AoL Scientific Computing

Nama: Reynard Hansel

NIM: 2602072003

1. Data:

A picture containing text, screenshot, number, font

Description automatically generated

Data in code:

years = np.array([1981, 1983, 1985, 1987, 1989, 1991, 1993, 1995, 1997, 1999])

temperatures = np.array([14.1999, 14.2411, 14.0342, 14.2696, 14.197, 14.3055, 14.1853, 14.3577, 14.4187, 14.3438])

x\_hat = np.array([1988]) *# For finding a single or specifics even year*

x\_hats = np.arange(1982, 2000, 2) *# For finding all even years*

Libraries to import:

import matplotlib.pyplot as plt

import numpy as np

1. Estimate the temperature in even years by linear, quadratic, and cubic interpolation order! Choose the method that you think is appropriate, and explain the difference

Method chosen: Linear & Quadratic interpolation

Linear interpolation assumes that the data to be guessed is in a straight line between the point before and the point after (constant change). This makes things and calculations simple, but it is limited to only a straight linear line between one data point to another.

Quadratic interpolation, on the other hand, uses quadratic curves to estimate values. This makes it possible to finally estimate a value outside of a linear straight line, making it usually being more accurate. However, this makes calculations more complicated

Linear interpolation:

def linear\_interpolate(x, y, x\_hats):

    n = len(x)

    results = []

    for x\_hat in x\_hats:

        for j in range(n-1):

            if x[j] <= x\_hat and x[j+1] >= x\_hat:

                i = j

        y\_hat = y[i] + ((y[i+1] - y[i]) \* (x\_hat - x[i]) / (x[i+1] - x[i]))

        results.append(y\_hat)

    return results

estimated\_temperatures = linear\_interpolate(years, temperatures, x\_hats)

*# Printing each value*

print('Linear Spline Interpolation:')

for year, temperature in zip(x\_hats, estimated\_temperatures):

    print(f"Estimated temperature in {year}: {temperature:.4f} °C")

print('\n\n')

*# Plot*

plt.figure(figsize=(10, 8))

plt.plot(years, temperatures, "-ob", label="Given Data")

plt.plot(x\_hats, estimated\_temperatures, "ro", label="Estimated Temperatures")

plt.xlabel("Year")

plt.ylabel("Temperature (°C)")

plt.title("Linear Interpolation")

plt.legend()

*# Loop to print each value of the red dots*

for x, y in zip(x\_hats, estimated\_temperatures):

    plt.text(x, y, f"{y:.2f}", ha="center", va="bottom")

*# ha --> align center*

*# va --> display below dot*

plt.show()

Output:

A screen shot of a computer

Description automatically generated with medium confidence

Quadratic Interpolation:

def quadratic\_spline\_interpolate(x, y, x\_hat):

    n = len(x)

    results = []

    for x\_val in x\_hat:

*# Find the indices of the closest values for x1, x2, x3*

        idx = np.argsort(np.abs(x - x\_val))[:3]

        x1, x2, x3 = x[idx]

        y1, y2, y3 = y[idx]

*# Evaluate the quadratic spline at x\_hat*

        y\_val = (y1 \* ((x\_val - x2) \* (x\_val - x3)) / ((x1 - x2) \* (x1 - x3)) +

                 y2 \* ((x\_val - x1) \* (x\_val - x3)) / ((x2 - x1) \* (x2 - x3)) +

                 y3 \* ((x\_val - x1) \* (x\_val - x2)) / ((x3 - x1) \* (x3 - x2)))

        results.append(y\_val)

    return results

estimated\_temperatures = quadratic\_spline\_interpolate(years, temperatures, x\_hats)

*# Print each value*

print('Quadratic Spline Interpolation:')

for year, temperature in zip(x\_hats, estimated\_temperatures):

    print(f"Estimated temperature in {year}: {temperature:.4f} °C")

print('\n\n')

