Jamison Lahman (lahmanja@gmail.com)
Computational Physics: FORTRAN Version
by Steven E. Koonin and Dawn C. Meredith
Chapter 01 Exercises — July 7th, 2018

Exercise 1.1: Using any function for which you can evaluate the derivatives analytically, investigate the accuracy of the formulas in Table 1[1] for various values of h.

Summary: To approximate the derivative numerically, we will utilize five equations based off the Maclaurin Series. The Maclaurin Series of function, *f(x)*, is given by the equation,

$$f(x) = f_0 + xf_0' + \frac{x^2 f_0''}{2!} + \frac{x^n f_0^n}{n!}. \tag{1}$$

Solving the above equation for *f'* yields,

$$f_0' = \frac{f(x) - f_0}{x} + \frac{-1}{x}\left(\frac{x^2 f_0''}{2!}\right) + \frac{-1}{x}\left(\frac{x^n f_0^{(n)}}{n!}\right) = \frac{f(x) - f_0}{x} + \mathcal{O}(x). \tag{2}$$

The above equation assumes the function is accurately described by a linear function over the interval $(0, h)$. This gives the first of the "2-point" methods, the forward 2-point method,

$$f'(x) \approx \frac{f(x_{k+1}) - f(x_k)}{h}, \tag{3}$$

where x is the value where the derivative is being evaluated and h is the step size used. Using the same method as above but using a previous point produces the equation,

$$f'(x) \approx \frac{f(x_k) - f(x_{k-1})}{h}. \tag{4}$$

This assumes the function is accurately described by a linear function over the interval $(-h, 0)$. Notice each of these equations are accurate up to one order of h. The following are python functions utilizing each respective 2-point method,

```python
def forward2(x,h):
#Performs the forward 2-point method on the function defined by myFunc
#Input:  x -- independent variable
#        h -- step-size
#Output: ans -- dependent variable
    ans = (myFunc(x)-myFunc(x-h))/h
    return(ans)

def backward2(x,h):
#Performs the backward 2-point method on the function defined by myFunc
#Input:  x -- independent variable
#        h -- step-size
#Output: ans -- dependent variable
    ans = (myFunc(x+h)-myFunc(x))/h
    return(ans)
```

We can improve the accuracy by including more terms. Using the immediate values forwards and backwards allows for the terms with even-powered values of h to cancel out. Eq 5 is the quadratic polynomial interpolation of the function over the two previous regions,

$$f' = \frac{f(x_{k+1}) - f(x_{k-1})}{2h} + \mathcal{O}(h^2). \tag{5}$$

Python interpration of Eq.5,

[1]Table 1.2 from Computational Physics: FORTRAN Version

```python
def symmetric3(x,h):
#Performs the symmetric 3-point method on the function defined by myFunc
#Input:  x -- independent variable
#        h -- step-size
#Output: ans -- dependent variable
    ans = (myFunc(x+h)-myFunc(x-h))/(2*h)
    return(ans)
```

The final two equations, the ”4-point” and ”5-point” formulas, are similarly found and include more interactions fowards and backwards to create higher-order approximations of the function over the given interval. They, along with their higher-order derivatives, are given in Table 1. The 5-point formula is the five-point stencil in one-dimension.

Table 1: 4- and 5-point difference formulas for derivatives

	4-point	5-point
hf'	$\pm\frac{1}{6}(-2f_{\mp1} - 3f_0 + 6f_{\pm1} - f_{\pm2})$	$\frac{1}{12}(f_{-2} - 8f_{-1} + 8f_1 - f_2)$
h^2f''	$f_{-1} - 2f_0 + f_1$	$\frac{1}{12}(-f_{-2} + 16f_{-1} - 30f_0 + 16f_1 - f_2)$
h^3f'''	$\pm(-f_{\mp1} + 3f_0 - 3f_{\pm1} + f_{\pm2})$	$\frac{1}{2}(-f_{-2} + 2f_{-1} - 2f_1 + f_2)$
$h^4f^{iv)}$	...	$f_{-2} - 4f_{-1} + 6f_0 - 4f_1 + f_2$

