Technical University of Moldova

Homework nr.4

on Numerical Analysis

executed in Python programming language

by the student from FAF – 213 academic group

Bajenov Sevastian

Chisinau - 2022

Problem 4.1:

Solution and output:

Fresnel integrals.py | Own integral code.py | Pi approximation.py | Gamma integration.py

```python
# this task requires studying Gamma function
# and the corresponding integral for computing it
from scipy import integrate, special
from numpy import linspace
import math
import warnings


# it is more convenient to ignore the accuracy warnings occurring
# while using the built-in integration methods
warnings.filterwarnings('ignore')


def Gamma_function(t, parameter):
    return pow(t, parameter - 1) * pow(math.e, -t)


# first up let us apply the built in integration methods
# we will use Gaussian quadrature method
print('Gaussian quadrature deviation:')
x = 1
while x <= 4:
    value = integrate.quadrature(lambda t: pow(t, x - 1) * pow(math.e, -t), 0, x)
    print(f'x = {x}; relative_error = {round(abs(value[0] - special.gamma(x)) / special.gamma(x), 5)}')
    x += 0.25
print()
print()
```

```python
# secondly we apply composite Simpson's rule
print('Composite Simpson method deviation:')
x = 1
while x <= 5:
    x_args = linspace(1, 10, 1000)
    y_args = list()
    for i in range(0, len(x_args)):
        y_args.append(Gamma_function(x_args, x))

    value = integrate.simps(y_args, x_args)
    print(f'x = {x}; relative_error = {round(abs(value[0] - special.gamma(x)) / special.gamma(x), 5)}')
    x += 0.25
print()
print()
```

```python
# also we can try to use general purpose quadrature method
# and see the error in the integration itself
print('General purpose quadrature method deviation and calculation error:')
x = 1
while x <= 4:
    value = integrate.quad(lambda t: pow(t, x - 1) * pow(math.e, -t), 0, x)
    print(f'x = {x}; integral_error = {value[1]}; relative_deviation = {round(abs(value[0] - special.gamma(x)) / special.gamma(x), 5)}')
    x += 0.25
print()
print()
```

```
Gamma integration
C:\Users\User\PycharmProjects\Laboratories\
Gaussian quadrature deviation:
x = 1; relative_error = 0.36788
x = 1.25; relative_error = 0.38146
x = 1.5; relative_error = 0.39162
x = 1.75; relative_error = 0.39957
x = 2.0; relative_error = 0.40601
x = 2.25; relative_error = 0.41135
x = 2.5; relative_error = 0.41588
x = 2.75; relative_error = 0.41978
x = 3.0; relative_error = 0.42319
x = 3.25; relative_error = 0.4262
x = 3.5; relative_error = 0.42888
x = 3.75; relative_error = 0.43129
x = 4.0; relative_error = 0.43347
```

```
Composite Simpson method deviation:
x = 1; relative_error = 0.63217
x = 1.25; relative_error = 0.5263
x = 1.5; relative_error = 0.42776
x = 1.75; relative_error = 0.34
x = 2.0; relative_error = 0.26474
x = 2.25; relative_error = 0.20232
x = 2.5; relative_error = 0.1521
x = 2.75; relative_error = 0.11286
x = 3.0; relative_error = 0.08307
x = 3.25; relative_error = 0.06118
x = 3.5; relative_error = 0.04573
x = 3.75; relative_error = 0.03546
x = 4.0; relative_error = 0.02932
x = 4.25; relative_error = 0.02652
x = 4.5; relative_error = 0.02644
x = 4.75; relative_error = 0.02867
x = 5.0; relative_error = 0.03291
```

```
General purpose quadrature method deviation and calculation error:
x = 1; integral_error = 7.017947987503856e-15; relative_deviation = 0.36788
x = 1.25; integral_error = 3.438695106439127e-09; relative_deviation = 0.38147
x = 1.5; integral_error = 6.827106657780746e-10; relative_deviation = 0.39163
x = 1.75; integral_error = 5.831357619001665e-11; relative_deviation = 0.39957
x = 2.0; integral_error = 6.594659821447934e-15; relative_deviation = 0.40601
x = 2.25; integral_error = 4.199973702156967e-12; relative_deviation = 0.41135
x = 2.5; integral_error = 5.199964141632447e-09; relative_deviation = 0.41588
x = 2.75; integral_error = 3.019555108556365e-09; relative_deviation = 0.41978
x = 3.0; integral_error = 1.2807753055302941e-14; relative_deviation = 0.42319
x = 3.25; integral_error = 1.3611053254449236e-08; relative_deviation = 0.4262
x = 3.5; integral_error = 7.2569228759849354e-09; relative_deviation = 0.42888
x = 3.75; integral_error = 3.1896394752304286e-08; relative_deviation = 0.43129
x = 4.0; integral_error = 3.7738470990419905e-14; relative_deviation = 0.43347
```

```python
# as we can see from our results the smallest errors are provided by
# the composite Simpson's method; the rule is better to use for x values
# especially from 2 to 4 because at that interval the deviation is bounded
# and is decreasing; at the same time for the general purpose quadrature method
# it is permanently increasing and the error in computing the integral is
# not stable; it is also important to mention that the Gaussian quadrature
# method provides us with the bounded deviation on the interval from 1 to 2;
# also, both first and the third methods show smaller ratio of the errors
#  when x is increasing;
```

Problem 4.2:

Solution and output:

Fresnel integrals.py | Own integral code.py | Pi approximation.py

```python
# in this task we will compute the integral which approximates
# the number pi using different numerical integration methods
from math import pi
from scipy import integrate
import timeit


def pi_function(x):
    return 4 / (1 + x ** 2)
```

```python
# let us start from the midpoint rule
# we are working on the interval [a, b]
# h is the step size, n - number of sub-intervals
def midpoint_rule(a, b, n):
    h = (b - a) / n
    integral_value = 0
    for i in range(n):
        integral_value += pi_function((a + h / 2.0) + i * h)

    return integral_value * h


# composite trapezoidal rule
def trapezoidal_rule(a, b, n):
    h = (b - a) / n
    integral_value = 0.5 * (pi_function(a) + pi_function(b))

    for i in range(1, n):
        k = a + i * h
        integral_value += pi_function(k)

    integral_value *= h

    return integral_value
```

```python
# composite Simpson rule
def Simpson_rule(a, b, n):
    h = (b - a) / n

    x_args = list()
    y_args = list()

    # calculating values of x and f(x)
    i = 0
    while i <= n:
        x_args.append(a + i * h)
        y_args.append(pi_function(x_args[i]))
        i += 1
```

```python
    # calculating the value of the integral
    integral_value = 0
    i = 0
    while i <= n:
        if (i == 0) or (i == n):
            integral_value += y_args[i]
        elif i % 2 != 0:
            integral_value += 4 * y_args[i]
        else:
            integral_value += 2 * y_args[i]
        i += 1

    integral_value *= (h / 3)

    return integral_value
```

```python
# and now let's start to study the behaviour of each method
# by measuring the error and the time needed for computations
# we will vary the value of n for each method

start = timeit.default_timer()

print('Midpoint rule:')
N = 2
prev_error = 1
while N <= 2**21:
    experimental_value = midpoint_rule(0, 1, N)
    error = abs(pi - experimental_value)
    if N < 128:
        ratio = round(prev_error / error, 4)
        prev_error = error
        print(f'N = {N}; error = {error}; ratio = {ratio}')
    else:
        print(f'N = {N}; error = {error}')
    N *= 2

end = timeit.default_timer()
print(f'\nTotal time required: {round(end - start, 5)} seconds')
print('\n\n\n')
```

```python
start = timeit.default_timer()

print('Composite trapezoidal rule:')
N = 2
prev_error = 1
while N <= 2**21:
    experimental_value = trapezoidal_rule(0, 1, N)
    error = abs(pi - experimental_value)
    if N < 128:
        ratio = round(prev_error / error, 4)
        prev_error = error
        print(f'N = {N}; error = {error}; ratio = {ratio}')
    else:
        print(f'N = {N}; error = {error}')
    N *= 2

end = timeit.default_timer()
print(f'\nTotal time required: {round(end - start, 5)} seconds')
print('\n\n\n')
```

```python
start = timeit.default_timer()

print('Composite Simpson rule:')
N = 2
while N <= 2**21:
    experimental_value = Simpson_rule(0, 1, N)
    error = abs(pi - experimental_value)
    prev_error = error
    print(f'N = {N}; error = {error}')
    N *= 2

end = timeit.default_timer()
print(f'\nTotal time required: {round(end - start, 5)} seconds')
print('\n\n\n')
```

```python
# we can do the same procedure with the in-built
# Gaussian quadrature routine

start = timeit.default_timer()

print('Gaussian quadrature:')
for j in range(1, 10):
    tolerance = 10 ** (-j)
    experimental_value = integrate.quadrature(lambda x: pi_function(x), 0, 1, tol=tolerance)
    error = abs(pi - experimental_value[0])
    print(f'tolerance = {tolerance}; error = {error}')

end = timeit.default_timer()
print(f'\nTotal time required: {round(end - start, 5)} seconds')
```

Run: Pi approximation

```
C:\Users\User\PycharmProjects\Laboratories\venv\Script
Midpoint rule:
N = 2; error = 0.02076028758667725; ratio = 48.1689
N = 4; error = 0.005207864804149587; ratio = 3.9863
N = 8; error = 0.0013020760018958022; ratio = 3.9997
N = 16; error = 0.0003255207187669029; ratio = 4.0
N = 32; error = 8.138020654424594e-05; ratio = 4.0
N = 64; error = 2.0345052054171475e-05; ratio = 4.0
N = 128; error = 5.086263021425452e-06
N = 256; error = 1.2715657535800062e-06
N = 512; error = 3.1789143761784544e-07
N = 1024; error = 7.947285940446136e-08
N = 2048; error = 1.9868213740892315e-08
N = 4096; error = 4.9670516588662394e-09
N = 8192; error = 1.2417640249395845e-09
N = 16384; error = 3.104312362722794e-10
N = 32768; error = 7.761213893786589e-11
N = 65536; error = 1.9383605831535533e-11
N = 131072; error = 4.8401282981558325e-12
N = 262144; error = 1.2803091919977305e-12
N = 524288; error = 3.1663560662309465e-13
N = 1048576; error = 3.9968028886505635e-15
N = 2097152; error = 1.2878587085651816e-14

Total time required: 2.20122 seconds
```

```
Composite trapezoidal rule:
N = 2; error = 0.04159265358979303; ratio = 24.0427
N = 4; error = 0.010416183001557666; ratio = 3.9931
N = 8; error = 0.0026041590987042618; ratio = 3.9998
N = 16; error = 0.0006510415484042298; ratio = 4.0
N = 32; error = 0.00016276041481821935; ratio = 4.0
N = 64; error = 4.0690104138985106e-05; ratio = 4.0
N = 128; error = 1.0172526041074548e-05
N = 256; error = 2.5431315116009046e-06
N = 512; error = 6.357828716829772e-07
N = 1024; error = 1.5894572102936877e-07
N = 2048; error = 3.9736433699033357e-08
N = 4096; error = 9.934109534981417e-09
N = 8192; error = 2.4835209444518114e-09
N = 16384; error = 6.208913383431991e-10
N = 32768; error = 1.5522472196494164e-10
N = 65536; error = 3.8819614189833374e-11
N = 131072; error = 9.673151168954064e-12
N = 262144; error = 2.3936408410918375e-12
N = 524288; error = 6.235012506294879e-13
N = 1048576; error = 1.270095140171179e-13
N = 2097152; error = 8.570921750106208e-14

Total time required: 2.13879 seconds
```

```
Run:  Pi approximation

Composite Simpson rule:
N = 2; error = 0.008259320256459812
N = 4; error = 2.4026138812693887e-05
N = 8; error = 1.5113108631226169e-07
N = 16; error = 2.3649704417039175e-09
N = 32; error = 3.695710404372221e-11
N = 64; error = 5.782041512247815e-13
N = 128; error = 9.769962616701378e-15
N = 256; error = 4.440892098500626e-16
N = 512; error = 4.440892098500626e-16
N = 1024; error = 1.3322676295501878e-15
N = 2048; error = 1.7763568394002505e-15
N = 4096; error = 0.0
N = 8192; error = 4.440892098500626e-15
N = 16384; error = 7.105427357601002e-15
N = 32768; error = 1.0658141036401503e-14
N = 65536; error = 2.6645352591003757e-15
N = 131072; error = 5.240252676230739e-14
N = 262144; error = 7.638334409421077e-14
N = 524288; error = 3.863576125695545e-14
N = 1048576; error = 2.353672812205332e-14
N = 2097152; error = 3.9968028886505635e-14

Total time required: 4.044 seconds
```

```
Gaussian quadrature:
tolerance = 0.1; error = 0.005948330016764203
tolerance = 0.01; error = 0.0005245136266260886
tolerance = 0.001; error = 1.925165601246448e-05
tolerance = 0.0001; error = 1.3705040213807251e-08
tolerance = 1e-05; error = 4.240220574658338e-08
tolerance = 1e-06; error = 4.240220574658338e-08
tolerance = 1e-07; error = 4.240220574658338e-08
tolerance = 1e-08; error = 4.240220574658338e-08
tolerance = 1e-09; error = 4.240220574658338e-08

Total time required: 0.00234 seconds
```

```python
# in conclusion I would say that analyzing the elapsed time for each method
# we can state that the Gaussian quadrature routine is the most optimal
# we may also notice that the Simpson's method works pretty slow in comparison
# with the others but it is important that at the same time its error converges
# much quicker than the error in the midpoint and trapezoidal rules

# the converges I am talking about means that at a certain point of decreasing h
# the error will stop decreasing and will become approximately constant or even larger

# that's because for midpoint and trapezoidal rules we get the error formulas equal to:
# C/n^p, where 2^p = ratio = 4 and therefore p = 2, h = (b-a)/n and the obtained result is
# C*h^2 (because h = 1/n);
# for the Simpson's rule however things are a little bit different because the ratio
# does not converge to a certain value and therefore it may vary from 150 to 60
# therefore I will write the expression C*h^4 as mentioned in the lecture presentations

# it is obvious now that if we increase the value of h to the infinity the error will
# tend to zero and therefore further increasings will not be necessary
```

Problem 4.3:

Solution and output:

Fresnel integrals.py | Own integral code.py

```python
# in this problem we will work with Fresnel integrals
# which are applied in the light diffraction
# then we will plot the obtained functions
from scipy import special, integrate
from numpy import linspace
import matplotlib.pyplot as plot
import math


# first of all we will compute their real values
# on the interval from zero to five
arguments = linspace(0, 5, 1000)
S, C = special.fresnel(arguments)

print("The numerical results for Fresnel integrals are:")
print(f"scipy.special.fresnel method: C(0:5) = {C[len(C) - 1]}; S(0:5) = {S[len(S) - 1]}")


# plotting the original accurate Fresnel functions
plot.subplot(1, 3, 1)
plot.plot(arguments, C, color='blue', label='C(x)')
plot.plot(arguments, S, color='red', label='S(x)')
plot.title('Fresnel integrals real values', color='violet', fontsize=14)
plot.xlabel('x - axis', color='green', fontsize=12)
plot.ylabel('y - axis', color='green', fontsize=12)
plot.legend()
plot.grid()
```

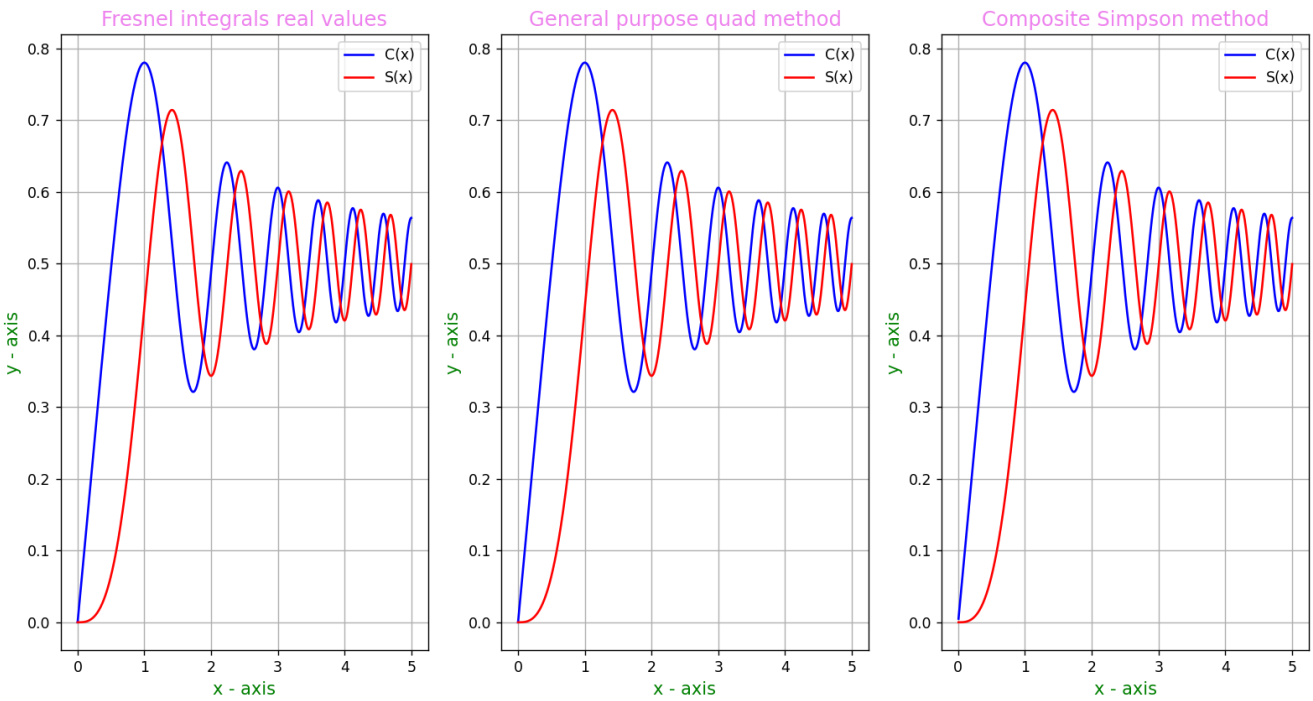
```python
# then we use the general purpose quadrature method
def C(t):
    return math.cos(math.pi * pow(t, 2) / 2)


def S(t):
    return math.sin(math.pi * pow(t, 2) / 2)


C_quad = list()
S_quad = list()
for i in range(0, len(arguments)):
    C1 = integrate.quad(lambda x: C(x), 0, arguments[i])
    S1 = integrate.quad(lambda x: S(x), 0, arguments[i])
    C_quad.append(C1[0])
    S_quad.append(S1[0])


# plotting the obtained functions
plot.subplot(1, 3, 2)
plot.plot(arguments, C_quad, color='blue', label='C(x)')
plot.plot(arguments, S_quad, color='red', label='S(x)')
plot.title('General purpose quad method', color='violet', fontsize=14)
plot.xlabel('x - axis', color='green', fontsize=12)
plot.ylabel('y - axis', color='green', fontsize=12)
plot.legend()
plot.grid()
```

```python
print(f"general purpose quadrature method: C(0:5) = {C_quad[len(C_quad) - 1]}; S(0:5) = {S_quad[len(S_quad) - 1]}")


# finally, we apply composite Simpson's rule
arguments = linspace(0, 5, 1000)
simps_C = list()
simps_S = list()
for i in range(0, len(arguments)):
    x_args = linspace(0, arguments[i], 1000)
    y_args_1 = list()
    y_args_2 = list()

    for j in range(0, len(x_args)):
        y_args_1.append(C(x_args[j]))
        y_args_2.append(S(x_args[j]))

    simps_C.append(integrate.simps(y_args_1, x_args))
    simps_S.append(integrate.simps(y_args_2, x_args))
```

```python
# plotting the results
plot.subplot(1, 3, 3)
plot.plot(arguments, simps_C, color='blue', label='C(x)')
plot.plot(arguments, simps_S, color='red', label='S(x)')
plot.title('Composite Simpson method', color='violet', fontsize=14)
plot.xlabel('x - axis', color='green', fontsize=12)
plot.ylabel('y - axis', color='green', fontsize=12)
plot.legend()
plot.grid()
plot.show()


print(f"composite Simpson's method: C(0:5) = {simps_C[len(simps_C) - 1]}; ", end='')
print(f"S(0:5) = {simps_S[len(simps_S) - 1]};")
```

```
C:\Users\User\PycharmProjects\Laboratories\venv\Scripts\python.exe "C:/Users/User/Pycha
The numerical results for Fresnel integrals are:
scipy.special.fresnel method: C(0:5) = 0.5636311887040122; S(0:5) = 0.4991913819171169
```

```
general purpose quadrature method: C(0:5) = 0.5636311887040117; S(0:5) = 0.49919138191711704
composite Simpson's method: C(0:5) = 0.563631135195887; S(0:5) = 0.4991901114424573;
```



