Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Generalized Gaussian Quadrature for Integrals with Singular Weights

Bachelor Thesis

Victor Kawasaki-Borruat

March 7, 2022

Supervisor: Prof. Dr. R. Hiptmair
Department of Applied Mathematics, ETH Zürich

Acknowledgements

Before we dive into the mathematics, I would like to make a few acknowledgements. First and foremost I must thank my supervisor Prof. Ralf Hiptmair for accepting my request to write a Bachelor's Thesis under his guidance. I am grateful to have been able to practice my skills with him, and can only hope that the feeling is mutual. Moreover I would also like to extend my acknowledgements to my parents, siblings, and close friends, who supported and witnessed the development of this project and my liking for mathematics, despite a rather unstable start in my studies.

This work is dedicated to the people who had faith in me in times where I couldn't.

Abstract

It goes without saying that the importance of numerical integration in the fields of computational science and engineering is immense. In cases where a) analytical solutions are too convoluted or simply nonexistent (especially in cases of singular integrands) or b) integrands are sampled and thus incomplete, it is often still of high interest to efficiently approximate the integral. There is thus a need for accurate and efficient computational methods in this domain. Gaussian quadrature rules are a highly efficient method of approximating integrals numerically. Their accuracy lies in the exploitation of associated orthogonal polynomials with respect to the used measure (i.e. weight function). This paper will cover the theoretical and computational aspects of border-singular Jacobi- resp. logarithmic- weighted Gaussian quadratures, and observe their accuracy.

The solving methods are based off of multiple works by W. Gautschi (both theoretical and programmatical), and implement different variations of the Golub-Welsch algorithm. As a result, a `C++` framework has been developed, which accurately approximates the desired singular integrals. Accuracy is quantified by observing a) exactness in polynomial quadratures and b) exponential convergence for smooth integrands. Moreover, the obtained runtimes of the algorithms are coherent with their expected computational complexity.

Contents

Acknowledgements **i**

Abstract **ii**

Contents **iii**

1 Introduction **1**

2 Mathematical Foundations **2**
2.1 (Weighted) Integrals 2
2.2 Polynomial Interpolation 4
2.3 Weight Functions and Orthogonal Polynomials 4
2.4 Matrix Eigenvalue Problem 6

3 Numerical Integration / Weighted Quadratures **8**
3.1 Quadratures 8
3.1.1 Affine Pullback 9
3.1.2 Errors and Precision 9
3.2 Newton-Cotes Formulae 10
3.2.1 Approximation through Interpolation 10
3.2.2 Maximal Order of Classical Numerical Integration . . 11
3.2.3 Interpolation Errors 11
3.3 Gaussian Quadratures 11
3.3.1 Maximal Order Guarantee for Gaussian Quadrature . 12
3.3.2 Error of Gaussian Quadratures 13

4 Formal Problem Statement **14**
4.1 Singular Gaussian Quadrature
& the Eigenvalue Problem 14
4.1.1 Computing the Nodes 15
4.1.2 Computing the Weights 16

4.2 Golub-Welsch Algorithm 16

5 Singular Weight Functions **18**
5.1 Jacobi Weight Function 18
5.1.1 Jacobi Polynomials 18
5.2 Special Cases of the Jacobi Weight Function 19
5.2.1 Legendre Weight Function 20
5.2.2 Chebyshev Weight Function 20
5.2.3 Affine Pullback 20
5.3 Logarithmic Weight Function 21
5.3.1 Modified Chebyshev Algorithm 22
5.3.2 Computing the Quadrature 23
5.3.3 Affine Pullback 23

6 Code Implementation: The SingGQ Library **24**
6.1 The GaussRule Class 24
6.1.1 Namespaces 25
6.2 The GaussJacobiRule Class 25
6.2.1 Computing the nodes and weights 26
6.2.2 Evaluating the integral 27
6.2.3 Different integral boundaries 27
6.2.4 Practical Subclasses 27
6.3 The GaussLogRule Class 28
6.3.1 Computing the special quadrature nodes and weights 29
6.3.2 Evaluating the integral 30
6.3.3 Different integral boundaries 30
6.4 Runtine Analysis 31

7 Accuracy Analysis **32**
7.1 Polynomial Exactness 32
7.1.1 Gauss-Jacobi Polynomial Exactness 32
7.1.2 Gauss-Log Polynomial Exactness 33
7.2 Validation with Smooth Integrands 33

8 Conclusions **36**

A Appendix **37**
A.1 Error Plots 37
A.1.1 Jacobi Weight Function 37
A.1.2 Logarithmic Weight Function 40
A.2 Other Plots 41

Bibliography **42**

Chapter 1

Introduction

Numerical quadrature is a term designating a wide range of algorithms relating to the numerical computation of one dimensional integrals. The term itself was historically used to designate calculating area. It is clear that computing integrals is an extremely important problem in most aspects of computational science, 'trading' the complexity of an integral for that of a sum.

Quadrature rules can be classified in different types of categories, some more or less accurate. This thesis will revolve around *Gaussian quadratures*, one of – if not the most – precise fixed-point non-adaptive quadrature rules.

This paper will first present the mathematical foundations and most concepts relevant to the topic. Off of these foundations, a general introduction to numerical quadrature will be presented, followed by a definition of *Gaussian quadratures*. From this general definition, a formal problem statement is derived, which will be symbolically solved via the Golub-Welsch algorithm.

Once the solving algorithm is clear, it will be adapted to two weight functions of interest, and some optimisation tweaking steps may be introduced.

Finally, the programming implementation is given, coherently linking all the aforementioned steps, along with a runtime and accuracy analysis of the code.

The SingGQ Framework

The framework developed in this paper is a `C++` library, and is available for download at https://github.com/itiskawa/singular-gauss-quadrature. A setup guide and examples of usage are available on the repository. Full documentation is available at https://itiskawa.github.io/SingGQ-doc/.

Chapter 2

Mathematical Foundations

Integral computation may be done over an uncountably infinite amount of intervals in $\mathbb{R}$. Thus, for sake of brevity the entirety of this paper will assume the domain of integration I to be *open*, such that $I =]a, b[$, with $a < b < \infty$. Moreover, any time a domain is referred to as I without further context, it is assumed to be as defined in this paragraph.

The following section may be treated as an index for notations, definitions, propositions and theorems which will be used later on.

2.1 (Weighted) Integrals

Definition 2.1 (Shorthand for Integration) *Let $f : I \longrightarrow \mathbb{R}$. Then,*

$$I[f] := \int_I f(x)dx = \int_a^b f(x)dx \tag{2.1}$$

Definition 2.2 (Singularity) *A function $f : I \longrightarrow \mathbb{R}$ is said singular if itself or its first derivative contains a singularity over I, i.e. there exists one or multiple points in I such that f or its derivative are not well-defined on these points.*

Definition 2.3 (Weight Function) *A weight function over I, $\mu : I \longrightarrow \mathbb{R}_+$ is an integrable function such that*

$$\int_I \mu(x)dx < \infty \tag{2.2}$$

Weight functions can be discrete or continuous. In this paper, when $w(x)$ is mentioned as a weight function, it is assumed to be continuous.

Definition 2.4 (Weighted Integral) *Let $f : I \longrightarrow \mathbb{R}$ be a function, $\mu : I \longrightarrow \mathbb{R}_+$ a weight function. The weighted integral of f with respect to μ is denoted*

$$I[f;\mu] := \int_I f(x)\mu(x)dx = \int_a^b (f \cdot \mu)(x)dx \tag{2.3}$$

Note that any 'regular' (unweighted) integral is in fact weighted, where $\mu(x) = 1, \forall x \in I$.

Example 2.5 *To further illustrate this concept, let $\mu(x)$ over I be the Jacobi weight function (further detailed in 5.1), which will be one of the foci of this paper.*

$$\mu(x) = (1-x)^{\alpha}(1+x)^{\beta} \quad \alpha, \beta > -1$$

Let $\alpha = \frac{1}{2}$ and $\beta = 1$, then this instance of the Jacobi weight function yields:

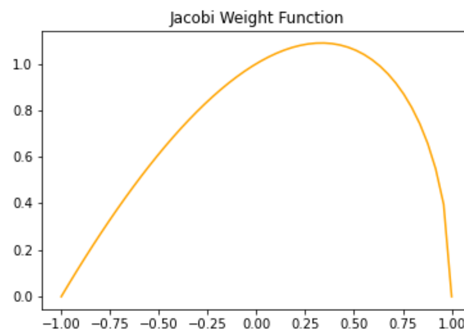
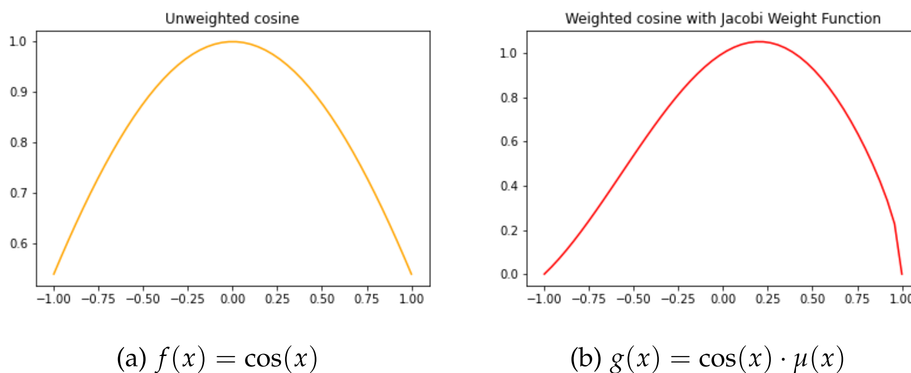


Figure 2.1: Jacobi weight function with $\alpha = 0.5, \beta = 1$

Now consider the function $f(x) = \cos(x)$ over I. The figure below demonstrates the effect of the aforementioned weight function on $f(x)$. Albeit slight, the changes occurring are non-negligible, as well as those acting on the surface under the weighted curve. This example demonstrates a subtle change, but all depends on the integrand f and the weight function μ.



(a) $f(x) = \cos(x)$

(b) $g(x) = \cos(x) \cdot \mu(x)$

Figure 2.2: Unweighted vs weighted cosine function

2.2 Polynomial Interpolation

Definition 2.6 (Partition of an Interval) *A partition Δ of I is a real finite sequence $\Delta = \{x_0, x_1, x_2, ..., x_n\}$ such that $a < x_0 < x_1 < ... < x_{n-1} < x_n < b$.*

Furthermore, a partition is said equidistant if the spacing between all x_i is equal. More formally, $h = \frac{x_n - x_0}{n} = x_{i+1} - x_i$ for all $i \in 0, 1, ..., n-1$

Definition 2.7 (Interpolation) *Let $\Delta = \{x_i\}_{i=0}^n \subset \mathbb{R}$ be a partition of I, and a set a points $S = \{x_i, y_i\}_{i=0}^n$. An interpolation over S is a mathematical operation which models a function f such that $f(x_i) = y_i$, for all $i \in \{0, 1, ..., n\}$.*

Definition 2.8 (Knot polynomial) *Let $S = \{x_0, x_1, ..., x_n\}$ be a partition of I. The knot polynomial associated to the points $\{x_i\}_{i=0}^n$ is denoted*

$$\omega(x) = \prod_{i=0}^{n} (x - x_i) \tag{2.4}$$

The zeros of the knot polynomial w.r. to S are the elements of S themselves.

Definition 2.9 (Lagrange Polynomial) *Let $S = \{x_i, y_i\}_{i=0}^n$ be a set of points such that $\{x_i\}_{i=0}^n$ is a partition of I. The associated i^{th} Lagrange polynomial is defined by*

$$l_i(x) := \frac{\omega(x)}{(x - x_i)\omega'(x)} = \prod_{j=0, j \neq i}^{n} \frac{x - x_j}{x_i - x_j} \tag{2.5}$$

Notice that $l_i(x_j) = \delta_{ij}$

Proposition 2.10 *Suppose we are looking to interpolate the points $\{x_i, y_i\}_{i=0}^n$, then*

$$p(x) = \sum_{i=0}^{n} y_i l_i(x) \tag{2.6}$$

is a valid polynomial interpolation of degree n.

Proof $p(x_i) = \sum_{i=0}^n y_i l_i(x_i) = y_i$ for all i, using the Lagrange polynomial property, thus making p a valid polynomial interpolation of degree n. □

2.3 Weight Functions and Orthogonal Polynomials

There is an important link to be made between these two concepts, which will directly impact the computation of 'optimal' quadratures. Since explaining the entire theory of space, measures and orthogonality in detail is by far beyond the scope of this thesis, the following section will present useful concepts relating to orthogonal polynomials and properties of continuous weight functions.

As described in definition 2.3, a weight function can be seen as distorting the Euclidean space (here $\mathbb{R}$), in such a way that typical measures are now rendered obsolete. This entails a redefinition of some measures with respect to the weight function at hand. Recall that (in definition 2.4) the weighted integral operator has already been defined, but one more relevant operator must be redefined for this paper.

Definition 2.11 ((Weighted) Inner Product of a Two Functions) *Let $\mu(x)$ be a weight function over I, p and q are functions defined over I. An inner product with respect to μ, denoted $\langle \cdot, \cdot \rangle_\mu$ is defined as follows:*

$$\langle p, q \rangle_\mu = \int_I p(x)q(x)\mu(x)dx \tag{2.7}$$

Notice that $\langle p, q \rangle_\mu = I[(p \cdot q); \mu]$, which yields a new well-defined $\mathcal{L}^2$-norm.

Definition 2.12 (Orthogonality) *The principle of orthogonality is defined between two vectors in a defined vector space as follows:*

Let x,$y \in \mathbb{K}$, an n-dimensional vector space, then x and y are said orthogonal if $\langle x, y \rangle = 0$, where $\langle \cdot, \cdot \rangle$ is a defined inner product over $\mathbb{K}$.

A vector x may also be orthogonal to a subspace Y if $\langle x, y \rangle = 0$, $\forall y \in Y$.

Combining both previous definitions implies that two polynomials $p, q \in \mathbb{P}$, $n \in \mathbb{N}$ can be *orthogonal with respect to a weight function μ.*

Example 2.13 *Let $x = \cos(\theta)$, $T_n(x) = T_n(\cos(\theta)) = \cos(n\theta)$. Under the Euclidean inner product, it can be shown that $T_n(x)$ and $T_p(x)$ are not orthogonal over the interval $]-1, 1[$.*

$$\int_{-1}^{1} \cos(n \cos^{-1}(x)) \cos(p \cos^{-1}(x))dx \neq 0 \;\; \text{if } p \neq n$$

However, given $x = \cos(\theta)$, notice that

$$\int_{-1}^{1} \frac{T_n(x)T_p(x)}{\sqrt{1-x^2}} = \int_0^{\pi} \cos(n\theta)\cos(p\theta)d\theta = 0 \iff n \neq p$$

Thus demonstrating orthogonality of $T_n(x)$ and $T_p(x)$ with respect to the weight function $\mu(x) = \frac{1}{\sqrt{1-x^2}}$

Definition 2.14 (Orthogonal Polynomials) *Let p, $q \in \mathbb{P}^n$, and w a weight function. Then p and q are said orthogonal if $\langle p, q \rangle_\mu = 0$.*

Definition 2.15 (Orthonormality) *Let $p, q \in \mathbb{P}^n$ be orthogonal polynomials such that $p \neq q$, and μ a weight function over I. p, q are said orthonormal if they are unit vectors with respect to μ, i.e. $\langle p, p \rangle_\mu^{1/2} = 1$*

Proposition 2.16 (3-step Recursion) *Let μ be a weight function and its associated inner product $\langle \cdot, \cdot \rangle_\mu$ over I. Then there exists a sequence $\{u_n\}_{n=0}^{\infty}$ with $u_n \in \mathbb{P}^n$ for all $n \in \mathbb{N}$ such that:*

- $u_n(x) = \sum_{i=0}^{n} \gamma_i x^i \ , \ \gamma_i > 0 \forall i \in \mathbb{N}$
- $\langle u_n, u_m \rangle = \delta_{n,m}$

These properties imply that the polynomials of the sequence $\{u_k\}_{k=0}^{n}$ generate an orthonomal basis of $\mathbb{P}^n$.

Set $u_{-1} = 0$, $u_0 = \gamma_0 = I[w]^{-1/2}$, then $\exists \alpha_k, \beta_k$ such that the following 3-term recusrion appears:

- $\beta_{n+1} u_{n+1}(x) = x u_n(x) - \alpha_{n+1} u_n(x) - \beta_n u_{n-1}(x)$
- $\alpha_{n+1} = \langle u_n(x), x u_n(x) \rangle$
- $\beta_{n+1} = \frac{\gamma_n}{\gamma_{n+1}}$, *and* $\beta_0 = 0$

Proof Provided as the proof of *Satz 33.1* in [7]. Understanding the demonstration of the above result was deemed out of the scope of this paper. □

Theorem 2.17 (Zeros of Orthogonal Polynomial) *Let μ be a weight function over I and $\{u_i\}_{i=0}^{\infty}$ its associated orthonormal polynomial sequence. Then the zeros of $u_n(x)$ are all real and lie within I.*

Proof Provided as the proof of *Satz 34.1* in [7]. Again, too many new concepts only relevant to the proof would need to be introduced. □

2.4 Matrix Eigenvalue Problem

As for the previous chapters, knowledge is assumed, here in matrix algebra. Concepts such as linear independence of vectors, matrix multiplication, matrix-vector multiplication, transposition and matrix inversion will be used.

The *Matrix Eigenvalue Problem* is a very common problem in Mathematics, specifically in Numerical Analysis, as solving linear systems is a very important part of this field. Many different numerical methods solving the Eigenvalue Problem exist, examples of which are the *Power Method*or the *Jacobi Method*.

This paper will not explain any of these methods – despite them being a very interesting read – as the main goal is not to implement an Eigenvalue Problem Solver. Rather, solving the Eigenvalue Problem will simply appear to be a natural means of solving one principal component of the puzzle at hand.

The following definitions are considered to be reminders.

Definition 2.18 (Eigenvalues and Eigenvectors) *Let A be a real-valued nxn matrix. Then $v \in \mathbb{R}^n$ is an eigenvector of A if $v \neq 0$ and satisfies the following:*

$$A \cdot v = \lambda v \ , \ \lambda \in \mathbb{R} \tag{2.8}$$

λ is v's corresponding eigenvalue.

Definition 2.19 (Eigendecomposition) *The eigendecomposition of a square matrix $A \in \mathbb{R}^{NxN}$ is possible when A's eigenvectors v_i are linearly independent, and λ_i is v_i's corresponding eigenvalue. Let $V, \lambda \in \mathbb{R}^{NxN}$ such that:*

$$V = \begin{bmatrix} v_1, v_2, ...v_N \end{bmatrix} \ , \Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_N \end{bmatrix}$$

Then A's eigendecomposition is $V \Lambda V^T$

Definition 2.20 (Similar Matrices) *Two square matrices $A, B \in \mathbb{R}^{NxN}$ are said to be similar if there exists an invertible matrix $P \in \mathbb{R}^{NxN}$ such that $B = P^{-1}AP$ i.e. A and B represent the same linear transformation, but under different bases. Similar matrices have matching eigenvalues.*

Definition 2.21 (Self-Adjoint (Hermitian) Matrix) *A square matrix A is said self-adjoint (or hermitian) if it is equal to its complex transpose A^H, i.e. $a_{ij} = \overline{a_{ji}}$.*

Real-valued self-adjoint matrices are called symmetrical.

By the fact that a symmetrical matrix $A = A^T$, then its eigendecomposition yields:

$$A = V \Lambda V^T = A^T = (V \Lambda V^T)^T = V \Lambda^T V^T \tag{2.9}$$

thus implying that the eigenvalues λ_i satisfy the equation:

$$\lambda_i = \lambda_{N-(i-1)} \tag{2.10}$$

for all $i \in 1, 2, ..., N$

This property will be later exploited for faster computation.

Chapter 3

Numerical Integration / Weighted Quadratures

3.1 Quadratures

As mentioned in the introduction, a *quadrature* is a numerical method to compute a one-dimensional integral. There are many different ways to compute quadratures, a few of which will be described below, but all of them follow the same principle:

Definition 3.1 (n-point Quadrature Rule) *Let $f : I \longrightarrow \mathbb{R}$ be a function, and $\mu : I \longrightarrow \mathbb{R}_+$ a weight function over I. An n-point quadrature rule $Q_n[\cdot;\mu]$ approximates the integral of $(f \cdot \mu)$ over I through a weighted sum of function values:*

$$Q_n[f;\mu] := \sum_{i=1}^{n} w_i f(x_i) \approx I[f;\mu] \tag{3.1}$$

Where $w_i \geq 0$ and $x_i \in I$ for all $i \in \{1, 2, \dots, n\}$.

It is important to note that the weights of the quadrature $w_i \in \mathbb{R}_+$ and the weight function $\mu(x) : I \to \mathbb{R}_+$ are not the same and must not be confused with each other!

Definition 3.2 (Quadrature Terminology) *Let $Q_n[\cdot;\mu]$ be defined as above in 3.1. Then*

- *the w_i are the quadrature weights*
- *the x_i are the quadrature nodes*

This description of a quadrature can be generalized to compute $I[f]$ over any $]a, b[$ (while maintaining boundary singularities), by computing the integral over $]-1, 1[$ of the *affine pullback* $\hat{f}$ of f.

3.1.1 Affine Pullback

Let $s \in]-1,1[$, and $t \in]a,b[$, then by applying the transformation

$$\Phi(s) := \frac{1}{2}(1-s)a + \frac{1}{2}(s+1)b \tag{3.2}$$

to the quadrature nodes x_i, and scaling the weights $w_i \to \hat{w}_i$ accordingly, the following holds:

$$\int_a^b f(t)dt \approx \sum_{i=1}^{n} w_i f(x_i) = \frac{1}{2}(b-a)\sum_{i=1}^{n} \hat{w}_i \hat{f}(\hat{x}_i) \approx \frac{1}{2}(b-a)\int_{-1}^{1} \hat{f}(s)ds \tag{3.3}$$

with

$$\begin{aligned} x_i &= \Phi(\hat{x}_i) \\ w_i &= \frac{1}{2}(b-a)\cdot \hat{w}_i \\ f(\Phi(s)) &= \hat{f}(s) \end{aligned} \tag{3.4}$$

Any continuous function can thus be 'stretched' from $]-1,1[$ to $]a,b[$, the integral of which can be computed. Concrete implementations relevant to the weight functions of interest are explicitly computed in 5.2.3 and 5.3.3.

3.1.2 Errors and Precision

Given that the very definition of quadrature rules implies that they may lead to errors, the quantification of the quadrature rule's error is necessary.

Definition 3.3 (Error of a Quadrature Rule) *Let $Q_n[\cdot;\mu]$ be defined as in 3.1, then it's error is*

$$E(Q_n[\cdot;\mu]) := |Q_n[\cdot;\mu] - I[\cdot;\mu]| \tag{3.5}$$

As explained, a quadrature rule $Q_n[\cdot;\mu]$ only considers the integrand at *distinct points* within the interval of integration. Due to this fact, most quadrature rules will attempt to approximate $I[f;w]$ by computing $I[p;\mu]$, where p is a Lagragian interpolation of $\{(x_i, f(x_i))\}$. Notice that $p(x_i) = f(x_i)\ \forall i \in \{1,2,...,n\}$, thus $Q_n[f;\mu] = Q_n[p;\mu]$. This approximation of f by interpolation thus requires a *guarantee* of precision in order to be somewhat reliable.

Definition 3.4 (Order of a Quadrature Rule) *Let $Q_n[\cdot;\mu]$ be as defined in 3.1. Then the order of $Q_n[.;\mu]$ is $n \in \mathbb{N}$ such that*

$$Q_n[p;\mu] = I[p;\mu]\ ,\ \forall p \in \mathbb{P}^{n-1}$$

i.e. the order of a quadrature rule $Q_n[\cdot;\mu]$ is $n \in \mathbb{N}$ such that the integral of all polynomials of degree $n-1$ are computed exactly through $Q_n[\cdot;\mu]$.

3.2 Newton-Cotes Formulae

The Newton-Cotes method to compute n-point quadratures is by considering an *equidistant partition of* I, $\Delta = \{x_i\}_{i=1}^{n}$ and the function evaluation at these points $\{f(x_i)\}_{i=1}^{n}$.

3.2.1 Approximation through Interpolation

Newton-Cotes quadrature rules approximate $I[f;w]$ by computing $\sum_{i=1}^{n} p(x_i)w_i$, where p is a Lagrangian interpolation of f over I.

$$I[f;\mu] := \int_I f(x)\mu(x)dx \approx \int_I p(x)\mu(x)dx = \int_I (\sum_{i=1}^{n} f(x_i)l_i(x)\mu(x))dx \quad (3.6)$$

Where f's interpolation is

$$p(x) := \sum_{i=1}^{n} f(x_i)l_i(x) \in \mathbb{P}^{n-1} \quad (3.7)$$

Since the quadrature nodes are pre-defined, the weights are now the only degrees of freedom left. The following proposition offers a way to compute them, while maintaining an order of n.

Theorem 3.5 (A general formula for weights and minimal quadrature order) *Let $Q_n[\cdot;\mu]$ be an n-point Newton-Cotes polynomial quadrature rule, and $\mu(x)$ be a weight function over I. Then, $Q_n[\cdot;\mu]$ has order $\geq n \iff$ the weights are given by:*

$$w_i = \int_I \mu(x)l_i(x)dx \ , \ 0 \leq i < n \quad (3.8)$$

Where $l_i(x)$ is the associated i^{th} Lagrange polynomial (see definition 2.9).

Proof Let $p(x) \in \mathbb{P}^{n-1}$. If $Q_n[\cdot;\mu]$ is to have order $\geq n$, it will be proven that n is a *lower bound* of $Q_n[\cdot;\mu]$'s order under the following conditions:

$$I[p;\mu] = Q_n[p;\mu] := \sum_{i=0}^{n-1} w_i p(x_i)$$

Notice that $p(x) \approx \sum_{i=0}^{n-1} p(x_i)l_i(x)$, thus implying

$$I[p;\mu] = \int_I p(x)\mu(x)dx = \int_I (\sum_{i=0}^{n-1} p(x_i)l_i(x))\mu(x)dx$$

$$= \sum_{i=0}^{n-1} p(x_i) \int_I l_i(x)\mu(x)dx = Q_n[p;\mu] \iff \int_I l_i(x)\mu(x)dx = w_i \qquad \square$$

A lower bound for n-point Netwon-Cotes quadrature order has been established and below an upper bound will be introduced.

3.2.2 Maximal Order of Classical Numerical Integration

Proposition 3.6 *The order of an n-point quadrature $Q_n[\cdot;\mu]$ is at most $2n$.*

Proof Let $q(x) = \omega^2(x)$ be the squared knot poynomial (see 2.8) over a partition Δ of the integration domain I.

Then

$$Q_n[q;\mu] = \sum_{i=1}^{n} w_i q(x_i) = 0 \neq \int_I q(x)\mu(x)dw := I[q;\mu] > 0 \qquad \square$$

However, Newton-Cotes quadrature rules are restricted to *equidistant* Lagrange interpolation. This leads to a low degree of freedom in the method, which seems marginal, but can lead to enormous problems, especially if the integrand isn't well-behaved (even if it is, it will be shown that huge errors can occur). A famous example is the *Runge phenomenon*.

3.2.3 Interpolation Errors

Runge's phenomenon is a classic example of bad interpolation approximations of seemingly well-behaved functions. The task is to approximate the *Runge function*, which is given by

$$f(x) = \frac{1}{1+25x^2} \tag{3.9}$$

over the interval $]-1,1[$.

The observed result is that equidistant interpolation for $f(x)$ yields an extremely poor approximation, as seen on Figure 3.1.

This is one of the major downfalls of Newton-Cotes quadrature formulae, as the quality of the quadrature is in direct relationship to the 'compliance' of the function to equidistant interpolation. A clear way to improve the accuracy of quadrature rules is to not use an equidistant partition of I, as to focus on the 'important' areas under the curve. The next section will cover a method of choosing nodes to guarantee maximal-order quadratures.

3.3 Gaussian Quadratures

Gaussian quadratures differ from classical quadrature methods, in the sense that the quadrature nodes are not taken from an equidistant partition over I. This is such in hopes to solve the problems encountered in Newton-Cotes

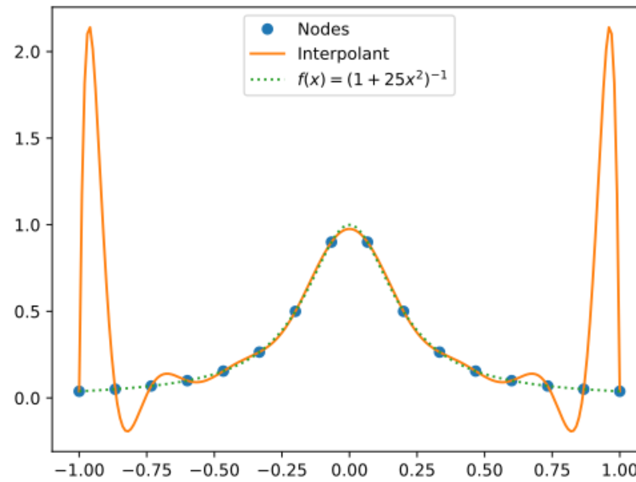


Figure 3.1: Runge Interpolation with $n = 14$, Source: [2]

quadrature rules, where the function is poorly interpolated over an equidistant partition.

The idea of approximating the integrand f with an interpolation over n points remains however the same, as well as approximating $I[f;\mu]$ by $Q_n[f;\mu]$, which is equivalent to $Q_n[p;\mu]$.

Recall that in proposition 2.16, it was stated that a weight function $\mu : I \to \mathbb{R}$ generates a sequence of *orthonormal polynomials* $\{u_n\}$ with respect to the weighted inner product

$$\langle p, q \rangle_\mu = \int_I p(x)q(x)\mu(x)dx$$

Also recall in Proposition 3.6 that the maximal order of an n-point quadrature is $2n$, an order that Gaussian quadratures achieve.

3.3.1 Maximal Order Guarantee for Gaussian Quadrature

Theorem 3.7 (Construction of a Gaussian Quadrature of order 2n) *An n-point weighted Gaussian quadrature $Q_n[\cdot;\mu]$ achieves order $2n$ if its nodes x_i correspond to the ith root of the nth orthogonal polynomial u_n associated to the weight function μ.*

Proof Let an orthonormal basis of $\mathbb{P}^n$ associated to the weight function w be denoted by $\{u_k\}_{k=0}^n$ and $p \in \mathbb{P}^{2n-1}$.

By long division, $\exists q, r \in \mathbb{P}^n$ such that $p(x) = q(x)u_n(x) + r(x)$, where q, r are of degree $< n$. Thus, to compute $I[p;\mu]$, we get the following:

$$I[p;\mu] := \int_I p(x)\mu(x)dx = \int_I q(x)u_n(x)\mu(x)dx + \int_I r(x)\mu(x)dx$$

$$= \langle q, u_n \rangle_\mu + \int_I r(x)\mu(x)dx = \int_I r(x)\mu(x)dx := I[r;\mu] = Q_n[r;\mu]$$

since $\langle q, u_n \rangle_\mu = 0$ by orthogonality with respect to μ, and $deg(r) < n$ so by Theorem 3.5:

$$Q_n[p;\mu] = Q_n[r;\mu]$$

Finally,

$$Q_n[p;\mu] = \sum_{i=1}^{n} w_i p(x_i) = \sum_{i=0}^{n} w_i q(x_i) u_n(x_i) + \sum_{i=0}^{n} w_i r(x_i)$$

$$\text{which holds} \iff \sum_{i=0}^{n} w_i q(x_i) u_n(x_i) = 0 \iff x_i \text{ are the zeros of } u_n$$

□

This is a key proof, as it also offers intuition on the inner mechanisms of Gaussian quadrature, which are now guaranteed to have maximal order if the above conditions are met.

3.3.2 Error of Gaussian Quadratures

Since it has been established that n-point Gaussian quadrature rules have a degree of $2n$, then the error term of such a rule will be:

$$|\int_I f(x)\mu(x)dx - \sum_{i=1}^{n} w_i f(x_i)| \leq \frac{f^{2n}(\eta)}{(2n)!} \langle u_n, u_n \rangle_\mu \tag{3.10}$$

for a continuous integrand f with at least $2n$ continuous derivatives, $a < \eta < b$, and u_n is the nth *monic* polynomial associated to μ (see definition 2.16).

This result is a consequence of f's Taylor expansion around η, explained in the proof of *Theorem 3.6.24* in [8].

Chapter 4

Formal Problem Statement

Now that all the foundations have been set, it is time to properly state the problem, along with the offered solution.

Let $I =]-1, 1[\in \mathbb{R}$, $f : I \to \mathbb{R}$ a well-behaved continuous function, and $\mu : I \to \mathbb{R}_+$ a singular weight function. *Find the nodes* $x_i \in I$ *and the weights* $w_i \in \mathbb{R}_+$ *yielding a maximal order Gaussian quadrature rule* $Q_n[\cdot; \mu]$ such that:

$$\int_I f(x)\mu(x)dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{4.1}$$

More specifically, $\mu(x)$ will only take on two forms[1] (denoted w and v), to which the details will be specified in the next chapter.

$$w(x) = (1-x)^\alpha (1+x)^\beta \tag{4.2}$$

$$v(x) = \ln(1+x) \tag{4.3}$$

4.1 Singular Gaussian Quadrature & the Eigenvalue Problem

This section will cover the theoretical aspect of solving the stated problem with the help of the Eigenvalue Problem (see definition 2.18).

[1]Generalizations of these weight functions are offered in 5.2.3 and 5.3.3 respectively. However, the basic problem over $]-1, 1[$ is the most common case.

4.1.1 Computing the Nodes

It has been shown that Gaussian quadratures and sequences of orthogonal polynomials are very tightly related. It has however yet to be shown how this helps solve the problem at hand, i.e. how to compute the nodes, i.e. how to find zeros of orthogonal polynomials.

Recall the 3-term recurrence relation (2.16) of orthonormal polynomials $\{u_n\}$ with respect to a weight function μ:

$$\beta_{n+1}u_{n+1} = (x - \alpha_{n+1})u_n - \beta_n u_{n-1} \tag{4.4}$$

which can be rewritten as

$$xu_n = \alpha_{n+1}u_n + \beta_n u_{n-1} + \beta_{n+1}u_{n+1} \tag{4.5}$$

Furthermore, if $\{u_n\}$ are considered to be *monic* (leading coefficient = 1), then equations 4.4 and 4.5 can be further simplified to

$$u_{n+1} = (x - a_{n+1})u_n - b_n u_{n-1} \tag{4.6}$$

and

$$xu_n = a_{n+1}u_n + b_n u_{n-1} + u_{n+1} \tag{4.7}$$

Let $\tilde{u} = (u_0, u_1, ..., u_{n-1})^T$, and e_n be the nth canonical basis vector. Equation 4.5 can be written as a matrix-vector product:

$$x\tilde{u} = A\tilde{u} + u_n e_n \tag{4.8}$$

where

$$A = \begin{bmatrix} a_0 & 1 & & & & \\ b_1 & a_1 & 1 & & & \\ & b_2 & a_2 & 1 & & \\ & & b_3 & \ddots & \ddots & \\ & & & \ddots & \ddots & 1 \\ & & & & b_{n-1} & a_n \end{bmatrix} \tag{4.9}$$

Consider $x_i \in I$ to be the ith root of $u_n(x)$, and consider evaluating equation 4.8 at $x = x_i$. Its last term thus vanishes, and it reduces to:

$$x_i\tilde{u}(x_i) = A\tilde{u}(x_i) \tag{4.10}$$

where $\tilde{u}(x_i) = (u_0(x_i), u_1(x_i), ...u_n(x_i))^T$. Notice that this forms an eigenvector problem, and implying that x_i is an eigenvalue of A.

By Theorem 3.7, it is now safe to conclude that *the eigenvalues of the tridiagonal matrix A in 4.9 correspond to the quadrature nodes of an n-point Gauss quadrature rule with respect to weight function μ.*

4.1.2 Computing the Weights

Let $\mu, \{u_n\}, a_n, b_n, A$ be defined as above.

Definition 4.1 *The Christoffel function of a sequence of orthonormal polyomials* $\{u_n\}$ *relative to a weight function* μ *is defined by:*

$$\Lambda_n(x) = (\sum_{k=0}^{n-1} |u_k(x)|^2)^{-1} \tag{4.11}$$

Theorem 4.2 *Let* x_i *be the eigenvalues of A. Then the weights* w_i *of an n-point Gauss quadrature can be computed through the Christoffel function as follows:*

$$\Lambda_n(x_i) = w_i \tag{4.12}$$

Proof Provided as the proof of *Satz 40.3* in [7] .

This relation yields a direct method to compute the quadrature weights.

4.2 Golub-Welsch Algorithm

As shown above, building a Gauss quadrature rule can be boiled down to a few steps. The Golub-Welsch Algorithm is the method that will be used to efficiently compute Gaussian quadrature nodes and weights. It is comprised of 3 steps:

1. computing the 3-term recurrence relation coefficients a_i, b_i relative to the desired weight function
2. computing the eigenvalues x_i by finding the eigenvalues of 4.9
3. computing the weights w_i

However, there are still many aspects of this computation that can be discussed, namely optimising the process.

First of all, finding the eigenvalues of matrix 4.9 may be tedious work, and it is preferable to solve a *Hermitian* eigenvalue problem (see definition 2.21). Consider the *Jacobi Matrix* $\mathcal{J}$:

$$\mathcal{J} = \begin{bmatrix} a_0 & \sqrt{b_1} & & & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & & & \\ & \sqrt{b_2} & a_2 & \sqrt{b_3} & & \\ & & \sqrt{b_3} & \ddots & \ddots & \\ & & & \ddots & \ddots & \sqrt{b_{n-1}} \\ & & & & \sqrt{b_{n-1}} & a_n \end{bmatrix} \tag{4.13}$$

which is symmetrical (or self-adjoint) and *similar* to that of A in 4.9. Then by similarity, the eigenvalues of A match with those of $\mathcal{J}$. (The derivation from A to $\mathcal{J}$ are explained in Chapter 5.3 on Gaussian Quadrature in [1]).

As seen in definition 2.21 of self-adjoint matrices , it is computationally cheaper to find eigenvalues of symmetrical matrices, which is why this algorithm favors it.

Secondly, computing the weights via the Christoffel function is highly inefficient. It must also be considered that the polynomials themselves may be unknown, given that only the coefficients of the recurrence relation are required to compute the nodes.

The Golub-Welsch Algorithm thus offers am optimized routine to compute the quadrature nodes and weights.

Proposition 4.3 *Let x_i be the ith eigenvalue of the tridiagonal matrix obtained by step 2, and $v_i = (v_{i,0}, v_{i,1}, ..., v_{i,n-1})^T$ be its corresponding unitary eigenvector, then:*

$$w_i = \mu_0 \cdot (v_{i,0})^2 \tag{4.14}$$

where

$$\mu_0 = \int_I \mu(x)dx \tag{4.15}$$

Proof Found in code implementation of Gauss quadrature solvers in [5] and in Chapter 5.3 on Gaussian Quadratures in [1]. Further understanding of this proposition is not necessary for the code implementation. □

Chapter 5

Singular Weight Functions

The computation of weighted integrals with a non-singular weight function is a rather straightforward process, and unless the integrand itself is not well-behaved, no particular problems arise. Singular weight functions, however, most often lead to the computational problem in which some regions may not be well-defined. The inaccuracy of this computation may be amortized using associated Gaussian quadrature rules, which base themselves off of the well-defined associated polynomials of the singular weight function. As explained in Section 3.3, the quadrature nodes and weights are fully determined by the polynomials, which are *guaranteed* to exist.

This section will present the necessary steps to compute the recurrence relation coefficients of both weight functions of interest.

5.1 Jacobi Weight Function

Definition 5.1 *The Jacobi weight function is defined over* $]-1,1[$ *by:*

$$w(x) = (1-x)^{\alpha}(1+x)^{\beta} \text{ ,where } \alpha, \beta > -1 \tag{5.1}$$

First of all, it is clear that if $\alpha, \beta \geq 0$, then the weight function will lose its singularity property. For this reason, a general solver will be implemented.

Dealing with $w(x)$ is a rather straightforward process, as the Jacobi weight function is very well researched and documented.[1]

5.1.1 Jacobi Polynomials

The (non-monic) polynomials associated to $w(x)$ are called *Jacobi polynomials*, and the nth polynomial is often denoted $P_n^{\alpha,\beta}(x)$. Their analytical form is

[1] Namely the orthogonal polynomials associated to $w(x)$ are one of the most widely used orthogonal polynomials.

known in multiple forms, but is not relevant to the quadrature computation of interest.

The inner product of Jacobi Polynomials with respect to $w(x)$ is defined as follows:

$$\int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}P_n^{\alpha,\beta}(x)P_m^{\alpha,\beta}(x)dx = \frac{2^{\alpha+\beta+1}\Gamma(n+\alpha+1)\Gamma(n+\beta+1)}{(2n+\alpha+\beta+1)\Gamma(n+\alpha+\beta+1)}\delta_{nm} \tag{5.2}$$

Note that $\langle P_n^{\alpha,\beta}(x)P_m^{\alpha,\beta}(x)\rangle_w \neq \delta_{nm}$, as these polynomials are *orthogonal, but not orthonormal*.

From the above equation, a very useful value can be derived: the *first moment* μ_0: [2]

$$\mu_0 = \int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}dx = \int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}P_0^{\alpha,\beta}(x)P_0^{\alpha,\beta}(x)dx \tag{5.3}$$
$$= \frac{2^{\alpha+\beta+1}\Gamma(\alpha+1)\Gamma(\beta+1)}{(\alpha+\beta+1)\Gamma(\alpha+\beta+1)} = \frac{2^{\alpha+\beta+1}\Gamma(\alpha+1)\Gamma(\beta+1)}{\Gamma(\alpha+\beta+2)}$$

Definition 5.2 (Recurrence Relation Coefficients) *Let $u_n^{(\alpha,\beta)}$ denote the nth polynomial associated to the Jacobi weight function, then the* 3-term recurrence relation *(see Proposition 2.16) describing monic $u_n^{(\alpha,\beta)}$ is given by:*

1. $u_{-1}^{(\alpha,\beta)} = 0$
2. $u_0^{(\alpha,\beta)}(x) = 1$
3. $u_{n+1}^{(\alpha,\beta)} = (x-a_{n+1})u_n^{(\alpha,\beta)} - b_n u_{n-1}^{(\alpha,\beta)}$

In this case, the coefficients a_{n+1} and b_{n+1} are given by:

1. $a_{n+1} = \frac{(\alpha^2-\beta^2)}{(2n+\alpha+\beta)(2n+\alpha+\beta-2)}$
2. $b_{n+1} = \frac{4n(n+\alpha)(n+\beta)(n+\alpha+\beta)}{(2n+\alpha+\beta)^2(2n+\alpha+\beta+1)(2n+\alpha+\beta-1)}$

The above formulae for the coefficients were found in the code implementation of Gauss-Jacobi quadrature in [5]. A manual computation or derivation of these coefficients adds no value to the paper, nor to the code implementation, as the source is deemed to be reliable and has been proven to be efficient.

5.2 Special Cases of the Jacobi Weight Function

The Jaocbi weight function is highly modulable, due to its two parameters, and here are some important special cases.

[2] Obtained via properties of the Γ-function, which will not be discussed in this paper.

5.2.1 Legendre Weight Function

The trivial case of $\alpha = \beta = 0$ yields $w(x) = 1$, which is the 'normal' weight function i.e. unweighted Euclidean space. The polynomials associated to this trivial weight function are the *Legendre Polynomials*.

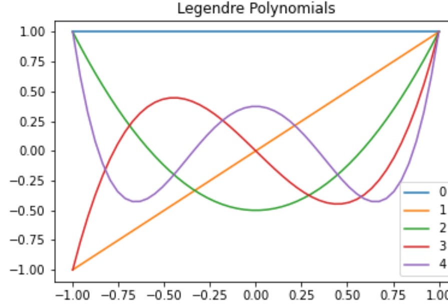


Figure 5.1: Legendre polynomials of degrees 0 to 4

5.2.2 Chebyshev Weight Function

Another symmetrical case where $\alpha = \beta = -\frac{1}{2}$ yields the *Chebyshev weight function* $\frac{1}{\sqrt{(1-x^2)}}$ (see Figure A.7), whose associated polynomials are the famous *Chebyshev Polynomials* of the first kind. The nodes of these polynomials are $x_i = \cos(\frac{(2k-1)\pi}{2n})$, for $k \in \{1, 2, ...n\}$.

5.2.3 Affine Pullback

The initial integral is:

$$I = \int_{-1}^{1} (1-x)^{\alpha}(1+x)^{\beta} f(x)dx \tag{5.4}$$

Using the affine pullback definition in 3.1.1, and requiring singularities at the new borders, (i.e. a and b) the new integral I' becomes

$$I' = \int_{a}^{b} (x-b)^{\alpha}(x-a)^{\beta} f(x)dx \tag{5.5}$$

$$= (\frac{b-a}{2})^{1+\alpha+\beta} \int_{-1}^{1} (s-1)^{\alpha}(s+1)^{\beta} f(\Phi(x))ds \tag{5.6}$$

Which can be compute simply with scaled version of a Gauss-Jacobi quadrature rule $Q_n[\hat{f}, w]$ over I, where $\hat{f}(s) = f(\Phi(x))$.

5.3 Logarithmic Weight Function

The second weight function of interest is a logarithmic one. Since there is no 'classical' logarithmic weight function, it has been decided to study the following:

Definition 5.3 (Logarithmic Weight Function) *The logarithmic weight function is defined over* $]-1,1[$ *and is given by:*

$$v(x) = \ln(x+1) \tag{5.7}$$

Using this weight function, the integral is given as:

$$I[f,v] = \int_{-1}^{1} f(t)v(t)dt = \int_{-1}^{1} f(t)\ln(t+1)dt \tag{5.8}$$

Notice that v changes signs over I, which makes computations of Gaussian quadratures rather ineffective. To remedy this, $I[f,v]$ is split into a sum of two integrals, at $t_0 = 0$, i.e. where v changes signs.

$$I[f,v] = \int_{-1}^{0} f(t)\ln(t+1)dt + \int_{0}^{1} f(t)\ln(t+1)dt \tag{5.9}$$

$$= \int_{0}^{1} f(t-1)\ln(t)dt + \int_{0}^{1} f(t)\ln(t+1)dx$$

$$= \int_{0}^{1} f(t)\ln(t+1)dx - \int_{0}^{1} f(t-1)\ln(\frac{1}{t})dx \tag{5.10}$$

The first term of the addition has no singularities (assuming f is well-behaved), so equation 5.7 can be rewritten as:

$$I[f,v] \approx Q_n[f,v] = \sum_{i=1}^{n}(\ln(x_i+1)f(x_i))w_i - \sum_{i=1}^{n}\tilde{w}_i f(\tilde{x}_i - 1) \tag{5.11}$$

$$= Q_{1,n}[\ln(x+1)f(x)] - Q_{2,n}[f(x-1),\tilde{v}]$$

Which reduces to the addition of two quadrature rules:

- a Gauss-Legendre rule $Q_{1,n}[\ln(x+1)f(x)]$ over $]0,1[$, which can easily be computed using a Gauss-Jacobi quadrature rule and affine pullback
- a *special* quadrature rule $Q_{2,n}[f(x-1),\tilde{v}]$ over $]0,1[$ with $\tilde{v}(t) = ln(\frac{1}{t})$, nodes $\tilde{x}_i$ and weights $\tilde{w}_i$. This *special* quadrature rule is subtler in its computation, and will be subject to more fine calculations.

The *special* quadrature rule is presented in Example 2.27 of [4]. The recurrence relation coefficients (α_n, β_n) of $\tilde{v}$'s associated orthogonal polynomials are solved via the *Modified Chebyshev Algorithm* presented below.

5.3.1 Modified Chebyshev Algorithm

In Chapter 2 of [4], W. Gautschi presents *moment-based methods* to compute the recurrence relation coefficients of the orthogonal polynomials associated to an arbitrary weight function $\mu(x)$ (*referred to as 'measure' in the book*).

Let $\pi_n(t)$ be the nth polynomial associated to $\mu(x)$, and $\{p_k(x)\}$ be a system of monic polynomials that satisfy the *three-term recurrence relation*. Let $(\alpha_n, \beta_n), (a_n, b_n)$ be the respective recurrence relation coefficients of π, p:

- $\pi_{n+1}(t) = (t - \alpha_n)\pi_n(t) - \beta_n \pi_{n-1}(t)$
- $p_{n+1}(t) = (t - a_n)p_n(t) - b_n p_{n-1}(t)$

Definition 5.4 (Moment / Modified Moment) *Let*

- $\mu_r = \int_{\mathbb{R}} t^r w(t)dt$ *be w's* r^{th} *moment, and*
- $m_k = \int_{\mathbb{R}} p_k(t)w(t)dt$ *be w's* k^{th} *modified moment*

Methods to compute m_k are given in section 3 of [6], although the code implementations of [3] differ.[3]

Let $m = [m_0, m_1, ..., m_{2n-1}]$, $\{a_k, b_k\}_{k=0}^{2n-1}$ as defined above be the inputs to the Modified Chebyshev Algorithm. The algorithm proceeds as follows: *Initialization:*

$$\begin{aligned} \alpha_0 &= a_0 + \frac{m_1}{\sigma_{m_0}} \\ \beta &= m_0 \\ \sigma_{-1,l} &= 0 \ , \ l \in \{1, 2, ..., 2n-2\} \\ \sigma_{0,l} &= m_l \ , \ l \in \{0, 1, ..., 2n-1\} \end{aligned} \tag{5.12}$$

Where σ is a $(n+2)$x$2n$ matrix, used as a stencil to allow reuse of previous computations.

For $k \in \{1, 2, ..., n-1\}$:

For $l \in \{k, k+1, ..., 2n-k-1\}$

$$\begin{aligned} \sigma_{k,l} &= \sigma_{k-1,l+1} - (\alpha_{k-1} - \alpha_l)\sigma_{k-1,l} \\ &\quad - \beta_{k-1}\sigma_{k-2,l} + b_l \sigma_{k-1,l-1} \\ \alpha_k &= a_k + \frac{\sigma_{k,k+1}}{\sigma_{k,k} - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}}} \\ \beta_k &= \frac{\sigma_{k,k}}{\sigma_{k-1,k-1}} \end{aligned} \tag{5.13}$$

The output is the sequence of coefficients $\{\alpha_k, \beta_k\}_{k=0}^{n-1}$ as defined above.

[3]The manual computation and further explanations of the modified moments is outside of the scope of this thesis. The implementation follows the methods presented in the MATLAB repository [3].

5.3.2 Computing the Quadrature

Applying this knowledge to $v(x)$ i.e. $\tilde{v}(x)$, it is now possible to approximate $I[f,v]$ in the following manner:

1. Let $g(x) = \ln(x+1)f(x)$, and $\tilde{f}(x) = f(x-1)$
2. Compute the Gauss-Legendre quadrature rule $Q_{1,n}[g]$ over $]0,1[$
3. Let $\{p_k\} = \{P_k^{\alpha,\beta}\}$ as defined in 5.2
4. Compute the first $2n$ modified moments $\{m_k\}$ of $\tilde{v}$ w.r. to $\{p_k\}$
5. using $\{a_k, b_k\}_{k=0}^{2n-1}$ and $\{m_k\}_{k=0}^{2n-1}$, find $\{\alpha_k, \beta_k\}_{k=0}^{n-1}$ via the modified Chebyshev Algorithm
6. Find the special quadrature rule $Q_{2,n}[\tilde{f}, \tilde{v}]$ via Golub-Welsch
7. $Q_n[f,v] = Q_{1,n}[g] - Q_{2,n}[\tilde{f}, \tilde{v}]$ [4]

5.3.3 Affine Pullback

Starting from the initial integral:

$$I = \int_{-1}^{1} \ln(x+1)f(x)dx \tag{5.14}$$

and generalizing it by conserving border singularities, I' becomes:

$$I' = \int_{a}^{b} \ln(x-a)f(x)dx \tag{5.15}$$

Let $s \in]0,1[$, then $x = s(b-a)+a$ and $dx = (b-a)ds$ yields the following:

$$I' = (b-a)\int_{0}^{1} \ln(s(b-a))f(s(b-a)+a)ds \tag{5.16}$$

$$= (b-a)(\ln(b-a)\int_{0}^{1} f(\hat{s})ds + \int_{0}^{1} \ln(s)f(\hat{s})ds) \tag{5.17}$$

with

$$\hat{s} = (b-a)s + a, d\hat{s} = (b-a)ds \tag{5.18}$$

Equation 5.17 is again in two parts, one with no singularity, and one weighted with $\tilde{v}(x)$. This is also solvable via the addition of a Gauss-Legendre and a *special* quadrature rules, explained in subsection 5.3.2.

[4] Note that $Q_n[\cdot, v]$ is the addition of two n-point quadratures

Chapter 6

Code Implementation: The SingGQ Library

The goal of this paper is to compute Gaussian quadratures. This has been done in the form of a C++ library, which offers a means of computing maximal order Gaussian quadrature rules w.r.t. to the aforementioned weight functions. The developed library is not a standalone, as it depends on `Eigen` and `Boost`, both of which are popular C++ libraries for scientific computing.

6.1 The GaussRule Class

Given the solving methods presented in the previous chapters, and their similarities it is only natural to implements an abstract template super-class with shared functions. These shared functions are namely the Jacobi recurrence relation coefficient computation, and the tridiagonalisation process. Its signature is as follows:

```
1 template<class T> //template for different precision types (float, double)
2 class GaussRule{
3     public:
4         virtual Eigen::Matrix<T, Eigen:Dynamic, Eigen:Dynamic> c_jacobi(↩
              std::size_t n, double a, double b);
5
6         virtual Eigen::Matrix<T, Eigen::Dynamic,Eigen::Dynamic> ↩
              tridiagCoeffs(Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>↩
               coeffs, std::size_t n);
7 }
```

The function `c_jacobi` returns a $nx2$ array containing the n first recurrence relation coefficients of the Jacobi Polynomials (a_k, b_k) (see 5.2). These coefficients are computed via the Gautschi routines from [3].

Trivially, `tridiagcoeffs` simply puts the (a_k, b_k) coefficients obtained from `c_jacobi` into a tridiagonal matrix of the form 4.13.

6.1.1 Namespaces

Two namespaces were added: `GQJacobi` and `GQLog`, to be able to differentiate between both in an easier manner. Moreover, the Gauss-Jacobi solver has implemented sub-classes, making the user-code more readable.

6.2 The GaussJacobiRule Class

This class computes and handles computations related to a Gaussian quadratures with respect to a Jacobi weight function (5.1), also called *Gauss-Jacobi quadratures*.

As stated in section 4.2, the first step to creating an n-point Gauss-Jacobi quadrature rule is to specify α, β, and n. These parameters are the construction parameters of the `GaussJacobiRule`, which is a class *template*, as to allow a higher-precision type – analogous to `double` – be the type of choice. Its members, accessors and constructors have the following signatures:

```
1  template <class T>
2  class GaussJacobiRule{
3      protected:
4          // members
5          std::vector<T> nodes;
6          std::vector<T> weights;
7          std::size_t degree; // number of evaluation points
8
9      public:
10         // basic accessors
11         std::size_t getDeg() const;
12         std::vector<T> getN() const; // for nodes
13         std::vector<T> getW() const; // for weights
14
15         //constructors
16         GaussJacobiRule(); // default
17         GaussJacobiRule(const GaussJacobiRule& gq); // copy
18         GaussJacobiRule(std::size_t n, double a, double b);
19
20         // operators
21         template<typename F>
22         T operator ()(F f) const;
23
24         template<typename F>
25         T operator ()(F f, T a, T b) const
26
27         // member functions
28         double gamma_zero(double a, double b)
29         Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> nw();
30 }
```

The interesting constructor being the last, it first ensures the validity of the arguments, and then computes and stores the nodes, resp. the weights associated to the desired quadrature rule right away. It does so by calling the member function `nw()`.

6.2.1 Computing the nodes and weights

```
1  Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> nw() {
2
3      double gamma_0 = gamma_zero();
4      std::size_t n = this->degree;
5      Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> coeffs = this->↩
           c_jacobi(n, this->alpha, this->beta);
6      Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> J_n = this->↩
           tridiagCoeffs(coeffs, n);
7
8      Eigen::SelfAdjointEigenSolver<Eigen::Matrix<T,Eigen::Dynamic, Eigen::↩
           Dynamic>> solve(J_n);
9
10     Eigen::Vector<T, Eigen::Dynamic> nodes = solve.eigenvalues().real();
11     Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> eigenvecs = solve.↩
           eigenvectors().real();
12
13     // Solving the weights using Golub-Welsch algorithm formula
14     Eigen::Vector<T, Eigen::Dynamic> weights = Eigen::Vector<T, Eigen::↩
           Dynamic>::Zero(n);
15     for(int i = 0; i < n; i++){
16         weights[i] = gamma_0*pow(eigenvecs.col(i).normalized()[0], 2);
17     }
18
19     // Preparing more compact return type
20     Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> nw = Eigen::Matrix<T,↩
            Eigen::Dynamic, Eigen::Dynamic>::Zero(n, 2);
21     nw.col(0) = nodes;
22     nw.col(1) = weights;
23
24     return nw;
25
26 }
```

This method computes the quadrature nodes and weights following the Golub-Welsch (4.2) algorithm. Lines 5 and 6 pre-compute and setup the recurrence relation coefficients in the 4.13 tridiagonal form. The call to `.real()` in lines 10 and 11 are justified by 2.17. As explained in 2.21, a hermitian solve is faster, so line 8 calls an optimized hermitian matrix eigenvalue solver.

`gamma_0` computes the first moment of the corresponding weight function via equation 5.3.

```
1  double gamma_zero(double a, double b)
2      { return (pow(2, a+b+1)*tgamma(a+1)*tgamma(b+1))/(tgamma(a+b+2)); }
```

Unlike most solvers, the end-result does not necessarily yield a *sorted* sequence of nodes x_i such that $x_i < x_{i+1}$, but it was deemed unnecessary to sort, as the integrands of interest are real functions.

6.2.2 Evaluating the integral

Finally, the `()` operator, which allows the evaluation of the quadrature for a given function, passed as a parameter.

```
1 template<typename F>
2 T operator ()(F f) const {
3     T quad = 0;
4     for(std::size_t i = 0; i < degree; i++){
5         quad += (weights[i] * std::real(f(nodes[i]))) ; //casting to real
6         }
7     return quad;
8 }
```

The cast to real numbers is required, as some functions with `cmath`, when called with `<double>` return complex values, with no imaginary part.

6.2.3 Different integral boundaries

The previous operator is also overloaded to support different integration boundaries, by applying affine pullback (see 3.3) to the Jacobi weight function, as explained in 5.2.3. The following operator takes as arguments the integrand and the integration bounds $]a, b[$.

```
1 template<typename F>
2 T operator ()(F f, T a, T b) const {
3 T quad = 0;
4 for(std::size_t i = 0; i < degree; i++){
5     T x_i = 0.5*((1-nodes[i])*a + (1+nodes[i])*b);
6     quad += (weights[i] * std::real(f(x_i))) ; }
7     // affine pullback sccaling ratio
8     quad *= pow((0.5*(b-a)), 1+this->getAlpha()+this->getBeta());
9 return quad; }
```

6.2.4 Practical Subclasses

The Jacobi weight function is quite broad, and has many very widespread and useful special cases (examples in 5.1). Such special cases have been modelled as subclasses of `GaussJacobiRule`; namely into `GausLegendreRule` and `GaussChebyshev`. The former simply represents 'unweighted' integration and its only construction parameter is the n.

```
1 template<typename T>
2 class GaussLegendreRule : public GaussJacobiRule<T>{
```

```
3       public:
4           GaussLegendreRule(std::size_t n)
5           : GaussJacobiRule<T>(n, 0, 0)
6           {}
7   };
```

It holds no extra members.

The latter represents a Gauss-Chebyshev rule, where the weight function is given by either

- $\frac{1}{\sqrt{1-x^2}}$, called a Chebyshev weight function of the *first kind*
- $\sqrt{1-x^2}$, called a Chebyshev weight function of the *second kind*

```
1   template<typename T>
2   class GaussChebyshevRule : public GaussJacobiRule<T>{
3       private:
4           int sgn; // indicates the sign of the exponents
5
6       public:
7           GaussChebyshevRule(std::size_t n, int sgn)
8           : GaussJacobiRule<T>(n, sgn*0.5, sgn*0.5)
9           {
10              this->sgn = sgn;
11          }
12
13          int getSgn() const{
14              return this->sgn;
15          }
16  };
```

It is thus justified to have an extra member sgn that will indicate the sign of the exponent.

6.3 The GaussLogRule Class

This class computes and handles computations related to a Gauss quadrature with logarithmic weight function (5.7). It highly resembles `GaussJacobiRule` in structure, but has extra member methods to compute subtleties, as mentioned in 5.3.

```
1   template<typename T>
2   class GaussLogRule : public GaussRule<T>{
3       protected:
4           std::vector<T> nodes;
5           std::vector<T> weights;
6           std::size_t degree;
7
8       public:
9           //basic accessors
10          std::size_t getDeg() const;
```

```
11         std::vector<T> getN() const;
12         std::vector<T> getW() const;
13 
14         //constructors
15         GaussLogRule(); // default
16         GaussLogRule(const GaussLogRule& gql); // copy
17         GaussLogRule(std::size_t n);
18 
19         //operators
20         template<typename F>
21         T operator()(F f) const;
22 
23         template<typename F>
24         T operator()(F f, T a, T b) const;
25 
26         //member functions
27         Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> shifted_c_log(std↩
               ::size_t n);
28 
29         Eigen::Vector<T, Eigen::Dynamic> mmom_log(std::size_t n);
30 
31         Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> chebyshev(std::↩
               size_t n,Eigen::Vector<T, Eigen::Dynamic> mom, Eigen::Matrix<T↩
               , Eigen::Dynamic, Eigen::Dynamic> abj;
32 
33         Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic> nw(std::size_t n)
34 }
```

6.3.1 Computing the special quadrature nodes and weights

- `shifted_c_log` computes the shifted recurrence realtion coefficients of the Jacobi weight function (a_n, b_n as defined in 5.3.1). It follows the routine `r_jacobi01` of [3].
- `mmom_log` computes the modified moments of the special quadrature, explained in 5.4.
- `chebyshev` is the entire process explained in 5.3.1. Explicitly, it follows the algorithm in section 2.1.7 of [4].
- `nw` computes the nodes and weights of the *special quadrature rule* (5.3.2), and is the same process 4.2 as 6.2.1, but the α, β coefficients are the result of `chebyshev`

6.3.2 Evaluating the integral

The () operator works in a different manner than that of `GaussJacobiRule`, as the computation of the quadrature is fundamentally different (5.3.2). The first part is computed via an instance of `GaussLegendreRule`, and the second part via the *special quadrature rule* mentioned above.

```
1  template<typename F>
2  T operator()(F f) {
3      T quad = 0;
4
5      //Gauss-Legendre part
6      GQJacobi::GaussLegendreRule<T> glg(this->degree);
7      quad += glg([&](T x){ return log(x+1)*f(x); }, 0, 1);
8
9      // logarithmic part
10     for(std::size_t i = 0; i < degree; i++){
11         quad -= (weights[i] * std::real(f(nodes[i]-1))) ;
12     }
13     return quad;
14 }
```

6.3.3 Different integral boundaries

This operator takes the integrand, and the integration boundaries $]a,b[$ as parameters, and computes the new integral via the affine pullback (generalization of 3.3 to]0,1[), explained in subsection 5.3.3.

```
1  template<typename F>
2  T operator()(F f, T a, T b) {
3      assert(a < b); // boundary condition check
4      T quad = 0;
5      GQJacobi::GaussLegendreRule<T> glg(this->degree); // Gauss-Legendre ↩
           computation
6      quad += glg([&](T x){return f((b-a)*x + a);}, 0, 1);
7      quad *= log(b-a); // scaling
8      // evaluation of second half of the integral using the special ↩
           quadrature rule
9      for(std::size_t i = 0; i < degree; i++){
10         quad -= (weights[i] * std::real(f(nodes[i]*(b-a)+a))) ;
11     }
12     return (b-a)*quad; // last scaling due to affine pullback
13 }
```

6.4 Runtine Analysis

A runtime test for generation and evaluation of different Gauss quadrature rules has been done.

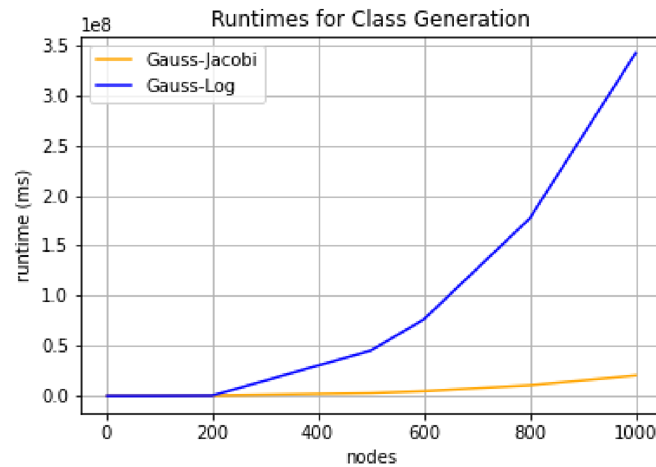


Figure 6.1: Jacobi vs. Logarithmic weight function generation

As expected, the logarithmic weight function yields more runtime, as the computations are more fine, and involve matrix generations, on size $\mathcal{O}(n^2)$ for an $n-$point quadrature rule.

Chapter 7

Accuracy Analysis

The accuracy of the developed Gaussian quadrature rule will be tested in two ways:

- Perfect accuracy for polynomial integrands[1]
- Witness exponential convergence against results of another quadrature calculator with 'smooth' integrands

7.1 Polynomial Exactness

7.1.1 Gauss-Jacobi Polynomial Exactness

The n-point Gauss-Jacobi quadrature rule in the `SingGQ` library does indeed provide polynomial exactness. Table 7.1.1 of computations confirms this claim.

The error was computed by `fabs(exp-obtained)`, where

$$\texttt{exp} = \int_{-1}^{1} x^{2n-2} dx = \frac{2}{2n-1} \tag{7.1}$$

is the analytical result, and `obtained` is the quadrature computation with an instance of `SingGQ::GQJacobi::GaussJacobiRule<double>`.

Note that this shows exactness for polynomials of degree $2n-2$, as $2n-1$ degree polynomials always have a surface of 0 over I, due to symmetry with respect to the y-axis. This property also holds with `SingGQ`.

[1] i.e. reaching numerical errors around machine epsilon, which is about 10^{-15} for double precision in the IEEE 754 Standard

n	obtained	error
2	0.66666666666666640761	2.22044604925031308808e-16
3	0.39999999999999980016	2.22044604925031308808e-16
4	0.28571428571428503229	6.66133814775093924254e-16
5	0.22222222222222193233	2.77555756156289135111e-16
6	0.18181818181818173996	8.32667268468867405320e-17
7	0.15384615384615368816	1.66533453693773481063e-16
8	0.13333333333333326931	6.38378239159465010742e-16
9	0.117647058823531242	1.83186799063150829173e-15
25	0.040816326530611561629	6.80011602582908381016e-16
55	0.0.018348623853213244517	2.23432383705812753763e-15
109	0.0.0092165898617522726971	1.120631365481017383e-15
239	0.0041928721173942638464	6.1556662545036999745e-15
540	0.0018535681186340202415	5.66061954254681865e-15

Table 7.1: Computational results of Jacobi moments and their error

7.1.2 Gauss-Log Polynomial Exactness

A similar test as that in 7.1.1 can be done for the weight function $v(x)$ as defined in 5.7. However, note that the quadrature rule with respect to $v(x)$ is comprised of the addition of two quadrature rules (see 5.3.2). The Gauss-Legendre quadrature rule has already been confirmed to by polynomial-accurate in 7.1.1. For this reason, the polynomial check will only be done with respect to the *special quadrature rule*. Table 7.1.2 confirms that it indeed yields perfect accuracy for polynomial integrands.

From the runtime tests in 6.4, it was decided to reduce the number of quadrature points, to avoid unnecessarily lengthy computations.

Again, the error was computed by `fabs(exp-obtained)`, where

$$\texttt{exp} = \int_0^1 \ln(x) x^n dx = -\frac{1}{(n+1)^2} \tag{7.2}$$

is the analytical result, and `obtained` is the quadrature computation with an instance of `SingGQ::GQLog::GaussLogRule<double>`.

7.2 Validation with Smooth Integrands

The quadrature calculator that was tested against is `scipy.integrate.quad`, as `scipy` is a very reputable library in the realm of scientific computing. The functions being tested were the following: $e^x, \cos(x), \frac{1}{1+x^2}$. All weighted

n	obtained	error
2	-0.11111111111111109107	1.38777878078144567550e-17
10	-0.00826446280991731396790	4.16333634234433702660e-17
15	-0.00390625000000000164799	1.64798730217796673970e-17
24	-0.00160000000000000527689	5.26922255827955154930e-17
30	-0.00104058272632674849410	5.42101086242752217e-18
43	-0.00051652892561985228916	1.75640751942651718310e-17
50	-0.00038446751249519178252	2.38524477946810975480e-18
100	-9.80296049407037103530e-05	1.16280682999070350550e-17
200	-2.47518625776737683480e-05	1.47993596544271355240e-17
240	-1.72173344122980253560e-05	1.17364885171555854980e-17

Table 7.2: Computational results of logarithmic moments and their error

with the Jacobi (respectively logarithmic) function, of course, and the tests were repeated 3 times, with different weight function parameters.

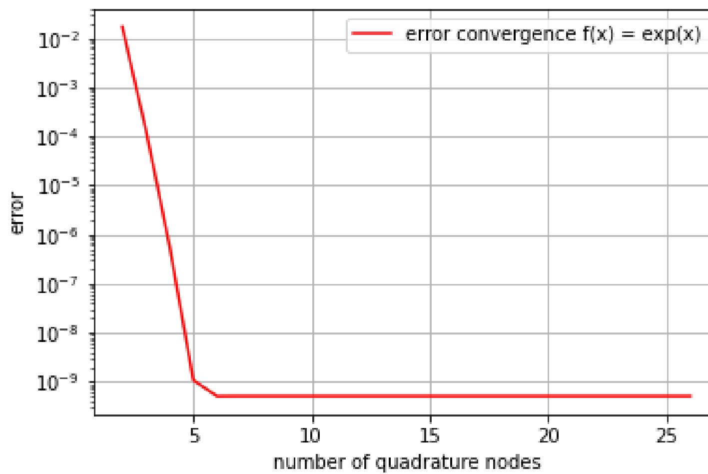


Figure 7.1: Error plot with $f(x) = e^x, \alpha = -0.7, \beta = -0.1$

First of all, it can be observed in Figure 7.1 that for a large n, the `GaussJacobiRule` quadrature rule exponentially converges for $f(x) = e^x$. Results have been plotted on a semilogarithmic scale for better observations of convergence.

In both cases, the resulted convergence is very satisfactory for the $\cos(x)$ and e^x functions, as a worst-case error under `1e-9` is attained with under 10 quadrature nodes. The last function, $\frac{1}{1+x^2}$, requires more quadrature nodes ($n \approx 25$) to reach an error of `1e-9`.

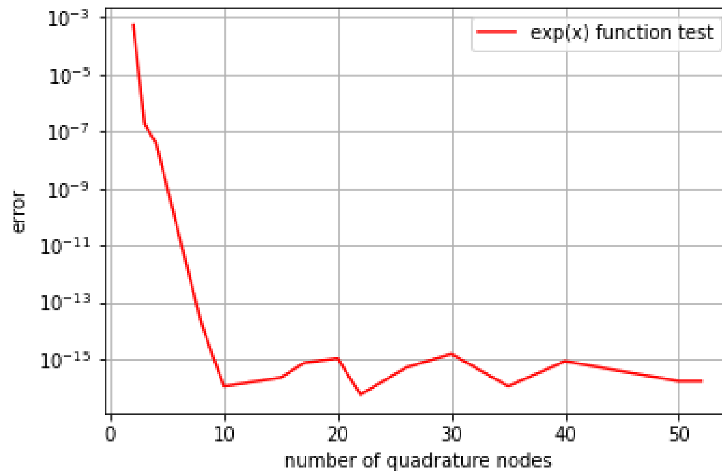


Figure 7.2: $f(x) = e^x$ with $v(x)$

The logarithmic weight function $v(x)$ shows a less stable error convergence, which was expected. As presented by W. Gautschi in [6], the numerical stability of logarithmic singularities is still under research. Acceptable accuracy is reached nevertheless. It must also be taken into account that Figure 7.2 plots errors down to `1e-15` (unlike Figure 7.1), which is around machine precision, thus introducing the possibility that round-off errors are tampering with the outcome of the error-convergence plot. Further plots of other functions and their errors have been deferred to the Appendix (A.1) for lighter reading.

Chapter 8

Conclusions

This paper has presented the `SingGQ` library, from a theoretical aspect to an implementation, accompanied by a runtime / accuracy analysis. The implementations of the Golub-Welsch algorithm exploited the orthogonal properties of the weight function's associated polynomials and the computational advantage of solving self-adjoint eigenvalue problems. In the logarithmic case, we exploited previous computations to avoid redundancies, which is reflected in the OOP structure of the code implementation. The results have been tested against reliable sources and indicate that `SingGQ` is functional and accurate, thus making it a working `C++` library. Despite maximal efforts to reduce computational complexity, the logarithmic weight function still yields quadratic runtime. This is due to the underlying algorithm requiring matrices of linear side-length with respect to the number of quadrature points. It is to be noted that this topic is still currently under research, and leaves many doors open to possible solvers.

Appendix A

Appendix

A.1 Error Plots

A.1.1 Jacobi Weight Function

For $w(x)$, three tests have been done for different values of α, β. Namely, $(\alpha, \beta) \in \{(-0.7, -0.1), (-0.5, -0.5), (-0.8, -0.5)\}$ for $f(x) \in \{e^x, \cos(x), \frac{1}{1+x^2}\}$.

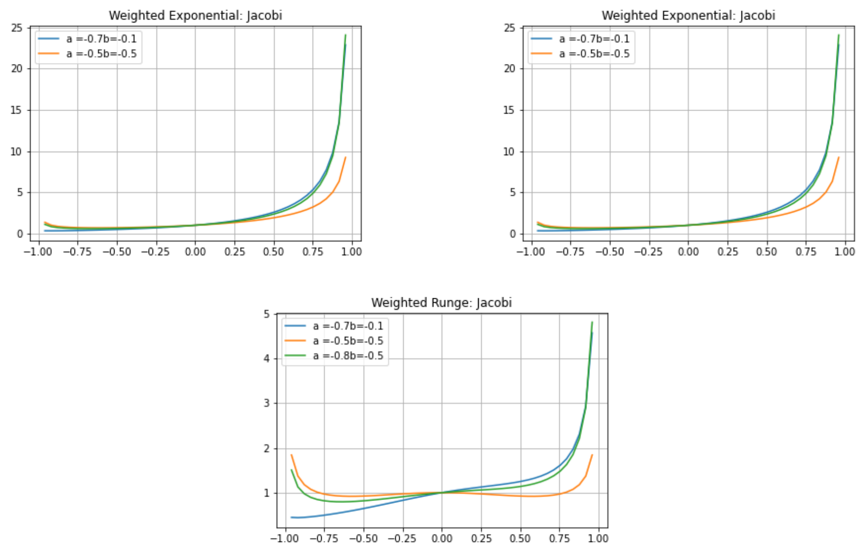


Figure A.1: All different tested functions

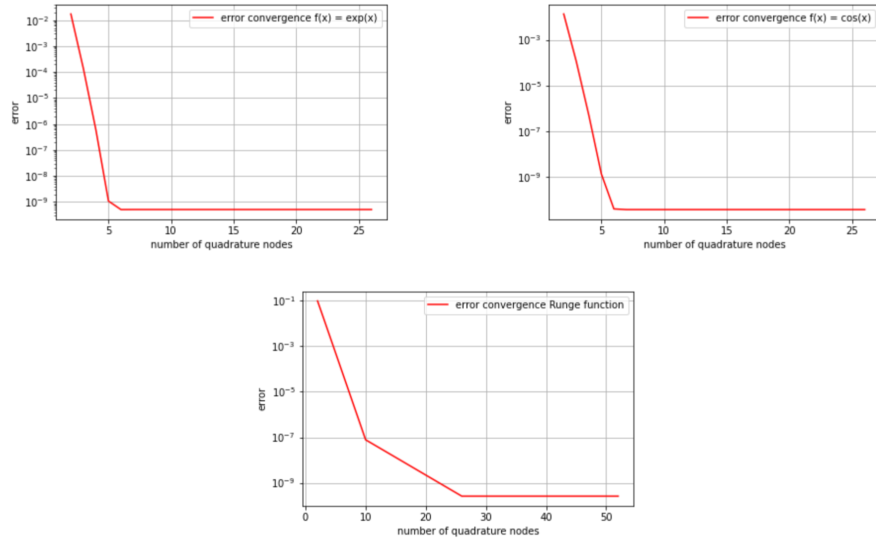


Figure A.2: Error Analysis for $\alpha = -0.7$, $\beta = -0.1$

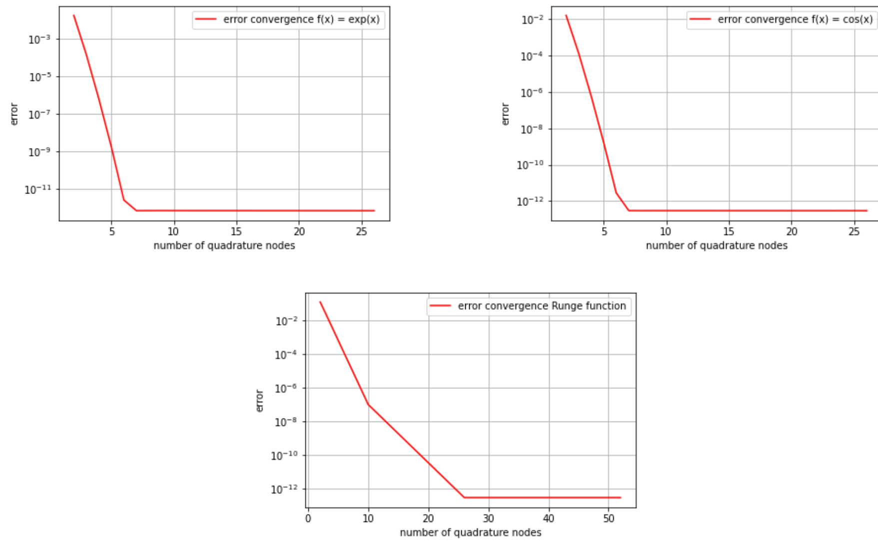


Figure A.3: Error Analysis for $\alpha = -0.5$, $\beta = -0.5$

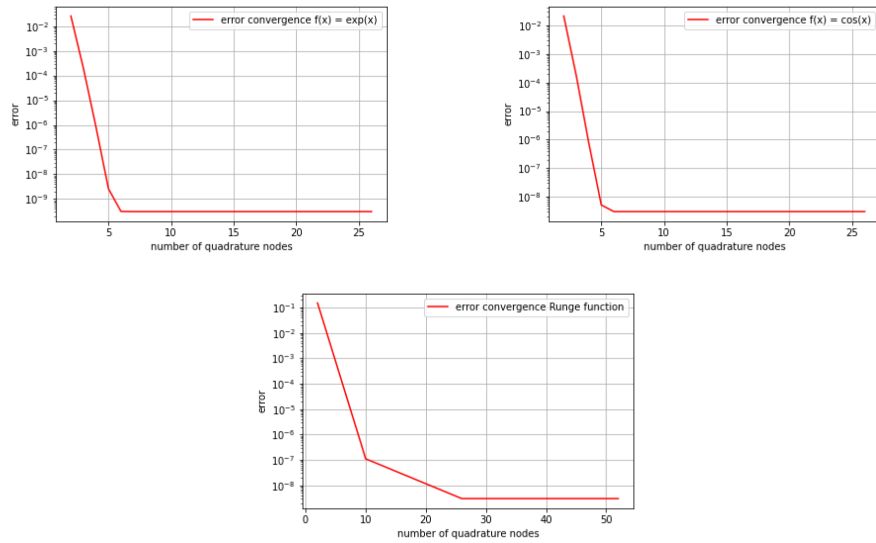


Figure A.4: Error Analysis for $\alpha = -0.8$, $\beta = -0.5$

A.1.2 Logarithmic Weight Function

Only one test per function has been made.

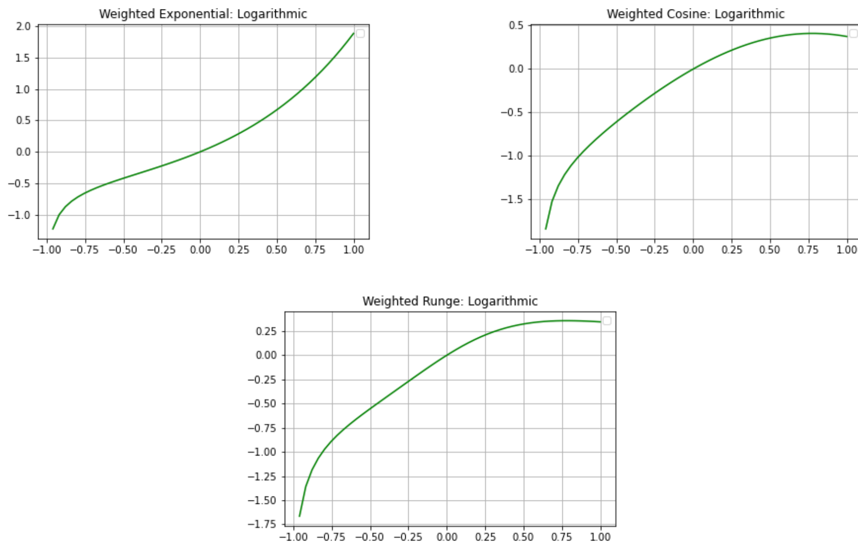


Figure A.5: All tested functions with $v(x)$

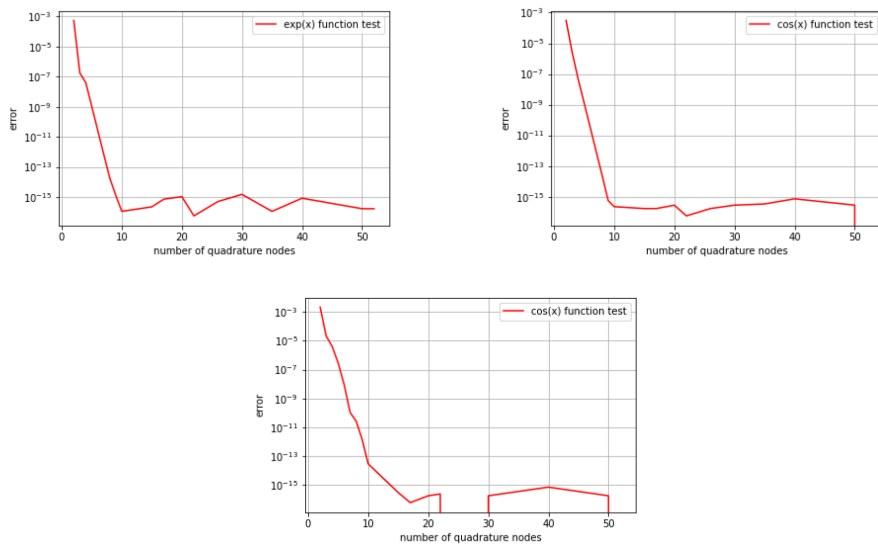


Figure A.6: Error Analysis $v(x)$

A.2 Other Plots

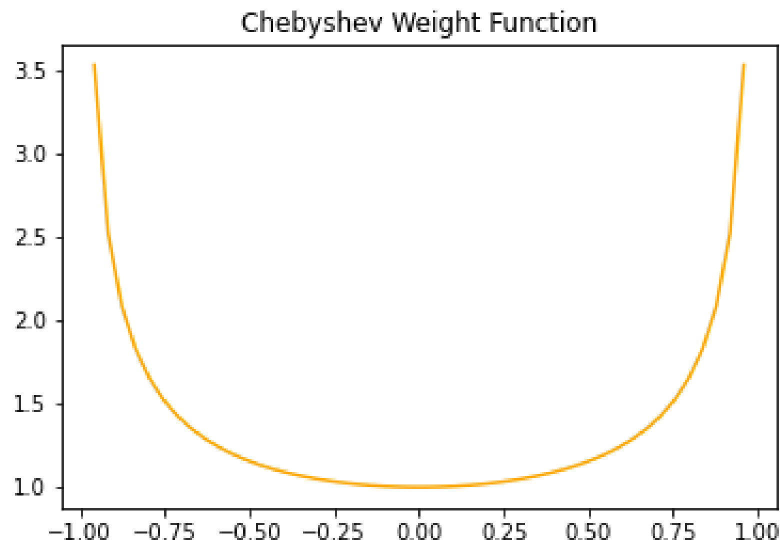


Figure A.7: Chebyshev Weight Function with $\alpha = \beta = -\frac{1}{2}$

Bibliography

[1] Nice M. Temme Amparo Gil, Javier Segura. *Numerical Methods for Special Functions*. Society for Industrial and Applied Mathematics, 2007.

[2] John D. Cook. *Runge Phenomena*. John D. Cook, 2017. URL: https://www.johndcook.com/blog/2017/11/18/runge-phenomena/.

[3] Walter Gautschi. *OPQ: A MATLAB SUITE OF PROGRAMS FOR GENERATING ORTHOGONAL POLYNOMIALS AND RELATED QUADRATURE RULES*. Oxford University Press, 2004. URL: https://www.cs.purdue.edu/archives/2002/wxg/codes/OPQ.html.

[4] Walter Gautschi. *Orthogonal Polynomials: Computation and Approximation*. Oxford University Press, 2004.

[5] Walter Gautschi. *Gauss Quadrature and Christoffel Function for Jacobi weight functions*. Purdue University Research Repository, Jun 2020. URL: https://purr.purdue.edu/publications/3407/1, doi:10.4231/17YY-MC20.

[6] Walter Gautschi. *Gauss Quadrature Routines for two classes of Logarithmic Weight Functions*. Sprenger Science Business + Media, Jan 2010. doi:10.1007/s11075-010-9366-0.

[7] Martin Hanke-Bourgeois. *Grundlage der Numerischen Mathematik und des Wissenschaftlichen Rechnens*. Vieweg & Teubner, 2009.

[8] Roland Bulirsch Josef Stoer. *Introduction to Numerical Analysis*. Springer-Verlag, 1992.