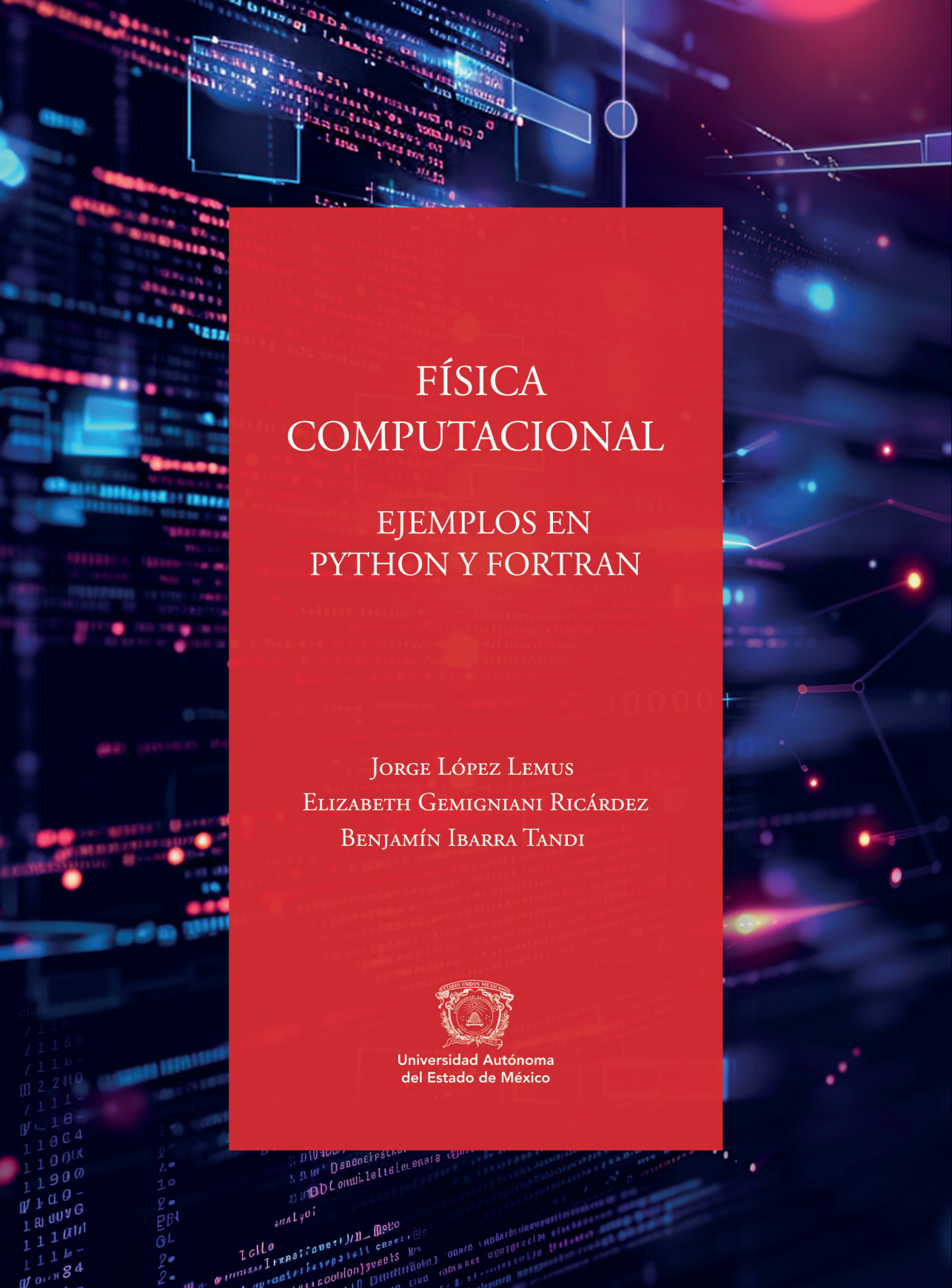


FÍSICA COMPUTACIONAL

EJEMPLOS EN PYTHON Y FORTRAN

Jorge López Lemus
Elizabeth Gemigniani Ricárdez
Benjamín Ibarra Tandi

Universidad Autónoma
del Estado de México

Doctor en Ciencias e Ingeniería Ambientales
Carlos Eduardo Barrera Díaz
Rector

Doctor en Ciencias Computacionales
José Raymundo Marcial Romero
Secretario de Docencia

Doctora en Ciencias Sociales
Martha Patricia Zarza Delgado
Secretaria de Investigación y Estudios Avanzados

Doctor en Ciencias de la Educación
Marco Aurelio Cienfuegos Terrón
Secretario de Rectoría

Doctora en Humanidades
María de las Mercedes Portilla Luja
Secretaria de Difusión Cultural

Doctor en Ciencias del Agua
Francisco Zepeda Mondragón
Secretario de Extensión y Vinculación

Doctor en Educación
Octavio Crisóforo Bernal Ramos
Secretario de Finanzas

Doctora en Ciencias Económico Administrativas
Eréndira Fierro Moreno
Secretaria de Administración

Doctora en Ciencias Administrativas
María Esther Aurora Contreras Lara Vega
Secretaria de Planeación y Desarrollo Institucional

Doctora en Derecho
Luz María Consuelo Jaimes Legorreta
Abogada General

Maestra en Salud Animal
Trinidad Beltrán León
Secretaria Técnica de la Rectoría

Licenciada en Comunicación
Ginarely Valencia Alcántara
Directora General de Comunicación Universitaria

Doctor en Ciencias Sociales
Luis Raúl Ortiz Ramírez
Director de Centros Universitarios y
Unidades Académicas Profesionales Región A
y Encargado del Despacho Región B

Física computacional: ejemplos en Python y Fortran

Dirección de Publicaciones Universitarias
Editorial de la Universidad Autónoma del Estado de México

Doctor en Ciencias e Ingeniería Ambientales
Carlos Eduardo Barrera Díaz
Rector

Doctora en Humanidades
María de las Mercedes Portilla Luja
Secretaria de Difusión Cultural

Doctor en Administración
Jorge Eduardo Robles Alvarez
Director de Publicaciones Universitarias

Física computacional: ejemplos en Python y Fortran

Jorge López Lemus
Elizabeth Gemigniani Ricárdez
Benjamín Ibarra Tandi

Universidad Autónoma del Estado de México

"2024, Conmemoración del 60 Aniversario de la Inauguración de Ciudad Universitaria"

López Lemus, Jorge.

Física computacional : ejemplos en Python y Fortran / Jorge López Lemus, Elizabeth Gemigniani Ricárdez, Benjamín Ibarra Tandi.

1ª ed.

Toluca, Estado de México : Universidad Autónoma del Estado de México, 2024

250 p. : il. ; 23 cm.

Incluye referencias bibliográficas (p. 245-248).

ISBN: 978-607- 633-923-7

1. Física -- Procesamiento de datos.

2. Python (Lenguaje de programación para computadora).

3. Fortran (Lenguaje de programación para computadora).

QC20.7.E4 L67 2024

Este libro fue positivamente dictaminado con el aval de dos revisores externos, conforme al Reglamento de la Función Editorial de la UAEMEX, y fue sometido a un proceso del identificación de duplicidad de la información mediante un *software* especializado.

Primera edición, octubre 2024

Física computacional: ejemplos en Python y Fortran
Jorge López Lemus
Elizabeth Gemigniani Ricárdez
Benjamín Ibarra Tandi

Universidad Autónoma del Estado de México
Av. Instituto Literario 100 Ote., Col. Centro
Toluca, Estado de México
C.P. 50000
Tel: (52) 722 481 1800
http://www.uaemex.mx

Registro Nacional de Instituciones y Empresas Científicas y Tecnológicas (Reniecyt: 1800233)

ISBN: 978-607-633-923-7

Hecho en México

Director del equipo editorial: Jorge Eduardo Robles Alvarez
Coordinación editorial: Ixchel Edith Díaz Porras
Corrección de estilo: Eduardo Acosta y Ma. Socorro Zepeda M.
Formación: Elizabeth Vargas Albarrán
Diseño de portada: Luis Alberto Maldonado Barraza

CONTENIDO

PRESENTACIÓN 15
INTRODUCCIÓN 17

BIBLIOTECAS DE PYTHON 21

1. ESTADÍSTICA DE DATOS 23
   Promedio y desviación estándar 23
   Método de mínimos cuadrados 28

2. CINEMÁTICA 37
   Caída libre 37
   Tiro parabólico 39
   Movimiento circular uniformemente acelerado 43

3. DERIVACIÓN NUMÉRICA 47
   Derivada en términos del límite 47
   Derivada n-ésima 51
   Desarrollo en serie de Taylor 54

4. INTEGRACIÓN NUMÉRICA 59
   Método del trapecio 59
   Método de Simpson 62
   Método de Simpson compuesto 63

5. MUESTREO ALEATORIO SIMPLE 67
   Cálculo de π 69
   Integral numérica de una función f(x) 73

Integral para el cálculo de π 80

6. RAÍCES DE UN POLINOMIO 83
Factorización 84
La fórmula de Bhaskara 87
Método de Newton-Raphson 91
Método de la secante 95

7. ECUACIONES DIFERENCIALES ORDINARIAS 101
Método de Taylor 101
Método de Euler 102
Método de Taylor de segundo orden 108
Método de Runge-Kutta 111
Método de Runge-Kutta de tercer orden 112
Método de Runge-Kutta de cuarto orden 114
Ecuaciones diferenciales de segundo orden homogéneas 119
Ecuaciones diferenciales de segundo orden no-homogéneas 131

8. MATRICES 137
Suma y producto de matrices 137
Determinante de una matriz 138
Matriz adjunta 141
Matriz traspuesta 145
Matriz inversa 146
Sistema de ecuaciones algebraicas 149
Método de Gauss-Jordan 154
Eigenvectores y eigenvalores 160

9. DINÁMICA MOLECULAR 167
Algoritmo de Verlet 171
Algoritmo leap-frog 172
Algoritmo de velocity Verlet 173

Condiciones de frontera periódicas y condición de mínima imagen 173
Correcciones de largo alcance 176
Método de sumas de Ewald 178
Propiedades termodinámicas y de estructura 180
Función de distribución radial 181
Lammps 183
Perfil de densidad y tensión superficial 187

10. MONTE CARLO 195
Ensamble canónico 199
Ensamble isotérmico-isobárico 201
Presión de un fluido de esferas duras 202

APÉNDICE A 209

APÉNDICE B 219

APÉNDICE C 227

REFERENCIAS 245

A las personas más valiosas en nuestras vidas:
Ana Paola, Andrea y Jorge Omar.

AGRADECIMIENTOS

A la Facultad de Ciencias por el apoyo recibido para elaborar esta obra.

PRESENTACIÓN

Esta obra es una recopilación de códigos desarrollados y escritos en Python que permiten hacer cálculos numéricos para resolver problemas en Física y que han servido de apoyo a la Unidad de Aprendizaje Física Computacional de la Licenciatura en Física que se oferta en la Facultad de Ciencias de la Universidad Autónoma del Estado de México.

Física computacional: ejemplos en Python y Fortran surge por la necesidad de contar con material de apoyo didáctico ante la contingencia sanitaria causada por el virus SARS-CoV-2, causante de Covid-19. A raíz de esto, se requirió la traducción de los códigos en lenguaje Fortran y C a un lenguaje de software libre, que se pueda utilizar en cualquier dispositivo que tenga conexión a internet, como lo es el lenguaje de Python.

La obra es muestra de que, en la Facultad de Ciencias, a 36 años de su creación, el personal docente está comprometido y enfrenta con responsabilidad los grandes desafíos en docencia, con la finalidad de fortalecer la formación integral de sus estudiantes a través de materiales con ejercicios que fomenten la generación de aprendizajes significativos y relevantes.

Aprovecho estas breves líneas para agradecer a los autores por su contribución. Sin duda, su esfuerzo coadyuvará a que nuestros estudiantes cuenten con elementos para garantizar una formación académica sólida e integral acorde a las necesidades actuales.

Doctor en Ciencias
Enrique Castañeda Alvarado

INTRODUCCIÓN

La educación científica de los jóvenes
es al menos tan importante, quizá incluso más,
que la propia investigación.

Glenn Theodore Seaborg

Esta obra es el resultado de varias propuestas de contenido en la Unidad de Aprendizaje Física Computacional, que se imparte en la Licenciatura en Física de la Facultad de Ciencias. En diferentes oportunidades se ha propuesto orientar el curso hacia temas que regularmente se revisan en análisis numérico; sin embargo, ha surgido la inquietud de proporcionar a los discentes herramientas numéricas que puedan serles útiles en su trabajo formativo y en el desarrollo de su trabajo final para graduarse. Si abordáramos solo temas de análisis numérico dejaríamos fuera y sin revisión algunos temas relevantes en la Licenciatura en Física. Al final, la propuesta que más satisfacía nuestras inquietudes fue la de hacer una revisión de temas incluidos en cursos obligatorios de Física y que son herramientas útiles para los discentes, no importando el área de especialización. También se incluyen códigos en Python para generar soluciones numéricas a cada problema planteado y que pueden servir como base para resolver otros más complejos de forma numérica.

Python es una herramienta poderosa empleada en diversos temas académicos, de investigación, de uso práctico (como el manejo de una gran cantidad de datos alojados en la nube digital), entre otros. En esta oportunidad se muestra su uso en algunos problemas que se revisan en la Licenciatura en Física, a fin de que los estudiantes tengan un primer acercamiento al manejo del intérprete Python a través de códigos simples aplicados a problemas concretos. Varios problemas físicos, cuyo planteamiento matemático requiere una solución numérica, no han sido considerados en este libro debido a que los conceptos físicos a manejar y las líneas de código en Python requieren mayor experiencia y, por tanto, van más allá del alcance de esta obra.

Solemos llamar física computacional a la rama de la física que, basándose en el uso de una computadora, propone diseñar y construir modelos de sistemas de interés para enseguida obtener una solución numérica reproducible y precisa, sobre todo en temas donde es muy difícil conseguir una solución analítica. Para tal fin, se hace uso de software de creación propia o de terceros. Una de las ventajas es que nos ayuda a calcular datos que permiten predecir un comportamiento de observables que, a su vez, dan información física del problema bajo estudio. De hecho, esta herramienta ha permitido llevar a cabo un análisis más detallado de sistemas complejos relacionados con la farmacéutica, la industria del petróleo, la industria alimentaria y la economía, por mencionar algunas áreas del conocimiento.

Es indudable que el desarrollo tecnológico ha permitido construir computadoras personales, laptops, servidores y clústeres que resultan ser muchas veces más poderosos que los primeros equipos desarrollados por la empresa IBM. Respecto al hardware, se ha podido reducir de manera significativa el tiempo de proceso debido al desarrollo de unidades de procesamiento gráfico, procesadores con múltiples núcleos y al paralelismo de códigos.

Ahora bien, acerca del software podemos mencionar que los lenguajes de programación han sufrido una evolución. A partir de la década de los cincuenta se crearon los primeros lenguajes de alto nivel, los cuales toman en cuenta la capacidad cognitiva de los humanos; por lo que su estructura semántica permite codificar algoritmos de manera natural. El lenguaje más utilizado para aplicaciones científicas fue Fortran, cuyo acrónimo tiene su origen en el título del manual The IBM Mathematical Formula Translating System; además, para aplicaciones profesionales fue Cobol, creado en 1969; y para el desarrollo de sistemas Pascal y C, a finales de los sesenta e inicios de los setenta, respectivamente. C también es considerado un lenguaje multipropósito, ya que se ha utilizado para desarrollar manejadores de bases de datos y lenguajes de programación. En la actualidad existen centenares de lenguajes de programación; algunos son ya obsoletos y otros han evolucionado con el objetivo de emplear más eficientemente los recursos del hardware y facilitar la tarea de programación. En el caso particular de Fortran, originalmente cada fabricante creó versiones para su propio hardware y a principios de los sesenta se lanzó Fortran IV, una versión estandarizada no dependiente de la computadora. Actualmente, Fortran tiene más de cincuenta años y, con su manejo dinámico de memoria, paralelización, estructuras de control y uso de funciones, sigue siendo uno de los principales lenguajes

usados para programación científica. Por otro lado, C, cuyo objetivo principal fue poder contar con un compilador que garantizara la consistencia y velocidad de una computadora con sistema operativo Unix, al poder mezclar lo más característico de los lenguajes de alto nivel con algunos rasgos de los lenguajes de bajo nivel, el rendimiento que ofrece permite que sea utilizado también para el desarrollo de software científico. Su predecesor C++, creado en la década de los ochenta, agregó la abstracción, encapsulación y ocultación, características de la programación orientada a objetos.

En este punto, es importante precisar la diferencia entre los compiladores y los intérpretes. Sin importar el lenguaje de alto nivel que se utilice para dar órdenes a la computadora, es necesario que se haga la traducción al código binario que entiende la unidad central de proceso (CPU). Existen dos formas diferentes de llevar a cabo esta tarea: una de ellas a partir de los compiladores, que se encargan de traducir el programa completo a código binario, dejándolo listo para ser comprendido por el CPU, por lo que se logra mayor rapidez en la ejecución y mejor rendimiento de la memoria, con la desventaja de que la traducción se realiza a lenguajes máquina específicos tales como OS X, Windows o Linux; por otro lado, los intérpretes traducen instrucción por instrucción en tiempo de ejecución; es decir, a medida que el programa se ejecuta favorece la interacción, depuración y puesta a punto del programa, siendo irrelevante el sistema operativo de la computadora (multiplataforma), con la desventaja de la velocidad de ejecución y la memoria utilizada.

En la actualidad, los lenguajes C, C++ y Java son lenguajes de programación, utilizados por empresas que ocupan profesionales de la programación, a fin de aprovechar la ventaja del poder de las herramientas de computación para el desarrollo de software libre o comercial, que ofrecen soluciones numéricas de sistemas de interés. Este tipo de software nos permite hacer cálculos numéricos en un tiempo corto, aprovechando todas las ventajas tecnológicas del hardware. No significa que debamos usar tal software como caja negra; es decir, sin saber cómo o por qué funciona.

Por otra parte, Python fue creado por Guido van Rossum, un programador nacido en Países Bajos en 1990. Este es un lenguaje de programación de alto nivel, con la característica de ser software libre al igual que C y Fortran; es ejecutado por un intérprete, fácil e intuitivo. También se le considera un lenguaje multiparadigmas, ya que proporciona facilidades para realizar una programación imperativa, funcional y orientada a objetos. Su principal filosofía es ser legible por cualquier persona

con conocimientos básicos de programación, siendo ideal para ser utilizado en las comunidades científicas y académicas (esta última en su carácter de formador de recursos humanos). En particular, en esta obra estamos interesados en usar Python y Fortran 77 en la elaboración de funciones que nos ayuden a conseguir el objetivo de cada ejercicio. Vale la pena mencionar que los códigos incluidos no están optimizados, en el sentido de que algunas líneas código pueden ser suprimidas y los procesos de cálculo posiblemente ser acelerados; sin embargo, dichos códigos están escritos de manera didáctica con la intención de ser lo más claros posible en el procedimiento. La razón al usar la versión 77 de Fortran es la facilidad de seguirlo, pues usa instrucciones reconocibles y simples, esto es de gran ayuda para estudiantes con poca experiencia en la programación.

Por todo esto, mencionamos que el espíritu de la física computacional es guiar a los estudiantes para tener una noción básica de las teorías, teoremas, métodos y técnicas que dan soporte a dicho software, el cual permite realizar cálculos numéricos útiles que requieren computadoras poderosas y mucho tiempo de cómputo para obtener una precisión aceptable. En este sentido, el contenido de esta obra provee de material didáctico que sirve como apoyo, el cual está dirigido a estudiantes de la Licenciatura en Física, tratando de ser guía para resolver problemas físicos y matemáticos de manera numérica, haciendo uso de software, que ya ha sido revisado en cursos previos. No es una guía para programar de manera óptima, solo se muestra el uso de algunas herramientas (software) que son de utilidad.

BIBLIOTECAS DE PYTHON

El intérprete Python reconoce la sintaxis de más de un lenguaje de programación (tal es el caso de C). Además, tiene la ventaja de que nos permite elaborar nuestros propios procedimientos o bibliotecas, aunque también tiene incluidas bibliotecas que solo es necesario importar para invocarlas. Es indispensable tener una idea de qué tipo de bibliotecas puede usarse con Python [1-4].

A continuación, se hace una descripción de las principales bibliotecas que se usarán en los códigos que presentaremos a lo largo del libro:

- Math: Proporciona funciones para operaciones matemáticas especializadas. Usa varias funciones reconocidas en la plataforma del lenguaje C para poder emplear operaciones matemáticas con valores de coma flotante, donde se incluyen funciones trigonométricas y logaritmos.
- NumPy: Permite llevar a cabo analisis de datos de buena forma, debido a que ayuda a un intercambio de datos entre diferentes algoritmos, lo que se conoce como el manejo de una estructura de datos universal.
- SciPi: Es una biblioteca de algoritmos y herramientas matematica, incluye interfases a bibliotecas científicas: BLAS, LAPACK, ODR, etc.
- Sympy: Es una biblioteca que puede hacer evaluaciones algebraicas, la diferenciación, expansiones, calcular números complejos, entre otros. Es empleada para manejar matematica simbolica.
- Sys: Es un módulo encargado de proveer variables y funcionalidades, directamente relacionadas con el intérprete.
- Scrapy: Es "open source", útil para la extracción de datos de páginas web.
- wxPyhton: Es un conjunto de bibliotecas de interfaces gráficas (programadas en C++), destinadas a crear GUIs de forma simple.
- Pillow o PIL: Para manejo de imágenes (Python Image Library). Sirve para abrir, modificar y almacenar imágenes de diferentes formatos, así como manipular los pixeles, trabajar con máscaras, transparencias, dimensiones, agregar texto, aplicar filtros, entre otras funciones.

- SQLAlchemy: Gestiona base de datos. Nos permite trabajar con las bases de datos mediante objetos; es decir, es un ORM. Es útil para crear, modificar, consultar y eliminar tablas; así como crear, leer, actualizar y eliminar registros.
- Request: Nos permite realizar peticiones HTTP, crear, enviar y recibir paquetes, modificar su contenido, trabajar con sesiones, cookies, formularios e, inclusive, trabajar con autenticación OAuth.
- Pandas: Permite manejar y analizar datos que pueden ser pocos o una cantidad importante.
- Matplotlib: Se usa para hacer gráficos, tiene muchos valores predeterminados incorporados y utiliza la instrucción plot().
- Pylab: Es un conjunto de varias bibliotecas entre las que se incluyen numpy, scipy, sympy, pandas, matplotlib, ipython. Con la ayuda de esta herramienta es posible hacer compatible Python con el software MatLab.
- Linear Regression: Ayuda a llevar a cabo ajustes de datos de interés a un modelo lineal.
- Scipy. interpolate. interp1d: Es una clase que regresa una función y usa la interpolación para hallar el valor de puntos nuevos.
- From sympy. Utilities. lambdify import lambdify: Este módulo proporciona funciones convenientes para transformar expresiones sympy en funciones lambda que se pueden usar para calcular valores numéricos rápidamente.
- Statistics: Ayuda a calcular estadísticas con números reales.

1. ESTADÍSTICA DE DATOS

Al hacer mediciones en un experimento o al generar datos mediante teorías o simulaciones por computadora, a menudo es necesario hacer una serie de tratamientos estadísticos para depurarlos y poder ofrecer un resultado numérico que sea representativo y lo más preciso posible. En particular, solo mostramos dos propiedades importantes a calcular, la media aritmética y la desviación cuadrática media.

Promedio y desviación estándar

El promedio de un conjunto de números, también conocido como la media aritmética, es una de las tres cantidades comúnmente estimadas en caso de que el número de datos generados no sea demasiado grande y se requiera una clasificación particular. La desviación cuadrática media es la raíz cuadrada de la varianza, y esta a su vez nos da información de cuánto se desvía un conjunto de datos respecto a su media.

La media aritmética de un conjunto N de números x_i es obtenida de manera tradicional mediante la expresión siguiente:

$$\overline{x} = \frac{1}{\mathrm{N}} \sum_{i=1}^{N} x_i \qquad 1.1)$$

Además, la desviación estándar se escribe como sigue:

$$\sigma = \sqrt{\frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} (x_i - \overline{x})^2} \quad \text{o} \quad \sigma = \sqrt{\left| \overline{x}^2 - \overline{x^2} \right|} \qquad 1.2)$$

De manera que al elaborar un reporte mostrando un promedio de una medición y su respectiva desviación, se escribe $\bar{x} \pm \sigma$ [5].

En el ejemplo de abajo se muestra el código 1.1 en Python, que calcula el promedio de datos usando las ecuaciones (1.1) y (1.2), haciendo las operaciones algebraicas explícitamente. Este es un procedimiento interactivo, ya que el usuario ingresa los datos.

```
#código1.1
import math
suma = 0
sumad = 0
promedio = 0
promediod = 0
print('     ')
n = int(input("Número de datos" ))
k = 0
while(k < n):
   x= float(input("valor de x" ))
   suma = suma + x
   sumad = sumad + x*x
   k = k + 1
promedio = suma / n
promsqrt = promedio*promedio
promediod= sumad / n
sig2     = abs(promsqrt-promediod)
sig      = math.sqrt(sig2)

print('    ')
print('El número de datos = ' + str(n))
print('El promedio          = ' + str(promedio))
print('La desviacion estandar = ' + str(sig))
print('    ')
```

Usaremos el primer código en Python que hemos incluido para explicar su estructura y sus componentes. Se muestra el inicio del código en Python en forma aislada. En las siguientes líneas de código vemos el llamado de la biblioteca *math* (import math), que es incluida para este ejercicio; particularmente se usa para reconocer la raíz cuadrada.

```
suma = 0
sumad = 0
promedio = 0
promediod = 0
print(‘     ‘)
n = int(input(“Número de datos” ))
k = 0
while(k < n):
   x= float(input(“valor de x” ))
   suma = suma + x
   sumad = sumad + x*x
   k = k + 1
promedio = suma / n
promsqrt = promedio*promedio
promediod= sumad / n
sig2     = abs(promsqrt-promediod)
sig      = math.sqrt(sig2)
```

Las operaciones algebraicas incluidas describen las ecuaciones (1.1) y (1.2). Enseguida, se muestran las líneas de código que corresponden a la presentación de los resultados obtenidos.

```
print(‘    ‘)
print(‘El número de datos     = ‘ + str(n))
print(‘El promedio            = ‘ + str(promedio))
print(‘La desviación estandar = ‘ + str(sig))
print(‘    ‘)
```

Este es el orden y el procedimiento que se usa en todos los códigos incluidos en este libro.

Regresando al ejercicio inicial, mencionamos que si guardamos el procedimiento anterior en el archivo *av.py*, entonces, para ejecutar dicho procedimiento, escribimos las siguientes líneas: *python2 av.py* o *python3 av.py*, dependiendo de la versión de Python que se tenga instalada. La instrucción contenida en el código anterior, *while(k<n),* hace referencia a que se repetirá una misma acción mientras se cumpla la condición *k<n,* De esta manera, se introduce el número de datos que queremos usar y luego cada uno de los datos que utilizaremos para estimar el promedio.

En el siguiente ejercicio se calcula el promedio de números que se leen de un archivo *datos.dat*. Es necesario editar previamente un archivo que contenga los datos a los que se les quiere extraer la media aritmética y su desviación estándar.

```
#código 1.2
import math
suma = 0
sumad = 0
promedio = 0
archivo = open("datos.dat","r")
lista = archivo.readlines()
print("  ")
print("Leyó datos del archivo datos.dat")
k = 0
for line in lista:
    x = float(line)
    suma = suma + x
    sumad = sumad + x*x
    k = k + 1
prom = suma/k
promk = prom*prom
promd = sumad/k
valabs = abs(promk-promd)
sig = math.sqrt(valabs)

print("  ")
print("Número de datos", k)
print("El promedio es", prom)
print("La desviación es", sig)
print("  ")
archivo.close()
```

La línea *archivo=open("datos.dat","r")* nos ayuda a abrir un archivo externo (*datos.dat*) y nombrarlo *archivo* de manera interna; *r* significa que se podrá leer (*read*) la información del archivo externo. Después se leen los datos por línea *lista = archivo.readlines()*. Para hacer la suma de datos se usa el comando *for*, de manera que las instrucciones deben anidarse. Al final, cerramos el archivo para evitar introducir información no deseada.

Mencionamos que la mediana de un conjunto de datos es aquel número que aparece en el medio de la lista de datos, después de acomodarlos de forma ascendente o progresiva. Finalmente, la moda de un conjunto de datos es el número de veces que aparece un mismo número en un listado de datos. Existen las funciones media, mediana, moda, varianza y desviación cuadrática media en bibliotecas de Python, lo que permite calcular tales cantidades con pocas líneas de programación. En el ejercicio de abajo se muestra el uso de tales funciones considerando 10 valores numéricos.

```
#código 1.3
import numpy
import statistics

variable = [9,6,7,8,11,8,10,7,9,10]
xmediana = numpy.median(variable)
xmedia = numpy.mean(variable)
xmoda = statistics.mode(variable)
xdesviación = numpy.std(variable)
xvarianza = numpy.var(variable)
print("   ")
print("mediana  = ",xmediana)
print("media      = ",xmedia)
print("moda       = ",xmoda)
print("std          = ",xdesviación)
print("varianza  = ",xvarianza)
```

Ejercicios

Usando los códigos en Python 1.1 y 1.2, realizar los siguientes ejercicios:
[1] Calcular el promedio de los números: 1,3,5,7,2,9,11,65,23,22,15
[2} Calcular la desviación estándar en los números: 1,3,5,7,2,9,11,65,23,22,15
[3] Determinar la mediana de los números: 2,4,3,4,5,6,8

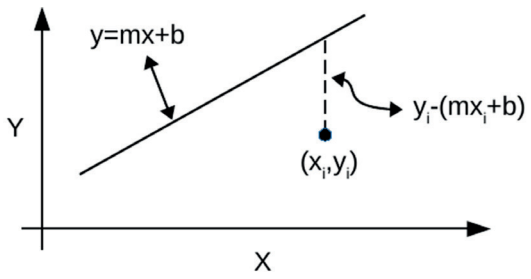
Método de mínimos cuadrados

Es un método numérico que se usa para ajustar datos y, de esta manera, identificar un comportamiento generalizado. Dicho método proporciona un criterio para obtener la mejor de las rectas que es el ajuste de los datos $y_i = mx_i + b$. Particularmente, se aspira a encontrar la recta que mejor se adapte a la serie de datos (x_i,y_i) corriendo el contador i de 1 a n, así que se estima la desviación de los datos respecto de la recta y se pide que la suma de los cuadrados de las diferencias sea lo más pequeña posible. [6]

$$y_i - (mx_i + b) \qquad 1.3)$$

Geométricamente podemos representar dichas diferencias, como se observa en la figura siguiente:

Figura 1.1. Un dato y una recta,
así como su diferencia entre ellos a un valor de x dado.



Ahora bien, si consideramos una colección de datos dispuestos de manera aleatoria y calculamos su distancia hacia la recta de cada uno de ellos, entonces podemos escribir:

$$f = [y_1 - (mx_1 + b)]^2 + [y_2 - (mx_2 + b)]^2 + [y_3 - (mx_3 + b)]^2 + \ldots + [y_n - (mx_n + b)]^2 \quad 1.4)$$

o de forma compacta:

$$f = \Sigma_{n=1}^{n} [y_i - (mx_i + b)]^2 \quad 1.5)$$

Para optimizar la función y hallar la desviación más pequeña debemos maximizar la función. Esta tiene dos variables, *m* y *b*; así que se hallan los puntos críticos en cada caso:

$$\frac{\partial f}{\partial b} = 0 \quad \text{y} \quad \frac{\partial f}{\partial m} = 0 \quad 1.6)$$

Consideramos primero la derivada respecto a b:

$$\frac{\partial f}{\partial b} = \Sigma_{i=1}^{n} 2[y_i - (mx_i + b)](-x_i) \quad 1.7)$$

Enseguida, igualamos a cero:

$$2\,\Sigma_{i=1}^{n}[y_i - (mx_i + b)](x_i) = 0 \quad 1.8)$$

Luego, aplicamos la ley distributiva:

$$\Sigma_{i=1}^{n} y_i - m\,\Sigma_{i=1}^{n} x_i - b\,\Sigma_{i=1}^{n} 1 = 0 \quad 1.9)$$

Para luego despejar *b:*

$$b = \tfrac{1}{n}\,\Sigma_{i=1}^{n} y_i - m\left[\tfrac{1}{n}\,\Sigma_{i=1}^{n} x_i\right] \quad 1.10)$$

Que también se puede escribir de la siguiente manera:

$$b = \bar{y} - m\bar{x} \quad 1.11)$$

Donde:

$$\bar{y} = \frac{1}{n} \Sigma_{i=1}^{n} y_i \qquad 1.12)$$

Y además:

$$\bar{x} = \frac{1}{n} \Sigma_{i=1}^{n} x_i \qquad 1.13)$$

Por otro lado, ahora llevaremos a cabo la derivada de *f* respecto a *m*:

$$\frac{\partial f}{\partial m} = \Sigma_{i=1}^{n} 2[y_i - (mx_i + b)](-x_i) \qquad 1.14)$$

Si igualamos a cero y sustituimos la ecuación (1.11):

$$\Sigma_{i=1}^{n} 2[y_i - (mx_i + \bar{y} - m\bar{x})](x_i) = 0 \qquad 1.15)$$

Después de hacer un poco de álgebra llegamos a la ecuación:

$$\Sigma_{i=1}^{n} y_i x_i - \Sigma_{i=1}^{n} mx_i x_i - \Sigma_{i=1}^{n} \bar{y} x_i + \Sigma_{i=1}^{n} m\bar{x} x_i = 0 \qquad 1.16)$$

$$\Sigma_{i=1}^{n} x_i [y_i - \bar{y}] - m \Sigma_{i=1}^{n} x_i [x_i - \bar{x}] \; 0 \qquad 1.17)$$

Despejando *m* de esta última ecuación obtenemos finalmente:

$$m = \frac{\Sigma_{i=1}^{n} x_i [y_i - \bar{y}]}{\Sigma_{i=1}^{n} x_i [x_i - \bar{x}]} \qquad 1.18)$$

El problema se reduce ahora a evaluar *b* y *m*, que representan los mínimos que maximizan la función (1.3). Es decir, hacemos uso de valores numéricos asociados a la variable independiente *X* y a la variable dependiente *Y.* Para fijar ideas, abordemos un primer ejemplo numérico.

Ejemplo 1. Calcular la curva que mejor ajusta la serie de números que aparecen en la tabla 1.1.

Tabla 1.1

X	Y	Y'
1	0.10	0.112
2	0.13	0.114
3	0.12	0.116
4	0.11	0.118
5	0.12	0.120

El paso inicial consiste en calcular el promedio de los valores de la variable Y y de la variable X. Para este caso en particular, calculamos dichos valores promedio usando las ecuaciones (1.12) y (1.13), respectivamente.

$$\bar{y} = \frac{0.10+0.13+0.12+0.11+0.12}{5} 0.116 \qquad \bar{x} = \frac{1+2+3+4+5}{5} = 3 \tag{1.19}$$

Luego, hacemos uso de la ecuación (1.18) para calcular el valor numérico de *m*:

$$m = \frac{1(0.1-0.116)+2(0.13-0.116)+3(0.12-0.116)+4(0.11-0.116)+5(0.12-0.116)}{1(1-3)+2(2-3)+3(3-3)+4(4-3)+5(5-3)} = 0.002 \tag{1.20}$$

Después. hacemos uso de la ecuación (1.11) para determinar *b*:

$$b = 0.116 - 0.002 * 3 = 0.11 \tag{1.21}$$

Finalmente, la recta que estamos buscando es:

$$y' = (0.002)\, x + 0.11 \tag{1.22}$$

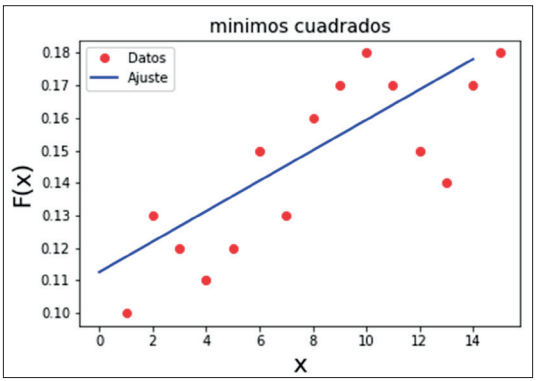
Esta es la solución numérica que podemos calcular con cualquier calculadora. La recta que describe la ecuación (1.22) es el mejor ajuste lineal de los datos numéricos que se encuentran en la tabla 1.1.

Por otra parte, los datos incluidos en la última columna de la tabla 1.1 se pueden obtener también con un procedimiento simple que se escribe en un código en Python, donde las bibliotecas *numpy*, *pandas*, *scipy.interpolate* y *matplotlib* son incluidas para este ejercicio. El código en Python a usar es el que se muestra a continuación, en donde se leen datos de un archivo externo.

```
#código 1.4

import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

data=pd.read_csv('datos.txt',header=0, delim_
whitespace=True)

x=data.iloc[:,0]
y=data.iloc[:,1]

plt.plot(x,y,'ro')
plt.ylabel('F(x)',fontsize=18)
plt.xlabel('x',fontsize=18)

#ajuste a un polinomio grado uno
coeficientes=np.polyfit(x,y,1)

#calcula los coeficientes de un ajuste lineal
recta=np.poly1d(coeficientes)
ys=recta(x)

print(" ")
print("Ecuación ajustada",recta)

plt.title("mínimos cuadrados",fontsize=18)
plt.plot(ys,'green')
plt.legend(('Datos', 'Ajuste'),loc="upper left")
plt.savefig("MínimosCuadrados.png")
plt.show()
```

En este ejercicio, se leen los datos del archivo *datos.txt* y se considera que hay por lo menos dos columnas de datos separados por un espacio en blanco. Se asignan los datos a las variables *X* y *Y*. Se usa la instrucción para determinar un polinomio que ajusta a una línea recta. Al final se genera la figura correspondiente *MínimosCuadrados.png*.

Figura 1.2. Ajuste de datos a través de un polinomio lineal. Los datos fueron leídos de un archivo externo.



Por supuesto, si se consideran muchos más datos, el ajuste lineal puede resultar ser mejor que el mostrado en el ejemplo 1. En la tabla 1.2 se muestran los datos numéricos usados en el ejercicio anterior.

Tabla 1.2

X	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Y	0.10	0.13	0.12	0.11	0.12	0.15	0.13	0.16	0.17	0.18	0.17	0.15	0.14	0.17	0.18

Por otro lado, si se generan varios datos de manera aleatoria y se requiere llevar a cabo un ajuste lineal, podemos usar el procedimiento en Python que se muestra a continuación. Las bibliotecas *numpy*, *pylab* y *sklearn* se incluyen para llevar a cabo este ejercicio.

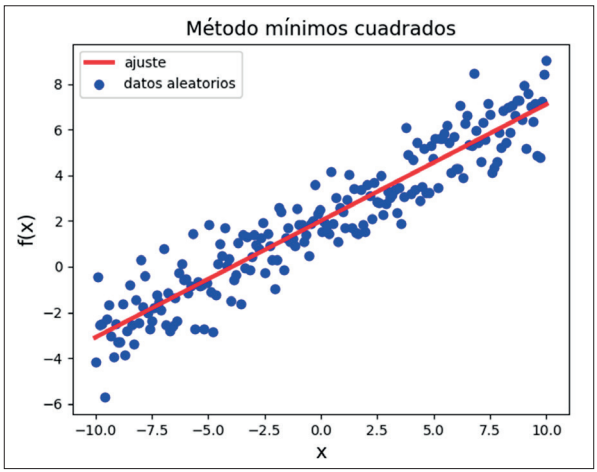
```
#código 1.5
import numpy as np
import pylab as plt
from sklearn import linear_model

# ajuste lineal de datos aleatorios
xmin, xmax = -10, 10
npoints = 200
X = [[i] for i in np.linspace(xmin, xmax,
npoints)]
Y = 2 + 0.5 * np.linspace(xmin, xmax,
npoints) \
    + np.random.randn(npoints, 1).ravel()
# ajuste
clf = linear_model.LinearRegression()
clf.fit(X, Y)
```

```
plt.scatter(X, Y, color='blue', label="datos
aleatorios")
plt.plot(X, clf.predict(X), color='red',
linewidth=3, label="ajuste")
plt.xticks(visible=True)
plt.yticks(visible=True)
plt.title("Método mínimos
cuadrados",fontsize=14)
plt.xlabel("x",fontsize=14)
plt.ylabel("f(x)",fontsize=14)
plt.legend(loc="upper left")
plt.savefig("Minimos_cuadrados.png")
plt.show()
```

En el código 1.5 se usan funciones que se encuentran ya definidas en las bibliotecas invocadas, lo cual nos da la oportunidad de hacer cada vez más corto el código

de programación. Particularmente, hacemos uso de la declaración *linear_model.LinearRegression()* para ajustar a los datos generados aleatoriamente. Al final del código en Python se genera la figura 1.3, que incluye los datos mencionados junto con el ajuste lineal.

Figura 1.3. Ajuste de datos a través de un polinomio lineal. Los datos fueron generados aleatoriamente.



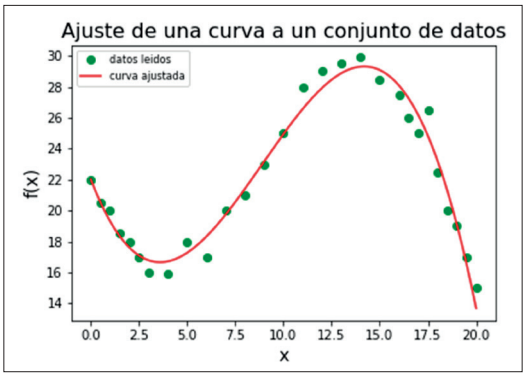
Vale la pena mencionar en esta sección que el ajuste lineal no es el único que puede ser obtenido usando las bibliotecas de Python. A manera de ejemplo, en el código 1.6 mostramos las instrucciones para conseguir un ajuste a un conjunto de datos leídos de un archivo externo con un código en Python, y que no corresponden a un comportamiento lineal; en particular, se muestra el ajuste a una curva sinusoidal u oscilante. Para este ejercicio se usan las bibliotecas *numpy* y *matplotlib*. La primera de ellas nos permite hacer el ajuste de los datos leídos de un archivo externo (*datos.dat*) previamente obtenidos. La segunda de las bibliotecas nos ayuda a graficar los datos leídos y la curva ajustada a dichos datos.

```
#código 1.6
import numpy as np
import matplotlib.pyplot as plt
x = []
y = []
archivo=open("datos.dat", "r")
lista = archive.readlines()

for line in lista:
    lines = [i for i in line.split()]
    x.append(float(lines[0]))
    y.append(float(lines[1]))
eq = np.poly1d (np.polyfit (x, y, 3))
ajuste = np.linspace (0, 20, 2000)

plt.plot (x,y, 'go', markersize='6', label='datos leidos')
plt.plot (ajuste, eq(ajuste), 'r', label='curva ajustada')
plt.xlabel("x",fontsize = 14)
plt.ylabel("f(x)",fontsize = 14)
plt.legend(fontsize=8, loc="upper left")
plt.title('Ajuste de una curva a un conjunto de datos',
fontsize=16)
plt.savefig("ajuste-curva.png")
plt.show()

print ('La ecuación ajustada:')
print (eq)
```

La instrucción *np.poly1d* está definida en la biblioteca de *numpy* y considera a X y Y como las variables independiente y dependiente, respectivamente. Además, el número "3" denota el orden del polinomio que se ajusta a los datos leídos al inicio del código.

