Assignment 4

20180282 Jimin Park

November 7, 2023

1. (a) We can implement composite trapezoidal rule as below code.

```
syms s;
f(s) = s * sqrt(1 - s);

% original intergation value
int_val = double(int(f, s, 0, 1));
fprintf("intergration value is %.4f\n", int_val);

% Composite Trapezoidal rule
n = 600;  % note points are x_0, ..., x_n
h = 1 / n;

x_0 = f(0) + f(1);
x_1 = 0;    % summation of f(x_j)

for i = 1:n-1
    x = h * i;
    x_1 = x_1 + f(x);
end

approx_int_val = double (h * (x_0 + 2 * x_1) / 2);

fprintf("approximated value is %.4f\n", approx_int_val);
```

If we run this code, we get the below result:

```
intergration value is 0.2667
approximated value is 0.2667
```

(b) We can implement composite Simpson's rule as below code.

```
syms s;
f(s) = s * sqrt(1 - s);

% original intergation value
int_val = double(int(f, s, 0, 1));
fprintf("intergration value is %.4f\n", int_val);

% Composite Trapezoidal rule
n = 300;  % note points are x_0, ..., x_n
h = 1 / n;

```

```
x_0 = f(0) + f(1);
x_1 = 0;     % summation of f(x_{2j-1})
x_2 = 0;     % summation of f(x_{2j})

for i = 1:n-1
    x = h * i;
    if rem(i, 2) == 0
        x_2 = x_2 + f(x);
    else
        x_1 = x_1 + f(x);
    end
end

approx_int_val = double (h * (x_0 + 2*x_2 + 4*x_1) / 3);
fprintf("approximated value is %.4f\n", approx_int_val);
```

If we run this code, we get the below result:

```
intergration value is 0.2667
approximated value is 0.2667
```

(c) Let's compare these two methods in terms of two perspectives.

i. Cost perspective.

Let's consider we use the algorithm as above codes to implement composite quadrature rules. Assume that we consider n (which is even) subintervals. Then composite trapezoidal rule performs $n+1$ additions, $n+1$ multiplications, and 0 if operations (see 12-20 lines of 1(a)), which implies that it performs totally $2n+2$ operations. However, composite Simpson's rule performs $n+2$ additions, $n+2$ multiplications, and $n-1$ if operations (see 12-25 lines of 1(b)), which implies that it performs totally $3n+2$ operations. Hence, even if we use same number of subintervals, (in here, this value is n), composite Simpson's rule takes more cost to compute value. Of course, composite Simpson's rule is more accurate than composite trapezoidal rule because the error term of composite Simpson's rule is $O(h^4)$ which is smaller than the error term of composite trapezoidal rule, $O(h^2)$.

ii. Efficiency perspective.

If we need less accurate approxmiation, than the composite Trapezoidal rule could be more efficient than the composite Simpson't rule. But in general case, the composite Simpson't rule is more efficient than the composite trapezoidal rule. It is because even if the composite Trapezoidal rule perfomrs better in terms of the number of operations, $O(n)$, the composite Simpson's rule performs much more better in terms of convergence, $O(h^4) \ll O(h^2)$.

2. Below code is the implementation of the adaptive Simpson's rule for the intergral in Problem 1:

```
syms s;
f(s) = s * sqrt(1 - s);

% original intergation value
int_val = double(int(f, s, 0, 1));
fprintf("intergration value is %.4f\n", int_val);

% Composite adaptive Simpson's rule
TOL = 10^(-7);
```

```
N = 50; % limit of levels

% Step 1
approx_int_val = 0;
i = 1;
TOLs(i) = 10 * TOL;
a(i) = 0;
h(i) = 1/2;
FA(i) = f(0);
FC(i) = f(h(i));
FB(i) = f(1);
S(i) = h(i)*(FA(i)+4*FC(i)+FB(i))/3; % Approx from Simpson's
    method
L(i) = 1;   % level

% Step 2
while i > 0
    % Step 3
    FD = f(a(i)+h(i)/2);
    FE = f(a(i)+3*h(i)/2);
    S1 = h(i)*(FA(i)+4*FD+FC(i))/6; % Approx for left
        subinterval
    S2 = h(i)*(FC(i)+4*FE+FB(i))/6;
    % save data at this level
    v1 = a(i);
    v2 = FA(i);
    v3 = FC(i);
    v4 = FB(i);
    v5 = h(i);
    v6 = TOLs(i);
    v7 = S(i);
    v8 = L(i);

    % Step 4
    i = i - 1;  % Delete the level

    % Step 5
    if abs(S1+S2-v7) < v6
        approx_int_val = approx_int_val + S1 + S2;
    else
        if v8 >= N  % level exceeds
            fprintf("LEVEL EXCEEDED");  % procedure fails
            return;
        else    % add one level
            % data for right-half subinterval
            i = i + 1;
            a(i) = v1 + v5;
            FA(i) = v3;
            FC(i) = FE;
            FB(i) = v4;
            h(i) = v5/2;
            TOLs(i) = v6/2;
            S(i) = S2;
```

```
            L(i) = v8 + 1;
            % data for left-half subinterval
            i = i + 1;
            a(i) = v1;
            FA(i) = v2;
            FC(i) = FD;
            FB(i) = v3;
            h(i) = h(i-1);
            TOLs(i) = TOLs(i-1);
            S(i) = S1;
            L(i) = L(i-1);
        end
    end
end

approx_int_val = double(approx_int_val);
fprintf("approximated value is %.4f\n", approx_int_val);
```

If we run this code, we get the below result:

```
intergration value is 0.2667
approximated value is 0.2667
```

3. (a) Below code is the implementation of gaussian quadrature with increasing order.

```
format long;

syms s;
f(s) = cos(s)^2;

% original intergation value
int_val = double(int(f, s, 0, pi/4));
fprintf("intergration value is %.12f\n", int_val);

% Gaussian quadrature
g(s) = f((pi/4*s+pi/4)/2)*(pi/4)/2;

% n=2
x(2,1) = 0.5773502692;
x(2,2) = -x(2,1);
c(2,1) = 1;
c(2,2) = 1;

% n=3
x(3,1) = 0.7745966692;
x(3,2) = 0;
x(3,3) = -x(3,1);
c(3,1) = 0.5555555556;
c(3,2) = 0.8888888889;
c(3,3) = c(3,1);

% n=4
x(4,1) = 0.8611363116;
x(4,2) = 0.3399810436;
```

```
x(4,3) = -x(4,2);
x(4,4) = -x(4,1);
c(4,1) = 0.3478548451;
c(4,2) = 0.6521451549;
c(4,3) = c(4,2);
c(4,4) = c(4,1);

% n=5
x(5,1) = 0.9061798459;
x(5,2) = 0.5384693101;
x(5,3) = 0;
x(5,4) = -x(5,2);
x(5,5) = -x(5,1);
c(5,1) = 0.2369268850;
c(5,2) = 0.4786286705;
c(5,3) = 0.5688888889;
c(5,4) = c(5,2);
c(5,5) = c(5,1);

approx_int_vals = zeros(1, 5);
for n = 2:5
    for i = 1:n
        approx_int_vals(n) = approx_int_vals(n) + c(n,i)*g(x(n,
           i));
    end
end

for n = 2:5
    fprintf("approximated value is %.12f with order n=%d\n",
       ...
        approx_int_vals(n), n)
end
```

If we run this code, we get the below result:

```
intergration value is 0.642699081699
approximated value is 0.642317235049 with order n=2
approximated value is 0.642701112122 with order n=3
approximated value is 0.642699075999 with order n=4
approximated value is 0.642699081680 with order n=5
```

(b) Below code is the implementation of composite gaussian quadrature with increasing number of subintervals.

```
format long;

syms s;
f(s) = cos(s)^2;

% original intergation value
int_val = double(int(f, s, 0, pi/4));
fprintf("intergration value is %.12f\n", int_val);

% composite gaussian quadrature
```

```
x(1) = 0.7745966692;
x(2) = 0;
x(3) = -x(1);
c(1) = 0.5555555556;
c(2) = 0.8888888889;
c(3) = c(1);

n = 6;
for k = 3:n
    X = [linspace(0, pi/4, k)];
    approx_int_val = 0;
    for j = 1:k-1
        g(s) = f(((X(j+1)-X(j))*s+(X(j+1)+X(j)))/2)*(X(j+1)-X(j
            ))/2;
        for i = 1:3
            approx_int_val = approx_int_val + c(i)*g(x(i));
        end
    end
    fprintf("approximated value is %.12f with %d intervals\n",
        ...
        approx_int_val, k-1);
end
```

If we run this code, we get the below result:

```
intergration value is 0.642699081699
approximated value is 0.642699111459 with 2 intervals
approximated value is 0.642699084310 with 3 intervals
approximated value is 0.642699082188 with 4 intervals
approximated value is 0.642699081851 with 5 intervals
```

(c) Let's compare these two methods in terms of three perspectives.

i. Accuracy perspective

Let's consider high order gaussian quadrature with n-th order Legendre polynomial. Then this formula is accurate up to polynomial of degree $2n-1$. Then we get $O(h^{2n})$ error term. (Note that this is from naive assumption because the nodes are not equally spaced.) As we increase the order, h decreases and $2n$ increases, which implies the error term is super small. Now consider composite gaussian quadrature with 3rd order Legendre polynomial with n subintervals. Since gaussian quadrature with 3rd order polynomial has $O(h^6)$ naive error term, we get $O(h^5)$ error term. as we increase the number of subintervals, h decreases but the order of error term is maintained. Hence, we can say that the high order gaussian quadrature method is more accurate than the composite gaussian quadrature method.

ii. Cost perspective

From the above codes, we can think that the number of operations in the method of composite gaussian quadrature is larger than the number of operations in the method of high order gaussian quadrature. But for high order gaussian quadrature, the cost for computing the roots of Legendre polynomial is really big. Hence, the high order gaussian quadrature method is expensive in terms of cost than the composite gaussian quadrature method.

iii. Ease of programming perspective

To use high order gaussian quadrature method, we should hand-write the roots of the high order Legendre polynomial. It is really annoying process. But if we use composite

gaussian quadrature method, we have a small portion of hand-written part, and we can easily consider any number of subintervals (see the line 19-30 3(b)). Hence, programming the composite gaussian method is easier than the high order gaussian method.

4. Below code is the implementation of composite Gaussian quadrature to approximate a triple integral and calculation of original triple intergral value.

```
syms s t u;
f(s,t,u) = s*t*sin(t*u);

% original intergation value
int_val = int(f, s, 0, pi);
int_val = int(int_val, t, 0, pi/2);
int_val = int(int_val, u, 0, pi/3);
fprintf("intergration value is %.12f\n", int_val);

% composite gaussian quadrature
x(1) = 0.7745966692;
x(2) = 0;
x(3) = -x(1);
c(1) = 0.5555555556;
c(2) = 0.8888888889;
c(3) = c(1);

X = linspace(0, pi, 3);
Y = linspace(0, pi/2, 3);
Z = linspace(0, pi/3, 3);
approx_int_val = 0;

% For s-axis
for j = 1:2
    g(s,t,u) = f(((X(j+1)-X(j))*s+(X(j+1)+X(j)))/2,t,u)*(X(j+1)-X(j))/2;
    for i = 1:3
        approx_int_val = approx_int_val + c(i)*g(x(i),t,u);
    end
end
f(t,u) = approx_int_val;
approx_int_val = 0;

% For t-axis
for j = 1:2
    g(t,u) = f(((Y(j+1)-Y(j))*t+(Y(j+1)+Y(j)))/2,u)*(Y(j+1)-Y(j))/2;
    for i = 1:3
        approx_int_val = approx_int_val + c(i)*g(x(i),u);
    end
end
f(u) = approx_int_val;
approx_int_val = 0;

for j = 1:2
    g(u) = f(((Z(j+1)-Z(j))*u+(Z(j+1)+Z(j)))/2)*(Z(j+1)-Z(j))/2;
    for i = 1:3
```

```
        approx_int_val = approx_int_val + c(i)*g(x(i));
    end
end

fprintf("approximated value is %.12f", approx_int_val);
```

If we run this code, we get the below result:

```
intergration value is 3.052124856966
approximated value is 3.052124270058
```