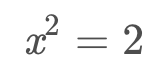
# **利用梯度下降法和牛顿法求开方**

假如现在要求的是根号2

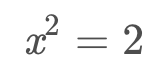
即 的解

使用梯度下降法

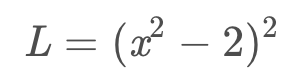
梯度下降主要通过求梯度为0的点，得到凸函数的全局最小值。

首先构建损失函数为凸函数的目标函数, 使得目标函数的最小值对应的是我们要求的值

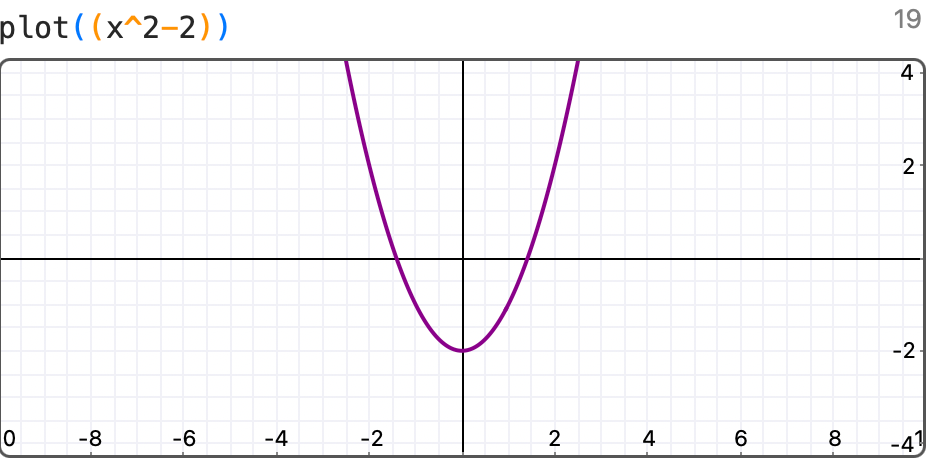
由

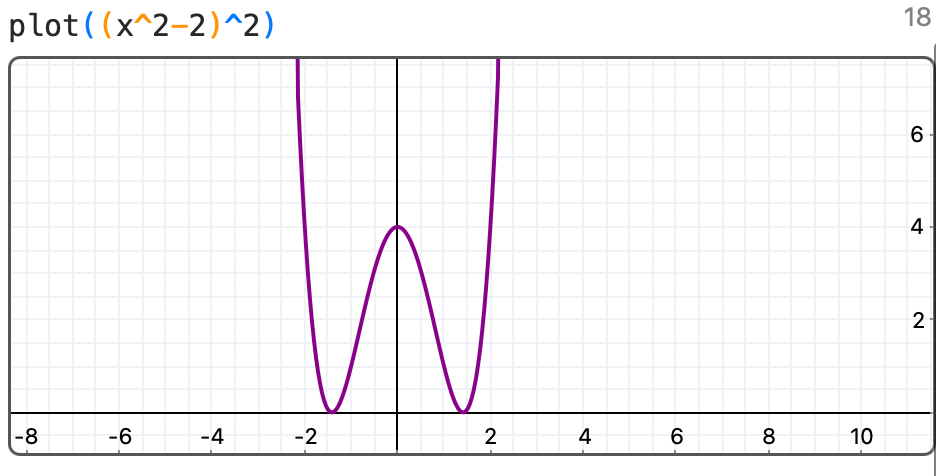


我们可以构建一个损失函数

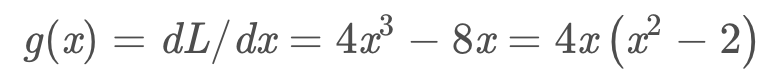


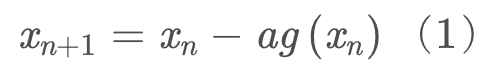
这里为什么不直接用  是因为其最小值对应的x不是我们要求的x.





