8 Differentiation

Lab Objective: *The derivative is critically important in many applications. Depending on the application and on the available information, the derivative may be calculated symbolically, numerically, or with differentiation software. In this lab we explore these three ways to take a derivative, discuss what settings they are each appropriate for, and demonstrate their strengths and weaknesses.*

Symbolic Differentiation

The derivative of a known mathematical function can be calculated symbolically with SymPy. This method is the most precise way to take a derivative, but it is computationally expensive and requires knowing the closed form formula of the function. Use `sy.diff()` to take a symbolic derivative.

```
>>> import sympy as sy

>>> x = sy.symbols('x')
>>> sy.diff(x**3 + x, x)      # Differentiate x^3 + x with respect to x.
3*x**2 + 1
```

Problem 1. Write a function that defines $f(x) = (\sin(x) + 1)^{\sin(\cos(x))}$ and takes its symbolic derivative with respect to x using SymPy. Lambdify the resulting function so that it can accept NumPy arrays and return the resulting function handle.

To check your function, plot f and its derivative f' over the domain $[-\pi, \pi]$. It may be helpful to move the bottom spine to 0 so you can see where the derivative crosses the x-axis.

```
>>> from matplotlib import pyplot as plt

>>> ax = plt.gca()
>>> ax.spines["bottom"].set_position("zero")
```

Numerical Differentiation

One definition for the derivative of a function $f : \mathbb{R} \to \mathbb{R}$ at a point x_0 is

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}.$$

Since this definition relies on h approaching 0, choosing a small, fixed value for h approximates $f'(x_0)$.

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h} \tag{8.1}$$

This quotient is called the *first order forward difference quotient*. Using the points x_0 and $x_0 - h$ in place of $x_0 + h$ and x_0, respectively, results in the *first order backward difference quotient*.

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - h)}{h} \tag{8.2}$$

Forward difference quotients use values of f at x_0 and points greater than x_0, while backward difference quotients use the values of f at x_0 and points less than x_0. A *centered difference quotient* uses points on either side of x_0, and typically results in a better approximation than the one-sided quotients. Adding (8.1) and (8.2) yields the *second order centered difference quotient*.

$$f'(x_0) = \frac{1}{2}f'(x_0) + \frac{1}{2}f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{2h} + \frac{f(x_0) - f(x_0 - h)}{2h} = \frac{f(x_0 + h) - f(x_0 - h)}{2h}$$

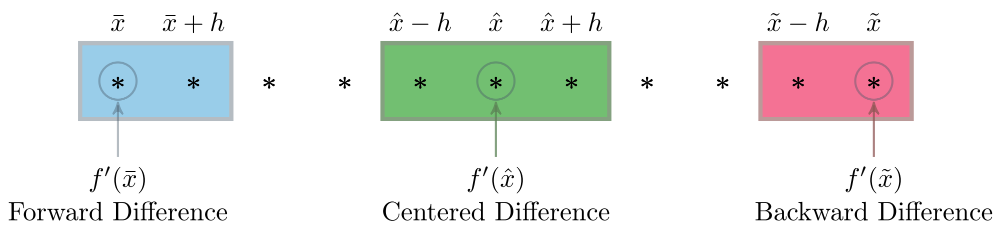


Figure 8.1

Note

The finite difference quotients in this section all approximate the first derivative of a function. The terms *first order* and *second order* refers to how quickly the approximation converges on the actual value of $f'(x_0)$ as h approaches 0, not to how many derivatives are being taken.

There are finite difference quotients for approximating higher order derivatives, such as f'' or f'''. For example, the following is a centered difference quotient for the second derivative.

$$f''(x_0) \approx \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2}$$

This particular quotient is important for finite difference methods that approximate numerical solutions to certain partial differential equations.

While we do not derive them here, there are other finite difference quotients that use more points to approximate the derivative, some of which are listed below. Using more points generally results in better convergence properties.

