IKC-MH.55
Scientific Computing with Python
Lecture Notes

Cem Özdoğan

January 9, 2024

Contents

List of Tables

List of Figures

Chapter 1

Preliminaries

1.1 First Meeting

- IKC-MH.55 Scientific Computing with Python 2023-2024 Fall
- FRIDAY 14:00-16:00 (T) H1-86
- Instructor: Cem Özdoğan, Engineering Sciences Dept.
  Faculty of Engineering and Architecture Building, H1-33
- TA: NA
- WEB page: http://cemozdogan.net/
- Announcements: Watch this space for the latest updates.

```
Wednesday, October 4, 2023 In the first lecture, there
will be first meeting. The lecture notes will be
published soon, see Course Schedule section.
```

- All the lecture notes will be accessible via Tentative Course Schedule & Lecture Notes.
- All the example py-files (for lecturing and hands-on sessions) will be accessible via the link.

1.1.1 Lecture Information

- Python is a well-designed, modern programming language and widely used in computational science and engineering.
- It is a powerful tool since it includes a wide range of features tailored for scientific computing.
- This course is not either a numerical methods or a programming python course.
- However, this course is designed to use computer programming to implement numerical algorithms for solving physics/engineering problems.
- Consequently, Python (fundamentals of programming in Python, NumPy, SciPy, Matplotlib libraries) and some numerical techniques (practice at physics/engineering problems) will be learned implicitly.
- You may be expected to do significant programming and problem solving.

- An understanding of the concepts of elementary calculus, in particular solutions of differential equations and Newtonian/wave mechanics are required but not mandatory since they will be explained as needed.

- Important announcements will be posted to the Announcements section of this web page, so please check this page frequently.

- You are responsible for all such announcements, as well as announcements made in lecture.

1.1.2 Course Overview

- IKC-MH.55 is intended to provide students a practical introduction for using the computer as a tool to solve physics and engineering problems.

- The fundamental advantage of using computers in science is the ability to treat systems that cannot be solved analytically.

- So that computing has become a major tool in science/engineering and it is called the third pillar along with experiments and theory.

- Numerical techniques such as: Interpolation & Model Fitting, Derivatives & Integrals, Basic Linear Algebra, Eigenvalue Problems, Differential equations, ODE and PDE solvers are used to solve problems from all areas of science and engineering.

- Python implementation of these algorithms will be covered only whenever necessary in the context of the course.

- Each class will be focused towards solving a particular physical/engineering problem.

- Problems will be drawn from diverse areas of real-life examples as much as possible.

- Theory or model, method of solution/algorithm, solution implementation (analytic, Python) and visualization /exploration will be outlined for the problem description.

- Upon completion of this course the students will be able to understand/explain/apply;

  - Learn how to work in a scientific computing environment.

- Get familiarized with Python as a programming language for numerical computation.
- Learn how to solve physics/engineering problems using numerical techniques.
- Can solve demanding tasks with Python.
- Learn to analyze problems, select appropriate numerical algorithms to solve the problem, implement them using Python.
- Possess the basic knowledge of numerical modeling, data analysis and visualizing large amount of data.

1.1.3 Text Book

- Lecture material will be based on them.
- It is strongly advised that student should read textbooks rather than only content with the lecture material supplied from the lecturer.
- Required: No & Recommended:
  - Computational Physics: Problem Solving with Python by by Rubin H. Landau, Manuel J. Páez, Cristian C. Bordeianu, 3rd edition, 2015, Wiley.
  - Learning Scientific Programming with Python by Christian Hill, 2nd edition, 2020, Cambridge University.
  - Fortran ve Python ile Sayısal Fizik by Bekir Karaoğlu, 2nd edition, 2013, Seçkin Yayıncılık.
  - Fizik ve Mühendislikte Python by R. Gökhan Türeci, Hamdi Dağıstanlı, İlkay Türk Çakır, 2021, Cengage Learning.

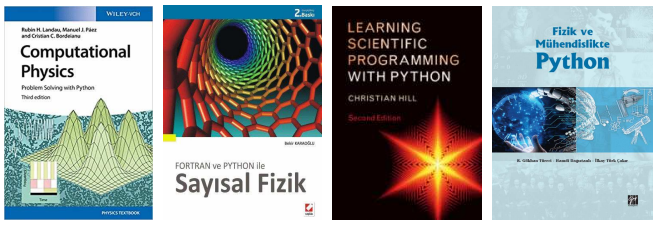


Figure 1.1: Recommended Text Books.

1.1.4 Online Resources

The following (some) resources are available online.

- https://python-course.eu/
- https://www.codecademy.com/catalog/language/python
- https://docs.python.org/
- https://scipy-lectures.org/
- https://matplotlib.org/stable/tutorials/index.html
- https://scipython.com/book2/
- https://pythonnumericalmethods.berkeley.edu/index.html

1.1.5 Grading Criteria

- Midterms & Final Exams: There will be one take-home midterm and one take-home final exam, will count 40% each and 60% of your grade, respectively.
- Homeworks/Assignments (or Term Project): ??

1.1.6 Policies

- Attendance is not compulsory (30%), but you are responsible for everything said in class.
- Academic Regulations:
  Derslere devam zorunluluğu ve denetlenmesi
  MADDE 18 - (1) Öğrencilerin derslere, uygulamalara, sınavlara ve diğer çalışmalara devamı zorunludur. Teorik derslerin % 30'undan, uygulamaların % 20'sinden fazlasına devam etmeyen ve uygulamalarda başarılı olamayan öğrenci, o dersin yarıyıl/yılsonu ya da varsa bütünleme sınavına alınmaz. Tekrarlanan derslerde önceki dönemde devam şartı yerine getirilmiş ise derslerde devam şartı aranıp aranmayacağı ilgili birim tarafından hazırlanarak Senato onayına sunulan usul ve esaslar ile belirlenir.
- You can use ideas from the literature (with proper citation).

- The code you submit must be written completely by you. You can use anything from the textbook/notes.
- I encourage you to ask questions in class. You are supposed to ask questions. Don't guess, ask a question!

1.2 Installation of Required Tools/Programs

1.2.1 Linux System

- Assuming you are using Windows OS.
- Download & Install VirtualBox-7.0.10-158379-Win.exe
- Download & Install kubuntu-22.04.3-desktop-amd64.iso under VirtualBox
- Post-Installation Steps of Kubuntu
  - ping google.com
  - # Setup "Display Configuration" for resolution
  - sudo apt-get install gcc make perl
  - sudo apt-get install python3
  - sudo snap install pycharm-community --classic
  - sudo apt-get install python3-tk
  - sudo apt-get install pyhton3-pip
  - sudo pip install numpy -U
  - sudo pip install scipy -U
  - sudo pip install matplotlib -U
  - # End of Post-Installation Steps of Kubuntu
  - sudo apt-get update # Regular Updates
  - sudo apt-get upgrade # Regular Upgrades
- See video for Installation of Kubuntu & PyCharm under VirtualBox.

1.2.2 Windows System

- Assuming you are using Windows OS.
- Download & Install Anaconda3-2023.09-0-Windows-x86_64.exe
- Download & Install pycharm-community-2023.2.2.exe
- See video for Installation of Anaconda & PyCharm.

1.2.3 Others

- Google Colaboratory and others !
- But, in take-home exams:
  - Prepare your report/codes.
  - Copy your files into a directory named as your ID.
  - Upload/send a single file by compressing this directory.
- Check the web page: IKC-MH.55 2023-2024 Fall frequently.

Chapter 2

Introduction

2.1 Introduction to Python

- Running a Computer Program.

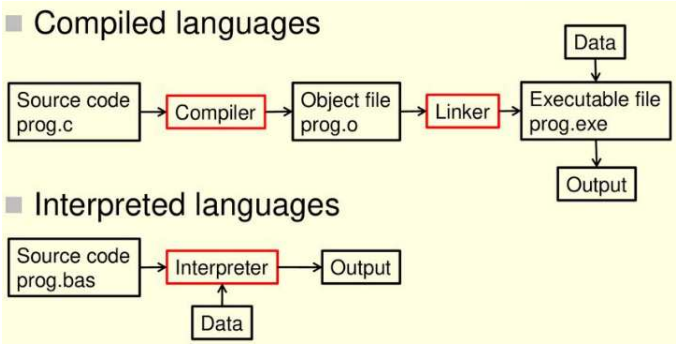


Figure 2.1: Running a Computer Program.

- Python is an interpreted language.
  - The code is pre-processed to produce a bytecode (similar to machine language) and then executed by the interpreter (virtual machine).
- **Code portability:** Runs on hardware/software platforms different from which used to develop the code.
  - Python is portable if the interpreter is available on the target platform.
- **Variables:** A variable stores a piece of data and gives it a name.
  - a variable name must start with a letter or the underscore character;
  - a variable name cannot start with a number;
  - a variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _);
  - white spaces and signs with special meanings, as "+" and "-" are not allowed;
  - variable names are case-sensitive (age, Age and AGE are three different variables).

- **Built-in data types:**
  - **Text Type:** str
  - **Numeric Types:** int, float, complex
  - **Sequence Types:** list, tuple, range
  - **Mapping Type:** dict
  - **Set Types:** set, frozenset
  - **Boolean Type:** bool
  - **Binary Types:** bytes, bytearray, memoryview
- **Lists:** What if we want to store many integers? We need a list!
- **Loops:** Repeat code until a conditional statement ends the loop.
- **Conditionals:** Sometimes you want to execute code only in certain circumstances.
- **Functions:** We can separate off code into functions, that can take input and can give output. They serve as black boxes from the perspective of the rest of our code.

2.2 Python Libraries for Data Science

- Extensive first and third party libraries. Top Python Libraries for Data Science.
  - **NumPy** (aka Numerical Python) is the core numeric and scientific computation library in Python. General-purpose array-processing package.
  - **SciPy** (aka Scientific Python) is extensively used for scientific and technical computations (extends NumPy).
  - **Matplotlib** is an essential library in Python for data visualization in data science. A plotting library.
  - **Seaborn** is another library in Python for data visualization. Extension of Matplotlib. Statistical and graphical analysis in data science.
  - **Pandas** (Python data analysis) is a foundational Python library for data analysis in data science. Data cleaning, data handling, manipulation, and modeling.

- Top Python Libraries for Data Science.
  - **SciKit-Learn** is a robust machine learning library in Python. Data mining, feature engineering, training and deploying machine learning models.
  - **Statsmodels** - provides functionalities for descriptive and inferential statistics for statistical models.
  - **TensorFlow** - a framework for defining and running computations that involve tensors. Machine learning and deep learning framework.
  - **Keras** is a neural network Python library for deep learning model development, training, and deployment.
  - **PyTorch** - scientific computing package that uses the power of graphics processing units.
  - **Scrapy** - for web crawling frameworks.
  - **BeautifulSoup** - for web crawling and data scraping.
  - **NLTK** (Natural Language Tool Kit) is a Python package essentially for natural language processing.

2.2.1 NumPy

Numpy (Numerical Python) - The Fundamental Package for Scientific Computing with Python. https://numpy.org/

- NumPy offers high-quality mathematical functions and supports logical operations on built-in *multi-dimensional array objects*.
- NumPy arrays are significantly faster than traditional Python lists and way more efficient in performance.
- Some of the features provided by NumPy
  - Basic array operations such as addition and multiplication
  - Mathematical, logical, shape manipulation operations
  - Indexing, slicing, flattening, and reshaping the arrays
  - Stacking, splitting, and broadcasting arrays
  - I/O Operations
  - Fourier transform capabilities
  - Basic linear algebra
  - Basic statistical operations

  - Random number generation
  - ...

Sub-Packages	Purpose	Comments
core	basic objects	all names exported to numpy
lib	Addintional utilities	all names exported to numpy
linalg	Basic linear algebra	LinearAlgebra derived from Numeric
fft	Discrete Fourier transforms	FFT derived from Numeric
random	Random number generators	RandomArray derived from Numeric
distutils	Enhanced build and distribution	improvements built on standard distutils
testing	unit-testing	utility functions useful for testing
f2py	Automatic wrapping of Fortran code	a useful utility needed by SciPy

Figure 2.2: NumPy Module Organization.

2.2.2 SciPy

SciPy is a scientific computation library in Python. A collection of mathematical functions and algorithms built on Python's extension NumPy https://scipy.org/.

- It provides the user with high-level commands and classes for manipulating and visualizing data.
- It is widely used in machine learning and scientific programming and comes with integrated support for linear algebra and statistics.
- Some of the features provided by SciPy
  - Search for minima and maxima of functions
  - Calculation of function integrals
  - Support for special functions
  - Signal processing
  - Multi-dimensional image processing

  - Work with genetic algorithms
  - Fourier transform capabilities
  - Solving ordinary differential equations
  - ...

- The scipy namespace itself only contains functions imported from numpy.

Therefore, importing only the scipy base package does only provide numpy content, which could be imported from numpy directly (NOT USED as *import scipy*).
i.e., from scipy import linalg, io

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distribution and function

Figure 2.3: SciPy Modules.

2.2.3 Matplotlib

Matplotlib is the core plotting and data visualization package in Python https://matplotlib.org/.

- A 2D graphical Python library which produces publication quality figures. However, it also supports 3D graphics (mplot3d toolkit), but this is very limited.

- Matplotlib is capable of producing high-quality figures in various formats. It offers interactive cross-platform environments for plotting.

- It provides a MATLAB/Mathematica-like interface for simple plotting pyplot submodule with secondary x-y axis support, and facilitates the creation of subplots, labels, grids, legends, use a logarithmic scale or polar coordinates etc.
  - Matplotlib also allows full control of axes properties, font styles, line and marker styles, and some more formatting entities.
- You can generate line plots (Charts), bar charts, histograms, power spectra, pie charts, error charts, box plots, scatter plots, stem plots, contour plots, etc., with just a few lines of codes in Matplotlib.

2.3 Programming Examples in Python

week2_HandsOn.py:

```
#!/usr/bin/python3
########## Variables ##########
# Each variable in python has a "type". The variable type is not pre-defined, it is "DYNAMICALLY" resolved at run-time
answer = 42   # "answer" contained an integer because we gave it
print(answer) # as an integer!
# 42
is_it_thursday = True   # These both are 'booleans' or true/false
is_it_wednesday = False #  values
pi_approx = 3.1415   # This will be a floating point number
my_string = "Value of pi number" # This is a string datatype
print(pi_approx, my_string)
# 3.1415 Value of pi number
print ("my_string[0]: ", my_string[0]) # Access substrings use []
# my_string[0]:  V
print ("my_string[1:5]: ", my_string[1:5]) # or indices
# my_string[1:5]:  alue
# print(pi_approx + my_string)
## TypeError: unsupported operand type(s) for +: 'float' and 'str'
print(my_string + " in four digits after .")
# Value of pi number in four digits after .
print(type(pi_approx)) # You can get the data type of any object
# <class 'float'>
# Addition, subtraction, multiplication, division are as you expect
float1 = 5.75; float2 = 2.25
print(float1 + float2); print(float1 - float2); print(float1 * float2); print(float1 / float2)
print(5 % 2) # Modulus
```

```
## More Complicated Data Types
# LIST. What if we want to store many integers? We need a list.
prices = [10, 20, 30, 40, 50] # A way to define a list in place
colors = [] #  We can also make an empty list and add to it
colors.append("Green")
colors.append("Blue")
colors.append("Red")
print(colors)
prices.append("Sixty") # We can also add unlike data to a list
print(prices) # Items in a list can be of different type
```

```
print(colors[0]) # Single list elements can be accessed
print(colors[2]) # with the operator [ ]
ourlist = [1, 2, 3, 4, 5] # Basic List Operations
print(ourlist+ourlist) # Concatenation
print(3*ourlist)       # Repetition
multiplied_ourlist = [value * 3 for value in ourlist] # Membership
print(multiplied_ourlist)  # Iteration
#
# TUPLE. A tuple is a sequence ordered data enclosed between ().
tuple1 = "a", "b", "c", "d"
tuple2 = ('physics', 'chemistry', 2022, 2023) # Data heterogeneous
tuple3 = (1, 2, 3, 4, 5 )
ikc_muh_55_info = ("IKC-MH", "55", 2023, "February", 28)
print("tuple1[0]: ", tuple1[0])     # access to single element
print("tuple2[1:5]: ", tuple2[1:5]) # access to slice
print(ikc_muh_55_info[0] + "." + ikc_muh_55_info[1])
print(tuple3)
```

```
# DICTIONARY. An unordered collection of items
person = {"name": "Mehmet", "age": 19}
print(f"{person['name']} is {person['age']} years old.")
print(person["name"])
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
    print(squares[i])
#
######### Loops in Python #########
# Repeat code until a conditional statement ends the loop
# WHILE.
list = [1, 1, 2, 3, 5, 8]
print(list)
print("i", "list[i]")
i = 0
while (i < len(list)): # While loops are the basic type
    print(i, list[i])
    i = i + 1
#
# FOR. The 'for' loop is the way to write it faster.
for i in range(0, len(list)):
    print(i, list[i])
# Or you can do so even neater
for e in list:
    print(e)
```

```
######### Conditionals in Python #########
# Sometimes you want to execute code only in certain circumstances
answer = 42 # Change answer and see what code is executed
if answer == 42:
    print('This is the answer to the ultimate question')
elif answer < 42:
    print('This is less than the answer to the ultimate question')
else:
    print('This is more than the answer to the ultimate question')
print('This print statement is run no matter what because it is not indented
    !')
# Using boolean operations. Question: How long does it take me to get to
    work?
snowy = True
day = "Monday"
rainy = True
if (snowy == False) and (day != "Monday"): # "and" is boolean and. True only
     if both are true. False otherwise
```

```
    time = 7
elif (snowy == True) and (day == "Monday"):
    time = 11
elif (rainy == True) or (day == "Monday"):
    time = 9
print("It takes me %d minutes" % (time))
```

```
# while & if statements example
number = 23
running = True
while running:
    guess = int(input('Enter an integer : '))
    if guess == number:
        print('Congratulations, you guessed it.')
        running = False # this causes the while loop to stop
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
print('Done') # Do anything else you want to do here
#
######### Functions in Python #########
# A function is a block of code which only runs when it is called and can be
    run repetitively.
# use the def keyword, and indent because this creates a new block
def print_me(string): # Function definition is here
    "This prints a passed string into this function" # The first statement
    of a function can be an optional statement
    print(string)
    return     # End with the "return" keyword
print_me("I'm first call to user defined function!") # Function call
print_me("Again second call to the same function") # Function call
```

```
def changelist( mylist ):
    """This changes a passed list into this function"""
    mylist = [1,20,3,4] # Call by  Value in Python. This assigns new
    reference in mylist
                  # Call by Reference when commenting this line
    print("Address inside the function: ", id(mylist))
    mylist.append(9)
    mylist[0]=7
    mylist.remove(20)
    print("Values inside the function: ", mylist)
mylist = [10,20,30] # Function call
print("Initial values outside the function: ", mylist)
print("Address outside the function: ", id(mylist))
changelist( mylist )
print("Values outside the function: ", mylist)
# Initial values outside the function:  [10, 20, 30]
# Address outside the function:  140390909223488 # Call by  Value
# Address inside the function:  140390919434048  # Call by  Value
# Values inside the function:  [7, 3, 4, 9]  # Call by  Value
# Values outside the function:  [10, 20, 30] # Call by  Value
#
# Initial values outside the function:  [10, 20, 30]
# Address outside the function:  140390919434048 # Call by Ref.
# Address inside the function:  140390919434048  # Call by Ref.
# Values inside the function:  [7, 30, 9]  # Call by Reference
# Values outside the function:  [7, 30, 9] # Call by Reference
```

```
def step(x): # Your functions can return data if you so choose
    if (x < 0):
        return -1
    elif (x > 0):
        return 1
print(step(-1)) # call functions by repeating their name, and putting your variable in the parenthesis
print(step(1))  # Your variable need not be named the same thing, but it should be the right type
# what happens for x = 0?
print(step(0)) # Python automatically adds in a "return none" statement if you are missing one.
#
# Fix the return none issue
def step_v2(x):
    if (x < 0):
        return -1
    elif (x >= 0):
        return 1
print(step_v2(0))
```

```
######### Importing in Python #########
# Just about every standard math function on a calculator has a python equivalent pre-made.
import math
float1 = 5.75
float2 = 2.25
print(math.log(float1)); print(math.exp(float2)); print(math.pow(2,5))
print(2.0**5.0) # There is a quicker way to write exponents
#
######### Numpy - "The Fundamental Package for Scientific Computing with Python" #########
# numpy has arrays, which function similarly to Python lists.
#
# Generally, it is used a convention on names used to import packages (such as numpy, scipy, and matplotlib)
# import [package] as [alias]
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
#
# Generally scipy is not imported as module because interesting functions in scipy are actually located in the submodules, so submodules or single functions are imported
# from [package] import [module] as [alias]
from scipy import fftpack
from scipy import integrate
```

```
import numpy as np # Here, we grab all of the functions and tools from the numpy package and store them in a local variable called np.
l = [1,2,3,4] # python list
print(l)
l_np = np.array(l)
print(l_np)
print(l*5)    # multiplying a python list replicates it
print(l_np*5) # numpy applies operation elementwise
#
## 1D array
a1 = np.array([1,2,3,4]) # initialized with a numpy list. Be careful with syntax. The parentheses and brackets are both required
print(a1)
print(a1.shape)  # shape indicates the rank of the array
```

```
#
## Rank 2 array
# row vector
a2 = np.array([[1,2,3,4]])
print(a2)
print(a2.shape)  # shape indicates the rank of the array. this looks more like a row vector
# column vector
a3 = np.array([[1],
               [2],
               [3],
               [4]])
print(a3)
print(a3.shape)  # this looks more like a column vector
```

```
import numpy as np
a = np.array([0, 10, 20, 30, 40])
print(a)
print(a[:])
print(a[1:3])
a[1] = 15
print(a)
b = np.arange(-5, 5, 0.5)
print(b)
print(b**2)
1/b
# <input>:1: divide by zero encountered in true_divide
1/b[10]
# <input>:1: divide by zero encountered in double_scalars
#
## Element-wise operations
a = np.array([1, 2, 3])
b = np.array([9, 8, 7])
print(a)
print(a.shape); print(b.shape)
print(a[0]) # Access elements from them just like a list
#
# Element-wise operations. This is different from MATLAB where you add a dot to get element wise operators.
c = a + b
d = a - b
e = a * b
f = a / b
print(c); print(d); print(e); print(f)
```

```
import numpy as np
# What about multi-dimensional arrays? Matrices! You just nest lists within lists
A = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]]); print(A)
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]
print(A.shape)
# (3, 3)
B = np.array([[1, 1, 1],[2, 2, 2],[3, 3, 3]]); print(B)
C = np.matmul(A, B); print(C) # Then matrix multiplication
print(np.linalg.det(A)) # Or determinants
#
import numpy as np
p = np.poly1d([3,4,5])
print(p)
#    2
```

```
# 3 x + 4 x + 5
print(p*p)
#     4       3       2
# 9 x + 24 x + 46 x + 40 x + 25
print(p.integ(k=6))
#     3      2
# 1 x + 2 x + 5 x + 6
print(p.deriv())
# 6 x + 4
p([4, 5])
# array([ 69, 100])
```

```
##### SciPy is a scientific computation library in Python #####
# A collection of functions to perform basic scientific programming and data analysis
# Integrate a list of numbers using scipy you might use a function called trapz from the integrate package
import scipy.integrate as integ
# from scipy import integrate as integ
result = integ.trapz([0, 1, 2, 3, 4, 5])
print(result)
# Integrate sin(x) from 0 to Pi you could use the quad function
import numpy as np
import scipy.integrate as integ
result = integ.quad(np.sin, 0, np.pi)
print(result)
#
######### Matplotlib #########
# Matplotlib, like many Python packages, is organized into a number of "modules" (essentially subsets of functions).
import matplotlib.pyplot as plt # import packages with alias
# from matplotlib  import pyplot as plt
x_vals = [-2, -1, 0, 1, 2]
y_vals = [-4, -2, 0, 2, 4]
print(x_vals,y_vals)
plt.xlabel('abscissa') # add a label to the x axis
plt.ylabel('ordinate') # add a label to the y axis
plt.title('A practice plot') # add a title
plt.plot(x_vals, y_vals, marker="o")
plt.savefig('plot_0.png') # save the figure to the current diretory as a png file
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np
t = np.arange(0.0, 2.0, 0.01)
print(t)
print(t.shape)
print(len(t))
s = 1 + np.sin(2*np.pi*t) # Degree to Radian conversion
print(s)
plt.plot(t, s)
plt.xlabel('Time (s)'); plt.ylabel('Voltage (mV)')
plt.title('Voltage vs Time')
plt.grid(True)
plt.savefig("test.png")
plt.show()
#
# Multiplotting
import numpy as np
import matplotlib.pyplot as plt
x1 = np.linspace(0.0, 5.0); x2 = np.linspace(0.0, 2.0)
```

```
y1 = np.cos(2*np.pi*x1)*np.exp(-x1); y2 = np.cos(2*np.pi*x2)
plt.subplot(2, 1, 1) # use the subplot function to generate multiple panels within the same plotting window
plt.plot(x1, y1, 'o-')
plt.title(' 2 subplots')
plt.ylabel('Damped oscillation')
plt.subplot(2, 1, 2)
plt.plot(x2, y2, '.-')
plt.xlabel('time (s)'); plt.ylabel('Undamped')
plt.show()
```

```
# importing all functions from pylab module
from pylab import * # Not the preferred methodology. Means to bring everything in to the top level name space
x = arange(1, 10, 0.5); print(x)
xsquare = x**2; print(xsquare)
xcube = x**3; print(xcube)
xsquareroot = x**0.5; print(xsquareroot)
figure(1)           # open figure 1
plot(x, xsquare)    # basic plot
xlabel('abscissa') # add a label to the x axis
ylabel('ordinate') # add a label to the y axis
title('A practice plot') # add a title
savefig('plot_1.png') # save the figure to the current diretory
figure(2)           # open a second figure
plot(x, xsquare,'ro',x,xcube,'g+--') # Two plots. Red circles with no line. Green plus signs joined by a dashed curve
xlabel('abscissa') # x and y labels, title
ylabel('ordinate') # x and y labels, title
title('More practice') # x and y labels, title
legend(('squared', 'cubed')) # add a legend
savefig('plot_2.png') # save the figure
figure(3)           # open a third figure
subplot(3,1,1); plot(x,xsquareroot,'k*:')# Black stars+dotted line
title('square roots') # add a title
subplot(3,1,2); plot(x, xsquare,'r>-') # Red triangles+dashed line
title('squares') # add a title
subplot(3,1,3); plot(x, xcube,'mh-') # Magenta hexagons+solid line
title('cubes') # add a title
savefig('plot_3.png') # save the figure
show()
```

```
######### How to find documentation #########
# The dir(module) function can be used to look at the namespace of a module or package, i.e. to find out names that are defined inside the module
# The help(function) function is available for each module/object and allows to know the documentation for each module or function
#
import math
dir()
dir(math.acos)
help(math.acos)
import matplotlib.pyplot
dir(matplotlib.pyplot)
```

Some links to study python.

- https://python-course.eu/
- https://www.codecademy.com/catalog/language/python

- https://scipy-lectures.org/
- https://computation.physics.utoronto.ca/tutorials/
- https://moodle2.units.it/course/view.php?id=6837
- https://jckantor.github.io/CBE30338/
- https://matplotlib.org/stable/tutorials/index.html

2.4 Numerical Fundamentals

1. to solve problems that may not be solvable by hand.
2. to solve problems (that you may have solved before) in a different way.

- Many of these simplified examples can be solved analytically (by hand)

$$x^3 - x^2 - 3x + 3 = 0, with\ solution\ \sqrt{3}$$

- But most of the examples can not be simplified and can not be solved analytically.
- Mathematical relationships $\Longrightarrow$ simulate some real word situations.

2.4.1 Analysis vs Numerical Analysis

- In mathematics, solve a problem through equations; **algebra, calculus, differential equations (DE), Partial DE,** ...
- In numerical analysis; four operations (add, subtract, multiply, division) and Comparison.
  - These operations are exactly those that computers can do

$$\int_0^{\pi} \sqrt{1 + cos^2 x} dx$$

    * length of one arch of the curve y-sinx; no solution with "a substitution' or "integration by parts"
    * numerical analysis can compute the length of this curve by standardised methods that apply to essentially any integrand

- Another difference between a numerical results and analytical answer is that the former is always an approximation
  - this can usually be as accurate as needed (level of accuracy)
- Numerical Methods require repetitive arithmetic operations $\Rightarrow$ a computer to carry out
- Also, a human would make so many mistakes

2.4.2 Some disasters attributable to bad numerical computing

Have you been paying attention in your numerical analysis or scientific computation courses? Here are some real life examples of what can happen when numerical algorithms are not correctly applied.

- The Patriot Missile failure, in Dharan, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to ***poor handling of rounding errors***.
- The explosion of the Ariane 5 rocket just after lift-off on its maiden voyage off French Guiana, on June 4, 1996, was ultimately the ***consequence of a simple overflow***.
- The sinking of the Sleipner A offshore platform in Gandsfjorden near Stavanger, Norway, on August 23, 1991, resulted in a loss of nearly one billion dollars. It was found to be the ***result of inaccurate finite element analysis***.

2.4.3 Floating-Point Arithmetic

- Performing an arithmetic operation $\Rightarrow$ no exact answers unless only integers or exact powers of 2 are involved,
- Floating-point (real numbers)$\rightarrow$ not integers,
- Resembles scientific notation,
- IEEE standard $\rightarrow$ storing floating-point numbers (see the Table 2.1).

Table 2.1: Floating$\rightarrow$ Normalised.

floating	normalised (shifting the decimal point)
13.524	$.13524 * 10^2$ $(.13524E2)$
-0.0442	$-.442E-1$

- the sign $\pm$
- the fraction part (called the *mantissa*)
- the exponent part

- What about the sign of the exponent? Rather than use one of the bits for the sign of the exponent, exponents are biased.
- For **single** precision (we have 8 bits reserved for the exponent):
  - 2^8=256
  - 0$\longrightarrow$00000000 = 0
  - 255$\longrightarrow$11111111=255
  - 0 (255)$\Longrightarrow$ -127 (128). An exponent of -127 (128) stored as 0 (255).
  - So biased$\longrightarrow$ $2^{128} = 3.40282E + 38$, mantissa gets 1 as maximum
  - **Largest:** 3.40282E+38; **Smallest:** 5.87747E-39 (!)
  - For **double** and **extended** precision the bias values are 1023 and 16383, respectively.
  - $\frac{0}{0}, 0 * \infty, \sqrt{-1} \Longrightarrow NaN$: Undefined.

There are three levels of precision (see the Fig. 2.4)

		Number of bits in			
Precision	Length	Sign	Mantissa	Exponent	Range
Single	32	1	23(+1)	8	$10^{\pm 38}$
Double	64	1	52(+1)	11	$10^{\pm 308}$
Extended	80	1	64	15	$10^{\pm 4931}$

Figure 2.4: Level of precision.

week3_HandsOn.py

```
import sys
print(sys.float_info)
# sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
    min=2.2250738585072014e-308, min_exp=-1021,
# min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix
    =2, rounds=1)
```

```
print(sys.float_info.max)
# 1.7976931348623157e+308
print("%10.6e " % 2**128)
# 3.402824e+38
print("%10.6e " % 2**1023)
# 8.988466e+307f
```

EPS: short for epsilon⟶used for represent the smallest machine value that can be added to 1.0 that gives a result distinguishable from 1.0!

- $eps \longrightarrow \varepsilon \Longrightarrow (1+\varepsilon)+\varepsilon = 1$ *but* $1+(\varepsilon+\varepsilon) > 1$
- Two numbers that are very close together on the *real* number line can not be distinguished on the *floating-point* number line if their difference is less than the least significant bit of their mantissas.

```
import sys
print(sys.float_info.epsilon)
# 2.220446049250313e-16
eps=sys.float_info.epsilon
print(1+eps*0.5)
# 1.0
print(1+eps*0.5)
# 1.0
print((1+eps*0.5)+eps*0.5)
# 1.0
print(1+eps*0.6)
# 1.0000000000000002
```

2.4.4 Computer Number Representation

Say we have six bit representation (not single, double) (see the Fig.)

- 1 *bit* → *sign*
- 3(+1) *bits* → *mantissa*
- 2 *bits* → *exponent*

Sign	Mantissa	Exponent	Value
0	(1)001	00	$9/16 * 2^{-1} = +9/32$
0	(1)111	11	$15/16 * 2^{2} = +15/4$

Figure 2.5: Computer numbers with six bit representation.

- For positive range $\frac{9}{32} \longleftrightarrow \frac{15}{4}$
- For negative range $\frac{-15}{4} \longleftrightarrow \frac{-9}{32}$; even discontinuity at point zero since it is not in the ranges.

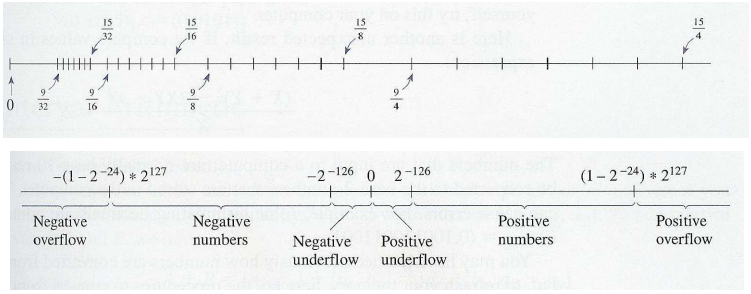


Figure 2.6: Upper: number line in the hypothetical system, Lower: IEEE standard.

- Very simple computer arithmetic system $\Rightarrow$ the gaps between stored values are very apparent.
- Many values can not be stored exactly. i.e., 0.601, it will be stored as if it were 0.6250 because it is closer to $\frac{10}{16}$, an error of 4%
- In IEEE system, gaps are much smaller but they are still present. (see the lower Fig. 2.6)

2.4.5 Kinds of Errors in Numerical Procedures

Computers use only a fixed number of digits to represent a number.

- As a result, the numerical values stored in a computer are said to have *finite precision*.
- Limiting precision has the desirable effects of increasing the speed of numerical calculations and reducing memory required to store numbers.
- But, what are the undesirable effects?

Kinds of Errors:

i **Round-off Error**

ii **Truncation Error**

iii **Propagated Error**

i **Round-off Error**:

```
#!/usr/bin/python3
x=(4/3)*3; print(x)
# 4.0
a=4/3; print(a) # store double precision approx of 4/3
# 1.3333333333333333
b=a-1; print(b) # remove most significant digit
# 0.33333333333333326
c=1-3*b; print(c) # 3*b=1 in exact math
# 2.220446049250313e-16 # should be 0!!
```

```
from math import *
# import numpy as np
# kfirst=1.0; klast=360.0; kincrement=0.1
# for j in np.arange(kfirst, klast + kincrement, kincrement):
for j in range(1,360): # In degrees (1-360) as int. increment
    jj=j*(2*pi/360) # Conversion to radian
    a=cos(jj) # Return the cosine of jj (measured in radians)
    b=sin(jj) # Return the cosine of jj (measured in radians)
    z=a-(a/b)*b # Expected as being 0 !!
    print(j,jj,z)
352 6.14355896702004 0.0
353 6.161012259539984 1.1102230246251565e-16
354 6.178465552059927 1.1102230246251565e-16
355 6.19591884457987 1.1102230246251565e-16
356 6.213372137099813 0.0
357 6.230825429619756 0.0
358 6.2482787221397 0.0
359 6.265732014659643 0.0
```

```
summation=1.0
for i in range(10000):  # Adding 0.00001 to 1.0 as 10000 times
    summation=summation+0.00001
print('summation = ', summation)
# summation =  1.1000000000006551 # Expected result is just 1.1 !!
print("summation = %f" % summation)
# summation = 1.100000 # Now expected result??
```

To see the effects of roundoff in a simple calculation, one need only to *force the computer to store the intermediate* results.

- All computing devices represents numbers, except for integers and some fractions, with some **imprecision.**
- Floating-point numbers of fixed word length; the true values are usually not expressed exactly by such representations.

```
import numpy as np
x=np.tan(np.pi/6);  print(x)
# 0.5773502691896257
y=np.sin(np.pi/6)/np.cos(np.pi/6); print(y)
```

```
# 0.5773502691896256
if x==y:
    print("x and y are equal ")
else:
    print("x and y are not equal : x-y=%e " % (x-y))
# x and y are not equal : x-y=1.110223e-16
```

- The test is true only if x and y are exactly equal in bit pattern.
- Although x and y are equal in exact arithmetic, their values differ by a small, but nonzero, amount.
- When working with floating-point values the question "are x and y equal?" is replaced by "are x and y close?" or, equivalently, "is $x - y$ small enough?"

ii **Truncation Error**: i.e., approximate e^x by the cubic power

$$P_3(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}; \qquad e^x = P_3(x) + \sum_{n=4}^{\infty} \frac{x^n}{n!}$$

- Approximating e^x with the cubic gives an inexact answer. The error is due to truncating the series,
- When to cut series expansion $\Longrightarrow$ be satisfied with an approximation to the exact analytical answer.
- Unlike roundoff, which is controlled by the hardware and the computer language being used, truncation error is under control of the programmer or user.
- Truncation error can be reduced by selecting more accurate discrete approximations. But, it can not be eliminated entirely.

Evaluating the Series for sin(x) (**Example py-file:** sinser.py)

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

- An efficient implementation of the series uses recursion to avoid overflow in the evaluation of individual terms. If T_k is the k^{th} term ($k = 1, 3, 5, \ldots$) then

$$T_k = \frac{x^2}{k(k-1)} T_{k-2}$$

- Study the effect of the parameters *tol* and *nmax* by changing their values (Default values are 5e-9 and 15, respectively).

```
import numpy as np
def sinser(x,tol,n):
    term = x
    ssum = term #  Initialize series
    print("Series approximation to sin(%f) \n  k       term          ssum" % (
    x*360/(2*np.pi)))
    print("  1  %11.3e  %20.16f " % (term,ssum))
    for k in range(3, 2*n-1, 2):
        term = -term * x*x/(k*(k-1)) #  Next term in the series
        ssum = ssum + term
        print("%3d  %11.3e  %30.26f " % (k,term,ssum))
        if abs(term/ssum)<tol:
            break  # True at convergence
    print("Truncation error after %d terms is %g " %  ((k+1)/2,abs(ssum-np.
    sin(x))))
sinser(np.pi/6,5e-9,10)
print("sin(%f)=%f with numpy library " % (np.pi/6*360/(2*np.pi),np.sin(np.pi
    /6)))
import math
print("sin(%f)=%f with math libarary" % (math.pi/6*360/(2*math.pi),math.sin(
    math.pi/6)))
```

```
Series approximation to sin(30.000000)
  k       term          ssum
  1    5.236e-01    0.5235987755982988
  3   -2.392e-02    0.49967417939436375995398976
  5    3.280e-04    0.50000213258879244726529123
  7   -2.141e-06    0.49999999186902321923753334
  9    8.151e-09    0.50000000002027988887931542
 11   -2.032e-11    0.49999999999996430632975830
Truncation error after 6 terms is 3.56382e-14
sin(30.000000)=0.500000 with numpy library
sin(30.000000)=0.500000 with math libarary
```

Figure 2.7: Output of sinser.py

Derivative of Sine function. Truncation & Round-off Errors (**Example py-file:** trunroun.py)

```
n    Step Length    Derivative     Error
1 0.10000000000000 0.4973637525 0.0429385533
2 0.01000000000000 0.5360859810 0.0042163249
3 0.00100000000000 0.5398814804 0.0004208255
4 0.00010000000000 0.5402602314 0.0000420744
5 0.00001000000000 0.5402980985 0.0000042074
6 0.00000100000000 0.5403018851 0.0000004207
7 0.00000010000000 0.5403022640 0.0000000418
8 0.00000001000000 0.5403023029 0.0000000030
9 0.00000000100000 0.5403023584 -0.0000000525
10 0.00000000010000 0.5403022474 0.0000000585
11 0.00000000001000 0.5403011372 0.0000011687
12 0.00000000000100 0.5403455461 -0.0000432402
13 0.00000000000010 0.5395683900 0.0007339159
14 0.00000000000001 0.5440092821 -0.0037069762
```

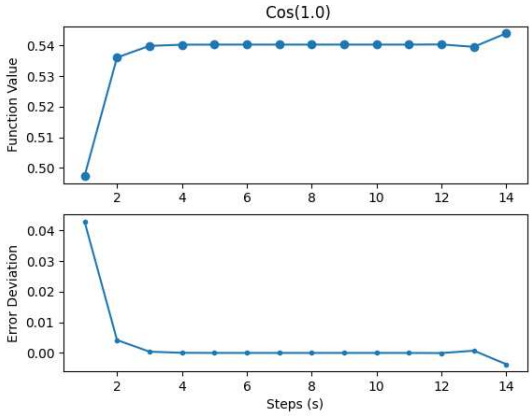


Figure 2.8: Output and Plot of trunroun.py

iii **Propagated Error**:

- more subtle (difficult to analyse)
- by propagated we mean an error in the succeeding steps of a process due to an occurrence of an earlier error
- of critical importance
- stable numerical methods; errors made at early points die out as the method continues
- unstable numerical method; does not die out

2.4.6 Absolute vs Relative Error, Convergence

- Accuracy (how close to the true value) $\longrightarrow$ great importance,
- $absolute\ error = |true\ value - approximate\ error|$
  A given size of error is usually more serious when the magnitude of the true value is small,
- $relative\ error = \frac{absolute\ error}{|true\ value|}$
- **Convergence of Iterative Sequences**:

– Iteration is a common component of numerical algorithms. In the most abstract form, an iteration generates a sequence of scalar values $x_k, k = 1, 2, 3, \ldots$. The sequence converges to a limit ξ if

$$|x_k - \xi| < \delta, \ for\ all\ k > N$$

where δ is a small number called the convergence tolerance. We say that the sequence has converged to within the tolerance δ after N iterations.

```
def newtsqrt(x,delta,maxit):
    r = x/2;  rold = x # Initialize, make sure convergence test fails on
    first try
    it = 0
    while (r!=rold) and (it<maxit): # Convergence test
    # while ((r-rold) > delta) and (it<maxit): # Convergence test
    # while (abs(r-rold) > delta) and (it<maxit): # Convergence test
    # while (abs((r-rold)/rold) > delta) and (it<maxit): # Convergence test
        rold = r # Save old value for next convergence test
        r = 0.5*(rold + x/rold) # Update the guess
        it = it + 1
    return r
# Test the newtsqrt function for a range of inputs
xtest = [4, 0.04, 4e-4, 4e-6, 4e-8, 4e-10, 4e-12] # arguments to test
print(" Absolute Convergence Criterion")
print("    x         sqrt(x)   newtsqrt(x)    error       relerr")
import math
for x in  xtest: # repeat for each elementin xtest
    r = math.sqrt(x)
    rn = newtsqrt(x,5e-9,25)
    err = abs(rn - r)
    relerr = err/r
    print("%10.3e  %10.3e  %10.3e  %10.3e  %10.3e" % (x,r,rn,err,relerr))
```

```
Absolute Convergence Criterion
   x         sqrt(x)   newtsqrt(x)    error       relerr
4.000e+00  2.000e+00  2.000e+00  0.000e+00  0.000e+00
4.000e-02  2.000e-01  2.000e-01  0.000e+00  0.000e+00
4.000e-04  2.000e-02  2.000e-02  0.000e+00  0.000e+00
4.000e-06  2.000e-03  2.000e-03  0.000e+00  0.000e+00
4.000e-08  2.000e-04  2.000e-04  2.711e-20  1.355e-16
4.000e-10  2.000e-05  2.000e-05  3.388e-21  1.694e-16
4.000e-12  2.000e-06  2.000e-06  0.000e+00  0.000e+00
```

Figure 2.9: Output of newtsqrt.py

Newton's method to compute the square root of a number.
Example py-file: newtsqrt.py

Chapter 3

Root Searching

3.1 Solving Nonlinear Equations

- Solve "$f(x) = 0$"
  - where $f(x)$ is a function of x.
  - The values of x that make $f(x) = 0$ are called the roots of the equation.
- The following non-linear equation can compute the friction factor,f:

$$\frac{1}{\sqrt{f}} = \left(\frac{1}{k}\right) ln(RE\sqrt{f}) + \left(14 - \frac{5.6}{k}\right)$$

- The equation for f is not solvable except by the numerical procedures.

1. **Interval Halving (Bisection)**. Describes a method that is very simple and foolproof but is not very efficient. We examine how the error decreases as the method continues.
2. **Linear Interpolation Methods**. Tells how approximating the function in the vicinity of the root with a straight line can find a root more efficiently. It has a better "rate of convergence".
3. **Newton-Raphson Method**. Explains a still more efficient method that is very widely used but there are pitfalls that you should know about. Complex roots can be found if complex arithmetic is employed.

3.1.1 Blackbody Radiation

- Some observations in physics experiments at the beginning of the 20^{th} century could not be explained by classical theories (Blackbody Radiation, The Photoelectric Effect, The Hydrogen Atom, ...).
- Among them, the phenomenon called **Blackbody Radiation** has a special place.
- By definition, a blackbody is an object that absorbs any heat radiation falling on it.
- Rayleigh and Jean proposed that infinitesimal amounts of energy were continuously added to the system when the frequency was increased.
- **Classical physics assumed that energy emitted by atomic oscillations could have any continuous value**.

- If we try and sum the energies at each frequency we find that there is an **infinite** energy in this system!
- This paradox was called the ULTRAVIOLET CATASTROPHE.

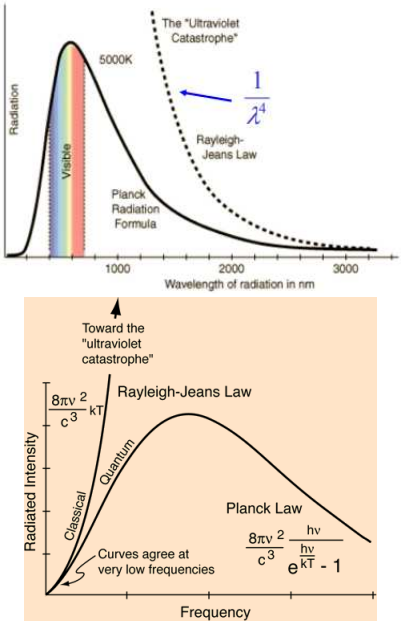


Figure 3.1: Variation of energy density with wavelength/frequency in blackbody radiation.

- Spectrum of the radiation inside the blackbody can be measured experimentally.

$$dU = u(\lambda, T)d\lambda$$

- The experimentally observed curve $u(\lambda, T)$ at T=5000 K is shown in the upper figure.
- It has not been possible to explain this observed spectrum of blackbody radiation with classical theories (in the classical limit of large λ, Rayleigh-Jeans Law).

- Later, in 1901, Planck was able to come up with the formula that fully explained this curve, with an assumption that predicted the *quantum nature of light*:

$$u(\lambda, T) = \frac{8\pi hc}{\lambda^5} \frac{1}{e^{hc/\lambda k_B T} - 1} \tag{3.1}$$

  Here c is the speed of light and $h = 6.63 \times 10^{-34}\ joule.s$ becomes a new constant in physics by the name of Planck's constant.

- Planck's assumption was as follows: **The energy was quantized and could be emitted or absorbed only in integral multiples of a small unit of energy, known as a quantum.**
- The energy distribution of a radiation with a frequency $\nu = hc/\lambda$ is a multiple of an amount of $h\nu$:

$$E = nh\nu \quad (n = 1, 2, 3, \ldots)$$

- Blackbody radiation helped to move our understanding in physics from a classical approach to a quantum one.

- Some notable features of the Planck distribution are:

  1. Total radiant energy (ie the area under the curve in the Figure)

  $$\int_0^\infty u(\lambda, T)d\lambda = \sigma T^4$$

  is proportional to the 4^{th} power of the temperature T. This is called the **Stefan-Boltzmann formula**.

  2. There is a simple relation between the wavelength λ_{max} at which each curve has a maximum value and the equilibrium temperature T

  $$\lambda_{max} T = 0.0029\ m.K$$

  This equation is called the **Wien's displacement law**.

- Here we will obtain the Wien's displacement law by numerical method.

- To find the maximum wavelength, it is sufficient to take the derivative of Equation 3.1 with respect to the wavelength λ and set it to zero. However, it is convenient to do this based on the dimensionless variable $x = hc/k_B T\lambda$:

$$u(x) = A\frac{x^5}{e^x - 1} \quad (x = hc/k_B T\lambda \ and \ A = 8\pi(k_B T)^5/(hc)^4)$$

$$\frac{du}{dx} = A\frac{5x^4(e^x - 1) - x^5 e^x}{(e^x - 1)^2} = 0$$

$$(5 - x) - 5e^{-x} = 0 \qquad (3.2)$$

- If this equation is solved, x_{max} and then λ_{max} can be found. However, Equation 3.2 has no analytical solution.

- We can find the answer with the numerical root finding methods.

3.1.2 Interval Halving (Bisection)

- Interval halving (bisection), an ancient but effective method for finding a zero of $f(x)$.
- It begins with two values for x that bracket a root.
- **The function $f(x)$ changes signs at these two x-values** and, if $f(x)$ is continuous, there must be at least one root between the values.
- The test to see that $f(x)$ does change sign between points a and b is to see if $f(a) * f(b) < 0$ (see Fig.).

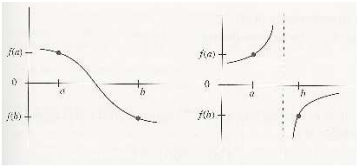


Figure 3.2: Testing for a change in sign of f(x) will bracket either a root or singularity.

The bisection method then

- successively divides the initial interval in half,
- finds in which half the root(s) must lie,
- and repeats with the endpoints of the smaller interval.

- A plot of $f(x)$ is useful to know where to start.
- **Example**: The function; $f(x) = 3x + sin(x) - e^x$
- Look at to the plot of the function to learn where the function crosses the x-axis.

```
1  import numpy as np
2  from math import *
3  def f(x):
4      return 3*x+sin(x)-exp(x)
5  xval=[]
6  funval=[]
7  kfirst=0.0; klast=2.0; kincrement=0.01
8  for j in np.arange(kfirst, klast +
       kincrement, kincrement):
9      xval.append(j)
10     funval.append(f(j))
11 import matplotlib.pyplot as plt
12 plt.title('$f(x)=3x+sin(x)-e^x$')
13 plt.xlabel('X Value')
14 plt.ylabel('Function Value')
15 plt.plot(xval, funval, '-')
16 plt.grid()
17 plt.savefig('function_plot.eps')
18 plt.show()
```

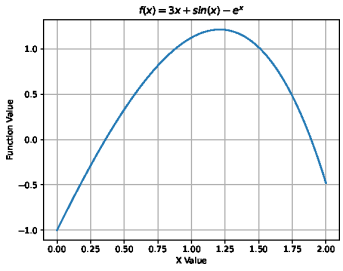


Figure 3.3: Code and plot of the function: $f(x) = 3x + sin(x) - e^x$

- We see from the figure that indicates there are zeros at about $x = 0.35$ and 1.9.

```
Tolerance values in x and f(x)  are 1.110223e-15 and 1.110223e-15.
Bisection iterations for function: 3x+sin(x)-e^x
   k             a                   xm                  b                fm
   1     0.0000000000000000    0.5000000000000000    1.0000000000000000  3.3070426790e-01
   2     0.0000000000000000    0.2500000000000000    0.5000000000000000 -2.8662145743e-01
   3     0.2500000000000000    0.3750000000000000    0.5000000000000000  3.6281114468e-02
   4     0.2500000000000000    0.3125000000000000    0.3750000000000000 -1.2189942659e-01
   5     0.3125000000000000    0.3437500000000000    0.3750000000000000 -4.1955965903e-02
   6     0.3437500000000000    0.3593750000000000    0.3750000000000000 -2.6196345703e-03
   7     0.3593750000000000    0.3671875000000000    0.3750000000000000  1.6885752947e-02
   8     0.3593750000000000    0.3632812500000000    0.3671875000000000  7.1467416292e-03
   9     0.3593750000000000    0.3613281250000000    0.3632812500000000  2.2669653023e-03
  10     0.3593750000000000    0.3603515625000000    0.3613281250000000 -1.7548279455e-04

  40     0.3604217029587744    0.3604217029596839    0.3604217029605934 -1.6024959137e-12
  41     0.3604217029596839    0.3604217029601386    0.3604217029605934 -4.6473935811e-13
  42     0.3604217029601386    0.3604217029603660    0.3604217029605934  1.0413891971e-13
  43     0.3604217029601386    0.3604217029602523    0.3604217029603660 -1.8030021920e-13
  44     0.3604217029602523    0.3604217029603092    0.3604217029603660 -3.8191672047e-14
  45     0.3604217029603092    0.3604217029603376    0.3604217029603660  3.3084646134e-14
  46     0.3604217029603092    0.3604217029603234    0.3604217029603376 -2.6645352591e-15
  47     0.3604217029603234    0.3604217029603305    0.3604217029603376  1.5321077740e-14
  48     0.3604217029603234    0.3604217029603269    0.3604217029603305  6.4392935428e-15
  49     0.3604217029603234    0.3604217029603252    0.3604217029603269  1.9984014443e-15
  50     0.3604217029603234    0.3604217029603243    0.3604217029603252 -2.2204460493e-16
Mybisect: Root is found as     0.3604217029603243 within tolerance after 50 iterations
SciPy bisect: Root is          0.3604217029587744. Accept that value as exact!
```

- Apply Bisection method to $f(x) = 3x + sin(x) - e^x$. (**Example py-file:** mybisect.py)

The root is (almost) never known exactly, since it is extremely unlikely that a numerical procedure will find the precise value of x that makes $f(x)$ exactly zero in floating-point arithmetic. Bisection is generally recommended

- The main advantage of interval halving is that it is **guaranteed to work** (continuous & bracket).
- The algorithm must decide how close to the root the guess should be before stopping the search (see Fig.).
- The major objection of interval halving has been that it is **slow to converge**.

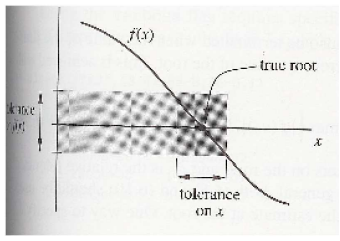


Figure 3.4: The stopping criterion for a root-finding procedure should involve a tolerance on x, as well as a tolerance on $f(x)$.

for finding an approximate value for the root, and then this value is refined by more efficient methods.

3.1.3 Linear Interpolation Methods-The Secant Method

- Bisection is simple to understand but it is not the most efficient way to find where $f(x)$ is zero.
- Most functions can be approximated by a straight line over a small interval.

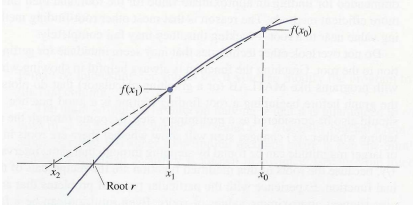


Figure 3.5: Graphical illustration of the Secant Method.

- The secant method begins by finding *two points on the curve* of $f(x)$, hopefully near to the root.
- As Figure illustrates, we draw the line through these two points and find where it intersects the x-axis.
- If $f(x)$ were truly linear, the straight line would intersect the x-axis at the root.

- From the obvious similar triangles we can write

$$\frac{(x_1 - x_2)}{f(x_1)} = \frac{(x_0 - x_1)}{f(x_0) - f(x_1)} \Longrightarrow x_2 = x_1 - f(x_1)\frac{(x_0 - x_1)}{f(x_0) - f(x_1)}$$

- Because f(x) is not exactly linear, x_2 is not equal to r,
- but it should be closer than either of the two points we began with. If we repeat this, we have:

$$\boxed{x_{n+1} = x_n - f(x_n)\frac{(x_{n-1} - x_n)}{f(x_{n-1}) - f(x_n)},\ n = 1, 2, \ldots}$$

- The net effect of this rule is to set $x_0 = x_1$ and $x_1 = x_2$, after each iteration.

The technique we have described is known as, the secant method because the line through two points on the curve is called the secant line.

- Apply Secant method to $f(x) = 3x + sin(x) - e^x$. (**Example py-file:** mysecant.py)

Table 3.1: The Secant method for $f(x) = 3x + sin(x) - e^x$, starting from $x_0 = 1, x_1 = 0$, using a tolerance value of 1E-16.

```
Tolerance values in x and f(x)  are 1.110223e-15 and 1.110223e-15.
Secant iterations for function: 3x+sin(x)-e^x
  k             a                   xm                  b                fm
  1     0.0000000000000000     0.4709895945962973     1.0000000000000000  2.6515881591e-01
  2     1.0000000000000000     0.3075084610161186     0.4709895945962973 -1.3482201684e-01
  3     0.4709895945962973     0.3626132418960405     0.3075084610161186  5.4785286937e-03
  4     0.3075084610161186     0.3604614817438951     0.3626132418960405  9.9517712217e-05
  5     0.3626132418960405     0.3604216717772825     0.3604614817438951 -7.8014178007e-08
  6     0.3604614817438951     0.3604217029607673     0.3604216717772825  1.1080025786e-12
  7     0.3604216717772825     0.3604217029603244     0.3604217029607673  0.0000000000e+00
Mysecant: Root is found as     0.3604217029603244 within tolerance after 7 iterations
SciPy secant: Root is          0.3604217029603243. Accept that value as exact!
```

- Table shows the results from the secant method for the same function that was used to illustrate bisection method.

- If $f(x)$ is far from linear near the root or not continuous, the method may fail. A plot of $f(x)$ is useful to know where/how to start.

3.1.4 Newton's Method

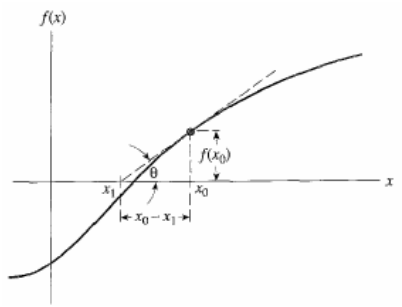


Figure 3.6: Graphical illustration of the Newton's Method.

- One of the most widely used methods of solving equations is Newton's method.

- Like the previous ones, this method is also based on a linear approximation of the function, but does so using a tangent to the curve (see Figure).

- Starting from a single initial estimate, x_0, that is not too far from a root, we move along the tangent to its intersection with the x-axis, and take that as the next approximation.

- This is continued until either the successive x-values are sufficiently close or the value of the function is sufficiently near zero.

- The calculation scheme follows immediately from the right triangle shown in Figure.

$$tan\theta = f^{'}(x_0) = \frac{f(x_0)}{x_0 - x_1} \Rightarrow x_1 = x_0 - \frac{f(x_0)}{f^{'}(x_0)}$$

and the general term is:

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f^{'}(x_n)},\ n = 0, 1, 2, \ldots}$$

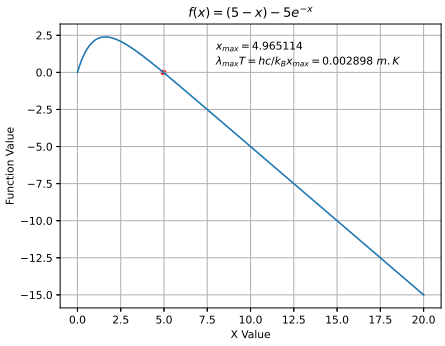
- Newton's algorithm is widely used because, it is more rapidly convergent than any of the methods discussed so far. **Quadratically convergent**
- The error of each step approaches a constant K times the square of the error of the previous step.
- The number of decimal places of accuracy nearly doubles at each iteration.
- There is the need for two functions evaluations at each step, $f(x_n)$ and $f'(x_n)$ and we must obtain the derivative function at the start.
- If a difficult problem requires many iterations to converge, the number of function evaluations with Newton's method may be many more than with linear iteration methods. Because Newton's method always uses two per iteration whereas the others take only one.
- The method may converge to a root different from the expected one or diverge if the starting value is not close enough to the root.
- Apply Newton's method to $f(x) = 3x + sin(x) - e^x$. (**Example py-file:** mynewton.py)
- Table shows the results from Newton's method for the same function that was used to illustrate bisection and secant methods.

(**Example py-file:** blackbodyradiation.py)

Table 3.2: Newton's method for $f(x) = 3x+sin(x)-e^x$, starting from $x_0 = 0$, using a tolerance value of 1E-16.

```
Newton iterations for function: 3x+sin(x)-e^x
   k             x                  xm                   fm                 dfdx
   1     0.0000000000000000     0.3333333333333333 -6.8417728290e-02  2.5493445212e+00
   2     0.3333333333333333     0.3601707135776337 -6.2798507057e-04  2.5022625478e+00
   3     0.3601707135776337     0.3604216804760197 -5.6251553193e-08  2.5018142443e+00
   4     0.3604216804760197     0.3604217029603242 -4.4408920985e-16  2.5018142041e+00
Mynewton: Root is found as     0.3604217029603242 within tolerance after 4 iterations
SciPy Newton: Root is          0.3604217029603244. Accept that value as exact!
```

```
--------------------------------------------------------------------------------
Mybisect: Root is found as     4.9651142317442760 within tolerance after 50 iterations
SciPy bisect: Root is          4.9651142317439962. Accept that value as exact!
--------------------------------------------------------------------------------
Mysecant: Root is found as     4.9651142317442760 within tolerance after 6 iterations
SciPy secant: Root is          4.9651142317442760. Accept that value as exact!
--------------------------------------------------------------------------------
Mynewton: Root is found as     4.9651142317442760 within tolerance after 4 iterations
SciPy Newton: Root is          4.9651142317442760. Accept that value as exact!
--------------------------------------------------------------------------------
```



Chapter 4

Differentiation and Integration

4.1 Numerical Differentiation and Integration with a Computer

4.1.1 Variable force in one dimension

- Consider the motion of a particle of mass m moving along the x-axis under the action of a force F.