构建了损失函数以后，就可以用梯度下降法来求全局最优的x的值，即为我们要求的根号2的值。





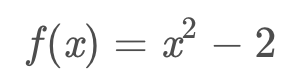
给x一个初始值，然后不断通过1式来更新x就可以逐渐逼近最优的x

*# -\*- coding:utf-8 -\*-*import random as rand  
import matplotlib.pyplot as plt  
  
def gradient\_descent(n):  
 x = float(rand.randint(1, 100))  
 array = []  
 while (abs(x \*\* 2 - n) > 0.0001):  
 x = x - 0.00001 \* 4 \* x \* (x \*\* 2 - n)  
 array.append(x)  
 return array  
n=2  
array = gradient\_descent(n)  
x = range(len(array))  
plt.plot(x, array, color=**'b'**)  
plt.show()  
import math  
print(math.sqrt(n), array[-1])

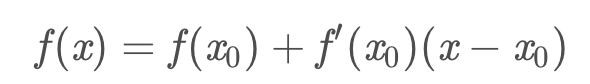
#### **使用牛顿法**

牛顿法的一大用处就是求方程的根。

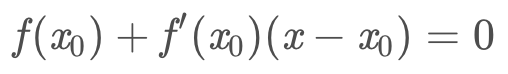
对于方程



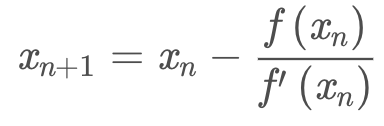
f(x)进行一阶泰勒展开后得



另f(x) 等于0得



可推出



这样就得到的x的迭代式

*# -\*- coding:utf-8 -\*-*import random  
import matplotlib.pyplot as plt  
import math  
def Newton(n):  
 array = []  
 x = float(random.randint(1, 100))  
 while(abs(x\*\*2 - n) > 0.000000001):  
 x = x - (x\*\*2 - n ) / (2\*x)  
 array.append(x)  
 return array  
n=2  
array = Newton(n)  
x = range(len(array))  
plt.plot(x, array, color=**'b'**)  
plt.show()  
print(math.sqrt(n), array[-1])

# **二分法、牛顿法和梯度下降法开根号和解方程**

## **二分法**

二分法的思路是每次排除一半样本的试错方法，把样本一分为二（A和B），那么目标值不在A就在B里，不断缩小范围。

就像在玩一个猜价格的游戏，通过告诉你猜高了还是低了，你总能猜到正确价格一样，设计好一个计算差值的函数能大体上判断出你下一轮尝试的值应该在前一半还是后一半，总能迭代到足够接近的结果。

对于求平方根来说，我们没什么过多的设计，直接对中值取平方，高了就取小的一半，低了就取大的一半，实测小的数字是没问题的，这里仅仅用来演示思路。

import math

def binary\_sqrt(n):

epsilon = 1e-10 # quit flag

start = 0

end = n

mid = start + (end - start) / 2

diff = mid \*\* 2 - n

while abs(diff) >= epsilon:

# 值过大则尝试小的一半，否则就尝试大的一半，修改边界值即可

if diff > 0:

end = mid

else:

start = mid

mid = start + (end - start) / 2

diff = mid \*\* 2 - n

return mid

for i in range(1,11):

print(f'estimated:\t{binary\_sqrt(i)}, \t sqrt({i}): \t {math.sqrt(i)}')

output:

estimated: 0.9999999999708962, sqrt(1): 1.0

estimated: 1.4142135623842478, sqrt(2): 1.4142135623730951

estimated: 1.7320508075645193, sqrt(3): 1.7320508075688772

estimated: 2.0000000000000000, sqrt(4): 2.0

estimated: 2.2360679775010794, sqrt(5): 2.23606797749979

estimated: 2.4494897427794060, sqrt(6): 2.449489742783178

estimated: 2.6457513110653963, sqrt(7): 2.6457513110645907

estimated: 2.8284271247393917, sqrt(8): 2.8284271247461903

estimated: 2.9999999999890860, sqrt(9): 3.0

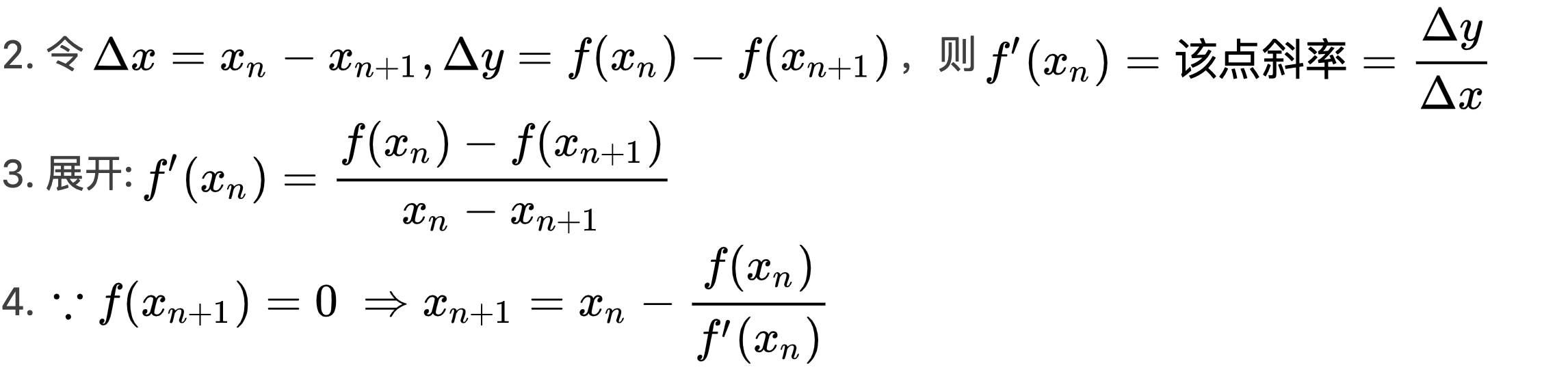
estimated: 3.1622776601579970, sqrt(10): 3.1622776601683795

## **牛顿法**

我就算不画图也能把这事说清楚。

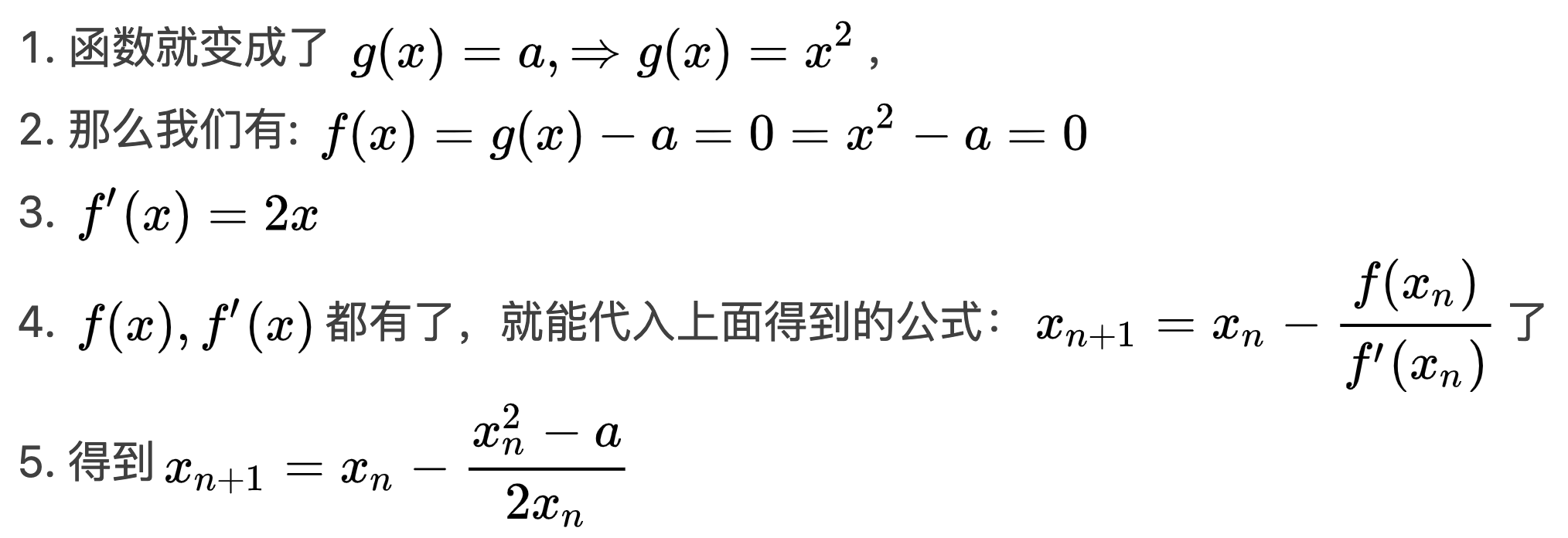
牛顿法用的是斜率的思想，对*f(x)=0*，选一个足够接近目标值(*x*)的点(*x0*)，计算其切线与X轴的交点(*x1*)，这个交点*x1*往往比*x0*更接近*x*，数次迭代后肯定越来越接近目标值*x*。