*# Plot*

plt.figure(figsize=(10, 8))

plt.plot(years, temperatures, "-ob", label="Given Data")

plt.plot(x\_hats, estimated\_temperatures, "ro", label="Estimated Temperatures")

plt.xlabel("Year")

plt.ylabel("Temperature (°C)")

plt.title("Quadratic Spline Interpolation")

plt.legend()

*# Loop to print each value of the red dots*

for x, y in zip(x\_hats, estimated\_temperatures):

    plt.text(x, y, f"{y:.2f}", ha="center", va="bottom")

plt.show()

Output:

A screenshot of a computer

Description automatically generated with medium confidence

1. Linear regression (least-square method)

def linear\_regression(x, y):

    n = len(x)

    sum\_x = np.sum(x)

    sum\_y = np.sum(y)

    sum\_xy = np.sum(x \* y)

    sum\_x\_squared = np.sum(x \*\* 2)

    slope = (n \* sum\_xy - sum\_x \* sum\_y) / (n \* sum\_x\_squared - sum\_x \*\* 2)

    intercept = (sum\_y - slope \* sum\_x) / n

    return slope, intercept

slope, intercept = linear\_regression(years, temperatures)

estimated\_temperatures = slope \* x\_hats + intercept

*# Print each values*

for year, temperature in zip(x\_hats, estimated\_temperatures):

    print(f"Estimated temperature in {year}: {temperature:.4f} °C")

*# Plot*

plt.figure(figsize=(10, 8))

plt.plot(years, temperatures, "bo", label="Given Data")

plt.plot(x\_hats, estimated\_temperatures, "ro", label="Estimated Temperatures")

plt.plot(years, linear\_regression(years, temperatures)[0] \* years + linear\_regression(years, temperatures)[1], "--g", label="Regression Line")

plt.xlabel("Year")

plt.ylabel("Temperature (°C)")

plt.title("Least Squares Regression")

plt.legend()

for x, y in zip(x\_hats, estimated\_temperatures):

    plt.text(x, y, f"{y:.2f}", ha="center", va="bottom")

plt.show()

Output:

A screen shot of a computer

Description automatically generated with low confidence

1. Generally, interpolations are used for figuring out/guessing an unknown data between two known data points. And regressions are used to figure out a trend that’s going on in a few known data points, this makes it possible to guess future possible data (forecasting) as long as the same trend is still going/happening.

Between linear and quadratic interpolation, I say that quadratic spline interpolation is better for this case. This is because the trend of the data is not linear, there’s lots of ups and downs, which makes it really possible that a data is not going linear upwards or downwards. If the known datas are continously increasing or decreasing, then using linear interpolation would be better.

With quadratic spline interpolation, it is possible to guess the unknown data outside of a linear line between two known data points, which is why I’m saying this method is better suited for this case.

Using linear regression, we figured out the linear trend that is going upwards, and the data for even years that we guessed are also within that linear trend, it’s locked to that linear trend, which is why it’s definitely not the best method. But if we were to guess what data might be possible for the unknown future, example: for the year 2002, then the linear regression would be more useful

Overall, I say/state that the best method to find data for even years, is using quadratic spline interpolation, however linear regression would be the best if we were trying to do a forecast.

1. Plotting

* Linear spline interpolation

*# Plot*

plt.figure(figsize=(10, 8))

plt.plot(years, temperatures, "-ob", label="Given Data")

plt.plot(x\_hats, estimated\_temperatures, "ro", label="Estimated Temperatures")

plt.xlabel("Year")

plt.ylabel("Temperature (°C)")

plt.title("Linear Interpolation")

plt.legend()

*# Loop to print each value of the red dots*

for x, y in zip(x\_hats, estimated\_temperatures):

    plt.text(x, y, f"{y:.2f}", ha="center", va="bottom")

*# ha --> align center*

*# va --> display below dot*

plt.show()

