现代编程思想

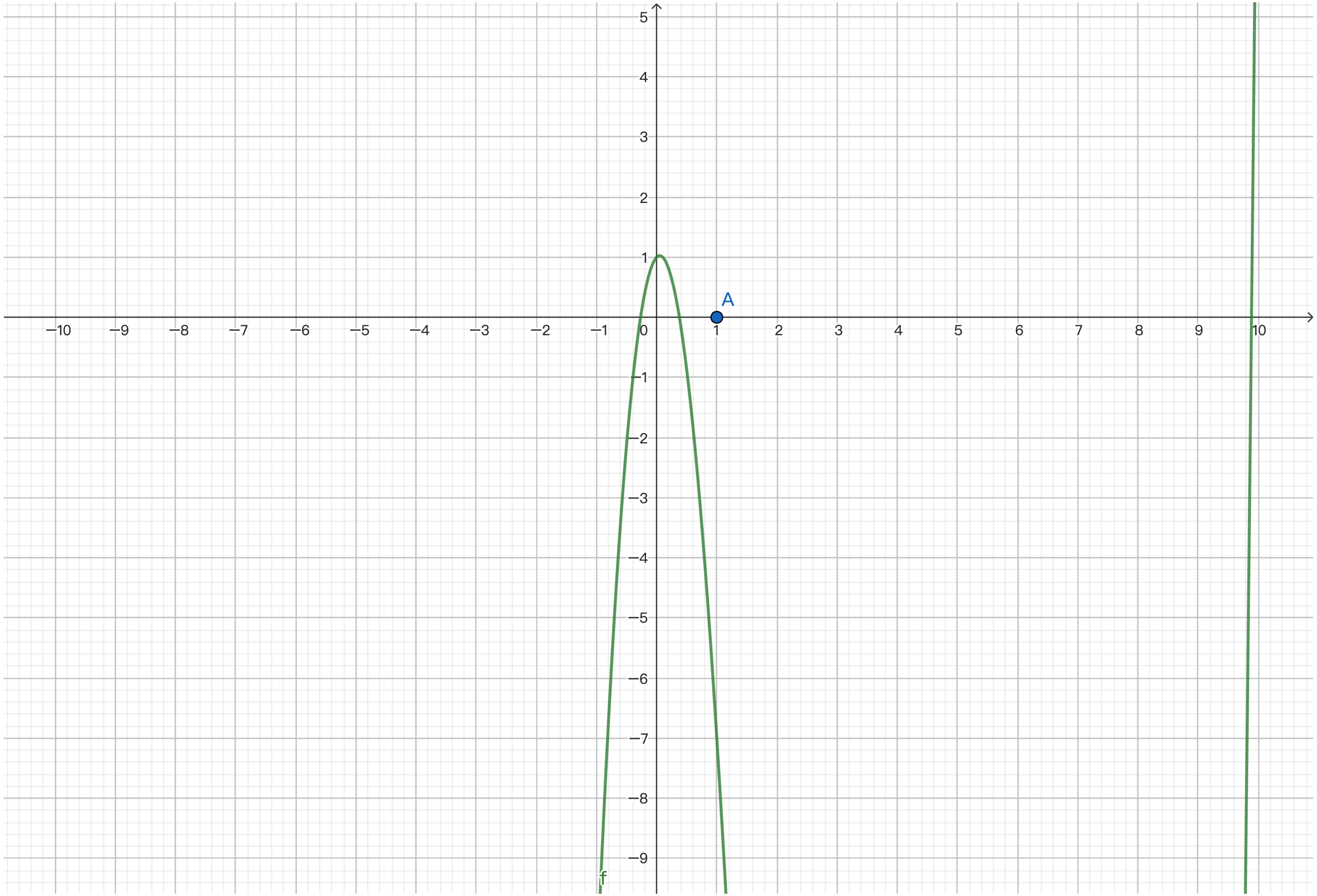
案例：自动微分

Hongbo Zhang

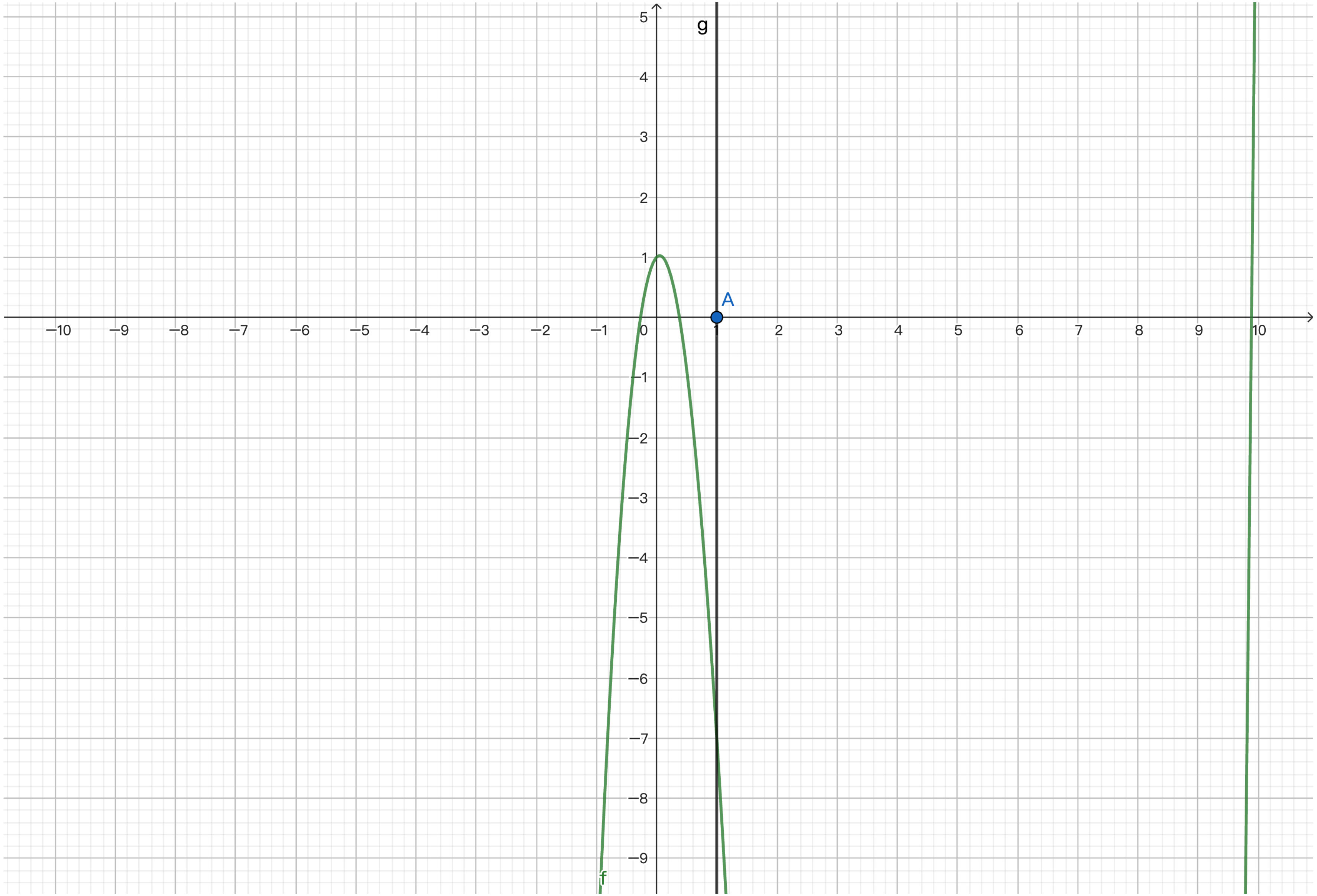
微分

- 微分被应用于机器学习领域
  - 利用梯度下降求局部极值
  - 牛顿迭代法求函数解： $x^3 - 10x^2 + x + 1 = 0$
- 我们今天研究简单的函数组合
  - 例： $f(x_0, x_1) = 5{x_0}^2 + x_1$
    - $f(10, 100) = 600$
    - $\frac{\partial f}{\partial x_0}(10, 100) = 100$
    - $\frac{\partial f}{\partial x_1}(10, 100) = 1$

