Numerical Analysis

Dr.Eng. Muhammad Zulhaj Aliansyah, S.T., M.Eng.

Approximation Theory

Nov 2025

Presented by

Nino & Rafly

Content

What & Why?

What is Approximation Theory? Approximation Theory is a branch of mathematics that studies how to find simple functions that closely approximate more complex functions or data. It focuses on minimizing the difference (error) between the true function and its approximation.

Why Is it Important?

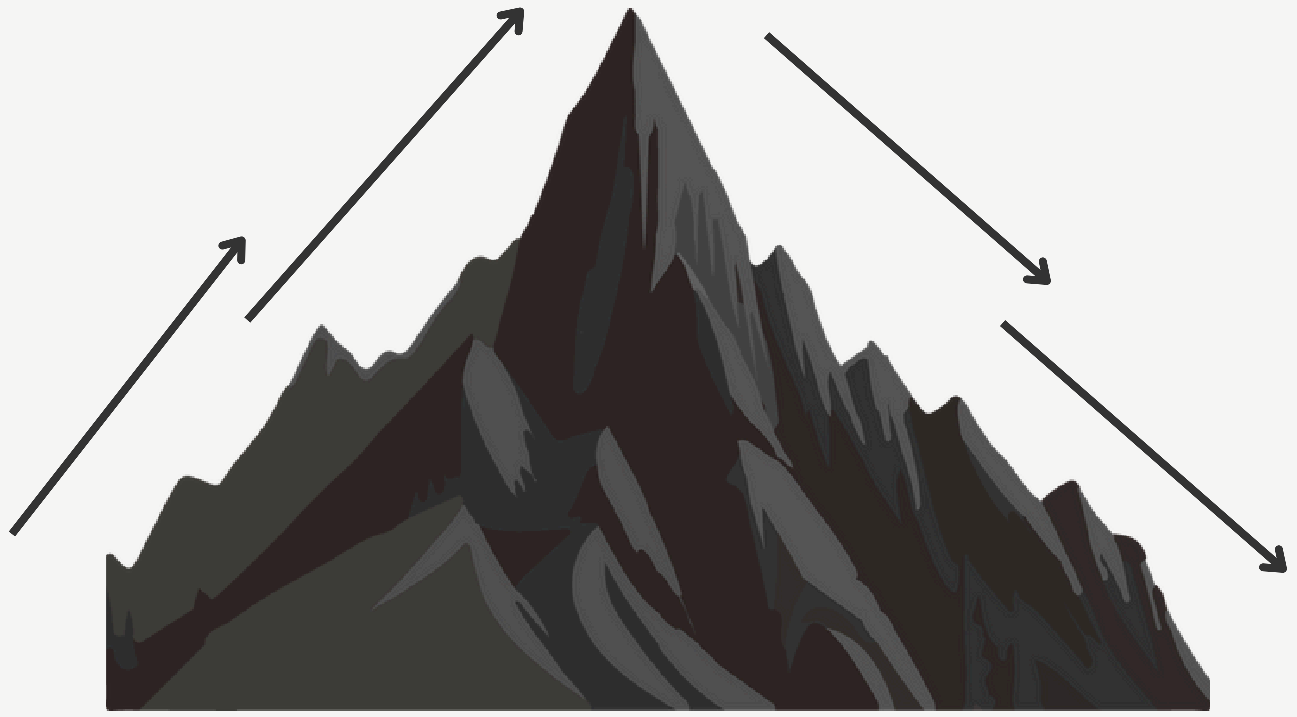
- Real-world data is often noisy or difficult to model exactly.
- Simplified models make computation, prediction, and analysis easier.
- Used in **data fitting**, **numerical simulation**, and **machine learning.**

Definition & Theorical

What & Why?

Analogy

Imagine trying to draw a mountain range with just a few smooth curves you don’t capture every rock, but you still represent the overall shape accurately.



Approximating complexity through simplicity.

Theorical Basis

Example:

$$\text{residual sum of squares} = \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

Simply put, Approximation Theory seeks to find a function or model that can produce a predicted value y topi (the regression value of the target variable) that is closest to the actual value y (the target variable).

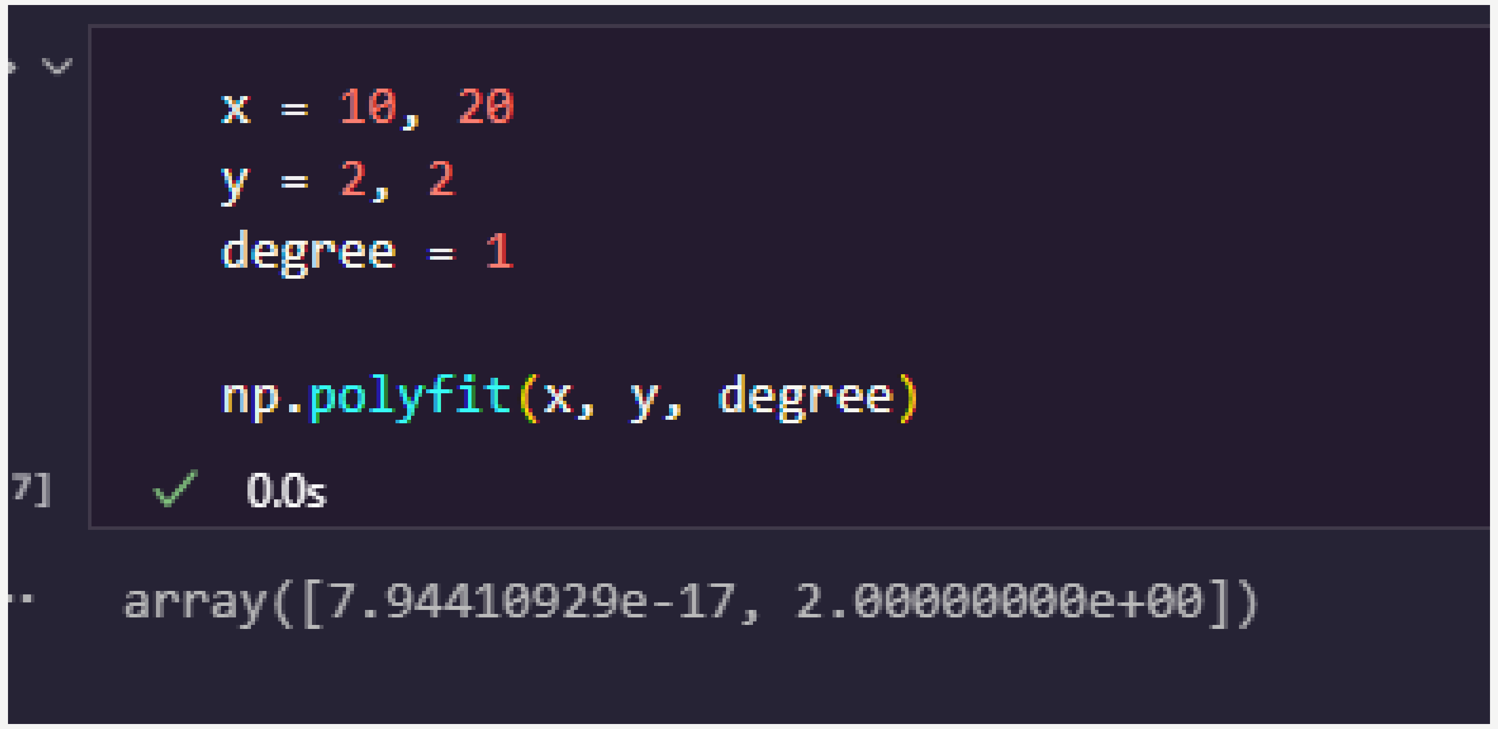
Purpose

To find the best-fitting for a set of discreate data points (xi,yi) by minimizing the total squared error

$$\textit{residual sum of squares} = \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

Instead of passing exactly through every point, the model finds a function that balances the error giving the smallest overall deviation from all data.

Python Implementation

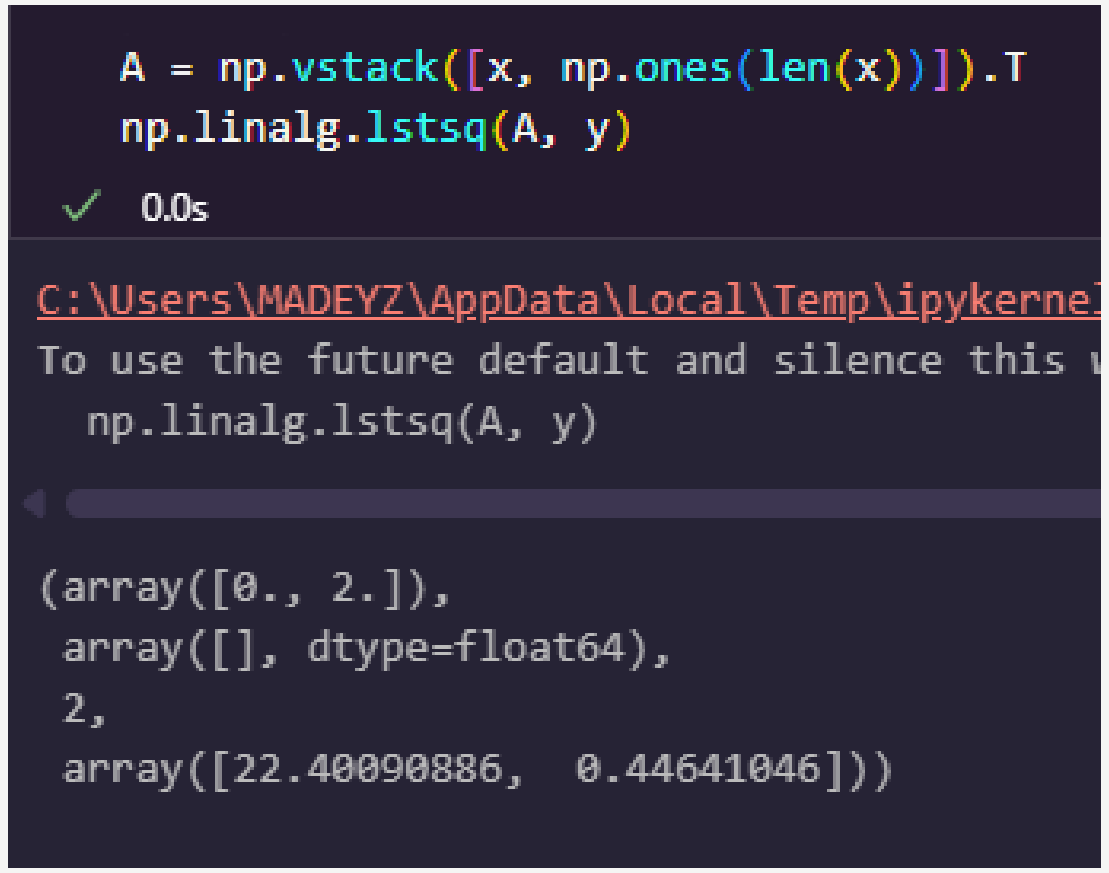


```
x = 10, 20
y = 2, 2
degree = 1

np.polyfit(x, y, degree)
```

```
✓ 0.0s
```

```
array([7.94410929e-17, 2.00000000e+00])
```

fits Polynomial of Given Degree



```
A = np.vstack([x, np.ones(len(x))]).T
np.linalg.lstsq(A, y)
```

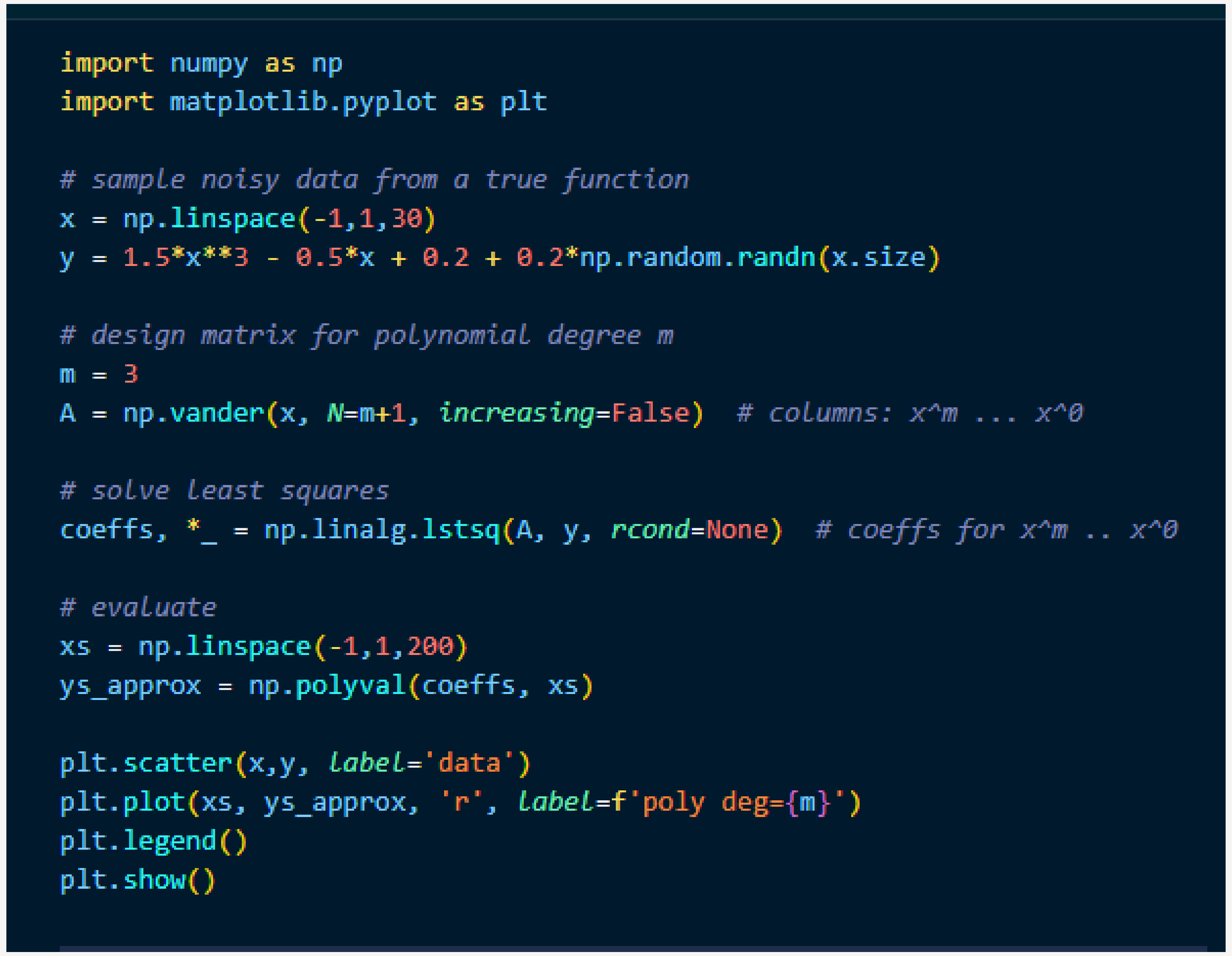
```
✓ 0.0s
```

```
C:\Users\MADEYZ\AppData\Local\Temp\ipykerne
To use the future default and silence this
  np.linalg.lstsq(A, y)

(array([0., 2.]),
 array([], dtype=float64),
 2,
 array([22.40090886,  0.44641046]))
```

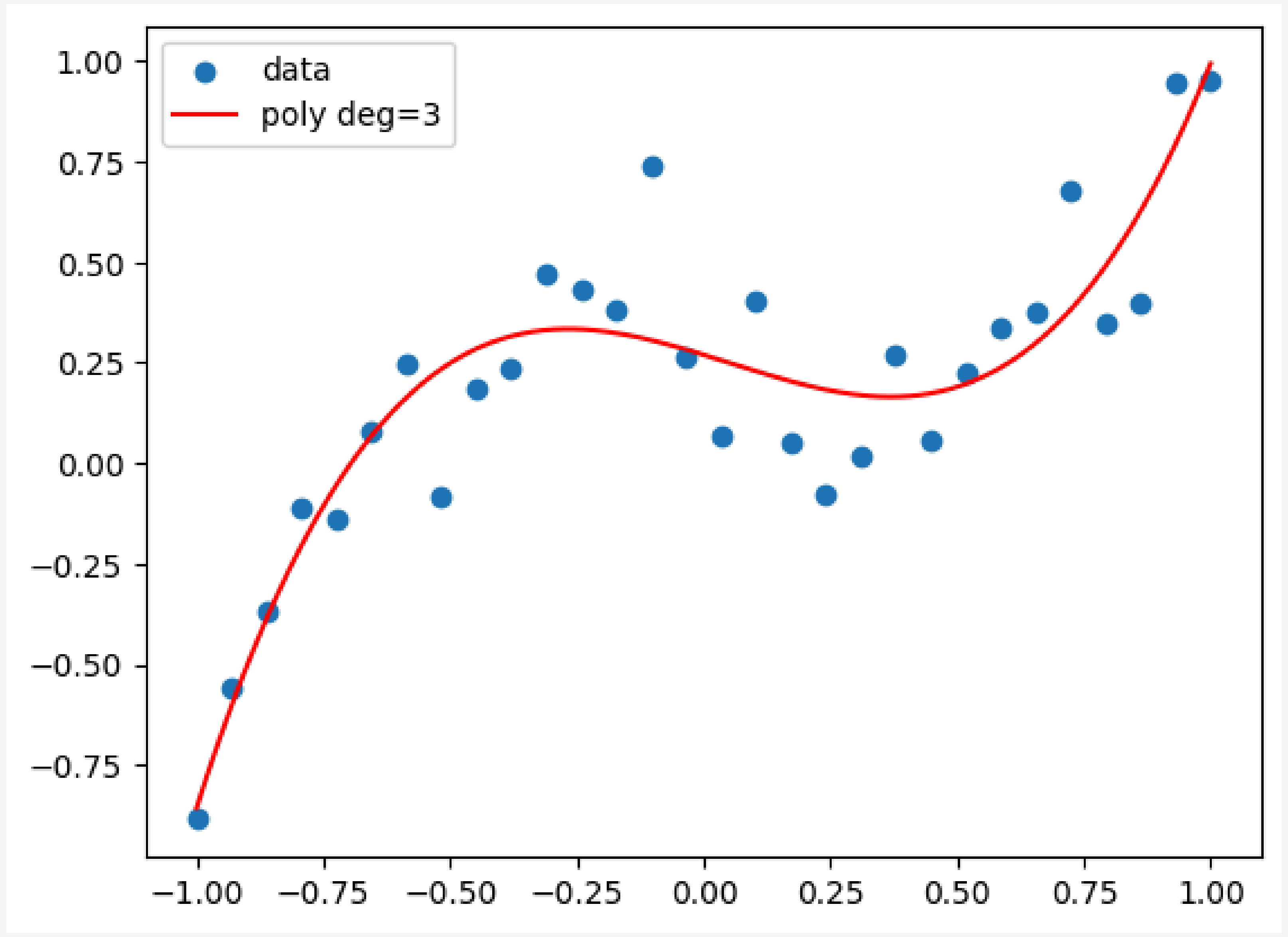
Solves Using Least Squares System

Discrete Least Square

Python Implementation



```python
import numpy as np
import matplotlib.pyplot as plt

# sample noisy data from a true function
x = np.linspace(-1,1,30)
y = 1.5*x**3 - 0.5*x + 0.2 + 0.2*np.random.randn(x.size)

# design matrix for polynomial degree m
m = 3
A = np.vander(x, N=m+1, increasing=False)  # columns: x^m ... x^0

# solve least squares
coeffs, *_ = np.linalg.lstsq(A, y, rcond=None)  # coeffs for x^m .. x^0

# evaluate
xs = np.linspace(-1,1,200)
ys_approx = np.polyval(coeffs, xs)

plt.scatter(x,y, label='data')
plt.plot(xs, ys_approx, 'r', label=f'poly deg={m}')
plt.legend()
plt.show()
```



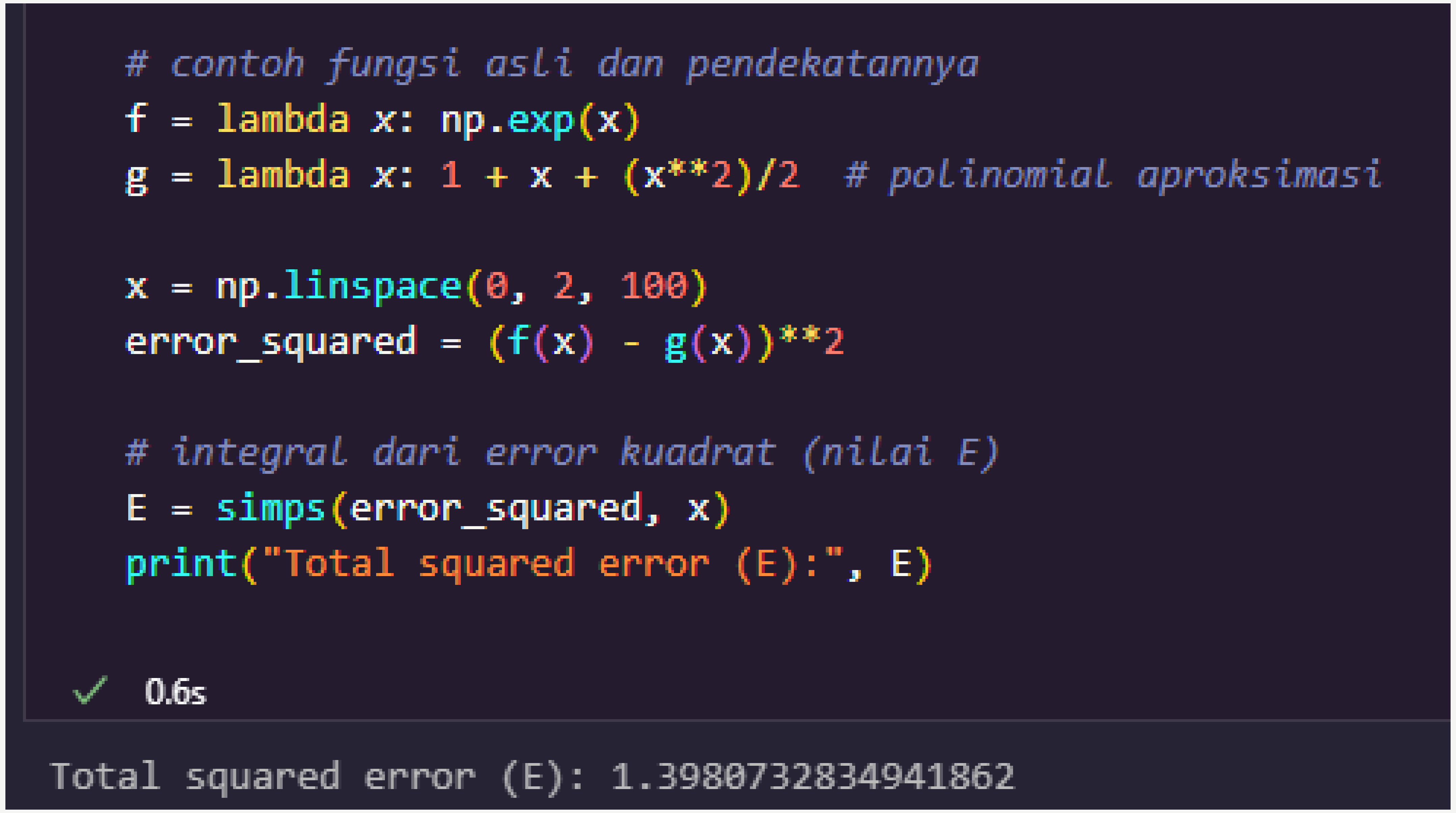
Purpose?

To find the best-fitting continuous function that minimizes the total squared difference between the true function f(x) and the approximating function f^(x)

$$\int_a^b [f(x) - \hat{f}(x)]^2 dx$$

- Similar idea as the discrete case, but now we use integrals instead of summations.
- The goal is to make the area of the squared error as small as possible.

Python Implementation



```
# contoh fungsi asli dan pendekatannya
f = lambda x: np.exp(x)
g = lambda x: 1 + x + (x**2)/2  # polinomial aproksimasi

x = np.linspace(0, 2, 100)
error_squared = (f(x) - g(x))**2

# integral dari error kuadrat (nilai E)
E = simps(error_squared, x)
print("Total squared error (E):", E)
```

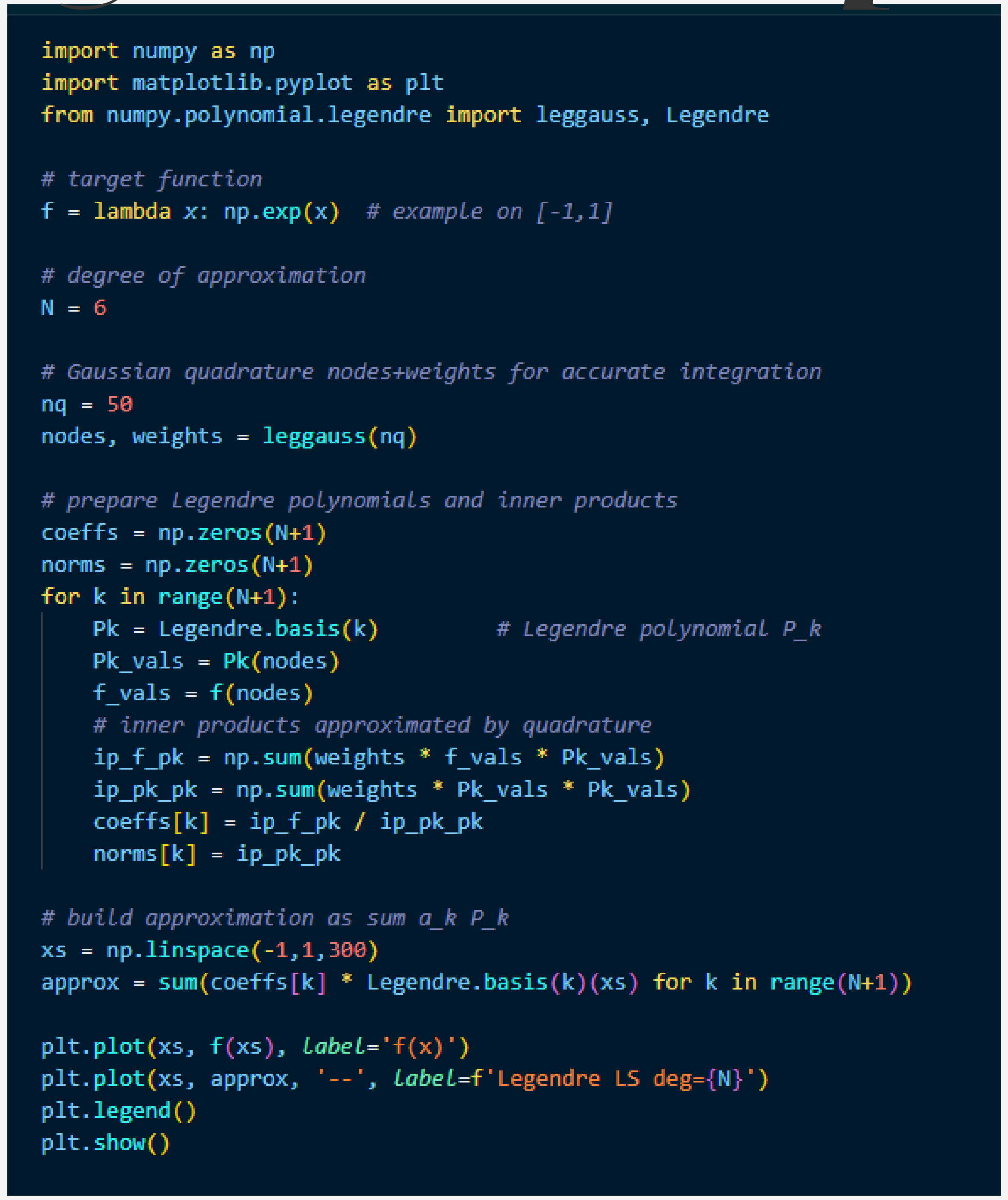
```
✓ 0.6s
```

```
Total squared error (E): 1.3980732834941862
```

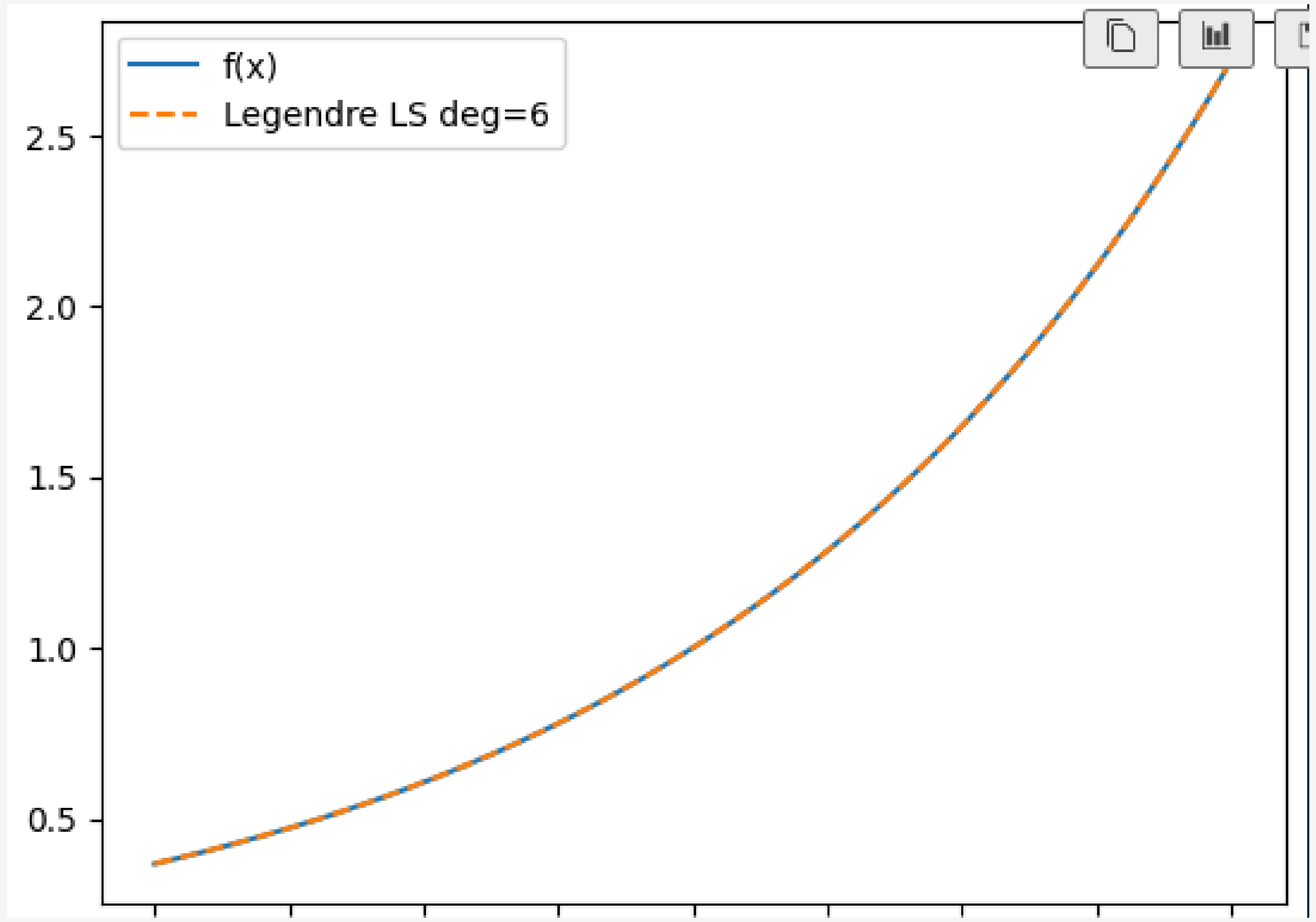
The least squares method for continuous data focuses on minimizing the total error area, ensuring the approximating function behaves closely to the original across the entire interval [a,b].

Continuous Least Squares

Python Implementation



```python
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial.legendre import leggauss, Legendre

# target function
f = lambda x: np.exp(x)  # example on [-1,1]

# degree of approximation
N = 6

# Gaussian quadrature nodes+weights for accurate integration
nq = 50
nodes, weights = leggauss(nq)

# prepare Legendre polynomials and inner products
coeffs = np.zeros(N+1)
norms = np.zeros(N+1)
for k in range(N+1):
    Pk = Legendre.basis(k)          # Legendre polynomial P_k
    Pk_vals = Pk(nodes)
    f_vals = f(nodes)
    # inner products approximated by quadrature
    ip_f_pk = np.sum(weights * f_vals * Pk_vals)
    ip_pk_pk = np.sum(weights * Pk_vals * Pk_vals)
    coeffs[k] = ip_f_pk / ip_pk_pk
    norms[k] = ip_pk_pk

# build approximation as sum a_k P_k
xs = np.linspace(-1,1,300)
approx = sum(coeffs[k] * Legendre.basis(k)(xs) for k in range(N+1))

plt.plot(xs, f(xs), label='f(x)')
plt.plot(xs, approx, '--', label=f'Legendre LS deg={N}')
plt.legend()
plt.show()
```



Purpose?

When using ordinary polynomials (1,x,x^2 ,...) numerical instability and large rounding errors often appear.

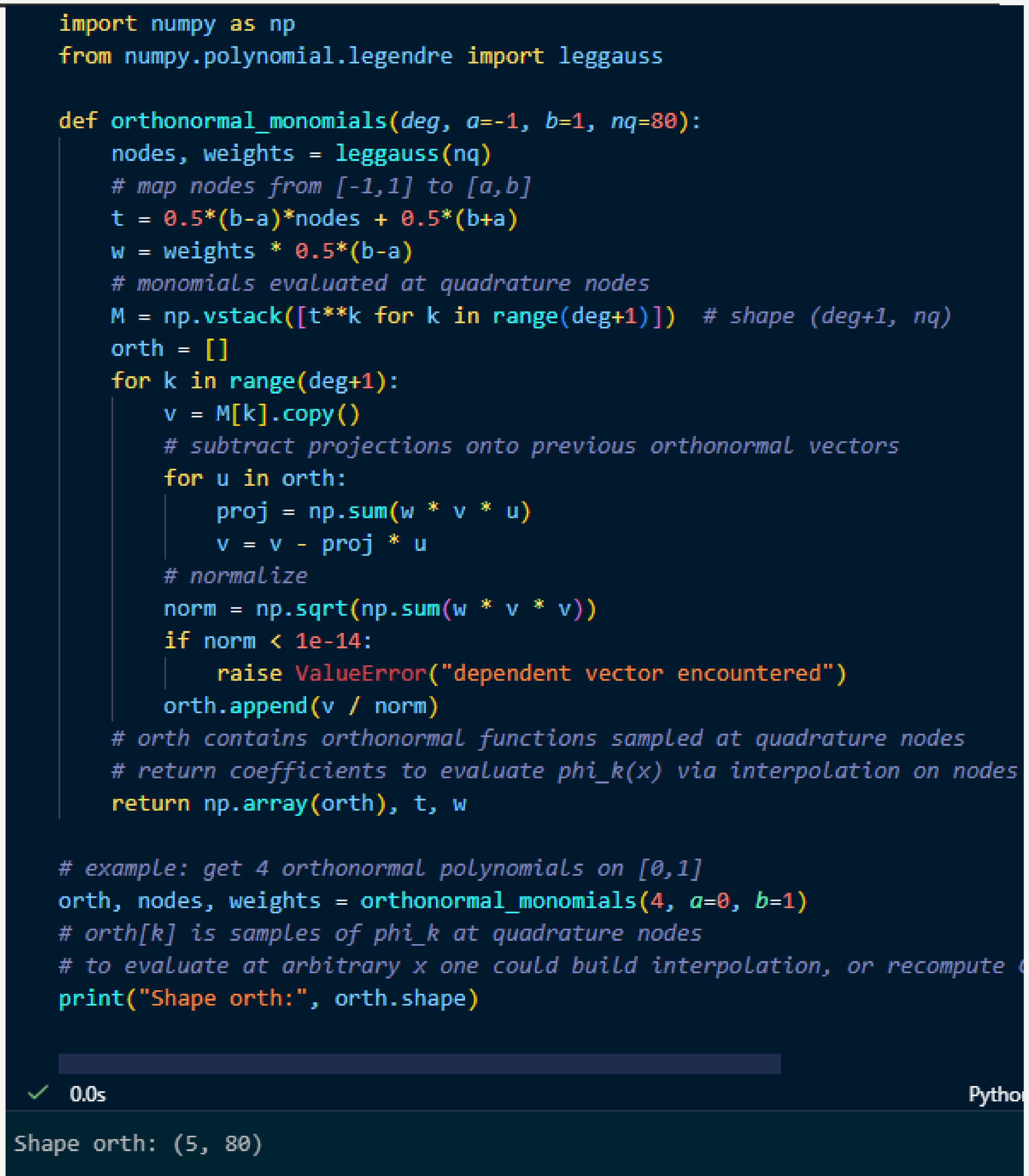
To solve this, we use orthogonal polynomias, function that are *mutually independent* and satisfy

$$\langle \phi_i, \phi_j \rangle = \int_a^b w(x)\phi_i(x)\phi_j(x)\,dx = 0 \quad (i \neq j)$$

where w(x) is a weighting function

- Avoids numerical instability from correlated terms (x, x^2, x^3, ...).
- Simplifies computation — no need to solve large systems of equations.
- Produces more accurate and efficient approximations.

Python Implementation



```python
import numpy as np
from numpy.polynomial.legendre import leggauss

def orthonormal_monomials(deg, a=-1, b=1, nq=80):
    nodes, weights = leggauss(nq)
    # map nodes from [-1,1] to [a,b]
    t = 0.5*(b-a)*nodes + 0.5*(b+a)
    w = weights * 0.5*(b-a)
    # monomials evaluated at quadrature nodes
    M = np.vstack([t**k for k in range(deg+1)])  # shape (deg+1, nq)
    orth = []
    for k in range(deg+1):
        v = M[k].copy()
        # subtract projections onto previous orthonormal vectors
        for u in orth:
            proj = np.sum(w * v * u)
            v = v - proj * u
        # normalize
        norm = np.sqrt(np.sum(w * v * v))
        if norm < 1e-14:
            raise ValueError("dependent vector encountered")
        orth.append(v / norm)
    # orth contains orthonormal functions sampled at quadrature nodes
    # return coefficients to evaluate phi_k(x) via interpolation on nodes
    return np.array(orth), t, w

# example: get 4 orthonormal polynomials on [0,1]
orth, nodes, weights = orthonormal_monomials(4, a=0, b=1)
# orth[k] is samples of phi_k at quadrature nodes
# to evaluate at arbitrary x one could build interpolation, or recompute
print("Shape orth:", orth.shape)
```

✓ 0.0s Pytho

```
Shape orth: (5, 80)
```

Main Result

Summary Of Approximation Theory

Approximation Theory focuses on finding a simple function that best represents a complex function or dataset, by minimizing the difference between the true and predicted values.

- Discrete Least Squares → minimizes error at specific data points using summation.
- Continuous Least Squares → minimizes the total error area using integration.

Main Result

Python Implementation

- **Tools: numpy.polyfit(), np.linalg.lstsq(), scipy.integrate.simps()**
- **Functions from the text: leastsqfit(), compsimpson(), polyChebCoeff()**

- Core workflow:
    a. Define data or target function
    b. Fit polynomial (discrete or continuous)
    c. Evaluate the approximation visually or numerically

Thank you

For your attention

Nino & Rafly