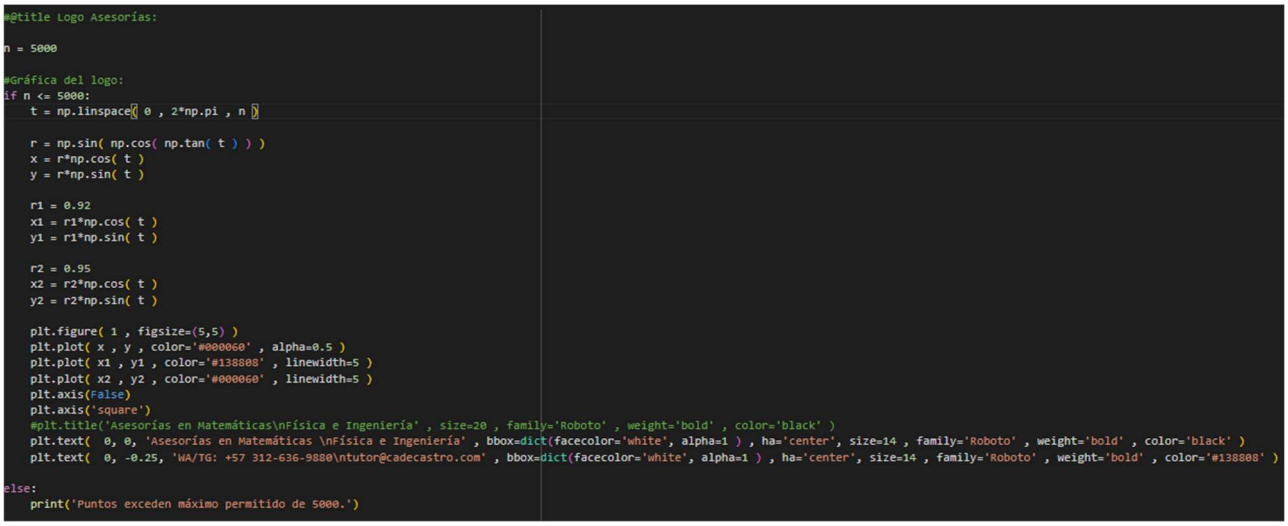
Métodos numéricos básicos para ingeniería con Python y Excel

1ª. Edición

Carlos A. De Castro P.

tutor.cadecastro.com

Código en Python con el que se genera el logo de la portada:



```python
#@title Logo Asesorías:

n = 5000

#Gráfica del logo:
if n <= 5000:
    t = np.linspace( 0 , 2*np.pi , n )

    r = np.sin( np.cos( np.tan( t ) ) )
    x = r*np.cos( t )
    y = r*np.sin( t )

    r1 = 0.92
    x1 = r1*np.cos( t )
    y1 = r1*np.sin( t )

    r2 = 0.95
    x2 = r2*np.cos( t )
    y2 = r2*np.sin( t )

    plt.figure( 1 , figsize=(5,5) )
    plt.plot( x , y , color='#000060' , alpha=0.5 )
    plt.plot( x1 , y1 , color='#138808' , linewidth=5 )
    plt.plot( x2 , y2 , color='#000060' , linewidth=5 )
    plt.axis(False)
    plt.axis('square')
    #plt.title('Asesorías en Matemáticas\nFísica e Ingeniería' , size=20 , family='Roboto' , weight='bold' , color='black' )
    plt.text(  0, 0, 'Asesorías en Matemáticas \nFísica e Ingeniería' , bbox=dict(facecolor='white', alpha=1 ) , ha='center', size=14 , family='Roboto' , weight='bold' , color='black' )
    plt.text(  0, -0.25, 'WA/TG: +57 312-636-9880\ntutor@cadecastro.com' , bbox=dict(facecolor='white', alpha=1 ) , ha='center', size=14 , family='Roboto' , weight='bold' , color='#138808' )

else:
    print('Puntos exceden máximo permitido de 5000.')
```


Latinoamérica

https://tutor.cadecastro.com

CONTENIDO

Introducción

¿Qué son y para qué sirven los métodos numéricos? Voy a tomar lo que dice Wikipedia[1]:

"*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 primera edición con códigos en Python, sin embargo, ya había habido otras versiones anteriores de este libro pero 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 un 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:

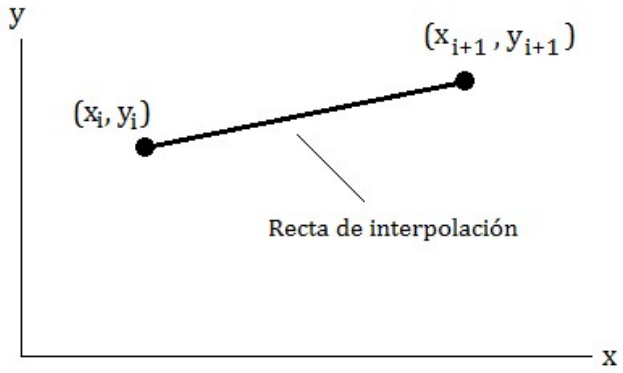


Figura 1.1. Interpolación lineal.

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

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

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
Entradas: valor a interpolar x, vectores conteniendo los puntos X y Y.
Salidas: valor interpolado y.

```
def IntLineal( x, X, Y ):
  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 es[2]:

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:

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

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

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

Tabla de datos	
x	y
72.7	35
78.7	45

Valor deseado x
75
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:

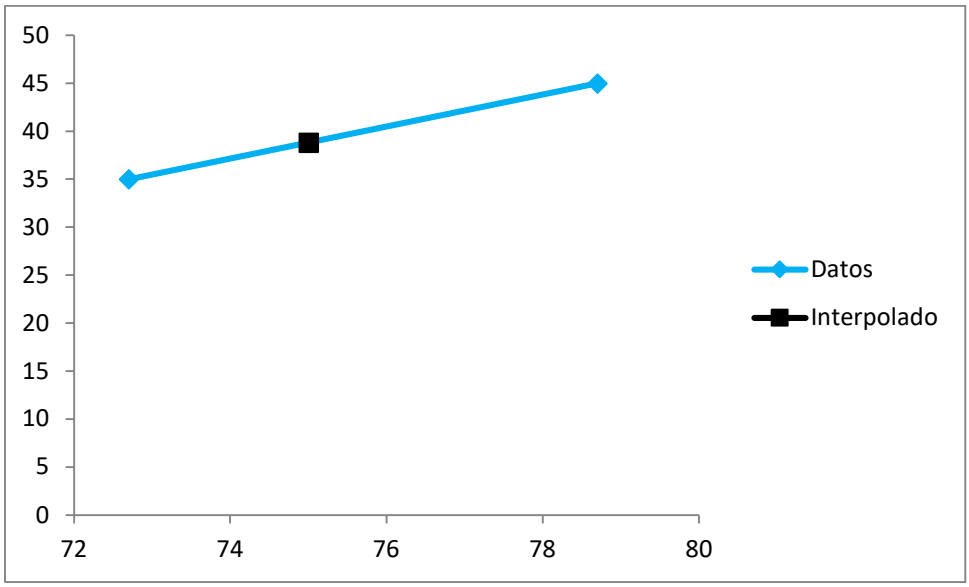


Figura 1.2. Interpolación de la tabla de saturación del agua.

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:

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

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
Entradas: valor a interpolar x, vectores conteniendo los puntos X y Y.
Salidas: valor interpolado y.