Type	Order	Formula
Forward	1	$\frac{f(x_0+h)-f(x_0)}{h}$
	2	$\frac{-3f(x_0)+4f(x_0+h)-f(x_0+2h)}{2h}$
Backward	1	$\frac{f(x_0)-f(x_0-h)}{h}$
	2	$\frac{3f(x_0)-4f(x_0-h)+f(x_0-2h)}{2h}$
Centered	2	$\frac{f(x_0+h)-f(x_0-h)}{2h}$
	4	$\frac{f(x_0-2h)-8f(x_0-h)+8f(x_0+h)-f(x_0+2h)}{12h}$

Table 8.1: Common finite difference quotients for approximating $f'(x_0)$.

Problem 2. Write a function for each of the finite difference quotients listed in Table 8.1. Each function should accept a function handle f, an array of points `x`, and a float h; each should return an array of the difference quotients evaluated at each point in `x`.

To test your functions, approximate the derivative of $f(x) = (\sin(x)+1)^{\sin(\cos(x))}$ at each point of a domain over $[-\pi, \pi]$. Plot the results and compare them to the results of Problem 1.

Convergence of Finite Difference Quotients

Finite difference quotients are typically derived using Taylor's formula. This method also shows how the accuracy of the approximation increases as $h \to 0$.

$$f(x_0+h) = f(x_0) + f'(x_0)h + R_2(h) \quad \Longrightarrow \quad \frac{f(x_0+h)-f(x_0)}{h} - f'(x_0) = \frac{R_2(h)}{h}, \tag{8.3}$$

where $R_2(h) = h^2 \int_0^1 (1-t) f''(x_0+th)\,dt$. Thus the absolute error of the first order forward difference quotient is

$$\left|\frac{R_2(h)}{h}\right| = |h| \left|\int_0^1 (1-t) f''(x_0+th)\,dt\right| \leq |h| \int_0^1 |1-t||f''(x_0+th)|\,dt.$$

If f'' is continuous, then for any $\delta > 0$, setting $M = \sup_{x\in(x_0-\delta,x_0+\delta)} f''(x)$ guarantees that

$$\left|\frac{R_2(h)}{h}\right| \leq |h|\int_0^1 M dt = M|h| \in O(h).$$

whenever $|h| < \delta$. That is, the error decreases at the same rate as h. If h gets twice as small, the error does as well. This is what is meant by a *first order* approximation. In a *second order* approximation, the absolute error is $O(h^2)$, meaning that if h gets twice as small, the error gets four times smaller.

Note

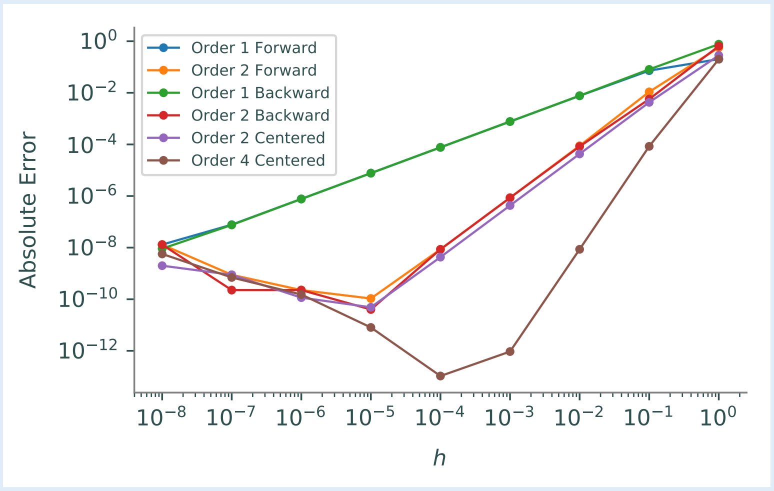
The notation $O(f(n))$ is commonly used to describe the temporal or spatial complexity of an algorithm. In that context, a $O(n^2)$ algorithm is much worse than a $O(n)$ algorithm. However, when referring to error, a $O(h^2)$ algorithm is **better** than a $O(h)$ algorithm because it means that the accuracy improves faster as h decreases.

