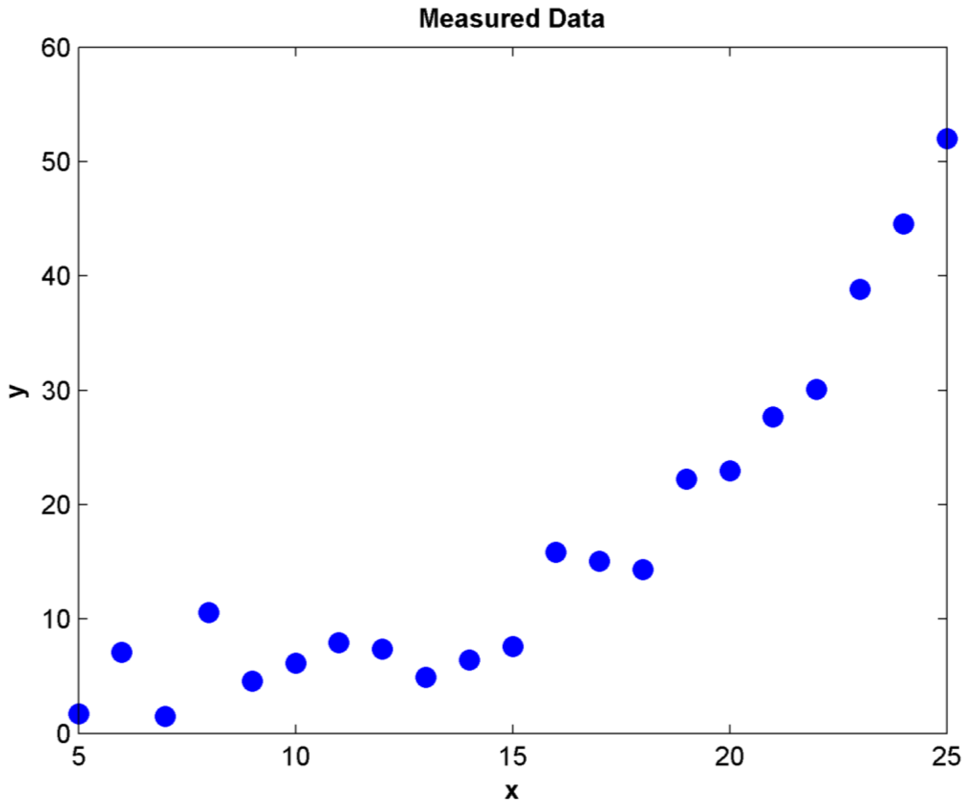
SECTION 4: CURVE FITTING

ESC 440 – Computational Methods for Engineers

Introduction

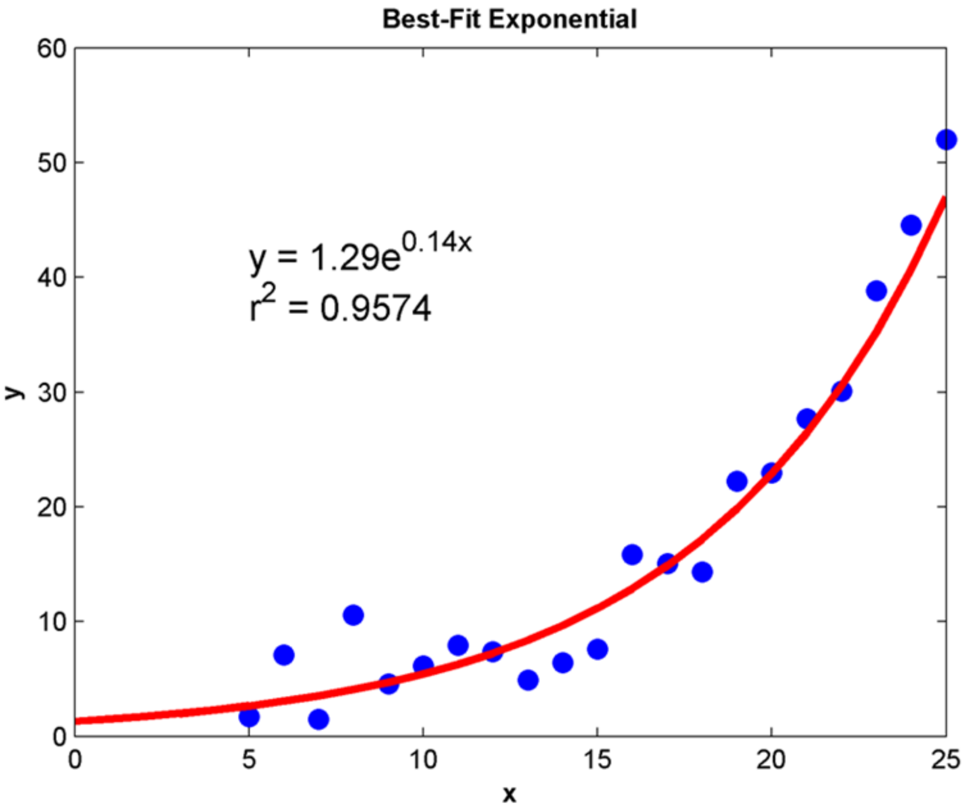
Curve Fitting

- Often, we have data, y, that is a function of some independent variable, x,
  - Possibly noisy measurement data
- Underlying relationship is unknown
  - Know x's and y's (approximately)
  - But, don't know $y = f(x)$



Curve Fitting

- May want to determine a function (i.e., a curve) that 'best' describes relationship between x and y
  - An approximation to (the unknown) $y = f(x)$
  - This is ***curve fitting***



Regression vs. Interpolation

We'll look at two categories of curve fitting:

- ***Least-squares regression***
  - Noisy data – uncertainty in y value for a given x value
  - Want "good" agreement between $f(x)$ and data points
    - Curve (i.e., $f(x)$) may not pass through any data points
- ***Polynomial interpolation***
  - Data points are known exactly – noiseless data
  - Resulting curve passes through all data points

Review of Basic Statistics

Before moving on to discuss least-squares regression, we'll first review a few basic concepts from statistics.

Basic Statistical Quantities

- ***Arithmetic mean*** – the average or expected value

$$\bar{y} = \frac{\sum y_i}{n}$$

- ***Standard deviation*** (unbiased) – a measure of the *spread* of the data about the mean

$$\sigma = \sqrt{\frac{S_t}{n-1}}$$

where S_t is the ***total sum of the squares of the residuals***

$$S_t = \sum (y_i - \bar{y})^2$$

Basic Statistical Quantities

- ***Variance*** – another measure of spread
  - The square of the standard deviation
  - Useful measure due to relationship with power and power spectral density of a signal or data set

$$\sigma^2 = \frac{S_t}{n-1} = \frac{\sum(y_i - \bar{y})^2}{n-1}$$

or

$$\sigma^2 = \frac{\sum y_i{}^2 - \frac{(\sum y_i)^2}{n}}{n-1}$$

Normal (Gaussian) Distribution

- Many naturally-occurring random process are normally-distributed
  - Measurement noise
  - Very often assume noise in our data is Gaussian
  - Probability density function (pdf):

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where σ^2 is the variance, and μ is the mean of the random variable, x

Random Number Generation – `default_rng()`

- Very often useful to generate ***random numbers***
  - Simulating the effect of noise
  - Monte Carlo simulation, etc.
- First, construct a random-number generator object using NumPy:

```
rng = np.random.default_rng(seed)
```

  - `seed`: *optional* initialization seed for generator
  - `rng`: initialized generator object – will run methods on this object to generate random numbers

Normally-Distributed Random Numbers

- Generate random values from a normal (Gaussian) distribution

```
x = rng.normal(loc=0, scale=1, size=1)
```

  - `rng`: generator object created with `default_rng()`
  - `loc`: *optional* mean of distribution – default: 0.0
  - `scale`: *optional* standard deviation – default: 1.0
  - `size`: *optional* dimension of resulting array
  - `x`: resulting array of random values

Uniformly-Distributed Random Numbers

- Generate random values from a uniform distribution on the interval `[low, high)`

```
x = rng.uniform(low=0, high=1, size=1)
```

  - `rng`: generator object created with `default_rng()`
  - `low`: *optional* lower bound of interval – default: 0.0
  - `high`: *optional* upper bound of interval – default: 1.0
  - `size`: *optional* dimension of resulting array – default: 1
  - `x`: resulting array of random values

- Half-open interval:
  - Resulting values are $\geq$ low and $<$ high

NumPy Statistical Functions

- NumPy includes many statistical functions, including:
    - np.max()
    - np.min()
    - np.mean()
    - np.std()
    - np.median()
    - np.var()
    - np.cov()

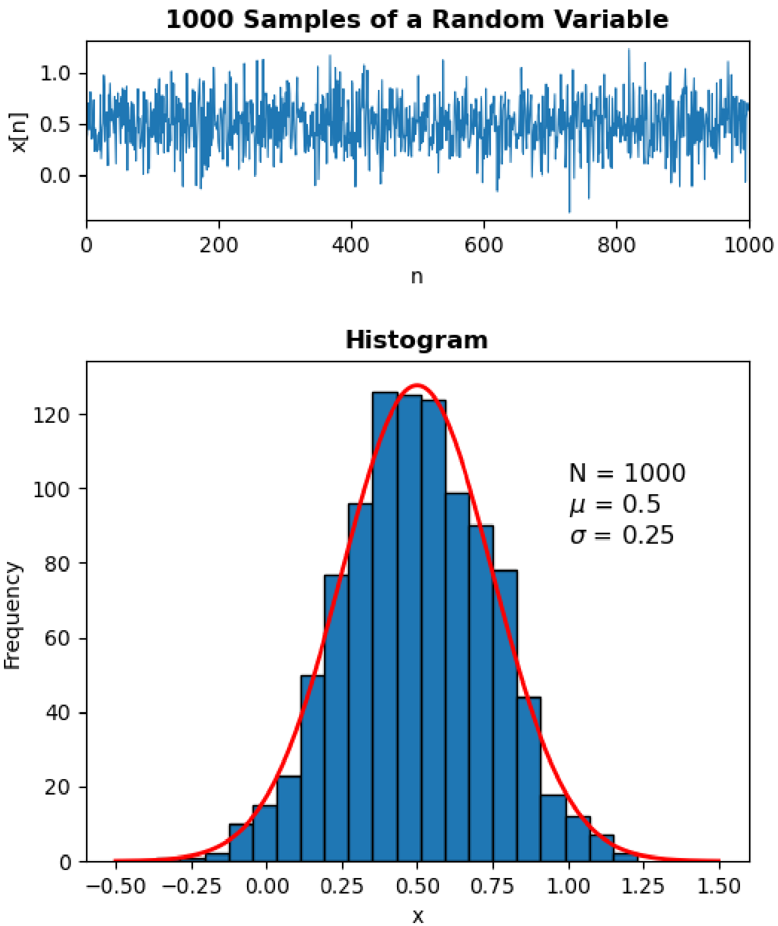
Histogram Plots

- ***Histogram plots***
  - Graphical depiction of the variation of random quantities
    - Plots the frequency of occurrence of ranges (bins) of values
  - Provides insight into the nature of the distribution

```
plt.hist(x, bins=20, edgecolor='k')
```

  - `x`: data to be histogrammed
  - `bins`: *optional* number of bins
  - `edgecolor`: *optional* color of bin outlines – default: none

Statistics in NumPy, matplotlib




```python
# GaussianDemo.m

import numpy as np
from matplotlib import pyplot as plt

u = 0.5
s = 0.25
x = np.arange(-0.5, 1.5, 1e-3)
f = 1/np.sqrt(2*np.pi*s**2)*np.exp(-((x-u)**2)/(2*s**2))

N = 1000

rng = np.random.default_rng()
y = rng.normal(loc=u, scale=s, size=N)

plt.figure(1); plt.clf()
plt.subplot(311)
plt.plot(y, linewidth=0.75)
plt.xlim(0, N)
plt.xlabel('n'); plt.ylabel('x[n]')
plt.title(f'{N} Samples of a Random Variable', fontweight='bold')

plt.subplot(3,1,(2,3))
plt.hist(y, bins=20, edgecolor='k')
plt.plot(x,80*f, '-r', linewidth=2)
plt.xlabel('x'); plt.ylabel('Frequency')
plt.title('Histogram', fontweight='bold')
plt.text(1,85, f'N = {N}\n$\mu$ = {u}\n$\sigma$ = {s}', fontsize=
```

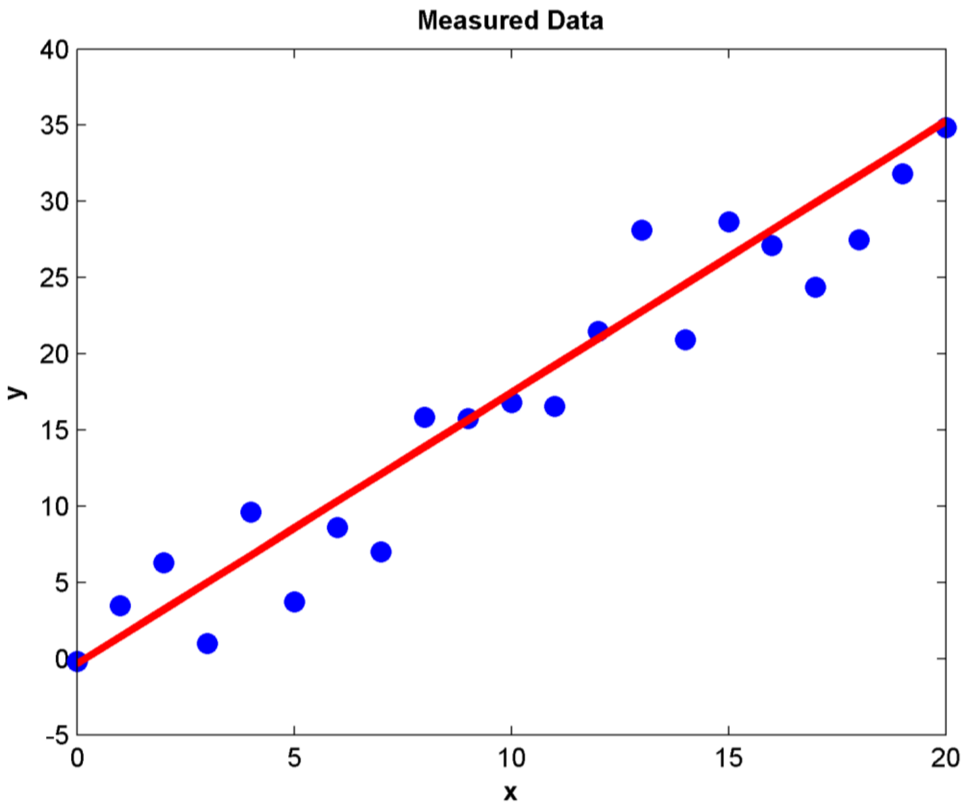
Linear Least-Squares Regression

Linear Regression

- Noisy data, y, values at known x values
- Suspect relationship between x and y is ***linear***
- i.e., assume

$$y = a_0 + a_1 x$$

- Determine a_0 and a_1 that define the "***best-fit***" line for the data



- How do we define the "***best fit***"?