Python interpretation of the previous equations,

```python
def symmetric4(x,h):
#Performs the symmetric 4-point method on the function defined by myFunc
#Input:  x -- independent variable
#        h -- step-size
#Output: ans -- dependent variable
    ans = (-2*myFunc(x-h)-3*myFunc(x)+6*myFunc(x+h)-myFunc(x+2*h))/(6*h)
    return(ans)

def symmetric5(x,h):
#Performs the symmetric 5-point method on the function defined by myFunc
#Input:  x -- independent variable
#        h -- step-size
#Output: ans -- dependent variable
    ans = (myFunc(x-2*h)-8*myFunc(x-h)+8*myFunc(x+h)-myFunc(x+2*h))/(12*h)
    return(ans)
```

Solution:

The function we will approximating the derivative numerically is,

$$F(x) = \sin(x).$$

The analytical derivative of the sine function is known to be,

$$\frac{d\sin(x)}{dx} = \cos(x).$$

For a formal proof of the derivation, see Wikipedia - derivative of the sine function. The exact value of the derivative at $x = 1$ (up to six decimal points) is 0.540302. Table 2 shows the accuracy of the formulas above for various values of h.[2]

[2]The full python script responsible for Table 2, see my GitHub.

Table 2: Error in evaluating the $d\sin/dx|_{x=1} = 0.540302$

h	Fwd 2-pnt Eq. 3	Bkwd 2-pnt Eq. 4	3-pnt Eq. 5	4-pnt Tab. 1	5-pnt Tab. 1
0.50000	-0.228254	0.183789	-0.022233	-0.009499	-0.001093
0.20000	-0.087462	0.080272	-0.003595	-0.000586	-0.000029
0.10000	-0.042939	0.041138	-0.000900	-0.000072	-0.000002
0.05000	-0.021257	0.020807	-0.000225	-0.000009	-0.000000
0.02000	-0.008450	0.008378	-0.000036	-0.000001	-0.000000
0.01000	-0.004216	0.004198	-0.000009	-0.000000	-0.000000
0.00500	-0.002106	0.002101	-0.000002	-0.000000	-0.000000
0.00200	-0.000842	0.000841	-0.000000	-0.000000	-0.000000
0.00100	-0.000421	0.000421	-0.000000	-0.000000	-0.000000
0.00050	-0.000210	0.000210	-0.000000	-0.000000	-0.000000
0.00020	-0.000084	0.000084	-0.000000	-0.000000	-0.000000
0.00010	-0.000042	0.000042	-0.000000	0.000000	0.000000
0.00005	-0.000021	0.000021	-0.000000	0.000000	0.000000
0.00002	-0.000008	0.000008	-0.000000	-0.000000	0.000000
0.00001	-0.000004	0.000004	-0.000000	-0.000000	-0.000000

Exercise 1.2: Using any function whose definite integral you can compute analytically, investigate the accuracy of various quadrature methods for various values of h.

Summary: To approximate the definite integral of a function over the interval $[a,b]$, we will employ similar methods as used previously for exercise 1.1. A simple approximation is to assume the function is linear over the entire interval (see the star-dashes in Fig.1). Further accuracy can be found by splitting the interval into N number of smaller lattices which can then be approximated linearly,

$$N = \frac{b-a}{h},$$

where h is the width of the lattice (see the dashed lines in Fig.1). The integral is then,

$$\int_a^b f(x)dx = \sum_{k=1}^{N} \frac{h}{2}[f(x_{k-1}) + f(x_k)] + \mathcal{O}(h^3) \approx T_N. \tag{6}$$