Problem 3. Write a function that accepts a point x_0 at which to compute the derivative of $f(x) = (\sin(x)+1)^{\sin(\cos(x))}$. Use your function from Problem 1 to compute the exact value of $f'(x_0)$. Then use each your functions from Problem 2 to get an approximate derivative $\tilde{f}'(x_0)$ for $h = 10^{-8}, 10^{-7}, \ldots, 10^{-1}, 1$. Track the absolute error $|f'(x_0) - \tilde{f}'(x_0)|$ for each trial, then plot the absolute error against h on a log-log scale (use `plt.loglog()`).

Instead of using `np.linspace()` to create an array of h values, use `np.logspace()`. This function generates logarithmically spaced values between two powers of 10.

```
>>> import numpy as np
>>> np.logspace(-3, 0, 4)            # Get 4 values from 1e-3 to 1e0.
array([ 0.001,  0.01 ,  0.1  ,  1.   ])
```

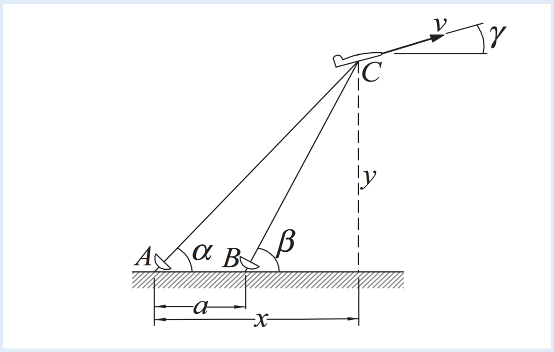
For $x_0 = 1$, your plot should resemble the following figure.



Achtung!

Mathematically, choosing smaller h values results in tighter approximations of $f'(x_0)$. However, Problem 3 shows that when h gets too small, the error stops decreasing. This numerical error is due to the denominator in each finite difference quotient becoming very small. The optimal value of h is usually one that is small, but not too small.

Problem 4. The radar stations A and B, separated by the distance $a = 500$ m, track a plane C by recording the angles α and β at one-second intervals. Your goal, back at air traffic control, is to determine the speed of the plane.



Let the position of the plane at time t be given by $(x(t), y(t))$. The speed at time t is the magnitude of the velocity vector, $\|\frac{d}{dt}(x(t), y(t))\| = \sqrt{x'(t)^2 + y'(t)^2}$. The closed forms of the functions $x(t)$ and $y(t)$ are unknown (and may not exist at all), but we can still use numerical methods to estimate $x'(t)$ and $y'(t)$. For example, at $t = 3$, the second order centered difference quotient for $x'(t)$ is

$$x'(3) \approx \frac{x(3+h) - x(3-h)}{2h} = \frac{1}{2}(x(4) - x(2)).$$

In this case $h = 1$ since data comes in from the radar stations at 1 second intervals.

Successive readings for α and β at integer times $t = 7, 8, \ldots, 14$ are stored in the file `plane.npy`. Each row in the array represents a different reading; the columns are the observation time t, the angle α (in degrees), and the angle β (also in degrees), in that order. The Cartesian coordinates of the plane can be calculated from the angles α and β as follows.

$$x(\alpha, \beta) = a\frac{\tan(\beta)}{\tan(\beta) - \tan(\alpha)} \qquad y(\alpha, \beta) = a\frac{\tan(\beta)\tan(\alpha)}{\tan(\beta) - \tan(\alpha)}$$

Load the data, convert α and β to radians, then compute the coordinates $x(t)$ and $y(t)$ at each given t. Approximate $x'(t)$ and $y'(t)$ using a forward difference quotient for $t = 7$, a backward difference quotient for $t = 14$, and a centered difference quotient for $t = 8, 9, \ldots, 13$ (see Figure 8.1). Return the values of the speed $\sqrt{x'(t)^2 + y'(t)^2}$ at each t.[a]
(Hint: `np.deg2rad()` will be helpful.)

