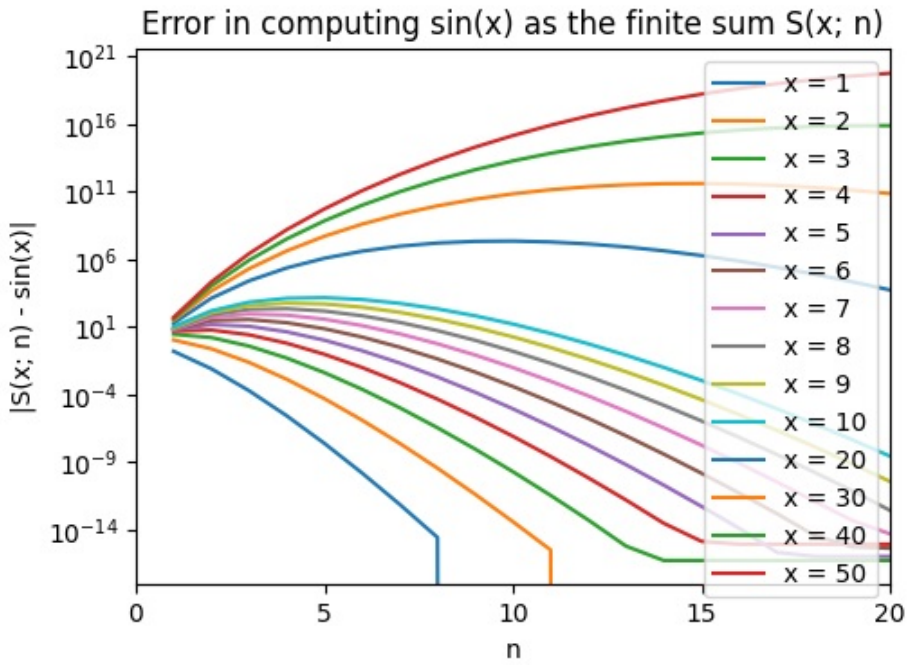
PHYS2160 Introductory Computational Physics
2021/22 Solutions to Exercise 4

1.
```
# sinerror.py
# This program investigates the error of computing the the function sin(x)
# as the finite sum S(x; n).
# Last update on 20 Jan 2021 by F K Chow

from math import sin
import matplotlib.pyplot as plt
import numpy as np

def sinsum1(x, tol):
    """ Compute the finite sum S(x; n) that approximates sin(x) for
        given value of x with relative accuracy tol """
    sum = x
    delta = x
    n = 1
    while abs(delta/sum) > tol:
        delta *= (-1)*x**2/(2*n*(2*n+1))
        sum += delta
        n += 1
    return sum, n

def sinsum2(x, n):
    """ Compute the finite sum S(x; n) that approximates sin(x) for
        given values of x and n """
    sum = x
    delta = x
    for i in np.arange(2, n+1):
        delta *= (-1)*x**2/((2*i-1)*(2*i-2))
        sum += delta
    return sum

# Generate a table of S(x;n), n, and the relative error of computing
# sin x as S(x;n) with relative accuracy tol for different values of x
tol = 1e-7
print("{:>4s} {:>11s} {:>3s} {:>15s}".format("x", "S(x; n)", "n",
                                             "relative error"))
for x in np.linspace(0.1,1.0,10):
    S, n = sinsum1(x, tol)
    relerror = abs((S - sin(x))/sin(x))
    print("{:4.1f} {:11.8f} {:3d} {:15.5e}".format(x, S, n, relerror))

# Plot the absolute error of computing sin x as S(x; n) vs n with log
# scaling on the y-axis for different values of x
fig, ax = plt.subplots()
```

```
xlist = list(range(1,11)) + list(range(20,51,10))
nlist = np.arange(1, 21)
for x in xlist:
    data = []
    for n in nlist:
        S = sinsum2(x, n)
        abserror = abs(S - sin(x))
        data.append(abserror)
    lab = "x = {:d}".format(x)
    ax.semilogy(nlist, data, label=lab)
ax.set_xlim(0, 20)
ax.set_xticks([0, 5, 10, 15, 20])
ax.set_xlabel("n")
ax.set_ylabel("|S(x;n) - sin(x)|")
ax.set_title("Error in computing sin x as the finite sum S(x; n)")
ax.legend()
plt.show()
```

