SECTION 9: ENGINEERING APPLICATIONS

ENGR 103 – Introduction to Engineering Computing

Systems of Equations

Systems of Equations

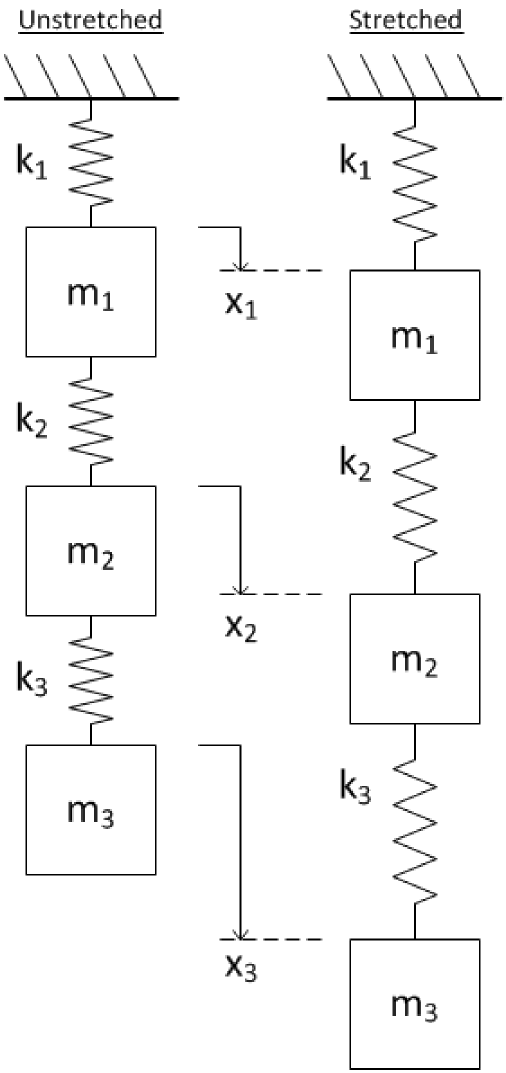
- Systems of equations common in all engineering disciplines
- For N unknown variables, we need a system of N equations
  - Can represent in matrix form:

$$\mathbf{Ax} = \mathbf{b}$$

    - A: $N \times N$ matrix of known, constant coefficients
    - x: $N \times 1$ vector of unknowns
    - b: $N \times 1$ vector of known constants

- Many tools exist for solving:
  - By hand – substitution, Gaussian elimination, etc.
  - Scientific calculators
  - Here, we will look at the tools available within Python

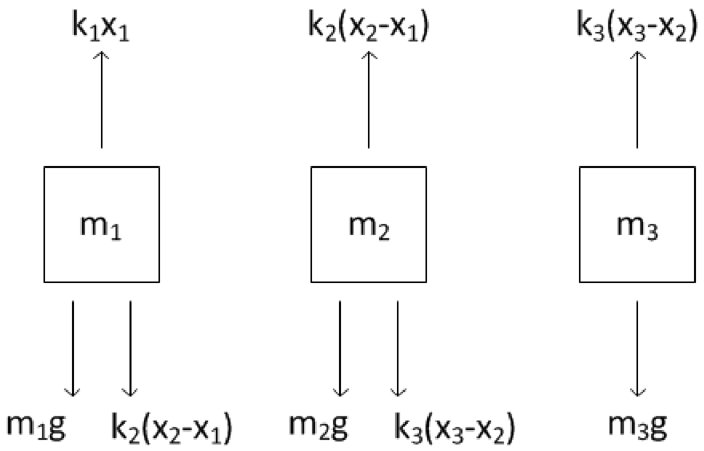
A System of Equations – Example



- Consider the following scenario
- Three masses
  - m_1, m_2, and m_3
- Three springs
  - k_1, k_2, k_3
- Connected in series and suspended
- Determine the displacement of each mass from its unstretched position

A System of Equations – Example

- Three unknown displacements: x_1, x_2, x_3
  - Need three equations to find displacements
- Apply Newton's second law to each mass



- Three equations result:

$$m_1\ddot{x}_1 = m_1 g + k_2(x_2 - x_1) - k_1 x_1$$
$$m_2\ddot{x}_2 = m_2 g + k_3(x_3 - x_2) - k_2(x_2 - x_1)$$
$$m_3\ddot{x}_3 = m_3 g - k_3(x_3 - x_2)$$

A System of Equations – Example

- Steady-state, so no acceleration: $\ddot{x}_i = 0, \ \forall i$

$$m_1 g + k_2(x_2 - x_1) - k_1 x_1 = 0$$

$$m_2 g + k_3(x_3 - x_2) - k_2(x_2 - x_1) = 0$$

$$m_3 g - k_3(x_3 - x_2) = 0$$

- Rearranging

$$\begin{aligned}
(k_1 + k_2)x_1 \quad & - k_2 x_2 \quad + 0x_3 = m_1 g \\
-k_2 x_1 \quad & + (k_2 + k_3)x_2 \quad - k_3 x_3 = m_2 g \\
0x_1 \quad & - k_3 x_2 \quad + k_3 x_3 = m_3 g
\end{aligned}$$

A System of Equations – Example

- Our system of three equations

$$\begin{aligned} (k_1+k_2)x_1 \quad & - k_2x_2 \quad + 0x_3 = m_1g \\ -k_2x_1 \quad + (k_2+k_3)x_2 \quad & - k_3x_3 = m_2g \\ 0x_1 \quad & - k_3x_2 \quad + k_3x_3 = m_3g \end{aligned}$$

can be put into matrix form

$$\begin{bmatrix} (k_1+k_2) & -k_2 & 0 \\ -k_2 & (k_2+k_3) & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} m_1g \\ m_2g \\ m_3g \end{bmatrix}$$

A System of Equations – Example

$$\begin{bmatrix} (k_1 + k_2) & -k_2 & 0 \\ -k_2 & (k_2 + k_3) & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} m_1 g \\ m_2 g \\ m_3 g \end{bmatrix}$$

- We can rewrite this matrix equation as

$$\mathbf{Ax} = \mathbf{b}$$

- Can apply tools of linear algebra to determine the vector of unknown displacements

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Solution Using Matrix Inverse

- We have a system of equations:

$$\mathbf{Ax} = \mathbf{b}$$

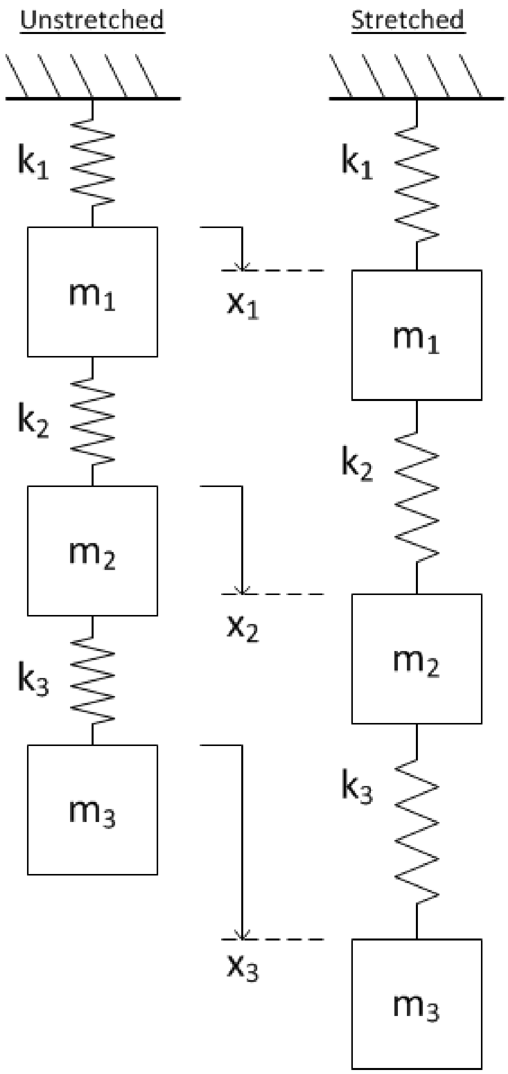
- If a solution exists, then the coefficient matrix, $\mathbf{A}$, is invertible
  - Not always the case
- Left-multiply by $\mathbf{A}^{-1}$ to solve for the vector of unknowns, x

$$\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b}$$

$$\mathbf{Ix} = \mathbf{A}^{-1}\mathbf{b}$$

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Solution Using Matrix Inverse



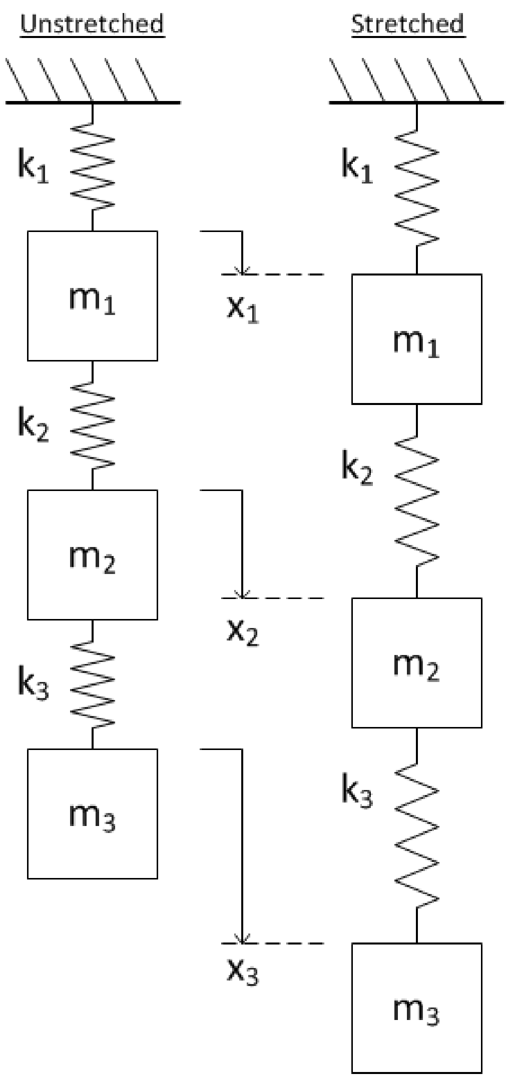
- Our linear system is described by the matrix equation

$$\begin{bmatrix} (k_1+k_2) & -k_2 & 0 \\ -k_2 & (k_2+k_3) & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} m_1 g \\ m_2 g \\ m_3 g \end{bmatrix}$$

$$\mathbf{Ax} = \mathbf{b}$$

- Find the displacements, $\mathbf{x}$, for the following system parameters
  - $k_1 = 500\frac{N}{m},\ k_2 = 800\frac{N}{m},\ k_3 = 400\frac{N}{m}$
  - $m_1 = 3kg,\ m_2 = 1kg,\ m_3 = 7kg$

