MATH2070: LAB 9: Legendre Polynomials and L^2 Approximation

Introduction	Exercise 1
Integration	Exercise 2
Legendre Polynomials	Exercise 3
Orthogonality and Integration	Exercise 4
Least squares approximations in $L^2([-1,1])$	Exercise 5
Legendre polynomial approximation	Extra Credit
Fourier series	
Piecewise constant approximation	
Piecewise linear approximation (Extra)	

1 Introduction

With *interpolation* we were given a formula or data about a function $f(x)$, and we made a model $p(x)$ that passed through a given set of data points. We now consider *approximation*, in which we are still trying to build a model, but specify some condition of "closeness" that the model must satisfy. It is still likely that $p(x)$ will be equal to the function $f(x)$ at some points, but we will not know in advance which points.

As with interpolation, we build this model out of a set of basis functions. The model is then a linear combination with coefficients c that specify how much of each basis function to use when building the model.

In this lab we will consider four different selections of basis functions in the space $L^2([-1,1])$. The first is the usual monomials 1, x, x^2, and so on. In this case, the coefficients c are exactly the coefficients Matlab uses to specify a polynomial. The second is the set of Legendre polynomials, which will yield the same approximations but will turn out to have better numerical behavior. The third selection is the trigonometric functions, and the final selection is a set of piecewise constant functions. Advantages and disadvantages of each for numerical computations will be presented.

Once we have our basis set, we will consider how we can determine the approximating function $p(x)$ as the "best possible" approximate for the given basis functions, and we will look at the behavior of the approximation error. Since we are working in $L^2[-1,1]$, we will use the L^2 norm to measure error.

It turns out that approximation by monomials results in a matrix similar to the Hilbert matrix whose inversion can be quite inaccurate, even for small sizes. This inaccuracy translates into poor L^2 approximations. Use of orthogonal polynomials such as the Legendre polynomials, results in a diagonal matrix that can be inverted almost without error, but the right side can be difficult and slow to compute to high accuracy. In addition, roundoff accumulation in function reconstruction can be a serious problem. Fourier approximation substantially reduces the roundoff errors, but is slow to compute and evaluate, although fast methods (not discussed in this lab) can improve speed dramatically. Approximation by piecewise constants is not subject to these sources of error until ridiculously large numbers of pieces are employed, but can be slow to converge.

We will be attempting to approximate several functions in this lab, all on the interval [-1,1]. These functions include:

- The Runge example function, $f(x) = 1/(1+x^2)$, `runge.m`.
- The function

$$f(x) = \begin{cases} 0 & -1 \le x < 0 \\ x(1-x) & 0 \le x \le 1 \end{cases} \tag{1}$$

  For the purpose of this lab, this function will be called "partly quadratic." It was chosen because it is simple, continuous and satisfies $f(-1) = f(1)$, but is not differentiable. A simple Matlab function m-file to compute this "partly quadratic" function can be found by copying the following code:

```
function y=partly_quadratic(x)
% y=partly_quadratic(x)
% input x (possibly a vector or matrix)
% output y, where
% for x<=0, y=0
% for x>0,  y=x(1-x)

y=(heaviside(x)-heaviside(x-1)).*x.*(1-x);
```

Remark: The "heaviside" function (sometimes called the "unit step function") is named after Oliver Heaviside and is defined as

$$f(x) = \begin{cases} 0 & x < 0 \\ 0.5 & x = 0 \\ 1 & 0 < x \end{cases} \tag{2}$$

The heaviside function is part of Matlab.

- A third function is a function whose graph is shaped like the teeth of a saw, similar to one used in Lab 6:

$$f(x) = \begin{cases} (x+1) & -1 \le x < 0 \\ 0 & x = 0 \\ (x-1) & 0 < x \le 1 \end{cases} \tag{3}$$

A simple Matlab function m-file to compute this sawshape function can be found by downloading `sawshape9.m` from the web site or by copying the following code:

```
function y=sawshape9(x)
% y=sawshape9(x)
% input x (possibly a vector or matrix)
% output y, where
%   y=(x+1) for -1<=x<0
%   y=0     for x=0
%   y=(x-1) for 0<x<=1

y= heaviside(-x).*(x+1) + heaviside(x).*(x-1);
```

Remark: In Lab 6, you saw a similar function defined using the Matlab `find` command, while `heaviside` is used here. There is no essential reason for this change.

This lab will take four sessions. If you print this lab, you may prefer to use the pdf version.

1.1 Matlab remarks

This kind of approximation requires evaluation of integrals. We will use the Matlab numerical integration function `integral`. Since we will be computing fairly small errors, will replace the default error tolerances with smaller ones. The Matlab command is

```
q=integral(func,-1,1,'AbsTol',1.e-14,'RelTol',1.e-12);
```

where `func` is a function handle to a function written using vector (array) syntax. This command will result in an approximation, q, satisfying

$$\left| q - \int_{-1}^{1} \text{func}(x)dx \right| \le \max\{10^{-14}, 10^{-12}q\}.$$

WARNING: The `integral` function was introduced into Matlab in 2012. If your version of Matlab is older than that, you can use the `quadgk` function for this lab. The calling sequence for that function is

```
q=quadgk(func,-1,1,'AbsTol',1.e-14,'RelTol',1.e-12, ...
                   'MaxIntervalCount',15000);
```

1.2 `ntgr8`, an abbreviation function

Since these integration functions involve extra parameters and extra typing, You should create an abbreviation for them for use in this lab. Create an m-file named `ntgr8` in the following way:

```
function q=ntgr8(func)
% q=ntgr8(func) integrates func over [-1,1] with tight tolerances

q=integral(func,-1,1,'AbsTol',1.e-14,'RelTol',1.e-12);
```

with a similar function if you are using `quadgk`. This function will save you some typing for this lab. The name looks strange in print, but it is easy to remember because it is a pun. The letter `n` can be pronounced "en," the letter `t` can be pronounced "tee" and `gr8` can be pronounced "grate." Put them together and it sounds like "integrate."

2 Least squares approximations in $L^2([-1,1])$

The problem of L^2 approximation can be described in the following way.

Problem: Given a function $f \in L^2$, and a (complete) set of functions $\phi_\ell(x) \in L^2$, $n = 1, 2, \ldots$, then for a given N, find the set of values $\{c_\ell | \ell = 1, 2, \ldots, N\}$ so that

$$f(x) \approx \sum_{\ell=1}^{N} c_\ell \phi_\ell \tag{4}$$

and the approximation is best in the sense of having the smallest L^2 error.

Quarteroni, Sacco, and Saleri, in Section 10.7, discuss least-squares approximation in function spaces such as $L^2([-1,1])$. The idea is to minimize the norm of the difference between the given function and the approximation. Given a function f and a set of approximating functions (such as the monomials $\{x^{k-1} : k = 1, 2, \ldots, n\}$), for each vector of numbers $\mathbf{c} = (c_\ell)$ define a functional

$$F(\mathbf{c}) = \int_{-1}^{1} \left(f(x) - \sum_{\ell=1}^{n} c_\ell x^{n-\ell} \right)^2 dx.$$

This continuous functional becomes large when $\|\mathbf{c}\|$ is large and it is bounded below by 0, so it must have a minimum, and because the function is differentiable as a function of $\mathbf{c}$, the minimum must occur when

$$\frac{\partial F}{\partial c_\ell} = 0 \qquad \text{for } \ell = 1, \ldots, n.$$

This expression is evaluated here for the case of quadratic approximations ($n = 3$).

Consider the functional

$$F = \int_{-1}^{1} \Big(f(x) - (c_1 x^2 + c_2 x + c_3) \Big)^2 dx$$

where f is the function to be approximated on the interval $[-1,1]$. Taking partial derivatives with respect to c_i yields the equations

$$\begin{aligned}
\frac{\partial F}{\partial c_1} &= -2\int_{-1}^{1}(f(x)-(c_1x^2+c_2x+c_3))x^2dx \\
\frac{\partial F}{\partial c_2} &= -2\int_{-1}^{1}(f(x)-(c_1x^2+c_2x+c_3))xdx \\
\frac{\partial F}{\partial c_3} &= -2\int_{-1}^{1}(f(x)-(c_1x^2+c_2x+c_3))dx
\end{aligned}$$

and setting each of these to zero yields the system of equations

$$\begin{array}{ccccccc}
c_1\int_{-1}^{1}x^4dx & + & c_2\int_{-1}^{1}x^3dx & + & c_3\int_{-1}^{1}x^2dx & = & \int_{-1}^{1}x^2f(x)dx \\
c_1\int_{-1}^{1}x^3dx & + & c_2\int_{-1}^{1}x^2dx & + & c_3\int_{-1}^{1}xdx & = & \int_{-1}^{1}xf(x)dx \\
c_1\int_{-1}^{1}x^2dx & + & c_2\int_{-1}^{1}xdx & + & c_3\int_{-1}^{1}dx & = & \int_{-1}^{1}f(x)dx
\end{array}$$

or

$$\begin{array}{ccccccc}
(2/5)c_1 & + & 0 & + & (2/3)c_3 & = & \int_{-1}^{1}x^2f(x)dx \\
0 & + & (2/3)c_2 & + & 0 & = & \int_{-1}^{1}xf(x)dx \\
(2/3)c_1 & + & 0 & + & (2/1)c_3 & = & \int_{-1}^{1}f(x)dx
\end{array} \tag{5}$$

(Since the interval of integration is symmetric about the origin, the integral of an odd monomial is zero.)

Equation (**??**) is related to Equations (10.1) and (10.2) in Quarteroni, Sacco, and Saleri, but their presentation focusses on orthogonal polynomials. For an arbitrary value of n, Equation (**??**) can be written in the following way, where the indexing corresponds with Matlab indexing (starting with 1 instead of 0) and with the Matlab convention for coefficients of polynomials (first coefficient is for the highest power of x.)

$$\sum_{\ell=1}^{n}\left(\int_{-1}^{1}x^{(2n-k-\ell)}dx\right)c_\ell = \int_{-1}^{1}x^{n-k}f(x)dx \qquad \text{for } k=1,\ldots,n. \tag{6}$$

The matrix in (**??**),

$$H_{k,\ell} = \int_{-1}^{1}x^{(2n-k-\ell)}dx = (1-(-1)^{(n-\ell)+(n-k)+1})/((n-\ell)+(n-k)+1)$$

is closely related to the Hilbert matrix. In fact, if the derivation above were done over the interval [0,1] instead of [-1,1], the matrix that arose would be the Hilbert matrix. The Hilbert matrix is notorious for having a very poor condition number and being difficult to invert without losing accuracy. The following exercise illustrates this difficulty and its implication for approximation.

Examining Equation (**??**), there are two sets of integrals that need to be evaluated in order to compute the coefficients c_k. On the left side, the integrands involve the products $x^{(n-k)}x^{(n-\ell)}$. On the right side, the integrands involve the products $x^{(n-k)}f(x)$. Please note that these expressions are made more complicated by the fact that they are indexed "backwards." This is done to be consistent with Matlab's numbering scheme for coefficients. Later in the lab when we switch to Legendre polynomials and are free to number the coefficients as we wish, we will switch to a simpler numbering scheme.

Once the coefficients c_k have been found, the Matlab `polyval` function can be used to evaluate the resulting polynomials.

> **Exercise 1**: In this exercise, you will be writing a function m-file to compute the coefficient vector of the best $L^2([-1,1])$ approximation to a function $f(x)$ using Equation (**??**) above. This m-file will have the signature

```
function c=coef_mon(func,n)
% c=coef_mon(func,n)
% func is a function handle
% ...  more comments  ...

% your name and the date
```

and be called `coef_mon.m`. It will solve the system (**??**) by constructing the matrix H and right side $\mathbf{b}$ and solving the resulting system for c_k. In `coef_mon.m`, `func` refers to a function handle.

(a) Begin `coef_mon.m` with the above code.

(b) $\mathbf{b}_k = \int_{-1}^{1} x^{n-k} f(x) dx$.

```
for k=1:n
  % force b to be a column vector with second index
  b(k,1)=ntgr8(@(x) func(x).*x.^(n-k));
end
```

Warning: You should already have created the abbreviation function `ntgr8.m` according to instructions in Section **??** above on Matlab remarks contained in the introduction to this lab.

(c) Complete the following code to compute the matrix elements `H(k,ell)` using the formula $H_{k,\ell} = \int_{-1}^{1} x^{(2n-k-\ell)} dx$. Your code will be similar to the above code for `b(k)`.

```
for k=1:n
  for ell=1:n
    H(k,ell)=ntgr8( ??? )
  end
end
```

Note 1: I try not to use the letter "`l`" as a variable because it looks so much like the number 1. Instead, I use `ell`.
Note 2: Time could be saved in the above code by taking advantage of the fact that `H` is a symmetric matrix.
Note 3: It is not necessary to write code to compute the quantities $H_{k,\ell}$ because you can easily write out the values of the integrals $\int_{-1}^{1} x^{(2n-k-\ell)} dx$. I prefer that you use the approach described above, though, to help you understand it.

(d) Solve for the coefficients with

```
c=H\b;
```

Do not be surprised if Matlab warns you that `H` is poorly conditioned for larger values of `n`.

(e) Verify your code is correct by computing the best 3-term approximation by monomials for the polynomial $f(x) = 3x^2 - 2x + 1$. The result should be the coefficient vector for the polynomial f itself.

(f) Write a script m-file named `test_mon.m` containing code similar to the following

```
func=@partly_quadratic;
c=coef_mon(func,n);

xval=linspace(-1,1,10000);
yval=polyval(c,xval);
yexact=func(xval);
plot(xval,yval,xval,yexact)
```

```
% relative Euclidean norm is approximating
% the relative integral least-squares (L2 norm)
% using an approximate trapezoid rule
relativeError=norm(yexact-yval)/norm(yexact)
```

and use it to evaluate the approximation for `n=1` and `n=5`. Look at the plot and estimate by eye if the area between the exact and approximate curves is divided equally between "above" and "below." Further, the error for `n=5` should be smaller than for `n=1` and the plot should look much better, but still far from perfect.

(g) You do not need to send me copies of the plots, but fill in the following table using the Runge example function. When you report errors, please use at least three significant digits. One easy way to get good precision is to use `format short e`. You should find that the error gets smaller for early values of `n` and then deteriorates. At what value of `n` does the smallest relative error occur? (You may get warnings that the matrix H is almost singular.)

Runge	
n	relative error
1	_____
2	_____
3	_____
4	_____
5	_____
6	_____
10	_____
20	_____
30	_____
40	_____

(h) You should notice that the errors in the Runge case for `n=1` and `n=2` are the same, as are the errors for `n=3` and `n=4`, as well as `n=5` and `n=6`. Explain why this should occur.

(i) Fill in the following table for the partly quadratic function.

partly_quadratic	
n	relative error
1	_____
2	_____
3	_____
4	_____
5	_____
6	_____
10	_____
20	_____
30	_____
40	_____

(j) Fill in the following table for the sawshape9 function.

sawshape9	
n	relative error
1	_____
2	_____
3	_____
4	_____

```
 5    ---------------
 6    ---------------
10    ---------------
20    ---------------
30    ---------------
40    ---------------
```

You should notice that much smaller errors are associated with the smooth Runge example function than with the non-differentiable partly quadratic function and with the discontinuous sawshape9 function.

You should see that the errors do not seem to be decreasing much for the final values of n, and wonder why the method becomes poor as n gets large. It may not be obvious, but the matrices (**??**) and (**??**) are related to the Hilbert matrix and are extremely difficult to invert. The reason inversion is difficult is because the monomials all start to look the same as n gets larger, that is, they become almost parallel in the L^2 sense. Even if the integration could be performed without error, you would observe roundoff errors in evaluating the resulting high-order polynomial.

One way to make the approximation problem easier might be to pick a better set of functions than monomials. The following section discusses a good alternative choice of polynomials. These polynomials allow much larger values of n.

3 Legendre Polynomials

The Legendre polynomials form an $L^2([-1,1])$-orthogonal set of polynomials. You will see below why orthogonal polynomials make particularly good choices for approximation. In this section, we are going to write m-files to generate the Legendre polynomials and we are going to confirm that they form an orthogonal set in $L^2([-1,1])$. Throughout this section, we will be representing polynomials as vectors of coefficients, in the usual way in Matlab.

The Legendre polyonomials are a basis for the set of all polynomials, just as the usual monomial powers of x are. They are appropriate for use on the interval [-1,1] because they are orthogonal when considered as members of $L^2([-1,1])$. Polynomials that are orthogonal are discussed by Quarteroni, Sacco, and Saleri in Chapter 10, with Legendre polynomials discussed in Section 10.1.2. The first few Legendre polynomials are:

$$\begin{aligned} P_0 &= 1 \\ P_1 &= x \\ P_2 &= (3x^2-1)/2 \\ P_3 &= (5x^3-3x)/2 \\ P_4 &= (35x^4-30x^2+3)/8 \end{aligned} \tag{7}$$

The value at x of any Legendre polynomial P_i can be determined using the following recursion:

$$\begin{aligned} P_0 &= 1, \\ P_1 &= x, \qquad \text{and,} \\ P_k &= ((2k-1)xP_{k-1} - (k-1)P_{k-2})/k \end{aligned}$$

The following recursive Matlab function computes the values of the k^{th} Legendre polynomial.

```
function yval = recursive_legendre ( k , xval )
% yval = recursive_legendre ( k , xval )
% yval = values of the k-th Legendre polynomial
% at values xval

if k<0
```

```
  error('recursive_legendre: k must be nonnegative.');
elseif k==0    % WARNING: no space between else and if!
  yval = ones(size(xval));
elseif k==1    % WARNING: no space between else and if!
  yval = xval;
else
  yval = ((2*k-1)*xval.*recursive_legendre(k-1,xval) ...
          - (k-1)*recursive_legendre(k-2,xval) )/k;
end
```

Unfortunately, this recursive function is too slow to be used in this lab. The alternative to recursive calculation of Legendre polynomials is one that uses loops. It is a general fact that any recursive algorithm can be implemented using a loop. The code for the loop is typically more complicated than the recursive formulation. In the following exercise, you will write an algorithm using loops for Legendre polynomials.

Exercise 2: Write a function m-file `legen.m` first to find the values of the n^{th} Legendre polynomial P_n using a loop. The strategy will be to first compute the values of P_0 and P_1 from their formulæ, then compute the values of P_k for larger subscripts by building up from lower subscripts, stopping at P_n. You should note is that if k is larger than 2, you only need to retain the values of P_{k-1} and P_{k-2} in order to compute the values of P_k.

(a) Use the signature

```
function yval = legen ( n , xval)
% yval = legen ( n , xval)
% more comments

% your name and the date
```

and add appropriate comments.

(b) Use `if` tests to define the cases `n<0`, `n==0` and `n==1`, and use the formulæ for these cases to compute the vector of values `yval`.

(c) When `n` is larger than 1, compute the vector of values of P_0 and call it `ykm1` (`ykm1` for "**y** sub **k** **m**inus **1**"), and compute the vector of values of P_1 and call it `yk`.

(d) Write a loop `for k=2:n` in which you first put the value of `ykm1` into `ykm2` ("**y** sub **k** **m**inus **2**") and then the value of `yk` into `ykm1`. You do this because you are changing the value of k to be one larger. Then compute P_k, calling it `yk`, using the values `ykm1` and `ykm2`. This line will be similar to the corresponding line in `recursive_legendre`.

(e) When the loop is complete, `k` has the value `n`. Set `yval=yk;`

(f) Test and verify your `legen` function for P_3 with the following code.

```
xval=linspace(0,1,10);
norm( legen(3,xval)-(5*xval.^3-3*xval)/2 )
```

and showing that the difference is of roundoff size.

(g) Your `legen` function and `recursive_legendre` should agree. Test this agreement for n=10 with the following code.

```
xval=linspace(0,1,20);
norm( legen(10,xval) - recursive_legendre(10,xval) )
```

The difference should be of roundoff size.

4 Orthogonality and Integration

The Legendre polynomials form a basis for the linear space of polynomials. One good characteristic of any set of basis vectors is to be orthogonal. For functions, we use the standard L^2 *dot product*, and say that two functions $f(x)$ and $g(x)$ are orthogonal if their dot product

$$(f, g) = \int_{-1}^{1} f(x)g(x)dx$$

is equal to zero. In Matlab, you could use `integral` or `quadgk` via the abbreviation `ntgr8` to compute this quantity in the following way:

```
q=ntgr8(@(x) ffunc(x).*gfunc(x) );
```

where `ffunc` gives the function f and `gfunc` gives g.

5 Legendre polynomial approximation

Legendre polynomial approximation in $L^2([-1, 1])$ follows the same recipe as monomial approximation:

1. Compute the matrix $H_{m,n} = \int_{-1}^{1} P_{m-1}(x)P_{n-1}(x)dx$. This matrix is diagonal (as opposed to the Hilbert matrix in the monomial case), with diagonal entries $H_{m,m} = 2/(2m-1)$, so integration is not necessary!
2. Compute the right side values $b_m = \int_{-1}^{1} f(x)P_{m-1}(x)dx$.
3. Solve $\mathbf{d} = H^{-1}\mathbf{b}$ using the formula $d_m = \frac{2m-1}{2}b_m$.
4. The approximation can be evaluated as

$$f(x) \approx f_{\text{legen}}(x) = \sum_{k=1}^{n} d_k P_{k-1}(x). \tag{8}$$

The coefficients d_k are not the same as the monomial coefficients c_k computed earlier, and Equation (**??**) must be used rather than `polyval` to evaluate the resulting approximations.

Exercise 3:

(a) Write a function m-file named `coef_legen.m` with signature

```
function d=coef_legen(func,n)
% d=coef_legen(func,n)
% comments

% your name and the date
```

to compute the coefficients of the approximation as $d_k = \frac{2k-1}{2}\int_{-1}^{1} f(x)P_{k-1}(x)dx$. You should use the `ntgr8` abbreviation function in the same way as used in Exercise **??** **Note:** The factor $(2k-1)/2$ comes from the inverse of the diagonal matrix $H_{k,k} = 2/(2k-1)$.

(b) Verify that `coef_legen` is correct by computing the best Legendre approximation to the Legendre function P_3, where $n \geq 4$. (Recall that n is the number of terms, not the degree of the polynomial.) The values you get for d are the coefficients in Equation (**??**), not the coefficients of the polynomial.

(c) Write a function m-file called `eval_legen.m` to be used to evaluate Legendre polynomials. It should evaluate Equation (**??**) and have the signature

```
function yval=eval_legen(d,xval)
% yval=eval_legen(d,xval)
% comments

% your name and the date
```

(d) Verify that `eval_legen` is correct by choosing `d` as the coefficients of P_3 (you computed `d` for this case above) and comparing the results of `eval_legen` and `legen` at the five values `[0,1,2,3,4]`.

(e) Write an m-file called `test_legen.m`, similar to the `test_mon.m` file you wrote above. It should use `eval_legen` and produce the relative error of the approximation. It is instructive if you plot the approximation as well, but you do not need to send me the plots.

(f) Place the Matlab command `tic;` at the beginning of the script and the Matlab command `toc;` at the end. This pair of commands will measure the elapsed time taken and print it.

(g) Fill in the following table for the Runge example function.

	Runge
n	relative error
1	_____
2	_____
3	_____
4	_____
5	_____
6	_____
10	_____
20	_____
30	_____
40	_____
50	_____

(h) Fill in the following table for the partly quadratic function.

	partly_quadratic	
n	relative error	elapsed time
5	_____	
10	_____	
20	_____	
40	_____	
80	_____	
160	_____	_____
320	_____	_____
640	_____	_____

(i) Fill in the following table for the sawshape9 function.

	sawshape9	
n	relative error	elapsed time
5	_____	
10	_____	
20	_____	
40	_____	
80	_____	
160	_____	_____
320	_____	_____
640	_____	_____

(j) Based on the above data, roughly estimate the value of p where elapsed time is proportional to n^p. Is $1 \le p \le 2$? $2 \le p \le 3$? $3 \le p \le 4$? $4 \le p \le 5$? $5 \le p \le 6$?

You should find the same values as for approximation by monomials for small `n`, and you can accurately compute with larger values of `n` using Legendre polynomials than using monomials. However, using large values of `n` can result in computing times that grow more rapidly than expected as `n` increases because the `integral` function must compensate for roundoff error arising from rapid oscillations. In fact, the time does grow more rapidly than $O(n^p)$, where you just estimated p.

6 Fourier series

There is another set of functions that is orthogonormal in the $L^2[-1,1]$ sense. This is the set of trigonometric functions

$$\frac{1}{\sqrt{2}},\ \cos(\pi x),\ \sin(\pi x),\ \cos(2\pi x),\ \sin(2\pi x),\ \cos(3\pi x),\ \ldots$$

and they can be used for approximating functions. We have seen trigonometric polynomials before in the context of interpolation using $e^{ik\pi x}$ for $k = -n, -n+1, \ldots, -1, 0, 1, \ldots, n-1, n$. Using complex exponentials is equivalent to sin and cos but the trigonometric functions are orthogonormal on $[-1,1]$ while the complex exponentials are not.

The sum of the first $2n+1$ terms of the Fourier series for a function f is given as

$$f(x) \approx \frac{z}{\sqrt{2}} + \sum_{k=1}^{n} s_k \sin k\pi x + c_k \cos k\pi x. \tag{9}$$

As usual, the coefficients can be found by multiplying both sides by $(1/\sqrt{2})$, $\sin(\ell\pi x)$, or $\cos(\ell\pi x)$ and integrating. Orthonormality leads to the expressions

$$\begin{aligned} z &= \int_{-1}^{1} \frac{f(x)}{\sqrt{2}} dx \\ s_k &= \int_{-1}^{1} f(x)\sin(k\pi x) dx \\ c_k &= \int_{-1}^{1} f(x)\cos(k\pi x) dx \end{aligned} \tag{10}$$

(terms involving $k \neq \ell$ are zero).

Exercise 4:

(a) Write a function m-file named `coef_fourier.m` that is similar to `coef_legen` and has signature

```
function [z,s,c]=coef_fourier(func,n)
% [z,s,c]=coef_fourier(func,n)
% more comments

% your name and the date
```

to compute the first $2n+1$ coefficients of the Fourier series using Equation (**??**).

(b) Test your coefficient function by using $f(x) = 1/\sqrt{2}$, $f(x) = \sin(2\pi x)$ and $f(x) = \cos(3\pi x)$, with $n \ge 3$. Of course, you should get $z = 1$, and all others zero in the first case, $s_2 = 1$ and all others zero in the second case, and $c_3 = 1$ with all others zero in the third case.
Warning: The `integral` function requires that its integrand be a function that returns a vector when its argument (`x`) is a vector. Be sure that you use such a function in the $f(x) = 1/\sqrt{2}$ case! You can check your function by computing $\int_{-1}^{1} (1/\sqrt{2})^2\, dx$ and checking that it equals 1.

(c) Write a function m-file called `eval_fourier.m` to evaluate Equation (**??**) and have the signature

```
function yval=eval_fourier(z,s,c,xval)
% yval=eval_fourier(z,s,c,xval)
% more comments

% your name and the date
```

(d) Test `eval_fourier.m` using the same three functions you used above: $f(x) = 1/\sqrt{2}$, $f(x) = \sin(2\pi x)$ and $f(x) = \cos(3\pi x)$. In each case, use `eval_fourier.m` with the appropriate choices of coefficients `z`, `s` and `c`, and compare the approximate values at a selection of values against the true values. Describe the values you chose and the results you obtained.

(e) Write an m-file called `test_fourier.m`, similar to the `test_mon.m` and `test_legen.m` file you wrote above. It should use `eval_fourier` and produce the relative error of the approximation. It is instructive if you plot the approximation as well, but you do not need to send me the plots.

(f) Fill in the following table for the Runge example function.

Runge		
n	relative error	elapsed time
1	_____	
2	_____	
3	_____	
4	_____	
5	_____	
6	_____	
10	_____	
50	_____	
100	_____	_____
200	_____	_____
400	_____	_____
800	_____	_____

(g) Fill in the following table for the partly quadratic function.

partly_quadratic		
n	relative error	elapsed time
1	_____	
2	_____	
3	_____	
4	_____	
5	_____	
6	_____	
10	_____	
50	_____	
100	_____	_____
200	_____	_____
400	_____	_____
800	_____	_____

(h) When you used trigonometric polynomial interpolation in Lab 6, you looked at the error for a sawshape function and saw the Gibb's phenomenon, which kept the error from going to zero. You have seen good performance of Fourier approximation on differentiable and continuous functions above. A discontinuous function exhibits the Gibb's phenomonon, but when convergence is measured using an integral norm it doesn't prevent convergence (although it slows it down). Fill in the following table for the `sawshape9` function.

	sawshape9	
n	relative error	elapsed time
1	_____	
2	_____	
3	_____	
4	_____	
5	_____	
6	_____	
10	_____	
50	_____	
100	_____	_____
200	_____	_____
400	_____	_____
800	_____	_____

(i) Based on the above data, roughly estimate the value of p where elapsed time is proportional to n^p. Is $1 \le p \le 2$? $2 \le p \le 3$? $3 \le p \le 4$?

You should be convinced that these series do not converge very rapidly, with execution times becoming much too large to acheive high accuracy. This increase in execution time is due to the adaptive quadrature used by `integral` requiring progressively more points. It turns out that increasing n beyond about 2000 results in failure of `integral` to achieve the desired accuracy. Thus, the sawshape9 and partly quadratic functions cannot be integrated to the accuracy that the Runge example function can achieve.

7 Piecewise constant approximation

We have learned that approximation is best done using matrices that are easy to invert accurately, like diagonal matrices. This is the reason for using sets of orthogonal basis functions. We would also like to be able to perform the right side integrals easily as well. A large part of the reason that the orders of approximations in the exercises above have been restricted is that the integrals are difficult to perform accurately because they "wiggle" a lot, a major source of inaccuracy in the approximation. Using the `integral` function hides the inaccuracy, but you pay for it because the integrations require substantial time for higher values of `n`, as you may have noticed.

In this section, we will look at approximation by piecewise constant functions. Approximation by piecewise linears or higher are also useful, but all the important steps are covered with piecewise constants. Furthermore, piecewise constants are easy to extend to higher dimensions.

Suppose that a number $N_{\rm pc}$ is given and that the interval $[-1,1]$ is divided into $N_{\rm pc}$ equal subintervals and $N_{\rm pc}+1$ points x_k, $k = 1, 2, \ldots, N_{\rm pc}+1$. For $k = 1, \ldots, N_{\rm pc}$, a function $u_k(x)$ can be defined as

$$u_k(x) = \begin{cases} 1 & x_k \le x < x_{k+1} \\ 0 & x < x_k \quad \text{or} \quad x > x_{k+1} \end{cases}$$

These functions clearly satisfy

$$\int_{-1}^{1} u_k(x) u_\ell(x) dx = \begin{cases} 2/N_{\rm pc} & k = \ell \\ 0 & k \ne \ell \end{cases} \tag{11}$$

This orthogonality immediately implies linear independence. In addition, any function in $L^2([-1,1])$ can be approximated as a sum of them (this is a deep theorem). As it turns out, these theoretical facts are not compromised by numerical difficulties and for reasonable values of n can be used for numerical approximation.

If a vector of coefficients **a** can be found to represent the piecewise constant approximation to a function $f(x)$, then the approximation can be evaluated as

$$f(x) \approx f_{\rm pc}(x) = \sum_{j=1}^{N_{\rm pc}} a_j u_j(x) = a_k \tag{12}$$

where k is the index satisfying $x_k \le x < x_{k+1}$.

In the following exercise, we will follow the same basic recipe as before to compute the coefficients **a** and the approximation to $f(x)$.

Exercise 5: In this exercise you will be working with these piecewise constant (pc) functions. You may assume that $N_{\rm pc}$ is even so that $x_k < 0$ for $k \le N_{\rm pc}/2$, that $x_k > 0$ for $k > N_{\rm pc}/2+1$ and $x_k = 0$ for $k = N_{\rm pc}/2+1$.

(a) Write a function m-file named `coef_pc.m` with signature

```
function a=coef_pc(func,Npc)
% a=coef_pc(func,Npc)
% comments

% your name and the date
```

to compute the coefficients of the approximation as

$$\begin{aligned} a_k &= \frac{N_{\rm pc}}{2}\int_{-1}^{1} f(x)u_k(x)dx \\ &= \frac{N_{\rm pc}}{2}\int_{x_k}^{x_{k+1}} f(x)dx \end{aligned}$$

Use `integral` (or, if you are using an older version of Matlab, `quadgk`), not `ntgr8` to compute these integrals, because the interval of integration is not `[-1,1]`. To write this function, you will need to use `linspace` to generate the points x_k. Be careful not to confuse the number of points with the number of intervals!

(b) Test your `coef_pc` on the function that is equal to one for all values of x. In Matlab, this can be done with `y=ones(size(x))`. Use `Npc=10`. Of course, all $a_k = 1$.

(c) Test `coef_pc` with `Npc=10` on the function $f(x) = x$. You should get

$$\begin{aligned} a_k &= \frac{N_{\rm pc}}{2}\int_{x_k}^{x_{k+1}} x dx \\ &= \frac{N_{\rm pc}}{4}(x_{k+1}^2 - x_k^2) \\ &= \frac{2k}{N_{\rm pc}} - 1 - \frac{1}{N_{\rm pc}} \end{aligned}$$

(d) We have already used a function called `bracket.m` that determines the values of k for which $x_k \le x < x_{k+1}$. You may use the one you downloaded earlier or download `bracket.m` again from the web site. Use `bracket` to write a function m-file called `eval_pc.m` to evaluate the piecewise constant approximation to f using Equation (**??**) and has the signature

```
function yval=eval_pc(a,xval)
% yval=eval_pc(a,xval)
% comments

% your name and the date
```

As in `coef_pc`, you will need to use `linspace` to generate the points x_k, and `bracket` to find the values of k corresponding to the values of `xval`.

(e) Generate the coefficients `a` using the function $f(x) = x$ and `Npc=10` that you used above. Test `eval_pc` using

```
xval=[-0.95,-0.65,-0.45,-0.25,-0.05,0.15,0.35,0.55,0.75,0.95]
```

Be sure your answers are correct before continuing.

(f) Write an m-file called `test_pc.m` similar to `test_mon.m` and `test_legen.m` above. You should use vector (componentwise) statements whenever possible or the calculations might take a long time. `test_pc.m` should:

i. Confirm that `Npc` is an even number (and call `error` if not),

ii. Evaluate the coefficients (`a`) of the approximation using `coef_pc.m`,

iii. Use `eval_pc.m` to evaluate the approximation and then compare the approximation against the exact solution, **Because we will be using large values of `Npc`, choose at least 20000 test points.**

iv. Use `tic` and `toc` to measure the time taken by computing the coefficients, computing the approximate and exact solutions, and computing the error.

It will be valuable to plot the approximation because it will help you debug your work and it will illustrate the process.

Look carefully and critically at the plot for the Runge function with `Npc=8`. You should be able to justify to yourself visually that no other piecewise constant function would produce a better approximation. Send me the plot for the Runge function with `Npc=8`.

(g) Fill in the following table for the Runge example function:

	Runge example	
Npc	relative error	elapsed time
4	_____	
8	_____	
16	_____	
64	_____	
256	_____	
1024	_____	_____
4096	_____	_____
16384	_____	_____

(h) Fill in the following table for the partly quadratic function:

	partly quadratic	
Npc	relative error	elapsed time
4	_____	
8	_____	
16	_____	
64	_____	
256	_____	
1024	_____	_____
4096	_____	_____
16384	_____	_____

(i) Fill in the following table for the sawshape9 function:

	sawshape9	
Npc	relative error	elapsed time
4	_____	
8	_____	
16	_____	
64	_____	
256	_____	
1024	_____	_____
4096	_____	_____
16384	_____	_____

(j) Based on the above data, roughly estimate the integer p where **relative error** is proportional to $(1/n)^p$.

(k) Based on the above data, roughly estimate the value of p where **elapsed time** is proportional to n^p.

This approximation may take a while to compute, but it does not deteriorate as `Npc` gets large! In fact, you should observe linear convergence. Further, the time required does not grow very quickly. For any of these three functions, accuracy higher than 10^{-10} can be achieved.

Remark: As you might imagine, approximation using piecewise linear functions will converge more rapidly than using piecewise constants. There are alternative approaches for using piecewise linears: piecewise linear functions on each interval with jumps at interval endpoints, as the piecewise constant functions have; and, piecewise linear functions that are *continuous* throughout whole interval. The first retains orthogonality and the diagonal form of the coefficient matrix H. The second sacrifices the diagonal form for a banded form that is almost as easy to solve, as you may see in the next exercise. Continuity, however, can be worth the sacrifice, depending on the application. Even higher order piecewise polynomial approximation is possible, if the application can benefit.

Remark: In Exercises 3, 4, and 5 you were asked to estimate the growth of the running time with n. You found in Exercises 3 and 4 that the running time increased more rapdily than linearly ("superlinear") as n increases, but in Exercise 5 it increased linearly. The distinction between superlinear and linear is important when choosing an algorithm to use. When running time increases too fast, algorithms can become too time-consuming to be useful. Since it is clear that approximation algorithms must scale at least linearly in n, Exercise 5 shows that piecewise constant approximation can be an attractive choice. In the extra credit problem below, you will find a similar result for piecewise linear approximation.

8 Extra credit: Piecewise linear approximation (8 points)

Piecewise linear approximations improve the rate of convergence over piecewise constant approximations, at the cost of increased work. In addition, piecewise linear approximations are commonly used in finite element approximations to differential equations. In this extra credit exercise, you will see how the same approach you saw above can be extended to piecewise linear approximations.

For this presentation, you again break the interval into equal subintervals. Denote the number of intervals by $N_{\rm pl}$, although it is the same as $N_{\rm pc}$ above. Thus, there are $N_{\rm pl}+1$ points defined over the interval $[-1,1]$ according to the Matlab function `x=linspace(-1,1,Npl+1)`. For each $k = 1, 2, \ldots, N_{\rm pl}+1$, define a set of "hat" functions t_k as

$$t_k(x) = \begin{cases} (x_{k+1}-x)/(x_{k+1}-x_k) & x_k \le x \le x_{k+1} \text{ and } k \le N_{\rm pl} \\ (x-x_{k-1})/(x_k-x_{k-1}) & x_{k-1} \le x \le x_k \text{ and } k \ge 2 \\ 0 & \text{otherwise} \end{cases}$$

so that t_k is a continuous piecewise linear function that is 1 at x_k and zero at all the other points x_ℓ for $\ell \neq k$. It is possible to show that these functions are linearly independent when considered as members of

the Hilbert space $L^2([-1,1])$. Further, an equation analogous to (**??**) is

$$\int_{-1}^{1} t_k(x)t_\ell(x)dx = \begin{cases} 4/(3N_{\rm pl}) & 2 \le k = \ell \le N_{\rm pl} \\ 2/(3N_{\rm pl}) & k = \ell = 1 \text{ or } k = \ell = N_{\rm pl}+1 \\ 1/(3N_{\rm pl}) & |k-\ell| = 1 \text{ (\textit{i.e.} } k = \ell \pm 1) \\ 0 & |k-\ell| > 1 \end{cases} \tag{13}$$

There are $N_{pl}+1$ functions $t_k(x)$!

There is an equation analogous to (**??**), (**??**), (**??**), and (**??**):

$$f(x) \approx f_{\rm pl}(x) = \sum_{j=1}^{N_{\rm pl}+1} a_j t_j(x) \tag{14}$$

Exercise 6:

(a) To see why the functions are called "hat" functions, plot the function $t_3(x)$ when $N_{\rm pl} = 4$. If you do this using Matlab, be sure that the points you use to plot t_3 includes the points x_k or your plot will not look right. You might find that forcing space around the plot makes it look more like a hat. You could use `axis([-1.1, 1.1, -0.1, 1.1])` .

(b) Write a function `coef_plin` to compute the coefficients a_j in (**??**). The system of equations you will need to generate and solve are analogous to (**??**) and can be constructed by replacing j with ℓ in (**??**), multiplying by $t_k(x)$ and integrating. Do not forget to construct a matrix analogous to $H_{k\ell}$. Devise a simple test meant for debugging and test `coef_plin`.
Hint: The functions $t_k(x)$ have limited support, especially for large $N_{\rm pl}$. Adjust the limits of integration to reflect the support in order to save time (and, it turns out) improve accuracy. You will need to use the `integral` or `quadgk` commands directly instead of through `ntgr8` since `ntgr8` assumes an integration interval of `[-1,1]`.

(c) Write a function `eval_plin`. Devise a test (I suggest one using $N_{\rm pl} = 2$ or 3) and test `eval_plin`.

(d) Write a function `test_plin`, including timing, and apply it to the three functions we have been using: `runge` `partly_quadratic` and `sawshape9`. Use values of `Npl`=[4,8,16,64,256,1024]. In each case, estimate the rate of convergence and rate of increased time as `Npl` varies.
Note: You should observe improved convergence rate for the two continuous functions and a poorer rate for the discontinuous `sawshape9`.

Last change `$Date: 2016/10/31 00:23:00 $`