Below is the output of this program:

x	S(x; n)	n	relative error
0.1	0.09983342	4	2.75239e-14
0.2	0.19866933	4	7.09923e-12
0.3	0.29552021	5	1.50086e-13
0.4	0.38941834	5	2.69546e-12
0.5	0.47942554	5	2.54740e-11
0.6	0.56464247	5	1.60595e-10
0.7	0.64421769	6	2.40961e-12
0.8	0.71735609	6	1.22697e-11
0.9	0.78332691	6	5.19105e-11
1.0	0.84147098	6	1.89939e-10



2.
```
# numintegral1.py
# This program evaluates the integral of sin x for 0 <= x <= pi using
# the trapezoidal and Simpson's rule with different number of slices N
# and outputs a table showing the results.
# Last update on 21 Jan 2021 by F K Chow

from math import sin, pi

def trapezoidal(f, a, b, N):
    """ Evaluate the integral of f for a <= x <= b using the
        trapezoidal rule with N slices """
    h = (b - a)/N
    s = 0.5*(f(a) + f(b))
    for k in range(1, N):
        s += f(a+k*h)
    return s*h

def simpsons(f, a, b, N):
    """ Evaluate the integral of f for a <= x <= b using Simpson's
        rule with N slices """
    h = (b - a)/N
    s = f(a) + f(b) + 4*f(a+(N-1)*h)
    for k in range(1, int(N/2)):
        s += 4*f(a+(2*k-1)*h) + 2*f(a+2*k*h)
    return s*h/3.0

# Evaluate the integral using the two methods with different number of
# slices N and output the results
a, b = 0, pi
Ie = 2
print("{:>4s} {:>24s} {:>24s}".format("N", "Trapezoidal (error)",
                                      "Simpson's (error)"))
for N in [4, 8, 16, 32, 64, 128, 256, 512, 1024]:
    It = trapezoidal(sin, a, b, N)
    Is = simpsons(sin, a, b, N)
    errt = abs(Ie - It)
    errs = abs(Ie - Is)
    print("{:4d} {:11.8f} ({:10.8f})".format(N, It, errt), end=" ")
    print("{:11.8f} ({:10.8f})".format(Is, errs))
```

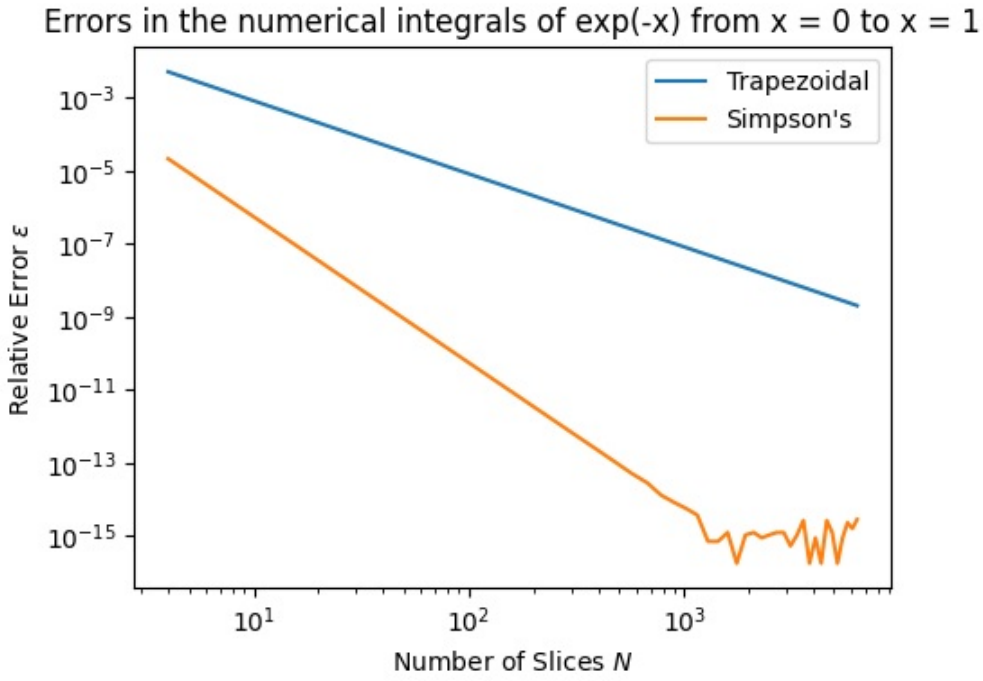
Below is the output of this program:

```
   N      Trapezoidal (error)        Simpson's (error)
   4  1.89611890 (0.10388110)  2.00455975 (0.00455975)
   8  1.97423160 (0.02576840)  2.00026917 (0.00026917)
  16  1.99357034 (0.00642966)  2.00001659 (0.00001659)
  32  1.99839336 (0.00160664)  2.00000103 (0.00000103)
  64  1.99959839 (0.00040161)  2.00000006 (0.00000006)
 128  1.99989960 (0.00010040)  2.00000000 (0.00000000)
 256  1.99997490 (0.00002510)  2.00000000 (0.00000000)
 512  1.99999373 (0.00000627)  2.00000000 (0.00000000)
1024  1.99999843 (0.00000157)  2.00000000 (0.00000000)
```

