Lecture 8

Automatic Differentiation and Root Finding

CS328 - Numerical Methods for Visual Computing and Machine Learning

Prof. Wenzel Jakob

EPFL RGL Realistic Graphics Lab

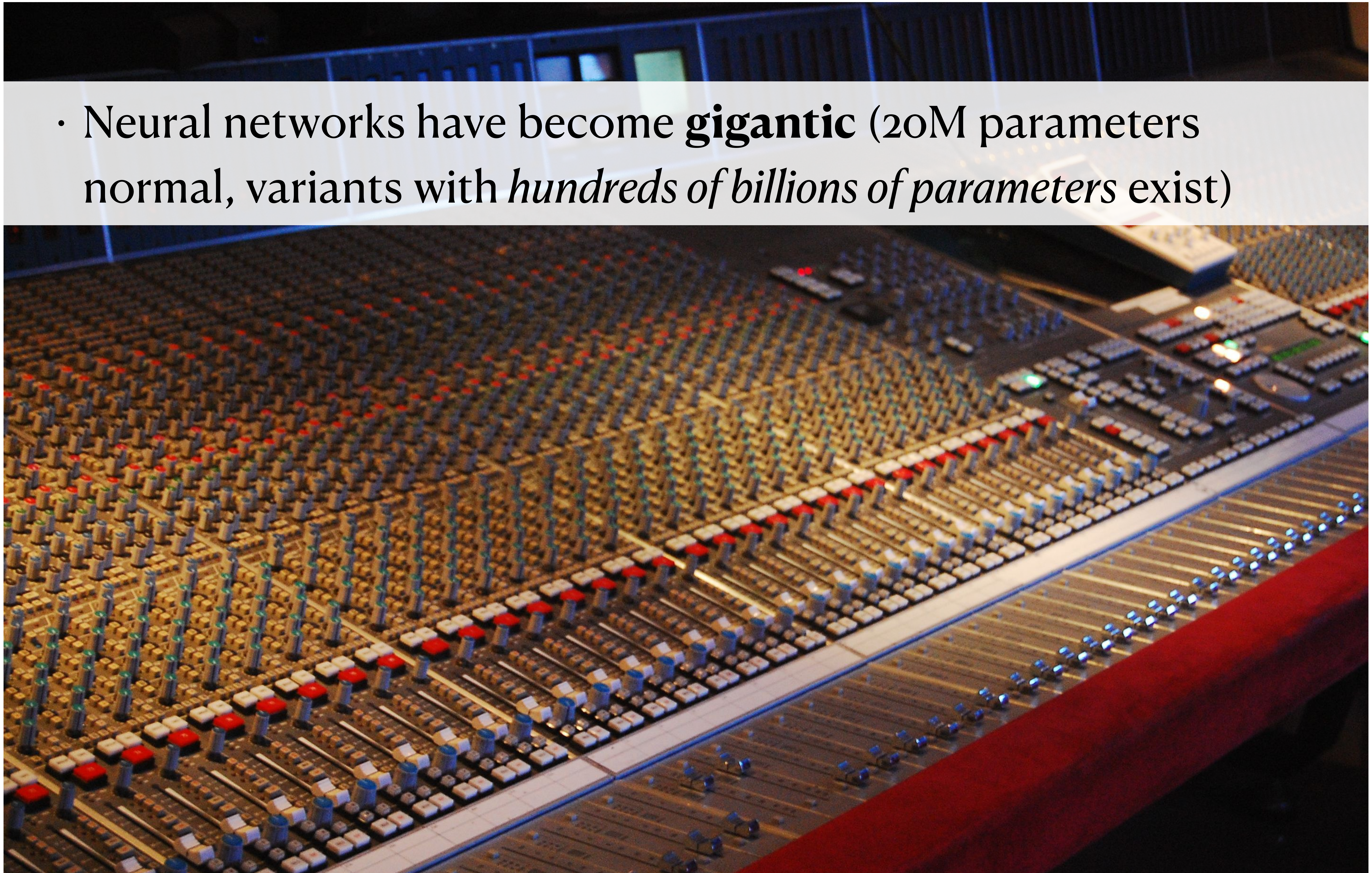
Motivation: training of neural networks

```
In [27]: for t in range(500):
             # Forward pass: compute predicted y by passing x to the model.
             y_pred = model(x)

             # Compute and print loss.
             loss = loss_fn(y_pred, y)

             # Zero out all gradients
             optimizer.zero_grad()

             # Compute gradient of the loss with respect to model parameters
             loss.backward()

             # Let optimizer update parameters to improve loss fct.
             optimizer.step()
```

Motivation: training of neural networks

[Wikimedia commons]

Motivation: training of neural networks

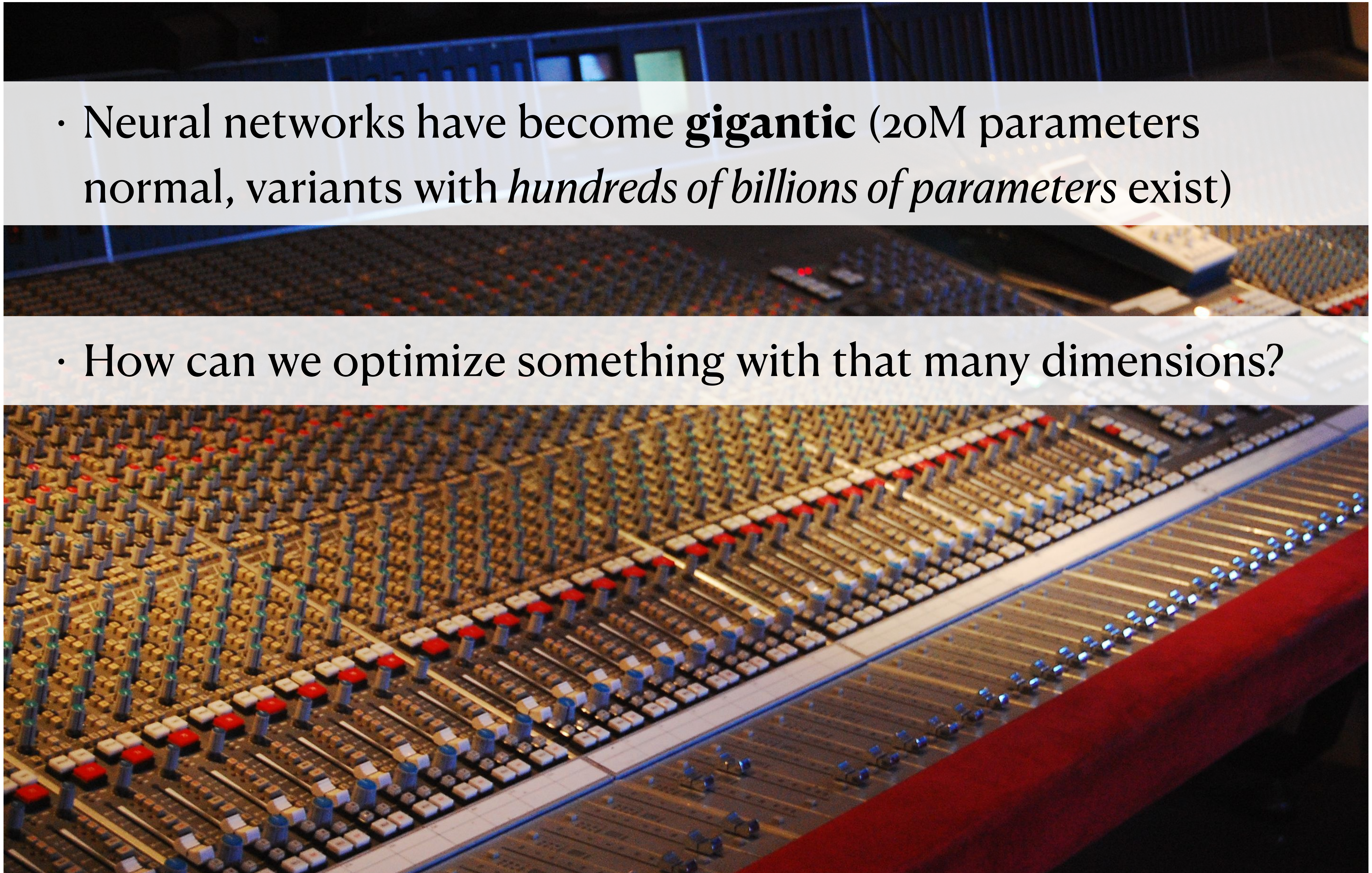


- Neural networks have become **gigantic** (20M parameters normal, variants with *hundreds of billions of parameters* exist)

[Wikimedia commons]

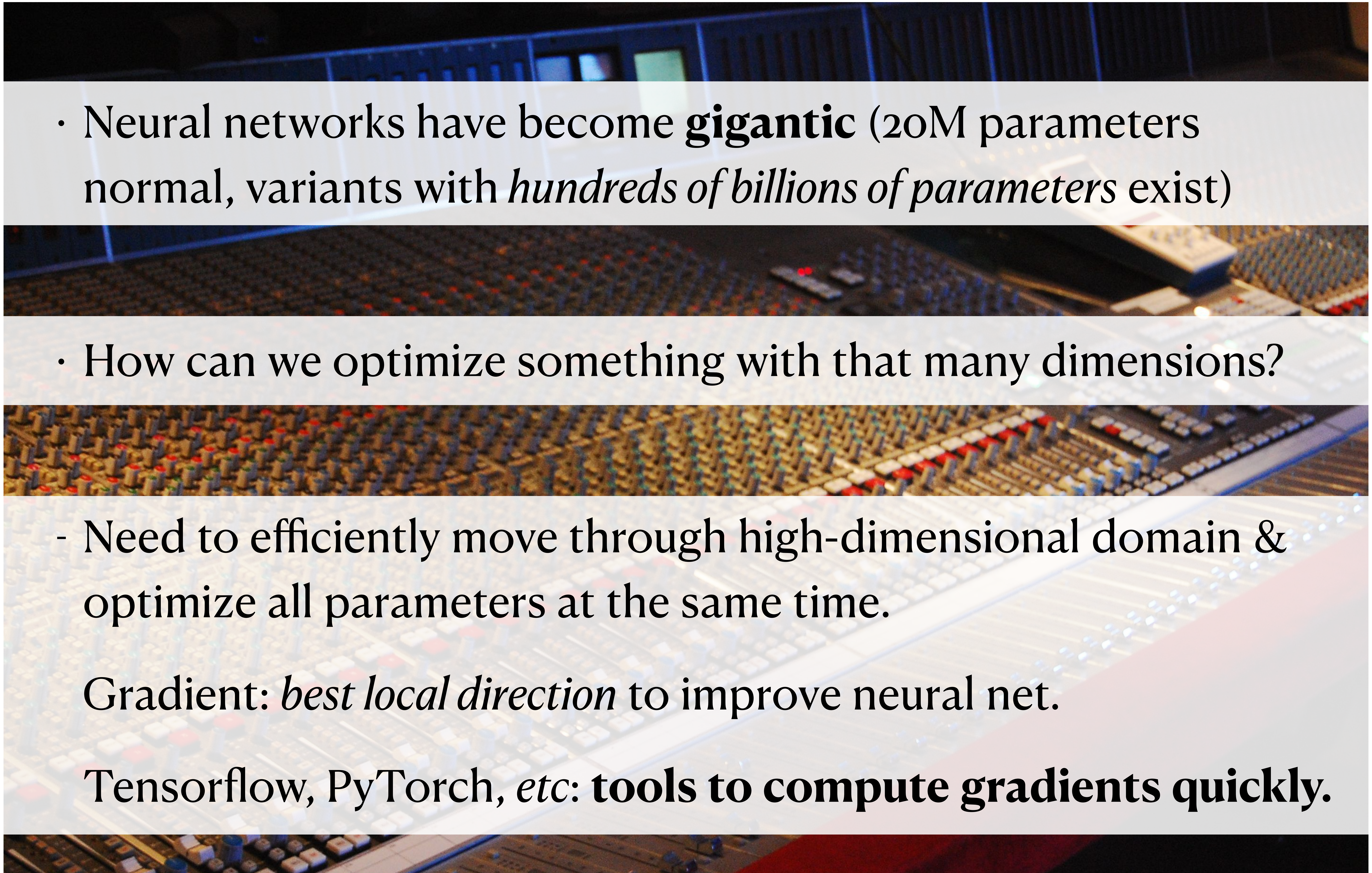
Motivation: training of neural networks

- Neural networks have become **gigantic** (20M parameters normal, variants with *hundreds of billions of parameters* exist)
- How can we optimize something with that many dimensions?



[Wikimedia commons]

Motivation: training of neural networks



[Wikimedia commons]

- Neural networks have become **gigantic** (20M parameters normal, variants with *hundreds of billions of parameters* exist)

- How can we optimize something with that many dimensions?

- Need to efficiently move through high-dimensional domain & optimize all parameters at the same time.

  Gradient: *best local direction* to improve neural net.

  Tensorflow, PyTorch, *etc*: **tools to compute gradients quickly.**

How to evaluate gradients?

Let's review some standaard techniques for evaluating derivatives.

- **Finite Differences**
  - Evaluate function at nearby points to estimate derivative

How to evaluate gradients?

Let's review some standaard techniques for evaluating derivatives.

- **Finite Differences**
  - Evaluate function at nearby points to estimate derivative

- **Symbolic (by hand, or with help of software)**
  - Apply all the rules you've learned in MATH-101

How to evaluate gradients?

Let's review some standaard techniques for evaluating derivatives.

- **Finite Differences**
  - Evaluate function at nearby points to estimate derivative
- **Symbolic (by hand, or with help of software)**
  - Apply all the rules you've learned in MATH-101
- **Automatic differentiation ("AD", "autodiff")**
  - Like "symbolic", but with a few extra tricks!

Finite Differences

Forward difference

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

Centered difference

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

Finite Differences

Forward difference

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

Centered difference

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

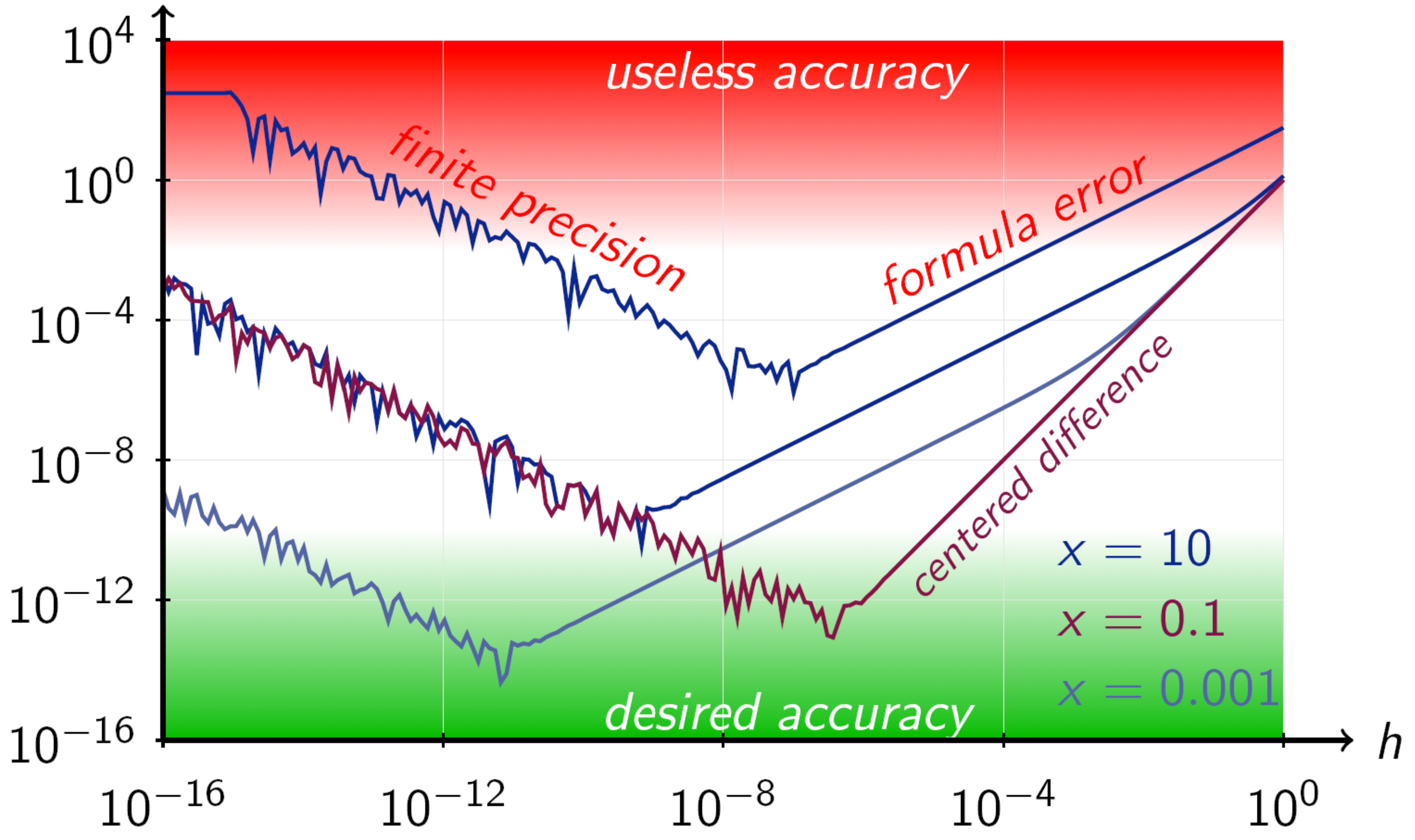
Finite Differences

Example: $f(x) = x^3$

(*plot shows* $f'_{\text{approx}}(x) - 3x^2$)

Forward difference

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



Centered difference

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

Finite Differences

Forward difference

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

Centered difference

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

Main problems:

- 1M dimensions? Must evaluate f at least 1M+1 times
- Complex tradeoff between approximation / cancellation error

Symbolic versus Automatic Derivatives

Original calculation:

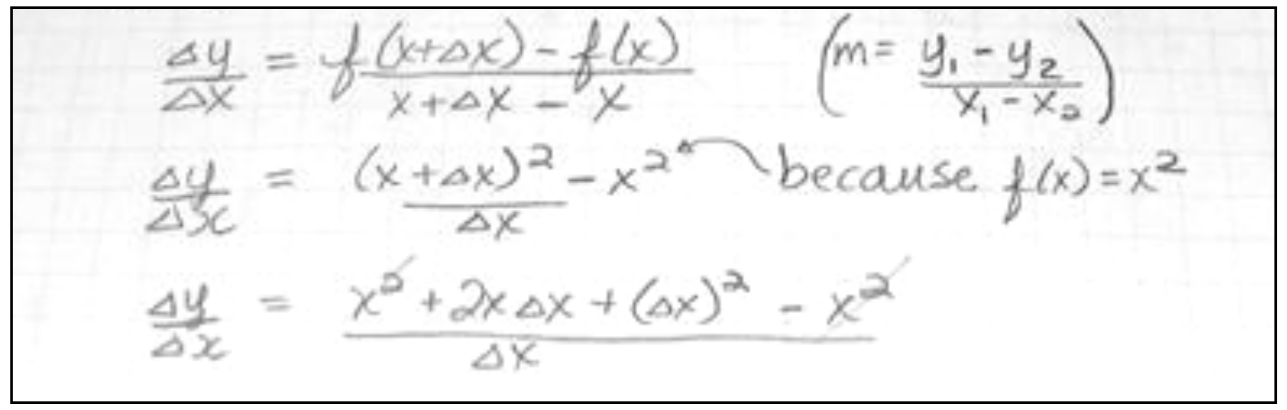
```
def f(x):
    for i in range(8):
        x = exp(x)
    return x
```

Symbolic versus Automatic Derivatives

Original calculation:

```
def f(x):
    for i in range(8):
         x = exp(x)
     return x
```

Symbolic derivative:



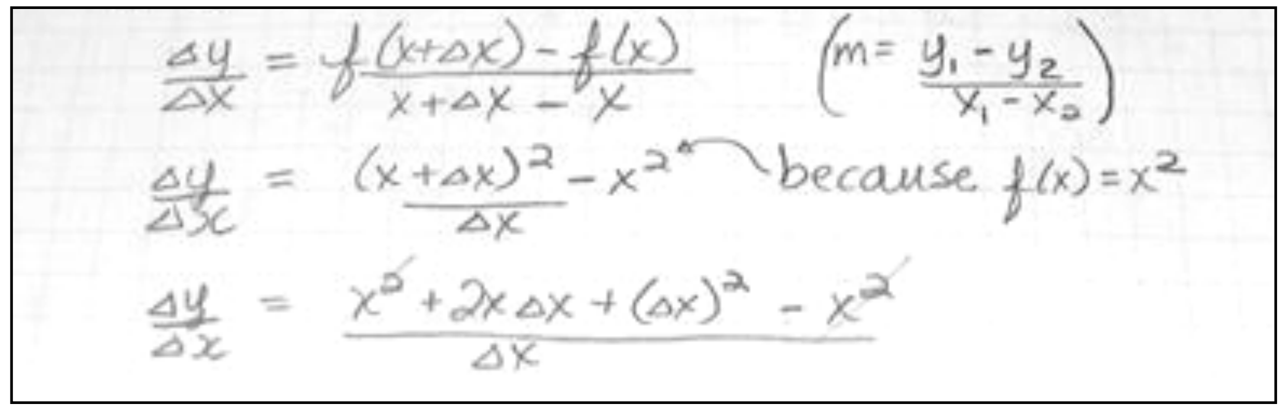
By hand

Symbolic versus Automatic Derivatives

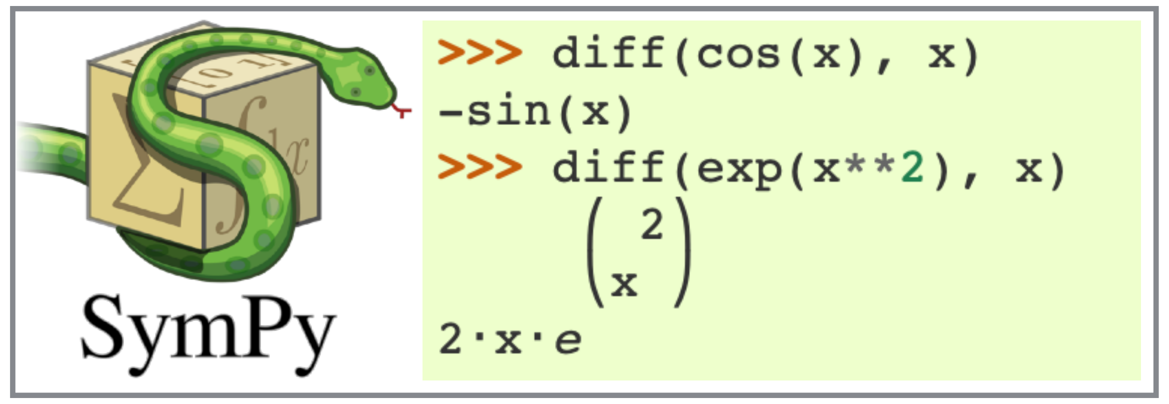
Original calculation:

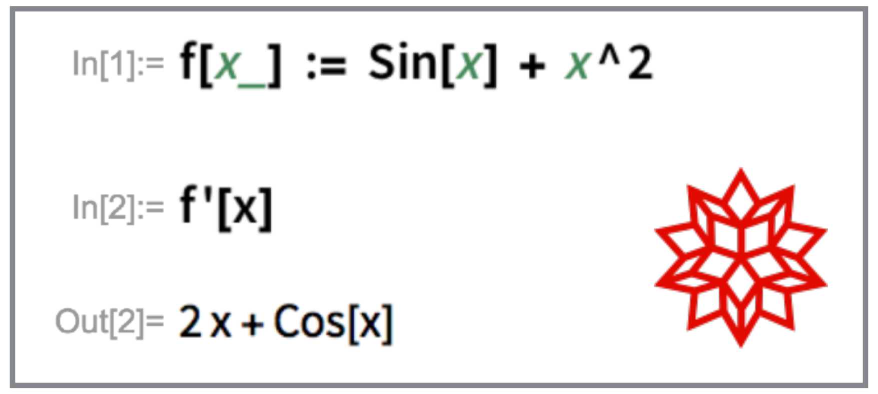
```python
def f(x):
    for i in range(8):
         x = exp(x)
     return x
```

Symbolic derivative:



By hand





SymPy Mathematica

Computer Algebra System (CAS)

Symbolic versus Automatic Derivatives

Original calculation:

```
def f(x):
    for i in range(8):
        x = exp(x)
    return x
```

Symbolic derivative:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = e^{x+e^{e^{e^{e^{e^{e^{e^{x}}}}}}}+e^{e^{e^{e^{e^{e^{x}}}}}}+e^{e^{e^{e^{e^{x}}}}}+e^{e^{e^{e^{x}}}}+e^{e^{e^{x}}}+e^{e^{x}}+e^{x}}$$

Symbolic versus Automatic Derivatives

Original calculation:

```python
def f(x):
    for i in range(8):
        x = exp(x)
    return x
```

8 exponentials

Symbolic derivative:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = e^{x+e^{e^{e^{e^{e^{e^{e^{x}}}}}}}+e^{e^{e^{e^{e^{e^{x}}}}}}+e^{e^{e^{e^{e^{x}}}}}+e^{e^{e^{e^{x}}}}+e^{e^{e^{x}}}+e^{e^{x}}+e^{x}}$$

37 exponentials (!)

Symbolic versus Automatic Derivatives

Original calculation:

```
def f(x):
    for i in range(8):
        x = exp(x)
    return x
```

8 exponentials

Automatic differentiation (forward mode):

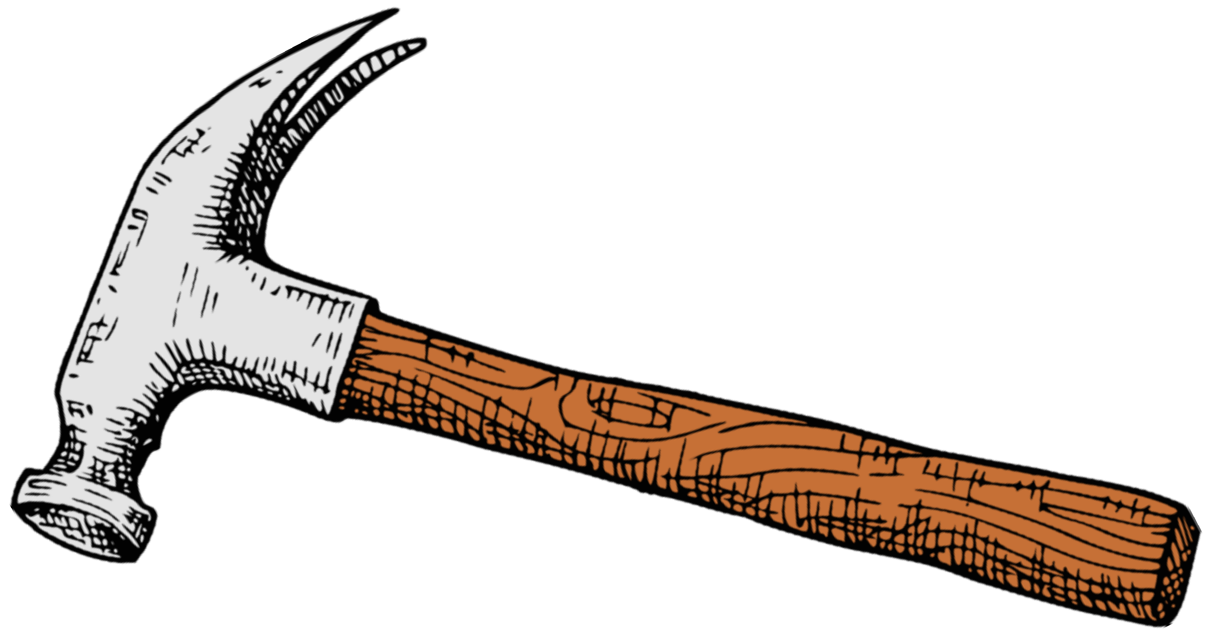
```
def df(x):
    d = 1
    for i in range(8):
        x = exp(x)
        d = d * x
    return d
```

8 exponentials

Automatic differentiation

$$f(\mathbf{x})$$

Automatic differentiation



$$f(\mathbf{x}) \xrightarrow{\text{AD}} \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x})$$

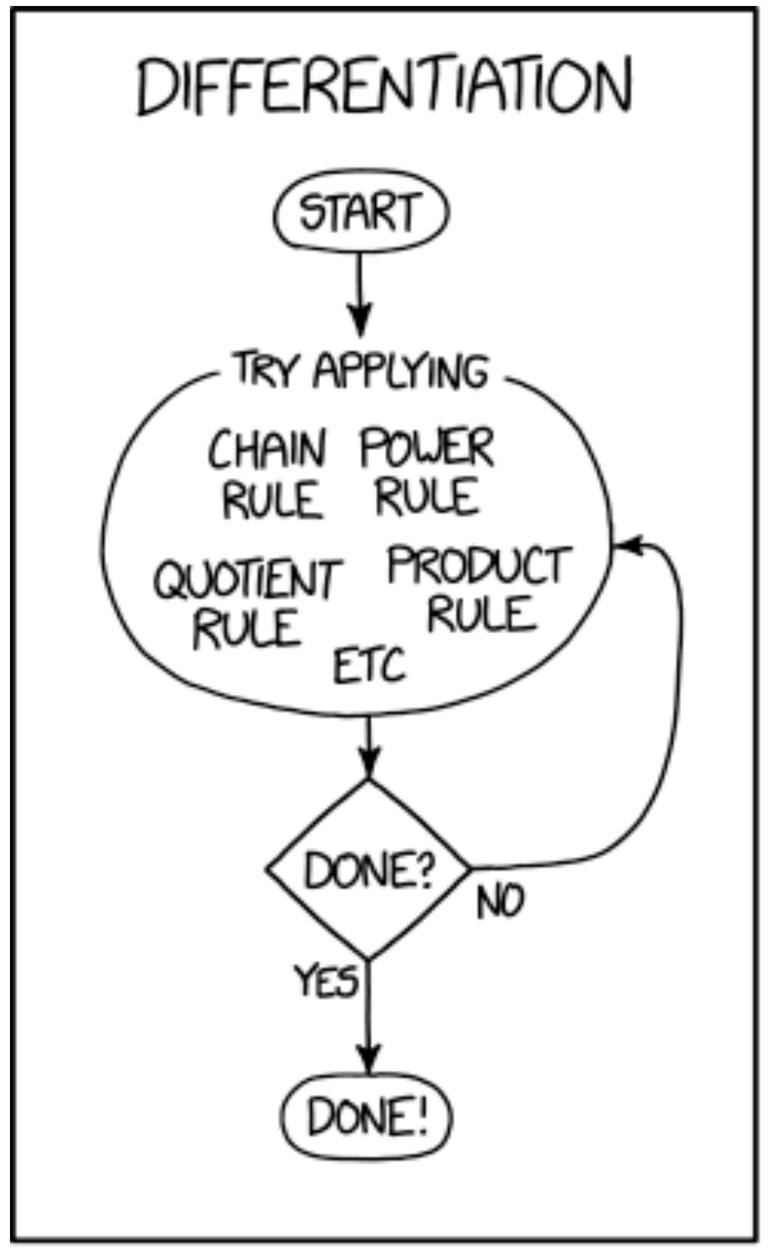
TensorFlow

PyTorch

AD Motivation

- Differention is **simple**, really! (at least *locally*)
  - Any program boils down to a sequence of tiny steps (`+`, `-`, `load`, `store`, etc.).
  - If we know how to handle each step, then we can differentiate the entire program!

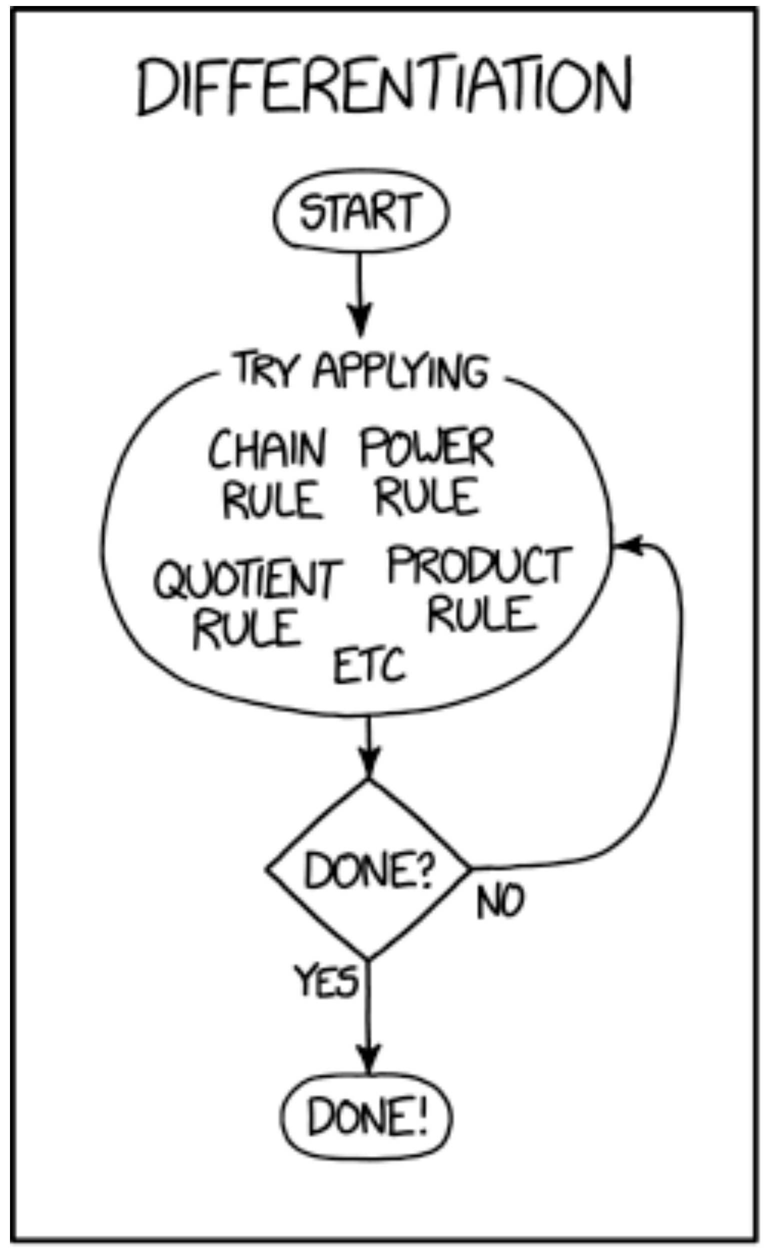
[xkcd.com]



AD Motivation

- Differention is **simple**, really! (at least *locally*)
  - Any program boils down to a sequence of tiny steps (`+`, `-`, `load`, `store`, etc.).
  - If we know how to handle each step, then we can differentiate the entire program!

[xkcd.com]

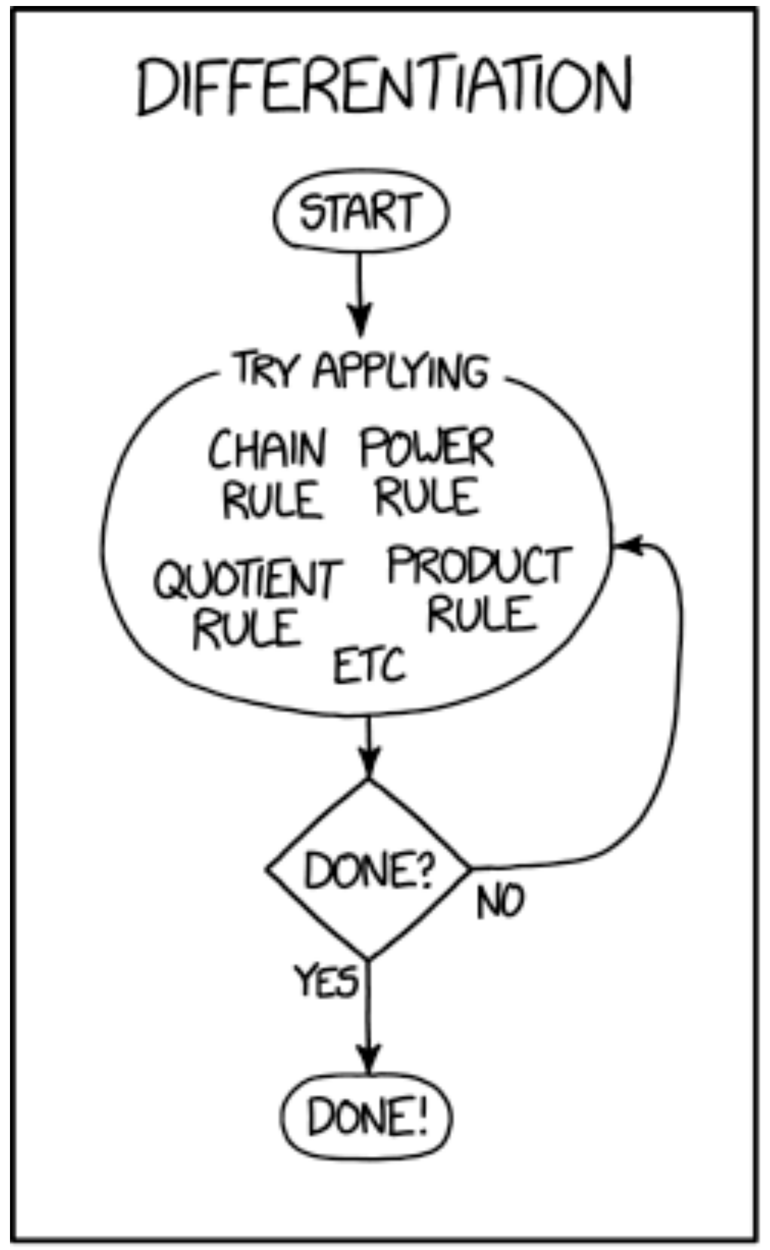


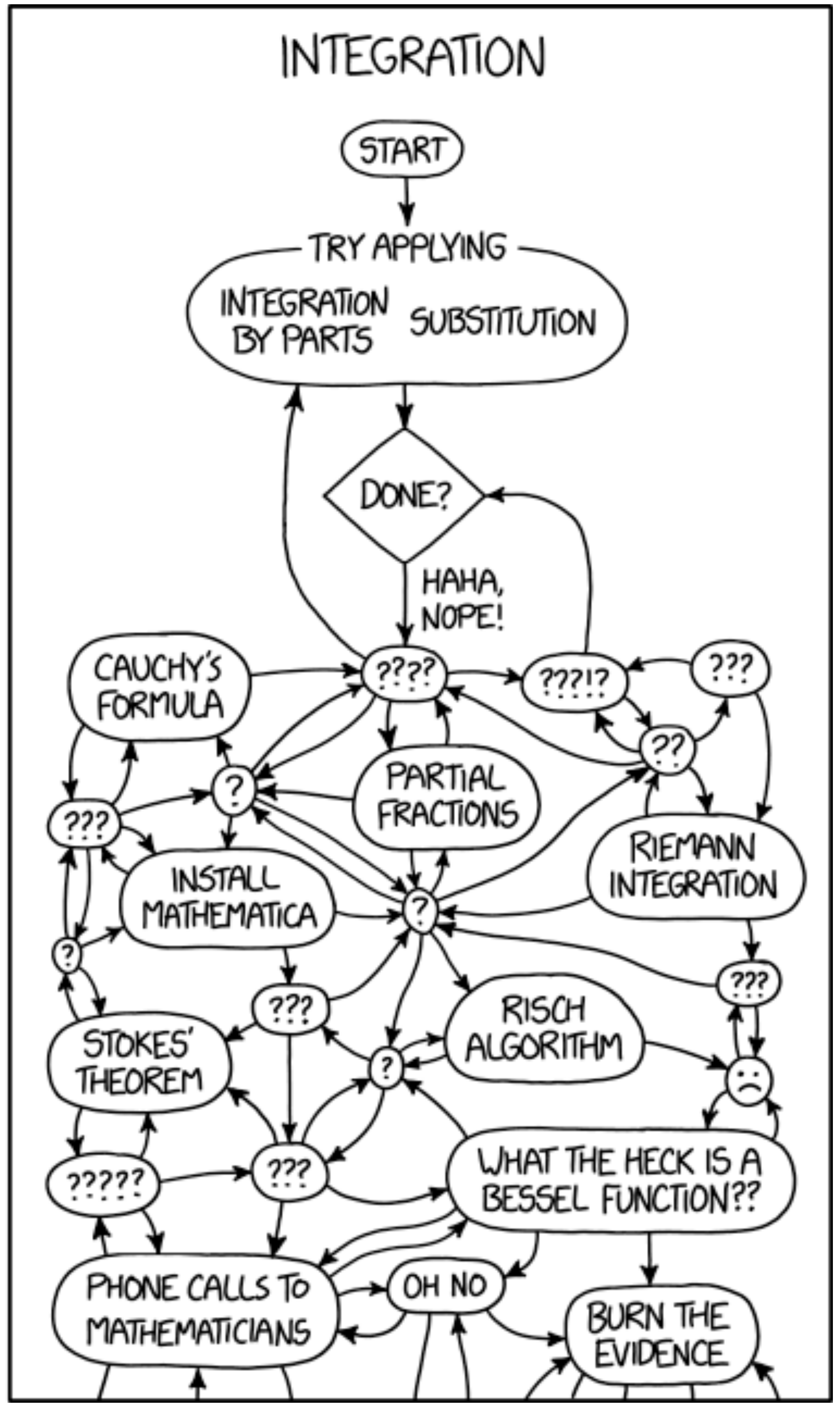
- AD has two key ingredients:
  - Chain rule
  - Common subexpressions

AD Motivation

- Differention is **simple**, really! (at least *locally*)
  - Any program boils down to a sequence of tiny steps (`+`, `-`, `load`, `store`, etc.).
  - If we know how to handle each step, then we can differentiate the entire program!

[xkcd.com]





- AD has two key ingredients:
  - Chain rule
  - Common subexpressions

AD Ingredient 1: The Chain Rule

Original calculation:

```
b = f(a)
c = g(b)
```

AD Ingredient 1: The Chain Rule

Original calculation:

```
b = f(a)
c = g(b)
```

Chain rule:

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}$$

AD Ingredient 1: The Chain Rule

Original calculation:

```
b = f(a)
c = g(b)
```

Chain rule:

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}$$

In graph form:

AD Ingredient 1: The Chain Rule

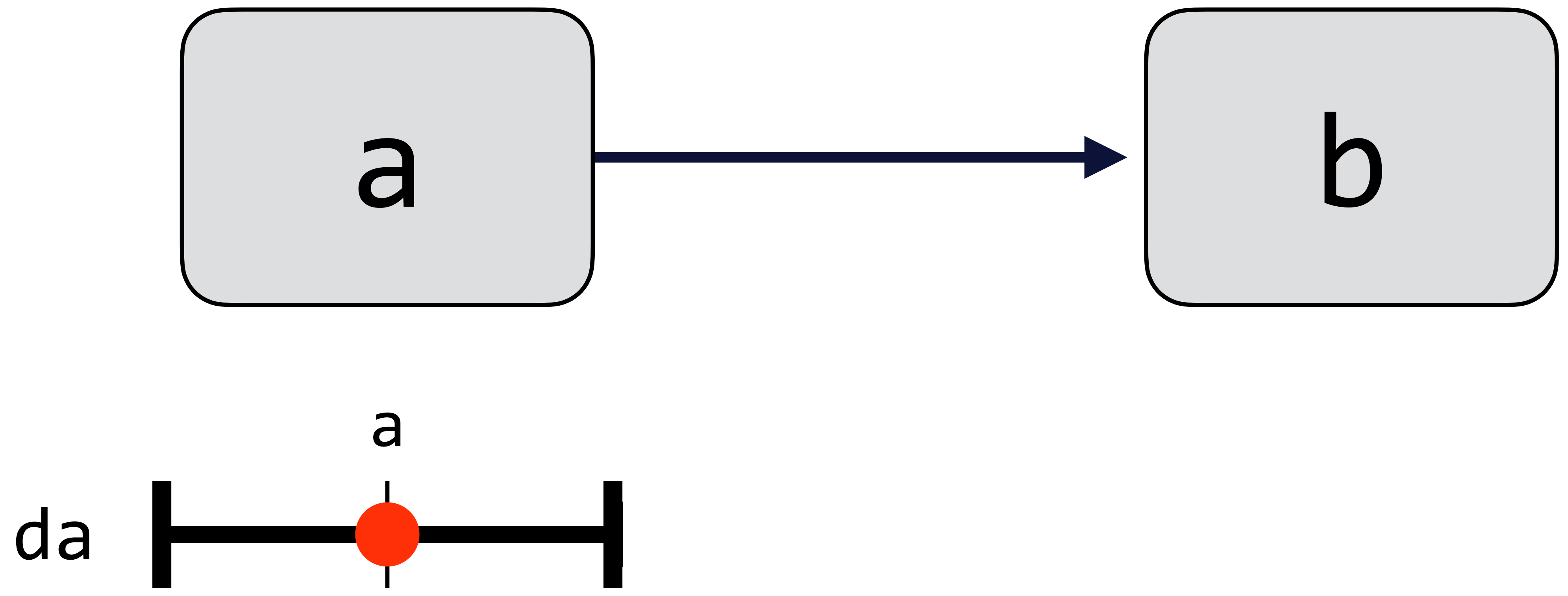
Original calculation:

```
b = f(a)
c = g(b)
```

Chain rule:

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}$$

In graph form:



AD Ingredient 1: The Chain Rule

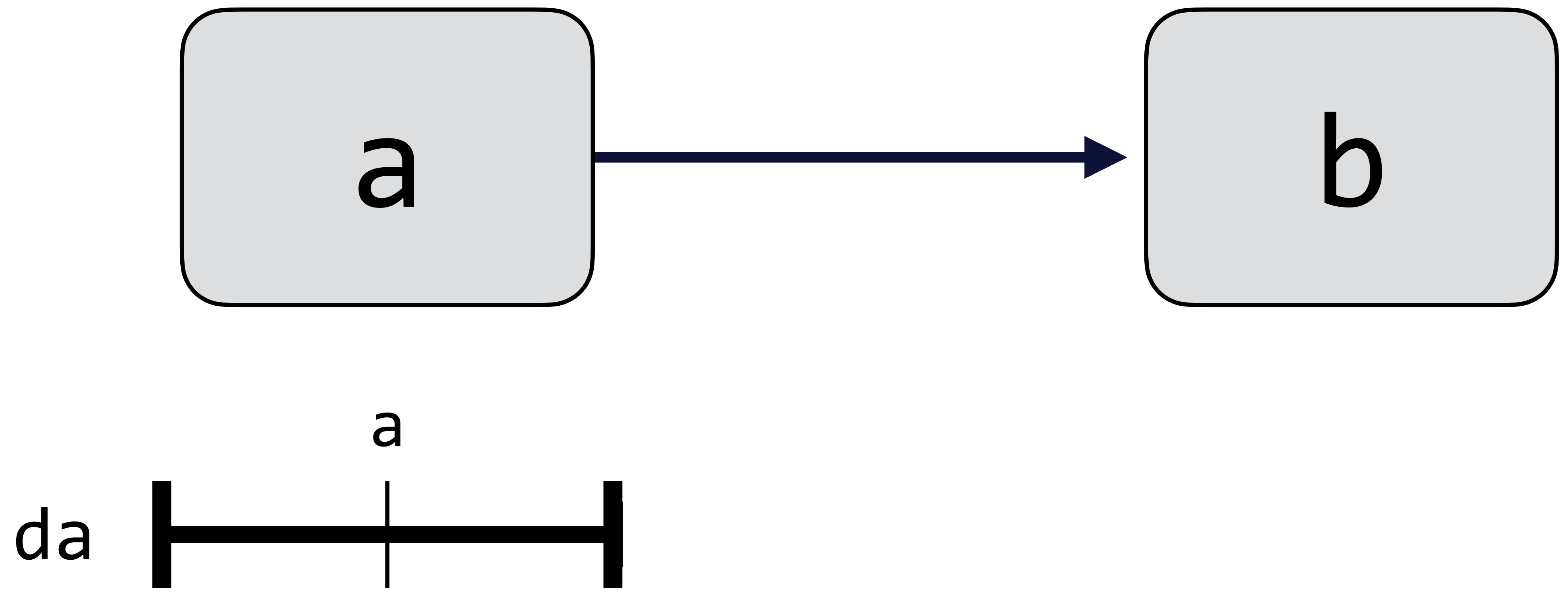
Original calculation:

```
b = f(a)
c = g(b)
```

Chain rule:

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}$$

In graph form:



AD Ingredient 1: The Chain Rule

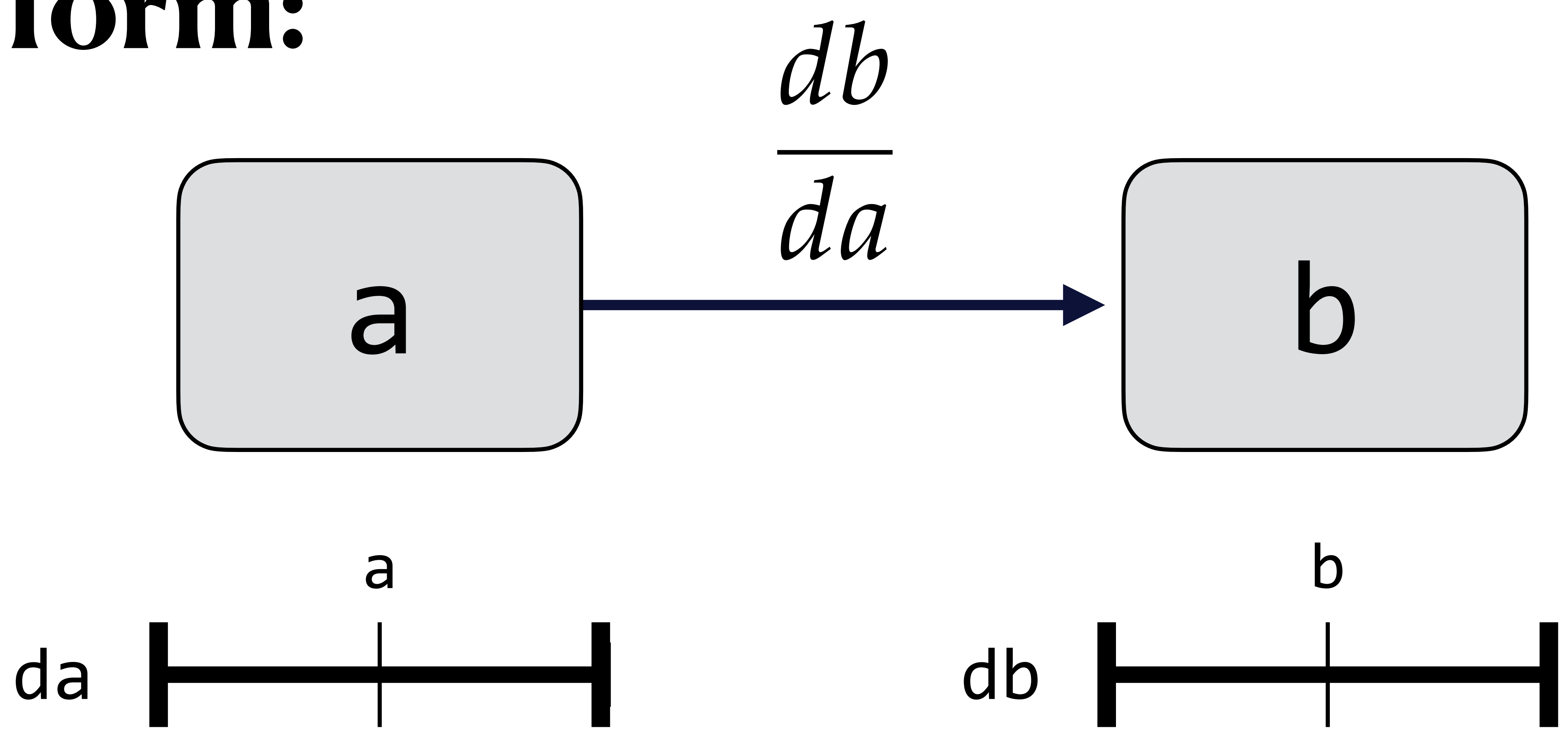
Original calculation:

```
b = f(a)
c = g(b)
```

Chain rule:

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}$$

In graph form:



AD Ingredient 1: The Chain Rule

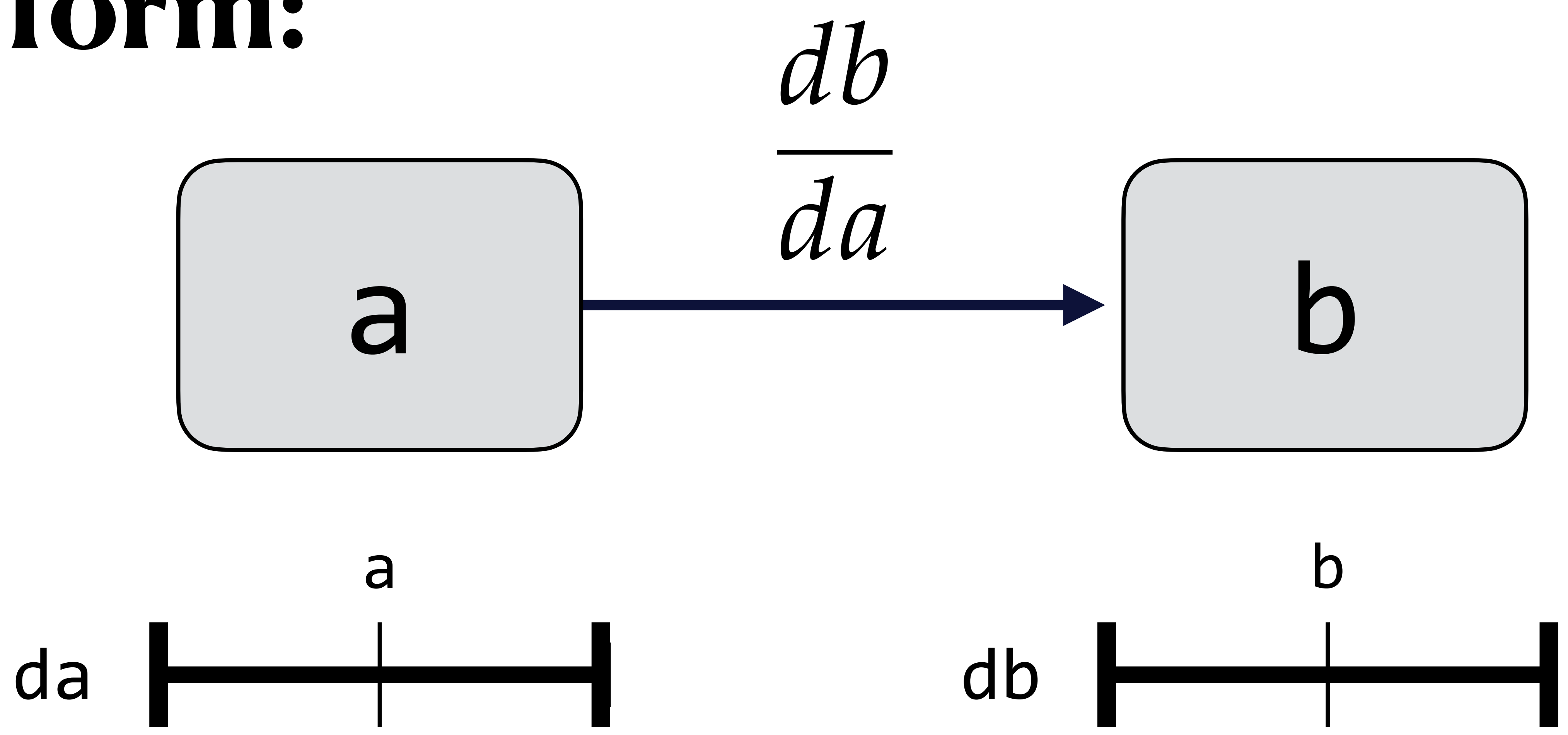
Original calculation:

```
b = f(a)
c = g(b)
```

Chain rule:

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}$$

In graph form:



AD Ingredient 1: The Chain Rule

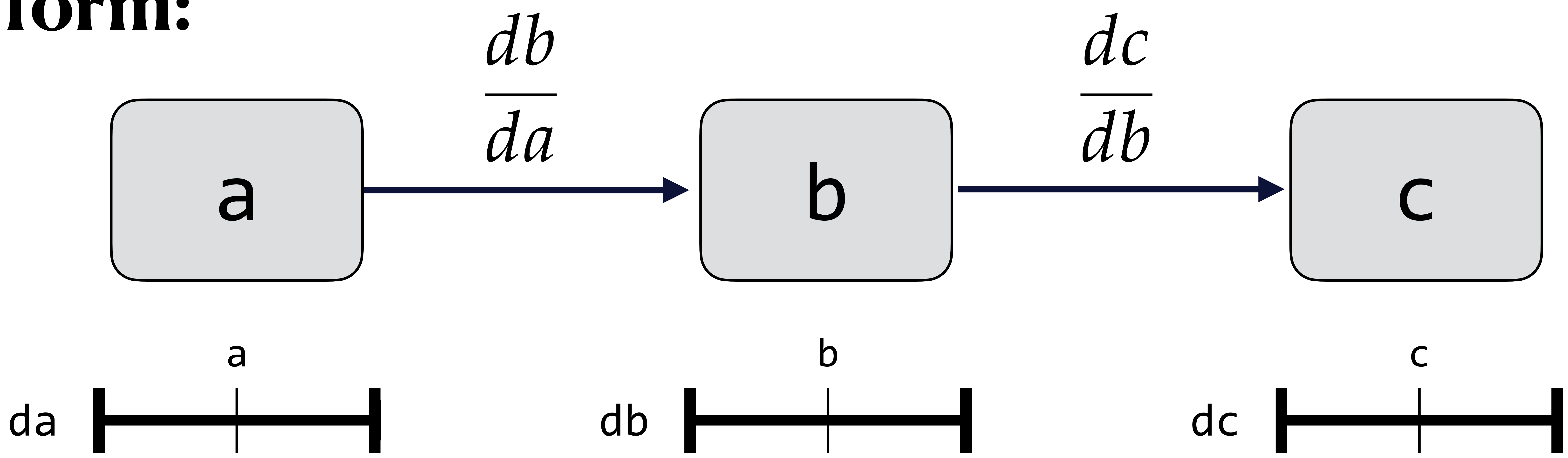
Original calculation:

```
b = f(a)
c = g(b)
```

Chain rule:

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}$$

In graph form:



AD Ingredient 1: The Chain Rule

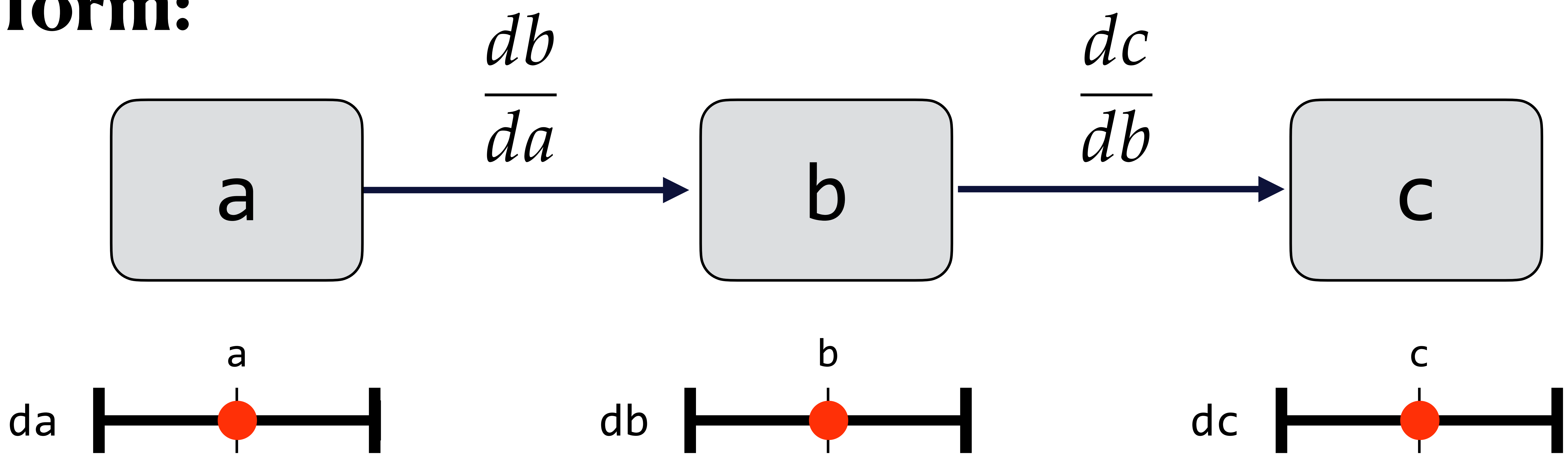
Original calculation:

```
b = f(a)
c = g(b)
```

Chain rule:

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}$$

In graph form:



AD Ingredient 2: Reuse of common subexpressions

Original calculation:

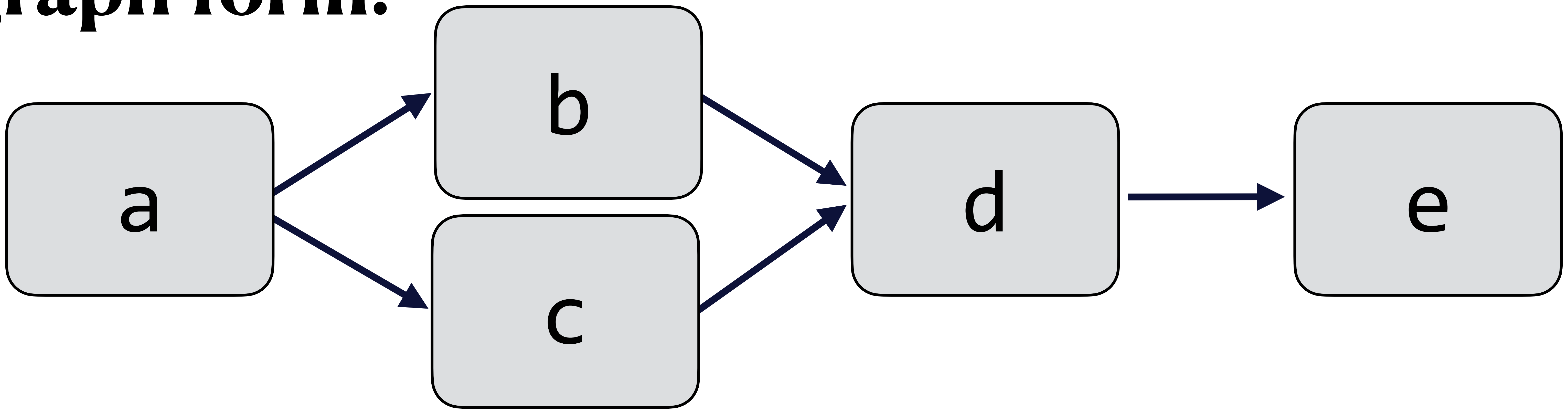
```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

AD Ingredient 2: Reuse of common subexpressions

Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

In graph form:



AD Ingredient 2: Reuse of common subexpressions

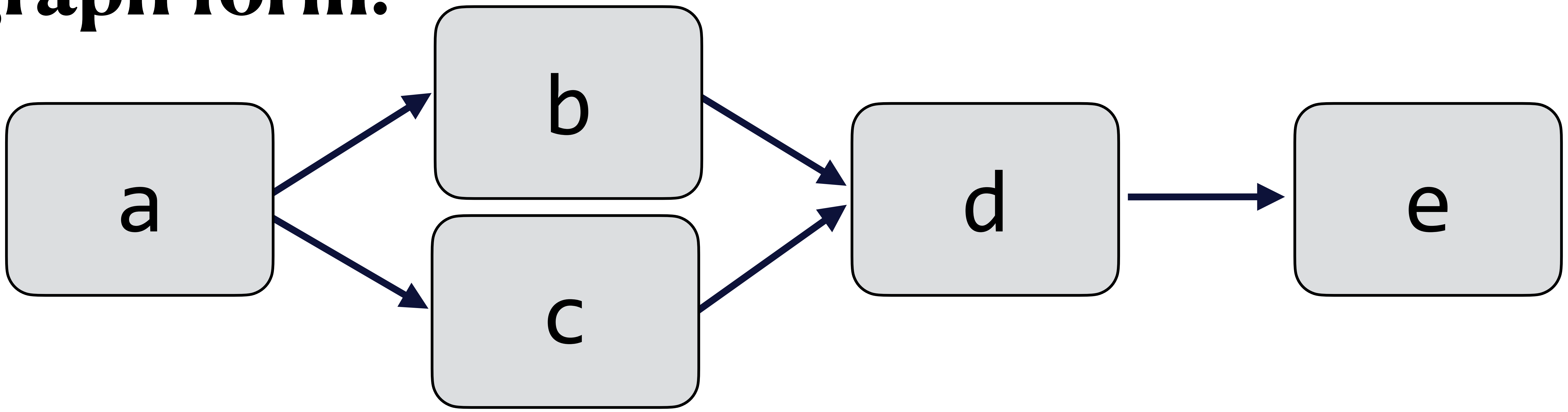
Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

In graph form:



AD Ingredient 2: Reuse of common subexpressions

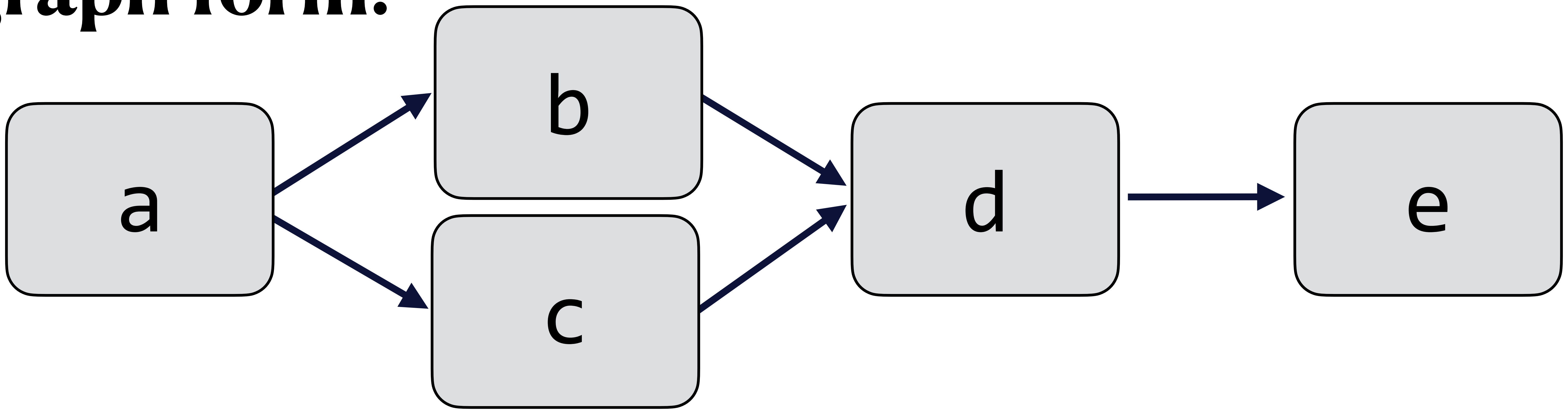
Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

In graph form:



AD Ingredient 2: Reuse of common subexpressions

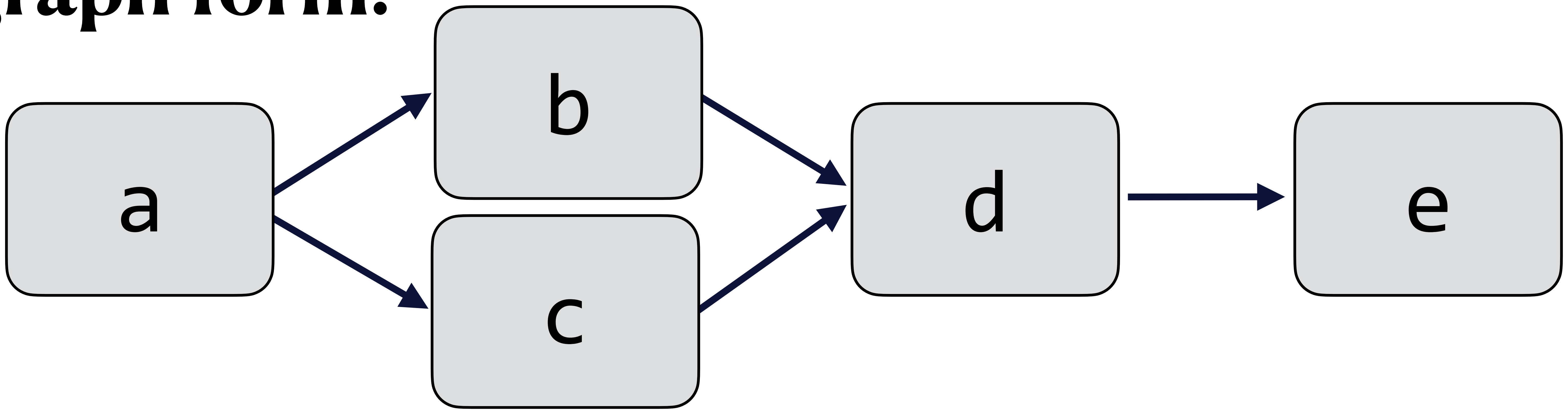
Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

In graph form:



AD Ingredient 2: Reuse of common subexpressions

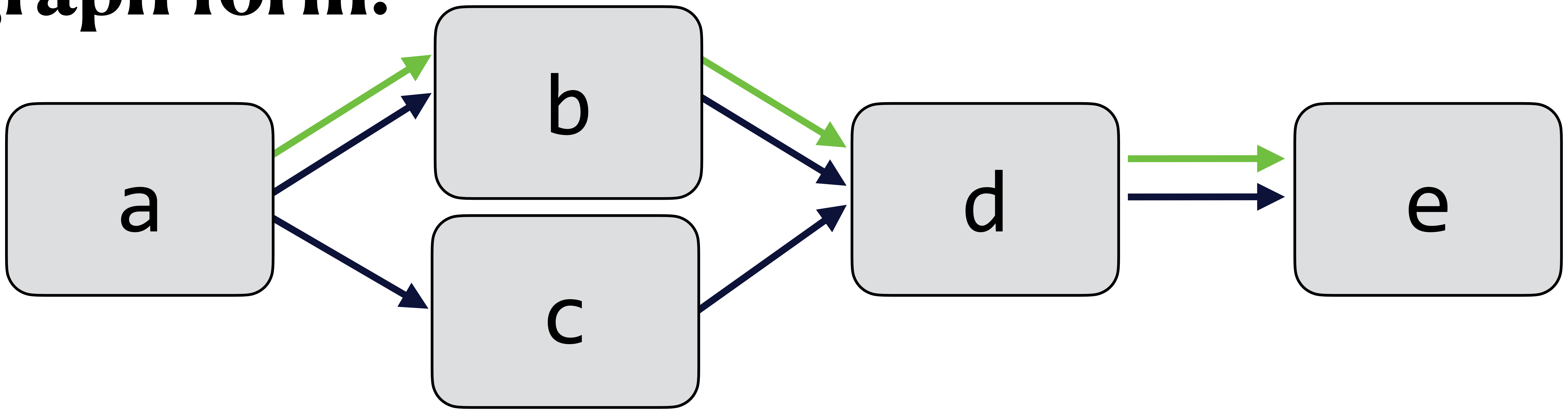
Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

In graph form:



AD Ingredient 2: Reuse of common subexpressions

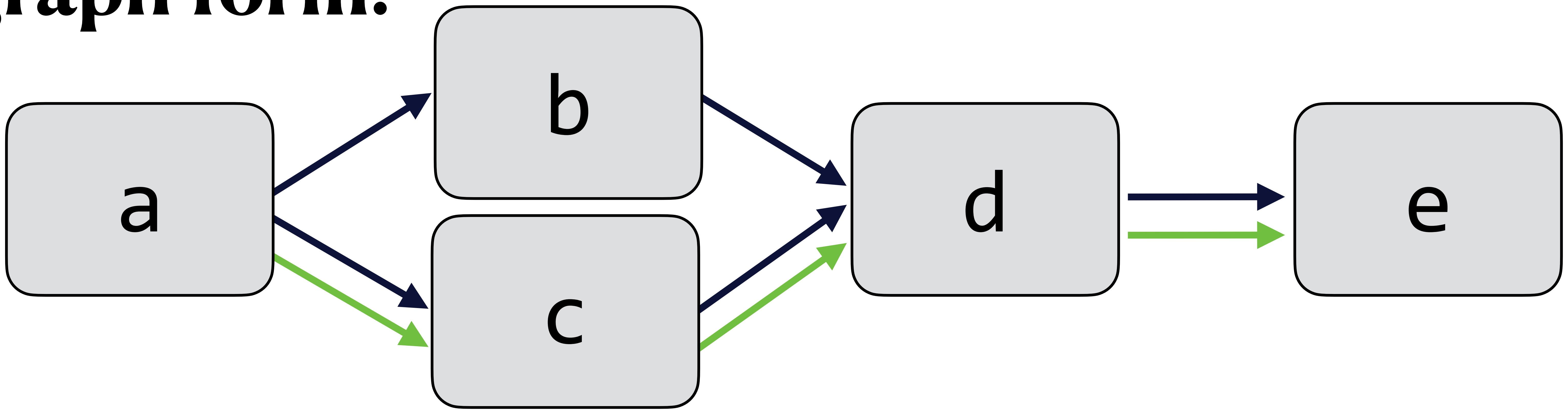
Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

In graph form:



AD Ingredient 2: Reuse of common subexpressions

Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

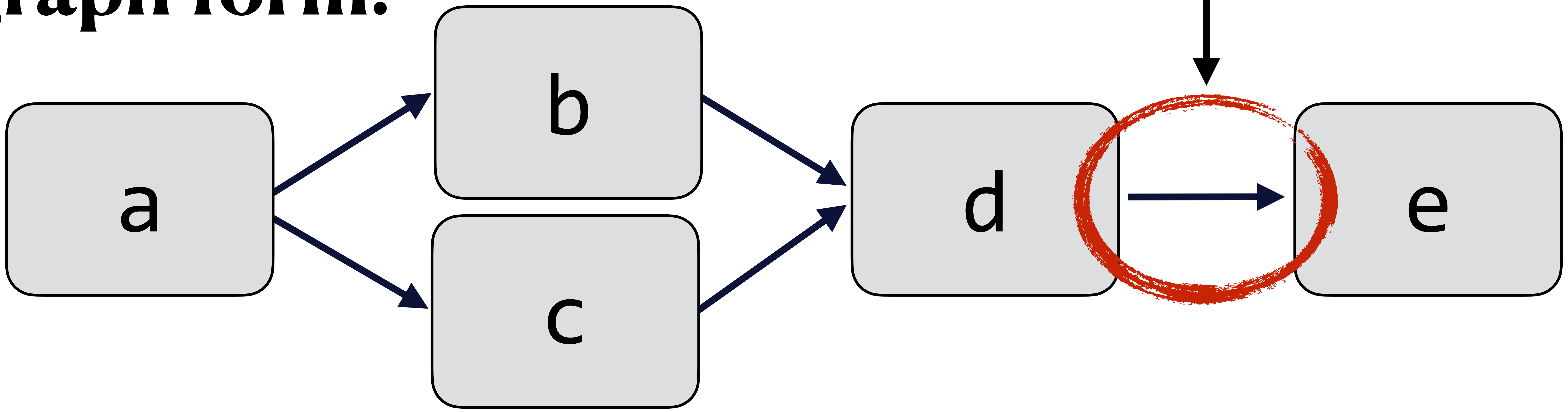
Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

Redundant computation

(as programs grow larger, such unnecessary computation can lead to exponential overheads)

In graph form:



Cheap Gradient Principle

The cost of computing the gradient is nearly the same (typ. < 5x) **as that of simply computing the function itself.**

[Griewank 2008]

- Enables fast computation of high-dimensional gradient (crucial for machine learning and many other applications.)

AD Ingredient 2: Reuse of common subexpressions

Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

Optimized:

```
b = f(a);        db = df(a);
c = g(a);        dc = dg(a);
d = h(b, c);     dd = dh_1(b, c) * db +
                      dh_2(b, c) * dc;
e = k(d);        de = dk(d) * dd;
```

Original

Derivatives

AD Ingredient 2: Reuse of common subexpressions

Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

Optimized:

```
b = f(a);         db = df(a);
c = g(a);         dc = dg(a);
d = h(b, c);      dd = dh_1(b, c) * db +
                       dh_2(b, c) * dc;
e = k(d);         de = dk(d) * dd;
```

Original

Derivatives

AD Ingredient 2: Reuse of common subexpressions

Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

Optimized:

```
b = f(a);          db = df(a);
c = g(a);          dc = dg(a);
d = h(b, c);       dd = dh_1(b, c) * db +
                        dh_2(b, c) * dc;
e = k(d);          de = dk(d) * dd;
```

Original

Derivatives

AD Ingredient 2: Reuse of common subexpressions

Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

Optimized:

```
b = f(a);           db = df(a);
c = g(a);           dc = dg(a);
d = h(b, c);        dd = dh_1(b, c) * db +
                         dh_2(b, c) * dc;
e = k(d);           de = dk(d) * dd;
```

Original

Derivatives

AD Ingredient 2: Reuse of common subexpressions

Original calculation:

```
b = f(a)
c = g(a)
d = h(b, c)
e = k(d)
```

Naïve symbolic derivative:

```
de = dk(h(f(a), g(a))) *
     dh_1(f(a), g(a)) * df(a)
   + dk(h(f(a), g(a))) *
     dh_2(f(a), g(a)) * dg(a)
```

Optimized:

```
b = f(a);        db = df(a);
c = g(a);        dc = dg(a);
d = h(b, c);     dd = dh_1(b, c) * db +
                      dh_2(b, c) * dc;
e = k(d);        de = dk(d) * dd;
```

Original

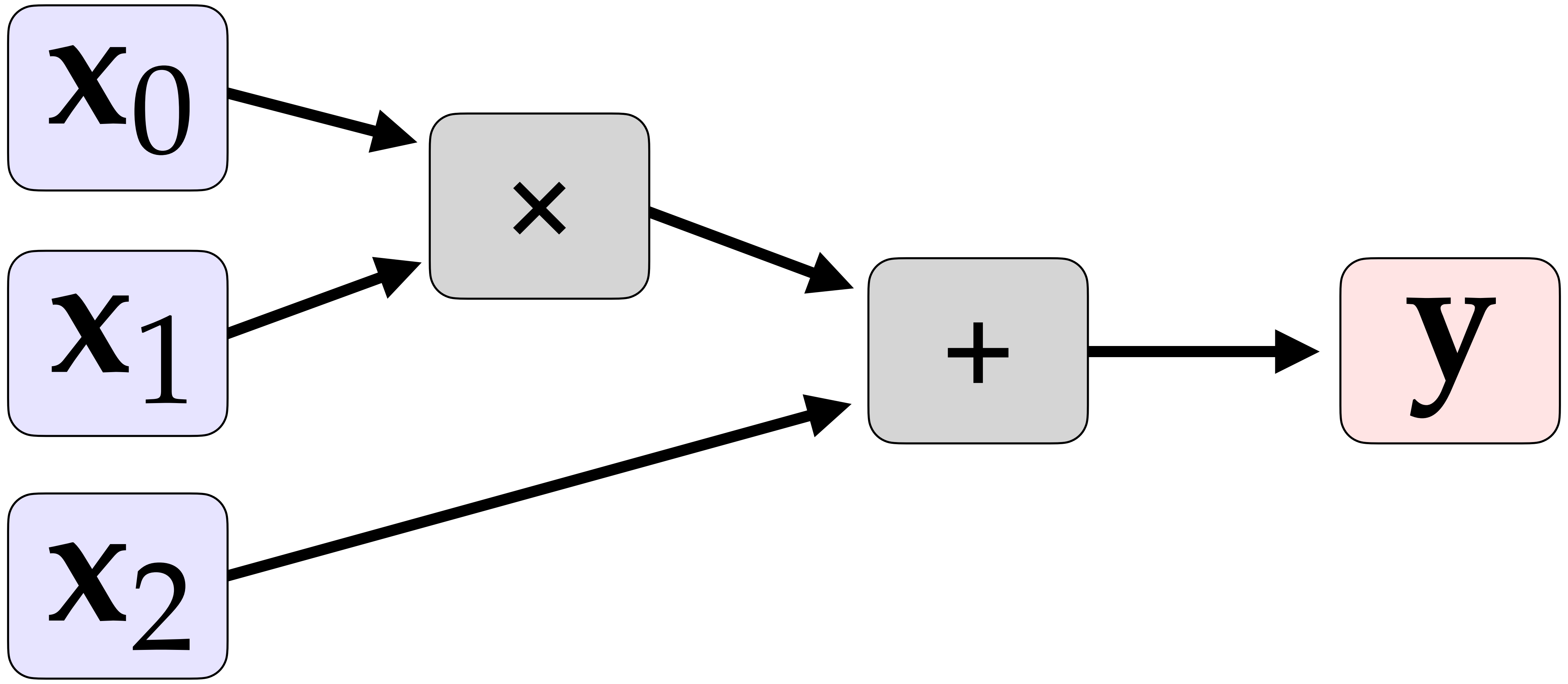
Derivatives

Differentiating a function with respect to an input

Differentiation depends on how variables are *connected*.

This is known as
Forward mode

$$\mathbf{y} = \mathbf{x}_0 \cdot \mathbf{x}_1 + \mathbf{x}_2$$

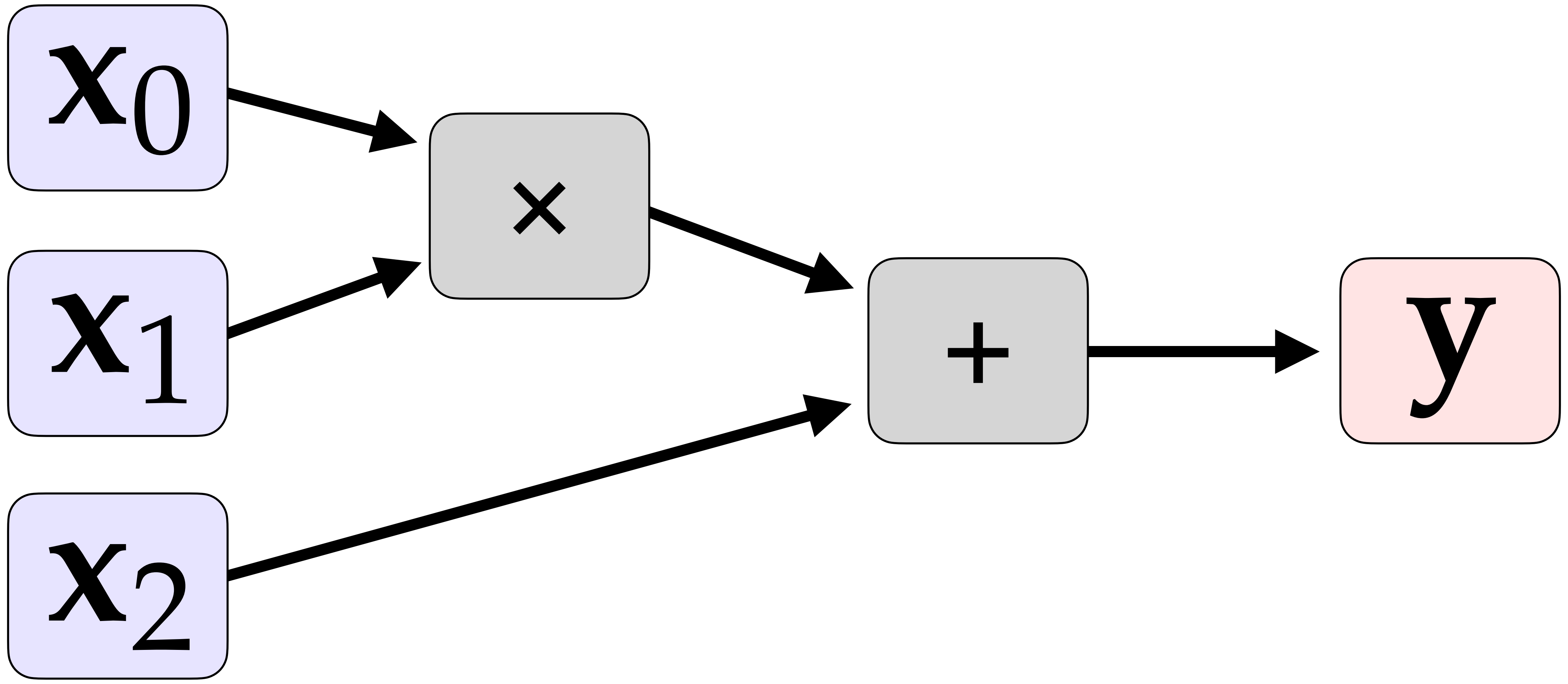


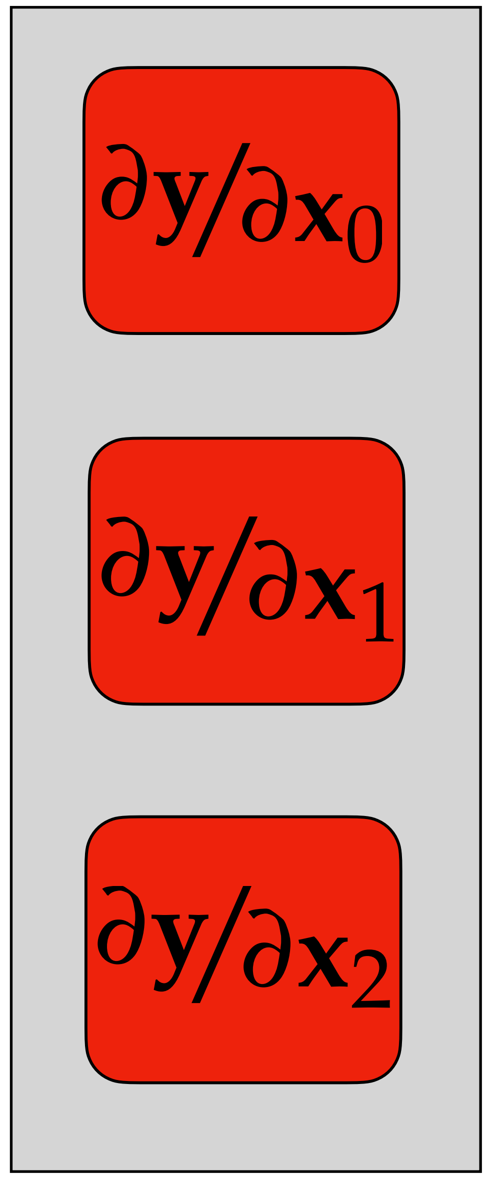
Differentiating a function with respect to an input

Differentiation depends on how variables are *connected*.

This is known as
Forward mode

$$\mathbf{y} = \mathbf{x}_0 \cdot \mathbf{x}_1 + \mathbf{x}_2$$





Gradient

Talk is cheap, show me the code!

One possible implementation strategy: *dual numbers*. (we'll discuss 3 today.)