The Python interpretation of the Trapezoid method is,

```python
def trapezoidal(x,h,N):
#Performs the trapezoidal rule (equation 1.9). The average of the function evaluated at
#the end points of the lattice multiplied by the change in the independent variable.
#Input:  x -- lower bound of the range of integration
#        h -- step-size
#        N -- number of lattices
#Output: ans -- approximate integral
    sum = 0
    for i in range(N):
        sum = sum + (myFunction(x+i*h)+myFunction(x+(i+1)*h)) * h / 2.0
    return sum
```

The behavior of the error for the trapezoid method is very much dependent on the behavior of the function over the given interval. When the function is concave up (such as the the function over the interval $[a{+}h,b]$ in Fig.1), the trapezoid method will over-estimate the true value. Similarly, when the function is concave down, (such as the function over the interval $[a,a{+}h]$ in Fig.1), the trapezoid method will under-estimate the true value.

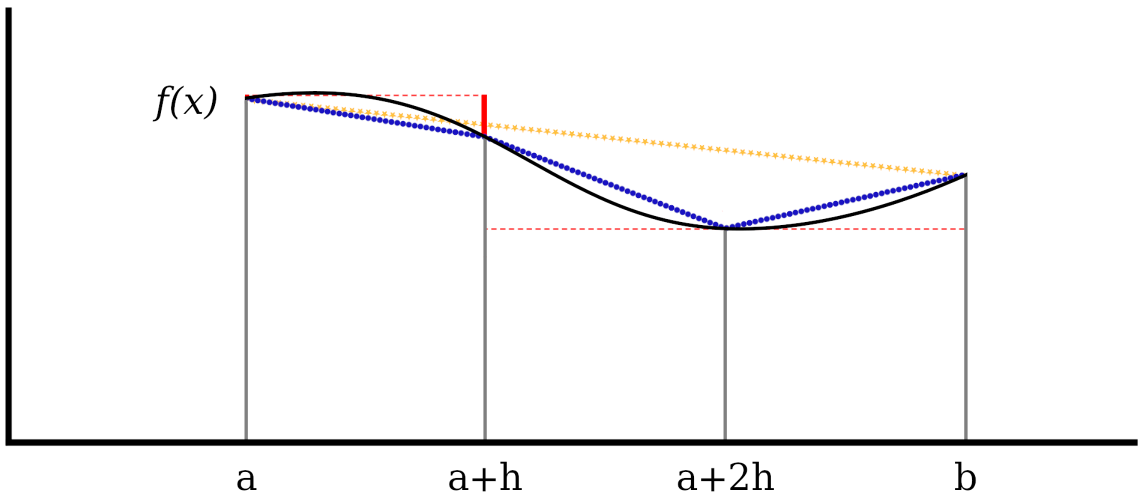

Figure 1: The yellow, star-dashed line show shows a linear interpolation of the function over the full domain. The blue, circle-dashed line shows a linear interpolation of the function with lattices of width h. The red, dashed line shows the midpoint method. It is easy to see the trapezoid method over-estimates the integral over the interval $[a{+}h, b]$ whereas the midpoint method under-estimates.

Conversely, the midpoint method,

$$\int_a^b f(x)dx = \sum_{k=1}^{N} h \cdot f\left(\frac{x_{k-1}+x_k}{2}\right) + \mathcal{O}(h^3) \approx M_N, \tag{7}$$

will consistently approximate in the integral in the opposite direction from the trapezoid method. Given they both error in opposite directions, they can be combined in a manner that reduces the overall error. To be exact, the errors EM_N and ET_N for a quadratic function are related by,

$$|EM_N| = \frac{1}{2}|ET_N|$$

and can be easily combined in a way to completely eliminate this error.[3] This equation is known as the Simpson's Rule,

$$S_{2N} = \frac{1}{3}(2M_N + M_N).$$

Note, in the equation above, S is given the subscript $_{2N}$ since the midpoint is evaluated sub interval points from the trapezoid method thus the number of intervals is twice that for either method. For evenly spaced lattices over the interval [a,b], Simpson's Rule becomes,

$$S_N = \frac{h}{3}[f(x_k) + 4f(x_{k+1}) + 2f(x_{k+2}) + ... + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)] + \mathcal{O}(h^5). \tag{8}$$