```
def PoliLagrange(x,X,Y):
  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. Polinomios de Lagrange (Python)

Se tiene el conjunto de datos:

x	y
1	1
2	3
3	-1
4	0
5	3
6	2

Corriendo el algoritmo 1.2 implementado dentro de un ciclo se muestra el polinomio interpolador de Lagrange junto a los datos originales:

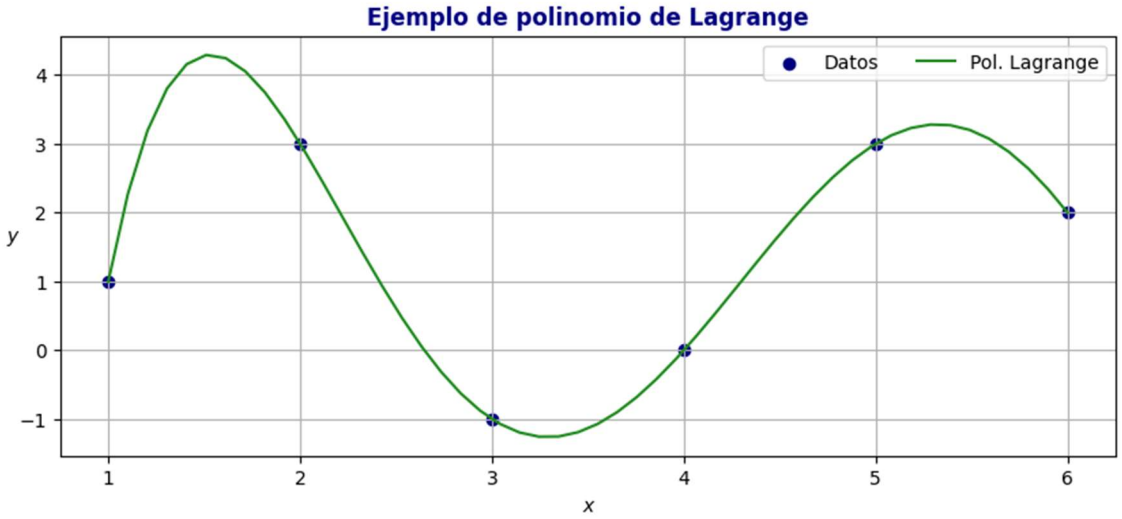


Figura 1.3. Polinomio de Lagrange interpolando los datos.

El código usado para generar esta gráfica usando la función definida para los polinomios de Lagrange es el siguiente:

```
X = [1,2,3,4,5,6]
Y = [1,3,-1,0,3,2]
xs = np.linspace( 1 , 6 , 50  )
ys = np.empty( shape=50 )

for i in range( len(xs) ):
  ys[i] = PoliLagrange( xs[i], X , Y )

plt.figure( 1, figsize=(10,4) )
plt.scatter( X , Y , color='navy', label='Datos' )
plt.plot( xs, ys, color='forestgreen', label='Pol. Lagrange' )
plt.title('Ejemplo de polinomio de Lagrange', color='navy',
weight='bold')
plt.ylabel('y', rotation=0 ,style='italic')
plt.xlabel('x', style='italic')
plt.legend( ncol=2 )
plt.grid()
```

2. Aproximación

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. Aproximación 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:

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

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

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

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/m2]	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 W/m²°C. La gráfica de la recta de aproximación con los puntos de datos se muestra a continuación:

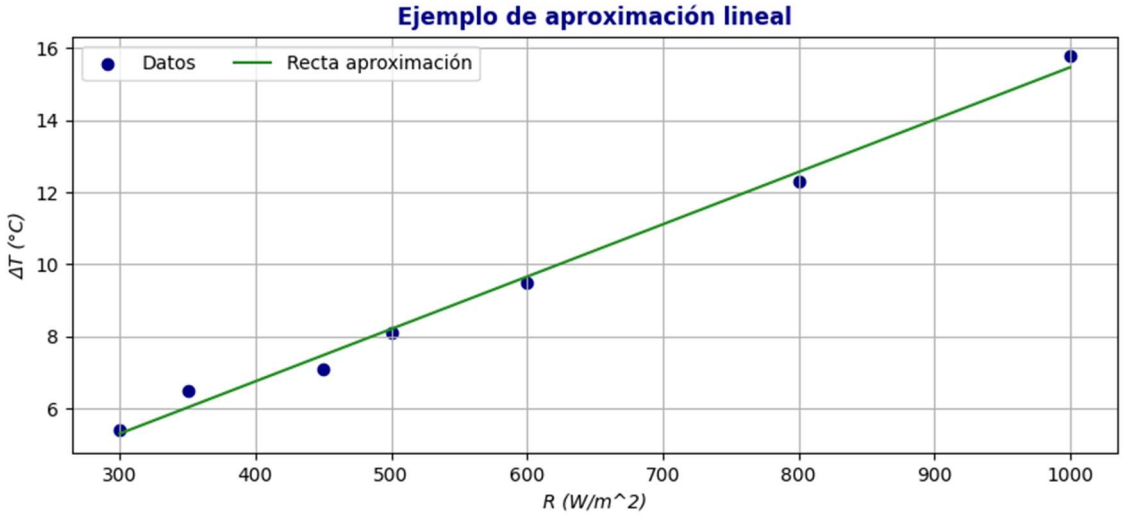


Figura 2.1. Aproximación lineal por mínimos cuadrados en Python.

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. Aproximación 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":

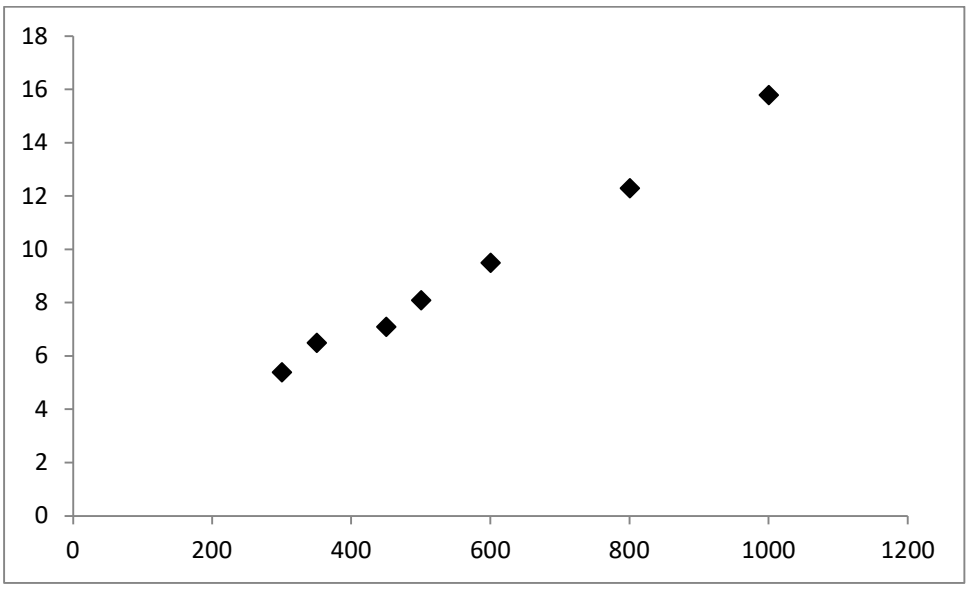
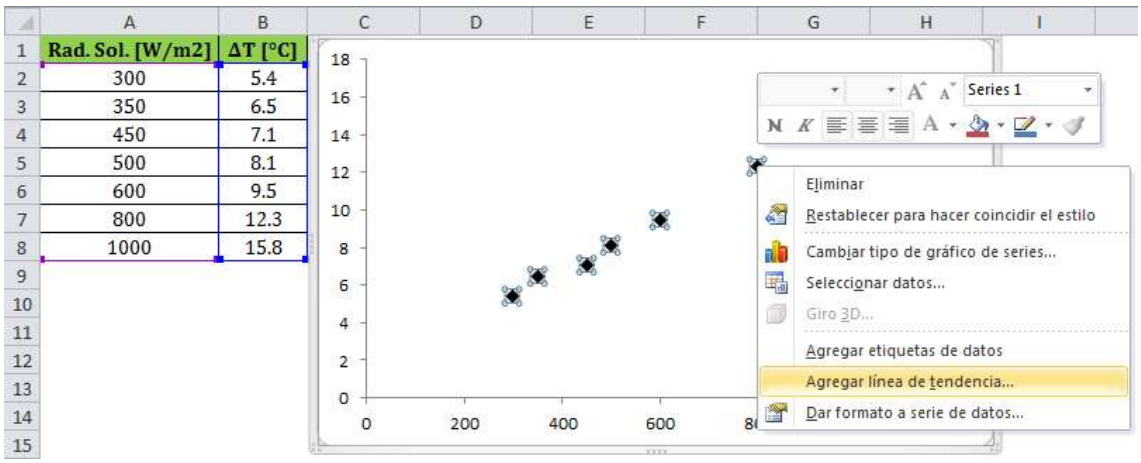
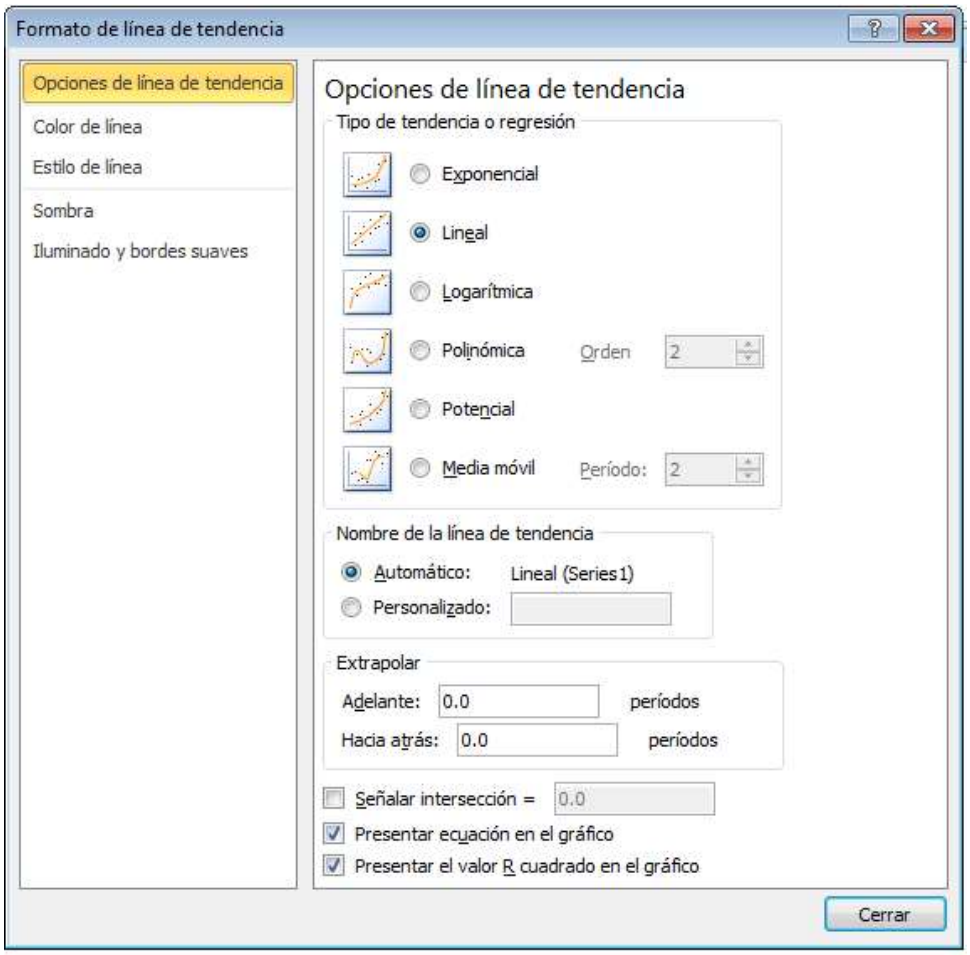


