Activity Project 2(Noe Lomidze)

Noe Lomidze

26.10.2024

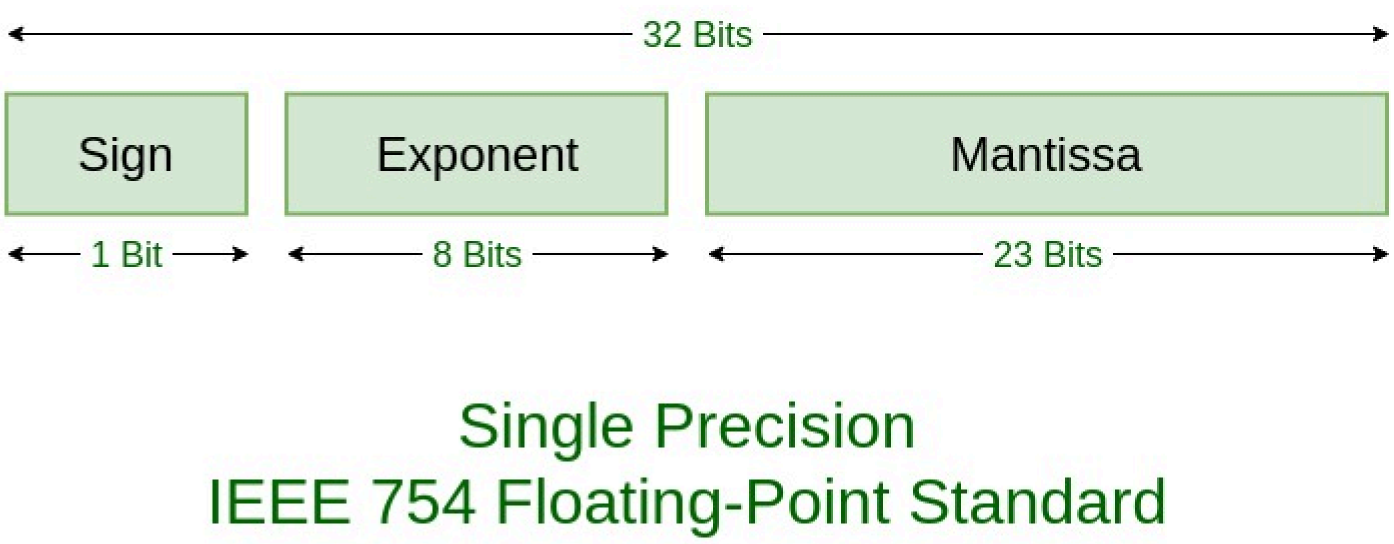
Most scientific and engineering computations on a computer are performed using floating point arithmetic. Computers may have different bases, though base 2 is most common. The other commonly used bases are 10 and 16. Most hand calculators use base 10, while IBM mainframes use base 16.

A t-digit floating point number in base β has the form

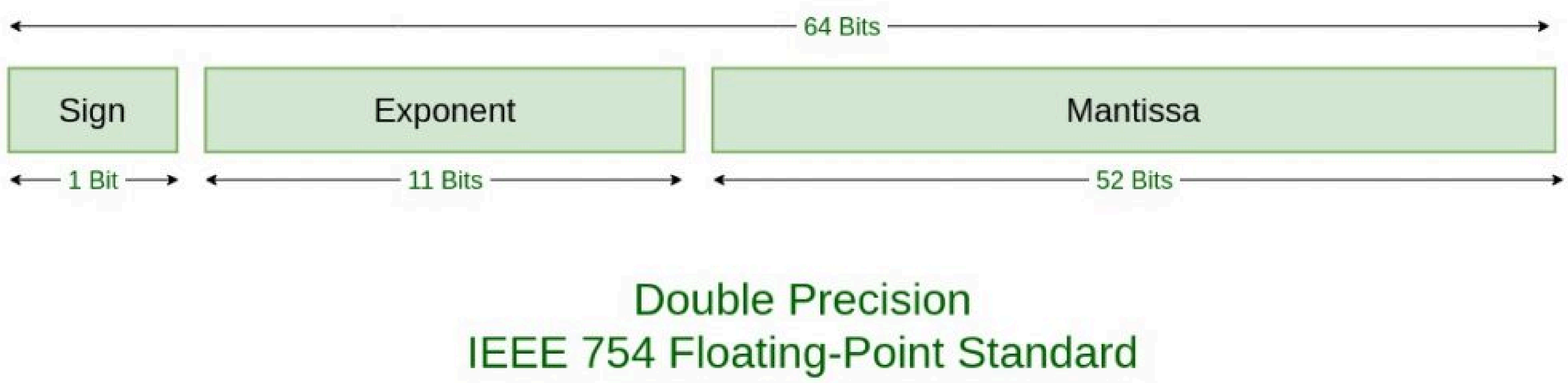
$$x = \pm m \cdot \beta^e$$

Where m is a t-digit fraction called the mantissa, and e is called the exponent.

If the first digit of the mantissa is different from zero, then the floating point number is called normalized. Thus 0.3457×10^5 is a 4-digit normalized decimal floating number, whereas 0.03457×10^6 is a five-digit unnormalized decimal floating point number.