The Python interpretation of Simpson's Rule is,

```python
def simpsons(x,h,N):
#Performs Simpons rule (equation 1.12).
#Input:  x -- lower bound of the range of integration
#        h -- step-size
#        N -- number of lattices
#Output: sum -- approximate integral
#Add the contribution from the first and last points of the domain
```

[3]Taken from Introduction to Numerical Methods and MATLAB Programming for Engineers by Todd Young and Martin J. Mohlenkamp

```
    sum = myFunction(x) + myFunction(x+N*h)
    for i in range(N-1):                        #N-1 ignores last point
        j = i+1                                 #i+1 ignores first point
#Adds the contribution from the even placed lattice points
        if (j % 2 == 1):
            sum = sum + 4.0*myFunction(x+j*h)
#Adds the contribution from the odd placed lattice points
        else:
            sum = sum + 2.0*myFunction(x+j*h)
    sum = sum*h/3.0                             #Apply leading factor
    return sum
```

Simpson's Rule is the quadratic interpolation of the function. Similar to numerical differentiation, further accuracy can be achieved by interpolating with higher-order polynomials. The Simpson's 3/8 Rule is the cubic interpolation and is given by,

$$\int_a^b f(x)dx = \frac{3h}{8}[f(x_k) + 3\sum_{i\neq 3j}^{N-1} f(x_i) + 2\sum_{j=1}^{N/3-1} (f(x_{3j}) + f(x_n)] + \mathcal{O}(h^5). \tag{9}$$

Notice the Simpson's 3/8 Rule accurate to the same order of h as Eq.8 however the constant term is slightly smaller. The quartic interpolation is called Boole's Rule and is given by,

$$\int_a^b f(x)dx = \frac{2h}{45}[7f(x_k) + 32f(x_{k+1}) + 12f(x_{k+2}) + 32f(x_{n-1}) + 7f(x_n)] + \mathcal{O}(h^7) \tag{10}$$

```
def booles(x,h,N):
#Performs Bode's rule (equation 1.13b)
#Input:  x -- lower bound of the range of integration
#        h -- step-size
#        N -- number of lattices
#Output: sum -- approximate integral
#Add the contribution from the first and last points of the domain
    sum = 7.0*(myFunction(x) + myFunction(x+N*h))
    for i in range(N-1):                        #N-1 ignores last point
        j = i+1                                 #i+1 ignores first point
#Adds the contribution from the even placed lattice points
        if (j % 2 == 1):
            sum = sum + 32.0*myFunction(x+j*h)
        elif(j % 4 == 2):
            sum = sum + 12.0*myFunction(x+j*h)
#Adds the contribution from the odd placed lattice points
        else:
            sum = sum + 14.0*myFunction(x+j*h)
    sum = sum * 2.0 * h / 45.0                  #Apply leading factor
    return sum
```

Using the functions from above, we are able to produce the values for Table 3. As we expected, Boole's Rule gives the greatest accuracy with largest value of h followed by Simpson's Rule then the Trapezoidal Method.[4]

Table 3: Errors in evaluating $\int_0^1 e^x dx = 1.718282$

N	h	Trapezoidal Eq.6	Simpson's Eq.8	Boole's Eq10
4.000000	0.25000	0.008940	0.000037	0.000001
8.000000	0.12500	0.002237	0.000002	0.000000
16.000000	0.06250	0.000559	0.000000	0.000000
32.000000	0.03125	0.000140	0.000000	0.000000
64.000000	0.01562	0.000035	0.000000	0.000000
128.000000	0.00781	0.000009	0.000000	-0.000000

[4]The full python script responsible for Table 2, see my GitHub.