Figura 2.2. Gráfica de los datos del ejemplo.

Se hace click derecho sobre alguno de los puntos y se selecciona "Agregar 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:

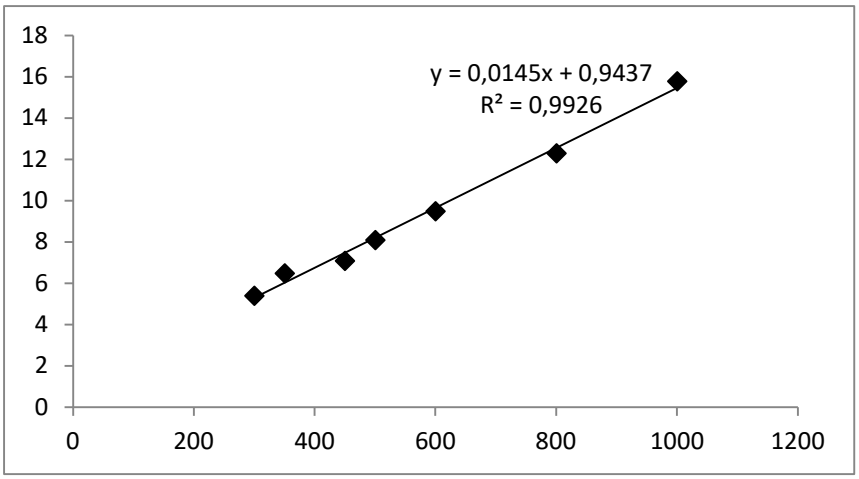


Figura 2.3. Aproximación lineal en Excel.

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

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

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 pseudocódigo debido a que no hay forma de escribir un código en Python general para este método.

Algoritmo 3.1: Método de punto fijo (pseudocódigo)
Entradas: aproximación inicial x0, tolerancia TOL
Salidas: valor x tal que f(x)=x

```
sw=1;
x1=x0;
while sw==1
    x2=f(x1);
    if abs(x2-x1)<=TOL
        x=x2;
        sw=0;
    end
     x1=x2;
end
```

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 = 2.5in, σ = 24.89ksi, K_{Ic} = 52ksi√in. Se elige como aproximación inicial a_0 = 0.250in. 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 $a_f = 0.620$in.

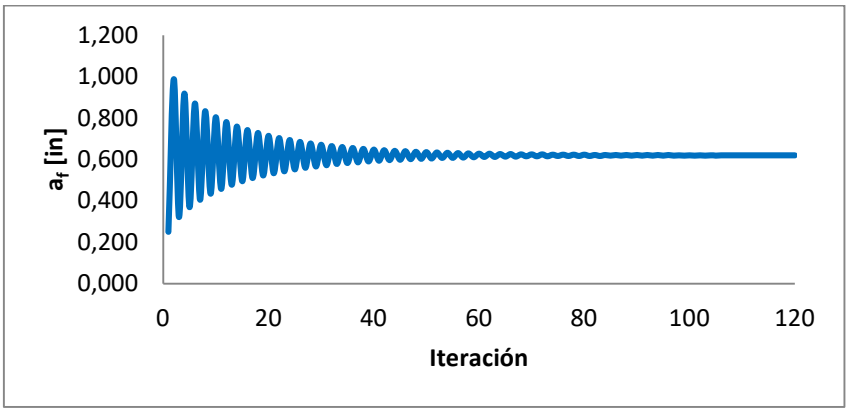


Figura 3.1. Valor calculado de a_f en cada iteración.

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

$$\boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}} \qquad (3.2)$$

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 pseudocódigo debido a que no hay forma de escribir un código en Python general para este método.

Algoritmo 3.2: Método de Newton-Raphson (pseudocódigo)
Entradas: aproximación inicial x0, tolerancia TOL
Salidas: valor x tal que f(x)=0

```
sw=1;
x1=x0;
while sw==1
    x2=x1-f(x1)/f'(x1);
    if abs(x2-x1)<=TOL
        x=x2;
        sw=0;
    end
    x1=x2;
end
```

Ejemplo 3.2. Raíz de polinomio (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

$$x_{k+1} = x_k - \frac{3x_k^3 - 2x_k + 8}{9x_k^2 - 2}$$

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

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

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

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 pseudocódigo debido a que no hay forma de escribir un código en Python general para este método.

Algoritmo 3.3: Método de la secante (pseudocódigo)
Entradas: aproximaciones iniciales x0, x1, tolerancia TOL
Salidas: valor x tal que f(x)=0

```
sw=1;
while sw==1
    x2=x1-(x1-x0)*f(x1)/(f(x1)-f(x0));
    if abs(x2-x1)<=TOL
        x=x2;
        sw=0;
    end
end
```

Ejemplo 3.3. 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

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

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

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:

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

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:

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

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 (Python y 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:

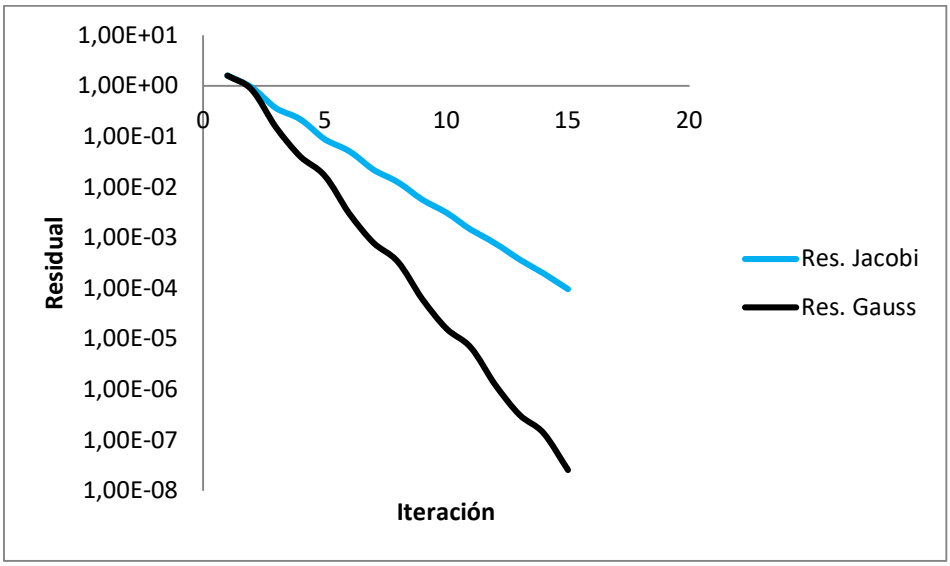


Figura 4.1. Residual en cada iteración.

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*.

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:

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

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.

Algoritmo 5.1: Método de punto fijo multivariable (pseudocódigo)
Entradas: vector aproximaciones iniciales x0, tolerancia TOL, iteraciones máximas MAX.
Salidas: vector solución x, residual res.

```
sw=1;
x1=x0;
while sw==1
    x2=f(x1);
    if norm(x2-x1)<=TOL
        x=x2;
        sw=0;
    end
    x1=x2;
end
```

Ejemplo 5.1. Transferencia de calor (Excel).

Consideremos dos placas paralelas entre sí, la placa superior recibe radiación solar equivalente de 1000 W/m² con absortividad de 0.9 e intercambia calor con el ambiente a 293K con coeficiente de convección de 10 W/m²K y con el cielo a 277K; la placa inferior intercambia calor por convección a un medio a 300K con coeficiente de convección de 15 W/m²K; las emitancias de las placas son de 0.9. Las placas intercambian calor entre ellas por convección a 5 W/m²K y por radiación.

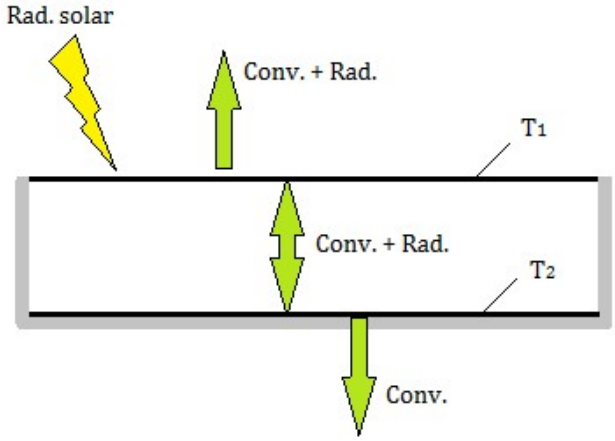


Figura 5.1. Diagrama del sistema.

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:

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

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

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

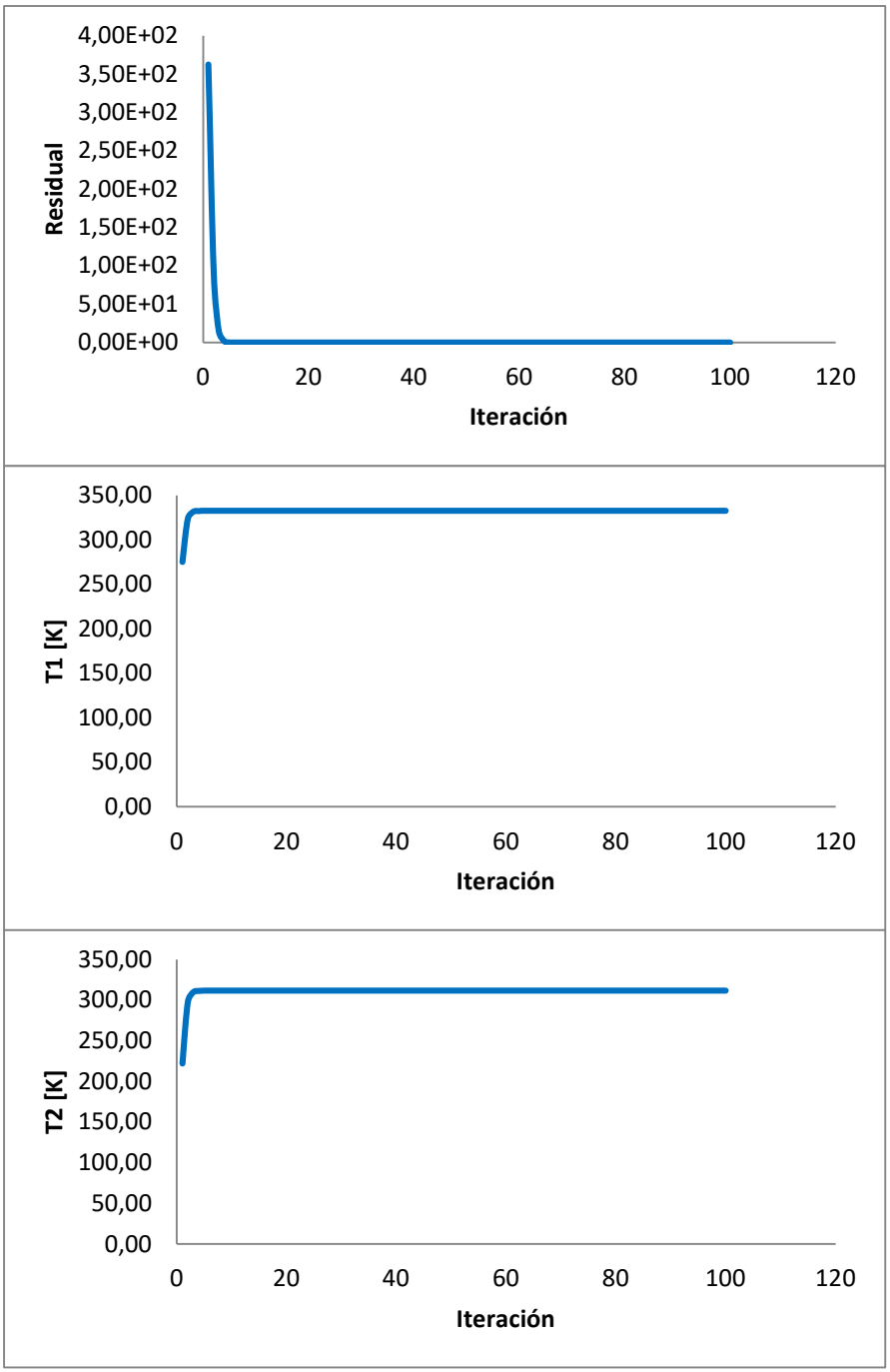


Figura 5.2. Evolución de las iteraciones.