1. 问题转化成一个求函数上任一点(*xn*)的切线与X轴的交点(*xn+1*)的问题(我们假设n+1在n的左边，即向左来逼近*x0*)



1. 至此，我们用*xn*推出了一个离*x0*更近的点*xn+1*：
2. 如此往复则可以推出足够精度的解。

而求任意正整数*a*的平方根，



现在可以写代码了，不断去迭代，求下一个*xn+1*:

def newton\_sqrt(n):

x\_n = n / 2

epsilon = 1e-10 # quit flag

f\_x = lambda a : a\*\*2 - n # f(x)=x^2 - a

df\_x = lambda a : 2\*a # derivative of f(x)

x\_next = lambda a : a - f\_x(a) / df\_x(a)

while abs(x\_n \*\* 2 - n) > epsilon:

x\_n = x\_next(x\_n)

return x\_n

for i in range(1, 10):

print(f'sqrt({i})\t{newton\_sqrt(i)}')

output

sqrt(1) 1.000000000000001

sqrt(2) 1.4142135623746899

sqrt(3) 1.7320508075688772

sqrt(4) 2.0

sqrt(5) 2.23606797749979

sqrt(6) 2.4494897427831788

sqrt(7) 2.6457513110646933

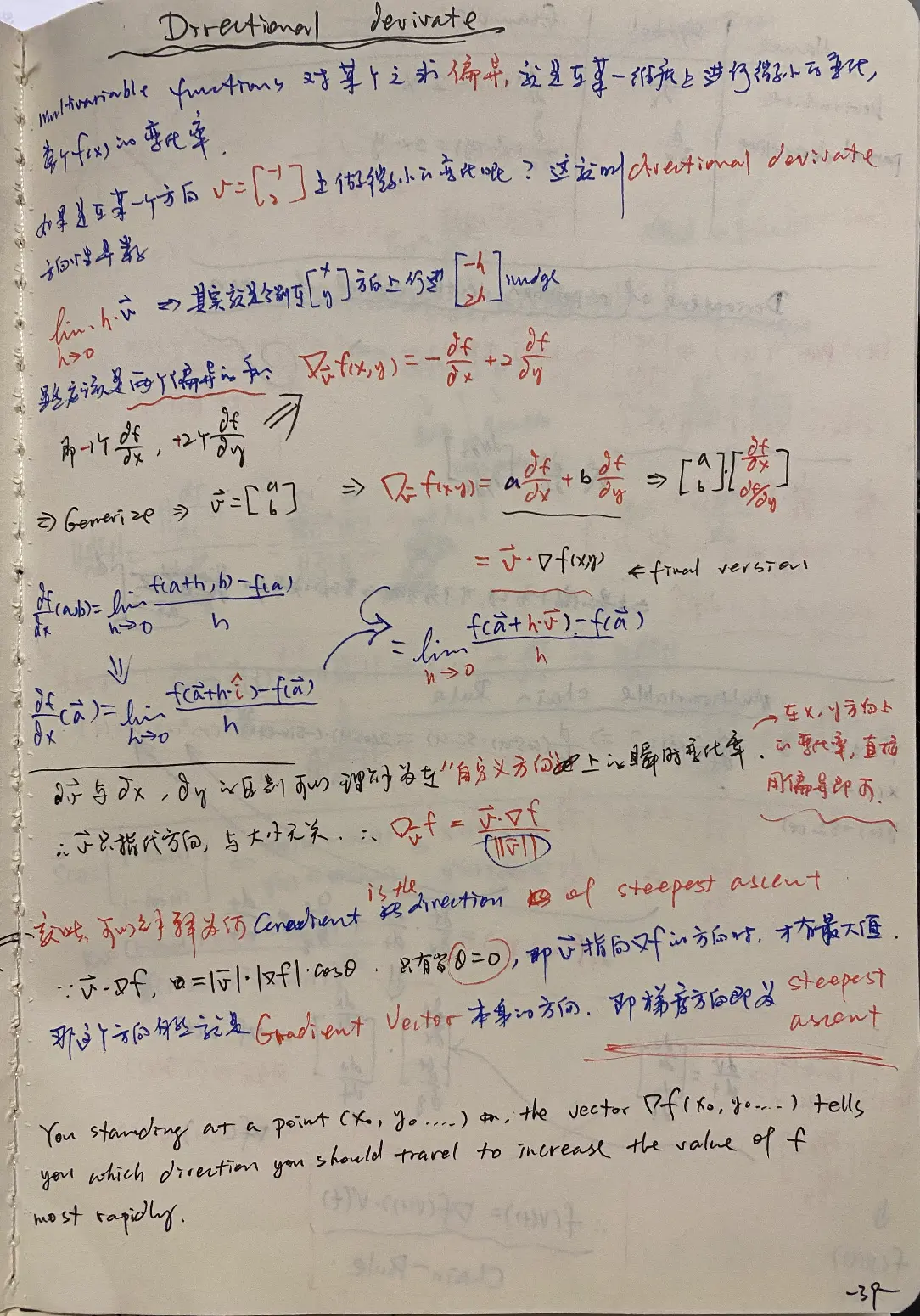
sqrt(8) 2.8284271247493797

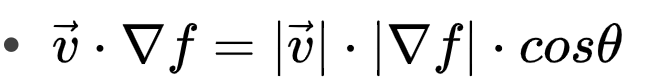
sqrt(9) 3.0

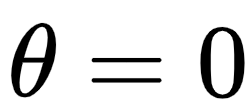
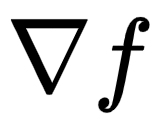
## **梯度下降法**

梯度下降法的数学原理是*f(x1,x2 ,...)*的gradient（）就是其最陡爬升方向（steepest ascent）。

可以拿这个当成结论，也可以去感性认识，而要去证明的话，网上有数不清的教程，在花书(《Deep Learning深度学习》)和可汗学院里，都是用的directional derivate来解释（证明）的，即”****自定义方向****上的瞬时变化率“，也是我认为在如果有多变量微积分的基础下，比较容易让人接受且简单直白的解释：

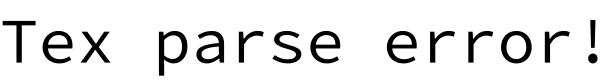




* 就是指的任意方向，如果是x, y等方向，那就是普通的偏导了。
* 显然上式当时拥有最大值，即指向的是的方向，那就是梯度的方向了
* 所以梯度方向就是爬升最陡峭的方向

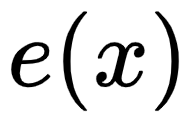
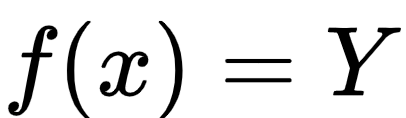
在一元方程里，”梯度“就是过某点的斜率（slope)，或者说函数的导数（derivative）。

