Math 5610/6860: Introduction to Numerical Analysis I **Fall 2023**

Project: Gauss-Legendre Quadrature

Instructor: Fernando Guevara Vasquez *Name: Cobi Toeun (u1230512)*

Objective

The goal is to explore the implementation of Gaussian Quadrature, specifically Gauss-Legendre Quadrature, to compute the roots and weights of Legendre Polynomials for various degrees $n = 2, \ldots, 10$. I will use the Julia programming language to compute and display a table of the roots and weights for each degree n.

Contents

Code and Outputs

1 Gauss-Legendre Quadrature

Gaussian Quadrature is a numerical integration technique use to approximate definite integrals. It yields exact results for polynomials of degree $2n-1$ or less by a sutiable choice of roots $x_1, x_2, ..., x_n$ in the interval $[a, b]$ and weights $w_1, w_2, ..., w_n$. A common domain of integration for this technique is $[-1, 1]$, where the approximation formula[1] is defined as

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

This form is known as **Gauss-Legendre Quadrature**. As discussed in [1] this quadrature rule gives us an accurate approximation to the integral if $f(x)$ is represented by a polynomial of degree $2n-1$ or less on $[-1, 1]$. For integrating $f(x)$ over $[-1, 1]$ the associated polynomials represented as $P_n(x)$, where n is the degree, are **Legendre polynomials**. These polynomials play a crucial role in this method due to their unique properties in the sense that they minimize the error in approximating the integrals.

The first few Legendre Polynomials[1][2] are

- $P_0(x) = 1$
- $P_1(x) = x$
- $P_2(x) = \frac{1}{2}(3x^2 - 1) = x^2 - \frac{1}{3}$
- $P_3(x) = \frac{1}{2}(5x^3 - 3x) = x^3 - \frac{3}{5}x$
- $P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3) = x^4 - \frac{6}{7}x^2 + \frac{3}{35}$

These polynomials are orthogonal on the interval $[-1, 1]$ with respect to the weight function $w(x) = 1$. They have distinct roots, symmetry with respect to the origin, and are the optimal choice for determining the parameters that yield the roots and weights for Gauss-Legendre Quadrature. Note that manually determining all the polynomials, along with their roots and weights, would be both tedious and challenging. To streamline this process, I've developed a couple of functions in Julia to efficiently solve this problem.

2 Gauss-Legendre Implementation

Calculating the Legendre polynomials for degrees $n = 2, \cdots, 10$ involve using a three-term recurrence relation that the polynomials satisfy. This relation[3][4] is defined as

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x),\ n \geq 1$$

For simplicity in the code, I set $n = n - 1$ to rewrite the relation[4] as

$$P_n(x) = \frac{1}{n}\left[(2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)\right],\ n \geq 2$$

It's important to note this formula uses recursion, which can lead to inefficiency in computation. Nevertheless, for simplicity, I've implemented it in Julia as a recursive function.

To determine the weights, I use the formulas[5][6] below, with a function defining $P_n'(x_i)$.

$$w_i = \frac{2}{(1-x_i^2)[P_n'(x_i)]^2},\ P_n'(x) = \frac{1}{(x^2-1)}\{n \cdot [xP_n(x) - P_{n-1}(x)]\}$$

Now I can calculate the roots and weights for each Legendre polynomial. The primary function computes the roots and weights of the Legendre polynomials based on the given degree n. To find the roots of each polynomial, I call my *legendre_poly()* function with the necessary values then essentially set $P_n(x_i) = 0$. In Julia, you can utilize the **Roots.jl** package and call the *findzeros()* function. From there the weights are found using the two formulas above.

https://juliapackages.com/p/roots

```julia
# find legendre poly for degree n
function legendre(n, x)
    if n == 0 return 1.0  # base case: p0(x) = 1
    elseif n == 1 return x  # base case: p1(x) = x
    else
        # three-term recurrence formula to calculate Legendre polynomial pn(x) for n
        return ((2 * n - 1) * x * legendre(n - 1, x) - (n - 1) * legendre(n - 2, x)) / n
    end
end

# calculate derivative of Legendre polynomial for degree n
function legendre_derivative(n, x)
    if n == 0 return 0.0  # derivative of 1 is 0
    else
        # recursive formula to calculate derivative
        return n * (x * legendre(n, x) - legendre(n - 1, x)) / (x^2 - 1)
    end
end

# find Legendre polynomial roots and weights
function legendre_roots_and_weights(n)
    if n <= -1 || typeof(n) != Int64 # error check
        throw(DomainError(n, "n must be non-negative integer!"))
    end
    # Use the roots library to find the roots of Legendre polynomial pn(x)
    roots = find_zeros(x -> legendre(n,x), -1.0, 1.0)

    # calculate the weights of the Legendre polynomial roots using weights formula
    weights = [2 / ((1 - root^2) * (legendre_derivative(n, root))^2) for root in roots]

    return roots, weights
end
```

Figure 1: Julia functions for computing Gauss-Legendre roots and weights

3 Unit Testing

Before code execution, it's important to conduct tests to verify all functions work. I start with importing the **FastGaussQuadrature.jl** and **LegendrePolynomials.jl** external packages, as well as the **Linear Algebra** and **Test** libraries. These external packages and libraries have been throughly tested and assist me in identifing potential flaws within my code. Note I don't cover all test cases, but these should be sufficient for what I'm trying to acheive. Refer to my code and comments for further details.

https://juliapackages.com/p/fastgaussquadrature : *https://juliapackages.com/p/legendrepolynomials*
https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/ : *https://docs.julialang.org/en/v1/stdlib/Test/*

```
my_roots_weights = []                                    # lists to store data
expected_roots_weights = []

# not clean and not optimized, but works enough
for n = 2:10
    my_roots, my_weights = legendre_roots_and_weights(n)     # call my function
    exp_roots, exp_weights = gausslegendre(n)                # function from package

    for (my_root, my_weight) in zip(my_roots, my_weights)          # whatever i get
        push!(my_roots_weights, [my_root, my_weight])
    end

    for (exp_root, exp_weight) in zip(exp_roots, exp_weights)     # what's expected
        push!(expected_roots_weights, [exp_root, exp_weight])
    end
end
```

Figure 2: Storing my function and external gauss function in lists for unit tests

```
@testset "Gauss-Legendre" begin

    @test_throws DomainError legendre_roots_and_weights(-2)    # check error cases
     @test_throws DomainError legendre_roots_and_weights(5.5)

    @test legendre(6, 0.5) ≈ Pl(0.5, 6)                       # legendre poly n=6
    @test legendre_derivative(6, 0.5) ≈ dnPl(0.5, 6, 1)       # first derivative

    exp_roots, exp_weights = gausslegendre(6)                      # external package
    my_roots, my_weights = legendre_roots_and_weights(6)       # my function

    @test exp_roots ≈ my_roots
    @test exp_weights ≈ my_weights

    f(x) = x^2                                                 # check integral
    I = dot(my_weights, f.(my_roots))
    @test I ≈ 2/3

    g(x) = x^4-4x^2+12                                         # another integral
    I = dot(my_weights, g.(my_roots))
    @test I ≈ 326/15

    @test my_roots_weights ≈ expected_roots_weights               # check from n=2-10

end
```

```
Test Summary:  | Pass  Total  Time
Gauss-Legendre |    9      9  0.1s
```

Figure 3: Unit tests

4 Results

After executing the main code and utilizing the **PrettyTables.jl** package for data visualization, I have successfully obtained the roots and weights for each degree $n = 2, \ldots, 10$. The output table below presents the results. Refer to the next page to view the table more clearly.

https://juliapackages.com/p/prettytables

```julia
df = DataFrame(Degree = String[], Roots = Float64[], Weights = Float64[]) # store data
n = 10                                                                     # set max degree
last_degree = 0                                                            # track degree n

# calculate and add the roots and weights for degrees 2 to 10
for n in 2:n
    roots, weights = legendre_roots_and_weights(n)

    # add the roots and weights for the current degree
    for (root, weight) in zip(roots, weights)                              # pair roots and weights
        if n != last_degree
            push!(df, [string(n), root, weight])                           # add data with degree n
            last_degree = n                                                # switch degree if needed
        else
            push!(df, ["", root, weight])                                  # makes degree col cleaner
        end
    end
end

#show(df, show_row_number = false, allcols=true, allrows=true)
pretty_table(df)                                                           # convert to a pretty table :D
```

Figure 4: Code to produce table of roots and weights of LP per degree n

Degree	Roots	Weights
2	-0.57735	1.0
	0.57735	1.0
3	-0.774597	0.555556
	0.0	0.888889
	0.774597	0.555556
4	-0.861136	0.347855
	-0.339981	0.652145
	0.339981	0.652145
	0.861136	0.347855
5	-0.90618	0.236927
	-0.538469	0.478629
	0.0	0.568889
	0.538469	0.478629
	0.90618	0.236927
6	-0.93247	0.171324
	-0.661209	0.360762
	-0.238619	0.467914
	0.238619	0.467914
	0.661209	0.360762
	0.93247	0.171324
7	-0.949108	0.129485
	-0.741531	0.279705
	-0.405845	0.38183
	0.0	0.417959
	0.405845	0.38183
	0.741531	0.279705
	0.949108	0.129485
8	-0.96029	0.101229
	-0.796666	0.222381
	-0.525532	0.313707
	-0.183435	0.362684
	0.183435	0.362684
	0.525532	0.313707
	0.796666	0.222381
	0.96029	0.101229
9	-0.96816	0.0812744
	-0.836031	0.180648
	-0.613371	0.260611
	-0.324253	0.312347
	0.0	0.330239
	0.324253	0.312347
	0.613371	0.260611
	0.836031	0.180648
	0.96816	0.0812744
10	-0.973907	0.0666713
	-0.865063	0.149451
	-0.67941	0.219086
	-0.433395	0.269267
	-0.148874	0.295524
	0.148874	0.295524
	0.433395	0.269267
	0.67941	0.219086
	0.865063	0.149451
	0.973907	0.0666713

Figure 5: Julia output table of roots and weights of LP degrees n = 2, ... , 10

Degree n	Roots x_i	Weights w_i
2	-0.57735	1.0
	0.57735	1.0
3	-0.774597	0.555556
	0.0	0.888889
	0.774597	0.555556
4	-0.861136	0.347855
	-0.339981	0.652145
	0.339981	0.652145
	0.861136	0.347855
5	-0.90618	0.236927
	-0.538469	0.478629
	0.0	0.568889
	0.538469	0.478629
	0.90618	0.236927
6	-0.93247	0.171324
	-0.661209	0.360762
	-0.238619	0.467914
	0.238619	0.467914
	0.661209	0.360762
	0.93247	0.171324
7	-0.949108	0.129485
	-0.741531	0.279705
	-0.405845	0.38183
	0.0	0.417959
	0.405845	0.38183
	0.741531	0.279705
	0.949108	0.129485
8	-0.96029	0.101229
	-0.796666	0.222381
	-0.525532	0.313707
	-0.183435	0.362684
	0.183435	0.362684
	0.525532	0.313707
	0.796666	0.222381
	0.96029	0.101229
9	-0.96816	0.0812744
	-0.836031	0.180648
	-0.613371	0.260611
	-0.324253	0.312347
	0.0	0.330239
	0.324253	0.312347
	0.613371	0.260611
	0.836031	0.180648
	0.96816	0.0812744
10	-0.973907	0.0666713
	-0.865063	0.149451
	-0.67941	0.219086
	-0.433395	0.269267
	-0.148874	0.295524
	0.148874	0.295524
	0.433395	0.269267
	0.67941	0.219086
	0.865063	0.149451
	0.973907	0.0666713

Table 1: Roots and Weights for LP of degrees n= 2, ... , 10

5 Optimization

Here you can see the computation time and memory allocation of my custom function and **FastGaussQuadrature.jl** gauss-legendre function for when n = 10.

```
@time roots, weights = gausslegendre(10)
@time _roots, _weights = legendre_roots_and_weights(10)

  0.000028 seconds (13 allocations: 1.297 KiB)
  0.002301 seconds (2.83 k allocations: 71.578 KiB)
```

Figure 6: Computation time for n=10

```
@time roots, weights = gausslegendre(25)
@time _roots, _weights = legendre_roots_and_weights(25)

  0.000032 seconds (13 allocations: 2.969 KiB)
  6.111748 seconds (6.10 k allocations: 152.406 KiB)
```

Figure 7: Computation time for n=25

Initially with n = 10, the difference is relatively insignificant. However when n = 25, the divergence in both computation time and memory allocation become more pronounced and is even bigger when n becomes larger. Nonetheless, I only needed to find the results for degress $n = 2, ..., 10$ so the recursive methods were simplest to implement and are sufficient.

Now what IF I want to use Gauss-Lengendre for $n \geq 1000$? My recursive methods are not efficient enough due to extra memory usage, and overhead of function calls and returns. In my code, the most significant potential for optimization lies in caching and reusing intermediate results and avoiding repeated calculations. Two basic techniques to solve this are using memoization (top-down) or an iterative (bottom-up) approach.[7] I won't go too in-depth, but here are some brief explanations.

Memoization Approach: From [7], memoization caches and reuses prior results, specifically storing intermediate Legendre polynomial calculations for different n values. This approach optimizes efficiency by eliminating redundant recursive function calls and efficiently retrieving precomputed Legendre polynomials.

```
# basic idea for memoization for legendre polys; can be applied to p'n(x)
memo = (use dictionary or hashmap) # cache poly calculations
function memo_legendre(n, x)
  if key=(n,x) in memo # reduces extra calculations
    return memo[key=(n,x)]
  if ... # base cases for n = 0 or 1
  else result = (three-term formula) # use recursion
  memo[key=(n,x)] = result # cache for future use
  return result
```

Iterative Approach: From [7], iteration employs loops and constructs to eliminate function call and recursion overhead. By initializing and updating variables within a loop, it creates efficient computation, minimizes memory usage, and prevents stack overflow issues.

```
# basic idea for iterative approach for legendre polys; can be applied to p'n(x)
function iterative_legendre(n,x)
  if ... # base cases for n = 0 or 1
  else
    p0, p1 = 1, x # init first two legendre polys
    for i in 2:n
      p0, p1 = p1, (three-term formula) # update values
    end
    return p1
```

6 Conclusion

I'll proceed with a quick application of my custom Gauss-Legendre functions to approximate

$$f(x) = e^x cos(x) \Longrightarrow \int_{-1}^{1} e^x cos(x) dx$$

This integral is an example from B&F 4.7[1]. I'll utilize the **QuadGK.jl** package to calculate a precise value and then compare it to the approximations I've generated. The outcomes are presented in code below.

https://juliamath.github.io/QuadGK.jl/stable/

```
f(x) = exp(x)*cos(x)
I = quadgk(f, -1, 1)[1]                          # compute integral using package

println("Expected solution: $I\n")               # solution to integral

println("Gauss-Legendre Approximation:")
for n = 2:10
    r, w = legendre_roots_and_weights(n)
    println("n = $n: I = ", dot(w, f.(r)))       # approx per degree n
end
```

```
Expected solution: 1.9334214962007135

Gauss-Legendre Approximation:
n = 2: I = 1.9629727607543537
n = 3: I = 1.9333904692642967
n = 4: I = 1.9334168941640448
n = 5: I = 1.933421497268504
n = 6: I = 1.9334214962992713
n = 7: I = 1.9334214962007055
n = 8: I = 1.9334214962007126
n = 9: I = 1.9334214962007132
n = 10: I = 1.9334214962007141
```

Figure 8: Approximations of definite integral from B&F 4.7

For $f(x)$, you can see a higher order rule generally gives a better approximation to the required integration. This continuous improvement in accuracy highlights the practical advantage of employing higher-order Gauss-Legendre quadrature rules.

In conclusion, Gauss-Legendre quadrature is a highly effective method for accurately approximating definite integrals within the interval $[-1, 1]$. This numerical integration technique strategically selects a set of roots and weights based on associated orthogonal Legendre polynomials, providing an efficient approach. Applications of Gauss-Legendre quadrature extends across various fields, including mathematics, physics, engineering, and other scientific disciplines. Its reliability in achieving high precision makes it a preferred choice for accurate numerical integration in various computational applications.

References

[1] Burden, R. L., & Faires, J. D. (2015). Numerical Analysis (10th ed.) Sec 4.7, pg 229-232. Brooks/Cole (Cengage Learning).

[2] Wolfram Research. (2023, November). Legendre Polynomial. MathWorld.
URL: https://mathworld.wolfram.com/LegendrePolynomial.html

[3] Koepf, W. Hypergeometric Summation: An Algorithmic Approach to Summation and Special Function Identities, pg 2. Braunschweig, Germany: Vieweg, 1998.

[4] Herman, Russell. University of North Carolina Wilmington, LibreTexts. (2023, November). Legendre Polynomials. In A First Course in Differential Equations for Scientists and Engineers (Herman).
URL: https://math.libretexts.org/Bookshelves/Differential_Equations/At_First_Course_in_Differential_Equations_for_Scientists_and_Engineers_(Herman)/04%3A_Series_Solutions/4.05%3A_Legendre_Polynomials

[5] Hildebrand, F. B. Introduction to Numerical Analysis, pg 324. New York: McGraw-Hill, 1956.

[6] Abramowitz, Milton; Stegun, Irene Ann, eds. (1983) [June 1964]. "Chapter 25.4, Integration". Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. Applied Mathematics Series. Vol. 55 (Ninth reprint with additional corrections of tenth original printing with corrections (December 1972); first ed.). Washington D.C.; New York: United States Department of Commerce, National Bureau of Standards; Dover Publications. ISBN 978-0-486-61272-0. LCCN 64-60036. MR 0167642. LCCN 65-12253.

[7] BasuMallick, Chiradeep. (2022, October 19) What is Dynamic Programming? SpiceWorks.
URL: https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/

Source code and jupyter notebook are available upon request.