Solution Using Matrix Inverse




```python
import numpy as np

# spring constants
k1 = 500
k2 = 800
k3 = 400

# masses
m1 = 3
m2 = 1
m3 = 7

g = 9.81    # gravitational acceleration

A = np.array([[k1+k2,   -k2,    0],
              [  -k2, k2+k3, -k3],
              [    0,   -k3,   k3]])

b = np.array([[m1*g],
              [m2*g],
              [m3*g]])

# solve using matrix inverse
x = np.linalg.inv(A)@b

print(x)
```

Console 1/A

```
In [144]: runfile('
Box/KWebb/Classes/E
[[0.21582 ]
 [0.31392 ]
 [0.485595]]
```

$$x_1 = 21.6cm, \quad x_2 = 31.4cm, \quad x_3 = 48.6cm$$

Solution Using `np.linalg.solve()`

- The linalg module in the NumPy package has a function for solving linear systems of equations
  - `np.linalg.solve()`
- Use `np.linalg.solve()` to solve

$$\mathbf{Ax} = \mathbf{b}$$

- If $\mathbf{A}^{-1}$ exists, then

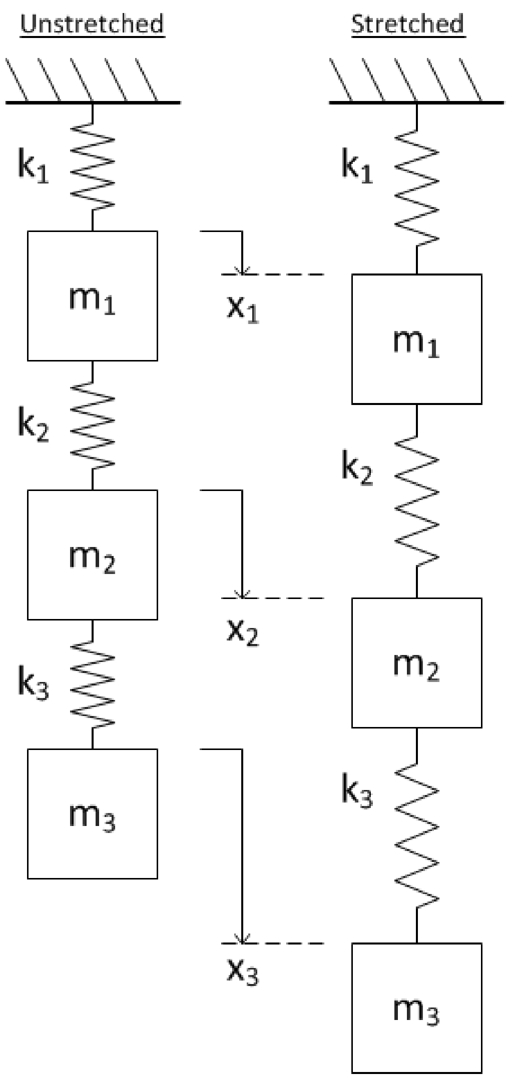
```
x = np.linalg.solve(A,b)
```

is equivalent to

```
x = np.linalg.inv(A)@b
```

- But, does not calculate $\mathbf{A}^{-1}$
  - Faster and more numerically robust

Solution Using np.linalg.solve()




```python
# spring constants
k1 = 500
k2 = 800
k3 = 400

# masses
m1 = 3
m2 = 1
m3 = 7

g = 9.81    # gravitational acceleration

A = np.array([[k1+k2,   -k2,    0],
              [  -k2, k2+k3, -k3],
              [    0,   -k3,  k3]])

b = np.array([[m1*g],
              [m2*g],
              [m3*g]])

# solve using matrix inverse
x = np.linalg.inv(A)@b

print(x)

print()

# solve using np.linalg.solve()
x = np.linalg.solve(A,b)

print(x)
```

Console 1/A

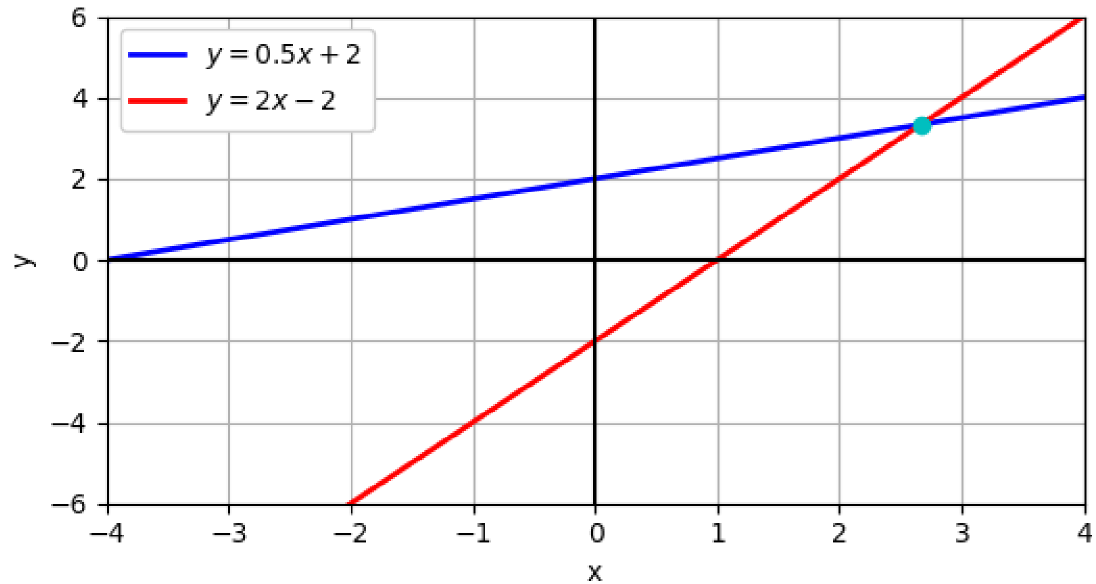
```
In [145]: runfile('C
Box/KWebb/Classes/EN
[[0.21582 ]
 [0.31392 ]
 [0.485595]]

[[0.21582 ]
 [0.31392 ]
 [0.485595]]

In [146]:
```

$$x_1 = 21.6cm, \quad x_2 = 31.4cm, \quad x_3 = 48.6cm$$

Exercise – System of Equations

Exercise

- Write a script in which you define and solve a system of equations to determine the point of intersection of the lines in the plot below



- Solve the system of equations two ways:
  - Using np.linalg.inv()
  - Using np.linalg.solve()

Numerical Differentiation

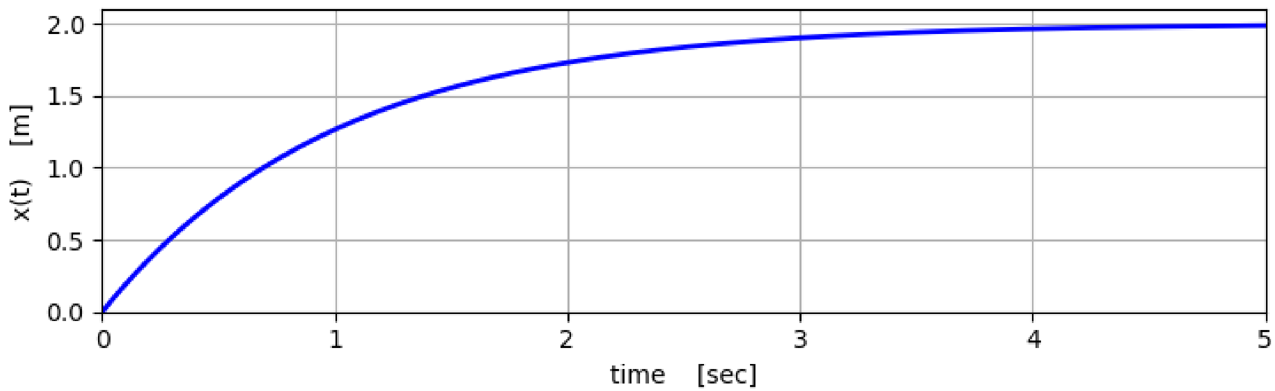
Differentiation

- As engineers, we often deal with ***rates***
  - ***Changes in one quantity with respect to another***
- Often these are rates with respect to time, e.g.:
  - *Velocity*: change in position w.r.t. time
  - *Acceleration*: change in velocity w.r.t. time
  - *Power*: time rate of energy transfer
  - Changes in *voltage* or *current* w.r.t. time
  - Etc.
- Mathematically, these rates are described by ***derivatives***
- Calculation of a derivative is ***differentiation***

Derivatives

- For example, consider an object whose ***position as a function of time*** is

$$x(t) = 2\,m \cdot (1 - e^{-t})$$



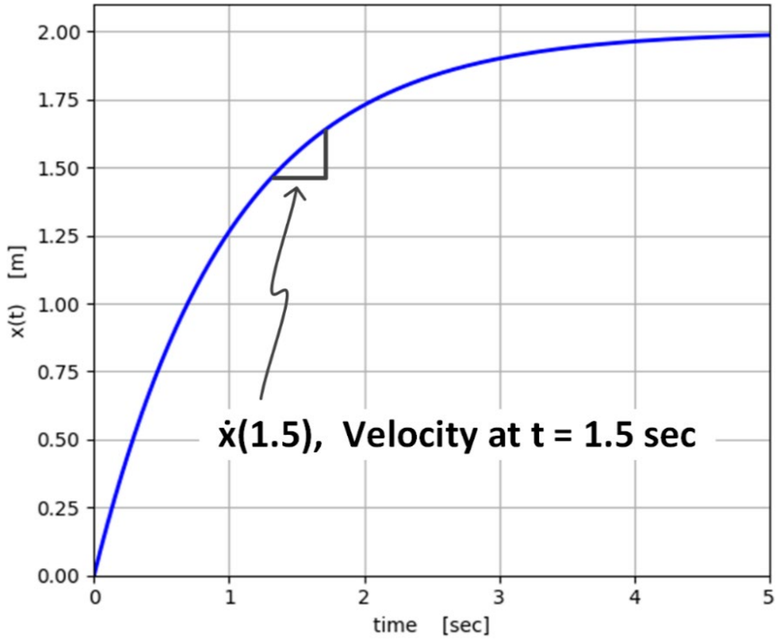
- At any point in time, t, the object's velocity, $v(t)$, is given by the time rate of change of position
  - That is, the ***derivative w.r.t. time*** of position

$$v(t) = \frac{dx}{dt} = \dot{x}(t) = x'(t)$$

Derivatives

- Velocity is the ***rate of change*** of position w.r.t. time
  - ***Slope*** of the position graph
  - The ***derivative*** of position

$$v(t) = \frac{dx}{dt} = \dot{x}(t)$$



- You know/will learn to differentiate mathematical expressions, e.g.