3.
```
# numintegral2.py
# This program evaluates the integral of exp(-x) for 0 <= x <= 1 using
# the trapezoidal and Simpson's rule with different number of slices N
# and make a log-log plot of the relative error of the integral versus
# the number of slices N in each case.
# Last update on 21 Jan 2021 by F K Chow

from math import exp
import matplotlib.pyplot as plt

def f(x):
    """ Compute the function f(x) = exp(-x) """
    return exp(-x)

def trapezoidal(f, a, b, N):
    """ Evaluate the integral of f for a <= x <= b using the
        trapezoidal rule with N slices """
    h = (b - a)/N
    s = 0.5*(f(a) + f(b))
    for k in range(1, N):
        s += f(a+k*h)
    return s*h

def simpsons(f, a, b, N):
    """ Evaluate the integral of f for a <= x <= b using Simpson's
        with N slices """
    h = (b - a)/N
    s = f(a) + f(b) + 4*f(a+(N-1)*h)
    for k in range(1, int(N/2)):
        s += 4*f(a+(2*k-1)*h) + 2*f(a+2*k*h)
    return s*h/3.0

# Evaluate the integral using the two methods with different N
a, b = 0, 1
Ie = 1.0 - exp(-1.0)
Nlist = [i**2 for i in range(2, 82, 2)]
relerrort = []
relerrors = []
for N in Nlist:
    It = trapezoidal(f, a, b, N)
    Is = simpsons(f, a, b, N)
    relerrort.append(abs(Ie - It)/Ie)
    relerrors.append(abs(Ie - Is)/Ie)

# Make a log-log plot of the relative error of the integrals vs N
fig, ax = plt.subplots()
ax.loglog(Nlist, relerrort, label="Trapezoidal")
ax.loglog(Nlist, relerrors, label="Simpson's")
```

```
ax.set_xlabel(r"Number of Slices $N$")
ax.set_ylabel(r"Relative Error $\epsilon$")
ax.set_title("Errors in the numerical integrals of exp(-x) from x = \
0 to x = 1")
ax.legend()
plt.show()
```

Below is the output of this program:



4.
```
# numintegral3.py
# This program uses the adaptive Simpson's rule to compute the value of
# the integral of sin^2(sqrt(100x)) for 0 <= x <= 1 to an accuracy of
# 1e-8 with an initial number of slices of 10 and then output the value
# of the integral and the number of slices used to get the result.
# Last update on 22 Mar 2021 by F K Chow

from math import sin, sqrt

def f(x):
    """ Compute the function f(x) = sin^2(sqrt(100x)) """
    return sin(sqrt(100*x))**2

def adpSimpsons(f, a, b, N, tol):
    """ Evaluate the integral of f for a <= x <= b using the adaptive
        Simpson's rule with an accuracy of tol and an initial number
        of slices of N """
    error = 1
    h = (b - a)/N
    So = f(a) + f(b)
```

```
        To = f(a+h)
        for k in range(2, N, 2):
            So += 2*f(a+k*h)
            To += f(a+(k+1)*h)
        So /= 3.0
        To *= 2.0/3.0
        Io = h*(So + 2*To)
        while abs(error) > tol:
            N *= 2
            h *= 0.5
            Sc = So + To
            Tc = 0
            for k in range(1, N, 2):
                Tc += f(a+k*h)
            Tc *= 2.0/3.0
            Ic = h*(Sc + 2*Tc)
            error = (Ic - Io)/15.0
            So, To, Io = Sc, Tc, Ic
        return Ic, N

    # Use the adaptive Simpson's rule to compute the integral and output
    # the result
    a, b = 0, 1
    tol = 1e-8
    N = 10
    I, N = adpSimpsons(f, a, b, N, tol)
    print("Using the adaptive Simpson's rule to compute the integral of")
    print("sin^2(sqrt(100x)) for 0 <= x <= 1 to an accuracy of 1e-8 with")
    print("an initial number of slices of 10, we find that:")
    print(" value of the integral = {:.8f},".format(I), end="")
    print(" number of slices used =", N)
```

Below is the output of this program:

```
Using the adaptive Simpson's rule to compute the integral of
sin^2(sqrt(100x)) for 0 <= x <= 1 to an accuracy of 1e-8 with
an initial number of slices of 10, we find that:
 value of the integral = 0.45583252, number of slices used = 640
```

5.
```
# numintegral4.py
# This program uses the Romberg integration to compute the value of
# the integral of sin^2(sqrt(100x)) for 0 <= x <= 1 to an accuracy of
# 1e-8 with an initial number of slices of 10 and then output the
# Romberg estimates of the integral.
# Last update on 7 Jan 2022 by F K Chow

from math import sin, sqrt
```

```
def f(x):
    """ Compute the function f(x) = sin^2(sqrt(100x)) """
    return sin(sqrt(100*x))**2

def trapezoidal(f, a, b, N):
    """ Evaluate the integral of f for a <= x <= b using the
        trapezoidal rule with N slices """
    h = (b - a)/N
    s = 0.5*(f(a) + f(b))
    for k in range(1, N):
        s += f(a+k*h)
    return s*h

def romberg(f, a, b, N, tol):
    """ Evaluate the integral of f for a <= x <= b using the Romberg
        integration with an accuracy of tol and an initial number of
        slices of N """
    R = [[trapezoidal(f, a, b, N)]] # Note that R[i-1][m-1] = R_{i, m}
    i = 2
    flag = 0
    while (flag == 0):
        R.append([0 for j in range(1,i+1)])
        N *= 2
        for m in range(i-1):
            if m == 0:
                R[i-1][m] = trapezoidal(f, a, b, N)
            else:
                R[i-1][m] = R[i-1][m-1] + (R[i-1][m-1] -
                                           R[i-2][m-1])/(4**m - 1)
            error = abs((R[i-1][m] - R[i-2][m])/(4**(m+1) - 1))
            if error <= tol:
                flag = 1
                break
        if flag != 1:
            R[i-1][i-1] = R[i-1][i-2] + (R[i-1][i-2] -
                                         R[i-2][i-2])/(4**(i-1) - 1)
            i += 1
    return R

# Evaluate the Romberg estimates of the integral and output the result
a, b = 0, 1
N = 10
tol = 1e-8
R = romberg(f, a, b, N, tol)
m = len(R)
print("Using the Romberg integration to compute the integral of")
print("sin^2(sqrt(100x)) for 0 <= x <= 1 to an accuracy of 1e-8")
```

```
print("with an initial number of slices of 10, we find that the")
print("Romberg estimates of the integral are:")
print("I1 = {:10.8f}".format(R[0][0]))
for i in range(1,m):
    print("{:>15s} {:>1s}".format("", "\u2198"), end="")
    for j in range(i-1):
        print("{:>13s}".format("\u2198"), end="")
    print()
    print("I{0:1d} = {1:10.8f}".format(i+1, R[i][0]), end="")
    for j in range(1, i+1):
        if R[i][j] != 0:
            print("{:>2s} {:10.8f}".format("\u2192", R[i][j]), end="")
        else:
            print("{:>2s} {:10s}".format("\u2192", "-"*10), end="")
    print()
```