Write Newton's second law ($F = ma$) in terms of velocity for a most general force **F(x,v,t)**

$$\begin{aligned} \frac{dv}{dt} &= \frac{F(x,v,t)}{m} \\ \frac{dx}{dt} &= v \end{aligned} \tag{4.1}$$

- In principle, the functions $v = v(t)$ and $x = x(t)$ can be found by solving Equation 4.1 for every $F(x, v, t)$ function (i.e., constant acceleration for F = constant, or harmonic oscillating motion for F= -kx).
- However, for more complex forces $F(x, v, t)$ the analytical solution may not always be available.
- In this case, we will consider the numerical solution.
- We want to find the solutions of the Equation 4.1 at the equally spaced times $t_1, t_2, \ldots, t_N$ and x_i and v_i.
- Write the velocity and position derivatives in the form of *forward-difference* derivative approximation. Take $dt = h$ as step length, then:

$$\frac{v_{i+1} - v_i}{h} = \frac{F(x,v,t)}{m} \longrightarrow \boxed{v_{i+1} = v_i + \frac{F_i}{m}h}$$

$$\frac{x_{i+1} - x_i}{h} = v_i \longrightarrow \boxed{x_{i+1} = x_i + v_i h}$$

- By given initial velocity v_1, initial position x_1 at the time $t_1 = 0$ and the values for $F_i = F(x_i, v_i, t_i)$; the values for v_i, x_i can be calculated for $i = 2.3, \ldots$ in a loop.
- When the function is explicitly known, we can emulate the methods of calculus.
- If we are working with experimental data that are displayed in a table of $[x, f(x)]$ pairs emulation of calculus is **impossible**.

  - We must ***approximate*** the function behind the data in some way.

- **Differentiation with a Computer:**
  - Employs the interpolating polynomials to derive formulas for getting derivatives.
  - These can be applied to functions known explicitly as well as those whose values are found in a table.

- **Numerical Integration-The Trapezoidal Rule:**
  - Approximates, the integrand function with a *linear* interpolating polynomial to derive a very simple but important formula for numerically integrating functions between given limits.

- We cannot often find the true answer numerically because the analytical value is the limit of the sum of an infinite number of terms.
- We must be satisfied with approximations for both derivatives and integrals but, for most applications, the **numerical answer is adequate**.
- The derivative of a function, $f(x)$ at $x = a$, is defined as

$$\frac{df}{dx}|_{x=a} = lim_{\Delta x \to 0} \frac{f(a + \Delta x) - f(a)}{\Delta x}$$

- This is called a ***forward-difference*** approximation.
- The limit could be approached from the opposite direction, giving a ***backward-difference*** approximation.

4.1.2 Differentiation with a Computer

- **Forward-difference** approximation. A computer can calculate an approximation to the derivative, *if a very small value is used for* Δx.

$$\boxed{\left.\frac{df}{dx}\right|_{x=a} = \frac{f(a + \Delta x) - f(a)}{\Delta x}}$$

  - Recalculating with smaller and smaller values of x starting from an initial value.
  - What happens if the value is not small enough?
  - We should expect to find an ***optimal value*** for x.

  - Because round-off errors in the numerator will become great as x approaches zero.

- When we try this for
$$f(x) = e^x sin(x)$$
at $x = 1.9$. **The analytical answer is 4.1653826.**

Apply Forward-difference approximation to $f(x) = e^x sin(x)$. (**Example py-file:** myforwardderivative.py)

Step	Delta x	Numerical Derivative	Error	Error Ratio
0	0.05/ 1	4.0501022943559306	-0.1152803056440694	-
1	0.05/ 2	4.1095606458263845	-0.0558219541736156	2.0651427803033999
2	0.05/ 4	4.1379199156034474	-0.0274626843965526	2.0326474050228986
3	0.05/ 8	4.1517625462243757	-0.0136200537756244	2.0163418477614425
4	0.05/ 16	4.1586002903972030	-0.0067823096027970	2.0081734060042704
5	0.05/ 32	4.1619983544313754	-0.0033842455686246	2.0040831746005470
6	0.05/ 64	4.1636921950123451	-0.0016904049876549	2.0020324084108938
7	0.05/ 128	4.1645378187649840	-0.0008447812350161	2.0009973204752387
8	0.05/ 256	4.1649603866707641	-0.0004222933292359	2.0004607615861758
9	0.05/ 512	4.1651714696399722	-0.0002111303600278	2.0001544504553519
10	0.05/ 1024	4.1652770308974141	-0.0001055691025860	1.9999256871196425
11	0.05/ 2048	4.1653298064920818	-0.0000527935079182	1.9996606921732443
12	0.05/ 4096	4.1653561930434080	-0.0000264069565921	1.9992272768787172
13	0.05/ 8192	4.1653693860280327	-0.0000132139719673	1.9984117309630689
14	0.05/ 16384	4.1653759824112058	-0.0000066175887943	1.9967955667965809
15	0.05/ 32768	4.1653792810393497	-0.0000033189606503	1.9938738332606181
16	0.05/ 65536	4.1653809300623834	-0.0000016699376166	1.9874758297878456
17	0.05/ 131072	4.1653817542828619	-0.0000008457171381	1.9745817382726043
18	0.05/ 262144	4.1653821710497141	-0.0000004289502860	1.9715970959427951
19	0.05/ 524288	4.1653823852539062	-0.0000002147460938	1.9974765471860429
20	0.05/ 1048576	4.1653824970126152	-0.0000001029873848	2.0851689177747628
21	0.05/ 2097152	4.1653824970126152	-0.0000001029873848	1.0000000000000000
22	0.05/ 4194304	4.1653826087713242	0.0000000087713241	-11.7413726289747711
23	0.05/ 8388608	4.1653826832771301	0.0000000832771301	0.1053269259276438
24	0.05/ 16777216	4.1653829813003540	0.0000003813003540	0.2184029708243117
25	0.05/ 33554432	4.1653829813003540	0.0000003813003540	1.0000000000000000

Table 4.1: Forward-difference approximation for $f(x) = e^x sin(x)$.

- Starting with $\Delta x = 0.05$ and halving Δx each time. Table gives the results.
- We find that the errors of the approximation decrease as Δx is reduced until about $\Delta x = 0.05/2097152$.
- Notice that each successive error is about $1/2$ of the previous error as Δx is halved until Δx gets quite small, **at which time round off affects the ratio**.
- At values for Δx smaller than $0.05/2097152$, the error of the approximation increases due to round off.
- In effect, the best value for Δx is **when the effects of round-off and truncation errors are balanced**.

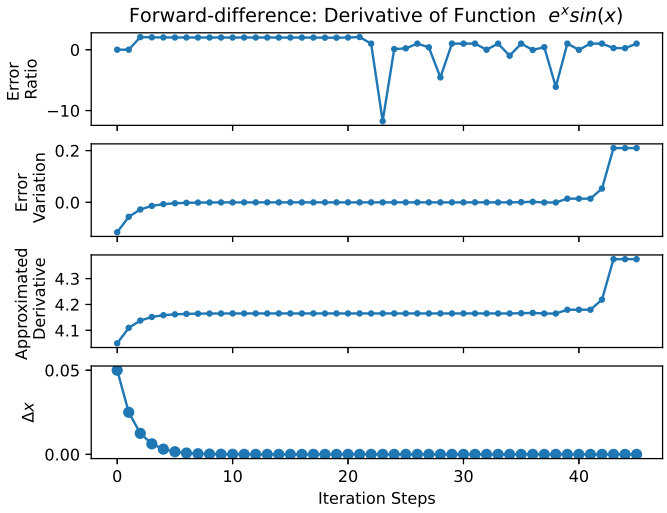


Figure 4.1: Forward-difference approximation for $f(x) = e^x sin(x)$.

- If a backward-difference approximation is used; similar results are obtained.
- **Backward-difference** approximation.

$$\boxed{\left.\frac{df}{dx}\right|_{x=a} = \frac{f(a) - f(a - \Delta x)}{\Delta x}}$$

Apply Backward-difference approximation to $f(x) = e^x sin(x)$. (**Example py-file:** mybackwardderivative.py)

- It is not by chance that the errors are about **halved each time** for both forward- and backward-difference approximations.
- Look at this Taylor series where we have used h for Δx:

$$f(x + h) = f(x) + f'(x) * h + f''(\xi) * h^2/2$$

- Where the last term is the error term. The value of ξ is at some point between x and $x + h$.
- If we solve this equation for $f'(x)$, we get

$$f'(x) = \frac{f(x + h) - f(x)}{h} - f''(\xi) * \frac{h}{2} \tag{4.2}$$

Step	Delta x	Numerical Derivative	Error	Error Ratio
0	0.05/ 1	4.2665138904463618	0.1011312904463617	-
1	0.05/ 2	4.2176675936831032	0.0522849936831031	1.9342316661509737
2	0.05/ 4	4.1919610325982859	0.0265784325982859	1.9671962780256349
3	0.05/ 8	4.1787815600844169	0.0133989600844169	1.9836190593027312
4	0.05/ 16	4.1721096042468275	0.0067270042468275	1.9918168017711522
5	0.05/ 32	4.1687529872206142	0.0033703872206141	1.9959143583513135
6	0.05/ 64	4.1670695083894316	0.0016869083894315	1.9979669564331812
7	0.05/ 128	4.1662264750743816	0.0008438750743816	1.9990025071753801
8	0.05/ 256	4.1658046347765776	0.0004220347765775	1.9995391878008091
9	0.05/ 512	4.1655936336883315	0.0002110336883314	1.9998455218901035
10	0.05/ 1024	4.1654881129034038	0.0001055129034038	2.0000746972514616
11	0.05/ 2048	4.1654353474586969	0.0000527474586969	2.0003409834418382
12	0.05/ 4096	4.1654089634539559	0.0000263634539559	2.0007795179306953
13	0.05/ 8192	4.1653957709786482	0.0000131709786482	2.0016321231806500
14	0.05/ 16384	4.1653891745954752	0.0000065745954752	2.0033139221949763
15	0.05/ 32768	4.1653858771314844	0.0000032771314844	2.0062043608692623
16	0.05/ 65536	4.1653842269442976	0.0000016269442975	2.0142862232140093
17	0.05/ 131072	4.1653834027238190	0.0000008027238190	2.0267796458202478
18	0.05/ 262144	4.1653829859569669	0.0000003859569668	2.0798272552576837
19	0.05/ 524288	4.1653827857226133	0.0000001857226133	2.0781366361207314
20	0.05/ 1048576	4.1653826646506786	0.0000000646506786	2.8727094184239901
21	0.05/ 2097152	4.1653825715184212	-0.0000000284815789	-2.2699120324883144
22	0.05/ 4194304	4.1653825342655182	-0.0000000657344819	0.4332821689348803
23	0.05/ 8388608	4.1653825342655182	-0.0000000657344819	1.0000000000000000
24	0.05/ 16777216	4.1653823852539062	-0.0000002147460938	0.3061032715227527
25	0.05/ 33554432	4.1653823852539062	-0.0000002147460938	1.0000000000000000

Table 4.2: Backward-difference approximation for $f(x) = e^x sin(x)$.

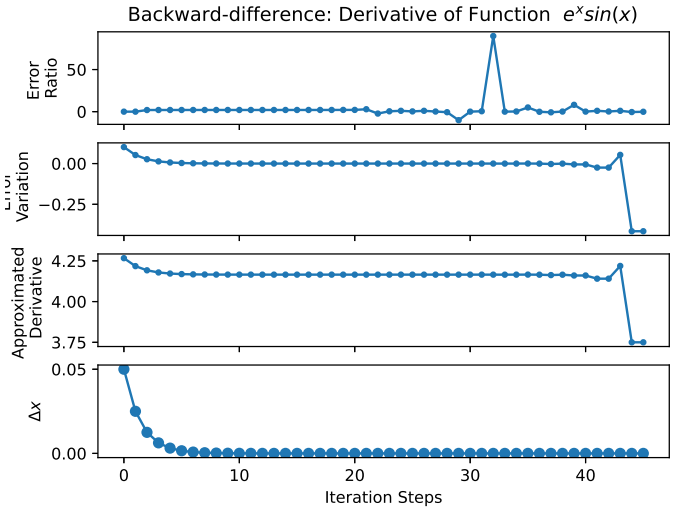


Figure 4.2: Backward-difference approximation for $f(x) = e^x sin(x)$.

- If we repeat this but begin with the Taylor series for $f(x-h)$, it turns out that
$$f'(x) = \frac{f(x) - f(x-h)}{h} + f''(\zeta) * \frac{h}{2} \tag{4.3}$$
- Where ζ is between x and $x-h$.
- The two error terms of Eqs. 4.2 and 4.3 are not identical though both are $O(h)$.
- If we add Eqs. 4.2 and 4.3, then divide by 2, we get the ***central-difference*** approximation to the derivative:
$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - f'''(\xi) * \frac{h^2}{6}$$
- We had to extend the two Taylor series by an additional term to get the error **because the $f''(x)$ terms cancel**.
- This shows that using a central-difference approximation is a much *preferred way* to estimate the derivative.
- Even though we use the same number of computations of the function at each step,
- We approach the answer **much more rapidly**.

$$\boxed{\left.\frac{df}{dx}\right|_{x=a} = \frac{f(x+h) - f(x-h)}{2h}}$$

Apply Central-difference approximation to $f(x) = e^x sin(x)$. (**Example py-file:** mycentralderivative.py)

Step	Delta x	Numerical Derivative	Error	Error Ratio
0	0.05/ 1	4.1583080924011462	-0.0070745075988539	-
1	0.05/ 2	4.1636141197547438	-0.0017684802452562	4.0003317073122826
2	0.05/ 4	4.1649404741008667	-0.0004421258991334	3.9999471840998186
3	0.05/ 8	4.1652720531543963	-0.0001105468456037	3.9994438259975551
4	0.05/ 16	4.1653549473220153	-0.0000276526779848	3.9976904104779085
5	0.05/ 32	4.1653756708259948	-0.0000069291740052	3.9907610869409487
6	0.05/ 64	4.1653808517008883	-0.0000017482991117	3.9633801555226653
7	0.05/ 128	4.1653821469196828	-0.0000004530803173	3.8586957876697707
8	0.05/ 256	4.1653824707236708	-0.0000001292763292	3.5047430575889518
9	0.05/ 512	4.1653825516641518	-0.0000000483358482	2.6745435112230909
10	0.05/ 1024	4.1653825719004089	-0.0000000280995911	1.7201619782859658
11	0.05/ 2048	4.1653825769753894	-0.0000000230246107	1.2204154723952758
12	0.05/ 4096	4.1653825782486820	-0.0000000217513181	1.0585386401779320
13	0.05/ 8192	4.1653825785033405	-0.0000000214966596	1.0118464227311168
14	0.05/ 16384	4.1653825785033405	-0.0000000214966596	1.0000000000000000
15	0.05/ 32768	4.1653825790854171	-0.0000000209145830	1.0278311363130717
16	0.05/ 65536	4.1653825785033405	-0.0000000214966596	0.9729224623288757
17	0.05/ 131072	4.1653825785033405	-0.0000000214966596	1.0000000000000000
18	0.05/ 262144	4.1653825785033405	-0.0000000214966596	1.0000000000000000
19	0.05/ 524288	4.1653825854882598	-0.0000000145117403	1.4813288542519762
20	0.05/ 1048576	4.1653825808316469	-0.0000000191683531	0.7570676603134135
21	0.05/ 2097152	4.1653825342655182	-0.0000000657344819	0.2916027111686003
22	0.05/ 4194304	4.1653825715184212	-0.0000000284815789	2.3079648129953259
23	0.05/ 8388608	4.1653826087713242	0.0000000087713241	-3.2471242096582569
24	0.05/ 16777216	4.1653826832771301	0.0000000832771301	0.1053269259276438
25	0.05/ 33554432	4.1653826832771301	0.0000000832771301	1.0000000000000000

Table 4.3: Central-difference approximation for $f(x) = e^x sin(x)$.

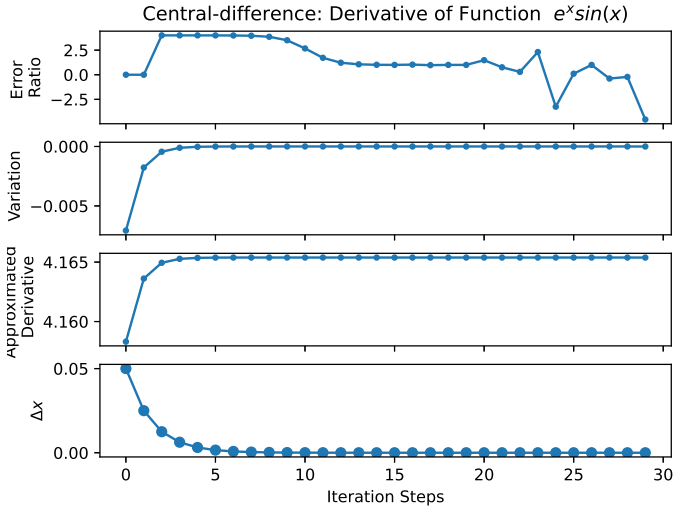


Figure 4.3: Central-difference approximation for $f(x) = e^x sin(x)$.

Apply Forward-difference approximation to Simple Harmonic Motion. (**Example py-file:** shm.py)

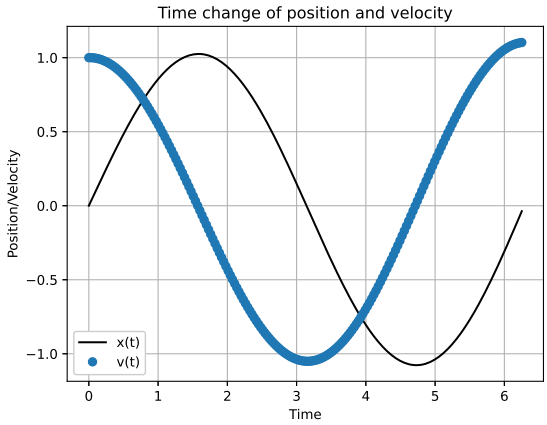


Figure 4.4: Time change of position and velocity in motion under the force F=-kx.

4.1.3 Simple Pendulum

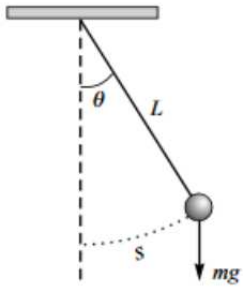


Figure 4.5: Simple pendulum.

- The motion of a simple pendulum, which consists of a point mass m suspended on the end of a rope of length L.
- Newton's second law for motion in the tangential direction:

$$F = ma \longrightarrow -mgSin\theta = m\frac{d^2s}{dt^2}$$

- Here s is the arc length and θ is the angle the rope makes with the vertical (see Figure).

- Since $s = L\theta$,

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}sin\theta$$

- This differential equation has no analytical solution.
- However, for small angle oscillations $sin\theta \approx \theta$ is approximated:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\theta \tag{4.4}$$

- This equation has a solution in terms of sinusoidal functions:

$$\theta(t) = \theta_0 cos\frac{2\pi t}{T}$$

$$\boxed{T = 2\pi\sqrt{\frac{L}{g}}}$$

- Here T is the period of the oscillation and θ_0 is the angular amplitude.
- This formula is small angle ($\theta_0 \leq 15°$) amplitudes gives approximately good results.
- We want to find the exact solution of the pendulum problem **for each amplitude θ_0 numerically**.
- For this purpose, we will transform the problem into a numerical integral.
- Let's multiply both sides of the equation 4.4 by $d\theta/dt$ and rearrange:

$$\begin{aligned} \frac{d\theta}{dt}\frac{d^2\theta}{dt^2} &= -\tfrac{g}{L} sin\theta \tfrac{d\theta}{dt} \\ \frac{1}{2}\frac{d}{dt}\left[\frac{d\theta}{dt}\right]^2 &= -\tfrac{g}{L}\tfrac{d}{dt} cos\theta \end{aligned}$$

- If the indefinite integral is taken side by side,

$$\frac{1}{2}\left(\frac{d\theta}{dt}\right)^2 = \frac{g}{L} cos\theta + C$$

- The initial velocity and angle values are used to determine the integration constant C.
  - If the pendulum is initially released from the maximum angle, its velocity will be zero.

  - So, if $d\theta/dt = 0$ and $\theta = \theta_0$ are substituted into the equation at time $t = 0$, $C = -(g/L)cos\theta_0$ is found.

- If the constant C is put in place and the arrangement is made,

$$\begin{aligned} \frac{d\theta}{dt} &= \sqrt{\tfrac{2g}{L}(cos\theta - cos\theta_0)} \\ dt &= \sqrt{\tfrac{L}{2g}}\tfrac{d\theta}{\sqrt{cos\theta - cos\theta_0}} \end{aligned}$$

- Integrating the right side of this expression from $\theta = 0$ to the angle $\theta = \theta_0$, the left side will be one quarter of a period.

$$\frac{T}{4} = \sqrt{\frac{L}{2g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{cos\theta - cos\theta_0}}$$

  Thus, we have written the period formula as an integral.

- For each given amplitude θ_0 we can calculate this integral numerically.

- However, since the value of the integrand diverges at the upper bound (for $\theta = \theta_0$), it is necessary to do a **variable replacement** first.

- For this purpose, first, the identities $cos\theta = 1 - 2sin^2(\theta/2)$ and $cos\theta_0 = 1 - 2sin^2(\theta_0/2)$ are inserted in the denominator,

- and then with the variable change $k = sin(\theta_0/2) = \frac{sin(\theta/2)}{sin\phi}$, the integral becomes:

$$\boxed{T = 4\sqrt{\frac{L}{g}} \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - sin^2(\theta_0/2)sin^2\phi}}}$$

- This integral whose denominator is never zero is known as an *elliptical integral of the first kind.*

- **There is no analytical solution.**

4.1.4 Numerical Integration - The Trapezoidal Rule

- Given the function, $f(x)$, the **antiderivative** is a function $F(x)$ such that $F'(x) = f(x)$.

- The definite integral

$$\int_a^b f(x)dx = F(b) - F(a)$$

  can be evaluated from the antiderivative.

- Still, there are functions that do not have an antiderivative expressible in terms of ordinary functions. Such as the function: $f(x) = e^x/log(x)$

```
from sympy import *
x = symbols('x')
f = exp(x)/log(x)
df = integrate(f, x) # Function derivative in symbolic form
print(df)
# Integral(exp(x)/log(x), x)
```

- Is there any way that the definite integral can be found when the antiderivative is unknown?

- We can do it numerically by using the **composite trapezoidal rule**

- The definite integral is the area between the curve of $f(x)$ and the x-axis.

- That is the principle behind all numerical integration;

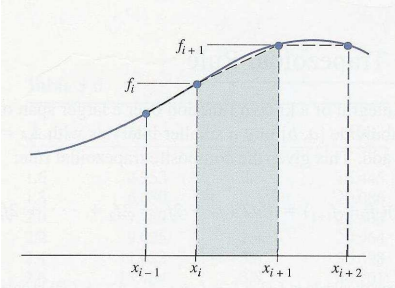


Figure 4.6: The trapezoidal rule.

  - We divide the distance from $x = a$ to $x = b$ into **vertical strips** and add the areas of these strips.
  - The strips are often made equal in widths but that is not always required.
  - Approximate the curve with a sequence of straight lines.
  - In effect, we slope the top of the strips to match with the curve as best we can.

- The gives us the ***trapezoidal rule***. Figure illustrates this.

- It is clear that the area of the strip from x_i to x_{i+1} gives an approximation to the area under the curve:

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{f_i + f_{i+1}}{2}(x_{i+1} - x_i)$$

- We will usually write $h = (x_{i+1} - x_i)$ for the width of the interval.
- Error term for the trapezoidal integration is

$$Error = -(1/12)h^3 f''(\xi) = O(h^3)$$

- What happens, if we are getting the integral of a known function over a larger span of x-values, say, from $x = a$ to $x = b$?
- We subdivide [a,b] into n smaller intervals with $\Delta x = h$, apply the rule to each subinterval, and add.

The Composite Trapezoidal Rule

- This gives the **composite trapezoidal rule**;

$$\boxed{\int_a^b \approx \sum_{i=0}^{n-1} \frac{h}{2}(f_i + f_{i+1}) = \frac{h}{2}(f_0 + 2f_1 + 2f_2 + \ldots + 2f_{n-1} + f_n)}$$

- The error is not the local error $O(h^3)$ but the global error, the sum of n local errors;

$$Global\ error = \frac{-(b-a)}{12}h^2 f''(\xi) = O(h^2)$$

where $nh = (b - a)$

- Use the trapezoidal rule to estimate the integral from $x = 1.8$ to $x = 3.4$ for $f(x) = e^x$.
- Applying the trapezoidal rule:

$$\begin{aligned}\int_{1.8}^{3.4} f(x)dx \quad &\approx \tfrac{0.2}{2}[f(1.8) + 2f(2.0) + 2f(2.2) + 2f(2.4) \\ &+2f(2.6) + 2f(2.8) + 2f(3.0) + 2f(3.2) \\ &+f(3.4)] = 23.9944\end{aligned}$$

Apply The trapezoidal rule to $f(x) = e^x$ in interval of 1.8 and 3.4. (**Example py-file:** mytrapezoid.py)

```
Mytrapezoid: Integration is       23.9941143322614288 for the function $e^x$ in interval of 1.800000 and 3.400000.
SciPy trapezoid: Integration is   23.9146518697568169 for the function $e^x$ in interval of 1.800000 and 3.400000.
```

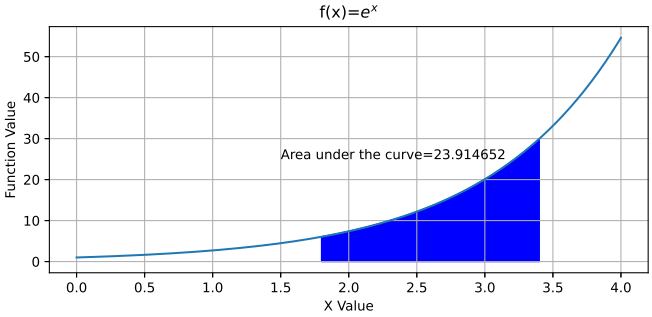


Figure 4.7: Integration for $f(x) = e^x$ by the trapezoidal rule.

Apply The trapezoidal rule to Simple Pendulum to find the period. (**Example py-file:** simplependulum.py)

Degree	Calculated Period MyTrapezeoid	Calculated Period SciPyTrapezeoid	Small Angle Approximation
0.00	2.0070946166038111	2.0070946166038111	2.0070899231544930
5.00	2.0080503420735032	2.0080503420734983	2.0070899231544930
10.00	2.0109225326358318	2.0109225326358349	2.0070899231544930
15.00	2.0157263062261905	2.0157263062261901	2.0070899231544930
20.00	2.0224871133944586	2.0224871133944538	2.0070899231544930
25.00	2.0312411268457220	2.0312411268457256	2.0070899231544930
30.00	2.0420358043998514	2.0420358043998550	2.0070899231544930
35.00	2.0549306441689779	2.0549306441689810	2.0070899231544930
40.00	2.0699981579512587	2.0699981579512610	2.0070899231544930
45.00	2.0873250976325295	2.0873250976325277	2.0070899231544930
50.00	2.1070139804141048	2.1070139804141186	2.0070899231544930
55.00	2.1291849728107688	2.1291849728107644	2.0070899231544930
60.00	2.1539782117886461	2.1539782117886550	2.0070899231544930
65.00	2.1815566658380487	2.1815566658380630	2.0070899231544930
70.00	2.2121096716366528	2.2121096716366564	2.0070899231544930
75.00	2.2458573268222062	2.2458573268222080	2.0070899231544930
80.00	2.2830559815494613	2.2830559815494604	2.0070899231544930
85.00	2.3240051589550292	2.3240051589550310	2.0070899231544930
90.00	2.3690563597150089	2.3690563597150192	2.0070899231544930

Table 4.4: Integration for $\frac{1.0}{\sqrt{1.0-(sin(\theta_0/2)sin(\phi))^2}}$ by the trapezoidal rule.

Chapter 5

Differential Equations

5.1 Initial Value Problems

- Most **problems in the real world** are modeled with **differential equations** because it is easier to see the relationship in terms of a derivative.
- e.g. Newton's Law: F=Ma, $d^2s/dt^2 = a = F/M$ (constant acceleration). **2^{nd} order ordinary differential equation.**
  - It is **ordinary** since it does not involve partial differentials.
  - **Second order** since the order of the derivative is two.
  - The solution to this equation is a function, $s(t) = (1/2)at^2 + v_0t + s_0$.
  - Two arbitrary constants, v_0 and s_0, the initial values for the velocity and position.
  - The equation for s(t) allows the computation of a numerical value for s, the position of the object, at any value for time, the independent variable, t.
- e.g. Harmonic oscillator problem in mechanics,
- e.g. One-dimensional Schrödinger equation in quantum mechanics,
- e.g. One-dimensional Laplace equation in electromagnetic theory, etc.
- Analytical solutions of these equations are often non-existent or very complicated.
- Numerical solutions are the remedy. In terms of solution technique, we can divide differential equations into three groups:

1. **Initial Value Problems:** In time-dependent problems, the initial state at time t=0 is given and a solution is searched for later t values. For example, in the following quadratic equation

$$\frac{d^2y}{dt^2} = f(y, y', t)$$

   two initial conditions must be given at t=0, namely $y(0)$ and $y'(0)$ values. (N^{th} order differential equation $\rightarrow$ N initial conditions).

2. **Boundary Value Problems.**

3. **Eigenvalue (characteristic-value) Problems.**

5.1.1 Projectile Motion with Air Resistance

- In addition to a vertical gravitational force on a 2D projectile motion, there is also a friction force to a certain extent due to air resistance.
- This frictional force is usually in the opposite direction to velocity and is proportional to the square of the velocity: $\vec{F}_r = -kv\vec{v}$ (Drag force, $F_D = -(1/2)c\rho A v^2 \vec{v}/|\vec{v}|$ here, c is the drag coefficient, ρ the air density, and A the projectile's cross-sectional area).

If we write Newton's 2^{nd} law as a vector in two dimensions,

$$\begin{aligned} m\vec{a} &= \vec{F}_{net} \\ m\frac{d^2\vec{r}}{dt^2} &= m\vec{g} - kv\vec{v} \end{aligned}$$

- and component wise (where $k/m = \gamma$):

$$\frac{d^2x}{dt^2} = -\gamma\left(\sqrt{v_x^2 + v_y^2}\right)v_x \qquad \& \qquad \frac{dx}{dt} = v_x$$
$$\frac{d^2y}{dt^2} = -g - \gamma\left(\sqrt{v_x^2 + v_y^2}\right)v_y \qquad \& \qquad \frac{dy}{dt} = v_y$$

- **Now, we have a set of equations.**

5.1.2 Planetary Motion

- In the previous projectile motion example, we used the gravitational force with the expression $F = mg$ and gravitational acceleration as being constant near the Earth's surface.
- However, the gravitational force between masses is most generally given by Newton's law of universal gravitation:

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

  Here, $G = 6.6743\times10^{-11}\ m^3kg^{-1}s^{-2}$ is called the universal gravitational constant. The force is attractive and along the direction connecting the two masses.
- This expression should be used when studying the motion of planets and moons.

- Let's study the motion of a planet (mass m)moving under the gravitational force of the Sun (mass M). If we take the sun at the origin, the vector expression of the force acting on the planet would be:

$$\vec{F} = -G\frac{Mm}{r^3}\vec{r}$$

- Since the orbit of the planet will be at a plane (2D), the position vector $\vec{r}$ and accordingly the acceleration vector $\vec{a}$ would have two components as:

$$\begin{aligned} \vec{r} &= x\hat{i} + y\hat{j} \\ \vec{a} &= \frac{d^2x}{dt^2}\hat{i} + \frac{d^2y}{dt^2}\hat{j} \end{aligned}$$

- Newton's 2^{nd} law as $\vec{a} = \vec{F}/m$ and also velocity expressions for the x- and y-components:

$$\begin{aligned} \frac{d^2x}{dt^2} &= -G\frac{M}{r^3}x \quad \& \quad \frac{dx}{dt} = v_x \\ \frac{d^2y}{dt^2} &= -G\frac{M}{r^3}y \quad \& \quad \frac{dy}{dt} = v_y \end{aligned}$$

- **Now, we have a set of equations.**

5.1.3 Euler Method

- In an **initial-value problem**, the numerical solution **begins at the initial point and marches from there** to increasing values for the independent variable.

- **The Euler method.** Describes a method that is easy to use but is not very precise unless the step size, the intervals for the projection of the solution, is very small.

- Consider the following first-order differential equation:

$$\frac{dy}{dx} = y'(x) = f(x,y) \quad \& \quad y(x_0) = y_0 \tag{5.1}$$

- Here x is the variable, y(x) and f(x,y) are real functions, and the initial condition y_0 is a real number.

- From the solution of this equation, we get $y_1, y_2, \ldots, y_n$ values for the function at the points $x_1, x_2, \ldots, x_n$ with equal step lengths h.
- **Equations of higher order are solved by converting them to a system of linear equations.**
- The expression given by Equation 5.1 is written as the forward-difference approximation at a point x_i by Euler's method.

$$\frac{y_{i+1} - y_i}{h} + O(h) = f(x_i, y_i)$$

- If we solve this expression for y_{i+1}, we get the Euler method formula:

$$\boxed{y_{i+1} = y_i + hf(x_i, y_i) + O(h^2)}$$

- This expression shows that the error in one step of Euler method is $O(h^2)$. But, this error is just the local error. Over many steps, the global error becomes $O(h)$ (as $NO(h^2) \approx O(h)$ for N steps).
- *The method is easy to program when we know the formula for* $y'(x)(\equiv f(x_i, y_i))$ *and a starting value,* $y_0 = y(x_0)$.
- Let's see the application of this method on an example. Given differential equation,

$$\frac{dy}{dx} = x + y$$

- The analytical solution of this equation is given as $y(x) = 2e^x - x - 1$. Initial condition: $y(x = 0) = 1$

(**Example py-file:** myeuler.py) As can be seen from the table, the margin of error is large in the Euler method.

5.1.4 Runge-Kutta Method

- Simple Euler method comes from using just one term from the Taylor series for y(x) expanded about $x = x_0$.
- What if we use more terms of the Taylor series? Runge and Kutta, developed algorithms from using more than two terms of the series.
- In the Euler method, the increment is directly from x_i to x_{i+1}.
- Second-order Runge-Kutta methods are obtained by using a weighted average of two increments to $y(x_0), k_1$ and k_2.

Step x	Euler y	Exact y	Euler-Exact Error	SciPy y
0.00	1.0000000000000000	1.0000000000000000	0.0000000000000000	1.0000000000000000
0.10	1.1000000000000001	1.1103418361512953	0.0103418361512952	1.1103418365038888
0.20	1.2200000000000002	1.2428055163203395	0.0228055163203393	1.2428055171581294
0.30	1.3620000000000001	1.3997176151520065	0.0377176151520064	1.3997176170100418
0.40	1.5282000000000000	1.5836493952825408	0.0554493952825408	1.5836493990278593
0.50	1.7210200000000000	1.7974425414002564	0.0764225414002564	1.7974425476900568
0.60	1.9431220000000000	2.0442376007810177	0.1011156007810177	2.0442376098866673
0.70	2.1974342000000000	2.3275054149409531	0.1300712149409531	2.3275054266863835
0.80	2.4871776200000002	2.6510818569849350	0.1639042369849348	2.6510818708124289
0.90	2.8158953820000003	3.0192062223138993	0.2033108403138990	3.0192062374603896
1.00	3.1874849202000002	3.4365636569180902	0.2490787367180900	3.4365636726612259

Table 5.1: Solution of the differential equation $dy/dx = x + y$ in the interval [0, 1] by Euler method.

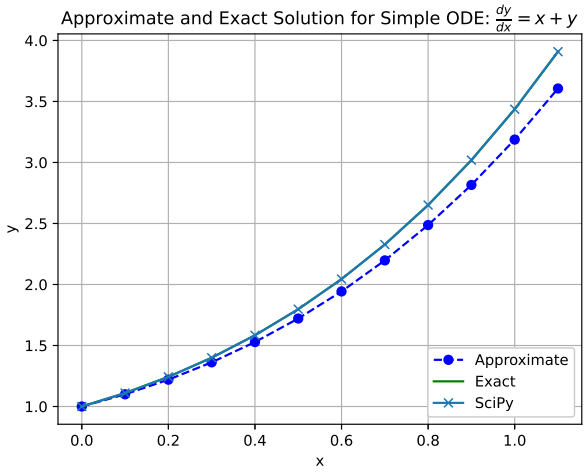


Figure 5.1: Solution of the differential equation $dy/dx = x + y$ in the interval [0, 1] by Euler method.

- Let's take a "trial step" in the middle and then increment to x_{i+1} by using these middle x- and y-values. Two quantities are defined here as k_1 and k_2,

$$\begin{aligned} k_1 &= hf(x_i, y_i) \\ k_2 &= hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1) \end{aligned}$$

- The parameter;
  - k_1 is for the calculation at x_i, y_i,

  - k_2 is for a half-step away $(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1)$ calculation.

- Accordingly, the 2^{nd} order Runge-Kutta formula becomes:

$$\boxed{y_{i+1} = y_i + k_2 + O(h^3)}$$

- In the Runge-Kutta method, the margin of error in one step is $O(h^3)$ and is $O(h^2)$ in the entire interval.

- It works better than the Euler method, but it comes at a **cost**: f(x, y) will be **calculated twice** at each step.

- This "trial step" technique can be taken even further. Fourth-order Runge-Kutta (RK4) methods are most widely used and are derived in similar fashion.

$$\begin{aligned}
k_1 &= f(x_i, y_i) \\
k_2 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1) \\
k_3 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_2) \\
k_4 &= f(x_i + h, y_i + hk_3) \\
y_{i+1} &= y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5)
\end{aligned}$$

- The local error term for the fourth-order Runge-Kutta method is $O(h^5)$; the global error would be $O(h^4)$.

- It is computationally more efficient than the (modified) Euler method because the steps can be manyfold larger for the same accuracy.

- **However, four evaluations of the function are required per step rather than two.**

- Let's apply the RK4 method on the previous example. Given differential equation,

$$\frac{dy}{dx} = x + y$$

- The analytical solution of this equation is given as $y(x) = 2e^x - x - 1$. Initial condition: $y(x = 0) = 1$

(**Example py-file:** myrungekutta.py) As can be seen from the Table, much more sensitive results are obtained compared to the Euler method.

Step x	RK4 y	Exact y	RK4-Exact Error	SciPy y
0.00	1.0000000000000000	1.0000000000000000	0.0000000000000000	1.0000000000000000
0.10	1.1103416666666668	1.1103418361512953	0.0000001694846286	1.1103418365038888
0.20	1.2428051417013890	1.2428055163203395	0.0000003746189505	1.2428055171581294
0.30	1.3997169941250756	1.3997176151520065	0.0000006210269310	1.3997176170100418
0.40	1.5836484801613715	1.5836493952825408	0.0000009151211693	1.5836493990278593
0.50	1.7974412771936765	1.7974425414002564	0.0000012642065799	1.7974425476900568
0.60	2.0442359241838663	2.0442376007810177	0.0000016765971513	2.0442376098866673
0.70	2.3275032531935538	2.3275054149409531	0.0000021617473993	2.3275054266863835
0.80	2.6510791265846310	2.6510818569849350	0.0000027304003041	2.6510818708124289
0.90	3.0192028275601421	3.0192062223138993	0.0000033947537572	3.0192062374603896
1.00	3.4365594882703321	3.4365636569180902	0.0000041686477581	3.4365636726612259

Table 5.2: Solution of the differential equation $dy/dx = x + y$ in the interval [0, 1] by 4th order Runge-Kutta method.

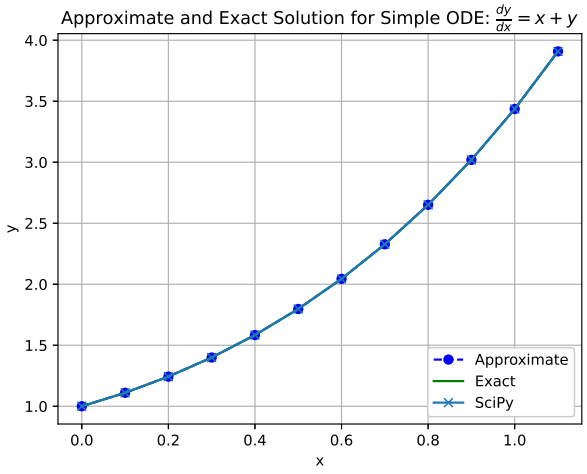


Figure 5.2: Solution of the differential equation $dy/dx = x+y$ in the interval [0, 1] by Euler method.

5.1.5 Second Degree Equations

- **Any second-order or higher-order differential equation can be converted into a system of first-order (linear) equations.** For example,

$$\frac{d^2y}{dx^2} + A(x)\frac{dy}{dx} + B(x)y(x) = 0$$

- Let's define two new functions for the equation, $y_1(x)$ and $y_2(x)$:

$$y_1(x) = y(x) \quad \& \quad y_2(x) = \frac{dy}{dx}$$

- With this transformation, instead of one 2^{nd} order equation, two 1^{st} order equations are formed:

$$\begin{aligned} (1)\ \frac{dy_1}{dx} &= y_2(x) \\ (2)\ \frac{dy_2}{dx} &= -A(x)y_2 - B(x)y_1 \end{aligned}$$

- All we need to do to solve higher-order equations, even a **system** of higher-order initial-value problems, is to reduce them to a system of first-order equations.

- Such as: **One M-order equation → a system with M first-order equations.**

- Let's take the most general system of differential equations with M unknowns:

$$\begin{aligned} \frac{dy_1}{dx} &= f_1(x, y_1, \ldots, y_M) \quad \& \quad y_1(0) = y_{10} \\ &\vdots \qquad\qquad\qquad\qquad \vdots \\ \frac{dy_M}{dx} &= f_M(x, y_1, \ldots, y_M) \quad \& \quad y_M(0) = y_{M0} \end{aligned} \tag{5.2}$$

- The next step for solving is to apply the methods (such as; Euler, Runge-Kutta) for the 1^{st}t order differential equation to these linear system.

We had a set of equations. Two second degree and two first degree differential equations with two unknowns.

$$(3)\quad \frac{d^2x}{dt^2} = -\gamma\left(\sqrt{v_x^2 + v_y^2}\right)v_x \quad \& \quad (1)\ \frac{dx}{dt} = v_x$$
$$(4)\ \frac{d^2y}{dt^2} = -g - \gamma\left(\sqrt{v_x^2 + v_y^2}\right)v_y \quad \& \quad (2)\ \frac{dy}{dt} = v_y$$

- To solve these two 2^{nd} degree equations (plus two 1^{st} degree equations) given above, we first convert them to a system of 4 1^{st} degree (linear) equations.
- To this end, let's define the four unknowns as follows:
  - $x \rightarrow y_1$
  - $y \rightarrow y_2$
  - $v_x \rightarrow y_3$
  - $v_y \rightarrow y_4$
- Accordingly, the above 2^{nd} degree system is written as:

$$\begin{aligned}
(1)\ \frac{dy_1}{dt} &= y_3 \\
(2)\ \frac{dy_2}{dt} &= y_4 \\
(3)\ \frac{dy_3}{dt} &= -\gamma \left(\sqrt{y_3^2 + y_4^2} \right) y_3 \\
(4)\ \frac{dy_4}{dt} &= -g - \gamma \left(\sqrt{y_3^2 + y_4^2} \right) y_4
\end{aligned}$$

- When $\gamma = 0$ in this system of equations, we obtain our usual parabolic curve $y = (v_{0y}/v_{0x})x - (g/2v_{0x}^2)x^2$.

To calculate the effect of air friction, let's take the initial conditions ($t = 0$) and constants (g & γ):

$$\begin{aligned}
x_0 = y_1(t=0) = 0 \quad &\& \quad y_0 = y_2(t=0) = 0 \\
v_{0x} = y_3(t=0) = 6.0 \quad &\& \quad v_{0y} = y_4(t=0) = 8.0 \\
g = 10.0 \quad &\& \quad \gamma = 0.01
\end{aligned}$$

We had a set of equations. Two second degree and two first degree differential equations with two unknowns.

$$\begin{aligned}
(3)\ \frac{d^2x}{dt^2} = -G\frac{M}{r^3}x \quad &\& \quad (1)\ \frac{dx}{dt} = v_x \\
(4)\ \frac{d^2y}{dt^2} = -G\frac{M}{r^3}y \quad &\& \quad (2)\ \frac{dy}{dt} = v_y
\end{aligned}$$

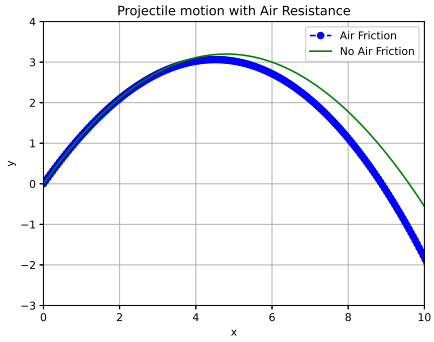


Figure 5.3: Numerical solution of projectile motion with and without air friction. (**Example py-file:** airfriction.py)

- To solve these two 2^{nd} degree equations (plus two 1^{st} degree equations) given above, we first convert them to a system of 4 1^{st} degree (linear) equations.
- To this end, let's define the four unknowns as follows:
  - $x \rightarrow y_1$
  - $y \rightarrow y_2$
  - $v_x \rightarrow y_3$
  - $v_y \rightarrow y_4$
- Accordingly, the above 2^{nd} degree system is written as:

$$\begin{aligned}
(1)\ \frac{dy_1}{dt} &= y_3 \\
(2)\ \frac{dy_2}{dt} &= y_4 \\
(3)\ \frac{dy_3}{dt} &= -\frac{GM}{[y_1^2+y_2^2]^{3/2}} y_1 \\
(4)\ \frac{dy_4}{dt} &= -\frac{GM}{[y_1^2+y_2^2]^{3/2}} y_2
\end{aligned} \tag{5.3}$$

- For the motion of the planets, we use the astronomical unit system. The Earth-Sun average distance would be in units of astronomical length: $1\ au \approx 1.5 \times 10^{11}\ m$. The time taken for the Earth to go around the Sun once is 1 year (y) as the unit of time.
- Calculated in these units, the product of GM,

$$GM \approx 40(au)^3/y^2$$

- To calculate the planetary motion, let's take the initial conditions at time t=0 in terms of four unknowns:

$$x_0 = y_1(t=0) = 1.0\ au \quad \& \quad y_0 = y_2(t=0) = 0$$
$$v_{0x} = y_3(t=0) = 0.0 \quad \& \quad v_{0y} = y_4(t=0) = 6.0\ au/y$$

- Then, also take $v_{0y} = y_4(t=0) = 8.0\ au/y$.

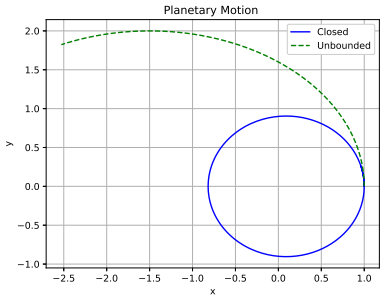


Figure 5.4: Numerical solution of planetary motion. There can be closed orbits (ellipse), or solutions going to infinity (unbounded, hyperbola) for different velocities. (**Example py-file:** planetarymotion.py)

5.2 Boundary Value Problems

1. **Initial Value Problems.**

2. **Boundary Value Problems** The solution of the differential equation is searched within *certain constraints*, called the **boundary conditions**. For example,

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

   - If the solution of the equation in the interval of $x : [0, L]$ is required,
   - the values at the y(0) and y(L) boundaries should be given.

3. **Eigenvalue (characteristic-value) Problems**

- Consider a 2nd-order differential equation in the interval [a,b]:

$$y'' = f(x, y, y')$$

Two necessary conditions for the solution of this equation are given at two extremes:

$$\begin{aligned} y(a) &= A \\ y(b) &= B \end{aligned}$$

- This problem is more difficult to solve than the initial value problem that we previously discussed.
  - In initial value problem, $y(0)$ and $y'(0)$ are both given at $x = 0$.
  - It was possible to start with these two initial values and progress the solution through the interval.
  - In the boundary value problem, the number of initial conditions is insufficient.
- **We cannot directly obtain the solution with methods such as Euler or Runge-Kutta.**
- Eigenvalue problems are even more difficult.

- See the following 2nd degree differential equation:

$$y'' = f(x, y, y' : \lambda)$$

- Again, let the boundary conditions of this equation to be given at both ends.
- If these conditions can only be satisfied for *certain* λ *values*, we call it the *eigenvalue problem*.
  - e.g.: Vibrations of a wire with both ends fixed give stable solution only for *certain wavelengths*.
  - e.g.: Solutions of the Schrödinger equation that are zero at infinity exist only for *certain energy eigenvalues*.
- **In terms of numerical solution, boundary value and eigenvalue problems are solved by the same method.**

5.2.1 Trial-and-Error (Linear Shooting) Method

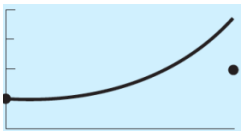


Figure 5.5: First guess.

Figure 5.6: Second guess.

...

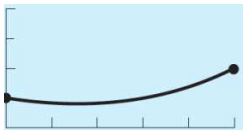


Figure 5.7: Expected result.

- The basic approach in solving **boundary value** and **eigenvalue problems** is known as the **trial-and-error method**.
  - At one boundary, the solution is started by giving an estimated value to the missing initial condition,
  - A trial solution is then found with either Euler or Runge-Kutta method at the other boundary (**First guess**),
  - How much deviation from the given boundary condition is determined,
  - Taking this deviation into account, a new solution is restarted with a new estimation (**Second guess**),

  - This process is repeated (...) until the other boundary condition is satisfied (**Expected result**).

- Let's see this method on an example:

$$y''(x) - (4x^2 - 2)y = 0 \qquad y(0) = 1.0$$
$$with\ BCs \qquad y(1) = 1/e = 0.36787944$$

- This differential equation has the solution of $y = e^{(-x^2)}$ (Gaussian function).

First, let's transform this boundary value problem into a first-order system of equations:

- $y_1 = y$
- $y_2 = y'$
- $\frac{dy_1}{dx} = y_2$
- $\frac{dy_2}{dx} = (4x^2 - 2)y_1$

- Notice that the boundary conditions are given only for y_1: $y_1(0) = 1.0$ and $y_1(1) = 0.368$.
- Notice that there is no initial condition for y_2.
- **Now, we have a set of equations.**
- Here, let's take an estimated initial value of a:

$$\textbf{first guess}: \ y_2(0) = a$$

- Now, find the solutions $y_1(x)$ and $y_2(x)$ (by let's say RK4) with these $y_1(0)$ and $y_2(0)$ values.
- Denote the value obtained for y_1 in the other boundary with $y_{1a}(1)$
- and find the difference (Δ_a) with the real one $y_1(1)$ (here, 0.368):

$$y_2(0) = a \rightarrow solution: \ y_{1a}(x) \rightarrow \Delta_a = y_{1a}(1) - y_1(1)$$

- Now, let's make a **second (another) guess** of b and calculate the error again at the other boundary:

$$y_2(0) = b \rightarrow solution: \ y_{1b}(x) \rightarrow \Delta_b = y_{1b}(1) - y_1(1)$$

- Remind the secant methods for root finding.

- After these two estimated shoots, the most accurate starting value to choose will be the extension of the line passing through two points:

$$y_2(0) = b - \frac{\Delta_b}{\Delta_b - \Delta_a}(b - a)$$

- Then, the calculation (RK4) is repeated with this selected value by secant method.

- Finally, the solution is found when the margin of error in the other boundary is smaller than a certain tolerance.

- (**Example py-file:** mylinearshooting.py)

Solution (y_2(0)) is found.

Step x	RK4 y	Exact y	RK4-Exact Error	SciPy y
0.00	1.0000000000000000	1.0000000000000000	0.0000000000000000	1.0000000000000000
0.01	0.9999000049788840	0.9999000049998333	0.0000000000209494	0.9999000049769683
0.02	0.9996000799477730	0.9996000799893344	0.0000000000415614	0.9996000797242913
0.03	0.9991004048166946	0.9991004048785274	0.0000000000618328	0.9991004045538127
0.04	0.9984012792358460	0.9984012793176064	0.0000000000817604	0.9984012793048143
0.05	0.9975031222961184	0.9975031223974601	0.0000000001013417	0.9975031227294181
0.06	0.9964064721104181	0.9964064722309933	0.0000000001205752	0.9964064724925857
0.07	0.9951119852763703	0.9951119854158298	0.0000000001394594	0.9951119851721190
0.08	0.9936204362211554	0.9936204363791490	0.0000000001579936	0.9936204356801045
0.09	0.9919327164293936	0.9919327166055711	0.0000000001761775	0.9919327149819753
0.10	0.9900498335551565	0.9900498337491681	0.0000000001940116	0.9900498322585485
0.90	0.4448580651753301	0.4448580662229407	0.0000000010476106	0.4448580640585380
0.91	0.4368785664741286	0.4368785675332217	0.0000000010590931	0.4368785643059327
0.92	0.4289563976083164	0.4289563986790725	0.0000000010707561	0.4289563954023433
0.93	0.4210936588053051	0.4210936598879114	0.0000000010826063	0.4210936573120806
0.94	0.4132923766756930	0.4132923777703437	0.0000000010946508	0.4132923758470025
0.95	0.4055545039564232	0.4055545050633201	0.0000000011068969	0.4055545032383834
0.96	0.3978819193318516	0.3978819204512042	0.0000000011193526	0.3978819180580180
0.97	0.3902764273314946	0.3902764284635206	0.0000000011320260	0.3902764252182211
0.98	0.3827397583031426	0.3827397594480686	0.0000000011449260	0.3827397559145742
0.99	0.3752735684599452	0.3752735696180068	0.0000000011580616	0.3752735662107565
1.00	0.3678794399999992	0.3678794411714418	0.0000000011714427	0.3678794377581407
1.01	0.3605588812968958	0.3605588824819751	0.0000000011850793	0.3605588789291024

Figure 5.8: Solution for the Boundary Value Problem for the ODE: $y''(x) - (4x^2 - 2)y = 0$.

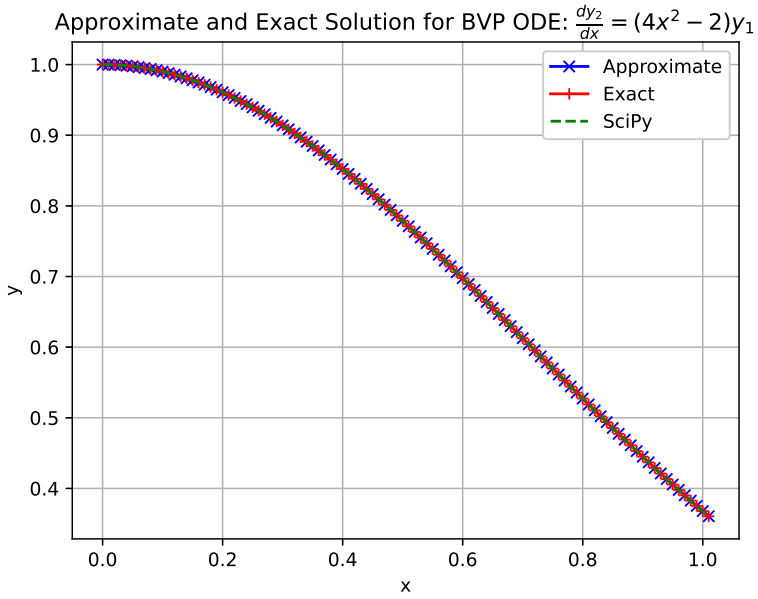


Figure 5.9: Solution for the Boundary Value Problem for the ODE: $y''(x) - (4x^2 - 2)y = 0$.

5.2.2 Laplace Equation in Electrostatics

- The electrostatic potential created by a static charge distribution at a charge-free region is given by the following Laplace equation:

$$\nabla^2 V = \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + + \frac{\partial^2 V}{\partial z^2} = 0$$

  Here, $V(x, y, z)$ is the potential within the region.

- The solution of this problem for particular charge distributions concerns the subject of **partial differential equations**.

- However, the dimensions of the problem can be reduced if the charge distribution exhibit a spatial symmetry.

- For example, in a system with **spherical symmetry**, the solution of the problem becomes easier if the partial derivatives in Laplace's equation are expressed in terms of **spherical coordinates** (r, θ, ϕ):

$$\nabla^2 V = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2 \frac{\partial V}{\partial r}\right) + \frac{1}{r^2 \sin\theta}\frac{\partial}{\partial \theta}\left(\sin\theta \frac{\partial V}{\partial \theta}\right) + \frac{1}{r^2 \sin^2\theta}\frac{\partial^2 V}{\partial \phi^2} = 0 \quad (5.4)$$

- Let the two concentric conductive spherical shells of radii R_a and R_b be held at constant potentials V_a and V_b (Figure 5.10).

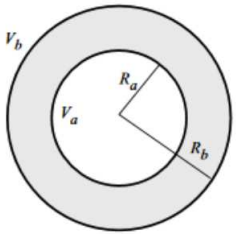


Figure 5.10: The region between two spherical shells of different potential.

- Because of spherical symmetry, the potential distribution in the region between the two spheres ($R_a < r < R_b$) will be a function of distance r only.
- Accordingly, the derivatives with respect to the variables (θ, ϕ) in Equation 5.4 become zero, and the partial derivative in the remaining term becomes the full derivative:

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial V}{\partial r}\right) \rightarrow \frac{d^2V}{dr^2} + \frac{2}{r}\frac{dV}{dr} = 0$$

The boundary conditions of this differential equation become:

$$\begin{aligned} V(R_a) &= V_a \\ V(R_b) &= V_b \end{aligned}$$

First, we transform this equation into a linear system of equations:

$$\begin{aligned} V &\rightarrow V_1 \\ \frac{dV}{dr} &\rightarrow V_2 \end{aligned}$$

with these values ($V_1 and V_2$), the system of equations to be solved

$$\begin{aligned} \frac{dV_1}{dr} &= V_2 \\ \frac{dV_2}{dr} &= -\frac{2}{r}V_2 \end{aligned}$$

and the boundary conditions are:

$$\begin{aligned} V(R_a) &= 100 \\ V(R_b) &= 0 \end{aligned}$$

- **Now, we have a set of equations.**
- The analytical solution to this spherically symmetric problem would be:

$$V(R) = \frac{R_a(R_b - r)}{(R_b - R_a)r}V_a$$

We had a set of equations.
We transformed this boundary value problem into a first-order system of equations.

$$
\begin{aligned}
V_1 &= V \\
V_2 &= V' \\
\frac{dV_1}{dr} &= V_2 \\
\frac{dV_2}{dr} &= -\frac{2}{r}V_2
\end{aligned}
$$

(**Example py-file:** laplaceequation.py)

Solution (y_2(0)) is found.

Step x	RK4 y	Exact y	RK4-Exact Error	SciPy y
1.00	100.0000000000000000	100.0000000000000000	0.0000000000000000	100.0000000000000000
1.01	98.0198019839246939	98.0198019801980251	0.0000000037266688	98.0198019367090438
1.02	96.0784313796942087	96.0784313725490051	0.0000000071452035	96.0784311100688484
1.03	94.1747572918313125	94.1747572815533829	0.0000000102779296	94.1747572423479085
1.04	92.3076923208377877	92.3076923076923066	0.0000000131454811	92.3076924358639843
1.05	90.4761904919574107	90.4761904761904674	0.0000000157669433	90.4761904781525459
1.06	88.6792453011787103	88.6792452830188580	0.0000000181598523	88.6792450778453087
1.07	86.9158878708077509	86.9158878504672714	0.0000000203404795	86.9158877756185717
1.08	85.1851852075089937	85.1851851851851762	0.0000000223238175	85.1851853046562582
1.09	83.4862385562338716	83.4862385321100788	0.0000000241237927	83.4862385529654887
1.10	81.8181818439350366	81.8181818181817988	0.0000000257532378	81.8181816662801964
1.90	5.2631579000375517	5.2631578947367981	0.0000000053007536	5.2631578973265096
1.91	4.7120418895737144	4.7120418848167098	0.0000000047570046	4.7120419114658265
1.92	4.1666666708828348	4.1666666666666217	0.0000000042162132	4.1666667076922632
1.93	3.6269430088597163	3.6269430051813032	0.0000000036784131	3.6269430347455405
1.94	3.0927835082982367	3.0927835051545949	0.0000000031436418	3.0927835227867848
1.95	2.5641025667144475	2.5641025641025195	0.0000000026119280	2.5641025680749587
1.96	2.0408163286138654	2.0408163265305679	0.0000000020832975	2.0408163401195418
1.97	1.5228426411516396	1.5228426395938643	0.0000000015577752	1.5228426702649041
1.98	1.0101010111363471	1.0101010101009658	0.0000000010353813	1.0101010409498454
1.99	0.5025125633301593	0.5025125628140259	0.0000000005161334	0.5025125850668605
2.00	0.0000000000000022	-0.0000000000000444	0.0000000000000466	0.0000000206041846
2.01	-0.4975124383238556	-0.4975124378109786	0.0000000005128770	-0.4975124163493223

Figure 5.11: Solution for the Boundary Value Problem for the ODE: $V'' = -\frac{2}{r}V'$.

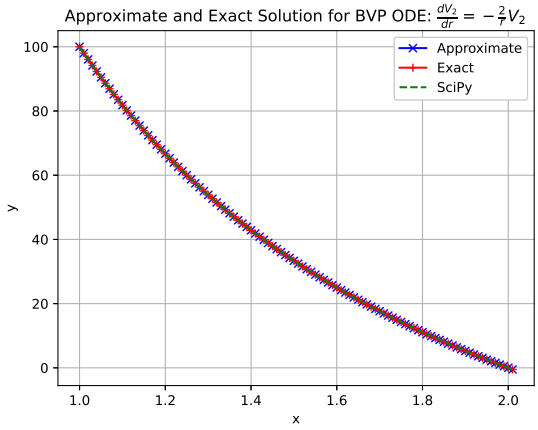


Figure 5.12: Solution for the Boundary Value Problem for the ODE: $V'' = -\frac{2}{r}V'$.

5.3 Eigenvalue Problems

1. **Initial Value Problems.**
2. **Boundary Value Problems**
3. **Eigenvalue (characteristic-value) Problems** Even if the boundary conditions of the differential equation are available, the solutions can only exist for some *specific values* of a parameter in the system. For example,

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi(x) = E\psi(x)$$

   - In Schrödinger equation, there are solutions that $\psi(x)$ goes to zero at infinity for certain values of the energy E.
   - These E_n values satisfying this condition are the **eigenvalues** of the differential equation.

- See the following 2nd degree differential equation:

$$y'' = f(x, y, y' : \lambda)$$

- Again, let the boundary conditions of this equation to be given at both ends.
- If these conditions can only be satisfied for *certain λ values*, we call it the *eigenvalue problem.*
  - e.g.: Vibrations of a wire with both ends fixed give stable solution only for *certain wavelengths.*
  - e.g.: Solutions of the Schrödinger equation that are zero at infinity exist only for *certain energy eigenvalues.*
- Some boundary value problems in physics/engineering have a solution based on an eigenvalue.
- **In terms of numerical solution, boundary value and eigenvalue problems are solved by the same method.**

1 For example, standing waves are the solutions of the following differential equation for a string fixed at both ends:

$$\frac{d^2y(x)}{dx^2} + k^2y(x) = 0$$

Here $k = 2\pi/\lambda$ represents the wavenumber.

- If the two ends of the L-length string are fixed, then corresponding boundary values are:
$$y(0) = y(L) = 0$$
- Although there are sufficient boundary conditions, there are only solutions for *certain* k values.
- It is impossible to satisfy these boundary conditions for other k values.
- The standing wave solution in the string:
$$y(x) = Asinkx + Bcoskx$$
- Boundary conditions to find the coefficients A, B;
$$\begin{aligned} y(0) = 0 \rightarrow \quad & A * 0 + B * 1 = 0 \quad && \rightarrow B = 0 \\ y(L) = 0 \rightarrow \quad & AsinkL = 0 && \rightarrow kL = n\pi \ (n = 1, 2, \ldots) \end{aligned}$$
- According to these results, there is only one set of solution as $k = \pi/L, 2\pi/L, \ldots$ values. ($k_n = n\pi/L \rightarrow$ *eigenvalue(s).*)

2 Another example is the Schrödinger equation in quantum mechanics. If we write it in one-dimensional space,
$$-\frac{\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x)$$
Here $V(x)$ is the potential energy function of the particle, and E is its total energy.

- Boundary conditions for the wave function are given for $x = \pm\infty$:
$$\psi(\pm\infty) \rightarrow 0$$
- Again, this differential equation has solutions satisfying the boundary conditions only for *certain* E eigenvalues.

For eigenvalue problems, the trial-and-error (shooting) method is also used.

- However, an estimated value is given to the eigenvalue instead of giving to the derivatives at the boundary.
- Then, a trial solution is obtained.
- By comparing this trial solution with the value at the boundary condition, the eigenvalue is readjusted and another trial is performed.
- Finally, the solution is to be found when the true eigenvalue is approached within a certain margin of error.

5.3.1 Standing Waves on a String

- The wave equation and boundary conditions in a string of length $L = 1\ m$ with both fixed ends are as follows:

$$\frac{d^2y(x)}{dx^2} + k^2y(x) = 0 \quad \& \quad y(0) = y(1) = 0$$

Firstly, transform this quadratic equation into a system of linear (first degree) equations:

$$y \to y_1$$
$$\frac{dy}{dx} \to y_2$$

with these values (y_1 *and* y_2), the system of equations to be solved: (**Now, we have a set of equations.**)

$$\frac{dy_1}{dr} = y_2$$
$$\frac{dy_2}{dr} = -k^2y_1$$

and the boundary conditions are:

$$y(0) = 0$$
$$y(1) = 0$$

- Here, the trial-and-error approach differs from the previously discussed boundary value problem.
- Different estimates for $y_2(0)$ values do not make it zero at the other boundary.
- Instead, an **estimated value for the k eigenvalue** is taken and a solution search is initiated.
- Searching continues by increasing the value of k until the boundary condition (here $y(1) = 0$) at the other end is satisfied.
- For example, the solution at the other boundary is $y_{1k}(1)$ for a given value of k.
- Accordingly, the next step is find the root of the following equation:

$$F(k) = y_{1k}(1) - y(1) = 0$$

When we encountered an eigenvalue k, then $F(k)$ will change sign as indicating the root.

- (**Example py-file:** The program to find the 5 smallest of the k eigenvalues in a string: standingwawes.py)
- The program can find the eigenvalues $k = n\pi/L = \pi, 2\pi, 3\pi, \ldots$ on a string of length L=1 m.
- However, the error margin is to be increased by increasing eigenvalues (see k_n/π values).

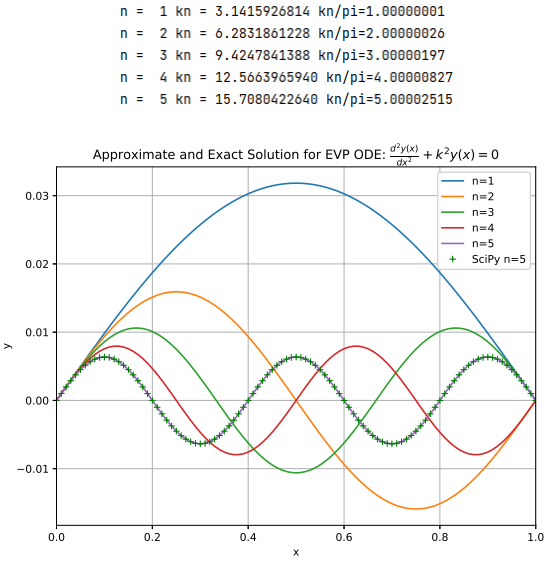


Figure 5.13: Solution for the Eigenvalue Problem for the ODE: $\frac{dy_2}{dx} = -k^2 y_1$.

5.3.2 Numerical Solutions of Schrödinger Equation

- In quantum mechanics, the Schrödinger equation is used to find the eigenvalues and eigenfunctions of the particle moving in one dimension at the potential $V(x)$:

$$-\frac{\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x)$$

- An **analytical solution** to this equation is available for only very few potential functions such as harmonic oscillator, infinite well, hydrogen atom, ...

- Therefore, **numerical solutions** of the Schrödinger equation is an indispensable tool in physics research.
- The numerical solution of the Schrödinger equation is complicated for general solutions of the problem.
- However, if we assume the potential function to be as symmetric, the problem can be solved in a much easier way.
- For a symmetric potential,

$$V(-x) = V(x)$$

- Therefore, the solutions of the Schrödinger equation also fall into two groups:

$$\begin{array}{ll} \text{Symmetrical wave functions:} & \psi(-x) = \psi(x) \\ \text{Antisymmetric wave functions:} & \psi(-x) = -\psi(x) \end{array}$$

- This property allows us to determine exactly the initial conditions necessary to start the eigenvalue problem.

$$\begin{array}{llll} \text{Symmetric (even) wave functions:} & \psi(0) = 1 & \& & \psi'(0) = 0 \\ \text{Antisymmetric (odd) wave functions:} & \psi(0) = 0 & \& & \psi'(0) = 1 \end{array}$$

5.3.3 Hydrogen Atom

- In quantum mechanics, the hydrogen atom is considered as a system of electrons with a charge of $-e$ around a proton with a charge of $+e$.
- If the electrostatic potential energy between the electron-proton is substituted in the Schrödinger equation as $V(r) = -e^2/r$,

$$-\frac{\hbar^2}{2m_r}\nabla^2\psi(\vec{r}) - \frac{e^2}{r}\psi(\vec{r}) = E\psi(\vec{r})$$

  Here $m_r = m_e m_p/(m_e + m_p)$ is the reduced mass of the electron-proton system.
- Since the potential energy depends only on the distance r, the solution is defined with the spherical coordinates (r, θ, ϕ) in three-dimensional space:

$$\psi(r, \theta, \phi) = R(r)Y(\theta, \phi)$$

- The solutions of the equation provided by the angular variables (θ, ϕ) are independent of the $V(r)$ potential and consist of functions called *spherical harmonics* $Y(\theta, \phi)$.

- The equation provided by the $R(r)$ function is called *radial Schrödinger equation.*

$$-\frac{\hbar^2}{2m_r}\left[\frac{d^2R}{dr^2}+\frac{2}{r}\frac{dR}{dr}\right]+\left[\frac{l(l+1)\hbar^2}{2m_r r^2}-\frac{e^2}{r}\right]R(r)=-|E|R(r)$$

- Attempt to find the bound energy (eigen)values and wave (eigen)functions of the radial Schrödinger equation numerically :

- First, it is necessary to make the radial equation dimensionless bu defining a new wavefunction:

$$u(r)=rR(r)$$

- The radial equation in terms of this new function $u(r)$ becomes simpler:

$$\frac{d^2u}{dr^2}-\left[\frac{l(l+1)}{r^2}-\frac{2m_re^2}{\hbar^2 r}\right]u(r)=-\frac{2m_r|E|}{\hbar^2}u(r)$$

- Now, a variable change is made as the following:

$$k=\frac{\sqrt{2m_r|E|}}{\hbar},\quad \rho=2kr,\quad \lambda^2=\frac{1}{ka_0}=\frac{\Re}{|E|}$$

  Here, the Rydberg constant $\Re$ and the Bohr radius a_0 are defined as:

$$a_0=\frac{\hbar^2}{m_re^2},\quad \Re=\frac{\hbar^2}{2m_ra_0^2}$$

- As a result of these changes, the dimensionless radial equation becomes:

$$\boxed{\frac{d^2u}{d\rho^2}-\frac{l(l+1)}{\rho^2}u+\left(\frac{\lambda}{\rho}-\frac{1}{4}\right)u=0} \tag{5.5}$$

  where l is *orbital quantum number* and λ is *principal quantum number.*

Numerical Solution. Firstly, transform this quadratic Equation 5.5 into a system of linear (first degree) equations:

$$u \to y_1$$
$$\frac{du}{d\rho} \to y_2$$

with these values (y_1 *and* y_2), the system of equations to be solved: (**Now, we have a set of equations.**)

$$\frac{dy_1}{d\rho} = y_2$$
$$\frac{dy_2}{d\rho} = \left[\frac{l(l+1)}{\rho^2} - \left(\frac{\lambda}{\rho} - \frac{1}{4}\right)\right] y_1$$

- For the initial conditions:
  - $u(0) \sim 0$
  - Since the absolute magnitude of the wave function has no physical meaning, the arbitrary value $u'(0) = 1$ can be taken.
- Then;

$$y_1(0) = 0 \quad \text{and} \quad y_2(0) = 1$$

(**Example py-file:** Program that solves the radial Schrödinger equation for the hydrogen atom: hydrogenatom.py)

- Program does not graph the $u(r)$ functions, but the $|u|^2$ probability densities, which is physically meaningful.
- It calculates according to the l quantum number which is supplied by the user.
- The error margin is to be increased by increasing n quantum number.

```
Enter Orbital Quantum Number l (0,1,2,3 as s,p,d,f) =0
n =  1 Eigenvalue (lambda) = 1.0000014120
n =  2 Eigenvalue (lambda) = 2.0001120977
n =  3 Eigenvalue (lambda) = 3.0035437774
```

Figure 5.14: Solution for the Eigenvalue Problem for the ODE: $\frac{dy_2}{d\rho} = \left[\frac{l(l+1)}{\rho^2} - \left(\frac{\lambda}{\rho} - \frac{1}{4}\right)\right] y_1$.

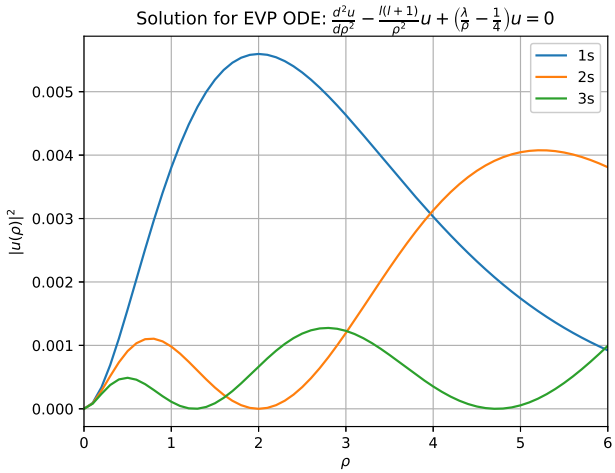


Figure 5.15: Solution for the Eigenvalue Problem for the ODE: $\frac{dy_2}{d\rho} = \left[\frac{l(l+1)}{\rho^2} - \left(\frac{\lambda}{\rho} - \frac{1}{4}\right)\right] y_1$.

5.4 Special Functions

- Consider a straight rod along which there is a uniform flow of heat.
  - Let $u(x,t)$ denote the temperature of the rod at time t and location x.
  - Let $q(x,t)$ denote the rate of heat flow.
- The expression $\partial q/\partial x$ denotes the rate at which the rate of heat flow changes per unit length and therefore measures the rate at which heat is accumulating at a given point x at time t.
- If heat is accumulating, the temperature at that point is rising, and the rate is denoted by $\partial u/\partial t$.

1 The principle of conservation of energy leads to $\partial q/\partial x = k\partial u/\partial t$, where k is the specific heat of the rod.

- This means that the rate at which heat is accumulating at a point is proportional to the rate at which the temperature is increasing.

2 A second relationship between q and u is obtained from Newton's law of cooling, which states that $q = K(\partial u/\partial x)$.

- Elimination of q between these equations leads to

$$\frac{\partial^2 u}{\partial x^2} = (k/K)\frac{\partial u}{\partial t}$$

the partial differential equation for one-dimensional heat flow.

- The partial differential equation for heat flow in three dimensions takes the form

$$\boxed{\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial y x^2} = (k/K)\frac{\partial u}{\partial t}}$$

- Often written as

$$\boxed{\nabla^2 u = (k/K)\frac{\partial u}{\partial t}}$$

where the symbol ∇, called del or nabla, is known as the Laplace operator.

- Another example to PDEs for dealing with wave propagation problem:

$$\nabla^2 u = (1/c^2)\frac{\partial^2 u}{\partial t^2}$$

  where c is the speed at which the wave propagates.

- PDEs are harder to solve than ordinary differential equations (ODEs).

- However, the PDEs associated with wave propagation and heat flow can be reduced to a system of ODEs through a process known as ***separation of variables***.

- These ODEs depend on the choice of coordinate system, which in turn is influenced by the physical configuration of the problem.

- The solutions of these ODEs form the majority of the ***special functions*** of mathematical physics.

- In the broad sense, a set of several classes of functions that arise in the solution of both theoretical and applied problems in various branches.

- In the narrow sense, the special functions of mathematical physics, which arise when solving PDEs by the method of separation of variables.

- Special functions can be defined by means of power series, generating functions, infinite products, repeated differentiation, integral representations, differential, difference, integral, and functional equations, trigonometric series, or other series in orthogonal functions.

- For example, in solving the equations of heat flow or wave propagation in cylindrical coordinates, the method of separation of variables leads to Bessel's differential equation, a solution of which is the Bessel function, denoted by $J_n(x)$.

- A polynomial ("many terms") is defined as an expression that consist of variables, coefficients and exponents.

- A polynomial can have:
  - variables (like x and y)
  - constants/coefficients (like 6, -10, or $^{3/2}$)
  - exponents (like the 2 in y^2)

  - that can be combined using addition, subtraction, multiplication and division
  - but not division by a variable (so something like 2/x is not correct)
  - a monomial is the product of non-negative powers of variables and will only have one term. 13, 3x, $4y^2$, ...
  - a binomial is the sum of two monomials. 3x + 1, 2x + y, ...
  - a trinomial is the sum of three monomials. x^2 + 2x + 1, 2x + 3y + 2, ...
  - can have one or more terms, but not an infinite number of terms.

- The standard form of a polynomial refers to writing a polynomial in the descending power of the variable.

$$2x^3 - 4x^2 + 7x - 4$$

5.4.1 Legendre Polynomials

- The Legendre polynomials $P_\ell(x)$, sometimes called Legendre functions of the first kind, Legendre coefficients, or zonal harmonics are solutions to the Legendre differential equation.

- The Legendre polynomials satisfy the second-order differential equation.

$$\boxed{(1-x^2)\frac{\mathrm{d}^2y}{\mathrm{d}x^2} - 2x\frac{\mathrm{d}y}{\mathrm{d}x} + \ell(\ell+1)y = 0}$$

  where $y = P_\ell(x)$

- This equation has two regular singular points $x = \pm 1$ where the leading coefficient $(1 - x^2)$ vanishes.

- Solutions of Legendre equations are Legendre polynomials

$$\boxed{P_\ell(x) = \frac{1}{2^\ell \ell!}\left(\frac{\mathrm{d}}{\mathrm{d}x}\right)^\ell (x^2-1)^\ell}$$

- If ℓ $(0 \leq \ell \leq \infty)$ is an integer, they are polynomials and make up an infinite set of functions of the variable x.

- We therefore have a function $P_0(x)$, another function $P_\ell(x)$, and an infinite number of additional functions belonging to the set of Legendre polynomials.

- Introduce a (**generating**) function $\Phi(x,h)$ of two variables, known as a generating function for the definition of the Legendre polynomials.

$$\boxed{\Phi(x,h) = (1-2xh+h^2)^{-1/2}}$$

  - The first variable, x, is the same variable that appears as the argument of the Legendre polynomials.
  - The second variable, h, is an auxiliary variable with no particular meaning.

- Think of Φ as a function of a single variable h ($\Phi = \Phi(h)$) and expand as a Taylor expansion in powers of h

$$\begin{aligned}\Phi(h) &= \Phi(0) + \left.\frac{d\Phi}{dh}\right|_{h=0} h + \frac{1}{2!}\left.\frac{d^2\Phi}{dh^2}\right|_{h=0} h^2 + \frac{1}{3!}\left.\frac{d^3\Phi}{dh^3}\right|_{h=0} h^3 + \ldots \\ &= \sum_{\ell=0}^{\infty} \frac{1}{\ell!}\left.\frac{d^\ell\Phi}{dh^\ell}\right|_{h=0} h^\ell\end{aligned}$$

- Restore the x-dependence of the generating function. This doesn't change the general appearance of the Taylor expansion but written as partial derivatives instead of total derivatives.

$$\Phi(x,h) = \sum_{\ell=0}^{\infty} \frac{1}{\ell!}\left.\frac{\partial^\ell\Phi}{\partial h^\ell}\right|_{h=0} h^\ell = \sum_{\ell=0}^{\infty} P_\ell(x) h^\ell$$

- Right hand side of this equation is the formal definition of the Legendre polynomials. They are identified as the coefficients in the Taylor expansion of the generating function about $h = 0$.

- Let us use this equation to calculate the first few polynomials.

  - For $\ell = 0$ we are instructed to take no derivatives, and to evaluate the generating function at $h = 0$. This gives $P_0(x) = 1$; the zeroth polynomial is actually a constant.
  - Moving on to $\ell = 1$, we must differentiate Φ once with respect to h. Evaluating this at $h = 0$ and dividing by $1! = 1$ gives $P_1(x) = x$.
  - For $\ell = 2$ we differentiate Φ twice. Evaluating this at $h = 0$ and dividing by $2! = 2$ produces $P_2(x) = 1/2(3x^2 - 1)$. We can just keep going like this, and generate any number of polynomials.

- When ℓ is even, $P_\ell(x)$ contains only even powers of x, starting with x^ℓ and ending with x^0.

- When ℓ is odd, $P_\ell(x)$ contains only odd powers of x, starting with x^ℓ and ending with x.
- $P_\ell(x)$ is an even function of x when ℓ is even, and an odd function of x when ℓ is odd.
- The first few Legendre polynomials are given by

$$\begin{aligned} P_0 &= 1 \\ P_1 &= x \\ P_2 &= \frac{1}{2}(3x^2-1) \\ P_3 &= \frac{1}{2}(5x^3-3x) \\ P_4 &= \frac{1}{8}(35x^4-30x^2+3) \\ P_5 &= \frac{1}{8}(63x^5-70x^3+15x) \\ P_6 &= \frac{1}{16}(231x^6-315x^4+105x^2-5) \end{aligned}$$

- Recursion relation:

$$\boxed{P_\ell(x) = \frac{1}{\ell}[(2\ell-1)xP_{\ell-1}(x) - (\ell-1)P_{\ell-2}(x)]}$$

Example py-file: The program to find first 6 Legendre polynomials: myLegendre.py

```
P_0= 1
P_1= x
P_2= 3*x**2/2 - 1/2
P_3= 5*x**3/2 - 3*x/2
P_4= 35*x**4/8 - 15*x**2/4 + 3/8
P_5= 63*x**5/8 - 35*x**3/4 + 15*x/8
P_6= 231*x**6/16 - 315*x**4/16 + 105*x**2/16 - 5/16

1 x  --> P1
     2
1.5 x - 0.5  --> P2
     3
2.5 x - 1.5 x  --> P3
       4             3        2
4.375 x + 4.857e-16 x - 3.75 x + 2.429e-16 x + 0.375  --> P4
       5        3             2
7.875 x - 8.75 x - 4.372e-16 x + 1.875 x  --> P5
       6         4             3         2
14.44 x - 19.69 x + 1.603e-15 x + 6.562 x - 0.3125  --> P6
```

Figure 5.16: First 6 Legendre Polynomials $P_\ell(x)$ with Recursion Relation: $P_\ell(x) = \frac{1}{\ell}[(2\ell-1)xP_{\ell-1}(x) - (\ell-1)P_{\ell-2}(x)]$.

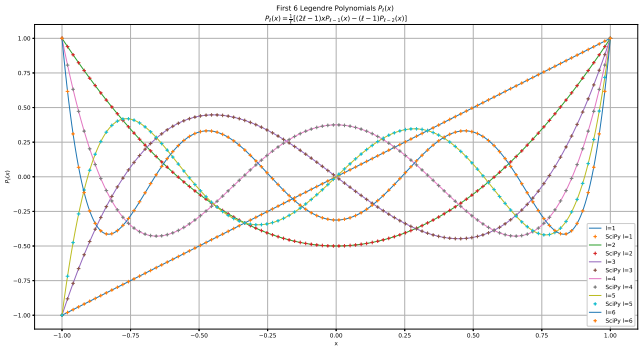


Figure 5.17: Plot of first 6 $P_\ell(x)$.

5.4.2 Hermite Polynomials

- The Hermite polynomials $H_k(x)$ are solutions to the Hermite differential equation of the form

$$a(x)y'' + b(x)y' + c(x)y = 0$$

where $a(x) = 1$, $b(x) = -2x$ and $c(x) = 2k$ (positive integer parameter k)

$$\boxed{\frac{d^2y}{dx^2} - 2x\frac{dy}{dx} + 2ky = 0}$$

- y_k is a solution of the Hermite equation. Therefore, defining $H^k(x) = y_k$.

- A natural one to define Hermite polynomials is through the so-called Rodrigues'formula:

$$\boxed{H_k(x) = (-1)^k e^{x^2} \frac{d^k}{dx^k}\left[e^{-x^2}\right]}$$

- The first few Hermite polynomials are given by

$$\begin{aligned} H_0 &= 1 \\ H_1 &= 2x \\ H_2 &= 4x^2 - 2 \\ H_3 &= 8x^3 - 12x \\ H_4 &= 16x^4 - 48x^2 + 12 \\ H_5 &= 32x^5 - 160x^3 + 120x \\ H_6 &= 64x^6 - 480x^4 + 720x^2 - 120 \end{aligned}$$

- Recursion relation:

$$\boxed{H_{k+1}(x) = 2xH_k(x) - 2kH_{k-1}(x)}$$

Example py-file: The program to find first 6 Hermite polynomials: myHermite.py

```
P_0= 1
P_1= x
P_2= 3*x**2/2 - 1/2
P_3= 5*x**3/2 - 3*x/2
P_4= 35*x**4/8 - 15*x**2/4 + 3/8
P_5= 63*x**5/8 - 35*x**3/4 + 15*x/8
P_6= 231*x**6/16 - 315*x**4/16 + 105*x**2/16 - 5/16

1 x  --> P1
     2
1.5 x - 0.5  --> P2
     3
2.5 x - 1.5 x  --> P3
       4            3        2
4.375 x + 4.857e-16 x - 3.75 x + 2.429e-16 x + 0.375  --> P4
       5        3            2
7.875 x - 8.75 x - 4.372e-16 x + 1.875 x  --> P5
       6         4             3         2
14.44 x - 19.69 x + 1.603e-15 x + 6.562 x - 0.3125  --> P6
```

Figure 5.18: First 6 Hermite Polynomials $H_k(x)$ with Recursion Relation: $H_{k+1}(x) = 2xH_k(x) - 2kH_{k-1}(x)$.

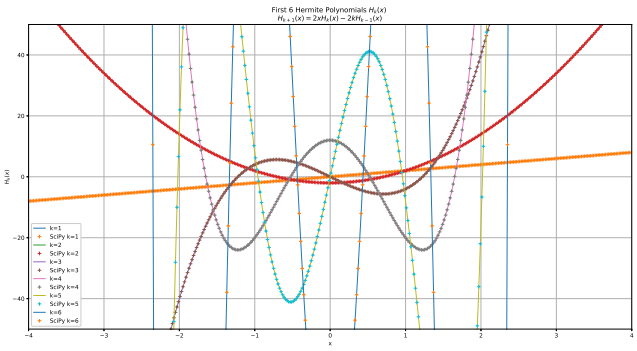


Figure 5.19: Plot of first 6 $H_k(x)$.

Quantum Harmonic Oscillator

- The quantum harmonic oscillator as analog of the classical one is often used as an approximate model for the behavior of some quantum systems.
- It is one of the few quantum-mechanical systems for which an exact, analytical solution is known.
- The Hamiltonian for a particle of mass m moving in one dimension in a potential $V(x) = 1/2kx^2$ is

$$\hat{H} = \frac{\hat{p}^2}{2m} + \frac{1}{2}k\hat{x}^2 = \frac{\hat{p}^2}{2m} + \frac{1}{2}m\omega^2\hat{x}^2$$

  where $\hat{x}$ is the position operator, and $\hat{p}$ is the momentum operator (given by $\hat{p} = -i\hbar\partial/\partial x$ in the coordinate basis).
- The first term in the Hamiltonian represents the kinetic energy of the particle, and the second term represents its potential energy, as in Hooke's law.
- Then, Schrödinger equation becomes

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + \frac{1}{2}kx^2\psi = E\psi$$

- with the change of variable, $q = (mk/\hbar^2)^{1/4}x$, this equation becomes

$$-\frac{1}{2}\frac{d^2\psi}{dq^2} + \frac{1}{2}q^2\psi = \frac{E}{\hbar\omega}\psi$$

where $\omega = \sqrt{k/m}$ is the angular frequency of the oscillator.

- This differential equation has an exact solution in terms of a quantum number $\nu = 0, 1, 2, \ldots$:

$$\boxed{\psi(q) = N_\nu H_\nu(q) e^{-q^2/2}}$$

where $N_\nu = (\sqrt{\pi} 2^\nu \nu!)^{-1/2}$ is a normalization constant.

- The function $H_\nu(q)$ is the physicists' Hermite polynomials of order ν, defined by:

$$H_\nu(q) = (-1)^\nu e^{q^2} \frac{d^\nu}{dq^\nu} \left(e^{-q^2} \right)$$

- The corresponding energy levels are

$$E_\nu = \hbar\omega \left(\nu + \frac{1}{2} \right) = (2\nu + 1) \frac{\hbar}{2} \omega$$

- Recursion formula:

$$\boxed{H_{\nu+1}(q) = 2qH_\nu(q) - 2\nu H_{\nu-1}(q)}$$

with the first two: $H_0 = 1$ and $H_1 = 2q$.

Example py-file: The program to find the harmonic oscillator wavefunctions/probability densities for up to 4 vibrational energy levels with the harmonic potential, $V = q^2/2$. QHO.py

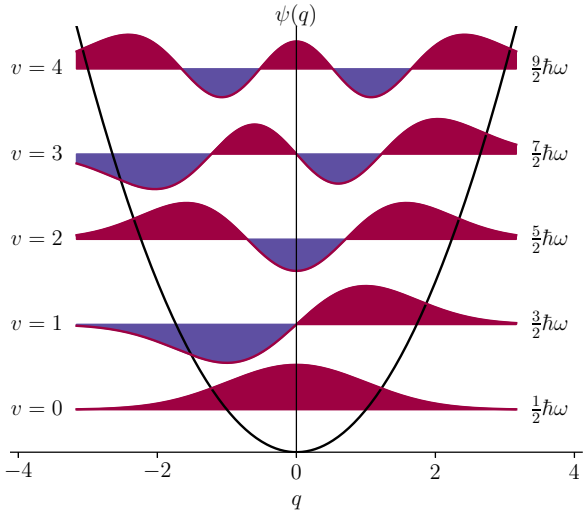


Figure 5.20: Wavefunction representations for the first 5 bound eigenstates, $\nu = 0 - 4$.

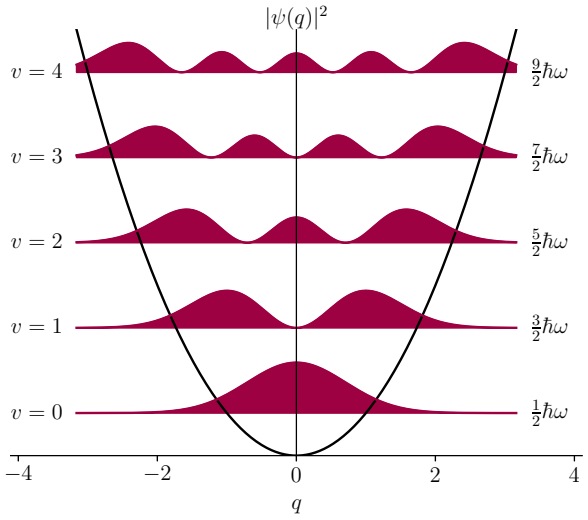


Figure 5.21: Corresponding probability densities.

