MATH2070: LAB 10: Quadrature

1 Introduction

The term "numerical quadrature" refers to the estimation of an area, or, more generally, any integral. (You might hear the term "cubature" referring to estimating a volume in three dimensions, but most people use the term "quadrature.") We might want to integrate some function $f(x)$ or a set of tabulated data. The domain might be a finite or infinite interval, it might be a rectangle or irregular shape, it might be a multi-dimensional volume.

We first discuss the "degree of exactness" (sometimes called the "degree of precision") of a quadrature rule, and its relation to the "order of accuracy." We consider some simple tests to measure the degree of exactness and the order of accuracy of a rule, and then describe (simple versions of) the midpoint and trapezoidal rules. Then we consider the Newton-Cotes and Gauss-Legendre families of rules, and discuss how to get more accurate approximations by either increasing the order of the rule or subdividing the interval. Finally, we will consider a way of adapting the details of the integration method to meet a desired error estimate. In the majority of the exercises below, we will be more interested in the error (difference between calculated and known value) than in the calculated value itself.

A word of caution. We discuss three similar-sounding concepts:

- "Degree of exactness:" the largest value of n so that all polynomials of degree n and below are integrated *exactly.* (Degree of a polynomial is the highest power of x appearing in it.)
- "Order of accuracy:" the value of n so that the error is $O(h^n)$, where h measures the subinterval size.
- "Index:" a number distinguishing one of a collection of rules from another.

These can be related to one another, but are not the same thing.

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

2 Matlab hint

As you recall, Matlab provides the capability of defining "anonymous" functions, using `@` instead of writing m-files to do it. This feature is very convenient when the function to be defined is very simple–a line of code, say–or when you have a function that requires several arguments but you only are interested in varying one of them. You can find out about anonymous functions on on-line reference for `function_handle (@)` Suppose, for example, you want to define a function `sq(x)=x^2`. You could do this by writing the following:

```
  sq=@(x) x.^2;       % define a function using @
```

You could then use `sq(x)` later, just as if you had defined it in an m-file. `sq` is a "function handle" and can be used wherever a function handle is used, such as in a call from another function. Remember, though, that another `@` should not appear before the name. In the next section you will be writing an integration function named `midpoint` that requires a function handle as its first argument. If you wanted to apply it to the integral $\int_0^1 x^2dx$, you might write

```
  q=midpointquad(sq,0,1,11)
```

or you could write it without giving the function a name as

```
  q=midpointquad(@(x) x.^2,0,1,11)
```

There is a nice way to use this form to streamline a sequence of calculations computing the integrals of ever higher degree polynomials in order to find the degree of exactness of a quadrature rule. The following statement

```
  q=midpointquad(@(x) 5*x.^4,0,1,11);1-q
```

first computes $\int_0^1 5x^4dx$ using the midpoint rule, and then prints the error (`=1-q` because the exact answer is 1). You would only have to change `5*x.^4` into `4*x.^3` to check the error in $\int_0^1 4x^3dx$, and you can make the change with judicious use of the arrow and other keyboard keys.

3 Reporting Errors

Errors should be reported in scientific notation (like 1.234e-3, not .0012). You can force Matlab to display numbers in this format using the command `format short e` (or `format long e` for 15 decimal places). This is particularly important if you want to visually estimate ratios of errors.

Computing ratios of errors should *always* be done using full precision, not the value printed on the screen. For example, you might use code like

```
err20=midpointquad(@runge,-5,5,20)-2*atan(5);
err40=midpointquad(@runge,-5,5,40)-2*atan(5);
ratio=err20/err40
```

to get a ratio of errors without loss of accuracy due to reading numbers off the computer screen.

When I compute ratios of this nature, I find it easier to compute them as "larger divided by smaller," yielding ratios larger than 1. It is easier to recognize that 15 is nearly 2^4 (=16) than to recognize that .0667 is nearly 2^{-4} (=0.0625).

4 The Midpoint Method

In general, numerical quadrature involves breaking an interval $[a, b]$ into subintervals, estimating or modelling the function on each subinterval and integrating it there, then adding up the partial integrals.

Perhaps the simplest method of numerical integration is the midpoint method (presented by Quarteroni, Sacco, and Saleri on p. 381). This method is based on interpolation of the integrand $f(x)$ by the constant $f(\frac{a+b}{2})$ and multiplying by the width of the interval. The result is a form of Riemann sum that you probably saw in elementary calculus when you first studied integration.

Break the interval $[a,b]$ into $N-1$ subintervals with endpoints $x_1, x_2, \dots, x_{N-1}, x_N$ (there is one more endpoint than intervals, of course). Then the midpoint rule can be written as

$$\textbf{Midpoint rule} = \sum_{k=1}^{N-1} (x_{k+1} - x_k) f(\frac{x_k + x_{k+1}}{2}). \tag{1}$$

In the exercise that follows, you will be writing a Matlab function to implement the midpoint rule.

Exercise 1:

(a) Write a function m-file called `midpointquad.m` with signature

```
function quad = midpointquad( func, a, b, N)
% quad = midpointquad( func, a, b, N)
% comments

% your name and the date
```

where `f` indicates the name of a function, `a` and `b` are the lower and upper limits of integration, and `N` is the *number of points*, not the number of intervals. The code for your m-file might look like the following:

```
xpts = linspace( ??? ) ;
h = ??? ; % length of subintervals
xmidpts = 0.5 * ( xpts(1:N-1) + xpts(2:N) );
fmidpts = ???
quad = h * sum ( fmidpts );
```

(b) Test your `midpointquad` routine by computing $\int_0^1 2x dx = 1$. Even if you use only one interval (*i.e.* `N=2`) you should get the exact answer because the midpoint rule integrates linear functions exactly.

(c) Use your `midpoint` routine to estimate the integral of our friend, the Runge function, $f(x) = 1/(1+x^2)$, over the interval $[-5,5]$. (If you do not have a copy of the Runge function handy, you can download my version of `runge.m`.) The exact answer is `2*atan(5)`. Fill in the following table, using scientific notation for the error values so you can see the pattern.

```
    N  h       Midpoint Result     Error

   11  1.0    _________________   __________________
  101  0.1    _________________   __________________
 1001  0.01   _________________   __________________
10001  0.001  _________________   __________________
```

(d) Estimate the order of accuracy (an integer power of `h`) by examining the behavior of the error when `h` is divided by 10. (In previous labs, we have estimated such orders by repeatedly doubling the number of subintervals. Here, we multiply by ten. The idea is the same.)

5 Exactness

If a quadrature rule can compute exactly the integral of any polynomial up to some specific degree, we will call this its *degree of exactness*. Thus a rule that can correctly integrate any cubic, but not quartics, has exactness 3. Quarteroni, Sacco, and Saleri mention it on p. 429.

To determine the degree of exactness of a rule, we might look at the approximations of the integrals

$$\begin{aligned}\int_0^1 1dx &= [x]_0^1 = 1\\ \int_0^1 2xdx &= [x^2]_0^1 = 1\\ \int_0^1 3x^2dx &= [x^3]_0^1 = 1\\ &\vdots\\ \int_0^1 (k+1)x^k dx &= [x^{k+1}]_0^1 = 1\end{aligned}$$

Exercise 2:

(a) To study the degree of exactness of the midpoint method, use a single interval (*i.e.* `N = 2`), and estimate the integrals of the test functions over `[0,1]`. The exact answer is 1 each time.

```
 func   Midpoint Result         Error

1       --------------------    --------------------
2 * x   --------------------    --------------------
3 * x^2 --------------------    --------------------
4 * x^3 --------------------    --------------------
```

(b) What is the degree of exactness of the midpoint rule?

(c) Recall that you computed the order of accuracy of the midpoint rule in Exercise 1. For some methods, but not all, the degree of exactness is one less than the order of accuracy. Is that the case for the midpoint rule?

6 The Trapezoid Method

The *trapezoid rule* breaks [a,b] into subintervals, approximates the integral on each subinterval as the product of its width times the average function value, and then adds up all the subinterval results, much like the midpoint rule. The difference is in how the function is approximated. The trapezoid rule can be written as

$$\textbf{Trapezoid rule} = \sum_{k=1}^{N-1}(x_{k+1}-x_k)\frac{f(x_k)+f(x_{k+1})}{2} \tag{2}$$

If you compare the midpoint rule (1) and the trapezoid rule (2), you will see that the midpoint rule takes f at the midpoint of the subinterval and the trapezoid takes the average of f at the endpoints. If each of the subintervals happens to have length h, then the trapezoid rule becomes

$$\frac{h}{2}f(x_1) + \frac{h}{2}f(x_N) + h\sum_{k=2}^{N-1} f(x_k). \tag{3}$$

To apply the trapezoid rule, we need to generate N points and evaluate the function at each of them. Then, apply either (2) or (3) as appropriate.

Exercise 3:

(a) Use your `midpointquad.m` m-file as a model and write a function m-file called `trapezoidquad.m` to evaluate the trapezoid rule. The signature of your m-file should be

```
function quad = trapezoidquad( func, a, b, N )
% quad = trapezoidquad( func, a, b, N )
% more comments

% your name and the date
```

You may use either form of the trapezoid rule.

(b) To test your routine and to study the exactness of the trapezoid rule, use a single interval (`N = 2`), and estimate the integrals of the same test functions used for the midpoint rule over `[0,1]`. The exact answer should be 1 each time.

```
  func   Trapezoid Result       Error

 1       -------------------    -------------------
 2 * x   -------------------    -------------------
 3 * x^2 -------------------    -------------------
 4 * x^3 -------------------    -------------------
```

(c) What is the degree of exactness of the trapezoid rule?

(d) Use the trapezoid method to estimate the integral of the Runge function over $[-5, 5]$, using the given values of `N`, and record the error using scientific notation.

```
     N  h      Trapezoid Result    Error

    11  1.0    -----------------   ------------------
   101  0.1    -----------------   ------------------
  1001  0.01   -----------------   ------------------
 10001  0.001  -----------------   ------------------
```

(e) Estimate the rate at which the error decreases as h decreases. (Find the power of h that best fits the error behavior.) This is the order of accuracy of the rule.

(f) For some methods, but not all, the degree of exactness is one less than the order of accuracy. Is that the case for the trapezoid rule?

7 Singular Integrals

The midpoint and trapezoid rules seem to have the same exactness and about the same accuracy. There is a difference between them, though. Some integrals have perfectly well-defined values even though the integrand has some sort of mild singularity. There are some sophisticated ways to perform these integrals, but there is a simple way that can be adequate for the case that the singularity appears at the endpoint of an interval. Something is lost, however.

Consider the integral

$$I = \int_0^1 \log(x)dx = -1,$$

where log refers to the natural logarithm. Note that the integrand "is infinite" at the left endpoint, so you could not use the trapezoid rule to evaluate it. The midpoint rule, conveniently, does not need the endpoint values.

Exercise 4: Apply the midpoint rule to the above integral, and fill in the following table.

```
    n  h      Midpoint Result     Error

   11  0.1    _________________   __________________
  101  0.01   _________________   __________________
 1001  0.001  _________________   __________________
10001  0.0001 _________________   __________________
```

Estimate the rate of convergence (power of h) as $h \to 0$. You should see that the singularity causes a loss in the rate of convergence.

8 Newton-Cotes Rules

Look at the trapezoid rule for a minute. One way of interpreting that rule is to say that if the function f is roughly linear over the subinterval $[x_k, x_{k+1}]$, then the integral of f is the integral of the linear function that agrees with f (*i.e.*, interpolates f) at the endpoints of the interval. What about trying higher order methods? It turns out that Simpson's rule can be derived by picking triples of points, interpolating the integrand f by a quadratic polynomial, and integrating the quadratic. The trapezoid rule and Simpson's rule are Newton-Cotes rules of index one and index two, respectively. In general, a Newton-Cotes formula uses the idea that if you approximate a function by a polynomial interpolant on uniformly-spaced points in each subinterval, then you can approximate the integral of that function with the integral of the polynomial interpolant. This idea does not always work for derivatives but usually does for integrals. The polynomial interpolant in this case being taken on a uniformly distributed set of points, including the end points. The number of points used in a Newton-Cotes rule is a fundamental parameter, and can be used to characterize the rule. The "index" of a Newton-Cotes rule is commonly defined as one fewer than the number of points it uses, although this common usage is not universal.

We applied the trapezoid rule to an interval by breaking it into subintervals and repeatedly applying a simple formula for the integral on a single subinterval. Similarly, we will be constructing higher-order rules by repeatedly applying Newton-Cotes rules over subintervals. But Newton-Cotes formulæ are not so simple as the trapezoid rule, so we will first write a helper function to apply the rule on a single subinterval.

Over a single interval, all (closed) Newton-Cotes formulæ can be written as

$$\int_a^b f(x)dx \approx Q_N(f) = \sum_{k=1}^{N} w_{k,N} f(x_k)$$

where f is a function and x_k are N **evenly-spaced** points between a and b. The weights $w_{k,N}$ can be computed from the Lagrange interpolation polynomials $\ell_{k,N}$ as

$$w_{k,N} = (b-a)\int_0^1 \ell_{k,N}(\xi)d\xi.$$

(The Lagrange interpolation polynomials arise because we are doing a polynomial interpolation. See Quarteroni, Sacco, and Saleri, p. 387.) The weights do not depend on f, and depend on a and b in a simple manner, so they are often tabulated for the unit interval. In the exercise below, I will provide them to you in the form of a function.

Remark: There are also open Newton-Cotes formulæ that do not require values at endpoints, but there is not time to consider them in this lab.

Exercise 5:

(a) Download `nc_weight.m`.

(b) Write a routine called `nc_single.m` with the signature

```
function quad = nc_single ( func, a, b, N )
% quad = nc_single ( func, a, b, N )
% more comments

% your name and the date
```

There are no subintervals in this case. The coding might look like something like this:

```
xvec = linspace ( a, b, N );
wvec = nc_weight ( N );
fvec = ???
quad = (b-a) * sum(wvec .* fvec);
```

(c) Test your function by showing its exactness is at least 1 for N=2: $\int_0^1 2x dx = 1$ exactly.

(d) Fill in the following table by computing the integrals over [0,1] of the indicated integrands using `nc_single`. (Quarteroni, Sacco, and Saleri, Theorem 9.2) indicates that the degree of exactness is equal to the (N-1) when n is even and the degree of exactness is N when N is odd . Your results should agree, further confirming that your function is correct. (Hint: You can use anonymous functions to simplify your work.)

func	Error N=4	Error N=5	Error N=6
4 * x^3	__________	__________	___________
5 * x^4	__________	__________	___________
6 * x^5	__________	__________	___________
7 * x^6	__________	__________	___________
Degree	___	___	___

The objective of numerical quadrature rules is to *accurately* approximate integrals. We have already seen that polynomial interpolation on uniformly spaced points does not always converge, so it should be no surprise that increasing the order of Newton-Cotes integration might not produce accurate quadratures.

Exercise 6: Attempt to get accurate estimates of the integral of the Runge function over the interval [-5,5]. Recall that the exact answer is `2*atan(5)`. Fill in the following table

n	nc_single Result	Error
3	_________________	__________________
7	_________________	__________________
11	_________________	__________________
15	_________________	__________________

The results of Exercise 6 should have convinced you that you raising N in a Newton-Cotes rule is not the way to get increasing accuracy. One alternative to raising N is breaking the interval into subintervals and using a Newton-Cotes rule on each subinterval. This is the idea of a "composite" rule. In the following exercise you will use `nc_single` as a helper function for a composite Newton-Cotes routine. You will also be using the "partly quadratic" function from Lab 9:

$$f_{\text{partly quadratic}} = \begin{cases} 0 & -1 \leq x < 0 \\ x(1-x) & 0 \leq x \leq 1 \end{cases}$$

whose Matlab implementation is

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

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

Clearly, $\int_{-1}^{1} f_{\text{partly quadratic}}(x)\,dx = \int_0^1 x(1-x)dx = \frac{1}{6}$.

Exercise 7:

(a) Write a function m-file called `nc_quad.m` to perform a composite Newton-Cotes integration. Use the following signature.

```
function quad = nc_quad( func, a, b, N, numSubintervals)
% quad = nc_quad( func, a, b, N, numSubintervals)
% comments

% your name and the date
```

This function will perform these steps: (1) break the interval into `numSubintervals` subintervals; (2) use `nc_single` to integrate over each subinterval; and, (3) add them up.

(b) The most elementary test to make when you write this kind of routine is to check that you get the same answer when `numSubintervals=1` as you would have obtained using `nc_single`. Choose at least one line from the table in Exercise 6 and make sure you get the same result using `nc_quad`.

(c) Test your routine by computing $\int_{-1}^{1} f_{\text{partly quadratic}}(x)\,dx$ using at least `N=3` and `numSubintervals=2`. Explain why your result should have an error of zero or roundoff-sized.

(d) Test your routine by computing $\int_{-1}^{1} f_{\text{partly quadratic}}(x)\,dx$ using at least `N=3` and `numSubintervals=3`. Explain why your result should *not* have an error of zero or roundoff-sized.

(e) Test your routine by checking the following value

```
nc_quad(@runge, -5, 5, 4, 10) = 2.74533025
```

(f) Fill in the following table using the Runge function on [-5,5].

```
Subin-          nc_quad
tervals    N    Error           Err ratio

   10      2    _____________   __________
   20      2    _____________   __________
   40      2    _____________   __________
   80      2    _____________   __________
  160      2    _____________   __________
  320      2    _____________

   10      3    _____________   __________
   20      3    _____________   __________
   40      3    _____________   __________
   80      3    _____________   __________
  160      3    _____________   __________
  320      3    _____________
```

```
    10      4    -------------  ----------
    20      4    -------------  ----------
    40      4    -------------  ----------
    80      4    -------------  ----------
   160      4    -------------  ----------
   320      4    -------------
```

(g) For each index, estimate the order of convergence by taking the sequence of ratios of the error for `num` subintervals divided by the error for `(2*num)` subintervals and guessing the power of two that best approximates the limit of the sequence.

In the previous exercise, the table served to illustrate the behavior of the integration routine. Suppose, on the other hand, that you had an integration routine and you wanted to be sure it had no errors. It is not good enough to just see that you can get "good" answers. In addition, it must converge at the correct rate. Tables such as the previous one are one of the most powerful debugging and verification tools a researcher has.

9 Gauss Quadrature

Like Newton-Cotes quadrature, Gauss-Legendre quadrature interpolates the integrand by a polynomial and integrates the polynomial. Instead of uniformly spaced points, Gauss-Legendre uses optimally-spaced points. Furthermore, Gauss-Legendre converges as degree gets large, unlike Newton-Cotes, as we saw above. Of course, in real applications, one does not use higher and higher degrees of quadrature; instead, one uses more and more subintervals, each with some fixed degree of quadrature.

The disadvantage of Gauss-Legendre quadrature is that there is no easy way to compute the node points and weights. See Quarteroni, Sacco, and Saleri, Section 10.2 and their program `zplege.m` for further information. Tables of values are generally available. We will be using a Matlab function to serve as a table of node points and weights.

One very careful way to compute the node points and weights is described in Algorithm 125 in the collected algorithms from the ACM, appearing in The Communications of the ACM, **5**, no. 10 (October 1962), pp. 510-511, and is available as a Fortran program `TOMS125`.

Normally, Gauss-Legendre quadrature is characterized by the number of integration points. For example, we speak of "three-point" Gauss.

The following two exercises involve writing m-files analogous to `nc_single.m` and `nc_quad.m`.

Exercise 8:

(a) Download the file `gl_weight.m`. This file returns both the node points and weights for Gauss-Legendre quadrature for N points.

(b) Write a routine called `gl_single.m` with the signature

```
function quad = gl_single ( func, a, b, N )
% quad = gl_single ( func, a, b, N )
% comments

% your name and the date
```

As with `nc_single` there are no subintervals in this case. Your coding might look like something like this:

```
[xvec, wvec] = gl_weight ( a, b, N );
fvec = ???
quad = sum( wvec .* fvec );
```

(c) Test your function by showing its exactness is at least 1 for `N=1` and one interval: $\int_0^1 2x dx = 1$ exactly. If the exactness is not at least 1, fix your code now.

(d) Fill in the following table by computing the integrals over [0,1] of the indicated integrands using `gl_single`. Quarteroni, Sacco, and Saleri, Corollary 10.2, shows that the degree of exactness of the method is $2N-1$, and your results should agree, further confirming that your function is correct. (Hint: You can use anonymous functions to simplify your work.)

f	Error	Error
	N=2	N=3
3 * x^2	----------	-----------
4 * x^3	----------	-----------
5 * x^4	----------	-----------
6 * x^5	----------	-----------
7 * x^6	----------	-----------
Degree	---	---

(e) Get accuracy estimates of the integral of the Runge function over the interval [-5,5]. Recall that the exact answer is `2*atan(5)`. Fill in the following table

N	gl_single Result	Error
3	-----------------	------------------
7	-----------------	------------------
11	-----------------	------------------
15	-----------------	------------------

You might be surprised at how much better Gauss-Legendre integration is than Newton-Cotes, using a single interval. There is a similar advantage for composite integration, but it is hard to see for small N. When Gauss-Legendre integration is used in a computer program, it is generally in the form of a composite formulation because it is difficult to compute the weights and integration points accurately for high order Gauss-Legendre integration. The efficiency of Gauss-Legendre integration is compounded in multiple dimensions, and essentially all computer programs that use the finite element method use composite Gauss-Legendre integration rules to compute the coefficient matrices.

Exercise 9:

(a) Write a function m-file called `gl_quad.m` to perform a composite Gauss-Legendre integration. Use the following signature.

```
function quad = gl_quad( f, a, b, N, numSubintervals)
% quad = gl_quad( f, a, b, N, numSubintervals)
% comments

% your name and the date
```

This function will perform two steps: (1) break the interval into `numSubintervals` subintervals; (2) use `gl_single` to integrate over each subinterval; and, (3) add them up.

(b) The most elementary test to make when you write this kind of routine is to check that you get the same answer when `numSubintervals=1` as you would have obtained using `gl_single`. Choose at least one line from the table in the previous exercise (8) and make sure you get the same result using `gl_quad`.

(c) Test your routine by computing $\int_{-1}^{1} f_{\text{partly quadratic}}(x)\,dx$ using `numSubintervals=2` and $N \geq 2$.

(d) Test your routine by computing $\int_{-1}^{1} f_{\text{partly quadratic}}(x)\,dx$ using `numSubintervals=3` and $N \geq 2$.

(e) Test your routine by checking the following value

```
gl_quad(@runge, -5, 5, 4, 10) = 2.7468113
```

(f) Fill in the following table using the Runge function on [-5,5].

Subin-tervals	N	gl_quad Error	Err ratio
10	1	____________	__________
20	1	____________	__________
40	1	____________	__________
80	1	____________	__________
160	1	____________	__________
320	1	____________	
10	2	____________	__________
20	2	____________	__________
40	2	____________	__________
80	2	____________	__________
160	2	____________	__________
320	2	____________	
45	3	____________	__________
90	3	____________	
46	3	____________	__________
92	3	____________	
47	3	____________	__________
94	3	____________	
48	3	____________	__________
96	3	____________	
49	3	____________	__________
98	3	____________	

(g) For indices 1 and 2, estimate the order of convergence by taking the sequence of ratios of the error for `num` subintervals divided by the error for `(2*num)` subintervals and guessing the power of two that best approximates the limit of the sequence.

(h) Estimate the order of accuracy for index 3 by taking the five ratios of errors $r_k = e_k/e_{2k}$ for $k = 45, \ldots, 49$, take their geometric mean $r = (\prod_{k=45}^{49} r_k)^{1/5}$, and guess the power of two that best approximates r. The reason that this case is different from the others is that the errors become near roundoff and averaging is necessary to smooth out the resulting values. Geometric averaging is appropriate because the theoretical error curve is a straight line on a log-log plot and geometrical averaging is the same as arithmetic averaging of logs.

The proofs you have seen about convergence of Gauss quadrature rely on bounds on higher derivatives of the function. When bounds are not available, higher-order convergence might not be observed.

Exercise 10: Consider the integral from Exercise 4

$$I = \int_0^1 \log(x)dx = -1.$$

(a) Use `gl_quad` to fill in the following table.

		log(x)	
Subin-		gl_quad	
tervals	N	Error	Err ratio
10	1	------------	----------
20	1	------------	----------
40	1	------------	----------
80	1	------------	

What is the order of accuracy of the method using `N=1`?

(b) Use `gl_quad` to fill in the following table.

		log(x)	
Subin-		gl_quad	
tervals	N	Error	Err ratio
10	2	------------	----------
20	2	------------	----------
40	2	------------	----------
80	2	------------	

What is the order of accuracy of the method using `N=2`?

(c) Use `gl_quad` to fill in the following table.

		log(x)	
Subin-		gl_quad	
tervals	N	Error	Err ratio
10	3	------------	----------
20	3	------------	----------
40	3	------------	----------
80	3	------------	

What is the order of accuracy of the method using `N=3`?

It is instructive to see how to compute integrals over infinite intervals. Basically, the best way to do that is to make a change of variables to make the interval finite. There are other ways, such as multiplying the integrand by a weighting function and then using an integration method based on weighted integrals, but you will see how a change of variables works in thee following exercise.

Exercise 11: Consider the integral

$$\int_0^\infty \frac{1}{1+x^2}dx = \frac{\pi}{2}.$$

Making a change of variables $u = 1/(1+x)$ or $x = (1-u)/u$ yields the integral

$$\int_0^1 \frac{1}{u^2+(1-u)^2}du.$$

Use any of the integration functions to evaluate this integral to an accuracy of $\pm 1.e-8$. Explain why you chose the method you used and how you determined the number of intervals necessary to achieve the specified accuracy.

Remark: There is no easy way to tell in advance which method to use to achieve a particular accuracy. You can pick a method by trial-and-error, switching methods when you cannot achieve the desired error in the time you are willing to wait. Of course, once you have some trial values, you can use theoretical rates of convergence to help you decide the number of subintervals necessary to reach your desired accuracy.

In the following section, you will see how accuracy might be improved during the integration process itself, so that a target accuracy can be achieved.

10 Adaptive quadrature

Our final task will consider "adaptive quadrature." Adaptive quadrature employs *non-uniform* division of the interval of integration into subintervals of non-equal length. It uses smaller subintervals where the integrand is changing rapidly and larger subintervals where it is flatter. The advantage of this approach is that it minimizes the work necessary to compute a given integral. Adaptive quadrature is discussed by Quarteroni, Sacco, and Saleri on page 402. In this section, you will investigate a recursive algorithm for adaptively computing quadratures.

Numerical integration is used often with integrands that are very complicated and take a long time to compute. Although this section will use simple integrands for illustrative purposes, you should think of each evaluation of the integrand ("function call") to take a long time. Thus, the objective is to reach a given accuracy with a minimum number of function calls. A good strategy for achieving a specified accuracy efficiently is to attempt to uniformly distribute the error over each subinterval. If no interval is particularly bad, there is no obvious place to improve the estimate and if no interval is particularly good, no work has been wasted.

Recall that, if an integration method has degree of exactness p, then the local error on an integration interval of length h satisfies an expression involving a constant C and a point located somewhere in the interval. Assuming the derivative of the function is roughly constant in an interval, then C can be estimated by dividing the interval into two subintervals of length $h/2$ each, estimating the error in the interval as the sum of the errors on the two subintervals, and then equating the two expressions. Denote by Q_h the integral over the interval of length h, then

$$Q_h = Q + Ch^{p+2}f^{(p+1)}(\xi) + O(h^{p+3})$$

and $Q^L_{h/2}$ and $Q^R_{h/2}$ the two estimates of the integral on the left and right subintervals are

$$Q^L_{h/2} + Q^R_{h/2} = Q + C(h/2)^{p+2}(f^{(p+1)}(\xi^L) + f^{(p+1)}(\xi^R)) + O(h^{p+3})$$

where C is a constant, and ξ, ξ^L, and ξ^R are appropriately chosen. Assuming that $f^{(p+1)}$ is roughly constant, $f^{(p+1)}(\xi) = f^{(p+1)}(\xi^L) = f^{(p+1)}(\xi^R) = f^{(p+1)}$, and assuming that the higher order terms can be neglected yields

$$Q_h = Q + Ch^{p+2}f^{(p+1)} \tag{4}$$

$$Q^L_{h/2} + Q^R_{h/2} = Q + C(h/2)^{p+2}(2f^{(p+1)}). \tag{5}$$

Eliminating $Ch^{p+2}f^{p+1}$ from the system (4)-(5) and defining the error as $|Q^L_{h/2} + Q^R_{h/2} - Q|$ yields the expression

$$\text{error estimate} = \frac{|Q^L_{h/2} + Q^R_{h/2} - Q_h|}{2^{p+1} - 1} \tag{6}$$

Supposing that the error estimate is small enough, should the value (4) or (5) be used for Q? In principle, either one will do, but it is clear that the error term in (5) is smaller (by a factor of $1/2^{p+1}$) than the error term in (4), so Q should be estimated from the two half-interval integrals.

The basic structure of one simple adaptive algorithm depends on using the error estimate (6) over each integration subinterval. If the error is acceptably small over each interval, the process stops, and, if not, continues recursively. In the following exercise, you will write a *recursive* function to implement this procedure.

Exercise 12:

(a) Create a function m-file named `adaptquad.m` with the following code, and fill in the places marked "???".

```
function [Q,errEst,x,recursions]= ...
         adaptquad(func,x0,x1,tol,recursions)
% [Q,errEst,x,recursions]=
%        adaptquad(func,x0,x1,tol,recursions)
% adaptive quadrature
%     input parameters
% func       = function to integrate
% x0         = left end point
% x1         = right end point
% tol        = desired accuracy
% recursions = number of allowable recursions left
%
%     output parameters
% Q          = estimate of the value of the integral
% errEst     = estimate of error in Q
% x          = all intermediate integration points
% recursions = minimum number of recursions remaining
%              after convergence

% Add a mid-point and re-estimate integral
xmid=(x0+x1)/2;

% Qleft and Qright are integrals over two halves
N=3;
Qboth=gl_single(func,x0,x1,N);
Qleft=gl_single(func,x0,xmid,N);
Qright=gl_single( ??? );

% p=degree of exactness of Gauss-Legendre
p=2*N-1;
errEst= ??? ;

if errEst<tol | recursions<=0  %vertical bar means "or"
  % either ran out of recursions or converged
  Q= ??? ;
  x=[x0 xmid x1];
else
  % not converged -- do it again
  [Qleft,estLeft,xleft,recursLeft]=adaptquad(func, ...
               x0,xmid,tol/2,recursions-1);
  [Qright,estRight,xright,recursRight]=adaptquad(func, ...
                ??? );
  % recursive work is all done, return answers
  % don't want xmid to appear twice in x
  x=[xleft xright(2:length(xright))];
  Q= ??? ;
  errEst= ??? ;
  recursions=min(recursLeft,recursRight);
end
```

Note: The input and output parameter `recursions` is not theoretically necessary, but is used to guard against infinite recursion. Since there are a fixed number of function calls per recursion, it also counts the number of function calls. For complicated functions, total running time will be proportional to `recursions`. The output vector `x` is not necessary, either, but will be used to show the effect of the adaptation.

(b) Test `adaptquad` by applying it to the polynomial $f_5(x) = 6x^5$ on the interval $[0, 1]$, using `tol=1.e-5` and `recursions=50`. Because Gauss-Legendre integration of index 3 is exact for this polynomial, the integral should equal 1 (*i.e.* error should be zero or roundoff), and `recursions=50`, and the values of `x` should be three equally-spaced points in the interval.

(c) Test `adaptquad` by applying it to the polynomial $f_6(x) = 7x^6$ on the interval $[0, 1]$, using `tol=1.e-5` and `recursions=50`. Because Gauss-Legendre integration of index 3 is *not* exact for this polynomial, the integral should be close to 1. It turns out that a single set of refinements is performed, so `recursions=49`, and the values of `x` should be five equally-spaced points in the interval. The estimated and true errors should agree to at least 3 significant digits. This excellent agreement is because there are no "higher order terms" in the expressions (4) and (5) and so (6) is almost exact. Please include both the estimated and true errors in your summary.

(d) Test `adaptquad` by applying it to the Runge function on the interval [-5,5]. Use `recursions=50`. Recall that the exact answer is `2*atan(5)`. Fill in the following table

```
 adaptquad for Runge
tol    est. error   exact error
1.e-3  __________   __________
1.e-6  __________   __________
1.e-9  __________   __________
```

You should find that the estimated and exact errors are close in size, and smaller than `tol`. For the two more accurate cases, the estimated error is slightly larger than the exact error. As you can see, the estimated error is not so good for the case that `tol=1.e-3`.

Exercise 13: Consider the following situation.

- A quadrature is being attempted with the call

  `[Q,estErr,x,recursions]=adaptquad(@funct,0,1,tol,50);`

- The estimated error is larger than tol at first, so new calls with `tol/2` are made for the intervals $I_{\text{left}} = [0, 0.5]$ and $I_{\text{right}} = [0.5, 1]$.
- Assume that the call for the interval I_{left} satisfies the convergence criterion.
- Assume that the call for the interval I_{right} does not satisfy the convergence criterion, thus requiring two more calls to adaptquad. Assume that each of these calls satisfies the convergence criterion.

What are the final values of `x` and `recursions` after the `adaptquad` function has completed its work? Explain your reasoning.

The next few exercises will help you look a little more closely at the results of this recursive adaptive algorithm. Some of the points that will be made are listed below.

- You will see the advantage of adaptive algorithms. They save work over fixed algorithms such as `gl_quad`.
- You will see the pattern of the integration points take. They can be distributed in a very nonuniform fashion.
- You will see what happens when you try "harder" integrands.
- You will see some of the weaknesses of the algorithm.

In the following exercises, you will examine two functions that are more difficult to integrate. The first is a scaled version of the Runge function, $1/(a^2 + x^2)$, where $a = 10^{-3}$ The value of the integral on [-1,1] of the scaled Runge function is

$$\int_{-1}^{1} \frac{1}{a^2 + x^2} dx = \frac{2}{a} \tan^{-1} \frac{1}{a}.$$

The scaled Runge function has a peak value of $1/a^2 = 10^6$, so is much more strongly peaked than the unscaled Runge function.

The second function is $\sqrt{|x - 0.5|}$, and the value of its integral over the interval [-1,1] is

$$\int_{-1}^{1} \sqrt{\left|x - \frac{1}{2}\right|} dx = \frac{\sqrt{2}}{6} + \frac{\sqrt{6}}{2}.$$

This function has a singularity in its derivatives at $x = 0.5$, thus invalidating the proof that the error estimator is reliable.

A third function that is difficult to integrate is $x^{-0.99}$. This function has an integrable singularity at $x = 0$ that is close to being nonintegrable.

Exercise 14:

(a) Write a function m-file named `srunge.m` for the scaled Runge function $f(x) = 1/(a^2 + x^2)$ with $a = 10^{-3}$.

(b) Evaluate the integral of `srunge` on the interval [-1,1] using the following call.

```
[Q,estErr,x,recursions]=adaptquad(@srunge,-1,1,1.e-10,50);
```

What are the estimated and true errors? Is `recursions` larger than zero?

(c) Use `gl_quad` with `index=3` on the scaled Runge function. Use trial-and-error to find the number of subintervals required to achieve a true error from `gl_quad` that is roughly comparable to the true error from `adaptquad`. How does this compare with `length(x)-1`, the number of subintervals that `adaptquad` used?

(d) Plot the sizes of the subintervals that `adaptquad` used with the following command.

```
xave=(x(2:end)+x(1:end-1))/2;
dx= x(2:end)-x(1:end-1);
semilogy(xave,dx,'*')
```

A semilog plot is appropriate here because of the wide range of interval sizes. Please include this plot with your summary.

(e) What are the lengths of the largest and smallest intervals? Explain (one sentence) where you might expect to find the smallest intervals for an arbitrary function.

Exercise 15:

(a) Approximate the integral of the function $f(x) = \sqrt{|x - 0.5|}$ over the interval [-1,1] to a tolerance of `1.e-10`. The exact value of this integral is $\sqrt{2}/6 + \sqrt{6}/2$. What are the estimated and true errors?

(b) What is the returned value of `recursions`? It should be positive, indicating that the subinterval convergence criterion was always reached successfully.

(c) Plot the subinterval sizes using the following command

```
xave=(x(2:end)+x(1:end-1))/2;
dx= x(2:end)-x(1:end-1);
semilogy(xave,dx,'*')
```

where `x` is replaced by the variable name you used. A semilog plot is appropriate here because of the wide range of interval sizes. Please include this plot with your summary.

In the following exercise you will apply `adaptquad` to a function that is almost not integrable. You will see the benefit of the `recursions` variable, whose reduction to 0 indicates that convergence was not achieved.

Exercise 16:

(a) Use `adaptquad` to approximate the integral

$$\int_0^1 x^{-0.99} dx = 100$$

to a tolerance of `1.e-10`. Use `recursions=50`. *Notice that the integration interval is [0,1].* What is the computed value of the integral? What are the estimated error and the true error? What is the value returned for `recursions`?

(b) Do the same approximation starting with `recursions=60`. What are the computed value of the integral, the estimated error, the true error, and the returned value of `recursions`? You will notice essentially no improvement. (You may have to use the command `set(0,'RecursionLimit',200)` in order that Matlab will allow more recursions.)

(c) If the variable `recursions` were not used, this recursive function would "never" terminate because the convergence test would never be satisfied. (Actually, it would abort because there is a practical limit on the recursion depth.) The reason is that in the presence of the singularity, halving the interval does result in a reduction of error, but the reduction is half or less (*c.f.* Exercise 10), so it never passes the convergence test.

11 Extra credit: Integration by Monte Carlo methods (10 points)

There is another approach to approximating integrals, one that can be used even when the integrand is not smooth or piecewise smooth, when the integral is taken over a region $\Omega \in \mathbb{R}^n$ that is not easily characterized or when its dimension, n is high. This versatility comes at a cost: the method is probabalistic and also slowly convergent. It is called the "Monte Carlo" method.

The basic idea behind the Monte Carlo method is that the formula for computing the average value of a function over a region

$$\langle f \rangle = \frac{1}{|\Omega|} \int_\Omega f$$

can be used to compute the integral on the right side if the average on the left side is known.

The Monte Carlo method is discussed in many places on the web. You can find a very clear description, with considerable detail at
`http://farside.ph.utexas.edu/teaching/329/lectures/node109.html`.

Eric Weisstein has an article in MathWorld,
`http://mathworld.wolfram.com/MonteCarloIntegration.html` that essentially states, for $\Omega \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$

$$\begin{aligned} \int_\Omega f(\mathbf{x}) dx_1 \dots dx_n &= Q + \epsilon \\ Q &\approx |\Omega| \langle f \rangle \qquad (7) \\ \epsilon &\approx |\Omega| \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \qquad (8) \end{aligned}$$

where the angle brackets denote an average taken over randomly-chosen points $\{\mathbf{x}_k\}_{k=1}^N \Omega$,

$$\begin{aligned} \langle f \rangle &= \tfrac{1}{N} \textstyle\sum_{k=1}^N f(\mathbf{x}_k), \text{ and} \\ \langle f^2 \rangle &= \tfrac{1}{N} \textstyle\sum_{k=1}^N f(\mathbf{x}_k)^2 \end{aligned} \qquad (9)$$

and $|\Omega|$ denotes the volume of Ω.

As an example, consider the problem of computing the area of the unit ball in $\mathbb{R}^2$. In this case, f is simply the characteristic function of the ball

$$\phi(x_1, x_2) = \begin{cases} 1 & x_1^2 + x_2^2 \le 1 \\ 0 & \text{otherwise} \end{cases}$$

and the area of the ball can be computed as $\int_{-1}^1 \int_{-1}^1 \phi(x_1, x_2)\, dx_1\, dx_2$. In this case, Ω is the unit square $[-1, 1] \times [-1, 1]$.

The following code, using `x` and `y` to denote x_1 and x_2 will estimate the area of the ball.

Remark: Computing the area of a figure is a simpler problem than computing the integral of a function, but the essential steps are all here.

```
CHUNK=10000;  % chosen for efficiency
NUM_CHUNKS=100;

VOLUME=4;  % outer square is 2 X 2

totalPoints=0;
insidePoints=0;
for k=1:NUM_CHUNKS
  x=(2*rand(CHUNK,1)-1);
  y=(2*rand(CHUNK,1)-1);
  phi=( (x.^2+y.^2) <= 1 ); % 1 for "true" and 0 for "false"
                            % just like characteristic function
  insidePoints=insidePoints+sum(phi);
  totalPoints=totalPoints+CHUNK;
end
average=insidePoints/totalPoints;
a=VOLUME*average;
disp(strcat('approx area=',num2str(a), ...
           ' with true error=',num2str(pi-a), ...
           ' and estimated error=', ...
     num2str(VOLUME*sqrt((average-average^2)/totalPoints))));
```

Remark: This code uses a programming trick. In Matlab, the value 0 represents “false” and the value 1 represents “true”. As a consequence, the characteristic function of the unit ball can easily be calculated by using a logical expression describing the interior of the ball.
Remark: Note that the points are divided into chunks of 10,000 points each. Then the function evaluation is done using vectors whose lengths are the chunk size. Working with these long vectors, Matlab will do the calculations efficiently. You don’t, however, want to do all 1,000,000 points at once, because such large vectors can take a while just for the memory to be allocated. That is why the problem is first divided into chunks and then repeated a number of times.

Exercise 17:

(a) Copy the above code to a script m-file and execute it several times. You should observe that the estimated area changes slightly each time and that the estimated error is usually, but not always, larger than the actual error. (These are probabalistic quantities, after all.) Explain why both $\langle f\rangle$ and $\langle f^2\rangle$ are given the same value (`average`) in computing the error.

(b) It is possible to integrate a function by embedding its graph into a rectangle, or other simple shape, and repeat the above approach to get the integral. A better approach is to use (9) and (7) to estimate the integral. Use (7) to estimate the following integral

$$\int_0^2 e^x dx$$

along with both the true error and error estimated from (8). Use enough trials to achieve an estimated accuracy of ± 0.001 or smaller, and include the number of trials in your summary. This function is easily integrated exactly. What is the true error of your estimate? It should be smaller, or at least not much bigger, than your error estimate.

(c) Computing the volume of the intersection of two unit cylinders with orthogonal axes in $\mathbb{R}^3$ is an exercise often given to calculus students because it is difficult, but not impossible, to compute using elementary calculus. This volume is called a Steinmetz Solid and more information can be found on the web at `http://mathworld.wolfram.com/SteinmetzSolid.html`. This volume is known to be 16/3. Use Monte Carlo integration to estimate this volume along with both the true and estimated errors. Use enough trials to achieve an estimated accuracy of ± 0.001 or smaller. Please include the number of trials in your summary.

(d) Compute the integral of $2e^{x_1+x_2+x_3}$ over the volume of the intersection of two unit cylinders along with the estimated error. Use enough trials to achieve an estimated accuracy of ± 0.005 or smaller. Please include the number off trials in your summary. **Hint:** Use (9) and (7) to estimate the integral, use the script you wrote for the volume of the Steinmetz solid to choose points at which to do averaging.

Last change `$Date: 2016/11/01 16:58:09 $`