# Autodifferentiation

Written by:

- Tony Menzo (School of Physics, University of Cincinnati)

In this tutorial we will build our own simplified autodifferentiation engine.

All modern machine learning paradigms rely on constructing complex expressions from a sequence of differentiable operations. These expressions often involve anywhere from $\mathcal{O}(\text{tens})$ to $\mathcal{O}(\text{billions})$ of operations and parameters. To efficiently compute gradients, the entire computation must be organized to enable low-overhead access to derivative information from the outset. This process (tracking derivatives throughout a computation) is known as **automatic differentiation**, or autodifferentiation (*autodiff* for short). The prefix auto- reflects the fact that the derivatives of each operation are made available automatically during execution.

Software systems that implement this functionality are called **autodifferentiation engines**, and all modern differentiable programming libraries are built around them. As we will see, the design choices behind these engines significantly shape how differential libraries are used in practice.

## Requirements

This notebook requires a few external dependencies which are imported here. First we set up our plotting with `matplotlib`.

```
import matplotlib.pyplot as plt
```

Additionally we'll use numpy as a general purpose array operation library.

```
# Import the `numpy` module.
import numpy as np
```

Finally, for nice visualization of computation trees we'll use an auxillary library called `graphviz`.

```
# If Graphviz is uninstalled:
# !pip install graphviz

from graphviz import Digraph
```

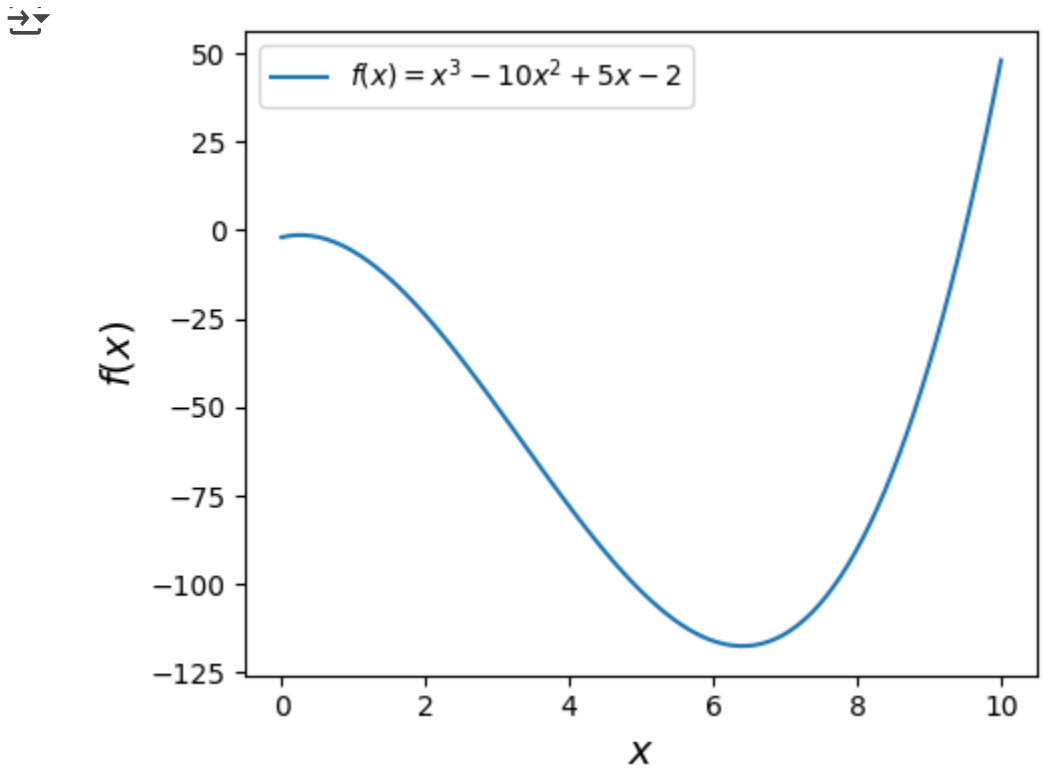
## Analytic versus numerical differentiation

## Analytic versus numerical differentiation

Differentiation should be a familiar concept - let's start by considering the following one-dimensional function:

$$f(x) = x^3 - 10x^2 + 5x - 2$$

```
# Define the function
f = lambda x: x**3 - 10 * x**2 + 5 * x - 2

# Define a range of x values
x = np.linspace(0, 10, 100)
y = f(x)

# Plot the function
fig, ax = plt.subplots(1, 1, figsize=(5, 4))
ax.plot(x, y, label=r"$f(x) = x^3 - 10x^2 + 5x - 2$")
ax.set_xlabel(r"$x$", fontsize=14)
ax.set_ylabel(r"$f(x)$", fontsize=14)
ax.legend()
fig.tight_layout()
```

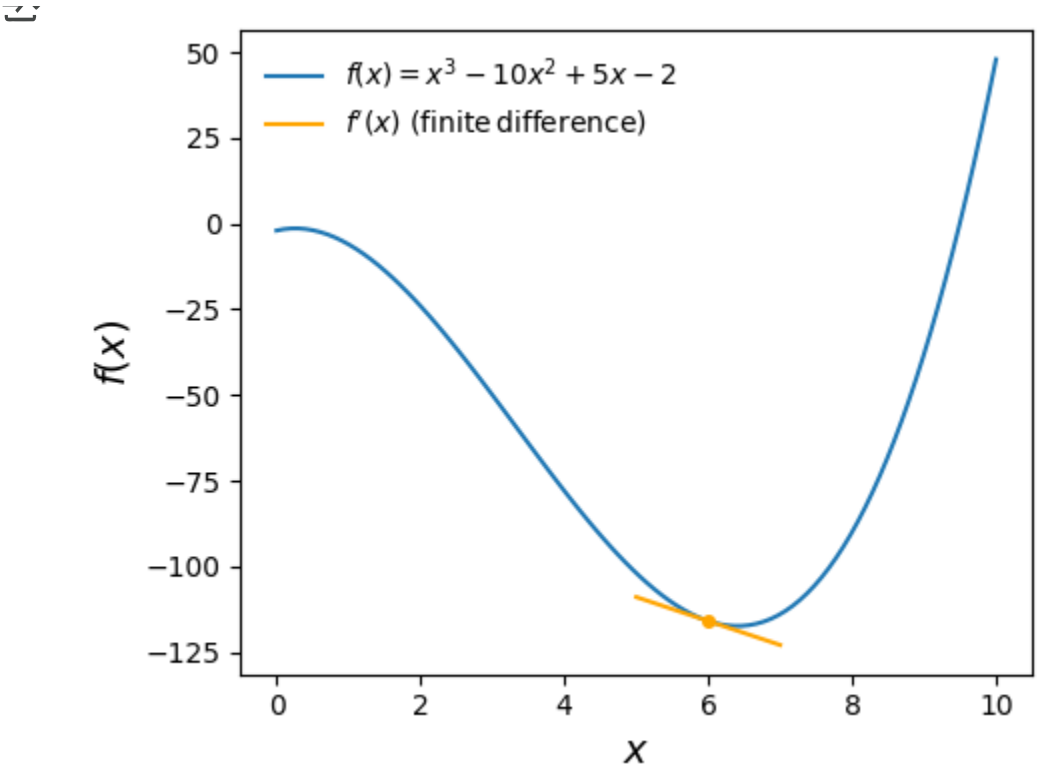



What is the derivative of this function with respect to $x$? This amounts to understanding the "slope" at a given point. The first way we learn to do this is via finite difference

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

which can be implemented practically as follows:

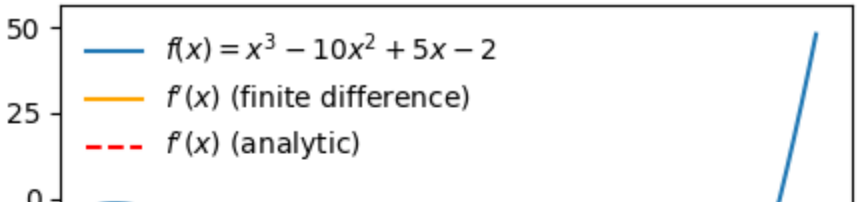
```
# Define the finite difference derivative
finite_diff_derivative = lambda f, x, h: (f(x + h) - f(x)) / h

# Set the step size
h = 1e-5
# Compute the derivative at a specific point
x0 = 6
df = finite_diff_derivative(f, x0, h)

# Initialize figure and axis for plotting
fig, ax = plt.subplots(1, 1, figsize=(5, 4))

# Plot the original function
ax.plot(x, y, label=r"$f(x) = x^3 - 10x^2 + 5x - 2$")

# Plot the tangent line
x_tangent_finite_diff = np.linspace(x0 - 1, x0 + 1, 100)
y_tangent_finite_diff = f(x0) + df * (x_tangent_finite_diff - x0)
ax.plot(
    x_tangent_finite_diff,
    y_tangent_finite_diff,
    color="orange",
    label=r"$f'(x)$ $(\mathrm{finite \, difference})$",
)
ax.plot(x0, f(x0), "o", ms=4, color="orange")

# Set labels and legend
ax.set_xlabel(r"$x$", fontsize=14)
ax.set_ylabel(r"$f(x)$", fontsize=14)
ax.legend(frameon=False)
fig.tight_layout()
```