Measured Data

- Assumed a linear relationship between x and y:

$$y = a_0 + a_1 x$$

- ***Due to noise, can't measure y exactly at each x***
  - Can only approximate y values

$$\hat{y} = y + e$$

- ***Measured values are approximations***
  - True value of y plus some random error or ***residual***

$$\hat{y} = a_0 + a_1 x + e$$

Best Fit Criteria

- Noisy data do not all line on a single line – discrepancy between each point and the line fit to the data
  - The error, or ***residual***:

$$e = \hat{y} - a_0 - a_1 x$$

- Minimize some measure of this residual:
  - Minimize the ***sum of the residuals***
    - Positive and negative errors can cancel
    - Non-unique fit
  - Minimize the ***sum of the absolute values of the residuals***
    - Effect of sign of error eliminated, but still not a unique fit
  - Minimize the maximum error – ***minimax criterion***
    - Excessive influence given to single outlying points

Least-Squares Criterion

- Better fitting criterion is to minimize the ***sum of the squares of the residuals***

$$S_r = \sum e_i^2 = \sum (\hat{y}_i - a_0 - a_1 x_i)^2$$

  - Yields a unique best-fit line for a given set of data
- The sum of the squares of the residuals is a function of the two fitting parameters, a_0 and a_1, $S_r(a_0, a_1)$
- Minimize S_r by setting its partial derivatives to zero and solving for a_0 and a_1

Least-Squares Criterion

- At its minimum point, partial derivatives of S_r with respect to a_0 and a_1 will be zero

$$\frac{\partial S_r}{\partial a_0} = -2\sum(\hat{y}_i - a_0 - a_1 x_i) = 0$$

$$\frac{\partial S_r}{\partial a_1} = -2\sum[(\hat{y}_i - a_0 - a_1 x_i)x_i] = 0$$

- Breaking up the summation:

$$\sum \hat{y}_i - \sum a_0 - \sum a_1 x_i = 0$$

$$\sum x_i \hat{y}_i - \sum a_0 x_i - \sum a_1 x_i^2 = 0$$

Normal Equations

- $\partial S_r/\partial a_0 = 0$ and $\partial S_r/\partial a_1 = 0$ form a system of two equations with two unknowns, a_0 and a_1

$$n \;\; a_0 + \left(\sum x_i\right) a_1 = \sum \hat{y}_i \qquad (1)$$

$$\left(\sum x_i\right) a_0 + \left(\sum x_i^2\right) a_1 = \sum x_i \hat{y}_i \qquad (2)$$

- In matrix form:

$$\begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum \hat{y}_i \\ \sum x_i \hat{y}_i \end{bmatrix} \qquad (3)$$

- These are the ***normal equations***

Normal Equations

- Normal equations can be solved for a_0 and a_1:

$$a_1 = \frac{n \sum x_i \hat{y}_i - \sum x_i \sum \hat{y}_i}{n \sum x_i^2 - \left(\sum x_i\right)^2}$$

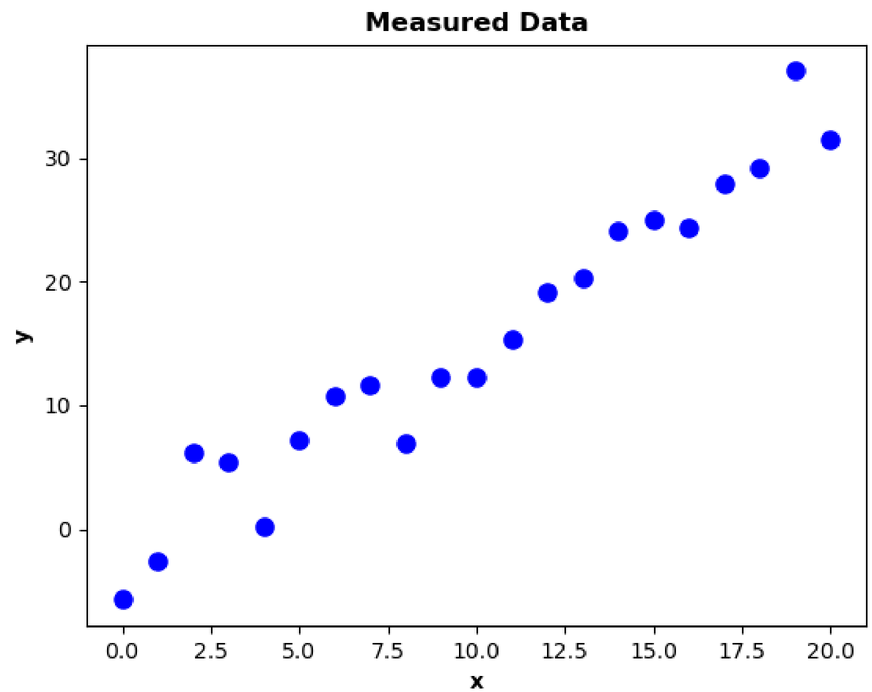
$$a_0 = \frac{\sum \hat{y}_i - a_1 \sum x_i}{n} = \bar{y} - a_1 \bar{x}$$

- Or solve the matrix form of the normal equations, (3), in Python using `np.linalg.solve()`

Linear Least-Squares - Example

- Noisy data with suspected linear relationship
- Calculate summation terms in the normal equations:
  - $n, \Sigma x_i, \Sigma \hat{y}_i, \Sigma x_i^2, \Sigma x_i \hat{y}_i$

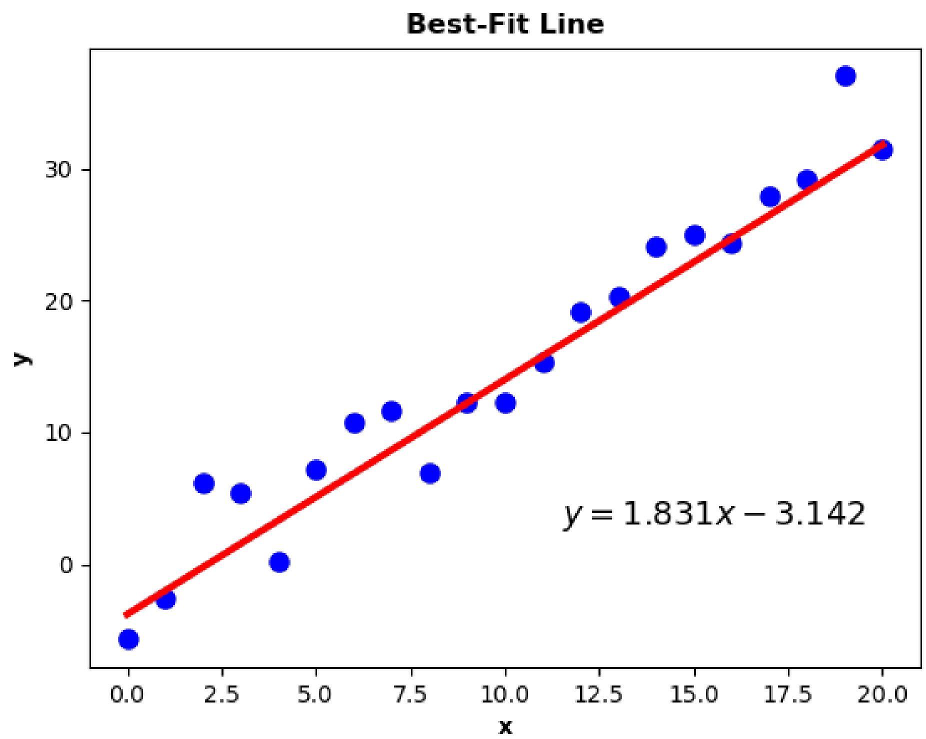
```
n = len(yn)
Sx = sum(x)
Sy = sum(yn)
Sxy = sum(x*yn)
Sx2 = sum(x**2)
```



Linear Least-Squares - Example

- Assemble normal equation matrices
- Solve normal equations for vector of coefficients, **a**, using `np.linalg.solve()`

```python
n = len(yn)
Sx = sum(x)
Sy = sum(yn)
Sxy = sum(x*yn)
Sx2 = sum(x**2)

Z = np.array([[n, Sx], [Sx, Sx2]])
b = np.array([Sy, Sxy])
a = np.linalg.solve(Z, b)

# %% the best-fit line
y1 = a[1]*x + a[0]
```




```
In [141]: a
Out[141]: array([-3.14223852,  1.83066766])

In [142]:
```

Goodness of Fit

- How well does a function fit the data?
- Is a linear fit best? A quadratic, higher-order polynomial, or other non-linear function?
- Want a way to be able to quantify ***goodness of fit***

- Quantify spread of data about the mean prior to regression:

$$S_t = \sum (\hat{y}_i - \bar{y})^2$$

- Following regression, quantify spread of data about the regression line (or curve):

$$S_r = \sum (\hat{y}_i - a_0 - a_1 x_i)^2$$

Goodness of Fit

- S_t quantifies the spread of the data about the mean
- S_r quantifies spread about the best-fit line (curve)
  - The spread that remains after the trend is explained
  - The ***unexplained sum of the squares***
- $S_t - S_r$ represents the reduction in data spread after regression explains the underlying trend
- Normalize to S_t - the ***coefficient of determination***

$$r^2 = \frac{S_t - S_r}{S_t}$$

Coefficient of Determination

$$r^2 = \frac{S_t - S_r}{S_t}$$

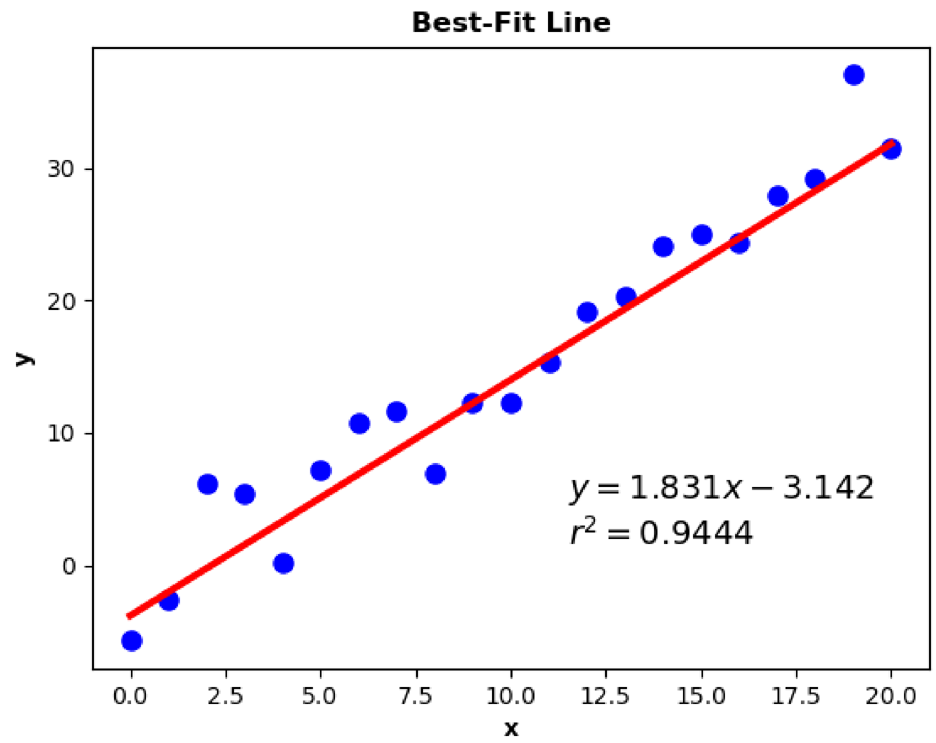
- For a perfect fit:
  - No variation in data about the regression line
  - $S_r = 0 \quad \rightarrow \quad r^2 = 1$
- If the fit provides no improvement over simply characterizing data by its mean value:
  - $S_r = S_t \quad \rightarrow \quad r^2 = 0$
- If the fit is worse at explaining the data than their mean value:
  - $S_r > S_t \quad \rightarrow \quad r^2 < 0$

Coefficient of Determination

- Calculate r^2 for previous example:

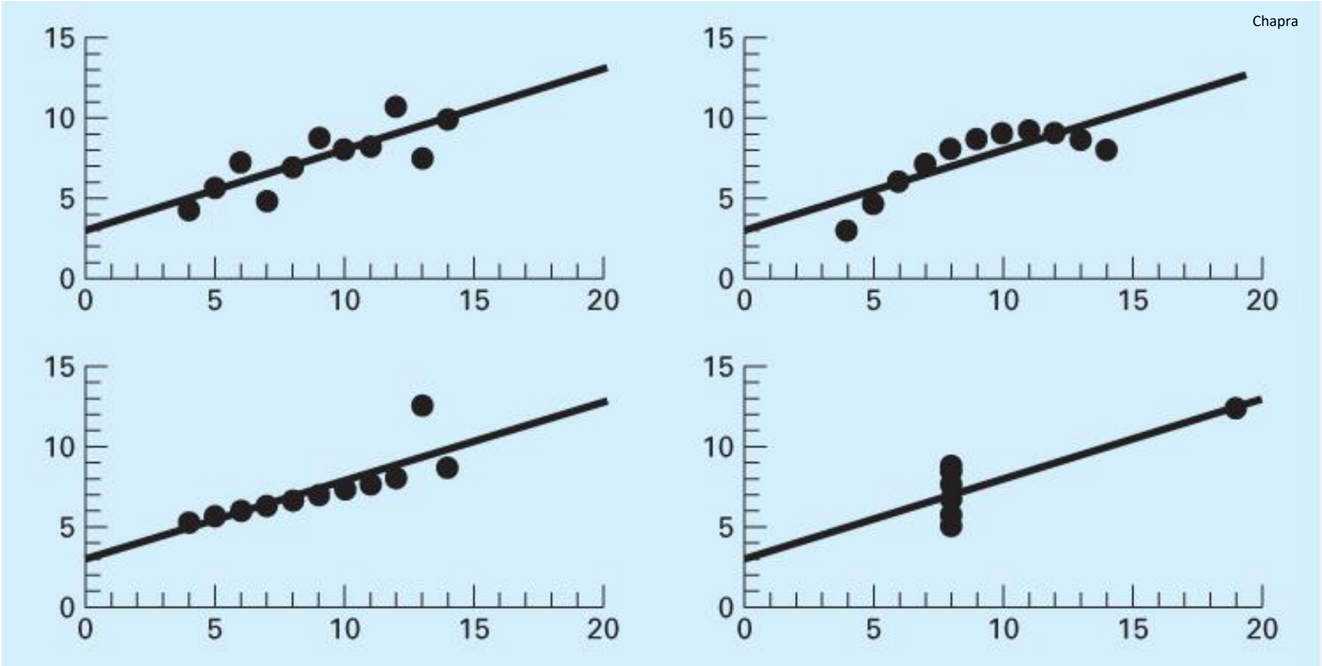
```python
# calculate the coefficient
# of determination
ybar = np.mean(yn)
St = sum((yn - ybar)**2)
Sr = sum((yn - y1)**2)
r2 = (St - Sr)/St
```

```
In [142]: r2
Out[142]: 0.9444329681572226

In [143]:
```