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:

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:

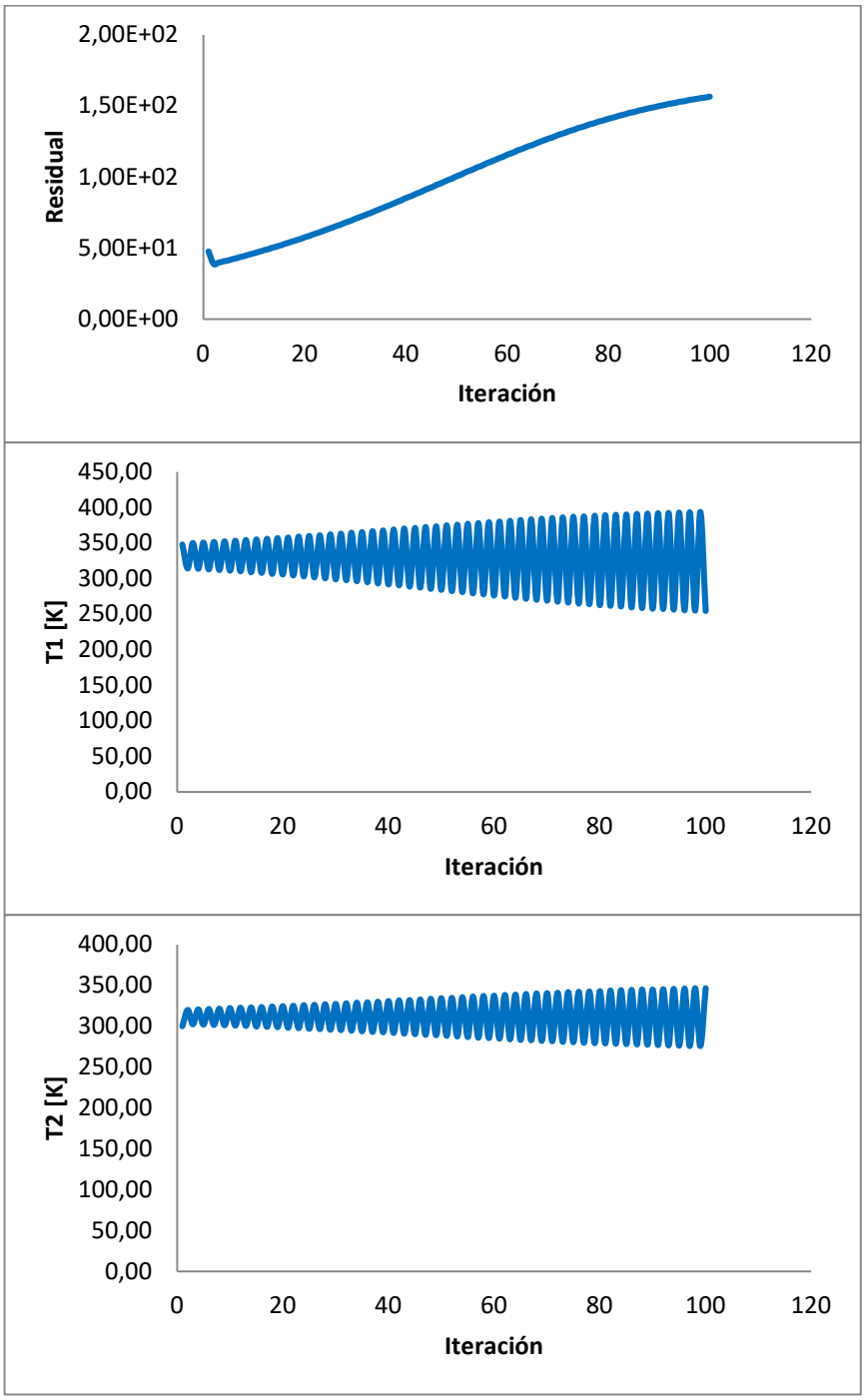


Figura 5.3. Sistema inestable debido a suposición inicial.

¿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:

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

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:

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

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

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.

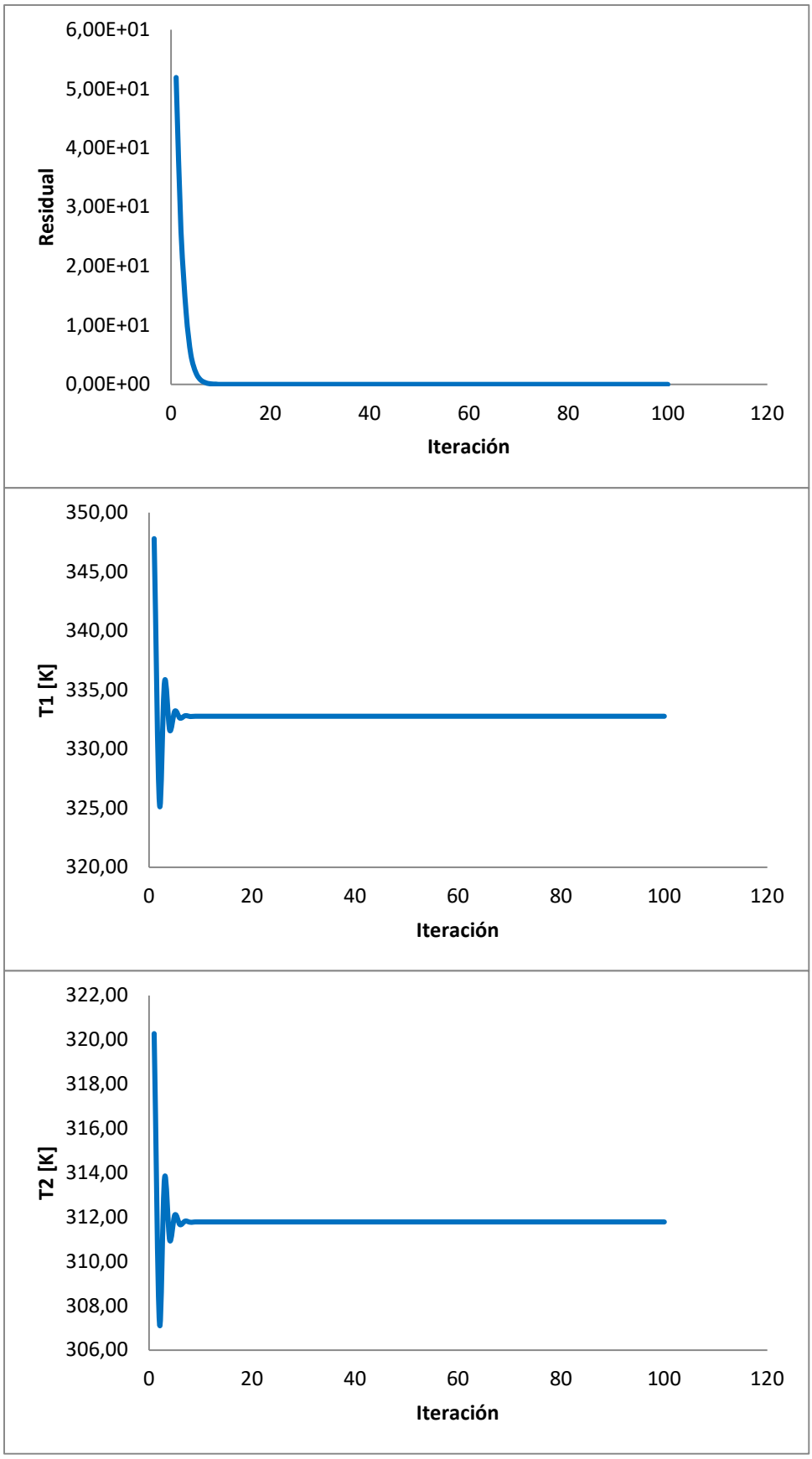


Figura 5.4. Evolución de las iteraciones.

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. Considérense tres puntos intermedios en una curva como se muestra en la figura 6.1:

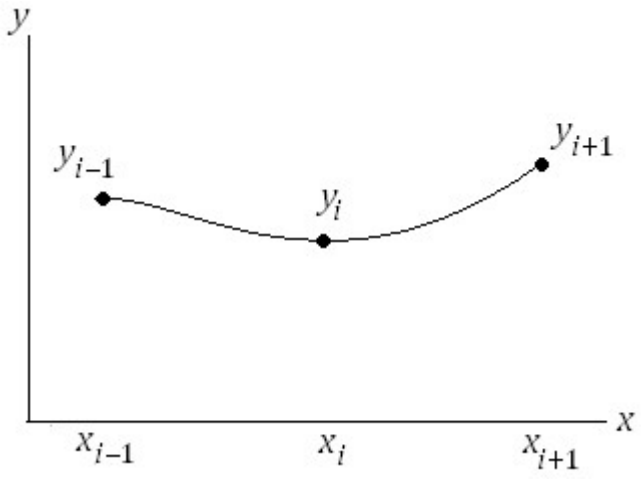


Figura 6.1. Curva discretizada.

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

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

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

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

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

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

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

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

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
Entradas: vectores de Numpy conteniendo los puntos X y Y.
Salidas: vector con el valor de las derivadas, df.

```
def derivada(X,Y):
  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. Curvas SVAJ para levas (Excel)

Se tiene una tabla datos de la curva de movimiento de un seguidor de leva (ver archivo de Excel que viene con el libro), determinar si la leva cumple con

condiciones de continuidad de las curvas de velocidad, aceleración y sobre-aceleración (también conocida como *jerk*).

Teniendo tabulados los datos de tiempo y posición podemos calcular con las ecuaciones (6.4) a (6.6) la velocidad, aceleración y sobreaceleración de forma numérica:

$$v = \frac{dx}{dt}\,, \qquad a = \frac{dv}{dt}\,, \quad j = \frac{da}{dt}$$

En las gráficas se observan saltos en la velocidad y aceleración, por lo cual se deduce que este diseño de leva no es adecuado.

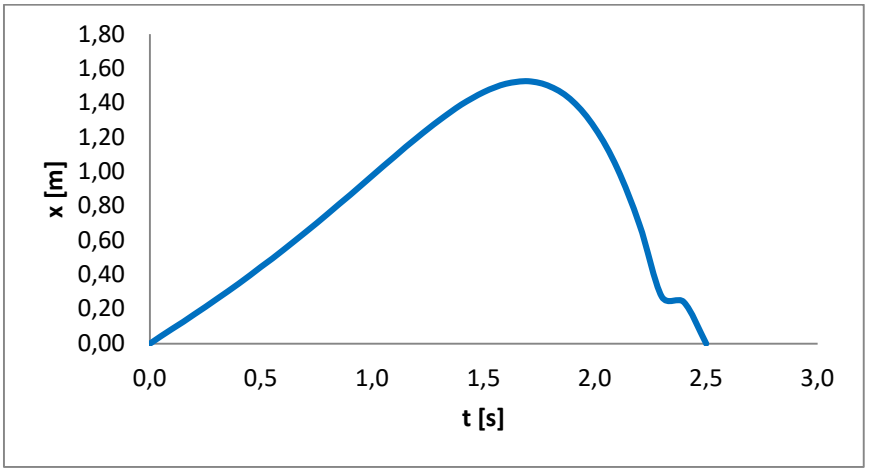


Figura 6.2. Posición del seguidor.

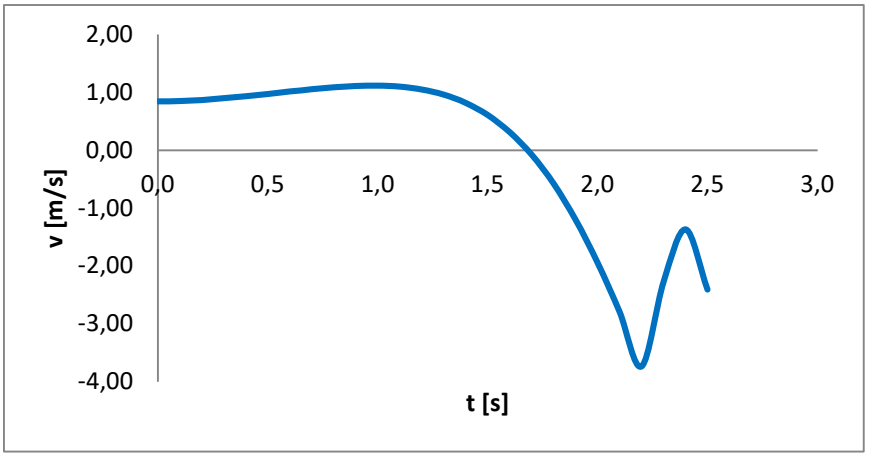


Figura 6.3. Velocidad del seguidor.

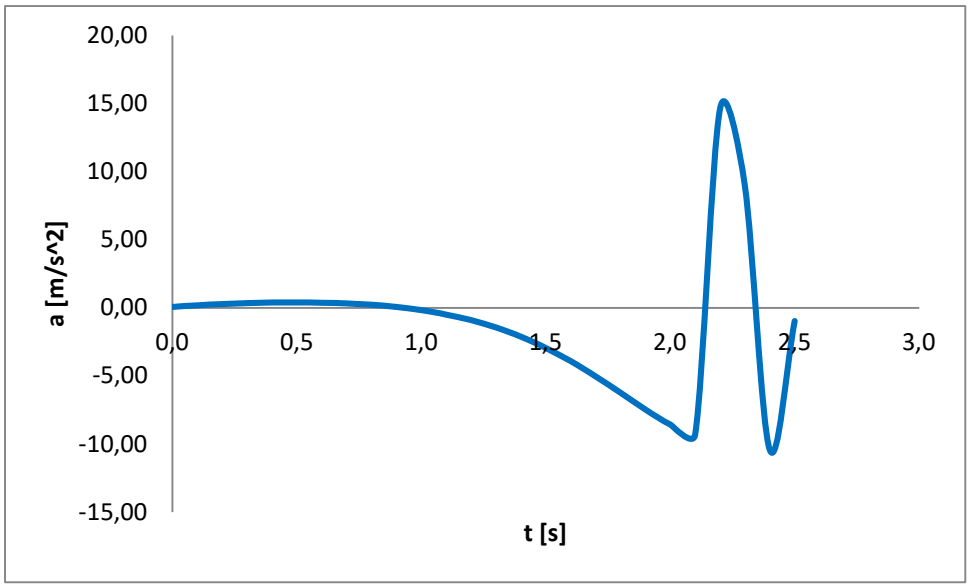


Figura 6.4. Aceleración del seguidor.

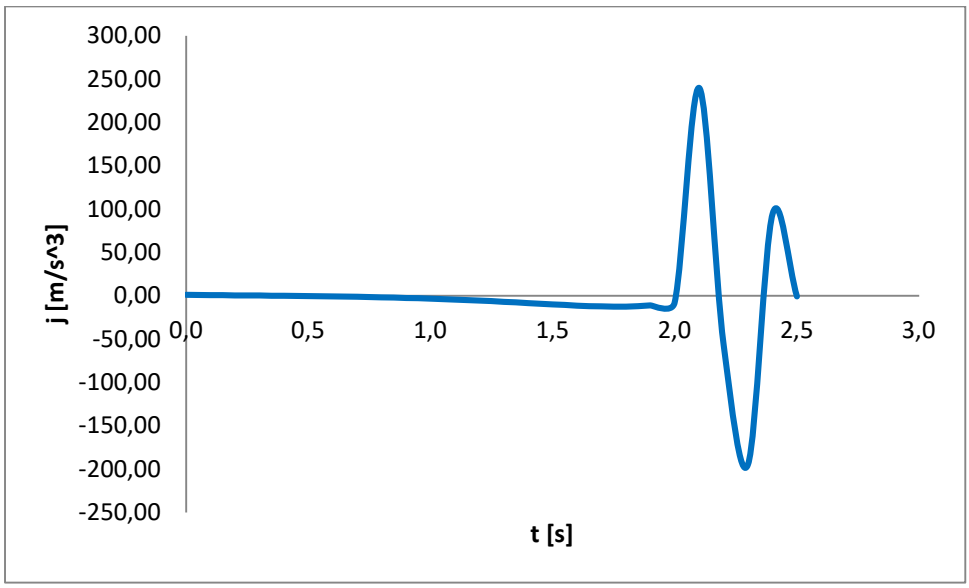


Figura 6.5. Sobreaceleración del seguidor.

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 en 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.

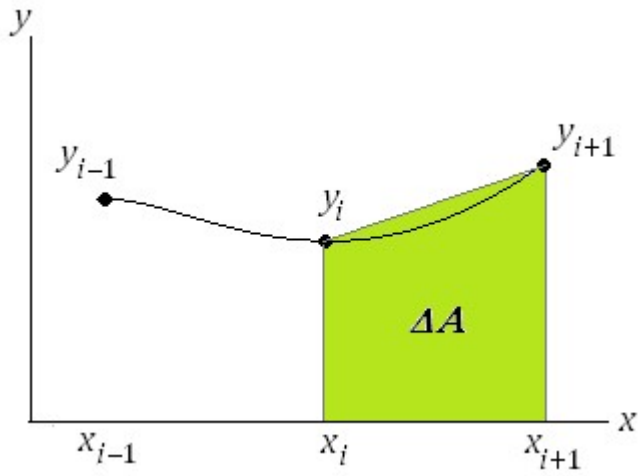


Figura 6.6. Área de un trapecio entre dos puntos de una curva.

El área del trapecio es:

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

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:

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

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

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

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

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

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
Entradas: valor inicial de la integral I0, vectores conteniendo los puntos X y Y.
Salidas: vector con el valor acumulativo de la integral, I.