Single Precision



Double Precision

[GeeksForGeeks # IEEE Standard 754 Floating Point Numbers](#)

Examples of overflow and underflow:

1. Let $\beta = 10, t = 3, L = -3, U = 3,$

$$a = 0.111 \times 10^3, \;\; b = 0.120 \times 10^3$$

$$c = a \times b = 0.133 \times 10^5$$

will result in an *overflow* because the exponent 5 is too large

2. Let $\beta = 10, t = 3, L = -2, U = 3,$

$$a = 0.1 \times 10^{-1}, \;\; b = 0.2 \times 10^{-1}$$

$$c = ab = 2 \times 10^{-4}$$

which will result in an underflow

Simple mathematical computations such as finding a square root, or exponent of a number or computing factorials can give *overflow*. For example, consider computing

$$c = \sqrt{a^2 + b^2}$$

> If a or b is very large, then we will get an *overflow*, while computing $a^2 + b^2$

Avoiding Overflow: An Example:

***Overflow* and *Underflow* can sometimes be avoided just by organizing the computations differently. Consider, for example, the task of computing the length**

of an n-vector x with components, denoted by $||x||_2^2$:

$$||x||_2^2 = x_1^2 + x_2^2 + \ldots + x_n^2$$

If some x_i is too big or too small, then we can get overflow or underflow with the usual way of computing $||x||_2$. However, if we normalize each component of the vector by dividing it by $m = max(|x_1|, \ldots, |x_n|)$ and then form the squares and the sum, then overflow problem can be avoided. Thus, a better way to compute $||x||_2^2$ would be the following:

1. $m = max(|x_1|, \ldots, |x_n|)$
2. $y_i = \frac{x_i}{m}, \;\; i = 1, \ldots n$
3. $||x||_2 = m\sqrt{y_1^2 + y_2^2 + \ldots + y_n^2}$

> Numerical Linear Algebra and Applications Second Edition (Biswa Nath Datta)

Let's dive into some Python examples:

```
num = 0.4
binary_rep = format(num, '.30f')

print(f"Decimal: {num}\nBinary: {binary_rep}")
```

```
num = 0.4
binary_rep = format(num, '.30f')

print(f"Decimal: {num}\nBinary: {binary_rep}")
✓ [4] < 10 ms
Decimal: 0.4
Binary: 0.400000000000000022204460492503
```

The errors occur because the computer stores numbers in a binary format with a finite number of bits. Not all decimal fractions (like `0.4`) can be exactly represented in binary.

In base-2, only fractions that can be expressed as a sum of powers of 2 (like `0.5`, `0.25`, etc.) can be represented precisely. However, `0.4` in binary would require an infinite series of bits, so the computer approximates it with the closest possible binary fraction that fits within its fixed number of bits (usually 64 for double-precision)(as I mentioned before)

```
result = 0.0
for i in range(10):
    result += 0.1
print(f"Expected result: 1.0\nActual result: {result}")
```

```python
result = 0.0
for i in range(10):
    result += 0.1
print(f"Expected result: 1.0\nActual result: {result}")
```
✓ [5] < 10 ms

```
Expected result: 1.0
Actual result: 0.9999999999999999
```

Really famous one:

```python
a = 0.2 + 0.1
b = 0.3
print(f"a : {a}\nb : {b}\nEqual: {a == b}")
```

```python
a = 0.2 + 0.1
b = 0.3
print(f"a : {a}\nb : {b}\nEqual: {a == b}")
```
✓ [8] < 10 ms

```
a : 0.30000000000000004
b : 0.3
Equal: False
```

Using Decimal Class for High Precision

```python
result = 1.2 - 1.0
rounded_result = round(result, 2)
print(f"Original Result: {result}\nRounded result: {rounded_result}")
```
✓ [9] < 10 ms

```
Original Result: 0.19999999999999996
Rounded result: 0.2
```

```python
from decimal import Decimal, getcontext

getcontext().prec = 4 # so here we can set precision to 4 decimal places
result = Decimal('1.2') - Decimal('1.0')
print(f"High Precision Result: {result}")
```
✓ [10] 10ms

```
High Precision Result: 0.2
```

```python
result = 1.2 - 1.0
rounded_result = round(result, 2)
print(f"Original Result: {result}\nRounded result: {rounded_result}")

from decimal import Decimal, getcontext
getcontext().prec = 4 # so here we can set precision to 4 decimal places
result = Decimal('1.2') - Decimal('1.0')
print(f"High Precision Result: {result}")
```

Finite differences in two spatial dimensions

Taylor Series for Functions of Several Variables

You've seen Taylor series for functions $y = f(x)$ of 1 variable. For a function $f : \mathbb{R} \to \mathbb{R}$ satisfying the appropriate conditions, we have

$$f(x) = f(c) + f'(c)(x-c) + \frac{f''(c)}{2!}(x-c)^2 + \cdots + \frac{f^{(n)}(c)}{n!}(x-c)^n + R_n(x,c).$$

$R_n(x,c)$ is the **remainder term**:

$$R_n(x,c) = \frac{f^{(n+1)}(z)}{(n+1)!}(x-c)^{n+1}.$$

Numerical Analysis

There is a similar formula for functions of *several variables*,(In our case 2), To make the notation a little better, I'll define higher-order differentials as follows. Let $h = (h_1, h_2, \ldots, h_n) \in R^n$

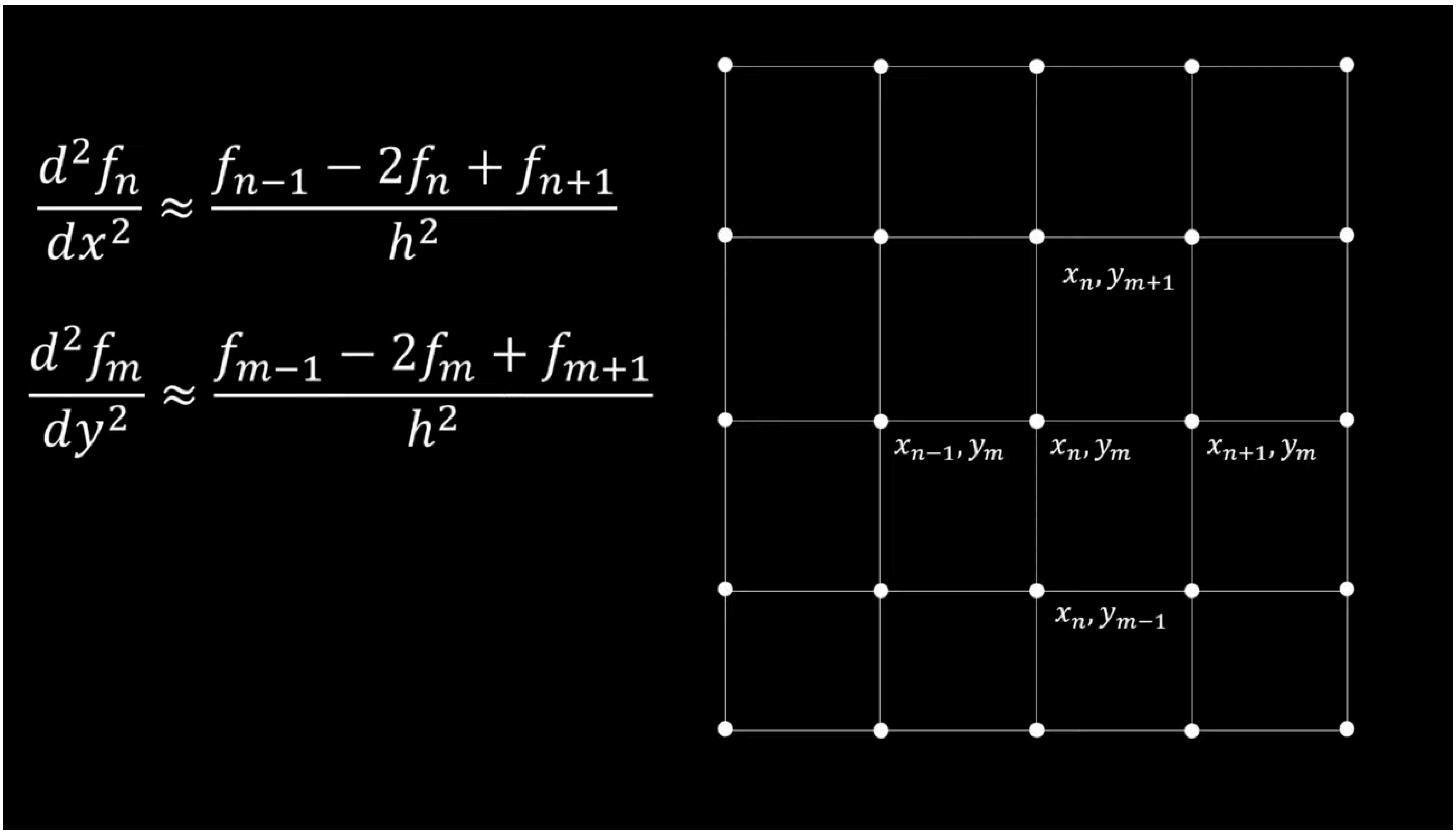
$$D^2 f(x,h) = \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial^2 f}{\partial x_i \partial x_j} \cdot h_i h_j.$$

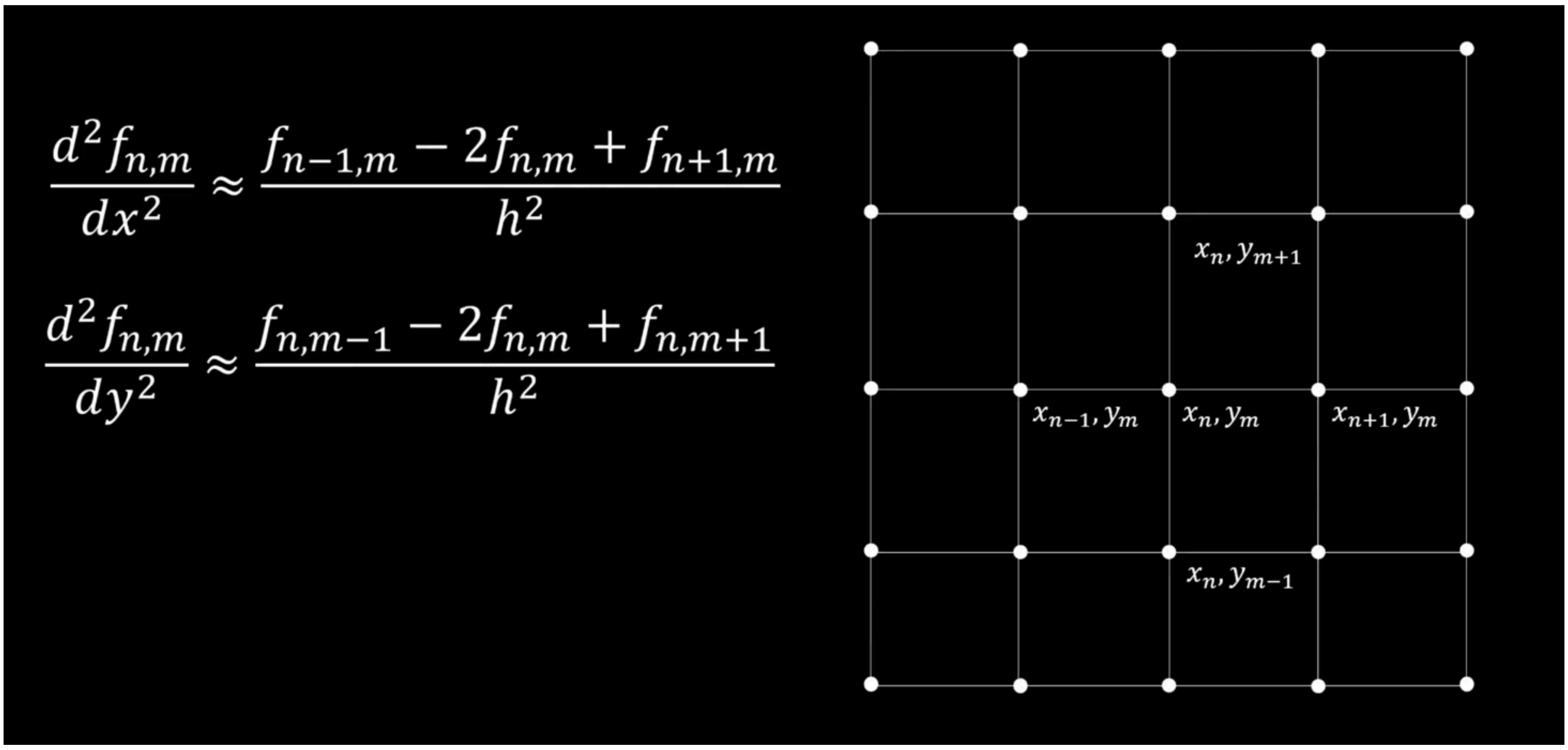
Let's say we're expanding at a point (c, d), Then:

$$f(x,y) = f(c,d) + \left(\frac{\partial f}{\partial x}(c,d)\cdot(x-c) + \frac{\partial f}{\partial y}(c,d)\cdot(y-d)\right) +$$
$$\frac{1}{2!}\left(\frac{\partial^2 f}{\partial x^2}(c,d)(x-c)^2 + 2\frac{\partial^2 f}{\partial x \partial y}(c,d)(x-c)(y-d) + \frac{\partial^2 f}{\partial y^2}(c,d)(y-d)^2\right) + ..$$

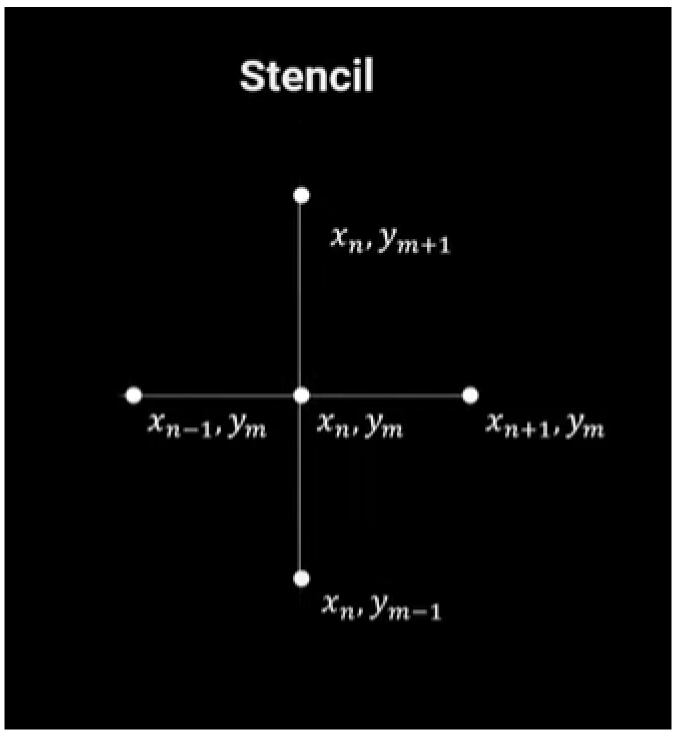
These are the famous finite difference formulas..

$$f_x(x,y) \approx \frac{f(x+h,y) - f(x-h,y)}{2h}$$
$$f_y(x,y) \approx \frac{f(x,y+k) - f(x,y-k)}{2k}$$
$$f_{xx}(x,y) \approx \frac{f(x+h,y) - 2f(x,y) + f(x-h,y)}{h^2}$$
$$f_{yy}(x,y) \approx \frac{f(x,y+k) - 2f(x,y) + f(x,y-k)}{k^2}$$
$$f_{xy}(x,y) \approx \frac{f(x+h,y+k) - f(x+h,y-k) - f(x-h,y+k) + f(x-h,y-}{4hk}$$





