AoL Scientific Computing

Nama: Reynard Hansel

NIM: 2602072003

1. Data:

Year (y)	Temperature (x, °C)
1981	14.1999
1983	14.2411
1985	14.0342
1987	14.2696
1989	14.197
1991	14.3055
1993	14.1853
1995	14.3577
1997	14.4187
1999	14.3438

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
```

   a. 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:

```
PS E:\Reynard\Uni\Sem 2\SC> python -u "e:\Reynard\Un
Linear Spline Interpolation:
Estimated temperature in 1982: 14.2205 °C
Estimated temperature in 1984: 14.1377 °C
Estimated temperature in 1986: 14.1519 °C
Estimated temperature in 1988: 14.2333 °C
Estimated temperature in 1990: 14.2512 °C
Estimated temperature in 1992: 14.2454 °C
Estimated temperature in 1994: 14.2715 °C
Estimated temperature in 1996: 14.3882 °C
Estimated temperature in 1998: 14.3812 °C
```

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:

```
Quadratic Spline Interpolation:
Estimated temperature in 1982: 14.2515 °C
Estimated temperature in 1984: 14.1687 °C
Estimated temperature in 1986: 14.0966 °C
Estimated temperature in 1988: 14.2718 °C
Estimated temperature in 1990: 14.2286 °C
Estimated temperature in 1992: 14.2740 °C
Estimated temperature in 1994: 14.2349 °C
Estimated temperature in 1996: 14.4021 °C
Estimated temperature in 1998: 14.3982 °C
```

b. 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:

```
Estimated temperature in 1982: 14.1580 °C
Estimated temperature in 1984: 14.1823 °C
Estimated temperature in 1986: 14.2067 °C
Estimated temperature in 1988: 14.2310 °C
Estimated temperature in 1990: 14.2553 °C
Estimated temperature in 1992: 14.2796 °C
Estimated temperature in 1994: 14.3039 °C
Estimated temperature in 1996: 14.3282 °C
Estimated temperature in 1998: 14.3525 °C
```

c. 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.

d. 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

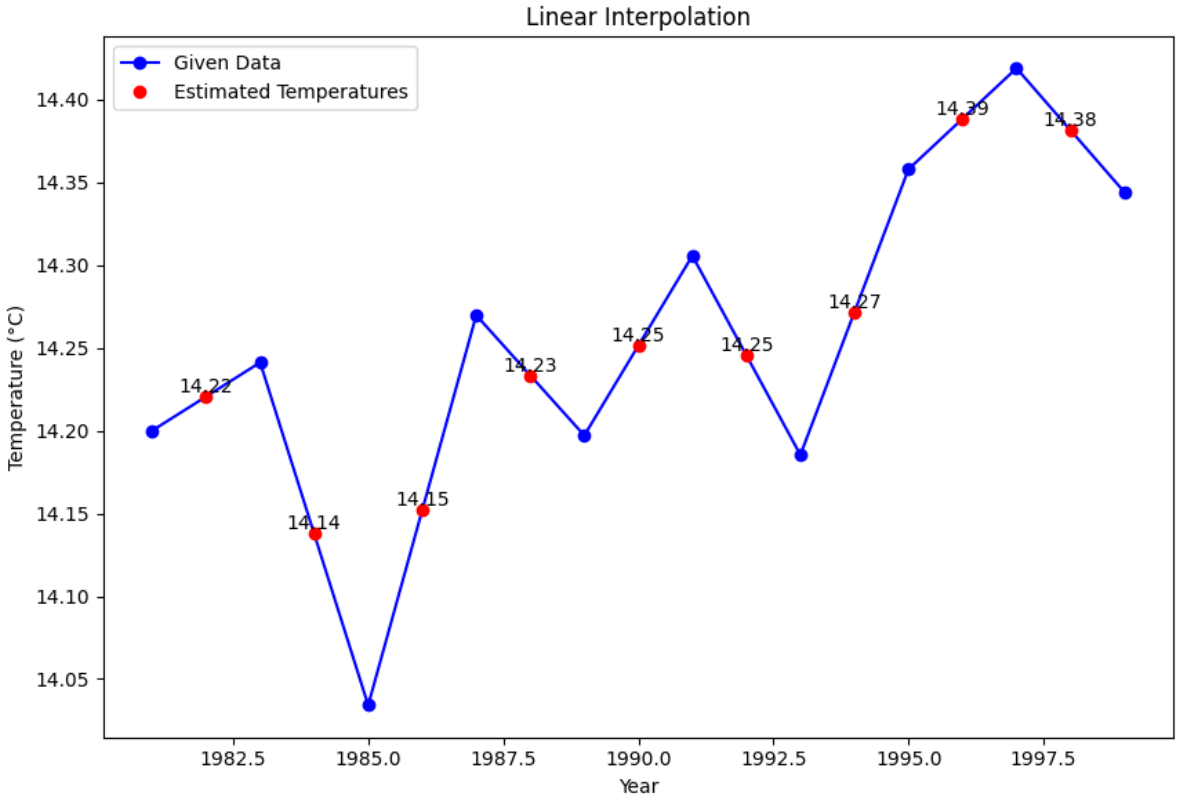
```
# Plot
plt.figure(figsize=(10, 8))
plt.plot(years, temperatures, "-ob", label="Given Data")
plt.plot(x_hats, estimated_temperatures, "ro", label="Estimated Temperatures")
plt.xlabel("Year")
plt.ylabel("Temperature (°C)")
plt.title("Quadratic Spline Interpolation")
plt.legend()