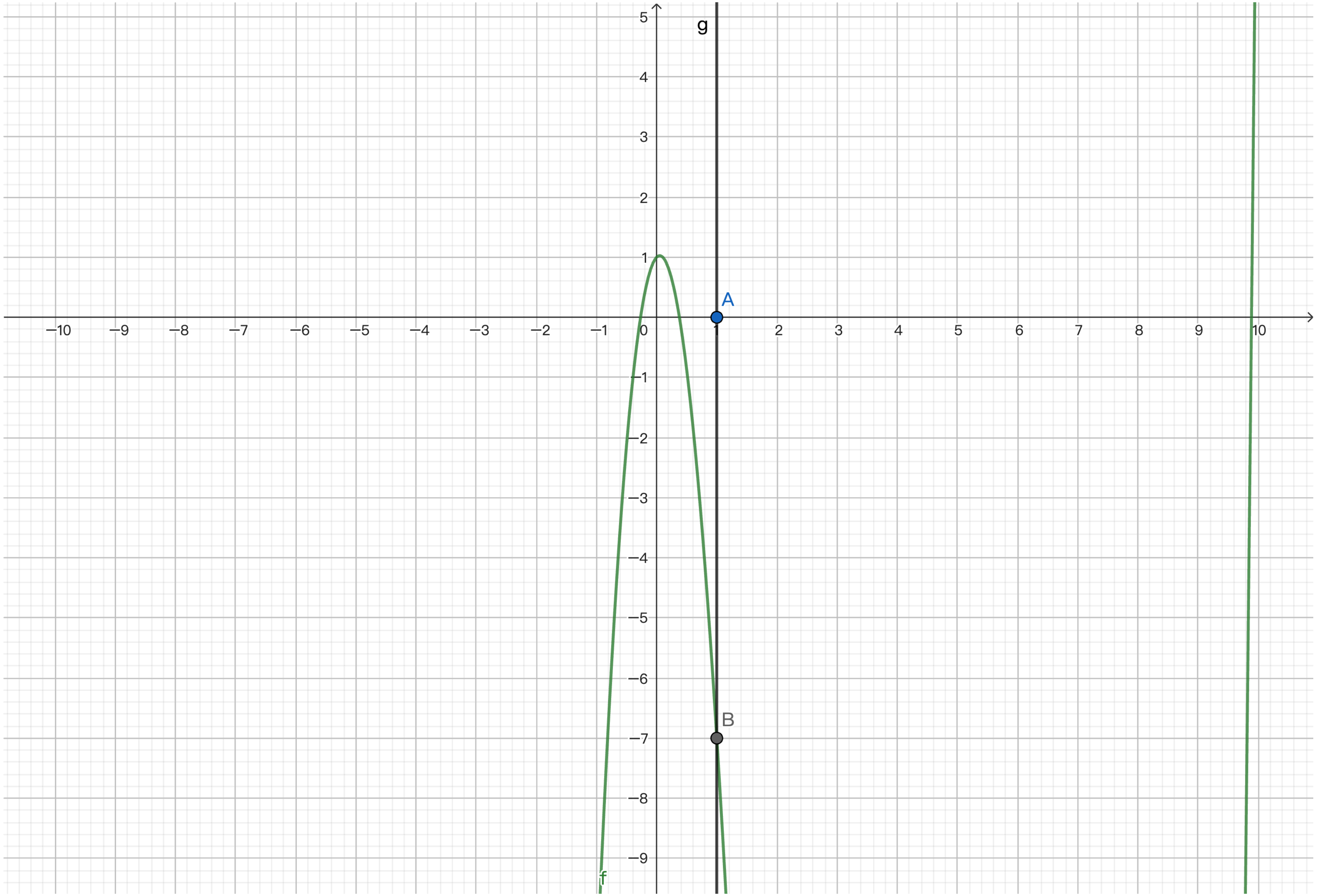
牛顿迭代法



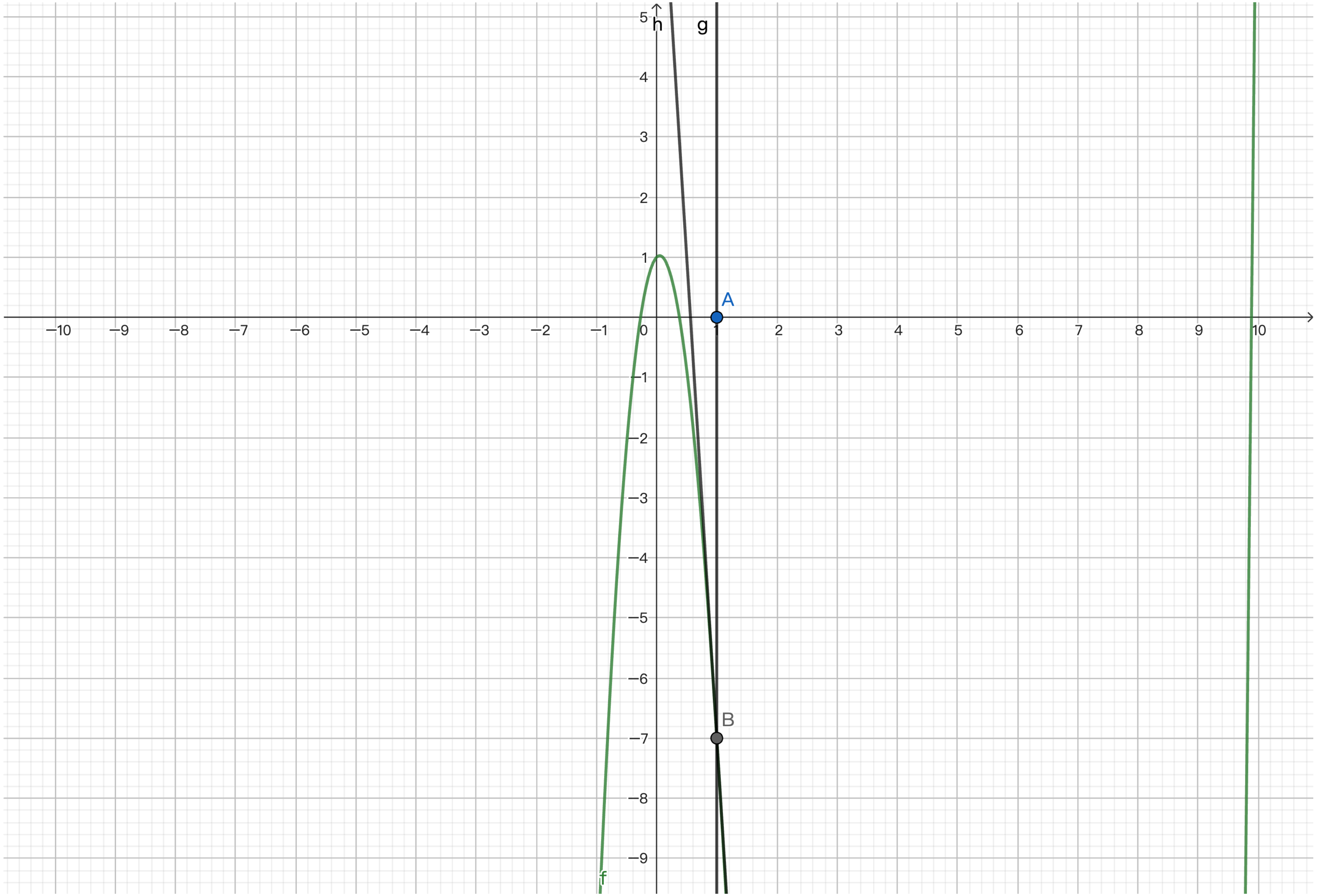
牛顿迭代法



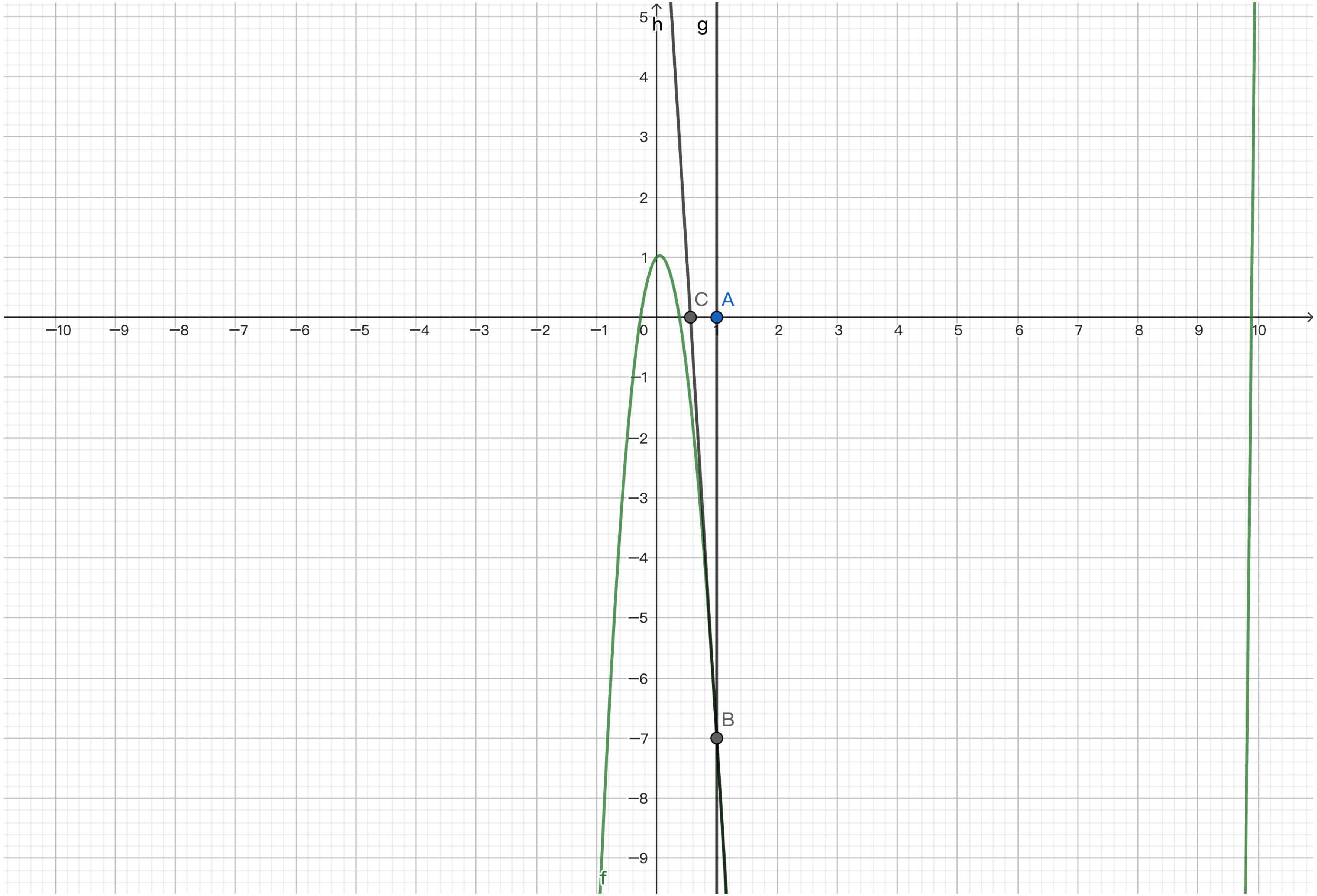