Coefficient of Determination

- Don't rely too heavily on the value of r^2
- Anscombe's famous data sets:



- Same line fit to all four data sets
- $r^2 = 0.67$ in each case

Linearization of Nonlinear Relationships

Nonlinear functions

- Not all data can be explained by a linear relationship to an independent variable, e.g.
  - ***Exponential model***

$$y = \alpha e^{\beta x}$$

  - ***Power equation***

$$y = \alpha x^{\beta}$$

  - ***Saturation-growth-rate equation***

$$y = \alpha \frac{x}{\beta + x}$$

Nonlinear functions

Methods for nonlinear curve fitting:

- ***Linearization of the nonlinear relationship***
  - Transform the dependent and/or independent data values
  - Apply linear least-squares regression
  - Inverse transform the determined coefficients back to those that define the nonlinear functional relationship
- ***Nonlinear regression***
  - Treat as an optimization problem – more later…

Linearizing an Exponential Relationship

- Have noisy data that is believed to be best described by an ***exponential relationship***

$$y = \alpha e^{\beta x}$$

- ***Linearize the fitting equation***:

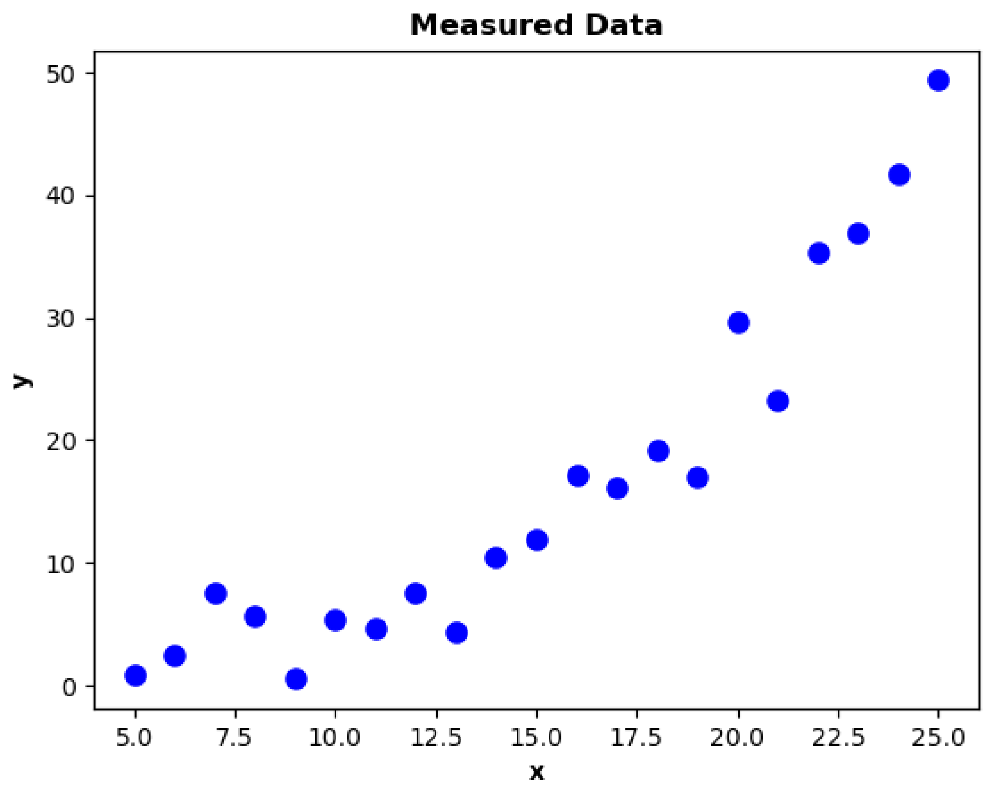
$$\ln(y) = \ln(\alpha) + \beta x$$

or

$$\ln(y) = a_0 + a_1 x$$

where

$$a_0 = \ln(\alpha),\ \ a_1 = \beta$$

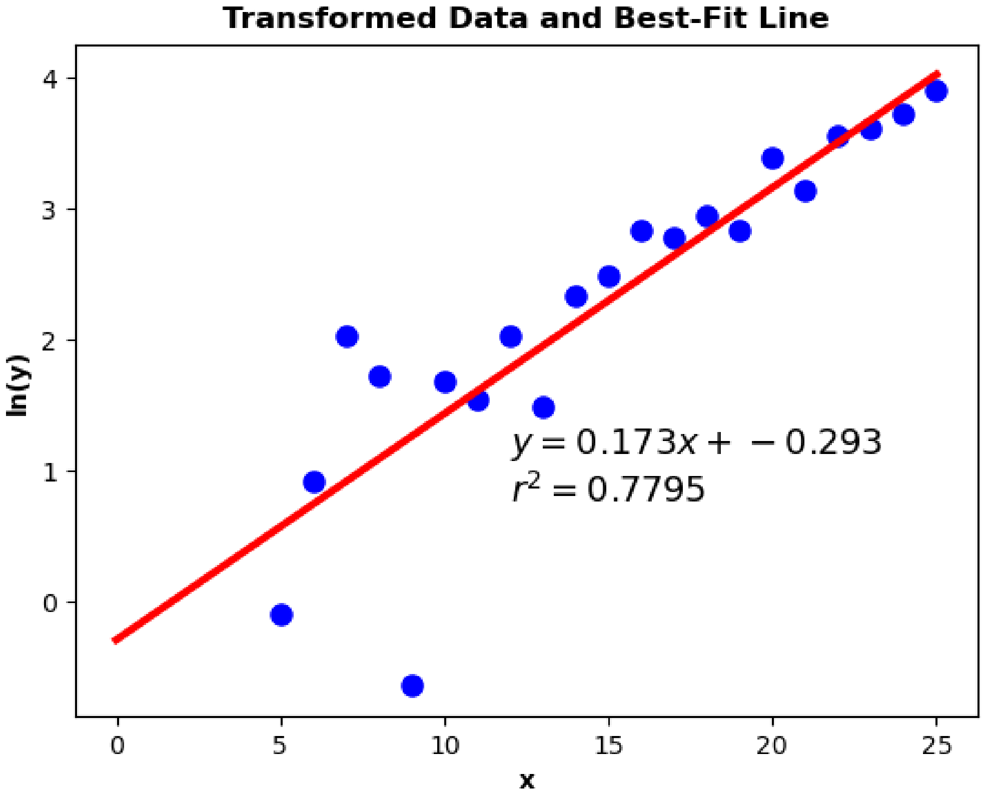


Linearizing an Exponential Relationship

- Fit a line to the transformed data using linear least-squares regression
- Determine a_0 and a_1:

$$\ln(y) = a_0 + a_1 x$$

- Can calculate r^2 for the line fit to the transformed data
- Note that original data must be positive



Linearizing an Exponential Relationship

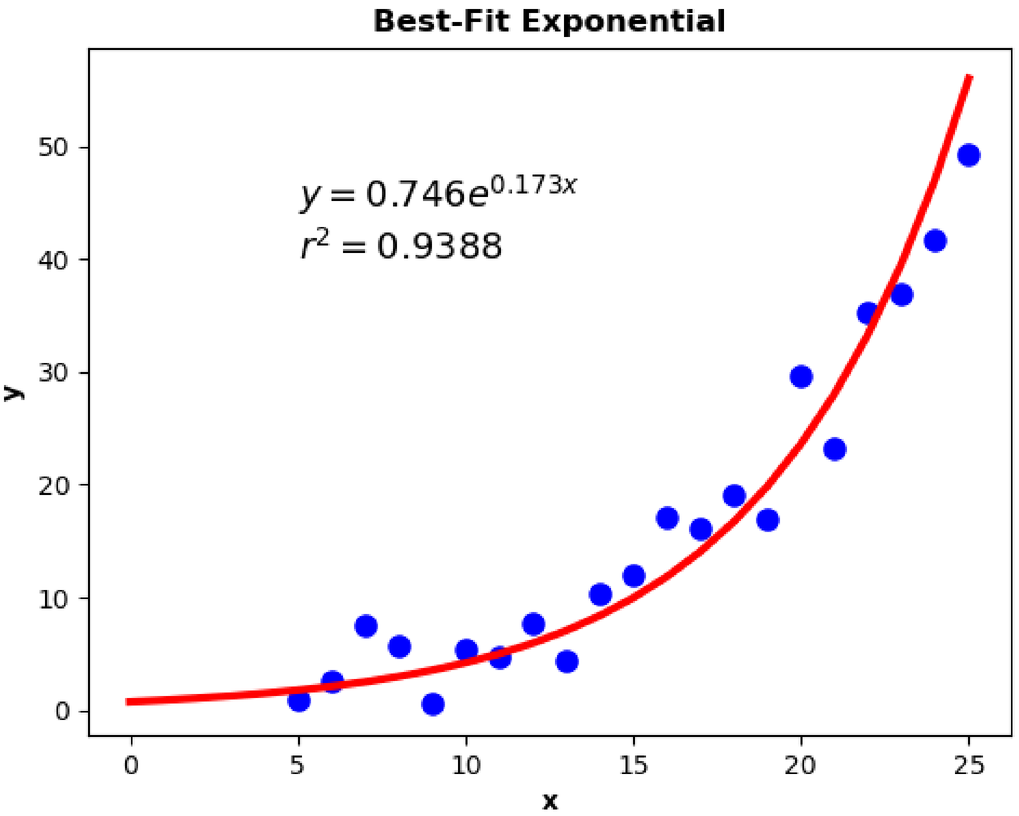
- Transform the linear fitting parameters, a_0 and a_1, back to the parameters defining the exponential relationship
- Exponential fit:

$$y = \alpha e^{\beta x}$$

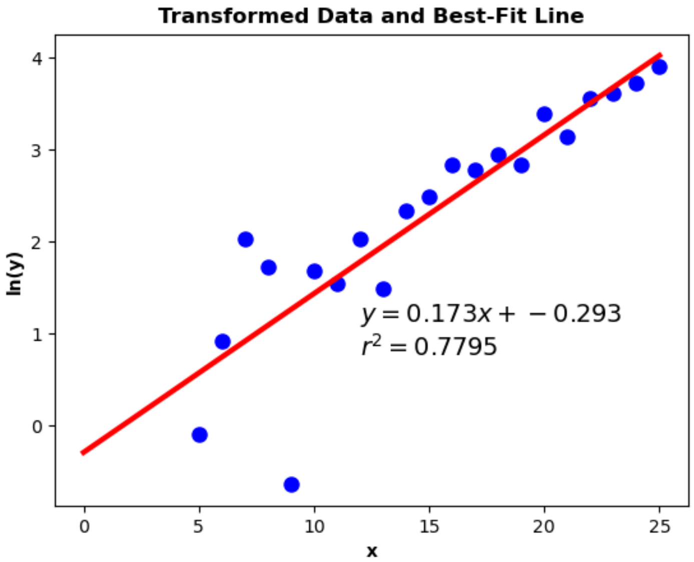
where

$$\alpha = e^{a_0}, \quad \beta = a_1$$

- Note that r^2 is different than that for the line fit to the transformed data

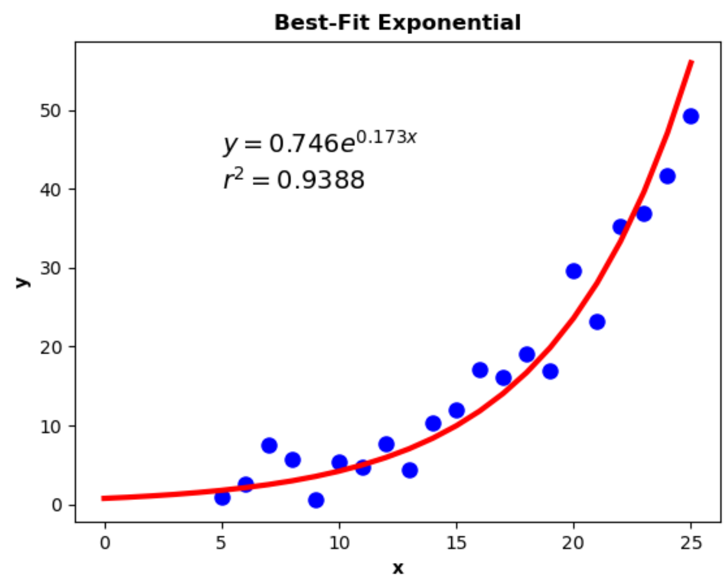


Linearizing an Exponential Relationship




```python
# %% transform the data vector, yn
lny = np.log(yn)

# %% solve normal equations for
# the transformed data set

n = len(yn)
Sx = sum(x)
Sy = sum(lny)
Sxy = sum(x*lny)
Sx2 = sum(x**2)

Z = np.array([[n, Sx], [Sx, Sx2]])
b = np.array([Sy, Sxy])
a = np.linalg.solve(Z, b)
```

```python
# %% inverse transform the linear
# fit coefficients to get the
# parameters for the exponential
alpha = np.exp(a[0])
beta = a[1]

# the exponential fit
yexp = alpha*np.exp(beta*xfit)

# calculate the coefficient
# of determination for exp fit
ybar = np.mean(yn)
St = sum((yn - ybar)**2)
Sr = sum((yn - yexp[-len(x):])**2)
r2 = (St - Sr)/St
```



Linearizing a Power Equation

- Have noisy data that is believed to be best described by an ***power equation***

$$y = \alpha x^{\beta}$$

- ***Linearize the fitting equation***:

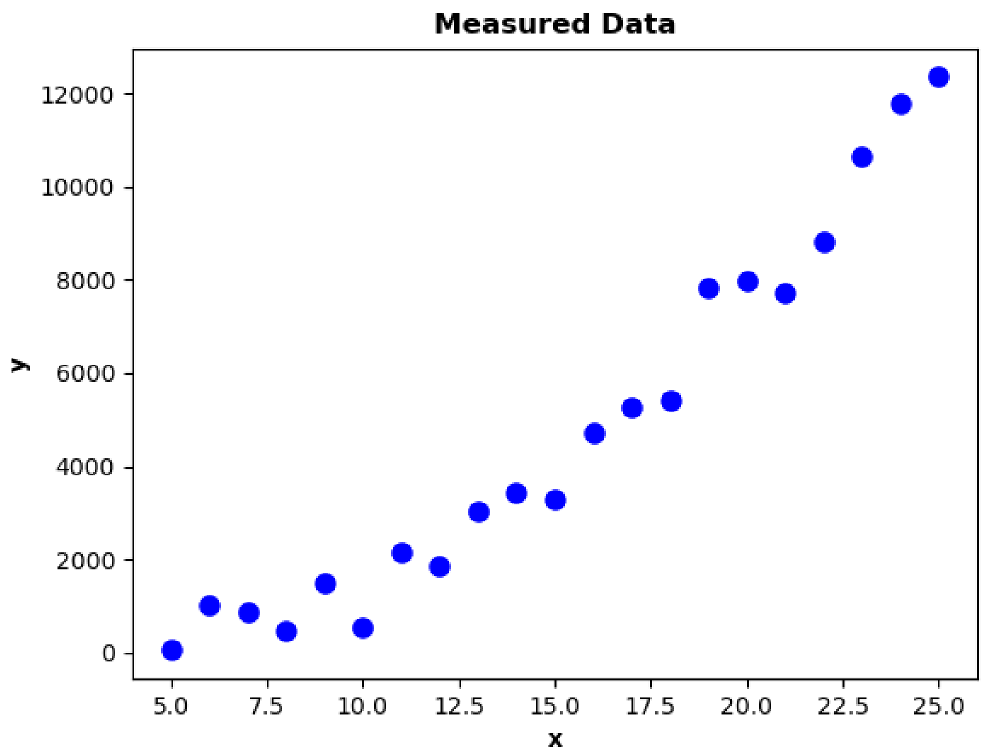
$$\log(y) = \log(\alpha) + \beta \log(x)$$

or

$$\log(y) = a_0 + a_1 \log(x)$$

where

$$a_0 = \log(\alpha), \;\; a_1 = \beta$$

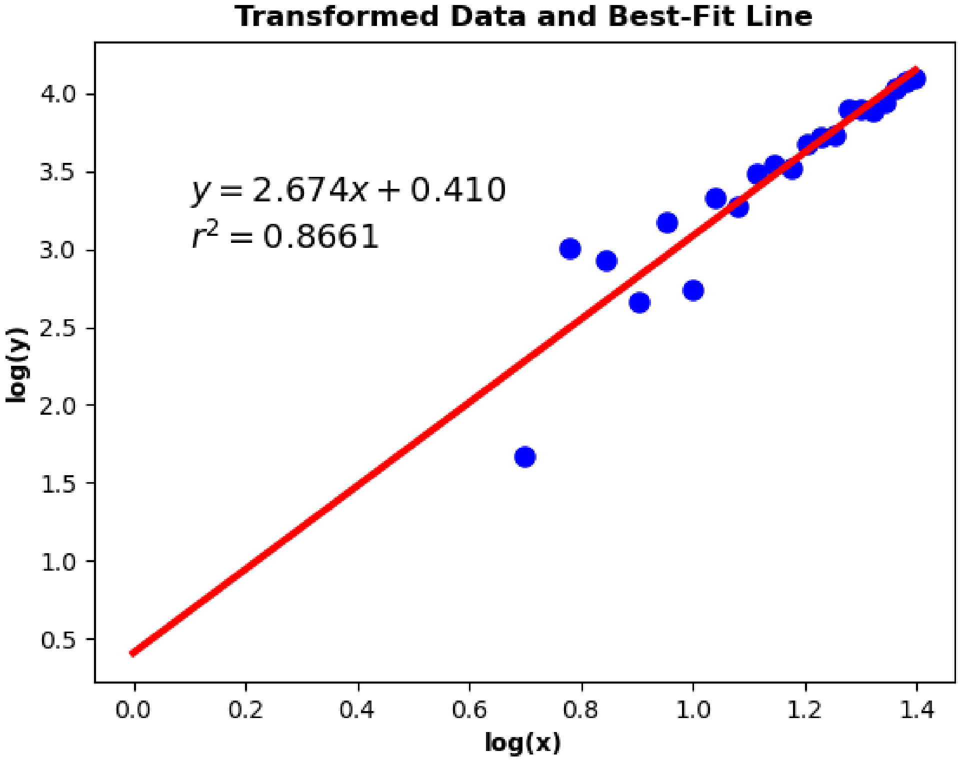


Linearizing a Power Equation

- Fit a line to the transformed data using linear least-squares regression
- Determine a_0 and a_1:

$$\log(y) = a_0 + a_1 \log(x)$$

- Can calculate r^2 for the line fit to the transformed data
- Note that original data – both x and y – must be positive



Linearizing a Power Equation

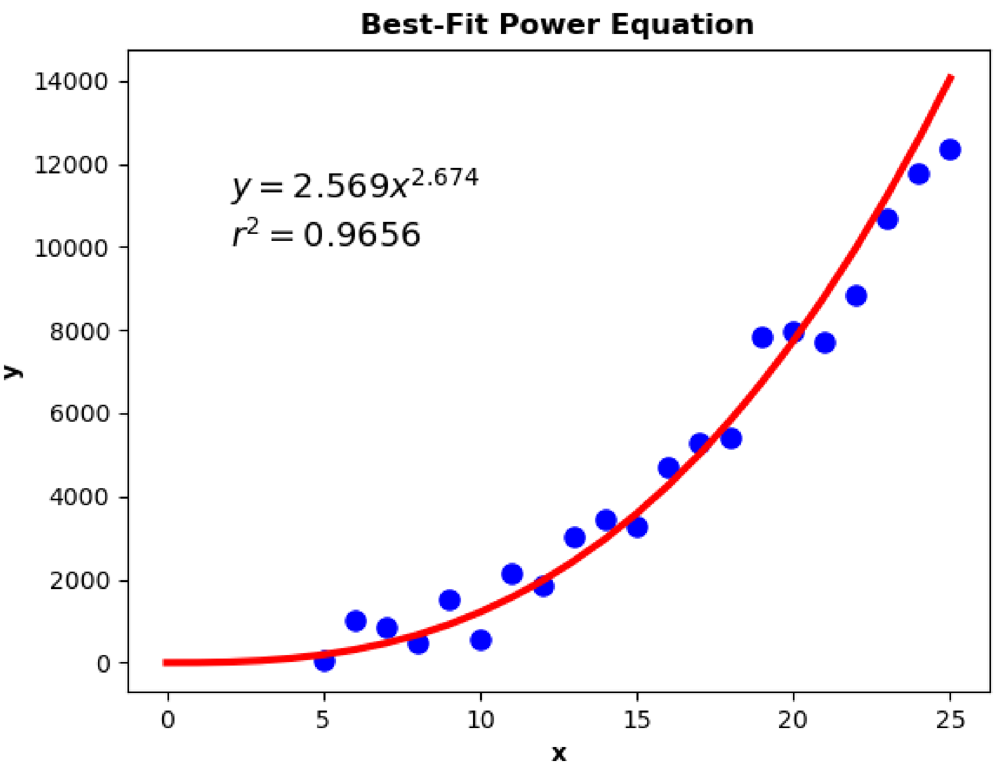
- Transform the linear fitting parameters, a_0 and a_1, back to the parameters defining the power equation
- Power equation:

$$y = \alpha x^{\beta}$$

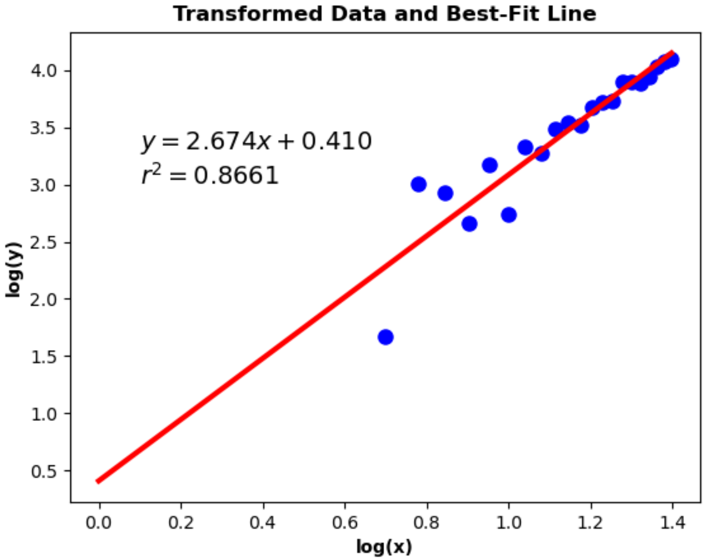
where

$$\alpha = 10^{a_0}\,, \quad \beta = a_1$$

- Note that r^2 is different than that for the line fit to the transformed data

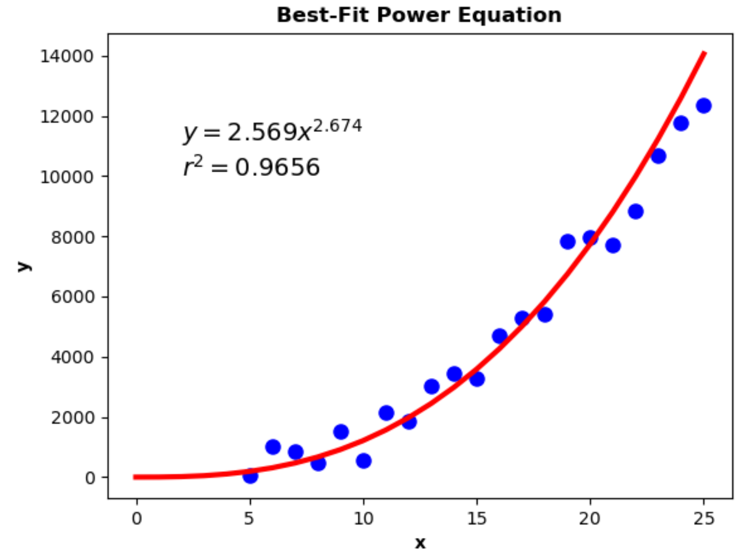


Linearizing a Power Equation




```python
# %% transform the data vectors, yn and x
logy = np.log10(yn)
logx = np.log10(x)

# %% solve normal equations for
# the transformed data set
n = len(yn)
Sx = sum(logx)
Sy = sum(logy)
Sxy = sum(logx*logy)
Sx2 = sum(logx**2)

Z = np.array([[n, Sx], [Sx, Sx2]])
b = np.array([Sy, Sxy])
a = np.linalg.solve(Z, b)
```

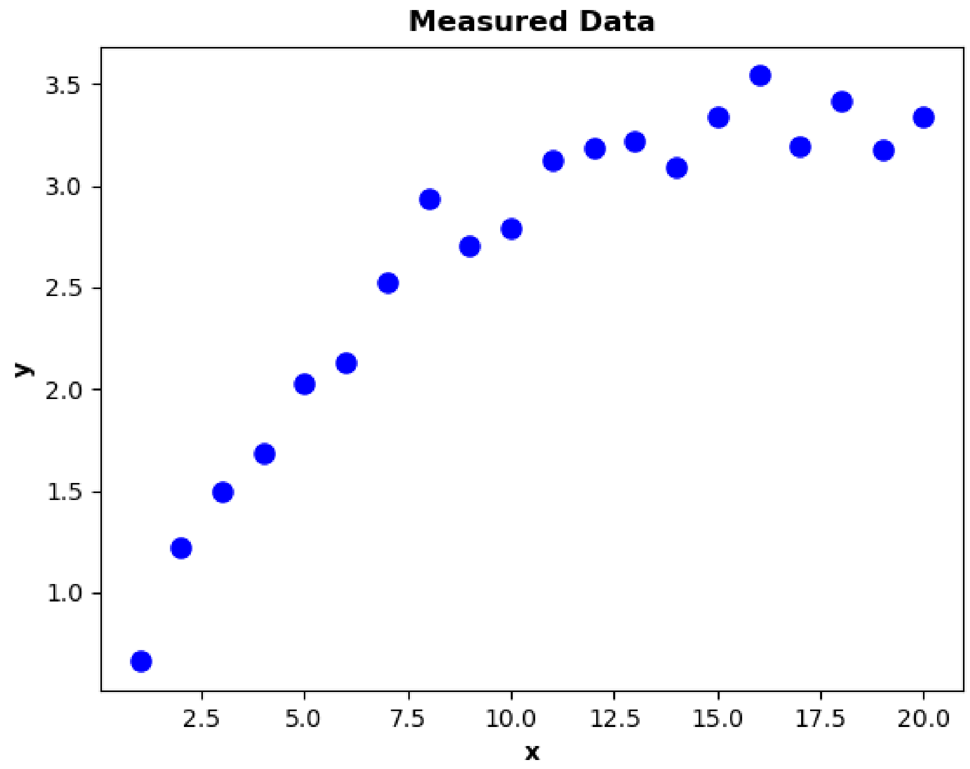
```python
# %% inverse transform the linear
# fit coefficients to get the
# parameters for the power equation
alpha = 10**(a[0])
beta = a[1]

# the power equation fit
xpow = np.arange(max(x)+1)
ypow = alpha*xpow**beta
ypowr2 = alpha*x**beta

# calculate the coefficient of
# determination for power eqn. fit
ybar = np.mean(yn)
St = sum((yn - ybar)**2)
Sr = sum((yn - ypowr2)**2)
r2 = (St - Sr)/St
```



