1 Differentiation

Lab Objective: *Derivatives are central 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 $f'(x_0)$:

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}. \tag{1.1}$$

This approximation 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*,

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - h)}{h}. \tag{1.2}$$

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. Combining (1.1) and (1.2) yields the *second order centered difference quotient*,

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

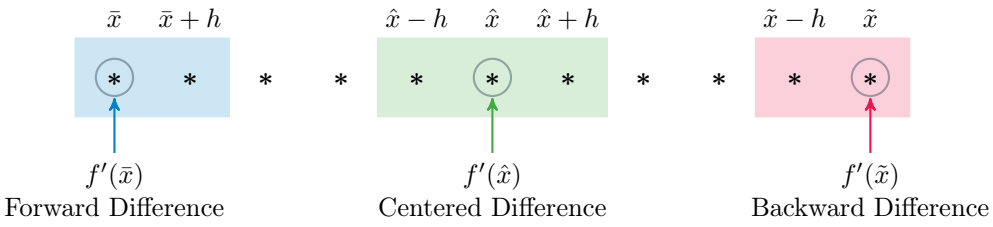


Figure 1.1

> **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 centered difference quotient
>
> $$f''(x_0) \approx \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2}$$
>
> approximates the second derivative. This particular quotient is important for finite difference methods that approximate numerical solutions to some 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 1.1: Common finite difference quotients for approximating $f'(x_0)$.

Problem 2. Write a function for each of the finite difference quotients listed in Table 1.1. 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 $h \to 0$:

$$f(x_0+h) = f(x_0) + f'(x_0)h + R_2(h) \quad \Longrightarrow \quad \frac{f(x_0+h)-f(x_0)}{h} - f'(x_0) = \frac{R_2(h)}{h}, \tag{1.3}$$