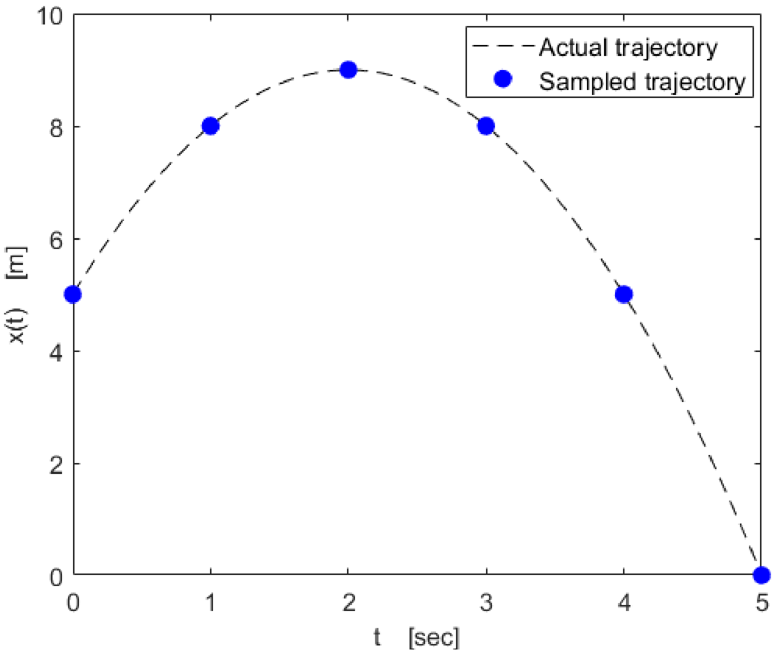
$$x(t) = 2\,m \cdot (1 - e^{-t})$$

$$\dot{x}(t) = v(t) = 2\frac{m}{s} \cdot e^{-t}$$

- Often, we would like to calculate a derivative, but we do not have a mathematical expression, e.g.
  - Measurement data
  - Simulation data, etc.
- Then, we can ***approximate*** the derivative ***numerically***

Numerical Differentiation

- Data we want to differentiate are ***discrete***
  - ***Sampled*** – not continuous
  - Data only exist at ***discrete*** points in time
  - Result of simulation or measurement, etc.

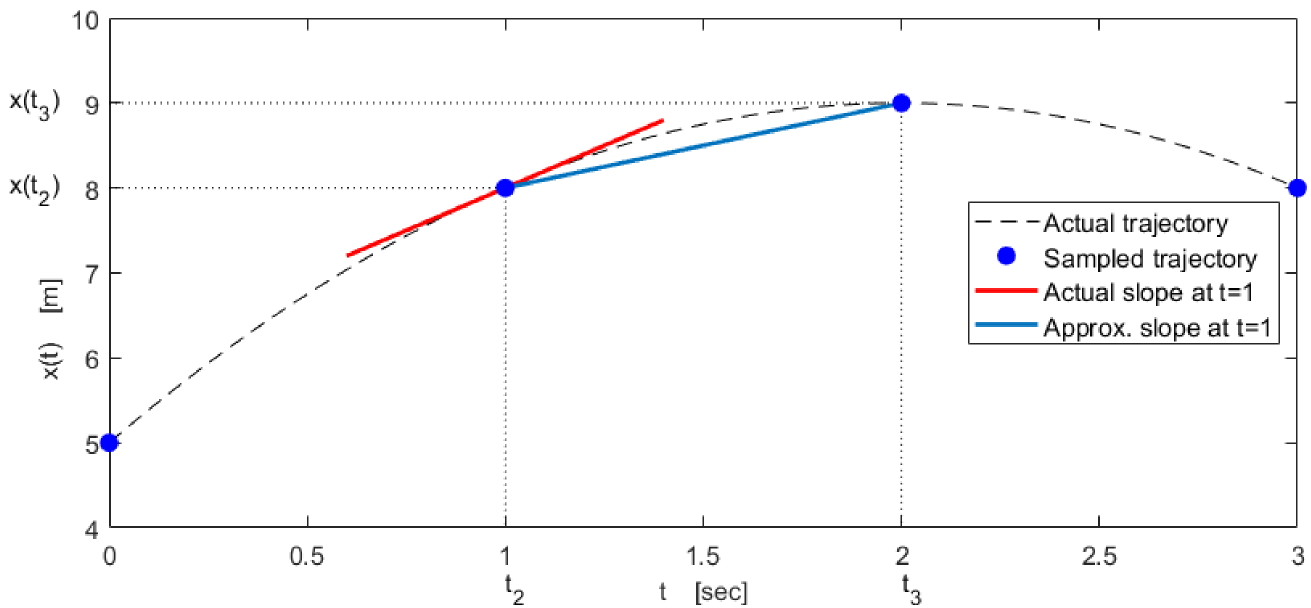


- ***Numerical differentiation***
  - Approximation of the slope at each discrete data point
- Several methods exist for numerical differentiation
  - Varying complexity and accuracy
- Here, we'll focus on the ***forward difference method***

Forward Difference Method

- ***Forward difference method***
  - Approximate $\dot{x}(t_i)$ using $x(t_i)$ and $x(t_{i+1})$
    - Data at the current time point and one time step ***forward***

$$\dot{x}(t_i) \approx \frac{x(t_{i+1}) - x(t_i)}{t_{i+1} - t_i} = \frac{\Delta x}{\Delta t}$$



Forward Difference in Python

- Numerical differentiation in Python using NumPy

$$\dot{x}(t_i) \approx \frac{x(t_{i+1}) - x(t_i)}{t_{i+1} - t_i} = \frac{\Delta x}{\Delta t}$$

- We would have:
  - Time vector, t
    - Possibly, but not necessarily evenly spaced
  - Data vector, $x(t)$
    - Function to be differentiated

- Use `np.diff()` to calculate Δx and Δt vectors
- Divide to calculate $\Delta x/\Delta t$ at each time point
  - No $\Delta x/\Delta t$ value at the last time point

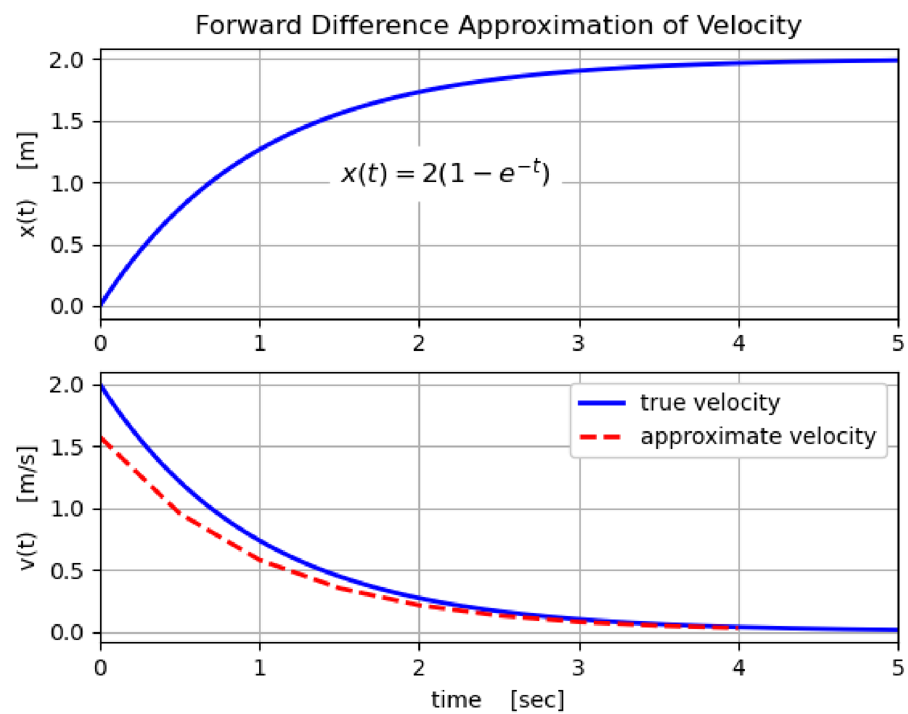
Numerical Differentiation – Example

- Consider again an object whose position is given by:

  $$x(t) = 2\,m \cdot (1 - e^{-t})$$

- Use forward difference to approximate velocity
  - Assume a 500 msec sample period
- Error would improve with smaller time steps

```python
# sampled function
ts = np.arange(0, 5, 500e-3)
xs = 2*(1 - np.exp(-ts))

dx = np.diff(xs)     # position differences
dt = np.diff(ts)     # time differences

dxdt = dx/dt         # approx. derivative

```



Exercise – Numerical Differentiation

Exercise

- Write a script in which you:
  - Calculate $y = \sin(x)$ over a range of $x = [0, 4\pi]$
  - Calculate the approximate derivative of y with respect to x, $\frac{dy}{dx}$
  - Plot $y(x)$ and $\frac{dy}{dx}$ on the same set of axes
- Does the plot make sense in terms of the slope of $y(x)$?
- Does the plot agree with the true derivative of $y(x)$?

Numerical Integration

Integration

$$\int_a^b f(t)dt$$

- ***Integration*** is a mathematical operation involving the calculation of a ***continuous sum*** over some interval
  - The inverse of differentiation – the antiderivative

$$\int f'(t)dt = f(t)$$

- We have seen that the derivative represents the rate of change of a function w.r.t. its independent variable
  - For example, consider the position of an object, $x(t)$
  - Velocity of the object is the derivative of position

$$v(t) = \frac{dx}{dt} = x'(t)$$

  - The rate of change of position w.r.t. time

Integration

- ***Integration is the inverse of differentiation***
  - Mathematical transform between a rate of a quantity (e.g., $v(t) = x'(t)$) and that quantity (e.g., $x(t)$)

$$x(t) = \int v(t)\, dt = \int x'(t)\, dt$$

- Examples of integral/derivative relationships:

	Integral → / ← Derivative	
Velocity		Position
Acceleration		Velocity
Power		Energy
Current		Electrical charge

Integration

- In your calculus class you learned/will learn to calculate the integral of functions, e.g.,

$$\int_0^1 e^{-\frac{t}{2}}\,dt = -2 \cdot e^{-\frac{t}{2}}\Big|_0^1$$

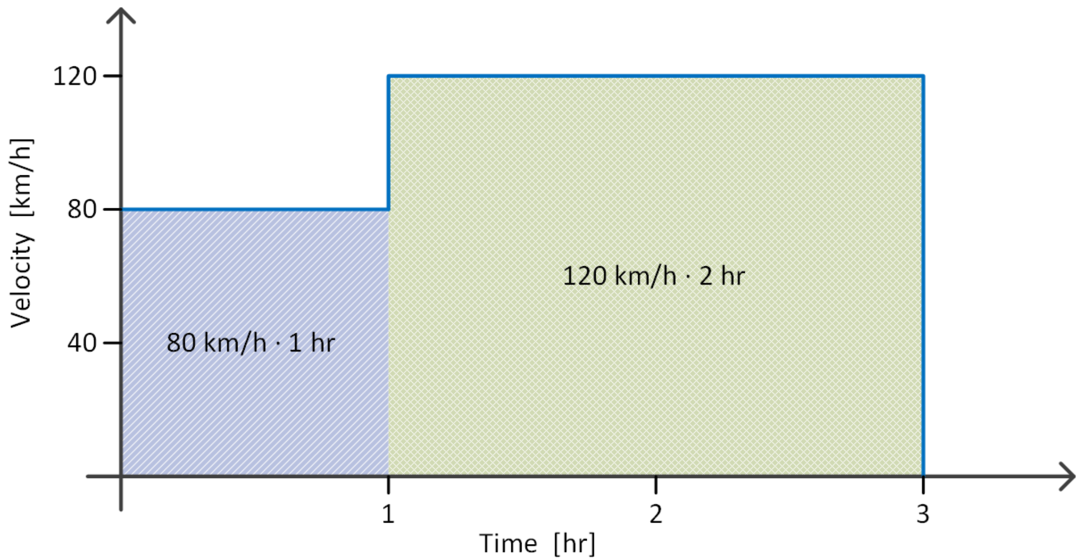
$$= -2(0.6065 - 1)$$

$$\int_0^1 e^{-\frac{t}{2}}\,dt = 0.787$$

- As was the case for differentiation, we often do not have a mathematical expression for the data we want to integrate
  - E.g., measurement data or simulation data
  - Only have discrete data points
  - Integrate ***numerically***

Numerical Integration

- The ***derivative*** of a function is the ***slope of its graph***
- The ***integral*** of a function is the ***area under its graph***
- For example, distance traveled is the integral of velocity
  - Consider a car that travels at a speed of 80 km/h for 1 hour and 120 km/h for 2 hours
    - How far has the car traveled after three hours?