* Quadratic spline interpolation

*# Plot*

plt.figure(figsize=(10, 8))

plt.plot(years, temperatures, "-ob", label="Given Data")

plt.plot(x\_hats, estimated\_temperatures, "ro", label="Estimated Temperatures")

plt.xlabel("Year")

plt.ylabel("Temperature (°C)")

plt.title("Quadratic Spline Interpolation")

plt.legend()

*# Loop to print each value of the red dots*

for x, y in zip(x\_hats, estimated\_temperatures):

    plt.text(x, y, f"{y:.2f}", ha="center", va="bottom")

plt.show()

* Linear regression (least square method)

*# Plot*

plt.figure(figsize=(10, 8))

plt.plot(years, temperatures, "bo", label="Given Data")

plt.plot(x\_hats, estimated\_temperatures, "ro", label="Estimated Temperatures")

plt.plot(years, linear\_regression(years, temperatures)[0] \* years + linear\_regression(years, temperatures)[1], "--g", label="Regression Line")

plt.xlabel("Year")

plt.ylabel("Temperature (°C)")

plt.title("Least Squares Regression")

plt.legend()

for x, y in zip(x\_hats, estimated\_temperatures):

    plt.text(x, y, f"{y:.2f}", ha="center", va="bottom")

plt.show()

Graph/plot output:

* A picture containing text, diagram, screenshot, line

  Description automatically generatedLinear interpolation
* Quadratic Interpolation
* A picture containing text, diagram, screenshot, line

  Description automatically generatedLinear regression

A picture containing text, screenshot, diagram, line

Description automatically generated

1. Compute the fourth order Taylor expansion for sin(x) and cos(x) and sin(x)cos(x) around 0
2. Calculate manually & using python

* Python

import math

def taylor\_series(term\_fn, x, n):

    series\_sum = 0

    for i in range(n):

        term = term\_fn(x, i)

        series\_sum += term

    return series\_sum

def sin(x, i):

    return (-1) \*\* i \* x \*\* (2 \* i + 1) / math.factorial(2 \* i + 1)

def cos(x, i):

    return (-1) \*\* i \* x \*\* (2 \* i) / math.factorial(2 \* i)

def sin\_cos(x, i):

    return (-1) \*\* i \* x \*\* (2 \* i + 1) / math.factorial(2 \* i + 1)

x = 0  *# Centered around 0 --> untuk soal a*

*# x = math.pi / 4 --> untuk soal c*

n = 4  *# Fourth-order*

sin\_approx = taylor\_series(sin, x, n)

cos\_approx = taylor\_series(cos, x, n)

sin\_cos\_approx = taylor\_series(sin\_cos, x, n)

print("sin(x) =", sin\_approx)

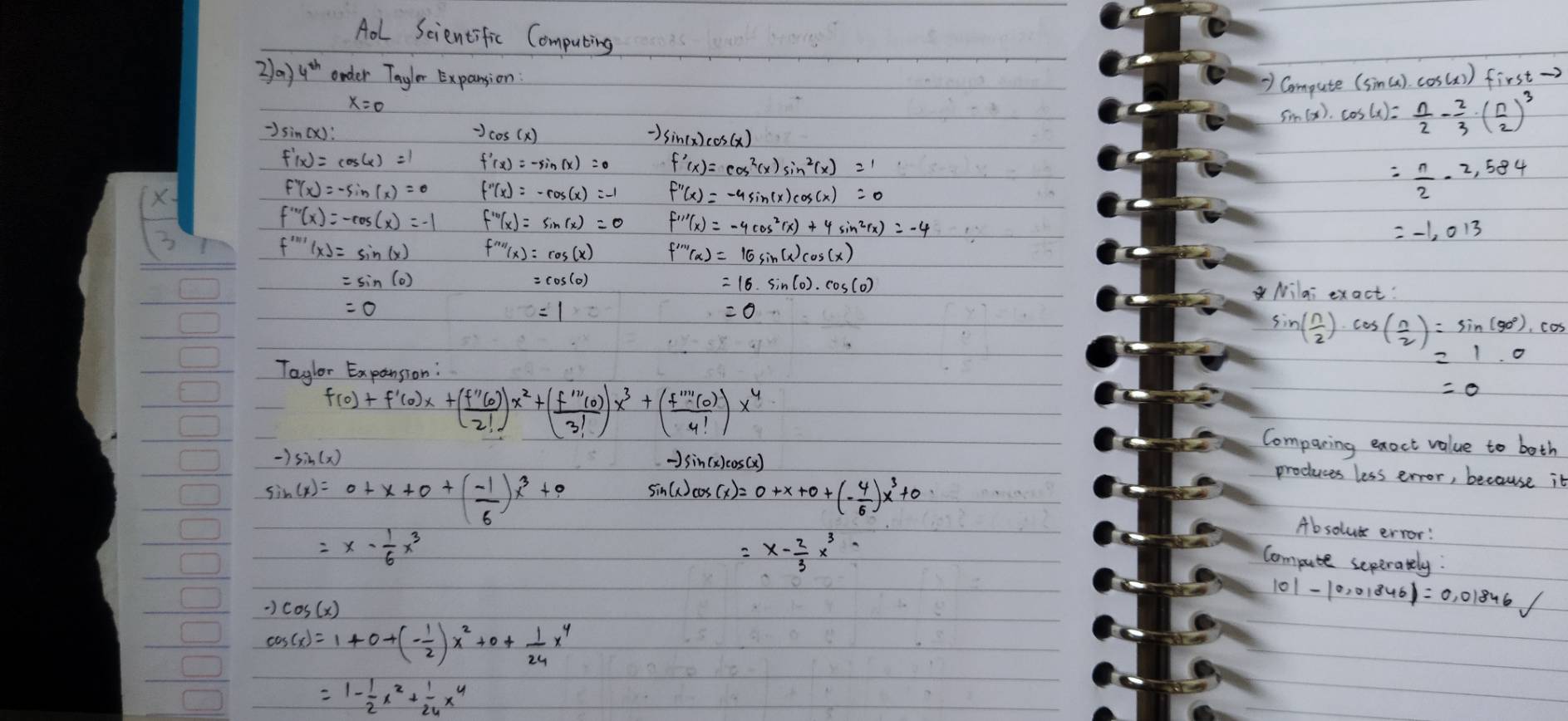
print("cos(x) =", cos\_approx)

print("sin(x)cos(x) =", sin\_cos\_approx)

Output:  
A picture containing text, screenshot, font

Description automatically generated

* Manually



1. Which produces less error for x=π/2: computing the Taylor expansion for sin and cos separately then multiplying the result together, or computing the Taylor expansion for the product first then plugging in x?

A picture containing text, handwriting, document, ink

Description automatically generatedA picture containing text, handwriting, notebook, book

Description automatically generated

1. Use the same order of Taylor series to approximate cos (π/4) and determine the truncation error bound.

Python:

import math

def taylor\_series(term\_fn, x, n):

    series\_sum = 0

    for i in range(n):

        term = term\_fn(x, i)

        series\_sum += term

    return series\_sum

def sin(x, i):

    return (-1) \*\* i \* x \*\* (2 \* i + 1) / math.factorial(2 \* i + 1)

def cos(x, i):

    return (-1) \*\* i \* x \*\* (2 \* i) / math.factorial(2 \* i)

def sin\_cos(x, i):

    return (-1) \*\* i \* x \*\* (2 \* i + 1) / math.factorial(2 \* i + 1)

*#* *x = 0  # Centered around 0 --> untuk soal a*

x = math.pi / 4 *#c --> untuk soal c*

n = 4  *# Fourth-order*

sin\_approx = taylor\_series(sin, x, n)

cos\_approx = taylor\_series(cos, x, n)

sin\_cos\_approx = taylor\_series(sin\_cos, x, n)

print("sin(x) =", sin\_approx)

print("cos(x) =", cos\_approx)

print("sin(x)cos(x) =", sin\_cos\_approx)

sin\_error = abs(math.sin(x) - sin\_approx)

cos\_error = abs(math.cos(x) - cos\_approx)

sin\_cos\_error = abs((math.sin(x) \* math.cos(x)) - sin\_cos\_approx)

print("The error for sin(x) is:", sin\_error)

print("The error for cos(x) is:", cos\_error)

print("The error for sin(x)cos(x) is:", sin\_cos\_error)

Note: Hanya mengganti ‘x = 0’ menjadi ‘x = math.pi / 4’, lalu menambah:

sin\_error = abs(math.sin(x) - sin\_approx)

cos\_error = abs(math.cos(x) - cos\_approx)

sin\_cos\_error = abs((math.sin(x) \* math.cos(x)) - sin\_cos\_approx)

untuk menghitung error-error nya

Output:

A screen shot of a computer program

Description automatically generated with low confidence

1. Given that
2. Approximate   with 20 evenly spaced grid points over the whole interval using Riemann Integral, Trapezoid Rule, and Simpson’s Rule. Explain the difference behind each of the method

Python:

import numpy as np

def riemann(a, b, h, t, v, mode='left'):

    i = 0

    j = len(t) - 1

    y = 0

    if mode == 'left':

        for k in v[i:j]:

            y += h \* k

    elif mode == 'right':

        for k in v[i+1 : j+1]:

            y += h \* k

    return y

def trapezoid(a, b, h, t, v):

    i = 0

    j = len(t) - 1

    n = 0

    for k in v[i+1 : j]:

        n += k

    return (v[i] + (2\*n) + v[j]) \* (h/2)

def simpsons(a, b, h, t, v):

    i = 0

    j = len(t) - 1

    n = len(t)

    if n < 3:

        print('Data range is too small')

        return -1

*# Rule 1/3 (Interval = even, length = odd)*

    if n % 2 != 0:

        sum1 = 0

        for k in v[i+1 : j : 2]:

            sum1 += k

        sum2 = 0

        for k in v[i+2 : j-1 : 2]:

            sum2 += k

        y = (v[i] + (4\*sum1) + (2\*sum2) + v[j]) \* (h/3)

*# Rule 3/8*

    elif n % 2 == 0:

        sum3 = (v[i] + (3 \* v[i+1]) + (3 \* v[i+2]) + v[i+3]) \* ((3\*h) / 8)

*# 1/3 rules*

        v\_new = v[i+3 : j+1] *# --> remaining data*

        sum1 = 0

        for k in v\_new[1 : -1 : 2]:

            sum1 += k

        sum2 = 0

        for k in v\_new[2 : -2 : 2]:

            sum2 += k

        y = sum3 + ((h/3) \* (v\_new[0] + (4\*sum1) + (2\*sum2) + v\_new[-1]))

    return y

*# Define the function f(x)*

def f(x):

    return x\*\*3 - 0.3\*x\*\*2 - 8.56\*x + 8.448

*# Define the interval [a, b]*

a = 0

b = 2 \* np.pi

*# Generate evenly spaced grid points*

num\_points = 20

t = np.linspace(a, b, num\_points)

v = f(t)

*# Calculate the step size*

h = (b - a) / (num\_points - 1)

*# Calculate the integral using different methods*

riemann\_integral\_left = riemann(a, b, h, t, v)

riemann\_integral\_right = riemann(a, b, h, t, v, mode='right')

trapezoid\_integral = trapezoid(a, b, h, t, v)

simpsons\_integral = simpsons(a, b, h, t, v)

*# Print the results*

print("Riemann Integral (Left):", riemann\_integral\_left)

print("Riemann Integral (Right):", riemann\_integral\_right)

print("Trapezoid Rule:", trapezoid\_integral)

print("Simpson's Rule:", simpsons\_integral)

Output:

A picture containing text, screenshot, font

Description automatically generated

* Riemann Integral approximates the area under the curve (basically what an integral is) by using rectangles or trapezoids. It has left, midpoint, and right methods, which uses the left, midpoint, and right endpoint of each interval respectively to determine the height of the rectangle or trapezoid.
* Trapezoid rule approximates the area under the curve by summing (adding) the areas of the trapezoids which height is determined by the function values. It provides a better approximation because instead of using a rectangle, it uses trapezoids which have curves, thus increasing the accuracy.
* Simpson's Rule provides an even more accurate approximation by using quadratic polynomial interpolation. It divides the interval into subintervals then uses a combination of the endpoints and the midpoint of each subinterval to determine the height of the quadratic polynomial. It then approximates the area under the curve by using quadratic polynomials to all pair of subintervals that are next to each other (bersebelahan)

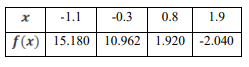
1. Compared to the methods above, do you think that analitycal integration could be more convenient to be done?

No, it would not be as convenient as using numerical integration in this case. This is because:

* The function has a degree of 3. Using analytical integration would be very time consuming and error-prone, and is quite complicated to count.
* It is stated that we have ‘20 evenly-spaced grid points over the whole interval’, which means the data points are limited, making obtaining an accurate exact value quite challenging. Plus, this suggests that the function is given in numerical form at specific points rather than as an explicit analytical expression, making numerical integration more natural of a choice because it is designed to handle this kind of scenario
* Analytic integration relies on known techniques and formulas for specific types of functions, translating them to computer code would take quite some time, unless the language has a built in function or library. But still, numerical would be easier

So, no, unless we’re in need of an exact accurate value from an analytical integration, numerical integration would be more convenient, plus, numerical integration functions could only be written once in a programming language and can be used for almost all types of functions given, including .

1. Use polynomial interpolation to compute f’(x) and f’’(x) at x = 0, using the discrete data below



*# Data*

x = np.array([-1.1, -0.3, 0.8, 1.9])

f\_x = np.array([15.180, 10.962, 1.920, -2.040])

*# Calculate coefficients of the interpolating polynomial*

coefficients = np.polyfit(x, f\_x, deg=len(x) - 1)

*# Create a polynomial using coefficients*

polynomial = np.poly1d(coefficients)

*# Compute f'(x) and f''(x)*

f\_prime = np.polyder(polynomial)

f\_double\_prime = np.polyder(f\_prime)

*# f'(x) & f''(x) at x = 0*

f\_prime\_0 = f\_prime(0)

f\_double\_prime\_0 = f\_double\_prime(0)

*# Print*

print("f'(0) =", f\_prime\_0)

print("f''(0) =", f\_double\_prime\_0)

Output:

A screenshot of a computer screen

Description automatically generated with low confidence

1. Calculate the accuracy result compared to the initial f(x)

*# Define analytic functions for f'(x) & f''(x)*

def f\_prime\_init(x):

    return 3\*x\*\*2 - 0.6\*x - 8.56

def f\_double\_prime\_init(x):

    return 6\*x - 0.6

*# Calculate when x = 0*

f\_prime\_initial\_0 = f\_prime\_init(0)

f\_double\_prime\_initial\_0 = f\_double\_prime\_init(0)

*# Calculate accuracy (absolute(numerical - analytic))*

accuracy\_f\_prime = abs(f\_prime\_0 - f\_prime\_initial\_0)

accuracy\_f\_double\_prime = abs(f\_double\_prime\_0 - f\_double\_prime\_initial\_0)

*#print*

print("Accuracy of f'(0) compared to the initial function:", accuracy\_f\_prime)

print("Accuracy of f''(0) compared to the initial function:", accuracy\_f\_double\_prime)

Output:

