 7.3. Velocidad en medios con arrastre (Python)

La ecuación diferencial que estamos trabajando es la misma del ejemplo inmediatamente anterior:

$$\frac{dv}{dt} = 9.8 - 0.0157 v^2$$

Tomando intervalo $\Delta t=0.1\text{s}$ y velocidad inicial nula e implementando los métodos de Euler y Runge-Kutta en Python con los algoritmos previamente mostrados tenemos:

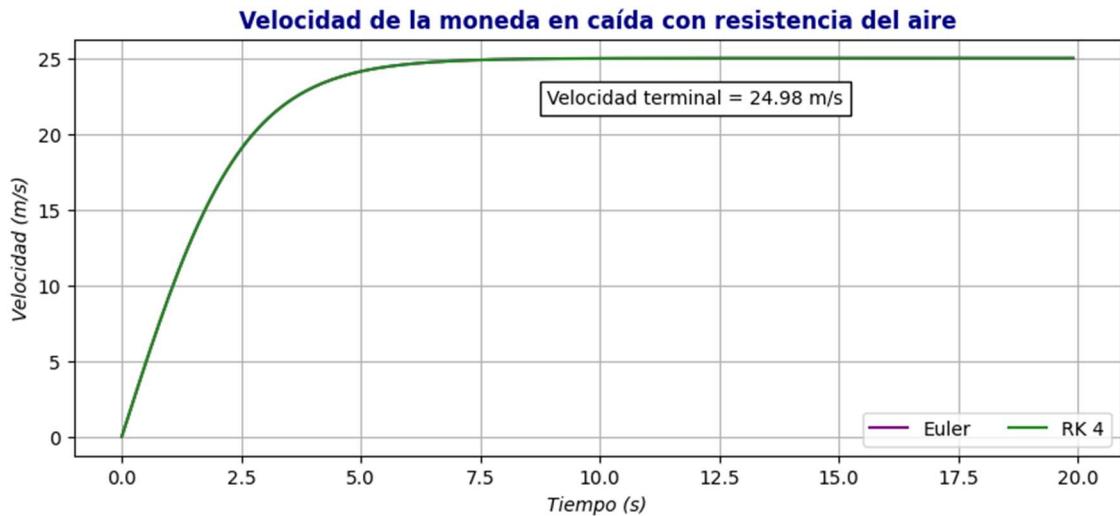


Figura 7.3. Solución de la moneda que cae en Python con $\Delta t=0.1s$, las soluciones dadas por Euler y Runge-Kutta en este caso son idénticas.

Nótese que usamos un intervalo mucho más pequeño que en el mismo ejemplo hecho en Excel y por lo tanto una mucha mejor precisión en la solución sin tener que hacer mayor esfuerzo (una de las ventajas de usar Python).

El código implementado fue el siguiente:

```

def f_arrastre( t, v ):
    res = 9.8 - 0.0157*v**2
    return res

t, v = metodo_euler( f_arrastre, 0 , 0, 0.1, 201 )
t_rk, v_rk = metodo_euler( f_arrastre, 0 , 0, 0.1, 201 )

plt.figure( 1, figsize=(10,4) )
plt.plot( t , v , color='purple', label='Euler' )
plt.plot( t_rk , v_rk , color='forestgreen', label='RK 4' )
plt.title('Velocidad de la moneda en caída con resistencia del aire', color='navy', weight='bold')
plt.ylabel('Velocidad (m/s)', rotation=90 ,style='italic')
plt.xlabel('Tiempo (s)', style='italic')
plt.legend( ncol=2 )
plt.text( 12, 22, 'Velocidad terminal = '+str( round( v[ len(v)-1 ] , 2 ) )+' m/s', ha='center' , bbox={'alpha':1 , 'facecolor':'white'} )
plt.grid()

```

8. Ecuaciones diferenciales con valores en la frontera

El método numérico para ecuaciones diferenciales que se presenta aplica para ecuaciones diferenciales de segundo orden en la forma:

$$\frac{d^2y}{dx^2} = f\left(x, y, \frac{dy}{dx}\right) \quad (8.1)$$

En el dominio $x = [0, L]$ y con condiciones de frontera diversas tales como valor de la primera derivada de y o valores de la función y conocidas en las fronteras 0 y L .

La solución numérica muy útil cuando se tienen ecuaciones diferenciales que no pueden resolverse por los métodos analíticos o cuya solución analítica es muy engorrosa.

8.1. Solución por diferencias finitas

Este método consiste en simplemente discretizar las derivadas por medio de diferencias finitas (Sección 6.1) para convertir la ecuación diferencial en un sistema algebraico que puede ser resuelto por cualquier de los métodos de los capítulos 4 o 5.

La discretización de la doble derivada se hace usando la Ecuación (6.4):

$$\begin{aligned} \frac{dy}{dx} &\approx \frac{y_{i+1} - y_i}{\Delta x} \\ \frac{d^2y}{dx^2} &= \frac{d}{dx}\left(\frac{dy}{dx}\right) \approx \frac{1}{\Delta x}\left(\frac{y_{i+1} - y_i}{\Delta x} - \frac{y_i - y_{i-1}}{\Delta x}\right) \end{aligned}$$

Entonces la segunda derivada se discretiza como:

$$\boxed{\frac{d^2y}{dx^2} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2}} \quad (8.2)$$

Con este resultado y la Ec. (6.4), la ecuación diferencial (8.1) se discretiza en el punto i como:

$$\boxed{\frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} = f\left(x_i, y_i, \frac{y_{i+1} - y_i}{\Delta x}\right)} \quad (8.3)$$

La Ecuación (8.3) representa un sistema de ecuaciones cuyo tamaño depende del número de intervalos que queramos utilizar.

Ejemplo 8.1. Transferencia de calor bajo el suelo.

Considere que tenemos un terreno que absorbe efectivamente 800 W/m^2 de radiación solar en su superficie, la cual intercambia calor con el ambiente a 293 K y con un coeficiente de $10 \text{ W/m}^2\text{K}$ y con el cielo a 277 K y una emitancia de 0.8 . A una profundidad de 1.00 m la temperatura se conoce constante en 283 K y la conductividad térmica del suelo es 2.2 W/m.K . Calcular la distribución de temperaturas bajo el suelo si está en estado estacionario.

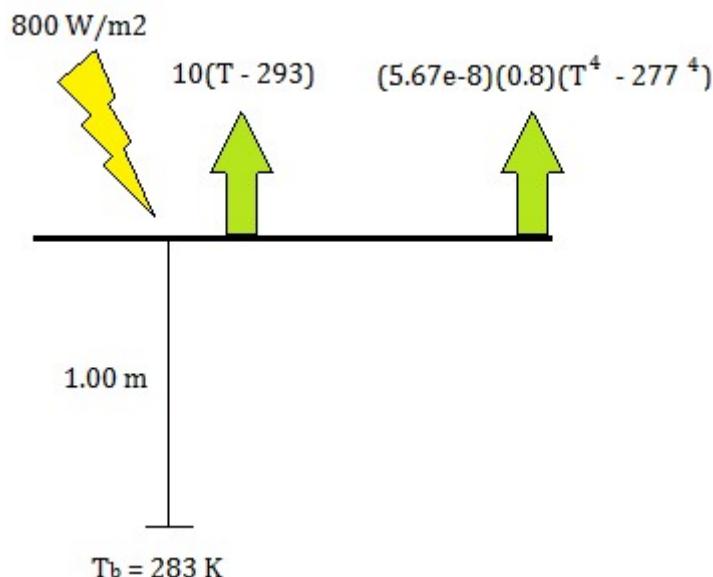


Figura 8.1. Diagrama de la transferencia de calor en el suelo.

La ecuación que modela la distribución de temperaturas bajo el suelo en este caso es la ecuación de calor 1D en estado estacionario:

$$\frac{d^2T}{dx^2} = 0 \quad (\text{E8.1})$$

Las condiciones de frontera son en el fondo:

$$T(1) = 283 \quad (\text{E8.2})$$

Y en la superficie es el balance de calor ($\sigma = 5.67 \times 10^{-8} \text{ W/m}^2\text{K}^4$):

$$2.2 \frac{dT(0)}{dx} + 800 + 10(293 - T(0)) + (0.8)\sigma(277^4 - T(0)^4) = 0 \quad (\text{E8.3})$$

Discretizamos primero el dominio de solución, en este caso vamos a calcular las temperaturas en 5 puntos (nodos) distanciados a 0.2 m cada uno ($\Delta x = 0.2 \text{ m}$). El diagrama se observa en la Figura 8.2:

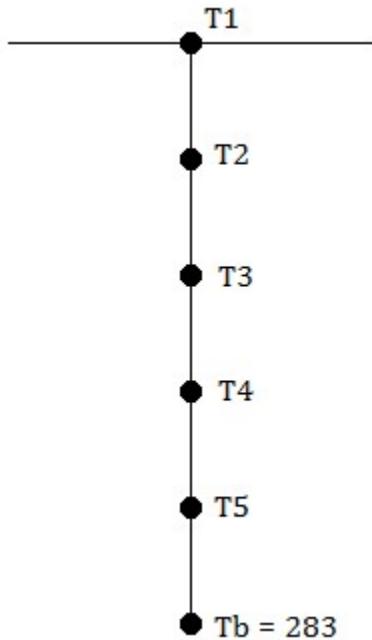


Figura 8.2. Discretización del dominio del problema.

