Motion in One Dimension

#2

Serdar ARITAN

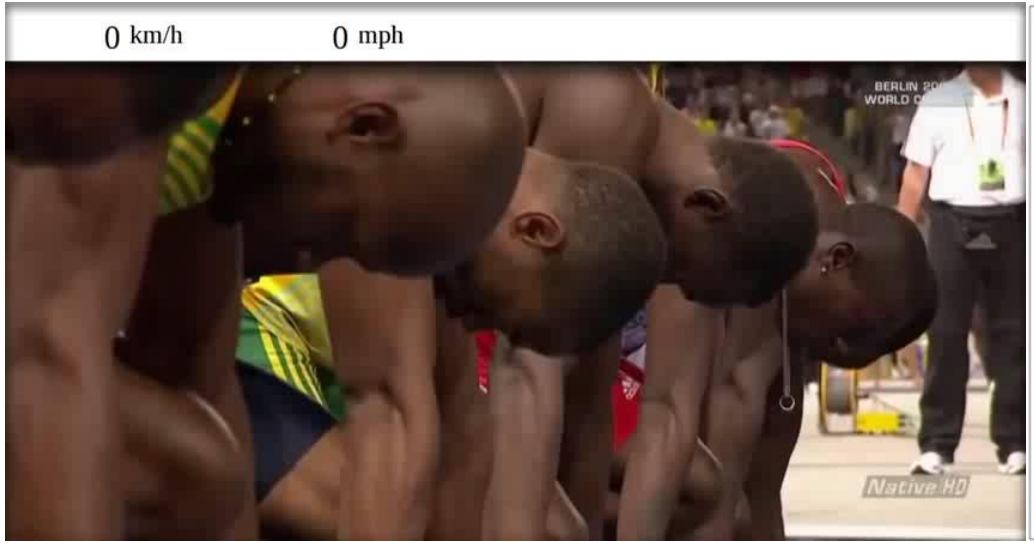
Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

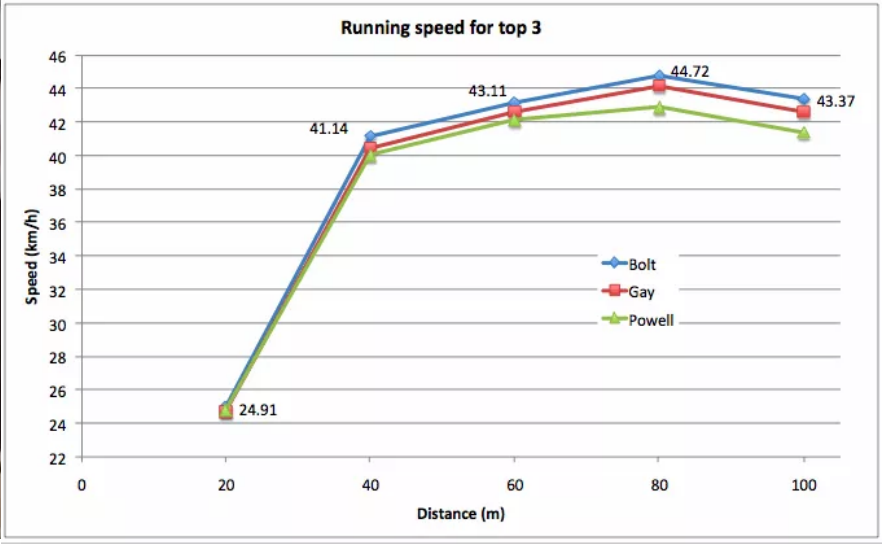
PHYSICS in COMPUTER ANIMATIONS and GAMES

In a fantastic race in the 100m finals of the 2008 Olympic Games in Beijing, Usain Bolt set a new world record of 9.69 s. He even took the time to celebrate his victory over the last 20 meters of the race. But did this affect his winning time? Could he have run even faster?

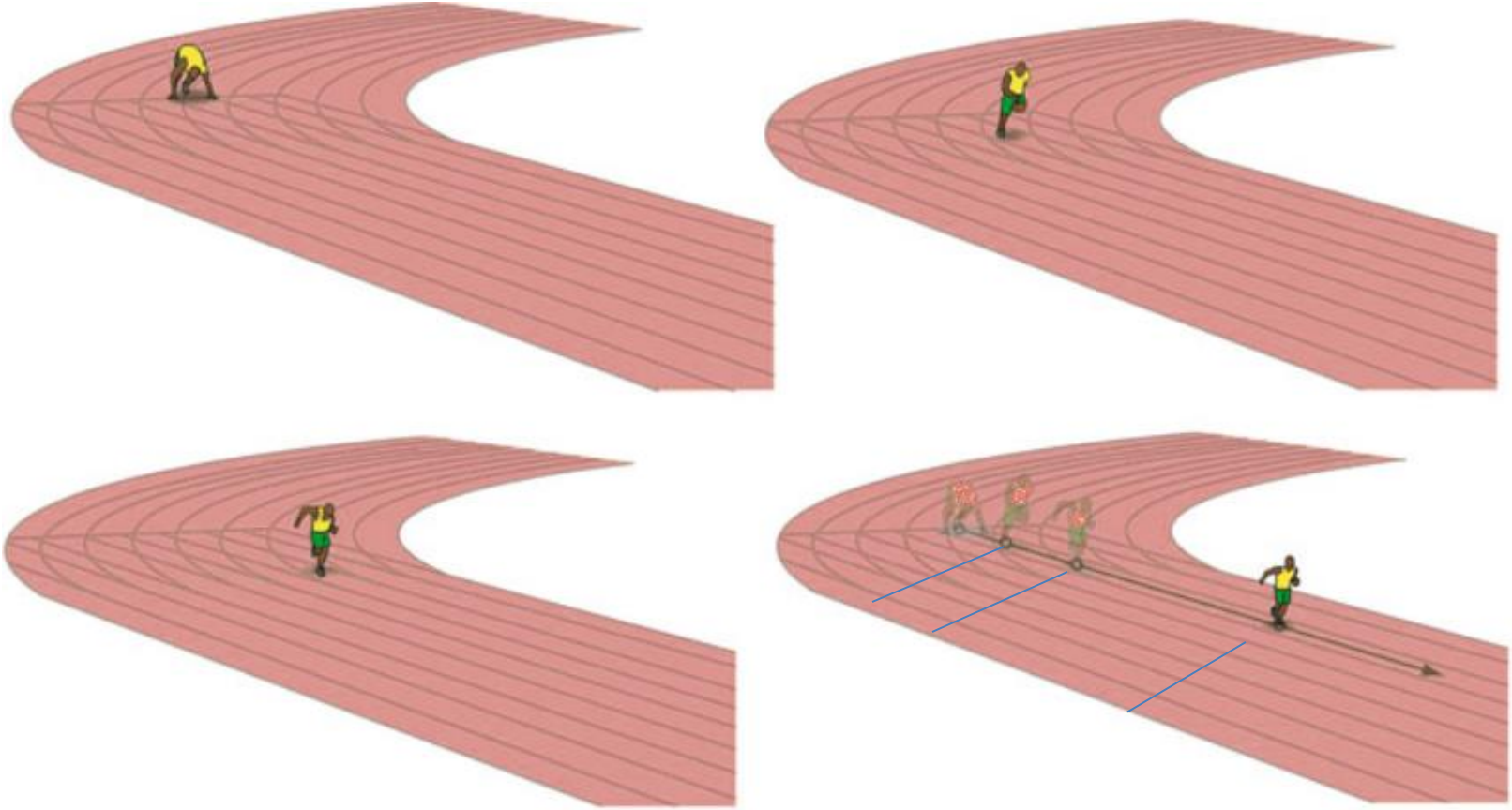
PHYSICS in COMPUTER ANIMATIONS and GAMES