# 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)

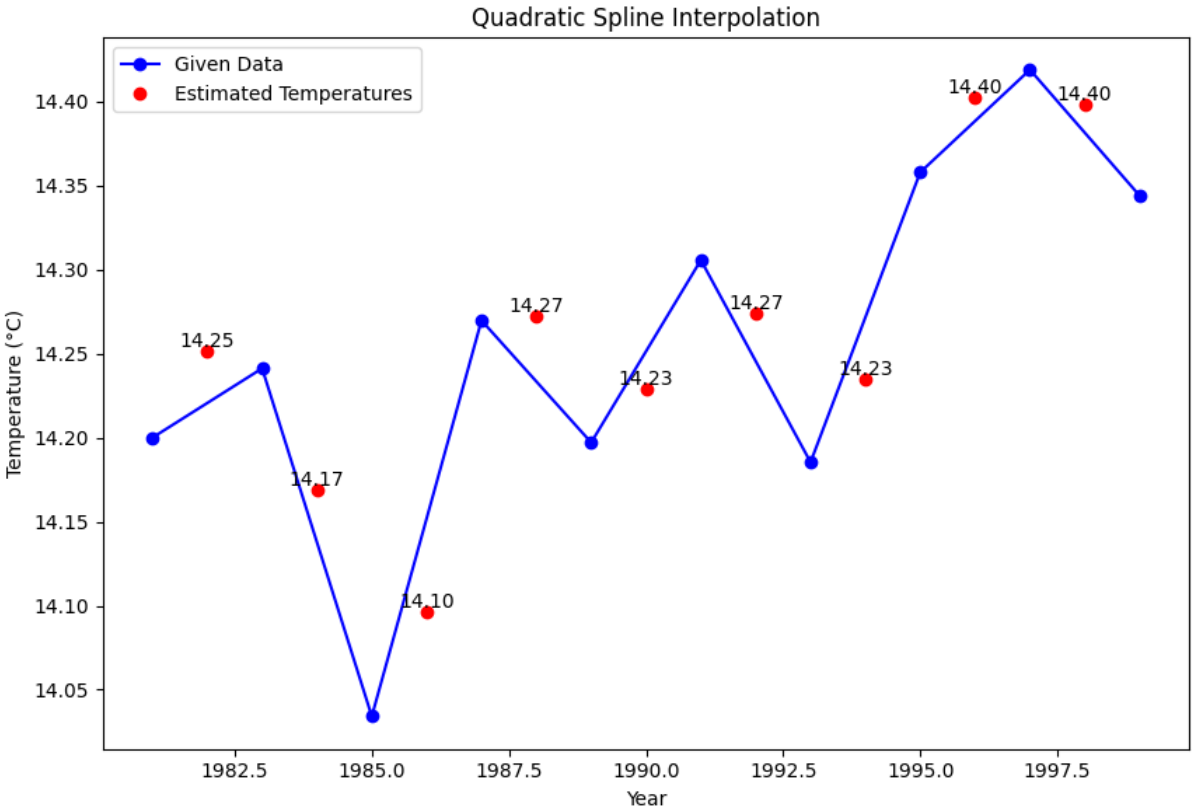
```
# Plot
plt.figure(figsize=(10, 8))
plt.plot(years, temperatures, "bo", label="Given Data")
plt.plot(x_hats, estimated_temperatures, "ro", label="Estimated Temperatures")
plt.plot(years, linear_regression(years, temperatures)[0] * years +
