EE5471: Interpolation and Integration

Harishankar Ramachandran

August 10, 2011

1 Introduction

In this lab we will try out techniques of interpolation and integration. The codes are in C but we will call them from Python and graph the results.

2 Polynomial Interpolation

Below is the C code in Numerical Recipes for interpolation. It has been linked to Python for your convenience. This python script has been uploaded in the Moodle site as `polint.py`.

- Note that the numerical recipes code assumes that the table is exactly the size required by the interpolation polynomial, i.e.., it has $n+1$ entries. I have converted the code so that it locates the best position in the table for a given x_i value (`n` points centred about x_i). Understand the changes made to the code to achieve this.
- I have eliminated the malloc and free by passing the vectors `yy`, `dyy`, `c` and `d` from Python. This greatly simplifies the C code, but possibly some efficiency can be gained by allocating in C.

This polint implementation takes a vectors `xarr` and `yarr` which contain a table of data. The length of these vectors is `M` elements. The interpolation is to be done over `n` elements. The locations at which the interpolation is to be done is sent in vector `xx` of length `N`. The resulting interpolated value is returned in vector `yy` of length `N` and the error estimate is returned in `dyy` also of length `N`. Vectors `c` and `d` of length `n` are work vectors used by `polint`.

1 ⟨*code* 1⟩≡

```
from scipy import *
from matplotlib.pyplot import *
import scipy.weave as weave
def polint(xarr,yarr,xx,n):
  M=len(xarr);N=len(xx)
  c=zeros((n+1,1));d=zeros((n+1,1))
  yy=zeros(xx.shape);dyy=zeros(xx.shape)
  code="""
  #include<math.h>
  int i,j,n1,m,ns,ns0;
  double den,dift,ho,hp,w;
  double *xa,*ya,*x,*y,*dy; // window to use by polint
  c[0]=d[0]=0.0;
  xa=xarr;ya=yarr; // initialize pointers
  for(j=0,n1=0;j<N;j++){ // loop over xarr points
    x=&xx[j];y=&yy[j];dy=&dyy[j];
```

We take advantage here of the fact that the the points in array `xo` are increasing and simplify the algorithm based on that assumption.

2a ⟨*code* 1⟩+≡

```
for(i=n1,ns=M-1;i<M;i++){ // loop over x points
  if(*x<=xarr[i]){ // found crossover
    ns=i;
    break;
  }
} // i loop
```

If we exited above loop at the very first element it means that the crossover point was to the left.

2b ⟨*code* 1⟩+≡

```
if( i==n1 ){
  for(ns=n/2;i>=0;i-){ // loop over x points
    if(*x>xarr[i]){ // found crossover
      ns=i;
      break;
    }
  } // i loop
}
```

Now we need to pin down the beginning of the window of size `n+1`.

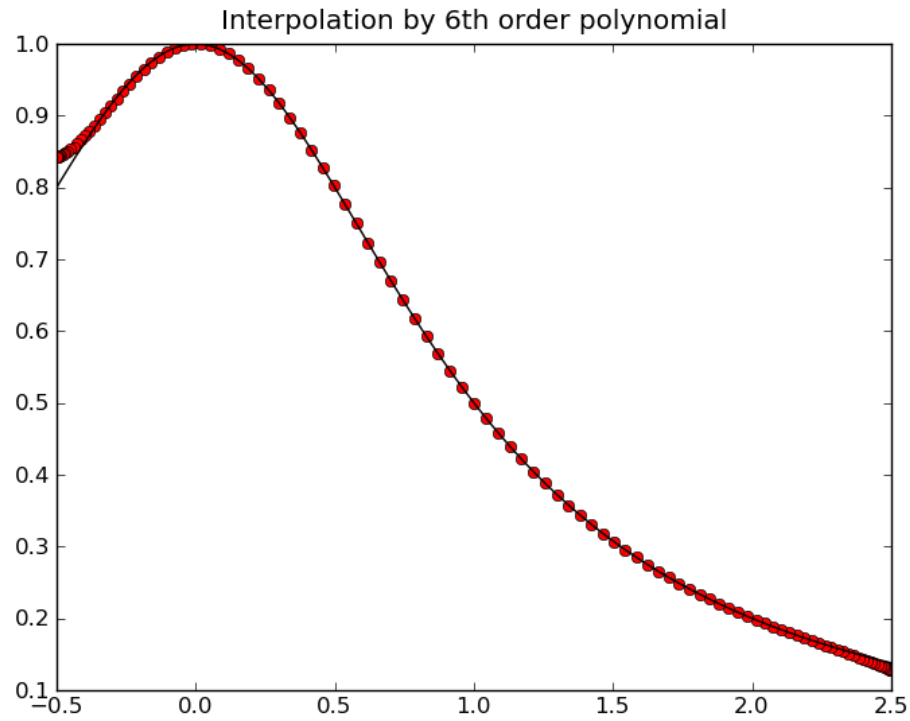
2c ⟨*code* 1⟩+≡

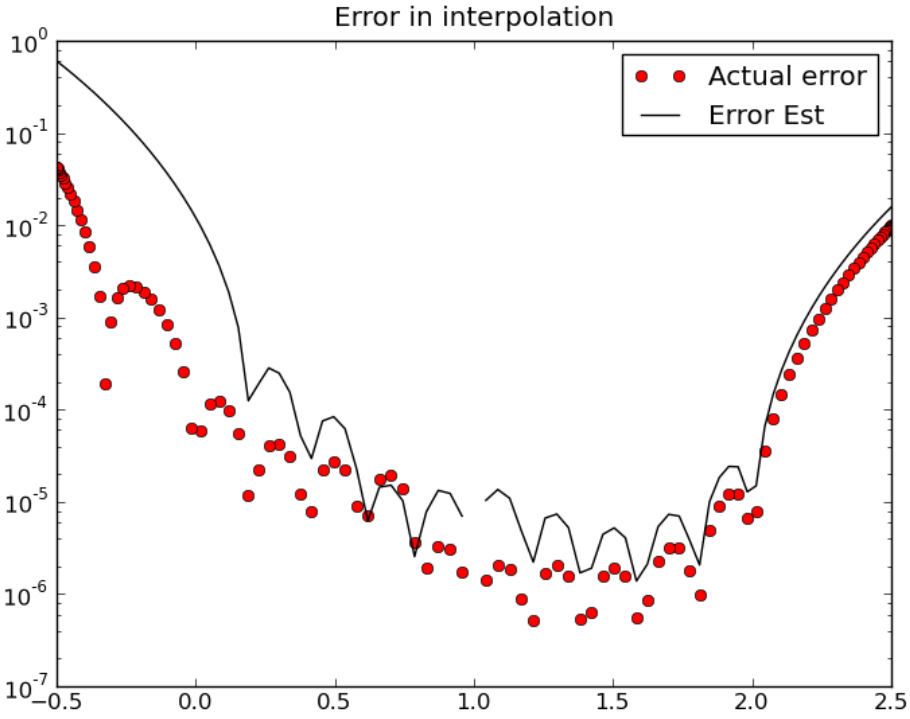
```
n1=ns-n/2;
n1=(n1<0?0:n1); // eliminate -ve n1
n1=(n1+n>=M?M-n-1:n1); // eliminate +v overflow
xa=&xarr[n1];ya=&yarr[n1];
ns-=n1;ns=(ns<n/2?n/2:ns);
ns0=ns;
// from here it is basically the NR code
for(i=0;i<=n;i++){ // initialize c and d
  c[i]=ya[i];
  d[i]=ya[i];
}
*y=ya[ns-];
for(m=1;m<=n;m++){
  for(i=0;i<=n-m;i++){
    ho=xa[i]-*x;
    hp=xa[i+m]-*x;
    w=c[i+1]-d[i];
    if((den=(ho-hp))==0.0)
      exit(1);
    den=w/den;
    d[i]=hp*den;
    c[i]=ho*den;
  } // i loop
  *y+=(*dy=(2*ns<(n-m)?c[ns+1]:d[ns-]));
} // m loop
} // j loop
"""
weave.inline(code,   ["xarr","yarr","M","xx","yy","dyy","N","c","d","n"],com
return([yy,dyy,xx])
```

Let us test this to see if it can interpolate $1/\left(1+x^2\right)$

3 ⟨*code1* 3⟩≡

```
n=6 # order of interpolation
xarr=linspace(0,2,11)
yarr=1/(1+xarr*xarr)
t=linspace(0,pi,111)
xx=cos(t)*1.5+1
z=polint(xarr,yarr,xx,n)
yy=z[0];dyy=z[1]
y0=1/(1+xx*xx)
figure(0)
plot(xx,yy,'ro',xx,y0,'k')
title("Interpolation by %dth order polynomial" % n)
figure(1)
semilogy(xx,abs(yy-y0),'ro',xx,abs(dyy),'k')
title("Error in interpolation")
legend(["Actual error","Error Est"])
show()
```





The agreement is good, but the estimate is too conservative for some x. This is typical. As I said in class, the error estimated is the error in the $n-1$ order fit.

1. In python sample $\sin\left(6x^2\right)$ from 0 to 1 at 5 points. Use these points as your table and do fourth order interpolation on

   ```
   xx=linspace(-0.5,1.5,200)
   ```

   Since all the points in the table are used for 4^{th} order interpolation, this allows you to see what the effect of choosing a window that is not centred about the desired `xx` value.

2. In python, create a table containing the values of the following truncated fourier series

$$f(x) = \sum_{k=0}^{10} (-1)^k \frac{\cos kx}{k^2}$$

for x going from 0 to 2π in 40 steps. Use Can you conclude from looking at the series whether the full series is continuous at all x? How many continuous derivitives will it have? Is this confirmed by the numerical plot? Remember back to the properties of a fourier series

3. Perform 5^{th} order interpolation on a grid of 999 points covering the same range of 0 to 2π. Plot the error vs x. Where is the error concentrated?

4. Vary the interpolation order n from 3 to 20 and determine the way the maximum error varies with order.

5. We require a 6 digit accurate method to compute the function

$$f(x) = \frac{x^{1+J_0(x)}}{\sqrt{1-x^2}} \tag{1}$$

between 0.3 and 0.7. The function is known *exactly* (to 15 digits) at certain locations, $x_k = x_0 + kdx$, $k = 0, \ldots, n$ where n is the order of interpolation required.

   (a) Convert the function to a table, spaced 0.05 apart, sampling it from 0.1 to 0.9.

   (b) In Python, plot this function and determine its general behaviour. Is it analytic in that region? What is the radius of convergence?

   (c) Use the `polint` routine to interpolate the function between 0.3 and 0.7 for different orders.

   (d) Vary the order of the interpolation used and determine the order required to achieve the 6 digit accuracy. Explain in terms of the radius of convergence and the table spacing.

6. Instead of polynomial interpolation, use spline interpolation on the same function. Note that scipy has no spline fitting routine. What it has is a B-spline fitting routine, which finds the smoothest spline through given points. This is very useful, but too advanced for our current stage in the course. So you will have to implement the C code as I have done for `polint`.

   (a) Use the *not-a-knot* method and obtain the spline coefficients and hence interpolate as above. How does the error vary with the number of spline points? How many points are required to achieve six digit accuracy?

   (b) Use Eq. 1 to compute the derivitives at the end points. Use the *clamped* method to obtain the spline coefficients. How does the error in the interpolated value compare with the *not-a-knot* method?

   (c) Use hundred times the actual derivitive as your boundary condition. How does the spline fit change? Plot the points near the edge and show the way that the errors decay.

   (d) For the 0.1 to 0.9 case, can a non-uniform spacing of spline points help reduce the number of spline points required? Try to determine the optimal spacing at which error appears to be uniform across the domain.

7. Use rational interpolation for this problem and study the dependence of error on order (i.e., repeat parts (c) and (d)).

8. Apply your interpolation methods to the following function instead

$$g(x) = \frac{1}{1+100x^2}$$

Note that you should sample it as above, but between -0.6 and $+0.6$ The question is whether the pole along the imaginary axis affects the quality of interpolation near $x = 0$ as compared to $x = \pm 0.5$. Clearly rational interpolation should work best. But what does polynomial interpolation do? Does changing order help? Can you explain your findings?

9. Use sinc interpolation on $g(x)$ and see what you get. The sampling distance should be 0.05 as in the previous problems. Study how the error falls off with number of terms. Does the error stagnate? Why or why not?

6 ⟨* 6⟩≡
⟨*code* 1⟩
⟨*code1* 3⟩