

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

Introduction	Exercise 1
Trapezoid integration over $[-1,1]$	Exercise 2
Legendre Polynomials	Exercise 3
Orthogonality and Integration	Exercise 4
Least squares approximations in $L^2([-1, 1])$	Exercise 5
Legendre polynomial approximation	Exercise 6
Fourier series	Exercise 7
Piecewise approximation	Exercise 8
	Exercise 9

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 recipe, a set of 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.

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.

This kind of approximation requires evaluation of integrals. In most applications this evaluation is done using numerical integration, but we have not yet discussed numerical integration in general. On the other hand, you have seen the trapezoid rule for numerical integration in lecture, and we will be using a specialized application of this method.

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 is inaccurate because of roundoff errors. L^2 approximation is improved but still not of high quality. Fourier approximation substantially reduces the roundoff errors, but is slow to compute and evaluate and still is subject to error for higher terms. Approximation by piecewise constants is not subject to error until ridiculously large numbers of pieces are employed.

Matlab has excellent symbolic capabilities, and it would be my preference to use the symbolic toolbox to do the integrals in this lab. Unfortunately, the symbolic toolbox does not work on the computers in GSCC, so we will not be using it.

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

- The Runge function, $f(x) = 1/(1 + x^2)$
- The function

$$f(x) = \begin{cases} 0 & -1 \leq x < 0 \\ 4x(1-x) & 0 \leq x \leq 1 \end{cases} \quad (1)$$

For the purpose of this lab, this function will be called “partly quadratic.” It was chosen because it is simple and continuous, but is not differentiable. A simple Matlab function m-file to compute this “partly quadratic” function can be found by downloading `partly_quadratic.m` from the web site or by copying the following code:

```
function y=partly_quadratic(x)
% y=partly_quadratic(x)
% input x (possibly a vector or matrix)
% output y, where
%   y=0 for x<=0
%   y=4*x*(1-x) for x>0

% your name and the date

% The Matlab expression for "true" evaluates to 1 and
% "false" evaluates to 0. The following trick sets
% y=0 when x<0 and works when x is a vector
y=4*(x>0).*x.*(1-x);
```

- A third function is a sawtooth function similar to one you saw in Lab 6:

$$f(x) = \begin{cases} (x+1) & -1 \leq x < 0 \\ (x-1) & 0 \leq x \leq 1 \end{cases} \quad (2)$$

A simple Matlab function m-file to compute this sawtooth function can be found by downloading `sawtooth9.m` from the web site or by copying the following code:

```
function y=sawtooth9(x)
% y=sawtooth9(x)
% input x (possibly a vector or matrix)
% output y, where
%   y=(x+1) for -1<x<0
%   y=(x-1) for 0<=x<1

% The Matlab expression for "true" evaluates to 1 and
% "false" evaluates to 0.
y=(x+1)-2*(x>=0);
```

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

2 Trapezoid integration

You have seen the formula for integration using the trapezoid rule. Using N *uniform* subintervals of $[a, b]$,

$$\int_{-1}^1 f(x)dx \approx \frac{b-a}{N-1} \left(\frac{1}{2}f_1 + \sum_{k=2}^{N-1} f_k + \frac{1}{2}f_N \right), \quad (3)$$

where $f_k = f(x_k)$ and $x_k, k = 1, 2, \dots, N$ represent N equally-spaced points over the interval $[a, b]$ (including the endpoints).

For the purpose of this lab, we will use Formula (3) to compute the integral of a function represented as a vector of values f_k for $k = 1, 2, \dots, N$. The Matlab function will be called `trapsum` (**t**rapezoid **i**ntegration **s**um).

as a **sum**). Because you will be using `trapsum` to perform integrals in most of the exercises below, it is essential to be sure it is well-tested. Also, you will be using it with large values of N , so use vector syntax and the Matlab `sum` function.

Exercise 1:

- (a) Write a function m-file with signature

```
function I=trapsum(fValues,a,b)
% comments
```

```
% your name and the date
```

that implements Equation (3) with the integrand given by a vector of values `fValues`.

- (b) Test your function by integrating the Runge function over the interval $[-1, 1]$. Fill in the following table, recalling that the exact integral is $2*\text{atan}(1)$.

N	Integral	Error	Err(N)/Err(2*N)
1000	-----	-----	-----
2000	-----	-----	-----
4000	-----	-----	-----
8000	-----	-----	-----

- (c) You know that the trapezoid rule is $O(h^2)$, with $h = 2/(N - 1)$, for differentiable functions. Does your result confirm this theoretical rate? If not, go back and correct your work.

Remark: Getting both the correct answer and also the theoretical convergence rate is a pretty sure sign that your code is correct. The correct answer alone is often not good enough to eliminate all bugs. The only further test I might recommend is to check that the time the code takes scales correctly with size, but this kind of testing is beyond the scope of this lab.

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 polynomials 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 $L^2([-1,1])$. They are discussed in Atkinson starting on page 210, and the first few Legendre polynomials are:

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

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

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

The Matlab language supports recursive function calls, so a Matlab m-file can be written in a way that mimics this recursive definition.

Exercise 2: Copy the following code to a function m-file named `rlegendre.m` or download `rlegendre.m`. This function evaluates the coefficient vector for the k^{th} Legendre polynomial P_k .

```
function c = rlegendre ( k )
% comments

if k==0
    c = 1;
elseif k==1    % WARNING: no space between else and if!
    c = [1 0];
else
    % look at this line carefully!
    c = ((2*k-1)*[rlegendre(k-1),0] - (k-1)*[0,0,rlegendre(k-2)])/k;
end
```

Warning: Some students trying to use the above code on PCs running MS-Windows, not Linux, have observed problems with the recursive statement above. If you observe these problems, use the following statements instead:

```
ca = (2*k-1)*[rlegendre(k-1),0];
cb = (k-1)*[0,0,rlegendre(k-2)];
c = (ca-cb)/k;
```

- (a) Add appropriate comments to the m-file.
- (b) What does the line following the comment “look at this line carefully” do for $k=2$?
 - i. What is `[rlegendre(k-1),0]`?
 - ii. Why is the “,0” there?
 - iii. What is `[0,0,rlegendre(k-2)]`?
 - iv. Why is the “[0,0,” there?
- (c) Use `rlegendre` to confirm the first five polynomials presented above in (4).
- (d) Finally, I would like to illustrate that recursive evaluation can be very time consuming. To see how long an instruction or group of instructions takes, Matlab provides two timing routines: `tic` and `toc`. `tic` starts the timer, `toc` prints out the elapsed time. Compute `rlegendre(25)`, and time the computation. The sequence of commands is

```
tic; rlegendre( 25 ); toc
```

(Putting all three commands on a single line guarantees that typing time is not what is being timed.) It should take less than 15 seconds for this computation on the black Dell computers in GSCC, but it could take much longer on a slower computer.

You can see in the previous exercise that recursive evaluation, although short and “elegant,” can require significant computing time. Further, the amount of time (in this example, but not in all cases) increases rapidly with size. Do *not* try to wait for `rlegendre(35)` to complete! The reason it takes so long is that each call to `rlegendre` generates two more calls, albeit with smaller values of k . Roughly speaking, then, a call to `rlegendre` with $i=25$ generates a total of 2^{24} calls, most of which are with repeated values of k . It turns out that this is a “worst-case” algorithm for recursion, and not all recursive algorithms are nearly this bad.

One does not see very many recursive functions in scientific work, but you can use them when they can easily be written directly from the mathematical definition of the function. The trade-off is between ease of writing (and debugging) and potential inefficiency. In the next lab, you will see recursion used for an adaptive quadrature algorithm that would be difficult to write without recursion.

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. In the following exercise, you will write a more efficient algorithm for Legendre polynomials.

Exercise 3: Write a function m-file `legendre.m` to evaluate the k^{th} Legendre polynomial P_k using a loop. The strategy will be to first compute P_0 and P_1 from their formulæ, then compute P_n for larger subscripts by building up from lower subscripts, stopping at P_k . You should note is that if n is larger than 2, you only need to retain the values P_{n-1} and P_{n-2} in order to compute P_n .

- (a) Use the signature

```
function c = legendre ( k )
```

and add appropriate comments.

- (b) Use `if` tests to define the cases `k==0` and `k==1`, and use the formulæ to compute `c`.
 (c) When `k` is larger than 1, compute the coefficient vector of P_0 and call it `cpnm1` (“`c` for **P** sub **n** minus 1”), and compute the coefficient vector of P_1 and call it `c`.
 (d) Write a loop for `n=2:k` in which you first put the value of `cpnm1` into `cpnm2` (“`c` for **P** sub **n** minus 2”) and then the value of `c` into `cpnm1`. You do this because you are changing the value of `n` to be one larger. Then compute the coefficient vector of P_n , calling it `c`, using the values `cpnm1` and `cpnm2`. This line will be similar to the corresponding line in `rlegendre`.
 (e) Test and verify your `legendre` function by reproducing the first five Legendre polynomials presented above in (4).
 (f) Finally, use `tic` and `toc` to time your routine by computing `legendre(25)` and compare it with the time required for `rlegendre(25)`. Loops are faster, aren't they?

4 Orthogonality and Integration

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

$$(p, q) = \int_{-1}^1 p(x)q(x)dx$$

is equal to zero.

You have computed the coefficients of the Legendre polynomials. In order to integrate them, you need to know the values of the polynomials. Recall that the `polyval(c, xval)` function will find the values of the polynomial with coefficient vector `c` at the values `xval`.

In the following exercise we will verify that the Legendre polynomials form an orthogonal set in $L^2([-1, 1])$.

Exercise 4:

- (a) Use `trapsum` to perform the integrals

$$\int_{-1}^1 P_n(x)P_n(x)dx \approx \frac{2}{2n+1}$$

for $n=0, 1, 2, 3, 4$, where P_n denotes a Legendre polynomial, using 25,000 points (`xval=linspace(-1, 1, 25000)`)

Hint: the integral of a product of Legendre polynomials can be found as

`I=trapsum(polyval(legendre(n),xval).*polyval(legendre(m),xval),-1,1);`

(b) Verify that

$$\int_{-1}^1 P_n(x)P_2(x)dx \approx 0$$

for $n=0,1,3,4$.

(c) To see the effect of roundoff error, show that

$$\int_{-1}^1 P_n(x)P_2(x)dx$$

is *not* approximately zero when $n=50$. Oscillations in P_{50} are the source of the error.

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

Atkinson, in Section 4.3, discusses 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 $\{x^{k-1} : k = 1, 2, \dots, n\}$), for each vector of numbers $\mathbf{c} = (c_k)$ define a functional

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

This continuous functional becomes large when $\|c\|$ 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

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

Atkinson evaluates this expression in general. For completeness, it is included here (using slightly different notation) for the case of quadratic approximations ($n = 3$). Consider the functional

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

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

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

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

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

or

$$\begin{aligned} 2\frac{c_1}{5} + 0 + 2\frac{c_3}{3} &= \int_{-1}^1 f x^2 dx \\ 0 + 2\frac{c_2}{3} + 0 &= \int_{-1}^1 f x dx \\ 2\frac{c_1}{3} + 0 + 2\frac{c_3}{1} &= \int_{-1}^1 f dx \end{aligned} \tag{5}$$

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

Equation (5) is related to Atkinson's Equation (4.3.14), which was derived for the monomials over the interval $[0, 1]$. For an arbitrary value of n , Equation (5) can be written in the following way, where I have modified the indexing to correspond 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 .)

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

The matrix in (6),

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

is closely related to the Hilbert matrix that Atkinson comes up with. Atkinson mentions that the Hilbert matrix is difficult to invert accurately. The following exercise illustrates this difficulty and its implication for approximation.

Examining Equation (6), 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 5: 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 (6) above. This m-file will have the signature

```
function c=approx_mon(f,n)
% comments
% f is the name of a function

% your name and the date
```

and be called `approx_mon.m` (for **approx**imation by **mon**omials. It will solve the system (6) by constructing the matrix H and right side \mathbf{b} and solving the resulting system for c_k . In `approx_mon.m`, \mathbf{f} refers to a function, not a vector.

(a) Begin `approx_mon.m` with the following code

```
function c=approx_mon(f,n)
% f is the name of a function
% comments
```

```
NUM_INTEGRATION_PTS=25000;
xval=linspace(-1,1,NUM_INTEGRATION_PTS);
```

- (b) Use `trapsum` to evaluate the components of the matrix $H_{k,\ell} = \int_{-1}^1 x^{(2n-k-\ell)} dx$, and the right side of the equation, $\mathbf{b}_k = \int_{-1}^1 x^{n-k} f(x) dx$. **Hint:**

```
H(k,e11)=trapsum(xval.^(2*n-k-e11),-1,1)
```

(I try not to use the letter “1” as a variable because it looks so much like the number 1.)

- (c) Solve for the coefficients \mathbf{c} by computing the inverse of \mathbf{H} and multiplying by \mathbf{b} . (Normally you would use the backslash command for this, but I do not want any optimization to go on behind the scenes. Also, because \mathbf{H} is very poorly conditioned, Matlab may give warnings to that effect.)
- (d) 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.
- (e) Write a script m-file named `test_mon.m` containing code similar to the following

```
c=approx_mon('partly_quadratic',n);
```

```
xtest=linspace(-1,1,2000);
ytest=polyval(c,xtest);
yexact=partly_quadratic(xtest);
plot(xtest,ytest,xtest,yexact)
```

```
% relative Euclidean norm is identical to using trapsum to compute
% integral least-squares (L2 norm)
relativeError=norm(yexact-ytest)/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.

- (f) You do not need to send me copies of the plots, but fill in the following table using the partly quadratic function. You should find that the error gets smaller for early values of n and then deteriorates substantially. The smallest value of the relative error should be less than .025. At what value of n does the smallest relative error occur? (You may get warnings that the matrix \mathbf{H} is almost singular.)

partly_quadratic	
n	relative error
1	-----
2	-----
3	-----
4	-----
5	-----
10	-----
15	-----
20	-----
25	-----

30 -----
 40 -----

(g) Fill in the following table for the Runge function. For what value of n does the smallest relative error occur?

Runge	
n	relative error
1	-----
2	-----
3	-----
4	-----
5	-----
10	-----
15	-----
20	-----
25	-----
30	-----
40	-----

Why does this method fail as n gets large? It may not be obvious, but the matrices (5) and (6) 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. 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.

In the following section we will use the Legendre polynomials to compute polynomial approximations.

6 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 $H_{m,m} = 2/(2m - 1)$
2. Compute the right side values $b_m = \int_{-1}^1 f(x)P_{m-1}(x)dx$.
3. Solve $\mathbf{C} = H^{-1}\mathbf{b}$ using the formula $C_m = \frac{2m-1}{2}b_m$.
4. The approximation can be evaluated as

$$f(x) \approx f_{\text{leg}}(x) = \sum_{k=1}^n C_k P_{k-1}(x). \tag{7}$$

The coefficients C_k are not the same as the monomial coefficients c_k computed earlier, and Equation (7) must be used rather than `polyval` to evaluate the resulting approximations.

Exercise 6:

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

```
function C=approx_leg(f,n)
% comments

% your name and the date
```

to compute the coefficients of the approximation as $C_k = \frac{2k-1}{2} \int_{-1}^1 f(x)P_{k-1}(x)dx$.

- (b) Verify that `approx_leg` 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 C are the coefficients in Equation (7), not the coefficients of the polynomial.
- (c) Write a function m-file called `eval_leg.m` to use instead of `polyval`. It should evaluate Equation (7) and have the signature

```
function yval=eval_leg(C,xval)
% comments
```

```
% your name and the date
```

You will probably find it convenient to use `polyval` inside `eval_leg`.

- (d) Write an m-file called `test_leg.m`, similar to the `test_mon.m` file you wrote above. It should use `eval_leg` 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.
- (e) Fill in the following table for the Runge function.

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

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

partly_quadratic	
n	relative error
1	-----
2	-----
3	-----
4	-----
5	-----
10	-----
20	-----
30	-----
40	-----
50	-----

You should find the same values as for approximation by monomials for small n , but you can get larger values using Legendre polynomials than using monomials. However, without more care (in computing $\int f(x)P_{n-1}(x)dx$) Legendre polynomials eventually succumb to roundoff errors as seen for $n=40$ in the table.

7 Fourier series

There is a different set of functions that are orthogonal in the $L^2[-1, 1]$ sense. These are the trigonometric functions

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

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 $k = -n, -n+1, \dots, -1, 0, 1, \dots, n-1, n$. Using complex exponentials is equivalent to sin and cos but the trigonometric functions are orthogonal while the complex exponentials are not.

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

$$f(x) \approx \frac{z}{\sqrt{2}} + \sum_{k=1}^n s_k \sin k\pi x + c_k \cos k\pi x. \quad (8)$$

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 (with ℓ replaced by k)

$$\begin{aligned} z &= \int_{-1}^1 \frac{f(x)}{\sqrt{2}} dx \\ s_k &= \int_{-1}^1 f(x) \sin(k\pi x) dx \\ c_k &= \int_{-1}^1 f(x) \cos(k\pi x) dx \end{aligned} \quad (9)$$

Exercise 7: Confirm that some of the trigonometric functions are orthonormal in the $L^2[-1, 1]$ sense by using `trapsum` to integrate the following integrands using 25,000 points (`xval=linspace(-1,1,25000)`). Be careful in the case of $(1/\sqrt{2})^2$.

Integrand	value
<code>(1/sqrt(2))^2</code>	-----
<code>cos(pi*xval).^2</code>	-----
<code>sin(2*pi*xval).^2</code>	-----
<code>cos(3*pi*xval).^2</code>	-----
<code>cos(pi*xval).*sin(2*pi*xval)</code>	-----
<code>cos(3*pi*xval).*sin(2*pi*xval)</code>	-----

Exercise 8:

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

```
function [z,s,c]=approx_fourier(f,n)
% comments
```

```
% your name and the date
```

to compute the first $2n + 1$ coefficients of the Fourier series using Equation (9).

- (b) Test your coefficient function by using $f(x) = \sin(2\pi x)$ and $f(x) = \cos(3\pi x)$, with $n \geq 3$. Of course, you should get $s_2 = 1$ and all others zero in the first case, and $c_3 = 1$ with all others zero in the second case.
- (c) Write a function m-file called `eval_fourier.m` to evaluate Equation (8) and have the signature

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

- (d) Write an m-file called `test_fourier.m`, similar to the `test_mon.m` and `test_leg.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.
- (e) Fill in the following table for the Runge function.

Runge	
n	relative error
1	-----
2	-----
3	-----
4	-----
5	-----
10	-----
20	-----
30	-----
40	-----
100	-----
200	-----
400	-----

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

partly_quadratic	
n	relative error
1	-----
2	-----
3	-----
4	-----
5	-----
10	-----
20	-----
30	-----
40	-----
100	-----
200	-----
400	-----
1000	-----

- (g) When you used trigonometric polynomial interpolation in Lab 7, you looked at the error for a sawtooth 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 phenomenon, 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 `sawtooth9` function.

sawtooth9	
n	relative error
1	-----
2	-----
3	-----

4	-----
5	-----
10	-----
20	-----
30	-----
40	-----
100	-----
200	-----
400	-----

You should be convinced that these series do not converge very rapidly and convergence at all is dependent on close attention to computing the integrals.

8 Piecewise 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 to relatively small numbers is that the integrals are difficult to perform accurately because they “wiggle” a lot, a major source of inaccuracy in the approximation.

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

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

These functions clearly satisfy

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

This orthogonality immediately implies linear independence, and 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 \mathbf{a} can be found to represent the piecewise constant approximation to a function $f(x)$, then the approximation can be evaluated as

$$f(x) \approx f_{\text{pc}}(x) = \sum_{j=1}^{N_{\text{pc}}} a_j u_j(x) = a_k \tag{10}$$

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

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

Exercise 9: In this exercise you will be working with these piecewise constant (pc) functions. You may assume that N_{pc} is even so that $x_k < 0$ for $k \leq N_{\text{pc}}/2$, that $x_k > 0$ for $k > N_{\text{pc}}/2 + 1$ and $x_k = 0$ for $k = N_{\text{pc}}/2 + 1$.

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

```
function a=approx_pc(f,Npc)
% comments
```

```
% your name and the date
```

to compute the coefficients of the approximation as

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

Again, use `trapsum` to compute these integrals but use just 25 integration points since the range of integration is (presumably) small and the integrand is relatively slowly varying over that range..

- (b) Test your `approx_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 `approx_pc` with `Npc=10` on the function $f(x) = x$. You should get

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

- (d) We have already written a function called `bracket.m` that determines the values of k for which $x_k < x < x_{k+1}$. You may use yours or download `bracket.m` 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 (10) and has the signature

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

```
% your name and the date
```

To write this function, 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) Write an m-file called `test_pc.m` similar to `test_mon.m` and `test_leg.m` above. It should confirm that `Npc` is an even number, evaluate the coefficients (a) of the approximation using `approx_pc.m`, 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 10000 test points. It will be valuable to plot the approximation because it will help you debug your work and it will illustrate the process. Send me the plot for the case `Npc=8`. (Hint: use vector statements whenever possible or the calculations will take a long time.)
- (f) Fill in the following table for the partly quadratic

partly quadratic	
<code>Npc</code>	relative error
4	-----
8	-----
16	-----
32	-----

64 -----
256 -----
1024 -----
8192 -----

This approximation may take longer to compute, but it does not deteriorate as N_{pc} gets large! In fact, you should observe linear convergence. If you had the time, you could choose N_{pc} as large as you like and the approximation would get as close as you like (within reason).

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 tridiagonal form (in one dimension) that is almost as easy to solve. Continuity, however, can be worth the sacrifice, depending on the application. Even higher order piecewise polynomial approximation is possible, if the application can benefit.