linear_regression(years, temperatures)[1], "--g", label="Regression 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:

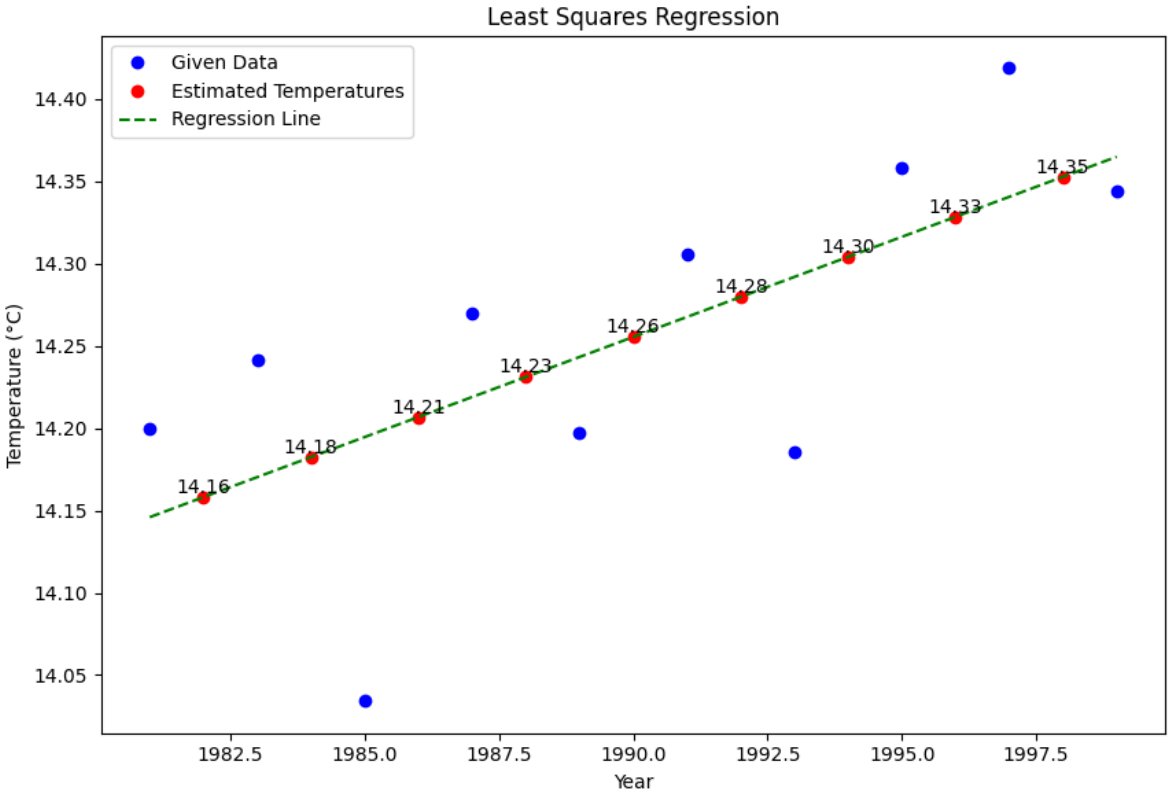
- Linear interpolation



- Quadratic Interpolation



- Linear regression



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

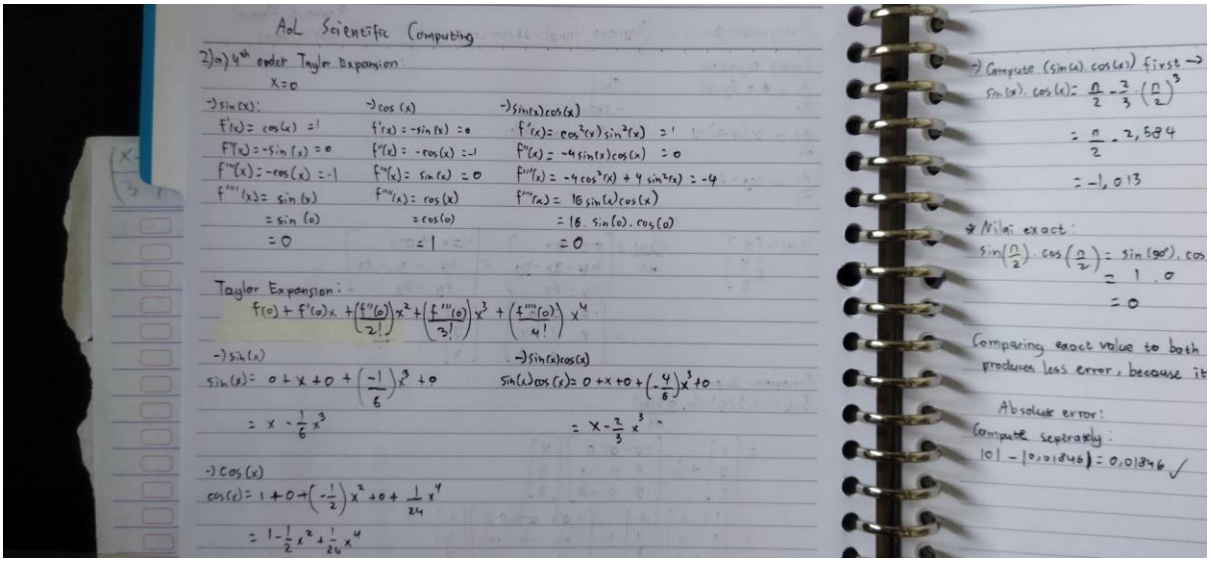
```python
import math

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)
```

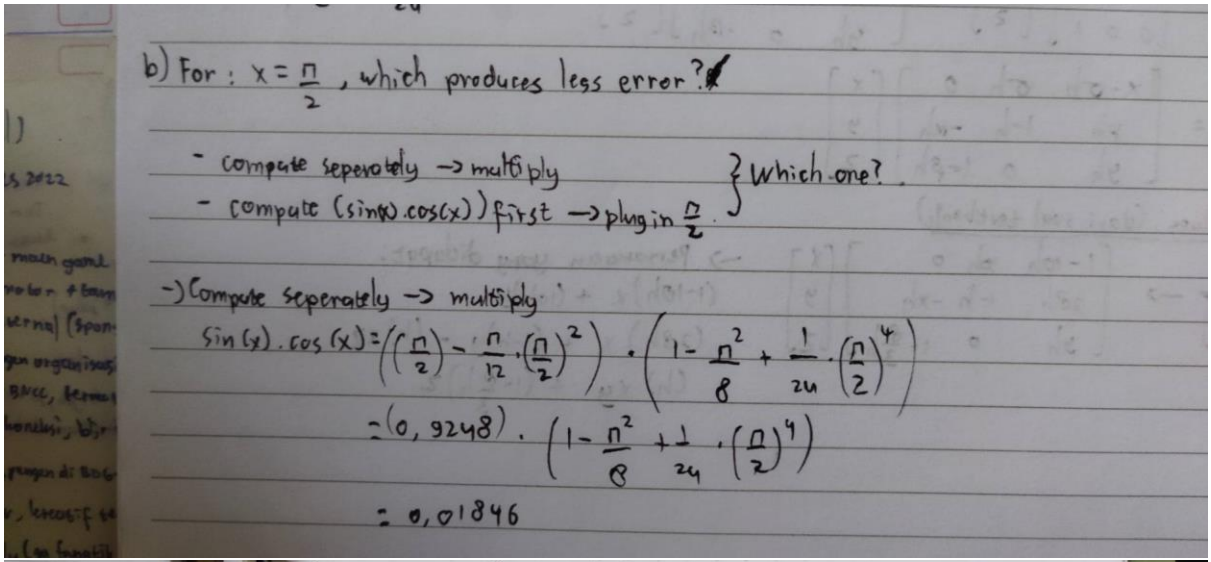
Output:

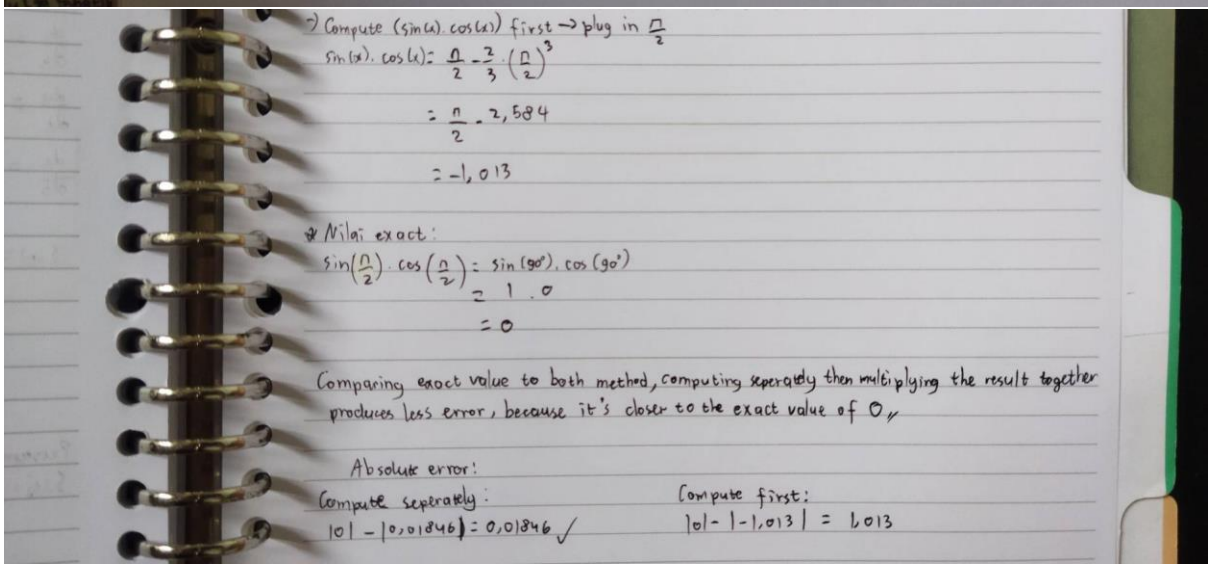
```
PS E:\Reynard\Uni\Sem 2\SC> python -u "e:\Reynard\Uni\Sem 2\SC\AoL\2.py"
sin(x) = 0.0
cos(x) = 1.0
sin(x)cos(x) = 0.0
```

- Manually



b. 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?





c. 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)
```

```python
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:

```python
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:

```
PS E:\Reynard\Uni\Sem 2\SC> python -u "e:\Reynard\Uni\Sem 2\SC\AoL\2.py"
sin(x) = 0.7071064695751781
cos(x) = 0.7071032148228457
sin(x)cos(x) = 0.7071064695751781
The error for sin(x) is: 3.116113694856537e-07
The error for cos(x) is: 3.566363701912323e-06
The error for sin(x)cos(x) is: 0.20710646957517798
PS E:\Reynard\Uni\Sem 2\SC>
```

3. Given that $f(x) = x^3 - 0.3x^2 - 8.56x + 8.448$

a. Approximate $\int_0^{2\pi} f(x)\,dx$ 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:

```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:

```
PS E:\Reynard\Uni\Sem 2\SC> python -u "e:\Reynard\Uni\Sem 2\SC\AoL\3.py"
Riemann Integral (Left): 219.82600410076236
Riemann Integral (Right): 280.15206398592125
Trapezoid Rule: 249.98903404334186
Simpson's Rule: 248.94406492017313
PS E:\Reynard\Uni\Sem 2\SC>
```

- 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)

b. 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 $f(x) = x^3 - 0.3x^2 - 8.56x + 8.448$ 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 $f(x) = x^3 - 0.3x^2 - 8.56x + 8.448$.

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

x	-1.1	-0.3	0.8	1.9
$f(x)$	15.180	10.962	1.920	-2.040

```
# 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:

```
PS E:\Reynard\Uni\Sem 2\SC> pyt
f'(0) = -8.405855263157898
f''(0) = -1.6421052631579118
```

d. 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
```

```python
# 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:

```
Accuracy of f'(0) compared to the initial function: 0.1541447368421025
Accuracy of f''(0) compared to the initial function: 1.0421052631579117
```

Full Code per number (1.py, 2.py, 3.py) with every source files:
https://github.com/ReynardHansel/AoL-Scientific-Computing