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

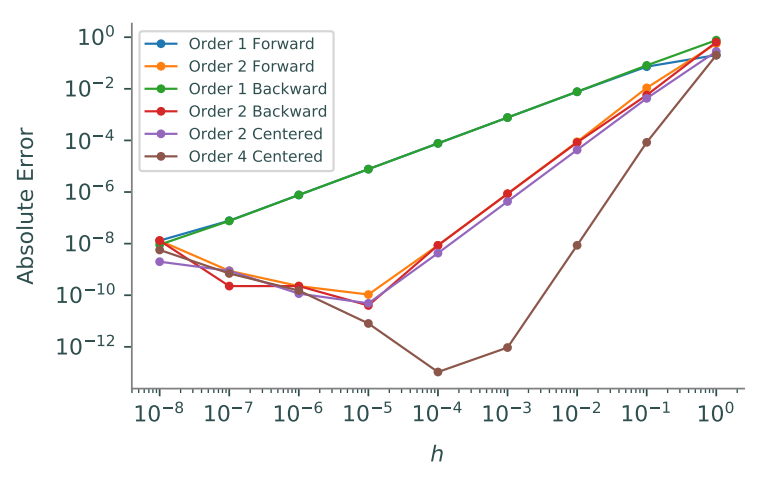
The notation $O(f(n))$ is commonly used to describe the temporal or spatial complexity of an algorithm. In that context, a $O(n^2)$ algorithm is much worse than a $O(n)$ algorithm. However, when referring to error, a $O(h^2)$ algorithm is **better** than a $O(h)$ algorithm because it means that the accuracy 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 $h = 10^{-8}, 10^{-7}, \dots, 10^{-1}, 1$. 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.   ])
```

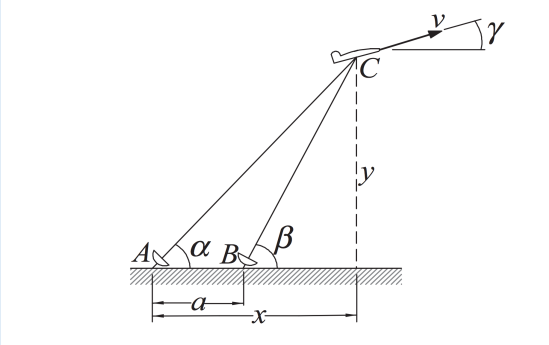
For $x_0 = 1$, your plot should resemble the following figure.



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.[a]



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.

$$x(\alpha, \beta) = a\frac{\tan(\beta)}{\tan(\beta) - \tan(\alpha)} \qquad y(\alpha, \beta) = a\frac{\tan(\beta)\tan(\alpha)}{\tan(\beta) - \tan(\alpha)} \tag{1.4}$$

Load the data, convert α and β to radians, then compute the coordinates $x(t)$ and $y(t)$ at each given t using 1.4. Approximate $x'(t)$ and $y'(t)$ using a first order forward difference quotient for $t = 7$, a first order backward difference quotient for $t = 14$, and a second order centered difference quotient for $t = 8, 9, \ldots, 13$ (see Figure 1.1). Return the values of the speed $\sqrt{x'(t)^2 + y'(t)^2}$ at each t.

(Hint: `np.deg2rad()` will be helpful.)

[a] This problem is adapted from an exercise in [?].

Numerical Differentiation in Higher Dimensions

Finite difference quotients can also be used to approximate derivatives in higher dimensions. The *Jacobian matrix* 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

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

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 jth standard basis vector. The second order centered difference approximation is

$$\frac{\partial f}{\partial x_j}(\mathbf{x}_0) \approx \frac{f(\mathbf{x}_0 + h\mathbf{e}_j) - f(\mathbf{x}_0 - h\mathbf{e}_j)}{2h}. \tag{1.5}$$

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 (1.5).
(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.

Differentiation Software

Many machine learning algorithms and structures, especially neural networks, rely on the gradient of a cost or objective function. To facilitate their research, several organizations have recently developed Python packages for numerical differentiation. For example, the Harvard Intelligent Probabilistic Systems Group (HIPS) started developing `autograd` in 2014 (`https://github.com/HIPS/autograd`) and Google released `tangent` in 2017 (`https://github.com/google/tangent`). These tools are incredibly robust: they can differentiate functions with NumPy routines,[1] `if` statements, `while` loops, and even recursion. We conclude with a brief introduction to Autograd.[2]

[1] See `https://github.com/HIPS/autograd/blob/master/docs/tutorial.md` for which features Autograd supports.
[2] Autograd is not included in Anaconda; install it with `pip install autograd`.

Autograd's `grad()` accepts a scalar-valued function and returns its gradient as a function that accepts the same parameters as the original. To support most of the NumPy features, Autograd comes with its own thinly-wrapped version of Numpy, `autograd.numpy`. Import this version of NumPy as `anp` to avoid confusion.

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

>>> g = lambda x: anp.exp(anp.sin(anp.cos(x)))
>>> dg = grad(g)                            # dg() is a callable function.
>>> dg(1.)                                  # Use floats as input, not ints.
-1.2069777039799139
```

Functions that `grad()` produces do not support array broadcasting, meaning they do not accept arrays as input. Autograd's `elementwise_grad()` returns functions that can accept arrays, like using `"numpy"` as an argument in SymPy's `sy.lambdify()`.

```
>>> from autograd import elementwise_grad

>>> pts = anp.array([1, 2, 3], dtype=anp.float)
>>> dg = elementwise_grad(g)        # Calculate g'(x) with array support.
>>> dg(pts)                         # Evaluate g'(x) at each of the points.
array([-1.2069777 , -0.55514144, -0.03356146])
```

SymPy would have no trouble differentiating $g(x)$ in these examples. However, Autograd can also differentiate Python functions that look nothing like traditional mathematical functions. For example, the following code computes the Taylor series of e^x with a loop.

```
>>> from sympy import factorial

>>> def taylor_exp(x, tol=.0001):
...     """Compute the Taylor series of e^x with terms greater than tol."""
...     result, i, term = 0, 0, x
...     while anp.abs(term) > tol:
...         term = x**i / int(factorial(i))
...         result, i = result + term, i + 1
...     return result
...
>>> d_exp = grad(taylor_exp)
>>> print(d_exp(2., .1), d_exp(2., .0001))
7.26666666667 7.38899470899
```

Problem 6. The *Chebyshev Polynomials* satisfy the recursive relation

$$T_0(x) = 1, \qquad T_1(x) = x, \qquad T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x).$$

Write a function that accepts an array x and an integer n and recursively computes $T_n(x)$. Use

Autograd and your first function to create a function for $T_n'(x)$. Use this last function to plot each $T_n'(x)$ over the domain $[-1, 1]$ for $n = 0, 1, 2, 3, 4$.
(Hint: Use `anp.ones_like(x)` to handle the case when $n = 0$.)