Con el propósito de hacer una gráfica que centre los datos y la curva de ajuste, se incluye la línea que denota el rango para la variable "x" de 0 a 20. Finalmente, se menciona que la figura resultante se guarda en el archivo *ajuste-curva.png*. La figura 1.4 muestra una curva no lineal que se ajusta a un conjunto de datos.

Figura 1.4. Ajuste de datos a través de un polinomio no lineal. Los datos fueron leídos de un archivo externo.



Ejercicios

[1] Calcular el ajuste a una línea recta de 200 datos generados aleatoriamente.
[2] Calcular el ajuste a una línea recta de al menos 200 datos leídos de un archivo externo.
[3] Calcular el ajuste a una curva de tercer orden de al menos 200 datos leídos de un archivo externo.

2. CINEMÁTICA

La cinemática es una rama de la física que se encarga de estudiar el movimiento de los cuerpos, junto con su trayectoria en función del tiempo [7,8], no importando la causa de dicho movimiento. Tradicionalmente, se revisan los tipos de movimientos abarcados por esta área de la física y sus correspondientes ecuaciones que describen dichos movimientos. De hecho, mencionamos aquí algunos de los movimientos que se estudian comúnmente: movimiento rectilíneo uniforme, movimiento parabólico, movimiento rectilíneo uniformemente acelerado, movimiento circular uniforme, movimiento circular uniformemente acelerado, movimiento armónico y movimiento armónico complejo.

Por lo general, se plantean problemas que involucran uno o más de un tipo de movimiento y se ofrecen datos iniciales para poder predecir trayectorias o simplemente arrojar un resultado numérico de una variable de interés. Sin embargo, no es tradición hacer un gráfico preciso de trayectorias descritas y menos hacer una animación de tales trayectorias. Justamente con herramientas que se usan en la física computacional, podemos desarrollar lo mencionado. En esta oportunidad vamos a revisar las ecuaciones concernientes a la caída libre, tiro parabólico y movimiento circular uniformemente acelerado. Posteriormente generaremos datos numéricos para graficar estas trayectorias, así como los archivos para hacer una animación.

Caída libre

Consideremos un objeto que dejamos caer desde una altura y_0 de acuerdo con la cinemática, la trayectoria seguida por tal objeto es:

$$y = y_0 + v_0\, t - \tfrac{1}{2} g t^2 \qquad 2.1)$$

siendo g la constante de la gravedad terrestre, t el tiempo y v_0 la velocidad inicial. Ahora, si queremos generar datos para graficar la trayectoria del objeto al caer, debemos

asignar valores a la variable independiente *t,* hasta llegar al suelo. A continuación, incluimos un código en Python que nos permite calcular los valores numéricos de las variables X y Y que describen la trayectoria de un objeto en caída libre.

```
#código 2.1
archivo = open("caida.xyz","w")
archiva = open("caida.dat","w")
print("    ")

x0 = float(input('x0: '))
y0 = float(input('y0: '))
n  = int(input('npasos: '))
print("    ")

sigmar = 3.16
grave = 9.81
dt = 0.005
tiempo = 0.0

for i in range(0,n+1,1):
    archivo.write(str(1) + '  ' + '\n');
    archivo.write(str('    ') + '  ' + '\n');
    x = 0.0
    z = 0.0
    y = y0 - (grave*tiempo*tiempo)/2.0
    y = y*sigmar
    archiva.write(str(x) + '   ' + str(y) + '\n');
    archivo.write("H" + '   ' + str(x) + '   ' + str(y) + '   ' + str(z) + '\n');
    tiempo = tiempo + dt

archivo.close;
archiva.close;
print("\nArchivo caida.xyz generado");
print("\nArchivo caida.dat generado");
```

De la ejecución del código 2.1 se generan dos archivos: *caída.xyz* y *caída.dat,* el primero de ellos contiene los datos para las coordenadas X y Y, en un formato que permite ver la "animación" de tal movimiento con el software libre VMD [9]. El segundo archivo contiene los mismos datos, pero con un formato que es posible graficar con el software libre grace. [10] Es importante destacar que el segundo archivo contiene la trayectoria del movimiento de caída libre y no una animación.

TIRO PARABÓLICO

En un movimiento parabólico se observa que se requiere un ángulo de inclinación (θ) y que la gravedad solo afecta a la trayectoria vertical. Las velocidades iniciales en los ejes *x-y* se escriben como:

$$v_{0x} = v_0 cos(\theta) \tag{2.2}$$

y:

$$v_{0y} = v_0 sen(\theta) \tag{2.3}$$

Para un tiempo posterior, las velocidades son:

$$v_x = v_{0x} \tag{2.4}$$

y:

$$v_y = v_0 sen(\theta) - gt \tag{2.5}$$

Las posiciones en función del tiempo se escriben de la siguiente manera

$$x = x_0 + v_x t \tag{2.6}$$

y:

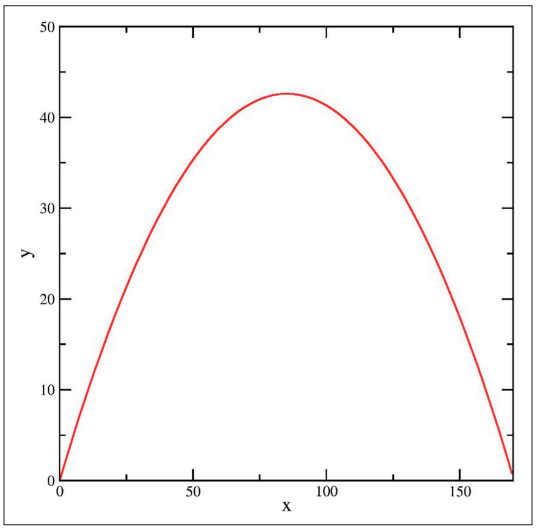
$$y = y_0 + v_y t - \frac{1}{2} g t^2 \tag{2.7}$$

Para el ejemplo que estamos considerando no se toma en cuenta la resistencia del aire o algún otro efecto sobre el movimiento parabólico. Ahora, si queremos generar los datos para graficar la trayectoria, así como generar datos para llevar a cabo una animación, seguimos el procedimiento que se muestra en el código en Python que a continuación se presenta.

```
#código 2.2
import math
archivo = open("tiroparabolico.xyz","w")
archiva = open("tiroparabolico.dat","w")

print("    ")

v0  = float(input('v0: '))
theta = float(input('theta: '))
x0 = float(input('x0: '))
y0 = float(input('y0: '))
n  = int(input('npasos: '))
print("    ")

# PROCEDIMIENTO
sigmar = 3.16
pi = 3.1416
fact = pi/180.0
theta = theta*fact
grave = 9.81
v0x = v0*math.cos(theta)
v0y = v0*math.sin(theta)
xmax = (v0*v0/(2.0*grave))*math.sin(2.0*theta)
ymax = (v0*v0/(2.0*grave))*math.sin(theta)*math.
sin(theta)
print("xmax = " + str(xmax))
print("ymax = " + str(ymax))
dt = 0.05
tiempo = 0.0
for i in range(0,n+1,1):
    cotay = 0.0
    archivo.write(str(1) + '  ' + '\n');
    archivo.write(str('    ') + '  ' + '\n');
    x = x0 + v0x*tiempo
    y = y0 + v0y*tiempo - 0.5*tiempo*tiempo*grave
    x = x*sigmar
    y = y*sigmar
    z = 0.0
    tiempo = tiempo + dt
    if(y >= cotay):
      archiva.write(str(x) + '   ' + str(y) + '\n');
      archivo.write("H" + '   ' + str(x) + '   ' + str(y)
+ '   ' + str(z) + '\n');

archivo.close;
archiva.close;
print("\nArchivo tiroparabolico.xyz generado");
print("\nArchivo tiroparabolico.dat generado");
```

El procedimiento de arriba genera un archivo *tiroparabólico.xyz* que contiene una animación compatible con el software VMD. [9] La trayectoria de un movimiento parabólico se aloja en el archivo *tiroparabólico.dat* y se muestra en la figura 2.1.

Figura 2.1. Trayectoria del movimiento parabólico.



Tiro parabólico con resistencia

Ahora nos planteamos la posibilidad de incluir la resistencia del aire a través de un coeficiente (b) y las coordenadas X y Y. Las ecuaciones de movimiento las escribimos a continuación:

$$x = x_0 + \frac{v_{0x}}{b}[1 - \exp(-bt)] \qquad 2.8)$$

y:

$$y = y_0 + \frac{1}{b}\left(\frac{g}{b} + v_{0y}\right)[1 - \exp(-bt)] - \frac{g}{b}t \qquad 2.9)$$

donde *g* es la constante de gravedad y *t* es el tiempo. Este es un modelo simple de tal tipo de movimiento, lo cual no significa que sea un modelo general. Se observa que la resistencia del aire al movimiento parabólico influye en ambas coordenadas, X y Y.

A continuación, se incluye el código 2.3 en Python que contiene las ecuaciones que describen el tiro parabólico con resistencia del aire a través de la incorporación de una constante (b). En dicho código se incluyen las bibliotecas *sys, math, sympy,* y *numpy*.

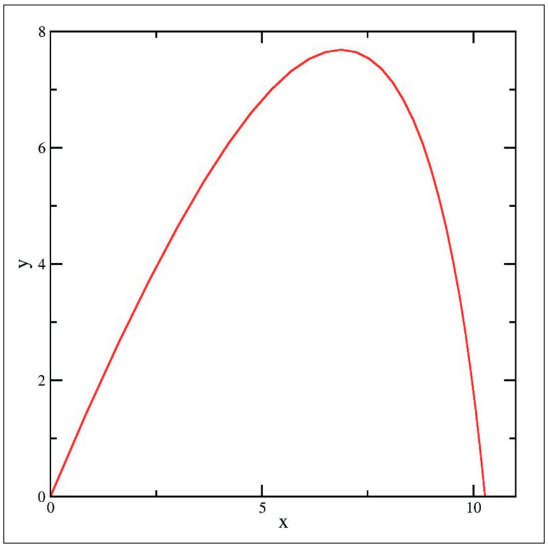
```
#código 2.3
import math
archivo = open("tiroparabolicoConAire.
xyz","w")
archiva = open("tiroparabolicoConAire.
dat","w")
print("    ")
v0  = float(input('v0: '))
theta = float(input('theta: '))
x0 = float(input('x0: '))
y0 = float(input('y0: '))
b  = float(input('b: '))
n  = int(input('npasos: '))
print("    ")

# PROCEDIMIENTO
sigmar = 3.16
pi = 3.1416
fact = pi/180.0
theta = theta*fact
grave = 9.81
b2 = b*b

v0x = v0*math.cos(theta)
v0y = v0*math.sin(theta)

dt = 0.05
tiempo = 0.0
for i in range(0,n+1,1):
    cotay = 0.0
    archivo.write(str(1) + '  ' + '\n');
    archivo.write(str('    ') + '  ' + '\n');
    x = x0 + (v0x/b)*(1.0-math.exp(-b*tiempo))
    y = y0 + (1.0/b2)*(grave+b*v0y)*(1.0-math.
exp(-b*tiempo))-tiempo*grave/b
    x = x*sigmar
    y = y*sigmar
    z = 0.0
    tiempo = tiempo + dt
    if(y >= cotay):
     archiva.write(str(x) + '   ' + str(y) + '\n');
     archivo.write("H" + '        ' + str(x) + '   ' +
str(y) + '   ' + str(z) + '\n');
archivo.close;
archiva.close;

print("\nArchivo tiroparabolico.xyz generado");
print("\nArchivo tiroparabolico.dat generado");
```

Aunque la variable independiente es el tiempo, hacemos notar que la trayectoria no precisa de grafiar el tiempo, sino solo las coordenadas X, Y y Z, para el lapso en donde se analiza el movimiento parabólico con resistencia del aire.

No olvidar que se deben anidar las instrucciones después del condicionante *if* o de la instrucción *for*.

Al ejecutar el código 2.3 de Python, se genera un archivo que contiene los datos de la trayectoria que describe un movimiento parabólico con resistencia del aire *tiroparabólicoConAire.dat*; así como un archivo que contiene varias fotos del objeto en movimiento *tiroparabólicoConAire.xyz*, y con el cual es posible observar una animación haciendo uso del software libre VMD. En la figura 2.2 se puede advertir la trayectoria ya mencionada.

Figura 2.2. Trayectoria del movimiento parabólico con resistencia



Movimiento circular uniformemente acelerado

Es aquel movimiento de un objeto que describe una circunferencia con aceleración angular (α) constante. La velocidad angular en función del tiempo se escribe como:

$$\omega = \omega_0 + \alpha t \qquad 2.10)$$

siendo w0 la velocidad angular inicial y *t* el tiempo. La trayectoria angular se escribe en términos de la velocidad y aceleración angular al tiempo (t).

$$\varphi = \varphi_0 + \omega_0 t + \frac{1}{2}\alpha t^2 \qquad 2.11)$$

A partir de la evaluación del ángulo que describe el movimiento circular, se estiman las coordenadas cartesianas de la trayectoria

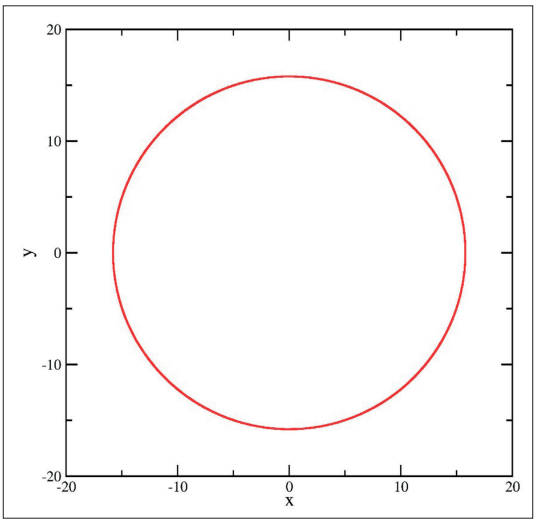
$$x = r\cos(\varphi) \qquad 2.12)$$

y:

$$y = r\mathrm{sen}(\varphi) \qquad 2.13)$$

donde *r* es el radio del círculo que describe la trayectoria. La trayectoria del movimiento uniformemente circular se observa en la figura siguiente

Figura 2.3. Trayectoria del movimiento circular uniformemente acelerado.



Para generar la trayectoria circular de un objeto con aceleración angular constante, se escribe un código en Python que contiene las ecuaciones (2.10)-(2.13), el cual se muestra a continuación:

```
#código 2.4
import math
archivo = open("circular.xyz","w")
archiva = open("circular.dat","w")

print("    ")
radio  = float(input('radio: '))
phi0 = float(input('phi0: '))
w0 = float(input('w0: '))
alpha = float(input('alpha: '))
n  = int(input('npasos: '))
print("    ")

# PROCEDIMIENTO
sigmar = 3.16
pi = 3.1416
fact = pi/180.0
phi0 = phi0*fact

dt = 0.005
```

```
tiempo = 0.0
for i in range(0,n+1,1):
   archivo.write(str(1) + '  ' + '\n');
   archivo.write(str('    ') + '  ' + '\n');
   phi = phi0 + w0*tiempo + 0.5*alpha*tiempo*tiempo
   x = radio*math.cos(phi)
   y = radio*math.sin(phi)
   x = x*sigmar
   y = y*sigmar
   z = 0.0
   tiempo = tiempo + dt
   archiva.write(str(x) + '   ' + str(y) + '\n');
   archivo.write("H" + '   ' + str(x) + '   ' + str(y) + '   ' +
   str(z) + '\n');

archivo.close;
archiva.close;
print("\nArchivo circular.xyz generado");
print("\nArchivo circular.dat generado");
```

Este procedimiento genera fotos de un objeto moviéndose en una trayectoria circular y las guarda en el archivo *circular.xyz*. Los datos graficados en la figura 2.3 se alojan en el archivo *circular.dat*.

Ejercicios

[1] Generar la foto y una animación del tiro parabólico de un objeto en la posición inicial x_0=0 y y_0=2, con velocidad inicial v_0=22 y un ángulo de inclinación $\theta=35^0$.

[2] Calcular la distancia máxima en x en el tiro parabólico con resistencia del aire, b=1.3, con posición inicial x_0=0, y_0=1, velocidad inicial v_0=65 y ángulo de inclinación $\theta=25^0$. Asímismo, generar una foto de la trayectoria y una animación del movimiento.

[3] Escribir un código en Python de tres objetos independientes en movimiento circular uniforme y generar una foto de las trayectorias y una animación del movimiento circular.

3. DERIVACIÓN NUMÉRICA

En esta sección se abordan ejercicios de derivación numérica. Con el propósito de recordar el concepto de derivada usamos la definición del límite de una función. [11,12] Es posible calcular la derivada de una función continua al escribir la función de interés, sumando el intervalo h a la variable x; posteriormente, se resta la misma función evaluada en la variable x, se divide la diferencia de funciones por h y se procede a sacar el límite cuando h tiende a cero. Para incrementar la precisión se necesita elegir el valor numérico de h cada vez más pequeño (mucho menor que la unidad). La noción de derivada es mostrada geométricamente en la figura 3.1. Por supuesto, este procedimiento no es práctico al querer derivar funciones en un problema más complejo. Otra opción para llevar a cabo la derivación numérica de una función continua es a través del uso de instrucciones/comandos ya definidos en la biblioteca *sympy*.

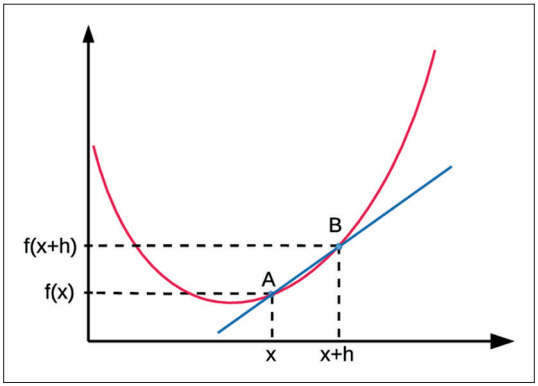
Derivada en términos del límite

La derivada de una función f(x) es posible calcularla a través del límite de la función:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3.1}$$

donde f(x) es una función continua en el dominio de interés de la cual existe su límite y h es un elemento que funciona como un incremento.

Figura 3.1. Sentido geométrico de la derivada



Mediante un ejemplo numérico mostramos cómo podemos calcular la derivada de una función usando la ecuación (3.1), de manera que con el código expuesto líneas abajo se calcula la derivada de la función $f(x) = x^2 + 6.0*x - 1.0/x$. Además, se evalúa en un valor particular de x, la precisión de la derivada es función de h.

```
#código 3.1
def f(x):
    fxi = x**2.0 + 6.0*x -1.0/x
    return (fxi)

print(‘   ‘)
print(‘Dame el valor de h’)
h = float(input(‘h :’))
print(‘ ‘)
print(‘Dame el valor de x’)
x = float(input(‘ x : ‘ ))
print(‘ ‘)
diferencia = (f(x+h) - f(x))
lim = diferencia/h
print(‘    ‘)
print(‘ límite y evaluación de x : ‘ + str(lim))
```

Ahora, identificamos la función f(x)=y(x). Se incluye el concepto de variación de la variable independiente h=Δx, y de la variación de la función y(x), como Δy. Así, la ecuación (3.1) se escribe como

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{y\,(x+\Delta x) - y\,(x)}{\Delta x} \qquad 3.2)$$

o bien

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \qquad 3.3)$$

De manera que podemos interpretar a la derivada como una razón de cambio con respecto a la variable independiente x. La primera derivada de una función $y(x) = x^n$, se escribe como:

$$\frac{dy(x)}{dx} = nx^{n-1} \qquad 3.4)$$

Esta operación también puede ser realizada de manera algebraica mediante un código en Python. Particularmente, la derivada de la función usada en el ejercicio anterior, $f(x) = x^2 + 6.0*x - 1.0/x$, es calculada a través del código 3.2, usando comandos incluidos en bibliotecas tales como *sympy,* que se renombra como *sp*. En particular se hace uso del comando *sp.diff(y,x)* y se evalúa la derivada en $x = x_{0.}$

```
#código 3.2
import sympy as sp
x = sp.Symbol('x')
y = x**2.0 + 6.0*x - 1.0/x
derivada = sp.diff(y,x)
values = {x: 2}        # x0=2
derivada.evalf(subs=values)
derivada2=sp.N(derivada, subs=values)

# RESULTADO DE LA DERIVADA NUMÉRICA
print('         ')
print('Función                 = ' + str(y))
print('Derivada evaluada en x0 = ' + str(derivada2))
print('siendo x0               = ' + str(values))
```

La instrucción que permite la derivación de una función *y(x)*, se escribe como: *derivada = sympy.diff(función a derivar, variable independiente)*, que al ejecutarse arroja como resultado la derivada de la función *y(x)*, la cual es una expresión algebraica.

$$\frac{dy(x)}{dx} = 2x + 6 + \frac{1}{x^2} \qquad 3.5)$$

esta misma expresión se evalúa en un valor particular de la variable independiente, digamos $x_0 = 2$, lo que arroja por resultado:

$$\left.\frac{dy(x)}{dx}\right|_{x_0=2} = \frac{17}{4} \qquad 3.6)$$

Derivada n-ésima

La segunda derivada de una función de una sola variable $y(x) = x^n$, se escribe como:

$$\frac{d^2y(x)}{dx^2} = n\,(n-1)\,x^{n-2} \tag{3.7}$$

es el resultado de derivar nuevamente la primera derivada de la función y(x) [11,12] y que es mostrada en la ecuación (3.4). En cinemática, a menudo la segunda derivada de una función está relacionada con una aceleración. Para ejemplificar este comentario se resuelve el siguiente ejercicio.

Si la función $y(t)=4t^2+3t-2$ representa la posición de un objeto al tiempo t, entonces la velocidad de movimiento del mismo objeto al tiempo t es la primera derivada de dicha función.

$$\frac{dy(t)}{dt} = (8t + 3)m/s \tag{3.8}$$

Así, la segunda derivada de la función representa la aceleración del mismo objeto.

$$\frac{d^2y(t)}{dt^2} = (8)m/s^2 \tag{3.9}$$

En particular, la aceleración es una constante.

Por otro lado, si el objetivo es calcular la derivada n-ésima de una función sin evaluar la función resultante, podemos hacer uso de la biblioteca *sympy* en Python. La segunda derivada de la función $f(x)=x^2+1$ es calculada con el procedimiento siguiente:

```
#código 3.3
import sympy

x, y = sympy.symbols('x y')
y = x**2 + 1
derivada = sympy.diff(y,x,2)

# RESULTADO

print('          ')
print('Funcion  = ' + str(y))
print('Derivada = ' + str(derivada))
```

La instrucción derivada = sympy.diff(y,x,2) nos permite obtener la segunda derivada de la función y cuya variable independiente es x.

Como caso específico, nos planteamos el calcular la tercera derivada de la función $f(x) = x^4 + 1$. Para ello, usamos también la biblioteca *sympy* de Python, indicando el orden en la derivada, tal y como se muestra en el código 3.4.

```
#código 3.4
import sympy

x, y = sympy.symbols('x y')
y = x**4 + 1
derivada = sympy.diff(y,x,3)

# RESULTADO DE LA DERIVADA
print('          ')
print('Funcion          = ' + str(y))
print('Derivada tercera = ' + str(derivada))
```

A continuación, en el código 3.5 se muestra la derivada de una función trigonométrica f(x) = cos(x).

```
#código 3.5
import sympy
x = sympy.Symbol('x')
derivada = sympy.diff(sympy.cos(x), x)
# RESULTADO DE LA DERIVADA
print('         ')
print('Funcion  = ' + str(sympy.cos(x)))
print('Derivada = ' + str(derivada))
```

Finalmente, incluimos un ejemplo donde se calcula la cuarta derivada de la función f(x) = cos(x) y que se muestra en el código 3.6.

```
#código 3.6
import sympy

x = sympy.Symbol('x')
derivada = sympy.diff(sympy.cos(x), x, 4)

# RESULTADO DE LA DERIVADA

print('         ')
print('función = ' + str(sympy.cos(x)))
print('Derivada cuarta = ' + str(derivada))
```

Ejercicios

[1] Calcular la quinta derivada de la función f(x)=2cos(x) + 1
[2] Calcular la sexta derivada de la función f(x) = $8x^5$ + 2x
[3] Calcular la derivada de la función f(x)= x^5 - $3.0x^2$ + x, y evaluar dicha derivada en el punto x=3

Desarrollo en serie de Taylor

La serie de Taylor [13] es un procedimiento que permite expresar una función f(x) de forma alternativa en términos de sus derivadas. De esta manera, la función f(x) debe ser infinitamente diferenciable en una vecindad de un número x_0. Cabe mencionar que este procedimiento también es válido en un espacio complejo.

$$f(x) = f(x_0) + \frac{f^1(x_0)}{1!}(x - x_0) + \frac{f^2(x_0)}{2!}(x - x_0)^2 + \frac{f^3(x_0)}{3!}(x - x_0)^3 + \ldots \quad 3.10)$$

De una forma compacta escribimos:

$$f(x) = \Sigma_{n=0}^{\infty} \frac{f^n(x_0)}{n!}(x - x_0)^n \quad 3.11)$$

$f^{(n)}(x)$ representa la derivada n-ésima de la función f(x). [14,15] Esta última ecuación es muy útil para escribir un código en Python y calcular un desarrollo en serie de una función, con la ayuda de herramientas tales como la derivación. Como ejemplo, se muestra el código 3.7 en Python para calcular el desarrollo en serie de la función f(x)=sen(x).

```
#código 3.7
import math
import sympy as sym
x=sym.Symbol('x')
fx=sym.sin(x)

x0 = 0
n = 10
k=0
polinomio = 0
while not (k>n):
  derivada = fx.diff(x,k)
  derivadax0 = derivada.subs(x,x0)
  denominador = math.factorial(k)
  terminok = (derivadax0/denominador)*(x-x0)**k
  polinomio = polinomio + terminok
  k = k + 1

print("    ")
print("P(x) = ", polinomio
```

Se ocuparon las bibliotecas *numpy* y *sympy,* y ambas se renombran como *np* y *sym*, respectivamente. Ahora bien, si se quiere mostrar las gráficas de la función original junto con el polinomio derivado del desarrollo en serie de Taylor de la misma función, es suficiente con correr el procedimiento en Python que se incluye líneas abajo. Necesitamos importar varias bibliotecas que nos permitan hacer el ajuste a la curva mediante un polinomio que resulta de aplicar el desarrollo en serie de Taylor, tal es el caso de *matplotlib.pyplot, sympy.utilities.lambdify* y de *sympy.plotting*.

Como segundo ejemplo, se calcula el desarrollo en serie de Taylor de la función f(x)=cos(x) alrededor del cero (0), y de grado 8 (n=8). En la figura 3.2 se muestra la función original, así como el polinomio resultante.

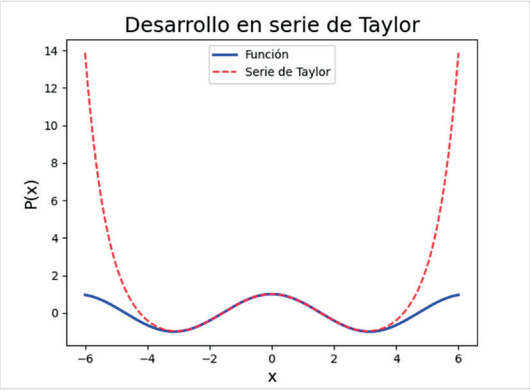
```
#codigo 3.8
from sympy import *
import math
import sympy as sym
import matplotlib.pyplot as plt
from sympy.plotting import *
from sympy.utilities.lambdify import lambdify

x=sym.Symbol('x')
print('La función de interés: ')
fx = eval(input())
print(r'Punto inicial x0: ')
x0 = float(input())

print('Grado del polinomio: ')
n = int(input())

k = 0
polinomio = 0
while not (k>n):
    derivada = fx.diff(x,k)
    derivadax0 = derivada.subs(x,x0)
    divisor = math.factorial(k)
    terminok = (derivadax0/divisor)*(x-x0)**k
    polinomio = polinomio + terminok
    k = k + 1

print("    ")
print("El polinómio resultante es")
print("P(x) = ", polinomio)
print("    ")

#grafica
f = lambdify(x, polinomio, 'numpy')
g = lambdify(x, fx, 'numpy')
t = np.linspace(-6.0, 6.0,1000)

# Se grafican las dos curvas
plt.plot(t,g(t), 'b-',linewidth=2,label="función")
plt.plot(t,f(t), 'r--',label='serie de Taylor')

#Se escriben letreros
plt.title("Serie de Taylor",fontsize=14)
plt.xlabel('x',fontsize=14)
plt.ylabel('P(x)',fontsize=14)

#Localización de un letrero
plt.legend(loc='center')

#Se salva la figura generada
plt.savefig('Serie_Taylor.png')
plt.show()
```

Se le pide al usuario introducir el grado del polinomio (n) que nos da información del orden de la derivada usada (n-1) —ver ecuación (3.1)—. No es obligatorio que el desarrollo en serie sea alrededor del cero. En este ejercicio usamos también la instrucción antes mencionada de la derivada de una función; así como la instrucción

divisor= np.math.factorial(k) para calcular el factorial de k. En general se construyen los términos de la serie de Taylor con la línea *terminok = (derivadax0/divisor)* (x-x0)**k*.

La figura 3.2 muestra la función original representada con una curva continua, y con una curva discontinua, el polinomio que resulta del ajuste obtenido mediante la serie de Taylor.

Figura 3.2.



Nota: Se muestran dos curvas: una de ellas corresponde a la función cos(x) y la otra al polinomio que resulta de desarrollar la función original en serie de Taylor alrededor del cero y hasta un grado n=8.

Ejercicios

[1] Calcular y graficar el polinomio resultante de desarrollar en serie de Taylor a la función f(x)=sen(x) alrededor del cero, con n=10. ¿En qué vecindad del dominio el polinomio ajustado concuerda con la función original?

[2] Calcular y graficar el polinomio resultante de desarrollar en serie de Taylor a la función $f(x)=\cos^2(x)$ alrededor del cero, con n=16. ¿En qué vecindad del dominio el polinomio ajustado concuerda con la función original?

4. INTEGRACIÓN NUMÉRICA

Existen diferentes métodos para calcular la integral numérica de funciones de una o más variables; así como también existen diversos softwares que son capaces de llevar a cabo dicha tarea; tal es el caso de MATHEMATICA [16], MATLAB [17], etc. Sin embargo, nuestro interés es mostrar al lector que no es requisito indispensable contar con algún software comercial para integrar de manera numérica una función. En esta oportunidad, vamos a describir de manera resumida el método del trapecio y el método de Simpson, los cuales son los más usados por la comunidad académica y que, a través de un procedimiento simple, es posible construir un código en lenguaje Fortran 77 y también en Python.

Método del trapecio

Más de una versión de dicho método podemos encontrar en la literatura. La aproximación inicial se le conoce como la regla del trapecio [11,12,14] simple, usa un polinomio interpolante de primer orden en la variable independiente, es una primera aproximación de las fórmulas cerradas de integración de Newton-Cotes, y es cerrada porque existen límites de integración. Dicho método es una variación del método Riemann, en donde se cubre el área bajo la curva (la función a integrar) con rectángulos (si es que usamos una función de una variable). En el método del trapecio se usan formas no regulares o más bien trapezoidales; en la figura 4.1 se muestran tales formas geométricas. El polinomio de primer orden usado para hallar la solución se escribe a continuación:

$$P(x) = f(a) + \frac{f(b)-f(a)}{b-a}(x-a) \tag{4.1}$$

De manera que la integral original se cambia por

$$\int f(x)dx \approx \int P(x)dx = \int\left[f(a) + \frac{f(b)-f(a)}{b-a}(x-a)\right]dx = (b-a)\left[\frac{f(b)+f(a)}{2}\right] \tag{4.2}$$

El error que se comete en este nivel de aproximación se calcula con la expresión siguiente:

$$Error = -\frac{(b-a)^3}{12} f^{(2)}(\xi) \qquad 4.3)$$

donde ξ es un número contenido en el intervalo [a,b], y $f^{(2)}(\xi)$ es la segunda derivada de la función de interés evaluada en dicha variable.

Por otra parte, una versión más precisa de este método se denomina regla del trapecio compuesta, donde consideramos n-intervalos cuya amplitud es h = (b-*a*)/n, de manera que

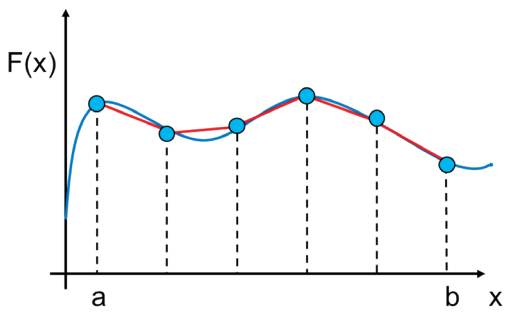
$$\int f(x)dx \approx \frac{h}{2}\left[f(a) + 2f(a+h) + 2f(a+2h) + \ldots + f(b)\right] \qquad 4.4)$$

En esta aproximación el error relacionado es

$$Error = -\frac{(b-a)^3}{12n^2} f^{(2)}(\xi) \qquad 4.5)$$

tomando en cuenta n-intervalos.

Figura 4.1. Regla del trapecio usando n-particiones



A continuación, abordaremos un ejercicio que permitirá ejemplificar el uso de este último método numérico. Antes, resolveremos una integral particular donde empleamos el teorema fundamental del cálculo integral [11], de manera que tengamos un resultado que sirva de referencia. Planteamos resolver la integral $\int_0^4 5xdx$, cuya solución es:

$$\int_0^4 5xdx = \left[5\frac{x^2}{2}\right]_0^4 = 40 \qquad 4.6)$$

Por otro lado, podemos resolver la integral de arriba usando la regla del trapecio compuesto, considerando n=4 particiones, entonces h=(4-0)/4=1.

$$\int_0^4 5xdx = 1\frac{1}{2}[5(0)+2[5(0+1*1)]+2[5(0+2*1)]+2[5(0+3*1)]+5(4)] = 40 \qquad 4.7)$$

Como se puede observar, encontramos el mismo resultado. Al integrar funciones más complejas se alcanza a observar la ventaja de este tipo de métodos.

A continuación, se incluye un código en Python que calcula la integral de la función f(x)=5x mediante el método del trapecio entre los límites inferior *a*=0 y superior b=4, usando n=4 particiones.

```
#código 4.1
# FUNCIÓN A INTEGRAR

def f(x):
    fxi = 5x
    return(fxi)

# LÍMITES
print("    ")
print("Dame los límites de integración")

a =  float(input('a: '))
b =  float(input('b: '))
n =  int(input('particiones: '))

# PROCEDIMIENTO

h = (b-a)/float(n)
suma = 0.0
x = 0.0

for i in range(0,n-1,1):
    x = x+h
    suma = suma + f(x)
integral = 0.5*h*(f(a) + 2.0*suma + f(b))

# RESULTADO DE LA INTEGRAL
print('         ')
print('Integral= ' + str(integral))
```

Método de Simpson

Otro de los métodos más usados para integrar numéricamente una función es el método de Simpson [11,12,14], una de las fórmulas cerradas de Newton-Cotes que emplea un polinomio de interpolación de segundo orden y pasa por tres puntos: $[a,f(a)]$, $[x_{int},f(x_{int})]$ y $[b,f(b)]$.

$$P_2(x) = f(a) + \frac{f(x_{int})-f(a)}{h}(x-a) + \frac{f(a)+f(b)-2f(x_{int})}{2h^2}(x-a)(x-x_{int}) \quad 4.8)$$

Donde a y b son los límites de integración, $x_{int}=(a+b)/2$ es el punto medio del intervalo que sirve como un tercer punto en la interpolación, y $h=(b-a)/2$ es el ancho en el intervalo que se usa como en el caso de la regla del trapecio. De manera que la integral original se aproxima como sigue:

$$\int_a^b f(x)dx \approx \int_a^b P_2(x)dx = \frac{h}{3}[f(a) + 4f(x_{int}) + f(b)] \quad 4.9)$$

cuyo error asociado se calcula como

$$Error = \frac{f^{(4)}(\xi)}{4!}(x-x_{int})^4 \quad 4.10)$$

donde $f^{(4)}$ denota la derivada cuarta de la función $f(\xi)$ donde ξ es una variable que se encuentra en 0 y x.

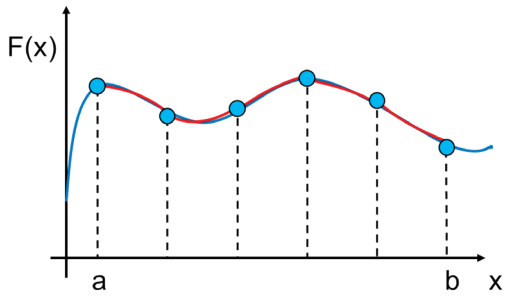
Método de Simpson compuesto

Una aproximación que mejora en precisión al último método mencionado es conocido como el método de Simpson compuesto. Este consiste en hacer subdivisiones de los intervalos, lo que imita el procedimiento llevado a cabo en la regla del trapecio. Ahora se consideran n-particiones $\{x_0, x_1, x_2, x_3, x_4, \ldots, x_n\}$, y se expresa el ancho de los n-intervalos como $h=(b-a)/n$, con la condición de que n debe ser un número par para que los límites queden contenidos en el método.

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(a) + 4\Sigma_{k=1,\text{impar}}^{n-1} f(x_k) + 2\Sigma_{k=2,\text{par}}^{n-2} f(x_k) + f(b)] \quad 4.11)$$

De modo que la primera sumatoria toma en cuenta términos que usan valores impares en el incremento y la segunda sumatoria los términos que involucran a la función evaluada en las particiones pares. En la figura 4.2 se muestra de manera geométrica la aproximación en la regla de Simpson.

Figura 4.2. Regla de Simpson usando n-particiones



Existen más aproximaciones que tienen una mejor precisión; sin embargo, la intención de este apartado es solo mostrar el método y, sobre todo, una forma de calcular integrales numéricas usando Python. Para ejemplificar el método, a continuación se muestra el código 4.2 en Python, que sirve para integrar la función $f(x)=\cos^2(x)$ entre los límites inferior *a* y superior *b*. Dicho código incluye la biblioteca *numpy*.

```
#código 4.2
import numpy as np

# FUNCIÓN A INTEGRAR
def f(x):
    fxi = np.cos(x)*np.cos(x)
    return(fxi)
# LÍMITES
print("    ")
print("método de Simpson ")
print("Dame los límites de integración
en grados")

a =  float(input('a: '))
b =  float(input('b: '))
n =  int(input('particiones: '))
PI= 3.141592
a =  (a*PI)/180.0
b =  (b*PI)/180.0
h = (b-a)/float(n)

print('límites en radianes')
print('a = ' + str(a))
print('b = ' + str(b))
print('h = ' + str(h))

pares = 0.0
impares = 0.0
for i in range(2,n+1,2):
    impares = impares + 4.0*f(a + float(i-1)*h)

for j in range(3,n,2):
    pares = pares + 2.0*f(a + float(j-1)*h)

integral = h*(f(a) + pares + impares + f(b))/3.0

# RESULTADO DE LA INTEGRAL
print('          ')
print('Número de particiones n = ' + str(n))
print('El valor de la integral es = ' + str(integral))
```

Un segundo ejemplo se muestra en el código 4.3 de Python que nos ayuda a integrar la función f(x)=1.0/(1+x) entre los límites inferior *a* y superior *b*. En esta ocasión no fue usada la biblioteca *numpy* debido a que no es una función trigonométrica la que se integra.

```
#código 4.3
# FUNCIÓN A INTEGRAR
def f(x):
      fxi = 1.0/(1.0 +x )
      return(fxi)

# LIMITES
print(“    “)
print(“Método de Simpson “)
print(“Dame los límites de integración”)
a =  float(input(‘a: ‘))
b =  float(input(‘b: ‘))
n =  int(input(‘particiones: ‘))

h = (b-a)/float(n)

impares = 0.0
for i in range(2,n+1,2):
      impares=impares+4.0*f(a+float(i-1)*h)

pares = 0.0
for j in range(3,n,2):
      pares = pares + 2.0*f(a + float(j-1)*h)

integral=h*(f(a) + pares + impares + f(b))/3.0

# RESULTADO DE LA INTEGRAL
print('            ')
print('Número de particiones n = ' + str(n))
print('El valor de la integral es = ' + str(integral))
```

Ejercicios

[1] Integrar la función $f(x)=x^4 + 2x +2/x^2$ en los límites de 1 a 4.
[2] Integrar la función $f(x)=3\tan(x) + 2$ en los límites de 0 a 30^0.
[3] Integrar la función $f(x)=\tanh(x) + 2x$ en los límites de -60^0 a 60^0.

5. MUESTREO ALEATORIO SIMPLE

El muestreo consiste en seleccionar un subconjunto de datos de un conjunto más grande, con la finalidad de llevar a cabo un análisis estadístico. Se pueden identificar dos tipos de muestreo: probabilístico y no probabilístico; el primero permite hacer inferencias y el segundo no.

En este apartado nos referimos al método del muestreo aleatorio simple, un método probabilístico que asigna, a todos los elementos que integran una población de interés, una etiqueta con la cual se identifican y, además, considera que estos elementos tienen la misma probabilidad de ser elegidos; son considerados eventos independientes sin importar el previo [18]. Se selecciona un elemento por evento de manera completamente aleatoria; de esta forma se garantiza que está presente la probabilidad de que todos los elementos de la población puedan ser elegidos en algún momento.

Con la intención de fijar ideas, podemos referirnos a una función cualquiera, mencionando que el muestreo nos permite sondear todos los valores que integran el dominio de una función de interés, eligiéndolos de manera aleatoria y analizando el co-dominio o imagen de la función. Existen más divisiones acerca de la técnica del muestreo; sin embargo, estamos interesados en mostrar solo aquel que usaremos al llegar al tema 10 de este libro.

Es importante revisar también el concepto de generador de números aleatorios antes de revisar ejemplos donde se usa el muestreo. Existen diversos procedimientos para generar números aleatorios [19], y todos ellos en algún momento dejan de ser efectivos; sin embargo, tienen un régimen de validez en el sentido de que en un rango de números son confiables. A continuación, se muestra un código en Fortran 77 que contiene un generador de números aleatorios, el cual fue tomado de la referencia [19]. El código consiste en líneas de programación que hace el llamado de la función que genera números aleatorios tantas veces como el usuario lo solicite, a través de la variable *NumPuntos*. Además, se hace uso de la declaración de las variables reales como doble precisión a través de la instrucción *implicit double precision (a-h,o-z)*, donde las variables implícitas, cuyos nombres comienzan con las letras comprendidas

entre la i y n, son consideradas como enteras. La razón de dicha declaración es para obtener una mayor precisión numérica; no obstante, es conveniente mencionar que el cálculo sugerido en el código 5.1 se puede lograr en precisión simple sin perder generalidad. Es necesario recordar al lector que en Fortran 77 se debe iniciar el código en la columna 7 y se debe procurar no rebasar la columna 70, debido a que este es el margen en el cual el compilador reconoce las instrucciones declaradas en el código.

```
#codigo 5.1
      program aleatorio
      implicit double precision (a-h,o-z)

      write(6,*)'numero de datos'
      read(5,*)NumPuntos

      do i = 1,NumPuntos
         write(6,*)i,ranf(dummy)
      enddo

      write(6,*)'          '
      write(6,*)'ya termine'
      write(6,*)'          '

      stop
      end
```

```
c  *  generador de números aleatorios entre 0 y 1 *
      real*8 function ranf ( dummy )
      implicit double precision (a-h,o-z)

      integer     l, c, m
      parameter ( l = 1029, c = 221591, m=1048576 )
      integer     seed
      real*8      dummy
      save        seed
      data        seed / 0 /

      seed = mod ( seed * l + c, m )
      ranf = dble ( seed ) / m

      return
      end
```

Algunos compiladores tienen en sus bibliotecas ciertos procedimientos definidos para generar números aleatorios, y es suficiente solo con invocar las bibliotecas y funciones involucradas.

