# Math review exercises in python

August 19, 2019

## 1 Math review exercises in python

```
In [1]: import numpy as np # import numpy
        import matplotlib.pyplot as plt

        #%matplotlib notebook
        plt.rcParams['figure.figsize'] = [10, 6]
```

## 2 E1

Define variable $x$ with value $x = \frac{}{2}$. Compute the result of $y = x + \sin(x) + \frac{1}{2}x^2$ and store it in a variable $y$

```
In [2]: x = np.pi/2
        x
```

```
Out[2]: 1.5707963267948966
```

```
In [3]: y = x + np.sin(x)+ 1/2*x**2
        y
```

```
Out[3]: 3.8044968769310663
```

## 3 E2

Compute $z = y^3 + 2.5\cos(y)$

```
In [4]: z = y**3 + 2.5*np.cos(y)
        z
```

```
Out[4]: 53.096514643947415
```

## 4 E3

Define functions that compute y and z

```
In [5]: def f1(x):
            y = x + np.sin(x)+ 1/2*x**2
            return y
```

```
In [6]: y = f1(x)
        y
```

```
Out[6]: 3.8044968769310663
```

```
In [7]: def f2(x):
            y =  x**3 + 2.5*np.cos(x)
            return y
```

```
In [8]: z = f2(y)
        z
```

```
Out[8]: 53.096514643947415
```

## 5 E4

Define vector

$$x = \begin{bmatrix} 3 \\ 2 \\ -1.5 \end{bmatrix}$$

```
In [9]: x = np.array([3,2,-1.5])
        x
```

```
Out[9]: array([ 3. ,  2. , -1.5])
```

## 6 E5

Define vector

$$b = \begin{bmatrix} 0 & 1 & -5 \end{bmatrix}$$

```
In [10]: b = np.array([0,1,5])
         b
```

```
Out[10]: array([0, 1, 5])
```

In the previous examples both column and row vector were represented by a 1 dimensional numpy array. This has several advantages but also means that you have to be careful when converting code from MATLAB, where row and column vectors are different.

# 7 E6

```
In [11]: A = np.array([[1,0,0.2],[8,-3,3],[-2,.4,1]])
         B = np.array([[1,2],[4,-1],[2,3]])
         C = np.array([[0,1],[-2,3]])
```

```
In [12]: A
```

```
Out[12]: array([[ 1. ,  0. ,  0.2],
                [ 8. , -3. ,  3. ],
                [-2. ,  0.4,  1. ]])
```

```
In [13]: B
```

```
Out[13]: array([[ 1,  2],
                [ 4, -1],
                [ 2,  3]])
```

```
In [14]: C
```

```
Out[14]: array([[ 0,  1],
                [-2,  3]])
```

Compute

$$x^T x = \begin{bmatrix} 3 & 2 & -1.5 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ -1.5 \end{bmatrix}$$

```
In [15]: x.T.dot(x)
```

```
Out[15]: 15.25
```

or alternatively

```
In [16]: np.dot(x.T,x)
```

```
Out[16]: 15.25
```

Compute \$b x \$

$$bx = \begin{bmatrix} 0 & 1 & -5 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ -1.5 \end{bmatrix}$$

```
In [17]: b.dot(x)
```

```
Out[17]: -5.5
```

Compute $xb$

$$xb = \begin{bmatrix} 3 \\ 2 \\ -1.5 \end{bmatrix} \begin{bmatrix} 0 & 1 & -5 \end{bmatrix}$$

```
In [18]: np.outer(x,b)

Out[18]: array([[ 0. ,  3. , 15. ],
               [ 0. ,  2. , 10. ],
               [-0. , -1.5, -7.5]])
```

It is also possible to differentiate from column and row vectors, in a way similar to MATLAB, by representing vectors as thin 2D numpy arrays. For instance:

```
In [19]: # x = np.array([[3],[2],[-1.5]]) # column vector in the form of 2D array
         # b = np.array([[0,1,5]]) # row vector in the form of 2D array
```

This may seem the easiest way when converting from MATLAB but in the long term it is not convenient. I recommend start thinking in terms of 1D arrays for vectors (independently of being column or row), as a pure data container, which will make the code more efficient.

Compute $AB$

```
In [20]: A.dot(B)

Out[20]: array([[ 1.4,  2.6],
               [ 2. , 28. ],
               [ 1.6, -1.4]])
```

Compute $B^T x$

```
In [21]: B.T.dot(x)

Out[21]: array([ 8. , -0.5])
```

Compute $BC$

```
In [22]: B.dot(C)

Out[22]: array([[-4,  7],
               [ 2,  1],
               [-6, 11]])
```

Compute $ABC$

```
In [23]: A.dot(B).dot(C)

Out[23]: array([[ -5.2,   9.2],
               [-56. ,  86. ],
               [  2.8,  -2.6]])
```

### 7.1 E7

```
In [24]: A = np.array([[1,0,1],[5,-3,2],[-2,4,3]])
         B = np.array([[2,5,1],[-3,1,1],[-1,2,9]])
         x = np.array([[1],[0],[-1]])
```

```
In [25]: A.T.dot(A)
```

```
Out[25]: array([[ 30, -23,   5],
                [-23,  25,   6],
                [  5,   6,  14]])
```

```
In [26]: A.dot(A.T)
```

```
Out[26]: array([[  2,   7,   1],
                [  7,  38, -16],
                [  1, -16,  29]])
```

Note that the result is not the same, which shows that matrix multiplication is not commutative

```
In [27]: A.dot(B)
```

```
Out[27]: array([[  1,   7,  10],
                [ 17,  26,  20],
                [-19,   0,  29]])
```

```
In [28]: B.dot(A)
```

```
Out[28]: array([[ 25, -11,  15],
                [  0,   1,   2],
                [ -9,  30,  30]])
```

```
In [29]: A.dot(x)
```

```
Out[29]: array([[ 0],
                [ 3],
                [-5]])
```

```
In [30]: x.T.dot(A)
```

```
Out[30]: array([[ 3, -4, -2]])
```

```
In [31]: x.T.dot(A).dot(x)
```

```
Out[31]: array([[5]])
```

### 7.2 E8: Random vectors and plots

#### 7.2.1 8a)

Create a random column vector with dimension 100, according to a Normal distribution, with zero mean and standard deviation 1. Plot the result. Experiment with different colors, line styles, and markers.