PHYSICS in COMPUTER ANIMATIONS and GAMES

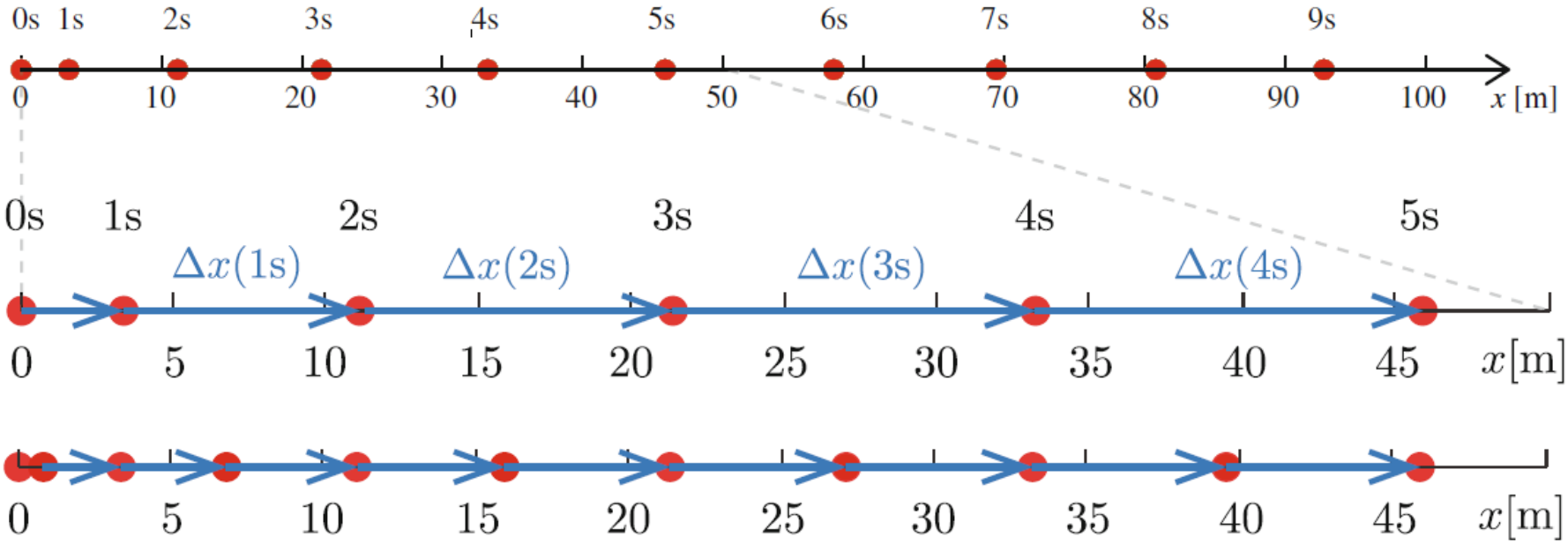




PHYSICS in COMPUTER ANIMATIONS and GAMES



i	0	1	2	3	4	5	6
t_i (s)	0.0	1.0	2.0	3.0	4.0	5.0	6.0
x_i (m)	0.0	3.4	11.1	21.3	33.2	45.8	57.9



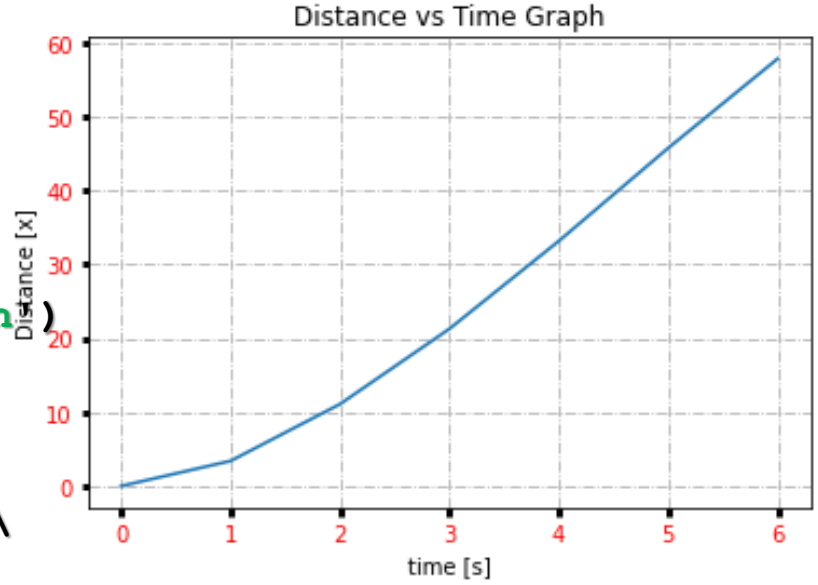
The position $x(t_i)$ of the runner is shown at 1 and 0.5 s intervals.

A **motion diagram** illustrates the motion by a sequence of positions x_i at subsequent times t_i for $i = 0, 1, 2, \ldots$, preferably at times $t_i = t_0 + i\,\Delta t$, where Δt is the time interval.

The motion of an object is described by the **position**, *x(t)*, as a function of time, t, measured in a given reference system. We have chosen to measure the position x along the lane. We call this direction the x-axis. The position x is measured from the starting line, which we call the origin—the point where x is zero. The choice of an origin and an axis is called a reference system. You are free to choose the axes and the origin of your reference system as you like, but we usually try to choose so that measurements become simple.

PHYSICS in COMPUTER ANIMATIONS and GAMES

```
import matplotlib.pyplot as plt
import numpy as np
# Data for plotting
t = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
x = np.array([0.0, 3.4, 11.1, 21.3, 33.2, 45.8, 57.9])

fig, ax = plt.subplots()
ax.plot(t, x)

ax.set(xlabel='time [s]',\
       ylabel='Distance [x]',
       title='Distance vs Time Graph')

ax.grid(True, linestyle='-.')
ax.tick_params(labelcolor='r', \
                  labelsize='medium', \
                  width=3)

plt.show()
```



The **displacement** $\Delta x(t_1)$ over the time interval from $t = t_1$ to $t = t_1 + \Delta t$ is defined as:

$$\Delta x(t_1) = x(t_1 + \Delta t) - x(t_1)$$

The displacement is read directly from the motion diagram as the length of the line from *x(1 s)* to *x(2 s)*. The displacement has a direction—it is the displacement from *x(ti)* to $x(t_i + \Delta t)$.

The **average velocity** from $t = t_1$ to $t = t_1 + \Delta t$ is:

$$\hat{v}(t_i) = \frac{x(t_1 + \Delta t) - x(t_1)}{\Delta t} = \frac{\Delta x(t_1)}{\Delta t}$$

The average velocity has units meters per second, m/s.

Average Velocity

Average Velocity

The **instantaneous velocity** is defined as the time derivative of the position:

$$v(t_i) = \lim_{\Delta t \to 0} \frac{x(t_1 + \Delta t) - x(t_1)}{\Delta t} = \frac{dx}{dt}$$

Notice that the notation *x'(t)* for the derivative that you may be used to from calculus, is not commonly used in physics. Instead, it is denoted the time derivative of a quantity by the placing a dot over it. The velocity is therefore often written as:

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

The velocity may also vary throughout the motion. As we introduced the velocity to characterize the rate of change of position, we introduce the acceleration to characterize the rate of change of the velocity:

The **average acceleration** over a time interval Δt from t to t + Δt is:

$$\hat{a}(t_i) = \frac{v(t_1 + \Delta t) - v(t_1)}{\Delta t} = \frac{\Delta v(t_1)}{\Delta t}$$

The average acceleration has units meters per second squared, m/s^2.

The **instantaneous acceleration** is defined as:

$$a(t_i) = \lim_{\Delta t \to 0} \frac{v(t_1 + \Delta t) - v(t_1)}{\Delta t} = \frac{dv}{dt} = \dot{v}$$

When we use the term acceleration we mean the instantaneous acceleration. The acceleration can be found as the slope of the *v(t)* curve.

$$a(t_i) = \frac{dv}{dt} = \frac{d}{dt}\frac{dx}{dt} = \frac{d^2x}{dt^2} = \ddot{x}$$

Using the dot-notation, we can write this as:

$$a(t) = \dot{v}(t) = \ddot{x}(t)$$

or in shorthand.

$$a = \dot{v} = \ddot{x}$$

Free Fall of an Object: An Experiment by Apollo15 Team

At the end of **the last Apollo 15 moon walk**, Commander David Scott performed a live demonstration for the television cameras. He held out a geologic hammer (1.32 kg) and a feather (0.03 kg) and dropped them at the same time. Because they were essentially in a vacuum, there was no air resistance and the feather fell at the same rate as the hammer, as **Galileo** had concluded hundreds of years before - ***all objects released together fall at the same rate regardless of mass*** *of mass*.

PHYSICS in COMPUTER ANIMATIONS and GAMES

X	Y
56.77	159.06
56.74	159.10
56.70	158.85
56.64	158.37
56.53	157.37
56.47	156.11
56.38	154.33
56.30	152.21
56.21	149.65
56.12	146.74
56.03	143.33
55.94	139.58
55.87	135.39
55.80	130.86
55.72	125.80
55.63	120.48
55.56	114.61
55.50	108.47
55.41	101.82
55.38	94.85
55.27	87.41
55.17	79.67
55.07	71.47
55.00	63.06
54.92	54.11
54.79	44.90
54.73	35.26
54.69	25.35
54.61	15.05
54.58	4.53

Free Fall of an Object: Table with calculated values

i	t_i (s)	y_i (m)	Δy_i (m)	$\bar{v}_i$ (m/s)	$\bar{a}_i$ (m/s^2)
1	0.0	1.60	−0.05	−0.5	
2	0.1	1.55	−0.15	−1.5	−10.0
3	0.2	1.40	−0.24	−2.4	−9.0
4	0.3	1.16	−0.34	−3.4	−10.0
5	0.4	0.82	−0.43	−4.3	−9.0
6	0.5	0.39			

Free Fall of an Object: Table with calculated values

The average velocities can be calculated from the data: For each ***i*** in Table we calculate the average velocity from t_i to t_{i+1} using:

$$v_i = \frac{y_{i+1} - y_i}{\Delta t}$$

The velocities are increasing in magnitude since the ball is accelerating downward. We estimate the average accelerations by

$$a_i = \frac{v_i - v_{i+1}}{\Delta t}$$

The fundamental package for scientific computing with Python

NumPy

NumPy is available at www.numpy.org

Nearly every scientist working in Python draws on the power of NumPy.

NumPy brings the computational power of languages like C and Fortran to Python, a language much easier to learn and use. With this power comes simplicity: a solution in NumPy is often clear and elegant.

Quantum Computing	Statistical Computing	Signal Processing	Image Processing	Graphs and Networks	Astronomy Processes	Cognitive Psychology
QuTiP PyQuil Qiskit	Pandas statsmodels Seaborn	SciPy PyWavelets	Scikit-image OpenCV	NetworkX graph-tool igraph PyGSP	AstroPy SunPy SpacePy	PsychoPy

Bioinformatics	Bayesian Inference	Mathematical Analysis	Simulation Modeling	Multi-variate Analysis	Geographic Processing	Interactive Computing
BioPython Scikit-Bio PyEnsembl	PyStan PyMC3	SciPy SymPy cvxpy FEniCS	PyDSTool	PyChem	Shapely GeoPandas Folium	Jupyter IPython Binder

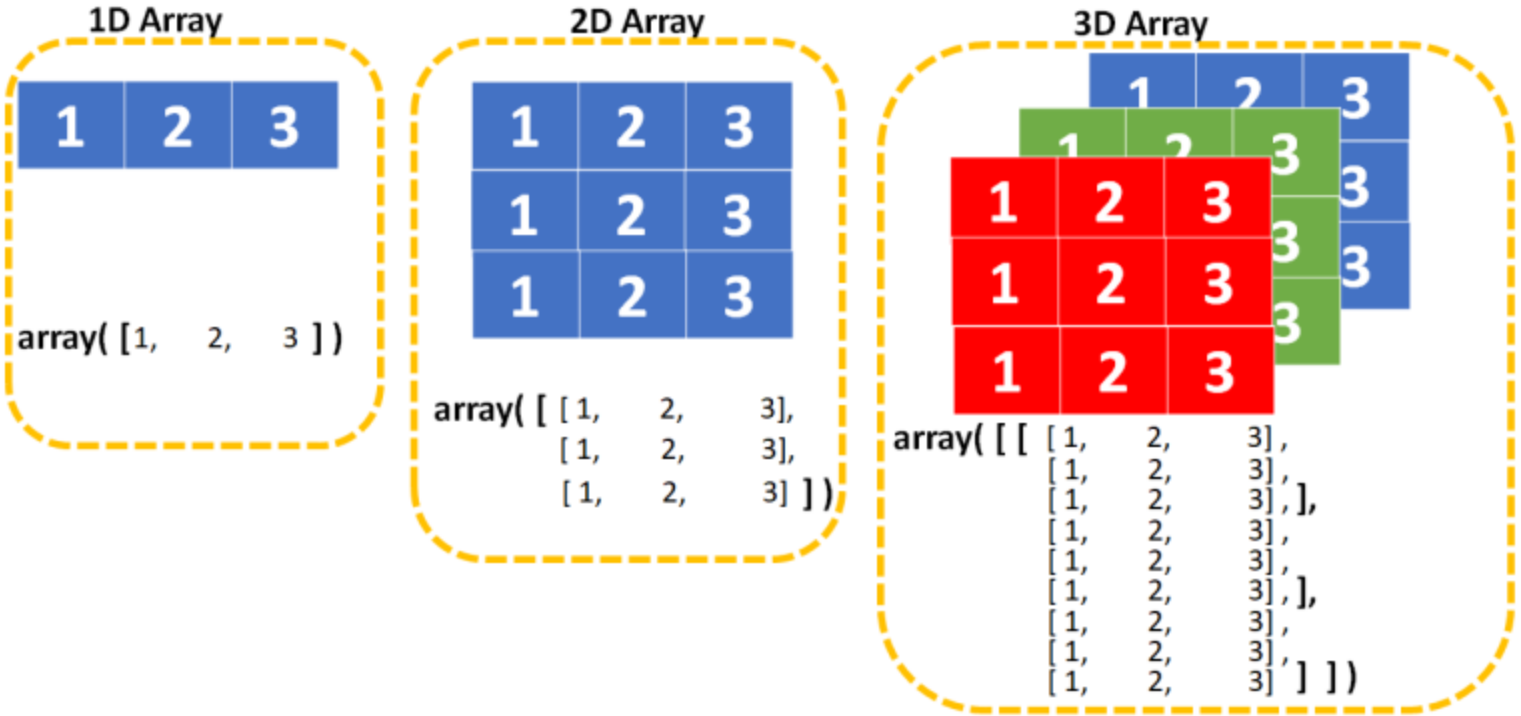
The NumPy Array Object

Vectors, matrices, and arrays of higher dimensions are essential tools in numerical computing. When a computation must be repeated for a set of input values, it is natural and advantageous to represent the data as arrays and the computation in terms of array operations. Computations that are formulated this way are said to be vectorized. Vectorized computing eliminates the need for many explicit loops over the array elements by applying batch operations on the array data. The result is concise and more maintainable code, and it enables delegating the implementation of (e.g., elementwise) array operations to more efficient low-level libraries. Vectorized computations can therefore be significantly faster than sequential element-by-element computations.

For example:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
```

The NumPy Array Object



The NumPy Array Object

2-D, or two-dimensional array, and so on. The NumPy ndarray class is used to represent both matrices and vectors. A vector is an array with a single dimension (there's no difference between row and column vectors), while a matrix refers to an array with two dimensions. For 3-D or higher dimensional arrays, the term **tensor** is also commonly used.

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

We can access the elements in the array using square brackets. When you're accessing elements, remember that indexing in **NumPy starts at 0**.

The NumPy Array Object

The core of the NumPy library is the data structures for representing multidimensional arrays of homogeneous data. Homogeneous refers to all elements in an array having the same data type. The main data structure for multidimensional arrays in NumPy is the ndarray class. In addition to the data stored in the array, this data structure also contains important metadata about the array, such as its shape, size, data type, and other attributes.

Attribute	Description
Shape	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array.
Size	The total number elements in the array.
Ndim	Number of dimensions (axes).
nbytes	Number of bytes used to store the data.
dtype	The data type of the elements in the array.

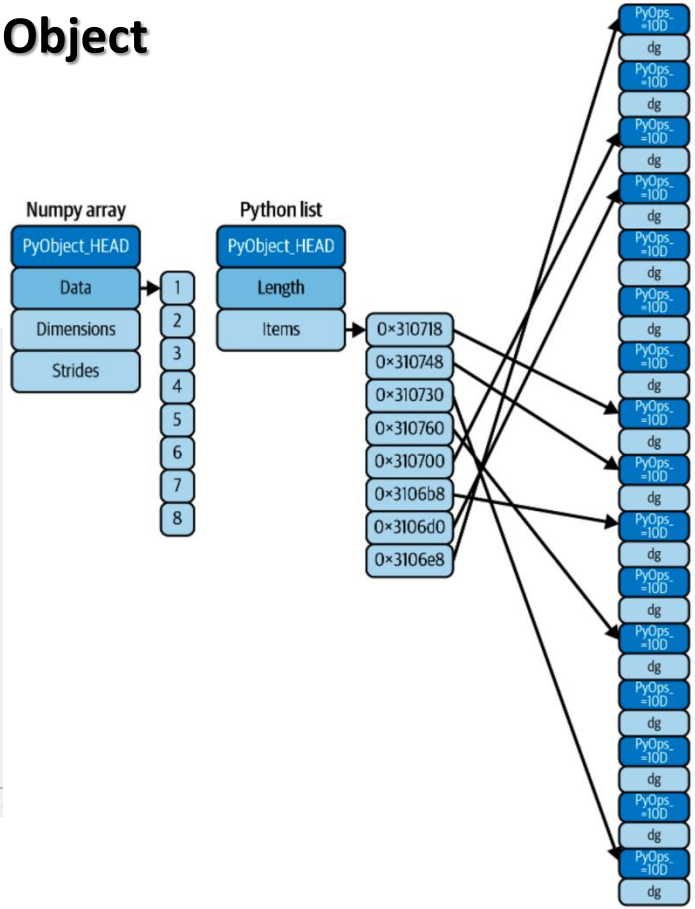
The NumPy Array Object

```
from objbrowser import browse
a = 16
browse(locals())
```

name	path	summary	unicode	repr	type name
__spec__	__spec__	None	None	None	NoneType
a	a	16	16	16	int
__abs__	a.__abs__		<method-wrapper '__abs__' of in...	<method-wrapper '__abs__' of in...	method-wrapp
__add__	a.__add__		<method-wrapper '__add__' of in...	<method-wrapper '__add__' of in...	method-wrapp
__and__	a.__and__		<method-wrapper '__and__' of in...	<method-wrapper '__and__' of in...	method-wrapp
__bool__	a.__bool__		<method-wrapper '__bool__' of i...	<method-wrapper '__bool__' of i...	method-wrapp
__ceil__	a.__ceil__		<built-in method __ceil__ of int o...	<built-in method __ceil__ of int o...	builtin_functior
__class__	a.__class__		<class 'int'>	<class 'int'>	type
__delattr__	a.__delattr__		<method-wrapper '__delattr__' of...	<method-wrapper '__delattr__' of...	method-wrapp
__dir__	a.__dir__		<built-in method __dir__ of int o...	<built-in method __dir__ of int o...	builtin_functior
__divmod__	a.__divmod__		<method-wrapper '__divmod__' ...	<method-wrapper '__divmod__' ...	method-wrapp
__doc__	a.__doc__	int([x]) -> integer↵int(x, base=1...	int([x]) -> integer↵int(x, base=1...	"int([x]) -> integer\nint(x, base=...	str
__eq__	a.__eq__		<method-wrapper '__eq__' of int ...	<method-wrapper '__eq__' of int ...	method-wrapp
__float__	a.__float__		<method-wrapper '__float__' of i...	<method-wrapper '__float__' of i...	method-wrapp
__floor__	a.__floor__		<built-in method __floor__ of int ...	<built-in method __floor__ of int ...	builtin_functior
__floordiv__	a.__floordiv__		<method-wrapper '__floordiv__' ...	<method-wrapper '__floordiv__' ...	method-wrapp
__format__	a.__format__		<built-in method __format__ of i...	<built-in method __format__ of i...	builtin_functior
__ge__	a.__ge__		<method-wrapper '__ge__' of int ...	<method-wrapper '__ge__' of int ...	method-wrapp
__getattrib...	a.__getattribute__		<method-wrapper '__getattribute...	<method-wrapper '__getattribute...	method-wrapp
__getnewar...	a.__getnewargs__		<built-in method __getnewargs__...	<built-in method __getnewargs__...	builtin_functior

Details

path: __file__



The NumPy Array Object

```
>>> import numpy as np
# To create a NumPy array, you can use the function np.array()

>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> type(data)
numpy.ndarray
>>> data
array([[1, 2],
       [3, 4],
       [5, 6]])
```

	0	1
0	1	2
1	3	4
2	5	6

The NumPy Array Object

```
>>> data.ndim
2
>>> data.shape
(3, 2)
>>> data.size
6
>>> data.dtype
dtype('int32')
>>> data.nbytes
24
```

	0	1
0	1	2
1	3	4
2	5	6

The NumPy Array Object

```
>>> np.zeros(2)
array([0., 0.])
>>> np.ones(2)
array([1., 1.])
>>> np.arange(4)
array([0, 1, 2, 3])
# specify the first number, last number, and the step size
>>> np.arange(2, 9, 2)
array([2, 4, 6, 8])
```

The NumPy Array Object

```
>>> # Create an empty array with 2 elements
>>> np.empty(2)
array([ 3.14, 42.  ])  # may vary
```

The function empty creates an array whose initial content is random and depends on the state of the memory. The reason to use empty over zeros (or something similar) is speed - just make sure to fill every element afterwards!

You can also use np.linspace() to create an array with values that are spaced linearly in a specified interval:

```
>>> np.linspace(0, 10, num=5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

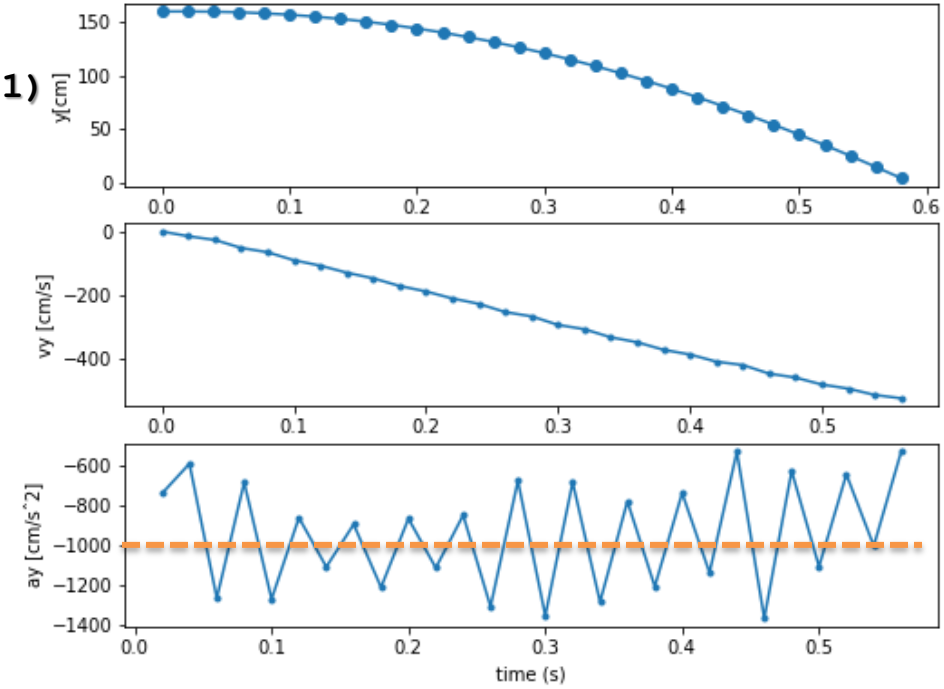
PHYSICS in COMPUTER ANIMATIONS and GAMES

Free Fall of an Object: How to calculate velocity and accelerations

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

x, y = np.loadtxt('Dropxy.dat', delimiter=',', usecols=[0,1],unpack=True )
n = len(x)
delta_t = 1/50       # 50 frames per second
time = np.linspace(0, (n-1)*delta_t, n)

vy = np.zeros(n-1, float)
for i in range(n-1):
    vy[i] = (y[i+1] - y[i])/delta_t

ay = np.zeros(n-2, float)
for i in range(n-2):
    ay[i] = (vy[i+1] - vy[i])/delta_t
```

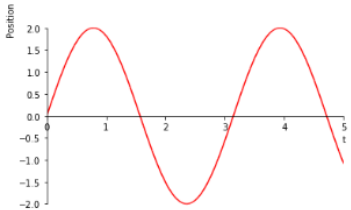
Free Fall of an Object: How to calculate velocity and accelerations

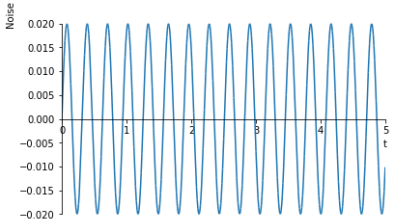
```
fig, (ax1,ax2,ax3)= plt.subplots(3, 1)

ax1.plot(time, y, 'o-')
ax1.set_ylabel('y[cm]')

ax2.plot(time[0:n-1],vy, '.-')
ax2.set_ylabel('vy [cm/s]')

ax3.plot(time[1:n-1],ay, '.-')
ax3.set_ylabel('ay [cm/s^2]')
ax3.set_xlabel('time (s)')

plt.show()
```



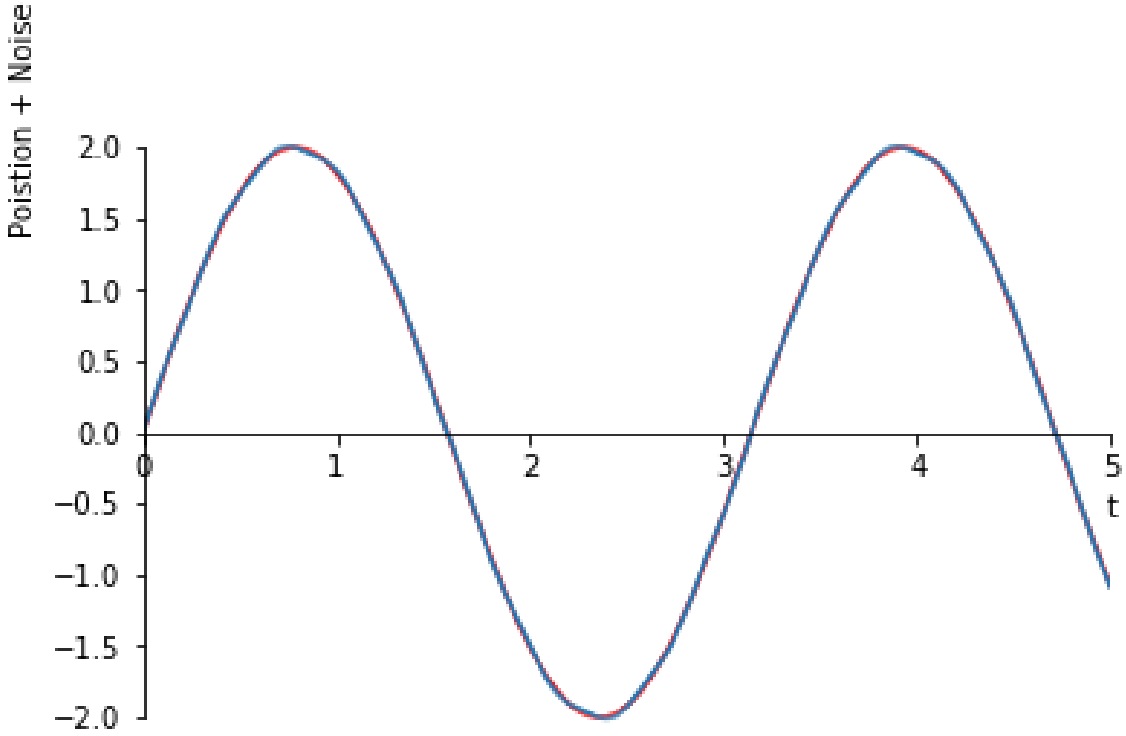
Calculation of Acceleration is Prone to Error

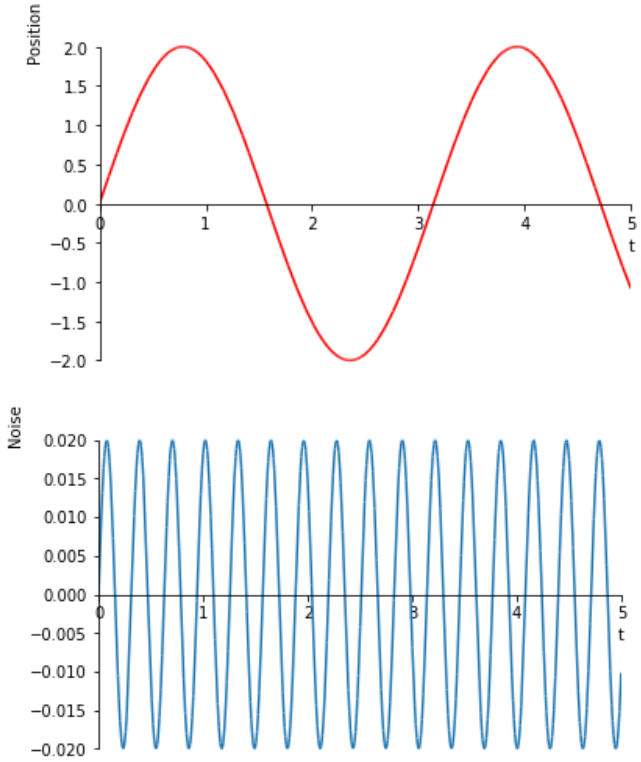

```
from sympy import *
v, a, t = symbols('v a t', real = True)
init_printing(use_unicode = True)
## Position
y = 2*sin(2*t)
print('Ideal Position : ', y)
plot(y, (t, 0, 5), line_color = 'r', ylabel = 'Position')
noise = 0.02*sin(20*t)
print('Noise : ', noise)
plot(noise, (t, 0, 5), ylabel = 'Noise')
yexp = y + noise
print('Position with noise : ', yexp)
yPlot = plot(y,yexp,(t,0,5), ylabel='Poistion+Noise', show = False)
yPlot[0].line_color = 'r'
yPlot.show()
```



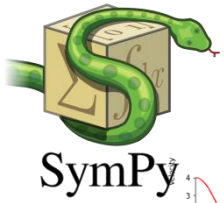


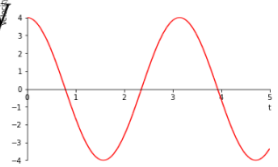
Calculation of Acceleration is Prone to Error



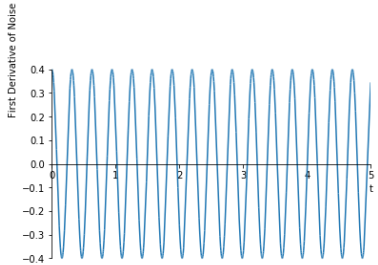


Calculation of Acceleration is Prone to Error

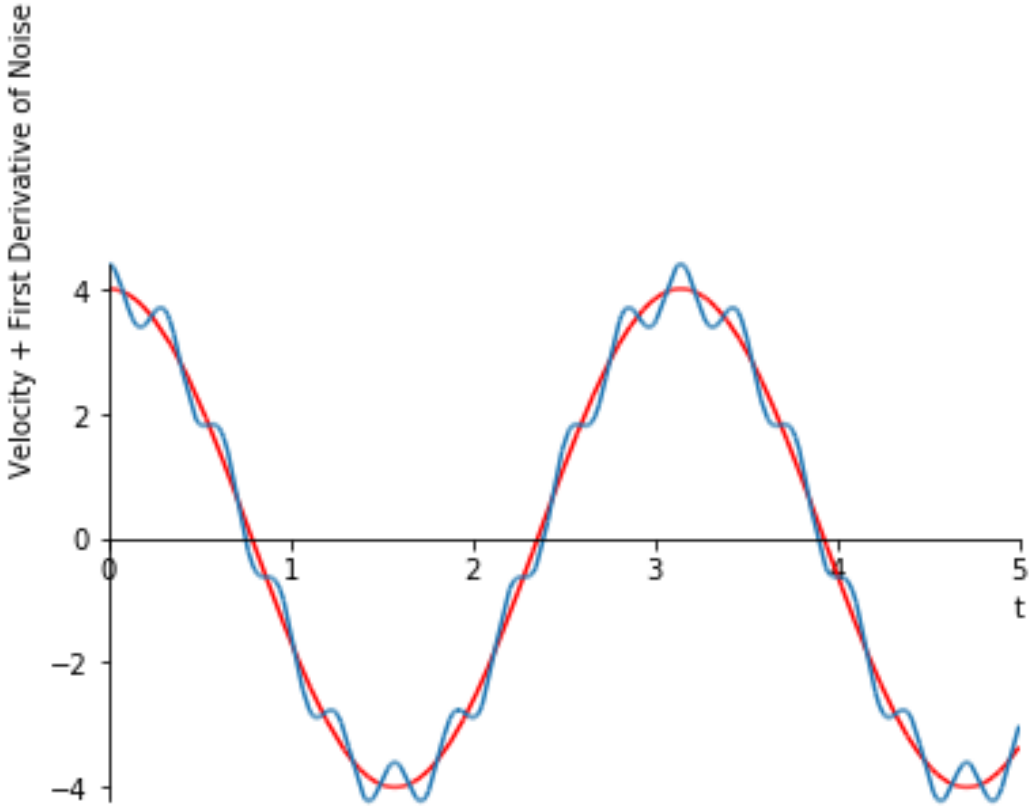


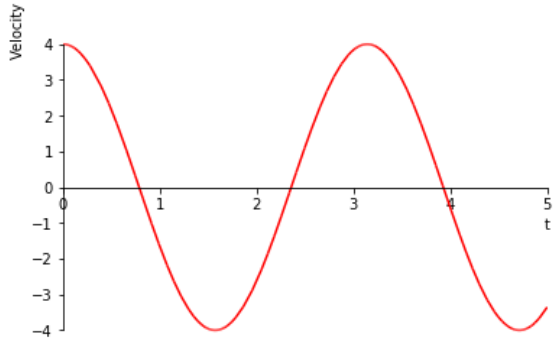


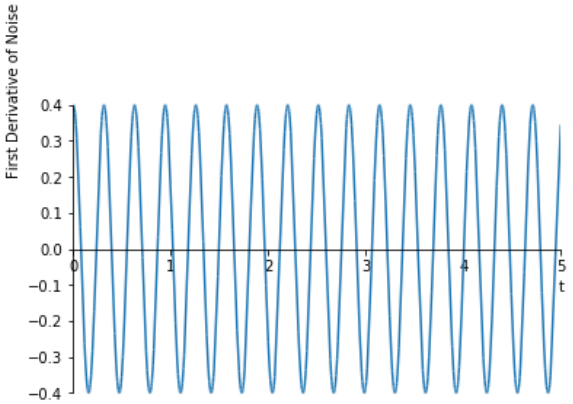
```
## Velocity
v = diff(y, t) # dy/dt
print('Velocity : ', v)
plot(v, (t, 0, 5), line_color = 'r', ylabel='Velocity')
v_noise = diff(noise, t) # dnoise/dt
plot(v_noise, (t, 0, 5), ylabel = 'First Derivative of Noise')
vexp = diff(yexp, t)
vPlot = plot(v,vexp,(t,0,5),ylabel='Velocity+First Derivative of Noise',
show = False)
vPlot[0].line_color = 'r'
vPlot.show()
```



PHYSICS in COMPUTER ANIMATIONS and GAMES



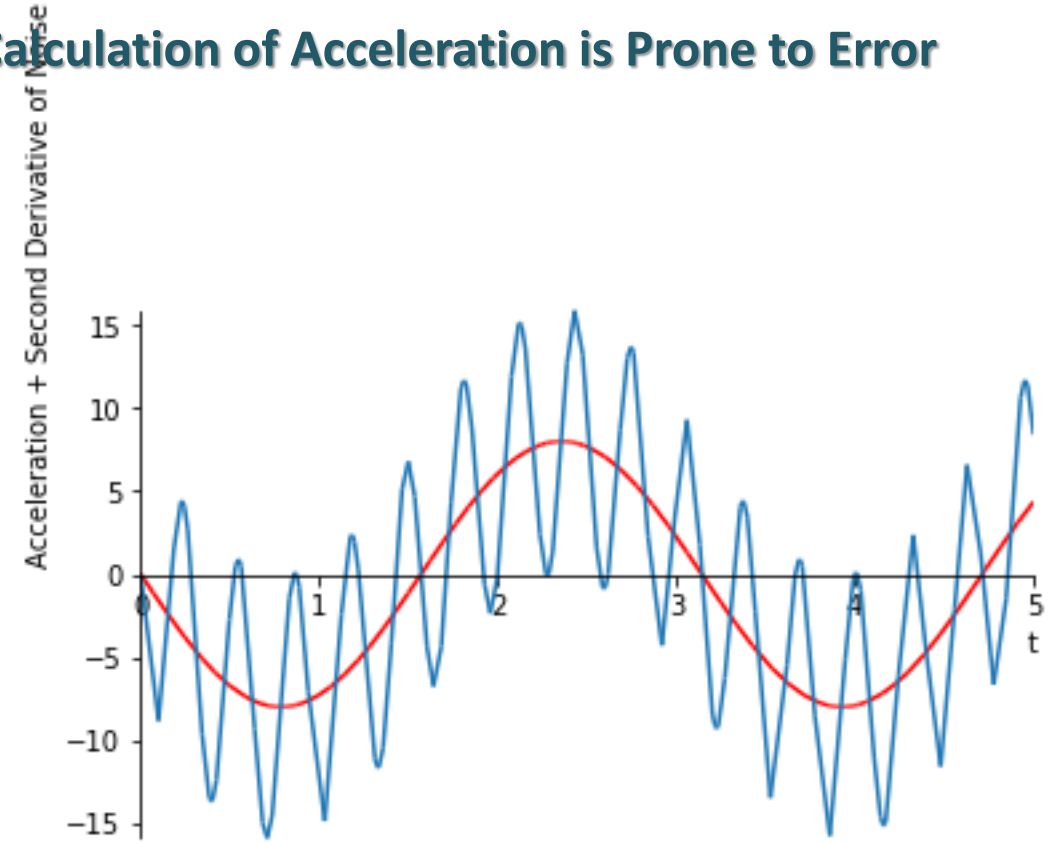


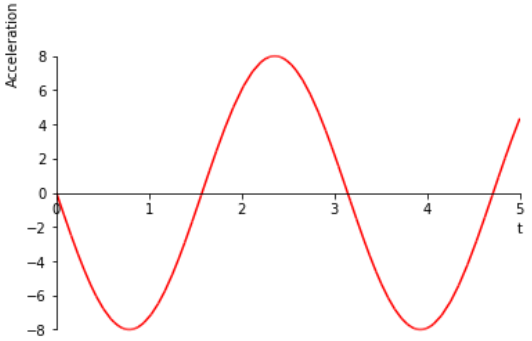


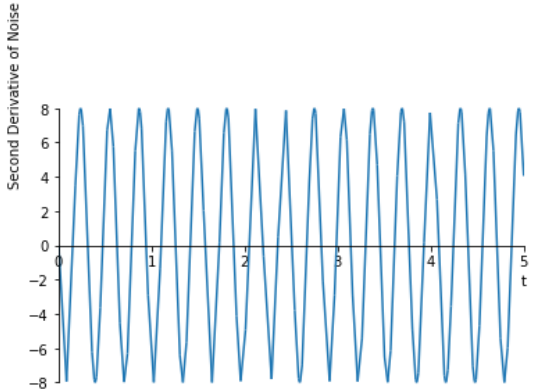
Calculation of Acceleration is Prone to Error


```
## Acceleration
a = diff(v, t) # dv/dt
print('Acceleration : ', a)
plot(a, (t, 0, 5), line_color = 'r', ylabel='Acceleration')
a_noise = diff(noise, t, 2) #d2noise/dt^2
plot(a_noise, (t, 0, 5), ylabel='Second Derivative of Noise')
aexp = diff(vexp, t)
aPlot = plot(a, aexp, (t, 0, 5), ylabel='Acceleration + Second
Derivative of Noise', show = False)
#change the color of aexp
aPlot[0].line_color = 'r'
aPlot.show()
```

Calculation of Acceleration is Prone to Error







We can compare the results better by studying the velocities and accelerations. In the mathematical model, we know ***y(t)***, and we can calculate the instantaneous velocity and acceleration by applying the definitions directly. The velocity of the ball is defined as:

$$v = \frac{dy}{dt} = \frac{d}{dt}\left(y_0 - \frac{1}{2}gt^2\right) = -gt$$

Similarly, the acceleration is defined as

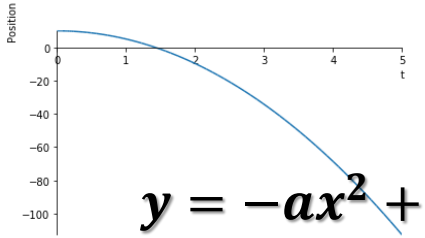
$$a = \frac{dv}{dt} = \frac{d}{dt}(-gt) = -g = 9.8\, m/s^2$$

Refreshing Calculus with SymPy

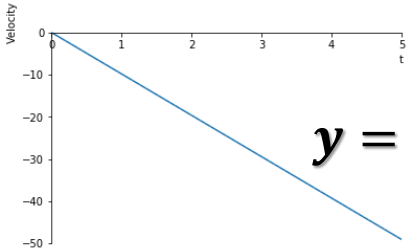
```
from sympy import *
v, a, t, g, y0 = symbols('v a t g y0', real = True)
init_printing(use_unicode = True)

y = y0 - 0.5*g*t**2
print('Position : ', y)

v = diff(y, t)
print('Velocity : ', v)

a = diff(v, t)
print('Acceleration : ', a)
# Prints
Position :  -0.5*g*t**2 + y0
Velocity :  -1.0*g*t
Acceleration :  -1.0*g
```

Refreshing Calculus with SymPy

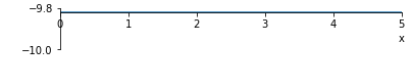
```
# Let's do some experiment
# Leave the ball from 10 meters on the earth
yexp = y.subs({y0:10, g: 9.8182})
plot(yexp, (t, 0, 5), ylabel='Position')
vexp = diff(yexp, t)
plot(vexp, (t, 0, 5), ylabel='Velocity')
aexp = diff(vexp, t)
plot(aexp, (t, 0, 5), ylabel='Acceleration')
```



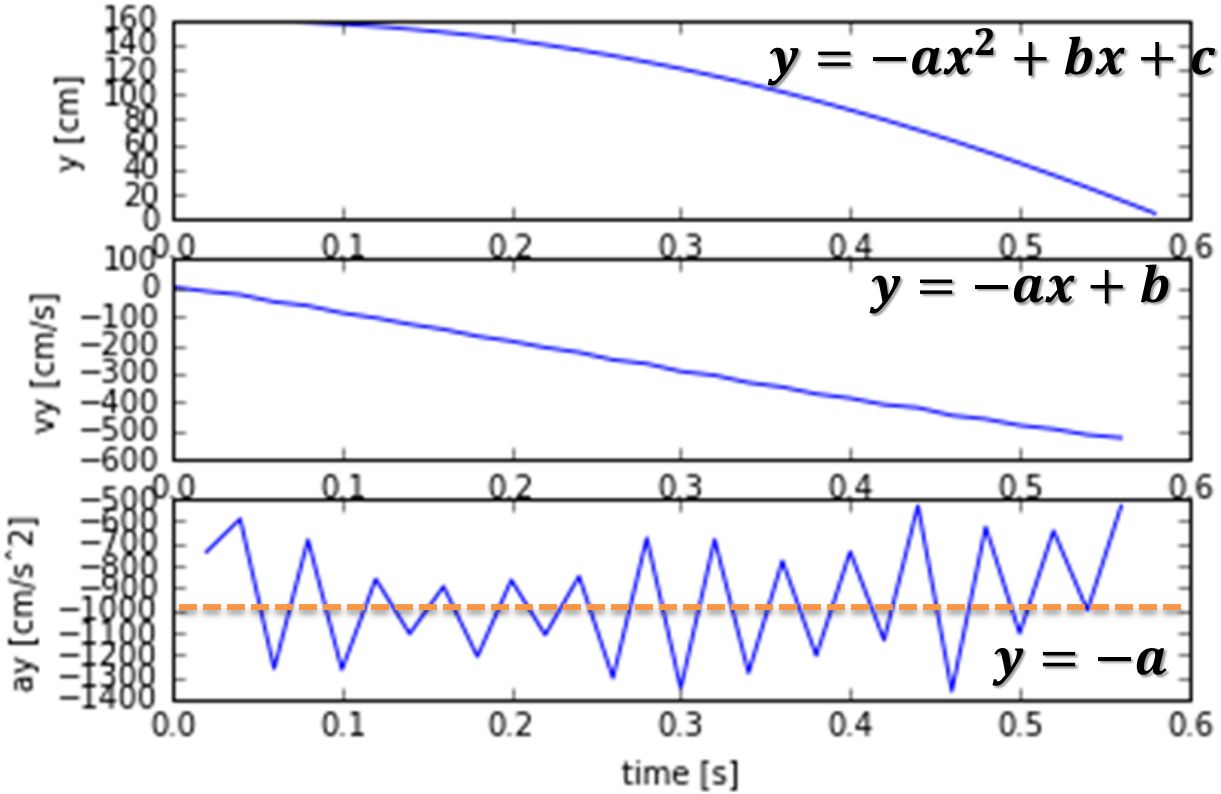
$$y = -ax^2 + bx + c$$



$$y = -ax + b$$



$$y = -a$$



$v = -gt$

$a = -g$

Free Fall of an Object: Mathematical Model

There is a **mathematical model** for the motion of a falling tennis ball when there is no air resistance

$$y(t) = y_0 - \frac{1}{2}gt^2$$

$$y(t) = 1.5908 - \frac{1}{2}9.81t^2$$

$$y(t) = -4.905t^2 + 1.5908$$

where $g = 9.8 m/s^2$ is a constant and y_0 is the position of the tennis ball at $t = 0$ s. Let us see how this model matches up with the observed data. We calculate the position of the ball for various times. From the experimental data, we see that $y(0) = 1.5905$ m. We use Python as a calculator to find the positions for all the times in table with a single line of code:

Curve fitting: Fitting lines and polynomial functions to data points

Curve fitting is the process of constructing a curve, or mathematical function, that has the best fit to a series of data points, possibly subject to constraints. Curve fitting can involve either interpolation, where an exact fit to the data is required, or smoothing.

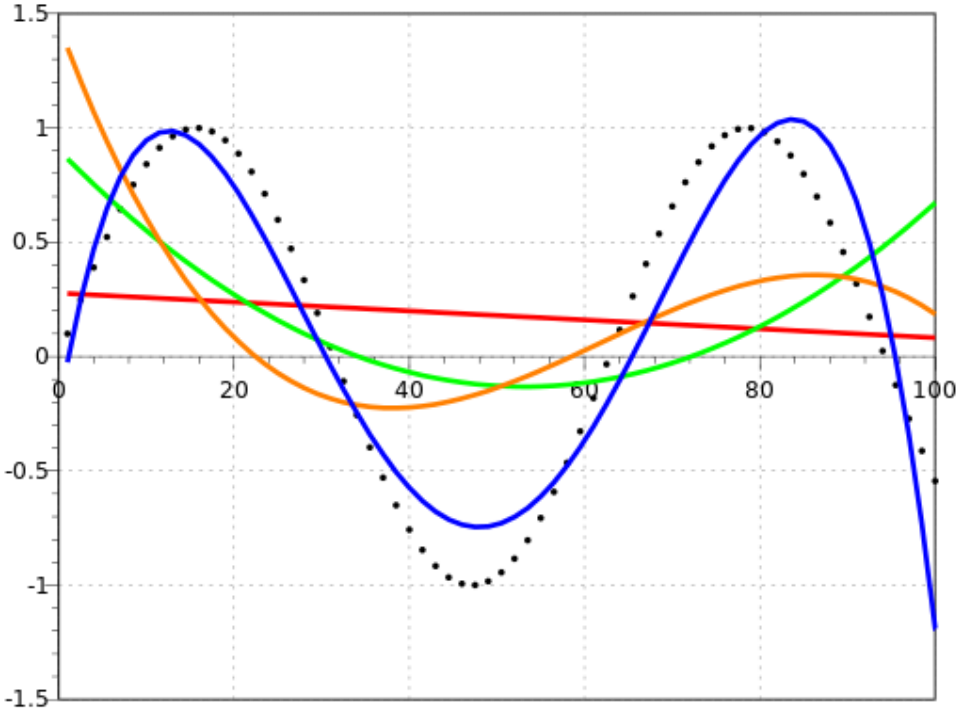
Most commonly, one fits a function of the form *y=f(x)*.

First degree polynomial equation $y = ax + b$

Second degree polynomial equation $y = ax^2 + bx + c$

Third degree polynomial equation $y = ax^3 + bx^2 + cx + d$

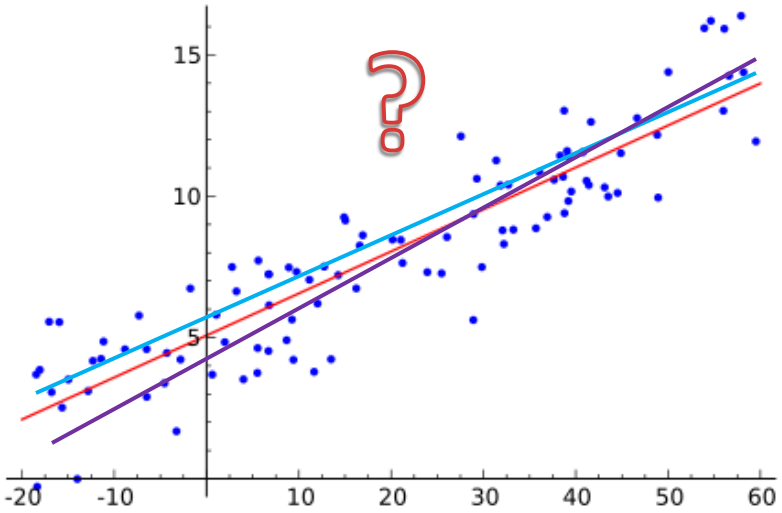
Curve fitting: Fitting lines and polynomial functions to data points



Polynomial curves fitting points generated with a sine function.

Red line is a first degree polynomial, green line is second degree, orange line is third degree and blue is fourth degree.

Goodness of fit: Coefficient of determination

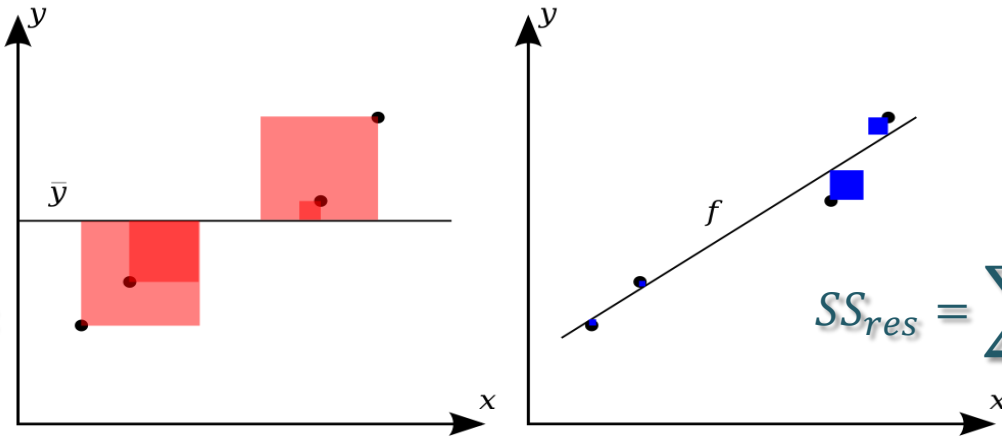


The goodness of fit of a statistical model describes how well it fits a set of observations. Measures of goodness of fit typically summarize the discrepancy between observed values and the values expected under the model in question.

The coefficient of determination, denoted R^2 or r^2 and pronounced ***R squared***, is a number that indicates how well data fit a statistical model.

Goodness of fit: Coefficient of determination

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$



$$SS_{tot} = \sum_i (y_i - \bar{y})^2$$

$$SS_{res} = \sum_i (y_i - f_i)^2$$

The better the linear regression (on the right) fits the data in comparison to the simple average (on the left), the closer the value of R^2 is to 1. The areas of the blue squares represent the squared residuals with respect to the linear regression. The areas of the red squares represent the squared residuals with respect to the average value.

Curve fitting: Python Applications


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

y = np.loadtxt('Dropxy.dat', delimiter=',', usecols=[1],unpack=True )
n = len(y)
delta_t = 1/50       # 50 frames per second
time = np.linspace(0, (n-1)*delta_t, n)
# Fit 2nd Order Polynomial
y_poly = np.polyfit(time, y, 2)
# Evaluate 2nd Order Polynomial with time values
y_val = np.polyval(y_poly, time)

correlation = np.corrcoef(time, y)[0,1]
print('R squared = {:2.2f}'.format(correlation**2))
```

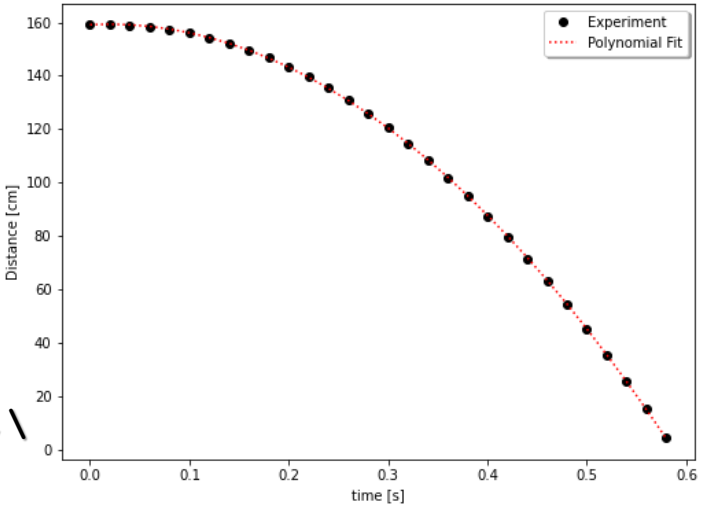
X	Y
56.77	159.06
56.74	159.10
56.70	158.85
56.64	158.37
56.53	157.37
56.47	156.11
56.38	154.33
56.30	152.21
56.21	149.65
56.12	146.74
56.03	143.33
55.94	139.58
55.87	135.39
55.80	130.86
55.72	125.80
55.63	120.48
55.56	114.61
55.50	108.47
55.41	101.82
55.38	94.85
55.27	87.41
55.17	79.67
55.07	71.47
55.00	63.06
54.92	54.11
54.79	44.90
54.73	35.26
54.69	25.35
54.61	15.05
54.58	4.53

R^2 = R squared = 0.92

Curve fitting: Python Applications

NumPy

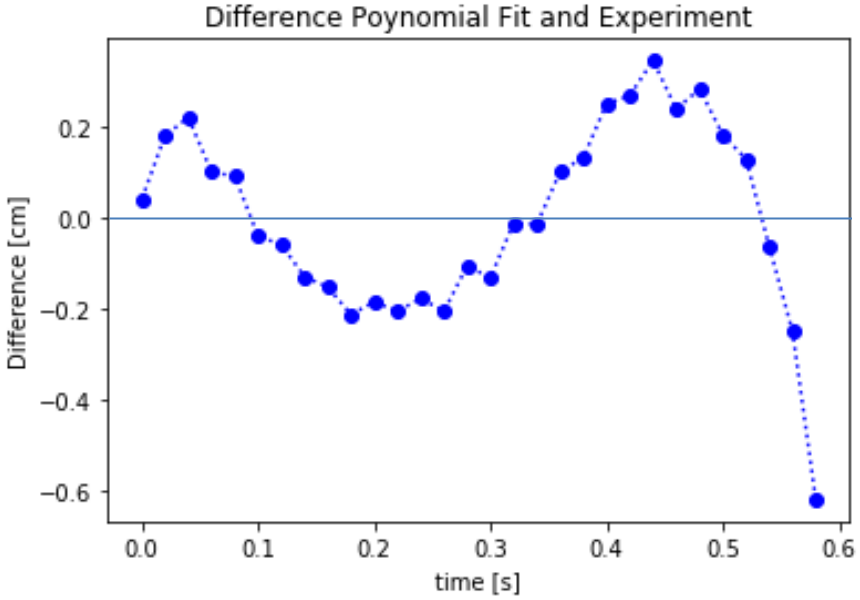
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
l1,l2= ax.plot(time,y,'ko',time,y_val,':r')
ax.set_ylabel('y [cm]')
ax.set_xlabel('time [s]')
ax.legend((l1, l2),\
          ('Experiment','Polynomial Fit'),\
           loc='upper right', shadow=True)
plt.show()
```




```
R squared = 0.92
```

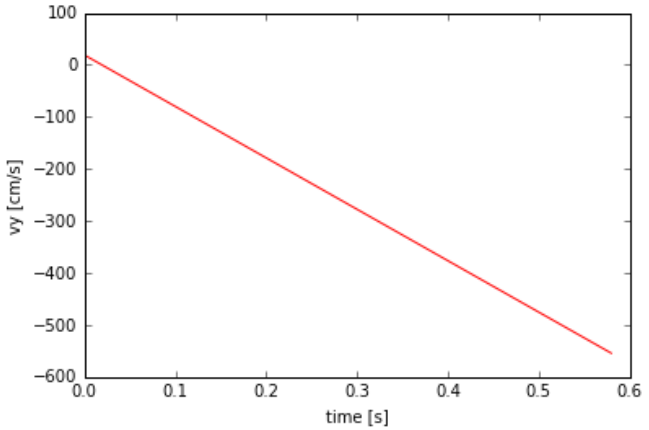
Curve fitting: Python Applications

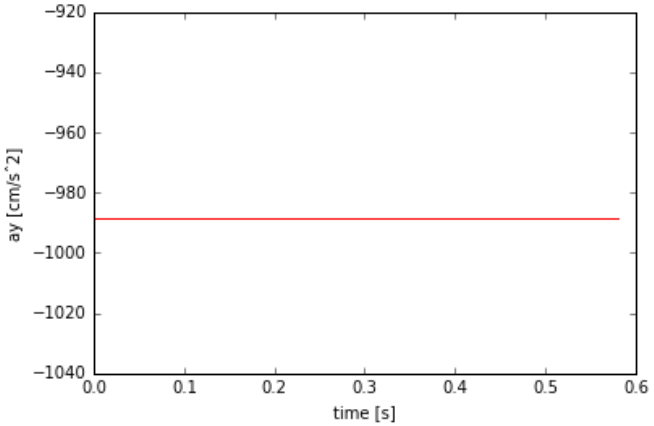
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
ax.plot(time, y_val - y, 'o:b')
plt.suptitle('Difference Poynomial Fit and Experiment')
ax.set_ylabel('Difference [cm]')
ax.set_xlabel('time [s]')
plt.show()
```



Curve fitting: Python Applications

```
v_poly = np.polyder(y_poly)
a_poly = np.polyder(y_poly, 2)
v_val = np.polyval(v_poly, time)
a_val = np.polyval(a_poly, time)

fig, ax = plt.subplots()
ax.plot(time, v_val, '-r')
ax.set_ylabel('vy [cm/s]')
ax.set_xlabel('time [s]')
plt.show()

fig, ax = plt.subplots()
ax.plot(time, a_val, '-r')
ax.set_ylabel('ay [cm/s^2]')
ax.set_xlabel('time [s]')
plt.show()
```





Modelling and Simulation

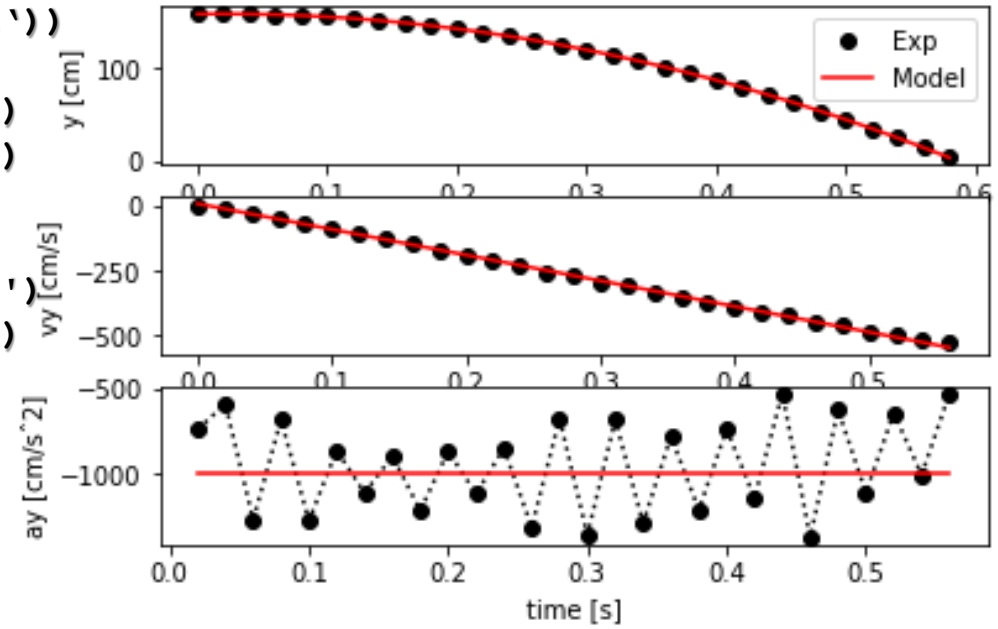
```
import matplotlib.pyplot as plt
import numpy as np

y_exp = np.loadtxt('Dropxy.dat', delimiter=',', usecols=[1],unpack=True )
n = len(y_exp )
delta_t = 1/50       # 50 frames per second
time = np.linspace(0, (n-1)*delta_t, n)

# Fit 2nd Order Polynomial
y_poly = np.polyfit(time, y_exp, 2)
# Evaluate 2nd Order Polynomial with time values
y_val = np.polyval(y_poly, time)
```

Modelling and Simulation

```
vy_exp = np.zeros(n-1, float)
vy_pol = np.zeros(n-1, float)

for i in range(n-1):
    vy_exp[i] = (y_exp[i+1] - y_exp[i])/delta_t
    vy_pol[i] = (y_val[i+1] - y_val[i])/delta_t

ay_exp = zeros(n-2, float)
ay_pol = zeros(n-2, float)

for i in range(n-2):
    ay_exp[i] = (vy_exp[i+1] - vy_exp[i])/delta_t
    ay_pol[i] = (vy_pol[i+1] - vy_pol[i])/delta_t
```

Modelling and Simulation

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1)
l1, l2 = ax1.plot(time, y_exp, 'ok', time, y_val, '-r')
ax1.set_ylabel('y [cm]')
ax1.legend((l1, l2),('Exp', 'Model'))

ax2.plot(time[0:n-1], vy_exp, 'ok')
ax2.plot(time[0:n-1], vy_pol, '-r')
ax2.set_ylabel('vy [cm/s]')

ax3.plot(time[1:n-1], ay_exp, 'ok:')
ax3.plot(time[1:n-1], ay_pol, '-r')
ax3.set_xlabel('time [s]')
ax3.set_ylabel('ay [cm/s^2]')
plt.show()
```



Class Study : Free Fall of an Object in the Moon and the Jupiter

The acceleration due to gravity on the surface of the Moon is 1.62 m/s^2

The acceleration due to gravity on the surface of the Jupiter is 24.79m/s^2