Cálculo de π

Para ilustrar el método del muestreo es muy útil revisar el cálculo del número irracional π, un ejercicio simple y muy común a la hora de abordar este tema. Para estimar π se puede hacer el cociente del área del cuarto de círculo de radio *r* entre el área del cuadrado de lado *r*.

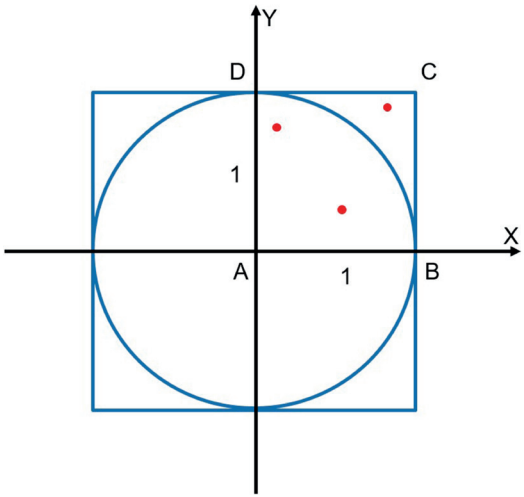
$$\frac{Área_{ABD}}{Área_{ABCD}} = \frac{\frac{1}{4}\pi r^2}{r^2} \tag{5.1}$$

En la figura 5.1 mostramos ambas figuras geométricas. El radio del círculo es r=1, y el lado del cuadrado se toma como r=1. Además, se considera el generar números aleatorios que llenen las áreas en común de ambas formas geométricas. Digamos que $N_{circulo}$ es el número de datos generados aleatoriamente que logren llenar el área del cuarto de círculo y $N_{cuadrado}$ es el número de datos generados aleatoriamente que cubren el área de cuadrado, que también se pueden identificar como el número de datos totales generados aleatoriamente. Entonces, en una primera aproximación, escribimos:

$$\frac{N_{circulo}}{N_{cuadrado}} = \frac{\pi}{4} \tag{5.2}$$

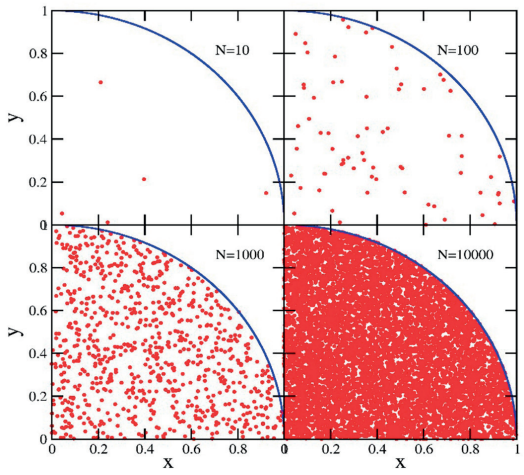
Se generan números aleatorios entre 0 y 1 para las coordenadas *x* y *y*. Ahora, si un punto con coordenadas (x,y) se ubica en el área común, entonces se considera como acierto, se contabiliza y se "pinta".

Figura 5.1. Cuadro formado por los puntos ABCD y cuarto de círculo formado por los puntos ABD



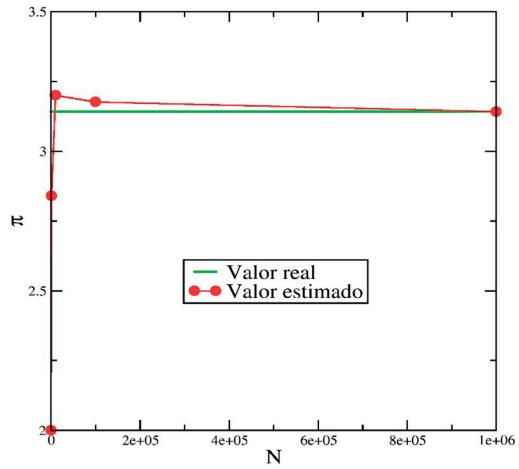
En la figura 5.2 se muestra la forma progresiva en que el área común del cuadrado y el del cuarto de círculo se va llenando de puntos generados aleatoriamente y, con ello, el valor numérico calculado de π es cada vez más preciso. Se observa que para alcanzar un grado de precisión aceptable es necesario tomar en cuenta un número cada vez más grande de datos numéricos del área en común (muestrear).

Figura 5.2. Aciertos aleatorios generados con 10, 100, 1 000 y 10 000 intentos



Con la ayuda del código en Fortran 77 que se incluye más adelante, podemos calcular el número π usando la ecuación (5.2). Si se compara el resultado numérico obtenido con el código en Fortran, y el valor ya conocido de π, vemos que, conforme el número de eventos al azar aumenta, ambos datos se van acercando cada vez más. En la figura 5.3 se observa que la convergencia del dato generado se logra con un número grande de eventos.

Figura 5.3. Comparación del valor de referencia del número π y los datos generados con el método del muestreo.Valores generados con 10, 100, 1E03, 1E05 y 1E06 intentos.



Enseguida mostramos un código en lenguaje Fortran 77 para calcular π, usando el método del muestreo donde se genera un archivo *CuartoDeCirculo.dat*, que contiene los datos generados aleatoriamente de X y Y.

```
#código 5.2
program pinum
      implicit double precision(a-h,o-z)
      real*8 dummy
      write(6,*)'    '
      write(6,*)'cuantos número aleatorios'
      read(5,*)NumPuntos
      write(6,*)'    '
      open(13, file='CuartoDeCirculo.dat')
      Ncirculo = 0
      do i = 1,NumPuntos
          x = ranf(dummy)
          y = ranf(dummy)
          cir = sqrt(x*x + y*y)
          if(cir.le.1) then
              Ncirculo = Ncirculo + 1
              write(13,23)x,y
          endif
      enddo
23    format(2f8.3)
      pi=4.0d0*float(Ncirculo)/float(NumPuntos)
       write(6,*)'   '
       write(6,*)'el valor de pi', pi
       write(6,*)'   '
       stop
       end
c     * números aleatorios entre 0 y 1 *
      real*8 function ranf ( dummy )
      implicit double precisión (a-h,o-z)
      integer     l, c, m
      parameter ( l = 1029, c = 221591, m =
1048576 )
      integer     seed
      real*8      dummy
      save         seed
      data         seed / 0 /
      seed = mod ( seed * l + c, m )
      ranf = dble ( seed ) / m
      return
      end
```

Se incluye también el código 5.3 en Python para calcular de manera numérica el número irracional π haciendo uso de la biblioteca *matplotlib* para generar un gráfico, y de la función *hypot(x,y),* contenida en la biblioteca *math,* que nos permite calcular la raíz cuadrada de la norma euclidiana, es decir $r = \sqrt{x^2 + y^2}$

```
#código 5.3
print(‘        ‘)
print(“Cuantos números aletorios?”)
NumPuntos =  int(input(‘NumPuntos: ‘))
Ncirculo = 0
for i in range(NumPuntos):
    x = random()
    y = random()
    if hypot(x, y) < 1:
        Ncirculo = Ncirculo + 1
        plt.plot(x,y,’ro’)
plt.savefig(“CuartoDeCirculo.png”)
plt.show()
print(‘        ‘)
print(‘pi =  ‘ + str(4.0 * Ncirculo / NumPuntos))
print(‘        ‘)
```

Es importante distinguir que en este último código no se incluye un procedimiento para generar números aleatorios, solo se invocan las bibliotecas *random* y se hace uso de la instrucción *random()*. Por último, se menciona que este mismo procedimiento genera una figura *CuartoDeCirculo.png*, que muestra los datos generados aleatoriamente y que están contenidos dentro del cuarto de círculo.

Integral numérica de una función f(x)

En esta sección mostramos el uso del método del muestreo en la integración de funciones. Cabe señalar que, integrar funciones de una variable independiente, es una tarea que puede ser realizada usando el método de Riemann, trapecio, Simpson, etc.; y que puede ser más barato en cuanto al tiempo de cómputo. Un ejemplo donde podemos observar una verdadera ventaja que ofrece el método del muestreo es la

integración de funciones de N-variables. En este apartado solo nos enfocamos en la solución numérica de integrales de funciones de una variable, de manera que podemos mostrar *Int* como la integral de una función f(x) y que escribimos:

$$Int = \int_a^b f(x)dx \tag{5.3}$$

En términos de una suma de Riemann, podemos escribir la integral como:

$$\int_a^b f(x)dx = \lim_{n\to\infty} \Sigma_{i=1}^{n} \Delta x \cdot f(x) \tag{5.4}$$

Donde

$$\Delta x = \frac{b-a}{n},\ x_i = a + \Delta \mathrm{x} \cdot i \tag{5.5}$$

siendo *i* el contador que identifica una partición en particular del total de particiones *n*. La idea es imitar la forma funcional de (5.4). Comenzamos con escribir la integral original, incluyendo una densidad de probabilidad ρ(x) que en principio es arbitraria.

$$Int = \int_a^b \left(\frac{f(x)}{\rho(x)}\right)\rho(x)dx \tag{5.6}$$

Ahora se considera un número importante de intentos t_0 que consisten en elegir un número aleatorio de la distribución ρ(x) en el rango (x_1,x_2).

$$Int = \left\langle \frac{f(t)}{\rho(t)} \right\rangle_{\text{eventos}} \tag{5.7}$$

Los paréntesis ⟨⋯⟩ representan un promedio sobre todos los intentos o eventos. Si se elige una distribución uniforme $\rho(x) = 1/(x_2-x_1)$ con $x_1<x<x_2$, entonces podemos escribir

$$Int' \approx \frac{(x_2-x_1)}{\tau_{\max}} \Sigma_{t=1}^{tmax} f(t) \tag{5.8}$$

Se identifican los límites x_2=b y x_1= *a*, e introducimos una variable que tome en cuenta la elección aleatoria de los datos que conforman el dominio, lo que nos permite reescribir la integral como:

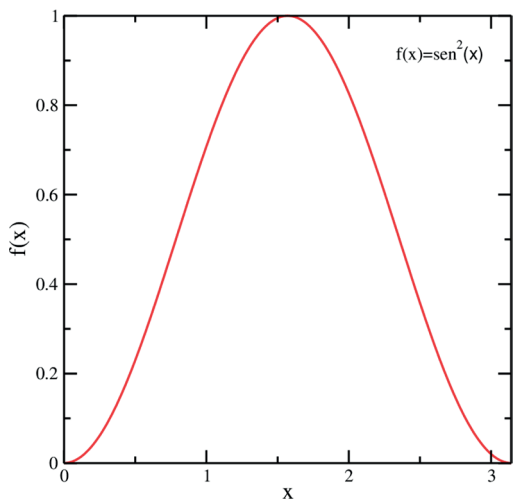
$$Int' = \frac{(b-a)}{m} \Sigma_{i=1}^{m} f(\chi_i) \qquad 5.9)$$

donde m es el número de datos aleatorios o intentos, χ_i es una variable aleatoria que depende del número aleatorio χ_i y recorre todo el dominio de la variable independiente.

$$\chi_i = a + (b-a)\xi_i \qquad 5.10)$$

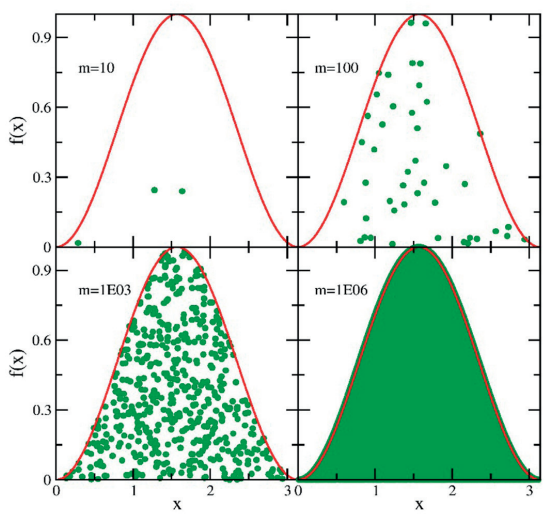
De manera que en el límite, cuando m → ∞, entonces se cumple que *Int´* → *Int,* lo cual es consistente con el teorema del límite central [20]. A continuación, se muestra un ejercicio a manera de ejemplo que consiste en integrar la función f(x)=sen^2(x) entre los límites a=0 y b=π (ver figura 5.4). En este ejercicio hacemos *m* intentos por llenar el área bajo la curva para poder estimar de mejor manera la integral de la misma función. Particularmente, usamos *m*=10, 100, 1000 y 1E06 intentos, tal que se obtiene una mejor precisión cada vez que se incrementa el número de datos generados aleatoriamente.

Figura 5.4. Curva de la función sen^2(x) de 0 a π



El número de intentos se refiere al número de datos aleatorios elegidos de aquellos que conforman el dominio de la función. En la figura 5.5 observamos este esquema de izquierda a derecha, donde al generar 10 datos aleatorios solo una pequeña fracción de ellos caen por debajo de la curva, por lo que es recomendable generar una gran cantidad de números aleatorios.

Figura 5.5. Área bajo la curva $sen^2(x)$ de 0 a π.
Siendo m el número de datos generados aleatoriamente



A fin de mostrar un camino para calcular la integral de la función f(x), usando el método del muestreo, incluimos un procedimiento en Python, que calcula la integral numérica y además genera el archivo *bajolacurva.dat,* el cual contiene los puntos que se encuentran bajo la curva y que fueron generados aleatoriamente. Así, también se genera una figura al ejecutar el ultimo código que contiene la misma información antes mencionada y que se guarda en el archivo *bajolacurva.png*.

```
#código 5.4
import math
import random
from matplotlib import pyplot as plt

archivo = open("bajolacurva.dat",'w')

# SE DEFINE LA FUNCIÓN
def f(x):
    return math.sin(x)*math.sin(x)

# LÍMITES DE INTEGRACIÓN
print(" ")
print("Dame los límites de integración")
xmin = float(input('a: '))
xmax = float(input('b: '))
NumPuntos=int(input('Números a generar: '))

Ndatos = 0
sumy = 0.0
for j in range(NumPuntos):
  x = xmin + (xmax - xmin) * random.
random()
  yrandom = random.random()
  sumy = sumy + f(x)
  funcion = f(x)
  if yrandom <= funcion:
    Ndatos = Ndatos + 1
    archivo.write(str(x) + ' ' + str(yrandom) +
'\n');
    plt.plot(x,yrandom,'ro')
plt.savefig("bajolacurva.png")
plt.show()
archivo.close;
print("\nArchivo creado bajolacurva.dat");
En = (xmax - xmin) * sumy / NumPuntos

# RESULTADO DE LA INTEGRAL
print(' ')
print('Integral= ' + str(En))
print('puntos bajo la curva=' + str(Ndatos))
print(' ')
```

El código en Fortran 77, mostrado a continuación, genera los datos de la función $f(x) = sen^2(x)$; es decir, evaluar la función entre los límites 0 a π. Los datos generados están contenidos en el archivo *lacurva.dat.*

```
#código 5.5
program pintar
      implicit double precision(a-h,o-z)
      dimension f(50000)
       open(12,file='lacurva.dat')
       rewind(12)
       pi = 3.14159
       x = 0.0
      dx = 0.01
      nstep = 50000
      do i = 1,nstep
          f(i) = sin(x)*sin(x)
          write(12,23)x,f(i)
          x = x + dx
          if(x.gt.pi) go to 100
      enddo
23    format(2F10.6)
100   continue
      close(12)
      stop
      end
```

En principio se evalúa la función f(x) *nstep*-veces, incrementando la variable independiente hasta que se cumple la condicionante x > p. Se incluye un formato particular a través de la instrucción *format(2F10.6)*, que significa que en el archivo *lacurva.dat* se escriben dos variables reales, que asigna 10 lugares a los enteros y 6 a las decimales. Es importante mencionar que este último código en Fortran 77, que genera la curva en el intervalo de 0 a π, delimita los puntos generados aleatoriamente (figura 5.5).

Por otro lado, se calcula nuevamente la integral numérica de la función $sen^2(x)$, pero ahora usando un código en lenguaje Fortran, en donde se incluyen bibliotecas que nos permiten generar números aleatorios a través de la función *ranf(dummy),* así

como declarar la función que deseamos integrar mediante el procedimiento *function f(x)*. En particular, usamos la declaración de todas las variables reales como de doble precisión por comodidad, pues aunque el utilizar doble precisión demanda más tiempo de cómputo, su uso es irrelevante, ya que es un procedimiento corto.

```
#código 5.6
      program integral
      implicit double precision (a-h,o-z)

      write(6,*)'numero de datos aleatorios'
      read(5,*)NumPuntos
      write(6,*)'  '
      pi   = 3.141592654
      xmax = pi
      xmin = 0.0
      sumf  = 0.0
      do i=1,NumPuntos
         Xi=xmin+ranf(dummy)*(xmax-xmin)
         sumf = sumf + f(Xi)
      enddo
      En=(xmax-xmin)*sumf/float(NumPuntos)
      write(*,*) 'La intergral es=',En
      write(6,*)'  '
      write(6,*)'  '
      stop
      end
```

```
C generador de números aleatorios entre 0 y 1

      real*8 function ranf(dummy)
      implicit double precision (a-h,o-z)
      integer     l, c, m
      parameter(l=1029, c=221591,
m=1048576)

      integer     seed
      real*8      dummy
      save        seed
      data        seed / 0 /
      seed = mod ( seed * l + c, m )
      ranf = dble ( seed ) / m
      return
      end

C     **  la función a integrar **
      real*8 function f(x)
      implicit double precision (a-h,o-z)
      f = sin(x)*sin(x)
      return
      end
```

De manera particular, en este mismo código en Fortran 77 se incluye un procedimiento sencillo que permite declarar la función a integrar, siendo únicamente necesario hacer el llamado de la función f(x) en el programa principal.

Integral para el cálculo de π

En el ejercicio siguiente se calcula el valor de π a través de la integración numérica de la función $f(x) = \sqrt{1 - x^2}$ entre los límites -1 y 1, lo que arroja como resultado el área de la mitad de una circunferencia de radio 1 y que es igual a π/2, lo que se escribe como:

$$\frac{\pi}{2} = \int_{-1}^{1} \sqrt{1 - x^2}\,dx \qquad 5.11)$$

Este ejercicio permite usar la información ya discutida acerca de la integración numérica usando el muestreo que se incluye en el código 5.7 en Python el cual se expone a continuación.

```
#código 5.7
import random
import math
from math import hypot
from  matplotlib import pyplot as plt

# SE DEFINE LA FUNCIÓN
def f(x):
    return math.sqrt(1-x**2)

# LÍMITES DE INTEGRACIÓN
xmin = -1
xmax =  1
NumPuntos= int(input('Números aleatorios: '))

print('límite inferior = ' + str(xmin))
print('límite superior =  ' + str(xmax))

sumf = 0.0
for j in range(NumPuntos):
    x=xmin+(xmax-xmin)*random.random()
    y=random.random()
    sumf = sumf + f(x)
    if hypot(x,y) < 1:
        plt.plot(x,y,'ro')
        plt.savefig("MitadDeCirculo.png")
pi = (xmax - xmin)*2*sumf/NumPuntos
plt.show()

# RESULTADO DE LA INTEGRAL
print('         ')
print('Estimación de pi = ' + str(pi))
print('         ')
```

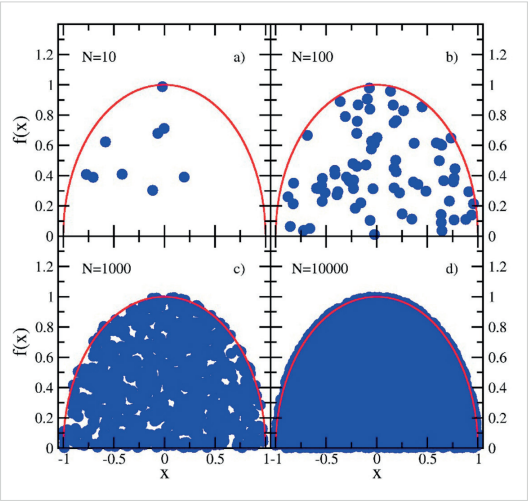
En este código se crea una figura que muestra los datos generados aleatoriamente, contenidos en la mitad de un círculo. Estos mismos datos se guardan en el archivo *MitadDeCirculo.png*.

El siguiente procedimiento permite evaluar la función $y = \sqrt{1 - x^2}$ en el intervalo [-1, 1].

```
#código 5.8
import matplotlib.pyplot as plt

archivo = open("funcionEvaluada.dat","w")
print("dame el numero de datos")
nstep = int(input())

dx=0.005
x=-1.0
for i in range(0,nstep,1):
   if(x<=1.005):
      y = ( 1.0 - x**2 )**(1.0/2.0)
      archivo.write(str(x) + ' ' + str(y) + '\n');
      x = x + dx
      plt.plot(x,y,'ro')
plt.show()
archivo.close()
```

La figura 5.6 muestra la generación de datos aleatorios para llenar el área bajo la curva, la cual corresponde a la mitad de un círculo de radio 1.

Figura 5.6. Generación de datos aleatorios para llenar el área bajo la curva



Nota. Se muestra la curva $f(x) = \sqrt{1 - x^2}$ con la línea continua en rojo, además, se incluyen los datos aleatorios que caen debajo de la curva de manera que llenan la misma. Dichos puntos generados aleatoriamente varian: a) N=10, b) N=100, c) N=1000, d) N=100000.

Ejercicios

Resuelva los siguientes ejercicios usando N=1E05, números aleatorios y que estén dentro del intervalo de integración.

[1] Calcular la integral de la función f(x)=3sin(x) entre los límites [0,π/3] radianes.
[2] Calcular la integral de la función f(x)=3x4 + 2x3 – 5x +4 entre los límites [1,4].
[3] Calcular la integral de la función f(x)=5coth(x) entre los límites [1,2] radianes.

6. RAÍCES DE UN POLINOMIO

Sea un polinomio P(x) de grado N**:**

$$P(x) = a_1x^N + a_2x^{N-1} + a_3x^{N-2} + a_4x^{N-3} + a_5x^{N-4} + a_6x^{N-5} + a_7x^{N-6} + \ldots \quad 6.1)$$

las raíces del polinomio [21-23] son los valores de x_i, que al sustituirlos en el polinomio da como resultado 0. La importancia de determinar raíces de un polinomio radica en que podemos extraer información relevante de dicho polinomio como: determinar máximos y mínimos, valores propios de matrices, solución de sistemas de ecuaciones lineales y diferenciales, etcétera.

Cabe señalar que hay al menos tres reglas que pueden ayudar a determinar las raíces de un polinomio; la primera menciona que el teorema de Bolzano establece que si una función continua, P(x), puede tomar valores de signo puesto en los extremos del intervalo [a,b] de interés, entonces existe al menos una raíz en dicho intervalo.

En la segunda se considera como el caso en que el polinomio P(x) es de grado N y cuenta con coeficientes reales, lo que garantiza que tendrá N raíces, ya sea que estas sean reales o complejas.

En la tercera se menciona que si el cociente p/q es una raíz racional de un polinomio que cuenta con coeficientes enteros

$$a_1x^N + a_2x^{N-1} + a_3x^{N-2} + a_4x^{N-3} + a_5x^{N-4} + \ldots a_N = 0 \quad 6.2)$$

entonces se tiene que q divide a los coeficientes a_1 y que p divide al término independiente a_N.

Factorización

Escribir un polinomio en términos de factores es uno de los procedimientos más simples para determinar raíces en un polinomio. Pongamos por caso el ejercicio siguiente.

Hallar las raíces del polinomio:

$$P(x) = x^2 + x - 12 \tag{6.3}$$

De manera que $a_1 = 1, a_2 = 2, a_3 = -12$. Si escribimos el polinomio en términos de dos factores tenemos:

$$x^2 + x - 12 = (x + 4)\ (x - 3) \tag{6.4}$$

Las raíces son aquellos números que hacen que cada uno de los factores por separado sean igual a cero, de manera que las raíces en este ejercicio son:

$$\begin{aligned} x_1 &= -4, \\ x_2 &= 3, \end{aligned} \tag{6.5}$$

Así que, al evaluar el polinomio en estos números, obtenemos que P(-4)=0 y P(3)=0; en este caso en particular las raíces son reales. Existen diferentes métodos para calcular las raíces de polinomios, pero la intención de este libro es acercar al lector con el intérprete Python para calcular raíces de un polinomio, usando algún método existente.

A continuación, se incluye el código 6.1 en Python, que calcula las raíces del polinomio descrito en la ecuación (6.1). Usamos las bibliotecas *numpy* de Python, lo que nos permite calcular las raíces reales de un polinomio. Primero se invoca al polinomio con la instrucción polinomio=[1.0,1.0,-12.0], y lo que se hace es incluir los coeficientes del polinomio. Posteriormente, se usa la instrucción *numpy.roots(polinomio),* que nos permite calcular las raíces reales del polinomio.

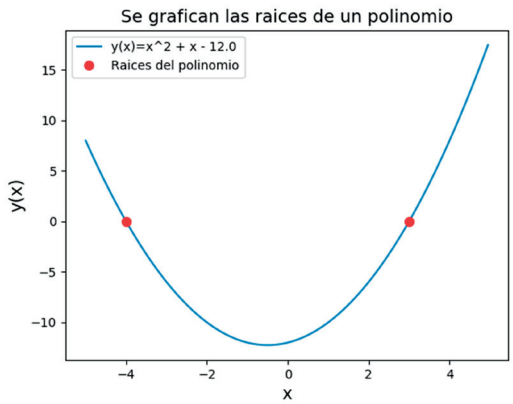
```
#código 6.1
import numpy
import sympy

#se escribe el polinomio
x, y = sympy.symbols('x y')
y = x**2 + x - 12.0
#El polinomio
polinomio = [1.0,1.0,-12.0]
#Se calculan las raíces
raiz = numpy.roots(polinomio)
print(" ")
#Se muestran las raíces
print("El polinomio es P(x) = ", str(y))
print("Las raíces del polinomio son:")
print("Raiz1 = %2.4f" % (raiz[0]))
print("Raiz2 = %2.4f" % (raiz[1]))
```

En este mismo código se imprime el polinomio de interés, así como sus raíces. Es importante señalar que este procedimiento no permite determinar raíces imaginarias. Además, si estamos interesados en graficar el polinomio junto con las raíces encontradas, el código 6.2 en Python se escribe como sigue:

```
#código 6.2
import numpy
import matplotlib.pyplot as plt
import sympy

#se escribe el polinomio
xx, yy = sympy.symbols('xx yy')
yy = xx**2 + xx - 12.0

polinomio = [1.0,1.0,-12.0]
# el polinomio es x**2 + x - 12.0

#la variable x toma valores de -5 a 5
x = numpy.arange(-5,5,.05)
y = numpy.polyval(polinomio,x)

#se calculan las raíces
raiz = numpy.roots(polinomio)
raices = numpy.polyval(polinomio,raiz)

#se muestran las raíces

print("El polinomio es P(x) = ", str(yy))
print("Las raíces del polinomio son:")
print("Raiz1 = %2.4f" % (raiz[0]))
print("Raiz2 = %2.4f" % (raiz[1]))

plt.figure
plt.plot(x,y,'-', label = 'y(x)=x**2 + x - 12.0')
plt.plot(raiz.real,raices.real, 'ro',
markersize='5',label = 'Raíces del polinomio')
plt.xlabel('x',fontsize=14)
plt.ylabel('y(x)',fontsize=14)
plt.title('Se grafican las raíces de un
polinomio',fontsize=14)
plt.legend()

plt.show()
```

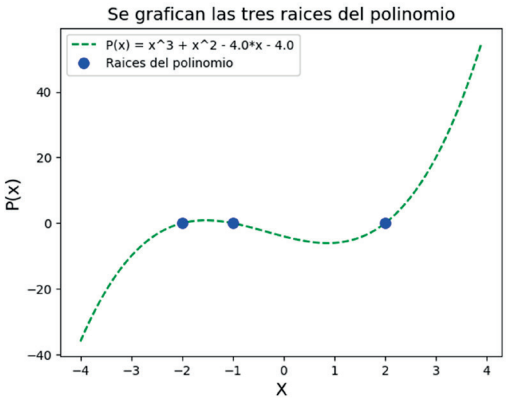
La gráfica del polinomio junto a las raíces calculadas se muestran en la figura siguiente:

Figura 6.1. Polinomio y sus raíces



En la figura 6.2 se muestran las tres raíces del polinomio $P(x) = x^3 + x^2 - 4x - 4$.

Figura 6.2. Polinomio y sus raíces



Ejercicios

[1] Calcular las raíces reales del polinomio f(x)=x^3-7x-6 mediante la factorización de sus términos y usar el procedimiento en Python mostrado.
[2] Determinar las raíces del polinomio f(x)=x^3+x^2-4x-4, usando el procedimiento en Python mostrado.

La fórmula de Bhaskara

Bhaskara Akaria (1114-1185) fue un matemático y astrónomo indio que dedujo la fórmula cuadrática para hallar raíces de polinomios de segundo grado [24]. A continuación, se presenta una deducción breve de dicha formulación.
Se escribe el polinomio de grado 2 de la forma

$$P(x) = ax^2 + bx + c \tag{6.6}$$

Esta misma ecuación se iguala a cero

$$ax^2 + bx + c = 0 \tag{6.7}$$

Posteriormente, se multiplica por $4a$ ambos lados de la igualdad

$$4a^2x^2 + 4abx + 4ac = 0 \qquad 6.8)$$

Se suma y se resta por el coeficiente b^2

$$4a^2x^2 + 4abx + 4ac + b^2 - b^2 = 0 \qquad 6.9)$$

Se reescriben los términos de la ecuación anterior

$$2^2a^2x^2 + 4abx + b^2 = b^2 - 4ac \qquad 6.10)$$

y se identifica un binomio cuadrado perfecto

$$(2ax + b)^2 = b^2 - 4ac \qquad 6.11)$$

Para el siguiente paso, se saca la raíz cuadrada a ambos lados de la igualdad

$$2ax + b = \pm\sqrt{b^2 - 4ac} \qquad 6.12)$$

Y finalmente se despeja x

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad 6.13)$$

De manera que si el discriminante

$$\Delta = b^2 - 4ac \qquad 6.14)$$

es positivo, se hallan raíces reales; en el caso de que sea negativo, las raíces serán imaginarias; y para el caso en que sea nulo, se obtendrá solo una solución.

Como ejemplo, resolvamos el siguiente ejercicio:

$$P(x) = 3.2x^2 + 2x + 1 \qquad 6.15)$$

Inmediatamente se identifican los coeficientes $a = 3.2$, $b = 2$, $c = 1$, con los cuales calculamos las raíces usando la ecuación (6.13).

Como siguiente paso, verificamos el signo que acompaña al discriminante

$$\Delta = 2^2 - 4(3.2)(1)$$
$$\Delta = -8.8 \qquad 6.16)$$

Observamos que es negativo, lo que nos informa que las raíces son imaginarias, de manera que podemos escribir la raíz cuadrada del discriminante como $\sqrt{\Delta} = i\sqrt{8.8}$. Así, podemos escribir las raíces del polinomio de interés como

$$x_1 = \frac{-2+i\sqrt{8.8}}{6.4}, \quad x_2 = \frac{-2-i\sqrt{8.8}}{6.4} \qquad 6.17)$$

A continuación, se incluye el código 6.3 en Python, que permite calcular las raíces del polinomio descrito en la ecuación (6.15), usando el método de Bhaskara.

```
#código 6.3

from math import sqrt
import sympy

x, y = sympy.symbols('x y')
y = 3.2*x**2 + 2*x + 1

print("  ")
print("Ecuaciones de segundo grado.")
a = float(input("dame el valor de a = "))
b = float(input("dame el valor de b = "))
c = float(input("dame el valor de c = "))

print(" ")
print("Discriminante (dis)")
dis = b**2 - 4*a*c
print("dis = b**2 - 4*a*c")
print("dis =",dis)
```

```
if (dis < 0):
    racu = sqrt(-dis)
    print("racu = sqrt(-dis)")
    print("racu = ",racu)
    term1 = -b/(2.0*a)
    term2 = racu/(2.0*a)
    print("El polinomio es P(x) = ", str(y))
    print("Las raíces son complejas")
    print('raiz1=%2.4f' % (term1), "+ i" '%2.4f' %
(term2))
    print('raiz2= %2.4f' % (term1), "- i" '%2.4f' %
(term2))
else:
    rc = sqrt(dis)
    print("racu = sqrt(dis)")
    print("racu = ",racu)
    print(" ")
    raiz1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
    raiz2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
    print("El polinomio es P(x) = ", str(y))
    print("Las raices son reales")
    print("raiz1 = %2.2f" % (raiz1))
    print("raiz2 = %2.2f" % (raiz2))
```

Ejercicios

[1] Determinar las raíces de los polinomios a) $P(x) = x^2 - 2x + 3$ y b) $P(x) = x^2 - 2x + 5$, y mencionar qué tipo de raíces son, si reales o imaginarias. Calcular las raíces de los polinomios algebraicamente y mediante un código en Python.

[2] Hallar las raíces del polinomio de segundo grado $P(x) = 3x^2 - 4x + 10$. Graficar el polinomio junto con sus raíces.

Método de Newton-Raphson

Este método muestra buena precisión, aunque hay casos en los que presenta algunas dificultades debido a que se pide que los polinomios sean diferenciales y continuos. El método está basado en un desarrollo en serie de Taylor del polinomio original [15], pero conservando los primeros términos. Esto introduce un error del orden de $(x\text{-}x_0)$.

Sea el polinomio P(x) = 0, el cual tiene al menos una solución x que pertenece al intervalo [a,b], luego al hacer un desarrollo en serie de Taylor alrededor de un punto x_0 suficientemente cercano a la raíz de interés

$$P(x) \approx P(x_0) + (x - x_0)\, P'(x_0) + \frac{(\xi - x_0)^2}{2} P''(x_0 + \theta h) \qquad 6.18)$$

con $0 < \theta < 1$, siendo $h = x\text{-}x_0$.

Como paso siguiente, suponemos que el polinomio P(x) es continuo en el intervalo [a,b] y, además, que x está muy cerca de x_0, de manera que la diferencia $x\text{-}x_0$ es muy pequeña. En el desarrollo en serie de Taylor de la ecuación anterior solo se conservan los primeros dos términos, de manera que nos queda la aproximación siguiente:

$$P(x) \approx P(x_0) + (x - x_0)\, P'(x_0) \qquad 6.19)$$

Ya que *x* son las raíces del polinomio y que se debe cumplir que P(x)=0, esto implica que se debe cumplir la relación siguiente:

$$P(x) = P(x_0) + (\xi - x_0)\, P'(x_0) \qquad 6.20)$$

Ahora despejamos *x*, que es la raíz buscada, de manera que después de hacer algunos pasos algebraicos obtenemos:

$$x = x_0 - \frac{P(x_0)}{P'(x_0)} \quad x = x_1 \qquad 6.21)$$

Así, la raíz buscada es:

$$\xi \approx x_1 \qquad 6.22)$$

Este es el procedimiento para determinar las raíces de un polinomio mediante el método de Newton-Raphson [25,26], donde debemos mencionar que *x* está más cerca del valor de la raíz en comparación del punto x_0. Esto hace que el procedimiento sea bueno; además, si uno está interesado en mejorar la precisión, basta con repetir el procedimiento nuevamente, tal que:

$$x_2 = x_1 - \frac{P(x_1)}{P'(x_1)} \tag{6.23}$$

Así sucesivamente hasta alcanzar una precisión satisfactoria

$$x_3 = x_2 - \frac{P(x_2)}{P'(x_2)} \tag{6.24}$$

La formulación más común para representar este método es la siguiente:

$$x_{k+1} = x_k - \frac{P(x_k)}{P'(x_k)} \tag{6.25}$$

Donde el método de Newton-Raphson requiere de una aproximación inicial x_k y, posteriormente, de forma recursiva se obtiene la sucesión:

$$\{x_k\}_{k=1}^{n} \tag{6.26}$$

A manera de ejemplo, resolver el siguiente ejercicio. Calcular las raíces del polinomio $P(x) = x^2 + x - 12$ mediante el método de Newton-Raphson elaborando un procedimiento contenido en el código 6.4 en Python.

```
#código 6.4
# El polinomio y su derivada

fx = lambda x: x**2 + x - 12
dfx = lambda x: 2*x - 1

#Tolerancia
tolerancia = float(input('tol: '))

# Declaramos el grado del polinomio
n=3

for i in range(0,n,1):
  print('dame el dato inicial')
  x0 = float(input('x_0: '))
  error = abs(2*tolerancia)
  x1=x0
  while (error>=tolerancia):
      xn1 = x1 - fx(x1)/dfx(x1)
      error = abs(xn1-x1)
      x1 = xn1

  print('A partir del dato inicial')
  print('la raíz es: ', '{:.2f}'.format(xn1))
```

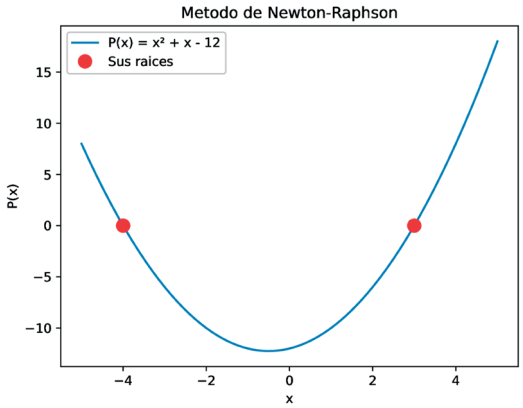
Este procedimiento calcula una raíz a la vez a partir del dato inicial que da el usuario. Esto hace que el procedimiento no sea inmediato, debido a que si no se tiene idea del intervalo [a,b] donde encuentran las raíces, entonces se tendrá que explorar varios valores para el dato inicial. En este código se usa el comando que permite calcular el valor absoluto *abs(x)*.

Para graficar el polinomio original junto con las raíces encontradas con el método de Newton-Raphson, se incluye el código 6.5 en Python.

```
#código 6.5
from sympy import symbols, sympify
import matplotlib.pyplot as plt

y,x = symbols('x y')
y = x**2 + x - 12.0
dy = y.diff(x)

print("    ")
x0 = input('valor inicial cercano a la raíz ')
raices=[]
while x0!='s':
  x0=sympify(x0)
  tolerancia=1E-05
  i=0
  error = abs(2*tolerancia)
  while(error >= tolerancia) :
    fx=y.subs(x,x0).evalf()
    dfx=dy.subs(x,x0).evalf()
    x1=x0-fx/dfx
    error = abs(x1-x0)
    x0=x1
    i=i+1
  raices.append(round(x0))
  print('la raiz es x= ' + str(round(x0)))
  print('se usaron ' + str(i), 'iteraciones')
  print("    ")
  print("elije una opción:")
  print("opción 1. para la siguiente raíz dame otro
valor inicial")
  print("opcion 2. para salir teclea.....[s]")
  x0=input("dame una opción: ")
print("  ")
print(f"las raíces son: {raíces}")
print("  ")
print("si quieres graficar presiona 1, en caso
contrario presiona cualquier tecla para salir ")
x0=input("  ")  #espera la opción, si es 1, se grafica
if(x0=="1"):

  #graficas
  N_int=5000
  xmin,xmax=-5,5
  rango = range(xmin*N_int,xmax*N_int)
  X = [(float(i)/N_int) for i in rango]
  def f(x):
      return x**2 + x - 12.0
  plt.plot(X,[f(i) for i in X], '-',label="P(x) = x² +
x - 12")
  plt.plot(raices,[f(i) for i in raices], 'ro',
markersize='10', label="Sus raices")
  plt.title("Metodo de Newton-Raphson")
  plt.xlabel("x")
  plt.ylabel("P(x)")
  plt.legend(loc="upper left")
  plt.savefig("Newton_Raphson.png")
  plt.show()
```

Al ejecutar este procedimiento se genera la figura 6.3, que muestra el polinomio de interés y sus raíces calculadas con el método de Newton-Raphson.

Figura 6.3. Gráfica del polinomio $P(x)=x^2+x-12$ y sus respectivas raíces, calculadas mediante el método de Newton-Raphson.



Ejercicios

[1] Calcular las raíces y generar un gráfico que incluya el polinomio, donde $f(x)=x^3-7x-6$. Usar el método de Newton-Raphson.

[2] Calcular las raíces y generar un gráfico que incluya el polinomio, donde $f(x)=x^2+2x+3$. Usar el método de Newton-Raphson.

Método de la secante

El método llamado de la secante [15] es un método iterativo que nos ayuda a encontrar también las raíces de un polinomio y es, de hecho, una variación del método de Newton-Raphson. La derivada involucrada en este último método se ve reemplazada por su expresión correspondiente en términos del límite de la diferencia de la misma función evaluada en distintos valores del dominio. Esta opción para determinar los ceros del polinomio surge debido a que en ocasiones, el tiempo de cómputo en calcular la derivada del polinomio de interés es muy alto, o es muy complicado determinar la derivada de la función de interés.

Retomemos la ecuación (6.12).

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

Nos indica que la raíz *k+1* se determina en términos de la raíz anterior, *k*. Ahora bien, si escribimos la derivada del polinomio en términos de límite hallamos la expresión siguiente:

$$P'(x_k) = \lim_{x_{k-1} \to x_k} \frac{P(x_k) - P(x_{k-1})}{x_k - x_{k-1}} \qquad 6.27)$$

Sustituyendo la ecuación (6.14) en (6.12), usando intervalos discretos de la variable independiente lo suficientemente pequeños, encontramos la expresión:

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{P(x_k) - P(x_{k-1})} P(x_k) \qquad 6.28)$$

A diferencia del método de Newton-Raphson, en el método de la secante se requiere de al menos dos datos para la variable independiente *x,* para poder determinar la pendiente asociada a la recta inicial que es secante al polinomio. Por definición, la recta secante toca en dos puntos a la curva de interés y cuando los puntos de *x* se juntan, dicha recta se convierte en una recta tangente a la curva, de esta manera se logra recuperar el método de Newton-Raphson.