Linearizing a Saturation Growth-Rate Equation

- Have noisy data that is believed to be best described by a ***saturation growth-rate equation***

$$y = \alpha \frac{x}{\beta + x}$$



- ***Linearize the fitting equation***:

$$\frac{1}{y} = \frac{1}{\alpha} + \frac{\beta}{\alpha}\frac{1}{x}$$

or

$$\frac{1}{y} = a_0 + a_1 \frac{1}{x}$$

where

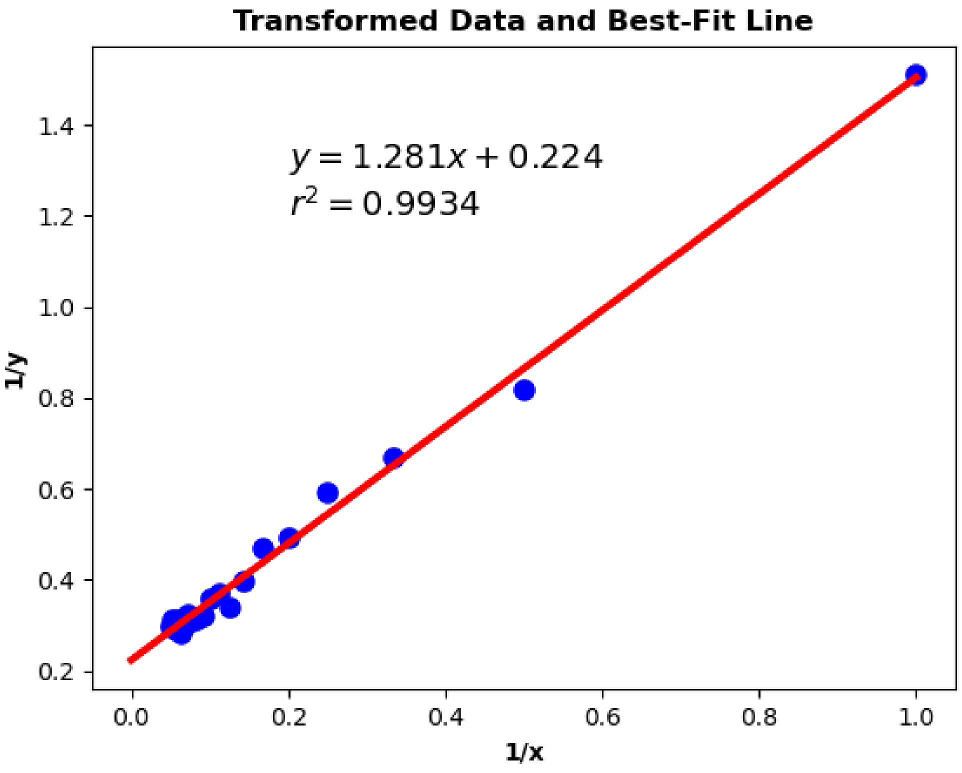
$$a_0 = \frac{1}{\alpha}, \quad a_1 = \frac{\beta}{\alpha}$$

Linearizing a Saturation Growth-Rate Equation

- Fit a line to the transformed data using linear least-squares regression
- Determine a_0 and a_1:

$$\frac{1}{y} = a_0 + a_1 \frac{1}{x}$$

- Can calculate r^2 for the line fit to the transformed data



Linearizing a Saturation Growth-Rate Equation

- Transform the linear fitting parameters, a_0 and a_1, back to the parameters defining the saturation growth-rate equation

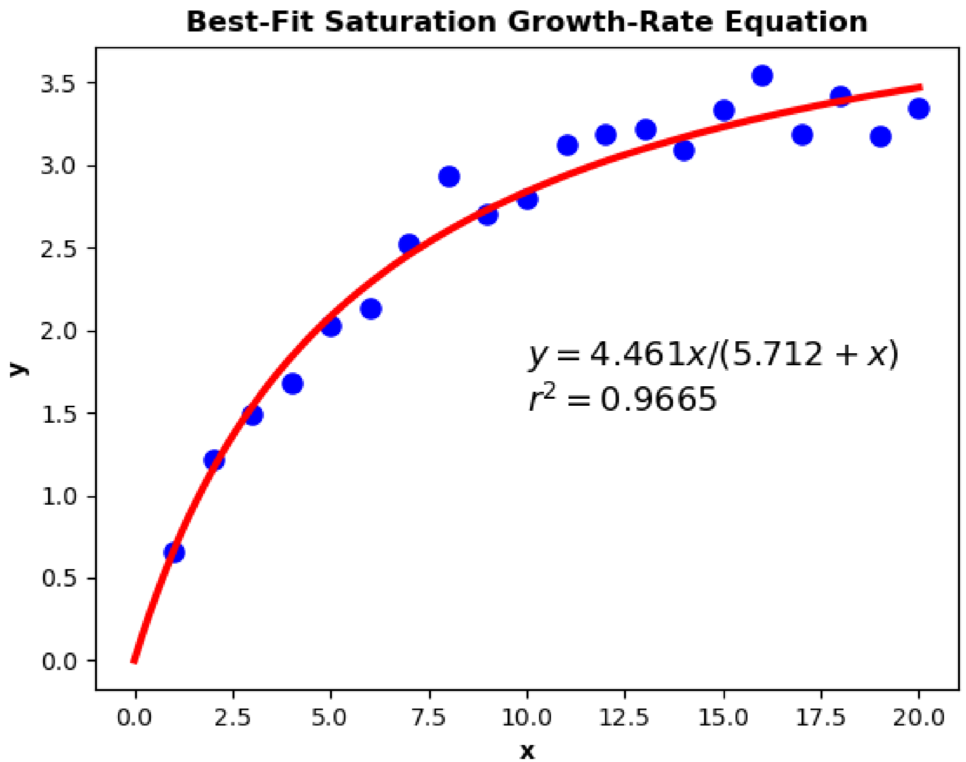
- Saturation growth-rate equation:

$$y = \alpha \frac{x}{\beta + x}$$

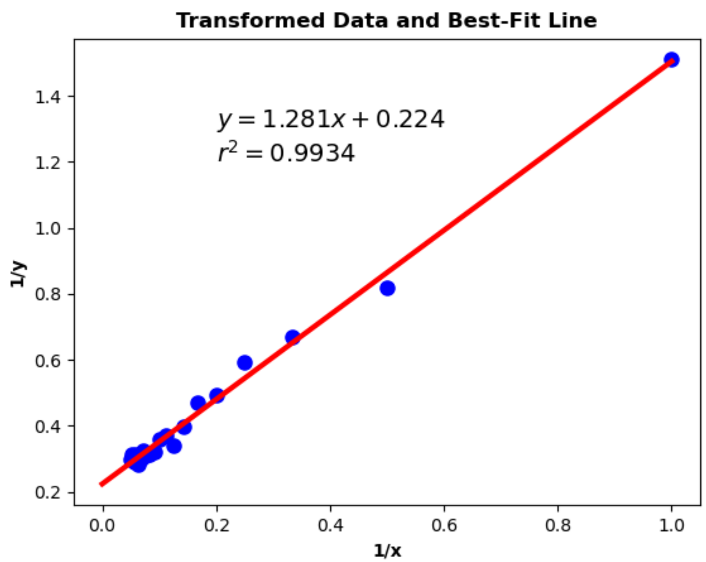
where

$$\alpha = \frac{1}{a_0}, \quad \beta = \frac{a_1}{a_0}$$

- Note that r^2 is different than that for the line fit to the transformed data

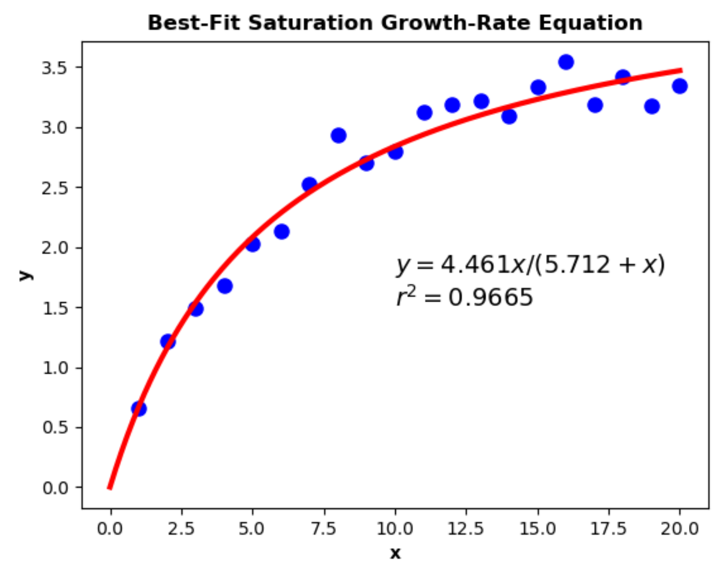


Linearizing a Saturation Growth-Rate Equation




```
# %% transform the data vectors, yn and x
invy = 1/yn
invx = 1/x

# %% solve normal equations for
# the transformed data set
n = len(yn)
Sx = sum(invx)
Sy = sum(invy)
Sxy = sum(invx*invy)
Sx2 = sum(invx**2)

Z = np.array([[n, Sx], [Sx, Sx2]])
b = np.array([Sy, Sxy])
a = np.linalg.solve(Z, b)
```

```
# %% inverse transform the linear
# fit coefficients to get the
# parameters for the sgr equation
alpha = 1/a[0]
beta = a[1]/a[0]

# the saturation growth-rate equation fit
xsgr = np.linspace(0,max(x),200)
ysgr = alpha*xsgr/(beta+xsgr)
ysgrr2 = alpha*x/(beta+x)

# calculate the coefficient
# of determination for sgr fit
ybar = np.mean(yn)
St = sum((yn - ybar)**2)
Sr = sum((yn - ysgrr2)**2)
r2 = (St - Sr)/St
```



Polynomial Regression

Polynomial Regression

- So far we've looked at fitting straight lines to ***linear*** and ***linearized*** data sets
- Can also fit ***m^{th}-order polynomials*** directly to data using ***polynomial regression***

- Same fitting criterion as linear regression:
  - Minimize the sum of the squares of the residuals
    - m+1 fitting parameters for an m^{th}-order polynomial
    - m+1 normal equations

Polynomial Regression

- Assume, for example, that we have data we believe to be ***quadratic*** in nature
- ***2nd-order polynomial regression***
- Fitting equation:

$$\hat{y} = a_0 + a_1 x + a_2 x^2 + e$$

- Best fit will minimize the sum of the squares of the residuals:

$$S_r = \sum \left(\hat{y}_i - a_0 - a_1 x_i - a_2 x_i^2 \right)^2$$

Polynomial Regression – Normal Equations

- Best-fit polynomial coefficients will minimize S_r
  - Differentiate S_r w.r.t. each coefficient and set to zero

$$\frac{\partial S_r}{\partial a_0} = -2\sum\left(\hat{y}_i - a_0 - a_1 x_i - a_2 x_i^2\right) = 0$$

$$\frac{\partial S_r}{\partial a_1} = -2\sum x_i\left(\hat{y}_i - a_0 - a_1 x_i - a_2 x_i^2\right) = 0$$

$$\frac{\partial S_r}{\partial a_2} = -2\sum x_i^2\left(\hat{y}_i - a_0 - a_1 x_i - a_2 x_i^2\right) = 0$$

Polynomial Regression – Normal Equations

- Rearranging the normal equations yields

$$
\begin{aligned}
n\,a_0 + (\Sigma x_i)a_1 + \left(\Sigma x_i^2\right)a_2 &= \Sigma \hat{y}_i \\
(\Sigma x_i)a_0 + \left(\Sigma x_i^2\right)a_1 + \left(\Sigma x_i^3\right)a_2 &= \Sigma x_i \hat{y}_i \\
\left(\Sigma x_i^2\right)a_0 + \left(\Sigma x_i^3\right)a_1 + \left(\Sigma x_i^4\right)a_2 &= \Sigma x_i^2 \hat{y}_i
\end{aligned}
$$

- Which can be put into matrix form:

$$
\begin{bmatrix} n & \Sigma x_i & \Sigma x_i^2 \\ \Sigma x_i & \Sigma x_i^2 & \Sigma x_i^3 \\ \Sigma x_i^2 & \Sigma x_i^3 & \Sigma x_i^4 \end{bmatrix}
\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}
=
\begin{bmatrix} \Sigma \hat{y}_i \\ \Sigma x_i \hat{y}_i \\ \Sigma x_i^2 \hat{y}_i \end{bmatrix}
$$

- This system of equations can be solved for the vector of unknown coefficients using NumPy's `linalg.solve()`

Polynomial Regression – Normal Equations

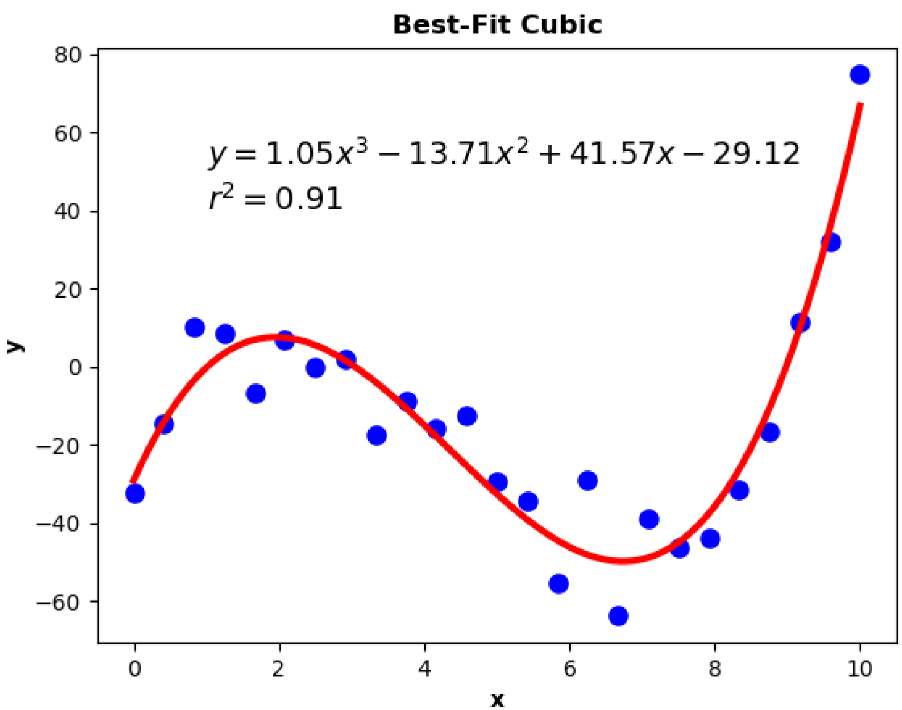
- For ***mth-order*** polynomial regression the ***normal equations*** are:

$$\begin{bmatrix} n & \Sigma x_i & \cdots & \Sigma x_i^m \\ \Sigma x_i & \Sigma x_i^2 & \cdots & \Sigma x_i^{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma x_i^m & \Sigma x_i^{m+1} & \cdots & \Sigma x_i^{2m} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} \Sigma \hat{y}_i \\ \Sigma x_i \hat{y}_i \\ \vdots \\ \Sigma x_i^m \hat{y}_i \end{bmatrix}$$

- Again, this system of $m+1$ equations can be solved for the vector of $m+1$ unknown polynomial coefficients using NumPy’s `linalg.solve()`

