# Interpolation and Curve Fitting

Mark A. Austin

University of Maryland

*austin@umd.edu*

*ENCE 201, Fall Semester 2023*

July 17, 2023

# Overview

# Motivating Ideas

# Motivating Ideas

## Curve Fitting

Curve fitting is the process of constructing a curve (or mathematical function) that has a best fit to a series of data points.

**Benefits of Curve Fitting:**

- Provides a means to observe and quantify general trends.
- Removes noise from a function.
- Can extract meaningful parameters when measured data is fitted to an analytical equation.
- Can derive finite difference approximations.

# Motivating Ideas

## Categories of Curve Fitting

### Exact Curve Fitting (Interpolation)

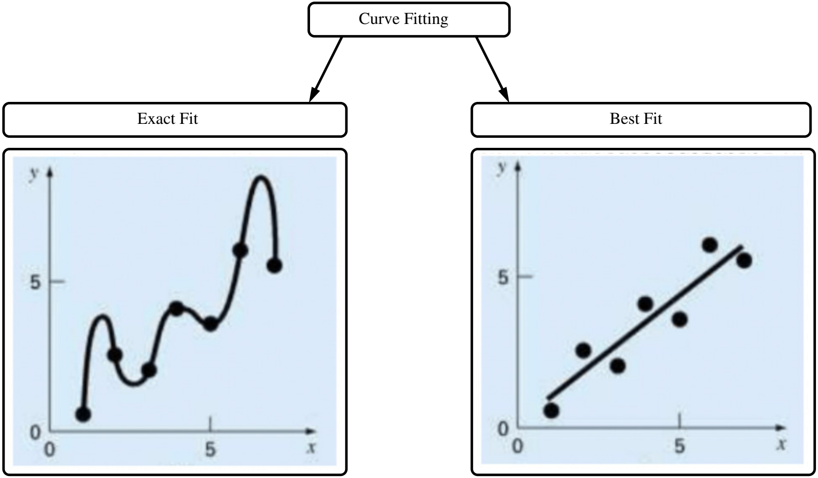
Exact fit (interpolation) occurs when we want to learn a curve that passes through the data points exactly.

### Best Curve Fitting (Least Squares Analysis)

Best fit curves make sense when we know that the data contains noise – rather than fit the data exactly, we aim to learn a function that minimizes some predefined error function on the data points.

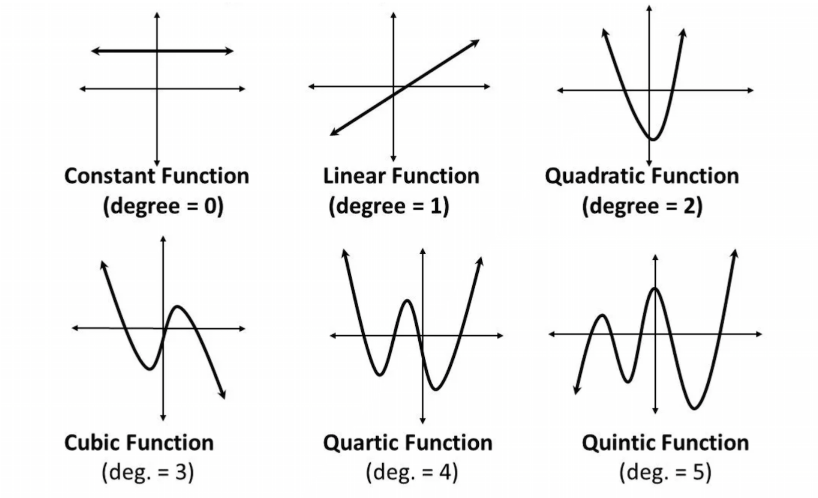
# Motivating Ideas

## Categories of Curve Fitting



# Motivating Ideas

**Graphs of Polynomial Functions:**



# Motivating Ideas

## Real-World Data

Data relating to (or collected from) a real-world application.

**Real-World Data:**

- Data collected from mobile systems (e.g., smart watches, automobiles, Google Street View).

**Opportunities and Challenges:**

- Provides real-world evidence needed for the design and operation of modern systems.
- Pathway to system-specific decision making procedures.
- Real-world data can be noisy.
- Easy to collect too much data.

# Motivating Ideas

**Model Fidelity Assessment:**

## Underfitting

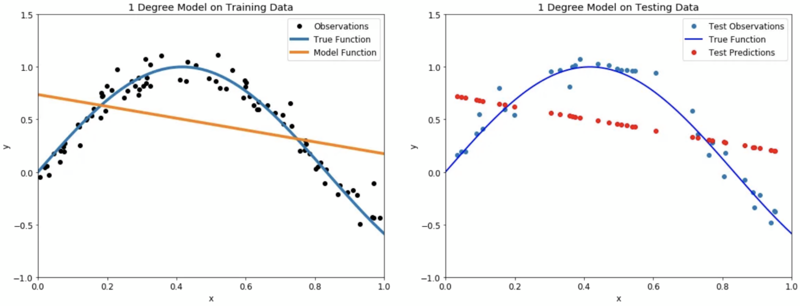
A condition where a curve fitting model is incapable of capturing the general trend in the data; this, in turn, affects the accuracy of a model.

## Overfitting

A condition where a curve fitting model begins to describe the random error (fluctuations) in the data rather than the underlying relationships among variables.

# Motivating Ideas

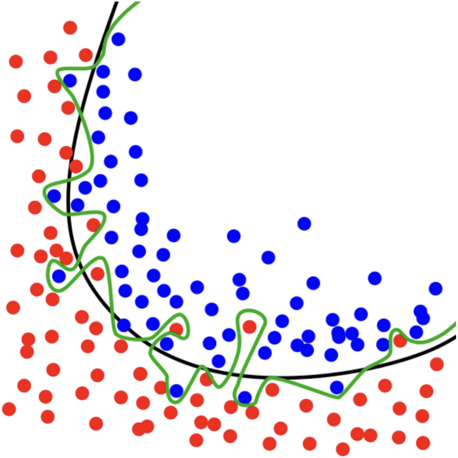
**Example. Underfitting in Data Science:** We wish to predict the relationship between input $x$ and output $y$.




Clearly, the linear function approximation underfits the true relationship between x and y.

# Motivating Ideas

**Example. Overfitting in Data Science:** We wish to find a low-order function that separates the red and blue data points.



# Method of

# Divided Differences

# Method of Divided Differences

## Divided Differences

Given a set of distinct points $x_0, x_1, \cdots, x_n$, and known function values $f_0 = f(x_0)$, $f_1 = f(x_1)$, ..., $f_n = f(x_n)$, the method of divided differences is a numerical procedure for interpolating the data with a polynomial fit.

**Polynomial Fit.** Let:

$$f(x) = a_o + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n(x - x_0)\cdots(x - x_{(n-1)}). \tag{1}$$

The method of divided differences provides systematic way to determine the polynomial coefficients $a_0$ through $a_{(n-1)}$.

# Method of Divided Differences

**Divided Difference Table and Formulae.** For $f(x)$ based on $x_0, x_1, x_2, x_3$.

| $x_i$ | $f[x_i] = f(x_i)$ | $f[,]$ | $f[,,]$ | $f[,,,]$ |
|---|---|---|---|---|
| $x_0$ | $f(x_0)$ | $f[x_0, x_1]$ | $f[x_0, x_1, x_2]$ | $f[x_0, x_1, x_2, x_3]$ |
| $x_1$ | $f(x_1)$ | $f[x_1, x_2]$ | $f[x_1, x_2, x_3]$ | |
| $x_2$ | $f(x_2)$ | $f[x_2, x_3]$ | | |
| $x_3$ | $f(x_3)$ | | | |

Here,

$$f[x_i] = f(x_i)$$
$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i}. \tag{2}$$

# Method of Divided Differences

Generally,

$$f[x_i, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}. \qquad (3)$$

The interpolated polynomial is:

$$\begin{aligned} f(x) =& f[x_0] + f[x_o, x_1](x - x_o) + f[x_0, x_1, x_2](x - x_o)(x - x_1) + \\ & f[x_0, x_1, x_2, x_3](x - x_o)(x - x_1)(x - x_2). \end{aligned}$$

**Note:** The algorithm depends only on the values $x_i, \ldots, x_{i+k}$, and not their order.

# Method of Divided Differences

**Example 1.** Find a polynomial that interpolates the data set:

```
   x   |     0     1     3
------*----------------
 f(x)  |     1     0    10
```

**Solution:** We seek:

| $x_i$ | $f[x_i] = f(x_i)$ | $f[,]$ | $f[,,]$ |
|---|---|---|---|
| 0 | 1 | $f[x_0, x_1]$ | $f[x_0, x_1, x_2]$ |
| 1 | 0 | $f[x_1, x_2]$ | |
| 3 | 10 | | |

where

$$f[x_0, x_1] = \left[\frac{f(x_1) - f(x_0)}{x_1 - x_0}\right] = \left[\frac{0-1}{1-0}\right] = -1. \tag{4}$$

# Method of Divided Differences

$$f[x_1, x_2] = \left[\frac{f(x_2) - f(x_1)}{x_2 - x_1}\right] = \left[\frac{10 - 0}{3 - 1}\right] = 5. \tag{5}$$

$$f[x_0, x_1, x_2] = \left[\frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}\right] = \left[\frac{5 + 1}{3 - 0}\right] = 2. \tag{6}$$

**Interpolated Polynomial:**

$$\begin{aligned} f(x) &= f[x_0] + f[x_o, x_1](x - x_o) + f[x_0, x_1, x_2](x - x_o)(x - x_1) \\ &= 1 + -1(x - 0) + 2(x - 0)(x - 1) \\ &= 1 - 3x + 2x^2 \end{aligned}$$

**Validate:** f(0) = 1; f(1) = 0; f(3) = 1 - 9 + 18 = 10.

# Method of Divided Differences

## **Python Code:** Divided difference test program

```
# ===================================================================================
# TestInterpolationDividedDifferences01.py: Compute divided differences polynomial.
#
# Written By: Mark Austin                                                   July 2023
# ===================================================================================

import math;
import numpy as np
import matplotlib.pyplot as plt

import Interpolation;

# main method ...

def main():
    print("--- Case Study 1: Small test problem ... ");

    x = np.array( [ 0, 1,  3 ] )
    y = np.array( [ 1, 0, 10 ] )

    print("---   Compute divided difference table ... ");

    dTable= Interpolation.divideddifference(x, y)[0, :]

    print("---   Evaluate on new data points ... ");

    x_new = np.arange( -1.0, 4.0, .2 )
    y_new = Interpolation.newtonpolynomial(dTable, x, x_new)
```

# Method of Divided Differences

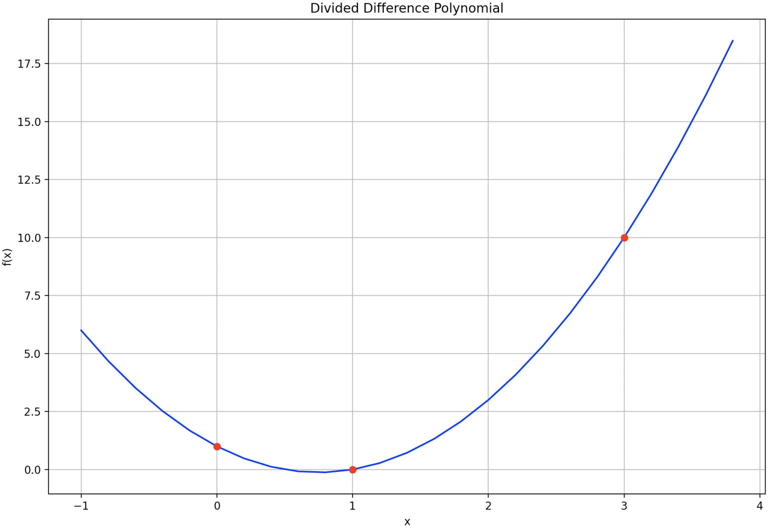
**Python Code:** Divided difference test program

```

    print("---   Plot divided difference polynomial ... ");

    plt.figure(figsize = (12, 8))
    plt.plot(x_new, y_new, 'b', x, y, 'ro')
    plt.title('Divided Difference Polynomial')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.grid()
    plt.show()

# call the main method ...

main()
```

**Abbreviated Output:**

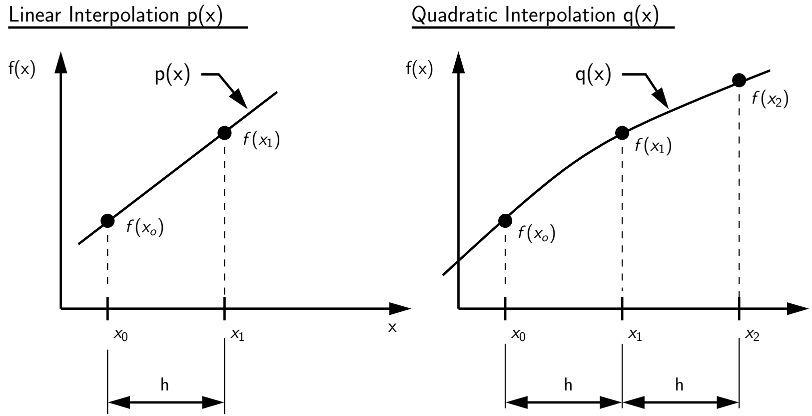
```
Matrix: divided difference table

  1.0000  -1.0000   2.0000
  0.0000   5.0000   0.0000
 10.0000   0.0000   0.0000
```

# Method of Divided Differences



# Method of Divided Differences

**Example 2.** Derive formulae for linear and quadratic interpolation when data points are equally spaced.



# Method of Divided Differences

**Linear Interpolation p(x).** Divided difference table is:

| $x_i$ | $f[x_i] = f(x_i)$ | $f[,]$ |
|---|---|---|
| $x_0$ | $f(x_0)$ | $f[x_0, x_1]$ |
| $x_1$ | $f(x_1)$ | |

where

$$f[x_0, x_1] = \left[\frac{f(x_1) - f(x_0)}{x_1 - x_0}\right] = \left[\frac{f(x_1) - f(x_0)}{h}\right]. \tag{7}$$

**Interpolated Polynomial:**

$$p(x) = f[x_0] + f[x_o, x_1](x - x_o) = f(x_o) + \left[\frac{f(x_1) - f(x_0)}{h}\right](x - x_o). \tag{8}$$

# Method of Divided Differences

**Quadratic Interpolation q(x).** Divided difference table is:

| $x_i$ | $f[x_i] = f(x_i)$ | $f[,]$ | $f[,,]$ |
|---|---|---|---|
| $x_0$ | $f(x_0)$ | $f[x_0, x_1]$ | $f[x_0, x_1, x_2]$ |
| $x_1$ | $f(x_1)$ | $f[x_1, x_2]$ | |
| $x_2$ | $f(x_2)$ | | |

where

$$f[x_0, x_1] = \left[\frac{f(x_1) - f(x_0)}{x_1 - x_0}\right] = \left[\frac{f(x_1) - f(x_0)}{h}\right]. \tag{9}$$

$$f[x_1, x_2] = \left[\frac{f(x_2) - f(x_1)}{x_2 - x_1}\right] = \left[\frac{f(x_2) - f(x_1)}{h}\right]. \tag{10}$$

# Method of Divided Differences

$$f[x_0, x_1, x_2] = \left[\frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}\right] = \left[\frac{f(x_2) - 2f(x_1) + f(x_o)}{2h^2}\right]. \quad (11)$$

**Interpolated Polynomial:**

$$q(x) = f[x_0] + f[x_o, x_1](x - x_o) + f[x_o, x_1, x_2](x - x_o)(x - x_1). \quad (12)$$

**Validate:** $q(x_o) = f(x_o)$, $q(x_1) = f(x_1)$, $q(x_2) = f(x_2)$.

# Method of Divided Differences

**Integration of p(x):**

$$\int_{x_o}^{x_1} p(x)dx = \int_{x_o}^{x_1} f(x_o)dx + \left[\frac{f(x_1) - f(x_0)}{h}\right] \int_{x_o}^{x_1} (x - x_o)dx$$
$$= \frac{h}{2}\left[f(x_o) + f(x_1)\right].$$

**Integration of q(x):**

$$\int_{x_o}^{x_1} q(x)dx = \frac{h}{3}\left[f(x_o) + 4f(x_1) + f(x_2)\right].$$

# Method of Divided Differences

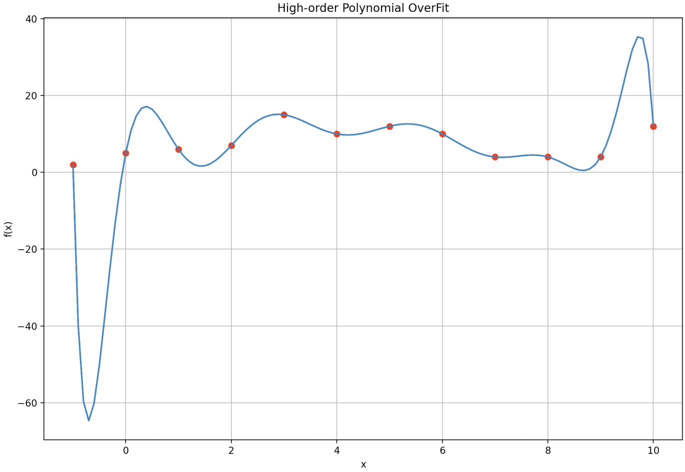
**Example 3.** High-order polynomial overfit for the dataset:

```
   x  |   -1   0   1   2   3   4   5   6   7   8   9  10
------*---------------------------------------------------
 f(x) |    2   5   6   7  15  10  12  10   4   4   4  12
```

**Abbreviated Output:**

```
Matrix: divided difference table
  2  3 -1.0  0.33  0.21 -0.27  0.14 -0.05  0.01 -0.002  0.000 -0.0001
  5  1  0.0  1.17 -1.13  0.56 -0.19  0.05 -0.01  0.002 -0.000  0.0000
  6  1  3.5 -3.33  1.67 -0.59  0.16 -0.03  0.00 -0.000  0.000  0.0000
  7  8 -6.5  3.33 -1.29  0.35 -0.06  0.00  0.00  0.000  0.000  0.0000
 15 -5  3.5 -1.83  0.46 -0.01 -0.04  0.02  0.00  0.000  0.000  0.0000
 10  2 -2.0  0.00  0.42 -0.22  0.08  0.00  0.00  0.000  0.000  0.0000
 12 -2 -2.0  1.67 -0.67  0.25  0.00  0.00  0.00  0.000  0.000  0.0000
 10 -6  3.0 -1.00  0.58  0.00  0.00  0.00  0.00  0.000  0.000  0.0000
  4  0  0.0  1.33  0.00  0.00  0.00  0.00  0.00  0.000  0.000  0.0000
  4  0  4.0  0.00  0.00  0.00  0.00  0.00  0.00  0.000  0.000  0.0000
  4  8  0.0  0.00  0.00  0.00  0.00  0.00  0.00  0.000  0.000  0.0000
 12  0  0.0  0.00  0.00  0.00  0.00  0.00  0.00  0.000  0.000  0.0000
```

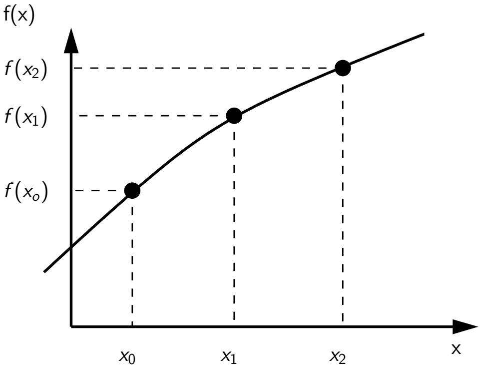
# Method of Divided Differences



# Lagrange Interpolation

# Lagrange Interpolation

**Basic Idea.** Assume that a curve passes through set of point: $(x_o, f(x_o))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$, $\cdots$, $(x_n, f(x_n))$.




We propose that:

# Lagrange Interpolation

$$
\begin{aligned}
f_o &= f_o p_o(x_o) + f_1 p_1(x_o) + \cdots + f_n p_n(x_o).\\
f_1 &= f_o p_o(x_1) + f_1 p_1(x_1) + \cdots + f_n p_n(x_1).\\
\cdots &= \cdots\\
f_n &= f_o p_o(x_n) + f_1 p_1(x_1) + \cdots + f_n p_n(x_n).
\end{aligned}
$$

where $p_i(x_j) = 1, i = j$, and $p_i(x_j) = 0, i \neq j$.

**Lagrange Formula:** $f(x) = p_0(x) f_o + p_1(x) f_1 + p_2(x) f_2 + ...,$
where

$$
p_i(x) = \frac{\prod_{i=0, i\neq j}^{n-1}(x - x_i)}{\prod_{i=0, i\neq j}^{n-1}(x_j - x_i)} \tag{13}
$$

# Lagrange Interpolation

**Example 1.** Find a polynomial that interpolates the data set:

```
   x  |     0    1    3
------*----------------
 f(x) |     1    0   10
```

**Solution.** For the given dataset,

$$f(x) = f(x_o)p_o(x) + f(x_1)p_1(x) + f(x_2)p_2(x) \tag{14}$$

where

$$p_o(x) = \frac{(x-1)(x-3)}{(0-1)(0-3)} = \left[\frac{x^2 - 4x + 3}{3}\right]. \tag{15}$$

$$p_1(x) = \frac{(x-0)(x-3)}{(1-0)(1-3)} = \left[\frac{x^2 - 3x}{-2}\right]. \tag{16}$$

# Lagrange Interpolation

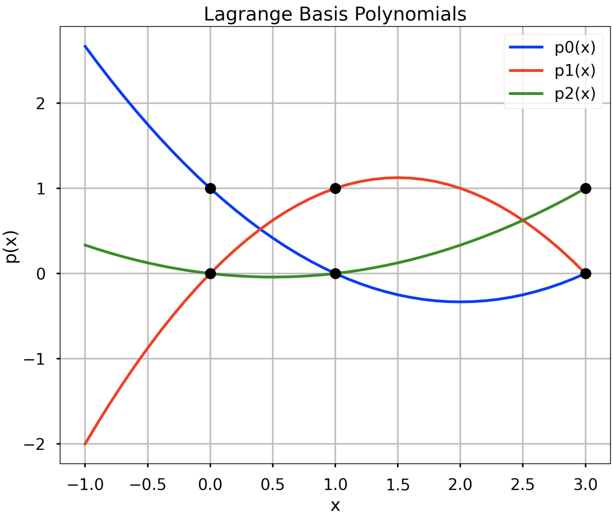
$$p_2(x) = \frac{(x-0)(x-1)}{(3-0)(3-1)} = \left[\frac{x^2 - x}{6}\right]. \qquad (17)$$

Plugging equations (15) - (17) into (14):

$$\begin{aligned} f(x) &= f(x_o)p_o(x) + f(x_1)p_1(x) + f(x_2)p_2(x) \\ &= 1 \cdot \left[\frac{x^2 - 4x + 3}{3}\right] + 0 \cdot \left[\frac{x^2 - 4x + 3}{2}\right] + 10 \cdot \left[\frac{x^2 - x}{6}\right], \\ &= 1 - 3x + 2x^2. \end{aligned}$$

**Validate:** f(0) = 1; f(1) = 0; f(3) = 1 - 9 + 18 = 10.

# Lagrange Interpolation



# Lagrange Interpolation

**Example 2.** For the set of data,

| x | -1 | 2 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| f(x) | 2 | 7 | 10 | 3 | 0 |

use the Lagrange interpolation formula to approximate the functional value at x = 3.5

**Solution.** For the given dataset,

$$f(x) = 2p_0(x) + 7p_1(x) + 10p_2(x) + 3p_3(x), \qquad (18)$$

where

$$p_o(x) = \frac{(x-2)(x-4)(x-5)(x-6)}{(-1-2)(-1-4)(-1-5)(-1-6)}$$
$$= (x-2)(x-4)(x-5)(x-6)/(630).$$

# Lagrange Interpolation

Similarly,

$$p_1(x) = (x+1)(x-4)(x-5)(x-6)/(-72). \tag{19}$$

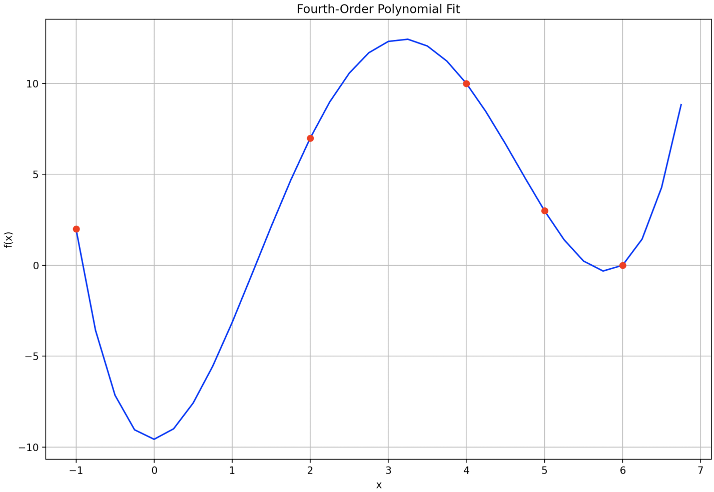
$$p_2(x) = (x+1)(x-2)(x-5)(x-6)/(20). \tag{20}$$

$$p_3(x) = (x+1)(x-2)(x-4)(x-6)/(-18). \tag{21}$$

**Let x = 3.5**. $p_0(3.5) = (3.5-2)(3.5-4)(3.5-5)(3.5-6)/630$ = -2.81/630. Similarly, $p_1(3.5)$ = -8.43/-72, $p_2(3.5) = 25.31/20$ and $p_3(3.5)$ = 8.43/-18. Hence,

$$f(3.5) = 2p_0(3.5)+7p_1(3.5)+10p_2(3.5)+3p_3(3.5) = 12.06. \tag{22}$$

# Lagrange Interpolation



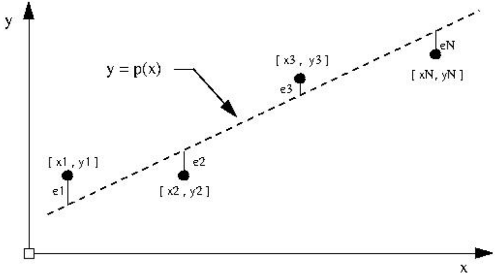
# Least Squares Analysis

# Least Squares Analysis

## Least Squares Analysis

Given a set of data points, least squares analysis is a numerical procedures for finding a best fit curve.

**Mathematical Approach.** Minimize the sum of the squares of the residuals between the data and data provided by the fitted model.



# Least Squares Analysis

**Least Squares Data:** $(x_0, y_0), (x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$.

**Mathematical Procedure:**

- Let $e_i = y_i - p(x_i)$.
- We aim to determine the curve fit parameters that minimize:

$$S = \sum_{i=0}^{n} e_i^2 = \sum_{i=0}^{n} [y_i - p(x_i)]^2. \tag{23}$$

- Mean square error (MSE) $= \left[\frac{S}{n}\right]$.

**Practical Considerations:**

- What sort of function, p(x), do we want to fit? Linear? Quadratic? Exponential? Sinusoidal? Which function provides the best fit?

# Least Squares Analysis

**Model 1.** Linear Approximation to the Data:

Let: $p(x) = a_o + a_1 x$. The sum of the squares:

$$S(a_o, a_1) = \sum_{i=0}^{n} e_i^2 = \sum_{i=0}^{n} [y_i - a_o - a_1 x]^2. \tag{24}$$

has a minumum value when:

$$\frac{\partial S}{\partial a_o} = 0. \longrightarrow \left(\sum_{i=0}^{n}\right) a_o + \left(\sum_{i=0}^{n} x_i\right) a_1 = \left(\sum_{i=0}^{n} y_i\right). \tag{25}$$

$$\frac{\partial S}{\partial a_1} = 0 \longrightarrow \left(\sum_{i=0}^{n} x_i\right) a_o + \left(\sum_{i=0}^{n} x_i^2\right) a_1 = \left(\sum_{i=0}^{n} x_i y_i\right). \tag{26}$$

# Least Squares Analysis

Writing equations 25 and 26 in matrix form:

$$\begin{bmatrix} n & \sum_{i=0}^{n} x_i \\ \sum_{i=0}^{n} x_i & \sum_{i=0}^{n} x_i^2 \end{bmatrix} \cdot \begin{bmatrix} a_o \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{n} y_i \\ \sum_{i=0}^{n} x_i y_i \end{bmatrix} \tag{27}$$

# Least Squares Analysis

**Model 2.** Quadratic Approximation to the Data:

Let: $p(x) = a_o + a_1 x + a_2 x^2$. The sum of the squares:

$$S(a_o, a_1, a_2) = \sum_{i=0}^{n} e_i^2 = \sum_{i=0}^{n} \left[ y_i - a_o - a_1 x - a_2 x^2 \right]^2. \tag{28}$$

has a minumum value when:

$$\frac{\partial S}{\partial a_o} = \frac{\partial S}{\partial a_1} = \frac{\partial S}{\partial a_2} = 0. \tag{29}$$

The individual equations are:

$$\left( \sum_{i=0}^{n} \right) a_o + \left( \sum_{i=0}^{n} x_i \right) a_1 + \left( \sum_{i=0}^{n} x_i^2 \right) a_2 = \left( \sum_{i=0}^{n} y_i \right). \tag{30}$$

# Least Squares Analysis

$$\left(\sum_{i=0}^{n} x_i\right) a_o + \left(\sum_{i=0}^{n} x_i^2\right) a_1 + \left(\sum_{i=0}^{n} x_i^3\right) a_2 = \left(\sum_{i=0}^{n} x_i y_i\right). \quad (31)$$

$$\left(\sum_{i=0}^{n} x_i^2\right) a_o + \left(\sum_{i=0}^{n} x_i^3\right) a_1 + \left(\sum_{i=0}^{n} x_i^4\right) a_2 = \left(\sum_{i=0}^{n} x_i^2 y_i\right). \quad (32)$$

Writing equations 30 and 32 in matrix form:

$$\begin{bmatrix} n & \sum_{i=0}^{n} x_i & \sum_{i=0}^{n} x_i^2 \\ \sum_{i=0}^{n} x_i & \sum_{i=0}^{n} x_i^2 & \sum_{i=0}^{n} x_i^3 \\ \sum_{i=0}^{n} x_i^2 & \sum_{i=0}^{n} x_i^3 & \sum_{i=0}^{n} x_i^4 \end{bmatrix} \cdot \begin{bmatrix} a_o \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{n} y_i \\ \sum_{i=0}^{n} x_i y_i \\ \sum_{i=0}^{n} x_i^2 y_i \end{bmatrix} \quad (33)$$

# Least Squares Analysis

**Example 1.** Find linear and quadratic least squares approximations to the set of data:

```
   x   |     0     3     6
-------*----------------
 f(x)  |     1     2     0
```

find linear and quadratic least squares approximations.

**Note:** Because there are only three data points, we expect that the quadratic model will interpolate the data exactly.

**Abbreviated Output:**

```
Matrix: data array
   0.0000000e+00     1.0000000e+00
   3.0000000e+00     2.0000000e+00
   6.0000000e+00     1.0000000e+00
```

# Least Squares Analysis

**Abbreviated Output:** Continued ...

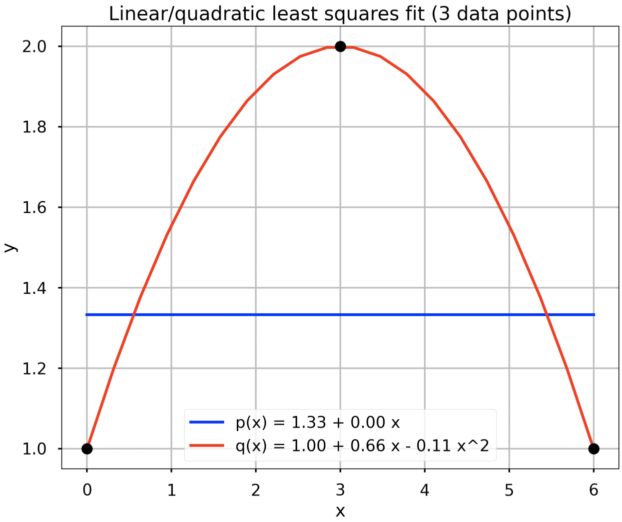
```
--- Part 1: Linear least squares fit ...
--- ------------------------------------

Matrix: A1                                                 Matrix: B1
   3.0000000e+00    9.0000000e+00                             4.0000000e+00
   9.0000000e+00    4.5000000e+01                             1.2000000e+01

--- Least squares coefficients and polynomial ...

Matrix: Coeff    -----> p(x) = 1.3333333333333333 + 0.0·x¹
   1.3333333e+00
   0.0000000e+00

--- Mean square error = 0.222 ...

--- Part 2: Quadratic least squares fit ...
--- ---------------------------------------

Matrix: A2
   3.0000000e+00    9.0000000e+00    4.5000000e+01
```

# Least Squares Analysis

**Abbreviated Output:** Continued ...

```
   9.0000000e+00    4.5000000e+01    2.4300000e+02
   4.5000000e+01    2.4300000e+02    1.3770000e+03

Matrix: B2
   4.0000000e+00
   1.2000000e+01
   5.4000000e+01

--- Least squares coefficients and polynomial ...

Matrix: Coeff
   1.0000000e+00  ---> q(x) = 1.000 + 0.667·x¹ - 0.111·x²
   6.6666667e-01
  -1.1111111e-01

--- Mean square error = 0.000 ...
```

# Least Squares Analysis



# Least Squares Analysis

**Example 2.** Find linear and quadratic least squares approximations to the set of data:

```
   x  |   0    1    2    3    4    5    6    7    8    9   10
------*-------------------------------------------------------
 f(x) |   0    1    2    3    4    5    6    6    6    6    6
```

**Abbreviated Output:**

```
Matrix: data array
   0.0000000e+00     0.0000000e+00
   1.0000000e+00     1.0000000e+00
   2.0000000e+00     2.0000000e+00
   ....              ....
   8.0000000e+00     6.0000000e+00
   9.0000000e+00     6.0000000e+00
   1.0000000e+01     6.0000000e+00
```

# Least Squares Analysis

**Abbreviated Output:** Continued ...

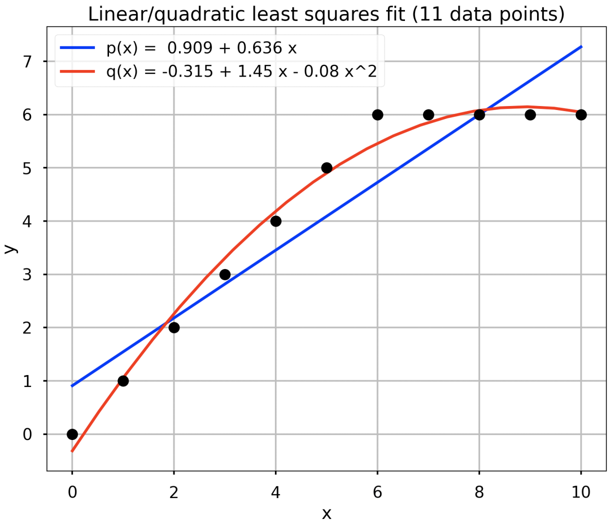
```
--- Part 1: Linear least squares fit ...
--- -----------------------------------

Matrix: A1                                        Matrix: B1
   1.1000000e+01    5.5000000e+01                    4.5000000e+01
   5.5000000e+01    3.8500000e+02                    2.9500000e+02

--- Least squares coefficients and polynomial ...

Matrix: Coeff     ---> p(x) = 0.909 + 0.636·x¹
   9.0909091e-01
   6.3636364e-01

--- Mean square error = 0.579 ...

--- Part 2: Quadratic least squares fit ...
--- ---------------------------------- ...

Matrix: A2
   1.1000000e+01    5.5000000e+01    3.8500000e+02
```

# Least Squares Analysis

**Abbreviated Output:** Continued ...

```
   5.5000000e+01    3.8500000e+02    3.0250000e+03
   3.8500000e+02    3.0250000e+03    2.5333000e+04

Matrix: B2
   4.5000000e+01
   2.9500000e+02
   2.2050000e+03

--- Least squares coefficients and polynomial ...

Matrix: Coeff      ---> q(x) = -0.314 + 1.452·x¹ - 0.081·x²
  -3.1468531e-01
   1.4522145e+00
  -8.1585082e-02

--- Mean square error = 0.059 ...  <-- Much better !!!
```

# Least Squares Analysis



# Python Code Listings

# Code 1: Method of Divided Differences

```
# ============================================================
# Interpolation.py: Functions to compute interpolation polynomial fits ...
# ============================================================

import math
import numpy as np

# =======================================================
# LinearMatrixEquations.printmatrix(): Print two-dimensional matrices.
#
# Args: name: string description of matrix.
#       A (nxn) matrix.
#
# Returns: void.
# =======================================================

def printmatrix(name, a):
    print("");
    print("Matrix: {:s} ".format(name) );
    for row in a:
        for col in row:
            print("{:8.4f}".format(col), end=" ")
        print("")

# ===============================================================
# Interpolation.divideddifference(): Compute Newton's divided difference table ..
#
# Args: x (float): array of x coordinates.
```

# Code 1: Method of Divided Differences

```
#       y (float): array of y values.
#
# Returns:
#       dtable (float): divided difference matrix/table.
# ==============================================================================

def divideddifference(x, y):

    # Create divided difference table ...

    n = len(y)
    dtable = np.zeros([n, n])

    # First column of table is y ...

    dtable[:,0] = y
    for j in range(1,n):
        for i in range(n-j):
            dtable[i][j] = (dtable[i+1][j-1] - dtable[i][j-1]) / (x[i+j]-x[i])

    # Print divided difference table ...

    printmatrix("divided difference table", dtable);

    return dtable

# ==============================================================================
# Interpolation.newtonpolynomial(): Evaluate Newton's polynomial at x.
#
# Args: dtable (float): divided difference matrix / table.
```

# Code 1: Method of Divided Differences

```
#        y (float): array of y values.
#        x_data (float): array of x_data points.
#
# Returns:
#        p (float): value of newtons polynomial evaluated at x.
# ===============================================================================

def newtonpolynomial( dtable, x_data, x):
    n = len(x_data) - 1
    p = dtable[n]
    for k in range(1,n+1):
        p = dtable[n-k] + (x -x_data[n-k])*p
    return p
```

# Code 2: Lagrange Interpolation

```
# =====================================================================
# Interpolation.py: Functions to compute interpolation polynomial fits ...
# =====================================================================

import math
import numpy as np

# =====================================================================
# Interpolation.lagrange(): Compute Lagrange polynomial through the
#                           points (x, y) and return its value at t.
#
# Args: x (float): array of x values ...
#       y (float): array of y values ...
#       t  (float): evaluate polynomial at point t.
#
# Returns:
#       t (float): polynomal value evaluated at point t.
# =====================================================================

def lagrange(x, y, t):

    # Check that the input arrays have the same length

    if len(x) != len(y):
        raise ValueError("The arrays x and y must have the same length.")

    # Initialize the polynomial
```

# Code 2: Lagrange Interpolation

```
    p = 0

    # Loop over the points

    for i in range(len(x)):

        # Get the current point
        xi, yi = x[i], y[i]

        # Initialize the term
        term = yi

        # Loop over the other points
        for j in range(len(x)):
            # Skip the current point
            if i == j:
               continue

            # Multiply the term by the appropriate factor
            term *= (t - x[j]) / (xi - x[j])

        # Add the term to the polynomial
        p += term

    return p
```

# Code 3: Lagrange Basis Polynomials

```
# ============================================================================
# TestInterpolationLagrange02.py: Work with Lagrange basis polynomials ...
#
# Written By: Mark Austin                                           July 2023
# ============================================================================

import math;

import numpy as np
import numpy.polynomial.polynomial as poly
import matplotlib.pyplot as plt

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

def main():
    print("--- Case Study 1. Interpolation.lagrange() ... ");

    x = [0, 1,  3]
    y = [1, 0, 10]

    print("---   Create arrays of basis function coefficients ... ");

    P0_coeff = [ 1, -4.0/3.0,  1.0/3.0 ]
    P1_coeff = [ 0,      1.5,     -0.5 ]
    P2_coeff = [ 0, -1.0/6.0,  1.0/6.0 ]

    # Get the polynomial function
```

# Code 3: Lagrange Basis Polynomials

```
    print("---   Create and print polynomials ...");

    P0 = poly.Polynomial( P0_coeff )
    P1 = poly.Polynomial( P1_coeff )
    P2 = poly.Polynomial( P2_coeff )

    np.polynomial.set_default_printstyle('ascii')

    print("---   Create array of x values for plotting ... ");

    x_new = np.arange(-1.0, 3.1, 0.1)

    print("---   Plot Lagrange polynomials ... ");

    fig = plt.figure(figsize = (10,8))
    plt.plot( x_new, P0(x_new), 'b', label = 'p0(x)')
    plt.plot( x_new, P1(x_new), 'r', label = 'p1(x)')
    plt.plot( x_new, P2(x_new), 'g', label = 'p2(x)')

    plt.plot(x, np.ones(len(x)), 'ko', x, np.zeros(len(x)), 'ko')
    plt.title('Lagrange Basis Polynomials')
    plt.xlabel('x')
    plt.ylabel('p(x)')
    plt.grid()
    plt.legend()
    plt.show()

# call the main method ...

main()
```

# Code 4: Least Squares Analysis

```
# ==========================================================================
# TestLeastSquares01.py: Compute least squares analysis for linear and quadratic
#                        fits to test data.
# ==========================================================================

import math
import numpy as np
import numpy.polynomial.polynomial as poly
import matplotlib.pyplot as plt

import LinearMatrixEquations as lme

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

def main():
    print("--- Enter TestLeastSquares01.main()          ... ");
    print("--- ====================================== ... ");

    print("--- Step 1: Create (x,y) data array ... ");

    data = np.array([ [ 0, 1], [ 3, 2], [ 6, 1] ]);

    lme.printmatrix("data array", data );

    print("--- Step 2: Extract x and y arrays from data array ... ");

    x = data [:,0];
    y = data [:,1];
```

# Code 4: Least Squares Analysis

```

    lme.printvector("x", x);
    lme.printvector("y", y);

    print("--- Part 1: Linear least squares fit ... ");
    print("--- -------------------------------- ... ");

    n   = len(x);     # <-- number of data points ....

    a11 = n;          a12 = sum(x);
    a21 = sum(x);     a22 = np.dot(x,x)
    b11 = sum(y)
    b21 = np.dot(x,y)

    A1 = np.array([ [ a11, a12 ], [ a21, a22 ] ]);
    B1 = np.array([ [ b11 ], [ b21 ] ]);

    lme.printmatrix("A1", A1);
    lme.printmatrix("B1", B1);

    print("--- Step 4: Compute least squares coefficients ... ");

    coeff = lme.solvesystem( A1, B1);
    lme.printmatrix("Coeff", coeff );

    print("--- Step 5: Create and print polynomials ...");

    p_coeff = [ coeff[0][0], coeff[1][0] ]
    p = poly.Polynomial( p_coeff )
    print (p)
```

# Code 4: Least Squares Analysis

```
    print("--- Step 6: Compute mean square error ...");

    yfit = p(x)
    print( y - yfit )
    mse  = np.dot(y - yfit, y- yfit)/len(y)

    print("--- Mean square error = {:.3f} ...".format(mse))

    print("--- Part 2: Quadratic least squares fit ... ");
    print("--- ----------------------------------- ... ");

    n   = len(x);         # <-- number of data points ....
    a11 = n;           a12 = sum(x);
    a21 = sum(x);      a22 = np.dot(x,x)
    a31 = np.dot(x,x)

    b11 = sum(y)
    b21 = np.dot(x,y)

    # manually assemble array coefficients ...

    a23 = 0; a32 = 0; a33 = 0
    b31 = 0;
    i = 0
    while i < n:
        xi = x[i]
        yi = y[i]
        a32 = a32 + xi**3
```

# Code 4: Least Squares Analysis

```
        a33 = a33 + xi**4
        b31 = b31 + xi*xi*yi
        i = i + 1

    a13 = np.dot(x,x)
    a23 = a32

    A2 = np.array([ [ a11, a12, a13 ],
                    [ a21, a22, a23 ],
                    [ a31, a32, a33 ] ]);

    B2 = np.array([ [ b11 ], [ b21 ], [ b31 ] ]);

    lme.printmatrix("A2", A2);
    lme.printmatrix("B2", B2);

    print("--- Step 8: Compute least squares coefficients ... ");

    coeff = lme.solvesystem( A2, B2);
    lme.printmatrix("Coeff", coeff );

    print("--- Step 9: Create and print polynomials ...");

    q_coeff = [ coeff[0][0], coeff[1][0], coeff[2][0] ]
    q = poly.Polynomial( q_coeff )
    print (q)

    print("--- Step 10: Compute mean square error ...");
```

# Code 4: Least Squares Analysis

```

    yfit = q(x)
    print( y - yfit )
    mse  = np.dot(y - yfit, y- yfit)/len(y)

    print("--- Mean square error = {:.3f} ...".format(mse))

    print("--- Step 11: Graph data and least squares fit equations ... ");
    print("--- -------------------------------------------------- ... ");

    x_new = np.linspace(0.0, 6.0, num=20)

    fig = plt.figure(figsize = (10,8))
    plt.plot(x_new, p(x_new), 'b', label = 'p(x) = 1.33 + 0.00 x')
    plt.plot(x_new, q(x_new), 'r', label = 'q(x) = 1.00 + 0.66 x - 0.11 x^2')
    plt.plot(x, y, 'ko')
    plt.title('Linear/quadratic least squares fit (3 data points)')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid()
    plt.legend()
    plt.show()

    print("--- ========================================= ... ");
    print("--- Leave TestLeastSquares01.main()            ... ");

# call the main method ...

main()
```