Problem 7. Let $f(x) = (\sin(x) + 1)^{\sin(\cos(x))}$ as in Problems 1 and 3. Write a function that accepts an integer N and performs the following experiment N times.

1. Choose a random value x_0.
2. Use your function from Problem 1 to calculate the "exact" value of $f'(x_0)$. Time how long the entire process takes, including calling your function (each iteration).
3. Time how long it takes to get an approximation $\tilde{f}'(x_0)$ of $f'(x_0)$ using the fourth-order centered difference quotient from Problem 3. Record the absolute error $|f'(x_0) - \tilde{f}'(x_0)|$ of the approximation.
4. Time how long it takes to get an approximation $\bar{f}'(x_0)$ of $f'(x_0)$ using Autograd (calling `grad()` every time). Record the absolute error $|f'(x_0) - \bar{f}'(x_0)|$ of the approximation.

Plot the computation times versus the absolute errors on a log-log plot with different colors for SymPy, the difference quotient, and Autograd. For SymPy, assume an absolute error of `1e-18` (since only positive values can be shown on a log plot).

For $N = 200$, your plot should resemble the following figure. Note that SymPy has the least error but the most computation time, and that the difference quotient takes the least amount of time but has the most error. Autograd might be considered a "happy medium," a least for this problem.

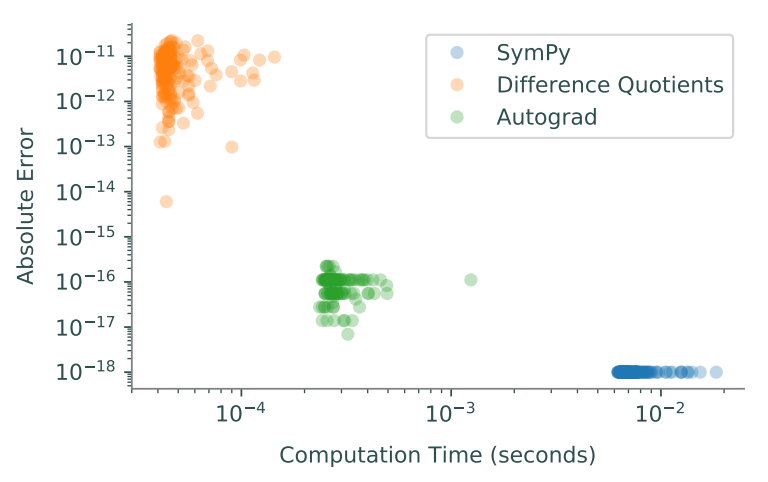


Figure 1.2: Solution with $N = 200$.

Additional Material

More Autograd

For scalar-valued functions with multiple inputs, the parameter `argnum` specifies the variable that the derivative is computed with respect to. Providing a list for `argnum` gives several outputs.

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

# Take the derivative of f with respect to the first variable, x.
>>> dfdx = grad(f, argnum=0)            # Should be dfdx(x,y) = 3y - 1,
>>> dfdx(5., 1.)                        # so dfdx(5,1) = 3 - 1 = 2.
2.0

# Take the gradient with respect to the second variable, y.
>>> dfdy = grad(f, argnum=1)            # Should be dfdy(x,y) = 3x + 2,
>>> dfdy(5., 1.)                        # so dfdy(5,1) = 15 + 2 = 17.
17.0

# Get the full gradient.
>>> grad_f = grad(f, argnum=[0,1])
>>> anp.array(grad_f(5., 1.))
array([  2.,  17.])
```

Finally, Autograd's `jacobian()` can differentiate vector-valued functions.

```
>>> from autograd import jacobian

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

Google Tangent

Google's `tangent` package is similar to Autograd, both in purpose and syntax. However, Tangent differentiates code ahead of time, while Autograd waits until the last second to actually do any calculations. Tangent also tends to be slightly faster than Autograd.

```
>>> import tangent              # Install with 'pip install tangent'.

>>> def f(x):                   # Tangent does not support lambda functions,
...     return x**2 - x + 3
...
>>> df = tangent.grad(f)
>>> df(10)                      # ...but the functions do accept integers.
19.0
```