Calculus I: Limits & Derivatives

This topic, *Calculus I: Limits & Derivatives*, introduces the mathematical field of calculus -- the study of rates of change -- from the ground up. It is essential because computing derivatives via differentiation is the basis of optimizing most machine learning algorithms, including those used in deep learning such as backpropagation and stochastic gradient descent.

Through the measured exposition of theory paired with interactive examples, you'll develop a working understanding of how calculus is used to compute limits and differentiate functions. You'll also learn how to apply automatic differentiation within the popular TensorFlow 2 and PyTorch machine learning libraries. The content covered in this class is itself foundational for several other topics in the *Machine Learning Foundations* series, especially *Calculus II* and *Optimization*.

Segment 1: Limits

- What Calculus Is
- A Brief History of Calculus
- The Method of Exhaustion
- Calculating Limits

Segment 2: Computing Derivatives with Differentiation

- The Delta Method
- The Differentiation Equation
- Derivative Notation
- The Power Rule
- The Constant Multiple Rule
- The Sum Rule
- The Product Rule
- The Quotient Rule
- The Chain Rule

Segment 3: Automatic Differentiation

- AutoDiff with PyTorch
- AutoDiff with TensorFlow 2
- Machine Learning via Differentiation
- Cost (or Loss) Functions
- The Future: Differentiable Programming

Segment 1: Limits

The Calculus of Infinitesimals

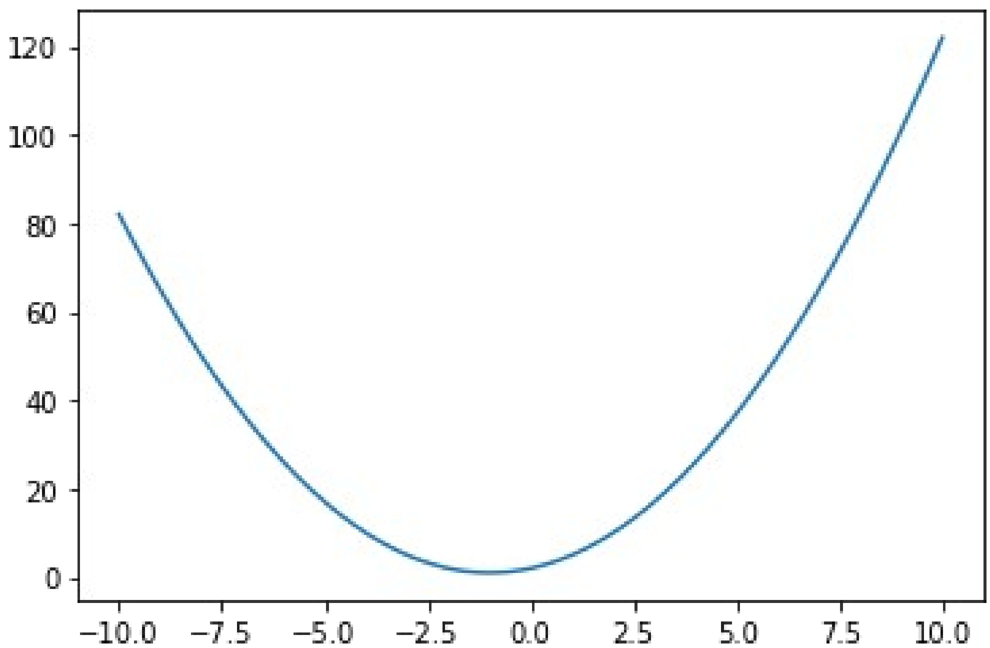
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 10000) # start, finish, n points
```

If $y = x^2 + 2x + 2$:

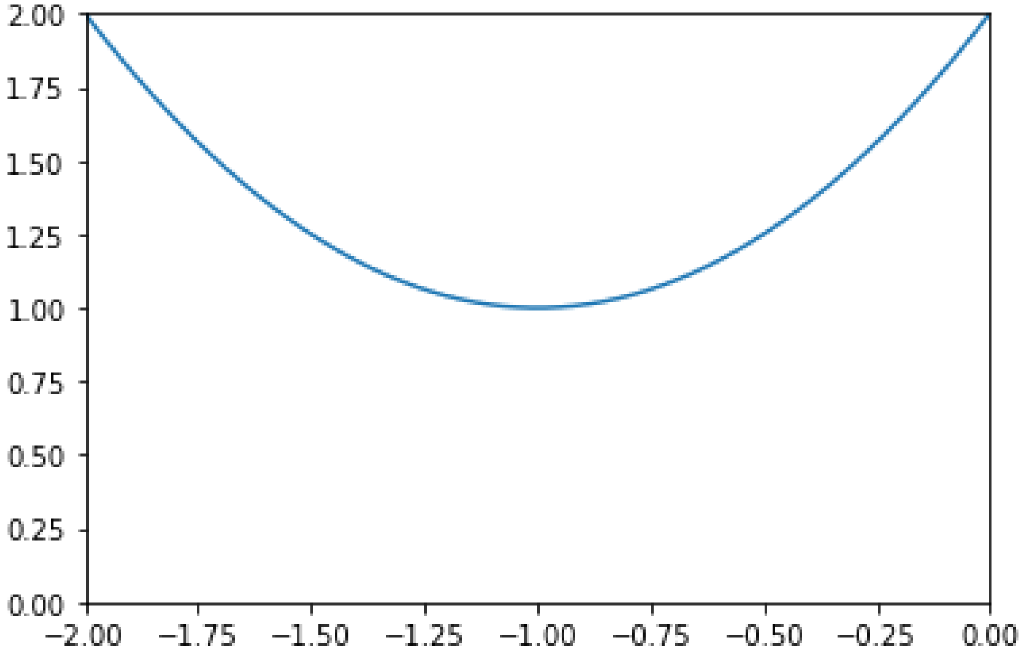
```
y = x**2 + 2*x + 2

fig, ax = plt.subplots()
_ = ax.plot(x,y)
```

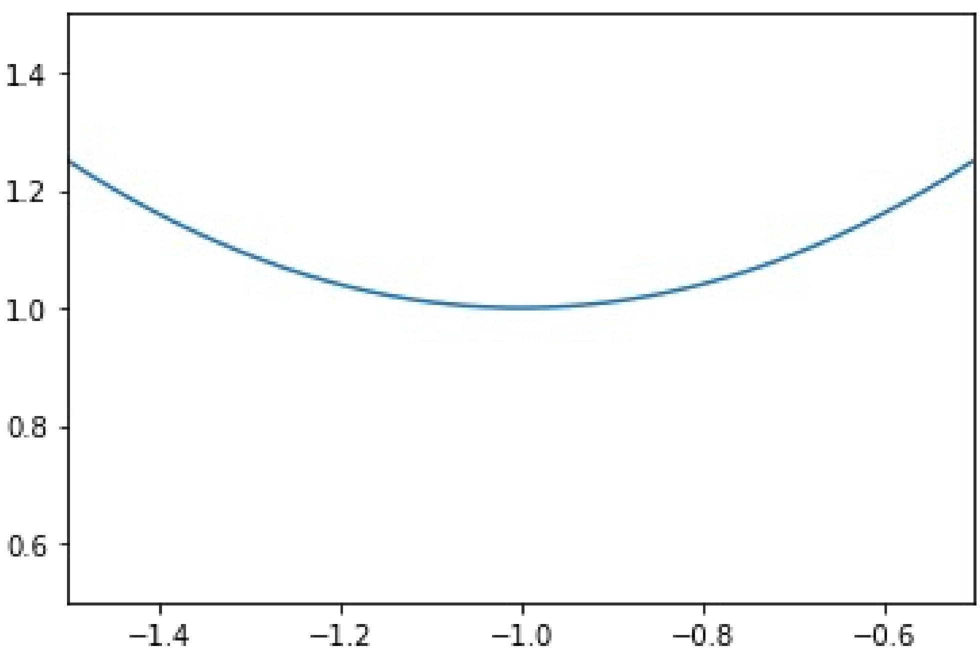


- There are no straight lines on the curve.
- If we zoom in *infinitely* close, however, we observe curves that *approach* lines.
- This enables us to find a slope m (tangent) anywhere on the curve, including to identify where $m = 0$:

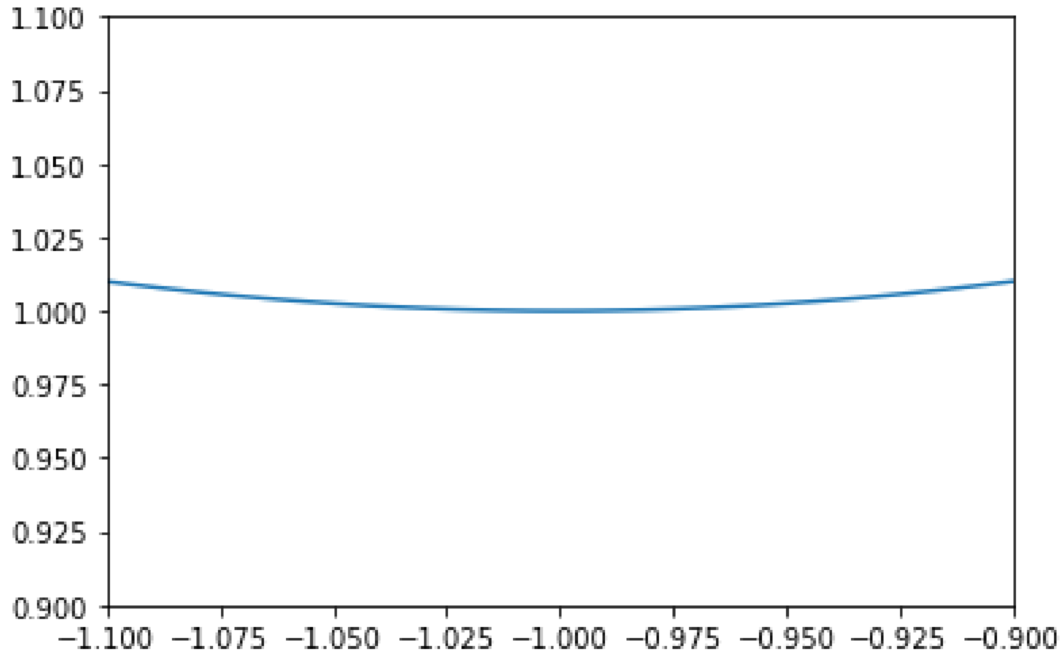
```
fig, ax = plt.subplots()
ax.set_xlim([-2, -0])
ax.set_ylim([0, 2])
_ = ax.plot(x,y)
```



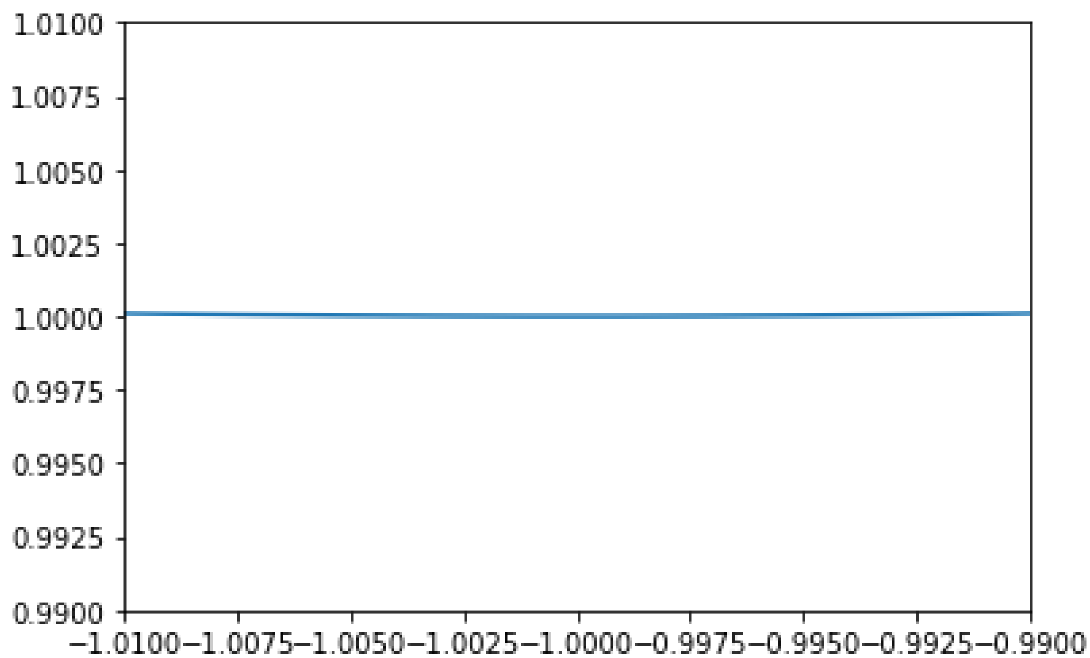

```
fig, ax = plt.subplots()
ax.set_xlim([-1.5, -0.5])
ax.set_ylim([0.5, 1.5])
_ = ax.plot(x,y)
```




```
fig, ax = plt.subplots()
ax.set_xlim([-1.1, -0.9])
ax.set_ylim([0.9, 1.1])
_ = ax.plot(x,y)
```



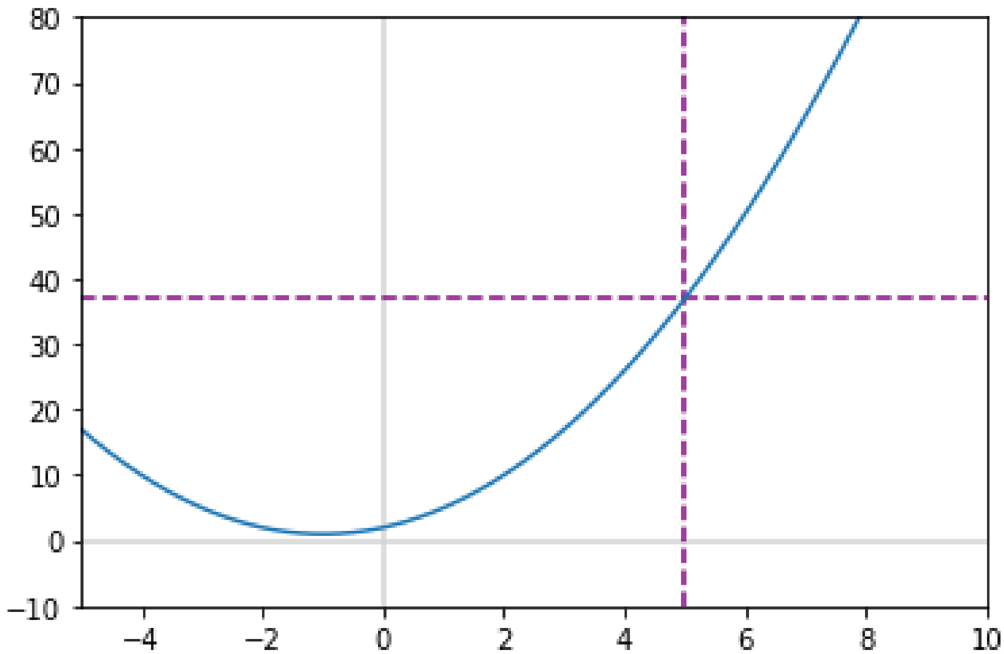

```
fig, ax = plt.subplots()
ax.set_xlim([-1.01, -0.99])
ax.set_ylim([0.99, 1.01])
_ = ax.plot(x,y)
```



Return to slides here.

Limits

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.xlim(-5, 10)
plt.ylim(-10, 80)
plt.axvline(x=5, color='purple', linestyle='--')
plt.axhline(y=37, color='purple', linestyle='--')
_ = ax.plot(x,y)
```



$$\lim_{x \to 1} \frac{x^2 - 1}{x - 1}$$

```
def my_fxn(my_x):
    my_y = (my_x**2 - 1)/(my_x - 1)
    return my_y
```

```
my_fxn(2)
```

```
3.0
```

```
# Uncommenting the following line results in a 'division by zero' 
error:
# my_fxn(1)
```

```
my_fxn(0.9)
```

```
1.9
```

```
my_fxn(0.999)
```

```
1.9989999999999712
```

```
my_fxn(1.1)
```

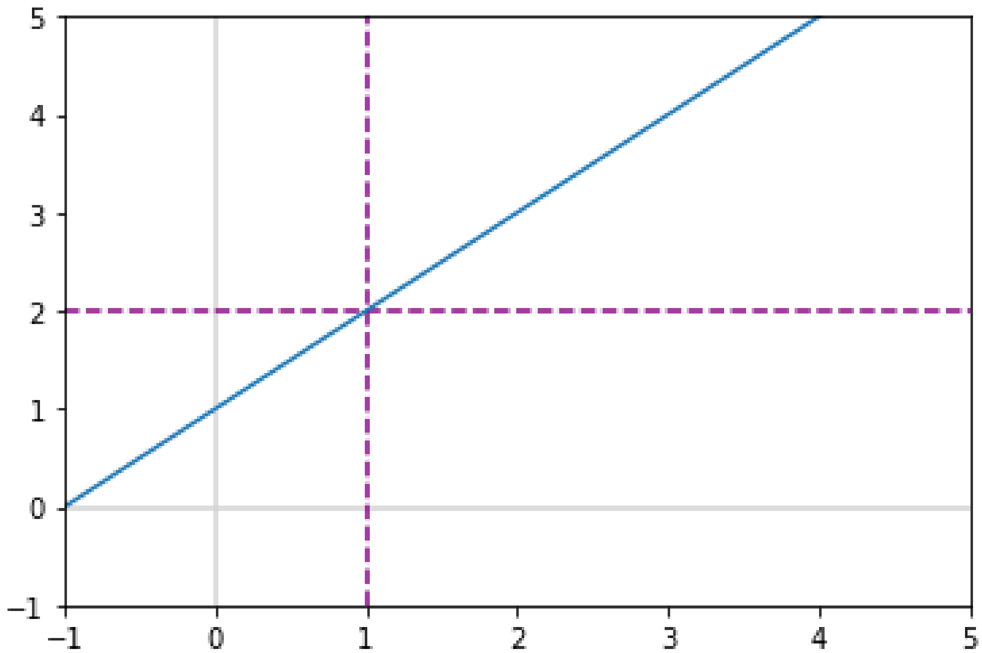
```
2.1
```

```
my_fxn(1.001)
```

```
2.0009999999999177
```

```
y = my_fxn(x)
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.xlim(-1, 5)
plt.ylim(-1, 5)
plt.axvline(x=1, color='purple', linestyle='--')
plt.axhline(y=2, color='purple', linestyle='--')
_ = ax.plot(x,y)
```



Return to slides here.

$$\lim_{x \to 0} \frac{\sin x}{x}$$

```
def sin_fxn(my_x):
    my_y = np.sin(my_x)/my_x
    return my_y
```

```
# Uncommenting the following line results in a 'division by zero'
error:
# y = sin_fxn(0)
```

```
sin_fxn(0.1)
```

```
0.9983341664682815
```

```
sin_fxn(0.001)
```

```
0.9999998333333416
```

```
sin_fxn(-0.1)
```

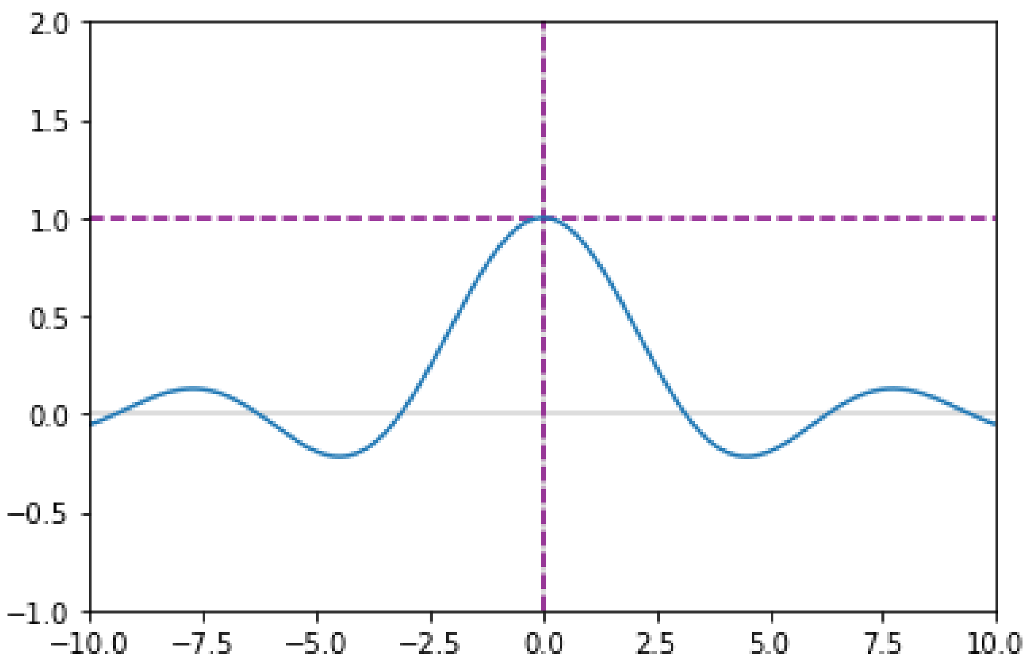
```
0.9983341664682815
```

```
sin_fxn(-0.001)
```

```
0.9999998333333416
```

```
y = sin_fxn(x)
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.xlim(-10, 10)
plt.ylim(-1, 2)
plt.axvline(x=0, color='purple', linestyle='--')
plt.axhline(y=1, color='purple', linestyle='--')
_ = ax.plot(x,y)
```



Return to slides here.

$$\lim_{x \to \infty} \frac{25}{x}$$

```
def inf_fxn(my_x):
    my_y = 25/my_x
    return my_y
```

```
inf_fxn(1e3)
```

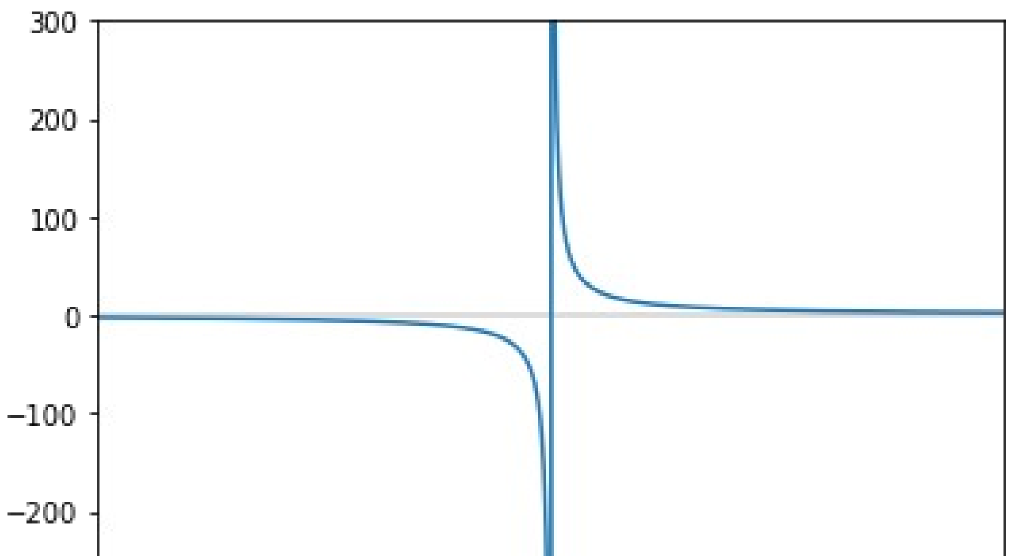
```
0.025
```

```
inf_fxn(1e6)
```

```
2.5e-05
```

```
y = inf_fxn(x)
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.xlim(-10, 10)
plt.ylim(-300, 300)
_ = ax.plot(x, y)
```

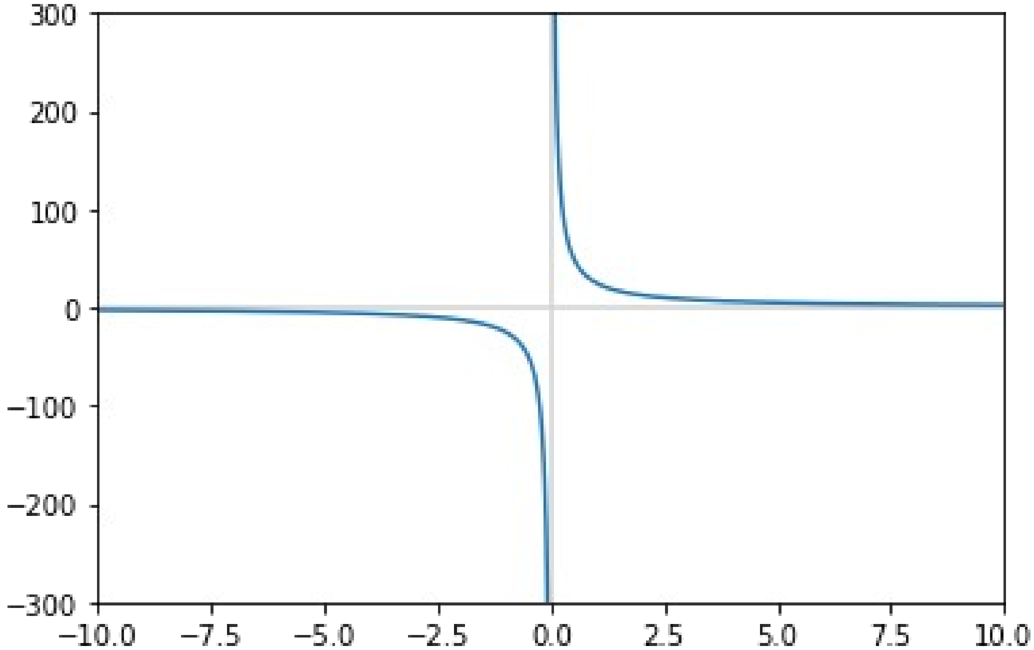



```
left_x = x[x<0]
right_x = x[x>0]
```

```
left_y = inf_fxn(left_x)
right_y = inf_fxn(right_x)
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
```

```
plt.axhline(y=0, color='lightgray')
plt.xlim(-10, 10)
plt.ylim(-300, 300)
ax.plot(left_x, left_y, c='C0')
_ = ax.plot(right_x, right_y, c='C0')
```



Exercises:

Evaluate the limits below using techniques from the slides or above.

$$\lim_{x \to 0} \frac{x^2 - 1}{x - 1}$$

$$\lim_{x \to -5} \frac{x^2 - 25}{x + 5}$$

$$\lim_{x \to 4} \frac{x^2 - 2x - 8}{x - 4}$$

$$\lim_{x \to -\infty} \frac{25}{x}$$

$$\lim_{x \to 0} \frac{25}{x}$$

FYI: While not necessary for ML nor for this *ML Foundations* curriculum, the SymPy symbolic mathematics library includes a `limits()` method. You can read about applying it to evaluate limits of expressions here.

Return to slides here.

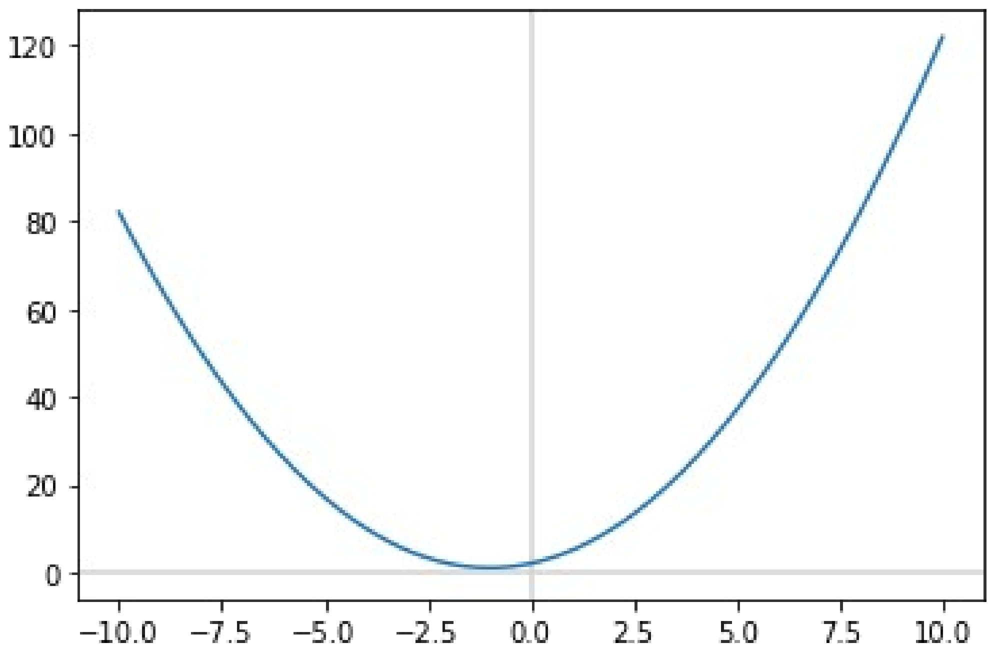
Segment 2: Computing Derivatives with Differentiation

Let's bring back our ol' buddy $y = x^2 + 2x + 2$:

```
def f(my_x):
    my_y = my_x**2 + 2*my_x + 2
    return my_y
```

```
y = f(x)
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
_ = ax.plot(x,y)
```



Let's identify the slope where, say, $x = 2$.

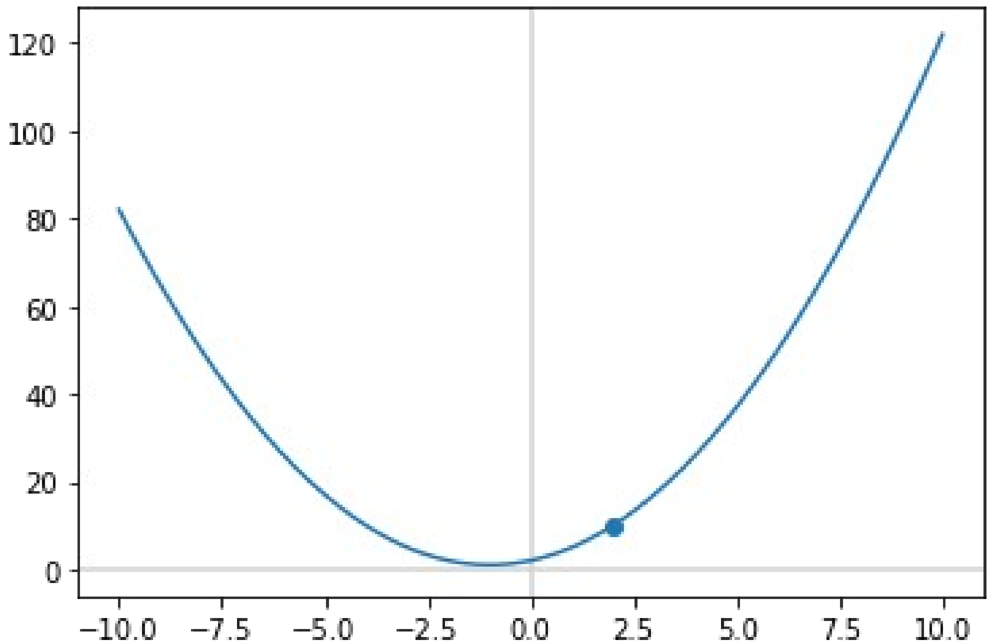
First, let's determine what y is:

```
f(2)
```

```
10
```

Cool. Let's call this point P, which is located at (2, 10):

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.scatter(2, 10) # new
_ = ax.plot(x,y)
```

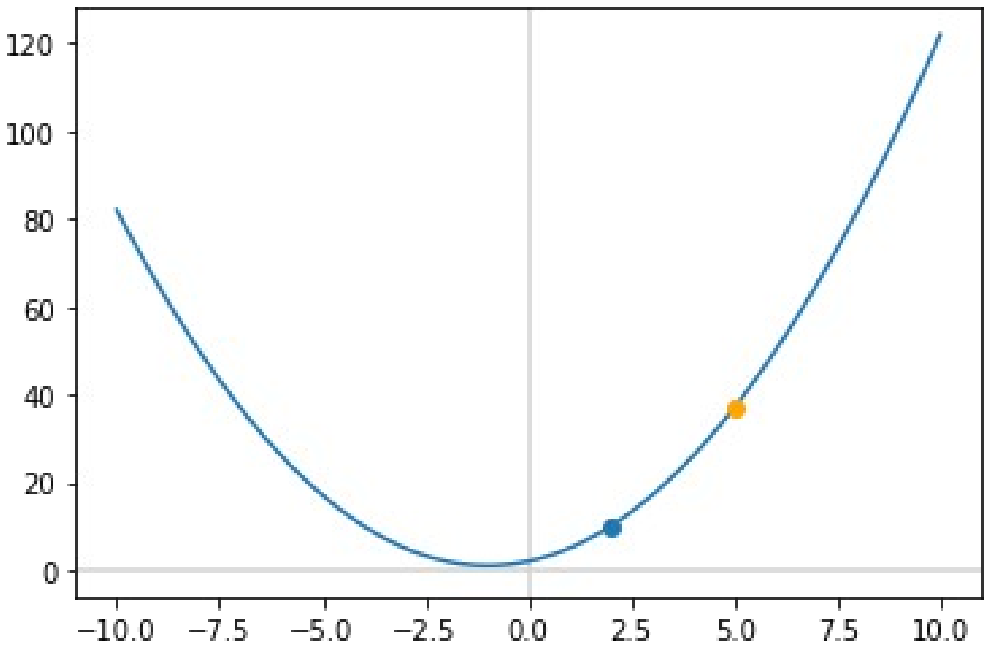


The *delta method* uses the difference between two points to calculate slope. To illustrate this, let's define another point, Q where, say, $x=5$.

```
f(5)
```

```
37
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.scatter(2, 10)
plt.scatter(5, 37, c = 'orange', zorder=3) # new
_ = ax.plot(x,y)
```



To find the slope m between points P and Q:

$$m=\frac{\text{change in } y}{\text{change in } x}=\frac{\Delta y}{\Delta x}=\frac{y_2-y_1}{x_2-x_1}=\frac{37-10}{5-2}=\frac{27}{3}=9$$

```
m = (37-10)/(5-2)
m
```

```
9.0
```

To plot the line that passes through P and Q, we can rearrange the equation of a line $y=mx+b$ to solve for b:

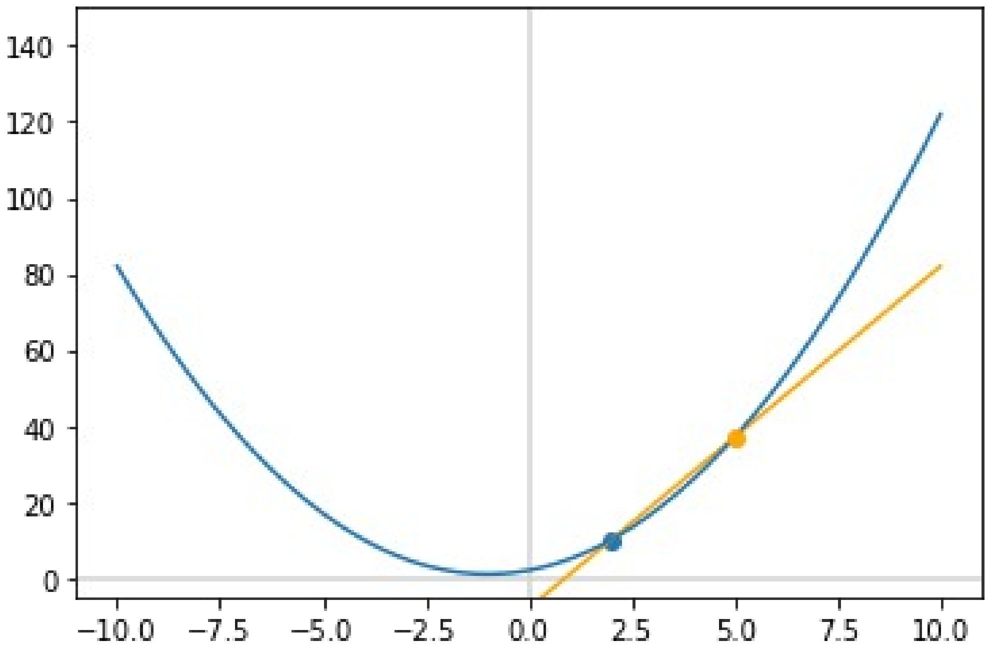
$$b=y-mx$$

```
b = 37-m*5
b
```

```
-8.0
```

```
line_y = m*x + b
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.scatter(2, 10)
plt.scatter(5, 37, c='orange', zorder=3)
plt.ylim(-5, 150) # new
plt.plot(x, line_y, c='orange') # new
_ = ax.plot(x,y)
```



The closer Q becomes to P, the closer the slope m comes to being the true tangent of the point P. Let's demonstrate this with another point Q at $x=2.1$.

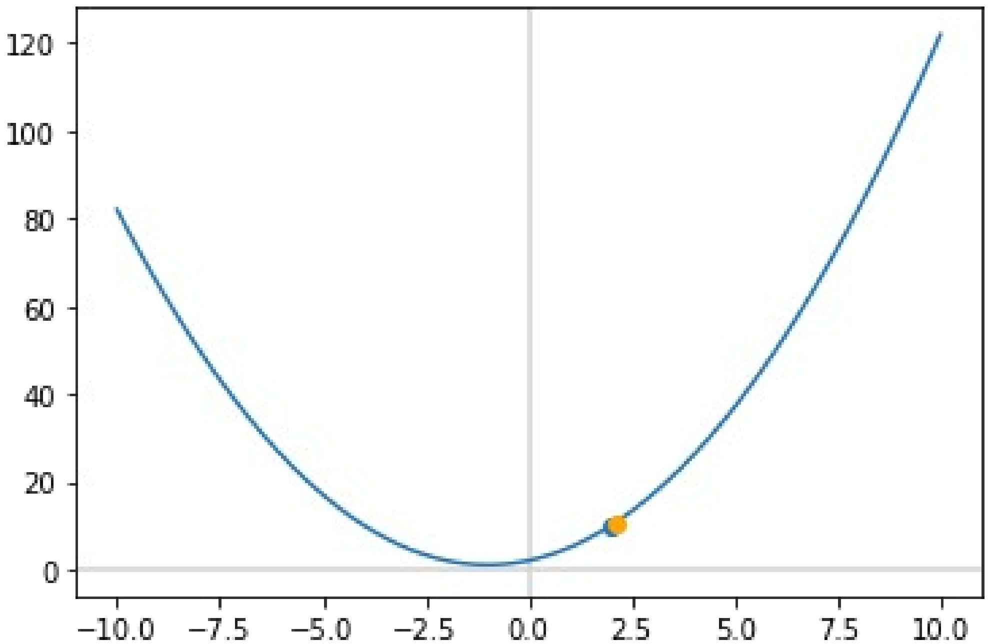
Previously, our Δx between Q and P was equal to 3. Now it is much smaller:

$$\Delta x = x_2 - x_1 = 2.1 - 2 = 0.1$$

```
f(2.1)
```

```
10.61
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.scatter(2, 10)
plt.scatter(2.1, 10.61, c = 'orange', zorder=3)
_ = ax.plot(x,y)
```




```
m = (10.61-10)/(2.1-2)
m
```

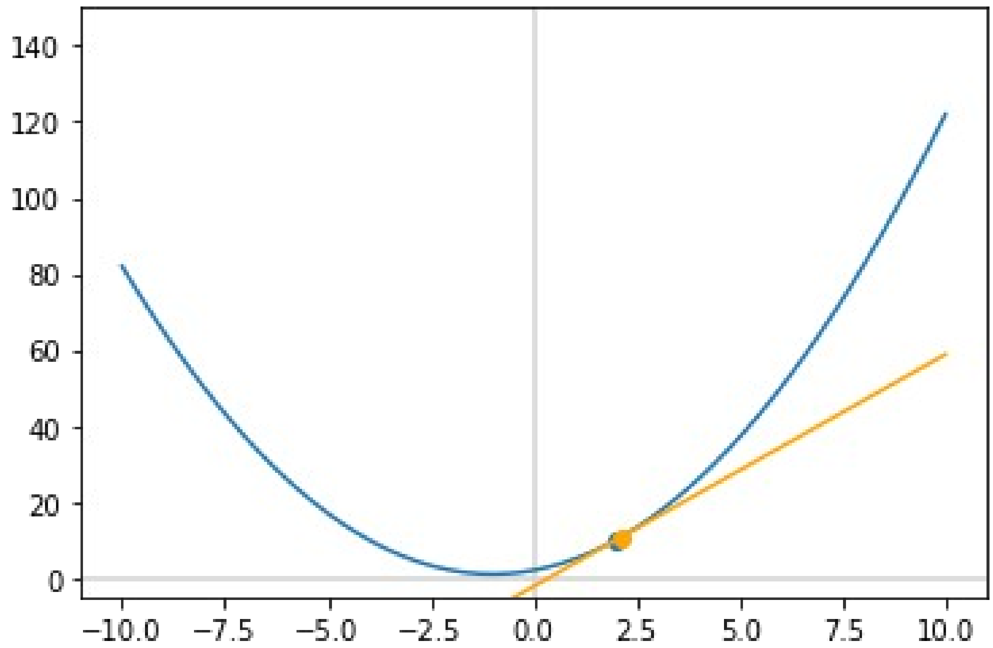
```
6.099999999999989
```

```
b = 10.61-m*2.1
b
```

```
-2.199999999999978
```

```
line_y = m*x + b
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.scatter(2, 10)
plt.scatter(2.1, 10.61, c='orange', zorder=3)
plt.ylim(-5, 150)
plt.plot(x, line_y, c='orange', zorder=3)
_ = ax.plot(x,y)
```



The closer Q becomes to P (i.e., Δx approaches 0), the clearer it becomes that the slope m at point P = (2, 10) is equal to 6.

Let's make Δx extremely small, 0.000001, to illustrate this:

```
delta_x = 0.000001
delta_x
```

```
1e-06
```

```
x1 = 2
y1 = 10
```

Rearranging $\Delta x = x_2 - x_1$, we can calculate x_2 for our point Q, which is now extremely close to P:

$$x_2 = x_1 + \Delta x$$

```
x2 = x1 + delta_x
x2
```

```
2.000001
```

y_2 for our point Q can be obtained with the usual function $f(x)$:

$$y_2 = f(x_2)$$

```
y2 = f(x2)
y2
```

```
10.000006000001001
```

To find the slope m, we continue to use

$$m=\frac{\Delta y}{\Delta x}=\frac{y_2-y_1}{x_2-x_1}$$

```
m = (y2 - y1)/(x2 - x1)
m
```

```
6.000001000088901
```

Boom! Using the delta method, we've shown that at point P, the slope of the curve is 6.

Exercise: Using the delta method, find the slope of the tangent where $x=-1$.

Spoiler alert! The solution's below.

```
x1 = -1
```

```
y1 = f(x1)
y1
```

```
1
```

Point P is located at (-1, 1)

```
delta_x
```

```
1e-06
```

```
x2 = x1 + delta_x
x2
```

```
-0.999999
```

```
y2 = f(x2)
y2
```

```
1.000000000001
```

Quick aside: Pertinent to defining differentiation as an equation, an alternative way to calculate y_2 is $f(x+\Delta x)$

```
y2 = f(x1 + delta_x)
y2
```

```
1.000000000001
```

Point Q is at (-0.999999, 1.000000000001), extremely close to P.

```
m = (y2-y1)/(x2-x1)
m
```

```
1.0000889005535828e-06
```

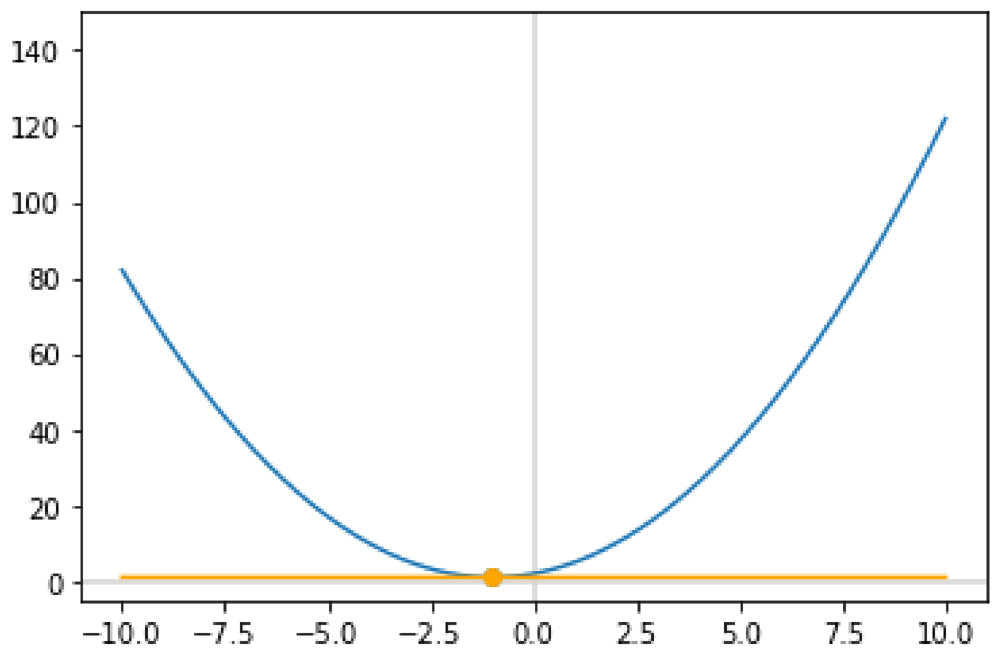
Therefore, as x_2 becomes infinitely close to x_1, it becomes clear that the slope m at $x_1 = -1$ is equal to zero. Let's plot it out:

```
b = y2-m*x2
b
```

```
1.0000010000889006
```

```
line_y = m*x + b
```

```
fig, ax = plt.subplots()
plt.axvline(x=0, color='lightgray')
plt.axhline(y=0, color='lightgray')
plt.scatter(x1, y1)
plt.scatter(x2, y2, c='orange', zorder=3)
plt.ylim(-5, 150)
plt.plot(x, line_y, c='orange', zorder=3)
_ = ax.plot(x,y)
```



As Q becomes infinitely close to P:

- x_2 - x_1 approaches 0
- In other words, Δx approaches 0
- This can be denoted as $\Delta x \rightarrow 0$

Using the delta method, we've derived the definition of differentiation from first principles. The derivative of y (denoted $d\,y$) with respect to x (denoted $d\,x$) can be represented as:

$$\frac{d\,y}{d\,x} = \lim_{\Delta x \rightarrow 0} \frac{\Delta y}{\Delta x}$$

Expanding Δy out to $y_2 - y_1$:

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{y_2 - y_1}{\Delta x}$$

Finally, replacing y_1 with $f(x)$ and replacing y_2 with $f(x + \Delta x)$, we obtain a common representation of differentiation:

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Let's observe the differentiation equation in action:

```
def diff_demo(my_f, my_x, my_delta):
    return (my_f(my_x + my_delta) - my_f(my_x)) / my_delta

deltas = [1, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001]

for delta in deltas:
    print(diff_demo(f, 2, delta))
```

```
7.0
6.099999999999994
6.009999999999849
6.000999999999479
6.000100000012054
6.000009999951316
6.000001000927568
```

```
for delta in deltas:
    print(diff_demo(f, -1, delta))
```

```
1.0
0.10000000000000009
0.009999999999998899
0.001000000000139778
9.99999993922529e-05
1.000000082740371e-05
1.000088900582341e-06
```

Return to slides here.

Segment 3: Automatic Differentiation

TensorFlow and **PyTorch** are the two most popular automatic differentiation libraries.

Let's use them to calculate dy/dx at $x = 5$ where:

$$y = x^2$$

$$\frac{dy}{dx} = 2x = 2(5) = 10$$

Autodiff with PyTorch

```
import torch
```

```
x = torch.tensor(5.0)
```

```
x
```

```
tensor(5.)
```

```
x.requires_grad_() # contagiously track gradients through forward pass
```

```
tensor(5., requires_grad=True)
```

```
y = x**2
```

```
y.backward() # use autodiff
```

```
x.grad
```

```
tensor(10.)
```

Autodiff with TensorFlow

```
import tensorflow as tf
```

```
x = tf.Variable(5.0)
```

```
with tf.GradientTape() as t:
    t.watch(x) # track forward pass
    y = x**2
```

```
t.gradient(y, x) # use autodiff
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=10.0>
```

Return to slides here.

As usual, PyTorch feels more intuitive and pythonic than TensorFlow. See the standalone *Regression in PyTorch* notebook for an example of autodiff paired with gradient descent in order to fit a simple regression line.