[a]This problem originates from *Numerical Methods in Engineering with Python 3* by Jaan Kiusalaas.

Numerical Differentiation in Higher Dimensions

Finite difference quotients can also be used to approximate derivatives in higher dimensions. The *Jacobian* of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ at a point $\mathbf{x}_0 \in \mathbb{R}^n$ is the $m \times n$ matrix J whose entries are given by

$$J_{ij} = \frac{\partial f_i}{\partial x_j}(\mathbf{x}_0).$$

For example, the Jacobian for a function $f : \mathbb{R}^3 \to \mathbb{R}^2$ is defined by

$$J = \left[\begin{array}{c|c|c} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{array} \right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{bmatrix}, \qquad \text{where} \qquad f(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

The difference quotients in this case resemble directional derivatives. The first order forward difference quotient for approximating a partial derivative is

$$\frac{\partial f}{\partial x_j}(\mathbf{x}_0) \approx \frac{f(\mathbf{x}_0 + h\mathbf{e}_j) - f(\mathbf{x}_0)}{h},$$

where $\mathbf{e}_j$ is the j^{th} standard basis vector. The second order centered difference approximation is

$$\frac{\partial f}{\partial x_j}(\mathbf{x}_0) \approx \frac{f(\mathbf{x}_0 + h\mathbf{e}_j) - f(\mathbf{x}_0 - h\mathbf{e}_j)}{2h}. \tag{8.4}$$

Problem 5. Write a function that accepts a function $f : \mathbb{R}^n \to \mathbb{R}^m$, a point $\mathbf{x}_0 \in \mathbb{R}^n$, and a float h. Approximate the Jacobian matrix of f at $\mathbf{x}$ using the second order centered difference quotient in (8.4).
(Hint: the standard basis vector $\mathbf{e}_j$ is the jth column of the $n \times n$ identity matrix I.)

To test your function, define a simple function like $f(x, y) = [x^2, x^3 - y]^\mathsf{T}$ where the Jacobian is easy to find analytically, then check the results of your function against SymPy or your own scratch work.

Autograd

Autograd is a package that allows for efficient automatic differentiation of NumPy and some SciPy code. It has the ability to handle taking the derivative of functions that contain almost all NumPy structures[1], including `if` statements, `while` loops and recursion. It is very beneficial when calculating derivatives of unconventional or very complex functions. Due to this feature, it is extremely useful in many applications such as machine learning and neural networks.

Autograd is installed by running `pip install autograd` in the terminal. See `https://github.com/HIPS/autograd` for more complete installation instructions.

To support most of the NumPy features, autograd uses a thinly-wrapped version of Numpy called `autograd.numpy`. This lab will denote the autograd's version of NumPy as `anp`. Use `anp` the same way NumPy is used.

The function `grad()` returns a function that is the gradient of the original function, if the original function returns a scalar. This new function accepts the same parameters as the original function. The following code computes the derivative of $e^{\sin(\cos(x))}$ at $x = 1$ using autograd.

[1]For a list of NumPy features that autograd does not support, refer to `https://github.com/HIPS/autograd/blob/master/docs/tutorial.md`.

```
>>> from autograd import grad
>>> import autograd.numpy as anp        # Use autograd's version of NumPy.

>>> g = lambda x: anp.exp(anp.sin(anp.cos(x)))
>>> grad_g = grad(g)
>>> grad_g(1.)
-1.2069777039799139
```

For multivariate functions, the parameter `argnum` specifies the variable with respect to which the gradient is computed.

```
>>> f = lambda x,y: 3*x*y+2*y- x

# Take the gradient with respect to the first variable x.
>>> grad_f = grad(f, argnum=0)
>>> grad_f(.25,.5)
array(0.5)

# Take the gradient with respect to the second variable y.
>>> grad_f = grad(f, argnum=1)
>>> grad_f(.25,.5)
array(2.75)
```

