The Scipy Package

January 18, 2026

[1]:
```
#importing libraries
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
```

****Basic Optimization****

[2]:
```
from scipy.optimize import minimize
```

One variable Optimization

Minimize $f(x) = (x - 3)^2$

[3]:
```
def f(x):
    return (x-3)**2
```

[4]:
```
f = lambda x: (x-3)**2
```

[5]:
```
min_f = minimize(f,2)
```

[6]:
```
min_f
```

[6]:
```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 5.551437397369767e-17
        x: [ 3.000e+00]
      nit: 2
      jac: [-4.325e-13]
 hess_inv: [[ 5.000e-01]]
     nfev: 6
     njev: 3
```

[7]:
```
min_f.x
```

[7]:
```
array([2.99999999])
```

[8]:
```
min_f.fun
```

[8]:
```
5.551437397369767e-17
```

```
[10]: #help(minimize)
```

Minimize $(x-1)^2 + (y-2.5)^2$

```
[15]: min_2v = minimize(lambda x: (x[0]-1)**2 + (x[1]-2.5)**2, (1,2))
      min_2v
```

```
[15]:   message: Optimization terminated successfully.
        success: True
         status: 0
            fun: 1.1102230246251565e-16
              x: [ 1.000e+00  2.500e+00]
            nit: 2
            jac: [ 0.000e+00  0.000e+00]
       hess_inv: [[ 1.000e+00  7.451e-09]
                  [ 7.451e-09  5.000e-01]]
           nfev: 12
           njev: 4
```

Exercise: Minimize $x^5 - 5x^3 - 20x + 5$

Multivariable with constraints

Minimize

$$f(x,y) = (x-1)^2 + (y-2.5)^2$$

subject to

$$x - 2y + 2 \geq 0$$

$$-x + 2y + 6 \geq 0$$

$$-x + 2y + 2 \geq 0$$

$$x \geq 0, \quad y \geq 0$$

```
[18]: f = lambda x: (x[0]-1)**2 + (x[1]-2.5)**2
```

```
[23]: constraints = ({'type':'ineq', 'fun': lambda x: x[0]-2*x[1]+2},
                     {'type':'ineq', 'fun': lambda x: -x[0]-2*x[1]+6},
                     {'type':'ineq', 'fun': lambda x: -x[0]+2*x[1]+2})
```

```
[24]: bounds = ((0,None),(0,None))
```

```
[35]: min_mul_f = minimize(f,x0 = (0,0), constraints = constraints,bounds= bounds)
```

```
[36]: min_mul_f
```

```
[36]:      message: Optimization terminated successfully
           success: True
            status: 0
               fun: 0.8000000000000044
                 x: [ 1.400e+00  1.700e+00]
```

```
         nit: 4
         jac: [ 8.000e-01 -1.600e+00]
        nfev: 12
        njev: 4
 multipliers: [ 8.000e-01  0.000e+00  0.000e+00]
```

Finding Roots of Polynomials

[38]:
```
from scipy.optimize import root
```

Find root of $x + \cos x = 0$.

[42]:
```
my_root = root(lambda x: x + np.cos(x), 0)
```

[43]:
```
my_root
```

[43]:
```
 message: The solution converged.
 success: True
  status: 1
     fun: [ 0.000e+00]
       x: [-7.391e-01]
  method: hybr
    nfev: 11
    fjac: [[-1.000e+00]]
       r: [-1.674e+00]
     qtf: [-2.668e-13]
```

[44]:
```
my_root.x
```

[44]:
```
array([-0.73908513])
```

Find root of $x^3 - x - 1 = 0$

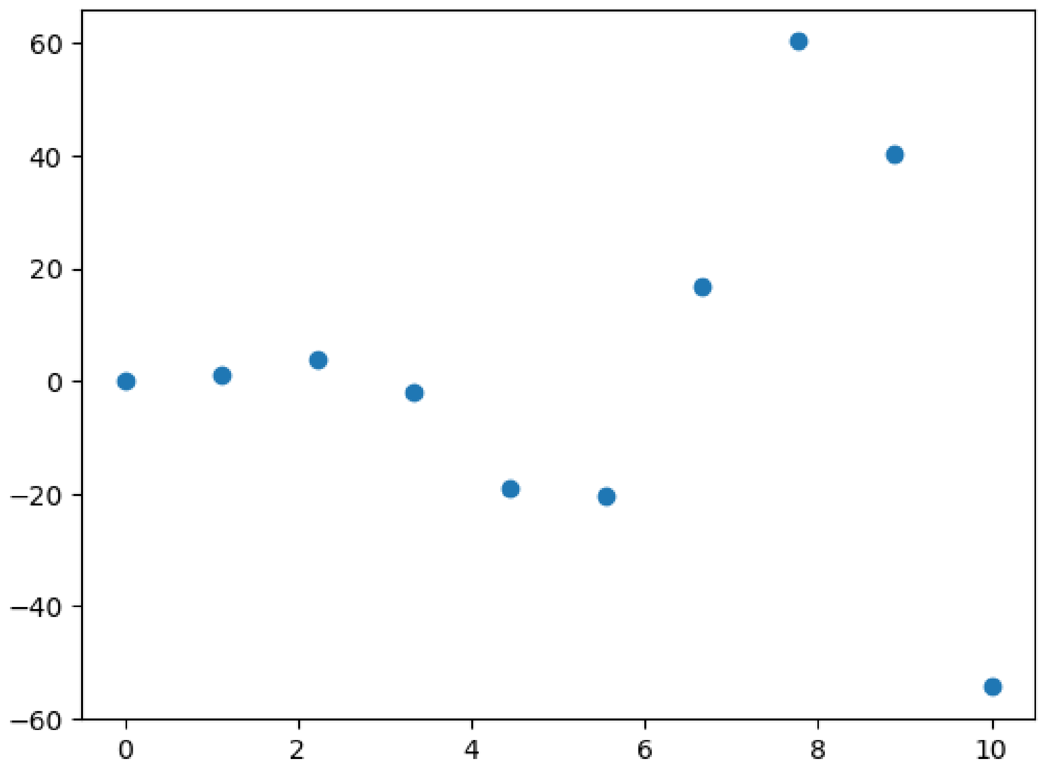
Interpolation

[49]:
```
from scipy.interpolate import interp1d
```

[47]:
```
#Sample data
x = np.linspace(0,10,10)
y = x**2*np.sin(x)
```

[48]:
```
plt.scatter(x,y)
```

[48]:
```
<matplotlib.collections.PathCollection at 0x12fd7b0e0>
```



[50]:
```
help(interp1d)
```

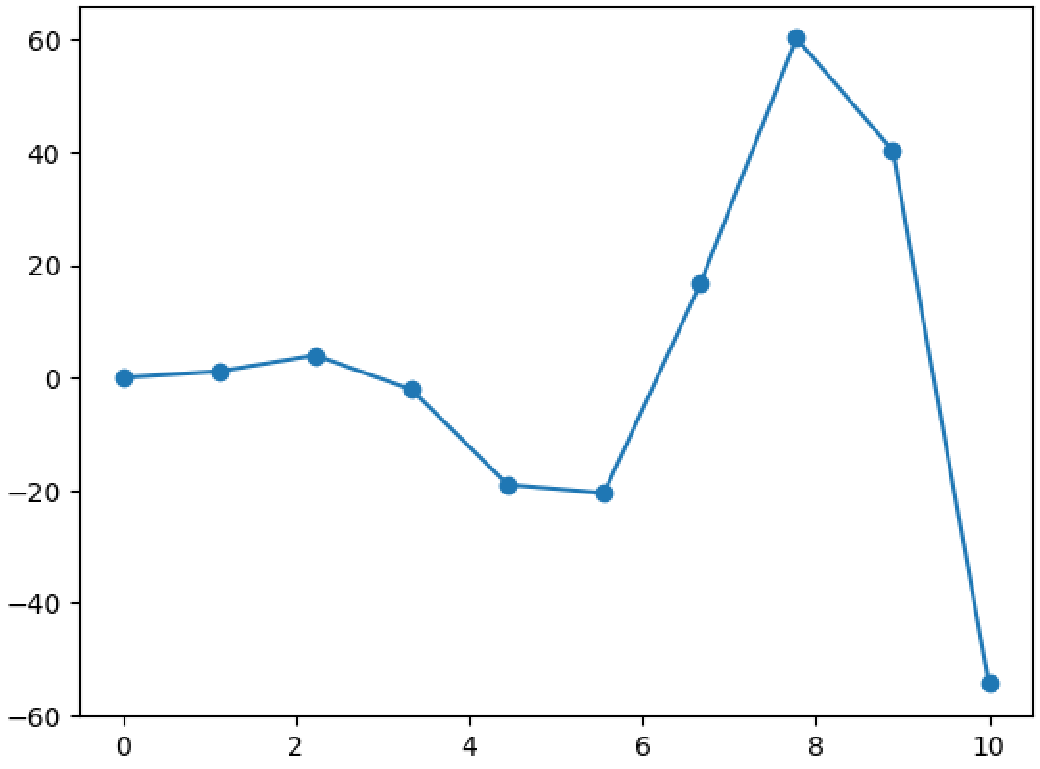
```
Help on class interp1d in module scipy.interpolate._interpolate:

class interp1d(scipy.interpolate._polyint._Interpolator1D)
 |  interp1d(
 |      x,
 |      y,
 |      kind='linear',
 |      axis=-1,
 |      copy=True,
 |      bounds_error=None,
 |      fill_value=nan,
 |      assume_sorted=False
 |  )
 |
 |  Interpolate a 1-D function (legacy).
 |
 |  .. legacy:: class
 |
 |      For a guide to the intended replacements for `interp1d` see
 |      :ref:`tutorial-interpolate_1Dsection`.
 |
 |  `x` and `y` are arrays of values used to approximate some function f:
 |  ``y = f(x)``. This class returns a function whose call method uses
 |  interpolation to find the value of new points.
 |
 |  Parameters
 |  ----------
 |  x : (npoints, ) array_like
 |      A 1-D array of real values.
 |  y : (…, npoints, …) array_like
 |      A N-D array of real values. The length of `y` along the interpolation
 |      axis must be equal to the length of `x`. Use the ``axis`` parameter
 |      to select correct axis. Unlike other interpolators, the default
 |      interpolation axis is the last axis of `y`.
 |  kind : str or int, optional
 |      Specifies the kind of interpolation as a string or as an integer
 |      specifying the order of the spline interpolator to use.
 |      The string has to be one of 'linear', 'nearest', 'nearest-up', 'zero',
 |      'slinear', 'quadratic', 'cubic', 'previous', or 'next'. 'zero',
 |      'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of
 |      zeroth, first, second or third order; 'previous' and 'next' simply
 |      return the previous or next value of the point; 'nearest-up' and
 |      'nearest' differ when interpolating half-integers (e.g. 0.5, 1.5)
 |      in that 'nearest-up' rounds up and 'nearest' rounds down. Default
 |      is 'linear'.
 |  axis : int, optional
 |      Axis in the ``y`` array corresponding to the x-coordinate values. Unlike
 |      other interpolators, defaults to ``axis=-1``.
 |  copy : bool, optional
 |      If ``True``, the class makes internal copies of x and y. If ``False``,
 |      references to ``x`` and ``y`` are used if possible. The default is to
copy.
 |  bounds_error : bool, optional
 |      If True, a ValueError is raised any time interpolation is attempted on
 |      a value outside of the range of x (where extrapolation is
 |      necessary). If False, out of bounds values are assigned `fill_value`.
 |      By default, an error is raised unless ``fill_value="extrapolate"``.
 |  fill_value : array-like or (array-like, array_like) or "extrapolate",
optional
 |      - if a ndarray (or float), this value will be used to fill in for
 |        requested points outside of the data range. If not provided, then
 |        the default is NaN. The array-like must broadcast properly to the
 |        dimensions of the non-interpolation axes.
 |      - If a two-element tuple, then the first element is used as a
 |        fill value for ``x_new < x[0]`` and the second element is used for
 |        ``x_new > x[-1]``. Anything that is not a 2-element tuple (e.g.,
 |        list or ndarray, regardless of shape) is taken to be a single
 |        array-like argument meant to be used for both bounds as
|       ``below, above = fill_value, fill_value``. Using a two-element tuple
|       or ndarray requires ``bounds_error=False``.
|
|       .. versionadded:: 0.17.0
|     - If "extrapolate", then points outside the data range will be
|       extrapolated.
|
|       .. versionadded:: 0.17.0
|  assume_sorted : bool, optional
|     If False, values of `x` can be in any order and they are sorted first.
|     If True, `x` has to be an array of monotonically increasing values.
|
|  Attributes
|  ----------
|  fill_value
|
|  Methods
|  -------
|  __call__
|
|  See Also
|  --------
|  splrep, splev
|     Spline interpolation/smoothing based on FITPACK.
|  UnivariateSpline : An object-oriented wrapper of the FITPACK routines.
|  interp2d : 2-D interpolation
|
|  Notes
|  -----
|  Calling `interp1d` with NaNs present in input values results in
|  undefined behaviour.
|
|  Input values `x` and `y` must be convertible to `float` values like
|  `int` or `float`.
|
|  If the values in `x` are not unique, the resulting behavior is
|  undefined and specific to the choice of `kind`, i.e., changing
|  `kind` will change the behavior for duplicates.
|
|
|  Examples
|  --------
|  >>> import numpy as np
|  >>> import matplotlib.pyplot as plt
|  >>> from scipy import interpolate
|  >>> x = np.arange(0, 10)
|  >>> y = np.exp(-x/3.0)
|  >>> f = interpolate.interp1d(x, y)
```

```
|
|  >>> xnew = np.arange(0, 9, 0.1)
|  >>> ynew = f(xnew)   # use interpolation function returned by `interp1d`
|  >>> plt.plot(x, y, 'o', xnew, ynew, '-')
|  >>> plt.show()
|
|  Method resolution order:
|      interp1d
|      scipy.interpolate._polyint._Interpolator1D
|      builtins.object
|
|  Methods defined here:
|
|  __init__(
|      self,
|      x,
|      y,
|      kind='linear',
|      axis=-1,
|      copy=True,
|      bounds_error=None,
|      fill_value=nan,
|      assume_sorted=False
|  )
|      Initialize a 1-D linear interpolation class.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables
|
|  __weakref__
|      list of weak references to the object
|
|  fill_value
|      The fill value.
|
|  ----------------------------------------------------------------------
|  Methods inherited from scipy.interpolate._polyint._Interpolator1D:
|
|  __call__(self, x)
|      Evaluate the interpolant
|
|      Parameters
|      ----------
|      x : array_like
|          Point or points at which to evaluate the interpolant.
```

```
 |
 |      Returns
 |      -------
 |      y : array_like
 |          Interpolated values. Shape is determined by replacing
 |          the interpolation axis in the original array with the shape of `x`.
 |
 |      Notes
 |      -----
 |      Input values `x` must be convertible to `float` values like `int`
 |      or `float`.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from scipy.interpolate._polyint._Interpolator1D:
 |
 |  dtype
```

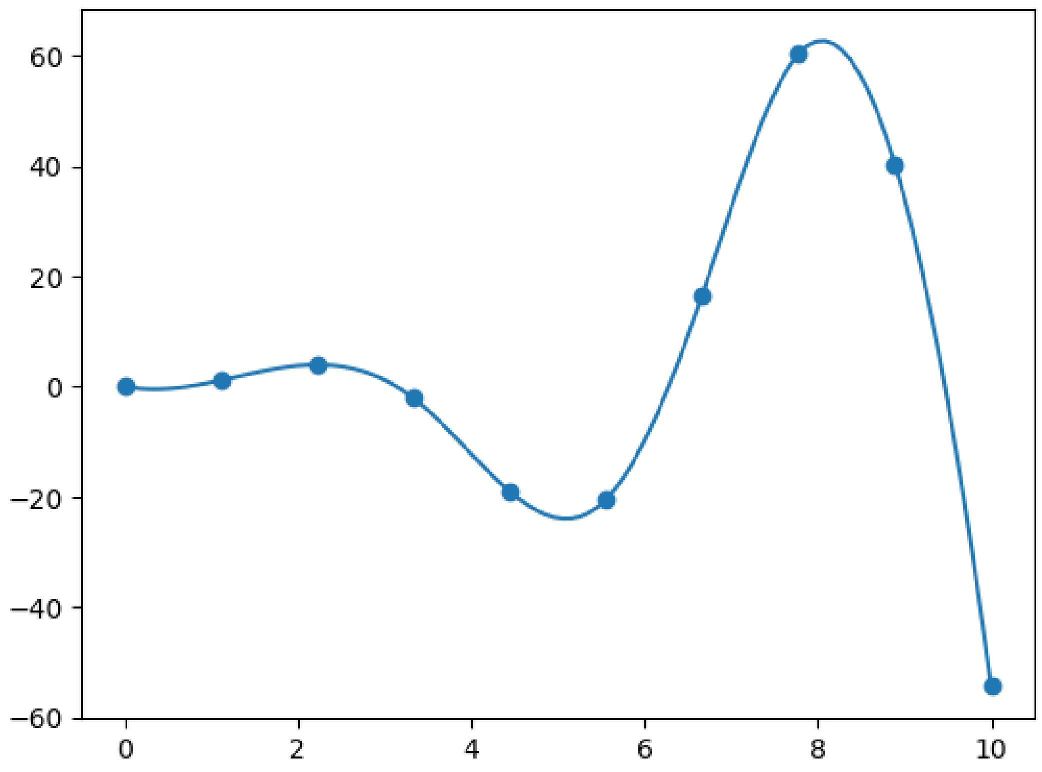
[53]:
```
f = interp1d(x,y,kind='linear')
x_dense = np.linspace(0,10,100)
y_dense = f(x_dense)
plt.plot(x_dense,y_dense)
plt.scatter(x,y)
```

[53]:
```
<matplotlib.collections.PathCollection at 0x13e770b90>
```



```
[54]: f_c = interp1d(x,y,kind='cubic')
      x_dense_c = np.linspace(0,10,100)
      y_dense_c = f_c(x_dense_c)
      plt.plot(x_dense_c,y_dense_c)
      plt.scatter(x,y)
```

```
[54]: <matplotlib.collections.PathCollection at 0x13e7c7250>
```



```
[55]: import pandas as pd
      df = pd.DataFrame(x_dense_c,y_dense_c)
```

```
[56]: df
```

```
[56]:                      0
        0.000000     0.00000
       -0.259188     0.10101
       -0.421656     0.20202
       -0.495659     0.30303
       -0.489453     0.40404
```

```
…               …
-11.704489   9.59596
-21.514938   9.69697
-31.904587   9.79798
-42.868593   9.89899
-54.402111  10.00000

[100 rows x 1 columns]
```

```
[57]: f_c(0.30303)
```

```
[57]: array(-0.49565904)
```

Curve Fitting Finding parameters

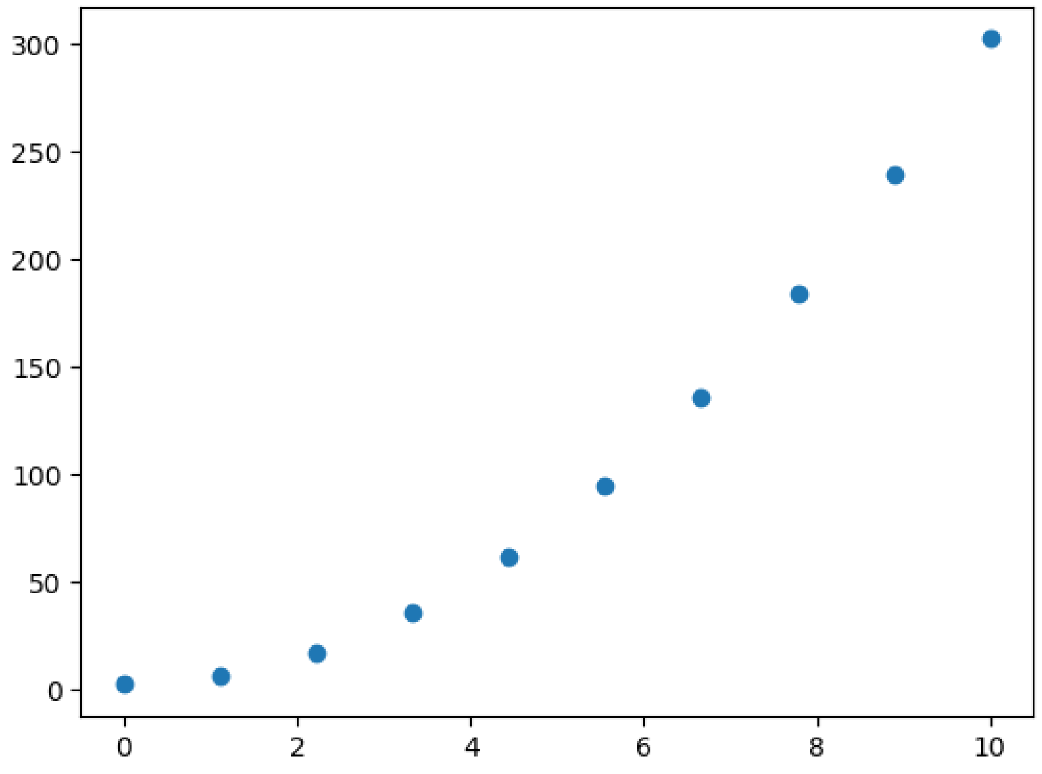
$y = ax^2 + b$

```
[61]: from scipy.optimize import curve_fit
```

```
[58]: x_data = np.linspace(0,10,10)
      y_data = 3*x_data**2+2
```

```
[60]: plt.scatter(x_data,y_data)
```

```
[60]: <matplotlib.collections.PathCollection at 0x14e270550>
```



[62]:
```
f = lambda x,a,b: a*x**2 + b
```

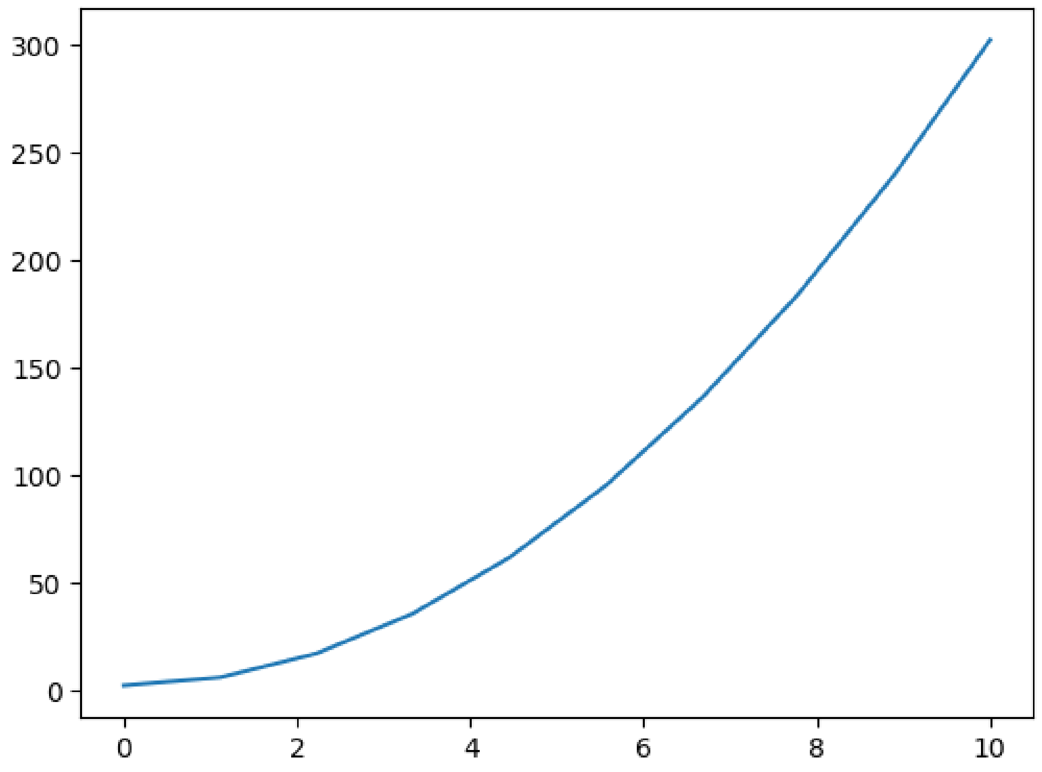
[64]:
```
popt,pcov = curve_fit(f,x_data,y_data,p0=(1,1))
```

[65]:
```
popt
```

[65]:
```
array([3., 2.])
```

[71]:
```
plt.plot(x,3*x**2+2)
```

[71]:
```
[<matplotlib.lines.Line2D at 0x14e3b3610>]
```



[66]:
```
#Noise term
x_data = np.linspace(0,10,10)
y_data_rand = 3*x_data**2+2 + 10* np.random.randn(len(x_data))
```

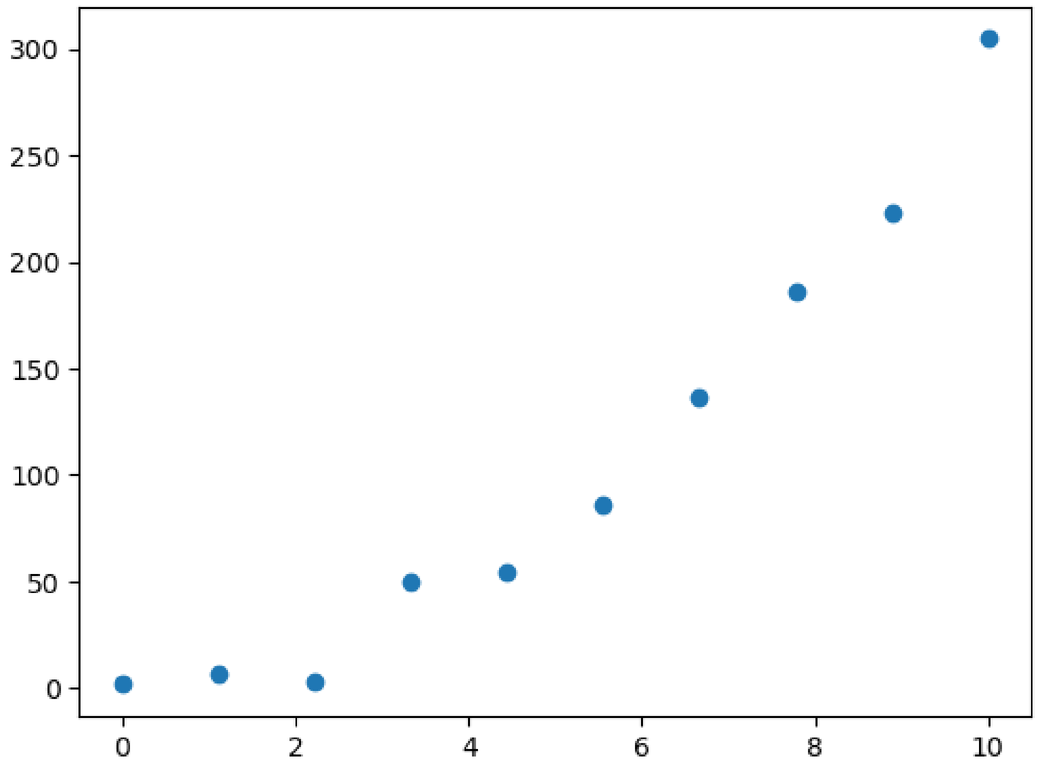
[67]:
```
popt,pcov = curve_fit(f,x_data,y_data_rand,p0=(1,1))
```

[68]:
```
popt
```

```
[68]: array([2.97466041, 0.45399515])
```

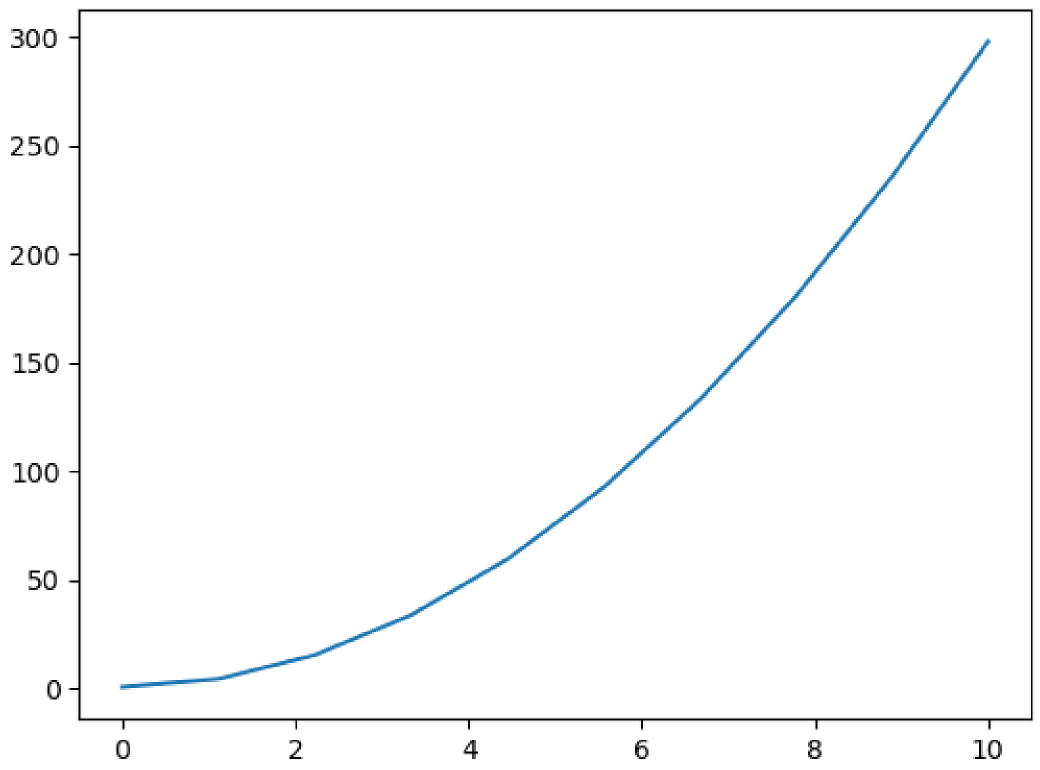
```
[69]: plt.scatter(x_data,y_data_rand)
```

```
[69]: <matplotlib.collections.PathCollection at 0x14e2dead0>
```



```
[70]: plt.plot(x,2.97466041*x**2+0.45399515)
```

```
[70]: [<matplotlib.lines.Line2D at 0x14e361090>]
```



Experimental Data

The equation spring motion is

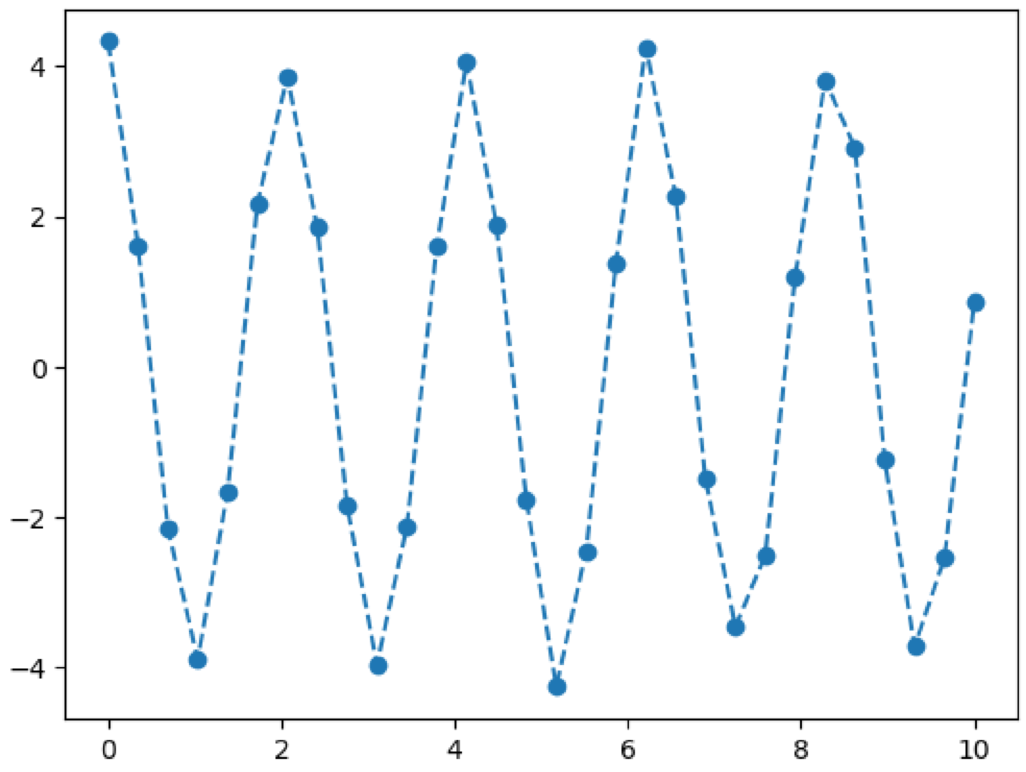
$$y(t) = A\cos(\omega t + \phi).$$

Suppose Want to find the natural frequency of oscillation ω. We have the following experimental data.

[72]:
```
t_data = np.array([ 0.        ,  0.34482759,  0.68965517,  1.03448276,  1.37931034,
        1.72413793,  2.06896552,  2.4137931 ,  2.75862069,  3.10344828,
        3.44827586,  3.79310345,  4.13793103,  4.48275862,  4.82758621,
        5.17241379,  5.51724138,  5.86206897,  6.20689655,  6.55172414,
        6.89655172,  7.24137931,  7.5862069 ,  7.93103448,  8.27586207,
        8.62068966,  8.96551724,  9.31034483,  9.65517241, 10.        ])
y_data = np.array([ 4.3303953 ,  1.61137995, -2.15418696, -3.90137249, -1.
 ↪67259042,
        2.16884383,  3.86635998,  1.85194506, -1.8489224 , -3.96560495,
       -2.13385255,  1.59425817,  4.06145238,  1.89300594, -1.76870297,
       -4.26791226, -2.46874133,  1.37019912,  4.24945607,  2.27038039,
       -1.50299303, -3.46774049, -2.50845488,  1.20022052,  3.81633703,
        2.91511556, -1.24569189, -3.72716214, -2.54549857,  0.87262548])
```

[75]:
```
plt.plot(t_data,y_data,'o--')
```

```
[75]: [<matplotlib.lines.Line2D at 0x14e52c910>]
```



```
[76]: spring_mot = lambda x,A,omega,phi: A*np.cos(omega*x+phi)
```

From theory we know $\omega = 2\pi f$ and $f = \frac{1}{T}$.

So a good starting point would be $A = 4, T = 2$. Hence $\omega = \pi$

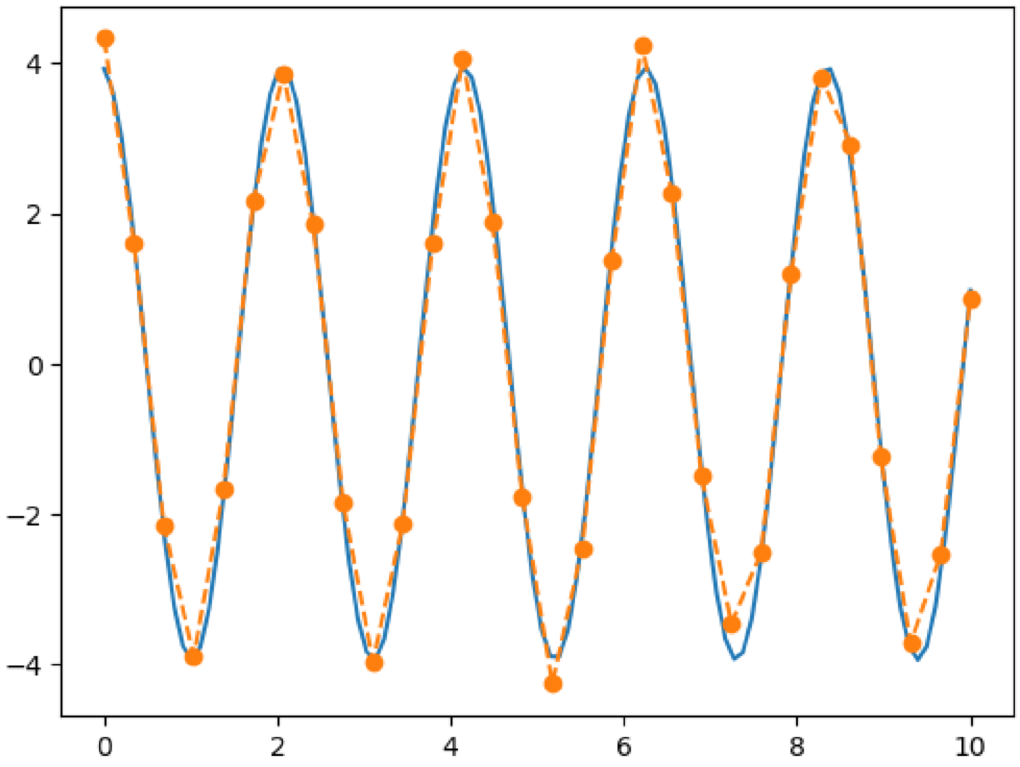
```
[77]: popt,pcov = curve_fit(spring_mot,t_data,y_data,(4,np.pi,0))
```

```
[78]: popt
```

```
[78]: array([3.94836218, 2.99899521, 0.10411349])
```

```
[79]: A,w,phi = popt
      t = np.linspace(0,10,100)
      y = spring_mot(t,A,w,phi)
      plt.plot(t,y)
      plt.plot(t_data,y_data,'o--')
```

```
[79]: [<matplotlib.lines.Line2D at 0x14e58a0d0>]
```



```
[80]: pcov
```

```
[80]: array([[ 2.61882717e-03, -4.94133567e-06,  3.47405339e-05],
             [-4.94133567e-06,  1.85637993e-05, -9.60757788e-05],
             [ 3.47405339e-05, -9.60757788e-05,  6.63424456e-04]])
```

```
[81]: np.sqrt(np.diag(pcov))
```

```
[81]: array([0.05117448, 0.00430857, 0.02575703])
```

Initial guess matters

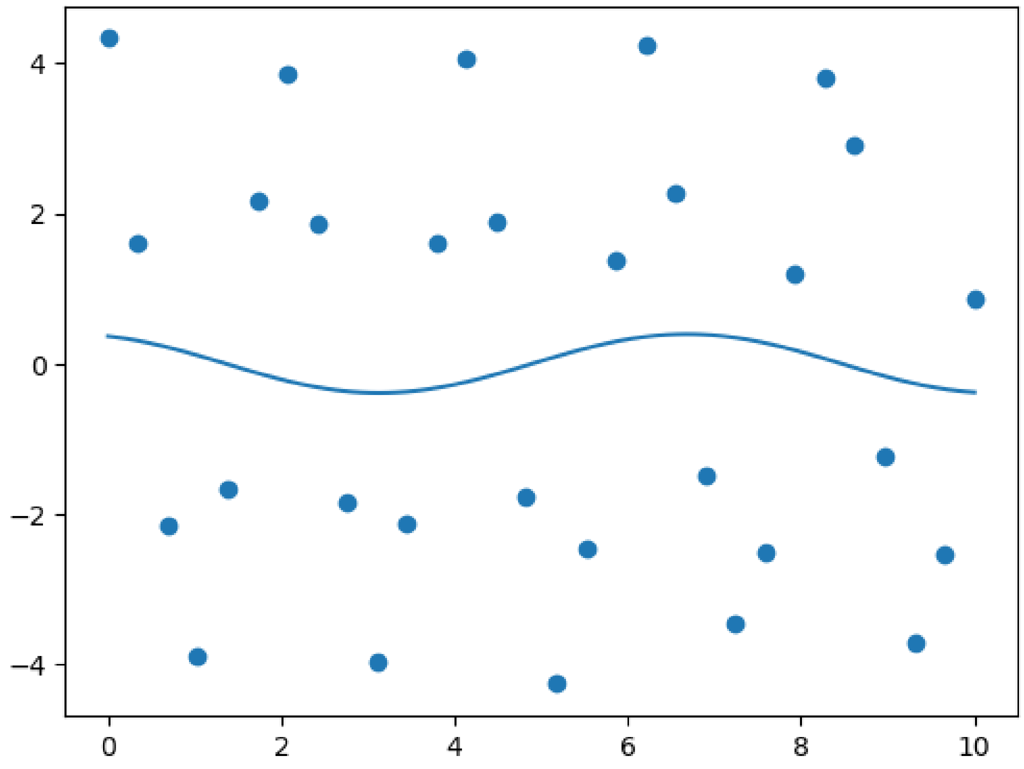
```
[82]: popt_1,pcov_1 = curve_fit(spring_mot,t_data,y_data,p0=(4,1,0))
```

```
[83]: popt_1
```

```
[83]: array([0.39113598, 0.88376295, 0.37821094])
```

```
[84]: A,w,phi = popt_1
      t = np.linspace(0,10,100)
      y = spring_mot(t,A,w,phi)
      plt.plot(t,y)
      plt.scatter(t_data,y_data)
```

```
[84]: <matplotlib.collections.PathCollection at 0x14e5e3890>
```



```
[ ]:
```