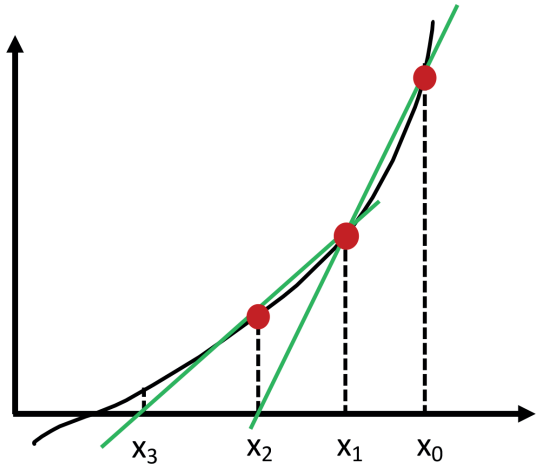
Se incluye un procedimiento en Python para calcular las raíces de un polinomio *P(x),* usando el método de la secante, donde se introducen al menos dos datos de la variable *x* para determinar la primera secante y comenzar así el proceso iterativo. En el código en Python se utilizan las librerías *sympy* y *matplotlib.pyplot*; la primera para manejar las variables *x* y *y* como símbolos para derivar la función de interés; y la segunda para graficar las curvas y las raíces.

```
# #código 6.6
from sympy import symbols
import matplotlib.pyplot as plt
x, y = symbols("x y")
f = x**2 + x - 12.0
print("El polinomio: \n" + str(f) + "\n")
tolerancia=1E-03
i = 0
raices= []
opción = "0"
while(opción != "2"):
  xk0 = float(input("Primer valor: "))
  xk1  = float(input("Segundo valor: "))
  error = abs(2.0*tolerancia)
  while(tolerancia <= error):
    fxk0 = f.subs(x,xk0).evalf()
    fxk1 = f.subs(x,xk1).evalf()
    xkf0 = xk1 - (xk0 - xk1)*fxk1/(fxk0-fxk1)
    error = abs(xkf0-xk1)
    xk0 = xk1
    xk1 = xkf0
    i = i + 1
  función = f.subs(x,xkf0).evalf()
  raices.append(round(xkf0,3))
  print("número de iteraciones: "+ str(i) + "\n")
  print ("La raíz x es:\n" + str(xkf0) + "\n")
  print("  ")
  print("elije una opción:")
 print("opción 1. para la siguiente raíz.....[1]")
  print("opción 2. para terminar, teclea.....[2]"
  opción = str(input("dame una opción: \n"))
print("Las raices del polinomio son " + str(raices) + "\n"))
```

```
print("opción 3. para graficar teclea.....[3]")
opción = str(input("dame una opción: \n "))
if(opción == "3"):
    N_int=5000
    xmin,xmax=-5,5
    rango = range(xmin*N_int,xmax*N_int)
    X = [(float(i)/N_int) for i in rango]
    def f(x):
        return x**2 + x - 12.0
    plt.plot(X,[f(i) for i in X], '-',label="P(x) = x**2 + x - 12")
    plt.plot(raices,[f(i) for i in raices], 'ro', markersize='10', label="Sus raices")
    plt.xlabel("x",fontsize = 14)
    plt.ylabel("P(x)",fontsize = 14)
    plt.title("Método de la secante",fontsize = 14)
    plt.legend(loc="upper left")
    plt.savefig("MetodoDeLaSecante.png")
    plt.show()
```

En la figura 6.4 se muestra la representación del método de la secante, donde se incluye el polinomio *P(x)* y dos líneas que son secantes al polinomio de interés. Se pintan puntos donde se indican los valores de *x* y su contraparte en la curva que resulta de graficar el polinomio *P(x)*.

Figura 6.4. Representación del método de la secante
Se incluye la curva del polinomio P(x) y dos rectas secantes



Ejercicios

[1] Calcular las raíces del polinomio $p(x)=x^3-7x-6$ usando el método de la secante.
[2] Calcular las raíces del polinomio $p(x)=x^2+2x+4$ mediante el método de la secante. Graficar la curva y sus raíces.

7. ECUACIONES DIFERENCIALES ORDINARIAS

Una ecuación diferencial ordinaria (EDO) relaciona la derivada de una función respecto de la única variable independiente con una función que relaciona a la misma variable y, en algunos casos, a la función en sí misma [27,28].
Existen diferentes métodos matemáticos para resolver ecuaciones diferenciales ordinarias de una variable de la forma

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

y que cuentan con una condición inicial $y(x_0)=y_0$. Enseguida se mostrarán al menos tres métodos para resolver este tipo de ecuaciones diferenciales, incluyendo códigos en Python basados en los métodos de Taylor, Euler y de Runge-Kutta.

Método de Taylor

Básicamente, el método de Taylor [28] consiste en aproximar la solución exacta de una ecuación diferencial ordinaria con un polinomio obtenido a través del desarrollo en serie de Taylor. Para ello se requiere que la función $y(x)$ sea diferenciable y que su k-ésima derivada exista alrededor del punto x_0. El polinomio resultante se escribe a continuación

$$y(x) = y(x_0) + \frac{1}{1!} y'(x_0,y_0)\,(x-x_0) + \frac{1}{2!} y''(x_0,y_0)\,(x-x_0)^2 + \frac{1}{3!} y'''(x_0,y_0)\,(x-x_0)^3 + \frac{1}{4!} y^{iv}(x_0,y_0)\,(x-x_0)^4 + \ldots + \frac{1}{k!} y^{(k)}(x_0,y_0)\,(x-x_0)^k \tag{7.2}$$

La solución es escrita, de manera que se puede hacer un procedimiento iterativo conociendo la solución inicial o anterior.

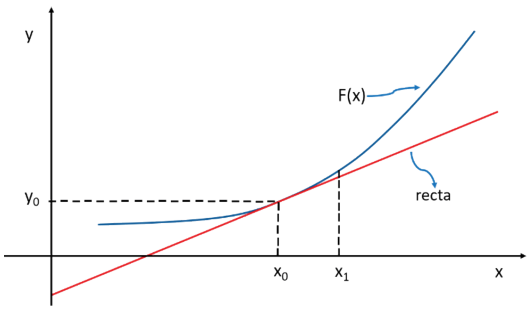
$$y_n = y_{n-1} + \frac{1}{1!} y'(x_{n-1},y_{n-1})\,(x_n - x_{n-1}) + \frac{1}{2!} y''(x_{n-1},y_{n-1})\,(x_n - x_{n-1})^2 + \frac{1}{3!} y'''(x_{n-1},y_{n-1})\,(x_n - x_{n-1})^3 + \frac{1}{4!} + y^{iv}(x_{n-1},y_{n-1})\,(x_n - x_{n-1})^4 + \ldots + \frac{1}{k!} y^{(k)}(x_{n-1},y_{n-1})\,(x_n - x_{n-1})^k \quad 7.3)$$

El error cometido en la aproximación se va reduciendo en tanto más términos en el desarrollo (7.2) se toman en cuenta. Cuando se trunca la aproximación hasta el término donde se involucra la primera derivada de la función *y(x)* ocurre algo que se conoce como el método de Euler.

Método de Euler

Este método debe su nombre al matemático y físico suizo Leonhard Euler [27,29]. Consiste en hacer una primera aproximación al método de Taylor. Se considera un cierto intervalo del dominio de la función, así como su imagen o, propiamente, la curva solución. Ahora ajustamos una recta que es tangente a la curva solución de la ecuación diferencial en un punto particular, dentro del intervalo en consideración.

Figura 7.1. Método de Euler



De manera que el método de Euler aproxima la función solución por medio de una línea poligonal. Una primera aproximación de la función, que es solución de la ecuación diferencial ordinaria, es escrita como:

$$y(x) = y_0 + f(x_0, y_0)\,(x - x_0) \quad 7.4)$$

considerando que la pendiente de la recta es tangente a la curva $m=y'(x_0)$, identificamos la pendiente de la recta como $m=f(x_0,y_0)$. Con base en lo anterior, podemos escribir la solución en una primera aproximación con el valor numérico en el eje de las abscisas x_1 como:

$$y(x_1) \approx y_1 = y_0 + f(x_0, y_0)\ (x_1 - x_0) \tag{7.5}$$

A partir de este dato calculado, enseguida se repite el procedimiento para estimar el siguiente punto en (x_2,y_2)

$$y(x_2) \approx y_2 = y_1 + f(x_1, y_1)\ (x_2 - x_1) \tag{7.6}$$

Para el siguiente paso se repite nuevamente el procedimiento, y así sucesivamente. Como parte de la metodología, se toman valores en el eje de las abscisas de forma equidistante, de manera que dichos valores son: $x_n = x_{n-1} + h$, o de la forma que $x_n = x_0 + nh$, siendo h el intervalo con el que varía "x". En general, la solución de la ecuación diferencial es escrita como:

$$y_n = y_{n-1} + f(x_{n-1}, y_{n-1})\ h \tag{7.7}$$

Con

$$x_n - x_{n-1} = h \tag{7.8}$$

Un ejercicio para ejemplificar el método de Euler es el siguiente. Resolver la ecuación diferencial ordinaria

$$\frac{dy}{dx} = \sqrt{x}\, y^{-2} \tag{7.9}$$

Con la condición inicial

$$y(1) = 2 \tag{7.10}$$

Resolviendo la ecuación diferencial $y^2dy = \sqrt{x}dx$ => $y^3 = 2x^{3/2} + 3C$, siendo C una constante a determinar. Si hacemos uso de la condición inicial, tenemos que:

$$y^3 = 2x^{3/2} + 3C \qquad 7.11)$$

Sustituyendo x_0 =1 y y_0 = 2

$$y^3 = 2x^{3/2} + 3C \qquad 7.12)$$

Despejando C

$$3C = 2^3 - 2(1)^{3/2} \qquad 7.13)$$

Finalmente, determinamos que C=2, de manera que la solución exacta es:

$$y = [2x^{3/2} + 6]^{1/3} \qquad 7.14)$$

Tabulando la solución exacta y la obtenida con el método de Euler, usando $h=0.1$, se observa que la precisión es muy buena, al menos en este ejercicio.

Tabla 7.1. Resultados numéricos de la solución de la ecuación diferencial

i	x_i	y_i(Euler)	y_i(exacta)
0	1	2.0	2.0
1	1.1	2.0250	2.0253
2	1.2	2.0508	2.0511
3	1.3	2.0768	2.0773
4	1.4	2.1033	2.1039
5	1.5	2.1300	2.1308
6	1.6	2.1570	2.1578
7	1.7	2.1842	2.1851

En el ejercicio en particular se identifica $f(x_0,y_0) = (\sqrt{x_0})\ (y_0^{-2})$, de manera que:

$$f(x_0,y_0) = (\sqrt{1})\ (2^{-2}) = 1/4 \qquad 7.15)$$

Partiendo de la ecuación $y_1 = y_0 + f(x_0,y_0)h$, determinamos y_1

$$y_1 = 2 + \left(\frac{1}{4}\right)(0.1) = 2.025 \qquad 7.16)$$

Ahora se determina el siguiente término en la solución $y_2 = y_1 + f(x_1,y_1)h$, así que identificamos:

$$f(x_1,y_1) = (\sqrt{x_1})\ (y_1^{-2}) \qquad 7.17)$$

Evaluando en x_1=1.1 y y_1=2.025:

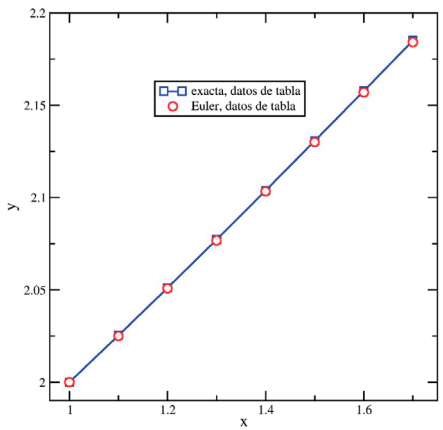
$$f(x_1,y_1) = (\sqrt{1.1})\ (2.025^{-2}) = 0.255768047 \qquad 7.18)$$

Calculamos y_2:

$$y_2 = 2.025 + (0.255768047)(0.1) = 2.05058 \qquad 7.19)$$

Y así sucesivamente. En la gráfica 7.2 se muestran las curvas obtenidas de manera exacta y con el método de Euler.

Figura 7.2. Comparación de las curvas obtenidas con la solución exacta y con el método de Euler



Nota: Los datos están contenidos en la tabla 7.1.

En la figura 7.2, la curva que incluye cuadros vacíos es el resultado de la solución de la ecuación diferencial, resuelta por medio de la separación variables. Los círculos vacíos representan los resultados obtenidos mediante el método de Euler con condiciones iniciales.

El código siguiente produce la solución numérica del ejercicio de arriba, usando el método de Euler $^{dy}/_{dx}=\sqrt{x}\ y^{-2}$.

```
#código 7.1
import sympy

archivo = open("eulerEjercicio1.dat",'w')
# ecuación
x, y = sympy.symbols('x y')
dy = x**(1.0/2.0)*(y**(-2.0))

print(' ')
print('Metodo de Euler ')
print('dy/dx    = ' + str(dy))
print(' ')
x0 =  float(input('x0: '))
y0 =  float(input('y0: '))
n =  int(input('particiones: '))

h = 0.1
x = x0
y = y0
print('  ')
for i in range(0,n+1,1):
    print ('X: ',i,x)
    print ('Y: ',i,y)
    archivo.write(str(x) + '  ' + str(y) + '\n');
    fxy  = x**(1.0/2.0)*(y**(-2.0))
    y = y + h*fxy
    x = x+h
archivo.close;
print("\nArchivo EulerEjercicio1.dat generado");
```

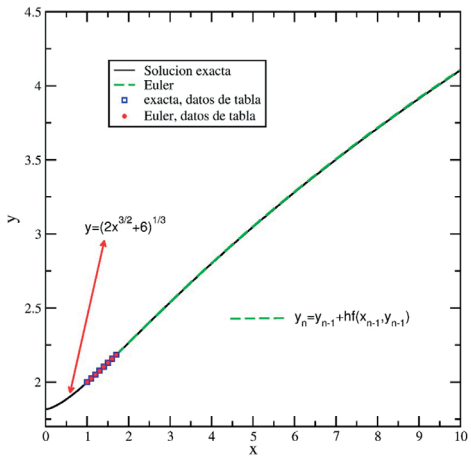
Usamos la biblioteca *sympy* en este ejercicio para declarar las variables X y Y como símbolos para llevar a cabo las operaciones algebraicas relacionadas con el método de Euler. Se genera el archivo *EulerEjercicio1.dat*, en donde se guardan los datos generados de la curva solución de la ecuación diferencial ordinaria.

La curva que corresponde a la solución exacta de la ecuación diferencial ordinaria del mismo ejercicio, la cual fue obtenida mediante el método de separación de variables, se obtiene usando el siguiente procedimiento:

```
#código 7.2
import matplotlib.pyplot as plt

archivo = open("funciónEvaluada.dat","w")
print("dame el número de datos")
nstep = int(input())
dx=0.1
x=1.0
for i in range(0,nstep,1):
   if(x<=10):
      y = ( 2*x**(3/2) + 6)**(1/3)
      archivo.write(str(x) + ' ' + str(y) + '\n');
      x = x + dx
      plt.plot(x,y,'ro')
plt.show()
archivo.close()
```

En la figura 7.3 se muestra la curva solución de la ecuación diferencial en el ejemplo anterior, pero ahora con un intervalo más amplio para la variable *x*.

Figura 7.3. Comparación de las curvas obtenidas con la solución exacta y con el método de Euler en un dominio más amplio



La curva que corresponde a la solución exacta, la cual fue obtenida mediante el método de separación de variables, se obtiene al evaluar la ecuación.

Ejercicios

[1] Resolver la ecuación diferencial $dy/dx = \sqrt{x}\, y^{-2}$ con la condición inicial y(1) =2, usando el método de Euler. Incluir todos los cálculos. Además, usando el código de Python, generar y graficar los datos de la solución obtenida con el método de Euler hasta x=20. Comparar con la curva que corresponde a la solución exacta.

[2] Resolver la ecuación diferencial $dy/dx = -y^{-2}$ con la condición inicial $y(0) = 1$, usando el método de Euler en el intervalo de [0 – 10] para la variable x. Graficar la solución y compararla con la curva de la solución exacta.

Método de Taylor de segundo orden

Se retoma la ecuación (7.2) y se conservan los términos correspondientes a la segunda derivada de la función *y(x)*; la expresión resultante se escribe como sigue:

$$y(x) = y(x_0) + y'(x_0,y_0)h + \frac{1}{2}y''(x_0,y_0)h^2 \qquad 7.20)$$

Se identifican las funciones *y'(x)* y *y"(x)*

$$f(x_0,y_0) = y'(x) \qquad 7.21)$$

$$f_2(x_0,y_0) = y''(x) \qquad 7.22)$$

Solo por comodidad identificamos $f(x_0,y_0) = f_1(x_0,y_0)$, de manera que la ecuación (7.20) se reduce a la ecuación siguiente

$$y(x) = y(x_0) + f_1(x_0,y_0)h + \frac{1}{2}f_2(x_0,y_0)h^2 \qquad 7.23)$$

o:

$$y(x) \approx y_1 = y_0 + f_1(x_0,y_0)h + \frac{1}{2} f_2(x_0,y_0)h^2 \qquad 7.24)$$

Falta por indicar la expresión de f_2, la cual se obtiene al usar la expresión que se muestra a continuación

$$f_2 = y''$$

Pero

$$y'' = \frac{df_1}{dx}$$

Así

$$f_2 = \frac{\partial f_1}{\partial x} + f_1 \frac{\partial f_1}{\partial y} \qquad 7.25)$$

Lo que nos permite estimar numéricamente la solución y_1. La solución en el siguiente paso tiene la expresión siguiente:

$$y_2 = y_1 + f_1(x_1,y_1)h + \frac{1}{2} f_2(x_1,y_1)h^2 \qquad 7.26)$$

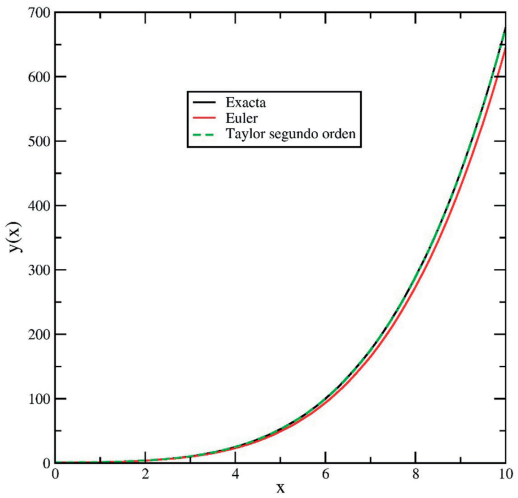
Se continúa con este procedimiento, sucesivamente, hasta el número de interacciones que se decida hacer.

A continuación, se incluye un código que permite calcular numéricamente la solución de la ecuación diferencial ordinaria (7.8) con condición inicial (7.9), usando el método de Taylor de segundo orden. El código en Python funciona solo para este ejercicio en particular.

```
#código 7.3
import sympy
archivo = open("Taylor2orden.dat",'w')

# ecuación
u, v = sympy.symbols('u v')
dv = u*(v**(1.0/2.0))
print('  ')
print('Método de Taylor segundo orden')
print('dv/du    = ' + str(dv))

x0 =  float(input('x0: '))
y0 =  float(input('y0: '))
n =  int(input('iteraciones: '))
h = 0.1
x = x0
y = y0

print('  ')
for i in range(0,n+1,1):
      archivo.write(str(x) + '  ' + str(y) + '\n');
      fxy1  = x*y**(1.0/2.0)
      dxf1 = y**(1.0/2.0)
      dyf1 = x/(2.0*y**(1.0/2.0))
      fxy2 = dxf1 + fxy1*dyf1
      y = y + h*fxy1 + 0.5*(h**2.0)*fxy2
      x = x + h

archivo.close;
print("\nArchivo Taylor segundo orden generado");
```

La curva solución se compara con las curvas que corresponden a la solución exacta y la obtenida con el método de Euler, con la intención de ver si hay diferencias entre ellas. En la figura 7.4 se muestran las curvas obtenidas, donde se observa que el método de Taylor de segundo orden mejora los resultados obtenidos con el método de Euler.

Figura 7.4. Curvas solución de la ecuación diferencial (7.9). Se incluyen los resultados de los métodos de Euler, Taylor de segundo orden y la solución exacta.



Método de Runge-Kutta

El método iterativo de Runge-Kutta [15,28,30,31,32] se usa para resolver ecuaciones diferenciales con condiciones iniciales. Sus autores son los matemáticos Carl David Tolmé Runge y Martin Wilhelm Kutta, quienes lo desarrollaron en 1900. Las ecuaciones diferenciales por resolver son del tipo:

$$y'(x) = f(x,y) \tag{7.27}$$

Con la condición inicial

$$y(x_0) = y_0 \tag{7.28}$$

El método consiste en aproximar la solución a la ecuación siguiente:

$$\int dy = \int f(x, y(x))dx \rightarrow y = y_0 + \int (x,y(x))dx \tag{7.29}$$

Usando la ecuación (7.29) se procede a resolver la ecuación diferencial original, de manera que resulta en un procedimiento iterativo

$$y_{n+1} = y_n + \int f(x, y(x))dx \qquad 7.30)$$

Luego, se aproxima la solución de la integral de la ecuación (7.29), usando el método de los trapecios:

$$y_{n+1} \approx y_n + \frac{1}{2} h(f(x_n, y_n) + f(x_{n+1}, y_{n+1})) \qquad 7.31)$$

Regularmente, el método se presenta usando las relaciones siguientes:

$$k_1 = hf(x_n, y_n) \quad \text{y} \quad k_2 = hf(x_{n+1}, y_n + k_1) \qquad 7.32)$$

De manera que

$$y_{n+1} = y_n + \frac{1}{2}(k_1 + k_2) \qquad 7.33)$$

es llamado el método de Runge-Kutta de segundo orden. El error cometido es del orden de h^2. Es importante mencionar que esta es solo una versión del método a segundo orden, ya que hay variantes de la expresión matemática de la solución; sin embargo, nuestro interés recae en comentar que hay diferentes niveles de aproximación en el método de Runge-Kutta.

Método de Runge-Kutta de tercer orden

La idea de hacer uso de un mayor orden es la de mejorar la precisión. Esto se consigue usando ahora el método de Simpson para resolver la integral involucrada en la ecuación, así:

$$\int f(x, y(x))dx \approx \frac{h/2}{3}\left(f(x_n, y_n) + 4f(x_{n+1/2}, y_{n+1/2}) + f(x_{n+1}, y_{n+1})\right) \qquad 7.34)$$

Vale la pena comentar que las funciones y_{n+1} y $y_{n+1/2}$ son solo estimaciones ya que no se conocen. Luego, para estimar $y_{n+1/2}$ se recurre al método de Euler:

$$y_{n+1/2} = y_n + \frac{h}{2}(x_n + y_n) \tag{7.35}$$

Ahora bien, para hallar y_{n+1} se recurre a la combinación de dos formas de obtener la misma función, una de ellas es el método de Euler directo

$$y_{n+1} = y_n + hf(x_n + y_n) \tag{7.36}$$

Y la segunda opción es encontrar una expresión para $y_{n+1,}$ usando el método de Euler con una variante. En vez de tomar la tangente en el punto inicial, se toma la pendiente de la recta en el punto medio.

$$y_{n+1} = y_n + hf(x_{n+1/2} + y_{n+1/2}) \tag{7.37}$$

Así, combinando las dos opciones mencionadas arriba, obtenemos:

$$y_{n+1} = y_n + h\left(2f(x_{n+1/2}, y_{n+1/2}) - f(x_n, y_n)\right) \tag{7.38}$$

Resumiendo, el método de Runge-Kutta de tercer orden

$$k_1 = hf(x_n,\ y_n) \tag{7.39}$$

$$k_2 = hf(x_n + \frac{h}{2},\ y_n + \frac{1}{2}k_2) \tag{7.40}$$

$$k_3 = hf(x_n + h,\ y_n - k_1 + 2k_2) \tag{7.41}$$

y la solución será determinada con la formulación siguiente:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 4k_2 + k_3) \tag{7.42}$$

El error asociado es proporcional a h^4.

Método de Runge-Kutta de cuarto orden

Para mejorar aún más la precisión se recurre a la aproximación de cuarto orden en el método de Runge-Kutta. Dicha aproximación también es conocida como el método Runge-Kutta clásico. Este se basa en el método de iteración de Simpson, introduciendo un nuevo paso intermedio al evaluar la derivada. La formulación es como sigue:

$$k_1 = hf(x_n, y_n) \tag{7.43}$$

$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1\right) \tag{7.44}$$

$$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \tag{7.45}$$

$$k_4 = hf(x_n + h, y_n + k_3) \tag{7.46}$$

La solución de la ecuación diferencial a este nivel de aproximación es:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{7.47}$$

El error asociado es del orden h^5.

Para ejemplificar el uso de la metodología, se procede a resolver la siguiente ecuación diferencial que cuenta con un valor inicial

$$\frac{dy}{dx} = \sqrt{x}\, y^2 \tag{7.48}$$

Con la condición inicial

$$y(1)=2 \tag{7.49}$$

Cuya solución exacta es

$$y = [2x^{3/2} + 6]^{1/3} \tag{7.50}$$

Por otro lado, si usamos el método de Runge-Kutta clásico para resolver la misma ecuación diferencial (7.48), con la misma condición inicial (7.49), primero identificamos $f(x,y) = \sqrt{x}y^{-2}$ y calculamos los coeficientes k_i. Para cada paso se calculan las constantes, se incrementa la variable x y se calcula la variable y.

Paso 1

$$k_1 = hf(x_0, y_0); \quad k_2 = hf\left(x_0 + \frac{h}{2}, y_0 + \frac{k_1}{2}\right) \tag{7.51}$$

$$k_3 = hf\left(x_0 + \frac{h}{2}, y_0 + \frac{k_2}{2}\right); \quad k_4 = hf(x_0 + h, y_0 + k_3) \tag{7.52}$$

Al sustituir los valores numéricos de las constantes k_i, encontramos que la solución en el paso 1 es:

$$y_1 = y_0 + \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4] \tag{7.53}$$

Paso 2

$$k_1 = hf(x_1, y_1); \quad k_2 = hf\left(x_1 + \frac{h}{2}, y_1 + \frac{k_1}{2}\right) \tag{7.54}$$

$$k_3 = hf\left(x_1 + \frac{h}{2}, y_1 + \frac{k_2}{2}\right); \quad k_4 = hf(x_1 + h, y_1 + k_3) \tag{7.55}$$

La solución en el paso 2 es:

$$y_2 = y_1 + \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4] \tag{7.56}$$

Paso 3

$$k_1 = hf(x_2, y_2); \quad k_2 = hf\left(x_2 + \frac{h}{2}, y_2 + \frac{k_1}{2}\right) \tag{7.57}$$

$$k_3 = hf\left(x_2 + \frac{h}{2}, y_2 + \frac{k_2}{2}\right); \quad k_4 = hf(x_2 + h, y_2 + k_3) \tag{7.58}$$

La solución en el paso 3 es $y_3 = y_2 + \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4]$, y así sucesivamente. En la tabla 7.2 concentramos los datos generados hasta una quinta iteración.

Tabla 7.2. Método de Runge-Kutta de cuarto orden, $h=0.1$, $f(x) = x^{1/2} y^{-2}$

i	x_i	k_1	k_2	k_3	k_4	$y_i=y(x_i)$
0	1	-	-	-	-	2.0
1	1.1	0.025	0.0253	0.025296	0.02557	2.0253
2	1.2	0.02557	0.025817	0.025814	0.02604	2.0511
3	1.3	0.02604	0.026241	0.02624	0.02676	2.0774
4	1.4	0.02642	0.026584	0.026582	0.0271	2.1040
5	1.5	0.02673	0.026858	0.02686	0.02697	2.1309

El código 7.4 en Python, mostrado a continuación, contiene el método de cuarto de Runge-Kutta.

```
#código 7.4
archivo  = open(“RungeKutta.dat”,’w’)

# FUNCIÓN
def f(x,y):
    return (x**(1/2)*y**(-2))
print(‘    ‘)

print(‘Runge-Kutta de cuarto orden’)

print(‘   ‘)
x0 =  float(input(‘x0: ‘))
y0 =  float(input(‘y0: ‘))
n  =  int(input(‘particiones: ‘))
h  =  float(input(‘h: ‘))
x11 = h*n + x0
print("x11",x11)
x = x0
y = y0
print('  ')
for i in range(0,n+1,1):
    archivo.write(str(x) + '    ' + str(y) + '\n');
    k1 = h*f(x, y);
    k2 = h*f(x + 0.5*h, y + 0.5*k1);
    k3 = h*f(x + 0.5*h, y + 0.5*k2);
    k4 = h*f(x + h, y + k3);

    y = y+(1.0/6.0)*(k1+2.0*k2+2.0*k3+k4);
    x = x+h;
archivo.close;
print("\nArchivo RungeKutta.dat generado");
```

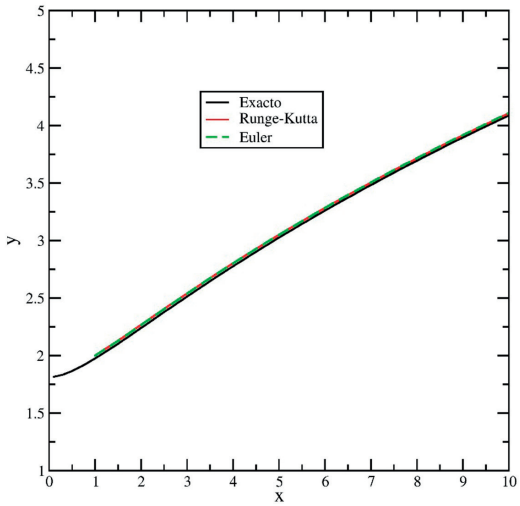
Adicionalmente, se escriben dos procedimientos para evaluar la función $f(x) = (2x^{(3/2)} + 6)^{(1/3)}$; uno de ellos en Fortran, código 7.5; y el segundo en Python, código 7.6.

```
#código 7.5
      program func
      implicit double precision (a-h,o-z)
      open(16,file='funcionEvaluada.dat')
      rewind (16)
      write(6,*)'número de datos'
      read(5,*)nstep
      dx = 0.1
      x  = 0.0
      do i =1,nstep
         if(x.eq.10)go to 200
         y = ( 2.0*x**(3.0/2.0) + 6.0 )**(1.0/3.0)
         x = x + dx
         write(16,*)x,y
      enddo
200   continue
      close(16)
      stop
      end
```

```
#código 7.6
import matplotlib.pyplot as plt

archivo = open("funcionEvaluada.dat","w")
print("dame el número de datos")
nstep = int(input())

dx=0.1
x=1.0
for i in range(0,nstep,1):
   if(x<=10):
      y = ( 2.0*x**(3.0/2.0) + 6.0 )**(1.0/3.0)
      archivo.write(str(x) + ' ' + str(y) + '\n');
      x = x + dx
      plt.plot(x,y,'ro')
plt.show()
archivo.close()
```

La gráfica 7.5 muestra la curva solución de la ecuación diferencial (7.48) obtenida mediante los métodos de Euler, Runge-Kutta de cuarto orden y la solución exacta, obtenida mediante la separación de variables.

Figura 7.5. Gráfica de la curva solución de la ecuación diferencial ordinaria del ejercicio 1



Nota. Junto con las curvas solución obtenidas con los métodos de Euler y de Runge-Kutta de cuarto orden.

Como se observa en la gráfica 7.5, las curvas obtenidas con los métodos de Euler y Runge-Kutta de cuarto orden coinciden entre sí y, a su vez, las dos coinciden con la curva solución de la ecuación diferencial, usando el método de separación de variables e integrando directamente. Este tipo de coincidencias no siempre ocurre al resolver ecuaciones diferencias ordinarias, usando los métodos ya mencionados.

Ejercicio 2. Se resuelve la ecuación diferencial con condiciones iniciales, como se muestra a continuación:

$$\frac{dy}{dx} = x + 2y \tag{7.59}$$

Con la condición inicial

$$y(0) = 1 \tag{7.60}$$

Se usa el método de Runge-Kutta clásico para hallar la solución. En la tabla 7.3 se muestran los resultados numéricos hasta cinco iteraciones.

Tabla 7.3. Método de Runge-Kutta de cuarto orden, h=0.1, f(x) = x+2y, f(0)=1.

i	x_i	k_1	k_2	k_3	k_4	$y_i=y(x_i)$
1	0.1	0.2	0.225	0.2275	0.2555	1.2267
2	0.2	0.2554	0.2859	0.2889	0.3231	1.5148
3	0.3	0.3230	0.3602	0.3640	0.4057	1.8776
4	0.4	0.4055	0.4511	0.4556	0.5066	2.3319
5	0.5	0.5064	0.5620	0.5676	0.6299	2.8978

Ejercicios

[1] Resolver la ecuación diferencial ordinaria $\frac{dy}{dx} = x + 2y$ con la condición inicial $y(0) = 1$ y graficar la curva solución.

[2] Resolver la ecuación diferencial ordinaria $^{dy}/_{dx} = -y^2$ con la condición inicial $y(0) = 1$, graficar la curva solución y comparar con la solución exacta y con la solución obtenida con el método de Euler.

Ecuaciones diferenciales de segundo orden homogéneas

Las ecuaciones diferenciales de segundo orden homogéneas con coeficientes constantes y condiciones iniciales pueden ser resueltas mediante una metodología bien establecida [15,27,28,33]. Existe más de un método teórico para dar solución a este tipo de ecuaciones diferenciales. Se muestran al menos tres casos que pueden presentarse en este tipo de ecuaciones. Se escribe una ecuación diferencial de segundo orden homogénea con coeficientes constantes a continuación:

$$a\frac{d^2y}{dx^2} + b\frac{dy}{dx} + cy = 0 \qquad 7.61)$$

o de la forma:

$$ay'' + by' + cy = 0 \qquad 7.62)$$

donde a, b y c son constantes, la ecuación asociada o auxiliar es:

$$ar^2 + br + c = 0 \qquad 7.63)$$

La cual se resuelve para hallar r_1 y r_2

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad 7.64)$$

Hay tres casos que se pueden presentar:

1. En caso de que $b - 4ac > 0$, entonces las raíces son reales y la solución general es:

$$y(x) = c_1 e^{r_1 x} + c_2 e^{r_2 x} \qquad 7.65)$$

De manera que c_1 y c_2 son constantes que se determinan al usar las condiciones iniciales impuestas para $y(x)$ y $y'(x)$.

2. En caso de que $b - 4ac = 0$, entonces solo se tiene una raíz y la solución general se escribe como sigue:

$$y(x) = c_1 e^{rx} + c_2 e^{rx} \qquad 7.66)$$

Las constantes se determinan c_1 y c_2 también usando las condiciones iniciales.

3. En caso de que $b - 4ac < 0$, tenemos raíces complejas, $r_1 = \alpha + i\beta$ y $r_2 = \alpha - i\beta$, y la solución general es

$$y(x) = c_1 e^{r_1 x} + c_2 e^{r_2 x} \qquad 7.67)$$

$$y(x) = c_1 e^{(\alpha + i\beta)x} + c_2 e^{(\alpha - i\beta)x} \qquad 7.68)$$

$$y(x) = c_1 e^{\alpha x} e^{i\beta x} + c_2 e^{\alpha x} e^{-i\beta x} \qquad 7.69)$$

$$y(x) = e^{\alpha x}[c_1 e^{i\beta x} + c_2 e^{-i\beta x}] \qquad 7.70)$$

$$y(x) = e^{\alpha x}[c_1[cos(\beta x) + isen(\beta x)] + c_2[cos(\beta x) - isen(\beta x)]] \quad 7.71)$$

$$y(x) = e^{\alpha x}[(c_1 + c_2]cos(\beta x) + i(c_1 + c_2)sen(\beta x)] \quad 7.72)$$

$$y(x) = e^{\alpha x}[d_1\, cos(\beta x) + d_2 sen(\beta x)] \quad 7.73)$$

Las constantes d_1 y d_2 se determinan usando las condiciones iniciales.
Por otro lado, se menciona que el método de Runge-Kutta de cuarto orden es útil para hallar la solución numérica de ecuaciones diferenciales de segundo orden con condiciones iniciales, de manera que se procede a calcular la solución numérica del mismo ejercicio. Escribimos la ecuación diferencial de la forma:

$$y''(x) = f(x, y, u) \quad 7.74)$$

o de forma equivalente:

$$\frac{d^2y}{dx^2} = f(x, y, u) \quad 7.75)$$

donde *u* es una función que surge al hacer un cambio de variable

$$u = \frac{dy}{dx} \quad 7.76)$$

Con las condiciones iniciales

$$y(x_0) = y_0 \quad 7.77)$$

Y

$$u_0 = \left(\frac{dy}{dx}\right)_{x=0} \quad 7.78)$$

La idea clave es manejar la ecuación diferencial original de segundo orden como un sistema de dos ecuaciones diferenciales de primer orden, tal como los ejercicios que se revisaron en la sección anterior. Esto es, el esquema mostrado en las ecuaciones (7.51) a (7.53) se aplica a cada una de las ecuaciones (7.75) y (7.76).

De manera que los coeficientes para la ecuación (7.76) son:

$$k_1 = f(x_n, y_{n,} u_n); \quad k_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2} q_1, u_n + \frac{h}{2} k_1\right) \tag{7.79}$$

$$k_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2} q_2, u_n + \frac{h}{2} k_2\right); k_4 = f\left(x_n + h, y_n + hq_3, u_n + \frac{h}{2} k_3\right) \tag{7.80}$$

$$u_{n+1} = u_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{7.81}$$

Y para la ecuación (7.75) son los siguientes:

$$q_1 = u_n; \quad q_1 = \left(u_n + \frac{h}{2}, k_1\right) \tag{7.82}$$

$$q_3 = \left(u_n + \frac{h}{2}, k_2\right); \quad q_4 = (u_n + hk_3) \tag{7.83}$$

La solución de la ecuación diferencial de segundo orden con condiciones iniciales tiene la siguiente expresión:

$$y_{n+1} = y_n + \frac{h}{6}(q_1 + 2q_2 + 2q_3 + q_4) \tag{7.84}$$

Se resuelve un ejercicio a manera de ejemplo donde se determina la solución mediante dos métodos. Escribimos la ecuación diferencial de segundo orden:

$$y'' - 2y' + y = 0 \tag{7.85}$$

Con condiciones iniciales $y(0) = 1$ y $y'(0) = 2$:

$$r = \frac{2 \pm \sqrt{4-4}}{2} = 1 \tag{7.86}$$

La solución general es:

$$y(x) = c_1 e^x + c_2 x e^x \tag{7.87}$$

Ahora se determinan las constantes c_1 y c_2; para ello se deriva la ecuación (7.87):

$$y'(x) = c_1 e^x + c_2 e^x + c_2 x e^x \quad 7.88)$$

Se introducen las condiciones iniciales:

$$y(0) = c_1 \quad 7.89)$$

$$1 = c_1 \quad 7.90)$$

Y

$$y'(0) = c_1 + c_2 \quad 7.91)$$

$$2 = c_1 + c_2 \quad 7.92)$$

$$1 = c_2 \quad 7.93)$$

Así que la solución final es:

$$y(x) = e^x + x e^x \quad 7.94)$$

A continuación, se usa el código 7.7 en Python para evaluar y graficar la función solución (7.94).

```
#código 7.7
import math
import matplotlib.pyplot as plt
archivo  = open("funcionEvaluada.dat",'w')
y  = 1.0
x  = 0.0
dx = 0.1
print('         ')
n  =  int(input('datos: '))
for i in range(0,n,1):
    archivo.write(str(x) + ' ' + str(y) + '\n');
    plt.plot(x,y,'ro')
    print('x,y',x,y)
    x = x + dx;
    y = math.exp(x) + x*math.exp(x);
plt.show()
archivo.close;
print("\nArchivo funcionEvaluada.dat generado");
```

Y genera el archivo *funcionEvaluada.dat*, el cual contiene los datos de la curva solución. Además, se resuelve la misma ecuación diferencial de segundo orden con condiciones iniciales usando el cuarto orden del método de Runge-Kutta.

$$y'' = 2u - y \tag{7.95}$$

Con las condiciones iniciales siguientes: $x_0 = 0$, $y_0 = 1$ y $u_0 = 2$.

Se identifica la función $f(x, y, u)$:

$$f(x, y, u) = 2u - y \tag{7.96}$$

Enseguida, se procede a calcular las constantes.

Paso 1:

$$k_1 = f(x_0, y_0, u_0); \qquad q_1 = u_0 \tag{7.97}$$

$$k_2 = f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}, q_1, u_0 + \frac{h}{2} k_1\right); \quad q_2 = \left(u_0 + \frac{h}{2} k_1\right) \tag{7.98}$$

$$k_3 = f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}, q_2, u_0 + \frac{h}{2} k_2\right); \quad q_3 = \left(u_0 + \frac{h}{2} k_2\right) \tag{7.99}$$

$$k_4 = f\left(x_0 + h, y_0 + hq_3, u_0 + \frac{h}{2} k_3\right); \qquad q_4 = \left(u_0 + hk_3\right) \tag{7.100}$$

Como siguiente paso, se introducen las constantes calculadas k_i y q_i en las ecuaciones (7.81) y (7.84), respectivamente. Se obtiene la expresión para u_1

$$u_1 = u_0 + \frac{h}{6}(q_1 + 2q_2 + 2q_3 + q_4) \tag{7.101}$$

Y la solución para el primer paso, y_1, tiene la siguiente expresión:

$$y_1 = y_0 + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{7.102}$$

Y así sucesivamente, hasta el paso que se desee estimar.

En las tablas 7.4 y 7.5 se incluyen los datos obtenidos de la solución de la ecuación diferencial de segundo orden en la aproximación de cuatro interacciones.

Tabla 7.4. Constantes k_i del método de Runge-Kutta.

i	x_i	k_1	k_2	k_3	k_4	$y_i=y(x_i)$
1	0.1	3.0	3.1999	3.2125	3.4265	1.2157
2	0.2	3.4260	3.6526	3.6667	3.9090	1.4657
3	0.3	3.9085	4.1650	4.1809	4.4551	1.7548
4	0.4	4.4545	4.7447	4.7626	5.0729	2.0885

Tabla 7.5. Constantes q_i del método de Runge-Kutta.

i	x_i	q_1	q_2	q_3	q_4	u_i=dy/dx
1	0.1	2.0	2.15	2.16	2.3212	2.3209
2	0.2	2.3209	2.4922	2.5035	2.6875	2.6871
3	0.3	2.6871	2.8825	2.8953	3.1052	3.1047
4	0.4	3.1047	3.3274	3.3419	3.5809	3.5804

Se incluye el código 7.8 en Python, que calcula la solución numérica de la ecuación diferencial de segundo orden mediante el método de Runge-Kutta de cuarto orden.

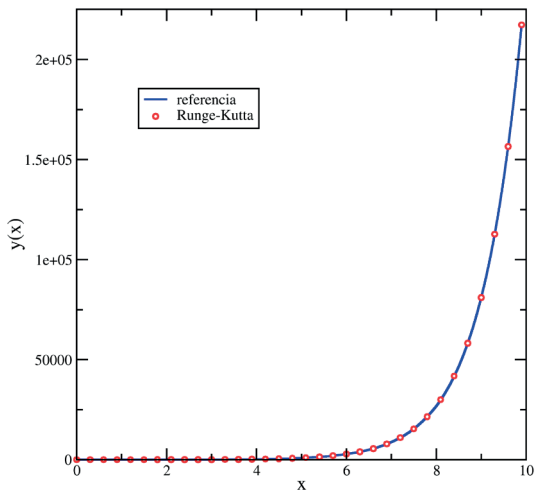
```
#código 7.8
import matplotlib.pyplot as plt
archivo = open("RungeKutta.dat","w")
# FUNCION
def f(x,y,u):
    return ( 2*u - y )
print('Metodo de Runge-Kutta de cuarto
orden')
x0 =  float(input('x0: '))
y0 =  float(input('y0: '))
u0 =  float(input('u0: '))
n  =  int(input('particiones: '))
h  =  float(input('h: '))

x11 = h*n + x0
print("x11",x11)
x = x0
y = y0
u = u0
```

```
for i in range(0,n+1,1):
    archivo.write(str(x) + ' ' + str(y) + '\n');
    plt.plot(x,y,'ro')
    print('x,y',x,y)
    k1 = f(x, y, u);
    m1 = u
    k2 = f(x + 0.5*h, y + 0.5*h*m1, u + 0.5*h*k1);
    m2 = u + 0.5*h*k1
    k3 = f(x + 0.5*h, y + 0.5*h*m2, u + 0.5*h*k2);
    m3 = u + 0.5*h*k2
    k4 = f(x + h, y + h*m3, u + h*k3);
    m4 = u + h*k3

    y = y + (h/6.0)*(m1 + 2.0*m2 + 2.0*m3 + m4);
    u = u + (h/6.0)*(k1 + 2.0*k2 + 2.0*k3 + k4);
    x = x+h;
plt.show()
archivo.close;
print("\nArchivo RungeKutta.dat generado");
```

Se usa la biblioteca *matplotlib* para generar una figura que muestra los puntos de la curva solución. Dichos puntos se guardan en el archivo *Runge-Kutta.dat*. En la figura (7.6) se muestra la curva solución obtenida con un método teórico, a la cual llamamos curva de referencia, y se incluyen círculos vacíos que representan la curva solución

obtenida con el método de Runge-Kutta de cuarto orden. Ambas curvas coinciden de muy buena manera, lo que indica que el método numérico usado da buenos resultados.

