AI VIET NAM

GRADIENT PROGRAMMING

QUANG VAN

OUTLINE

1. HÀM NHIỀU BIẾN
2. TYPES OF DIFFERENTIATION
3. GRADIENT PROGRAMMING
4. EXERCISE

BẠN ĐÃ TỪNG THẮC MẮC VÌ SAO NETWORK CÓ THỂ TRAINING CHƯA?

HAY VÌ SAO CÁC FRAMEWORK CÓ THỂ ĐẠO HÀM MỘT CÁCH TỰ ĐỘNG NHƯ VẬY ĐƯỢC?

NGÀY HÔM NAY BẠN SẼ TỰ TAY XÂY DỰNG ĐƯỢC MỘT TOOL MINI ĐẠO HÀM TỰ ĐỘNG!

ARE YOU READY?

HÀM NHIỀU BIẾN

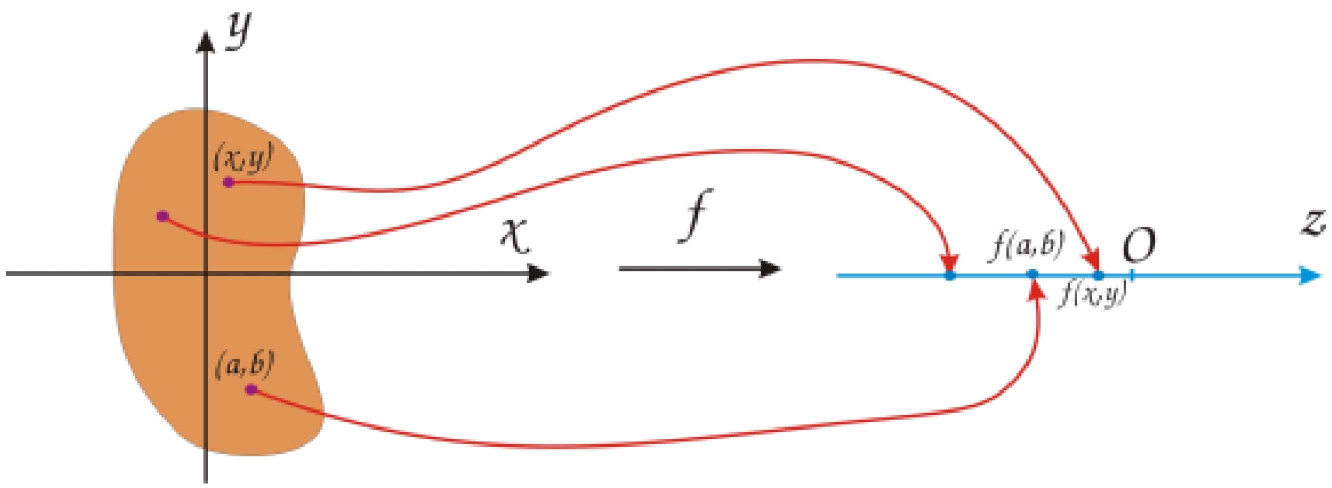
HÀM 2 BIẾN

Định nghĩa: Hàm hai biến là **một quy luật** ứng với một cặp số thực được sắp xếp thứ tự $(x, y) \in D$, ta luôn xác định được **duy nhất** một số thực $z = f(x, y)$

$$\boxed{\begin{aligned} &f : D \subset \mathbb{R}^2 \to \mathbb{R} \\ &(x, y) \longmapsto z = f(x, y) \end{aligned}}$$

Tập hợp D được gọi là **miền xác định** của hàm số f và được kí hiệu D(f).

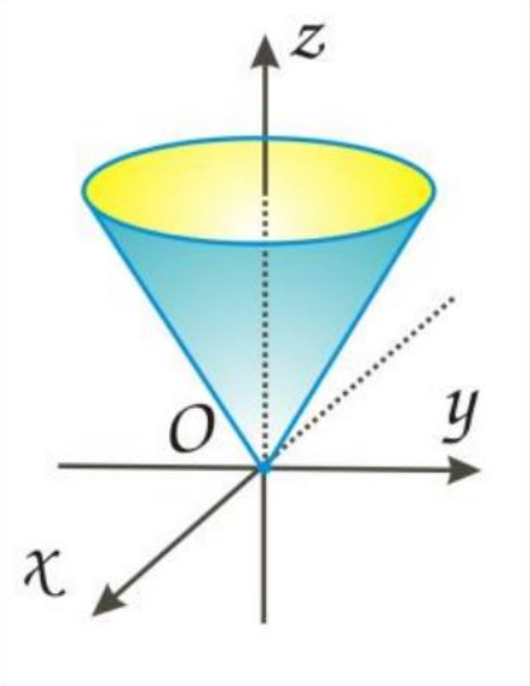
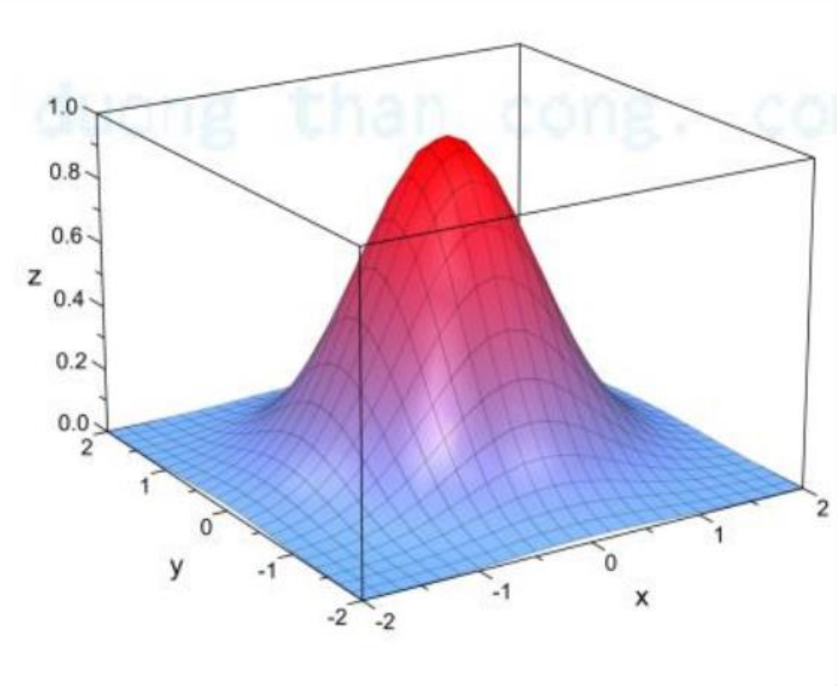
Tập hợp $E = \{z, \exists (x, y) \in D: z = f(x, y)\}$ được gọi là **tập giá trị** của hàm số f



HÀM 2 BIẾN

Định nghĩa: *Đồ thị của hàm hai biến* z = f (x, y) là tập hợp tất cả những điểm $(x, y, z) \in R^3$ sao cho z = f (x, y) và $(x, y) \in D$

Đồ thị của hàm một biến y = f (x) là *một đường cong*, còn đồ thị của hàm hai biến z = f (x, y) là *một mặt cong*.

	
Đồ thị hàm số $z = \sqrt{x^2 + y^2}$	Đồ thị hàm số $z = e^{-x^2-y^2}$

HÀM n BIẾN

$$\boldsymbol{f} : D \subset \mathbb{R}^n \to \mathbb{R}$$

$$(x_1, x_2, \dots, x_n) \longmapsto z = f(x_1, x_2, \dots, x_n)$$

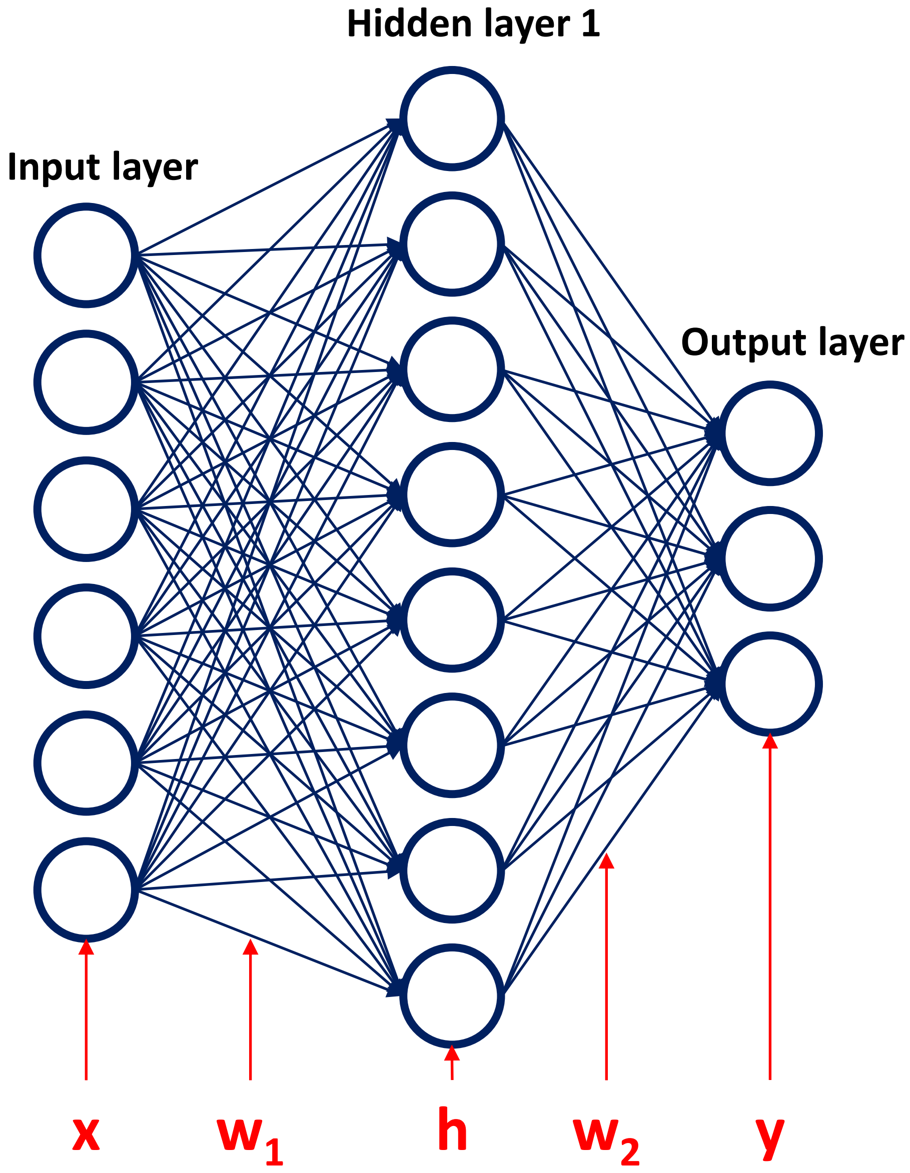
HÀM NHIỀU BIẾN

Giả sử ta có nhiều luật $\boldsymbol{f}_1,\ \boldsymbol{f}_2,\ \dots, \boldsymbol{f}_n$
là các hàm nhận n giá trị $\boldsymbol{x}_1,\ \boldsymbol{x}_2,\ \dots, \boldsymbol{x}_m$

$$
\begin{aligned}
z_1 &= f_1(x_1, x_2, \dots, x_m)\\
z_2 &= f_2(x_1, x_2, \dots, x_m)\\
&\vdots\\
z_n &= f_n(x_1, x_2, \dots, x_m)
\end{aligned}
$$

$$\boxed{\boldsymbol{f}: D \subset \mathbb{R}^m \to \mathbb{R}^n}$$

Similar



HÀM NHIỀU BIẾN

scalar to scalar

$$\boldsymbol{f} : \mathbb{R} \to \mathbb{R}$$
$$x \in R, y \in R$$

scalar to vector

$$\boldsymbol{f} : \mathbb{R} \to \mathbb{R}^{nx1}$$
$$x \in \mathbb{R}, y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} \in \mathbb{R}^{nx1}$$

vector to scalar

$$\boldsymbol{f} : \mathbb{R}^{nx1} \to \mathbb{R}$$
$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \in \mathbb{R}^{nx1}, y \in \mathbb{R}$$

vector to vector

$$\boldsymbol{f} : \mathbb{R}^{nx1} \to \mathbb{R}^{mx1}$$
$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \in \mathbb{R}^{nx1}, y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ x_n \end{bmatrix} \in \mathbb{R}^{mx1}$$

TYPES OF DIFFERENTIATION

TYPES OF DIFFERENTIATION

1. Scalar differentiation: $f : \mathbb{R} \to \mathbb{R}$

   $y \in \mathbb{R}$ w.r.t. $x \in \mathbb{R}$

2. Multivariate case: $f : \mathbb{R}^N \to \mathbb{R}$

   $y \in \mathbb{R}$ w.r.t. vector $\boldsymbol{x} \in \mathbb{R}^N$

3. Vector fields: $f : \mathbb{R}^N \to \mathbb{R}^M$

   vector $\boldsymbol{y} \in \mathbb{R}^M$ w.r.t. vector $\boldsymbol{x} \in \mathbb{R}^N$

4. General derivatives: $f : \mathbb{R}^{M \times N} \to \mathbb{R}^{P \times Q}$

   matrix $\boldsymbol{y} \in \mathbb{R}^{P \times Q}$ w.r.t. matrix $\boldsymbol{X} \in \mathbb{R}^{M \times N}$

TYPES OF DIFFERENTIATION

SCALAR DIFFERENTIATION

scalar to scalar

$$\boldsymbol{f} : \mathbb{R} \rightarrow \mathbb{R}$$
$$x \in R, y \in R$$

Công thức đạo hàm một bên

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x + \epsilon) - f(x)}{\epsilon}$$

Công thức đạo hàm trung tâm

$$f'(x) = \lim_{\epsilon \to 0} \frac{f\left(x + \frac{\epsilon}{2}\right) - f\left(x - \frac{\epsilon}{2}\right)}{\epsilon}$$

Theo lý thuyết đạo hàm, epsilon càng nhỏ thì giá trị đạo hàm tại một điểm càng chính xác!

TYPES OF DIFFERENTIATION

MULTIVARIATE CASE

vector to scalar

$$\boldsymbol{f} : \mathbb{R}^{nx1} \rightarrow \mathbb{R}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \in \mathbb{R}^{nx1}, y \in \mathbb{R}$$

$$y = f(\boldsymbol{x}), \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \in \mathbb{R}^N$$

- Partial derivative (change one coordinate at a time):

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \dots, x_{i-1}, x_i + h, x_{i+1}, \dots, x_N) - f(\boldsymbol{x})}{h}$$

- Jacobian vector (gradient) collects all partial derivatives:

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_N} \end{bmatrix} \in \mathbb{R}^{1 \times N}$$

Note: This is a row vector.

TYPES OF DIFFERENTIATION

VECTOR FIELDS

vector to vector

$$\boldsymbol{f} : \mathbb{R}^{nx1} \rightarrow \mathbb{R}^{mx1}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \in \mathbb{R}^{nx1}, y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ x_m \end{bmatrix} \in \mathbb{R}^{mx1}$$

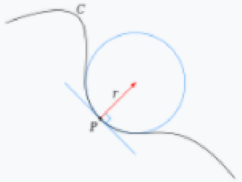
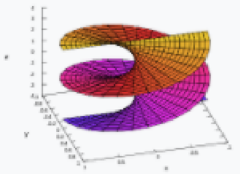
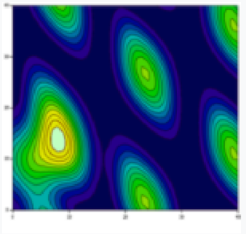
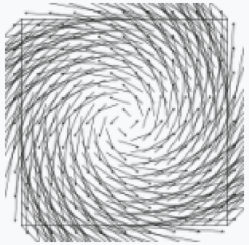
vector to scalar

$$J = \frac{dy}{dx} = \begin{bmatrix} \boxed{\frac{dy_1}{dx_1}, \frac{dy_1}{dx_2}, \dots, \frac{dy_1}{dx_n}} \\ \frac{dy_2}{dx_1}, \frac{dy_2}{dx_2}, \dots, \frac{dy_2}{dx_n} \\ \dots \\ \frac{dy_m}{dx_1}, \frac{dy_m}{dx_2}, \dots, \frac{dy_m}{dx_n} \end{bmatrix} \in \mathbb{R}^{mxn}$$

Jacobian matrix

TYPES OF DIFFERENTIATION

APPLICATIONS AND USES

		Type of functions	Applicable techniques
Curves		$f:\mathbb{R}\to\mathbb{R}^n$ for $n>1$	Lengths of curves, line integrals, and curvature.
Surfaces		$f:\mathbb{R}^2\to\mathbb{R}^n$ for $n>2$	Areas of surfaces, surface integrals, flux through surfaces, and curvature.
Scalar fields		$f:\mathbb{R}^n\to\mathbb{R}$	Maxima and minima, Lagrange multipliers, directional derivatives, level sets.
Vector fields		$f:\mathbb{R}^m\to\mathbb{R}^n$	Any of the operations of vector calculus including gradient, divergence, and curl.

https://en.wikipedia.org/wiki/Multivariable_calculus

GRADIENT PROGRAMMING

GRADIENT PROGRAMMING

1. **Manual Differentiation**
   dễ xảy ra lỗi, quá khó để triển khai đạo hàm

2. **Numberical Differentiation:**
   dễ xảy ra lỗi do giới hạn tính toán của máy tính

3. **Dual Numbers for AD:**
   kết quả chính xác như tính tay, và hoàn toàn tự động

① Manual Differentiation

Let's define a very simple function:

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

$$\Rightarrow f'(x) = 9x^2 - 10x$$

```
[1]  #In Python code:
     def func(x):
         return 3 * x ** 3 - 5 * x ** 2

     def func_der(x):
         return 9 * x ** 2 - 10 * x
```

```
[3]  func_der(1.5)
```

```
5.25
```

① Manual Differentiation

Ok, let try...

$$f(x) = sin(tan(x)^{cos(x)} cos(x)^{tan(x)})$$
$$f'(x)?$$

?

or

SoftRelu

$$log(1 + e^{wx+b})$$

After 2 layer, derivative become ...

$$\frac{e^{b_1+b_2+w_1x+w_2log(1+e_1^b+w_1x)}w_2x)}{(1+e^{b_1+w_1x})(1+e^{b_2+w2log(1+e^{b_1+w1x})})}$$

Next layer?

X

1 Manual Differentiation

dễ xảy ra lỗi, quá khó để triển khai đạo hàm

GRADIENT PROGRAMMING

1. Manual Differentiation
   dễ xảy ra lỗi, quá khó để triển khai đạo hàm

2. Numberical Differentiation:
   dễ xảy ra lỗi do giới hạn tính toán của máy tính

3. Dual Numbers for AD:
   kết quả chính xác như tính tay, và hoàn toàn tự động

② Numberical Differentiation

$$\boxed{f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}}$$

Với $\varepsilon \ll 0$

Tránh tại x không xác định:

$$\boxed{f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon/2) - f(x-\varepsilon/2)}{\varepsilon}}$$

② Numberical Differentiation

$$f(x) = sin(tan^{cos(x)} cos^{tan(x)})$$

$$f'(x)?$$

```
def gradient(f, x, epsilon=1.0e-13):
    return (f(x + epsilon/2) - f(x - epsilon/2))/epsilon
```

```
from math import sin, tan, cos

def ff(x):
    return sin(tan(x)**cos(x) * cos(x)**tan(x))
```

```
gradient(ff, 1.0)
```

```
-1.5987211554602254
```

! Chỉ cần định nghĩa **function**

② Numberical Differentiation

Ok, keep...

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

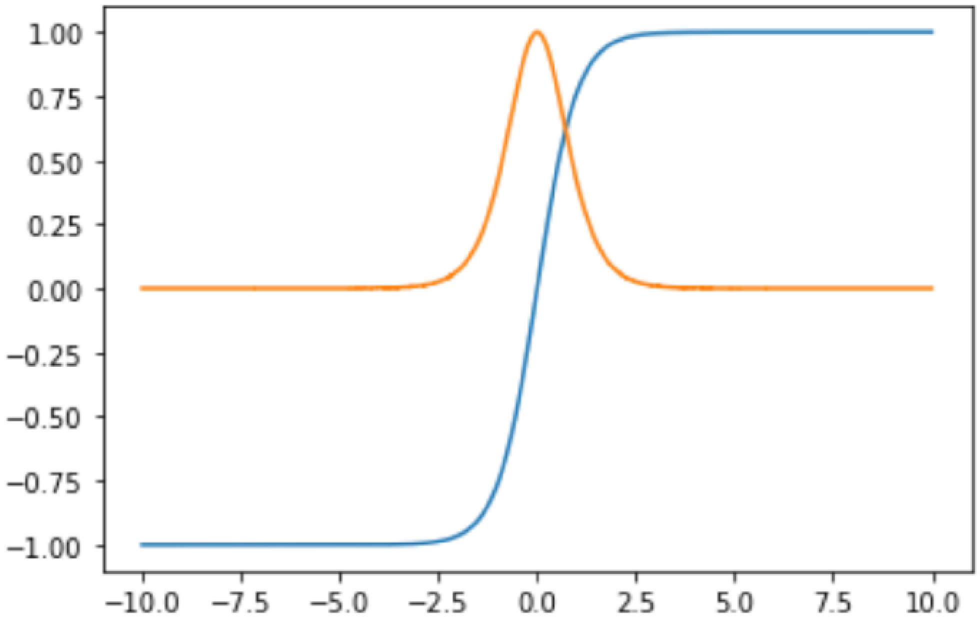
```
[15] import numpy as np

     def f1_3(x):
         return (np.exp(x)-np.exp(-x)) / (np.exp(x)+np.exp(-x))

     gradient(f1_3, 2.0)

     0.0716093850883226
```

② Numberical Differentiation

Plot it...

!

```
import matplotlib.pyplot as plt
x = np.arange(-10, 10, 0.01)

plt.plot(x, f1_3(x),
         x, gradient(f1_3, x))
plt.show()
```



$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```
[15] import numpy as np

     def f1_3(x):
         return (np.exp(x)-np.exp(-x)) / (np.exp(x)+np.exp(-x))

     gradient(f1_3, 2.0)

     0.0716093850883226
```

② Numberical Differentiation

So on...

```
[47] def f1_1(x):
         return x*(3*x+2)**(1/2)

     gradient(f1_1, 2)
```

$\boxed{4.440892098500626}$

Wrong!

$$y = x\sqrt{3x+2}$$

$$y' = \frac{9x+4}{2\sqrt{3x+2}}$$

$$y'(2) = \frac{9*2+4}{2\sqrt{3*2+2}} = 3.8890$$

Hey, we need new method pro men!!

2

Numberical Differentiation:

dễ xảy ra lỗi do giới hạn tính toán của máy tính

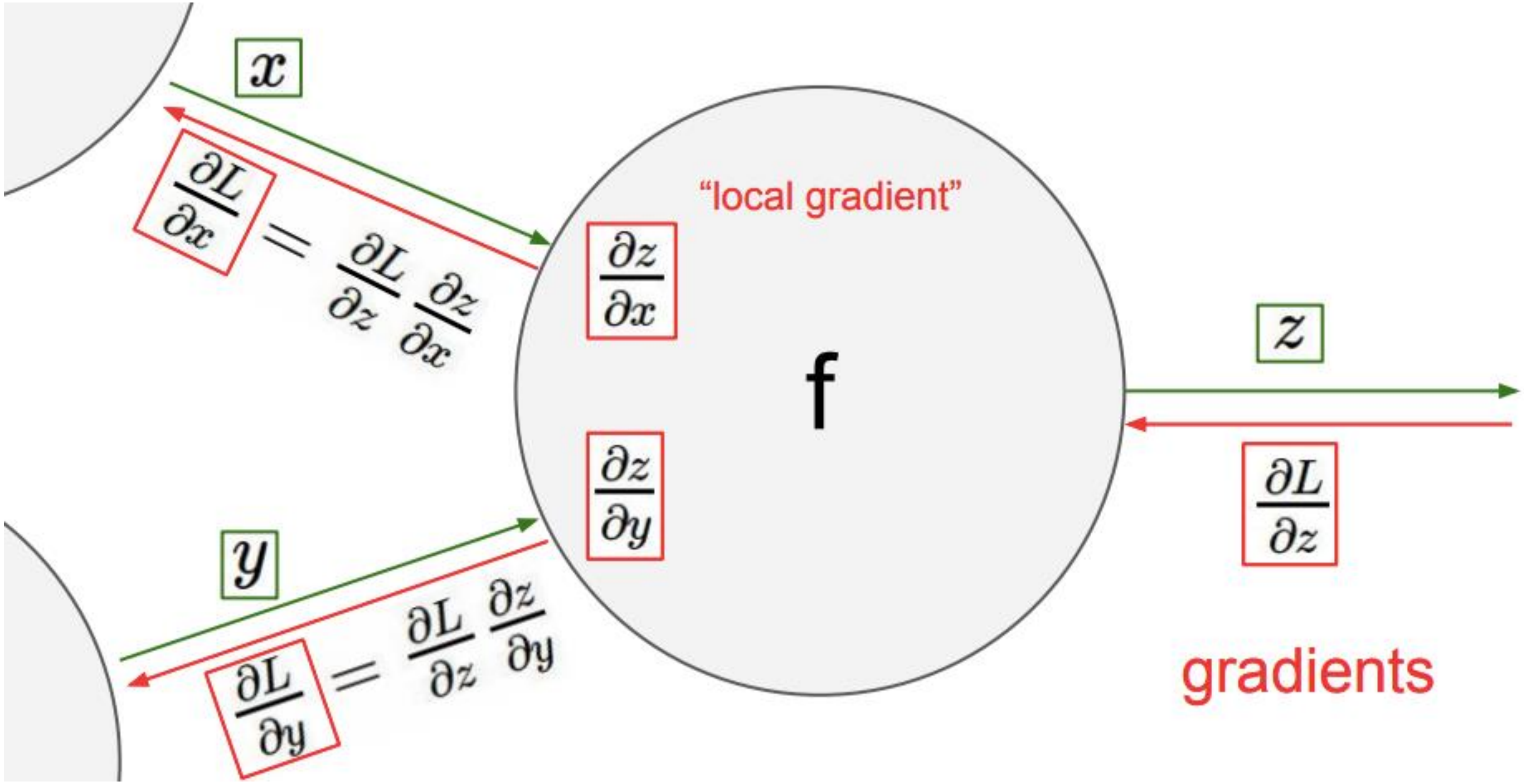
GRADIENT PROGRAMMING

1. Manual Differentiation
   dễ xảy ra lỗi, quá khó để triển khai đạo hàm

2. Numberical Differentiation:
   dễ xảy ra lỗi do giới hạn tính toán của máy tính

3. Dual Numbers for AD:
   kết quả chính xác như tính tay, và hoàn toàn tự động

GRADIENT PROGRAMMING



③ Dual Numbers for AD

Expand the **taylor** function **f(x) at x_0**

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \ldots$$

With $x \sim x_0$: $x - x_0 = \varepsilon$, we have:

$$f(x_0 + \varepsilon) = f(x_0) + \frac{f'(x_0)}{1!}\varepsilon + \frac{f''(x_0)}{2!}\varepsilon^2 + \ldots$$

$$f(x_0 + \varepsilon) = f(x_0) + f'(x_0)\varepsilon$$

③ Dual Numbers for AD

With $\boldsymbol{x} \sim \boldsymbol{x_0}$: $\boldsymbol{x} - \boldsymbol{x_0} = \boldsymbol{\varepsilon}$, we have:

$$\underline{f(x_0 + \varepsilon)} = \underline{f(x_0)} + \underline{f'(x_0)}\varepsilon$$

Dual Numbers

$$z = a + b\epsilon$$
$$\epsilon^2 = 0$$
$$a' = b$$

$$a = x$$
$$\Rightarrow z = x + 1\varepsilon$$

$$a = \sin(x)$$
$$\Rightarrow z = \sin(x) + \cos(x)\varepsilon$$

$$a = u^n$$
$$\Rightarrow z = u^n + u' n * u^{n-1}\varepsilon$$

GRADIENT PROGRAMMING

```python
class DualNumber:
    def __init__(self, value, prime):
        self.value = value
        self.prime = prime

    def __add__(self, other):

        if not isinstance(other, DualNumber):
            other = DualNumber(other, 0)

        return DualNumber(self.value + other.value,
                          self.prime + other.prime)

    def __radd__(self, other):
        return DualNumber(other, 0) + self


    def __sub__(self, other):

        if not isinstance(other, DualNumber):
            other = DualNumber(other, 0)

        return DualNumber(self.value - other.value,
                          self.prime - other.prime)

    def __rsub__(self, other):
        return DualNumber(other, 0) - self


    def __mul__(self, other):
        if not isinstance(other, DualNumber):
            other = DualNumber(other, 0)

        return DualNumber(self.value * other.value,
                          self.value * other.prime + self.prime * other.value)
```

```python
def __rmul__(self, other):
    return DualNumber(other, 0) * self


def __truediv__(self, other):
    if not isinstance(other, DualNumber):
        other = DualNumber(other, 0)

    return DualNumber(self.value/other.value,
                      (self.prime*other.value -
                       self.value*other.prime)/(other.value**2))

def __rtruediv__(self, other):
    return DualNumber(other, 0) / self

def __pow__(self, other):

    if not isinstance(other, DualNumber):
        other = DualNumber(other, 0)

    return DualNumber(self.value**other.value,
                      np.exp(other.value*np.log(self.value))
                      *(other.prime*np.log(self.value)
                        + other.value * self.prime * 1.0/self.value
                        )
                      )

def __repr__(self):
    return repr(self.value) + ' + ' + repr(self.prime) + '*epsilon'

def __neg__(self):
    return DualNumber(-self.value, -self.prime)

def __pos__(self):
    return DualNumber(self.value, self.prime)
```

GRADIENT PROGRAMMING

```python
def func(f):

    def fdeclare(dark):
        if not isinstance(dark, DualNumber):
            def wrapper(x):
                return f(dark(x))
            return wrapper

        return f(dark)

    return fdeclare

```

```python
def f(x):
    return x*(3*x+2)**(1/2)

auto_diff(f, [2.0])
```

```
array([3.8890873])
```

Ảo thật đấy!

```python
@func
def exp(x):
    return DualNumber(np.exp(x.value),
                      x.prime * np.exp(x.value))
@func
def sin(x):
    return DualNumber(np.sin(x.value),
                      x.prime * np.cos(x.value))
@func
def cos(x):
    return DualNumber(np.cos(x.value),
                      -x.prime * np.sin(x.value))
@func
def tan(x):
    return DualNumber(np.tan(x.value),
                      x.prime/np.cos(x.value)**2)
@func
def log(x):
    return DualNumber(np.log(x.value),
                      x.prime*(1.0/x.value))
@func
def pow(x, n):
    return DualNumber(x.value**n,
                      x.prime*n*(x.value**(n-1)))
@func
def fabs(x):
    return DualNumber(np.abs(x.value),
                      x.value/(x.value**2)**0.5)
@func
def fsum(x, axis=None, dtype=None, keepdims=np._NoValue):
    return DualNumber(np.sum(x.value, axis=axis, dtype=dtype, keepdims=keepdims),
                      x.prime)

def auto_diff(f, x):

    if isinstance(x, int):
        x = [x]

    x = np.array(x, dtype=np.float64)

    return f(DualNumber(x, np.ones(x.shape))).prime
```

3 Dual Numbers for AD:

kết quả chính xác như tính tay, và hoàn toàn tự động

OK, THỰC RA NÃY GIỜ TA MỚI TRIỂN KHAI SCALAR DIFFERENTIATION ^^

ĐỂ DỄ TIẾP CẬN, TA SẼ TRIỂN KHAI VÀ SỬ DỤNG NUMBERICAL DIFFERENTIATION

scalar to scalar

$$\boldsymbol{f} : \mathbb{R} \rightarrow \mathbb{R}$$
$$x \in R, y \in R$$

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon/2) - f(x - \varepsilon/2)}{\varepsilon}$$

```python
def gradient(f, x, epsilon=1.0e-13):
    return (f(x + epsilon/2)
            - f(x - epsilon/2))/epsilon
```

GRADIENT PROGRAMMING

scalar to scalar

$$f : \mathbb{R} \to \mathbb{R}$$
$$x \in R, y \in R$$

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon/2) - f(x-\varepsilon/2)}{\varepsilon}$$

```python
def gradient(f, x, epsilon=1.0e-13):
    return (f(x + epsilon/2)
        - f(x - epsilon/2))/epsilon
```

vector to scalar

$$y = f(\boldsymbol{x}), \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \in \mathbb{R}^N$$

- Partial derivative (change one coordinate at a time):

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_N) - f(\boldsymbol{x})}{h}$$

- Jacobian vector (gradient) collects all partial derivatives:

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_N} \end{bmatrix} \in \mathbb{R}^{1 \times N}$$

Note: This is a row vector.

vector to vector

vector to scalar

$$J = \frac{dy}{dx} = \begin{bmatrix} \boxed{\frac{dy_1}{dx_1}, \frac{dy_1}{dx_2}, \ldots, \frac{dy_1}{dx_n}} \\ \frac{dy_2}{dx_1}, \frac{dy_2}{dx_2}, \ldots, \frac{dy_2}{dx_n} \\ \cdots \\ \frac{dy_m}{dx_1}, \frac{dy_m}{dx_2}, \ldots, \frac{dy_m}{dx_n} \end{bmatrix} \in \mathbb{R}^{mxn}$$

```python
# From scratch
def d_vec_to_vec(f, w, epsilon=1e-7):
    result = []
    w = np.array(w, dtype=np.float64).reshape(-1, 1)
    for i in range(w.shape[0]):
        w_t = w.copy()
        w_p = w.copy()
        w_t[i] += epsilon/2
        w_p[i] -= epsilon/2
        result.append((f(w_t) - f(w_p))/epsilon)
    return np.concatenate(result, -1)
```

GRADIENT PROGRAMMING

scalar to scalar   scalar to vector   vector to scalar   vector to vector

```
# From scratch
def d_vec_to_vec(f, w, epsilon=1e-7):
    result = []
    w = np.array(w, dtype=np.float64).reshape(-1, 1)
    for i in range(w.shape[0]):
        w_t = w.copy()
        w_p = w.copy()
        w_t[i] += epsilon/2
        w_p[i] -= epsilon/2
        result.append((f(w_t) - f(w_p))/epsilon)
    return np.concatenate(result, -1)
```

EXERCISE

EXERCISE

TÌM NGHIỆM

$$x^4 - 8x^3 + 21x^2 - 24x + 9 = 0$$

$$(x+4)(x+6)(x-2)(x-12) = 25x^2$$

Optimization

$$z = 2x^4 + y^4 - 4x^2 + 2y^2$$

$$z = 2x^2 + 3y^2 - e^{-(x^2+y^2)}$$

LOSS FUNCTION

TÌM NGHIỆM

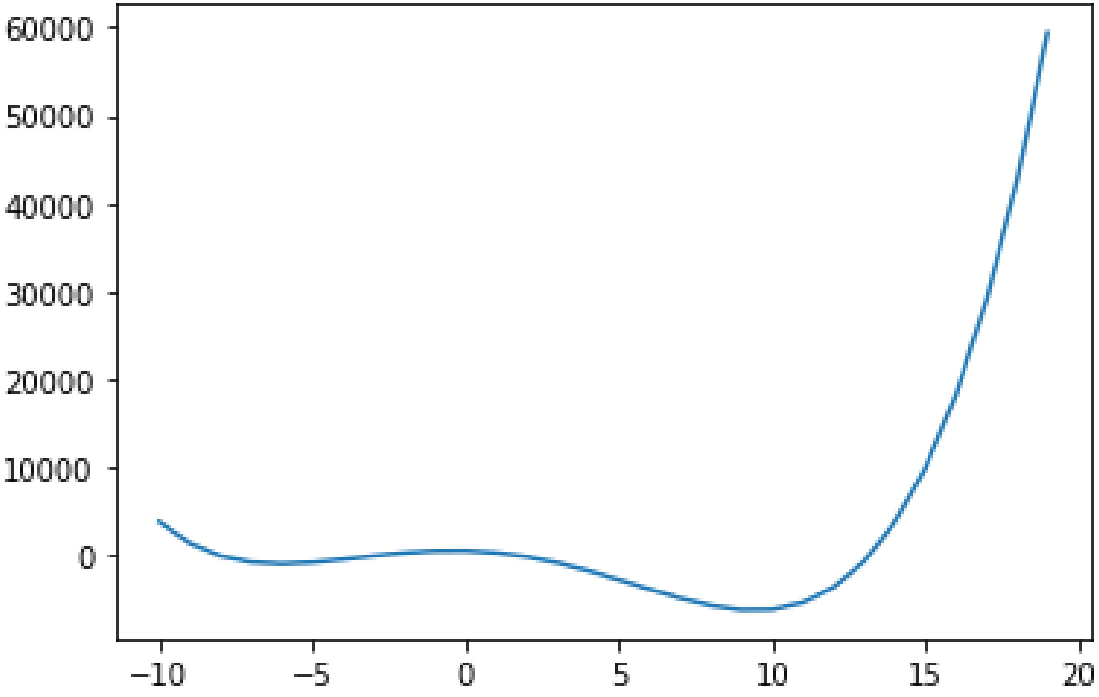
$$(x+4)(x+6)(x-2)(x-12) = 25x^2$$

$$\Rightarrow \left[\begin{array}{l} x = -8 \\ x = -3 \\ x = \dfrac{15 \pm \sqrt{129}}{2} \end{array}\right.$$

Chọn điểm rơi

```
epochs = 1000
x = 9
for _ in range(epochs):
    x = x - f(x)/d_f(f, x)
print(x)
```

```
-8.0
```



TÌM NGHIỆM

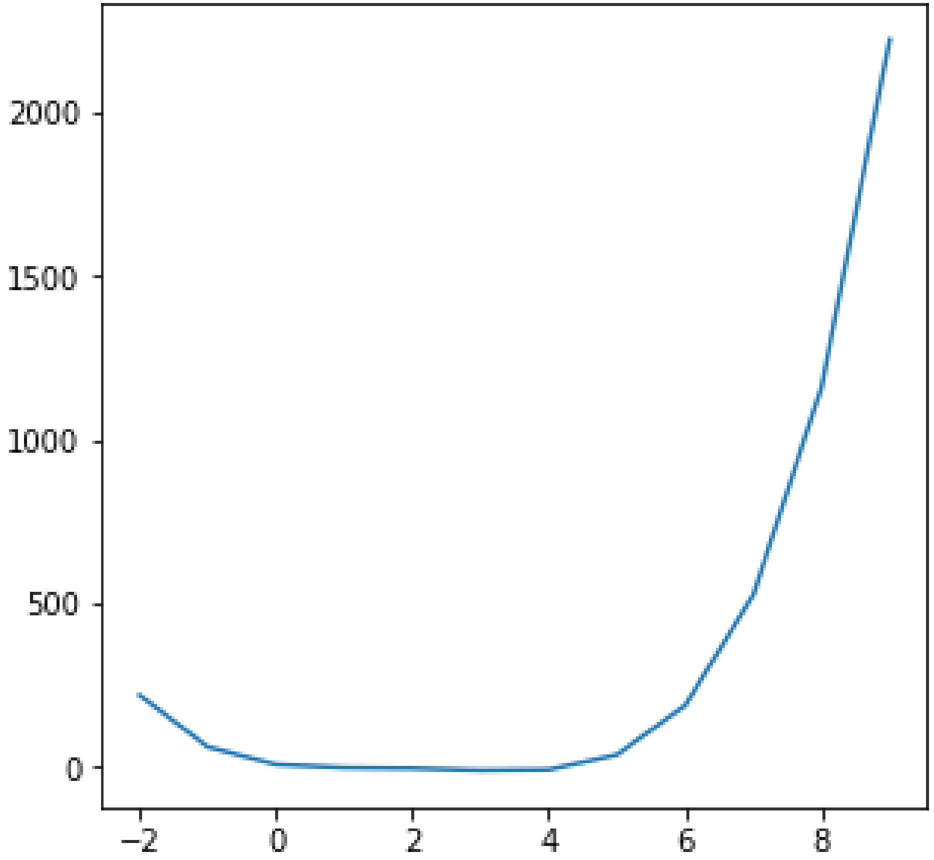
$$x^4 - 8x^3 + 21x^2 - 24x + 9 = 0$$

$$\Rightarrow x = \frac{5 \pm \sqrt{13}}{2}$$

Chọn điểm rơi

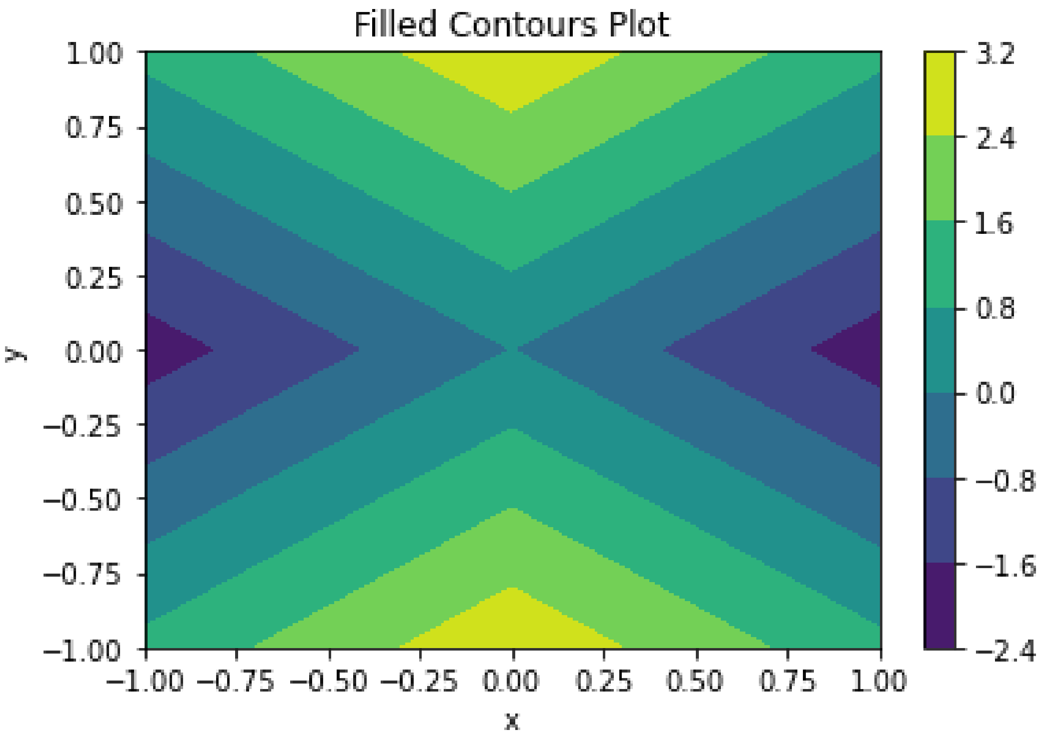
```
epochs = 1000
x = 2
for _ in range(epochs):
    x = x - f(x)/d_f(f, x)
print(x)
```

```
0.697224362268052
```

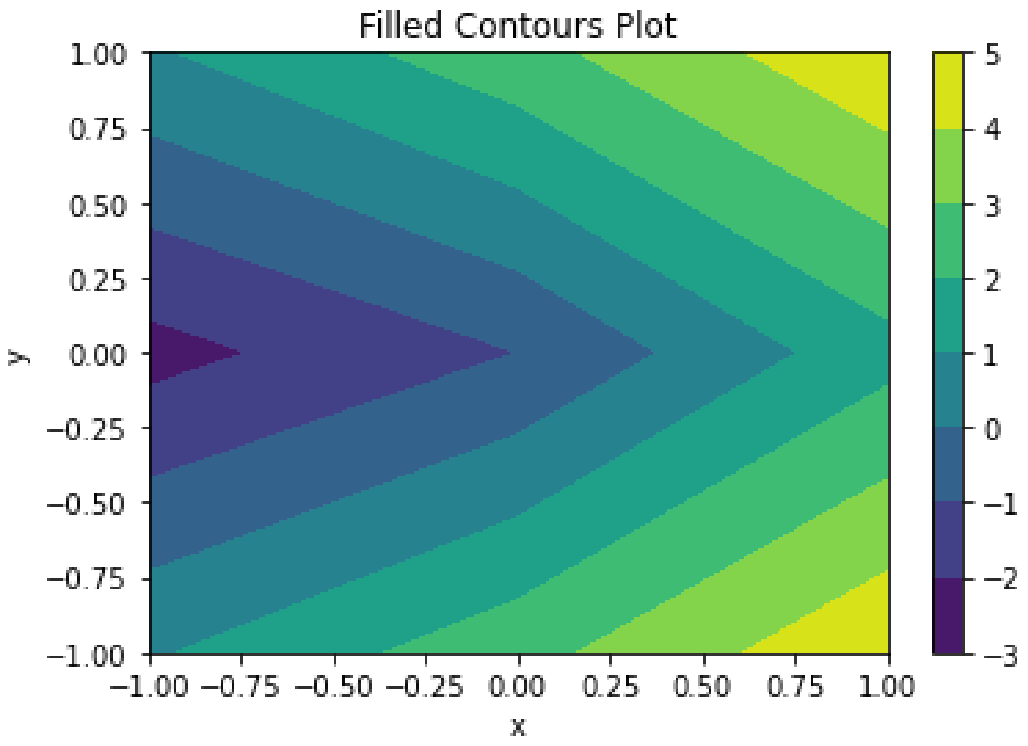


Optimization

$$z = 2x^4 + y^4 - 4x^2 + 2y^2$$



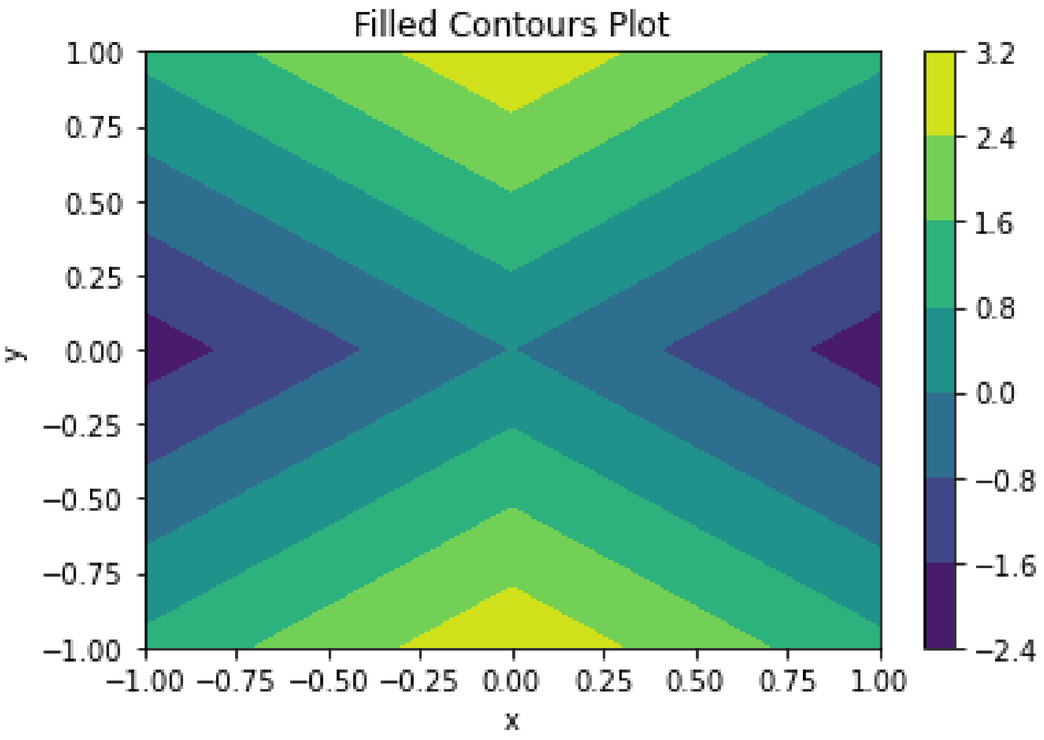
$$z = 2x^2 + 3y^2 - e^{-(x^2+y^2)}$$



Optimization

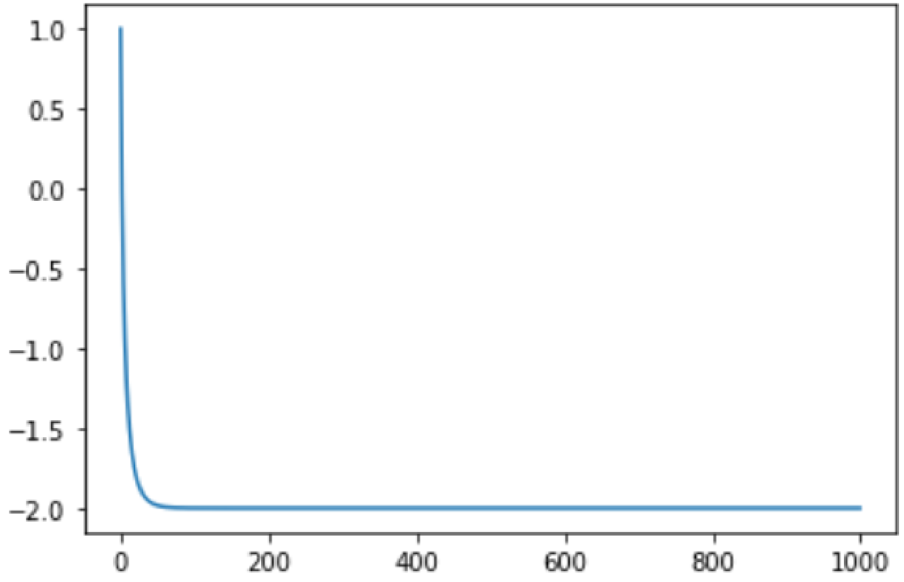
Chọn điểm rơi

$$z = 2x^4 + y^4 - 4x^2 + 2y^2$$




```python
x = 1.0
y = 1.0
lr = 0.01
epoch = 1000
f_val = []
for _ in range(epoch):
    z = f1([x,y])
    dx, dy = d_vec_to_vec(f1, [x,y])
    x -= dx*lr
    y -= dy*lr
    f_val.append(z)
plt.plot(f_val)
print(f"min = {f_val[-1]} with x,y={x,y}")
```

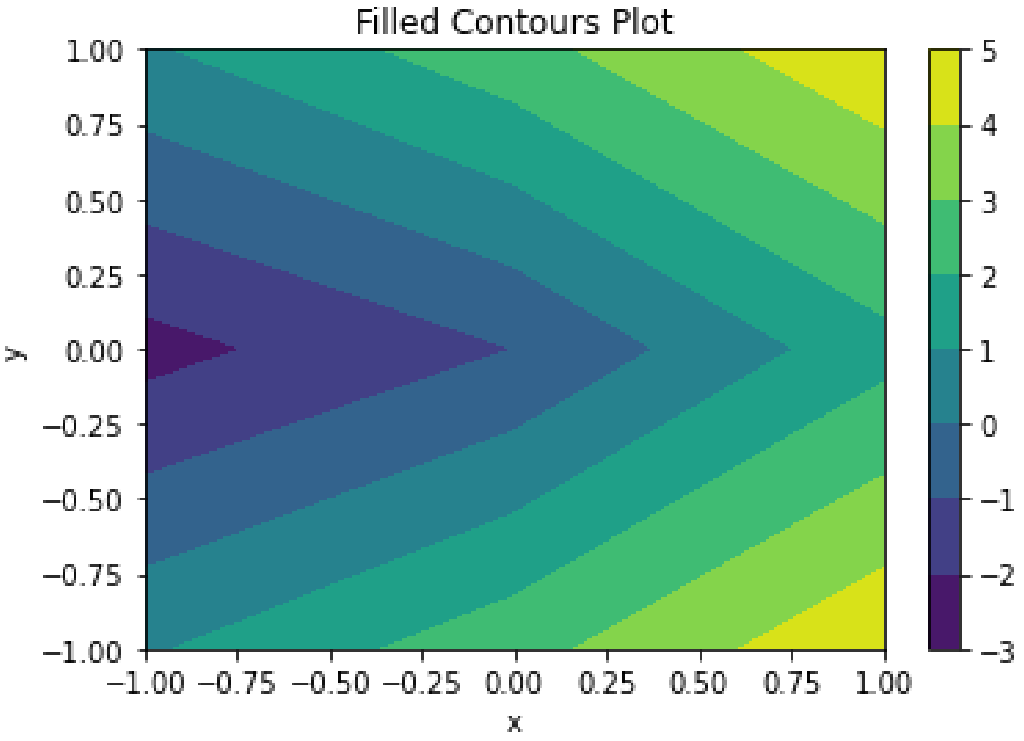
```
min = -2.0 with x,y=(0.9999999999111822, 4.551026222543442e-12)
```



Optimization

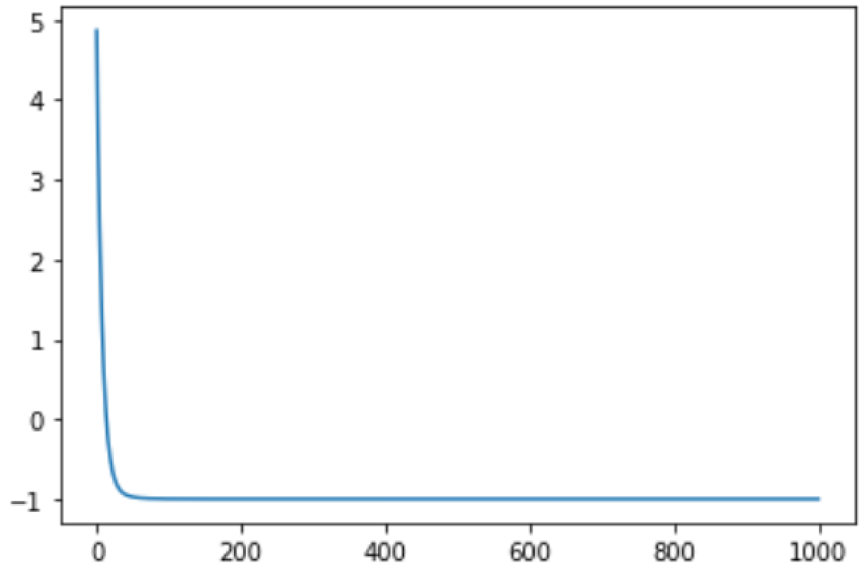
Chọn điểm rơi

$$z = 2x^2 + 3y^2 - e^{-(x^2+y^2)}$$




```python
x = 1.0
y = 1.0
lr = 0.01
epoch = 1000
f_val = []
for _ in range(epoch):
    z = f2([x,y])
    dx, dy = d_vec_to_vec(f2, [x,y])
    x -= dx*lr
    y -= dy*lr
    f_val.append(z)
plt.plot(f_val)
print(f"min = {f_val[-1]} with x,y={x,y}")
```

```
min = -1.0 with x,y=(5.763158839044991e-10, 4.551026222543442e-12)
```



EXERCISE

Optimization loss function (eg. Linear regression)

Advertising.csv

	TV	Radio	Newspaper	Sales
1	230.1	37.8	69.2	22.1
2	44.5	39.3	45.1	10.4
3	17.2	45.9	69.3	12
4	151.5	41.3	58.5	16.5
5	180.8	10.8	58.4	17.9
6	8.7	48.9	75	7.2
7	57.5	32.8	23.5	11.8
8	120.2	19.6	11.6	13.2
9	8.6	2.1	1	4.8
10	199.8	2.6	21.2	15.6
11	66.1	5.8	24.2	12.6
12	214.7	24	4	17.4
13	23.8	35.1	65.9	9.2
14	97.5	7.6	7.2	13.7
15	204.1	32.9	46	19
16	195.4	47.7	52.9	22.4
17	67.8	36.6	114	12.5
18	281.4	39.6	55.8	24.4

EXERCISE

Optimization loss function (eg. Linear regression)

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```python
# From scratch
def d_vec_to_vec(f, w, epsilon=1e-7):
    result = []
    w = np.array(w, dtype=np.float64).reshape(-1, 1)
    for i in range(w.shape[0]):
        w_t = w.copy()
        w_p = w.copy()
        w_t[i] += epsilon/2
        w_p[i] -= epsilon/2
        result.append((f(w_t) - f(w_p))/epsilon)
    return np.concatenate(result, -1)
```

[3]:
```python
data = np.genfromtxt("advertising.csv", delimiter=',', skip_header=True)
```

[4]:
```python
x = data[:, :-1]
y = data[:, -1:]
```

[5]:
```python
n_sample, n_features = x.shape # Lấy số sample, và số features
```

[6]:
```python
x = (x - x.mean())/x.std()
```

[7]:
```python
x = np.concatenate((x, np.ones((n_sample, 1))), axis = 1) # Thêm cột 1 vào x
```

[8]:
```python
x.shape
```

[8]:
```
(200, 4)
```

[9]:
```python
def linear(const):
    def handle(x):
        return const@x
    return handle
```

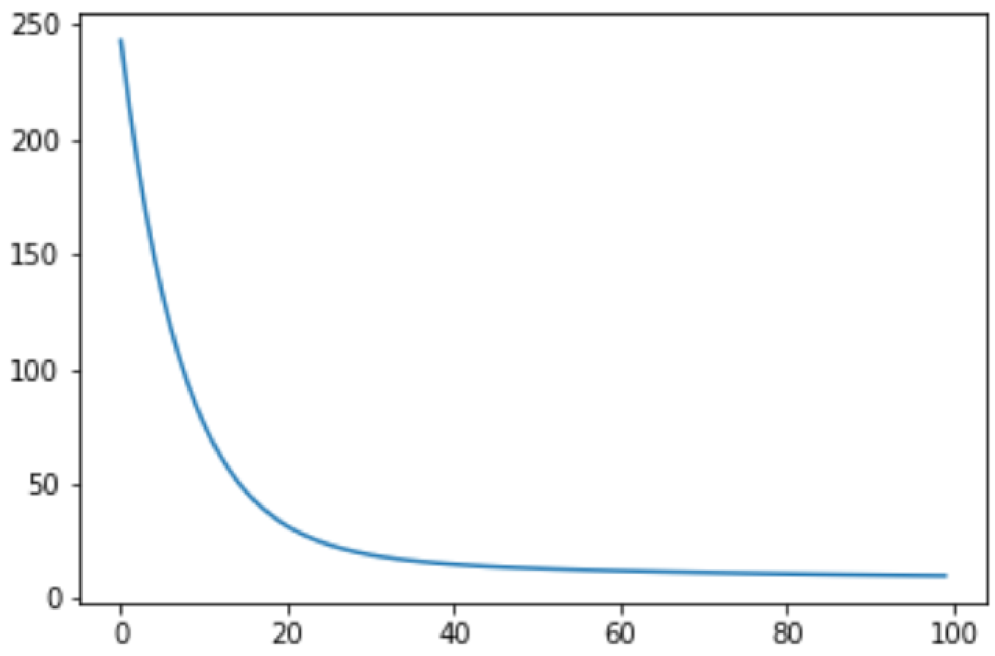
[10]:
```python
def mean_square_error(const):
    def handle(x):
        return np.mean((const-x)**2, keepdims=True)
    return handle
```

[11]:
```python
# init parameters (weight, learning_rate, epoch, batch_size)
w = np.random.rand(n_features + 1, 1)
learning_rate = 0.01
batch_size = n_sample
n_epochs = 100


#for debug
losses = []
```

[12]:
```python
# Tìm w, b
for epoch in range(n_epochs):

    # May be shufle

    for i in range(0, n_sample, batch_size):

        # pick sample
        x_i = x[i:i+batch_size, :]
        y_i = y[i:i+batch_size]

        # predict
        y_pred = linear(x_i)(w)

        # loss (debug)
        l = mean_square_error(y_i)(y_pred) # np.abs(o - y_i)
        losses.extend(l)

        # compute gradient (d_weight)
        dw = (d_vec_to_vec(mean_square_error(y_i), y_pred) @ d_vec_to_vec(linear(x_i), w)).T

        # Update weight
        w = w - learning_rate*dw
```

Optimization loss function (eg. Linear regression)

[13]:
```
plt.plot(losses);
```



[29]:
```
# Evaluate
np.abs(x.dot(w) - y).mean()
```

[29]: 2.517039010286411