```
class Number:
    def __init__(self, value: float, grad: float):
        self.value = value
        self.grad = grad
```

Talk is cheap, show me the code!

One possible implementation strategy: *dual numbers*. (we'll discuss 3 today.)

```
class Number:
    def __init__(self, value: float, grad: float):
        self.value = value
        self.grad = grad

    def __add__(self, other: Number):
        return Number(
            value = self.value + other.value,
            grad  = self.grad + other.grad
        )
```

Talk is cheap, show me the code!

One possible implementation strategy: *dual numbers*. (we'll discuss 3 today.)

```python
class Number:
    def __init__(self, value: float, grad: float):
        self.value = value
        self.grad = grad

    def __add__(self, other: Number):
        return Number(
            value = self.value + other.value,
            grad  = self.grad + other.grad
        )

    def __mul__(self, other: Number):
        return Number(
            value = self.value * other.value,
            grad  = self.grad*other.value + self.value*other.grad
        )
```

Talk is cheap, show me the code!

One possible implementation strategy: *dual numbers*. (we'll discuss 3 today.)

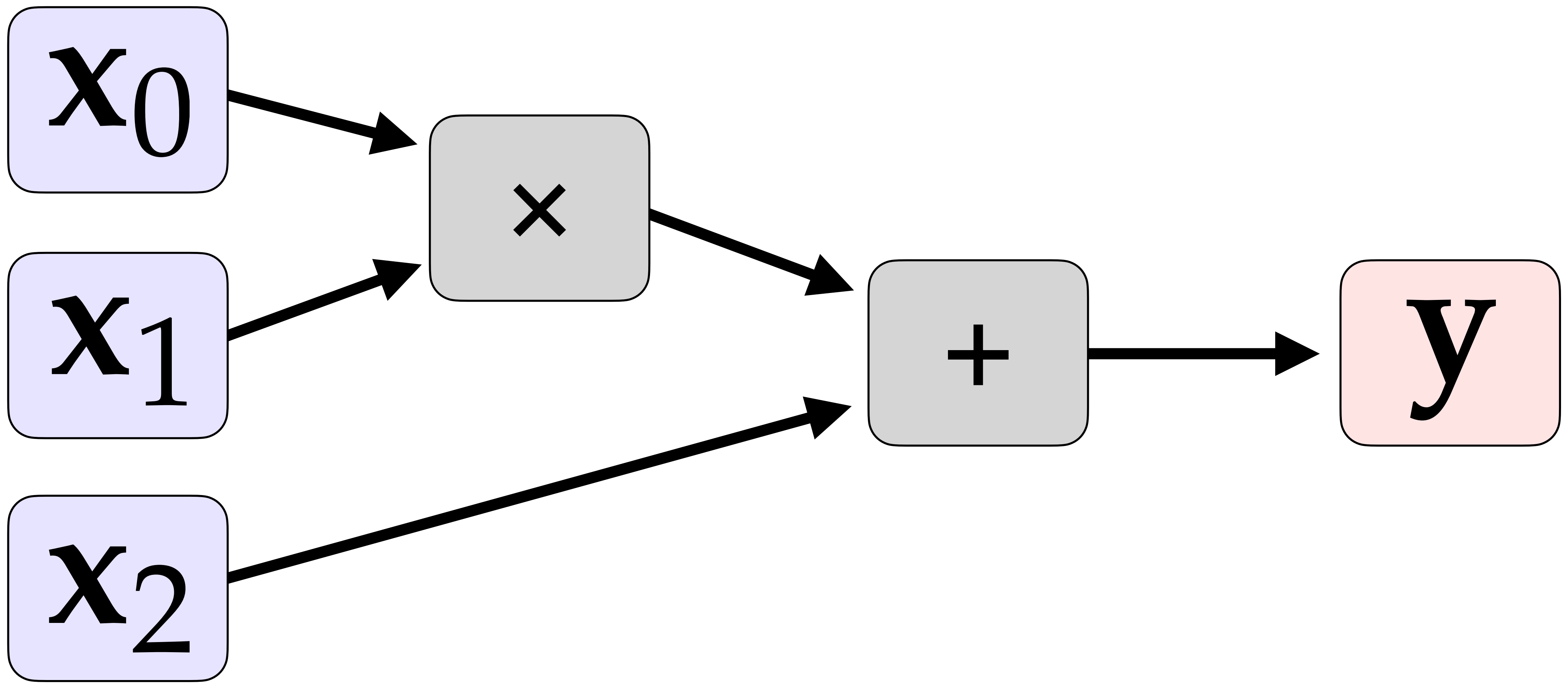
```
class Number:
    def __init__(self, value: float, grad: float):
        self.value = value
        self.grad = grad

    def __add__(self, other: Number):
        return Number(
            value = self.value + other.value,
            grad  = self.grad + other.grad
        )

    def __mul__(self, other: Number):
        return Number(
            value = self.value * other.value,
            grad  = self.grad*other.value + self.value*other.grad
        )
```

Note: this is *not* the approach used in homework 4, please don't copy code from here.

Demo time

Directionality of differentiation

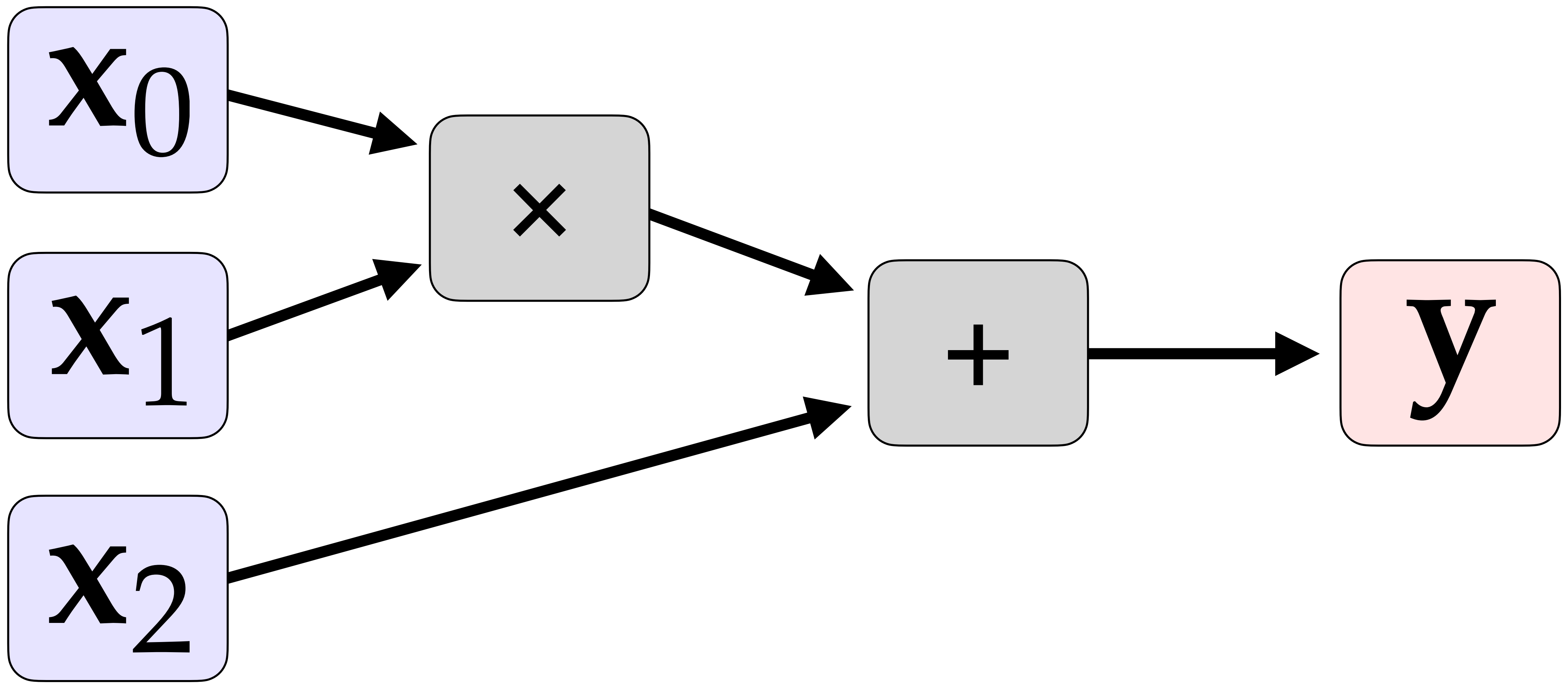
$$\mathbf{y} = \mathbf{x}_0 \cdot \mathbf{x}_1 + \mathbf{x}_2$$



Directionality of differentiation

Reverse mode
aka. backward mode

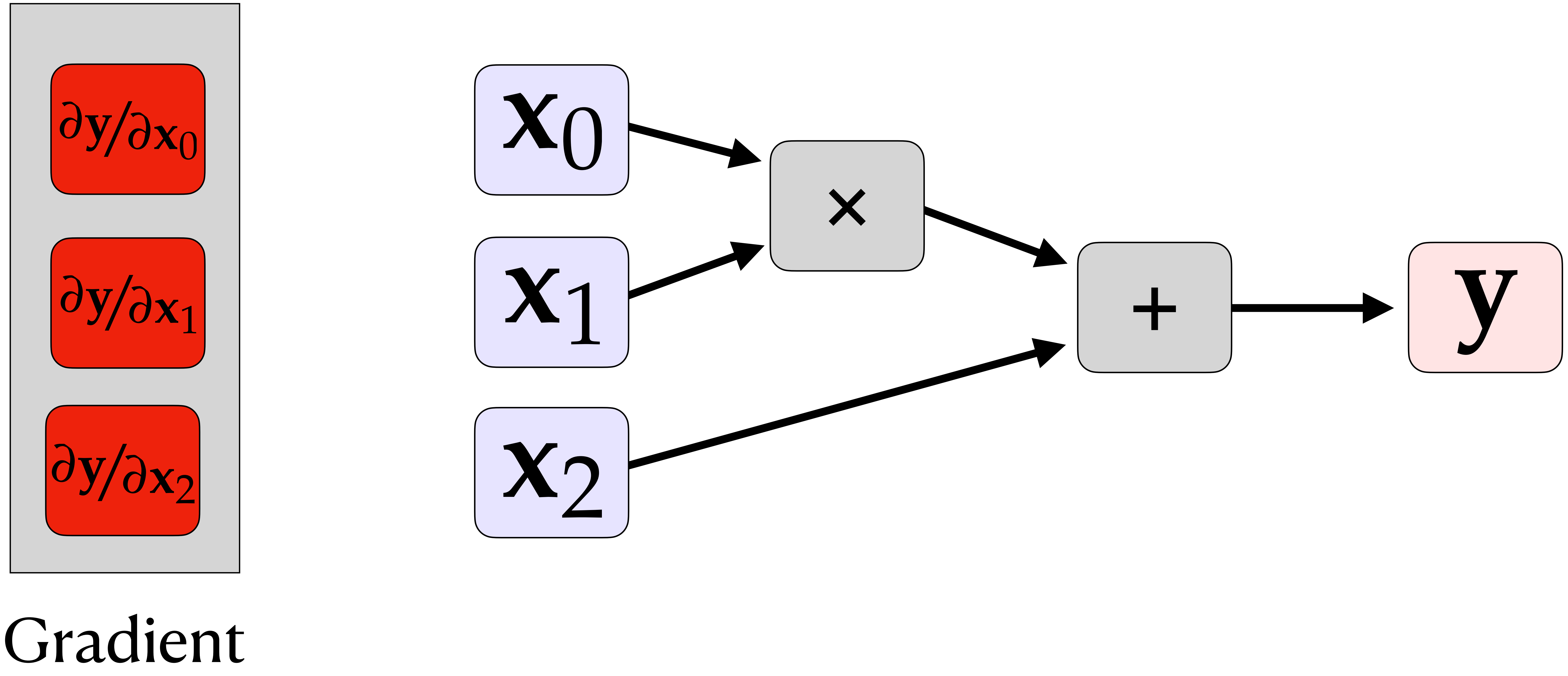
$$\mathbf{y} = \mathbf{x}_0 \cdot \mathbf{x}_1 + \mathbf{x}_2$$



Directionality of differentiation

Reverse mode
aka. backward mode

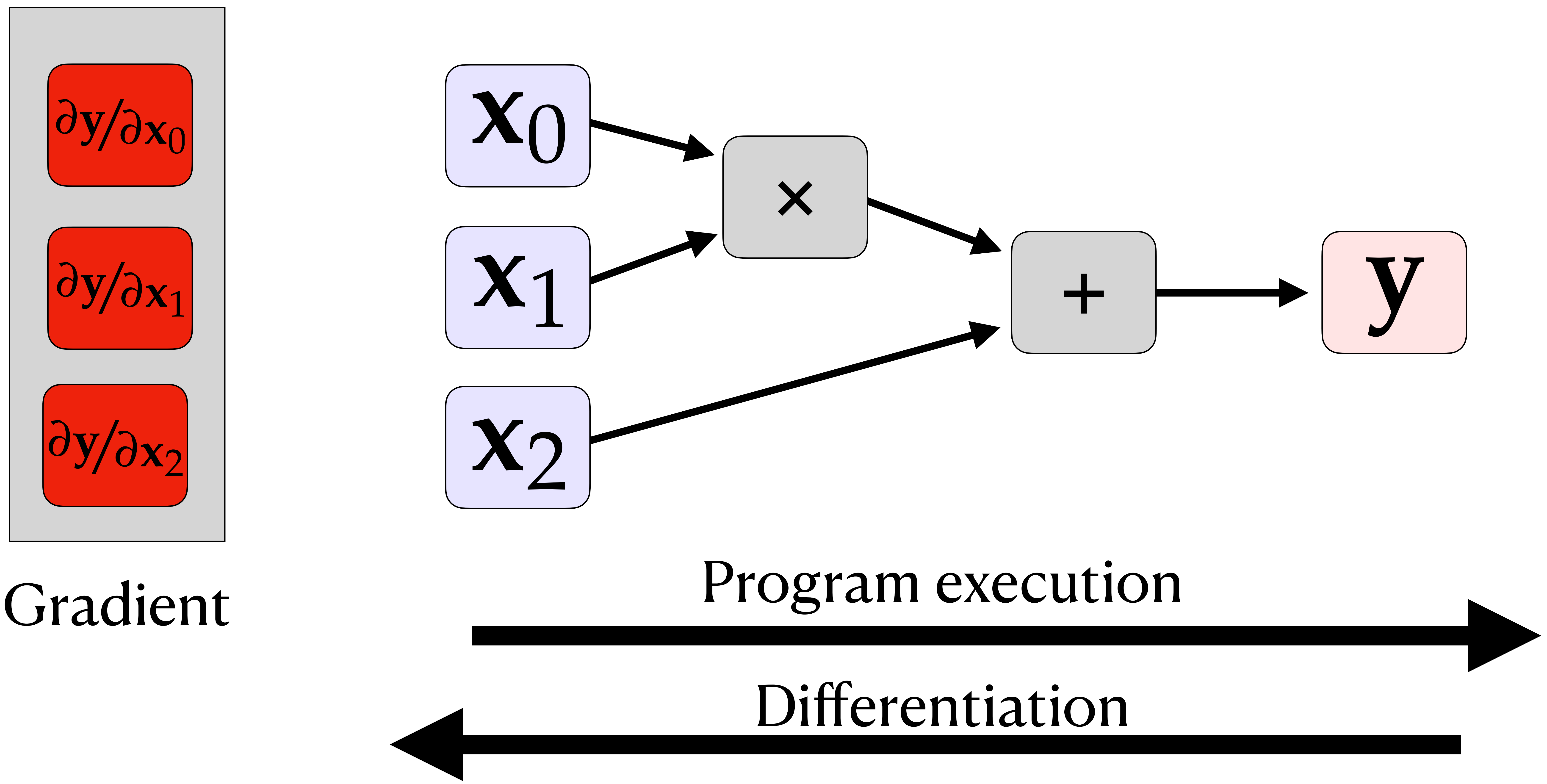
$$\mathbf{y} = \mathbf{x}_0 \cdot \mathbf{x}_1 + \mathbf{x}_2$$



Directionality of differentiation

Reverse mode
aka. backward mode

$$\mathbf{y} = \mathbf{x}_0 \cdot \mathbf{x}_1 + \mathbf{x}_2$$



The Jacobian of multidimensional functions

Another way to think about forward/reverse mode AD

$$\mathbf{J}_f(\mathbf{x})$$

The Jacobian of multidimensional functions

Another way to think about forward/reverse mode AD

$$\mathbf{J}_f(\mathbf{x}) = \frac{f(\mathbf{x})}{\partial \mathbf{x}}$$

The Jacobian of multidimensional functions

Another way to think about forward/reverse mode AD

$$\mathbf{J}_f(\mathbf{x}) = \frac{f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial \mathbf{x}} \\ \vdots \\ \frac{f_n(\mathbf{x})}{\partial \mathbf{x}} \end{bmatrix}$$

The Jacobian of multidimensional functions

Another way to think about forward/reverse mode AD

$$\mathbf{J}_f(\mathbf{x}) = \frac{f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial \mathbf{x}} \\ \vdots \\ \frac{f_n(\mathbf{x})}{\partial \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial x_1} & \dots & \frac{f_1(\mathbf{x})}{\partial x_m} \\ \vdots & & \\ \frac{f_n(\mathbf{x})}{\partial x_1} & \dots & \frac{f_n(\mathbf{x})}{\partial x_m} \end{bmatrix}$$

The Jacobian of multidimensional functions

Another way to think about forward/reverse mode AD

$$\mathbf{J}_f(\mathbf{x}) = \frac{f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial \mathbf{x}} \\ \vdots \\ \frac{f_n(\mathbf{x})}{\partial \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{f_1(\mathbf{x})}{\partial x_m} \\ \vdots & & \\ \frac{f_n(\mathbf{x})}{\partial x_1} & \cdots & \frac{f_n(\mathbf{x})}{\partial x_m} \end{bmatrix}$$

In practice, m and n may be *very large* (> 1 million). **Can't store** a 1M x 1 M entry matrix.

The Jacobian of multidimensional functions

Another way to think about forward/reverse mode AD

$$\mathbf{J}_f(\mathbf{x}) = \frac{f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial \mathbf{x}} \\ \vdots \\ \frac{f_n(\mathbf{x})}{\partial \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{f_1(\mathbf{x})}{\partial x_m} \\ \vdots & & \\ \frac{f_n(\mathbf{x})}{\partial x_1} & \cdots & \frac{f_n(\mathbf{x})}{\partial x_m} \end{bmatrix}$$

In practice, m and n may be *very large* (> 1 million). **Can't store** a 1M x 1 M entry matrix.

"*Magic*" of AD: can efficiently multiply by that matrix without ever having to build it. Two kinds of matrix-vector products are commonly implemented:

The Jacobian of multidimensional functions

Another way to think about forward/reverse mode AD

$$\mathbf{J}_f(\mathbf{x}) = \frac{f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial \mathbf{x}} \\ \vdots \\ \frac{f_n(\mathbf{x})}{\partial \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{f_1(\mathbf{x})}{\partial x_m} \\ \vdots & & \\ \frac{f_n(\mathbf{x})}{\partial x_1} & \cdots & \frac{f_n(\mathbf{x})}{\partial x_m} \end{bmatrix}$$

In practice, m and n may be *very large* (> 1 million). **Can't store** a 1M x 1 M entry matrix.

"*Magic*" of AD: can efficiently multiply by that matrix without ever having to build it. Two kinds of matrix-vector products are commonly implemented:

Forward mode: computes $\mathbf{J}_f\,\mathbf{y}$

The Jacobian of multidimensional functions

Another way to think about forward/reverse mode AD

$$\mathbf{J}_f(\mathbf{x}) = \frac{f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial \mathbf{x}} \\ \vdots \\ \frac{f_n(\mathbf{x})}{\partial \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial x_1} & \dots & \frac{f_1(\mathbf{x})}{\partial x_m} \\ \vdots & & \\ \frac{f_n(\mathbf{x})}{\partial x_1} & \dots & \frac{f_n(\mathbf{x})}{\partial x_m} \end{bmatrix}$$

In practice, m and n may be *very large* (> 1 million). **Can't store** a 1M x 1 M entry matrix.

"*Magic*" of AD: can efficiently multiply by that matrix without ever having to build it. Two kinds of matrix-vector products are commonly implemented:

Forward mode: computes $\mathbf{J}_f\,\mathbf{y}$

(for some given input vector $\mathbf{y}$*)*

The Jacobian of multidimensional functions

Another way to think about forward/reverse mode AD

$$\mathbf{J}_f(\mathbf{x}) = \frac{f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial \mathbf{x}} \\ \vdots \\ \frac{f_n(\mathbf{x})}{\partial \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \frac{f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{f_1(\mathbf{x})}{\partial x_m} \\ \vdots & & \\ \frac{f_n(\mathbf{x})}{\partial x_1} & \cdots & \frac{f_n(\mathbf{x})}{\partial x_m} \end{bmatrix}$$

In practice, m and n may be *very large* (> 1 million). **Can't store** a 1M x 1 M entry matrix.

"*Magic*" of AD: can efficiently multiply by that matrix without ever having to build it. Two kinds of matrix-vector products are commonly implemented:

Forward mode: computes $\mathbf{J}_f\,\mathbf{y}$ *Reverse* mode: computes $\mathbf{J}_f^T\,\mathbf{y}$

(for some given input vector $\mathbf{y}$*)*

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Strategy 2: AD by recording onto a Tape

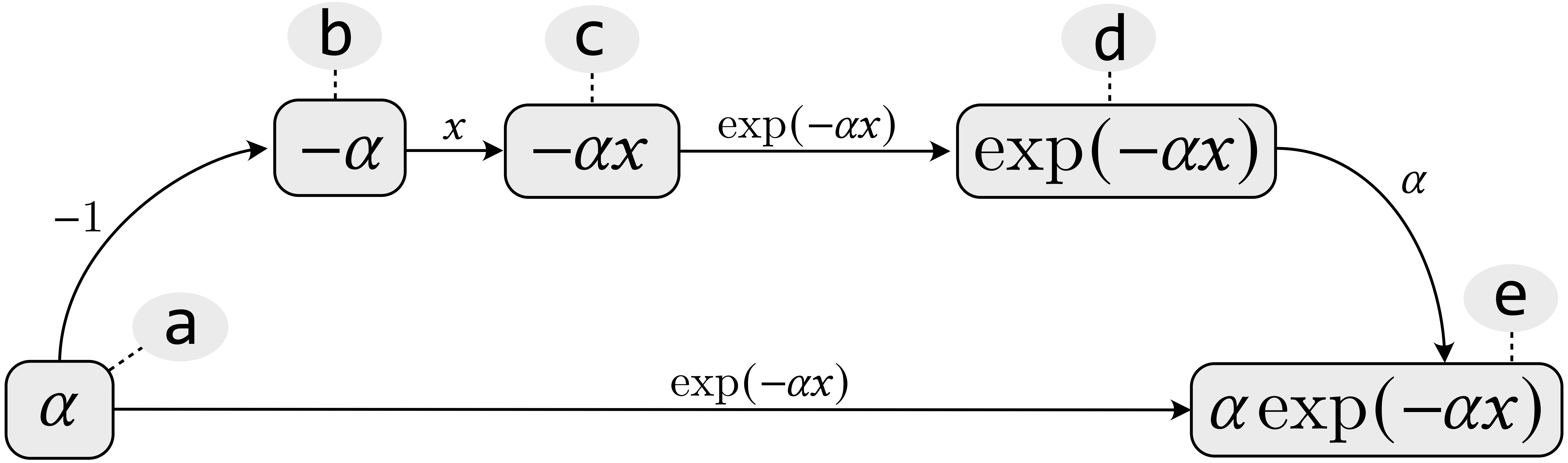
Differentiation task

$$\frac{\partial}{\partial\alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Tape / computation graph

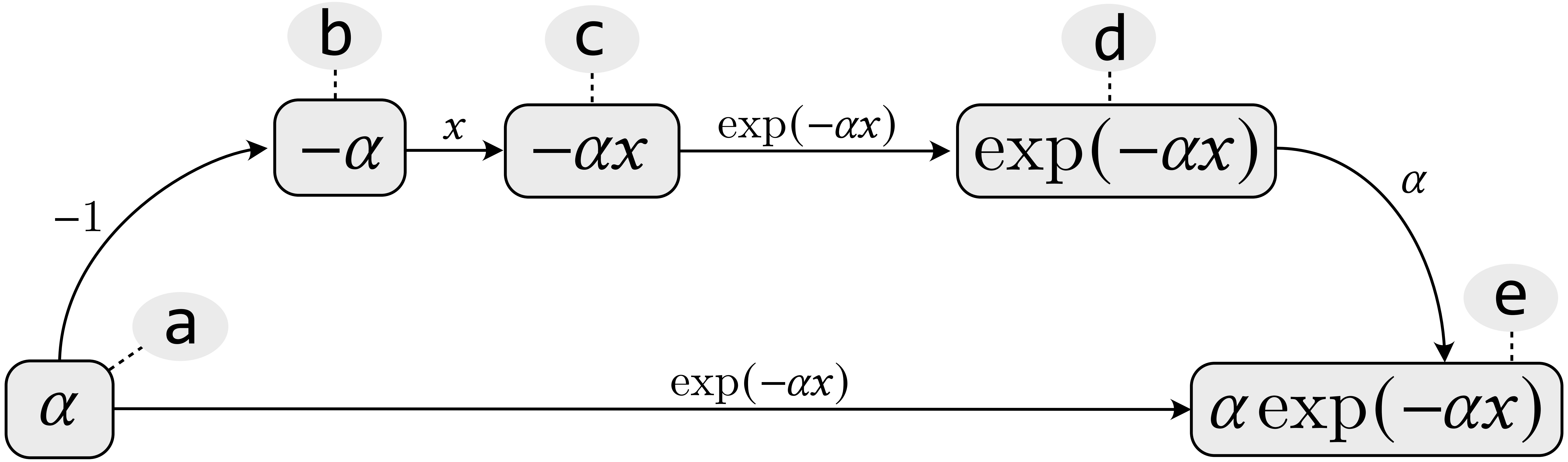


Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

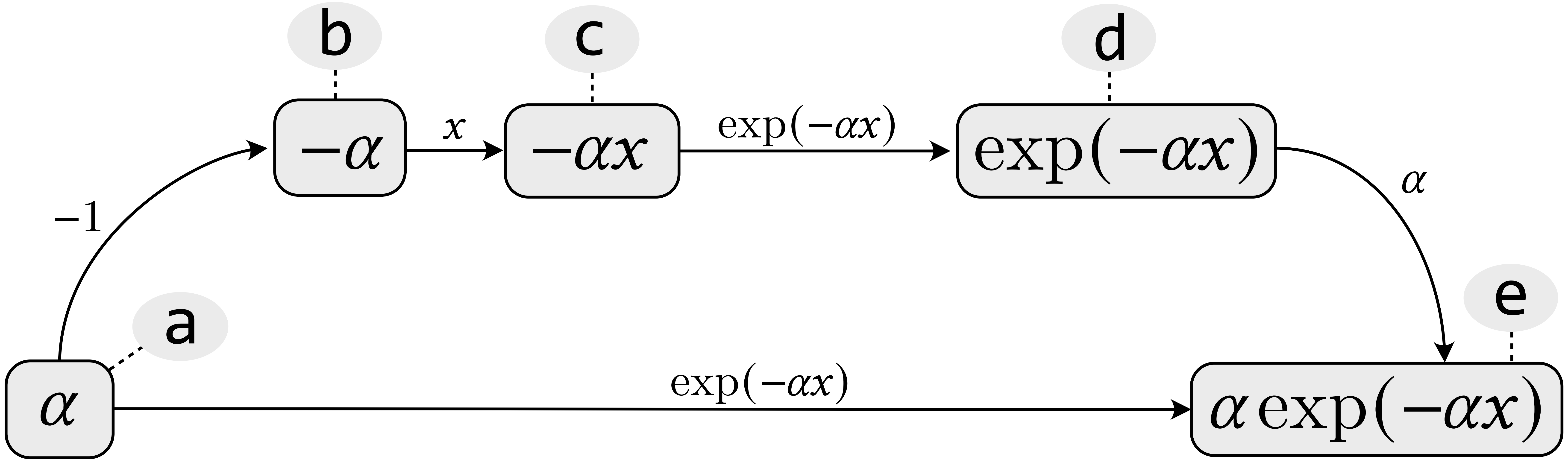
```
δa  = 1
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

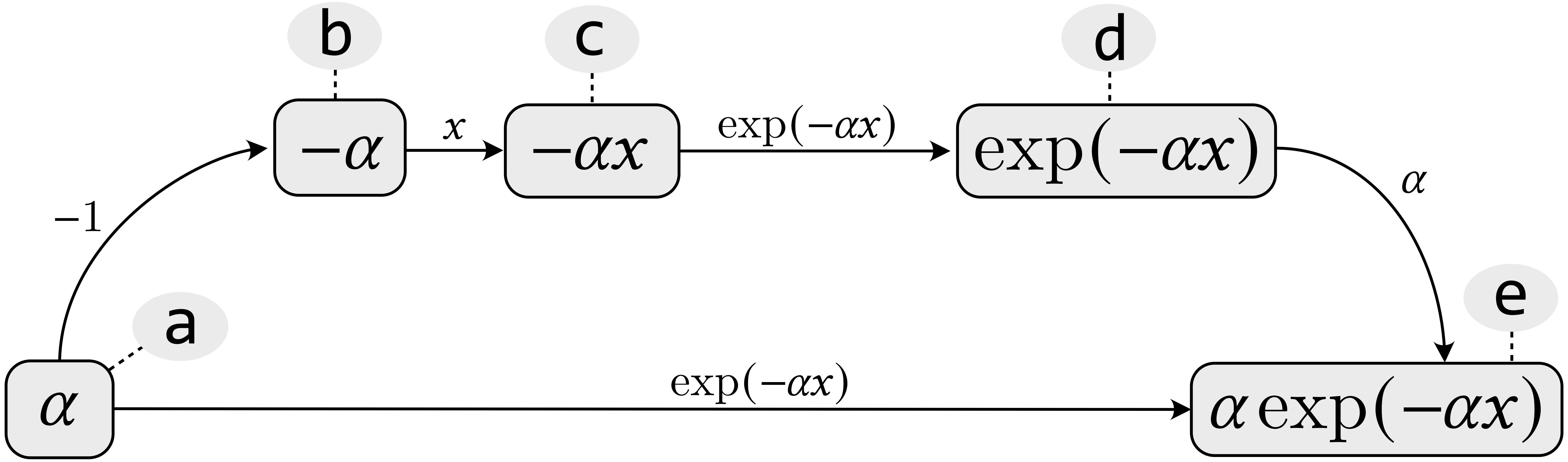
```
δa  = 1
δb  =     -δa = -1
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

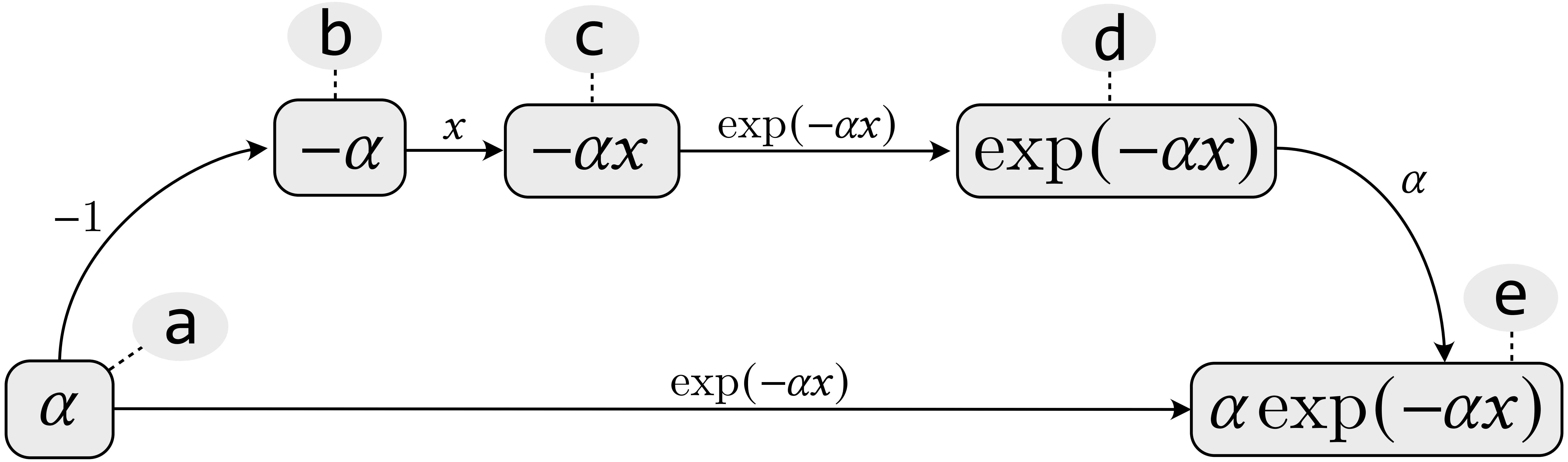
```
δa  = 1
δb  =     -δa = -1
δc  = x · δb = -x
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

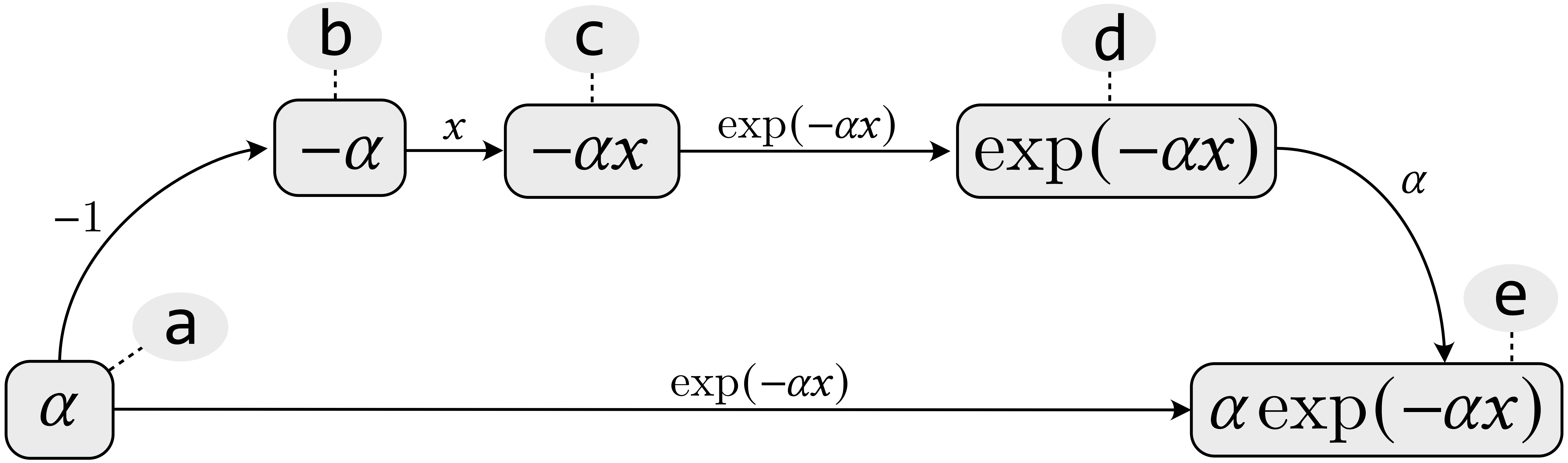
```
δa  = 1
δb  =     -δa = -1
δc  = x · δb = -x
δd  = d · δc = d · -x
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

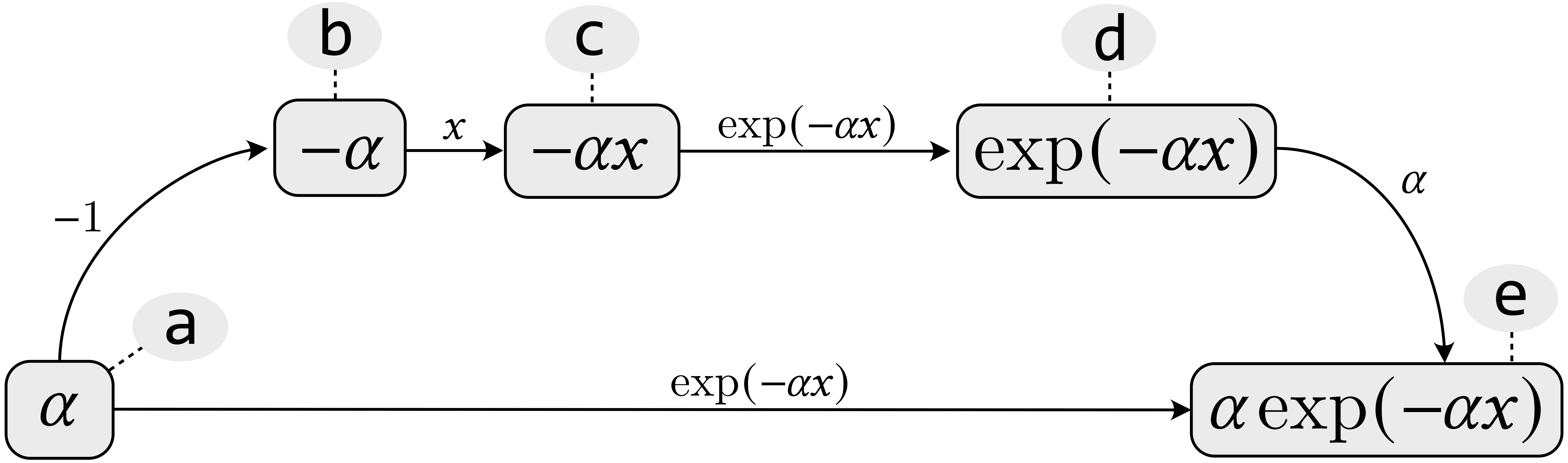
```
δa  = 1
δb  =     -δa = -1
δc  = x · δb = -x
δd  = d · δc = d · -x
δe  = a · δd = a · d · -x
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

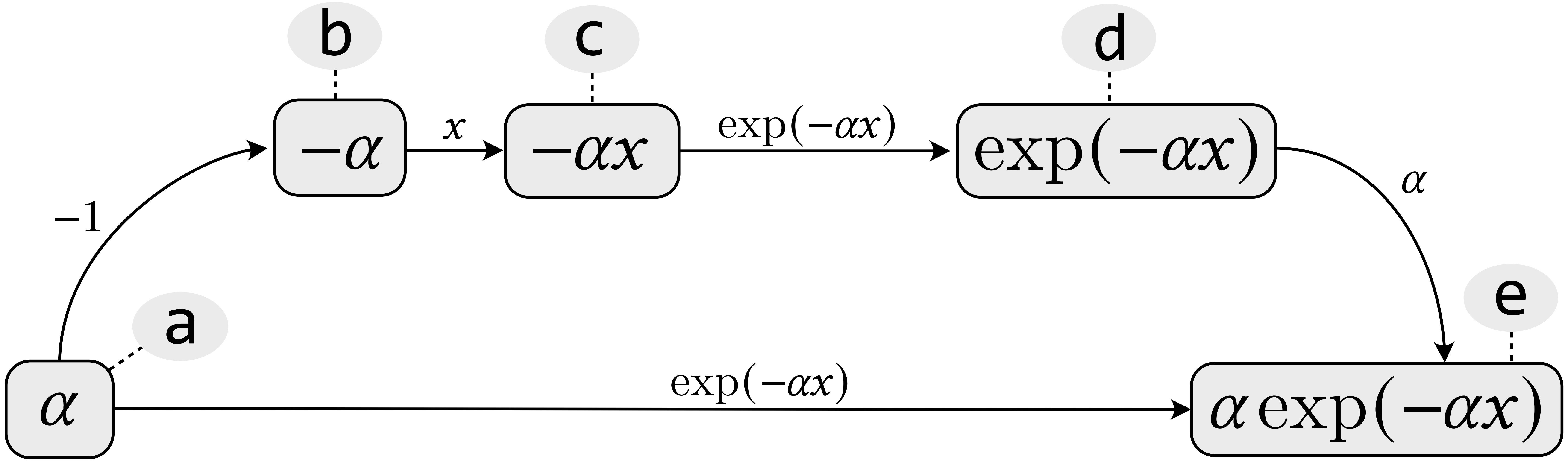
```
δa  = 1
δb  =     -δa = -1
δc  = x · δb = -x
δd  = d · δc = d · -x
δe  = a · δd = a · d · -x
δe += d · δa = a · d · -x + d
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

```
δa  = 1
δb  =     -δa = -1
δc  = x · δb = -x
δd  = d · δc = d · -x
δe  = a · δd = a · d · -x
δe += d · δa = a · d · -x + d
```

Reverse mode

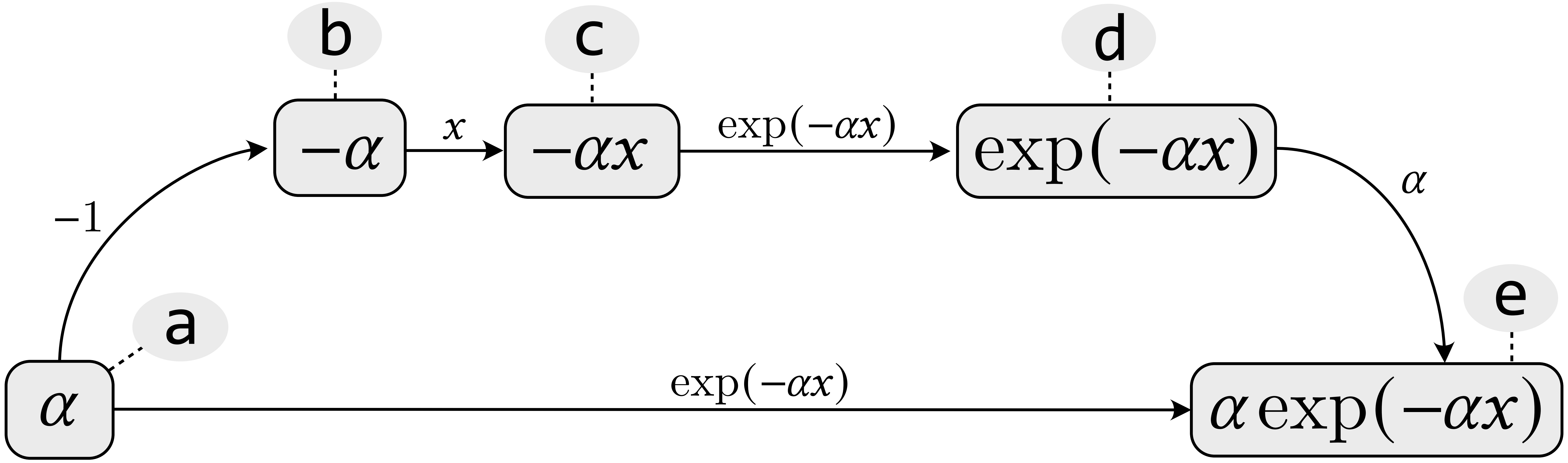
```
δe  = 1
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

```
δa  = 1
δb  =      -δa = -1
δc  = x · δb = -x
δd  = d · δc = d · -x
δe  = a · δd = a · d · -x
δe += d · δa = a · d · -x + d
```

Reverse mode

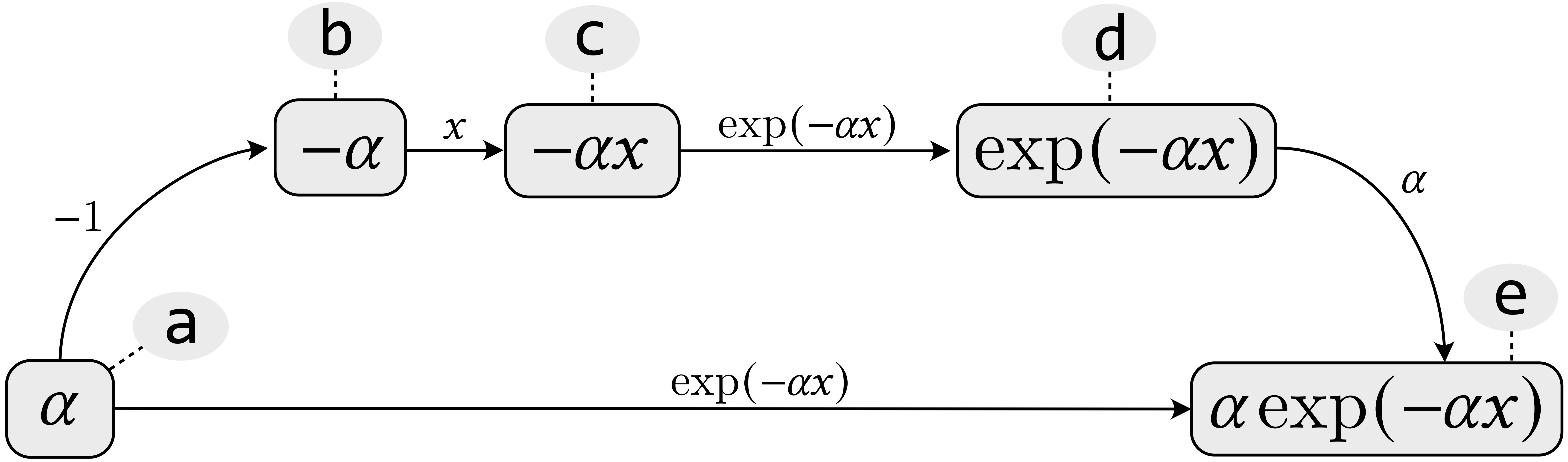
```
δe  = 1
δd  = a · δe = a
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

```
δa  = 1
δb  =     -δa = -1
δc  = x · δb = -x
δd  = d · δc = d · -x
δe  = a · δd = a · d · -x
δe += d · δa = a · d · -x + d
```

Reverse mode

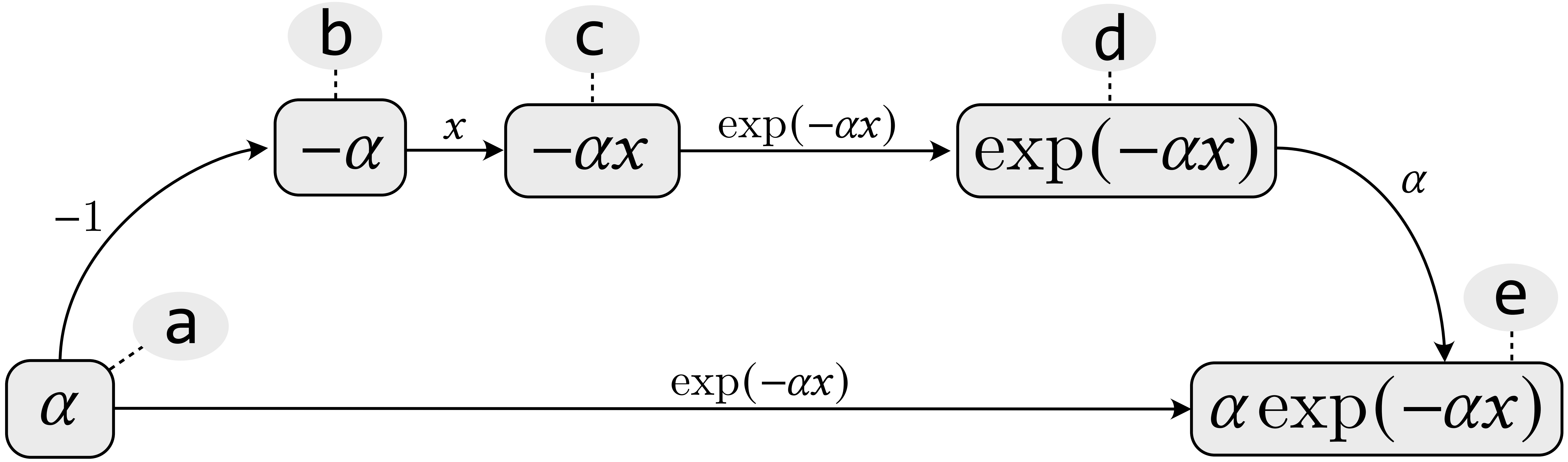
```
δe  = 1
δd  = a · δe = a
δc  = d · δd = a · d
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

```
δa  = 1
δb  =     -δa = -1
δc  = x · δb = -x
δd  = d · δc = d · -x
δe  = a · δd = a · d · -x
δe += d · δa = a · d · -x + d
```

Reverse mode

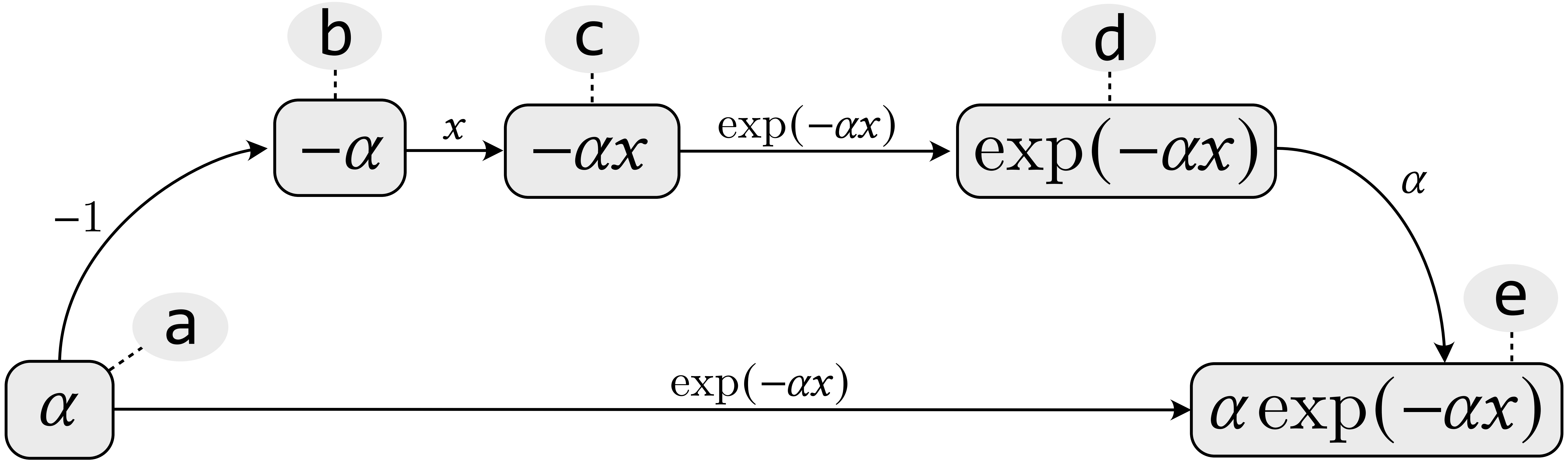
```
δe  = 1
δd  = a · δe = a
δc  = d · δd = a · d
δb  = x · δc = x · a · d
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

```
δa  = 1
δb  =     -δa = -1
δc  = x · δb = -x
δd  = d · δc = d · -x
δe  = a · δd = a · d · -x
δe += d · δa = a · d · -x + d
```

Reverse mode

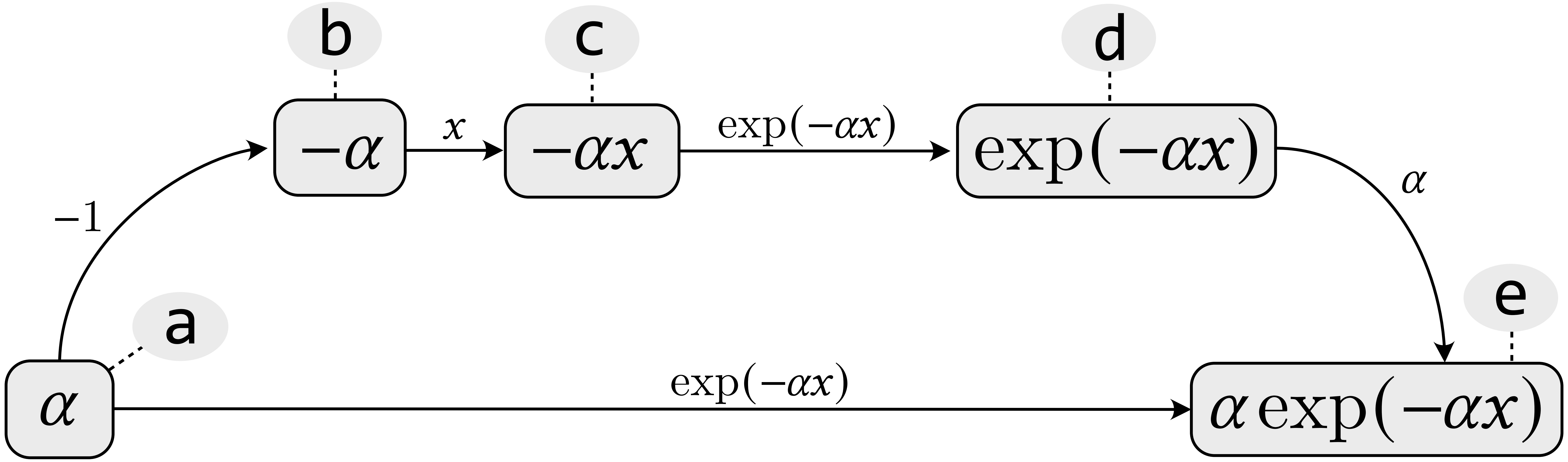
```
δe  = 1
δd  = a · δe = a
δc  = d · δd = a · d
δb  = x · δc = x · a · d
δa  =     -δb = -x · a · d
```

Strategy 2: AD by recording onto a Tape

Differentiation task

$$\frac{\partial}{\partial \alpha}\left[\alpha \cdot \exp(-\alpha x)\right]$$

Tape / computation graph



Primal

```
a = α
b = -a
c = x · b
d = exp(c)
e = a · d
```

Forward mode

```
δa  = 1
δb  =     -δa = -1
δc  = x · δb = -x
δd  = d · δc = d · -x
δe  = a · δd = a · d · -x
δe += d · δa = a · d · -x + d
```

Reverse mode

```
δe  = 1
δd  = a · δe = a
δc  = d · δd = a · d
δb  = x · δc = x · a · d
δa  =     -δb = -x · a · d
δa += d · δe = -x · a · d + d
```

Strategy 3: AD by code transformation

(One can of course also do this by hand, but computer assistance is helpful.)

```python
def pow(x: float, n: int):
    y = 1
    for i in range(n):
        y *= x
    return y
```

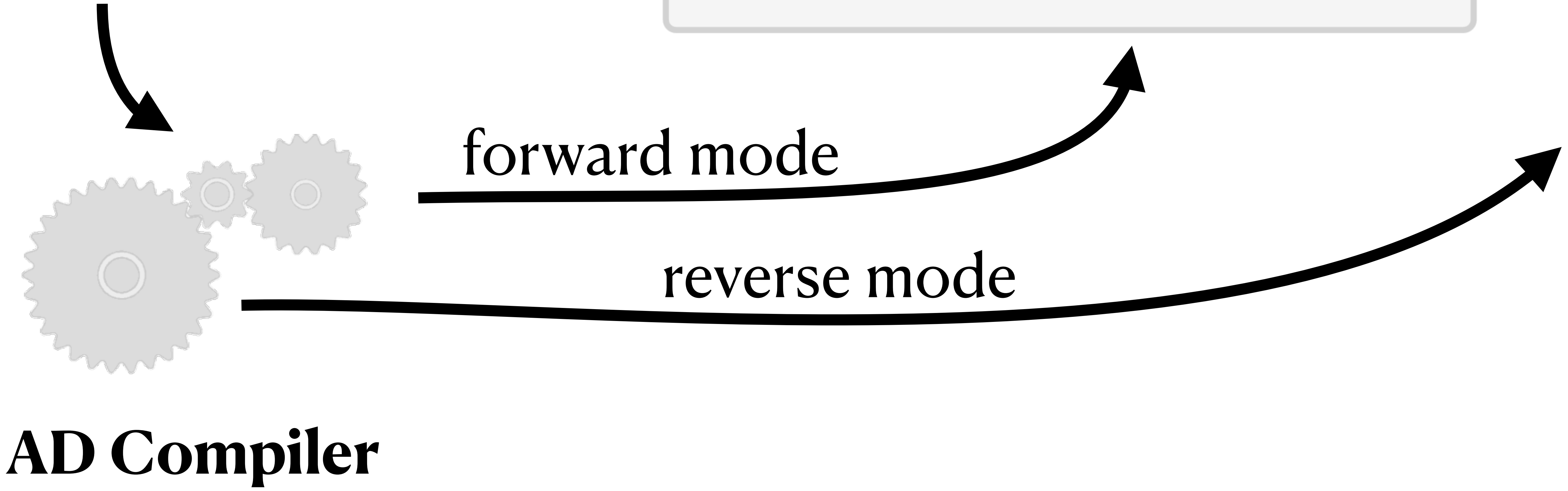
Strategy 3: AD by code transformation

(One can of course also do this by hand, but computer assistance is helpful.)

```python
def pow(x: float, n: int):
    y = 1
    for i in range(n):
        y *= x
    return y
```

```python
def pow_jvp(x, n, dx):
    y, dy = 1, 0
    for i in range(n):
        dy = x*dy + y*dx
        y *= x
    return dy
```

```python
def pow_vjp(x, n, dy):
    y, dx = 1, 0
    stack = []
    for i in range(n):
        stack.append(y)
        y *= x
    for i in reversed(range(n)):
        y = stack.pop()
        dx += y*dy
        dy *= x
    return dx
```



Technique tree

Technique tree

Forward mode

Use when the function has **few inputs** and **many outputs**,
and if you want the derivative wrt. all outputs at once.

Technique tree

Forward mode

Use when the function has **few inputs** and **many outputs**, and if you want the derivative wrt. all outputs at once.

Reverse mode

Use when the function has **few outputs** and **many inputs**, and if you want the derivative wrt. all inputs at once.

Technique tree

Forward mode

Use when the function has **few inputs** and **many outputs**, and if you want the derivative wrt. all outputs at once.

Reverse mode

Use when the function has **few outputs** and **many inputs**, and if you want the derivative wrt. all inputs at once.

AD Implementation strategies

Technique tree

Forward mode

Use when the function has **few inputs** and **many outputs**, and if you want the derivative wrt. all outputs at once.

Reverse mode

Use when the function has **few outputs** and **many inputs**, and if you want the derivative wrt. all inputs at once.

AD Implementation strategies

Dual numbers

Simple and efficient.

Technique tree

Forward mode

Use when the function has **few inputs** and **many outputs**, and if you want the derivative wrt. all outputs at once.

Reverse mode

Use when the function has **few outputs** and **many inputs**, and if you want the derivative wrt. all inputs at once.

AD Implementation strategies

Dual numbers

Simple and efficient.

Tape

Simple, but inefficient (memory usage).

Technique tree

Forward mode

Use when the function has **few inputs** and **many outputs**, and if you want the derivative wrt. all outputs at once.

Reverse mode

Use when the function has **few outputs** and **many inputs**, and if you want the derivative wrt. all inputs at once.

AD Implementation strategies

Dual numbers

Simple and efficient.

Tape

Simple, but inefficient (memory usage).

Code transformation / by hand

Complicated, ends up being effectively the same as dual numbers.

Technique tree

Forward mode

Use when the function has **few inputs** and **many outputs**, and if you want the derivative wrt. all outputs at once.

Reverse mode

Use when the function has **few outputs** and **many inputs**, and if you want the derivative wrt. all inputs at once.

AD Implementation strategies

Forward mode

Dual numbers

Simple and efficient.

Tape

Simple, but inefficient (memory usage).

Code transformation / by hand

Complicated, ends up being effectively the same as dual numbers.

Reverse mode

Dual numbers

Does not make sense for reverse mode.

Technique tree

Forward mode

Use when the function has **few inputs** and **many outputs**, and if you want the derivative wrt. all outputs at once.

Reverse mode

Use when the function has **few outputs** and **many inputs**, and if you want the derivative wrt. all inputs at once.

AD Implementation strategies

Forward mode

Dual numbers

Simple and efficient.

Tape

Simple, but inefficient (memory usage).

Code transformation / by hand

Complicated, ends up being effectively the same as dual numbers.

Reverse mode

Dual numbers

Does not make sense for reverse mode.

Tape

What everyone uses in practice.

Technique tree

Forward mode

Use when the function has **few inputs** and **many outputs**, and if you want the derivative wrt. all outputs at once.

Reverse mode

Use when the function has **few outputs** and **many inputs**, and if you want the derivative wrt. all inputs at once.

AD Implementation strategies

Forward mode

Dual numbers

Simple and efficient.

Tape

Simple, but inefficient (memory usage).

Code transformation / by hand

Complicated, ends up being effectively the same as dual numbers.

Reverse mode

Dual numbers

Does not make sense for reverse mode.

Tape

What everyone uses in practice.

Code transformation / by hand

Promising but not widely used.

Technique tree

Forward mode	Reverse mode
Use when the function has **few inputs** and **many outputs**, and if you want the derivative wrt. all outputs at once.	Use when the function has **few outputs** and **many inputs**, and if you want the derivative wrt. all inputs at once.

AD Implementation strategies

Forward mode	Reverse mode
Dual numbers Simple and efficient.	**Dual numbers** Does not make sense for reverse mode.
Tape Simple, but inefficient (memory usage).	**Tape** What everyone uses in practice.
Code transformation / by hand Complicated, ends up being effectively the same as dual numbers.	**Code transformation / by hand** Promising but not widely used.

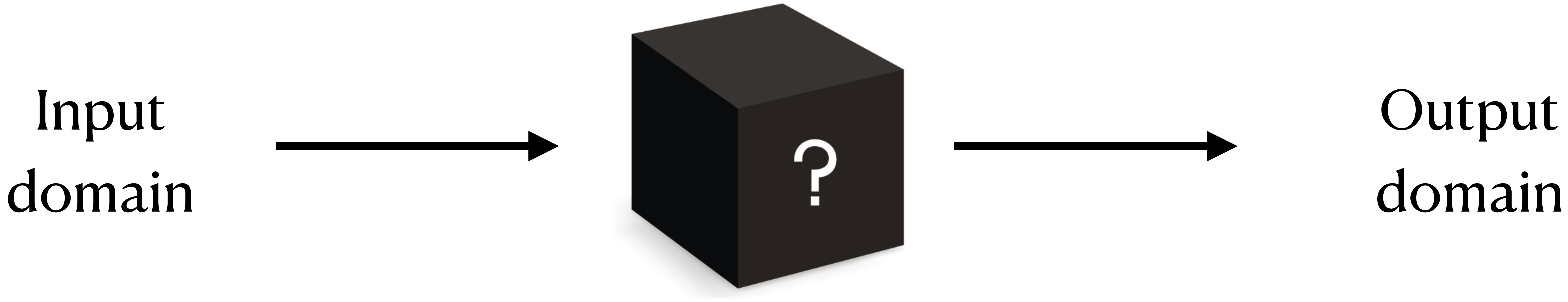
We have gradients.

Now, what?

"Inverse problems"

Previously, f was a linear function, which was very useful. But what if it isn't?

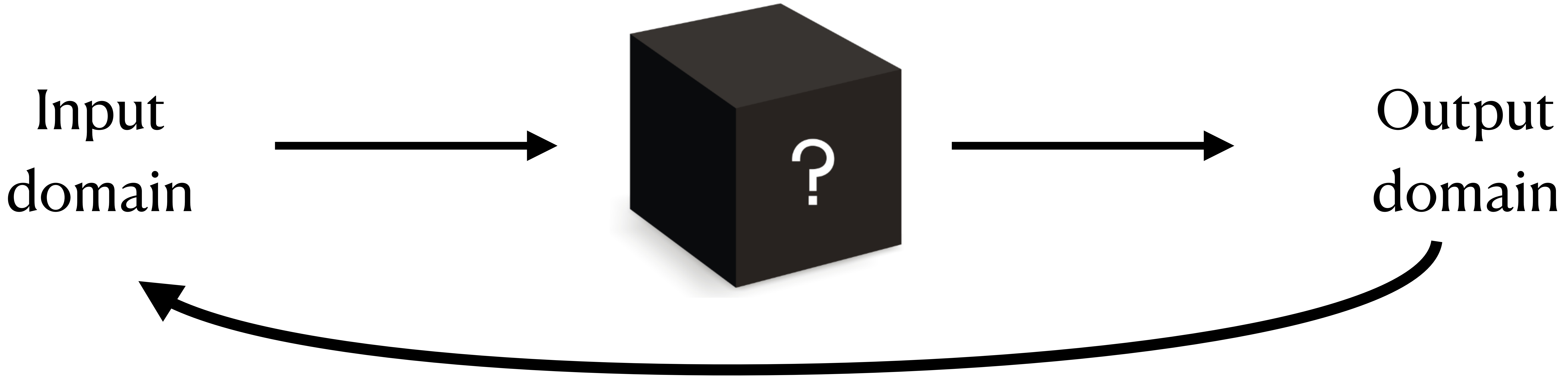
$$f(\mathbf{x})$$



"Inverse problems"

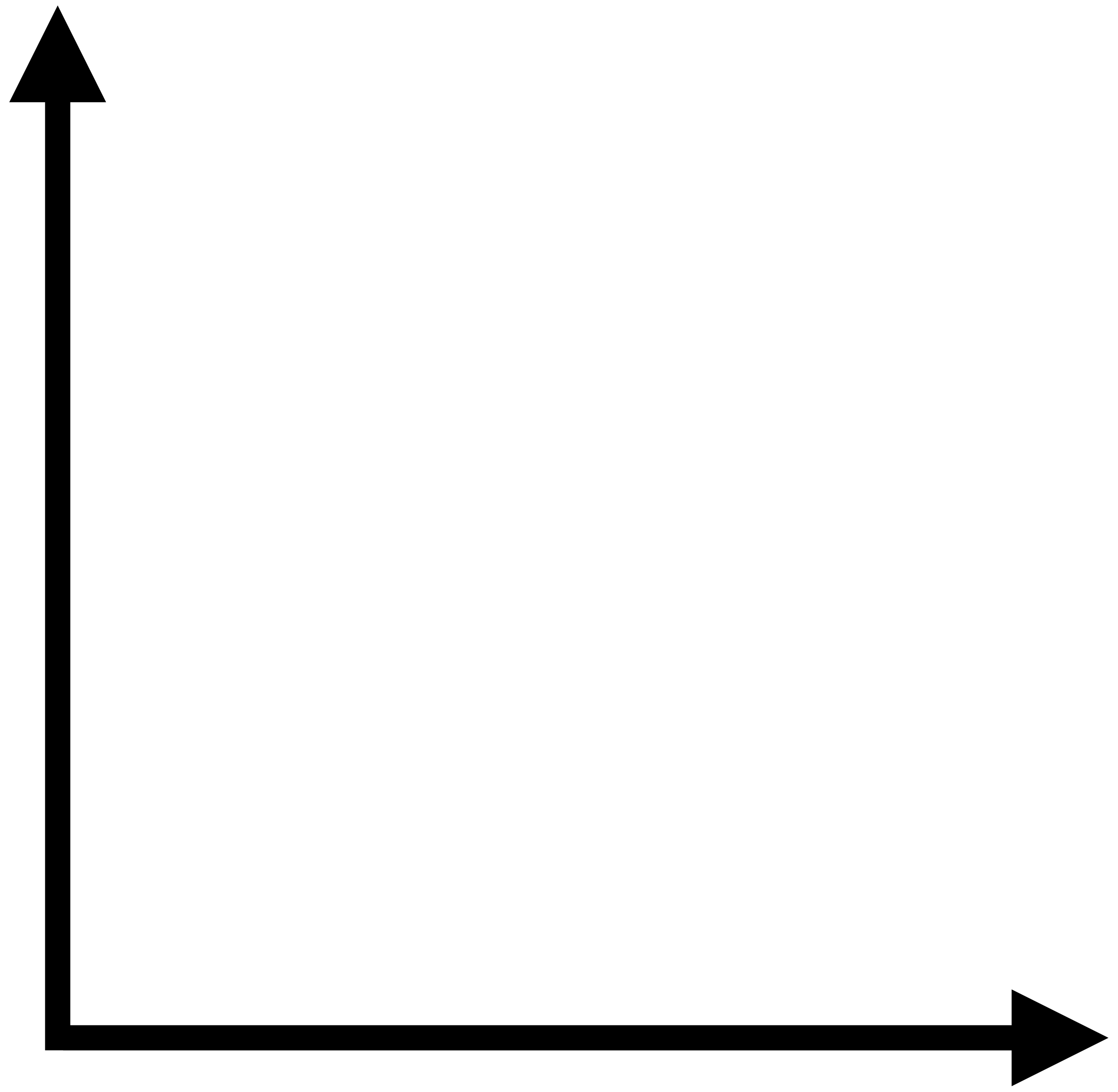
Previously, f was a linear function, which was very useful. But what if it isn't?

$$f(\mathbf{x})$$



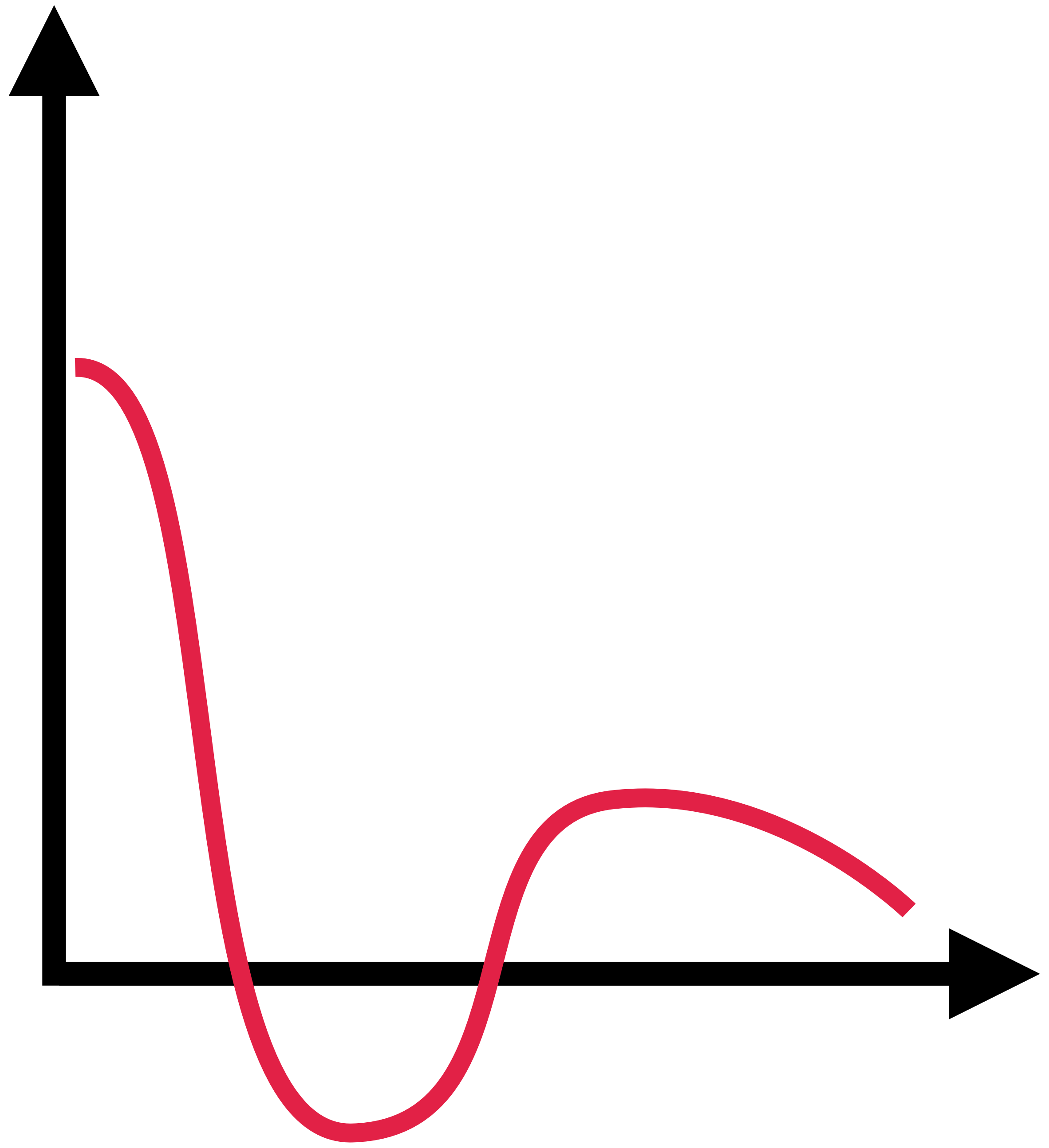
$$f^{-1}(\mathbf{x}) \ ?$$

Two important kinds of inverse problems



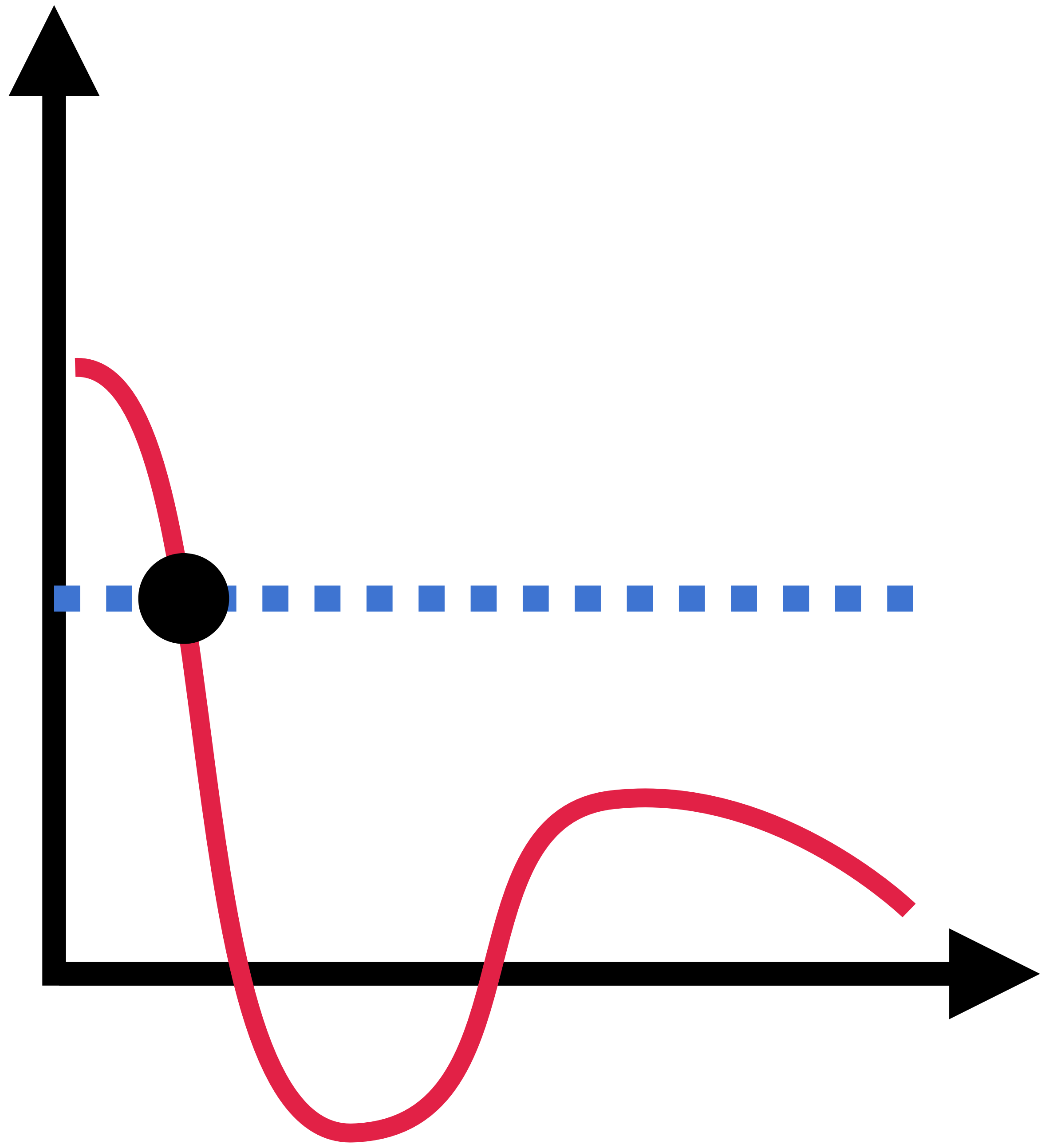
Root finding

Two important kinds of inverse problems



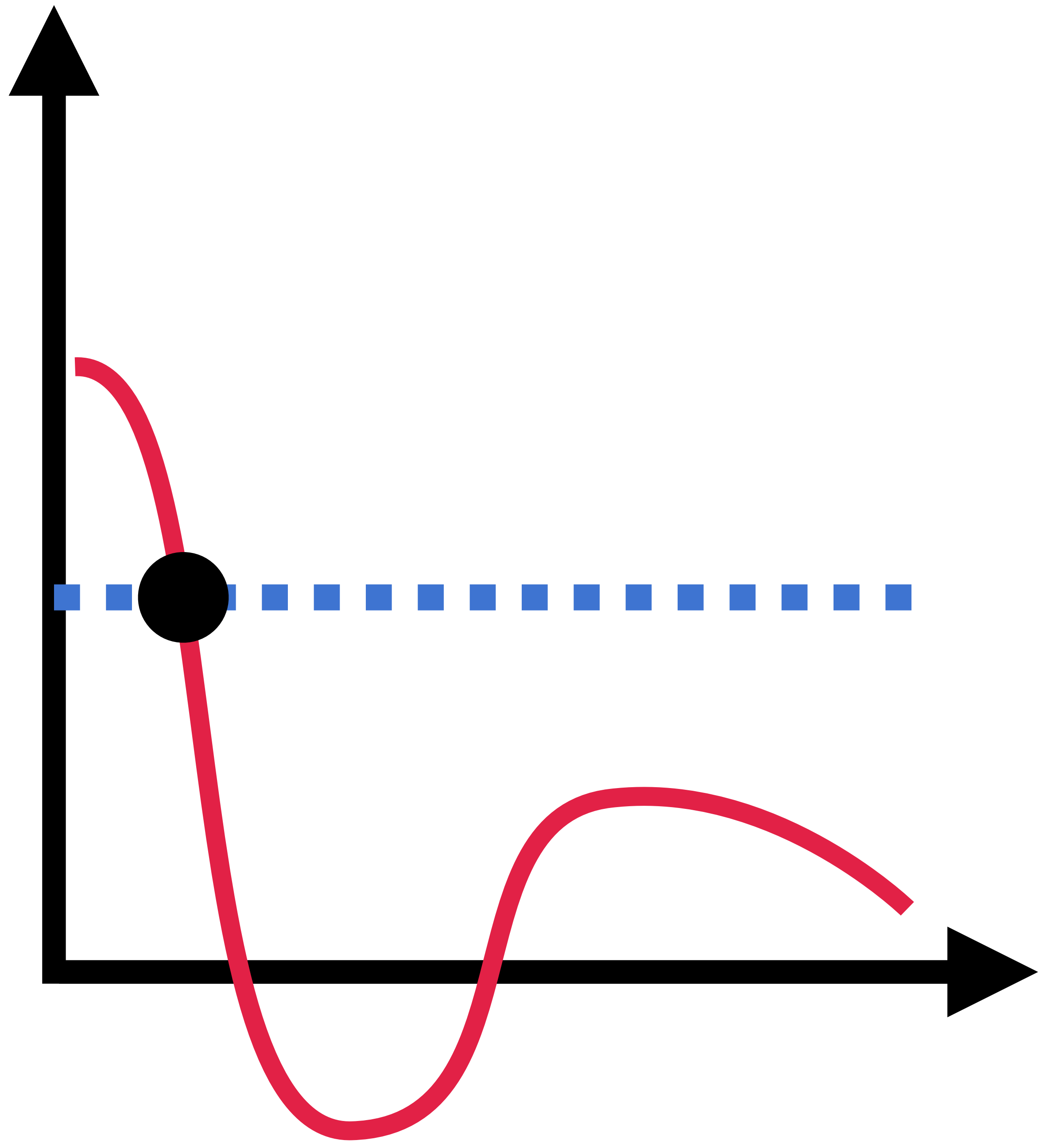
Root finding

Two important kinds of inverse problems

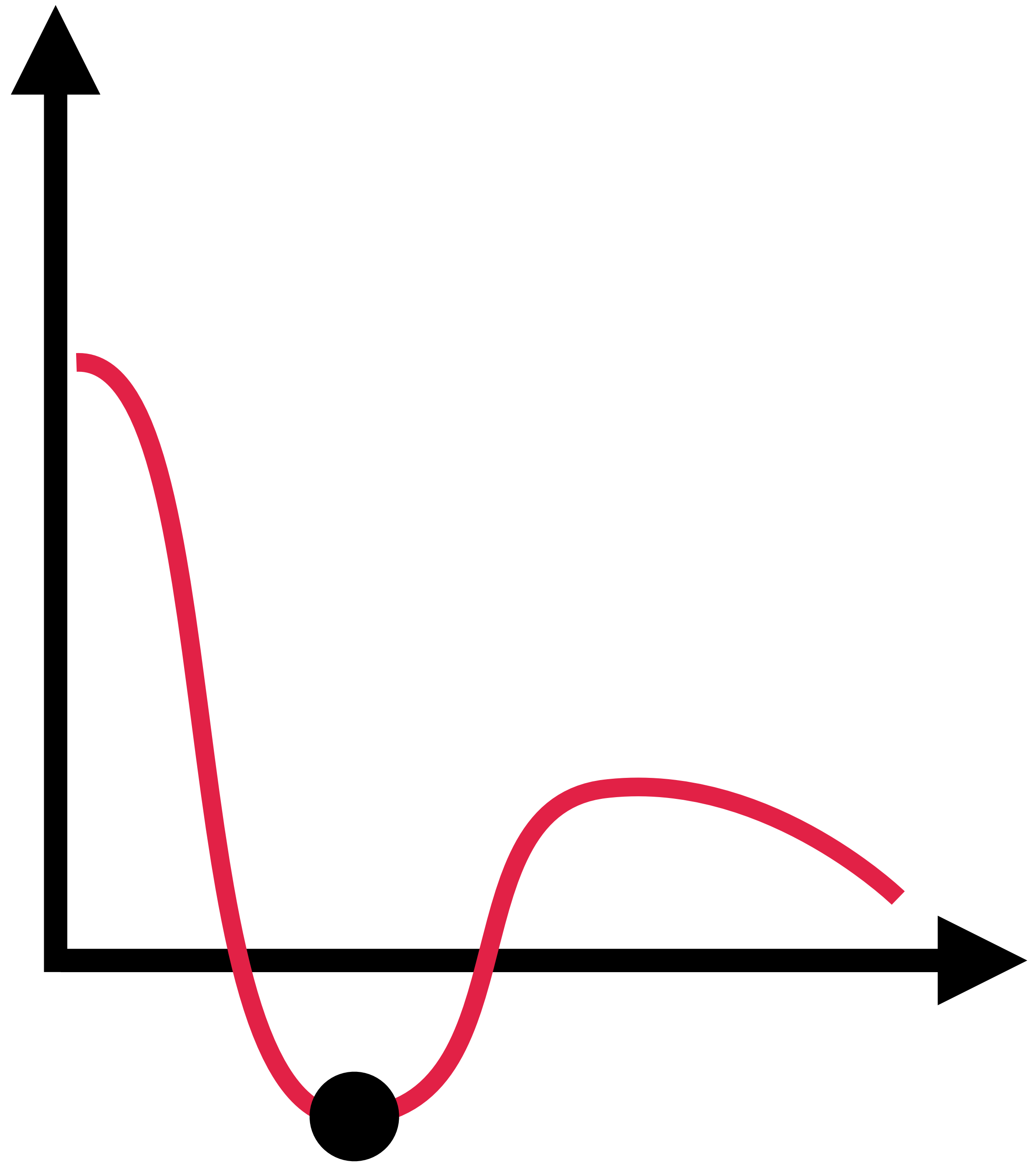


Root finding

Two important kinds of inverse problems

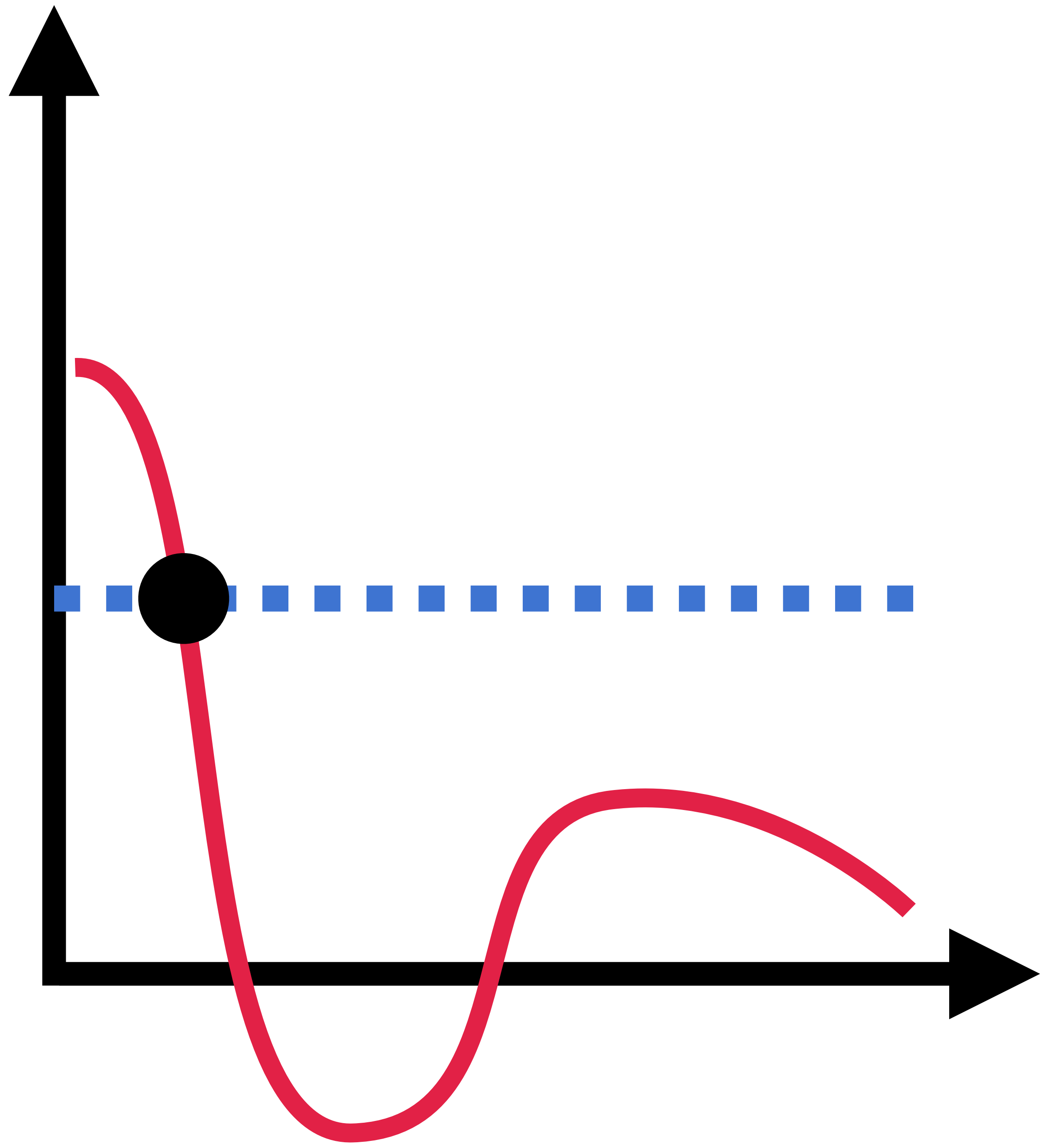


Root finding

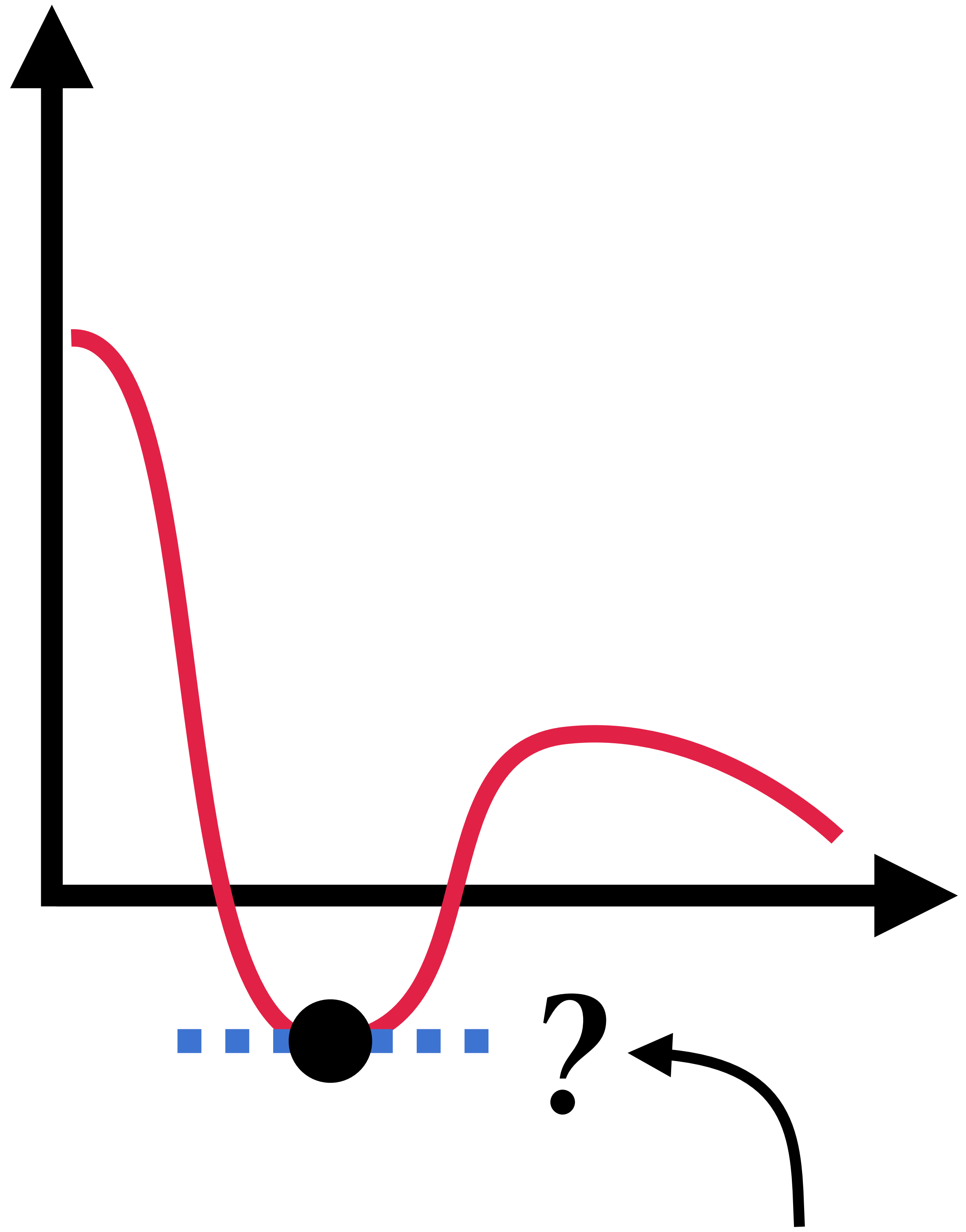


Optimization

Two important kinds of inverse problems



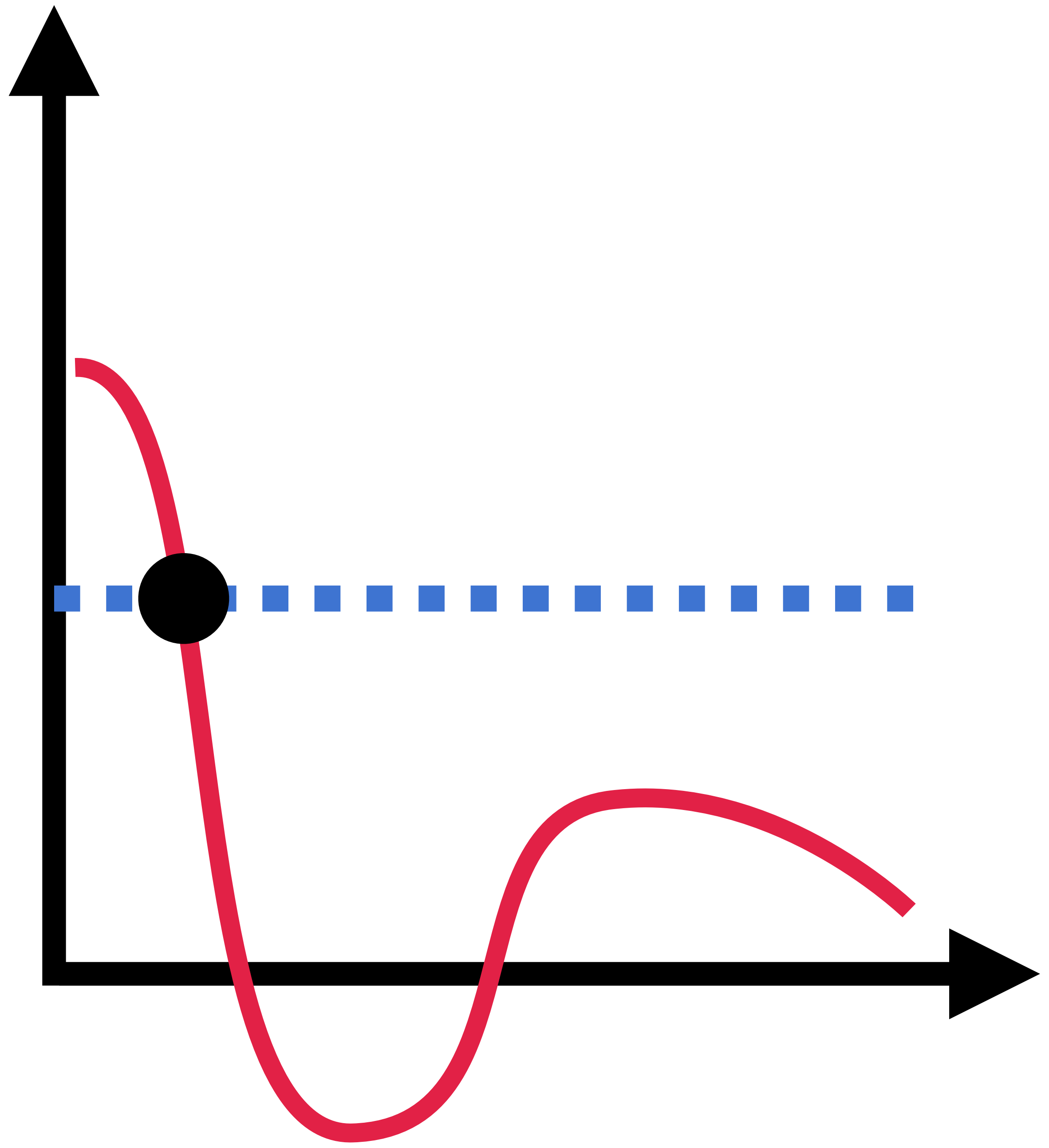
Root finding

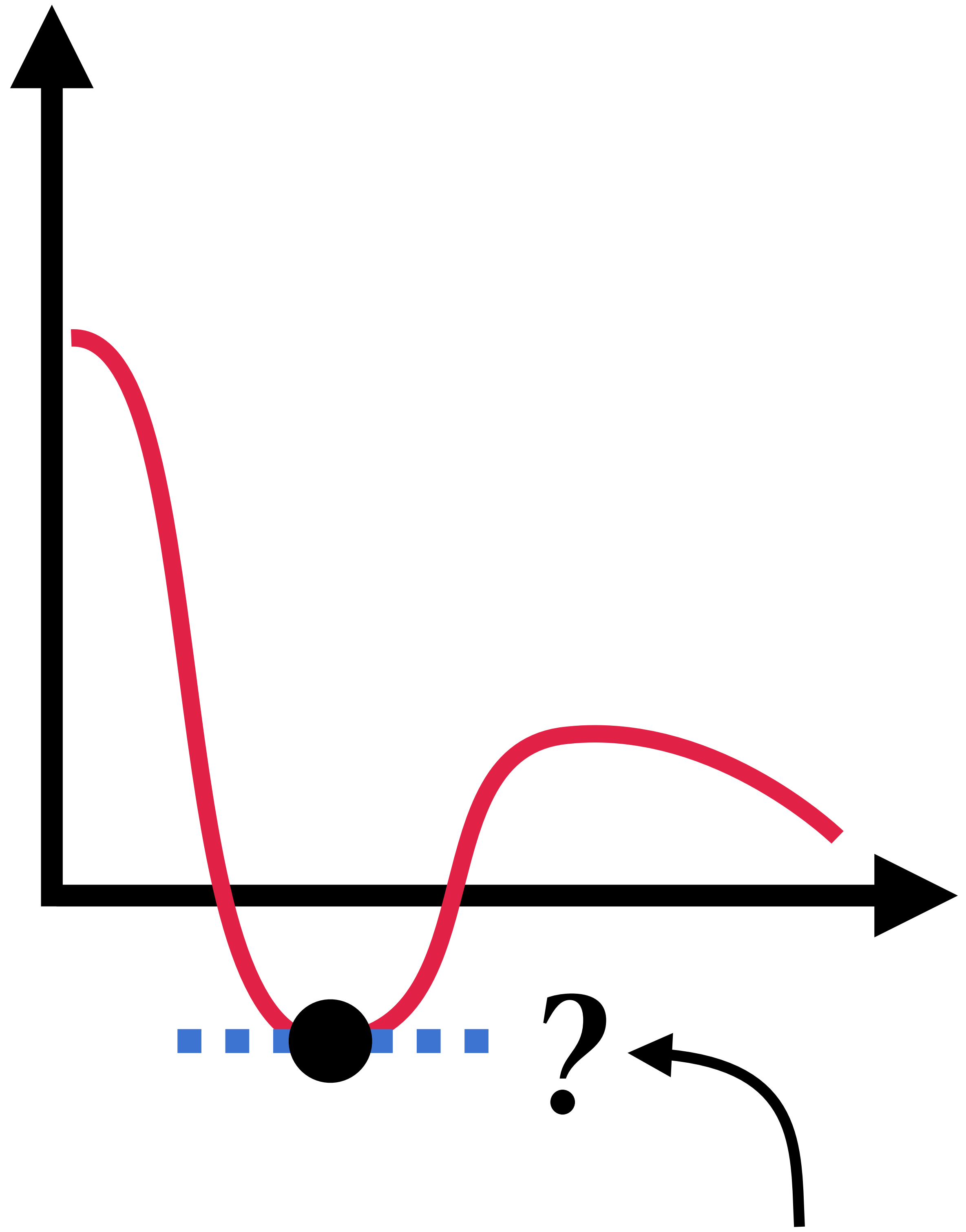


minimum Y value unknown, so this isn't exactly the same as root finding

Optimization

Two important kinds of inverse problems



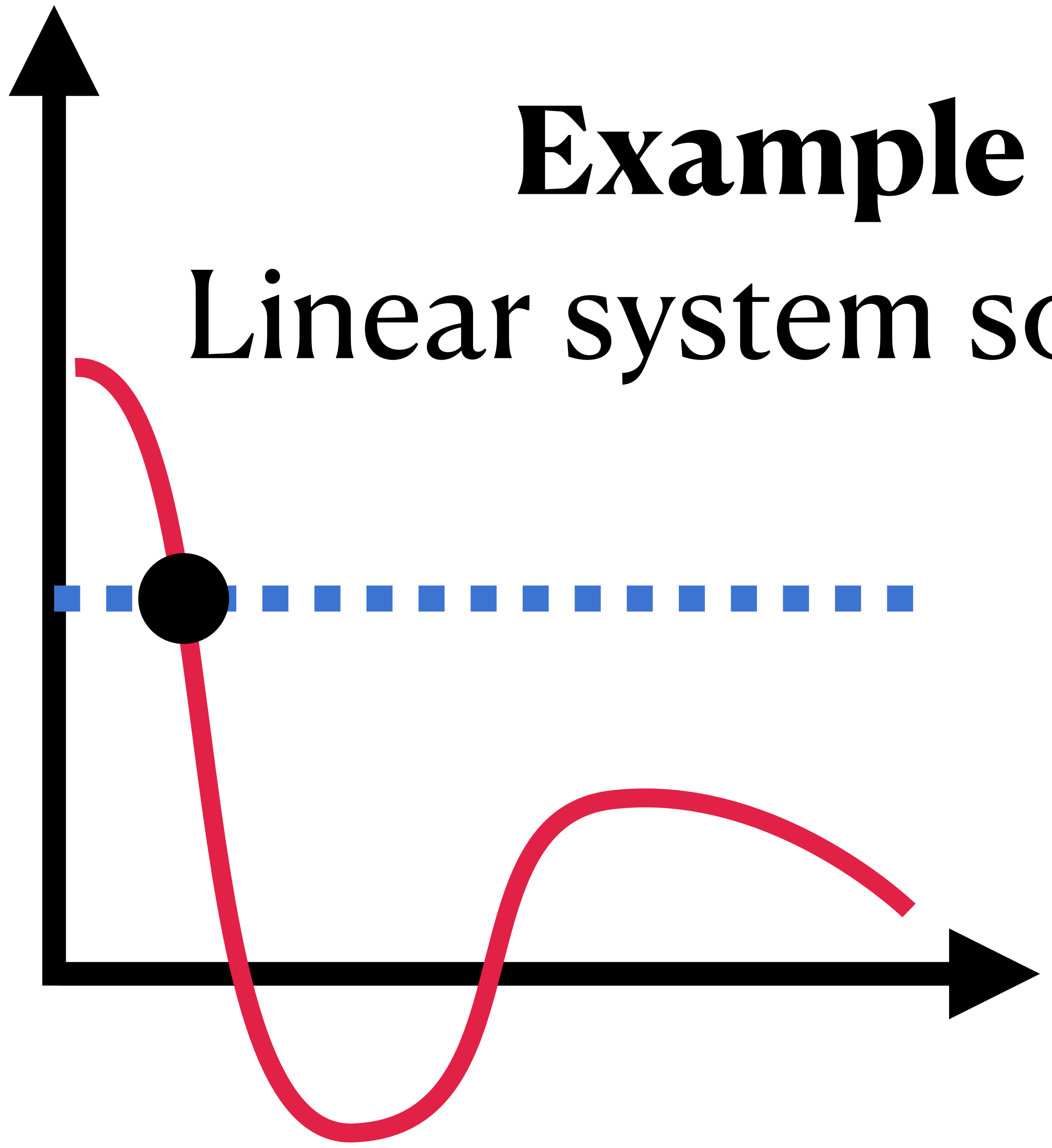


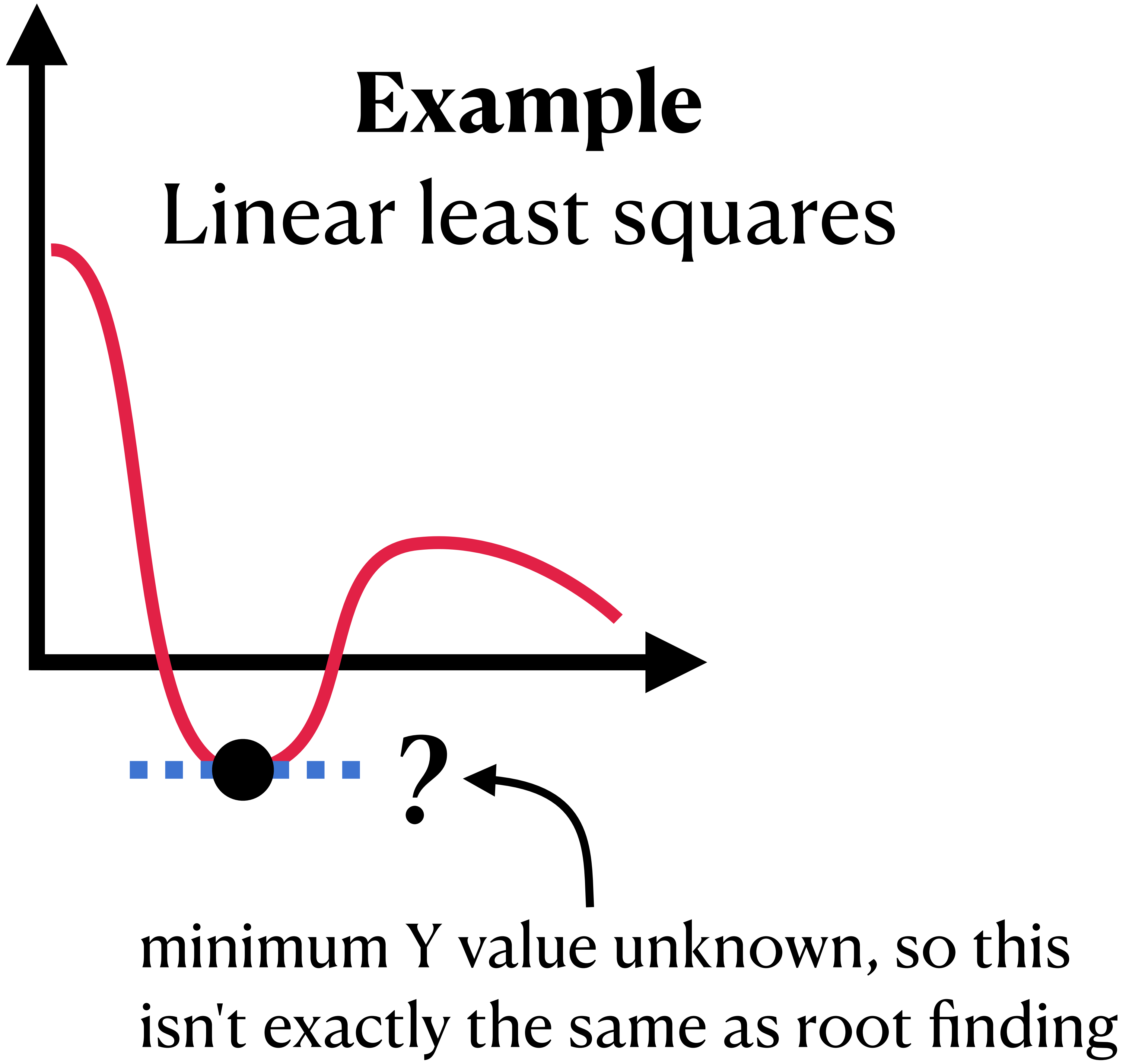
minimum Y value unknown, so this
isn't exactly the same as root finding

Root finding ⟷ Optimization

problems can often be converted

Two important kinds of inverse problems

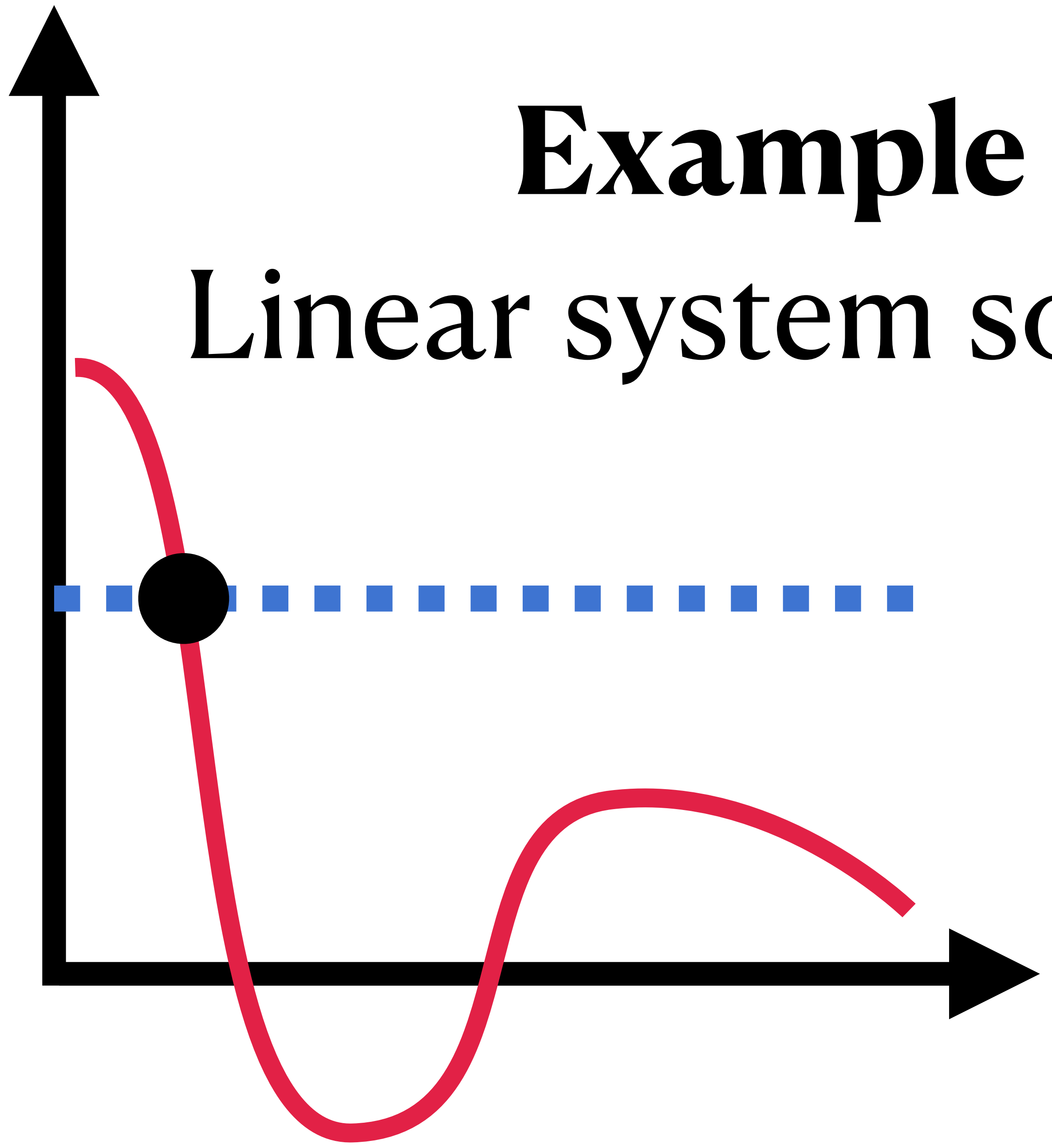




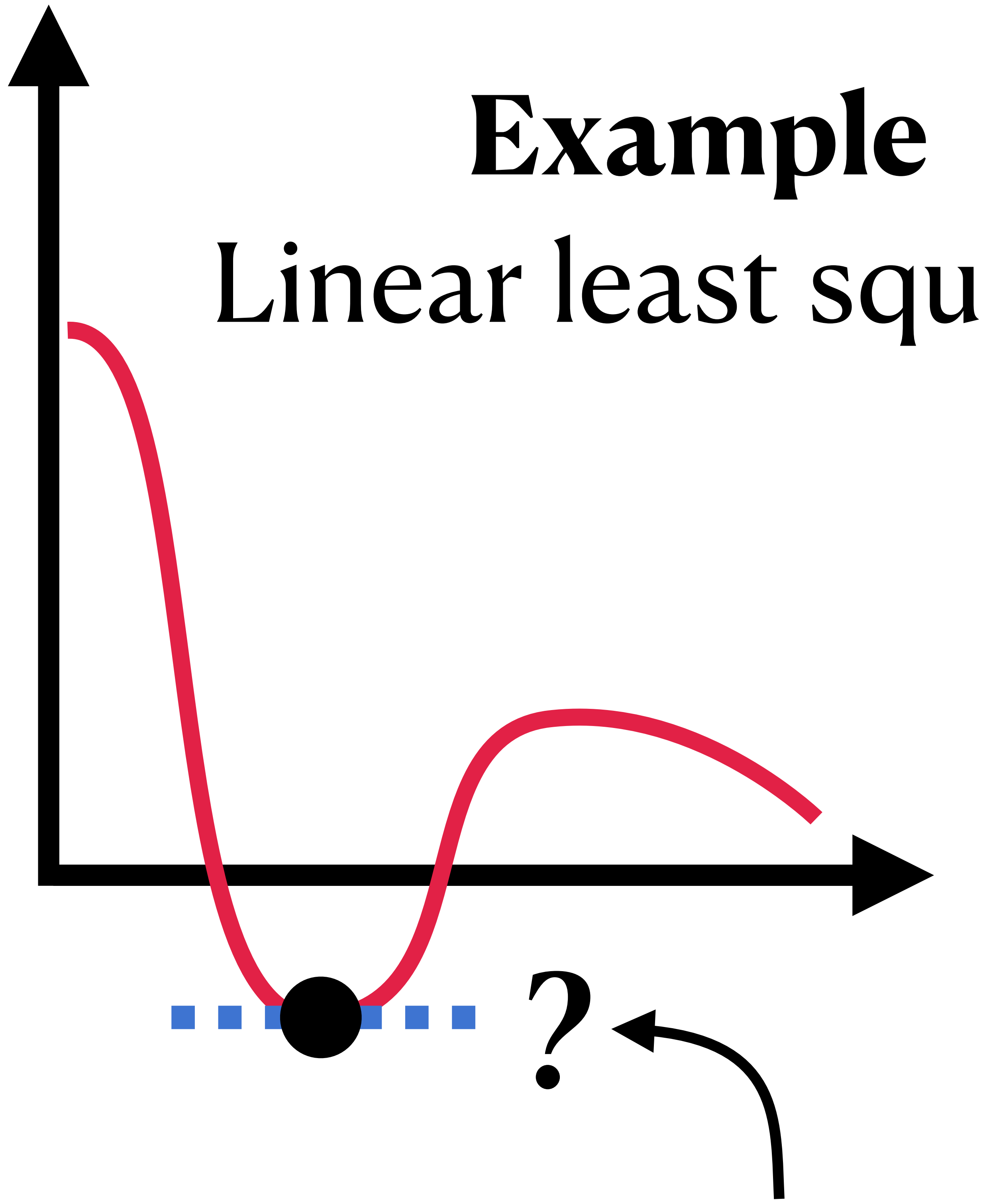
Root finding ⟷ **Optimization**

problems can often be converted

Two important kinds of inverse problems



Example
Linear system solving



Example
Linear least squares

minimum Y value unknown, so this isn't exactly the same as root finding

Conversion:
normal equations

Root finding ⟷ Optimization

problems can often be converted

Root finding

Let's start with a simple 1D problem

$$f : \mathbb{R} \to \mathbb{R}$$

$$\text{Find x so that } f(x) = 0.$$

How many solutions?

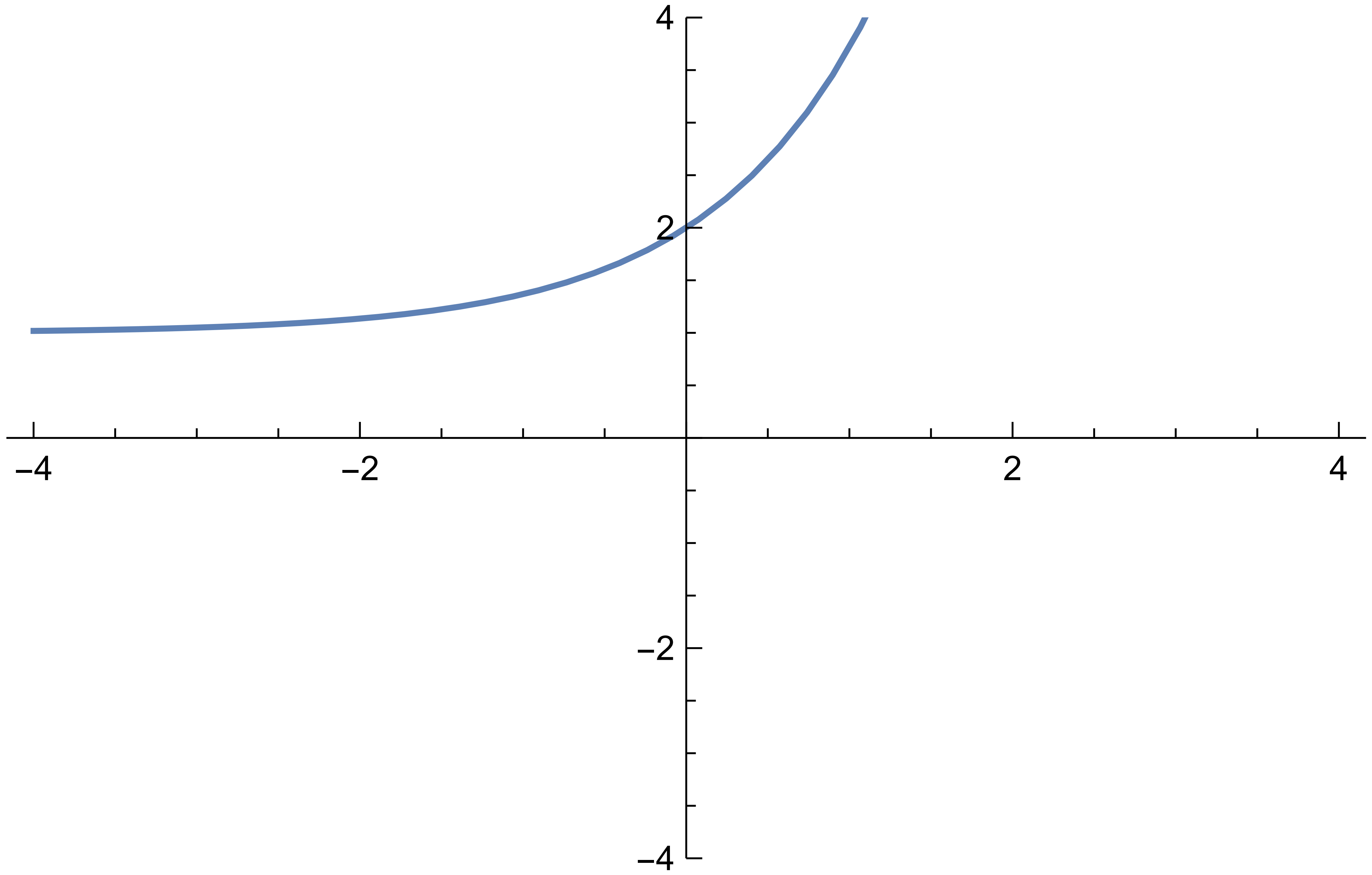
Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

- A few examples from [Heath 2002]:

How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

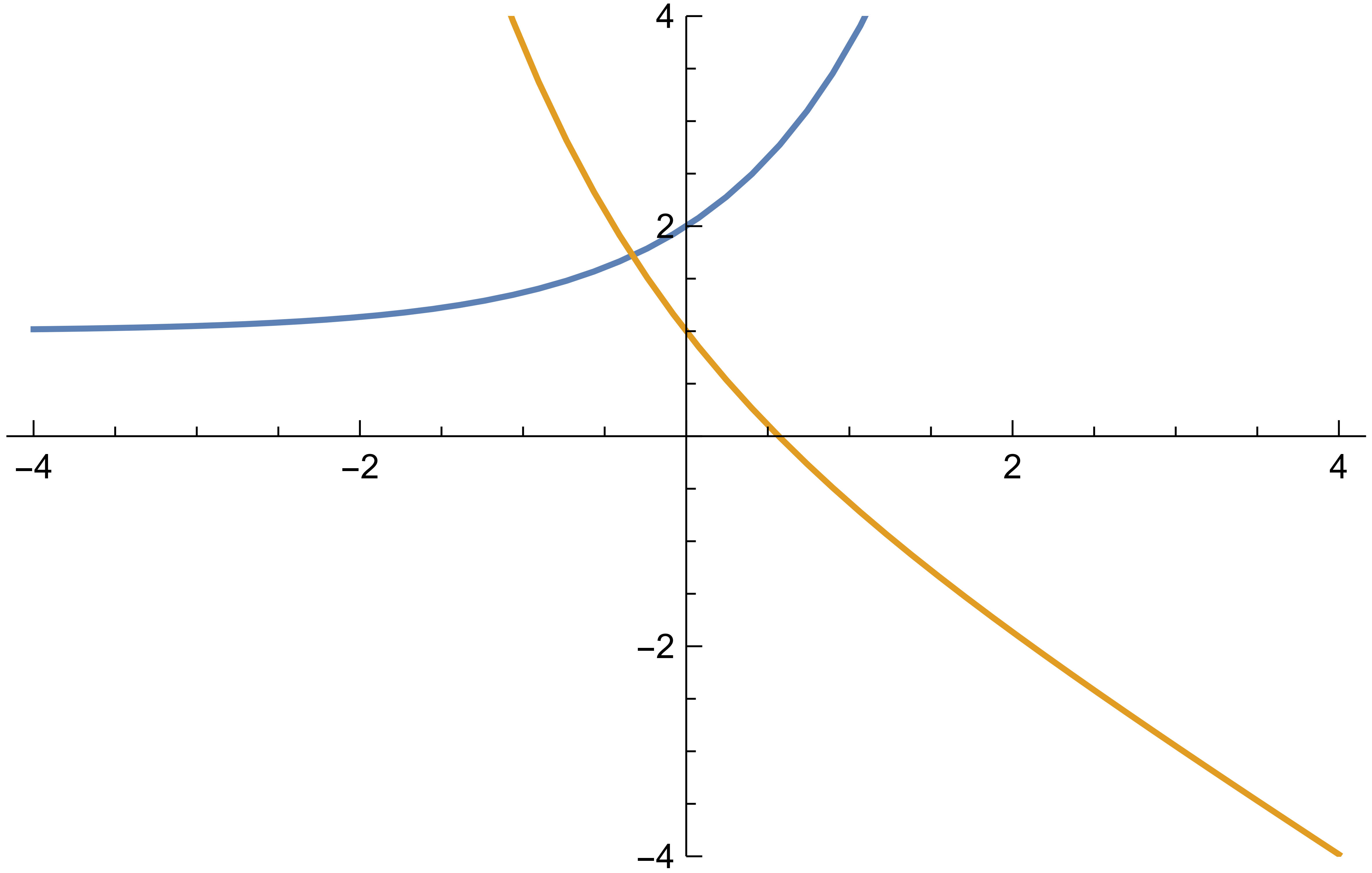
- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$



How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

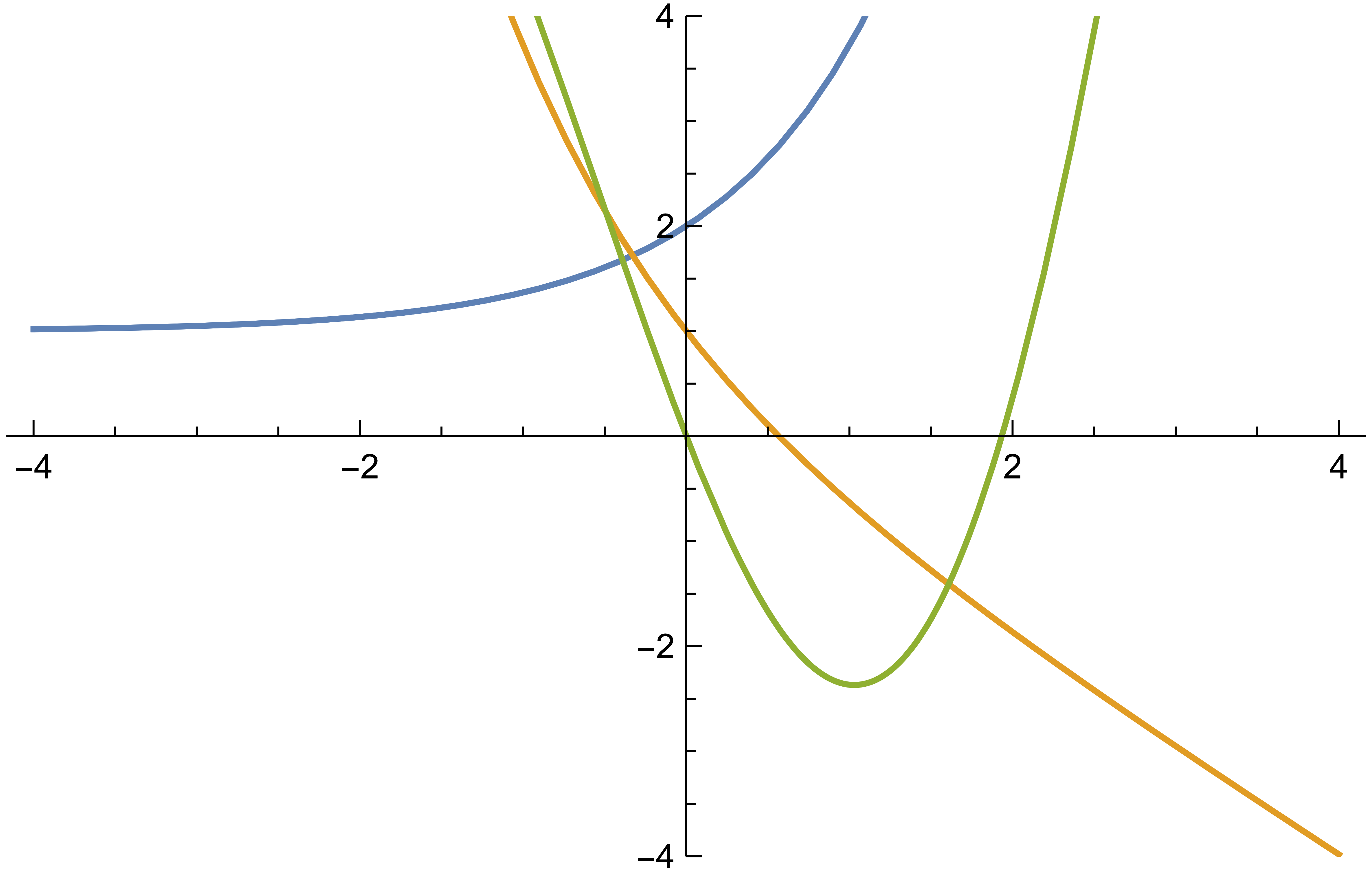
- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$
  - $\exp(-x) - x = 0$



How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

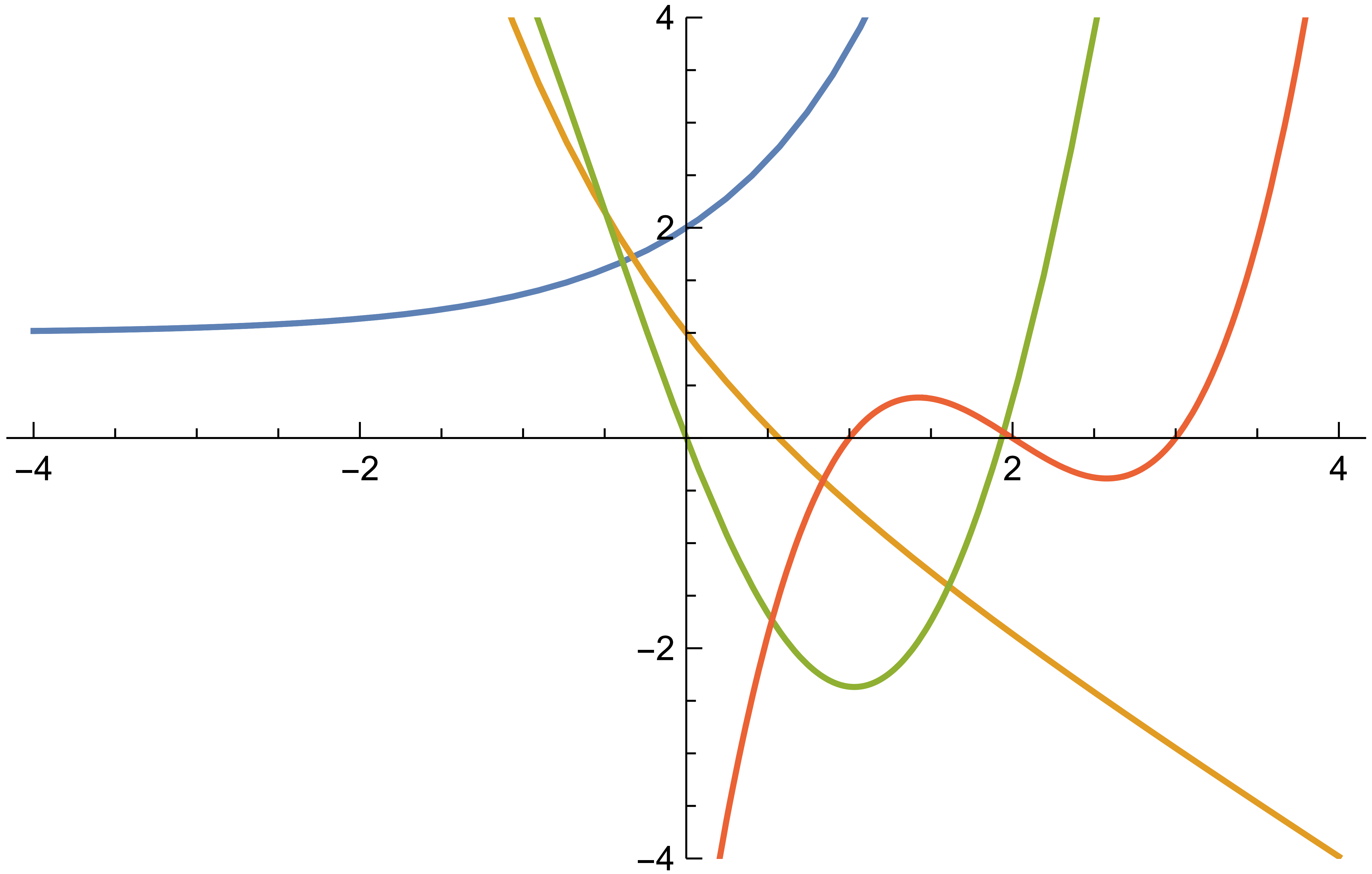
- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$
  - $\exp(-x) - x = 0$
  - $x^2 - 4\sin(x) = 0$



How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

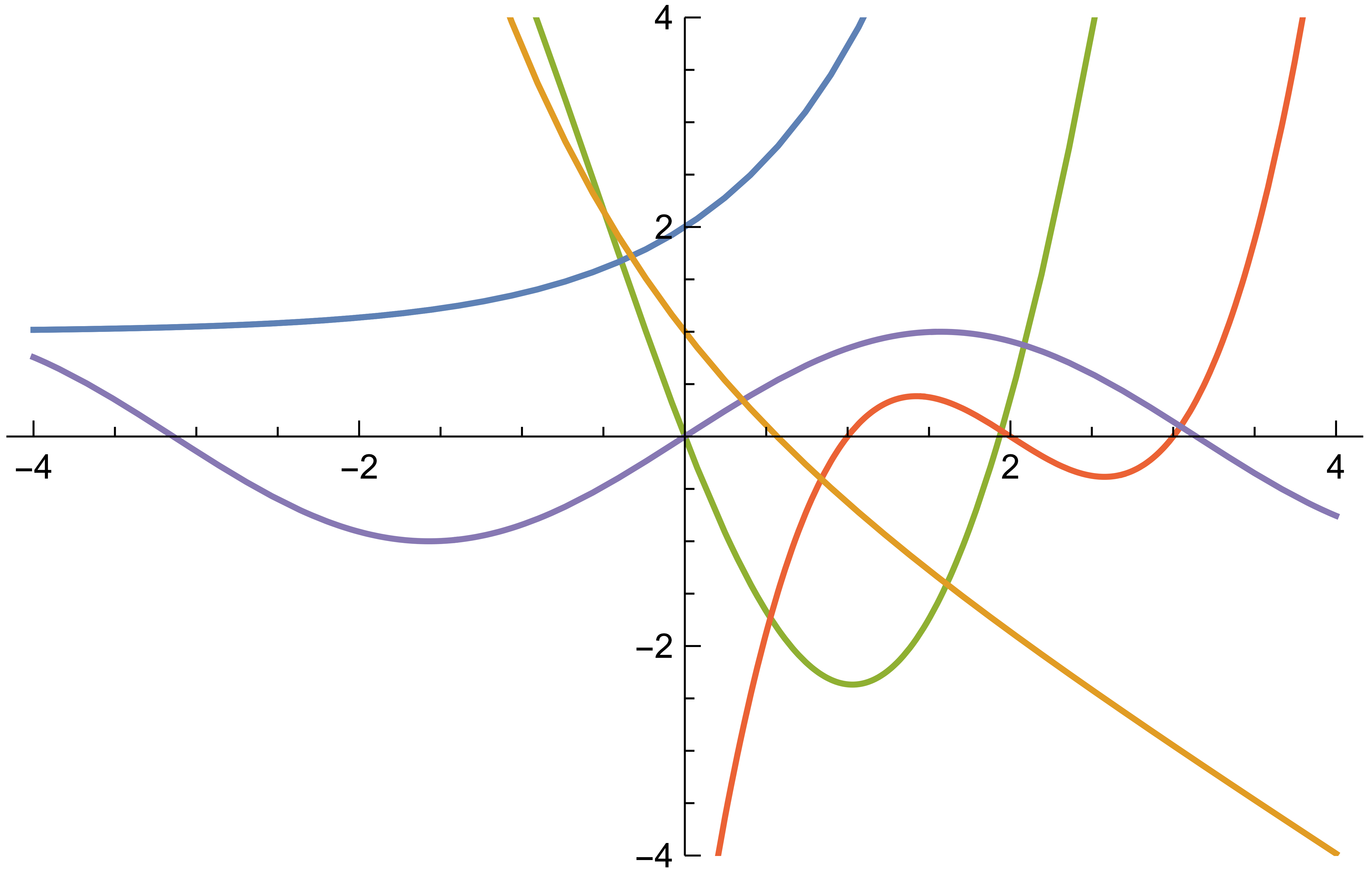
- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$
  - $\exp(-x) - x = 0$
  - $x^2 - 4\sin(x) = 0$
  - $x^3 - 6x^2 + 11x - 6 = 0$



How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$
  - $\exp(-x) - x = 0$
  - $x^2 - 4\sin(x) = 0$
  - $x^3 - 6x^2 + 11x - 6 = 0$
  - $\sin(x) = 0$



"Evil functions"

All sorts of things could be lurking inside. How are we expected to deal with such functions?

"Evil functions"

All sorts of things could be lurking inside. How are we expected to deal with such functions?

$$f_1(x) := \begin{cases} 1, & x \in \mathbb{Q} \\ -1, & x \in \mathbb{R} \setminus \mathbb{Q} \end{cases}$$

"Evil functions"

All sorts of things could be lurking inside. How are we expected to deal with such functions?

$$f_1(x) := \begin{cases} 1, & x \in \mathbb{Q} \\ -1, & x \in \mathbb{R} \setminus \mathbb{Q} \end{cases}$$

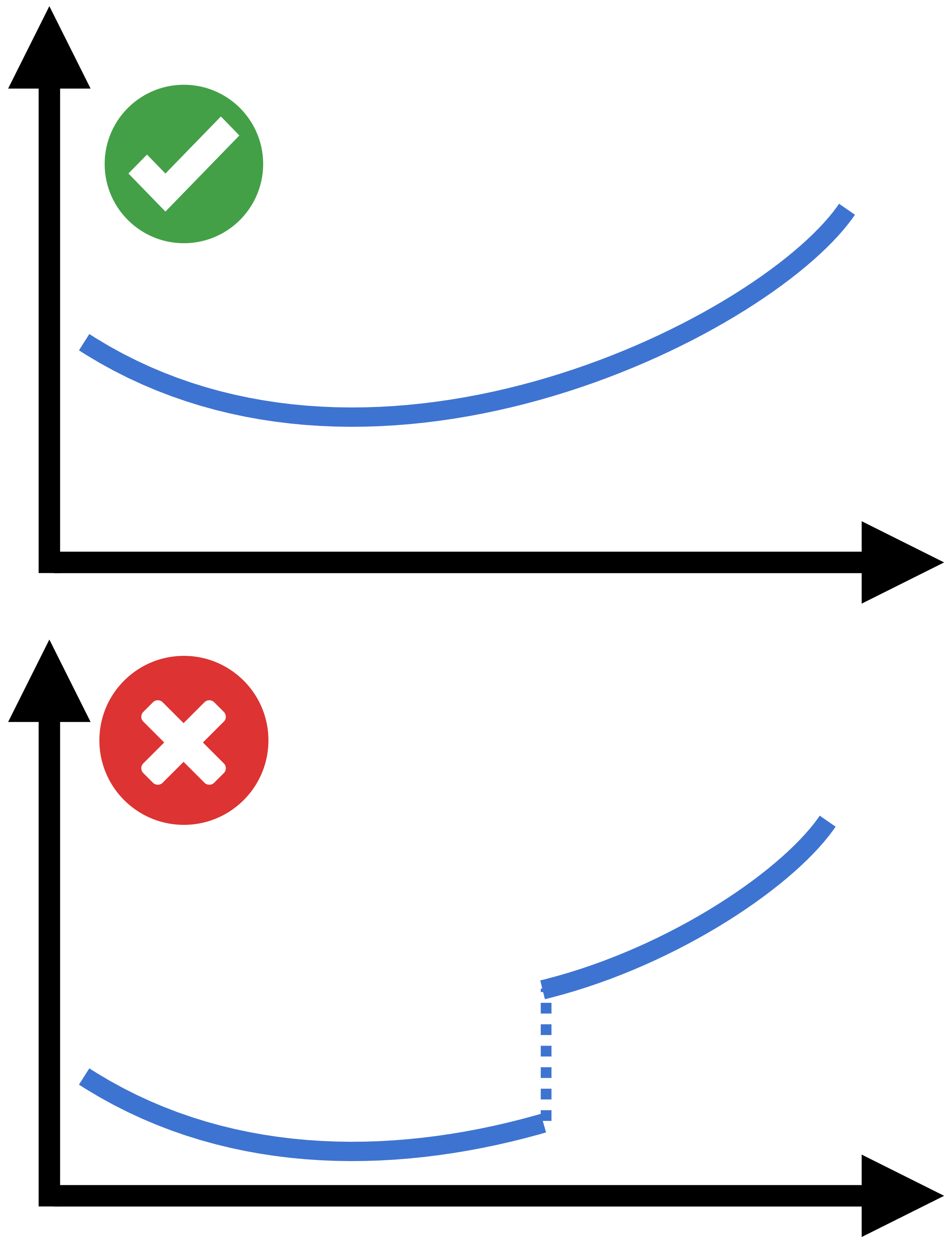
$$f_2(x) := \begin{cases} 1, & x \neq 0.13525634 \\ 0, & x = 0.13525634 \end{cases}$$

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

Common assumptions

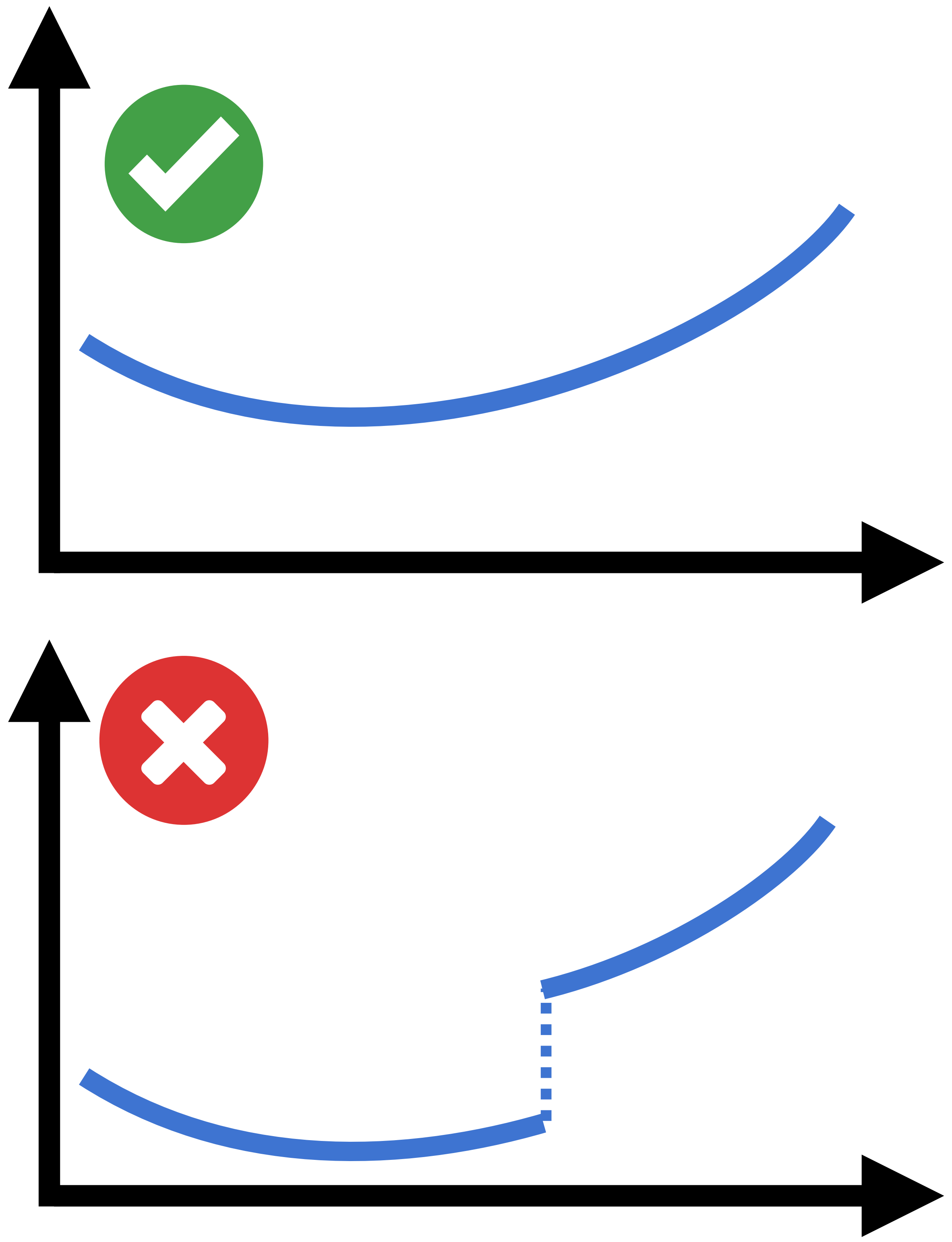
Must assume *something*. This determines how the solution algorithm will work.



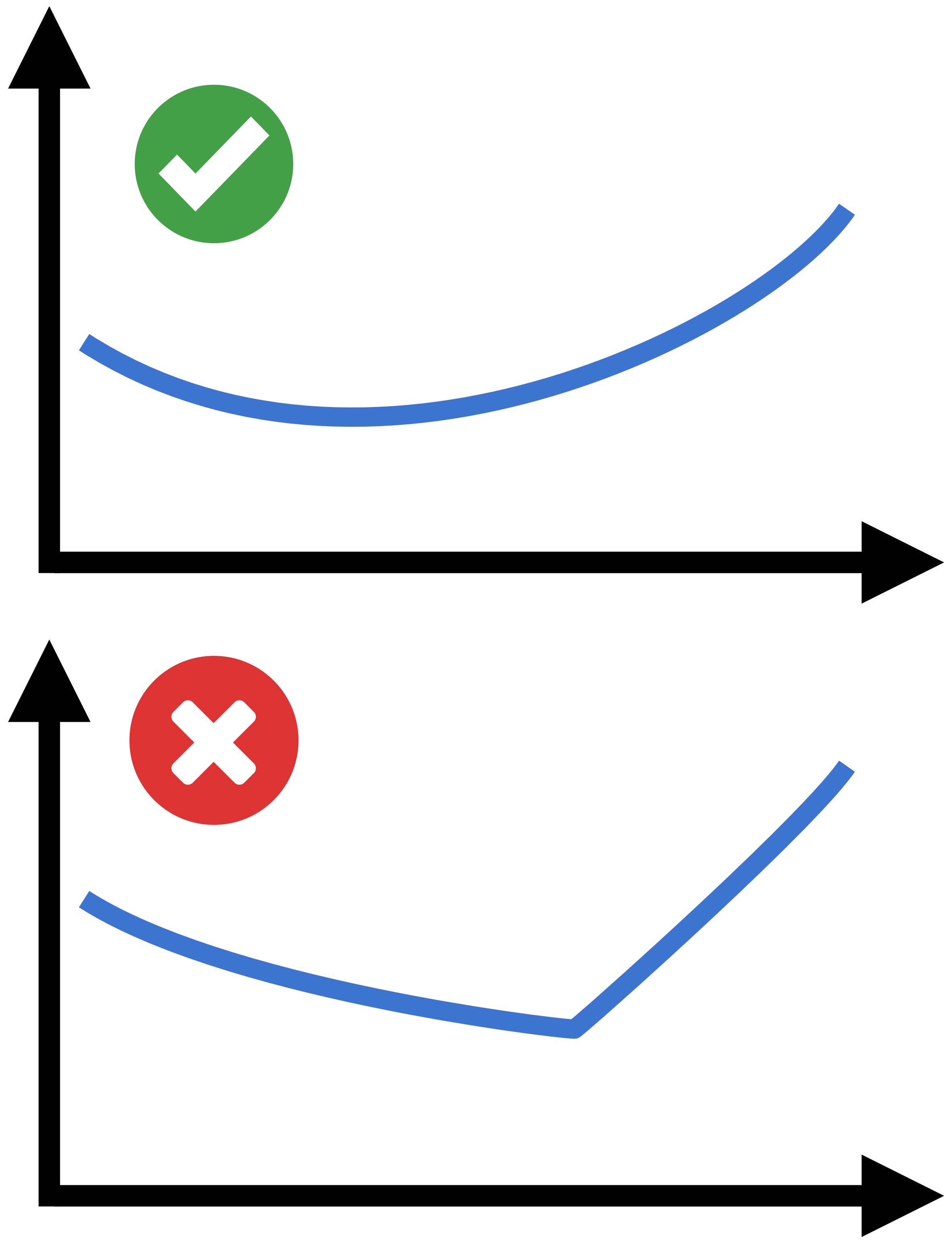
Continuity

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.



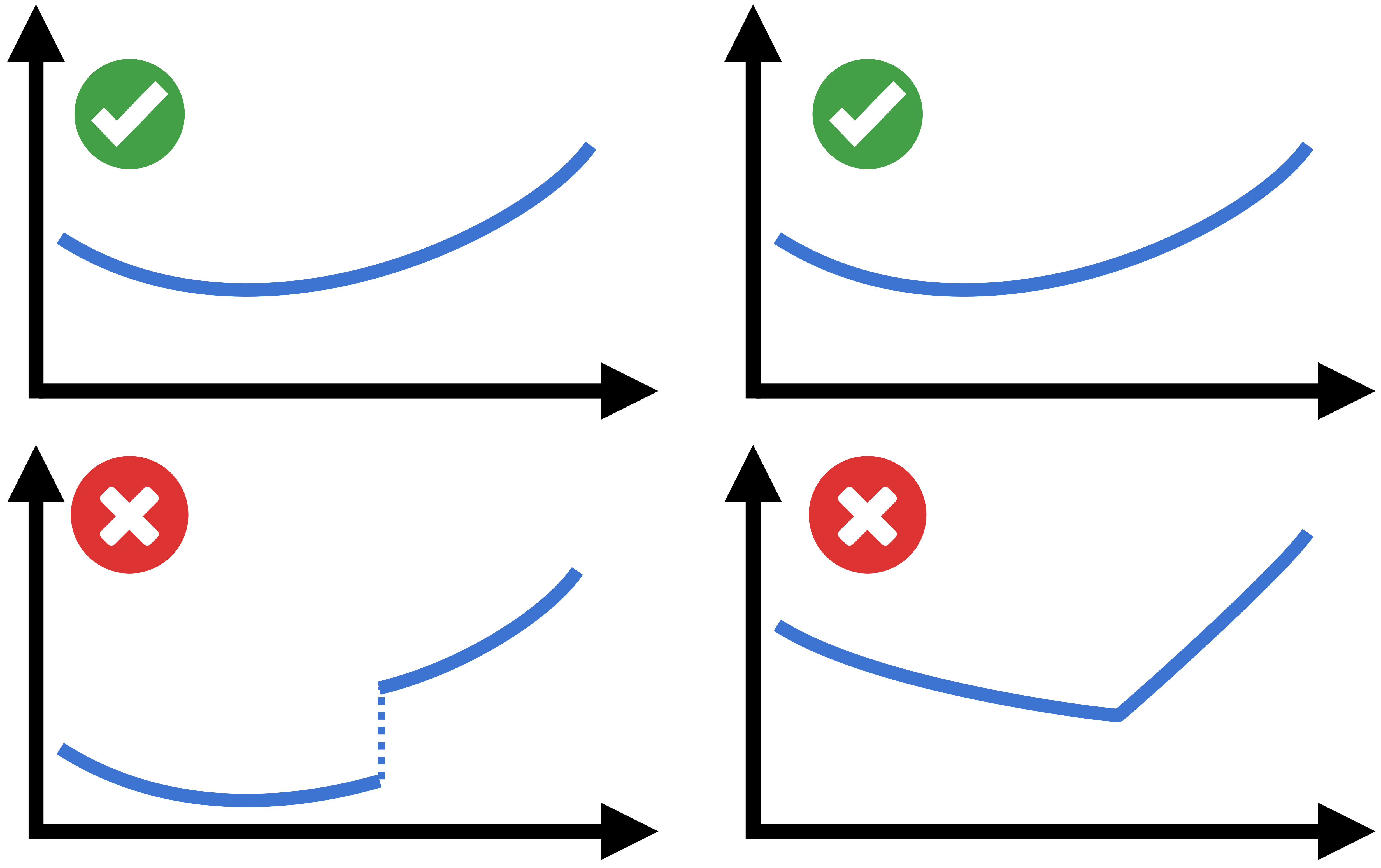
Continuity



Differentiability

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

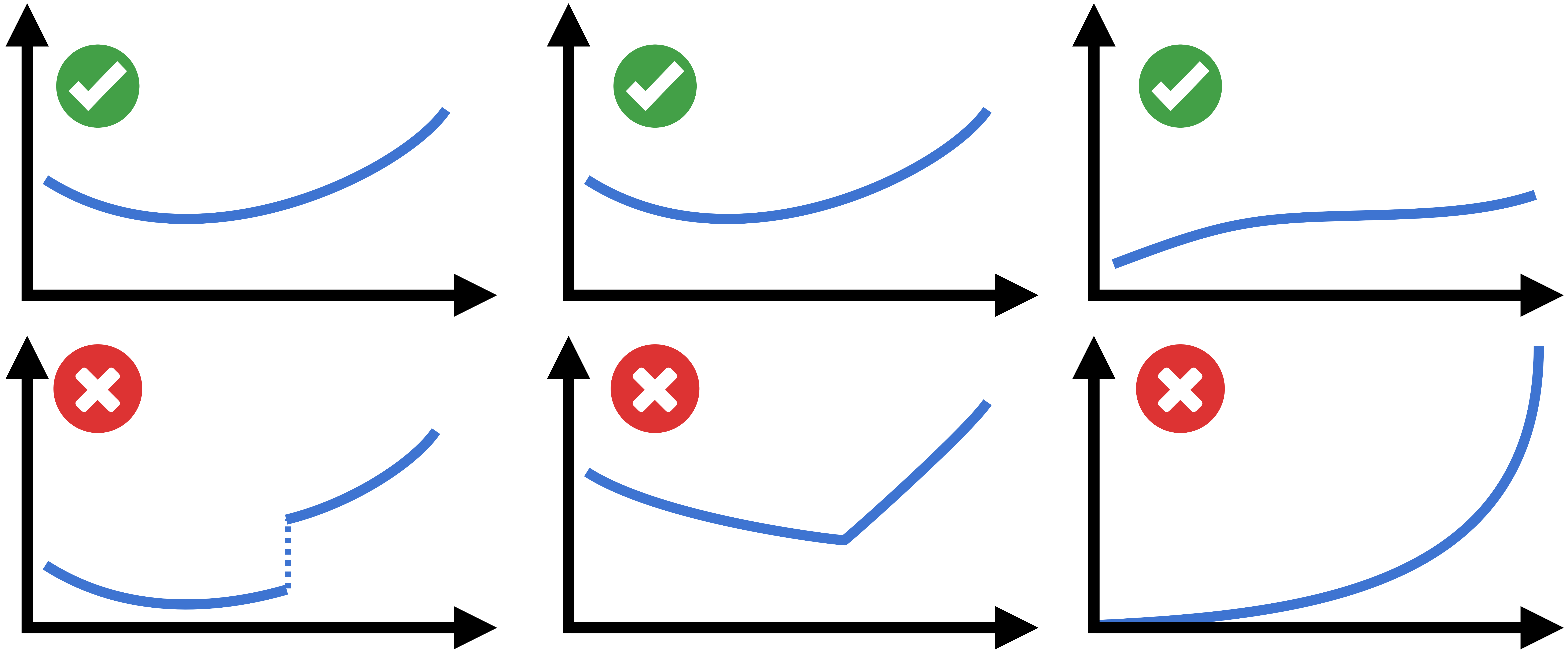


Continuity **Differentiability**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

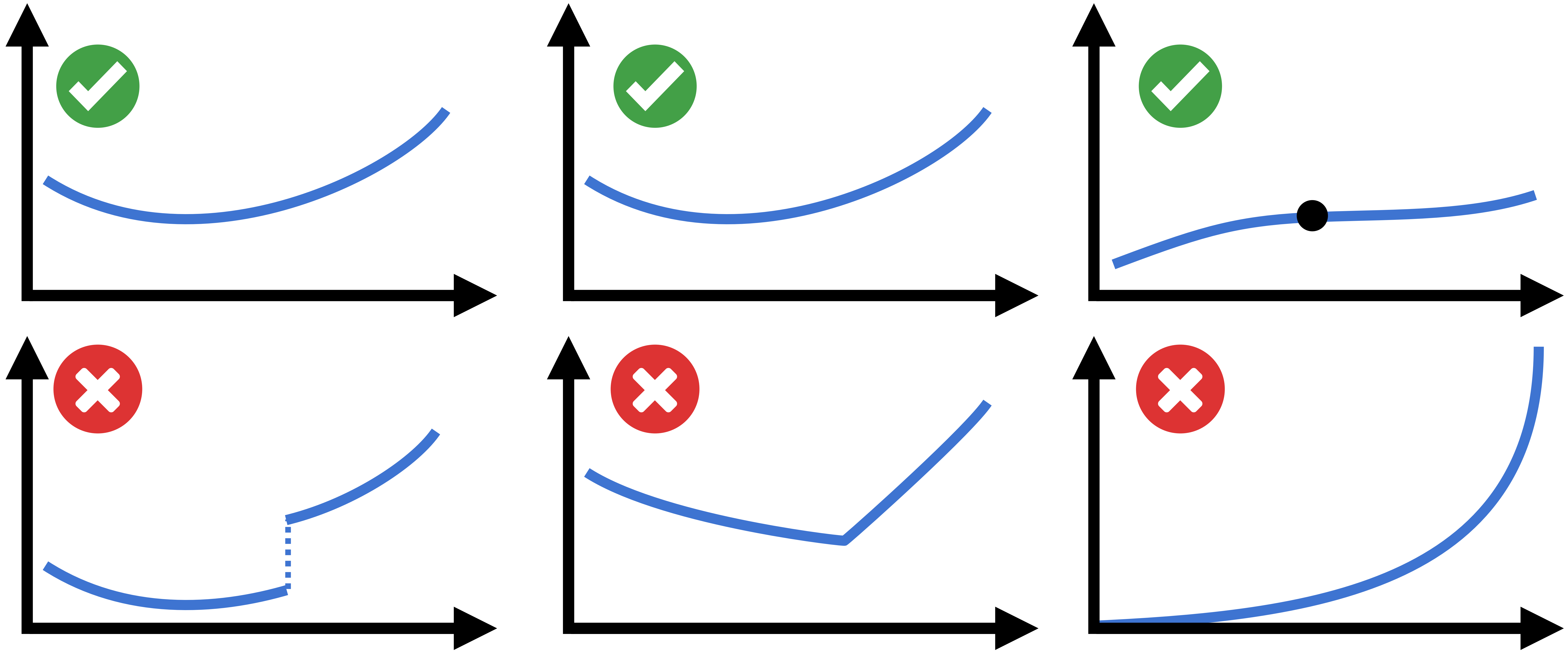


Continuity | **Differentiability** | ***Lipschitz* continuity**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

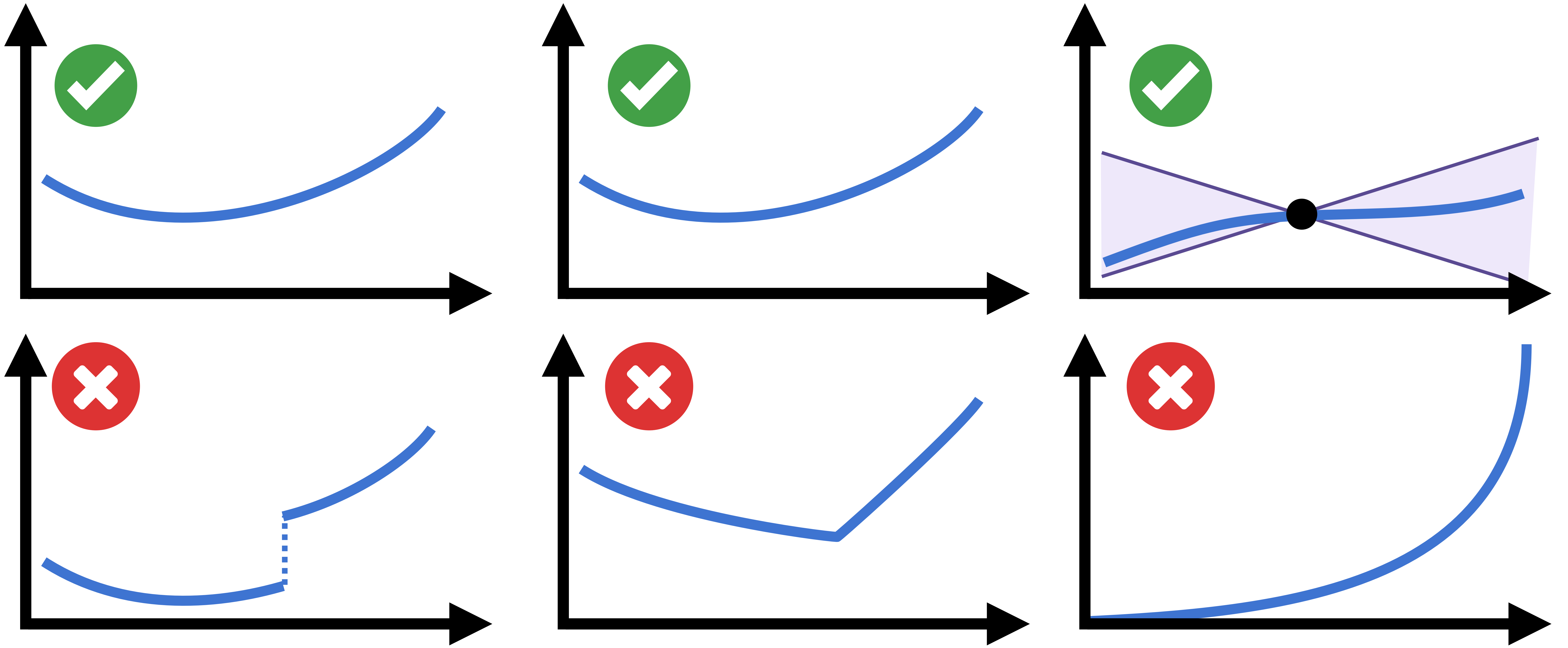


Continuity **Differentiability** ***Lipschitz* continuity**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

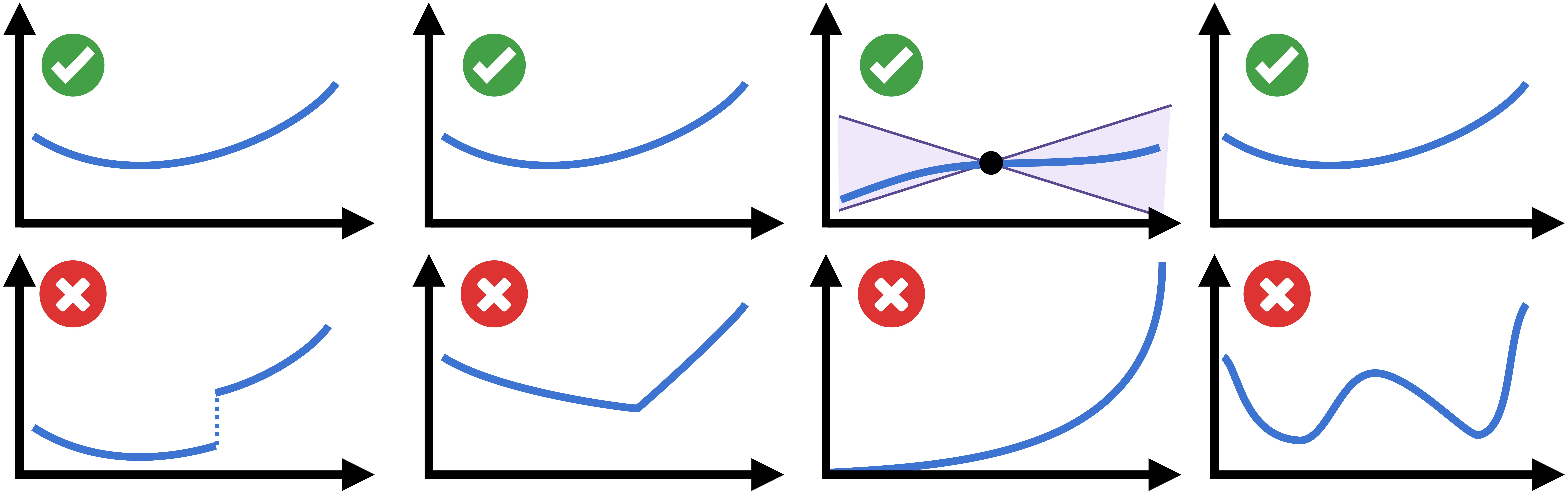


Continuity **Differentiability** ***Lipschitz* continuity**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

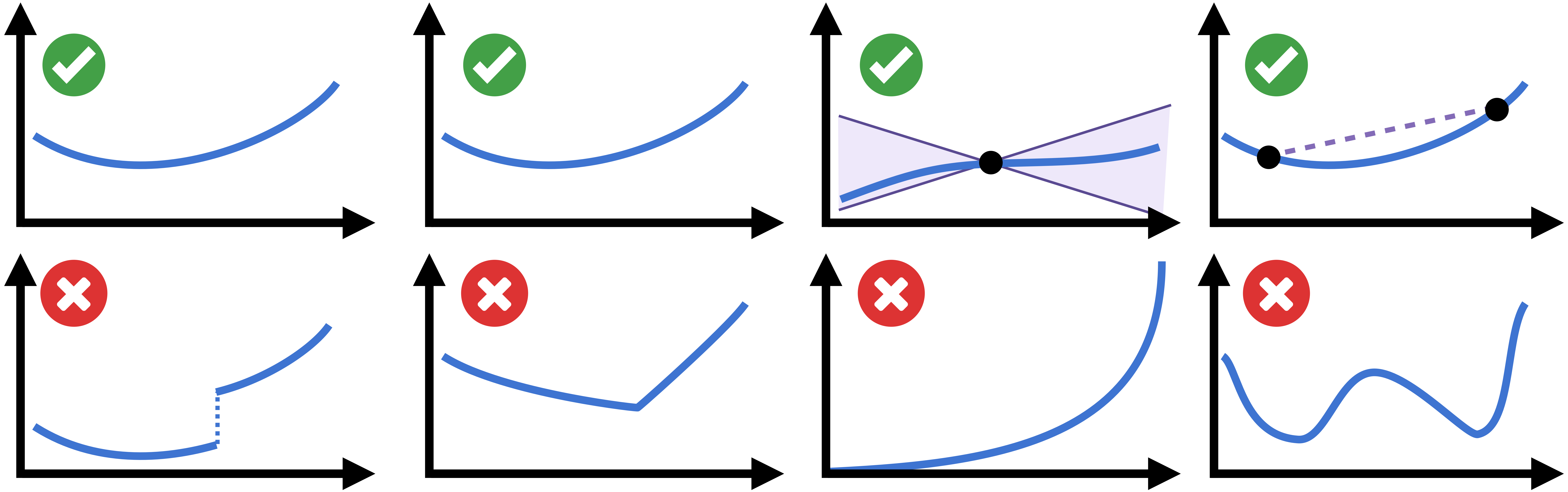


Continuity | **Differentiability** | ***Lipschitz* continuity** | **Convexity**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

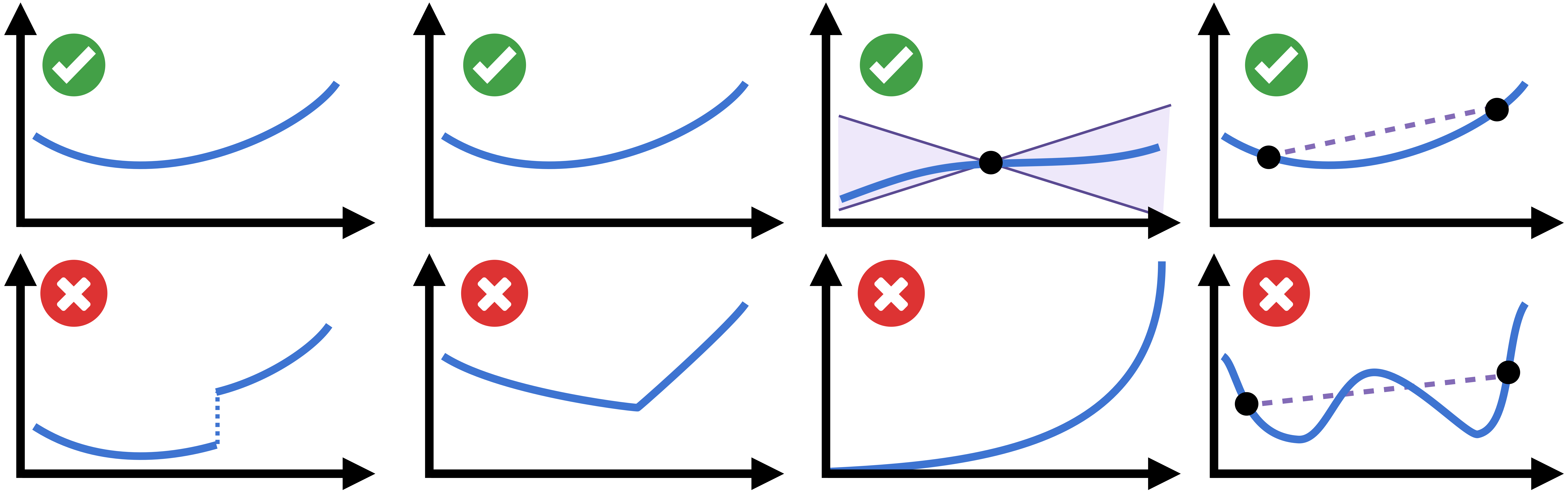


Continuity | Differentiability | *Lipschitz* continuity | Convexity

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

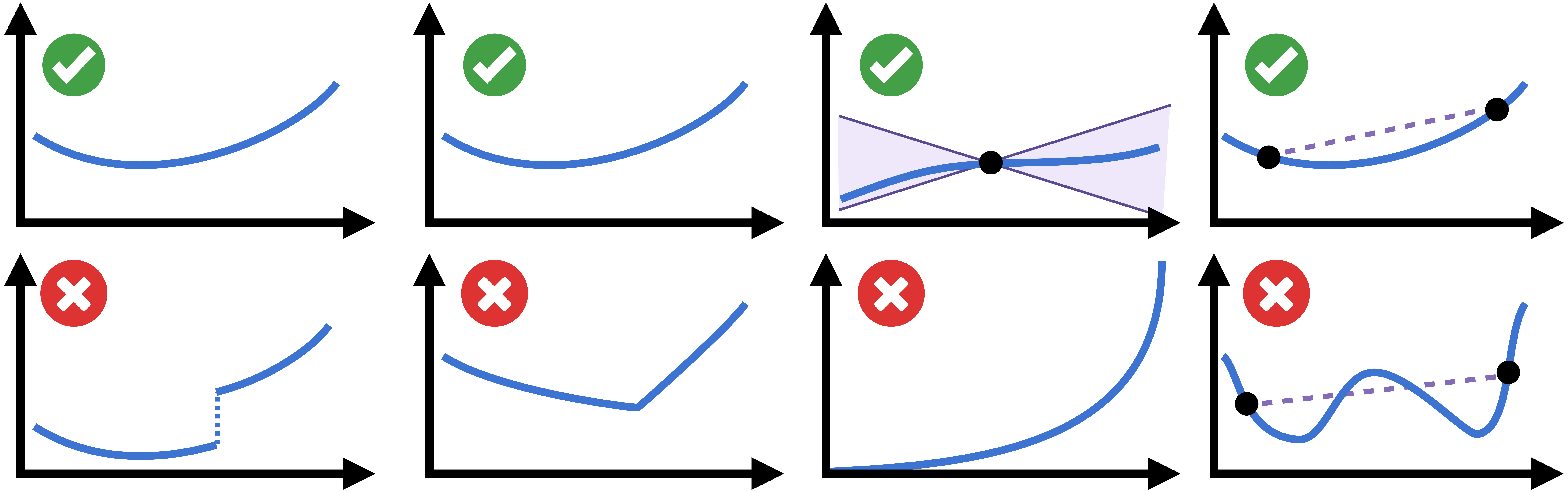


Continuity | Differentiability | *Lipschitz* continuity | Convexity

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.



Continuity | **Differentiability** | ***Lipschitz* continuity** | **Convexity**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

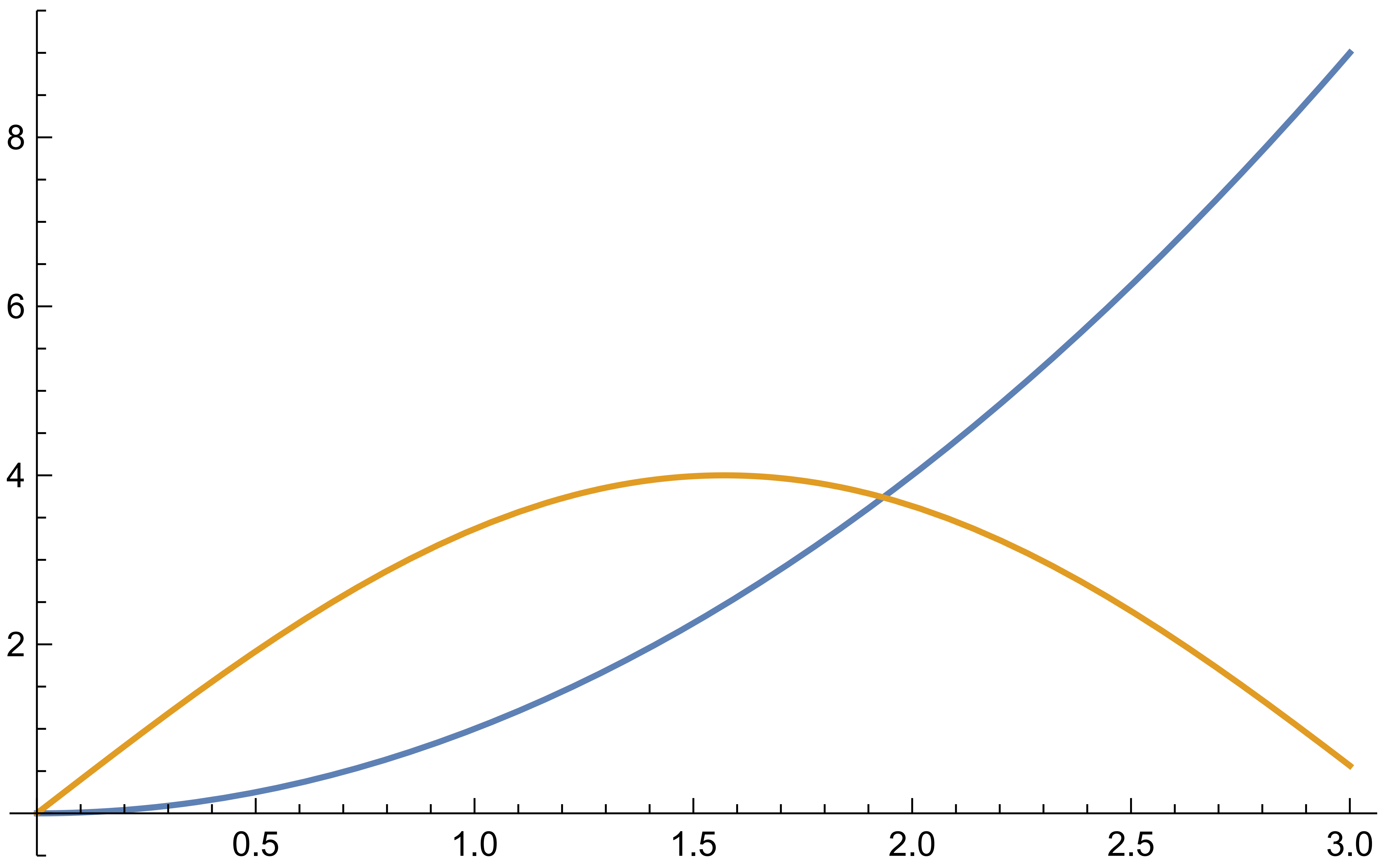
Mostly used for optimization.

A model problem

$$x^2 = 4 \sin x$$

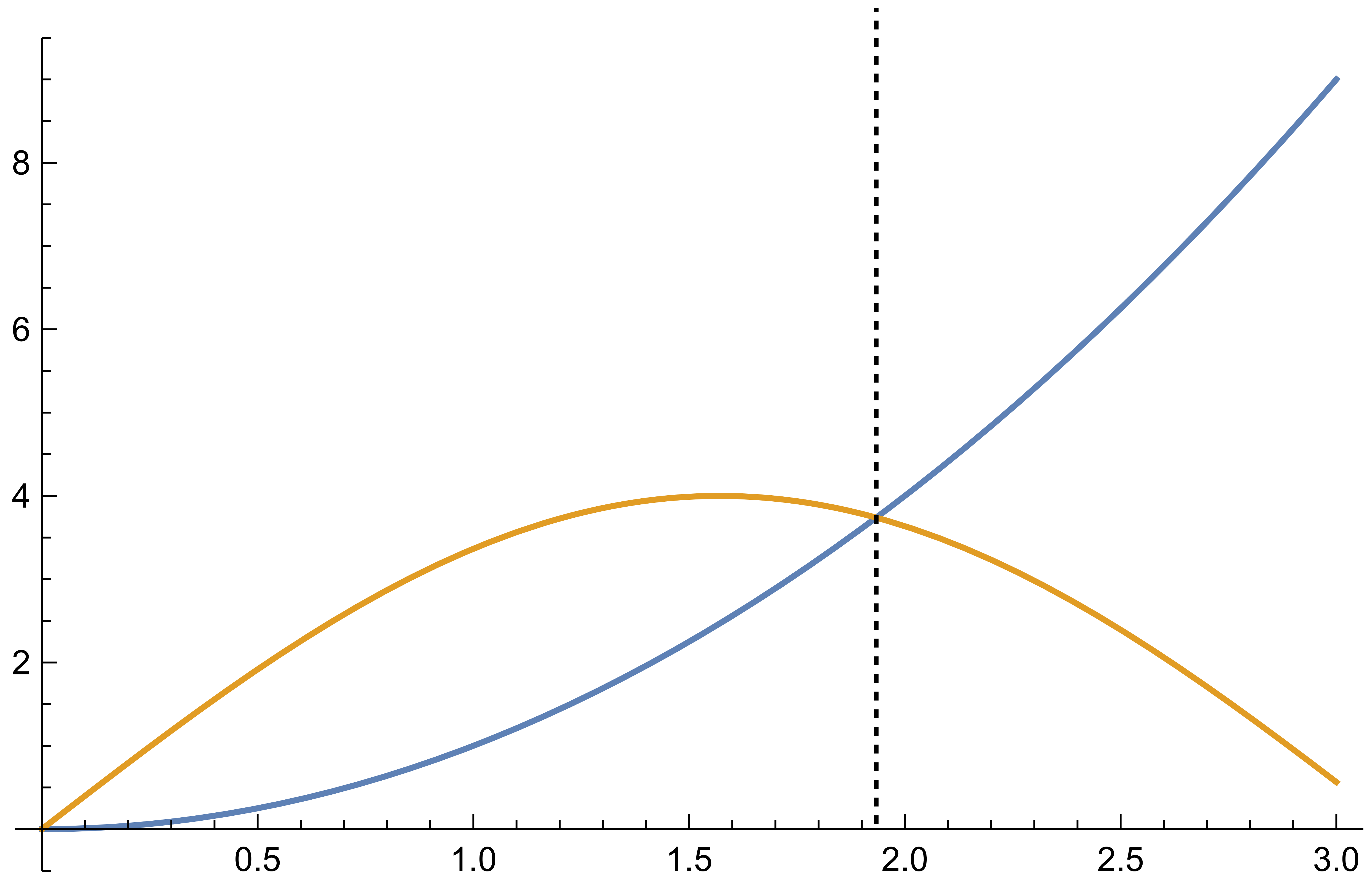
A model problem

$$x^2 = 4 \sin x$$



A model problem

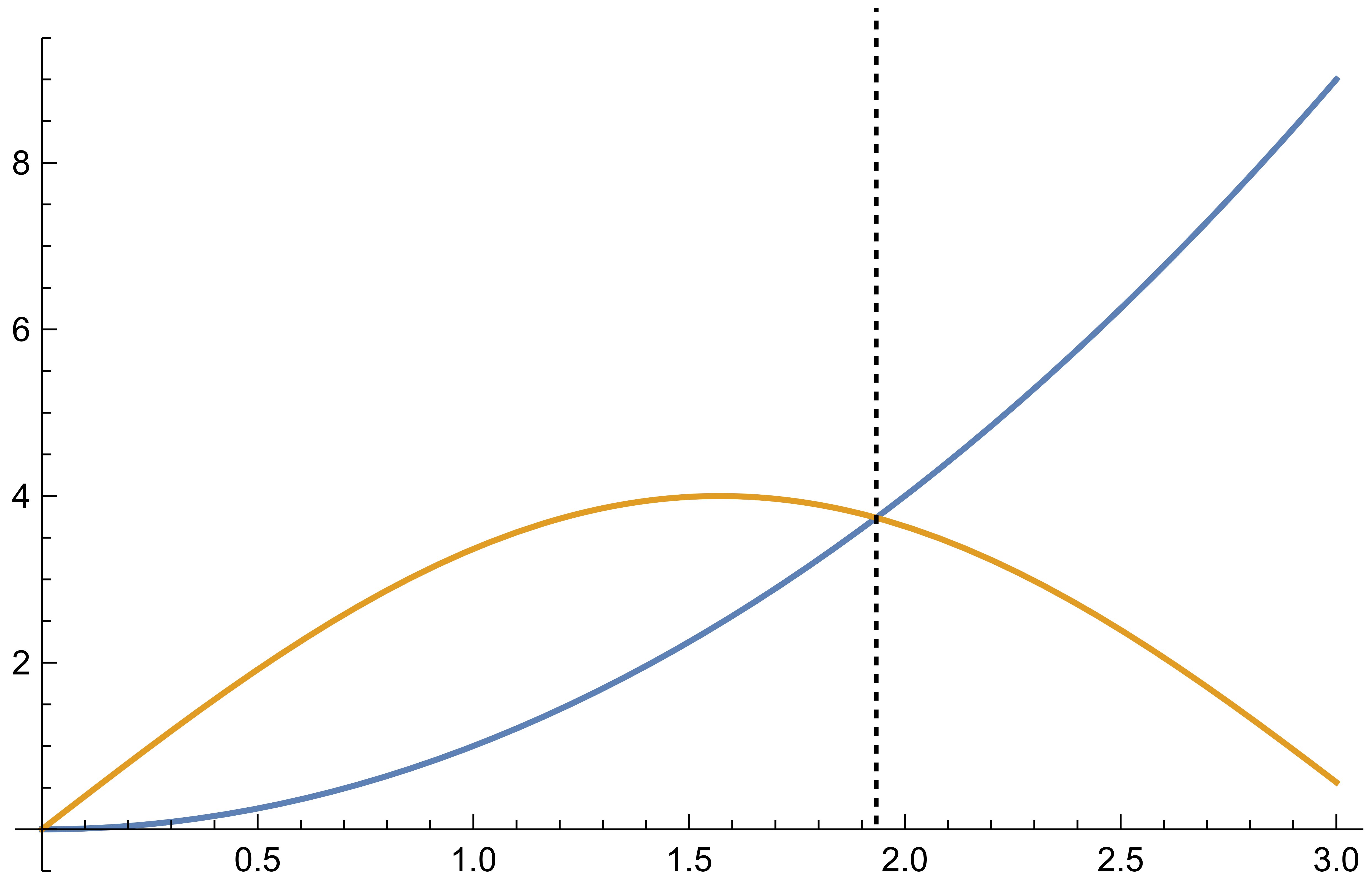
$$x^2 = 4 \sin x$$



A model problem

$$x^2 = 4 \sin x$$

Solution is not analytic!



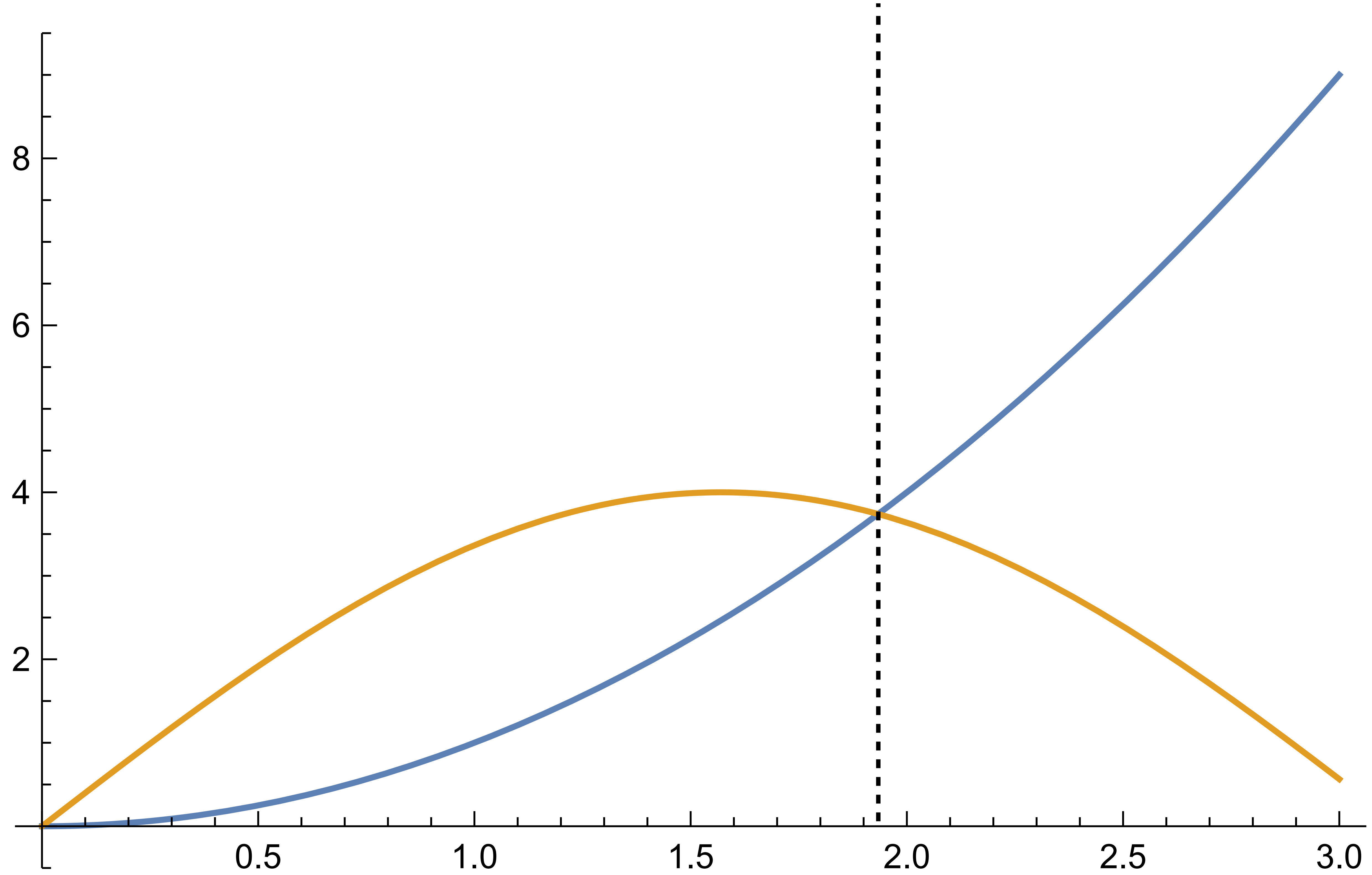
A model problem

Mathematica: In[1]:= `Solve[x^2 == 4 Sin[x], x]`

Solve: This system cannot be solved with the methods available to Solve.

$$x^2 = 4\sin x$$

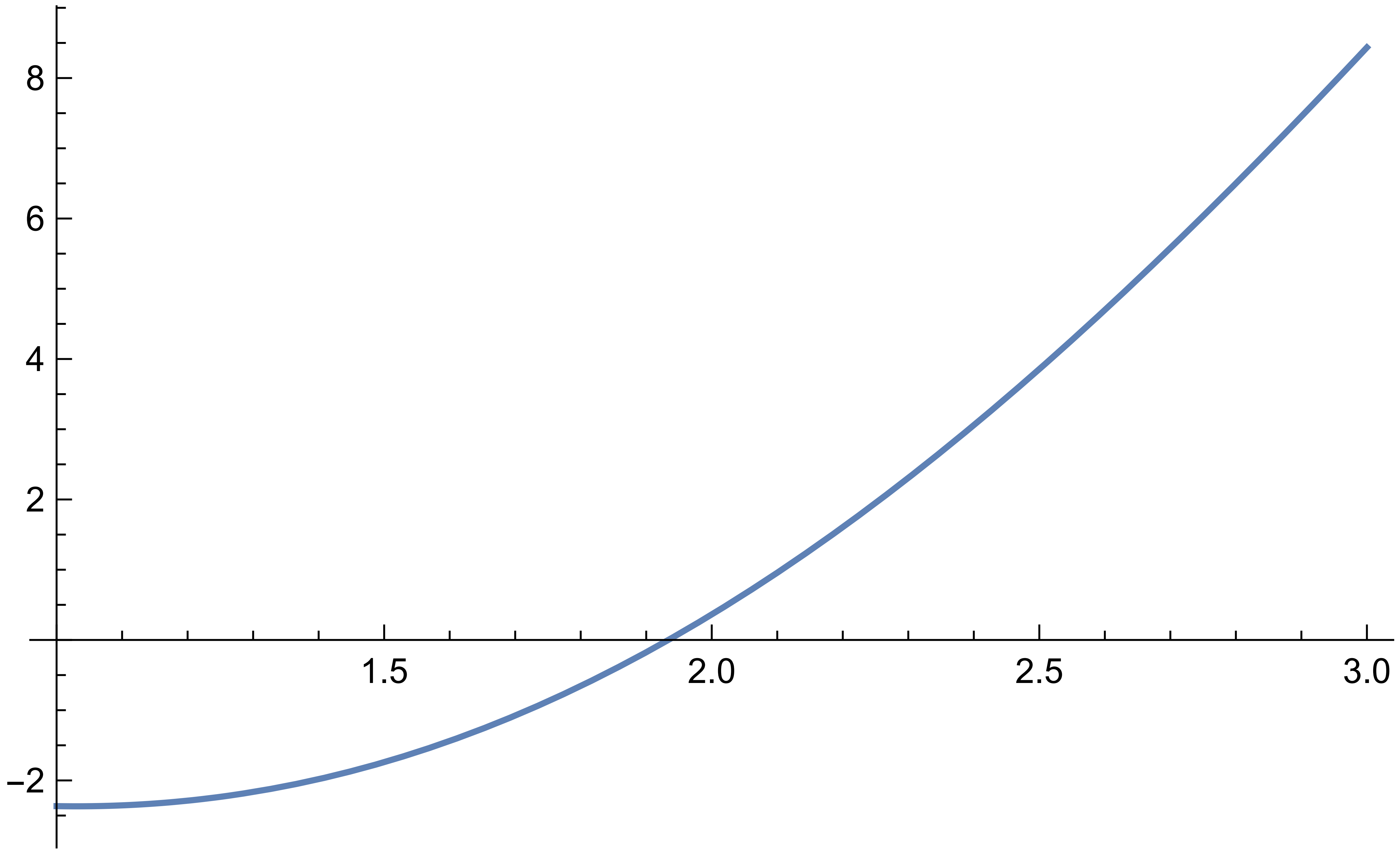
Solution is not analytic!



A model problem

Can more or less read off solution from graph, *how difficult can it be?*

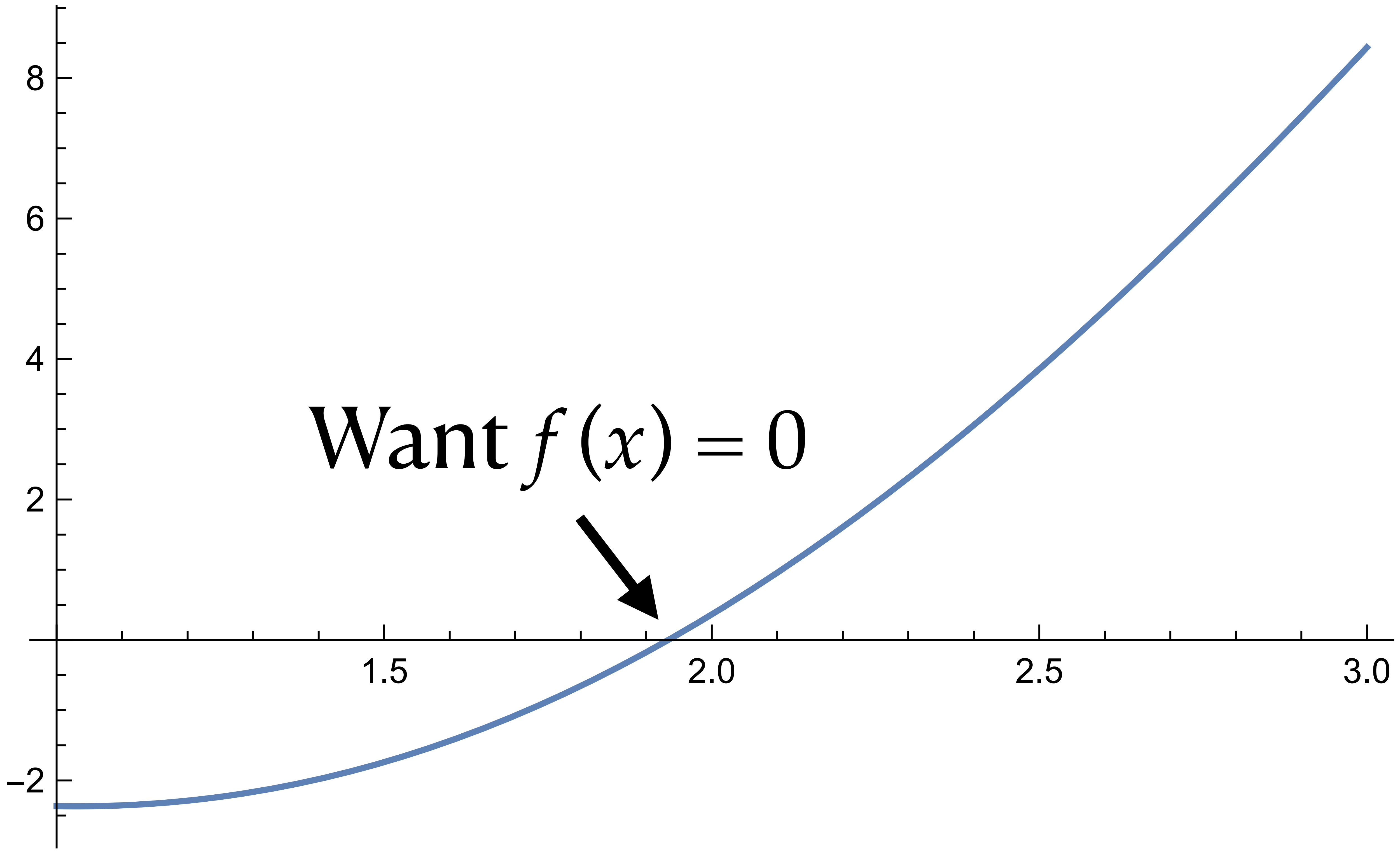
$$f(x) := x^2 - 4\sin x$$



A model problem

Can more or less read off solution from graph, *how difficult can it be?*

$$f(x) := x^2 - 4\sin x$$

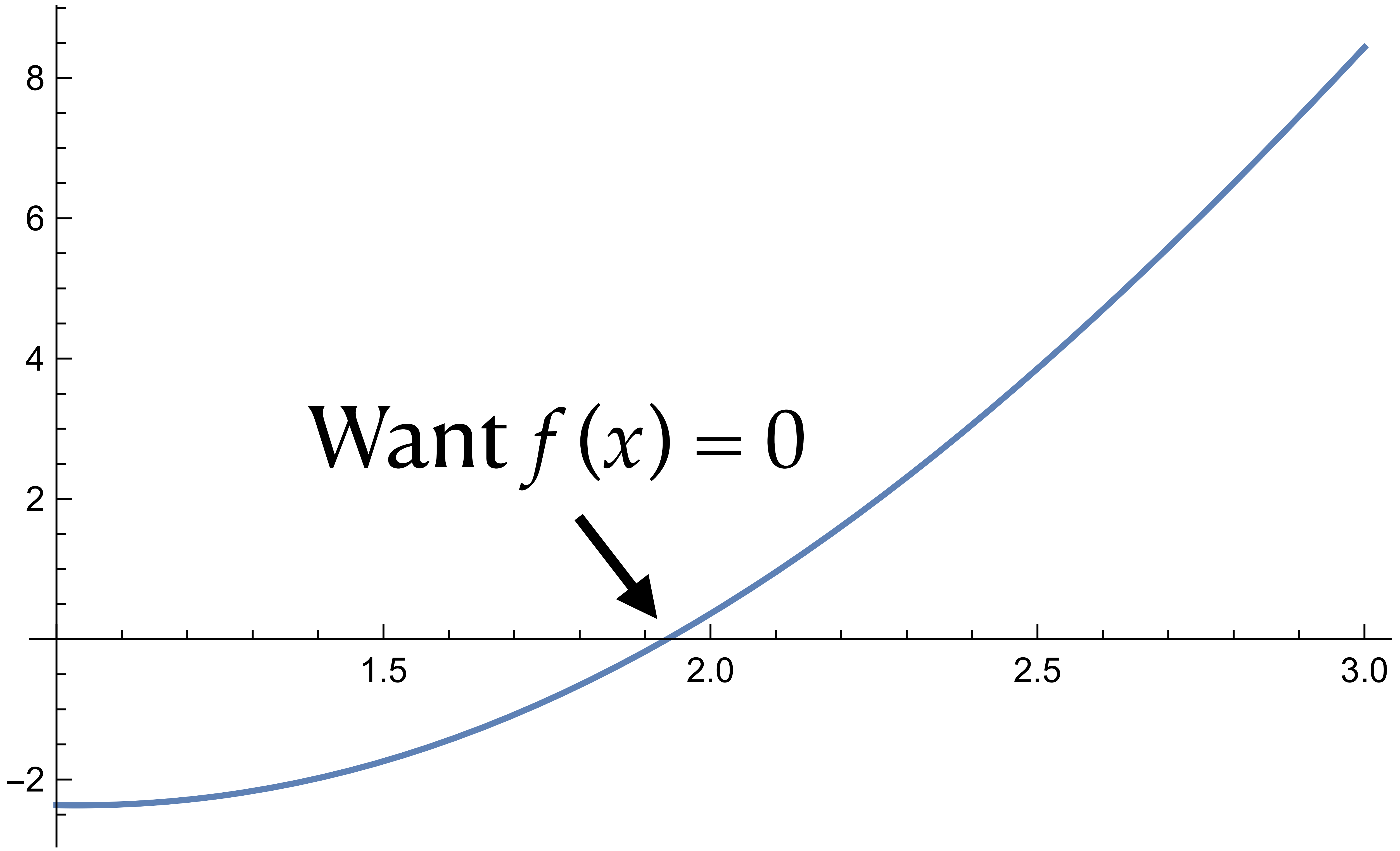


A model problem

Can more or less read off solution from graph, *how difficult can it be?*

$$f(x) := x^2 - 4\sin x$$

*Plot required thousands of function evaluations, each one is potentially **very expensive** (evaluating f(x) once: test a new drug compound on patients, build a rocket and collect a sample on Mars.)*



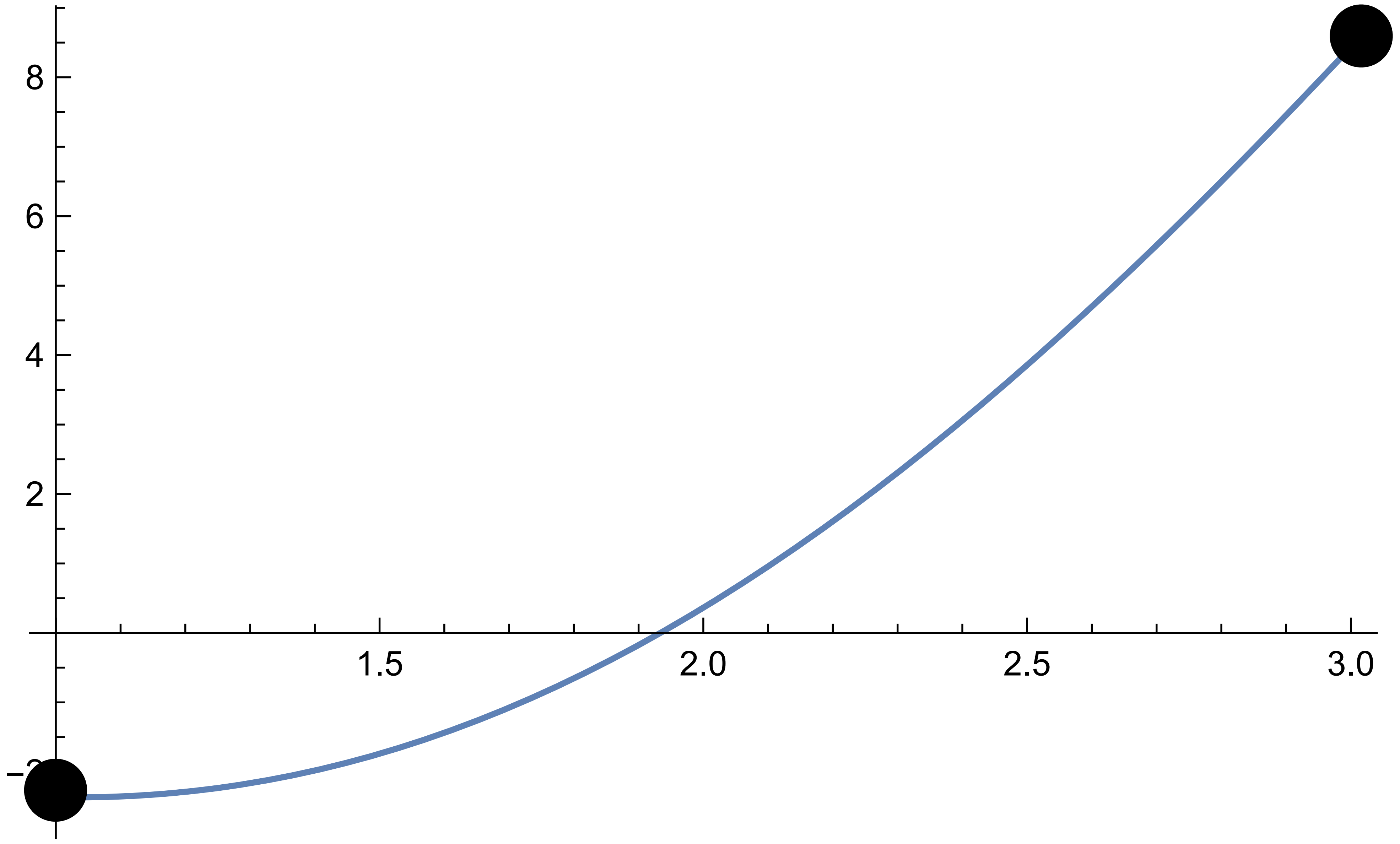
Intermediate Value Theorem

Intermediate Value Theorem

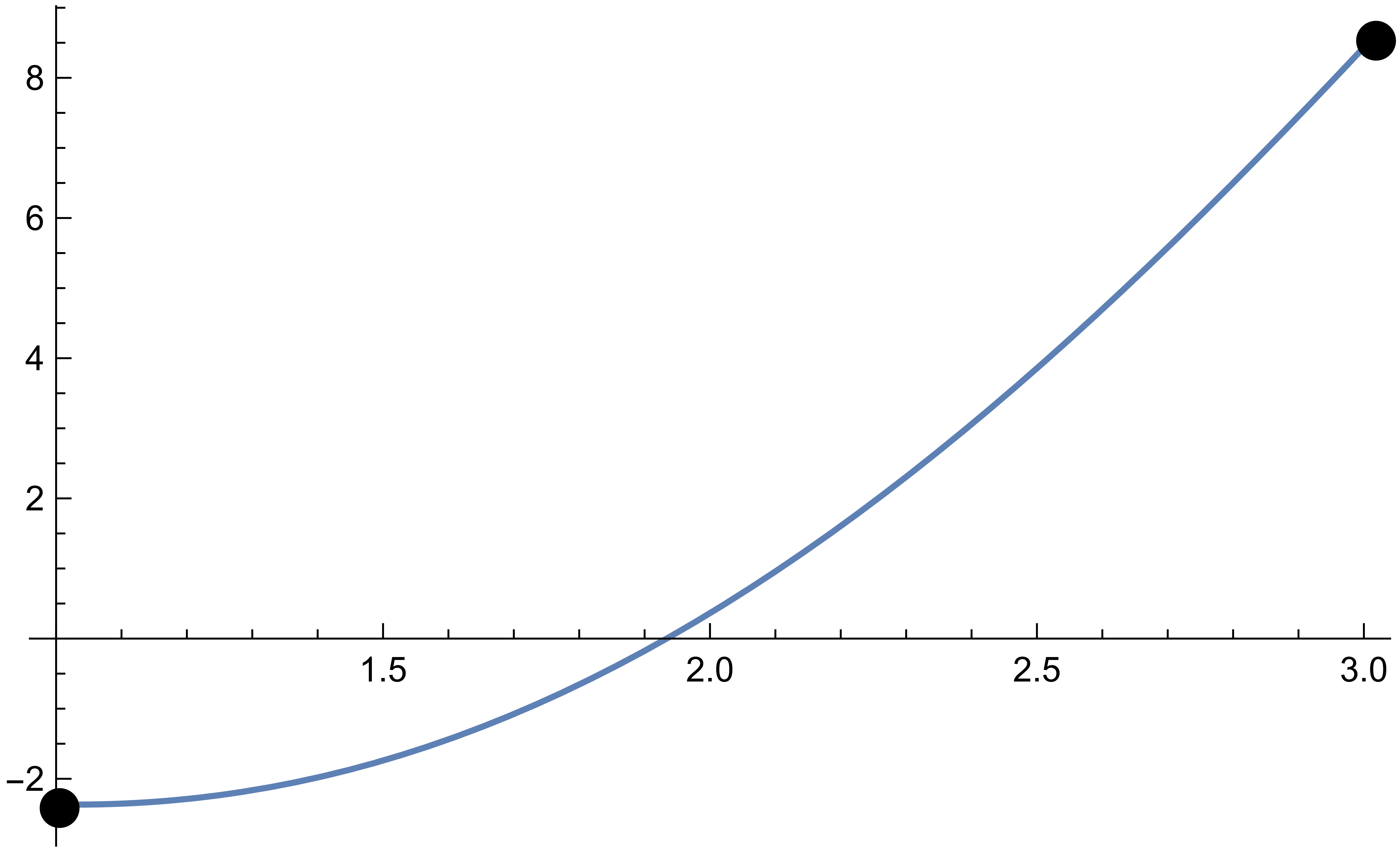
If f is continuous and $f(x_0) = y_0,\ f(x_1) = y_1,$ then $f(x)$ on (x_0, x_1) must pass through every value between y_0 and y_1.

Intermediate Value Theorem

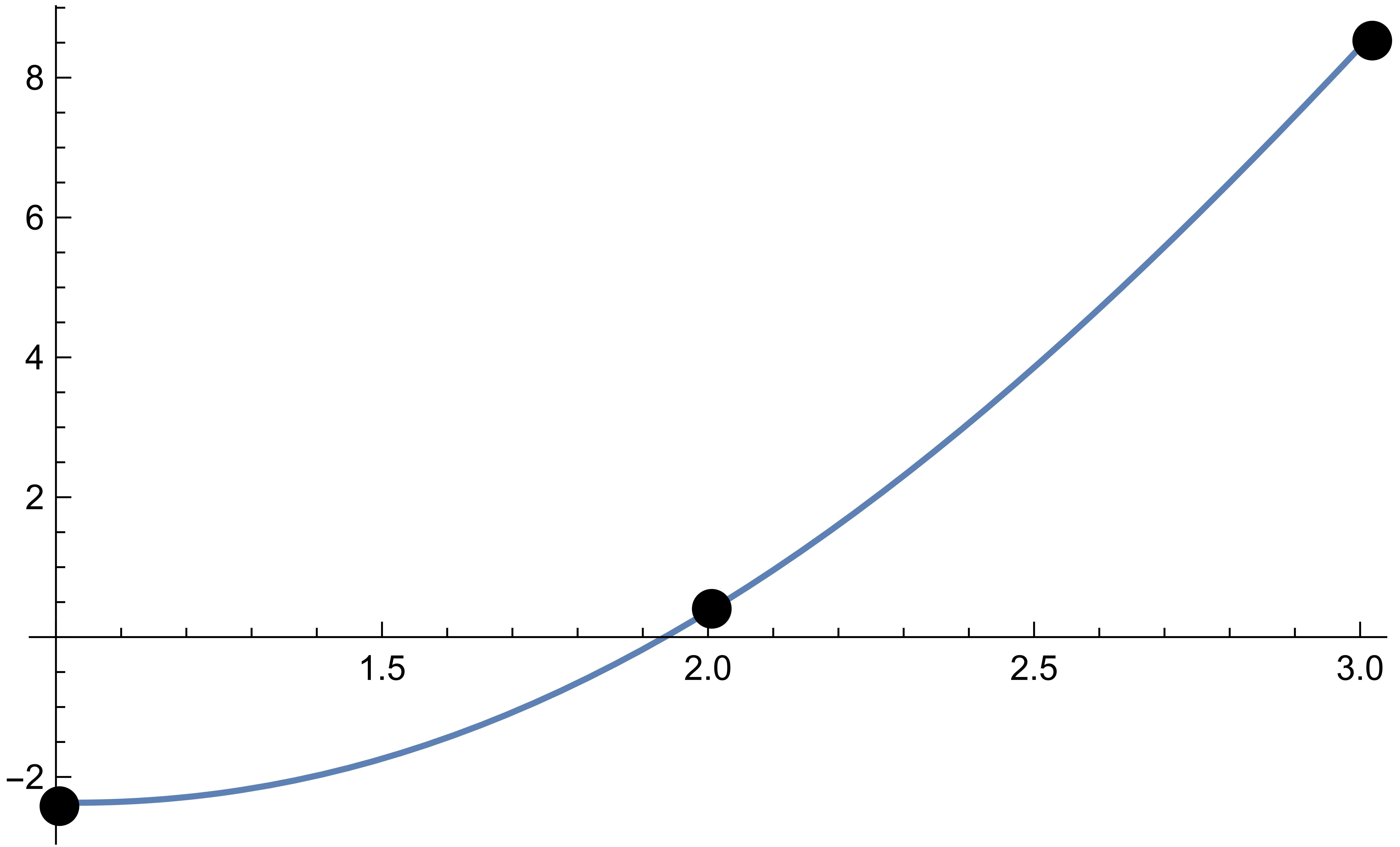
If f is continuous and $f(x_0) = y_0,\ f(x_1) = y_1,$ then $f(x)$ on (x_0, x_1) must pass through <u>every</u> value between y_0 and y_1.



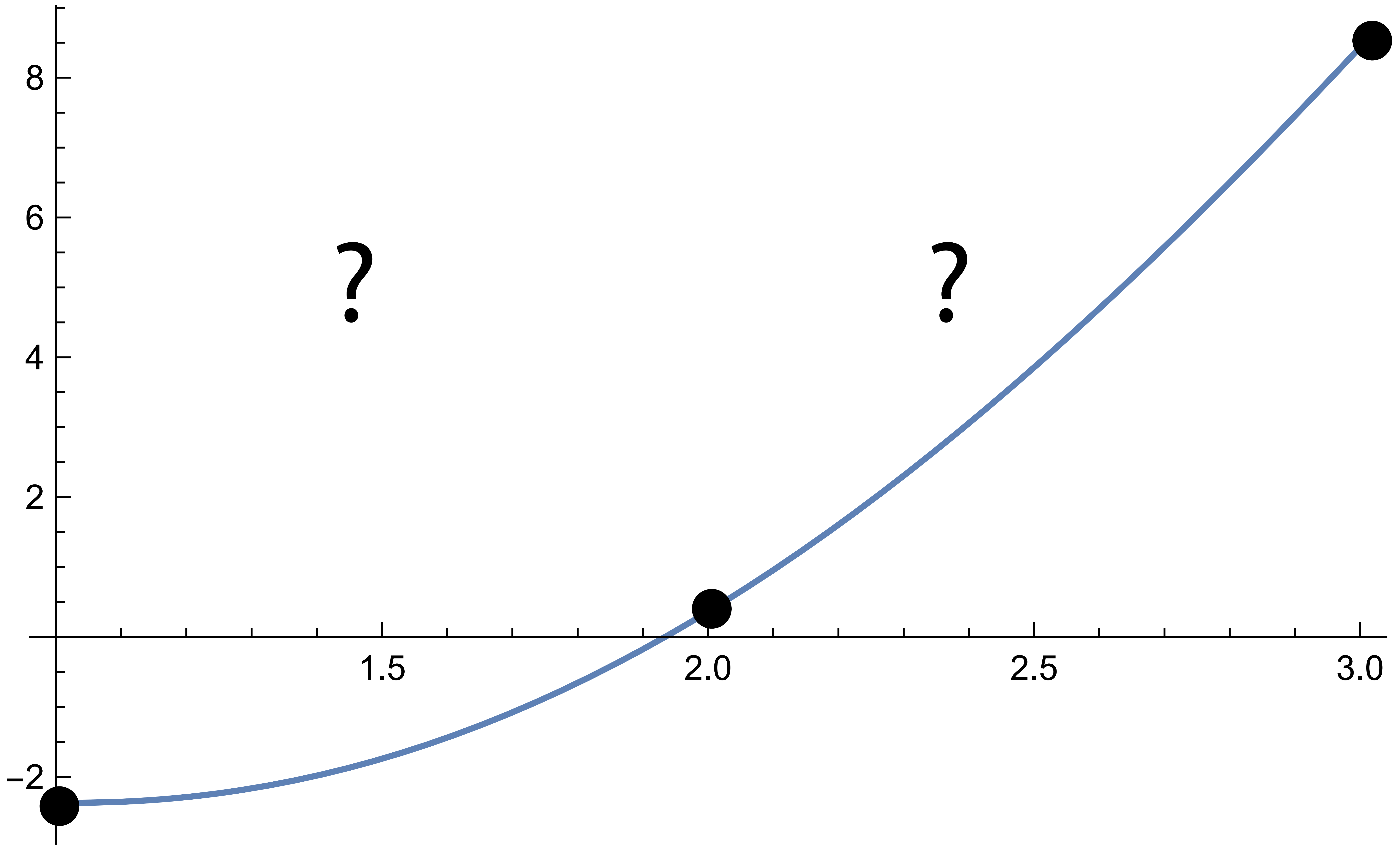
Use of continuity in our model problem



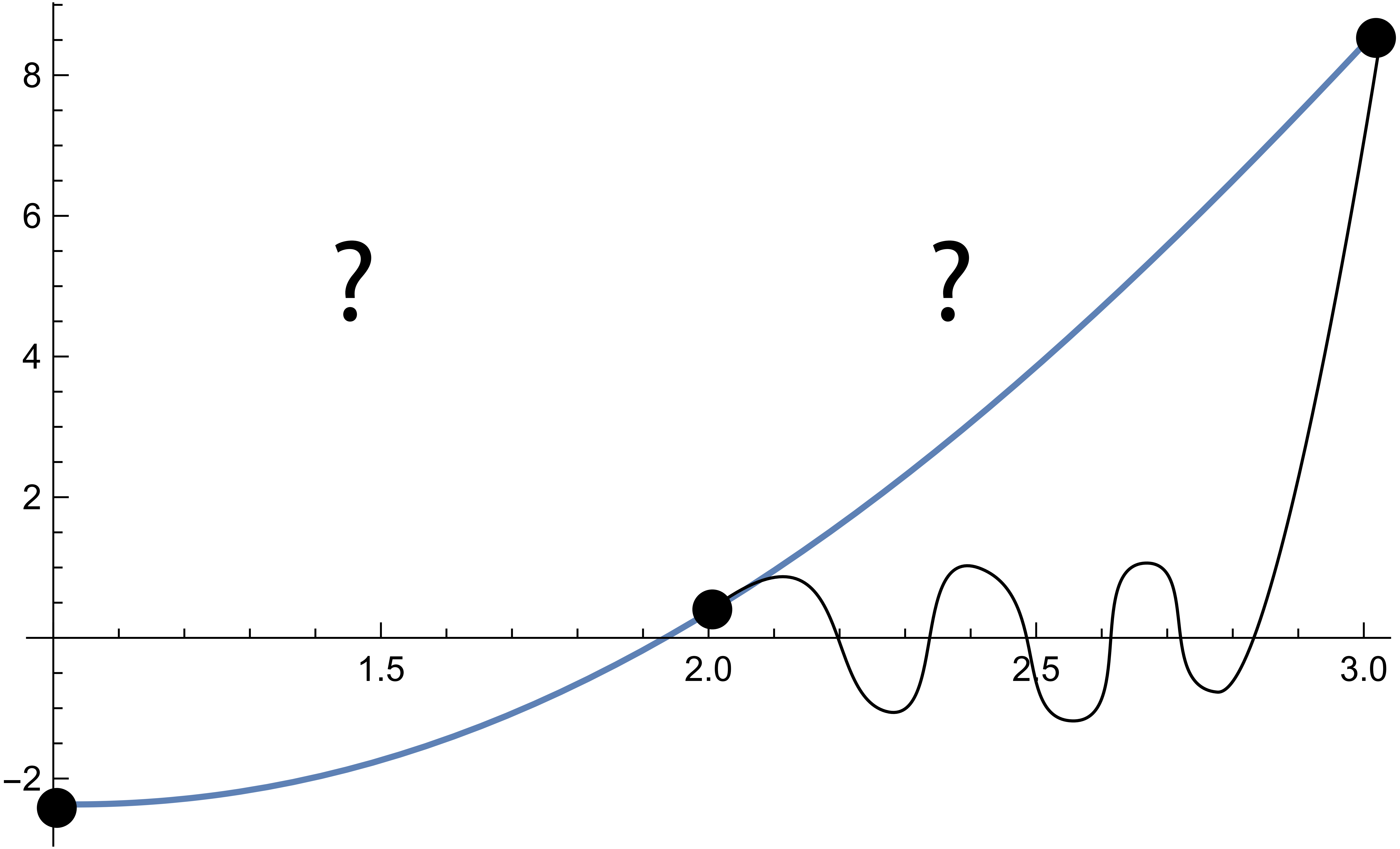
Use of continuity in our model problem



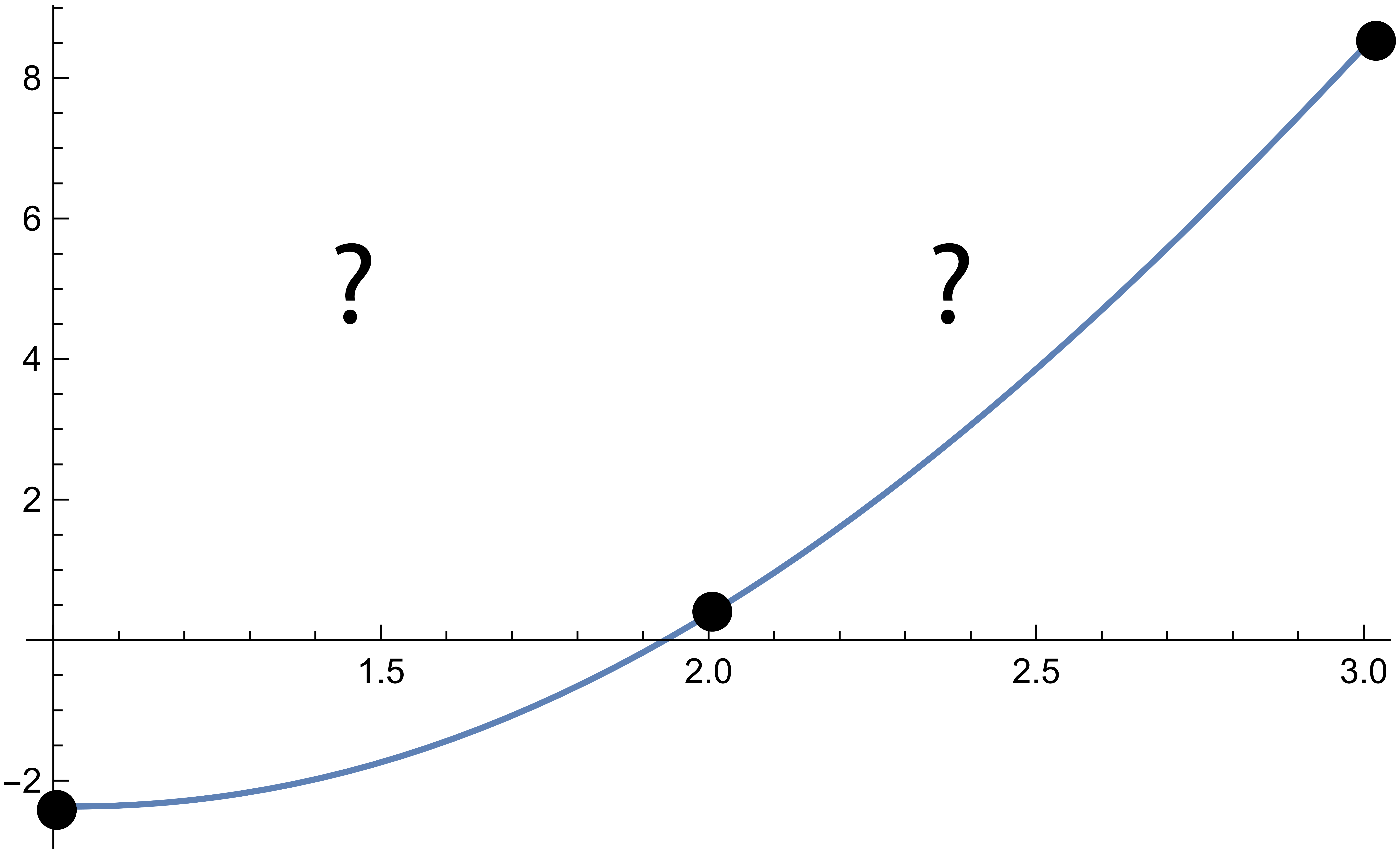
Use of continuity in our model problem



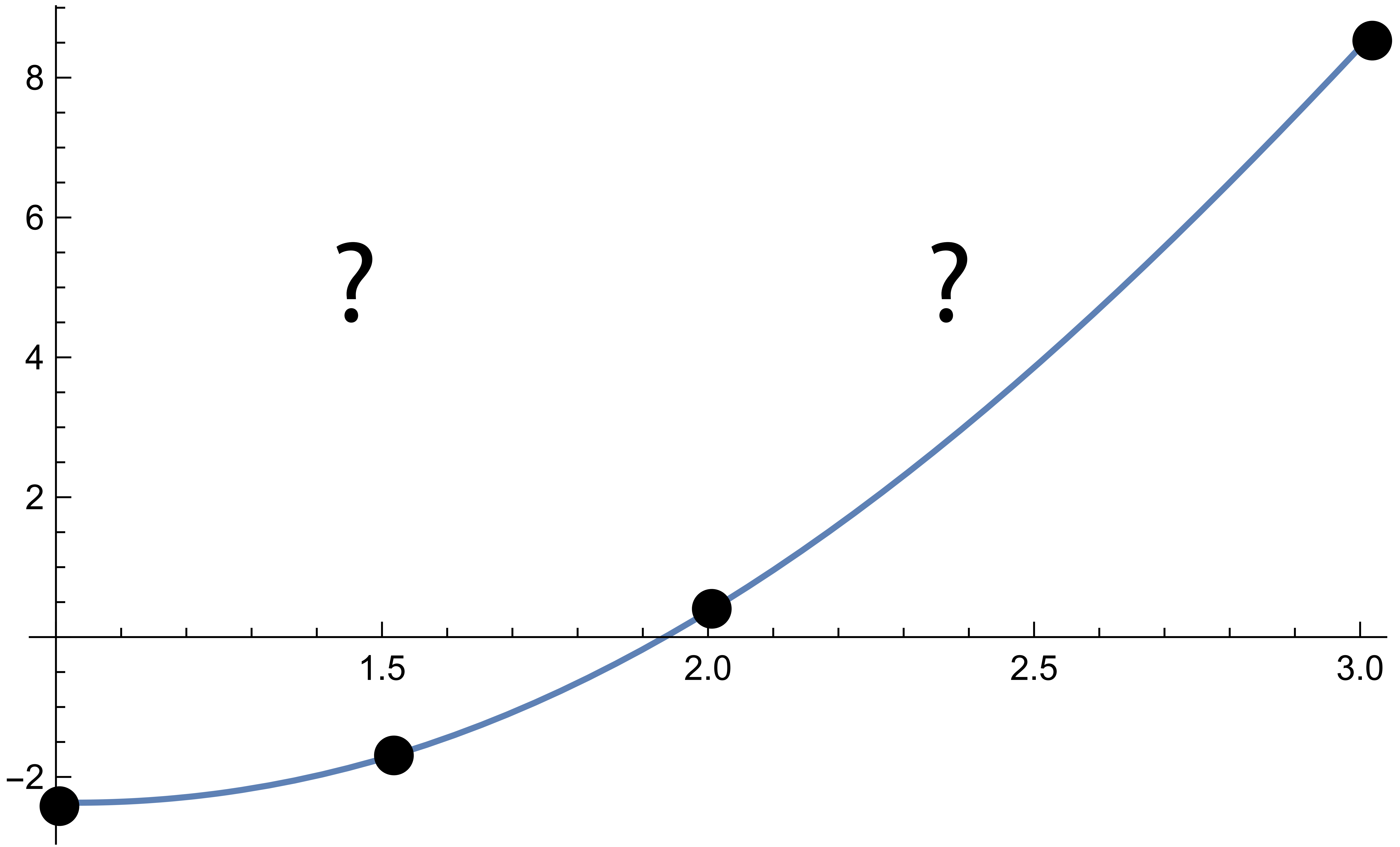
Use of continuity in our model problem



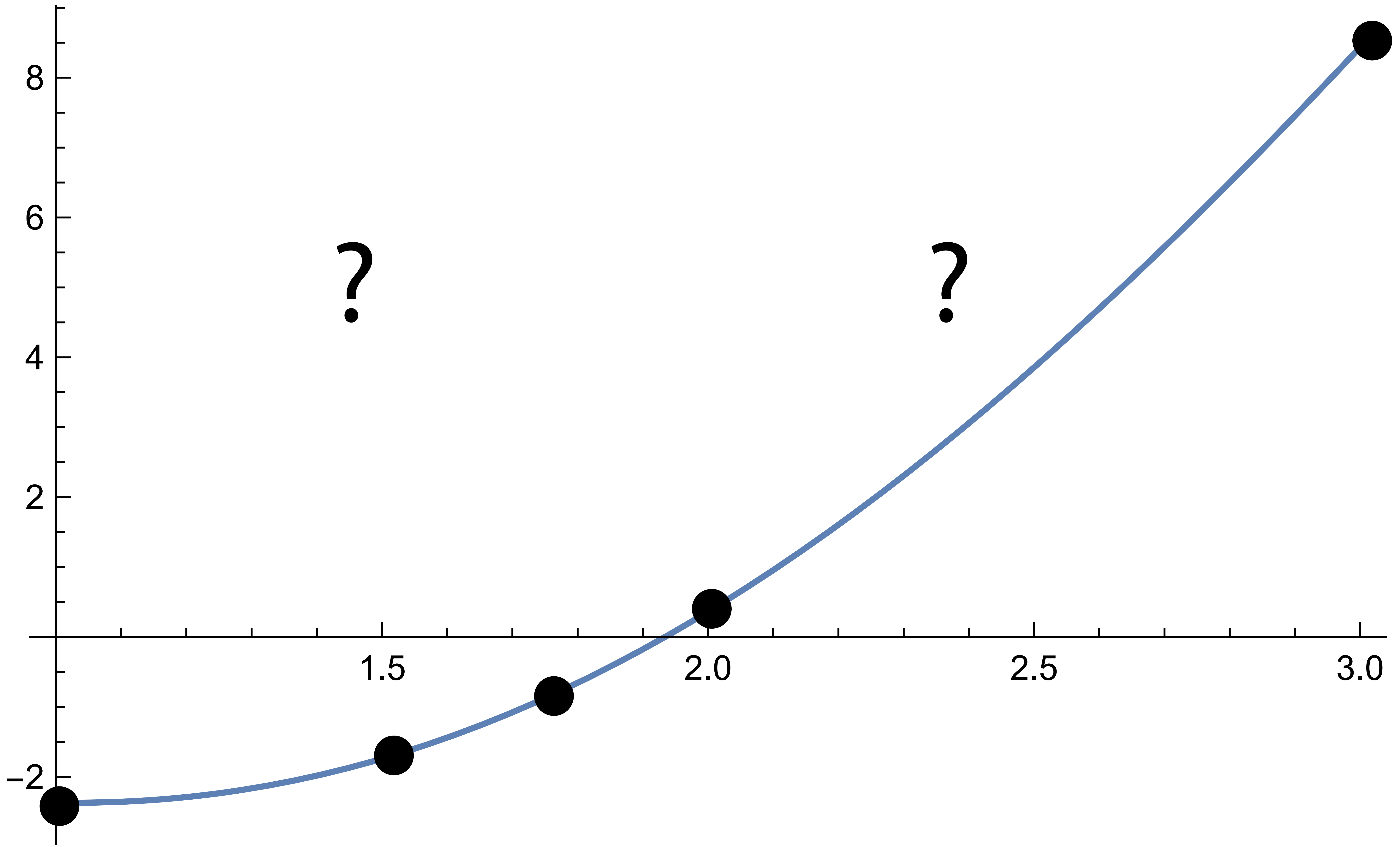
Use of continuity in our model problem



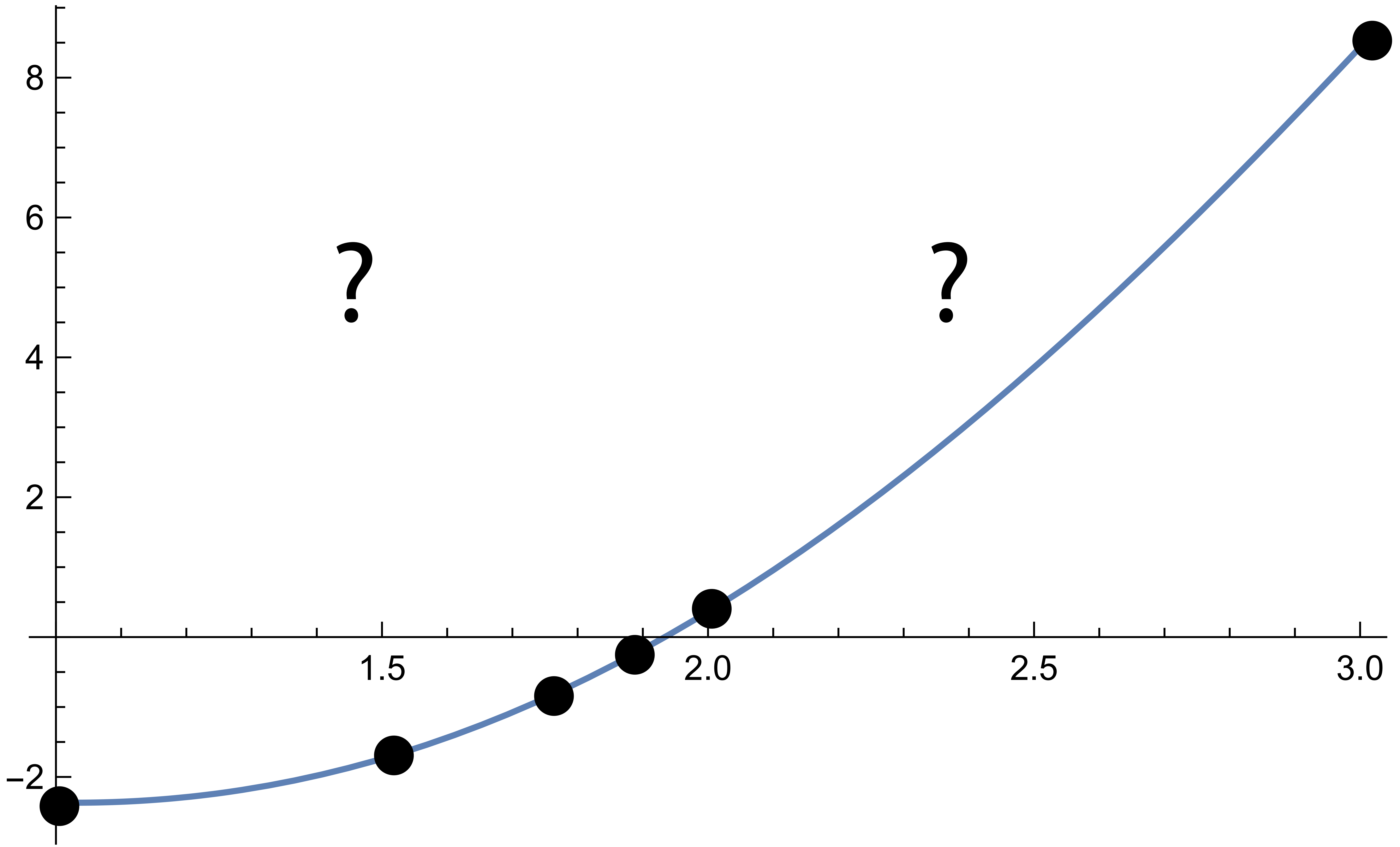
Use of continuity in our model problem



Use of continuity in our model problem

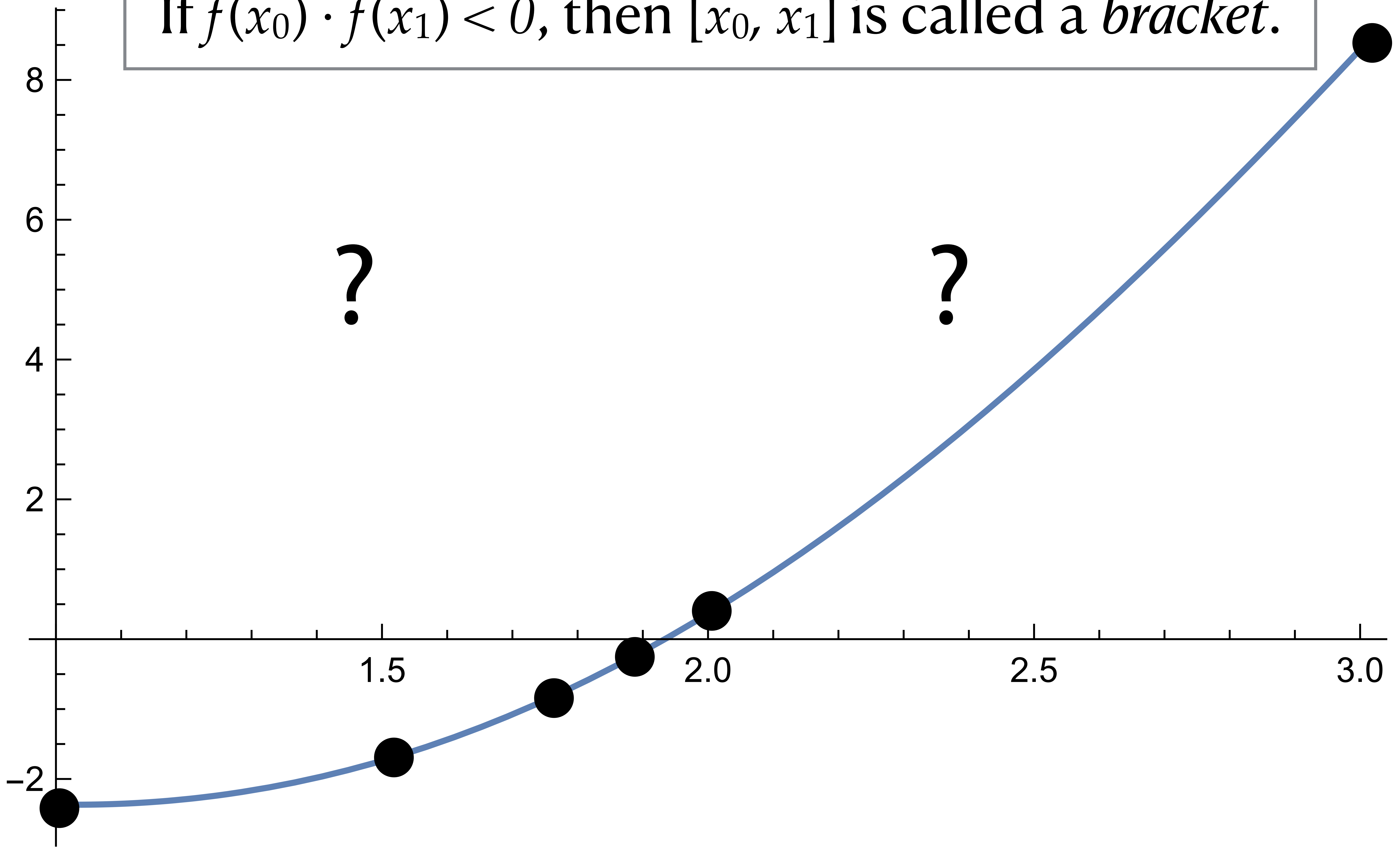


Use of continuity in our model problem



Use of continuity in our model problem

If $f(x_0) \cdot f(x_1) < 0$, then $[x_0, x_1]$ is called a *bracket*.



Bisection search

function *"bracket"* *stopping criterion thresholds*

```
def bisect(f, l, r, eps₁, eps₂):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Bisection search

function *“bracket”* *stopping criterion thresholds*

```
def bisect(f, l, r, eps1, eps2):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps1 or np.abs(l - r) < eps2:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Bisection search

function *"bracket"* *stopping criterion thresholds*

```
def bisect(f, l, r, eps₁, eps₂):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Bisection search

function *“bracket”* *stopping criterion thresholds*

```
def bisect(f, l, r, eps₁, eps₂):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Is this code optimal?

Bisection search

function *"bracket"* *stopping criterion thresholds*

```
def bisect(f, l, r, eps₁, eps₂):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Is this code optimal?

No, it could be implemented with ~1/3 the number of *f*-evaluations.

Bisection search

function → f, *"bracket"* → l, r, *stopping criterion thresholds* → eps1, eps2

```
def bisect(f, l, r, eps1, eps2):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps1 or np.abs(l - r) < eps2:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Does not use function values!
(only their sign is relevant)

Is this code optimal?

No, it could be implemented with ~1/3 the number of *f*-evaluations.

Bisection search

function *"bracket"* *stopping criterion thresholds*

```
def bisect(f, l, r, eps₁, eps₂):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

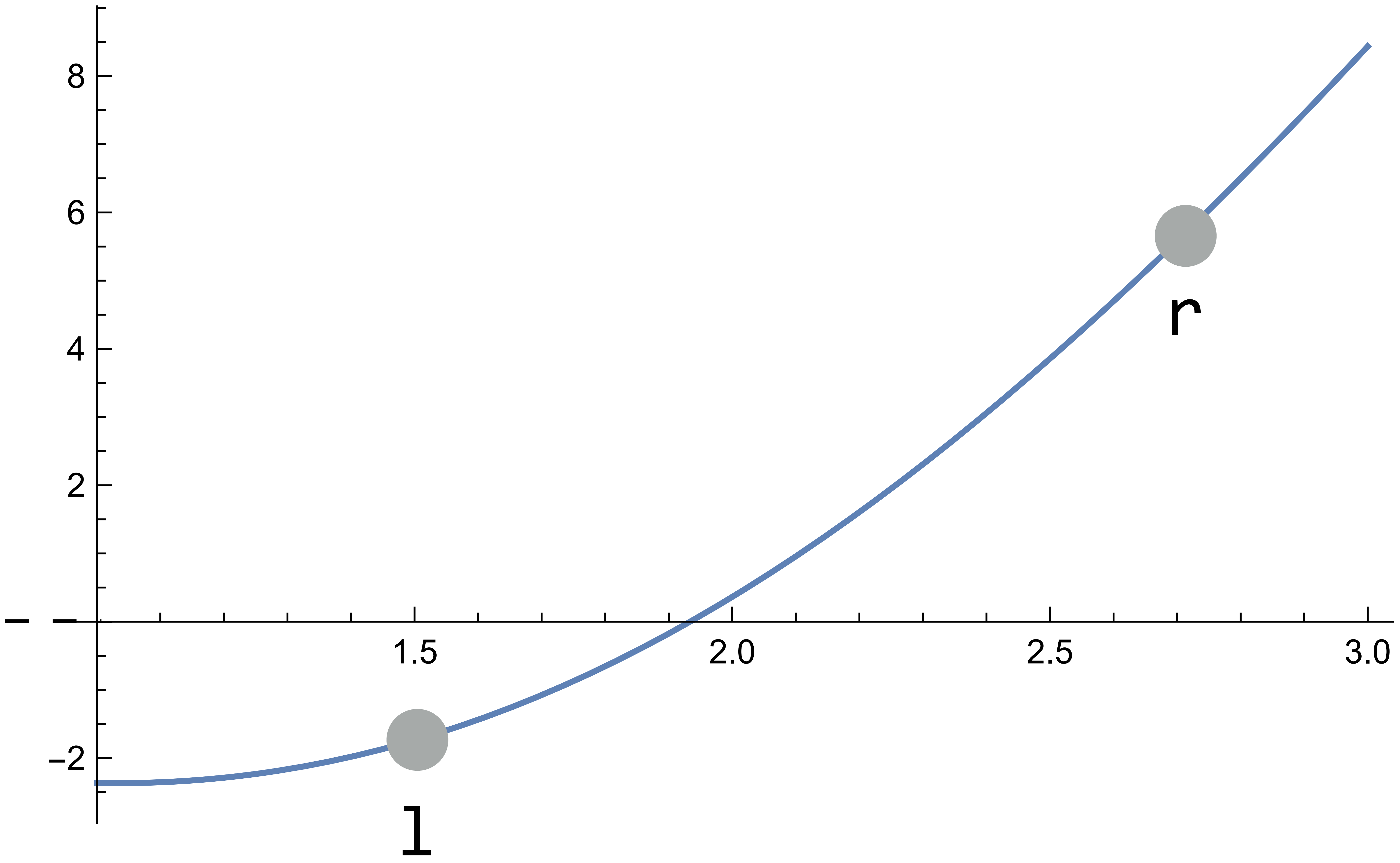
Does not use function values!
(only their sign is relevant)

Is this code optimal?

No, it could be implemented with ~1/3 the number of *f*-evaluations.

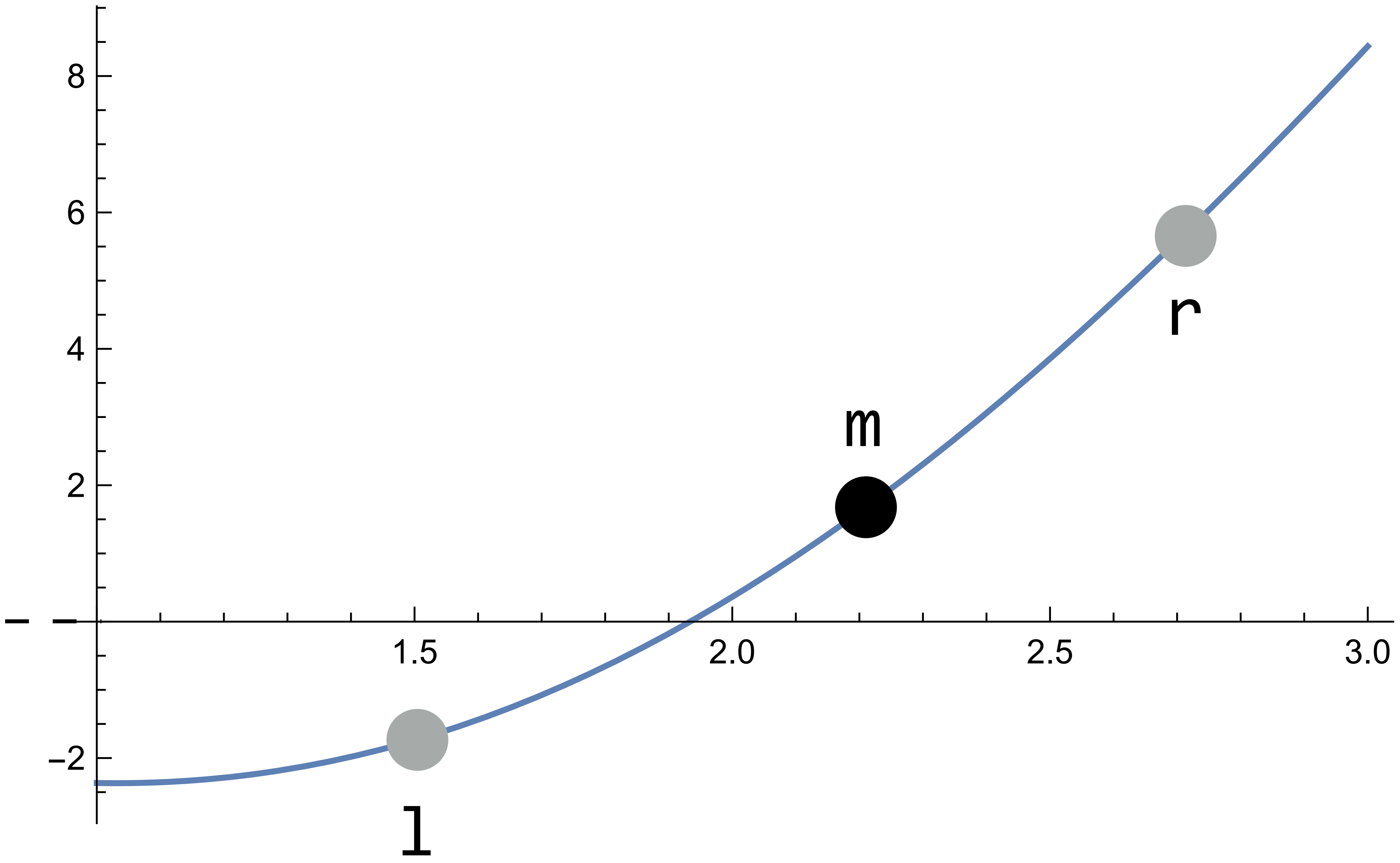
Stopping criteria for nonlinear methods

```
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
    return m
```



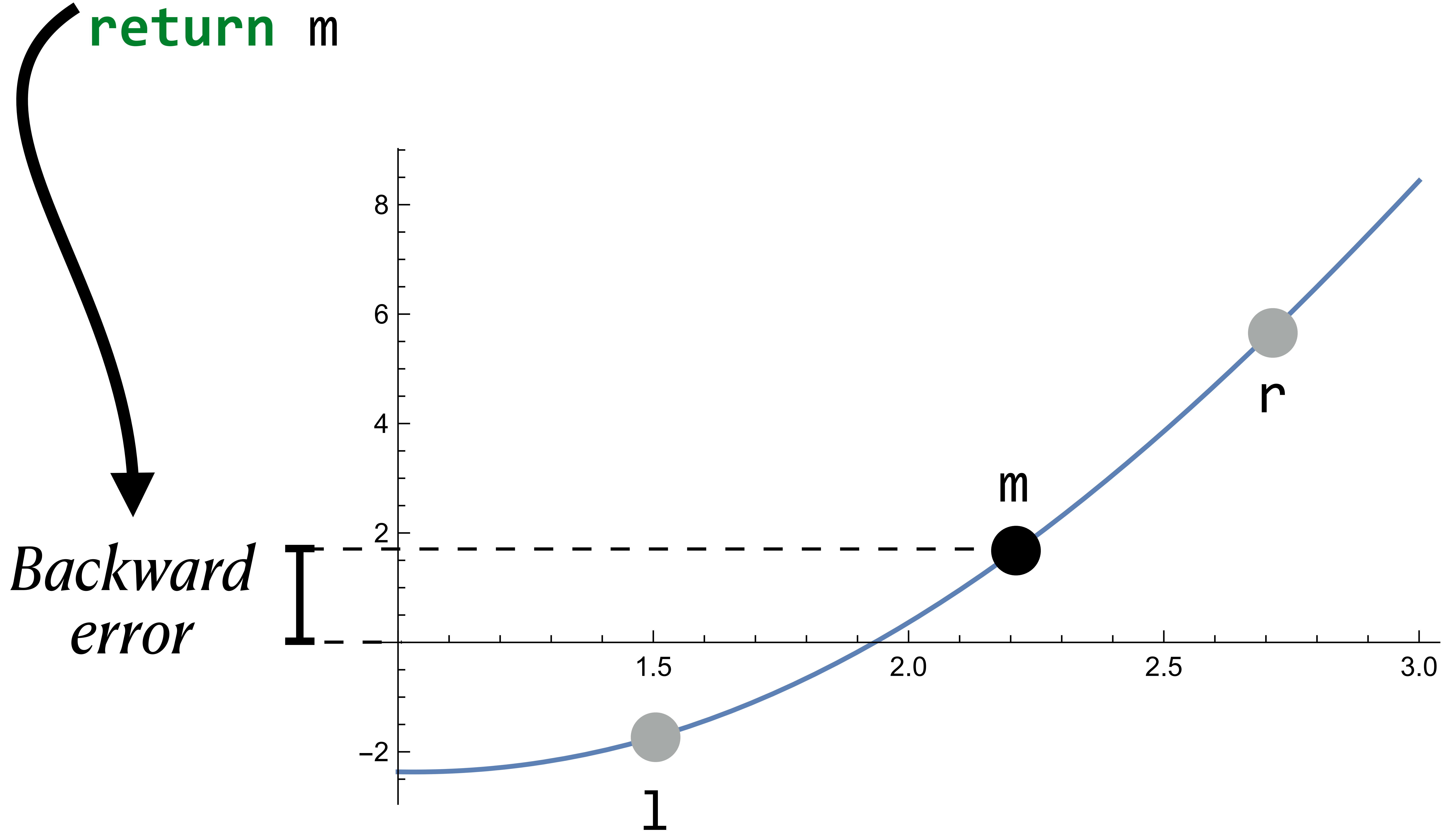
Stopping criteria for nonlinear methods

```python
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
    return m
```



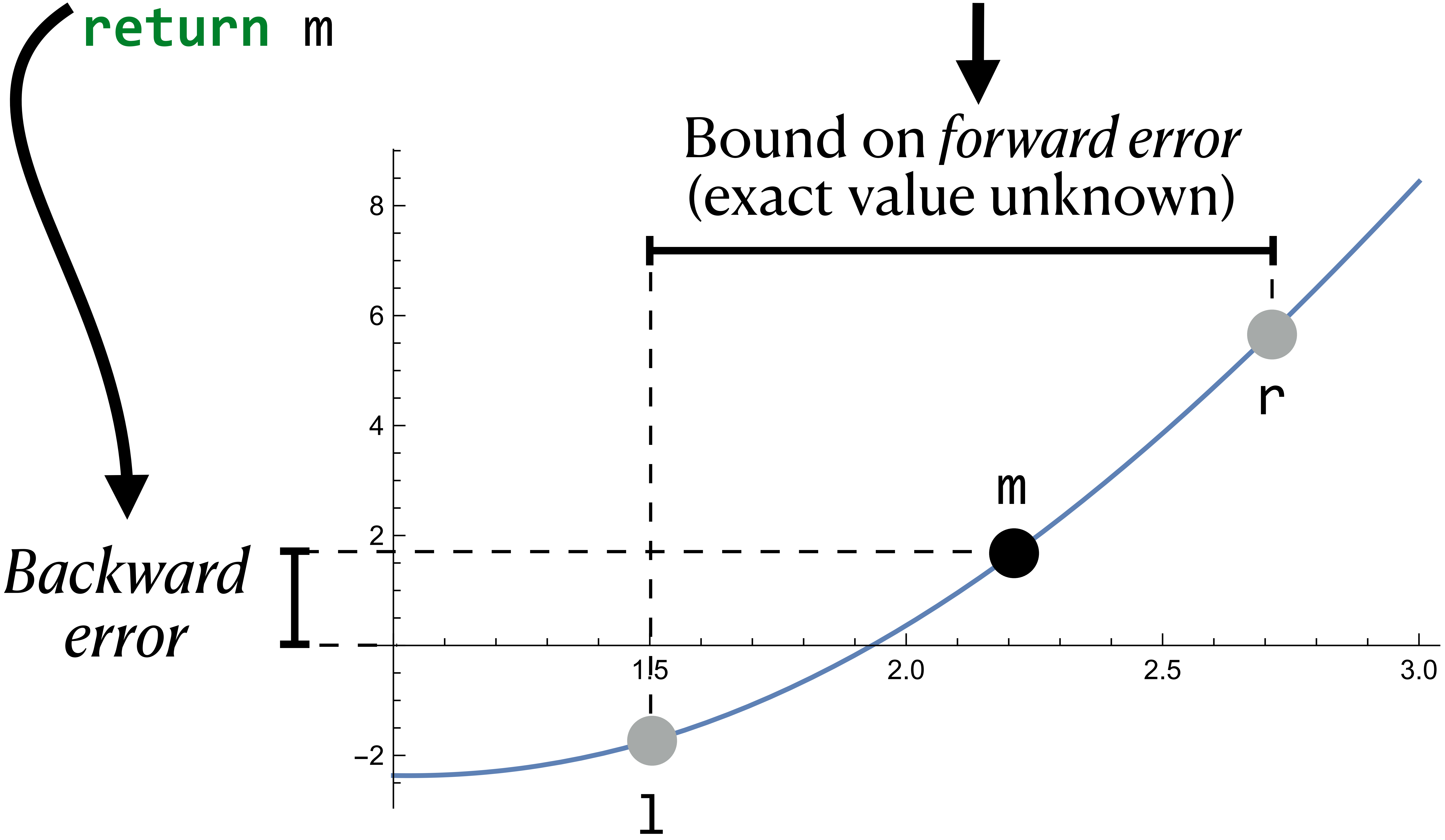
Stopping criteria for nonlinear methods

```
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
    return m
```



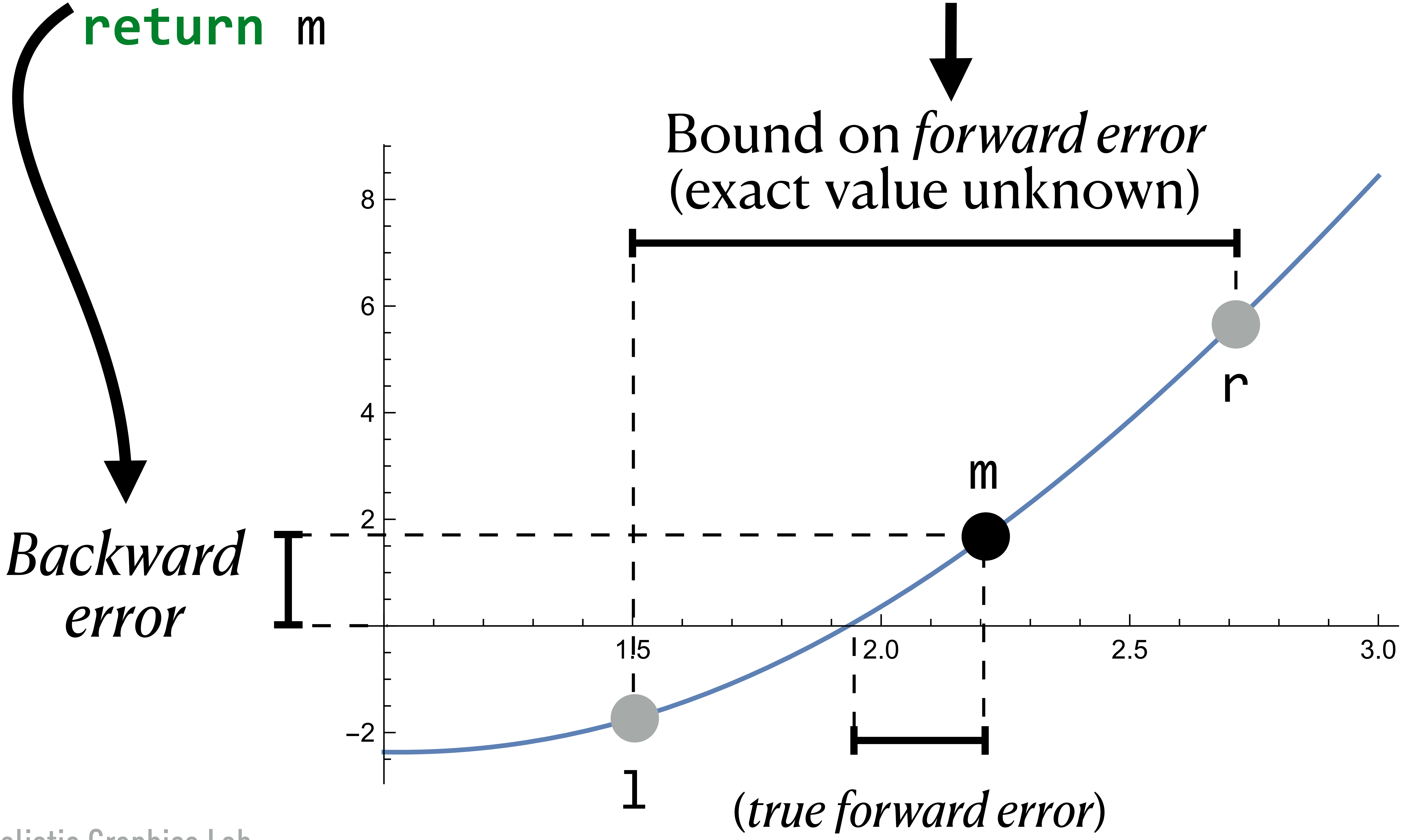
Stopping criteria for nonlinear methods

```
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
    return m
```

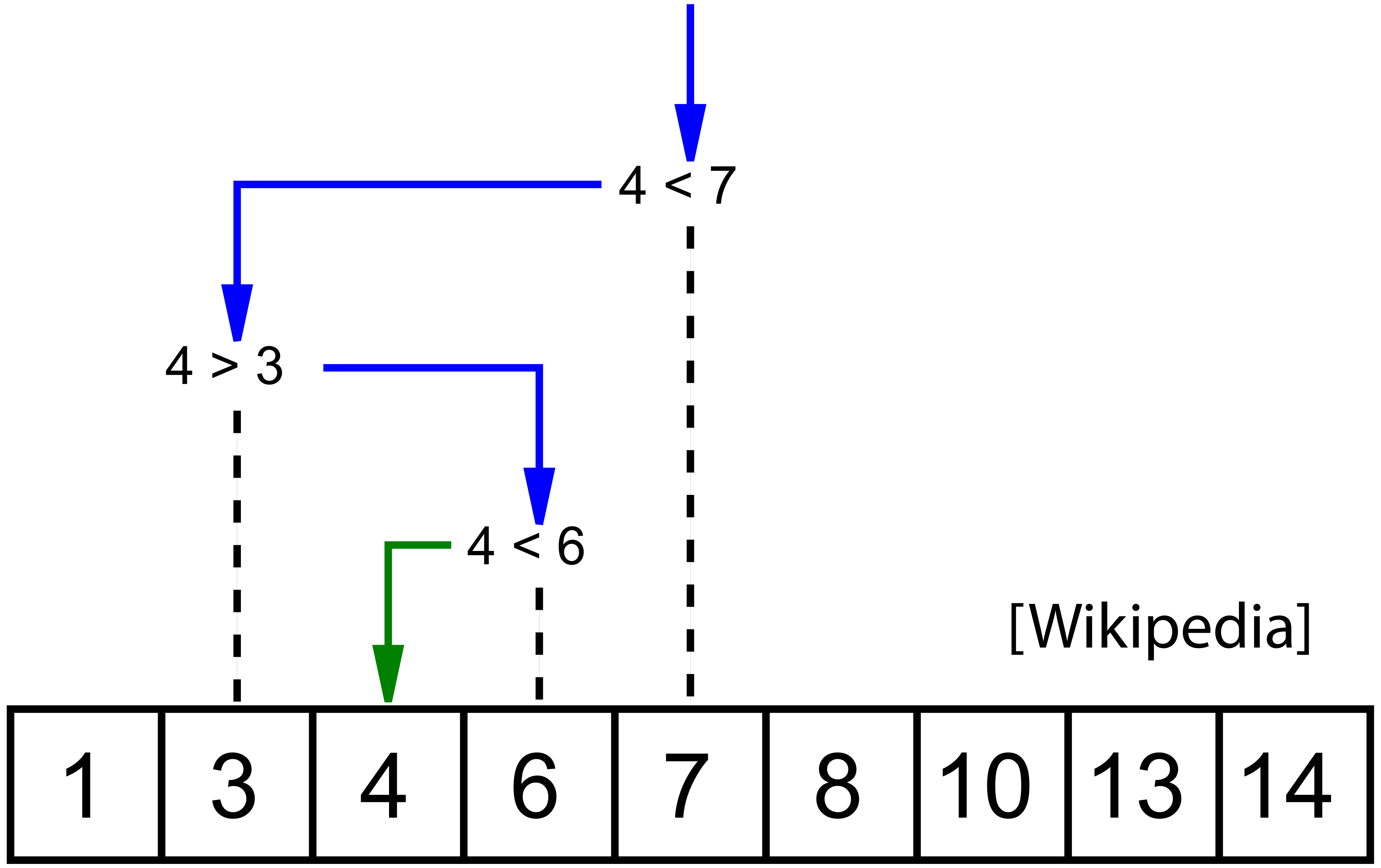


Stopping criteria for nonlinear methods

```
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
    return m
```



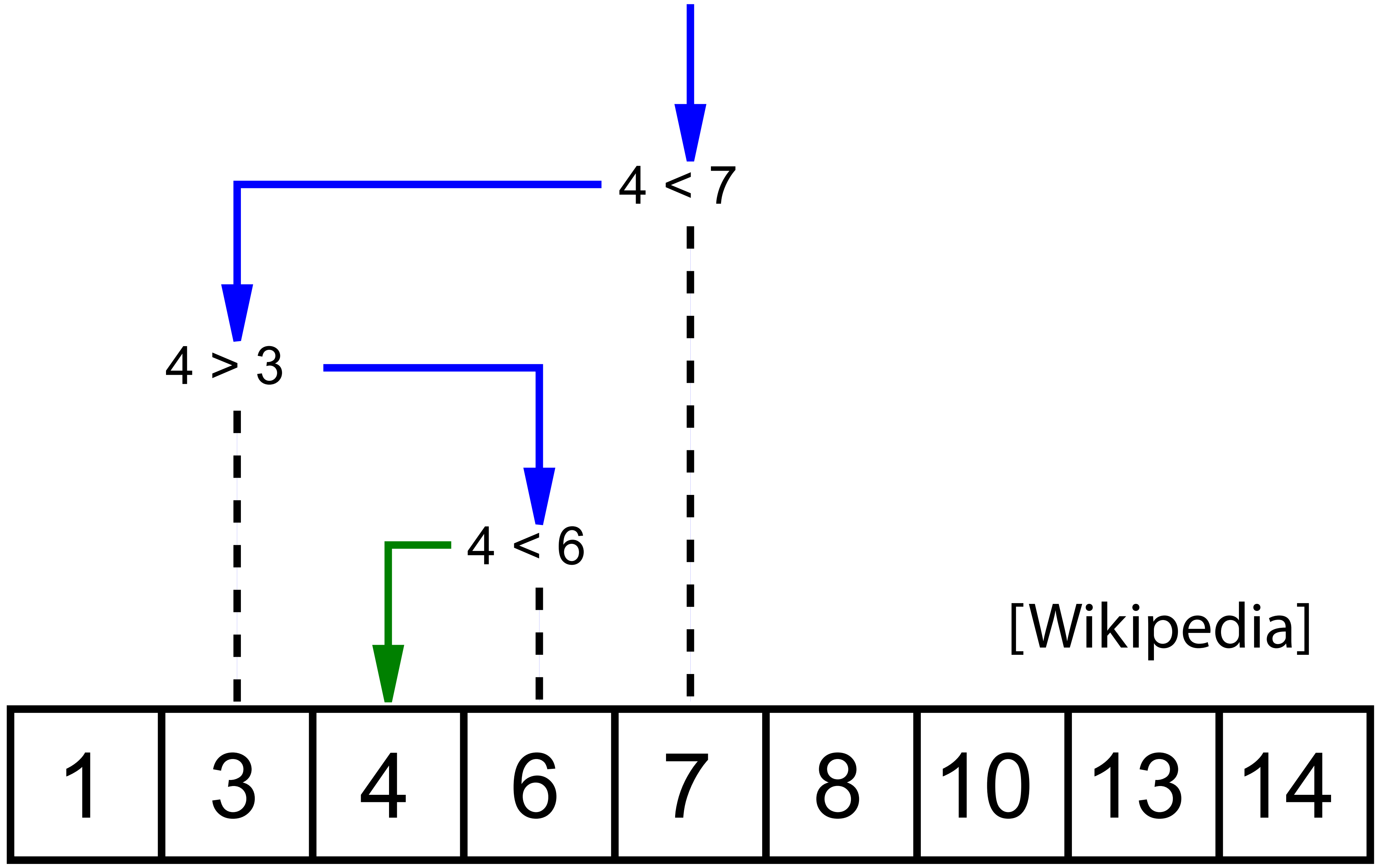
Does this seem at all familiar?



[Wikipedia]

Binary search for 4 in sorted list.

Does this seem at all familiar?

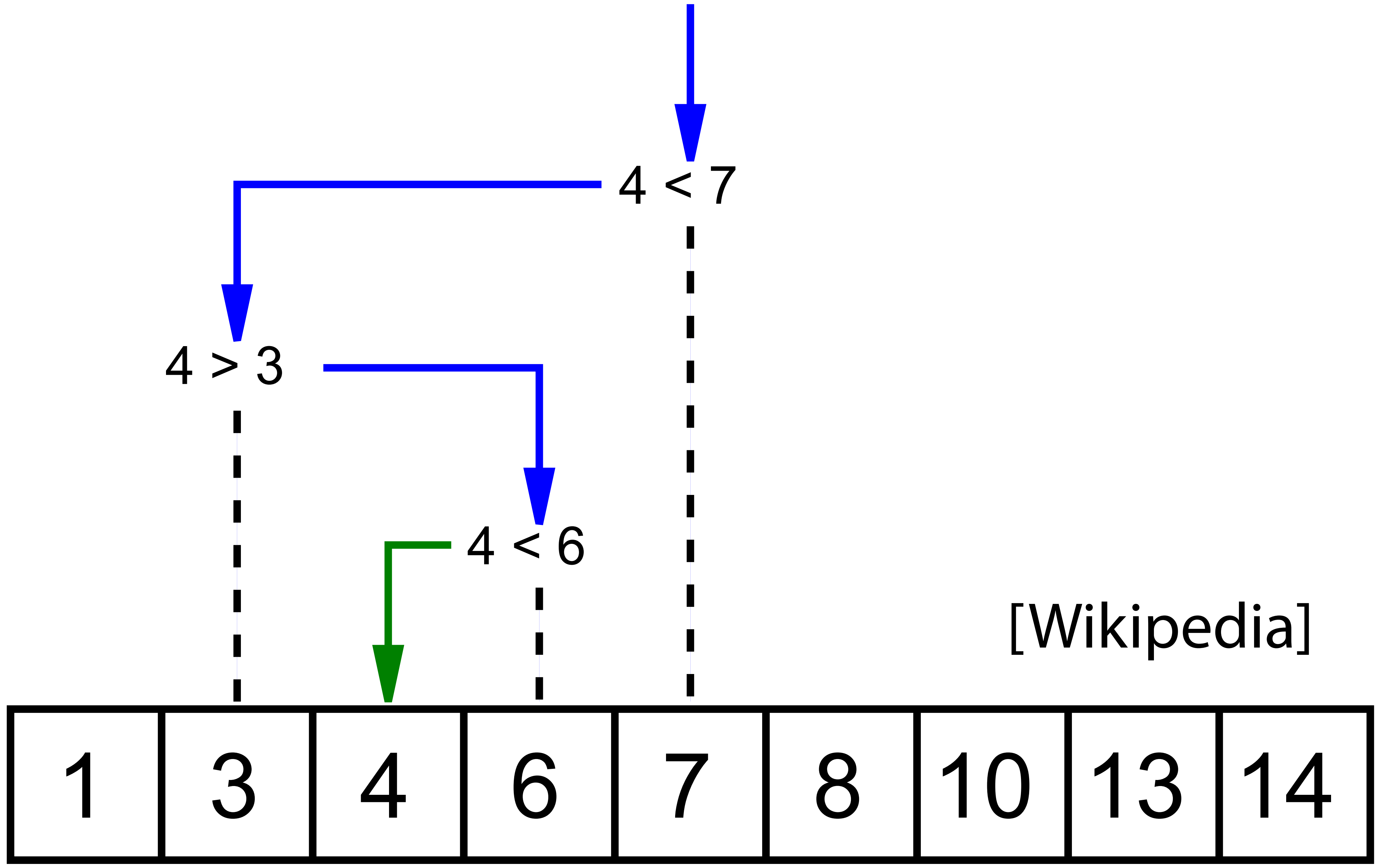


[Wikipedia]

Binary search for 4 in sorted list.

Complexity of (discrete) algorithm: $O(\log n)$

Does this seem at all familiar?



Binary search for 4 in sorted list.

Complexity of (discrete) algorithm: $O(\log n)$

Can we find an analogy of "*complexity*" for root finding?

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

- o is called the **order of convergence**, which tells us how quickly the algorithm convergences.
  - $o = 1$: linear convergence, $o = 2$: quadratic convergence, etc.

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

- o is called the **order of convergence**, which tells us how quickly the algorithm convergences.
  - $o = 1$: linear convergence, $o = 2$: quadratic convergence, etc.
- r is called the **rate of convergence**. It distinguishes convergence speed of algorithms with the same order.

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Generally: $E_{k+1} = \frac{1}{2}E_k$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Generally: $E_{k+1} = \frac{1}{2} E_k \Leftrightarrow \frac{E_{k+1}}{E_k^1} = \frac{1}{2}$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Generally: $E_{k+1} = \frac{1}{2} E_k \Leftrightarrow \frac{E_{k+1}}{E_k^1} = \frac{1}{2}$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Generally: $E_{k+1} = \frac{1}{2} E_k \Leftrightarrow \frac{E_{k+1}}{E_k^1} = \frac{1}{2}$

In other words: **order** of convergence = 1 (*linear*)
rate of convergence = 1 / 2

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l) / 2$

Generally: $E_{k+1} = \frac{1}{2} E_k \Leftrightarrow \frac{E_{k+1}}{E_k^1} = \frac{1}{2}$

In other words: **order** of convergence = 1 (*linear*)
rate of convergence = 1 / 2

A method with a **linear order of convergence** gains a fixed number of accurate digits per iteration (depending on **rate**). Here:

- 1 base-2 digit every iteration. 1 base-10 digit every $\frac{\log 10}{\log 2} \approx 3$ iterations.

Convergence speed of bisection

Let's apply bisection to the root-finding example function with

$$f(x) := x^2 - 4\sin x$$

Here, l and r denote the bracket; the solution lies in between.

l	$f(l)$	r	$f(r)$
1.000000	−2.365884	3.000000	8.435520
1.000000	−2.365884	2.000000	0.362810
1.500000	−1.739980	2.000000	0.362810
1.750000	−0.873444	2.000000	0.362810
1.875000	−0.300718	2.000000	0.362810
1.875000	−0.300718	1.937500	0.019849
1.906250	−0.143255	1.937500	0.019849
1.921875	−0.062406	1.937500	0.019849
1.929688	−0.021454	1.937500	0.019849

l	$f(l)$	r	$f(r)$
1.933594	−0.000846	1.937500	0.019849
1.933594	−0.000846	1.935547	0.009491
1.933594	−0.000846	1.934570	0.004320
1.933594	−0.000846	1.934082	0.001736

[Heath]

Convergence speed of bisection

Let's apply bisection to the root-finding example function with

$$f(x) := x^2 - 4\sin x$$

Here, l and r denote the bracket; the solution lies in between.

l	$f(l)$	r	$f(r)$
1.000000	−2.365884	3.000000	8.435520
1.000000	−2.365884	2.000000	0.362810
1.500000	−1.739980	2.000000	0.362810
1.750000	−0.873444	2.000000	0.362810
1.875000	−0.300718	2.000000	0.362810
1.875000	−0.300718	1.937500	0.019849
1.906250	−0.143255	1.937500	0.019849
1.921875	−0.062406	1.937500	0.019849
1.929688	−0.021454	1.937500	0.019849

l	$f(l)$	r	$f(r)$
1.933594	−0.000846	1.937500	0.019849
1.933594	−0.000846	1.935547	0.009491
1.933594	−0.000846	1.934570	0.004320
1.933594	−0.000846	1.934082	0.001736

[Heath]

Convergence speed of bisection

Let's apply bisection to the root-finding example function with

$$f(x) := x^2 - 4\sin x$$

Here, l and r denote the bracket; the solution lies in between.

l	$f(l)$	r	$f(r)$
1.000000	−2.365884	3.000000	8.435520
1.000000	−2.365884	2.000000	0.362810
1.500000	−1.739980	2.000000	0.362810
1.750000	−0.873444	2.000000	0.362810
1.875000	−0.300718	2.000000	0.362810
1.875000	−0.300718	1.937500	0.019849
1.906250	−0.143255	1.937500	0.019849
1.921875	−0.062406	1.937500	0.019849
1.929688	−0.021454	1.937500	0.019849

l	$f(l)$	r	$f(r)$
1.933594	−0.000846	1.937500	0.019849
1.933594	−0.000846	1.935547	0.009491
1.933594	−0.000846	1.934570	0.004320
1.933594	−0.000846	1.934082	0.001736

[Heath]

Convergence speed of bisection

Let's apply bisection to the root-finding example function with

$$f(x) := x^2 - 4\sin x$$

Here, l and r denote the bracket; the solution lies in between.

l	$f(l)$	r	$f(r)$
1.000000	−2.365884	3.000000	8.435520
1.000000	−2.365884	2.000000	0.362810
1.500000	−1.739980	2.000000	0.362810
1.750000	−0.873444	2.000000	0.362810
1.875000	−0.300718	2.000000	0.362810
1.875000	−0.300718	1.937500	0.019849
1.906250	−0.143255	1.937500	0.019849
1.921875	−0.062406	1.937500	0.019849
1.929688	−0.021454	1.937500	0.019849

l	$f(l)$	r	$f(r)$
1.933594	−0.000846	1.937500	0.019849
1.933594	−0.000846	1.935547	0.009491
1.933594	−0.000846	1.934570	0.004320
1.933594	−0.000846	1.934082	0.001736

[Heath]

Midpoint = **1.933**838

True solution = 1.93375376282..

(13 iterations for ~4 digits)

Newton's method

- Based on first-order Taylor expansion:

$$f(x+h) \approx f(x) + f'(x)h$$

Newton's method

- Based on first-order Taylor expansion:

$$f(x+h) \approx f(x) + f'(x)h$$

- Let's set this approximation to zero and solve for h.

$$f(x+h) = 0 \Leftrightarrow h = -\frac{f(x)}{f'(x)}$$

Newton's method

- Based on first-order Taylor expansion:

$$f(x+h) \approx f(x) + f'(x)h$$

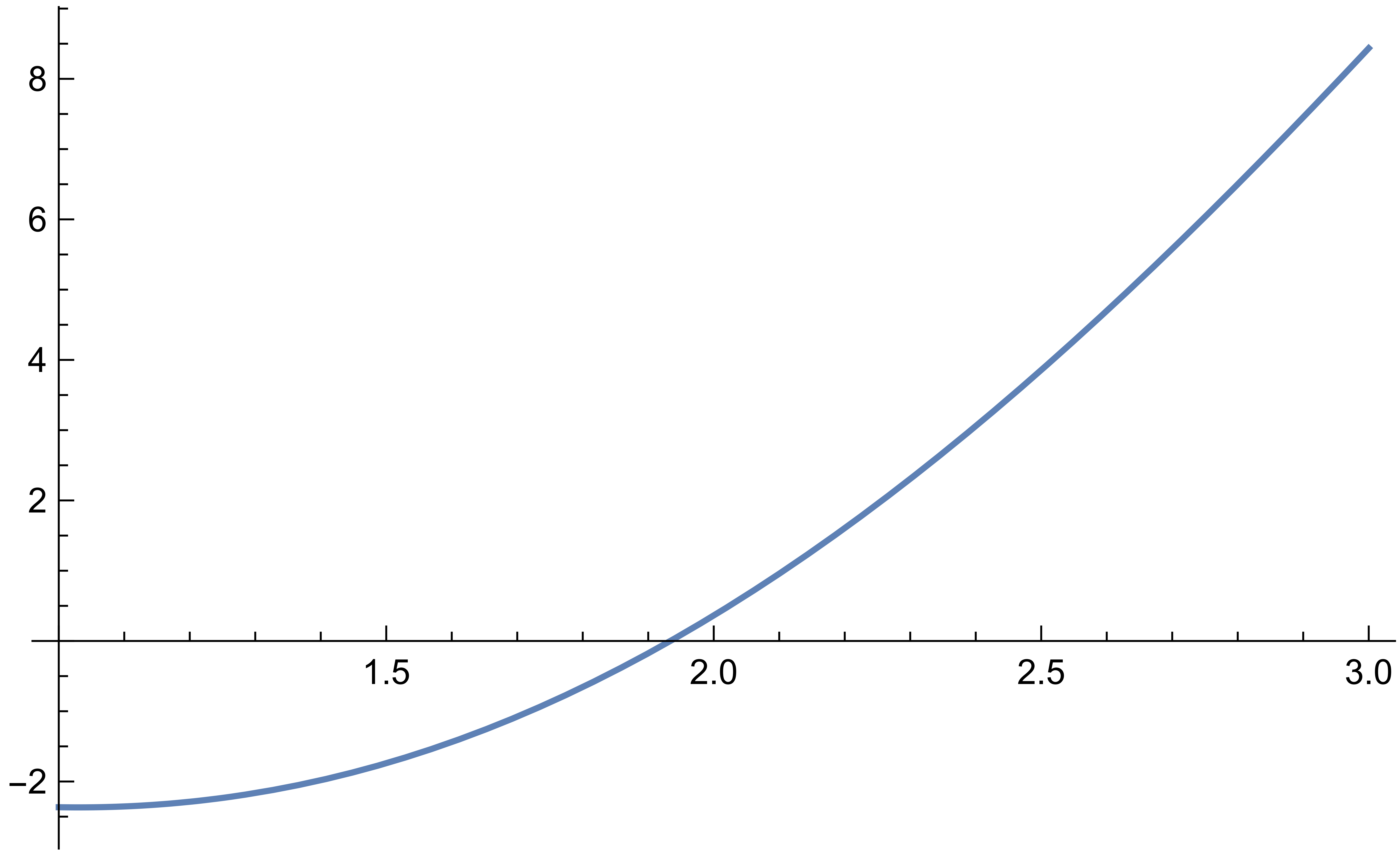
- Let's set this approximation to zero and solve for h.

$$f(x+h) = 0 \Leftrightarrow h = -\frac{f(x)}{f'(x)}$$

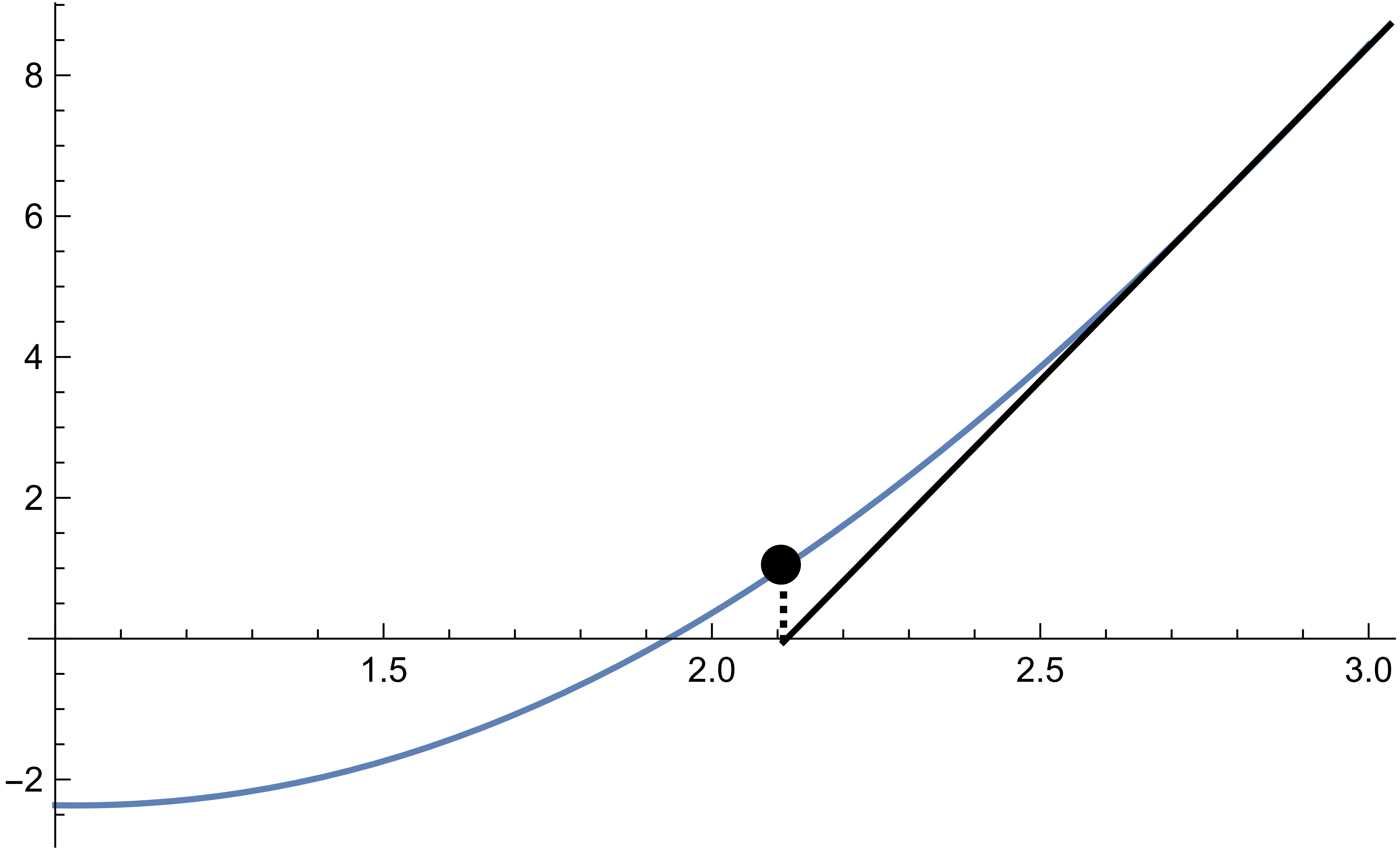
- Move to that position, and repeat..

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

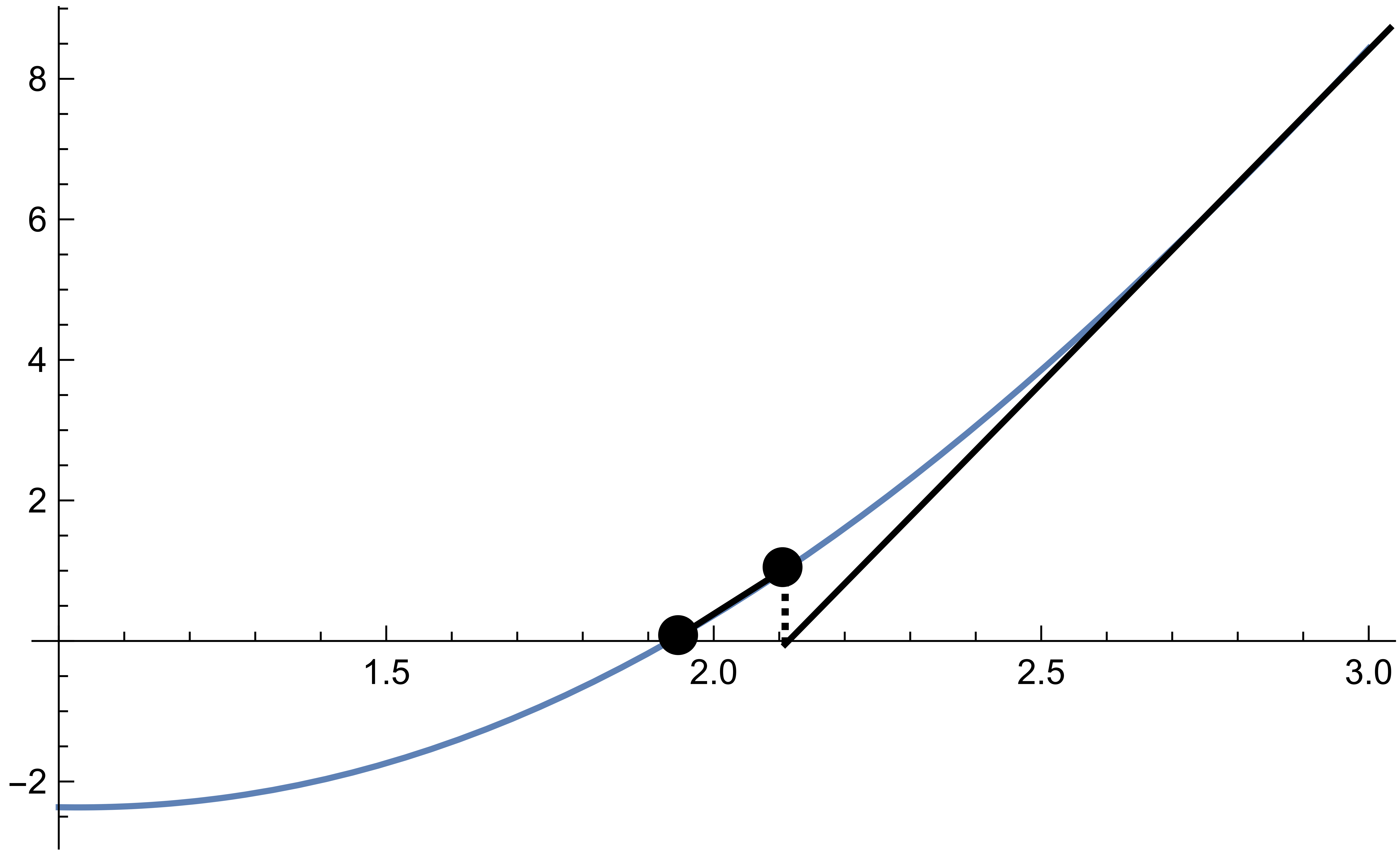
Visualization of Newton's method



Visualization of Newton's method



Visualization of Newton's method



Convergence of Newton's method

$$f(x) := x^2 - 4\sin x$$

Newton's method requires the derivative of f. Here,

$$f'(x) = 2x - 4\cos x$$

Convergence of Newton's method

x	$f(x)$	$f'(x)$	h
3.000000	8.435520	9.959970	−0.846942
2.153058	1.294772	6.505771	−0.199019
<u>1.9</u>54039	0.108438	5.403795	−0.020067
<u>1.933</u>972	0.001152	5.288919	−0.000218
<u>1.933754</u>	0.000000	5.287670	0.000000

(5 iterations for ~7 digits)

[Heath]

Convergence of Newton's method

x	$f(x)$	$f'(x)$	h
3.000000	8.435520	9.959970	−0.846942
2.153058	1.294772	6.505771	−0.199019
1.954039	0.108438	5.403795	−0.020067
1.933972	0.001152	5.288919	−0.000218
1.933754	0.000000	5.287670	0.000000

(5 iterations for ~7 digits)

[Heath]

The method has a quadratic order of convergence, meaning that the number of valid digits approximately **doubles** per iteration.

1D Root Finding: Pros/Cons

Property	Bisection	Newton's method

1D Root Finding: Pros/Cons

Property	Bisection	Newton's method
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)

1D Root Finding: Pros/Cons

Property	Bisection	Newton's method
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)
Reliability	😊 Always works	😞 Divergence, multiple roots, ...

1D Root Finding: Pros/Cons

Property	Bisection	Newton's method
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)
Reliability	😊 Always works	😞 Divergence, multiple roots, ...
Requirements	😐 Continuity, bracket	😐 Derivative

1D Root Finding: Pros/Cons

Property	Bisection	Newton's method
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)
Reliability	😊 Always works	😞 Divergence, multiple roots, …
Requirements	😐 Continuity, bracket	😐 Derivative

Can combine both: Newton-Bisection

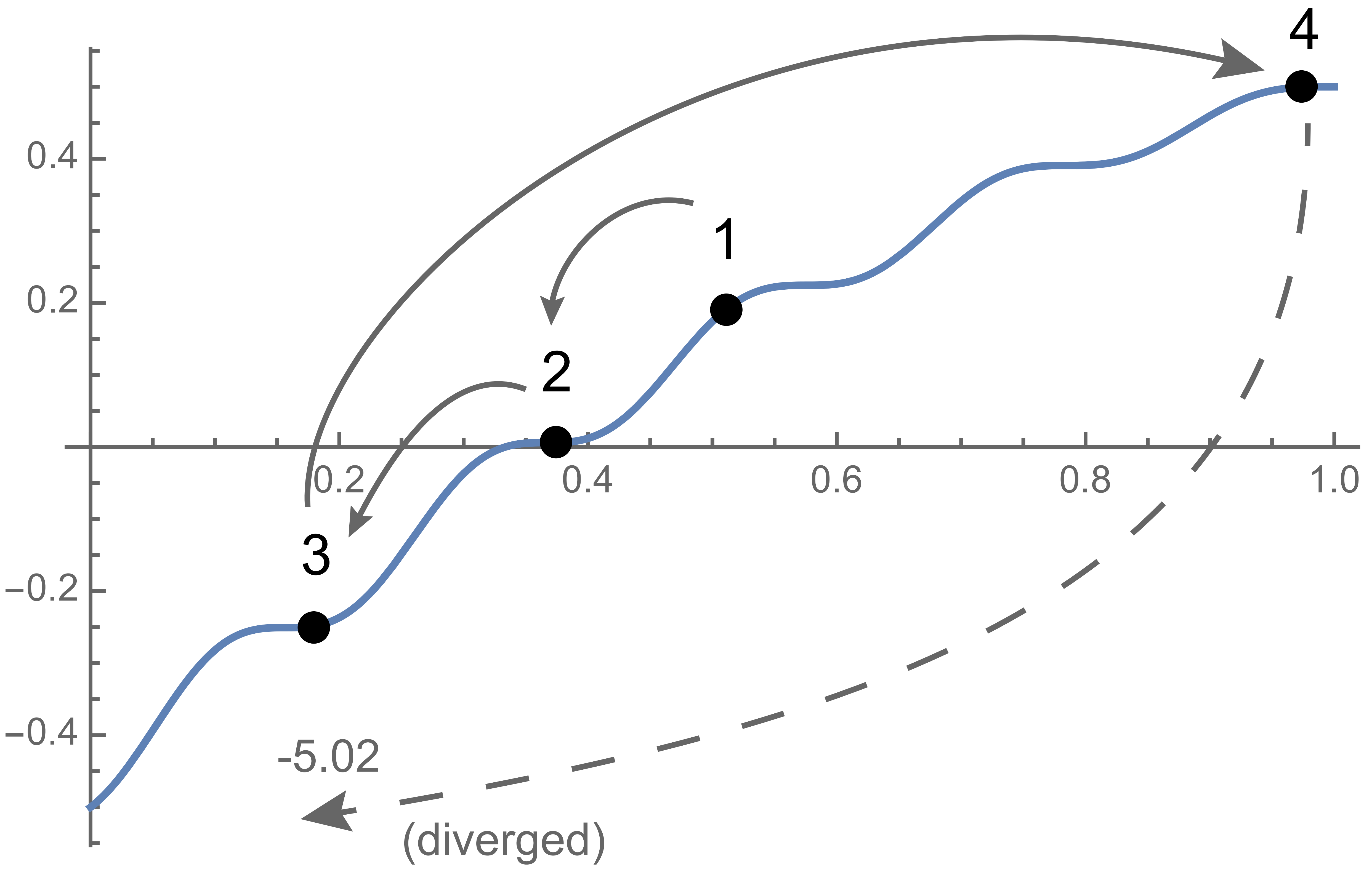
1D Root Finding: Pros/Cons

Property	Bisection	Newton's method
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)
Reliability	😊 Always works	😞 Divergence, multiple roots, ...
Requirements	😐 Continuity, bracket	😐 Derivative

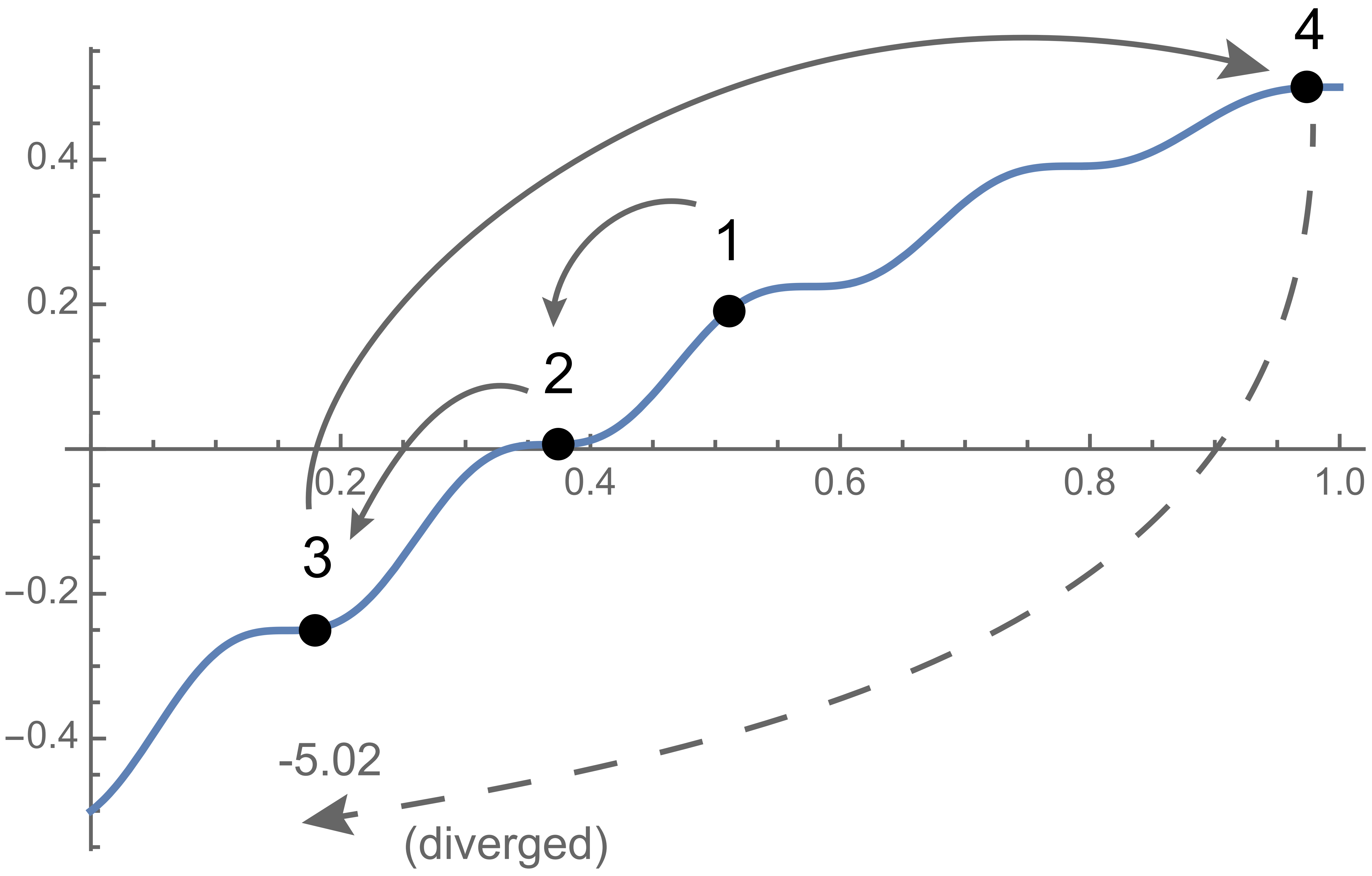
Can combine both: Newton-Bisection

Speed	**Reliability**	**Requirements**
😊 Extremely fast	😊 Always works	😬 Continuity, bracket, derivatives

Failure cases of Newton's Method

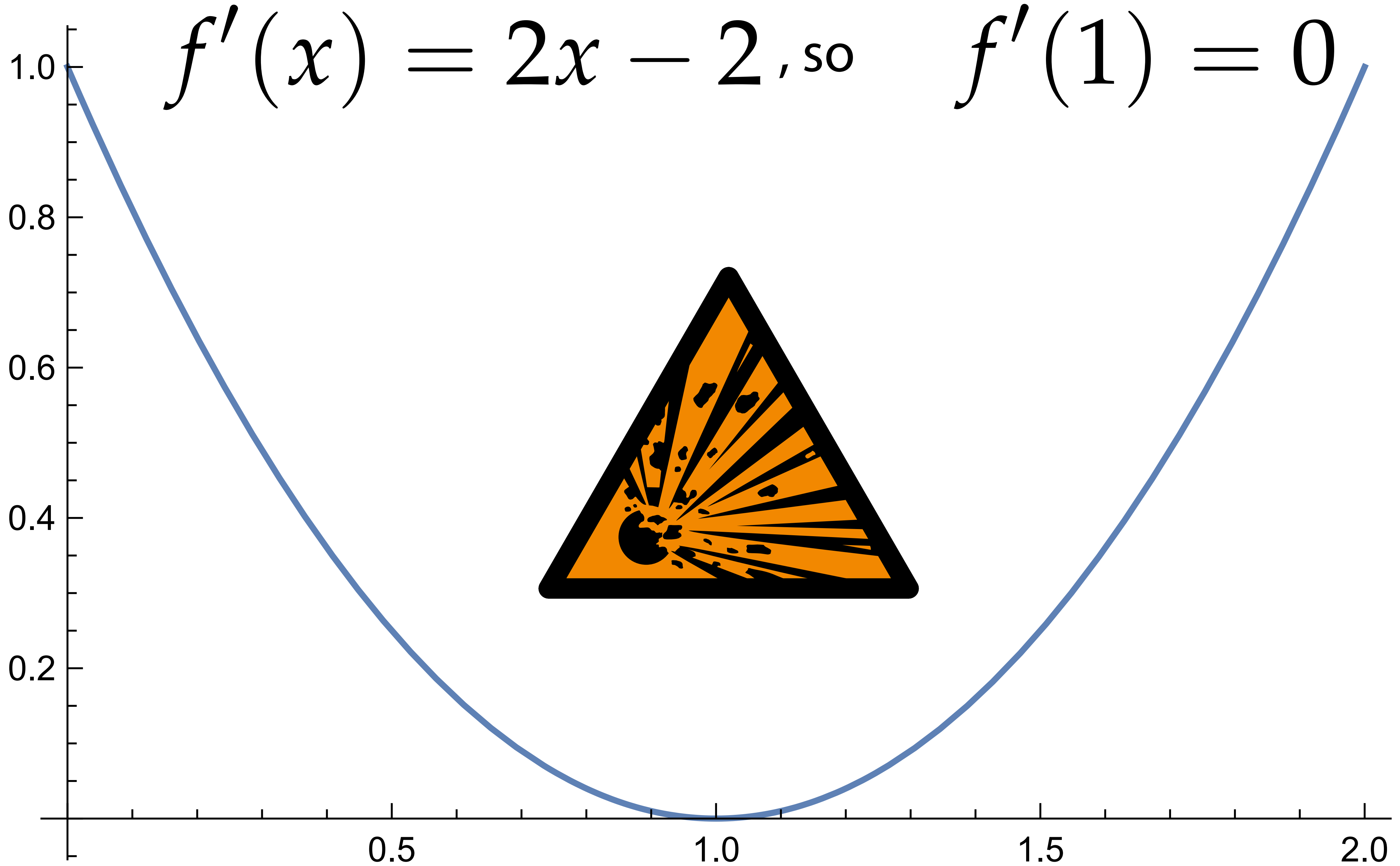


Failure cases of Newton's Method

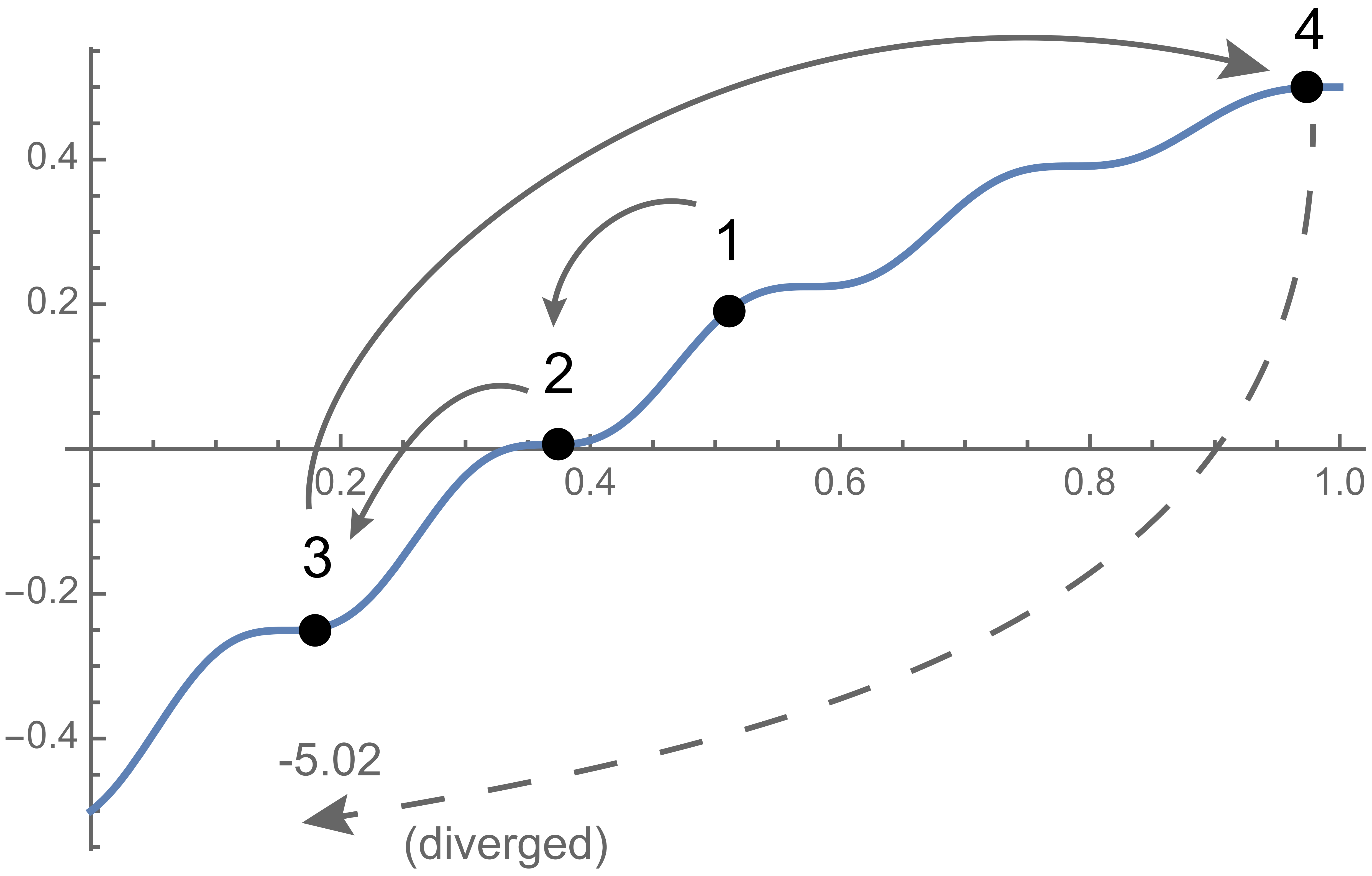


$$f(x) = (x-1)^2$$

$f'(x) = 2x - 2$, so $f'(1) = 0$

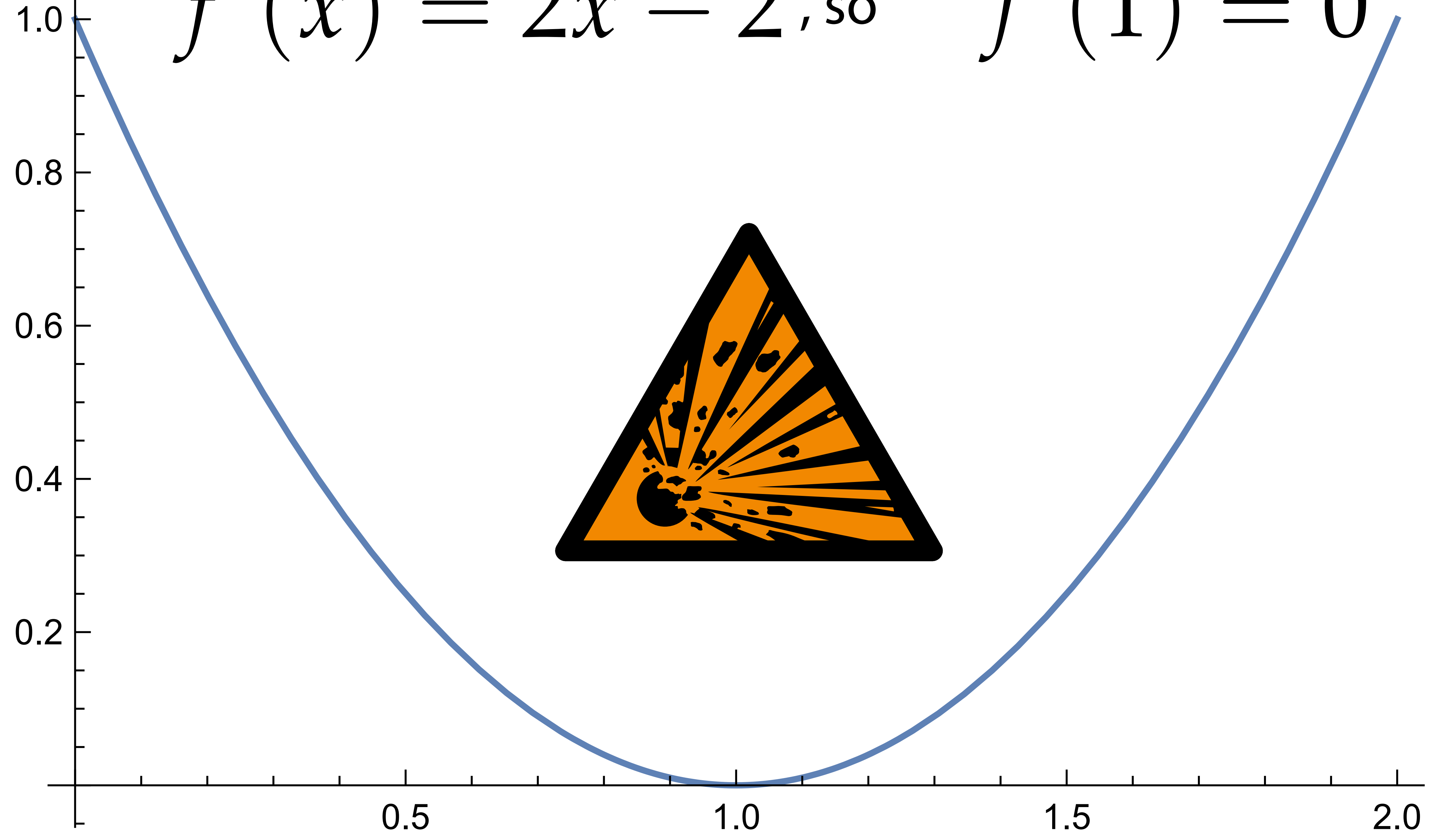


Failure cases of Newton's Method



$$f(x) = (x-1)^2$$

$f'(x) = 2x - 2$, so $f'(1) = 0$



In theory: division by zero at $x = 1$.

In practice: slow convergence.

N dimensions

Multivariate root finding

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

Find $\mathbf{x}$ so that $f(\mathbf{x}) = \mathbf{0}$.

High dimensional spaces

- Derivative are crucial especially in **higher dimensions**.
  - In 1-D, can move in two directions
  - in N-D can move in 2^N "diagonal" directions alone. That's just *too many* to check.
  - The gradient points into the direction of ascent and maps the behavior of the function locally.
  - Foundation of all breakthroughts in ML in the last years. You *cannot* train a neural network without gradients.

MidJourney: A sign post with many different hikes in Switzerland.

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

1D case: $f(x+h) \approx f(x) + f'(x)h = 0$

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

1D case: $f(x+h) \approx f(x) + f'(x)h = 0$

N-D case: $\mathbf{f}(\mathbf{x}+\mathbf{h}) \approx \mathbf{f}(\mathbf{x}_{k-1}) + \nabla \mathbf{f}(\mathbf{x}_{k-1})\mathbf{h} = 0$

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

1D case: $f(x+h) \approx f(x) + f'(x)h = 0$

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

N-D case: $\mathbf{f}(\mathbf{x}+\mathbf{h}) \approx \mathbf{f}(\mathbf{x}_{k-1}) + \nabla\mathbf{f}(\mathbf{x}_{k-1})\mathbf{h} = 0$

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

1D case: $f(x+h) \approx f(x) + f'(x)h = 0$

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

N-D case: $\mathbf{f}(\mathbf{x}+\mathbf{h}) \approx \mathbf{f}(\mathbf{x}_{k-1}) + \nabla\mathbf{f}(\mathbf{x}_{k-1})\mathbf{h} = 0$

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla\mathbf{f}(\mathbf{x}_{k-1})\right]^{-1}\mathbf{f}(\mathbf{x}_{k-1})$$

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..
- No simple+safe hybrid method (e.g. Newton-Bisection) in N-D.

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..
- No simple+safe hybrid method (e.g. Newton-Bisection) in N-D.
- Need to compute Jacobian & solve linear system:
  requires $O(n^3)$ operations *per iteration!!*

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..
- No simple+safe hybrid method (e.g. Newton-Bisection) in N-D.
- Need to compute Jacobian & solve linear system:
  requires $O(n^3)$ operations *per iteration!!*
  - Linear system solve could fail (ill-conditioned/singular. More dimensions in which things can go wrong..)

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

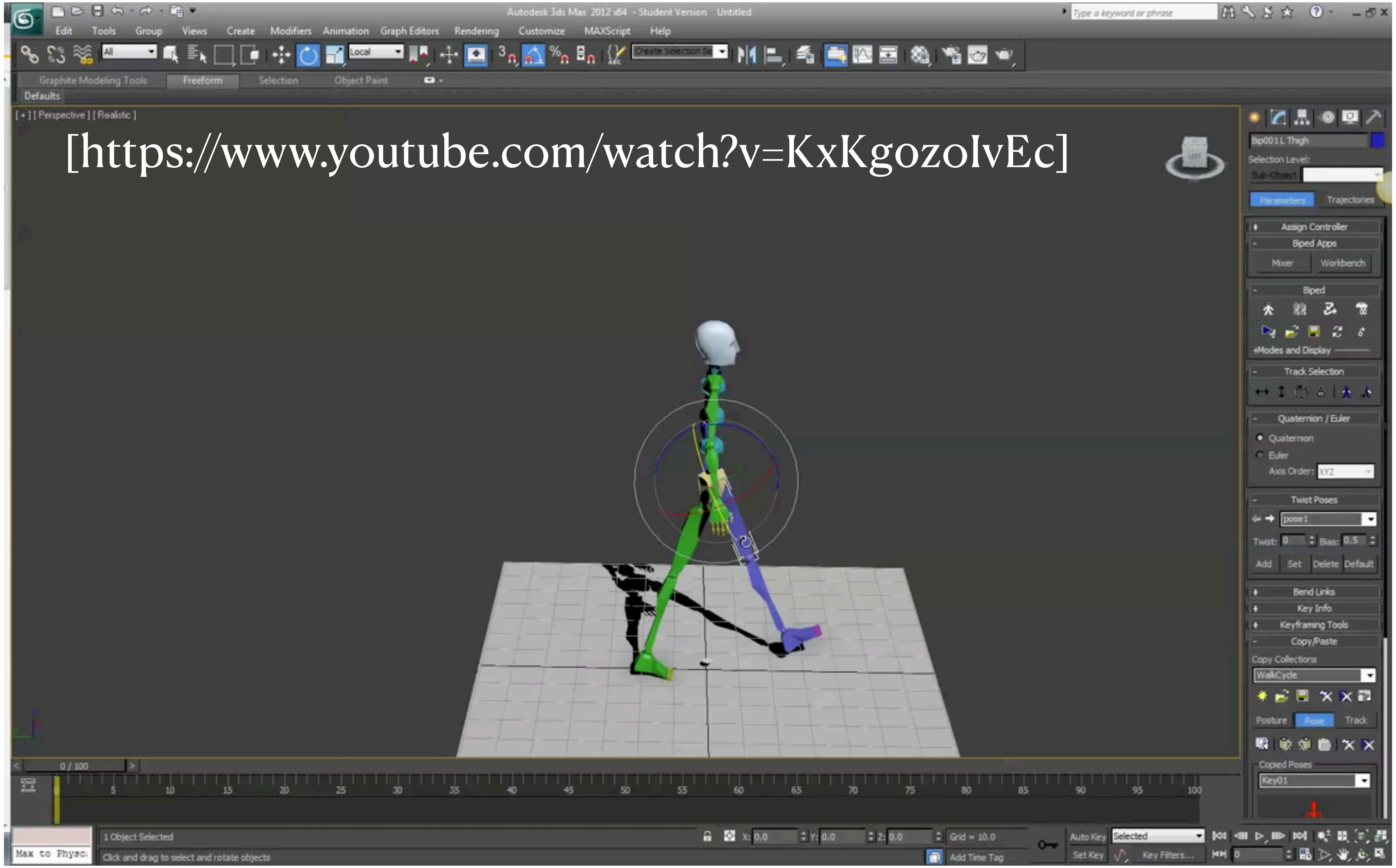
Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..
- No simple+safe hybrid method (e.g. Newton-Bisection) in N-D.
- Need to compute Jacobian & solve linear system: requires $O(n^3)$ operations *per iteration!!*
  - Linear system solve could fail (ill-conditioned/singular. More dimensions in which things can go wrong..)
  - Assumes input and output spaces have matching dimension.

Relation to Character Animation

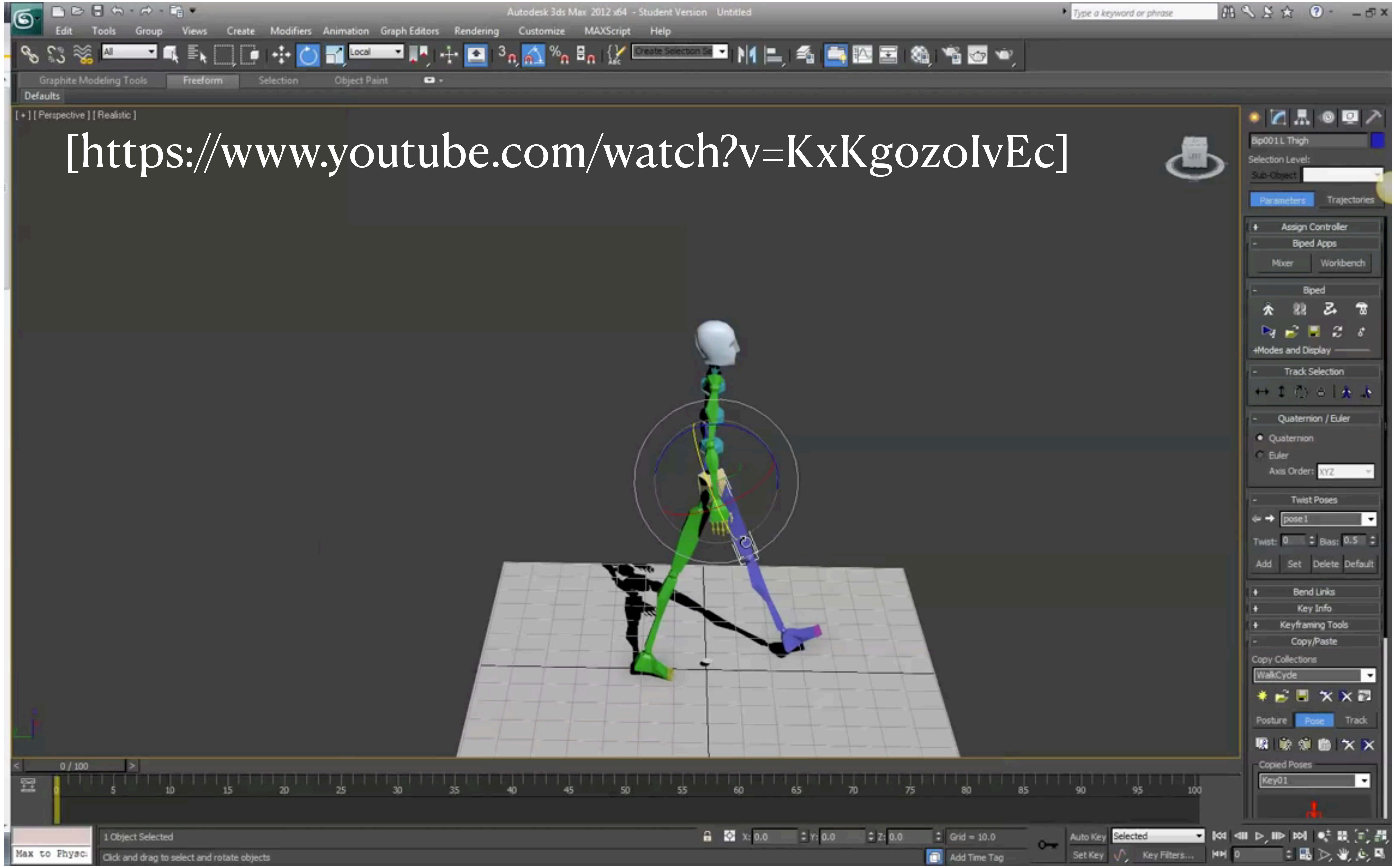
Relation to Character Animation

Connection to Character Animation



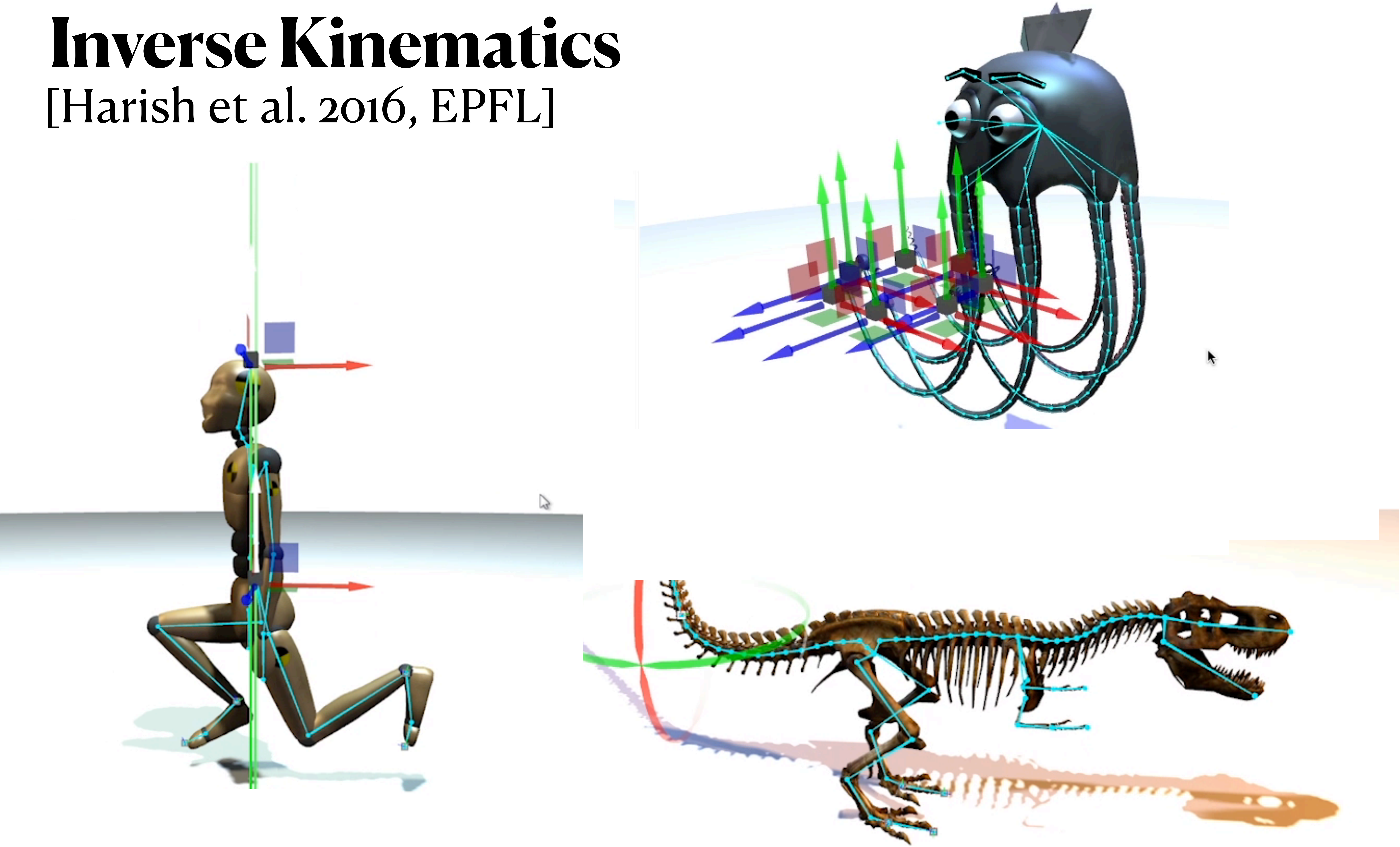
[https://www.youtube.com/watch?v=KxKgozoIvEc]

Connection to Character Animation



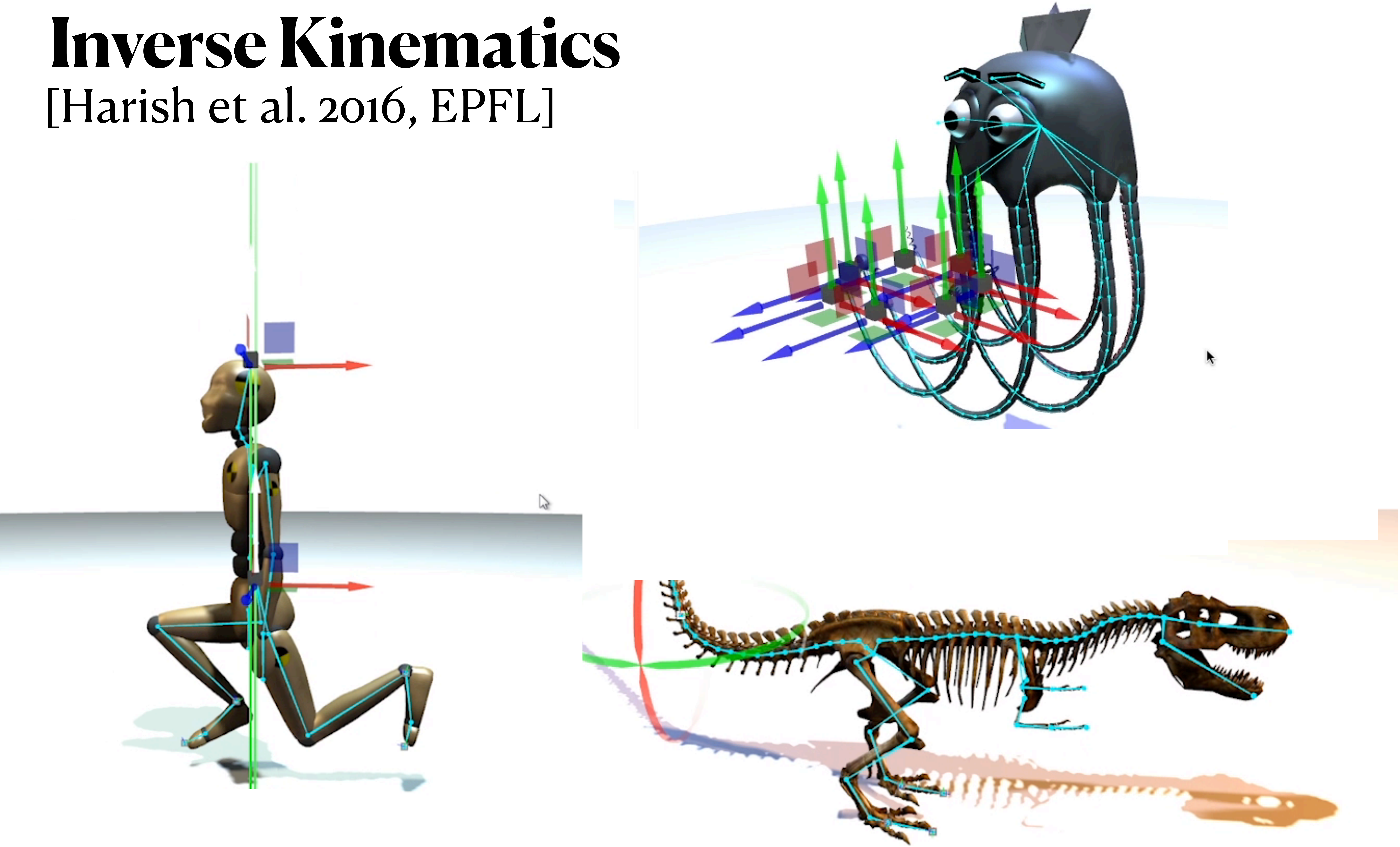
[https://www.youtube.com/watch?v=KxKgozoIvEc]

Inverse Kinematics
[Harish et al. 2016, EPFL]



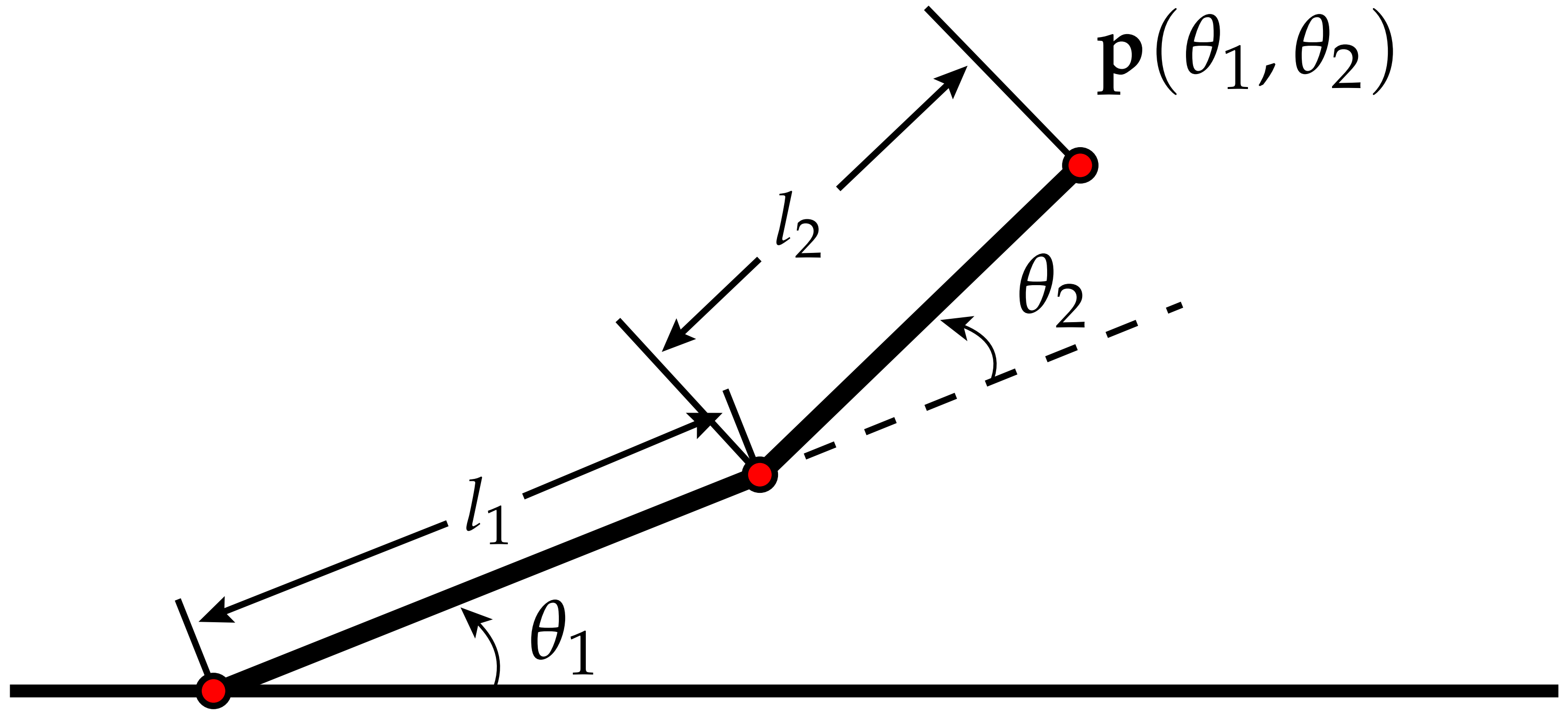
Inverse Kinematics

[Harish et al. 2016, EPFL]



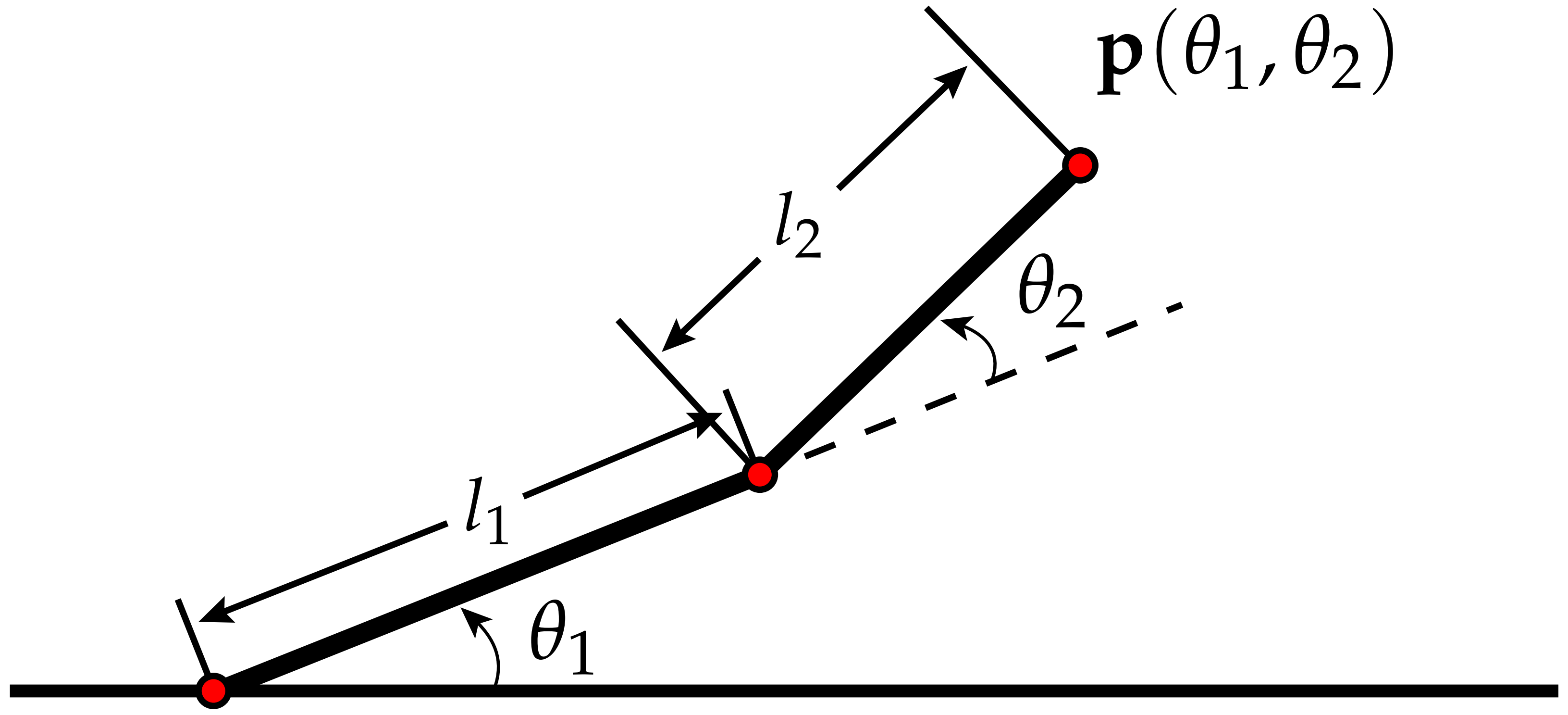
Inverse Kinematics via Newton's method

$$\mathbf{p}(\theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \mathbf{p}_{\text{target}}$$



Inverse Kinematics via Newton's method

$$\mathbf{p}(\theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \mathbf{p}_{\text{target}}$$



Idea: solve with Newton's method

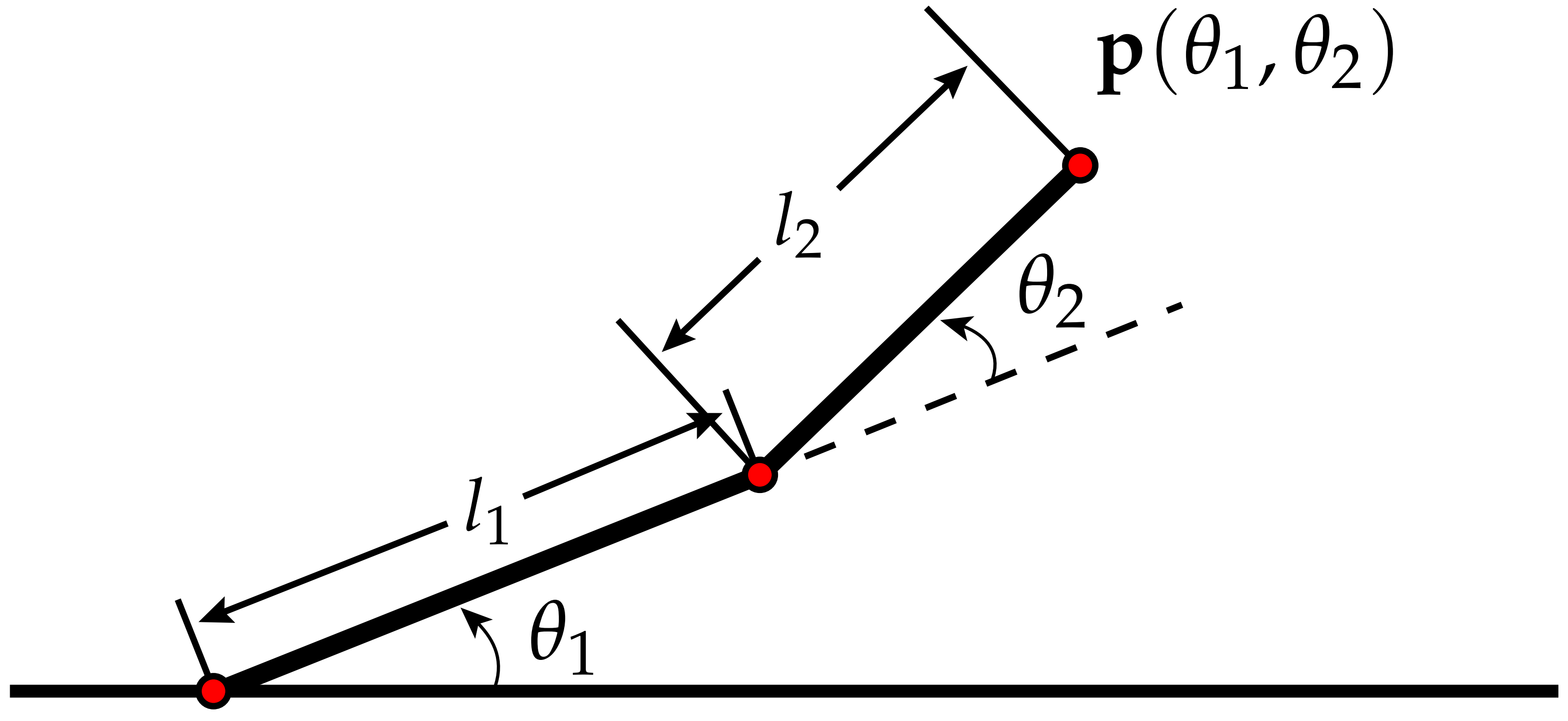
$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \left[\nabla \mathbf{p}(\boldsymbol{\theta}_{k-1})\right]^{-1} \mathbf{p}(\boldsymbol{\theta}_{k-1})$$

Inverse Kinematics via Newton's method

n unknowns

$$\mathbf{p}(\theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \mathbf{p}_{\text{target}}$$

2 equations



Idea: solve with Newton's method

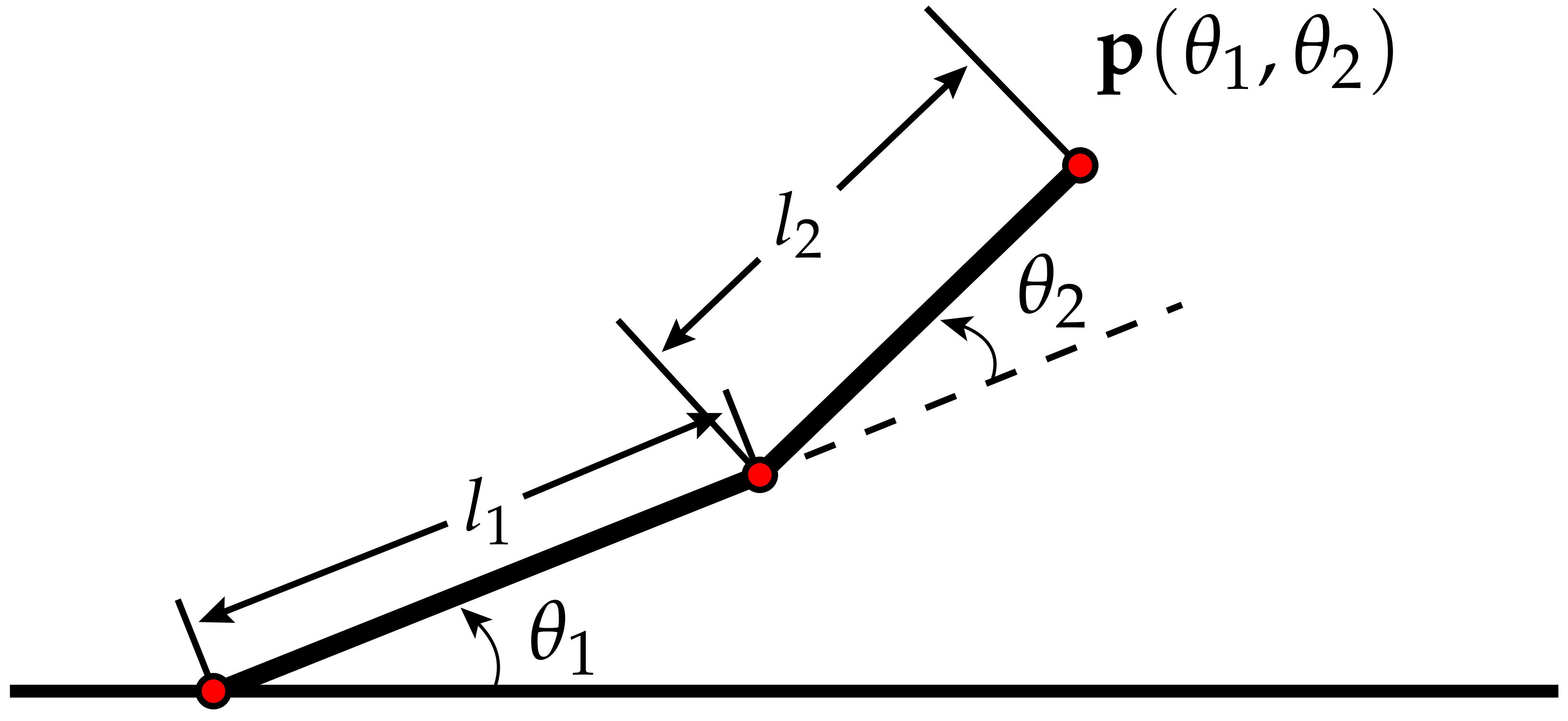
$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - [\nabla \mathbf{p}(\boldsymbol{\theta}_{k-1})]^{-1} \mathbf{p}(\boldsymbol{\theta}_{k-1})$$

Inverse Kinematics via Newton's method

n unknowns

$$\mathbf{p}(\theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \mathbf{p}_{\text{target}}$$

2 equations



Idea: solve with Newton's method

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \left[\nabla \mathbf{p}(\boldsymbol{\theta}_{k-1})\right]^{-1} \mathbf{p}(\boldsymbol{\theta}_{k-1})$$

Idea 2: solve with Newton's method + pseudoinverse

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \left[\nabla \mathbf{p}(\boldsymbol{\theta}_{k-1})\right]^{+} \mathbf{p}(\boldsymbol{\theta}_{k-1})$$

Root finding

Let's start with a simple 1D problem

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

Find x so that $f(x) = 0.$

How many solutions?

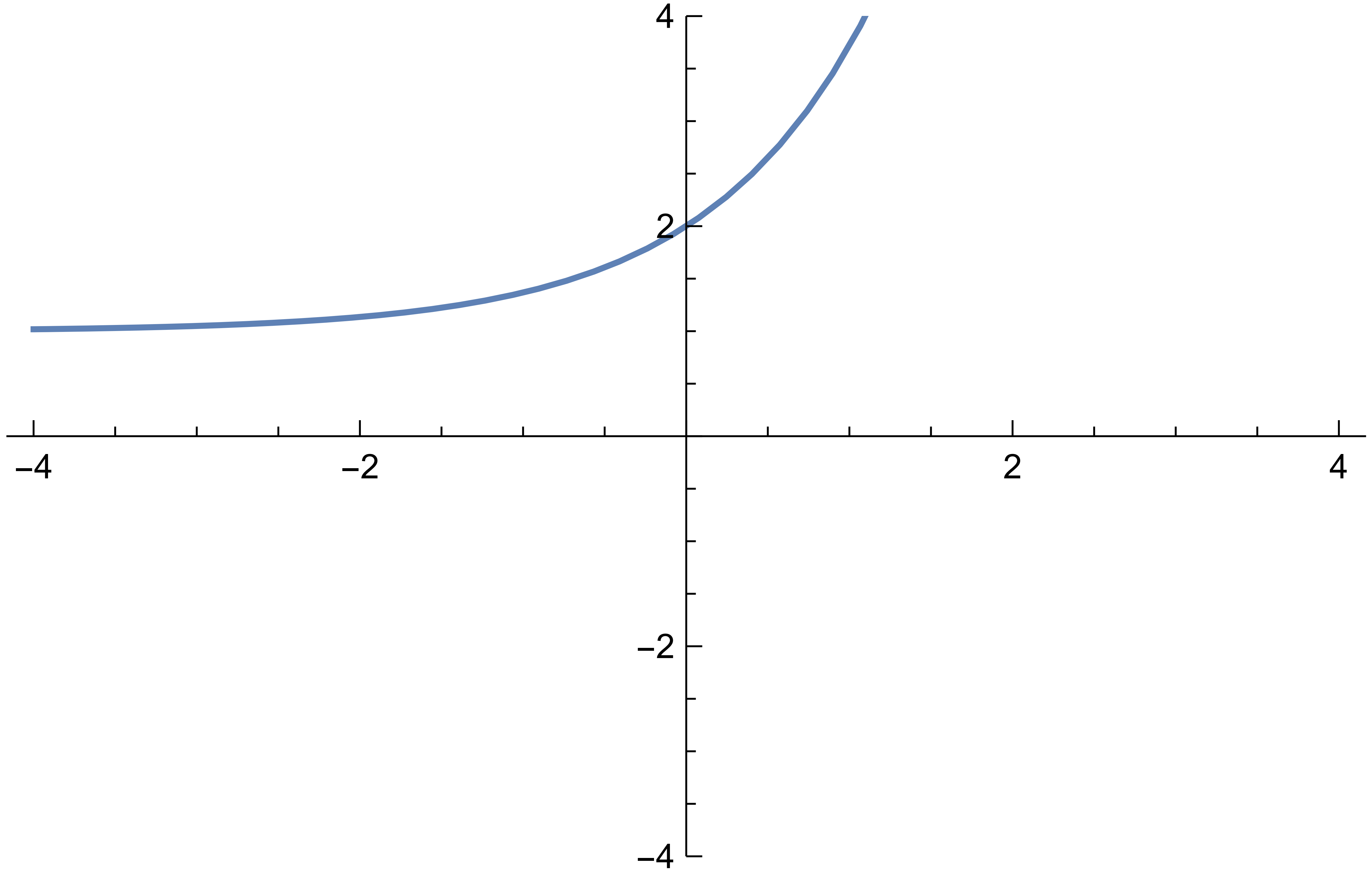
Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

- A few examples from [Heath 2002]:

How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

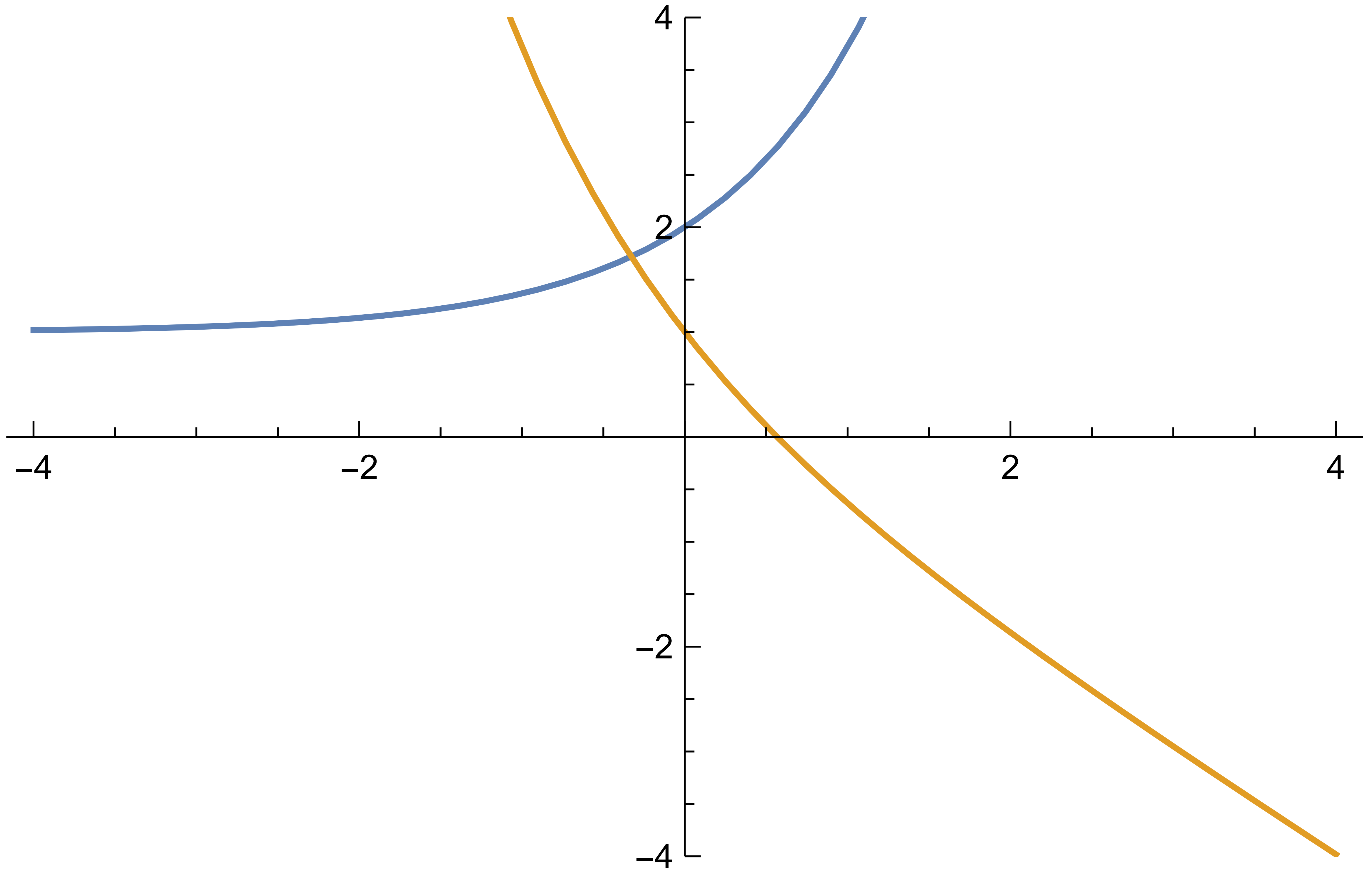
- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$



How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

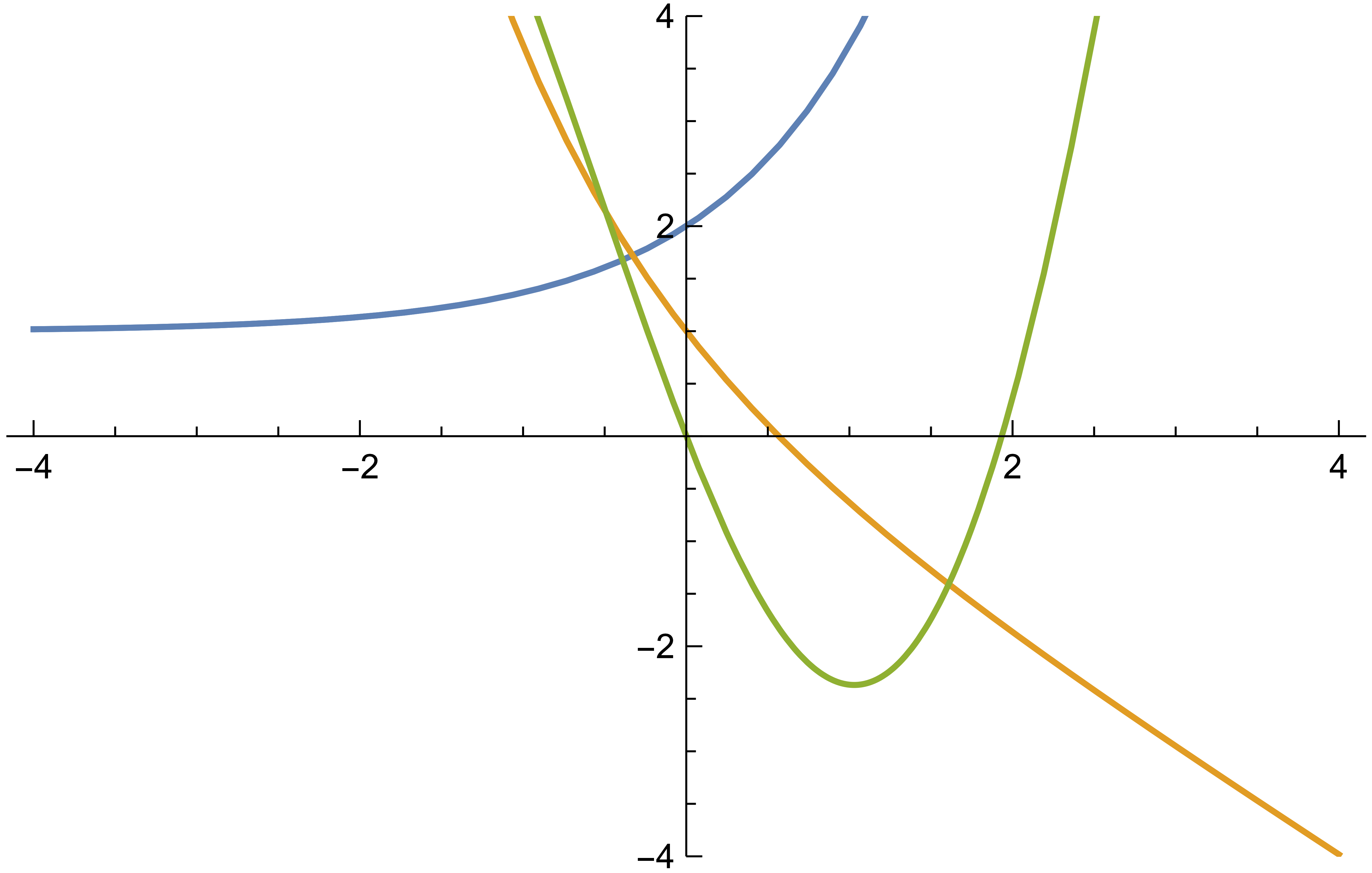
- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$
  - $\exp(-x)$ -x $= 0$



How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

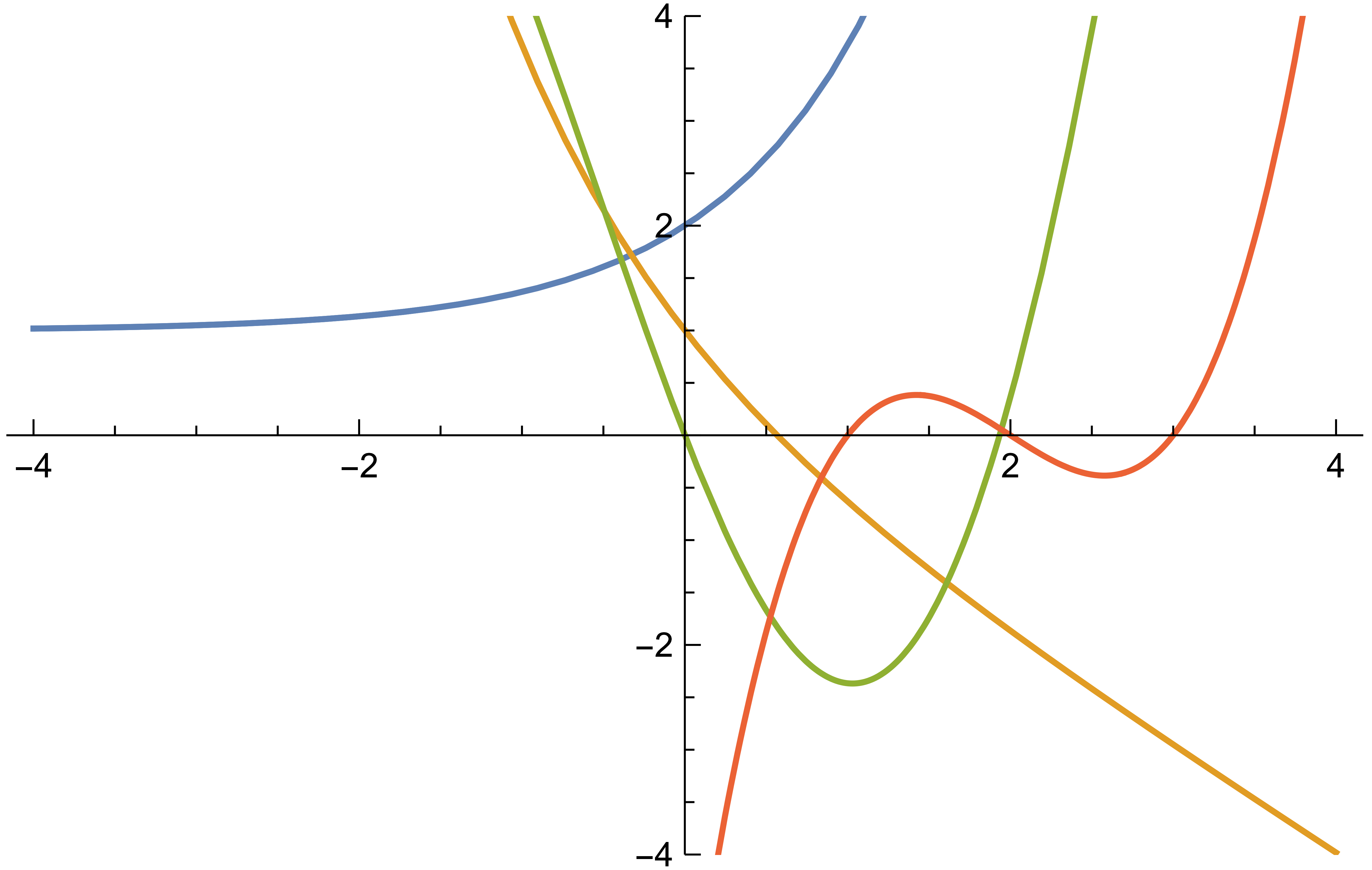
- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$
  - $\exp(-x) - x = 0$
  - $x^2 - 4\sin(x) = 0$



How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

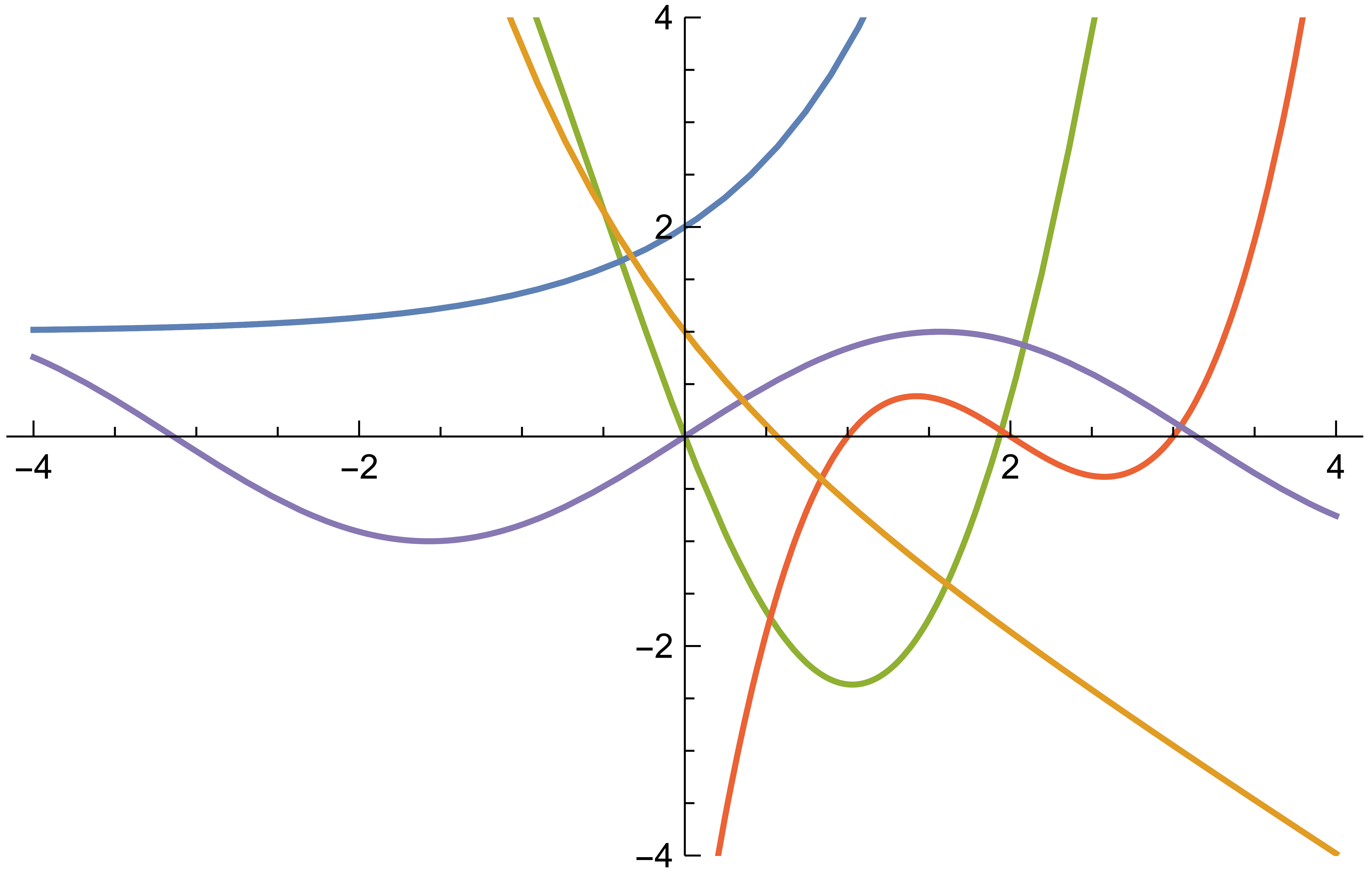
- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$
  - $\exp(-x) - x = 0$
  - $x^2 - 4\sin(x) = 0$
  - $x^3 - 6x^2 + 11x - 6 = 0$



How many solutions?

Linear systems have 0, 1, or ∞ solutions. Anything possible in the nonlinear case

- A few examples from [Heath 2002]:
  - $\exp(x) + 1 = 0$
  - $\exp(-x)$ -x = 0
  - $x^2 - 4\sin(x) = 0$
  - $x^3 - 6x^2 + 11x - 6 = 0$
  - $\sin(x) = 0$



"Evil functions"

All sorts of things could be lurking inside. How are we expected to deal with such functions?

"Evil functions"

All sorts of things could be lurking inside. How are we expected to deal with such functions?

$$f_1(x) := \begin{cases} 1, & x \in \mathbb{Q} \\ -1, & x \in \mathbb{R} \setminus \mathbb{Q} \end{cases}$$

"Evil functions"

All sorts of things could be lurking inside. How are we expected to deal with such functions?

$$f_1(x) := \begin{cases} 1, & x \in \mathbb{Q} \\ -1, & x \in \mathbb{R} \setminus \mathbb{Q} \end{cases}$$

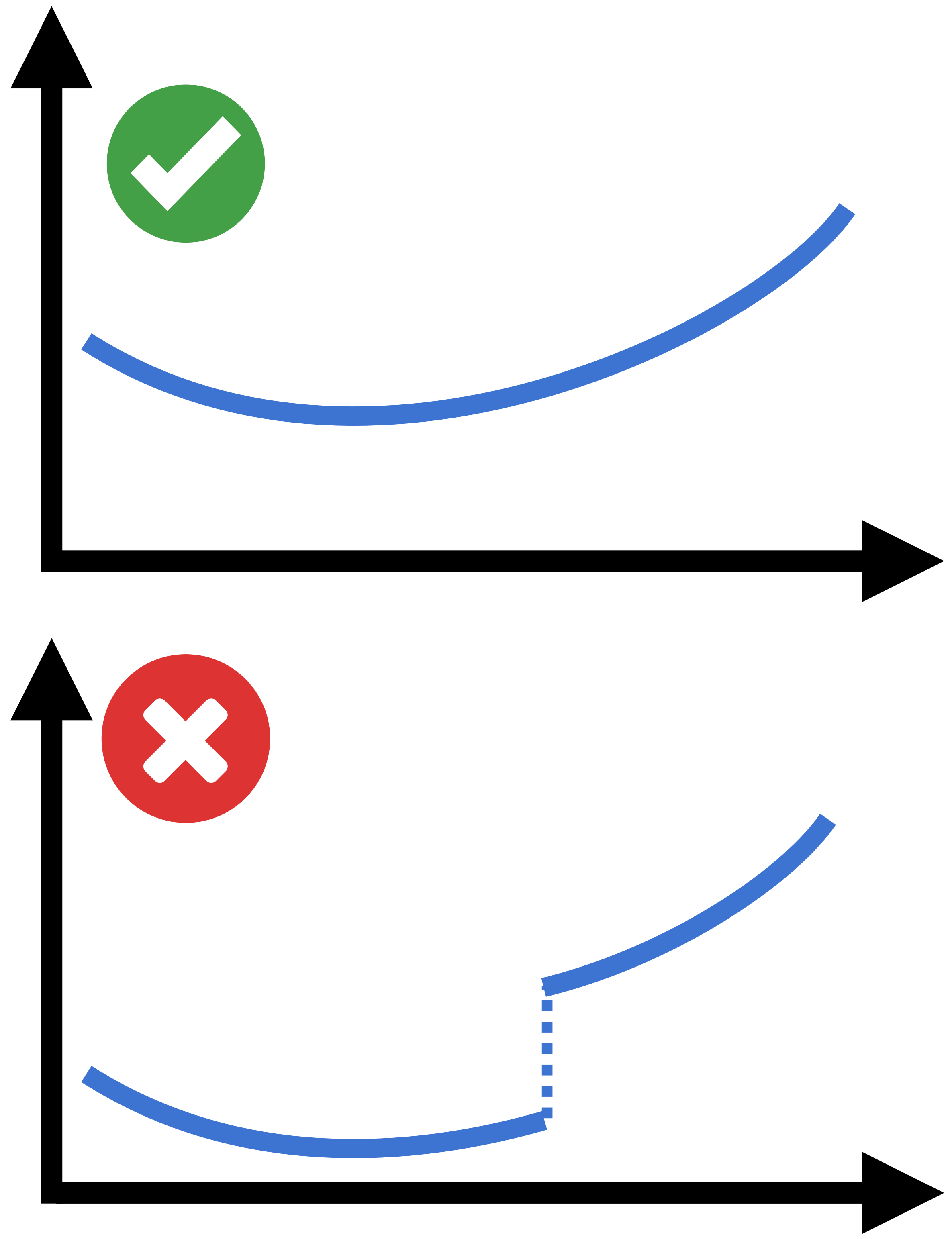
$$f_2(x) := \begin{cases} 1, & x \neq 0.13525634 \\ 0, & x = 0.13525634 \end{cases}$$

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

Common assumptions

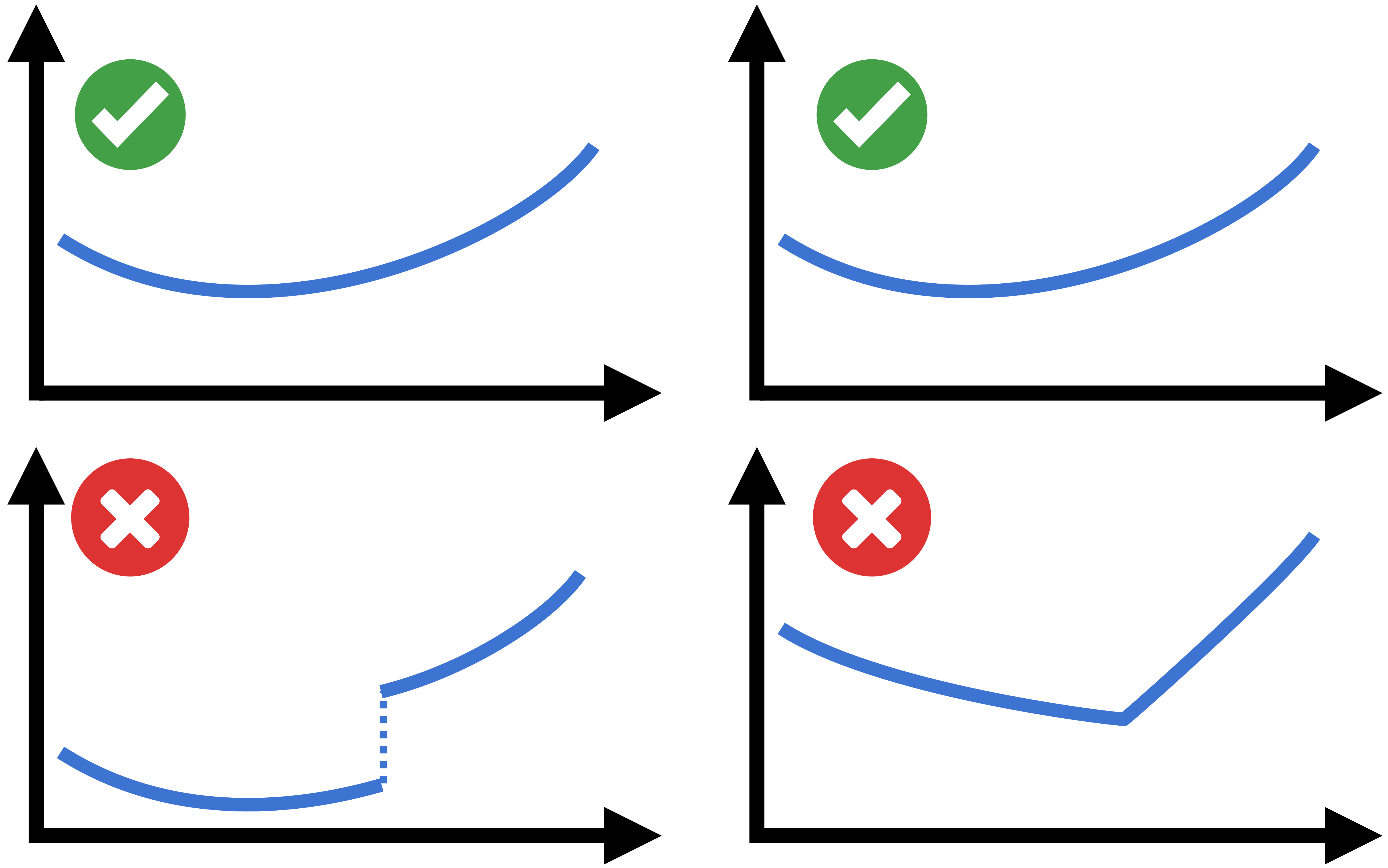
Must assume *something*. This determines how the solution algorithm will work.



Continuity

Common assumptions

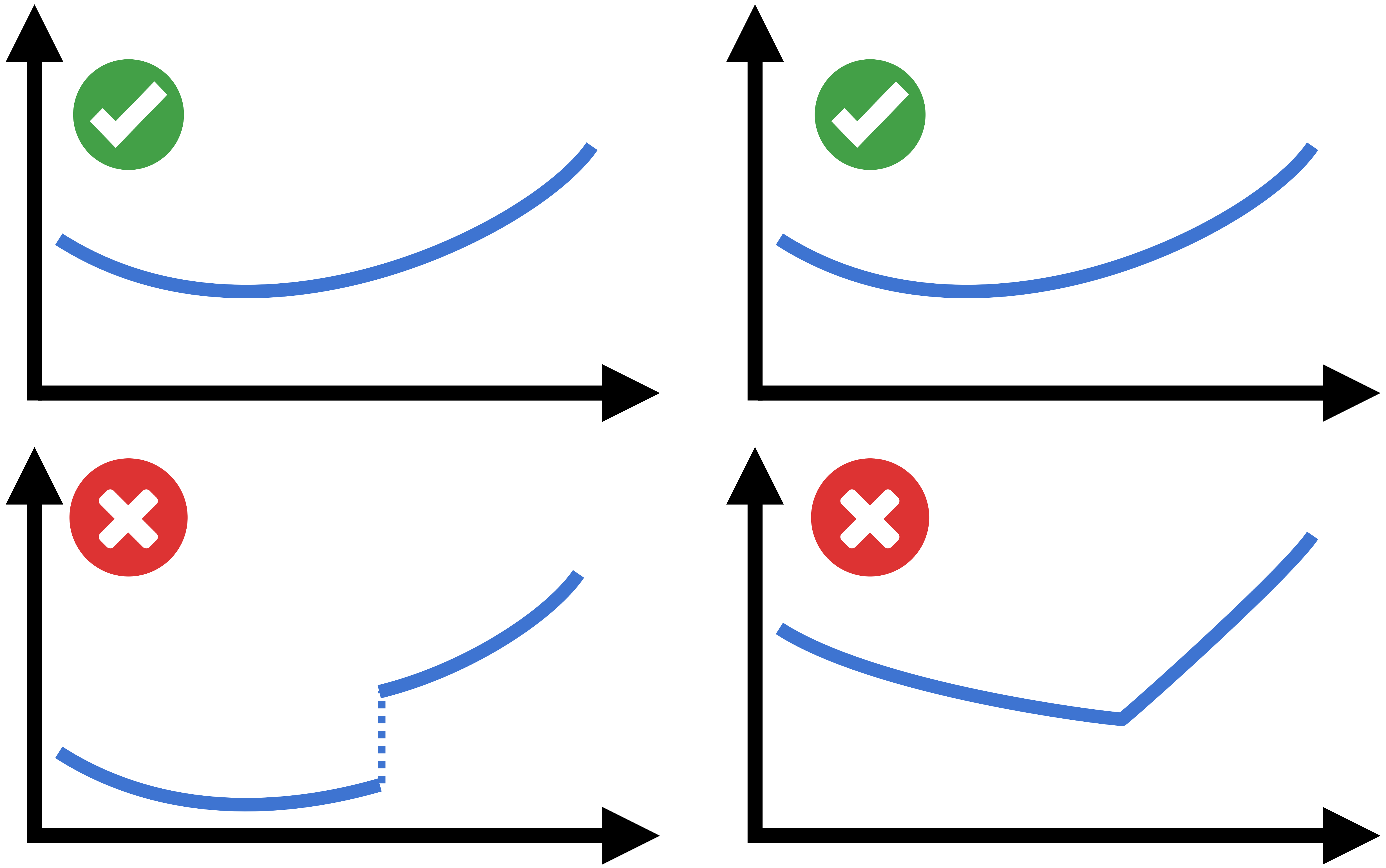
Must assume *something*. This determines how the solution algorithm will work.



Continuity **Differentiability**

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

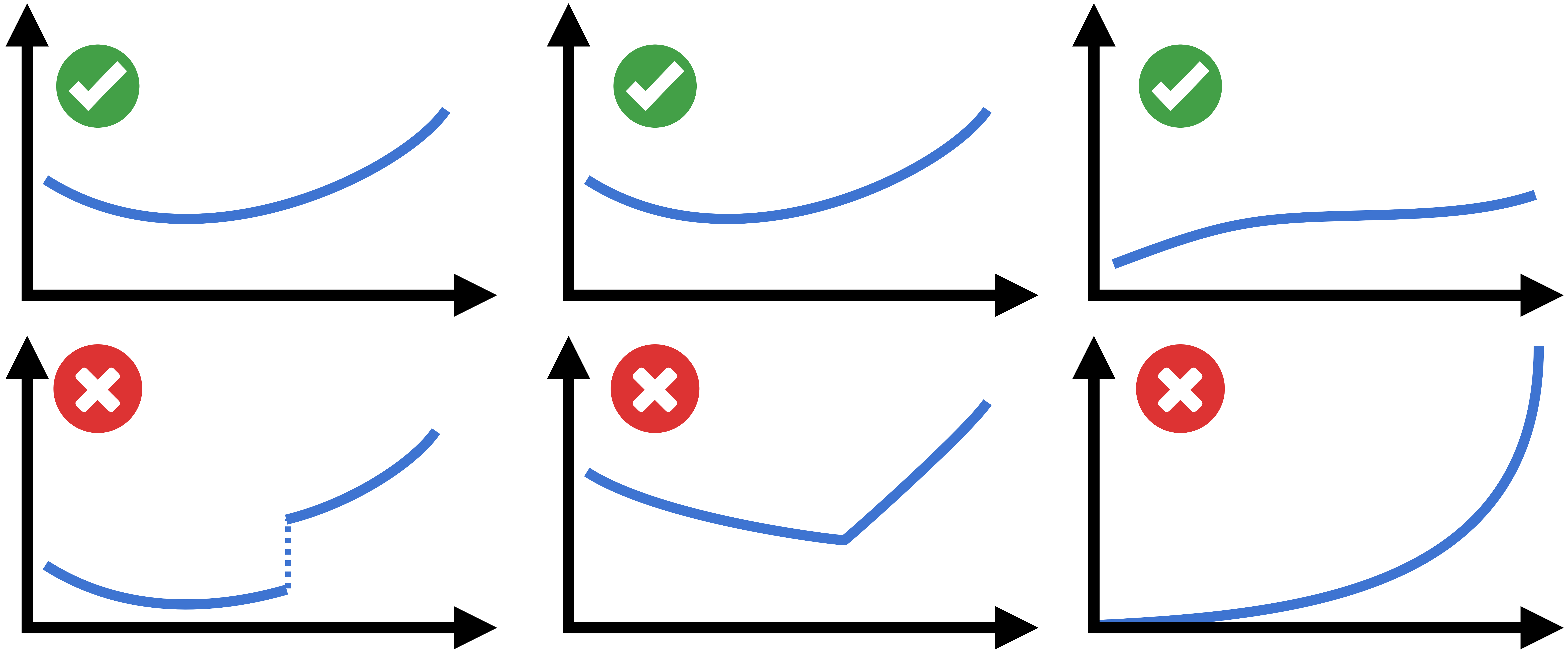


Continuity **Differentiability**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

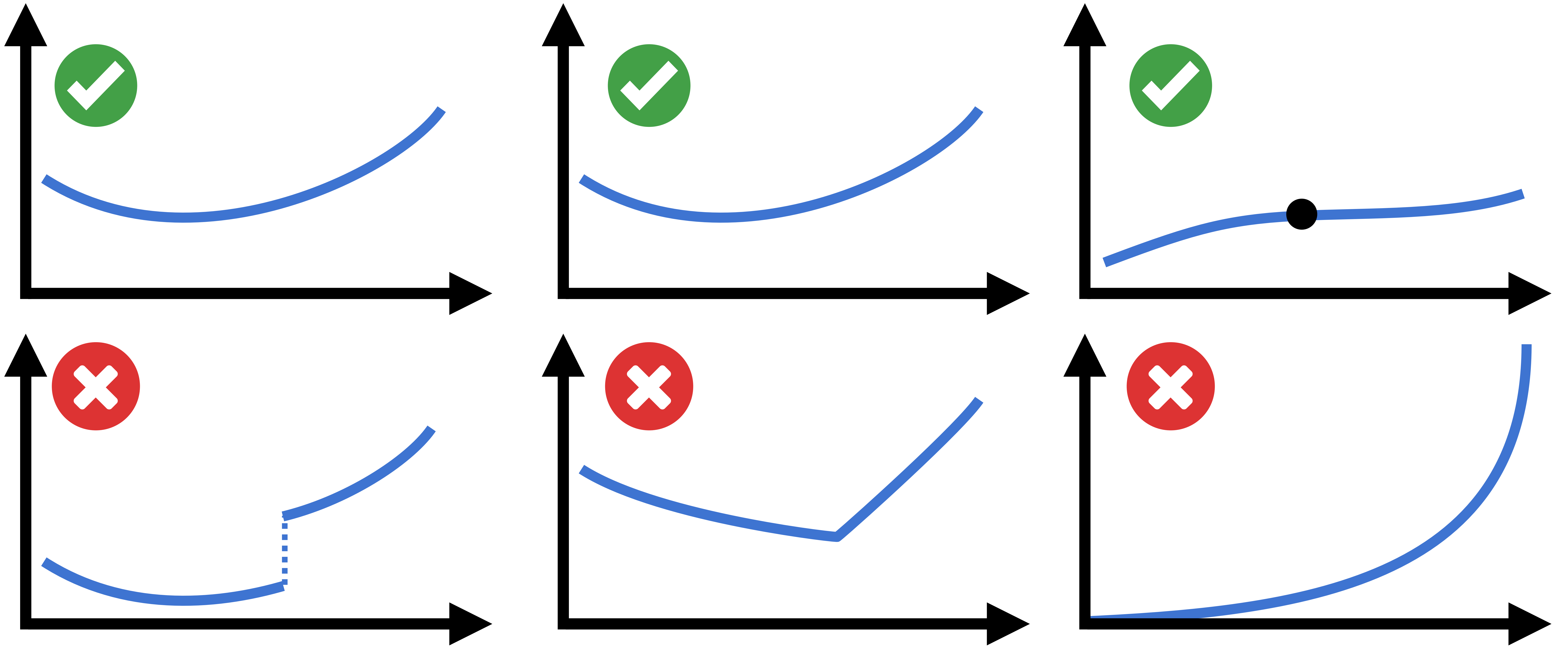


Continuity | Differentiability | *Lipschitz* continuity

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

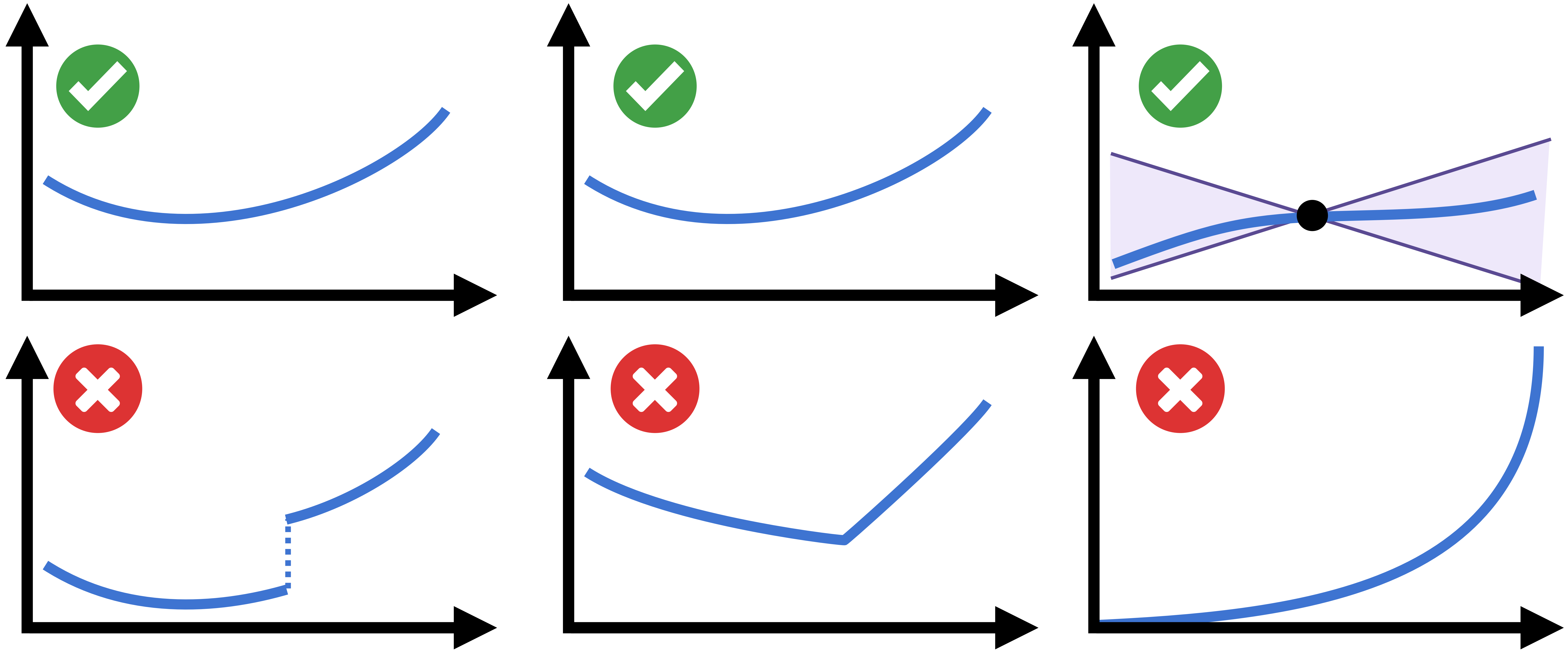


Continuity **Differentiability** ***Lipschitz* continuity**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

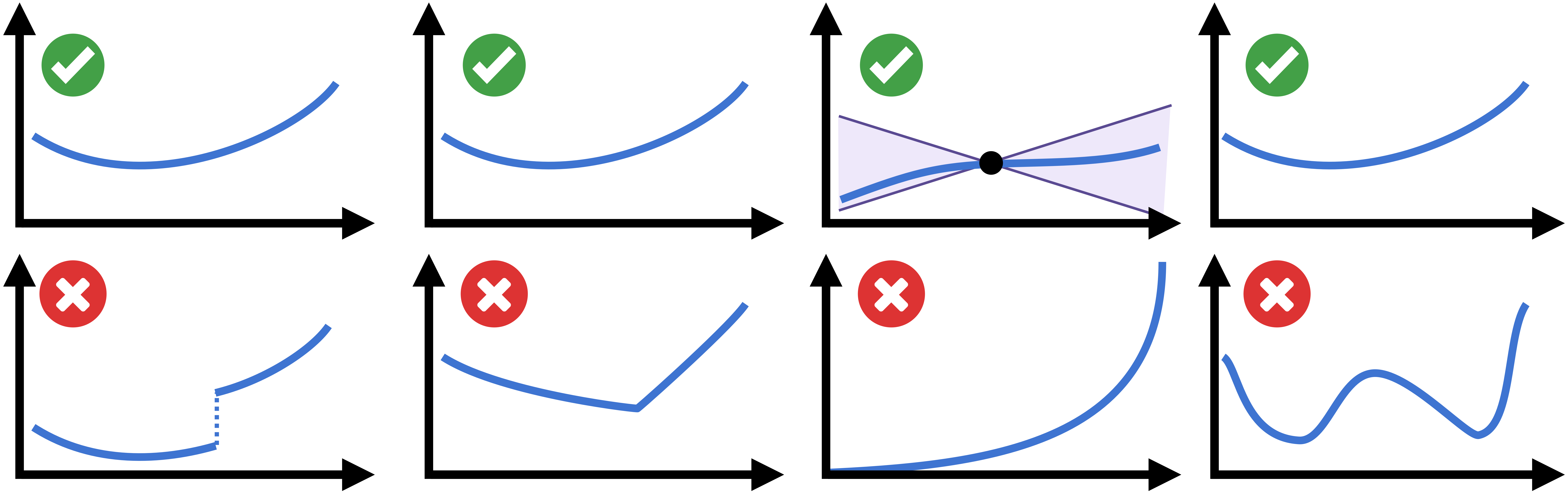


Continuity **Differentiability** ***Lipschitz* continuity**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

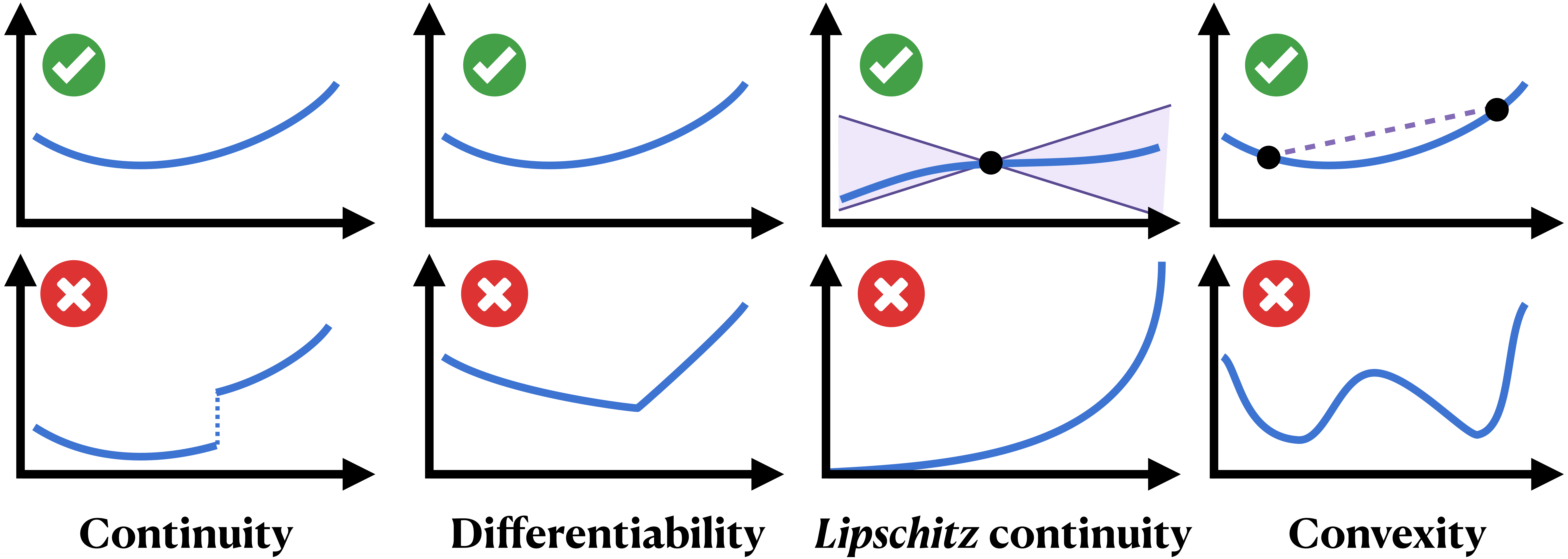
Must assume *something*. This determines how the solution algorithm will work.



Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

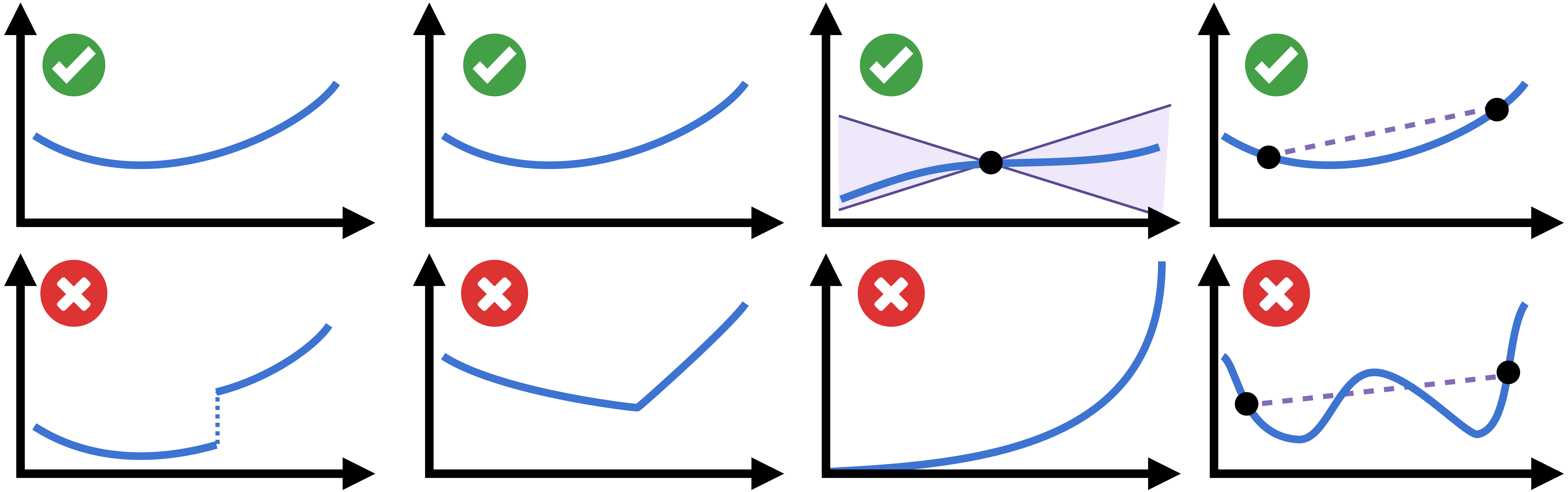
Must assume *something*. This determines how the solution algorithm will work.



Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.

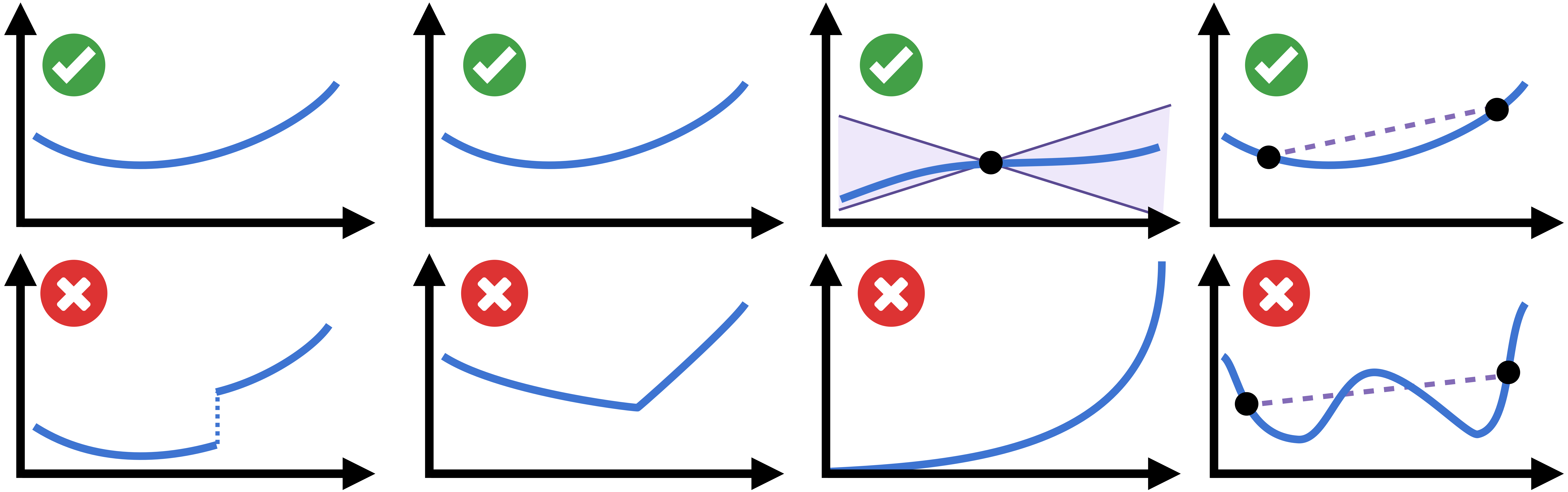


Continuity | Differentiability | *Lipschitz* continuity | Convexity

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

Common assumptions

Must assume *something*. This determines how the solution algorithm will work.



Continuity | **Differentiability** | ***Lipschitz* continuity** | **Convexity**

Actually: kinks usually OK, we mainly need the ability to *evaluate* derivatives.

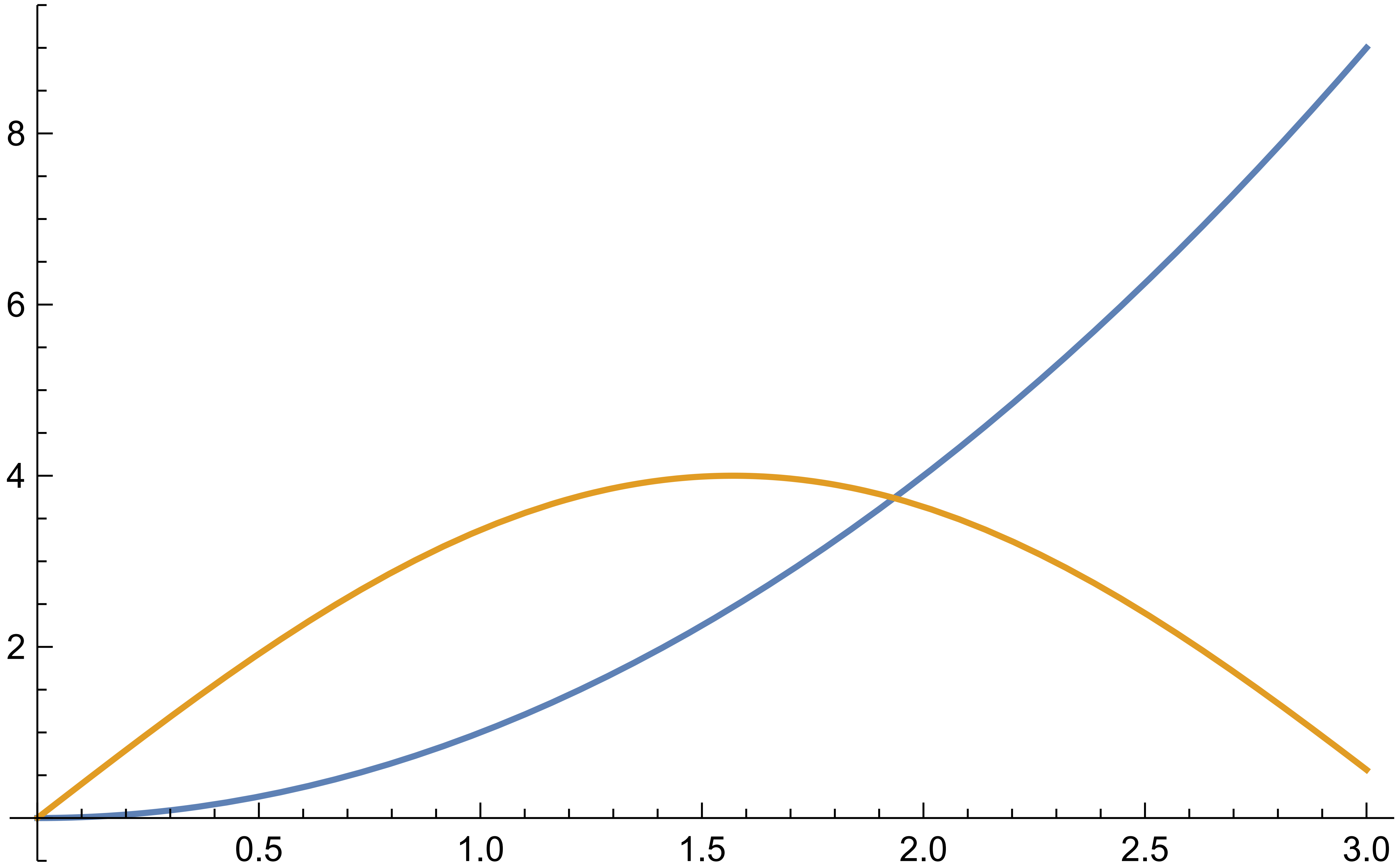
Mostly used for optimization.

A model problem

$$x^2 = 4 \sin x$$

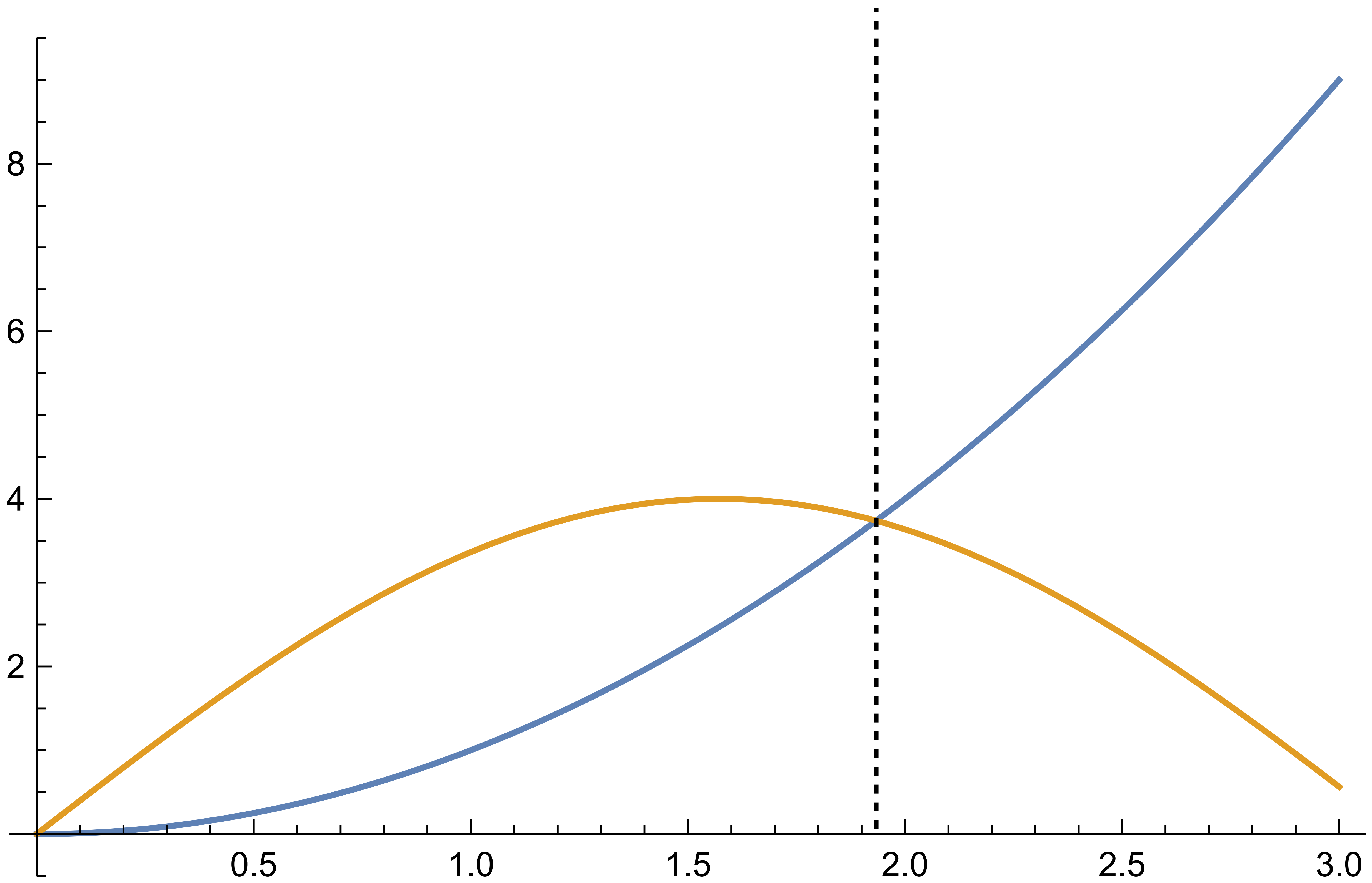
A model problem

$$x^2 = 4 \sin x$$



A model problem

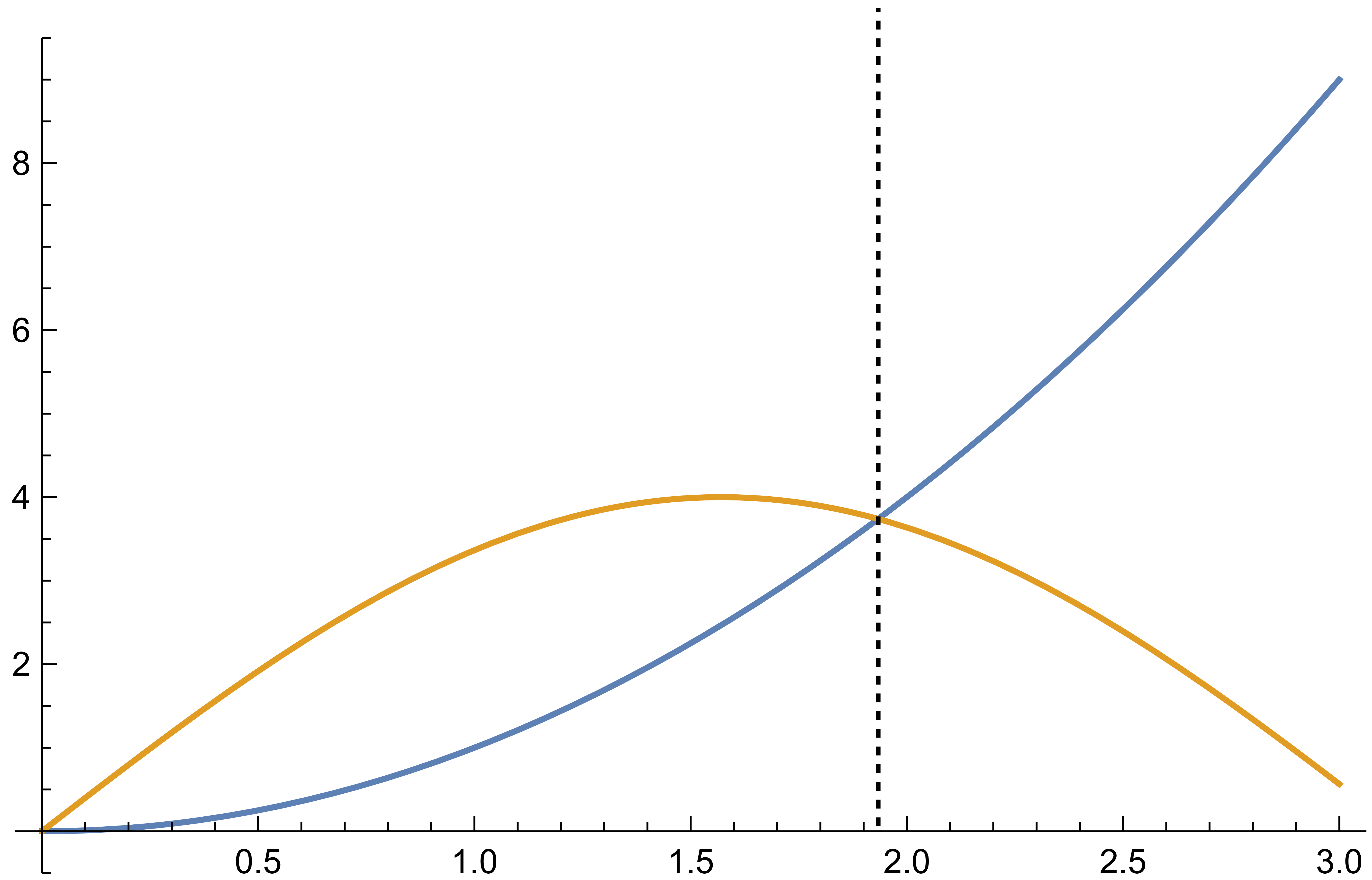
$$x^2 = 4\sin x$$



A model problem

$$x^2 = 4 \sin x$$

Solution is not analytic!



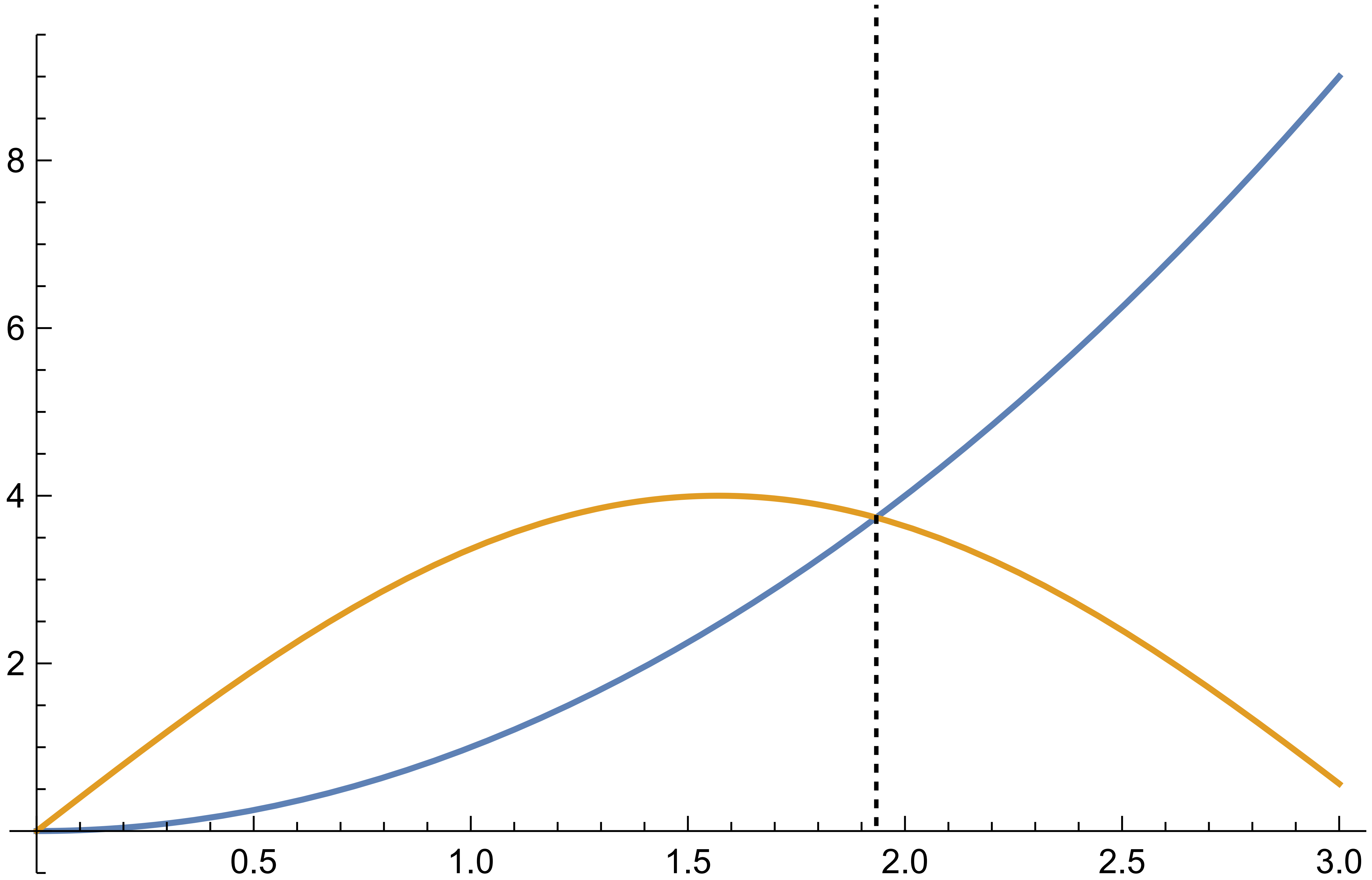
A model problem

Mathematica: `In[1]:= Solve[x^2 == 4 Sin[x], x]`

Solve: This system cannot be solved with the methods available to Solve.

$$x^2 = 4\sin x$$

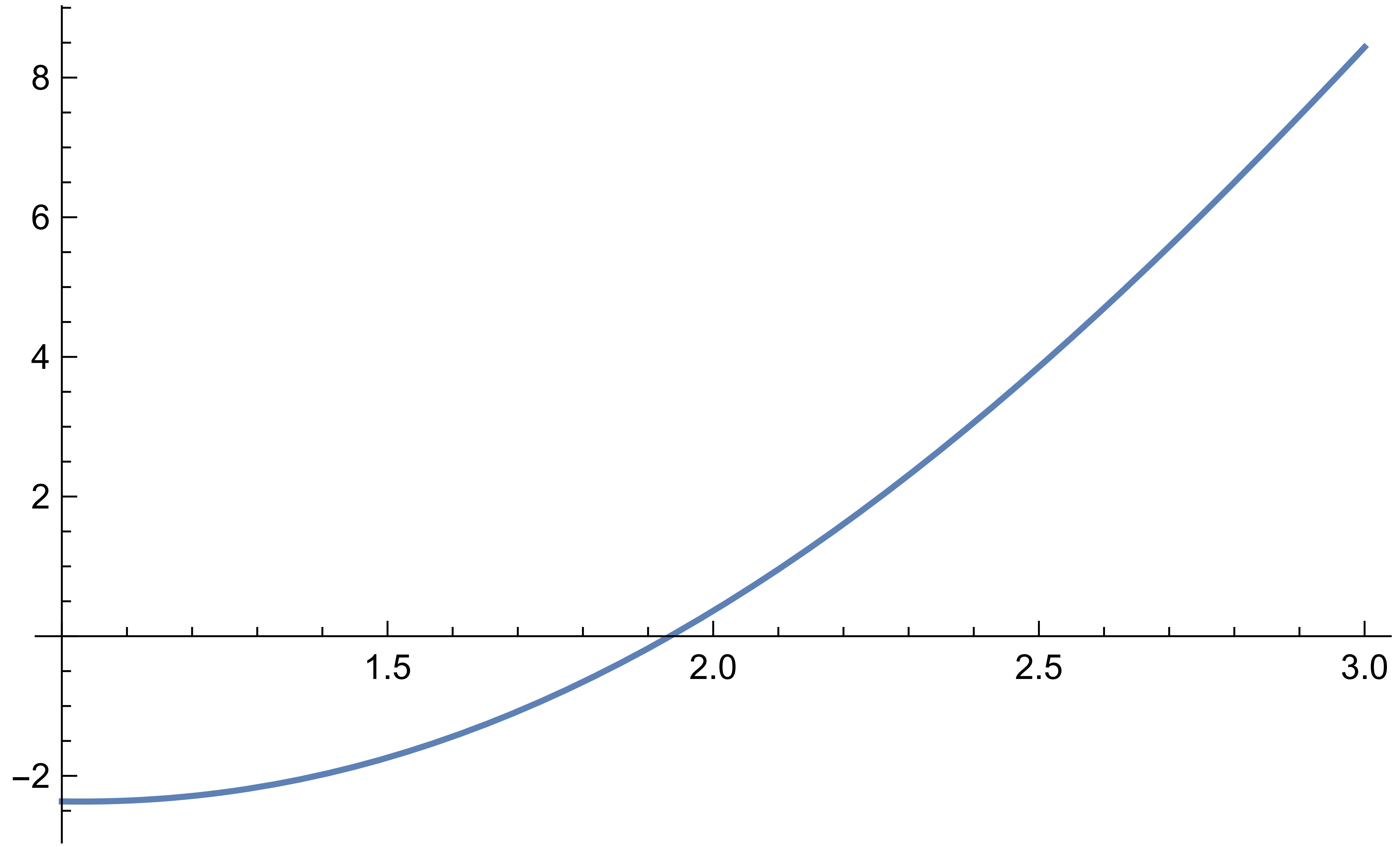
Solution is not analytic!



A model problem

Can more or less read off solution from graph, *how difficult can it be?*

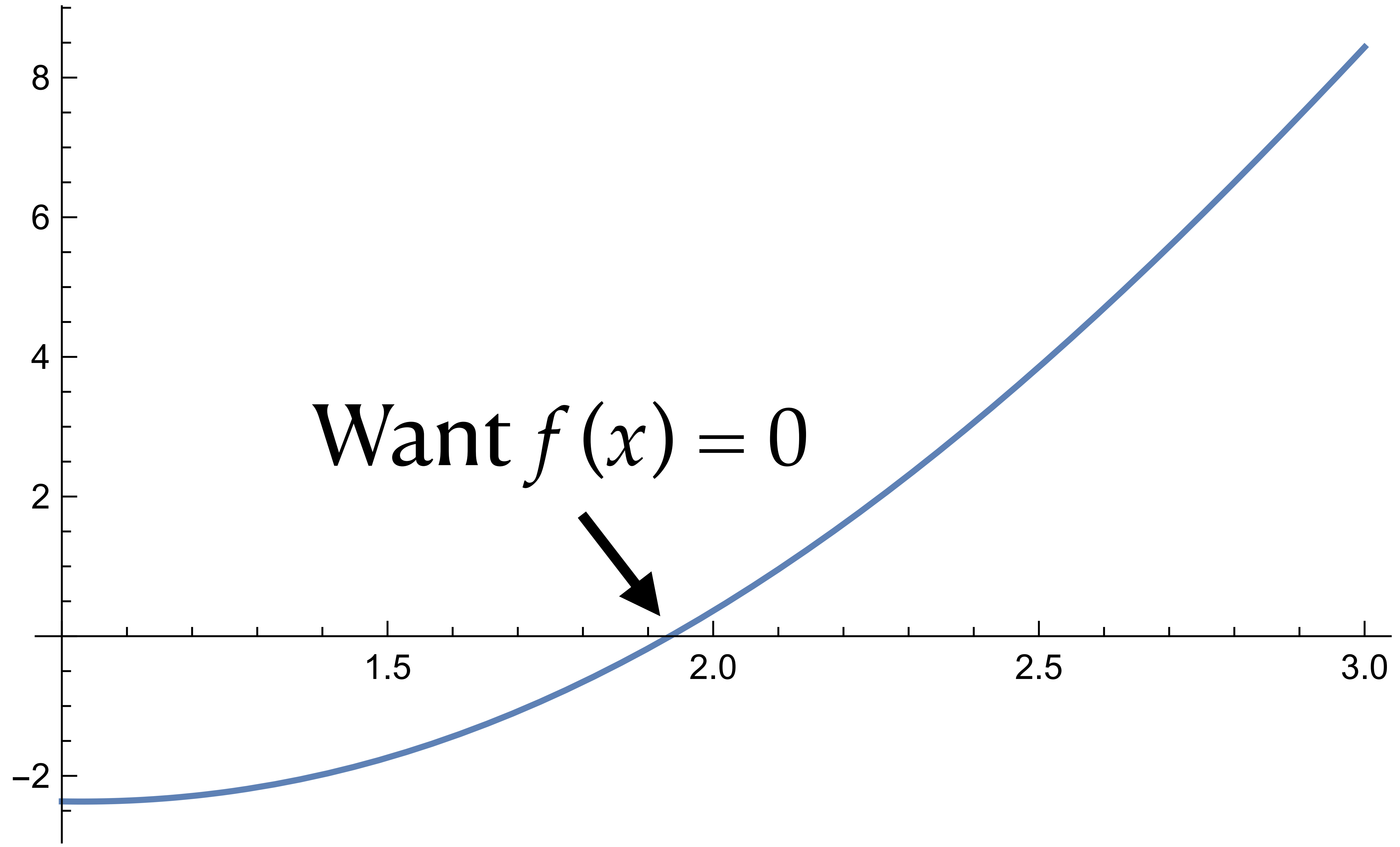
$$f(x) := x^2 - 4\sin x$$



A model problem

Can more or less read off solution from graph, *how difficult can it be?*

$$f(x) := x^2 - 4\sin x$$

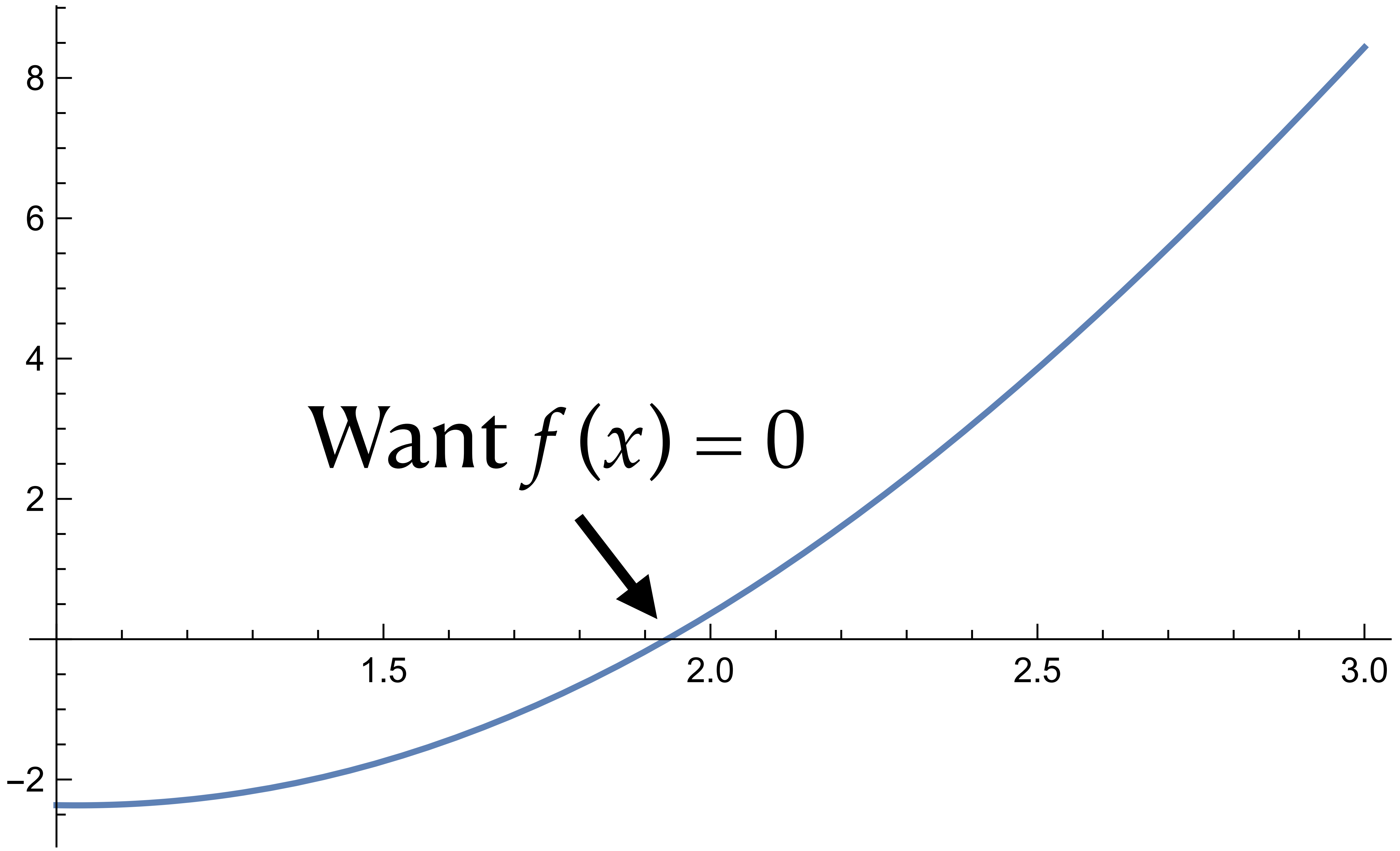


A model problem

Can more or less read off solution from graph, *how difficult can it be?*

$$f(x) := x^2 - 4\sin x$$

*Plot required thousands of function evaluations, each one is potentially **very expensive** (evaluating $f(x)$ once: test a new drug compound on patients, build a rocket and collect a sample on Mars.)*



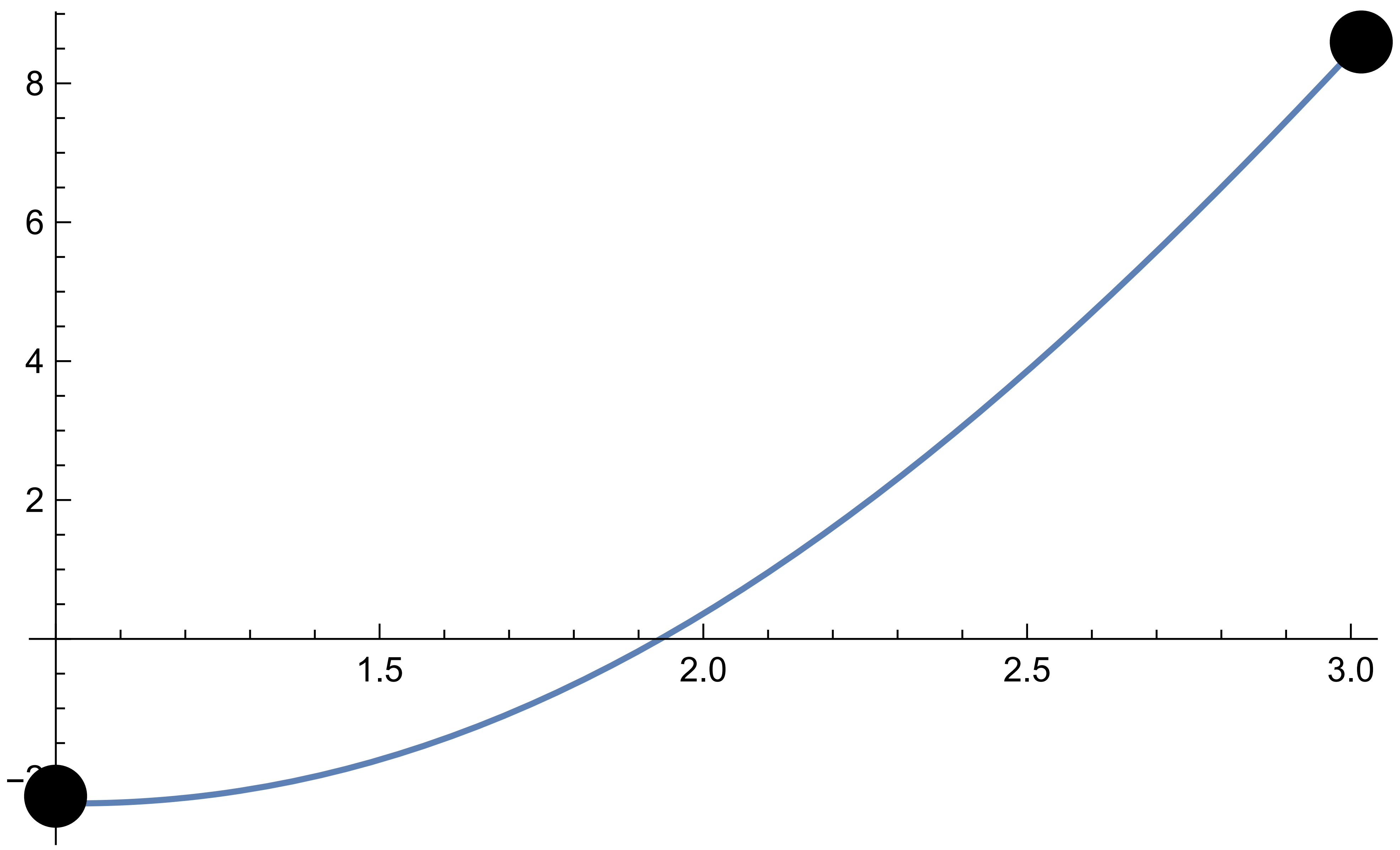
Intermediate Value Theorem

Intermediate Value Theorem

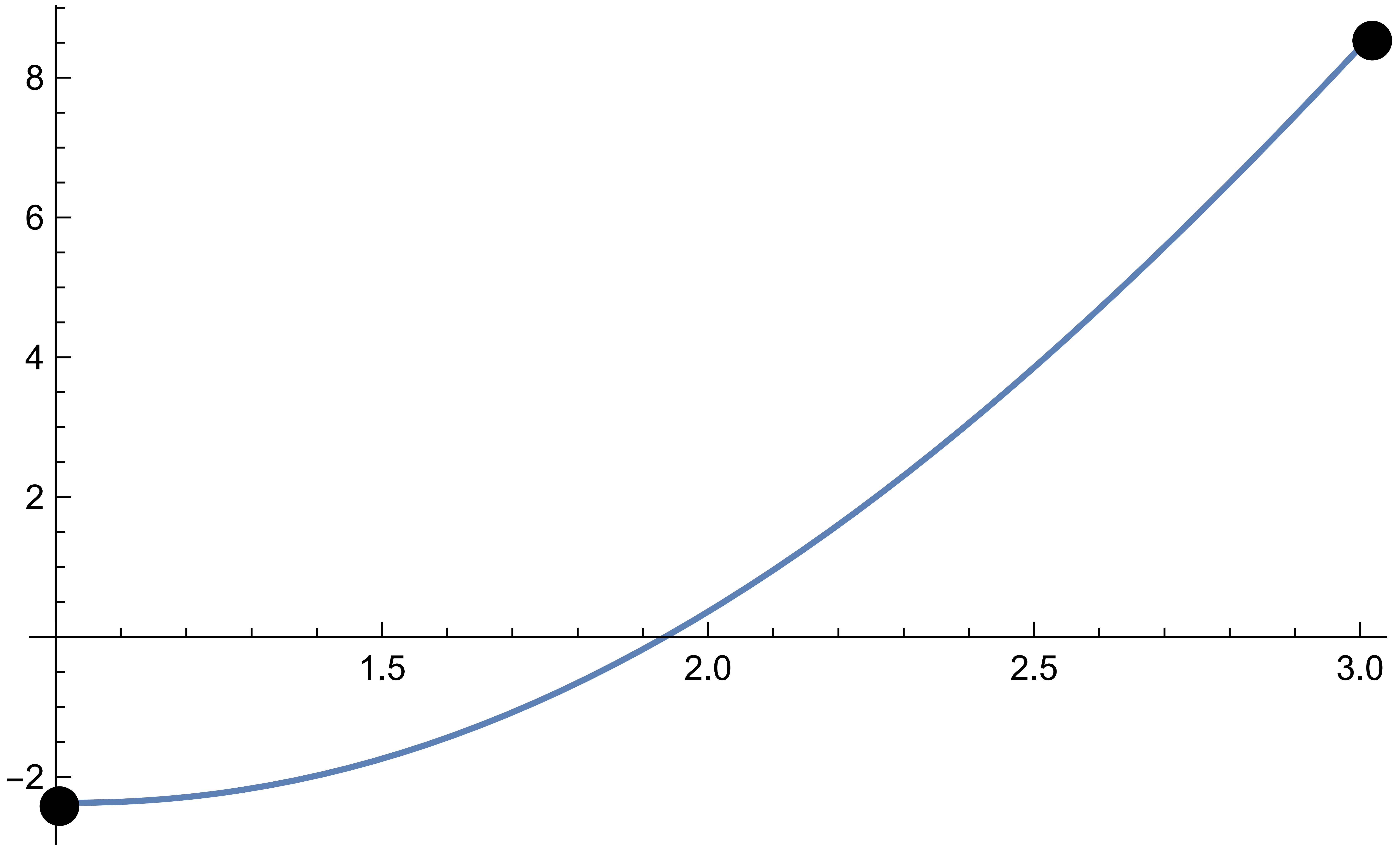
If f is continuous and $f(x_0) = y_0,\ f(x_1) = y_1,$
then $f(x)$ on (x_0, x_1) must pass through
<u>every</u> value between y_0 and y_1.

Intermediate Value Theorem

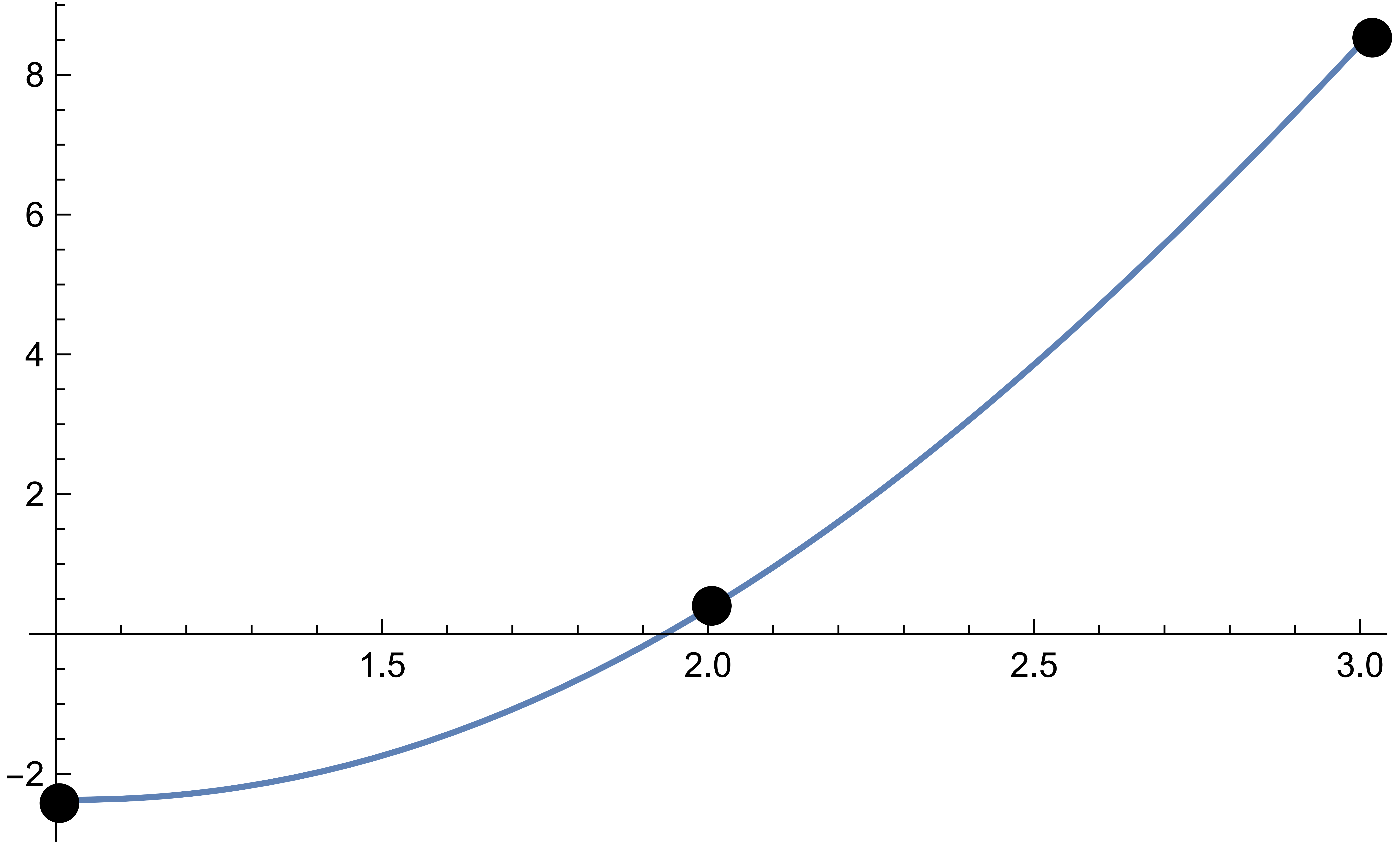
If f is continuous and $f(x_0) = y_0$, $f(x_1) = y_1$,
then $f(x)$ on (x_0, x_1) must pass through
<u>every</u> value between y_0 and y_1.



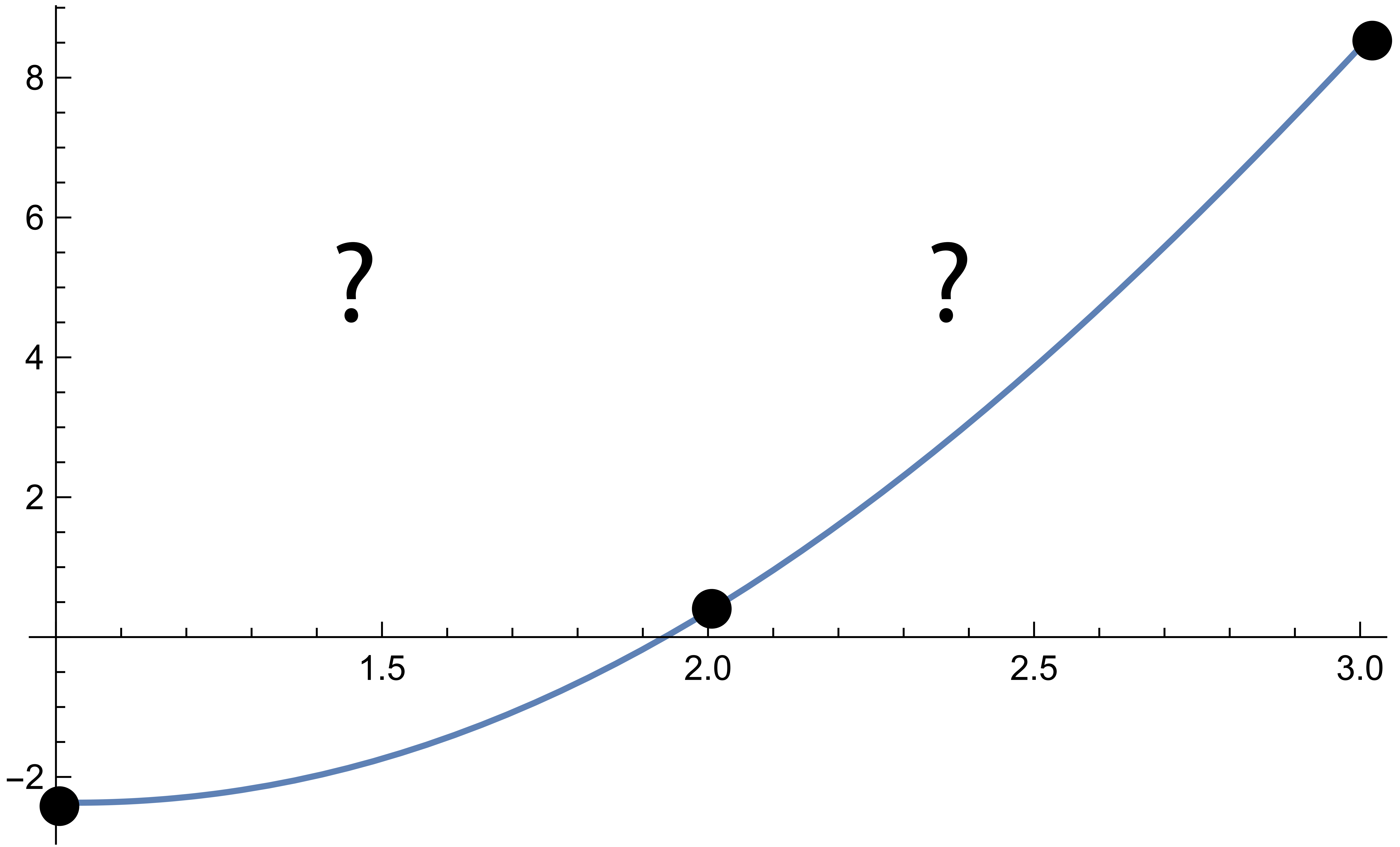
Use of continuity in our model problem



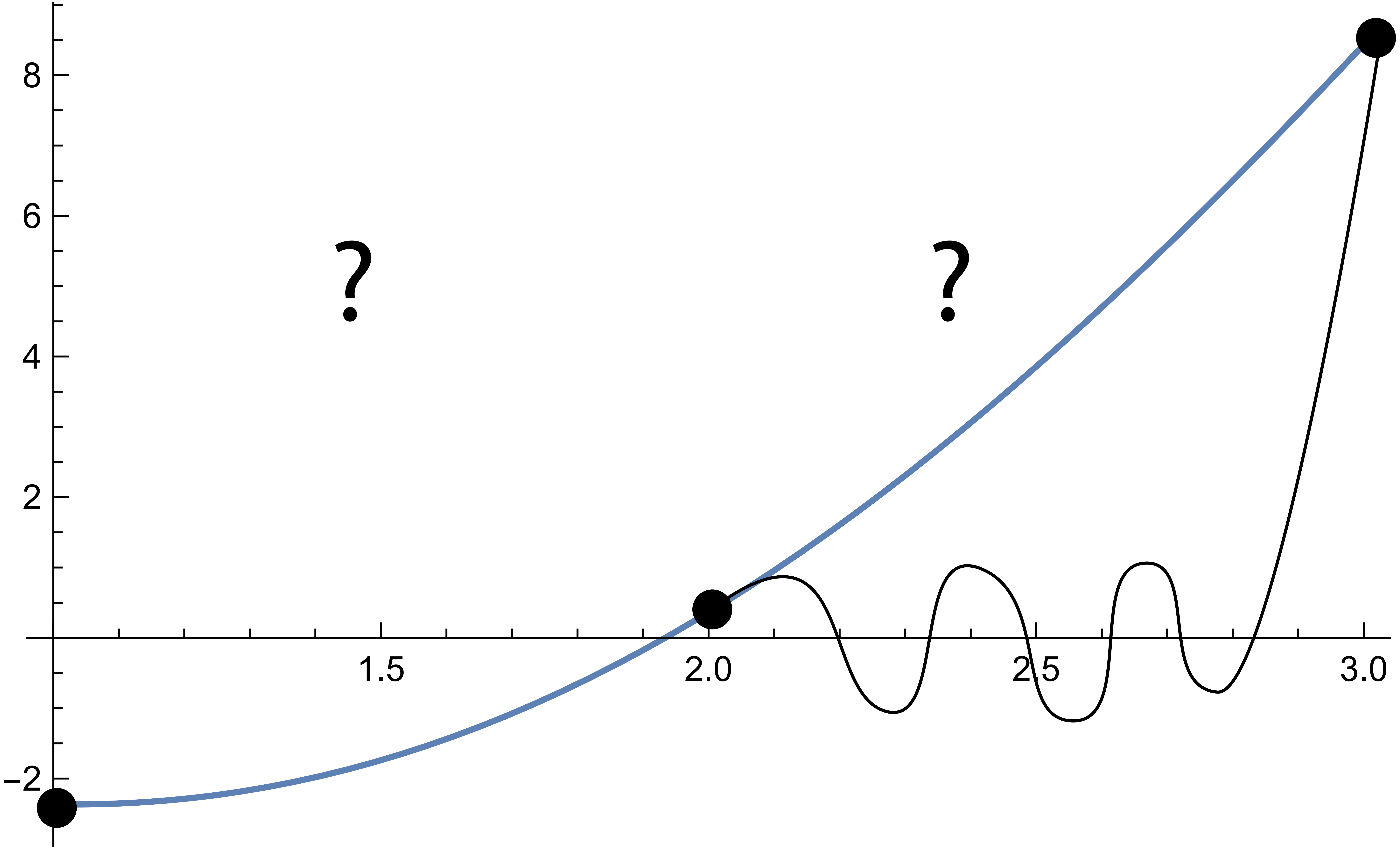
Use of continuity in our model problem



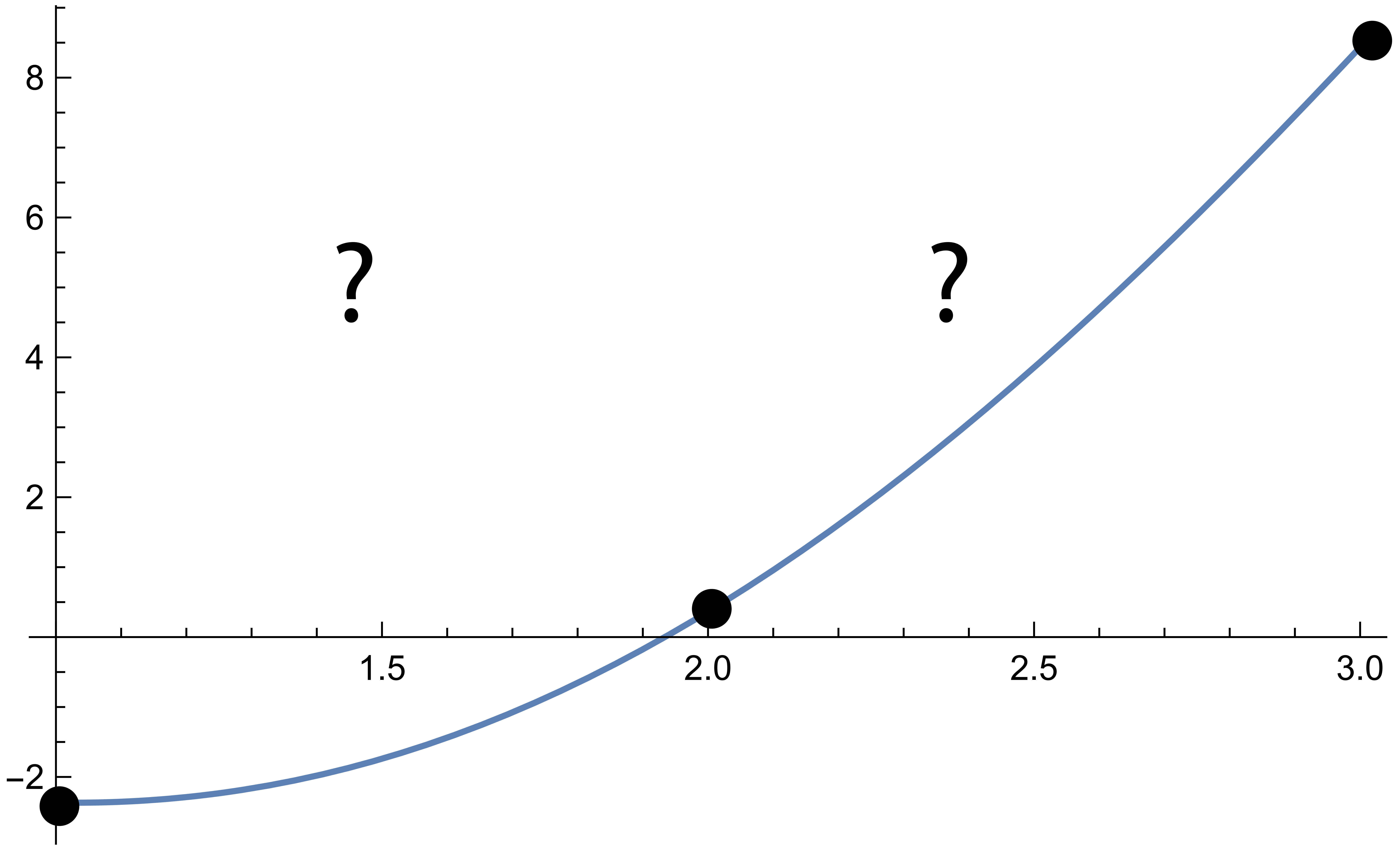
Use of continuity in our model problem



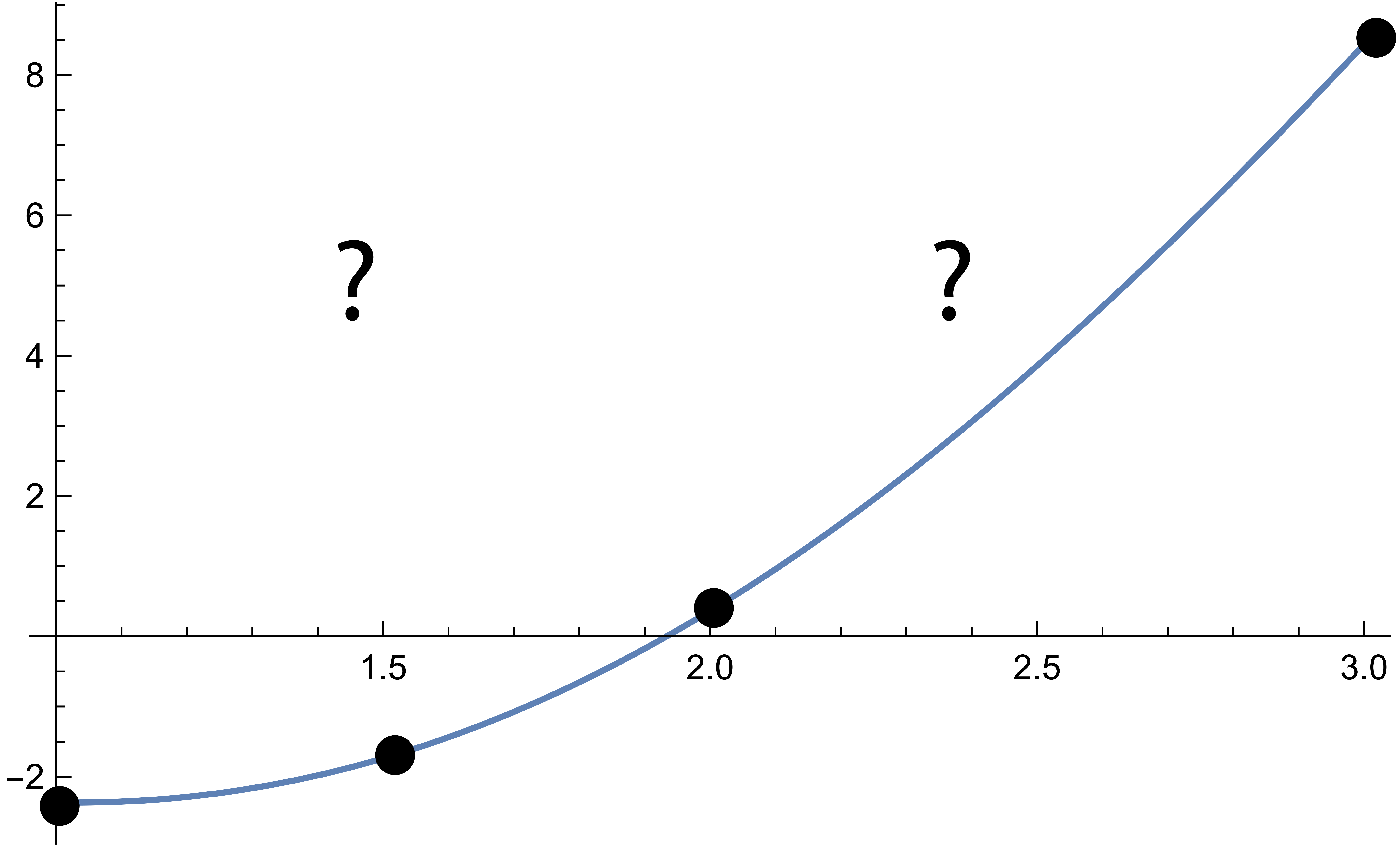
Use of continuity in our model problem



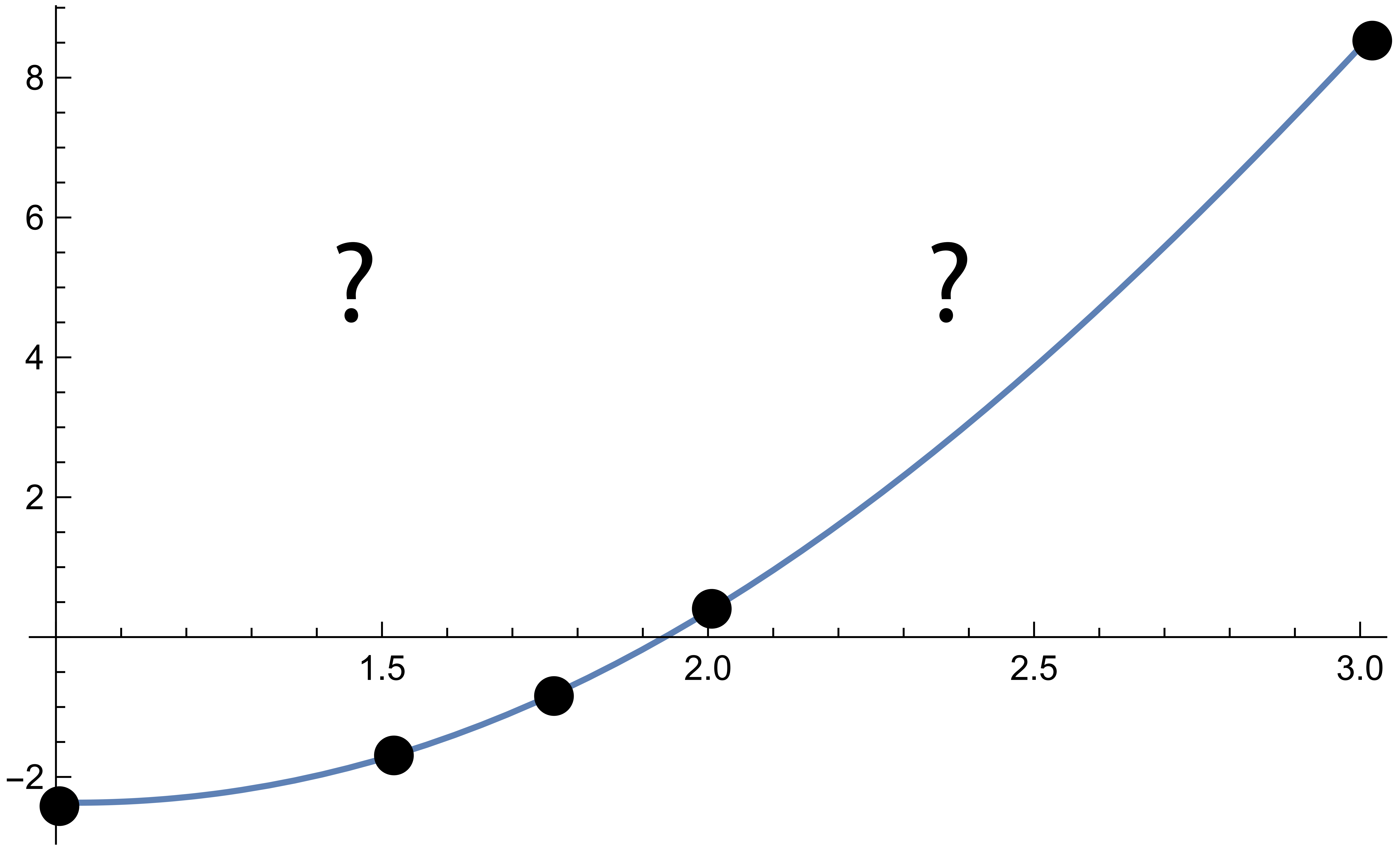
Use of continuity in our model problem



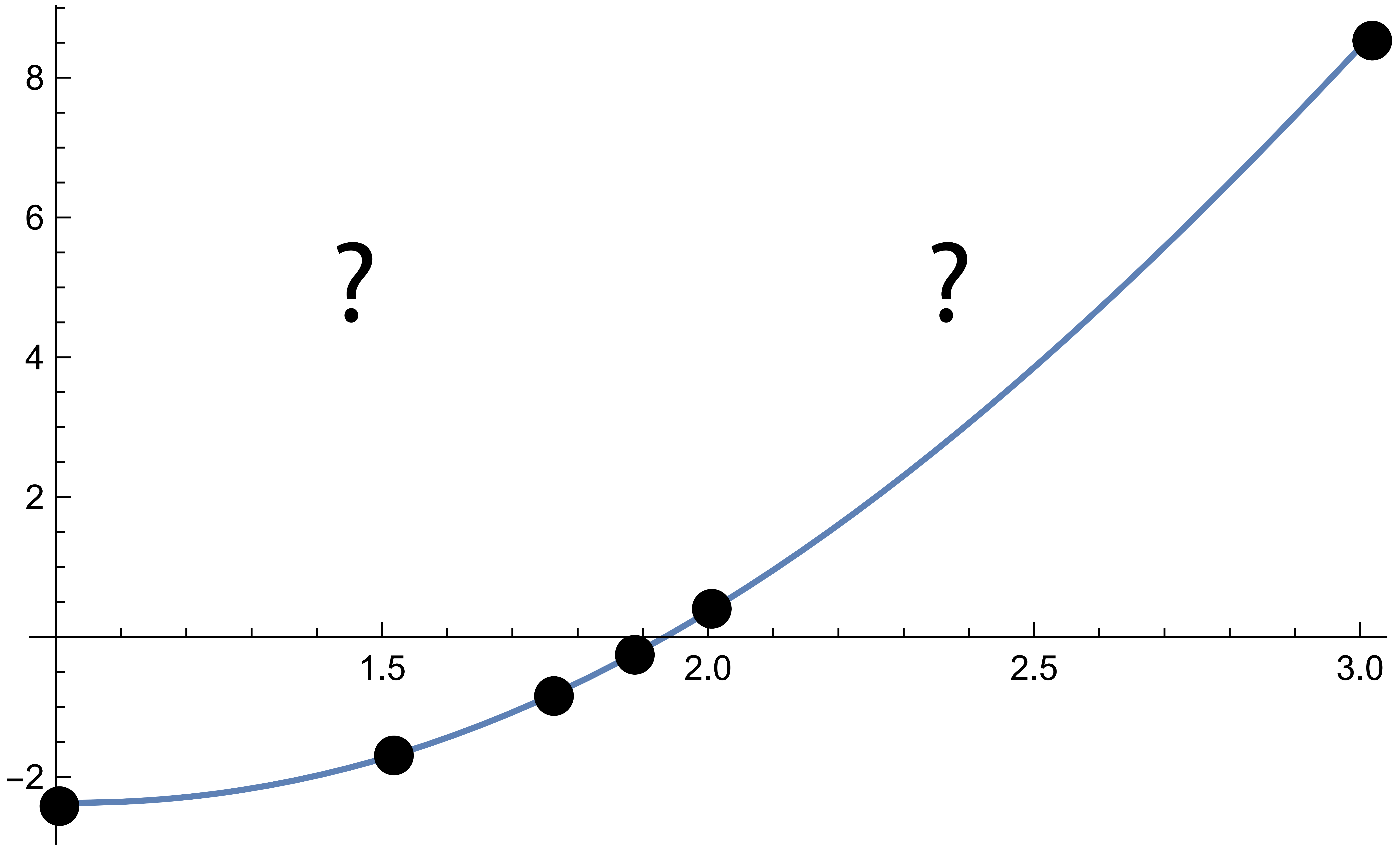
Use of continuity in our model problem



Use of continuity in our model problem

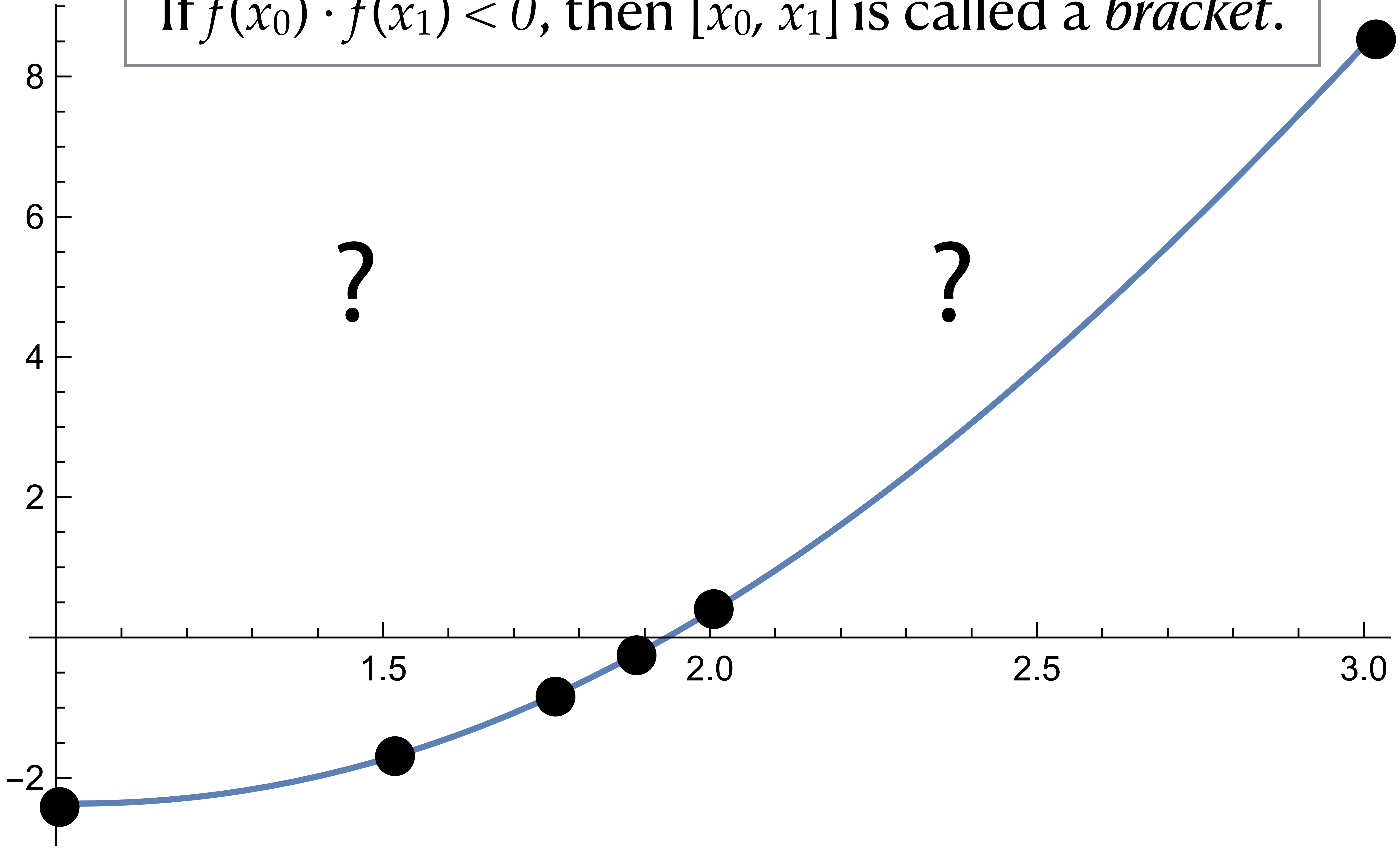


Use of continuity in our model problem



Use of continuity in our model problem

If $f(x_0) \cdot f(x_1) < 0$, then $[x_0, x_1]$ is called a *bracket*.



Bisection search

function — *"bracket"* — *stopping criterion thresholds*

```
def bisect(f, l, r, eps₁, eps₂):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Bisection search

function *"bracket"* *stopping criterion thresholds*

```
def bisect(f, l, r, eps1, eps2):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps1 or np.abs(l - r) < eps2:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Bisection search

function *"bracket"* *stopping criterion thresholds*

```
def bisect(f, l, r, eps1, eps2):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps1 or np.abs(l - r) < eps2:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Bisection search

function *"bracket"* *stopping criterion thresholds*

```
def bisect(f, l, r, eps1, eps2):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps1 or np.abs(l - r) < eps2:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Is this code optimal?

Bisection search

function *"bracket"* *stopping criterion thresholds*

```
def bisect(f, l, r, eps₁, eps₂):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Is this code optimal?

No, it could be implemented with ~1/3 the number of *f*-evaluations.

Bisection search

function *"bracket"* *stopping criterion thresholds*

```
def bisect(f, l, r, eps₁, eps₂):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

Does not use function values!
(only their sign is relevant)

Is this code optimal?

No, it could be implemented with ~1/3 the number of *f*-evaluations.

Bisection search

function *“bracket”* *stopping criterion thresholds*

```python
def bisect(f, l, r, eps1, eps2):
    while True:
        m = l + (r - l) / 2

        if np.abs(f(m)) < eps1 or np.abs(l - r) < eps2:
            return m

        if f(l) * f(m) < 0:
            r = m
        else
            l = m
```

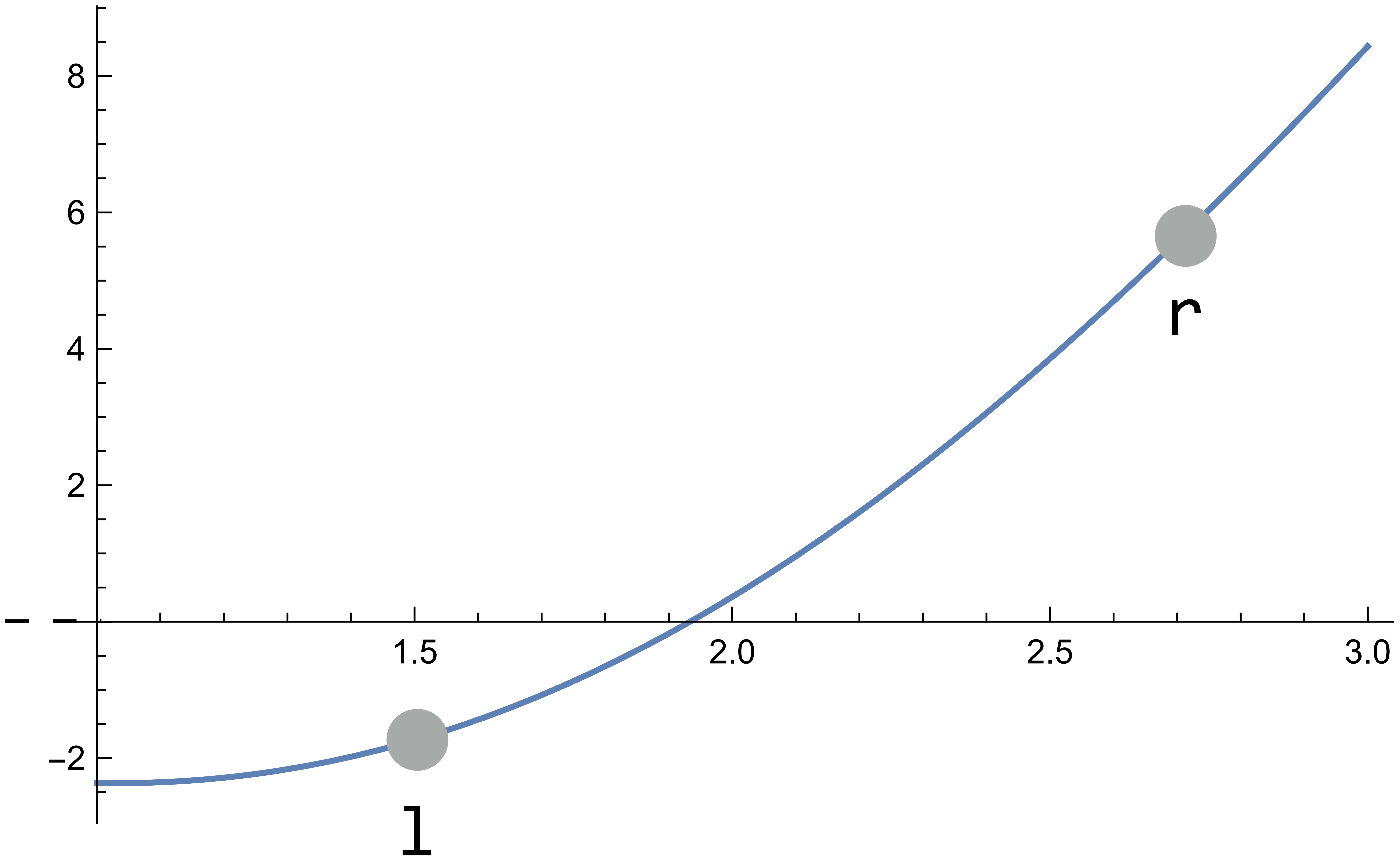
Does not use function values!
(only their sign is relevant)

Is this code optimal?

No, it could be implemented with ~1/3 the number of *f*-evaluations.

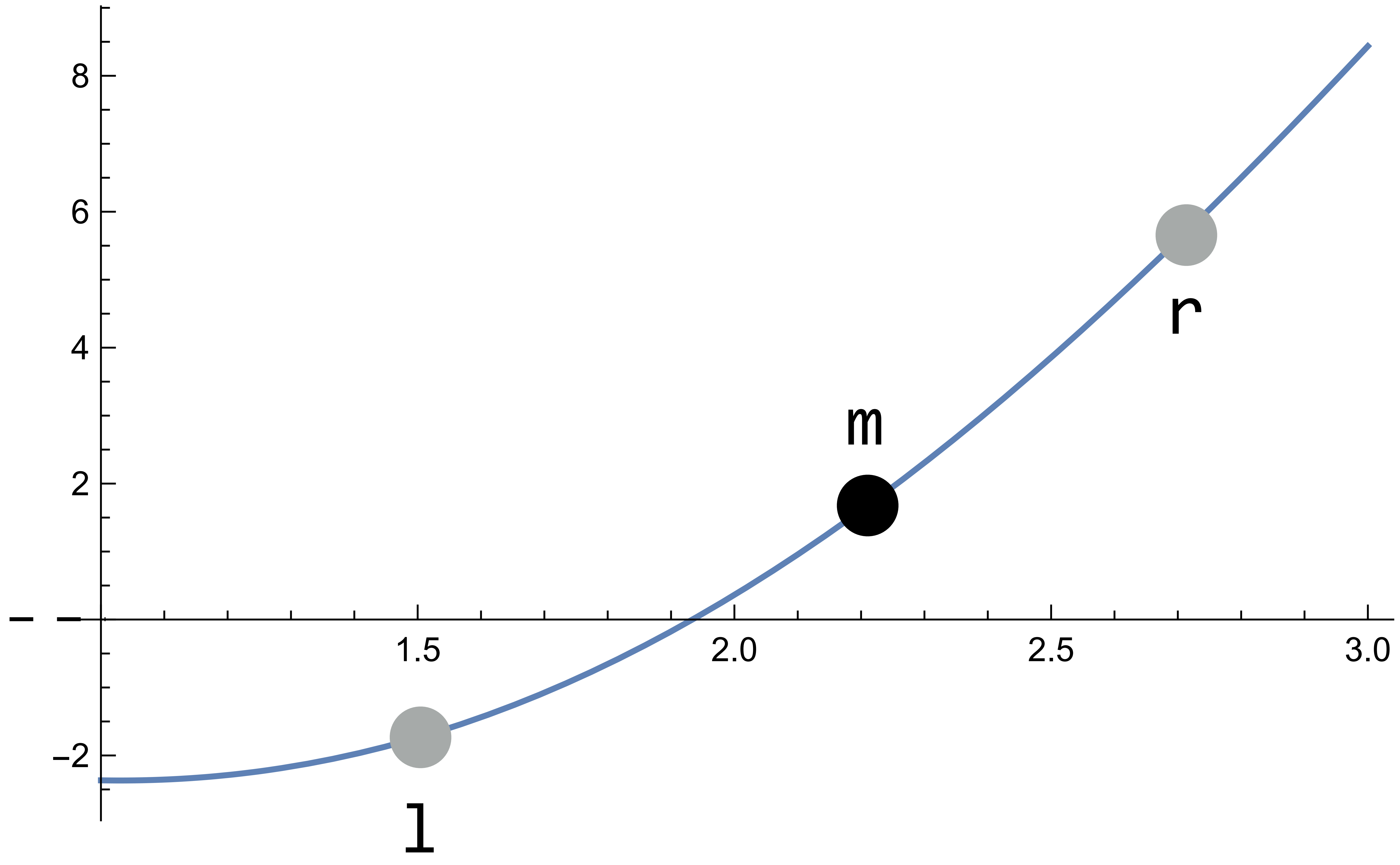
Stopping criteria for nonlinear methods

```
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
    return m
```



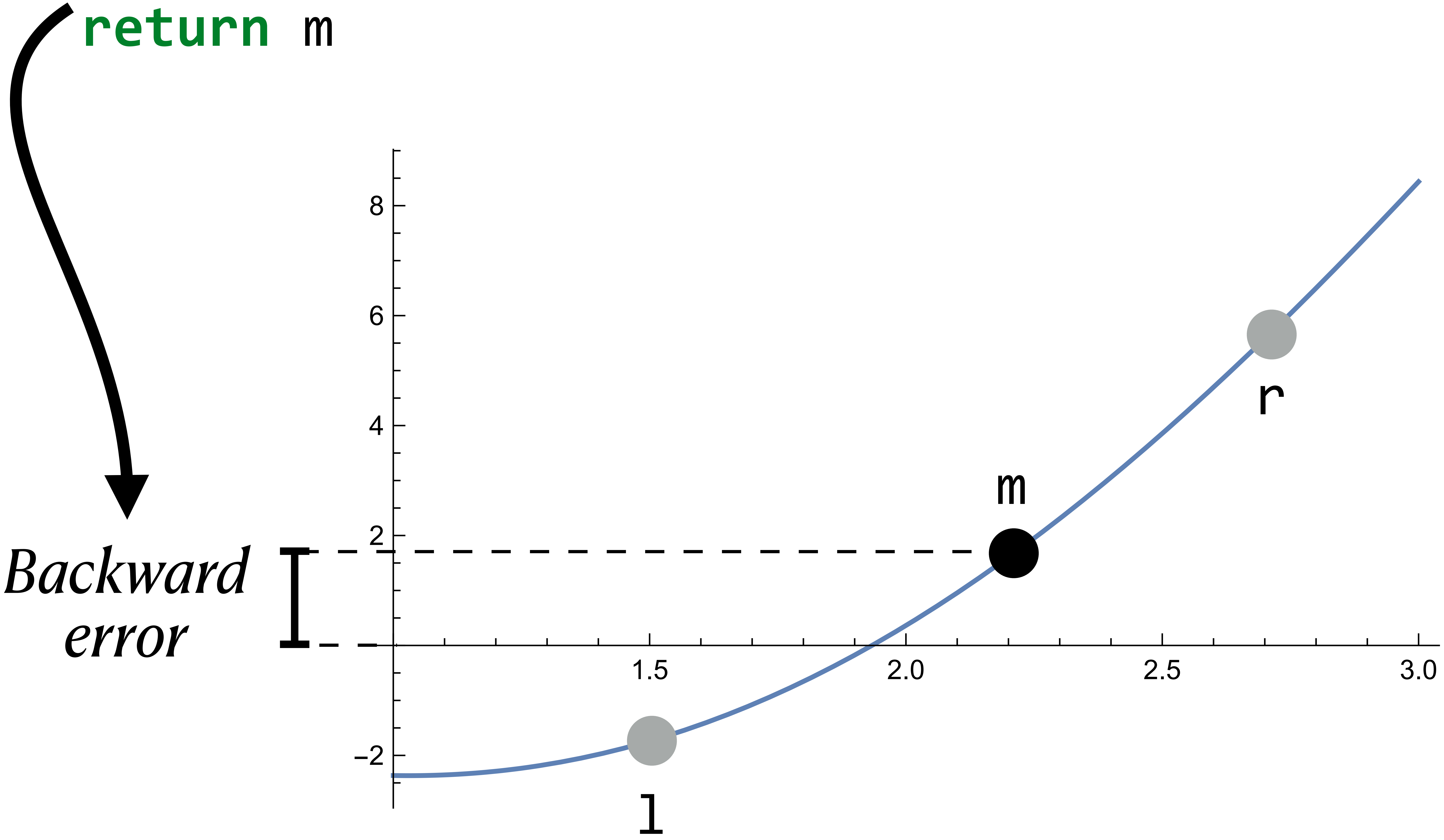
Stopping criteria for nonlinear methods

```
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
    return m
```



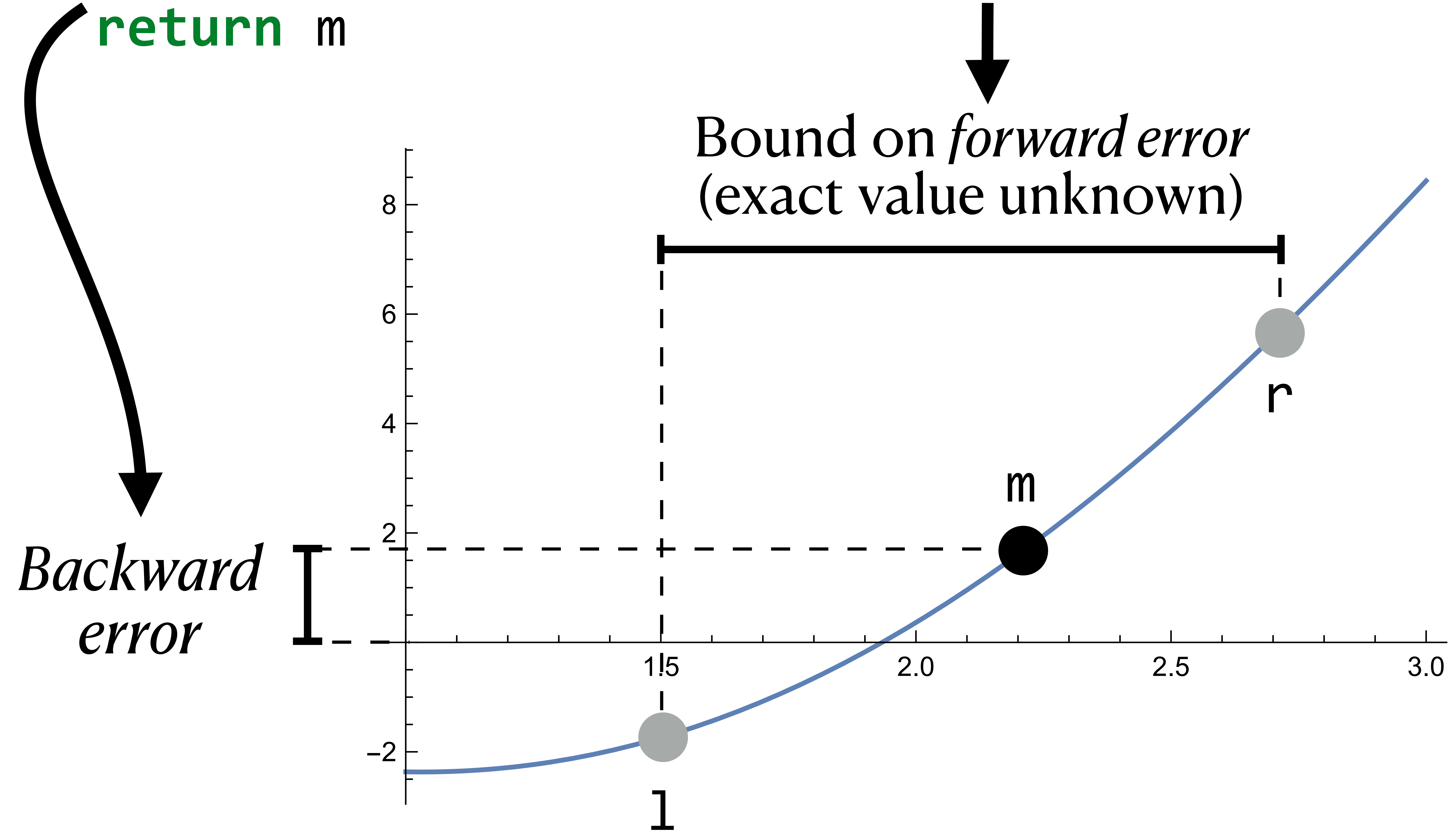
Stopping criteria for nonlinear methods

```
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
    return m
```



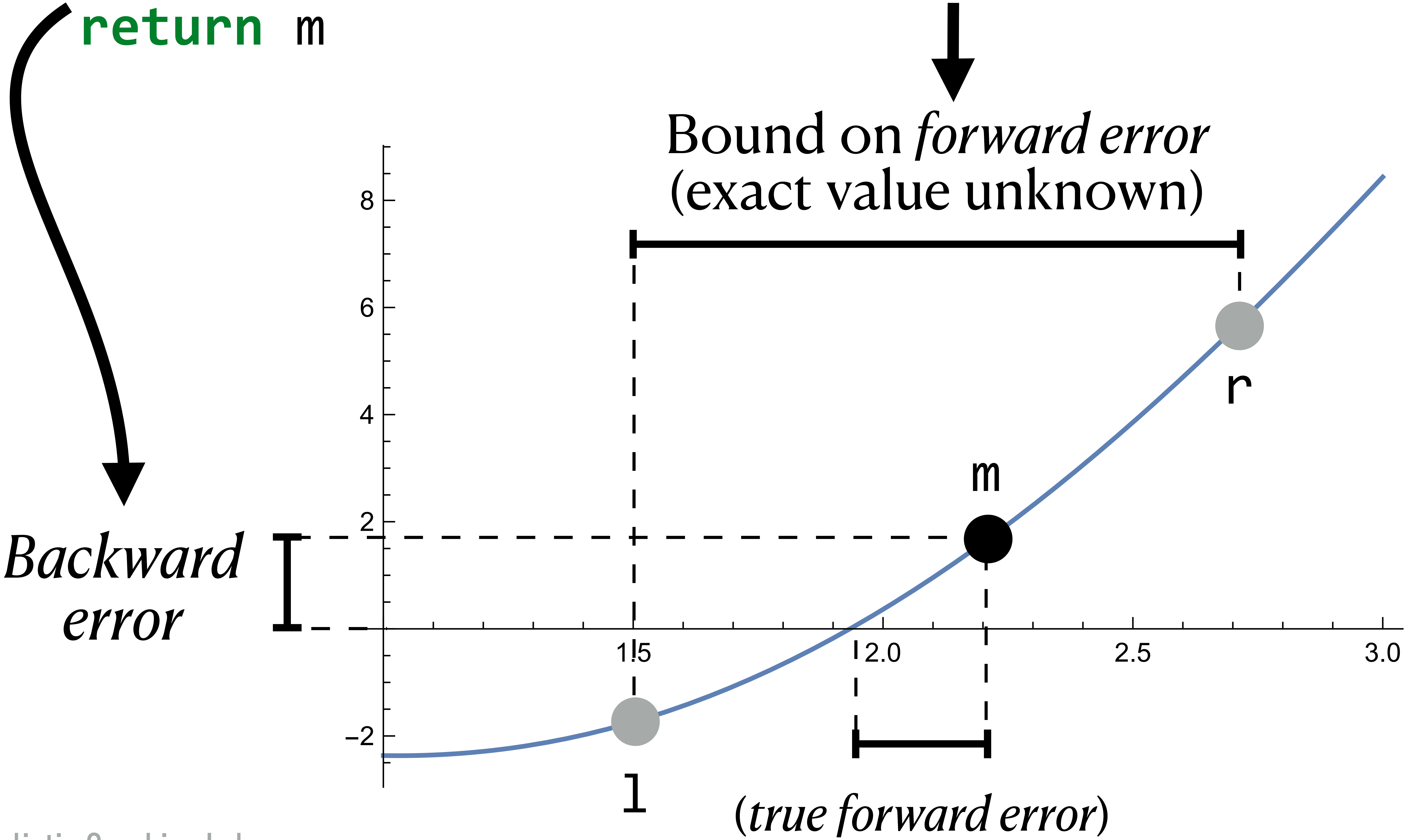
Stopping criteria for nonlinear methods

```
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
   return m
```

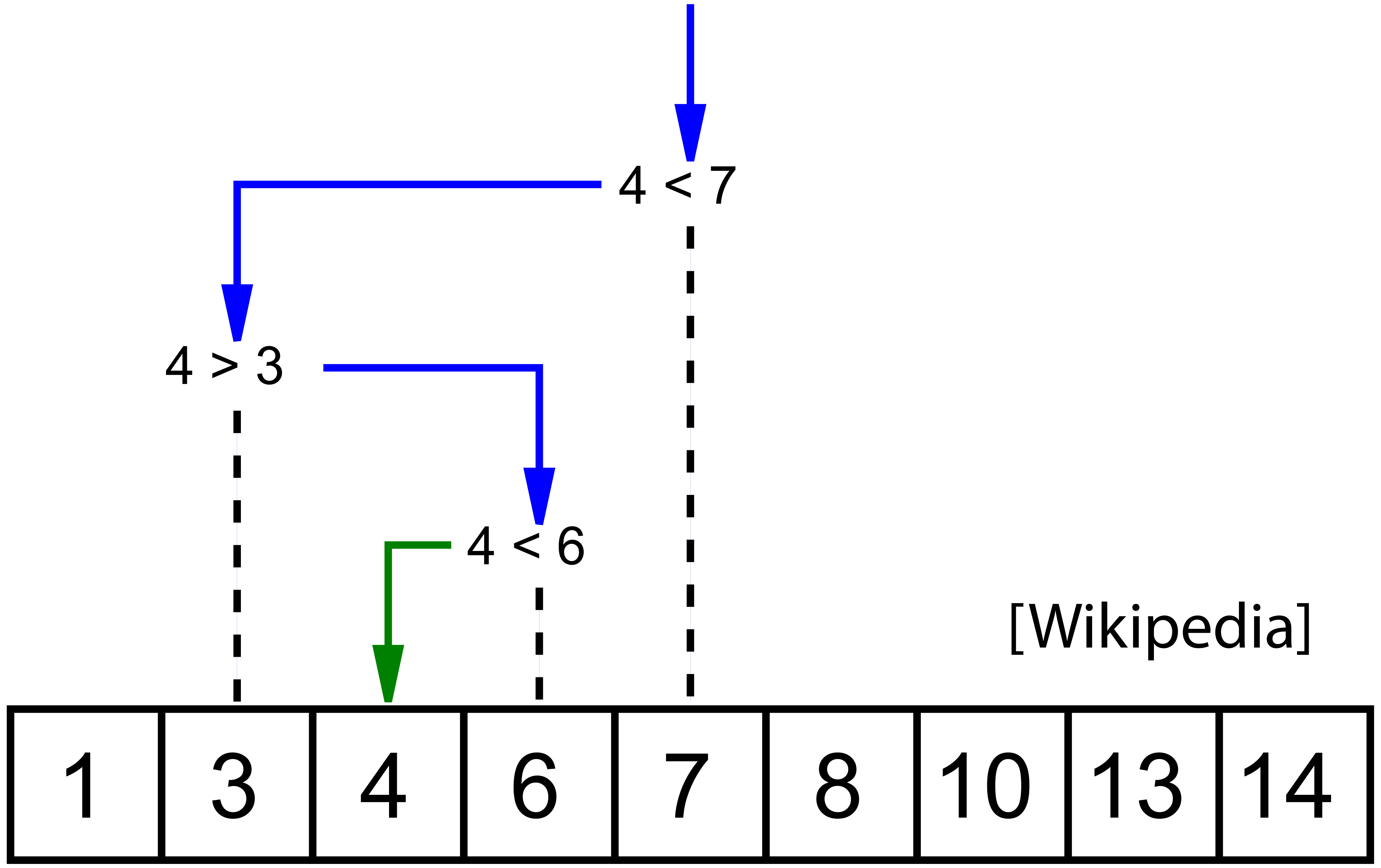


Stopping criteria for nonlinear methods

```
if np.abs(f(m)) < eps₁ or np.abs(l - r) < eps₂:
    return m
```

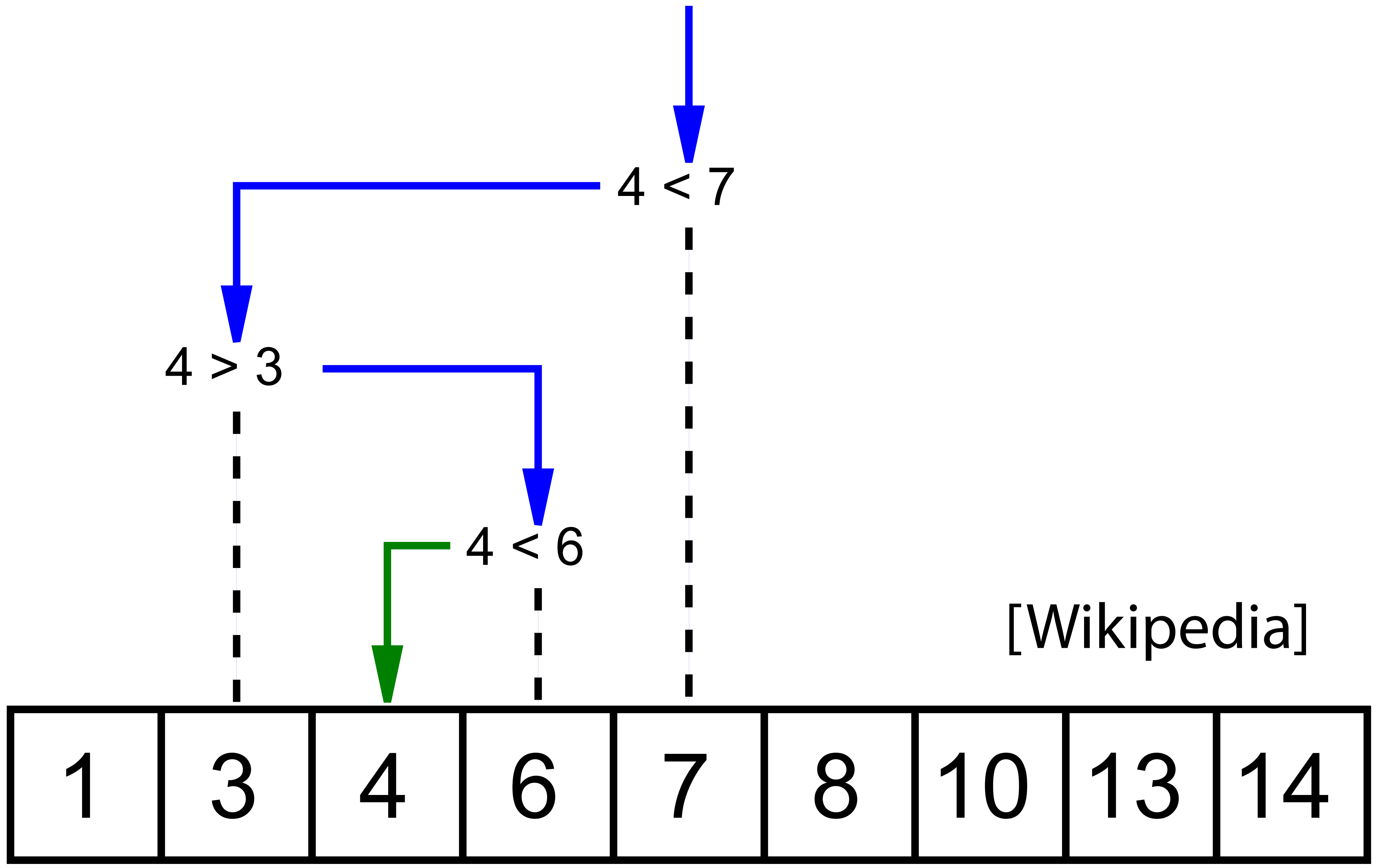


Does this seem at all familiar?



Binary search for 4 in sorted list.

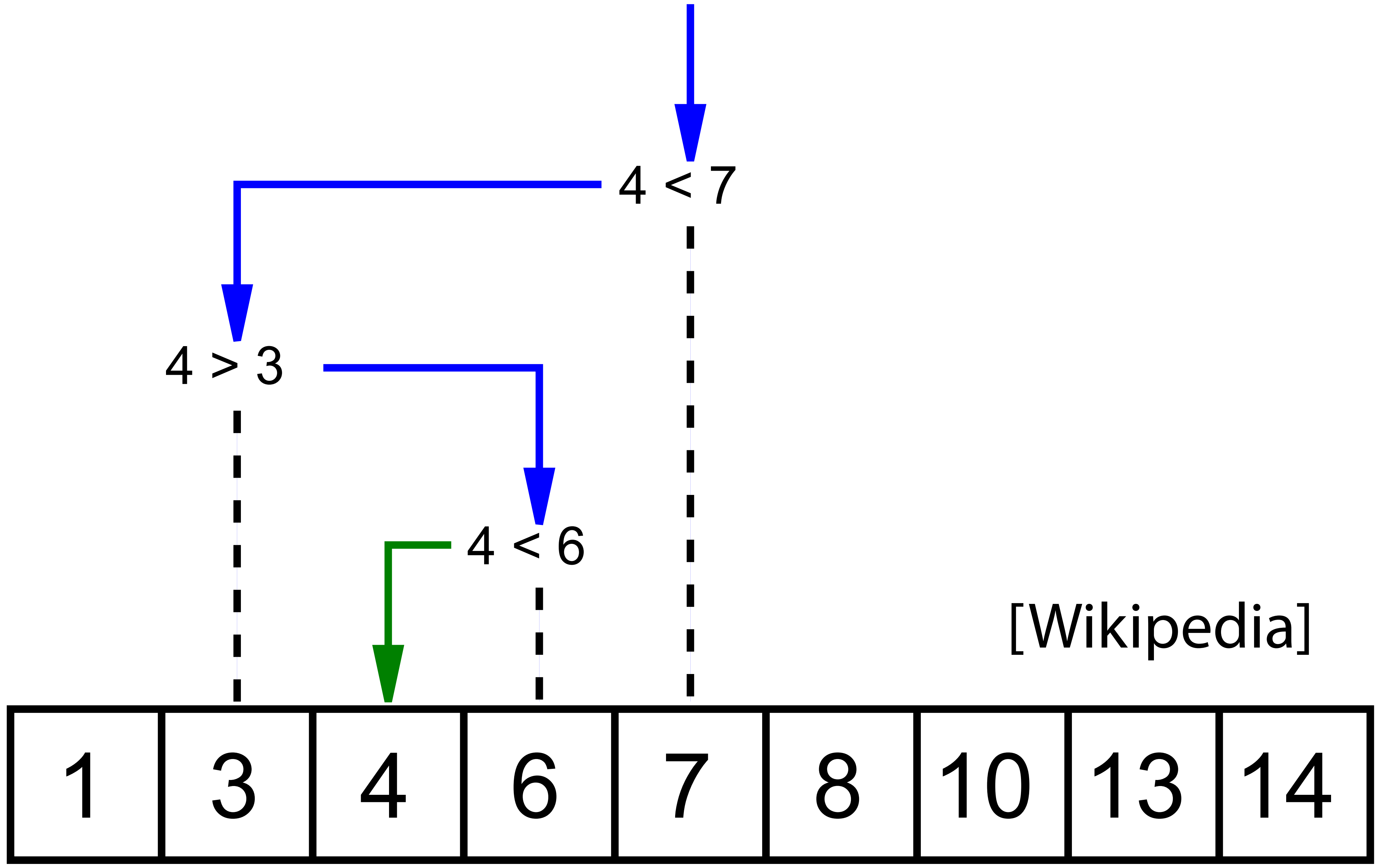
Does this seem at all familiar?



Binary search for 4 in sorted list.

Complexity of (discrete) algorithm: $O(\log n)$

Does this seem at all familiar?



Binary search for 4 in sorted list.

Complexity of (discrete) algorithm: $O(\log n)$

Can we find an analogy of "*complexity*" for root finding?

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

- o is called the **order of convergence**, which tells us how quickly the algorithm convergences.
  - $o = 1$: linear convergence, $o = 2$: quadratic convergence, etc.

Order and rate of convergence

Suppose that we can find numbers o and r so that

$$\lim_{k\to\infty} \frac{E_{k+1}}{E_k^o} = r.$$

where E_k is the error after iteration k, then:

- o is called the **order of convergence**, which tells us how quickly the algorithm convergences.
    - $o = 1$: linear convergence, $o = 2$: quadratic convergence, etc.
- r is called the **rate of convergence**. It distinguishes convergence speed of algorithms with the same order.

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Generally: $E_{k+1} = \frac{1}{2} E_k$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Generally: $E_{k+1} = \frac{1}{2} E_k \Leftrightarrow \frac{E_{k+1}}{E_k^1} = \frac{1}{2}$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Generally: $E_{k+1} = \frac{1}{2} E_k \Leftrightarrow \frac{E_{k+1}}{E_k^1} = \frac{1}{2}$

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l)/2$

Generally: $E_{k+1} = \frac{1}{2} E_k \Leftrightarrow \frac{E_{k+1}}{E_k^1} = \frac{1}{2}$

In other words: **order** of convergence = 1 (*linear*)
rate of convergence = 1 / 2

Convergence order and rate of bisection

Error bound (before 1st iteration) : $r - l$

Error bound (after 1st iteration) : $(r - l) / 2$

Generally: $E_{k+1} = \frac{1}{2} E_k \Leftrightarrow \frac{E_{k+1}}{E_k^1} = \frac{1}{2}$

In other words: **order** of convergence = 1 (*linear*)
rate of convergence = 1 / 2

A method with a **linear order of convergence** gains a fixed number of accurate digits per iteration (depending on **rate**). Here:

- 1 base-2 digit every iteration. 1 base-10 digit every $\frac{\log 10}{\log 2} \approx 3$ iterations.

Convergence speed of bisection

Let's apply bisection to the root-finding example function with

$$f(x) := x^2 - 4\sin x$$

Here, l and r denote the bracket; the solution lies in between.

l	$f(l)$	r	$f(r)$
1.000000	−2.365884	3.000000	8.435520
1.000000	−2.365884	2.000000	0.362810
1.500000	−1.739980	2.000000	0.362810
1.750000	−0.873444	2.000000	0.362810
1.875000	−0.300718	2.000000	0.362810
1.875000	−0.300718	1.937500	0.019849
1.906250	−0.143255	1.937500	0.019849
1.921875	−0.062406	1.937500	0.019849
1.929688	−0.021454	1.937500	0.019849

l	$f(l)$	r	$f(r)$
1.933594	−0.000846	1.937500	0.019849
1.933594	−0.000846	1.935547	0.009491
1.933594	−0.000846	1.934570	0.004320
1.933594	−0.000846	1.934082	0.001736

[Heath]

Convergence speed of bisection

Let's apply bisection to the root-finding example function with

$$f(x) := x^2 - 4\sin x$$

Here, l and r denote the bracket; the solution lies in between.

l	$f(l)$	r	$f(r)$
1.000000	−2.365884	3.000000	8.435520
1.000000	−2.365884	2.000000	0.362810
1.500000	−1.739980	2.000000	0.362810
1.750000	−0.873444	2.000000	0.362810
1.875000	−0.300718	2.000000	0.362810
1.875000	−0.300718	1.937500	0.019849
1.906250	−0.143255	1.937500	0.019849
1.921875	−0.062406	1.937500	0.019849
1.929688	−0.021454	1.937500	0.019849

l	$f(l)$	r	$f(r)$
1.933594	−0.000846	1.937500	0.019849
1.933594	−0.000846	1.935547	0.009491
1.933594	−0.000846	1.934570	0.004320
1.933594	−0.000846	1.934082	0.001736

[Heath]

Convergence speed of bisection

Let's apply bisection to the root-finding example function with

$$f(x) := x^2 - 4\sin x$$

Here, l and r denote the bracket; the solution lies in between.

l	$f(l)$	r	$f(r)$
1.000000	−2.365884	3.000000	8.435520
1.000000	−2.365884	2.000000	0.362810
1.500000	−1.739980	2.000000	0.362810
1.750000	−0.873444	2.000000	0.362810
1.875000	−0.300718	2.000000	0.362810
1.875000	−0.300718	1.937500	0.019849
1.906250	−0.143255	1.937500	0.019849
1.921875	−0.062406	1.937500	0.019849
1.929688	−0.021454	1.937500	0.019849

l	$f(l)$	r	$f(r)$
1.933594	−0.000846	1.937500	0.019849
1.933594	−0.000846	1.935547	0.009491
1.933594	−0.000846	1.934570	0.004320
1.933594	−0.000846	1.934082	0.001736

[Heath]

Convergence speed of bisection

Let's apply bisection to the root-finding example function with

$$f(x) := x^2 - 4\sin x$$

Here, *l* and *r* denote the bracket; the solution lies in between.

l	$f(l)$	r	$f(r)$
1.000000	−2.365884	3.000000	8.435520
1.000000	−2.365884	2.000000	0.362810
1.500000	−1.739980	2.000000	0.362810
1.750000	−0.873444	2.000000	0.362810
1.875000	−0.300718	2.000000	0.362810
1.875000	−0.300718	1.937500	0.019849
1.906250	−0.143255	1.937500	0.019849
1.921875	−0.062406	1.937500	0.019849
1.929688	−0.021454	1.937500	0.019849

l	$f(l)$	r	$f(r)$
1.933594	−0.000846	1.937500	0.019849
1.933594	−0.000846	1.935547	0.009491
1.933594	−0.000846	1.934570	0.004320
1.933594	−0.000846	1.934082	0.001736

[Heath]

Midpoint = **1.933**838

True solution = 1.93375376282..

(13 iterations for ~4 digits)

Newton's method

- Based on first-order Taylor expansion:

$$f(x+h) \approx f(x) + f'(x)h$$

Newton's method

- Based on first-order Taylor expansion:

$$f(x+h) \approx f(x) + f'(x)h$$

- Let's set this approximation to zero and solve for h.

$$f(x+h) = 0 \Leftrightarrow h = -\frac{f(x)}{f'(x)}$$

Newton's method

- Based on first-order Taylor expansion:

$$f(x+h) \approx f(x) + f'(x)h$$

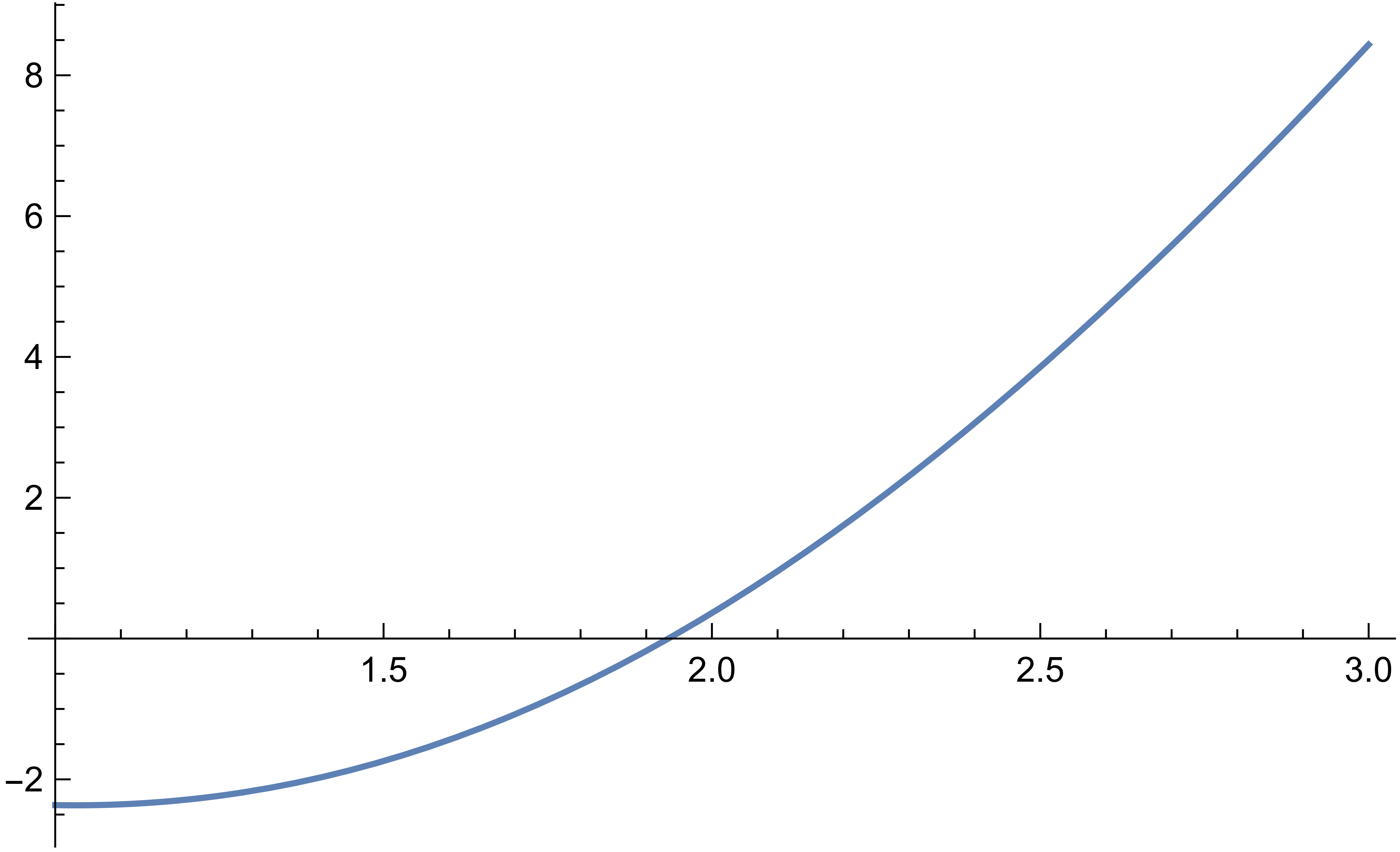
- Let's set this approximation to zero and solve for h.

$$f(x+h) = 0 \Leftrightarrow h = -\frac{f(x)}{f'(x)}$$

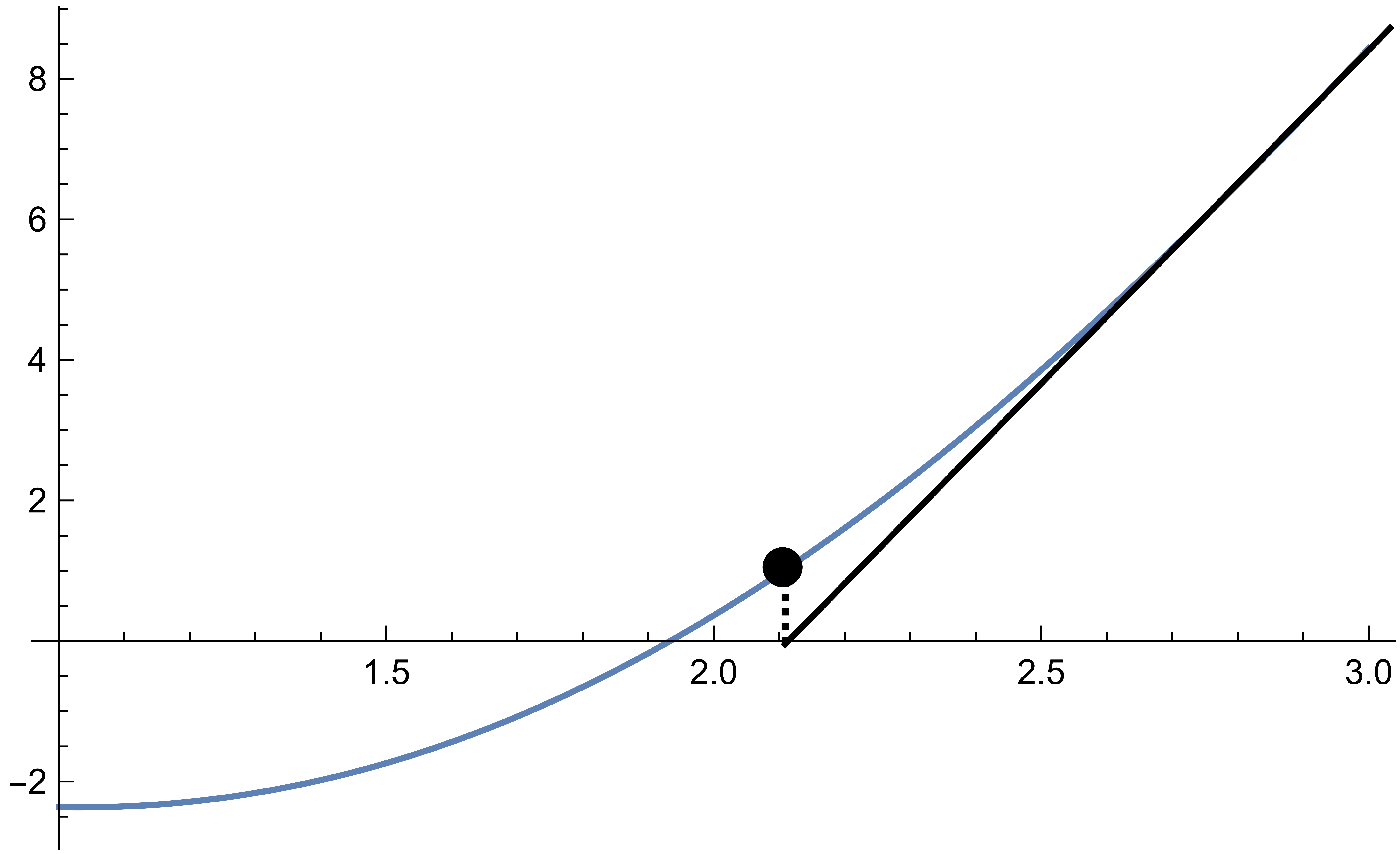
- Move to that position, and repeat..

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

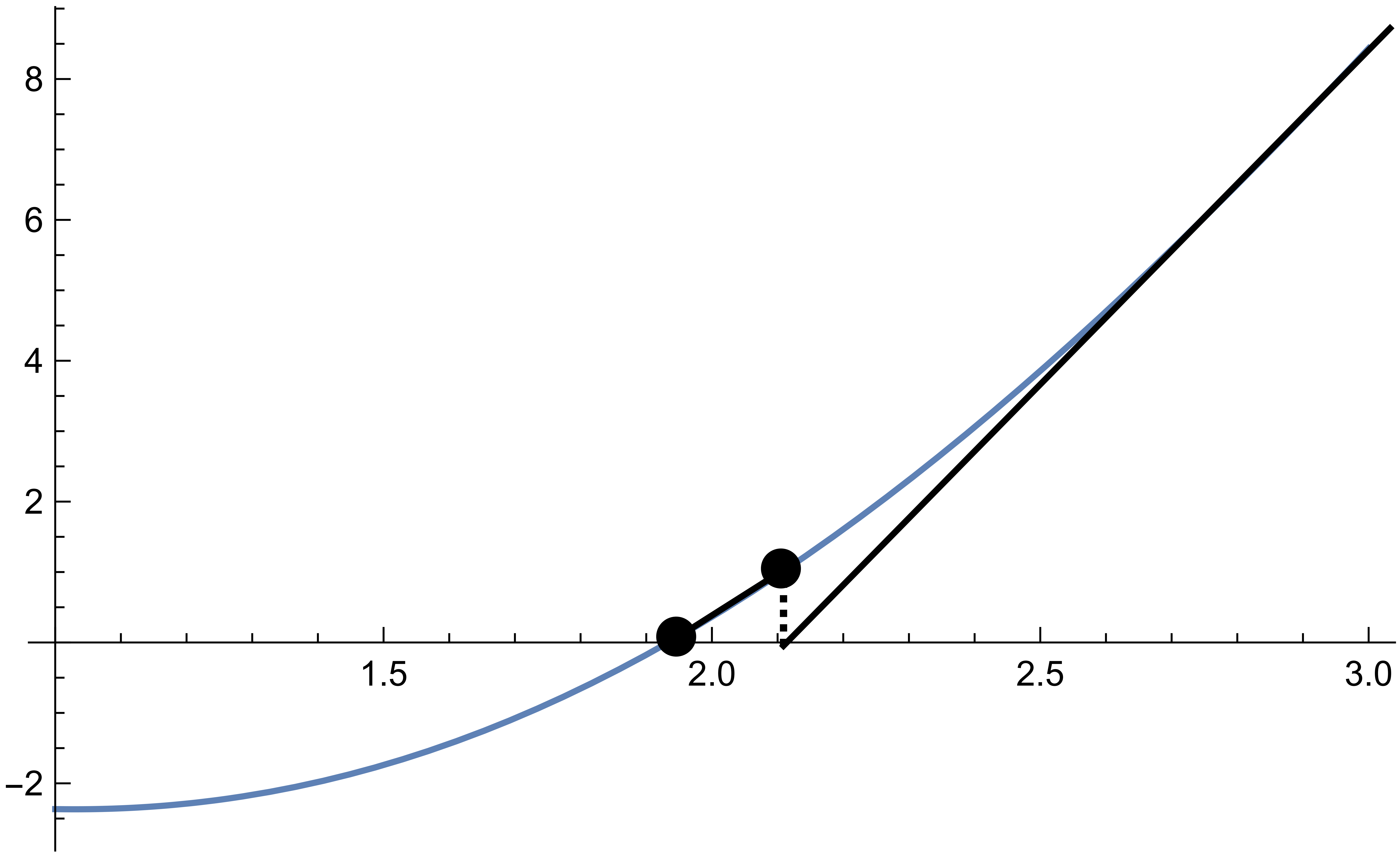
Visualization of Newton's method



Visualization of Newton's method



Visualization of Newton's method



Convergence of Newton's method

$$f(x) := x^2 - 4\sin x$$

Newton's method requires the derivative of f. Here,

$$f'(x) = 2x - 4\cos x$$

Convergence of Newton's method

x	$f(x)$	$f'(x)$	h
3.000000	8.435520	9.959970	−0.846942
2.153058	1.294772	6.505771	−0.199019
1.954039	0.108438	5.403795	−0.020067
1.933972	0.001152	5.288919	−0.000218
1.933754	0.000000	5.287670	0.000000

(5 iterations for ~7 digits)

[Heath]

Convergence of Newton's method

x	$f(x)$	$f'(x)$	h
3.000000	8.435520	9.959970	−0.846942
2.153058	1.294772	6.505771	−0.199019
1.954039	0.108438	5.403795	−0.020067
1.933972	0.001152	5.288919	−0.000218
1.933754	0.000000	5.287670	0.000000

(5 iterations for ~7 digits)

[Heath]

The method has a quadratic order of convergence, meaning that the number of valid digits approximately **doubles** per iteration.

1D Root Finding: Pros/Cons

Property	Bisection	Newton's method

1D Root Finding: Pros/Cons

Property	Bisection	Newton's method
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)

1D Root Finding: Pros/Cons

Property	Bisection	Newton's method
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)
Reliability	😊 Always works	😞 Divergence, multiple roots, ...

1D Root Finding: Pros/Cons

Property	Bisection	Newton's method
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)
Reliability	😊 Always works	😞 Divergence, multiple roots, ...
Requirements	😐 Continuity, bracket	😐 Derivative

1D Root Finding: Pros/Cons

Property	**Bisection**	**Newton's method**
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)
Reliability	😊 Always works	😞 Divergence, multiple roots, …
Requirements	😐 Continuity, bracket	😐 Derivative

Can combine both: Newton-Bisection

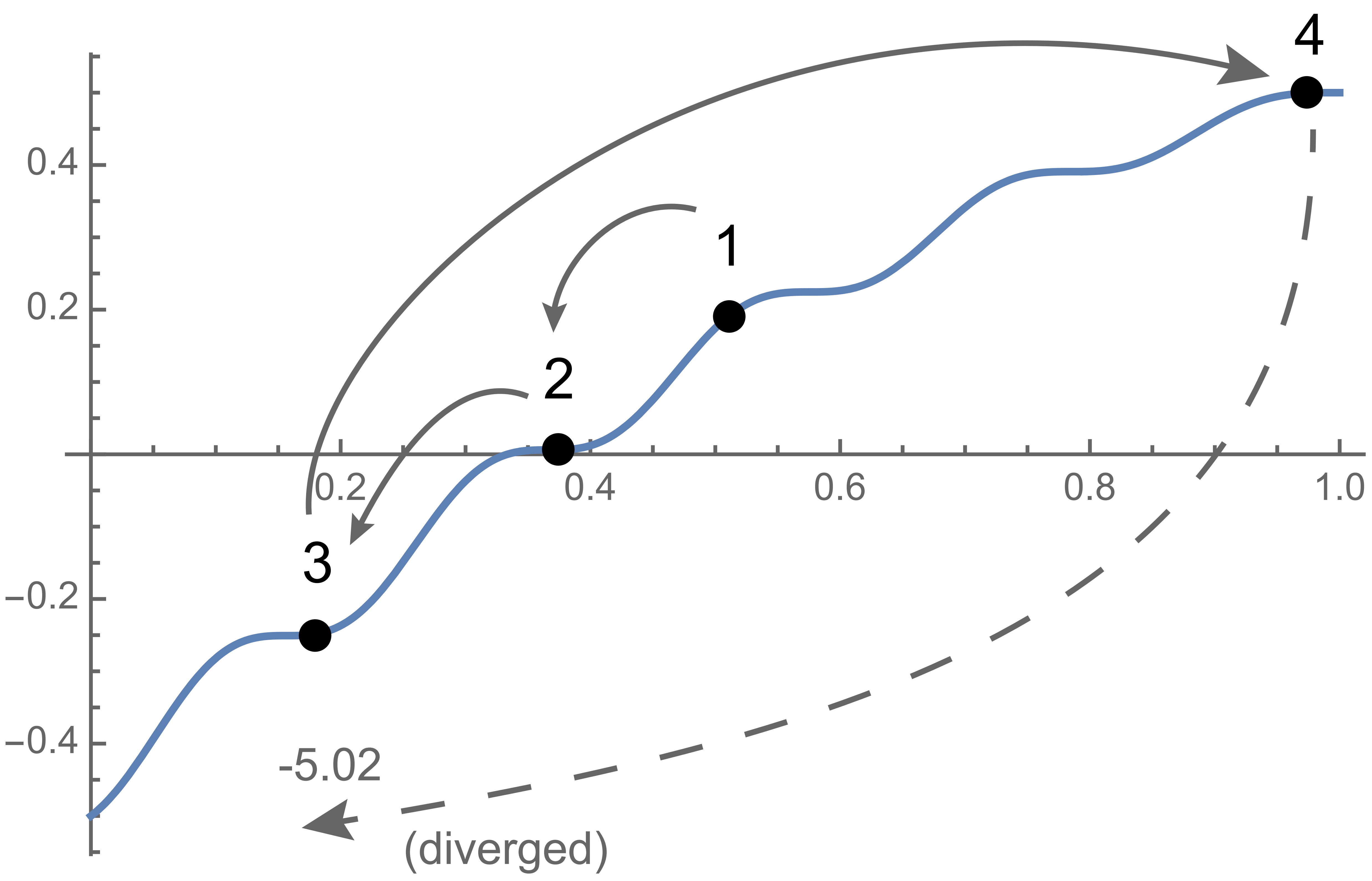
1D Root Finding: Pros/Cons

Property	**Bisection**	**Newton's method**
Speed	😞 Slow	😊 Extremely fast (only a few iterations once we're sufficiently close to root)
Reliability	😊 Always works	😞 Divergence, multiple roots, ...
Requirements	😐 Continuity, bracket	😐 Derivative

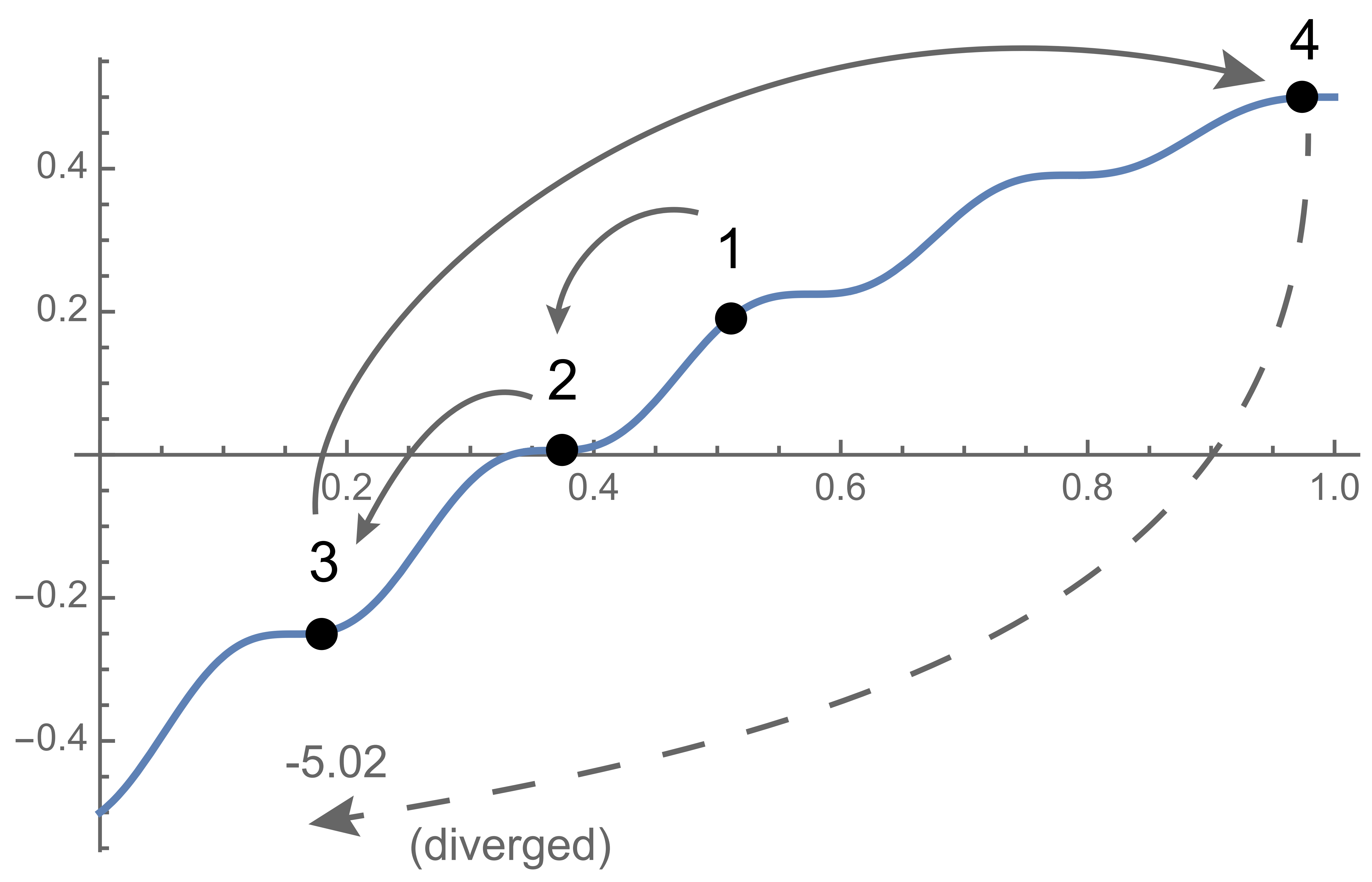
Can combine both: Newton-Bisection

Speed	**Reliability**	**Requirements**
😊 Extremely fast	😊 Always works	😬 Continuity, bracket, derivatives

Failure cases of Newton's Method

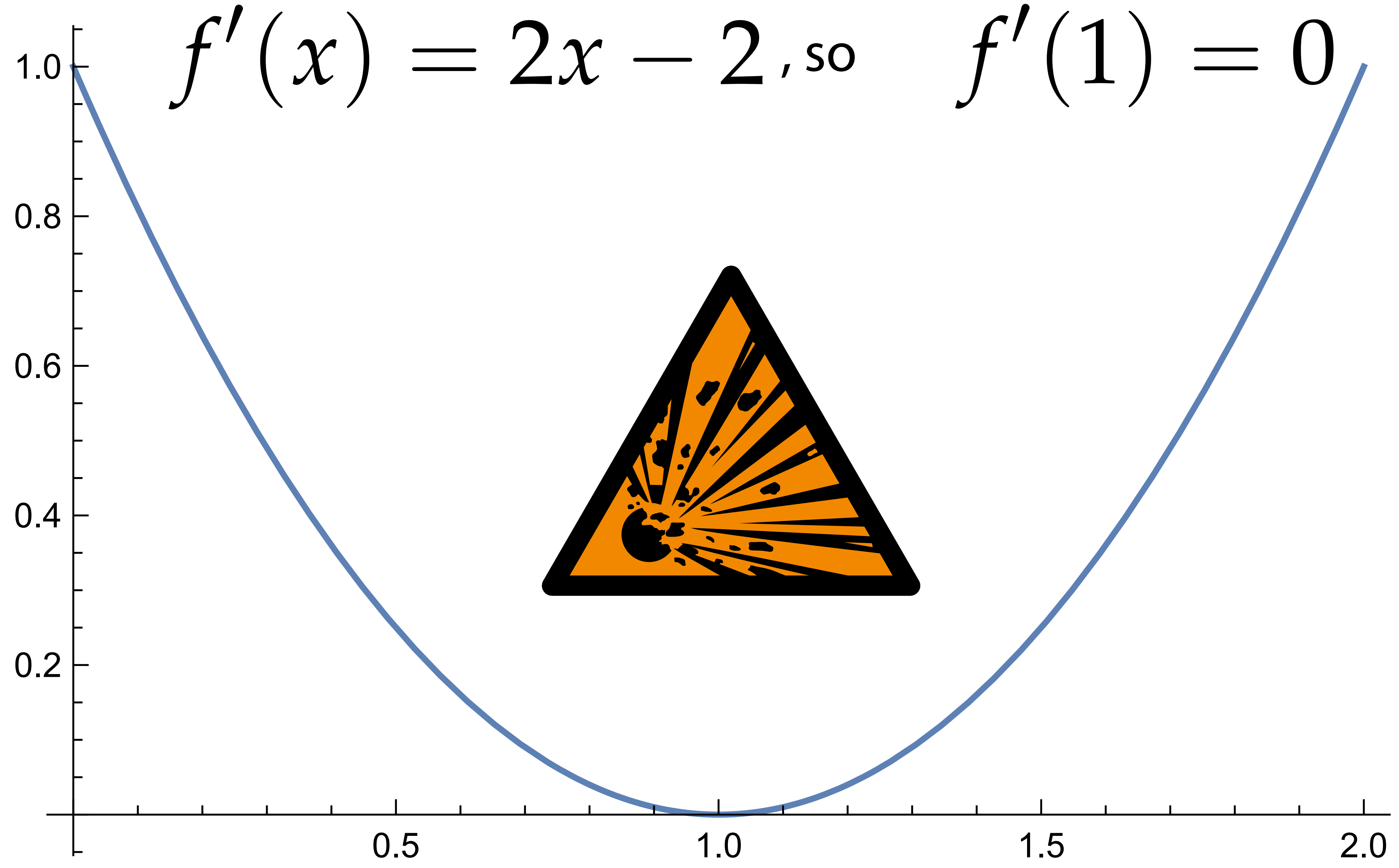


Failure cases of Newton's Method

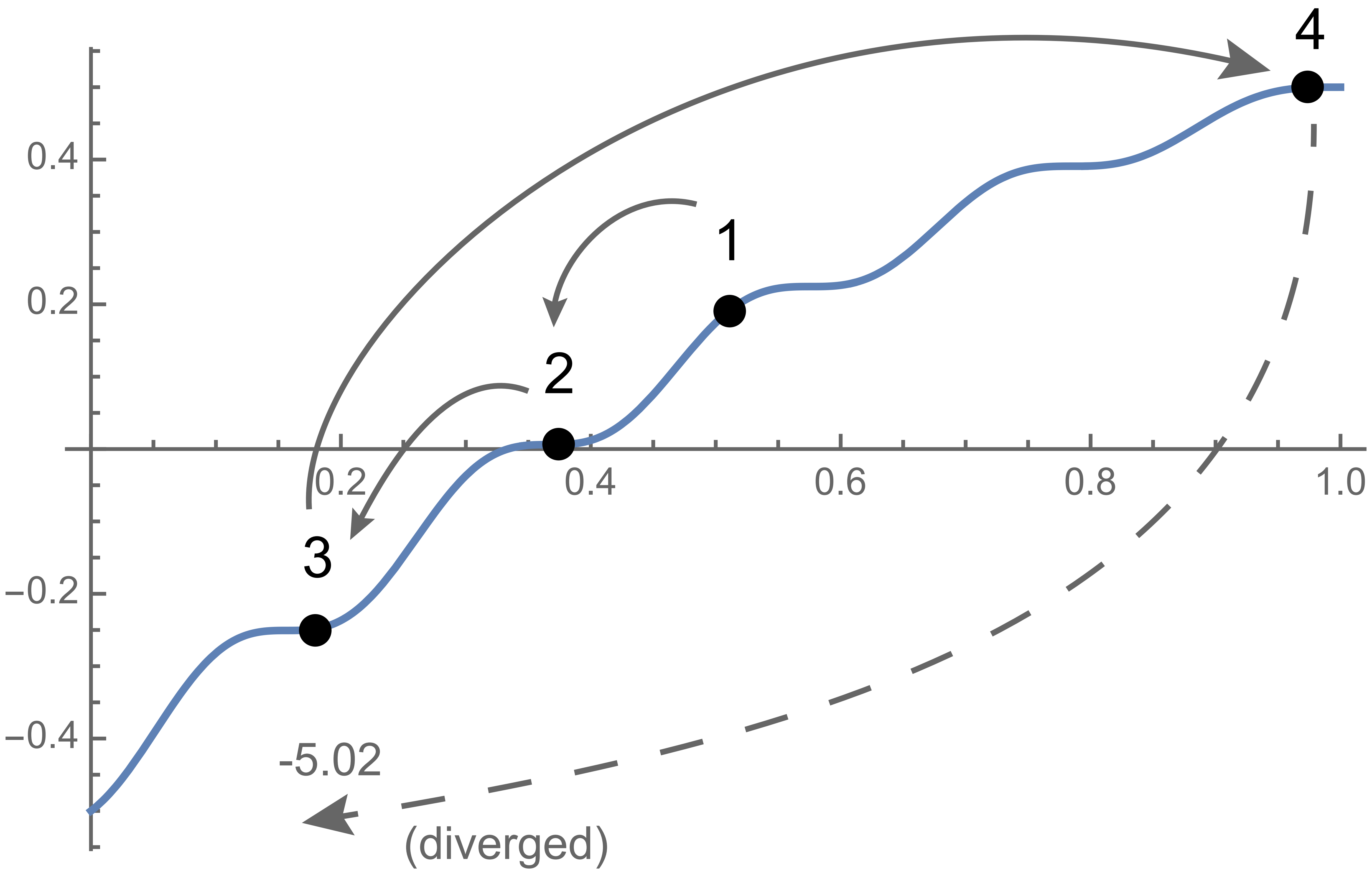


$$f(x) = (x-1)^2$$

$$f'(x) = 2x - 2$$, so $$f'(1) = 0$$

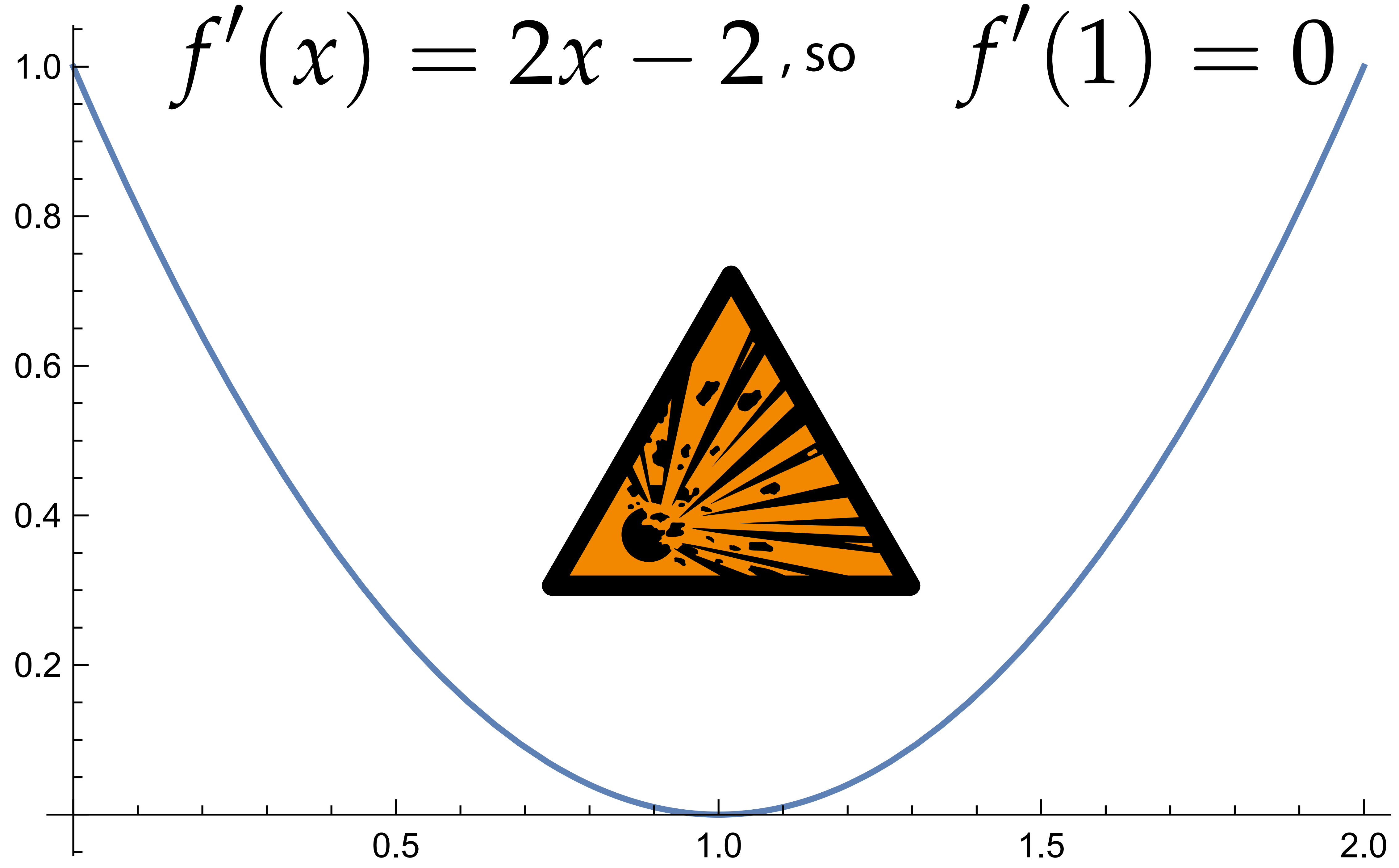


Failure cases of Newton's Method



$$f(x) = (x-1)^2$$

$$f'(x) = 2x - 2, \text{ so } \quad f'(1) = 0$$



In theory: division by zero at $x = 1$.

In practice: slow convergence.

N dimensions

Multivariate root finding

$$f : \mathbb{R}^n \to \mathbb{R}^n$$

Find $\mathbf{x}$ so that $f(\mathbf{x}) = \mathbf{0}$.

High dimensional spaces

- Derivative are crucial especially in **higher dimensions**.
  - In 1-D, can move in two directions
  - in N-D can move in 2^N "diagonal" directions alone. That's just *too many* to check.
  - The gradient points into the direction of ascent and maps the behavior of the function locally.
  - Foundation of all breakthroughts in ML in the last years. You *cannot* train a neural network without gradients.

MidJourney: A sign post with many different hikes in Switzerland.

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

1D case: $f(x+h) \approx f(x) + f'(x)h = 0$

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

1D case: $f(x+h) \approx f(x) + f'(x)h = 0$

N-D case: $\mathbf{f}(\mathbf{x}+\mathbf{h}) \approx \mathbf{f}(\mathbf{x}_{k-1}) + \nabla\mathbf{f}(\mathbf{x}_{k-1})\mathbf{h} = 0$

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

1D case: $f(x+h) \approx f(x) + f'(x)h = 0$

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

N-D case: $\mathbf{f}(\mathbf{x}+\mathbf{h}) \approx \mathbf{f}(\mathbf{x}_{k-1}) + \nabla\mathbf{f}(\mathbf{x}_{k-1})\mathbf{h} = 0$

Newton's method for root finding in N dimensions

This algorithm trivially genaralizes

1D case: $f(x+h) \approx f(x) + f'(x)h = 0$

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

N-D case: $\mathbf{f}(\mathbf{x}+\mathbf{h}) \approx \mathbf{f}(\mathbf{x}_{k-1}) + \nabla\mathbf{f}(\mathbf{x}_{k-1})\mathbf{h} = 0$

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla\mathbf{f}(\mathbf{x}_{k-1})\right]^{-1}\mathbf{f}(\mathbf{x}_{k-1})$$

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..
- No simple+safe hybrid method (e.g. Newton-Bisection) in N-D.

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..
- No simple+safe hybrid method (e.g. Newton-Bisection) in N-D.
- Need to compute Jacobian & solve linear system:
  requires $O(n^3)$ operations *per iteration!!*

Newton's method for root finding in N dimensions

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..
- No simple+safe hybrid method (e.g. Newton-Bisection) in N-D.
- Need to compute Jacobian & solve linear system:
  requires $O(n^3)$ operations *per iteration!!*
  - Linear system solve could fail (ill-conditioned/singular. More dimensions in which things can go wrong..)

Newton's method for root finding in N dimensions

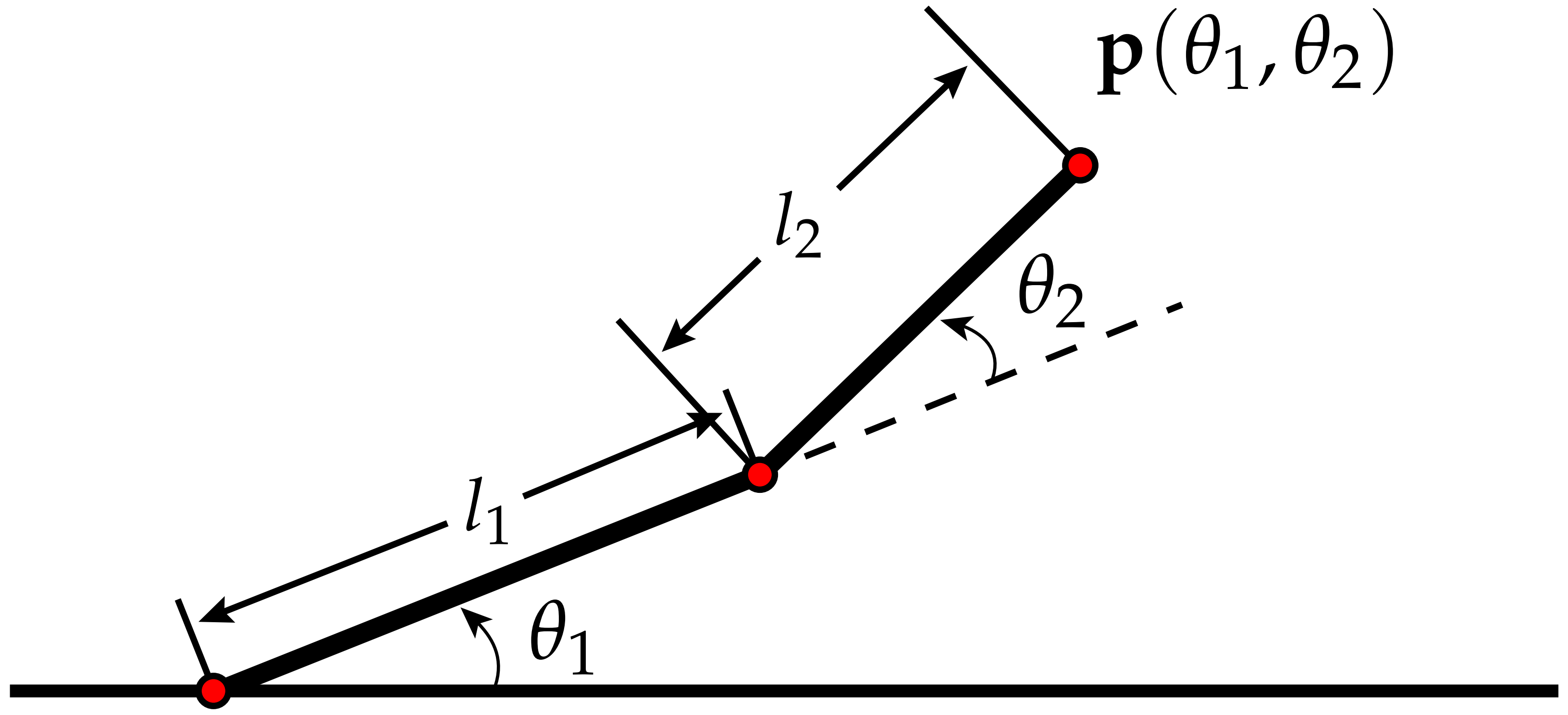
$$\mathbf{x}_k = \mathbf{x}_{k-1} - \left[\nabla \mathbf{f}(\mathbf{x}_{k-1})\right]^{-1} \mathbf{f}(\mathbf{x}_{k-1})$$

Advantages and disadvantages:

- Rapid quadratic convergence, but may diverge..
- No simple+safe hybrid method (e.g. Newton-Bisection) in N-D.
- Need to compute Jacobian & solve linear system:
  requires $O(n^3)$ operations *per iteration!!*
  - Linear system solve could fail (ill-conditioned/singular. More dimensions in which things can go wrong..)
  - Assumes input and output spaces have matching dimension.

HW3: Inverse Kinematics via Newton's method

$$\mathbf{p}(\theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \mathbf{p}_{\text{target}}$$

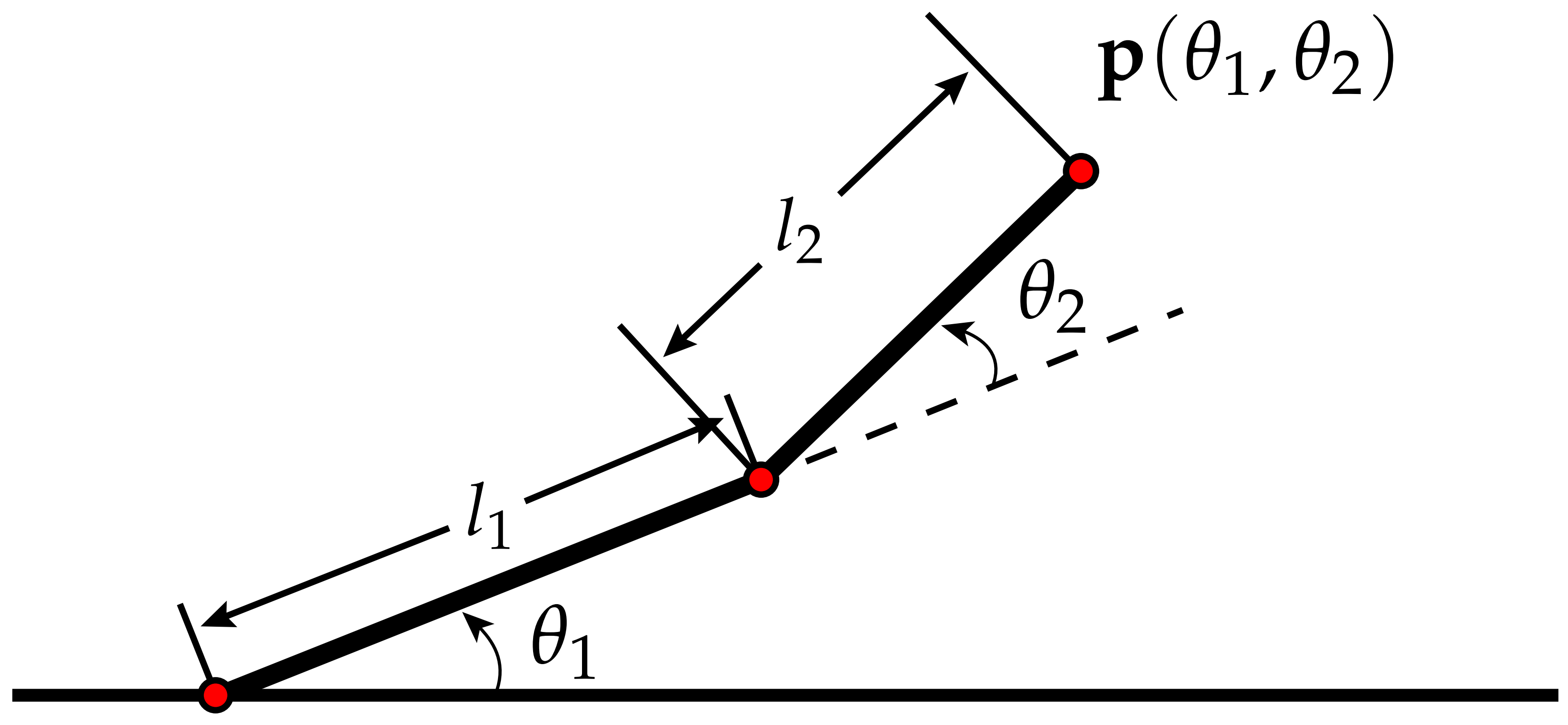


HW3: Inverse Kinematics via Newton's method

$$\mathbf{p}(\theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \mathbf{p}_{\text{target}}$$

Idea: solve with Newton's method

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \left[\nabla \mathbf{p}(\boldsymbol{\theta}_{k-1})\right]^{-1} \mathbf{p}(\boldsymbol{\theta}_{k-1})$$

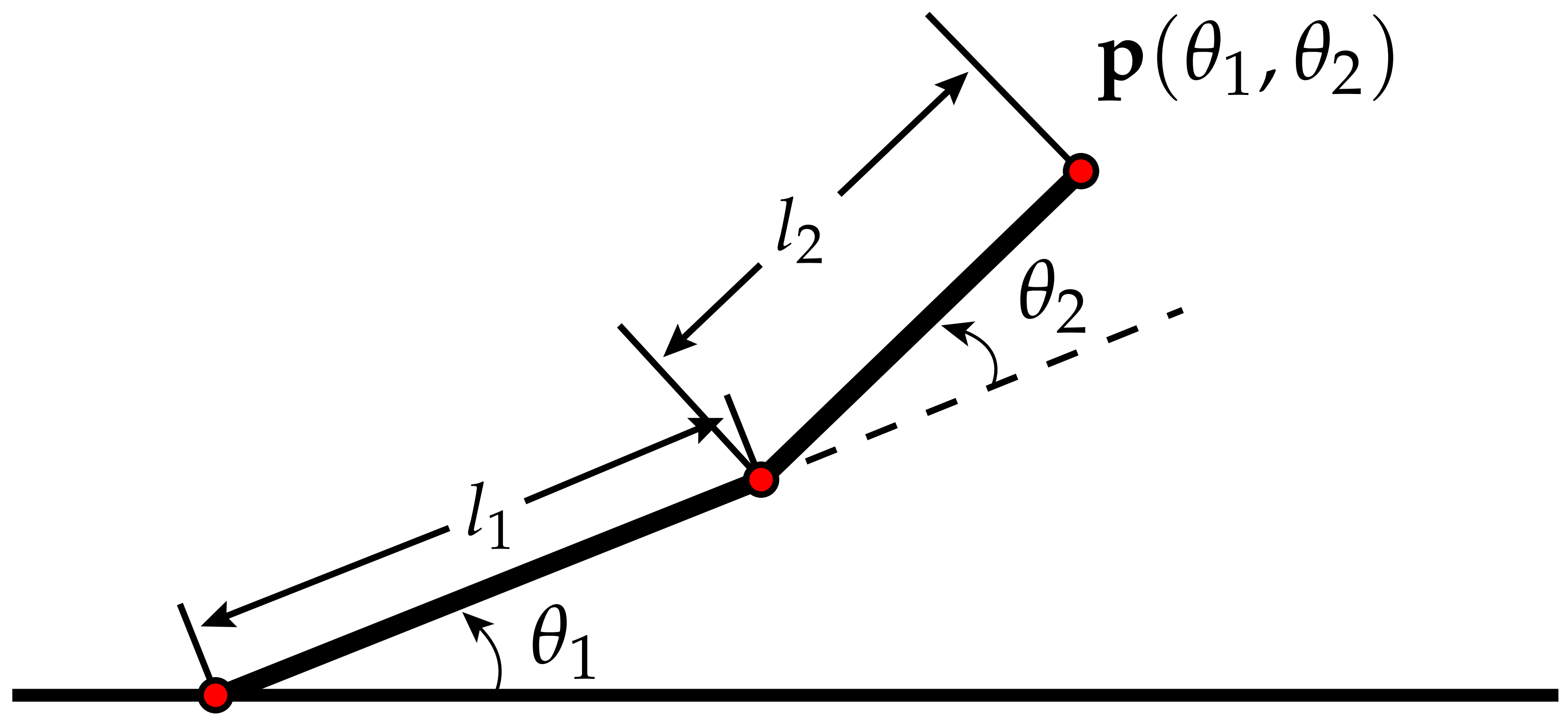


HW3: Inverse Kinematics via Newton's method

n unknowns

$$\mathbf{p}(\theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \mathbf{p}_{\text{target}}$$

2 equations



Idea: solve with Newton's method

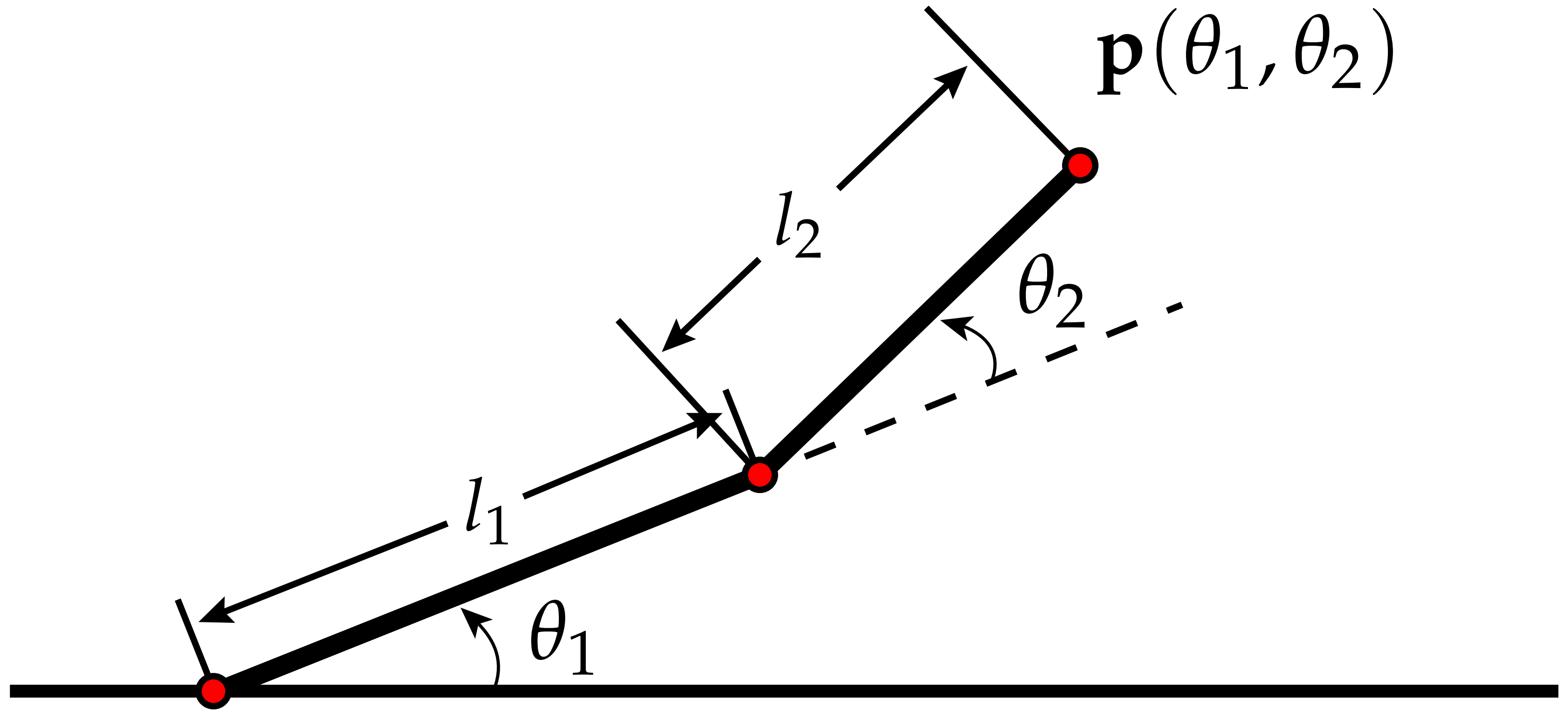
$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - [\nabla \mathbf{p}(\boldsymbol{\theta}_{k-1})]^{-1} \mathbf{p}(\boldsymbol{\theta}_{k-1})$$

HW3: Inverse Kinematics via Newton's method

n unknowns

$$\mathbf{p}(\theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \mathbf{p}_{\text{target}}$$

2 equations



Idea: solve with Newton's method

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \left[\nabla \mathbf{p}(\boldsymbol{\theta}_{k-1})\right]^{-1} \mathbf{p}(\boldsymbol{\theta}_{k-1})$$

Idea 2: solve with Newton's method + pseudoinverse

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \left[\nabla \mathbf{p}(\boldsymbol{\theta}_{k-1})\right]^{+} \mathbf{p}(\boldsymbol{\theta}_{k-1})$$

HW4: MyTorch

- Gradient-based optimization framework.
- A simple neural network with 3 layers (discussed next lecture)
- Reaches ~85% accuracy on Fashion-MNIST
- All built by yourself using only NumPy!

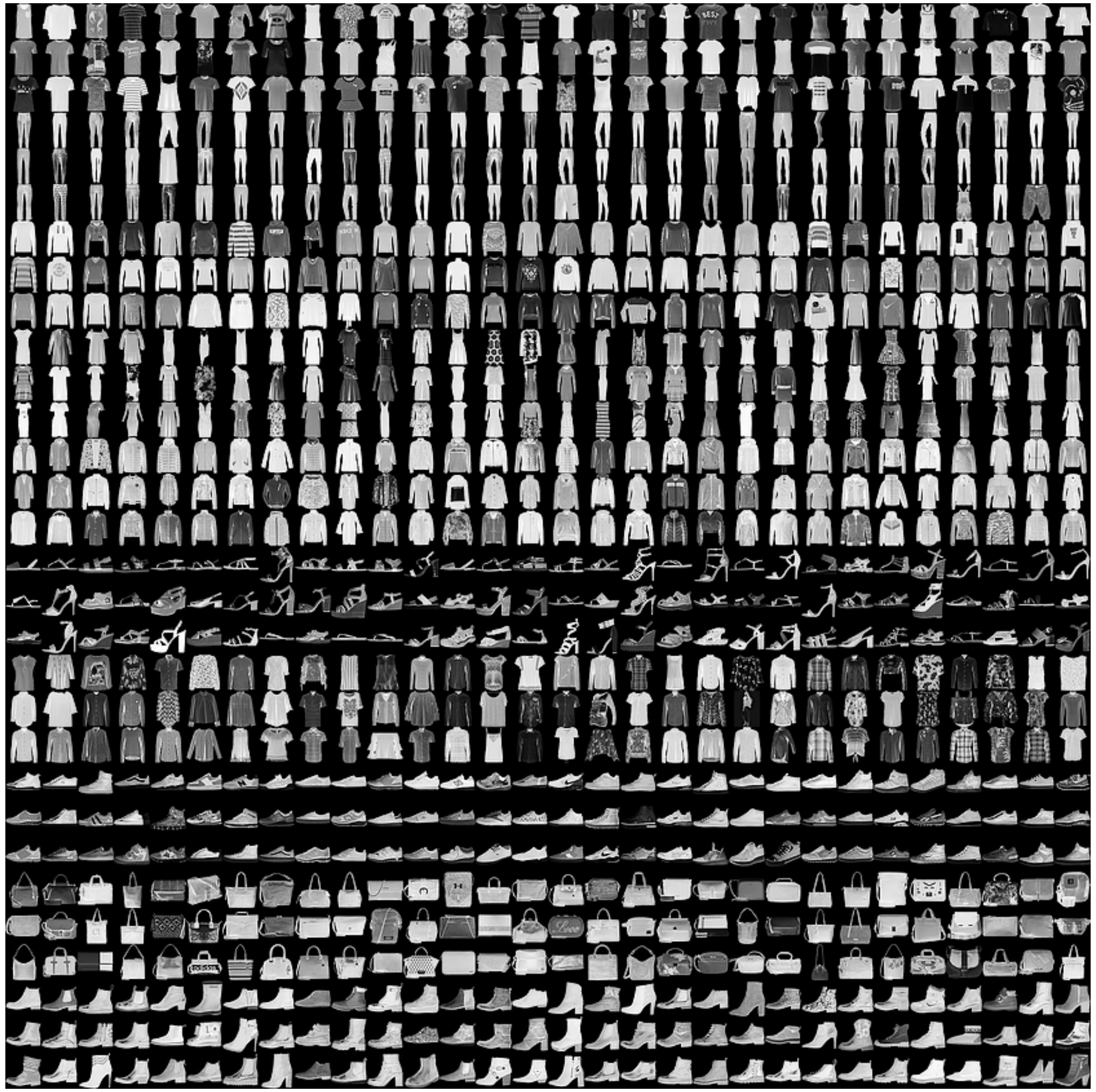
MyTorch

A benchmark problem

"MNIST" and "Fashion-MNIST"

A benchmark problem

"MNIST" and "Fashion-MNIST"



zalando


60'000 training images
10'000 testing images