Figura 7.6. Curva solución obtenida con un método teórico



Nota: Gráfica de la curva soluciónde la ecuación diferencial de segundo orden obtenida con el método de Runge-Kutta de cuarto orden.

En un segundo ejemplo, se resuelve la ecuación diferencial del oscilador armónico libre; primero, empleando la metodología descrita en la ecuación (7.73); y segundo, usando el método de Runge-Kutta de cuarto orden. Se escribe la ecuación diferencial de segundo orden:

$$m\frac{d^2x}{dt^2} + kx = 0 \qquad 7.103)$$

donde x es el desplazamiento, t el tiempo, m la masa y k la constante del resorte. Al dividir por m ambos lados de la igualdad, escribimos:

$$\frac{d^2x}{dt^2} + \omega^2 x = 0 \qquad 7.104)$$

siendo la frecuencia angular $\omega = \sqrt{\frac{k}{m}}$. Las condiciones iniciales son $m = 4kg$, $k = 16N/m$, $t_0 = 0s$, $x(t_0) = 1m$, $y'(t_0) = 2m/s$, y se determina $\omega = \sqrt{\frac{16}{4}}$ así $\omega = 2rad/s$. La ecuación auxiliar asociada a la ecuación diferencial es

$$r^2 + \omega^2 = 0 \qquad 7.105)$$

Cuyas raíces son imaginarias

$$r_1 = +i\omega \qquad 7.106)$$

$$r_2 = -i\omega \qquad 7.107)$$

De manera que la solución de la ecuación diferencial es de la forma

$$x(t) = e^{\alpha t}[d_1 cos(\beta t) + d_2 sen(\beta t)] \qquad 7.108)$$

En particular, se identifica $\alpha = 0$ y $\beta = \omega$. La ecuación anterior se reduce a la expresión siguiente:

$$x(t) = d_1 cos(\omega t) + d_2 sen(\omega t)] \qquad 7.109)$$

Si se elige $d_1 = Asen(\varphi)$ y $d_2 = Acos(\varphi)$, donde A es la amplitud y φ es la fase. Usando identidades trigonométricas se puede escribir la ecuación anterior de la siguiente forma:

$$x(t) = Asen(\omega t + \varphi) \qquad 7.110)$$

Por otro lado, se introducen las condiciones iniciales en la ecuación 7.109 para determinar d_1 y d_2

$$x(t_0) = 1 = d_1 \qquad 7.111)$$

La primera derivada de la ecuación (7.109) es

$$x'(t) = -d_1\omega\text{sen}(\omega t) + d_2\omega\cos(\omega t) \qquad 7.112)$$

Luego, se introduce la condición inicial $x'(t_0) = 2$

$$x'(t_0) = 2 = 2d_2 \qquad 7.113)$$

De esta manera, se determina que las constantes d_1 y d_2 son:

$$d_1 = 1 \quad \text{y} \quad d_2 = 1 \qquad 7.114)$$

así que al elevar al cuadrado d_1 y $d_{2,}$ tenemos la expresión siguiente:

$$d_1^2 = A^2 sen^2(\varphi) \qquad 7.115)$$

$$d_2^2 = A^2 cos^2(\varphi) \qquad 7.116)$$

Al sumar tenemos:

$$d_1^2 + d_2^2 = A^2 sen^2(\varphi) + A^2 cos^2(\varphi) \qquad 7.117)$$

Se sustituyen los valores 7.114

$$1^2 + 1^2 = A^2[cos^2(\varphi) + sen^2(\varphi)] \qquad 7.118)$$

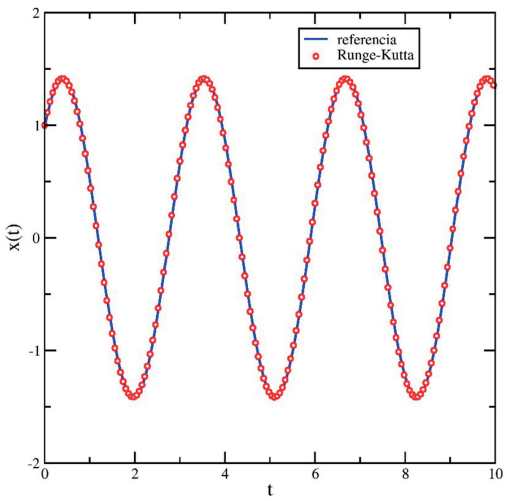
Se calcula la amplitud $A = \sqrt{1^2 + 1^2} = \sqrt{2}$ y la fase $\varphi = arctang(4) = 0.7854$.Finalmente, se escribe la solución en términos de la amplitud y de la fase

$$x(t) = \sqrt{2}\text{sen}(2t + 0.7854) \qquad 7.119)$$

En la gráfica 7.7 se muestra la curva solución de la ecuación diferencial de segundo orden (7.103), obtenida mediante un procedimiento algebraico que arroja como resultado la ecuación (7.119), la cual es identificada como la solución de referencia.

También se incluye, en la misma figura, la curva solución obtenida con el método de Runge-Kutta de cuarto orden.

Figura 7.7. Gráfica de la curva solución de la ecuación diferencial de segundo orden 7.103



Nota: La curva se obtiene con el método de Runge-Kutta de cuarto orden. También se incluye la solución de referencia.

Ejercicios

Resolver la ecuación diferencial de segundo orden con condiciones iniciales, de manera algebraica y numérica, usando el método de Runge-Kutta de cuarto orden, hasta un valor de x en el intervalo de 0 a 10.

$\frac{d^2y}{dx^2} - 2\frac{dy}{dx} + 2y = 0$, con condiciones iniciales $y(0) = 1$ y $y'(0) = 2$.

$\frac{d^2y}{dx^2} - \frac{dy}{dx} + 2y = 0$ con condiciones iniciales $y(0) = 1$ y $y'(1) = 3$.

Ecuaciones diferenciales de segundo orden no-homogéneas

Para hallar la solución de las ecuaciones diferenciales de segundo orden no-homogéneas [15,27,28,33] se procede a determinar la solución de la ecuación homogénea (y_h) y, posteriormente, se calcula la solución particular (y_p). Se escribe el tipo de ecuaciones de interés:

$$ay'' + by' + cy = g(x) \tag{7.116}$$

donde a, b y c son constantes. Una vez determinada la solución de la ecuación diferencial de segundo orden homogénea, se propone la forma algebraica de la función $g(x)$ de forma genérica:

$$g(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0 \tag{7.117}$$

Cuya solución asociada es:

$$y_p(x) = x^s[A_n x^n + A_{n-1}x^{n-1} + A_{n-2}x^{n-2} + \ldots + A_0] \tag{7.118}$$

En caso de que esté presente un término proporcional a uno exponencial

$$g(x) = ae^{\alpha x} \tag{7.119}$$

La solución asociada es:

$$y_p(x) = x^s[e^{\alpha x}] \tag{7.120}$$

En caso de que la función tenga la forma

$$g(x) = a_1 sen(\beta x) + a_2 cos(\beta x) \tag{7.121}$$

Así, la solución particular es:

$$y_p(x) = x^s[A_1 sen(\beta x) + A_2 cos(\beta x)] \tag{7.122}$$

La solución general de la ecuación diferencial de segundo orden de interés se expresa de la siguiente manera:

$$y(x) = y_h(x) + y_p(x) \qquad 7.123)$$

Como ejemplo, se resuelve una ecuación diferencial de segundo orden no homogénea

$$y'' + 4y = x^2 + 3e^x \qquad 7.124)$$

Con las condiciones iniciales $y(0) = 0$ y $y'(0) = 2$.

Como paso inicial, se determina la solución de la ecuación homogénea $y_h''(x) + 4y_h = 0$. La ecuación auxiliar a resolver es:

$$r^2 + 4 = 0 \qquad 7.125)$$

$$r_1 = i2 \quad \text{y} \quad r_2 = -i2 \qquad 7.126)$$

así que $\alpha = 0$ y $\beta = 2$.

La solución de la ecuación homogénea es de la forma:

$$y(x) = e^{\alpha x}[c_1 sen(\beta x) + c_2 cos(\beta x)] \qquad 7.127)$$

$$y(x) = e^{0x}[c_1 sen(2x) + c_2 cos(2x)] \qquad 7.128)$$

$$y_h(x) = c_1 sen(2x) + c_2 cos(2x) \qquad 7.129)$$

Como siguiente paso, se estima la solución particular. Para ello, se escribe la función $g(x)$

$$g(x) = x^2 + 3e^x \qquad 7.130)$$

Luego, se propone la solución particular

$$y_p(x) = Ax^2 + Bx + C + De^x \tag{7.131}$$

La cual debe satisfacer la ecuación diferencial no-homogénea

$$y_p'' + 4y_p = x^2 + 3e^x \tag{7.132}$$

La función (7.131) se deriva en dos ocasiones:

$$y_p'(x) = 2Ax^2 + B + De^x \tag{7.133}$$

$$y_p''(x) = 2A + De^x \tag{7.134}$$

Y se sustituyen las ecuaciones (7.131) y (7.134) en la ecuación 7.132

$$2A + De^x + 4[Ax^2 + Bx + C + De^x] = x^2 + 3e^x \tag{7.135}$$

Asociando los términos, escribimos la ecuación en la forma siguiente:

$$4Ax^2 + 4Bx\,(2A + 4C) + (D + 4D)e^x = x^2 + 3e^x \tag{7.136}$$

Al comparar los factores, resultan las ecuaciones algebraicas siguientes:

$$4A = 1 \quad \text{y} \quad 4B = 0 \tag{7.137}$$

$$2A + 4C = 0 \quad \text{y} \quad 5D = 3 \tag{7.138}$$

Lo que nos permite determinar las constantes

$$A = \tfrac{1}{4} \quad \text{y} \quad B = 0 \tag{7.139}$$

$$C = -\tfrac{1}{8} \quad \text{y} \quad D = \tfrac{3}{5} \tag{7.140}$$

La solución particular es:

$$y_p(x) = \frac{1}{4}x^2 - \frac{1}{8} + \frac{3}{5}e^x \qquad 7.141)$$

Así, la solución general es:

$$y(x) = c_1 sen(2x) + c_2 cos\,(2x) + \frac{x^2}{4} - \frac{1}{8} + \frac{3}{5}e^x \qquad 7.142)$$

Ahora se determinan las constantes usando las condiciones iniciales, sustituyendo y(0) = 0 y y'(0) = 2:

$$y(0) = 0 = c_2 - \frac{1}{8} + \frac{3}{5} \qquad 7.143)$$

$$y'(0) = 2 = 2c_1 + \frac{3}{5} \qquad 7.144)$$

Así hallamos las constantes c_1 y c_2

$$c_1 = \frac{7}{10} \quad \text{y} \quad c_2 = -\frac{19}{40} \qquad 7.145)$$

La solución es:

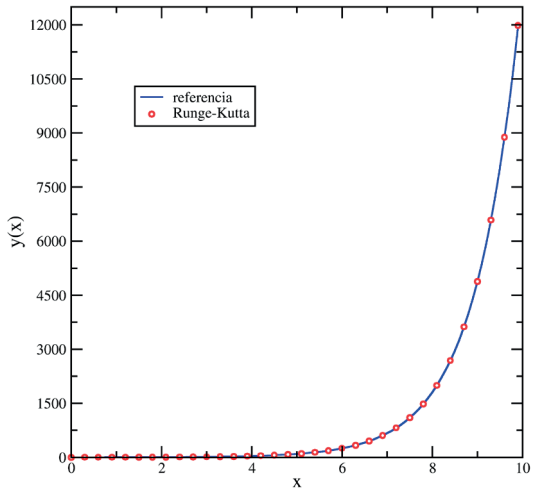
$$y(x) = \frac{7}{10}sen(2x) - \frac{19}{40}cos(2x) + \frac{x^2}{4} - \frac{1}{8} + \frac{3}{5}e^x \qquad 7.146)$$

Esta misma ecuación se grafica en el intervalo de 0 a 10 en la variable independiente *x*. De hecho, para generar la curva solución, se sigue el mismo procedimiento usado en el ejercicio anterior y solo hay que actualizar la expresión de la función .

```
#código 7.9
import matplotlib.pyplot as plt
archivo = open("RungeKutta.dat","w")
# FUNCION
def f(x,y,u):
    return ( x**2. + 3.*math.exp(x) - 4.*y )
```

Además, se calcula la solución mediante el método de Runge-Kutta de cuarto orden. Las dos curvas obtenidas se incluyen en la figura 7.8.

Figura 7.8. Gráfica de la curva solución de la ecuación diferencial de segundo orden



Nota: Es obtenida con el método de Runge Kutta de cuarto orden. Tambien se incluye la solución de referencia.

Se observa que ambas curvas muestran un buen acuerdo. Se incluye el código 7.10 en Python, que permite calcular la curva solución de la ecuación diferencial de segundo orden mediante el método de Runge-Kutta de cuarto orden.

```
#código 7.10
import matplotlib.pyplot as plt
import math

archivo = open("RungeKutta.dat","w")
# FUNCION
def f(x,y,u):
    return ( x**2. + 3.*math.exp(x) – 4.*y )

print('Runge-Kutta de cuarto orden')
x0 =  float(input('x0: '))
y0 =  float(input('y0: '))
u0 =  float(input('u0: '))
n  =  int(input('particiones: '))
h  =  float(input('h: '))
x11 = h*n + x0
print("x11",x11)
x = x0
y = y0
u = u0
```

```
for i in range(0,n+1,1):
    archivo.write(str(x) + ' ' + str(y) + '\n');
    yy = (7/10)*sin(2*x)-(19/40)*cos(2*x);
    zz = (1/4)*x**4-(1/8) +(3/5)*math.exp(x)
    v = yy + zz
    plt.plot(x,y,'bo',markersize=6)
    plt.plot(x,v,'*r',markersize=3)
    k1 = f(x, y, u);
    m1 = u
    k2 = f(x + 0.5*h, y + 0.5*h*m1, u + 0.5*h*k1);
    m2 = u + 0.5*h*k1
    k3 = f(x + 0.5*h, y + 0.5*h*m2, u + 0.5*h*k2);
    m3 = u + 0.5*h*k2
    k4 = f(x + h, y + h*m3, u + h*k3);
    m4 = u + h*k3
    y = y + (h/6.0)*(m1 + 2.0*m2 + 2.0*m3 + m4);
    u = u + (h/6.0)*(k1 + 2.0*k2 + 2.0*k3 + k4);
    x = x+h;
plt.show()
archivo.close;
print("\nArchivo RungeKutta.dat generado");
```

Ejercicios

Resuelva la ecuación de segundo orden con condiciones iniciales mediante el método numérico de Runge-Kutta de cuarto orden:

$$\frac{d^2y}{dx^2} - 4y = 12x$$

con $x_0 = 0, y(x_0) = 4,$ y $y'(x_0) = 1$ obtenga la función solución mediante un procedimiento algebraico y compare ambas curvas obtenidas.

8. MATRICES

Las matrices son entes matemáticos comúnmente llamados arreglos bidimensionales de números. Su representación es a través de renglones y columnas de números dispuestos en arreglos, que pueden contar con el mismo número de estos mismos [34-37]. Las matrices fueron propuestas en 1850, por el británico James Joseph Silverton, y su posterior desarrollo se debe al irlandés Hamilton y al inglés Cayley, en 1853. Por lo general, los elementos de una matriz se representan con una letra minúscula acompañada de dos subíndices (*aij*), que indican el renglón y la columna en donde está ubicado dicho elemento. Se denota a las matrices con letras mayúsculas, por ejemplo, *A.*

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1m} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2m} \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nm} \end{pmatrix}$$

Sus dimensiones se reconocen al indicar los renglones (*n*) y sus columnas (*m*) A_{nxm}. Las matrices tienen definidas las operaciones sumas y el producto, así como la matriz unidad o identidad, entre otras propiedades. Enseguida, se muestran de manera esquemática algunas propiedades de las matrices, así como algunos procedimientos en Python para usar algunos comandos que nos permiten el manejo de matrices.

Suma y producto de matrices

Se exploran algunas propiedades básicas de matrices como la suma de matrices y su producto. Sean las matrices 2x2

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{y} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \qquad 8.1)$$

La suma de A y B se escribe como:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{pmatrix} \quad 8.2)$$

Y el producto de las mismas matrices es:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}\, b_{11} + a_{12}\, b_{21} & a_{11}\, b_{12} + a_{12}\, b_{22} \\ a_{21}\, b_{11} + a_{22}\, b_{21} & a_{21}\, b_{12} + a_{22}\, b_{22} \end{pmatrix} \quad 8.3)$$

Se incluyen dos procedimientos en Python para construir matrices 2x2 y 3x3, donde se incorpora el comando array definido en la biblioteca *numpy.*

```
#código 8.1
import numpy as np
print(“ “)
print(“Matrices 2x2”)
A = np.array([[1, 2], [3, 4]])
print(A)
```

```
#código 8.2
import numpy as np
print(" ")
print("Matrices 3x3")
B = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
print(B)
```

Además, se muestran los procedimientos para sumar y multiplicar matrices 2x2 y 3x3, a través de un código en Python.

Suma

```
#código 8.3
import numpy as np
print(“ “)
print(“Suma de matrices 2x2”)
A = np.array([[-1, 2], [-3, 4]])
B = np.array([[4, 3], [-1, -5]])
C = A + B
print(C)
```

```
#código 8.4
import numpy as np
print(" ")
print("Suma de matrices 3x3")
D = np.array([[-4, 7, -3], [-1, 2, 8] ,  [3, 6, -2]])
E = np.array([[4, 5, 1],   [-8, -5, 5],  [-3, 9, -4]])
F = np.array([[-1, 6, -4], [2, 1, -8],   [7, -2, 3]])
G = D + E - F
print(G)
```

Multiplicación

```
#código 8.5
import numpy as np
print(“ “)
print(“Multiplicación de matrices 2x2, AB”)
A = np.array([[-1, 2], [-3, 4]])
B = np.array([[4, 3], [-1, -5]])
H = A.dot(B)
print(H)
```

```
#código 8.6
import numpy as np
print(" ")
print("Multiplicación de matrices 3x3, AB")
A = np.array([[-1, 2, 3], [-3, 4, 1], [4, -1, 3]])
B = np.array([[-2, 3, 1], [1, 3, -3], [-5, 0, 2]])
I = A.dot(B)
print(I)
```

Determinante de una matriz

El determinante de una matriz cuadrada es la resta de la multiplicación de los elementos de las diagonales, la principal y la secundaria. Particularmente, la matriz A, que es de tamaño 2x2 se calcula como:

$$\det(A) = |A| \tag{8.4}$$

$$\det(A) = (a_{11}\, a_{22} - a_{21}\, a_{12}) \tag{8.5}$$

A manera de ejemplo, establecemos el determinante de la matriz 2x2

$$A = \begin{pmatrix} 3 & -2 \\ -6 & 5 \end{pmatrix} \tag{8.6}$$

El determinante se calcula como sigue:

$$\det(A) = ((3)\ (5) - (-6)\ (-2)) \tag{8.7}$$

$$\det(A) = 3 \tag{8.8}$$

Para el caso de una matriz 3x3, el determinante se puede calcular mediante el método de Laplace, sea la matriz A

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{8.9}$$

El método de los cofactores se escribe de manera esquemática como:

$$|A| = \Sigma_{j=1}^{n} a_{ij}(-1)^{i+j}|A_{ij}| \tag{8.10}$$

donde *i* y *j* son los índices que identifican las componentes de la matriz 3x3:

$$|A| = a_{11}(-1)^{1+1}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} + a_{12}(-1)^{1+2}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}(-1)^{1+3}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \tag{8.11}$$

Se resuelve un ejercicio particular para ejemplificar el método, escribimos la matriz A

$$A = \begin{pmatrix} 1 & 2 & 3 \\ -5 & 5 & 0 \\ -2 & -3 & -1 \end{pmatrix} \tag{8.12}$$

Siguiendo el método de Laplace, escribimos

$$|A| = 1(-1)^{1+1}\begin{vmatrix} 5 & 0 \\ -3 & -1 \end{vmatrix} + 2(-1)^{1+2}\begin{vmatrix} -5 & 0 \\ -2 & -1 \end{vmatrix} + 3(-1)^{1+3}\begin{vmatrix} -5 & 5 \\ -2 & -3 \end{vmatrix} \tag{8.13}$$

Simplificando:

$$|A| = 1(-5) - 2(5) + 3(25) \tag{8.14}$$

Finalmente, hallamos:

$$|A| = 60 \tag{8.15}$$

Se incluyen dos procedimientos en Python para calcular el determinante de una matriz 2x2 y el de una matriz 3x3, los cuales se escriben como sigue:

```
#código 8.7
import numpy as np
print(“ “)
print(“La matriz A 2X2”)
A = np.array([[-1, 2], [-2, 1]])
B = np.linalg.det(A)
print(A)
print(“ “)
print(“El determinante de la matriz A”)
print(B)
```

```
#código 8.8
import numpy as np
print(" ")
print("La matriz A 3X3")
A = np.array([[2, 4, 1], [-3, 5, 4], [-3,5,1]])
B = np.linalg.det(A)
print(A)
print(" ")
print("El determinante de la matriz A")
print(B)
```

La biblioteca necesaria y suficiente es *numpy,* donde está definido el método para calcular el determinante de una matriz.

Matriz adjunta

La matriz adjunta resulta de una transformación lineal de la matriz original. Sea la matriz A

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \tag{8.16}$$

La adjunta de la matriz A se calcula con la expresión siguiente:

$$adj_{ij}(A) = (-1)^{i+j} \det(a_{ij}) \tag{8.17}$$

Se desarrolla la ecuación escribiendo las componentes

$$adj_{11}(A) = (-1)^{1+1} \det(a_{11}) \tag{8.18}$$

$$adj_{12}(A) = (-1)^{1+2} \det(a_{12}) \tag{8.19}$$

$$adj_{21}(A) = (-1)^{2+1} \det(a_{21}) \tag{8.20}$$

$$adj_{22}(A) = (-1)^{2+2}\det(a_{22}) \tag{8.21}$$

La expresión de la matriz adjunta se escribe con base en estos coeficientes calculados. Enseguida, mostramos de manera general la adjunta de una matriz 2x2

$$adj(A) = \begin{pmatrix} adj_{11}(A) & adj_{12}(A) \\ adj_{21}(A) & adj_{22}(A) \end{pmatrix} \tag{8.22}$$

Se resuelve un problema particular con la intención de fijar ideas, sea la matriz A

$$A = \begin{pmatrix} -1 & 6 \\ -3 & -2 \end{pmatrix} \tag{8.23}$$

Se procede a aplicar la fórmula (8.17):

$$adj_{11}(A) = adj_{11}\begin{pmatrix} -1 & 6 \\ -3 & -2 \end{pmatrix} = (-1)^{1+1}\det(-2) = (1)(-2) = -2 \tag{8.24}$$

$$adj_{12}(A) = adj_{12}\begin{pmatrix} -1 & 6 \\ -3 & -2 \end{pmatrix} = (-1)^{1+2}\det(-3) = (-1)(-3) = 3 \tag{8.25}$$

$$adj_{21}(A) = adj_{21}\begin{pmatrix} -1 & 6 \\ -3 & -2 \end{pmatrix} = (-1)^{2+1}\det(6) = (-1)(6) = -6 \tag{8.26}$$

$$adj_{22}(A) = adj_{22}\begin{pmatrix} -1 & 6 \\ -3 & -2 \end{pmatrix} = (-1)^{2+2}\det(-1) = (1)(-1) = -1 \tag{8.27}$$

Con estos elementos calculados construimos la matriz adjunta

$$adj(\mathrm{A}) = \begin{pmatrix} -2 & 3 \\ -6 & -1 \end{pmatrix} \tag{8.28}$$

Para el caso de una matriz 3x3, escribimos la matriz A

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{8.29}$$

Para calcular la matriz adjunta, seguimos el mismo método mostrado a través de la ecuación (8.17), lo que nos permite escribir la matriz adjunta de A de la siguiente manera:

$$adj(A) = \begin{pmatrix} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} & -\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \\ -\begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} & -\begin{vmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{vmatrix} \\ \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix} & -\begin{vmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \end{pmatrix} \tag{8.30}$$

Si resolvemos un problema en particular, podremos identificar de manera más clara las operaciones llevadas a cabo. Como ejemplo, se determina la matriz adjunta de una matriz *3x3*. Sea la matriz *A*

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 5 & 6 & 8 \end{pmatrix} \tag{8.31}$$

Luego, identificamos las componentes de la matriz

$$a_{11} = 1, \quad a_{12} = 2, \quad a_{13} = 3 \tag{8.32}$$

$$a_{21} = 3, \quad a_{22} = 4, \quad a_{23} = 5 \tag{8.33}$$

$$a_{31} = 5, \quad a_{32} = 6, \quad a_{33} = 8 \tag{8.34}$$

Sustituyendo los valores numéricos de las componentes a_{ij} en la ecuación (8.30), obtenemos la matriz adjunta de *A*

$$adj(A) = \begin{pmatrix} \begin{vmatrix} 4 & 5 \\ 6 & 8 \end{vmatrix} & -\begin{vmatrix} 3 & 5 \\ 5 & 8 \end{vmatrix} & \begin{vmatrix} 3 & 4 \\ 5 & 6 \end{vmatrix} \\ -\begin{vmatrix} 2 & 3 \\ 6 & 8 \end{vmatrix} & \begin{vmatrix} 1 & 3 \\ 5 & 8 \end{vmatrix} & -\begin{vmatrix} 1 & 2 \\ 5 & 6 \end{vmatrix} \\ \begin{vmatrix} 2 & 3 \\ 4 & 5 \end{vmatrix} & -\begin{vmatrix} 1 & 3 \\ 3 & 5 \end{vmatrix} & \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} \end{pmatrix} \tag{8.35}$$

Resolviendo los determinantes y simplificando los términos, hallamos que:

$$adj\,(A) = \begin{pmatrix} 2 & 1 & -2 \\ 2 & -7 & 4 \\ -2 & 4 & -2 \end{pmatrix} \qquad 8.36)$$

Se incluye el procedimiento en el código 8.9 de Python para calcular la adjunta de una matriz, el cual se escribe como sigue:

```
#código 8.9
import numpy as np
from numpy import size
from numpy import zeros
from numpy import matrix

#La matriz original
A=matrix([[1,2,-1],[0,1,2],[5,1,1]])
#Se vacia la matriz de cofactores, Acof
Acof=matrix(zeros((3,3)))
#Se elijen las entradas de la matriz
aij=matrix(range(3))

print(“Orden de los elementos, aij\n”,aij)
print("Se vacía la matriz Acof\n",Acof)

for i in range(size(A,0)):
   for j in range(size(A,1)):
      factij = aij[aij != i]
      Mij = aij[aij != j]
      B = A[[[factij[0,0]],[factij[0,1]]],Mij]
      detA = np.linalg.det(B)
      Acof[i,j]=detA*np.power(-1,i+j)
print(" ")
print( "matriz original A\n",A)
print(" ")
print( "Matriz adjunta\n",Acof)
print(" ")
```

Ejercicios

[1] 1Calcular la matriz adjunta de una matriz 2x2, cuya expresión es:

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

[2] Sea la matriz A, de tamaño 3x3, cuyas componentes se muestran a continuación:

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

hallar la matriz adjunta de A.

Matriz traspuesta

Otra propiedad relevante de las matrices es la traspuesta de una matriz, la cual se determina intercambiando renglones por columnas. Sea la matriz A:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad 8.37)$$

Su traspuesta es:

$$tras(A) = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{pmatrix} \quad 8.38)$$

A manera de ejemplo,determinamos la traspuesta de la matriz:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad 8.39)$$

Y obtenemos la matriz:

$$tras\ (A) = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \quad 8.40)$$

Se muestran dos procedimientos en Python para calcular la traspuesta de una matriz 2x2 y 3x3.

```
#código 8.10
import numpy as np
A = np.array([[1, 2], [3, 4], [5, 6]])
print(“ “)
print(“Matriz A, 3x2, “)
print(A)

print(“ “)
print(“Matriz traspuesta de A, 2x3”)
print(A.transpose())
```

```
#código 8.11
import numpy as np
B = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(" ")
print("Matriz B, 3x3")
print(B)

print(" ")
print("Matriz traspuesta de B, 3x3")
print(B.transpose())
```

Matriz inversa

La inversa de una matriz se calcula mediante el uso de las propiedades de las matrices ya mostradas. La expresión matemática a seguir es:

$$inv(A) = \frac{1}{det(A)}\, tras(adj(A)) \tag{8.41}$$

-

Siendo $tras(adj(A))$ la traspuesta de la matriz adjunta de A y $det(A)$ el determinante de la misma matriz.

Resolvemos un ejercicio particular para ejemplificar el procedimiento descrito. Sea la matriz A, 2x2:

$$A = \begin{pmatrix} 2 & -1 \\ -2 & 3 \end{pmatrix} \tag{8.42}$$

El determinante de la matriz A es:

$$det(A) = 4 \tag{8.43}$$

La matriz adjunta de A se escribe como sigue:

$$adj(A) = \begin{pmatrix} 3 & 2 \\ 1 & 2 \end{pmatrix} \tag{8.44}$$

Como tercer paso, se determina la traspuesta de la matriz adjunta de A:

$$tras(adj(A)) = \begin{pmatrix} 3 & 1 \\ 2 & 2 \end{pmatrix} \tag{8.45}$$

De esta manera, la matriz inversa de A tiene la siguiente expresión:

$$inv(A) = \frac{1}{4} \begin{pmatrix} 3 & 1 \\ 2 & 2 \end{pmatrix} \tag{8.46}$$

De manera inmediata, verificamos si la matriz hallada es realmente la inversa de A. La forma de comprobarlo es multiplicar la matriz A con su inversa y, como resultado, debemos obtener la matriz unitaria:

$$(A)(inv(A)) = \begin{pmatrix} 2 & -1 \\ -2 & 3 \end{pmatrix} \frac{1}{4} \begin{pmatrix} 3 & 1 \\ 2 & 2 \end{pmatrix} \quad 8.47)$$

Después de hacer las operaciones algebraicas requeridas, hallamos que:

$$(A)(inv(A)) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad 8.48)$$

Que es precisamente la matriz unitaria.

Se incluyen dos códigos en Python (8.12 y 8.13) para calcular la matriz inversa de A, usando la instrucción *np.linalg.inv(A),* definida en la biblioteca *numpy.* Se abordan dos casos: uno donde está involucrada una matriz 2x2, y el segundo donde se trabaja con una matriz 3x3. En los mismos códigos se incluye un procedimiento para verificar que es correcta la matriz inversa, por lo que se multiplica el inverso de A por la misma matriz original A, lo cual tiene que dar como resultado la matriz unitaria.

```
#código 8.12
import numpy as np
print(“ “)
print(“La matriz C”)
A = np.array([[1, 2], [2, -1]])
B = np.linalg.inv(A)
C = A.dot(B)
print(“La matriz inversa de A”)
print(B)
print(“  “)
print(“La matriz unitaria C”)
print(C)
```

```
#código 8.13
import numpy as np
print(" ")
print("La matriz D")
D = np.array([[-1, 3, 4], [1, -2, 5], [-3, 7, 5]])
E = np.linalg.inv(D)
F = E.dot(D)
print("La matriz inversa de D")
print(E)
print("  ")
print("La matriz unitaria F")
print(F)
```

Ahora se resuelve un ejercicio donde se involucra una matriz 3x3, sea la matriz *A:*

$$A = \begin{pmatrix} 1 & 3 & -1 \\ 2 & -1 & 3 \\ 1 & 2 & -3 \end{pmatrix} \quad 8.49)$$

Se determina la matriz inversa siguiendo el procedimiento involucrado en la ecuación (8.41). Primero se calcula la matriz adjunta, usando la ecuación (8.17).

Se calculan los términos adjuntos de la matriz *A:*

$$adj_{11}(A) = (-1)^{1+1}\begin{vmatrix} -1 & 3 \\ 2 & -3 \end{vmatrix} = -3 \qquad 8.50)$$

$$adj_{12}(A) = (-1)^{1+2}\begin{vmatrix} 2 & 3 \\ 1 & -3 \end{vmatrix} = 9 \qquad 8.51)$$

$$adj_{13}(A) = (-1)^{1+3}\begin{vmatrix} 2 & -1 \\ 1 & 2 \end{vmatrix} = 5 \qquad 8.52)$$

$$adj_{21}(A) = (-1)^{2+1}\begin{vmatrix} 3 & -1 \\ 2 & -3 \end{vmatrix} = 7 \qquad 8.53)$$

$$adj_{22}(A) = (-1)^{2+2}\begin{vmatrix} 1 & -1 \\ 1 & -3 \end{vmatrix} = -2 \qquad 8.54)$$

$$adj_{23}(A) = (-1)^{2+3}\begin{vmatrix} 1 & 3 \\ 1 & 2 \end{vmatrix} = 1 \qquad 8.55)$$

$$adj_{31}(A) = (-1)^{3+1}\begin{vmatrix} 3 & -1 \\ -1 & 3 \end{vmatrix} = 8 \qquad 8.56)$$

$$adj_{32}(A) = (-1)^{3+2}\begin{vmatrix} 1 & -1 \\ 2 & 3 \end{vmatrix} = -5 \qquad 8.57)$$

$$adj_{33}(A) = (-1)^{3+3}\begin{vmatrix} 1 & 3 \\ 2 & -1 \end{vmatrix} = -7 \qquad 8.58)$$

La matriz adjunta de *A* se escribe a continuación:

$$adj(A) = \begin{pmatrix} -3 & 9 & 5 \\ 7 & -2 & 1 \\ 8 & -5 & -7 \end{pmatrix} \qquad 8.59)$$

Posteriormente, se calcula la inversa de la matriz de *A:*

$$inv(A) = \frac{1}{19}\begin{pmatrix} -3 & 7 & 8 \\ 9 & -2 & -5 \\ 5 & 1 & -7 \end{pmatrix} \tag{8.60}$$

Finalmente, se procede a comprobar que la matriz inversa de *A* es correcta, por lo que se multiplica la matriz *A* por su inversa:

$$inv(A) = \frac{1}{19}\begin{pmatrix} -3 & 7 & 8 \\ 9 & -2 & -5 \\ 5 & 1 & -7 \end{pmatrix}\begin{pmatrix} 1 & 3 & -1 \\ 2 & -1 & 3 \\ 1 & 2 & -3 \end{pmatrix} \tag{8.61}$$

Al llevar a cabo las operaciones involucradas, hallamos que:

$$inv(A) = \frac{1}{19}\begin{pmatrix} -3+14+8 & -9-7-16 & 3+21-24 \\ 9-4-5 & 27+2-10 & -9-6+15 \\ 5+2-7 & 15-1-14 & -5+3+21 \end{pmatrix} \tag{8.62}$$

Así, encontramos la matriz identidad:

$$inv(A)A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{8.63}$$

Ejercicios

[3] Calcular la matriz inversa de la matriz $A = \begin{pmatrix} 1 & 4 \\ 2 & 3 \end{pmatrix}$

[4] Determinar la matriz inversa de tamaño 3x3, cuya expresión es $A = \begin{pmatrix} 1 & 2 & -1 \\ 2 & -1 & 0 \\ -1 & 3 & 1 \end{pmatrix}$

Sistema de ecuaciones algebraicas

Un sistema de ecuaciones algebraicas [21] está compuesto por ecuaciones que involucran *n* incógnitas, las cuales son solución de tal conjunto de ecuaciones. Sean a_{ij} los coeficientes que acompañan a las variables x_i y b_{ii}, las constantes independientes

que, a su vez, se representan como un vector B. Por consiguiente, un conjunto de ecuaciones algebraicas lineales se escribe como sigue:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + \ldots + a_{1m}x_m = b_1 \quad 8.64)$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + \ldots + a_{2m}x_m = b_2 \quad 8.65)$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + \ldots + a_{3m}x_m = b_3 \quad 8.66)$$

$$\ldots + \ldots + \ldots + \ldots + \ldots + \ldots = \ldots$$

$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + a_{n4}x_4 + \ldots + a_{nm}x_m = b_n \quad 8.67)$$

El sistema de ecuaciones algebraicas lineales es inmediato de resolver cuando n=m. El método de la matriz inversa es simple y sirve para darle solución a este tipo de sistemas de ecuaciones algebraicas. Dicho método consiste en calcular primero el determinante de la matriz A, para asegurarse que no es una matriz singular:

$$\det(A) \neq 0 \quad 8.68)$$

Luego, se calcula la matriz inversa de A:

$$inv(A) = \frac{1}{\det(A)} \text{ tras (adj } (A)) \quad 8.69)$$

Al final, se multiplica la matriz inversa por el vector B, que contiene los números de solución del sistema de ecuaciones algebraicas:

$$B = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \quad 8.70)$$

Si X es el vector de variables:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} \tag{8.71}$$

Entonces la solución del sistema de ecuaciones algebraicas se determina al hacer la operación siguiente:

$$X = inv(A)B \tag{8.72}$$

A continuación, a manera de ejemplo se resuelve un sistema de ecuaciones algebraicas particular. El sistema de interés se escribe como:

$$3x + y - 2z = 4 \tag{8.73}$$

$$x + 3y - 4z = -2 \tag{8.74}$$

$$-4x - 2y + 5z = 3 \tag{8.75}$$

Posteriormente, se identifica la matriz que contiene los coeficientes del sistema de ecuaciones:

$$A = \begin{pmatrix} 3 & 1 & -2 \\ 1 & 3 & -4 \\ -4 & -2 & 5 \end{pmatrix} \tag{8.76}$$

cuyo determinante es *det(A) = 28*. La matriz inversa se calcula de acuerdo con la ecuación (8.69), y tiene la expresión siguiente:

$$inv(A) = \begin{pmatrix} 0.583 & -0.083 & 0.166 \\ 0.916 & 0.583 & 0.833 \\ 0.833 & 0.166 & 0.666 \end{pmatrix} \tag{8.77}$$

La solución del sistema de ecuaciones algebraicas se obtiene siguiendo la formulación de la ecuación (8.72), la cual se muestra a continuación:

$$X = \begin{vmatrix} 0.583 & -0.083 & 0.166 \\ 0.916 & 0.583 & 0.833 \\ 0.833 & 0.166 & 0.666 \end{vmatrix} \begin{vmatrix} 4 \\ -2 \\ 3 \end{vmatrix} \tag{8.78}$$

Donde:

$$B = \begin{vmatrix} 4 \\ -2 \\ 3 \end{vmatrix} \tag{8.79}$$

Finalmente, haciendo los pasos algebraicos señalados, encontramos la solución del sistema de ecuaciones (8.73 a 8.75)

$$X = \begin{vmatrix} 3 \\ 5 \\ 5 \end{vmatrix} \tag{8.80}$$

El procedimiento descrito involucra operaciones algebraicas que es posible llevar a cabo mediante un código en Python, que se incluye líneas abajo y donde la biblioteca *numpy* es usada para invocar las instrucciones que permiten llevar a cabo las operaciones requeridas.

```
#código 8.14
import numpy as np

A = np.array([[ 3,1,-2], [1,3,-4], [-4,-2,5]])
B = np.array([[4], [-2], [3]])
print(" ")
print("El sistema de ecuaciones ")
print(" 3x +   y - 2z  = 4")
print("   x + 3y - 4z = -2")
print("-4x - 2y + 5z  = 3")

print(" ")
print("La matriz A es:")
print(A)
print("El vector, B, del lado derecho de la igualdad es:")
print(B)

print(" ")
print("Se verifica el determinante de la matriz A")
det = np.linalg.det(A)
print(det)

if (det == 0.0) :
  print("La matriz es singular, nada que hacer")
else:
  print(" ")
  print("Se calcula la matriz inversa de A")
  C = np.linalg.inv(A)
  print(C)
  print(" ")
  print("Se multiplica la inversa de A con el vector B,")
  print("y con ello encontramos el vector solución (Vect):")
  print(" ")
  print("Vect = ")
  print(" [[x ]")
  print("  [y ]")
  print("  [z ]]")
  print(" ")
  print("es decir")
  R = C.dot(B)
  print(" ")
  print("Vect = \n",R)
```

Ejercicios

Resolver el sistema de ecuaciones algebraicas:

[1]
$3x + y - 2z = 4$
$x + 3y + 4z = 5$
$-4x - 2y + 5z = 3$

[2]
$x + 3y - 1z = 1$
$2x - y + 7z = 2$
$-x + 3y + 2z = -1$

Método de Gauss-Jordan

El método de Gauss-Jordan [38-40] utiliza operaciones con matrices para resolver sistemas de ecuaciones de N número de variables. También se usa para hallar la matriz inversa de una matriz de interés. Dicho método debe su nombre a los alemanes Carl Friedrich Gauss (1777-1855) y Wilhelm Jordan (1842-1899).

Las operaciones requeridas obedecen a una serie de algoritmos; es decir, son acciones algebraicas y de procedimiento efectivas, que tienen como objetivo reducir el sistema de ecuaciones planteado originalmente a un sistema de ecuaciones equivalente, que muestre una incógnita de menos. Dicho procedimiento da como resultado una matriz escalonada.

En caso de que se requiera determinar la matriz inversa de la matriz A, entonces se agregan términos a la matriz; es decir, una matriz extendida donde los elementos nuevos corresponden a una matriz identidad. Posteriormente, mediante operaciones de suma, resta, multiplicación e intercambio de líneas, se determina una matriz identidad donde antes estaban los elementos de la matriz original; y donde estaban los elementos de la matriz identidad surgen los elementos de la matriz inversa. Hagamos un ejemplo numérico para fijar ideas.

Sea A la matriz 3x3, cuyos elementos son los siguientes:

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

Que es la misma que usamos en la sección donde se discutió la matriz inversa, y que se identifica con la ecuación (8.49), lo que nos obliga a obtener el mismo resultado de esa sección. Ahora escribimos la matriz extendida

$$A_{ext} = \left(\begin{array}{ccc|ccc} 1 & 3 & -1 & 1 & 0 & 0 \\ 2 & -1 & 3 & 0 & 1 & 0 \\ 1 & 2 & -3 & 0 & 0 & 1 \end{array}\right) \qquad 8.81)$$

Lo que buscamos es llevar los elementos de la matriz original a elementos de una matriz unitaria. De manera que comenzamos con modificar las filas 2 y 3, llamadas f_2 y f_3, respectivamente. Las operaciones son las siguientes: $f_2 = f_2 - 2f_1$ y $f_3 = f_3 - f_1$, lo que nos lleva a la matriz:

$$A_{ext} = \left(\begin{array}{ccc|ccc} 1 & 3 & -1 & 1 & 0 & 0 \\ 0 & -7 & 5 & -2 & 1 & 0 \\ 0 & -1 & -2 & -1 & 0 & 1 \end{array}\right) \qquad 8.82)$$

El siguiente paso es dividir todos los miembros de la fila 2 por -7; $f_2 = f_2/(-7)$, así obtenemos

$$A_{ext} = \left(\begin{array}{ccc|ccc} 1 & 3 & -1 & 1 & 0 & 0 \\ 0 & 1 & -5/7 & 2/7 & -1/7 & 0 \\ 0 & -1 & -2 & -1 & 0 & 1 \end{array}\right) \qquad 8.83)$$

Posteriormente, modificamos las filas 1 y 3, $f_1 = f_1 - 3f_2$ y $f_3 = f_3 + f_2$. Estas operaciones algebraicas nos llevan a la siguiente expresión de la matriz extendida

$$A_{ext} = \left(\begin{array}{ccc|ccc} 1 & 0 & 8/7 & 1/7 & 3/7 & 0 \\ 0 & 1 & -5/7 & 2/7 & -1/7 & 0 \\ 0 & 0 & -19/7 & -5/7 & -1/7 & 1 \end{array}\right) \qquad 8.84)$$

Ahora, buscamos que la diagonal principal esté constituida por solo tres números, 1. Para ello, hacemos la operación algebraica $f_3 = (-7/19)f_3$; lo que nos lleva a la ecuación siguiente:

$$A_{ext} = \left(\begin{array}{ccc|ccc} 1 & 0 & 8/7 & 1/7 & 3/7 & 0 \\ 0 & 1 & -5/7 & 2/7 & -1/7 & 0 \\ 0 & 0 & 1 & 5/19 & 1/19 & -7/19 \end{array}\right) \tag{8.85}$$

Hasta aquí hemos anulado los elementos por debajo de la diagonal principal; falta anularlos por arriba de la diagonal principal. Para lograr nuestro objetivo, hacemos las siguientes operaciones algebraicas: $f_3 = f_1 - (8/7)f_3$ y $f_2 = f_2 + (5/7)f_3$, lo que nos lleva a la expresión siguiente:

$$A_{ext} = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & -3/19 & 7/19 & 8/19 \\ 0 & 1 & 0 & 9/19 & -2/19 & -5/19 \\ 0 & 0 & 1 & 5/19 & 1/19 & -7/19 \end{array}\right) \tag{8.86}$$

Finalmente, factorizando (1/19), la matriz que buscamos es:

$$inv(A) = \frac{1}{19}\begin{pmatrix} -3 & 7 & 8 \\ 9 & -2 & -5 \\ 5 & 1 & -7 \end{pmatrix} \tag{8.87}$$

que coincide con la ecuación (8.60).

A continuación, se incluye el código 8.15 en Python, donde se calcula la matriz inversa usando el método de Gauss-Jordan. La matriz A es la misma que se usó para ejemplificar el método.

```
#código 8.15
import numpy as np
A = np.array([[3,   1, -2],
              [1,   3, -4],
              [-4, -2,  5]], dtype=float)
n = len(A[0])
m = len(A)
n2 = n*2
print('Matriz original n x m:',n,'x',m)
print(A)
Midentidad = np.identity(n)
B=np.concatenate((A,Midentidad),axis=1)
n  = len(B)
m = len(B[0])
print('Matriz aumentada n x m:',n,'x',m)
print(B)
for i in range(0,n-1,1):
    coli = abs(B[i:,i])
    cota = np.argmax(coli)
    if (cota !=0):
        aux = np.copy(B[i,:])
        B[i,:] = B[cota+i,:]
        B[cota+i,:] = aux

for i in [i for i in range(0,n-1,1) if B[i,i] != 0]:
    q = i+1
    for k in range(q,n,1):
        alfa = -B[k,i]/B[i,i]
        B[k,:] += B[i,:]*alfa

nend = n-1
for i in [i for i in range(nend,0-1,-1) if B[i,i] != 0]:
    q = i-1
    for k in range(q,0-1,-1):
        beta = -B[k,i]/B[i,i]
        B[k,:] += B[i,:]*beta
    B[i,:] = B[i,:]/B[i,i]

AInversa = np.copy(B[:,n:])
C = np.linalg.inv(A)

print("  ")
print('Inversa de A, Gauss-Jordan: ')
print(AInversa)
print("  ")
print('Inversa de A, linalg.inv: ')
print(C)
```

En el código 8.15 se usa la biblioteca *numpy* para definir matrices. Primero, se construye la matriz de interés A, y se incluye la información de las dimensiones *n* y *m*. Posteriormente, se calcula la matriz identidad y se concatena a la matriz original para formar la matriz aumentada B, que se muestra junto con sus nuevas dimensiones. Después, se lleva a cabo el proceso de pivoteo y se procede con la diagonalización de la matriz original, colocando solo tres números *1* en ella; comenzando con la parte inferior de la matriz y terminando con la parte superior. Se calcula también la matriz inversa de A, llamada C, por medio del comando *np.linalg.inv(A)*, para verificar que se obtenga el mismo resultado.

Por otro lado, si el objetivo es dar solución a un sistema de ecuaciones algebraicas, el método de Gauss-Jordan también es efectivo. Para ejemplificar su aplicación, se procede a dar solución al sistema de ecuaciones mostrado con las ecuaciones (8.73), (8.74) y 8.75):

$3x + y - 2z = 4$

$x + 3y - 4z = -2$

$-4x - 2y + 5z = 3$

Este sistema de ecuaciones algebraicas se puede representar como $AX = B$, de manera que nos interesa saber los valores del vector X.

Como primer paso, se escribe la matriz extendida

$$A_{ext} = \left(\begin{array}{ccc|c} 3 & 1 & -2 & 4 \\ 1 & 3 & -4 & -2 \\ -4 & 2 & 5 & 3 \end{array}\right) \qquad 8.88)$$

Se intercambian los reglones 1 y 2

$$A_{ext} = \left(\begin{array}{ccc|c} 1 & 3 & -4 & -2 \\ 3 & 1 & -2 & 4 \\ -4 & 2 & 5 & 3 \end{array}\right) \qquad 8.89)$$

Se llevan a cabo las operaciones algebraicas siguientes: $f_2 = f_2 - 3f_1$ y $f_3 = f_3 + 4f_1$, lo que arroja la expresión siguiente

$$A_{ext} = \left(\begin{array}{ccc|c} 1 & 3 & -4 & -2 \\ 0 & -8 & 10 & 10 \\ 0 & 10 & -11 & -5 \end{array}\right) \qquad 8.90)$$

Enseguida, se realizan las operaciones indicadas a continuación: $f_3 = f_3/10$ y $f_2 = -f_2/8$

$$A_{ext} = \left(\begin{array}{ccc|c} 1 & 3 & -4 & -2 \\ 0 & 1 & -10/8 & -10/8 \\ 0 & 1 & -11/8 & -5/10 \end{array}\right) \qquad 8.91)$$

Ahora, se resta la línea 2 de la línea 3, $f_3 = f_3 - f_2$

$$A_{ext} = \left(\begin{array}{ccc|c} 1 & 3 & -4 & -2 \\ 0 & 1 & -10/8 & -10/8 \\ 0 & 0 & 3/20 & 3/4 \end{array}\right) \qquad 8.92)$$

El paso siguiente para lograr tener la diagonal principal constituida por tres números 1, es multiplicar por el factor (20/3) a la línea 3, $f_3 = (20/3)f_3$:

$$A_{ext} = \left(\begin{array}{ccc|c} 1 & 3 & -4 & -2 \\ 0 & 1 & -10/8 & -10/8 \\ 0 & 0 & 1 & 5 \end{array}\right) \qquad 8.93)$$

Toca el turno de la parte de arriba de la diagonal principal. Se requiere cambiar los números relacionados por ceros y, con ello, obtener la matriz unitaria. Para ello, hacemos las operaciones algebraicas siguientes: $f_1 = f_1 - 3f_2$ y $f_2 = (10/8)f_3 + f_2$

$$A_{ext} = \left(\begin{array}{ccc|c} 1 & 0 & -1/4 & 7/4 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 5 \end{array}\right) \qquad 8.94)$$

Por último, se realiza la operación $f_1 = f_1 + (1/4)f_3$, lo que nos lleva a la expresión siguiente:

$$A_{ext} = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 5 \end{array}\right) \qquad 8.95)$$

De manera que la solución del sistema de ecuaciones algebraicas es:

$$X = \begin{pmatrix} 3 \\ 5 \\ 5 \end{pmatrix} \qquad 8.96)$$

Se incluye el código 8.16 en Python, que resuelve el sistema de ecuaciones algebraicas usando el método de Gauss-Jordan.

```
#código 8.16
import numpy as np
A = np.array([[3,   1, -2],
              [1,   3, -4],
              [-4, -2,  5]], dtype=float)
B = np.array([[4],
              [-2],
              [3]])
n = len(A[0])
m = len(A)
print(“  “)
print(‘Matriz original n x m:’,n,’x’,m)
print(A)
C= np.concatenate((A,B),axis=1)
n = len(C)
m = len(C[0])
print(“  “)
print(‘Matriz aumentada n x m:’,n,’x’,m)
print(C)
for i in range(0,n-1,1):
    coli = abs(C[i:,i])
    cota = np.argmax(coli)
    if (cota !=0):
        aux = np.copy(C[i,:])
        C[i,:] = C[cota+i,:]
        C[cota+i,:] = aux
for i in [i for i in range(0,n-1,1) if C[i,i] != 0]:
    q = i+1
    for k in range(q,n,1):
        alfa = -C[k,i]/C[i,i]
        C[k,:] += C[i,:]*alfa

nend = n-1
mend = m-1
for i in [i for i in range(nend,0-1,-1) if C[i,i] != 0]:
    q = i-1
    for k in range(q,0-1,-1):
        beta = -C[k,i]/C[i,i]
        C[k,:] += C[i,:]*beta
    C[i,:] = C[i,:]/C[i,i]
X = np.copy(C[:,mend])
print("  ")
print('Vector solución X = [x,y,z]: ')
print(X)
```

Eigenvectores y eigenvalores

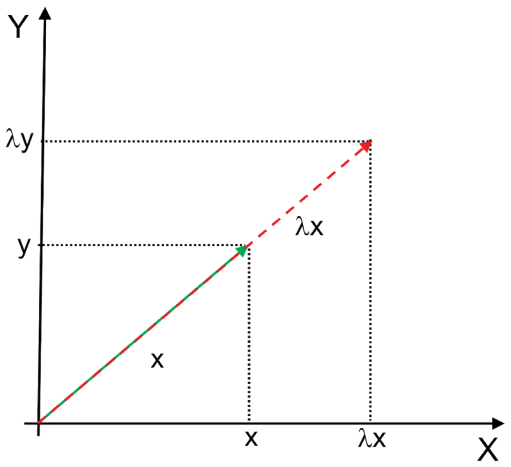
Los llamados eigenvectores de un operador lineal son aquellos vectores no nulos y que, cuando sufren una transformación por algún operador, esto los lleva a un múltiplo escalar de sí mismos, lo cual se traduce a que no hay cambios en su dirección. La raíz eigen tiene su origen del alemán y significa propio, lo que nos permite usar los términos en castellano valores y vectores, propios de una transformación lineal o de una matriz. Más y extensa información acerca de este tema puede hallarse en libros de álgebra lineal [37].

Sea A una matriz cuadrada (n por n) y X un vector diferente de cero, de manera que la operación *AX* nos da como resultado un escalar λ multiplicado por el vector X:

$$AX = \lambda X \tag{8.97}$$

cuando la ecuación de arriba se cumple, se dice que λ es un eigenvalor o valor propio, y que X es un eigenvector o vector propio. En la figura 8.1 se esquematiza esta idea.

Figura 8.1. El escalar λ afecta al vector original X



La ecuación (8.97) se puede escribir como:

$$(\lambda I - A)X = 0 \tag{8.98}$$

Donde I es la matriz identidad. De manera que tenemos una diferencia de matrices, lo cual nos da como resultado una nueva matriz. Entonces (λI-A), debe cumplir con algunas condiciones. A saber, esta debe ser singular y no invertible, por lo que su determinante debe ser igual a cero.

$$det(\lambda I - A) = 0 \tag{8.99}$$

A través de esta ecuación se pueden hallar los eigenvalores de la matriz A. Al calcular el determinante encontramos el polinomio característico de grado n, por lo que podemos hallar n eigenvalores. Finalmente, con la ayuda de los eigenvalores podemos determinar los eigenvectores. Incluimos un ejercicio para ejemplificar el procedimiento descrito hasta aquí.

Ejercicio 1. Sea A una matriz 2 por 2 definida como:

$$A = \begin{pmatrix} 2 & -12 \\ 1 & -5 \end{pmatrix} \qquad 8.100)$$

Hallar sus valores y sus vectores propios.

Primero, determinamos la matriz (λI-A):

$$(\lambda I - A) = \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 2 & -12 \\ 1 & -5 \end{pmatrix} = \begin{pmatrix} \lambda - 2 & 12 \\ -1 & \lambda + 5 \end{pmatrix} \qquad 8.101)$$

De manera que:

$$det(A - \lambda \mathrm{I}) = det \begin{pmatrix} \lambda - 2 & 12 \\ -1 & \lambda + 5 \end{pmatrix} = \lambda 2 + 3\lambda + 2 = (\lambda + 1)(\lambda + 2) \qquad 8.102)$$

Así:

$$(\lambda + 1)(\lambda + 2) = 0 \qquad 8.103)$$

Los eigenvalores son -1 y -2.

Para calcular los eigenvectores escribimos:

$$(\lambda \mathrm{I} - A)X = 0 \qquad 8.104)$$

Lo que nos lleva a la ecuación:

$$\begin{pmatrix} \lambda - 2 & 12 \\ -1 & \lambda + 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad 8.105)$$

Ahora se hace uso de los eigenvalores. Sea $\lambda = -1$, encontramos el sistema de ecuaciones siguiente:

$$\begin{aligned} -3x_1 + 12x_2 &= 0 \\ -x_1 + 4x_2 &= 0 \end{aligned} \qquad 8.106)$$

De manera que hallamos que $x_1=4t$ y $x_2=t$, donde t es un número real. Más adelante determinaremos el valor de *t*, así:

$$X_1 = \begin{pmatrix} 4t \\ t \end{pmatrix} \qquad 8.107)$$

Para $\lambda=-2$, encontramos el sistema de ecuaciones:

$$\begin{aligned} -4x_1 + 12x_2 &= 0 \\ -x_1 + 3x_2 &= 0 \end{aligned} \qquad 8.108)$$

Al resolver el sistema, hallamos que $x_1=3t$ y $x_2=t$, por lo que podemos escribir:

$$X_2 = \begin{pmatrix} 3t \\ t \end{pmatrix} \qquad 8.109)$$

Para determinar *t* debemos exigirles a los vectores X_1 y X_2 que estén normalizados. De manera que debemos calcular la norma de cada uno de ellos $|X|=1$. Se despeja *t* para el caso del eigenvector X_1.

Debido a que cada eigenvector debe estar normalizado y que

$$|X_1| = t\sqrt{17} \qquad 8.110)$$

Determinamos el valor de *t*

$$t = \frac{1}{\sqrt{17}} \qquad 8.111)$$

Por otro lado, despejamos t para el caso del eigenvector X_2

$$\begin{aligned} |X_2| &= 1 \\ |X_2| &= t\sqrt{10} \\ t &= \frac{1}{\sqrt{10}} \end{aligned} \qquad 8.112)$$

Finalmente, los eigenvectores son:

$$X_1 = \begin{pmatrix} \frac{4}{\sqrt{17}} \\ \frac{1}{\sqrt{17}} \end{pmatrix} \quad \text{y} \quad X_2 = \begin{pmatrix} \frac{3}{\sqrt{10}} \\ \frac{1}{\sqrt{10}} \end{pmatrix} \qquad 8.113)$$

Este mismo ejercicio se puede resolver de manera numérica, usando el procedimiento que se incluye en el código 8.17 en Python. Se hace uso de la biblioteca *numpy* para construir la matriz M de interés y, en particular, se usa la instrucción *linalg.eig(M)* para calcular los valores y vectores propios.

```
#código 8.17
import numpy as np
# Se escribe la matriz 2x2 (M)
M = np.array([[2, -12],
                    [1, -5]])
print(“  “)
print(“Se imprime la matriz original:\n”, M)
print(“  “)

# La instrucción linalg.eig(m) calcula los
valores y vectores propios

val, vec = np.linalg.eig(M)

# Los valores propios de la matriz M
print("Eigenvalores de la matriz M:\n", val)
print("  ")
# printing eigen vectors
print("Eingenvectores de la matriz M:\n",
vec)
print("  ")
print("los números del lado izquierdo
corresponden al primer eigenvector")
print("[x1  ] ")
print("[x2  ] ")
print("los números del lado derecho
corresponden al segundo eigenvector")
print("[  x1] ")
print("[  x2] ")
```

Ejercicios

[1] Calcular los eigenvectores y los eigenvalores de la matriz cuadrada $A = \begin{pmatrix} 3 & 2 \\ 1 & 4 \end{pmatrix}$

[2] Calcular los eigenvectores y los eigenvalores de la matriz cuadrada A, donde:

$$A = \begin{pmatrix} 5 & 3 & 4 \\ 3 & 2 & -3 \\ -1 & 4 & 1 \end{pmatrix}$$

[3] Calcular los eigenvectores y los eigenvalores de la matriz A con tamaño 4x4

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

9. DINÁMICA MOLECULAR

Esta es una técnica de simulación a nivel molecular determinista en el sentido de que la segunda ley de Newton describe la evolución espacial y temporal de los átomos durante la simulación por computadora. Dicha solución es también llamada algoritmo, con el cual movemos a todos los átomos de manera simultánea. La fuerza de interacción entre sitios es el parámetro más importante, pues se emplea para evaluar el algoritmo y, con ello, mover a los átomos o moléculas [41,42]. Hay procedimientos que nos permiten mantener en constante promedio la variable que es conveniente en el análisis del sistema de interés. Estos procedimientos están basados en ecuaciones determinadas o construidas para dichos propósitos. En caso de necesitar mantener la temperatura o presión, se recurre a un termostato o a un barostato, entre otros recursos.

De manera esquemática, se muestran los elementos o secciones que componen un seudocódigo típico de dinámica molecular.

```
# 9.1
inicio del programa

parámetros de entrada (potencial de interacción)
parámetros de entrada (temperatura, presión, etc.)
se lee la configuración inicial (posiciones y velocidades de las moléculas)
do i = 1, npasos
    cálculo de fuerza
    se mueven las moléculas usando un algoritmo (segunda ley de Newton)
    cálculo de propiedades termodinámicas (valores instantáneos)
enddo
cálculo de propiedades termodinámicas (valores promedios)
se escribe la configuración final (posiciones de las moléculas)

Fin del programa
```

La fuerza de interacción entre cada dos sitios es escrita de acuerdo con la segunda ley de Newton:

$$F_1 = m_i \frac{d^2 r_i}{dt^2} \tag{9.1}$$

Donde m_i, r_i y F_i son la masa, la posición del átomo i y la fuerza que se aplica sobre la misma partícula. El tiempo es representado por t.

$$F_1 = \Sigma F_{ij} \tag{9.2}$$

Donde F_{ij} es la fuerza de interacción entre átomos i y j. Dicha fuerza es considerada central, por lo que es posible derivarla de una función potencial, como se muestra en la ecuación siguiente:

$$\Sigma F_{ij} = -\Sigma \frac{dU(r_{ij})}{dr_{ij}} \frac{r_{ij}}{|r_{ij}|} \tag{9.3}$$

Donde U es la función potencial que modela la interacción entre átomos. $r_{ij} = |r_i - r_j|$ es la magnitud de la distancia entre centros de átomos. Particularmente, la función potencial es posible representarla como una colección de términos que involucra las interacciones entre varios sitios

$$U(r_{ij}) = \Sigma U_1(r_i) + \Sigma\Sigma U_2(r_i, r_j) + \Sigma\Sigma\Sigma U_3(r_i, r_j, r_k) + \ldots \quad 9.4)$$

El primer término representa la fuerza externa sobre el átomo *i*. El segundo es la interacción entre cada dos sitios de interacción. El siguiente representa la interacción entre cada tres sitios, y así sucesivamente. La forma funcional más usada es el potencial efectivo por pares. Por lo general, la interacción entre átomos de diferentes moléculas se modela a través de dos funciones potenciales, que toman en cuenta la interacción de naturaleza débil de corto alcance o fuerza de van der Waals, en honor al físico nacido en Países Bajos, Johannes Diderik van der Waals [43,44], más una función que toma en cuenta las interacciones un poco más intensas y de largo alcance, también llamadas interacciones polares o entre cargas. Para estas interacciones, por lo general se usa el potencial que propuso el físico-matemático británico John Edward Lennard-Jones [45], el cual se escribe como sigue:

$$U(r_{ij}) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \quad 9.5)$$

σ y ε son el diámetro de los átomos y la profundidad del pozo atractivo, respectivamente. La función está compuesta por dos términos, el primero representa la repulsión entre átomos a distancias cortas y el segundo término representa la atracción entre átomos a distancias relativamente largas.

A lo largo de los años, las interacciones de van der Waals han sido modeladas a través del potencial de Lennard-Jones, también hay contribuciones donde el potencial propuesto por Philip McCord Morse [46] o la función que propuso Gustav Mie [47] han sido utilizadas para el mismo propósito. Estas funciones se escriben como sigue:

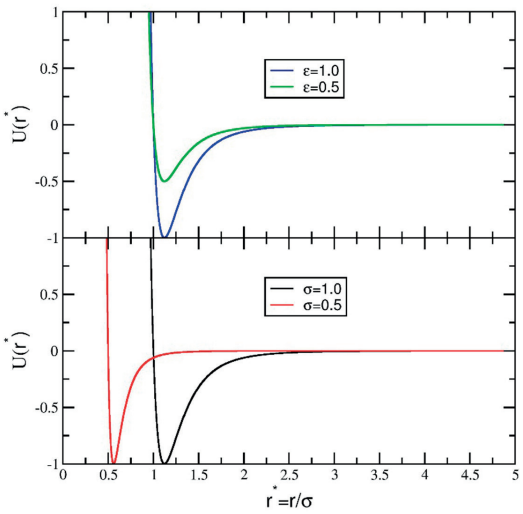
$$U(r) = \varepsilon \left[1 - \exp\left[\frac{-\mathrm{In}(2)}{\sqrt[6]{2}-1}\left(\frac{r}{\sigma} - \sqrt[6]{2}\right)\right]\right]^2 - \varepsilon \quad 9.6)$$

Que es la función de Morse, donde $r=2^{1/6}\sigma$ es la ubicación del mínimo del potencial, y el potencial de Mie es:

$$U(r)=\left(\frac{n}{n-m}\right)\left(\frac{n}{m}\right)^{m/(n-m)}\varepsilon\left[\left(\frac{\sigma}{r}\right)^{n}-\left(\frac{\sigma}{r}\right)^{m}\right] \qquad 9.7)$$

Cuando n=12 y m=6 la ecuación (9.7) se reduce a la ecuación (9.5).

Figura 9.1. Potencial de Lennard-Jones a diferentes profundidades del pozo atractivo



Nota. En la figura de arriba se mantienen los diámetros iguales. La figura de abajo muestra el mismo potencial, pero a diferentes diámetros, manteniendo ε sin cambios.

Acerca de este tipo de función potencial, se dice que los átomos que interactúan mediante la ley de interacción (9.5) conforman un fluido de Lennard-Jones.

Por otro lado, las interacciones entre átomos de diferentes moléculas consideradas de largo alcance, se refieren a la interacción entre sitios cargados; es decir, si los sitios de interacción cuentan con una carga parcial asociada, dicha interacción se modela comúnmente con el potencial propuesto por el físico francés Charles Augustin de

Coulomb [48] —también llamado potencial de Coulomb—, el cual se escribe a continuación:

$$U(r_{ij}q_i) = \frac{1}{4\pi\varepsilon_0}\frac{q_i q_j}{r_{ij}} \tag{9.8}$$

donde ε_0 es la permitividad en el vacio, q_i es la carga parcial del sitio i, y r_{ij} es la distancia de separación entre centros de los átomos i y j. Por lo general, la superposición de los potenciales de Lennard-Jones más Coulomb modelan las interacciones intermoleculares. Las interacciones intramoleculares, que suceden entre átomos de una misma molécula, se modelan a través de funciones armónicas para dos y tres sitios. Para torciones de ángulos diedros, es común usar la fórmula de Rickaert-Bellemans [49]. Una vez definida la ley de interacción entre átomos o moléculas se procede a moverlas. Esto se logra mediante un algoritmo, el cual se obtiene como solución de la segunda ley de Newton. A continuación, se muestran algunos de los más simples algoritmos.

Algoritmo de Verlet

Para construir el algoritmo de Verlet, se hace un desarrollo en serie de la posición al tiempo t+Δt y t-Δt:

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)\Delta t^2 - \frac{1}{6}b(t)\Delta t^3 + 0(\Delta t^4) \tag{9.9}$$

$$\vec{r}(t-\Delta t) = \vec{r}(t) - \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)\Delta t^2 - \frac{1}{6}b(t)\Delta t^3 + 0(\Delta t^4) \tag{9.10}$$

Sumando las ecuaciones de arriba

$$\vec{r}(t+\Delta t) = 2\vec{r}(t) - \vec{r}(t-\Delta t) + \vec{a}(t)\Delta t^2 + 0(\Delta t^4) \tag{9.11}$$

$$\vec{v}(t) = \frac{\vec{r}(t+\Delta t) - \vec{r}(t-\Delta t)}{2\Delta t} \tag{9.12}$$

Dicho algoritmo ofrece una expresión para la posición, que es a su vez solución de la segunda ley de Newton. La velocidad de cada átomo es calculada de manera simple

pero no precisa [41], debido principalmente a que se estima como una diferencia de posiciones calculadas a un tiempo anterior y posterior, dividido entre dos veces el intervalo de tiempo.

Algoritmo leap-frog

Este algoritmo es una variante del anterior [41]. Las posiciones al tiempo t + Δt son expresadas en una expansión en serie:

$$\vec{r}_i(t + \Delta t) = 2\vec{r}_i(t) - \vec{r}_i(t - \Delta t) + \vec{a}_i(t)\Delta t^2 \qquad 9.13)$$

La velocidad es modificada por la fuerza de interacción entre átomos, de modo que:

$$\vec{v}_i\left(t + \frac{\Delta t}{2}\right) = \vec{v}_i\left(t - \frac{\Delta t}{2}\right) + \vec{a}_i(t)\Delta t \qquad 9.14)$$

La velocidad es ahora un promedio de estas, evaluadas a diferente tiempo. Este es un cambio con respecto al algoritmo de Verlet

$$\vec{v}_i(t) = \frac{\vec{v}_i\left(t + \frac{\Delta t}{2}\right) + \vec{v}_i\left(t - \frac{\Delta t}{2}\right)}{2} \qquad 9.15)$$

Combinando las ecuaciones anteriores, hallamos la expresión para las posiciones

$$\vec{r}_i(t + \Delta t) = \vec{r}_i(t) + \vec{v}\left(t + \frac{\Delta t}{2}\right) + \Delta t \qquad 9.16)$$

Este algoritmo es mejor que el original de Verlet, pero no más preciso que un Predictor-Corrector. Sin embargo, es más barato en términos de tiempo de cómputo.

Algoritmo de velocity Verlet

Este algoritmo propone evaluar las posiciones, velocidades y aceleraciones al mismo tiempo. Se evalúan posiciones y velocidad a un tiempo t+Δt

$$\vec{r}_i(t + \Delta t) = \vec{r}_i(t) + \vec{v}_i(t)\Delta t + \frac{1}{2}\vec{a}_i(t)\Delta t^2 \quad 9.17)$$

$$\vec{v}_i(t + \Delta t) = \vec{v}_i(t) + \frac{1}{2}[\vec{a}_i(t) + \vec{a}_i(t + \Delta t)]\Delta t \quad 9.18)$$

Se calcula la velocidad a un tiempo intermedio

$$\vec{v}_i\left(t + \frac{1}{2}\Delta t\right) = \vec{v}_i(t) + \frac{1}{2}\vec{a}_i(t)\Delta t \quad 9.19)$$

El movimiento se complementa al aplicar la aceleración al tiempo t+Δt, de manera que se escribe la velocidad final como:

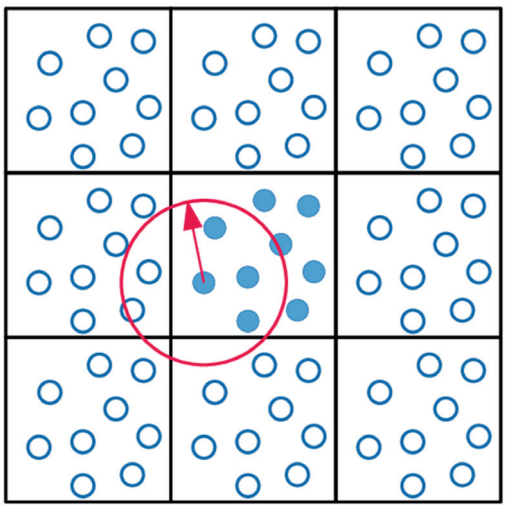
$$\vec{v}_i(t + \Delta t) = \vec{v}_i\left(t + \frac{1}{2}\Delta t\right) + \frac{1}{2}\vec{a}_i(t)\Delta t \quad 9.20)$$

Esta última ecuación permite contar con velocidades al tiempo t+Δt y, así, la energía cinética se estima al mismo tiempo.

Condiciones de frontera periódicas y condición de mínima imagen

Con la intención de simular un sistema real con un conjunto de moléculas representativo, se hace uso de las condiciones de frontera periódicas; la figura 9.2 muestra un ejemplo para el caso 2-dimensional (2D).

Figura 9.2. Condiciones de frontera periódicas



Esta figura es resultado de construir una caja de simulación que contiene N moléculas, y que se coloca al centro en la figura. Luego, se construyen réplicas de la caja de simulación original y se colocan alrededor de la misma. Las cajas replicadas albergan moléculas que son imágenes de las moléculas originales; así, este procedimiento nos permite simular un sistema infinito. Agregado a lo anterior, se muestra la interacción entre las moléculas al interior de un radio de corte (r_{cut}), que está colocado en una molécula cualquiera e indicado con una flecha. Las moléculas que se encuentran al interior de la circunferencia descrita por el radio de corte contribuyen a la energía y fuerza intermolecular del sistema. Si una molécula está fuera de dicha circunferencia, pero su imagen si lo está, entonces se considera la interacción con su imagen. A este procedimiento se le conoce como la convención de mínima imagen.

Particularmente, en un código en Python podemos programar las condiciones de frontera periódicas con las líneas siguientes:

$$rx = rx - np.rint\left(\frac{rx}{Lx}\right) * Lx \qquad 9.21)$$

$$ry = ry - np.rint\left(\frac{ry}{Ly}\right) * Ly \qquad 9.22)$$

$$rz = rz - np.rint\left(\frac{rz}{Lz}\right) * Lz \tag{9.23}$$

Que agregan una distancia Lx, Ly o Lz a las posiciones de los átomos si estos se salen por algún lado de la caja de simulación y, entonces, sus imágenes entran a la misma caja, por el lado contrario de donde salió el átomo inicialmente. La instrucción *np.rint* está definida en la biblioteca *numpy* y actúa como una función escalón:

$$f(x) = \begin{cases} 1, si\ x > 1 \\ 0, si\ x < 1 \end{cases} \tag{9.24}$$

Si $rx / Lx > 1$, significa que el átomo se salió de la caja de simulación.

Si $rx / Lx < 1$, significa que el átomo no se salió de la caja de simulación.

Por lo general, las propiedades termodinámicas dependen del radio de corte usado. Entre más largo es el radio de corte, el cálculo de las propiedades termodinámicas es más preciso; sin embargo, el tiempo de cómputo crece demasiado. Una opción para corregir el cálculo de la presión, la energía potencial y el potencial químico es hacer uso de las llamadas correcciones de largo alcance, que se calculan a partir del radio de corte [41], y que son válidas en una fase homogénea. Otra opción que podemos encontrar en la literatura especializada para evitar la dependencia en el radio de corte, es el uso del método de sumas de Ewald [50], que nos permite calcular la contribución de la energía/fuerza hasta el radio corte en el espacio real; y la contribución complementaria de la energía/fuerza se calcula en el espacio de Fourier, definiendo y usando el correspondiente radio de corte. Dicho método no es el único, pero es el más usado por la comunidad.

Particularmente, solo abordamos el material relacionado con las dos opciones mencionadas: las correcciones de largo alcance y el método de sumas de Ewald.

Correcciones de largo alcance

Se calcula la energía potencial, la presión y el potencial químico durante la simulación molecular, tomando la interacción entre las moléculas hasta el radio de corte r_{cut}

$$u_{total} = u(r \leq r_{cut}) + u_{lrc} \tag{9.25}$$

$$p_{total} = p(r \leq r_{cut}) + p_{lrc} \tag{9.26}$$

$$\varphi_{total} = \varphi(r \leq r_{cut}) + \varphi_{lrc} \tag{9.27}$$

Se complementan estas propiedades termodinámicas con los valores obtenidos a través de las correcciones de largo alcance [38], las cuales se calculan al inicio o al final de la simulación molecular. Para la energía potencial, se muestra la corrección respectiva de largo alcance:

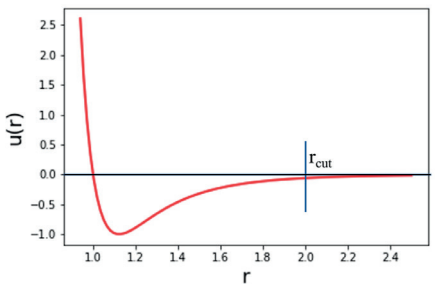
$$u_{lrc} = \frac{1}{2}\rho N \int_{r_{cut}}^{\infty} u(r) 4\pi r^2 dr \tag{9.28}$$

Si usamos como potencial de interacción a la función de Lennard-Jones, encontramos la siguiente expresión:

$$u_{lrc} = \frac{8}{3}\pi\rho N\varepsilon\sigma^3 \left[\frac{1}{3}\left(\frac{\sigma}{r_{cut}}\right)^9 - \left(\frac{\sigma}{r_{cut}}\right)^3\right] \tag{9.29}$$

Se incluye la gráfica del potencial de Lennard-Jones, donde se indica el radio de corte que señala la distancia hasta donde se toma en cuenta la interacción entre átomos.

Figura 9.3. Potencial de Lennard-Jones, indicando un radio de corte



Escribimos la corrección de largo alcance de la energía en unidades reducidas, considerando $\varepsilon^*=1$ y $\sigma^*=1$:

$$u^*_{lrc} = \frac{8}{9}\pi\rho^* N r^{-9}_{cut} - \frac{8}{3}4\pi\rho^* N r^{-3}_{cut} \qquad 9.30)$$

Por otro lado, la corrección de largo alcance para la presión se calcula como se indica a continuación:

$$p_{lrc} = -\rho^2 N \frac{1}{6}\int_{r_{cut}}^{\infty} r\frac{du(r)}{dr}4\pi r^2 dr \qquad 9.31)$$

Nuevamente, si usamos la función de Lennard-Jones como potencial de interacción, hallamos la expresión siguiente:

$$p_{lrc} = \frac{16}{3}\pi\rho^2\varepsilon N\sigma^3\left[\frac{2}{3}\left(\frac{\sigma}{r_{cut}}\right)^9 - \left(\frac{\sigma}{r_{cut}}\right)^3\right] \qquad 9.32)$$

En unidades reducidas, podemos escribir la corrección de largo alcance de la presión como:

$$p^*_{lrc} = \frac{32}{9}\pi\rho^{*2} r^{-9}_{cut} - \frac{16}{3}\pi\rho^{*2} r^{-3}_{cut} \qquad 9.33)$$

La corrección de largo alcance para el potencial químico se estima al llevar a cabo la integral que se muestra enseguida:

$$\varphi_{lrc} = \rho\int_{r_{cut}}^{\infty} u(r)4\pi r^2 dr \qquad 9.34)$$

Posteriormente, hacemos uso del potencial de Lennard-Jones, de manera que hallamos la expresión siguiente:

$$\varphi_{lrc} = \frac{16}{3}\pi\rho\varepsilon\sigma^3\left[\frac{1}{3}\left(\frac{\sigma}{r_{cut}}\right)^9 - \left(\frac{\sigma}{r_{cut}}\right)^3\right] \qquad 9.35)$$

Finalmente, escribimos la corrección de largo alcance del potencial químico en unidades reducidas como:

$$\varphi^*_{lrc} = \frac{16}{9}\pi\rho^* r^{-9}_{cut} - \frac{16}{3}\pi\rho^* r^{-3}_{cut} \qquad 9.36)$$

Método de sumas de Ewald

Las interacciones intermoleculares de largo alcance (que describe el potencial de Coulomb o el potencial entre dipolos permanentes) convergen lentamente; es decir, tienden al cero a distancias relativamente largas. Sin embargo, los potenciales considerados relativamente de corto alcance (como el potencial de Lennard-Jones) requieren también el uso de tal metodología, sobre todo al estimar propiedades termodinámicas en sistemas heterogéneos (dos fases o más). El método de sumas de Ewald fue propuesto por Paul Peter Ewald en 1921 [50], usándolo para calcular la energía de interacción de sistemas periódicos como sistemas sólidos (cristales). Se ha aplicado dicha metodología a diferentes funciones potenciales, como: Lennard-Jones, Yukawa, Coulomb y dipolo-permanente [51-54].

En general, el cálculo de la energía/fuerza total se divide en dos partes: una se calcula hasta el radio de corte en el espacio real; y la complementaria, en el espacio de Fourier. Este procedimiento es preciso, pero no rápido, ya que se tiene que invertir un tiempo considerado en el cálculo de la energía/fuerza en el espacio de Fourier; y para acelerar dicho proceso se recurre al método Particle Mesh Ewald (PME por sus siglas en inglés) que usa la transformada rápida de Fourier [55]. El objetivo es determinar la expresión para la energía potencial, la fuerza y la presión del sistema con base en la metodología propuesta por Paul Ewald.

Básicamente, se considera un potencial de la forma $1/r^p$, siendo p una potencia cualquiera y r la distancia de separación entre átomos,

$$\frac{1}{r^p} = \frac{erfc(vr)}{r^p} + \frac{1-erfc(vr)}{r^p} \qquad 9.37)$$

siendo $erfc(vr)$ la función error complementaria. El segundo término converge lentamente, de manera que se aplica la transformada de Fourier, lo que causa que se acelere solo un poco. En particular, se muestran las sumas de Ewald para la parte atractiva (dispersiva) del potencial de Lennard-Jones:

$$U^6 = \sum_{i=1}^{N-1} \sum_{j>i}^{N} \frac{\lambda_{ij}}{r_{ij}^6} \qquad 9.38)$$

donde $\lambda_{ij} = -\lambda_i\lambda_j$, siendo $\lambda_i = 2\sigma^3\varepsilon^{(1/2)}$. En particular, la energía potencial en el espacio real se escribe como:

$$U^{real} = v^6 \sum_L \sum_{i=1}^{N-1} \sum_{j>i}^{N} \lambda_{ij} \left(a^{-6} + a^{-4} + \frac{a^{-2}}{2} \right) e^{-a^2} \tag{9.39}$$

donde **L** es el vector de red de un arreglo periódico de las celdas imagen, a = $|\mathbf{r}_i - \mathbf{r}_j - \mathbf{R}_L|$n, siendo $\mathbf{r}_i$ la posición de la molécula i, y $\mathbf{R}_L$ el vector de traslación de la red. n es el parámetro de decaimiento y nos indica qué tan rápido converge el término real de la energía.
La contribución de la energía en el espacio de Fourier se escribe como:

$$U^{Four} = -\frac{\pi^{3/2}}{24V} \sum_{h=0} S(h)S(-h)B(h) + \frac{\pi^{3/2}v^3}{6V} \sum_{i=1}^{N}\sum_{i=1}^{N}\lambda_{ij} \tag{9.40}$$

Donde:

$$B(h) = h^3 \left[\sqrt{\pi} erfc(b) + \left(\frac{1}{2b^3} - \frac{1}{b} \right) e^{-b^2} \right] \tag{9.41}$$

siendo b=h/2n, V el volumen de la celda unitaria, **h** el vector de red recíproca (cuya magnitud es h = $|\mathbf{h}|$) y S(**h**) el factor de estructura, cuya expresión es:

$$S(h) = \sum_{j=1}^{N} \lambda_j e^{ih \cdot rj} \tag{9.42}$$

El segundo término de la ecuación (9.29) surge del espacio de Fourier cuando $h=0$. Por último, se debe incluir un término que corrija la autointeracción entre moléculas/átomos. Esto sucede cuando $i=j$ y $L=0$.

$$U^{auto} = -\frac{v^6}{12} \sum_{i=1}^{N} \lambda_{ii} \tag{9.43}$$

Para el caso del potencial de Coulomb en términos de las sumas de Ewald, comenzamos mostrando la contribución a la energía en el espacio real

$$U^1 = \frac{1}{2} \sum_{i=1}^{N} \sum_{i,j}^{N} q_i q_j \frac{erfc(vr_{ij})}{r_{ij}} \tag{9.44}$$

donde q_i es la carga parcial asociada al átomo *i*. Luego, la contribución en el espacio de Fourier es:

$$U^{Four} = \frac{1}{2\pi V} \Sigma_{i,j}^{N} q_i q_j \Sigma_{h=0} \frac{exp\left[-\left(\frac{\pi h}{v}\right) + i2\pi h(r_i - r_j)\right]}{h^2} \quad 9.45)$$

El término de autointeracción es:

$$U^{auto} = -\frac{v}{\sqrt{\pi}} \Sigma_{i=1}^{N} q_i^2 \quad 9.46)$$

En ocasiones, se incluye un término de corrección considerando un arreglo esférico de celdas imágenes rodeado por un continuo, representado por una constante dieléctrica arbitraria (ε') que generalmente se toma como 1.

$$U^{superf} = \frac{2\pi}{(1+2\varepsilon')V} \left|\Sigma_{i=1}^{N} q_i r_i\right|^2 \quad 9.47)$$

El último término se interpreta como la energía que surge de la esfera que interactúa con la superficie del material. Cuando se utiliza un material dieléctrico como el vacío, el denominador de (1+2ε') se reemplaza con la cantidad 3. Por último, las expresiones de la fuerza y la presión de ambos potenciales se pueden encontrar en la referencia [52].

Propiedades termodinámicas y de estructura

A través de teoremas de la Mecánica Estadística [56-58] podemos pasar de una descripción microscópica a una macroscópica; es decir, de usar posiciones y velocidades a propiedades termodinámicas. Haciendo uso del teorema de la equipartición de la energía, obtenemos la temperatura de una configuración a través del promedio de la energía cinética. Dicho teorema indica que cada grado de libertad contribuye con ½ de $k_B T$, de manera que escribimos la relación:

$$\langle E_k \rangle = \frac{f}{2} k_B T \quad 9.48)$$

donde *f* son los grados de libertad del sistema, $<E_k>$ es el promedio de la energía cinética, k_B es la constante de Boltzmann y T es la temperatura calculada. En un código de simulación es común calcular la temperatura instantánea a través de la expresión:

$$T = \frac{2(E_K)}{k_B(fN-3)} \qquad 9.49)$$

donde *N* es el número de átomos.

Por otra parte, para calcular la presión debemos mencionar que se obtiene a través de sus dos contribuciones: la parte cinética y la parte de las interacciones entre átomos o moléculas. La expresión virial nos permite escribir:

$$VP_{\alpha,\beta} = \Sigma_{i=1}^{N} m_i (\vec{v}_i)_\alpha (\vec{v}_i)_\beta + \Sigma_{i=1}^{N} \Sigma_{j>i}^{N-1} \vec{F}_{ij} \cdot \vec{r}_{ij} \frac{\vec{r}_{ij}}{r_{ij}} \qquad 9.50)$$

donde V es volumen que contiene al sistema, v_i y m_i son la velocidad y la masa de la partícula *i, N* es el número de átomos, F_{ij} es la fuerza de interacción entre los átomos *i* y *j*, r_{ij} es la distancia entre los mismos átomos y *r* su magnitud.