Numerical Integration

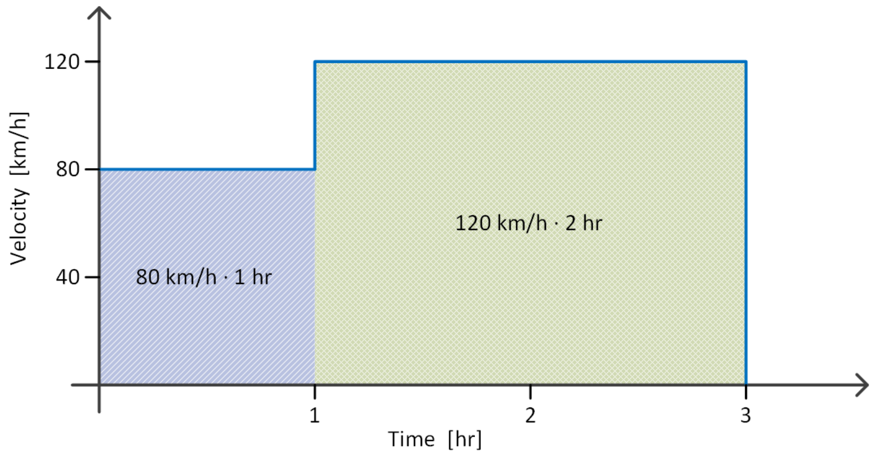
- Distance at $t = 3\ hr$:

$$x(3) = \int_0^3 v(t)\,dt$$

$$x(3) = \int_0^1 v(t)\,dt + \int_1^3 v(t)\,dt$$

$$x(3) = 80\frac{km}{h}\cdot 1\ hr + 120\frac{km}{h}\cdot 2\ hr$$

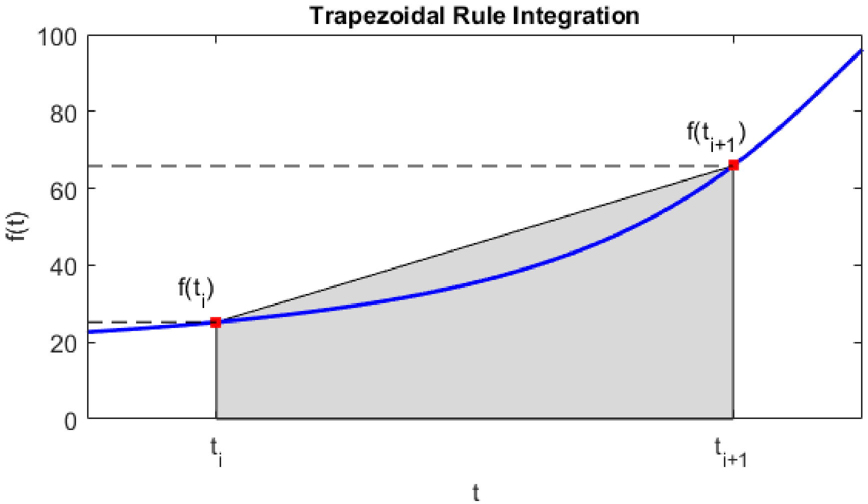
$$x(3) = 320\ km$$



- ***Numerical integration***
  - Numerical approximation of area under a curve defined by a function or a discrete data set
  - We will focus on one simple method: the ***trapezoidal rule***

Trapezoidal Rule Integration

- Approximate the integral between adjacent time point:
  - Approximate area under the curve between those time points
    - Area of a trapezoid

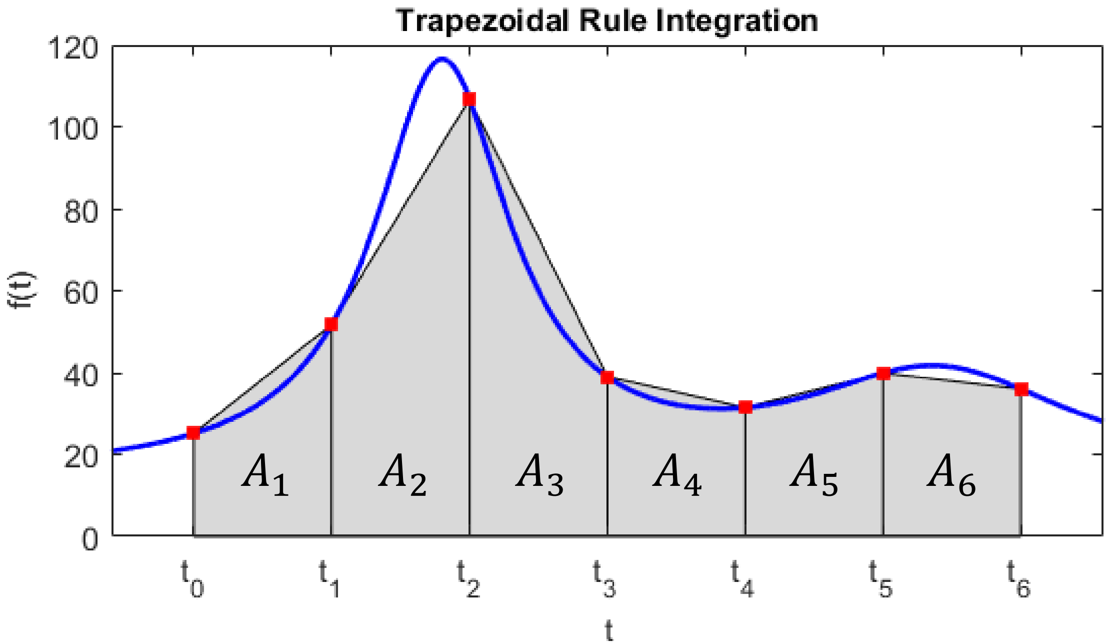


$$Area \approx \frac{f(t_i) + f(t_{i+1})}{2} \cdot (t_{i+1} - t_i)$$

$$Area \approx (Avg.height) \cdot (width)$$

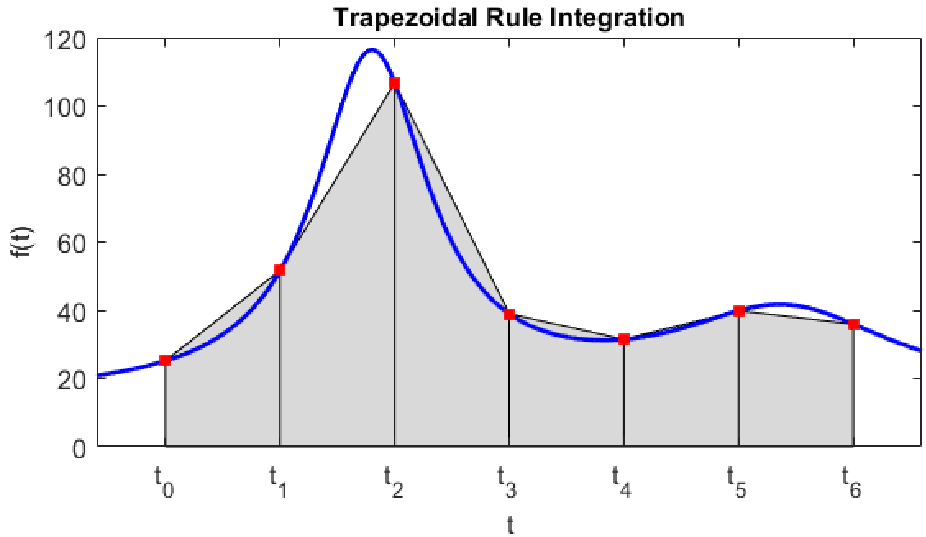
Trapezoidal Rule Integration

- Overall integral approximated by the approximate total area
  - Sum of all individual trapezoidal segment areas



$$\int_{t_0}^{t_6} f(t)\,dt \approx \sum_{i=1}^{6} A_i = A_1 + A_2 + A_3 + A_4 + A_5 + A_6$$

Trapezoidal Rule Integration



$$\int_{t_0}^{t_6} f(t)\,dt \approx \sum_{i=0}^{5} \frac{f(t_i) + f(t_{i+1})}{2} \cdot (t_{i+1} - t_i)$$

$$\int_{t_0}^{t_6} f(t)\,dt \approx \left[\frac{f(t_0) + f(t_1)}{2} \cdot (t_1 - t_0)\right] + \left[\frac{f(t_1) + f(t_2)}{2} \cdot (t_2 - t_1)\right] + \cdots$$

$$\cdots + \left[\frac{f(t_5) + f(t_6)}{2} \cdot (t_6 - t_5)\right]$$

Trapezoidal Rule in Python – `trapezoid()`

- We will use the ***integrate module*** from the ***SciPy package*** for integrating in Python
  - Must import it first:

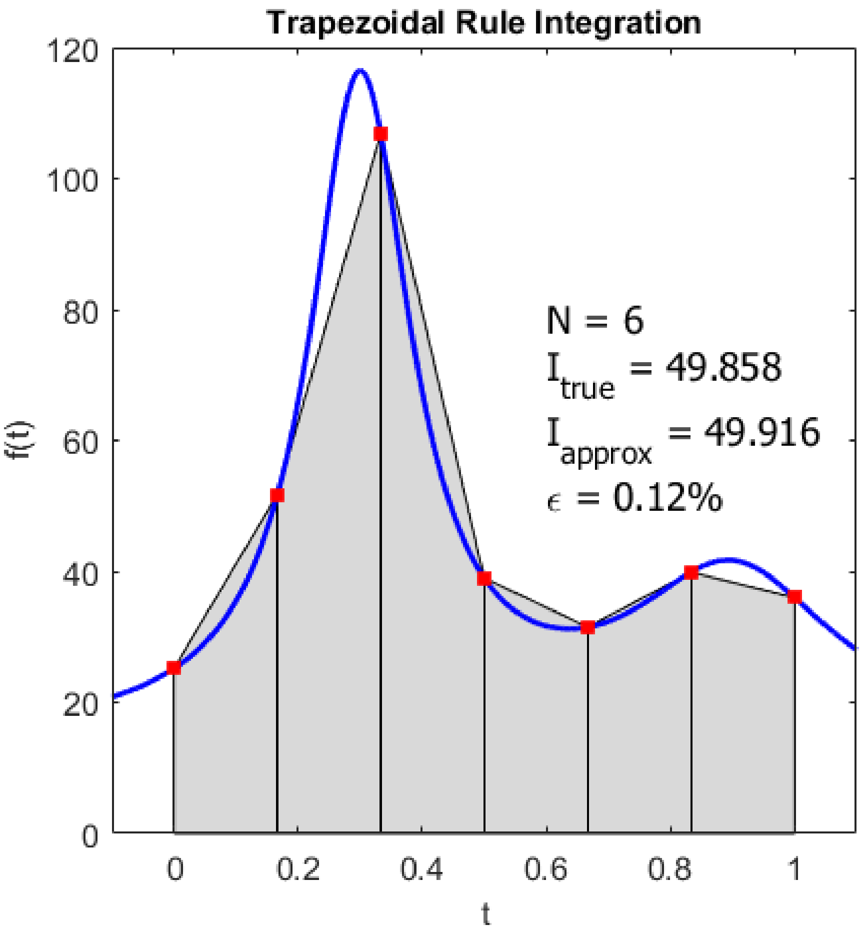
```
from scipy import integrate
```

```
I = integrate.trapezoid(y, x)
```

  - `y`: vector of dependent variable data
  - `x`: vector of independent variable data
  - `I`: trapezoidal rule approximation to the integral of `y` with respect to `x` (a scalar)

- Data need not be equally-spaced
  - Segment widths calculated from `x` values

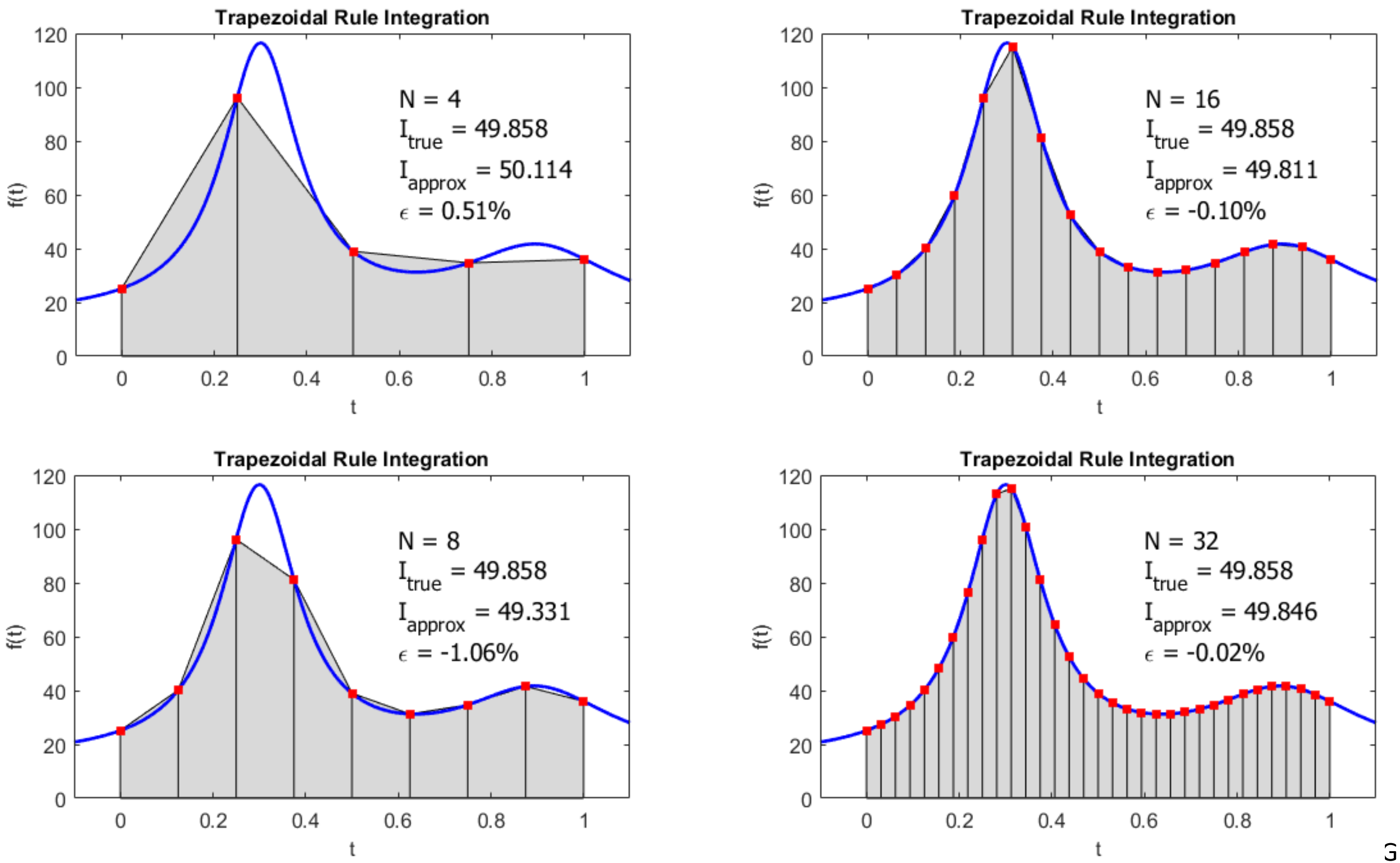
Trapezoidal Rule – Example




```python

import numpy as np
from scipy import integrate

# the function to be integrated
# in practice, we would generally not have this
f = lambda t: 1 / ((t-0.3)**2 + .01) + 1 / ((t-0.9)**2 + 0.04) + 14

# function for the true integral
# generally would not have this either
intf = lambda t: 14*t + 10*np.arctan(10*t - 3) + 5*np.arctan(5*t - 9/2)

# evaluate f(t) over [a,b] with N segments, N+1 samples
a = 0
b = 1
N = 6
t = np.linspace(a, b, N+1)

y = f(t)

# approx. the integral over [a,b] using trapezoid()
Ihat = integrate.trapezoid(y, t)

# true value of the integral over [a,b]
I = intf(b) - intf(a)

# percent error of numerical approximation
err = (Ihat - I)/I * 100

```

Trapezoidal Rule – Example

- Error decreases as
  - Number of segments (sampling frequency) increases
  - Segment size (sampling period) decreases



Indefinite Integrals

- Sometimes, we want to know the result of an integral from a to b
  - A ***definite integral***
  - A number
  - E.g., given velocity $v(t)$, find the total distance traveled

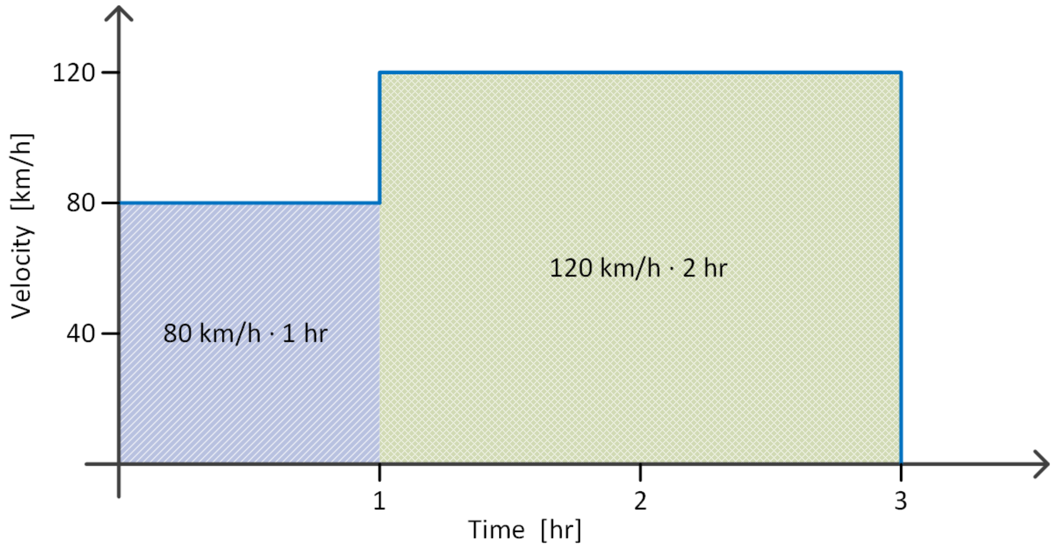
$$\Delta x = x(b) - x(a) = \int_a^b v(t)\, dt$$

- Other times, we would like the result of an integral as a function of time
  - An ***indefinite integral*** or a ***cumulative integral***
  - E.g., given $v(t)$, find the distance traveled as a function of time

$$x(t) = \int_0^t v(\tau)\, d\tau$$

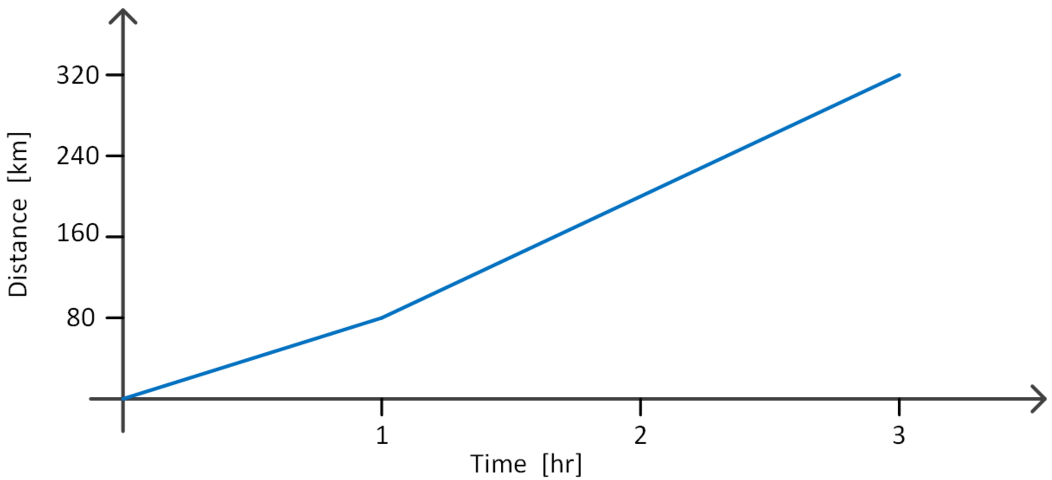
Indefinite Integrals

- Velocity, $v(t)$:



- Integrate velocity to get distance as a function of time:

$$x(t) = \int v(t)\,dt$$



Cumulative Integral – `cumulative_trapezoid()`

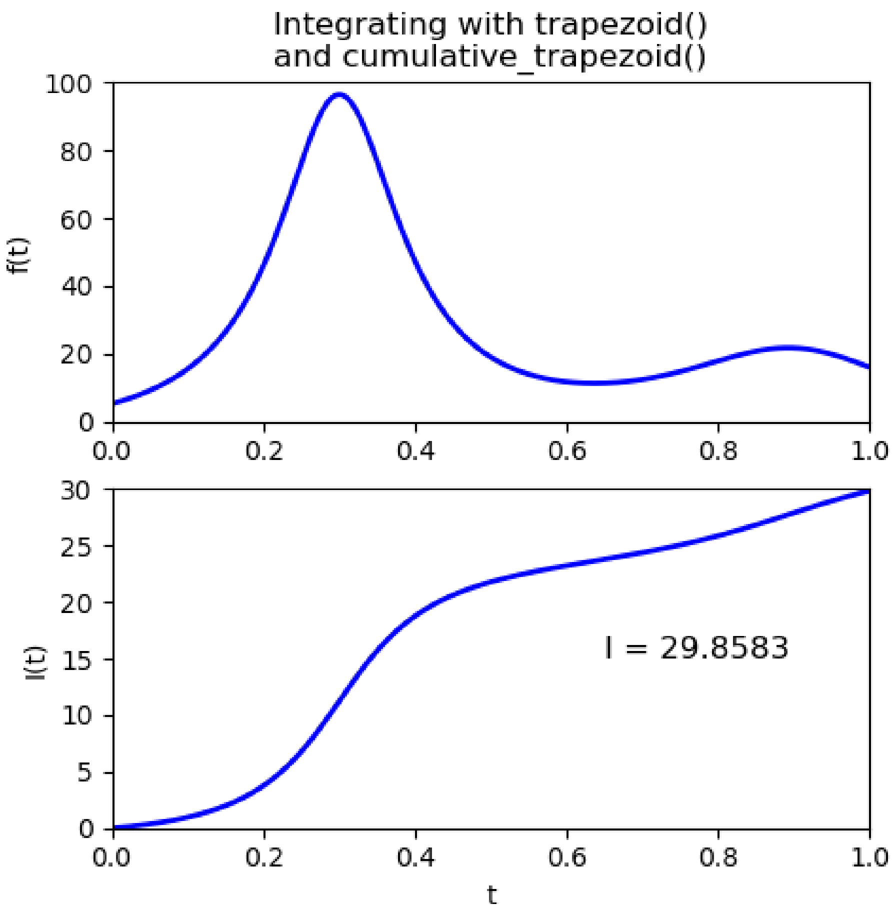

```
I = integrate.cumulative_trapezoid(y, x,
initial=0)
```

- `y`: n-vector of dependent variable data
- `x`: n-vector of independent variable data
- `initial`: optional initial value inserted as the first value in `I` – if not given, `I` is an (n-1)-vector
- `I`: trapezoidal rule approximation to the ***cumulative integral*** of `y` with respect to `x` (an n-vector)

- Result is a vector – equivalent to:

$$I(x) = \int_{x_1}^{x} y(\tilde{x})\, d\tilde{x}$$

trapezoid() and cumulative_trapezoid()




```python
import numpy as np
from scipy import integrate
from matplotlib import pyplot as plt

def humps(x):
    y = 1 / ((x-0.3)**2 + .01) + 1 / ((x-0.9)**2 + 0.04) - 6
    return y

t = np.linspace(0, 1, 2000)
y = humps(t)

# definite integral
I = integrate.trapezoid(y, t)

# cumulative or indefinite integral
Ic = integrate.cumulative_trapezoid(y, t, initial=0)

plt.figure(1).clf()
plt.subplot(211)
plt.plot(t, y, '-b', linewidth=2)
plt.ylabel('f(t)')
plt.title('''Integrating with trapezoid()
and cumulative_trapezoid()''')
plt.xlim(0, 1)
plt.ylim(0, 100)

plt.subplot(212)
plt.plot(t, Ic, '-b', linewidth=2)
plt.xlabel('t')
plt.ylabel('I(t)')
plt.text(0.65, 15, 'I = {:1.4f}'.format(I),
         fontsize=12)
plt.xlim(0, 1)
plt.ylim(0, 30)
```

Integrating Functions – `integrate.quad()`

- If we do have an expression for the function to be integrated, we can use SciPy's `integrate.quad()` function:

```
I = integrate.quad(f,a,b)
```

  - `f`: the function to be integrated
  - `a`: lower integration limit
  - `b`: upper integration limit
  - `I`: numerical approximation of the integral

- Calculates $I = \int_a^b f(x)dx$

Exercise – Numerical Integration

Exercise

- Add to your script from the previous exercise (numerical differentiation) to do the following:
  - Numerically approximate the integral of what you calculated as the approximate derivative of

$$y(x) = \sin(x)$$

  - The result should be approximately the function you started with, i.e.,

$$\hat{y}(x) \approx \sin(x)$$

  - Add $\hat{y}(x)$ to your plot along with $y(x)$ and its approximate derivative.
- Play around with the number of points in your x vector, and see how that affects the results

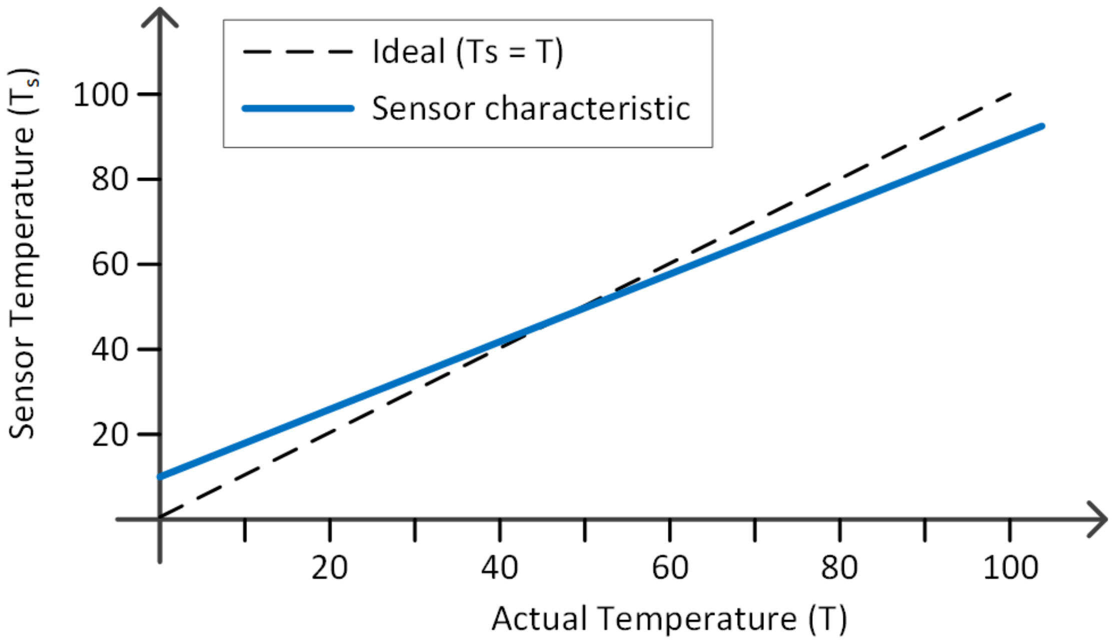
Curve Fitting

Curve Fitting

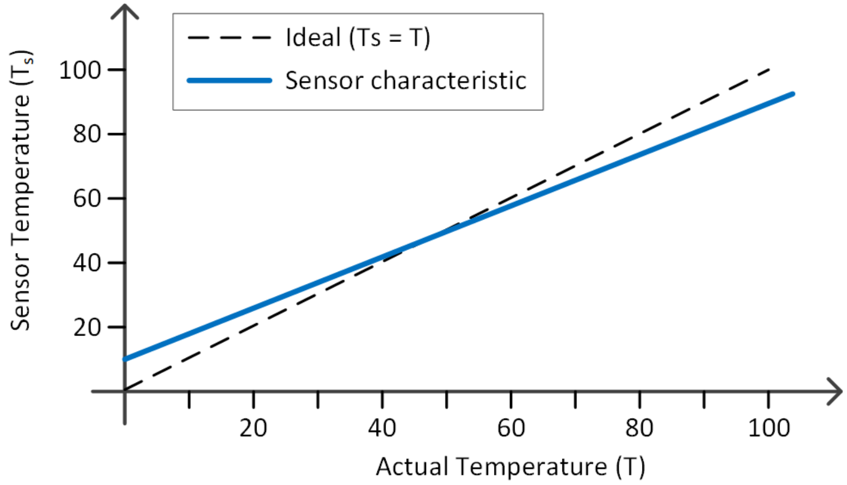
- Engineers often deal with discrete data sets, e.g.
  - E.g., measurement or simulation data
- Typically, that data is noisy
  - Measurement noise
  - Random variations, external disturbances, etc.
- Typically don't have a mathematical expression for the data
  - But, we may want one
  - Sometimes, we may know the data should follow a certain type of function
    - E.g., linear, quadratic, exponential, etc.
- We can ***fit a curve to the data***
  - Determine function parameters that best fit the data
    - E.g., slope and intercept values for a linear relationship
  - Or, determine what type of function provides the best fit
    - E.g., linear, quadratic, exponential, etc.

Curve Fitting

- Consider the following engineering example:
- An inexpensive temperature sensor is to be used to measure ambient temperature
  - Temperature measured and recorded by a micro-controller
  - Low accuracy (inexpensive)
- Sensor output compared to actual temperature may look like:



Curve Fitting



- Ideally, the sensor temperature, T_s, would equal the true temperature, T:

$$T_s = T$$

- But, due to inaccuracy:

$$T_s = a_1 \cdot T + a_0$$

  - a_1: proportional error
  - a_0: offset error

Curve Fitting

- To achieve accurate measurements, we could ***calibrate*** the sensor
  - Measure a range of temperatures with the inexpensive sensor and an accurate sensor
  - Obtain a dataset representing sensor temperature, T_s, as a function of true temperature, T
  - That is, determine a_1 and a_0 such that

