EE5471: Exploring Orthogonality

Harishankar Ramachandran
EE Dept, IIT Madras

September 5, 2011

I have written the solutions to this lab as part of the LyX file itself. Note that blocks are labelled, for exampe `<Q1a 1b>=`. This means that this piece of code is called `Q1a` and later on you will see a line like `<<Q1a 1b>>` which means that this block of code goes there. You can think of it as an include file mechanism.

- Use the Gram Schmidt orthonormalization procedure to construct an orthogonal set (upto fifth order polynomial) from 1, x, x^2 etc. The inner product is defined to be

  $$< f,g >= \int_0^1 f(x)g(x)dx \tag{1}$$

  I had asked you to use `quad`, but when I started coding I found that the integrals could be obtained directly and hence the code below. `Quad` can also be used.

1a ⟨* 1a⟩≡

```
from scipy import *
from matplotlib.pyplot import *
from scipy.integrate import quad
```

The function below implements the inner product given in Eq. (1). Each polynomial is represented by a vector, whose j^{th} component is the coefficient of x^j. The main line below implements the formula

$$\int_0^1 c_k d_l x^k x^l dx = \frac{c_k d_l}{k+l+1}$$

1b ⟨*Q1a* 1b⟩≡

```
def mydot(c,d):
  """ function that returns <f,g> where they are
  polynomials """
  e=0.0
  for k in range(len(c)):
    for l in range(len(d)):
      e += c[k]*d[l]/(k+l+1.0)
  return e
```

This function does the Gram Schmidt orthonormalization process. It uses the `mydot` function to get the inner product. The method is as follows:

- eigen functions 0 to $k-1$ are the column vectors of matrix `c`.
- Start with $x^k \equiv \delta_{ik}$, ie a column vector of zeros with 1 in the $k+1$ position alone. This is in vector `d`.
- Compute the inner product of `d` and the l^{th} column of matrix `c`.
- Subtract the projection $< d, c_l > c_l$ from d. Note that this assumes that the columns of `c` are orthonormal.
- Iterate till done
- Orthonormalise `d` and copy into the k^{th} column of matrix `c`.

2 $\langle gs\ 2\rangle\equiv$

```
def gs(n):
  """ Gram Schmidt procedure upto x^n """
  c=zeros((n,n));
  c[0,0]=1; # P_0=1
  for k in range(1,n):
    c[k,k]=1;
    d=c[:,k]
    for l in range(k):
      e=mydot(c[:,k],c[:,l])
      d -= e*c[:,l]
    c[:,k]=d/sqrt(mydot(d,d))
  return(c)
```

- Plot the orthogonal functions over $[0,1]$.

  Note that the code above is included as the blocks `Q1a` and `gs`.

  There are some Python pecularities in this code you should note. Python treats column or row vectors as 1 dimensional objects. So when we construct `x=linspace(0,1,N)`, we have a 1-D object. We need a column vector. So we "reshape" it using `x=x.reshape((N,1))`.

  Similarly, *in reverse*, when building up matrix A, we wish to copy over the vector y into the columns of A. But `A[:,k]` expects a 1-D object, so I convert back to a 1-D type via `y.reshape((101,))`. Note that the comma there is just to tell Python that the argument is a list and not an expression.

  The algorithm is simple. We have coefficient vectors c_i in matrix `egn`. We wish to build the function value at values given in x. To do this, we build the matrix

  $$A = \left[1|x|x^2|\ldots|x^{n-1}\right]$$

  where each is a column. Then

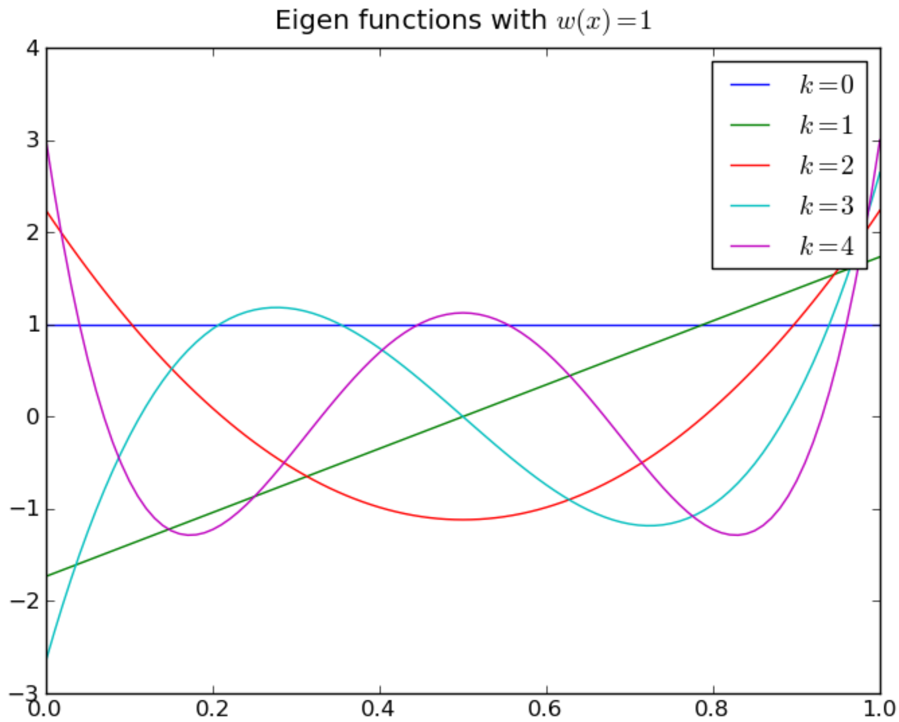
  $$A \cdot c = A_{ij}c_j = \sum_{j=0}^{n-1} c_j x_i^j$$

  This is exactly the desired vector of values. I have written the code so that n and N can be varied. Try it and see how the higher harmonics look.

3 $\langle$*Q2* 3$\rangle\equiv$

```
⟨Q1a 1b⟩
⟨gs 2⟩
n=5;N=101
x=linspace(0,1,N)
x=x.reshape((N,1)) # make column vector
egn=gs(n) # get the first n eigenfunctions
# Build the polynomials from x^0 to x^n-1
A=zeros((N,n))
y=ones((N,1))
for k in range(n):
  A[:,k]=y.reshape((101,))
  y=y*x
figure(0)
s=[]
for k in range(n):
  y=dot(A,egn[:,k]) # compute A.c_k
  plot(x,y)
  s.append("$k=%d$" % k)
title(r"Eigen functions with $w(x)=1$")
legend(s)
```

Here is what I get for these eigenfunctions:



- Repeat the procedure with the inner product

$$< f, g >= \int_0^1 f(x)g(x)xdx$$

Just define a new `mydot` function. Rest is the same.

4 ⟨*Q3a* 4⟩≡

```
def mydot(c,d):
  """ function that returns <f,g> where they are
  polynomials, with w(x)=x """
  e=0.0
  for k in range(len(c)):
    for l in range(len(d)):
      e += c[k]*d[l]/(k+l+2.0)
  return e
```

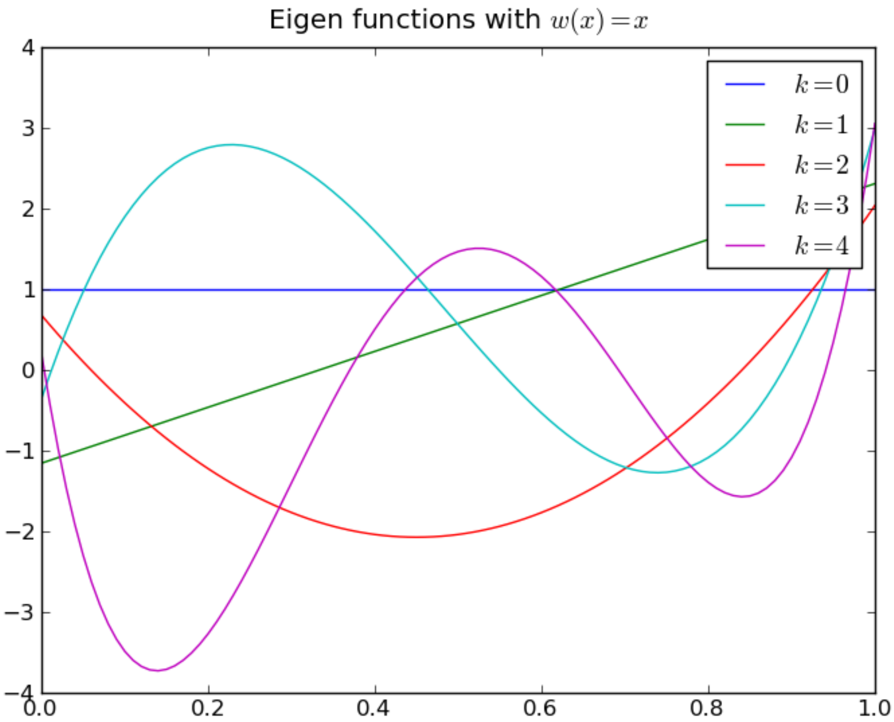
- Plot the new set.

  Code is the same as above, except that I have included <<Q3a>> instead of <<Q1a>>.

5 ⟨*Q4* 5⟩≡

```
⟨Q3a 4⟩
⟨gs 2⟩
n=5;N=101
x=linspace(0,1,N)
x=x.reshape((N,1)) # make column vector
egn=gs(n) # get the first n eigenfunctions
# Build the polynomials from x^0 to x^n-1
A=zeros((N,n))
y=ones((N,1))
for k in range(n):
  A[:,k]=y.reshape((101,))
  y=y*x
figure(1)
s=[]
for k in range(n):
  y=dot(A,egn[:,k])
  plot(x,y)
  s.append("$k=%d$" % k)
title(r"Eigen functions with $w(x)=x$")
legend(s)
```

For this set, here are my eigenfunctions:



Note that when we compare the two sets of eigen functions, the second set is not symmetric. That is because the weight function $w(x)$ is not symmetric in $(0,1)$.

- Consider the function x^6. We wish to approximate it using $1, x, x^2, x^3, x^4$ and x^5. Generate the Taylor series approximation, centred around 0.5

  The Taylor series is given by

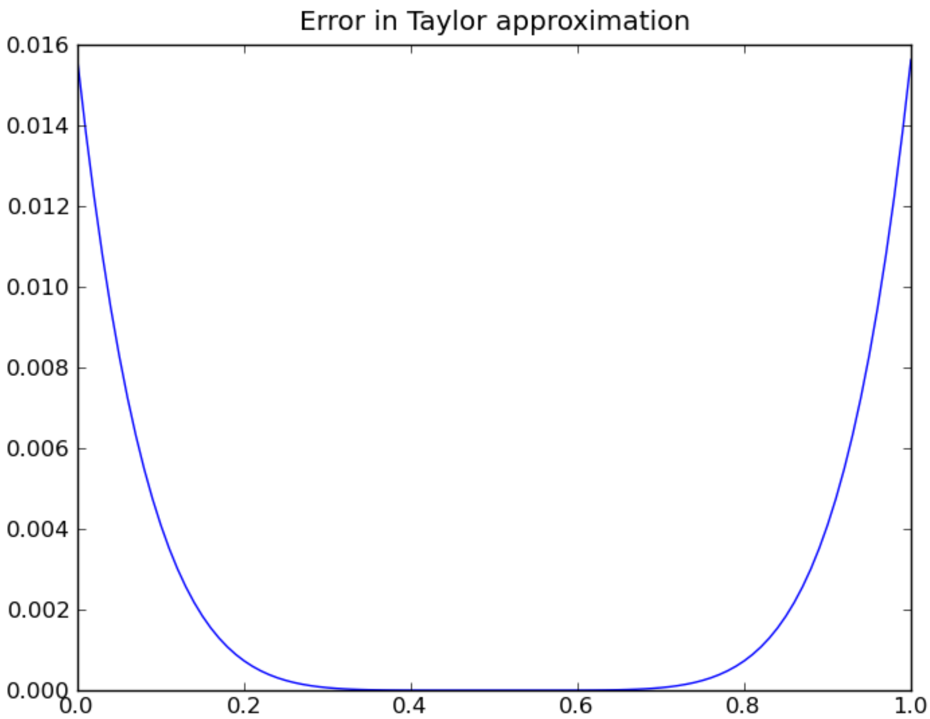
$$\begin{aligned} x^6 &= f^o + (x-0.5)f^1 + \frac{(x-0.5)^2}{2!}f^2 + \dots \\ &= \sum_{k=0}^{6} \frac{6!}{(6-k)!k!} 0.5^{6-k}(x-0.5)^k \end{aligned}$$

6 ⟨Q5 6⟩≡

```
n=5;N=101
x=linspace(0,1,N)
x=x.reshape((N,1))
xx=x-0.5
y=0.5**6 # this f(0.5)
fac=y
t=ones((N,1))
for k in range(1,n+1):
  fac=2.0*fac*(7.0-k)/k
  t=t*xx # (x-0.5)^k
```

```
  y=y+fac*t
figure(4)
plot(x,y)
fexact=x*x*x*x*x*x
plot(x,fexact)
title("Taylor approximation of x^6")
legend(["Taylor approx","Exact"])
figure(5)
plot(x,fexact-y)
title("Error in Taylor approximation")
```

The error plot for Taylor approximation looks as follows:



- Use the orthogonal series to do the same thing.

  The method is simply to compute

$$x^6 \approx \sum_{j=0}^{n-1} < x^6, c_j > c_j$$

The `mydot` function does the work. I am showing here only the first set of eigenfunctions. Again note the line

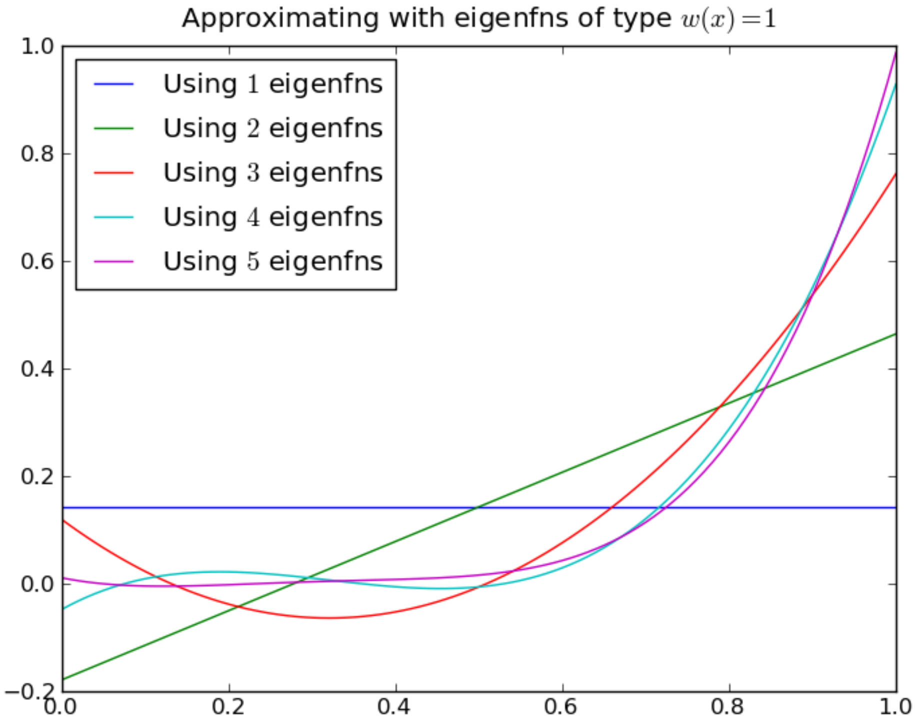
```
d += e*egn[:,k].reshape((n,1))
```

where I convert the 1-D vector to a column vector prior to addition.

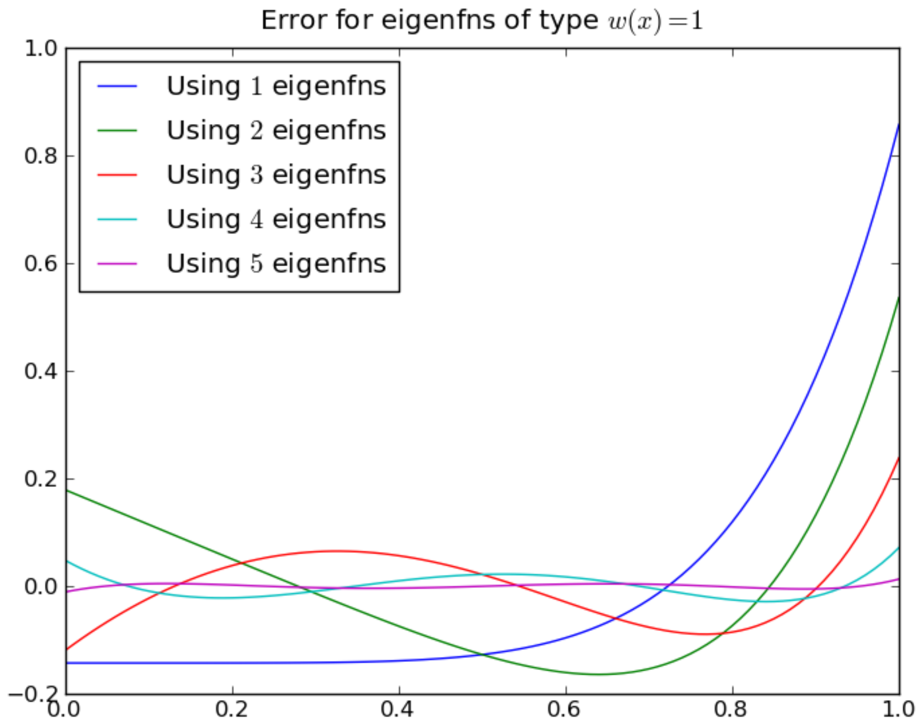
8 ⟨*Q6* 8⟩≡

```
# Eigen functions of Q1
n=5;N=101
x=linspace(0,1,N)
x=x.reshape((N,1)) # make column vector
A=zeros((N,n))
y=ones((N,1))
for k in range(n):
  A[:,k]=y.reshape((101,))
  y=y*x
```

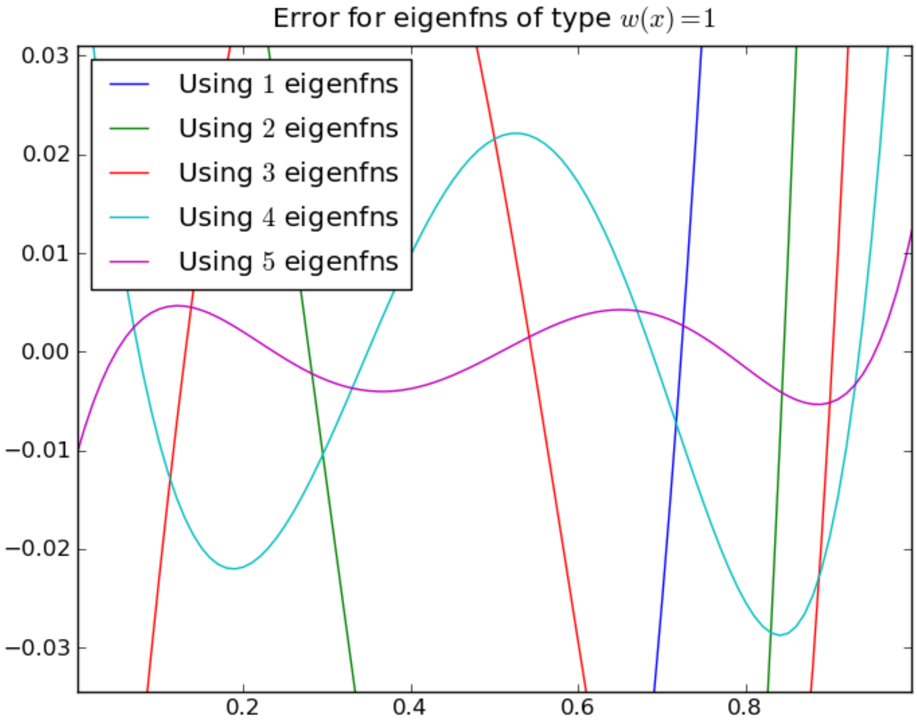
```
⟨Q1a 1b⟩ # here is where I choose the 1st set
⟨gs 2⟩
egn=gs(n) # get the first n eigenfunctions
# given function is x^6
ff=zeros((7,1));ff[6]=1
y=x*x*x*x*x*x # function to be fitted
# build the approximation vector in d
d=zeros((n,1))
s=[]
for k in range(n):
  e=mydot(ff,egn[:,k])
  d += e*egn[:,k].reshape((n,1))
  z=dot(A,d)
  figure(2)
  plot(x,z)
  figure(3)
  plot(x,y-z)
  s.append("Using $%d$ eigenfns" % (k+1))
figure(2)
title(r"Approximating with eigenfns of type $w(x)=1$")
legend(s,loc="upper left")
figure(3)
title(r"Error for eigenfns of type $w(x)=1$")
legend(s,loc="upper left")
```



Approximating with eigenfns of type $w(x) = 1$
Using 1 eigenfns
Using 2 eigenfns
Using 3 eigenfns
Using 4 eigenfns
Using 5 eigenfns
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0



The error plot is very instructive. I have not done the Taylor approximation in the solutions, but you should. COmpare the error from that and the error in this case. This is the main difference between an eigen approximation and a Taylor approximation. The Taylor approximation converges only near the point $x = 0.5$ while the orthogonal approximation converges uniformly. Here is a zoomed plot of the error:



It is clear that the error is distributed over the range.

I thing some of you got a different plot here. You had a pile up of error near $x = 1$ where x^6 increased too fast. That kind of error should not happen with an eigen fit, except in some bizarre cases.

12 $\langle$* 1a$\rangle$+≡
  $\langle$*Q2* 3$\rangle$
  $\langle$*Q4* 5$\rangle$
  $\langle$*Q5* 6$\rangle$
  $\langle$*Q6* 8$\rangle$

- Compare the two approximations with the real function.