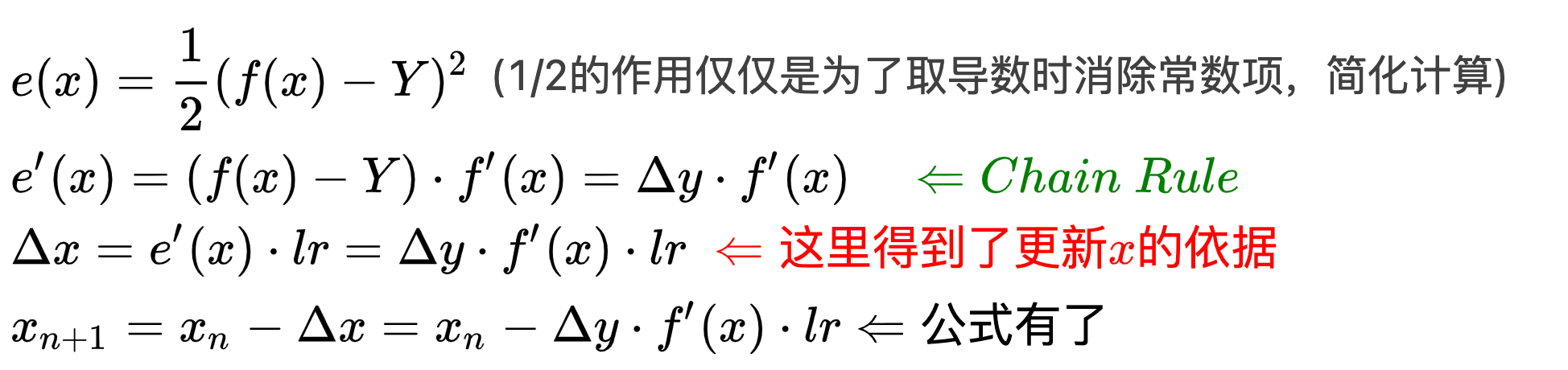
我们要到局部最小值，显然就应该向相向的方向走。并且由于越接近目标值（谷底），斜率越小，所以即使我们选择一个固定的步长（learning rate），也是会有一个越来越小的步进值去逼近极值，而无需刻意去调整步长。

以上是思路，只是要注意它上去的，而是精心设计的loss，或者说diff、error函数。

其实它跟前面的二分法很类似，就是不断指导里应该在哪个区间里去尝试下一个x值，再来结果与真值的差异（而牛顿法则是直接朝着直值去逼近，并不是在“尝试“）。

二分法里我随便设计了一个判断loss的函数（即中值的平方减真值），而梯度下降里不能那么随便了，它需要是一个连续的函数（即可微分），还要至少拥有局部极小值：

我们令表示不同的x取值下与目标值的差的平方（损失函数loss），既然是一个简单二次函数，就能求极值，且它的最小值意味着当x值为该值时估算原函数的****误差最小****，有：



这时可以写代码了

def gradient\_sqrt(n):

x = n / 2 # first try

lr = 0.01 # learning rate

epsilon = 1e-10 # quit flag

f\_x = lambda a : a\*\*2

df\_dx = lambda a : 2\*a

delta\_y = lambda a : f\_x(a) -n

e\_x = lambda a : delta\_y(a)\*\*2 \* 0.5 # funcon of loss

de\_dx = lambda a : delta\_y(a) \* df\_dx(a) # derivative of loss

delta\_x = lambda a : de\_dx(a) \* lr

count = 0

while abs(x\*\*2 - n) > epsilon:

count += 1

x = x - delta\_x(x)

return x, count

for i in range(1, 10):

print(f'sqrt({i}): {gradient\_sqrt(i)}次')