Chapter 6

Linear Algebra and Matrix Computing

6.1 Solving Sets of Equations

- Solving sets of linear equations and eigenvalue problems are the most frequently used numerical procedures when real-world situations are modelled.
- Analytical solution may be feasible when the number of unknowns is small.
- However, computers outperforms to solve large systems of linear equations such as with 100 unknowns in a reasonable time.

1. **Matrices and Vectors**. Reviews concepts of matrices and vectors in preparation for their use.
2. **Elimination Methods.** Describes classical methods that change a system of equations to forms that allow getting the solution by back-substitution and shows how the errors of the solution can be minimized.
3. **The Inverse of a Matrix.** Shows how an important derivative of a matrix, its inverse, can be computed. It shows when a matrix **cannot be inverted** and tells of situations where **no unique solution** exists to a system of equations.
4. **Iterative Methods.** It is described how a linear system can be solved in an entirely different way, by beginning with an initial estimate of the solution.

6.2 Matrices and Vectors

- When a system of equations has more than two or three equations, it is difficult to discuss them without using *matrices* and *vectors*.
- A *matrix* is a rectangular array of numbers in which not only the value of the number is important but also its position in the array.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} = [a_{ij}], \quad \begin{matrix} i = 1, 2, \dots, n, \\ j = 1, 2, \dots, m \end{matrix}$$

- An $m \times n$ matrix times as $n \times 1$ vector gives an $m \times 1$ product ($m \times \mathbf{nn} \times 1$)
.

- The general relation for $\boxed{Ax = b}$ is

$$b_i = \sum_{k=1}^{No.of.cols.} a_{ik}x_k, \qquad i = 1, 2, \ldots, \#\ of\ rows$$

  where A is a matrix, x and b are vectors (column vectors).

- This definition of matrix multiplication permits us to write the set of linear equations

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

  - If the equations in any two rows is interchanged, the solution does not change.
  - Multiplying the equation in any row by a constant does not change the solution.
  - Adding or subtracting the equation in a row to another row does not change the solution.

- Much more simply in matrix notation, as $Ax = b$ where

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- Example,

Matrix notation:

$$\begin{bmatrix} 3 & 2 & 4 \\ 1 & -2 & 0 \\ -1 & 3 & 2 \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 14 \\ -7 \\ 2 \end{bmatrix} \Leftrightarrow$$

Set of equations:

$$\begin{aligned} 3x_1 + 2x_2 + 4x_3 &= 14 \\ x_1 - 2x_2 &= -7 \\ -x_1 + 3x_2 + 2x_3 &= 2 \end{aligned}$$

- **Square matrices** are particularly important when a system of equations is to be solved.

6.2.1 Some Special Matrices and Their Properties

- **Symmetric matrix.**

A square matrix is called a **symmetric** matrix when the pairs of elements in similar positions across the diagonal are equal.

$$\begin{bmatrix} 1 & x & y \\ x & 2 & z \\ y & z & 3 \end{bmatrix}$$

- The **transpose** of a matrix is the matrix obtained by writing the rows as columns or by writing the columns as rows.
  - The symbol for the transpose of matrix A is A^T.

$$A = \begin{bmatrix} 3 & -1 & 4 \\ 0 & 2 & 3 \\ 1 & 1 & 2 \end{bmatrix}, \qquad A^T = \begin{bmatrix} 3 & 0 & 1 \\ -1 & 2 & 1 \\ 4 & -3 & 2 \end{bmatrix}$$

- If all the elements above/below the diagonal are zero, a matrix is called ***lower/upper-triangular*** (L/U);

$$L = \begin{bmatrix} x & 0 & 0 \\ x & x & 0 \\ x & x & x \end{bmatrix}, \qquad U = \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & 0 & x \end{bmatrix}$$

- **We will deal with square matrices.**
- **Sparse matrix.** In some important applied problems, only a few of the elements are nonzero.
- Such a matrix is termed a **sparse** matrix and procedures that take advantage of this sparseness are of value.
- Division of matrices is not defined, but we will discuss the **inverse** of a matrix.
- The **determinant** of a square matrix is a number.
  - The method of calculating determinants is a lot of work if the matrix is of large size.
  - Methods that triangularize a matrix, as described in next section, are much better ways to get the determinant.

- If a matrix, B, is triangular (either upper or lower), its determinant is just the product of the diagonal elements:

$$det(B) = \Pi B_{ii}, \qquad i = 1, \ldots, n$$

$$det \begin{vmatrix} 4 & 0 & 0 \\ 6 & -2 & 0 \\ 1 & -3 & 5 \end{vmatrix} = -40$$

If we have a square matrix and the coefficients of the determinant are nonzero, there is a unique solution.

- Determinants can be used to obtain the **characteristic polynomial** and the **eigenvalues** of a matrix, which are the roots of that polynomial.
- If a matrix is triangular, **its eigenvalues are equal to the** diagonal elements.
- This follows from the fact that
  - its determinant is just the product of the diagonal elements and
  - its characteristic polynomial is the product of the terms $(a_{ii} - \lambda)$ with i going from 1 to n, the number of rows of the matrix.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix},$$

$$det(A - \lambda I) = det \begin{vmatrix} 1-\lambda & 2 & 3 \\ 0 & 4-\lambda & 5 \\ 0 & 0 & 6-\lambda \end{vmatrix} = (1-\lambda)(4-\lambda)(6-\lambda)$$

  whose roots are clearly 1, 4, and 6.

- It does not matter if the matrix is upper- or lower-triangular.

6.3 Elimination Methods

- Solve a set of linear (variables with first power) equations.
- If we have a system of equations that is of an *upper- triangular* form

$$\begin{aligned} 5x_1 + 3x_2 - 2x_3 &= -3 \\ 6x_2 + x_3 &= -1 \\ 2x_3 &= 10 \end{aligned}$$

  Then, we have the solution as: $x_1 = 2,\ x_2 = -1,\ x_3 = 5$

- If NOT: Change the matrix of coefficients $\Longrightarrow$ upper- triangular. Consider this example of three equations:

$$\begin{array}{rrr} 4x_1 - 2x_2 + x_3 = 15 & 4x_1 - 2x_2 + x_3 = 15 & 4x_1 - 2x_2 + x_3 = 15 \\ -3x_1 - x_2 + 4x_3 = 8 & -10x_2 + 19x_3 = 77 & -10x_2 + 19x_3 = 77 \\ x_1 - x_2 + 3x_3 = 13 & -2x_2 + 11x_3 = 37 & -72x_3 = -216 \end{array}$$

  Now we have a **triangular system** and the solution is readily obtained;

  1. obviously $x_3 = 3$ from the third equation,
  2. and **back-substitution** into the second equation gives $x_2 = -2$.
  3. We continue with back-substitution by substituting both x_2, and x_3 into the first equation to get $x_1 = 2$.

- Notice to the values in this example. They are getting bigger!
- The essence of any elimination method is to reduce the coefficient matrix to a triangular matrix and then use back-substitution to get the solution.
- We now present the same problem, solved in exactly the same way, in matrix notation;

$$\begin{bmatrix} 4 & -2 & 1 \\ -3 & -1 & 4 \\ 1 & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 15 \\ 8 \\ 13 \end{bmatrix}$$

- So we work with the matrix of coefficients *augmented* with the right-hand-side vector.
- We perform elementary row transformations to convert A to upper-triangular form:

$$\begin{bmatrix} 4 & -2 & 1 & 15 \\ -3 & -1 & 4 & 8 \\ 1 & -1 & 3 & 13 \end{bmatrix}, \qquad \begin{array}{r} 3R_1 + 4R_2 \rightarrow \\ (-1)R_1 + 4R_3 \rightarrow \end{array} \qquad \begin{bmatrix} 4 & -2 & 1 & 15 \\ 0 & -10 & 19 & 77 \\ 0 & -2 & 11 & 37 \end{bmatrix},$$

$$2R_2 - 10R_3 \rightarrow \qquad \begin{bmatrix} 4 & -2 & 1 & 15 \\ 0 & -10 & 19 & 77 \\ 0 & 0 & 72 & -216 \end{bmatrix} \Rightarrow \begin{array}{r} 4x_1 - 2x_2 + x_3 = 15 \\ -10x_2 + 19x_3 = 77 \\ -72x_3 = -216 \end{array}$$

- The back-substitution step can be performed quite mechanically by solving the equations in reverse order. That is, $x_3 = 3, x_2 = -2, x_1 = 2$. *Same solution with the non-matrix notation.*

- During the triangularization step, if a **zero** is encountered on the diagonal, we cannot use that row to eliminate coefficients below that zero element.

  – However, in that case, we can continue by **interchanging rows** and eventually achieve an upper-triangular matrix of coefficients.

- The real trouble is finding a zero on the diagonal after we have triangularized.

  – If that occurs, the back-substitution fails, for we cannot divide by zero.

  – It also means that the determinant is zero. There is no solution.

Possible approaches for the solution of the following system of equations with coefficient matrix A.

1 ***Cramer's rule***.

$$\begin{aligned} 6x_1 - 3x_2 + x_3 &= 11 \\ 2x_1 + x_2 - 8x_3 &= -15 \\ x_1 - 7x_2 + x_3 &= 10 \end{aligned} \qquad A = \begin{bmatrix} 6 & -3 & 1 \\ 2 & 1 & -8 \\ 1 & -7 & 1 \end{bmatrix},$$

- Let's denote the matrix by A_j in which the right-hand-side vector are substituted to the j^{th} column of A matrix.

- e.g., the solution for x_1 is expressed as:

$$x_1 = \frac{det(A_1)}{det(A)} = \frac{det\begin{vmatrix} 11 & -3 & 1 \\ -15 & 1 & -8 \\ 10 & -7 & 1 \end{vmatrix}}{det\begin{vmatrix} 6 & -3 & 1 \\ 2 & 1 & -8 \\ 1 & -7 & 1 \end{vmatrix}} = \frac{-285}{-285} = 1$$

  Similarly, the solutions x_2 and x_3 can be written.

- This method is very convenient when the number of unknowns is as few as 3-5.
- However, it is not feasible for systems with a large number of unknowns since determinant calculation requires many multiplication operations.
- For example, calculating a 20×20 determinant with Cramer's rule requires $\approx 10^{20}$ multiplications!

2 ***Inverse matrix***. Another solution is to use the A^{-1} matrix, which is the inverse of the A matrix.

- Multiplying both sides of the equation $A\vec{x} = \vec{b}$ by A^{-1} and considering that $A^{-1}A = 1$,

$$\begin{aligned} A^{-1}A\vec{x} &= A^{-1}\vec{b} \\ \underbrace{A^{-1}A}_{1}\vec{x} &= A^{-1}\vec{b} \\ \vec{x} &= A^{-1}\vec{b} \end{aligned}$$

- However, this approach is also not reasonable since calculating matrix inverses requires a large number of operations.

6.3.1 Gaussian Elimination

- **Therefore, adequate methods should be used in linear equation system solutions such as without calculating determinant or inverse matrix.**
- Two of the most useful methods are **Gaussian elimination** and **L-U decomposition**.
- *Elimination Methods.* While it may be satisfactory for hand computations with small systems, it is inadequate for a large system (numbers may getting bigger!).
- The method that is called *Gaussian elimination* avoids this by subtracting a_{i1}/a_{11} times the first equation from the i^{th} equation to make the transformed numbers in the first column equal to zero.
- We do similarly for the rest of the columns.

- Observe that zeros may be created in the diagonal positions even if they are not present in the original matrix of coefficients.

- A useful strategy to avoid (if possible) such zero divisors in the diagonal positions is to rearrange the equations so as to put the coefficient of largest magnitude on the diagonal at each step.

- This is called **pivoting**.

- *Generalization.* Let the system of equations with N unknowns be given as:

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

  - **First Stage**. Let $a_{11} \neq 0$ (If not, pivoting), then multiply the 1^{st} equation by (a_2/a_{11})) and subtract it from the 2^{nd} equation.
  - **Next,** again multiply the 1^{st} equation by (a_{31}/a_{11}) and subtract from the 3^{rd} equation.
  - **Repeat** this procedure up to the n^{th} equation.

- As a result, the variable x_1 is eliminated in the other equations and the new system of equations becomes:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ \left(a_{22} - \frac{a_{21}}{a_{11}}a_{12}\right)x_2 + \ldots + \left(a_{2n} - \frac{a_{21}}{a_{11}}a_{1n}\right)x_n &= b_2 - \frac{a_{21}}{a_{11}}b_1 \\ \left(a_{32} - \frac{a_{31}}{a_{11}}a_{13}\right)x_2 + \ldots + \left(a_{3n} - \frac{a_{31}}{a_{11}}a_{1n}\right)x_n &= b_3 - \frac{a_{31}}{a_{11}}b_1 \\ &\vdots \\ \left(a_{n2} - \frac{a_{n1}}{a_{11}}a_{1n}\right)x_2 + \ldots + \left(a_{nn} - \frac{a_{nn}}{a_{11}}a_{1n}\right)x_n &= b_n - \frac{a_{n1}}{a_{11}}b_1 \end{aligned}$$

- Notice that in this new system of equations at **first stage**, the coefficient of the j^{th} term in the i^{th} row and the constant b_i are as follows:

$$\begin{aligned} a_{ij}^{(1)} &= a_{ij} - \frac{a_{i1}}{a_{11}}a_{1j} \qquad (i, j = 2, \ldots, n) \\ b_i^{(1)} &= b_i - \frac{a_{i1}}{a_{11}}b_i \qquad (i = 2, \ldots, n) \end{aligned}$$

- **Next,** let $a_{22}^{(1)} \neq 0$ (If not, pivoting) in this new system of equations, then multiply the 2^{nd} equation by $(a_{32}^{(1)}/a_{22}^{(1)})$ and subtract it from the 3^{rd} equation.

- **Repeat** this procedure up to equation n as $n-1$ times to obtain the upper-triangular form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= b_1 \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \ldots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ a_{33}^{(2)}x_3 + \ldots + a_{3n}^{(2)}x_n &= b_3^{(2)} \\ \vdots & \\ a_{nn}^{(n-1)}x_n &= b_n^{(n-1)} \end{aligned}$$

- As seen, the number of unknowns decreases by one in the 2^{nd} and other equations as 1^{st} stage, decreases once again in the 3^{rd} and other equations as 2^{nd} stage and so on. In $(N-1)^{th}$ stage, a single unknown is obtained.

- Then, the j^{th} coefficient of the i^{th} equation as k^{th} stage is as follows:

$$\boxed{a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}a_{kj}^{(k-1)}}{a_{kk}^{(k-1)}}} \quad (i,j = 1,\ldots,n) \tag{6.1}$$

$$\boxed{b_{i}^{(k)} = b_{i}^{(k-1)} - \frac{a_{ik}^{(k-1)}b_{k}^{(k-1)}}{a_{kk}^{(k-1)}}} \quad (i = 1,\ldots,n) \tag{6.2}$$

- After reaching to the upper-triangular form, the solution is almost readily obtained.

- From the last equation in the upper-triangular form:

$$x_n = b_n^{(n-1)}/a_{nn}^{(n-1)}$$

- All other unknowns are obtained consequently by backward-substitution. The general expression would be:

$$\boxed{x_k = \frac{1}{a_{kk}^{(k-1)}}\left[b_k^{(k-1)} - \sum_{j=k+1}^{n} a_{kj}^{(k-1)}x_j\right]} \quad (k = n-1,\ldots,1) \tag{6.3}$$

- Repeat the example of the previous section,

$$\begin{bmatrix} 4 & -2 & 1 & 15 \\ -3 & -1 & 4 & 8 \\ 1 & -1 & 3 & 13 \end{bmatrix}, \qquad \begin{matrix} R_2 - (-3/4)R_1 \rightarrow \\ R_3 - (1/4)R_1 \rightarrow \end{matrix} \qquad \begin{bmatrix} 4 & -2 & 1 & 15 \\ 0 & -2.5 & 4.75 & 19.25 \\ 0 & -0.5 & 2.75 & 9.25 \end{bmatrix},$$

$$R_3 - (-0.5/-2.5)R_2 \rightarrow \qquad \begin{bmatrix} 4 & -2 & 1 & 15 \\ 0 & -2.5 & 4.75 & 19.25 \\ 0 & 0 & 1.8 & 5.40 \end{bmatrix}$$

- The method we have just illustrated is called *Gaussian elimination.*
- In this example, **no pivoting was required** to make the largest coefficients be on the diagonal.
- Back-substitution, gives us $x_3 = 3,\ x_2 = -2,\ x_1 = 2$

Example py-file: Show steps in Gaussian elimination and back substitution without pivoting. myGEshow.py

```
Tolerance value in pivoting is 1.110223e-15.
Begin forward elimination with Augmented system:
[[ 4. -2.  1. 15.]
 [-3. -1.  4.  8.]
 [ 1. -1.  3. 13.]]
Stage 0
After elimination in column 0 with pivot = 4.000000
[[ 4.   -2.    1.   15.  ]
 [ 0.   -2.5   4.75 19.25]
 [ 0.   -0.5   2.75  9.25]]
Stage 1
After elimination in column 1 with pivot = -2.500000
[[ 4.   -2.    1.   15.  ]
 [ 0.   -2.5   4.75 19.25]
 [ 0.    0.    1.8   5.4 ]]
Stage 2
After elimination in column 2 with pivot = 1.800000
[[ 4.   -2.    1.   15.  ]
 [ 0.   -2.5   4.75 19.25]
 [ 0.    0.    1.8   5.4 ]]
x2=3.000000
x1=(19.250000-14.250000)/-2.500000)=-2.000000
x0=(15.000000-7.000000)/4.000000)=2.000000
Solution Vector is  [ 2. -2.  3.]
```

Figure 6.1: Steps in Gaussian elimination and back substitution without pivoting.

- if we had stored the ratio of coefficients in place of zero (we show these in parentheses), our final form would have been

$$\begin{bmatrix} 4 & -2 & 1 & 15 \\ (-0.75) & -2.5 & 4.75 & 19.25 \\ (0.25) & (0.20) & 1.8 & 5.40 \end{bmatrix}$$

- The original matrix can be written as the product:

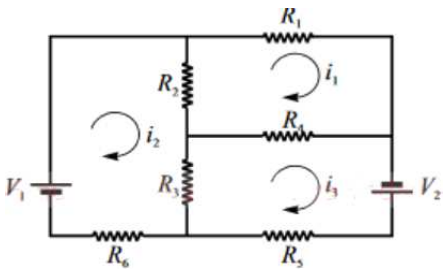
$$A = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ -0.75 & 1 & 0 \\ 0.25 & 0.20 & 1 \end{bmatrix}}_{L} * \underbrace{\begin{bmatrix} 4 & -2 & 1 \\ 0 & -2.5 & 4.75 \\ 0 & 0 & 1.8 \end{bmatrix}}_{U}$$

- This procedure is called a ***LU-decomposition of*** A. $(A = L * U)$
- We have $det(A) = det(U) = (4) * (-2.5) * (1.8) = -18$
- When there are m row interchanges

$$\boxed{det(A) = (-1)^m * u_{11} * \ldots * u_{nn}}$$

Kirchhoff's Rules

Find currents at each loop by using Kirchhoff's Junction & Loop Rules:



3 unkowns, 3 equations

$$\begin{aligned} (R_1 + R_2 + R_4)i_1 - R_2 i_2 - R_4 i_3 &= 0 \\ -R_2 i_1 + (R_2 + R_3 + R_6)i_2 - R_3 i_3 &= V_1 \\ -R_4 i_1 - R_3 i_2 + (R_3 + R_4 + R_5)i_3 &= V_2 \end{aligned}$$

With the values of $R_1 = R_2 = 1\ \Omega$, $R_3 = R_4 = R_5 = R_6 = 2\ \Omega$ and $V_1 = 1\ V$, $V_2 = 5\ V$. System of linear equations becomes as follows:

$$\begin{aligned} 4i_1 - i_2 - 2i_3 &= 0 \\ -i_1 + 5i_2 - 2i_3 &= 1 \\ -2i_1 - 2i_2 + 6i_3 &= 5 \end{aligned}$$

Example py-file: Kirchhoff's Rules in Gaussian elimination & back substitution. No pivoting. myGEshow_Kirchhoff.py

```
Tolerance value in pivoting is 1.110223e-15.
Gaussian Elimination with Augmented system:
[[ 4. -1. -2.  0.]
 [-1.  5. -2.  1.]
 [-2. -2.  6.  5.]]
--- Stage 0
After elimination in column 0 with pivot = 4.000000
[[ 4.   -1.   -2.    0.  ]
 [ 0.    4.75 -2.5   1.  ]
 [ 0.   -2.5   5.    5.  ]]
--- Stage 1
After elimination in column 1 with pivot = 4.750000
[[ 4.         -1.         -2.          0.        ]
 [ 0.          4.75       -2.5         1.        ]
 [ 0.          0.          3.68421053  5.52631579]]
--- Stage 2
After elimination in column 2 with pivot = 3.684211
[[ 4.         -1.         -2.          0.        ]
 [ 0.          4.75       -2.5         1.        ]
 [ 0.          0.          3.68421053  5.52631579]]
Backward substitution
x2=1.500000
x1=(1.000000--3.750000)/4.750000)=1.000000
x0=(0.000000--4.000000)/4.000000)=1.000000
MyGEshow - Solution Vector :  [[1.  1.  1.5]]
NumPy Solve - Solution Vector : [1.  1.  1.5]
```

Figure 6.2: Kirchhoff's Rules in Gaussian elimination & back substitution. No pivoting.

Example. Solve the following system of equations using Gaussian elimination.

$$\begin{array}{rrrr} & 2x_2 & & +x_4 = 0 \\ 2x_1 & +2x_2 & +3x_3 & +2x_4 = -2 \\ 4x_1 & -3x_2 & & x_4 = -7 \\ 6x_1 & +x_2 & -6x_3 & -5x_4 = 6 \end{array}$$

In addition, obtain the determinant of the coefficient matrix and the LU decomposition of this matrix.

1 The augmented coefficient matrix is

$$\begin{bmatrix} 0 & 2 & 0 & 1 & 0 \\ 2 & 2 & 3 & 2 & -2 \\ 4 & -3 & 0 & 1 & -7 \\ 6 & 1 & -6 & -5 & 6 \end{bmatrix}$$

2 We cannot permit a **zero** in the a_{11} position because that element is the pivot in reducing the first column.

3 We could interchange the first row with any of the other rows to avoid a zero divisor, but interchanging the first and fourth rows is our best choice. This gives

$$\begin{bmatrix} 6 & 1 & -6 & -5 & 6 \\ 2 & 2 & 3 & 2 & -2 \\ 4 & -3 & 0 & 1 & -7 \\ 0 & 2 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 6 & 1 & -6 & -5 & 6 \\ 0 & 1.6667 & 5 & 3.6667 & -4 \\ 0 & -3.6667 & 4 & 4.3333 & -11 \\ 0 & 2 & 0 & 1 & 0 \end{bmatrix}$$

4 We again interchange before reducing the second column, not because we have a zero divisor, but because we want to preserve accuracy. Interchanging the second and third rows puts the element of largest magnitude on the diagonal.

$$\begin{bmatrix} 6 & 1 & -6 & -5 & 6 \\ 0 & -3.6667 & 4 & 4.3333 & -11 \\ 0 & 1.6667 & 5 & 3.6667 & -4 \\ 0 & 2 & 0 & 1 & 0 \end{bmatrix}$$

5 Now we reduce in the second column

$$\begin{bmatrix} 6 & 1 & -6 & -5 & 6 \\ 0 & -3.6667 & 4 & 4.3333 & -11 \\ 0 & 0 & 6.8182 & 5.6364 & -9.0001 \\ 0 & 0 & 2.1818 & 3.3636 & -5.9999 \end{bmatrix}$$

6 No interchange is indicated in the third column. Reducing, we get

$$\begin{bmatrix} 6 & 1 & -6 & -5 & 6 \\ 0 & -3.6667 & 4 & 4.3333 & -11 \\ 0 & 0 & 6.8182 & 5.6364 & -9.0001 \\ 0 & 0 & 0 & 1.5600 & -3.1199 \end{bmatrix}$$

7 Back-substitution gives

$$x_1 = -0.50000, x_2 = 1.0000, x_3 = 0.33325, x_4 = -1.9999.$$

- The **correct (exact)** answers are $x_1 = -1/2, x_2 = 1, x_3 = 1/3, x_4 = -2$.
- In this calculation we have carried five significant figures and rounded each calculation.
- Even so, we do not have five-digit accuracy in the answers. The discrepancy is due to **round off**.
- **Example py-file:** Show steps in Gauss elimination and back substitution with pivoting. myGEPivShow.py

```
Tolerance value in pivoting is 1.110223e-15.
Gaussian Elimination with Augmented system:
[[ 0.  2.  0.  1.  0.]
 [ 2.  2.  3.  2. -2.]
 [ 4. -3.  0.  1. -7.]
 [ 6.  1. -6. -5.  6.]]
--- Stage 0
Swap rows 0 and 3;  new pivot = 6.000000
After elimination in column 0 with pivot = 6.000000
[[  6.           1.          -6.          -5.           6.        ]
 [  0.           1.66666667   5.           3.66666667  -4.        ]
 [  0.          -3.66666667   4.           4.33333333 -11.        ]
 [  0.           2.           0.           1.           0.        ]]
--- Stage 1
Swap rows 1 and 2;  new pivot = -3.666667
After elimination in column 1 with pivot = -3.666667
[[  6.           1.          -6.          -5.           6.        ]
 [  0.          -3.66666667   4.           4.33333333 -11.        ]
 [  0.           0.           6.81818182   5.63636364  -9.        ]
 [  0.           0.           2.18181818   3.36363636  -6.        ]]
--- Stage 2
After elimination in column 2 with pivot = 6.818182
[[  6.           1.          -6.          -5.           6.        ]
 [  0.          -3.66666667   4.           4.33333333 -11.        ]
 [  0.           0.           6.81818182   5.63636364  -9.        ]
 [  0.           0.           0.           1.56        -3.12      ]]
--- Stage 3
After elimination in column 3 with pivot = 1.560000
[[  6.           1.          -6.          -5.           6.        ]
 [  0.          -3.66666667   4.           4.33333333 -11.        ]
 [  0.           0.           6.81818182   5.63636364  -9.        ]
 [  0.           0.           0.           1.56        -3.12      ]]
Backward substitution
x3=-2.000000
x2=(-9.000000--11.272727)/6.818182)=0.333333
x1=(-11.000000--7.333333)/-3.666667)=1.000000
x0=(6.000000-9.000000)/6.000000)=-0.500000
MyGEPivShow - Solution Vector :  [[-0.5         1.          0.33333333 -2.        ]]
NumPy Solve - Solution Vector : [-0.5        1.         0.33333333 -2.       ]
```

Figure 6.3: Steps in Gaussian elimination and back substitution with pivoting.

Continue with the previous example.

- If we had replaced the zeros below the main diagonal with the ratio of coefficients at each step, the resulting augmented matrix would be

$$\begin{bmatrix} 6 & 1 & -6 & -5 & 6 \\ (0.66667) & -3.6667 & 4 & 4.3333 & -11 \\ (0.33333) & (-0.45454) & 6.8182 & 5.6364 & -9.0001 \\ (0.0) & (-0.54545) & (0.32) & 1.5600 & -3.1199 \end{bmatrix}$$

- This gives a LU decomposition as

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.66667 & 1 & 0 & 0 \\ 0.33333 & -0.45454 & 1 & 0 \\ 0.0 & -0.54545 & 0.32 & 1 \end{bmatrix} \qquad \begin{bmatrix} 6 & 1 & -6 & -5 \\ 0 & -3.6667 & 4 & 4.3333 \\ 0 & 0 & 6.8182 & 5.6364 \\ 0 & 0 & 0 & 1.5600 \end{bmatrix}$$

- It should be noted that the product of these matrices produces a permutation of the original matrix, call it A', where

$$A' = \begin{bmatrix} 6 & 1 & -6 & -5 \\ 4 & -3 & 0 & 1 \\ 2 & 2 & 3 & 2 \\ 0 & 2 & 0 & 1 \end{bmatrix}$$

- The determinant of the original matrix of coefficients can be easily computed according to the formula

$$det(A) = (-1)^2 * (6) * (-3.6667) * (6.8182) * (1.5600) = -234.0028$$

  which is close to the exact solution: -234.

- The exponent 2 is required, because there were *two row interchanges* in solving this system.

- To summarize

  1. The solution to the four equations
  2. The determinant of the coefficient matrix
  3. A LU decomposition of the matrix, A', which is just the original matrix, A, after we have interchanged its rows.

- **"These" are readily obtained after solving the system by Gaussian elimination method.**

Example py-files: LU factorization without pivoting. myLUshow.py LU factorization with pivoting. myLUPivShow.py

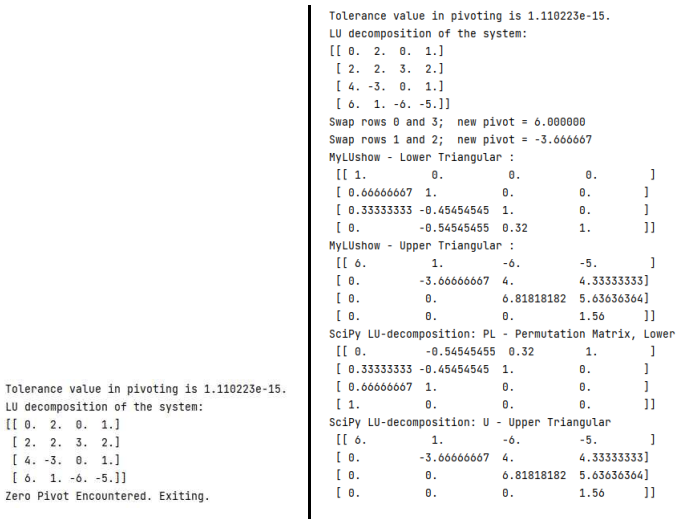


Figure 6.4: (a) Without Pivoting (b) With Pivoting.

6.3.2 Using the LU Matrix for Multiple Right-Hand Sides

- Many physical situations are modelled with a large set of linear equations.
- The equations will depend on the geometry and certain external factors that will determine the right-hand sides.
- For example, in electrical circuit problems, the resistors at the circuit (A matrix) are unchanged with the varying applied voltages (b vector). (e.g., Kirchhoff's Rule)
- If we want the solution for **many different values of these right-hand sides**,
  - it is inefficient to solve the system from the start with each one of the right-hand-side values.
  - Using the LU equivalent of the coefficient matrix is preferred.
- Suppose we have solved the system $Ax = b$ by Gaussian elimination.
- We now know the LU equivalent of A: $A = L * U$
- We can write

$$Ax = b$$

$$LUx = b$$

$$Ly = b$$

- e.g., Solve $Ax = b$, where we already have its L and U matrices:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.66667 & 1 & 0 & 0 \\ 0.33333 & -0.45454 & 1 & 0 \\ 0.0 & -0.54545 & 0.32 & 1 \end{bmatrix} * \begin{bmatrix} 6 & 1 & -6 & -5 \\ 0 & -3.6667 & 4 & 4.3333 \\ 0 & 0 & 6.8182 & 5.6364 \\ 0 & 0 & 0 & 1.5600 \end{bmatrix}$$

- Suppose that the b-vector is $[6 \ -7 \ -2 \ 0]^T$.
- We first get $y(= Ux)$ from $Ly = b$ by forward substitution:

$$y = [6 \ -11 \ -9 \ -3.12]^T$$

- and use it to compute x from $Ux = y$:

$$x = [-0.5 \ 1 \ 0.3333 \ -2]^T.$$

- Exercise: $b = [1 \ 4 \ -3 \ 1]^T \Longrightarrow x = [0.0128 \ -0.5897 \ -2.0684 \ 2.1795]^T$

Phyton Code:

```
import numpy as np
from scipy.linalg import lu
A = np.array([[0.0, 2.0, 0.0, 1.0], [2.0, 2.0, 3.0, 2.0], [4.0, -3.0, 0.0, 1.0], [6.0,
    1.0, -6.0, -5.0]])
P, L, U = lu(A)
print("SciPy LU-decomposition: P - Permutation Matrix \n", P)
print("SciPy LU-decomposition: L - Lower Triangular with unit diagonal elements \n", L)
print("SciPy LU-decomposition: U - Upper Triangular \n", U)
def forward(L, b):
    y=np.zeros(np.shape(b),dtype=float)
    for i in range(len(b)):
        y[i]=np.copy(b[i])
        for j in range(i):
            y[i]=y[i]-(L[i, j]*y[j])
        y[i] = y[i]/L[i, i]
    return y
b = np.array([[6.0], [-7.0], [-2.0], [0.0]])
# b = np.array([[1.0], [4.0], [-3.0], [1.0]])
y=forward(L,b)
print("y vector from Ly=b by forward substitution :", np.transpose(y))
def backward(U, y):
    x=np.zeros(np.shape(y),dtype=float)
    ylen=len(y)-1
    x[ylen] =y[ylen]/U[ylen, ylen] # Print the last stage x value
    for i in range(ylen-1,-1,-1):
        x[i]=np.copy(y[i])
        for j in range(ylen,i,-1):
            x[i]=x[i]-(U[i, j]*x[j])
        x[i] = x[i]/U[i, i]
    return x
x=backward(U,y)
print("x vector from Ux=y by backward substitution :", np.transpose(x))
```

6.4 The Inverse of a Matrix

- Division by a matrix is not defined but the equivalent is obtained from the ***inverse*** of the matrix.
- If the product of two square matrices, $A * B$, equals to the *identity matrix*, I, B is said to be the inverse of A (and also A is the inverse of B).
- By multiplying each element with its cofactor to find the inverse of the matrix is not useful since N^3 multiplication and division are required for an N-dimensional matrix.
- To find the inverse of matrix A, use an elimination method.
- We augment the A matrix with the identity matrix of the same size and solve. **The solution is A^{-1}.** Example;

$$A = \begin{bmatrix} 1 & -1 & 2 \\ 3 & 0 & 1 \\ 1 & 0 & 2 \end{bmatrix} \rightarrow \qquad \left[\begin{array}{ccc|ccc} 1 & -1 & 2 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 2 & 0 & 0 & 1 \end{array}\right] \qquad \left\| \begin{array}{l} R_2 - (3/1)R_1 \rightarrow \\ R_3 - (1/1)R_1 \rightarrow \end{array} \right\|$$

$$\left[\begin{array}{ccc|ccc} 1 & -1 & 2 & 1 & 0 & 0 \\ 0 & 3 & -5 & -3 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 \end{array}\right] \underbrace{\left[\begin{array}{ccc|ccc} 1 & -1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 3 & -5 & -3 & 1 & 0 \end{array}\right]}_{Row\ Interchange} \qquad \left\| \begin{array}{l} \\ R_3 - (3/1)R_2 \rightarrow \end{array} \right\|$$

- Contd.

$$\left[\begin{array}{ccc|ccc} 1 & -1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & -5 & 0 & 1 & -3 \end{array}\right] \left\| \begin{array}{l} \\ R_3/(-5) \rightarrow \end{array} \right\| \qquad \left\| \begin{array}{l} R_1 - (2/1)R_3 \rightarrow \\ \\ \end{array} \right\|$$

$$\left[\begin{array}{ccc|ccc} 1 & -1 & 0 & 1 & 2/5 & -6/5 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & -1/5 & 3/5 \end{array}\right] \qquad \left\| \begin{array}{l} R_2 - (1/-1)R_1 \rightarrow \end{array} \right\|$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2/5 & -1/5 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & -1/5 & 3/5 \end{bmatrix}$$

- We confirm the fact that we have found the inverse by multiplication:

$$\underbrace{\begin{bmatrix} 1 & -1 & 2 \\ 3 & 0 & 1 \\ 1 & 0 & 2 \end{bmatrix}}_{A} * \underbrace{\begin{bmatrix} 0 & 2/5 & -1/5 \\ -1 & 0 & 1 \\ 0 & -1/5 & 3/5 \end{bmatrix}}_{A^{-1}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{I}$$

- It is more *efficient* to use Gaussian elimination. We show only the final triangular matrix; we used pivoting:

$$\begin{bmatrix} 1 & -1 & 2 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 2 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 0 & 1 & 0 & 1 & 0 \\ (0.333) & -1 & 1.667 & 1 & -0.333 & 0 \\ (0.333) & (0) & 1.667 & 0 & -0.333 & 1 \end{bmatrix}$$

- After doing the back-substitutions, we get

$$\begin{bmatrix} 3 & 0 & 1 & 0 & 0.4 & -0.2 \\ (0.333) & -1 & 1.667 & -1 & 0 & 1 \\ (0.333) & (0) & 1.667 & 0 & -0.2 & 0.6 \end{bmatrix}$$

- If we have the inverse of a matrix, we can *use it to solve a set of equations*, $Ax = b$,

- because multiplying by A^{-1} gives the answer (x):

$$A^{-1}Ax = A^{-1}b$$
$$x = A^{-1}b$$

- Phyton Code:

```
import numpy as np
A = np.array([[1.0,-1.0,2.0],[3.0,0.0,1.0],[1.0,0.0,2.0]])
b = np.array([[1.0,0.0,0.0],[0.0,1.0,0.0],[0.0,0.0,1.0]])
x = np.linalg.solve(A, b)
print("NumPy - Inverse Matrix: \n",x)
from scipy import linalg
x=linalg.solve(A,b)
print("SciPy - Inverse Matrix: \n",x)
```

6.5 Eigenvalues and Eigenvectors of a Matrix

- For a square matrix A,

$$A\vec{u} = \lambda\vec{u}$$

  $\vec{u}$ vectors satisfying this condition are called the *eigenvectors* of the matrix A, and the lambda scalar coefficients are called the *eigenvalues*.

- An $N \times N$ matrix has N different eigenvectors. However, the corresponding lambda eigenvalues for these eigenvectors may not be different.

- **State of a system can be expressed in terms of the eigenvectors of the system of linear equations and in terms of their eigenvalues for the measured quantities.**

- Create an U matrix by arranging eigenvectors side by side:

$$U = \overbrace{\begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ u_{21} & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ u_{n1} & u_{n2} & \dots & u_{nn} \end{bmatrix}}^{\vec{u_1} \quad \vec{u_2} \quad ... \quad \vec{u_n}}$$

- When we multiply the matrix A with the matrix U and its inverse matrix U^{-1} from both sides, we get

$$A' = U^{-1}AU = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix}$$

- That is, the similarity transformation with the eigenvectors matrix makes A as being diagonalized and the elements on the diagonal become the eigenvalues of A.

- In principle, the eigenvalue problem is easy to solve. So-called characteristic equation is to be solved:

$$det|A - \lambda I| = 0$$

- After finding the roots of this n-degree polynomial equation, the corresponding eigenvectors can be obtained by solving the following system of equations:

$$(A - \lambda I)\vec{v} = 0$$

- **Since this method requires determinant calculation, it is not useful for large dimensional matrices.**

6.5.1 Normal Modes of Coupled Oscillation

- Consider the coupled oscillations problem of two equal masses m connected by springs of constant k (see Figure).

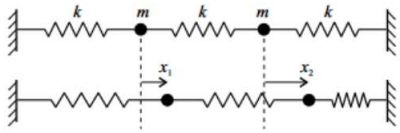


Figure 6.5: Mass-Spring system.

- The differential equation provided by each mass is written using Newton's law of motion as follows

$$\begin{aligned} k(x_2 - x_1) - kx_1 &= m\frac{d^2x_1}{dt^2} \\ -kx_2 - k(x_2 - x_1) &= m\frac{d^2x_2}{dt^2} \end{aligned}$$

- In this system, the frequencies (ω) that both masses oscillate as in common are called **normal oscillation modes**.

- To find the normal mode frequencies, try a solution for both unknowns as:

$$x_1 = x_{10}cos\omega t \quad \& \quad x_2 = x_{20}cos\omega t$$

- Substitute these solutions into the system of linear equations above and simplify by removing $cos\omega t$,

$$\begin{aligned} \frac{2k}{m}x_{10} - \frac{k}{m}x_{20} &= \omega^2 x_{10} \\ -\frac{k}{m}x_{10} + \frac{2k}{m}x_{20} &= \omega^2 x_{20} \end{aligned}$$

- This system of equations can be written as the product of a matrix and a column vector as follows (replace ω^2 by λ):

$$\begin{bmatrix} 2k/m & -k/m \\ -k/m & 2k/m \end{bmatrix} \begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix} = \lambda \begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix}$$

- This structure can also be written as:

$$\begin{bmatrix} 2k/m - \lambda & -k/m \\ -k/m & 2k/m - \lambda \end{bmatrix} \begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix} = 0$$

- The determinant must be zero for this linear system of equations to have a unique solution:

$$det \begin{vmatrix} 2k/m - \lambda & -k/m \\ -k/m & 2k/m - \lambda \end{vmatrix} = 0$$

$$\longrightarrow (2k/m - \lambda)^2 - k^2/m^2 = 0$$

- There are two oscillation frequencies (ω) and their corresponding amplitudes $\vec{x_0} = (x_{10}, x_{20})$:

$$\lambda_1 = k/m \longrightarrow \vec{x_{0,1}} = \begin{pmatrix} 0.71 \\ 0.71 \end{pmatrix}$$

$$\lambda_2 = 3k/m \longrightarrow \vec{x_{0,2}} = \begin{pmatrix} -0.71 \\ 0.71 \end{pmatrix}$$

- The first of these solutions represents the mode in which the two masses oscillate in the **same phase** ($-- > -- >$)
- and the second represents the mode in which they oscillate in the **opposite phase** ($-- >< --$).

Phyton Code:

```
print("****************SymPy Solution for Characteristic Equation: ")
from sympy import Matrix, symbols, pprint, factor
M = Matrix([[2, -1], [-1, 2]])
lamda = symbols('lamda')
poly = M.charpoly(lamda) # Get the characteristic polynomial
print(poly) # Printing polynomial
pprint(factor(poly.as_expr())) # Prints expr in pretty form.
print("****************NumPy Solution for Characteristic Equation: ")
import numpy as np
A = np.array([[2, -1], [-1, 2]])
print(np.poly(A))
print("****************NumPy Solution for Eigenvalues and Eigenvectors: ")
```

```
w,v=np.linalg.eig(A)
print('Eigenvalue:', w)
print('Eigenvector1:', v[0])
print('Eigenvector2:', v[1])
print("**********SciPy Solution for Eigenvalues and Eigenvectors: ")
import scipy.linalg as la
w,v = la.eig(A)
print('Eigenvalue:', w)
print('Eigenvector1:', v[0])
print('Eigenvector2:', v[1])
```

6.6 Iterative Methods

- Gaussian elimination and its variants are called ***direct methods***.
- An entirely different way to solve many systems is through ***iteration***.
- In this way, we start with an initial estimate of the solution vector and proceed to refine this estimate.
- An $n \times n$ matrix A is *diagonally dominant* if and only if;

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|, \quad i = 1, 2, \ldots, n$$

- Example. Given matrix & After reordering;

$$\begin{aligned} 6x_1 - 2x_2 + x_3 &= 11 \\ x_1 + 2x_2 - 5x_3 &= -1 \\ -2x_1 + 7x_2 + 2x_3 &= 5 \end{aligned} \quad \& \quad \begin{aligned} 6x_1 - 2x_2 + x_3 &= 11 \\ -2x_1 + 7x_2 + 2x_3 &= 5 \\ x_1 + 2x_2 - 5x_3 &= -1 \end{aligned}$$

- The solution is $x_1 = 2, x_2 = 1, x_3 = 1$ (for both cases?).
- Before we begin our iterative scheme we must first reorder the equations so that the coefficient matrix is diagonally dominant.

6.6.1 Jacobi Method

- The iterative methods depend on the rearrangement of the equations in this manner:

$$\boxed{x_i = \frac{b_i}{a_{ii}} - \sum_{j=1, j\neq i}^{n} \frac{a_{ij}}{a_{ii}} x_j}, i = 1, 2, \ldots, n, \mapsto x_1 = \frac{11}{6} - \left(\frac{-2}{6} x_2 + \frac{1}{6} x_3 \right) \tag{6.4}$$

	First	Second	Third	Fourth	Fifth	Sixth	...	Ninth
x_1	0	1.833	2.038	2.085	2.004	1.994	...	2.000
x_2	0	0.714	1.181	1.053	1.001	0.990	...	1.000
x_3	0	0.200	0.852	1.080	1.038	1.001	...	1.000

Table 6.1: Successive estimates of solution (Jacobi method)

- Each equation now solved for the variables in succession:

$$\begin{aligned} x_1 &= 1.8333 + 0.3333x_2 - 0.1667x_3 \\ x_2 &= 0.7143 + 0.2857x_1 - 0.2857x_3 \\ x_3 &= 0.2000 + 0.2000x_1 + 0.4000x_2 \end{aligned}$$

- We begin with some initial approximation to the value of the variables.
- Say initial values are; $x_1 = 0, x_2 = 0, x_3 = 0$. Each component might be taken equal to *zero if no better initial estimates* are at hand.
- The new values are substituted in the right-hand sides to generate a second approximation,
- and the process is repeated until successive values of each of the variables are sufficiently alike.
- Now, general form

$$\begin{aligned} x_1^{(n+1)} &= 1.8333 + 0.3333x_2^{(n)} - 0.1667x_3^{(n)} \\ x_2^{(n+1)} &= 0.7143 + 0.2857x_1^{(n)} - 0.2857x_3^{(n)} \\ x_3^{(n+1)} &= 0.2000 + 0.2000x_1^{(n)} + 0.4000x_2^{(n)} \end{aligned} \tag{6.5}$$

- Starting with an initial vector of $x^{(0)} = (0, 0, 0,)$, we obtain Table 6.1
- Rewrite in matrix notation; let $A = L + D + U$,

$$Ax = b \Longrightarrow \begin{bmatrix} 6 & -2 & 1 \\ -2 & 7 & 2 \\ 1 & 2 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 11 \\ 5 \\ -1 \end{bmatrix}$$

$$L = \begin{bmatrix} 0 & 0 & 0 \\ -2 & 0 & 0 \\ 1 & 2 & 0 \end{bmatrix}, D = \begin{bmatrix} 6 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & -5 \end{bmatrix}, U = \begin{bmatrix} 0 & -2 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{aligned} Ax &= (L + D + U)x = b \\ Dx &= -(L + U)x + b \\ x &= -D^{-1}(L + U)x + D^{-1}b \end{aligned}$$

- From this we have, identifying x on the left as the new iterate,

$$x^{(n+1)} = -D^{-1}(L+U)x^{(n)} + D^{-1}b$$

- In Eqn. 6.5,

$$b' = D^{-1}b = \begin{bmatrix} 1.8333 \\ 0.7143 \\ 0.2000 \end{bmatrix}$$

$$D^{-1}(L+U) = \begin{bmatrix} 0 & -0.3333 & 0.1667 \\ -0.2857 & 0 & 0.2857 \\ -0.2000 & -0.4000 & 0 \end{bmatrix}$$

- This procedure is known as the Jacobi method, also called "*the method of simultaneous displacements*",

- because each of the equations is simultaneously changed by using the most recent set of x-values (see Table 6.1).

- **Example py-file:** The Jacobi approximation to the solution of AX = B. myJacobi.py

Chapter 7

Interpolation and Curve Fitting

7.1 Interpolation and Curve Fitting

- Sines, logarithms, and other nonalgebraic functions *from tables*.
- Those tables had values of the function at *uniformly spaced values* of the argument.
- Most often *interpolated* linearly:
  The value for $x = 0.125$ was computed as at the halfway point between $x = 0.12$ and $x = 0.13$.
- If the function does not vary too rapidly and the tabulated points are close enough together, this linearly estimated value would be accurate enough.
- As a conclusion: **Data can be interpolated to estimate values**.
- **Interpolating Polynomials:** Describes a straightforward but computationally inconvenient way to fit a polynomial to a set of data points **so that an interpolated value can be computed**.
- **Divided Differences:** These provide a more efficient way to construct an interpolating polynomial, one that allows one to readily change the degree of the polynomial. If the data are at evenly spaced x-values, there is some simplification.
- **Spline Curves:** Using special polynomials, *splines*, one can fit polynomials to data more accurately than with an interpolating polynomial. At the expense of added computational effort, some important problems that one has with interpolating polynomials is overcome.
- **Least-Squares Approximations:** are methods by which polynomials and other functions can be fitted to data that are subject to errors likely in experiments. These approximations are widely used **to analyze experimental observations**

7.1.1 Interpolating Polynomials

- We have a table of x and y-values.
- Two entries in this table might be
  $y = 2.36$ at $x = 0.41$ and
  $y = 3.11$ at $x = 0.52$.

- If we desire an estimate for y at $x = 0.43$, we would use the two table values for that estimate.

- Why not interpolate as if $y(x)$ was linear between the two x-values (similar triangles)?

$$y(0.43) \approx 2.36 + \frac{2}{11}(3.11 - 2.36) = 2.50 \quad \Big| \quad \text{where} \quad \frac{2}{11} \Longrightarrow \frac{0.43 - 0.41}{0.52 - 0.41}$$

- We will be most interested in techniques adapted to situations where the data are far from linear.

- The basic principle is to fit a polynomial curve to the data.

Interpolation versus Curve Fitting

- Given a set of data
  $y_i = f(x_i) \qquad i = 1, \ldots, n$
  obtained from an experiment or from some calculation.

- **In curve fitting,** the approximating function **passes near the data points**, but (usually) not exactly through them. There is some uncertainty in the data.

- **In interpolation**, process inherently assumes that the data have no uncertainty. The interpolation function **passes *exactly* through** each of the known data points.

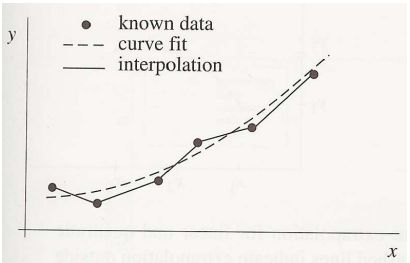


Figure shows a plot of some hypothetical experimental data, a curve fit function and interpolating with piecewise-linear function.

Fitting a Polynomial to Data

- Interpolation involves constructing and then evaluating an interpolating function.
- **interpolant**, $y = F(x)$, determined by requiring that **it pass through the known data** (x_i, y_i).
- In its most general form, interpolation involves **determining the coefficient**s $a_1, a_2, \ldots, a_n$
- in the linear combination of ***n basis functions***, $\Phi(x)$, that constitute the interpolant

$$F(x) = a_1\Phi_1(x) + a_2\Phi_2(x) + \ldots + a_n\Phi_n(x)$$

  - such that $F(x) = y_i$ for $i = 1, \ldots, n$. The **basis function** may be **polynomial**

$$F(x) = a_1 + a_2x + a_3x^2 + \ldots + a_nx^{n-1}$$

  - or **trigonometric**

$$F(x) = a_1 + a_2e^{ix} + a_3e^{i2x} + \ldots + a_ne^{i(n-1)x}$$

  - or some other **suitable set of functions**.

Polynomials are often used for interpolation because they are easy to evaluate and easy to manipulate analytically.

Table 7.1: Fitting a polynomial to data.

x	f(x)
3.2	22.0
2.7	17.8
1.0	14.2
4.8	38.3
5.6	51.7

- Suppose that we have a data set.
- First, we need to select the points that determine our polynomial.
- The maximum degree of the polynomial is always one less than the number of points.
- Suppose we choose the first four points. If the cubic is $\boxed{ax^3 + bx^2 + cx + d}$,

- We can write four equations involving the unknown coefficients a, b, c, and d;

$$\begin{aligned} when\ x = 3.2 &\Rightarrow a(3.2)^3 + b(3.2)^2 + c(3.2) + d = 22.0 \\ when\ x = 2.7 &\Rightarrow a(2.7)^3 + b(2.7)^2 + c(2.7) + d = 17.8 \\ when\ x = 1.0 &\Rightarrow a(1.0)^3 + b(1.0)^2 + c(1.0) + d = 14.2 \\ when\ x = 4.8 &\Rightarrow a(4.8)^3 + b(4.8)^2 + c(4.8) + d = 38.3 \end{aligned}$$

- Solving these equations gives

$$\begin{aligned} a = &\quad -0.5275 \\ b = &\quad 6.4952 \\ c = &\quad -16.1177 \\ d = &\quad 24.3499 \end{aligned}$$

- and our polynomial is

$$\boxed{-0.5275x^3 + 6.4952x^2 - 16.1177x + 24.3499}$$

- At $x = 3.0$, the **estimated value** is 20.212.

- if we want a new polynomial that is also made to fit at the point $(5.6, 51.7)$?

- or if we want to see what difference it would make to use a quadratic instead of a cubic?

- **Example py-file:** Polynomial Interpolation. Gaussian elimination & back substitution. No pivoting. myGEshow_interpolation.py

```
MyGEshow - Solution Vector :  [[ -0.52748013   6.49522788 -16.11768944  24.3499417 ]]
          3         2
-0.5275 x + 6.495 x - 16.12 x + 24.35
MyGEshow - Estimated y-value for x=3.000000 : 20.211961
NumPy Solve - Solution Vector : [[ -0.52748013   6.49522788 -16.11768944  24.3499417 ]]
          3         2
-0.5275 x + 6.495 x - 16.12 x + 24.35
NumPy Solve - Estimated y-value for x=3.000000 : 20.211961
```

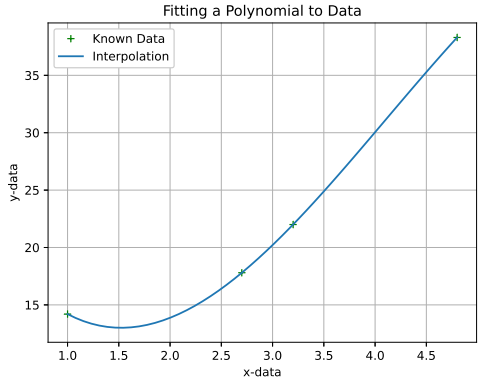


Figure 7.1: Polynomial Interpolation.

Table 7.2: Interpolation of gasoline prices.

year	price
1986	133.5
1988	132.2
1990	138.7
1992	141.5
1994	137.6
1996	144.2

- Another example;
- Use the polynomial order 5, why?

$$P = a_1 + a_2 y + a_3 y^2 + a_4 y^3 + a_5 y^4 + a_6 y^5$$

- Make a guess about the prices of gasoline at year of 1991 (2011).

- **Example py-file:** Interpolation of gasoline prices. Gaussian elimination & back substitution. No pivoting. myGEshow_gasoline.py
  - Now, try with the shifted dates (Xdata=Xdata-np.mean(Xdata)).
  - Make the necessary corrections for the array A.
  - What differs in the plot and why?

```
MyGEshow - Solution Vector :  [[-2.39583357e-01  1.42985014e+03 -2.84447812e+06  1.88622242e+09]]
          3         2
-0.2396 x + 1430 x - 2.844e+06 x + 1.886e+09
MyGEshow - Estimated y-value for x=1991.000000 : 141.281250
NumPy Solve - Solution Vector : [[-2.39583352e-01  1.42985011e+03 -2.84447807e+06  1.88622238e+09]]
          3         2
-0.2396 x + 1430 x - 2.844e+06 x + 1.886e+09
NumPy Solve - Estimated y-value for x=1991.000000 : 141.281251
```

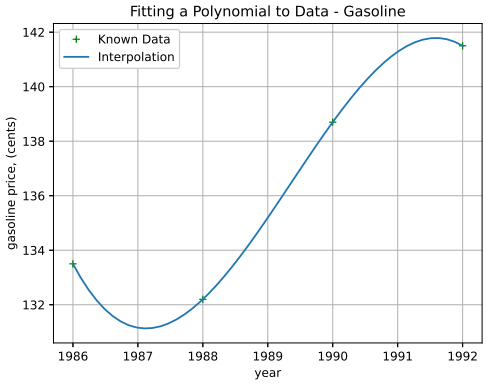


Figure 7.2: Polynomial Interpolation - Gasoline Case.

Lagrangian Polynomials

- *Straightforward approach*-the Lagrangian polynomial.
- The simplest way to exhibit the existence of a polynomial for interpolation with *unevenly* spaced data.
  - **Linear interpolation**
  - **Quadratic interpolation**
  - **Cubic interpolation**
- Lagrange polynomials have two important advantages over interpolating polynomials.
  1. the construction of the interpolating polynomials does not require the solution of a system of equations (such as Gaussian elimination).
  2. the evaluation of the Lagrange polynomials is much less susceptible to roundoff.
- **Linear interpolation**

$$P_1(x) = c_1 x + c_2$$

- put the values

$$\begin{array}{rl} y_1 = & c_1 x_1 + c_2 \\ \hline y_2 = & c_1 x_2 + c_2 \end{array}$$

- then

$$c_1 = \frac{y_2 - y_1}{x_2 - x_1} \qquad\qquad c_2 = \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1}$$

- substituting back and rearranging

$$P_1(x) = y_1 \frac{x - x_2}{x_1 - x_2} + y_2 \frac{x - x_1}{x_2 - x_1}$$

- redefining as

$$\boxed{P_1(x) = y_1 L_1(x) + y_2 L_2(x)}$$

- where Ls are the first-degree ***Lagrange interpolating polynomials***.

- **Quadratic interpolation**

$$\boxed{P_2(x) = y_1 L_1(x) + y_2 L_2(x) + y_3 L_3(x)}$$

where L_s are not the same with the previous L_s!

$$L_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)}, \qquad L_2(x) = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)},$$

$$L_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}.$$

- **Cubic interpolation**

- Suppose we have a table of data with four pairs of x- and $f(x)$-values, with x_i indexed by variable i:

i	x	$f(x)$
0	x_0	f_0
1	x_1	f_1
2	x_2	f_2
3	x_3	f_3

Through these four data pairs we can pass a cubic.

- The Lagrangian form is

$$P_3(x) = \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)}f_0 + \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)}f_1$$
$$+\frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)}f_2 + \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)}f_3$$

- This equation is made up of four terms, each of which is a cubic in x; hence the sum is a cubic.
- The pattern of each term is to form the numerator as a product of linear factors of the form $(x - x_i)$, omitting one x_i in each term.
  - The omitted value being used to form the denominator by replacing x in each of the numerator factors.
  - In each term, we multiply by the f_i.
  - It will have $n + 1$ terms when the degree is n.
- Fit a cubic through the first four points of the preceding Table 7.1 (first four points) and use it to find the interpolated value for $x = 3.0$.

Table 7.3: Fitting a polynomial to data.

x	f(x)
3.2	22.0
2.7	17.8
1.0	14.2
4.8	38.3
5.6	51.7

$$P_3(x) = \frac{(x-2.7)(x-1.0)(x-4.8)}{(3.2-2.7)(3.2-1.0)(3.2-4.8)}22.0+$$
$$\frac{(x-3.2)(x-1.0)(x-4.8)}{(2.7-3.2)(2.7-1.0)(2.7-4.8)}17.8+$$
$$\frac{(x-3.2)(x-2.7)(x-4.8)}{(1.0-3.2)(1.0-2.7)(1.0-4.8)}14.2+$$
$$\frac{(x-3.2)(x-2.7)(x-1.0)}{(4.8-3.2)(4.8-2.7)(4.8-1.0)}38.3$$

- Carrying out the arithmetic, $P_3(3.0) = 20.21$.
- In general

$$P_{n-1}(x) = y_1L_1(x) + y_2L_2(x) + \ldots + y_nL_n(x) = \sum_{j=1}^{n} y_jL_j(x)$$

$$\textit{where} \quad \boxed{L_j(x) = \prod_{k=1,k\neq j}^{n} \frac{x-x_k}{x_j-x_k}}$$

Example py-file: Interpolation of gasoline prices. Lagrange Interpolation. myLagInt_gasoline.py

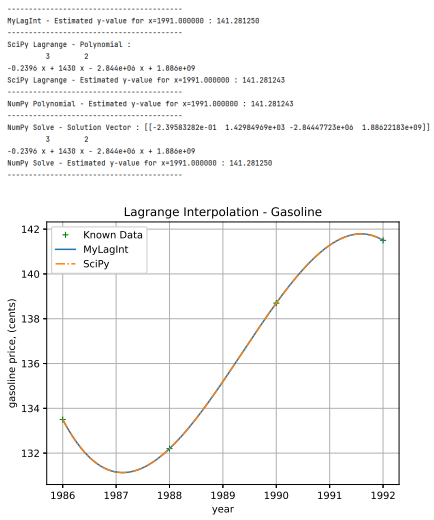


Figure 7.3: Lagrange Polynomial Interpolation - Gasoline Case.

- **Error of Interpolation**; When we fit a polynomial $P_n(x)$ to some data points, it will pass exactly through those points,
  - but between those points $P_n(x)$ will not be precisely the same as the function $f(x)$ that generated the points (unless the function is that polynomial).
  - How much is $P_n(x)$ different from $f(x)$?
  - How large is the error of $P_n(x)$?
- It is most important that you never fit a polynomial of a degree higher than 4 or 5 to a set of points.
- If you need to fit to a set of more than six points, be sure to break up the set into subsets and fit separate polynomials to these.
- You cannot fit a function that is discontinuous or one whose derivative is discontinuous with a polynomial.

Neville's Method

- The trouble with the standard *Lagrangian polynomial technique* is that we **do not know which degree** of polynomial to use.

  - If the degree is **too low**, the interpolating polynomial does **not give good estimates** of $f(x)$.
  - If the degree is **too high**, **undesirable oscillations** in polynomial values can occur.

- **Neville's method** can overcome this difficulty.
  - It computes the interpolated value with polynomials of successively higher degree,
  - *stopping when the successive values are close together.*

- The successive approximations are actually computed by linear interpolation from the previous values.

- The Lagrange formula for linear interpolation to get $f(x)$ from two data pairs, (x_1, f_1) and (x_2, f_2), is

$$f(x) = \frac{(x - x_2)}{(x_1 - x_2)} f_1 + \frac{(x - x_1)}{(x_2 - x_1)} f_2$$

- Neville's method begins by arranging the given data pairs, (x_i, f_i).

- Such that the successive values are in order of the closeness of the x_i to x.

- Suppose we are given these data

x	f(x)
10.1	0.17537
22.2	0.37784
32.0	0.52992
41.6	0.66393
50.5	0.63608

and we want to interpolate for $x = 27.5$.

We first *rearrange* the data pairs in order of closeness to $x = 27.5$:

i	—x-x_i—	x_i	f_i=P_{i0}
0	4.5	32.0	0.52992
1	5.3	22.2	0.37784
2	14.1	41.6	0.66393
3	17.4	10.1	0.17537
4	23.0	50.5	0.63608

- Neville's method begins by renaming the f_i as P_{i0}.

- We build a table

i	x	P_{i0}	P_{i1}	P_{i2}	P_{i3}	P_{i4}
0	32.0	**0.52992**	**0.46009**	**0.46200**	**0.46174**	**0.45754**
1	22.2	0.37784	0.45600	0.46071	0.47901	
2	41.6	0.66393	0.44524	0.55843		
3	10.1	0.17537	0.37379			
4	50.5	0.63608				

- Thus, the value of P_{01} is computed by

$$f(x) = \frac{(27.5 - x_1)}{(x_0 - x_1)} * 0.52992 + \frac{(27.5 - x_0)}{(x_1 - x_0)} * 0.37784$$

substituting all;

$$P_{01} = \frac{(27.5 - 22.2)}{(32.0 - 22.2)} * 0.52992 + \frac{(27.5 - 32.0)}{(22.2 - 32.0)} * 0.37784 = 0.46009$$

- Once we have the column of P_{i1}'s, we compute the next column.

$$P_{22} = \frac{(27.5 - 41.6) * 0.37379 + (50.5 - 27.5) * 0.44524}{50.5 - 41.6} = 0.55843$$

- The remaining columns are computed similarly.

- The general formula for computing entries into the table is

$$\boxed{p_{i,j} = \frac{(x - x_i) * P_{i+1,j-1} + (x_{i+j} - x) * P_{i,j-1}}{x_{i+j} - x_i}}$$

- The **top line of the table** represents Lagrangian interpolates at $x = 27.5$ using polynomials of *degree equal to the second subscript* of the $P's$.

i	x	P_{i0}	P_{i1}	P_{i2}	P_{i3}	P_{i4}
		0.52992	**0.46009**	**0.46200**	**0.46174**	**0.45754**

- The preceding data are for sines of angles in degrees and the correct value for $x = 27.5$ is 0.46175.

7.2 Divided Differences

- There are two disadvantages to using the Lagrangian polynomial or Neville's method for interpolation.

  1. It involves more arithmetic operations than does the divided-difference method.
  2. More importantly, if we desire to add or subtract a point from the set used to construct the polynomial, we essentially have to start over in the computations.

- Both the Lagrangian polynomials and Neville's method also must repeat all of the arithmetic if we must interpolate at a new x-value.

- The divided-difference method avoids all of this computation.

- Actually, we will not get a polynomial different from that obtained by Lagrange's technique.

- Every n^{th}-degree polynomial that **passes through the same** $n+1$ **points** is identical.

- Only the way that the polynomial is expressed is different.

The function, $f(x)$, is known at several values for x:

x_0	f_0
x_1	f_1
x_2	f_2
x_3	f_3

- We do not assume that the x's are evenly spaced or even that the values are arranged in any particular order.

- Consider the n^{th}-degree polynomial written as:

$$P_n(x) = a_0 + (x - x_0)a_1 + (x - x_0)(x - x_1)a_2 + (x - x_0)(x - x_1)\dots(x - x_{n-1})a_n$$

- If we chose the a_i's so that $P_n(x) = f(x)$ at the $n+1$ known points, then $P_n(x)$ is an interpolating polynomial.

- The a_i's are readily determined by using what are called the **divided differences of the tabul**

- A special standard notation for divided differences is

$$\boxed{f[x_0, x_1] = \frac{f_1 - f_0}{x_1 - x_0}}$$

called the *first divided difference* between x_0 and x_1.

- And, $f[x_0] = f_0 = f(x_0)$ (zero-order difference).

$$f[x_s] = f_s$$

- Second- and higher-order differences are defined in terms of lower-order differences.

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

- For n-terms,

$$\boxed{f[x_0, x_1, \ldots, x_n] = \frac{f[x_1, x_2, \ldots, f_n] - f[x_0, x_1, \ldots, f_{n-1}]}{x_n - x_0}}$$

Using the standard notation, a divided-difference table is shown in symbolic form in Table 7.4.

x_i	f_i	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]$
x_0	f_0	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	$f[x_0, x_1, x_2, x_3]$
x_1	f_1	$f[x_1, x_2]$	$f[x_1, x_2, x_3]$	$f[x_1, x_2, x_3, x_4]$
x_2	f_2	$f[x_2, x_3]$	$f[x_2, x_3, x_4]$	
x_3	f_3	$f[x_3, x_4]$		

Table 7.4: Divided-difference table in symbolic form.

- Table 7.10 shows specific numerical values.

$$f[x_0, x_1] = \frac{f_1 - f_0}{x_1 - x_0} = \frac{17.8 - 22.0}{2.7 - 3.2} = 8.4$$

$$f[x_1, x_2] = \frac{f_2 - f_1}{x_2 - x_1} = \frac{14.2 - 17.8}{1.0 - 2.7} = 2.1176$$

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = \frac{2.1176 - 8.4}{1.0 - 3.2} = 2.8556$$

and the others..

x_i	f_i	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, \ldots, x_{i+3}]$	$f[x_i, \ldots, x_{i+4}]$
3.2	22.0	8.400	2.856	-0.528	0.256
2.7	17.8	2.118	2.012	0.0865	
1.0	14.2	6.342	2.263		
4.8	38.3	16.750			
5.6	51.7				

Table 7.5: Divided-difference table in numerical values.

$$\begin{array}{ll} x = x_0: & P_0(x_0) = a_0 \\ x = x_1: & P_1(x_1) = a_0 + (x_1 - x_0)a_1 \\ x = x_2: & P_2(x_2) = a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2 \\ \vdots & \vdots \\ x = x_n: & P_n(x_n) = a_0 + (x_n - x_0)a_1 + (x_n - x_0)(x_n - x_1)a_2 + \ldots \\ & +(x_n - x_0)\ldots(x_n - x_{n-1})a_n \end{array}$$

- If $P_n(x)$ is to be an interpolating polynomial, it must match the table for all $n + 1$ entries:

$$P_n(x_i) = f_i \; for \; i = 0, 1, 2, \ldots, n.$$

- Each $P_n(x_i)$ will equal f_i, if $a_i = f[x_0, x_1, \ldots, x_i]$. We then can write:

$$P_n(x) = f[x_0] + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2]$$
$$+(x - x_0)(x - x_1)(x - x_2)f[x_0, \ldots, x_3]$$
$$+(x - x_0)(x - x_1)\ldots(x - x_{n-1})f[x_0, \ldots, x_n]$$

- Write interpolating polynomial of degree-3 that fits the data of Table 7.10 at all points $x_0 = 3.2$ to $x_3 = 4.8$.

$$P_3(x) = 22.0 + 8.400(x - 3.2) + 2.856(x - 3.2)(x - 2.7)$$
$$-0.528(x - 3.2)(x - 2.7)(x - 1.0)$$

- What is the fourth-degree polynomial that fits at all five points?
- **We only have to add one more term to** $P_3(x)$

$$P_4(x) = P_3(x) + 0.2568(x - 3.2)(x - 2.7)(x - 1.0)(x - 4.8)$$

- If we compute the interpolated value at $x = 3.0$, we get the same result: $P_3(3.0) = 20.2120$.

- **This is not surprising, because all third-degree polynomials that pass through the same four points are identical.**
- **They may look different but they can all be reduced to the same form.**

Example py-file: Constructs a table of divided-difference coefficients. Diagonal entries are coefficients of the polynomial. mydivDiffTable_interpolation.py

```
Constructed MydivDiffTable:
[[22.         0.         0.         0.        ]
 [17.8        8.4        0.         0.        ]
 [14.2        2.11764706 2.85561497 0.        ]
 [38.3        6.34210526 2.01164676 -0.52748013]]
Diagonal entries are the coefficients of the polynomial:
[22.         8.4        2.85561497 -0.52748013]
MydivDiffTable - Estimated y-value for x=3.000000 : 20.211961
```

- **Divided differences for a polynomial**
- It is of interest to look at the divided differences for $f(x) = P_n(x)$.
- Suppose that $f(x)$ is the cubic

$$f(x) = 2x^3 - x^2 + x - 1.$$

- Here is its divided-difference table:
- Observe that the third divided differences are all the same.
- It then follows that all higher divided differences will be zero.

$$P_3(x) = f[x_0] + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2] + (x - x_0)(x - x_1)(x - x_2)f[x_0, x_1, x_2, x_3]$$

```
import numpy as np
D=np.array([[ -0.736],  [2.480], [3.000], [2.000]])
print(np.transpose(D))
# [[-0.736   2.48    3.      2.    ]]
import sympy as sym
x = sym.Symbol('x')
P3=D[0]+(x-0.3)*D[1]+(x-0.3)*(x-1)*D[2]+(x-0.3)*(x-1)*(x-0.7)*D[3]
print(P3)
# [2.48*x + 2.0*(x - 1)*(x - 0.7)*(x - 0.3) + 3.0*(x - 1)*(x - 0.3) - 1.48]
print(sym.expand(2.48*x + 2.0*(x - 1)*(x - 0.7)*(x - 0.3) + 3.0*(x - 1)*(x -
    0.3) - 1.48))
# 2.0*x**3 - 1.0*x**2 + 1.0*x - 1.0
```

which is same with the starting polynomial.

x_i	$f[x_i]$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, \ldots, x_{i+3}]$	$f[x_i, \ldots, x_{i+4}]$	$f[x_i, \ldots, x_{i+5}]$
0.30	-0.736	2.480	3.000	2.000	0.000	0.000
1.00	1.000	3.680	3.600	2.000	0.000	
0.70	-0.104	2.240	5.400	2.000		
0.60	-0.328	8.720	8.200			
1.90	11.008	21.020				
2.10	15.212					

Table 7.6: Divided-difference table in numerical values for a polynomial.

7.3 Spline Curves

- There are times when fitting an interpolating polynomial to data points is very difficult.
- Figure 7.4a is plot of $f(x) = cos^{10}(x)$ on the interval $[-2, 2]$.
- It is a nice, smooth curve but has a pronounced maximum at $x = 0$ and is near to the x-axis for $|x| > 1$.
- The curves of Figure 7.4b,c, d, and e are for polynomials of degrees $-2, -4, -6$, and -8 that match the function at evenly spaced points.
- **None of the polynomials is a good representation of the function**.

- One might think that a solution to the problem would be to break up the interval $[-2, 2]$ into subintervals
- and **fit separate polynomials** to the function in these smaller intervals.
- Figure 7.5 shows a much better fit if we use a quadratic between $x = -0.65$ and $x = 0.65$ and with $P(x) = 0$ outside that interval.
- That is better but there are discontinuities in the slope where the separate polynomials join.

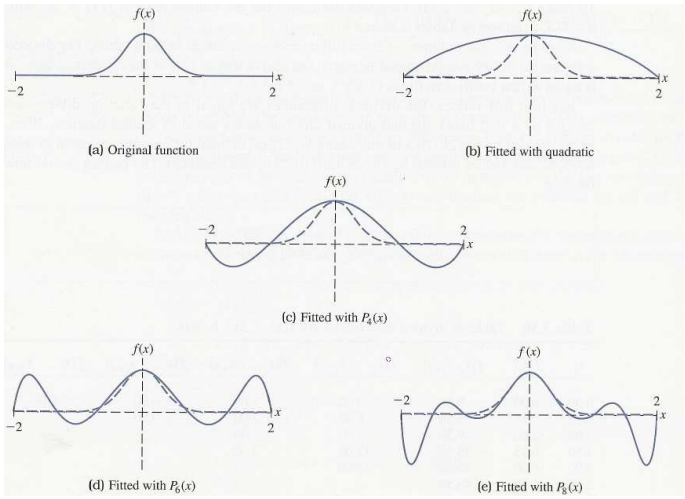


Figure 7.4: Fitting with different degrees of the polynomial.

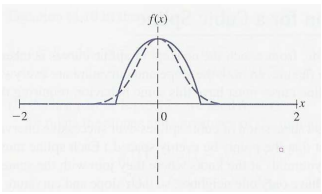


Figure 7.5: Fitting with quadratic in subinterval.

- **This solution is known as spline curves.**
- Suppose that we have a set of $n+1$ points (which do not have to be evenly spaced):
$$(x_i, y_i), \ with \ i = 0, 1, 2, \ldots, n.$$
- A spline fits a set of n^{th}-degree polynomials, $g_i(x)$, between each pair of points, from x_i to x_{i+1}.
- The points at which the splines join are called knots.

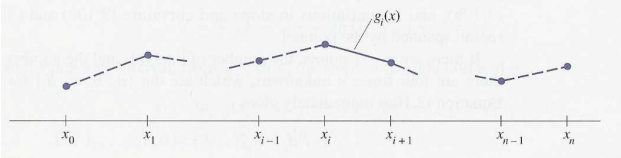


Figure 7.6: Linear spline.

- If the polynomials are all of degree 1, we have a *linear spline* and the curve would appear as in the Fig. 7.6.
- **The slopes are discontinuous where the segments join.**

7.3.1 The Equation for a Cubic Spline

- We will create a succession of cubic splines over successive intervals of the data (See Fig. 7.7).
- Each spline must join with its neighbouring cubic polynomials at the knots where they join with the **same slope and curvature**.
- **We write the equation for a cubic polynomial, $g_i(x)$, in the i^{th} interval, between points** $(x_i, y_i), (x_{i+1}, y_{i+1})$ *(solid line)*.

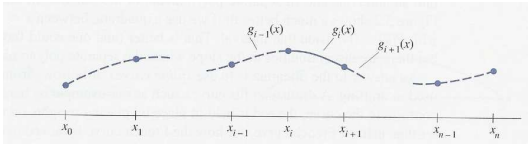


Figure 7.7: Cubic spline.

- It has this equation:

$$g_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$$

- The *dashed curves* are other cubic spline polynomials.
- Thus, the cubic spline function is of the form

$$g(x) = g_i(x) \; on \; the \; interval [x_i, x_{i+1}], \; for \; i = 0, 1, \ldots, n-1$$

- and meets these conditions:

$$g_i(x_i) = y_i, \; i = 0, 1, \ldots, n-1 \; and \; g_{n-1}(x_n) = y_n \tag{7.1}$$

$$g_i(x_{i+1}) = g_{i+1}(x_{i+1}), \; i = 0, 1, \ldots, n-2 \tag{7.2}$$

$$g_i^{'}(x_{i+1}) = g_{i+1}^{'}(x_{i+1}), \; i = 0, 1, \ldots, n-2 \tag{7.3}$$

$$g_i^{''}(x_{i+1}) = g_{i+1}^{''}(x_{i+1}), \; i = 0, 1, \ldots, n-2 \tag{7.4}$$

- Equations say that the cubic spline fits to each of the points Eq. 7.1, is continuous Eq. 7.2, and is continuous in slope and curvature Eq. 7.3 and Eq. 7.4, throughout the region spanned by the points.

7.4 Least-Squares Approximations

- Until now, we have assumed that the data are accurate,
- but when these values are derived **from an experiment**, there is **some error in the measurements**.

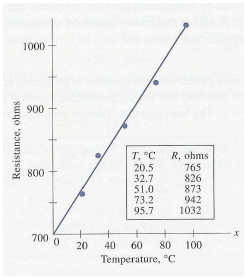


Figure 7.8: Resistance vs Temperature graph for the Least-Squares Approximation.

- Some students are assigned to find the effect of temperature on the resistance of a metal wire.
- They have recorded the temperature and resistance values in a table and have plotted their findings, as seen in Fig. 7.8.
- **The graph suggest a linear relationship**.

$$R = aT + b$$

- Values for the parameters, a & b, can be obtained from the plot.
- If someone else were given same data and asked to draw the line,
- it is not likely that they would draw exactly the same line and they would get different values for a & b.
- A way of fitting a line to experimental data that is to **minimize the deviations** of the points from the line.
- The usual method for doing this is called the **least-squares method**.
- The deviations are determined by the **distances between the points and the line**.

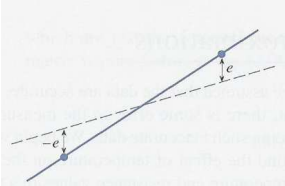


Figure 7.9: Minimizing the deviations by making the sum a minimum.

  - Consider the case of only two points (See Fig. 7.9).
  - Obviously, the best line passes through each point,
  - but any line that passes through the midpoint of the segment connecting them has a *sum of errors equal to zero*.

- We might first suppose we could minimize the deviations by making their sum a minimum, but this is **not an adequate criterion**.
- We might accept the criterion that we make the magnitude of the maximum error a minimum (the so-called *minimax* criterion).
- The usual criterion is to minimize the sum of the *squares* of the errors, the *least-squares* principle.
- Let Y_i represent an experimental value, and let y_i be a value from the equation

$$y_i = ax_i + b$$

  where x_i is a particular value of the variable assumed to be free of error.
- We wish to determine the best values for a & b so that the y's predict the function values that correspond to x-values.
- Let errors defined by $e_i = Y_i - y_i = Y_i - (ax_i + b)$

- The least-squares criterion requires that S be a minimum.

$$\begin{aligned} S &= e_1^2 + e_2^2 + \ldots + e_n^2 = \textstyle\sum_{i=1}^{N} e_i^2 \\ &= \textstyle\sum_{i=1}^{N} (Y_i - ax_i - b)^2 \end{aligned}$$

  N is the number of (x, Y)-pairs.

- We reach the minimum by proper choice of the parameters a & b, so they are the *variables* of the problem.

- At a minimum for S, the two partial derivatives will be zero.

$$\partial S/\partial a \qquad \& \qquad \partial S/\partial b$$

- Remembering that the x_i and Y_i are data points unaffected by our choice our values for a and b, we have

$$\begin{aligned} \tfrac{\partial S}{\partial a} = 0 &= \textstyle\sum_{i=1}^{N} 2(Y_i - ax_i - b)(-x_i) \\ \tfrac{\partial S}{\partial b} = 0 &= \textstyle\sum_{i=1}^{N} 2(Y_i - ax_i - b)(-1) \end{aligned}$$

- Dividing each of these equations by -2 and expanding the summation, we get the so-called **normal equations**

$$\boxed{\begin{aligned} a\textstyle\sum x_i^2 + b\sum x_i &= \sum x_i Y_i \\ a\textstyle\sum x_i + bN &= \sum Y_i \end{aligned}}$$

  From $i = 1$ to $i = N$.

- Solving these equations simultaneously gives the values for slope and intercept a & b.

- For the data in Fig. 7.8 we find that

$$N = 5, \quad \sum T_i = 273.1, \quad \sum T_i^2 = 18607.27,$$

$$\sum R_i = 4438, \quad \sum T_i R_i = 254932.5$$

- Our **normal equations** are then

$$\begin{aligned} 18607.27a + 273.1b &= 254932.5 \\ 273.1a + 5b &= 4438 \end{aligned}$$

- From these we find $a = 3.395$, $b = 702.2$, and

$$R = 702.2 + 3.395T$$

7.4.1 Nonlinear Data (Curve Fitting)

- In many cases, data from experimental tests are *not linear*, so we need to fit to them some function other than a first-degree polynomial.
- Popular forms are the exponential form

$$y = ax^b \quad or \quad y = ae^{bx}$$

- The exponential forms are usually linearized by taking logarithms before determining the parameters, for the case $y = ax^b$:

$$lny = lna + blnx \quad or \quad lny = lna + bx$$

- We now fit the new variable $z = lny$ as a linear function of lnx or x as described earlier.
- Here we do not minimize the sum of squares of the deviations of Y from the curve, but rather the deviations of lnY.
  - In effect, this amounts to minimizing the squares of the percentage errors, which itself may be a desirable feature.

7.4.2 Least-Squares Polynomials

- Because polynomials can be readily manipulated, fitting such functions to data that do not plot linearly is common.
- We assume the functional relationship

$$y = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n \tag{7.5}$$

  with errors defined by

$$e_i = Y_i - y_i = Y_i - (a_0 + a_1x + a_2x^2 + \ldots + a_nx^n)$$

- We again use Y_i to represent the observed or experimental value corresponding to x_i, with x_i free of error.
- We minimize the sum of squares;

$$S = \sum_{i=1}^{N} e_i^2 = \sum_{i=1}^{N} (Y_i - a_0 - a_1x - a_2x^2 - \ldots - a_nx^n)^2$$

  At the minimum, all the partial derivatives $\partial S/\partial a_0, \partial S/\partial a_n$ vanish.

- Writing the equations for these gives $n+1$ equations:

$$\begin{aligned}
\tfrac{\partial S}{\partial a_0} &= 0 = \textstyle\sum_{i=1}^{N} 2(Y_i - a_0 - a_1 x_i - a_2 x_i^2 - \ldots - a_i x_i^n)(-1) \\
\tfrac{\partial S}{\partial a_1} &= 0 = \textstyle\sum_{i=1}^{N} 2(Y_i - a_0 - a_1 x_i - a_2 x_i^2 - \ldots - a_i x_i^n)(-x_i) \\
&\vdots \\
\tfrac{\partial S}{\partial a_n} &= 0 = \textstyle\sum_{i=1}^{N} 2(Y_i - a_0 - a_1 x_i - a_2 x_i^2 - \ldots - a_i x_i^n)(-x_i^n)
\end{aligned}$$

- Dividing each by -2 and rearranging gives the $n+1$ **normal equations** to be solved simultaneously:

$$\begin{aligned}
a_0 N + a_1 \textstyle\sum x_i + a_2 \sum x_i^2 + \ldots + a_n \sum x_i^n &= \textstyle\sum Y_i \\
a_0 \textstyle\sum x_i + a_1 \sum x_i^2 + a_2 \sum x_i^3 + \ldots + a_n \sum x_i^{n+1} &= \textstyle\sum x_i Y_i \\
a_0 \textstyle\sum x_i^2 + a_1 \sum x_i^3 + a_2 \sum x_i^4 + \ldots + a_n \sum x_i^{n+2} &= \textstyle\sum x_i^2 Y_i \\
&\vdots \\
a_0 \textstyle\sum x_i^n + a_1 \sum x_i^{n+1} + a_2 \sum x_i^{n+2} + \ldots + a_n \sum x_i^{2n} &= \textstyle\sum x_i^n Y_i
\end{aligned} \tag{7.6}$$

- Putting these equations in matrix form shows the coefficient matrix;

$$\begin{bmatrix}
N & \sum x_i & \sum x_i^2 & \sum x_i^3 & \ldots & \sum x_i^n \\
\sum x_i & \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \ldots & \sum x_i^{n+1} \\
\sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \ldots & \sum x_i^{n+2} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\sum x_i^n & \sum x_i^{n+1} & \sum x_i^{n+2} & \sum x_i^{n+3} & \ldots & \sum x_i^{2n}
\end{bmatrix} [a] = \begin{bmatrix}
\sum Y_i \\ \sum x_i Y_i \\ \sum x_i^2 Y_i \\ \vdots \\ \sum x_i^n Y_i
\end{bmatrix} \tag{7.7}$$

All the summatins in Eqs. 7.6 and 7.7 run from 1 to N. We will let B stand for the coefficient matrix.

- Equation 7.7 represents a linear system.

- Degrees higher than 4 are used very infrequently. It is often better to fit a series of lower-degree polynomials to subsets of the data.

- Matrix B of Eq. 7.7 is called the *normal matrix* for the least-squares problem.

- There is another matrix that corresponds to this, called the *design matrix*. It is of the form;

$$A = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
x_1 & x_2 & x_3 & \ldots & x_N \\
x_1^2 & x_2^2 & x_3^2 & \ldots & x_N^2 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
x_1^n & x_2^n & x_3^n & \ldots & x_N^n
\end{bmatrix}$$

- AA^T is just the coefficient matrix of Eq. 7.7. It is easy to see that Ay, where y is the column vector of Y-values, gives the right-hand side of Eq. 7.7. We can rewrite Eq. 7.7 in matrix form, as

$$AA^T a = Ba = Ay$$

  so it is to find the solution.

- It is illustrated the use of Eq. 7.6 to fit a quadratic to the data of Table 7.7. Figure 7.8 shows a plot of the data.

- The data are actually a perturbation of the relation $y = 1 - x + 0.2x^2$.

To set up the **normal equations**, we need the sums tabulated in Table 7.7.

x_i	0.05	0.11	0.15	0.31	0.46	0.52	0.70	0.74	0.82	0.98	1.171
Y_i	0.956	0.890	0.832	0.717	0.571	0.539	0.378	0.370	0.306	0.242	0.104

$\Sigma x_i = 6.01$ $\quad N = 11$

$\Sigma x_i^2 = 4.6545$ $\quad \Sigma Y_i = 5.905$

$\Sigma x_i^3 = 4.1150$ $\quad \Sigma x_i Y_i = 2.1839$

$\Sigma x_i^4 = 3.9161$ $\quad \Sigma x_i^2 Y_i = 1.3357$

Table 7.7: Data to illustrate curve fitting.

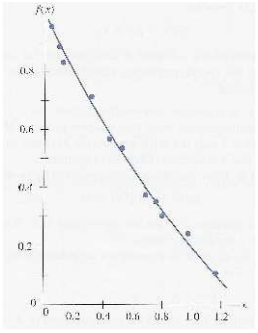


Table 7.8: Figure for the data to illustrate curve fitting.

- The equations to be solved are:

$$\begin{aligned} 11a_0 + 6.01a_1 + 4.6545a_2 &= 5.905 \\ 6.01a_0 + 4.6545a_1 + 4.1150a_2 &= 2.1839 \\ 4.6545a_0 + 4.1150a_1 + 3.9161a_2 &= 1.3357 \end{aligned}$$

  The result is $a_0 = 0.998$, $a_2 = -1.018$, $a_3 = 0.225$.

- So the least- squares method gives

$$y = 0.998 - 1.018x + 0.225x^2$$

  which we compare to $y = 1 - x + 0.2x^2$. Errors in the data cause the equations to differ.

Example py-file: Fitting an 4^{th} order polynomial to $y = cosx$ function in $[0,\pi]$ by Least-Square Approximation. Gaussian elimination & back substitution. Pivoting. mylsa.py

```
MyGEPivShow - Solution Vector :  [[ 9.88494865e-01  8.91986965e-02 -6.82520463e-01  1.42129999e-01
   7.82504827e-04]]
NumPy Solve - Solution Vector : [[ 9.88494865e-01  8.91986965e-02 -6.82520463e-01  1.42129999e-01
   7.82504827e-04]]
 Step          Exact        Least-Square Approx.
  x            cos x              p(x)                Error
----- ------------------- ------------------- ---------------------
 0.32   0.9492354180824408   0.9518138732787098   -0.0025784551962690
 0.64   0.8020957578842927   0.8034114580487246   -0.0013157001644319
 0.96   0.5735199860724567   0.5715268997183653    0.0019930863540913
 1.28   0.2867152096319555   0.2845964026657791    0.0021188069661764
 1.60   -0.0291995223012888  -0.0287469049633018   -0.0004526173579870
 1.92   -0.3421496511508982  -0.3396729711555569   -0.0024766799953413
 2.24   -0.6203616120126798  -0.6191548202300811   -0.0012067917825987
 2.56   -0.8355887771314077  -0.8379685526383827    0.0023797755069750
 2.88   -0.9659793123979747  -0.9666933450643868    0.0007140326664120
 3.20   -0.9982947757947531  -0.9757114504044269   -0.0225833253903263
```

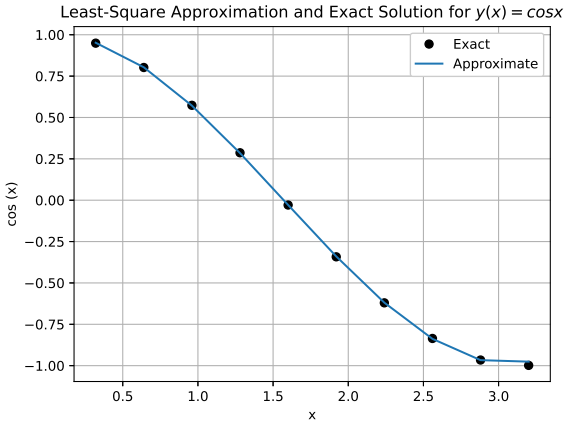


Figure 7.10: Polynomial Least-Square Approximation.

7.4.3 Millikan oil-drop experiment

- In 1910, Millikan succeeded in measuring the electron charge for the first time with a very sensitive experiment.
- In this experiment, electrically charged oil droplets remained suspended in the air under the influence of a force that balanced the electric field applied between the plates of a capacitor and the gravitational force.
- From the equation $qV/d = mg$, the electric charge of each droplet could be calculated by measuring the potential difference and masses.
- When Millikan listed these electric charges from smallest to largest, he showed that they were multiples of a basic unit of charge, and from there he determined the electric charge as $e = 1.65 \times 10^{-9}\ C$ (Today's value is $e = 1.602 \times 10^{-9}\ C$).

Example py-file: Millikan oil-drop experiment by Least-Square Approximation. Gaussian elimination & back substitution. Pivoting. mylsa_millikan.py

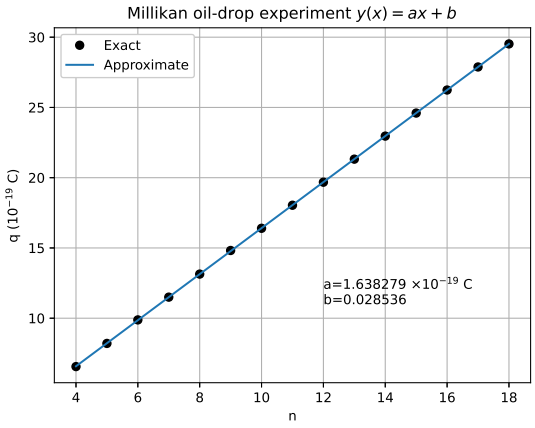


Figure 7.11: Millikan oil-drop experiment.

Chapter 8

References:

- https://github.com/gtaffoni/Learn-Python/tree/master/Lectures
- https://python-course.eu/
- https://www.codecademy.com/catalog/language/python
- https://scipy-lectures.org/
- https://www.naukri.com/learning/articles/top-10-powerful-python-libraries-for-data-science/
- https://computation.physics.utoronto.ca/tutorials/
- https://moodle2.units.it/course/view.php?id=6837
- https://jckantor.github.io/CBE30338/
- https://matplotlib.org/stable/tutorials/index.html
- http://hyperphysics.phy-astr.gsu.edu/
- https://chem.libretexts.org/