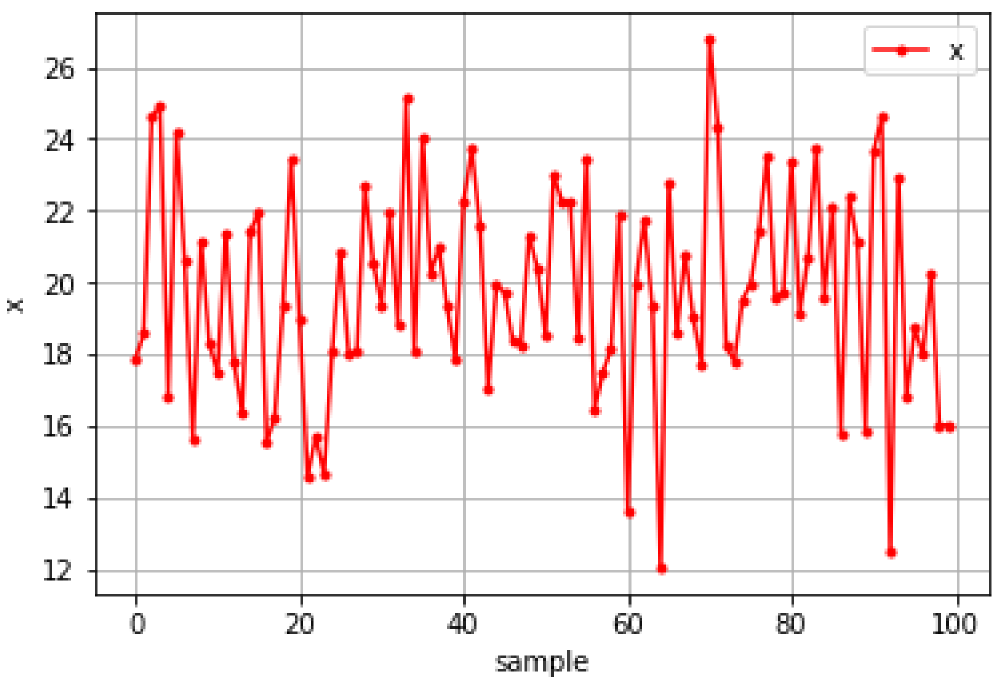
```
In [32]: x = np.random.randn(100);
```

More generally, to generate samples with a standard deviation $\sigma$ and mean $\mu$

```
In [33]: sigma = 3 # standard deviation
         mu = 20 # mean
         N = 100 # number of samples

         x = sigma*np.random.randn(N) + mu;
```

```
In [34]: plt.figure(1)
         plt.plot(x,'.-r')
         plt.grid()
         plt.xlabel('sample')
         plt.ylabel('x')
         plt.legend('x')
```

```
Out[34]: <matplotlib.legend.Legend at 0x1f90bf67a58>
```



To see all available colors and marker types see https://matplotlib.org/2.1.1/api/_as_gen/matplotlib.pyplot.

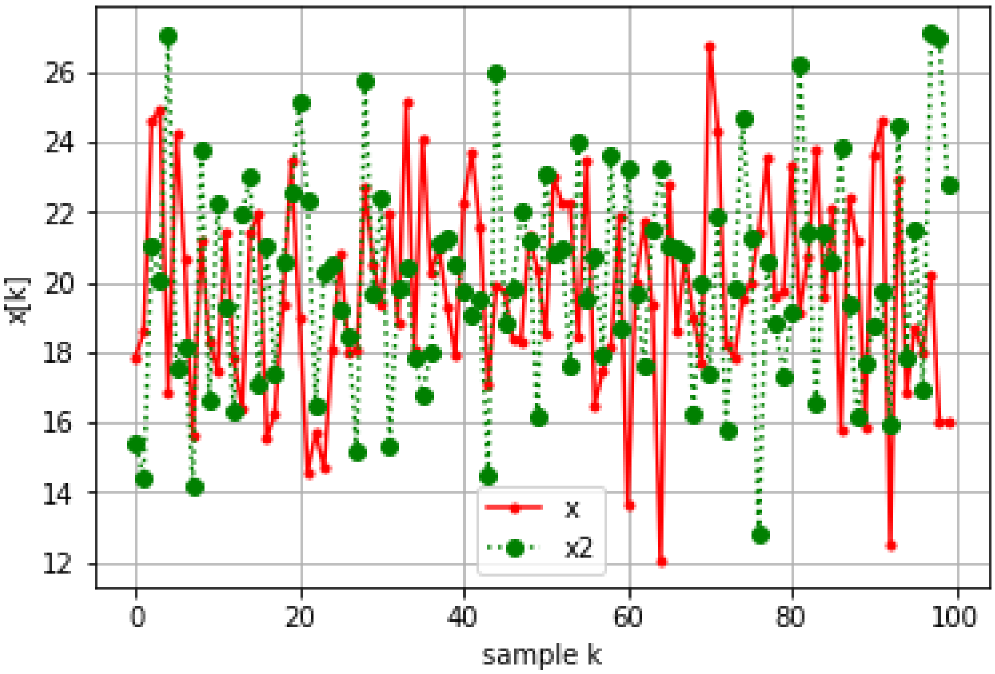
Another way of creating vector x is using a for loop this is less efficient that the previous solution but gives same result

```
In [35]: N = 100 # number of samples
         x2 = np.zeros(N); # initialize, creates a vector of zeros and dimension Nx1
         for i in range(N):
             x2[i] = sigma*np.random.randn() + mu
         # randn when called without arguments returns a scalar random number with normal dist
         x2

Out[35]: array([15.4033803 , 14.37972759, 21.02510938, 20.01300862, 27.02686488,
                17.52472883, 18.16960063, 14.16432513, 23.80549509, 16.63799606,
                22.23418148, 19.30764632, 16.34018454, 21.93161049, 22.97801817,
                17.03432295, 21.02449487, 17.35055419, 20.59886503, 22.52034875,
                25.15614528, 22.35145758, 16.49810378, 20.2558294 , 20.50223905,
                19.20423921, 18.46386548, 15.15488571, 25.73336837, 19.6791264 ,
                22.41156142, 15.3138892 , 19.85182576, 20.39531674, 17.84694411,
                16.80086094, 17.98489519, 21.10034333, 21.29282718, 20.48404581,
                19.70430433, 19.03503198, 19.48786222, 14.46949754, 26.01645174,
                18.81730243, 19.84292908, 22.01413316, 21.21891787, 16.14088876,
                23.11596898, 20.80680217, 20.96133053, 17.58355007, 23.99636351,
                19.48464875, 20.71986725, 17.92142501, 23.61286989, 18.65783938,
                23.26792653, 19.69418032, 17.57690379, 21.50981472, 23.25040532,
                21.05498341, 20.94388891, 20.78667207, 16.2384649 , 19.99720183,
                17.36577609, 21.85401995, 15.81219425, 19.84404194, 24.66206002,
                21.2973309 , 12.81278517, 20.59952049, 18.81711588, 17.32360177,
                19.15168573, 26.21409057, 21.42825637, 16.52139468, 21.41869146,
                20.55712315, 23.84069394, 19.33230765, 16.11986141, 17.69849218,
                18.76662693, 19.74036864, 15.96487488, 24.42612976, 17.82948226,
                21.50593625, 16.92565651, 27.11181796, 26.95459124, 22.7720112 ])
```

### 7.2.2 8b)

Plot both vectors in the same figure and with different colors/markers. Use legend command to label each of the plots.

```
In [36]: plt.figure(2)
         plt.plot(x,'.-r',label='x') # here we pass the label that will be used in the legend
         plt.plot(x2,'o:g',label='x2')
         plt.grid()
         plt.xlabel('sample k')
         plt.ylabel('x[k]')
         plt.legend()

Out[36]: <matplotlib.legend.Legend at 0x1f90c455fd0>
```

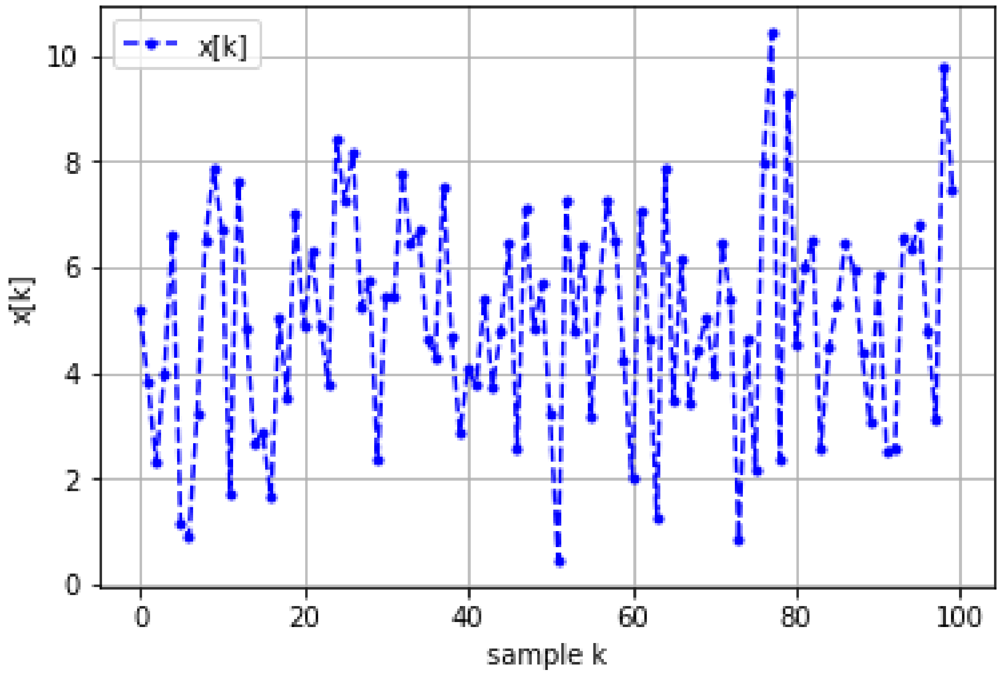



### 7.2.3 c)

Create another random column vector with dimension 100, according to a uniform distribution, mean 5 and standard deviation 2. Plot the result.

In [37]:
```
sigma = 2 # standard deviation
mu = 5 # mean
N = 100 # number of samples

x = sigma*np.random.randn(N,1) + mu;

plt.figure()
plt.plot(x,'.--b',label='x[k]')
plt.grid()
plt.xlabel('sample k')
plt.ylabel('x[k]')
plt.legend()
```

Out[37]:
```
<matplotlib.legend.Legend at 0x1f90c47f278>
```



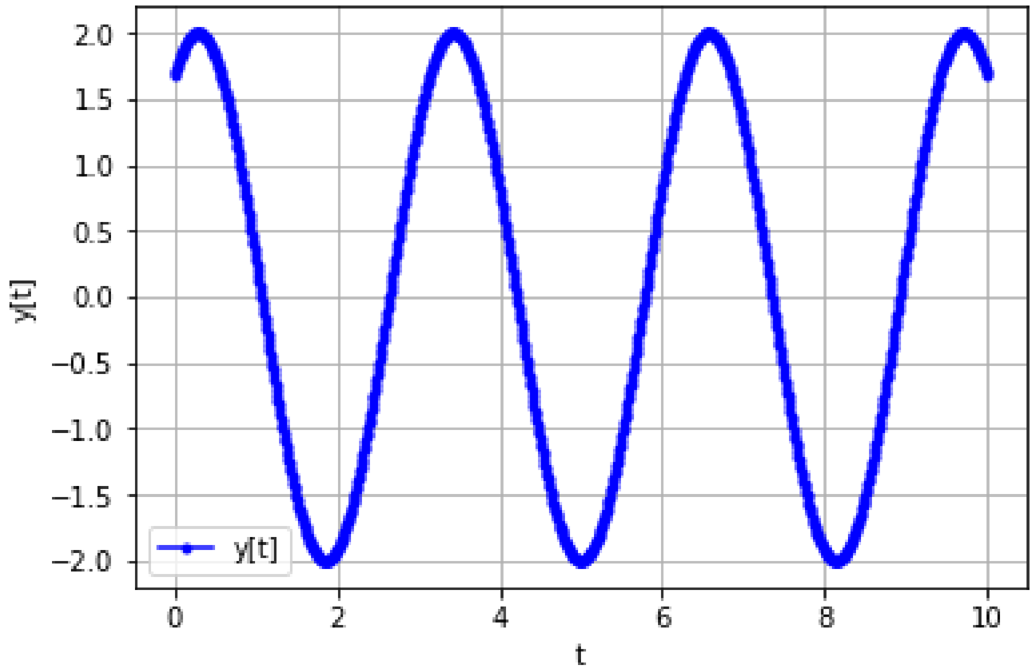

## 8 E9

Plot the signal

$$y = 2sin(t + 1)$$

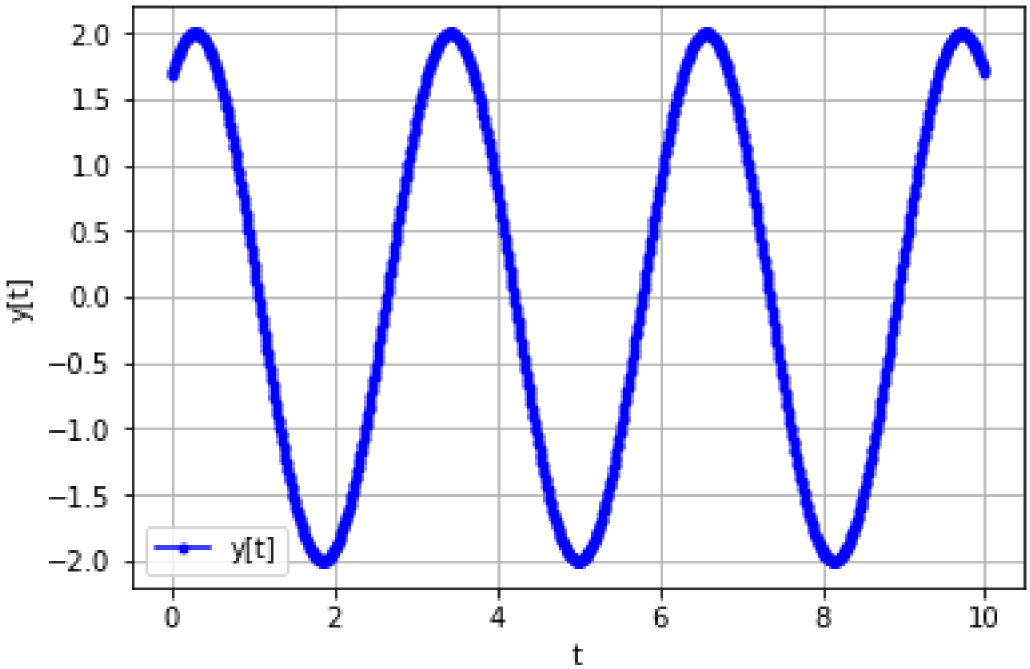
Between time t=0s and t=10s, with =2. Hint: Decide on how many samples you need. Experiment what happens when changing the values of w. One option is to use a for loop to create the signal but it is much simpler and efficient to make an operation on the time vector t.

```
In [38]: N = 1000 # number of samples,
         w = 2
         t = np.linspace(0,10,N)

         y = 2*np.sin(w*t+1)

         plt.figure()
         plt.plot(t,y,'.-b',label='y[t]')
         plt.grid()
         plt.xlabel('t')
         plt.ylabel('y[t]')
         plt.legend()

Out[38]: <matplotlib.legend.Legend at 0x1f90c518668>
```



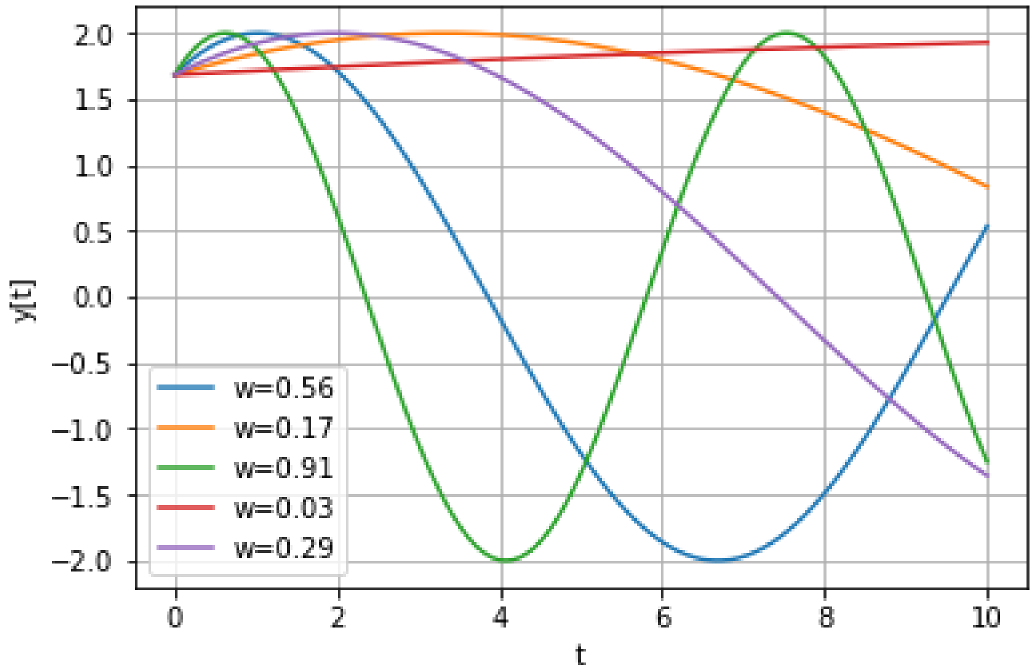

In [39]:
```
# More inefficient solution using a for loop
N = 1000 # number of samples,
w = 2
t0 = 0
tend = 10
h = (tend-t0)/N

t = np.zeros(N) # initialize vectors. This is always a god practice, as in MATLAB
y = np.zeros(N) # initialize

for i in range(N):
    ti = i*h
    yi = 2*np.sin(w*ti+1)
    t[i] = ti # store values
    y[i] = yi

plt.figure()
plt.plot(t,y,'.-b',label='y[t]')
plt.grid()
plt.xlabel('t')
plt.ylabel('y[t]')
plt.legend()
```

Out[39]: <matplotlib.legend.Legend at 0x1f90c596a90>




In [40]:
```
# use foor loop to plot different values of w

N = 1000 # number of samples,
t = np.linspace(0,10,N)

Nplots = 5

plt.figure()
for i in range(Nplots):
    w = np.random.random()
    y = 2*np.sin(w*t+1)
    plt.plot(t,y,'-',label="w={0:4.2}".format(w),)
plt.grid()
plt.xlabel('t')
plt.ylabel('y[t]')
plt.legend()
```

Out[40]:
```
<matplotlib.legend.Legend at 0x1f90c633898>
```



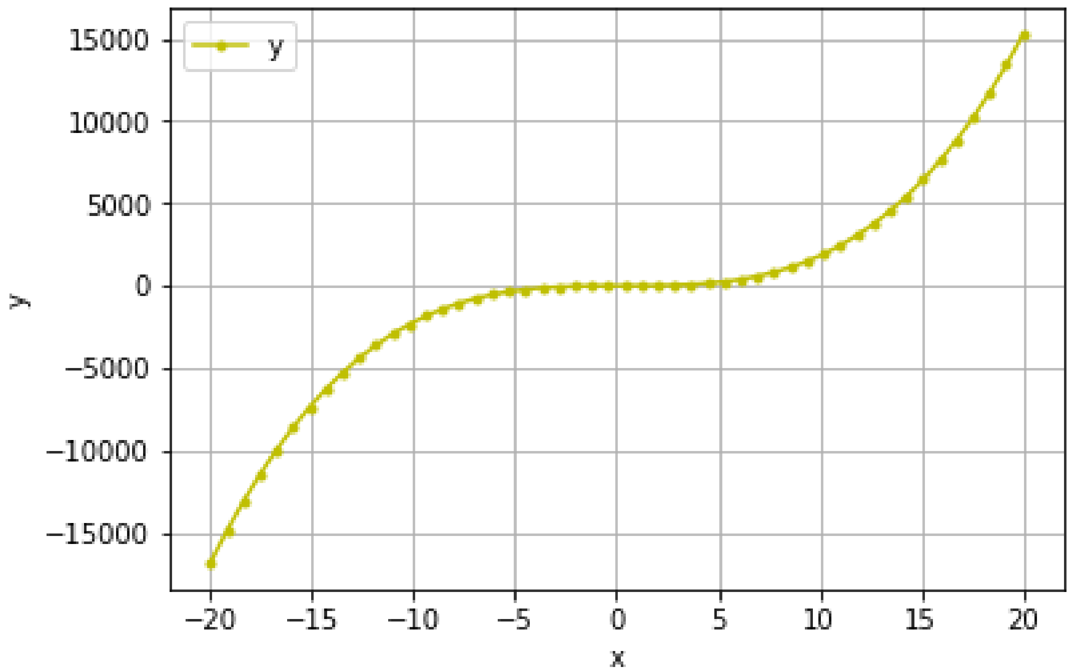

# 9 E10

Plot the signal

$$y = 2x^3 - x^2 + x - 1$$

In the interval x[-20,20].

```
In [41]: N = 50 #
         x = np.linspace(-20,20,N)

         y = 2*x**3 -2*x**2 +x -1

         plt.figure()
         plt.plot(x,y,'.-y',label='y')
         plt.grid()
         plt.xlabel('x')
         plt.ylabel('y')
         plt.legend()

Out[41]: <matplotlib.legend.Legend at 0x1f90c6789e8>
```




## 10 E11

Use MATLAB to solve the following linear system of equations and determine the values of a,b,c:

$$2a + b - 2c = 1a + 4.5b + 5c = 4 - 2a + 3b - 8c = -1$$

Hint: Write the system in matrix form and use matrix inverse function "inv()".
First we write the equations in matrix form $Ax = y$

$$\begin{bmatrix} 2 & 1 & -2 \\ 1 & 4.5 & 5 \\ -2 & 3 & -8 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ -1 \end{bmatrix}$$

which then we can solve using the matrix inverse (if it exists) $x = A^{-1}y$

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 2 & 1 & -2 \\ 1 & 4.5 & 5 \\ -2 & 3 & -8 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 4 \\ -1 \end{bmatrix}$$

```
In [42]: A = np.array([[2, 1, -2],[1, 4.5, 5],[-2, 3, -8]])
         y = np.array([1, 4, -1])
         x = np.linalg.inv(A).dot(y)
         x

Out[42]: array([0.4453125, 0.546875 , 0.21875  ])
```

```
In [43]: A.dot(x)-y # chec result, it should be near zero (except numerical errors)
```

```
Out[43]: array([0., 0., 0.])
```

Note that in general there will be small errors, due to limited precision (double) of representation of numpy arrays.

```
In [44]: A = np.random.randn(4,4)
         np.linalg.inv(A).dot(A)
```

```
Out[44]: array([[ 1.00000000e+00, -5.55111512e-17, -2.22044605e-16,
                  4.44089210e-16],
                [ 5.55111512e-17,  1.00000000e+00,  0.00000000e+00,
                  0.00000000e+00],
                [ 0.00000000e+00,  5.55111512e-17,  1.00000000e+00,
                  0.00000000e+00],
                [ 0.00000000e+00,  2.77555756e-17,  5.55111512e-17,
                  1.00000000e+00]])
```

Note the non zero results of the non diagonal elements is due to numerical precision limitations

## 11 E12

```
Use MATLAB to solve the following linear system of equations and determine the value of x from
```

$$A = \begin{bmatrix} 1 & 2 & 6.5 & -2 \\ 0 & 2.3 & 3 & 0 \\ 3.2 & 0 & 3.5 & 7 \\ -2 & 2 & 1.25 & 9 \end{bmatrix} y = \begin{bmatrix} 0 \\ -2 \\ 1 \\ 4 \end{bmatrix}$$

```
In [45]: A = np.array([[1, 2, 6.5, -2],[0, 2.3, 3, 0],[3.2, 0, 3.5, 7],[-2,2,1.25,9]])
         y = np.array([0,-2, 1, 4])
         x = np.linalg.inv(A).dot(y)
         x
```

```
Out[45]: array([-1.80005649, -2.32840979,  1.11844751,  0.40651636])
```

```
In [46]: A.dot(x)-y # chec result, it should be near zero (except numerical errors)
```

```
Out[46]: array([-8.8817842e-16,  0.0000000e+00, -8.8817842e-16,  8.8817842e-16])
```

## 12 E13 Least Squares polynomial fitting

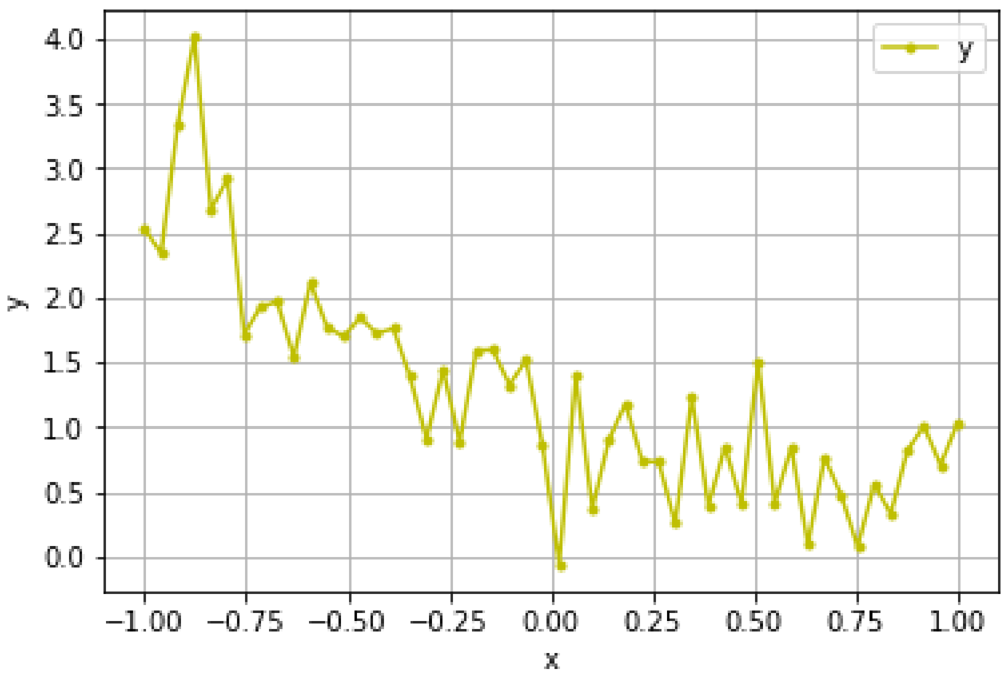
#### 12.0.1 13a)

```
In [47]: N = 50
         sigma = 0.5
         w = sigma*np.random.randn(N)
         x = np.linspace(-1,1,N)
         y = 0.4*x**3 + x**2 -1.5*x +1 + w
         w
```

```
Out[47]: array([-0.56982837, -0.66239949,  0.4256998 ,  1.19983467, -0.03695876,
                 0.2907042 , -0.81437184, -0.50324351, -0.37442199, -0.70732292,
                -0.03669225, -0.29073084, -0.27177771, -0.03996726, -0.0694155 ,
                 0.05207725, -0.23495592, -0.63536182, -0.01837698, -0.50002878,
                 0.28427325,  0.3637441 ,  0.163399  ,  0.4201154 , -0.17716098,
                -1.03531439,  0.48433568, -0.49006728,  0.09257114,  0.42048381,
                 0.02318896,  0.04779775, -0.39170434,  0.61413124, -0.20263228,
                 0.27558176, -0.15281071,  0.95904675, -0.13198375,  0.29984084,
                -0.46127477,  0.19878592, -0.11046371, -0.53622145, -0.0845803 ,
                -0.35151169,  0.09421858,  0.23148765, -0.12855179,  0.12381262])
```

```
In [48]: plt.figure()
         plt.plot(x,y,'.-y',label='y')
         plt.grid()
         plt.xlabel('x')
         plt.ylabel('y')
         plt.legend()
```

```
Out[48]: <matplotlib.legend.Legend at 0x1f90c702048>
```




#### 12.0.2 13b)

```
In [49]: A = np.vstack((x**3, x**2, x, np.ones(N))).T

         # a more ineffìcient way to compute matrix A would be using a for loop
```

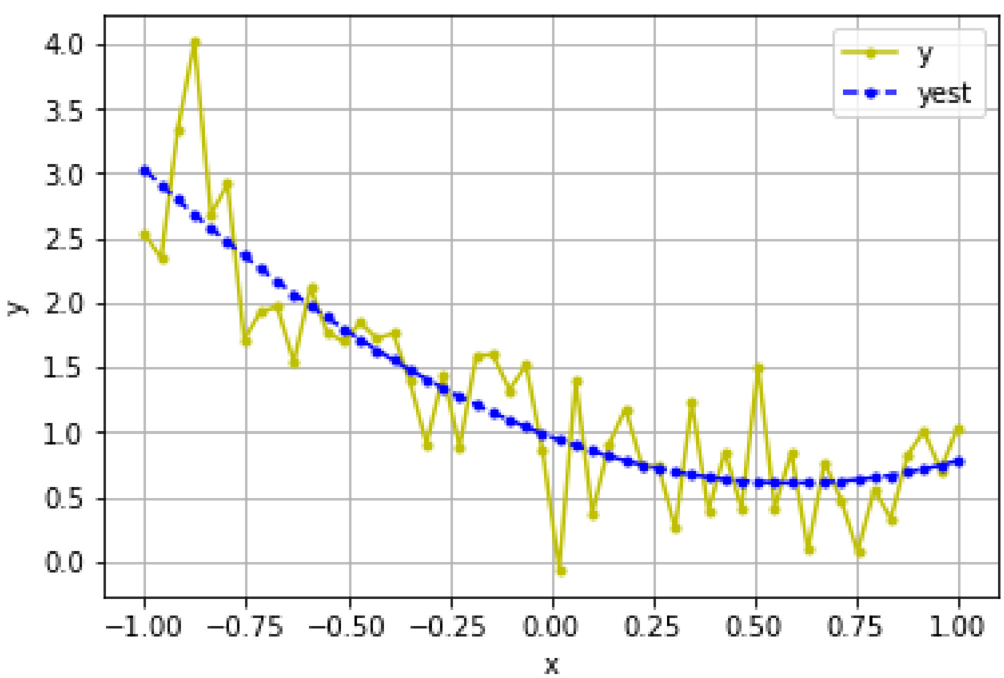
```
A = np.zeros((N,4)) # initialize A to a matrix of zeros
for i in range(N):
    A[i,0] = x[i]**3
    A[i,1] = x[i]**2
    A[i,2] = x[i]
    A[i,3] = 1

theta = np.linalg.pinv(A).dot(y)
```

In [50]:
```
theta
```

Out[50]:
```
array([ 0.04894951,  0.93882233, -1.17433207,  0.96212488])
```

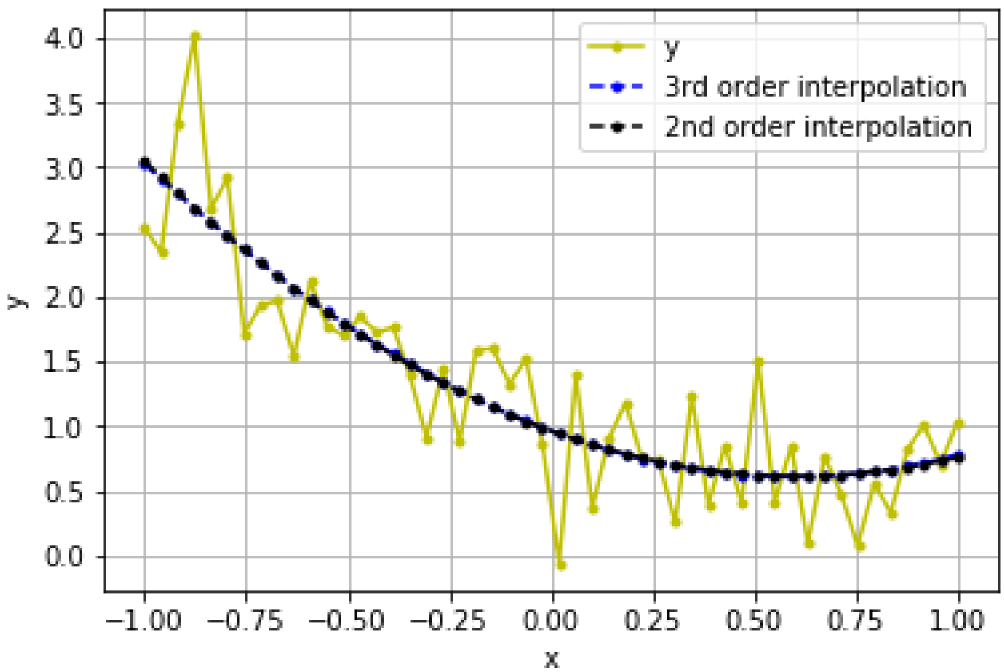
In [51]:
```
yest = A.dot(theta)

plt.figure()
plt.plot(x,y,'.-y',label='y')
plt.plot(x,yest,'.--b',label='yest')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

Out[51]:
```
<matplotlib.legend.Legend at 0x1f90c7c2470>
```



### 12.0.3 13c)

```
In [52]: A2 = np.vstack((x**2, x, np.ones(N))).T
         theta2 = np.linalg.pinv(A2).dot(y)

In [53]: yest2 = A2.dot(theta2)

         plt.figure()
         plt.plot(x,y,'.-y',label='y')
         plt.plot(x,yest,'.--b',label='3rd order interpolation')
         plt.plot(x,yest2,'.--k',label='2nd order interpolation')
         plt.grid()
         plt.xlabel('x')
         plt.ylabel('y')
         plt.legend()

Out[53]: <matplotlib.legend.Legend at 0x1f90c845ac8>
```

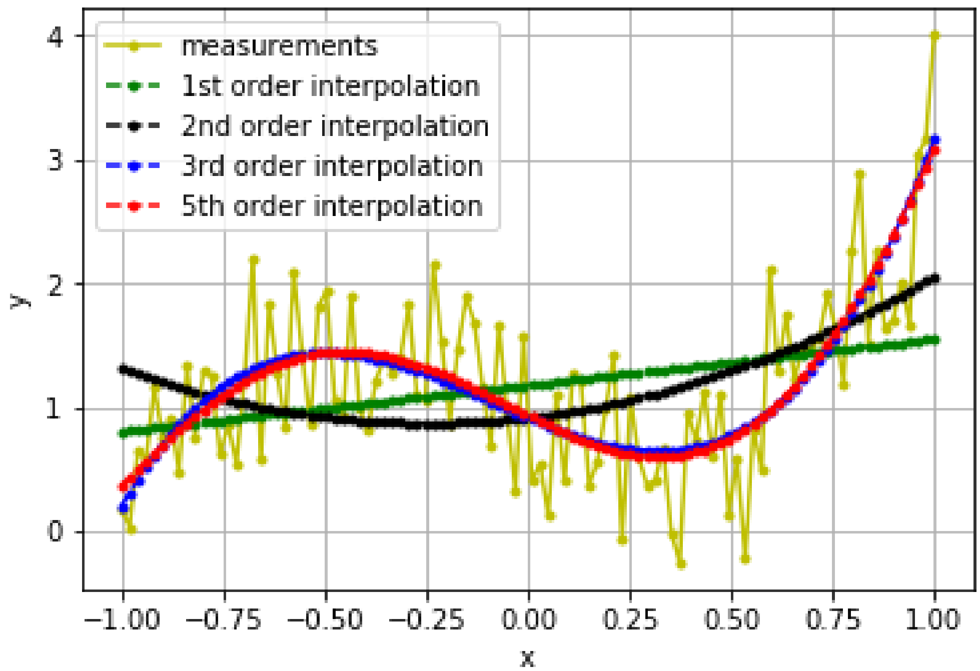



### 12.0.4 13d)

```
In [54]: N = 100
         sigma = 0.5
         w = sigma*np.random.randn(N)
         x = np.linspace(-1,1,N)
         y = 3*x**3 + 0.5*x**2 -1.5*x +1 + w
```

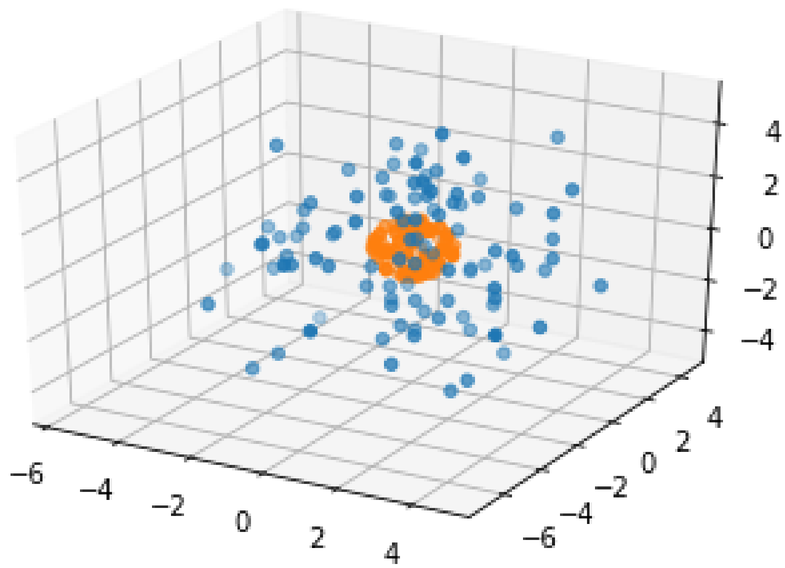
```
# first order interpolation
A1 = np.vstack((x, np.ones(N))).T
theta1 = np.linalg.pinv(A1).dot(y)
# second order interpolation
A2 = np.vstack((x**2, x, np.ones(N))).T
theta2 = np.linalg.pinv(A2).dot(y)
# third order interpolation
A3 = np.vstack((x**3, x**2, x, np.ones(N))).T
theta3 = np.linalg.pinv(A3).dot(y)
# fifth order interpolation
A5 = np.vstack((x**5, x**4, x**3, x**2, x, np.ones(N))).T
theta5 = np.linalg.pinv(A5).dot(y)

yest1 = A1.dot(theta1)
yest2 = A2.dot(theta2)
yest3 = A3.dot(theta3)
yest5 = A5.dot(theta5)

plt.figure()
plt.plot(x,y,'.-y',label='measurements')
plt.plot(x,yest1,'.--g',label='1st order interpolation')
plt.plot(x,yest2,'.--k',label='2nd order interpolation')
plt.plot(x,yest3,'.--b',label='3rd order interpolation')
plt.plot(x,yest5,'.--r',label='5th order interpolation')
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

```
Out[54]: <matplotlib.legend.Legend at 0x1f90c8e4780>
```




## 13 E15 Map into unit sphere

```
In [55]: def f(x):
             y = x/np.sqrt(x.dot(x))
             return y

In [56]: N = 100
         X = 2*np.random.randn(N,3)
         Y = np.zeros((N,3))
         for i in range(N):
          Y[i,:] = f(X[i,:])

In [57]: from mpl_toolkits import mplot3d

In [58]: fig = plt.figure()
         ax = plt.axes(projection='3d')
         ax.scatter3D(X[:,0],X[:,1],X[:,2],'.g')
         ax.scatter3D(Y[:,0],Y[:,1],Y[:,2],'.g')

Out[58]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1f90d981b38>
```




## 14 E17 Derivative

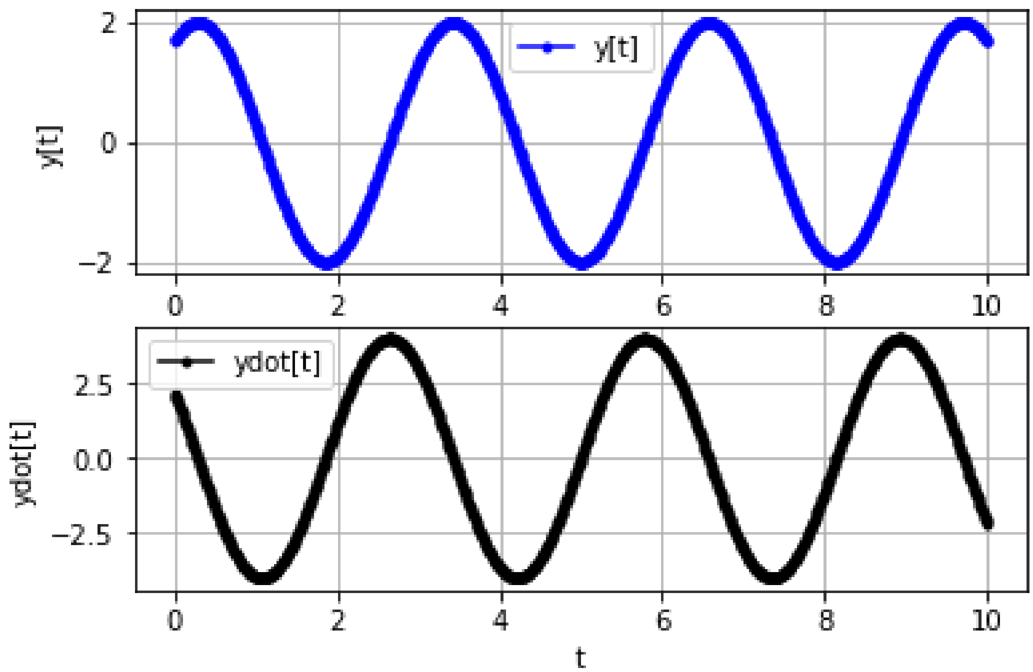
Compute the drivative of the signal

$$y = 2\sin(t+1)$$

$$\frac{dy}{dt} = 2w\cos(t+1)$$

In [59]:
```
N = 1000 # number of samples,
w = 2
t = np.linspace(0,10,N)

y = 2*np.sin(w*t+1)
ydot = 2*w*np.cos(w*t+1)

plt.figure()
plt.subplot(211)
plt.plot(t,y,'.-b',label='y[t]')
plt.grid()
plt.xlabel('t')
plt.ylabel('y[t]')
plt.legend()
plt.subplot(212)
plt.plot(t,ydot,'.-k',label='ydot[t]')
plt.grid()
plt.xlabel('t')
plt.ylabel('ydot[t]')
plt.legend()
```

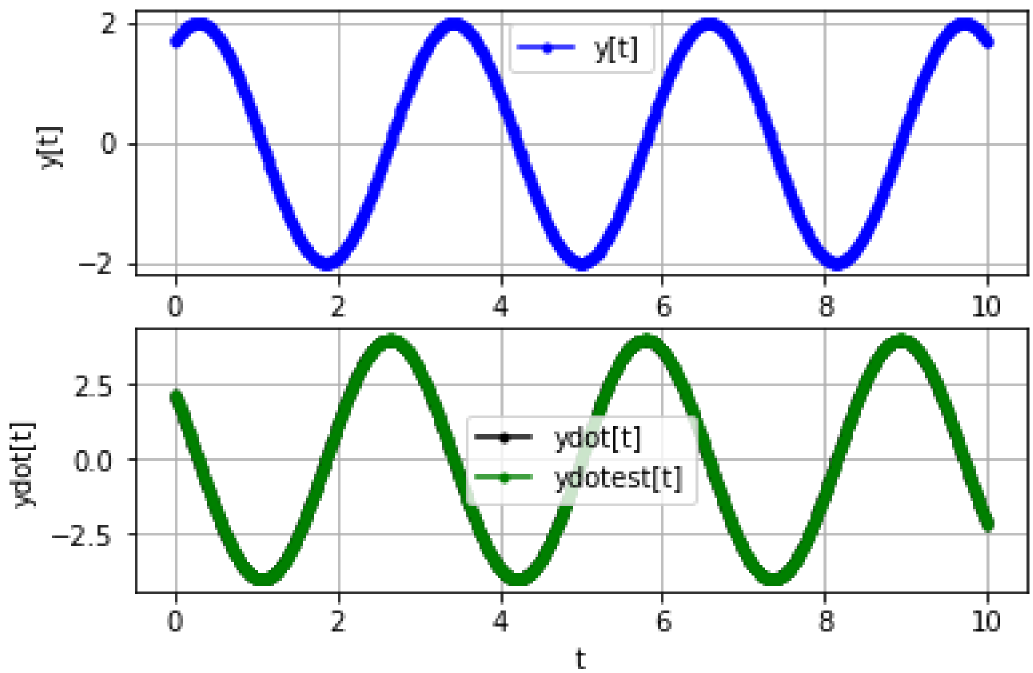
Out[59]: <matplotlib.legend.Legend at 0x1f90da3f4e0>




In [60]:
```
# Compute dirty derivative
h = 10/N # time step

ydotest = np.zeros(N-1) # initialize, it has N-1 size
for i in range(N-1):
    ydotest[i] = (1/h)*(y[i+1]-y[i])

#Instead of using a for loop, a more efficient solution is to usethe function diff (M
ydotest = 1/h*np.diff(y)

plt.figure()
plt.subplot(211)
plt.plot(t,y,'.-b',label='y[t]')
plt.grid()
plt.xlabel('t')
plt.ylabel('y[t]')
plt.legend()
plt.subplot(212)
plt.plot(t,ydot,'.-k',label='ydot[t]')
plt.plot(t[1:],ydotest,'.-g',label='ydotest[t]')
# note that we used t[1:] which so that its size is N-1
# in MATLAB
plt.grid()
```

```
plt.xlabel('t')
plt.ylabel('ydot[t]')
plt.legend()
```

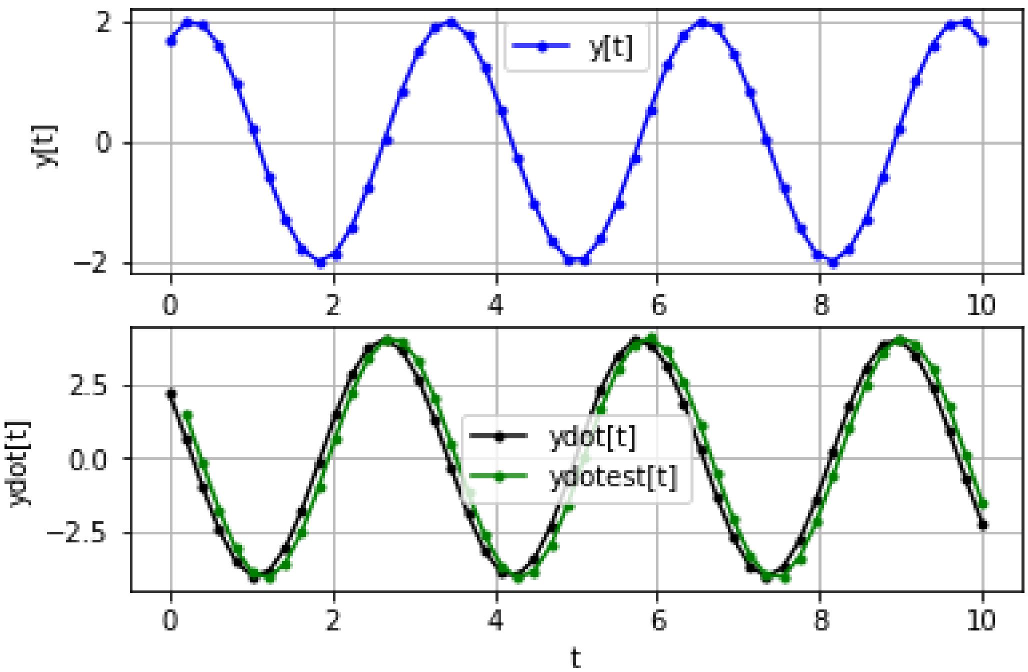
Out[60]: <matplotlib.legend.Legend at 0x1f90dafd7b8>




```
In [61]: N = 50 # number of samples,
         w = 2
         t = np.linspace(0,10,N)
         y = 2*np.sin(w*t+1)
         ydot = 2*w*np.cos(w*t+1)
         # Compute dirty derivative
         h = 10/N # time step

         #Instead of using a for loop, a more efficient solution is to usethe function diff (M
         ydotest = 1/h*np.diff(y)

         plt.figure()
         plt.subplot(211)
         plt.plot(t,y,'.-b',label='y[t]')
         plt.grid()
         plt.xlabel('t')
         plt.ylabel('y[t]')
         plt.legend()
         plt.subplot(212)
```

```
plt.plot(t,ydot,'.-k',label='ydot[t]')
plt.plot(t[1:],ydotest,'.-g',label='ydotest[t]')
# note that we used t[1:] which so that its size is N-1
# in MATLAB
plt.grid()
plt.xlabel('t')
plt.ylabel('ydot[t]')
plt.legend()
```

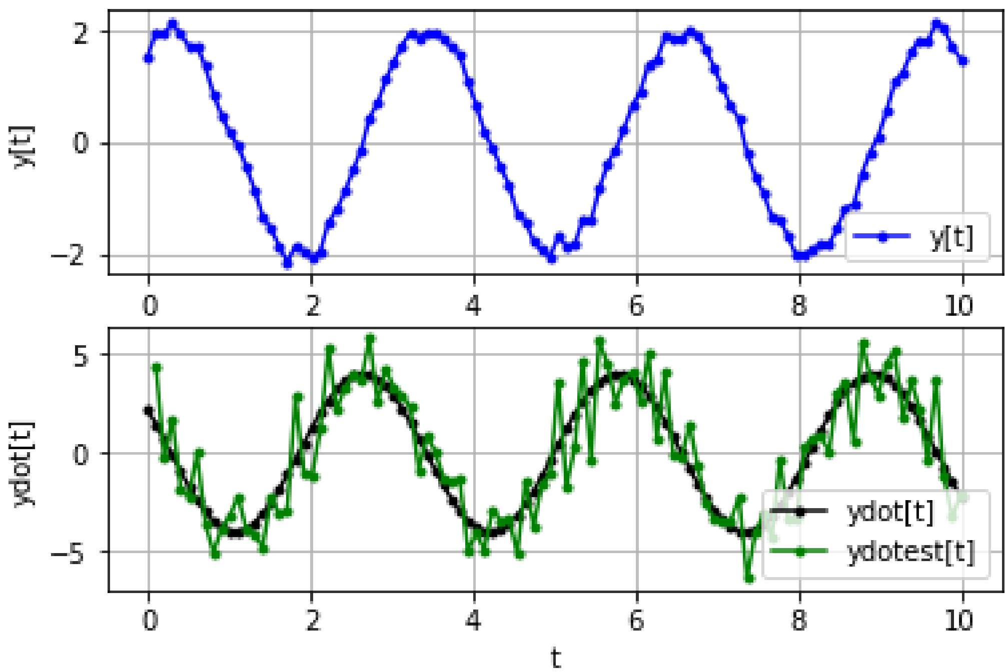
Out[61]: <matplotlib.legend.Legend at 0x1f90dbb6b38>




```
In [62]: N = 100 # number of samples,
w = 2
t = np.linspace(0,10,N)
y = 2*np.sin(w*t+1) + 0.1*np.random.randn(N)
ydot = 2*w*np.cos(w*t+1)
# Compute dirty derivative
h = 10/N # time step

#Instead of using a for loop, a more efficient solution is to usethe function diff (M
ydotest = 1/h*np.diff(y)

plt.figure()
plt.subplot(211)
plt.plot(t,y,'.-b',label='y[t]')
```

```
plt.grid()
plt.xlabel('t')
plt.ylabel('y[t]')
plt.legend()
plt.subplot(212)
plt.plot(t,ydot,'.-k',label='ydot[t]')
plt.plot(t[1:],ydotest,'.-g',label='ydotest[t]')
# note that we used t[1:] which so that its size is N-1
# in MATLAB
plt.grid()
plt.xlabel('t')
plt.ylabel('ydot[t]')
plt.legend()
```

Out[62]: <matplotlib.legend.Legend at 0x1f90dc73fd0>




In [ ]: