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

IF420 – ANALISIS NUMERIK

Pertemuan ke 6 – Least Squares Regression

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

Capaian Pembelajaran Mingguan Mata Kuliah (Sub-CPMK):

Sub-CPMK 6: Mahasiswa mampu memahami dan menerapkan regresi kuadrat terkecil – C3

Reviews

- Eigenvalues and Eigenvectors Problem Statement
- The Power Method
- The QR Method
- Eigenvalues and Eigenvectors in Python

Outlines

- 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

Motivation

- Often in science and engineering coursework, we are asked to determine the **state** of a **system** given the **parameters** of the system.
- For example, the relationship between the force exerted by a linear spring, F, and the displacement of the spring from its natural length, x, is usually represented by the model

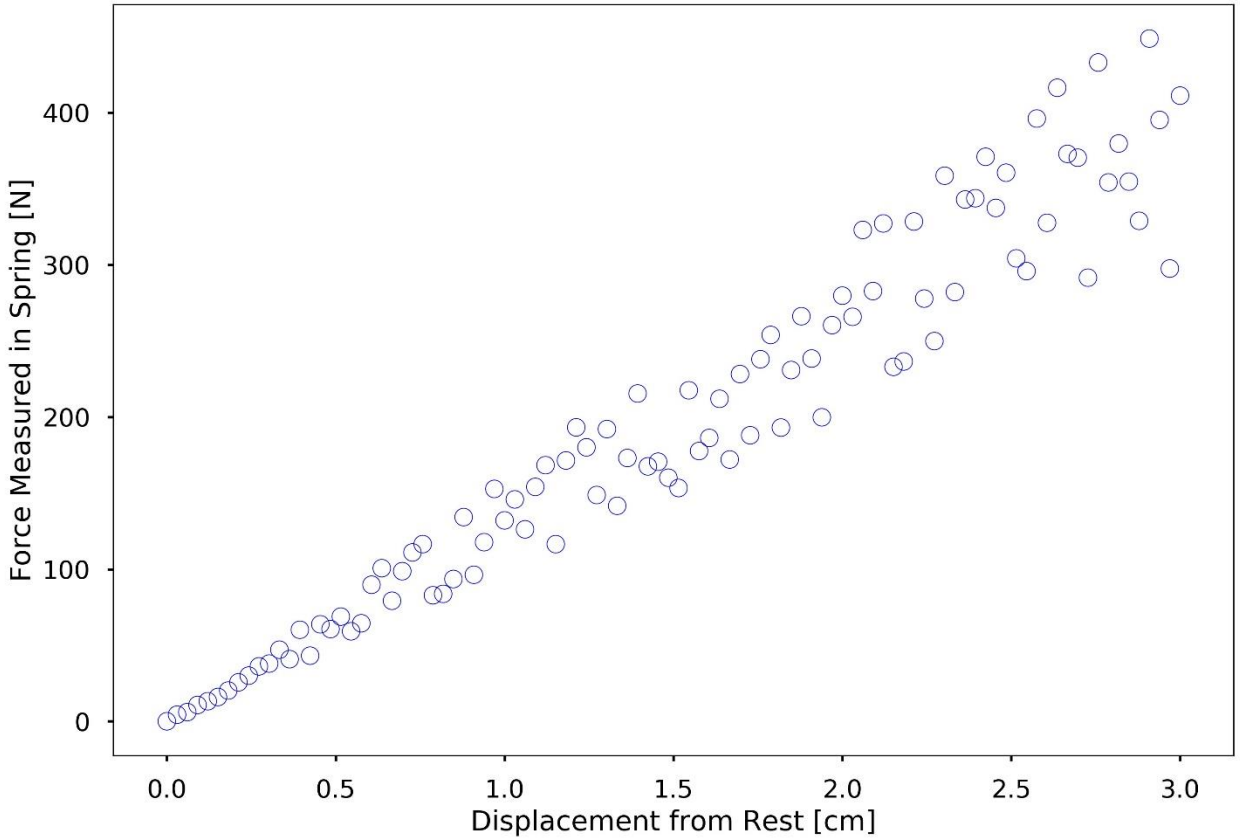
$$F = kx$$

  where k is the spring stiffness. We are then asked to compute the force for a given k and x value.

- However in practice, the stiffness and in general, most of the **parameters** of a system, are **not known** a **priori**.
- Instead, we are usually presented with **data points** about how the system has **behaved** in the past.

- For our spring example, we may be given (x, F) data pairs that have been previously recorded from an experiment. Ideally, all these data points would lie exactly on a line going through the origin (since there is no force at zero displacement). We could then measure the slope of this line and get our **stiffness value** for k. However, practical data usually has some measurement **noise** because of **sensor inaccuracy**, **measurement error**, or a variety of other reasons.

- The following figure shows an example of what data might look like for a simple spring experiment.
- Today lesson teaches **methods** of finding the "**most likely**" **model parameters given a set of data**, e.g., how to find the **spring stiffness** in our mock experiment.



The Problem Statement

- Given a set of **independent** data points x_i and **dependent** data points $y_i, i = 1, \dots, m$, we would like to find an **estimation function**, $\hat{y}(x)$, that describes the data **as well as** possible. Note that $\hat{y}$ can be a **function** of **several variables**, but for the sake of this discussion, we **restrict** the domain of $\hat{y}$ to be a **single variable**.
- In **least squares regression**, the **estimation function** must be a **linear combination** of **basis functions**, $f_i(x)$. That is, the estimation function must be of the form

$$\hat{y}(x) = \sum_{i=1}^{n} \alpha_i f_i(x)$$

- The **scalars** α_i are referred to as the **parameters** of the estimation function, and each **basis function** must be **linearly independent** from the others. In other words, in the proper "**functional space**" no basis function should be expressible as a linear combination of the other functions. **Note**: In general, there are significantly **more data points**, m, **than basis functions**, n (i.e., $m >> n$).

The Problem Statement

- **Example**: Create an estimation function for the force-displacement relationship of a linear spring. Identify the **basis function(s)** and **model parameters**.
- The **relationship** between the force, F, and the displacement, x, can be described by the function $F(x) = kx$ where k is the spring stiffness. The only basis function is the function $f_1(x) = x$ and the model parameter to find is $\alpha_1 = k$.
- The **goal** of **least squares regression** is to find the **parameters** of the **estimation function** that **minimize the total squared error**, E, defined by $E = \sum_{i=1}^{m} (\hat{y} - y_i)^2$.
- The **individual errors** or **residuals** are defined as $e_i = (\hat{y} - y_i)$. If e is the vector containing **all the individual errors**, then we are also trying to minimize $E = \|e\|_2^2$, which is the $\boldsymbol{L_2}$ **norm** defined in the previous lesson.
- Next, we will derive the **least squares method** of finding the **desired parameters**.

LSR Derivation (Linear Algebra)

- First, we enumerate the estimation of the data at each data point x_i

$$\hat{y}(x_1) = \alpha_1 f_1(x_1) + \alpha_2 f_2(x_1) + \cdots + \alpha_n f_n(x_1),$$
$$\hat{y}(x_2) = \alpha_1 f_1(x_2) + \alpha_2 f_2(x_2) + \cdots + \alpha_n f_n(x_2),$$
$$\ldots$$
$$\hat{y}(x_m) = \alpha_1 f_1(x_m) + \alpha_2 f_2(x_m) + \cdots + \alpha_n f_n(x_m).$$

- Let $X \in \mathbb{R}^n$ be a **column vector** such that the i-th element of X contains the value of the i-th x-data point, x_i, $\hat{Y}$ be a **column vector** with elements, $\hat{Y} = \hat{y}(x_i)$, β be a **column vector** such that $\beta_i = \alpha_i$, $F_i(x)$ be a **function** that returns a column vector of $f_i(x)$ computed on every element of x, and A be an $m \times n$ **matrix** such that the i-th column of A is $F_i(x)$.
- Given this notation, the previous **system of equations** becomes $\hat{Y} = A\beta$.

- Now if Y is a column vector such that $Y_i = y_i$, the **total squared error** is given by $E = \left\|\hat{Y} - Y\right\|_2^2$. You can verify this by substituting the definition of the L_2 **norm**.
- Since we want to make E as **small** as possible and norms are a **measure of distance**, this previous expression is equivalent to saying that we want $\hat{Y}$ and Y to be a "close as possible." Note that in general Y will not be in the range of A and therefore $E > 0$.
- The vector in the range of A, $\hat{Y}$, that is closest to Y is the one that can point **perpendicularly** to Y. Therefore, we want a vector $Y - \hat{Y}$ that is perpendicular to the vector $\hat{Y}$.
- Recall from Linear Algebra that two vectors are **perpendicular**, or **orthogonal**, if their **dot product** is 0. Noting that the dot product between two vectors, v and w, can be written as $dot(v, w) = v^T w$, we can state that $\hat{Y}$ and $Y - \hat{Y}$ are **perpendicular** if $dot(\hat{Y}, Y - \hat{Y}) = 0$.
- Therefore, $\hat{Y}^T\left(Y - \hat{Y}\right) = 0$, which is **equivalent** to $(A\beta)^T(Y - A\beta) = 0$.

LSR Derivation (Linear Algebra)

- Noting that for two matrices A and B, $(AB)^T = B^T A^T$ and using **distributive** properties of **vector multiplication**, this is equivalent to

$$\beta^T A^T Y - \beta^T A^T A\beta = \beta^T (A^T Y - A^T A\beta) = 0.$$

- The solution, $\beta = 0$, is a **trivial solution**, so we use $A^T Y - A^T A\beta = 0$ to find a more **interesting solution**.
- Solving this equation for β gives the **least squares regression** formula:

$$\beta = (A^T A)^{-1} A^T Y$$

- Note that $(A^T A)^{-1} A^T$ is called the **pseudo-inverse** of A and exists when $m > n$ and A has **linearly independent** columns.

LSR Derivation (Multivariable Calculus)

- Recall that the **total error** for m data points and n basis functions is:

$$E = \sum_{i=1}^{m} e_i^2 = \sum_{i=1}^{m} (\hat{y}(x_i) - y_i)^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} \alpha_j f_j (x_i) - y_i \right)^2 .$$

  which is an ***n*-dimensional paraboloid** in α_k.

- From Calculus, we know that the **minimum** of a **paraboloid** is where all the **partial derivatives equal zero**. So taking partial derivative of E with respect to the variable α_k (remember that in this case the parameters are our variables), setting the system of equations equal to 0 and solving for the α_k's should give the correct results.

LSR Derivation (Multivariable Calculus)

- The **partial derivative** with respect to α_k and setting equal to 0 yields:

$$\frac{\partial E}{\partial \alpha_k} = \sum_{i=1}^{m} 2\left(\sum_{j=1}^{n} \alpha_j f_j(x_i) - y_i\right) f_k(x_i) = 0$$

- With some rearrangement, the previous expression can be manipulated to the following:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} \alpha_j f_j(x_i) f_k(x_i) - \sum_{i=1}^{m} y_i f_k(x_i) = 0$$

and further rearrangement taking advantage of the fact that addition **commutes** results in

$$\sum_{j=1}^{n} \alpha_j \sum_{i=1}^{m} f_j(x_i) f_k(x_i) = \sum_{i=1}^{m} y_i f_k(x_i)$$

LSR Derivation (Multivariable Calculus)

- Now let X be a column vector such that the $i-$th element of X is x_i and Y similarly constructed, and let $F_j(X)$ be a column vector such that the $i-$th element of $F_j(X)$ is $f_j(x_i)$.
- Using this notation, the previous expression can be rewritten in **vector notation** as:

$$\left[F_k^T(X)F_1(X), F_k^T(X)F_2(X), \dots, F_k^T(X)F_j(X), \dots, F_k^T(X)F_n(X)\right]\begin{bmatrix}\alpha_1\\ \alpha_2\\ \cdots\\ \alpha_j\\ \cdots\\ \alpha_n\end{bmatrix} = F_k^T(X)Y$$

LSR Derivation (Multivariable Calculus)

- If we repeat this equation for every k, we get the following **system of linear equations** in **matrix form**:

$$\begin{bmatrix} F_1^T(X)F_1(X), F_1^T(X)F_2(X), \ldots, F_1^T(X)F_j(X), \ldots, F_1^T(X)F_n(X) \\ F_2^T(X)F_1(X), F_2^T(X)F_2(X), \ldots, F_2^T(X)F_j(X), \ldots, F_2^T(X)F_n(X) \\ \\ F_n^T(X)F_1(X), F_n^T(X)F_2(X), \ldots, F_n^T(X)F_j(X), \ldots, F_n^T(X)F_n(X) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \ldots \\ \alpha_j \\ \ldots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} F_1^T(X)Y \\ F_2^T(X)Y \\ \ldots \\ F_n^T(X)Y \end{bmatrix}.$$

- If we let $A = \left[F_1(X), F_2(X), \ldots, F_j(X), \ldots, F_n(X)\right]$ and β be a column vector such that j-th element of β is α_j, then the previous system of equations becomes $A^T A\beta = A^T Y$, and solving this matrix equation for β gives $\beta = (A^T A)^{-1} A^T Y$, which is exactly the same formula as the previous derivation.

LSR in Python

- Recall that if we enumerate the estimation of the data at each data point, x_i, this gives us the following system of equations:

$$\begin{aligned}
\hat{y}(x_1) &= \alpha_1 f_1(x_1) + \alpha_2 f_2(x_1) + \cdots + \alpha_n f_n(x_1),\\
\hat{y}(x_2) &= \alpha_1 f_1(x_2) + \alpha_2 f_2(x_2) + \cdots + \alpha_n f_n(x_2),\\
&\cdots\\
\hat{y}(x_m) &= \alpha_1 f_1(x_m) + \alpha_2 f_2(x_m) + \cdots + \alpha_n f_n(x_m).
\end{aligned}$$

- If the data was absolutely **perfect** (i.e., **no noise**), then the estimation function would go through all the data points, resulting in the following system of equations:

$$\begin{aligned}
y_1 &= \alpha_1 f_1(x_1) + \alpha_2 f_2(x_1) + \cdots + \alpha_n f_n(x_1),\\
y_2 &= \alpha_1 f_1(x_2) + \alpha_2 f_2(x_2) + \cdots + \alpha_n f_n(x_2),\\
&\cdots\\
y_m &= \alpha_1 f_1(x_m) + \alpha_2 f_2(x_m) + \cdots + \alpha_n f_n(x_m).
\end{aligned}$$

- If we take A to be as defined previously, this would result in the matrix equation $Y = A\beta$.

LSR in Python

- However, since the data is **not perfect**, there will not be an **estimation function** that can go through all the data points, and this system will have **no solution**.
- Therefore, we need to use the **least square regression** that we derived in the previous two sections to get a solution.

$$\beta = (A^T A)^{-1} A^T Y$$

- **Example**: Consider the artificial data created by $x = np.linspace(0, 1, 101)$ and $y = 1 + x + x * np.random.random(len(x))$. Do a least squares regression with an estimation function defined by $\hat{y} = \alpha_1 x + \alpha_2$. Plot the data points along with the least squares regression. Note that we expect $\alpha_1 = 1.5$ and $\alpha_2 = 1.0$ based on this data. Due to the random noise we added into the data, your results maybe slightly different.

Use Direct Inverse Method

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

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

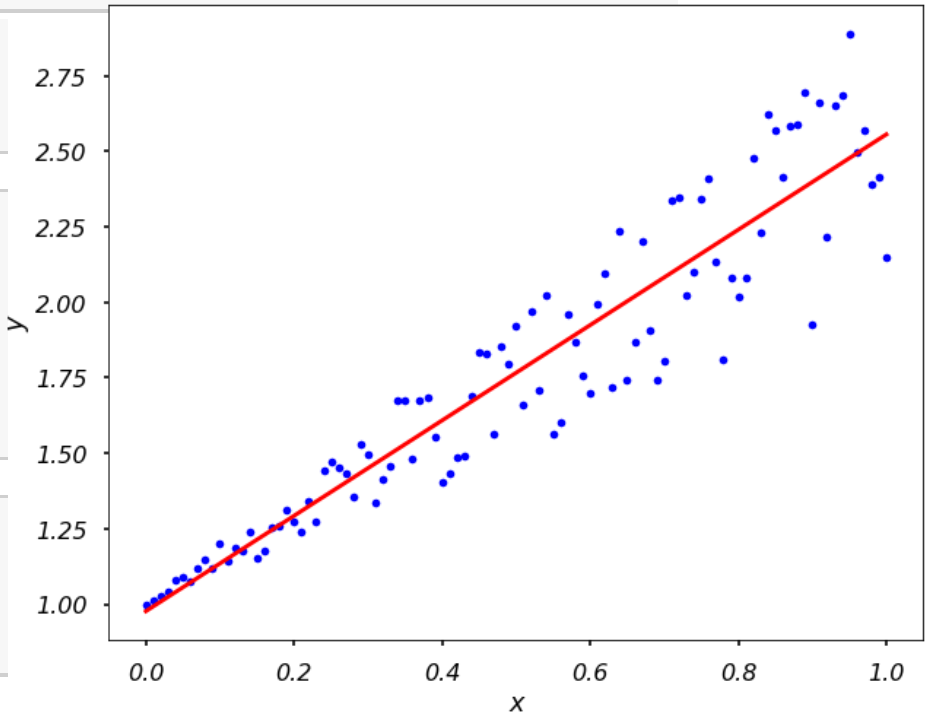
In [2]:
```
# generate x and y
x = np.linspace(0, 1, 101)
y = 1 + x + x * np.random.random(len(x))
```

In [3]:
```
# assemble matrix A
A = np.vstack([x, np.ones(len(x))]).T

# turn y into a column vector
y = y[:, np.newaxis]
```

In [4]:
```
# Direct least square regression
alpha = np.dot((np.dot(np.linalg.inv(np.dot(A.T,A)),A.T)),y)
print(alpha)
```

```
[[1.57877489]
 [0.976713  ]]
```

In [5]:
```
# plot the results
plt.figure(figsize = (10,8))
plt.plot(x, y, 'b.')
plt.plot(x, alpha[0]*x + alpha[1], 'r')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



IF

Use the Pseudoinverse

- We talked before that the $(A^T A)^{-1} A^T$ is called the **pseudo-inverse**, therefore, we could use the **pinv** function in **numpy** to directly calculate it.

```
In [6]: pinv = np.linalg.pinv(A)
        alpha = pinv.dot(y)
        print(alpha)
```

```
[[1.57877489]
 [0.976713  ]]
```

Use numpy.linalg.lstsq

- Actually, **numpy** has already implemented the **least square methods** that we can just call the function to get a solution.
- The function will return more things than the solution itself, please check the documentation for details.

```
In [7]: alpha = np.linalg.lstsq(A, y, rcond=None)[0]
        print(alpha)
```

```
[[1.57877489]
 [0.976713  ]]
```

Use optimize.curve_fit from Scipy

- This **scipy** function is actually very powerful, that it can fit not only **linear** functions, but many different function forms, such as **non-linear** function. Here we will show the linear example from above.
- Note that, using this function, we don't need to turn y into a column vector.

In [8]:
```
# generate x and y
x = np.linspace(0, 1, 101)
y = 1 + x + x * np.random.random(len(x))
```

In [9]:
```
def func(x, a, b):
    y = a*x + b
    return y

alpha = optimize.curve_fit(func, xdata = x, ydata = y)[0]
print(alpha)
```

```
[1.61564576 0.96182858]
```

LSR for Nonlinear Functions

- A **least squares regression** requires that the **estimation function** be a **linear combination** of **basis** **functions**.
- There are some functions that cannot be put in this form, but where a least squares regression is **still appropriate**.
- Introduced later are several ways to deal with **nonlinear** **functions**.
  - We can accomplish this by taking advantage of the properties of **logarithms**, and **transform** the **non-linear** function into a **linear** function.
  - We can use the **curve_fit** function from **scipy** to estimate directly the parameters for the **non-linear** function using least square.

Log Tricks for Exponential Functions

- Assume you have a function in the form $\hat{y}(x) = \alpha e^{\beta x}$ and data for x and y, and that you want to perform **least squares regression** to find α and β.
- Clearly, the previous set of basis functions (**linear**) would be inappropriate to describe $\hat{y}(x)$; however, if we take the **log** of both sides, we get $\log(\hat{y}(x)) = \log(\alpha) + \beta x$.
- Now, say that $\tilde{y}(x) = \log(\hat{y}(x))$ and $\tilde{\alpha} = \log(\alpha)$, then $\tilde{y}(x) = \tilde{\alpha} + \beta x$.
- Now, we can perform a **least squares regression** on the **linearized** expression to find $\tilde{y}(x)$, $\tilde{\alpha}$, and β, and then recover α by using the expression $\alpha = e^{\tilde{\alpha}}$.

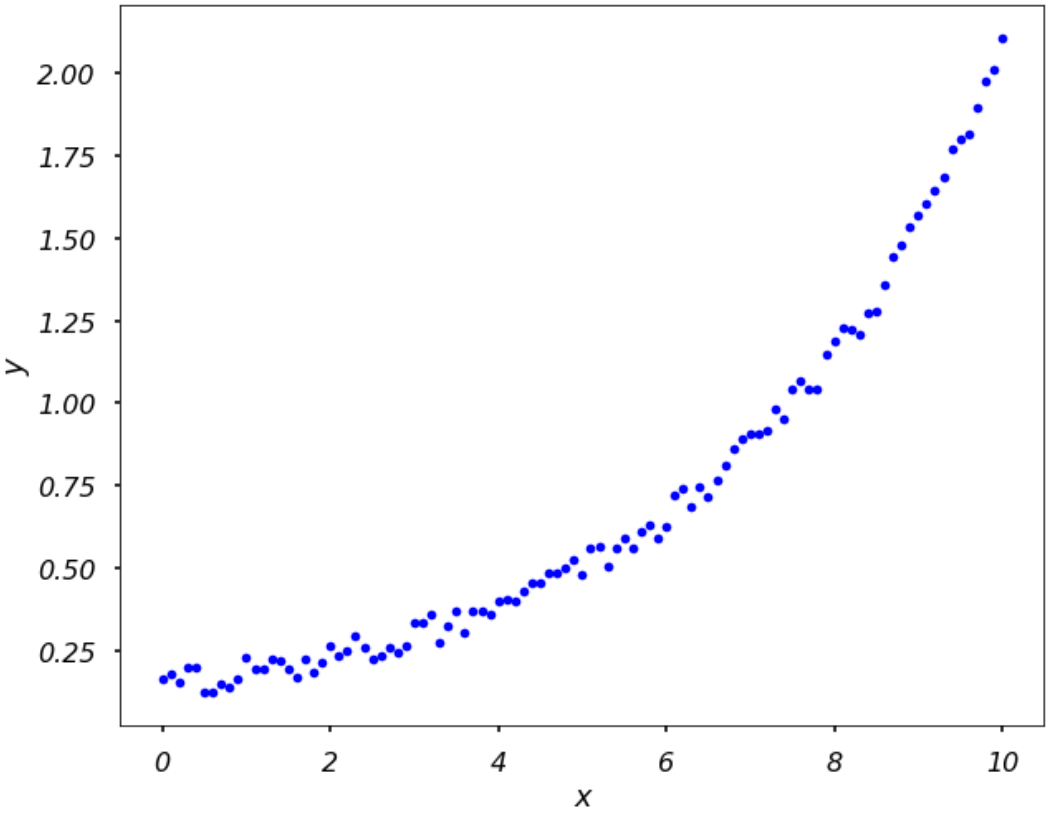
Log Tricks for Exponential Functions

- For the example below, we will generate data using $\alpha = 0.1$ and $\beta = 0.3$.

In [13]:
```
import numpy as np
from scipy import optimize
import matplotlib.pyplot as plt

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

In [14]:
```
# Let's generate x and y, and add some noise into y
x = np.linspace(0, 10, 101)
y = 0.1*np.exp(0.3*x) + 0.1*np.random.random(len(x))
```

In [15]:
```
# Let's have a look of the data
plt.figure(figsize = (10,8))
plt.plot(x, y, 'b.')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



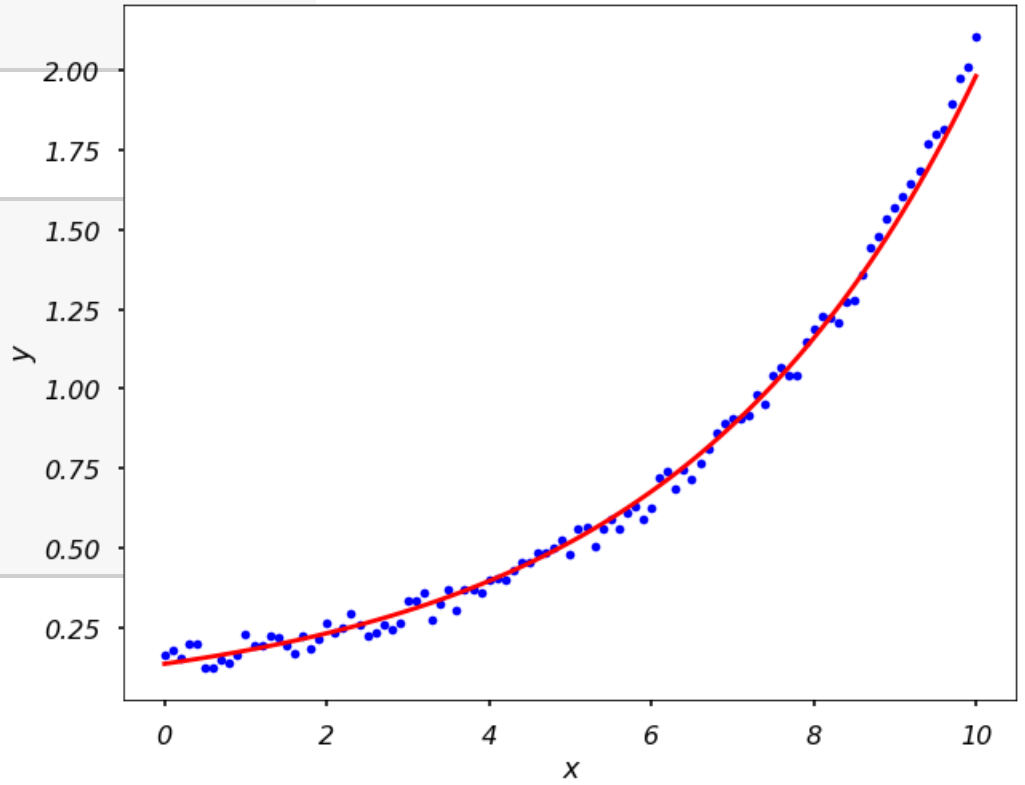
Log Tricks for Exponential Functions

- Let's fit the data after we applied the **log trick**.

In [16]:
```
A = np.vstack([x, np.ones(len(x))]).T
beta, log_alpha = np.linalg.lstsq(A, np.log(y), rcond = None)[0]
alpha = np.exp(log_alpha)
print(f'alpha={alpha}, beta={beta}')
```

```
alpha=0.13504743256802007, beta=0.2686771402218146
```

In [17]:
```
# Let's have a look of the data
plt.figure(figsize = (10,8))
plt.plot(x, y, 'b.')
plt.plot(x, alpha*np.exp(beta*x), 'r')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Log Tricks for Power Functions

- The **power function** case is very similar.
- Assume we have a function in the form $\hat{y}(x) = bx^m$ and data for x and y.
- Then we can turn this function into a **linear** form by taking **log** to both sides

$$\log(\hat{y}(x)) = m\log(x) + \log b\,.$$

- Therefore, we can solve this function as a **linear regression**.
- Since it is very similar to the previous example, we will not spend more time on this.

Polynomial Regression

- We can also use **polynomial** and **least squares** to fit a **nonlinear** function.
- Previously, we have our functions all in **linear** form, that is, $y = ax + b$.
- But **polynomials** are functions with the following form:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x^1 + a_0$$

  where $a_n, a_{n-1}, \ldots, a_2, a_1, a_0$ are the **real number coefficients**, and n, a nonnegative integer, is the **order** or **degree** of the polynomial.

- If we have a set of data points, we can use **different order of polynomials** to fit it.
- The coefficients of the polynomials can be estimated using the least squares method as before, that is, **minimizing the error** between the real data and the polynomial fitting results.

Polynomial Regression

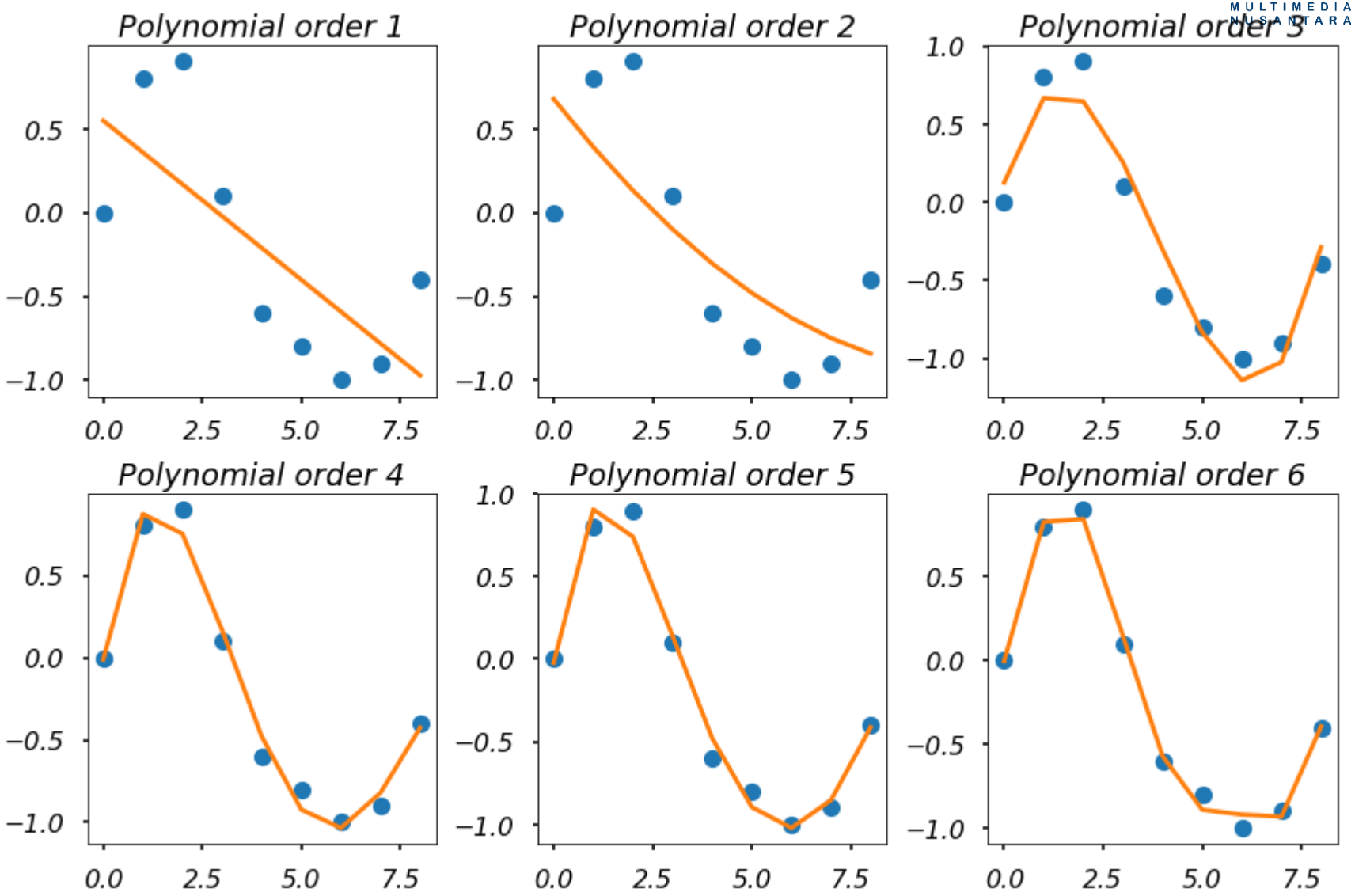
- In Python, we can use **numpy.polyfit** to obtain the **coefficients** of different order polynomials with the least squares.
- With the coefficients, we then can use **numpy.polyval** to get **specific values** for the given **coefficients**.
- Let us see an example how to perform this in Python.

In [18]:

```
x_d = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
y_d = np.array([0, 0.8, 0.9, 0.1, -0.6, -0.8, -1, -0.9, -0.4])

plt.figure(figsize = (12, 8))
for i in range(1, 7):

    # get the polynomial coefficients
    y_est = np.polyfit(x_d, y_d, i)
    plt.subplot(2,3,i)
    plt.plot(x_d, y_d, 'o')
    # evaluate the values for a polynomial
    plt.plot(x_d, np.polyval(y_est, x_d))
    plt.title(f'Polynomial order {i}')

plt.tight_layout()
plt.show()
```

Polynomial Regression

- The figure shows that we can use different order of polynomials to fit the same data.
- The **higher** the **order**, the curve we used to fit the data will be **more flexible to fit** the data. But what order to use is not a simple question, it depends on the specific problems in science and engineering.



Use optimize.curve_fit from Scipy

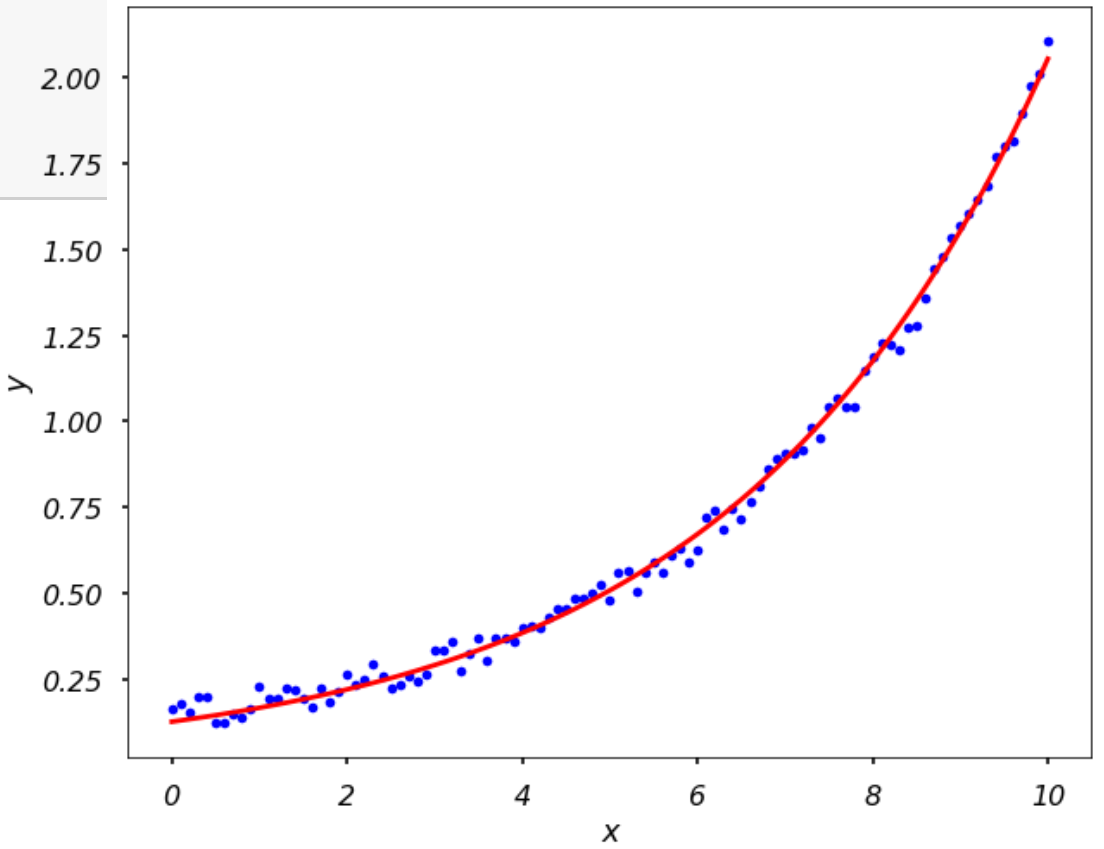
- We can use the **curve_fit** function to fit any form function and estimate the parameters of it.
- Here is how we solve the above problem in the **log** tricks section using the **curve_fit** function.

```
In [19]:
# Let's define the function form
def func(x, a, b):
    y = a*np.exp(b*x)
    return y

alpha, beta = optimize.curve_fit(func, xdata = x, ydata = y)[0]
print(f'alpha={alpha}, beta={beta}')
```

```
alpha=0.1254246477296184, beta=0.2796455030268306
```

Use optimize.curve_fit from Scipy

In [20]:
```
# Let's have a look of the data
plt.figure(figsize = (10,8))
plt.plot(x, y, 'b.')
plt.plot(x, alpha*np.exp(beta*x), 'r')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Practice

1. Write a function **my_ls_params(f, x, y)**, where **x** and **y** are arrays of the same size containing experimental data, and **f** is a list with each element a function object to a basis vector of the estimation function. The output argument, **beta**, should be an array of the parameters of the least squares regression for **x**, **y**, and **f**.

```
# Test case
x = np.linspace(0, 1, 101)
y = 1 + x + x * np.random.random(len(x))
```

```
beta = my_ls_params(func, x, y)
print(beta)
```

```
[[1.31459484]
 [1.05853804]]
```

Next Week's Outline

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

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.