MATH2070: LAB 7: Polynomial and Piecewise Linear Interpolation

Introduction	Exercise 1
Matlab Hints	Exercise 2
Recursive functions	Exercise 3
An Experimental Framework	Exercise 4
Chebyshev Points	Exercise 5
Bracketing	Exercise 6
Piecewise Linear Interpolation	Exercise 7
	Exercise 8
Approximating the derivative (Extra)	Exercise 9
	Exercise 10
	Exercise 11
	Exercise 12
	Exercise 13
	Exercise 14
	Extra Credit

1 Introduction

We saw in the last lab that the interpolating polynomial could get *worse* (in the sense that values at intermediate points are far from the function) as its degree increased. This means that our strategy of using equally spaced data for high degree polynomial interpolation is *a bad idea*. It turns out that equidistant spacing must always result in poor asymptotic convergence rates! See the excellent article by Platte, Trefethen, and Kuijlaars, *Impossibility of Fast Stable Approximation of Analytic Functions from Equispaced Samples*, SIAM Review, 53#(2) pp. 308-318, `www.siam.org/journals/sirev/53-2/77470.html`. In this lab, we will test other possible strategies for spacing the data and then we will look at an alternative way to do interpolation with a guarantee that the error will get smaller when we take more points.

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

2 Matlab Hints

- When you encounter an error while using an m-file look at the *first* error message you receive, not the last. This message will identify the file from which it came, and the line number. Be careful: the file that caused the error may not be the file you think you are using–it could be called from the one you think you are using.
- The error message `Not enough input arguments.` often means that you forgot the `@` sign in front of a function name when you used it as an argument in an m-file.

3 Recursive functions

In this section, we discuss programming recursive functions. As you will see, recursive functions can be very short and elegant when explicit formulas or loops are awkward and difficult to understand.

Consider the factorial function. Its mathematical definition can be written as

$$0! = 1$$
$$n! = n\Big((n-1)!\Big) \qquad \text{for } n \geq 1$$

A function defined this way is termed "recursive" because it is defined in terms of itself. You have seen such definitions often in your theoretical work, but they can be confusing when you try to program them.

In this section, you will see how to write recursive functions in Matlab. It can be shown that any function written in a recursive manner can be rewritten using loops; however, the recursive form of the function is generally shorter and easier to understand.

Exercise 1: The following code implements the factorial function. We will call it `rfactorial` because Matlab already has the factorial function and it is called `factorial`.

(a) Copy the following code to a file named `rfactorial.m`.

```
function f=rfactorial(n)
% f=rfactorial(n) computes n! recursively

% your name and the date

if n<0
  error('rfactorial: cannot compute factorial of negative integer.');
elseif n==0  %double equals sign for logical testing
  f=1;
else
  f=n*rfactorial(n-1);
end
```

(b) Confirm that `rfactorial` gets the same result for 5! as Matlab's `factorial` function.

(c) Suppose you are computing `factorial(3)`. When `rfactorial` starts up, the first thing it does is check if `n<0`. Since `n=3`, it goes on and discovers that before it can continue, it must start up a *new copy* of `rfactorial` and compute `factorial(2)`. In turn, this new copy checks that `2>1` and starts up *another new copy* of `rfactorial` so that there are now two copies of `rfactorial` waiting and one copy active. In your own words, explain how the computation continues to get a result of 6.

(d) If you attempt to use `rfactorial` for a negative integer, it will print an error message and stop because of the `error` function. What would have happened if the code did not test if `n<0`?

The factorial function is really very simple, and recursion is, perhaps, too powerful a method to employ. After all, it can be written as a simple loop or, even more simply, as `prod(1:n)`. Consider the Fibonacci series

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \qquad \text{for } n > 1. \end{aligned} \tag{1}$$

Values of f_n can also be computed using Binet's formula

$$f_n = \frac{(1+\sqrt{5})^{n+1} - (1-\sqrt{5})^{n+1}}{2^{n+1}\sqrt{5}}.$$

You can find more information about the Fibonacci series from the Wolfram (Mathematica) web site http://mathworld.wolfram.com/FibonacciNumber.html.

Exercise 2:

(a) Write a Matlab function named `fibonacci` that accepts a value of n and returns the element f_n of the Fibonacci series according to the prescription given in (1).

(b) Confirm that your function correctly computes the first few Fibonacci numbers: `1,1,2,3,5,8,13`.

(c) Confirm that Binet's formula gives the approximately same value when `n=13` as your `fibonacci` gives.

The message you should be getting is that recursion is a powerful programming technique that allows you to do some things more elegantly than using loops (*e.g.*, `fibonacci`), and to do some things with essentially the same effort as using loops (*e.g.*, `factorial`). Although not presented in this lab, there are examples where a recursive program is simply the wrong approach.

It is also true that some recursive programs (such as `factorial`) run just about as fast as their loop counterparts while others (such as `fibonacci`) can be *much* slower than the same thing written as a loop.

4 An Experimental Framework

In the previous lab, we compared several different methods of polynomial interpolation, and we used a common approach that standardized the comparisons. Part of this lab will be to generate polynomial interpolants for a few different functions on different sets of points. We will be comparing the accuracy of the interpolating polynomials, just as we did last lab. Instead of repeating basically the same code all the time, it is more convenient to automate the process in an m-file. The utility function in the next exercise is designed to test interpolation for different functions on different sets of points.

Remark: I often use Matlab to do comparisons of several different methods. I find it very helpful to build a utility such as the one below to standardize the comparisons as well as to make them quick and easy. It is both error-prone and bothersome to repeat the same sequence of commands several times.

Exercise 3:

In this exercise, you will construct a utility function m-file that will take as input a function (handle) to interpolate and a set of points `xdata` on which to perform the interpolation. In the previous lab, the points `xdata` were uniformly distributed but in this lab we will investigate non-uniform distributions. The interpolation error will be computed by picking a much larger number of test points than are used for interpolation and computing the maximum error on this larger set of test points. The function uses `eval_lag.m` and `lagrangep.m` from last lab. You can use your versions or download mine. An outline of this function follows:

```
function max_error=test_poly_interpolate(func,xdata )
% max_error=test_poly_interpolate(func,xdata )
% utility function used for testing polynomial interpolation
% func is the function to be interpolated
% xdata are abscissae at which interpolation takes place
% max_error is the maximum difference between the function
%   and its interpolant

% your name and the date

% Choose the number of the test points and generate them
% Use 4001 because it is odd, capturing the interval midpoint
NTEST=4001;
% construct NTEST points evenly spaced so that
```

```
% they cover the interpolation interval in a standard way, i.e.,
% xval(1)=xdata(1) and xval(NTEST)=xdata(end)
xval= ???

% we need the results of func at the points xdata to do the
% interpolation WARNING: this is a vector statement
% In a real problem, ydata would be "given" somehow, and
% a function would not be available
ydata=func(xdata);

% use Lagrange interpolation from lab6 to do the interpolation
% WARNING: these use componentwise (vector) statements.
% Generate yval as interpolated values corresponding to xval
yval=eval_lag( ???

% we will be comparing yval with the exact results of func at xval
% In a real problem, the exact results would not be available.
yexact= ???

% plot the exact and interpolated results on the same plot
% this gives assurance that everything is reasonable
plot( ???

% compute the error in a standard way.
max_error=max(abs(yexact-yval))/max(abs(yexact));
```

Use cut-and-paste to copy this code from the web page.

(a) Look a the code in `test_poly_interpolate.m` and complete the statements that have question marks in them.

(b) Test your `test_poly_interpolate.m` function by getting the same approximation values for the Runge example function $f(x) = 1/(1 + x^2)$ on the interval $[-\pi, \pi]$ with uniformly spaced points as you should have seen in last lab and are reproduced here Recover your `runge.m` file from last lab or rewrite it. Do not forget to use componentwise (vector) syntax.

```
       Runge function, evenly spaced points
ndata =  5  Max Error = 0.31327
ndata = 11  Max Error = 0.58457
ndata = 21  Max Error = 3.8607
```

(c) Construct `xdata` containing `ndata=5` evenly spaced points between -5 and +5 (*not* the same interval) and use the utility `test_poly_interpolate` to plot and evaluate the error between the interpolated polynomial and the exact values of the Runge example function, and fill in the following table.

```
    Runge function, evenly spaced points
ndata =  5  Max Error = ________
ndata = 11  Max Error = ________
ndata = 21  Max Error = ________
```

After looking at the plotted comparisons between the Runge example function and its polynomial fit, you might guess that the poor fit is because the points are evenly spaced. Maybe they should be concentrated

near the endpoints (so they don't oscillate wildly there) or near the center (maybe the oscillations are caused by too many points near the endpoints). Let's examine these two hypotheses.

The strategy used in the next exercise is to use a function to change the distribution of points. That is, pick a nonlinear function f that maps the interval $[-1, 1]$ into itself (we will use x^2 and $x^{1/2}$) and also pick an affine function g that maps the given interval, $[-5, 5]$ into $[-1, 1]$. Then use the points $x_k = g^{-1}(f(g(\tilde{x}_k)) = g^{-1} \circ f \circ g(\tilde{x}_k)$, where $\tilde{x}_k$ are uniformly distributed.

Exercise 4: We have been looking at evenly-spaced points between -5 and 5. One way to concentrate these points near zero would be to recall that, for numbers less than one, $|x|^2 < |x|$. Hence, if `xdata` represents a sequence of numbers uniformly distributed between -5 and 5, then the expression `5*( sign(xdata).*abs(xdata./5).^2 )` yields a similar sequence of numbers concentrated near 0. (The `sign` and `abs` functions are used to get the signs correct for negative values.)

(a) Construct 11 points uniformly distributed between $[-5, 5]$ in a vector called `xdata`. Use the transformation

```
xdata=5*( sign(xdata).*abs(xdata./5).^2 )
```

Look at the concentrated distribution of points using the command

```
plot(xdata,zeros(size(xdata)),'*')
```

to plot the redistributed points along the x-axis. Your points should be concentrated near the center of the interval $[-5, 5]$ but with endpoints $+5$ and -5. Be sure that the endpoints are correct. You do not need to send me this plot.

(b) Use this transformation and the `test_poly_interpolate` utility from Exercise 3 to fill in the following table for the Runge example function with `ndata` points concentrated near 0.

```
     Runge function, points concentrated near 0
ndata =  5  Max Error = ________
ndata = 11  Max Error = ________
ndata = 21  Max Error = ________
```

You should observe that the error is rising dramatically faster than in the uniformly distributed case.

Exercise 5: Points can be concentrated near the endpoints in a similar manner, by forcing them away from zero using the $|x|^{1/2} = \sqrt{|x|}$ function.

(a) Use a transformation similar to the one in the previous exercise but with $|x|^{1/2}$ with 11 points. Take care to get the signs of negative `xdata` correct.

(b) Plot your points using the following command.

```
plot(xdata,zeros(size(xdata)),'*')
```

Please send me this plot.

(c) Use this transformation and the `test_poly_interpolate` utility from Exercise 3 to fill in the following table for the Runge example function with `ndata` points concentrated near the endpoints.

```
     Runge function, points concentrated near endpoints
ndata =  5  Max Error = ________
ndata = 11  Max Error = ________
ndata = 21  Max Error = ________
```

You should observe that the error is shrinking, in contrast with the uniformly distributed case.

You could ask why I chose x^2 and $x^{1/2}$ out of the many possibilities. Basically, it was an educated guess. It turns out, though, that there is a systematic way to pick optimally-distributed (Chebyshev) points. It is also true that the Chebyshev points are closely related to trigonometric interpolation (discussed last time).

5 Chebyshev Points

If we have no idea what function is generating the data, but we are allowed to pick the locations of the data points beforehand, the Chebyshev points are the smartest choice. (There is more on Chebyshev points and interpolation in Quarteroni, Sacco, and Saleri in Sections 10.1-10.3.)

Quarteroni, Sacco, and Saleri (Theorem 8.2) show that the interpolation error between a function f and its polynomial interpolant p at any point `x` is given by an expression of the form:

$$f(x) - p(x) = \left[\frac{(x - x_1)(x - x_2)\ldots(x - x_n)}{n!}\right] f^n(\xi) \tag{2}$$

where ξ is an unknown nearby point. This is something like an error term for a Taylor's series.

We can't do a thing about the expression $f^n(\xi)$, because f is an arbitrary (smooth enough) function, but we *can* affect the magnitude of the bracketed expression. For instance, if we are only using a single data point ($n = 1$) in the interval `[10,20]`, then the very best choice for the data point would be $x_1 = 15$, because then the maximum absolute value of the expression

$$\left[\frac{(x - x_1)}{1!}\right]$$

would be 5. The worst value would be to choose x at one of the endpoints, in which case we'd double the maximum value. This doesn't guarantee better results, but it improves the chance.

The Chebyshev polynomial of degree n is given by

$$T_n(x) = \cos(n \cos^{-1} x). \tag{3}$$

This formula doesn't look like it generates a polynomial, but the trigonometric formulæ for sums of angles and the relation that $\sin^2\theta + \cos^2\theta = 1$ can be used to show that it does generate a polynomial.

These polynomials satisfy an orthogonality relationship and a three-term recurrence relationship

$$T_n = 2xT_{n-1}(x) - T_{n-2}(x), \text{ with } T_0(x) = 1 \text{ and } T_1(x) = x. \tag{4}$$

The facts that make Chebyshev polynomials important for interpolation are

- The peaks and valleys of T_n are smallest of all polynomials of degree n over [-1,1] with $T_n(1) = 1$. (Quarteroni, Sacco, and Saleri, Property 10.2.) See also `http://en.wikipedia.org/wiki/Chebyshev_polynomials`); and,
- On the interval [-1,1], each polynomial oscillates about zero with peaks and valleys all of equal magnitude. (Quarteroni, Sacco, and Saleri, Property 10.1.)

Thus, when $\{x_1, x_2, \ldots, x_n\}$ in (2) are chosen to be the roots of $T_n(x)$, then the bracketed expression in (2) is proportional to T_n, and the bracketed expression is minimized over all polynomials.

Before going on to use the Chebyshev points, it is a good idea to confirm that the two expressions (3) and (4) do, in fact, refer to the same polynomials. Mathematically speaking, the best way to approach the problem is to use the symbolic toolbox, which can produce the identity that can become the central portion of a proof. Instead, we will approach the problem numerically. This approach will yield a convincing example, but no proof.

Exercise 6:

(a) Write a Matlab function with the signature and first few lines,

```
function tval=cheby_trig(xval,degree)
% tval=cheby_trig(xval,degree)

% your name and the date

if nargin==1
  degree=7;
end
```

to evaluate $T_n(x)$ as defined using trigonometric functions in (3) above. If the argument `degree` is omitted, it defaults to 7, a fact that will be used below.

(b) Write a recursive Matlab function with the signature and first few lines,

```
function tval=cheby_recurs(xval,degree)
% tval=cheby_recurs(xval,degree)

% your name and the date

if nargin==1
  degree=7;
end
```

to evaluate $T_n(x)$ as defined recursively in (4) above. If the argument `degree` is omitted, it defaults to 7, a fact that will be used below.

(c) Show that `cheby_trig` and `cheby_recurs` agree for degree=4 (T_4) at the points `x=[0,1,2,3,4]` by computing the largest absolute value of the differences at those points. What is this value?
Remark: If two polynomials of degree 4 agree at 5 points, they agree everywhere. Hence *if (3) defines a polynomial,* then (3) and (4) produce identical values for T_4. This is why only five test points are required.

(d) Plot values of `cheby_trig` and `cheby_recurs` on the interval [-1.1, 1.1] for `degree=7` (T_7). Use at least 100 points for this plot in order to see the details. The two lines should lie on top of one another. You should also observe visually that the peaks and valleys of T_7 are of equal absolute value. Please include this plot with your summary.
Suggestion: You can plot the first curve with a wider line than the second, as you did in the previous lab, to see both lines.

(e) Visually choose change-of-sign intervals around the largest and second-largest roots in your plot of T_7. Using your `bisect.m` file from Lab 3 or my version of `bisect.m`, find the largest and second-largest roots of $T_7(x) = 0$ on the interval [-1.1, 1.1].

5.1 Constructing the Chebyshev points

The Chebyshev points are the zeros of the Chebyshev polynomials. They can be found from (3):

$$\begin{aligned} \cos(n\cos^{-1}x) &= 0, \\ n\cos^{-1}x &= (2k-1)\pi/2, \\ \cos^{-1}x &= (2k-1)\pi/(2n), \\ x &= \cos\left(\frac{(2k-1)\pi}{2n}\right) \end{aligned}$$

For a given number n of data points, then, the Chebyshev points on the interval $[a, b]$ can be constructed in the following way.

1. Pick equally-spaced angles $\theta_k = (2k-1)\pi/(2n)$, for $k = 1, 2, \ldots, n$.
2. The Chebyshev points on the interval $[a, b]$ are given as

$$x_k = 0.5(a + b + (a - b)\cos\theta_k),$$

for $k = 1, 2, \ldots, n$.

Exercise 7:

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

```
function xdata = cheby_points ( a, b, ndata )
% xdata = cheby_points ( a, b, ndata )
% more comments

% your name and the date
```

that returns in the vector `xdata` the values of the `ndata` Chebyshev points in the interval `[a,b]`. You can do this in 3 lines using vector notation:

```
k = (1:ndata);  %vector, NOT A LOOP
theta = (vector expression involving k)
xdata = (vector expression involving theta)
```

or you can use a `for` loop. (The vector notation is more compact and runs faster, but is a little harder to understand.)

(b) To check your work, use `cheby_points` to find the Chebyshev points on the interval [-1,1] for `ndata=7`. Do the largest and second largest roots agree with your roots of T_7 computed above?

(c) What are the five Chebyshev points for `ndata=5` and `[a,b]=[-5,5]`?

(d) You should observe that the Chebyshev points are not uniformly distributed on the interval but they are *symmetrical about its center*. It is easy to see from (3) that this fact is true in general.

Exercise 8: Repeat Exercise 4 but with Chebyshev points on [-5,5]. Fill in the table. (Note the extra row!) You should see smaller errors than before, especially for larger `ndata`.

```
     Runge function, Chebyshev points
ndata =  5  Max Error = ________
ndata = 11  Max Error = ________
ndata = 21  Max Error = ________
ndata = 41  Max Error = ________
```

In the following exercise, you will *numerically* examine the hypothesis that the Chebyshev points are the best possible set of interpolation points. You will do this by running many tests, each one using a randomly-generated set of interpolation points. If the theory is correct, none of these tests should result in an error smaller than the error obtained using the Chebyshev points.

Exercise 9:

(a) First, remove or comment out the plotting statements in `test_poly_interpolate` because we will be running it many times.

(b) The Matlab rand function generates a matrix filled with randomly-generated numbers uniformly distributed in the interval $[0, 1]$. What does the following statement do?

```
xdata=[-5,sort(10*(rand(1,19)-.5)),5];
```

(c) If you execute the previous command twice, will you get the same values for `xdata`?

(d) Use the following loop to test a number of different sets of 21 interpolation points. This loop should take much less than a minute.

```
for k=1:500
  xdata=[-5,sort(10*(rand(1,19)-.5)),5];
  err(k)=test_poly_interpolate(@runge,xdata);
end
```

(e) What are the largest and smallest observed values of `err`? How do they compare with the error you found in the previous exercise for 21 Chebyshev points?

Remark It is, in fact, possible for a randomly-generated set of points to yield a smaller error than the Chebyshev points when computed using `test_poly_interpolate`. The Chebyshev points are not necessarily optimal for the Runge example function–they are optimal over the set of all smooth functions. It is also possible because `test_poly_interpolate` computes the error at only 4001 points.

Remark The larger the number of trials, the more rigorous this kind of testing becomes. It is never, of course, a proof, but it can be a way of discovering a counterexample.

We now leave the subject of interpolation using a single polynomial and discuss interpolation using different polynomials on different subintervals. This "piecewise" interpolation is a much better strategy than using single polynomials for most applications.

6 Bracketing

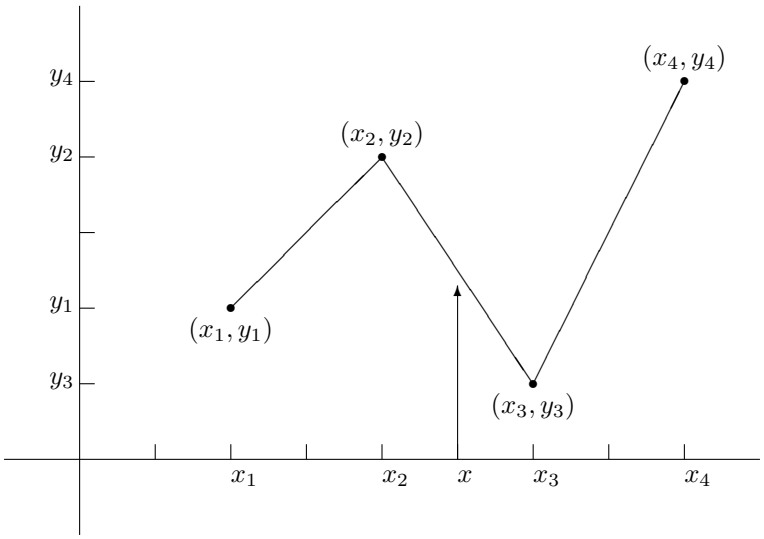
In the next section, we are going to consider interpolation using *piecewise linear* functions, instead of polynomial functions. That means that, in order to evaluate the interpolating function we first must know in which of the intervals (pieces) x lies, and then compute the value of the function depending on the endpoints of the interval. This section addresses the issue of how to find the left and right end points of the subinterval on which x lies.

We will write a *utility routine* to perform this task. We want to get the utility routine right, because we need to be able to rely on it. As usual, it's one of those things that's very easy to describe, and not quite so easy to program.

Suppose we are given a set of N abscissæ, in increasing order

$$x_1 < x_2 < \cdots < x_N$$

with functional values y_n for $n = 1, 2, \ldots, N$ that together define a piecewise linear function $\ell(x)$. In order to evaluate $\ell(x)$ for a given x, you need to know the index, n, so that $x \in [x_n, x_{n+1}]$. This situation is illustrated below, with $N = 4$, and $x \in [x_2, x_3]$. so $n = 2$.



In Matlab notation, this amounts to the following discussion. Denote N by `ndata` and the x_n by the Matlab vector `xdata`. We will assume that the components of `xdata` are strictly increasing. Now suppose that we are given a value `xval` and we are asked to find an integer named `left_index` so that the subinterval `[ xdata(left_index), xdata(left_index+1) ]` contains `xval`. The index `left_index` is that of the left end point of the subinterval, and the index `(left_index+1)` is that of its right end point. (The term *index* in a computer program generally means the same thing as *subscript* in a mathematical expression.)

The following clarifications are needed. We seek an integer value `left_index` so that one of the following cases holds.

Case 1 If `xval` is less than `xdata(1)`, then regard `xval` as in the first interval and `left_index=1`; or

Case 2 If `xval` is greater than or equal to `xdata(ndata)`, then regard `xval` as in the final interval and `left_index=ndata-1` *(read that value carefully!)*; or,

Case 3 If `xval` satisfies `xdata(k)<=xval` and `xval<xdata(k+1)`, then `left_index=k`.

We would like to be able to write a Matlab function to perform this task even when `xval` is a vector of values, but it is not easy to do it efficiently. Instead, in the following exercise, you will write a function `scalar_bracket` to do it for the case `xval` is a scalar, not a vector. In Exercise 11, you will see how it can be written for vectors.

Exercise 10:

In this exercise, we will be writing a Matlab function m-file called `scalar_bracket.m` and test it thoroughly. Later, you will see how to write it for vectors.

(a) Write a function m-file called `scalar_bracket.m` that completes the following skeleton. The "cases" mentioned in the skeleton refer to the list of three cases above.

```
function left_index=scalar_bracket(xdata,xval)
% left_index=scalar_bracket(xdata,xval)
% more comments

% your name and the date

ndata = ??? "number of x data points" ???
```

```
% first check Case 1
if  ??? "condition on xval" ???
  left_index = ???
  return
% then Case 2
elseif ??? "condition on xval" ???
  left_index = ???
  return
% finally Case 3
else
  for k = 1:ndata-1
    if ??? "condition on xval" ???
      left_index = ???
      return
    end
  end
  error('Scalar_bracket: this cannot happen!')
end
```

(b) Now, make up a set of at least five values for `xdata` (don't forget to make them ascending!), and a set of at least seven test values `xval`, and test `scalar_bracket` with the values you have selected, one at a time. Make the tests as comprehensive as you can. Your tests should include:

- One point to the left of all `xdata`.
- One point to the right of all `xdata`.
- One point equal to one of the `xdata` points in the interior.
- One point not equal to any `xdata` but in the interior.

Include your tests in your summary.

Note: The call to the Matlab `error` function may seem superfluous to you because, indeed, it can never be executed. It is a powerful debugging tool, however. If you make an error in the cases, it might turn out that the error message *does* get executed. In that case, you are alerted to the error immediately instead of having an incorrect value of `left_index` cause an obscure error hundreds of statements later. Anything that saves debugging time is worthwhile.

Exercise 11: In this exercise, you will see how the same task can be accomplished using vector statements. The code presented below is written to be as efficient as possible for the case of very large vectors `xval`.

(a) Copy the following code to a file named `bracket.m`.

```
function left_indices=bracket(xdata,xval)
% left_indices=bracket(xdata,xval)
% more comments

% your name and the date

ndata=numel(xdata);

left_indices=zeros(size(xval));
% Case 1
left_indices( find( xval<xdata(2) )  )=1;
% Case 2
```

```
left_indices( find( xdata(ndata-1)<=xval )  )=ndata-1;
% Case 3
for k=2:ndata-2
  left_indices(find( (xdata(k)<=xval) & (xval<xdata(k+1)) ))=k;
end
if any(left_indices==0)
  error('bracket: not all indices set!')
end
```

(b) Suppose that the letters `a,b,c,...,z` represent an ascending set of numbers. If `xval=[b,m,f,g,h,a]`, what would the value of `indices` be as a result of the following statement?

```
indices = find( (g <= xval) & (xval < p) )
```

(c) What would the value of `left_indices` be after the following statements, assuming as above that `xval=[b,m,f,g,h,a]`,

```
left_indices=zeros(size(xval));
left_indices(find( (g<=xval) & (xval<p) ))=4
```

(d) Explain the meaning of the statement beginning "`if any`". Explain why it indicates that "not all indices set."

(e) The loop in `bracket` has `k=2:ndata-2`, but the loop in `scalar_bracket` has `k=1:ndata-1`. Why do the two functions give the same results?

(f) Add a descriptive set of comments to the code.

(g) Use the *same* sets of values for `xdata` and `xval` as you used for `scalar_bracket.m` in the previous exercise. Summarize your results in a table such as this one.

```
xdata       =  ____  ____  ____  ____  ____
xval        =  ____  ____  ____  ____  ____  ____  ____
left_indices
from scalar =  ____  ____  ____  ____  ____  ____  ____
left_indices
from vector =  ____  ____  ____  ____  ____  ____  ____
```

Be sure that you get the same set of values for `left_indices`.

7 Piecewise Linear Interpolation

Now we are ready to consider *piecewise linear interpolation*. The idea is that our interpolating function is not going to be a smooth polynomial defined by a formula. Instead, it will be defined by piecing together linear interpolants that go through each consecutive pair of data points. As such, it will be continouous but not necessarily differentiable. To handle values below the first data point, we'll just extend the first linear interpolant all the way to the left. Similarly, we'll extend the linear function in the last interval all the way to the right.

The graph of a piecewise linear function may not be smooth, but one thing is certain: it will *never* oscillate between the data values. And the interpolation error is probably going to involve the size of the intervals, and the norm of the second derivative of the function, both of which are usually easy to estimate. As the number of points gets bigger, the interval size will get smaller, and the norm of the second derivative won't change, so we have good reason to hope that we can assert a convergence result:

> Given an interval $[a, b]$ and a function $f(x)$ with a continuous second derivative over that interval, and any sequence $\lambda_n(x)$ of linear interpolants to f with the property that h_n, the size of the maximum interval, goes to zero as n increases, then the linear interpolants converge to f both *pointwise*, and *uniformly*.

If C is a bound for the maximum absolute value of the second derivative of the function over the interval, then the pointwise interpolation error and the L^∞ norm are bounded by Ch_n^2, while the L^1 norm is bounded by $(b-a)Ch_n^2$, and L^2 norm is bounded by $(\sqrt{b-a})Ch_n^2$.

Uniform convergence is convergence in the L^∞ norm, and is a much stronger result than pointwise convergence.

Thus, if the only thing you know is that your function has a bounded second derivative on $[a, b]$, then you are guaranteed that the maximum interpolation error you make decreases to zero as you make your intervals smaller.

Stop and think: The convergence result for piecewise linear interpolation is so easy to get. Is it really better than polynomial interpolation? Why did we have such problems with polynomial interpolation before? One reason is that the error result for polynomial interpolation couldn't be turned into a convergence result. Using 10 data points, we had an error estimate in terms of the 10-th derivative. Then using 20 points, we had another error estimate in terms of the 20-th derivative. These two quantities can't be compared easily, and for nonpolynomial functions, successively higher derivatives can easily get successively and endlessly *bigger* in norm. The linear interpolation error bound involved a particular *fixed* derivative and held that fixed, so then it was easy to drive the error to zero. because the other factors in the error term depended only on interval size.

To evaluate a piecewise linear interpolant function at a point `xval`, we need to:

1. Determine the interval `(xdata(left_index),xdata(left_index+1))` containing `xval`;
2. Determine the equation of the line which passes through the points `(xdata(left_index),ydata(left_index))` and `(xdata(left_index+1),ydata(left_index+1))`;
3. Evaluate that line at `xval`.

The `bracket.m` function from the previous exercise does the first of these tasks, but what of the second and third?

Exercise 12: This exercise involves writing a Matlab function m-file called `eval_plin.m` (**eval**uate **p**iecewise **lin**ear interpolation) to do steps 2 and 3 above.

(a) Write down a general formula for the linear function through an arbitrary pair of points.

(b) Write a file called `eval_plin.m` starting with the signature

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

% your name and the date
```

Add appropriate comments.

(c) Add lines that use `bracket.m` to find the vector of indices in `xdata` identifying the intervals that each of the values in `xval` lies. (It is possible to do this in one line of code.)

(d) Add lines evaluating the formula you found in step 1 above on the interval you found in step 3. (It is possible to do this in one line of code.)

(e) Test `eval_plin.m` on the piecewise linear function $f(x) = 3|x| + 1$ on the interval $[-2, 2]$ using the following sequence of commands.

```
xdata=linspace(-1,1,7);
ydata=3*abs(xdata)+1;  % the function y=3*|x|+1
plot(xdata,ydata,'*');
hold on
```

```
xval=linspace(-2,2,4001);
plot(xval,eval_plin(xdata,ydata,xval));
hold off
```

(Note: this test procedure is very similar to the `test_poly_interpolate` function we wrote earlier.) Your plot should pass through the five data points and consist of two straight lines forming a "V". Please send me a copy of your plot.

A piecewise linear function was chosen for testing in the last part of this exercise for both theoretical and practical reasons. In the first place, you are fitting a piecewise linear function to a function that is already piecewise linear and both the original and the fitted functions have breaks at $x = 0$, so the two functions will agree. This makes it easy to see that the interpolation results are correct. In the second place, it is unlikely you will get it right for this piecewise lineear data but have it wrong for other data.

Exercise 13: In this exercise, we will examine convergence of piecewise linear interpolants to the Runge example function we have considered before.

(a) Copy the file `test_poly_interpolate.m` to a file called `test_plin_interpolate.m`. Change the signature to be

```
function max_error=test_plin_interpolate(func,xdata)
% max_error=test_plin_interpolate(func,xdata)
% more comments

% your name and the date
```

and change from using polynomial interpolation to piecewise linear interpolation using `eval_plin`. Do not forget to change to comments to reflect your coding changes.

(b) Restore the `plot` statement, if it is not active.

(c) Consider the same Runge example function we used above. Over the interval [-5,5], use `ndata=5` equally spaced data points, use `test_plin_interpolate` to plot the interpolant and evaluate the interpolation error. Please send me this plot.

(d) Repeat the evaluation for progressively finer meshes and fill in the following table. You do not need to send me the plots that are generated. **Warning:** The final few of these tests should take a few seconds, but if you have written code that happens to be inefficient they can take much longer. You will not be graded on the efficiency of your code.

```
Runge function, Piecewise linear, uniformly-spaced points
 ndata =  5  Max Error(  5) = ________
 ndata = 11  Max Error( 11) = ________
 ndata = 21  Max Error( 21) = ________
 ndata = 41  Max Error( 41) = ________
 ndata = 81  Max Error( 81) = ________
 ndata =161  Max Error(161) = ________
 ndata =321  Max Error(321) = ________
 ndata =641  Max Error(641) = ________
```

(e) Repeat the above convergence study using Chebyshev points instead of uniformly distributed points. You should observe no particular advantage to using Chebyshev points.

```
Runge function, Piecewise linear, Chebyshev points
 ndata =  5  Max Error(  5) = ________
 ndata = 11  Max Error( 11) = ________
```

```
ndata = 21  Max Error( 21) = ________
ndata = 41  Max Error( 41) = ________
ndata = 81  Max Error( 81) = ________
ndata =161  Max Error(161) = ________
ndata =321  Max Error(321) = ________
ndata =641  Max Error(641) = ________
```

The errors in the table decrease as `ndata` increases, as they should. But decreasing error is only half the story: we are interested in the *rate* that the errors decrease. You know from the lectures that, generally, errors tend to be bounded by Ch^p where C is a constant, often related to a derivative of the function being interpolated, h is approximately proportional to the subinterval size (in the uniform case, $h = 1/\text{ndata}$), and p is an integer, often a small integer.

One way to estimate the rate errors decrease is to plot log(error) versus $\log h$. From the general bound, $\log(\text{error}) \approx \log C + p \log h$ for small enough h, so log(error) versus $\log h$ should yield a curve that becomes straight as h becomes small. Furthermore, plotting log(error) versus $\log h$ is the same thing as plotting error versus h as a log-log plot.

It is possible to visually estimate the error from a log-log plot. To do so, simply plot lines with known slope q until you see one that is roughly parallel to the error plot. You will try this in the following exercise. You could also pick points on the plot and estimate the slope of the line directly, but it is easier to use visual comparison.

Another way to estimate the rate of error decrease is to successively halve the subinterval lengths. You may note that the `ndata` values chosen above result in subintervals halving in length. If the error is roughly proportional to Ch^p, halving h should result in reducing the error by $(1/2)^p$. Thus, `|Max Error(321)|/|Max Error(641)|` should be roughly 2^p for some integer p.

Exercise 14: This exercise investigates the behavior of the errors in the table (for uniformly-spaced points) in the previous exercise.

(a) Plot the values of `Max Error` *vs.* h =`(10/ndata)` (h is approximately proportional to interval size) in the first table in Exercise 13 (uniform points) using a log-log plot (the Matlab function `loglog` is used just like `plot`, but results in a log-log plot). Your points should be roughly a straight line on the plot, especially for larger values of `ndata`. Send me a copy of this plot.

(b) Visually estimate the slope of the line in the following manner.

i. Choose a value of C_3 so that the line $y = C_3h^3$ passes through the point `h=10/641`, `y=Max Error(641)`.

ii. With this value of C_3, plot the line $y = C_3h^3$ on the same log-log plot as the error plot above (use `hold on`). If you have computed C_3 correctly, the two plots should pass through the same point at `h=10/641`.

iii. Do the same thing for $y = C_2h^2$.

iv. Do the same thing for $y = C_1h$.

v. Which of the values `p` best approximates the slope of the error curve, `p=3`, `p=2`, or `p=1`? Be sure to send me this plot with your summary file.

(c) Compute the ratios `Max Error(5)/Max Error(11)`, `Max Error(11)/Max Error( 21)`, `Max Error(21)/Max Error( 41)`, *etc.* Do they appear to approach 2^p for some integer p? If so, what is your estimate of the value of p?

Remark: When I compute ratios in order to estimate rates of convergence, I always choose the ratios with the larger number in the numerator and the smaller number in the denominator. That way, the ratios approach integers. I find it much easier to recognize numbers such as 15.5 as being nearly $2^4 = 16$ than to recognize 0.06452 as being nearly $2^{-4} = 0.0625$.

8 Approximating the derivative (Extra credit 8 points)

It may seem likely that if you have a good approximation of a differentiable function then you could differentiate it to get a good approximation of its derivative. On second thought, you have seen how the Runge example results in wiggles when it is approximated by polynomials, so polynomials probably cannot be used to approximate both a function and its derivative.

Standard theorems indicate that a twice-differentiable function can be approximated to $O(h^2)$ by piecewise linear functions (as you saw above in Exercises 13-14), *and* the derivative of the approximation approximates the derivative of the function to $O(h)$. In the following exercise, you will confirm this for the case of the Runge example function.

Exercise 15:

(a) Modify your `eval_plin.m` from Exercise 13 so that it returns *both* `yval` and `y1val`, where `y1val` approximates the derivative of `yval`. (`yval` comes from a piecewise linear expression. `y1val` should come from the derivative of that piecewise linear expression.)

(b) Write a Matlab function `test_plin1_interpolate.m` similar to `test_plin_interpolate.m`, that interpolates the given function (`func`), but plots and measures accuracy of the derivative of the function.

(c) Test `test_plin1_interpolate.m` on the function $f(x) = 3|x| + 1$ on the interval $[-2, 2]$. (Recall this is a test function used in Exercise 12.) Send me the plot. Explain why you believe your result is correct.

(d) Using either the plot method (first two parts of Exercise 14) or the ratio method (third part of Exercise 14), estimate the rate of convergence of the derivative of the interpolant to the derivative of Runge's example function on the interval [-5,5] as the number of points becomes large. You should observe a slower rate of convergence for the derivative of Runge's example function than for Runge's example function itself (in Exercise 13). Explain your methodology.

Last change `$Date: 2016/10/03 00:35:00 $`