```
def integral_trapecios(I0, X, Y ):
  N=len(X)
  I = np.empty( shape=N )
  I[0]=I0
  for n in range(1,N):
      I[n] = I[n-1]+0.5*(Y[n]+Y[n-1])*(X[n]-X[n-1]);

  return I
```

Ejemplo 6.2. 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

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

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:

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

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:

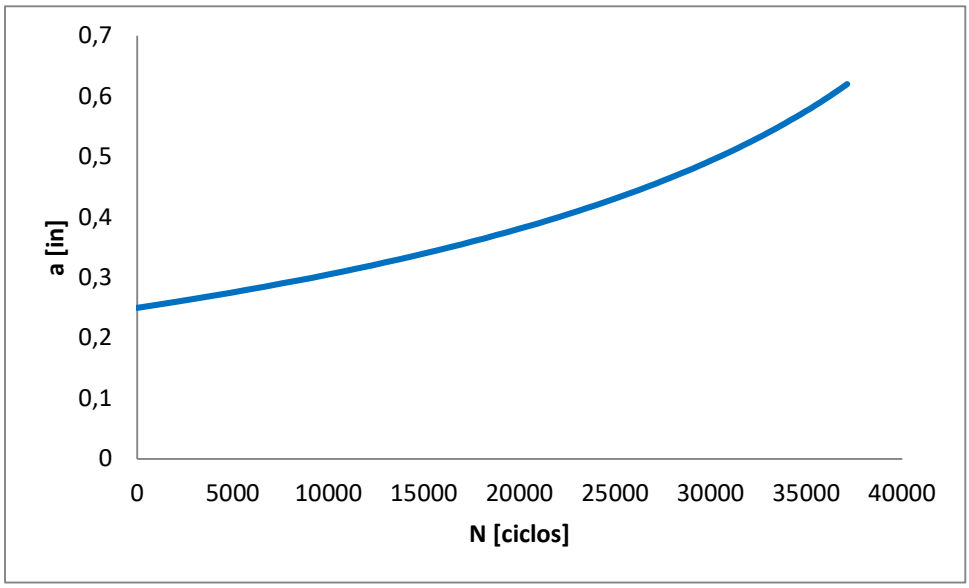


Figura 6.7. Crecimiento de grieta calculado por integración numérica.

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

$$N_f = 37120$$

7. Ecuaciones diferenciales con valor inicial

Los métodos numéricos para ecuaciones diferenciales que se presentan aplican para ecuaciones diferenciales ordinarias con condiciones iniciales de tipo

$$\frac{dy}{dt} = f(t, y)$$
$$y(t_0) = y_0 \qquad (7.1)$$

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

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

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

$$y_{n+1} = y_n + \Delta t \cdot f(t_n, y_n) \qquad (7.2)$$

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

En el algoritmo 7.1 se muestra el algoritmo del método de Euler en pseudocódigo debido a que no hay forma de escribir un código en Python general para este método.

Algoritmo 7.1: Método de Euler (pseudocódigo)
Entradas: valor inicial y0, tiempo inicial t0, tamaño de paso dt, número de puntos N
Salidas: valores y tal que dy/dt = f(t,y)

```
y(1)=y0;
t(1)=t0;
for n=2:N
    y(n)=y(n-1)+dt*f(t(n-1),y(n-1));
    t(n)=t(n-1)+dt;
end
```

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.

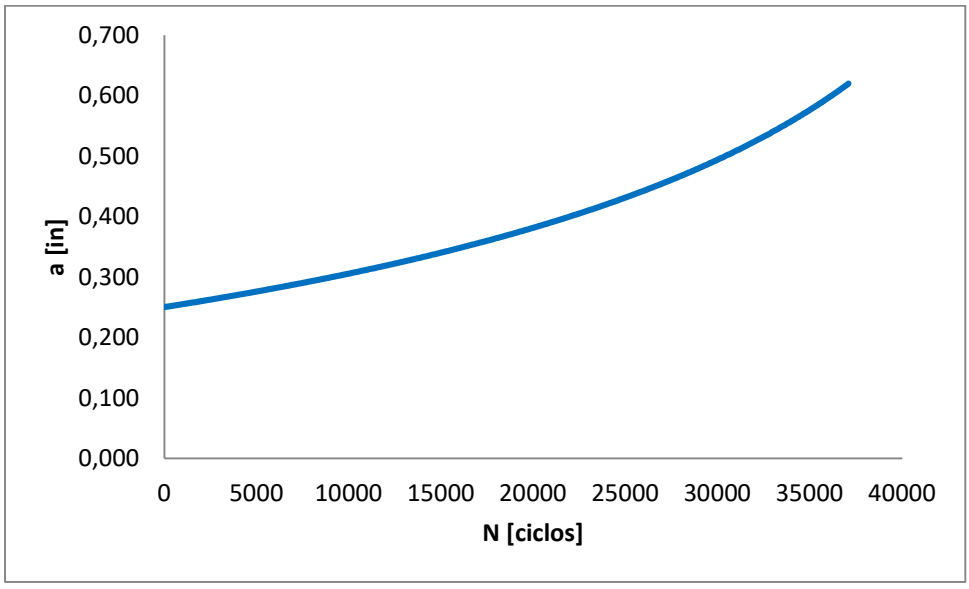


Figura 7.1. Crecimiento de grieta calculado con el método de Euler.

7.2. Método de Runge-Kutta de 4° 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 Δt y un número máximo de iteraciones N tal que

$$k_1 = \Delta t \cdot f(t_k, y_k)$$

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

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

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

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

Para k = 0,..., N-1. La solución se da a lo largo del intervalo (t_o, t_o+ Δt N). En el algoritmo 7.2 se muestra el algoritmo del método de Runge-Kutta de orden 4 en pseudocódigo, debido a que no hay forma de escribir un código en PYTHON general para este método.

Algoritmo 7.2: Método de Runge-Kutta de orden 4 (pseudocódigo)
Entradas: valor inicial y0, tiempo inicial t0, tamaño de paso dt, número de puntos N
Salidas: valores y tal que dy/dt = f(t,y)

```
y(1)=y0;
t(1)=t0;
for n=2:N
    k1=dt*f(t(n-1),y(n-1));
    k2=dt*f(t(n-1)+dt/2,y(n-1)+k1/2);
    k3=dt*f(t(n-1)+dt/2,y(n-1)+k2/2);
    k4=dt*f(t(n-1)+dt,y(n-1)+k3);
    y(n)=y(n-1)+1/6*(k1+2*k2+2*k3+k4);
    t(n)=t(n-1)+dt;
end
```

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

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

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

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

Supóngase una moneda con m = 0.010kg y A = 3.1416×10^{-4} m^2, que cae de un edificio, entonces ρ = 1kg/m^3. La velocidad terminal según E7.3 es $v_{f,teórica} = 24.98 m/s$. 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 Δt=1s 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:

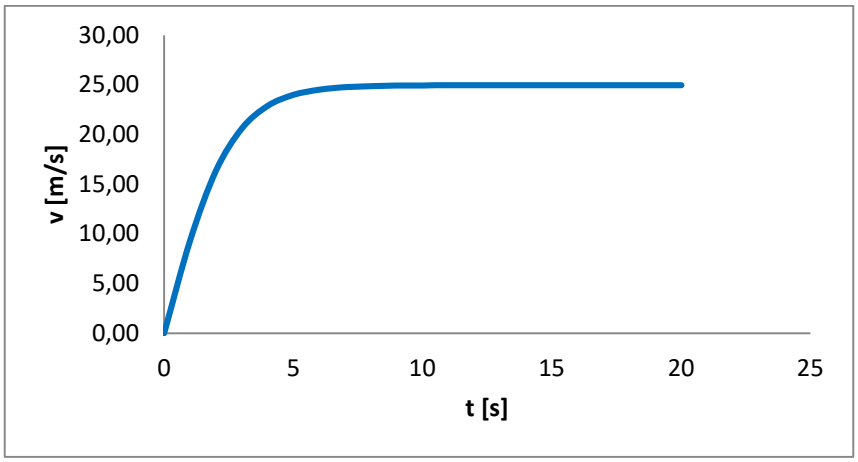


Figura 7.2. Velocidad de la moneda a medida que cae.

8. Ecuaciones diferenciales con valores en la frontera

El método numérico para ecuaciones diferenciales que se presenta aplica para ecuaciones diferenciales de tipo:

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

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):

$$\frac{dy}{dx} \approx \frac{y_{i+1} - y_i}{\Delta x}$$

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

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 W/m^{2}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.

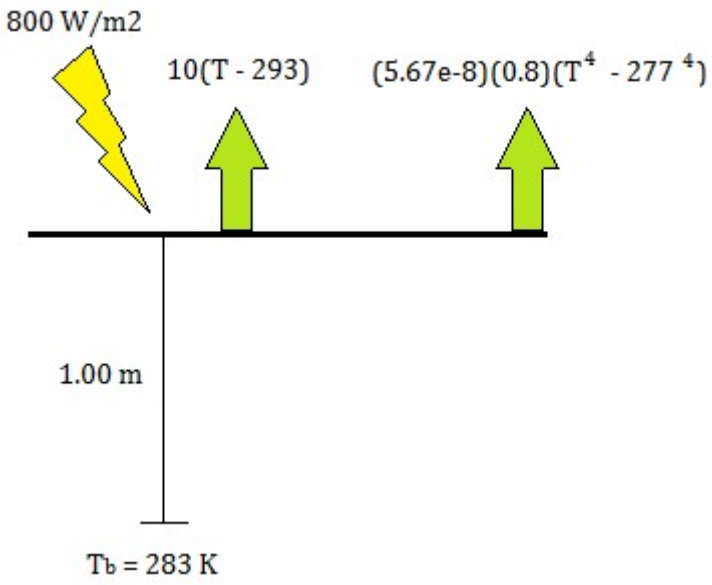


Figura 8.1. Diagrama de la transferencia de calor en el suelo.

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

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

Las condiciones de frontera son en el fondo:

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

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

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

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 (Δx = 0.2 m). El diagrama se observa en la Figura 8.2:

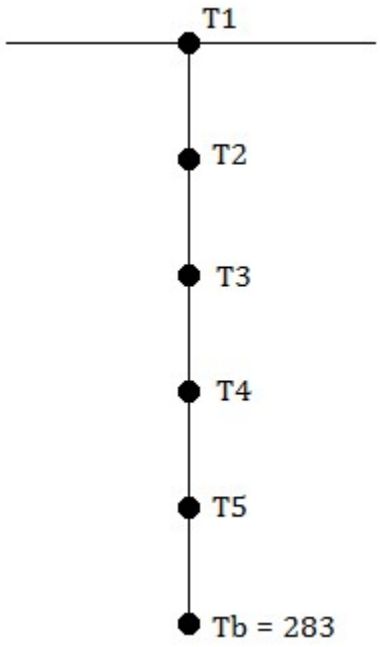


Figura 8.2. Discretización del dominio del problema.

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:

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

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:

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

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:

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

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:

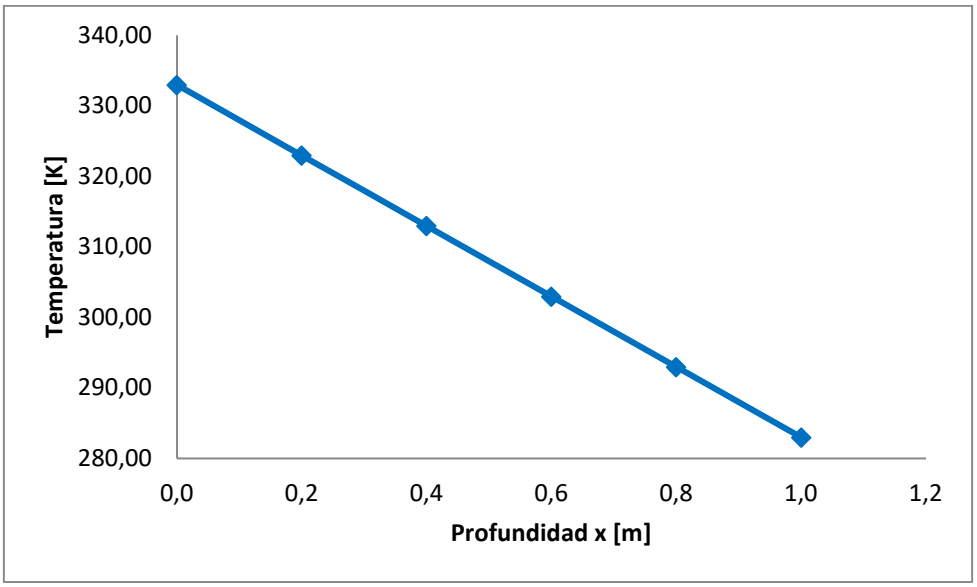


Figura 8.3. Distribución de la temperatura bajo el suelo.

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

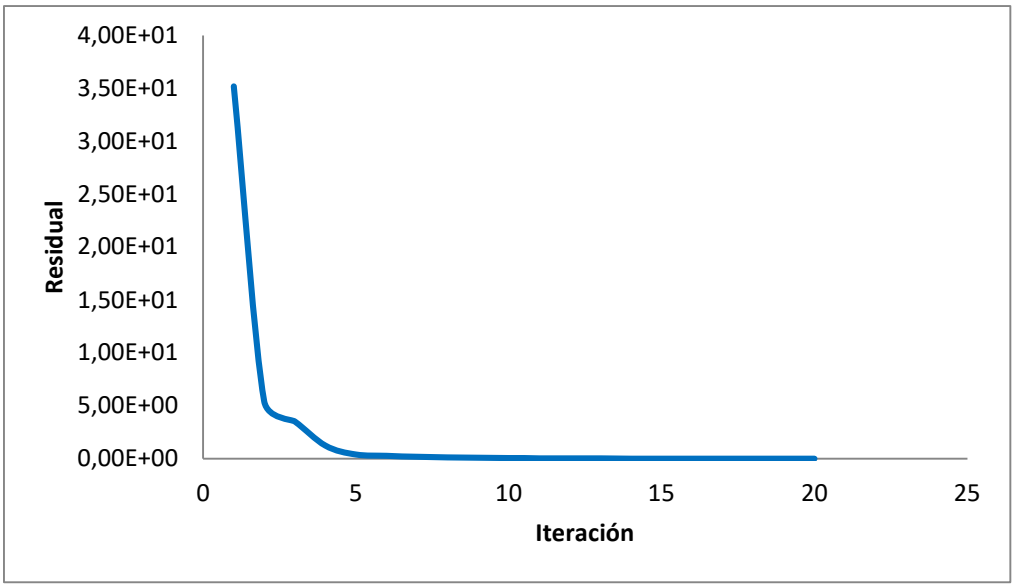
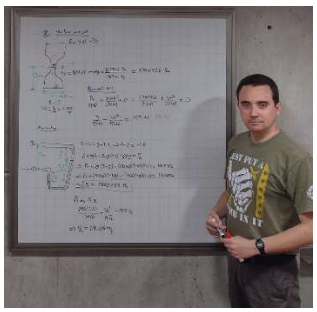


Figura 8.4. Evolución del residual con las iteraciones.

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

Sobre el autor



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.