MATH2070: LAB 6: Interpolation on evenly-spaced Points

1 Introduction

This lab will consume three sessions. It covers material from Sections 8.1, 8.2 and 10.1 of Quarteroni, Sacco, and Saleri on interpolation on evenly-spaced points. If you print this lab, you may prefer to use the pdf version.

This lab is concerned with interpolating data with polynomials and with trigonometric functions. There is a difference between interpolating and approximating. Interpolating means that the function *passes through* all the given points with no regard to the points between them. Approximating means that the function is close to all the given points and and has some additional mathematical properties often making it close to nearby points as well. It might seem that interpolating is always better than approximating, but there are many situations in which approximation is the better choice.

Polynomials are commonly used to interpolate data. There are several methods for defining, constructing, and evaluating the polynomial of a given degree that passes through data values. One common way is to solve for the coefficients of the polynomial by setting up a linear system called *Vandermonde's equation*. A second way is to represent the polynomial as the sum of simple *Lagrange polynomials*. Other ways are available and are discussed in the text and the lectures. In addition, we will discuss interpolation using trigonometric polynomials.

We will examine these two ways of finding the interpolating polynomial and a way of finding the interpolating trigonometric polynomial. We will see examples showing that interpolation does not necessarily mean good approximation and that one way that a polynomial interpolant can fail to approximate is because of a bad case of "the wiggles." Trigonometric polynomial interpolation does better, but can also break down.

1.1 Notation

This lab is focussed on finding functions that pass through certain specified points. In customary notation, you are given a set of points $\{(x_k, y_k)|k = 1, 2, \ldots, n\}$ and you want to find a function $f(x)$ that passes through each point ($f(x_k) = y_k$ for $k = 1, 2, \ldots, n$). That is, for each of the abscissæ, x_k, the function value $f(x_k)$ agrees with the ordinate y_k. The given points are variously called the "given" points, the "specified" points, the "data" points, *etc.* In this lab, we will use the term "data points."

Written in the customary notation, it is easy to see that the quantities x and x_k are essentially different. In Matlab, without kerning and font differentiation, it can be difficult to keep the various quantities straight. In this lab, we will use the name `xval` to denote the values of x and the names `xdata` and `ydata` to denote

the data x_k and y_k. The variable names `fval` or `yval` are used for the value $y = f(x)$ to emphasize that it is an interpolated value.

Generally, one thinks of the values `xval` as not equal to any of the data values `xdata` although, in this lab, we will often set `xval` to one or more of the `xdata` values in order to test that our interpolation is correct.

2 Matlab Tips

At some point in this lab, you will need to determine if an integer `m` is not equal to an integer `n`. The Matlab syntax for this is

```
if m ~= n
```

Sometimes it is convenient to enclose the logical expression in parentheses, but this is not required. Numbers with decimal parts should *never* be tested for equality! Instead, the absolute value of the difference should be tested to see if it is small. The reason is that numbers are rarely known to full 16-digit precision and, in addition, small errors are usually made in representing them in binary instead of decimal form. To check if a number `x` is equal to `y`, you should use

```
if abs(x-y) <= TOLERANCE
```

where `TOLERANCE` represents some reasonable value chosen based on the problem.

If you have a (square) linear system of equations, of the form

```
A * x = b
```

where `A` is an `N` by `N` matrix, and `b` and `x` are *column* vectors, then Matlab can solve the linear system *either* by using the expression:

```
x = inv(A)*b
```

or, *better*, via the "backslash" command:

```
x = A \ b
```

that you used in the previous labs. The backslash command is strange looking. You might remember it by noting that the matrix `A` appears underneath the backslash. The backslash is used because matrix multiplication is not commutative, so a square matrix should appear to the left of a column vector. The difference between the two commands is merely that the backslash command is about three times faster because it solves the equation without constructing the inverse matrix. You will not notice a difference in this lab, but you might see one later if you use Matlab for more complicated projects.

The backslash command is actually more general than just multiplication by the inverse. It can find solutions when the matrix `A` is singular or when it is not a square matrix! We will discuss how it does these things later in the course, but for now, be very careful when you use the backslash operator because it can find "solutions" when you do not expect a solution.

Matlab represents a polynomial as the vector of its coefficients. Matlab has several commands for dealing with polynomials:

`c=poly(r)` Finds the coefficients of the polynomial whose roots are given by the vector `r`.

`r=roots(c)` Finds the roots of the polynomial whose coefficients are given by the vector `c`.

`c=polyfit( xdata, ydata, n)` Finds the coefficients of the polynomial of degree `n` passing through, or approaching as closely as possible, the points `xdata(k),ydata(k)`, for `k` $= 1, 2, \ldots, K$, where $K - 1$ need not be the same as `n`.

`yval=polyval( c, xval)` Evaluates a polynomial with coefficients given by the vector `c` at the values `xval(k)` for $k = 1, 2, \ldots, K$ for $K \geq 1$.

When there are more data values than the minimum, the `polyfit` function returns the coefficients of a polynomial that "best fits" the given values in the sense of least squares. This polynomial approximates, but does not necessarily interpolate, the data. In this lab, you will be writing m-files with functions similar to `polyfit` but that generate polynomials of the precise degree determined by the number of data points. (`N=numel(xdata)-1`).

The coefficients c_k of a polynomial in Matlab are, by convention, defined as

$$p(x) = \sum_{k=1}^{N} c_k x^{N-k} \tag{1}$$

and in Matlab are represented as a vector `c`. **Beware:** With this definition of the vector `c` of coefficients of a polynomial:

1. `N=numel(c)` is one higher than the degree of $p(x)$.
2. The subscripts run *backwards*! `c(1)` is the coefficient of the term with degree `N-1`, and the constant term is `c(N)`.

In this and some later labs, you will be writing m-files with functions analogous to `polyfit` and `polyval`, using several different methods. Rather than following the matlab naming convention, functions with the prefix `coef_` will generate a vector of **coef**ficients, as by `polyfit`, and functions with the prefix `eval_` will **eval**uate a polynomial (or other function) at values of `xval`, as with `polyval`.

In the this lab and often in later labs we will be using Matlab functions to construct a known polynomial and using it to generate "data" values. Then we use our interpolation functions to recover the original, known, polynomial. This strategy is a powerful tool for illustrative and debugging purposes, but practical use of interpolation starts from arbitrary data, not contrived data.

3 The Polynomial through Given Data

In Lab 4 we discussed Newton's method, which can be derived by determining a linear polynomial (degree 1) that passes through the point (a, f_a) with derivative f'_a. That is $p(a) = f_a$ and $p'(a) = dp/dx(a) = f'_a$. One way of looking at this is that we are constructing an interpolating function, in this case a linear polynomial, that explains all the data that we have. We may then want to examine the graph of the polynomial, evaluate it at other points, determine its integral or derivative, or do other things with it. This is a reason that polynomial interpolation is important even when the functions we are interested in are not polynomials.

4 Vandermonde's Equation

Here's one way to see how to organize the computation of the polynomial that passes through a set of data.

Suppose we wanted to determine the linear polynomial $p(x) = c_1 x + c_2$ that passes through the data points (x_1, y_1) and (x_2, y_2). We simply have to solve a set of linear equations for c_1 and c_2 constructed by plugging in the two data points into the general linear polynomial. These equations are

$$\begin{aligned} c_1 x_1 + c_2 &= y_1 \\ c_1 x_2 + c_2 &= y_2 \end{aligned}$$

or, equivalently,

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

which (usually) has the solution

$$\begin{aligned} c_1 &= (y_2 - y_1)/(x_2 - x_1) \\ c_2 &= (x_2 y_1 - x_1 y_2)/(x_2 - x_1) \end{aligned}$$

Compare that situation with the case where we want to determine the quadratic polynomial $p(x) = c_1 x^2 + c_2 x + c_3$ that passes through three sets of data values. Then we have to solve the following set of three linear equations for the polynomial coefficients `c`:

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

These are examples of second and third order *Vandermonde Equations*. It is characterized by the fact that for each row (sometimes column) of the coefficient matrix, the succesive entries are generated by a decreasing (sometimes increasing) set of powers of a set of variables.

You should be able to see that, for *any* collection of abscissæ and ordinates, it is possible to define a linear system that should be satisfied by the (unknown) polynomial coefficients. If we can solve the system, and solve it accurately, then that is one way to determine the interpolating polynomial.

Now, let's see how to construct and solve the Vandermonde equation using Matlab. This involves setting up the coefficient matrix `A`. We use the Matlab variables `xdata` and `ydata` to represent the quantities x_k and y_k, and we will assume them to be row vectors of length (`numel`) `N`.

```
for j = 1:N
  for k = 1:N
    A(j,k) = xdata(j)^(N-k) ;
  end
end
```

Then we have to set the right hand side to the ordinates `ydata`, that is assumed to be a row vector. If we can get all of that set up, then actually solving the linear system is easy. We just write:

```
c = A \ ydata';
```

Recall that the backslash symbol means to solve the system with matrix `A` and right side `ydata'`. Notice that `ydata'` is the transpose of the row vector `ydata` in this equation. (By default, Matlab constructs row vectors unless told to do otherwise.)

> **Exercise 1**: The Matlab built-in function `polyfit` finds the coefficients of a polynomial through a set of points. We will write our own using the Vandermonde matrix. (This is the way that the Matlab function `polyfit` works.)
>
> (a) Write a Matlab function m-file, `coef_vander.m` with signature
>
> ```
> function c = coef_vander ( xdata, ydata )
> % c = coef_vander ( xdata, ydata )
> % xdata= ???
> % ydata= ???
> % c= ???
> % other comments
>
> % your name and the date
> ```

that accepts a pair of row vectors `xdata` and `ydata` of arbitrary but equal length, and returns the coefficient vector `c` of the polynomial that passes through that data. Be sure to complete the comments with question marks in them.
Warning: Think carefully about what to use for `N`.

(b) Test your function by computing the coefficients of the polynomial through the following data points. (This polynomial is $y = x^2$, so you can check your coefficient vector "by inspection.")

```
xdata= [  0   1   2 ]
ydata= [  0   1   4 ]
```

(c) Test your function by computing the coefficients of the polynomial that passes through the following points

```
xdata= [  -3  -2 -1  0  1   2    3]
ydata= [1636 247 28  7  4  31  412]
```

(d) Confirm using `polyval` that your polynomial passes through these data points.

(e) Double-check your work by comparing with results from the Matlab `polyfit` function. Please include both the full polyfit command you used and the coefficient vector it returned in your summary.

In the following exercise you will construct a polynomial using `coef_vander` to interpolate data points and then you will see what happens *between* the interpolation points.

Exercise 2:

(a) Consider the polynomial whose roots are `r=[-2 -1 1 2 3]`. Use the Matlab `poly` function to find its coefficients. Call these coefficients `cTrue`.

(b) This polynomial obviously passes through zero at each of these five "data" points. We want to see if our `coef_vander` function can reproduce it. To use our `coef_vander` function, we need a sixth point. You can "read off" the value of the polynomial at `x=0` from its coefficients `cTrue`. What is this value?

(c) Use the `coef_vander` function to find the coefficients of the polynomial passing through the "data" points

```
xdata=[ -2 -1 0  1 2 3];
ydata=[  0  0 ?? 0 0 0];
```

Call these coefficients `cVander`.

(d) Use `coef_vander` to find the coefficients using only the five roots as `xdata`. Name these coefficients something other than `cVander`. Explain your results.

(e) Use the following code to compute and plot the values of the true and interpolant polynomials on the interval [-3,2]. If you look at the last line of the code, you will see an estimate of the difference between the two curves. How big is this difference? (We will be using essentially this same code in several following exercises. You should be sure you understand what it does. You might want to copy it to a script m-file.)

```
xval=linspace(-2,3,4001);  % test abscissae for plotting
yvalTrue=polyval(cTrue,xval);  % true ordinates
yvalVander=polyval(cVander,xval);  % our ordinates
plot(xval,yvalTrue,'g','linewidth',4); % true curve: thick green
hold on
plot(xval,yvalVander,'k');  % interpolant curve: thin black
hold off
max(abs((yvalTrue-yvalVander)))/max(abs(yvalTrue))
```

Please include a copy of this plot with your summary. (You should observe the two curves are the same. The second curve appears as a thin black curve overlaying the thick green curve.)
Note: The relative error is used here so that the case where `yvalTrue` are very large or very small numbers does not cause difficulty.

5 Lagrange Polynomials

Suppose we fix the set of N distinct abscissæ x_k, $k = 1, \ldots, N$ and think about the problem of constructing a polynomial that has (not yet specified) values y_i at these points. Now suppose I have a polynomial $\ell_7(x)$ whose value is zero at each x_k, $k \neq 7$, and is 1 at x_7. Then the intermediate polynomial $y_7\ell_7(x)$ would have the value y_7 at x_7, and be 0 at all the other x_i. Doing the same for each abscissa and adding the intermediate polynomials together results in the polynomial that interpolates the data without solving any equations!

In fact, the *Lagrange polynomials* ℓ_k are easily constructed for any set of abscissae. Each Lagrange polynomial will be of degree $N-1$. There will be N Lagrange polynomials, one per abscissa, and the k^{th} polynomial $\ell_k(x)$ will have a special relationship with the abscissa x_k, namely, it will be 1 there, and 0 at the other abscissæ.

In terms of Lagrange polynomials, then, the interpolating polynomial has the form:

$$\begin{aligned} p(x) &= y_1\ell_1(x) + y_2\ell_2(x) + \ldots + y_N\ell_N(x) \\ &= \sum_{k=1}^{N} y_k\ell_k(x) \end{aligned} \tag{2}$$

Assuming we can determine these magical polynomials, this is a *second* way to define the interpolating polynomial to a set of data.

Remark: The strategy of finding a function that equals 1 at a distinguished point and zero at all other points in a set is very powerful. If $y_k = 1$ in (2), then $p(x) \equiv 1$, so the $\ell_k(x)$ are an example of a "partition of unity." One of the places you will meet it again is when you study the construction of finite elements for solving partial differential equations.

In the next two exercises, you will be constructing polynomials through the same points as in the previous exercise. Since there is only one nontrivial polynomial of degree (N-1) through N data points, the resulting interpolating polynomials are the same in these and the previous exercise.

In the next exercise, you will construct the Lagrange polynomials associated with the data points, and in the following exercise you will use these Lagrange polynomials to construct the interpolating polynomial.

Exercise 3: In this exercise you will construct Lagrange polynomials based on given data points. Recall the data set for $y = x^2$:

```
   k :    1   2   3
xdata= [  0   1   2   ]
ydata= [  0   1   4   ]
```

(Actually, `ydata` is immaterial for construction of $\ell_k(x)$.) In general, the formula for $\ell_k(x)$ can be written as:

$$\ell_k(x) = (f_1(x))(f_2(x)) \cdots (f_{k-1}(x))(f_{k+1}(x)) \cdots (f_N(x)) \tag{3}$$

(skipping the k^{th} factor), where each factor has the form

$$f_j(x) = \frac{(x - x_j)}{(x_k - x_j)}. \tag{4}$$

(a) Explain in a sentence or two why the formula (3) using the factors (4) yield a function

$$\ell_k(x_j) = \begin{cases} 1 & j = k \\ 0 & j \neq k \end{cases} \tag{5}$$

(b) Write a Matlab function m-file called `lagrangep.m` that computes the Lagrange polynomials (3) for any `k`. (One of the Matlab toolboxes has a function named "`lagrange`", so this one is named "`lagrangep`".) The signature should be

```
function pval = lagrangep( k , xdata, xval )
% pval = lagrangep( k , xdata, xval )
% comments
% k= ???
% xdata= ???
% xval= ???
% pval= ???

% your name and the date
```

and the function should evaluate the `k`-th Lagrange polynomial for the abscissæ `xdata` at the point `xval`. Hint, you can implement the general formula using code like the following.

```
pval = 1;
for j = 1 : ???
  if j ~= k
    pval = pval .* ???   % elementwise multiplication
  end
end
```

Note: If `xval` is a vector of values, then `pval` will be the vector of corresponding values, so that an elementwise multiplication (`.*`) is being performed.

(c) Using (5), determine the values of `lagrangep( 1, xdata, xval)` for `xval=xdata(1)`, `xval=xdata(2)` and `xval=xdata(3)`.

(d) Does `lagrangep` give the correct values for `lagrangep( 1, xdata, xdata)`? For `lagrangep( 2, xdata, xdata)`? For `lagrangep( 3, xdata, xdata)`?

Exercise 4: The hard part is done. Now we want to use our `lagrangep` routine as a helper for a second replacement for the `polyfit-polyval` pair, called `eval_lag`, that implements Equation (2). Unlike `coef_vander`, the coefficient vector of the polynomial does not need to be generated separately because it is so easy, and that is why `eval_lag` both fits and evaluates the Lagrange interpolating polynomial.

(a) Write a Matlab function m-file called `eval_lag.m` with the signature

```
function yval = eval_lag ( xdata, ydata, xval )
% yval = eval_lag ( xdata, ydata, xval )
%  comments

% your name and the date
```

This function should take the data values `xdata` and `ydata`, and compute the value of the interpolating polynomial at `xval` according to (2), using your `lagrangep` function for the Lagrange polynomials. Be sure to include comments to that effect.

(b) Test `eval_lag` on the simplest data set we have been using.

```
   k :    1   2   3
xdata= [  0   1   2 ]
ydata= [  0   1   4 ]
```

by evaluating it at `xval=xdata`. Of course, you should get `ydata` back.

(c) Test your function by interpolating the polynomial that passes through the following points, again by evaluating it at `xval=xdata`.

```
xdata= [  -3  -2 -1  0  1   2    3]
ydata= [1636 247 28  7  4  31  412]
```

(d) Repeat Exercise 2 using Lagrange interpolation.

- Return to the polynomial constructed in Exercise 2 with roots `r=[-2 -1 1 2 3].` and coefficients `cTrue`.
- Reconstruct (using `polyval`) or recall the `ydata` values associated with `xdata=[-2 -1 0 1 2 3]`.
- Compute the values of the Lagrange interpolating polynomial at the same 4001 test points between -2 and 3. Call these values `yvalLag`.
- Plot `yvalLag` and `yvalTrue` and compute the error between them using the test plotting approach used in Exercise 2. Include a copy of this plot with your summary.

6 Interpolating a function that is not a polynomial

Interpolating functions that are polynomials and using polynomials to do it is cheating a little bit. You have seen that interpolating polynomials can result in interpolants that are essentially identical to the original polynomial. Results can be much less satisfying when polynomials are used to interpolate functions that are not themselves polynomials. At the interpolation points, the function and its interpolant agree exactly, so we want to examine the behavior *between* the interpolation points. In the following exercise, you will see that some non-polynomial functions can be interpolated quite well, and in the subsequent exercise you will see one that cannot be interpolated well. The example used here is due in part to C. Runge, *Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten,* Z. Math. Phys., 46(1901), pp. 224-243.

Exercise 5: In this exercise you will construct interpolants for the hyperbolic sine function $\sinh(x)$ and see that it and its polynomial interpolant are quite close.

(a) We would like to interpolate the function $y = \sinh(x)$ on the interval $[-\pi, \pi]$, so use the following outline to examine the behavior of the polynomial interpolant to the exponential function for five evenly-spaced points. It would be best if you put these commands into a script m-file named `exer5.m`.

```
% construct N=5 data points
N=5;
xdata=linspace(-pi,pi,N);
ydata=sinh(xdata);

% construct many test points
xval=linspace(???,???,4001);
% construct the true test point values, for reference
yvalTrue=sinh(???);

% use Lagrange polynomial interpolation to evaluate
% the interpolant at the test points
```

```
yval=eval_lag(???,???,xval);

% plot reference values in thick green
plot(xval,yvalTrue,'g','linewidth',4);
hold on
% plot interpolation data points
plot(xdata,ydata,'k+');
% plot interpolant in thin black
plot(xval,yval,'k');
hold off

% estimate the approximation error of the interpolant
approximationError=max(abs(yvalTrue-yval))/max(abs(yvalTrue))
```

Please send me this plot.

(b) By zooming, *etc.*, confirm visually that the exponential and its interpolant agree at the interpolation points.

(c) Using more data points gives higher degree interpolation polynomials. Fill in the following table using Lagrange interpolation with increasing numbers of data points.

```
N =  5  Approximation Error = ________
N = 11  Approximation Error = ________
N = 21  Approximation Error = ________
```

You should have observed in Exercise 5 that the approximation error becomes quite small. The exponential function is entire, as are polynomials, so they share one essential feature. In the following exercise, you will see that attempts to interpolate functions that are not entire can give poor results.

Exercise 6:

(a) Construct a function m-file for the Runge example function $y = \frac{1}{1+x^2}$. Name the file `runge.m` and give it the signature

```
function y=runge(x)
% y=runge(x)
% comments

% your name and the date
```

Use componentwise (vector) division and exponentiation (`./` and `.^`).

(b) Copy `exer5.m` to `exer6.m` and modify it to use the Runge example function. Please send me the plot it generates.

(c) Confirm visually that the Runge example function and its interpolant agree at the interpolation points, but not necessarily between them.

(d) Using more data points gives higher degree interpolation polynomials. Fill in the following table using Lagrange interpolation with increasing numbers of data points.

```
N =  5  Approximation Error = ________
N = 11  Approximation Error = ________
N = 21  Approximation Error = ________
```

(e) Are you surprised to see that the errors do not decrease?

Many people expect that an interpolating polynomial $p(x)$ gives a good approximation to the function $f(x)$ everywhere, no matter what function we choose. If the approximation is not good, we expect it to get better if we increase the number of data points. These expectations will be fulfilled only when the function does not exhibit some "essentially non-polynomial" behavior. You will see why the Runge example function cannot be approximated well by polynomials in the following exercise.

Exercise 7: The Runge example function has Taylor series

$$\frac{1}{1+x^2} = 1 - x^2 + x^4 - x^6 + \dots \tag{6}$$

as you can easily prove. This series has a radius of convergence of 1 in the complex plane. Polynomials, on the other hand, are entire functions, *i.e.,* their Taylor series converge everywhere in the complex plane. No finite sum of polynomials can be anything but entire, but no entire function can interpolate the Runge example function on a disk with radius larger than one about the origin in the complex plane. If there were one, it would have to agree with the series (6) inside the unit disk, but the series diverges at $x = i$ and an entire function cannot have an infinite value (a pole).

(a) Make a copy of `exer6.m` called `exer7.m` that uses `coef_vander` and `polyval` to evaluate the interpolating polynomial rather than `eval_lag`.

(b) Confirm that you get the same results as in Exercise 6 when you use Vandermonde interpolation for the Runge example function.

```
N =  5  Approximation Error = ________
N = 11  Approximation Error = ________
N = 21  Approximation Error = ________
```

(c) Look at the nontrivial coefficients (c_k) of the interpolating polynomials by filling in the following table.

```
N=5   c( 5)= _____  c( 3)= _____  c( 1)= _____
N=11  c(11)= _____  c( 9)= _____  c( 7)= _____  c( 5)= _____
N=21  c(21)= _____  c(19)= _____  c(17)= _____  c(15)= _____
limiting      +1            -1            +1            -1
```

(d) Look at the trivial coefficients (c_k) of the interpolating polynomials by filling in the following table. (Look carefully at what the colon notation does.)

```
N=5   max(abs(c(2:2:end)))= _____
N=11  max(abs(c(2:2:end)))= _____
N=21  max(abs(c(2:2:end)))= _____
```

You should see that the interpolating polynomials are "trying" to reproduce the Taylor series (6). These polynomials cannot agree with the Taylor series at all points, though, because the Taylor series does not converge at all points.

7 Trigonometric polynomial interpolation

Quarteroni, Sacco, and Saleri discuss interpolation by trigonometric polynomials in Section 10.1. Trigonometric interpolation is closely related to approximation by Fourier series, but we will focus on interpolation in this lab.

The basic expression for trigonometric interpolation using $(2N+1)$ functions over the interval $[-\pi, \pi]$ is

$$f(x) = \sum_{k=1}^{2N+1} a_k e^{i(k-N-1)x}. \tag{7}$$

(We consider $(2N+1)$ points because we want to use real functions and real interpolants. Thus one trigonometric function $e^{i(k-N-1)x}$ is the constant term when $k = N+1$, and all the rest come in complex conjugate pairs.)

Using the $2N+1$ evenly-spaced points in the interval $(-\pi, \pi)$ given by

$$x_j = \frac{2\pi(j-N-1)}{2N+1}, \text{ for } j = 1, 2, \ldots, (2N+1), \tag{8}$$

the coefficients can be determined from

$$a_k = \frac{1}{2N+1} \sum_{j=1}^{2N+1} e^{-i(k-N-1)x_j} f(x_j), \text{ for } k = 1, 2, \ldots, (2N+1). \tag{9}$$

These equations can be found directly by multiplying (7) through by each of the functions $e^{-i(k-N-1)x}$ in turn, applying it to the values x_k, and solving the resulting system for a_j, taking advantage of the properties of the complex exponential to simplify the equations.

Remark: The trigonometric coefficients a_k in (9) play the same role as the polynomial coefficients c_k in (1).

In the following exercises, we are going to write functions `coef_trig` to be analogous to `polyfit` and `coef_vander`, and also `eval_trig` to be analogous to `polyval` and `eval_lag`.

Exercise 8:

(a) Construct a function m-file named `coef_trig.m` with the signature

```
function a=coef_trig(func,N)
% a=coef_trig(func,N)
% func=???
% N=???
% a=???
% comments

% your name and the date
```

This function should evaluate the trigonometric coefficients a_k according to Equation (9). Use Equation (8) to determine the points x_k. It is more efficient to use vector (componentwise) notation and the Matlab `sum` function, but if you cannot see how to do that, just use `for` loops.

Hint: You can use the following code to generate the points x_k without writing a `for` loop.

```
xdata = 2*pi*(-N:N)/(2*N+1);
```

Note: The length (`numel`) of `a` should be `2*N+1`.

(b) Test your function by applying it to the function $f(t) = e^{ix}$, for N=10. (You can either write an m-file for e^{ix} or use the `@` command to define an "anonymous function.") By examining Equation (7), you should be able to see that $a_k = 1$ for $k = N+2$ and zero otherwise.

(c) Test your function again by applying it to $f(x) = \sin 4x$ with `N=10`. You should see that a_k is zero for all but two subscripts. What subscripts (k) have non-zero a_k and what are the values?

Exercise 9:

(a) Construct a function m-file named `eval_trig.m` with the signature

```
function fval=eval_trig(a,xval)
% fval=eval_trig(a,xval)
% a=???
```

```
% xval=???
% fval=???
% comments

% your name and the date
```

This function should evaluate Equation (7) at an arbitrary collection of points, `xval`.

(b) Test your function by first using `coef_trig` to find the coefficients for the function $\sin 4x$ with `n=10` (you did this in the previous exercise), and then applying `eval_trig` to those coefficients at 4001 equally-spaced points in the interval $[-\pi, \pi]$. Compare the interpolated values versus $\sin 4x$. Plot them on the same plot: the lines should overlap. You can modify the m-file `exer5.m` that you wrote for Exercise 5. Call your modified m-file `exer9.m`. (**Warning:** Your interpolated values may appear to Matlab to be complex, and Matlab may not plot them the way you expect. In that case, verify that their imaginary parts (Matlab function `imag`) are zero and plot only the real (Matlab function `real`) parts. Please send me the plot.

Trigonometric polynomial interpolation does well even when the functions are not themselves trigonometric polynomials.

Exercise 10:

(a) Modify `exer9.m` to use the function $y = x(\pi^2 - x^2)$, and call the modified file `exer10.m`. Plot the function and its interpolant for `N=5`, `10`, `15` and `N=20`. You do not have to send me the plots, but fill in the following table.

```
N =  5: y = x.*(pi^2-x.^2) Approximation Error = ________
N = 10: y = x.*(pi^2-x.^2) Approximation Error = ________
N = 15: y = x.*(pi^2-x.^2) Approximation Error = ________
N = 20: y = x.*(pi^2-x.^2) Approximation Error = ________
```

(b) Do the same thing for the Runge example function and fill in the following table.

```
N = 2  Runge Approximation Error = ________
N = 4  Runge Approximation Error = ________
N = 5  Runge Approximation Error = ________
```

You do not have to send me the plots.

(c) For the `N=2` case, and the Runge example function, what are the five points at which the trigonometric interpolation should *interpolate* the Runge example function? Check that `runge` and `eval_trig` agree (up to roundoff) at those points.

(d) If you have not done so already, modify your script file to include plotting plus signs at each interpolation point.

The previous exercise shows that trigonometric polynomial interpolation does well for some functions. It does much less well when the function is not continuous or not periodic in $[-\pi, \pi]$. Consider, for example, the function

$$f(x) = \begin{cases} x + \pi & \text{for } x < 0 \\ x - \pi & \text{for } x \geq 0 \end{cases}$$

Exercise 11:

(a) Copy the following code into a function m-file named `sawshape6.m`

```
function y=sawshape6(x)
% y=sawshape6(x)
% x=???
% y=???
% comments

% your name and the date

kless=find(x<0);
kgreater=find(x>=0);
y(kless)=x(kless)+pi;
y(kgreater)=x(kgreater)-pi;
```

Be sure to add the usual comments.

(b) The Matlab `find` command is a *very* useful function. Use the `help find` command or the Help facility to see what it does.

(c) If the letters `a` through `j` represent a sequence of ten increasing values, and if `x=[a b c d e f g h i j i h g]`, what is the result of the function calls `find(x==c)`, `find(x==h)`, and `find(x>=g)`?

(d) Use a script file based on `exer9.m` to fill in the following table by interpolating the `sawshape6` function using trigonometric interpolation.

```
N =   5  Approximation Error = ________
N =  10  Approximation Error = ________
N = 100  Approximation Error = ________
```

You can see that the approximation error is not shrinking to zero, but if you look at the plots you will see that the error really is shrinking everywhere except near $x = 0$, where the oscillations get more rapid but do not get smaller. This behavior is typical and is called Gibbs's phenomenon. Please send me the plots for `N=5` and `N=100`.

(e) Examine the plot for the `N=5` case. Are the plus signs in the proper places to indicate the interpolation points?

Increasing the order of polynomial interpolation can lead to divergence of approximation because of "the wiggles." Trigonometric interpolation does not diverge as $n \to \infty$, but Gibbs phenomenon slows convergence drastically at discontinuities.

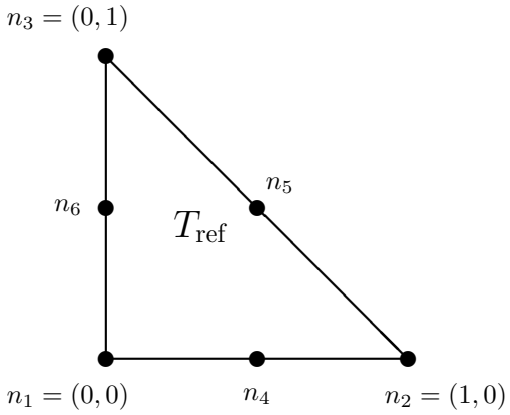
8 Extra credit: Interpolation over triangles (8 points)

In this extra credit exercise, you will see how to find a quadratic interpolating polynomial for a function given on an arbitrary triangle in the plane. The method used will be a two-stage one

1. Transform the given triangle into a "reference" triangle; and,
2. Use the two-dimensional Lagrange interpolating polynomials over the reference triangle to interpolate the function.

The reason for this two-stage approach is that is is more convenient to generate the Lagrange polynomials once over a standard triangle than to generate Lagrange polynomials for an arbitrary triangle. A second reason is that each of the stages is easily programmed and can be used repeatedly. Computer programs using finite element methods may involve thousands or millions of triangles.

Consider the following reference triangle, $T_{\text{ref}} = \{(\xi, \eta) \,|\, 0 \le \xi \le 1,\ 0 \le \eta \le (1 - \xi)\}$.



In this triangle, the three nodes n_4, n_5, and n_6 are midway along the indicated sides.

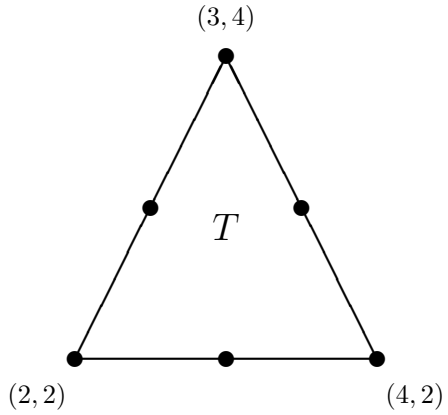
There are three linear Lagrange polynomials defined on T_{ref}. It should be clear that any linear polynomial can be written as a linear combination of $\ell_1 \dots \ell_3$.

$$\begin{aligned}
\ell_1(\xi,\eta) &= 1 - \xi - \eta \\
\ell_2(\xi,\eta) &= \xi \\
\ell_3(\xi,\eta) &= \eta
\end{aligned}$$

There are six quadratic Lagrange polynomials defined on T_{ref}. It should be clear that any quadratic polynomial can be written as a linear combination of $q_1 \dots q_6$.

$$\begin{aligned}
q_1(\xi,\eta) &= 2(1 - \xi - \eta)(.5 - \xi - \eta) \\
q_2(\xi,\eta) &= 2\xi(\xi - .5) \\
q_3(\xi,\eta) &= 2\eta(\eta - .5) \\
q_4(\xi,\eta) &= 4\xi(1 - \xi - \eta) \\
q_5(\xi,\eta) &= 4\xi\eta \\
q_6(\xi,\eta) &= 4\eta(1 - \xi - \eta)
\end{aligned}$$

In the following exercise, you will be using these polynomials to interpolate functions on the following triangle.



The relationship between the reference triangle T_{ref} expressed in (ξ, η) coordinates and the triangle T expressed in (x, y) coordinates, as well as the function $p(x, y) = e^{0.1xy}$, can be summarized in the following table.

(x, y)	(ξ, η)	p
$(2, 2)$	$(0, 0)$	$e^{0.4}$
$(4, 2)$	$(1, 0)$	$e^{0.8}$
$(3, 4)$	$(0, 1)$	$e^{1.2}$
$(3, 2)$	$(0.5, 0)$	$e^{0.6}$
$(3.5, 3)$	$(0.5, 0.5)$	$e^{1.05}$
$(2.5, 3)$	$(0, 0.5)$	$e^{0.75}$

Exercise 12: **Note:** It is possible to do this exercise without writing any Matlab code. Please include a description of the work you do outside of Matlab in your summary (or in a separate file, or give me a handwritten explanation) and send me any Matlab m-files you use. If you wish to use the symbolic toolbox for this lab, it would be a good learning exercise.

(a) Confirm that ℓ_i are Lagrange polynomials by directly computing the values $\ell_i(\xi, \eta)$, $i = 1, \ldots, 3$, for the nodal points n_j $j = 1, \ldots, 3$. Each ℓ_i should be 1 at n_i and 0 at n_j for $j \neq i$.

(b) Confirm that q_i are Lagrange polynomials by directly computing the values $q_i(\xi, \eta)$, $i = 1, \ldots, 6$, for the nodal points n_j $j = 1, \ldots, 6$. Each q_i should be 1 at n_i and 0 at n_j for $j \neq i$.

(c) Find a linear function $x = x(\xi, \eta)$ so that $x(n_1) = 2$, $x(n_2) = 4$ and $x(n_3) = 3$. Note that 2, 4, and 3 are the x-coordinates of the three vertices of T. You can use the Lagrange polynomials ℓ_i. Write this as a formula for x in terms of ξ and η.

(d) Similarly, find a linear function $y = y(\xi, \eta)$ so that $y(n_1) = 2$, $y(n_2) = 2$ and $y(n_3) = 4$. Note that 2, 2, and 4 are the y-coordinates of the three vertices of T. You can use the Lagrange polynomials ℓ_i. Write this as a formula for y in terms of ξ and η.

(e) You have just constructed a mapping from $(\xi, \eta) \in T_{\text{ref}}$ to $(x, y) \in T$. This mapping can be written in vector form as

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + J \begin{pmatrix} \xi \\ \eta \end{pmatrix}$$

where J is a 2×2 matrix. What are the vector components x_0, y_0, and the components of the matrix J?

(f) The inverse of this mapping can be written as

$$\begin{pmatrix} \xi \\ \eta \end{pmatrix} = J^{-1} \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix}$$

This mapping can also be denoted as $\xi = \xi(x, y)$ and $\eta = \eta(x, y)$.

(g) The quadratic polynomial interpolant $p(x, y)$ of the function $e^{0.1xy}$ over the triangle T can be written as

$$p(x, y) = \sum_{k=1}^{6} a_k q_k(\xi(x, y), \eta(x, y)) \tag{10}$$

(also see the table at the beginning of this exercise). What are the values a_k for $k = 1, \ldots, 6$? **Hint**: The top of the triangle is the point $x = 3$ and $y = 4$. What are the right and left sides of (10) at that point?

(h) What is the value of $p(3.5, 2.5)$?

Last change `$Date: 2016/09/30 00:23:47 $`