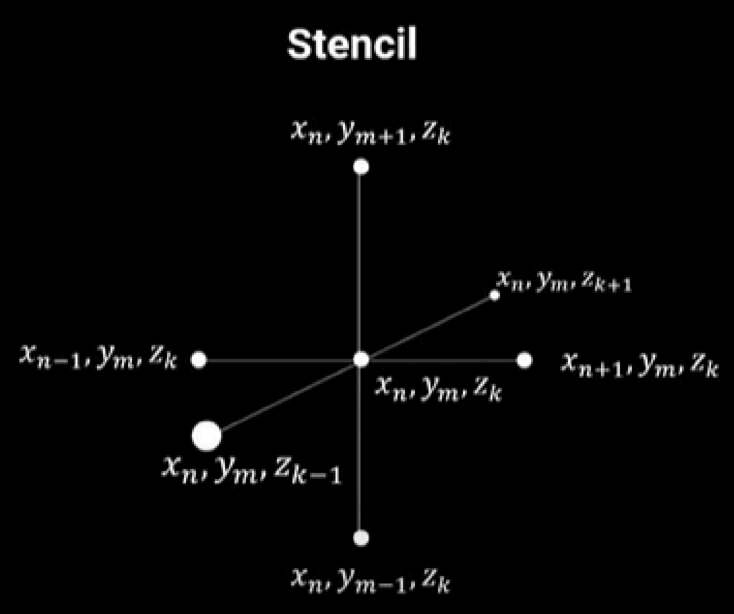
The nodes at which we evaluate the function is called a stencil



$$\frac{d^2 f_{n,m,k}}{dx^2} \approx \frac{f_{n-1,m,k} - 2f_{n,m,k} + f_{n+1,m,k}}{h^2}$$

$$\frac{d^2 f_{n,m,k}}{dy^2} \approx \frac{f_{n,m-1,k} - 2f_{n,m,k} + f_{n,m+1,k}}{h^2}$$

$$\frac{d^2 f_{n,m,k}}{dz^2} \approx \frac{f_{n,m,k-1} - 2f_{n,m,k} + f_{n,m,k+1}}{h^2}$$



> In case of 3D it would be something like that

So now lets use these approximations to solve the following equation

$$\frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} = -(\alpha^2 + \beta^2) f(x,y)$$

Boundary conditions:

$$\begin{aligned} f(-1,y) &= \sin(-\alpha)\sin(\beta y), \\ f(1,y) &= \sin(\alpha)\sin(\beta y), \\ f(x,-1) &= \sin(\alpha x)\sin(-\beta), \\ f(x,1) &= \sin(\alpha x)\sin(\beta). \end{aligned}$$

$\alpha = 1.5\pi$

$\beta = 2.5\pi$

so if anyone is interested the solution of that is :

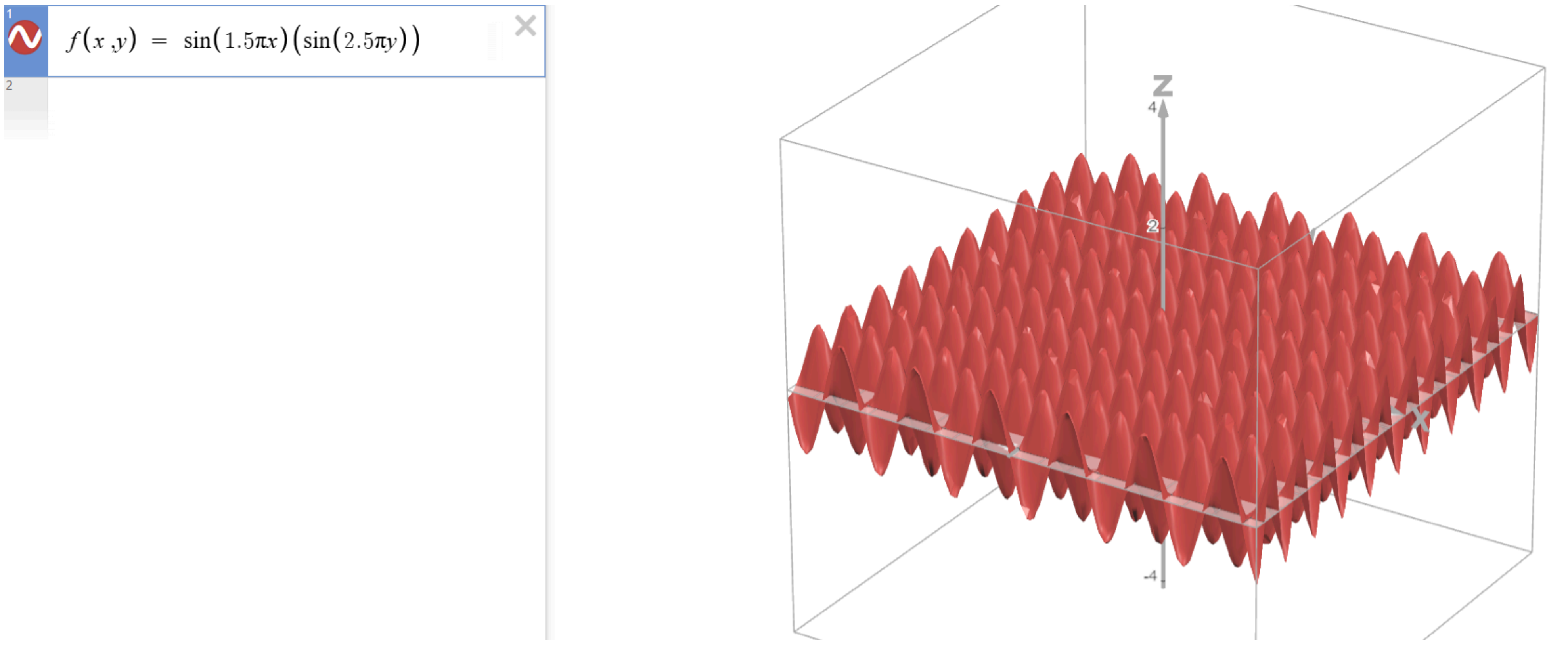
$f(x,y) = sin(\alpha x)sin(\beta y)$

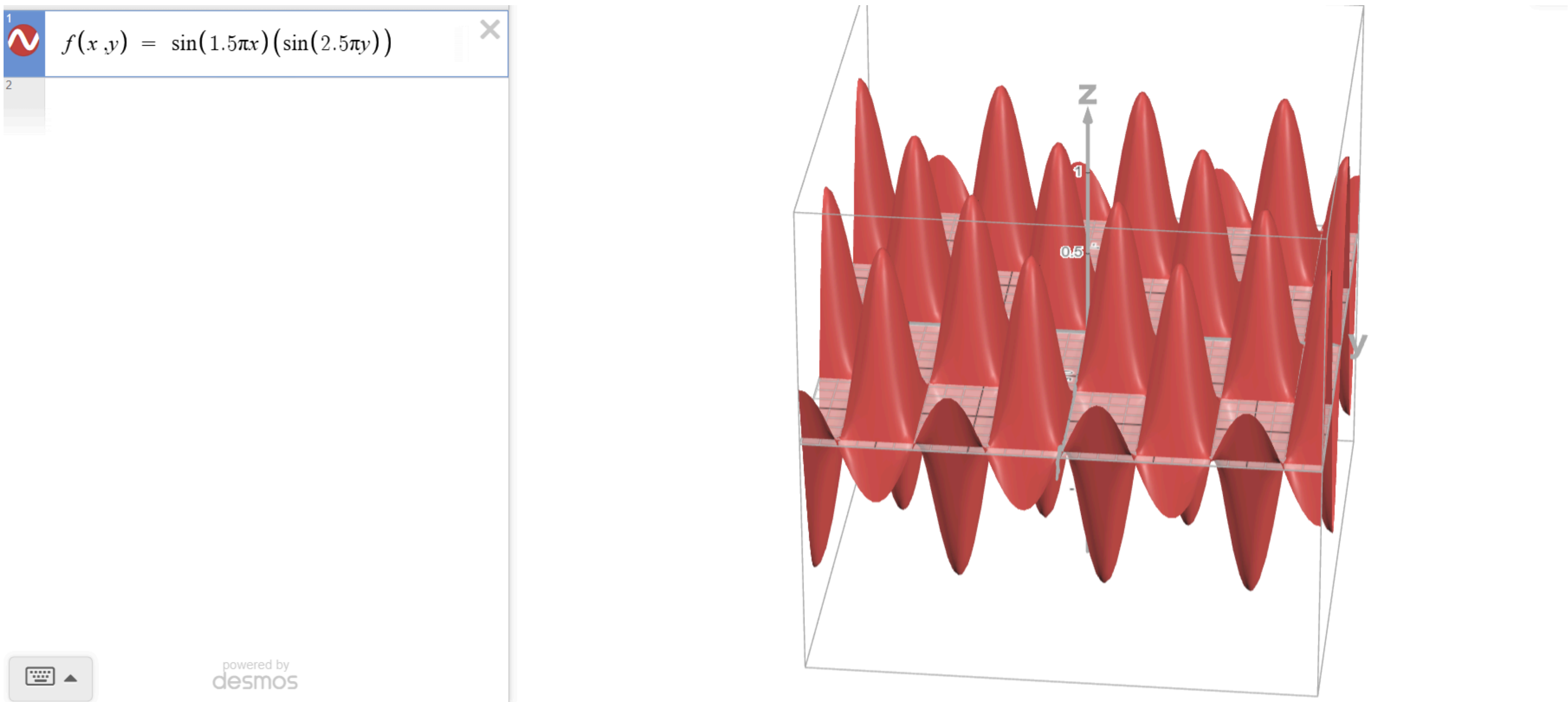
Let's get into work

Substitute finite difference formulas into the equation:

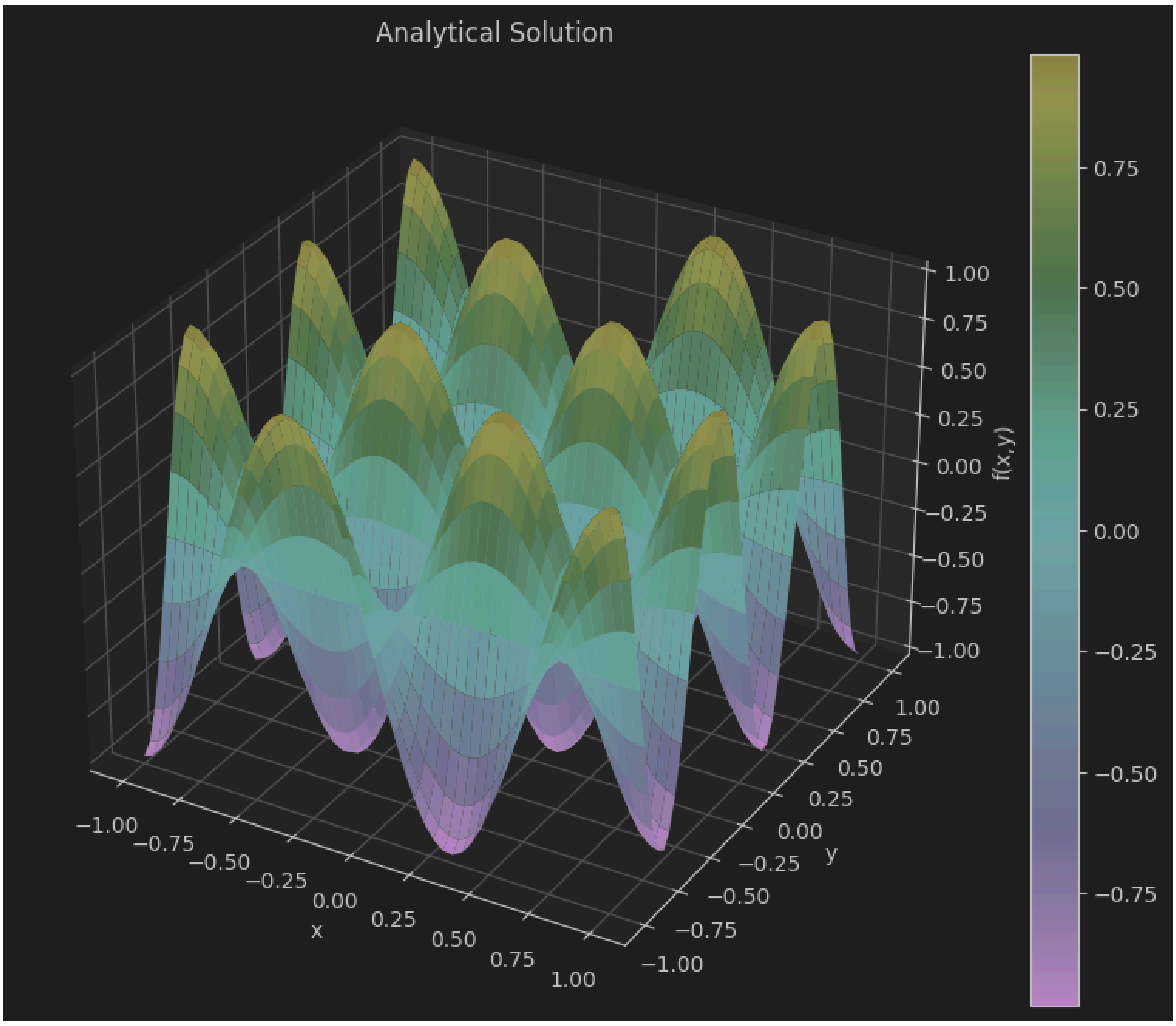
$$\frac{f(x_{i+1}, y_j) - 2f(x_i, y_j) + f(x_{i-1}, y_j)}{h^2} +$$
$$+ \frac{f(x_i, y_{j+1}) - 2f(x_i, y_j) + f(x_i, y_{j-1})}{h^2} = -(\alpha^2 + \beta^2) f(x_i, y_j)$$

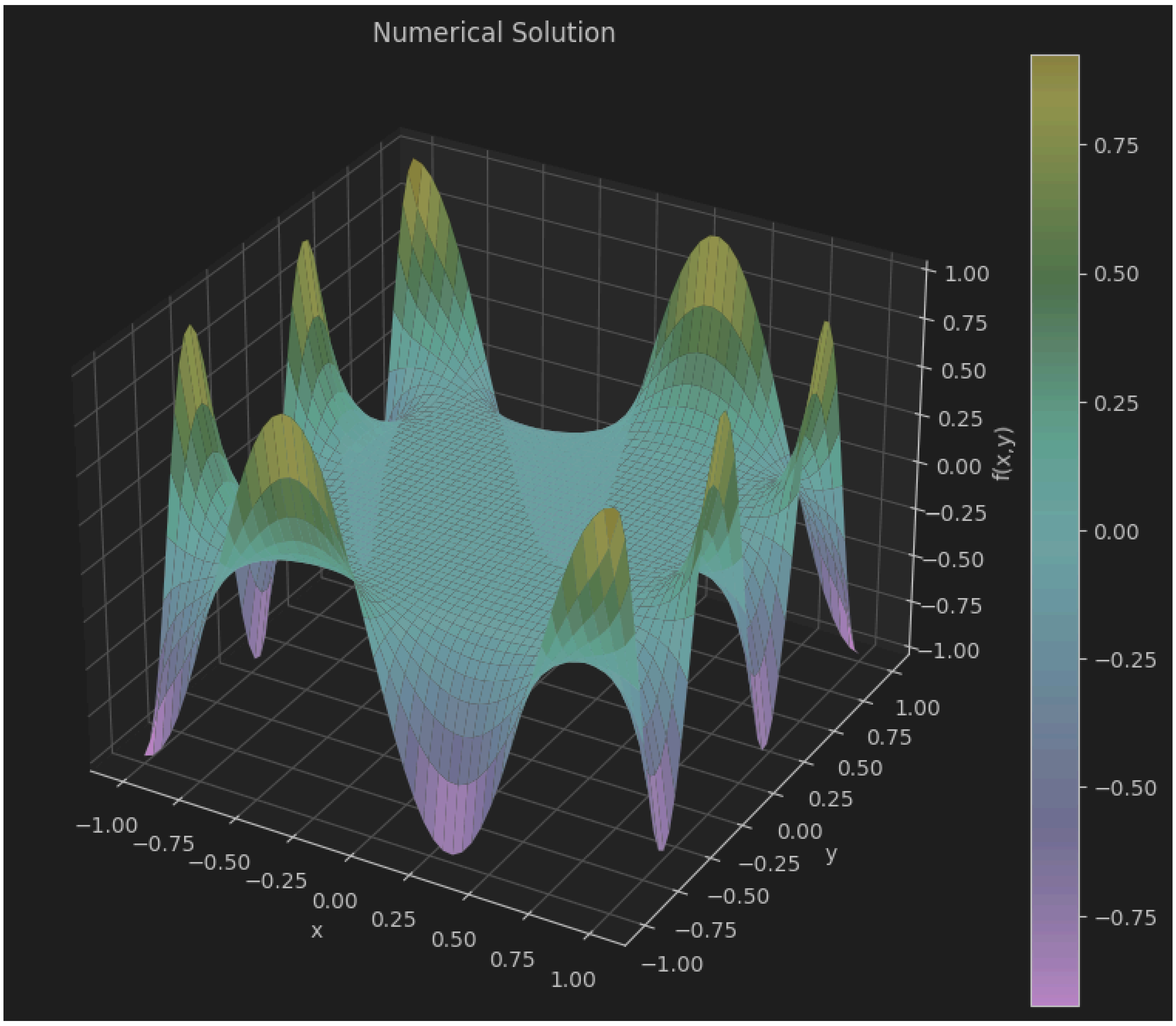
> We get that

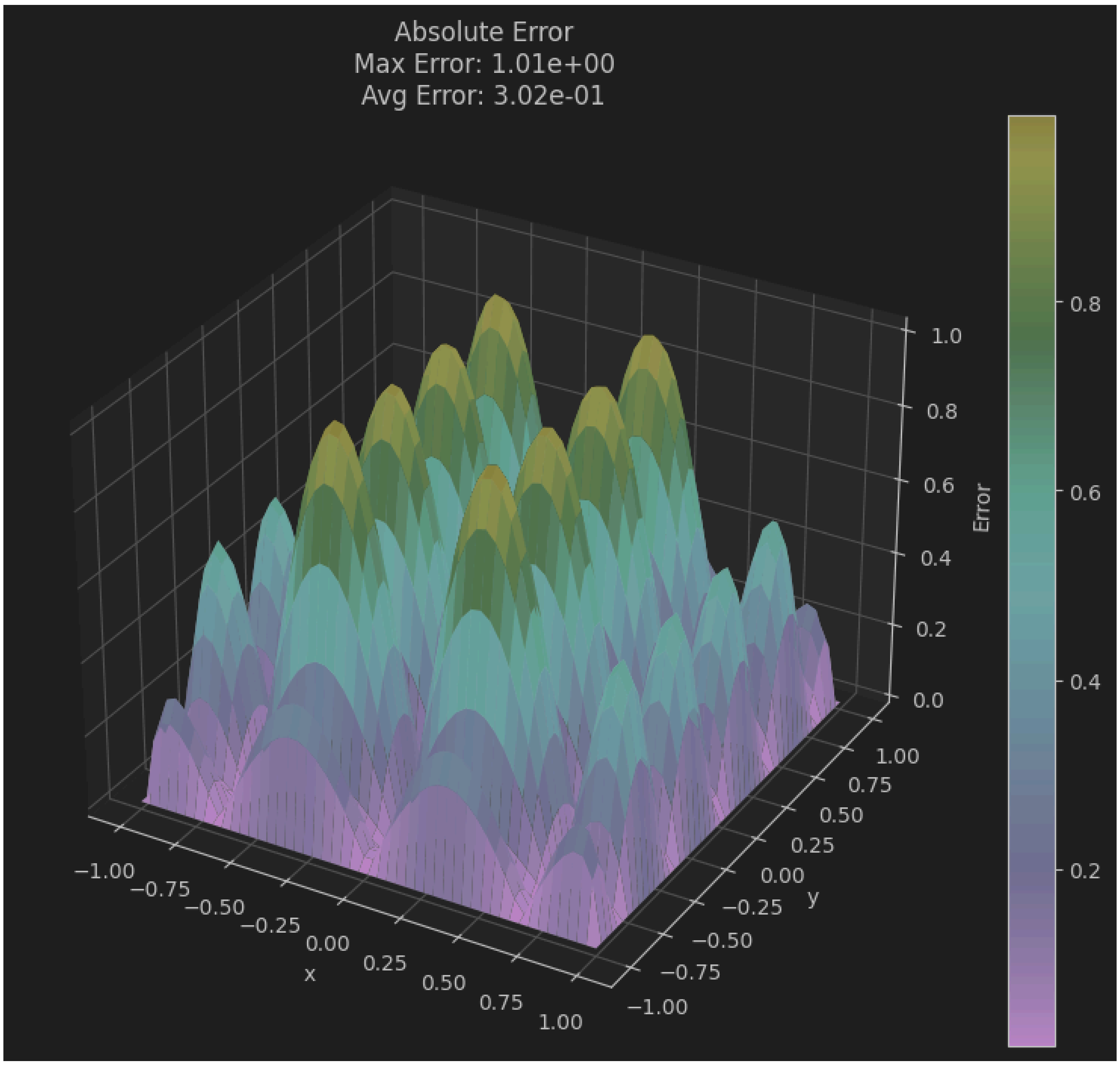




This is the original solution


Analytical Solution
x
y
f(x,y)


Numerical Solution
x
y
f(x,y)


Absolute Error
Max Error: 1.01e+00
Avg Error: 3.02e-01
x
y
Error
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
0.0
0.2
0.4
0.6
0.8
1.0

That's it