Función de distribución radial

La función de distribución radial [41] *g*(*r*) es una propiedad de estructura que nos da información de cómo se acomodan espacialmente las moléculas/átomos en una simulación a nivel molecular. La variable *r* nos indica la distancia a partir de una molécula/átomo de referencia, y la función *g*(*r*) la probabilidad de que las moléculas/átomos se encuentren a una distancia r en comparación con el gas ideal. La función de distribución radial, que también denotamos como rdf, se calcula en dos pasos. El primero de ellos consiste en estimar los átomos que caen en un elemento de volumen con simetría esférica; es decir, la caja de simulación se ve ahora como una "cebolla" constituida por capas del mismo espesor, y en cada una de las capas se contabiliza el número de átomos en promedio, N(r, r+Δr). El segundo consiste en calcular el volumen de cada capa $V_{capa}=(4/3)\pi\rho[r^3, (r+\Delta r)^3]$ y dividir ambas expresiones:

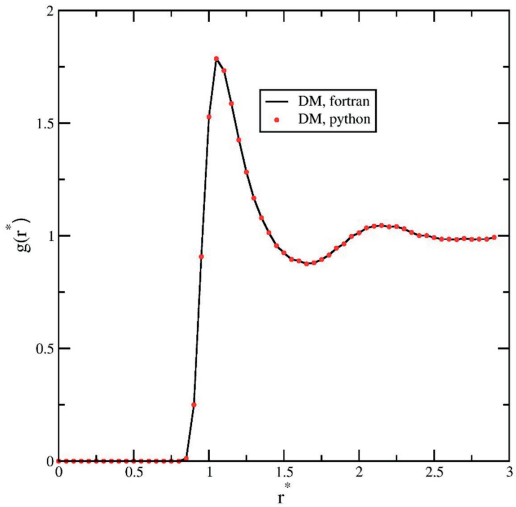
$$g(r) = \frac{N(r, r+\Delta r)}{\frac{4}{3}\pi\rho[r^3, (r+\Delta r)^3]} \qquad 9.51)$$

A continuación, en un seudocódigo se indica, de manera general, cómo es el cálculo de la función de distribución radial mediante un código de simulación escrito en Fortran, solo se indica dónde se debe hacer el llamado de las dos subrutinas involucradas: una de ellas calcula el número de átomos N(r+Δr) en cada elemento de volumen cada *ngr* pasos de simulación; la segunda, el volumen de cada división (capa). Además, en esta última subrutina se importa el valor numérico de N(r+Δr) y se calcula de manera completa la función de distribución radial.

```
#9.2
C     Se calcula N(r+Δr)
      do i = 1, nstep
...
        if(mod(istep,ngr).eq.0) then
          call nrdf
        endif
      enddo
...
C     Se calcula la RDF
      nstepg = nstep/ngr
      if(nstepg.gt.0) then
          call rdf(nstepg)
      endif
```

En el apéndice A se incluye el código en Python de dinámica molecular, que simula un fluido de Lennard-Jones y calcula la función de distribución radial. Los datos se guardan en el archivo *rdf.dat*, y se pueden graficar con el software *grace* o cualquier otro. En particular, se mueven 108 átomos a una densidad reducida de 0.5 (de hecho, todas las variables son reducidas o sin dimensiones). La temperatura se fija a 2, el tiempo de paso es 0.005 y el radio de corte es 2.5. En la figura 9.4 se muestra la función de distribución por pares de un fluido Lennard-Jones, usando un código en Fortran y otro más en Python; ambos dan los mismos resultados.

Figura 9.4. Función de distribución radial de un fluido Lennard-Jones, ρ*=0.5, T*=2



Lammps

El software libre llamado Large-scale Atomic/Molecular Massively Parallel Simulator [59], se identifica comúnmente por sus siglas LAMMPS. Dicho software nos permite llevar a cabo simulaciones de dinámica molecular de manera rápida debido a que está paralelizado y puede hacer uso de Unidades de Procesamiento Gráfico (GPU por sus siglas en inglés). Es común observar que los archivos que contienen los datos de salida de las simulaciones están en un formato que no es inmediato manejar para hacer promedios, por lo que es conveniente construir códigos en Python para extraer la información de interés en un orden específico.

Para hacer una simulación molecular con LAMMPS es necesario colectar una serie de instrucciones 9.3, que sirven como parámetros de entrada (input). De hecho, el usuario interesado puede consultar el manual ubicado en el sitio oficial. Un ejemplo sencillo es el cálculo de la función de distribución radial de un fluido de Lennard-Jones. A continuación se muestran las instrucciones necesarias.

```
#9.3
# instrucciones para calcular la función
#de distribución radial de un fluido
Lennard-jones

variable        x equal 5
variable        y equal 5
variable        z equal 5
variable        rho equal 0.8
variable        t equal 4.0
variable        rc equal 4.0
variable        dt equal 0.001

units           lj
atom_style      atomic
lattice         fcc ${rho}
region          box block 0 $x 0 $y 0 $z
create_box      1 box
create_atoms    1 box

mass            1 1.0
group           1  type 1
velocity        all create $t 97287

pair_style      lj/cut ${rc}
pair_coeff      1 1 1.0  1.0  ${rc}

neighbor        0.3 bin
neigh_modify    delay 0 every 20 check no
compute         RDF all rdf 100 1 1
fix             1 all nvt temp $t $t 0.5
thermo_style    custom step temp density pe ke
etotal press
fix             22 all ave/time 10 1 100
c_RDF[*] file rdfLJ.dat mode vector
dump            91 all custom 100 dump.lj id
type x y z

thermo          1000
timestep        ${dt}
run             20000
velocity        all scale $t
write_data      continua-rdf.dat
unfix           1
undump          91
print           "     "
print           "     "
print           "calcula la rdf  "
print           "      "
```

El archivo *dump.lj*, generado en esta corrida, contiene una animación del sistema compuesto por N átomos, que interactúan mediante el potencial de Lennard-Jones. De acuerdo con el formato de este mismo archivo, el visualizador *Ovito* [60] lo puede reproducir sin mayor problema. Además, se genera el archivo *rdfLJ.dat*, que contiene varias funciones de distribución radial correspondientes a diferentes configuraciones; lo que sigue es calcular el promedio de ellas. A continuación, se explica cómo calcular dicho promedio. Primero, se verifica el orden y el formato de la información contenida en el archivo *rdfLJ.dat*. Después, se identifica cuántas líneas dan información de los datos, pero no forman parte de la función de distribución radial. Finalmente, se identifica cuántos renglones contienen información que conforma una sola foto de la función de distribución radial. Con esta información en mente es posible escribir

un procedimiento para extraer la información requerida y, posteriormente, hacer un promedio de las fotos (en otras palabras, se hace un promedio de las distintas *rdfs* almacenadas en el archivo de trabajo). A continuación, se incluye el procedimiento usado para tal fin.

```
#código 9.4
import sys

nSaltos=0;
nDatos =0;
nFotos =1;
nCont  =0;
strIn=" ";

if(len(sys.argv)==0):
   print("Indique los siguientes datos
con las directivas indicadas:\nNúmero de
Saltos inicial (s:)\nNúmero de datos por
foto(n:)\nArchivo fuente de datos (f:)\n")
else:
   for i in range(len(sys.argv)):
      strPar=sys.argv[i][0:2];
      if(strPar=='s:'):
         nSaltos=int(sys.argv[i][2:]);
      elif(strPar=='n:'):
         nDatos=int(sys.argv[i][2:]);
      elif(strPar=='f:'):
         strIn=sys.argv[i][2:];
   if(nSaltos==0 or nDatos==0 or
strIn==""):
      print("Indique los siguientes datos
con las directivas indicadas:\nNúmero de
Saltos inicial (s:)\nNúmero de datos por
foto(n:)\nArchivo fuente de datos(f:)\n")
   else:
      print("Número de saltos iniciales:"
+ str(nSaltos) + "\nNúmero de datos por
foto:" + str(nDatos) + "\nArchivo fuen-
te:"+strIn);

      fAcum1 = [];
      fAcum2 = [];
      fAcum3 = [];
      fAcum4 = [];

      for x in range(nDatos):
         fAcum1.append(0.0);
         fAcum2.append(0.0);
         fAcum3.append(0.0);
         fAcum4.append(0.0);
      archivo = open(strIn,'r')
      for i in range(nSaltos):
         linea = archivo.readline()
      while True:
         nCont+=1;
         linea = archivo.readline()
         if not linea:
            break
         if( nCont % (nDatos+1) == 0):
            print("\rFoto " + str(nFotos) + "
agregada." , end='');
            nFotos+=1;
            nCont=0;
         else:
            linea= linea.strip();
            li = list(linea.split(" "));
            fAcum1[nCont-1]+=float(li[0]);
            fAcum2[nCont-1]+=float(li[1]);
            fAcum3[nCont-1]+=float(li[2]);
            fAcum4[nCont-1]+=float(li[3]);
      archivo.close
      print("\rFoto " + str(nFotos) + " agregada." ,
end="");
      archivo = open("promedioOut.dat",'w')
      for i in range(nDatos):
         fAcum1[i]/=float(nFotos);
         fAcum2[i]/=float(nFotos);
         fAcum3[i]/=float(nFotos);
         fAcum4[i]/=float(nFotos);
         archivo.write( str(fAcum2[i]) +  ' ' +
str(fAcum3[i]) + '\n');
      archivo.close;
      print("\nArchivo promedioOut.dat
generado");
```

Los datos contenidos en el archivo *rdfLJ.dat* tienen la siguiente estructura:

```
#9.5
# Time-averaged data for fix 11
# TimeStep Number-of-rows
# Row c_RDF[1]
0 100
1 0.02 0 0
2 0.06 0 0
3 0.1 0 0
4 0.14 0 0
5 0.18 0 0
...                                  datos intermedios
95 3.78 0 176
96 3.82 4.091 200
97 3.86 0 200
98 3.9 0 200
99 3.94 0 200
100 3.98 0 200
100 100
1 0.02 0 0
2 0.06 0 0
3 0.1 0 0
4 0.14 0 0
5 0.18 0 0
...                                  datos intermedios
95 3.78 0.982361 183.293
96 3.82 0.942805 188.824
97 3.86 0.922531 194.35
98 3.9 0.872796 199.687
99 3.94 0.852761 205.009
100 3.98 0.844971 210.39
200 100
```

Identificamos que la primera columna solo nos da el número de renglón que contiene datos útiles, la segunda corresponde a la variable independiente *r*, y la tercera es el

dato de la *rdf*. Para hacer el promedio de las diferentes *rdf*s se leen cuatro renglones no útiles para el promedio, luego, cien renglones de datos (*r,rdf*) y un renglón no útil. Posteriormente, el inmediato bloque de datos de la siguiente *rdf*, y así sucesivamente hasta terminar de leer los datos. Después de correr el código en Python se genera el archivo *promedioOut.dat,* que contiene el promedio de las diferentes *rdfs*.

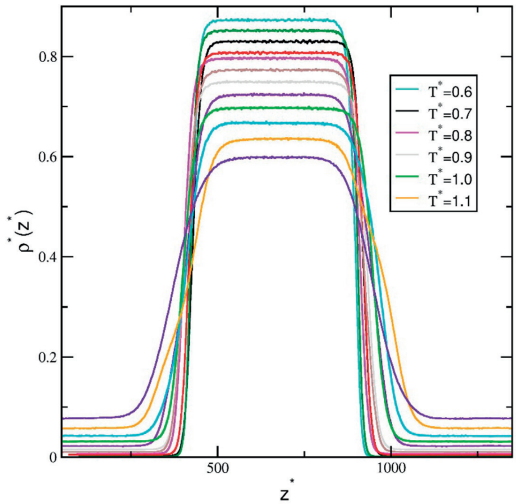
Perfil de densidad y tensión superficial

En caso de que se simule un sistema heterogéneo y se desee calcular las densidades del equilibrio líquido-vapor, es necesario usar una celda de simulación paralelepípeda. En particular, las densidades del equilibrio líquido-vapor se calculan a través de la ecuación siguiente [41]:

$$\rho(z) = \frac{1}{2}(\rho_{liq} + \rho_{vap}) - \frac{1}{2}(\rho_{liq} - \rho_{vap}) \tanh\left(\frac{[z - z_0]}{\delta/2}\right) \qquad 9.52)$$

donde ρ(z) es el perfil de densidad calculada a través de la dinámica molecular; r_{vap} y r_{liq} son las densidades de equilibrio del vapor y del líquido, respectivamente; z_0 es la posición de la división de Gibbs; y δ/2 es el ancho de la interfase. Se inicializan los parámetros a determinar (r_{vap}, r_{liq}, z_0 y δ), igualándolos a la unidad en el primer proceso de ajuste. Este proceso se repite hasta alcanzar un error relativo 0.001. El procedimiento interactivo es llamado cuasi-newtoniano, que desarrollaron varios autores [61]. El perfil de densidad se calcula durante la simulación molecular para cada temperatura fijada. En la figura 9.5 se muestran diferentes perfiles de densidad para fluidos Lennard-Jones. Líneas abajo se incluye el input de LAMMPS para calcular las densidades del equilibrio líquido-vapor de un fluido Lennard-Jones.

Figura 9.5. Perfil de densidad de un fluido Lennard-Jones a diferentes temperaturas. De arriba hacia abajo la temperatura varía de 0.6 a 1.15.



```
#9.6
# LVE for Lennard-Jones fluid
variable        l  equal 8
variable        rc equal 2.5
variable        T  equal 0.7

units           lj
atom_style      atomic
region          box block 0.0  ${l}  0.0
${l}  0.0  ${l}
create_box      1 box
change_box      all z final 0.0  12.0
create_atoms    1 random 1000 9879 box
group           1 type 1
mass            1 1.0

replicate       1 1 3
group           3 id <> 1      1000
group           3 id <> 2001   3000
delete_atoms    group 3

neigh_modify    delay 3
velocity        all create $T 87287 mom
yes rot yes dist gaussian

pair_style      lj/cut ${rc}
pair_coeff      1 1 1.0 1.0 ${rc}

compute         tempi all temp
compute         7 all pressure tempi

fix             2 all nvt temp $T $T 0.05
thermo          2000
timestep        0.005
variable        xPress equal c_thermo_press[1]
variable        yPress equal c_thermo_press[2]
variable        zPress equal c_thermo_press[3]
variable        st equal 0.5*lz*(v_zPress-     0.5*
(v_xPress+v_yPress))
fix             3 all ave/time 10 1 100 c_7[3] file
Pv.dat
fix             4 all ave/time 10 1 100 v_st file
SurfTension.dat
compute         denz 1 chunk/atom bin/1d z center
0.1 units box
fix             51 ave/chunk 10 1 1000 denz
density/mass file DensProf.dat
run             200000

unfix           2
unfix           3
unfix           4
unfix           51
write_data      lve-continua.dat
```

Una vez concluida la simulación con LAMMPS, se generan los archivos *SurfTension.dat*, *Pv.dat* y *DensProf.dat,* que contienen datos de la tensión superficial, presión de vapor y perfiles de densidad de distintas configuraciones, por lo que es necesario hacer promedios. Las dos primeras propiedades termodinámicas se calculan sin mayor dificultad; sin embargo, el promedio de perfiles de densidad representa un desafío de cómo contar. Pongamos por ejemplo el archivo *DensProf.dat*, que contiene datos de m-perfiles de un fluido Lennard-Jones. Para extraer los datos del archivo *DensProf.dat* en forma correcta y hacer un promedio con ellos, se sigue el mismo procedimiento en Python que para el caso de la función de distribución radial. En el recuadro

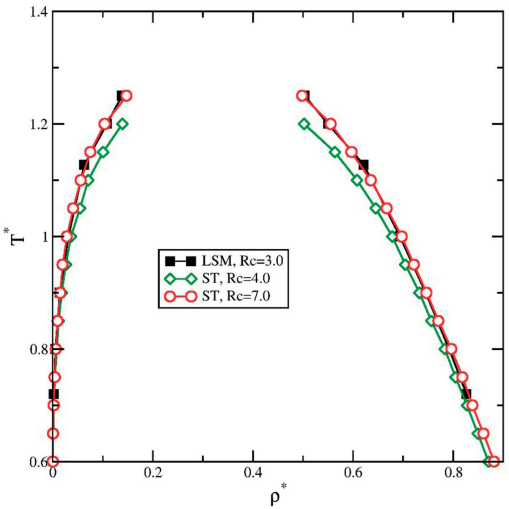
siguiente solo se muestran los datos de dos perfiles de densidad. La primera columna corresponde a un número entero que identifica el dato útil para hacer los promedios, la segunda, la variable independiente, y la cuarta el perfil de densidad.

```
#9.7
# Chunk-averaged data for fix 34 and group file
# Timestep Number-of-chunks Total-count
# Chunk Coord1 Ncount density/mass
0 690 4000
  1 0.05 0 0
  2 0.15 0 0
  3 0.25 0 0
  4 0.35 0 0
  5 0.45 0 0
...                                  datos intermedios
  685 68.45 0 0
  686 68.55 0 0
  687 68.65 0 0
  688 68.75 0 0
  689 68.85 0 0
  690 68.95 0 0
100 690 4000
  1 0.05 0 0
  2 0.15 0 0
  3 0.25 0 0
  4 0.35 0 0
  5 0.45 0 0
...                                  datos intermedios
  685 68.45 0 0
  686 68.55 0 0
  687 68.65 0 0
  688 68.75 0 0
  689 68.85 0 0
  690 68.95 0 0
200 690 4000
  1 0.05 0 0
  2 0.15 0 0
  3 0.25 0 0
  4 0.35 0 0
  5 0.45 0 0
```

Es necesario extraer los datos de la segunda y cuarta columnas en cada foto. La primera foto del perfil de densidad comienza en el renglón identificado con el número 1 en la primera columna, y termina en 690 de la misma columna; después hay un renglón que no es útil para el promedio. La segunda foto del perfil de densidad comienza en el renglón que se identifica con el número 1 de la primera columna, pero representa el renglón 966, y así sucesivamente. Para hacer el promedio de perfiles de densidad, usamos el mismo procedimiento en Python que utilizamos para el caso de la *rdf*. Este procedimiento requiere los siguientes datos de entrada: el número de renglón donde se comienzan a leer los datos de las columnas, el número de datos por foto y el nombre del archivo que contiene los datos (*DensProf.dat*).

En la figura 9.6 se observan las densidades del equilibrio líquido-vapor de un fluido de Lennard-Jones, usando la aproximación esféricamente truncada (ST por sus siglas en inglés) y el método de sumas de Ewald (LSM por sus siglas en inglés).

Figura 9.6. Densidades del equilibrio líquido-vapor de un fluido Lennard-Jones



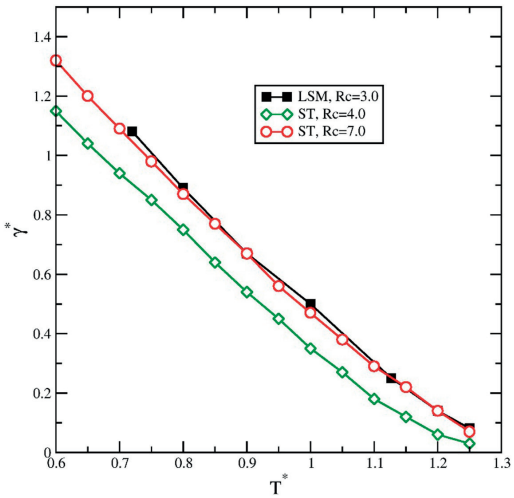
Nota. Los cuadros llenos son resultados tomados de [51]. Los diamantes verdes vacíos y los círculos rojos vacíos son resultados obtenidos usando un radio de corte de 4.0 y 7.0, respectivamente.

Como ya se ha mencionado, otra de las propiedades interfaciales de interés y que se pueden calcular con el mismo script, es la tensión superficial, cuya definición mecánica es la siguiente:

$$\gamma = \frac{L_z}{2}\left[\langle P_{zz}\rangle - \frac{1}{2}\langle P_{xx} + P_{yy}\rangle\right] \qquad 9.53)$$

donde L_z es el lado más largo de la caja de simulación, la cual es un paralelepípedo. P_{ii} son las componentes de la diagonal de la presión (recordar que la presión tiene una representación matricial 3x3). Hacemos uso de la expresión virial para hallar dichas componentes (ver ecuación 9.50). En la figura 9.7 se muestra la tensión superficial de un fluido Lennard-Jones, calculada mediante la aproximación ST, y se incluyen los datos reportados en [51], obtenidos usando LSM.

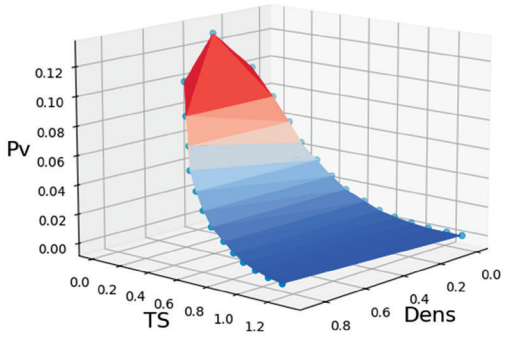
Figura 9.7. Tensión superficial de un fluido Lennard-Jones



Nota. Los cuadros llenos son resultados tomados de (51). Los diamantes verdes vacíos y los círculos rojos vacíos son resultados obtenidos usando un radio de corte de 4.0 y 7.0, respectivamente.

En la figura 9.8 se construye una gráfica en 3D de los datos obtenidos para las densidades del equilibrio líquido-vapor (densidades ortobáricas), tensión superficial y presión de vapor de un fluido de Lennard-Jones, usando un radio de corte de 7σ.

Figura 9.8. Densidades líquido-vapor (Dens), tensión superficial (TS) y presión de vapor (Pv) del fluido Lennard-Jones



Los datos de la densidad y presión críticas fueron tomados de las referencias [62,63]. Sabemos que la tensión superficial se anula cuando la temperatura alcanza su valor crítico. Esta información se observa en la figura 9.8. El procedimiento en Python para generar la gráfica en 3D hace uso de las librerías *mpl_toolkits* y *matplotlib*. Para tal propósito, primero se debe generar el archivo *datos.dat,* que contiene los datos de las densidades, la presión de vapor y la tensión superficial. Dicho procedimiento se incluye a continuación.

```
#código 9.8
from mpl_toolkits import mplot3d
import numpy as np
import sys
import math
import pandas as pd
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from matplotlib import pyplot as plt

data=pd.read_csv('datos.dat',header=0,delim_whitespace=True)
print("data",data)
x=data.iloc[:,0]
y=data.iloc[:,1]
z=data.iloc[:,2]

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.tricontour(x, y, z, colors = 'k')
ax.scatter(x, y, z )
ax.set_title('dens-ST-Pzz')
ax.set_xlabel('dens')
ax.set_ylabel('ST')
ax.set_zlabel('Pzz')

plt.savefig("output.png")
surf=ax.plot_trisurf(x, y, z, cmap=cm.coolwarm, edge-
color='none')
ax.plot_trisurf(x, y, z, cmap=cm.coolwarm, edgecolor='none')
plt.tight_layout()
plt.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

Ejercicios

[1] Calcular la función de distribución radial de un fluido de Lennard-Jones a $T^*=2$, y densidades $\rho^*=0.1$, 0.5 y 0.9

[2] Calcular las densidades de coexistencia de un fluido de Lennard-Jones, usando un radio de corte de 2.5.

[3] Calcular la tensión superficial de un fluido de Lennard-Jones, usando un radio de corte de 2.5.

10. MONTE CARLO

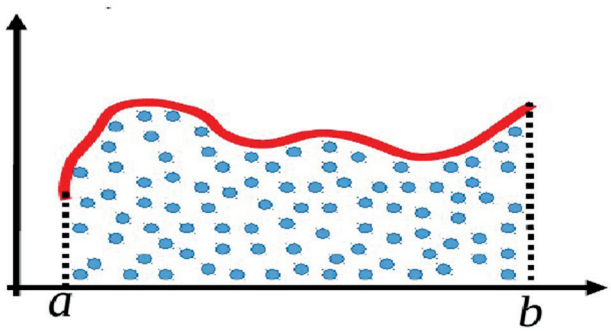
El método directo de Monte Carlo [41,64] es un método numérico que sirve para calcular observables que ayudan a analizar un sistema de interés, está basado en el muestreo estadístico, una metodología que genera datos que caen dentro del conjunto del dominio y, con ello, es posible acercarse al valor real de la observable estimada. Este procedimiento requiere de generar números aleatorios. Ahora bien, los valores que se obtienen de las observables de interés están restringidos a las distribuciones de probabilidad imperantes en el sistema. Este procedimiento es identificado como un proceso del matemático ruso Andréi Márkov [65,66], pues para dar el paso siguiente, solo depende del paso anterior; ha sido usado para resolver problemas matemáticos, físicos y químicos. El nombre de la técnica de simulación se toma del casino Montecarlo, ubicado en el Principado de Mónaco, debido a que podemos hallar sucesos aleatorios en la ruleta rusa. Lo usaremos para analizar fluidos sujetos a diferentes condiciones termodinámicas, que podemos modelar al fijar un ensamble en particular. El concepto de ensamble propuesto por Joshua Gibbs es el que habremos de utilizar. De manera general, modelaremos los fluidos de interés como sistemas constituidos por átomos o moléculas. El método muestrea la evolución espacial de dichos átomos, moviéndolos de manera aleatoria y determinando el valor más probable de la observable de interés.

Se modela la interacción entre átomos o moléculas a través de potenciales de interacción efectivos para tomar en cuenta interacciones inter e intramoleculares: la primera toma en cuenta la interacción entre átomos de distintas moléculas y la segunda considera interacciones entre átomos de una misma molécula. La energía configuración total de N átomos o moléculas toma la expresión:

$$\langle U(r^N)\rangle = \frac{\Sigma\, U_i(r^N)\exp[-\beta U_i(r^N)]}{\Sigma \exp[-\beta U_i(r^N)]} \qquad 10.1)$$

donde $\beta=1/k_BT$, siendo k_B la constante de Boltzmann y T la temperatura dada. Aquí se presenta el problema de que el denominador no puede ser evaluado analíticamente; sin embargo, existe otra opción, que es la que ofrece el muestreo.

Figura 10.1. Integral de una función de N-variables



Una forma de resolver la integral es generar configuraciones que hagan la mayor contribución a dicha integral, ya que no todas tienen relacionadas energías físicas (demasiado altas). Para ello se recurre al llamado método de Metrópolis, que básicamente genera configuraciones adecuadas con la probabilidad:

$$\exp(-\beta U(r^N)) \qquad 10.2)$$

El método de Metrópolis genera una cadena de Markov, donde el resultado depende del paso anterior inmediato, pero no de todos los pasos anteriores. Lo que hace la técnica de simulación MC es escoger aleatoriamente un átomo y calcular su energía configuracional respecto de las demás U_n, como primer paso. Después de elegir un átomo al azar, este se mueve aleatoriamente con un desplazamiento espacial, Δr = *deltaR*. En particular, los números aleatorios se generan con la instrucción *random()* en Python. Si *rand=random(),* entonces podemos escribir el desplazamiento en las tres coordenadas espaciales como:

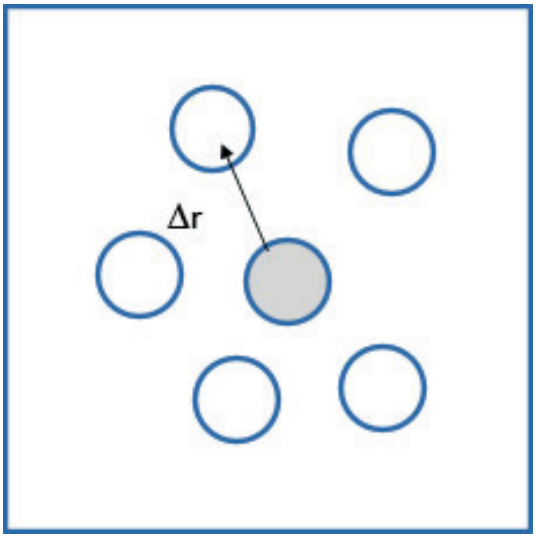
$$rxinew = rxiold + (2 * rand - 1) + deltaR \qquad 10.3)$$

$$ryinew = ryiold + (2 * rand - 1) + deltaR \quad 10.4)$$

$$rzinew = rziold + (2 * rand - 1) + deltaR \quad 10.5)$$

rxiold es la posición anterior en el eje *x* del átomo *i,* y *rxinew* es la posición nueva del mismo átomo en el eje x, moviéndola a un desplazamiento *deltaR.*

Figura 10.2. Muestra el desplazamiento Δr entre átomos en una celda de simulación



Después de mover dicha partícula aleatoriamente, se vuelve a calcular la energía configuracional, Um. Dicho movimiento se acepta si la energía nueva es menor que la energía configuracional antes del movimiento ($\Delta U_{nm} = U_n - U_m < 0$), o con mayor energía, pero con cierta probabilidad que depende del factor de Boltzmann.

La razón de probabilidad de pasar del estado n al estado m es:

$$\frac{P_n}{P_m} = e^{-\beta \Delta U_{nm}} \quad 10.6)$$

con $\beta=1/Tk_B$. Para aceptar el movimiento se genera un número aleatorio de 0 a 1, y se compara con el factor de Boltzmann. Si aquél es menor que $\exp(-\beta\Delta U_{nm})$, se acepta el movimiento. ΔU_{nm} es la diferencia de energía entre dos estados n y m.

El criterio de Metrópolis es uno de los más usados y efectivos para distinguir qué movimientos de los átomos son aceptados o no. A continuación, se escriben algunas líneas de código en Python que describen de manera esquemática tal criterio.

```
deltaU = Unew-Uold
deltaP = Pnew-Pold
deltaUb = beta*deltaU

if(deltaU<0.0):
    U = U + deltaU
    P = P + deltaP
    posix = xinew
    posiy = yinew
    posiz = zinew
    aceptacion = aceptacion + 1
else:
    rand=random()
    if(exp(-deltaUb)>rand):
      U = U + deltaU
      P = P + deltaP
      posix = xinew
      posiy = yinew
      posiz = zinew
      aceptacion = aceptacion + 1
```

Si la energía potencial de la nueva configuración es menor que la energía del paso anterior, entonces se acepta el movimiento de las moléculas/átomos.

Si un número aleatorio es menor que el factor de Boltzmann entonces se acepta el movimiento.

beta = $1/(k_B Temp)$. Es el inverso de la temperatura (Temp) multiplicado por la constante de Boltzmann (k_B). La variable *aceptación* da información del número de movimientos aceptados, que cumplen con el criterio del método de Metrópolis.

Ensamble canónico

El concepto de ensamble lo introdujo Joshua Gibbs. Representa la forma en que se distribuye un conjunto de sistemas de acuerdo con las condiciones termodinámicas impuestas [56,57,58]. En particular, en el desarrollo de un código en Python, siguiendo la técnica de simulación del método directo de Monte Carlo, el ensamble canónico se refiere a mantener constantes el número de átomos, el volumen y la temperatura.

A continuación, se muestra, de forma simple, la densidad de distribución representativa del sistema. Se considera un sistema no aislado con energía U_1 y número de moléculas N_1 en equilibrio térmico, con un sistema más grande (baño térmico) con energía U_2 y número de moléculas N_2. Se supone $U_2 >> U_1$ y $N_2 >> N_1$. Además, se satisface la relación:

$$U < (U_1 + U_2) < U + \Delta U \tag{10.7}$$

La probabilidad de hallar al sistema 1 en dp_1dq_1, que es el elemento de volumen en el espacio de coordenadas generalizadas: posiciones y momentos, sin importar el estado del sistema más grande, es proporcional a $dp_1dq_1W_2(U_2)$, siendo W_2 el volumen ocupado por el sistema 2, así que

$$\rho(p^1, q^1) \propto \Omega_2(U - U_1) \tag{10.8}$$

Si se considera ahora $U>>U_1$, podemos hacer un desarrollo en series de la entropía del sistema 2, de modo que

$$S_2(U) = k_B \log \Omega_2(U - U_1) \tag{10.9}$$

Con lo cual se puede obtener:

$$k_B \log \Omega_2(U - U_1) \approx S_2(U) - \frac{U_1}{T} \tag{10.10}$$

De donde

$$\Omega_2(U - U_1) \approx \exp\left[\frac{1}{k_B} S_2(U)\right] \exp\left(\frac{-U_1}{k_B T}\right) \quad 10.11)$$

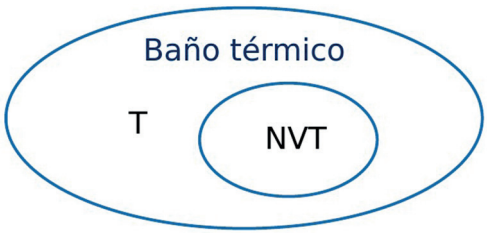
El primer factor en esta ecuación es una constante, lo que nos lleva a la expresión siguiente:

$$\rho(p, q) = e^{-U_1(p,q)/k_B T} \quad 10.12)$$

donde ρ es la densidad de distribución del ensamble; tal que el número de moléculas, el volumen y la temperatura se mantienen constantes (Figura 10.3). Al llevar a cabo simulaciones a nivel molecular, el criterio de aceptación del movimiento de los átomos/moléculas es directamente proporcional al factor de Boltzmann, de manera que la probabilidad de pasar de un estado n a otro estado m se escribe como:

$$\frac{Pnm}{Pmn} = e^{(-\beta \Delta U_{nm})} \quad 10.13)$$

Figura 10.3. Ensamble canónico, NVT constantes



En una simulación molecular se verifica la aceptación o rechazo del movimiento al azar de las partículas. Si la energía nueva es menor se acepta el movimiento; de lo contrario, se compara el factor de Boltzmann con un número aleatorio generado entre 0 y 1. En el apéndice B se incluye el código en Python que contiene el procedimiento del método de Monte Carlo para simular un fluido de Lennard-Jones en un ensamble canónico a las condiciones termodinámicas reducidas: T*=2, ρ*=0.5 y N=108.

Ensamble isotérmico-isobárico

Ahora se considera el número de partículas, la presión y la temperatura constantes, NPT

$$\rho \approx e^{(U+pV)/k_B T} \qquad 10.14)$$

La densidad de probabilidad para un ensamble isotérmico-isobárico es:

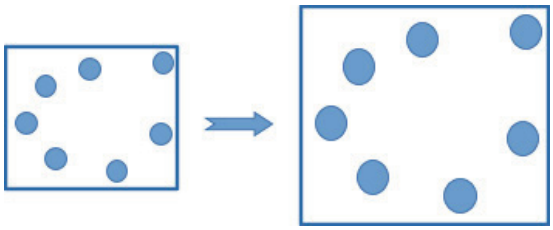
$$Q_{NPT} \approx \frac{1}{N!h^{3N}V_0} \int dV \int dpdq \exp[-\beta(U + PV)] \qquad 10.15)$$

Si se considera un sistema de N partículas, la función de partición estará dada por

$$Q_{NPT} \approx \exp[-\beta G(N, P, T)] \qquad 10.16)$$

donde la energía libre de Gibbs G(N,P,T) resulta ser el potencial termodinámico importante. En este tipo de ensamble se mantiene fija la temperatura, el número de partículas totales y la presión. En forma práctica, para mantener constante la presión en una simulación, se permiten variaciones de las dimensiones de la caja de simulación, pero también se reescalan las posiciones de las partículas.

Figura 10.4. Redimensión de las posiciones en un cambio de volumen



El criterio de aceptación para los movimientos en el escalamiento del volumen es impuesto por el balance detallado, llegando a la ecuación

$$\frac{Pnm}{Pmn} = \left(\frac{Vm}{Vn}\right)^{N} e^{(-\beta U - \beta \Delta PV)} \qquad 10.17)$$

Entonces, durante una simulación por computadora, primero se verifica la aceptación o rechazo del movimiento al azar de las partículas, que cumple con el criterio de Metrópolis para el ensamble canónico; y después se verifica la aceptación o rechazo de la variación en el volumen de la caja de simulación, con la intención de mantener fija la presión del sistema.

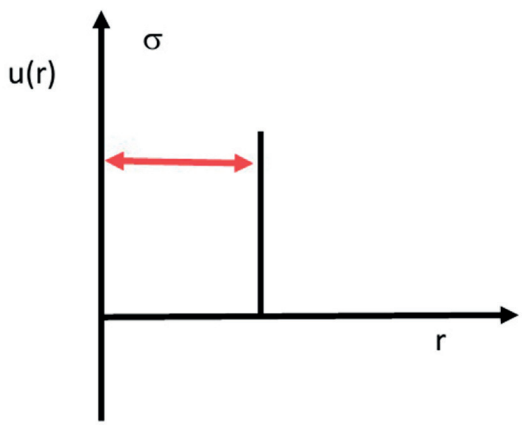
Presión de un fluido de esferas duras

Como ejemplo en el uso de la técnica de simulación MC, calculamos la presión hidrostática de un sistema compuesto por esferas duras; es decir, que la repulsión entre átomos que están en contacto es infinita y que no sienten atracción entre ellas. El potencial de interacción se escribe como:

$$u(r) = \begin{cases} \infty, \; si \; r < \sigma \\ 0, \; si \; r \geq \lambda\sigma \end{cases} \qquad 10.18)$$

donde σ es el diámetro efectivo de los átomos y λ es la distancia. El gráfico que corresponde al potencial de interacción de esferas duras es:

Figura 10.5. Potencial de esfera dura



El método MC usa de manera directa funciones discontinuas. Ahora, si hacemos uso de la expresión virial de la presión, entonces podemos calcular las componentes de la presión

$$P_{zz} = \rho k_B T + \frac{1}{V} \Sigma_{i=1}^{N-1} \Sigma_{j>i}^{N} z_{ij} F(z_{ij}) \quad 10.19)$$

donde $z_{ij} = z_i - z_j$, siendo z_i la coordenada z del átomo i

$$F(z_{ij}) = -\frac{du(r)}{dr}\frac{z}{r} \quad 10.20)$$

Ahora el tema es cómo calcular la derivada asociada a una función discontinua. La fuerza (F) del potencial discontinuo es calculada en cada discontinuidad como:

$$\frac{du(r)}{dr} = -k_B T \, \delta(r - \sigma) \quad 10.21)$$

La función delta se estima mediante la función escalón $\Theta(x)$

$$\delta(r - \sigma) = \frac{\Theta(r - \sigma) - \Theta(r - [\sigma + \Delta\sigma])}{\Delta\sigma} \quad 10.22)$$

Esta aproximación es válida cuando

$$\Delta\sigma \rightarrow 0 \quad 10.23)$$

En general, tendremos la contribución a la presión de n-discontinuidades

$$P_{zz} = \rho k_B T + \frac{1}{V} \Sigma_{i=1}^{N-1} \Sigma_{j>i}^{N} z_{ij} [F1 + F2 + F3 + F4 + \ldots] \quad 10.24)$$

Para el caso de una sola discontinuidad

$$P_{zz} = \rho k_B T - \frac{1}{V} \Sigma_{i=1}^{N-1} \Sigma_{j>i}^{N} z_{ij} \, k_B T \left[\frac{\Theta(r - \sigma) - \Theta(r - [\sigma + \Delta\sigma])}{\Delta\sigma} \right] \quad 10.25)$$

se asignan valores cada vez más pequeños de $\Delta\sigma$. Esta metodología fue empleada para calcular propiedades termodinámicas de fluidos que interactúan mediante potenciales discontinuos [67,68,69].

Por otro lado, la presión hidrostática de esferas duras se puede calcular de manera teórica mediante la ecuación de estado de Carnahan-Starling [70].

$$Z = \frac{1 + \eta + \eta^2 - \eta^3}{(1 - \eta)^3} \qquad 10.26)$$

donde Z es el factor de compresibilidad, que es común estimar como:

$$Z = \frac{pV}{NK_BT} \qquad 10.27)$$

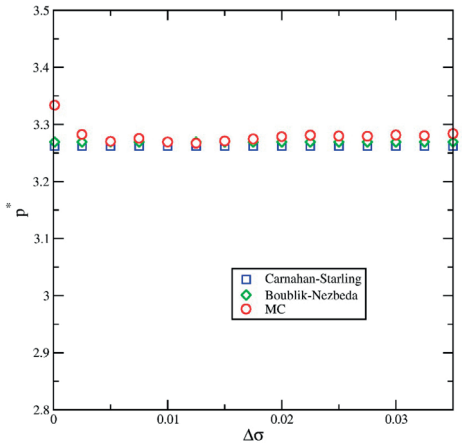
donde p es la presión, V el volumen, N el número de átomos, k_B la constante de Boltzmann, T la temperatura absoluta y σ el diámetro de esfera dura. Además, η es la fracción de empaquetamiento

$$\eta = \frac{\pi N \sigma^3}{6V} \qquad 10.28)$$

Adicionalmente, se considera la ecuación de estado de Boublík-Nezbeda [71] para comparar los resultados obtenidos.

La presión se calcula para cada Δσ. Cuando se encuentra una región horizontal, se realiza una interpolación hasta el eje de las abscisas para determinar el valor de la presión que buscamos. La comparación de los resultados derivados de ambas opciones se muestra en la figura 10.6.

Figura 10.6. Presión de un fluido de esfera



En un código donde se hace uso del método directo de Monte Carlo para simular un fluido de Lennard-Jones, tenemos un orden en el cálculo de la energía y de la implementación del criterio de Metrópolis. A continuación, se muestra esquemáticamente la estructura de un código simple no optimizado.

```
#10.1
inicio del programa

parámetros de entrada (potencial de interacción)
parámetros de entrada (temperatura, presión, etc.)
se lee la configuración inicial (posiciones de las moléculas)
do istep = 1, nstep
  do i = 1, nat
    cálculo de energía(energía-previa)
    se elige y se mueve de manera aleatoria una molecular
    cálculo de energía(energía-posterior)
    criterio de Metrópolis
    cálculo de propiedades termodinámicas
  enddo
enddo
se escribe la configuración final (posiciones de las moléculas)
Fin del programa
```

En el recuadro de abajo, se indica de manera general cómo es el cálculo de la función de distribución radial mediante un código de simulación escrito en Fortran. Solo se indica dónde se debe hacer el llamado de las dos subrutinas involucradas: una de ellas calcula el número de átomos $N(r+\Delta r)$ en cada elemento de volumen; y otra, el volumen de cada división (capas).

```
#10.2
      For istep in range(0,nat-1):
#     Se calcula N(r+Δr)
        if(istep % ngr == 0):
          nrdf(nat,nga,pos)
      return
...
#     Se calcula la RDF
      nstepg = nstep/ngr
      if(nstepg > 0):
        rdf(nstepg)

      Upot  = Upot / aceptados
      Pres   = Pres / aceptados
```

Donde aceptados es una variable que indica el número de movimientos aceptados. Para calcular la presión de un fluido de esferas duras se usa la metodología mencionada líneas arriba, y se modifica la subrutina que concluye la función de distribución radial de la forma en que está indicado en el código en Fortran, que se incluye en el apéndice C. Para este caso se usan 10 valores diferentes de Dp = $\Delta\sigma$. Es importante señalar que el código en Fortran incluido requiere de los datos contenidos en el archivo param, por lo que es necesario editar un archivo con este nombre que contenga la información que a continuación se muestra.

