

Exercise 1.1: Using any function for which you can evaluate the derivatives analytically, investigate the accuracy of the formulas in Table 1¹ 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,

$$f(x) = f_0 + xf'_0 + \frac{x^2 f''_0}{2!} + \frac{x^n f^{(n)}_0}{n!}. \quad (1)$$

Solving the above equation for f' yields,

$$f'_0 = \frac{f(x) - f_0}{x} + \frac{-1}{x} \left(\frac{x^2 f''_0}{2!} \right) + \frac{-1}{x} \left(\frac{x^n f^{(n)}_0}{n!} \right) = \frac{f(x) - f_0}{x} + \mathcal{O}(x). \quad (2)$$

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,

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \quad (3)$$

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,

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}. \quad (4)$$

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,

```

1 def forward2(x,h):
2     #Performs the forward 2-point method on the function defined by myFunc
3     #Input:  x -- independent variable
4     #        h -- step-size
5     #Output: ans -- dependent variable
6     ans = (myFunc(x)-myFunc(x-h))/h
7     return(ans)
8
9 def backward2(x,h):
10    #Performs the backward 2-point method on the function defined by myFunc
11    #Input:  x -- independent variable
12    #        h -- step-size
13    #Output: ans -- dependent variable
14    ans = (myFunc(x)-myFunc(x-h))/h
15    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,

$$f' = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2). \quad (5)$$

Python interpretation of Eq.5,

¹Table 1.2 from Computational Physics: FORTRAN Version

```

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

```

The final two equations, the "4-point" and "5-point" formulas, are similarly found and include more interactions forwards 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'	$\pm \frac{1}{6}(-2f_{\mp 1} - 3f_0 + 6f_{\pm 1} - f_{\pm 2})$	$\frac{1}{12}(f_{-2} - 8f_{-1} + 8f_1 - f_2)$
$h^2 f''$	$f_{-1} - 2f_0 + f_1$	$\frac{1}{12}(-f_{-2} + 16f_{-1} - 30f_0 + 16f_1 - f_2)$
$h^3 f'''$	$\pm(-f_{\mp 1} + 3f_0 - 3f_{\pm 1} + f_{\pm 2})$	$\frac{1}{2}(-f_{-2} + 2f_{-1} - 2f_1 + f_2)$
$h^4 f^{iv}$...	$f_{-2} - 4f_{-1} + 6f_0 - 4f_1 + f_2$

Python interpretation of the previous equations,

```

1 def symmetric4(x,h):
2     #Performs the symmetric 4-point method on the function defined by myFunc
3     #Input:  x -- independent variable
4     #        h -- step-size
5     #Output: ans -- dependent variable
6     ans = (-2*myFunc(x-h)-3*myFunc(x)+6*myFunc(x+h)-myFunc(x+2*h)) / (6*h)
7     return(ans)
8
9 def symmetric5(x,h):
10    #Performs the symmetric 5-point method on the function defined by myFunc
11    #Input:  x -- independent variable
12    #        h -- step-size
13    #Output: ans -- dependent variable
14    ans = (myFunc(x-2*h)-8*myFunc(x-h)+8*myFunc(x+h)-myFunc(x+2*h)) / (12*h)
15    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 .

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

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

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,

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

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 interval $[a+h, b]$ in Fig.1), trapezoid method will over-estimate the true value. Similarly, when the function is concave down, (such as the interval $[a, a+h]$ in Fig.1), the trapezoid method will under-estimate the true value. Conversely, the midpoint method,

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

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.² This equation is known as the Simpson's 1/3 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 intervals over the interval $[a, b]$, Simpson's Rule becomes,

$$S_N = \frac{h}{3} [f(a) + 4f(a+h) + 2f(a+2h) + \dots + 2f(b-2h) + 4f(b-h) + f(b)] + \mathcal{O}(h^5) \quad (8)$$

²Taken from Introduction to Numerical Methods and MATLAB PROGRAMMING FOR ENGINEERS BY TODD YOUNG AND MARTIN J. MOHLENKAMP

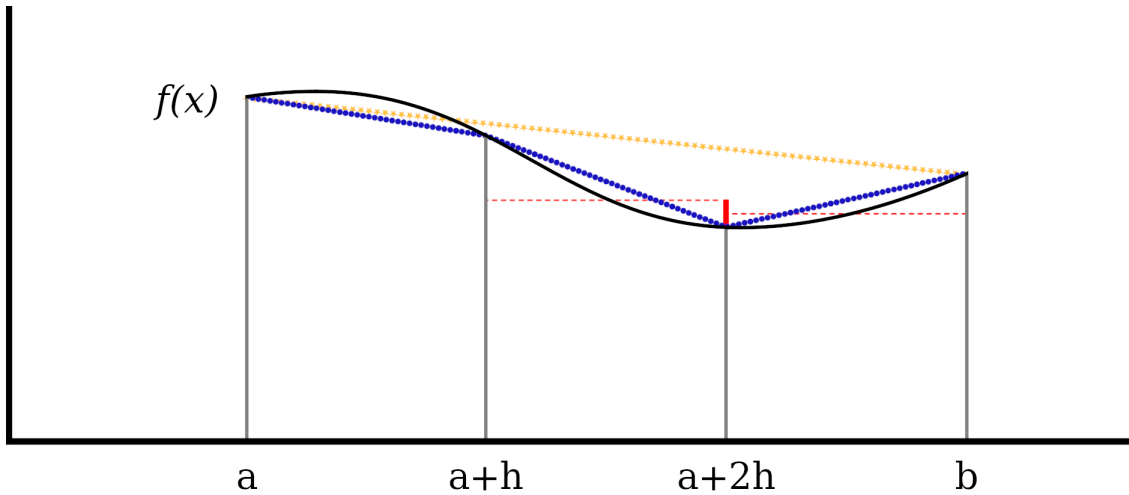


Figure 1: The yellow, star-dashed line shows a linear interpolation of the function over the full domain. The blue, circle-dashed line shows a linear interpolation of the function with lattices of width h . The red, dashed line shows the midpoint method. It is easy to see the trapezoid method over-estimates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

```

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

```

```

1 def simpsons(x,h,N):
2     #Performs Simpons rule (equation 1.12).
3     #Input: x -- lower bound of the range of integration
4     #       h -- step-size
5     #       N -- number of lattices
6     #Output: sum -- approximate integral
7     #Add the contribution from the first and last points of the domain
8     sum = myFunction(x) + myFunction(x+N*h)
9     for i in range(N-1):
10        j = i+1
11    #Adds the contribution from the even placed lattice points
12    if (j % 2 == 1):
13        sum = sum + 4.0*myFunction(x+j*h)
14    #Adds the contribution from the odd placed lattice points
15    else:
16        sum = sum + 2.0*myFunction(x+j*h)
17    sum = sum*h/3.0
18    return sum

```

```

1 def booles(x,h,N):
2     #Performs Bode's rule (equation 1.13b)
3     #Input: x -- lower bound of the range of integration
4     #       h -- step-size
5     #       N -- number of lattices
6     #Output: sum -- approximate integral

```

```

7 #Add the contribution from the first and last points of the domain
8     sum = 7.0*(myFunction(x) + myFunction(x+N*h))
9     for i in range(N-1):
10         j = i+1
11 #Adds the contribution from the even placed lattice points
12     if (j % 2 == 1):
13         sum = sum + 32.0*myFunction(x+j*h)
14     elif(j % 4 == 2):
15         sum = sum + 12.0*myFunction(x+j*h)
16 #Adds the contribution from the odd placed lattice points
17     else:
18         sum = sum + 14.0*myFunction(x+j*h)
19     sum = sum * 2.0 * h / 45.0
20     return sum

```

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

N	h	Trapezoidal Eq. 1.9	Simpson's Eq. 1.12	Boole's Eq 1.13
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