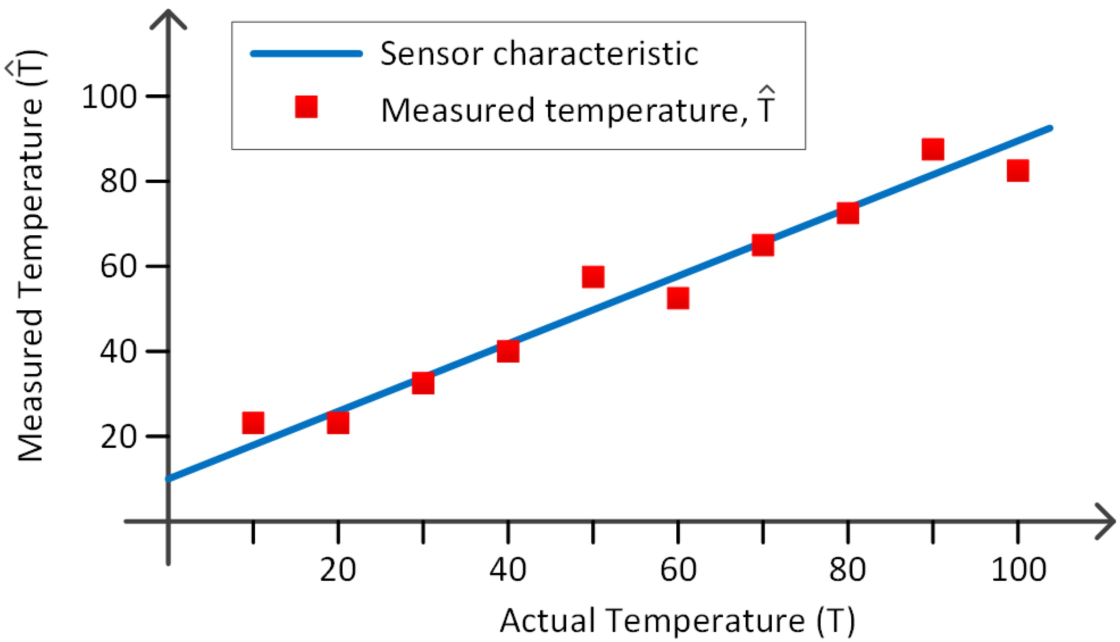
$$T_s = f(T) = a_1 T + a_0$$

- Then, we can map sensor temperature to true temperature

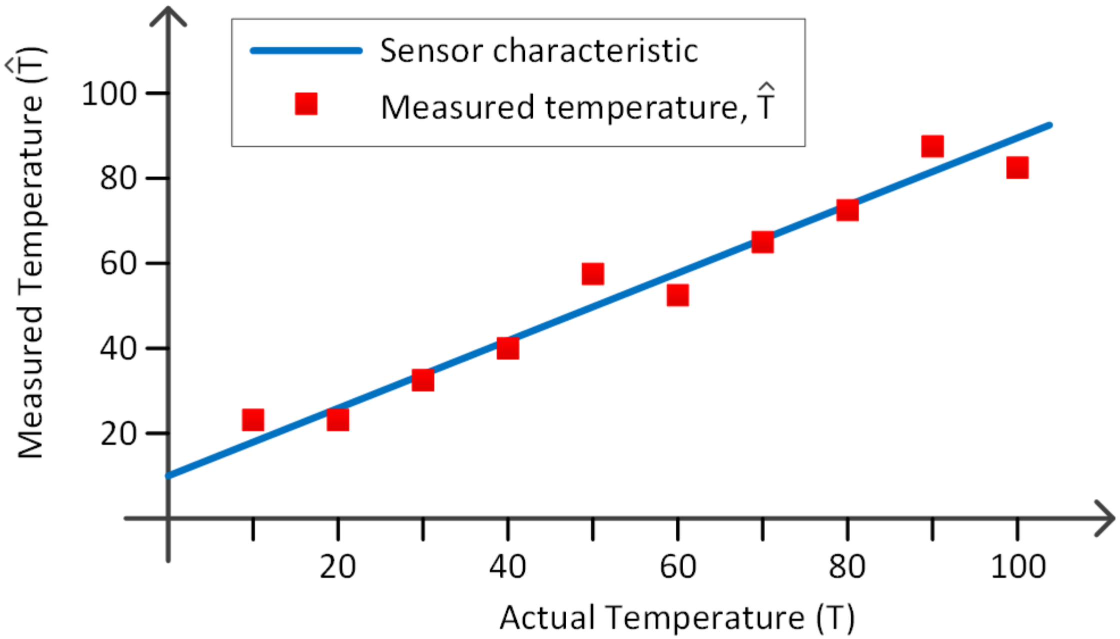
$$T = \frac{T_s}{a_1} - \frac{a_0}{a_1}$$

Curve Fitting

- In practice, there would be two sources of error between actual and measured temperatures
  - Inherent sensor inaccuracy
  - ***Measurement noise***
- Actual ***measured*** data, $\hat{T}$, may look like:



Curve Fitting



- Determine the blue line (a_1 and a_0) that provides the ***best fit*** to the measured data (red squares)
- How do we define "***best fit***"?

Least-Squares Fit

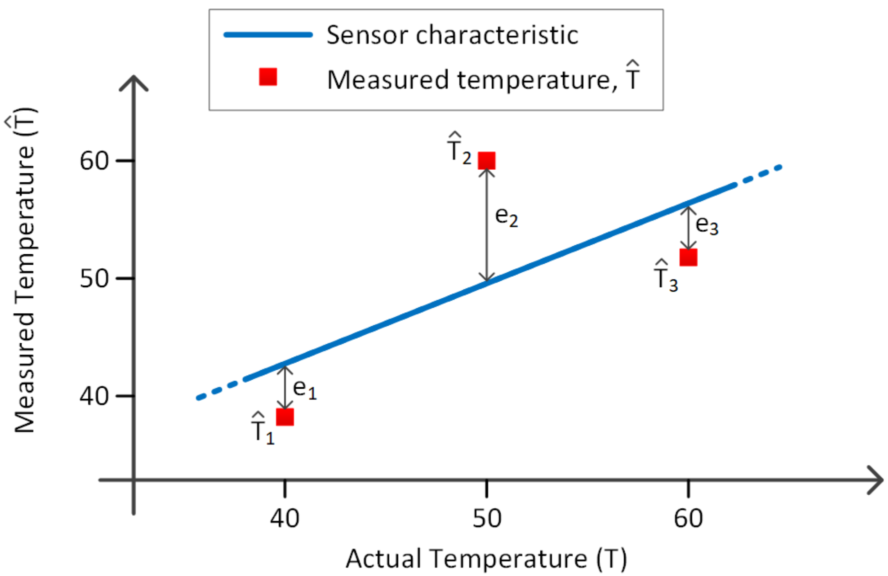
- What constitutes the ***best fit***?
- Want to determine inherent sensor behavior,

$$T_s = a_1 \cdot T + a_0$$

given noisy measurement data,

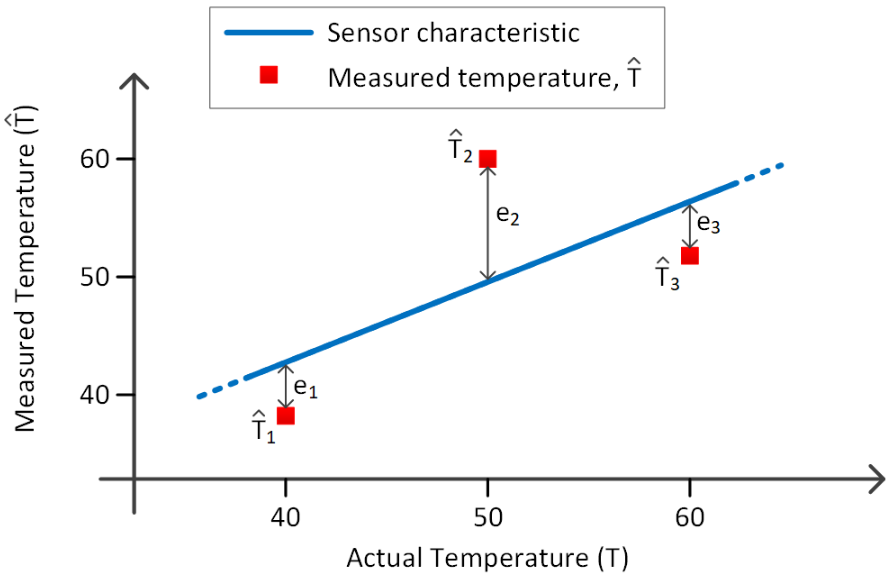
$$\hat{T} = T_s + e$$

where e represents measurement error



Least-Squares Fit

- Errors between data points and the line fit to the data are called ***residuals***
- Best fit criterion:
  - Minimize the ***sum of the squares of the residuals***
  - ***A least-squares fit***



- Minimize:

$$S_r = \sum_i e_i^2 = \sum_i \left[\hat{T}_i - (a_1 T_i + a_0)\right]^2$$

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***

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

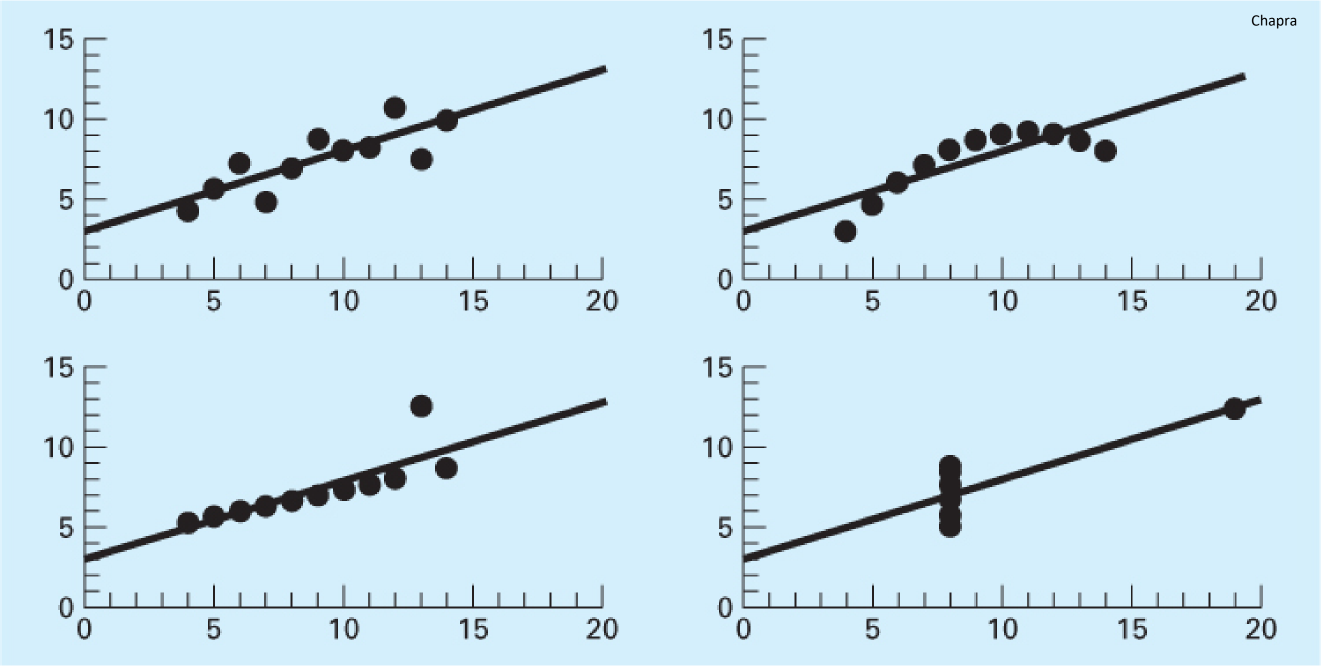
Coefficient of Determination

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

- 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

- 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

Curve Fitting in Python

- So far we have considered fitting a line to data
  - A linear least-squares line fit
- Can also fit other functions to data, e.g.,
  - Higher-order polynomials – quadratic, cubic, etc.
  - Exponentials
  - Sinusoids
  - Power equation, etc.
- We'll look at two curve fitting methods
  - Polynomials:
    - `np.polyfit()`
  - Any other user-specified function:
    - `scipy.optimize.curve_fit()`