```
parameter(maxnat=5000)
parameter(deltar=0.05)
parameter(nmax=5000)
parameter(pi=3.14159)
```

Ejercicios

[1] Calcular las densidades para las presiones 5, 10 y 15 de un fluido de esferas duras.
[2] Calcular la función de distribución radial a las presiones 5, 10 y 15 de un fluido de esferas duras.
[3] A manera de proyecto, modificar el código para simular una mezcla binaria de esferas duras, con $\varepsilon_{AA} = 1.0$ y $\varepsilon_{BB} = 0.5$

APÉNDICE A

Código en Python para llevar a cabo una simulación a nivel molecular de un fluido de Lennard-Jones, mediante la técnica de dinámica molecular en un ensamble canónico y en unidades reducidas. Los parámetros de entrada son los siguientes: número de pasos de simulación (nsteps) = 200, densidad reducida (dens) = 0.5, número de réplicas (nc) = 3 (lo que nos genera 108 átomos en un arreglo cúbico centrado en las caras), Nátomos = $4*nc^3$ y temperatura reducida (Temp) = 2.

```
#código A.1
import numpy as np

m = 5000
nga = np.zeros(([m]),dtype=np.int_)
acEPI = 0.0
acEKI = 0.0
acETI = 0.0
acTemp = 0.0
acPres = 0.0
deltar = 0.05
ngr = 100

def prog ( nc=3, dens=0.5, Temp=2.0, nsteps=200 ):

    global acEPI,acEKI,acETI,acTemp,acPres,ngr
    nstep_print = 100
    ngr = 100
    print(" ")
    print("{:10}{:6d}    ".format(' pasos de simulación', nsteps) )
    pos,box,nat = fcc(nc,dens)
```

```
   vel = velocidades(nat,Temp)

   print('  ')
   print('  istep     TEMPI        EPI         EKI         ETI
PRESSI')
   print('   ')
   for istep in range ( 1, nsteps + 1 ):

     upot, fij, pvir = fuerza ( nat, pos, vel, box )
     ukin,Tempi,pos,vel,pkin =
mover(nat,pos,vel,fij,Temp,box)
     PRESSI  = pvir + pkin
     EPI = upot / nat
     EKI = ukin / nat
```

```
    ETI = EPI + EKI

     acEPI = acEPI + EPI
     acEKI = acEKI + EKI
     acETI = acETI + ETI
     acTemp = acTemp + Tempi
     acPres = acPres + PRESSI

     if ( istep % nstep_print == 0 ):
        print('%6d %12f %12f %12f %12f %12f' % (istep,Te
mpi,EPI,EKI,ETI,PRESSI))
     if( istep % ngr == 0 ):
        nrdf(nat,box,pos)
```

```
    promedios(nsteps)
    nstepg = nsteps / ngr
    rdf(nat,box,nstepg)
    plotar()

    print(‘  ‘)
    print(“{:11}{:12.3f}”.format(‘ densidad’,          dens) )
    print(“{:10}{:6d}   “.format(‘ numero de atomos’,    nat) )
    print(“{:10}{:6d}   “.format(‘ numero de pasos ‘, nsteps) )
    print(‘  ‘)
    print(‘  ‘)
    return

def nrdf(nat,box,pos):

    import numpy as np
    global nga,deltar
    rij = np.zeros ( 3 )
    box2 = box / 2.0

    for i in range(0,nat-1):
      posxi = pos[i,0]
      posyi = pos[i,1]
      poszi = pos[i,2]
      for j in range(i+1,nat):
        dsq = 0.0

        for k in range ( 0, 3 ):
          rij[0] = posxi - pos[j,0]
          rij[1] = posyi - pos[j,1]
          rij[2] = poszi - pos[j,2]
          rij[k] = rij[k] - np.rint( rij[k] / box ) * box
          dsq = dsq + np.power(rij[k],2.0)
```

```
      d = np.sqrt ( dsq )
       if d <= box2:
         irij = int(d/deltar) + 1
         nga[irij] = nga[irij] + 2
   return

def rdf(nat,box,nstepg):

   import numpy as np
   global nga,deltar
   archive  = open("rdf.dat",'w')
   vol = box * box * box
   nig = int(box / deltar / 2.0)
   rhoj = nat / vol
   denx = float(nstepg * nat)
   fact = 4.0 * np.pi * rhoj / 3.0

   for jj in range(1,nig+1):
      r0  = ( jj - 1 ) * deltar
      r   = r0 + deltar
      den = fact * (np.power(r,3.0) - np.power(r0,3.0))
      gar = int(nga[jj]) / den
      gr  = gar / (denx)
      archive.write(str(r0) + '    ' + str(gr) + '\n' );

   archive.close;
   return

def plotar():
   import pandas as pd
   from  matplotlib import pyplot as plt
```

```
    data=pd.read_csv('rdf.dat',header=0,delim_
whitespace=True)
    x=data.iloc[:,0]
    y=data.iloc[:,1]

    plt.plot(x,y, color='red', linewidth=3, label="dm")
    plt.ylabel('g(r)',fontsize=14)
    plt.xlabel('r',fontsize=14)

    plt.title("Función de distribución radial",fontsize=16)
    plt.legend(loc="upper left")
    plt.savefig("rdf-dm.png")
    plt.show()
    return
```

```
def promedios(nsteps):

    global acEPI,acEKI,acETI,acTemp,acPres
    promEPI = acEPI/nsteps
    promEKI = acEKI/nsteps
    promETI = acETI/nsteps
    promTemp = acTemp/nsteps
    promPres = acPres/nsteps

    print('  ')
    print('  PROMEDIOS')
    print('  ')
    print ('<Temp> ', ' %12f   ' % (promTemp ) )
    print ('<EK>   ', ' %12f   ' % (promEKI  ) )
```

```
    print ('<EP>   ', ' %12f   ' % (promEPI  ) )
    print ('<ET>   ', ' %12f   ' % (promETI  ) )
    print ('<Press>', ' %12f   ' % (promPres ) )
    print('  ')
    return

def fcc(nc,dens):

    import numpy as np
    from itertools import product

    nat = 4*np.power(nc,3)
    box = (nat/dens)**(1.0/3.0)
    print("{:9}{:8.2f}".format(' tamaño de la caja',box) )
    print("{:14}{:6d}   ".format(' número de átomos  ',   nat) )
    print("  ")
    r = np.zeros ( ( nat, 3 ) )
    cell = box / nc
    cell2 = 0.5*cell
    r = np.empty((nat,3),dtype=np.float_)
    r_fcc = np.array([[0.0,0.0,0.0],[cell2,cell2,0.0],[cell2,0.0,ce
ll2],[0.0,cell2,cell2]],dtype=np.float_)
    i = 0
    for ix, iy, iz in product(range(1,nc+1),repeat=3):  # 3 bucles
anidados
        for k in range(4):
            r[i,:] = r_fcc[k,:] + cell*np.array ( [ix-1,iy-1,iz-1]
).astype(np.float_)
            i = i + 1
    return r,box,nat

def velocidades ( nat, Temp ):

    import numpy as np
    mass = 1.0
```

```
  vel    = np.random.randn ( nat, 3 )
   dof    = 3.0 * nat - 3.0
   Tempi  = 2.0 * np.sum(mass*vel**2.0) / dof
   ratio  = np.sqrt(Temp/Tempi)
   vel    = vel * ratio
   return vel

def fuerza ( nat, pos, vel, box ):

  import numpy as np
  eps = 1.0
  sigma = 1.0
  rcut = 2.5
  fij = np.zeros ( [nat,3] )
  pij = np.zeros ( 3 )
  rij = np.zeros ( 3 )
  upot = 0.0
  vol = box * box * box

  for i in range ( 0, nat - 1 ):
    for j in range ( i + 1, nat ):
      dsq = 0.0
      for k in range ( 0, 3 ):
        rij[k] = pos[i,k] - pos[j,k]
        rij[k] = rij[k] - np.rint( rij[k] / box ) * box
        dsq = dsq + np.power(rij[k],2.0)

      d = np.sqrt ( dsq )
      if(d <= rcut):
       upot=upot+4.0*eps*(np.power(sigma/d,12.0)-np.
power(sigma/d,6.0))
```

```
        dupot=-24.0*eps*(2.0*np.power(sigma/d,12.0)-np.
power(sigma/d,6.0))/d

        for k in range ( 0, 3 ):
          fxij = -dupot * rij[k] / d
          fij[i,k] = fij[i,k] + fxij
          fij[j,k] = fij[j,k] - fxij
          pij[k] = pij[k] + rij[k] * fxij
  for k in range ( 0, 3 ):
    pvir = np.sum(pij) / 3.0 / vol
  return upot, fij, pvir

def mover (nat,pos,vel,fij,Temp,box):
  import numpy as np
  velT = np.zeros ([nat,3])
  velk = np.zeros ([nat,3])
  pkij = np.zeros (3)
```

```
 ukin = 0.0
  mass = 1.0
  dt = 0.005
  vol = box * box * box
  for j in range ( 0, nat ):
    for i in range ( 0, 3 ):
      velT[j,i] = vel[j,i]

  for i in range ( 0, nat ):
    for k in range ( 0, 3 ):
```

```
    vel[i,k] = vel[i,k] + dt * ( fij[i,k] / mass )
    pos[i,k] = pos[i,k] + vel[i,k] * dt
    pos[i,k] = pos[i,k] - np.rint( pos[i,k] / box ) * box

ukin = 0.0
for i in range ( 0, nat ):
  for k in range ( 0, 3 ):
    velk[i,k] = 0.5*(vel[i,k] + velT[i,k])
    ukin = ukin + velk[i,k] ** 2.0
    pkij[k] = pkij[k] + mass * velk[i,k] * velk[i,k]

for k in range (0,3):
   pkin = np.sum(pkij) / 3.0 / vol

ukin = 0.5 * mass * ukin
dof = 3.0 * nat - 3.0
Tempi = 2.0 * ukin / dof

sumv = np.zeros (3)
for i in range ( 0, nat ):
  for k in range ( 0, 3 ):
     sumv[k] = sumv[k] + vel[i,k] * mass

for k in range ( 0, 3 ):
   sumv[k] = sumv[k] / float(nat)

for i in range ( 0, nat ):
  for k in range ( 0, 3 ):
    vel[i,k] = vel[i,k] - sumv[k]

ratio = np.sqrt(Temp/Tempi)
```

```
  for i in range ( 0, nat ):
    for k in range ( 0, 3 ):
      vel[i,k] = vel[i,k] * ratio
  return ukin,Tempi,pos,vel,pkin

prog ( nc=3, dens=0.5, Temp=2, nsteps=200 )
```

APÉNDICE B

Código en Python para simular un fluido de Lennard-Jones con Monte Carlo en un ensamble NVT. Los parámetros de entrada son los siguientes: número de pasos de simulación (nsteps) = 2000, densidad reducida (dens) = 0.5, número de réplicas (nc) = 3 (lo que nos genera 108 átomos en un arreglo cúbico centrado en las caras), Nátomos = $4*nc^3$ y temperatura reducida (Temp) = 2.

```
#código B.1
def prog(nc=3, dens=0.5, Temp=2.0, nsteps=2000):

    global pos
    import sys
    import random
    import numpy as np
    from math import exp
    global nga,deltar
    dr     = 0.2
    beta   = 1.0/Temp
    cont   = 0
    acEPI  = 0.0
    nstep_print = 100

    pos,box,nat = fcc(nc,dens)
    vol = box * box * box
    print("{:7}{:4.1f}".format(' vol =',vol) )
    print(" ")
    Elrc = lrc(nat,dens,box)
    E,overlap = energiaall(nat,box)
```

```
if(overlap):
    print('traslape en configuración inicial')
    sys.exit()

Ei     = (E + Elrc) / float(nat)

print('   ')
print("{:10}{:9.4f}".format(' energía potencial',  Ei) )
print('  ')
print('    istep       EPI    ')
print('   ')
```

```
for istep in range(1,nsteps+1):
  for i in range(0, nat):

    xiold = pos[i,0]
    yiold = pos[i,1]
    ziold = pos[i,2]
    Eold = energiai(i,nat,xiold,yiold,ziold,box)

    rand = random.random()
    xinew = xiold + ( 2.0*rand - 1.0 ) * dr
    yinew = yiold + ( 2.0*rand - 1.0 ) * dr
    zinew = ziold + ( 2.0*rand - 1.0 ) * dr

    xinew = xinew - np.rint( xinew / box ) * box
    yinew = yinew - np.rint( yinew / box ) * box
    zinew = zinew - np.rint( zinew / box ) * box
```

```
        Enew = energiai(i,nat,xinew,yinew,zinew,box)
        deltaE = Enew - Eold

        if( deltaE < 0.0 ):
            E       = E + deltaE
            pos[i,0] = xinew
            pos[i,1] = yinew
            pos[i,2] = zinew
        else:
            rand = random.random()
            if( exp(-beta*deltaE) > rand ):
              E       = E + deltaE
              pos[i,0] = xinew
              pos[i,1] = yinew
              pos[i,2] = zinew

        cont = cont + 1
        EPI = ( E + Elrc ) / float( nat )
        acEPI = acEPI + EPI

       if(istep % nstep_print == 0):
          print('%8d %12.4f' % (istep,EPI))

    EP     = acEPI / cont
    print('  ')
    print("{:10}{:8.4f}".format(' energía potencial = ',    EP) )
    print("{:11}{:15.4f}".format(' densidad =   ',          dens) )
    print("{:11}{:15.4f}".format(' temperatura =  ',
Temp) )
    print("{:10}{:8d}   ".format(' número de átomos = ',    nat) )
    print("{:10}{:8d}   ".format(' número de pasos =  ', nsteps) )
```

```
    print('  ')
    print('  ')
    print('Ya termine!')
    print('  ')
    print('  ')
    return

def fcc(nc,dens):
    import numpy as np
    from itertools import product

    nat = 4*np.power(nc,3)
    box = (nat/dens)**(1.0/3.0)
    print("  ")
    print("{:9}{:8.4f}".format(' tamaño de la caja = ',box) )
    print("{:14}{:6d}   ".format(' número de átomos = ',   nat) )
    print("  ")
    r = np.zeros ( ( nat, 3 ) )
    cell = box / nc
    cell2 = 0.5*cell
    r = np.empty((nat,3),dtype=np.float_)
    r_fcc = np.array([[0.0,0.0,0.0],[cell2,cell2,0.0],[0.0,cell2,cel
l2],[cell2,0.0,cell2]],dtype=np.float_)
    i = 0
    for ix, iy, iz in product(range(1,nc+1),repeat=3):  # 3 bucles
anidados
        for k in range(4):
                r[i,:] = r_fcc[k,:] + cell*np.array ( [ix-1,iy-1,iz-1]
).astype(np.float_)

            i = i + 1
    return r,box,nat
```

```
def lrc(nat,dens,box):

    import numpy as np
    rcut = 2.5
    sigma = 1.0

    sr3 = ( sigma / rcut )**3.0
    sr9 = sr3 ** 3.0
    Elrc12 =   8.0 * np.pi * dens * float(nat) * sr9 / 9.0
    Elrc6  = - 8.0 * np.pi * dens * float(nat) * sr3 / 3.0
    Elrc   =   Elrc12 + Elrc6

    EElrc = Elrc/nat

    print(" Correcciones de largo alcance")
    print("{:11}{:15.4f}".format(' Elrc',  EElrc) )
    return Elrc
```

```
def energiaall(nat,box):
    import numpy as np
    eps = 1.0
    sigma = 1.0
    rcut = 2.5
    E = 0.0
    rmin = 0.7*sigma
    overlap = False
    global pos

    for i in range(0,nat):
```

```
    xii = pos[i,0]
    yii = pos[i,1]
    zii = pos[i,2]

    for j in range(i+1,nat):

      rxij = xii - pos[j,0]
      ryij = yii - pos[j,1]
      rzij = zii - pos[j,2]

      rxij = rxij - np.rint( rxij / box ) * box
      ryij = ryij - np.rint( ryij / box ) * box
      rzij = rzij - np.rint( rzij / box ) * box

      dsq = rxij*rxij + ryij*ryij + rzij*rzij

      if(dsq < rmin*rmin):
        overlap = True
        return

      else:
        if(dsq < rcut*rcut):

          E=E+4.0*eps*(np.power(sigma/dsq,6.0)-np.
power(sigma/dsq,3.0))

  return E,overlap

def energiai(i,nat,xi,yi,zi,box):
  import numpy as np
  eps = 1.0
  sigma = 1.0
  rcut = 2.5
  E = 0.0
  global pos
```

```
  for j in range (0,nat):
    if(i != j):

      rxij = xi - pos[j,0]
      ryij = yi - pos[j,1]
      rzij = zi - pos[j,2]

      rxij = rxij - np.rint( rxij / box ) * box
      ryij = ryij - np.rint( ryij / box ) * box
      rzij = rzij - np.rint( rzij / box ) * box

      dsq = rxij*rxij + ryij*ryij + rzij*rzij

      if(dsq <= rcut*rcut):
        E=E+4.0*eps*(np.power(sigma/dsq,6.0)-np.
power(sigma/dsq,3.0))
   return E

prog(nc=3, dens=0.5, Temp=2.0, nsteps=2000)
```

APÉNDICE C

Código en Fortran 77 para realizar una simulación a nivel molecular de un fluido de esferas duras mediante el método directo de Monte Carlo. Se fijan los parámetros termodinámicos reducidos: $\rho' = 0.5$, $T' = 0.5$. Se consideran 108 átomos.

```
#código C.1
      program mcsh
      implicit double precision(a-h,o-z)
      include 'param'

      common / posicion / rx(maxnat), ry(maxnat), rz(maxnat)
      common /posfcc/ rxfcc(maxnat),ryfcc(maxnat),rzfcc(ma
xnat)
      common / caja / boxx,boxy,boxz
      common / cajainv / boxix,boxiy,boxiz
      common /numatom/ n
      common / press  / zx,zy,zz
      common / p1 / sumpxx(10),sumpyy(10),sumpzz(10)
      common / p2 / dp(10),sigma_dp(10)
      logical     overlap

      write(6,*)'   '
      write(6,*)'simulacion desde inicio.....[1]'
      write(6,*)'continuar simulación.....[2]'
      read(5,*)noption
      if(noption.eq.1) then
        call fcc(nfcc,dens)
        n = nfcc
```

```
    rx(i) = rxfcc(i)
  do i=1,n
    ry(i) = ryfcc(i)
    rz(i) = rzfcc(i)
  enddo

  boxix = 1.0/boxx
  boxiy = 1.0/boxy
  boxiz = 1.0/boxz
endif

if(noption.eq.2) then
  call leer
endif
```

```
vol  = boxx*boxy*boxz
 dens = n/vol

 write(6,*)'    '
 write(*,'('' número de átomos = '', I10   )' )n
 write( *, '( '' boxx,...,          '',3F12.4)')boxx,boxy,boxz
 write( *, '( '' densidad =         '',1F12.4)')dens

 write(6,*)'   '
 write(*,'('' número de pasos de simulación                '')')
 read (*,*) nstep
 write(*,'('' frecuencia de impresión de datos             '')')
 read (*,*) iprint
 write(*,'('' intervalo de actualización del max. displ.   '')')
```

```
    read (*,*) iratio
    write(*,'('' frecuencia de cálculo de la gdr              '')')
    read (*,*) ngr
    write(6,*)'   '

    sigma  = 1.0
    drmax  = 0.15

    if ( rcut .gt. 0.5*boxx ) stop 'radio de corte muy grande'

    acm    = 0.0
    acmmva = 0.0
    write( 6,* )'       '
    write( 6, '( ''     paso      ratio                  '')')

    do k=1,10
      sumpxx(k) = 0.0
      sumpyy(k) = 0.0
      sumpzz(k) = 0.0
    enddo

    do 100 istep = 1, nstep
     do 99 i = 1, n

        rxiold = rx(i)
        ryiold = ry(i)
        rziold = rz(i)

        call energía(rxiold, ryiold, rziold, i, rcut,
sigma,overlap)
        rxinew = rxiold + ( 2.0 * ranf ( dummy ) - 1.0 ) * drmax
```

```
          ryinew = ryiold + ( 2.0 * ranf ( dummy ) - 1.0 ) *
drmax
          rzinew = rziold + ( 2.0 * ranf ( dummy ) - 1.0 ) *
drmax
```

```
         rxinew = rxinew - dnint ( rxinew*boxix )*boxx
          ryinew = ryinew - dnint ( ryinew*boxiy )*boxy
          rzinew = rzinew - dnint ( rzinew*boxiz )*boxz

          call energia(rxinew, ryinew, rzinew, i, rcut,
sigma,overlap)

          if(overlap)goto 99
          if ( .not. overlap ) then
            rx(i)  = rxinew
            ry(i)  = ryinew
            rz(i)  = rzinew
            acmmva = acmmva + 1.0
          endif
          acm = acm + 1.0
99        continue

        if ( mod ( istep, iratio ) .eq. 0 ) then
           ratio = acmmva / real ( n * iratio )
           if ( ratio .gt. 0.5 ) then
              drmax = drmax * 1.05
           else
              drmx = drmx * 0.95
           endif
           acmmva = 0.0
        endif
```

```
      if ( mod ( istep, iprint ) .eq. 0 ) then
         write(6,13)istep, ratio
      endif
13    format(I8,1F12.4)
      if(mod(istep,ngr).eq.0) then
          call nrdf
      endif
100   continue

      call equation(dens,compress_cs,compress_bn)
      nstepg = nstep/ngr
      call presion(nstepg,compress_cs,compress_bn)
      call escribir

      nstepg = nstep/ngr
      if(nstepg.gt.0) then
         call rdf(nstepg)
      endif
      call pdb
      stop
      end
```

```
     subroutine fcc(nfcc,dens)
      implicit double precision(a-h,o-z)
      include 'param'

      common / posicion / rx(maxnat), ry(maxnat), rz(maxnat)
      common /posfcc/ rxfcc(maxnat),ryfcc(maxnat),rzfcc(ma
xnat)
```

```
      common / caja / boxx,boxy,boxz
      common / cajainv / boxix,boxiy,boxiz

      write(6,*) 'dame nr;  nr=número de replicas'
      write(6,*) 'número de átomos = 4*nc*nc*nc'
      read(5,*) nr
      nfcc=4.0*nr*nr*nr

      write(6,*) 'número de moléculas ',nfcc
      read(5,*) dens
      boxx = (dfloat(nfcc)/dens)**(1.0d0/3.0d0)
      boxy = boxx
      boxz = boxx

      write(6,*) 'boxx',boxx

c     lado de la celda unitaria
      cellx=boxx/float(nr)
      celly=boxy/float(nr)
      cellz=boxz/float(nr)
      cell2x=0.5*cellx
      cell2y=0.5*celly
      cell2z=0.5*cellz

c     primer atomo A
      rxfcc(1)=0.0
      ryfcc(1)=0.0
      rzfcc(1)=0.0
c     segundo atomo B
      rxfcc(2)=cell2x
      ryfcc(2)=cell2y
      rzfcc(2)=0.0
```

```
c     tercer atomo C
      rxfcc(3)=0.0
      ryfcc(3)=cell2y
      rzfcc(3)=cell2z
c     cuarto atomo D
      rxfcc(4)=cell2x
      ryfcc(4)=0.0
      rzfcc(4)=cell2z
```

```
c     se replica la celda unitaria
      m=0
      do 99 iz=1,nr
      do 98 iy=1,nr
      do 97 ix=1,nr
      do 96 iref=1,4
      rxfcc(iref+m)=rxfcc(iref)+cellx*float(ix-1)
      ryfcc(iref+m)=ryfcc(iref)+celly*float(iy-1)
      rzfcc(iref+m)=rzfcc(iref)+cellz*float(iz-1)
96    continue
      m=m+4
97    continue
98    continue
99    continue
      return
      end

C     *******************************************************
```

```
      subroutine leer
      implicit double precision(a-h,o-z)
      include 'param'

      common / posicion / rx(maxnat), ry(maxnat), rz(maxnat)
      common / caja / boxx,boxy,boxz
      common / cajainv / boxix,boxiy,boxiz
      common /numatom/ n

      open(1, file ='posiciones.dat')
      read(1,*) n
      read(1,*) boxx,boxy,boxz,boxix,boxiy,boxiz
      do i = 1,n
        read(1,*)rx(i),ry(i),rz(i)
      enddo
      close(1)
      return
      end

C     ******************************************************

      subroutine escribir
      implicit double precision(a-h,o-z)
      include 'param'

      common / posicion / rx(maxnat), ry(maxnat), rz(maxnat)
      common / caja / boxx,boxy,boxz
      common / cajainv / boxix,boxiy,boxiz
```

```
      common /numatom/ n

       open (1, file ='posiciones.dat')
       write (1,*) n
       write (1,*) boxx,boxy,boxz,boxix,boxiy,boxiz
       do i = 1,n
         write (1,*) rx(i), ry(i), rz(i)
       enddo
       close (1)

       return
       end

C      ******************************************************

       real*8 function ranf(dummy)

       integer     n
       integer     l, c, m
       parameter ( l = 1029, c = 221591, m = 1048576 )

       integer     seed
       real*8      dummy
       save        seed
       data        seed / 0 /
       seed = mod ( seed * l + c, m )
       ranf = real ( seed ) / m
       return
       end

C      ******************************************************
```

```
subroutine nrdf
implicit double precision(a-h,o-z)
include 'param'

common / posicion / rx(maxnat), ry(maxnat), rz(maxnat)
common / caja / boxx,boxy,boxz
common / cajainv / boxix,boxiy,boxiz
common /numatom/ n
common / gr / nga(nmax),gr(nmax)
common / press  / zx,zy,zz
common / p1 / sumpxx(10),sumpyy(10),sumpzz(10)
common / p2 / dp(10),sigma_dp(10)

box2    = 0.5*boxx
dp(1)   = 0.0001
```

```
dp(2)    = 0.0025
dp(3)    = 0.005
dp(4)    = 0.0075
dp(5)    = 0.01
dp(6)    = 0.0125
dp(7)    = 0.015
dp(8)    = 0.0175
dp(9)    = 0.02
dp(10)   = 0.0225
sigma    = 1.0

do k=1,10
```

```
      sigma_dp(k) = sigma + dp(k)
    enddo
    do  i = 1,n-1
     rxi =  rx(i)
     ryi =  ry(i)
     rzi =  rz(i)

     do j = i+1, n
      dx = rxi - rx(j)
      dy = ryi - ry(j)
      dz = rzi - rz(j)

      dx = dx - dnint(dx*boxix)*boxx
      dy = dy - dnint(dy*boxiy)*boxy
      dz = dz - dnint(dz*boxiz)*boxz
      rij = sqrt(dx*dx+dy*dy+dz*dz)
      if (rij.le.box2) then
          irij     = rij/deltar
          nga(irij) = nga(irij) + 2
      endif

C    **** presión ****

      do k = 1,10
       if(rij.ge.sigma.and.rij.le.sigma_dp(k))then
        sumpxx(k) = sumpxx(k) + dx*dx/rij/dp(k)
        sumpyy(k) = sumpyy(k) + dy*dy/rij/dp(k)
        sumpzz(k) =  sumpzz(k) + dz*dz/rij/dp(k)
       endif
      enddo
     enddo
```

```
enddo
return
end
```

```
subroutine rdf (nstepg)
 implicit double precision(a-h,o-z)
 include 'param'

 common / posicion / rx(maxnat), ry(maxnat), rz(maxnat)
 common / caja / boxx,boxy,boxz
 common / boxixi / boxix,boxiy,boxiz
 common /numatom/ n
 common / gr / nga(nmax),gr(nmax)

 open(31,file='gr.dat')

 nig  = 0.5*boxx/deltar
 vol  = boxx*boxy*boxz
 rhoj = n/vol
 denx = dfloat(nstepg*n)
 fact = 4.0d0*pi*rhoj/3.0d0

 do jj=1,nig
    r0     = (jj) *deltar
    r      = r0 + deltar
    den    = fact * ( r**3 - r0**3 )
    gar    = nga(jj)/den
    gr(jj) = gar / denx
```

```
      write(31,'(2F10.4)') r0,gr(jj)
    enddo

    return
    end

C   ************************************************

    subroutine energia (rxi,ryi,rzi,i,rcut,sigma,overlap)
    implicit double precision(a-h,o-z)
    include 'param'

    common / posicion / rx(maxnat), ry(maxnat), rz(maxnat)
    common / caja / boxx,boxy,boxz
    common / cajainv / boxix,boxiy,boxiz
    common /numatom/ n
    logical   overlap
    character cnfile*80

    overlap = .false.
    rcutsq  = rcut * rcut
    sigsq   = sigma * sigma
```

```
  do j = 1, n
     if ( i .ne. j ) then
        rxij  = rxi - rx(j)
        ryij  = ryi - ry(j)
        rzij  = rzi - rz(j)
```

```
          rxij  = rxij - dnint ( rxij*boxix )*boxx
          ryij  = ryij - dnint ( ryij*boxiy )*boxy
          rzij  = rzij - dnint ( rzij*boxiz )*boxz
          rijsq = rxij * rxij + ryij * ryij + rzij * rzij
          rij = sqrt(rijsq)

          if(rij.lt.sigma)then
             overlap = .true.
             return
          endif

        endif
      enddo

      return
      end

C     ********************************************************
*****

      subroutine pdb
      implicit double precision(a-h,o-z)
      include 'param'

      common / posicion / rx(maxnat), ry(maxnat), rz(maxnat)
      common / caja / boxx,boxy,boxz
      common / boxixi / boxix,boxiy,boxiz
      common /numatom/ n
      dimension symbol(5000)
      dimension  rxf(maxnat), ryf(maxnat), rzf(maxnat)
      character*4 w1,w2,w4,symbol
```

```
      character*6 w3

      open(7,file='FotoFinal.pdb')
      rewind(7)
      w1='ATOM'
      w3='    '
      w4='    '
      n1= 1
      do i=1,n
        symbol(i) = '  Si'
      enddo
      sigmar = 3.405
```

```
      do i=1,n
        rxf(i) = rx(i)*sigmar
        ryf(i) = ry(i)*sigmar
        rzf(i) = rz(i)*sigmar
      enddo
      do i=1,n
          w2 = symbol(i)
          write(7,10) w1,i,w2,w3,n1,i,w4,
     &                    rxf(i),ryf(i),rzf(i)
      enddo
10    format(A4,I7,A4,A5,I2,I4,A4,3F8.3)
      return
      end

C     ***************************************************
```

```
      subroutine equation(dens,compress_cs,compress_bn)
      implicit double precision(a-h,o-z)
      include 'param'
      common / caja / boxx,boxy,boxz
      common /numatom/ n

      vol = boxx*boxy*boxz
      rho = n/vol
      eta = pi*dens/6.0

C     Ecuación de Carnahan-Starling
      sum0 = 1.0 + eta + eta**2
      den0 = ( 1.0 - eta)**3
      compress_cs = ( sum0 - eta**3 )/den0
      densty0 = sqrt(2.0)

C     Ecuación de Boublík-Nezbeda
      b1 = 0.764314
      b2 = 0.151532
      b3 = 0.654551
      sum1 = b1*eta**3 + b2*eta**4 + b3*eta**5
      compress_bn = ( sum0 - sum1)/den0
      write(6,*)'   '
      write(*,'('' eta          es       '',F15.8)') eta
      write(*,'('' rho/rho_0    es       '',F15.8)') dens/densty0
      write(*,'('' pv/nkt(cs)   es       '',F15.8)') compress_cs
      write(*,'('' pv/nkt(bn)   es       '',F15.8)') compress_bn
      write(6,*)'   '

      return
      end
```

```
      subroutine presion(nstepg,compress_cs,compress_bn)
       implicit double precision(a-h,o-z)
       include 'param'

       common / caja / boxx,boxy,boxz
       common /numatom/ n
       common / press  / zx,zy,zz
       common / p1 / sumpxx(10),sumpyy(10),sumpzz(10)
       common / p2 / dp(10),sigma_dp(10)

       open(56,file='presión.dat')
       write(6,*)'   '
       write(6,*)'k    dp   zxx   zyy   zzz  pvnkt'

       do k = 1,10
          sumpxx(k) = sumpxx(k)/nstepg
          sumpyy(k) = sumpyy(k)/nstepg
          sumpzz(k) = sumpzz(k)/nstepg
          zxx = 1 + sumpxx(k)/n
          zyy = 1 + sumpyy(k)/n
          zzz = 1 + sumpzz(k)/n
          pvnkt = (zxx + zyy + zzz)/3.0
          write(6,'(I4,5F10.4)')k,dp(K),zxx,zyy,zzz,pvnkt
          write(56,'(4F12.6)')dp(k),compress_cs,compress_
bn,pvnkt
          write(6,*)'     '
       enddo

       return
       end
```

REFERENCIAS

1. https://www.python.org/ consultado en junio 2023.
2. Langtangen, H.P. (2012). *A Primer on Scientific Programming with Python*, 3th ed. Berlin: Spriger-Verlag.
3. Kalb, I. (2016). *Learn to Program with Python*, California USA: Apress.
4. Cane, A. (2019). *Programacion con Python: Guía completa para principiantes, aprende sobre los reinos de la programación con Python*. 1a ed. Michigan, USA: Independently Published.
5. Mendenhall, W., Beaver, R.J., Beaver, B.M. (2006). *Introduction to Probability and Statistics*, 13th ed. Canada, Cengage Learning, Inc.
6. Gutiérrez-González, E., Vladimirovna Panteleeva, O. (2014). *Probabilidad y estadística: Aplicaciones a la ingeniería y las ciencias*. 1a ed. México: Grupo editorial Patria.
7. Alonso, M., Finn, E.J. (1976). *Física.* Vol. 1. México: Fondo Educativo Interamericano.
8. Resnick, R., Halliday, D.(2004). *Física.* 4a ed. Vol. 1. México: CECSA.
9. https://www.ks.uiuc.edu/Research/vmd/ consultado en junio 2023.
10. https://plasma-gate.weizmann.ac.il/Grace/ consultado en junio 2023.
11. Marsden, J.E., Tromba, A.J. (1991). *Cálculo vectorial*. 3a ed. Addison-Wesley Iberoamérica.
12. Leithold, L. (1998). *El c*álculo. 7a ed. México: Oxford University Press.
13. Kline, M. (1990). *Mathematical Thought from Ancient to Modern Times*. Vol. 1, 1st Ed. Oxford University Press.
14. Courant, R., John, F. (1965). *Introduction to calculus and analysis*, V. 1, Interscience Publishers, Wiley & Sons, Inc.
15. Maron, M.J., López, R.J. (1995). *Análisis numérico: Un enfoque práctico.* 1a ed. México: CECSA.
16. HTTPS://WWW.WOLFRAM.COM/MATHEMATICA/ consultado en junio 2023.
17. https://www.mathworks.com/products/matlab.html consultado en junio 2023.
18. Bencardino, C.M. (2012). *Estadística y muestreo*, 13a ed. México: ECOE ediciones.
19. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.(2007). *Numerical recipe.* 3a ed. New York: Cambridge University Press.

20. Bacchini, R.D., Vázquez, L.V., Bianco, M.J., García-Fronti, J.I. (2018). *Introducción a la probabilidad y a la estadística.*, 1a ed. Argentina: CMA-IADCOM-UBA.
21. Courant, R., Robbins, H. (1979). ¿What is mathematics? New York and London: Oxford University Press.
22. Birkhoff, G., Mac Lane, S. (2010). *A survey of Modern Algebra.* 5a ed. New York: CRC Press, Taylor & Francis Group.
23. Pastor, J.R., Pi Calleja, P., Trejo, C.A. (1969). *Análisis matemático*. Volumen I. Capítulo IV. 8a ed. ("Algoritmo Algebraico"). Buenos Aires: Editorial Kapelusz.
24. Baldor, A. (2019). *Álgebra,* 4a ed. México: Editorial Patria.
25. Tjalling J. Y. (1995). Historical development of the Newton-Raphson method, SIAM Review 37 (4), 531–551.
26. Burden, R.L., Faires, D.J., y Burden, A.M. (2017). *Análisis numérico*, 10a ed. México: Cengage Learning Inc.
27. Boyce, W.E., DiPrima, R.C. (1994). *Introducción a las ecuaciones diferenciales*, 4a ed. México: UNAM, Editorial Limusa.
28. Burden, R.L., Faires, D.J., y Burden, A.M. (2017). *Análisis numérico*, 10a ed. México: Cengage Learning Inc.
29. Atkinson, K.E. (1978). *An Introduction to Numerical Analysis*. New Jersey, USA: Wiley.
30. Bulirsch, R., Stoer, J. (1980). *Introduction to Numerical Analysis*. New York, USA: Springer.
31. Butcher, J.C., 2003. *Numerical Methods for O.D.E.* 2th ed. New Jersey. USA: Wiley.
32. Chapra S.C., R.O. Canale. (2007). *Métodos numéricos para ingenieros*, 5a ed. México: McGraw-Hill, Interamericana.
33. Zill, D.G., M.R. Cullen. (2008). *Matemáticas avanzadas para ingeniería*, Vol. 1. Ecuaciones diferenciales, 3a ed. México: McGraw-Hill.
34. Ayres, F. Jr. (1985). *Matrices.* 1a ed. USA: Schaum-McGraw-Hill.
35. Dorf, R.C. (1980). *Introducción al Álgebra de matrices*. 1a ed. México: Limusa.
36. Torres-Arias, J.J. (2015). *Matrices y sistemas de ecuaciones lineales*, 2a ed. Colombia: Universidad de Medellín.
37. Lang, S. (1987). *Linear algebra*, 3er edition, New York: Springer Science Business Media.
38. Poole, D. (2004). *Álgebra lineal: Una introducción moderna*. México: Thomson.
39. Kolman, B. y Hill, D. R. (2006). *Álgebra lineal*, 8a ed. México: Pearson.
40. Merino Gonzalez, L.M y Santos Alez, E. (2021). Álgebra lineal con métodos elementales. 3a ed. Madrid: Ediciones Paraninfo.

41. Allen, M.P., Tildesley, D.J., 1991. *Computer Simulation of Liquids*, First Edition, Oxford University Press.
42. Allen, M.P. (2004). *Introduction to Molecular Dynamics Simulation*. United Kingdom: University of Warwick.
43. van der Waals, J.D. (1873). *On the Continuity of the Gaseous and Liquid State. Ph.D. Dissertation*. Netherlands: Leiden University.
44. Valderrama, J.O. (2010). El legado de Johannes Diderik van der Waals y su Conferencia Nobel, *Información Tecnológica.* Vol. 21, 3-12. Santiago de Chile.
45. Lennard-Jones, J.E. (1931). *Cohesion. Proceedings of the Physical Society*. 43, 461-482.
46. Galicia-Pimentel, U.F., Osorio-González, D., y López-Lemus, J. (2006). On the morse potential in liquid phase and at liquid-vapor interface, *Revista Mexicana de F*ísica. México: Sociedad Mexicana de Física. 52, 422-428.
47. Mie, G. (1903). *Zur kinetischen Theorie der einatomigen Körpe. Annalen der Physik,* 42, 657-697.
48. Coulomb, C.A. (1789). *Mémoires sur l'électricité et la magnétisme.* Académie Royale des Sciences, France, 569.
49. Ryckaert, J.P., Bellemans, A. (1978). Molecular dynamics of liquids alkanes, *Faraday Discussions of the Chemical Society*. 66, 95-106.
50. Ewald, P.P. (1921). Die Berechnung optischer und elektrostatischer Gitterpotentiale, *Annals of Physics*. 369, 253-287.
51. López-Lemus, J., Alejandre, J. (2002). Thermodynamic and transport properties of simple fluids using lattice sums: bulk phases and liquid-vapour interface. *Molecular Physics*. 100, 2983-2992.
52. Karasawa, N., Goddard III, W.A. (1989). Acceleration of convergence for lattice sums. *The Journal of Chemical Physics*. 93, 7320-7327.
53. Salin, G., Caillol, J.-M., 2000. Ewald sums for Yukawa potentials. *The Journal of Chemical Physics*. 113, 10459-10463.
54. Schoen, M., Klapp, S. (2007). *Reviews in computational chemistry*, Vol. 24 Wiley-VCH.
55. Darden, T., York, D., Pedersen, L. (1993). Particle mesh Ewald: An Nlog(N) method for Ewald sums in large systems, *The Journal of Chemical Physics.* 98, 10089-10092.
56. Pathria, R.K. (1972). *Statistical Mechanics. International Series of Monographs in Natural Philosophy*, volume 45. Pergamon Press.
57. Reif, F. (1993). *Elementary Statistical Physics*. (Berkeley Physics Course). 2a ed, Vol. 5. McGraw-Hill.

58. Huang, K. (1987). *Statistical Mechanics*. John Wiley and Sons (second edition).
59. https://www.lammps.org/ consultado en junio 2023.
60. https://www.ovito.org/ consultado en junio 2023.
61. Moran, J.C. (1982). *The automatic allocation of tolerances through cost optimization*, Bachelor's Degree Thesis, M.I.T.
62. Heyes, D.M. (2015). *The Lennard-Jones fluid in the liquid-vapour critical reg*ion, CMST 21, 169-179.
63. Potoff, J.J., Panagiotopoulos, A.Z. (1998). Critical point and phase behavior of the pure fluid and a Lennard-Jones mixture. *The Journal of Chemical Physics*. 109, 10914-10920.
64. Frenkel, D., Smit, B. (2002). *Understanding Molecular Simulation from Algorithms to Applications*, Academic Press.
65. Caballero, M.E., Hernández, N.S., Rivero, V.M., Uribe-Bravo, G., Velarde, C. (2004). *Cadenas de Markov. Un enfoque elemental.* Vol. 29. México: UNAM, Sociedad Matemática Mexicana.
66. Puterman, M.L. (1994). Markov Decision Processes: *Discrete Stochastic Dynamic Programming*. Wiley.
67. González-Melchor, M., Trokhymchuk, A., Alejandre, J. (2001). Surface tension at the vapor/liquid interface in an attractive hard-core Yukawa fluid, *The Journal of Chemical Physics.* 115, 3862-3872.
68. Orea, P., Duda, Y., Alejandre, J. (2003). Surface tension of a square well fluid. *The Journal of Chemical Physics*. 118, 5635-5639.
69. Orea, P., et al. (2004). Liquid–vapor interface of square-well fluids of variable interaction range, *The Journal of Chemical Physics*. 120, 11754-11764.
70. Carnahan, N.F., Starling, K.E. (1969). Equation of State for Nonattracting Rigid Spheres. *The Journal of Chemical Physics.* 51, 635.
71. Boublík, T., Nezbeda, I. (1986). P-V-T behavior of hard body fluids theory and experiment, *Collection of Czechoslovak Chemical Communications*. 51, 2301-2432.

Jorge López Lemus

Doctor en Ciencias (Física) por la UAM-I. Nivel II del SNII. Profesor de tiempo completo en la Facultad de Ciencias de la UAEMEX. Experiencia en el desarrollo de *software*.

Elizabeth Gemigniani Ricárdez

Licenciatura en Sistemas Computacionales por la BUAP. Programadora en Servicios Educativos Integrados al Estado de México. Experiencia en el desarrollo de *softwar*e.

Benjamín Ibarra Tandi

Doctor en Ciencias (Física) por la UAM-I. Miembro del SNII, Nivel I. Profesor de tiempo completo en la Facultad de Ciencias de la UAEMEX. Experiencia en el desarrollo de *software*.

El objetivo de esta obra es brindar herramientas a los alumnos para analizar problemas académicos particulares en física. Se muestran códigos escritos en el intérprete Python para obtener soluciones numéricas. Se revisan temas generales que son útiles para los estudiantes en el desarrollo de temas finales para graduarse. El público al que está dirigido esta obra es la población estudiantil de la Licenciatura en Física, pero también sirve de instrucción para la población interesada en general, ya que son incluidos los códigos completos en Python. La temática abordada es la utilidad del uso de *software* y *hardware* en la solución y análisis de problemas físicos. Se muestra la estructura de los códigos en Python.

SDC

ANIVERSARIO
UAEMéx