Below is the output of this program:

```
Using the Romberg integration to compute the integral of
sin^2(sqrt(100x)) for 0 <= x <= 1 to an accuracy of 1e-8
with an initial number of slices of 10, we find that the
Romberg estimates of the integral are:
I1 = 0.40818652
                ↘
I2 = 0.43818325 → 0.44818217
                ↘           ↘
I3 = 0.45100552 → 0.45527961 → 0.45575278
                ↘           ↘           ↘
I4 = 0.45459892 → 0.45579672 → 0.45583120 → 0.45583244
                ↘           ↘           ↘           ↘
I5 = 0.45552244 → 0.45583027 → 0.45583251 → 0.45583253 → ----------
```

6.
```
# planckslaw.py
# This program evalautes the integral in Planck's law of thermal
# radiation using Simpson's rule with 100 slices and then uses this
# integral to compute the value of the Stefan-Boltzmann constant.
# Last update on 21 Jan 2021 by F K Chow

from math import exp, isclose, pi
from scipy.constants import c, h, k, sigma

def f(z):
    """ Compute the function f(x) = x^3/(exp(x) - 1) with the
        change of variable z = x/(1 + x) """
    if isclose(z, 0) or isclose(z, 1):
        return 0
    x = z/(1 - z)
    return (x**3/(exp(x) - 1))/(1 - z)**2

def simpsons(f, a, b, N):
    """ Evaluate the integral of f for a <= x <= b using Simpson's
```

```
        rule with N slices """
    h = (b - a)/N
    s = f(a) + f(b) + 4*f(a+(N-1)*h)
    for k in range(1, int(N/2)):
        s += 4*f(a+(2*k-1)*h) + 2*f(a+2*k*h)
    return s*h/3.0

# Evaluate the integral in Planck's law of thermal radiation using
# Simpson's rule and then uses this integral to compute the value of
# of the Stefan-Boltzmann constant
a, b = 0, 1
N = 100 # Too big N will lead to overflow due to exp(x) term in f!
I = simpsons(f, a, b, N)
sigmaI = 2*pi*k**4*I/(c**2*h**3)

# Output the result with its theoretical value
print("Using the value of the integral in Planck's law of thermal \
radiation")
print("computed by Simpson's rule with 100 slices, we find that:")
print(" Stefan-Boltzmann constant = {:.8e},".format(sigmaI))
print(" its theoretical value = {:.8e}.".format(sigma))
```

Below is the output of this program:

```
Using the value of the integral in Planck's law of thermal radiation
computed by Simpson's rule with 100 slices, we find that:
 Stefan-Boltzmann constant = 5.67037524e-08,
 its theoretical value = 5.67037442e-08.
```

7.
```
# firstderiv.py
# This program computes the first-order derivative of the function f(x)
# = cos(x) at 9 uniformly spaced points for 0 <= x <= pi using different
# methods based only on the values of f(x) at 11 uniformly spaced points
# for -pi/8 <= x <= 9pi/8. Then it outputs a table of each derivative
# with its error for different values of x.
# Last update on 21 Mar 2022 by F K Chow

from math import cos, sin, pi
import numpy as np

def fdiff(f, x, h):
    """ Compute the first-order derivative of f(x) using the forward
        difference with step size h """
    return (f(x + h) - f(x))/h

def bdiff(f, x, h):
    """ Compute the first-order derivative of f(x) using the backward
        difference with step size h """
    return (f(x) - f(x - h))/h
```

```
def cdiff(f, x, h):
    """ Compute the first-order derivative of f(x) using the central
        difference with step size h """
    return 0.5*(f(x + h) - f(x - h))/h

# Compute the first-order derivative of f(x) = cos(x) using different
# methods and output a table of each derivative with its error for
# different values of x
a, b = 0, 1
N = 9
h = (b-a)*pi/(N-1)
print("[First-order derivative of f(x) = cos(x) computed by different \
methods]")
print("{:>7s} {:>21s} {:>22s}".format("x", "Forward diff (error)",
                                      "Backward diff (error)"), end="")
print("{:>21s}".format("Central diff (error)"))
for i in np.linspace(a, b, N):
    x = i*pi
    fpf = fdiff(cos, x, h)
    fpb = bdiff(cos, x, h)
    fpc = cdiff(cos, x, h)
    efpf = abs(fpf + sin(x))
    efpb = abs(fpb + sin(x))
    efpc = abs(fpc + sin(x))
    print("{:5.3f}pi {:11.5f} ({:.5f})".format(i, fpf, efpf), end=" ")
    print("{:12.5f} ({:.5f}) {:11.5f} ({:.5f})".format(fpb, efpb,
                                                       fpc, efpc))
```

