PROGRAM STUDI INFORMATIKA
FAKULTAS TEKNIK DAN INFORMATIKA
UNIVERSITAS MULTIMEDIA NUSANTARA
SEMESTER GENAP TAHUN AJARAN 2021/2022

IF420 – ANALISIS NUMERIK

Pertemuan ke 7 – Interpolation

Seng Hansun, S.Si., M.Cs.

Capaian Pembelajaran Mingguan Mata Kuliah (Sub-CPMK):

Sub-CPMK 7: Mahasiswa mampu memahami dan menerapkan berbagai teknik Interpolasi – C3

Reviews

- Least Squares Regression Problem Statement
- Least Squares Regression Derivation (Linear Algebra)
- Least Squares Regression Derivation (Multivariable Calculus)
- Least Squares Regression in Python
- Least Squares Regression for Nonlinear Functions

Outlines

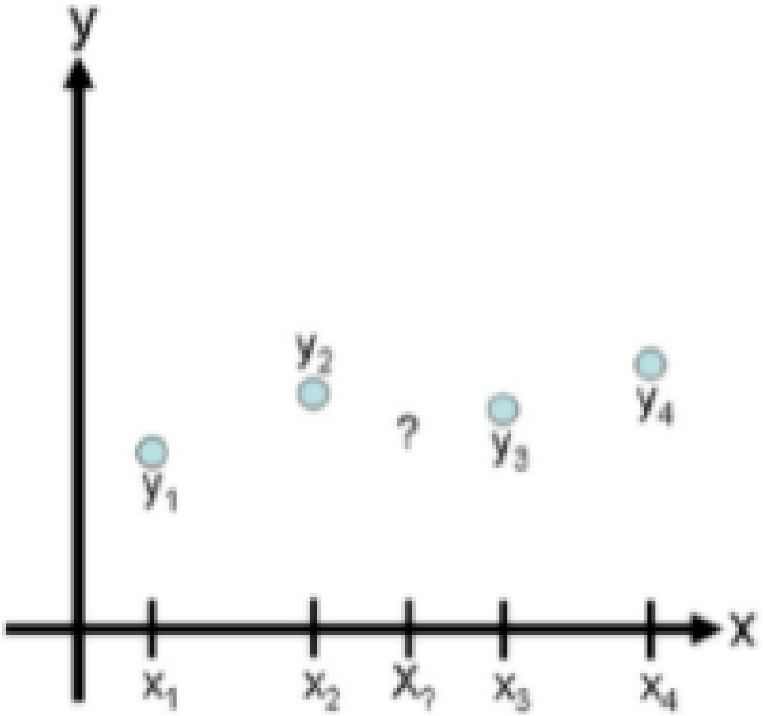
- Interpolation Problem Statement
- Linear Interpolation
- Cubic Spline Interpolation
- Lagrange Polynomial Interpolation
- Newton’s Polynomial Interpolation

Motivation

- The previous lesson used **regression** to find the **parameters** of a function that best estimated a **set of data points**.
- **Regression** assumes that the data set has measurement **errors**, and that you need to find a set of model **parameters** that **minimize** the **error** between your model and the data.
- However, sometimes you have measurements that are assumed to be **very reliable**; in these cases, you want an **estimation function** that **goes through** the **data points** you have.
- This technique is commonly referred to as **interpolation**.

The Problem Statement

- Assume we have a data set consisting of **independent** data values, x_i, and **dependent** data values, y_i, where $i = 1, \ldots, n$.
- We would like to find an **estimation function** $\hat{y}(x)$ such that $\hat{y}(x_i) = y_i$ for every point in our data set.
- This means the estimation function **goes through** our data points.
- Given a new $x*$, we can **interpolate** its function value using $\hat{y}(x*)$.
- In this context, $\hat{y}(x)$ is called an **interpolation function**.
- The following figure shows the interpolation problem statement.



The Problem Statement

- Unlike **regression**, **interpolation** **does not require** the user to have an **underlying model** for the data, especially when there are many **reliable data points**.
- However, the processes that underly the data must still inform the user about the **quality** of the **interpolation**.
- For example, our data may consist of (x, y) coordinates of a car over time.
- Since **motion** is restricted to the **maneuvering physics** of the car, we can expect that the points between the (x, y) coordinates in our set will be "**smooth**" rather than **jagged**.
- In the following sections we derive several common **interpolation methods**.

Linear Interpolation

- In **linear interpolation**, the **estimated** point is assumed to lie on the **line** joining the **nearest** **points** to the left and right.
- Assume, **without loss** of **generality**, that the x-data points are in **ascending** order; that is, $x_i < x_{i+1}$, and let x be a point such that $x_i < x < x_{i+1}$.
- Then the **linear interpolation** at x is:

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

Linear Interpolation

- **Example**: 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

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

$$= 3 + \frac{(2 - 3)(1.5 - 1)}{(2 - 1)} = 2.5.$$

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

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

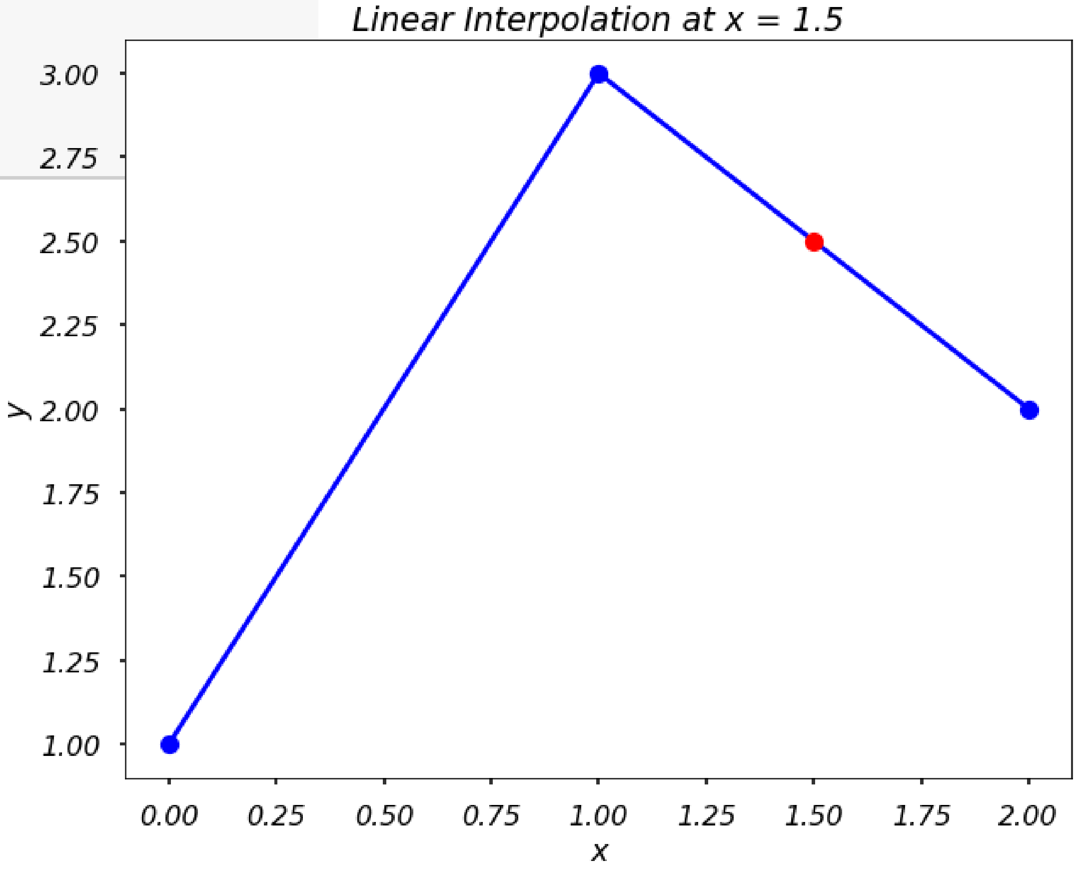
In [2]:
```
x = [0, 1, 2]
y = [1, 3, 2]

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

```
2.5
```

Linear Interpolation

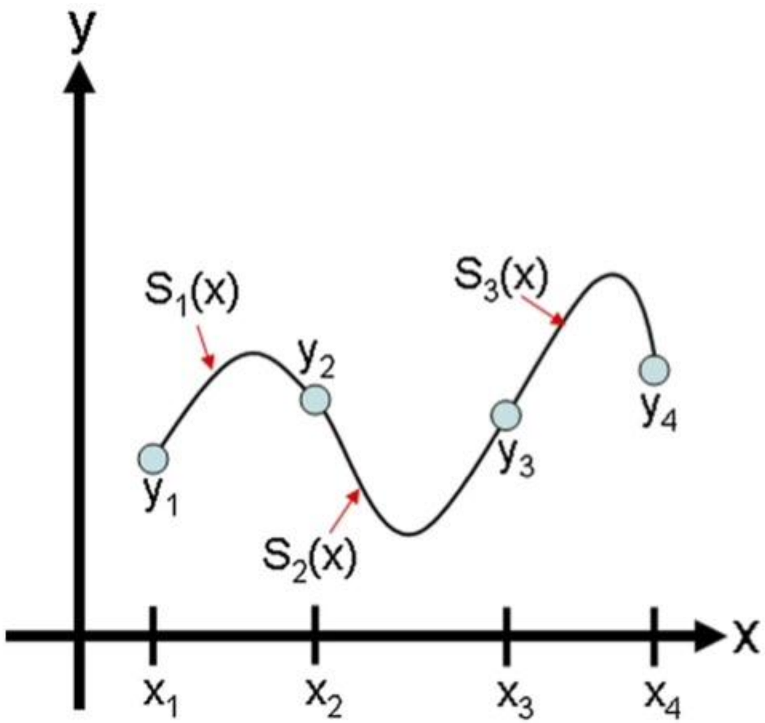
In [3]:

```
plt.figure(figsize = (10,8))
plt.plot(x, y, '-ob')
plt.plot(1.5, y_hat, 'ro')
plt.title('Linear Interpolation at x = 1.5')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Cubic Spline Interpolation

- In **cubic spline interpolation** (as shown in the following figure), the **interpolating function** is a set of **piecewise cubic functions**.
- Specifically, we assume that the points (x_i, y_i) and (x_{i+1}, y_{i+1}) are joined by a **cubic polynomial** $S_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$ that is valid for $x_i < x < x_{i+1}$ for $i = 1, \ldots, n-1$.
- To find the **interpolating** function, we must first determine the **coefficients** a_i, b_i, c_i, d_i for each of the **cubic functions**.
- For n points, there are $n-1$ cubic functions to find, and each cubic function requires **four coefficients**.
- Therefore we have a total of $4(n-1)$ **unknowns**, and so we need $4(n-1)$ **independent equations** to find all the **coefficients**.



Cubic Spline Interpolation

- First we know that the **cubic** functions must **intersect** the data the points on the left and the right:

$$S_i(x_i) = y_i, \qquad i = 1, \dots, n-1$$
$$S_i(x_{i+1}) = y_{i+1}, \qquad i = 1, \dots, n-1$$

which gives us $2(n-1)$ equations.

- Next, we want each cubic function to join as **smoothly** with its **neighbors** as possible, so we constrain the **splines** to have **continuous first** and **second derivatives** at the data points $i = 2, \dots, n-1$.

$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1}), \qquad i = 1, \dots, n-2$$
$$S_i''(x_{i+1}) = S_{i+1}''(x_{i+1}), \qquad i = 1, \dots, n-2$$

which gives us $2(n-2)$ equations.

Cubic Spline Interpolation

- **Two more equations** are required to compute the **coefficients** of $S_i(x)$.
- These last two constraints are **arbitrary**, and they can be chosen to **fit** the **circumstances** of the interpolation being performed.
- A common set of **final constraints** is to assume that the **second derivatives** are **zero** at the **endpoints**. This means that the curve is a "**straight line**" at the end points. Explicitly,

$$S_1''(x_1) = 0$$
$$S_{n-1}''(x_n) = 0$$

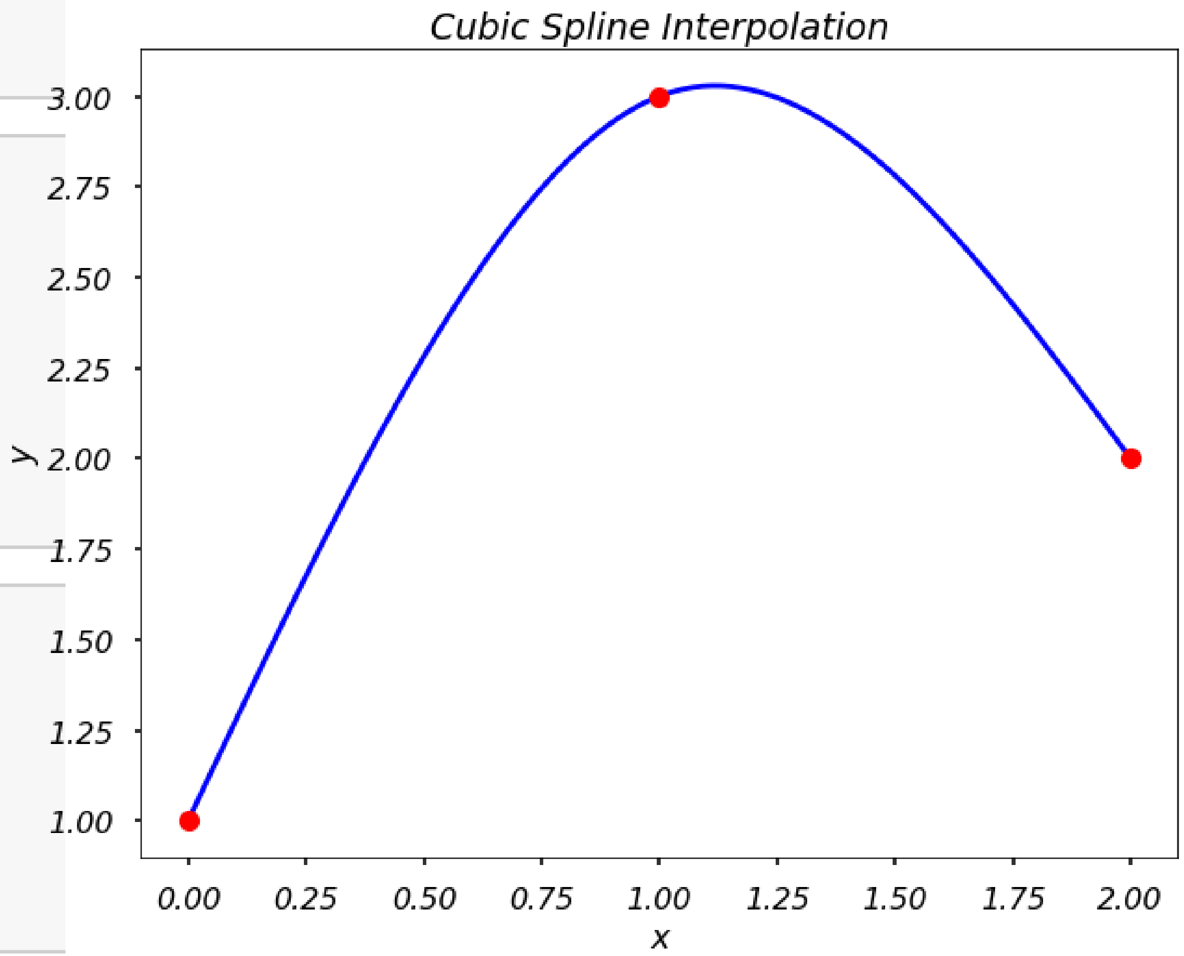
- In Python, we can use **scipy's** function **CubicSpline** to perform **cubic spline interpolation**.
- Note that the above constraints are **not the same** as the ones used by scipy's **CubicSpline** as default for performing cubic splines, there are **different ways** to **add** the **final two constraints** in scipy by setting the **bc_type** argument.

- **Example**: Use **CubicSpline** to plot the **cubic spline interpolation** of the data set $x = [0, 1, 2]$ and $y = [1, 3, 2]$ for $0 \le x \le 2$.

In [4]:
```python
from scipy.interpolate import CubicSpline
import numpy as np
import matplotlib.pyplot as plt

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

In [5]:
```python
x = [0, 1, 2]
y = [1, 3, 2]

# use bc_type = 'natural' adds the constraints
# as we described above
f = CubicSpline(x, y, bc_type='natural')
x_new = np.linspace(0, 2, 100)
y_new = f(x_new)
```

In [6]:
```python
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()
```



Cubic Spline Interpolation

- To determine the **coefficients** of each **cubic** function, we write out the **constraints** explicitly as a system of linear equations with $4(n-1)$ unknowns.
- For n data points, the **unknowns** are the coefficients a_i, b_i, c_i, d_i of the **cubic spline**, S_i joining the points x_i and x_{i+1}.

- For the **constraints** $S_i(x_i) = y_i$ we have

$$a_1 x_1^3 + b_1 x_1^2 + c_1 x_1 + d_1 = y_1,$$
$$a_2 x_2^3 + b_2 x_2^2 + c_2 x_2 + d_2 = y_2,$$
$$\dots$$
$$a_{n-1} x_{n-1}^3 + b_{n-1} x_{n-1}^2 + c_{n-1} x_{n-1} + d_{n-1} = y_{n-1}.$$

- For the **constraints** $S_i(x_{i+1}) = y_{i+1}$ we have

$$a_1 x_2^3 + b_1 x_2^2 + c_1 x_2 + d_1 = y_2,$$
$$a_2 x_3^3 + b_2 x_3^2 + c_2 x_3 + d_2 = y_3,$$
$$\dots$$
$$a_{n-1} x_n^3 + b_{n-1} x_n^2 + c_{n-1} x_n + d_{n-1} = y_n.$$

Cubic Spline Interpolation

- For the **constraints** $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$ we have

$$3a_1x_2^2 + 2b_1x_2 + c_1 - 3a_2x_2^2 - 2b_2x_2 - c_2 = 0,$$
$$3a_2x_3^2 + 2b_2x_3 + c_2 - 3a_3x_3^2 - 2b_3x_3 - c_3 = 0,$$
$$\ldots$$
$$3a_{n-2}x_{n-1}^2 + 2b_{n-2}x_{n-1} + c_{n-2} - 3a_{n-1}x_{n-1}^2 - 2b_{n-1}x_{n-1} - c_{n-1} = 0.$$

- For the **constraints** $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$ we have

$$6a_1x_2 + 2b_1 - 6a_2x_2 - 2b_2 = 0,$$
$$6a_2x_3 + 2b_2 - 6a_3x_3 - 2b_3 = 0,$$
$$\ldots$$
$$6a_{n-2}x_{n-1} + 2b_{n-2} - 6a_{n-1}x_{n-1} - 2b_{n-1} = 0.$$

Cubic Spline Interpolation

- Finally for the **endpoint constraints** $S_1''(x_1) = 0$ and $S_{n-1}''(x_n) = 0$, we have:

$$6a_1x_1 + 2b_1 = 0,$$
$$6a_{n-1}x_n + 2b_{n-1} = 0.$$

- These equations are **linear** in the **unknown coefficients** $a_i, b_i, c_i,$ and d_i.
- We can put them in **matrix form** and solve for the **coefficients** of each spline by **left division**.
- Remember that whenever we solve the matrix equation $Ax = b$ for x, we must make be sure that A is **square** and **invertible**.
- In the case of finding **cubic spline equations**, the A matrix is always square and invertible as long as the x_i values in the data set are **unique**.

Cubic Spline Interpolation

- **Example**: Find the cubic spline interpolation at $x = 1.5$ based on the data $x = [0, 1, 2]$, $y = [1, 3, 2]$.
- First we create the appropriate **system of equations** and find the **coefficients** of the **cubic splines** by solving the system in matrix form.
- For the **constraints** $S_i(x_i) = y_i$ we have

$$a_1x_1^3 + b_1x_1^2 + c_1x_1 + d_1 = y_1 \rightarrow d_1 = 1$$

$$a_2x_2^3 + b_2x_2^2 + c_2x_2 + d_2 = y_2 \rightarrow a_2 + b_2 + c_2 + d_2 = 3$$

- For the **constraints** $S_i(x_{i+1}) = y_{i+1}$ we have

$$a_1x_2^3 + b_1x_2^2 + c_1x_2 + d_1 = y_2 \rightarrow a_1 + b_1 + c_1 + d_1 = 3$$

$$a_2x_3^3 + b_2x_3^2 + c_2x_3 + d_2 = y_3 \rightarrow 8a_2 + 4b_2 + 2c_2 + d_2 = 2$$

Cubic Spline Interpolation

- For the **constraints** $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$ we have

$$3a_1x_2^2 + 2b_1x_2 + c_1 - 3a_2x_2^2 - 2b_2x_2 - c_2 = 0 \rightarrow 3a_1 + 2b_1 + c_1 - 3a_2 - 2b_2 + c_2 = 0$$

- For the **constraints** $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$ we have

$$6a_1x_2 + 2b_1 - 6a_2x_2 - 2b_2 = 0 \rightarrow 6a_1 + 2b_1 - 6a_2 - 2b_2 = 0$$

- Finally for the **endpoint constraints** $S_1''(x_1) = 0$ and $S_{n-1}''(x_n) = 0$, we have:

$$6a_1x_1 + 2b_1 = 0 \rightarrow 2b_1 = 0$$

$$6a_2x_3 + 2b_2 = 0 \rightarrow 12a_2 + 2b_2 = 0$$

Cubic Spline Interpolation

- The **matrix form** of the **system of equations** is:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 4 & 2 & 1 \\ 3 & 2 & 1 & 0 & -3 & -2 & -1 & 0 \\ 6 & 2 & 0 & 0 & -6 & -2 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 12 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ a_2 \\ b_2 \\ c_2 \\ d_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 3 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Cubic Spline Interpolation

```
In [7]:  b = np.array([1, 3, 3, 2, 0, 0, 0, 0])
         b = b[:, np.newaxis]
         A = np.array([[0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 1, 1, 1, 1], [1, 1, 1, 1, 0, 0, 0, 0], \
                       [0, 0, 0, 0, 8, 4, 2, 1], [3, 2, 1, 0, -3, -2, -1, 0], [6, 2, 0, 0, -6, -2, 0, 0],\
                       [0, 2, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 12, 2, 0, 0]])
```

```
In [8]:  np.dot(np.linalg.inv(A), b)

Out[8]:  array([[-0.75],
                [ 0.  ],
                [ 2.75],
                [ 1.  ],
                [ 0.75],
                [-4.5 ],
                [ 7.25],
                [-0.5 ]])
```

- Therefore, the **two cubic polynomials** are

$$S_1(x) = -.75x^3 + 2.75x + 1, \qquad for\ 0 \le x \le 1$$

$$S_2(x) = .75x^3 - 4.5x^2 + 7.25x - .5, \qquad for\ 1 \le x \le 2$$

- So for $x = 1.5$ we evaluate $S_2(1.5)$ and get an estimated value of $\mathbf{2.78125}$.

Lagrange Polynomial Interpolation

- Rather than finding **cubic polynomials** between **subsequent pairs** of **data points**, **Lagrange polynomial interpolation** finds a **single polynomial** that goes through **all** the **data points**.

- This polynomial is referred to as a **Lagrange polynomial**, $L(x)$, and as an **interpolation** function, it should have the property $L(x_i) = y_i$ for every point in the data set.

- For computing **Lagrange polynomials**, it is useful to write them as a **linear combination** of **Lagrange basis polynomials**, $P_i(x)$, where

$$P_i(x) = \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}, \qquad and \qquad L(x) = \sum_{i=1}^{n} y_i P_i(x).$$

- You will notice that by construction, $P_i(x)$ has the property that $\boldsymbol{P_i(x_j) = 1}$ **when** $\boldsymbol{i = j}$ and $\boldsymbol{P_i(x_j) = 0}$ **when** $\boldsymbol{i \neq j}$. Since $L(x)$ is a **sum of these polynomials**, you can observe that $\boldsymbol{L(x_i) = y_i}$ for every point, exactly as desired.

Lagrange Polynomial Interpolation

- **Example**: Find the **Lagrange basis polynomials** for the data set $x = [0, 1, 2]$ and $y = [1, 3, 2]$. Plot each polynomial and verify the property that $P_i\left(x_j\right) = 1$ when $i = j$ and $P_i\left(x_j\right) = 0$ when $i \neq j$.

$$P_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} = \frac{(x - 1)(x - 2)}{(0 - 1)(0 - 2)} = \frac{1}{2}(x^2 - 3x + 2),$$

$$P_2(x) = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} = \frac{(x - 0)(x - 2)}{(1 - 0)(1 - 2)} = -x^2 + 2x,$$

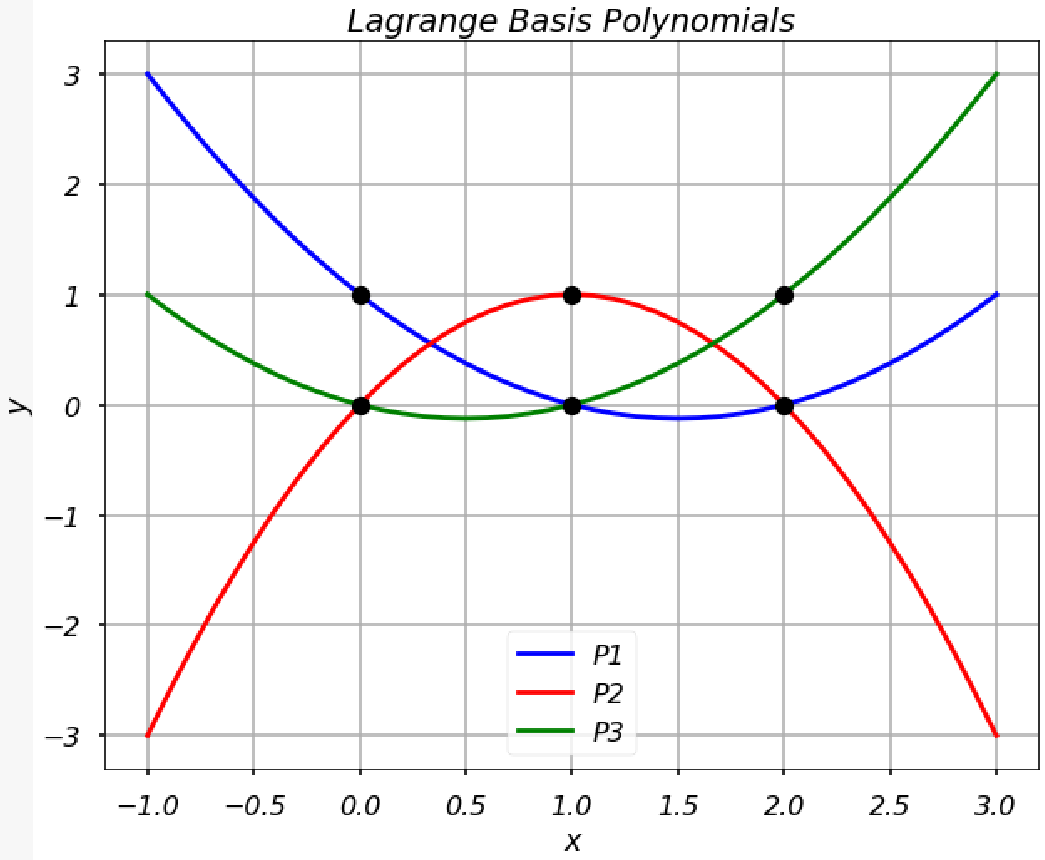
$$P_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} = \frac{(x - 0)(x - 1)}{(2 - 0)(2 - 1)} = \frac{1}{2}(x^2 - x).$$

In [11]:

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

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

In [12]:

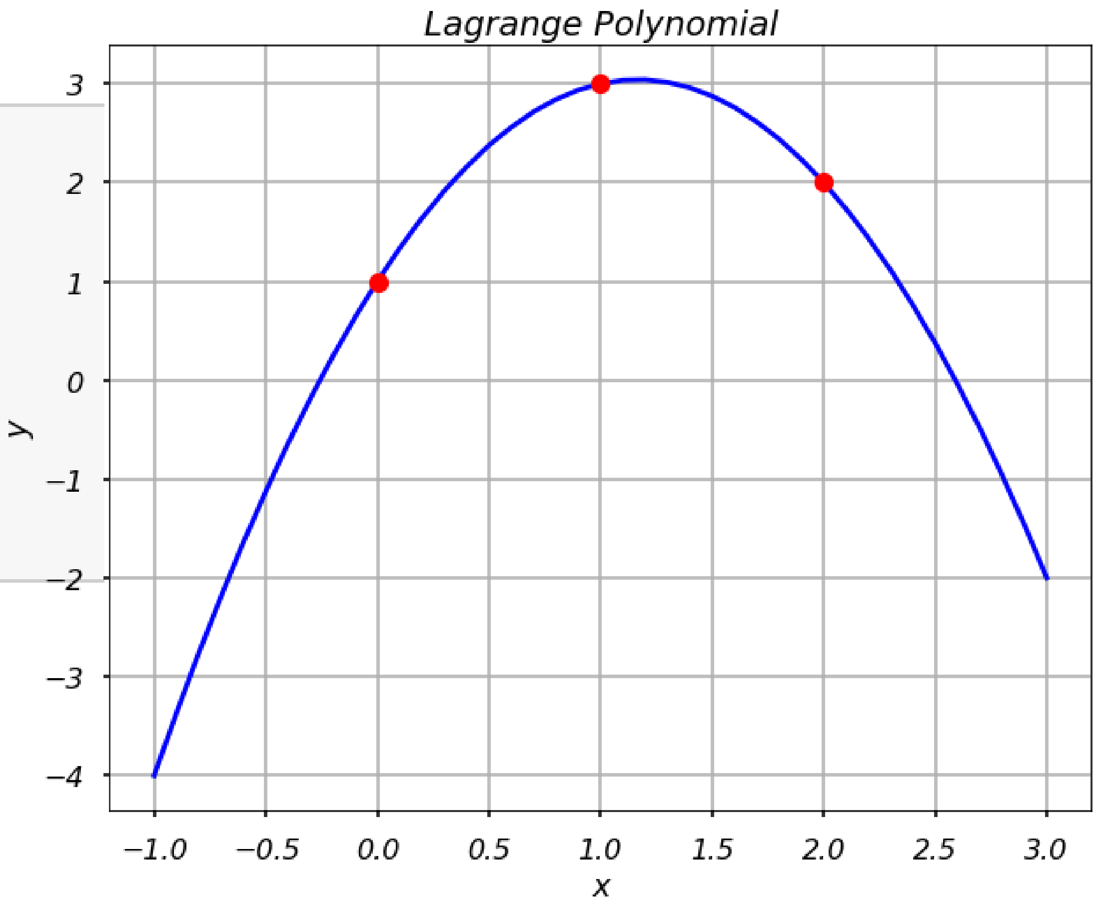
```python
x = [0, 1, 2]
y = [1, 3, 2]
P1_coeff = [1,-1.5,.5]
P2_coeff = [0, 2,-1]
P3_coeff = [0,-.5,.5]

# get the polynomial function
P1 = poly.Polynomial(P1_coeff)
P2 = poly.Polynomial(P2_coeff)
P3 = poly.Polynomial(P3_coeff)

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

fig = plt.figure(figsize = (10,8))
plt.plot(x_new, P1(x_new), 'b', label = 'P1')
plt.plot(x_new, P2(x_new), 'r', label = 'P2')
plt.plot(x_new, P3(x_new), 'g', label = 'P3')

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



Lagrange Polynomial Interpolation

- **Example**: For the previous example, compute and plot the **Lagrange polynomial** and verify that it goes through each of the data points.

In [13]:
```
L = P1 + 3*P2 + 2*P3

fig = plt.figure(figsize = (10,8))
plt.plot(x_new, L(x_new), 'b', x, y, 'ro')
plt.title('Lagrange Polynomial')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



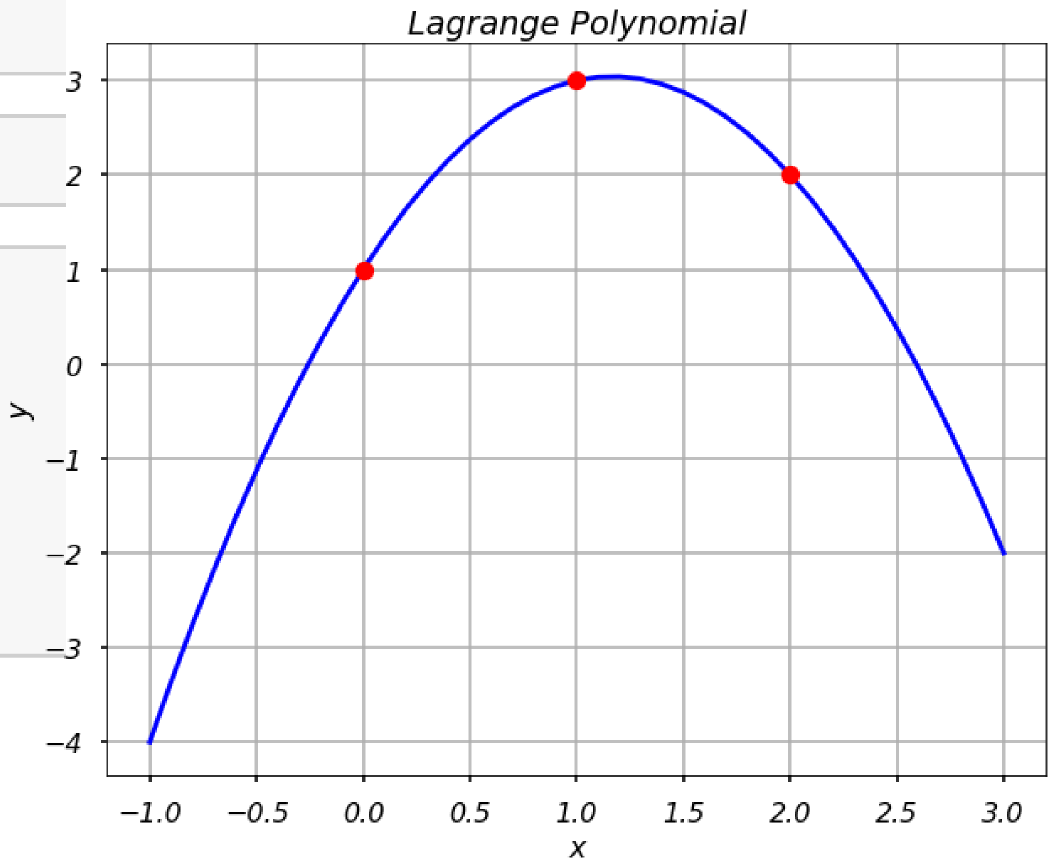
Using Lagrange from Scipy

- Instead of calculating everything from scratch, in **scipy**, we can use the **lagrange** function directly to interpolate the data.

In [14]:
```
from scipy.interpolate import lagrange
```

In [15]:
```
f = lagrange(x, y)
```

In [16]:
```
fig = plt.figure(figsize = (10,8))
plt.plot(x_new, f(x_new), 'b', x, y, 'ro')
plt.title('Lagrange Polynomial')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Newton's Polynomial Interpolation

- **Newton's polynomial interpolation** is another popular way to **fit exactly** for a set of data points.
- The **general form** of the an $n-1$ **order Newton's polynomial** that goes through n points is:

$$f(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n(x - x_0)(x - x_1) \ldots (x - x_{n-1})$$

which can be **re-written** as:

$$f(x) = \sum_{i=0}^{n} a_i n_i(x)$$

where $n_i(x) = \prod_{j=0}^{i-1}(x - x_j)$.

Newton's Polynomial Interpolation

- The **special** **feature** of the **Newton's polynomial** is that the coefficients a_i can be determined using a very simple Mathematical procedure.
- For example, since the polynomial goes through each data points, therefore, for a data points (x_i, y_i), we will have $f(x_i) = y_i$, thus we have

$$f(x_0) = a_0 = y_0 \quad \text{and} \quad f(x_1) = a_0 + a_1(x_1 - x_0) = y_1$$

by **rearranging** it to get a_1, we will have:

$$a_1 = \frac{y_1 - y_0}{x_1 - x_0}$$

- Now, **insert** data points (x_2, y_2), we can calculate a_2, and it is in the form:

$$a_2 = \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0}$$

Newton's Polynomial Interpolation

- Let's do one **more data points** (x_3, y_3) to calculate a_3, after **insert** the data point into the equation, we get:

$$a_3 = \frac{\dfrac{\dfrac{y_3 - y_2}{x_3 - x_2} - \dfrac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1} - \dfrac{\dfrac{y_2 - y_1}{x_2 - x_1} - \dfrac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0}}{x_3 - x_0}$$

- Now, see the patterns? These are called **divided differences**, if we define:

$$f[x_2, x_1, x_0] = \frac{\dfrac{y_2 - y_1}{x_2 - x_1} - \dfrac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0} = \frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_0}.$$

- We continue write this out, we will have the following **iteration equation**:

$$f[x_k, x_{k-1}, \dots, x_1, x_0] = \frac{f[x_k, x_{k-1}, \dots, x_2, x_1] - f[x_{k-1}, x_{k-2}, \dots, x_1, x_0]}{x_k - x_0}$$

Newton's Polynomial Interpolation

- We can see one beauty of the method is that, once the **coefficients** are **determined**, adding **new data** points **won't change** the **calculated ones**, we only need to calculate **higher differences** continues in the **same manner**.
- The **whole procedure** for finding these coefficients can be summarized into a **divided differences table**.
- Let's see an example using 5 data points:

$$
\begin{array}{cccccc}
x_0 & y_0 & & & & \\
 & & f[x_1,x_0] & & & \\
x_1 & y_1 & & f[x_2,x_1,x_0] & & \\
 & & f[x_2,x_1] & & f[x_3,x_2,x_1,x_0] & \\
x_2 & y_2 & & f[x_3,x_2,x_1] & & f[x_4,x_3,x_2,x_1,x_0] \\
 & & f[x_3,x_2] & & f[x_4,x_3,x_2,x_1] & \\
x_3 & y_3 & & f[x_4,x_3,x_2] & & \\
 & & f[x_4,x_3] & & & \\
x_4 & y_4 & & & &
\end{array}
$$

Newton's Polynomial Interpolation

- **Each element** in the table can be calculated using the **two previous elements** (to the left).

- In reality, we can calculate each element and store them into a **diagonal matrix**, that is the **coefficients matrix** can be written as:

$$
\begin{matrix}
y_0 & f[x_1, x_0] & f[x_2, x_1, x_0] & f[x_3, x_2, x_1, x_0] & f[x_4, x_3, x_2, x_1, x_0] \\
y_1 & f[x_2, x_1] & f[x_3, x_2, x_1] & f[x_4, x_3, x_2, x_1] & 0 \\
y_2 & f[x_3, x_2] & f[x_4, x_3, x_2] & 0 & 0 \\
y_3 & f[x_4, x_3] & 0 & 0 & 0 \\
y_4 & 0 & 0 & 0 & 0
\end{matrix}
$$

- Note that, the **first row** in the matrix is actually **all** the **coefficients** that we need, i.e. a_0, a_1, a_2, a_3, a_4. Let's see an example how we can do it.

- **Example**: Calculate the **divided differences** table for $x = [-5, -1, 0, 2]$, $y = [-2, 6, 1, 3]$.

In [17]:

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

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

%matplotlib inline
```

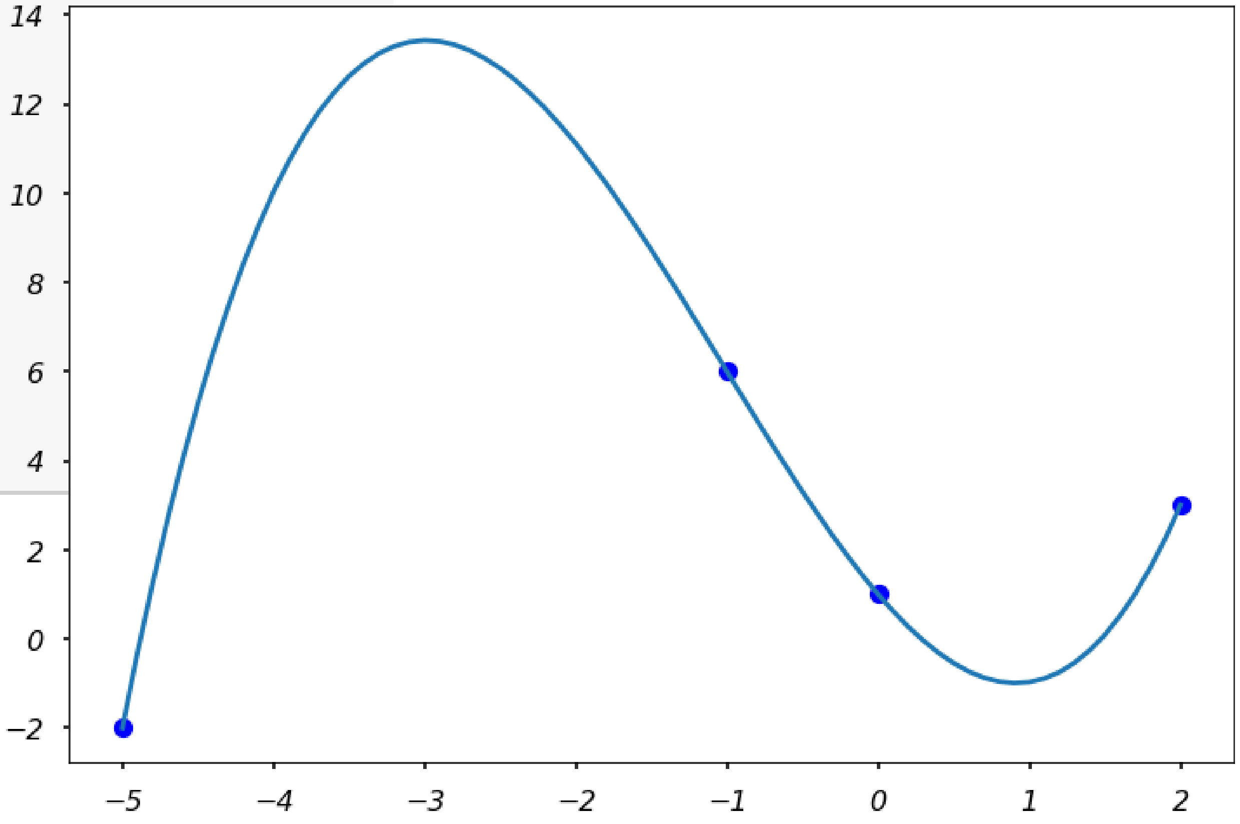
In [18]:

```python
def divided_diff(x, y):
    '''
    function to calculate the divided
    differences 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
```

Newton's Polynomial Interpolation

In [19]:

```
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)
```



- We can see that the Newton's polynomial goes through all the data points and fit the data.

Practice

1. Write a function **my_lin_interp(x, y, X)**, where **x** and **y** are arrays containing experimental data points, and **X** is an array. Assume that **x** and **X** are in ascending order and have unique elements. The output argument, **Y**, should be an array, the same size as **X**, where **Y[i]** is the **linear interpolation** of **X[i]**. You should **not** use **interp** from numpy or **interp1d** from scipy.

```
# Test case
x = [0, 1, 2]
y = [1, 3, 2]
X = [0.0,0.5,1.0,1.5,2.0]

Y = my_lin_interp(x,y,X)
Y
```

```
array([1. , 2. , 3. , 2.5, 2. ])
```

```
# Another test case
x = [-2, 0, 2, 3, 6]
y = [2, 0, 2, 1, 2]
X = [-1, -0.5, 0.5, 1, 2.5, 4, 5]

Y = my_lin_interp(x,y,X)
Y
```

```
array([1.        , 0.5       , 0.5       , 1.        , 1.5       ,
       1.33333333, 1.66666667])
```

Next Week's Outline

- Mid-term Exam

References

- Kong, Qingkai; Siauw, Timmy, and Bayen, Alexandre. 2020. Python Programming and Numerical Methods: A Guide for Engineers and Scientists. Academic Press. https://www.elsevier.com/books/python-programming-and-numerical-methods/kong/978-0-12-819549-9
- Other online and offline references

Visi

Menjadi Program Studi Strata Satu Informatika **unggulan** yang menghasilkan lulusan **berwawasan internasional** yang **kompeten** di bidang Ilmu Komputer (*Computer Science*), **berjiwa wirausaha** dan **berbudi pekerti luhur**.

INFORMATIKA
UMN
UNIVERSITAS MULTIMEDIA NUSANTARA

Misi

1. Menyelenggarakan pembelajaran dengan teknologi dan kurikulum terbaik serta didukung tenaga pengajar profesional.
2. Melaksanakan kegiatan penelitian di bidang Informatika untuk memajukan ilmu dan teknologi Informatika.
3. Melaksanakan kegiatan pengabdian kepada masyarakat berbasis ilmu dan teknologi Informatika dalam rangka mengamalkan ilmu dan teknologi Informatika.