TEXAS TECH UNIVERSITY

David S. Gilliam
Department of Mathematics
Texas Tech University
Lubbock, TX 79409

806 742-2566
gilliam@texas.math.ttu.edu
http://texas.math.ttu.edu/~gilliam

Mathematics 4330/5344 – # 5
Approximation of Functions

In this lesson we will consider the use of Matlab in a brief introduction to some isolated topics in approximation theory. Often in applications one is confronted with the need to approximate a function $f(x)$ by some other, perhaps more convenient function $p(x)$ or it may happen that we only know the function $y = f(x)$ at a discrete set of points (data) of the form $y_j = f(x_j)$ and an analytic approximation $p(x)$ is need to, for example, find the maximum or minimum of the function $f(x)$ over a range of x values. It cn also happen that the original $f(x)$ is to complicated to study either directly or efficiently in which case we want to approximate $f(x)$ by a simpler function. One of the first classes of functions to use for approximations and certainly the easiest to evaluate on the computer is the class of polynomials. Every polynomial $p(x)$ of degree at most n is a finite linear combination of the basic functions

$$1,\ x,\ x^2,\ \cdots,\ x^n,$$

i.e.,

$$p(x) = a_1 + a_2x + a_3x^2 + \cdots + a_{n+1}x^n = \sum_{j=1}^{n} a_j x^{j-1}.$$

One question to ask is "what functions can be approximated by these simple functions?" A very famous result – the Wierstrass Approximation Theorem – in matheamtics is that every continuous function on an interval $[a, b]$ can be approximated to any desired accuracy uniformly by polynomials. Unfortunately, this theorem doesn't say how well a given function might be approximated. As we will see, if the function to be approximated has some smoothness – differentiability – in addition to continuity, we can often say more about how well it can be approximated. Furthermore, even for continuous functions there are many different polynomials that one can use to carryout the approximations. We will discuss these issues briefly in this lesson.

1 Polynomial Interpolation

In this section we consider the question of polynomial approximation by way of interpolation at a mesh of points. Some of the material in this section is taken from van Loan's book [2].

The specific problem can be stated as follows:

> Given $x_1, x_2, \cdots, x_n$ (distinct) and $y_1, y_2, \cdots, y_n$, find a polynomial $p_{n-1}(x)$ of degree $(n-1)$ such that $p_{n-1}(x_i) = y_i$ for $i = 1 : n$.

Vandermonde Method

We will first consider the Vandermonde Method for the basis $\{x^{j-1}\}_{j=1}^n$. In this case we seek numbers $\{a_j\}_{j=1}^n$ so that

$$p_{n-1}(x) = a_1 + a_2 x + \cdots + a_n x^{n-1}$$

satisfies the following system of n equations in n unknowns

$$a_1 + a_2 x_i + a_3 x_i^2 + \cdots + a_n x_i^{n-1} = y_i, \quad i = 1, 2, \cdots, n.$$

This system of equations can be written as

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

You have seen several methods for building the Vandermonde coefficient matrix for this system of equations. Given a vector $x = [x(1)\ x(2)\ \cdots\ x(n)]$, the simplest way to build the coefficient matrix V is to use the `vander` command in Matlab:

```
V=fliplr(vander(x))
```

where we have reversed the order of the coefficients since in Matlab a polynomial $p(x)$ of degree $(n-1)$ is considered as a vector $a = [a_n, a_{n-1}, \cdots a_1]$ of its coefficients written in descending order, i.e.

$$p(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + \cdots + a_1 = \sum_{j=1}^{n} a_{n+1-j} x^{n-j}.$$

Consider an example where we have some function $y = f(x)$ giving data $y_i = f(x_i)$ with $i = 1, 2, 3, 4$ and we want to build a polynomial of degree 3:

```
x=[-2; -1; 1; 2];
y=[10; 4; 6; 3];
V=fliplr(vander(x)) %build proper coefficient matrix
a=V\y;
aa=flipud(a); %reverse of of the coefficients
xx=-2:.1:2;   %build a mesh of points
p=polyval(aa,xx); %evaluate the poly at the mesh
plot(xx,p)
grid
```

The programs *InterpV.m* and *ShowV.m* are contained in Chapter 2 of the book [2].
This file solves the interpolation problem and returns the vector a:

```
  function a = InterpV(x,y)
%
% Pre:
%   x:  column n-vector with distinct components.
%   y:  column n-vector.
%
% Post:
%   a:  column n-vector with the property that
%       if p(x) = a(1) + a(2)x + ... a(n)x^(n-1) then
%       p(x(i)) = y(i), i=1:n
%
  n = length(x);
  V = ones(n,n);
  for j=2:n
     % Set up column j.
     V(:,j) = x.*V(:,j-1);
  end
  a = V\y;
```

This next file is a nice example of what the effect on approximation is using interpolation at four different sets of points randomly chosen in the interval $[0, 2\pi]$.

```
% Script File: ShowV
%
% Plots 4 random cubic interpolants of sin(x) on [0,2pi].
% Uses the Vandermonde method.
```

```
close all
x0 = linspace(0,2*pi,100)';
y0 = sin(x0);
for eg=1:4
   x = 2*pi*sort(rand(4,1));
   y = sin(x);
   a = InterpV(x,y);
   pvals = polyval(flipud(a),x0);
subplot(2,2,eg)
   plot(x0,y0,x0,pvals,x,y,'*')
axis([0 2*pi -2 2])
end
```

You will notice that I changed the line

```
pvals = polyval(flipud(a),x0);
```

from what it is in the file in you directory. The reason is that I did not go into building the function file *HornerV.m* to evaluate the polynomial a at $x0$. Instead I used Matlabs builtin polynomial evaluation command *polyval* after using *flipud* to reorder the polynomial coefficients.

Newton Method

Sometimes it is advantageous to use a basis different from the set $\{x^{j-1}\}_{j=1}^{n}$. For example we might consider products of monomials like $\{(x-a_i)\}$ for some numbers a_i.

As an example of this method suppose, just as in the example above, we have data

$$(x_1, y_1),\ (x_2, y_2),\ (x_3, y_3),\ (x_4, y_4).$$

In this case, instead of using the basis $1, x, x^2, \cdots, x^{n-1}$ we use

$$1,\ (x-x_1),\ (x-x_1)(x-x_2),\ (x-x_1)(x-x_2)(x-x_3)$$

and we seek coefficients $c_1,\ c_2,\ c_3,\ c_4$ so that

$$p_3(x) = c_1 + c_2(x-x_1) + c_3(x-x_1)(x-x_2) + c_4(x-x_1)(x-x_2)(x-x_3),$$

satisfies the equations

$$y_i = p_3(x_i) \text{ for } \ i = 1:4.$$

Solving for the c_i reduces to solving the system of equations

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & (x_2 - x_1) & 0 & 0 \\ 1 & (x_3 - x_1) & (x_3 - x_1)(x_3 - x_2) & 0 \\ 1 & (x_4 - x_1) & (x_4 - x_1)(x_4 - x_2) & (x_4 - x_1)(x_4 - x_2)(x_4 - x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

In the general case of n data points to find a polynomial $p_{n-1}(x)$ of degree $(n-1)$, we first observe that if we set $c_1 = y_1$, then find the polynomial

$$q(x) = c_2 + c_3(x - x_2) + \cdots + c_n(x - x_2)(x - x_3) \cdots (x - x_{n-1})$$

that interpolates the data

$$\left(x_i, \frac{y_i - y_1}{x_i - x_1} \right) \quad i = 2 : n.$$

Then our desired polynomial is

$$p_{n-1}(x) = c_1 + (x - x_1)q(x).$$

There are, of course, many ways to solve the resulting system of equations.
One method is given by the floowing code from [2]

```
  function c = Interp_N(x,y)
%
% Pre:
%   x:  column n-vector with distinct components.
%   y:  column n-vector.
%
% Post:
%   c:  a column n-vector with the property that if
%       p(x) = c(1) + c(2)(x-x(1))+...+ c(n)(x-x(1))...(x-x(n-1))
%       p(x(i)) = y(i), i=1:n.
%
%
   n = length(x);
   for k = 1:n-1
      y(k+1:n) = (y(k+1:n)-y(k)) ./ (x(k+1:n) - x(k));
   end
   c = y;
```

The following program from [2] uses recursive programming to solve for the c_j and is definitely the most efficient.

```
  function c = InterpNRecur(x,y)
%
% Pre:
%   x:  column n-vector with distinct components.
%   y:  column n-vector.
%
% Post:
%   c:  a column n-vector with the property that if
%       p(x) = c(1) + c(2)(x-x(1))+...+ c(n)(x-x(1))...(x-x(n-1))
%       p(x(i)) = y(i), i=1:n.
%
%
   n = length(x);
   c = zeros(n,1);
   c(1) = y(1);
   if n > 1
      c(2:n) = InterpNRecur(x(2:n),(y(2:n)-y(1))./(x(2:n)-x(1)));
   end
```

Another difficulty encountered here is that we need an efficient method to evaluate the resulting polynomial. The following is a Matlab code takne from [2] that provides a generalized Horner's method (synthetic division) for efficiently evaluating the resulting Newton Polynomials.

```
  function pval = HornerN(c,x,z)
%
% Pre:
%    c:   a vector.
%    x:   a vector with at least length(c)-1 components
%    z:   a vector.
%
% Post:
% pval:   a vector the same size as z with the property that if
%         p(x) = c(1) +  c(2)(x-x(1))+ ... + c(n)(x-x(1))...(x-x(n-1))
%         then pval(i) = p(z(i)) for i=1:m.
%

  n = length(c);
  pval = c(n)*ones(size(z));
  for k=n-1:-1:1
     pval = (z-x(k)).*pval + c(k);
  end
```

Efficiency and Accuracy of These Methods

The following is a code from [2] for comparing the efficiency of the Vandermode and Newton Methods:

```
% Script File: InterpEff
%
% Compares the Vandermonde and Newton Approaches

   clc home
   disp('Flop Counts:')
   disp(' ')
   disp('  n   InterpV   Interp_N     InterpNRecur')
   disp('---------------------------------------')
   for n = [4 8 16]
      x = linspace(0,1,n)'; y = sin(2*pi*x);
  flops(0); a = InterpV(x,y);      f1 = flops;
  flops(0); c = Interp_N(x,y);      f2 = flops;
  flops(0); c = InterpNRecur(x,y); f3 = flops;
  disp(sprintf('%3.0f %7.0f    %7.0f     %7.0f',n,f1,f2,f3));
end
```

This shows that the Vandermonde method is much less efficient and that the recusively solved Newton method is the most efficient.

We now know that we can interpolate a function at given points, but the question remains, "how well does it approximate the function?" The answer depends on derivatives of the function to be approximated. First we note that a polynomial of degree $(n-1)$ that interpolates n pairs (x_j, y_j) is unique. This follows because if $p_1(x)$ and $p_2(x)$ interpolate these n points then $p(x) = p_1(x) - p_2(x)$ is zero at x_i, $i = 1, 2 \cdots, n$ so that $p(x)$ is identically zero by the Fundamental Theorem of Algebra. It can be shown that

Theorem *If $p_{n-1}(x)$ interpolates a function $f(x)$ at the distinct points $\{x_j\}_{j=1}^n$ and if f is n times continuously differentiable on an interval $I = [a, b]$, then for any x in I*

$$f(x) = p_{n-1}(x) + \frac{f^{(n)}(\eta)}{n!}(x - x_1)(x - x_2)\cdots(x - x_n)$$

for some $\eta \in [a, b]$.

The following from [2] is a classic example of the problem encountered in using interpolating polynomials to approximate a function. The function used in this example, first given by Runge, is

$$f(x) = 1/(1 + 25x^2)$$

on the interval $[-1, 1]$.

```
% Script File: RungeEg
%
% For n=10:13, interpolants of f(x) = 1/(1+25x^2) on [-1,1]'
% are of plotted.
%
  close all
  x = linspace(-1,1,100)';
  y = ones(100,1)./(1 + 25*x.^2);
  for n=10:13
     figure
     xEqual = linspace(-1,1,n)';
     yEqual = ones(size(xEqual))./(1+25*xEqual.^2);;
     cEqual=Interp_N(xEqual,yEqual);
     pvalsEqual = HornerN(cEqual,xEqual,x);
     plot(x,y,x,pvalsEqual,xEqual,yEqual,'*')
     title(sprintf('Equal Spacing (n = %2.0f)',n))
  end
```

As you will see in Exercise 2, below, the choice of interpolating nodes plays a big role.

2 Approximation of smooth functions – Taylor's Method

From calculus you know that if a function is "smooth" enough (analytic) then it can be represented, at least near a certain point, as a converegent power series. For example, if f is analytic in a neighborhood of a point x_0, then we have

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

for x near x_0. By truncating the series we obtain a polynomial approximation to f:

$$f_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k.$$

On the other hand, if f is only $(n + 1)$ times differentiable near a point x_0 with the $(n+1)$st derivative continuous near x_0, then we can still approximate f with a polynomial, called the Taylor polynomial, with remainder which gives an estimate of the error.

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \frac{1}{n!}\int_{x_0}^{x} f^{(k+1)}(t)(x - t)^n \, dt$$

for x near x_0.

This formula can be used to estimate the error in approximation:

$$\left| f(x) - \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k \right| \leq \frac{M_{n,x_0}}{(n+1)!}(x-x_0)^{(n+1)}$$

where

$$M_{n,x_0} = \max_{t\in[x_0,x]} \left| f^{(n+1)}(t) \right|.$$

On the other hand there are functions for which this gives little information. Consider the C^∞ function

$$f(x) = \begin{cases} \exp(-x^{-2}), & x \neq 0 \\ 0, & x = 0 \end{cases}.$$

f is smooth but $f^{(j)}(0) = 0$ for all j so that for x near zero the Taylor expansion gives no information for x near 0. Indeed the Taylor polynomial is zero and the only nonzero term is the error term.

Similarly, there are functions which are no-where differentiable or are only a few times differentiable at a point x_0. For example, $f(x) = x^{5/2}$ is only twice continuously differentiable at $x = 0$. So the best you could do with a Taylor polynomial expansion at 0 is (for x near zero) a linear polynomial.

The following function, which is defined in terms of an infinite sum, defines a *continuous* but *no-where differentiable* function:

$$f(x) = \sum_{k=0}^{\infty} \frac{\cos(3^k x)}{2^k}.$$

Try ploting a few of the finite partial sums over an interval like $[-\pi/2, \pi/2]$ to get a feeling for what it looks like

```
% Script File: NoDiff.m
%
% Plots 4 partial sums of the continuous nowhwere differentiable function
$ on the interval [-pi/2, pi/2].

   close all
   x = linspace(-pi/2,pi/2,600);

   for k=1:4
     kk=2*k;
     d=2.^(0:-1:-kk);
     n=3.^(0:kk);
     y =d*cos(n'*x);
```

```
  subplot(2,2,k)
      plot(x,y)
  axis([-pi/2 pi/2 -2 2])
   end
```

One other disadvantage with the Taylor Method is that one has to know the derivatives of f at the point of expansion. Computing the derivatives is not so easy in some cases. So we hope there is a better way. There are several. One is to use a symbolic software package to compute the derivatives for you. Better yet, most of the computer algebra systems contain simple commands for constructing Taylor polynomials. For now my goal is for you to learn how to write programs so we will pretend the best way doesn't exist.

As an aid to get you started build the file named *fun1.m*

```
function y=fun1(x)
global pickfun

if pickfun == 1
  y=sin(x);
elseif pickfun == 2
  y=exp(x);
end
```

Now build the file *taylapp1.m* which will compute the Taylor polynomial about $x_0 = 0$ up to order 16.

```
clear
global pickfun

pickfun=input('pick function for approx, pickfun = (1 or 2 )  ');
n=input('degree of taylor poly, n = ');
a=input('approx on [-a,a], a = ');

if pickfun == 1
  v=[0 1 0 -1 0 1 0 -1 0 1 0 -1 0 1 0 -1]; % sixteen values
elseif pickfun ==2
  v=[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]; %sixteen values
end

%----- build factorials and derivatives at $0$
for k=0:n
  c(k+1)=v(k+1);
```

```
    if k==0
      d(1)=1;
    else
      d(k+1)=d(k)*k;
    end
end
%-----------

x=-a:.05:a; %grid of x values
p1=c./d;   %this is a poly in matlab
p=fliplr(p1); %this reverses the order of the coefs
fapprox=polyval(p,x); %this is the Taylor poly at  x
fxact=fun1(x); %this builds exact sol on x

err=max(abs(fxact-fapprox)); %this is the error

plot(x,fxact,x,fapprox)   %plots the exact and approximate
title(['n = ', int2str(n),' error = ', num2str(err)])
xlabel(['x-axis'])
grid
```

Run the m-file taylapp1.m for a few choices of n and for a few different choices of a. You will see that for a fixed a, as you increase n (the degree of the polynomila) the approximation gets better.

Here is your next introduction to the symbolic toolbox. Build the m-file *taylapp2.m*

```
clear
global pickfun

pickfun=input('pick function for approx, pickfun = (1 or 2 )  ');
n=input('degree of taylor poly, n = ');
a=input('approx on [-a,a], a = ');

if pickfun == 1
  f='sin(x)'
elseif pickfun == 2
  f='exp(x)'
end

for k=0:n
  y=diff(f,k);     %kth derivative using symbolic diff
  yy=subs(y,0);    %substitutes x=0
  c(k+1)=eval(yy); %convert to a floating point number
```

```
    if k==0
       d(1)=1;
    else
       d(k+1)=d(k)*k;
    end
end

p1=c./d;
p=p1((n+1):-1:1);

xx=-a:.05:a;
fxact=fun1(xx);
fapprox=polyval(p,xx);

err=max(abs(fxact-fapprox));

plot(xx,fxact,xx,fapprox)
title(['n = ', int2str(n),' error = ', num2str(err)])
xlabel(['x-axis'])
grid
```

Run the program *taylapp2.m* for $n = 10, 15, 20, 25$ for $a = \pi$. Then repeat this for $a = 3\pi$. By now you should get the idea that to get a good approximation for a given interval $[-a, a]$, you will need to increase n.

3 Approximation of Continuous Functions

The Taylor Method cannot be used to approximate functions which happen to only be continuous. Nevertheless there is a very famous theorem in mathematics which states:

Theorem (Weierstrass Approximation Theorem) If f is a continuous function on an interval $[a, b]$ and $\epsilon > 0$ is given, then there exists a polynomial $p(x)$ such that

$$\sup_{x\in[a,b]} |f(x) - p(x)| \leq \epsilon.$$

The proof of this result is often given constructively using the Bernstein polynomials. In particular, given a continuous function f we have

Theorem *Given $f \in C[0,1]$, define*

$$B_n(f,x) = \sum_{k=0}^{n} f\left(\frac{k}{n}\right) \left(\begin{array}{c} n \\ k \end{array} \right) x^k (1-x)^{(n-k)}$$

Then we have

$$\sup_{x \in [0,1]} |f(x) - B_n(f,x)| \leq \epsilon(n).$$

where $\epsilon(n) \to 0$ as $n \to \infty$.

Here is a code *bern1.m* for approximating a continuous function of the interval $[0,1]$.

```
clear
global pickfun

pickfun=input('pick function for approx, pickfun = (1 or 2 )  ');
n=input('degree of Bernstein poly, n = ');

x=0:.05:1;

fxact=fun1(x);
vec=(0:n)/n;
fvec=fun1(vec);
p=pascal(n+1);
comb=diag(rot90(p))';
fc=fvec.*comb;% builds the vector f(k/n)c(n,k)

bapprox=0; %set the approx = 0
for k=1:(n+1)
bapprox=bapprox+fc(k)*x.^(k-1).*(1-x).^(n-(k-1));
end

err=max(abs(fxact-bapprox));

plot(x,fxact,x,bapprox)
title(['n = ', int2str(n),' error = ', num2str(err)])
xlabel(['x-axis'])
grid
```

Now is a very good time to learn an important lesson. Most famous algorithms like the Bernstein polynomials for approximating a function are given on a specific interval $[c,d]$, say. But we would like to use these procedures on some other interval $[a,b]$. Fortunately

there is a simple way around this problem. Suppose you have a scheme for approximating a function $\phi(t)$ on $[c,d]$ with $\{t_j\} \subset [c,d]$

$$\phi(t) \approx \sum_{j=1}^{n} \phi(t_j)\ell_j(t), \quad t \in [c,d],$$

and you want to use this to approximate a function $f(x)$ on an interval $[a,b]$.

The formula

$$x = a + \frac{(b-a)}{(d-c)}(t-c)$$

maps the interval $[c,d]$ onto the interval $[a,b]$ and the formula

$$t = c + \frac{(d-c)}{(b-a)}(x-a)$$

maps the interval $[a,b]$ onto the interval $[c,d]$. Using these we approximate the function f on $[a,b]$ by

$$f(x) \approx \sum_{j=1}^{n} f\big(a + \frac{(b-a)}{(d-c)}(t_j - c)\big)\ \ell_j\big(c + \frac{(d-c)}{(b-a)}(x-a)\big).$$

The following code *bern2.m* includes this modification.

```
% bern2.m used for an interval [a,b] instead of [0,1]
clear
global pickfun

a=input('left endpoint a = ');
b=input('right endpoint b = ');
c=0;
d=1;
pickfun=input('pick function for approx, pickfun = (1 or 2 )  ');
n=input('degree of Bernstein poly, n = ');
x=a:.05:b;
t=c+(d-c)/(b-a)*(x-a);
fxact=fun1(x);
tvec=(0:n)/n;
fvec=fun1(a+(b-a)/(d-c)*(tvec-c));

p=pascal(n+1);
comb=diag(rot90(p))';
% builds the vector f(k/n)c(n,k)
fc=fvec.*comb;
```

```
%set the approx = to 0
bapprox=0;
for k=1:(n+1)
bapprox=bapprox+fc(k)*t.^(k-1).*(1-t).^(n-(k-1));
end

err=max(abs(fxact-bapprox));
plot(x,fxact,x,bapprox)
title(['n = ', int2str(n),' error = ', num2str(err)])
xlabel(['x-axis'])
grid
```

You will note that the approximation can be rather slow. Please try a few examples by altering the function file *fun1.m* to the file *fun2.m* as follows

```
function y=fun2(x)
global pickfun

if pickfun == 1
  y=sin(x);
elseif pickfun == 2
  y=exp(x);
elseif pickfun == 3
  y=1./(x.^2+1);
elseif pickfun == 4
  y=x.*cos(2*x.^2);
elseif pickfun == 5
  y=abs(x);
elseif pickfun == 6
  y=x.^(2/3);
end
```

For the new program *bern2.m* change the call from the function *fun1.m* to *fun2.m* and try some of the functions in this function file. Note that the last two examples cannot be handled using Taylor polynomials. For pickfun $= 6$ you should use $a \geq 0$.

Chebyshev Polynomial Approximation

We have already seen that the choice of nodes and basis functions can have considerable impact on how well a polynomial approximation works. We now turn to an almost magical set of basis functions, the Chebyshev polynomials, $T_n(x)$. These basis functions are only one of a special general class of polynomials refered to as orthogonal polynomials – what ever that means doesn't matter now. We consider the Chebyshev

polynomials $T_n(x)$ which are polynomials that can be defined a great many ways. Each of the following can be used to define the Chebyshev polynomials:

1. $T_n(x) = \cos(n \operatorname{acos}(x))$ for $n = 0, 1, 2, \cdots$.
2. With $T_0(x) = 1$, $T_1(x) = x$, then $T_{n+1}(x) = 2xT_n - T_{n-1}(x)$ for $n \geq 1$.
3. $T_n(x) = \dfrac{1}{2}\left((x - \sqrt{x^2-1})^n - (x + \sqrt{x^2-1})^n\right)$
4. $\dfrac{1-t^2}{1-2xt+t^2} = T_0(x) + 2\displaystyle\sum_{n=1}^{\infty} T_n(x)t^n$
5. Up to a constant $T_n(x)$ is the polynomial solution of the differential equation
$$(1-x^2)u'' - xu' + n^2 u = 0 \quad \text{on} \quad [-1, 1].$$
6. They are the orthogonal polynomials with respect to the weight function $1/\sqrt{1-x^2}$ on the interval $[-1, 1]$, i.e.,
$$\int_{-1}^{1} \frac{T_j(x)T_k(x)}{\sqrt{1-x^2}}\, dx = \begin{cases} \dfrac{\pi}{2} & j = k \\ 0 & j \neq k \end{cases}$$

Now lets use the Chebyshev polynomials to approximate a function on the interval $[-1, 1]$. The formula is really quite simple:

Theorem: *A continuous function f on the interval $[-1, 1]$ has the following Chebyshev approximation*

$$f(x) \approx \sum_{k=0}^{n} c_k T_k(x)$$

where

$$c_0 = \frac{1}{n+1}\sum_{k=0}^{n} f(x_k)$$

$$c_j = \frac{2}{n+1}\sum_{k=0}^{n} f(x_k)\cos(j \operatorname{acos}(x_k)),\ j = 1, \cdots, n$$

and where

$$x_k = \cos\left(\left(\frac{2k+1}{2(n+1)}\right)\pi\right),\ k = 0, 1, \cdots, n$$

$$T_k(x) = \cos(k \operatorname{acos}(x)),\ k = 0, 1, \cdots, n$$

Here is a program that, I call cheb.m that works on the interval $[-1, 1]$ to approximate functions in the function file fun2.m which you have already built. Notice a new feature in this program which makes a fancy display menu for you to pick the given function. When you run the program a menu will appear and you can click the mouse button on the problem of your choice.

```
clear
clear global
global pickfun

pickfun=menu('choose an f','sin(x)','exp(x)','1/(x^2+1)','log(x^2+5)', ...
'x cos(2x^2)','abs(x)','x^(3/2)');

n=input('degree of chebyshev poly, n = ');

x=linspace(-1,1,40);
%build the Chebyshev nodes x_k
vec=(0:n);
nodes=cos((2*vec+1)*pi/(2*(n+1)));
%evaluate f at the nodes
fnodes=fun2(nodes);

fxact=fun2(x); %builds exact solution on x

%build the coefficients for Chebyshev approximation
%note we have to start indexing of vectors with 1 not 0
c(1)=sum(fnodes)/(n+1);
for j=2:(n+1)
c(j)=(2/(n+1))*fnodes*cos((j-1)*acos(nodes))';
end
%build T_k(x)
for k=1:(n+1)
cheb(k,:)=cos((k-1)*acos(x));
end
%finally here is the approximation
chebapp=c*cheb;
%This is the error
err=max(abs(fxact-chebapp));
plot(x,fxact,x,chebapp)
title(['n = ', int2str(n),' error = ', num2str(err)])
xlabel(['x-axis'])
grid
```

Notice that the approximation of f requires the values of f at the zeros of the $(n+1)$st Chebyshev polynomial.

The following is a sample code I called *cheb2.m* that carries out Chebyshev approximation on an interval $[a, b]$.

```
clear
clear global
global pickfun

pickfun=menu('choose an f','sin(x)','exp(x)','1/(x^2+1)','log(x^2+5)', ...
'x cos(2x^2)','abs(x)','x^(3/2)');

n=input('degree of chebyshev poly, n = ');
a=input('left endpoint a = ');
b=input('right endpoint b = ');
c=-1;
d=1;
x=linspace(a,b,fix(b-a)*20);
t=c+(d-c)/(b-a)*(x-a);
fxact=fun2(x); %builds exact solution on x
%build the Chebyshev nodes x_k
vec=(0:n);
nodes=cos((2*vec+1)*pi/(2*(n+1)));
%evaluate f at the nodes
fnodes = fun2(a+(b-a)/(d-c)*(nodes-c));
c(1)=sum(fnodes)/(n+1);
for j=2:(n+1)
c(j)=(2/(n+1))*fnodes*cos((j-1)*acos(nodes))';
end
%build T_k(x)
for k=1:(n+1)
cheb(k,:)=cos((k-1)*acos(t));
end

%finally here is the approximation
chebapp=c*cheb;

%This is the error
err=max(abs(fxact-chebapp));
plot(x,fxact,x,chebapp)
title(['n = ', int2str(n),' error = ', num2str(err)])
xlabel(['x-axis'])
grid
```

Non-Polynomial Approximation

A good question is, "what can be said concerning non-polynomial approximation?" The fact is that there are many such approximations. We now consider just one such approximation method that requires knowledge of certain values of the function we want to approximate in order to obtain an approximation on the whole real number line. This is a very important example both historically and in practice. The historical underpinnngs goes back to the following famous theorem.

Theorem (Shannon Sampling Theorem) If f is an analytic function such that

1. $\displaystyle\int_{-\infty}^{\infty} |f(x)|^2\, dx < \infty$

2. There exist $h > 0$ and $K > 0$ such that $|f(z)| \leq K \exp\left(\dfrac{\pi|z|}{h}\right)$ for all complex numbers z.

Then

$$f(x) = \sum_{k=-\infty}^{\infty} f(kh)\,\mathrm{sinc}\left(\frac{x-kh}{h}\right)$$

for all real numbers x where

$$\mathrm{sinc}(x) = \begin{cases} \dfrac{\sin(\pi x)}{\pi x}, & x \neq 0 \\ 1, & x = 0 \end{cases}.$$

The conditions on f in the electrical engineering literature are that f is a band limited signal in the sense of low pass filters. Once we have talked about Fourier transforms this will make more sense. What the result says is that we only need to know the values of such a function at the equally spaced points kh where h is a positive number called the bandwith and the k values are the positive and negative integers and zero. This is a basis for digital signal processing for signals of known frequency limits.

The important thing for us is that for quite general smooth functions, the truncation of the infinite sum on the right above gives a very good approximation. Namely,

$$f(x) \approx \sum_{k=-N}^{N} f(kh)\,\mathrm{sinc}\left(\frac{x-kh}{h}\right), \quad h = \left(\frac{\pi}{N}\right)^{1/2}.$$

This method of approximation, just like the earlier ones, can be applied to approximate functions which are only given on an interval (a, b) but, in contrast, the functions

must be zero at the end points. The procedure goes as follows, let f be a function given on an interval (a, b) and such that $f(a) = 0$ and $f(b) = 0$. Then define

$$\phi(x) = \log\left(\frac{x-a}{b-x}\right), \quad x_k = \frac{b\exp(kh)+a}{\exp(kh)+1}, \ k = -N, \cdots, N.$$

With this we have, for all $x \in (a, b)$,

$$f(x) \approx \sum_{k=-N}^{N} f(x_k)\,\mathrm{sinc}\left(\frac{\phi(x)-kh}{h}\right), \quad h = \left(\frac{\pi}{N}\right)^{1/2}.$$

The points x_k are the image of the values kh under the inverse of the function ϕ, i.e., if $z = \phi(x)$ then $x = (be^z + a)/(e^z + 1)$.

Heres one possible version of the file sinc.m

```
function y=sinc(x)
y=sin(pi*x+eps)./(pi*x+eps);
```

In this program I have introduced the value *eps* (see help eps) which is a number in matlab just larger than machine zero so that our sinc function can be evaluated at $x = 0$. More importantly it is set up so that it can be evaluated at a vector x.

Here is one version of the file for sinc approximation of functions that are zero at the end points (I called it *sincapp0.m*) To use this program you will need to build a function file named *funs0.m* which can be constructed by loading *fun2.m*, modifying it and saving it as *funs0.m*. Use the functions

1. $\sin^2(\pi x)$ on $[-L, L]$
2. $x(1-x)$ on $[0, 1]$
3. $\exp(-1/(1-x^2)$ on $[-1, 1]$
4. $(1-x^2)^{1/2}$ on $[-1, 1]$

```
clear
clear global

global pickfun

N=input(' Number of sinc nodes,  N = ');
h=sqrt(pi/N);

pickfun=menu('choose an f and (a,b)','sin^2(pi*x) on [-L,L]',...
'x(1-x) on [0,1]','exp(-1/(1-x^2)) on [-1,1]','sqrt(1-x^2) on [-1,1]');
```

```
%set up the intervals for each function
if pickfun == 1
L=input('approx on [-L,L],  L =  ')
x=-L:.05:L;
a=-L;
b=L;
x(1)=a+eps;
x(length(x))=b-eps;
elseif pickfun ==2
x=0:.05:1;
a=0;
b=1;
x(1)=a+eps;
x(length(x))=b-eps;
elseif pickfun ==3
x=-1:.05:1;
a=-1;
b=1;
x(1)=a+eps;
x(length(x))=b-eps;
elseif pickfun ==4
x=-1:.05:1;
a=-1;
b=1;
x(1)=a+eps;
x(length(x))=b-eps;
end

kh=(-N:N)*h;
node=(b*exp(kh)+a)./(exp(kh)+1);

fnodes=funs0(node);

fxact=funs0(x);
phix=log((x-a)./(b-x));
    % now build (phi(x)-kh)/h as a matrix
mat=(ones(size(kh'))*phix-kh'*ones(size(phix)))/h;
    % this next matrix product builds the sinc approximation
sapprox = (fnodes*sinc(mat));

err = max(abs(fxact-sapprox));
h=figure;
```

```
plot(x,fxact,x,sapprox)

title([' N = ',num2str(N),'  error = ',num2str(err)]);
grid
```

Now this method would not be very practical if we could only approximate functions that are zero at the end points. Let us rectify this situation now. Suppose we have a general function $f(x)$ on (a, b). Let us define a new function

$$g(x) = -\frac{(b-x)}{(b-a)}f(a) + f(x) - \frac{(x-a)}{(b-a)}f(b).$$

This function satisfies $g(a) = 0$ and $g(b) = 0$ so we can apply the above procedure to approximate $g(x)$ on (a, b) say by $g_a(x)$, then we obtain an approximation for f, say $f_a(x)$ by

$$f_a(x) = \frac{(b-x)}{(b-a)}f(a) + g_a(x) + \frac{(x-a)}{(b-a)}f(b).$$

```
clear
clear global

global pickfun pp cc

N=input(' Number of sinc nodes,  N = ');

h=sqrt(pi/N);

pickfun=menu('choose an f','sin(x)','exp(x)','1/(x^2+1)','log(x^2+5)', ...
'x cos(2x^2)','abs(x)','x^2/3');

a=input(' left end point  a = ');
b=input(' right end point  b = ');


kh=(-N:N)*h;
node=(b*exp(kh)+a)./(exp(kh)+1);

fnodes=fun2(node);
gnodes=-(b-node)/(b-a)*fnodes(1)+fnodes- ...
(node-a)/(b-a)*fnodes(2*N+1);

x=a:.05:b;
lx=length(x);
```

```
x(1)=a+10^(-5);
x(lx)=b-10^(-5);

fxact=fun2(x);
phix=log((x-a)./(b-x));
mat=(ones(size(kh'))*phix-kh'*ones(size(phix)))/h;
sapprox1 = (gnodes*sinc(mat));
sapprox=(b-x)/(b-a)*fun2(x(1))+sapprox1+(x-a)/(b-a)*fun2(x(lx));

err = max(abs(fxact-sapprox));

plot(x,fxact,x,sapprox)
title([' N = ',num2str(N),'  error = ',num2str(err)]);
grid
```

ASSIGNMENT 5 – Math 4330 and 5344

1. Write a function file to build the system of equations for Newton's Method and then solve the system of equations using the simple \ in Matlab to obtain the coefficients of the Newton interpolating polynomial. Check the number of flops to do this compared to the file *InterpNRecur.m*. Simply modify the file *InterpEff.m* to do this comparison. Note to plot the result you must use *HornerN.m* to evaluate at a set of point $x \in [-1, 3]$. Plot your Newton polynomial with coefficients c against the polynomial $y = x^2 - 2x + 1$ on $[-1, 3]$.

   ```
   xx=-1:.05:3;
   plot(xx,polyval([1 -2 1],xx),xx,HornerN(c,x,xx'))
   ```

   What is you conclusion concerning these two polynomials?

2. Redo the Runge example *RungeEG.m* that gave bad approximation but this time using the interpolation points xEqual $= \cos((2(1:n)-1)\pi/(2n))'$. These points are called the Chebyshev nodes. Also run the problem for $n = 25$ and 27.

3. Write a matlab program to build the Chebyshev polynomials $T_n(x)$ of degrees 1 to n (where n can be input). Use the definitions

   (a) $T1_n(x) = \cos(n \operatorname{acos}(x))$ for $n = 0, 1, 2, \cdots$.

   (b) With $T2_0(x) = 1$, $T2_1(x) = x$, then $T2_{n+1}(x) = 2xT2_n - T2_{n-1}(x)$ for $n \geq 1$.

   (c) $T3_n(x) = \dfrac{1}{2}\left((x - \sqrt{x^2-1})^n - (x + \sqrt{x^2-1})^n\right)$

to construct the Chebyshev polynomials three different ways. Plot the first six Chebyshev polynomials for $T1$, $T2$ and $T3$ on the interval $[-1, 1]$ on the same axis. Notice that $|T_n(x)| \leq 1$ for all x. Also notice that the resulting plot should only show six curves since the six curves for each method should be on top of each other.

4. Write an m-file to input a positive integer N and compare the approximation given by the Taylor polynomial of degree N, the Vandermode and Berstein methods at N equal spaced nodes, the sinc method of order N at the sinc nodes and the Chebyshev method of degree N at the Chebyshev nodes. Make this comparison for the function $f(x) = \exp(-x^2)$ on the interval $[-2, 2]$. Print out a table of the errors for each along the row and for $N = 2, 4, 6, 8$.

Math 5344 only

1. Using the definition in terms of cos and acos, apply the calculus method of trig substitution to show that the sixth property (orthogonality) is valid.

2. Try to show that T_n satisfies the differential equation in statement five above.

3. Show that the zeros of $T_n(x)$ are

$$x_k = \cos\left(\frac{(2k+1)\pi}{2n}\right), \quad k = 0, 1, 2, \cdots (n-1).$$

References

[1] *The Matlab Primer,* Kermit Sigmon

[2] *Introduction to scientific computing: a matrix vector approach using Matlab*, Prentice Hall, 1997, Charles Van Loan

[3] *Mastering Matlab,* Printice Hall, 1996, Duane Hanselman and Bruce Littlefield

[4] *Advanced Mathematics and Mechanics Applications Using Matlab,* CRC Press, 1994, Howard B. Wilson and Louis H. Turcotte

[5] *Engineering Problem Solving with Matlab*, Printice Hall, 1993, D.M Etter

[6] *Solving Problems in Scientific Computing Using Maple and Matlab,* Walter Gander and Jiri Hrebicek

[7] *Computer Exercises for Linear Algebra*, Printice Hall, 1996, Steven Leon, Eugene Herman, Richard Faulkenberry.

[8] *Contemporary Linear Systems using Matlab*, PWS Publishing Co., 1994, Robert D. Strum, Donald E. Kirk