Interpolation and reconstruction

RONAK MEHTA - MHTRON001

Tasks

These tasks involve writing code, or modifying existing code, to meet the objectives described.

1. Suppose we have samples $x[n] = x(nT)$ of the signal
$$x(t) = \cos\left(\frac{t-5}{5}\right) - \left(\frac{t-5}{5}\right)^3$$
for $n = 0, \ldots, N-1$, with $N = 10$ and $T = 1.2$. Consider the reconstruction equation
$$x_r(t) = \sum_{n=0}^{N-1} x[n] b_0(t - nT).$$
On the same set of axes plot both $x(t)$ and $x_r(t)$ over the range $t = 0$ to $t = (N-1)T$ for the case of a box filter interpolant $b_0(t) = p_T(t) = p_1(t/T)$, where $p_T(t)$ is the unit pulse of total width T centered on the origin.

2. Generate two new plots with the same specifications as for the previous task, but using these interpolation kernels:

   A. Cubic kernel $b_0(tT) = \begin{cases} (a+2)|t|^3 - (a+3)|t|^2 + 1 & 0 \le |t| < 1 \\ a|t|^3 - 5a|t|^2 + 8a|t| - 4a & 1 \le |t| < 2 \\ 0 & 2 \le |t| \end{cases}$
   with $a = -1/2$.

   B. Truncated or windowed sinc function $b_0(tT) = \frac{\sin(\pi t)}{\pi t} p_{2w}(t)$ for $w = 3$.

3. Use a representation of the form
$$x_r(t) = \sum_{n=0}^{N-1} c_n b_n(t)$$
with a polynomial basis $b_n(t) = t^n$ to approximate the signal $x(t) = e^{(t/10 + cos(t))}$. The coefficients should be calculated using least squares from the samples $x[n] = x(nT)$ for $T = 1$ and $n = 0, \ldots, M-1$. On a single set of axes show the signal $x(t)$, the reconstruction $x_r(t)$, and the sampled points, for the case of $N = 9$ and $M = 11$. The domain of the plot should be from $t = 0$ to $t = 10$.

4. (optional) Repeat the previous task using a cosine basis $b_n(t) = \cos(\omega_0 nt)$ with $\omega_0 = 2\pi/10$ and $N = 6$.

Task 1

In [13]:

```
# 1.A) Set up symbolic signal x
# Create a symbolic variable 't', and specify the given x in terms of t. Use 'lambdify'
 to create a function 'lam_x'
# that takes t as an input and gives x(t) out.

import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
%matplotlib notebook

# Set up symbolic signal to approximate
t = sp.symbols('t');
x = sp.cos((t-5)/5) - ((t-5)/5)**3;
lam_x = sp.lambdify(t, x, modules=['numpy']); #creates lambda function of x that allows
 you to sub t in
```

In [14]:

```
# 1.B) Get the continuous x(t)
# Create an array tv of 2500 time points over the range you're interested in. Get xv, t
he continuous approximation of the signal
# x, by applying the lambda function you just created to each value in tv.

# Dense set of points for plotting "continuous" signals
tv = np.linspace(-2, tmax+2, 2500);
xv = lam_x(tv); # xv is x(t)
```

In [15]:

```
# 1.C) Get the discrete x[n]
# Create the an array 'tnv' that contains the values of nT for n = 0 to n = N. Sample x
 by applying the lam_x function to each
# value in tnv. Call the resulting array xnv.

# Discrete samples of signal over required range
T = 1.2;  tmax = 12;
nm = np.floor(tmax/T); #'floor' rounds down to an integer. This line calculates how man
y periods occur in the required time.
tnv = T*np.arange(0,nm+1); #this is the set of values nT, since 'arange' gives the inte
gers n from 0 up to nm.
xnv = lam_x(tnv); # xnv is x[n]
```

In [16]:

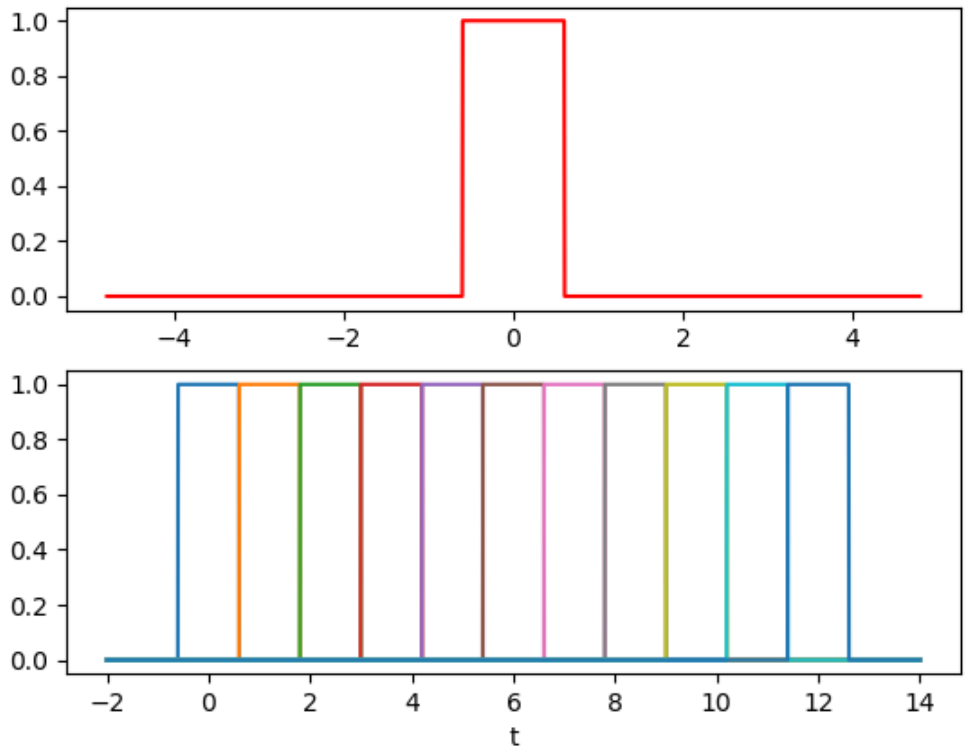
```
# 1.D) Get b0
# Use 'piecewise' to create the rectangular pulse b0. (Hint: the line of code you'll ne
ed should be in the 'Interpolation
# kernel' cell.) Create a lambda function lam_b0 that takes t as the input and returns
 b0(t).

# OPTIONAL: create an array 'b0v' by applying lam_b0 to a time array (tv or some new ar
ray, tbv, over a preferable range.)
# plot b0v to see what each individual basis function looks like. You can also use a fo
r loop to plot all the basis functions
# over the range of interest.

# Interpolation kernel
b0 = sp.Piecewise( (0, t/T<=-1/2), (1, t/T<1/2), (0, True));  # rectangular

lam_b0 = sp.lambdify(t, b0.simplify(), modules=['numpy']); #create lambda function of b
0

# Plot kernel and reconstruction basis
tbv = np.linspace(-4*T, 4*T, 1500); # time array for plotting interpolation function
b0v = lam_b0(tbv);
fh, ax = plt.subplots(2); # plot 1: individual function
ax[0].plot(tbv, b0v, 'r-');
for tn in tnv:
    ax[1].plot(tv, lam_b0(tv-tn)); #plot 2: all the functions in the basis over the ran
ge
ax[1].set_xlabel('t');
```



In [17]:

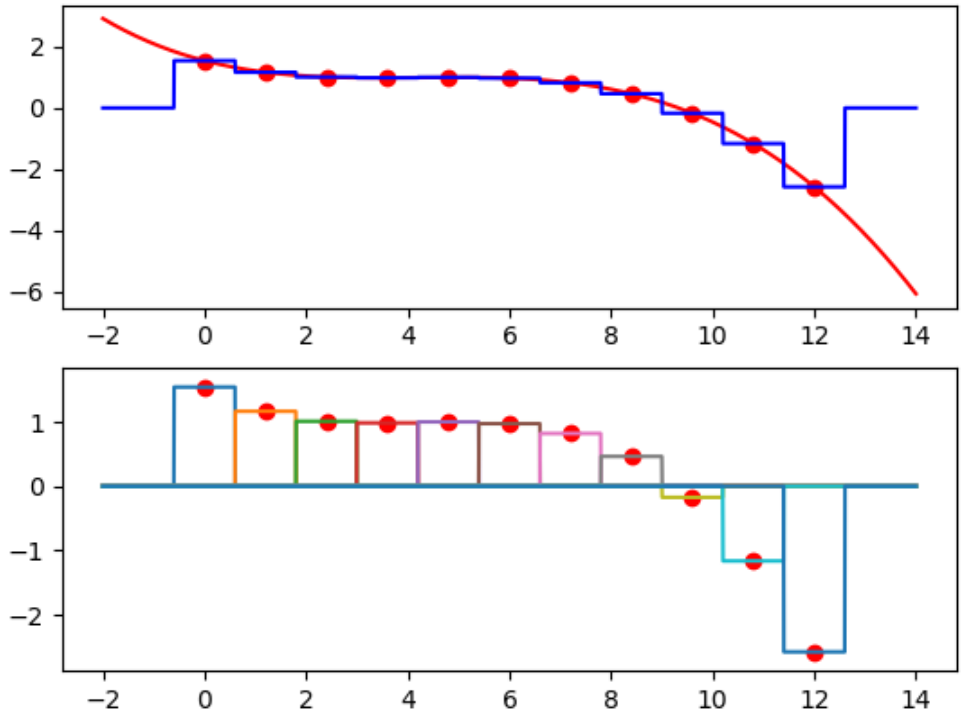
```
# 1.E) Reconstruct
# Create an array 'xav' to hold the reconstructed signal. It should be the same size as
 tv. Use a for loop to add each scaled
# basis function to the reconstruction.

# Numerically evaluate function and approximation
xav = np.zeros(tv.shape);
for i in range(0,len(xnv)):
    xav = xav + xnv[i]*lam_b0(tv - tnv[i]);
```

In [18]:

```
# 1.F) Output
# Plot the original signal xv and the reconstructed signal xav on the same set of axes
 vs tv. If you like, you can use subplot
# to create a second set of axes and plot each basis function on it using a for loop to
 better visualize what's happening.

# Plot
fh, ax = plt.subplots(2);
ax[0].plot(tv,xv,'r-',tv,xav,'b-');
ax[0].scatter(tnv, xnv, c='r');
for i in range(0,len(xnv)):
    ax[1].plot(tv,xnv[i]*lam_b0(tv - tnv[i]));
ax[1].scatter(tnv, xnv, c='r');
```



Task 2.A

In [19]:

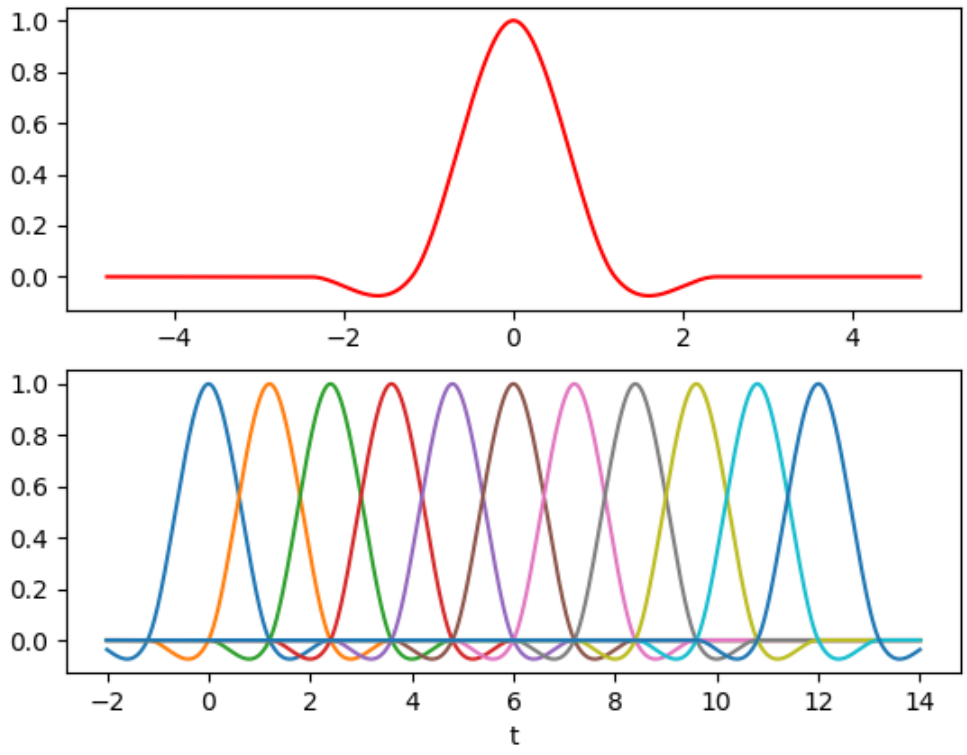
```
# 2.A.A) Get b0
# You've already set up the signal and created continuous and discrete versions of it i
n task 1.
# Use 'piecewise' to create the cubic interpolation function b0 and lambdify it. (Agai
n, Prof's done it for you. You just have to
# uncomment it and change the parameters.)

# OPTIONAL: plot b0 as described in 1.D

at = sp.Abs(t/T);  a = -0.5;  b0 = sp.Piecewise( ((a+2)*at**3 - (a+3)*at**2 + 1 , at<1
), (a*at**3 - 5*a*at**2 + 8*a*at - 4*a, at<2), (0, True));  # cubic

lam_b0 = sp.lambdify(t, b0.simplify(), modules=['numpy']); #create lambda function of b
0

# Plot kernel and reconstruction basis
tbv = np.linspace(-4*T, 4*T, 1500); # time array for plotting interpolation function
b0v = lam_b0(tbv);
fh, ax = plt.subplots(2); # plot 1: individual function
ax[0].plot(tbv, b0v, 'r-');
for tn in tnv:
    ax[1].plot(tv, lam_b0(tv-tn)); #plot 2: all the functions in the basis over the ran
ge
ax[1].set_xlabel('t');
```

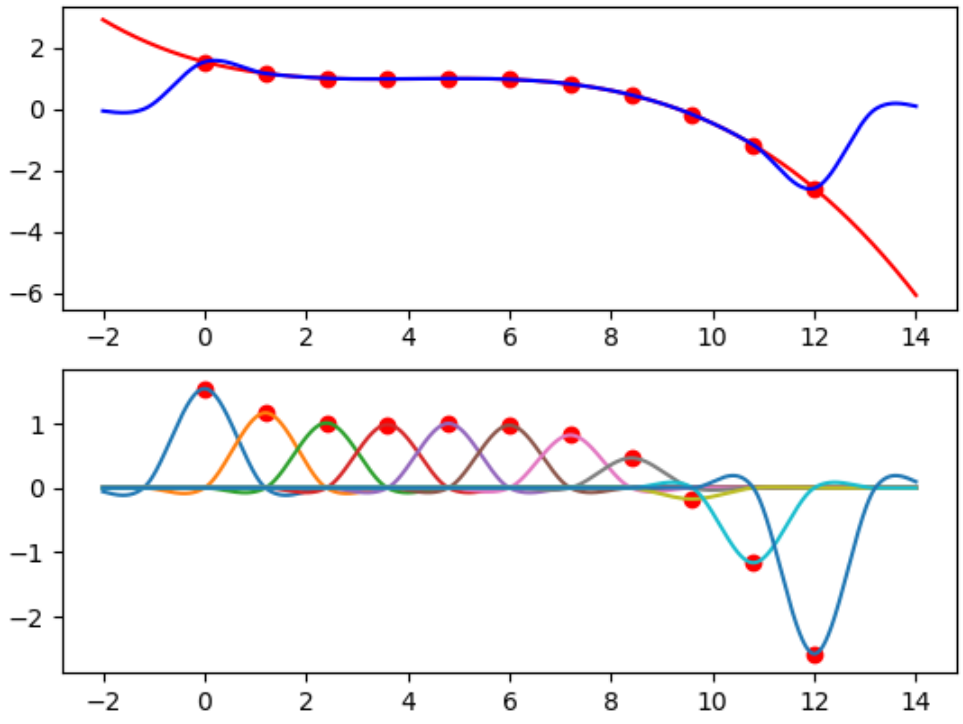


In [20]:

```
# 2.A.B) Reconstruct and output
# Create the reconstructed signal xav and plot it on the same set of axes as the origin
al signal, as described in 1.E and 1.F

# Numerically evaluate function and approximation
xav = np.zeros(tv.shape);
for i in range(0,len(xnv)):
    xav = xav + xnv[i]*lam_b0(tv - tnv[i]);

# Plot
fh, ax = plt.subplots(2);
ax[0].plot(tv,xv,'r-',tv,xav,'b-');
ax[0].scatter(tnv, xnv, c='r');
for i in range(0,len(xnv)):
    ax[1].plot(tv,xnv[i]*lam_b0(tv - tnv[i]));
ax[1].scatter(tnv, xnv, c='r');
```



Task 2.B

In [21]:

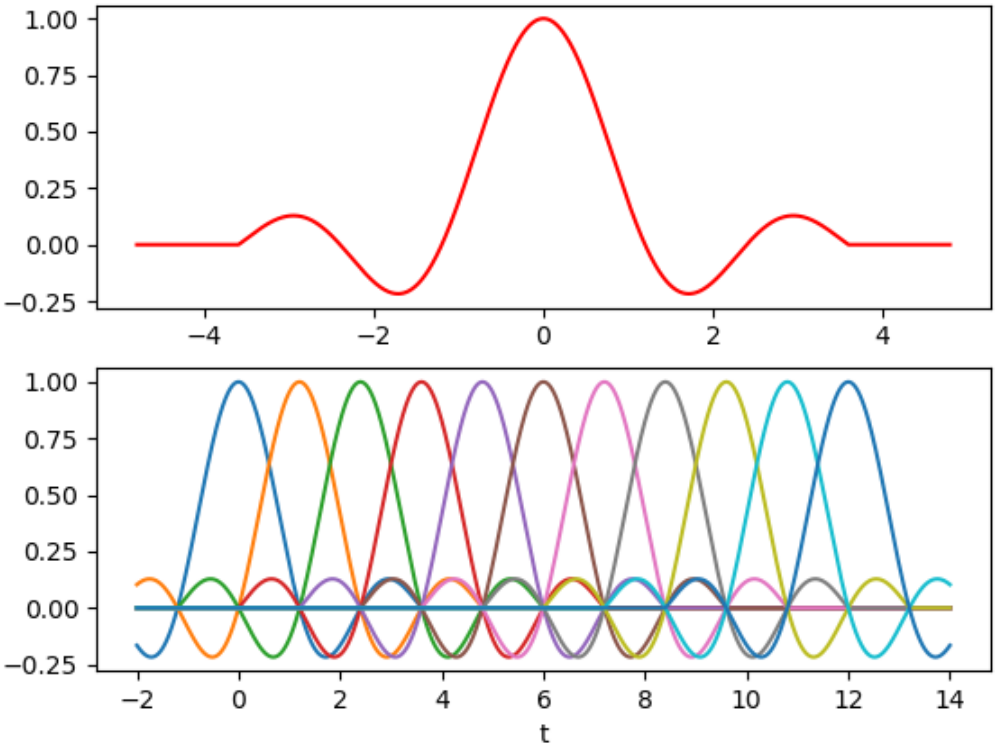
```
# 2.B.A) Get b0
# Use 'piecewise' to create the sinc interpolation function b0 and lambdify it.

# OPTIONAL: plot b0 as described in 1.D

at = sp.Abs(t/T);  w = 3;  b0 = sp.Piecewise( (sp.sin(sp.pi*at)/(sp.pi*at), at<=w), (0,
 True) );  # windowed sinc

lam_b0 = sp.lambdify(t, b0.simplify(), modules=['numpy']); #create lambda function of b
0

# Plot kernel and reconstruction basis
tbv = np.linspace(-4*T, 4*T, 1500); # time array for plotting interpolation function
b0v = lam_b0(tbv);
fh, ax = plt.subplots(2); # plot 1: individual function
ax[0].plot(tbv, b0v, 'r-');
for tn in tnv:
    ax[1].plot(tv, lam_b0(tv-tn)); #plot 2: all the functions in the basis over the ran
ge
ax[1].set_xlabel('t');
```

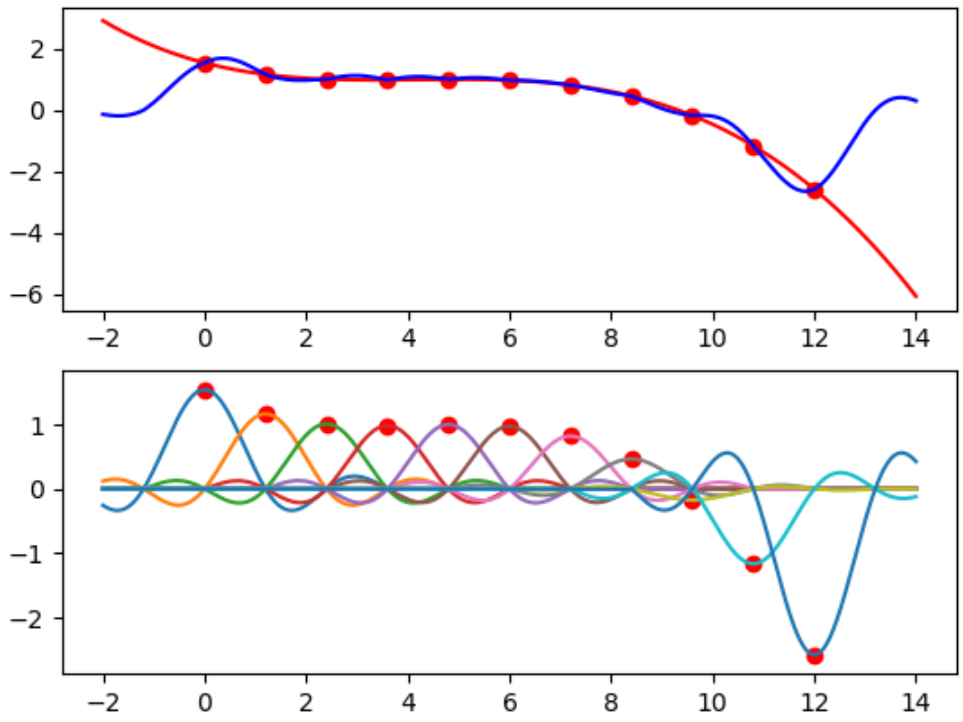


In [22]:

```
# 2.B.B) Reconstruct and output
# Create the reconstructed signal xav and plot it on the same set of axes as the origin
al signal, as described in 1.E and 1.F

# Numerically evaluate function and approximation
xav = np.zeros(tv.shape);
for i in range(0,len(xnv)):
    xav = xav + xnv[i]*lam_b0(tv - tnv[i]);

# Plot
fh, ax = plt.subplots(2);
ax[0].plot(tv,xv,'r-',tv,xav,'b-');
ax[0].scatter(tnv, xnv, c='r');
for i in range(0,len(xnv)):
    ax[1].plot(tv,xnv[i]*lam_b0(tv - tnv[i]));
ax[1].scatter(tnv, xnv, c='r');
```



Task 3

The idea of reconstruction from basis functions and the way this task is formulated probably seem unfamiliar to you, but if you expand out that sum, you get $x(t) = c_0 b_0 + c_1 b_1 + \ldots + c_N b_N$

Substituting in $b_n = t^n$, you get $x(t) = c_0 + c_1 t + c_2 t^2 \ldots + c_N t^N$ so all we're actually trying to do here is come up with an Nth order polynomial we can use to approximate $x(t)$

In [51]:

```
# 3.A) Set up symbolic x
# Specify the signal x in terms of the symbolic variable t and lambdify it to create la
m_x
# Create the continuous approximation x(t) as described in 1.B

t = sp.symbols('t');
x = sp.exp(sp.cos(t)+t/10)
lam_x = sp.lambdify(t, x, modules=['numpy']);

tv = np.linspace(0,10,2500);
xv = lam_x(tv);
```

3.B Finding coefficients with least squares

We're going to use samples of $x(t)$ at M points to evaluate the coefficients.

At each individual sample, $x(kT) = \sum_{n=0}^{N-1} c_n b_n(kT) = c_0 + c_1(kT) + c_2(kT)^2 + \ldots c_N(kT)^N$, which can be written as the matrix expression:

$$x(kT) = \begin{pmatrix} 1 & kT & (kT)^2 & \cdots & (kT)^N \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_N \end{pmatrix}.$$

By repeating this for k values from 0 through M, we get

$$\begin{pmatrix} x(0T) \\ x(1T) \\ \vdots \\ x(MT) \end{pmatrix} = \begin{pmatrix} 1 & 0T & \cdots & (0T)^N \\ 1 & 1T & \cdots & (1T)^N \\ \vdots & & & \\ 1 & MT & \cdots & (MT)^N \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{pmatrix}$$

or $b = Bc$

We can calculate each term in b using the lam_x function. The matrix B is also simple to evaluate. From there, we just have to invert B and multiply it by b to get the coefficient matrix c out.

In [52]:

```
# 3.B)
# Use 'zeros' to create the M x 1 matrix 'b' and the M x N matrix 'B'. Populate these m
atrices with the values described above
# using for loops. Solve for the coefficient matrix cv by taking the dot product of the
 inverse of B and b.
# (Hint: you can get an idea of how to do each of these tasks by looking at Prof's leas
t squares code.)

from numpy.linalg import pinv

b0r = t ;
lam_b0r = sp.lambdify(t, b0r, modules=['numpy']);

# Evaluate basis matrix and value for periodic observations
M = 11;  # M = 2*len(tnv); # M = how many samples of x(t) you're going to use to calcul
ate the coefficients
T = 1;
N = 9;
tmax=10;

nm = np.floor(tmax/T);
tnv = T*np.arange(0,nm+1);

B = np.zeros((M,N));
# B = M x N matrix containing the values of the N basis functions at each of the M samp
led points
# B(k,n) = b_n(kT) = b0(kT-t_n), for k = 0 to N and n = 0 to M

b = np.zeros(M);
# b = 1 x M matrix containing the value of x(t) at each of the M sampled points
# b(k) = x(kT), for k = 0 to M

for k in range(0,M): # populates b and B arrays
    b[k] = lam_x(k*T);
    for n in range(0,N):
        B[k,n] = (lam_b0r(k*T))**n;

# Solve for coefficients
cv = pinv(B).dot(b);
print(cv);
```

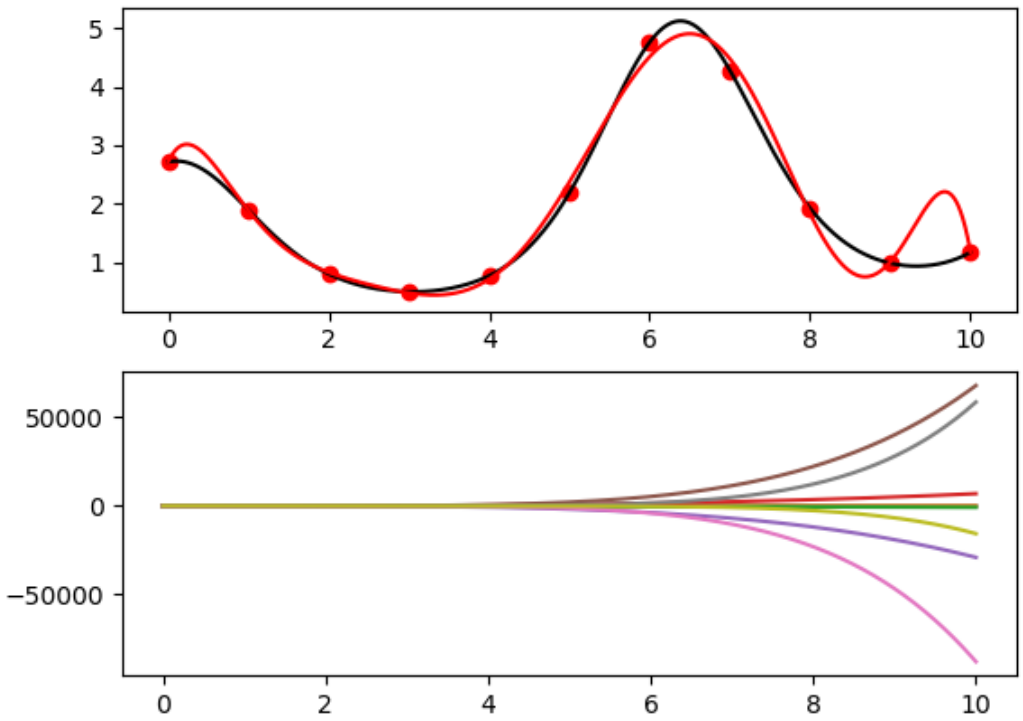
```
[ 2.71980239e+00  2.87171771e+00 -8.30609764e+00  6.91196829e+00
 -2.90882597e+00  6.80109524e-01 -8.80358314e-02  5.86779063e-03
 -1.56763915e-04]
```

In [53]:

```
# 3.C) Reconstruct and output
# Create an array xav, the same size as tv, to use for your polynomial approximation. U
se a for loop to build the polynomial by
# adding each c_n*t**n term from 0 up to N.
# plot the original signal and the polynomial on the same set of axes.

# Numerically evaluate approximation (same as before)
xav = np.zeros(tv.shape);
for n in range(0,N):
    xav = xav + cv[n]*((lam_b0r(tv))**n);

# Plot
fh, ax = plt.subplots(2);
ax[0].plot(tv,lam_x(tv),'k-',tv,xav,'r-');
ax[0].scatter(np.arange(0,M)*T, b, c='r');
for n in range(0,N):
    ax[1].plot(tv,cv[n]*((lam_b0r(tv))**n));
```



<img src = "https://pbs.twimg.com/media/DVMxloCX0AE_aI8.jpg (https://pbs.twimg.com/media/DVMxloCX0AE_aI8.jpg)" width="200"> *"Pumpkin Spice is the smallest but toughest of all my chihuahuas, that's why she's the real champ"* - Hulk Hogan