Finding the gradient with respect to multiple variables can by done using `multigrad()` by specifying the variables in the `argnums` parameter.

```
from autograd import multigrad

>>> grad_f = multigrad(f, argnums=[0,1])
>>> grad_f(.25,.5)
(array(0.5),array(2.75))
```

Problem 6. Compute the derivative of $f(x) = \ln\sqrt{\sin(\sqrt{x})}$ at $x = \frac{\pi}{4}$ using SymPy, the second order centered difference quotient and autograd. Print the computation time and error for each method. Do not include initializations of functions or variables in the computation time. Return the value of the autograd approximation.

SymPy will take the exact derivative of the function yielding zero error. However, SymPy will also have the longest computation time. The second order centered difference quotient will take the least amount of time and produce the greatest error. Autograd will have a shorter computation time than SymPy and a smaller error than the second order centered difference quotient.

As seen in the previous problem, autograd can be an efficient tool in differentiation. Although autograd does not calculate exact derivatives, the resulting error is relatively small with less computation time than SymPy.

Autograd can differentiate a function as many times as desired.

```
>>> f = lambda x: anp.sin(x)+3**anp.cos(x)

# Calculate the first derivative.
>>> grad_f = grad(f)

# Calculate the second derivative and so forth.
>>> grad_f2 = grad(grad_f)
>>> grad_f3 = grad(grad_f2)
>>> df3(1.)
array(2.683445898750351)
```

The main advantage of using autograd in differentiation is that it can differentiate many structures in Python functions. In fact, it can handle functions that include loops, `if` statements and recursion. For example, the following code computes the Taylor series of e^x evaluated at $x = 2$.

```
import numpy as np

# Define the Taylor series.
# Note that this function does not account for array broadcasting.
>>> def taylor_exp(x, tol=.0001):
...     result = 0
...     cur_term = x
...     i = 0
...     while anp.abs(cur_term) >= tol:
# Autograd's version of NumPy doesn't have the math attribute so use NumPy.
...         cur_term = x**i/np.math.factorial(i)
...         result += cur_term
...         i += 1
...     return result

# Compute the gradient.
>>> d_taylor_exp = grad(taylor_exp)
# Note that differentiation in autograd only works with float values.
>>> d_taylor_exp(2.)
array(7.388994708994709)
```

Problem 7. Write a function that uses autograd to take the first and second derivatives of the Taylor series of $\sin(x)$. Plot the original function along with its two derivatives on the interval $[-\pi, \pi]$.

Although `grad()` is very efficient, it does not allow for array broadcasting. However, autograd has another function `elementwise_grad()` that does.

```
>>> from autograd import elementwise_grad

>>> grad_f = elementwise_grad(f)
>>> grad_f(anp.array([1.,2.,3.]))
array([-1.13338111, -1.04855565, -1.04224253])
```

While the `grad()` function can only find the gradient of functions that output scalars, `jacobian()` can find the gradient of vectors. The following example shows how to find the Jacobian of $f(x,y) = \begin{bmatrix} x^2 \\ x+y \end{bmatrix}$ evaluated at $(1,1)$.

```
>>> from autograd import jacobian

>>> f = lambda x: anp.array([x[0]**2, x[0]+x[1]])
>>> jacobian_f = jacobian(f)
>>> jacobian_f(anp.array([1.,1.]))
array([[ 2.,  0.],
       [ 1.,  1.]])
```

Problem 8.

Let $f : \mathbb{R}^2 \to \mathbb{R}^2$ be defined by

$$f(x,y) = \begin{bmatrix} e^x \sin(y) + y^3 \\ 3y - \cos(x) \end{bmatrix}.$$

Find the Jacobian using SymPy, the second order centered difference quotient and autograd. Print the time it takes to compute each Jacobian evaluated at $(x,y) = (1,1)$. Do not include the initialization of any variables or functions in the computation time. Return the value of the autograd approximation.

See `https://github.com/HIPS/autograd` for more examples with autograd.