Problem 4.4:

Solution and output:

Own integral code.py

```python
# in this exercise we will not use special python methods
# in order to apply numerical integration of different types
# but instead will use our own computer routine and develop
# some code for Simpson's rule and Gaussian quadrature
from scipy.special import legendre
import numpy as np


# the first rule finds the corresponding quadratic polynomial
# in order to bound the respective region
# a and b are the ends of the interval
# n is the number of its divisions

# it is also important to mention that we are dealing with improper
# integral of type 2 with x = 1 being the vertical asymptote
# as the function tends to infinity
# therefore we will have to integrate not exactly from 1
def function(x):
    return pow(x - 1, -5 / 2)
```

```python
def Simpson_method(f, a, b, n):
    h = (b - a) / n
    x = np.linspace(a, b, n + 1)
    y = f(x)

    riemann_sum = (h / 3) * np.sum(y[0:-1:2] + 4 * y[1::2] + y[2::2])
    return riemann_sum


# showing the results
print('Results for the Simpson method:')
N = 4
for i in range(0, 5):
    print(f'n = {N}; integral value = {Simpson_method(lambda x: function(x), 1 + 1e-8, 4, N)}')
    N *= 2
```

```python
# then we apply the Gaussian quadrature which initially
# works for the interval [-1; 1]
# for that we will need to generate nodes and weights
# of the quadrature using Legendre polynomials of degree n
# their solutions will become nodes and the weights will be
# determined using the special formula
def Legendre_derivative(coefficients, degree, x):
    value = 0
    index = degree
    j = 1
    while index != 0:
        value += coefficients[j - 1] * index * pow(x, degree - j)
        j += 1
        index -= 1

    return value
```

```python
# a and b are the limits of integration
def Legendre_function(a, b, t):
    return pow((b + a + t * (b - a)) / 2 - 1, -5/2)


def Gaussian_quadrature(nod, weight, a, b):
    gaussian_sum = 0
    for index in range(0, len(nod)):
        gaussian_sum += weight[index] * Legendre_function(a, b, nod[index])

    return gaussian_sum * (b - a) / 2
```

```python
# the integral will be calculated by multiplying
# the corresponding nodes and weights consecutively
print()
print()
print()
print('Results for the Gaussian quadrature method:')
N = 4
for i in range(0, 5):
    coefs = list(legendre(N))
    nodes = np.roots(coefs)
    weights = list()
    for k in range(0, len(nodes)):
        weights.append(2 / ((1 - pow(nodes[k], 2)) * (Legendre_derivative(coefs, N, nodes[k])) ** 2))

    print(f'n = {N}; integral value = {Gaussian_quadrature(nodes, weights, 1, 4)}')
    N *= 2
```

```python
# in conclusion I would say that the Gaussian quadrature method
# converges much faster than the Simpson's method in approaching
# to the real value of the integral, but we I also observed that
# the quadrature at first tends to infinity and later starts to
# tend to the real value; however we can state that the Gaussian
# quadrature converges as n goes to infinity
```

```
Run: Own integral code
C:\Users\User\PycharmProjects\Laboratories\venv\S
Results for the Simpson method:
n = 4; integral value = 2.500000029650861e+19
n = 8; integral value = 1.2500000148254304e+19
n = 16; integral value = 6.250000074127152e+18
n = 32; integral value = 3.125000037063576e+18
n = 64; integral value = 1.5625000185317883e+18



Results for the Gaussian quadrature method:
n = 4; integral value = 27.564429010282502
n = 8; integral value = 183.50164887430475
n = 16; integral value = 1336.8142693912134
n = 32; integral value = 9271.89927886087
n = 64; integral value = (0.4305645559152461+0j)
```