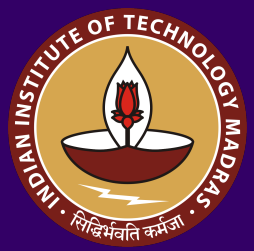
Scientific Computing & Advanced Programming Lab

SAURAV SAMANTARAY

Department of Mathematics
IIT Madras

Lecture-Slides-4



sauravsray@iitm.ac.in

Inheritance: Reusing a Class

- Suppose we already have a `Point` class:
- Now suppose we want a new type of point with additional functionality:

```
class ColoredPoint:
    ...
```

We would like `ColoredPoint` to:

- have x and y coordinates, have all the functionality of `Point`,
- additionally store a color.
- **Inheritance**: Create a new class based on an existing class, automatically obtaining its attributes and methods.

`Point` $\longrightarrow$ `ColoredPoint`

Creating a Derived Class

- A new class can inherit from an existing class:

```
class ColoredPoint(Point):
    def __init__(self, x, y, color):
        super().__init__(x, y)
        self.color = color
```

- Now a `ColoredPoint` has both the original and new functionality:

```
p = ColoredPoint(3, 4, "red")

print(p.x)
print(p.y)
print(p.color)
print(p.distance_from_origin())
```

Creating a Derived Class

- `ColoredPoint` automatically inherits the methods of `Point`.

  `ColoredPoint` is a `Point`

- `super()` gives access to the parent class.

- `super().__init__(x, y)`

  calls the `__init__()` method of `Point` and initializes

```
self.x = x
self.y = y
```

- The derived class then adds its own attribute:

```
self.color = color
```

$$\texttt{ColoredPoint} = \underbrace{\texttt{Point}}_{\text{inherited}} + \underbrace{\texttt{color}}_{\text{new}}$$

Inheritance: Vehicle Example

- Suppose we define a general `Vehicle` class:

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand
    def start(self):
        print("Vehicle started")
```

- A `Car` is a specialized type of `Vehicle`:

```
class Car(Vehicle):
    def __init__(self, brand, number_of_doors):
        super().__init__(brand)
        self.number_of_doors = number_of_doors
```

`Vehicle` $\longrightarrow$ `Car`

`Car` inherits the attributes and methods of `Vehicle`.

Inheritance: Using the Derived Class

- Create an object of the derived class:

```
car = Car("Toyota", 4)
```

The object has access to the inherited attributes and methods:

```
print(car.brand)
car.start()
```

and also its own attributes:

```
print(car.number_of_doors)
```

We can add functionality specific to `Car`:

```
def honk(self):
    print("Beep!")

car.honk()
```

`Car` = `Vehicle` + additional functionality

Why Inheritance?

- Suppose every employee has a name and salary:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def display(self):
        print(self.name, self.salary)
```

- But different types of employees may have additional responsibilities.

```
class Manager(Employee):
    def manage(self):
        print("Managing team")
```

Why Inheritance?

```
class Developer(Employee):
def program(self):
    print("Writing code")
```

- ```
  m = Manager("Alice", 80000)
  d = Developer("Bob", 60000)

  m.display()          # inherited
  m.manage()           # Manager-specific

  d.display()          # inherited
  d.program()          # Developer-specific
  ```

Common functionality is written once and reused.
  ```

Inheritance: Numerical Integration

- Consider the problem of approximating

$$I = \int_a^b f(x)\,dx.$$

- We can define a general class for a numerical integration method:

```
class Integrator:
    def __init__(self, f, a, b, n):
        self.f = f
        self.a = a
        self.b = b
        self.n = n
```

Inheritance: Numerical Integration

- Different numerical methods use different formulas.

Numerical Integration

↓

Trapezoidal Rule
Simpson's Rule

- Instead of creating completely unrelated classes, we can build the numerical methods from a common `Integrator` class.

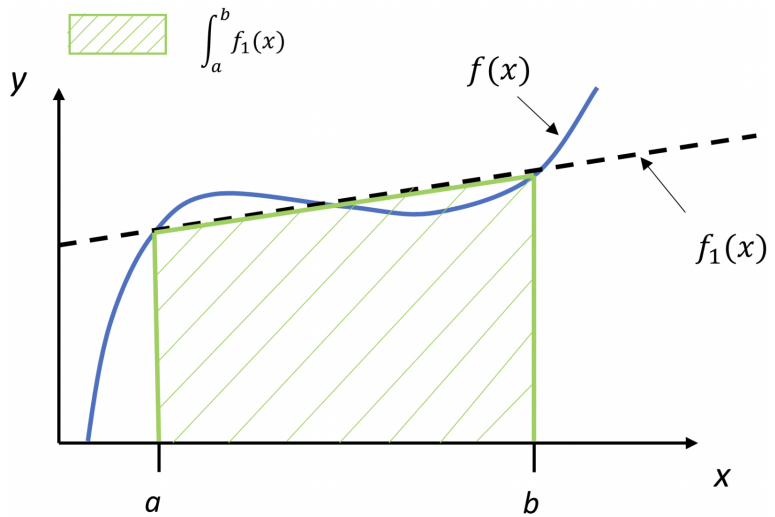
Trapezoidal Rule

- Consider

$$I = \int_a^b f(x)\, dx.$$

- Approximate $f(x)$ by the straight line joining

$$(a, f(a)) \qquad \text{and} \qquad (b, f(b)).$$



Trapezoidal Rule

- The area under this line is a trapezoid:

$$\boxed{I \approx \frac{b-a}{2}\left[f(a)+f(b)\right]}$$

- **Error:**

$$I - I_T = -\frac{(b-a)^3}{12} f''(\xi), \qquad \xi \in (a,b).$$

$$\boxed{I - I_T = O\left((b-a)^3\right)}$$

for a single interval.

Idea:

Function $\longrightarrow$ Linear approximation $\longrightarrow$ Trapezoid

Composite Trapezoidal Rule

- Instead of approximating $f(x)$ over the entire interval $[a, b]$, divide it into n subintervals:

$$h = \frac{b-a}{n}.$$

- Apply the trapezoidal rule on each subinterval.

$$\boxed{I \approx h\left[\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(a+ih)\right]}$$

- The global error is

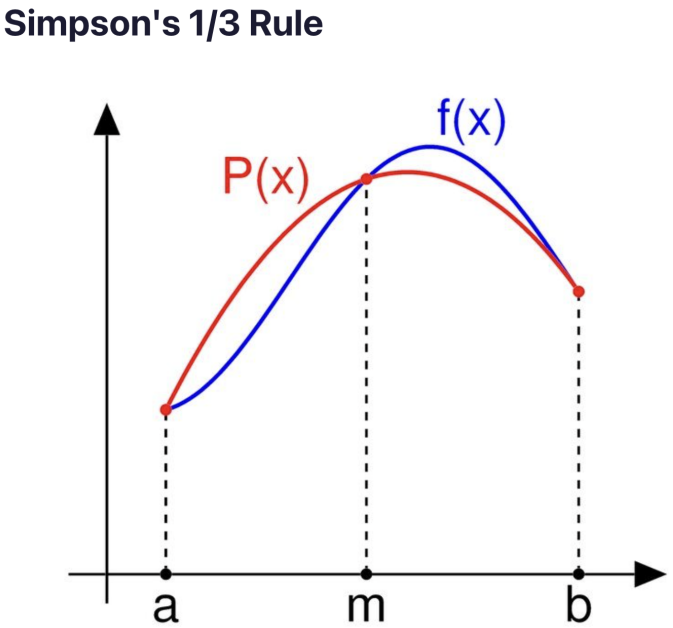
$$I - I_T = -\frac{b-a}{12}h^2 f''(\xi), \qquad \xi \in (a, b).$$

$$\boxed{I - I_T = O(h^2)}$$

Many small trapezoids $\longrightarrow$ better approximation

Simpson's Rule

- Simpson's rule: quadratic polynomial on each subinterval of a partition to approximate the function;
- improvement over the trapezoid rule(approximates by a straight line)



Simpson's Rule

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a)+4f(\frac{a+b}{2})+f(b)\right)$$

- **Error:**