Tenemos entonces para la superficie la discretización de (E8.3):

$$2.2 \frac{T_2 - T_1}{0.2} + 800 + 10(293 - T_1) + (0.8)\sigma(277^4 - T_1^4) = 0$$

Escribiéndolo para una solución por iteración por método de punto fijo modificado (Sección 5.2) tenemos:

$$\boxed{T_1^{k+1} = \frac{1}{21} [3997.05 + 11T_2^k - (0.8)(5.67 \times 10^{-8})(T_1^k)^4]} \quad (E8.4)$$

Para los nodos 2 a 4 discretizamos (E8.1):

$$\frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} = 0$$

Escribiéndolo para una solución por iteración por método de punto fijo modificado (Sección 5.2) tenemos:

$$\boxed{T_i^{k+1} = \frac{1}{2} (T_{i-1}^{k+1} + T_{i+1}^k)} \quad (E8.5)$$

Y para el nodo 5:

$$\frac{283 - 2T_5 + T_4}{\Delta x^2} = 0$$

Escribiéndolo para una solución por iteración por método de punto fijo modificado (Sección 5.2) tenemos:

$$T_5^{k+1} = \frac{1}{2}(T_4^{k+1} + 283) \quad (E8.6)$$

Implementando las ecuaciones (E8.4) a (E8.6) en una hoja de Excel con la suposición inicial de que todas son 300 K tenemos los resultados:

Resultados	
x [m]	T [K]
0.0	332.96
0.2	322.97
0.4	312.97
0.6	302.98
0.8	292.99
1.0	283.00
Iteraciones	20
Residual	2.44E-03

La gráfica muestra una distribución lineal de la temperatura bajo el suelo como se ve en la Figura 8.3:

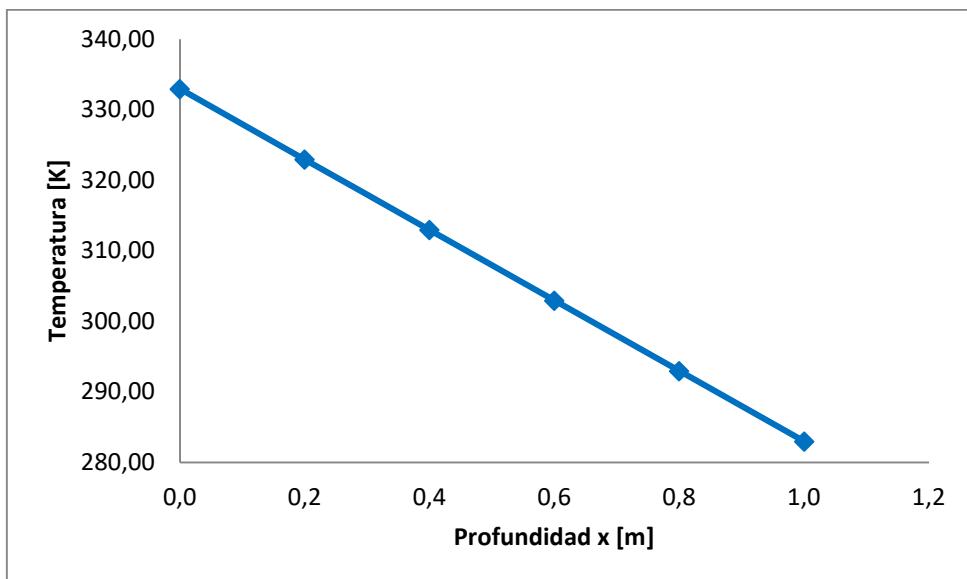


Figura 8.3. Distribución de la temperatura bajo el suelo.

Y la evolución de los residuales con cada iteración se observa en la Figura 8.4:

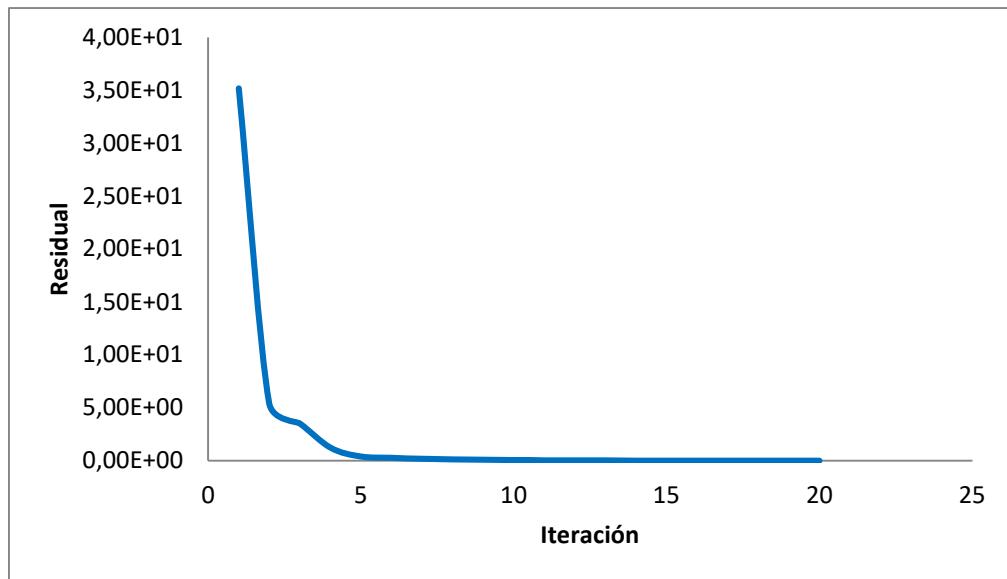


Figura 8.4. Evolución del residual con las iteraciones.

Bibliografía

- **Análisis numérico.** Richard L. Burden, J. Douglas Faires.
- **Calculus Vol. 1 y 2.** Tom Apostol.
- **Manual de Numpy:** <https://numpy.org/doc/2.1/reference/index.html>
- **Manual de Matplotlib:** <https://matplotlib.org/stable/api/index.html>

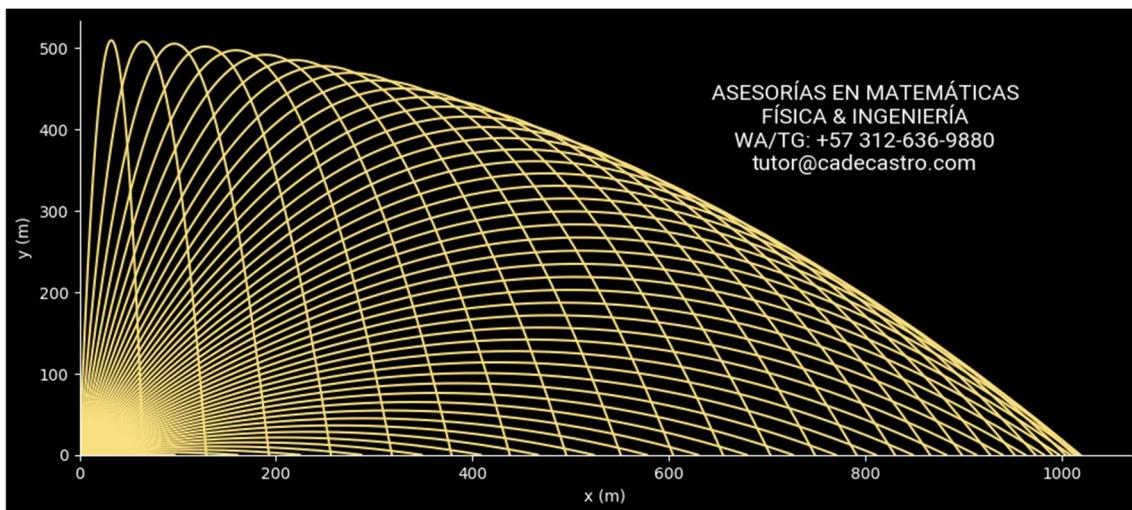
Sobre el autor



Carlos A. De Castro P.

Ingeniero mecánico egresado de la *Universidad de los Andes* de Bogotá, Colombia, donde estudió gracias a la beca *Bachilleres por Colombia* otorgada en 2004 por la Empresa Colombiana de Petróleos a los mejores bachilleres del país; es piloto deportivo de aviones ultralivianos graduado del *Club Colombiano de Aviación Deportiva*; ha presentado ponencias en el VII Congreso Colombiano de Métodos Numéricos, el Seminario Internacional Secado de Productos Agrícolas y el I Congreso de Energía Sostenible y tiene algunas publicaciones que usan métodos numéricos para analizar problemas reales en el ámbito de la investigación.

Asesorías en Matemáticas, Física e Ingeniería



- **Matemáticas:** álgebra, trigonometría, cálculo diferencial, integral y vectorial, ecuaciones diferenciales, álgebra lineal.
- **Física:** mecánica, eléctrica y térmica, dinámica de aviones.
- **Materias básicas de Ingeniería:** estática, sistemas dinámicos, mecánica de materiales sólidos, mecánica de fluidos.
- **Materias avanzadas de Ingeniería Mecánica:** termodinámica, transferencia de calor, mecanismos, diseño mecánico, aerodinámica básica.
- **Métodos numéricos:** uso de Python y Excel para solución por medio de métodos numéricos de diferentes problemas en materias avanzadas y proyectos de grado.
- Clases de refuerzo en uso de Excel para estudiantes y profesionales.