Chapter-08 Numerical Differentiation

Date	@November 1, 2024
tag	done

▼ First and Second Order Difference Approximations

Forward Differences

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

Steps:

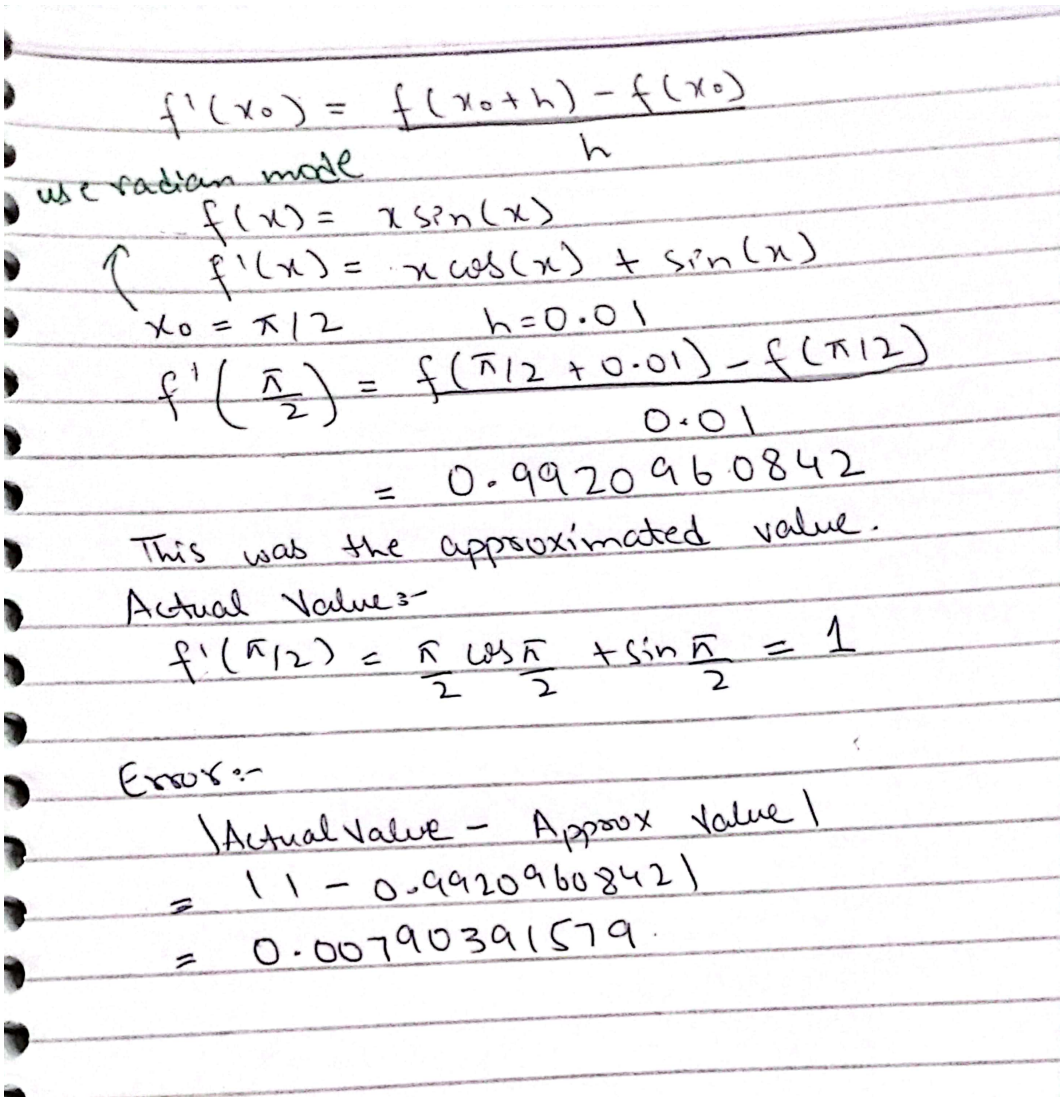
- Find Approximate Value using above formula
- Find Actual Derivative Value
- Compute the Error using |Actiual Value - Aprrox Value|

```
import numpy as np
def f(x):
    y = x*np.sin(x)
    return y

x0 = np.pi/2

h = 0.01

derivative = ( f(x0+h) - f(x0) ) / h
print('df(x) =',derivative, 'Error =',np.abs(derivative-1.0)
```

df(x) = 0.992096084232319 Error = 0.007903915767681013

> The smaller the h, the lesser the error, the closer the approximation!

if h=0.001 → Error = 0.0007858980980426367

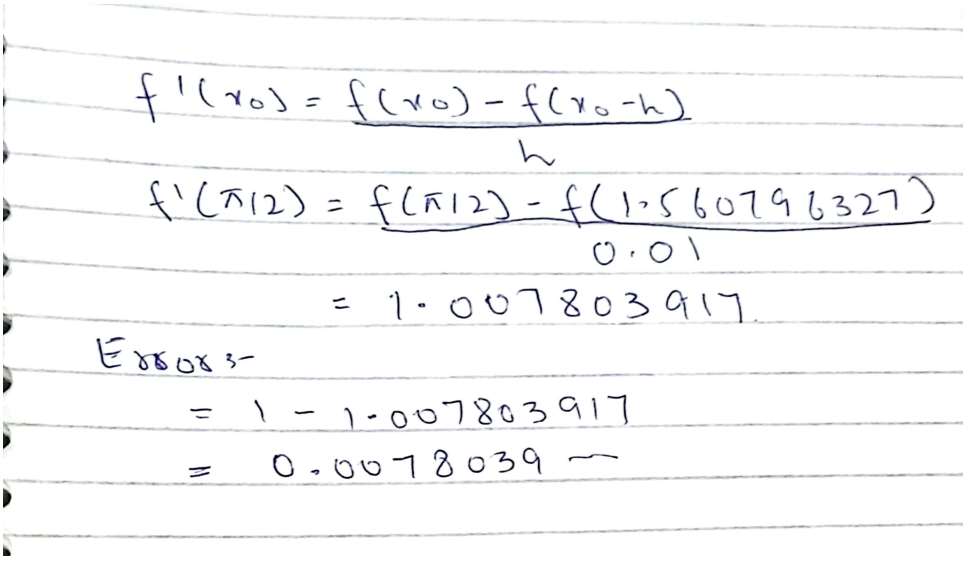


Backward Differences

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - h)}{h}$$

```
x0 = np.pi/2
h = 0.01

derivative = ( f(x0) - f(x0-h) ) / h
print('df(x) =',derivative, 'Error =',np.abs(derivative-1.0))
```

df(x) = 1.0078039166010244 Error = 0.007803916601024419



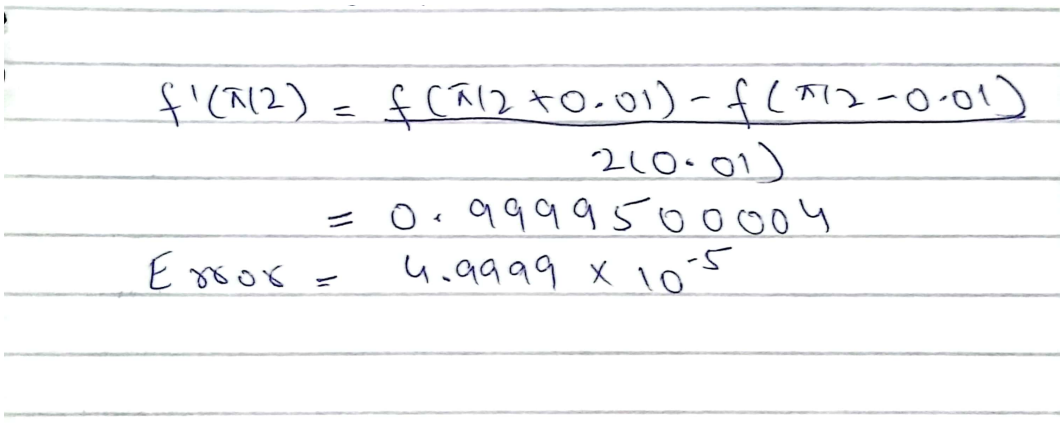
Central Differences/ Second Order Finite Difference

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0-h)}{2h}$$

```
x0 = np.pi/2.0
h = 0.01

derivative = (f(x0+h) - f(x0-h))/(2.0*h)
print('df(x)=',derivative, 'Error=',np.abs(derivative-1.0))
```

df(x)= 0.9999500004166717 Error= 4.999958332829735e-05

> Observe that for h=0.01 the forward and backward differences give the derivative with 2 correct decimal digits, while the central difference approximation gives the derivative with 4 correct decimal digits. The reason for that is that the central difference approximation has convergence rate 2 while for the other approximations is 1.



▼ Second-Order Derivatives

$$f''(x_0) = \frac{f(x_0+h) - 2f(x_0) + f(x_0-h)}{h^2} + \mathcal{O}(h^2)$$

$f''(x) = 2\cos(x) - x\sin(x)$

$f''(\pi/2) = -\pi/2$

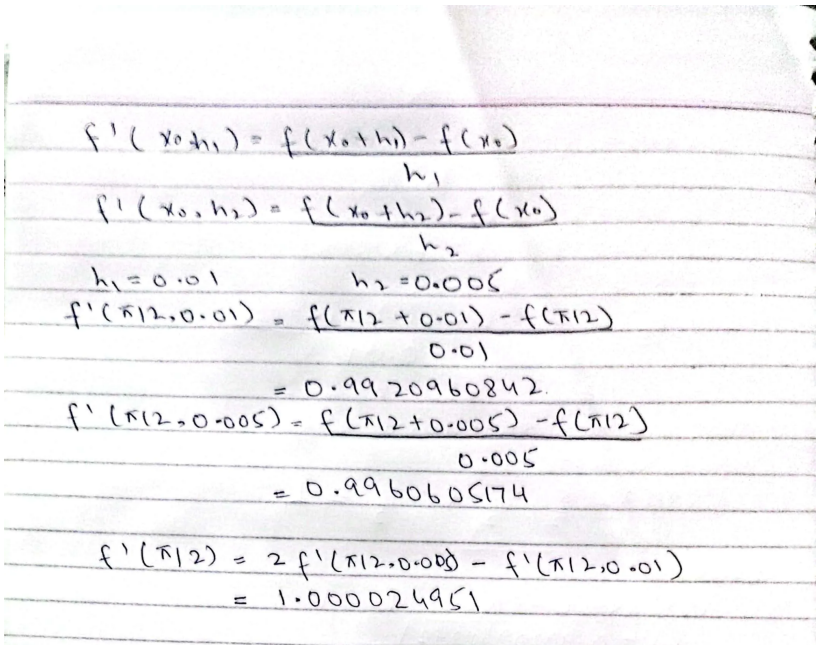
```
x0 = np.pi/2
h = 0.01

derivative2 = (f(x0+h) - 2.0*f(x0) + f(x0-h))/(h**2)
print('d2f(x) =',derivative2, 'Error =',np.abs(derivative2-(-
```

d2f(x) = -1.5707832368705432 Error = 1.3089924353337778e-05

▼ Richardson’s Extrapolation

Forward Difference

```
#forward
x0 = np.pi/2
h = 0.01

d1 = (f(x0+h) - f(x0))/h
d2 = (f(x0+h/2.0) - f(x0))/(h/2.0)

derivative = 2.0*d2-d1
print('df(x) =',derivative, 'Error =',np.abs(derivative-1.0))
```



$$f'(x_0, h_1) = \frac{f(x_0+h_1) - f(x_0)}{h_1}$$

$$f'(x_0, h_2) = \frac{f(x_0+h_2) - f(x_0)}{h_2}$$

$h_1 = 0.01 \qquad h_2 = 0.005$

$$f'(\pi/2, 0.01) = \frac{f(\pi/2 + 0.01) - f(\pi/2)}{0.01}$$

$$= 0.9920960842$$

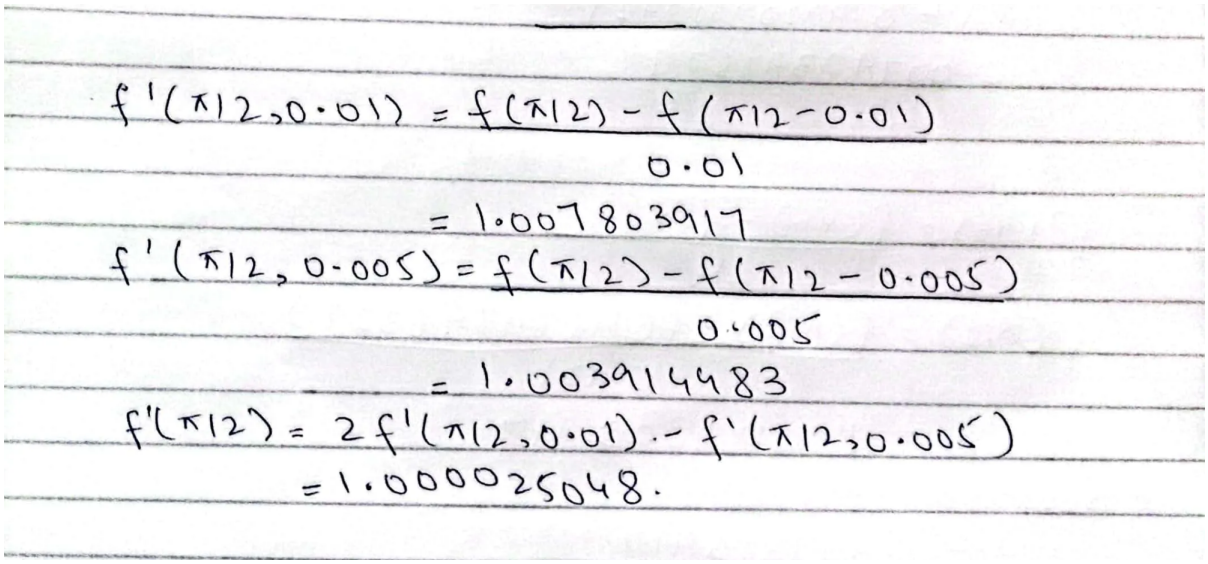
$$f'(\pi/2, 0.005) = \frac{f(\pi/2 + 0.005) - f(\pi/2)}{0.005}$$

$$= 0.9960605174$$

$$f'(\pi/2) = 2f'(\pi/2, 0.005) - f'(\pi/2, 0.01)$$

$$= 1.000024951$$

Backward Difference



$$f'(\pi/2, 0.01) = \frac{f(\pi/2) - f(\pi/2 - 0.01)}{0.01}$$

$$= 1.007803917$$

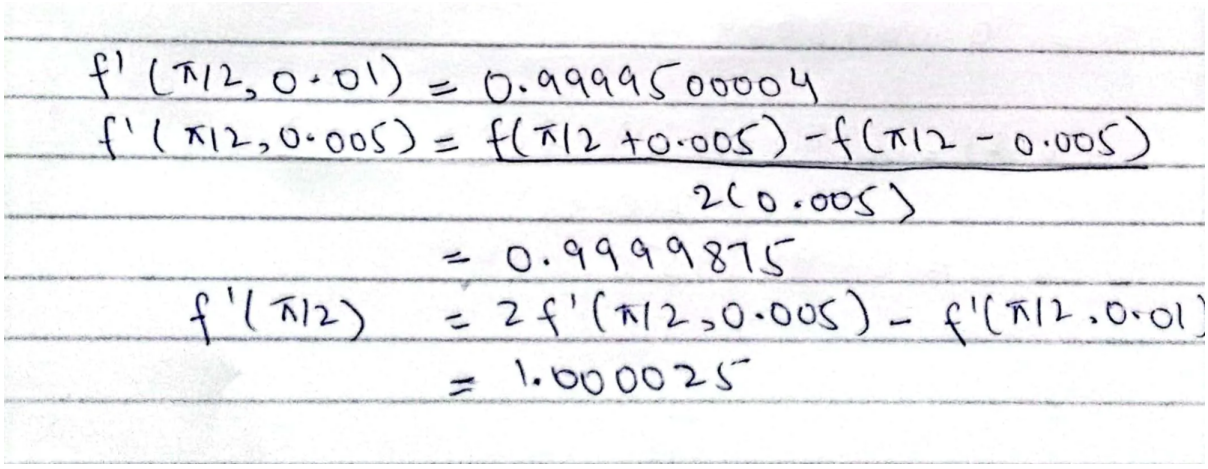
$$f'(\pi/2, 0.005) = \frac{f(\pi/2) - f(\pi/2 - 0.005)}{0.005}$$

$$= 1.003914483$$

$$f'(\pi/2) = 2f'(\pi/2, 0.01) - f'(\pi/2, 0.005)$$

$$= 1.000025048.$$

Central Difference



$$D = \frac{2^p d(h_1/2) - d(h_1)}{2^p - 1}$$

Taking p=1

$$D = 2d(h_1/2) - d(h_1)$$

▼ Euler Method

The Euler method is perhaps the simplest numerical method for the solution of initial value problems of first-order ordina differential equations. Assume that we are looking for approximations to the solution $y(t)$ of the initial-value problem

$$\frac{d}{dt}y(t) = f(t, y(t)), \quad a \le t \le b, \quad y(a) = y_0$$

The function $y(t)$ is a differentiable function, obviously, since the differential equation involves its first derivative.

Given a uniform stepsize

$$h = \frac{b-a}{N} = t_{i+1} - t_i$$

we consider the uniform grid

$$a = t_0 < t_1 < \cdots < t_N = b$$

Then Euler's method computes approximation $y_i \approx y(t_i)$ at the nodes t_i via the explicit recursive formula

$$y_{i+1} = y_i + hf(t_i, y_i)$$

with given y_0

```
def f(t, y):
    return y

def euler(t, f, y0):
    n = len(t)
    y = np.zeros(n)
    h = t[1]-t[0]
    y[0]=y0
    for i in range(n-1):
        y[i+1] = y[i]+h*f(t[i], y[i])
    return y

a = 0.0; b = 2.0; N = 20
t = np.linspace(a, b, N+1)
y0 = 1.0
y = euler(t,f,y0)
# Plot the results
fig = plt.figure()
axes = fig.add_subplot(1, 1, 1)
axes.plot(t, y, '-o', label="Euler")
axes.plot(t, np.exp(t), '-d', label="Exact solution")
```

```
axes.set_xlabel("t")
axes.set_ylabel("y")
axes.legend(loc=2)
plt.show()
```

Improved Euler:

```
def f(t, y):
    return y

def improved_euler(t, f, y0):
    n = len(t)
    y = np.zeros(n)
    h = (t[-1]-t[0])/(n-1)
    y[0]=y0
    for i in range(n-1):
        k1 = f(t[i], y[i])
        k2 = f(t[i+1], y[i] + h*k1)
        y[i+1] = y[i] + 0.5*h*(k1+k2)
    return y

a = 0.0; b = 2.0; N = 10
t = np.linspace(a, b, N+1)
y0 = 1.0
y = improved_euler(t, f, y0)
# Plot the results
fig = plt.figure()
axes = fig.add_subplot(1, 1, 1)
axes.plot(t, y, '-o', label="Improved Euler")
axes.plot(t, np.exp(t), '-d', label="Exact solution")
axes.set_xlabel("t")
axes.set_ylabel("y")
axes.legend(loc=2)
plt.show()
```

▼ Scipy

```
from scipy.integrate import solve_ivp

def f(t, y):
    return y

a = 0.0; b = 2.0
tspan = [a, b]
y0 = [1.0]

sol = solve_ivp(f, tspan, y0, 'RK45', rtol=1.e-6)

# Plot the results
fig = plt.figure()
axes = fig.add_subplot(1, 1, 1)
axes.plot(sol.t, sol.y[0], '-o', label="RK45")
axes.plot(sol.t, np.exp(sol.t), '-d', label="Exact solutio
n")
axes.set_xlabel("t")
axes.set_ylabel("y")
axes.legend(loc=2)
plt.show()
```