Below is the output of this program:

```
[First-order derivative of f(x) = cos(x) computed by different methods]
      x  Forward diff (error)  Backward diff (error)  Central diff (error)
0.000pi    -0.19384 (0.19384)      0.19384 (0.19384)     0.00000 (0.00000)
0.125pi    -0.55201 (0.16932)     -0.19384 (0.18884)    -0.37292 (0.00976)
0.250pi    -0.82614 (0.11903)     -0.55201 (0.15510)    -0.68907 (0.01803)
0.375pi    -0.97450 (0.05062)     -0.82614 (0.09774)    -0.90032 (0.02356)
0.500pi    -0.97450 (0.02550)     -0.97450 (0.02550)    -0.97450 (0.02550)
0.625pi    -0.82614 (0.09774)     -0.97450 (0.05062)    -0.90032 (0.02356)
0.750pi    -0.55201 (0.15510)     -0.82614 (0.11903)    -0.68907 (0.01803)
0.875pi    -0.19384 (0.18884)     -0.55201 (0.16932)    -0.37292 (0.00976)
1.000pi     0.19384 (0.19384)     -0.19384 (0.19384)    -0.00000 (0.00000)
```

8.
```
# secondderiv.py
# This program computes the second-order derivative of the function
# f(x) = cos(x)*sinh(x) at 51 uniformly spaced points for 0 <= x <=
# pi/2 using the central difference based only on the values of f(x)
# at 53 uniformly spaced points for -pi/100 <= x <= 51pi/100. It then
# outputs a table of the derivative with its error for different
# values of x.
# Last update on 21 Jan 2021 by F K Chow
```

```
from math import cos, sin, cosh, sinh, pi
import numpy as np

def f(x):
    """ Compute the function f(x) = cos(x)*sinh(x) """
    return cos(x)*sinh(x)

def cdiff(f, x, h):
    """ Compute the second-order derivative of f(x) using the central
        difference with step size h """
    return (f(x + h) - 2*f(x) + f(x - h))/h**2

# Compute the second-order derivative of f(x) = cos(x) sinh(x) using
# the central difference and output a table of the derivative with its
# error for different values of x
a, b = 0, 0.5
N = 51
h = (b-a)*pi/(N-1)
print("[Second-order derivative of f(x) = cos(x)*sinh(x)")
print("computed by the central difference]")
print("{:>6s} {:>12s} {:>12s}".format("x", "f"(x)", "Error"))
for i in np.linspace(a, b, N):
    x = i*pi
    fppc = cdiff(f, x, h)
    efppc = abs(fppc + 2*sin(x)*cosh(x))
    print("{:.2f}pi {:12.8f} {:12.8f}".format(i, fppc, efppc))
```

