Quadrature Project

@Fire2002

19 October 2023

Introduction

The computation of continuous least squares approximations to $f \in \mathcal{C}[a,b]$ makes evaluations of the inner product $\langle f, \phi_j \rangle = \int_a^b f(x)\phi_j(x)dx$ necessary, where ϕ_j is a polynomial [1]. Many times, such integrals may appear difficult or impossible to evaluate exactly; thus, algorithms show desirable to approximate such integrals speedily and effectively. The goal of this project is to derive the Gaussian quadrature to high precision and apply it to a double integral. Through further study, we analyze the literature on Clenshaw-Curtis quadrature for comparison to Gaussian quadrature.

Mathematical Issues

Gaussian quadrature results from the approach in which both the abscissas x_i and weights α_i are to be determined such that the quadrature

$$Q_n(f) = \sum_{i=1}^{n} \alpha_i f(x_i) \tag{1}$$

has a maximal degree of precision. Since there exist $2n$ unknowns for $Q_n(f)$, we require

$$E_n(x^k) = 0, k = 0, 1, ..., 2n-1 \tag{2}$$

To supply $2n$ equations[2]. Then the expected maximal degree of precision is $\geq 2n-1$. It's additionally noteworthy that the condition $E_n(x^k) = 0$ is equivalent to the nonlinear system

$$\sum_{i=1}^{n} \alpha_i x_i^k = \frac{b^{k+1} - a^{k+1}}{k+1}, k = 0, 1, ..., 2n-1. \tag{3}$$

Note that it's also unclear whether there exists a solution for any a or b, or if the solution is real [3].

It should also be recognized that any quadrature $Q_n(f)$ with degree of precision $\geq n-1$ must be a Newton-Cotes quadrature. That is,

$$\alpha_i = \int_a^b l_j(t)dt, \quad l_j(t) = \prod_{i=1, i\neq j}^{n} \frac{t - x_i}{x_i - x_j} \tag{4}$$

where we recall that the Newton-Cotes formula for the integral $I(f) = \int_a^b f(x)dx$ is based on integrating a polynomial $p(x)$ that interpolates $f(x)$ at a set of equally-spaced points in $[a,b]$ [2].

[1] Interpolatory Quadrature, Rice University.
[2] Gaussian Quadrature, NCSU.
[3] Gaussian Quadrature, Differentiation and Integration, NCSU.

Part I Problem 1

Problem I.1 is to list the first five Legendre polynomials, $P_k(x), k = 1, ..., 5$ over the interval $[-1, 1]$. The first five Legendre polynomials (past the trivial polynomial $P_0(x) = 1$) over the interval $[-1, 1]$ is given by the table

$P_k(x)$	k
x	1
$\frac{1}{2}(3x^2 - 1)$	2
$\frac{1}{2}(5x^3 - 3x)$	3
$\frac{1}{8}(35x^4 - 30x^2 + 3)$	4
$\frac{1}{8}(63x^5 - 70x^3 + 15x)$	5

Expectedly, we see that the highest exponent in the Legendre polynomials corresponds exactly to the k value on the right-hand side (RHS) of the table.

Part I Problem 2

Problem I.2 is to use a table to display the roots of the first five Legendre polynomials from Problem I.1 up to the 16th significant digits. These roots are to represent the abscissas in the corresponding Gaussian quadrature. Below is a table displaying the roots of the first five Legendre polynomials from Problem I.1 up to the 16th significant digits:

Legendre polynomials	roots x
$P_1(x)$	0
$P_2(x)$	$\pm$0.5773502691896257
$P_3(x)$	0, $\pm$0.7745966692414833
$P_4(x)$	$\pm$0.3399810435848562, $\pm$0.8611363115940525
$P_5(x)$	0, $\pm$0.5384693101056830, $\pm$0.9061798459386639

As anticipated, the number of roots corresponds exactly to the degree k of the Legendre polynomial (degree 1 polynomial with 1 root, degree 2 polynomial with 2 roots, etc.). That is, this demonstrates the Fundamental Theorem of Algebra in that any nth degree polynomial has exactly n roots (counting the multiplicity). Altogether, these roots serve as the Gaussian abscissas in the Gaussian quadrature.

Part I Problem 3

Problem I.3 is to use the Gaussian abscissas found in Part I.2 to calculate the corresponding Gaussian weights α_i up to 16th significant digits for each of these polynomials.

Legendre polynomials	weights α_i
$P_1(x)$	0
$P_2(x)$	1.000000000000000, 1.000000000000000
$P_3(x)$	0.8888888888888888, 0.5555555555555555, 0.5555555555555555
$P_4(x)$	0.6521451548625461, 0.6521451548625461, 0.3478548451374538, 0.3478548451374538
$P_5(x)$	0.5688888888888888, 0.4786286704993664, 0.4786286704993664, 0.2369268850561890, 0.2369268850561890

Just in I.2, it's expected that the number of weights correspond exactly to the number of roots, and are given with the same digits of precision.

Part II Problem 1

Problem II.1 is to implement a general purpose code in the form $Q = \textit{my\_single\_integral(@f,A,B)}$ that returns an approximation of a scalar-valued function $f(t)$'s definite integral over the interval $[A, B]$. The Problem requires that the code is sufficiently robust to take in any continuous single-variable function $f(t)$ specified by a user.

The construction of this general purpose code to take in any continuous-single variable function was successful and is provided in the programming issues.

Part II Problem 2

Problem II.2 is to use the 1-D integrator we have previously built to construct a general purpose code for the double integral (of type-I) in the form

$$\int_a^b \int_{g(x)}^{h(x)} u(x, y) dy dx \tag{5}$$

Where the code is of the form $W = \textit{my\_double\_integral(@u,A,B,G,H)}$ and $u(x, y)$ is defined over the planar region $\Omega \coloneqq \{(x, y)|A \leq x \leq B, G(x) \leq y \leq H(x)\}$. Additionally, the code requires allowing users to specify any function u and type-I domain Ω, where G and H may be scalars or functions.

The utilization of the 1-D integrator to construct another general-purpose code for the double integral showed further successful and is also provided in the programming issues.

Part III Problem 1

Problem III.1 is to integrate the integral below (as is) and display results up to 16 digits:

$$\int_0^{\pi/4} \int_0^{\sin x} \frac{2y \sin x + (\cos x)^2}{\sqrt{1 - y^2}} dy dx \tag{6}$$

This, of course, is done by applying our code *my_double_integral* to the double integral. After doing so, we find that the calculator calculation up to 16 digits comes out to be

$$\int_0^{\pi/4} \int_0^{\sin x} \frac{2y \sin x + (\cos x)^2}{\sqrt{1 - y^2}} dy dx = 0.3113484646076979 \tag{7}$$

Part III Problem 2

Part III Problem 2 is to divide the interval $[0, \frac{\pi}{4}]$ into 10 evenly-spaced sub-intervals. And for each node of x, divide the corresponding interval $[0, \sin x]$ in the y direction into the same number of evenly-spaced sub-intervals. As a consequence, the analysis becomes possible for 100 sub-regions.

Under doing so, the task is to write a code to repeatedly use *my_double_integral* over each sub-region and compute the total sum. With this, we are to display results up to 16 digits, and compare our answer to the result from Part III.1 to observe how many digits are the same, determining the precision.

Having divided the intervals $[0, \frac{\pi}{4}]$ and $[0, \sin x]$ into 10 evenly-spaced sub-intervals to obtain 100 sub-regions, we get that the total sum is 0.3801881591106864. The first digit is of the decimal is the same as in Part III, but the rest of the digits are different.

Part IV Problem 1

Part IV Problem 1 is to conduct literature research on reading of the Clenshaw-Curtis quadrature. This would include what the Clenshaw-Curtis quadrature is, how it is defined, how it works, and why this method is preferred in practice.

In the literature analysis of Clenshaw-Curtis quadrature, Trefethen states that the idea of Clenshaw-Curtis quadrature is to utilize Chebyshev nodes instead of optimal nodes [4]. Relative to Gaussian quadrature, Embree explains that Clenshaw-Curtis quadrature is preferred in practice due to the Chebyshev points providing a faster convergence for a fixed number of function evaluations [1].

The Clenshaw-Curtis quadrature is defined under the quadrature formula:

$$\int_{-1}^{1} f(x)dx \approx \sum_{j=0}^{N} f(x_j)w_j \tag{8}$$

where for each $j = 0, 1, ..., N$, the abscissas x_j and weights w_j are defined by the formulas

$$x_j = \cos\left(\frac{j\pi}{N}\right) \tag{9}$$

and

$$w_j = \begin{cases} \frac{2}{N}\sum_{m=0}^{[N/2]} \frac{T_{2m}(x_j)}{1-4m^2} & \text{if} \quad j = 0, N \\ \frac{4}{N}\sum_{m=0}^{[N/2]} \frac{T_{2m}(x_j)}{1-4m^2} & \text{otherwise} \end{cases} \tag{10}$$

is exact for polynomials of degree $\leq N$ (this is known as the $(N+1)$-point Clenshaw-Curtis rule) [5]. Based on the fast Hermite interpolation, Liu et al. shows that the Clenshaw-Curtis quadrature rule is implementable in $\mathcal{O}(N\,logN)$ operations via FFT for N Clenshaw-Curtis nodes.[6]

Part IV Problem 2

Part IV Problem 2 asked we compare our numerical answer for Part III.2 above with that obtained from the calculator calculation, and determine whose answer is more accurate.

If we proceed with a test calculation of the integral through a online calculator (e.g. Symbolab), we would find that (up to 16 digits) the integral equals 0.3113485472432882. This, of course, is closer to the integral computation in Part III.1 than Part III.2. Thus, the answer to Part III.1 is more accurate.

Programming Issues

For Part I, we ensured that the code was able to display the first five Legendre polynomials $P_k(x), k = 1, .., 5$ over the interval $[-1, 1]$ alongside the roots x_i and weights α_i. For Part I, the displaying of the Legendre polynomials with their roots is given through the code:

[4] Gauss Quadrature v. Clenshaw-Curtis. Trefethen, Lloyd.

[5] Curtis-Clenshaw, UC Davis.

[6] Clenshaw-Curtis-type quadrature for hypersingular integrals with high oscillatory kernels, Journal of Applied Mathematics and Computation.

```
%% Part 1
addpath('myFunctions')
syms x y z
syms t

digits(16);

l_1 = legendreP(1,x);
l_2 = legendreP(2,x);
l_3 = legendreP(3,x);
l_4 = legendreP(4,x);
l_5 = legendreP(5,x);

disp("Legendre Polynomials")
disp(l_1)
disp(l_2)
disp(l_3)
disp(l_4)
disp(l_5)
disp("")

roots_1 = vpa(solve(l_1), 16);
roots_2 = vpa(solve(l_2), 16);
roots_3 = vpa(solve(l_3), 16);
roots_4 = vpa(solve(l_4), 16);
roots_5 = vpa(solve(l_5), 16);

legendrePolynomial = [string(l_1);string(l_2);string(l_3);string(l_4);string(l_5)];
first_root = [string(roots_1(1));string(roots_2(1));string(roots_3(1));string(roots_4(1));string(roots_5(1))];
second_root = ["NA";string(roots_2(2));string(roots_3(2));string(roots_4(2));string(roots_5(2))];
third_root = ["NA";"NA";string(roots_3(3));string(roots_4(3));string(roots_5(3))];
fourth_root = ["NA";"NA";"NA";string(roots_4(4));string(roots_5(4))];
fifth_root = ["NA";"NA";"NA";"NA";string(roots_5(5))];

table(legendrePolynomial, first_root, second_root, third_root, fourth_root, fifth_root)
```

Figure 1: Legendre Polynomial Root-Finding Code

Note that at the bottom we use an interpolatory lookup table given by

```
function [Y_lookup, Y] = Table(X, Y, X_lookup)
    n = size(X, 2);

    for j = 2:1:n
        for i = n:-1:j
            Y(i) = (Y(i) - Y(i - 1)) / (X(i) - X(i - j + 1));
        end
    end

    Y_lookup = Y(n);

    for j = n-1:-1:1
        Y_lookup = Y_lookup.* (X_lookup - X(j)) + Y(j);
    end
end
```

Figure 2: lookup table

Below this we iterate through the various degrees of the Legendre polynomials to determine their weights α_i. Here depicts the first 3 iterative loops for calculating the weights:

```
w2 = ones(2,1);
w3 = ones(3,1);
w4 = ones(4,1);
w5 = ones(5,1);

for i=1:1:2
    l = 1;
    for j=1:1:2
        if j ~= i
            l = l * ((t-roots_2(j)) / (roots_2(i) - roots_2(j)));
        end
    end
    w2(i) = int(l, -1, 1);
end

for i=1:1:3
    l = 1;
    for j=1:1:3
        if j ~= i
            l = l * ((t-roots_3(j)) / (roots_3(i) - roots_3(j)));
        end
    end
    w3(i) = int(l, -1, 1);
end

for i=1:1:4
    l = 1;
    for j=1:1:4
        if j ~= i
            l = l * ((t-roots_4(j)) / (roots_4(i) - roots_4(j)));
        end
    end
    w4(i) = int(l, -1, 1);
end
```

Figure 3: iterations of first 3 loops for α_i calculations

And here shows the last 2 loops for calculating the weights and printing them:

```
for i=1:1:4
    l = 1;
    for j=1:1:4
        if j ~= i
            l = l * ((t-roots_4(j)) / (roots_4(i) - roots_4(j)));
        end
    end
    w4(i) = int(l, -1, 1);
end

for i=1:1:5
    l = 1;
    for j=1:1:5
        if j ~= i
            l = l * ((t-roots_5(j)) / (roots_5(i) - roots_5(j)));
        end
    end
    w5(i) = int(l, -1, 1);
end

disp("Weights for P_2")
disp(w2)

disp("Weights for P_3")
disp(w3)

disp("Weights for P_4")
disp(w4)

disp("Weights for P_5")
disp(w5)
```

Figure 4: iterations of last 2 loops for α_i calculations

For Part II.1 we verified that our code is correctly implemented in the form $Q = my\_single\_integral(@f,A,B)$ and returns an approximation of a scalar-valued function $f(t)$'s definite integral over the interval $[A, B]$. The code that returns this approximation is given by

```
%% Part 2

a = 0;
b = 1;
f = x;
single_int_ex = my_single_integral(f, a, b);
fprintf("Example of the single integral function for %s from %d to %d: %s\n\n", string(f), a, b, string(single_int_ex))
```

Figure 5: $Q =$ *my_single_integral* coding calculation

Where the method for computing any continuous single-variable function $f(t)$ is

```
function [Y] = my_single_integral(f, a, b)
    syms x
    syms t

    l_5 = legendreP(5,x);
    roots_5 = vpa(solve(l_5), 16);
    w5 = ones(5,1);

    for i=1:1:5

        l = 1;
        for j=1:1:5
            if j ~= i
                l = l * ((t-roots_5(j)) / (roots_5(i) - roots_5(j)));
            end
        end

        w5(i) = int(l, -1, 1);
    end

    Y = 0;
    for i=1:1:size(w5, 1)
        x = a + ((b-a)/2 * (roots_5(i) + 1)) ;
        Y = Y + (w5(i) * subs(f));
    end

    Y = Y * (b-a)/2;
end
```

Figure 6: $Q =$ *my_single_integral* coding method

Under numerous tests, this code demonstrates sufficiently robust to take in any continuous single-variable function $f(t)$ specified by a user. For Part II.2 we made sure to utilize the 1-D integrator we have previously built to construct a code for the double integral (of type-I). The code that computes the type-I double integral is given as

```
u = z * y.^2;
a = 0;
b = 1;
g = 0;
h = z;

double_int_ex = my_double_integral(u, a, b, g, h);
fprintf("Example of the double integral function for %s where a= %d, b= %d, g= %d, h= %s: %s\n\n", string(u), a, b, g, string(h), string(double_int_ex))
```

Figure 7: *my_double_integral* coding calculation

And the method for computing the type-I double integrals is

```
function [Y] = my_double_integral(u, a, b, g, h)
    addpath('myFunctions')
    syms x y t z
    l_5 = legendreP(5,x);
    roots_5 = vpa(solve(l_5), 16);
    w5 = ones(5,1);
    for i=1:1:5
        l = 1;
        for j=1:1:5
            if j ~= i
                l = l * ((t-roots_5(j)) / (roots_5(i) - roots_5(j)));
            end
        end
        w5(i) = int(l, -1, 1);
    end
    Y = 0;
    for i=1:1:size(w5, 1)
        aot = a + ((b-a)/2 * (roots_5(i) + 1));

        z = aot;
        y = x;

        G = g;
        H = h;

        if class(g) == "sym"
            G = subs(G);
        end
        if class(h) == "sym"
            H = subs(H);
        end
        Y = Y + w5(i) * my_single_integral(subs(u), G, H);
    end
    Y = ((b - a) / 2) * Y;
end
```

Figure 8: *my_double_integral* coding method

Part III.1 found no issues in integrating the integral provided (as is) and displayed results up to 16 digits. The code to integrate the integral of III.1 is provided below:

```
%% Part 3

u = (2*y * sin(z) + (cos(z)).^2) / (sqrt(1-y.^2));
a = 0 ;
b = pi/4;
g = 0;
h = sin(z);

doubl_int_ans = vpa(my_double_integral(u, a, b, g, h), 16);

fprintf("Double Integral Approximation: %s\n\n", string(doubl_int_ans))

sum = 0;

for i=0:1:9

    for j=0:1:9

        a_temp = (a+b) / 10 * i;
        b_temp = (a+b) / 10 * (i+1);
        g_temp = (g+h) / 10 * i;
        h_temp = (g+h) / 10 * (i + 1);

        sum = sum + my_double_integral(u, a_temp, b_temp, g_temp, h_temp);
    end
end

doubl_int_ans_100 = vpa(sum, 16);

fprintf("Double Integral Approximation(100 sub intervals): %s\n\n", string(doubl_int_ans_100))
```

Figure 9: III.1 integration code

Conclusion

Importantly, we must realize that tackling a polynomial system of nonlinear equations is difficult for having no direct techniques. Fortunately, the existence of quadratures such as the Gaussian quadrature and Clenshaw-Curtis quadrature make solving these systems of equations (for the $E_n(x^k) = 0$ condition) possible. In conclusion, the Gaussian quadrature is derived to high precision and shows successfully applicable to the computation of a type-I double integral.

References

Lecture 23: Interpolatory quadrature 4. quadrature. (n.d.). http://sci.utah.edu/~beiwang/teaching/cs6210-fall-2016/lecture23.pdf

Gaussian quadrature - North Carolina State University. (n.d.b).https://mtchu.math.ncsu.edu/Teaching/Lectures/MA427/By_subjects/18_gaussian_quadrature.pdf

Differentiation and Integration - North Carolina State University. (n.d.-a). https://mtchu.math.ncsu.edu/Teaching/Lectures/MA530/chapter8.pdf

Is Gauss quadrature better than Clenshaw–Curtis? — siam review. (n.d.-b). https://epubs.siam.org/doi/epdf/10.1137/060659831

MAT128A: Numerical Analysis Lecture Nineteen: Curtis-Clenshaw...(n.d.e).https://www.math.ucdavis.edu/~bremer/classes/fall2018/MAT128a/lecture19.pdf

Liu, G., & Xiang, S. (2019, January 1). Clenshaw–Curtis-type quadrature rule for hypersingular integrals with highly oscillatory kernels. ScienceDirect. https://www.sciencedirect.com/science/article/pii/S009630031830643X