Polynomial Regression – Example


```python
# %% noiseless data
p = np.poly([1,3,9])
x = np.linspace(0,10,25)
y = np.polyval(p,x)

# %% add noise to y data
sig = 8

# set the random number generator seed
seed = 4

rng = np.random.default_rng(seed)
v = rng.normal(scale=sig, size=len(x))
yn = y + v
```




```python
# %% Construct and solve the
# normal equations
n = len(yn)
Sx = sum(x)
Sx2 = sum(x**2)
Sx3 = sum(x**3)
Sx4 = sum(x**4)
Sx5 = sum(x**5)
Sx6 = sum(x**6)
Sy = sum(yn)
Sxy = sum(x*yn)
Sx2y = sum(x**2*yn)
Sx3y = sum(x**3*yn)

Z = [[n,   Sx, Sx2, Sx3],
     [Sx,  Sx2, Sx3, Sx4],
     [Sx2, Sx3, Sx4, Sx5],
     [Sx3, Sx4, Sx5, Sx6]]
b = [Sy, Sxy, Sx2y, Sx3y]
a = np.linalg.solve(Z,b)

# %% reverse the order of coefficients to
# conform to NumPy's convention
pfit = a[::-1]
# or
pfit = np.flip(a)

# np.polyfit will give the same answer
# pfit = np.polyfit(x,yn,3)

# %% evaluate the best-fit cubic
xfit = np.linspace(min(x),max(x),200)
y3 = np.polyval(pfit,xfit)
y3r2 = np.polyval(pfit,x)

# %% calculate the coefficient
# of determination for the fit
ybar = np.mean(yn)
St = sum((yn - ybar)**2)
Sr = sum((yn - y3r2)**2)
r2 = (St - Sr)/St
```

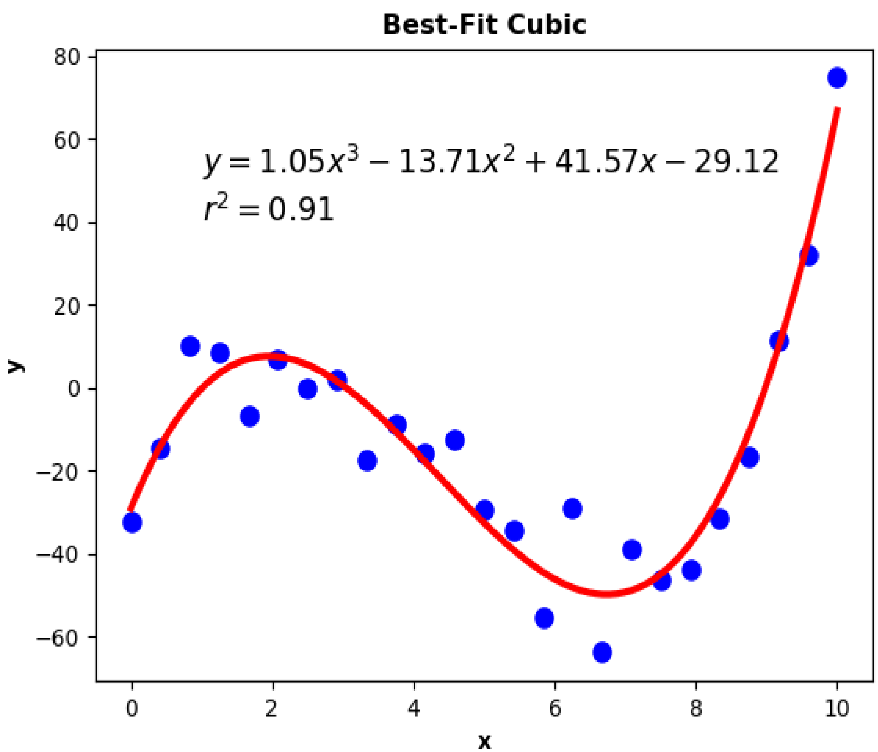
Polynomial Regression – `np.polyfit()`


```
p = np.polyfit(x,y,m)
```

- `x`: n-vector of independent variable data values
- `y`: n-vector of dependent variable data values
- `m`: order of the polynomial to be fit to the data
- `p`: $(m+1)$-vector of best-fit polynomial coefficients

- Least-squares polynomial regression if:
  - $n > m + 1$
  - i.e., for over-determined systems
- Polynomial interpolation if:
  - $n = m + 1$
  - Resulting fit passes through all (x,y) points – more later

Polynomial Regression – np.polyfit()



- Note that the result matches that obtained by solving normal equations

```python
# polyfit3.py

import numpy as np
from matplotlib import pyplot as plt


# %% noiseless data
p = np.poly([1,3,9])
x = np.linspace(0,10,25)
y = np.polyval(p,x)

# %% add noise to y data
sig = 8

# set the random number generator seed
seed = 4

rng = np.random.default_rng(seed)
v = rng.normal(scale=sig, size=len(x))
yn = y + v

# %% use np.polyfit to perform the regression
pfit = np.polyfit(x,yn,3)

# %% evaluate the best-fit cubic
xfit = np.linspace(min(x),max(x),200)
y3 = np.polyval(pfit,xfit)
y3r2 = np.polyval(pfit,x)

# %% calculate the coefficient
# of determination for the fit
ybar = np.mean(yn)
St = sum((yn - ybar)**2)
Sr = sum((yn - y3r2)**2)
r2 = (St - Sr)/St
```

Polynomial Regression Using `np.polyfit()`

Exercise

- Determine the 4th-order polynomial with roots at $x = \{1, 5, 16, 19\}$
- Generate noiseless data points by evaluating this polynomial at integer values of x from 0 to 20
- Add Gaussian white noise with a standard deviation of $\sigma = 180$ to your data points
- Use `np.polyfit()` to fit a 4th-order polynomial to the noisy data
- Calculate the coefficient of determination, r^2
- Plot the noisy data points, along with the best-fit polynomial

Multiple Linear Regression

Multiple Linear Regression

- We have so far fit lines or curves to data described by functions of a single variable
- For functions of multiple variables, ***fit planes or surfaces to data***
- Linear function of two independent variables: ***multiple linear regression***

$$\hat{y} = a_0 + a_1 x_1 + a_2 x_2 + e$$

- Sum of the squares of the residuals is now

$$S_r = \sum \left(\hat{y}_i - a_0 - a_1 x_{1,i} - a_2 x_{2,i} \right)^2$$

Multiple Linear Regression – Normal Equations

- Differentiate S_r w.r.t. fitting coefficients and equate to zero
- The ***normal equations***:

$$\begin{bmatrix} n & \Sigma x_{1,i} & \Sigma x_{2,i} \\ \Sigma x_{1,i} & \Sigma x_{1,i}^2 & \Sigma x_{1,i}x_{2,i} \\ \Sigma x_{2,i} & \Sigma x_{1,i}x_{2,i} & \Sigma x_{2,i}^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \Sigma \hat{y}_i \\ \Sigma x_{1,i}\hat{y}_i \\ \Sigma x_{2,i}\hat{y}_i \end{bmatrix}$$

- Solve as before – now fitting coefficients, a_i, define a ***plane***

General Linear Least-Squares Regression

General Linear Least-Squares

- We've seen three types of least-squares regression
  - ***Linear regression***
  - ***Polynomial regression***
  - ***Multiple linear regression***
- All are special cases of ***general linear least-squares regression***

$$\hat{y} = a_0 z_0 + a_1 z_1 + \cdots + a_m z_m + e$$

- The z_i's are $m + 1$ ***basis functions***
  - Basis functions may be nonlinear
  - This is ***linear*** regression, because dependence on fitting coefficients, a_i, is linear

General Linear Least-Squares

$$\hat{y} = a_0 z_0 + a_1 z_1 + \cdots + a_m z_m + e$$

- For ***linear regression*** – simple or multiple:

$$z_0 = 1,\ z_1 = x_1,\ z_2 = x_2,\ \ldots\ z_m = x_m$$

- For ***polynomial regression***:

$$z_0 = 1,\ z_1 = x,\ z_2 = x^2,\ \ldots\ z_m = x^m$$

- In all cases, this is a ***linear combination of basis function***, which may, themselves, be ***nonlinear***

General Linear Least-Squares

- The general linear least-squares model:

$$\hat{y} = a_0 z_0 + a_1 z_1 + \cdots + a_m z_m + e$$

- Can be expressed in matrix form:

$$\hat{\mathbf{y}} = \mathbf{Z}\,\mathbf{a} + \mathbf{e}$$

where $\mathbf{Z}$ is an $n \times (m+1)$ matrix, the ***design matrix***, whose entries are the $(m+1)$ basis functions evaluated at the n independent variable values corresponding to the n measurements:

$$\mathbf{Z} = \begin{bmatrix} z_{01} & z_{11} & \cdots & z_{m1} \\ z_{02} & z_{12} & \cdots & z_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ z_{0n} & z_{1n} & \cdots & z_{mn} \end{bmatrix}$$

where z_{ij} is the i^{th} basis function evaluated at the j^{th} independent variable value. (Note: i is not the row index and j is not the column index, here.)

General Linear Least-Squares

- The least-squares model is:

$$\begin{bmatrix} z_{01} & z_{11} & \cdots & z_{m1} \\ z_{02} & z_{12} & \cdots & z_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ z_{0n} & z_{1n} & \cdots & z_{mn} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}$$

- ***More measurements than coefficients***
  - $n > (m+1)$
  - $\mathbf{Z}$ is not square – tall and narrow
  - Over-determined system
  - $\mathbf{Z}^{-1}$ does not exist

General Linear Least-Squares – Design Matrix Example

- For example, consider fitting a quadratic to five measured values, $\hat{\mathbf{y}}$, at $\mathbf{x} = [1, 2, 3, 4, 5]^T$
- Model is:

$$\hat{y} = a_0 + a_1 x + a_2 x^2 + e$$

- Basis functions are $z_0 = 1$, $z_1 = x$, and $z_2 = x^2$
- Least-squares equation is

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \hat{y}_4 \\ \hat{y}_5 \end{bmatrix} - \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \end{bmatrix}$$

General Linear Least-Squares – Residuals

- Linear least-squares model is:

$$\hat{\mathbf{y}} = \mathbf{Z}\,\mathbf{a} + \mathbf{e} \qquad (1)$$

- Residual:

$$\mathbf{e} = \hat{\mathbf{y}} - \mathbf{y} = \hat{\mathbf{y}} - \mathbf{Z}\,\mathbf{a} \qquad (2)$$

- Sum of the squares or the residuals:

$$S_r = \sum e_i^2 = \mathbf{e}^{\mathbf{T}}\mathbf{e} = [\hat{\mathbf{y}} - \mathbf{Z}\,\mathbf{a}]^{\mathbf{T}}[\hat{\mathbf{y}} - \mathbf{Z}\,\mathbf{a}] \qquad (3)$$

- Expanding,

$$S_r = \hat{\mathbf{y}}^{\mathbf{T}}\hat{\mathbf{y}} - \mathbf{a}^{\mathbf{T}}\mathbf{Z}^{\mathbf{T}}\hat{\mathbf{y}} - \hat{\mathbf{y}}^{\mathbf{T}}\mathbf{Z}\mathbf{a} + \mathbf{a}^{\mathbf{T}}\mathbf{Z}^{\mathbf{T}}\mathbf{Z}\mathbf{a} \qquad (4)$$

Deriving the Normal Equations

- Best fit will minimize the sum of the squares of the residuals
  - Differentiate S_r with respect to the coefficient vector, $\mathbf{a}$, and set to zero

$$\frac{dS_r}{d\mathbf{a}} = \frac{d}{d\mathbf{a}}\left(\hat{\mathbf{y}}^{\mathbf{T}}\hat{\mathbf{y}} - \mathbf{a}^{\mathbf{T}}\mathbf{Z}^{\mathbf{T}}\hat{\mathbf{y}} - \hat{\mathbf{y}}^{\mathbf{T}}\mathbf{Z}\mathbf{a} + \mathbf{a}^{\mathbf{T}}\mathbf{Z}^{\mathbf{T}}\mathbf{Z}\mathbf{a}\right) = \mathbf{0} \quad (5)$$

- We'll need to use some ***matrix calculus identities:***
  - $\frac{d}{d\mathbf{a}}\left(\mathbf{a}^{\mathbf{T}}\mathbf{Z}^{\mathbf{T}}\mathbf{y}\right) = \mathbf{Z}^{\mathbf{T}}\mathbf{y}$
  - $\frac{d}{d\mathbf{a}}\left(\mathbf{y}^{\mathbf{T}}\mathbf{Z}\mathbf{a}\right) = \mathbf{Z}^{\mathbf{T}}\mathbf{y}$ (6)
  - $\frac{d}{d\mathbf{a}}\left(\mathbf{a}^{\mathbf{T}}\mathbf{Z}^{\mathbf{T}}\mathbf{Z}\mathbf{a}\right) = 2\mathbf{Z}^{\mathbf{T}}\mathbf{Z}\mathbf{a}$

Deriving the Normal Equations

$$\frac{dS_r}{d\mathbf{a}} = \frac{d}{d\mathbf{a}}\left(\hat{\mathbf{y}}^\mathrm{T}\hat{\mathbf{y}} - \mathbf{a}^\mathrm{T}\mathbf{Z}^\mathrm{T}\hat{\mathbf{y}} - \hat{\mathbf{y}}^\mathrm{T}\mathbf{Z}\mathbf{a} + \mathbf{a}^\mathrm{T}\mathbf{Z}^\mathrm{T}\mathbf{Z}\mathbf{a}\right) = \mathbf{0}$$

- Using the matrix derivative relationships, (6),

$$\frac{dS_r}{d\mathbf{a}} = -2\mathbf{Z}^\mathrm{T}\hat{\mathbf{y}} + 2\mathbf{Z}^\mathrm{T}\mathbf{Z}\mathbf{a} = \mathbf{0} \tag{7}$$

- Equation (7) is the matrix form of the ***normal equations:***

$$\mathbf{Z}^\mathrm{T}\mathbf{Z}\mathbf{a} = \mathbf{Z}^\mathrm{T}\hat{\mathbf{y}} \tag{8}$$

- Solution to (8) is the vector of least-squares fitting coefficients:

$$\mathbf{a} = \left(\mathbf{Z}^\mathrm{T}\mathbf{Z}\right)^{-\mathbf{1}}\mathbf{Z}^\mathrm{T}\hat{\mathbf{y}} \tag{9}$$

Solving the Normal Equations

$$\mathbf{a} = \left(\mathbf{Z}^{\mathrm{T}}\mathbf{Z}\right)^{-1}\mathbf{Z}^{\mathrm{T}}\hat{\mathbf{y}} \qquad (9)$$

- Remember, our starting point was the linear least-squares model:

$$\mathbf{y} = \mathbf{Z}\,\mathbf{a} \qquad (10)$$

- Couldn’t we have solved (10) for fitting coefficients as

$$\mathbf{a} = \mathbf{Z}^{-1}\mathbf{y} \qquad (11)$$

- No, must solve using (9), because:
  - Don’t have $\mathbf{y}$, only noisy approximations, $\hat{\mathbf{y}}$
  - We have an over-determined system
    - $\mathbf{Z}$ is not square
    - $\mathbf{Z}^{-1}$ does not exist