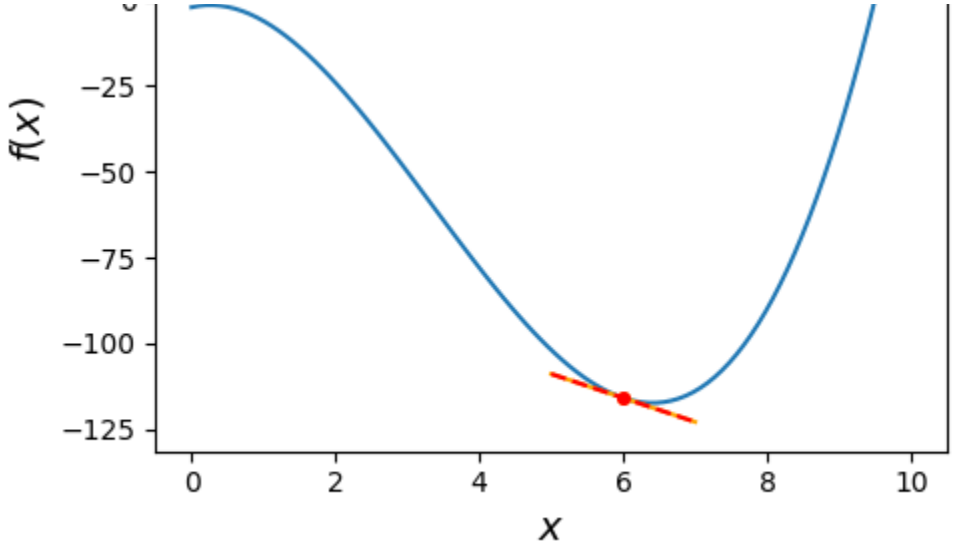


Of course, we also know that the derivative of this function can be computed completely analytically

$$\frac{df}{dx} = 3x^2 - 20x + 5$$

```
# Define the analytic derivative
analytic_derivative = lambda x: 3 * x**2 - 20 * x + 5

# Compute the derivative at a specific point
x0 = 6
df_analytic = analytic_derivative(x0)

# Initialize figure and axis for plotting
fig, ax = plt.subplots(1, 1, figsize=(5, 4))

# Plot the original function
ax.plot(x, y, label=r"$f(x) = x^3 - 10x^2 + 5x - 2$")

# Plot the tangent line
x_tangent_analytic = np.linspace(x0 - 1, x0 + 1, 100)
y_tangent_analytic = f(x0) + df_analytic * (x_tangent_analytic - x0)

# Finite difference
ax.plot(
    x_tangent_finite_diff,
    y_tangent_finite_diff,
    color="orange",
    label=r"$f'(x)$ (finite difference)",
)
ax.plot(x0, f(x0), "o", ms=4, color="orange")
# Analytic
ax.plot(
    x_tangent_analytic,
    y_tangent_analytic,
    "--",
    color="red",
    label=r"$f'(x)$ (analytic)",
)
ax.plot(x0, f(x0), "o", ms=4, color="red")

# Set labels and legend
ax.set_xlabel(r"$x$", fontsize=14)
ax.set_ylabel(r"$f(x)$", fontsize=14)
ax.legend(frameon=False)
fig.tight_layout()
```






## Building `Scalar` autodiff

In the remainder of this tutorial, we will slowly build up a simplified autodiff engine from scratch. We will restrict ourselves to scalar functions (functions which map arbitrary dimensional input to a single scalar value $f(\vec{x}) \rightarrow \mathbb{R}$). To start let's build up a new class called `Scalar` that will handle and define the core organization of our autodiff engine.

```
class Scalar:
    def __init__(self, data):
        self.data = data

    def __repr__(self):
        return f"Scalar({self.data})"


a = Scalar(1.0)
print(a)
```

```
Scalar(1.0)
```

Above, our class only has the ability to define new data types - if we also want to perform operations on these class instances, we can define them using the dunder methods (https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types). For example, we can define the `Scalar` "addition" operation using the `__add__` method:

```
class Scalar:
    def __init__(self, data):
        self.data = data

    def __repr__(self):
        return f"Scalar({self.data})"
```

```
    def __add__(self, other):
        if isinstance(other, Scalar):
            return Scalar(self.data + other.data)
        else:
            return Scalar(self.data + other)


# Now we can add two Scalar instances
a = Scalar(1.0)
b = Scalar(2.0)
c = a + b
print(c)
```

```
Scalar(3.0)
```

Let's add some more common utility operations that we may want to use ( `__sub__`, `__mul__`, for subtraction and multiplication, respectively)

```
class Scalar:
    def __init__(self, data):
        self.data = data

    def __repr__(self):
        return f"Scalar({self.data})"

    def __add__(self, other):
        if isinstance(other, Scalar):
            return Scalar(self.data + other.data)
        else:
            return Scalar(self.data + other)

    def __sub__(self, other):
        if isinstance(other, Scalar):
            return Scalar(self.data - other.data)
        else:
            return Scalar(self.data - other)

    def __mul__(self, other):
        if isinstance(other, Scalar):
            return Scalar(self.data * other.data)
        else:
            return Scalar(self.data * other)


# Check that the dunder operations work as expected
a = Scalar(1.0)
b = Scalar(2.0)
c = a + b
d = a - b
e = c * d
f = e * b
```

```
print("c = a + b =>", a.data, "+", b.data, "=", c.data)
print("d = a - b =>", a.data, "-", b.data, "=", d.data)
print("e = c * d =>", c.data, "*", d.data, "=", e.data)
print("f = e * b =>", e.data, "*", b.data, "=", f.data)
```

```
c = a + b => 1.0 + 2.0 = 3.0
d = a - b => 1.0 - 2.0 = -1.0
e = c * d => 3.0 * -1.0 = -3.0
f = e * b => -3.0 * 2.0 = -6.0
```

## Tracing and computation graphs

Above, we iteratively constructed the quantity `f`:

```
f = b * ((a + b) * (a - b))
```

through a set of successive operations. In the end, we want to be able to understand how a change in any of the values in each operation may effect the final output value. In other words, in order to understand how the variables (`a` and `b`) influence the final *expression* (`f`) we need to track which operations are used in constructing the expression (also known as constructing a ***computational graph***). In differential programming, the process of recording operations and generating a computational graph is known as ***tracing***.

In our implementation, we'll trace the preceding operations via the iterable `set()` function. In Python the `set()` function stores an unordered collection of unique and immutable (cannot be changed in-place) objects. In the context of tracing, `set()` is nice because it ensures each node contains a unique set of operations, however, this is a convention - Python lists or tuples could also be used.

```
class Scalar:
    def __init__(self, data, _children=(), _op=""):
        self.data = data
        self._prev = set(_children)
        self._op = _op

    def __repr__(self):
        return f"Scalar({self.data})"

    def __add__(self, other):
        return Scalar(self.data + other.data, _children=(self, other), _op="+")

    def __sub__(self, other):
        return Scalar(self.data - other.data, _children=(self, other), _op="-")

    def __mul__(self, other):
        return Scalar(self.data * other.data, _children=(self, other), _op="*")
```

```
# f = b * ((a + b) * (a - b))
a = Scalar(1.0)
b = Scalar(2.0)
c = a + b
d = a - b
e = c * d
f = e * b

# Now we have access to our computation graph
# Print in the syntax ({input_1, input_2}, operation) = value
print("(", f._prev, ",", f._op, ") =", f)
print("(", e._prev, ",", e._op, ") =", e)
print("(", d._prev, ",", d._op, ") =", d)
print("(", c._prev, ",", c._op, ") =", c)
print()

# Note the a and b do not have children, values without children
# are commonly referred to as "leaf nodes" in the computation graph
print("(", a._prev, ",", a._op, ") =", a)
print("(", b._prev, ",", b._op, ") =", b)
print()

# Note that we should be cautious to only use binary operations when set() is used,
# information about the computation graph can easily be lost -- for example:
g = a + b + a
print("(", g._prev, ",", g._op, ") =", g)
```

```
     ( {Scalar(-3.0), Scalar(2.0)} , * ) = Scalar(-6.0)
     ( {Scalar(-1.0), Scalar(3.0)} , * ) = Scalar(-3.0)
     ( {Scalar(1.0), Scalar(2.0)} , - ) = Scalar(-1.0)
     ( {Scalar(1.0), Scalar(2.0)} , + ) = Scalar(3.0)

     ( set() ,  ) = Scalar(1.0)
     ( set() ,  ) = Scalar(2.0)

     ( {Scalar(3.0), Scalar(1.0)} , + ) = Scalar(4.0)
```

```
# It can be useful to output the computation graph
def graph(scalar):
    """
    Recursively prints the computation graph of a Scalar object.
    """
    if scalar._prev:
        print(f"({scalar._prev}, '{scalar._op}') = {scalar.data}")
        for child in scalar._prev:
            graph(child)
    else:
        print(f"Leaf node, {scalar.data}")


# f = b * ((a + b) * (a - b))
a = Scalar(1.0)
b = Scalar(2.0)
```

```
b = Scalar(2.0)
c = a + b
d = a - b
e = c * d
f = e * b

# Now we can "see" the graph
graph(f)
```

```
({Scalar(-3.0), Scalar(2.0)}, '*') = -6.0
({Scalar(3.0), Scalar(-1.0)}, '*') = -3.0
({Scalar(1.0), Scalar(2.0)}, '+') = 3.0
Leaf node, 1.0
Leaf node, 2.0
({Scalar(1.0), Scalar(2.0)}, '-') = -1.0
Leaf node, 1.0
Leaf node, 2.0
Leaf node, 2.0
```

For a more powerful and informative computation graph output, we can add optional labels to each `Scalar()` instance:

```
class Scalar:
    def __init__(self, data, _children=(), _op="", label=None):
        self.data = data
        self._prev = set(_children)
        self._op = _op
        if label is not (None):
            self.label = label

    def __repr__(self):
        return f"Scalar({self.data})"

    def __add__(self, other):
        return Scalar(self.data + other.data, _children=(self, other), _op="+")

    def __sub__(self, other):
        return Scalar(self.data - other.data, _children=(self, other), _op="-")

    def __mul__(self, other):
        return Scalar(self.data * other.data, _children=(self, other), _op="*")


def graph(scalar):
    """
    Recursively prints the computation graph of a Scalar object.
    """
    if scalar._prev:
        print(f"{scalar.label} = ({scalar._prev}, '{scalar._op}') = {scalar.data}")
        for child in scalar._prev:
            graph(child)
    else:
        print(f"Leaf node, {scalar.label} = {scalar.data}")
```

```
# f = b * ((a + b) * (a - b))
a = Scalar(1.0)
a.label = "a"
b = Scalar(2.0)
b.label = "b"
c = a + b
c.label = "c"
d = a - b
d.label = "d"
e = c * d
e.label = "e"
f = e * b
f.label = "f"

# Now we can see the graph
graph(f)
```

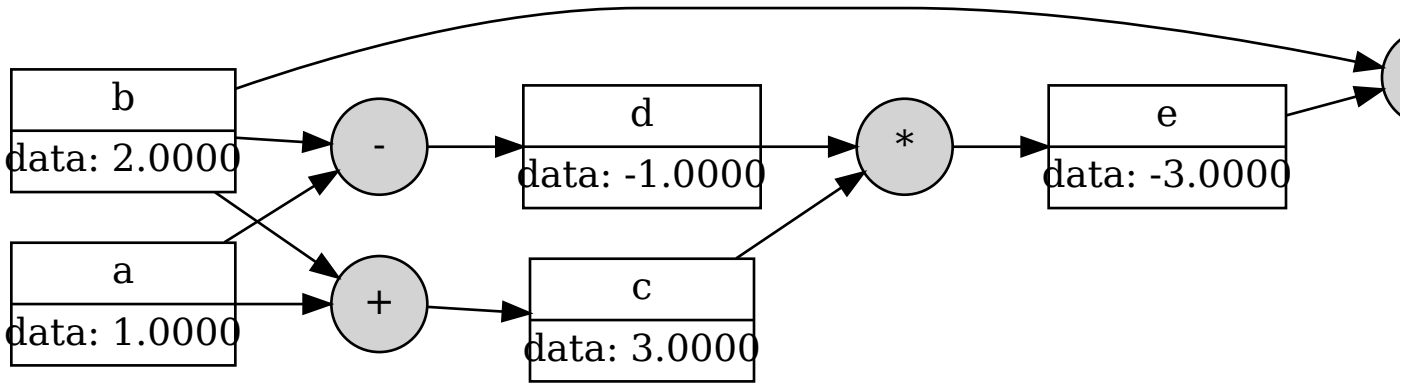
```
    f = ({Scalar(2.0), Scalar(-3.0)}, '*') = -6.0
    Leaf node, b = 2.0
    e = ({Scalar(3.0), Scalar(-1.0)}, '*') = -3.0
    c = ({Scalar(1.0), Scalar(2.0)}, '+') = 3.0
    Leaf node, a = 1.0
    Leaf node, b = 2.0
    d = ({Scalar(1.0), Scalar(2.0)}, '-') = -1.0
    Leaf node, a = 1.0
    Leaf node, b = 2.0
```

The above "graph" is rudimentary at best, it would be much nicer to have a full visualization. Without going into details, we can define the following helper functions:

```
def trace(scalar):
    """
    Trace the computation graph of the Scalar object and return nodes and edges.
    """
    nodes, edges = set(), set()

    def build(v):
        if v not in nodes:
            nodes.add(v)
            for child in v._prev:
                edges.add((child, v))
                build(child)

    build(scalar)
    return nodes, edges


def visualize_graph(scalar):
    """
    Visualize the computation graph of the Scalar object using Graphviz.
```

```
    """
    nodes, edges = trace(scalar)
    graph = Digraph(format="svg", graph_attr={"rankdir": "LR"})

    for n in nodes:
        graph.node(
            name=str(id(n)),
            label=f"{n.label} | data: {n.data:.4f}",
            shape="record",
            style="filled",
            fillcolor="white",
        )
        if n._op:
            graph.node(
                name=str(id(n)) + n._op,
                label=n._op,
                shape="circle",
                facecolor="lightgray",
                style="filled",
            )
            graph.edge(str(id(n)) + n._op, str(id(n)))

    for n1, n2 in edges:
        graph.edge(str(id(n1)), str(id(n2)) + n2._op)

    return graph
```

```
# We can now output a nice visualization of the computation graph
visualize_graph(f)
```




## The goal of autodiff

Now that we have a nice visualization of the computation graph, we can start to answer the question we've been aiming for since the first construction of the `Scalar()` class:

**"How does a change in the values `a`, `b`, `c`, `d`, or `e` influence the value of `f`?"**

There is a very crude way to understand this question: increase the value of interest by a small amount

and see how the value of `f` changes.

```
# Define step size
h = 1e-5

# Default value of f
a = Scalar(1.0)
b = Scalar(2.0)
c = a + b
d = a - b
e = c * d
f1 = e * b

# Change in f with respect to a
a = Scalar(1.0 + h)
b = Scalar(2.0)
c = a + b
d = a - b
e = c * d
f2 = e * b
print("Change in f with respect to a:", (f2.data - f1.data) / h)

# Change in f with respect to b
a = Scalar(1.0)
b = Scalar(2.0 + h)
c = a + b
d = a - b
e = c * d
f2 = e * b
print("Change in f with respect to b:", (f2.data - f1.data) / h)

# Change in f with respect to c
a = Scalar(1.0)
b = Scalar(2.0)
c = a + b
c += Scalar(h)
d = a - b
e = c * d
f2 = e * b
print("Change in f with respect to c:", (f2.data - f1.data) / h)

# Change in f with respect to d
a = Scalar(1.0)
b = Scalar(2.0)
c = a + b
d = a - b
d += Scalar(h)
e = c * d
f2 = e * b
print("Change in f with respect to d:", (f2.data - f1.data) / h)

# Change in f with respect to e
```

```
a = Scalar(1.0)
b = Scalar(2.0)
c = a + b
d = a - b
e = c * d
e += Scalar(h)
f2 = e * b
print("Change in f with respect to e:", (f2.data - f1.data) / h)

# Change in f with respect to f
a = Scalar(1.0)
b = Scalar(2.0)
c = a + b
d = a - b
e = c * d
f2 = e * b
f2 += Scalar(h)
print("Change in f with respect to f:", (f2.data - f1.data) / h)
```

```
Change in f with respect to a: 4.00002000002786
Change in f with respect to b: -11.000060000210253
Change in f with respect to c: -2.0000000000131024
Change in f with respect to d: 5.9999999999504885
Change in f with respect to e: 2.0000000000131024
Change in f with respect to f: 0.9999999999621422
```

Clearly what we have done above is just taking the derivative of `f` with respect to each of it's dependent parameters. The above example would may make it seem like this would be a perfectly good way to do things, however, I hope it's clear that this "manual" or finite-difference prescription above is not scaleable. Take, for instance, an expression with 1,000 leaf nodes (independent variables)!

Ideally we'd like to keep the exact gradient value for each parameter either as the computation graph is built (forward-mode autodifferentiation) or by traversing backwards after the graph is constructed (reverse-mode autodifferentiation). Because many machine learning tasks and neural networks rely heavily on the latter, we will mainly focus on reverse-mode autodiff and comment briefly on forward mode at the end of this notebook.

## Reverse-mode autodiff

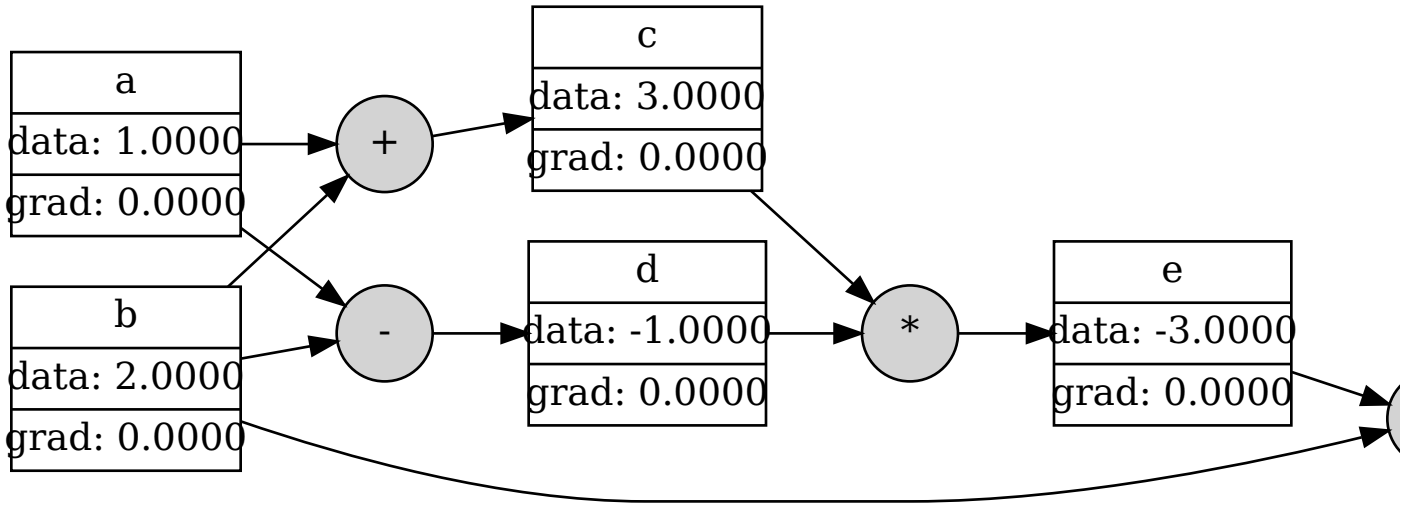
Reverse-mode autodiff works by first storing the dependencies of the expression tree in memory (tracing), typically referred to as the ***forward-pass***, followed by a reverse or ***backward-pass*** through the computation graph in which partial derivatives of the final output with respect to each intermediate variables (adjoints) are computed.

Up to now we have fully implemented the forward-pass, specifically tracing the operations constructing the computational graph. For the backward-pass, the main goal is to compute gradients for each `Scalar` instance in the computation graph. This requires each instance to have a gradient

(`self.grad`) attribute which we will conventionally initialize to 0.

```
class Scalar:
    def __init__(self, data, _children=(), _op="", label=None):
        self.data = data
        self.grad = 0.0
        self._prev = set(_children)
        self._op = _op
        if label is not (None):
            self.label = label

    def __repr__(self):
        return f"Scalar({self.data})"

    def __add__(self, other):
        return Scalar(self.data + other.data, _children=(self, other), _op="+")

    def __mul__(self, other):
        return Scalar(self.data * other.data, _children=(self, other), _op="*")

    def __sub__(self, other):
        return Scalar(self.data - other.data, _children=(self, other), _op="-")


def visualize_graph(scalar):
    """
    Visualize the computation graph of the Scalar object using Graphviz.
    """
    nodes, edges = trace(scalar)
    graph = Digraph(format="svg", graph_attr={"rankdir": "LR"})

    for n in nodes:
        graph.node(
            name=str(id(n)),
            label=f"{n.label} | data: {n.data:.4f} | grad: {n.grad:.4f}",
            shape="record",
            style="filled",
            fillcolor="white",
        )
        if n._op:
            graph.node(
                name=str(id(n)) + n._op,
                label=n._op,
                shape="circle",
                facecolor="lightgray",
                style="filled",
            )
            graph.edge(str(id(n)) + n._op, str(id(n)))

    for n1, n2 in edges:
        graph.edge(str(id(n1)), str(id(n2)) + n2._op)
```

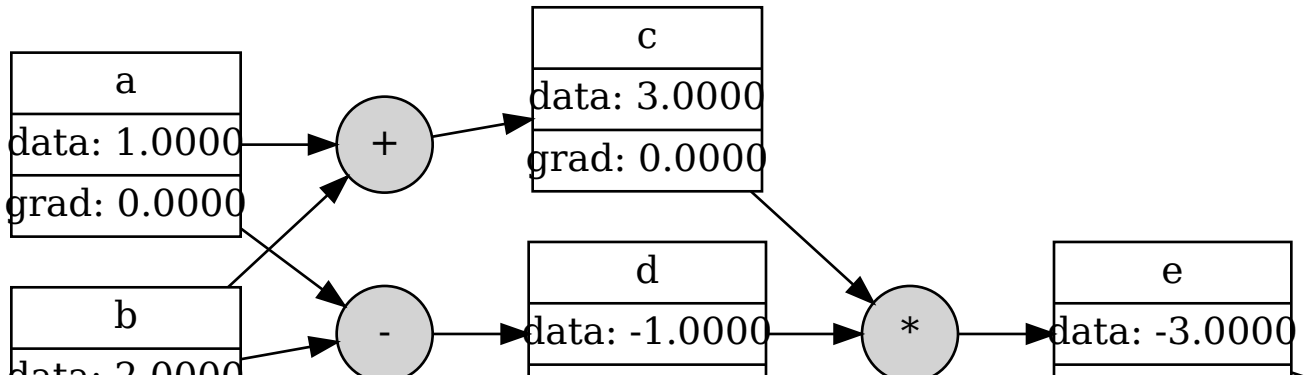
```
    return graph

# Now we have access to our computation graph
a = Scalar(1.0, label="a")
b = Scalar(2.0, label="b")
c = a + b
c.label = "c"
d = a - b
d.label = "d"
e = c * d
e.label = "e"
f = e * b
f.label = "L"

visualize_graph(f)
```

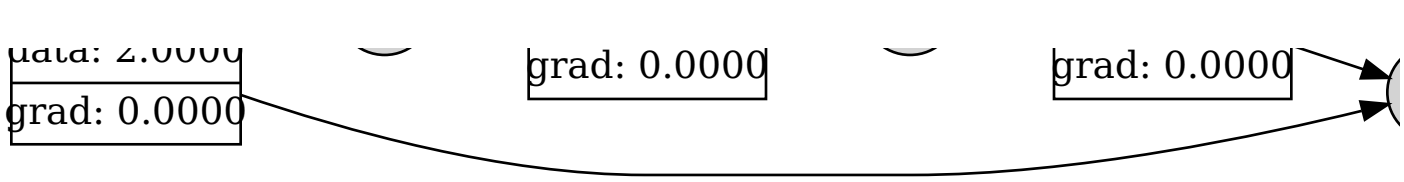



It is instructive to go through each node manually first, and then think about ways to automate the process. The simplest node is the output node (and the one, at the end of the day, that we will always set manually as a seed for the rest of the gradients to grow off of)

$$\frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1.$$

```
f.grad = 1.0
visualize_graph(f)
```





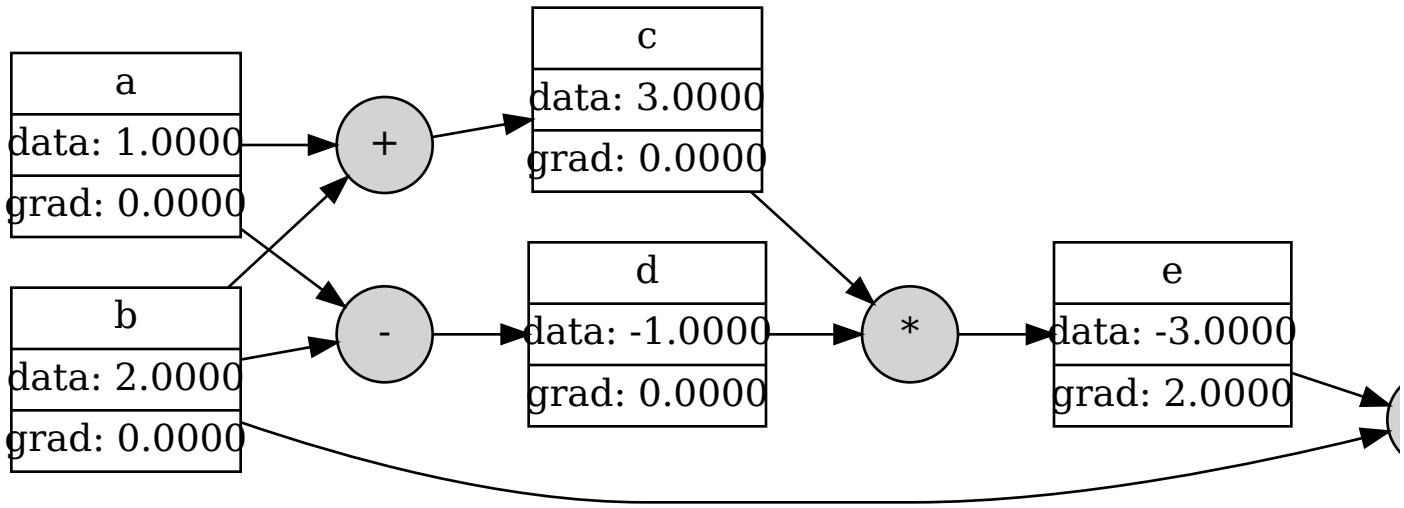

Now we can perform what would typically be called the first `backward` call (typically referred to as calling "backward" on `L` and what will be called `L._backward` later), specifically, we'd like to set the gradients of each of the nodes that flow into the output node, i.e. `e` and `b`. The gradient with respect to `e` is trivial if we remember our rules for differentiation from kindergarten

$$\frac{\partial \mathcal{L}}{\partial e} = b,$$

the derivative with respect to `b`, on the other hand, is slightly less trivial -- we will return to this once we have made our way through the whole graph.

For now, let's set the gradient for `e`:

```
e.grad = b.data
visualize_graph(f)
```




Now we have reached a key crossroad in autodifferentiation. If you understand how to compute the gradient at these nodes, you more or less understand how all neural networks work. We want to compute the gradient of $\mathcal{L}$ with respect to `c` and `d`. The key to the whole castle rests in the hands of the chain rule

$$\texttt{c.grad} = \frac{\partial \mathcal{L}}{\partial c} = \frac{\partial \mathcal{L}}{\partial e}\frac{\partial e}{\partial c},$$

remember that we have already traversed the "`e`" node when we performed the backward call on `L`, we already have access to $\partial \mathcal{L}/\partial e$ through `e`'s `self.grad`! All we need to compute is the "local" derivative of `e` with respect to `c`

$$\frac{\partial e}{\partial c} = d$$

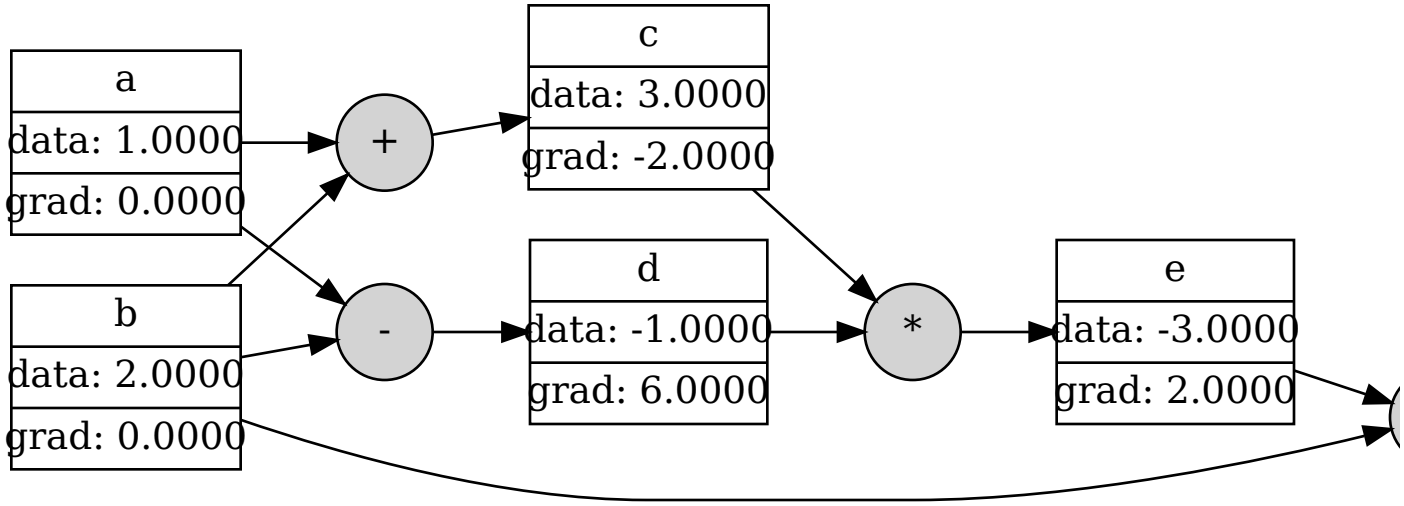
which, when combined with `e.grad` gives

$$\frac{\partial \mathcal{L}}{\partial c} = b \times d.$$

Likewise, for `d` we have

$$\texttt{d.grad} = \frac{\partial \mathcal{L}}{\partial d} = \frac{\partial \mathcal{L}}{\partial e}\frac{\partial e}{\partial d} = b \times c.$$

```
c.grad = d.data * b.data
d.grad = c.data * b.data
visualize_graph(f)
```




Let's now take a look at the final two gradients, the gradients `L` with respect to `a` and `b` can be found following the same prescription as above but with a small twist. We continue with the chain rule

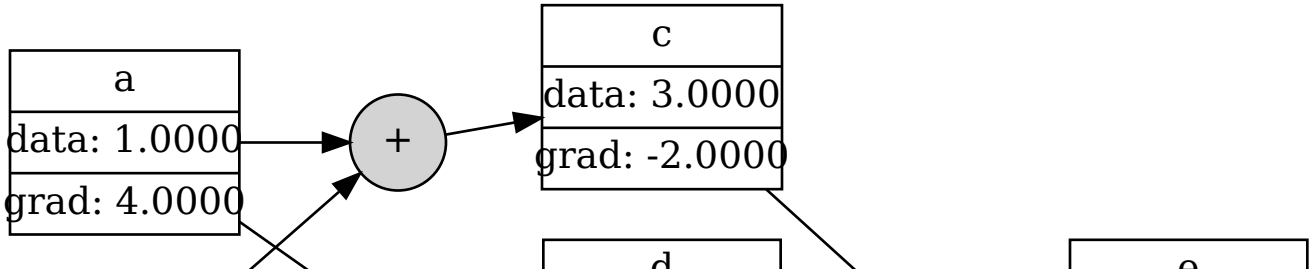
$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial e}\frac{\partial e}{\partial a}$$

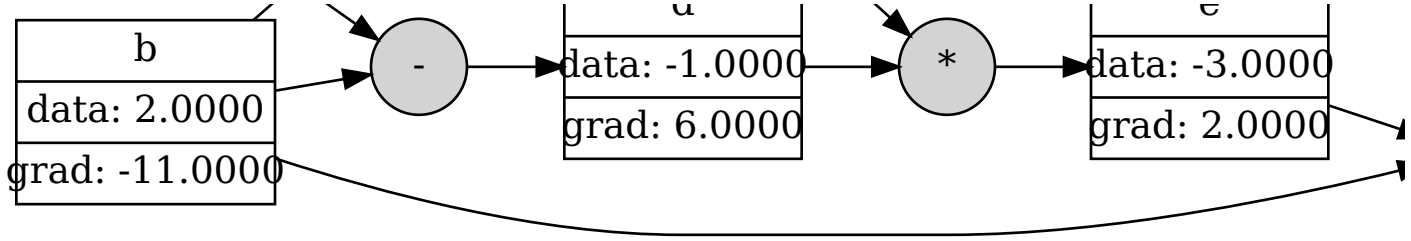
however, in this case, `e` is a function of two parameters, `e = c(a) * d(a)`, each with a dependence on `a`. Luckily, we also know how differentiation behaves with respect to composite functions

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial e}\left(\frac{\partial e}{\partial c}\frac{\partial c}{\partial a} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial a}\right) = \frac{\partial \mathcal{L}}{\partial c}\frac{\partial c}{\partial a} + \frac{\partial \mathcal{L}}{\partial d}\frac{\partial d}{\partial a} = b(c+d) = 2ab$$

and we again see that we only need to compute the "local" derivatives $\partial c/\partial a$ and $\partial d/\partial a$.

```
a.grad = 2 * a.data * b.data
b.grad = -3 * b.data**2 + a.data**2
visualize_graph(f)
```






## Exercise: verify gradient

Verify:

$$\frac{\partial \mathcal{L}}{\partial b} = a^2 - 3b^2.$$

$$\frac{\partial \mathcal{L}}{\partial b} = e\frac{\partial b}{\partial b} + b\frac{\partial e}{\partial b} = e + b\left(\frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b}\right) = (a-b)(a+b) + b(c-d) = a^2 - 3b^2$$

## Exercise: implementation choice for sub

a. Why don't we want to implement `__sub__` as it's own method with it's own `_backward` function?

Hint: A similar reason would arise when implementing, for example, division.

b. What item in the `Scalar` constructor would we need to modify if we wanted to fix the problem presented above?

Non-commutativity! The order in which the edges feed into the operation node matters. This is definitely an issue when using `set()`'s to track operations. A different design decision one could think about implementing would be to use lists or tuples (as mentioned earlier).

# Automated backward-pass

Now we'd like to embed the backwards pass into our `Scalar` class. From our setup, each `Scalar` instance has it's own "local" operation through `_op` as well as the inputs through `_children` (`_prev`). During the backward-pass, we need to assign each instance a "derivative-function" that will compute the gradient of the inputs with respect to the outputs and assign these gradients to the inputs (`_children`) when the backward function is called on the output. These "backward" functions are operation dependent and therefore should be implemented in the dunder methods.

The main things to remember are that:

1. At any given position in the computation graph during the backward pass we always have access

to the gradient of the output with respect to the forward-adjacent node.

2. the chain rule tells us that if we have access to the gradient of the forward-adjacent node, all we need is the derivative of the local operation (the derivative of the current output with respect to its inputs).

Let's work this out explicitly for `Scalar` with a simple example of adding.

There are two cases to consider, the first case is if we are distributing gradients from the final output (remember that the backward-pass starts by assigning the gradient of the final output to 1), `output = a + b`, in which case the gradients of the inputs are trivial

$$\frac{\partial \texttt{output}}{\partial \texttt{a}} = \frac{\partial (\texttt{a + b})}{\partial \texttt{a}} = 1.$$

The second relevant situation is anywhere inside the computation graph (say, for example, one beyond adjacent to the output). Take, for example,

```
a = b + c
output = a * d.
```

We're interested in the derivative of the final `output` with respect to `b` (which we will compute later in `Scalar` by calling `a._backward()`) - in this case, we would have already computed the derivative of the `output` with respect to `a`, and thus have access to $\partial$`output`$/\partial$ a

$$\frac{\partial \texttt{output}}{\partial \texttt{b}} = \frac{\partial \texttt{output}}{\partial \texttt{a}} \frac{\partial \texttt{a}}{\partial \texttt{b}} = \frac{\partial \texttt{output}}{\partial \texttt{a}} \frac{\partial (\texttt{b + c})}{\partial \texttt{b}} = \frac{\partial \texttt{output}}{\partial \texttt{a}}.$$

In both cases, the backwards pass needs to assign the same gradient.

Take a moment and convince yourself of two things:

1. The argument above applies no matter the position in the computation graph.
2. The same result also applies for the gradients of the output with respect to the `other` output (`c` in the second example). Both cases can be accommodated by including a `_backward` function to the mother (output in the forward pass) which assigns gradients to the daughters (inputs during the forward pass) as:

```
def __add__(self, other):
 output = Scalar(self.data + other.data, _children = (self, other), _op = '+')
 def _backward():
     self.grad += output.grad
     other.grad += output.grad
 output._backward = _backward
 return output
```

It may seem weird that during the backward pass we can call `_backward` on an output node and have this assign gradient values to its children by calling `self.grad` and `other.grad`. Why does the output, which is its own `Scalar` instance with its own `self` variables, remember its inputs without

output, which is its own `Scalar` instance with its own `self` variables, remember its inputs without referring to `_children`? This is due to Python's ***closure*** abilities.

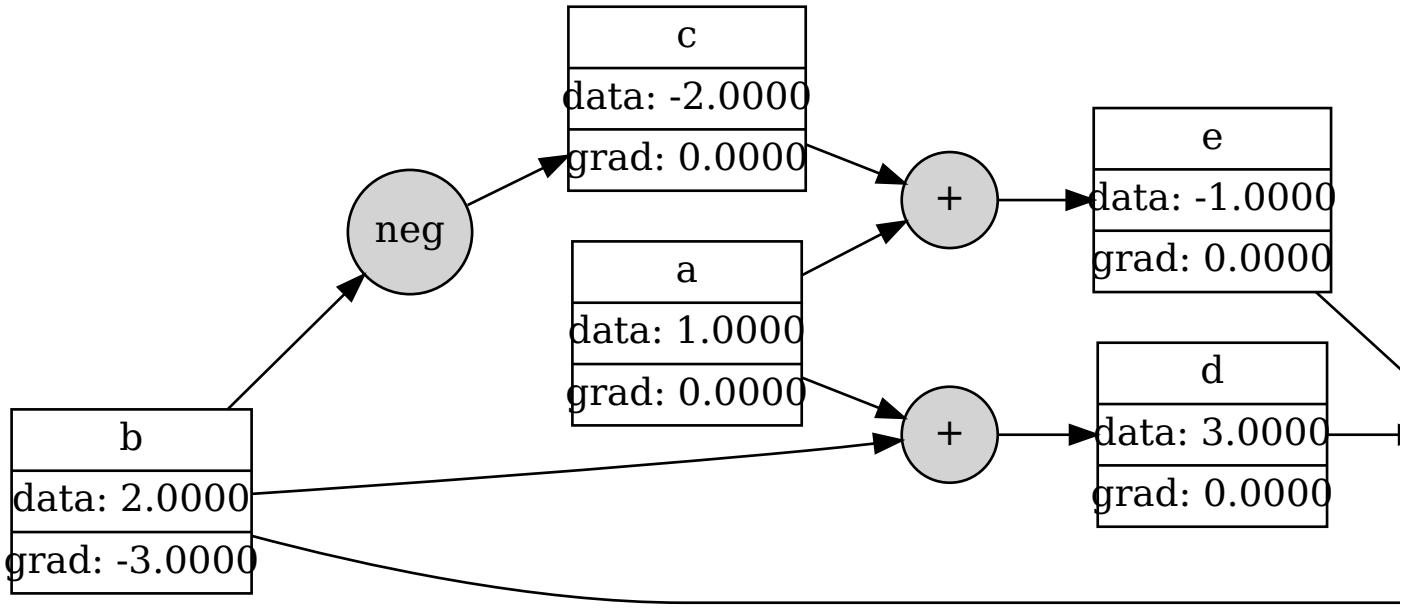
A **closure** in Python is a function that "remembers" the variables from the scope in which it was created, even after that scope is done executing. When we define the `_backward` function inside a method like `__add__` or `__mul__`, the function captures the variables from its enclosing scope—specifically, the `self`, `other`, and `output` objects. This means that even after the `__add__` or `__mul__` method has returned and the output node is being used elsewhere, the `_backward` function still has access to the original input nodes (`self` and `other`). This is why, when we call `output._backward()`, the function can update the gradients of the input nodes directly (e.g., `self.grad += output.grad`), even though we're calling it from the context of the output node. The closure "remembers" the references to the input nodes.

## Exercise: implement multiplication

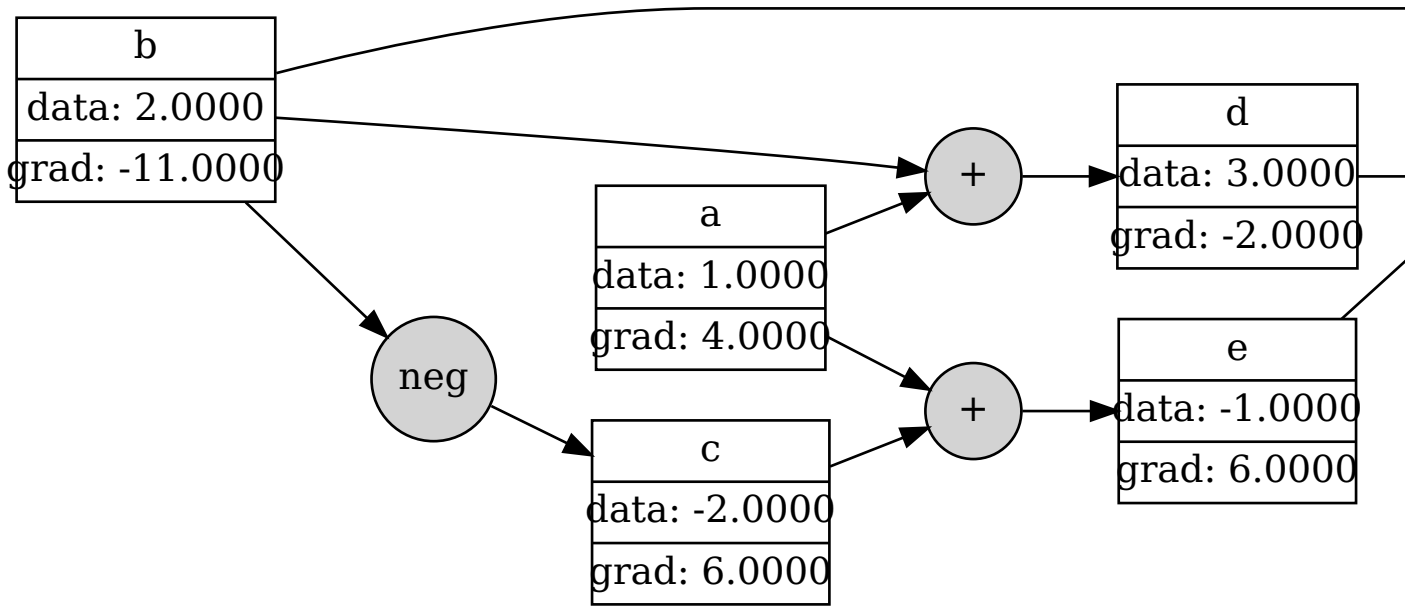
Following the arguments above, what should the `_backward` function be for the `__mul__` operation?

```
class Scalar:
    def __init__(self, data, _children=(), _op="", label=None):
        self.data = data
        self.grad = 0.0
        self._prev = set(_children)
        self._backward = lambda: None
        self._op = _op
        if label is not (None):
            self.label = label

    def __repr__(self):
        return f"Scalar({self.data}, grad = {self.grad})"

    def __add__(self, other):
        output = Scalar(self.data + other.data, _children=(self, other), _op="+")

        def _backward():
            self.grad += output.grad
            other.grad += output.grad

        output._backward = _backward
        return output

    def __mul__(self, other):
        output = Scalar(self.data * other.data, _children=(self, other), _op="*")

        def _backward():
            self.grad += other.data * output.grad
            other.grad += self.data * output.grad
```

```
        output._backward = _backward
        return output

    def __neg__(self):
        output = Scalar(-self.data, _children=(self,), _op="neg")

        def _backward():
            self.grad -= output.grad

        output._backward = _backward
        return output

    def __sub__(self, other):
        return self + (-other)


a = Scalar(1.0, label="a")
b = Scalar(2.0, label="b")
c = -b
c.label = "c"
d = a + b
d.label = "d"
e = a + c
e.label = "e"
f = d * e
f.label = "f"
L = f * b
L.label = "L"

# Manually set the gradient of the final output
L.grad = 1.0
# Look at the first backward iteration
L._backward()
visualize_graph(L)
```



```
a = Scalar(1.0, label="a")
b = Scalar(2.0, label="b")
c = -b
c.label = "c"
d = a + b
d.label = "d"
e = a + c
e.label = "e"
f = d * e
f.label = "f"
L = f * b
L.label = "L"

# String together the full backward pass
L.grad = 1.0
L._backward()
f._backward()
e._backward()
d._backward()
c._backward()

# Visualize the final graph
visualize_graph(L)
```

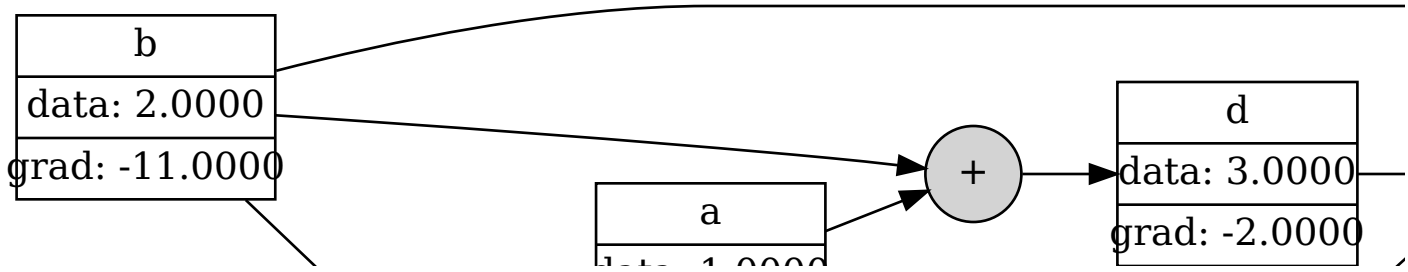


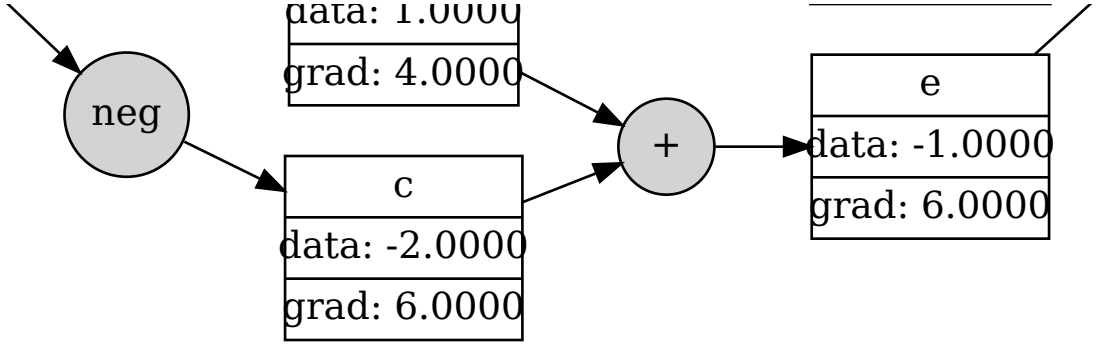

## Topological ordering and the fully automating the backward pass

As we can see above, the order in which we perform the `_backward` call for each node is key to obtaining the correct gradients during the backward pass. The final task is to automate the calls to each `_backward` function in the correct order. The single output scalar function graphs we are

interested have been classified and studied in graph theory - they are called directed acyclic graphs (DAGs) and many algorithms exist for ordering nodes such that they don't become "tangled".

```
def backward(self):
    """
    Recursively build the topological order of the computation graph and perform the ba
    """
    # Topological order all of the children in the graph
    topology = []
    visited = set()

    def build_topology(v):
        if v not in visited:
            visited.add(v)
            for child in v._prev:
                build_topology(child)
            topology.append(v)

    build_topology(self)

    # Manually set the gradient of the final output
    self.grad = 1
    # Iterate one variable at a time
    for v in reversed(topology):
        v._backward()


# Perform forward pass
a = Scalar(1.0, label="a")
b = Scalar(2.0, label="b")
c = -b
c.label = "c"
d = a + b
d.label = "d"
e = a + c
e.label = "e"
f = d * e
f.label = "f"
L = f * b
L.label = "L"

# Perform backward pass
backward(L)
visualize_graph(L)
```

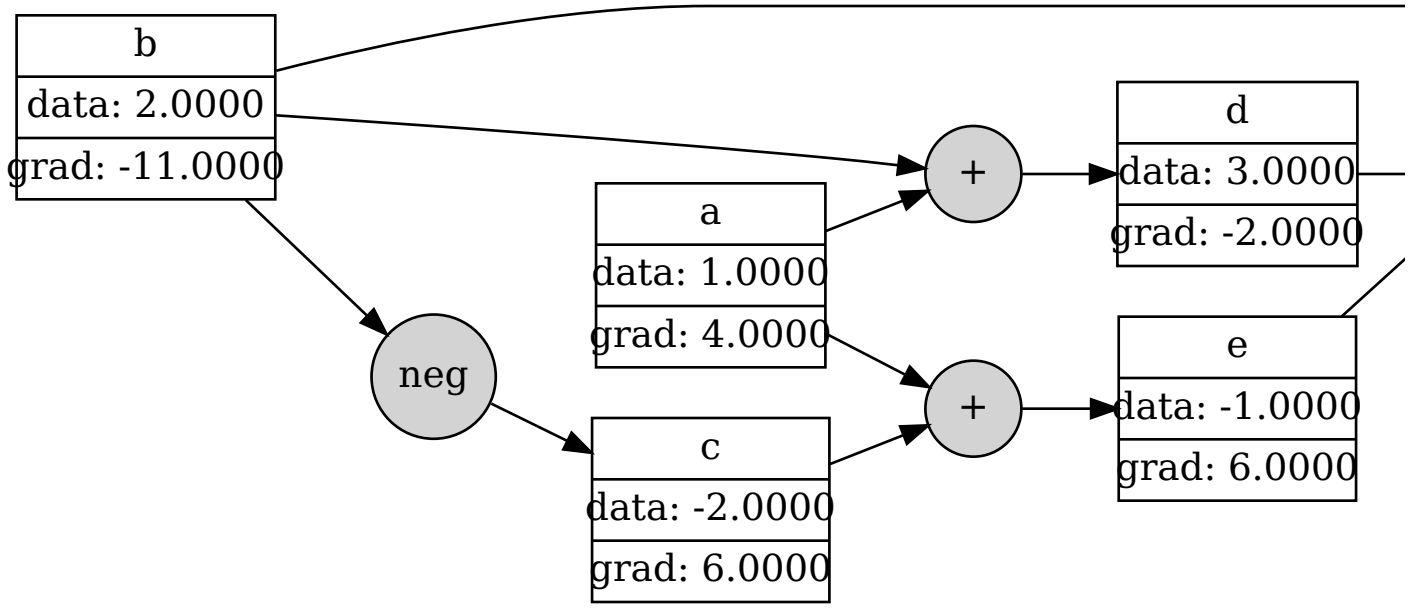





```
class Scalar:
    def __init__(self, data, _children=(), _op="", label=None):
        self.data = data
        self.grad = 0.0
        self._prev = set(_children)
        self._backward = lambda: None
        self._op = _op
        if label is not (None):
            self.label = label

    def __repr__(self):
        return f"Scalar({self.data}, grad = {self.grad})"

    def __add__(self, other):
        if not isinstance(other, Scalar):
            other = Scalar(other)
        output = Scalar(self.data + other.data, _children=(self, other), _op="+")

        def _backward():
            self.grad += output.grad
            other.grad += output.grad

        output._backward = _backward
        return output

    def __mul__(self, other):
        if not isinstance(other, Scalar):
            other = Scalar(other)
        output = Scalar(self.data * other.data, _children=(self, other), _op="*")

        def _backward():
            self.grad += other.data * output.grad
            other.grad += self.data * output.grad

        output._backward = _backward
        return output

    def __neg__(self):
        if not isinstance(self, Scalar):
            self = Scalar(self)
        output = Scalar(-self.data, _children=(self,), _op="neg")
```

```
        def _backward():
            self.grad -= output.grad

        output._backward = _backward
        return output

    def __sub__(self, other):
        if not isinstance(other, Scalar):
            other = Scalar(other)
        output = self + (-other)
        return output

    def backward(self):
        """
        Recursively build the topological order of the computation graph and perform th
        """
        # Topological order all of the children in the graph
        topology = []
        visited = set()

        def build_topology(v):
            if v not in visited:
                visited.add(v)
                for child in v._prev:
                    build_topology(child)
                topology.append(v)

        build_topology(self)

        # Manually set the gradient of the final output
        self.grad = 1
        # Iterate one variable at a time
        for v in reversed(topology):
            v._backward()


# Perform forward pass
a = Scalar(1.0, label="a")
b = Scalar(2.0, label="b")
c = -b
c.label = "c"
d = a + b
d.label = "d"
e = a + c
e.label = "e"
f = d * e
f.label = "f"
L = f * b
L.label = "L"

# Perform backward pass
L.backward()
```

```
visualize_graph(L)
```




## Exercise: check scalar implementation

Pick one of the following functions and check that the `Scalar` class is performing as expected by

1. Writing out the explicit computation graph,
2. Computing the gradients manually, and
3. Using the tools developed above to check your work (visualize the computation graph and double check the backward pass).

$$\mathcal{L} = (a^3 - b^2)a - ab$$
$$\mathcal{L} = (a + b + c)(-a + b + c)(a - b + c)(a + b - c)$$

## Custom operations

In some cases, we may desire to have custom operations as a node within our network that are not available via the traditional dunder methods. For example, in the next tutorial where we build out a neural network using the `Scalar` class, in some examples we will want to terminate our network with something called an **activation function**. For our purposes, this function will just be a simple $\tanh$ but if we want it to jive with the backward-pass, we will need to implement the function itself in the `Scalar` class with instructions on how to pass derivatives to inputs (via the `_backward` method).

For now, we'll implement $\tanh$ using `NumPy`, which is already loaded. The only other thing we'll need is the derivative of $\tanh$

$$\frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

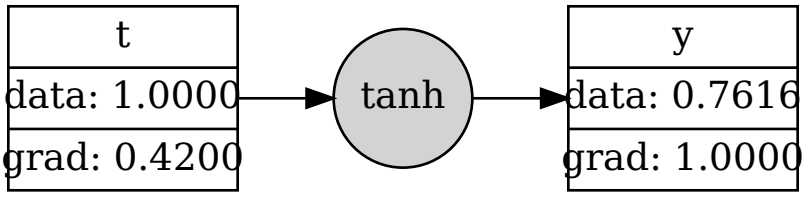
$\frac{\partial x}{\partial x}$

so we can implement its `_backward` function.

```
class Scalar:
    def __init__(self, data, _children=(), _op="", label=None):
        self.data = data
        self.grad = 0.0
        self._prev = set(_children)
        self._backward = lambda: None
        self._op = _op
        if label is not (None):
            self.label = label

    def __repr__(self):
        return f"Scalar({self.data}, grad = {self.grad})"

    def __add__(self, other):
        if not isinstance(other, Scalar):
            other = Scalar(other)
        output = Scalar(self.data + other.data, _children=(self, other), _op="+")

        def _backward():
            self.grad += output.grad
            other.grad += output.grad

        output._backward = _backward
        return output

    def __mul__(self, other):
        if not isinstance(other, Scalar):
            other = Scalar(other)
        output = Scalar(self.data * other.data, _children=(self, other), _op="*")

        def _backward():
            self.grad += other.data * output.grad
            other.grad += self.data * output.grad

        output._backward = _backward
        return output

    def __neg__(self):
        if not isinstance(self, Scalar):
            self = Scalar(self)
        output = Scalar(-self.data, _children=(self,), _op="neg")

        def _backward():
            self.grad -= output.grad

        output._backward = _backward
        return output

    def __sub__(self, other):
        if not isinstance(other, Scalar):
```

```
            other = Scalar(other)
        output = self + (-other)
        return output

    def tanh(self):
        output = Scalar(np.tanh(self.data), _children=(self,), _op="tanh")

        def _backward():
            self.grad += (1 - output.data * output.data) * output.grad

        output._backward = _backward
        return output

    def backward(self):
        """
        Recursively build the topological order of the computation graph and perform th
        """
        # Topological order all of the children in the graph
        topology = []
        visited = set()

        def build_topology(v):
            if v not in visited:
                visited.add(v)
                for child in v._prev:
                    build_topology(child)
                topology.append(v)

        build_topology(self)

        # Manually set the gradient of the final output
        self.grad = 1
        # Iterate one variable at a time
        for v in reversed(topology):
            v._backward()


t = Scalar(1.0, label="t")
y = t.tanh()
y.label = "y"
print("y =", y)
y.backward()
visualize_graph(y)
```

```
y = Scalar(0.7615941559557649, grad = 0.0)
```



## Exercise: implement hyperbolic tangent

Implement `tanh` using it's exponential definition.

## Minimization

We have now fully completed one forward-pass followed by a backward-pass. After the backward pass, we have the gradients for each of the nodes in the computation graph. As we discussed at the beginning of this tutorial, one of the main goals of autodiff is to unlock the ability to optimize the final output of an arbitrarily complicated functions. So, how does access to gradients help us do this? Say I wanted to ***decrease*** the value of $\mathcal{L}$. Clearly, I would want to move each input (leaf node) in a direction which would decrease the final output value. This can be achieved straightforwardly using the gradient information of the leaf nodes:

$$a \rightarrow a - \alpha \frac{\partial \mathcal{L}}{\partial a}$$

where $\alpha$ is an arbitrary parameter determining how large of a step to take when varying $\mathcal{L}$, typically referred to in machine learning parlance as the ***learning rate***.

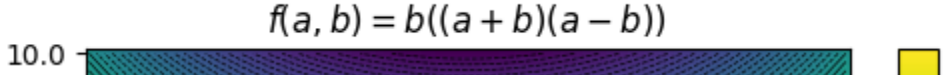
```
# Perform forward pass
a = Scalar(1.0)
b = Scalar(2.0)
c = -b
d = a + b
e = a + c
f = d * e
L = f * b
# Perform backward pass
L.backward()

print("Base L =", L)

agrad = a.grad
bgrad = b.grad
print("a.grad =", a.grad)
print("b.grad =", b.grad)

# Define learning rate
alpha = 0.01

# We want to minimize L
a = a - alpha * a.grad
b = Scalar(2.0)
c = -b
d = a + b
e = a + c
```

```
f = d * e
La = f * b
# La.backward()

print("a updated L =", La)

# We want to minimize L
a = Scalar(1.0)
b = b - alpha * bgrad
c = -b
d = a + b
e = a + c
f = d * e
Lb = f * b
# Lb.backward()

print("b updated L =", Lb)
```

```
Base L = Scalar(-6.0, grad = 1)
a.grad = 4.0
b.grad = -11.0
a updated L = Scalar(-6.1568000000000005, grad = 0.0)
b updated L = Scalar(-7.283930999999998, grad = 0.0)
```

What if we wanted to find the global minimum through an iterated optimization?

```
# Let's visualize our function
fig, ax = plt.subplots(1, 1, figsize=(6, 5))
# Create a 2d contour of f(a,b) = b * ((a + b) * (a - b))
a_vals = np.linspace(-10, 10, 100)
b_vals = np.linspace(-10, 10, 100)
A, B = np.meshgrid(a_vals, b_vals)
Z = B * ((A + B) * (A - B))
# Plot a pcolomesh
pcolormesh = ax.pcolormesh(A, B, Z, shading="auto", cmap="viridis")
# Plot the contour
contour = ax.contour(A, B, Z, levels=50, colors="black", linewidths=0.5)
# Plot the initial point
ax.plot(a.data, b.data, "ro", label="Initial point (a, b)")
# Add a colorbar
# cbar = fig.colorbar(contour, ax=ax)
cbar = fig.colorbar(pcolormesh, ax=ax)
cbar.set_label(r"$f(a, b)$")
ax.set_xlabel(r"$a$")
ax.set_ylabel(r"$b$")
ax.set_title(r"$f(a, b) = b ((a + b)(a - b))$")
fig.tight_layout()
```




```
# We can now, very easily create a gradient vector field over any domain of interest
```

```
# We can now, very easily create a gradient vector field over any domain of interest
def gradient_vector_field(x_range, y_range, step=0.5):
    x_vals = np.arange(x_range[0], x_range[1], step)
    y_vals = np.arange(y_range[0], y_range[1], step)
    X, Y = np.meshgrid(x_vals, y_vals)

    # Compute the gradient
    dX = np.zeros_like(X)
    dY = np.zeros_like(Y)

    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            a = Scalar(X[i, j])
            b = Scalar(Y[i, j])
            c = -b
            d = a + b
            e = a + c
            f = d * e
            L_val = f * b
            L_val.backward()
            dX[i, j] = a.grad
            dY[i, j] = b.grad

    return X, Y, dX, dY


# Define the range for the gradient vector field
x_range = (-10, 10)
y_range = (-10, 10)

# Compute the gradient vector field
X, Y, dX, dY = gradient_vector_field(x_range, y_range)

# Create figure
fig, ax = plt.subplots(1, 1, figsize=(6, 5))
# Plot the function as a pcolormesh
pcolormesh = ax.pcolormesh(A, B, Z, shading="auto", cmap="viridis")
# Plot the gradient vector field
ax.quiver(
    X, Y, dX, dY, color="black", alpha=1.0, scale=3000
)  # , headlength = 3, width = 0.01)
# Plot the contour
contour = ax.contour(A, B, Z, levels=50, colors="black", linewidths=0.5)

# Add a colorbar
cbar = fig.colorbar(pcolormesh, ax=ax)
cbar.set_label(r"$f(a, b)$")

# Set axis labels
ax.set_xlabel(r"$a$")
ax.set_ylabel(r"$b$")
fig.tight_layout()
```


```
# Now we can define an automated optimization that looks to maximize or minimize the fu
def minimize(a_init, b_init, alpha=0.01, steps=10):
    a = Scalar(a_init, label="a")
    b = Scalar(b_init, label="b")

    a_b_trajectory = []
    a_b_trajectory.append((a.data, b.data))

    for _ in range(steps):
        c = -b
        d = a + b
        e = a + c
        f = d * e
        L = f * b

        # Perform backward pass
        L.backward()

        # Update parameters in direction that minimizes L
        a.data -= alpha * a.grad
        b.data -= alpha * b.grad

        a_b_trajectory.append((a.data, b.data))

        # Reset gradients for the next iteration
        a.grad = 0.0
        b.grad = 0.0

    return a_b_trajectory


# Iterate through n_trajectories trajectories with different initial conditions assigne
n_trajectories = 50
trajectories = []
for i in range(n_trajectories):
    a_init = np.random.uniform(-10, 10)
    b_init = np.random.uniform(-10, 10)
    trajectory = minimize(a_init, b_init, alpha=1e-3, steps=25)
    trajectories.append(trajectory)


# Set the minimum and maximum values for a and b for plotting
a_min = -20
a_max = 20
b_min = -20
b_max = 20

# Create a grid for the pcolormesh
a_vals = np.linspace(a_min, a_max, 100)
b_vals = np.linspace(b_min, b_max, 100)
A, B = np.meshgrid(a_vals, b_vals)
```

```
A, B = np.meshgrid(a_vals, b_vals)
# Compute the function values for the pcolormesh
Z = B * ((A + B) * (A - B))

# Create a figure for the trajectories
fig, ax = plt.subplots(1, 1, figsize=(6, 5))
# Plot the function as a pcolormesh
pcolormesh = ax.pcolormesh(A, B, Z, shading="auto", cmap="viridis")

# Produce a unique color for each trajectory
colors = plt.cm.hsv(np.linspace(0, 1, len(trajectories)))

# For each trajectory, plot connected points
for trajectory in trajectories:
    a_vals, b_vals = zip(*trajectory)
    # Make the initial point black and the rest of the trajectory colored
    if len(trajectory) > 0:
        # Plot the first point in black
        ax.plot(a_vals[0], b_vals[0], "ko", markersize=5, label="Initial point")
        # Plot the rest of the trajectory with the assigned color
        ax.plot(
            a_vals,
            b_vals,
            marker="o",
            markersize=3,
            alpha=0.5,
            color=colors[trajectories.index(trajectory)],
        )

# Set the limits
ax.set_xlim(a_min, a_max)
ax.set_ylim(b_min, b_max)

# Add the gradient vector field
X, Y, dX, dY = gradient_vector_field((a_min, a_max), (b_min, b_max), step=1.0)
# Plot the gradient vector field
ax.quiver(X, Y, dX, dY, color="black", alpha=0.75, scale=5000)

# Add a colorbar
cbar = fig.colorbar(pcolormesh, ax=ax)
cbar.set_label(r"$f(a, b)$")

# Set labels
ax.set_xlabel(r"$a$")
ax.set_ylabel(r"$b$")
fig.tight_layout()
```


## Tools

For real-world research and workflows, there are a number of existing differential programming libraries that offer sophisticated autodiff capabilities. Each library differs slightly in underlying design decisions which may or may not impact usefulness for your application. All options will offer well-documented, highly optimized algorithms that can leverage optional GPU acceleration within high-level abstractions. This lowers the barrier to entry and allows new users to write powerful programs very quickly. There is a dedicated notebook but I differentiate the same function we worked so hard to get right above using `Scalar` in two popular differential programming libraries, PyTorch and JAX.

## PyTorch

```
import torch


# Perform forward pass
a = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(2.0, requires_grad=True)
L = b * ((a + b) * (a - b))

# Note that PyTorch will parse L into its own internal computation
# graph representation, so we do not need to manually construct it
# as we've been doing. In PyTorch, only the leaf nodes need to be
# defined with requires_grad=True.

# Perform backward pass
L.backward()

# Print out gradients of leaf nodes
print("a.grad =", a.grad)
print("b.grad =", b.grad)
```

```
    a.grad = tensor(4.)
    b.grad = tensor(-11.)
```

## JAX

```
import jax


f = lambda a, b: b * ((a + b) * (a - b))

A = 1.0
B = 2.0

df_da = jax.grad(f, argnums=0)
print("a.grad =", df_da(A, B))

df db = jax.grad(f, argnums=1)
```

```
print("b.grad =", df_db(A, B))

# Because JAX is a functional programming library, second derivatives are automatically
d2f_da2 = jax.jacobian(df_da, argnums=0)
d2f_db2 = jax.jacobian(df_db, argnums=1)
print("a.second_grad =", d2f_da2(A, B))
print("b.second_grad =", d2f_db2(A, B))
```

```
a.grad = 4.0
b.grad = -11.0
a.second_grad = 4.0
b.second_grad = -12.0
```

## References


**Blogs**:

- https://jingnanshi.com/blog/autodiff.html
- https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation
- https://www.stochasticlifestyle.com/engineering-trade-offs-in-automatic-differentiation-from-tensorflow-and-pytorch-to-jax-and-julia/

**Repos**:

- https://github.com/karpathy/micrograd
- https://github.com/rasmusbergpalm/nanograd
- https://github.com/breandan/picograd
- https://github.com/HIPS/autograd/tree/master

**Textbook**:

Griewank, Andreas, and Andrea Walther. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2008.