牛顿迭代法



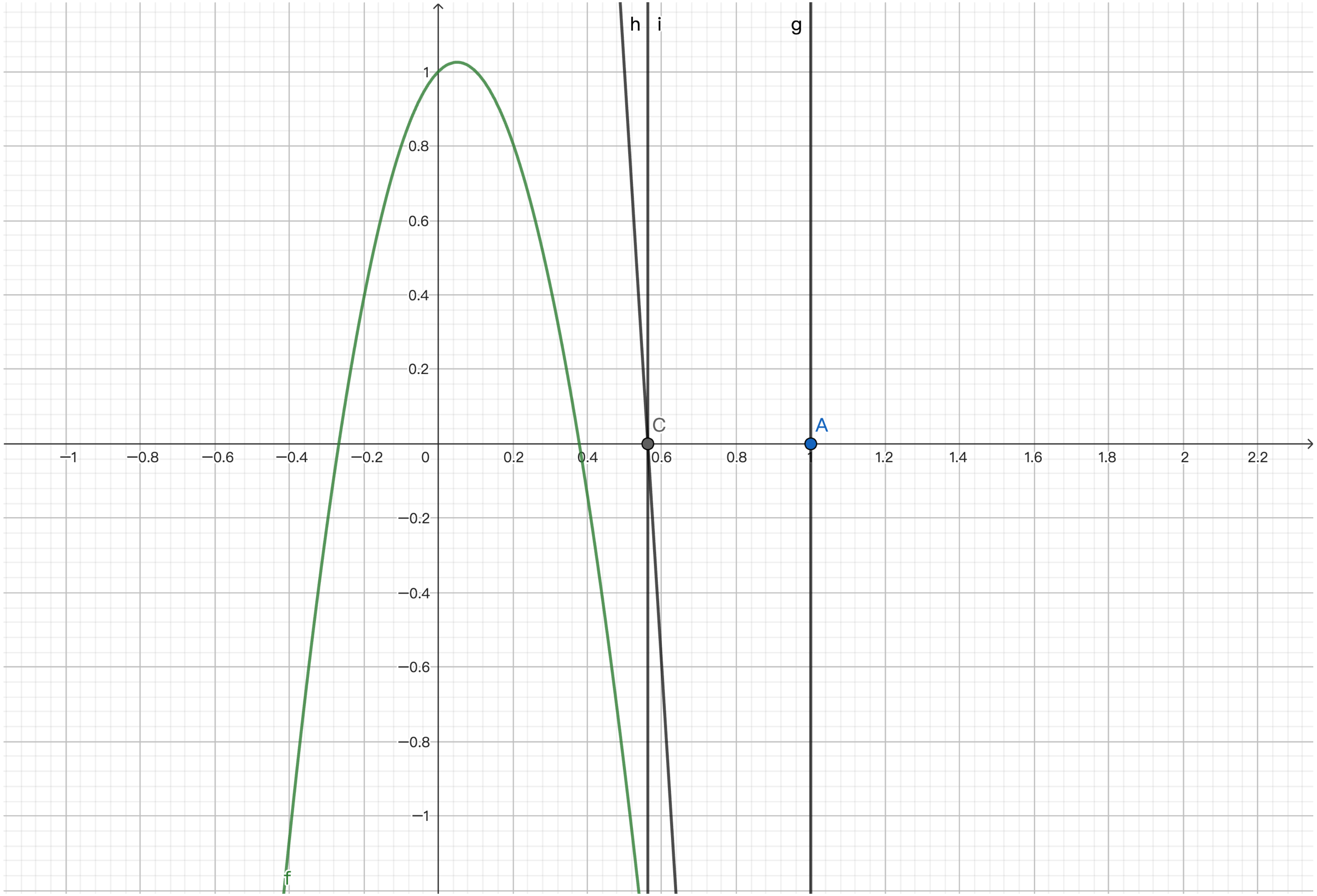
牛顿迭代法



牛顿迭代法

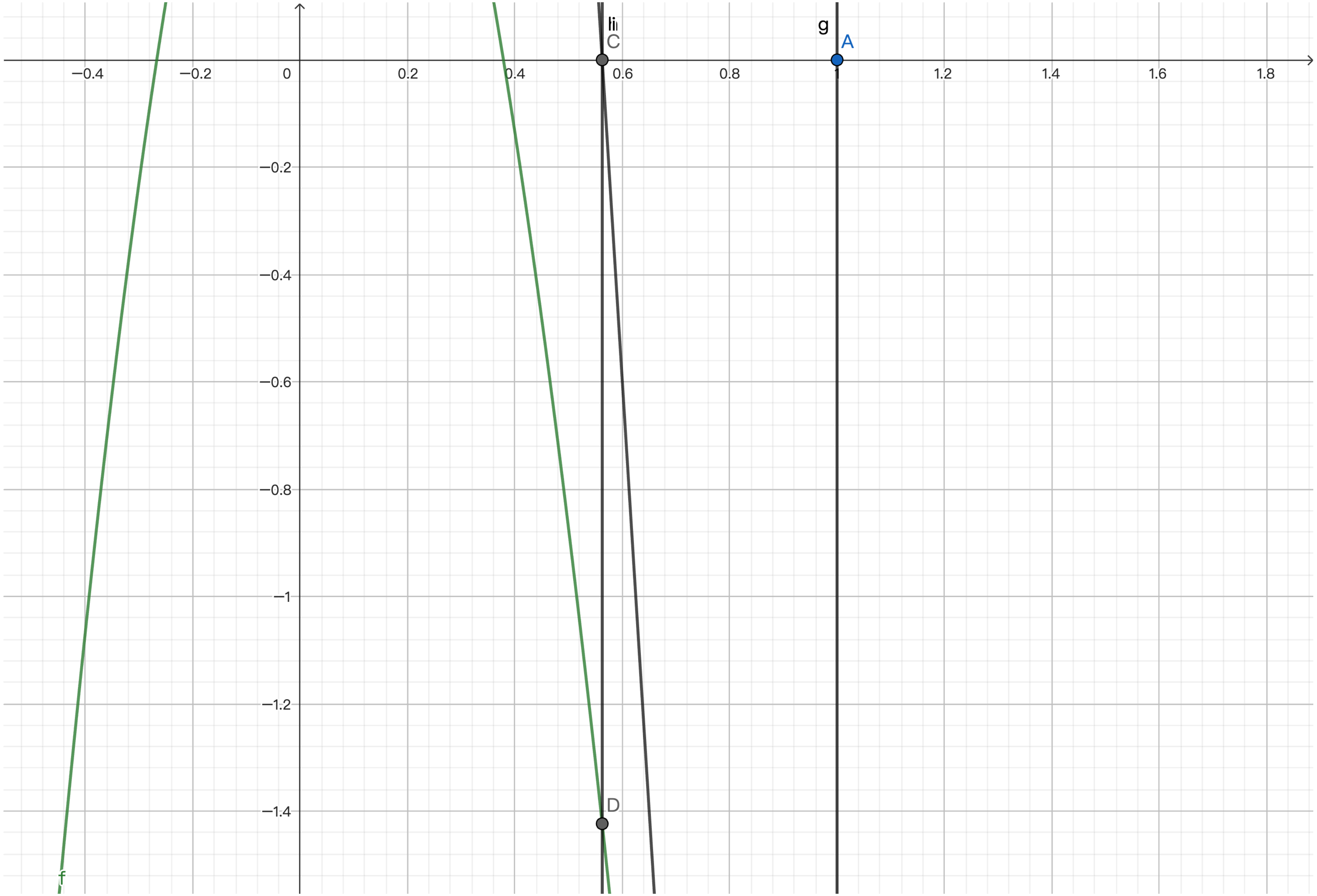


牛顿迭代法

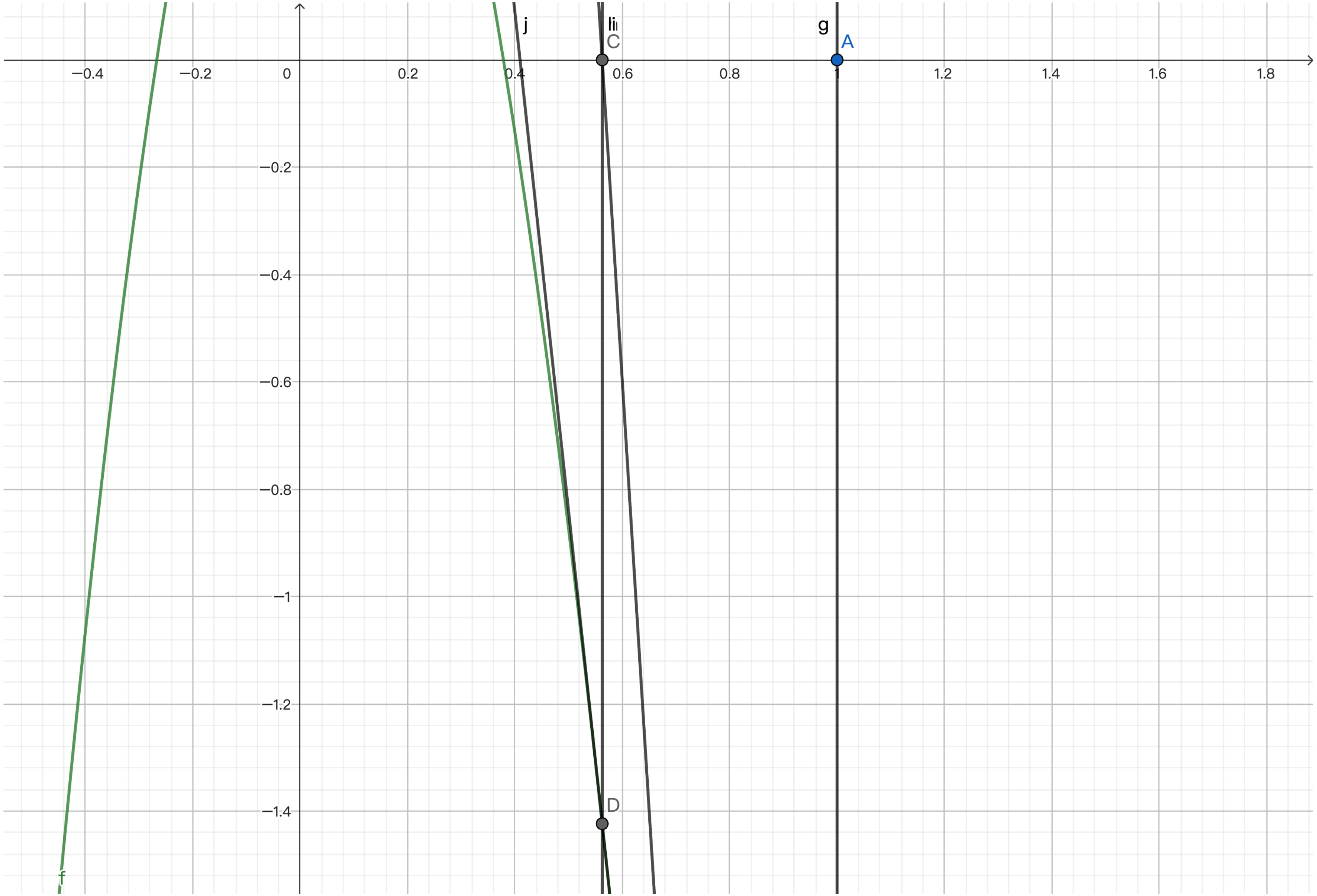


h
i
g
C
A
f
1
0.8
0.6
0.4
0.2
−0.2
−0.4
−0.6
−0.8
−1
−0.8
−0.6
−0.4
−0.2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2

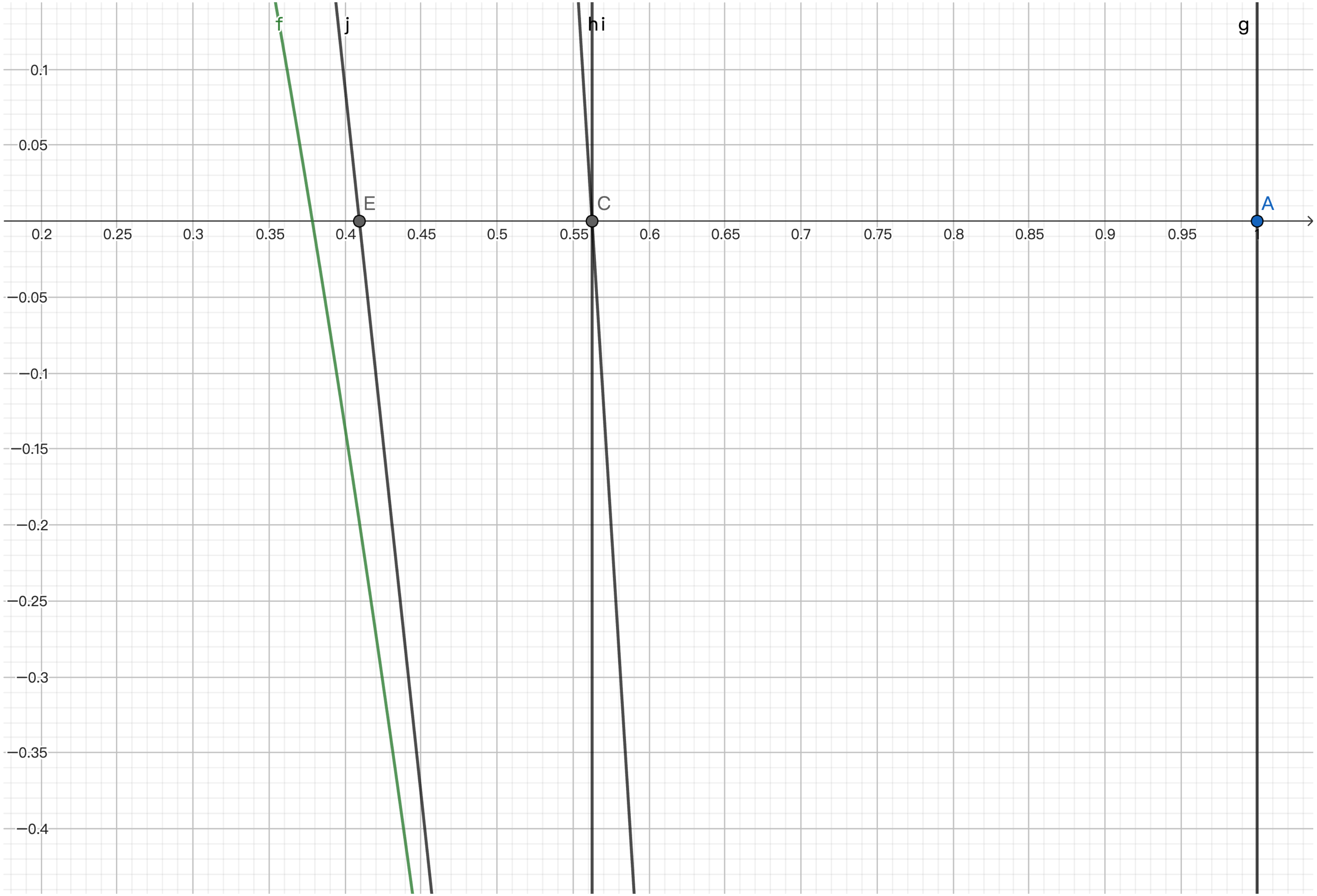
牛顿迭代法



牛顿迭代法



牛顿迭代法


f
j
h i
g
0.1
0.05
E
C
A
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0.55
0.6
0.65
0.7
0.75
0.8
0.85
0.9
0.95
1
−0.05
−0.1
−0.15
−0.2
−0.25
−0.3
−0.35
−0.4

微分

- 微分被应用于机器学习领域
  - 利用梯度下降求局部极值
  - 牛顿迭代法求函数解：$x^3 - 10x^2 + x + 1 = 0$
- 我们今天研究简单的函数组合
  - 例：$f(x_0, x_1) = 5{x_0}^2 + x_1$
    - $f(10, 100) = 600$
    - $\frac{\partial f}{\partial x_0}(10, 100) = 100$
    - $\frac{\partial f}{\partial x_1}(10, 100) = 1$

微分

- 函数微分的几种方式
  - 手动微分：纯天然计算器
    - 缺点：对于复杂表达式容易出错
  - 数值微分：$\frac{\texttt{f}(x+\delta x)-\texttt{f}(x)}{\delta x}$
    - 缺点：计算机无法精准表达小数，且绝对值越大，越不精准
  - 符号微分：`Mul(Const(2), Var(1)) -> Const(2)`
    - 缺点：计算结果可能复杂；可能重复计算；难以直接利用语言原生控制流

```
// 需要额外定义原生算子以实现相同效果
fn max[N : Number](x : N, y : N) -> N {
    if x.value() < y.value() { x } else { y }
}
```

微分

- 函数微分的几种方式
  - 手动微分：纯天然计算器
    - 缺点：对于复杂表达式容易出错
  - 数值微分：$\frac{\texttt{f}(x+\delta x)-\texttt{f}(x)}{\delta x}$
    - 缺点：计算机无法精准表达小数，且绝对值越大，越不精准
  - 符号微分：`Mul(Const(2), Var(1)) -> Const(2)`
    - 缺点：计算结果可能复杂；可能重复计算；难以直接利用语言原生控制流
  - 自动微分：利用复合函数求导法则、由基本运算组合进行微分
    - 分为前向微分和后向微分

符号微分

- 我们以符号微分定义表达式构建的一种语义

```
enum Symbol {
  Constant(Double)
  Var(Int) // x0, x1, x2, ...
  Add(Symbol, Symbol)
  Mul(Symbol, Symbol)
} derive(Debug)

// 定义简单构造器，并重载运算符
fn Symbol::constant(d : Double) -> Symbol { Constant(d) }
fn Symbol::var(i : Int) -> Symbol { Var(i) }
fn Symbol::op_add(f1 : Symbol, f2 : Symbol) -> Symbol { Add(f1, f2) }
fn Symbol::op_mul(f1 : Symbol, f2 : Symbol) -> Symbol { Mul(f1, f2) }

// 计算函数值
fn Symbol::compute(f : Symbol, input : Array[Double]) -> Double { ... }
```

符号微分

- 利用函数求导法则，我们计算函数的（偏）导数
  - $\frac{\partial f}{\partial x_i} = 0$ 如果 f 为常值函数
  - $\frac{\partial x_i}{\partial x_i} = 1, \frac{\partial x_j}{\partial x_i} = 0, i \neq j$
  - $\frac{\partial (f+g)}{\partial x_i} = \frac{\partial f}{\partial x_i} + \frac{\partial g}{\partial x_i}$
  - $\frac{\partial (f \times g)}{\partial x_i} = \frac{\partial f}{\partial x_i} \times g + f \times \frac{\partial g}{\partial x_i}$
- 月兔实现

```
fn differentiate(self : Symbol, val : Int) -> Symbol {
  match self {
    Constant(_) => Constant(0.0)
    Var(i) => if i == val { Constant(1.0) } else { Constant(0.0) }
    Add(f1, f2) => f1.differentiate(val) + f2.differentiate(val)
    Mul(f1, f2) => f1 * f2.differentiate(val) + f1.differentiate(val) * f2
  }
}
```

符号微分

- 利用符号微分，先构建抽象语法树，再转换为对应的微分，最后进行计算

```
fn example() -> Symbol {
  Symbol::constant(5.0) * Symbol::var(0) * Symbol::var(0) + Symbol::var(1)
}
fn init {
  let input : Array[Double] = [10., 100.]
  let func : Symbol = example() // 函数的抽象语法树
  let diff_0_func : Symbol = func.differentiate(0) // 对x_0的偏微分
  let _ = diff_0_func.compute(input)
}
```

- 其中，`diff_0` 为

```
let diff_0: Symbol =
  (Symbol::Constant(5.0) * Var(0)) * Constant(1.0) +
  (Symbol::Constant(5.0) * Constant(1.0) + Symbol::Constant(0.0) * Var(0)) * Var(0) +
  Constant(0.0)
```

符号微分

- 我们可以在构造期间进行化简

```
fn Symbol::op_add(f1 : Symbol, f2 : Symbol) -> Symbol {
  match (f1, f2) {
    (Constant(0.0), a) => a // 0 + a = a
    (Constant(a), Constant(b)) => Constant(a * b)
    (a, Constant(_) as const) => const + a
    _ => Add(f1, f2)
} }
```

符号微分

- 我们可以在构造期间进行化简

```
fn Symbol::op_mul(f1 : Symbol, f2 : Symbol) -> Symbol {
  match (f1, f2) {
    (Constant(0.0), _) => Constant(0.0) // 0 * a = 0
    (Constant(1.0), a) => a             // 1 * a = 1
    (Constant(a), Constant(b)) => Constant(a * b)
    (a, Constant(_) as const) => const * a
    _ => Mul(f1, f2)
} }
```

- 化简效果

```
let diff_0 : Symbol = Mul(Constant(5.0), Var(0))
```

自动微分

MoonBit 月兔编程

- 通过接口定义我们想要实现的运算

```
trait Number  {
  constant(Double) -> Self
  op_add(Self, Self) -> Self
  op_mul(Self, Self) -> Self
  value(Self) -> Double // 获取当前计算值
}
```

- 可以利用语言原生的控制流计算，动态生成计算图

```
fn max[N : Number](x : N, y : N) -> N {
  if x.value() > y.value() { x } else { y }
}
fn relu[N : Number](x : N) -> N {
  max(x, N::constant(0.0))
}
```

前向微分

- 利用求导法则直接计算微分，同时计算$f(a)$与$\frac{\partial f}{\partial x_i}(a)$
  - 简单理解：计算$(fg)' = f' \times g + f \times g'$需要同时计算$f$与$f'$
  - 专业术语：线性代数中的二元数（Dual Number）

```
struct Forward {
  value : Double      // 当前节点值   f
  derivative : Double // 当前节点微分 f'
} derive(Debug)

fn Forward::constant(d : Double) -> Forward { { value: d, derivative: 0.0 } }
fn Forward::value(f : Forward) -> Double { f.value }

// 是否对当前变量进行微分
fn Forward::var(d : Double, diff : Bool) -> Forward {
  { value : d, derivative : if diff { 1.0 } else { 0.0 } }
}
```

前向微分

- 利用求导法则直接计算微分

```
fn Forward::op_add(f : Forward, g : Forward) -> Forward { {
  value : f.value + g.value,
  derivative : f.derivative + g.derivative // f' + g'
} }

fn Forward::op_mul(f : Forward, g : Forward) -> Forward { {
  value : f.value * g.value,
  derivative : f.value * g.derivative + g.value * f.derivative // f * g' + g * f'
} }
```

前向微分

- 对输入的参数需逐个计算微分，适用于输出参数多于输入参数

```
relu(Forward::var(10.0, true))  |> debug // {value: 10.0, derivative: 1.0}
relu(Forward::var(-10.0, true)) |> debug // {value: 0.0, derivative: 0.0}
// d(x * y) / dy : {value: 1000.0, derivative: 10.0}
Forward::var(10.0, false) * Forward::var(100.0, true) |> debug
```

案例：牛顿迭代法求零点

- $f = x^3 - 10x^2 + x + 1$

```
fn example_newton[N : Number](x : N) -> N {
  x * x * x + N::constant(-10.0) * x * x + x + N::constant(1.0)
}
```

案例：牛顿迭代法求零点

- 通过循环进行迭代
  - $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

```
fn init {
  fn abs(d : Double) -> Double { if d >= 0.0 { d } else { -d } }
  loop Forward::var(1.0, true) {  // 迭代起点
    x => {
      let { value, derivative } = example_newton(x)
      if abs(value / derivative) < 1.0e-9 {
        break x.value // 精度足够，终止循环
      }
      continue Forward::var(x.value - value / derivative, true)
    }
  } |> debug // 0.37851665401644224
}
```

后向微分

- 利用链式法则
  - 若有$w = f(x, y, z, \cdots), x = x(t), y = y(t), z = z(t), \cdots$，那么 $\frac{\partial w}{\partial t} = \frac{\partial w}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial w}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial w}{\partial z}\frac{\partial z}{\partial t} + \cdots$
  - 例如：$f(x_0, x_1) = {x_0}^2 x_1$
    - 分解：$f = gh, g(x_0, x_1) = {x_0}^2, h(x_0, x_1) = x_1$
    - 微分：$\frac{\partial f}{\partial g} = h = x_1, \frac{\partial g}{\partial x_0} = 2x_0, \frac{\partial f}{\partial h} = g = {x_0}^2, \frac{\partial h}{\partial x_0} = 0$
    - 组合：$\frac{\partial f}{\partial x_0} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x_0} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x_0} = x_1 \times 2x_0 + {x_0}^2 \times 0 = 2x_0x_1$
- 从$\frac{\partial f}{\partial f}$开始，向后计算中间过程的偏微分$\frac{\partial f}{\partial g_i}$，直至输入参数的微分$\frac{\partial g_i}{\partial x_i}$
  - 可以同时求出每一个输入的偏微分，适用于输入参数多于输出参数

后向微分

- 需前向计算，再后向计算微分

```
struct Backward {
  value : Double                   // 当前节点计算值
  backward : (Double) -> Unit // 对当前子表达式微分并累加
}

fn Backward::var(value : Double, diff : Ref[Double]) -> Backward {
  // 更新一条计算路径的偏微分 df / dvi * dvi / dx
  { value, backward: fn { d => diff.val = diff.val + d } }
}

fn Backward::constant(d : Double) -> Backward {
  { value: d, backward: fn { _ => () } }
}

fn Backward::backward(b : Backward, d : Double) { (b.backward)(d) }

fn Backward::value(backward : Backward) -> Double { backward.value }
```

后向微分

- 需前向计算，再后向计算微分
  - $f = g + h, \frac{\partial f}{\partial g} = 1, \frac{\partial f}{\partial h} = 1$
  - $f = g \times h, \frac{\partial f}{\partial g} = h, \frac{\partial f}{\partial h} = g$
  - 经过f, g: $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial f}\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$，其中$\frac{\partial y}{\partial f}$对应`diff`

```
fn Backward::op_add(g : Backward, h : Backward) -> Backward {
  {
    value: g.value + h.value,
    backward: fn(diff) { g.backward(diff * 1.0); h.backward(diff * 1.0) },
  }
}

fn Backward::op_mul(g : Backward, h : Backward) -> Backward {
  {
    value: g.value * h.value,
    backward: fn(diff) { g.backward(diff * h.value); h.backward(diff * g.value) },
  }
}
```

后向微分

```
fn init {
  let diff_x = Ref::{ val: 0.0 } // 存储x的微分
  let diff_y = Ref::{ val: 0.0 } // 存储y的微分

  let x = Backward::var(10.0, diff_x)
  let y = Backward::var(100.0, diff_y)

  (x * y).backward(1.0) // df / df = 1.0

  debug(diff_x) // { val : 100.0 }
  debug(diff_y) // { val : 10.0 }
}
```

总结

- 本章节介绍了自动微分的概念
  - 展示了符号微分
  - 展示了前向微分与后向微分
- 拓展阅读
  - 3Blue1Brown：深度学习系列（梯度下降法、反向传播算法）