output

sqrt(1): (0.9999999999519603, 593)次

sqrt(2): (1.4142135623377403, 285)次

sqrt(3): (1.7320508075423036, 181)次

sqrt(4): (2.0, 0)次

sqrt(5): (2.236067977522142, 103)次

sqrt(6): (2.449489742798969, 87)次

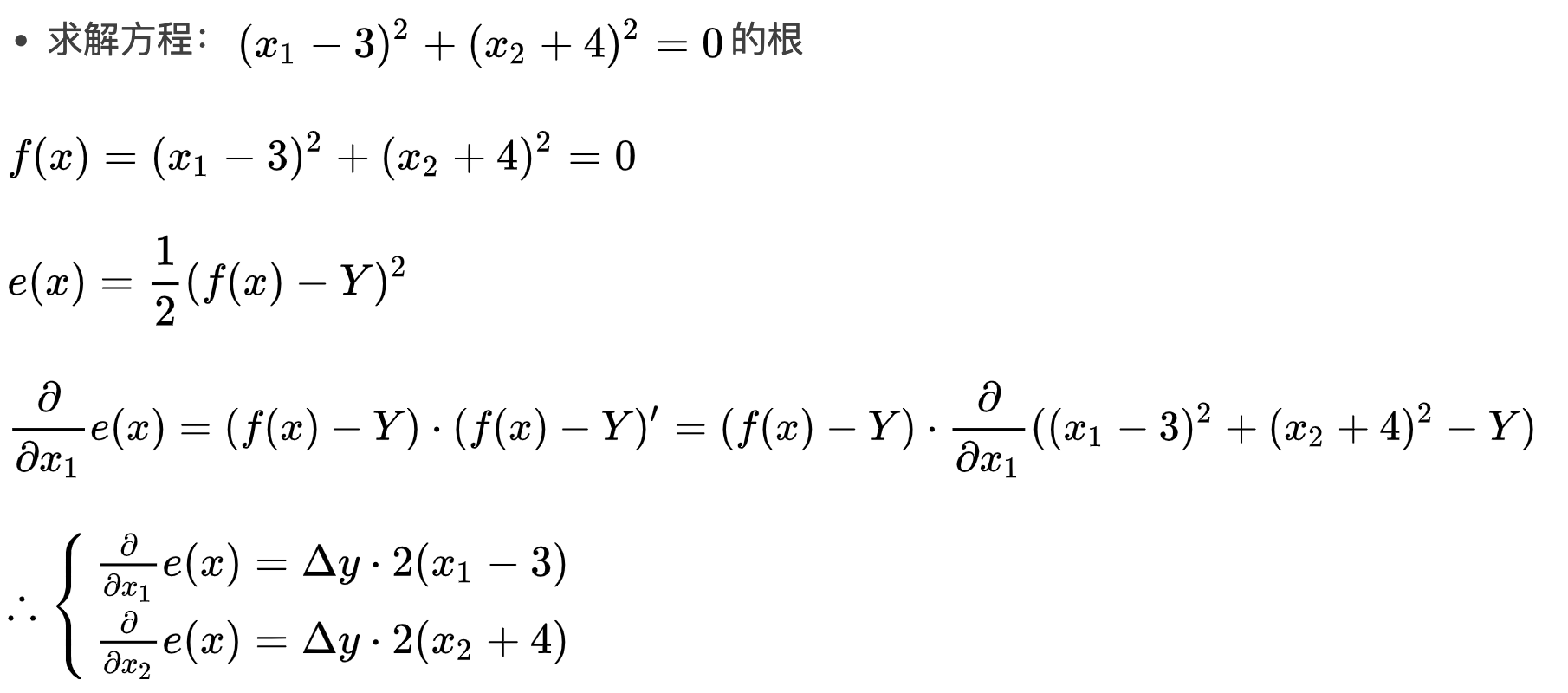
sqrt(7): (2.645751311082885, 73)次

sqrt(8): (2.828427124761154, 63)次

sqrt(9): (3.00000000001166, 55)次

****Bonus****

## **梯度下降解二次方程**



def gradient\_f(n):

x1, x2 = 1, 1 # first try

lr = 0.01 # learning rate

epsilon = 1e-4 # quit flag

f\_x = lambda x1, x2 : (x1-3)\*\*2 + (x2+4)\*\*2

dfx1 = lambda x : 2 \* (x - 3)

dfx2 = lambda x : 2 \* (x + 4)

delta\_y = lambda x1, x2 : f\_x(x1, x2) - n

e\_x = lambda x1, x2 : delta\_y(x1, x2)\*\*2 \* 0.5 # cost function

dedx1 = lambda x1, x2 : delta\_y(x1, x2) \* dfx1(x1) # partial derivative of loss \

dedx2 = lambda x1, x2 : delta\_y(x1, x2) \* dfx2(x2) # with Chain Rule

delt\_x1 = lambda x1, x2 : dedx1(x1, x2) \* lr

delt\_x2 = lambda x1, x2 : dedx2(x1, x2) \* lr

count = 0

while abs(f\_x(x1, x2) - n) > epsilon:

count += 1

x1 -= delt\_x1(x1, x2)

x2 -= delt\_x2(x1, x2)

return x1, x2, count

a, b, c = gradient\_f(0)print(f'''

a \t= {a}

b \t= {b}

f(a, b) = {(a-3)\*\*2 + (b+4)\*\*2}

count \t= {c}''')

output

a = 2.9967765158140387

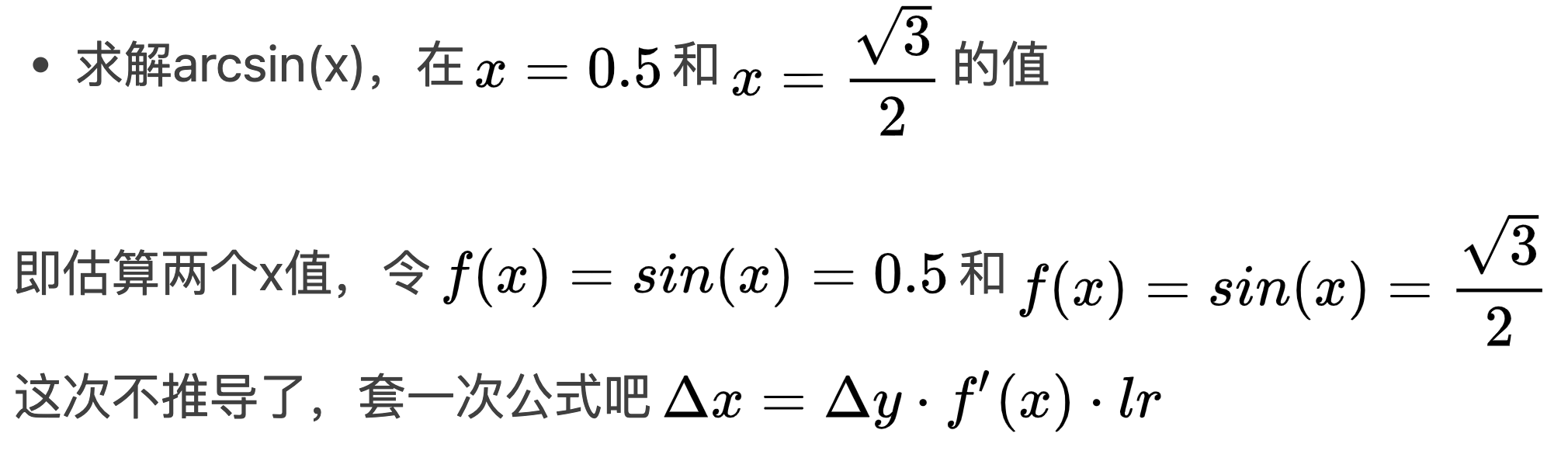
b = -3.9905337923806563

f(a, b) = 9.999993698966316e-05

count = 249990

之所以做两个练习， 就是因为第一个是故意把过程写得非常详细，如果直接套公式的话，而不是把每一步推导都写成代码，可以简单很多（其实就是最后一步的结果）:





import math

def arcsin(n):

x = 1 # first try

lr = 0.1 # learning rate

epsilon = 1e-8 # quit flag

f\_x = lambda x : math.sin(x)

delta\_y = lambda x : f\_x(x) - n

delta\_x = lambda x : delta\_y(x) \* math.cos(x) \* lr

while abs(f\_x(x) - n) > epsilon:

x -= delta\_x(x)

return math.degrees(x)

print(f'''sin({arcsin(0.5)}) ≈ 0.5

sin({arcsin(math.sqrt(3)/2)}) ≈ sqrt(3)/2

''')

output

sin(30.000000638736502) ≈ 0.5

sin(59.999998857570986) ≈ sqrt(3)/2