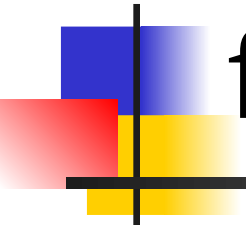
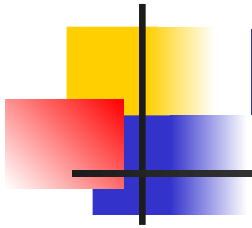


Interpolation and Root finding

Dr. Bernard Chen

Troy University



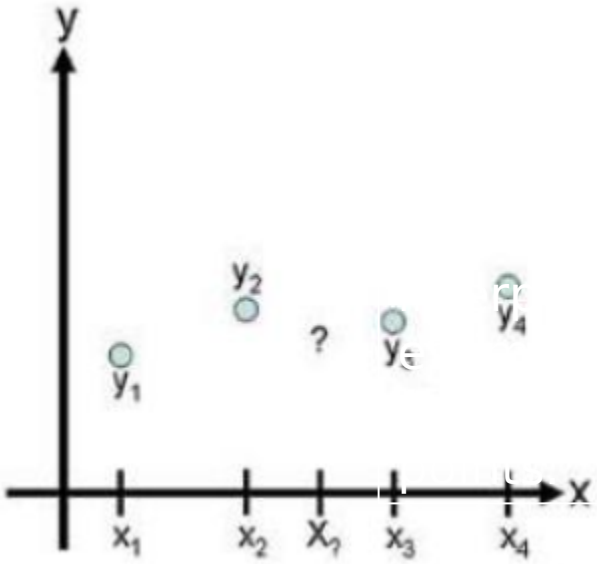
Interpolation

- A dataset consisting of independent data values, x_i and dependent data values, y_i where i = 1, 2, …, n

- Goal: find an estimation function $\hat{y}$ such that $\hat{y}(x_i) = y_i$ for every point in our dataset.

Interpolation

- Given a new x^*, we can interpolate its function value using $\hat{y}(x^*)$ which is called interpolation function.

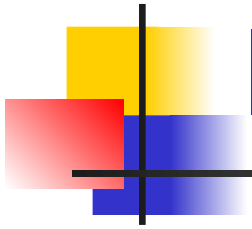


Interpolation

- In linear interpolation, the estimated point is assumed to lie on the line joining the nearest points to the left and right.

- Given x-data points in ascending order ($x_i < x_{i+1}$) and x being a point such that $x_i < x < x_{i+1}$, the linear interpolation at x is

$$\hat{y}(x) = y_i + \frac{(y_{i+1} - y_i)(x - x_i)}{(x_{i+1} - x_i)}$$



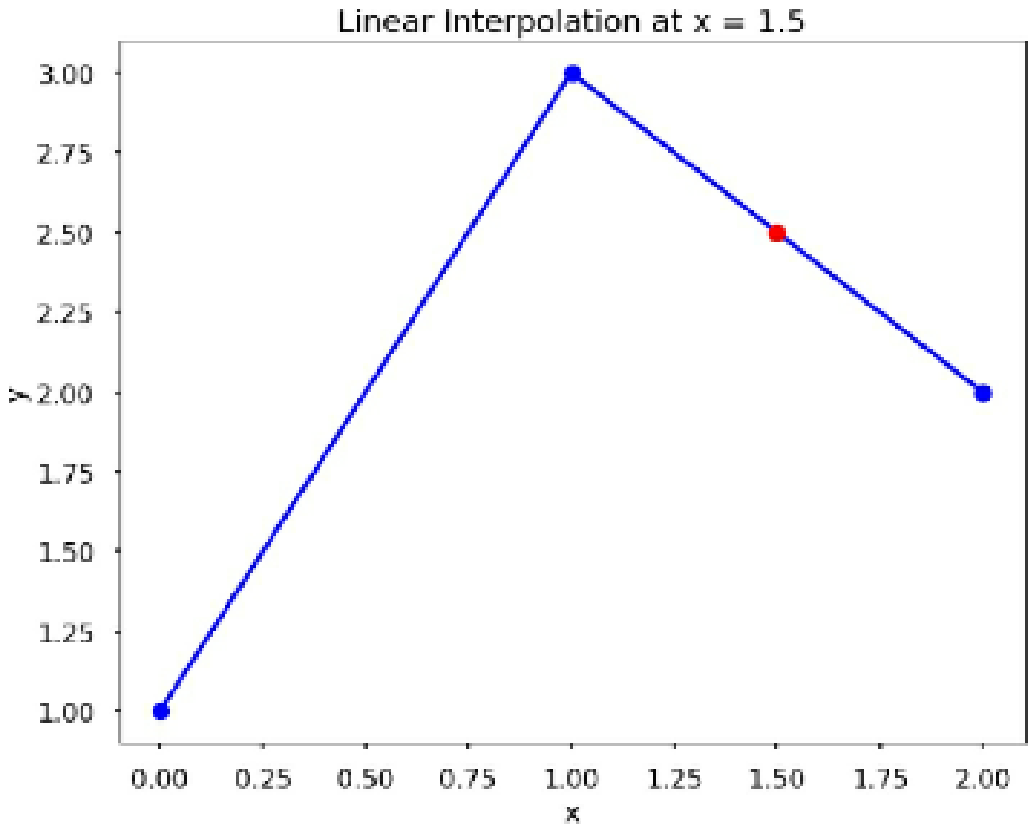
Interpolation

TRY IT! Find the linear interpolation at $x = 1.5$ based on the data `x = [0, 1, 2]`, `y = [1, 3, 2]`. Verify the result using `SciPy`'s function `interp1d`.

Since $1 < x < 2$, we use the second and third data points to compute the linear interpolation. Plugging in the corresponding values gives

$$\hat{y}(x) = y_i + \frac{(y_{i+1} - y_i)(x - x_i)}{(x_{i+1} - x_i)} = 3 + \frac{(2-3)(1.5-1)}{(2-1)} = 2.5.$$

Interpolation



Interpolation

- Python code:

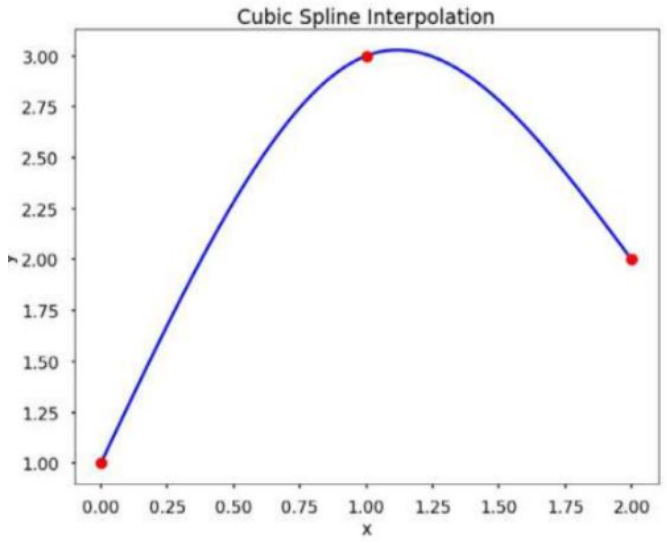
```
In [1]: from scipy.interpolate import interp1d
        import matplotlib.pyplot as plt

        plt.style.use("seaborn-poster")

In [2]: x = [0, 1, 2]
        y = [1, 3, 2]

        f = interp1d(x, y)
        y_hat = f(1.5)
        print(y_hat)

2.5
```

Cubic Spline Interpolation in Python

- Use SciPy's function, CubicSpline.

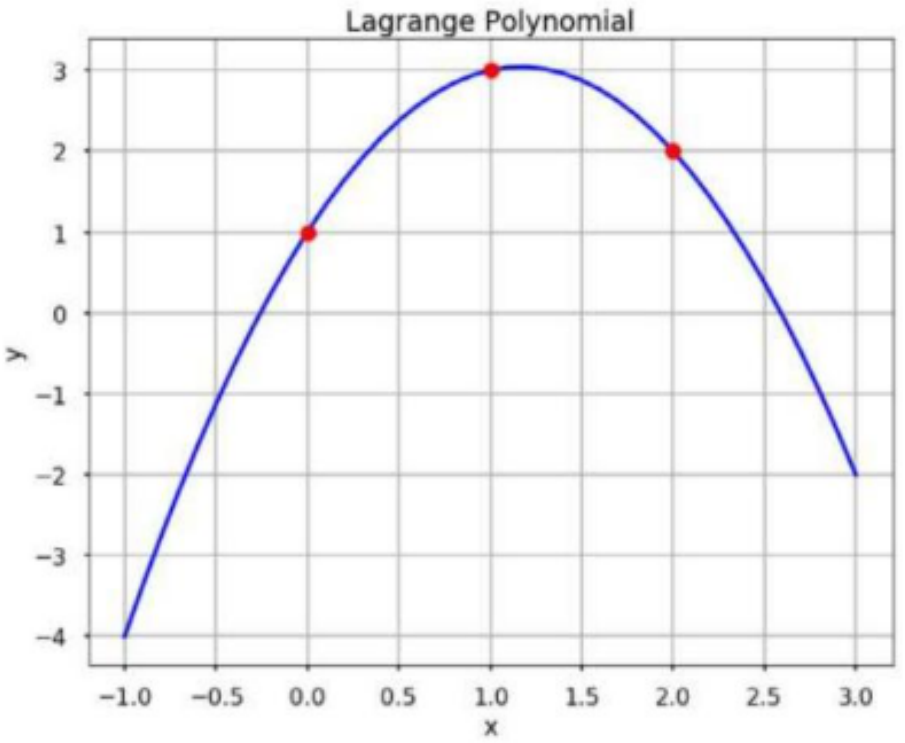
TRY IT! Use `CubicSpline` to plot the cubic spline interpolation of the dataset `x = [0, 1, 2]` and `y = [1, 3, 2]` for $0 \leq x \leq 2$.

```
In [1]: from scipy.interpolate import CubicSpline
        import numpy as np
        import matplotlib.pyplot as plt

        plt.style.use("seaborn-poster")

In [2]: x = [0, 1, 2]
        y = [1, 3, 2]

        # use bc_type = "natural" adds the constraints
        f = CubicSpline(x, y, bc_type="natural")
        x_new = np.linspace(0, 2, 100)
        y_new = f(x_new)

In [3]: plt.figure(figsize = (10,8))
        plt.plot(x_new, y_new, "b")
        plt.plot(x, y, "ro")
        plt.title("Cubic Spline Interpolation")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.show()
```



Lagrange Polynomial Interpolation in Python using Lagrange Function from Scipy

```
In [4]: from scipy.interpolate import lagrange

In [5]: f = lagrange(x, y)

In [6]: fig = plt.figure(figsize = (1
        plt.plot(x_new, f(x_new), "b"
        plt.title("Lagrange Polynomia
        plt.grid()
        plt.xlabel("x")
        plt.ylabel("y")
        plt.show()
```



Newton's Polynomial Interpolation in Python

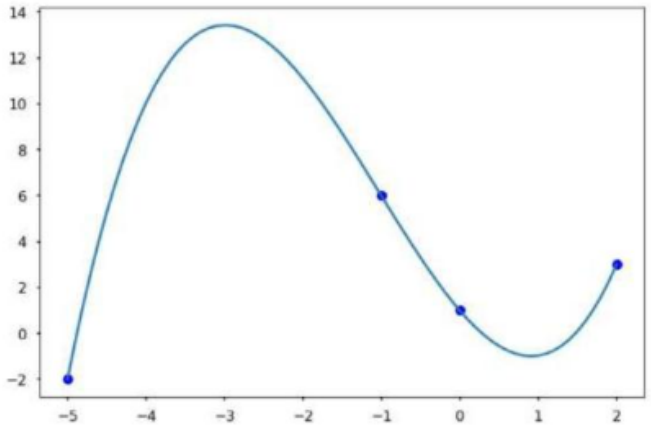
TRY IT! Calculate the divided difference table for x = [-5, -1, 0, 2], y = [-2, 6, 1, 3].

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

        plt.style.use("seaborn-poster")

        %matplotlib inline
```

```
In [2]: def divided_diff(x, y):
            """
            function to calculate the divided
            difference table
            """
            n = len(y)
            coef = np.zeros([n, n])
            # the first column is y
            coef[:,0] = y

            for j in range(1,n):
                for i in range(n-j):
                    coef[i][j] = (coef[i+1][j-1]-coef[i][j-1])/(x[i+j]-x[i])

            return coef

        def newton_poly(coef, x_data, x):
            """
            evaluate the Newton polynomial
            at x
            """
            n = len(x_data) - 1
            p = coef[n]
            for k in range(1,n+1):
                p = coef[n-k] + (x -x_data[n-k])*p
            return p
```

```
In [3]: x = np.array([-5, -1, 0, 2])
        y = np.array([-2, 6, 1, 3])
        # get the divided difference coef
        a_s = divided_diff(x, y)[0, :]

        # evaluate on new data points
        x_new = np.arange(-5, 2.1, .1)
        y_new = newton_poly(a_s, x, x_new)

        plt.figure(figsize = (12, 8))
        plt.plot(x, y, "bo")
        plt.plot(x_new, y_new)
```

```
Out[3]: [<matplotlib.lines.Line2D at 0x11bd4e630>]
```

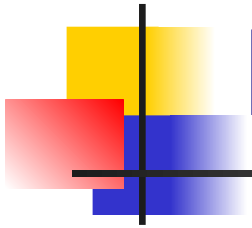


Root Finding

- The root or zero of a function, f(x): x_r such that $f(x_r) = 0$.

  - For some functions, it is clear finding the root of a function.
    $f(x) = x^2 - 9$ → the roots are 3 and -3

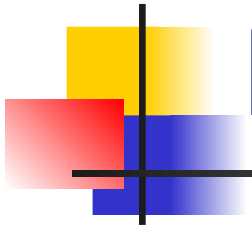
  - For some functions, determining an analytic or exact solution for the roots of functions can be difficult. $f(x) = \cos(x) - x$



Root Finding

- **Error**: a deviation from an expected or computed value.
- **Tolerance**: the level of error that is acceptable for an engineering application.

- "Computer program has **converged** to a solution when it has found a solution with an error smaller than the tolerance."
- For computing roots, we want an x_r such that $f(x_r)$ is very close to zero.



Root Finding

- Newton’s Method starts with a guess and repeatedly improves it.

- The update formula is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Root Finding

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Where:

- x_n = current guess
- x_{n+1} = better guess
- $f(x)$ = function
- $f'(x)$ = derivative

Root Finding

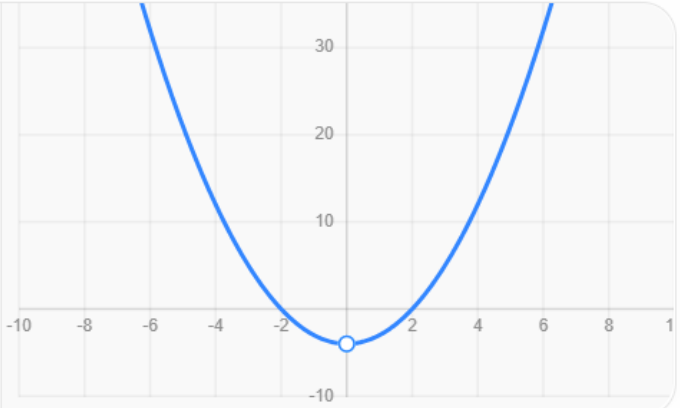
Example

Solve:

$$x^2 - 4 = 0$$

Function:

$$y = x^2 - 4$$



Derivative:

$$f'(x) = 2x$$

Root Finding

Step 1 — Initial Guess

Choose:

$$x_0 = 3$$

Step 2 — Apply Formula

Newton formula:

$$x_{n+1} = x_n - \frac{x_n^2 - 4}{2x_n}$$

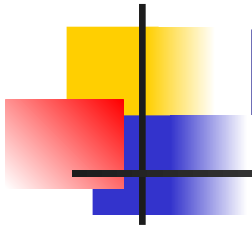
Root Finding

Iteration 1

$$x_1 = 3 - \frac{3^2 - 4}{2(3)}$$

$$= 3 - \frac{5}{6}$$

$$= 2.1667$$

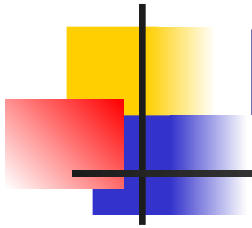


Root Finding

Iteration 2

$$x_2 = 2.1667 - \frac{2.1667^2 - 4}{2(2.1667)}$$

$$\approx 2.0064$$



Root Finding

Iteration 3

$$x_3 \approx 2.00001$$

Very close to the exact root:

$$x = 2$$

Root Finding

Python Example

Python

```python
def f(x):
    return x**2 - 4

def df(x):
    return 2*x

x = 3

for i in range(5):
    x = x - f(x)/df(x)
    print(x)
```

Output:

Python

```
2.1666666666666665
2.0064102564102564
2.0000102400262145
2.000000000026214
2.0
```