Solving the Normal Equations

- Solution to the linear least-squares problem is:

$$\mathbf{a} = \left(\mathbf{Z}^{\mathbf{T}}\mathbf{Z}\right)^{-1}\mathbf{Z}^{\mathbf{T}}\hat{\mathbf{y}} = \mathbf{Z}^{\dagger}\hat{\mathbf{y}} \tag{12}$$

where

$$\mathbf{Z}^{\dagger} = \left(\mathbf{Z}^{\mathbf{T}}\mathbf{Z}\right)^{-1}\mathbf{Z}^{\mathbf{T}} \tag{13}$$

is the ***Moore-Penrose pseudo-inverse*** of $\mathbf{Z}$

- Use the pseudo-inverse to find the least-squares solutions to an over-determined system

Coefficient of Determination

- Goodness of fit characterized by the ***coefficient of determination:***

$$r^2 = \frac{S_t - S_r}{S_t}$$

where S_r is given by (3)

$$S_r = [\hat{\mathbf{y}} - \mathbf{Z}\,\mathbf{a}]^{\mathbf{T}}[\hat{\mathbf{y}} - \mathbf{Z}\,\mathbf{a}] \qquad (14)$$

and

$$S_t = [\hat{\mathbf{y}} - \bar{\mathbf{y}}]^{\mathbf{T}}[\hat{\mathbf{y}} - \bar{\mathbf{y}}] \qquad (15)$$

General Least-Squares in Python

- Have n measurements

$$\hat{y} = [\hat{y}_0 \quad \hat{y}_1 \quad \cdots \quad \hat{y}_{n-1}]^T$$

- at n known independent variable values

$$x = [x_0 \quad x_1 \quad \cdots \quad x_{n-1}]^T$$

- and a model, defined by $m + 1$ basis functions

$$\hat{y} = a_0 z_0 + a_1 z_1 + \cdots + a_m z_m + e$$

- Generate design matrix by evaluating $m + 1$ basis functions at all n values of x

$$\mathbf{Z} = \begin{bmatrix} z_0(x_0) & z_1(x_0) & \cdots & z_m(x_0) \\ z_0(x_1) & z_1(x_1) & \cdots & z_m(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ z_0(x_{n-1}) & z_1(x_{n-1}) & \cdots & z_m(x_{n-1}) \end{bmatrix}$$

General Least-Squares in Python

- Solve for vector of fitting coefficients as the solution to the normal equations

$$\mathbf{a} = \left(\mathbf{Z}^{\mathbf{T}}\mathbf{Z}\right)^{-1}\mathbf{Z}^{\mathbf{T}}\hat{\mathbf{y}}$$

- Or by using `np.linalg.lstsq()`

```
a = np.linalg.lstsq(Z, yhat)
```

- Result is the same, though the methods are different

Nonlinear Regression

Nonlinear Regression – `minimize()`

- ***Nonlinear models:***
  - Have nonlinear dependence on fitting parameters
  - E.g., $y = \alpha x^{\beta}$
- Two options for fitting nonlinear models to data
  - Linearize the model first, then use linear regression
  - ***Fit a nonlinear model directly by treating as an optimization problem***
- Want to ***minimize*** a ***cost function***
  - Cost function is the ***sum of the squares of the residuals***

$$J = S_r = \sum (\hat{y} - y)^2$$

- Find the minimum of J – a ***multi-dimensional optimization***
  - Use SciPy's `optimize.minimize()`

Nonlinear Regression – `minimize()`

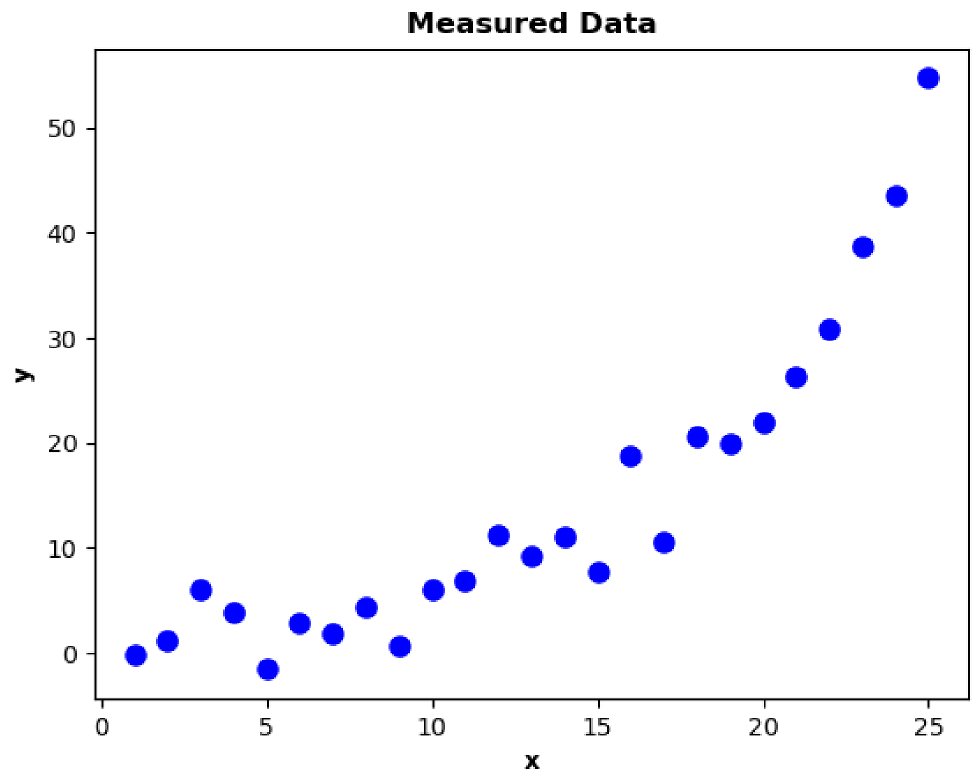
- Have noisy data that is believed to be best described by an ***exponential relationship***

$$y = \alpha e^{\beta x}$$

- ***Cost function:***

$$J = \sum \left(\hat{y} - \alpha e^{\beta x}\right)^2$$

- Find α and β to minimize J
  - Use SciPy's `optimize.minimize()`



Multi-Dimensional Optimization – `minimize()`

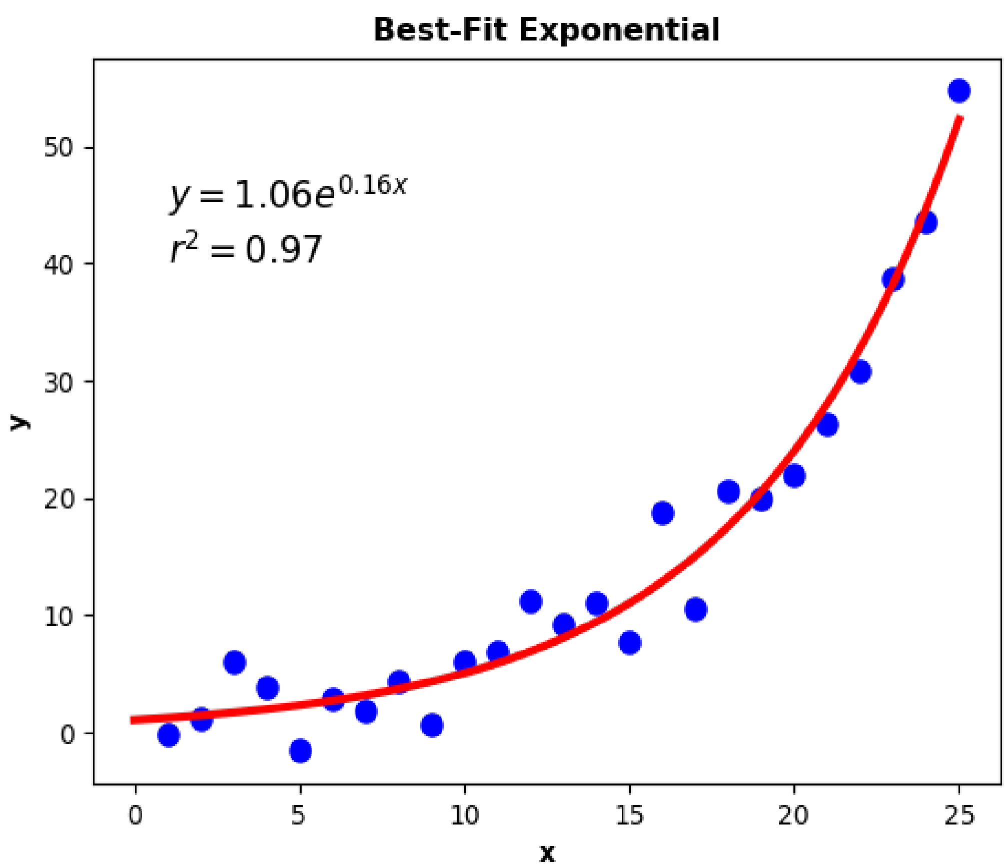
- Find the minimum of a function of two or more variables

```
opt = minimize(f, x0)
```

  - `f`: function to be optimized
  - `x0`: array of initial values
  - `opt`: `optimizeResult` object returned – includes:
    - `opt.x`: the solution of the optimization (i.e., x_{opt})
    - `opt.fun`: value of objective function at the optimum (i.e., $f(x_{opt})$)
    - `opt.nit`: number of iterations

Nonlinear Regression – minimize()


```
# %% noiseless data
x = np.arange(1,26)
y = 1.2*np.exp(0.15*x);

# %% add noise to y data
sig = 2.5

# set the random number generator seed
seed = 4

rng = np.random.default_rng(seed)
v = rng.normal(scale=sig, size=len(x))
yn = y + v

# %% define the cost function
J = lambda a: sum((yn - a[0]*np.exp(a[1]*x))**2)

# %% initial guess for parameters
a0 = [10, 1]

# %% perform the optimization
a = minimize(J, a0, method='Nelder-Mead')

# parameters for the exponential
alpha = a.x[0]
beta = a.x[1]

# %% the exponential fit
xfit = np.linspace(0,max(x),200)
yexp = alpha*np.exp(beta*xfit)
yexpr2 = alpha*np.exp(beta*x)

# %% calculate the coefficient
# of determination for exp fit
ybar = np.mean(yn)
St = sum((yn - ybar)**2)
Sr = sum((yn - yexpr2)**2)
r2 = (St - Sr)/St
```



Nonlinear Regression – `curve_fit()`

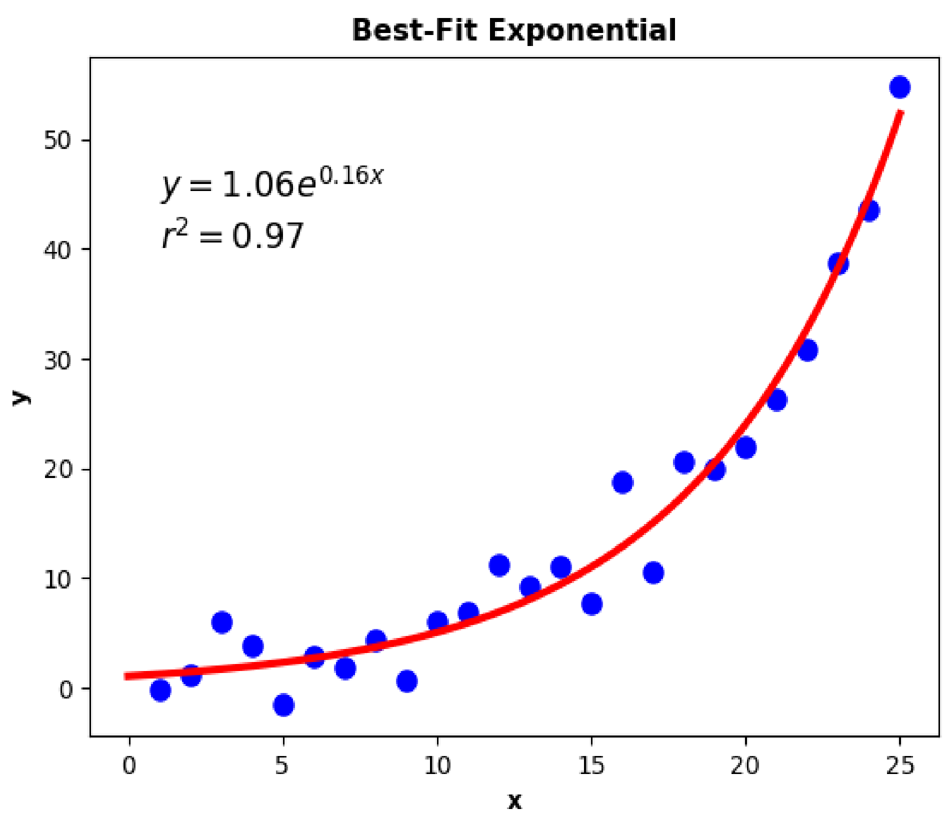
- An alternative to minimizing a cost function using `scipy.optimize.curve_fit()`:

```
popt, pcov = curve_fit(f, x, y, p0=None)
```

  - `f`: handle to the fitting function – independent variable must be listed first
    - e.g., `f = lamba x, A, B: A*exp(B*x)`
  - `x`: independent variable data
  - `y`: dependent variable data
  - `p0`: initial guess for `popt` - optional
  - `popt`: best-fit parameters
  - `pcov`: estimated covariance of `popt`

Nonlinear Regression – curve_fit()


```python
# %% noiseless data
x = np.arange(1,26)
y = 1.2*np.exp(0.15*x);

# %% add noise to y data
sig = 2.5

# set the random number generator seed
seed = 4

rng = np.random.default_rng(seed)
v = rng.normal(scale=sig, size=len(x))
yn = y + v

# %% the fitting function
f = lambda x, alpha, beta: alpha*np.exp(beta*x)

# %% perform the fit with lsqcurvefit.m
popt, pcov = curve_fit(f, x, yn)

# parameters for the exponential
alpha = popt[0]
beta = popt[1]

# %% the exponential fit
xfit = np.linspace(0,max(x),200)
yexp = alpha*np.exp(beta*xfit)
yexpr2 = alpha*np.exp(beta*x)

# %% calculate the coefficient
# of determination for exp fit
ybar = np.mean(yn)
St = sum((yn - ybar)**2)
Sr = sum((yn - yexpr2)**2)
r2 = (St - Sr)/St
```



Polynomial Interpolation

Polynomial Interpolation

- Sometimes we ***know both x and y values exactly***
  - Want a function that describes $y = f(x)$
    - Allows for ***interpolation*** between know data points
  - Fit an n^{th}-order polynomial to $n+1$ data points

$$y = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_n$$

  - Polynomial will pass through all points
- We'll look at ***polynomial interpolation*** using
  - ***Newton's polynomial***
  - The ***Lagrange polynomial***

Polynomial Interpolation

$$y = a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_n$$

- Can approach similar to linear least-squares regression

$$y = a_0 z_0 + a_1 z_1 + \cdots + a_n z_n$$

where

$$z_0 = x^n,\ z_1 = x^{n-1}, \ldots\ z_n = 1$$

- For an n^{th}-order polynomial, we have $n+1$ equations with $n+1$ unknowns
- In matrix form

$$\mathbf{y} = \mathbf{Z}\,\mathbf{a}$$

Polynomial Interpolation

- Now, unlike for linear regression
  - All $n+1$ values in $\mathbf{y}$ are known exactly
  - $n+1$ equations with $n+1$ unknown coefficients
  - $\mathbf{Z}$ is square $(n+1)\times(n+1)$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & 1 \\ x_2^n & x_2^{n-1} & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ x_{n+1}^n & x_{n+1}^{n-1} & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

- Could solve by inverting $\mathbf{Z}$ or by using NumPy's `linalg.solve()`

```
a = np.linalg.solve(Z, y)
```

- $\mathbf{Z}$ is a ***Vandermonde matrix***
  - Tend to be ill-conditioned
  - The techniques that follow are more numerically robust

Newton Interpolating Polynomial

Linear Interpolation

- Fit a line (1st-order polynomial) to two data points using a truncated ***Taylor series*** (or simple trigonometry):

$$f_1(x) = f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1)$$

  where $f_1(x)$ is the function for the line fit to the data, and $f(x_i)$ are the known data values

- This is the ***Newton linear-interpolation formula***

Quadratic Interpolation

- To fit a 2nd-order polynomial to three data points, consider the following form

$$f_2(x) = b_0 + b_1(x - x_1) + b_2(x - x_1)(x - x_2)$$

- Evaluate at $x = x_1$ to find b_0

$$b_0 = f(x_1)$$

- Back-substitution and evaluation at $x = x_2$ and at $x = x_3$ will yield the other coefficients

$$b_1 = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \quad \text{and} \quad b_2 = \frac{\frac{f(x_3) - f(x_2)}{x_3 - x_2} - \frac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_3 - x_1}$$

Quadratic Interpolation

$$f_2(x) = b_0 + b_1(x - x_1) + b_2(x - x_1)(x - x_2)$$

- Can still view this as a Taylor series approximation
  - b_0 represents an offset
  - b_1 is slope
  - b_2 is curvature
- Choice of initial quadratic form (Newton interpolating polynomial) was made to facilitate the development
  - Resulting polynomial would be the same for any initial form of an n^{th}-order polynomial
  - Solution is unique

n^{th}-Order Newton Interpolating Polynomial

- Extending the quadratic example to n^{th}-order

$$f_n(x) = b_0 + b_1(x - x_1) + \cdots + b_n(x - x_1)(x - x_2)\cdots(x - x_n)$$

- Solve for coefficients as before with back-substitution and evaluation of $f(x_i)$

$$\begin{aligned} b_0 &= f(x_1) \\ b_1 &= f[x_2, x_1] \\ b_2 &= f[x_3, x_2, x_1] \\ &\vdots \\ b_n &= f[x_{n+1}, x_n, \ldots, x_2, x_1] \end{aligned}$$

- $f[\cdots]$ denotes a ***finite divided difference***

Finite Divided Differences

- First finite divided difference

$$f[x_i, x_j] = \frac{f(x_i) - f(x_j)}{x_i - x_j}$$

- Second finite divided difference

$$f[x_i, x_j, x_k] = \frac{f[x_i, x_j] - f[x_j, x_k]}{x_i - x_k}$$

- n^{th} finite divided difference

$$f[x_{n+1}, x_n, \ldots, x_2, x_1] = \frac{f[x_{n+1}, \ldots, x_2] - f[x_n, \ldots, x_1]}{x_{n+1} - x_1}$$

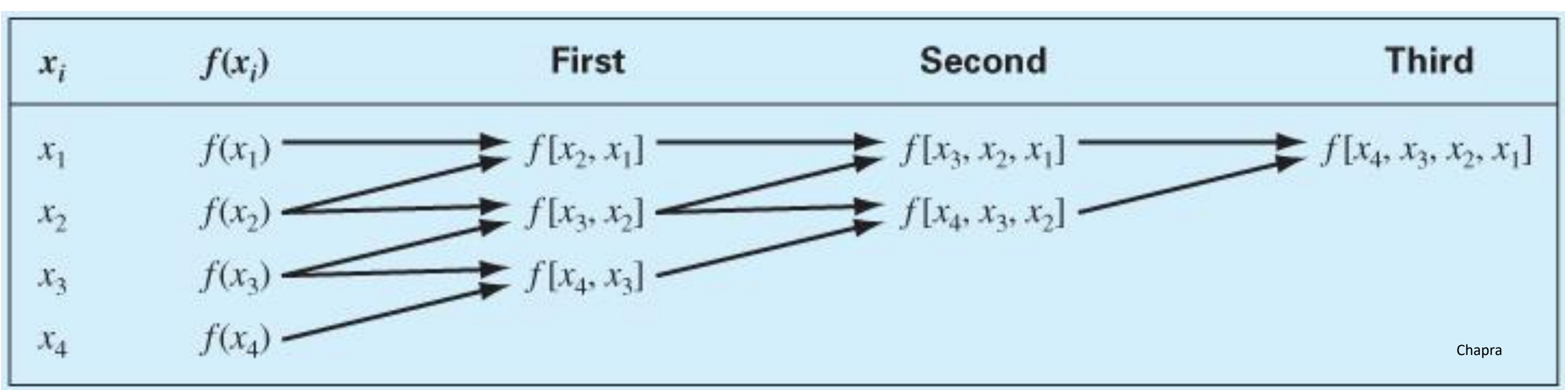
- Calculate recursively

n^{th}-Order Newton Interpolating Polynomial

- n^{th}-order Newton interpolating polynomial in terms of divided differences:

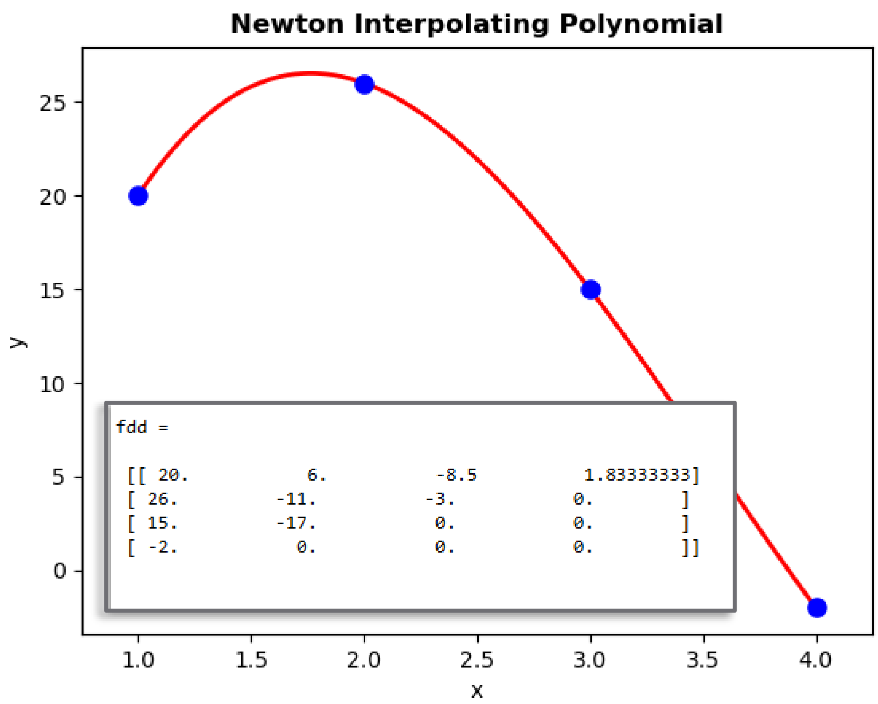
$$f_n(x) = f(x_1) + f[x_2, x_1](x - x_1) + \cdots$$
$$+ f[x_{n+1}, x_n, \ldots, x_2, x_1](x - x_1)(x - x_2) \cdots (x - x_n)$$

- Divided difference table for calculation of coefficients:



Newton Interpolating Polynomial – Example


```python
# newtpoly_test.py

import numpy as np
from matplotlib import pyplot as plt
from poly_interp import newtpoly

x = [1,2,3,4]
y = [20,26,15,-2]

xint = np.linspace(x[0], x[-1], 500)

yint = newtpoly(x,y,xint)
```




```
fdd =

[[ 20.          6.         -8.5         1.83333333]
 [ 26.        -11.         -3.          0.        ]
 [ 15.        -17.          0.          0.        ]
 [ -2.          0.          0.          0.        ]]
```

```python
def newtpoly(x,y,xint):
    '''
    Calculates an interpolating polynomial through
    points in x and y, using a Newton polynomial.
    Order of the polynomial is n=length(x)-1

    Parameters
    ----------
    x : array of independent variable data
    y : array of dependent variable data
    xint : array of independent variable values at which interpolation is
           performed

    Returns
    -------
    yint: array of values of the interpolating polynomial evaluated at xint
    '''

    # initialize the finite divided difference matrix
    n = len(x)       # order of polynomial is n-1
    fdd = np.zeros((n,n))
    fdd[:,0] = y

    # recursively calculate all divided differences
    for j in range(1, n):    # column index steps from col 1 to col n-1
        for i in range(0, n-j):  # row index steps from 1 down to the off diag.
            fdd[i,j] = (fdd[i+1,j-1] - fdd[i,j-1])/(x[i+j] - x[i])

    # first row of fdd are the b_i coefficients
    # b = fdd(0,-1:-1:-1-n)
    b = fdd[0,:]

    # generate interpolating polynomial
    yint = np.zeros(len(xint))
    yint[0] = b[0]            # first term in the polynomial sum at x=xint[0]
    # loop through all values of xint, interpolating at each
    for k in range(len(xint)):
        xt = 1
        yint[k] = b[0]
        for m in range(1, n):
            # xt is product of (x-xi) terms - one more for each incr. of m
            xt = (xint[k] - x[m-1])*xt
            yint[k] = yint[k] + b[m]*xt

    return yint
```

Lagrange Interpolating Polynomial

Linear Lagrange Interpolation

- Fit a first-order polynomial (a line) to two known data points: $\left(x_1, f(x_1)\right)$ and $\left(x_2, f(x_2)\right)$

$$f_1(x) = L_1(x) \cdot f(x_1) + L_2(x) \cdot f(x_2)$$

- $L_1(x)$ and $L_2(x)$ are ***weighting functions***, where

$$L_1(x) = \begin{cases} 1, & x = x_1 \\ 0, & x = x_2 \end{cases}$$

$$L_2(x) = \begin{cases} 1, & x = x_2 \\ 0, & x = x_1 \end{cases}$$

- The interpolating polynomial is a ***weighted sum of the individual data point values***

Linear Lagrange Interpolation

- For linear (1st-order) interpolation, the weighting functions are:

$$L_1(x) = \frac{x - x_2}{x_1 - x_2}$$

$$L_2(x) = \frac{x - x_1}{x_2 - x_1}$$

- The ***linear Lagrange interpolating polynomial*** is:

$$f_1(x) = \frac{x - x_2}{x_1 - x_2} f(x_1) + \frac{x - x_1}{x_2 - x_1} f(x_2)$$

n^{th}-Order Lagrange Interpolation

- Lagrange interpolation technique can be extended to n^{th}-order polynomials

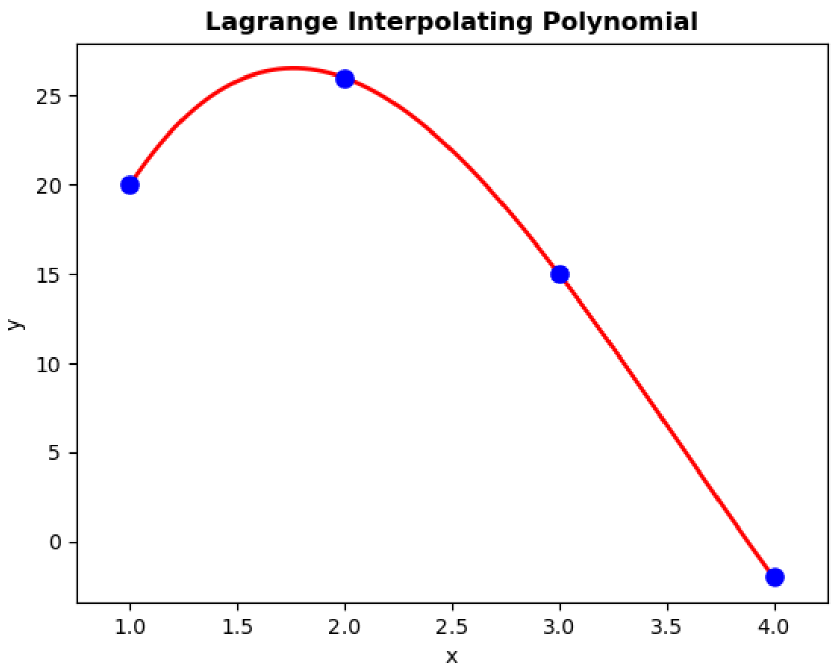
$$f_n(x) = \sum_{i=1}^{n+1} L_i(x) \cdot f(x_i)$$

where

$$L_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{x - x_j}{x_i - x_j}$$

Lagrange Interpolating Polynomial – Example


```python
# lagpoly_test.py

import numpy as np
from matplotlib import pyplot as plt
from poly_interp import lagpoly

x = [1,2,3,4]
y = [20,26,15,-2]

xint = np.linspace(x[0],x[-1],500)

yint = lagpoly(x,y,xint)
```




```python
# %% Lagrange interpolating polynomial

def lagpoly(x,y,xint):
    '''
    Calculates an interpolating polynomial through
    points in x and y, using a Lagrange polynomial.
    Order of the polynomial is n=len(x)-1

    Parameters
    ----------
    x : array of independent variable data
    y : array of dependent variable data
    xint : array of independent variable values at which interpolation is
           performed

    Returns
    -------
    yint: array of values of the interpolating polynomial evaluated at xint
    '''

    n = len(x)
    yint = np.zeros(len(xint))

    for k in range(len(xint)):
        for i in range(n):
            L = 1
            for j in range(n):
                if j != i:
                    L = L*(xint[k] - x[j])/(x[i] - x[j])

            yint[k] = yint[k] + L*y[i]

    return yint
```