Below is the output of this program:

```
[Second-order derivative of f(x) = cos(x)*sinh(x)
computed by the central difference]
     x       f''(x)        Error
0.00pi   0.00000000   0.00000000
0.01pi  -0.06286285   0.00001033
0.02pi  -0.12584965   0.00002064
0.03pi  -0.18908409   0.00003091
0.04pi  -0.25268938   0.00004112
0.05pi  -0.31678799   0.00005125
0.06pi  -0.38150140   0.00006127
0.07pi  -0.44694988   0.00007117
0.08pi  -0.51325217   0.00008093
0.09pi  -0.58052532   0.00009051
0.10pi  -0.64888433   0.00009991
0.11pi  -0.71844196   0.00010910
0.12pi  -0.78930845   0.00011806
0.13pi  -0.86159123   0.00012676
0.14pi  -0.93539465   0.00013518
0.15pi  -1.01081971   0.00014329
0.16pi  -1.08796375   0.00015108
0.17pi  -1.16692018   0.00015851
0.18pi  -1.24777819   0.00016557
0.19pi  -1.33062238   0.00017222
0.20pi  -1.41553251   0.00017844
0.21pi  -1.50258317   0.00018420
0.22pi  -1.59184341   0.00018947
0.23pi  -1.68337642   0.00019422
```

```
0.24pi  -1.77723920   0.00019843
0.25pi  -1.87348220   0.00020206
0.26pi  -1.97214891   0.00020508
0.27pi  -2.07327554   0.00020746
0.28pi  -2.17689063   0.00020917
0.29pi  -2.28301461   0.00021017
0.30pi  -2.39165943   0.00021043
0.31pi  -2.50282817   0.00020991
0.32pi  -2.61651456   0.00020858
0.33pi  -2.73270259   0.00020640
0.34pi  -2.85136605   0.00020334
0.35pi  -2.97246809   0.00019935
0.36pi  -3.09596073   0.00019439
0.37pi  -3.22178442   0.00018843
0.38pi  -3.34986755   0.00018142
0.39pi  -3.48012593   0.00017332
0.40pi  -3.61246236   0.00016409
0.41pi  -3.74676603   0.00015369
0.42pi  -3.88291209   0.00014208
0.43pi  -4.02076107   0.00012920
0.44pi  -4.16015838   0.00011502
0.45pi  -4.30093375   0.00009949
0.46pi  -4.44290072   0.00008256
0.47pi  -4.58585605   0.00006418
0.48pi  -4.72957920   0.00004432
0.49pi  -4.87383177   0.00002293
0.50pi  -5.01835690   0.00000005
```

9.
```
# interpolate.py
# This program performs linear interpolation on the given data of
# f(x) = exp(sin(x)) at several points to find its values at x = 0.1,
# 0.2, 0.3, ..., 4.9 and then plot the interpolated values and exact
# values of f(x) as well as the data for interpolation versus x for
# 0 <= x <= 5 on the same graph.
# Last update on 21 Jan 2021 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

def f(x):
    """ Compute the function f(x) = exp(sin(x)) """
    return np.exp(np.sin(x))

def interpolate(fa, fb, a, b, x):
    """ Compute the interpolated value of f(x) at x from its value at
        x = a and x = b """
    return ((b-x)*fa + (x-a)*fb)/(b-a)

# Given data for performing interpolation
xd = [0, 0.36, 1.22, 2.15, 2.78, 3.62, 4.27, 5]
fxd = [1, 1.42229, 2.55768, 2.30919, 1.42442, 0.63105, 0.40505,
       0.38331]

# Perform linear interpolation on the given data to find the values of
```

```
# f(x) at x = 0.1, 0.2, 0.3, ..., 4.9
xi = np.arange(0, 5.1, 0.1)
fxi = [fxd[0]]
i = 0
for x in xi[1:-1]:
    if x == xd[i+1]:
        fxi.append(fxd[i+1])
    else:
        while x > xd[i+1]:
            i += 1
        fxi.append(interpolate(fxd[i], fxd[i+1], xd[i], xd[i+1], x))
fxi.append(fxd[-1])

# Plot the interpolated values and the exact values of f(x) as well as
# the data for interpolation vs x for 0 <= x <= 5 on the same graph
fig, ax = plt.subplots()
ax.plot(xi, f(xi), "b-", label="Exact values")
ax.plot(xd, fxd, "kx", label="Data for interpolation")
ax.plot(xi, fxi, "r-", label="Interpolated values")
ax.set_xlim(-0.5, 5.5)
ax.set_xlabel("x")
ax.set_ylabel("f(x)")
ax.set_title("Linear Interpolation of f(x) = exp(sin(x))")
ax.legend()
plt.show()
```

Below is the output of this program: