M382 – PROJECT

PROJECT 2
Due date: June 19 , 2022

During submission process (via email to : smerdan@metu.edu.tr) I want you submit

1. **Diary file**
2. **Summary file (there is an example in the attached file. It can be text file or doc files)**
3. **Submit m-files and jpeg files .**

When you submit your project, I want you to write all of group's members names who contribute it.

NOTE: THERE ARE SOME M. FILES YOU CAN USE IT. YOU CAN FIND THEM IN THE FOLDER.

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.

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 $\int_0^1 x^2 dx$, 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 $\int_0^1 5x^4 dx$ 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 $\int_0^1 4x^3 dx$, and you can make the change with judicious use of the arrow and other keyboard keys.

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).

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

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

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

Exercise 1:

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

```
2. function quad = midpointquad( func, a, b, N)
3. % quad = midpointquad( func, a, b, N)
4. % comments
5.
6. % 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 );
```

2. 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.
3. 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 `runge.m`. in the file.) 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.

4.	N	h	Midpoint Result	Error
5.				
6.	11	1.0	____________________	_____________________
7.	101	0.1	____________________	_____________________
8.	1001	0.01	____________________	_____________________
9.	10001	0.001	____________________	_____________________

10. 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.)

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

$$\int_0^1 1 dx = [x]_0^1 = 1$$

$$\int_0^1 2x dx = [x^2]_0^1 = 1$$

$$\int_0^1 3x^2 dx = [x^3]_0^1 = 1$$

$$\vdots$$

$$\int_0^1 (k+1)x^k dx = [x^{k+1}]_0^1 = 1$$

Exercise 2:

1. 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.

```
2.     func    Midpoint Result         Error
3.
4.    1        ___________________     ___________________
5.    2 * x    ___________________     ___________________
6.    3 * x^2  ___________________     ___________________
7.    4 * x^3  ___________________     ___________________
```

8. What is the degree of exactness of the midpoint rule?
9. 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?

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:

1. 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

```
2. function quad = trapezoidquad( func, a, b, N )
3. % quad = trapezoidquad( func, a, b, N )
4. % more comments
5.
6. % your name and the date
```

You may use either form of the trapezoid rule.

7. 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.

```
8.    func   Trapezoid Result       Error
9.
10.     1       ___________________
     ___________________
11.     2 * x   ___________________
     ___________________
12.     3 * x^2 ___________________
     ___________________
13.     4 * x^3 ___________________
     ___________________
```

14. What is the degree of exactness of the trapezoid rule?
15. 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.

```
16.       N  h     Trapezoid Result    Error
17.
18.      11  1.0   _________________
     ___________________
19.     101  0.1   _________________
     ___________________
20.    1001  0.01  _________________
     ___________________
21.   10001  0.001 _________________
     ___________________
```

22. 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.
23. 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?

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 $h \rightarrow 0$. You should see that the singularity causes a loss in the rate of convergence.

.

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.) 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, which we will not consider.

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:

1. Download `nc_weight.m`.
2. Write a routine called `nc_single.m` with the signature

```
3. function quad = nc_single ( func, a, b, N )
4. % quad = nc_single ( func, a, b, N )
5. % more comments
6.
7. % 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);
```

8. Test your function by showing its exactness is at least 1 for N=2: $\int_0^1 2x dx = 1$ exactly.
9. 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:

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

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);
```

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

Clearly,

Exercise 7:

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

```
2. function quad = nc_quad( func, a, b, N,
   numSubintervals)
3. % quad = nc_quad( func, a, b, N, numSubintervals)
4. % comments
5.
6. % 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.

7. 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`.
8. 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.
9. 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.
10. Test your routine by checking the following value

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

12. Fill in the following table using the Runge function on [-5,5].

```
13.  Subin-          nc_quad
14.  tervals    N    Error           Err ratio
15.
16.      10     2    _____________   __________
17.      20     2    _____________   __________
18.      40     2    _____________   __________
19.      80     2    _____________   __________
20.     160     2    _____________   __________
21.     320     2    _____________
22.
23.      10     3    _____________   __________
24.      20     3    _____________   __________
25.      40     3    _____________   __________
26.      80     3    _____________   __________
27.     160     3    _____________   __________
28.     320     3    _____________
29.
30.      10     4    _____________   __________
31.      20     4    _____________   __________
```

```
32.      40     4    ______________   ___________
33.      80     4    ______________   ___________
34.     160     4    ______________   ___________
35.     320     4    ______________
```

36. 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.

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. Tables of values are generally available. We will be using a Matlab function to serve as a table of node points and weights.

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:

1. Download the file `gl_weight.m`. This file returns both the node points and weights for Gauss-Legendre quadrature for N points.
2. Write a routine called `gl_single.m` with the signature

```
3. function quad = gl_single ( func, a, b, N )
4. % quad = gl_single ( func, a, b, N )
5. % comments
6.
7. % 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 );
```

8. 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.
9. Fill in the following table by computing the integrals over [0,1] of the indicated integrands using `gl_single`. It is known 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.)

10.	f	Error	Error
11.		N=2	N=3
12.	3 * x^2		
13.	4 * x^3		
14.	5 * x^4		
15.	6 * x^5		
16.	7 * x^6		
17.	Degree		

18. 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

19.	N	gl_single Result	Error
20.			
21.	3		
22.	7		
23.	11		
24.	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.