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 (m < n)
- p: (m+1)-vector of best-fit polynomial coefficients

- Polynomial coefficients in Python
  - Consider a polynomial created by np.polyfit()

$$y = a_2x^2 + a_1x + a_0$$

  - np.polyfit() would return

$$p = [a_2,\ a_1,\ a_0]$$

Polynomial Evaluation – `np.polyval()`

- n^{th}-order polynomial represented as (n+1)-vector
- For example, the cubic polynomial

$$y = 2x^3 - 8x^2 + 3x - 4$$

would be represented as

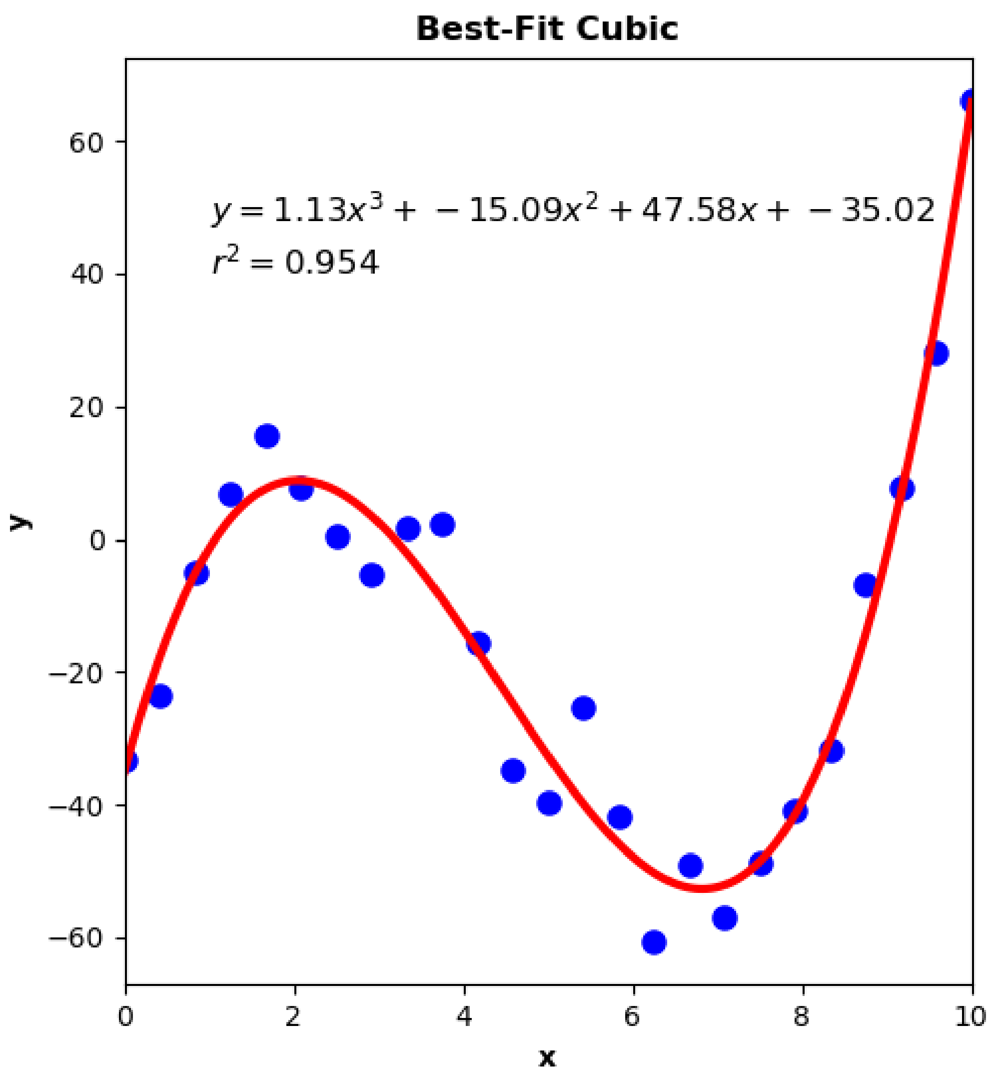
```
p = [2, -8, 3, -4]
```

- Use `np.polyval()` to evaluate that polynomial over a vector of independent variable values

```
y = np.polyval(p, x)
```

  - p: (n+1)-vector of n^{th}-order polynomial coefficients
  - x: vector of independent variable data values
  - y: vector result of evaluating the polynomial at all values in x

Polynomial Fit – Example




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

# %% create noisy dataset
# polynomial with roots at 1, 3, and 9
# y = x**3 - 13*x**2 + 39*x - 27
p = np.poly([1, 3, 9])
x = np.linspace(0, 10, 25)
y = np.polyval(p, x)

# add noise to data
rng = np.random.default_rng(seed=5)

sig = 8
v = rng.normal(scale=sig, size=len(y))

yn = y + v


# %% perform the fit using np.polyfit()
m = 3
pfit = np.polyfit(x, yn, m)


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


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

User-Specified Curves – `curve_fit()`

- To fit a curve other than a polynomial, use `curve_fit()` from the `optimize` module of the `SciPy` package

```
from scipy.optimize import curve_fit
```

```
popt, pcov = curve_fit(f, x, y)
```

  - `f`: function defining the model for the fit
  - `x`: independent variable data values
  - `y`: dependent variable data values
  - `popt`: array of optimal parameter values – the parameters from `f`
  - `pcov`: estimated covariance of parameters in `popt`

Specifying the Model

- Let's say we have voltage data, $v(t)$, at discrete instants of time, t
- And, we'd like to fit an exponential curve to the data

$$v(t) = V_f\left(1 - e^{-\frac{t}{\tau}}\right)$$

  - In other words, we want to determine V_f and τ to best fit the data
- Define the exponential model as a ***standard function***:

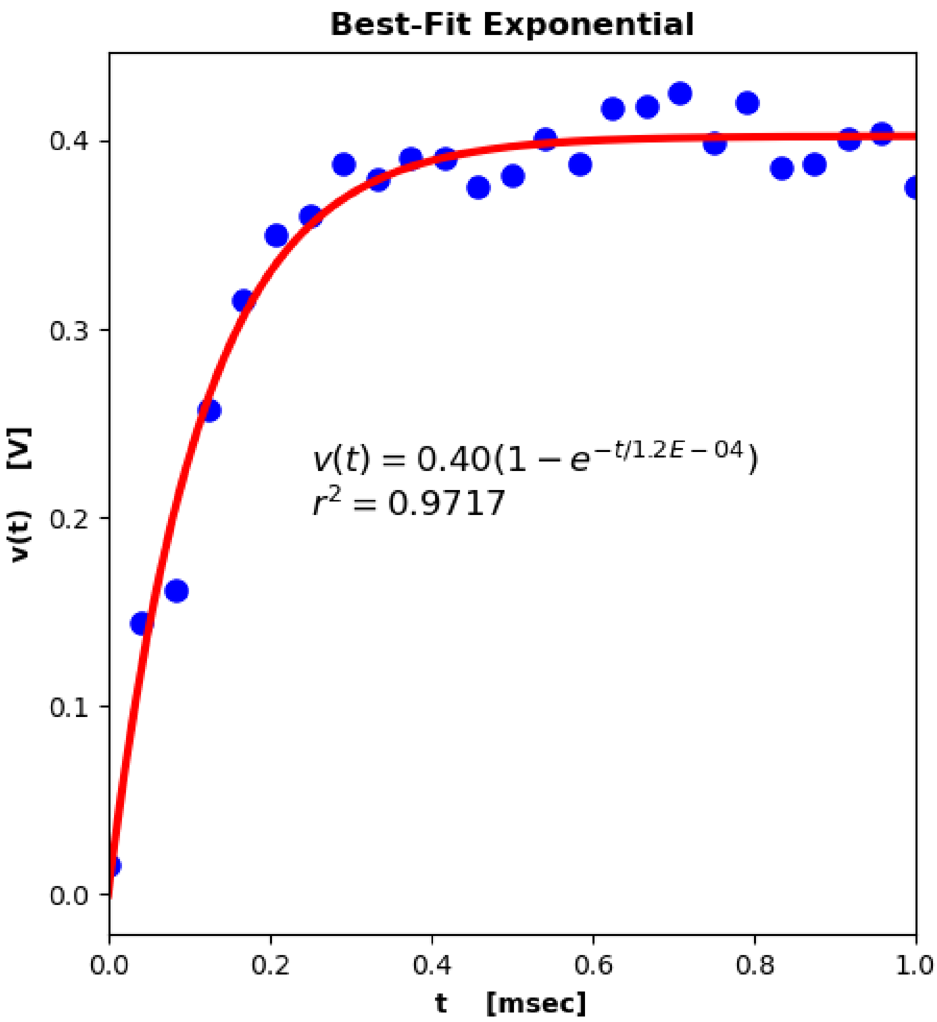
```
def fit_func(t, Vf, tau)
    v = Vf*(1 – np.exp(-t/tau))
    return v
```

- Or as a ***lambda function***:

```
fit_func = lambda t, Vf, tau: Vf*(1 – np.exp(-t/tau))
```

- In either case, the ***independent variable must be the first argument***

Exponential Fit - Example




```python
import numpy as np
from scipy.optimize import curve_fit
from matplotlib import pyplot as plt

# %% create dataset
t = np.linspace(0, 1e-3, 25)
tau = 120e-6
Vf = 400e-3
v = Vf*(1 - np.exp(-t/tau))

rng = np.random.default_rng(seed=6)

sig = 15e-3
n = rng.normal(scale=sig, size=len(t))

vn = v + n


# %% define the fitting function
fit_func = lambda x, a, b: a*(1 - np.exp(-x/b))


# %% perform the fit
popt, pcov = curve_fit(fit_func, t, vn)

print(popt)

Vf_fit = popt[0]
tau_fit = popt[1]


# %% evaluate the fit
tfit = np.linspace(0, t[-1], 2000)
vfit = fit_func(tfit, Vf_fit, tau_fit)
vfitr2 = fit_func(t, Vf_fit, tau_fit)

# coefficient of determination
vbar = np.mean(vn)
St = sum((vn - vbar)**2)
Sr = sum((vn - vfitr2)**2)
r2 = (St - Sr)/St
```

Exercise – Polynomial Curve Fitting

Exercise

- Download the data file, `polyDat.xlsx`, from the Section 9 page on Canvas
- Write a script to do the following:
  - Read the data in using Pandas:

```
df_poly = pd.read_excel('polyDat.xlsx')
x = df_poly['x']
y = df_poly['y']
```

  - Fit an appropriate-order polynomial to the data
  - Plot the data as discrete points along with the best-fit polynomial, plotted as a solid line
- If you have time:
  - Calculate the r^2 value
  - Display the polynomial and the r^2 value on the plot